Mixed Precision Vector Processors

Albert Ou
Krste Asanović, Ed.
Vladimir Stojanovic, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-265
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-265.html

December 19, 2015

Acknowledgement

Research is partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architectures Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

Mixed-Precision Vector Processors

by

Albert Ji-Hung Ou

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Krste Asanović, Chair
Associate Professor Vladimir Stojanović

Fall 2015

Mixed-Precision Vector Processors

Contents

Contents **ii**

List of Figures **iv**

List of Tables **v**

1 Introduction **2**
1.1 Justification for Mixed Precision 3
1.2 Justification for Vector Processors 4
1.3 Thesis Organization 5
1.4 Project History 6

2 Related Work **8**

3 Comparison of ISA Approaches **10**
3.1 Subword SIMD 10
3.2 Vectors 13
3.3 Reduced Precision 14
3.4 Mixed Precision 15

4 Baseline Vector Architecture **17**
4.1 Programming Model 17
4.2 System Architecture 19
4.3 Microarchitecture 20

5 Mixed-Precision Vector Architecture **29**
5.1 Programming Model 29
5.2 Microarchitecture 34

6 Evaluation Framework **39**
6.1 Microbenchmarks 40
6.2 OpenCL Compiler 41
6.3 Samsung Exynos 5422 and the ARM Mali-T628 MP6 GPU 42

6.4 RTL Development and VLSI Flow 44

7 Preliminary Evaluation Results **46**
7.1 Memory System Validation 46
7.2 Area and Cycle Time 48
7.3 Performance Comparison 51
7.4 Energy Comparison 54

8 Conclusion **56**

Bibliography **57**

List of Figures

3.1 Double-precision SIMD addition 11
3.2 Evolution of SIMD extensions in the x86 ISA 12
3.3 Double-precision vector addition 13
3.4 Single-precision addition 14
3.5 Widening FMA 15

4.1 Conditional SAXPY kernels written in C 18
4.2 Conditional SAXPY kernels mapped to pseudo-assembly 18
4.3 System architecture provided by the Rocket Chip SoC generator 19
4.4 Block diagram of Hwacha 21
4.5 Block diagram of the Vector Execution Unit 23
4.6 Systolic bank execution 25
4.7 Block diagram of the Vector Memory Unit 26
4.8 Mapping of elements across a four-lane machine 28
4.9 Example of redundant memory requests by adjacent lanes 28

5.1 Simplified logical view of an 8-entry VRF 30
5.2 Conditional SAXPY kernels updated for HOV 30
5.3 Actual striped mapping to a 2-bank physical VRF 35
5.4 Hypothetical naive mapping to a 2-bank physical VRF 35

6.1 Hwacha evaluation framework 40
6.2 Samsung Exynos 5422 block diagram 43

7.1 ccbench "caches" memory system benchmark 47
7.2 Layout of the single-lane Hwacha design with HOV 49
7.3 Area distribution 50
7.4 Preliminary performance results 53
7.5 Preliminary energy results 55

List of Tables

6.1 Listing of all microbenchmarks 41
6.2 Used Rocket Chip SoC generator parameters 44

7.1 VLSI quality of results 48

Abstract

Mixed-Precision Vector Processors

by

Albert Ji-Hung Ou

Master of Science in Computer Science

University of California, Berkeley

Professor Krste Asanović, Chair

Mixed-precision computation presents opportunities for programmable accelerators to improve performance and energy efficiency while retaining application flexibility. Building on the Hwacha decoupled vector-fetch accelerator, we introduce *high-occupancy vector lanes* (HOV), a set of mixed-precision hardware optimizations which support dynamic configuration of multiple architectural register widths and high-throughput operations on packed data. We discuss the implications of HOV for the programming model and describe our microarchitectural approach to maximizing register file utilization and datapath parallelism. Using complete VLSI implementations of HOV in a commercial 28 nm process technology, featuring a cache-coherent memory hierarchy with L2 caches and simulated LPDDR3 DRAM modules, we quantify the impact of our HOV enhancements on area, performance, and energy consumption compared to the baseline design, a decoupled vector architecture without mixed-precision support. We observe as much as a 64.3% performance gain and a 61.6% energy reduction over the baseline vector machine on half-precision dense matrix multiplication. We then validate the HOV design against the ARM Mali-T628 MP6 GPU by running a suite of microbenchmarks compiled from the same OpenCL source code using our custom HOV-enabled compiler and the ARM stock compiler.

Chapter 1

Introduction

The prominent breakdown of Dennard scaling, amid ongoing difficulties with interconnect scaling, has driven VLSI design into a new regime in which severe power constraints and communication costs prevail. Technology trends have culminated in an evolutionary plateau for conventional superscalar processors, as overly aggressive pipelining, speculation, and out-of-order execution become counterproductive beyond a certain degree of complexity. These challenges have thus prompted a retreat away from maximal sequential performance, a pursuit which heretofore has characterized mainstream computer architecture since its infancy, and has initiated a transition towards *throughput-oriented computing* that instead emphasizes sustained performance per unit energy. These throughput-oriented architectures derive performance from parallelism and efficiency from locality [1], [2].

Fully harnessing the abundant parallelism and locality in emerging applications generally entails some form of hardware specialization, a familiar concept made progressively feasible by the vast transistor budgets afforded by modern CMOS processes. How to balance efficiency with flexibility—and ideally elevate both—is the eternal research question. Mixed-precision optimizations may serve as one answer.

Compared to fixed-function hardware, programmable processors contend with inefficiencies from at least two major sources: overhead of instruction delivery and wasteful power consumption by over-provisioned datapaths. In this work, we propose a dual strategy of seamless mixed-precision computation on a vector architecture to address both aspects.

1.1 Justification for Mixed Precision

Relying on prior knowledge within a restricted scope of applications, fixed-function accelerators notably exploit heterogeneous and minimal word sizes in their custom datapaths. Fundamentally, less energy dissipation and higher performance both ensue with fewer bits communicated and computed. In contrast, commodity processors must support a generic assortment of conventional number representations to fulfill a general-purpose role. Thus, the datapath width is often conservatively fixed to the maximum precision and widest dynamic range imposed by any plausible application.

Recently, growing attention is being paid to the impact of limited numerical precision across a variety of application domains relevant to high-throughput processing. It is becoming increasingly common practice in high-performance computing to forgo double-precision floating-point in favor of single precision, owing to the two-fold disparity in peak FLOPS that GPGPUs and SIMD processors customarily manifest. There is cause to be optimistic about the general-purpose viability of reduced-precision computation. In the context of deep learning, for example, the presence of statistical approximation and estimation errors renders high precision largely unnecessary [3]. Moreover, additional noise during training can in fact enhance the performance of a neural network. Workloads that exhibit a natural error resilience at the algorithmic level therefore seem to offer a compelling basis for hardware optimizations capitalizing on precision reduction, while the diversity of applications clearly motivates a programmable solution.

One can imagine a more versatile accelerator capable of programmatically switching its datapath from one reduced-precision mode to another at a time. However, rarely is one global precision optimal or even sufficient throughout all phases of computation. In areas that historically have been the exclusive realm of double-precision floating-point arithmetic, such as scientific computing, there is potential for reformulating existing problems by means of *mixed-precision methods* [4]. These schemes combine the use of different precisions to minimize cost, completing the bulk of the work in lower precision, while also preserving overall accuracy. Iterative refinement techniques, widely applied to linear systems, improve the quality of an approximate solution via gradual updates calculated in extra precision. Similarly, instability in an algorithm may be overcome by selectively resorting to higher

precision during more sensitive steps.

The design space of mixed-precision hardware extends from the simple coexistence of variable-width information, such as integer data versus memory addresses, to widening arithmetic operations, such as a *fused multiply-add* (FMA) that sums the product of two n-bit values and a $2n$-bit value without intermediate rounding.

1.2 Justification for Vector Processors

Vector architectures are an exceptional match for the data-level parallelism (DLP) prevalent in throughput computing applications. Compared to instruction-level parallelism (ILP) and thread-level parallelism (TLP), DLP is the most rigid form of machine parallelism but is therefore the most efficient with respect to control overhead.

Each vector instruction concisely expresses an entire set of homogeneous and independent elementary operations, thereby amortizing the cost of instruction fetch, decode, and scheduling. Fewer instructions that encapsulate more work also relaxes bandwidth demands on instruction supply. Whereas managing all possible dependencies among the same number of scalar operations incurs quadratic complexity, the explicit data parallelism reinforces constraints which dramatically simplify the control logic.

From an implementation perspective, the inherent regularity of vector operations promotes extensive modularity and scalability to meet varying performance/power targets. A massively parallel machine may be constructed from a multiplicity of identical and largely self-contained *lanes*, each housing a portion of the vector register file and functional units but together sharing a common scalar and control unit.

Vector data accesses adhere to highly structured and predictable communication patterns. This permits microarchitectural optimizations such as register file banking to reduce port count. Long vectors expose substantial memory-level parallelism to more easily saturate memory bandwidth. In the common case, constant-stride memory operations are particularly conducive to access/execute decoupling and prefetching techniques for hiding latency, since the address stream can be generated separately from other computations.

Vector architectures are readily adaptable to reduced-precision computation. Due to the

intrinsic data independence, a wide datapath can be naturally partitioned to operate on several narrower elements concurrently. For compute-bound workloads, this *subword parallelism* offers an area-efficient mechanism for multiplying arithmetic throughput by an integer factor. The marginal implementation cost principally involves the replication of functional units—crucially, the existing register file ports and interconnection fabric for operand traffic remain unaltered yet enjoy improved utilization.

Reduced precision alleviates memory-bound workloads as well. In general, it is fairly straightforward to increase functional unit bandwidth by instantiating extra ALUs, for their individual area penalties are exceedingly modest as a result of advances in semiconductor device technology, but system-wide memory bandwidth cannot be increased with the same ease. With global communication as a limiting factor, a more compact representation of data realizes more effective use of available bandwidth.

Precision truncation can be regarded as a lossy compression scheme that makes larger problem sizes more economical across all levels of the memory hierarchy. Denser storage of elements directly translates into longer vectors with the same register file capacity. This enables greater register blocking, aiding in-register operand reuse and correspondingly lessening memory pressure. The expanded buffering provided by the vector register file further assists with decoupled execution to better tolerate memory latency. Lastly, a greater concentration of useful data is resident in a cache—if the working set now manages to fit within the L1 or L2 caches where it formerly could not, then a superlinear speedup becomes attainable.

1.3 Thesis Organization

Chapter 2 reviews existing architectural techniques for mixed-precision computation and some of their disadvantages. Chapter 3 provides a deeper analysis of subword SIMD and vector architectures, discussing why vector processing serves as a more refined foundation for a mixed-precision programming model. Chapter 4 describes the baseline vector accelerator design, and Chapter 5 elaborates on *high-occupancy vectors* (HOV), the architectural and microarchitectural augmentations to support efficient mixed-precision vector computation. Chapter 6 details the evaluation framework, including RTL development methodology with

Chisel and the physical design flow in a commercial 28 nm process technology. Chapter 7 compares complete VLSI implementations of the HOV and baseline designs, first to each other and then against an ARM Mali-T628 MP6 GPU, using a suite of compiled OpenCL and manually optimized assembly microbenchmarks.

1.4 Project History

Collaboration

This thesis is the result of a collaborative group project. Other people have made direct contributions to the ideas and results included here. The Hwacha vector-fetch accelerator was developed by Yunsup Lee, myself, Colin Schmidt, Sagar Karandikar, Krste Asanović, and others from 2011 through 2015. As the lead architect of Hwacha, Yunsup Lee directed the development and evaluation of the architecture, microarchitecture, RTL, compiler, verification framework, microbenchmarks, and application kernels. I was primarily responsible for the RTL implementation of the Vector Memory Unit and the mixed-precision extensions described in this thesis. Colin Schmidt took the lead on the definition of the Hwacha ISA, RTL implementation of the scalar unit, C++ functional ISA simulator, vector torture test generator, Hwacha extensions to the GNU toolchain port, and the OpenCL compiler and benchmark suite. Sagar Karandikar took the lead on the `bar-crawl` tool for design-space exploration, VLSI floorplanning, RTL implementation of the Vector Runahead Unit, ARM Mali-T628 MP6 GPU evaluation, and the assembly microbenchmark suite. Palmer Dabbelt took the lead on the physical design flow and post-PAR gate-level simulation in the 28 nm process technology used in this study. Henry Cook took the lead on the RTL implementation of the uncore components, including the L2 cache and the TileLink cache coherence protocol. Howard Mao took the lead on the dual LPDDR3 memory channel support in the testbench and provided critical fixes for the outer memory system. Andrew Waterman took the lead on the definition of the RISC-V ISA, the RISC-V GNU toolchain port, and the RTL implementation of the Rocket core. Andrew also helped to define the Hwacha ISA. John Hauser took the lead on the development of the hardware floating-point units. Many others contributed to the surrounding infrastructure, such as the Rocket Chip SoC generator. Huy

Vo, Stephen Twigg, and Quan Nguyen contributed to earlier versions of Hwacha. Finally, Krste Asanović was integral in all aspects of the project.

Previous Publications

This thesis is a direct descendant of the publication "MVP: A Case for Mixed-Precision Vector Processors" [5], presented at the PRISM-2 workshop at ISCA 2014. That paper itself culminated from two graduate-level course projects: CS250 (Fall 2013) and CS252 (Spring 2014), both times undertaken with co-conspirator Quan Nguyen. Whereas all previous work involved retrofitting two earlier incarnations of Hwacha, retroactively designated *V2* and *V3*, this thesis builds on the *V4* redesign. *V4* represents a complete overhaul of both the ISA and RTL, and unlike its predecessors, it was architected from the very beginning with mixed-precision support as an explicit design objective. The result is an improved mixed-precision implementation with respect to functionality, performance, and ease of integration.

The text has been extensively rewritten to cover additional design rationale, deeper explanation of mechanisms, and other details beyond the scope of the MVP paper. Such material include: in Chapter 3, discussion of vector length control (or lack thereof) and its implications on programming abstractions and, in Chapter 5, sections on portability considerations, polymorphic instructions, and updated microarchitecture.

In addition, this thesis was authored concurrently with a set of UCB technical reports, resulting in some shared content. Chapter 4 appear as expanded passages in the "Hwacha Vector-Fetch Architecture Manual" [6] and the "Hwacha Microarchitecture Manual" [7]. Chapters 6 and 7 are adapted from "Hwacha Preliminary Evaluation Results" [8].

Funding

Research is partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architectures Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

Chapter 2

Related Work

Subdividing a wide datapath to perform multiple reduced-precision operations in parallel is a classic technique employed by early vector supercomputers, such as the CDC STAR-100 [9]. Asanović [10] and Kozyrakis [11] similarly describe how a vector machine may be viewed as an array of "virtual processors" whose widths are set collectively through a virtual processor width (`VPW`) control register. However, the element size of an individual vector cannot be configured independently, complicating the intermixture of different types in the same register file: Either `VPW` must be initialized for the widest datatype in a given block of code, possibly wasting space, or must be appropriately manipulated before each register access, enlarging the instruction footprint.

A related approach is the *subword single-instruction multiple-data* (SIMD) architectural pattern whose earliest instance was perhaps the Lincoln TX-2, which featured separate instructions for treating a full 36-bit machine as four 9-bit parallel machines, among other variations [12]. This style of design has widely endured in multimedia accelerators such as the IBM Cell Broadband Engine [13], and it has witnessed tremendous popular adoption by contemporary general-purpose instruction sets through various forms of packed SIMD extensions, e.g., x86 AVX [14] and ARM NEON [15]. Because vector and subword SIMD architectures are so frequently conflated, chapter 3 aims to elucidate the salient differences between the two—in particular, how subword SIMD's inferior programming abstractions inconvenience a mixed-precision environment.

In what may be considered an inversion of subword packing, an architecture might store

datatypes larger than its nominal bit width by splitting them across multiple adjacent registers. Graphics processing units (GPUs), for example, traditionally possess 32-bit registers; double-precision values occupy aligned register pairs referenced via even-numbered specifiers. This arrangement, while adequate for limited variable-precision support, essentially halves the architectural register set available to an application.

