UNIVERSITY OF CAMBRIDGE

A SIMD architecture for hard real-time systems

Roy Spliet

Fitzwilliam

This dissertation is submitted on March 2020 for the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or am concurrently submitting, for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or is being concurrently submitted, for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This dissertation does not exceed the prescribed limit of 60 000 words.

Roy Spliet
March 2020

Abstract

A SIMD architecture for hard real-time systems

Roy Spliet

Emerging safety-critical systems require high-performance data-parallel architectures and, problematically, ones that can guarantee tight and safe worst-case execution times. Given the complexity of existing architectures like GPUs, it is unlikely that sufficiently accurate models and algorithms for timing analysis will emerge in the foreseeable future. This motivates a clean-slate approach to designing a real-time data-parallel architecture.

In this work I present Sim-D: a wide-SIMD architecture for hard real-time systems. Similar to GPUs, Sim-D performs *hardware strip-mining* to schedule the work for a compute kernel in entities called *work-groups*. Sim-D schedules the work for each work-group as a sequence of uninterruptible *access-* and *execute program phases*, interleaving the phases of two work-groups. By providing performance isolation between the memory- and compute resources, the execution time of each phase can be tightly bound through static analysis.

I present a predictable closed-page DRAM controller that processes requests for large 1D- and 2D blocks of data, as well as indirect *indexed* transfers. These large transfers coalesce the data requests of a whole work-group. For a linear 4KiB transfer over a 64-bit data bus, the utilisation provably exceeds 78% for DDR4-3200AA DRAM. For 2D blocks, a well-chosen tiling configuration can achieve near-similar efficiency. I show that bounds on the execution time of indexed transfers are pessimistic by nature, but propose a novel *snoopy indexed transfer* mechanism that permits more reasonable bounds when the buffer size is limited.

Finally, I present a worst-case execution time calculation algorithm for Sim-D. This algorithm is paired with two hardware work-group scheduling policies that deterministically reduce run-time variance. The worst-case execution time analysis algorithm combines static control flow analysis with a simulation-based cost model for execution and DRAM transfers. Its key novelty is the addition of a stage that considers work-group scheduling effects. I show that the work-group scheduling policies degrade performance on average by 8.9%, but permit the calculation of worst-case execution time bounds that are tight within 14.3% on average for benchmarks that avoid inefficient indexed transfers.

Acknowledgements

First and foremost I would like to thank my supervisor, Dr. Robert Mullins, for his continuous support, numerous interesting discussions and pointers to relevant work. I am deeply grateful for the freedom I was given to pursue this project.

Numerous other people in the Department deserve credit for the many on- and off-topic conversations we had throughout the years: my secondary supervisor Prof. Simon Moore, my office mates Alex Bradbury, Sam Ainsworth, Tobias Kohn and Wei Song, my colleagues Daniel Bates, Alex Chadwick, Tim Jones, Theo Markettos and the many others I was privileged to share a corridor or two with.

A special thanks goes out to Dr. Björn Brandenburg for offering me the hands-on introduction to real-time systems research, which provided much inspiration for this work.

I am grateful to my many friends from different colleges and societies who made Cambridge feel like home, the Havana Londres gang for being a solid counterweight on the work-life balance and making London feel like home, my family and long-time friends who remained a beacon of light and made Utrecht feel like home every time I returned, not to forget the many people I've met along the road that have now scattered across the globe. I hope to encounter you again some day.

Finally, I wish to express my gratitude to the late Claire Sparkes who, in the short time I've been privileged to know her, has shared incredible joy and taught me many valuable lessons on empathy, (self-)compassion and looking out for one another on this planet.

Contents

1 Introduction 19
1.1 Contributions 20
1.2 Publication 21

2 Background 23
2.1 Data-parallel architectures 23
2.1.1 Vector processors 23
2.1.2 Packed SIMD extensions 24
2.1.3 Graphics processing units 25
2.1.4 Digital signal processors 26
2.1.5 Domain-specific accelerators 26
2.1.6 Comparison with Sim-D 27
2.2 DRAM and data layout optimisation 28
2.2.1 DDR4 DRAM organisation 28
2.2.2 Data layout optimisation 29
2.2.3 Edges in filter operations 32
2.3 Real-time systems 32
2.3.1 Hard real-time DRAM controllers 33
2.3.1.1 Closed page: Memory pattern-based HRT DRAM controllers 34
2.3.1.2 Open page: Bank privatisation 36
2.3.1.3 Sim-D 36
2.3.2 Hard real-time worst-case execution time analysis 37
2.3.2.1 Control flow analysis 37
2.3.2.2 Processor behaviour analysis 38
2.3.2.3 Bound calculation 39
2.3.2.4 Sim-D 39
2.4 Miscellaneous related work 40

3 A wide-SIMD architecture 43
3.1 Overview 43

3.1.1 Instruction set architecture . . . 45
3.2 Control logic . . . 45
3.2.1 Program launch and hardware strip-mining . . . 46
3.2.2 Assigning global IDs to work-items . . . 46
3.3 Pipeline . . . 49
3.3.1 Fetch . . . 49
3.3.2 Decode . . . 50
3.3.3 Execute . . . 50
3.3.3.1 Compute resources . . . 51
3.3.3.2 Load/store units . . . 52
3.3.4 Write-back . . . 53
3.4 Control flow . . . 53
3.4.1 Vector control flow . . . 53
3.4.1.1 Predicate registers . . . 53
3.4.1.2 Control stack . . . 54
3.4.2 Scalar control flow . . . 54
3.4.3 Usage example . . . 55
3.5 Data path . . . 56
3.5.1 Scratchpads . . . 57
3.5.2 Register Files . . . 58
3.5.2.1 Operand interface . . . 59
3.5.2.2 DRAM- and scratchpad interface . . . 59
3.5.2.3 Vector register file implementation . . . 60
3.5.2.4 n-vector load/stores . . . 63
3.5.3 Mapped buffers: BufToPhysXlat . . . 63
3.5.4 Scoreboard . . . 64
3.5.4.1 Special cases . . . 65
3.6 Summary . . . 68

4 Experimental set-up **69**
4.1 Cycle-accurate simulation . . . 69
4.1.1 DRAM configuration . . . 70
4.2 Real-world comparison and OpenCL data acquisition . . . 72
4.3 Benchmarks . . . 73
4.3.1 CNN . . . 74
4.3.2 KinectFusion . . . 74
4.3.3 SRAD . . . 74
4.3.4 MRI-Q . . . 75
4.3.5 SPMV . . . 76

4.3.6 LU Decomposition . . . 76
4.3.7 Stencil . . . 76
4.3.8 FFT . . . 76
4.4 Limitations . . . 77
4.5 Summary . . . 77

5 A hard real-time DRAM controller **79**
5.1 Limitations of pattern-based hard real-time HRT DRAM controllers . . . 80
5.1.1 Inefficiency on modern DRAM . . . 81
5.1.2 Variation in transfer sizes . . . 82
5.1.3 Alignment constraints . . . 83
5.1.4 Takeaway points . . . 85
5.2 Buffer characterisation and data locality . . . 85
5.3 Architecture . . . 90
5.3.1 Pipeline . . . 91
5.3.1.1 Front-end . . . 92
5.3.1.2 Command generator . . . 97
5.3.1.3 Command arbiter . . . 99
5.3.1.4 DQ scheduler . . . 101
5.3.2 Snoopy indexed transfers . . . 101
5.3.2.1 Data snoop unit . . . 102
5.3.2.2 Stride sequencer . . . 104
5.3.3 Takeaway points . . . 104
5.4 Worst-case request execution time and issue delay . . . 105
5.4.1 Unit-stride . . . 105
5.4.1.1 Four or more bank-groups . . . 110
5.4.2 Non-unit-stride . . . 111
5.4.3 Index iteration . . . 111
5.4.3.1 Four or more bank groups . . . 115
5.4.4 Snoopy indexed transfers . . . 116
5.4.5 Takeaway points . . . 116
5.5 Evaluation . . . 117
5.5.1 Unit-stride transfers . . . 117
5.5.2 Non-unit-stride transfers . . . 120
5.5.3 Indexed transfers . . . 123
5.6 Summary . . . 125

6 Design space exploration **127**
6.1 Resource usage characterisation . . . 128
6.1.1 NVIDIA Kepler . . . 128
6.1.2 Sim-D . . . 134
6.1.3 Takeaway points . . . 137
6.2 Parameter sensitivity . . . 137
6.2.1 Scratchpad bus width . . . 138
6.2.2 Compute configurations . . . 139
6.2.3 Takeaway points . . . 143
6.3 Average case performance comparison . . . 144
6.4 Hardware-supported optimisation . . . 147
6.4.1 n-vector load/stores . . . 147
6.4.2 Warp trimming . . . 150
6.4.3 Takeaway points . . . 152
6.5 Summary . . . 152

7 Worst-case execution time analysis **155**
7.1 Motivation for work-group scheduling constraints . . . 156
7.1.1 DRAM variance and schedule search space . . . 157
7.1.2 Scheduling policies . . . 160
7.2 Program- and execution model . . . 161
7.2.1 Worst-case minimal valid schedule . . . 163
7.2.2 Work-group early exit . . . 166
7.2.3 Complexity and bounds . . . 168
7.2.4 Takeaway points . . . 170
7.3 Hard real-time work-group scheduling policies . . . 170
7.3.1 Launching work-groups in pairs . . . 171
7.3.2 Two-resource scheduling . . . 173
7.3.3 Takeaway points . . . 174
7.4 WCET computation algorithm . . . 174
7.4.1 Parsing the program . . . 175
7.4.2 Control flow analysis . . . 176
7.4.3 Worst-case performance simulation . . . 177
7.4.4 Construct a DAG . . . 179
7.4.5 Critical path analysis . . . 180
7.4.6 Access/execute program phase lists . . . 180
7.4.7 WCET computation . . . 181
7.4.8 DRAM refresh inflation . . . 182
7.5 Evaluation . . . 183

7.5.1 Average-case performance 183
7.5.2 Worst-case execution time 185
7.5.3 Software optimisation techniques 187
7.5.3.1 Loop unrolling 187
7.5.3.2 Instruction scheduling 189
7.5.3.3 2D tiling 190
7.5.4 Takeaway points 191
7.6 Summary 192

8 Conclusion **195**
8.1 Future work 196

Bibliography **199**

A ISA **211**
A.1 Conventions 211
A.2 Register specifications 211
A.3 Floating point arithmetic 212
A.4 Reciprocal/Trigonometry (expensive FP arith) 215
A.5 Integer/Boolean arithmetic 215
A.6 Data copy, conversion and intra-lane shuffle 222
A.7 Load/Store 224
A.8 Control flow 229
A.9 Predicate manipulation 232
A.10 Debug 233

Glossary

address mapping As a noun, refers to the scheme used to translate a physical address to its corresponding DRAM channel, rank, bank-group, bank, row and column. Used as a verb to describe the act of translation.

ALAP as late as possible.

ASOM addition, single-overflow modulo.

BB basic block.

CAM content-addressable memory.

CAS Column Access Strobe, delay between issuing a read command and the first data word appearing on the DQ.

CCD CAS to CAS delay.

CFA control flow analysis.

CFG control-flow graph.

CMASK control mask.

CNN convolutional neural network.

compute unit Hardware unit that performs the computation for some or all work-groups.

CSTACK control stack.

CWD Column Write Delay, delay between issuing a write command and writing the first data word to the DQ.

DAG directed acyclic graph.

device Massively-parallel accelerator, e.g. GPU, FPGA, DSP.

DMA direct memory access.

DQ data bus.

DQM data mask.

DRAM pattern Fixed schedule of DRAM commands, statically scheduled by a DRAM controller to service a request spanning a fixed number of consecutive, aligned bytes of data.

DSA domain specific architecture.

DSP digital signal processor.

FAW four-activate window.

FFT fast-fourier transform.

GPGPU General-Purpose compute on Graphics Processing Unit.

GPR general-purpose register.

GPU graphics processing unit.

GTRR greedy-then-round-robin.

HRT hard real-time.

ILP integer linear programming.

IMem instruction memory.

index iterator Sub-component in the DRAM controller front-end responsible for translating a request in the form of a set of indexes into a buffer into burst requests.

IPET implicit path enumeration.

IR intermediate representation.

ISA instruction set architecture.

JDS jagged diagonal storage.

kernel A non-interactive function designed to run on a device.

kernel-instance An instantiation of a kernel on the device as it is launched by the host system, with it's parameters, buffer objects and compiled kernel..

LID longest issue delay.

local memory A fast addressable storage region private to a work-group. CUDA: shared memory.

LUT look-up table.

NDRange N-Dimensional range, describing the dimensions of the grid of work-items launched for a kernel-instance.

pattern transaction Execution of a single DRAM pattern.

PBA processor-behaviour analysis.

PBGI paired bank-group interleaving.

PC program counter.

PR (vector) predicate register.

RCP-unit reciprocal/trigonometry unit.

remainder work-group work-group for which at least one thread falls outside the kernel-instances' thread dimensions.

RF register file.

RRD row-activate to row-activate delay.

SGPR general purpose scalar register.

SIMD single-instruction multiple-data.

SimdCluster Sim-D's compute unit.

SIMT single-instruction multiple-threads.

SLAM simultaneous localisation and mapping.

SP-unit single-precision unit.

SPMD single program multiple-data.

SSP special purpose scalar register.

stride sequencer Sub-component in the DRAM controller front-end responsible for translating a request in the form of a 2D stride pattern into burst requests.

TID global ID (OpenCL) or thread ID (CUDA), a multi-dimensional identifier within the NDRange unique to a work-item.

VGPR general purpose vector register.

VRF vector register file.

VSP special purpose vector register.

warp trimming Technique for reducing the number of scheduled warps when executing remainder work-groups.

WCET worst-case execution time.

WCRET worst-case request execution time.

work-group Collection of work-items that are guaranteed to run on a single compute unit.

work-item A single thread of a kernel execution.

CHAPTER 1

Introduction

There are many emerging examples of cyber-physical systems that demand both significant compute and hard real-time (HRT) support. In automotive, the shift towards more autonomous vehicles requires running time-critical image processing, AI classification and decision making algorithms on-board [1]. In medical imaging, many algorithms use graphics processing units (GPUs) to achieve lower processing times and higher resolution visualisations [2]. The image processing, neural networks and dense-matrix operations required for these and many other safety-critical applications all exhibit large amounts of data parallelism. To pursue ambitious goals in these fields, researchers and equipment manufacturers are increasingly looking at applying semi-specialised massively parallel accelerators in their devices. Hardware vendors are keen to fill this gap in the market, with e.g. NVIDIA recently releasing their Drive PX1 platform for automotive [3].

An important property distinguishing hard real-time (HRT) systems from regular systems is the stringent deadlines such systems must meet. For HRT systems, the throughput offered by any given hardware platform is only valuable if a *safe* and *tight* bound can be placed on the worst-case execution time (WCET) of the tasks it performs.

It is then unfortunate that, although WCET calculation methods have been researched for GPUs programs [4, 5], none of these methods are able to derive safe bounds for commercially available hardware. The main problem preventing the derivation of safe bounds is the complexity of the hardware. GPUs have over the years developed themselves into a Swiss army knife of graphics processing, containing dedicated subcomponents for compute, video encoding and decoding, data transfers between the host and the device, display scan-out, texture operations, rasterising and possibly many other purposes. When these subcomponents all make use of the GPU's shared memory hierarchy in parallel, latencies on read and write requests are highly unpredictable. This can lead to unexpected timing anomalies [6] that may be difficult to model or reason about in a real-time context.

Even if it is possible to disable all those components that are unused in HRT systems, the GPU's execution model is difficult to reason about. The very fine grain warp-scheduling mechanisms, designed to make the GPU strongly resilient to high DRAM latencies, permit such a wide range of warp interleavings that practical analysis of all possible program executions is infeasible. Without a solid understanding of the worst-case warp interleavings, it seems impossible to claim a WCET bound as safe.

Given the problem of WCET analysis of programs running on massively parallel accelerators, the potential solutions are obvious: either construct more sophisticated models for existing hardware, or create simpler hardware. In the light of the corporate secrecy that exists around current architectures, it is my expectation that the former approach leads to a dead end. Taking the GPU's memory system as an example, is it reasonable to assume that the DRAM controller re-orders requests to maximise throughput. Request re-ordering could theoretically lead to starvation, where a request is continuously placed at the back of the queue in favour of higher priority requests. A DRAM controller can safeguard against starvation by taking a request's age into account when determining its priority. However, there is no public knowledge available for any GPU that unequivocally guarantees starvation-free request prioritisation. Without even such basic guarantees, it seems unwise to assume that safe WCET bounds exist, let alone such bounds being sufficiently tight for practical purposes. A more promising approach is then to design hardware from the ground up with hard real-time requirements and principles in mind.

In this work, I present Sim-D: a wide-SIMD architecture designed for hard real-time systems. Similar to GPUs, Sim-D performs *hardware strip-mining* to schedule the work for a compute kernel in entities called *work-groups*. Inspired by the *PRedictable Execution Model* (PREM) [7], Sim-D schedules the work for each work-group as a sequence of uninterruptible access- and execute *program phases*, interleaving the phases of up to two work-groups at a time. Owing to the strict performance isolation between Sim-D's compute- and storage resources, the execution time of each access- and execute phase can be tightly bound through static analysis. Static WCET derivation of a kernel-instance is then achieved through an analysis of the possible interleavings of these phases. Various scheduling policies are enforced in hardware to reduce the number of possible interleavings. The result is a WCET analysis algorithm tailored to the Sim-D architecture that derives a safe bound tight within 14.3% on average.

1.1 Contributions

My thesis is that **an efficient wide-SIMD accelerator can feasibly be designed that permits the derivation of safe and tight bounds on the execution time of data-parallel programs**. To support this thesis, in this work I contribute the following:

- I introduce the Sim-D architecture, a wide-SIMD processor that is designed to permit

WCET analysis. Sim-D provides *performance isolation* between its compute and data storage resources such that parallel occupation does not introduce pessimism to a program's WCET, (Chapter 3)

- I introduce the experimental set-up used for the evaluation in this work, (Chapter 4)
- I present Sim-D's closed-page DRAM controller, capable of servicing large requests for 1D or 2D regions of data in bound time. Along with its design, I provide the necessary analysis methods to derive worst-case request execution time bounds on each transfer, (Chapter 5)
- I perform a design space exploration of Sim-D, in order to justify its design decisions and to derive sensible parameters for the experiments in this work, (Chapter 6)
- I show that Sim-D is capable of achieving performance on par with an embedded-grade commercial GPU, (Chapter 6)
- I introduce a WCET analysis algorithm that permits efficient derivation of safe bounds on a program's execution time. To support this algorithm, I introduce both a tailored program- and system model, and I introduce two work-group scheduling policies that introduce a 8.9% run-time overhead on average. Ignoring outliers, the resulting WCET bounds are shown to be tight within 14% on average, (Chapter 7)
- I evaluate the impact of several known program optimisations on the WCET of a program running on Sim-D. (Chapter 7)

1.2 Publication

Research carried out as part of this study has resulted in the following publication:

- R. Spliet and R. Mullins. *The case for limited-preemptive scheduling in GPUs for real-time systems.* In ECRTS, Operating Systems Platforms for Embedded Real-Time applications, Jul 2018. [6]

CHAPTER 2

Background

In this chapter I set out to explain the necessary terminology and relevant work that led up to the design and evaluation of Sim-D.

Specifically I contribute the following:

- An overview of the classes identified in the landscape of data-parallel architectures and the position Sim-D occupies in this space (Section 2.1),
- A short introduction to DDR4 DRAM and data layout optimisation techniques relevant to GPUs (Section 2.2),
- An explanation of related work in hard real-time DRAM controllers, worst-case timing analysis techniques and other related work that inspired the design of Sim-D (Section 2.3).

2.1 Data-parallel architectures

This section summarises several classes of data-parallel architectures, with the goal of explaining Sim-D's position in the landscape. The presented taxonomy is largely drawn from Hennessy and Patterson [8].

2.1.1 Vector processors

In broad terms, vector processors are processors that support instructions to perform arithmetic operations on every element of an array. A defining feature of a vector processor is that the length of this array is run-time configurable, anywhere between 1 and the maximum number of elements that can be stored by their SRAM- or register-backed array storage. Such vector processor designs go back to the 1960s with the presentation of the

ILLIAC IV [9]. More recent examples include the HWACHA architecture [10] and ARM's Scalar Vector Extension (SVE) [11].

Three complementary techniques are used to process vectors of independent elements in a single-instruction multiple-data (SIMD) fashion. Firstly, *pipelining* permits a throughput of one vector element per cycle. Control logic ensures that for a given vector instruction, the operation is issued as many times as required to process the every element in the vector. Secondly, *vector chaining* [12] permits multiple vector instructions to overlap provided there are no hazards. For example, if a vector load operation is followed by a vector addition of said vector with a different element, the vector addition can start processing the first element of the vector as soon as it arrives, rather than waiting for the entire load operation to finish. Chaining m instructions could therefore result in an IPC approaching m. Finally, duplication of arithmetic units permits performing a vector operation for multiple elements in parallel in the same cycle. A parallel processor capable of performing n arithmetic operations in parallel are said to have n (vector-)*lanes*.

Transforming a loop of scalar operations into a loop of vector operations is called *strip mining* [13]. Given a vector processor with a maximum array length l_{max}, strip-mining is performed explicitly in assembly by replacing any scalar loop iterating over n elements with a loop that performs $\left\lceil \frac{n}{l_{max}} \right\rceil$ iterations, each iteration issuing the vector arithmetic for up to n elements.

2.1.2 Packed SIMD extensions

To speed up applications that perform digital signal processing or graphics operations on application processors, several instruction set architectures (ISAs) have been extended with packed SIMD operations. These operations borrow from the parallel-processing concepts of vector processors. Early extensions were designed to re-use existing resources from high-end application processors. For example, Intel's MMX-extension [14] does not introduce specialised vector storage, but *packs* vectors of 2×32-bit, 4×16-bit or 8×8-bit elements into existing x87 floating point registers.

Over time, packed SIMD extensions have converged with vector processor architectures. This is most evident from the storage reserved for vector elements: where for MMX the size of a vector was limited to 64-bits [14], the current-generation AVX-512 [15] extension offers dedicated vector registers of 512 bits each. For comparison, ARM SVE [11] can offer up to 2048 bits of storage per vector.

Today, the main distinguishing property of packed SIMD extensions is that the number of elements in a vector is fixed to the width of the register rather than configurable at run-time. Packed SIMD extensions may define instruction variants for different register widths. If arrays are shorter or elements must be skipped, a predicate mask can inform instructions which elements of a vector register must be processed. Predicate masks have no influence on performance, rather they determine which results must be written back to the destination vector register and which should be discarded.

Packed SIMD operations are part of the application processor's ISA. As such, their execution does not incur overheads paid for uploading the kernel code and data-set to an external accelerator, like it would be the case with GPUs or digital signal processors (DSPs). However, their throughput is generally only a fraction of the throughput achieved even on a embedded GPU.

2.1.3 Graphics processing units

Since the advent of 3D gaming, GPUs have gradually developed into a distinct class of massively-parallel processors. After several research projects aimed at utilising the GPUs resources for non-graphics applications (e.g. BrookGPU [16]), NVIDIA's CUDA was the first commercially available architecture to offer general purpose usage of their GPU [17]. Like vector processors and packed SIMD extensions, GPUs apply pipelining and lock-step parallel execution of operations on vector elements to reduce control overhead, resulting in energy-efficient high-throughput processing of data-parallel kernels. However, their *hardware strip-mining* single-instruction multiple-threads (SIMT) programming model make GPUs unique in their capabilities to effectively occupy several thousands arithmetic units and other resources in parallel.

GPUs rigorously part with the concept of processing vector elements in order. Instead, conceptually a GPU work scheduler breaks vectors into *warps*, groups of 32 (NVIDIA) or 64 (AMD) elements. Each warp is processed in a separate hardware thread, with the operations for the warp performed in SIMD. Each *compute unit* in a GPU can issue multiple instructions per cycle from a plurality of warps. The large pool of active warps helps to mask data movement latencies between the memory hierarchy and the individual warps, maximising occupancy of the available compute- and memory resources.

To illustrate the scale of a compute unit, each “SMX” in an NVIDIA Kepler GPU [18] contains 192 single-precision units (SP-units) for integer and floating-point arithmetic, 32 load/store units, 32 special function units (used for e.g. trigonometry operations) and various other units required mostly for graphics operations. A high-end GeForce 780Ti GPU contains 15 compute units [19]. At any time, a compute unit can have 64 active warps divided over four warp-schedulers. Each warp-scheduler can issue one or two instructions per cycle.

Warp scheduling requires the GPUs to perform strip-mining in hardware. To this end, an application that wishes to off-load computation to the GPU launches a *kernel-instance* parametrised with its desired vector size. This size, called the *NDRange*, is specified in up to 3 dimensions, and dictates how many work-items must be launched. Work-items are identified by a global ID (TID), a unique identifier in the NDRange space. Work-items are grouped into *warps*, in turn grouped into *work-groups*. All warps in a work-group execute on a single compute unit, allowing a work-group to share data between its work-items.

Contrary to vector processors and packed SIMD extensions, developers treat the GPU as an external device. Relevant vectors and other parameters must be uploaded to dedicated DRAM local to the GPU. Kernels, non-interactive functions running on the GPU, are developed according to the single program multiple-data (SPMD) paradigm: code specifies the work that needs to be performed for a single work-item, relying on hardware strip-mining to run this code for every work-item.

2.1.4 Digital signal processors

DSPs are frequently found in embedded systems that perform filtering, transformation and error-correction of digital signals. Although their architectures vary, most are best categorised as VLIW processors with data-parallel processing capabilities, using narrow-word SIMD techniques [20–22] for increased throughput of arithmetic operations like multiply-accumulate.

DSPs are used to fulfil HRT tasks. For example, Qualcomm's baseband modem is paired with two dedicated DSPs to perform latency-sensitive audio-processing tasks [20]. To this end, pipelines are kept simple, multi-threading is limited to a few hardware threads and input data may be processed through dedicated channels rather than over shared buses. Software running on DSPs is generally persistent by nature, rather than acting as a CPU-controlled accelerator device.

Exceptionally, the Qualcomm Hexagon DSP permits user-space applications running on their mobile SoCs to upload custom kernels for execution. This brings the DSP's role and design closer to that of a GPU.

2.1.5 Domain-specific accelerators

Domain specific architectures (DSAs) are architectures that are highly optimised for a single task, trading general applicability for performance and power efficiency. Such architectures play a prominent role in the domains of video decoding (e.g. NVIDIA Falcon [23]) and machine learning (e.g. TPU [24], NVDLA [25] and GraphCore [26]). Besides their application in specialised high performance computing, DSAs are often used in power-constrained mobile systems.

DSAs can be tailored to their task in various ways. Firstly, compute resources are designed to closely match the precision requirements of the task. Secondly, the ISA of a specialised accelerator usually contains application-specific vector instructions. Such instructions may have non-standard result modifiers or even follow VLIW encoding practices to maximise code density. These modifiers or fused instructions are matched with a non-standard pipeline to perform the task at hand as quickly as possible. Finally, memory hierarchies are tuned to the data requirements of the task, for example by replacing associative caches with scratchpads or introducing single-purpose buffers.

2.1.6 Comparison with Sim-D

Sim-D is designed as a general-purpose wide-SIMD accelerator tailored to HRT systems. The targeted applications are inspired by recent trends in autonomous safety-critical systems, such as assisted driving. Domains include computer vision, artificial intelligence but also digital signal processing. Latencies and deadlines of such applications are often expressed in (tens of) milliseconds [1, 27], one or two orders of magnitude larger than that of most DSP applications and of control tasks like e.g. software-controlled fuel injection in car engines. In the light of such latencies, fixed overheads for communication and task scheduling have a relatively small impact on the overall execution time of a task. This combination of large data-parallel workloads and relatively low cost of fixed scheduling overheads resulted in Sim-D's design to resemble existing GPUs. Besides wide-SIMD execution, this is reflected in the way Sim-D performs hardware strip-mining and permits parallel scheduling of multiple hardware threads processing the same vector.

Compared to a contemporary GPU, Sim-D's unit of scheduling is more coarse-grain: where GPUs operate on warps, Sim-D's compute unit schedules operations at a work-group granularity, each work-group containing 1024 work-items. Work-items within a work-group are processed like in a regular multi-lane pipelined vector processor, without the use of chaining techniques. Instead, parallel occupation of resources is increased by interleaving the execution of two work-groups at any point in time. While such coarser-grain scheduling reduces the ability for parallel occupation of resources, it increases the predictability of instruction scheduling and data movements. This predictable execution allows Sim-D's to provide WCET bounds on the run-time of kernel-instances, as explained in Chapter 7.

In terms of resources, Sim-D currently positions itself as an embedded-grade accelerator. It features a 64-bit DRAM bus and a single compute unit. Scaling this design up towards the DRAM bus and compute resources of a discrete GPU comes with specific challenges to the memory controller and the WCET analysis algorithm. I will outline these challenges throughout the dissertation, but leave their resolution for future work.

Despite its resemblance to a GPU, Sim-D does not aim to implement a graphics pipeline. Although it is conceivable that the compute resources can be used for some graphics-related computation, it lacks features specific to 3D-rendering such as texture sampling or rasterising.

Sim-D's ISA more closely resembles that of a vector processor, explicitly mixing vector- and scalar instructions in an otherwise RISC-like ISA. That being said, Sim-D is not an application processor. Its ISA borrows heavily from NVIDIA's GPU ISAs, and omits the hardware I/O and timing features required to run an operating system. Unique to Sim-D is the ability to issue large DRAM requests for 1D- and 2D blocks of data for an entire work-group, which is then processed by the DRAM controller in parallel with compute for another work-group.

2.2 DRAM and data layout optimisation

In this section, I present an abstraction of DRAM as relevant to the understanding of this thesis. This explanation is followed by an explanation of common data layout optimisation considerations for GPUs. For a more complete overview of the internals and operation of DRAM, I refer the reader to "Memory Systems : Cache, DRAM, Disk" [28].

2.2.1 DDR4 DRAM organisation

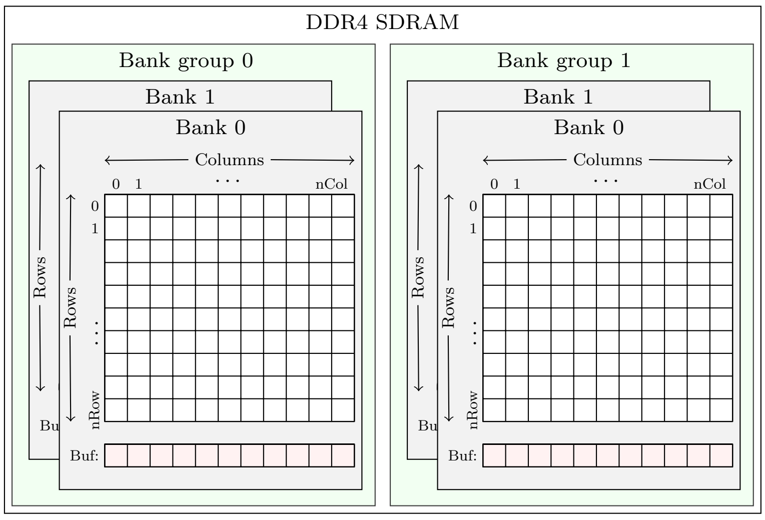


Figure 2.1: Schematic abstraction of a DDR4 DRAM chip

A DDR4 DRAM chip consists of multiple *bank groups*, each containing an equal number of *banks*. Each bank contains a 2D-grid of memories, organised as *rows* and *columns*. Each bank additionally contains a *row buffer*. A DRAM chip's interface consists of a clock input, command- and address signals, a chip select input, a data mask (DQM) input and a bidirectional data bus (DQ).

The *DQ width* of a memory chip specifies the number of data lines, and thus the number of bits it transfers in a single cycle. A *rank* is generally formed of multiple DRAM chips in parallel. For example, a 64-bit wide rank can consist of 8 chips with a DQ width of 8, or 4 chips with a DQ width of 16. A *channel* consists of a set of data- and command lines that are shared between one or more ranks. Each channel has a one-hot bit mask to select the destination rank for each command. In this work, I only consider a channel containing a single rank.

Data is requested in *bursts*. For DDR4 DRAM, a burst consists of 8 beats. In other words, each request issues a read or write to 8 consecutive columns in a row. Write operations can be performed at a byte granularity by clearing each byte's corresponding bit in the DQM for each beat of a burst. In this work all latencies are measured in clock cycles of the command bus. Being *double data rate* DRAM, a burst is thus said to complete its transfer in 4 cycles. Transfers must be aligned to a multiple of a burst.

Operating DRAM is done using three operations: activation, read/write and precharge. Activation brings the data from a row into the row buffer of the respective bank. Read and write operations transfer data between the row buffer and the requestor. Finally, precharge conceptually writes back data to the DRAM cells and prepares the row-buffer for the next activation of that bank.

Physical properties of the DRAM chip place constraints both on the latencies of each operations and on the distance that must be guaranteed between two operations. A chip's set of constraints is referred to as its *DRAM timings*. Although banks are conceptually parallel independent entities within a chip, constraints exist on the minimal distance between e.g. two activation operations to different banks. In DDR4, the minimal distance between two consecutive operations issued to different bank groups is shorter than the same operations issued to banks in the same bank group. A full overview of all dimensions and timings for the Micron MT40A512M16JY-062E and MT40A1G8SA-062E DDR4-3200 DRAM chips is given in Section 4.1.1.

2.2.2 Data layout optimisation

It is generally accepted that the performance of most GPU compute (GPGPU) kernels is bound by DRAM throughput. In turn, optimisation guides (e.g. AMD [29], Intel [30], NVIDIA [31]) discuss programming techniques that improve memory throughput of kernel-instances. Three themes emerge: data alignment with respect to DRAM or cache lines, coalesced accesses of work-items in a warp and reducing DRAM accesses by using local memory to share data between work-items in a work-group. From the architect's perspective, these techniques are interesting as they both determine the requirements applications impose on a hardware design as well as set out the scope for optimisations of common cases within the memory subsystem.

GPUs have the ability to coalesce memory requests of work-items when they access data elements from the same cache line or DRAM burst. A straightforward way of maximising the potential of request coalescing is to structure data such that there is a linear mapping from a work-item's TID and its requested data element from a buffer.

For this reason, structuring data as a *struct-of-arrays* is preferred over an *array-of-*

structs. As a concrete example, consider a buffer for which each data element consists of four components: x, y, z and w. Figure 2.2 shows how this data is laid out in memory under both an array-of-structs and an struct-of-arrays arrangement for 16 work-items. Each row aligns to a DRAM burst.

0,x	0,y	0,z	0,w	1,x	1,y	1,z	1,w	2,x	2,y	2,z	2,w	3,x	3,y	3,z	3,w
4,x	4,y	4,z	4,w	5,x	5,y	5,z	5,w	6,x	6,y	6,z	6,w	7,x	7,y	7,z	7,w
8,x	8,y	8,z	8,w	9,x	9,y	9,z	9,w	10,x	10,y	10,z	10,w	11,x	11,y	11,z	11,w
12,x	12,y	12,z	12,w	13,x	13,y	13,z	13,w	14,x	14,y	14,z	14,w	15,x	15,y	15,z	15,w

a) Array-of-structs

	0,x	1,x	2,x	3,x	4,x	5,x	6,x	7,x	8,x	9,x	10,x	11,x	12,x	13,x	14,x
15,x	0,y	1,y	2,y	3,y	4,y	5,y	6,y	7,y	8,y	9,y	10,y	11,y	12,y	13,y	14,y
15,y	0,z	1,z	2,z	3,z	4,z	5,z	6,z	7,z	8,z	9,z	10,z	11,z	12,z	13,z	14,z
15,z	0,w	1,w	2,w	3,w	4,w	5,w	6,w	7,w	8,w	9,w	10,w	11,w	12,w	13,w	14,w
15,w															

b) Struct-of-arrays, misaligned

0,x	1,x	2,x	3,x	4,x	5,x	6,x	7,x	8,x	9,x	10,x	11,x	12,x	13,x	14,x	15,x
0,y	1,y	2,y	3,y	4,y	5,y	6,y	7,y	8,y	9,y	10,y	11,y	12,y	13,y	14,y	15,y
0,z	1,z	2,z	3,z	4,z	5,z	6,z	7,z	8,z	9,z	10,z	11,z	12,z	13,z	14,z	15,z
0,w	1,w	2,w	3,w	4,w	5,w	6,w	7,w	8,w	9,w	10,w	11,w	12,w	13,w	14,w	15,w

c) Struct-of-arrays, aligned

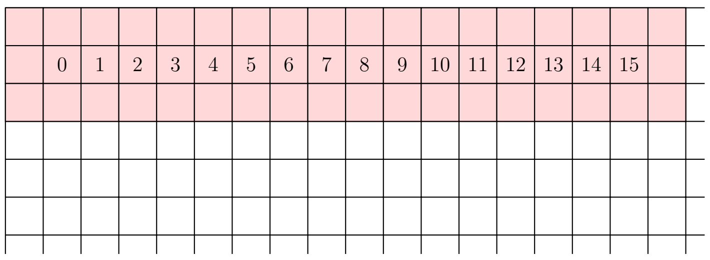
Figure 2.2: Data lay-out examples for 4-element entries

As the red elements demonstrate, a request for the x-components of each work-item results in loading four bursts when the data is structured as an array-of-structs, while a well aligned struct-of-arrays buffer can provide the same data in a single burst.

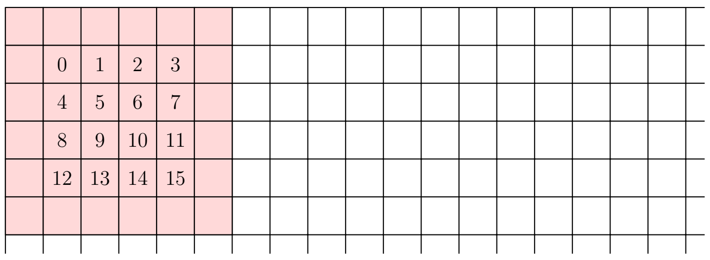
Figure 2.2b demonstrates the importance of data alignment. Misaligned data necessitates the request of a second burst or cache line, resulting in more pressure on the memory system. If the nature of the data permits aligning the elements a warp requests to the boundaries of a burst, this pressure can be reduced. However, as the class of filter algorithms demonstrates, this is not always within control of the programmer.

Filter kernels read a $n \times m$-region of data for each work-item, performing a weighted multiply-addition on each element in this region to compute a final output. Regions read by adjacent work-items may overlap. Examples of such filter kernels include image anti-aliasing and the max-pooling operation for convolutional neural networks (CNNs).

A common technique to maximise the number of words shared within a work-group is called *tiling*. To demonstrate this effect, consider a 3×3 filter processed by a work-group containing 16 work-items. Figure 2.3 shows the tile of data accessed by one work-group for both a 16×1- and a 4×4 tiling configuration. The centre of each work-item's 3×3 data region is marked with its TID.



a) 16 × 1 tiling



a) 4 × 4 tiling

Figure 2.3: Data regions processed by one work-group for a 3 × 3 filter operation, stride 1

As Figure 2.3 demonstrates, an efficient tiling strategy can reduce the number of elements requested by a work-group. Where for the 16 × 1 configuration $18 * 3 = 54$ elements must be read from memory, the 4 × 4 configuration reads only $6 * 6 = 36$ elements. It is not immediately clear which tiling strategy gives the best DRAM throughput, as alignment of data influences the number of bursts issued for the request of a given tile. Tile dimensions must thus be chosen on a per-case basis.

This technique assumes an effective mechanism to share data elements within a work-group, such that tiles are only requested once from DRAM. GPUs support both transparent caching of tile data in associative caches, and explicit tile caching in local memory.

Even when data is not shared between work-items in a work-group, it could pay off to load data into local memory. Consider for example a 2 × 2 filter operation with a pitch of 2. The arrows in Figure 2.4 show how elements from the input buffer map to the first two work-items. Each work-item requests its elements in the order *red, green, blue, yellow.*

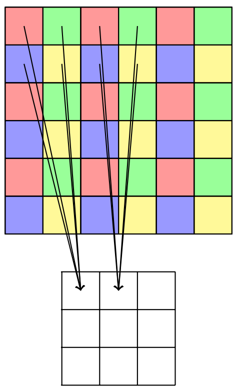


Figure 2.4: Example 2 × 2 filter operation with a pitch of 2.

Looking at the coalesced request for all red items, we can see that the pitch between two work-items causes every other (green) data element in a burst to be discarded even though these are required later. Performance of the application can be improved by pre-loading the work-group's entire block into local memory. This way the cost of issuing the second request for the green data words can be serviced by the lower-latency local storage, reducing the occupation of the shared DRAM bus.

2.2.3 Edges in filter operations

Tiling techniques for filter operations must consider the way an application handles the borders of a data set. Figure 2.5 demonstrates three edge-case strategies: trimmed edges, constant edges and extrapolated edges.

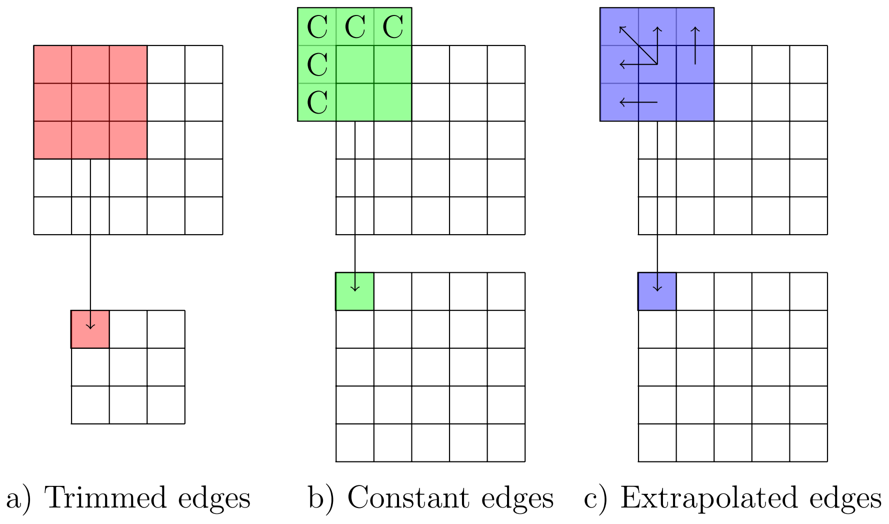


Figure 2.5: Variations of 3*3 filter processing.

In all these cases, the most convenient TID mapping is a linear mapping of work-items to output elements. For the trimmed edges case this mapping ensures that no boundary cases exist for transferring tiles of data from DRAM to the compute cores' registers. The required tile of data for a work-group is simply of size $xdim + 2, ydim + 2$.

Requests that load a tile of data to local memory must compensate offsets and dimensions for constant- and extrapolated edges. If edges are simply ignored, transfers can end up reading beyond the bounds of the buffer. This poses a risk to program correctness and could violate process isolation principles.

2.3 Real-time systems

Real-time systems are a specific class of systems that can guarantee real time constraints, expressed as deadlines on work, that are imposed by the physical world. Consider for example the response time of car brakes: one deadline could express the maximum tolerable delay between the *event* of operating the brake pedal and the *response* of an engaged brake. An appropriate deadline is chosen by car manufacturers to ensure that the braking distance stays within safe limits.

To reason about a system's ability to meet such deadlines, a large body of research analyses the problem of *schedulability*. Using the abstraction of a task model [32–34], algorithms can decide for a given set of tasks and a given scheduling policy (e.g. fixed-priority (FP) [35], earliest-deadline first (EDF) [35, 36]) whether all deadlines are met. In these task models, a task is described by a *deadline*, a *WCET* (*cost*), the (minimum) interval between two events of the same task (*period*), and for some scheduling policies the *priority* of a task relative to others. To determine whether a task is guaranteed to meet it's deadline, its *worst-case response time (WCRT)* must be calculated. The WCRT is the maximum time it takes for a job to complete from the moment of launch, taking into account the worst-case blocking that can be caused by other active tasks. The WCRT is computed either explicitly or implicitly as part of a schedulability test for a given task set.

This dissertation contributes a wide-SIMD architecture and algorithm that together permit the derivation of a safe WCET for a given kernel-instance. In the context of scheduling, the WCET of either a single kernel-instance or a sequence of kernel-instances may form the cost of a task. Policies for scheduling tasks or kernels on Sim-D and schedulability tests to assess whether tasks meet their deadlines are considered beyond the scope of this work. Section 8.1 discusses various avenues for future work on task scheduling with Sim-D.

In the remainder of this section I discuss related work on two topics from the field of real-time systems relevant to this thesis: real-time DRAM controllers and WCET computation algorithms.

2.3.1 Hard real-time DRAM controllers

DRAM controllers can be divided in two groups: closed-page and open-page. Closed-page DRAM controllers ensure that every request starts and finishes with all banks precharged. Open-page DRAM controllers leave rows open after servicing a request, anticipating that more columns from that row will be requested later. The latter strategy is mostly employed in throughput-oriented systems, as the overhead for precharging and activating rows are substantial.

From a real-time perspective, a closed-page policy has the favourable property of performance isolation: a requests' execution time does not vary based on those that precede it. This paves the way for worst-case execution analysis of individual request. At the same time, within each request it is possible to exploit bank locality and make use of parallel execution of activate and read/write commands to different banks. In other words: the deterministic closed-page property exists on the *boundaries* of each request, while *within* a request open-page performance optimisation techniques can be employed.

To evaluate the performance of DRAM controllers, I use two definitions defined by Paolieri et al [37], to describe the (worst-case) timing behaviour of DRAM requests: worst-case request execution time and longest issue delay.

The worst-case request execution time (WCRET) marks the maximum delay, excluding the time waiting for requests to wait for other requests to finish, before a work-group can continue execution. For read operations, the WCRET spans the interval between issuing the first bank activate command of a request and the arrival of the last data word. For write operations, this interval spans from the first bank activate command to the instance the last written data word is transferred on the DRAM data bus, either originating from Sim-D's register file or scratchpad.

The issue delay for a request is defined as the time required between starting the current request and starting the next. For a closed-page DRAM controller, this spans the interval from the first activate command to the first time instant at which all banks are precharged. Bar the refresh interval counter(s), the state of the DRAM controller is indistinguishable before a request and after passing the issue delay, a key property for allowing the analysis of requests in isolation from others. Longest issue delay (LID) is defined as the upper bound on a request's issue delay.

2.3.1.1 Closed page: Memory pattern-based HRT DRAM controllers

A large body of research has focussed on *memory pattern*-based DRAM controllers [38–42]. In this context, a memory pattern is a predetermined schedule of DRAM commands that services a read or write request of fixed size to an arbitrary (but aligned) address. A request may span more than one burst of data, and may address more than a single row. Patterns have fixed timing properties and implement a closed-page policy at the boundaries, striking a balance between the analysability of statically scheduled DRAM controllers on one hand, and performance and flexibility of a dynamically scheduled DRAM controller on the other.

In the literature, the assumption is made that a request always reads or writes as many words as are serviced by one execution of a DRAM pattern. In the light of CPUs, issuing aligned cache-line sized requests, this assumption is valid, but in the context of our Sim-D architecture this assumption does not acknowledge the fact that coalesced requests can be of arbitrary size. Hence in this work I define a *request* to be of arbitrary size. An execution of the commands in a DRAM pattern will be referred to as a *pattern transaction*. A request could therefore require multiple pattern transactions to be performed by a pattern-based DRAM controller.

Akesson et al. [38] describe the basic architecture of a pattern-based DRAM controller and show how its design leads to both performance-predictability and -composability. These properties imply that the performance of individual consumers can be analysed independently from one another, following the Latency-Rate model [43].

The first pattern-based DRAM controller was Akesson et al.'s "Predator" [40]. This DDR2 memory controller guarantees bandwidth and latency of requesters by pre-computing the latency for the execution of each cache-line sized DRAM patterns (Akesson: "memory groups"). The memory configuration is carefully chosen such that each pattern transaction hits all banks, helping to reduce the distance between two requests. By the time request R_n processes the data transfer of its last bank $m+3$, the first bank will already be precharged. Request R_{n+1} is now ready to send its first request to bank m despite bank $m+3$ not being precharged yet. By using a novel arbitration policy called "credit-controlled static priority" (CCSP) [44], Predator is capable of providing both minimum bandwidth guarantees as well as a bound latency on individual requests.

Paolieri et al. [45] propose a similar solution called "AMC". Rather than providing latency-rate guarantees with a CCSP-based arbiter, AMC utilises a simpler round-robin policy. AMC provisions for mixed-criticality workloads by always prioritising HRT requests over non-real-time requests. This leads to a simple static latency analysis where every request is assumed to be delayed by at most the number of other HRT tasks in the system (plus one non-HRT task) multiplied by the maximum latency of any cache-line sized request. Like Predator, AMC makes each request iterate over all banks in the DRAM chip for efficient operation. In [37], they discuss an extension where non-HRT requests can be preempted between two bank-requests, thus breaking with the concept of issuing fixed DRAM patterns. Assuming their timing analysis model, wherein every request is treated independent rather than analysed as part of a global "sequential history", this preemption support reduces the analysed worst-case latency for HRT requests.

Goossens et al. [41] identify how reconfigurability could bring additional guaranteed bandwidth to pattern-based HRT memory controllers. They extend Akesson et al.'s model [38] in two ways. First they adjust the DRAM patterns such that read and write operations always take the same time, regardless of the interleaving of the two. The memory controller services clients following a TDM schedule, providing composable performance to the clients. Secondly, they add a mechanism for reconfiguring both the client TDM slot allocations and the memory patterns. In HRT systems following e.g. partitioned sporadic task scheduling, they can alter the client bandwidth based on the active tasks, thus cater for a wider range of task-sets.

Li et al. [42] identify an increasing need for a variable DRAM request size. They observe that allowing arbitrary sized requests leads to timing anomalies caused by the history of transactions executed. They propose a round-robin scheduling policy of requests of variable size using a closed-page DRAM policy. For individual transactions, they propose a method to derive a tight bound on worst-case execution time. In subsequent work, they propose WCET analysis methods for DRAM requests based on data-flow programming [46] and timed automata [47], both resulting in tighter bounds.

All previous methods rely on efficient DRAM patterns to be determined. To complement the various ad-hoc algorithms for generating memory patterns, Goossens et al. [48] propose an ILP-based method for finding optimal DRAM patterns.

Study towards the performance characteristics of memory pattern-based DRAM controllers uncovered several draw-backs. Akesson et al. [39] observe that to achieve a net memory efficiency above 80% with DDR3 memory, they require transfers to be up to four times larger than with DDR2 technology. They conclude that to achieve high worst-case bandwidth, large requests are fundamentally required.

Krishnapillai et al. [49] observe that these techniques lose efficiency further on modern SoCs as the data bus width has increased. This results in situations where an entire cache line can be fetched in a single burst, invalidating earlier assumptions that requests can effectively be interleaved over multiple banks [37, 40, 45].

2.3.1.2 Open page: Bank privatisation

Bank privatisation has first been proposed for multi-requester systems by Reineke et al. [50]. The idea is that by giving individual requesters access to their own banks only, cross-task bank collisions are eradicated. The result is that tasks can now predict the state of the relevant row-buffers regardless of other tasks running in parallel, making it feasible to adopt an open-row policy while maintaining performance isolation.

Farshchi et al. [51] revisit this idea by pairing bank privatisation with buffer-specific cache write-back policies, a feature widely available in most processors implementing the ARMv7 or ARMv8 instruction set [52]. By distinguishing private- from shared memory and changing policies accordingly, tighter latency bounds can be given for all request to private memory without sacrificing functionality.

2.3.1.3 Sim-D

Inspired by Akesson [38] and Krishnapillai's [49] observations, Sim-D explores new techniques to exploit the bandwidth opportunities offered by processing larger DRAM requests. To this end, this work presents an architecture and DRAM controller that lets developers issue DRAM requests which explicitly coalesce the loads or stores for all work-items in a work-group.

Sim-D applies a closed-page policy between requests, but parts with the pattern-based DRAM command scheduling researched in prior work. Pattern-based DRAM controllers have limitations that make them unsuitable for Sim-D, as analysed in detail in Section 5.1.

Instead, Sim-D uses a deterministic greedy DRAM command scheduler that permits static analysis of DRAM request execution times.

Meanwhile, I consider bank privatisation techniques a bad match for Sim-D as they rely on the presence of multiple requestors in a system. For Sim-D, a requestor can either be defined as a kernel-instance or as a work-group. Neither of these definitions are helpful. In the former case, at any point in time only a single requestor is active, implying that no interleaving of requests from different requestors may be assumed. In the latter case, partitioning of DRAM is impossible as it would prevent work-groups from sharing input data with neighbouring work-group as is required by e.g. filter kernels.

2.3.2 Hard real-time worst-case execution time analysis

HRT WCET analysis techniques are broadly categorised into two groups: static methods and measurement-based methods [53]. Measurement-based methods execute (parts of) a real-time task either on their targeted hardware or on simulators derived from that hardware, and measure meaningful timing information from them. Static methods instead analyse the code of a program to derive a bound on execution time.

Measurement-based methods provide meaningful insight in the distribution of possible execution times for a given task. However, it is generally accepted that such methods are insufficient to derive hard bounds on execution times, as even for moderately complex programs it is infeasible to characterise their timing properties under all possible combinations of inputs and context. Static methods give better means to reason about code path coverage and permit safe over-approximations of hardware-induced timing effects.

In practice, algorithms may combine static analysis with measurements to derive a WCET. For example, Park et al. [54] combine a static program analysis with a processor-behaviour analysis (PBA) that uses the dual-loop measurement technique to determine the run time of single- or small blocks of instructions. This particular approach may not be advisable for modern processors, as Altman et al. [55] show that minor variations in hardware, program binary or run-time conditions can lead to large variations in measured execution time. However, carefully designed measurement-based hardware characterisations can provide a compelling alternative to complex formal processor models.

Static analysis based algorithms perform three major tasks: control flow analysis (CFA), processor-behaviour analysis (PBA) and bound calculation. I next explain the purpose and some of the challenges for each.

2.3.2.1 Control flow analysis

The purpose of control flow analysis (CFA) is to represent a program's code in the form of a graph. The type of graph produced depends on the choice of algorithms later on: for path-based algorithms a control-flow graph (CFG) with explicit control information is produced, while for structure-based algorithms the control flow is more implicitly encoded in a syntax tree.

To derive *tight* WCET bounds, the major challenge in CFA is to eliminate *false code paths*: paths that are either impossible to reach or forbid computing a WCET altogether. False paths may contain unbounded loops, if-statements with (mutually) exclusive conditions or sequences of loops for which a bound exist on the iteration count of the sequences as a whole but not for the individual loop. Analysis can eliminate paths or bound (re-)entry of certain code regions by deriving *flow information*; additional information that excludes or limits the entry of certain code regions. Such flow information can be either conveyed by the system developer as annotations [56, 57] or extracted automatically from a program by means of value analysis of control-flow-deciding variables [58–60].

2.3.2.2 Processor behaviour analysis

The processor-behaviour analysis (PBA) task is responsible for computing the execution cost of the various paths in the graph. This analysis takes into account the targeted processor's pipeline, memory subsystem and other relevant I/O-devices. Generally the more complex a processor is, the less precise the results of this step will be. As an alternative to computing such overheads with extensive system models, measurement-based techniques like simulation can be applied.

Many different hardware effects must be modelled. To account for pipelining, Zhang et al [61] provide a technique that captures pipelining effects for a 2-stage pipelined processor which prioritises memory read/write over instruction fetch. They perform analysis on a basic block (BB) level using an elaborate and tailored formal model that tries to derive the level of overlap of both pipeline stages. Lim et al [62] point out that this does not take into account pipeline effects crossing BB boundaries and deeper pipeline problems, and improves on this result by tracking the pipeline state in a *resource reservation table*. This table contains one row for every pipeline stage. Engblom et al. [63] use a pipeline simulation technique to capture pipeline effects. Unfortunately, none of the mentioned methods explicitly address the issue of accounting for data hazards. However, I note that Lim's approach can easily be extended to address data hazards by adding one resource to the table for every general purpose register in the system.

Colin et al. [64] present a safe analysis technique for *Branch Target Buffer*-based branch predictors like the Pentium processor, allowing to more accurately predict pipeline flushes in a program. For single-core application processors, analysis techniques exist that safely estimate the state of instruction caches [65, 66], L1 data caches [67] and instruction DRAM in Harvard architectures [68]. It is deemed unlikely that tight PBA techniques emerge for shared associative caches [69] as they prevent analysing the timing properties of applications in isolation. Hardware can promote performance isolation between tasks on different cores by supporting cache locking and partitioning [70].

2.3.2.3 Bound calculation

The final step in a static WCET derivation algorithm is bound calculation. In this step, the critical path is sought in the timing-annotated graph that results from PBA. Three methods are identified for such calculation: structure-based, path-based and implicit path enumeration (IPET).

Structure-based bound calculation methods take a timing-annotated *syntax tree* as its input. This tree is processed depth-first to find the globally critical path. For example, Colin et al. [64] apply structure-based bound calculation in their branch-predictor aware algorithm. Although structure-based methods are effective for bound calculation, the structure of a syntax tree is limited in its abilities to represent the true control flow of programs. This may result in an inability to represent an arbitrary program binary after performing optimisations.

A more natural structure to capture program control flow is a CFG. To perform critical-path analysis on such graphs directly, path-based bound calculations (e.g. Healy et al. [71]) are used. Compared to syntax trees, such graphs can represent a wider range of programs as it makes control flow more explicit. This expressiveness comes with the downside of conveying accurate cost information. For example, Wilhelm et al. [53] highlight the difficulty with conveying context- and cost information across loops at different nesting depths.

A third method for bound calculation is implicit path enumeration (IPET). These methods express the critical path problem as a set of constraints, to be solved through integer linear programming (ILP). A major advantage of IPET-based methods is that the CFA's false code-path elimination can be performed by generating additional constraints for the ILP model, removing the need for two separate analysis passes. For example, Engblom et al. [57] demonstrate a technique encoding loop-bound and control-flow "facts" into their ILP model constraints. However, ILP is a known NP-complete problem [72], meaning a solution is not guaranteed in feasible time.

2.3.2.4 Sim-D

The WCET analysis work performed for Sim-D primarily serves as a proof-of-concept of the architectural decisions. By gathering empirical data on the WCET of a range of benchmarks, and contrasting them to measured average-case performance times, the analysis is used to show Sim-D's ability to support derivation of tight and safe WCET bounds.

I do not make new contributions to the practice of CFA. Predicated execution of if-then-else blocks already limit the scope of benefiting from such analysis, as in the worst-case execution of a work-group both the if- and the else-branch must be executed. Instead, Sim-D applies path-based analysis on a regular CFG and applies a fairly simple annotation-based technique to bound loop iterations. This analysis covers most special cases in all the benchmarks investigated.

Sim-D's PBA is performed by means of pipeline simulation. Sim-D's performance-isolated in-order pipeline permits capturing all possible timing effects safely and tightly. One interesting finding in this area is that for each BB there are only two pipeline states to consider for timing: either a pipeline warmed up by the BB(s) directly preceding it in memory, or a cold pipeline when branching from any other BB. This significantly limits the amount of code paths for which the pipeline must be simulated.

The main contribution for Sim-D's WCET analysis algorithm is in its final bound calculation phase. For a hardware strip-mining architecture, the longest path algorithm through the program generates the critical path of a work-group rather than a program. I present a technique that uses the timing information encoded in this critical path and Sim-D's work-group scheduling behaviour to derive the kernel-instance's WCET. To the best of my knowledge, no prior work has looked at WCET analysis for architectures that perform hardware strip-mining.

2.4 Miscellaneous related work

The access/execute paradigm [73–75] architecturally decouples data access from computation. In its strictest form, the paradigm mandates separate instruction *streams* for access and execute operations. Decoupling access from execute essentially provides a lot of scheduling freedom to provide prefetching of data, at the cost of complex synchronisation between the two streams to ensure data arrives just in time for processing.