Newer generations of mobile GPUs have acquired native support for half-precision floating-point arithmetic (FP16), despite the IEEE 754-2008 standard intending it only as a storage format. It is well known that graphics workloads do not always require the highest fidelity due to limitations in human perception. The NVIDIA Tegra X1, based on the Maxwell architecture, executes FP16 FMAs at "double-rate" throughput, essentially two-wide SIMD operation from the normal FP32 perspective [16]. This development is by no means unique—the ARM Midgard, Imagination PowerVR Rogue, and AMD Graphics Core Next Generation 3 (GCN3) families all incorporate FP16 in their rendering pipelines, with implementations sharing varying degrees of resources with the FP32 units.

Classic fixed-point digital signal processors (DSPs) are distinguished by special-purpose features to curtail overflow and rounding errors. These include multiplier-accumulator (MAC) units supporting widening arithmetic, dedicated extended-precision accumulator registers, and supplementary guard bits for saturation. The C54x generation of the famous Texas Instruments TMS320 family serves as a good example with its 17×17 multiplier and 40-bit accumulators [17]. However, DSPs characteristically suffer from irregular architectures as a result of ad-hoc evolution, which render them a challenging compiler target and hinder their suitability for a more general-purpose mixed-precision function.

Chapter 3

Comparison of ISA Approaches

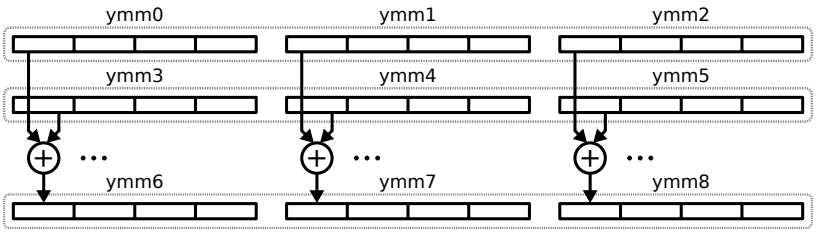
Vectors and subword SIMD are two approaches to instruction set architecture (ISA) design that, to a casual observer, appear outwardly alike: Both leverage data-level parallelism by operating on one-dimensional arrays of quantities and admit microarchitectural optimizations based on subword parallelism. However, a deeper examination reveals significant differences in the ways that mixed-precision computation can be mapped to each architecture. We argue that the advantages of true vector processing revolve around a key innovation in abstraction: flexible hardware vector length control.

3.1 Subword SIMD

The term *subword SIMD*, also known as *"short-vector" extensions* in the context of a scalar ISA, broadly describes the practice of explicitly packing contiguous subwords of equal size within a single *architectural* register. These are acted upon in parallel by special instructions provided per subword data type.

Fixed-width Property

Opcodes designate a fixed "vector" length. In other words, the SIMD width is exposed as part of the ISA. This is the most distinguishing feature of subword SIMD and also its most decisive drawback, as the subsequent discussion will prove.



(a) Visualization

```
vaddpd ymm0, ymm3, ymm6
vaddpd ymm1, ymm4, ymm7
vaddpd ymm2, ymm5, ymm8
```

(b) Pseudo-assembly

Figure 3.1: Double-precision SIMD addition

Figure 3.1a depicts how four 64-bit subwords may be packed into a single 256-bit SIMD register. Each register may alternatively hold 8 32-bit, 16 16-bit, or 32 8-bit values.

Short-vector Property

SIMD widths tend to be relatively short. Modern SIMD extensions originated as an ad-hoc retrofit of scalar microarchitectures for a multimedia focus. The reuse of existing scalar register files presented what was then the path of least resistance towards subword parallelism, and it offers some historical basis for the habit of embedding the SIMD width in the ISA as well as for the short length of SIMD "vectors".

Nowadays, a separate register file is normally dedicated to the SIMD unit, decoupling it from the scalar width. Nevertheless, conventional SIMD units remain tightly integrated with the host scalar pipeline. Implementations rely nearly exclusively on *spatial execution*, forcing use of superscalar issue mechanisms to saturate functional units and hide long latencies. Control complexity and power consumption are both adversely impacted, especially if it involves out-of-order scheduling and register renaming.

Moreover, this imposes a practical upper bound on SIMD width—now typically either 128 or 256 bits, fairly modest in relation to the degree of parallelism commonly found in DLP-intensive workloads, which may comprise as much as hundreds of parallel operations. In combination with the inflexible nature of SIMD registers, utilizing the entire register file capacity often entails splitting a longer *application vector* across multiple architectural registers. Doing so inevitably requires more instructions. Figure 3.1b demonstrates such a situation where vectors of twelve elements must be subdivided among three SIMD registers each, resulting in three separate (though independent) `vaddpd` instructions.

These effects all counteract the efficiency gains from DLP.

Non-Portability

SIMD extensions contribute directly to unrestrained instruction set growth. As technology advances, the desire to scale performance across processor generations, along with commercial pressure, begets a natural tendency towards successively wider SIMD implementations. Each transition to a new width involves introducing a complete set of new instructions. The x86 architecture's own cluttered progression from MMX to AVX-512, summarized in Figure 3.2, perfectly illustrates such a trend.

The proliferation of new instructions leads to rapid depletion of opcode encoding space, despite instructions being largely redundant except to distinguish SIMD width. As an example, four versions of packed double-precision floating-point add operations exist in x86: `addpd` from SSE2 (with opcode `66 0F 58 /r`) and `vaddpd` from AVX (with prefixes `VEX.128`, `VEX.256`, and `EVEX.512`) [18]. AVX instructions have therefore become as long as 11 bytes.

Worse, applications targeting a particular SIMD width are inherently not portable. Legacy applications cannot automatically benefit from the increased SIMD widths. Code must undergo explicit migration: recompilation at the very least to include the new instructions, but more often a tedious rewrite when intrinsics are used instead of auto-vectorization. Conversely, code compiled for wider SIMD registers fail to execute on older machines with narrower ones. Such incompatibilities necessitate dynamic dispatch to different versions of code at runtime.

Figure 3.2: Evolution of SIMD extensions in the x86 ISA

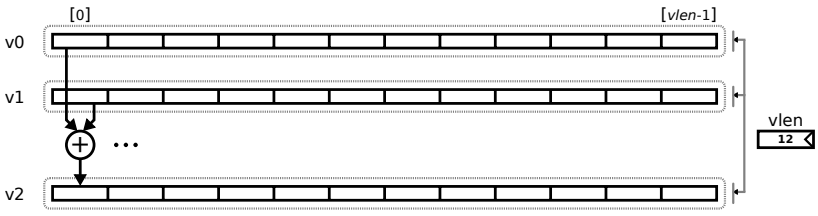
3.2 Vectors

The deficiencies of subword SIMD are not fundamental to data-parallel architectures, rather being artifacts of poor design. In fact, a solution has already existed decades before commodity microprocessors first adopted SIMD extensions, in traditional vector machines descended from the Cray archetype [19].

Vector architectures provide longer, genuine vectors counted in terms of elements instead of bits. The hardware vector length generally ranges from 64 to thousands of elements, an aspect of their supercomputer heritage. Unlike the vast majority of subword SIMD implementations, vector machines incorporate *temporal execution* to process long vectors over multiple cycles, supplemented by *chaining* (the vector equivalent of register bypassing) to overlap dependent operations.

The hardware vector length is dynamically adjustable. A critical architectural feature is the *vector length register* (`VLR`), which indicates the number of elements to be processed per vector instruction, up to the maximum hardware vector length (HVL). Software manipulates the `VLR` by requesting a certain application vector length (AVL) with the `vsetvl` instruction; hardware responds with the lesser of the AVL and HVL. In Figure 3.3, for example, the `VLR` is set to 12. Note that only three architectural vector registers are needed here to represent the operands and the result, unlike the SIMD case, producing code that is expressively minimal.

This method of controlling the vector length enables greater scalability while preserving forward and backward compatibility. The `VLR` informs the program at runtime how to *strip-mine* application vectors of arbitrary length. Essentially, it confers a level of indirection such that software remains oblivious to the HVL: The same code executes correctly and with maximal efficiency on implementations of any HVL. Most importantly, this portability



(a) Visualization

```
vsetvl vlen, 12
vadd v2, v0, v1
```

(b) Pseudo-assembly

Figure 3.3: Double-precision vector addition

is achieved without any additional developer effort or ISA changes. (In practice, the ISA may specify a minimum HVL, e.g. 4, to elide stripmining overheads for short vectors.)

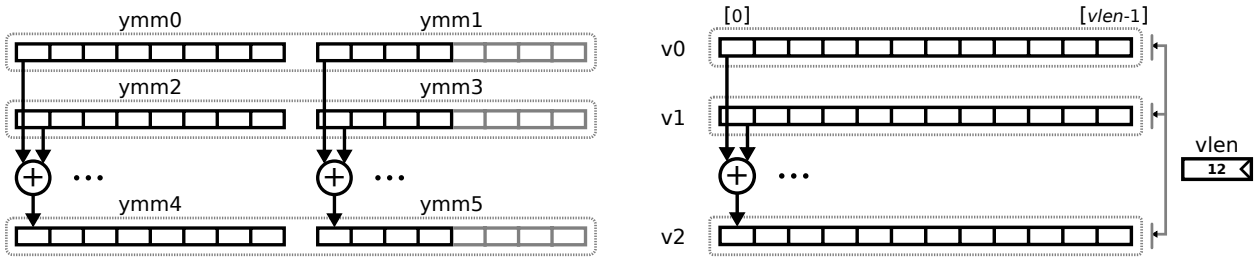
3.3 Reduced Precision

Now we consider how well the vector and subword SIMD models each accommodate smaller data types, while maintaining the invariant of identical input and output precisions. Our analysis builds upon the previous examples by substituting double precision with single.

With a fixed bit-width, SIMD registers contain a variable number of elements depending on the precision. This burdens the software with a potential *fringe case* when the application vector length is not a clean multiple of the SIMD width, resulting in partially utilized registers for the tail of the vector. A reduction in precision raises the likelihood of such an occurrence for a given SIMD width.

Simple cases akin to Figure 3.4a might be tolerated by padding the vectors in memory, but others require the insertion of extra fringe code. A compiler generally has two options: either predicate the trailing SIMD instructions based on the element indices or, in the absence of full support for conditional execution, resort to a scalar loop for the remainder.

Conversely, the reduced-precision vector example shown in Figure 3.4b does not deviate conceptually from its predecessor, Figure 3.3. Note that the `VLR` is agnostic to the precision of the elements and entirely averts all fringe cases for non-2^n problem sizes. The programming model therefore scales more gracefully.



(a) SIMD (b) Vector

Figure 3.4: Single-precision addition

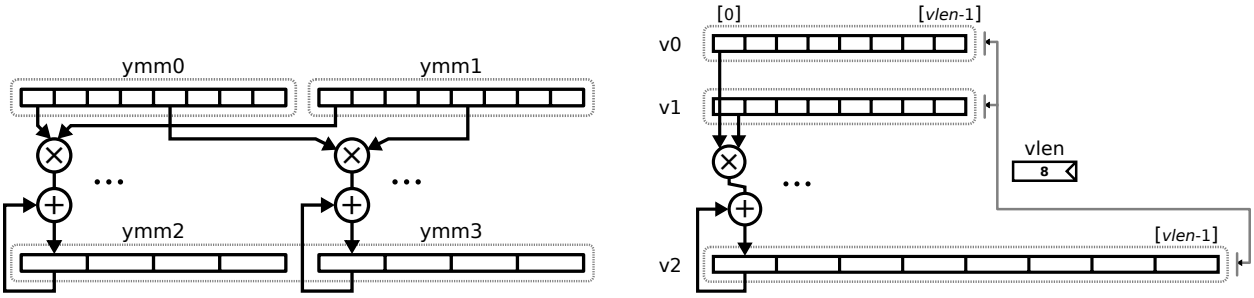
3.4 Mixed Precision

Next, we proceed to the generalized case in which the sources and destinations of an operation may differ in precision. Our analysis concentrates on a hypothetical FMA operation whose output precision is exactly twice that of the input precision, but the same observations also apply if that ratio were reversed.

Figure 3.5a portrays how mixed-precision computation can be quite intricate to convey with subword SIMD. The main difficulty lies in the unequal element counts of the source and destination registers. Consequently, the 8-wide FMA must be decomposed into two separate operations, each reading from either the upper or lower segments of the two sources and then writing to a different destination. Yet SIMD extensions typically provide only instructions that work on the lower end of registers, so extra shuffling to rearrange the operands must precede the upper operation. In the x86 ISA, for example, this might involve a `VPERM2F128` instruction ("permute float-point values") to exchange the upper/lower 128 bits of a 256-bit AVX register [18].

A simpler alternative is to store 4 lower-precision operands in a register pair, rather than the maximum of 8 in one, to match the quantity in the destination registers—basically, to restrict the number of elements per register according to the highest precision in order to equalize their lengths. Although tempting, it is still far from ideal, as the inefficient register usage would limit the extent of software pipelining and register blocking.

Contrast that with the vector model shown in Figure 3.5b, which continues to be as



(a) SIMD (b) Vector

Figure 3.5: Widening FMA. For brevity, the destination register implicitly supplies the addend in this visualization, although in practice it would usually originate from a third source register.

elegant as before. *Hardware vectors always contain the same number of elements regardless of precision*—again, a benefit of the `VLR`. Therefore merely one instruction suffices per vector, no matter the peculiar combination of input/output precisions.

To summarize the fundamental weakness of SIMD, a static encoding of the SIMD width in the ISA exposes what is essentially a microarchitectural detail—namely, subword packing being software-visible. Vector architectures successfully avoid this by providing indirection through a vector length register.

There remains a question, however, about how exactly the vector register file could leverage subword packing for improved storage density if that aspect is completely invisible to software. Chapter 5 describes how it is possible, by extending the vector ISA with optional configuration hints about register usage, for a microarchitecture to automatically recoup and near-optimally redistribute the space among mixed-precision vectors.

Chapter 4

Baseline Vector Architecture

Hwacha is a decoupled vector-fetch data-parallel accelerator which serves as the vehicle for our architectural exploration. Its microarchitecture and programming paradigm combines facets of predecessor vector-threaded architectures, such as Maven [20], [21], with concepts from traditional vector machines—in particular, divergence handling mechanisms in software.

4.1 Programming Model

The Hwacha accelerator supports a novel programming model, *vector fetch*. It is best explained by contrast with the traditional vector programming model. As an example, we use a conditionalized SAXPY kernel, CSAXPY. Figure 4.1 shows CSAXPY expressed in C as both a vectorizable loop and as a *Single-Program Multiple-Data* (SPMD) kernel. CSAXPY takes as input an array of conditions, a scalar `a`, and vectors `x` and `y`; it computes `y += ax` for the elements for which the condition is true.

Hwacha builds on traditional vector architectures, with a key difference: The vector operations have been hoisted out of the stripmine loop and placed in their own *vector fetch block*. This allows the scalar control processor to send only a program counter to the vector processor. The control processor is then able to complete the current stripmine iteration faster and continue doing useful work, while the vector processor is independently executing the vector instructions.

Like a traditional vector machine, Hwacha has vector data registers (`vv`0–255) and vector

```
1  void csaxpy(size_t n, bool cond[],
2     float a, float x[], float y[])
3  {
4     for (size_t i = 0; i < n; ++i)
5       if (cond[i])
6         y[i] = a*x[i] + y[i];
7  }
```

(a) Vectorizable loop

```
1  void csaxpy_spmd(size_t n, bool cond[],
2    float a, float x[], float y[])
3  {
4    if (tid < n)
5      if (cond[tid])
6        y[tid] = a*x[tid]+y[tid];
7  }
```

(b) SPMD

Figure 4.1: Conditional SAXPY kernels written in C. The SPMD kernel launch code for (b) is omitted for brevity.

```
1  csaxpy_tvec:
2  stripmine:
3       vsetvl  t0, a0
4       vlb     vv0, (a1)
5       vcmpez  vp0, vv0
6  !vp0 vlw     vv0, (a3)
7  !vp0 vlw     vv1, (a4)
8  !vp0 vfma    vv0, vv0, a2, vv1
9  !vp0 vsw     vv0, (a4)
10      add     a1, a1, t0
11      slli    t1, t0, 2
12      add     a3, a3, t1
13      add     a4, a4, t1
14      sub     a0, a0, t0
15      bnez    a0, stripmine
16      ret
```

(a) Traditional vector

```
1  csaxpy_control_thread:
2    vsetcfg 2, 1
3    vmcs vs1, a2
4  stripmine:
5    vsetvl  t0, a0
6    vmca    va0, a1
7    vmca    va1, a3
8    vmca    va2, a4
9    vf      csaxpy_worker_thread
10   add     a1, a1, t0
11   slli    t1, t0, 2
12   add     a3, a3, t1
13   add     a4, a4, t1
14   sub     a0, a0, t0
15   bnez    a0, stripmine
16   ret
17
18 csaxpy_worker_thread:
19      vlb     vv0, (va0)
20      vcmpez  vp0, vv0
21 !vp0 vlw     vv0, (va1)
22 !vp0 vlw     vv1, (va2)
23 !vp0 vfma    vv0, vv0, vs1, vv1
24 !vp0 vsw     vv0, (va2)
25      vstop
```

(b) Hwacha

Figure 4.2: Conditional SAXPY kernels mapped to pseudo-assembly. `a0` holds variable `n`, `a1` holds pointer `cond`, `a2` holds scalar `a`, `a3` holds pointer `x`, and `a4` holds pointer `y`.

predicate registers (`vp`0–15), but it also has two flavors of scalar registers. These are the *shared* registers (`vs`0–63, with `vs`0 hardwired to constant 0), which can be read and written within a vector fetch block, and *address* registers (`va`0–31), which are read-only within a vector fetch block. This distinction supports non-speculative access/execute decoupling, described in Section 4.3.

Figure 4.2b shows the CSAXPY code for Hwacha. The *control thread* (lines 1–16) first

executes the `vsetcfg` instruction (line 2), which adjusts the maximum hardware vector length taking the register usage into account. `vmcs` (line 3) moves the value of a scalar register from the control thread to a `vs` register. The stripmine loop sets the vector length with a `vsetvl` instruction (line 5), moves the array pointers to the vector unit with `vmca` instructions (line 6–8), and then executes a vector-fetch (`vf`) instruction (line 9), which causes the *worker thread* to begin executing the vector fetch block (lines 18–25). The code in the vector fetch block is equivalent to the vector code in Figure 4.2a, with the addition of a `vstop` instruction to signify the end of the block.

4.2 System Architecture

Figure 4.3 illustrates the overall system architecture surrounding Hwacha. The open-source Rocket Chip SoC generator is used to elaborate the design [22]. The generator consists of highly parameterized RTL libraries written in Chisel [23].

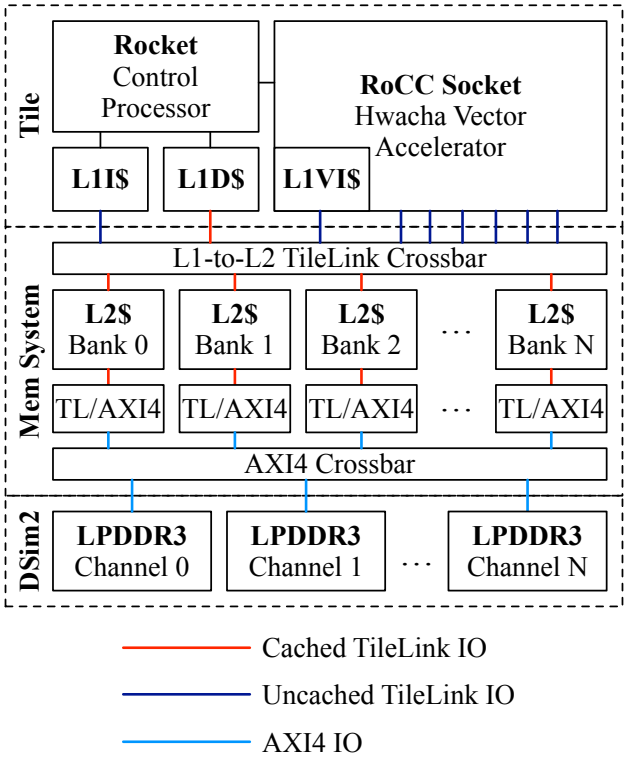


Figure 4.3: System architecture provided by the Rocket Chip SoC generator

A tile consists of a Rocket control processor and a Rocket Custom Coprocessor (RoCC) socket. Rocket is a five-stage in-order RISC-V scalar core attached to a private blocking L1 instruction cache and non-blocking L1 data cache. The RoCC socket provides a standardized interface for issuing commands to a custom accelerator, as well as an interface to the memory system. The Hwacha decoupled vector accelerator, along with its blocking vector instruction cache, is designed to fit within the RoCC socket. The control thread and the worker thread of the vector-fetch programming model are mapped to Rocket and Hwacha, respectively.

The shared L2 cache is banked, set-associative, and fully inclusive of the L1. Addresses are interleaved at cache line granularity across banks. The tile and L2 cache banks are connected via an on-chip network that implements the TileLink cache coherence protocol [24].

The refill ports of the L2 cache banks are connected to a bank of cached TileLink IO to AXI4 converters. The AXI4 interfaces are then routed to the appropriate LPDDR3 memory channels through the AXI4 crossbars. The LPDDR3 channels are implemented in the testbench, which simulates the DRAM timing using DRAMSim2 [25].

The memory system parameters such as the cache size, associativity, number of L2 cache banks and memory channels, and cache-coherence protocol are set with a configuration object during elaboration. This configuration object also holds design parameters that are chosen for the Rocket control processor and the Hwacha vector accelerator.

4.3 Microarchitecture

The Hwacha vector accelerator combines ideas from access/execute decoupling [26], decoupled vector architectures [27], and cache refill/access decoupling [28], applying them to work within a cache-coherent memory system without risk of deadlock. Extensive decoupling enables the microarchitecture to effectively tolerate long and variable memory latencies with an in-order design.

Figure 4.4 presents the high-level anatomy of the vector accelerator. Hwacha is situated as a discrete coprocessor with its own independent frontend. This vector-fetch decoupling relieves the control processor so that it can resolve address calculations for upcoming vector fetch blocks, among other bookkeeping actions, well in advance of the accelerator.

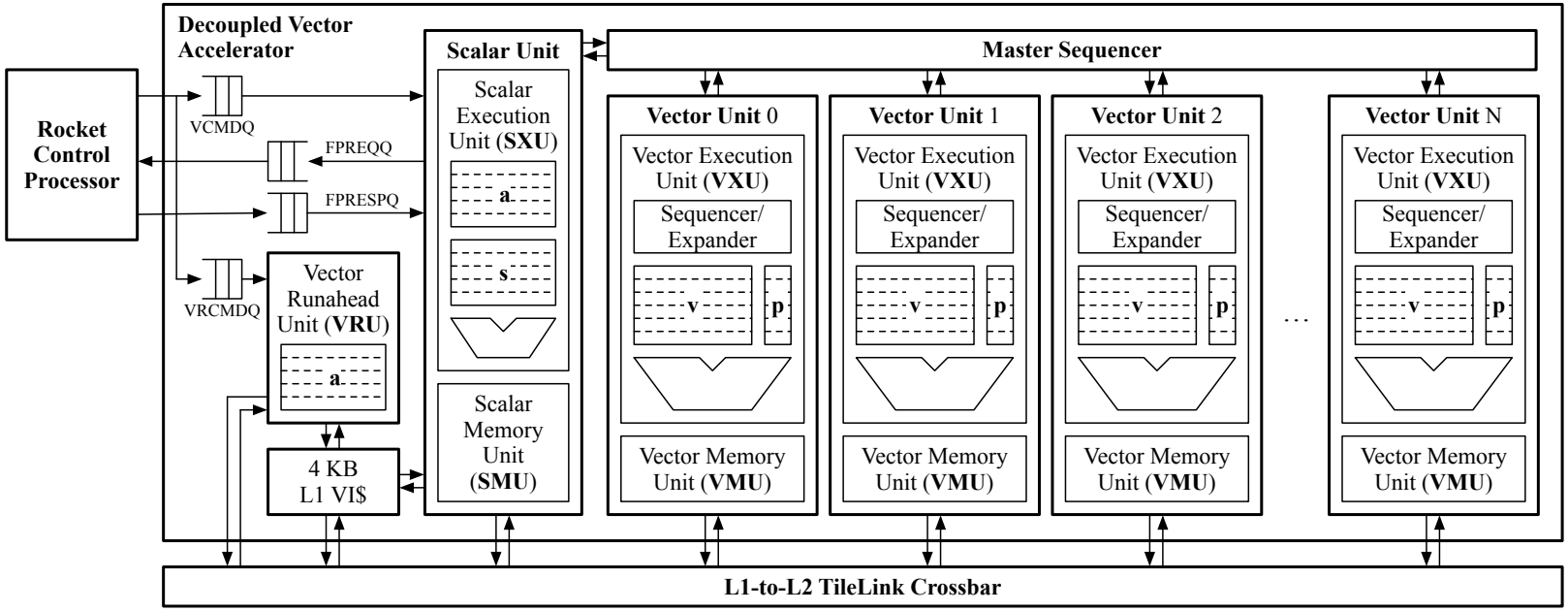


Figure 4.4: Block diagram of Hwacha. VCMDQ = vector command queue, VRCMDQ = vector runahead command queue, FPREQQ = floating-point request queue, FPRESPQ = floating-point response queue.

Hwacha consists of one or more replicated vector lanes assisted by a scalar unit. Internally, the lane is bifurcated into two major components: the *Vector Execution Unit* (VXU), which encompasses the vector data and predicate register files and the functional units, and the *Vector Memory Unit* (VMU), which coordinates data movement between the VXU and the memory system.

Hwacha also features a *Vector Runahead Unit* (VRU) that exploits the inherent regularity of constant-stride vector memory accesses for aggressive yet extremely accurate prefetching. Unlike out-of-order cores with SIMD that rely on reorder buffers and GPUs that rely on multithreading, vector architectures are particularly amenable to prefetching without requiring a large amount of state.

RoCC Frontend and Scalar Unit

Control thread instructions arrive through the Vector Command Queue (VCMDQ). Upon encountering a `vf` command, the scalar unit begins fetching at the accompanying PC from the 4 kB two-way set-associative vector instruction cache (VI$), continuing until it reaches a `vstop` in the vector-fetch block.

The scalar unit includes the address and shared register files and possesses a fairly conventional single-issue, in-order, four-stage pipeline. It handles purely scalar computation, loads, and stores, as well as the resolution of consensual branches and reductions resulting from the vector lanes. The FPU is shared with the Rocket control processor via the floating-point request queue (FPREQQ) and floating-point response queue (FPRESPQ). At the decode stage, vector instructions are steered to the lanes along with any scalar operands.

Vector Execution Unit

The VXU, depicted in Figure 4.5, is broadly organized around four banks. Each contains a 256×128b 1R/1W 8T SRAM that forms a portion of the vector register file (VRF), alongside a 256×2b 3R/1W predicate register file (PRF). Also private to each bank are a local integer arithmetic logic unit (ALU) and predicate logic unit (PLU). A crossbar connects the banks to the long-latency functional units, grouped into clusters whose members share the same operand, predicate, and result lines.

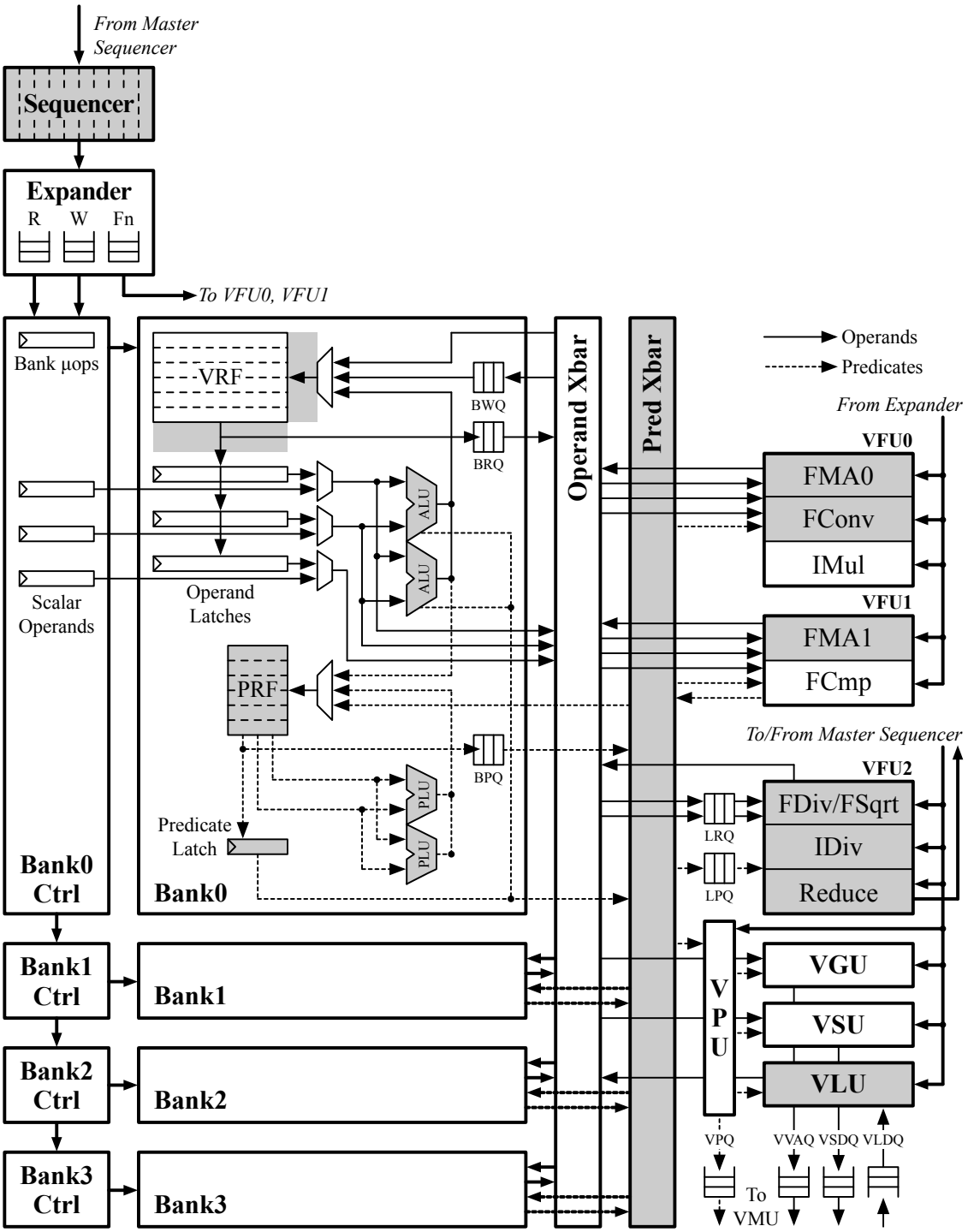


Figure 4.5: Block diagram of the Vector Execution Unit (VXU).
VRF = vector register file, PRF = predicate register file, ALU = arithmetic logic unit, PLU = predicate logic unit, BRQ = bank operand read queue, BWQ = bank operand write queue, BPQ = bank predicate read queue, LRQ = lane operand read queue, LPQ = lane predicate read queue, VFU = vector functional unit, FP = floating-point, FMA = FP fused multiply add unit, FConv = FP conversion unit, FCmp = FP compare unit, FDiv/FSqrt = FP divide/square-root unit, IMul = integer multiply unit, IDiv = integer divide unit, VPU = vector predicate unit, VGU = vector address generation unit, VSU = vector store-data unit, VLU = vector load-data unit, VPQ = vector predicate queue, VVAQ = vector virtual address queue, VSDQ = vector store-data queue, VLDQ = vector load-data queue.