Loosely inspired by this access/execute paradigm, the PRedictable Execution Model (PREM) [7, 76, 77] transforms programs into a succession of compute- and memory *phases*. These phases execute on separate resources such as compute, DRAM, scratchpads and other peripherals. They observe that the WCET of a program can be made significantly less pessimistic by taking control of the scheduling of program phases and external events on (shared) resources. PREM mainly uses software techniques to minimise performance interference between components, which reduces pessimism in their derived WCET bounds while allowing their model to apply to commercial off-the-shelf hardware.

This execution model provides the basis for Sim-D's predictable work-group scheduling. However, rather than isolating resources in COTS processors through sophisticated software scheduling and cache prefetching techniques, Sim-D enforces this execution model architecturally. Sim-D achieves strict performance isolation between access- and compute phases by replicating local storage resources for two active work-groups, as will be explained in detail in Chapter 3.

Chen et al. [78] experimented with a compiler pass that would recognise scalar values in SPMD-programs for GPUs, and transform them to use scalar registers and operations. This involved extending the GPU's ISA with scalar instructions, coincidently bringing it more in line with the ISA of a traditional vector processor. This work has inspired Sim-D adoption of a mixed scalar-vector ISA, aiding with the desire to encode DRAM- and scratchpad requests as work-group-wide "scalar" instructions.

Huangfu et al. [5] introduce a WCET analysis method for a simplified GPU model. In this work, they introduce the greedy-then-round-robin (GTRR) warp scheduling strategy that allows modelling the execution of a kernel-instance as a sequence of code segments, interleaving the code segments of multiple warps. Sim-D schedules work-groups in a similar fashion to GTRR, executing instructions greedily from a single work-group until a DRAM- or scratchpad request is encountered.

CHAPTER 3

A WIDE-SIMD ARCHITECTURE

In this chapter I present the Sim-D wide-SIMD architecture for hard real-time systems. The goal of this chapter is to demonstrate feasibility, within the limitations of a cycle-accurate simulation model, of a SIMD processor pipeline that provides performance isolation among its resources. I make the following contributions:

1. A high level overview of the Sim-D wide-SIMD architecture and its mixed scalar-vector ISA (Section 3.1),
2. An explanation of Sim-D's control logic (Section 3.2),
3. Details about Sim-D's four-phase in-order pipeline, for which the decode and execute phases have a configurable number of stages (Section 3.3),
4. An overview of the control flow mechanisms present in this mixed scalar-vector architecture (Section 3.4),
5. An overview of the performance-isolated storage facilities present on Sim-D's data path: scratchpads, register files, buffer mapping tables and a scoreboard for tracking RAW hazards. A justification is given for a three-stage decode pipeline phase, permitting efficient fetching of vector operands of warps using 1R1W storage cells (Section 3.5).

3.1 Overview

In this section I present the Sim-D hard real-time massively-parallel processor simulator. The key feature of this architecture is performance isolation between the compute, DRAM and scratchpad resources. What this means is that once a work-group obtains exclusive access to a resource for executing a section of it's program, be it compute or a DRAM request, the time it takes to complete this *program phase* is independent of what happens on other resources. Treating each access- and execute *phase* in a program as an independent critical section, free of performance interference, reduces the problem of finding a kernel's WCET to one of finding the worst-case schedule of program phases.

There are two ways of maximising the occupancy of these performance-isolated resources: either overlap multiple program phases from the same work-group, or schedule phases from multiple work-groups in parallel. The latter approach is more likely to yield high occupancy: kernel-instances provide ample work-groups to overlap, and the absence of intra-work-group dependencies offer substantial freedom on how work-groups can interleave. To exploit intra-work-group parallelism efficiently and predictably, Sim-D implements a *double-buffered* execution model processing up to two work-groups in parallel. Hardware scheduling of work-groups follows a policy similar to greedy-then-round-robin (GTRR) [5].

A high-level overview of the Sim-D architecture is given in Figure 3.1.

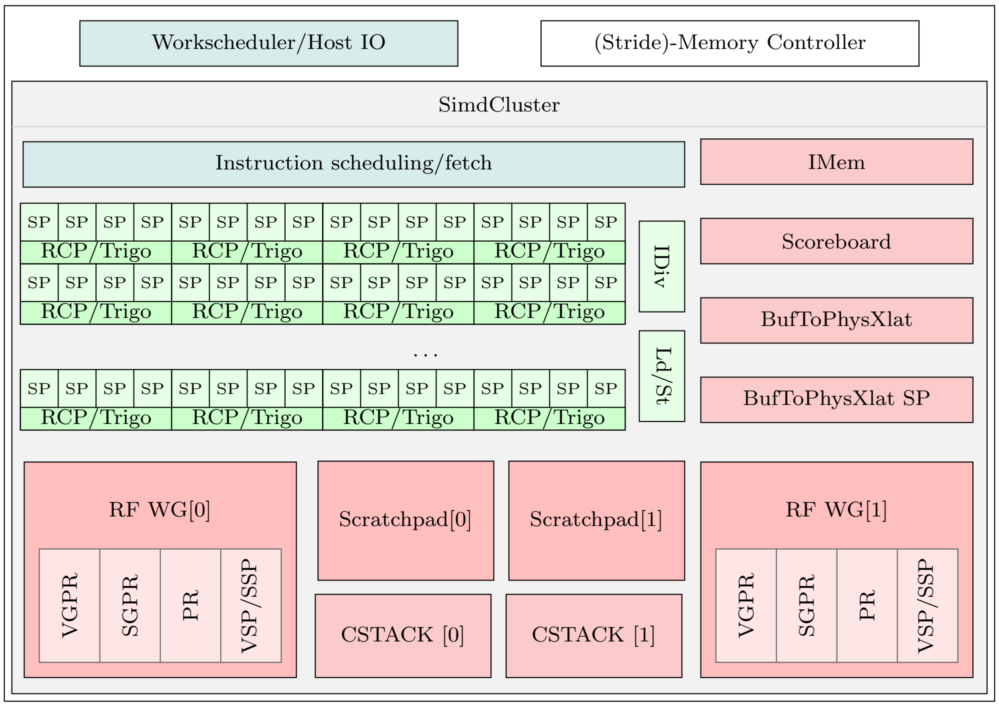


Figure 3.1: High level overview of the Sim-D simulator.

At the heart of the architecture lies one compute unit called the *SimdCluster*. It contains a configurable number of arithmetic units, plus logic and storage for executing two work-groups.

Performance isolation of resources is achieved through two design decisions. Firstly, local storage resources (i.e. register files, scratchpads and control stacks (CSTACKs)) are replicated for both active work-groups to permit a DRAM↔register file or DRAM↔scratchpad transfer to occur in parallel with a compute phase of a different work-group. Secondly, Sim-D adopts a Harvard architecture with a dedicated instruction memory (IMem) to eliminate interference on the DRAM bus between instruction fetch and data fetch. Section 6.1

provides evidence to justify the choice for a Harvard architecture.

I have implemented a full simulation model of Sim-D architecture in SystemC, an event-driven hardware modelling library for C++ [79, 80]. SystemC allows easy integration of the Ramulator [81] DRAM timing model and the DRAMPower [82] DRAM power model.

3.1.1 Instruction set architecture

For this simulation model, I define the ISA as a higher level specification, without considering a specific instruction encoding. I constrain Sim-D's ISA design in line with NVIDIA's Kepler ISA as documented in e.g. Envytools [83]:

- Each opcode is 64-bits,
- Instructions have no more than 3 source operands,
- Vector arithmetic/boolean logic instructions have no more than one immediate-, scalar- or special-purpose source operand,
- Instructions have no more than 1 destination operand.

Defining a complete instruction encoding could uncover mandatory changes to the ISA. Specifically, limitations in code space may forbid encoding immediate values as operands. The alternative of allowing immediate values only in *mov* instructions would have a small impact on performance as a program will be made up of different and potentially more instructions than when encoded in the current ISA. I believe that in the light of the high level objectives (analysable WCET, direct memory access (DMA)-style DRAM requests), this minor effect on simulated performance bears insufficient significance to justify a more low-level definition of the ISA at this stage of research. A full overview of the ISA is given in Appendix A.

3.2 Control logic

The control logic of Sim-D's simulation model has two responsibilities: program launch and hardware strip-mining. In the process of strip-mining, many work-items are spawned each of which must be identified with a unique global ID (TID). In this section, I explain how both responsibilities are fulfilled, followed by a discussion on the assignment of TIDs to work-items.

Note that the simulator currently does not model data transfers between the host system and the device's DRAM. It is assumed that buffers and programs have already been uploaded to DRAM prior to kernel launch. Studying such data movements is left for future work.

3.2.1 Program launch and hardware strip-mining

The host launches a kernel-instance by sending a *launch descriptor* to the workscheduler. This launch descriptor contains the NDRange, the work-group dimensions, a pointer to the program binary and a list of buffer mappings. Upon receiving a launch descriptor, the workscheduler will transition through three states: SimdCluster initialisation, work-group enumeration and wait for completion.

In the first state, the workscheduler distributes the buffer mappings (see Section 3.5.3) and kernel code to the SimdCluster's dedicated memories. Sim-D models program upload to IMem as a load from DRAM. Buffer mappings are part of the launch descriptor and hence distributed to the SimdCluster in parallel with program upload through dedicated channels. The initialisation latency is thus that of a DRAM read, for which the cost is a function of the program size and DRAM timings. An equation for computing this latency will be introduced in Section 5.4.1.

In the work-group enumeration state, strip-mining is performed to instantiate work-groups covering the kernel-instance's NDRange. To this end, the workscheduler fills a FIFO with work-group descriptors, to be consumed by the SimdCluster. Every work-group consists of 1024 work-items, each of which is uniquely identified with a two-dimensional TID (tid_x,tid_y). A work-group descriptor contains the TID of its first work-item and the (x,y)-dimensions of the work-group. The workscheduler can sustain a rate of one work-group per cycle.

A SimdCluster has two active-work-group slots. When a slot becomes free and all scheduling constraints have been met, the SimdCluster consumes a work-group from the FIFO and initialises all relevant state. Once the compute resource becomes available, the SimdCluster starts scheduling this work-group's instructions.

Once the workscheduler adds the last work-group to the FIFO, it transitions into the *wait for completion* state. It will then wait for the SimdCluster to consume and finish execution of the final work-group. Once the SimdCluster signals completion, the host is notified of kernel execution completion and the workscheduler returns to idle, ready to accept the next launch descriptor.

No overlapping of work-groups from different kernel-instances is considered, as it is likely that their contexts (IMem content, mapped buffers) differ. I leave research towards parallel execution under multiple contexts, and the broader topic of spatial- and temporal multitasking for future work.

3.2.2 Assigning global IDs to work-items

Work-items in CUDA and OpenCL kernel-instances use their global ID (TID) for three main purposes: for calculating offsets within input- and output buffers, for special handling of the edges of a data set, and for early termination of out-of-bound work-items.

Given these use-cases, the mapping from TIDs to SIMD-lanes influences Sim-D's design in three ways. Firstly, the logic required in each SIMD-lane to calculate the TID of an active work-group differs between mappings. Secondly, Sim-D's large coalesced 1D and 2D block transfers, which assume a linear relationship from data elements to work-item, require the DRAM controller to compute the corresponding vector register column for each data element in such a transfer. Finally, for remainder work-groups the chosen mapping impacts which SIMD lanes must be disabled. Although in Sim-D the disabling of lanes is done in software upon work-group launch, the observation that this can lead to disabling entire warps poses an opportunity to trim these warps from a work-group. The potential efficacy of such warp trimming differs between mappings.

In the light of these implementation details, I next justify Sim-D's *linear* TID-to-SIMD-lane mapping scheme by contrasting it against a *compacted* scheme. To illustrate the difference between these two schemes, assume a kernel-instance with work-groups of 16 work-items, processing a $62 * 62$ image in tiles of $8 * 2$. Work-groups processing the right edge of this image require only $6 * 2$ work-items, the remaining four work-items fall outside the NDRange. Figure 3.2 demonstrates both mappings for the top-right work-group:

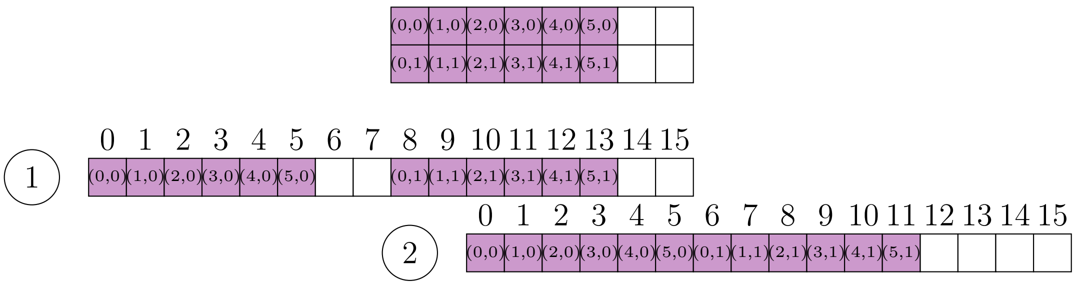


Figure 3.2: Two mappings from TID to SIMD lane: 1) linear, 2) compacted.

TID compute logic Each work-items's TID is determined as a mapping-specific function of the work-group's offset within the NDRange, its width, the warp number and the SIMD lane identifier. When there are no restrictions on the TID for an individual lane, the computation of TIDs can require significant arithmetic logic that must replicated for each SIMD-lane. To simplify such logic, it is thus desirable to choose a mapping that restricts possible TID values for a SIMD-lane.

No such restrictions can be established for the compacted mapping. As a result, TID computation requires expensive integer operations, which include a modulo operation to transform a 1D TID into a 2D TID.

For a linear mapping, restricting the x- and y-dimensions of a work-group to powers-of-two greatly reduces this logic. Integer modulo with a powers-of-two modulus can be implemented using boolean OR and AND operations, and the lower bits of the TID can be hard-wired to the SIMD lane number. Hence for the linear mapping, the TID for each SIMD-lane can be derived using much simpler boolean logic and muxes. This requires fewer hardware resources and their critical path likely fits in a single cycle.

DRAM destination calculation When transferring a 1D or 2D tile of data directly into a vector register, the DRAM controller must calculate the destination vector register column for each word within that tile. Chapter 5 explains in greater detail how a block transfer descriptor is sequentially translated into a number of DRAM burst requests. On a high level, the DRAM controller iterates over all the bursts contained in the range from start to end address, and for each word in each burst it tests whether that word falls lies within the 1D or 2D block. For this test, the DRAM controller contains one subcomponent for every word in a burst. Each subcomponent keeps track which 2D-coordinate in the buffer it is currently processing. Conceptually, translating this coordinate to a vector register offset is no more complex as calculating the coordinate for the next burst from the current coordinate. Although the destination calculation logic would differ for both mappings, there is no reason for one to be significantly more expensive than the other.

Warp trimming Some remainder work-groups end up disabling entire warps. When executing vector instructions, such empty warps can potentially be skipped to save cycles. *Warp trimming* is the concept of removing warps from the work-group at run-time.

To illustrate the influence of the TID mapping on warp trimming, consider a hypothetical configuration for which each work-group consists of 64 work-items, 8 work-items per warp. The "bottom-right" remainder work-group processes a $6 * 6$ tile from an input buffer. Figure 3.3 shows the resulting mapping under both schemes.

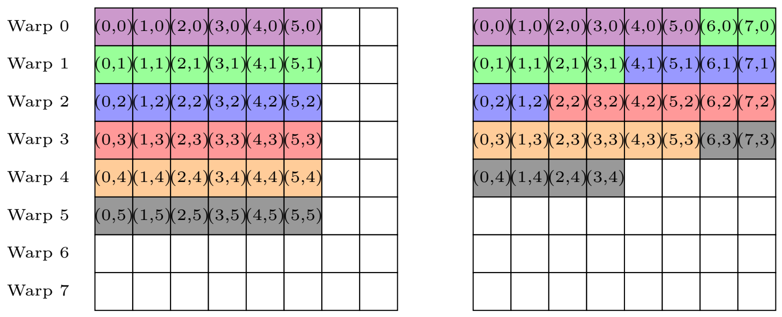


Figure 3.3: Two mappings from global ID (TID) to SIMD lane: linear (left), compacted (right).

As this figure shows, in theory the compacted mapping can result in higher performance: under the linear mapping the demonstrated work-group has six active warps, while the compacting mapping allows to trim this work-group to 5 warps. In practice, Section 6.4.2 shows that warp trimming has only a negligible potential for improved performance when considering all work-groups of a kernel-instance. For this reason, I instead opt for pipeline- and WCET analysis simplicity and always run every work-group with 8 warps.

3.3 Pipeline

Sim-D's SimdCluster implements an in-order, single issue four-phase pipeline: fetch, decode, execute, write-back. Both the decode and execute phases consist of a configurable number of pipeline stages. In this section I explain each pipeline phase in greater detail.

3.3.1 Fetch

The fetch stage, *IFetch*, is responsible for timely fetching the next instruction of the active work-group. IFetch contains one program counter (PC) per active work-group. On each cycle, it increments the value of the active PC and requests the instruction at this PC asynchronously from IMem, such that the instruction is presented to the decoder in the next cycle. When the execute pipeline stage commits a control flow or load/store instructions, it instructs IFetch to overwrite the work-group's PC.

The fetch logic is slightly more complex than in a regular in-order pipeline. This is due to three differences in design.

- Double-buffered execution requiring switching between two work-groups,
- Instructions not flowing through the pipeline at a uniform rate,
- Injection of *cpop* instructions into the pipeline.

Double-buffered execution has the implication that IFetch needs to maintain two PCs. To make sure IFetch issues a valid PCs even in the event of a work-group switch, PCs are post-increment. Because the PC issued to IMem could be from a different work-group than the PC provided by instruction write-back, the execute pipeline phase must explicitly provide the target work-group upon issuing a PC update.

To ensure that program phases execute uninterruptibly, a work-group switch occurs at the end of a phase: either upon issuing a DRAM or scratchpad request, or on work-group exit. Note that the former situation does not cause a work-group switch if there is only a single active work-group, for instance when executing the last work-group of a kernel-instance. IExecute explicitly issues a PC write upon issuing a DRAM- or scratchpad transfer and on work-group exit, to ensure the PC is valid upon continuation. The value of the written PC is either the instruction directly after the load/store operation, or 0 if the switch is caused by work-group exit.

Instructions do not flow through the pipeline at a uniform rate due to *vector instruction enumeration*. For DRAM efficiency reasons, a work-group contains 1024 work-items.

However, to balance data transfer time with compute time, a SimdCluster contains fewer SP-units. In Section 6.2 I experiment with configurations of 64, 128 and 256 SP-units. A single vector instruction in a program will therefore occupy the instruction decode phase between 4 and 16 cycles, providing back-pressure on IFetch to ensure correct program flow.

Finally, as part of Sim-D's control flow mechanisms (explained in Section 3.4), control stack pop (*cpop*) operations could be injected in program execution whenever a control mask (CMASK) write disables all work-items in a work-group. The popped control stack entry contains the PC value for continuation of program execution, which will be written by the execute pipeline phase.

3.3.2 Decode

The instruction decoder is responsible for reading operands from the register files. The length of this pipeline phase is configurable: depending on the register file configuration, either one or three cycles can be spent on fetching operands. A more elaborate analysis of register file configurations follows in Section 3.5.2.

On each cycle, the instruction decoder translates one instruction into control signals to perform any of the following data operations:

1. Register file reads,
2. Scoreboard match and destination write,
3. Control stack read,
4. BufToPhysXlat physical address lookup for DRAM and/or scratchpad buffers.

These data operations, explained in more detail in Section 3.5, all occur in parallel. For instructions with implicit destination operands, IDecode generates the write control signals to pass on to both the execute pipeline stage as well as to the scoreboard. This scoreboard is used to detect and stall on read-after-write hazards, as explained in Section 3.5.4.

The decode phase *enumerates* vector instructions. The number of warps to replicate a vector instruction for is determined by dividing the number of work-items per work-group (1024) by the number of computational resources provisioned for this instruction. When a vector instruction appears at IDecode's inputs, IDecode applies back-pressure on IFetch to retain the current instruction on its outputs while enumeration takes place.

3.3.3 Execute

The execute phase performs the actual computation requested through the incoming control signals and operands, and generates the required output signals for write-back of results. To this end, the execute unit contains a multitude of compute resources.

As the number of pipeline stages required for each operation cannot be determined from a high-level simulation model, Sim-D instead resorts to a configurable number of pipeline stages. In Section 6.2 I analyse the sensitivity of benchmark performance to the execute pipeline depth.

IExecute also issues DMA-style DRAM- and scratchpad read/write requests. A separate load/store unit generates the control signals for the respective memory controllers. Both types of resources are explained in greater detail next.

3.3.3.1 Compute resources

A SimdCluster contains an array of SP-units performing 32-bit integer- and floating-point arithmetic, several ROMs and logic for reciprocal- and trigonometric operations, an integer divider and a load/store unit. In the simulator, the number of SP-units is configurable in powers of two between 4 and the maximum number of work-items in a work-group.

Based on NVIDIA patents [84–86], pipelined floating-point reciprocal, reciprocal square root and trigonometric operations are modelled using four multipliers and various look-up tables (LUTs) per lane. Rather than adding dedicated multipliers for these operations, I assume the SP-units' multipliers can be re-used. Hence Sim-D models one reciprocal/trigonometry unit (RCP-unit) for every four SP-units, the same ratio as the NVIDIA GP100 (Pascal) GPUs [87]. Under this implementation, the throughput for (vector) reciprocal and trigonometric instructions is $\frac{1}{4}$ that of regular arithmetic.

Inspired by Chen et al. [78], Sim-D supports bit-wise and integer scalar operations. The main use-cases for scalar operations are loop iteration, computing values shared with all work-items in a work-group, and the construction of (stride) DRAM requests. Scalar operations are performed on the first SP-unit. To effectively use shared scalar values in a kernel, many vector instructions support taking one scalar source operand.

From benchmarks I observed that integer division is valuable for shared loop invariants and DRAM/scratchpad pointer arithmetic. For such operations a single scalar *IDiv*-unit is modelled after Intel's 8-cycle non-pipelined radix-16 SRT divider [88]. I have not identified a need for vector integer division. If required, programmers could refer to Juffa's method for performing pipelined integer division using the vector floating-point reciprocal [89], an operation implemented by the RCP-units.

Researching reduced-precision numbers, a trend observed in neural network processing [24, 90, 91], is left for future work. The implications of reduced-precision arithmetic on DRAM transfers and compute resource provisioning are expected to differ between HRT accelerators and general-purpose accelerators. However, I believe results from experiments in this area are orthogonal to the theory of WCET analysis presented in this work.

3.3.3.2 Load/store units

At the heart of Sim-D's architecture lies the ability to issue explicitly-coalesced load/store operations to DRAM and scratchpads. The load/store unit is responsible for preparing such requests. Chapter 5 presents a memory controller that supports such transfers.

To cover all general cases, Sim-D supports two different types of load/store requests: scalar transfers to one- or multiple consecutive scalar registers, and indexed (indirect) transfers to a vector register. To optimise for common use-cases, Sim-D additionally supports instructions that transfer 1D- or 2D tiles of data into a vector register or a scratchpad. Sim-D's DRAM controller supports two types of indexed transfers (iterative and snoopy) with different performance characteristics, as analysed in detail in Chapter 5. Finally, since the scratchpad and the DRAM controller share a similar front-end, Sim-D supports 1D/2D tile transfers and indexed transfers between a scratchpad and a vector register.

A snoopy indexed transfer can either cover an entire buffer, or its range can be limited to a 1D- or 2D block within a buffer. Limiting the range of a snoopy indexed transfer reduces both average-case and worst-case performance. This is particularly useful when reading or writing elements from an array-of-structs.

LD/ST instruction type	Loads	Stores	Total
1D/2D block	41	20	61
Scalar	17	-	17
DRAM snoopy indexed, full buffer	7	1	8
DRAM snoopy indexed, 1D/2D block	12	6	18
DRAM iterative indexed	9	0	9
Scratchpad snoopy indexed, full buffer	17	14	31
Total	103	41	144

Table 3.1: Frequency of occurrence, load/store instructions.

Table 3.1 shows for each class of load/store instructions how frequently they occur in the set of benchmarks ported to Sim-D. To justify the set of supported transfers, I highlight two things. Firstly, more than 4 in 10 data transfers are effectively described as 1D or 2D block transfers. Secondly, over half of all indexed DRAM transfers (18 out of 35) perform best when performed as a snoopy indexed transfer over a limited 1D or 2D block.

To prepare each of the transfers above, the load/store unit must compute the necessary parameters. The full set of parameters is described in Section 5.3.1.1. For computing the start- and end addresses of a transfer, an integer multiply-addition of various values from the buffer mapping (provided by the BufToPhysXLat components) and the work-group descriptor is required. Most other parameters are determined by a min-operation on values derived from the NDRange, the work-item descriptor and the instruction issuing the request. In Sim-D's simulation model I assume that either the existing SP-units or specialised arithmetic units can be used to calculate these parameters without additional pipeline delays.

3.3.4 Write-back

The write-back pipeline stage commits the results from the execute stage to the register file, the DRAM or scratchpad request FIFOs, a CMASK, the control stack (CSTACK) and/or the active work-group's PC in the IFetch component. For writes to the register file and the CSTACK, the scoreboard is updated accordingly.

No write-back component is modelled explicitly inside Sim-D. Rather, all necessary control signals for write-back are provided on the outputs of the execute pipeline phase.

3.4 Control flow

Sim-D supports two forms of control flow: per-work-item *vector control flow*, and work-group-wide scalar (un)conditional branches. This section describes the mechanisms and required hardware for both forms.

3.4.1 Vector control flow

To allow work-items within a work-group to follow diverging code paths, Sim-D makes use of *implicit predicated execution*. Implicit means that every vector operation executes conditionally on the same single per-work-group predicate mask; no explicit predicate mask is encoded in the instruction. Coon et al. [92] describe how GPUs can implement implicit predicated execution with support for arbitrary nesting of for-loops, while-loops and function calls which may contain (properly scoped) break-, continue- and return statements. Following these principles, Sim-D supports such rich control flow using two components: hardware-managed predicate registers and a control stack.

3.4.1.1 Predicate registers

Sim-D derives its implicit predicated execution mask from four special vector predicate registers, henceforth CMASK registers: *vc.ctrl_run*, *vc.ctrl_brk*, *vc.ctrl_ret* and *vc.ctrl_exit*. Each of these registers have one bit for each work-item in a work-group. Upon work-group launch these registers are initialised to all-1, indicating that all work-items are enabled. The implicit predicate that applies to each vector instruction is the bitwise boolean AND result of all four CMASK registers.

Each CMASK register permit disabling work-items for different reasons. By convention, vc.ctrl_run is used to disable work-items that do not meet the condition of an if-statement. vc.ctrl_brk is used to disable work-items that hit a break statement inside a loop. vc.ctrl_ret is used to let work-items return early from a function call. Finally, vc.ctrl_exit indicates (early) exit of a work-item.

Separating the predicate mask into these four registers is crucial for allowing the nesting of various constructions. For example, breaking out of a loop inside an if-block is facilitated by clearing the work-items' bits in the ctrl_brk mask. At the end of the if-block, the previous ctrl_run mask is restored. As these registers are separate, restoration of the ctrl_run mask does not result in re-enabling the work-items that hit the break statement.

A control flow operation may cause all work-item in a work-group to be disabled. To detect this situation, the register file (RF) generates a *work-item active* signal from the boolean OR of all bits in the implicit predicate mask. When this signal becomes 0, the instruction decoder *injects* a *cpop* operation in the instruction stream, causing a CSTACK entry to be popped. This cpop-injection repeats until at least one work-item is reactivated and the work-item-active signal becomes 1.

3.4.1.2 Control stack

Under implicit predicated execution, the control flow operation at the start of an loop, if-(else-)block or function call pushes one or more entries to the CSTACK. Each entry contains the CMASK that must be restored after this conditional execution, and the *reconvergence point* in the program where execution should continue.

Each control stack entry consists of a (PC, predicate mask, predicate type) 3-tuple. The PC points to the *reconvergence point.* The predicate mask contains the restore value for the CMASK register identified by the predicate type, which is equal to its value prior to diverging. The predicate type is any of *run*, *brk* or *call*, matching three out of four CMASK registers. The exit mask cannot be restored as early exit of a work-item is final.

3.4.2 Scalar control flow

In addition to implicit predicated vector control flow, Sim-D supports conditional- and unconditional scalar jumps to provide an easy-to-analyse, low-overhead mechanism for implementing for-loops and if-statements with scalar iterators and conditions. These methods do not make use of the control stack or predicate registers, but simply update the program counter.

3.4.3 Usage example

The example OpenCL kernel in Listing 3.1 implements a toy kernel that nests an if-statement inside a for-loop. To explain the control flow in this example code, I analyse the body from the inner to the outer scope.

```
__kernel void scratch(
        int __global *in,
        int __global *out) {
      int x = get_global_id(0);
      int v = in[x];
      int i;

      for (i = 0; i < x; i++) {
            v = v + x;

            if (v % 8) {
                  v += 128;
                  break;
            }

            v += i;
      }

      out[x] = v;
}
```

Listing 3.1: OpenCL C

```
.data
      0 0x0       640 480 //in
      1 0x12c000 640 480 //out

.text
      mov v0, vc.tid_x
      smov s0, 0
      ldglin v1, 0

      cpush.brk out

loop:
      isub v2, v1, s0
      itest.le p0, v2
      brk p0

      iadd v1, v1, v0
      and v2, v1, 0x7
      cpush.if no_early_brk
      itest.nz p0, v2
      itest.ez p1, v2
      cmask p0
      iadd v1, v1, 128
      brk p1
      cpop

no_early_brk:
      iadd v1, v1, s0
      siadd s0, s0, 1
      j loop

out:
      stglin v1, 1
      exit
```

Listing 3.2: Sim-D assembly

Table 3.2: Side-by-side comparison of a toy kernel in OpenCL C and Sim-D assembly.

The assembly implementation in Listing 3.2 requires two CSTACK entries: One entry is pushed in line A10 with a reconvergence point labelled “out” at line A32, and another entry is pushed in line A19 reconverging at the “no_early_brk” label in A27.

The inner if-block in lines C11-C14 translate to the assembly in lines A17-A25. Lines A17-A21 perform the test “v % 8”, generating both a predicate mask for the case this test evaluates to true (p0) and false (p1). The *cmask* operation in line A22 disables all work-items for which the condition ”v % 8” is false, continuing with the enabled work-items down the conditional code in C12-C13.

All work-items that are left enabled will execute the *brk* instruction in A24. After executing this instruction, all work-items must necessarily be disabled, either by the run-CMASK, or by the brk-CMASK. This triggers IDecode to inject a *cpop* into the pipeline. Indeed, the *cpop* instruction in line A25 will never be executed in practice, but in absence of a static analysis technique that infers the values of the predicate registers, this line is required for correct control-flow analysis.

If the *cmask* instruction in line A22 results in disabling all work-items, IDecode will inject a *cpop* instruction into the pipeline, popping the top entry of the stack. Control then resumes from the “no_early_brk” reconvergence point at line A27 without entering the body of the if-block.

The loop body from lines C8-C17 translates to the assembly in lines A7 and A10-A30. The loop invariant is tested in A13-A15, using a *brk* instruction to exit the loop when finished. Loop re-entry and iterator increment is achieved in A29-A30 using scalar operations.

Exiting the loop occurs when all work-items have executed the *brk* instruction in either line A15 or A24. If the last work-item is disabled by executing line A15, the top entry on the CSTACK is the reconvergence point past the for-loop, accompanied with a break-mask that re-enables all work-items. If the last work-item was disabled by the *brk* instruction in line A24, the “run”-type entry pushed in line A19 is popped from the stack instead. Popping this “run”-type entry and updating vc.ctrl_run will not re-enable any work-items. Hence after executing the injected *cpop* instruction, the *no work-items active* flag remains set, causing a second injected *cpop* to pop the entry from the control stack that was pushed in line A10. Control will then continue from the reconvergence point in line A32.

3.5 Data path

A high level schematic overview of the data path is depicted in Figure 3.4. The remainder of this section describes the data sources marked in red in greater detail.

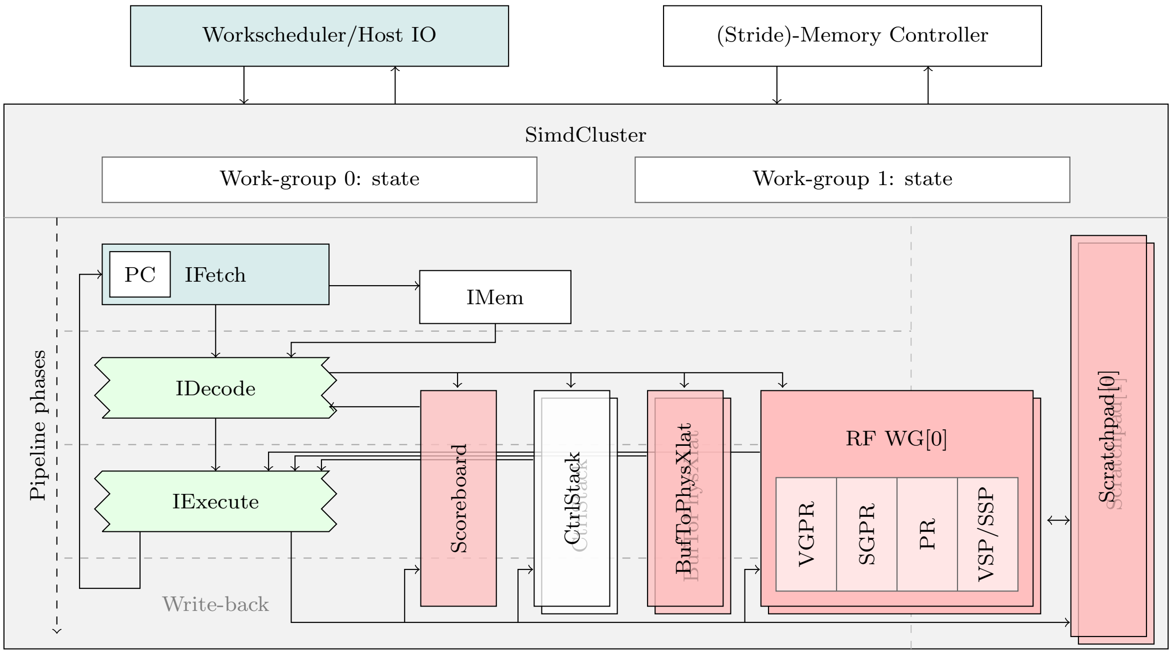


Figure 3.4: Data path overview of the Sim-D simulator.

3.5.1 Scratchpads

Sim-D contains two addressable local storage units, *scratchpads*, each dedicated to one work-group slot for performance isolation reasons. These scratchpads fulfil the role of OpenCL's local memory. A scratchpad can host multiple buffers. The mapping from a buffer index to an offset in the scratchpad is stored in a dedicated BufToPhysXLat components, explained in greater detail in Section 3.5.3.

Scratchpads are implemented in hardware using (1R1W) SRAM cells. Such cells facilitate lower latency access than DRAM cells, as SRAM does not suffer delays caused by bank precharging, activation and refresh. As a beneficial side-effect of SRAM's simpler timing properties, upper bounds on scratchpad data arrival times are tighter than those on DRAM requests. Discrepancy between average- and worst-case DRAM request latencies are mainly the result of data alignment uncertainty. With SRAM storage, this uncertainty can only cause a 1-cycle difference between the best and worst case.

Each scratchpad has two interfaces: one to transfer data between scratchpads and the RF, and one to transfer data between DRAM and scratchpads.

For scratchpad↔RF communication, scratchpads have a similar front-end to the DRAM controller. The SimdCluster's load/store unit issues large coalesced requests to transfer data between a scratchpad buffer and a register file. Additionally, snoopy indexed transfers are supported. More details on such transfers are given in Chapter 5.

DRAM can transfer data directly into a scratchpad, even if the dimensions of the tile exceed the dimensions of a work-group. To initiate such a transfer, the SimdCluster issues a request to the DRAM controller specifying the scratchpad as its target destination. The dimensions for this transfer are taken from the mapping in the BufToPhysXLat unit.

Applications on Sim-D are expected to have a higher demand for scratchpad memory than on traditional GPUs due to Sim-D's absence of a transparent cache hierarchy. The primary function for local memory is to cache tiles of data shared among the work-items in a work-group, for example when performing filter operations. Pre-loading a tile of data to local memory can both improve throughput and reduce congestion on the DRAM bus.

Benchmarks use Sim-D's scratchpads in various other ways. Firstly, scratchpad buffers are used to efficiently construct array-of-struct values one struct member at a time, before writing these values back to DRAM in one contiguous transfer. Secondly, scratchpad buffers serve to cache scalar values; values shared among all work-items of a work-group. For example, the CNN convolution benchmark shares $7 * 7 * 3$ kernel values among a whole work-group. Pre-loading these kernel values into the scratchpad allows issuing a single 147-word DRAM transfer without the need for 147 scalar registers.

The size of each scratchpad is currently (over)provisioned at 128KiB, and its data bus width is configurable during compilation. Static benchmark analysis in Section 6.1.2 provides data on the scratchpad usage of different benchmarks. Section 6.2.1 quantifies the influence of the scratchpad bus width on performance.

3.5.2 Register Files

To provide fast temporary storage to the compute units with strong performance isolation guarantees, Sim-D contains one register file (RF) per work-group. Each RF manages different types of register banks:

- General purpose vector registers (VGPRs),
- General purpose scalar registers (SGPRs),
- (vector) predicate registers (PRs),
- Special purpose vector registers (VSPs), e.g. TID, CMASK,
- Special purpose scalar registers (SSPs). e.g. kernel dimensions.

Logically, each vector register file (VRF) is organised as a 2D grid of 32-bit registers, rows representing the addressable vector registers (e.g. $v0..v63$) and columns mapping to work-items in a work-group. The predicate register files are organised similarly, but contain 1-bit registers. An overview of the vector- and scalar special registers is given in Appendix A.2.

Each RF has two interfaces: one for the SimdCluster's operand fetch and write-back, and one to transfer data between the register file and DRAM or scratchpad storage. Access to these two interfaces is mutually exclusive as a work-group can only occupy one compute-

or storage resource at a time. The Sim-D simulator assumes that the RF can be clocked at the speed of the DRAM interface (up to 1.6GHz) when transferring data to storage, and can be synchronised to the SimdCluster's pipeline with the use of an efficient clock switching mechanism.

3.5.2.1 Operand interface

Through the operand interface, the RF is responsible for fetching up to three operands per cycle. These operands could be vector registers, scalar registers, predicate registers or special purpose registers. Additionally, the RF can process one vector- or scalar write operation per cycle.

The second operand of a vector instruction can be a scalar register or immediate. For these instructions, the RF broadcasts the required value to the operand input of each work-group. The instruction decoder can provide an immediate value to this broadcast network using a dedicated input.

3.5.2.2 DRAM- and scratchpad interface

The RFs DRAM- and scratchpad interface consists of several control signals produced by the storage resources, along with a wide bi-directional data bus. These control signals determine the direction of the data bus and the register(s) being targeted. For snoopy indexed transfers, these control signals convey the buffer offsets of the data elements that are currently being read or written.

As for indexed transfers any value on the data bus can potentially be routed to every vector register column, routing data from DRAM and scratchpads to the vector registers is performed using a per-RF on-chip crossbar. The dimensions of these crossbars depend on the width of the scratchpad data bus, as this is wider than the DRAM data bus. Under Sim-D's widest scratchpad data bus configuration of 32 words, the dimensions of each crossbar is 32×1024 with 32-bit words. Control signals for this crossbar are generated by the DRAM controller in the case of block transfers and iterative indexed transfers, and by the register file's array-of-CAMs (see Section 5.3.2) for snoopy indexed transfers.

To justify feasibility of a crossbar with such dimensions and latency requirements, I highlight that Cakir et al. [93] demonstrated a 256×256 crossbar with 32-bit data words. This crossbar runs at $\sim$800MHz and is produced with a 40nm process. Bearing in mind recent advances in processing technology, I therefore assume that a crossbar meeting Sim-D's requirements is currently at the edge of feasibility, and leave studying hardware implications for future work.

3.5.2.3 Vector register file implementation

The biggest challenge when implementing a VRF is meeting the bandwidth requirements of the compute resources with minimal area- and power consumption. The bandwidth requirements are determined by the frequently-used multiply-accumulate (MAD) instruction, which reads three vector operands and writes back one vector operand. Each vector read and write can target as many columns as there are work-items in a warp.

Without data placement considerations, four-port 3R1W SRAM cells seem required to achieve a potential IPC of 1. Literature (e.g. Gebhart et al. [94]) suggest that commodity GPUs design their VRF around dual-ported 1R1W SRAM banks instead. Lindholm et al. [95] describe a technique that creates the illusion of multi-ported SRAMs using multiple 1R1W SRAM banks. For each active warp, a *collector unit* gathers the operands required for its current instruction. Operand requests are placed in per-SRAM-bank FIFOs. Once all operands for an instruction have been collected, the instruction and its operands are issued to the execute pipeline phase. This mechanism implicitly re-orders instructions from different warps such that each instruction is issued at the earliest moment its operands become available. To balance access to the SRAM banks in the common case, a register's bank number is determined by hashing the register number with the warp ID.

To motivate the use of SRAMs with fewer ports from the perspective of power consumption, Gebhart et al. [94] estimate using the GPUWattch power model [96] that the 1R1W register file of the NVIDIA GeForce GTX480, a Fermi-generation graphics card with a similar ratio of registers per SP-unit as Sim-D, makes up 13.4% of its total power consumption. Using McPat, Lim et al. [97] estimate that the dynamic power consumption of the register file in a similar GeForce GTX580 GPU is ∼7%. Using their power model, GPU-wide dynamic power consumption is estimated to increase by over 21% if 3R1W SRAMs were used in the same banking organisation as the current 1R1W SRAM configuration.

From this perspective, there is a strong incentive to design Sim-D's VRF around 1R1W SRAM banks. Unfortunately, using collector units is not practical for Sim-D as the freedom they permit in scheduling the warps of a work-group complicates static worst-case performance analysis. Instead, I evaluate two schemes that allow simple round-robin scheduling of warps for each vector instruction. Figure 3.5 depicts an example bank mapping of both schemes for warps of 128 work-items.

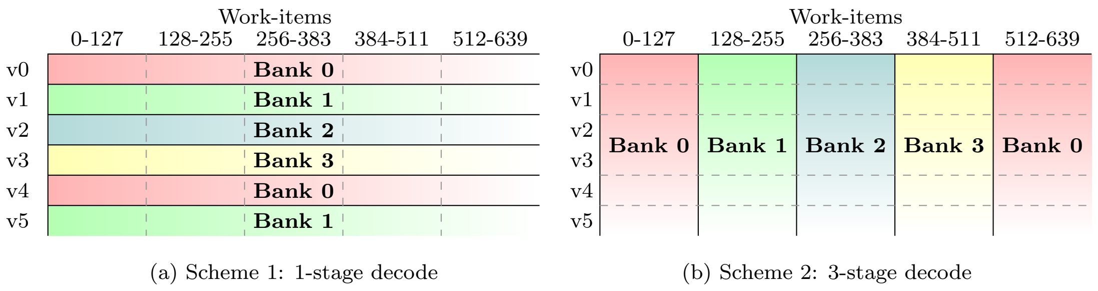


Figure 3.5: Example bank-to-VGPR mappings

Scheme 1 proposes a simple mapping from register number to bank, relying on the compiler's register assignment pass to avoid conflicts. The second scheme maps warps to banks and extends the operand fetch pipeline phase from 1 to 3 cycles, one per operand, to avoid bank conflicts. Figure 3.6 shows the resulting two pipelines.

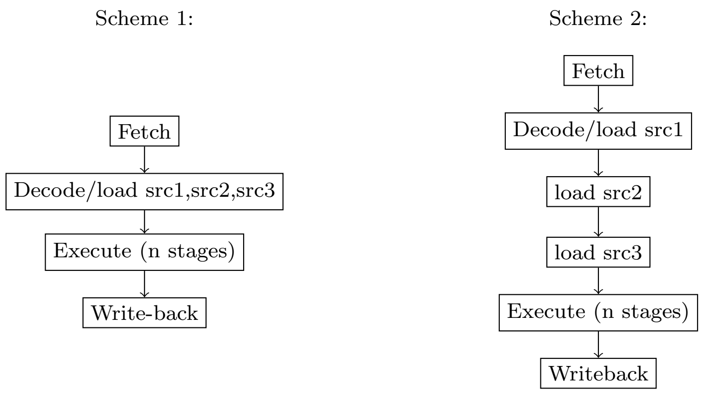


Figure 3.6: Pipeline stages for both VRF partitioning schemes.

Scheme 1: 1-stage decode Under scheme 1, the VRF is banked row-wise, mapping VGPRs to banks. The example depicted in Figure 3.5a maps a VGPR v_i to one of four banks $b = i \ \& \ 0x3$. Any configuration with three or more banks is able to satisfy the peak bandwidth requirements of three-operand vector operations.

Whether this bandwidth can be sustained at run-time depends on the compiler's ability to allocate hardware registers such that bank conflicts are avoided. If a conflict occurs, a pipeline bubble is inserted for every warp. These conflicts can be determined statically. Increasing the number of banks decreases the probability of bank conflicts occurring, but at the potential cost of reduced SRAM density.

Scheme 2: 3-stage decode Under scheme 2, registers are banked by warp as shown in Figure 3.5b. An example mapping with 4 banks would be $b = warp\_id \ \& \ 0x3$. For this scheme, the decode pipeline phase is split up in three stages. In every stage, one operand of an instruction is fetched. For a power-of-two number of banks and no warp trimming, round-robin scheduling of the warps in a work-group ensure that bank conflicts can never occur under this scheme.

There are two reasons that suggest that under this scheme it is best to match the number of banks with the number of warps in a work-group. Firstly, adjacent words transferred from DRAM or scratchpads in one cycle could be directed to disjoint columns of the same vector register. If multiple warps map to the same register bank, a worst-case transfer may require buffering of data elements before writing them back to the register file to deal with bank conflicts. Buffering would increase the complexity of the RF's DRAM/scratchpad interface.

A second reason for matching the number of banks with the number of warps is that it guarantees conflict-free pipelined operand fetch even when applying warp trimming on remainder work-groups. I deem this reason of minor importance, as the limit study in Section 6.4.2 suggests that the benefit from warp trimming is minimal.

Evaluation A thorough evaluation of both schemes requires me to quantify area and performance implications. Dynamic power consumption for read- and write operations is less relevant, as techniques like divided word-line (DWL) [98] or COMA [99] can help make power consumption independent of the SRAM bank's row width. These techniques allow writing only parts of an SRAM row. In Sim-D's case the parts of a row can be selected by a warp's implicit predicate mask. Using a technique like DWL, the dynamic power consumption of the VRF will vary minimally between the possible banking organisations, at an area overhead of 4%. As an added benefit, insertion of additional levels of restoring logic on the word-line (the wire from the last level of a row decoder into each SRAM cell) reduces its fan-out, in turn reducing parasitic capacitance originating from the SRAM cells [28]. This results in reduced latency and power consumption of the SRAM banks.

For area and latency bounds, I need details of a hardware implementation of the VRF under various banking schemes. Unfortunately, initial experiments with the CACTI 7.0 [100] power-, area- and latency model of SRAM-based storage structures have instilled little confidence in the accuracy of its results when generating oblong SRAM designs. Modelling an SRAM bank of 16 entries of width 2048b results in an SRAM density of just over 6%. I suspect this low density is caused by CACTI's fixed four-level hierarchy (bank, subbank, mat, subarray) and preference for a square organisation with H-tree interconnects being calibrated towards larger L2 and L3 caches rather than oblong VRF banks. A custom design could well yield better results, but creating such an implementation is beyond the scope of this work. As a general observation: given the VRF logically has many times more columns than rows, the banking strategy for scheme two has the benefit of producing less-oblong banks.

Performance wise, scheme 1 has the theoretical benefit of a shorter pipeline, reducing the pipeline warm-up and flush penalty as well as the probability and cost of RAW-hazard resolution. The performance comparison of the two register banking schemes in Section 6.2.2 indicates that for most benchmarks this performance effect is negligible.

3.5.2.4 n-vector load/stores

When handling structs of data, developers are encouraged to organise this data in memory as a struct-of-arrays. This way, neighbouring work-items request values which are stored consecutively in memory rather than spaced apart by the width of the struct, improving net DRAM data bus utilisation. If the application's requirements forbid such a data organisation, adding hardware support for n-vector data elements can significantly speed up data transfers from array-of-struct buffers.

Support for such transfers imposes additional constraints on the partitioning of the register file, as it must be able to write to the same column of up to n consecutive vector registers in a single cycle. This is likely to increase the area overhead of the register file. Furthermore, as this increases the data routing options, the number of outputs from the vector register distribution crossbar must be multiplied by n as well.

From the set of benchmarks I identified two use-cases for 2-vector loads: FFT and the SRAD reduce kernels. Furthermore, KFusion reads 3-vector arrays. Zooming in on the FFT benchmark, an alternative approach to yield good DRAM throughput from its array-of-structs buffer is to preload all structs into the scratchpad, and perform a series of indexed loads from there. In Section 6.4.1 I argue against the implementation of n-vector loads by contrasting the performance and hardware cost of these two approaches.

3.5.3 Mapped buffers: BufToPhysXlat

Each SimdCluster contains two BufToPhysXlat units per SimdCluster: one for DRAM buffers and one for scratchpad buffers. Each BufToPhysXlat component contains a mapping from a buffer ID number to a (physical address, x-dimension, y-dimension) 3-tuple. Their purpose is to provide a memory protection mechanism that isolates data between different kernel-instances, and to provide necessary parameters to load/store instructions. For example, the load/store unit uses the x-dimension of a buffer to determine the *period* of a 2D stride request. Stride requests are discussed in detail in Section 5.3.1.1. Buffer dimensions are also used to perform out-of-bounds checks on DRAM and scratchpad requests, providing memory protection across kernels.

Upon launching a kernel-instance, the workscheduler uploads all mappings to the two BufToPhysXlat units. These mappings persist throughout kernel execution.

The DRAM and scratchpad BufToPhysXlat units can be queried in parallel, simplifying the pipeline in the presence of DRAM↔scratchpad transfers. Querying a BufToPhysXlat entry is done in constant time, providing a latency benefit over page tables. This benefit comes at the cost of two hardware-imposed restrictions: a fixed limit on the number of buffer entries and a requirements that all buffers are contiguous in physical memory. Section 5.2 demonstrates that most kernels only map a limited number of buffers. Developers can opt to fuse buffers of similar x-dimensions if more buffers are required than can be mapped in a BufToPhysXlat unit. I expect that the requirement for contiguous buffers is an acceptable limitation to be paid by HRT systems.

3.5.4 Scoreboard

The scoreboard is used to mitigate read-after-write (RAW) hazards resulting from write operations pending in the pipeline. Write operations of an instruction are added to the scoreboard in the first cycle of the instruction decoder, and will be removed upon committing results in the write-back pipeline stage. When the instruction decoder issues an operand read operation for a register that matches a pending write operation in the scoreboard, the pipeline will stall until the write operation is complete.

Write-after-read (WAR) and write-after-write (WAW) hazards cannot occur in Sim-D's simple single-issue in-order pipeline, and thus do not need a hardware resolution mechanism.

Rather than storing write operations in a bit-mask of registers as is common in CPUs, Sim-D implements the scoreboard as a queue backed by a ring-buffer of CAMs. Two observations motivate this decision. Firstly, the single-issue in-order pipeline means that register write reservations are first-in-first-out. Secondly, the Sim-D pipeline contains significantly more registers to track than ordinary CPUs. Given n warps per work-group, the default Sim-D configuration must track $64 * n$ VGPRs, 32 SGPRs, $6 * n$ VSPs and 3 SSPs per work-group. For $n = 8$, this equates 1190 registers. If each register is represented as a bit in a bit-map, in any cycle the vast majority of these bits will be 0.

By contrast, for a pipeline of m stages between the first instruction decode stage and the write-back stage, a queue requires m entries for safe operation. For $m = 8$, this reduces the number of required storage bits from 1190 to $8 * \lceil log_2(1190) \rceil = 88$b. CAMs with three match-lines are used for storage so that IDecode can query three registers per cycle, satisfying the instruction throughput requirement of three register loads per cycle.

The order of operations in cycle n is as follows. First, a *commit* request from the write-back pipeline stage, generated in cycle $n - 1$, is processed by removing the oldest entry from the list. Second, a new write reservation, generated by the instruction decoder in cycle $n - 1$, is added to the queue. Finally, the instruction decoder's three operand queries for cycle n are checked. If one of the queries matches, the scoreboard issues a stall signal indicating which register is currently reserved. Scoreboard querying is performed in parallel with the register reads. If the scoreboard issues a stall-signal for one of the words, the data returned by the RF is undefined and must be discarded. The instruction decoder must repeat a request for the reserved word(s) every cycle until the stall signal is cleared.

3.5.4.1 Special cases

Although the general mechanism behind the scoreboard effectively detects RAW hazards, four cases require special care: false WAR-hazards, pipeline flushes, control stack operations and custom DRAM stride-requests that implicitly rely on special scalar registers. I next explain how Sim-D handles these special cases.

False WAR-hazards When Sim-D is configured with a three-stage instruction decoder, following scheme 2 from Section 3.5.2, false WAR-hazards could lead to a pipeline deadlock. To explain the problem, consider the following sequence of scalar instructions:

```
1 smad s0, s1, s0, s3
2 sadd s3, s0, s4
```

Listing 3.3: Code example exposing false WAR hazard.

This code example contains a RAW hazard on register s0. In cycle 0, the instruction decoder issues an operand fetch for s1, the first operand for the instruction in line 1, and reserves s0 for writing with the scoreboard. In cycle 1, the *sadd* instruction will enter the instruction decoder, causing it to reserve s3 for writing in the scoreboard and attempt to fetch its first operand s0. This fetch must be stalled until the *smad* operation reaches write-back.

There are two problems in this program that each potentially cause a deadlock in the pipeline. Firstly, the *smad* operation's second operand s0 is marked on the scoreboard as a write target by the very same instruction. Without a mechanism to convey to the scoreboard that this operand should indeed be read before write-back, the scoreboard will flag a false RAW-hazard in cycle 1 preventing this instruction from advancing through the pipeline.

A second problem occurs in cycle 2. In the case that both instructions did proceed to the next pipeline stage, in cycle 1 the *sadd* instruction would have reserved s3 for writing in the scoreboard. In cycle 2, the *smad* operation would request its third operand, s3. The scoreboard now indicates a false WAR-hazard on s3 by virtue of the reservation made in cycle 1. The pipeline deadlocks as *sadd* will never execute before *smad.* The expected behaviour would instead be for the *smad* operation to read the old operand value.

To mitigate these two problems, the scoreboard keeps track of the currently active ring buffer entries in two bit-mask, one per work-group. These bitmaps are shared with the instruction decoder through an output port called *entries_pop*. Additionally, for each read operand query, the scoreboard allows the instruction decoder to specify which ring buffer entries to match against by providing a bit-mask of equal length. The scoreboard performs a boolean AND of this bit-mask with it's own *entries_pop* bit-mask when determining which content-addressable memorys (CAMs) to match an operand against to avoid querying inactive ring buffer entries.

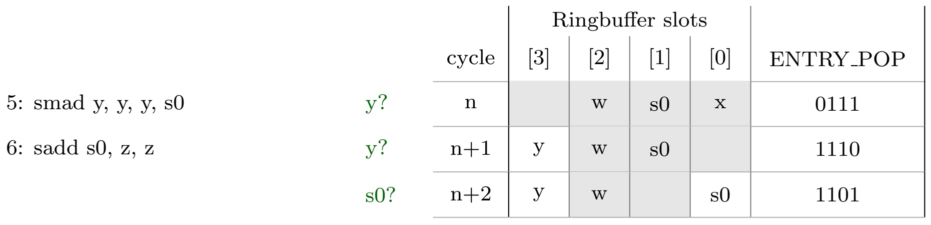
The three-stage instruction decoder is modified as follows. Instructions progressing down the pipeline are accompanied with a *req_sb_pop* register. Upon arrival of an instruction in the instruction decoder's first pipeline stage, this register is initialised to all-1. The first operand is now queried with this (all-1) register value as its bit-mask. On every cycle that this instruction remains in the decoder's pipeline stages, it updates its *req_sb_pop* value by performing a boolean AND with the scoreboard's *entries_pop* value. This disables ring buffer entries that were just removed from the queue or were otherwise inactive, without ever (re-)enabling entries that may have been added later. For each operand request, the instruction decoder provides the updated *req_sb_pop* to the scoreboard.

The one-stage instruction decoder can strap each *reg_sb_pop* mask to all-1 without consequences.

To demonstrate how and why this solution works, consider the following example program snippet. Irrelevant registers are denoted with w,x,y,z.

```
1 [...]
2 sadd x, x, x
3 smad s0, x, x, x
4 sadd w, w, w
5 smad y, x, x, s0
6 sadd s0, x, x
```

Assume this code runs on an architecture with a 3-stage instruction decoder, a single execute stage, and a 4-entry scoreboard ringbuffer. The state of the scoreboard and the requests issued by the instruction in line 5 is visualised in in Figure 3.7.



			Ringbuffer slots				
		cycle	[3]	[2]	[1]	[0]	ENTRY_POP
5: smad y, y, y, s0	y?	n	▨	w	s0	x	0111
6: sadd s0, z, z	y?	n+1	y	w	s0	▨	1110
	s0?	n+2	y	w	▨	s0	1101

Figure 3.7: Example execution with 4-entry scoreboard

The table on the right of Figure 3.7 shows the contents of the scoreboard and the ENTRY_POP mask for the corresponding work-group for cycles n to n+2. Grey columns denote the *smad* instruction's *req_sb_pop* register value. At cycle n, instruction 5 from the example program ("smad y, y, y, s0") enters the instruction decoder. As derived from the population in the scoreboard at cycle n, this instruction will flow through the instruction decoder with an all-1 initial entry_pop.

At cycle n+1, the instruction from line 2 has finished execution and its scoreboard entry 0 is removed accordingly. Additionally, the target for the *smad* in line 5 is added to column 3. The instruction decoder updates the *req_sb_pop* with the *entry_pop* value from the previous cycle, resulting in a mask that excludes its own write. In the same cycle n+1 the *sadd* instruction from line 6 enters the instruction decoder. Its first operand does not appear in any of the scoreboard entries, and neither does the second operand of the *smad* instruction, hence the pipeline will not stall.

In cycle n+2, the *sadd* instructions destination operand s0 is enqueued on the scoreboard. The *req_sb_pop* value for *smad* (0111) is ANDed with the *entry_pop* value of the previous cycle (1110) to create the new value 0110. Note that this mask prevents matching its third operand, s0, against the scoreboard entry in slot 0. As a result, no false RAW hazard is flagged and execution will continue as expected.

Pipeline flushes When a control flow operation commits its new PC and triggers a pipeline flush, the scoreboard contains write reservations for instructions that will not commit. To avoid stalling unnecessarily on these now-obsolete reservations during pipeline warm-up, IExecute instructs the scoreboard to clear the *entry_pop* bit-mask for the corresponding work-group. This disables matching against all ring buffer entries. As the now-dead instructions pass through the write-back stage of the pipeline in subsequent cycles, entries are dequeued from the ring-buffer to free up space for new reservations.

Control stack operations CSTACK pops can conflict with CSTACK pushes as reads from the CSTACK are performed in the first IExecute pipeline stage while writes commit in the write-back stage. To avoid CSTACK hazards, the scoreboard uses a per work-group counter to track the number of CSTACK operations in the pipeline. *cpop* operations stall in the instruction decoder as long as the counter is non-zero. The maximum value of this counter is determined by the number of CSTACK instructions that can be in the pipeline at any given time. Since CSTACK operations are vector operations, this maximum value is 2 for a pipeline distance between the first instruction decode stage and the final execute stage of 8 and 8 warps per work-group. Each counter thus adds two bits of storage.