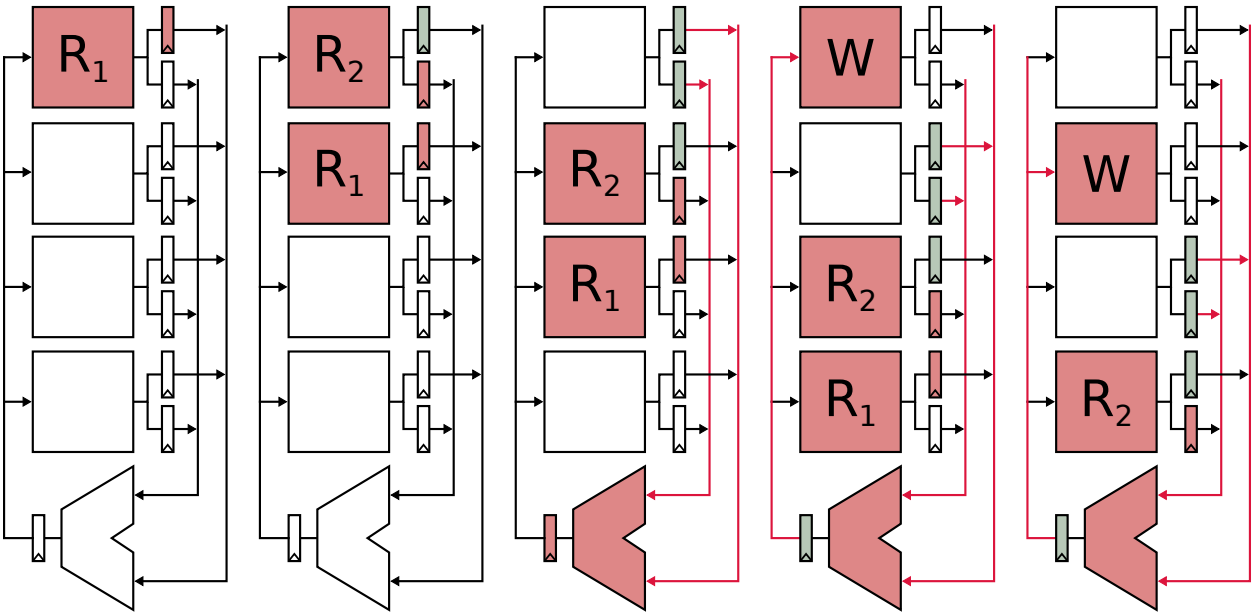
Vector instructions are issued into the sequencer, which monitors the progress of every active operation within that particular lane. The master sequencer, shared among all lanes, holds the common dependency information and other static state. Execution is managed in "strips" that complete eight 64-bit elements worth of work, corresponding to one pass through the banks. The sequencer acts as an out-of-order, albeit non-speculative, issue window: Hazards are continuously examined for each operation; when clear for the next strip, an age-based arbitration scheme determines which ready operation to send to the expander.

The expander converts a sequencer operation into its constituent *micro-ops* (µops), low-level control signals that directly drive the lane datapath. These are inserted into shift registers with the displacement of read and write µops coinciding exactly with the functional unit latency. The µops iterate through the elements as they sequentially traverse the banks cycle by cycle. As demonstrated by the bank execution example in Figure 4.6, this stall-free systolic schedule sustains n operands per cycle to the shared functional units after an initial n-cycle latency. Variable-latency functional units instead deposit results into per-bank queues (BWQs) for decoupled writes, and the sequencer monitors retirement asynchronously. Vector chaining arises naturally from interleaving µops belonging to different operations.

Vector Memory Unit

The per-lane VMUs are each equipped with a 128-bit interface to the shared L2 cache. This arrangement delivers high memory bandwidth, albeit with a trade-off of increased latency that is overcome by decoupling the VMU from the rest of the vector unit. Figure 4.7 outlines the organization of the VMU.

As a memory operation is issued to the lane, the VMU command queue is populated with the operation type, vector length, base address, and stride. Address generation for constant-stride accesses proceeds without VXU involvement. For indexed operations such as gathers, scatters, and AMOs, the Vector Generation Unit (VGU) reads offsets from the VRF into the Vector Virtual Address Queue (VVAQ). Virtual addresses are then translated and deposited into the Vector Physical Address Queue (VPAQ), and the progress is reported to the VXU. The departure of requests is regulated by the lane sequencer to facilitate restartable



(a) Cycle 0 (b) Cycle 1 (c) Cycle 2 (d) Cycle 3 (e) Cycle 4

Figure 4.6: Systolic bank execution of a 2-ary arithmetic operation with a 1-cycle fixed latency. In cycle i, µop R_1 reads the first operand into a buffer at Bank i. In cycle $i+1$, µop R_2 reads the second operand into a buffer at Bank i, while R_1 simultaneously reads the first operand into a buffer at Bank $i+1$. In cycle $i+2$, a crossbar µop (not shown) connects the buffers at Bank i to the two operand lines feeding the shared functional unit at the bottom. In cycle $i+3$, µop W commits the finished result to Bank i. Thus, the operand supply is fully pipelined after an initial 1-cycle latency.

exceptions.

The address pipeline is assisted by a separate predicate pipeline. Predicates must be examined to determine whether a page fault is genuine, and are used to derive the store masks. The VMU supports limited density-time skipping given 2^n runs of false predicates [29].

Unit strides represent a very common case for which the VMU is specifically optimized. The initial address generation and translation occur at a page granularity to circumvent predicate latency and accelerate the sequencer check. To more fully utilize the available memory bandwidth, adjacent elements are coalesced into a single request prior to dispatch. The VMU correctly handles edge cases with base addresses not 128-bit-aligned and irregular vector lengths not a multiple of the packing density.

The Vector Store Unit (VSU) multiplexes elements read from the VRF banks into the Vector Store Data Queue (VSDQ). An aligner module following the VSDQ shifts the entries appropriately for scatters and unit-stride stores with non-ideal alignment.

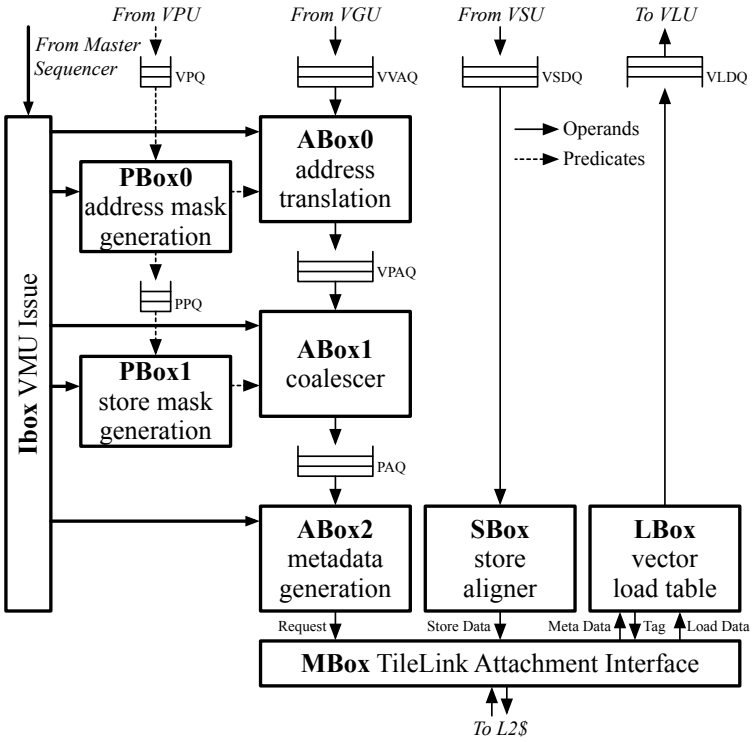


Figure 4.7: Block diagram of the Vector Memory Unit (VMU). Consult Figure 4.5 for VPU, VGU, VSU, VLU, VPQ, VVAQ, VSDQ, VLDQ. VPAQ = vector physical address queue, PPQ = pipe predicate queue, PAQ = pipe address queue.

The Vector Load Unit (VLU) routes data from the Vector Load Data Queue (VLDQ) to their respective banks, using a rotation-based permutation network to handle alignment. As the memory system may arbitrarily order responses, two VLU optimizations become crucial. The first is an opportunistic writeback mechanism that permits the VRF to accept elements out of sequence; this reduces latency and area compared to a reorder buffer. The VLU is also able to simultaneously manage multiple operations to avoid artificial throttling of successive loads by the VMU.

Vector Runahead Unit

The Vector Runahead Unit (VRU) takes advantage of the decoupled nature of Hwacha to hide memory latency through prefetching. The VRU consumes a separate Vector Runahead Command Queue (VRCMDQ) ahead of the scalar unit, identifying future vector-fetch targets and maintaining a runahead copy of the address register file. As it populates the vector instruction cache with impending vector-fetch blocks, it decodes and pre-executes unit-strided vector memory operations to refill the L2 cache in anticipation of the actual requests by the vector lanes. Unlike in other machines, these prefetches are in most cases non-speculative. Since the address registers and the vector length cannot be changed by the worker thread, the information provided to the VRU is certain to be accurate.

Multi-lane Configuration

Hwacha is parameterized to support any 2^n number of identical lanes. Although the master sequencer issues operations to all lanes synchronously, each lane executes entirely decoupled from one another.

To achieve more uniform load-balancing, elements of a vector are striped across the lanes by a runtime-configurable multiple of the sequencer strip size (the "lane stride"), as shown in Figure 4.8. This also simplifies the base calculation for memory operations of arbitrary constant stride, enabling the VMU to reuse the existing address generation datapath as a short iterative multiplier. The striping does introduce gaps in the unit-stride operations performed by an individual VMU, but the VMU issue unit can readily compensate by decomposing the vector into its contiguous segments, while the rest of the VMU remains oblivious. Unfavorable alignment, however, incurs a modest waste of bandwidth as adjacent lanes request the same cache line at these segment boundaries. Figure 4.9 illustrates an example.

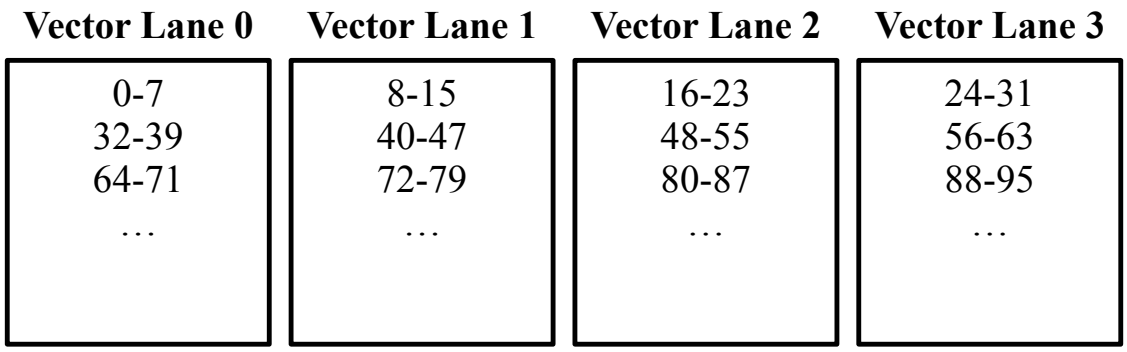


Figure 4.8: Mapping of elements across a four-lane machine

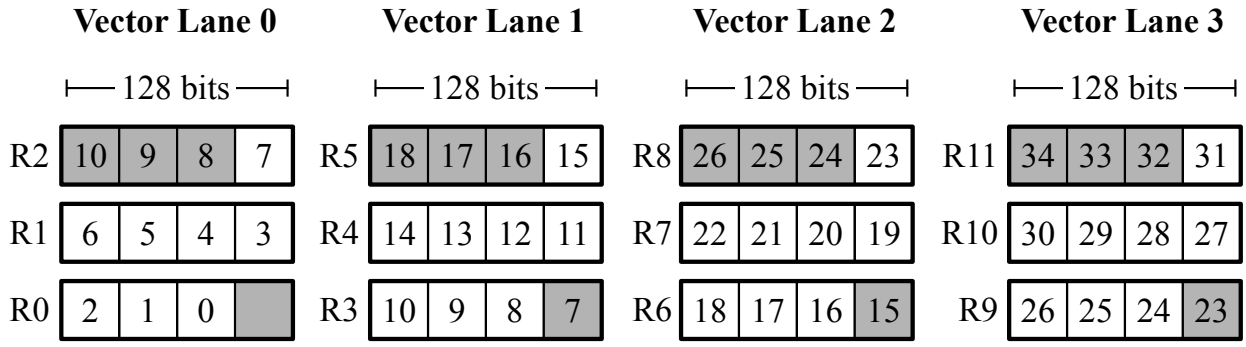


Figure 4.9: Example of redundant memory requests by adjacent lanes. These occur when the base address of a unit-strided vector in memory is not aligned at the memory interface width (128 bits)—in this case, `0x???????4`. Each block represents a 128-bit TileLink beat containing four 32-bit elements. Shaded cells indicate portions of a request ignored by each lane. Note that R2 overlaps with R3, R5 with R6, R8 with R9, etc.

Chapter 5

Mixed-Precision Vector Architecture

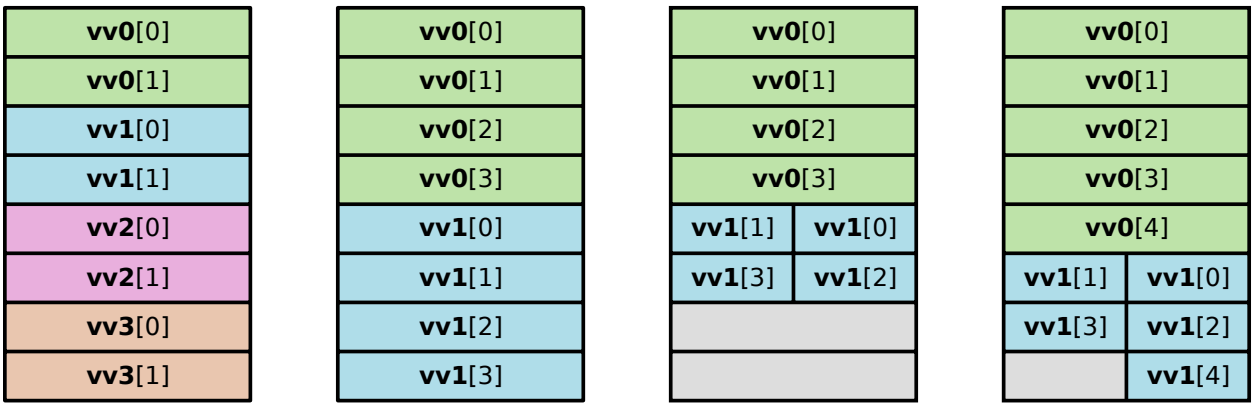
We extend the baseline vector architecture with a set of *High-Occupancy Vector* (HOV) enhancements: automatic subword packing for longer vector lengths and higher-throughput execution of reduced-precision operations.

5.1 Programming Model

Reconfigurable Vector Register File

Since there are advantages to both longer vectors and more numerous registers, Hwacha treats the vector register file (VRF) as a fine-grained configurable resource to maximize utilization. From the control thread, the program indicates the number of architectural registers desired for the subsequent vector-fetch blocks through the `vsetcfg #vv,#vp` instruction. The `#vv` immediate gives the number of vector data registers from 1 to 256. Similarly, `#vp` gives the number of vector predicate registers from 0 to 16; if 0, all vector operations execute unconditionally. In response, the vector unit repartitions the physical register file and recalculates the maximum hardware vector length to fill the capacity. Essentially, this provides software with a mechanism to exchange unused architectural registers for longer hardware vectors.

As a simplified demonstration, Figure 5.1a illustrates one possible mapping of four vector registers to an 8-entry VRF, yielding an HVL of 2. Halving the set of vector registers then doubles the HVL to 4, as shown by Figure 5.1b.



(a) `vsetcfg 4` HVL = 2 (b) `vsetcfg 2` HVL = 4 (c) `vsetcfg 1,1` HVL = 4 (d) `vsetcfg 1,1` HVL = 5

Figure 5.1: Simplified logical view of an 8-entry VRF

```
1  csaxpy_control_thread_hov:
2    vsetcfg 0, 2, 0, 1
3    vmcs vs1, a2
4  stripmine:
5    vsetvl  t0, a0
6    vmca    va0, a1
7    vmca    va1, a3
8    vmca    va2, a4
9    vf      csaxpy_worker_thread
10   add     a1, a1, t0
11   slli    t1, t0, 2
12   add     a3, a3, t1
13   add     a4, a4, t1
14   sub     a0, a0, t0
15   bnez    a0, stripmine
16   ret
17
18 csaxpy_worker_thread:
19       vlb     vv0, (va0)
20       vcmpez  vp0, vv0
21 !vp0 vlw      vv0, (va1)
22 !vp0 vlw      vv1, (va2)
23 !vp0 vfma     vv0, vv0, vs1, vv1
24 !vp0 vsw      vv0, (va2)
25       vstop
```

Figure 5.2: Conditional SAXPY kernels updated for HOV. Except for the expanded `vsetcfg` instruction, highlighted in red, the assembly code is otherwise identical to that of Figure 4.2b.

HOV introduces another dimension of reconfigurability in the VRF: Apart from specifying the number of named vector registers, it also allows for selecting their data type widths from a preset range. The programming model is unaffected except for a singular change to the `vsetcfg` instruction. `#vv` is now decomposed into individual subfields `#vvd`, `#vvw` and `#vvh`, denoting the number of requested doubleword (64 bit), word (32 bit), and halfword (16 bit) registers, respectively. Register identifiers are assigned in aggregate by ascending numerical order; higher addresses correspond to decreasing precision. Given `vsetcfg 1,2,4,0`, for example, the program interacts with `vv0` as doublewords; `vv1` and `vv2` as words; and `vv3` through `vv6` as halfwords.

From this specification, the hardware automatically determines the appropriate register file mapping, subdivides physical registers as needed into multiple narrower architectural registers, and extends the hardware vector length in proportion to gains in storage density. As is apparent in Figure 5.1c, reduced precision leaves vacant space in the high-order bits of registers. With some rearrangement of elements, of which Figure 5.1d is simply one proposal, contiguous subword packing permits the HVL to be increased by 1.

The developer effort to embrace this optimization is minimal. For many cases, it suffices to modify the `vsetcfg` instructions alone without rewriting any other code. The CSAXPY kernel, updated for HOV in Figure 5.2, now uses vector registers configured as words instead of doublewords, but it is otherwise identical to the baseline example. For higher-level programming languages, the compiler's register allocator performs most of the work.

Overall, this approach offers a cleaner abstraction than subword SIMD to express mixed precision, particularly in regard to portability. From the perspective of software, architecture register widths are being readjusted instead of packed elements being exposed; the subword packing occurs *implicitly* from the vector register configuration, invisible to the program. An implementation lacking full mixed-precision support may ignore configuration hints and still execute the same code, albeit at reduced efficiency.

Portability Considerations

It is the responsibility of the architecture to enforce a minimum set of guarantees to ensure consistent program behavior across the range of possible vector machine implementations.

Irrespective of the microarchitectural decision to support subword packing for all, none, or a subset of precisions, all operations and trap conditions should be reproducible for the requested vector register file configuration. This concerns, in other words, the ability to execute HOV code with the same result whether or not subword packing is active at the physical level. Portability in this sense can be viewed as matters of both functional correctness and real-world implementation flexibility.

To reason about portability, it is first necessary to distinguish between the size of the data type, as determined by the opcode (the "precision"), and the size of the enclosing container (the register "width").[1] Then the portability issue can be reduced to the problem of determining all legal combinations of precisions and widths for reads and writes. The original baseline configuration, in which all registers are statically constrained to the largest architectural width (64 bits), exemplifies maximal flexibility in the sense that registers are permitted to hold any data of equal or lesser precision. Ideally, in a fully orthogonal model generalized for multiple widths, the default baseline behavior should manifest naturally as the degenerate case, not as an isolated special case.

The desire to impose as few restrictions as possible on the placement of data informs an intuitive maxim: *An operation involving a given set of registers is valid if and only if it causes no unrecoverable loss of information.* Rules governing the four possible relationships between precision and width follow accordingly:

1. Writing higher-precision result to narrower destination: Prohibited on obvious grounds.

2. Writing lower-precision result to wider destination: Permitted; must be accompanied by sign extension.

3. Reading lower-precision operand from wider source: Permitted; upper bits irrelevant to the operation itself can be safely discarded, which already occurs implicitly inside the functional unit.

4. Reading higher-precision operand from narrower destination: Permitted; must be accompanied by sign extension.

[1] In this discussion, the term "register" refers to the individual storage allocated per element, rather than the entire vector, unless otherwise noted.

Most importantly, these enable an implementation to allocate an architecturally narrow register as a physically wider register without requiring it to emulate the former's truncation and sign extension behavior. In other words, the baseline datapath could remain wholly agnostic to the configuration of the architectural register; the only mandatory task is the sign extension of a lower-precision result in Case 2, which is exclusively based on the opcode known *a priori* by the datapath.

The practice of universal sign extension does occasionally leads to some non-intuitive situations. Suppose a `VLHU` instruction ("vector load halfword unsigned") writes to a vector register configured as halfwords, and those contents are then stored back to memory with a `VSD` ("vector store doubleword"). From the above specification, one might expect the high-order bits of the doublewords to be sign-extended rather than uniformly cleared. However, if the implementation elects to physically allocate the vector register in doublewords, the store data would happen to consist of zero-extended values instead. The contradiction stems from the fact that the 16-bit load data must be treated as implicitly 17-bit to accurately capture their unsigned nature. Therefore, Case 1 renders invalid any `VLHU` with a halfword-sized destination; the same applies to `VLWU` and word-sized destinations.

Case 1 more broadly prescribes that conformant implementations, including the baseline machine if it aims to run HOV code, respect the extended `vsetcfg` command. Although a microarchitecture may internally assign all requested halfword-sized and word-sized registers to doublewords, it must nonetheless trap on invalid instructions as if those registers were of the intended type.

Polymorphic Instruction Sets

Thus far, it has been assumed that a distinct opcode corresponds to each supported data type (e.g., `VFMADD.D`, `VFMADD.S`, `VFMADD.H` for double/single/half-precision FMAs, respectively). One could imagine *polymorphic instructions* whose input/output precisions are instead determined by the source and destination register specifiers in conjunction with the runtime configuration of the vector register file.

Orthogonality in the instruction set could therefore be achieved without excessive consumption of opcode space or encoding complexity, an increasingly desirable proposition as

new mixed-precision operations are added. However, no longer would vector registers be able to hold values of narrower precision than the configured width, which may significantly constrain register reuse by a compiler.

We choose not investigate this concept any further here, as it would involve a drastic overhaul of the vector ISA, and leave it as future work.

5.2 Microarchitecture

The microarchitectural extensions for HOV focus on the modules shaded in Figure 4.4, with modifications falling into two broad categories:

- Datapath: parallel functional units and subword compaction/extraction logic
- Control: data hazard checking when chaining vector operations of unequal throughput

Register Mapping

The vector register file banks are segmented into doubleword, word, and halfword regions, with the vector registers interleaved as shown in Figure 5.3. Thus, the starting physical address of a vector register can be straightforwardly calculated by the sum of the architectural register identifier and the associated region offset, and traversing the elements in the vector involves a constant stride equal to the total number of architectural registers of that type.

It is first tempting to assign consecutive element indices to contiguous subwords within a physical entry, but this naive approach soon fails. Consider an operation which reads from both a doubleword register (i.e., `vv0`) and a word register (i.e., `vv2`). Should the vector registers be arranged as depicted in Figure 5.4, elements 0 and 1 of `vv0` reside in Bank 0 while elements 2 and 3 reside in Bank 1. During each strip, the read µop for `vv0` visits Bank 0 for one cycle and then proceeds to Bank 1 in the next, as usual. However, elements 0 to 3 of `vv2` are all located in Bank 0. The read µop for `vv2` must therefore reserve Bank 0 for two cycles, retrieving a different half of the same entry each time, and then halt before Bank 1. For the following strip, elements 4 to 7, the `vv2` µop must begin directly at Bank 1 instead, where it again persists for two cycles.

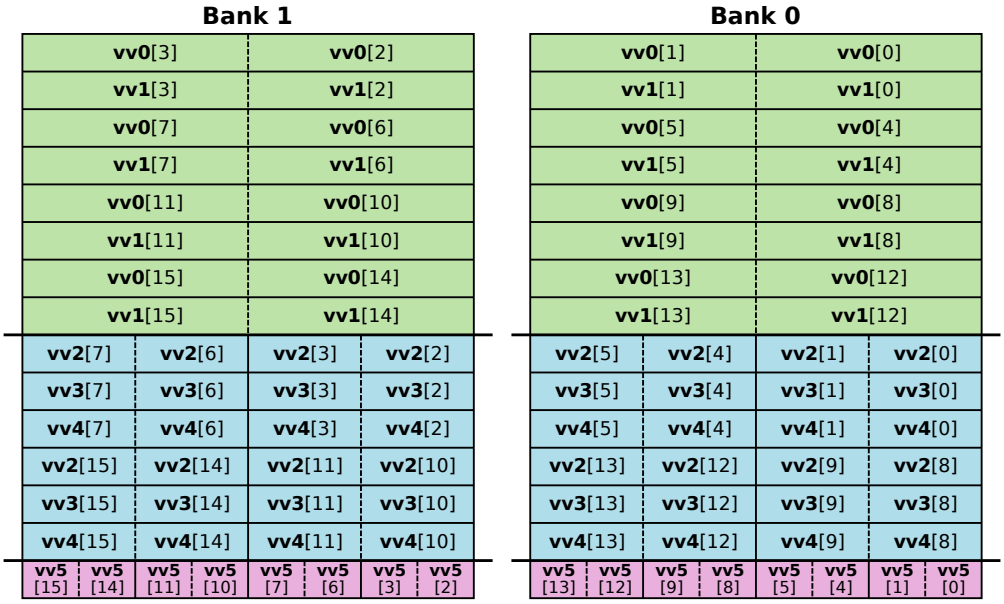


Figure 5.3: Actual striped mapping to a 2-bank physical VRF

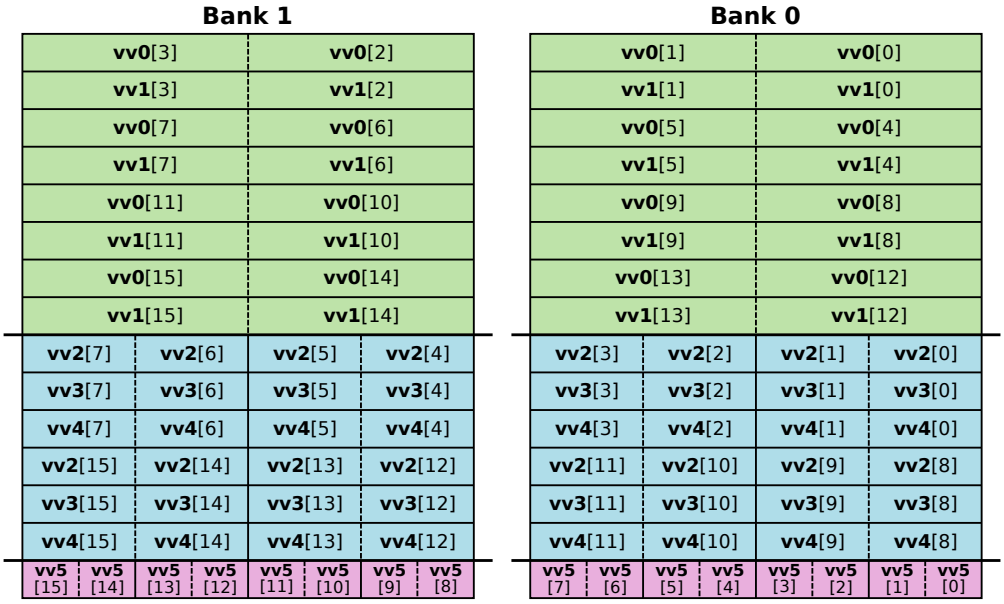


Figure 5.4: Hypothetical naive mapping to a 2-bank physical VRF

With only one read and write port per bank, the non-uniform pace of μops would lead to myriad bank conflicts. Circumventing these structural hazards would require generating unique schedules for each bank, counter to the systolic execution principle. The control logic would then become quickly intractable. To preserve a regular access schedule, all elements with the same index should reside within the same bank. This inspires the *element-partitioned* scheme demonstrated by Figure 5.3, which stripes elements such that adjacent slices (pairs of elements) coincide with indices separated by the strip size.

The size of the SRAM arrays remain unchanged. However, the predicate register file is widened to 8 bits so that all predicates associated with a maximally packed SRAM entry of eight halfwords are accessible through a single port in one cycle. The predicate mapping is also striped in the manner above.

Maximum Hardware Vector Length

The maximum hardware vector length depends on the number of SRAM entries across all banks and lanes in the machine, as well as the number of architectural registers requested by the program. The extended HVL is calculated with the following formula, implemented as a lookup table in hardware:

$$\text{HVL} = \left\lfloor \frac{4r}{4d + 2w + h} \right\rfloor \times n_{\text{slices}} \times n_{\text{banks}} \times n_{\text{lanes}} \tag{5.1}$$

Variables d, w, and w refer to the desired number of doubleword, word, and halfword vector registers, respectively. For the default configuration, $r = 256$ (number of SRAM entries per bank), $n_{\text{slices}} = 2$, and $n_{\text{banks}} = 4$. If subword packing is disabled at design elaboration, the formula reverts to one equivalent to the baseline:

$$\text{HVL} = \left\lfloor \frac{r}{d + w + h} \right\rfloor \times n_{\text{slices}} \times n_{\text{banks}} \times n_{\text{lanes}} \tag{5.2}$$

In practice, the HVL is also constrained by the number of predicate registers requested, but the concomitant widening of the predicate register file, while retaining the same depth, ensures that this is no more a limiting factor for the HOV design than the baseline.

Multi-Rate Vector Chaining

Subword parallelism enables a subset of operations to execute at a higher throughput proportional to the packing density. The overall *rate* of an operation is constrained by the maximum of the source and destination register widths, as well as by the availability of parallel functional units for that particular operation. Both the register type and operation rate are determined during the decode stage and recorded in the master sequencer entry.

Chaining allows vector operations to execute in an interleaved manner. Interim results may be consumed by subsequent operations without waiting for the entire vector to complete. Fundamentally, newer and faster operations must be prevented from overtaking older and slower operations on which a data dependency exists. Comparing the remaining vector lengths between active sequencer entries is a necessary but not sufficient check: As these values are updated at time of sequencing rather than commit, the preceding strip may still be partially in-flight. Although a conservative solution is possible by maintaining a separation greater than a strip between dependent operations, this injects excessive dead cycles and significantly degrades the performance gains of chaining.

To facilitate closer scheduling of successive operations, the sequencer additionally examines the expander queues for pending write µops. The baseline implementation detects hazards by searching for collisions in the physical register addresses. This method remains correct in the multi-rate case, but it is suboptimal as it unnecessarily prohibits strips from simultaneously working on disjoint parts of the same SRAM entries—circumstances which may be likened to false sharing.

The HOV implementation instead uses a finer-grained interval check to determine any overlaps between strips. Expander µops are tagged with a starting *strip index*, denoted t, and rate information, denoted $r \in \{1, 2, 4\}$. Given a valid expander entry, its strip interval is delineated by t and $t' = t + r_{\text{exp}}$, the ending strip index. Analogously define s and $s' = s + r_{\text{seq}}$ for the sequencer operation to be next issued. Since an operation always progresses by r strips whenever sequenced, the starting strip indices are always some multiple of r. This eliminates any possibilities of a partial overlap: Either $[s, s')$ entirely contains $[t, t')$ if $r_{\text{exp}} \leq r_{\text{seq}}$, or vice-versa. The checking logic therefore simplifies to:

$$(s \leq t \wedge t' \leq s') \vee (t \leq s \wedge s' \leq t') \tag{5.3}$$

which asserts whenever a hazard (overlap) exists.