Special scalar register dependencies Some memory operations depend on the values of the three SSP *stride descriptor* registers: *sc.sd_words, sc.sd_period and sc.sd_period_cnt*. These instructions do not load these registers explicitly as instruction operands, but rather they are hard-wired to the load/store unit. As such, the instruction decoder does not query write dependencies on these registers through the regular query ports. To correctly eliminate RAW hazards for these instructions, an extra match-line is added to each CAM that tests whether the register is a special scalar register. The instruction decoder can use the dedicated *ssp_match* signal to query a write reservation for any such register, and stall if the Scoreboard reports a match for the given work-group. No extra storage is required for matching of these special scalar registers. Since these stride-descriptor registers are the only writeable SSPs, a single query bit suffices.

3.6 Summary

In this chapter I presented the Sim-D wide-SIMD parallel architecture for HRT systems. Sim-D performs strip-mining in hardware, making its execution model similar to that of a GPU. It features a four-phase in-order pipeline of configurable length, supporting a mix of scalar and vector instructions. Scalar instructions are used to issue large work-group-wide DRAM requests, which I explain in Chapter 5.

Two key features make this architecture suitable for HRT systems: performance isolation and double-buffered execution of work-groups. Together this permits a coarse grain interleaving of the compute- and data access phases of work-groups, with predictable execution times on each phase. This performance-isolation permits safe and tight WCET analysis, as will be explained and evaluated in Chapter 7.

CHAPTER 4

Experimental set-up

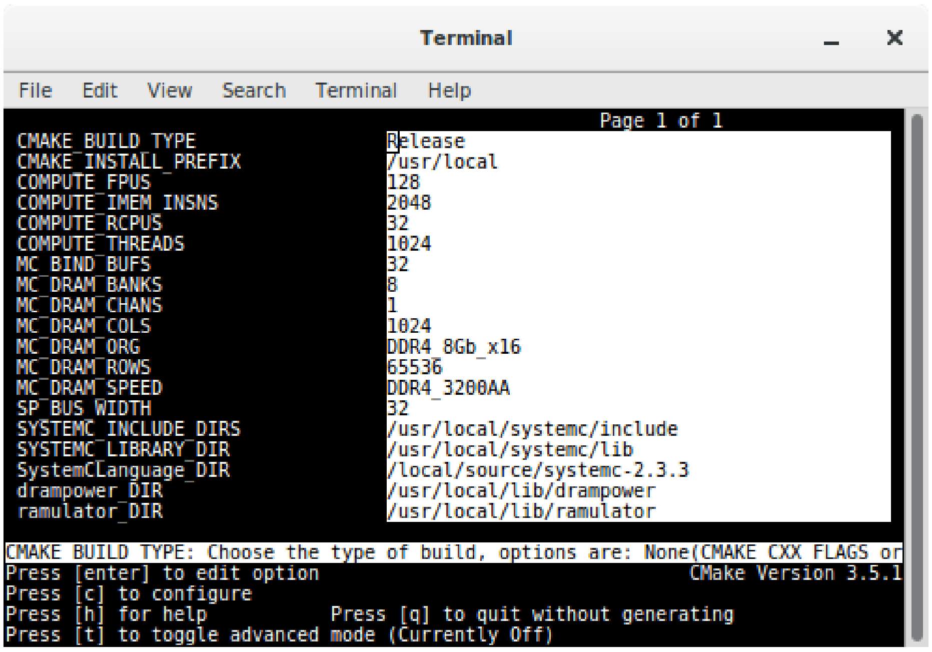
This chapter explains the components that comprise the experimental set-up used for feasibility-, performance- and design space studies presented in the remainder of this thesis. This chapter provides details on the following:

- The cycle-accurate "Sim-D" simulator and its DRAM configuration (Section 4.1),
- The software set-up and NVIDIA hardware used for data extraction and performance comparisons (Section 4.2),
- The set of benchmarks used for evaluation (Section 4.3),
- A discussion on the limitations of experiments carried out with these components (Section 4.4).

4.1 Cycle-accurate simulation

To perform cycle-accurate measurements on the architecture presented in Chapter 3, I implemented a simulator in SystemC [79] comprising both the compute resources and the memory controller. This platform was chosen for simulation performance, ease of development and easy integration with third party simulation models written in C or C++. As the Sim-D's data path and mixed vector-scalar ISA differ significantly from conventional GPU architectures, I decided against re-using existing simulation models like GPGPU-Sim [101] or Multi2sim [102].

The Sim-D simulator integrates two external projects: Ramulator [81] and DRAM-Power [82]. These projects provide a DDR4 DRAM timing model and a power estimation model for streams of DRAM commands respectively. I have extended both projects with matching timing- and power models for DDR4-1866M and DDR4-3200AA memory, the latter for which parameters are derived from the Micron MT40A512M16 datasheet [103].



```
                                       Page 1 of 1
 CMAKE_BUILD_TYPE                 Release
 CMAKE_INSTALL_PREFIX             /usr/local
 COMPUTE_FPUS                     128
 COMPUTE_IMEM_INSNS               2048
 COMPUTE_RCPUS                    32
 COMPUTE_THREADS                  1024
 MC_BIND_BUFS                     32
 MC_DRAM_BANKS                    8
 MC_DRAM_CHANS                    1
 MC_DRAM_COLS                     1024
 MC_DRAM_ORG                      DDR4_8Gb_x16
 MC_DRAM_ROWS                     65536
 MC_DRAM_SPEED                    DDR4_3200AA
 SP_BUS_WIDTH                     32
 SYSTEMC_INCLUDE_DIRS             /usr/local/systemc/include
 SYSTEMC_LIBRARY_DIR              /usr/local/systemc/lib
 SystemCLanguage_DIR              /local/source/systemc-2.3.3
 drampower_DIR                    /usr/local/lib/drampower
 ramulator_DIR                    /usr/local/lib/ramulator

CMAKE_BUILD_TYPE: Choose the type of build, options are: None(CMAKE_CXX_FLAGS or
Press [enter] to edit option                                 CMake Version 3.5.1
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Figure 4.1: CCMake displaying Sim-D compile-time configuration options.

Sim-D's high configurability enables easy design space exploration. Compile-time options (displayed in Figure 4.1) include the number of single-precision units, work-items per work-groups, size of the instruction memory, maximum number of bound buffers, DRAM configuration and scratchpad data bus width. These options must be chosen at compile time because they affect widths of internal signals, which in SystemC are chosen through C++'s object template values. C++ template values must be known statically to the compiler. At run-time, the pipeline depth of the decode and execute phases can be configured.

For a typical benchmark, the Sim-D simulator simulates in excess of 45,000 cycles per second on a mid-range desktop system from 2013 containing an Intel Core i5-4670 (3.40GHz) CPU.

4.1.1 DRAM configuration

For all Sim-D experiments, the DRAM data bus is configured to be 64 bits wide. For evaluating the full Sim-D processor, this bus width is achieved by assuming a rank of either four parallel DRAM chips with a 16-bit data bus, or 8 chips with an 8-bit data bus each. A full overview of both DDR4-3200 DRAM configurations and their timings is provided in Table 4.1.

Micron DDR4-3200AA			
Symbol	(1) 4*16b	(2) 8*8b	Description
f_c	1600MHz, 1T		Cmd bus frequency, cycles per command
f_d	3200MHz		Data bus frequency
nCh	1		Channels
nRa	1		Ranks
	64-bit		Effective bus width
nB	8	16	Banks / Rank
nBG	2	4	Bank groups / Rank
nR	65536		Rows / Bank
nC	1024		Columns / Row
nBW	16		Words/burst (8 beats, 64B)
Latency in cycles, $tCK = 0.625ns$			
$tRCD$	22		Row-activate to CAS delay
$tCAS$	22		Column access strobe, RD $\rightarrow$ first burst distance
$tCWD$	16		Column write delay, WR $\rightarrow$ first burst distance
tRP	22		Row Precharge delay
$tRPRE$	1		Read preamble
$tWPRE$	1		Write preamble
$tBURST$	4		Cycles per burst (for DDR: beats / 2)
$tRAS$	52		Row Activate Strobe, min. ACT $\rightarrow$ PRE distance
$tRTP$	12		Read-to-Precharge
tWR	24		Write Recovery
$tFAW$	48	34	Four activate (sliding) window
$tRFC$	560		Refresh cycle (act, pre) time
$tREFI$	12480		Refresh interval
$tCCD_S$	4		Column R/W to CAS delay, different bank group
$tCCD_L$	8		Column R/W to CAS delay, same bank group
$tRRD_S$	9	4	Row-activate to Row-activate delay, different bank group
$tRRD_L$	11	8	Row-activate to Row-activate delay, same bank group
$tWTR_S$	4		Write-to-Read, different bank group
$tWTR_L$	12		Write-to-Read, same bank group

Table 4.1: Configuration and timing properties of Micron (1) MT40A512M16JY-062E and (2) MT40A1G8SA-062E DDR4-3200AA.

4.2 Real-world comparison and OpenCL data acquisition

All experiments are run on an Intel Core i5-4670 (3.40GHz) based mid-range desktop computer from 2013. This system runs Ubuntu 16.04 with Linux kernel 4.4 LTS. Several NVIDIA graphics cards from the Kepler generation were used for static data acquisition and performance measurements. For these cards, the NVIDIA GeForce driver version 367.27 was installed.

As a reference point for average case performance, I run OpenCL benchmarks on an NVIDIA GeForce GT710 graphics card. This graphics card is of similar specifications to the NVIDIA Tegra K1 embedded GPU. Like Sim-D, this card contains a 64-bit DRAM bus. For comparative experiments, Sim-D and the GeForce GT710 card are configured to match as closely as possible. To this end, Sim-D is configured with the DDR4-1866M DRAM timing profile. Furthermore, the graphics card is modestly overclocked: the compute clock is increased from 954MHz to 1GHz and the DRAM clock from 1800MHz to 1866MHz.

To guide Sim-D's design space exploration, I analyse static program properties like register usage, instruction mix and NDRange parameters from OpenCL kernel-instances running on an NVIDIA Kepler GTX 650 graphics card. Data is extracted from this system using two tools: *valgrind-mmt* [104] and *demmt* [83]. *Valgrind-mmt* is an extension to the Valgrind tool that adds capabilities to intercept communication between NVIDIA's user-space device driver and its kernel module. This communication contains data not otherwise exposed to users or developers, such as command buffers, launch parameters and assembled program binaries. Note that these binaries contains more low-level information that exposed through NVIDIA's public PTX intermediate representation (IR) [105]. Architecture-specific optimisation- and register allocation passes have been executed, resulting in more detailed information about instruction mix and register usage.

Demmt is capable of decoding and displaying the trace output of *valgrind-mmt* in a human-readable form. *Demmt*'s output contains kernel launch parameters and disassembled kernel binaries in NVIDIA's architecture-specific ISA.

An initial evaluation of GPGPU-Sim [101] led me to believe that its simulation results for embedded-grade GPUs are not representative for real world performance. When simulating a GPU resembling the NVIDIA GeForce GT710, for at least one of the CNN benchmarks I observed a simulated execution time more than double the measured execution time on actual hardware. For this reason I decided against using GPGPU-Sim for quantitative experiments.

4.3 Benchmarks

With Sim-D's focus on hard real-time systems, I selected a set of benchmarks representative for the expected workloads on safety-critical systems. Their use-cases include (autonomous) robotics and medical imaging, covering massively-parallel algorithms in the classes of neural networks, computer vision, DSP and generic linear algebra.

Where available I used existing kernels from KinectFusion [106] and SPEC Accel [107], the latter being a benchmark suite derived from the Rodinia [108] and Parboil [109] suites. Unfortunately, public neural network benchmarks use the proprietary cuDNN library for acceleration, obscuring the mapping between GPGPU kernel code and the generated assembly. For this reason, I additionally ported and optimised common convolutional neural networks (CNNs) operations from Dr. Bates' C implementations [110] to OpenCL.

Application domain	Benchmark	Source	Kernel	Sim-D port
AI	CNN	Dr. Bates	convolution	√
			relu	√
			maxpool	√
Computer vision	KinectFusion	SLAMBench	halfSampleRobustImage	√
			depth2vertex	√
			vertex2normal	√
			track	√
Healthcare	SRAD	SPEC	srad	√
			srad2	√
			reduce	√
			reduce_fpatom	
	MRI-Q	Parboil/SPEC	ComputePiMag	√
			ComputeQ	√
Sparse matrix	SPMV	Parboil/SPEC	spmv_jds_naive	√
Dense matrix	LU Decomp.	Rodinia/SPEC	diagonal	
			perimeter	
			internal	
Partial Diff. Eq.	Stencil	Parboil/SPEC	naive_kernel	√
FFT	FFT	Parboil/SPEC	GPU_FFT_Global	√

Table 4.2: List of selected benchmarks and kernels.

All selected benchmarks are listed in Table 4.2. Kernels with a matching implementation in Sim-D are marked in the final column. The remainder of this section describes the benchmarks in greater detail and justifies my modifications to some of the benchmarks.

4.3.1 CNN

The CNN benchmarks implement the operations for three types of layers: convolution, RELU and max-pooling. Each kernel performs dense-matrix operations on large 2D- or 3D inputs. Input sizes and parameters for *max-pooling* and *convolution* are taken from the conv1 and maxpool1 layers in Squeezenet [111]. *RELU* dimensions were taken from Dr. Bates' original kernel [110], which are deliberately small to aid architecture simulation. As none of these benchmarks exhibit data-dependent performance variance, input data is randomly generated.

4.3.2 KinectFusion

KinectFusion [106] implements a set of algorithms designed to map surroundings from data gathered using a moving Microsoft Kinect camera. Central to this benchmark is the *iterative closest point* algorithm which performs simultaneous localisation and mapping (SLAM) on a succession of frames. The KinectFusion kernels used in this work are those implemented in SLAMBench [112].

The closest point algorithm works by estimating and iteratively refining a motion vector that describes the movement between two frames. The accuracy of a candidate vector is assessed by calculating the per-pixel error between frame n and a transformation of the frame $n+1$ by this vector. Edges of this transformation are extrapolated from the edges of frame $n+1$.

I modified the halfSampleRobustImage kernel to obtain a more realistic kernel binary. This kernel contains a nested loop that is unrolled by NVIDIA's driver. Unfortunately, the inner loop is unrolled with a factor larger than the number of iterations this loop takes in practice. Manually providing the true iteration count using the *#pragma unroll* annotation reduces the binary size with no measurable effects on performance.

4.3.3 SRAD

The Speckle Reducing Anisotropic Diffusion [113] benchmark implement a filtering technique to remove locally correlated noise from an image while preserving edges of features. Of the six kernels that constitute this benchmark, I selected the three kernels performing non-trivial work: *SRAD*, *SRAD2* and *SRAD reduce*

The *SRAD reduce* kernel uses a binary-tree reduction to compute the sum of a list of numbers in $O(n*log(n))$ operations. Although this kernel performs well, it makes extensive use of work-group barriers to guarantee consistency. Without diverging from the OpenCL standard, a binary-tree reduction is the most efficient way to perform this summation. A kernel performing this task with $O(n)$ operations can be constructed using floating point atomic operations. Such a kernel eliminates both all work-group synchronisation points, and launching of subsequent kernels that aggregate the per-work-group results. The OpenCL standard does not support floating point atomic addition operations, but such operations can be used on NVIDIA hardware using in-line assembly. To generate comparison data on NVIDIA hardware for both approaches, I developed the non-portable "reduce_fpatom" kernel using in-line assembly.

The *SRAD reduce* kernel contains lots of duplicate code to deal with various edge cases. Generally work-items in the same warp all tend to follow the same code path, so despite the complex control structure there are no divergent work-items at run-time. To report more meaningful numbers on binary size and instruction mix, I have deduplicated this code. Despite a marked reduction in control flow around the main loop(s), this has no measurable effect on execution time as this kernel's inner loop is fully unrolled by NVIDIA's OpenCL compiler.

The *srad* and *srad2* kernels perform complex filter operations on image data. I applied two optimisations to both kernels. Firstly, many literals in the original kernel omitted the *f* modifier and thus were provided as doubles. I have corrected these to single precision floating point values, removing the overhead of run-time conversion to single-precision values. Secondly, I replaced the open-coded clamp operation with the OpenCL-provided clamp() function. Together these optimisations result in a reduction of code size of approximately $20 - 24\%$, and a $\sim$5% improvement in run time.

4.3.4 MRI-Q

The MRI-Q benchmark [109] contains a subset of a program that reconstructs an image from non-Cartesian data obtained from MRI equipment. Input data is provided as a collection of complex numbers. As such, this benchmark requires trigonometric operations and is strongly compute-bound.

The main loop of the *computeQ* kernel as shipped with SPEC Accel is manually unrolled by 2. Experimenting with other unroll factors shows that 2 provides the best trade-off between binary size and performance on NVIDIA hardware. As such, I maintain the original structure of the kernel and refrain from transforming the loop to facilitate compiler unrolling.

In this kernel I replaced calls to sin() and cos() with native_sin() and native_cos(). OpenCL's precision requirements on sin() and cos() force NVIDIA's compiler to emit software-emulated versions of these operations. This increases the binary size of each kernel with 111 instructions for each sine or cosine operation when compared to using OpenCL's native_sin() and native_cos() functions. I believe that this lowering code constitutes a loss of information, as it obscures the use of trigonometric operations in kernels, and thus opt to replace these calls with their native counterparts. I observe no notable differences in the quality of output data generated by this kernel.

4.3.5 SPMV

The SPMV benchmark performs a matrix-vector multiplication on a jagged diagonal storage (JDS)-compressed sparse matrix [114]. Decompression of the sparse matrix relies heavily on indirect (indexed) data loads.

4.3.6 LU Decomposition

The LUD benchmark computes a set of triangular matrices from a dense matrix. The elimination of values from a square matrix to form a triangular matrix is performed using operations on entire rows and columns.

These row-wise and column-wise operations limit the opportunity for parallelism. Although the dimensions of the input matrix can be chosen as a parameter, the kernel-instances' NDRanges are small by GPU-compute standards. The *lud_diagonal* benchmark is launched with 16 work-items. The run-time of each work-item varies greatly as a result of different loop iteration bounds. The *lud_perimeter* kernel splits a matrix into blocks of 16 rows, launching $\lceil \frac{rows}{16} \rceil$ work-groups with 16 work-items each. This sparseness of work-groups leads to very poor utilisation of GPU resources. The *lud_internal* kernel is the only of the three that effectively utilises the GPU's parallel execution, launching one work-item per output word.

For this reason I deem the LUD benchmark as non-representative for a workload targeting massively parallel accelerators. I report data obtained from the OpenCL implementation, but omit porting this benchmark to the Sim-D simulator.

4.3.7 Stencil

The stencil benchmarks performs a filter operation for each point in a 3D grid using values from its direct neighbours, trimming edge values. Using separate input- and output buffers, this benchmark is trivially parallelised by mapping each work-item to a point in the output grid.

4.3.8 FFT

Fast-Fourier transformations are commonplace in many signal processing workloads, transforming discrete values into the frequency domain or vice versa. This benchmark performs an fast-fourier transform (FFT) operation on a (real-valued) data set with a window size of 256. Each work-item processes two data elements, meaning 128 work-items per window are launched. The mapping from input values to work-items is linear when considering adjacent (r,i)-pairs as 4-vector values. Write-back of each iteration follows a butterfly distribution pattern, providing a particular challenge to a data path designed to hard real-time requirements.

I made two modifications to this benchmark that improve the performance and binary size on NVIDIA hardware. Firstly, the original FFT benchmark unintentionally performs a double-precision floating point division. This is the result of using OpenCL's M_PI macro rather than the M_PI_F macro to represent the value π. For my experiments, the benchmark instead uses the 32-bit single precision macro. Secondly, like in the MRI-Q benchmark, I adjusted the FFT benchmark to use the native_sin() and native_cos() functions provided by hardware rather than the software emulated sin() and cos(). Both alterations lead to a more realistic discussion on binary size and instruction mix, with a modest speed improvement.

4.4 Limitations

Existing CUDA and OpenCL benchmarks have been large designed and optimised towards contemporary GPUs. Inevitably, this has created a feedback loop in which the design choices made by the benchmark developer are guided by the performance characteristics of the targeted devices. As an unintentional consequence, any data extracted from these benchmarks comes with the risk of predisposing conclusions and hardware design decisions to mirror existing hardware.

That being said, within the scope of data-parallel processing there have been a lot of common optimisation techniques shared between GPUs, DSPs, tiled multiprocessors (e.g. Loki [115]) and packed-SIMD CPU extensions. Such techniques include loop unrolling (e.g. [116]) and data placement strategies for contiguous memory access (e.g. [117, 118]). The similarity of optimisation techniques for different types of platforms hints at the existence of good data-parallel programming practice that transcends wide-SIMD accelerator architectures.

The real-world data acquired in this work must be read in this context: they are useful as a guidance in design space exploration, but should not be interpreted as generic universal truths on the best hardware-software combination for accelerating a given application. Stronger evidence-based claims around hardware- and software design risk making a flawed circular argument.

4.5 Summary

In this chapter I presented the configurable, cycle-accurate Sim-D simulator developed in SystemC. I present 12 benchmarks taken from KFusion, Rodinia and Parboil, plus three CNN kernels developed by my colleague Dr. Bates and myself. These benchmarks will be used throughout this document for evaluation purposes. To provide reference points and to extract static information about benchmarks, NVIDIA's GeForce GT710, GTX650 and GTX780 Ti are used.

CHAPTER 5

A hard real-time DRAM controller

A key design priority of contemporary GPUs is maximising memory throughput. It is generally assumed that GPGPU kernels implement computationally simple data-parallel algorithms, and that synchronisation between work-items in a kernel-instance is rarely required. As a result many kernels end up being I/O-bound.

The large data sets processed by GPGPU kernels render DRAM the only practical memory technology. To facilitate high throughput DRAM transfers, memory controllers in CPUs and GPUs perform optimisations that maximise parallel execution of DRAM commands on different DRAM banks, ranks and channels. These optimisations minimise the overhead of expensive activate- and precharge operations on the critical path by re-ordering requests. An unfortunate downside of request re-ordering is that it makes the response time of individual requests difficult, if not impossible to predict. In the context of hard real-time (HRT) systems this is undesirable as pessimistic bounds on DRAM requests result in pessimistic bounds on program execution time.

Meanwhile, HRT DRAM controllers researched in the past have not kept up with the development of DRAM. These DRAM controllers suffer from diminishing utilisation of the data bus with each successive generation of DRAM, as a result of both the increased latencies between issuing and finishing DRAM commands [39], and of wider DRAM buses reducing the potential for bank parallelism within a request [49].

Data locality properties of GPGPU programs provide interesting opportunities to increase DRAM bank parallelism in a deterministic manner. To exploit these opportunities, Sim-D's DRAM controller parts with prior work by supporting large DRAM requests. These large requests *explicitly coalesce* the requests of individual work-items into a single request for an entire work-group. Deterministically re-ordering DRAM commands *within* large requests often aids in effectively exploiting bank parallelism, while the closed-page policy *between* requests helps to provide tight bounds on the request as a whole.

This chapter makes the following contributions:

- An analysis of the limitations of pattern-based DRAM controllers in the context of SIMD processing of GPGPU programs (Section 5.1),
- An analysis of buffer usage in selected benchmarks, to characterise the types of big transfers that are encountered in GPGPU programs (Section 5.2),
- A DRAM controller design that accepts large, explicitly coalesced DRAM transfers capable of facilitating the needs of the benchmarks investigated (Section 5.3),
- Methods for deriving the longest issue delay (LID) and worst-case request execution time (WCRET) of transfers processed by Sim-D's DRAM controller (Section 5.4),
- An evaluation of the performance of DRAM transfers issued to Sim-D's DRAM controller configured with DDR4-3200AA DRAM (Section 5.5).

5.1 Limitations of pattern-based hard real-time HRT DRAM controllers

As discussed in Section 2.3.1, prior work on HRT DRAM controllers has mainly focussed on application in multi-core CPUs. The Sim-D architecture differs from a multi-core CPU in three crucial ways. Firstly, kernel-instances share the Sim-D processor temporally and non-preemptively. This means that all requests in a DRAM controller's request queue must originate from the same kernel-instance. Secondly, the absence of associative caches removes the requirement to align requests to cache-lines. Sim-D's requests only need to be aligned to 32-bit word boundaries. Finally, the data parallel nature of its applications permits larger, coalesced DRAM transfers.

In the context of these differences, I previously dismissed the idea of basing Sim-D's DRAM controller around existing open-page partitioned DRAM concepts. In this section I further justify my work on Sim-D's novel coalesced large-request DRAM controller design. To this end, I first demonstrate why a closed-page policy loses efficiency with each successive generation of DRAM, and how this effect is mitigated by issuing larger requests. Following from this observation, I demonstrate how the performance of existing pattern-based DRAM controllers configured for large requests is conditional on a request's exact size and data alignment. I argue that these conditions are too stringent for many GPGPU workloads, for example for implementations of common filter operations.

5.1.1 Inefficiency on modern DRAM

Akesson et al. [39] explained how closed-page DRAM controllers' performance is limited by the mandatory latencies between successive DRAM commands. In terms of (nano-)seconds these latencies have barely changed for successive DRAM generations. As a consequence, as clock frequencies increase, the latencies between commands in terms of number of clock cycles increase while the amount of data transferred by each command remains equal. This in turn results in diminished data bus utilisation.

Goossens et al. [119] demonstrate the negative impact these higher latencies have on the performance of pattern-based DRAM controllers with successive generations of DDR2 and DDR3 DRAM. Figure 5.1 shows the results of repeating this experiment for a wider range of DRAM generations, ranging from DDR2 at 400MHz to DDR4 running at 3200MHz. The reported latencies of the DRAM read pattern transactions are a function of DRAM timing parameters and number of bursts per transfer. Latencies are generated using Goossens' et al. [48] ILP-based method for generating DRAM patterns. When a burst size allows for multiple DRAM patterns as a result of multiple (#banks, burst/bank) configurations, the most optimal pattern is chosen. For transfers of more than 64 bursts, the ILP-method, backed by CPLEX, is unable to generate a command sequence within reasonable time. However, analysis of the patterns revealed that the number of cycles for n bursts can be accurately modelled by $4n + tRCD - 3 + (tRRD_s - tCCD_s)^+$.

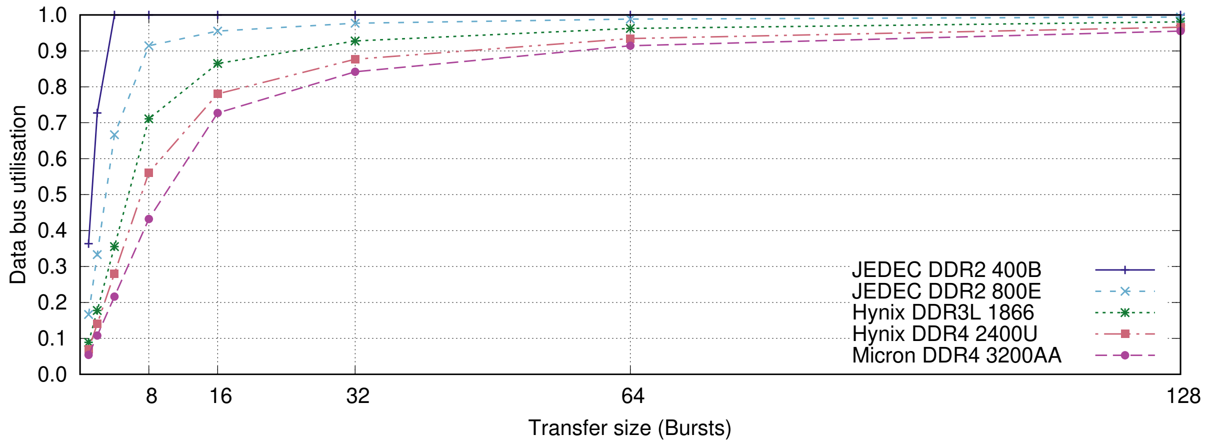


Figure 5.1: Data bus utilisation for read commands on successive generations of DDR DRAM

.

In line with Akesson et al.'s observations [39], Figure 5.1 demonstrates that where for DDR2 running at 400MHz, a 100% DRAM bus utilisation can be achieved with patterns of four bursts (one to each bank), on the latest generation of DRAM the same configuration achieves a mere 28% bus utilisation. To achieve 80% utilisation on DDR4-3200AA DRAM using memory-pattern based DRAM controllers, patterns must issue at least 32 bursts.

To make matters worse, reasoning about data bus utilisation degradation as a function of the number of bursts obscures the true scale of the problem. Krishnapillai et al. [49] observe that practical applicability of closed-page DRAM controllers degrades further as DRAM buses get wider. Where in the days of DDR2, a 16-bit DRAM bus was commodity, today most SoCs feature a 64-bit DRAM bus. This means that where previously a memory controller would issue four burst requests to read or write back a 64B cache line, today this has been reduced to only a single burst.

On the upside, the problem of regressing data bus utilisation with each generation of DRAM can be mitigated by issuing larger requests. Larger requests allow for more bank-parallelism, reducing the time spent on activate- and precharge command execution [39]. To illustrate the efficacy of larger requests: a 2KiB memory pattern is capable of achieving 80% utilisation on a 64-bit wide DDR4-3200AA DRAM bus.

Contrary to CPUs whose request size is generally dictated by the size of a cache line (typically 64B), Sim-D can coalesce DRAM requests at the granularity of a work-group, resulting in requests of sizes that permit closed-page DRAM controllers to achieve high data bus utilisation. However, I believe there are two mismatches between Sim-D's requirements and the performance characteristics of the pattern-based DRAM controllers researched in prior work: their narrow set of transfer sizes for which a single configuration can achieve good bus utilisation, and their stricter alignment constraints.

5.1.2 Variation in transfer sizes

The command patterns generated for pattern-based DRAM controllers assume transferring a power-of-two number of bursts. If not, transfers of addresses near the end of a row will require more activate commands than transfers addressing the start of a row, causing divergence in command scripts and their timing.

To give a concrete example, consider a DRAM configuration with each row containing 1024 columns of data, corresponding with 128 burst lengths. A pattern that requests three bursts worth of data from the start of a row would issue bursts for columns 0-7, 8-15 and 16-23. This transfer requires a single row activation. However, if the request targets the end of the row a pattern transaction must issue bursts for columns 1008-1015 and 1016-1023 in one row, and a burst for columns 0-7 in the next row. A read of the same size now requires twice as many row activations. Enforcing a constraint limiting patterns to powers-of-two number of bursts ensures that a pattern transaction always reads from the same number of rows, leading to reduced pessimism when determining a pattern's WCET.

Having the configured pattern fixed to a power-of-two number of bursts has implications for requests of a size that is not an exact multiple of a pattern transaction size. Figure 5.2 shows the correlation between request size and data bus utilisation on DRAM controllers configured with patterns of 1, 2, 4 8, 16, 32 and 64 bursts, requesting between 64B and 8KiB per pattern transaction.

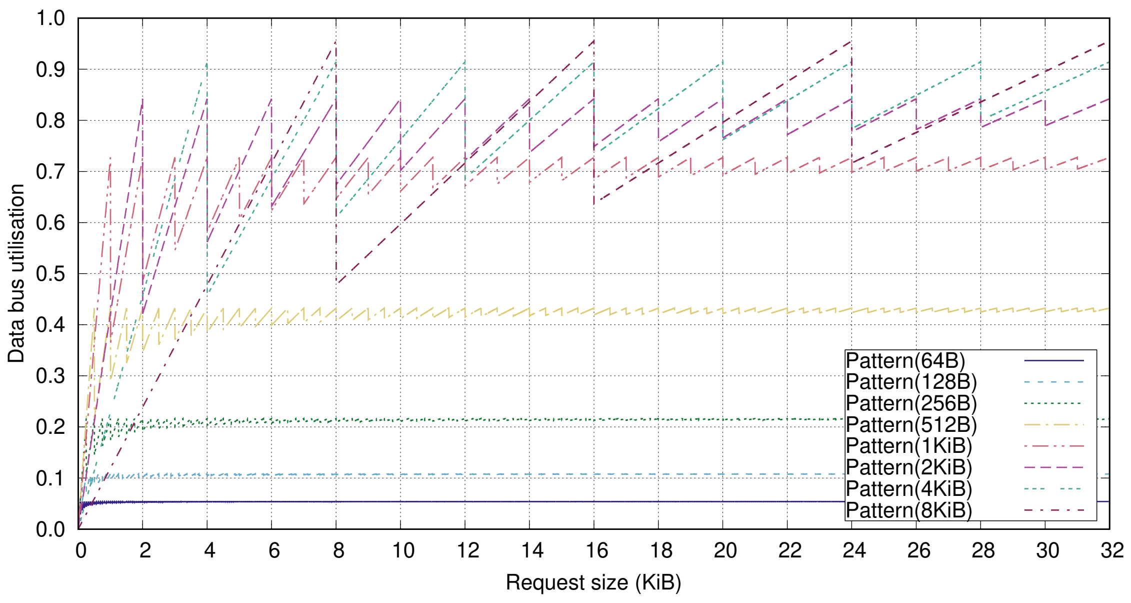


Figure 5.2: Data bus utilisation for unaligned transfers processed by a pattern-based DRAM controllers configured with different-sized patterns, DDR4-3200AA

.

This figure shows that the correlation between DRAM transfer size and net data bus utilisation follows a sawtooth shape. Between a transfer size of 1B and the optimal point for each configuration there is a linear correlation between transfer size and bus utilisation. Dropping over a configuration's pattern transaction size immediately halves bandwidth. As transfer sizes increase, performance converges to its optimum, passing over progressively improving local minima.

This sawtooth-shaped performance characteristic has significant drawbacks for various common algorithms, for example those performing $n \times n$-filter operations for $n > 1$. The work-items of such kernels require access to neighbouring elements of data. An efficient kernels preloads the tile of data required for all work-items in a work-group into local memory. For a work-group with its x-dimension m a power of two, a requirement of Sim-D, this tile consists of rows containing $m + n - 1$ elements. The size of these rows mean that their throughput on the graph is found at an x-coordinate just over one of the summits of a sawtooth wave.

5.1.3 Alignment constraints

Besides the power-of-two size constraint outlined in the previous subsection, patterns generated for pattern-based DRAM controllers also make the assumption that requests are *aligned* to multiples of a pattern transaction size. CPUs with associative caches guarantee this alignment naturally. However, this assumption does not hold for GPGPU systems where work-groups can request overlapping tiles of data. For example: for a kernel that implements an extrapolated- or constant-edges filter algorithm, tiling causes not only the size of a request to exceed a power-of-two, but also the first element of most work-groups to be slightly before a power-of-two boundary. The alignment of the tile preload request differs for each work-group.

With pattern-based DRAM controllers, misalignment can be dealt with in two ways. The first way is accepting that for unaligned accesses, extra pattern transactions must be issued. For large contiguous data requests, this implies adding one extra transactions to the worst-case latency. When data requests are non-contiguous, the number of transactions could potentially double in the worst-case.

If a DRAM controller is configured with a pattern reading nPW words, the number of transactions nT through a pattern for a contiguous transfer of w words is determined by:

$$nT = \left\lceil \frac{w + nPW - 1}{nPW} \right\rceil$$

To demonstrate the impact of these consequences, Figure 5.3 shows, for different configurations of a pattern-based DRAM controller, the data bus utilisation for a range of contiguous transfer sizes under this worst-case alignment assumptions.

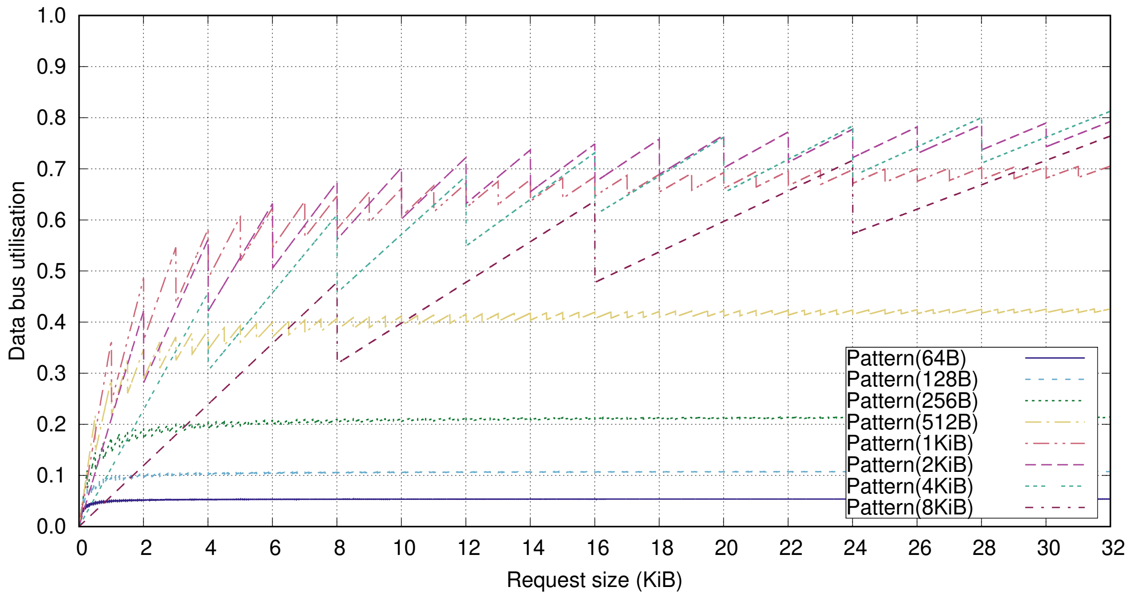


Figure 5.3: Data bus utilisation for unaligned transfers processed by a pattern-based DRAM controllers configured with different-sized patterns, DDR4-3200AA

Comparing Figure 5.3 with Figure 5.2 shows that the additional price for assuming worst-case data alignment is significant. For almost every transfer an extra pattern transaction must be issued. For transfer sizes in the lower range, this leads to a doubling of the response time. The percentage-wise penalty drops with every sawtooth-peak, but remains significant: where the largest two pattern-configurations can achieve $> 80\%$

utilisation for 4KiB transfers if perfect alignment may be observed, without such alignment constraints the 80% utilisation point shifts to transfers of 28KiB.

A second approach for dealing with unaligned requests would be to drop the alignment requirement when generating the patterns. This has two consequences for the resulting patterns. Firstly, patterns can no longer overlap precharge operations of request n with bank activations of request $n+1$, as the last bank of request n could now be the first bank of request $n+1$. This results in longer memory patterns, as they must include the worst-case latency for precharging all activated banks. Secondly, memory patterns can no longer be generated with the assumption that that multiple column operations on the same bank hit the same row, as requests could be aligned near the end of a row. Without these alignment assumptions, precharge and activate commands may be required between any two read/write operations on the same bank, cancelling out any gains made by increasing the pattern size.

5.1.4 Takeaway points

By means of experiments, I confirmed Akesson et al.'s claim [39] that the performance of closed-page DRAM controllers scales poorly with successive generations of DRAM. To mitigate the drop in efficiency, larger DRAM requests should be issued. I showed that pattern-based DRAM controllers do not provide satisfying performance characteristics for Sim-D for two reasons:

- They are optimised for a narrow set of transfer sizes, which do not include all common cases. Particularly, 2D filter kernels are expected to perform poorly,
- Transfers performance diminishes further if data is not aligned to multiples of a pattern size. Such alignment is unlikely when buffer widths are not a multiple of the pattern size or the kernel performs 2D filter operations.

5.2 Buffer characterisation and data locality

Having dismissed the use of existing HRT DRAM controllers for performance reasons, in this work I propose and study the concept of large, *explicitly-coalesced* DRAM requests. To scale requests to the size required for high DRAM bus utilisation, the processor pipeline supports *scalar* instructions that request to read or write data for an entire work-group. These requests will be serviced by the DRAM controller *uninterruptedly* and *deterministically*, benefiting from the bank parallelism within each large request. Determinism allows to statically determine a tight bound on the WCRET of each DRAM request in a kernel.

This approach is particularly efficient for kernels for which the correlation between work-item and data element(s) is easy to convey to the DRAM controller. As I show in this section, in many cases this is true: looking at benchmarks at a work-group granularity shows often their requests can be statically coalesced into a 1D or 2D block transfer.

By inspecting the buffer usage of all OpenCL kernels outlined in Section 4.3, I justify four Sim-D design choices: the inclusion of 1D and 2D (strided) load/store operations, the addition of a scratchpad local storage, the support of scalar registers and arithmetic and the omission of atomic read-modify-write operations. Table 5.1 summarises the results of this experiment. The columns in this table are interpreted as follows:

Type The type of a buffer could either be a primitive (float or int), a vector of primitives (e.g. float3, three consecutive 32-bit floating point numbers) or a struct consisting of multiple primitives. No nested structs are observed in any of the benchmarks.

R/W/A(tomic) These columns indicate whether the buffer is being read from, written to and/or whether atomic (read-modify-write) operations are used.

#refs/WI The number of read and write operations a single work-item performs on this buffer. If a kernel reads and writes the same word of a buffer, both are counted. It is assumed that a work-item does not read a data word more than once except through data-dependent indirect loads whose target cannot be determined in advance. For vectors and structs, each access to one of its primitive members is counted separately. For example, loading all elements of a float3 vector will result in three references.

Word (re-)use The number of work-items in a work-group that ideally use the same word in a buffer. E.g. a kernel performing a 3×3 filter operation on all 3×3 areas of an input buffer has a word re-use count of 9. Words at the borders of a buffer may be re-used less frequently. A word-reuse factor exceeding one indicates that caching or preloading data to a scratchpad can reduce the number of loads issued to DRAM.

"All" indicates that each word in the buffer is used by every thread in the work-group. For the 3×3 filter example, a buffer containing the 3×3 filter weights is used by every work-item. These weights can be loaded to Sim-D's scalar registers rather than stored redundantly in every column of a vector register, as is required on NVIDIA GPUs. For a three-dimensional NDRange, individual work-groups often only span two dimensions. In this case, "all" does not imply that *every* word is re-used by every work-item, but merely that there is a likely optimal work-group configuration for which each work-item in a work-group accesses the same word.

Buffer offset parameters These columns describes which variables are used to compute each work-item's index into a given buffer. The *TID* column indicates whether the global ID is used to determine the data point within the buffer. The *const* column indicates whether the index is offset by either a fixed amount or a fixed stride. *Dim* is ticked when the buffer dimensions are used to determine the offset in the buffer. Finally *Data* indicates that the buffer is indexed by a value loaded from another buffer in DRAM, i.e. an indirect memory access.

For example: a 3×3 filter would use the TID to find the start of the 3×3 region of pixels that this work-item will process. A constant is used to determine the offset from this central pixel for loading the pixels left and right. *Dim* would be used to index the pixels of the previous and next rows in the image.

Kernel	Buffer object(s)	Type	Access			#refs	Word	Buffer offset params			
			R	W	A	/WI	re-use	TID	Const	Dim	Data
CNN	in	float	✓			147	49	✓	✓		
convolution	kernels	float	✓			147	all		✓		
$(7 \times 7 \times 3)$	out	float		✓		1	1	✓			
relu	in	float	✓			1	1	✓			
	biases	float	✓			1	all	✓			
	out	float		✓		1	1	✓			
maxpool	in	float	✓			9	4	✓	✓		
$(3 \times 3$, pitch. 2)	out	float		✓		1	1	✓			
KFusion	in	float	✓			5	1	✓	✓	✓	
halfSampleRobustImage	out	float		✓		1	1	✓			
depth2vertex	depth	float	✓			1	1	✓			
	vertex	float3		✓		3	1	✓			
vertex2normal	vertex	float3	✓			12	4	✓	✓	✓	
	normal	float3		✓		3	1	✓	✓		
track	inNormal, inVertex, refNormal, refVertex	float3	✓			3	1	✓	✓	✓	
	output	struct		✓		8	1	✓	✓		
SRAD	d_iN,d_iS	int	✓			1	$\sqrt{n}$	✓			
srad	d_jW,d_jE	int	✓			1	$\sqrt{n}$	✓			
	d_I	float	✓			1	1	✓	✓	✓	
			✓			4	-	✓	✓	✓	✓
	d_dN, d_dS, d_dE, d_dW, d_c	float		✓		1	1	✓		✓	
srad2	d_iS,d_jE	int	✓			1	$\sqrt{n}$	✓			
	d_dN, d_dS, d_dE, d_dW	float	✓			1	1	✓	✓	✓	
	d_c	float	✓			1	1	✓	✓	✓	
			✓			2	-	✓	✓	✓	✓
	d_I	float	✓	✓		2	1	✓	✓	✓	
reduce	d_sums, d_sums2	float	✓	✓		$log(n)$	$log(n)$	✓	✓		
reduce_fpatom	d_sums, d_sums2	float	✓	✓	✓	9	1/all	✓	✓		
MRI-Q	phiR, phiI	float	✓			1	1	✓			
computePhiMag	phiMag	float		✓		1	1	✓			
computeQ	x,y,z	float	✓			1	1	✓			
	Qr, Qi	float	✓	✓		2	1	✓			
	ck	struct	✓			4096	all		✓		
SPMV	d_data, x_vec	float	✓			n	n				✓
jds_naive	d_index	int	✓			n	n				✓
	d_perm	int	✓			1	1	✓			
	jds_ptr_int	int	✓			n	all		✓		
	sh_zcnt_int	int	✓			1	all	✓	✓		
	dst_vector	float		✓		1	1				✓
LUD diagonal	m	float	✓	✓		< 705	< 44	✓	✓	✓	
perimeter	m	float	✓	✓		39	1	✓	✓	✓	
internal	m	float	✓	✓		33	33	✓	✓	✓	
Stencil	A0	float	✓			7	7	✓	✓	✓	
naive_kernel	Anext	float		✓		1	1	✓			
FFT	data0	float2	✓			4	1	✓	✓		
	data1	float2		✓		4	1	✓	✓		

Table 5.1: Memory access properties for all buffers present in the selected benchmarks.

For read-buffers, five interesting conclusions can be drawn from Table 5.1. Firstly, the vast majority of these buffers are indexed into using only the TID, constants and buffer dimensions. This means that work-groups can statically coalesce accesses from neighbouring work-items at compile time to form a 1D or 2D block DRAM request. Not all of these work-item-to-data mappings are linear. The most notable exception is FFT, which exhibits a non-trivial data access patterns. Although on the granularity of a work-group data is consumed from a large consecutive chunk, the distribution of individual words to the work-items requires data elements to be shuffled.

Secondly, 24 out of 64 buffers contain words that are re-used by multiple work-items in a work-group. This strongly motivates the inclusion of scratchpads in a SimdCluster.

Thirdly, several kernels implement variants of a filter algorithm: The CNN convolution and max-pooling benchmarks apply filtering with trimmed edges, the stencil benchmark performs a trimmed-edges filter operations on all direct neighbours in 3D space, and KFusion's vertex2normal kernel performs an extrapolated-edges filter operation on its neighbours in 2D space. Furthermore, the KFusion halfSampleRobustImage performs a 2×2 filter operation with a pitch of 2, which means that although no words are shared between adjacent work-items, pre-loading tiles of data to a scratchpad will still yield potential performance improvements.

Five read-buffers from four kernels contain scalar values that are re-used by *all* work-items of each work-group. For example, work-items from the CNN RELU benchmark share values from its "biases" buffer. Sim-D's scalar registers permit these values to be stored once rather than redundantly in each column of a vector register.

Fourthly, 11 buffers contain structs or primitive-vectors. When a kernel requests a single data word from such a buffer, the data of two adjacent work-items are separated by a pitch larger than one but generally smaller than nBW. Without caching, this leads to poor net data bus utilisation as many data elements in a burst are discarded. However, in all cases each work-item eventually requires all elements from its vector or struct. DRAM data bus utilisation can be improved in such cases by preloading a tile of structs or vectors to a scratchpad, and subsequently accessing their individual data words using indexed transfers into this scratchpad buffer.

Finally, the SRAD and SPMV kernels perform indirect accesses into buffers, meaning the index into the buffer is determined by data from a different buffer. From a real-time perspective these accesses are very difficult to optimise. The unpredictability of the required indexes results in a pessimistic upper bound on bank conflicts in DRAM. This results in pessimistic latency estimates for such requests, as is explained in Section 5.4.3 and demonstrated empirically in Section 5.5.3.

For write-buffers, I first note that none of the original kernels makes use of atomic read-modify-write operations. I developed the SRAD reduce fpatom kernel as an example of how atomic addition can improve performance of reduction kernels. Despite the potential for performance optimisation, the absence of atomic operations in existing benchmarks led to the decision to leave the implementation of atomic operations in Sim-D as future work.

Apart from the SRAD reduce kernel, none of the kernels output buffers see multiple work-items alias the same element. Most kernels map their work-items linearly to one or more output elements. Again the notable exception is the FFT benchmark which, like for its read operations, requires output data of work-items to be shuffled within a work-group.

The output buffers from the KFusion depth2vertex, vertex2normal and track kernels are laid out as an array of either primitive-vectors or structs. Similar to read methods of such data, Sim-D can facilitate efficient writing of such output data by first preparing a tile of vector- or struct-elements in local memory using indexed write operations. This tile is then written back to the target DRAM buffer using a 1D or 2D block transfer.

5.3 Architecture

The goal of Sim-D's DRAM controller is to efficiently service large DRAM requests such that the WCRET and LID of each request can be statically analysed. To this end, it implements a pipeline that dynamically translates a large request, issued by a work-group, into a sequence of DRAM commands. To eliminate variations in latency caused by interference between successive requests, it implements a closed-page policy on the boundary of each large DRAM requests. Within a request, the deterministic command scheduling policies allow exploitation of bank locality in a statically analysable manner. Optimisations that shift the worst case to less probable scenarios are considered out of scope for this work.

Following from the benchmark observations, Sim-D's DRAM controller supports the following large transfers:

- 1D- and 2D block transfers between DRAM and a SimdCluster's scratchpad or RF,
- Indexed transfers, requesting one word for each active work-item in a work-group.

Indexed transfers provide work-groups with a universal mechanism to request data of any layout. In this work, I evaluate two types of indexed transfers: *iterative indexed transfers*, where the DRAM controller iterates over indexes one-by-one, and *snoopy indexed transfers*, where (a region of) a buffer is streamed on the data bus, from which data elements are snooped using an array of address-matching CAMs. This array contains one CAM for each work-item in a work-group. For snoopy indexed transfers, Sim-D supports narrowing down the scope from an entire DRAM buffer to a 1D or 2D block within that buffer. This reduces the amount of bursts that are streamed over the data bus, improving performance both on average and in the worst case.

In the common case, a request routes data to vector register columns following a well-defined linear mapping between work-item and data element(s). 1D- and 2D block transfers can perform these transfers with high efficiency, making use of deterministic optimisations of the DRAM command stream.

Sim-D's DRAM controller is not intended as a drop-in replacement for a pattern-based DRAM controller. Most importantly, Sim-D's DRAM controller does not provide bandwidth or latency guarantees to each requestor in a multi-requestor systems. Rather, Sim-D can only provide bandwidth guarantees on individual requests. Combined with the analysis technique explained in Chapter 7, this guarantee is sufficient for Sim-D's programming model to allow WCET analysis of kernels.

This work is restricted to DRAM configurations comprising a single channel and a single rank. The DRAM data bus is assumed to be 64 bits wide. This bus width is in line with the embedded-grade NVIDIA Tegra K1 SoC [120]. Experiments are performed using a simulator developed in SystemC. This simulator integrates Ramulator [81] for guaranteeing DRAM timing properties, and DRAMPower [82] for generating power estimates of transfers.

5.3.1 Pipeline

Sim-D's DRAM controller is implemented as a four-stage pipeline: front-end, DRAM command generation, DRAM command arbitration and data movement scheduling (DQ scheduler). The *front-end* accepts either a *1D or 2D stride* request or an *iterative indexed* request, and translates this to a set of *burst requests*. The *command generator* performs address translation on these *burst requests* and determines which commands must be sent to DRAM. Scheduling these commands is done by the *command arbiter*. For read- and write commands, it creates a *DQ reservation*. In the final stage, the *DQ scheduler* generates control signals for the SimdCluster to synchronise DRAM data movement with the register file or scratchpad.

FIFOs are added between each pipeline stage to allow buffering of messages from each stage to the next. Two reasons necessitate this buffering. Firstly, the command arbiter and DQ scheduler do not consume their input elements at a constant rate, but rather at a rate dictated by DRAM latencies. Secondly, the command generator can generate multiple commands in a cycle, one per DRAM bank.

An overview of the pipeline is given in Figure 5.4. The remainder of this subsection provides details on each of the pipeline stages.

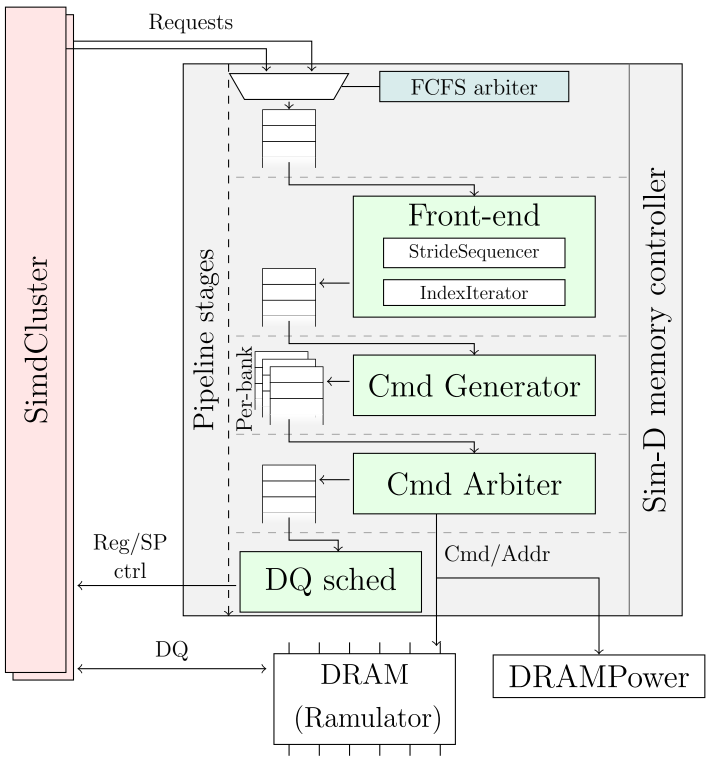


Figure 5.4: High-level architecture for Sim-D memory controller

5.3.1.1 Front-end

The front-end is responsible for translating a request into a sequence of required DRAM bursts. Two types of requests are supported: 1D/2D stride requests and iterative indexed requests. To support these two types of requests, two subcomponents are instantiated: the *stride sequencer* and the *index iterator*. The request input format differs between the two subcomponents.

Both subcomponents generate DRAM burst requests a rate of one per clock cycle. The format of a burst request is described in Table 5.2.

Name	Bits	Type	Description
addr	32	uint	Requested physical address, burst-aligned.
addr_next	32	uint	Physical address of next request, hint for linear_precharge policy
wordmask	nBW	array[bool]	Mask of requested (32-bit) words from burst
write	1	bool	True iff operation is a write operation
pre_pol	1	enum	Chosen precharge policy (linear/ALAP)
sp_offset	17	uint	Offset in scratchpad of first word
reg_offset	nBW * $\log_2$(nWGS)	array[uint]	Destination register offset
last	1	bool	True iff this burst request is the last for a DRAM req.

Table 5.2: Burst request message format

The front-end is designed as a state machine with six states: *IDLE, FETCH, INIT_STATE, RUN_STRIDESEQ, RUN_IDXIT* and *WAIT_ALLPRE*. It resets to the IDLE state.

When a requestor triggers a request, the front-end enters the FETCH state. In this state, it reads a request from the incoming request FIFO. When reading is completed, it enters the INIT_STATE state.

Depending on the request type, the INIT_STATE state initialises the registers of either the stride sequencer or the index iterator subcomponent. Additionally, the output routing logic is configured to make sure data is read or written to the correct register or scratchpad. It then sets its state to RUN_STRIDESEQ or RUN_IDXIT.

In these states, the associated subcomponent iteratively generates burst requests, one per cycle. After the last burst request is added to the output FIFO, the front-end goes into the WAIT_ALLPRE state. Here it waits until the DQ scheduler indicates that the last precharge command has finished and all DRAM banks are precharged, marking the completion of this request. The front-end returns to IDLE, ready to accept the next request.

Stride sequencer The stride sequencer converts a stride request into a series of burst requests. Incoming requests are encoded according to the format in Table 5.3. This format allows encoding arbitrary 1D and 2D stride patterns. Granularity for requests is in 32-bit words.

Name	Bits	Type	Description
start_addr	30	uint	Requested physical start address (aligned to 32-bit words).
period	20	uint	Length of the period of the stride pattern.
words_period	20	uint	Number of words to fetch for each period.
period_count	20	uint	Number of periods to cycle for.
end_addr	30	uint	$start\_addr + (period\_count - 1) * period + words\_period$
dst	3	struct	type: (SP, reg, CAM), work-group
dst_reg	~11	struct	Targeted register.
dst_offset	22	uint	Destination offset in scratchpad or first lane in VGPR.
dst_period	20	uint	Periodicity for the destination SP buffer or register file.
write	1	bool	True iff write operation, false iff read.

Table 5.3: Stride descriptor format

An example of a stride pattern reading a 5 × 3 tile from a 7 × 7 buffer is given in Figure 5.5. Reading is performed with a start_addr of 2 (DRAM address 0x8), a period of 7, words_period of 5 and a period_count of 3.

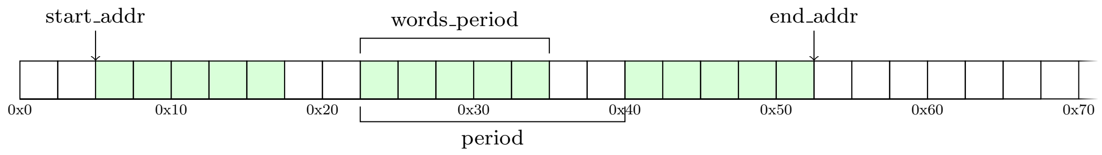


Figure 5.5: Example stride request

Given a DRAM burst must be aligned to the size of a burst, in Sim-D's case 64 bytes or 16 words, this stride pattern translates into two burst requests: one burst starting at address 0x0, and a second request at address 0x40. The stride sequencer sequentially creates these burst requests. For each burst request, it generates a wordmask to indicate which words from each burst, marked in green, must be routed to registers.

An abstract representation of the sequencing logic used to generate these burst requests is shown in Figure 5.6.

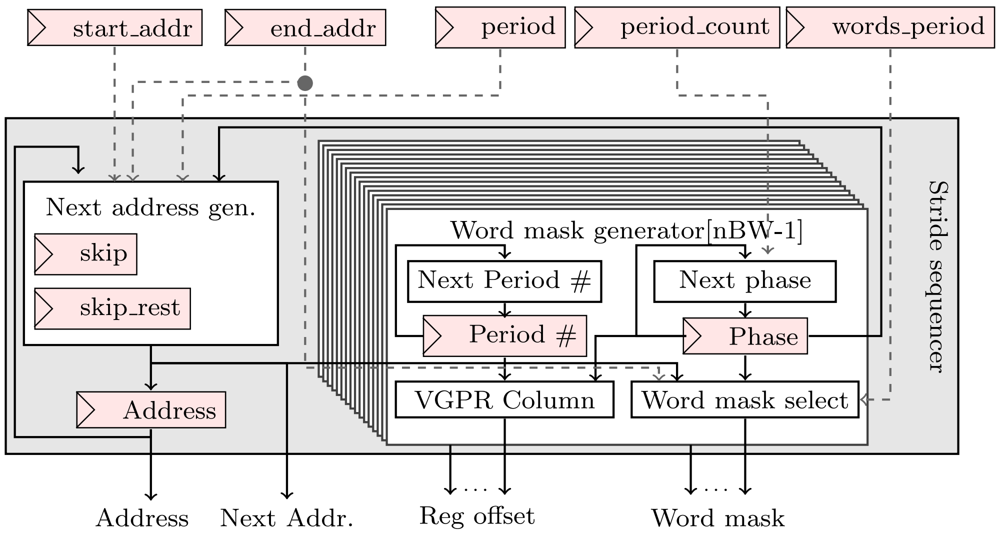


Figure 5.6: Stride sequencer

The stride sequencer contains a *next address generator*, plus one *word mask generator* for every word in a burst. Each word mask generator contains a *phase* counter, containing the offset of the generator's word relative to its current period. Because periods are not aligned to bursts, different word mask generators could work on different periods. The *word mask select* logic outputs a binary 1 iff its phase is smaller than *words_period* and the corresponding address is between *start_addr* and *end_addr*.

On each clock cycle, each phase counter is incremented by a separately computed *phase shift* c, after which the result is scaled to the range [0 :period) using a modulo operation.

Performing a full modulo operation is too costly. Even a reasonably sized non-pipelined radix-16 divider, like Intel's Core2 Radix-16 divider design [88] would take 5 cycles to perform a modulo operation on a 20-bit integer. With a peak DRAM throughput of one burst every 4 cycles, a 5-cycle modulo operation would form a bottleneck.

To avoid the high cost of a full modulo operation, the stride sequencer calculates the phase shift c such that $phase[n] + c$ never exceeds $2 * period$ for any n. By guaranteeing this property, each word mask generator's phase can be scaled to the range [0 :period) with a single conditional subtraction. To perform the addition and conditional subtraction, one *addition, single-overflow modulo (ASOM)* component is instantiated in each word mask generator. Each component is connected as ASOM(phase[n],c,period). The pseudo-code for an addition, single-overflow modulo (ASOM) module is shown in Listing 5.1.