Comparators constitute the primary hardware cost of this scheme. The comparison must be performed for every sequencer entry against every expander entry. As an optimization, less expensive identity comparators can replace two magnitude comparators in each check; given the signal $a \leq b$, its mirror signal $b \leq a$ can be obtained by $(a = b) \vee (\neg(a \leq b))$.

Compaction/Extraction Logic

At the interface between the register file and the rest of the lane datapath, bank read/write µops use the register type and rate information to unpack operands and repack results. A read can be viewed as a transformation between an "input" format based on type—how subwords are packed in a physical register—and an "output" format based on rate—how the functional units expects sign-extend subwords to be positioned. A write is essentially the inverse transformation. There exist six valid combinations of (type, rate): $(d, 1)$, $(w, 1)$, $(w, 2)$, $(h, 1)$, $(h, 2)$, and $(h, 4)$. Of these, three are degenerate "pass-through" cases where the type and rate match, and the contents are conveyed unaltered.

The unpack module consists of a right shift based on the subword index, followed by extraction and sign extension appropriate for the type/rate combination. This corresponds to approximately four levels of 2:1 multiplexers. As the logic sits between the SRAM read port and the operand buffers, care must be taken to not impact the SRAM critical path.

The repack module, inserted after the result crossbar, performs the reverse maneuver: concatenation of the properly sign-extended results followed by a left shift. Identical logic also resides in the decoupled functional units (i.e., VFU2 and VSU). As these use an alternative writeback path through the BWQs, they must be aware of subword addressing scheme.

Functional Units

To enable full throughput, two more single-precision and six more half-precision FMA units are instantiated in both FMA0 and FMA1. Additionally, there are two more half-to-single and two more single-to-half floating-point conversion units in FConv. The integer ALUs local to each bank are appropriately partitioned, and the 1 b PLUs are replicated six more times to match the widened predicate register file.

Chapter 6

Evaluation Framework

This section outlines how we evaluate HOV against the baseline Hwacha design, and validate our decoupled vector-fetch architecture against a commercial GPU that can run OpenCL kernels. The evaluation framework used in this study is described in Figure 6.1. The high-level objective of our evaluation framework is to compare realistic performance, power/energy, and area numbers using detailed VLSI layouts and compiled OpenCL kernels, not only hand-tuned assembly code.

As a first step towards that goal, we wrote a set of OpenCL microbenchmarks for the study (see Section 6.1), and developed our own LLVM-based scalarizing compiler that can generate code for the Hwacha vector-fetch assembly programming model (see Section 6.2). These microbenchmarks are compiled with our custom compiler and ARM's stock compiler. We then selected realistic parameters for the Rocket Chip SoC generator to match the Samsung Exynos 5422 SoC, which has an ARM Mali-T628 MP6 GPU. We chose that specific SoC because it ships with the ODROID-XU3 development board that has instrumentation capabilities to separately measure power consumption of the Mali GPU (see Section 6.3 for more details). We synthesize and place-and-route both Hwacha designs (the baseline and HOV) in a commercial 28 nm process akin to the 28 nm high-κ metal gate (HKMG) process used to fabricate the Exynos 5422, and run the compiled microbenchmarks on both gate-level simulators to obtain accurate switching activities and performance numbers (see Section 6.4 for more details). The switching activities are then combined with the VLSI layout to generate accurate power/energy numbers. These numbers are subsequently compared against

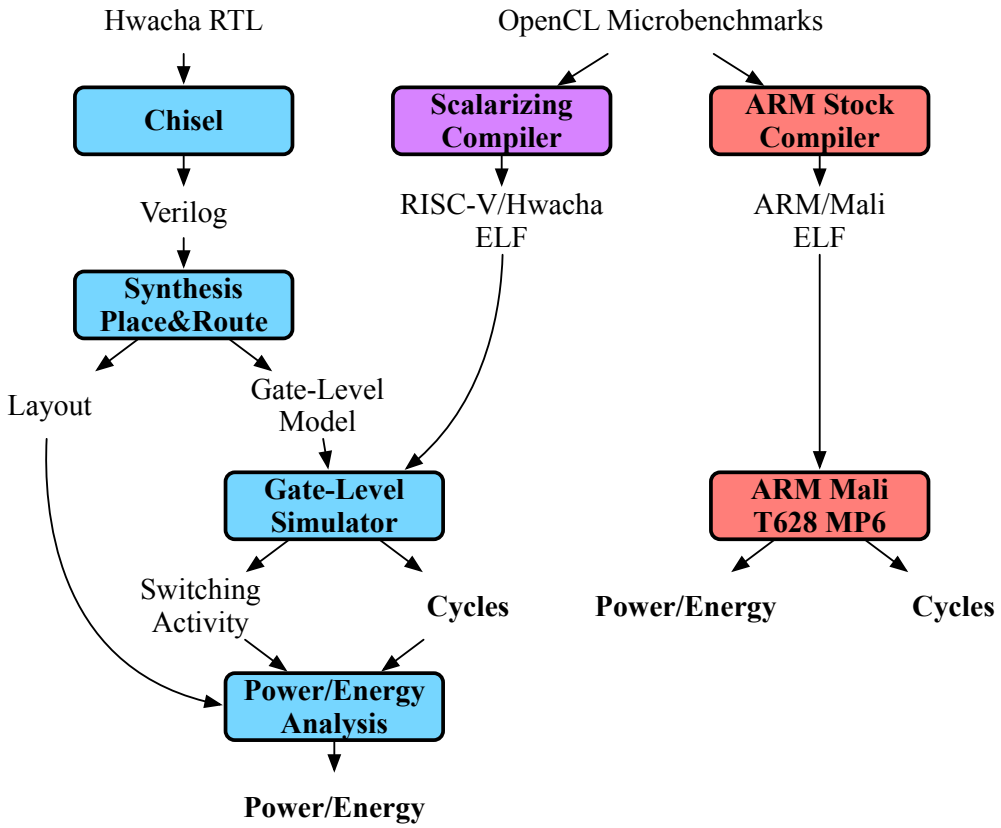


Figure 6.1: Hwacha evaluation framework

each other, but are also validated against ARM Mali performance and power/energy numbers obtained from the ODROID-XU3 board. Due to time constraints, we were only able to push through single-lane configurations for both baseline and HOV designs.

6.1 Microbenchmarks

For the study, we wrote four types of OpenCL kernels in four different precisions. Table 6.1 lists all microbenchmarks.

The microbenchmarks are named with a prefix and a suffix. The suffixes denote the type of the kernel: `axpy` for $\mathbf{y} \leftarrow a\mathbf{x} + \mathbf{y}$, a scaled vector accumulation; `gemm` for dense matrix-matrix multiplication; and `filter` for a Gaussian blur filter, which computes a stencil over an image. The `mask` versions of `filter` accept an additional input array determining whether to

Kernel	Mixed-Precision	Predication
{s,d}axpy		
{hs,sd}axpy	✓	
{s,d}gemm		
{hs,sd}gemm	✓	
{s,d}gemm-opt		
{hs,sd}gemm-opt	✓	
{s,d}filter		
{hs,sd}filter	✓	
mask-{s,d}filter		✓
mask-{hs,sd}filter	✓	✓

Table 6.1: Listing of all microbenchmarks

compute that point, thus exercising predication. For reference, we also wrote hand-optimized versions of `gemm-opt` in order to gauge the code generation quality of our custom OpenCL compiler. For $C \leftarrow A \times B$, `gemm-opt` loads unit-strided vectors of C into the vector register file, keeping them in place while striding through the A and B matrices. The values from B are unit-stride vectors; the values from A reside in scalar registers.

The prefix denotes the precision of the operands: `h`, `s` and `d` for half-, single-, and double-precision, respectively. `sd` signifies that the benchmark's inputs and outputs are in single-precision, but the intermediate computation is performed in double-precision. Similarly, `hs` signifies that the inputs and outputs are in half-precision, but the computation is performed in single-precision.

6.2 OpenCL Compiler

We developed an OpenCL compiler based on the PoCL OpenCL runtime [30] and a custom LLVM [31] backend. The main challenges in generating Hwacha vector code from OpenCL kernels are moving thread-invariant values into scalar registers [32]–[34], identifying stylized memory access patterns, and using predication effectively [35]–[37]. Thread-invariance is determined using the variance analysis presented in [34], and is performed at both the LLVM IR level and machine instruction level. This promotion to scalar registers avoids redundant values being stored in vector registers, improving register file utilization. In addition to

scalarization, thread invariance can be used to drive the promotion of loads and stores to constant or unit-strided accesses. Performing this promotion is essential to the decoupled architecture because it enables prefetching of the vector loads and stores.

To fully support OpenCL kernel functions, the compiler must also generate predicated code for conditionals and loops. Generating efficient predication without hardware divergence management requires additional compiler analyses. The current compiler has limited predication support, but we plan to soon generalize it for arbitrary control flow, based on [37].

Collecting energy results on a per-kernel basis requires very detailed, hence time-consuming, simulations. This presents a challenge for evaluating OpenCL kernels, which typically make heavy use of the OpenCL runtime and online compilation. Fortunately, OpenCL supports offline compilation, which we rely upon to avoid simulating the compiler's execution. To obviate the remaining runtime code, we augmented our OpenCL runtime with the ability to record the inputs and outputs of the kernels. Our runtime also generates glue code to push these inputs into the kernel code and, after execution, to verify that the outputs match. The effect is that only the kernel code of interest is simulated with great detail, substantially reducing simulation runtime.

6.3 Samsung Exynos 5422 and the ARM Mali-T628 MP6 GPU

Figure 6.2 shows the block diagram of the Samsung Exynos 5422 SoC. The quad Cortex-A15 complex, quad Cortex-A7 complex, and the ARM Mali-T628 MP6 are connected through the CCI-400 cache coherent interconnect to talk to two LPDDR3 channels of 1 GB running at 933 MHz [38], [39]. Table 6.2 presents the specific Rocket Chip SoC generator parameters chosen to match the Samsung Exynos 5422 SoC.

The Mali-T628 MP6 GPU has six shader cores (termed MPs, or multiprocessors) that run at 600 MHz, exposed as two sets of OpenCL devices. Without explicitly load balancing the work on these two devices by software, the OpenCL kernel can either only run on the two shader core device or on the four shader core device. We first run the microbenchmarks on the two shader core device, called *Mali2*, and again on the four shader core device, called

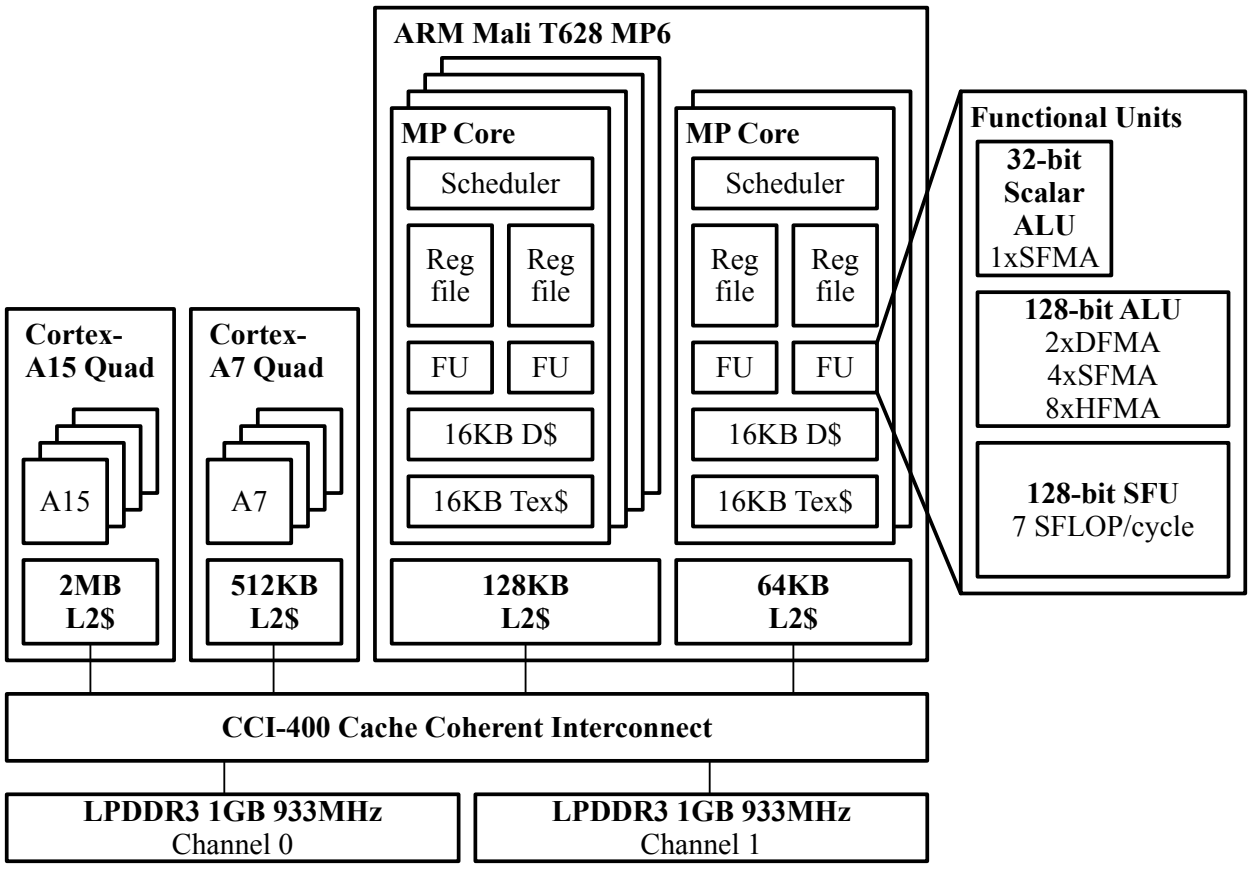


Figure 6.2: Samsung Exynos 5422 block diagram

Mali4. Each shader core has four main pipes: two arithmetic pipes, one load/store pipe with a 16 KB data cache, one texture pipe with a 16 KB texture cache. Threads are mapped to either arithmetic pipe, which is a 128-bit wide VLIW SIMD execution pipeline. The compiler needs to pack three instructions per very long instruction word. The three instruction slots are a 32-bit scalar FMA (fused multiply add) unit, a 128-bit SIMD unit (which supports two 64-bit FMAs, four 32-bit FMAs, or eight 16-bit FMAs), and a 128-bit SFU (special functional unit) unit for dot products and transcendentals. Each shader core has an associated 32 KB of L2 cache, making the total capacity 192 KB. Further details on the Mali architecture and how the L2 cache is split among these two devices are sparse; however, [40] and [41] provide some insight into the organization of Mali.

To measure power consumption of the various units, we sample the current through three separate power rails, distinguishing the power consumption of the CPU complex, the GPU,

Component	Settings
Hwacha vector unit	baseline/HOV, 1/2/4 lanes
Hwacha L1 vector inst cache	4 KB, 2-ways
Rocket L1 data cache	16 KB, 4 ways
Rocket L1 inst cache	16 KB, 4 ways
L2 Cache	256 KB/bank, 8 ways, 4 banks
Cache coherence	MESI protocol directory bits in L2$ tags
DRAMSim2	LPDDR3, 933 MHz 1 GB/channel, 2 channels

Table 6.2: Used Rocket Chip SoC generator parameters

and the memory system. We average these samples over the kernel execution, and use the average power and kernel runtime to compute the energy consumed by the Mali GPU during kernel execution. We examine this comparison in detail in the next section.

One MP possesses approximately the same arithmetic throughput as one Hwacha vector lane with mixed-precision support. Each vector lane is 128 bits wide, and has two vector functional units that each support two 64-bit FMAs, four 32-bit FMAs, or eight 16-bit FMAs.

6.4 RTL Development and VLSI Flow

The Hwacha RTL is written in Chisel [23], a domain-specific hardware description language embedded in the Scala programming language. Because Chisel is embedded in Scala, hardware developers can apply Scala's modern programming language features, such as parameterized types and object-oriented and functional programming, for increased productivity. Chisel generates both a cycle-accurate software model as well as synthesizable Verilog that can be mapped to standard FPGA or ASIC flows. We also use a custom random instruction generator tool to facilitate verification of the vector processor RTL.

We use the Synopsys physical design flow (Design Compiler, IC Compiler) to map the Chisel-generated Verilog to a standard cell library and memory-compiler-generated SRAMs in a widely used commercial 28 nm process technology, chosen for similarity with the 28 nm HKMG process in which the Exynos 5422 is fabricated. We use eight layers out of ten

for routing, leaving two for the top-level power grid. The flow is highly automated to enable quick iterations through physical design variations. When coupled with the flexibility provided by Chisel, this flow allows a tight feedback loop between physical design and RTL implementation. The rapid feedback is vital for converging on a decent floorplan to obtain acceptable quality of results: A week of physical design iteration produced approximately 100 layouts and around a 50% clock frequency increase when tuning the single-lane design.

We measure power consumption of the design using Synopsys PrimeTime PX. Parasitic RC constants for every wire in the gate-level netlist are computed using the TLU+ models. Each microbenchmark is executed in gate-level simulation to produce activity factors for every transistor in each design. The combination of activity factors and parasitics are fed into PrimeTime PX to produce an average power number for each benchmark run. We derive energy dissipation for each benchmark from the product of average power and runtime (i.e., cycle count divided by implementation clock rate).

Chapter 7

Preliminary Evaluation Results

We compare the HOV design against the baseline Hwacha design in terms of area, performance, and energy dissipation to observe the impact of our mixed-precision extensions. After validating our simulated memory system against that of the Exynos 5422 SoC, we use our OpenCL microbenchmark suite to compare the two Hwacha implementations to the ARM Mali-T628 MP6 GPU.

Due to time constraints, only the single-lane Hwacha configurations have been fully evaluated. Consequently, it must be noted that the comparisons against the Mali2 and Mali4 devices were not perfectly fair from Hwacha's perspective, given that the former have the advantage of twice and quadruple the number of functional units, respectively. Nevertheless, the results are encouraging in light of this fact, although they should be considered still preliminary, as there remain substantial opportunities to tune the benchmark code for either platform.

7.1 Memory System Validation

We configure DRAMSim2 [25] with timing parameters from a Micron LPDDR3 part to match those of the dual-channel 933 MHz LPDDR3 modules on the Samsung Exynos 5422 SoC. We then use `ccbench` to empirically confirm that our simulated memory hierarchy is similar to that of the Exynos 5422. The `ccbench` benchmarking suite [42] contains a variety of benchmarks to characterize multi-core systems. We use `ccbench`'s `caches` benchmark,

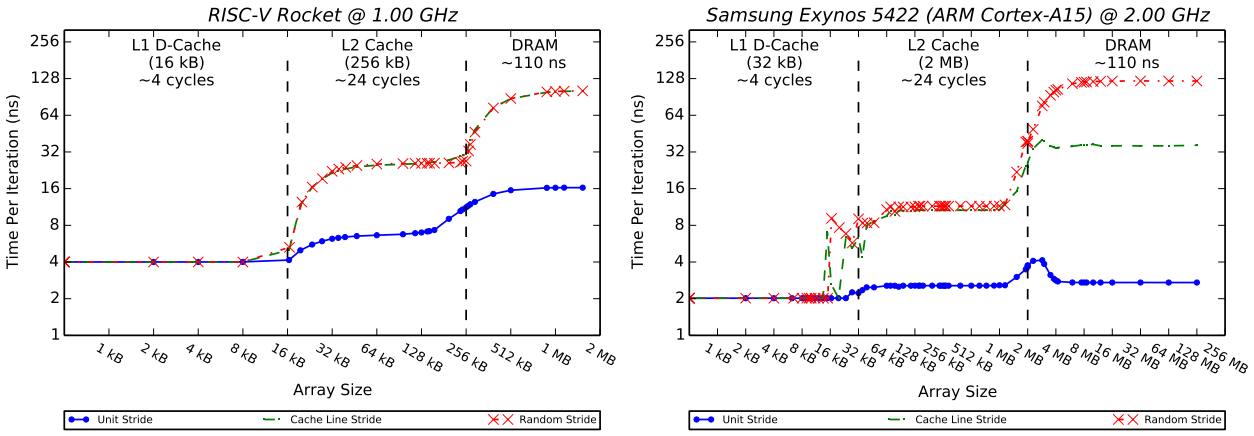


Figure 7.1: ccbench "caches" memory system benchmark

which performs a pointer chase to measure latencies of each level of the memory hierarchy.

Figure 7.1 compares the performance of our cycle-accurate simulated memory hierarchy against the Exynos 5422. On the simulated Rocket core, ccbench measures cycles, normalized to nanoseconds by assuming the 1 GHz clock frequency attained by previous silicon implementations of Rocket [22]. On the Exynos 5422, ccbench measures wall-clock time.

This comparison reveals that our simulated system and the Exynos 5422 match closely in terms of both cache latency and LPDDR3 latency. On both a 1 GHz Rocket and a 2 GHz ARM Cortex-A15, the L1 hit latency is approximately 4 cycles, and the L2 hit latency is approximately 22 cycles. The simulated LPDDR3 used in our experiments and the LPDDR3 in the Exynos 5422 exhibit similar latencies of approximately 110 ns.

Nevertheless, one significant difference remains in the inclusion of a streaming prefetcher within the Cortex-A15, which reduces the latency of unit-stride and non-unit-stride loads and stores [43].

7.2 Area and Cycle Time

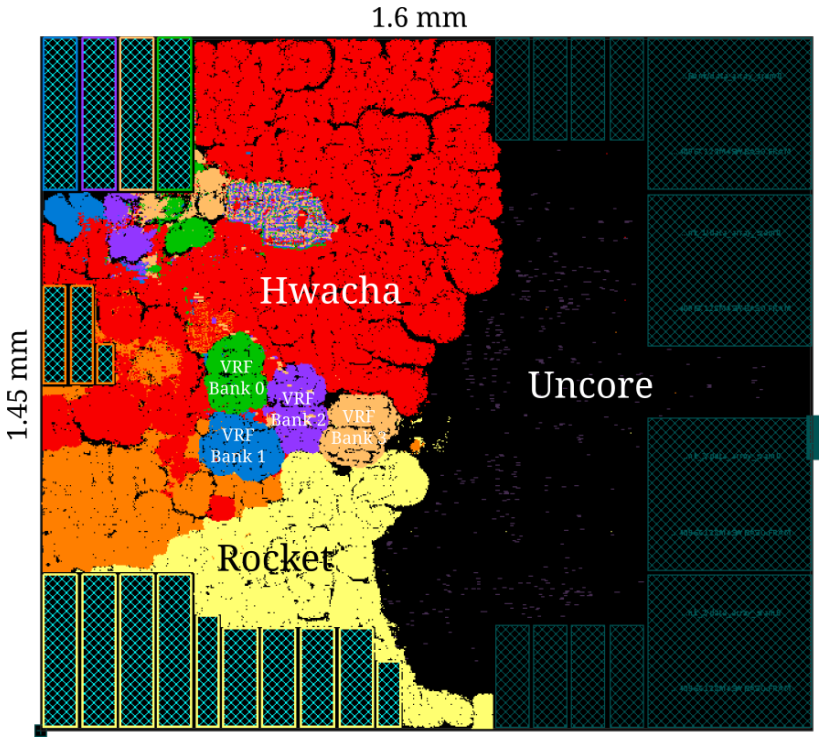
Table 7.1 lists the clock frequencies and total area numbers obtained for a variety of Hwacha configurations. Figure 7.2 shows the layout for the single-lane HOV design.

Recall that Mali2 is clocked at 600 MHz but contain approximately twice the number of functional units. To attempt to match functional unit bandwidth, we target Hwacha for a nominal frequency of 1.2 GHz. While actual frequencies fall slightly short of the ideal, they are still generally above 1 GHz. However, the aggressive physical design does involve a trade-off in area.

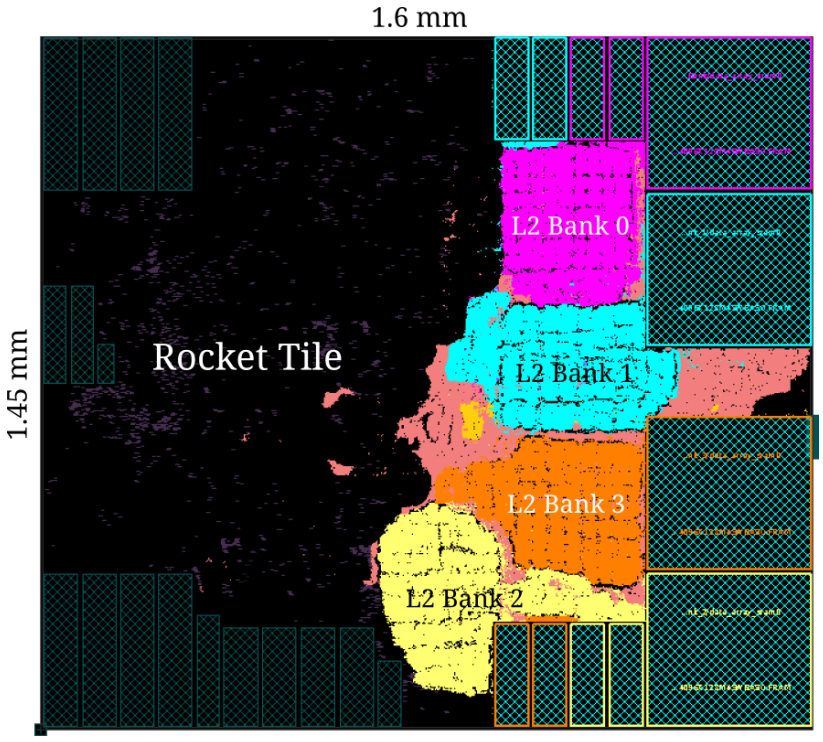
As seen in Figure 7.3, the area overhead for the HOV extensions spans from 4.0% in the single-lane design to 12.5% in the four-lane design. The additional functional units account for a large portion of the increase. The banks also become somewhat larger from the widening of the predicate register file, as does the control due to the non-trivial amount of comparators needed to implement the expander hazard check in the sequencers.

Table 7.1: VLSI quality of results. Columns not marked as "PAR" are results from synthesis.

	Hwacha				Hwacha + HOV			
Lanes	1	1	2	4	1	1	2	4
mm^2	2.11	2.23	2.60	3.59	2.21	2.32	2.82	4.04
ns	0.90	0.95	0.93	0.93	0.94	0.98	1.02	1.08
GHz	1.11	1.05	1.08	1.08	1.06	1.02	0.98	0.93
PNR?		✓				✓		



(a) Tile: Rocket and Hwacha



(b) Uncore: L2 cache and interconnect

Figure 7.2: Layout of the single-lane Hwacha design with HOV

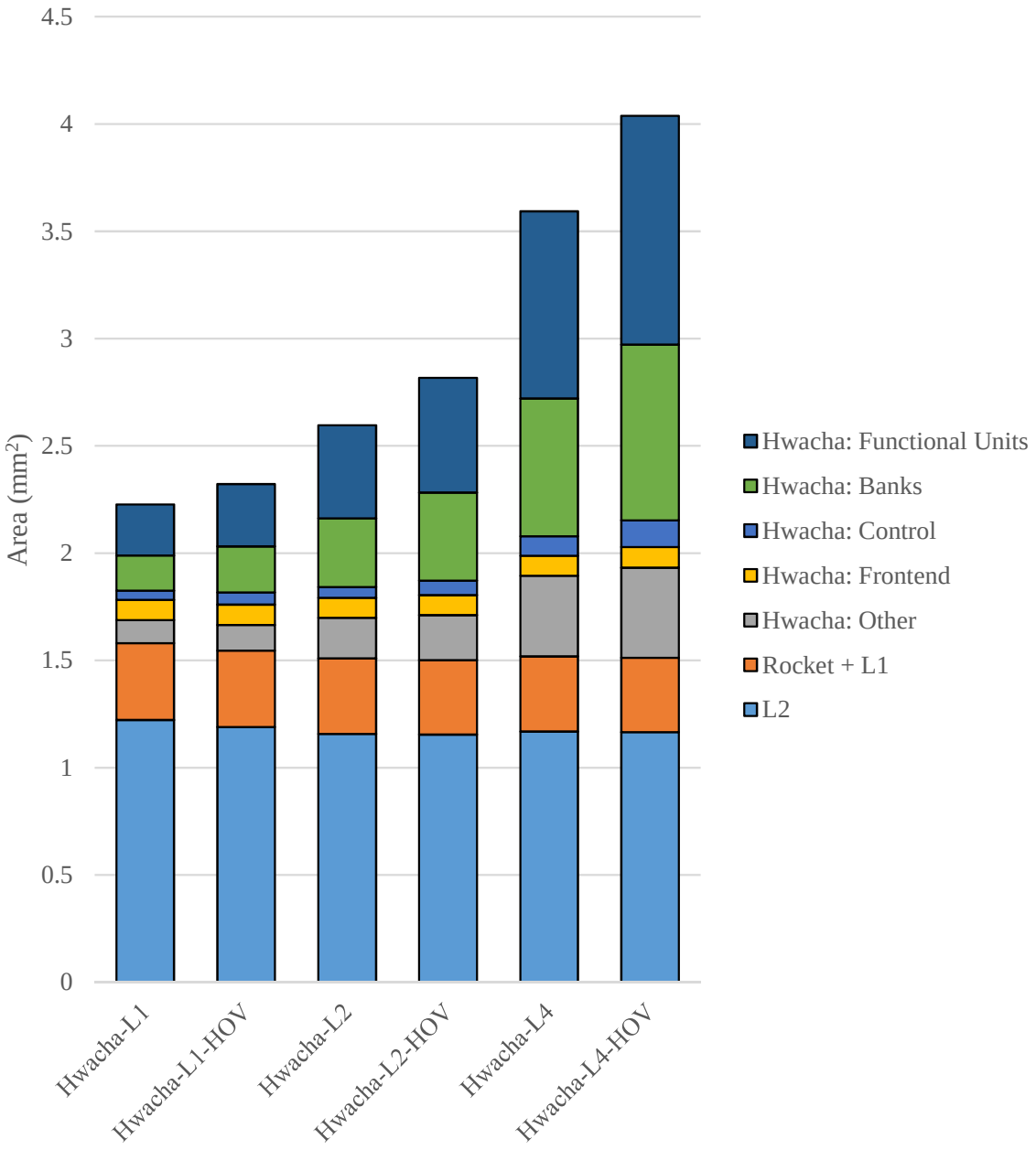


Figure 7.3: Area distribution for the 1/2/4-lane baseline and HOV designs. Data for the 2-lane and 4-lane designs are from synthesis only.

7.3 Performance Comparison

For the set of hand-optimized assembly and OpenCL benchmarks, Figure 7.4 graphs the speedup normalized to baseline Hwacha running the OpenCL versions. Compared to Mali2, Hwacha suffers from a slight disadvantage in functional unit bandwidth from being clocked closer to 1 GHz rather than the ideal 1.2 GHz. Compared to Mali4, Hwacha has less than half the functional unit bandwidth.

For **axpy*, HOV has a marginal effect on performance. As a streaming kernel, it is primarily memory-constrained and therefore benefits little from the higher arithmetic throughput offered by HOV. **axpy* is also the only set of benchmarks in which Mali2 and Mali4 consistently outperforms Hwacha by a factor of 1.5 to 2. This disparity most likely indicates some low-level mismatches in the outer memory systems of Mali and our simulated setup—for example, in the memory access scheduler. We used the default parameters for the memory access scheduler that were shipped with the DRAMSim2 project.