```
ASOM(a,b,modulo) :
        result = a + b
        resultMod = result - modulo

        if (resultMod >= 0) /* Test for MSB == 0 */
                return resultMod
        else
                return result
```

Listing 5.1: Addition+Single-Overflow-Modulo functional representation

Besides calculating the phase increment c, the stride sequencer also calculates the increment inc used to obtain the address for the next burst request. The address increment and the phase increment are related, but not equal. To calculate c and inc, three cases must be considered depending on the value of period:

1. $period \geq nBW, words\_period < period < words\_period + nBW$
2. $period < nBW$
3. $period > words\_period + nBW$

Case 1 is the normal mode of operation. In this case, $c = inc = nBW$.

For case 2, a burst spans more than one phase. In this case phase counters can overflow more than once upon incrementing. To avoid multiple counter overflows for small periods, the stride sequencer contains a look-up ROM lut containing for each $period \in [1 : nBW)$ a pre-scaled value $c = (nBW \mod period)$. With nBW equal to 16 for a 64-bit wide DRAM data bus, this ROM contains 15 entries each of size $log_2(nBW) = 4$ bits.

In case 3, where $period$ is significantly larger than $words\_period$, it is possible to skip over those addresses where the resulting word mask is all-0. To this end the stride sequencer implements additional skip logic, that allows the stride sequencer to sustain a burst request generation rate of one per cycle.

To give a formal definition of this skip logic, first of all let $skip\_words = period - words\_period$. In the definitions that follow, assume all variables are in an unsigned binary form. The bit-wise AND and OR operators are depicted as & and | respectively. The ∼ prefix-operator describes a bit-wise inverse of the value that follows. $skip$ is defined as follows:

Definition 1. ***skip*** *is the minimum number of words contained in* ***whole bursts*** *between the last burst containing words from period n and the first burst in period $n + 1$.*

$$skip = (skip\_words - (nBW - 1)) \,\&\, (\sim (nBW - 1)) \tag{5.1}$$

Under certain alignment constraints, another burst of nBW words can be skipped over. To test whether this alignment holds, Sim-D pre-computes a *skip_rest* value during value initialisation that gives us the maximum value of $phase[nBW-1]$ for which another burst can be skipped:

Definition 2. ***skip_rest*** *is the largest possible $phase[nBW-1]$ for which the distance between the last burst containing words from period n and the first burst in period $n+1$ is nBW larger than skip.*

$$skip\_rest = ((skip\_words - (nBW-1))\ \&\ (nBW-1)) + words\_period - 1 \tag{5.2}$$

Given these two definitions, *inc* is defined as follows:

$$inc = \begin{cases} nBW & \text{iff } period < nBW\ OR\ phase[nBW-1] < (words\_period - 1) \\ skip + nBW & \text{iff } period \geq nBW\ AND\ phase[nBW-1] \geq skip\_rest \\ skip + 2 * nBW & \text{otherwise} \end{cases}$$

And c is defined as follows:

$$c = \begin{cases} lut[period] & \text{iff } period < nBW \\ ASOM(inc, 0, period) & \text{otherwise} \end{cases}$$

Upon receiving a stride request, each word mask generator must initialise its phase value. For $period \geq nBW$, this can be easily achieved in parallel. Defining *align_off* as the low $log_2(nBW)$ bits from the start address, $period[n] = ASOM(n, -align\_off, period)$. Note that using this assignment, period will be negative whenever the word from the word mask generator would fall before *start_addr*. Hence *period* must be a suitably provisioned signed integer.

For $0 < period < nBW$, a two-dimensional look-up table ROM $init\_lut[period][align\_off]$ is added to every word mask generator, containing pre-computed initialisation values. For the default configuration with $nBW = 16$, the size of each *init_lut* ROM is $15{*}16{*}4 = 960$b.

The stride sequencer always selects the linear precharging policy. This policy is explained in greater detail in Section 5.3.1.2.

To calculate the destination column in a vector register, each word mask generator additionally keeps track of its current *period number*. Following from the linear TID mapping scheme, the column offset is computed by multiplying this period number by the work-group width of the running kernel-instance, then adding the current phase value to the result. Because the work-group width is always a power of two, this is implemented using a left shift and a boolean OR operation.

For transfers to a scratchpad, the destination address is only given for the first word in every burst request. On each cycle, this address is incremented with the pop-count of the word mask from the previous cycle.

Index iterator The index iterator takes a buffer and a sequence of buffer offsets as input, and generates one burst request for each index. No attempt is made to coalesce burst requests as such logic has the potential of becoming quite complex yet offers no obvious benefits to the worst case LID or WCRET. In terms of average power consumption there is merit to such coalescing attempts for HRT systems, but lacking a full system power model, evaluation of a coalescing solution is beyond the scope of this work.

The index iterator has a small input FIFO to which the SimdCluster's register file pushes the requested indexes from the *vc.cam_idx* special purpose vector register. This FIFO contains (buffer offset, vector register lane, last request)-tuples. On each cycle, the index iterator pops one entry off the FIFO, calculates the address from the buffers base address plus the entry's offset, then uses the lower $log_2(nBW)$ bits of the address to generate both a one-hot word mask and the upper bits to generate an aligned burst request address. When the *last request* boolean is set, the front-end transitions to the WAIT_ALLPRE state and the "last" bit is propagated to the command generator.

The index iterator always selects the as late as possible (ALAP) precharging policy. This policy is explained in greater detail in Section 5.3.1.2.

5.3.1.2 Command generator

The command generator takes burst requests generated by the front-end, and generates per-bank DRAM commands. The command generator outputs 11-tuples as described in Table 5.4. The *Row* and *column* fields contain the mapped address. *pre_pre, act, read, write* and *pre_post* are booleans indicating which DRAM operations must be performed. *Word mask* indicates which words from each burst must be routed to storage, and the *target*, *sp_offset* and *reg_offset* identify the exact destination of the output words. These 11-tuples are stored in per-bank FIFOs, hence no *bank* field is required.

Name	Bits	Type	Description
row	16	uint	Requested DRAM row.
column	8	uint	Requested column in DRAM row, ex. burst-bits.
pre_pre	1	bool	Precharge before activate.
act	1	bool	Activate row.
read	1	bool	Read operation.
write	1	bool	Write operation.
pre_post	1	bool	Precharge after processing act/read/write
wordmask	nBW	array[bool]	Mask of requested (32-bit) words from burst
target	2	enum	Register/CAM array/Scratchpad WG0/Scratchpad WG1
sp_offset	22	uint	Offset in scratchpad of first word.
reg_offset	nBW * log_2(nWGS)	array[uint]	Target lane in vector register.

Table 5.4: DRAM command message format

The DRAM address mapping roughly follows the paired bank-group interleaving (PBGI) strategy [121], where two adjacent bursts always access alternating bank groups except on the boundary of a bank-group pair. This optimises for the common case of contiguous transfers by minimising the amount of times the inter-bank-group $tCCD_L$ penalty must be paid. The exact mapping is dependent on the DRAM chip configuration. For the two DRAM configurations outlined in Section 4.1.1, where a rank consists of either four 16-bit DRAM chips with 8 banks (2 bank-groups) or eight 8-bit DRAM chips with 16 banks (4 bank-groups) each, this mapping looks as follows:

Bit	3	3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1										
index	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
4*16b		r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	b	b	c	c	c	c	c	c	c	B	c	c	c			
8*8b	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	b	b	B	c	c	c	c	c	c	c	B	c	c	c			

Table 5.5: Address mapping (B: bank group, b: bank, c: column, r: row)

Internally, the command generator stores for each bank which row is most recently activated. When a burst request's row doesn't match the row stored as most-recently activated for this request's bank, the *act* bit of the output is set.

Generation of the *precharge* bit depends on the precharging strategy chosen by the front-end. Two strategies exist: *linear* and *as late as possible (ALAP).*

The linear strategy is optimised for strided transfers where it is known that the sequence of addresses requested is strictly monotonically increasing. In this case, when the input's *next_address* exceeds the latest address that can be serviced from the bank-pair's active rows, the command generator performs the following three actions:

1. Set the *pre_post* bit on the output alongside the *read* or *write* bit.

2. Issue a separate precharge command for the paired bank in the other bank group.

3. Reset the *most recently activated* row for both banks to a reserved value signifying "no active row".

The as late as possible (ALAP) strategy is a heuristic that works better when nothing is known about the sequence of addresses, e.g. for transfers that require the "indexed iterative" method. In this case, a bank is precharged only when a request is encountered that addresses a bank that has a different open row. This is achieved by setting the "pre_pre" bit on the associated read or write command. When the last command is generated, indicated by the *last* bit in the input message, an additional precharge command is issued for all remaining open banks.

5.3.1.3 Command arbiter

The command arbiter has two responsibilities: selecting the next command from the per-bank input FIFOs to schedule, and timely scheduling of refresh commands. For command selection, the following prioritisation rules apply in order:

1. Read/write commands are scheduled as early as possible.
2. Read/write commands are scheduled from the *currently active bank-pair* until a precharge is encountered. When a precharge is encountered, the next active bank pair is selected.
3. Read/write commands have priority over activate.
4. Activate commands have priority over precharge.
5. Row activate commands are prioritised according to the number of column access strobe (CAS) operations present in the respective bank FIFOs targeting said row, tie-broken by distance from the currently active bank pair.
6. Refresh operations are always scheduled between two stride requests, but may be scheduled within an indexed-iterative request.

The priority of rule 1 over rule 2 is of particular importance for performance: when there is a read/write command schedulable from outside the currently active bank-pair, but none from within the currently active bank-pair, this command will still be scheduled. With non-unit stride transfers, where certain stride-patterns could lead to an imbalance between requests for both banks in the bank-pair, this reduces idle cycles on the data bus. Because the worst-case stride pattern for which this behaviour can be exploited requests subsequent bursts from the same bank shifting the relevant word within each consecutive burst by one word, I suggest that the depth of the per-bank input FIFOs (including the banked head) is at least equal to nBW to avoid back-pressure from the command arbiter on the command scheduler. For a 64-bit wide DRAM data bus, this corresponds to 16 entries per FIFO.

Rule 2 is a heuristic to achieve “greedy precharging” when the linear precharge strategy is followed. This heuristic helps preparing a bank for activation as early as possible, which in turn reduces the risk of a precharge or activate delay ending up on the critical path.

Rule 5 presents a small optimisation for DRAM configurations with two bank-groups, as it maximises the opportunity for performing CAS operations in parallel with back-to-back bank activations. In practice this optimisation only saves unit-stride transfers a single cycle, while the benefit for non-unit stride transfers depend on data alignment. This benefit completely disappears for DRAM configurations with four or more bank groups. Given the cost of implementing this optimisation in hardware, predominantly for the required per-FIFO CAS-operation counter, a hardware implementation may omit this optimisation. However, the LID and WCRET analysis presented in Section 5.4 assumes presence of this prioritisation rule.

During operation, the top entry of each FIFO is banked in the command arbiter. As the various activate, read/write and precharge commands are issued by the command arbiter, their respective bits in the banked commands are cleared. Once all bits of a banked entry are cleared, the entry is discarded and a new entry is loaded from the corresponding input FIFO.

Using the value of an *active bank-group* register for prioritisation, the arbiter will first select three candidate commands, being the highest priority activate, read/write and precharge request. Following the prioritisation rules 3-5, the command for this cycle is then selected from the candidates. When a banked candidate has both its activate and read or write bits set, the request is considered an activate request until the corresponding “ACT” is scheduled. Similarly, a request for which the “pre_pre” bit is set is considered a precharge command until this precharge has been scheduled. The “pre_post” bit promotes a DDR4 “RD” or “WR” command to a “RDA”/“WRA” implicit-precharge command.

When issuing a read or write command, a DQ reservation is generated on the output FIFO to schedule the data movement from the DRAM DQ lines from/to the scratchpad or registers. The DQ reservation format is described in Table 5.6.

A hardware implementation of a command arbiter would consist of many downward binary counters that ensure timing requirements of the DRAM chips are satisfied. Instead, the simulation model relies on the timing model provided by Ramulator [81] to ensure correct, cycle-accurate behaviour of the memory controller. One downward counter that is modelled explicitly by Sim-D is the *last precharge*-counter. When the precharge for the last incoming DRAM command is scheduled, this counter is reset to the number of cycles it takes for the precharge to complete. When this counter reaches zero, the *all banks precharged* signal is asserted such that the front-end can transition from the WAIT_ALLPRE to the FETCH state.

Name	Bits	Type	Description
cycle	64	uint	Cycle when data must/will be available on DQ.
write	1	bool	Operation is a write operation (false: read).
wordmask	nBW	array[bool]	Mask of requested (32-bit) words from burst
reg_offset	$nBW * log_2(nWGS)$	array[uint]	Target lane in a vector register
sp_offset	22	uint	Offset in scratchpad of first word.

Table 5.6: DQ reservation message format

5.3.1.4 DQ scheduler

The DQ scheduler is responsible for generating the control signals required to instruct the requesters' storage resources (scratchpad, register file) to read or write data to the DRAM data bus. The arrival of these control signals is synchronised with DRAM, taking into account the CAS and column write delay (CWD) latencies of the read/write commands issued by the command arbiter. The DQ scheduler's input is a FIFO of (cycle, DRAM word mask, scratchpad/register offset(s), R/W) 4-tuples describing at which cycle the first beat of a burst will be available and where the data should go. When the value from the global cycle counter matches that of the top entry on the input FIFO, the DQ scheduler pops the entry and generates the control signals required to load or store the data from the targeted scratchpad or register file.

In the Sim-D simulator, the DQ scheduler emulates the DRAM storage. In a real implementation, the DQ scheduler is instead expected to buffer the DRAM data offered on both the rising and falling edge of a clock cycle in order to transfer the data to the register file or scratchpad at the rate of the command bus rather than at the double data rate of the data bus.

5.3.2 Snoopy indexed transfers

For HRT systems, indexed transfers are problematic. Regardless of how indexed transfers are implemented in hardware, there exists a worst case for which the relevant data is stored so sparsely that for every word a full DRAM burst read/write must be issued. With DDR4 DRAM, all words could reside in the same bank group, spacing consecutive bursts $tCCD_L$ cycles apart. Even with an oracle to schedule DRAM commands with minimal overhead, the worst-case data bus utilisation for such large buffers can never exceed:

$$\frac{1}{\frac{nBW}{tBURST} * tCCD_L} \quad \rightarrow \quad \frac{tBURST}{nBW * tCCD_L} \tag{5.3}$$

For the two bank-group DDR4-3200AA configuration this bounds worst-case bus utilisation to $\frac{4}{16*8} = 3.125\%$. Despite this low data bus utilisation, use-cases might still necessitate indexed transfers.

For the index iteration method described in Section 5.3.1.1, the given bound is an upper bound on performance regardless of the buffer size. Worse, as demonstrated in Section 5.5.3, for sufficiently large buffers the worst-case efficiency of the index iteration method does not exceed $\frac{1}{tRC} < 0.34\%$. This leaves scope for improvement.

In this work, I evaluate a second, snoopy, method for supporting indexed transfers. For a snoopy indexed transfer, a sequence of burst read or write commands is issued to cover an entire buffer. For read operations, dedicated per-work-item CAMs detect on each cycle whether the data currently present on the bus is for the index requested by this work-item. If so, it snoops the word from the bus into its corresponding data register. For write-operations, when a CAM indicates that its index matches one of the words written in a cycle, it signals a match to set the corresponding DQM bits and writes the word from its data register to the correct DQ lines.

The main advantage of this approach is that the DRAM command stream generated by this method is that of a (non-)unit-stride transfer. This implies that rows will never be activated more than once and bursts can often be scheduled $tCCD_S$ apart. Snoopy transfers efficiently coalesce multiple reads/write burst requests to the same data word.

Compared to indexed iterative reads, two downsides are identified. Firstly, because all data from a buffer or tile is read without regard for the values in the indexes, bursts can be read or written whose words remains unused. Secondly, the cost of implementation is substantial. The design of this data snooping mechanism is broadly comparable to a fully associative cache with as many entries as there are work-items in a work-group. In the Sim-D configurations evaluated, this unit requires 1024 CAMs and a data distribution network from each word on the data bus to 1024 registers. The wiring overhead and fan-out of the data bus are expected to dominate the hardware overhead.

Literature gives some hints about the feasibility of this scheme in terms of latency. To keep up with the data rate of DDR4-3200AA DRAM, comparison and data retrieval must be completed in 625ps. Agarwal et al. [122] demonstrate how on 32nm technology, a CAM can perform a 128-bit search in 145ps. If “don’t cares” are required in the comparison word, Onizawa et al. [123] show that a 32-bit single cycle search is possible with today’s technology in under the target latency of 625ps.

Implementation of this scheme requires two adaptations to the current design. Firstly, the stride sequencer explained in Section 5.3.1.1 must be extended to augment transferred words with their indexes. Secondly, the SimdClusters’ RFs must be extended with a data snoop unit that performs the snoopy reads and writes. This component is replicated once per RF to allow usage with both DRAM and scratchpad transfers.

5.3.2.1 Data snoop unit

The data snoop unit is responsible for monitoring the stream of data on the data bus. To this end, one logical *lane* is instantiated for each work-item in a work-group. Figure 5.7

displays a potential logical design of one lane in the data snoop unit, omitting the configuration path of the address- and data registers. The displayed unit assumes a DQ of 4 words, but when re-using this component with a wider scratchpad data bus, components must be scaled and replicated accordingly.

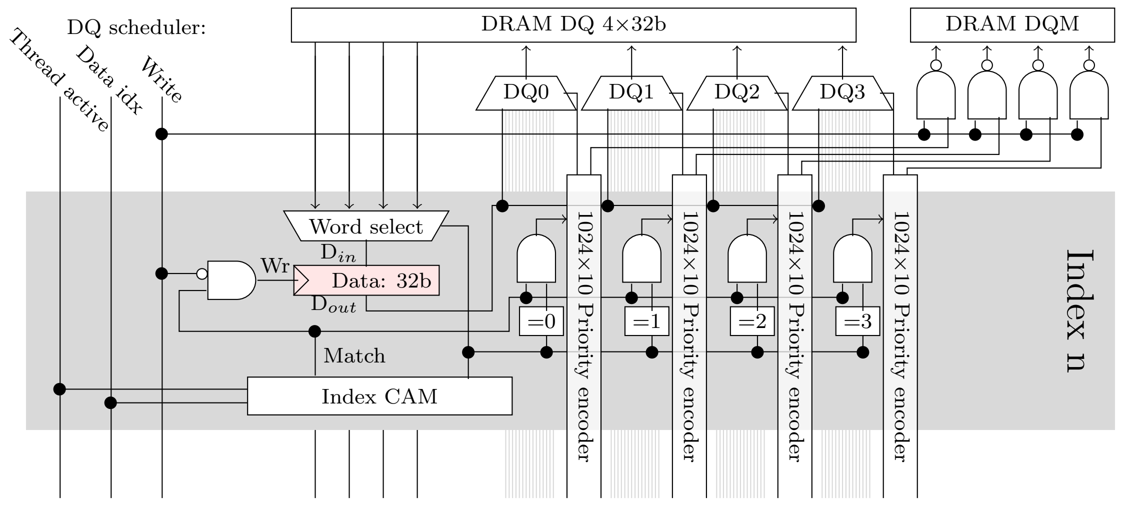


Figure 5.7: Logic for snoopy indexed read/write operation, single index

Before a snoopy indexed transfer is initiated, each work-item must set the requested index in the *vc.mem_idx* register, which maps to each lane's *Index CAM* register. For write-operations, the data to be written must be stored in the *vc.mem_data* data register, which maps to the data register marked in red.

When the DRAM controller processes a read request, the index of the first element is provided on the match lines. Contiguous data words on the DQ must necessarily contain data from contiguous indexes. Hence only a single comparison is required per-cycle. On a match, the CAM sets the write-bit of the data register. The difference between the *data idx* value and the value in the index cam register provides the select value for the word-select MUX. The data register is never written to if its corresponding thread active bit is cleared. This bit is provided by the SimdCluster's implicit predicate mask.

For a write request, again the DQ scheduler publishes the index of the first word on the CAM's match line. If the index matches the elements in the address register, the priority encoder corresponding with the correct data-out word is notified. This priority encoder then selects one lane whose data word is placed on the data bus through a wide MUX. If multiple matches on the same word occur, for example because multiple work-items try to write to the same location, the priority encoder ensures that only one of the words is written while the others are discarded, ensuring validity of the resulting value. The priority encoder additionally ensures that the DQM bit for this word is cleared correctly, to avoid overwriting data for which no lane wishes to write a data word.

Details about the index matching logic have not been decided on at this point, and depend on the resulting hardware propagation delay. An important design decision is the content of the address register. By the nature of DRAM, it is known that addresses must be aligned to word widths. If the SimdCluster fills this register with physical addresses, the match logic can be kept simple as it can just compare the high bits of the *data idx* signal with the value of the *index CAM*. A downside of this approach is that applications require additional arithmetic to transform their word offsets (indexes) into addresses.

Alternatively the index CAM contains actual indexes, offsets within a buffer at a word granularity. Unfortunately, buffers have no strict alignment requirement. Hence in this case the match logic will be slightly more elaborate: it would first subtract the first index from its own index, then test whether the result is a positive number between 0 and the DQ width. If so, it would snoop the corresponding element off the bus. This approach allows for applications to remain oblivious of the physical address of data. As a downside, the comparator inside the CAMs will have a longer latency.

Hardware synthesis would give more insight into the feasibility of either scheme. This is deemed beyond the scope of this work. However, as a general note: if synthesis of the second implementation indicates that the comparator violates timing constraints, it is possible to instead enforce alignment constraints on the buffers that are used for snoopy indexed transfers. This would allow to simplify the comparator to that of the first implementation, or even smaller if limitations on buffer sizes are enforced.

5.3.2.2 Stride sequencer

Depending on the chosen hardware implementation, the stride sequencer must be extended to convey either the address or index of the first element of the data on the DQ. This information will then traverse down the pipeline such that the DQ scheduler can emit them at the right time.

For the implementation where addresses are matched, this could simply be the value of the *global_addr* register. If an implementation is chosen that matches index lines, the stride sequencer will need a new counter register that keeps track of the currently addressed index in the first word of the DQ. Its increment on each cycle would be equal to the increment of the global address, shifted by 2 locations to count words rather than bytes.

5.3.3 Takeaway points

This section introduced Sim-D's DRAM controller as a four-stage pipeline: front-end, DRAM command generation, DRAM command arbitration and DQ scheduling. It supports *1D and 2D stride requests*, *iterative indexed requests* and *snoopy indexed requests*. For the latter, changes are presented both to the front-end and the SimdCluster's register file.

5.4 Worst-case request execution time and issue delay

This section presents techniques for statically deriving the worst-case request execution time (WCRET) and longest issue delay (LID) of any DRAM request processed by the Sim-D DRAM controller. For all these derivation methods it is assumed that:

1. The size of a buffer is known a-priori,
2. The start address for a request is only known at run-time,
3. The number of requested words w can vary during run-time but can be upper bound statically.

None of the methods in this section consider the cost of DRAM refresh. In Section 7.4.8 I explain how refresh is safely accounted for when performing WCET analysis of Sim-D kernel-instances.

5.4.1 Unit-stride

For contiguous, unit-stride reads or writes, two request parameters determine the worst-case latency: request length and start address alignment.

The start address alignment influences the worst case in two ways. Firstly, because bursts are aligned to multiples of nBW words, the alignment of data determines the number of bursts that must be issued to access each word. Secondly, data alignment affects the number of banks that are accessed by a request. If data is aligned such that it starts near the end of a pair of rows, a transfer that could in the best case be serviced from two banks could now require data from four banks (two bank pairs). The activation delay for these four rows ends up on the critical path, increasing the delay of the request.

The length of the request can be used to bound alignment-related overhead. To do so, I first define the length of a request in terms of the number of bursts required. For a given maximum number of words $w > 0$, the maximum number of bursts n is given by the following equation:

$$n = \lceil (w-1)/nBW \rceil + 1 \tag{5.4}$$

For unit-stride transfers the WCRET and the LID are both monotonically increasing with the number of requested bursts[1]. For this reason, for a given w only transfers with exactly n bursts as given in Equation 5.4 need to be considered.

[1]When inspecting all schedules, I found that for $n = 8$ the worst case schedule for the two-bank DDR4-3200AA configuration is one cycle shorter than that for $n = 7$, as the extra burst causes the greedy algorithm to take a globally better scheduler decision. This provides the only known counter-example

The LID and WCRET of a unit-transfer can be derived by considering all possible alignments of its block of n bursts. For the address mapping outlined in Section 5.3.1.2, the number of possible alignments is equal to four times the number of columns in a row: two words per column for a DQ width of 64 bits and two rows in a bank pair. With the DRAM configurations described in Section 4.1.1, this totals to 4096 unique alignments. This bound is motivated by the observation that it makes no difference in which bank pair the first burst of the request is situated. Performance is uniform among all bank pairs, and command scheduling decisions taken by the DRAM controller will not change depending on the index of the bank pair for the first issued read or write operation.

The limited number of alignment possibilities makes it feasible to exhaustively simulate all possible schedules for a given unit-stride (or indeed non-unit-stride) request. On my test set-up, simulating a unit-stride request for all 4096 possible alignments using the SystemC model takes less than a minute. By analysing the simulated worst-case schedules, I found that for unit-stride transfers it is possible to express the LID and WCRET as a function of request size and DRAM timing parameters. This function increases monotonically with the request size. The remainder of this subsection presents the equations resulting from this analysis.

Traces of issued DRAM commands reveals that the command stream can be broken up into three phases, each dominated by the following constraints:

1. Activation of first few banks, constrained by the row-to-row activation delay (tRRD).

2. Read/write of activated banks, constrained by the column-to-column delay (tCCD).

3. Precharge of the last accessed bank, constrained by tRTP+tRP.

These three phases bear some resemblance to stages of pipelined execution: *warm-up, maximum throughput* and *drain.* The latency of the worst-case command schedules for the first two phases is equal for both read- and write requests.

The length of the *warm-up* phase spans from the first activate command until the time that at least one bank from each bank group is activated such that multiple reads or

against the property of monotonicity of latencies. Because I set out to derive an upper bound on latency, I present equations for which the delay with $n = 8$ is equal to that of $n = 7$ to maintain the property of monotonicity for the WCRET and LID equations.

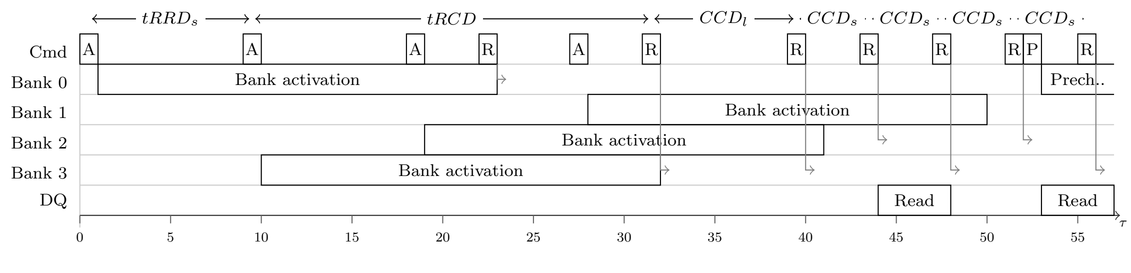


Figure 5.8: Partial DRAM schedule for worst-case unit-stride read, n odd.

writes can be issued to each in an alternating fashion. A perfect oracle DRAM command scheduler would be able to reach the second phase with exactly two activate commands. The second activate would furthermore be overlapping with the first read or write to the first bank, such that some work is already performed for the second phase early on. Unfortunately, Sim-D's greedy command scheduling strategy allows for a worst-case situation where the first two activated rows are only accessed exactly once.

Figures 5.8 and 5.9 demonstrate, for odd and even n respectively, the first 58 cycles of the worst-case schedules for the warm-up phase. These schedules are obtained from simulation of a two bank-group DDR4-3200AA configuration. The arrows above the schedule indicate the latencies that make up their critical paths. The schedule in Figure 5.8 is achieved with an alignment such that from both bank 0 and 1 only a single burst must be read or written from the first row. The schedule in Figure 5.9 represents the case where two bursts must be read or written from the first addressed row in bank 0 and 1.

For both worst-case schedules, at cycle 0 the only command ready to issue is an activate for a row in bank 0. Unfortunately, this activation limits the opportunities for executing read commands in parallel with subsequent activates, the number of reads from row 0 is limited.

The second activate command is issued $tRRD_S$ cycles after this row activation. At this point, the command scheduler can issue an activate for a row in either bank 1 or bank 3. The command arbiter's fifth command prioritisation rule (prioritise activate commands based on the number of read or write operations present in the FIFO targeting the activated row) demonstrates its merit by picking bank 3 over bank 1. Both banks can be activated at this point in time, but for bank 3 there are more read or write operations available in the command arbiter's FIFO, hence offers more potential for masking the latency of subsequent activate commands. This effectively substitutes $tRRD_L$ with $tCCD_L$ on the critical path of the warm-up phase, the difference between the two being one cycle.

The third activate of bank 2, another bank for which many reads or writes are available, is issued another $tRRD_S$ cycles later. Because bank 3 is targeted by more than one burst, the $tRRD_S$ penalty between activating bank 3 and 2 is masked by the $tCCD_L$ latency required for two subsequent read or write operations on bank 3. As displayed by the annotations at the top of Figure 5.8, the total time for the warm-up phase is

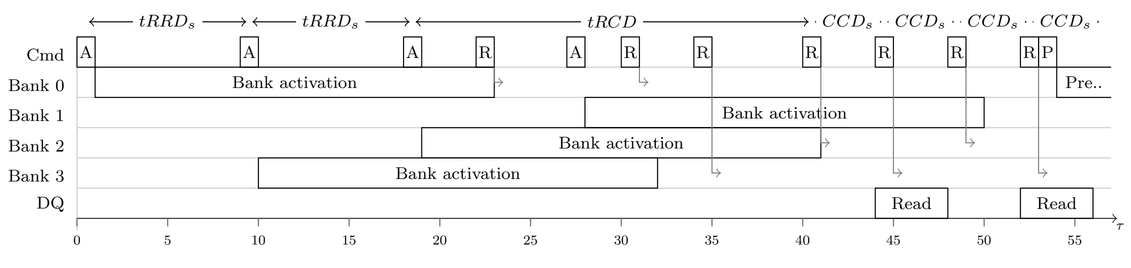


Figure 5.9: Partial DRAM schedule for worst-case unit-stride read, n even.

$tRCD + tRRD_S + tCCD_L$ cycles.

In the *throughput* phase, the command arbiter simply issues a DRAM read or write command every $tCCD_S$ cycles. The chosen address mapping ensures that after the warm-up, there is always one bank in each bank group available to read or write from. Transfers that require more than four banks are of such size that while reading from the third and fourth bank, there is plenty of time to perform activation of the fifth and sixth in parallel. Figure 5.8 shows that when the throughput phase starts at cycle 39, the command arbiter has already issued three read commands. This means there are $n - 3$ commands left to be issued. As the read commands issued at cycle 31 and 39 are to the same bank group, two reads from the same bank group remain at the end of this phase. The total length of the throughput phase phase is thus $(n - 4) * tCCD_S + tCCD_L$. Likewise, Figure 5.9 demonstrates that when n is even, the length of the throughput phase for the worst case schedule starting at cycle 40 is $(n - 5) * tCCD_S + tCCD_L$.

For requests with $n \leq 8$, different worst-case critical paths emerge as a result of disappearing delays and reduced parallelism. For example, with $n = 1$ there is no requirement to wait $tRRD_S$ during the warm-up phase as there is only a single bank to activate. Likewise, the number of reads or writes to parallelise with activates on the critical path is limited when there is only one read/write operation per bank to begin with.

Note that the paired bank-group interleaving (PBGI) address mapping guarantees that addressing more than four banks with a unit-stride transfer is only possible if the length of the transfer exceeds two rows by at least three bursts. The time it takes to issue read or write requests for these two rows greatly exceeds the four-activate window. Similarly, for small transfers DRAM, latencies are such that the row cycle time will exceed the four-activate window. This means that tFAW does not need to be taken into account when performing worst-case analysis of unit-stride transfers.

An inspection of worst-case schedules results in the following definition of a function $ACTCAS(n)$ which, for given n, determines the time required to perform the first two pipeline phases.

$$ACTCAS(n) = \begin{cases} (n-1) * tRRD_s + tRCD & \text{iff } n \leq 4 \\ 2 * tRRD_S + tRCD + (n-4) * tCCD_L + tCCD_S & \text{iff } n \in [5,6] \\ 3 * tRRD_S + tRCD + tCCD_L + tCCD_S & \text{iff } n \in [7,8] \\ tRRD_S + tRCD + 2 * tCCD_L + (n-4) * tCCD_S & \text{iff n odd} \\ 2 * tRRD_S + tRCD + tCCD_L + (n-5) * tCCD_S & \text{otherwise} \end{cases}$$

The final *drain* phase determines the difference between the LID and the WCRET. The distance between the last read operation of the current request and the first activate of the next is $tRTP + tRP$, while the distance between the last write operation and the first activate of the next request is $tCWD + tBURST + tWR + tRP$. Any other bank that has been activated during this transfer will have a read or write operation before this last one and hence will be precharged earlier by the command arbiter. To calculate the WCRET of a read or write operation, the drain latency is the moment the last beat of data arrives on the data bus, defined as $tCAS + tBURST$ and $tCWD + tBURST$ respectively.

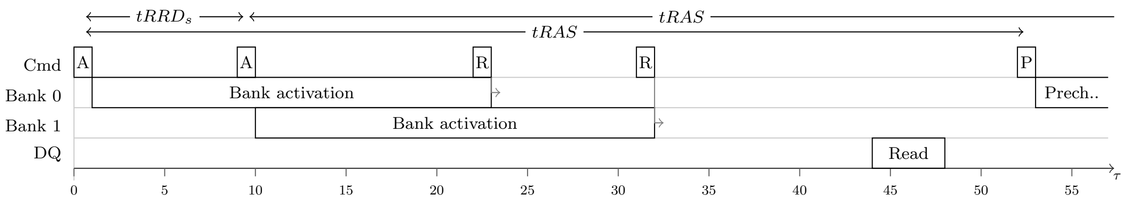


Figure 5.10: DRAM schedule for worst-case read, n=2

Before a full equation for the LID can be composed, it is important to account for read operations with $n \leq 8$. For these transfers the minimum distance between precharge and activate of a bank, $tRAS$, can end up on the critical path. Figure 5.10 demonstrates the first 58 cycles for the worst-case schedule when reading two burst, one per bank. In this example we can see that the last read operation is issued at cycle 31, as predicted by $ACTCAS(2)$. However, the precharge of the first activated bank only occurs after $tRRD + tRAS$, which is larger than $ACTCAS(2) + tRTP$.

The lengths of $tRAS$-bound schedules are bound solely by the number of banks that can be addressed, and hence the issue delay for read operations can simply be characterised by $min(n-1, 3) * tRRD_S + tRAS + tRP$. When n becomes large enough for $tCCD_S$ to end up on the critical path, the worst-case is naturally described by the schedules demonstrated in Figures 5.8 and 5.9. The LID is thus the maximum number of cycles determined for the regular schedule and the $tRAS$-bound schedule.

The same effect does not occur for write operations, as here the minimum precharge distance $tRCD + tCWD + tBURST + tWR > tRAS$. Hence, even a request with $n = 1$ will not have the $tRAS$ delay on its critical path for its issue delay.

Taking all these observations into account, the following equations provide a tight bound on the worst-case read- and write latencies by combining the components of these three phases:

$$\begin{aligned} tID_R(n) =& max \begin{pmatrix} ACTCAS(n) + tRTP + tRP, \\ min(n-1,3) * tRRD_S + tRAS + tRP \end{pmatrix} \\ tWCRET_R(n) =& ACTCAS(n) + tCAS + tBURST \\ \\ tID_W(n) =& ACTCAS(n) + tCWD + tBURST + tWR + tRP \\ tWCRET_W(n) =& ACTCAS(n) + tCWD + tBURST \end{aligned} \tag{5.5}$$

The pipeline of Sim-D's DRAM controller additionally adds a fixed latency of 3 cycles from the moment a SimdCluster kicks off a request in the front-end.

Note that for DDR4 DRAM, these equations imply that the LID is always greater than the WCRET. For read operations this is true because $tRTP + tRP > tCAS + tBURST$. For write operations, this is trivially true as $tCWD + tBURST + tWR + tRP > tCWD + tBURST$.

The difference in cycles between the two metrics is small. For example, for the DDR4-3200AA system the difference between $tID_R(n)$ and $tWCRET_R(n)$ is 9 cycles for sufficiently large n. Technically these cycles can be put to use by resuming a work-group after expiration of the WCRET while scheduling the next DRAM request after LID has passed, helping to reduce unnecessary stalls in applications where phases are unbalanced. However, for strictly I/O-bound applications, the difference between a request's WCRET and LID is not exploitable for higher performance because the next request will not be finished earlier, while for compute-bound applications there are no observable benefits given the total compute time is not reduced. The scope of benefiting from this difference in number of cycles when determining an application's worst-case execution time (WCET) is thus very limited.

5.4.1.1 Four or more bank-groups

For DRAM configurations with 16 banks and 4 bank groups, it is possible to simplify the calculation of the warm-up latency. There are two main reasons for this. Firstly, four adjacent bank pairs now all come from different groups. This means that it is less likely to incur the higher $tRRD_L$ and $tCCD_L$ latencies for "same bank group" operations. Secondly, for the chips used for such DRAM configurations, $tRRD_S$ is equal to $tCCD_S$. This means that where the greedy scheduler would previously schedule two reads to the same bank during the activation phase on account of no banks being activated yet, now the scheduler will have two active banks to pick its second and subsequent commands from. As a result, the long $tCCD_L$ delay can be avoided. Both effects result in a more straightforward command schedule.

For these reasons, $ACTCAS(n)$ can be simplified for DDR4 configurations with four or more bank groups to:

$$ACTCAS_{4bg}(n) = tRCD + (n-1) * tCCD_S \tag{5.6}$$

Equations 5.5 continue to apply with this modified definition.

5.4.2 Non-unit-stride

For non-unit-stride transfers, the WCRET and LID are determined by the three stride request parameters introduced in Section 5.3.1.1: the period length *period*, a number of words per period *words_period* and a number of periods *period_count.*

For 2D block transfers the *period* of a stride is equal to the x-dimension of the targeted buffer. which is known a-priori. The number of requested words is equal to $words\_period * period\_count$. Like the number of words w for unit-stride transfers, I assume that both *words_period* and *period_count* can be upper bounded a-priori. At run-time, a kernel-instance may issue stride requests with smaller *words_period* and *period_count*, for example when requesting data for remainder work-groups processing the edges of a 2D data structure. Since this does not change the dimensions of the 2D data structure, the *period* remains fixed.

Unfortunately, the relationship between the *period* parameter and the request latency is non-monotonic. To illustrate this problem, consider a stride pattern for which $words\_period = 1$ and a sufficiently large $period\_count$. If *period* is a multiple of $2 * nBW$, for example 32, all burst requests are mapped to the same bank group. However, for a *period* of either 31 or 33 and sufficiently large *period_count*, the accessed bank alternates between the two banks in the bank-pair every 16 burst requests. When the per-bank command FIFOs are adequately provisioned with 16 entries each, the command arbiter is able to effectively exploit bank parallelism after the warm-up phase, resulting in a significantly more efficient transfer than when $period = 32$.

As for the other parameters: when *period* remains fixed, neither *words_period* nor *period_count* can influence the alignment of data. Hence there exists a monotonic relation between latency and the latter two parameters. This is an important observation, as it means that the LID and WCRET found for upper-bound values of *words_period* and *period_count* are *safe* for requests issued by remainder work-groups.

The non-monotonic relationship between a stride's *period* and the request latency makes it unlikely that a useful equation is found to bound the LID and the WCRET for non-unit-stride transfers in the same way Equations 5.5 bound these latencies for unit-stride transfers. Instead, given a fixed *period* and an upper bound on *words_period* and *period_count*, the LID and WCRET of a request are determined by simulating all possible alignments within a bank pair using the SystemC simulator.

5.4.3 Index iteration

For transfers using the indexed-iterative method, the LID and WCRET depend on two parameters: the number of requested words w, and the size s of the buffer that data is requested from. In the absence of logic that coalesce multiple requested words into a single burst request, the number of bursts $n = w$ both in the average- and worst case.

Exhaustive simulation of all possible combinations of (1 bank, 2 bank-groups) and (2 bank, 1 bank-group) followed by analysis of the generated trace output from Sim-D's DRAM controller has led to the equations for LID and WCRET presented in the remainder of this subsection. Given timing of requests is not influenced by the column that is accessed within a (bank,row) combination, the search space of all possible worst-case DRAM command schedules is limited to unique sequences of (bank,row)-pairs. From this analysis, two worst-case access patterns emerge: one for buffer sizes large enough to permit bank conflicts, and one for buffers of a size that doesn't.

For the case where the buffer is large enough to permit bank conflicts, a worst-case sequence of DRAM commands is one where every pair of consecutive requests is for a different row in the same bank. This results in the DRAM controller issuing an expensive (activate,read/write,precharge) *row cycle* for every word. For reads, the LID is $n*(tRAS+tRP)$, while for writes the LID is $n*(tRCD+tCWD+tBURST+tWR+tRP)$.

For buffers of smaller size, LID and WCRET can be determined the same way unit-stride transfers are characterised, by breaking the latency down into the warm-up, throughput and drain phases.

The length of the warm-up phase depends on the number of row activations on the critical path. This in turn is bound by the size of the buffer to be addressed, for which I identified three cases: buffers spanning one row, buffers spanning one row per bank-group, and buffers spanning more than one row per bank-group. The remainder of this subsection presents the relevant equations for the two bank-group DRAM configuration, followed by a separate analysis for the configuration with four bank-groups. For readability, I define the number of rows accessed for a given buffer size s in words, assuming worst-case alignment, as:

$$rows(s) = \begin{cases} 1 & \text{iff } s \leq 1 \\ 2 & \text{iff } 1 < s \leq nBW + 1 \\ 2 + \left\lceil \frac{s-(nBW+1)}{2*nC} \right\rceil & \text{otherwise} \end{cases} \tag{5.7}$$

The simplest case occurs when a buffer size only permits addressing a single row in a single bank. With Sim-D's address mapping, this can only occur in the pathological case of a one-word buffer. In this case, one activate appears on the critical path, hence only a latency of $tRCD$ is paid in the warm-up phase. Since all CAS operations target the same bank, the latency of the throughput phase is determined by $(n-1)*tCCD_L$.

The case where an indexed buffer request spans one row in multiple banks, but no more than one per bank-group, is slightly worse. Given Sim-D's address mapping, this case occurs when $rows(s) \leq nBG$. In our example DRAM system with $nBG = 2$, this corresponds with a buffer of size between 2 and 17 words. For this case, exhaustive simulation of requests with $n = 16$ confirms that the worst case occurs for example when the first request accesses bank 0 and all remaining requests access bank 1. This pattern maximises activation delay while paying the long (same bank-group) column-to-column delay as much as possible during the throughput phase. The resulting schedule is displayed in Figure 5.11

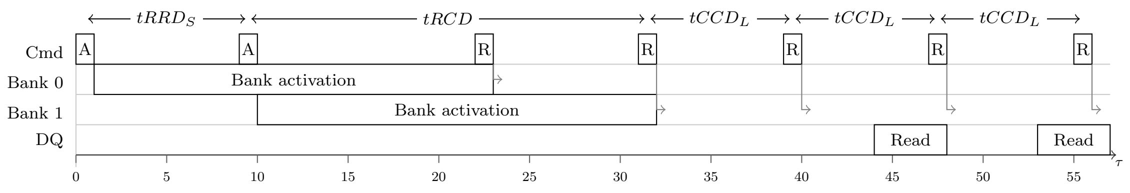


Figure 5.11: Partial DRAM schedule for worst-case iterative indexed read, 2 banks.

As this schedule shows, the warm-up phase now needs to activate both banks. Since the address mapping ensures that these banks must come from different bank groups, this delay is equal to $tRRD_S + tRCD$.

Compared to the single-bank case, the throughput phase is shortened by exactly $tCCD_L$. This is because the first read, to bank 0, is issued in parallel with the activation of bank 1. Hence the throughput phase will take $(n - 2) * tCCD_L$ cycles.

Note that a perfect oracle memory controller would benefit from activating bank 1 before bank 0, as this allows to process more reads in the warm-up phase in parallel with row activation. However, the knowledge that bank 1 must be activated only becomes available at time $T = 1$. The greedy command arbiter will instead at $T = 0$ decide that the activation of bank 0 is the best option, by virtue of being the only option.

Analysis shows that, for sufficiently large n, requests that maximise precharge delay are no worse than requests that maximise activation delay. The reason is two-fold. Firstly, there is no minimum distance between two precharge commands to different banks like there is between two row activation commands. Secondly, for sufficiently large n the depth of the per-bank command buffers rules out paying $tRAS$ on the critical path in the drain phase. For the DDR4-3200AA timing parameters presented in Section 4.1.1, $tRAS = 6.5 * tCCD_L$. As long as the per-bank command FIFOs can contain more than 6 entries each, all activate commands must be available $tRAS$ cycles before the last precharge. This means that $tRAS$ cycles pass while the last 6 reads to the final bank are performed, keeping this delay off the critical path. In the previous section I already suggested that per-bank command FIFOs of 16 entries can help with the throughput of specific non-unit stride patterns, providing a safe margin for disregarding such precharge-maximising access patterns when determining the LID and WCRET.

For an indexed iterative request on a buffer that spans multiple rows in the same bank-group, a slightly worse worst-case emerges. Exhaustive search reveals the following adversary pattern of targeted banks, assuming banks 0 and 2 are in the same bank group:

$$0, ((tRRD_L - 1) \times 2), 1, 2, 2, 2 \ldots$$

Figure 5.12 demonstrates the first 58 cycles of the resulting schedule.

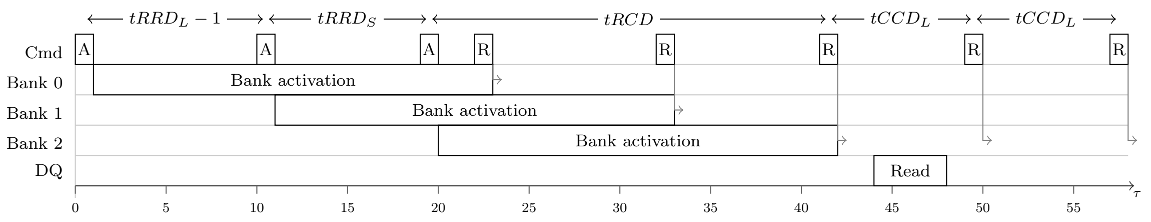


Figure 5.12: Partial DRAM schedule for worst-case iterative indexed read, 3 banks.

This pattern exploits another weakness in the greedy algorithm. Under normal circumstances, the activation of bank 1 can happen as early as $tRRD_S$ cycles after the activation of bank 0. However, because the pipeline only feeds the command arbiter one read request per cycle, this activation can be postponed to time $T = tRRD_L - 1$ by instead requesting $tRRD_L - 1$ words from bank 2 before the first request from bank 1. Because bank 2 is in the same bank group as bank 0, it is not activated during any of these cycles. The only command ready to be issued at time $tRRD_L - 1$ is thus the activation of bank 1, further delaying the activation of bank 2 whose reads dominate the critical path of the throughput phase. Impact from this decision is reduced slightly only by the fact that bank 2 is now in a different bank group than the previous, and thus can be activated $tRRD_s$ time-units after activated bank 1.

Because $tRRD_S + tRRD_L - 1 > tRRD_S * 2$, this pattern is worse than an extension of the pattern found for two banks. Furthermore, any other position of the index mapping to bank 1 in the list will lead to a better execution time. Earlier would lead to an earlier execution of the activation of bank 1, while later in the list will result in bank 2 being activated first, enabling parallel execution of the activation of bank 1 with several reads from bank 2.

For these buffer sizes, the worst-case warm-up phase requires $tRRD_S + tRRD_L - 1$ cycles. Given this pattern permits another read to be performed during the warm-up phase, the worst-case throughput phase is determined by $(n - 3) * tCCD_L$.

For a buffer spanning beyond four banks, the command arbiter's fifth prioritisation rule (activation of banks with many requests in its FIFO over those with fewer requests), combined with the depth of the per-FIFO buffers, has the implication that after three activations, there will always be at least two request ready to be issued in parallel with another activation. To see why, consider the following sequence of targeted banks:

$$0, 2, 2, 2, 4, 4, 4, 6, 6, 6, 1, 1, 1, \ldots$$

When the command generator activates a row in each of these banks, one per cycle, the resulting command schedule is similar to that of the worst case for three banks. An activation for bank 0 is issued in cycle 0, and an issue for bank 1 is issued in cycle 10. At this point the third activation must be one of bank 2, 4, 6 or 8, each of which have at least two requests ready. In fact, one of these is guaranteed to have three entries as $\lceil \frac{tRRD_L - 1}{3} \rceil = 3$, in this case bank 6. Because $tRRD_S + tRRD_L - 1 + 3 * tCCD_L > tRRD_S + 2 * tRRD_L$, the fourth activation is no longer on the critical path as it can be successfully executed in parallel with these three read operations. By the fourth activation, the command arbiter's fifth prioritisation rule guarantees that there are always sufficient read/write requests available to mask the latency of all activation related delays, including the four-activate window (FAW).

Any different set of indexes, not following the worst-case pattern for three banks, ensures that at the time instances of any bank activation beyond the first there are always two requests available, either in the newly activated bank or to one of the previously activated banks. These requests can always be parallelised with subsequent row activation latencies. Furthermore, the variation in rows accessed by an activation-latency-maximising pattern introduces more opportunities to schedule adjacent read or write requests from alternating banks, reducing the length of the throughput phase.

Using these observations, I define a function that computes the worst case latency for the warm-up and throughput phases of index-iterative requests as:

$$IIACTCAS(n,s) = \begin{cases} tRCD + (n-1) * tCCD_L & \text{iff } rows(s) = 1 \\ tRCD + tRRD_S + (n-2) * tCCD_L & \text{iff } 1 < rows(s) \leq nBG \\ tRCD + tRRD_L + tRRD_S - 1 + \text{(n-3) * tCCDL} & nBG < rows(s) \leq nB \end{cases} \tag{5.8}$$

For all buffer sizes, the drain phase is characterised in the same way as it is for unit-stride transfers, thus $tRTP + tRP$ and $tCWD + tBURST + tWR + tRP$ for the LID of reads and writes respectively, and $tCAS + tBURST$ and $tCWD + tBURST$ for the respective WCRETs. This results in the following equations for LID and WCRET:

$$\begin{aligned} tIILID_r(n,s) &= \begin{cases} \text{IIACTCAS(n,s) + tRTP + tRP} & \text{iff } rows(s) \leq nB \\ \text{n * (tRAS+tRP)} & \text{otherwise} \end{cases} \\ tIIWCRET_r(n,s) &= \begin{cases} \text{IIACTCAS(n,s) + tCAS + tBURST} & \text{iff } rows(s) \leq nB \\ \text{(n-1) * (tRAS+tRP) + tRCD + tCAS + tBURST} & \text{otherwise} \end{cases} \\ tIILID_w(n,s) &= \begin{cases} \text{IIACTCAS(n,s) + tCWD + tBURST + tWR + tRP} & \text{iff } rows(s) \leq nB \\ \text{n * (tRCD+tCWD+tBURST+tWR+tRP)} & \text{otherwise} \end{cases} \\ tIIWCRET_w(n,s) &= \begin{cases} \text{IIACTCAS(n,s) + tCWD + tBURST} & \text{iff } rows(s) \leq nB \\ \text{(n-1) * (tWR+tRP) + n * (tRCD+tCWD+tBURST)} & \text{otherwise} \end{cases} \end{aligned} \tag{5.9}$$

5.4.3.1 Four or more bank groups

Re-evaluating the three cases for four bank groups results in interesting findings that again permits to simplify the $IIACTCAS(n,s)$ definition.

Looking at the second case, where buffers span multiple rows but only a single row per bank-group, for the two bank-group case a worst-case was found where $tCCD_L$ latency could be traded for $tRRD_S$. Table 4.1 shows that $tRRD_S > tCCD_L$ for a two bank-group configuration, hence this results in worse timing. The same is not true for the DRAM chips containing four bank-groups. Hence, the pattern that describes the worst-case latency for the single-row case is also the worst case latency for the warm-up and throughput phases for buffers that span multiple rows.

Similarly, the pattern that emerged for the case where a buffer spans multiple rows in the same bank-group is worse than that of the single-row case for chips with 2 bank-groups because $tRRD_L > tCCD_L$. Again, this is not true for the DRAM chip containing four bank-groups.

As a result, the definition for $IIACTCAS(n, s)$ can be greatly simplified:

$$IIACTCAS_{4BG}(n, s) = tRCD + (n - 1) * tCCD_L \quad \text{iff } rows(s) < nB \tag{5.10}$$

To derive the LID and WCRET, this equation can simply replace $IIACTCAS(n, s)$ in Equation 5.9.

5.4.4 Snoopy indexed transfers

From the DRAM controller's point of view, a CAM-based snoopy indexed transfer is indistinguishable from either a unit- or a non-unit-stride transfer. Therefore, methods described in Sections 5.4.1 and 5.4.2 can be used to derive the WCRET and LID.

5.4.5 Takeaway points

This section presented methodology for deriving the LID and WCRET of DRAM requests processed by Sim-D's DRAM controller. Specifically, I contributed the following:

- Equations to bound the LID and WCRET for large unit-stride transfers and snoopy indexed requests that stream a whole buffer or a 1D block within a buffer,
- A simulation method for bounding the LID and WCRET for non-unit-stride transfers and snoopy indexed requests that stream a 2D block within a buffer,
- Equations to bound the LID and WCRET for iterative indexed transfers.

Additionally, I presented the worst-case DRAM command schedules of iterative indexed transfers. justifying the low worst-case performance of such transfers.

5.5 Evaluation

In this section I evaluate the worst-case performance of the different types of requests supported by the Sim-D DRAM controller. In Sections 5.5.1 and 5.5.2 I compare the performance of unit- and non-unit-stride transfers against the theoretical throughput of pattern-based DRAM controllers like Predator [40]. Section 5.5.3 presents a comparison between the iterative- and the snoopy indexed transfer methods. Unless otherwise specified, evaluation is performed using the DDR4-3200AA configurations, for which parameters can be found in Section 4.1.1.

I omit comparing Sim-D against a generic throughput-optimised DRAM controller for CPUs, as their use-cases are incomparable. Sim-D's DRAM controller is a specialised component for HRT SIMD processors and optimised towards processing large requests. It is expected to perform poorly when used for the single-burst accesses, which is the normal mode of operation in application processors. Conversely, throughput-optimised DRAM controllers for CPUs are incapable of meeting Sim-D's requirement of processing requests at a bound latency.

5.5.1 Unit-stride transfers

Recall from Section 5.4.1 that the LID of a unit-stride transfer, processed by Sim-D's DRAM controller, has a monotonic relation with the size of the requested data. To quantify the efficiency of these transfers, Figure 5.13 shows the utilisation of Sim-D's contiguous memory transfers of sizes between 4B and 16KiB as a fraction of ideal throughput. For comparison, the utilisation of different configurations of a pattern-base DRAM controller under alignment assumptions, as presented in Section 5.1.2, are included in the graph. By contrast, the LIDs used to calculate Sim-D's data bus utilisation only assumes requests are aligned to a 32-bit boundary.

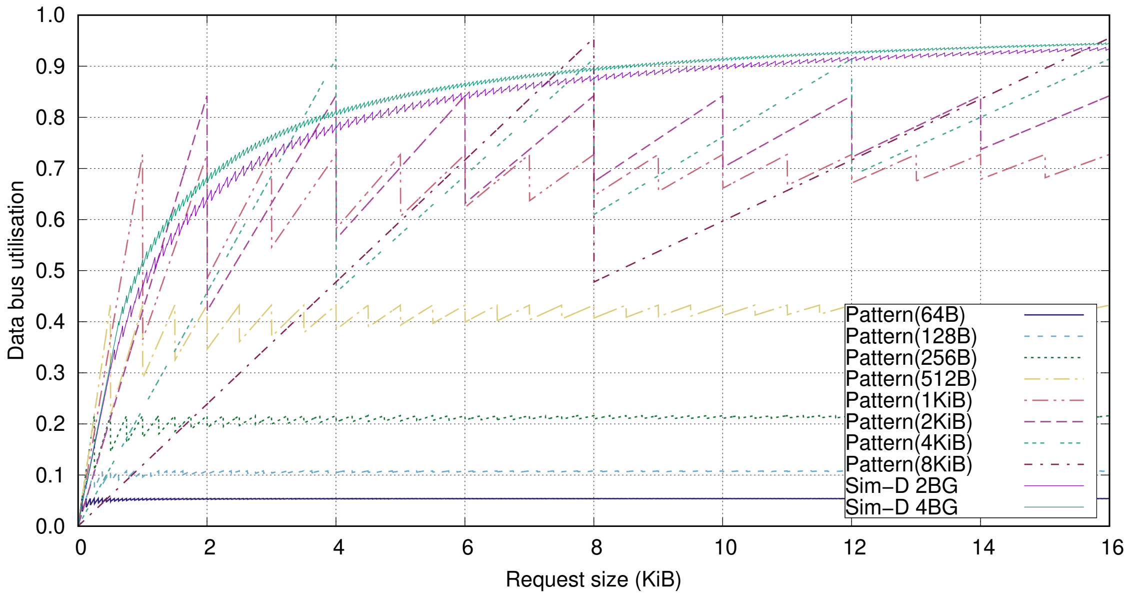


Figure 5.13: Data bus utilisation on Sim-D vs. pattern-based DRAM controller, DDR4 3200AA.

As a first observation, Sim-D's data bus utilisation trend follows a much finer grain sawtooth than the pattern-based DRAM controller configurations. This is caused by Sim-D never requesting a burst whose data will be discarded entirely. On the other hand. a static pattern-based controller configured with patterns of n bursts can end up discarding all data of up to $n-1$ bursts. This more uniform correlation between transfer size and DRAM bus utilisation makes Sim-D a favourable choice for workloads containing a variation of transfer sizes.

Secondly, the four bank-group DRAM configuration consistently out-performs the configuration with two bank groups. This is both the result of lower row-activate to row-activate delay (RRD), as well as an increase in bank-groups making it more likely to hit the short CAS to CAS delay (CCD) and row-activate to row-activate delay (RRD) latencies in the worst case. Although returns diminish as transfers grow larger, a ~3% benefit is observed for 4KiB transfers.

Finally, whenever there is a pattern-based DRAM controller configured for a given size, Sim-D performs slightly worse at each pattern-configuration's best case. This can be explained by the penalties paid for facilitating unaligned access and the 3-cycle pipeline warm-up required by Sim-D's memory controller. In return, Sim-D reclaims a lot of the performance in situations where pattern based DRAM command scheduling results in a low net-bandwidth.

To show this observation in greater detail, Figure 5.14 present heat bars that visualise the absolute difference in bus utilisation between Sim-D's two bank-group configuration and each pattern-configuration. In this figure, the colour indicates whether Sim-D outperforms a pattern-based DRAM controller for a given transfer size (green) or vice-versa (blue).

The more intense the shade, the larger the absolute difference in bus utilisation. The difference between the two is clamped between $(-0.5, 0.5)$ for legibility.

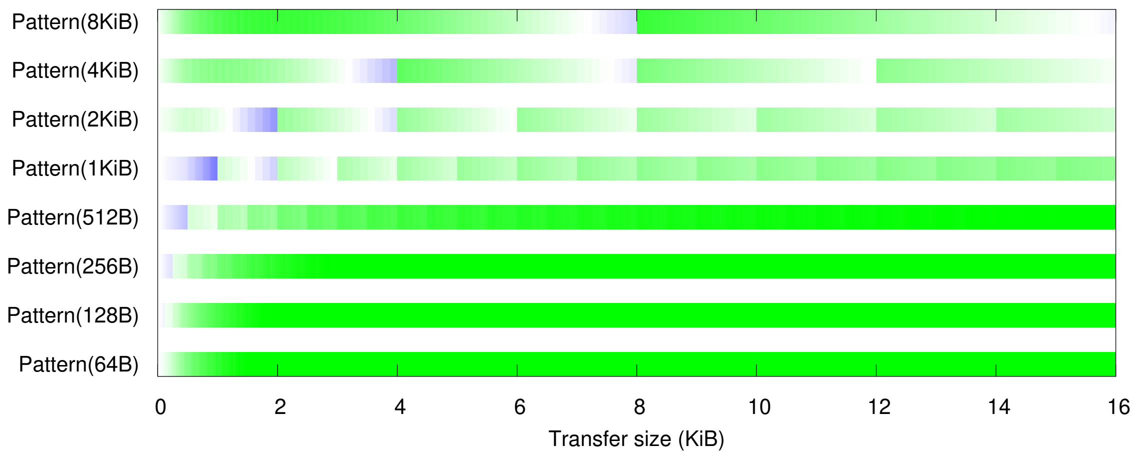


Figure 5.14: Pattern-based DRAM controller configuration vs. Sim-D, DDR4-3200AA with 2 bank-groups

This diagram highlights four interesting trends. Firstly, for every configuration of a pattern-based DRAM controller, there is at least one window in which it outperforms Sim-D. The advantage within the window(s) of pattern-based DRAM controllers is biggest in the configurations for 16- and 32 bursts. For larger bursts, the fixed-overhead paid by Sim-D loses dominance in the total cycle time, whereas for smaller patterns the performance for both memory controllers is so poor that there is no significant observable difference in bus utilisation.

Secondly, patterns up to 8 bursts (512B/transfer) are quickly out-performed by Sim-D. These patterns perform relatively well for transfers below their configured burst size as they can utilise the performance benefits of their stricter alignment constraints, but once exceeding their optimum they are consistently outperformed by Sim-D as their performance converges to a relatively low peak on modern DDR4 DRAM.

Thirdly, the larger pattern configurations each have three windows in which they outperform Sim-D. For Pattern(n), these windows end at $64 \times n$, $128 \times n$ and $192 \times n$ bytes. The absolute advantage gets successively smaller with each window as the bus utilisation of Sim-D continues to climb, but as both Sim-D and the pattern based memory controller converge to their optimum performance, the gradients will start repeating themselves eventually.

Finally, as represented by the saturation of the colours, the absolute benefit of pattern-based solutions at their optimum is not as significant as their disadvantage at other points. What this means in practice should be studied on a per use-case basis, as Sim-D's configuration is optimised to issue requests of 4KiB for the common case of DRAM$\leftrightarrow$RF transfers. However, this data indicates that throughput-wise Sim-D's memory controller is a strong all-round solution for large transfers in a HRT SIMD processor.

The performance advantage of Sim-D's DRAM controller comes despite only enforcing alignment of transfers to 32-bit word-boundaries. Figure 5.15 overlays Sim-D's performance curves over the performance of pattern-based DRAM controller configurations under equal alignment assumptions, as previously shown in Figure 5.3. Figure 5.15 clearly demonstrates the throughput performance advantage of Sim-D's deterministic DRAM command scheduling solution over the more rigid static DRAM command patterns.

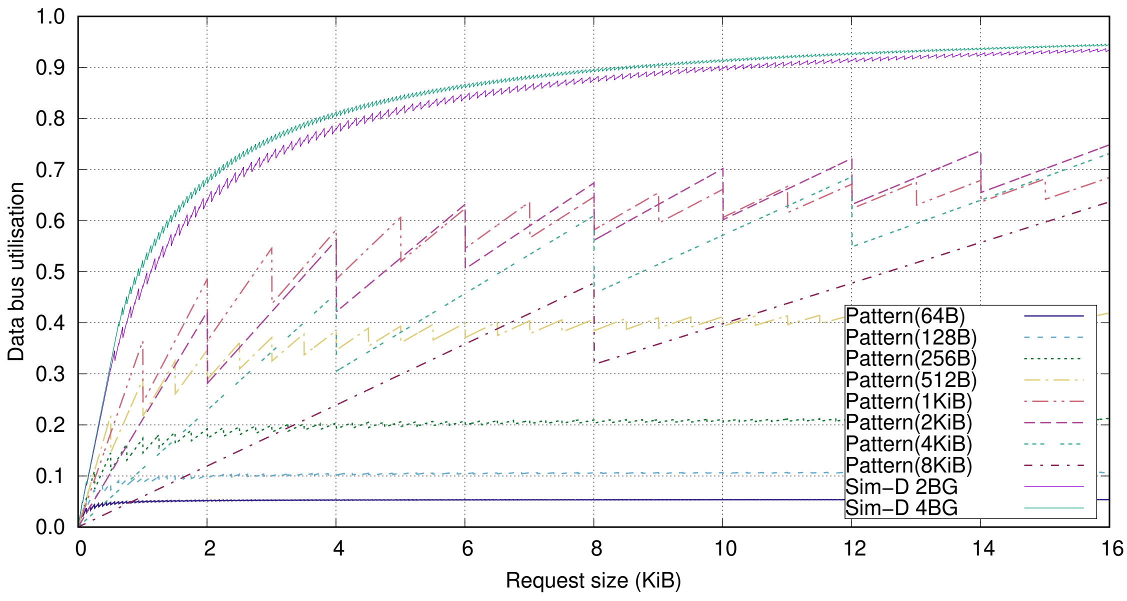


Figure 5.15: Data bus utilisation on Sim-D vs. pattern-based, alignment to 4B boundaries, DDR4 3200AA.

5.5.2 Non-unit-stride transfers

Section 2.2.3 discussed how some kernels (e.g. filter kernels) employ tiling as an optimisation technique to reduce pressure on the DRAM bus. To transfer a 2D tile of data into local memory, a non-unit stride transfer is issued to the DRAM controller. This section presents some evidence that Sim-D's memory controller can efficiently handle such transfers.

To this end, Tables 5.7, 5.8 and 5.9 compare the worst-case LID of Sim-D against that of different pattern-based DRAM controller configurations for three different filter size- and pitch combinations. For each experiment, the buffer width is set to the number of elements a work-group configuration of (1024,1) requires. Latencies reported for the Sim-D DRAM controller include a 3-cycle pipeline overhead. No pipeline overhead is assumed for the pattern-based DRAM controller as DRAM command scheduling is static. Reported energy consumption numbers are estimates from DRAMPower [82], and do not include DRAM controller overhead.

	Net	Sim-D (min-max)		Predator (n-burst fixed size, cycles)							Best
Config	wrds	Cycles	Energy(nJ)	n = 1	2	4	8	16	32	64	nJ
(1024,1)	3078	830 - 840	182.2 - 188.4	14282	7178	3626	1850	1144	**1064**	1120	282.6
(512,2)	2056	586 - 608	126.9 - 133.8	9842	5106	2738	1554	**1144**	1368	1400	335.4
(256,4)	1548	466 - 496	104.8 - 112.0	7622	4070	2294	1406	1144	**1064**	1960	282.6
(128,8)	1300	**418 - 456**	**98.9 - 107.2**	6734	3774	2294	1554	**968**	1672	3080	284.0
(64,16)	1188	418 - 492	108.9 - 118.9	6734	4070	2738	**1406**	1672	2888	5320	368.9
(32,32)	**1156**	466 - 560	139.8 - 152.3	7622	5106	3774	**2590**	3080	5320	9800	678.9
(16,64)	1188	591 - 755	207.3 - 225.6	9842	7326	6142	**5550**	5896	10184	18760	1453.8
(8,128)	1300	844 - 1148	302.2 - 373.4	14430	12062	10878	**10286**	11880	19912	36680	2693.7

Table 5.7: 3*3 filter, pitch 1, image width (period) 1026 words, DDR4-3200AA, $nBG = 2$

	Net	Sim-D (min-max)		Predator (n-burst fixed size, cycles)							Best
Config	wrds	Cycles	Energy(nJ)	n = 1	2	4	8	16	32	64	nJ
(1024,1)	6147	1598 - 1608	361.2 - 367.4	28490	14282	7178	3626	2200	1976	**1960**	485.5
(512,2)	5125	1358 - 1383	311.8 - 318.1	24124	12284	6364	3404	**2288**	2432	3080	669.7
(256,4)	4617	1246 - **1312**	**296.5 - 304.4**	22052	11396	6068	3404	**2464**	2888	2800	721.1
(128,8)	4369	**1214** - 1313	309.2 - 319.0	21460	11396	6364	3848	3080	**2736**	5040	724.9
(64,16)	4257	1270 - 1380	357.7 - 368.3	22052	12284	7400	4958	**2992**	5168	9520	875.4
(32,32)	**4225**	1358 - 1462	461.9 - 472.1	24124	14504	9694	**4958**	5808	10032	18480	1298.9
(16,64)	4257	1707 - 1900	683.3 - 697.9	28712	19166	14356	11988	**11440**	19760	36400	3343.9
(8,128)	4369	3158 - 3264	1179.1 - 1190.0	38110	28564	23828	**21460**	24112	39216	72240	5619.1

Table 5.8: 3*3 filter, pitch 2, image width (period) 2049 words, DDR4-3200AA, $nBG = 2$

	Net	Sim-D (min-max)		Predator (n-burst fixed size, cycles)							Best
Config	wrds	Cycles	Energy(nJ)	n = 1	2	4	8	16	32	64	nJ
(1024,1)	5140	1350 - 1359	304.3 - 310.5	23828	11914	5994	3034	1848	**1672**	1680	443.4
(512,2)	3096	850 - 872	191.7 - 199.3	14726	7622	4070	2294	**1672**	1976	1960	489.7
(256,4)	2080	606 - 616	135.8 - 142.7	10064	5328	2960	1776	1408	**1216**	2240	322.8
(128,8)	1584	490 - **520**	**115.2 - 123.8**	7992	4440	2664	1776	**1056**	1824	3360	309.7
(64,16)	1360	**458** - 544	118.0 - 130.8	7400	4440	2960	1776	**1760**	3040	5600	515.4
(32,32)	**1296**	490 - 620	145.6 - 163.1	7992	5328	3848	3256	**3168**	5472	10080	926.8
(16,64)	1360	608 - 784	212.1 - 236.6	10064	7400	6216	**5624**	6336	10336	19040	1473.2
(8,128)	1584	855 - 1147	347.1 - 389.9	14652	12136	10952	**10360**	11968	20368	36960	2713.1

Table 5.9: 5*5 filter, pitch 1, image width (period) 1028 words, DDR4-3200AA, $nBG = 2$

From these results, three conclusions can be drawn. Firstly, Sim-D's best tile configuration consistently outperforms all configurations for a pattern-based DRAM controller. Sim-D's most optimistic worst case is between 49.4% and 112% faster than the pattern-based DRAM controller's best case, with a similar improvement in estimated energy consumption. This can be attributed to the fact that Sim-D's DRAM controller benefits maximally from bank-hits while not issuing requests for bursts whose data is never read. By contrast, the pattern-based DRAM controller configured with low burst count DRAM command schedules fail to benefit from bank hits, whereas a high burst-count configuration causes many bursts to be issued for words that lie outside a requested tile of data.

Note that stride patterns like the ones evaluated are bad for pattern-based DRAM controllers because the number of words read per row only barely exceeds power-of-two boundaries. For large patterns in particular, this means a lot of net bandwidth is wasted on issuing bursts for which all words are subsequently discarded. However, looking at the design of commodity GPUs it is a fair assumption that SIMD accelerators function best with a power-of-two number of work-items per work-group, hence the tested cases have relevance for practical applications.

Secondly, for non-unit stride transfers a pattern-based DRAM controller performs best when configured with a pattern that issues between 16 and 64 bursts, transferring between 1 and 4KiB per pattern transaction. However, such coarse-grain pattern transactions prevent these DRAM controllers from performing well when processing data-conserving tiling configurations: the data bus utilisation for the (128,8) case does not exceed 39.9%. For the two filters with a pitch of 1, the higher utilisation achieved with less data-conserving tiling configurations are the result of loading more words rather than reducing the number of cycles.

Finally, the most data-conserving tiling configuration is the square (32,32) configuration. However, Sim-D's DRAM controller conserves the most cycles for a more oblong tiling configuration. To give insight into the reason behind this discrepancy, Table 5.10 shows the number of activate and burst read DRAM commands issued for each tiling configuration in the worst case for the 3×3 filter example with a pitch of 1.

	Net	Cycles	DRAM cmds (max)		Bus util.
Config	words	min - max	Activate	Burst read	%
(1024,1)	3078	830 - 840	**4**	194	91.6
(512,2)	2056	586 - 608	**4**	133	84.5
(256,4)	1548	466 - 496	6	103	78.0
(128,8)	1300	**418 - 456**	8	**92**	71.3
(64,16)	1188	418 - 492	12	93	60.4
(32,32)	**1156**	466 - 560	20	107	51.6
(16,64)	1188	591 - 755	36	141	39.3
(8,128)	1300	844 - 1148	68	212	28.3

Table 5.10: Number of cycles, activate and burst read commands, 3*3 filter, pitch 1, image width (period) 1028 words, DDR4 3200AA

As this table shows, there are two reasons behind this discrepancy. The first reason is that the more vertically oblong a tile is, the more DRAM rows it spans. This is reflected by a higher number of row activations. As long as a sufficient number of burst requests are issued to each row, these activations can be performed in parallel with burst read/write operations. However, for vertically oblong tiles this may no longer be the case.

The second reason is that for each stride pattern, the *words_period* parameter slightly exceeds a perfect multiple of the burst size. The more periods in a stride request, the more often a penalty is incurred for issuing a burst request for a limited number of words. For this reason, the smallest number of burst requests is achieved with the (128,8) configuration. Beyond this point, the number of issued burst reads start increasing again despite a decrease in the net number of words read. The ideal tiling configuration is thus a trade-off between reducing the number of net words and maintaining good data locality.

The last column in Table 5.10 translates the worst-case number of cycles to a net DRAM data bus utilisation percentage. For the best tiling configuration, the bus utilisation is 71.3%. To place this number in perspective, a unit-stride transfer of 1024 words takes 327 cycles in the worst case, achieving a worst-case data bus utilisation of 78.3%. This means that Sim-D's non-unit-stride transfers can be of practical value to applications.

5.5.3 Indexed transfers

As explained in Section 5.3.2, (indirect) indexed transfers inherently perform poorly in the worst case. A particularly bad set of requested indexes is one where every index maps to the same bank group, and for every DRAM burst only a single word is used in practice. Equation 5.3 defines an upper bound on worst-case data bus utilisation of $\frac{tBURST}{nBW * tCCD_L}$, which for the DDR4-3200AA configurations is $\frac{4}{16*8} = 3.125\%$.

When the size of the buffer, or the size of the relevant region within a buffer, is known a priori and is relatively small, better results can be achieved. Sim-D implements two schemes for indexed accesses: index iteration and snoopy indexed transfers. This section compares both techniques under the two DDR4-3200AA configurations explained in Section 4.1.1.

For a given DRAM configuration, the worst-case latency of indexed transfers can be characterised for four discrete ranges of buffer sizes using Equations 5.9. Table 5.11 lists the resulting worst-case latencies and data bus utilisation percentages for indexed iterative requests on the two- and four-bank group DDR4-3200AA configurations.

Buffer-	2 Bank-groups					4 Bank-groups				
span	Buffer size	LID (n=1024)				Buffer size	LID (n=1024)			
Rows	Max (B)	R	%	W	%	Max (B)	R	%	W	%
1	4	8243	3.11	8271	3.10	4	8243	3.11	8271	3.10
$\leq nBG$	8-68	8244	3.11	8272	3.09	8-16452	8243	3.11	8271	3.10
$\leq nB$	72-49220	8246	3.10	8274	3.09	16456-114756	8243	3.11	8271	3.10
$> nB$	49224-	75776	0.33	90115	0.28	114760-	75776	0.33	90115	0.28

Table 5.11: Worst-case latency and data bus utilisation for indexed iterative DRAM read and write requests

Three observations can be made. Firstly, the latency differences for the first three buffer size ranges are minimal. For the two-bank-group configuration the difference between the first and third case, buffers spanning 1 row versus spanning 8 or fewer rows, is merely 3 cycles. For the four bank-group configuration no difference exists in LID, as a result of simplifying the $IIACTCAS(n, s)$ definition in Equation 5.10.

Secondly, for small and medium-sized buffers, the data bus utilisation approaches the theoretical bound of 3.125%. For buffers whose size exceeds one row per bank, permitting a worst-case access pattern causing row ping-pong, worst-case utilisation drops to less than 0.34%.

Finally, the difference in latency between the two- and four-bank-group configuration for the different cases is negligible. However, the four-bank-group configuration is more resilient to variations in buffer size as a result of doubling the number of banks without changing the size of each bank. Where the two bank-group configuration can sustain its higher performance mode for buffers up to ∼48KiB, the four bank-group configuration sustains this performance for all buffers up to ∼112KiB.

Snoopy indexed transfers have in a much more fine-grain correlation between buffer size and latency. To compare the two indexed transfer methods, Figure 5.16 depicts the correlation between buffer size and absolute read latency for both methods.

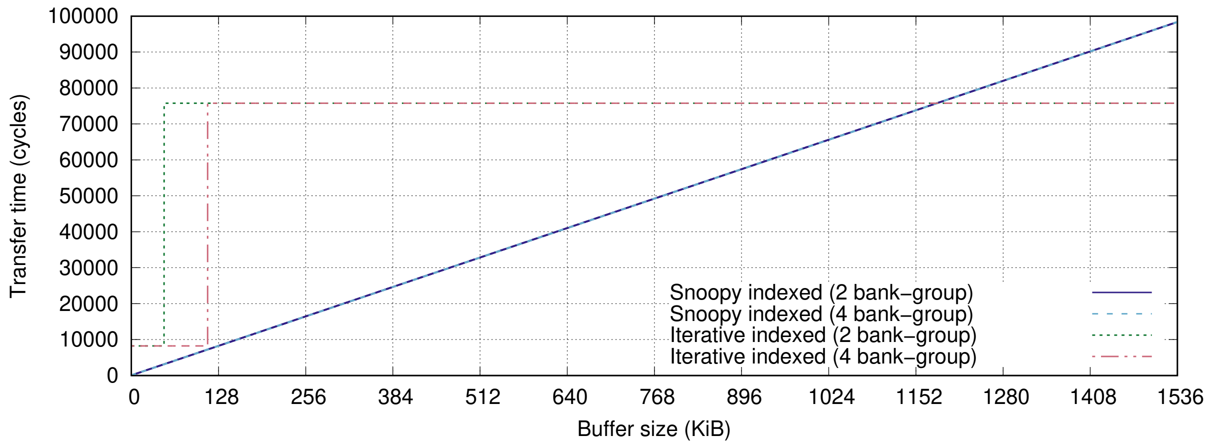


Figure 5.16: Worst-case latency for iterative- and snoopy indexed reads.

This figure shows that for both the 2- and 4-bank-group DRAM configurations, snoopy indexed transfers remain faster than iterative indexed transfers for small- and medium-sized buffers. The cross-over point where iterative indexed transfers become more efficient is for buffers of over ∼1.15MiB, corresponding to a square buffer exceeding 550 × 550 data elements.

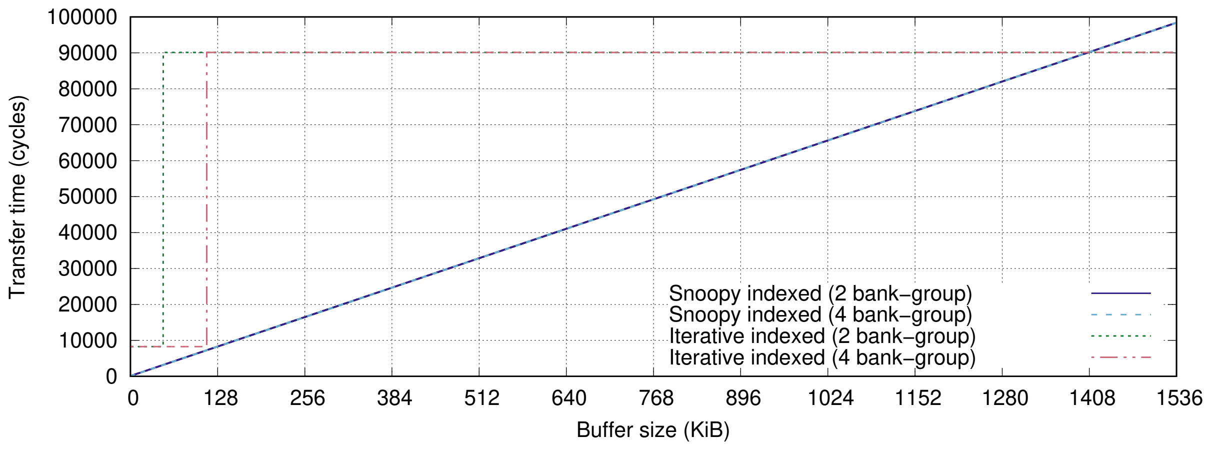


Figure 5.17: Worst-case latency for iterative- and snoopy indexed writes.

For write-operations the same trend is observed. The higher worst-case LID for iterative indexed write transfers causes the cross-over point to shift to buffers exceeding ∼1.37MiB.

It is worth re-iterating that snoopy indexed transfers permit to narrow down the region of a buffer containing the requested indexes, provided this region can be described by a (non-)unit stride pattern. If the use-case permits, bounding this region can result in a lower LID. This makes snoopy indexed transfers worth considering even when the buffer size exceeds the cross-over points.

5.6 Summary

In this chapter I presented a closed-page DRAM controller suitable for a HRT wide-SIMD architecture. Its defining feature is the ability to service large, explicitly coalesced requests in bound time. The DRAM controller is designed as a pipeline. In the front-end, I proposed two subcomponents: the *stride sequencer* and the *index iterator*. Both subcomponents translate a specific type of request into a sequence of DRAM burst requests. Subsequent pipeline stages deterministically schedule these burst requests for maximum throughput. This determinism helps to bound the execution time of each request.

The *stride sequencer* processes 1D or 2D stride requests. Analysis of buffer usage by the selected benchmarks show that the majority of read and write requests can be expressed as such 1D or 2D stride transfers. To bound the execution time of a stride request, I provided various equations for different types of 1D stride transfers, and an exhaustive simulation method for 2D stride transfers that can run in time proportional to the number of columns in a pair of rows. Evaluation shows that a 1D block transfer can achieve over 78% data bus utilisation with DDR4-3200AA DRAM, and 2D block transfers can often nearly match this efficiency.

A limited number of benchmarks require indexed transfers for data dependent loads/stores. To service these use-cases, the *index iterator* iterates over a set of indexes into a buffer and requests the associated bursts one by one. Evaluation shows that such transfers perform poorly in the worst case, with an upper bound on data bus utilisation of 3.125%. To improve upon this bound for cases where the targeted (region of the) buffer is small, I introduced *snoopy indexed transfers*. These transfers work by streaming the contents of either a whole buffer or a (1D- or 2D) block of data from a buffer over the data bus. An array of CAMs, one per work-group monitor the addresses associated with the data currently on the data bus. On a match, it snoops the correct element off the bus. At the front-end, snoopy indexed transfers are 1D or 2D stride transfers, meaning their worst-case performance can be analysed using the same methods.

The evaluation of non-unit stride transfers shows how kernels that use tiling can benefit from optimising their tile dimensions. Simulated DRAM transfers show that the most data-conserving square tiling configuration is not the configuration with the lowest WCRET. This is caused by the generally higher number of required row activations for such a square transfer. In Section 7.5.3.3, I will put these results in perspective by means of a case-study, analysing the performance of the KFusion vertex2normal benchmark with various tiling configurations.

CHAPTER 6

Design space exploration

Sim-D's focus on HRT applications resulted in a high level architecture design that provides strict performance isolation between subcomponents. Within its high-level framework, there is a large design space to be explored. This design space spans parameters such as pipeline depth, number of SP-units and RCP-units, register file size, scratchpad size and scratchpad data bus width. These parameters inherently present a trade-off between performance and area.

This chapter aims to achieve two goals. Firstly, I justify several design decisions relating to resource provisioning by means of a parameter sensitivity study. Additionally, some more tailored performance-optimising features are discussed by contrasting them against alternatives. These studies are best interpreted as a limit study, guiding those who wish to implement some of Sim-D's principles in hardware. Secondly, I demonstrate the feasibility of the high-level Sim-D performance-isolation concepts by showing the achievable performance of several design points. The projected performance is put into context by comparing it against an NVIDIA Kepler-generation GPU of comparable specifications, as well as a high-end model from the same generation.

Given Sim-D is currently implemented as a simulator, rather than a synthesisable design, the scope of this chapter is largely limited to performance evaluation. In absence of area-, power- and cost models for Sim-D, the chosen parameter ranges used in this evaluation are mainly justified by their proximity to NVIDIA's hardware design decisions as evident from the specifications of real hardware. Some decisions are discussed in the context of hardware modelling techniques found in literature.

This chapter makes the following contributions:

- An analysis of the resource usage of benchmarks on both NVIDIA hardware and on Sim-D's simulation model, justifying the provisioning of data storage resources (Section 6.1),
- A parameter sensitivity study, characterising Sim-D's performance for a varying

number of configurations (Section 6.2),

- An average case performance comparison between Sim-D and an NVIDIA Kepler-generation GPU of similar specifications (Section 6.3),
- A benchmark-driven discussion of two specialised hardware-assisted optimisations: vector DRAM load/stores and warp trimming (Section 6.4).

6.1 Resource usage characterisation

This section characterises the resource usage of the benchmarks outlined in Section 4.3. These characterisations are the result of an analysis of two types of assembly code: NVIDIA Kepler-specific assembly as compiled by their binary drivers, and hand-written Sim-D assembly code. This analysis results in data that aids with the following design decisions:

- Provisioning of registers,
- Provisioning of SP-units in a SimdCluster,
- Provisioning of dividers, trigonometric operators,
- Instruction memory type.

6.1.1 NVIDIA Kepler

As a first experiment, I analyse benchmarks as compiled for NVIDIA's Kepler-generation hardware. To this end, static program data is gathered by running each benchmark's OpenCL implementation on an NVIDIA GeForce GTX650 GPU, and obtaining the compiled program and launch parameters using the "valgrind-mmt" and "demmt" [83] tools. The dynamic instruction count of benchmarks is then manually reconstructed from the assembly by analysing the control flow. Table 6.1 summarises the results of this data acquisition. The columns signify the following:

GPRs This column list the number of 32-bit general-purpose registers (GPRs) required for each work-item. NVIDIA Kepler does not distinguish between per-work-item vector registers and per-work-group scalar registers; the latter values are duplicated in all columns of a vector register. 64-bit pointers are contained in two adjacent GPRs.

Exec. # insns. The maximum number of (dynamic) executed instructions for a work-item when launched using the launch parameters defined for the specific benchmark. For if-else blocks, the common path is chosen. When no path is obviously dominant, the longest of the paths is chosen instead. Arithmetic instructions that directly address constant memory in one of their operands are counted as two instructions: one constant load/store, and one arithmetic instruction.

Binary The two *binary* columns list the size of the resulting binary both in number of instructions and in bytes. The latter is 8 times the number of instructions as on NVIDIA Kepler, each instruction requires 8 bytes of data. The resulting binary size does not include the overhead added by Kepler's static scheduling codes, as this overhead describes solely architecture-specific properties rather than properties of the benchmark program.

Div, Trigo The last three columns indicate whether a benchmark contains **I**nteger or **F**loating-point divisions and trigonometric instructions (sine, cosine).

Benchmark	Kernel	GPRs	Exec.	Binary		Div		Trigo
			# insn	# insn	B	I	F	
CNN	convolution	25	2317	133	1064			
	relu	9	52	43	344			
	maxpool	21	216	113	904			
KFusion	halfSampleRobustImage	21	152	84	672		√	
	depth2vertex	10	69	51	408			
	vertex2normal	24	268	248	1984		√	
	Track	24	282	254	2032		√	
SRAD	srad	20	207	167	1336	√	√	
	srad2	22	127	97	776	√		
	reduce	20	149	351	2808			
	reduce_fpatom	*18*	*186*	*42*	*336*			
MRI-Q	ComputePiMag	9	29	22	176			
	ComputeQ	20	17451	84	672			√
SPMV	spmv_jds_naive	21	2075	82	656	√		
LU Decomp.	diagonal	31	2718	244	1952		√	
	perimeter	34	608	803	6424		√	
	internal	15	104	89	712			
Stencil	naive_kernel	18	96	68	544			
FFT	GPU_FFT_Global[4]	23	120	106	848	√	√	√

Table 6.1: Program characteristics

The first thing to notice is that for all but the LU decomposition perimeter benchmark, 32 GPRs is sufficient to contain all required data without spilling into main memory. This is no coincidence: NVIDIA's compiler register allocation pass optimises for a register usage not exceeding 32, as a higher number of registers limits the numbers of warps the hardware scheduler can have in-flight at any given time. The fact that NVIDIA succeeds in optimising programs to fit within this register budget gives an indication that Sim-D probably requires a similar number of VGPRs and SGPR.

The larger GPR requirement for the LU Decomposition's perimeter benchmark is caused by aggressive loop unrolling. In Section 7.5.3.1 I show that this is an effective technique for predictably reducing the control flow overhead of certain kernels. For this reason, it is worth considering Sim-D configurations with a larger number of registers.

In justification of a Harvard architecture, Table 6.1 demonstrates that most kernels contain no more than a few kilobytes of code. Assuming a similar code density for Sim-D, this means that kernels can be contained in full in a typical contemporary L1I cache[1]. Considering the cost of a Harvard architecture is not larger than the cost of a comparable von Neumann architecture with an associative I-cache, while it provides the benefit of timing predictability, I believe that the limited storage requirement provides a compelling argument in favour of a Harvard architecture.

Looking at the type of instructions used, there appears to be a hard requirement for floating-point and integer division. Floating point divisions are used for many purposes. By contrast, integer divisions are only used for converting a multi-dimensional TID into a 1D offset into a given buffer. With Sim-D's work-group-wide DRAM requests and scalar operations, a single scalar divider suffices for performing the same operations.

Trigonometric operations are found in 2 of the 19 benchmarks. Despite this low occurrence, the low implementation cost and the high latency for software emulation of these instructions still provide a compelling motivation to support such operations in hardware.

To understand the significance of the dynamic instruction counts, it is worth breaking down the number of executed instructions into various categories. This break-down helps derive an initial estimate for the required number of processing units. To this end, I have manually annotated each dynamically executed instruction with its relevant category. Figure 6.1 provides an overview of the instruction mix of each benchmark, normalised to the total number of instructions executed in each. Instructions are categorised into nine different categories:

Int/bool and **FP** cover all regular integer plus binary (bit-wise) operations, and floating point operations respectively. When a conversion ("cvt" instruction) between floating point and integer values is encountered, the destination type determines the category.

Trigonometry refers to all instructions related to sine and cosine computation.

[1]For example, the ARM Cortex A72 mobile processor contains 48KiB of L1I per core [124].

Ld/st global and **Ld/st local** cover the regular loads and stores to global memory and local(/shared) memory respectively

Ld/st const covers all reads and writes to constant memory. Whenever a floating point or integer arithmetic instruction addresses constant memory as one of its operands, we count this as a ld/st const instruction *in addition to* an arithmetic instruction. Thus, for these operations two instructions are counted.

Atomic/reduction int and **Atomic FP** covers the integer+binary and floating point atomic operations respectively.

Ctrl flow covers all binary-, integer- and floating point test operations, (conditional) branches, join operations, memory fences, barriers and the final exit instruction of a program.

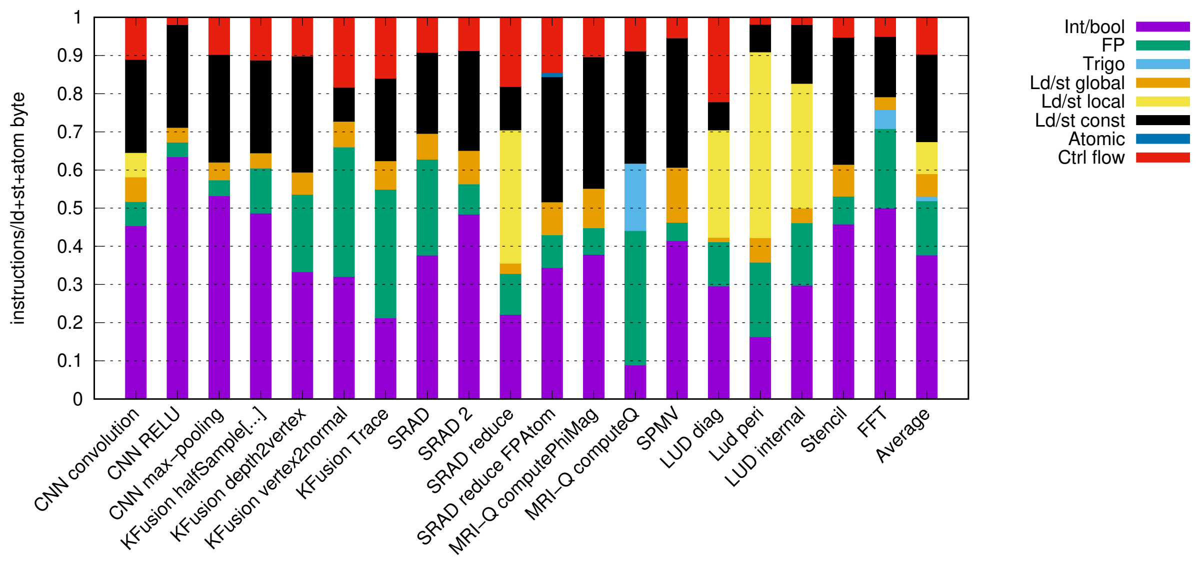


Figure 6.1: Dynamic executed # instructions, normalised

The first thing to notice is the large proportion of loads from constant memory. On NVIDIA hardware, many arithmetic operations are able to address a constant buffer directly in one of their source operands. The frequency of such operations and the number of times the same constant is loaded from a buffer hints at the existence of specialised constant cache techniques[2]. Such caching techniques risk making the execution time of program phases dependent on context, adding undesired complexity to WCET analysis

[2]Unfortunately I am unable to confirm caching of constant buffers from official NVIDIA hardware documentation, but constant buffer caching is documented for OpenCL programs targeting Intel FPGA hardware [125].

methods. An architecture tailored for HRT systems is likely to use different techniques, such as immediate values or persistent scratchpad values, to provide constant support with predictable latency.

To more closely match Sim-D's ISA, Figure 6.2 depicts the instruction mix for benchmarks when constant-memory load/stores are filtered out.

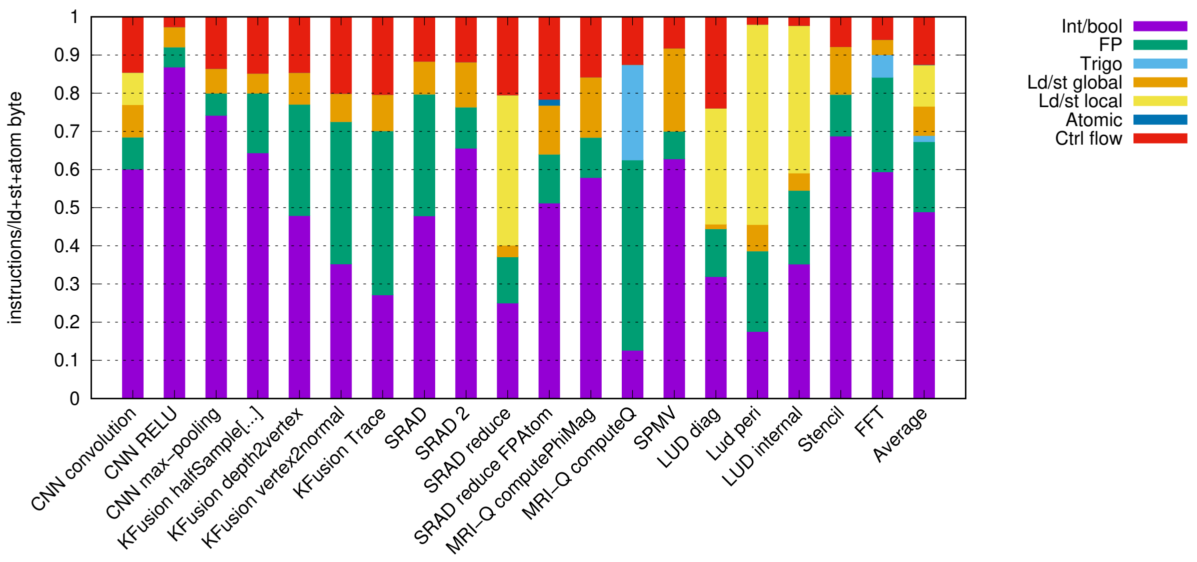


Figure 6.2: Dynamic executed # instructions, excluding constant memory accesses, normalised

Two interesting observations are highlighted. Firstly, the majority of the benchmarks are dominated not by floating point arithmetic, but rather by integer arithmetic. Most integer operations fall into two usage categories: pointer arithmetic and loop invariants. At first sight it seems that for good throughput, the number of ALUs should thus match or even exceed the number of FPUs.

Loop invariants are often equal for all work-items in a work-group. Sim-D can replace such vector integer arithmetic with scalar arithmetic, reducing the storage overhead.

Furthermore, Sim-D's reliance on implicitly-coalesced DRAM requests helps to eliminate pointer arithmetic or reduce this vector arithmetic to scalar arithmetic. However, this is not true for all pointer arithmetic. Kernels that perform indirect accesses, such as the SRAD and SPMV kernels, will compute per-work-item offsets within the buffers.

For such indexed accesses it is still required to support vector integer arithmetic in Sim-D. Such support is also beneficial for neural networks or computer vision use-cases that process e.g. unpacked RGB image data.

As a second observation, the two benchmarks that require trigonometric operations, MRI-Q and FFT, execute such instructions for 17.6% and 5% of all instructions respectively. Both benchmarks are examples of digital signal processing, that require conversion of data points between discrete points and the frequency domain.

To get a feel for the total number of resources that a SIMD architecture requires to match a given DRAM bandwidth target, Figure 6.3 presents the dynamic instruction count in terms of instructions executed per byte of DRAM data consumed. The number of bytes consumed by a work-item is estimated by multiplying the number of global load/store operations by their data size. This disregards the potential efficiency gains from coalescing requests in the (uncommon) case that multiple work-items access the same word, and should therefore be treated as an estimate only.

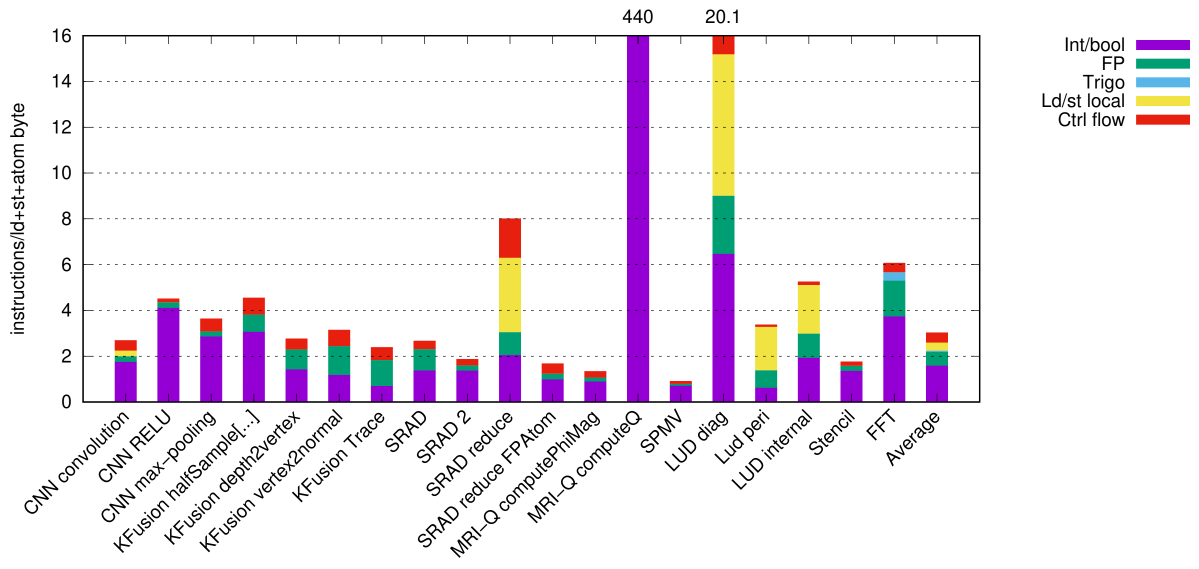


Figure 6.3: # instructions per byte of DRAM data consumed

From these numbers, a rough approximation of required processing units (SP-units) is derived as follows. Assume targeting a 64-bit wide DRAM bus running at 3.2GHz with an average utilisation of 80%. This implies a sustained bandwidth of $\frac{3.2*10^9*8*8}{10} \simeq 19.07$GiB/s. Figure 6.3 shows that three instructions are executed per byte of data transferred from DRAM on average. Assuming this average, a SIMD processor must process $19.07*3 \simeq 57.21$ giga-instructions per second. Aiming for a 1GHz clock frequency and assuming that each processing element achieves a throughput of 1 IPC, at least 58 processing elements are required to balance DRAM bandwidth with processing power.

It would be wrong to interpret these results as rigid, as there are quite a few sources of imprecision. Firstly, for in-order processors an IPC of 1 per SIMDs-lane is unrealistic. Control-flow induced pipeline flushes, divergent branches and read-after-write induced stalls all contribute to a lower net IPC. Furthermore, the throughput of specialised components such as the RCP-units and the integer divider is lower than that of regular vector arithmetic. Secondly, the number of instructions per bytes calculated assumes that each load/store operation loads 4 bytes. However, associative caches can eliminate aliasing DRAM requests. Both sources of imprecision make the estimated optimum of 58 processing elements an underestimate. For the design space exploration that follows, I thus choose to gather data for configurations with 64, 128 and 256 SP-units.

Note that the two benchmarks requiring trigonometric operations have an above average number of instructions executed per byte of data consumed. This means that if the number of processing units is balanced with bandwidth according to the average instructions/byte, these benchmarks are expected to be compute-bound. Given software emulation of such operations requires about 110 instructions, hardware support for trigonometric operations provides a measurable benefit to these benchmarks. Given the low projected cost of implementing these operations in a ROM-backed scheme that re-uses existing floating-point multiply addition resources, I expect addition of such hardware to be a valuable investment despite their limited scope.

The instructions/byte ratios show a few other outliers, the most significant being MRI-Q's computeQ kernel. When inspecting the assembly, I conclude that this ratio is the result of its 512-entry "kValues" buffer being placed in constant memory. This is permitted in OpenCL due to the buffer's limited size. At first sight it seems that if each value were to be loaded from global memory instead, the instructions/byte ratio would approach 6. However, each of these 512 values are used by every work-item. Efficient coalescing logic would effectively only add one global load per work-item rather than 512 to this kernel, resulting in a ratio of 384.6.

The LUD diagonal and SRAD reduce benchmarks have a relatively high instruction/byte ratio because they successfully make use of local memory. Values that are re-used among multiple work-items are only loaded from DRAM to local memory once per work-group, significantly reducing the required bandwidth for this benchmark. Local memory is thus an important optimisation to achieve high throughput for benchmarks whose performance would otherwise be bound by DRAM bandwidth.

6.1.2 Sim-D

Architectural differences between NVIDIA Kepler and Sim-D have the consequence that porting benchmarks between these two architectures is more involved than a one-to-one translation of instructions. As a result, static benchmark characterisations for Sim-D differ from those presented in Section 6.1.1. Besides coalesced large DRAM transfers, five architectural differences lie at the heart of this:

1. On NVIDIA hardware, every operation is a vector operation. By contrast, Sim-D provisions scalar registers and operations to be used for global loop invariants, for work-group-wide shared values and for issuing large memory requests,

2. NVIDIA uses software emulation for (vector) integer division [89], whereas Sim-D includes a scalar Radix-16 integer divider to cover the most important use-cases,

3. On NVIDIA hardware, the sin and cos instructions require the use of a “presin” operation to pre-process the values. If both cos and sin of the same value are required, only a single presin operation is required for both. Sim-D simulates both sin and cos as a single vector instruction,

4. Similarly, on NVIDIA the same construction is used to calculate single-precision 2^x using two instructions “preex2” and “ex2”. None of our benchmarks used this instruction, hence Sim-D’s current ISA does not model an implementation for 2^x,

5. On NVIDIA hardware, many arithmetic instructions directly address constant buffers as one of their source operands without the need for an explicit load/store instruction, whereas immediate values can only be passed as a source operand to the mov instruction. To reduce GPR usage and instruction count, NVIDIA thus transforms programs to put immediates into constant buffers. Sim-D does not support constant buffers, and instead permits instructions to accept an immediate value as one of its source operands.

To demonstrate the consequences of these architectural differences, Table 6.2 shows the resource usage for all the kernels ported to Sim-D. In line with Sim-D’s inclusion of scalar registers, GPRs are split out into general purpose vector registers (VGPRs) and general purpose scalar registers (SGPRs). In addition, columns are added to show (vector) predicate registers (PRs), CSTACK- and scratchpad usage.

Benchm.	Kernel	GPRs			PRs	CSTACK	SP	Binary	
		Total	Vec	Sca		max depth	B/WG	# insn	B
CNN	convolution	20	4	16	2	0	6364	45	360
	relu	3	2	1	0	0	0	10	80
	maxpool	14	6	8	2	0	16900	46	368
KFusion	halfSampleRobustImage	14	7	7	2	1	16384	54	432
	depth2vertex	21	8	13	2	1	12288	55	440
	vertex2normal	24	16	8	2	0	26160	120	960
	Track	45	29	16	4	2	32864	232	1856
SRAD	srad	18	14	4	2	0	4016	78	624
	srad2	11	7	4	2	0	0	42	336
	reduce	18	9	9	2	1	8192	74	592
MRI-Q	ComputePhiMag	2	2	0	0	0	0	6	48
	ComputeQ	16	8	8	0	0	4096	30	240
SPMV	spmv_jds_naive	11	7	4	1	1	200	38	304
Stencil	naive_kernel	10	3	7	2	0	0	36	288
FFT	GPU_FFT_Global	19	13	6	0	0	0	60	480

Table 6.2: Program statistics of kernels ported to Sim-D

Comparing register usage between NVIDIA Kepler and Sim-D assembly, we observe that the total number of required GPRs remains broadly equal. On itself this observation does not bear much significance for two reasons. Firstly, NVIDIA's compiler doesn't aim to minimise GPR usage but rather aims to maximise instruction scheduling potential while keeping GPR usage below a threshold of 32. Secondly, Sim-D kernels have their registers allocated by hand rather than an optimising compiler. That being said, a trend can be observed where kernels that rely on linear mappings from work-items to their data element (e.g. the CNN relu and maxpool benchmarks) have a slightly reduced register requirement on Sim-D, as they no longer require the GPRs reserved for pointer arithmetic.

The use of scalar instructions has reduced the need for vector registers in all-but-one case, providing a strong motivation for the concept of mixed scalar/vector code. Only KFusion's tracking kernel has a slightly higher VGPR occupancy. This is largely because the original kernel writes results to DRAM as soon as they are computed, whereas the Sim-D implementation ensures that every work-group executes the same number of program phases by aggregating all DRAM and scratchpad writes at the end of the program.

These benchmarks demonstrate that to avoid register spilling, the minimum number of registers per work-group is 32 VGPRs, 16 SGPRs and 4 PRs. Common optimisation techniques like loop unrolling could increase both the VGPR and SGPR requirement. Section 7.5.3.1 demonstrates the cost and merit of loop unrolling in greater detail. As an optimising compiler is beyond the scope of this project, I will not further explore the consequences of minimising the register file size, and perform subsequent experiments under the assumption that each work-group has access to double this number of vector- and scalar GPRs.

Sim-D kernels binaries tend be smaller than their NVIDIA Kepler equivalents. As explained, this is not entirely an apples-to-apples comparison. However, a large reason for achieving smaller binaries with Sim-D's architecture is a reduction in boiler plate code surrounding DRAM transfers: code that calculates buffer offsets from TIDs is not required for strided DRAM requests.

Defining an instruction encoding can result in mandatory changes to the ISA, which will have an impact on binary size. A specific area of concern is the feasibility of encoding instructions with immediate operands of the type and size required by the current ISA specifications. If immediate operand support needs to be removed from some instructions, the introduction of mov instructions to load immediates to (scalar) registers instead will result in an increased binary size.

The scratchpad usage of all but one kernel is bound to 32KiB per work-group. The KFusion tracking kernel forms the only exception. This kernel uses a 32KiB scratchpad buffer to compose 8-element struct entries for storing into a global array-of-structs. Rather than issuing 8 inefficent sparse write requests to DRAM, this kernel instead sends 8 sparse write requests to the scratchpad, followed by a single dense transfer from the aggregate scratchpad results to DRAM. With the SRAM-backed scratchpads offering higher bandwidth and lower latency than DRAM, this local temporary data structure reduces the penalty of the sparse writes significantly without specialist hardware support.

Given the provisioning of 128KiB of combined L1 caches and shared memory per compute unit in NVIDIA's recent Volta architecture [126], I believe that a scratchpad between 32 and 64KiB per work-group is within reasonable bounds.

Finally, benchmarks only require a handful of CSTACK entries. In absence of unbound recursive methods in HRT systems, I do not foresee a large rise in CSTACK depth requirements. Given this small CSTACK size, the power- and area overhead for this dedicated storage is expected to be negligible.

6.1.3 Takeaway points

Through static data extraction from kernels assembled both for NVIDIA hardware and for Sim-D, I have motivated the following design choices:

- Given the limited size of kernels, a Harvard-architecture using dedicated IMem is a natural choice for low-cost program storage with guaranteed latency,
- (Hand-optimised) Sim-D kernels are rarely observed to require more than 32 vector- and 16 scalar registers per work-group. In the light of more advanced optimisation techniques such as loop unrolling and instruction scheduling, this should be considered a lower bound,
- It is expected that more than 58 SP-units are needed to balance DRAM throughput of a 64-bit DDR4-3200AA DRAM bus with throughput of compute resources,
- The maximum observed scratchpad allocation for any Sim-D kernel is just over 32KiB per work-group,
- None of the benchmarks ported to Sim-D require more than 2 CSTACK entries. A small hardware CSTACK is expected to suffice for most kernels.

6.2 Parameter sensitivity

This section quantifies how Sim-D's performance scales with three major design parameters: scratchpad bus width, number of compute resources and pipeline depth. Based on the benchmark resource usage presented in the previous section, I narrow down the parameter ranges to the values listed in Table 6.3

Domain	Parameter	Sim-D
Compute	Clock	1GHz
	Work-items/WG	1024
	SP/RCP-units	64/16, 128/32, 256/64
	Decode stages	1, 3
	Execute stages	3 - 6
Register file (per WG)	VGPRs	64
	SGPRs	32
	PRs	4
DRAM	Configuration	DDR4-1866M, DDR4-3200AA
	Bus width	64 bits (DDR)
	Throughput	128 bits/cycle
Scratchpad	Clock	DRAM clock
	Bus width	128, 256, 512, 1024 bits
		4, 8, 16, 32 words
	Throughput	128, 256, 512, 1024 bits/cycle
	Capacity	128KiB/WG

Table 6.3: Summary of configurations for parameter sensitivity experiments.

6.2.1 Scratchpad bus width

Table 6.2 lists the 11 benchmarks that use scratchpads for work-group-local storage. Invariably the performance of these benchmarks is linked to the bandwidth provided by the scratchpads. To understand this correlation, I executed each of these 11 benchmarks with four different scratchpad data bus widths, ranging from 4-32 words.

The bus width of a scratchpad is not an arbitrary choice, but trades off performance with the size of a crossbar that routes data between the scratchpad and the GPRs. Recall from Section 3.5.2.2 that routing N words from a data bus to any of M columns in a vector register without buffering or back-pressure requires an $N * M$ crossbar transferring 32-bit words. For the widest scratchpad bus evaluated, 32 words, this crossbar must be of dimensions $32 * 1024$.

To provide a justification for implementing such large crossbars, Figure 6.4 demonstrates the performance of each benchmark under different scratchpad data bus widths. Performance is normalised to that of the 4-word scratchpad bus width. DRAM and scratchpads are clocked at 933MHz. Numbers reported are for a 3-cycle decode phase and a 5-cycle execute pipeline phase, as I will motivate further in Section 6.2.2.

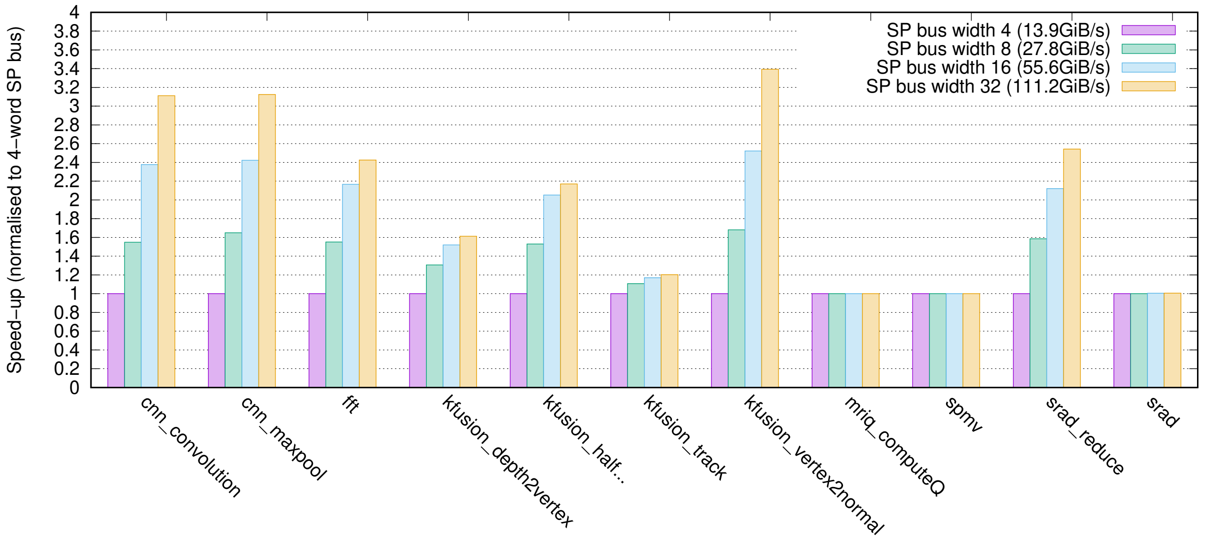


Figure 6.4: Sim-D performance for different scratchpad bus widths, clocked at 933MHz

This figure shows seven benchmarks for which performance in absolute terms increases noticeably with scratchpad bandwidth. Expressed as a function of scratchpad bus width, the observed growth of these benchmarks approaches at best a logarithmic trend, indicating a diminishing return on investment. Negative outliers are the KFusion depth2vertex and halfsample kernels, which seem to benefit little from a scratchpad bus wider than 8 words. The remaining five benchmarks still gain significant additional performance with a bus width of 16 or even 32 words.

Four benchmarks show little to no benefit from increasing the scratchpad data bus width: KFusion track, SPMV, SRAD and MRI-Q computeQ. The first three benchmarks are mainly bound by the execution of indexed transfers to/from large DRAM buffers, for which Section 5.4 demonstrates poor expected performance. The MRI-Q computeQ kernel is strongly compute bound.

6.2.2 Compute configurations

I next evaluate the influence of Sim-D's pipeline depth and the number of SP/RCP-units on benchmarks' performance. When reasoning about compute performance in isolation, the number of provisioned SP-units determines the upper bound on throughput. Net achieved compute throughput is limited by the ratio of scalar and vector instructions, the occurrence of pipeline bubbles due to RAW hazards and the frequency of pipeline flushes caused by control flow- and load/store operations. For data parallel programs, pipeline effects replicate for each work-group in a kernel-instance the same way pipeline effects inside a for-loop replicate for each iteration. Hence even small increases in pipeline-related costs can have large run-time consequences. However, in practice many benchmarks are expected to be I/O bound rather than compute bound.

To demonstrate to what degree benchmarks are sensitive to variations in the compute configuration, I gathered average case execution times for the cartesian product of all configurations listed in the compute domain in Table 6.3. To magnify the performance influence of compute resources, the DRAM and scratchpads are configured for high throughput: DRAM is modelled as a two-bank-group DDR4-3200AA configuration, and the scratchpad has a 32-word wide data bus.

Figure 6.5 shows the performance results for the tested Sim-D compute configurations. Each configuration is represented as a *(SP-units, decode stages, execute stages)* 3-tuple. Performance is normalised to that of the lowest-performing (64,3,6)-configuration. To help track correlations visually, configurations with the same number of SP-units share the same colour. Lines connect all benchmark results of one configuration.

Looking at the influence of pipeline depths, this figure shows four benchmarks whose performance varies more than 10% between the shortest and longest pipeline for configurations with 128 or 256 SP-units: CNN convolution, CNN maxpool, MRI-Q computeQ and SRAD reduce. The first three are long-running compute-bound benchmarks, the fourth is a shorter kernel that efficiently uses the scratchpad for communication between work-items in a work-group and is thus compute- or scratchpad I/O bound. For all other benchmarks, the pipeline depth has very little influence on run-time. These are either memory-bound benchmarks or benchmarks with ample vector instructions and little control flow. As a result they require only infrequent pipeline flushes and thus introduce few pipeline bubbles.

Both the MRI-Q computeQ and the CNN convolution benchmark's preference for a short pipeline are caused by large control flow overhead. These benchmarks contain a tight inner loop with a significant number of iterations: 2048 and 147 respectively. Each iteration causes a pipeline flush, and additionally the short distance between iterator increment and testing the loop invariant form a RAW-hazard introducing additional stalls on each iteration. In Section 7.5.3.1 I evaluate the effects of loop unrolling on both of these kernels, demonstrating how this technique can both reduce a benchmark's run-time and it's sensitivity to pipeline depth.

The main loop inside the SRAD reduce kernel is more difficult to unroll, as the iteration count of this loop varies between work-groups and kernel-instances. This benchmark may benefit from a simple statically analysable branch predictor, such as static hints or a "backwards taken, forwards not-taken" policy. Although these solutions are analysable, they complicate the pipeline analysis pass of the WCET-analysis algorithm presented in Section 7.4. For this reason, I leave the exploration of branch prediction techniques as future work.

Looking at the influence of the number of compute resources on performance, the benchmarks can broadly be grouped into three categories: I/O bound, compute-bound

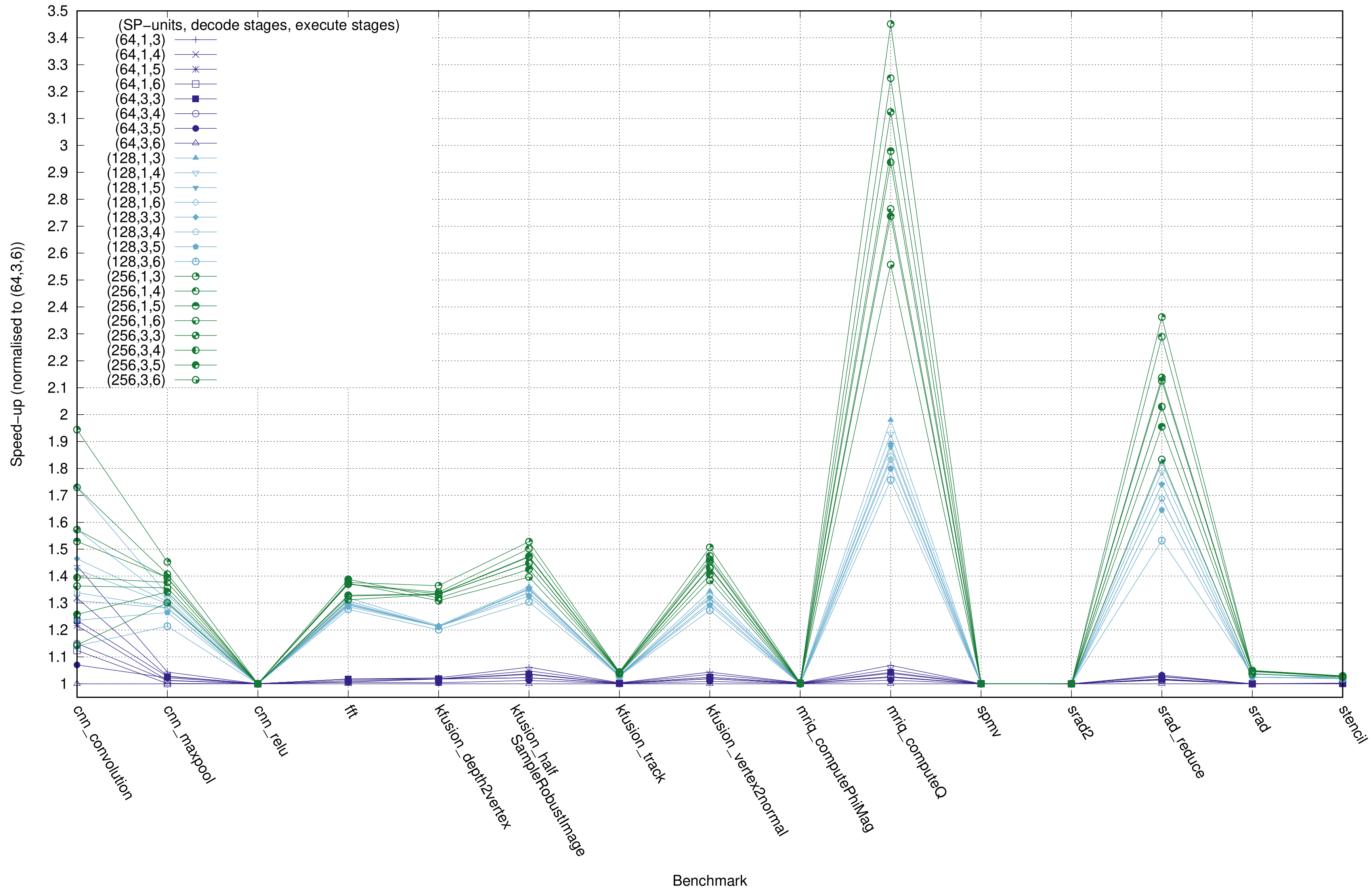


Figure 6.5: Sensitivity of benchmarks' performance to compute configurations, 2 bank-group DDR4-3200AA, 32-word SP bus.

and mixed I/O- and compute-bound.

The category of I/O benchmarks contain the CNN RELU, KFusion track, MRI-Q computePhiMag, SPMV, SRAD2, SRAD and stencil kernels. These benchmarks show very little difference in performance between different compute resource provisions.

The second category contains the benchmarks that are strongly bound by compute and control flow: CNN convolution, MRI-Q computeQ and SRAD reduce. Of these benchmarks, only the MRI-Q computeQ kernel benefits nearly linearly from extra SP-units. The other two benchmarks benefit to a lesser extent as the pipeline also stalls due to RAW hazards and integer division latencies.

The third category of benchmarks (CNN max-pool, FFT, KFusion depth2vertex, KFusion halfsample, KFusion vertex2normal) contain a mix of I/O-bound and compute-bound sections on their critical path. For this category, increasing the number of compute resources shows diminishing returns. Doubling the number of SP-units from 64 to 128 delivers a speed up of 30% or more. Again doubling the number of SP-units to 256 only gives an additional ∼15% speed-up as the number of pipeline stalls increases.

As a final note, one effect I expected to observe is that Sim-D would perform significantly better for decode and execute configurations whose combined pipeline depth does not exceed the number of warps in a work-group. The reason is that for such configurations, RAW-dependencies between two consecutive vector instructions cannot cause pipeline stalls. This effect would be most visible in the three compute-bound benchmarks under the configurations with 256 SP-units. From the graph, I conclude that this effect is negligible for the examined configurations. Although some clustering of results by pipeline depth occurs, in Section 7.5.3.1 I show that this is mostly accounted for by the cost of control flow instead.

	Speed-up vs 64 SP-units	
Benchmark	128	256
CNN Convolution	1.15	1.18
CNN Maxpool	1.24	1.31
CNN RELU	1.00	1.00
FFT	1.28	1.38
KFusion depth2vertex	1.21	1.31
KFusion halfSampleRobustImage	1.31	1.41
KFusion track	1.03	1.04
KFusion vertex2normal	1.28	1.39
MRI-Q computePhiMag	1.00	1.00
MRI-Q computeQ	1.78	2.70
SPMV	1.00	1.00
SRAD2	1.00	1.00
SRAD reduce	1.59	1.89
SRAD	1.02	1.05
Stencil	1.02	1.03
Average	1.19	1.31
Avg. mixed/compute-bound	1.35	1.57

Table 6.4: Speed-up obtained for number of SP-units, normalised to 64.

Deciding upon the best all-round configuration requires additional metrics, like area and power consumption. In absence of a physical implementation, the speed-ups listed in Table 6.4 provide a preliminary justification for a configuration with 128 SP-units. For a first-order estimate, assume that the area of a SimdCluster with 64 SP-units is 50% parallel compute resources and 50% pipeline and storage for register file and scratchpad. Choosing a configuration with 128 SP-units would then increase the total area of the SimdCluster by 1.5×, while 256 SP-units results in a SimdCluster 2.5× the area. Given the speed-up for mixed- and compute-bound applications on a 256 SP-unit configuration averages 1.57×, this configuration seems uneconomical. A configuration with 128 SP-units is expected to give a better return-on-investment, with the speed-up for mixed- and compute-bound applications averaging at 1.35×.

6.2.3 Takeaway points

Through parameter sensitivity analysis, comparing average case performance of benchmarks on various Sim-D configurations, I draw the following conclusions:

- The correlation between scratchpad bus width and performance is logarithmic at best. In absolute terms, many benchmarks benefit from a bus width of 8 or 16 words,

- The pipeline depth of the decode and execute phase generally has very little influence on performance. For the three benchmarks that are sensitive to pipeline depth, control flow-induced flushes are the main source of pipeline overhead,
- Assuming area of the compute resources scale linear with the number of SP-units, configurations with 128 FPUs are expected to balance area with performance.

6.3 Average case performance comparison

To put in perspective the performance potential of Sim-D, this section compares Sim-D under two DRAM configurations with the NVIDIA GeForce GT710 and NVIDIA GeForce GTX780 graphics cards. The most important parameters of both systems are summarised in Table 6.5.

		NVIDIA GeForce	
Parameter	Sim-D	GT710	GTX780
	Compute		
Clock	1GHz	1GHz	992MHz
Work-items/WG	1024	< 1024	< 1024
SP/RCP-units	128/32	192/32[3]	2304/384
Decode stages	3	?	?
Execute stages	5	?	?
	DRAM		
Configuration	DDR4-1866M, DDR4-3200AA	DDR3 @ 1866MHz	GDDR5 @ 6.4GHz
Bus width	64 bits (DDR)	64-bits (DDR)	384-bits (DDR)
Throughput	14.4, 23.8 GiB/s	14.4 GiB/s	288.4 GiB/s
	Scratchpads		
Clock	DRAM clock	?	
Bus width	128, 256, 512, 1024 bits	2048 bits/SMX [19]	
	4, 8, 16, 32 words	64 words/SMX	
Capacity	64KiB/WG	16KiB/SMX (+L1)	

Table 6.5: Summary of configurations for average case performance experiments.

[3] "SFUs", special function units, described by Oberman et al [86].

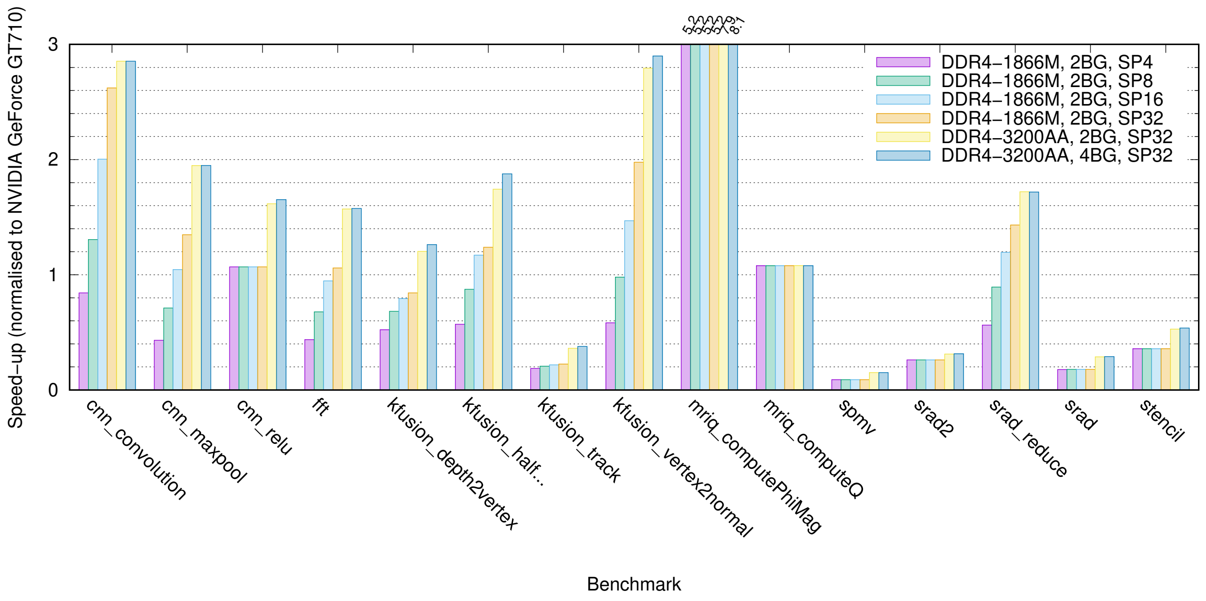


Figure 6.6: Average-case performance of Sim-D, normalised to NVIDIA GeForce GT710.

Figure 6.6 shows the measured performance of Sim-D, normalised to the performance of the low-end NVIDIA GPU. This figure shows mixed results. In some of the more trivial benchmarks, such as CNN convolution, CNN RELU, FFT and the compute-bound MRI-Q computeQ, Sim-D is able to match or even slightly outperform NVIDIA's low-end GPU provided sufficient scratchpad bandwidth. This is also true for MRI-Q computePhiMag, but with a runtime of less than 2000 cycles on our system, I have little trust in the significance of this observation. Such short kernels might exacerbate NVIDIA's fixed-cost overheads caused by the OpenCL run-time (e.g. command submission through the PCIe bus). These overheads are not included in the run-times reported for Sim-D.

In many other cases Sim-D lags behind, often achieving around 40-50% performance, with less than 10% in the worst example in the case of the SPMV benchmark. This difference is explained by the absence of transparent caches in Sim-D. Although our scratchpad can help fulfil some roles of transparent caches, there are two situations in which a scratchpad-based solution ends up performing poorly: indexed requests into large buffers and data sharing across work-groups.

SRAD and SRAD2 are benchmarks that relies on indexed transfers, loading arbitrary data points from a DRAM buffer of ∼898KiB. Unfortunately, this benchmark does not have any scope to bound the region its indexes can refer to. As Figure 5.16 shows, such a buffer size is large enough to make both indexed transfer methods perform poorly. NVIDIA benefits from the inclusion of an L2 cache to drastically speed up such DRAM accesses in the average case. Unfortunately, I am unaware of any techniques that utilise associative caches to improve the worst case.

For kernels that share read-only data among all work-groups, there is scope for better performance. Currently, to provide constants or kernel parameters, Sim-D requires each work-group to load the same static data from DRAM. As these constant values tend to be of a scalar nature, the DRAM bus utilisation of such transfers will be poor.

One solution to this problem that can be explored in future work involves marking scratchpad data as persistent and read-only, loaded upon launching a kernel. Scratchpads have significantly lower latencies than DRAM and already often serve as a cache for scalar values when pressure on scalar registers is high. As long as non-preemptive execution is enforced, the loads from DRAM to a persistent scratchpad buffer only needs to occur once per kernel-instance rather than once per work-group, taking pressure off the DRAM bus. The potential speed-up of static scratchpad buffers is limited given most such transfers only take up less than 100 cycles per work-group, but it can be achieved with little additional hardware support. For the KFusion halfSampleRobustImage benchmark, reading kernel parameters from a persistent scratchpad buffer has the potential of reducing execution time by $\sim 1.3\%$.

Comparing benchmark performance between a DDR4-1866M and a DDR4-3200AA configuration, Figure 6.6 shows that the architecture still benefits from higher DRAM throughput. The performance of memory-bound benchmarks scales nearly linearly with the theoretical bandwidth provided by the DRAM technology. Furthermore, four bank-group DRAM chips consistently outperform two bank-group configurations.

	Cycles		
Benchmark	Sim-D	GTX780	Speed-up
CNN Convolution	24462378	5901185	×4,15
CNN Maxpool	368726	78122	×4,72
CNN RELU	58985	10895	×5,41
FFT	216062	20106	×10,75
KFusion depth2vertex	350894	40212	×8,73
KFusion halfSampleRobustImage	102770	19008	×5,41
KFusion track	7015098	256715	×27,33
KFusion vertex2normal	522664	137421	×3,80
MRI-Q computePhiMag	1291	5091	×0,25
MRI-Q computeQ	86547947	7669264	×11,29
SPMV	1438668	38622	×37,25
SRAD2	1653467	34357	×48,13
SRAD reduce	278721	58928	×4,73
SRAD	1745937	47182	×37,00
Stencil	970810	46040	×21,09
Average difference			×15.33

Table 6.6: Performance comparison of Sim-D (DDR4-3200AA, 2BG, 32-word SP bus) and NVIDIA GeForce GTX780

Finally, Table 6.6 shows that the high-end NVIDIA GeForce GTX780 achieves over 15× better performance on average when compared to Sim-D. This is unsurprising considering it's DRAM bandwidth is over 10× larger and it contains 18× as many SP-units.

Scaling Sim-D to the size of a high-end GPU is still an open problem. This scalability problem poses two main challenges for future research: increasing the DRAM bus width without sacrificing data bus utilisation, and increasing the number of SimdClusters while remaining able to derive a tight WCET for kernel-instances. I expect that a solution to the latter challenge can provide a foundation for research towards temporal- and spatial multitasking methods for HRT data-parallel accelerators.

6.4 Hardware-supported optimisation

While conducting average-case performance experiments, I identified two hardware optimisations that could benefit some of the benchmarks: n-vector loads and warp trimming. This section briefly discusses the trade-offs and potential gains for both techniques.

6.4.1 n-vector load/stores

Some benchmarks store data as arrays-of-structs. For example, the input and output buffers of the FFT benchmark are list of complex numbers stored as (real, imaginary)-pairs. If a kernel requires all elements of the struct, naively loading such data from DRAM into vector registers is unnecessarily expensive on Sim-D: a program would first load all real numbers into one vector register, followed by a load of all imaginary parts. The nature of DRAM burst transfers is such that on an architecture without associative caches, two transfers of the same memory segment are issued, each discarding half the data.

If an array-of-structs data layout is a strict requirement, it appears worthwhile to add specialised instructions that can transfer multiple-element vectors between DRAM and consecutive VGPRs. To this end, I next evaluate a modified FFT kernel that utilises specialised *ldglin.vec2* and *stglin.vec2* 2-vector load/store instructions.

To put the results in perspective, I compare this kernel with an FFT kernel variant that does not require specialised n-vector load/store instructions. Instead, this variant uses the scratchpad as a cache. It first loads the DRAM tile containing all (real,imaginary)-pairs for a work-group into a scratchpad buffer, and then uses indexed transfers to load both components from the scratchpad buffer. Compared to a naive kernel that just uses indexed transfers to load both components directly from DRAM, this scratchpad-cached technique reduces contention on the DRAM bus and promotes parallel execution of the two active work-groups. Furthermore, if the scratchpad bandwidth is more than twice as high as that from DRAM, this solution will yield higher performance.

Given these two load strategies, I next argue argue against adding specialised 2-vector load/store support in hardware. To motivate this claim, I first explain how I believe the cost of implementing such specialised support is roughly equal to the cost of doubling the scratchpad data bus width. To this end, Table 6.7 summarises the hardware changes required to implement each solution.

Component	2-vector	Double scratchpad bus width
VRF banks	$\times 2$, half # rows each	(Unaltered)
VRF distribution crossbar	$\times 2$ outputs	$\times 2$ inputs
StrideSequencer DRAM	+1 mux per lane	(Unaltered)
StrideSequencers SP	(Unaltered)	$\times 2$ lanes
StorageArray SP	(Unaltered)	Additional constraints

Table 6.7: Estimated implementation cost of 2-vector load/store and doubling SP bus width.

From the listed components, I expect that the cost of the crossbar size dominates. This expectation is based on Cakir et al.'s observation [93] that a crossbar with a number of input/output lines of the same order as required for our distribution network takes up an area of 3.2mm^2 when fabricated using a 40nm process. They report modelling the area of the crossbar as follows:

$$wire\ length_{i/o} = \frac{wire\ pitch_{o/i} \times DW \times N}{number\ of\ metal\ layers} \tag{6.1}$$

$$area = wire\ length_i \times wire\ length_o \tag{6.2}$$

Under the simplifying assumption that the wire pitches for input and output are equal, doubling the scratchpad width increases the size of the cross-bar by as much as implementing 2-vector load/store.

Table 6.8 lists the binary size for five possible kernels, each representing one strategy for loading and one for storing. For loading the options are either to use 2-vector loads or to cache the tile of 2-vector elements in the scratchpad. For storing the options represent either writing to DRAM directly using two indexed transfers or first precomposing the tile of 2-vector elements in the scratchpad before writing back the data using a single DRAM transfer. For comparison, a base-line kernel is included that performs direct reads and writes. Note that storing results requires a permutation of work-items to data elements owing to the butterfly pattern of FFT write-backs. I deem the implementation of *indexed* 2-vector transfers prohibitively expensive as they require doubling the number of CAMs in the snoopy indexed transfer subsystem.

Load	Store	VGPRs	SGPRs	PRs	SP Alloc	Binary	
					B/WG	# insn	B
Direct	Direct	13	6	0	0	76	608
2-vector	Direct	13	6	0	0	59	472
2-vector	SP buffered	13	6	0	16384	57	456
SP buffered	Direct	13	6	0	16384	77	616
SP buffered	SP buffered	13	6	0	16384	75	600

Table 6.8: Program statistics for four variants of the FFT kernel

Compared to the baseline, 2-vector loads allow a modest reduction in binary size. Additionally, the bottom three variants require a scratchpad allocation of 16KiB per work-group. When compared to the resource usage of the other 14 benchmarks as listed in Table 6.2, these differences bear no significance.

Table 6.9 lists the performance of each benchmark in number of cycles. In all cases, Sim-D was configured with 128 SP-units, a 3-stage decode pipeline phase and a 5-stage execute pipeline phase. DRAM is configured as two-bank-group DDR4-3200AA.

	Load					
	Direct	2-vector			SP-buffered	
SP width (words)		8	16	32	16	32
Store direct	744948	479900	479900	479900	506283	496640
Store SP buffered		276899	*249964*	223410	247405	*216062*

Table 6.9: Cycle count for various FFT implementations, (128,3,5)-configuration.

The first row demonstrates how both the 2-vector load variant and the scratchpad cache variant gain significant performance over directly loading the two data pairs from DRAM using four snoopy indexed requests, eliminating one third of the overhead. Replacing the direct storage with a scratchpad-assisted storage eliminates another third of overhead from the baseline. Both improvements are attributed to the improved efficiency of DRAM transfers in this I/O-bound benchmark.

Comparing the 2-vector load variant with the scratchpad buffered load variant for equal scratchpad configurations shows that performance is within 7% of each other. The first row shows how 2-vector loads on themselves have a slight advantage over scratchpad buffered indexed loads. However, the variants in the second row, performing scratchpad-buffered write-back of data, shows a different trend. Analysis of this data reveals that the biggest deciding factor for performance here is no longer the chosen data load mechanism, but rather the way the program phases of two active work-groups interleave. By shifting DRAM refresh operations or applying scheduling restrictions to the simply greedy work-group scheduler, the measured average-case performance of these benchmarks vary by more than 10% either way. As I will demonstrate and explain in greater detail in Chapter 7, understanding and constraining this performance variation is the main challenge for determining the WCET of a program. On a like-for-like basis though, this data suggests that there is no reason to assume that either load strategy is universally superior when considering the behaviour of the whole kernel.

A slightly different picture emerges when comparing two solutions that under the assumptions detailed above require roughly equal area, e.g.the 2-vector load variant with a scratchpad bus of 16 words versus the SP-buffered load variant with a scratchpad bus of 32 words wide. For a kernel that uses unoptimised write-back of data, the 2-vector load variant remains a faster option. However, 2-vector stores cannot be used for write-back of data because FFT requires a permutation of data elements. This permutation breaks the linear relationship between TID and data elements, which is a prerequisite for large 1D/2D block transfers. Once the 2-vector load variant relies on the scratchpad for write-back of data, the scratchpad's throughput becomes a bottleneck, For this reason, the scratchpad buffered load variant with a bus width of 32 words is ∼14% more efficient than the 2-vector load variant with a 16-word wide scratchpad bus.

This latter comparison demonstrates how scratchpad load/stores are more universally applicable than dedicated n-vector load/stores. This is backed by the static benchmark analysis: Section 6.2.1 lists seven benchmarks that benefit from widening the scratchpad bus, whereas Table 5.1 lists only four benchmarks that may benefit from n-vector load/store operations. Of these benchmarks, two would require 3-vector loads, for which implementation is all but trivial. Whether a kernel benefits from either a wider scratchpad bus or specialised n-vector loads ultimately depends on the nature of the kernel, but data suggests that within a given area budget, a wider scratchpad bus provides higher returns than n-vector load/stores.

6.4.2 Warp trimming

Section 3.2.2 introduced the concept of warp-trimming. I explained how the TID-to-SIMD-lane mapping scheme (linear vs. compacted) influences the number of warps that could potentially be trimmed from remainder work-groups. Generally the more warps can be trimmed, the better these remainder work-groups perform. When mapping threads onto a 3D grid of data elements, as done by the CNN convolution and Stencil benchmarks, the number of remainder work-groups can be potentially large.

By analysing the remainder work-groups of each benchmark, I next argue that the performance benefit of warp trimming is expected to be negligible. To this end, Table 6.10 lists the number of dynamic instructions executed for each benchmark whose kernel-instances have at least one remainder work-group. The number of cycles is divided into three categories: vector instructions, scalar instructions and no-ops. The last category includes the cycles that the compute resources spend idling while both work-groups are occupying other resources. Using these numbers, I extrapolate the number of cycles required to execute remainder work-groups under both TID mapping schemes if warp trimming were implemented. These cycle counts are derived by scaling the number of executed vector instructions linearly with the number of active warps in each work-group. From these scaled number, a projected total number of cycles is calculated which indicates the best-case savings. For 1D kernel-instances, the results for the compressed- and linear TID mapping scheme are equal. Fractional cycle counts are the results of different work-groups taking different code paths.

					Linear				Compressed			
Cycles/WG		Work-group			Warps	Cycles/WG			Warps	Cycles/WG		
Scalar	NOP	Location	Dim	#	#	Vector	Total	%	#	Vector	Total	%
CNN Convolution												
876	5676.5	Default	$32 * 32$	2304	8	1248	7800.5					
		East	$26 * 32$	384	8	1248	7800.5	0%	7	1092	7644.5	-2%
		South	$32 * 26$	384	7	1092	7644.5	-2%	7	1092	7644.5	-2%
		South east	$26 * 26$	64	7	1092	7644.5	-2%	6	936	7488.5	-4%
		Total		3136			24392490	-0.3%			24322602	-0.6%
CNN Maxpool												
65	895.1	Default	$32 * 32$	64	8	480	1440.1					
		East	$23 * 32$	64	8	480	1440.1	0%	6	360	1320.1	-8.3%
		South	$32 * 23$	64	6	360	1320.1	-8.3%	6	360	1320.1	-8.3%
		South east	$23 * 23$	64	6	360	1320.1	-8.3%	5	300	1260.1	-12.5%
		Total		256			353368	-4.2%			341848	-7.3%
SPMV												
355	116556.75	Default	$1024 * 1$	11	8	2977.41	119889.17					
		East	$684 * 1$	1	6	2233.06	119144.82	-0.7%	1D			
		Total		12			1437925.6	-0.05%				
SRAD Reduce												
111.9	1094.4	Default	$1024 * 1$	112	8	1287.1	2493.4					
		East	$270 * 1$	1	3	482.7	1689	-32.3%	1D			
		Total		113			280947.6	-0.3%				
Stencil												
19	1875.5	Default	$64 * 16$	210	8	128	2022.5					
		East	$62 * 16$	210	8	128	2022.5	0%	8	128	2022.5	0%
		South	$64 * 14$	30	7	112	2006.5	-0.8%	7	112	2006.5	-0.8%
		South east	$62 * 14$	30	7	112	2006.5	-0.8%	7	112	2006.5	-0.8%
		Total		480			969850	-0.1%			969850	-0.1%

Table 6.10: Projected best-case speed-up from optimising work-group size for trimmed edges.

These results show that the biggest potential speed-up is for the CNN Maxpool benchmark, about 4.2% under the linear TID mapping and up to a 7.3% speed-up for a compressed mapping scheme. All other benchmarks show a potential improvement of less than 0.6%.

As argued in Section 6.2.2, many of these benchmarks are DRAM bound. For purely DRAM-bound benchmarks, but also benchmarks whose performance is bound by the scratchpad throughput, the reduction of committed vector instructions gained by warp trimming is expected to be matched by an increase in the number of idle cycles that the work-groups spend waiting for data to be read or written. For the most promising benchmark, CNN Max-pooling, Sim-D's compute resource occupation is 56.4%, while DRAM and scratchpads are occupied for 77.5% and 26.5% respectively. This is a strong indicator that the benchmark are I/O bound rather than compute-bound, diminishing the expected returns of warp trimming in practice.

Bar the projected speed up for the CNN max-pooling benchmark, I deem the benefits of warp-trimming too insignificant to justify its implementation. I thus leave further research of warp trimming techniques for future work.

6.4.3 Takeaway points

In previous chapters I have mentioned two hardware-assisted performance optimisations: vector load/stores and warp trimming. From an initial performance evaluation of these techniques, I conclude that neither are expected to have a significant positive effect on performance. Specifically, I showed that:

- Tailored n-vector load/store operations can improve performance for four out of the 15 benchmarks. At a comparable area budget, better improvements are expected from doubling the data bus width of the scratchpad,
- An initial estimate of the performance effects of warp trimming shows that four benchmarks might perform 0.6% less computations and one benchmark could reduce computation by up to 7.3%. The I/O-bound nature of these benchmarks make it unlikely that these improvements translate to a significant reduction in run-time.

6.5 Summary

This chapter presented the results of a design space exploration for the Sim-D architecture. The goal is to justify its design decisions and to present the performance of a potential hardware implementation in the perspective of both an embedded grade- and a high-end GPU.

From benchmark resource usage I have justified many of Sim-D's design decisions. Small observed kernel sizes justify a Harvard-architecture, CSTACK usage is small enough to justify a limited hardware-backed stack and register usage is generally limited to 32 VGPRs and 16 SGPRs. Scratchpad usage of ported benchmarks rarely exceeds 32KiB per work-group, in line with local memory sizes on modern NVIDIA GPUs.

Static analysis of benchmarks compiled for NVIDIA Kepler reveals that the average number of performed operations per byte of data consumed is $\sim$3. Taking this into account, I experimented with Sim-D hardware designs containing 64, 128 and 256 SP-units to balance the bandwidth from a 64-bit wide DRAM bus with the compute requirements of these application. Based on a first-order estimate of relative area for these design points, a design with 128 SP-units seems to offer the best trade-off between performance and area.

Comparing performance results for different pipeline depths, I conclude that most benchmarks are not very sensitive to the number of pipeline stages. Two of the three benchmarks that show a higher sensitivity are mainly bound by control overhead. In Section 7.5.3.1 I will show that loop unrolling can both achieve a significant reduction in execution time of these benchmarks and reduce these benchmarks' sensitivity to the pipeline depth. From this insensitivity I conclude that a 10-stage pipeline with 3 decode stages and 5 execute stages is a reasonable design point for further experiments.

Benchmarks show that Sim-D, configured with a 10-stage pipeline, is able to match or exceed the performance of the NVIDIA GeForce GT710 GPU for over half of the benchmarks. As a condition for achieving such performance, Sim-D must be equipped with a scratchpad that can sustain a throughput of at least 16 words per cycle. For wider scratchpad data buses, Sim-D's performance exceeds that of NVIDIA's GPU for some benchmarks. However, kernels that require indexed transfers into large buffers fail to come close to the performance offered by NVIDIA's GPU on account of Sim-D lacking the transparent cache hierarchy required to speed up such transfers in the average case.

Comparing Sim-D's performance against the high-end NVIDIA GeForce GTX780 graphics card reveals that there is still a large gap to potentially bridge. On average the NVIDIA GeForce GTX780's performance is 15.3× that of Sim-D. An interesting avenue for future research is to explore ways in which Sim-D's DRAM data bus and SP-unit count can scale up to match high-end GPUs without compromising on Sim-D's ability to provide tight WCET bounds on the execution of kernel-instances.

Finally, experiments have dissuaded me from pursuing implementation of two hardware performance optimisations: n-vector load/stores and warp trimming. N-vector load/stores can help performance for benchmarks that organise data as arrays-of-struct. The practical scope of this optimisation is limited to 2 out of 15 benchmarks. At the same area budget, widening the scratchpad data bus has a larger positive effect on performance. Meanwhile, out of the 5 benchmarks that may benefit from warp trimming, 4 have a maximum projected performance increase of 0.6%. A fifth benchmark could gain as much as 7.6% performance, but unless this benchmark is strictly compute bound it is unlikely to achieve such speed-ups in practice.

CHAPTER 7

Worst-case execution time analysis

The previous chapters presented and evaluated the design and analysis of the various resources in the Sim-D hard real-time SIMD architecture: a memory controller, a scratchpad and a processor pipeline. A distinct feature of Sim-D is the strict performance isolation between these resources. By virtue of this isolation, the simplest way to schedule and hence reason about the worst-case execution time (WCET) of kernels is by modelling each work-group as a sequence of critical sections, henceforth *program phases* or simply phases. Each phase requires exclusive access to precisely one resource. To maximise hardware utilisation and performance, Sim-D permits program phases of two different work-groups to execute in parallel as long as they require different resources. In this chapter, I exploit this *coarse-grain program phase scheduling* model to statically determine the WCET of a given kernel in the light of this parallelism.

The main challenge with determining the WCET of a kernel is finding the worst-case interleaving of these program phases. The size of the search space of all plausible interleavings is non-trivial, owing to variation in the run-time of each program phase. These variations are caused by three effects. Firstly, execution time of access phases can vary at run-time as a result of data alignment differences and net data size differences when processing remainder work-groups. Secondly, varying request times for the periodic DRAM refresh can lead to variations in DRAM blocking times. Finally, the run-time of execute-phases vary in the light of divergent control flow.

In this chapter I present an efficient solution to find the worst-case execution time of a Sim-D program. To this end, I introduce a formal abstraction called a *serialisation*. This abstraction is used to reduce the WCET-derivation problem from finding the worst-case schedule of program phases to finding the worst-case serialisation of phases. Architecturally I introduce two easily implementable work-group scheduling policies, each resulting in execution following a single worst-case serialisation. Together, these results allow a static WCET analysis algorithm to only consider one schedule under each of the two policies.

Specifically, I make the following contributions:

- An analysis that establishes the intuition behind the search problem of a worst-case schedule (Section 7.1),
- A formal description of a kernel, a system, a serialisation and a schedule. I prove that for each serialisation only a single worst-case schedule exists, and that run-time may safely skip program phases provided the order of program phases executed does not change otherwise (Section 7.2),
- Two work-group scheduling policies that guarantee that run-time execution always follows a single serialisation (Section 7.3),
- A full description of the WCET algorithm for either scheduling policy (Section 7.4),
- An evaluation of the performance of both scheduling policies in terms of average- and worst-case execution times. Additionally, three case-studies evaluate the average-case- and worst-case execution time benefits of common software optimisation techniques: loop unrolling, instruction scheduling and tiling (Section 7.5).

7.1 Motivation for work-group scheduling constraints

In this section I provide an intuition for the interactions between run-time work-group scheduling and static WCET analysis. Sim-D schedules each work-group as a sequence of uninterruptible *program phases*, each phase occupying exactly one resource. The work-group scheduler interleaves the execution of program phases from up to two work-groups at a time. Owing to Sim-D's strict performance isolation between resources, the WCET of each phase can be determined independently. Given a sequence of program phases and their WCETs, static analysis must find the worst-case interleaving of phases permitted by the on-line work-group scheduler.

In this section I make two claims. Firstly, the ways in which program phases could be scheduled on-line by a greedy scheduler is large. Variation in schedules are mainly caused by variance in DRAM request response times. Scheduling variations are not contained to work-groups or pairs of work-groups, but spill over to subsequent work. As a result, the search space of schedules to evaluate when determining the WCET of a kernel-instance is non-trivial. This motivates either a need for containing this variance using more constrained work-group scheduling policies or a need for formally sound reasoning about the worst-case schedule in presence of variance.

The second claim is that a more constrained scheduling policy does not necessarily translate to worse performance. Indeed, for one of the FFT variants in Section 6.4.1 I observed that a stricter work-group scheduling policy led to better average- and worst-case performance. This observation motivates my approach of analysing the performance of two simple scheduling policies that significantly reduce the search space of potential schedules.

To support these claims, I visualise partial executions of benchmarks using *occupation graphs*. An occupation graph is a time-line showing for both active work-groups which resource it occupies at each point in time. The horizontal axis depicts time, counted in discrete cycles at the rate of the compute resource. Different colours are used for the different resources, while white indicates idle time and black indicates time waiting for the DRAM resource to become available.

7.1.1 DRAM variance and schedule search space

I first demonstrate how variance in the response time of a DRAM request can result in different program phase schedules. Variance in DRAM response time has three sources: the number of (gross) bytes transferred, the alignment of data with respect to DRAM burst alignment and refresh-induced blocking.

Variance in the number of transferred bytes is commonly caused by the execution of remainder work-groups. For example, consider a kernel operating in $32 * 32$ work-groups on a $257 * 257$ dataset. A 2D block transfer for the work-group processing the top-left tile of data will transfer $32 * 32 = 1024$ words of data. However, the bottom left remainder work-group only reads $32 * 1$ words, while the bottom right remainder work-group only processes a single data word. Translated to cycles, this means the read time of a tile varies between 31 and 311 compute cycles.

This upper bound on response time must be increased further if blocking occurs as a result of a DRAM refresh. For the example above, refresh-induced blocking increases the worst-case response time to 661 compute cycles.

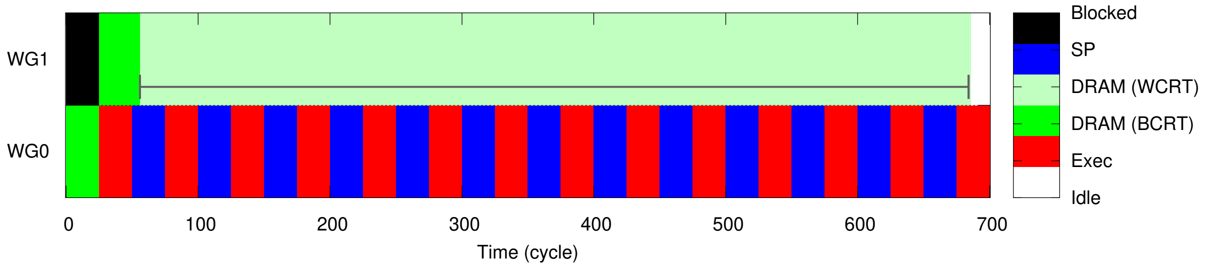


Figure 7.1: Occupation graph: scheduling difference between best- and worst-case DRAM response time

Figure 7.1 displays a fragment of a occupation graph for two pathological work-groups.

Each work-group must execute a DRAM request, after which compute and scratchpad access alternate at 25 cycle intervals. The difference between best-case (31 cycles) and worst-case (661 cycles) DRAM response time is displayed in light green. Note that in non-pathological graphs, the interval spent blocking on refresh would be displayed in black rather than light green.

This figure demonstrates how variation in a DRAM request's response time can cause one work-group to drift from the other. In the worst case, the work-group in slot 0 (henceforth simply *work-group 0*) can execute 26 program phases in full while work-group 1 waits for its data. In the best case work-group 0 only executes a single program phase before work-group 1 finishes its DRAM request. Since every DRAM request must necessarily be followed by a compute phase (or work-group exit), this difference will affect how the phases of the two work-groups interleave, hence influence the schedule later on.

To demonstrate how drift can lead to different schedules, Figure 7.2 depicts two actual executions of the CNN convolution kernel. Both diagrams in Figure 7.2 show the first 5000 cycles of execution for the CNN convolution kernel. The top diagram demonstrates an execution where refresh does not cause blocking, while in the bottom diagram refresh causes blocking on the third DRAM request for work-group 1.

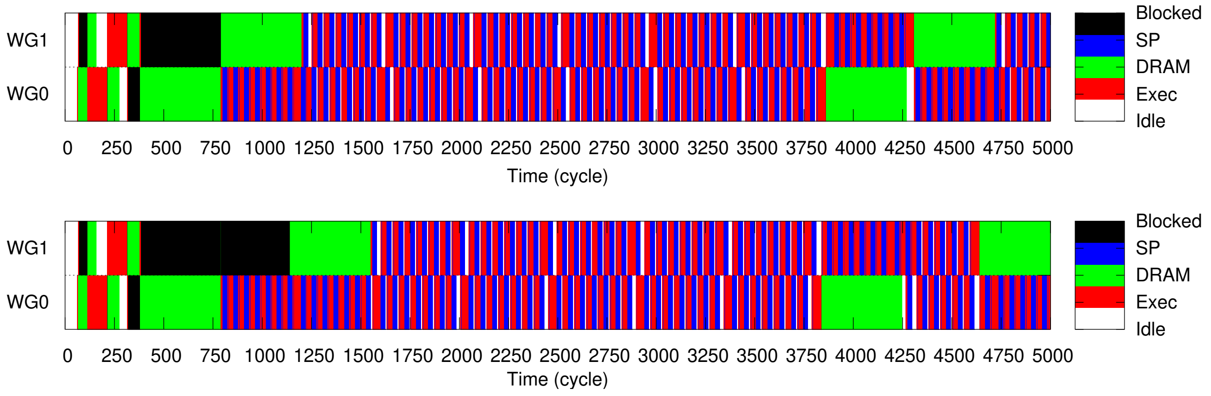


Figure 7.2: Occupation graphs for CNN convolution: no refresh-induced blocking (top), refresh-induced blocking (bottom)

As we can see, resource usage of this kernel resembles that of the pathological case presented in Figure 7.1. Where the top schedule shows thirty resource switches for work-group 0 while work-group 1 waits for its DRAM request to complete, in the bottom execution this number of resource switches increased to 54.

This work-group drift has a subtle but important effect on scheduling later on. As expected, work-group 1 executes its next DRAM request later in the bottom schedule. However, work-group 0's DRAM request starts and finishes its next DRAM request a fraction earlier. From the demonstrated interval alone it is impossible to judge which of the two executions will finish earlier.

As it turns out, the bottom schedule finishes its first two work-groups 519 cycles later than the top schedule. However, letting both executions process all 3136 work-groups, the bottom schedule finishes 120 cycles earlier than the top execution.

It is important to realise how the effects from small scheduling variations ripple through the entire schedule. The two CNN convolution kernel executions in Figure 7.3 show how scheduling decisions affect program phase execution across work-group boundaries. The top graph displays execution when launching two work-groups. The bottom shows the same kernel when launched with four work-groups and thus twice as many work-items.

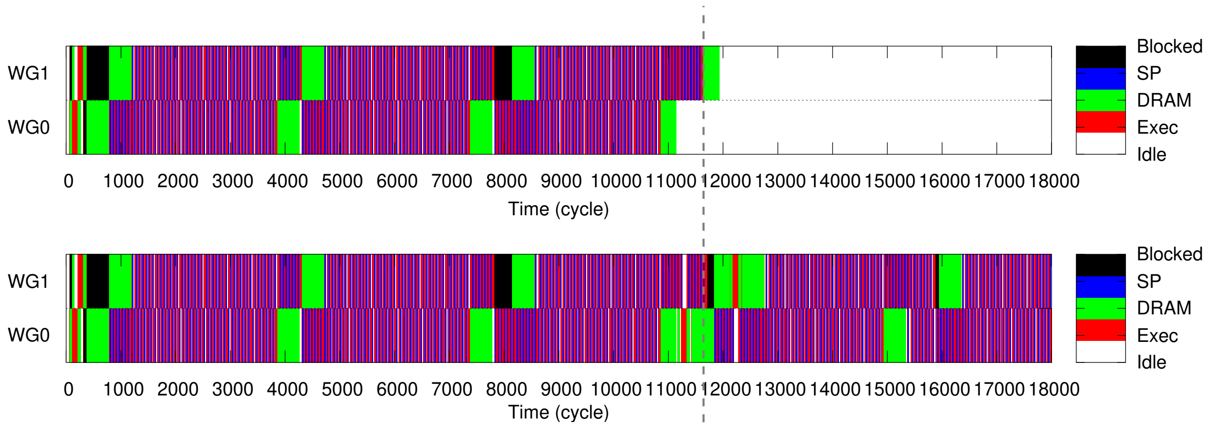


Figure 7.3: Occupation graphs for CNN convolution. 2 work-groups (top), four work-groups (bottom).

Note how in this figure, the penultimate DRAM request for the first work-group in slot 1, around cycle 8000, is blocked by a refresh operation. This increase in DRAM response time allows the work-group in slot 0 to drift from the work-group in slot 1 by several program phases. The bottom occupancy graph shows the consequence of this drift. The second work-group in slot 0 manages to execute six program phases before the final DRAM request of the first work-group in slot 1. As illustrated by the dashed line, execution of the second work-group in slot 0 causes additional blocking on the last DRAM request of the first work-group in slot 1, causing the work-group in slot 1 to finish later.

This figure shows how a simple greedy scheduling policy allows work-groups to drift with respect to each other. This drift can cause blocking that *locally* leads to longer work-group response times than when drift is disallowed. It is not obvious whether this local inefficiency is a reflection of *global* performance of this scheduling algorithm.

On a higher level, this example shows how for a given kernel-instance, run-time variation in the execution time of program phases can result in many possible schedules. As it stands, the only way to derive the WCET of an application would be to generate all possible schedules, taking into account possible variations in response times of thousands of DRAM requests and compute sections. As I will show in Section 7.2, evaluating all possible schedules to find the WCET is intractable.

There is thus a need to narrow down the set of candidate worst-case schedules. As the examples above demonstrate, intuitions like "the worst-case schedule is one for which all execution- and DRAM request times are maximised" are not trivially true. This motivates a need for architectural- and formal methods to narrow down the set of candidate worst-case schedules.

7.1.2 Scheduling policies

Having motivated the need for methods to narrow down the set of candidate worst-case schedules, I next illustrate why architectural restrictions on work-group scheduling can generate promising results. Run-time work-group scheduling policies must pursue two goals: to limit the number of possible program phase schedules, and to maximise resource occupancy. At first sight these two goals appear to conflict. However, the FFT benchmark provides evidence to the contrary; even with simple policies it is possible to both improve a kernel-instance's performance and to reduce its program phase scheduling variance.

Figure 7.4 shows two executions of the FFT benchmark. The top execution is under a simple unconstrained greedy work-group scheduling policy, while the bottom execution enforces the additional constraint that scratchpad accesses may not overlap with compute. This and other constraints are discussed in greater detail in Section 7.3.

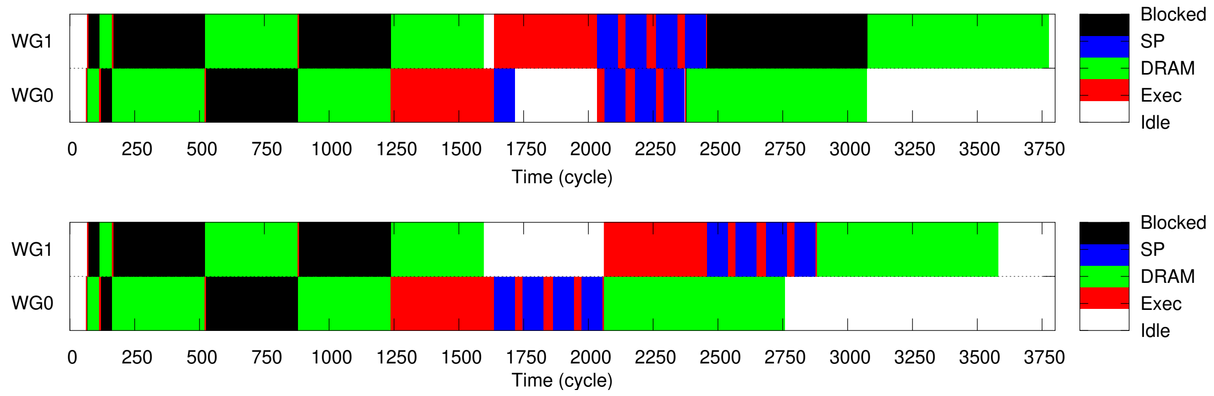


Figure 7.4: Occupation graphs for FFT: Unrestricted greedy scheduling (top), greedy scheduling, scratchpad access not parallel with compute (bottom)

These diagrams show how placing additional constraints on the work-group scheduler improves performance of both work-groups. Indeed, for this benchmark the constrained scheduler achieves 6% higher performance than the unconstrained scheduler.

To break this example down, the top occupation diagram shows a long idle period for work-group 0 between cycle 1717 and 2034, waiting for work-group 1 to finish its compute phase. Subsequently, work-group 1's final DRAM request suffers from a long blocking delay. The bottom diagram shows how under this more constrained scheduling policy, work-group 0 no longer waits for work-group 1's long compute section, allowing it to issue its DRAM request 317 cycles earlier. In turn, work-group 1's long compute section is now scheduled in the time it would have other spent blocking on work-group 0's final DRAM request. The end result is a schedule under which both work-groups finish earlier, reducing the run-time of this pair of work-groups by 195 cycles.

At the same time, this *scratchpad access as compute* policy narrows down the number of candidate worst-case schedules to one. To explain why it is sufficient to evaluate only a single schedule for deriving the WCET of a kernel-instance scheduled under this policy, the next section introduces the necessary theoretical concepts and theorems to model program execution on Sim-D. This is followed by an explanation and evaluation of relevant scheduling constraints.

7.2 Program- and execution model

The formal foundation under Sim-D's worst-case timing analysis relies on four abstractions: a *system*, a *kernel-instance*, a *serialisation* and a *schedule*. These abstractions omit definitions for modelling DRAM refresh. Section 7.4.8 explains how cost inflation is used to account for refresh overhead instead. Definitions for the four abstractions follow.

A *system* is described by a set of *resources* R, each resource R_i represented by a (τ_i, I_i)-pair containing the resource type $\tau \in$ {COMPUTE, DRAM, SP}, and the instance number I. The Sim-D architecture is represented by the set of resources $R = \{(\text{COMPUTE}, 1), (\text{DRAM}, 1), (\text{SP}, 1), (\text{SP}, 2)\}$.

A *kernel-instance* $K = (w, (\Phi_1, \ldots, \Phi_n))$ is modelled as a sequence of n program phases Φ and a number of work-groups w. Each program phase Φ_i is described as a (ρ_i, c_i)-pair describing the resource type $\rho_i \in$ {COMPUTE, DRAM, SP} and the phase's WCET (cost) c_i. Each work-group executes a kernel-instance's program phases Φ in order.

Owing to the Sim-D execution model, a work-group must alternate between compute and storage access phases. Formally, all odd program phases $(\Phi_1, \Phi_3, \ldots, \Phi_{n-1})$ must require the compute resource, and all even program phases $(\Phi_2, \Phi_4, \ldots, \Phi_n)$ perform either a DRAM or scratchpad access. This follows from the fact that all DRAM and scratchpad requests must be issued by an instruction. The first phase must therefore be compute, and the distance between two successive accesses must be at least one compute instruction. There is no need for any compute past the final store operation as without a store its results can only be discarded. Hence the last program phase must be a DRAM store, and thus a program always has an even number of program phases n.

To reason about the problem of scheduling program phases on resources, two further abstractions are introduced: *serialisations* and *schedules*.

A *serialisation* of a kernel-instance describes an ordering of the program phases of every work-group in a kernel-instance. Formally, a serialisation $S(K) = (\nu_1, ..., \nu_m)$ is a sequence of program phase instances $\nu_j = (\phi_j, w_j, r_j, z_j)$, $\phi_j \in \Phi$ the corresponding program phase from K, w_j the work-group number, $r_j \in R$ the resource occupied by this program phase instance and $z_j \in \{0, 1\}$ the work-group slot this phase instance occupies. The order of program phases in a serialisation determines the order in which phases are launched at run-time. For the Sim-D system, a *valid* serialisation is one for which the order of phases is preserved for each work-group, and for each subsequence containing all program phase instances of one work-group slot, no two work-groups interleave their phases.

As a second abstraction, a *schedule* can be informally thought of as a (run-time) instance of a serialisation with associated time information. Formally, a *schedule* $s(K) = (\sigma_1, ..\sigma_n)$ is a sequence of resource reservations, each entry representing a 5-tuple $\sigma_i = (\phi_i, w_i, r_i, z_i, [t_i^{start}, t_i^{end}])$ with $\phi_i \in \Phi$ and w_i the program phase and work-group for this reservation, $r_i \in R$ the resource occupied by this reservation, z_i its work-group slot and $[t_i^{start}, t_i^{end}]$ denoting the interval during which this reservation is active. In this work, all time is measured in discrete clock cycles at the rate of the compute resource. A schedule is said to run for the interval $[1, t^{end}]$, with t^{end} defined as follows:

Definition 3. *For a given schedule, t^{end} is the last time-instant that the corresponding kernel-instance runs.*

$$t^{end} = max(\{t_m^{end} \mid \sigma_m \in s(K)\})$$

There is a one-to-many relationship between serialisations and schedules: From a schedule, its corresponding serialisation can be obtained by ordering the elements by start time, and extracting the relevant program phase instances from it. Schedules are derived from a serialisation by augmenting each program phase instance with start- and end times. To honour the ordering of a serialisation, for every schedule derived from a serialisation $S(K)$ each reservation σ_j corresponding with program phase instance ν_j must start before or at the same time as the reservation corresponding with ν_{j+1}.

The number of resource reservations in a schedule is denoted with $|s(K)|$. The set of all active reservations in a schedule s at time $\tau \in [1 : t^{end}]$ is given by the function $A(s, \tau)$.

A summary of the symbols used in this work is given in Table 7.1.

Symbol	Description
R_i	Resource
τ_i	Type of resource R_i
I_i	Resource instance of resource R_i
$K = (w, (\Phi_1, \ldots, \Phi_n))$	Kernel-instance
w	# Work-groups launched for kernel-instance K
Φ_j	Program phase j for kernel-instance K
ρ_j	Resource type required by program phase Φ_j
c_j	Worst-case execution time (cost) of program phase Φ_j
$S(K)$	Serialisation of kernel-instance K
$\nu_m = (\phi_j, w_j, r_j, z_j)$	Program phase instance of serialisation $S(K)$
ϕ_m	Program phase for m'th entry in $S(K)$
w_m	Work-group of m'th entry in $S(K)$
r_m	Resource occupied by m'th entry in $S(K)$, $r_m \in R$
z_m	Work-group slot for the m'th entry in $S(K)$
$s(K)$	Schedule of kernel-instance K, sequence of reservations
$\|s(K)\|$	Number of resource reservations in schedule $s(K)$
$\sigma_i = (\phi_i, w_i, r_i, z_i, [t_i^{start}, t_i^{end}])$	Reservation i of schedule $s(K)$
ϕ_i	Program phase for i'th entry in $s(K)$, $\phi_i \in \Phi$
w_i	Work-group of i'th entry in $s(K)$
r_i	Resource occupied by i'th entry in $s(K)$, $r_i \in R$
z_i	Work-group slot for the i'th entry in $s(K)$
t_i^{start}	Start time of i'th entry in schedule $s(K)$
t_i^{end}	End time of i'th entry in schedule $s(K)$
t^{end}	End time of the last-finishing entry in schedule $s(K)$
$A(s, \tau)$	Set of reservations in schedule s active at $\tau \in [1, t^{end}]$

Table 7.1: Common symbols and definitions.

7.2.1 Worst-case minimal valid schedule

The purpose of this section is to formally justify why for each serialisation there is only one schedule relevant for WCET analysis. This schedule is called the *worst-case minimal valid schedule*. To this end, I first give a formal definition of *valid* and *minimal* in the context of a schedule derived from a specific valid serialisation, after which I prove that the worst-case valid minimal schedule is the one where the execution time of each program phase is maximised.

A *valid schedule* is one for which the following condition holds:

Definition 4. *A **valid schedule** $s(K)$ for a kernel-instance K is a schedule for which a valid serialisation $S(K)$ exists, and additionally no resource is allocated to more than one interval at any point in time:*

$$\forall \tau \in [1 : t^{end}], \forall \sigma_x, \sigma_y \in A(s(K), \tau), x \neq y : r_x \neq r_y \tag{7.1}$$

A *minimal valid schedule* derived from a serialisation is a valid schedule that leaves no resource idle when constraints permit scheduling work on them. In other words, a minimal valid schedule schedules each program phase as early as possible, provided the requested resource is available meeting all constraints, the serialisation's order is honoured and for each work-group its program phases do not overlap. In Section 7.3 I use this property of minimality to define *greedy* scheduling algorithms, suitable for on-line evaluation, that behave deterministically under varying phase execution times. Note that this property of minimality can be considered weaker than *work-conserving*, as it allows for the definition of deterministic scheduling constraints that leave some, but not all, resources idle even when a work-group is otherwise ready to use them. Before presenting a formal definition of a minimal valid schedule, I first provide three auxiliary functions:

Definition 5. *$RRes(s, \rho)$ is a function that for a given schedule s returns the end time of the last resource reservation $\sigma_i \in s$ for which $r_i = \rho$, or 0 if no such reservation exists.*

$$RRes(s, \rho) = max(\{t_i^{end} \mid \forall \sigma_i \in s : r_i = \rho\} \cup \{0\}) \tag{7.2}$$

Definition 6. *$WEnd(s, \zeta)$ is a function that for a given schedule s returns the end time of the last resource reservation $\sigma_i \in s$ for which $z_i = \zeta$, or 0 if no such reservation exists.*

$$WEnd(s, \zeta) = max(\{t_i^{end} \mid \forall \sigma_i \in s : z_i = \zeta\} \cup \{0\}) \tag{7.3}$$

Definition 7. *$Pre(s, m)$ is a function that returns the sub-schedule (prefix) of schedule s containing all elements up to but excluding element m:*

$$Pre(s, m) = (\sigma_i \in s \;:\; i < m) \tag{7.4}$$

Using these three functions, the minimal valid schedule is defined as follows:

Definition 8. *A **minimal valid schedule** $s(K)$ with respect to a serialisation $S(K)$ is* *a valid schedule for which* $\forall \sigma_m \in s(K)$ *the following holds:*

$$t_m^{start} = \begin{cases} 1 & \textit{if } m = 1 \\ max \begin{pmatrix} t_{m-1}^{start}, \\ RRes(Pre(s(K), m), r_m) + 1, \\ WEnd(Pre(s(K), m), z_m) + 1 \end{pmatrix} & \textit{otherwise} \end{cases} \tag{7.5}$$

$$t_m^{start} < t_m^{end} \le t_m^{start} + c_m \tag{7.6}$$

Phases in a minimal valid schedule are required to finish within c_i cycles from start, but no other bounds are placed on the execution time of a phase. Hence a serialisation can result in many different minimal valid schedules.

The next theorem introduces the worst-case minimal valid schedule derived from a serialisation $S(K)$, being the one that maximises the execution time for every element.

Theorem 1. *The* ***worst-case*** *minimal valid schedule* $s(K)$ *for a given serialisation* $S(K)$ *is one where each phase executes for its worst case execution time:*

$$\forall \sigma_i \in s(K) : t_i^{end} = t_i^{start} + c_i \tag{7.7}$$

Proof. By induction. Let $s(K)$ and $s'(K)$ be minimal valid schedules derived from $S(K)$, $s(K)$ being a *worst-case minimal valid schedule* and $s'(K)$ being a minimal valid schedule where at least one program phase executes for a shorter amount of time. I prove $t'^{end} \le t^{end}$ by proving two invariants: $\forall i \in [1 : |s(K)|] : t_i'^{start} \le t_i^{start}$ and $\forall i \in [1 : |s(K)|] : t_i'^{end} \le t_i^{end}$.

Let d be the index of the first resource reservation in both $s(K)$ and $s'(K)$ such that $t_d^{end} \ne t_d'^{end}$ and $Pre(s(K), d) = Pre(s'(K), d)$. If $d = 1$, the first element in $s(K)$ and $s'(K)$ already differ in execution time. Trivially, $\forall i \in [1 : d) : t_d'^{start} \le t_d^{start}$ and $t_d'^{end} \le t_d^{end}$ by definition of d.

Consider d the base case for the induction argument. Consider two cases. If $d = 1$, $t_d'^{start} = t_d^{start} = 1$ per Equation 7.5. For $d > 1$, Equation 7.5 evaluated to the maximum of three components. The first component is equal for both scheduler as per the start-time invariant. The second and third component are equal as, by definition, $Pre(s(K), d) = Pre(s'(K), d)$. Because all three components are equal for both schedules I conclude that $t_d^{start} = t_d'^{start}$. The first invariant holds.

Since $t_d^{start} = t_d'^{start}$, the definitions given by Equations 7.6 and 7.7 guarantee that $t_d'^{end} \le t_d^{end}$. Hence the second invariant holds.

For the induction case $d+1$, the start-time invariant guarantees that $t_d'^{start} \le t_d^{start}$. For the second and third component of the maximum in Equation 7.5, the end-time invariant guarantees that:

$$RRes(Pre(s'(K),d+1), r_{d+1}) \leq RRes(Pre(s(K),d+1), r_{d+1}) \text{ , and}$$
$$WEnd(Pre(s'(K),d+1), z_{d+1}) \leq WEnd(Pre(s(K),d+1), z_{d+1})$$

Since all three components of the maximum in Equation 7.5 must be smaller or equal for $s'(K)$, the maximum must be smaller or equal too.

For the second invariant, recall from Equation 7.7 that $t^{end}_{d+1} = t^{start}_{d+1} + c_{d+1}$. Substituting this into Equation 7.6 gives $t'^{end}_{d+1} \leq t'^{start}_{d+1} + c_{d+1} \leq t^{start}_{d+1} + c_{d+1} = t^{end}_{d+1}$, proving the second invariant holds too.

This concludes the induction argument demonstrating that both invariants hold for all elements in the schedules. From Definition 3 it now follows that:

$$t'^{end} = max(\{t'^{end}_i : \forall \sigma_i \in s'(K)\}) \leq max(\{t^{end}_i : \forall \sigma_i \in s(K)\}) = t^{end}_i \qquad \square$$

This theorem reduces the problem of finding the worst-case schedule to one of finding the worst-case serialisation, as each serialisation only has a single worst-case minimal valid schedule to consider. I use this result in Section 7.3 to justify designing work-group scheduling policies that strictly limit the number of serialisations that could occur at run-time, thus bounding the WCET analysis problem.

7.2.2 Work-group early exit

There are kernels that may require whole work-groups to exit early, thus skipping their remaining program phases. The next theorem provides a limited scope for doing so, following an induction-based proof very similar to that of Theorem 1.

Theorem 2. *Removing a program phase from a serialisation does not increase the run-time of its worst-case minimal valid schedule.*

Proof. By induction. Let $S(K)$ be an arbitrary serialisation of a kernel-instance K, and $S'(K) = S(K) \setminus \{\nu_x\}$ the serialisation with one element ν_x removed. Let $s(K)$ and $s'(K)$ be the worst-case minimal valid schedules derived from $S(K)$ and $S'(K)$ respectively. Assume that both $S(K)$ and $S'(K)$ are indexed contiguously. The following function maps an index for an element in $S'(K)$ to its corresponding index in $S(K)$:

$$map(i) = \begin{cases} i & \text{iff } i < x \\ i+1 & \text{otherwise} \end{cases}$$

Let d be the index of the first element after x, such that ν'_{d-1} and $\nu'_d \in S'(K)$ map to the elements directly before and after the removed program phase ν_x respectively.

I prove $t'^{end} \leq t^{end}$. To this end, I prove the start-time invariant $\forall i \in [1 : |S(K)|] : t'^{start}_i \leq t^{start}_{map(i)}$. The related end-time invariant $\forall i \in [1 : |S(K)|] : t'^{end}_i \leq t^{end}_{map(i)}$ follows trivially as, per Equation 7.7, $t^{end}_i = t^{start}_i + c_i$. Therefore, if the start-time invariant holds, then $t'^{end}_{d+1} = t'^{start}_{d+1} + c'_{d+1} \leq t^{start}_{map(d+1)} + c_{map(d+1)} = t^{end}_{map(d+1)}$. For $i \in [1 : d-1]$, the start-time invariant is trivially true as $Pre(s(K), i) = Pre(s'(K), i)$ by definition.

Consider d the base case for the induction argument. Equation 7.5 justifies that $t'^{start}_{d-1} \leq t^{start}_{d-1} \leq t^{start}_x$. Furthermore, by definition $Pre(s(K), x) = Pre(s'(K), d)$ and $Pre(s(K), x) \subset Pre(s(K), d)$. As the RRes and WEnd functions return a maximum value from the sequence of end-times, I may conclude:

$$\begin{aligned}
RRes(Pre(s'(K), d), r'_d) &= RRes(Pre(s(K), x), r_{map(d)}) \\
&\leq RRes(Pre(s(K), map(d)), r_{map(d)}) \\
WEnd(Pre(s'(K), d), z'_d) &= WEnd(Pre(s(K), x), z_{map(d)}) \\
&\leq WEnd(Pre(s(K), map(d)), z_{map(d)})
\end{aligned}$$

As all components of the maximum in Equation 7.5 are smaller or equal for $s'(K)$, the maximum must be smaller or equal too. Therefore the invariant $t'^{start}_i \leq t^{start}_{map(i)}$ holds.

For the induction case $d+1$, the start-time invariant guarantees that $t'^{start}_d \leq t^{start}_{map(d)}$. For the second and third component of the maximum in Equation 7.5, the end-time invariant guarantees that:

$$\begin{aligned}
RRes(Pre(s'(K), d+1), r'_{d+1}) &\leq RRes(Pre(s(K), map(d+1)) \setminus \sigma_x, r_{map(d+1)}) \quad \text{and} \\
WEnd(Pre(s'(K), d+1), z'_{d+1}) &\leq WEnd(Pre(s(K), map(d+1)) \setminus \sigma_x, z_{map(d+1)})
\end{aligned}$$

As adding σ_x to the prefix schedule cannot decrease the outcome of the maxima, it is also safe to assume that:

$$\begin{aligned}
RRes(Pre(s'(K), d+1), r'_{d+1}) &\leq RRes(Pre(s(K), map(d+1)), r_{map(d+1)}) \quad \text{and} \\
WEnd(Pre(s'(K), d+1), z'_{d+1}) &\leq WEnd(Pre(s(K), map(d+1)), z_{map(d+1)})
\end{aligned}$$

Since all three components of the maximum in Equation 7.5 must be smaller or equal for $s'(K)$, the maximum must be smaller or equal too. This proves that the start-time invariant holds for element $d+1$.

This concludes the induction argument demonstrating that both invariants hold for all elements in the schedules. From Definition 3 it now follows that:

$$t'^{end} = max(\{t'^{end}_i : \forall \sigma_i \in s'(K)\}) \leq max(\{t^{end}_i : \forall \sigma_i \in s(K)\}) = t^{end} \qquad \square$$

Using this theorem, Sim-D can permit work-groups to exit early without affecting the

WCET of the kernel-instance, under the condition that this does not alter the serialisation of all other program phases. The *pairwise work-group* scheduling constraint explained in Section 7.3.1 guarantees just that.

This theorem provides additional scope to skip arbitrary program phases at run-time, for example to skip DRAM requests in a converging algorithm. However, care must be taken when constructing such programs as, in Sim-D assembly, program phase boundaries are implicit upon launching a DRAM or scratchpad request. Branching over a DRAM request causes Sim-D to execute the compute phases directly before and after this request as a single uninterruptible program phase. If this combined phase has a cost that exceeds the cost of the compute phase directly before the (skipped) DRAM request, the run-time scheduler produces an execution that is not allowed by the specification of the serialisation that produced the worst-case minimal valid schedule. To mitigate this, a mechanism must be found to separate the two program phases explicitly. Altering Sim-D to permit such arbitrary program phase skipping is considered beyond the scope of this work.

7.2.3 Complexity and bounds

Theorem 1 justifies why for each serialisation there is only one schedule to consider for finding the worst-case execution time. This reduces the search problem of a worst-case schedule to one of a worst-case serialisation. Theorem 2 allows us to only consider the serialisations where all program phases are present.

Despite the absence of useful constraints like deadlines or periodicity as provided by existing hard real-time task models, the work-group schedulers' freedom is quite limited: work-groups are indistinguishable before execution, the order of program phases within each work-group is fixed, and Sim-D disallows more than two work-groups to be active at any point in time. Essentially the only freedom the two on-line work-group schedulers have are two binary decisions: whether or not to fetch a work-group, and whether or not to enqueue the next program phase for execution.

Given the limited freedom of the scheduler, a serialisation is in essence a permutation of a multiset containing two elements "work-group slot 0" and "work-group slot 1", each with a multiplicity equal to the number of work-groups assigned to each slot multiplied by the number of program phases in the kernel-instance. Under the simplifying assumption that work-groups are evenly distributed over the two slots (an assumption that is not necessarily true), the number of possible permutations p for a kernel-instance K is determined by:

$$p = \frac{(w * |\Phi|)!}{\left(\lceil \frac{w}{2} \rceil * |\Phi|\right)! * \left(\lfloor \frac{w}{2} \rfloor * |\Phi|\right)!} \tag{7.8}$$

If WCET analysis were to consider all possible serialisations, it's time complexity would be factorial. Given both w and $|\Phi|$ can run into hundreds, solving this problem exhaustively is clearly intractable.

Most of these serialisations are undesirable. For example, a serialisation with first all work-groups assigned for slot 0, followed by all work-groups for slot 1 would exploit next to no parallelism. A good work-group scheduling policy is one that limits the number of candidate worst-case serialisations while simultaneously maximising parallel resource occupation. Such a policy does not have to be complex: even the least constrained greedy work-group scheduler would not generate schedules for the vast majority of serialisations. Moreover, many schedules would only be generated if at least one of the program phases does not execute for its maximum time.

To quantify the performance of a work-group scheduling policy, I next present both an upper- and lower bound on WCET of a kernel-instance, assuming the cost (WCET) of each of its program phase is tight. Firstly, the upper bound is determined as follows:

Definition 9. *For any work-group scheduler that schedules only following minimal valid schedules, the WCET of a given kernel-instance K is upper bounded by:*

$$WCET^{UB}(K) = w * \sum_{i=1}^{|\Phi|} c_i \tag{7.9}$$

The intuition behind this definition is that the WCET can never be worse than serial execution of the program, making no use of any resources in parallel. Minimal valid schedules forbid leaving any resource idle when there is an active work-group that can use it and no constraints preventing this use. No scheduling constraint may result in a schedule in which all resources remain idle for one or more cycles. Therefore this property rules out schedules whose execution time exceeds $WCET^{UB}(K)$.

The following definition of a per-resource sum of cost aids in deriving a lower bound.

Definition 10. *The sum of costs of a kernel-instance K for a given resource $r \in \{COMPUTE, DRAM, SP\}$ is given by:*

$$C(K, r) = \sum_{i=1}^{|\Phi|} \begin{cases} c_i & if\, r = \rho_i \\ 0 & otherwise \end{cases} \tag{7.10}$$

A lower bound on the WCET is then obtained by taking the maximum of the sum of costs on each resource. Formally:

Definition 11. *For any work-group scheduler, the WCET of a given kernel-instance K running on Sim-D is lower bounded by:*

$$WCET^{LB1}(K) = w * max(C(K, COMPUTE), C(K, DRAM), C(K, SP)) \tag{7.11}$$

Note that this bound is not tight as it does not take into account any constraints on schedules that result from data dependencies between phases. Serialisation is expected to occur as compute phases cannot start until their data is loaded or stored in full. As a result, for a kernel-instance K no valid schedule might exist with an execution time not exceeding $WCET^{LB1}(K)$.

A second non-tight lower bound can be provided by the following equation:

Definition 12. *For any work-group scheduler, the WCET of a given kernel-instance K running on Sim-D is lower bounded by:*

$$WCET^{LB2}(K) = \left\lceil \frac{w}{2} \right\rceil * \sum_{i=1}^{|\Phi|} c_i \tag{7.12}$$

This second lower bound takes into account dependencies between program phases and the fact that at any point in time two work-groups can be active in parallel, but does not consider resource double-booking. Hence this bound is also not tight. In the remainder of this work, the lower bound is reported as the maximum of these two:

Definition 13. *For any work-group scheduler, the WCET of a given kernel-instance K running on Sim-D is lower bounded by:*

$$WCET^{LB}(K) = max(WCET^{LB1}(K), WCET^{LB2}(K)) \tag{7.13}$$

7.2.4 Takeaway points

A formal definition is provided for a *system*, a *kernel-instance*, a *schedule* and a *serialisation*. I showed how deriving the WCET of a kernel-instance by evaluating all possible serialisations is of factorial time complexity, which deems this approach intractable. Definitions are given for an upper- and lower bound on the WCET of a kernel-instance. Additionally, for serialisations the following two properties are proven:

- The worst-case minimal schedule of a serialisation is one where the worst-case execution time of each of its elements is maximised (Theorem 1),
- Removing an element from a serialisation does not result in a longer WCET (Theorem 2).

7.3 Hard real-time work-group scheduling policies

In this section I explain two greedy work-group scheduling policies. These two policies have in common that they each schedule a kernel-instance following a single worst-case serialisation. This reduces the problem of WCET derivation to finding the WCET of a single worst-case minimal valid schedule.

These policies are a combination of constraints. The first constraint, pairwise work-group launching, prevents work-group drift to influence scheduling beyond the boundaries of a pair of work-groups. This constraint applies to both policies. This constraint is combined with either a constraint preventing scratchpad access from overlapping with compute, or one preventing scratchpad access from overlapping with DRAM access, to form two scheduling policies. In the remainder of this section I justify these three constraints and explain at a high level how they can be easily enforced by hardware.

7.3.1 Launching work-groups in pairs

The first scheduling constraint to consider is to always schedule work-groups in pairs. When a work-group slot frees up, this slot is kept empty until the other slot enters its final phase.

The benefits of this constraint are twofold. Firstly, by forcing reconvergence of two work-group slots at the end of a pair of work-groups, the only valid serialisations are sequences consisting of $\left\lfloor \frac{w}{2} \right\rfloor$ serialisations of two work-groups, plus a fixed serialisation of one work-group at the end in the case w is odd. The number of permutations is therefore reduced to:

$$p = \left(\frac{(2 * |\Phi|)!}{|\Phi|! * |\Phi|!} \right)^{\left\lfloor \frac{w}{2} \right\rfloor} \tag{7.14}$$

Although the reduction in search space is not asymptotic, it is significant as w can grow to multiples of 100.

Secondly, this constraint enables early exit of work-groups. If a work-group in one of the slots exits early, this policy will not launch the next work-group until the work-group in the other slot starts its final program phase. As such, under the pair-wise work-groups policy, early exit does not alter the order of any of the remaining program phases in the serialisation, a condition required for Theorem 2 to apply.

To demonstrate the need for this constraint, reconsider the resource utilisation diagrams from Figure 7.3, as taken from execution on Sim-D with a (128,1,3)-configuration and a 128B scratchpad data bus.

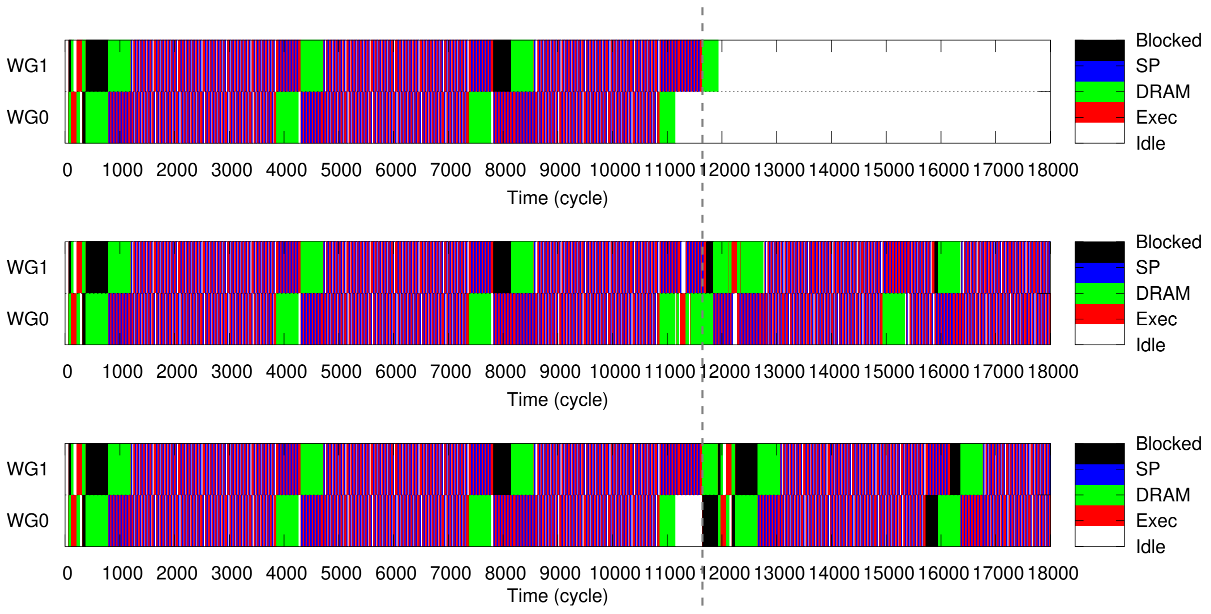


Figure 7.5: Occupation graphs for CNN convolution: 2 work-groups (top), 4 work-groups (middle), 4 work-groups, pairwise (bottom)

I explained earlier how drifting work-group slots cause scheduling variations beyond work-group boundaries. As demonstrated by the dashed line, allowing immediate release of a new work-group in slot 0 once its previous work-group has finished causes a delay on the final DRAM request for work-group 1. In the unconstrained execution (2), at the dashed line the two work-group slots have drifted apart by several program phases with no sign of reconvergence in the investigated window.

In the bottom occupation diagram we see how enforcing a *pairwise work-group* policy delays the start of the second work-group for slot 0. As a result, the final write request of the first work-group in slot 1 is now no longer blocked by a work-group that is launched later, restoring the execution schedule of the first two work-groups to that in the top occupation diagram. As the white-coloured gap for work-group slot 0 between cycle 11000 and 12000 demonstrates, this resynchronisation comes at the expense of additional idle time between the final DRAM write request of work-group 0 and that of work-group 1.

Section 7.5.1 presents benchmark measurements that assess the impact of this scheduling constraint on the average case performance.

In hardware, this policy is implemented by maintaining a boolean for each work-group slot. A slot is only permitted to fetch a new work-group when this boolean is set. Initially this boolean is set for both slots. When a slot fetches a work-group for execution, it clears its boolean. When a slot either exits early or begins the execution of its final DRAM write request, it sets the boolean of the other slot.

As a run-time optimisation, a one bit *exit* flag is added to DRAM write instructions, which indicates that no more program phases follow this write and the work-group finishes. This allows the other work-group slot to fetch and start executing the compute phase of the next work-group in parallel with this DRAM write-back, deterministically reducing idle time. Furthermore, a compute program phase consisting of a single *exit* instruction is eliminated, shortening each work-group by the number of cycles required for pipeline warm up plus executing a vector instruction.

7.3.2 Two-resource scheduling

To further reduce the number of possible serialisations, I propose two scheduling policies: *scratchpad as compute* and *scratchpad as access*. Both of these build on the observation that when a system is modelled as exactly two resources, a greedy scheduler will only schedule program phases according to a single serialisation. This serialisation is one where the two active work-groups swap resources every time they both finish their current program phase. Because programs always contain phases alternating between compute and access, as explained in Section 7.2, the resulting serialisation is one where work-groups bounce back and forth between the two modelled resources. Both policies apply the *pairwise work-group* constraint to permit early work-group exit.

These policies limit the scope for parallel occupation of resources. In the case of modelling the scratchpads as access resources, they cannot be actively transferring data for either work-group to the RF while a DRAM transfer is in operation and vice versa. Modelling a scratchpad as part of the compute resource will prevent a scratchpad request from running in parallel with another work-group's compute phase.

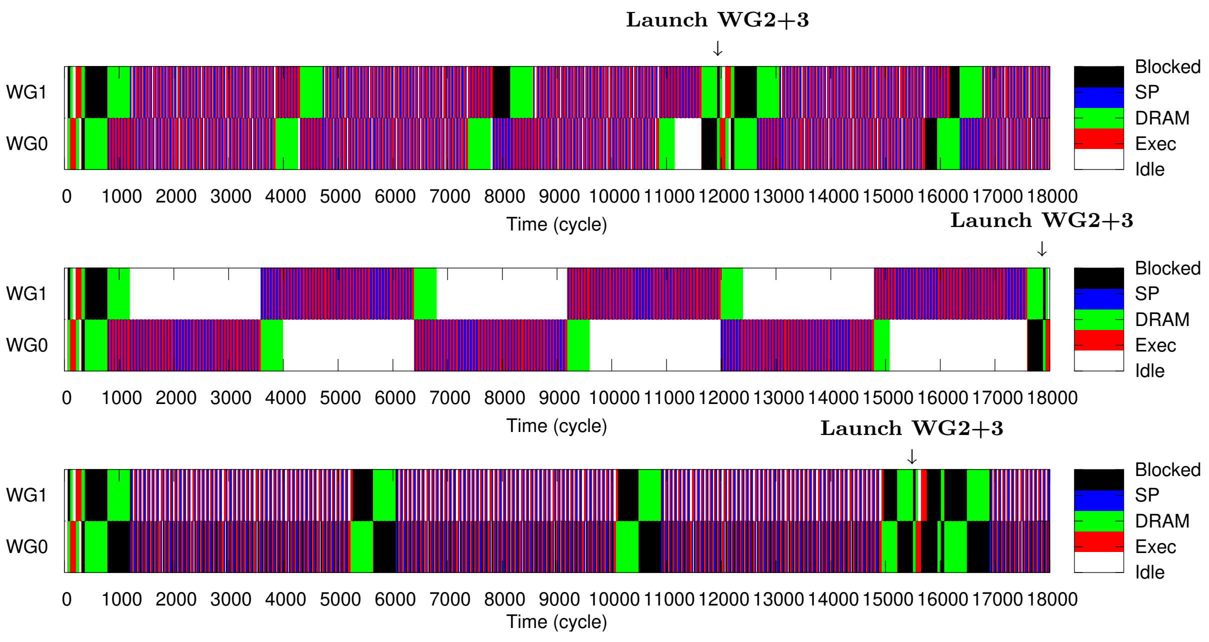


Figure 7.6: Occupation graphs for CNN convolution: pairwise work-groups (top), *scratchpad as execute* (middle), *scratchpad as access* (bottom).

The resource occupation diagrams in Figure 7.6 show that both scheduling policies impact average case performance. The *scratchpad as execute* policy causes large intervals of idle time when a work-group waits for the other to complete its sequence of compute- and scratchpad accesses. These program phases are now packed tighter together, as fine-grain blocking incurred from the other slot is eliminated. The run-time of a pair of work-groups increases under this policy by approximately 30%.

The *scratchpad as access* policy results in a different execution pattern. The big blocks of idle time observed with the *scratchpad as compute* policy have made place for a much more fine-grain interleaving of compute- and scratchpad phases. This has the effect of slightly slowing down one work-group, at the benefit of the pair of work-groups as a whole.

Which of the two policies is more efficient depends on the application. For this CNN convolution kernel, the *scratchpad as access* policy results in better performance as the use of the scratchpad has made this kernel compute-bound. DRAM-bound kernels benefit from modelling the scratchpads as compute resources to minimise the *access* critical path length. The evaluation in Section 7.5 confirms that both policies have kernels for which they outperform the other.

Note that these strong scheduling restrictions do not always yield worse performance than unconstrained greedy work-group scheduling, as shown in Section 7.1.2.

Implementation of both policies in hardware is fairly straightforward. *Scratchpad as compute* is achieved by letting the instruction fetch unit block when a scratchpad request is issued, preventing the handover of the resource to the other work-group. *Scratchpad as access* can be implemented by having all DRAM and scratchpad request be issued from a single request FIFO, rather than one per resource. In the Sim-D simulation model, the latter is achieved without merging FIFOs by implementing a ticket-locking mechanism. A ticket number is assigned to each DRAM- and scratchpad request, and a central ticket counter determines when the next request is issued to its destination resource.

7.3.3 Takeaway points

This section introduced two greedy policies for scheduling work-groups' program phases on resources: *scratchpad as compute* and *scratchpad as access*. Both policies require work-groups to be scheduled pair-wise. These policies guarantee execution following a single worst-case serialisation. Using the theorems introduced in Section 7.2, this guarantee allows performing WCET analysis of a kernel-instance by evaluating a single worst-case minimal valid schedule.

7.4 WCET computation algorithm

This section presents an algorithm that determines the WCET of a kernel-instance running on Sim-D. This algorithm combines path-based control flow analysis (CFA) with a processor-behaviour analysis (PBA) that is mostly simulation-based. The main contribution of Sim-D's WCET algorithm is its bound calculation. After identifying the critical path through the kernel's code, this path is used to characterise each work-group's execution as a sequence of *program phases*. A safe WCET bound is derived from this sequence by computing the cost of the *worst-case minimal valid schedule*. As explained in Section 7.3, this schedule is composed trivially for both the *scratchpad as access* and *scratchpad as compute* scheduling policies.

Sim-D's WCET analysis tool performs the following 8 steps, explained in detail in the remainder of this section:

1. Parse the kernel source,
2. Perform CFA to construct a CFG,
3. Perform PBA: Compute/simulate bounds for DRAM and scratchpad requests, for the execution time of each BB and for the cost of BB$\rightarrow$BB transitions,
4. Transform the CFG into a weighted directed acyclic graph (DAG), unrolling loops using iteration bounds provided by program annotations,
5. Find the critical path through the DAG,
6. Transform the critical path into a list of access and compute phases of the program,
7. Compute the WCET of the program phase list in accordance with Sim-D's worst-case work-group scheduling policies,
8. Inflate the WCET with the worst-case cost of DRAM refresh.

7.4.1 Parsing the program

The first step is to parse the program into an intermediate representation. Currently, Sim-D's program parser is not designed as a comprehensive optimising compiler. As such, generating syntax trees and SSA-form is skipped. Instead, assembly is translated 1-to-1 to instructions, which are grouped into BBs.

BBs terminate with an instruction that performs control flow, issues a DRAM or scratchpad request or that could possibly result in an injected CSTACK pop. Including DRAM and scratchpad requests into the set of BB terminators aids both with accounting for the cost of pipeline warm-up after each request, as well as with the construction of the DAG in step four and the program phase listing in step six.

7.4.2 Control flow analysis

Step two transforms the list of BBs into a CFG. CFA for Sim-D differs from conventional CFA in two ways. Firstly, Sim-D's lack of indirect branch instructions removes the need for complex or pessimistic heuristics to resolve the potential targets of such branches. Secondly, analysis must be extended to include branch targets reached by instructions that implicitly unroll the control stack. Recall that *cpop* operations are injected into the pipeline whenever the implicit predicate mask changes to all-0. Thus any instruction that writes to one of the control masks can potentially unroll the stack.

To add a safe set of outgoing edges to the BBs ending with such an instruction, the CFA-pass annotates each edge with the state of the control stack at the end of executing its source's BB. Each entry of the annotated stack state is a *(branch target, entry type)*-pair. Currently, the CFA pass enforces the constraint that all incoming edges for a BB must have an equal stack state. By enforcing this constraint, the potential branch targets of an injected *cpop* are known a-priori, allowing their edges to be generated. Control flow that violates this constraint can currently not be analysed.

At present I assume that CMASK writes may lead to unrolling the entire CSTACK. Upon encountering such an instruction at the end of a BB, an edge will be created for each entry in the stack state found on the BB's incoming edges. If the BB contains CSTACK push operations, these are included as potential branch targets. To account for the overhead of control stack unrolling, each generated edge is annotated with the number of CSTACK entries that must be popped to reach this branch.

I suspect that in practice CSTACK-unrolling stops when a stack entry is found whose type matches the type of the written CMASK. However, I found that the assumption that the CSTACK may unroll in full neither hinders WCET analysis nor causes pessimism in the determined execution bound, Therefore I leave verification of this suspicion as a future optimisation.

This CSTACK analysis correctly derives paths resulting from divergent branches. In Sim-D assembly a divergent branch is achieved with two instructions. The first pushes the reconvergence address onto the CSTACK. The second instruction updates the CMASK to execute the if-block with the correct work-items enabled, and pushes the PC and CMASK of the else-block onto the CSTACK. At the end of the if-block, the else-block entry is popped off the CSTACK and restored. At the end of the else-block, the reconvergence PC and CMASK are popped and restored. Since the state of the stack is known at each point in the program, the resulting CFG edges will represent the code path through the if- and else-block.

The CSTACK constraint on incoming edges of a BB limits the legal control flow constructions that a developer may apply. Specifically it forbids code sharing through function calls, recursive calling and loops where the stack grows on each iteration. The latter two are common limitations in software for HRT systems as such loops and recursive calls prohibit analysis of the worst-case control flow [56]. Code sharing, although desirable in the interest of code maintainability, is not functionally required. Since a program binary size is generally not a constraint, developers can easily in-line subroutines as is frequently done by NVIDIAs kernel compiler.

I believe that with sufficient engineering effort the CSTACK constraint can be relaxed. For example: a function does not need each entry point to carry the same stack state, as functions do not require to break out of the loop of the caller. As long as the top entry of the stack is of the return type, the number of return paths is bound. Furthermore, a function's WCET can be re-analysed for each invocation even if the underlying code is not replicated. I leave devising a more general solution for CFA of *cpop*-injecting instructions as future work.

7.4.3 Worst-case performance simulation

Step three annotates the CFG with worst-case execution times of both compute and DRAM/scratchpad accesses.

Determining the worst-case execution time of a DRAM request can be done using the equations and simulation techniques outlined in Section 5.4. The resulting worst-case request times are stored as metadata on the instruction that issues the request. To match Sim-D's work-group scheduling behaviour, the LID is used as a DRAM request's WCET. The WCET of a scratchpad request is determined by counting one cycle for every line read/written in the worst-case alignment, plus one cycle for its front-end overhead.

For some load/store instructions, the values determining a request's stride parameters are not computed from the kernel-instance's NDRange and buffer mapping, but rather provided explicitly by the program in SSP registers. When these SSP register values are computed at run-time, the Sim-D prototype requires the developer to annotate these register writes with their upper bounds. Implementing a constant expression analysis to derive these values automatically is left for future work.

The WCET of a BB's instruction execution is determined by simulating the timing of the Sim-D pipeline twice: once with a cold pipeline and once with a warm pipeline. The warm-pipeline case is simulated by simulating the program in linear order, as if all branches are not taken. The absence of a branch predictor in the Sim-D design ensures that any other BB entry is with a cold pipeline.

The compute time simulation accounts for all pipeline behaviour, specifically:

- The expansion of vector instructions into many sub-vector instructions. The exact number of sub-vector instructions depends on the compute resources required to perform the requested operation and their provisioning. The RCP-units are assumed to be provisioned at $\frac{1}{4}^{th}$ the number of SP-units,

- Pipeline stalls due to RAW hazards and register file bank conflicts. Using the 3-stage decode pipeline design, VRF bank conflicts should not occur in practice,
- Issue delays caused by the non-pipelined implementation of the scalar integer divider/modulo unit, both to enforce instruction commit ordering and to keep a minimal distance between two issued integer divide/modulo operations.

For both the cold- and the warm pipeline simulation, the cost of each BB is computed by counting the number of cycles between it's first instructions' write-back and the write-back of the following BB's first instruction. The cost of a cold-run is the base cost of a BB's execution. The difference between the cost for a hot and a cold run determines the penalties incurred by pipeline effects between a BB and the next. These penalties are accounted for as a weight on the BB's outgoing "fall-through" edge. Branch-taken edges have their weight set to the cost of a pipeline flush. The edges for injected *cpops* have their weight set to the cost of an injected *cpop*, multiplied by the number of CSTACK entries between the top of the CSTACK and the entry represented by the edge. For the latter two types of edges, if the destination BB starts with a scalar integer division or modulo instruction, the pipeline penalty is adjusted to take into account the time required for such an operation to reach its write-back stage.

It is worth noting that pipeline effects can propagate through multiple BBs. To justify why the two simulation paths produce safe WCET bounds, consider the example CFG in Figure 7.7.

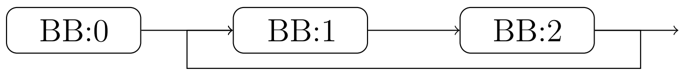


Figure 7.7: Example control flow graph fraction

Following the simulation scheme, the cost for transitioning from BB:1 to BB:2 is given by the warm-pipeline overhead as simulated along the straight-line path BB:0→BB:1→BB:2. However, entry of BB:2 could also occur through the alternative [...]BB:2→BB:1→BB:2 path. Theoretically the pipeline state on the transition from BB:1 to BB:2 could differ between these two paths. However, remember that pipeline effects that cross BB boundaries are the result of temporarily reserved resources like registers or functional units. The alternative path must enter BB:1 with no resource reservations. From this it follows that cost at run-time for executing BB:1 plus the transition to BB:2 must be smaller than the bounds derived from the straight-line warm-pipeline path. From this observation I conclude that the two simulations (warm-pipeline branch never taken, and cold-pipeline) are sufficient to produce safe bounds on execution time.

7.4.4 Construct a DAG

Given a weighted CFG, finding its worst case execution amounts to analysing its critical path. Critical path analysis requires a directed acyclic graph (DAG) in order to finish in bound time. In the fourth step of this algorithm, the CFG is thus transformed to a weighted DAG. This is achieved by eliminating cycles from the CFG through replication of nodes.

The resulting DAG has the cost for a DRAM or scratchpad request stored as weights associated with each vertex, or 0 if the corresponding BB does not perform such a request. All compute costs are represented as the weights on the edges of the DAG. Weights for these edges are computed by adding the CFG vertex's (cold-pipeline) compute cost to the weight of each of the CFG vertex's outgoing edges.

Cycles in the CFG are the result of for- and while-loops in the program. To eliminate these cycles from the CFG, each loop is transformed into a DAG containing one node for each *execution* of a BB, similar to loop unrolling. Loops are processed in depth-first order according to their nesting depth. After transforming a loop, its resulting DAG is cached in a map indexed by their entry BB, potentially overwriting an inner loop's cache entry. If during the transformation of a loop or the main program body, an edge is added to the DAG that points to a BB present in the loop-DAG cache, the cache entry's DAG is simply appended to the parent's DAG.

The number of iterations of a loop must be bound with a *branchcycle* annotation. This annotation describes the outcome of a branch instruction as a cyclical event using three parameters: #branches taken, #branches not taken and start of cycle. To give an example, the annotation "branchcycle 7 1 0" describes a branch which is taken seven out of eight encounters, the first seven encounters being taken.

These branchcycle annotations may also be used for forward branches. For example, the MRI-Q computeQ benchmark contains a main loop that is iterated over 2048 times. Every 256 iterations, starting with the first, the kernel loads a tile of data from DRAM into the scratchpad. The conditional branch jumping over this DRAM request in the other 255 iterations is annotated with "branchcycle 255 1 255".

Caching and in-lining partial DAGs for loops imposes the following limitations:

- Each loop has a single entry point and exit target,
- Loops must be properly nested,
- Inner loop branch cycle annotations may not depend on outer scopes.

Multiple jump instructions with the same loop (re-)entry target are analysed as if they are multiple nested loops. Early breaking out of a loop is correctly analysed as long as each exit branch is annotated with its branch cycle.

7.4.5 Critical path analysis

Given a DAG, its critical path is found using a dynamic programming algorithm that determines the longest path for successively larger sub-graphs. By adding vertices from the DAG to this sub-graph in topological order, no vertex needs to be processed more than once. For a DAG $D = (v, e)$ with no unconnected vertices, the resulting algorithm is of $O(|e|)$ time complexity.

During execution of the critical path algorithm, three pieces of information are stored along each vertex: a *visited* counter, the direct *predecessor* of a candidate critical path from source to this node and the cumulative *cost* of this candidate critical path.

When a node from the DAG is *processed*, it visits each of the vertices reachable through its outgoing edges. Upon visiting a destination vertex, two actions are performed. Firstly, it tests whether the cumulative cost of the current vertex plus the weight of the edge is larger than the cumulative cost of the current candidate critical path of the destination vertex. If so, the destination vertex's predecessor and critical path cost are updated to reflect the just-found longer path.

Secondly, the visited counter of the destination vertex is incremented. This visited counter is used to iterate over the vertices of the DAG in topological order. Once the visited counter of a vertex becomes equal to its indegree, the vertex is placed on a work-queue. Initially this work-queue only contains the start node of the DAG. Topological iteration of vertices is achieved by repeatedly dequeuing and processing the top element of the work-queue until the last element is dequeued.

After all vertices have been processed, the critical path is extracted from the DAG by following the trail of predecessor back-edges from the sink- to the source vertex.

This algorithm requires that all DRAM and SP requests are present on the critical path. This requirement is critical for safe WCET analysis, as violation could cause the run-time to schedule program phases following different serialisations from the one assumed by the WCET computation in Section 7.4.7 with potential worse run-times. As an implication, conditional DRAM or scratchpad requests are only permitted in two cases: either when the condition can be described using a branchcycle annotation, such as with the MRI-Q computeQ example given in the previous subsection, or when a whole work-groups exits early. In all other cases, the use of conditional DRAM or scratchpad requests is forbidden.

7.4.6 Access/execute program phase lists

Sim-D's execution model guarantees that once a work-group is assigned a compute or execute resource, it will retain exclusive access to this resource until it either requires a different resource to continue execution or the *work-group* finishes all its work. Following the model introduced in Section 7.2, a work-group's execution can thus be modelled as a sequence of program phases, alternating between compute and access.

In this step, this list of program phases $\Phi = \{(\rho_1, c_1)..(\rho_i, c_i)\}$ is extracted from the critical path generated in the previous step. To this end, the critical path is traversed from source to sink, aggregating the weight of each edge in an accumulator until a vertex with an associated DRAM or scratchpad transfer is encountered. Depending on the scheduling strategy chosen (*scratchpad as access* or *scratchpad as compute*) this access cost must now be accounted for.

For both strategies, if the vertex has an associated DRAM request, two program phases are added to the list: a compute phase whose cost is the compute cost gathered in the accumulator, followed by an access phase with its cost equal to the DRAM request WCET. Subsequently, the accumulator is reset to 0 and the algorithm continues path traversal.

If the encountered node is a scratchpad access, depending on the scheduling strategy chosen there are two ways to proceed. For the *scratchpad as access* policy, two program phases are created similarly to the DRAM request case. For the *scratchpad as compute* scheduling policy, the scratchpad access cost is added to the compute-time accumulator and traversal of the critical path continues without creating two new entries in the list.

7.4.7 WCET computation

From a program phase list, the WCET is extracted by constructing the worst-case serialisation. Under the constraints of both scheduling policies, this is the serialisation under which two work-groups alternate between resources.

Since program phases alternate between access and execute, the cost for executing all n phases for a pair of work-groups is determined by $c_{2wg} = \sum_{i=1}^{n} max(c_i, c_{(i+1)\%n})$. If a work-group contains an odd number of work-groups, one work-group will execute serially without interleaving. The cost of such execution c_{1wg} is simply the summation of the cost of all compute and execute phases.

The WCET is found by multiplying c_{2wg} with the number of work-group pairs in a program. If an odd number of work-groups was launched, c_{1wg} is added to the total. For an even number of work-groups, the calculated total must be compensated for the tails of the schedule by adding the $min(c_1, c_n)$ to the total. Finally, the program upload time must be added, which is calculated from the size of the program using the equations in Section 5.4.1. Formally, for a program binary spanning b bursts of data in DRAM the WCET (cost) c of a program is given by:

$$c_{edge} = \begin{cases} min(c_1, c_n) & \text{iff } w \text{ is even} \\ c_{1wg} & \text{otherwise} \end{cases} \tag{7.15}$$

$$c = \left\lfloor \frac{w}{2} \right\rfloor * c_{2wg} + c_{edge} + tID_R(b) \tag{7.16}$$

The program phase list constructed for the *scratchpad as access* scheduling policy is additionally used to calculate $WCET^{UB}(K)$ (Equation 7.9) and $WCET^{LB}(K)$ (Equation 7.13).

7.4.8 DRAM refresh inflation

Finally, following an approach proposed by Park et al [54], DRAM refresh is accounted for by inflating the derived WCET or bound. Assuming the ratio between the DRAM clock and Sim-D's compute clock is rCK, inflation is performed using the following equation:

$$c^{inflated} = c + \left\lceil \frac{c * rCK}{tREFI - tRFC} \right\rceil * \frac{tRFC}{rCK} \tag{7.17}$$

Inflation of WCET equates to a case where refresh occurs in a "stop the world" fashion as soon as required, halting both compute and DRAM. This is a pessimistic model of accounting for refresh cost as it differs from the actual working of the Sim-D pipeline in two ways:

1. Compute/scratchpads continue to run during a refresh operation,
2. Refresh is deferred until after the current DRAM stride request.

Point 2 is covered safely by this inflation method despite assuming a refresh penalty of $tRFC$, thus without introducing a precharge-activate cycle required to *preemptively* execute the refresh. Preemptive refresh is only required during DRAM requests that take longer than $8 * (tREFI - tRFC)$ cycles. In practice, such latencies can only occur for snoopy indexed requests into very large buffers. In these cases, Section 5.5.3 shows that the indexed iterative method should be used instead as it would give a better worst-case LID. In all other cases refresh can safely be deferred until a request finishes, at which point all banks are precharged.

Point 1 implies that inflation introduces a pessimism, specifically for compute- or scratchpad I/O bound kernels. The total cost of refresh inflation assuming DDR4-3200AA DRAM is ~4.5%, which is also an upper bound on this pessimism. In this work I accept this pessimism and leave research towards a more optimistic approach (e.g. based on blocking-time analysis) for future work.

7.5 Evaluation

In this section I discuss the results of running Sim-D's WCET tool on the full set of benchmarks. I present evidence to answer the following questions:

- How effective are the architecture design choices in facilitating tightly bound execution times?
- How does worst-case performance of the proposed Sim-D architecture compare to real-world devices?
- How well do the proposed scheduling restrictions perform in practice?

The configuration used for evaluation is a Sim-D accelerator with 128 SP-units, a 3-stage decode and 5-stage execute pipeline. The DRAM configuration is DDR4-3200AA with two bank-groups, and the scratchpad's clock matches the 1.6GHz of the DDR4 data bus. Unless specified otherwise, scratchpads are configured with a data bus width of 32 words or 128B.

As an indication of real-world tractability of the WCET analysis method, data acquisition for all benchmarks combined requires ∼15 minutes on my experimental set-up.

7.5.1 Average-case performance

To understand the overhead introduced by the *scratchpad as access* and *scratchpad as compute* work-group scheduling policies, and the fraction of this overhead attributable to the *pairwise work-groups* scheduling constraint, Figure 7.8 demonstrates the run time of benchmarks under all three constraints. Execution times are normalised to the unconstrained scheduling case.

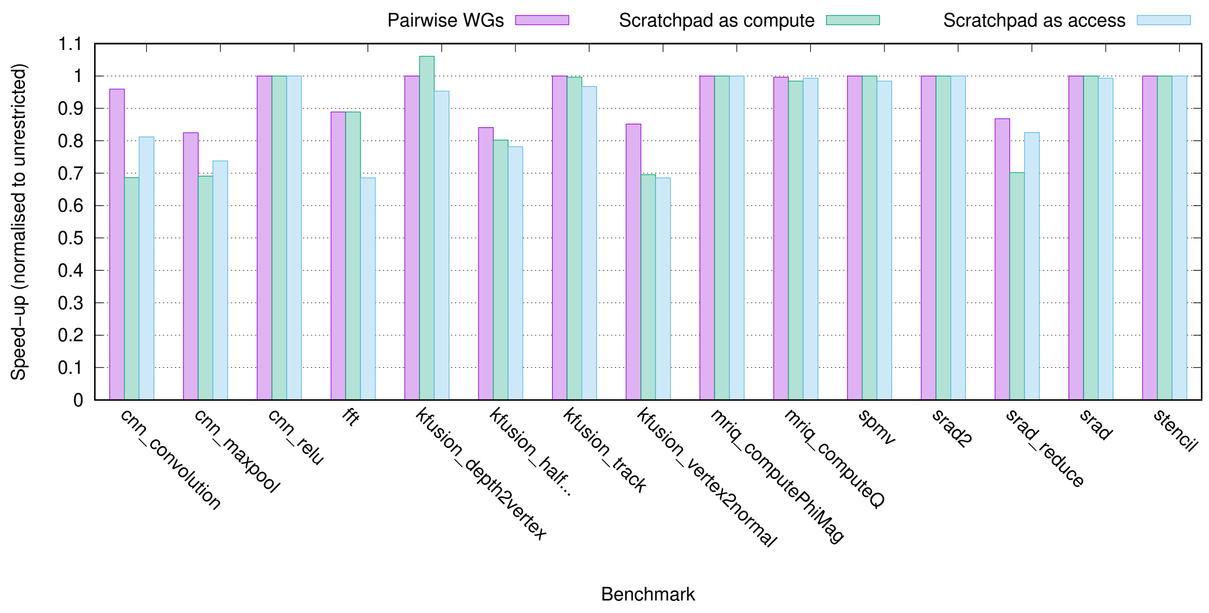


Figure 7.8: Scheduling policy impact on avg. case performance.

In line with expectations, this figure shows that kernels without scratchpad buffers (i.e. CNN RELU, SRAD2, MRI-Q computePhiMag and Stencil) are unaffected by the proposed constraints. These benchmarks use only two resources and never exit early, hence even the unconstrained scheduler will always schedule them following a single serialisation.

Performance of the other benchmarks drops by 7.1% on average when forcing pairwise work-group scheduling. This penalty reflects the cost of resynchronisation of work-groups that drift apart as a result of executing alternating compute and scratchpad accesses in parallel with a single DRAM request, as demonstrated in Section 7.1.1. This penalty varies between 0.3% and 17.5%.

The penalty for treating scratchpad requests of these kernels as either access or compute is 14.3% and 13.5% on average respectively. This cost reflects the reduced scope for parallel occupation of resources. The loss of performance is observed to be up to 31.5% under either policy.

Like the FFT variant discussed in Section 7.1.2 (not shown here), the KFusion depth2vertex kernel benefits from imposing a stricter work-group scheduling policy. Its performance under the *scratchpad as compute* policy improved by 6.1% when compared to the measurement under unconstrained scheduling.

Neither the *scratchpad as access* or the *scratchpad as compute* scheduling policy is a universally superior choice. For the KFusion and SPMV benchmarks the *scratchpad as compute* policy delivers better performance, whereas the CNN convolution, CNN maxpool and SRAD reduce benchmarks perform better under the *scratchpad as access* policy. These latter benchmarks are at least partially compute bound, hence being able to perform scratchpad transfers in parallel with compute maximises their resource occupancy.

Judging by the ease with which both scheduling policies are implemented in the simulation model, I expect that both can be present in the same hardware implementation. Driven by static analysis results or benchmark runs, developers can make an informed decision on the most beneficial scheduling policy on a per-kernel basis. This appears valuable: when comparing for each benchmark their performance under unconstrained scheduling to that of the best-performing constrained scheduler, the measured performance degradation is 10.8% on average, or 7.9% when including the unaffected kernels.

7.5.2 Worst-case execution time

Next I evaluate the WCET as determined by the algorithm presented in Section 7.4 along two metrics: tightness of the produced bounds for each scheduling policy with respect to simulated execution, and performance of both scheduling policies under worst case conditions with respect to the lower- and upper bounds on WCET as established in Section 7.2.3.

Table 7.2 displays simulated average-case times alongside compute worst-case execution times and bounds. The data for columns labelled “avg” is obtained using the cycle-accurate simulation model, while the columns labelled “wcet” contain the worst-case execution times of a kernel as determined by the algorithm described in Section 7.4.

Benchmark	Unconstr.	$WCET^{LB}$	$WCET^{UB}$	SP as access			SP as compute		
	avg			avg	wcet	% diff	avg	wcet	% diff
cnn_convolution	24462378	22852093	42466835	30118455	32403513	7.6	35643582	38237985	7.3
cnn_maxpool	368726	373469	746877	499764	543363	8.7	533841	565405	5.9
cnn_relu	58985	60966	67076	58985	60984	3.4	58985	60984	3.4
fft	216062	192880	379584	315167	327966	4.1	242970	282090	16.1
kfusion_depth2vertex	350894	307462	503712	368126	417001	13.2	330741	377601	14.1
kfusion_halfSample[...]	102770	119902	222852	131463	176890	34.5	128090	155800	21.6
kfusion_track	7015098	91111265	92146915	7249308	91744883	1165.6	7042173	91488533	1202.8
kfusion_vertex2normal	522664	555830	1111230	762549	862701	13.1	751659	826301	9.9
mriq_computePhiMag	1291	1669	1777	1291	1687	30.6	1291	1687	30.6
mriq_computeQ	86547947	87581238	93628654	87146263	92326446	5.9	87930877	93161976	5.9
spmv	1438668	1458574	1546186	1461661	1492070	2.1	1438766	1459116	1.4
srad2	1653467	11814286	11890837	1653467	11814605	618.8	1653467	11814605	618.8
srad_reduce	286514	281544	552850	347266	443790	27.8	408329	501904	22.9
srad	1745937	23199814	23413990	1737462	23252862	1238.3	1725240	23238710	1247.0
stencil	970810	1160046	1286606	970810	1171324	20.7	970810	1171324	20.7

Table 7.2: Run-time vs. WCET of kernels under various scheduling constraints, in compute cycles.

When analysing the tightness of the WCET bounds, three negative outliers stand out: KFusion track, SRAD and SRAD2. For these three benchmarks, the discrepancy between the measured execution time and calculated WCET is explained by their reliance on indexed read requests from large buffers. Section 5.4 explained that such transfers have highly pessimistic bounds on LID that are unlikely to reflect average case performance.

Unfortunately I see little room for improvement for these cases unless more details about the indexes into these buffers is known.

Ignoring these outliers, the WCET derivation algorithm produces a bound which is on average tight within 14.2% and 13.3% under the *scratchpad as access* and *scratchpad as compute* scheduling policies respectively.

Figure 7.9 visualises the data from Table 7.2 for a subset of the benchmarks. The range depicted in red represents the (non-tight) lower- and upper bounds on WCET. Not shown in these visualisations are the data points for the short-running CNN RELU and MRI-Q ComputePhiMag benchmarks, nor does the graph include the outliers that rely on indexed DRAM transfers from large buffers.

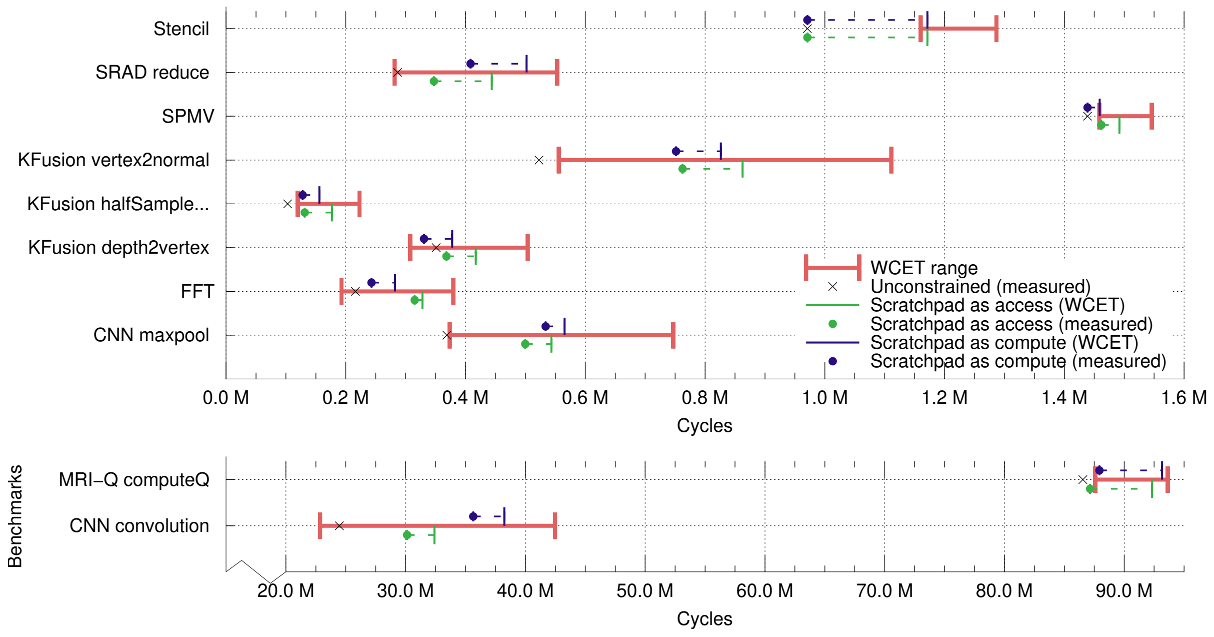


Figure 7.9: Performance of scheduling constraints relative to worst-case bounds.

Bar two exceptions, MRI-Q ComputeQ and SPMV, the distance between the lower- and upper bounds indicate that there is much potential for improving kernel run times by occupying compute- and access resources in parallel. As the figure shows, the most effective scheduling policy for a benchmark generally achieves half of that potential.

In line with the findings from Section 7.5.1, this graph shows that neither scheduling policy is universally better than the other. There appears to be an interesting (but unproven) correlation between which scheduling policy produces the best WCET for a given benchmark and which policy performs better during run-time.

The distance between the derived WCETs and the lower bound indicates that there is still potential for more efficient program phase scheduling. Although I leave research towards better policies as future work, I list two ideas which may be worth pursuing. Firstly, the two-resource scheduling idea could be further refined by allowing a developer or compiler to decide for each scratchpad access whether it must be scheduled as a compute- or an access phase. Secondly, explicit synchronisation points could allow limited drift in some parts of a work-group-pair. This way it may be possible to give the greedy scheduler more freedom in scheduling program phases in parallel while strictly limiting the number of candidate serialisations.

7.5.3 Software optimisation techniques

One of the benefits of using a simple in-order pipeline with performance isolation guarantees between program phases is that its performance can be modelled and simulated very accurately. As a result, software optimisations that improve pipeline throughput on average are also expected to improve the WCET of a kernel. In the remainder of this section I evaluate three software optimisation techniques and their impact on average-case and worst-case execution time: loop unrolling, instruction scheduling and 2D tiling.

7.5.3.1 Loop unrolling

In Section 6.3 I claimed that the MRI-Q computeQ and CNN convolution benchmarks are bound by control flow. Specifically, these benchmarks contain a tight loop that is iterated over a significant number of times. A cheap way to improve performance of these benchmarks, without the need for branch prediction hardware, is to perform loop unrolling.

To show the importance of this pass, I have manually unrolled the main loop of the MRI-Q computeQ kernel by a factor of 2 and 4. For the CNN convolution kernel, I unrolled the inner loop completely, eliminating 126 branches from each work-group's execution. The resulting kernel binaries are summarised in Table 7.3

Kernel	VGPRs	SGPRs	PRs	SP Alloc	Binary	
				B/WG	# insn	B
MRI-Q computeQ	8	8	0	4096	30	240
(unroll x2)	11	12	0	4096	40	320
(unroll x4)	17	20	0	4096	60	480
CNN convolution	4	16	2	6364	44	352
(unroll x7)	4	16	2	6364	66	528

Table 7.3: Program statistics of loop-unrolled benchmarks

Loop unrolling caused a modest increase in the number of registers used by the MRI-Q computeQ benchmark, as a result of both pre-loading more scalar kernel values from the scratchpad buffer with a single read, and of interleaving the computation of multiple iterations. The register usage of the convolution kernel has remained equal. In both cases the binary size has grown significantly when compared to their baselines, but in absolute terms these kernels remained small.

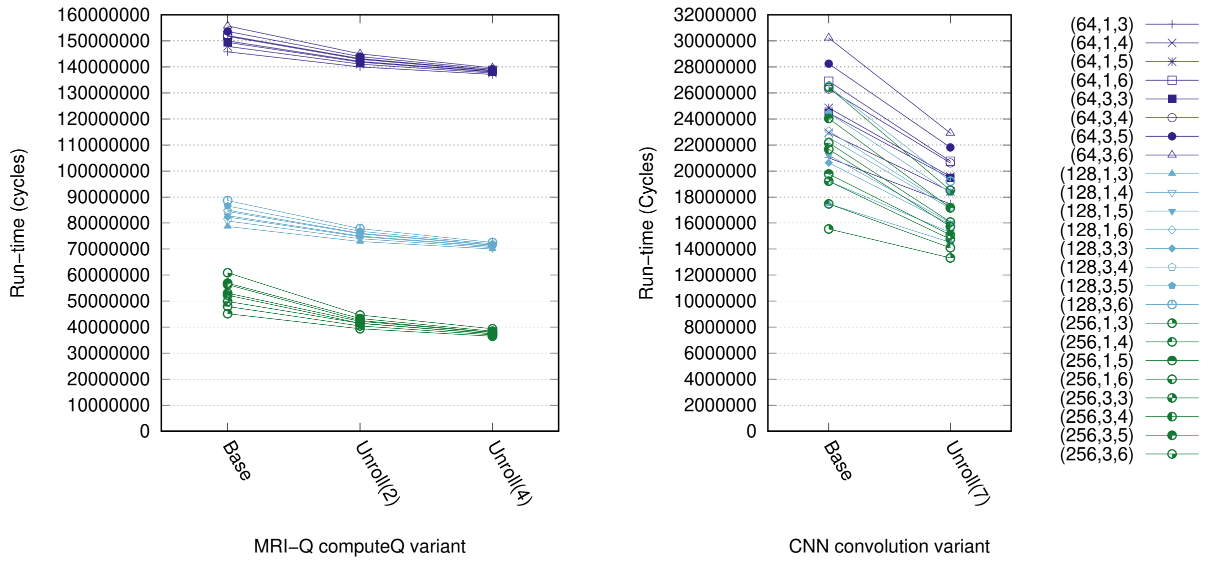


Figure 7.10: Performance effect of loop-unrolling on MRI-Q computeQ (left) and CNN convolution (right).

Figure 7.10 shows the average-case performance implications of unrolling the main loop in both kernels. For the shortest pipeline configurations, unrolling MRI-Q computeQ's loop by 2 reduces run-time by over 5.7M cycles and unrolling by 4 reduces run-time by over 8.6M cycles. For the longest pipeline configurations this benefit grows to over 10.7M and 16.1M cycles respectively. The absolute number of cycles saved by unrolling these loops shows little dependency on the number of SP-units in a SimdCluster, as the delay of a branch-induced pipeline flush does not depend on the number of SP-units.

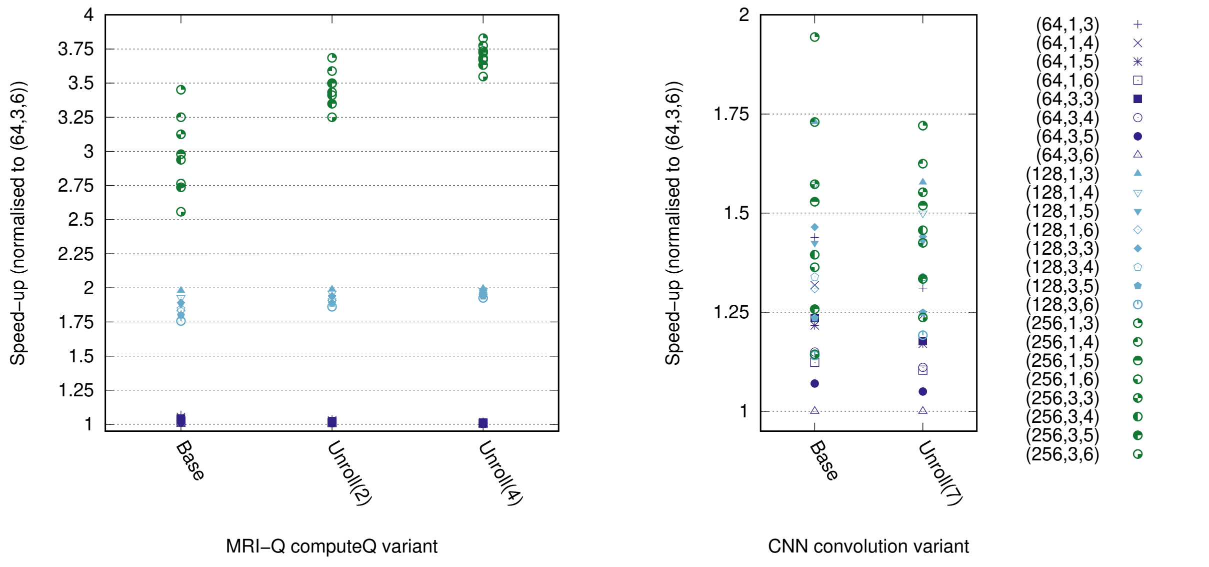


Figure 7.11: Pipeline depth sensitivity effect of loop-unrolling on MRI-Q computeQ (left) and CNN convolution (right).

As demonstrated in Figure 7.11, reducing the number of branches makes the benchmarks' performance less sensitive to pipeline depth variation. Where for a non-unrolled loop the pipeline length could impact the performance of the MRI-Q computeQ benchmark by as much as 35%, unrolling the benchmark by four reduces the performance variance between configurations of the same number of SP-units to less than 8%. Although this effect is less pronounced for the CNN convolution benchmarks, the distance between points for the same number of SP-units has decreased significantly as well.

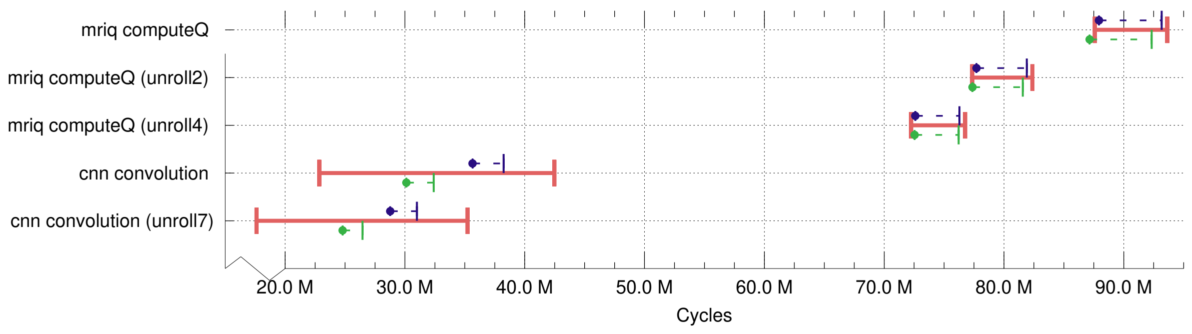


Figure 7.12: WCET bounds for loop-unrolled kernels

Figure 7.12 demonstrates how loop unrolling improves the WCET of both kernels. The WCET bound for MRI-Q computeQ under *the scratchpad as access* scheduling policy improved by 16.7% when unrolling the main loop by 4. For the CNN convolution benchmark, unrolling the inner loop reduces this WCET bound by 18.4%.

7.5.3.2 Instruction scheduling

Owing to its in-order pipeline, Sim-D is susceptible to performance degradation due to RAW-hazards. In particular, pipeline performance degrades when dependent scalar instructions are executed back-to-back. In the benchmarks studied, concentrated regions of scalar instructions tend to mainly serve three purposes:

- DRAM/scratchpad address and offset calculation,
- Loop invariants, and
- Data shared among threads within a work-group.

The first case, RAW-hazards in a region of scalar code calculating DRAM addresses, does not necessarily degrade a benchmark's performance. When multiple DRAM loads are issued in short succession, the compute resource tends to be underutilised. Hence eliminating the RAW hazards would simply allow the compute resources to block earlier as it waits for DRAM data to arrive. In this case DRAM is the bottleneck, and increased efficiency of compute will not lead to lower execution time of the program.

The second case, RAW-hazards in looping control flow code, allows for plenty of opportunity for eliminating RAW hazards. Loop invariant code often follows a pattern similar to the following: increment a counter, then subtracting a constant or variable from it and finally jumping backwards if the result of subtraction is smaller than 0. Although this loop control code is only three instructions long, the frequency of execution can be high. More so, because each instruction relies on the result of the previous, each iteration of the loop has the potential to block as many cycles as twice the number of pipeline stages between the first decode cycle and the final execute cycle.

The SRAD reduce benchmark is an example that follows the loop invariant code pattern outlined. By moving the two scalar instructions for loop counter update to an earlier point in the loop, RAW-hazards can be completely from the invariant code, reducing the benchmark's total average-case run-time by $\sim$3.5% for most pipeline lengths. For Sim-D's (128,3,5)-configuration, the WCET of this benchmark was reduced by $\sim$1.4%.

For the final case, scalar values shared among different work-items in the work-group, computation tends to cluster around the start of a loop to prepare values loaded from e.g. a small buffer to be used as an operand in vector instructions. As such code appears early in a loop, there is little opportunity of increasing the distance between instructions. However, when multiple such values exist in a loop body, clever interleaving can reduce the number of stall cycles paid for each hazard. Multiplicity of such values can be encouraged by unrolling the main loop.

7.5.3.3 2D tiling

Section 5.5.2 shows that for filter operations, the chosen tiling configuration has a marked impact on the provided DRAM throughput. To demonstrate the impact of improved DRAM throughput on benchmark run-time, I have run the KFusion vertex2normal benchmark with three different work-group configurations: 128×8, 64×16 and 32×32.

This benchmark implements a filter, each work-item reading its four directly-adjacent 3-vector elements. It has an in- and an out-buffer of 1920×480 words, 640×480 elements. In addition, two scratchpad buffers are used: one to load the tile of data used by a work-group, and a second to prepare the write-back data. Defining (x, y) as the dimensions of a work-group, this benchmark starts by reading a $(3x + 2) \times (y + 2)$ tile of data into an SP buffer. Next it performs 12 snoopy indexed reads into the scratchpad. Finally, it performs 3 writes into the output scratchpad buffer to store its resulting 3-vector element, after which a single transfer writes the tile of scratchpad values into the output buffer in DRAM.

Table 7.4 characterises the DRAM latency, scratchpad latency and execution times of this benchmark under the three tiling configurations. All measurements and analysis were performed for Sim-D parametrised with a (128,3,5)-configuration and a two bank-group

DDR4-3200AA DRAM configuration.

Tiling	Read LID	Write LID	SP idx	Avg	DRAM	WCET	
Config.	min - max	min - max	cycs	cycs	cycs	SP as access	SP as compute
(128,8)	1090 - **1153**	**862 - 945**	122	526927	**389449**	**807151**	815751
(64,16)	**1058** - 1179	863 - 1039	112	**515737**	399029	810751	**805051**
(32,32)	1138 - 1295	863 - 1218	**109**	522664	412972	862701	826301

Table 7.4: Performance of three different tiling configurations for KFusion vertex2normal on Sim-D (128,3,5). DDR4-3200AA 2 bank-groups.

Looking solely at DRAM performance, we can see that the worst-case LID for DRAM reads and writes are minimised when using an (128,8) tiling configuration. This is in line with findings in Section 5.5.2. However, while these two DRAM transfers reached their maximum efficiency, the scratchpad performance of this configuration is the lowest of the three as a result of requiring the largest scratchpad buffer.

When evaluating average-case performance, it turns out that the inefficiency of the 12 scratchpad accesses outweighs the benefits gained from the faster DRAM transfers. As a result, the (128,8) tiling configuration performs worst when running without scheduling constraints despite spending the least amount of time on DRAM transfers. The best tiling configuration for average-case performance is the (64,16) configuration, outperforming the (128,8)-case by ∼2.1%.

A similar picture emerges when looking at the WCETs of this benchmark. depending on whether the work-group scheduler treats scratchpad requests as DRAM accesses or as compute, the best tiling configuration is either (128,8) or (64,4), with the latter allowing for a ∼0.3% lower WCET.

In all cases, the data-conserving (32,32) tiling configuration performs worst as a result of its inefficient DRAM transfers. The difference between the achieved WCET bounds for the (32,32)- and (64,16) tiling configurations is 2.6%, demonstrating the value of 2D tiling optimisation.

7.5.4 Takeaway points

Evaluation has shown the following properties of the WCET analysis methods and scheduling policies:

- The *scratchpad as access* and *scratchpad as execute* policies could degrade run-time performance by up to 31.5% for applications that utilise the scratchpad. 17.5% of this degradation can be accounted for by forcing work-groups to execute in pairs. On average, the performance degradation caused by each benchmark's best scheduling policy is 10.8%,

- Using the WCET derivation algorithm described in Section 7.4, bounds can be derived for both scheduling policies with, ignoring outliers, a pessimism averaging at ∼13.5-14.3%,
- Outliers with pessimism exceeding 600% are the result of requiring indexed transfers into large buffers. Bounds on such requests are pessimistic in line with conclusions from Section 5.4,
- Neither scheduling policy performs better than the other under all cases,
- Software optimisation techniques, such as instruction scheduling and 2D tiling, can reduce the WCET of applications running on Sim-D by a few percent. Specifically, loop unrolling is shown to improve both average- and worst-case execution times by up to 18.4% for relevant benchmarks.

7.6 Summary

In this chapter I introduced a theoretical framework and a WCET computation algorithm for kernel-instances running on the Sim-D architecture. The resulting WCET bounds are shown to be both safe, and sufficiently tight to permit practical use of this architecture in real systems.

The major problem I identified is that of dynamically scheduling the *program phases* of work-groups. Based on observations, there is a strong indication that an unconstrained greedy work-group scheduler is able to generate a large number of different schedules, each of which would potentially be a worst-case schedule. Computing the WCET for all possible schedules is deemed infeasible. The solution to reducing this search space is two-fold. First I introduced the notion of a serialisation as an abstraction capturing the order in which program phases can be scheduled, and proved that there exists a *worst-case minimal valid schedule* for each serialisation. In the architecture, I then made small modifications to the work-group-scheduler to ensure that execution can only follow a single *worst-case serialisation.* Together, this allowed me to implement a static WCET derivation algorithm that performs its *bound calculation* in linear time.

Two such scheduling policies were proposed: *scratchpad as compute* and *scratchpad as access.* Both of these policies effectively treat Sim-D as an architecture consisting of two resources, causing the worst-case serialisation to be the serialisation for which the two active work-groups swap their resources after each program phase. I have proven that any other valid serialisation permitted by these two scheduling policies must result in an equal or better *worst-case minimal valid schedule.*

Using experiments, I have shown that these scheduling constraints can degrade average case performance by up to 31.5%. On average, degradation of performance is 13.5% and

14.3% for the *scratchpad as compute* and *scratchpad as access* policies respectively. Picking the best-performing policy on a per-kernel basis results in a performance degradation of 10.8% on average. Disregarding three outliers, calculated WCETs are on average tight within 13.3% and 14.2% under the *scratchpad as compute* and *scratchpad as access* policies respectively. As predicted in Section 5.4, the outliers are the result of benchmarks requiring indexed transfers into large buffers. For such transfers no practical techniques exist that can predict a tight and low LID.

One area for future work is in compiler-assisted optimisations. I have shown that existing techniques like loop unrolling and instruction scheduling are capable of positively impacting both average- and worst-case performance of kernels. Besides researching other interesting optimisations (e.g. loop merging [127]), there is merit to investigating whether Sim-D's large coalesced data transfers introduce new challenges when applying existing optimisations.

CHAPTER 8

Conclusion

In this dissertation I studied the thesis that **an efficient wide-SIMD accelerator can feasibly be designed that permits the derivation of safe and tight bounds on the execution time of data-parallel programs**. To support this thesis I introduced the Sim-D architecture: a wide-SIMD processor designed for hard real-time systems. Like a GPU, Sim-D performs hardware strip-mining to maximally benefit from the parallelism present in the targeted data parallel programs. To meet the demands of HRT systems, Sim-D is designed to provide performance isolation between its compute- and memory resources. At any time, Sim-D has up to two work-groups run in parallel, with the WCET of each work-group's program phases free of interference from the other work-group.

To make efficient use of today's DDR4 DRAM, Sim-D schedules its work in work-groups of 1024 work-items. DRAM requests are issued as scalar operations, requesting the data for an entire work-group. To support such large explicitly-coalesced transfers, I presented a DRAM controller that can process requests for large 1D and 2D blocks of data. For common-case linear transfers of 4KiB, Sim-D is able to provably achieve a DRAM bus utilisation exceeding 78%. Filter kernels that use tiling to optimise data flow can achieve a bus utilisation over 70% for 2D block transfers of a similar size.

Experiments show that Sim-D is capable of achieving performance on par with an embedded-grade NVIDIA Tegra K1 GPU. A condition for meeting such performance is providing sufficient bandwidth between the scratchpad and the vector register file. Some benchmarks perform poorly on Sim-D. Such benchmarks have in common that they require indexed transfers from large buffers in DRAM. I have demonstrated that such transfers are expected to perform poorly in the worst case due to DRAM properties, and that known techniques using associative caches are not expected to improve the worst case. To alleviate the problem, I introduced a mechanism providing *snoopy indexed transfers*. Such transfers can provably perform better than iterative indexed transfers if the size of the buffer is either small or if at compile time the relevant section for this indexed transfer can be reduced to a 1D or 2D block.

Finally, I presented a solution to derive safe and tight bounds on the execution time of kernel-instances running on Sim-D. On the architectural side, I present two work-group scheduling policies, *scratchpad as compute* and *scratchpad as access*, that restrict the possible interleavings of program phases from different work-groups at run-time. Subsequently, I introduced a novel method that determines a kernel-instance's WCET by evaluating a single *worst-case minimal valid schedule* for each policy. Evaluation showed that if developers choose the best policy for their kernel, the average performance loss of these work-group scheduling policies is 8.9%. In return, WCET analysis can provide a bound on execution time which is safe and, for those benchmarks that don't require indexed transfers into large buffers, are tight within 13.5% and 14.3% for the respective scheduling policies.

8.1 Future work

My main motivation for the work on Sim-D was to address the discrepancy between the recent surge in demand for data-parallel processing in safety-critical systems on the one hand, and the scarce offer of architectural solutions that deliver such processing capabilities for HRT systems on the other. Given the breadth of applications in the domain of safety-critical systems, the choice for a GPU-like accelerator seemed natural. However, given the stringent power constraints for certain classes of safety-critical systems, I expect there is additional scope for research in domain-specific HRT accelerators, for example for machine learning or computer vision workloads. It is my hope that the contributions on novel large-transfer hard real-time (HRT) DRAM controllers, processor-level work scheduling and WCET analysis of data-parallel programs can transcend the presented Sim-D architecture and inspire novel research in the broader space of HRT data-parallel architectures.

Looking at the future of Sim-D, I believe that the work presented is merely a starting point. There are many challenges left to take Sim-D from its current simulation model to a tangible chip, and to scale Sim-D's performance to that of a high-end GPU. I next highlight some of these challenges.

Firstly, for practical purposes Sim-D will require a front-end. Besides kernel launch, this front-end should facilitate the upload and download of buffers to Sim-D's dedicated DRAM. From a research perspective, it is interesting to analyse how such transfers can interfere with the DRAM transfers of kernel execution, and how the worst-case response time of a kernel-instance is calculated when taking these interference effects into account.

Secondly, in Section 6.3 I showed that Sim-D's current performance approaches that of an embedded-grade GPU. Such performance may be valuable for some applications, but it is currently unknown how its design can be scaled up to achieve the performance of a discrete high-end GPU. The main challenge in scalability lies in widening the DRAM data bus. This can be done in two ways: creating a wider channel, or adding more channels. For either solution it is currently unknown how to sustain good DRAM data bus utilisation and process such large streams of data.

One option to increase the compute throughput of Sim-D is by replicating SimdClusters. This complicates the worst-case timing analysis as, from DRAM's perspective, it increases the number of requestors in a system. However, the presence of multiple SimdClusters also brings new opportunities for sharing the device spatially among multiple kernel-instances. Spatial multi-tasking can be beneficial when scheduling kernels with a wide range of latencies, as schedulability of sets of tasks on multiprocessors tends to improve when tasks of similar deadlines are clustered on each core.

In a similar vein, it is interesting to study the impact of temporal multi-tasking. Extending Sim-D with preemption support allows high-priority kernels to preempt running instances, increasing the schedulability of sets of hard real-time tasks on the processor. It is currently unknown how such a preemption mechanism might be implemented, and what the resulting preemption-related run-time overheads would be.

Both temporal- and spatial multi-tasking bring architectural- and theoretical challenges surrounding real-time scheduling on Sim-D and the determination of WCRTs. For temporal multi-tasking, challenges include the implementation of kernel scheduling mechanisms that follow real-time policies, such as FP or EDF, and facilitating the preemption of kernels either at arbitrary times (full preemption) or at specified preemption points (limited preemption [6, 128]). Similarly, for spatial multi-tasking where different tasks run on different compute units in parallel, real-time multi-core scheduling policies must be implemented to permit schedulability analysis of task sets in the presence of shared resources like the DRAM bus. A starting point for schedulability analysis under spatial multi-tasking could be the Distributed Priority Ceiling Protocol (D-PCP) [129], which allows to treat the DRAM bus as a remote processor running the DRAM requests as "local agents" on behalf of the kernels running on the different compute units.

As a first step towards solving these schedulability problems, it is important to define the notion of a task in the context of a GPU-like accelerator. Modelling each kernel as a separate task is probably not a good approach, as the outside world may not dictate a deadline on a per-kernel basis. For example, a neural network algorithm used for classification generally consists of many layers, each layer implemented as its own kernel. Real-time deadlines can be defined for the classification task as a whole, but imposing individual deadlines on each layer is needlessly restrictive. Defining a task as a set of kernels thus seems more productive, but this comes with new challenges. One such challenge is how to determine the maximum preemption-induced blocking time of a task under limited-

or full preemption. This maximum blocking time differs from kernel to kernel, but it is not yet known whether this information can be used to improve schedulability of task sets.

As a final avenue for future research, in Section 7.5.3 I demonstrated several common software optimisation techniques and their positive impact on a kernel's WCET. To make such techniques practically available to system developers, a good compiler infrastructure is indispensable. The development of a compiler opens up new opportunities for research, both on compiling existing languages like OpenCL C for a mixed scalar-vector processor like Sim-D, as well as on investigating compiler optimisation techniques from the perspective of their projected impact on WCET.

Bibliography

[1] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. An Open Approach to Autonomous Vehicles. *Micro, IEEE*, 35(6):60–68, Nov 2015.

[2] A. Eklund, P. Dufort, D. Forsberg, and S.M. LaConte. Medical image processing on the GPU – Past, present and future. *Medical Image Analysis*, 17(8):1073 – 1094, 2013.

[3] NVIDIA Tegra X1 - NVIDIA'S New Mobile Superchip, 2015. Retreived March 2020, http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf.

[4] A. Betts and A. Donaldson. Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis. In *25th Euromicro Conference on Real-Time Systems*, pages 193–202, July 2013.

[5] Y. Huangfu and W. Zhang. Static WCET Analysis of GPUs with Predictable Warp Scheduling. In *IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 101–108, May 2017.

[6] R. Spliet and R. Mullins. The case for limited-preemptive scheduling in GPUs for real-time systems. In *ECRTS, Operating Systems Platforms for Embedded Real-Time applications*, Jul 2018.

[7] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011.

[8] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2019.

[9] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The ILLIAC IV Computer. *IEEE Transactions on Computers*, C-17(8):746–757, Aug 1968.

[10] Y. Lee, C. Schmidt, A. Ji-Hung Ou, A. Waterman, and K. Asanovic. The Hwacha Vector-Fetch Architecture Manual, Version 3.8.1. Technical report, dec 2015.

[11] ARM. *ARM Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for ARMv8-A*, Feb 2020. Retreived 29th March 2020, https://developer.arm.com/documentation/ddi0584/ag/.

[12] H. Cheng. Vector pipelining, chaining, and speed on the IBM 3090 and Cray X-MP. *Computer*, 22(9):31–42, Sep. 1989.

[13] M. Weiss. Strip Mining on SIMD Architectures. In *Proceedings of the 5th International Conference on Supercomputing*, ICS '91, page 234–243, New York, NY, USA, 1991. Association for Computing Machinery.

[14] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38, 1997.

[15] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Oct. 2019. Retrieved 2nd March 2020, https://software.intel.com/sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf.

[16] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[17] D. Kirk et al. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[18] NVIDIA. NVIDIA GeForce GTX 680, whitepaper. Technical report, 2012. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[19] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/GK210, 2014.

[20] Qualcomm Technologies Inc. Architecture of the Hexagon 680 DSP for Mobile Imaging and Computer Vision, Aug. 2015. Retrieved 2nd March 2020, https://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.24-Monday-Epub/HC27.24.20-Multimedia-Epub/HC27.24.211-Hexagon680-Codrescu-Qualcomm.pdf.

[21] S. Wilson. FirePath Processor Architecture and Microarchitecture, Aug 2002. Retrieved 2nd March 2020, https://www.hotchips.org/wp-content/uploads/hc_archives/hc14/3_Tue/23_wilson.pdf.

[22] Analog Devices. SHARC+ Core Programming Reference, Aug. 2019. Retrieved 2nd March 2020, https://www.analog.com/media/en/dsp-documentation/processor-manuals/SC58x-2158x-prm.pdf.

[23] J. Xie. NVIDIA RISC-V Evaluation Story. 4th RISC-V Workshop, 2016.

[24] N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[25] F. Sijstermans. The NVIDIA deep learning accelerator. In *Hot Chips*, 2018.

[26] GraphCore. How to build a processor for machine intelligence (part 2), Jul. 2017. Retrieved on 29th March 2020, https://www.graphcore.ai/posts/how-to-build-a-processor-for-machine-intelligence-part-2.

[27] X. Wang, C. Kiwus, C. Wu, B. Hu, K. Huang, and A. Knoll. Implementing and Parallelizing Real-time Lane Detection on Heterogeneous Platforms. In *IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8, July 2018.

[28] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, 2010.

[29] AMD. AMD APP SDK OpenCL Optimization Guide, Aug. 2015. Retrieved on 5 Feburary 2020, http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide2.pdf.

[30] Intel. OpenCL Developer Guide for Intel Processor Graphics, Mar. 2019. Retrieved on 5 Feburary 2020, https://software.intel.com/en-us/iocl-opg-optimizing-opencl-usage-with-intel-processor-graphics.

[31] NVIDIA. NVIDIA OpenCL Best Practices Guide, Jul 2009.

[32] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[33] S.K. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized Multiframe Tasks. *Real-Time Systems*, 17:5–22, 1999.

[34] M. Stigge, P. Ekberg, N. Guan, and W. Yi. The Digraph Real-Time Task Model. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 71–80, April 2011.

[35] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[36] M.L. Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings IF IP Congress*, 1974.

[37] M. Paolieri, E. Quiñones, and F.J. Cazorla. Timing Effects of DDR Memory Systems in Hard Real-time Multicore Architectures: Issues and Solutions. *ACM Trans. Embed. Comput. Syst.*, 12(1s):64:1–64:26, March 2013.

[38] B. Akesson and K. Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[39] B. Akesson, W. Hayes Jr., and K. Goossens. Classification and analysis of predictable memory patterns. In *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 367–376, Aug 2010.

[40] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A Predictable SDRAM Memory Controller. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '07, pages 251–256, New York, NY, USA, 2007. ACM.

[41] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens. A Reconfigurable Real-time SDRAM Controller for Mixed Time-criticality Systems. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 2:1–2:10, Piscataway, NJ, USA, 2013. IEEE Press.

[42] Y. Li, B. Akesson, and K. Goossens. Dynamic Command Scheduling for Real-Time Memory Controllers. In *26th Euromicro Conference on Real-Time Systems*, pages 3–14, July 2014.

[43] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. In *Proceedings of the 15th Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation*, volume 1, pages 111–119 vol.1, Mar 1996.

[44] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 3–14, Aug 2008.

[45] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero. An Analyzable Memory Controller for Hard Real-Time CMPs. *IEEE Embedded Systems Letters*, 1(4):86–90, Dec 2009.

[46] Y. Li, H. Salunkhe, J. Bastos, O. Moreira, B. Akesson, and K. Goossens. Mode-controlled data-flow modeling of real-time memory controllers. In *13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pages 1–10, Oct 2015.

[47] Y. Li, B. Akesson, K. Lampka, and K. Goossens. Modeling and Verification of Dynamic Command Scheduling for Real-Time Memory Controllers. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.

[48] S. Goossens, K. Chandrasekar, B. Akesson, and K. Goossens. ILP Formulation and Solution Strategies for Memory Command Scheduling. 2014.

[49] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni. A Rank-Switching, Open-Row DRAM Controller for Time-Predictable Systems. In *26th Euromicro Conference on Real-Time Systems*, pages 27–38, July 2014.

[50] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proceedings of the 9th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 99–108, Oct 2011.

[51] F. Farshchi, P.K. Valsan, R. Mancuso, and H. Yun. Deterministic Memory Abstraction and Supporting Multicore System Architecture. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:25, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[52] ARM ltd. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition.* 2007.

[53] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[54] C. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. In *Proceedings 11th Real-Time Systems Symposium*, pages 72–81, Dec 1990.

[55] N. Altman and N. Weiderman. Timing Variation in Dual Loop Benchmark. *Ada Lett.*, VIII(3):98–106, April 1988.

[56] P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, 1989.

[57] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proceedings 21st IEEE Real-Time Systems Symposium*, pages 163–174, Nov 2000.

[58] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *European Conference on Parallel Processing*, pages 1298–1307. Springer, 1997.

[59] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proceedings 4th IEEE Real-Time Technology and Applications Symposium*, pages 12–21, June 1998.

[60] C. Healy and D. Whaley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proceedings 5th IEEE Real-Time Technology and Applications Symposium*, pages 79–88, June 1999.

[61] N. Zhang, A. Burns, and M. Nicholson. Pipelined Processors and Worst Case Execution Times. *Real-Time Syst.*, 5(4):319–343, October 1993.

[62] S.-S. Lim, Y.H. Bae, G.T. Jang, B.-D. Rhee, S.L. Min, C.Y. Park, H. Shin, K. Park, Moon S.-M., and C.S. Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[63] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proceedings 6th International Conference on Real-Time Computing Systems and Applications. RTCSA'99*, pages 88–95, Dec 1999.

[64] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems*, 18(2):249–274, 2000.

[65] R.D. Arnold, F. Mueller, D.B. Whalley, and M.G. Harmon. Bounding worst-case instruction cache performance. In *Proceedings Real-Time Systems Symposium*, pages 172–181, Dec 1994.

[66] J.C. Liu and H.J. Lee. Deterministic upperbounds of the worst-case execution times of cached programs. In *Proceedings Real-Time Systems Symposium*, pages 182–191, Dec 1994.

[67] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings 3rd IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.

[68] R. Bourgade, C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Accurate analysis of memory latencies for WCET estimation. In *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, Rennes, France, Oct 2008. Isabelle Puaut.

[69] V. Suhendra and T. Mitra. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores. In *Proceedings 45th Annual Design Automation Conference*, DAC '08, page 300–303, New York, NY, USA, 2008. Association for Computing Machinery.

[70] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *Proceedings. Real-Time Systems Symposium*, pages 229–237, Dec 1989.

[71] C. Healy and D. Whaley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proceedings 5th IEEE Real-Time Technology and Applications Symposium*, pages 79–88, June 1999.

[72] R.M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Plenum Press, 1972.

[73] J.E. Smith. Decoupled Access/Execute Computer Architectures. *SIGARCH Comput. Archit. News*, 10(3):112–119, April 1982.

[74] L.W. Howes, A. Lokhmotov, P.H.J. Kelly, and A.F. Donaldson. Decoupled Access/Execute metaprogramming for GPU-accelerated systems. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC*, 2009.

[75] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras. Towards More Efficient Execution: A Decoupled Access-execute Approach. In *Proceedings 27th International ACM Conference on Supercomputing*, ICS '13, pages 253–262, New York, NY, USA, 2013. ACM.

[76] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, 2012.

[77] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo. Memory-Aware Scheduling of Multicore Task Sets for Real-Time Systems. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 300–309, 2012.

[78] Z. Chen, D. Kaeli, and N. Rubin. Characterizing scalar opportunities in GPGPU applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 225–234, April 2013.

[79] P.R. Panda. SystemC: A Modeling Platform Supporting Multiple Design Abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 75–80, New York, NY, USA, 2001. ACM.

[80] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, Jan 2012.

[81] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, Jan 2016.

[82] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens. *DRAMPower: Open-source DRAM Power & Energy Estimation Tool*, 2012. Source retrieved 12 December, 2017 from https://github.com/tukl-msd/DRAMPower.

[83] M. Kościelnicka, B. Skeggs, M. Peres, M. Lankhorst, R. Spliet, C. Bumiller, M. Ślusarz, E. Velikov, F. Jerez, K. Herbst, R. Kidd, and many others. *Envytools: Tools for people envious of nvidia's blob driver*, Jan. 2020. https://envytools.readthedocs.io/, generated from https://github.com/envytools/envytools.

[84] N.J. Foskett, R. J. Prevett Jr., and S. Treichler. Method and system for performing pipelined reciprocal and reciprocal square root operations, Oct 2006. US Patent 7,117,238.

[85] J. Piñeiro, S. F. Oberman, J. . Muller, and J. D. Bruguera. High-speed function approximation using a minimax quadratic interpolator. *IEEE Transactions on Computers*, 54(3):304–318, March 2005.

[86] S. F. Oberman and M. Y. Siu. A high-performance area-efficient multifunction interpolator. In *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, pages 272–279, June 2005.

[87] Whitepaper: NVIDIA Tesla P100, 2016.

[88] J. Coke, N. Cooray, E. Gamsaragan, P. Smith, K. Yoon, J Abel, and A. Valles. Improvements in the Intel Core2 Penryn Processor Family Architecture and Microarchitecture. Technical report, Intel Corporation, 2008.

[89] N. Juffa. Pipelined integer division using floating-point reciprocal, May 2007. US Patent 8,140,608.

[90] V. Vanhoucke, A. Senior, and M.Z. Mao. Improving the speed of neural networks on CPUs. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS*, 2011.

[91] K. Hwang and W. Sung. Fixed-point feedforward deep neural network design using weights +1, 0, and -1. In *IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6, 2014.

[92] B.W. Coon, J.E. Lindholm, and S.D. Tzvetkov. Structured programming control flow using a disable mask in a SIMD architecture, Nov 2009. US Patent 7,617,384.

[93] C. Cakir, R. Ho, J. Lexau, and K. Mai. Modeling and Design of High-Radix On-Chip Crossbar Switches. In *Proceedings of the 9th International Symposium on Networks-on-Chip*, NOCS '15, pages 20:1–20:8, New York, NY, USA, 2015. ACM.

[94] M. Gebhart, D.R. Johnson, D. Tarjan, S.W. Keckler, W.J. Dally, E. Lindholm, and K. Skadron. Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors. *SIGARCH Comput. Archit. News*, 39(3):235–246, June 2011.

[95] J.E. Lindholm, M.Y. Siu, S. S. Moy, S. Liu, and J.R. Nickolls. Simulating multiported memories using lower port count memories, Mar 2008. US Patent 7,339,592.

[96] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N.S. Kim, T.M. Aamodt, and V.J. Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. *SIGARCH Comput. Archit. News*, 41(3):487–498, June 2013.

[97] J. Lim, N.B.Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung. Power Modeling for GPU Architectures Using McPAT. *ACM Trans. Des. Autom. Electron. Syst.*, 19(3):26:1–26:24, June 2014.

[98] M. Yoshimoto, K. Anami, H. Shinohara, T. Yoshihara, H. Takagi, S. Nagao, S. Kayano, and T. Nakano. A divided word-line structure in the static RAM and its application to a 64K full CMOS RAM. *IEEE Journal of Solid-State Circuits*, 18(5):479–485, Oct 1983.

[99] M. Abdel-Majeed and M. Annavaram. Warped register file: A power efficient register file for GPGPUs. In *IEEE 19th Int. Symp. on High Performance Comp. Arch. (HPCA)*, pages 412–423, Feb 2013.

[100] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens. *CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model*, 2017. Source retrieved 16 January 2020 from https://github.com/HewlettPackard/cacti.

[101] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174, April 2009.

[102] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012.

[103] Micron Technology Inc. 8Gb: x4, x8, x16 DDR4 SDRAM datasheet, 2017. Retrieved 15th June, 2018 from https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf.

[104] S. Marchesin. *Linux Graphics Drivers: an Introduction*, Mar. 2012.

[105] nVidia corp. *Parallel Thread Execution ISA Version 6.5*, Nov. 2019. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[106] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136, Oct 2011.

[107] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.M.W. Hwu, H. Li, M.S. Müller, W.E. Nagel, M. Perminov, P. Shelepugin, K. Skadron, J. Stratton, A. Titov, K. Wang, M. van Waveren, B. Whitney, S. Wienke, R. Xu, and K. Kumaran. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 46–67, Cham, 2015. Springer International Publishing.

[108] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 44 –54, oct. 2009.

[109] Parboil benchmark suite, 2010. http://impact.crhc.illinois.edu/Parboil/parboil.aspx.

[110] D. Bates. CNN kernels. personal communication, Sept. 2016.

[111] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

[112] L. Nardi, B. Bodin, M.Z. Zia, J. Mawer, A. Nisbet, P.H.J. Kelly, A.J. Davison, M. Luján, M.F.P. O'Boyle, G. Riley, N. Topham, and S. Furber. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2015. arXiv:1410.2167.

[113] Y. Yu and S.T. Acton. Speckle reducing anisotropic diffusion. *IEEE Transactions on Image Processing*, 11(11):1260–1270, Nov 2002.

[114] Y. Saad. Krylov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, 1989.

[115] D. Bates, A. Bradbury, A. Koltes, and R.D. Mullins. Exploiting Tightly-Coupled Cores. *Journal of Signal Processing Systems*, 80(1):103–120, Jul 2015.

[116] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan. Optimal loop unrolling for GPGPU programs. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–11. IEEE, 2010.

[117] K. Knobe, J.D. Lukas, and G.L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102 – 118, 1990.

[118] I-Jui Sung, Geng Daniel Liu, and Wen-Mei W Hwu. DL: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar)*, pages 1–11. IEEE, 2012.

[119] B. Akesson. *Predictable and Composable System-on-Chip Memory Controllers*. PhD thesis, Eindhoven University of Technology, February 2010. ISBN: 978-90-386-2169-2.

[120] NVIDIA Corporation. *Tegra K1 Embedded Platform Design Guide*, July 2016. Retrieved February 14th, 2017 from https://developer.nvidia.com/embedded/downloads.

[121] S. Goossens, K. Chandrasekar, B. Akesson, and K. Goossens. Power/Performance Trade-Offs in Real-Time SDRAM Command Scheduling. *IEEE Transactions on Computers*, 65(6):1882–1895, June 2016.

[122] A. Agarwal, S. Hsu, S. Mathew, M. Anders, H. Kaul, F. Sheikh, and R. Krishnamurthy. A 128x128b high-speed wide-and match-line content addressable memory in 32nm CMOS. In *Proceedings of the ESSCIRC*, pages 83–86, Sept 2011.

[123] N. Onizawa, S. Matsunaga, V. C. Gaudet, W. J. Gross, and T. Hanyu. High-Throughput Low-Energy Self-Timed CAM Based on Reordered Overlapped Search Mechanism. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(3):865–876, March 2014.

[124] ARM ltd. *ARM Cortex-A72 MPCore Processor Technical Reference Manual.* 2016.

[125] Intel. Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide, Oct. 2019. Retrieved on 18th February 2020, https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407/mwh1391807508964/mwh1391807512554/mwh1391807512881.html.

[126] NVIDIA. NVIDIA Tesla V100 GPU Architecture. page 18, 2017.

[127] T.D. Han and T.S. Abdelrahman. Reducing Divergence in GPGPU Programs with Loop Merging. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, page 12–23, New York, NY, USA, 2013. Association for Computing Machinery.

[128] S.K. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conf. on Real-Time Systems (ECRTS'05)*, pages 137–144, July 2005.

[129] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real-Time Systems Symposium*, pages 259–269, Dec 1988.

APPENDIX A

ISA

A.1 Conventions

For all instructions, an “s” prefix denotes a scalar instruction. The “i” prefix is used for integer arithmetic. When no prefix is given, the instruction is either a floating point or untyped vector instruction.

Optional operands are denoted between [brackets].

Special purpose vector and scalar registers can be referred to either by their alias, e.g. vc.tid_x, or by their index, e.g. vc4. We recommend the use of aliassed registers for code readability. A full list of all special purpose registers is given in Section A.2.

A.2 Register specifications

Special vector registers:

Idx	Alias	Perm.	Description
0	vc.ctrl_run	rw	Run control mask.
1	vc.ctrl_break	rw	Break control mask.
2	vc.ctrl_ret	rw	Return control mask.
3	vc.ctrl_exit	rw	Exit control mask.
4	vc.tid_x	ro	Thread ID in X-dimension.
5	vc.tid_y	ro	Thread ID in Y-dimension.
6	vc.lid_x	ro	Local thread ID (within work-group) in X-dimension.
7	vc.lid_y	ro	Local thread ID (within work-group) in X-dimension.
8	vc.zero	ro	Hard-coded 0.
9	vc.one	ro	Hard-coded integer 1.
10	vc.mem_idx	rw	Indexes for CAM based memory r/w.
11	vc.mem_data	rw	Values to read/write for CAM based memory r/w

Special scalar registers:

Idx	Alias	Perm.	Description
0	sc.dim_x	ro	Kernel size (#threads) in X-dimension.
1	sc.dim_y	ro	Kernel size (#threads) in Y-dimension.
2	sc.wg_off_x	ro	Work-group offset within kernel invocation, TID_X of thread 0.
3	sc.wg_off_y	ro	Work-group offset within kernel invocation, TID_Y of thread 0.
4	sc.wg_width	ro	Width of a workgroup as scheduled.
5	sc.sd_words	rw	Stride descriptor: Numer of words fetched in every period.
6	sc.sd_period	rw	Stride descriptor: Numer of words in a period.
7	sc.sd_period_cnt	rw	Stride descriptor: Numer of periods to repeat.

A.3 Floating point arithmetic

A.3.1 nop

No operation

Syntax nop

A.3.2 mul

Floating-point multiply

Syntax mul[.op] vdst, v0, s1
mul[.op] vdst, v0, v1
mul[.op] vdst, v0, imm1

op $\in$ {,neg}

Description For each vector element n, performs vdst[n] = v0[n] * v1[n]. Operand 1 may also be a scalar register or immediate.

.op	Description
(omit)	Normal operation.
neg	Negate second operand.

A.3.3 add

Floating-point addition

Syntax add[.op] vdst, v0, s1
add[.op] vdst, v0, v1
add[.op] vdst, v0, imm1

op $\in$ {,neg}

Description For each vector element n, performs vdst[n] = v0[n] + v1[n]. Operand 1 may also be a scalar register or immediate.

.op	Description
(omit)	Normal operation.
neg	Negate second operand.

A.3.4 mad

Multiply-Accumulate

Syntax mad[.op] vdst, v0, s1, v2
mad[.op] vdst, v0, v1, v2
mad[.op] vdst, v0, imm1, v2

op $\in$ {,neg}

Description For each vector element n, performs vdst[n] = v0[n] * v1[n] + v2[n]. Operand 1 may also be a scalar register or immediate

.op	Description
(omit)	Normal operation.
neg	Negate second operand.

A.3.5 min

Floating-point min

Syntax min vdst, v0, s1
min vdst, v0, v1
min vdst, v0, imm1

Description For each vector element n, performs vdst[n] = min(v0[n], v1[n]). Operand 1 may also be a scalar register or immediate.

A.3.6 max

Floating-point max

Syntax max vdst, v0, s1
max vdst, v0, v1
max vdst, v0, imm1

Description For each vector element n, performs vdst[n] = max(v0[n], v1[n]). Operand 1 may also be a scalar register or immediate.

A.3.7 abs

Floating-point absolute

Syntax abs vdst, v0

Description For each vector element n, performs vdst[n] = |v0[n]|.

A.4 Reciprocal/Trigonometry (expensive FP arith)

A.4.1 rcp

Floating-point reciprocal

Syntax rcp vdst, v0

Description For each vector element n, performs vdst[n] = 1 / v0[n]

A.4.2 rsqrt

Floating-point reciprocal square root

Syntax rsqrt vdst, v0

Description For each vector element n, performs vdst[n] = 1 / sqrt(v0[n])

A.4.3 sin

Floating-point sine

Syntax sin vdst, v0

Description For each vector element n, performs vdst[n] = sin(v0[n])

A.4.4 cos

Floating-point cosine

Syntax cos vdst, v0

Description For each vector element n, performs vdst[n] = cos(v0[n])

A.5 Integer/Boolean arithmetic

A.5.1 iadd

(Signed) integer addition

Syntax iadd vdst, v0, s1
iadd vdst, v0, v1
iadd vdst, v0, imm1

Description For each vector element n, performs vdst[n] = v0[n] + v1[n]. Operand 1 may also be a scalar register or immediate.

A.5.2 isub

Signed integer subtraction

Syntax isub vdst, v0, s1
isub vdst, v0, v1
isub vdst, v0, imm1

Description For each vector element n, performs vdst[n] = v0[n] - v1[n]. Operand 1 may also be a scalar register or immediate.

A.5.3 imul

Signed integer multiply

Syntax imul vdst, v0, s1
imul vdst, v0, v1
imul vdst, v0, imm1

Description For each vector element n, performs vdst[n] = v0[n] * v1[n]. Operand 1 may also be a scalar register or immediate.

A.5.4 imad

Signed integer Multiply-Accumulate

Syntax imad vdst, v0, s1, v2
imad vdst, v0, v1, v2
imad vdst, v0, imm1, v2

Description For each vector element n, performs vdst[n] = v0[n] * v1[n] + v2[n]. Operand 1 may also be a scalar register or immediate.

A.5.5 imin

Signed integer min

Syntax imin vdst, v0, s1
imin vdst, v0, v1
imin vdst, v0, imm1

Description For each vector element n, performs vdst[n] = min(v0[n], v1[n]). Operand 1 may also be a scalar register or immediate.

A.5.6 imax

Signed integer max

Syntax imax vdst, v0, s1
imax vdst, v0, v1
imax vdst, v0, imm1

Description For each vector element n, performs vdst[n] = max(v0[n], v1[n]). Operand 1 may also be a scalar register or immediate.

A.5.7 shl

Left shift.

Syntax shl vdst, v0, s1
shl vdst, v0, imm1

Description Shift each value v0[n] left by s1/imm1 bits, store the result in vdst.

A.5.8 shr

Right shift.

Syntax shr vdst, v0, s1
shr vdst, v0, imm1

Description Shift each value v0[n] right by s1/imm1 bits, store the result in vdst.

A.5.9 and

Boolean AND

Syntax and vdst, v0, s1
and vdst, v0, v1
and vdst, v0, imm1

Description For each vector element n, performs vdst[n] = v0[n] & v1[n]. Operand 1 may also be a scalar register or immediate.

A.5.10 or

Boolean OR

Syntax or vdst, v0, s1
or vdst, v0, v1
or vdst, v0, imm1

Description For each vector element n, performs vdst[n] = v0[n] | v1[n]. Operand 1 may also be a scalar register or immediate.

A.5.11 xor

Boolean XOR

Syntax xor vdst, v0, s1
xor vdst, v0, v1
xor vdst, v0, imm1

Description For each vector element n, performs vdst[n] = v0[n] $\oplus$ v1[n]. Operand 1 may also be a scalar register or immediate.

A.5.12 not

Boolean NOT

Syntax not vdst, v0

Description For each vector element n, performs vdst[n] = $\sim$v0[n].

A.5.13 siadd

Scalar integer addition.

Syntax siadd sdst, s0, s1
siadd sdst, s0, imm1

Description Add the value of the two scalar integer operands, store in sdst.

A.5.14 sisub

Scalar integer subtraction.

Syntax sisub sdst, s0, s1
sisub sdst, s0, imm1

Description Subtract the value of the two scalar integer operands, store in sdst.

A.5.15 simul

Scalar integer multiplication.

Syntax simul sdst, s0, s1
simul sdst, s0, imm1

Description Multiply the value of the two scalar integer operands, store in sdst.

A.5.16 simad

Scalar integer multiply-addition.

Syntax simad sdst, s0, s1, s2
simad sdst, s0, imm1, s2

Description Multiply the value of the two integer scalar operands, add the third, store in sdst.

A.5.17 simin

Scalar signed integer min

Syntax simin sdst, s0, s1
simin sdst, s0, imm1

Description Performs sdst = min(s0, s1). Operand 1 may also be an immediate.

A.5.18 simax

Scalar signed integer max

Syntax simax sdst, s0, s1
simax sdst, s0, imm1

Description Performs sdst = max(s0, s1). Operand 1 may also be an immediate.

A.5.19 sineg

Scalar signed integer negate

Syntax sineg sdst, s0

Description Performs sdst = -s0.

A.5.20 sibfind

Find first non-sign bit in a scalar integer register.

Syntax sibfind sdst, s0

Description Return the index of the most significant non-sign bit in s0, or ∼0 if no bit is found. Resembles a round-down log2(s0) on any positive integer s0.

A.5.21 sshl

Scalar left shift.

Syntax sshl sdst, s0, s1
sshl sdst, s0, imm1

Description Shift the value of s0 left by s1/imm1 bits, store the result in sdst.

A.5.22 sshr

Scalar right shift.

Syntax sshr sdst, s0, s1
sshr sdst, s0, imm1

Description Shift the value of s0 right by s1/imm1 bits, store the result in sdst.

A.5.23 sidiv

Scalar integer division.

Syntax sidiv sdst, s0, s1
sidiv sdst, s0, imm1

Description Divide integer s0 by s1 or imm1, store in sdst.

A.5.24 simod

Scalar integer modulo.

Syntax simod sdst, s0, s1
simod sdst, s0, imm1

Description Divide integer s0 by s1 or imm1, store modulo in sdst.

A.5.25 sand

Scalar boolean AND.

Syntax sand sdst, s0, s1
sand sdst, s0, imm1

Description Performs sdst = s0 & s1 resp. sdst = s0 & imm1.

A.5.26 sor

Scalar boolean OR.

Syntax sor sdst, s0, s1
sor sdst, s0, imm1

Description Performs sdst = s0 | s1 resp. sdst = s0 | imm1.

A.5.27 snot

Scalar boolean NOT.

Syntax snot sdst, s0

Description Performs sdst = ∼s0.

A.6 Data copy, conversion and intra-lane shuffle

A.6.1 mov

Move immediate or special register to vdst.

Syntax mov vdst, vsp0
mov vdst, imm0

Description Move an immediatevalue or special purpose vector register into the lanes of vector register vdst.

A.6.2 movvsp

Move immediate or vector register to vsp.

Syntax movvsp vsp, v0
movvsp vsp, imm0

Description Move an immediate or vector register into every lane of a special purpose vector register in vsp. Used primarily for cam-based indexed load/store.

A.6.3 smov

Load scalar special register into an SGPR.

Syntax smov sdst, ssp0
smov sdst, imm0

Description Load scalar special register into an SGPR.

A.6.4 smovssp

Move immediate or scalar register to ssp.

Syntax smovssp ssp, s0
smovssp ssp, imm0

Description Move an immediate or scalar register value into a special purpose scalar register ssp. Used primarily for setting custom stride descriptor parameters.

A.6.5 cvt

Convert vector between floating point and integer formats

Syntax cvt.op vdst, v0
cvt.op vdst, vsp0
cvt.op vdst, ssp0

op ∈ {i2f,f2i}

Description Moves a vector- or special purpose register into vector register vdst, converting between float and integer.

.op	Description
i2f	Integer to Float.
f2i	Float to Integer.

A.6.6 scvt

Convert scalar between floating point and integer formats

Syntax scvt.op sdst, s0
scvt.op sdst, ssp0

op ∈ {i2f,f2i}

Description Moves a (special purpose) scalar register into scalar register sdst, converting between float and integer.

.op	Description
i2f	Integer to Float.
f2i	Float to Integer.

A.6.7 bufquery

Query global buffer properties.

Syntax bufquery.op sdst, imm0

op ∈ {dim_x,dim_y}

Description Queries the property of a mapped buffer defined in .op.

.op	Description
dim_x	Buffer width, in number of elements (32-bit words).
dim_y	Buffer height.

A.7 Load/Store

A.7.1 ldglin

Load from global buffer linear to thread configuration.

Syntax ldglin[.op] vdst, imm0[, s1[, s2]]
ldglin[.op] vdst, imm0[, s1[, imm2]]
ldglin[.op] vdst, imm0[, imm1[, s2]]
ldglin[.op] vdst, imm0[, imm1[, imm2]]
ldglin[.op] vsp, imm0[, s1[, s2]]
ldglin[.op] vsp, imm0[, s1[, imm2]]
ldglin[.op] vsp, imm0[, imm1[, s2]]
ldglin[.op] vsp, imm0[, imm1[, imm2]]

op $\in$ {,vec2,vec4}

Description This operation will load one word for each thread from the buffer specified in imm0, the offset for which is primarily determined by the thread configuration. Optionally offset by the x and y coordinates provided in imm1 and imm2. A destination of vc.mem_data will trigger an "indexed" load, where the indexes are taken from vc.mem_idx.

.op	Description
(omit)	Unit mapped elements.
vec2	Vec2 elements to consecutive registers.
vec4	Vec4 elements to consecutive registers.

A.7.2 stglin

Store global linear

Syntax stglin[.op] vdst, imm0[, s1[, s2]]
stglin[.op] vdst, imm0[, s1[, imm2]]
stglin[.op] vdst, imm0[, imm1[, s2]]
stglin[.op] vdst, imm0[, imm1[, imm2]]
stglin[.op] vsp, imm0[, s1[, s2]]
stglin[.op] vsp, imm0[, s1[, imm2]]
stglin[.op] vsp, imm0[, imm1[, s2]]
stglin[.op] vsp, imm0[, imm1[, imm2]]

op ∈ {,vec2,vec4}

Description This operation will store one word for each thread to the global (DRAM) buffer specified in imm0, the offset for which is primarily determined by the thread configuration. Optionally offset by the x and y coordinates provided in imm1 and imm2. A destination of vc.mem_data will trigger an "indexed" store, where the indexes are taken from vc.mem_idx.

.op	Description
(omit)	Unit mapped elements.
vec2	Vec2 elements to consecutive registers.
vec4	Vec4 elements to consecutive registers.

A.7.3 ldgbidx

LOad whole Buffer to CAM-based InDeX registers.

Syntax ldgbidx imm0

Description This operation launches an indexed load, streaming the entire buffer through the CAMs shared bus.

A.7.4 stgbidx

STore whole Buffer to CAM-based index registers.

Syntax stgbidx imm0

Description This operation launches an indexed store, streaming the entire buffer through the CAMs shared bus.

A.7.5 ldgcidx

LOad Custom stride descriptor to CAM-based InDeX registers.

Syntax ldgcidx imm0[, s1[, s2]]
ldgcidx imm0[, s1[, imm2]]
ldgcidx imm0[, imm1[, s2]]
ldgcidx imm0[, imm1[, imm2]]

Description This operation launches an indexed load with a custom stride descriptor for which words, periods and period_count are taken from the special-purpose scalar registers. s1/imm1 ands2/imm2 respectively describe the x- and y-offsets into the buffer.

A.7.6 stgcidx

Store Custom Stride Descriptor to CAM-based index registers.

Syntax stgcidx imm0[, s1[, s2]]
stgcidx imm0[, s1[, imm2]]
stgcidx imm0[, imm1[, s2]]
stgcidx imm0[, imm1[, imm2]]

Description This operation launches an indexed store with a custom stride descriptor for which words, periods and period_count are taken from the special-purpose scalar registers. s1/imm1 ands2/imm2 respectively describe the x- and y-offsets into the buffer.

A.7.7 ldgidxit

LOad from DRAM to CAMs, iterating over indexes.

Syntax ldgidxit vdst, imm0

Description This operation launches an indexed load, iterating over indexes one by one.

A.7.8 stgidxit

Store Custom Stride Descriptor to CAM-based index registers.

Syntax stgidxit vdst, imm0

Description This operation launches an indexed store, iterating over indexes one by one.

A.7.9 ldg2sptile

Load tile from DRAM buffer imm0 to scratchpad buffer dimm.

Syntax ldg2sptile dimm, imm0[, s1[, s2]]
ldg2sptile dimm, imm0[, s1[, imm2]]
ldg2sptile dimm, imm0[, imm1[, s2]]
ldg2sptile dimm, imm0[, imm1[, imm2]]

Description This operation will load a tile of data from a DRAM buffer imm0 to scratchpad buffer dimm. Size is determined by the scratchpad buffer size.

A.7.10 stg2sptile

Store tile to DRAM buffer imm0 from scratchpad buffer dimm.

Syntax stg2sptile dimm, imm0[, s1[, s2]]
stg2sptile dimm, imm0[, s1[, imm2]]
stg2sptile dimm, imm0[, imm1[, s2]]
stg2sptile dimm, imm0[, imm1[, imm2]]

Description This operation will store a tile of data from scratchpad buffer dimm to DRAM buffer imm0. Size is determined by the scratchpad buffer size.

A.7.11 ldsplin

Load from scratchpad buffer linear to thread configuration.

Syntax ldsplin vdst, imm0[, s1[, s2]]
ldsplin vdst, imm0[, s1[, imm2]]
ldsplin vdst, imm0[, imm1[, s2]]
ldsplin vdst, imm0[, imm1[, imm2]]
ldsplin vsp, imm0[, s1[, s2]]
ldsplin vsp, imm0[, s1[, imm2]]
ldsplin vsp, imm0[, imm1[, s2]]
ldsplin vsp, imm0[, imm1[, imm2]]

Description This operation will load one word for each thread from the scratchpad buffer specified in imm0, the offset for which is primarily determined by the thread configuration. Optionally offset by the x and y coordinates provided in imm1 and imm2. A destination of vc.mem_data will trigger an "indexed" load, where the indexes are taken from vc.mem_idx.

A.7.12 stsplin

Store to scratchpad buffer from linear

Syntax stsplin vdst, imm0[, imm1[, s2]]
stsplin vdst, imm0[, imm1[, imm2]]
stsplin vsp, imm0[, imm1[, s2]]
stsplin vsp, imm0[, imm1[, imm2]]

Description This operation will store one word for each thread to the scratchpad buffer specified in imm0, the offset for which is primarily determined by the thread configuration. Optionally offset by the x and y coordinates provided in imm1 and imm2. A destination of vc.mem_data will trigger an "indexed" store, where the indexes are taken from vc.mem_idx.

A.7.13 ldspbidx

LOad whole ScratchPad Buffer to CAM-based InDeX registers.

Syntax ldspbidx imm0

Description This operation launches an indexed load, streaming the entire buffer specified by imm0 through the CAMs shared bus.

A.7.14 stspbidx

STore whole ScratchPad Buffer to CAM-based index registers.

Syntax stspbidx imm0

Description This operation launches an indexed store, streaming the entire buffer specified by imm0 through the CAMs shared bus.

A.7.15 sldg

Scalar load

Syntax sldg sdst, imm0[, imm1]

Description Load one or more words from DRAM buffer imm0 to sdst and subsequent scalar registers. imm1 specifies the number of words to be loaded, defaults to 1.

A.7.16 sldsp

Load scalar from scratchpad

Syntax sldsp sdst, imm0[, s1[, s2]]
sldsp sdst, imm0[, s1[, imm2]]
sldsp sdst, imm0[, imm1[, s2]]
sldsp sdst, imm0[, imm1[, imm2]]

Description Load one or more words from scratchpad buffer imm0 into sdst and subsequent scalar registers. imm1/s1 determines the x-offset, imm2/s2 the y-offset. The number of words loaded is controlled by sc.sd_words.

A.8 Control flow

A.8.1 j

Jump to an absolute location in the program.

Syntax j imm0

Description Update PC with the value given by imm0.

A.8.2 sicj

Scalar Integer Conditional Jump to an absolute location.

Syntax sicj.op imm0, s1

op $\in$ {ez,nz,g,ge,l,le}

Description If the integer in s1 passes the test specified by the suboperation, update PC with the value given by imm0.

.op	Description
ez	Equal to Zero.
nz	Non-equal to Zero.
g	Greater than zero.
ge	Greater than or Equal to zero.
l	Less than zero.
le	Less than or equal to zero.

A.8.3 bra

Conditional (divergent) branch,

Syntax bra imm0, p1

Description Perform a branch conditional on p1 to a destination PC given in imm0.

A.8.4 call

Call

Syntax call imm0[, p1]

Description Call a function at the PC given by imm0. Conditional on p1. Will push a call type entry onto the control stack for return purposes.

A.8.5 cpush

Push an element onto the control stack.

Syntax cpush.op imm0[, p1]

op $\in$ {if,brk,jc}

Description Store a control flow entry onto the control stack. imm0 specifies the PC to push. p1 defines an optional predicate register to push. If p1 is omitted, the CMASK corresponding to the given suboperation will be loaded.

.op	Description
if	Control mask.
brk	Break mask.
jc	Call/return mask.

A.8.6 cmask

Manipulate the “control” CMASK directly

Syntax cmask p0

Description Disable all threads t for which p0[t] is set to 1. Used in part to implement C and C++'s “continue” statement to skip to the next iteration of a for-loop.

A.8.7 cpop

Pop an element off the control stack.

Syntax cpop

Description Pops an entry off the control stack, which is equivalent to either ending the innermost control flow action (such as brk or call) or, in the case of bra, to continue execution of the else branch.

A.8.8 ret

Conditional return.

Syntax ret p0

Description Return from call conditional on predicate register p0. For unconditional return, use CPOP.

A.8.9 brk

Conditional break.

Syntax brk p0

Description Disable all threads t for which p0[t] is set to 1. Used in part to implement C and C++'s "break" statement to break out of a for-loop. For an unconditional break, use CPOP.

A.8.10 exit

Exit program,

Syntax exit [p0]

Description Exits program. Can optionally be conditional on predicate register p0.

A.9 Predicate manipulation

A.9.1 test

Test floating point number against given condition.

Syntax test.op pdst, v0

op $\in$ {ez,nz,g,ge,l,le}

Description Tests each element in vector v0 against the condition provided in .op, produce 1 in the corresponding predicate register bit if the condition holds, 0 otherwise.

.op	Description
ez	Equal to Zero (0.f or -0.f).
nz	Non-equal to Zero.
g	Greater than zero.
ge	Greater than or Equal to zero.
l	Less than zero.
le	Less than or equal to zero.

A.9.2 itest

Test integer number against given condition.

Syntax itest.op pdst, v0

op $\in$ {ez,nz,g,ge,l,le}

Description Tests each element in vector v0 against the condition provided in .op, produce 1 in the corresponding predicate register bit if the condition holds, 0 otherwise.

.op	Description
ez	Equal to Zero.
nz	Non-equal to Zero.
g	Greater than zero.
ge	Greater than or Equal to zero.
l	Less than zero.
le	Less than or equal to zero.

A.9.3 pbool

Perform a boolean operation on two predicate registers.

Syntax pbool.op pdst, p0, p1

op $\in$ {and,or,nand,nor}

Description For each element n in the (vector) predicate register, perform pdst[n] = p0[n] (op) p1[n].

.op	Description
and	Boolean AND.
or	Boolean OR.
nand	Boolean Not-AND
nor	Boolean Not-OR

A.10 Debug

A.10.1 printsgpr

Print the value of a scalar register

Syntax printsgpr s0

A.10.2 printvgpr

Print the value of a vector register lane

Syntax printvgpr v0, imm1

Description imm1 specifies the lane number to print.

A.10.3 printpr

Print the values of a predicate register

Syntax printpr p0

A.10.4 printcmask

Print the value of a CMASK.

Syntax printcmask.op

op ∈ {if,brk,jc,exit}

.op	Description
if	Control mask.
brk	Break mask.
jc	Call/return mask.
exit	Exit mask.

A.10.5 printtrace

Enable/disable trace printing in the simulator.

Syntax printtrace imm0