The benefits of HOV become clearer with **gemm* as it is more compute-bound, and more opportunities for in-register data reuse arise. As expected for *dgemm** and *sdgemm**, no difference in performance is observed between the baseline and HOV, since the two designs possess the same number of double-precision FMA units. A modest speedup is seen with *sgemm-unroll*, although still far from the ideal factor of 2 given the single-precision FMA throughput. Curiously, HOV achieves almost no speedup on *sgemm-unroll-opt*. It is possible that the matrices are simply too undersized for the effects to be major. *hgemm** and *hsgemm** demonstrate the most dramatic improvements; however, the speedup is sublinear since, with the quadrupled arithmetic throughput, memory latency becomes more problematic per Amdahl's law.

A significant gap is apparent between the OpenCL and hand-optimized versions of the same benchmarks. The primary reason is that the latter liberally exploits *inter-vector-fetch optimizations* whereby data is retained in the vector register file and reused across vector fetches. It is perhaps a fundamental limitation of the programming model that prevents this behavior from being expressed in OpenCL, resulting in redundant loads and stores at the beginning and end of each vector fetch block.

For all precisions of **gemm*, Mali2 performs surprisingly poorly, by a factor of 3 or 4

slowdown relative to the Hwacha baseline. Mali4 performs about 2× better than Mali2, however, is still slower than the Hwacha baseline. We speculate that the working set is simply unable to fit in cache. Thus, this particular run should not be considered to be entirely fair.

Finally, the baseline and HOV perform about equivalently on **filter*. A slight improvement is discernible for *sfilter* and *mask-filter*, and a more appreciable speedup is evident with *hsfilter* and *mask-hsfilter*. The performance of Mali2 is generally about half that of Hwacha, and Mali4 is on par with Hwacha.

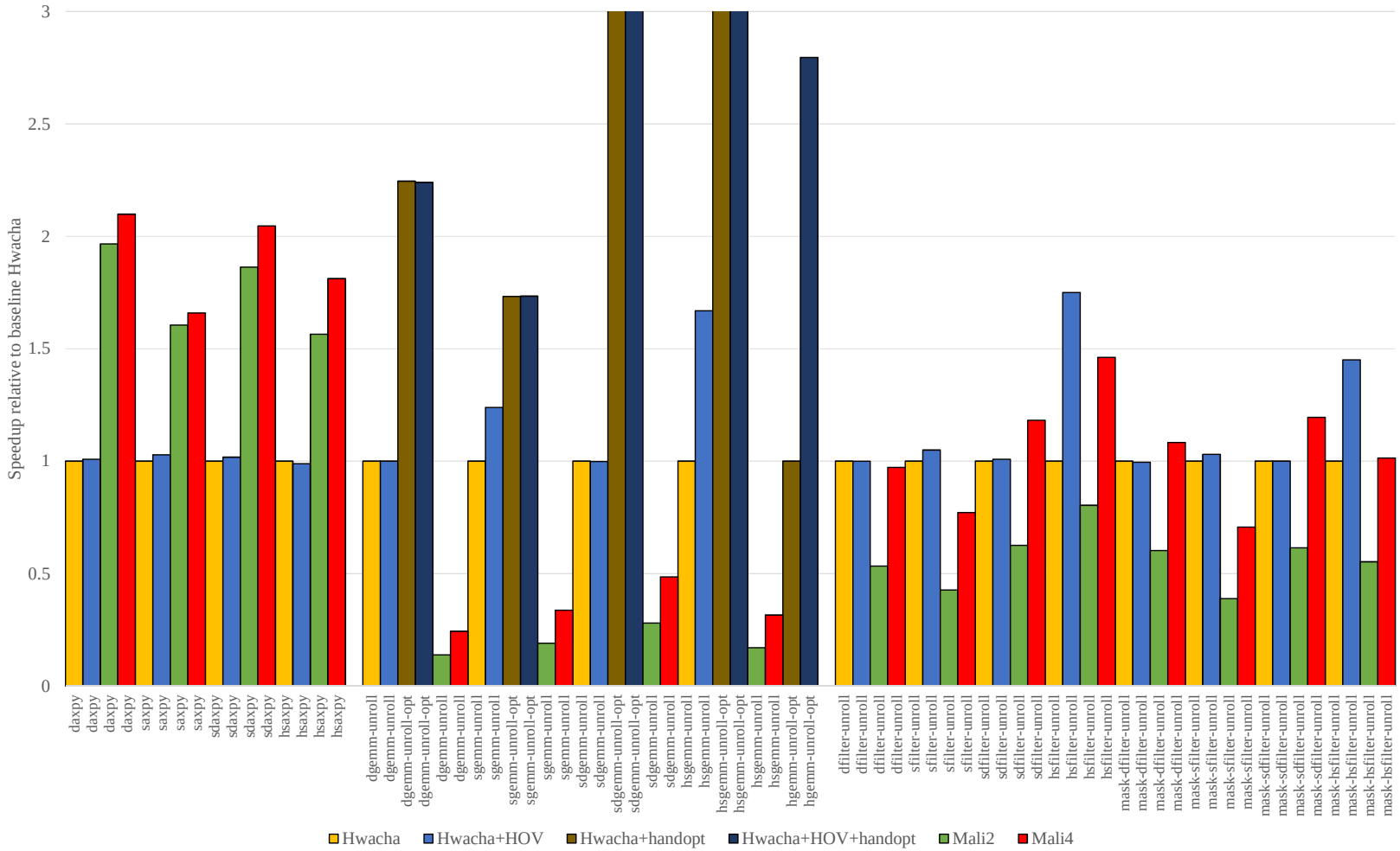


Figure 7.4: Preliminary performance results. (Higher is better.) Due to scale, bars for certain benchmarks have been truncated. *sdgemm-unroll-opt* has speedups 14.0× on the baseline and 13.8× on HOV. *hsgemm-unroll-opt* has speedups 12.0× on the baseline and 19.0× on HOV.

7.4 Energy Comparison

Figure 7.5 graphs the energy consumption for each benchmark, normalized once again to the baseline Hwacha results for the OpenCL versions. Note that the Mali GPU runs on a supply voltage of 0.9 V, whereas we overdrive Hwacha with a 1 V supply to meet a 1 GHz frequency target.

For **axpy*, HOV dissipates slightly more energy than the baseline, with dynamic energy from the functional units comprising most of the difference. In other benchmarks as well, the functional units account for a higher proportion of losses in HOV, which may indicate sub-par effectiveness of clock gating with the extra functional units. Consistent with its performance advantage, Mali2 and Mali4 are twice as energy-efficient on **axpy* than Hwacha.

The results for **gemm* are much more varied. Although HOV is less energy-efficient than the baseline on benchmarks for which it can provide no performance advantage, such as *dgemm*, it is more so on *sgemm*, *hsgemm*, and *hgemm*. These collectively demonstrate a consistent downward trend of increasingly significant reductions in energy consumption as the precision is lowered. Mali data points are again an outlier here, and no conclusion should be drawn.

Energy dissipation on **filter* generally mirrors performance. Overall, HOV is slightly worse than the baseline except on *hsfilter*. Mali similarly retains an advantage as it does with performance, with some exceptions involving reduced-precision computation, i.e., both masked and non-masked versions *sfilter* and *hsfilter*. On these, the energy efficiency of HOV is on par with Mali2 and worse when compared to Mali4.

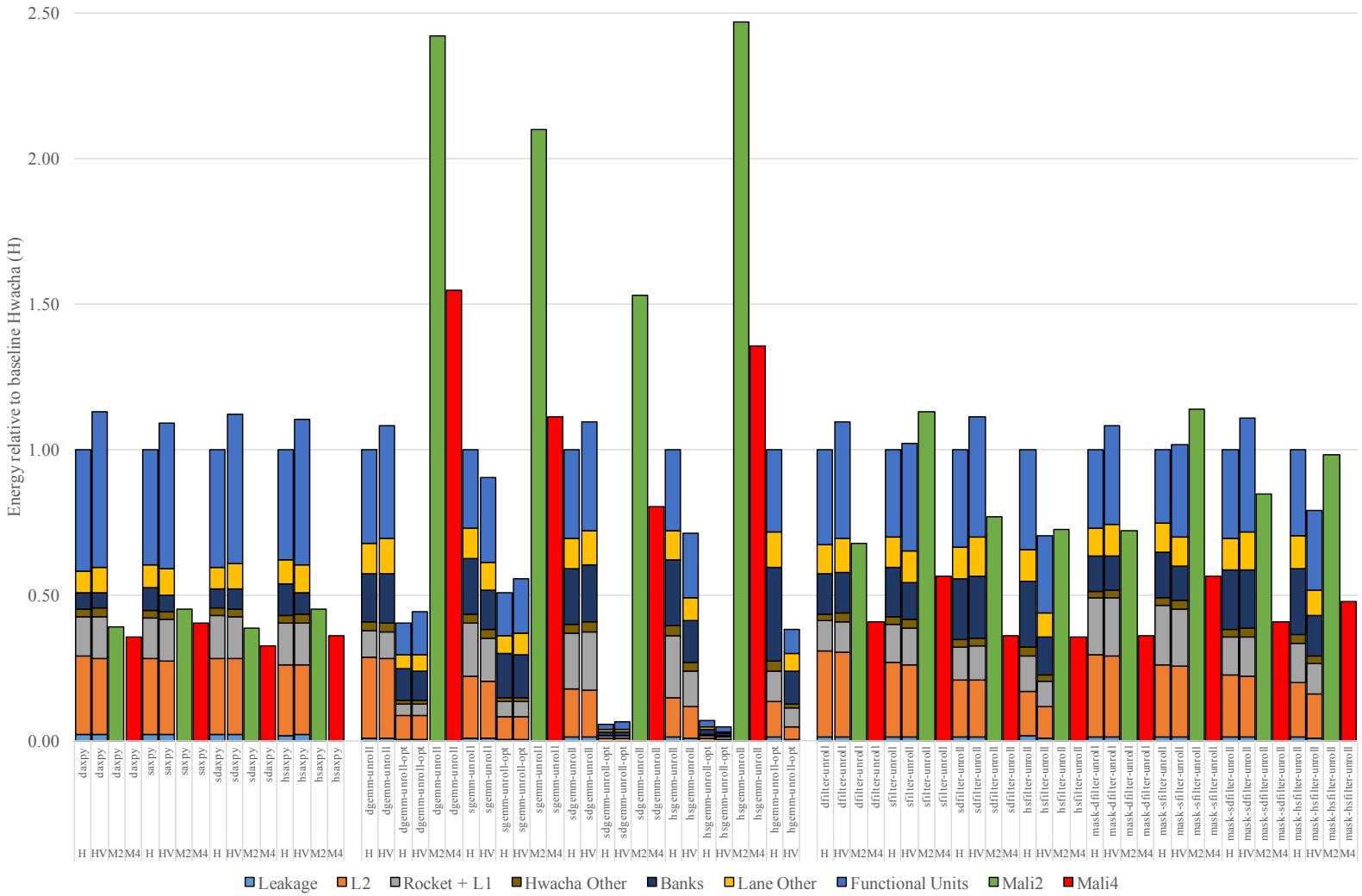


Figure 7.5: Preliminary energy results. (Lower is better.) H = Hwacha baseline, HV = Hwacha+HOV, M2 = Mali2, M4 = Mali4.

Chapter 8

Conclusion

We have presented HOV, a decoupled mixed-precision vector architecture that supports seamless dynamic configuration of multiple architectural register widths and operations on packed data. HOV is an efficient, systematic approach to mixed-precision computation that maintains software compatibility across a wide range of implementation choices. We gauge the performance and energy efficiency gains from these enhancements by executing compiled OpenCL kernels and hand-tuned microbenchmarks on VLSI implementations with and without mixed-precision support. Validation of the VLSI implementations against an ARM Mali GPU demonstrates that our vector architecture is competitive to commercial designs.

Existing data-parallel architectures, including subword SIMD, can support mixed-precision operations, but only by adding new opcodes, packing narrow datatypes inefficiently, or spreading wider types across multiple registers, reducing the effective number of architectural registers. Yet these approaches are suboptimal compared to specifying the precision information with a simple `vsetcfg` instruction.

In our future work, we plan to add widening functional units to avoid extra conversion operations and reduce functional unit energy and latency. We are also interested in exploring tools that automatically determine the appropriate floating-point precision based on values observed at runtime, which would relieve programmers of some of the effort of manually determining precisions for all data.

Bibliography

[1] M. Garland and D. B. Kirk, "Understanding throughput-oriented architectures," *Commun. ACM*, vol. 53, no. 11, pp. 58–66, Nov. 2010. DOI: 10.1145/1839676.1839694.

[2] W. J. Dally, "Throughput computing," in *Proceedings of the 24th ACM International Conference on Supercomputing*, (Tsukuba, Ibaraki, Japan, Jun. 2–4, 2010), T. Boku, H. Nakashima, and A. Mendelson, Eds., New York, NY, USA: ACM, 2010, p. 2. DOI: 10.1145/1810085.1810088.

[3] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning*, (Lille, France, Jul. 6–11, 2015), F. R. Bach and D. M. Blei, Eds., ser. JMLR Proceedings, vol. 37, Journal of Machine Learning Research, 2015, pp. 1737–1746.

[4] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications*, vol. 180, no. 12, pp. 2526–2533, 2009. DOI: 10.1016/j.cpc.2008.11.005.

[5] A. Ou, Q. Nguyen, Y. Lee, and K. Asanović, "MVP: Mixed-precision vector processors," in *2nd International Workshop on Parallelism in Mobile Platforms (PRISM-2), at the International Symposium on Computer Architecture (ISCA-2014)*, (Minneapolis, MN, USA), Jun. 2014.

[6] Y. Lee, C. Schmidt, A. Ou, A. Waterman, and K. Asanović, "The Hwacha vector-fetch architecture manual, version 3.8.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262, Dec. 2015. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-262.html.

[7] Y. Lee, A. Ou, C. Schmidt, S. Karandikar, H. Mao, and K. Asanović, "The Hwacha microarchitecture manual, version 3.8.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-263, Dec. 2015. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-263.html.

[8] Y. Lee, C. Schmidt, S. Karandikar, D. Dabbelt, A. Ou, and K. Asanović, "Hwacha preliminary evaluation results, version 3.8.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-264, Dec. 2015. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-264.html.

[9] R. G. Hintz and D. P. Tate, "Control Data STAR-100 processor design," in *Proceedings of the 6th IEEE Computer Society International Conference*, (San Francisco, CA, USA, Sep. 12–14, 1972), ser. COMPCON, IEEE Computer Society, 1972, pp. 1–4.

[10] K. Asanović, "Vector microprocessors," PhD thesis, University of California, Berkeley, May 1998.

[11] C. Kozyrakis, "Scalable vector media processors for embedded systems," PhD thesis, University of California, Berkeley, May 2002.

[12] W. A. Clark, "The Lincoln TX-2 computer development," in *Proceedings of the Western Joint Computer Conference*, (Los Angeles, CA, USA, Feb. 26–28, 1957), Institute of Radio Engineers, 1957, pp. 153–155.

[13] C. R. Johns and D. A. Brokenshire, "Introduction to the Cell Broadband Engine architecture," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 503–520, 2007. DOI: 10.1147/rd.515.0503.

[14] *Intel 64 and IA-32 architectures software developer's manual volume 1: Basic architecture*, Intel Corporation, Sep. 2015.

[15] *ARM architecture reference manual: ARMv8, for ARMv8-A architectural profile*, ARM Limited, Sep. 2015.

[16] "NVIDIA Tegra X1: NVIDIA's new mobile superchip," Nvidia Corporation, Jan. 2015. [Online]. Available: http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf.

[17] *TMS320C54x DSP functional overview*, SPRU307A, Texas Instruments, May 2000.

[18] *Intel 64 and IA-32 architectures software developer's manual volume 2: Instruction set reference*, Intel Corporation, Sep. 2015.

[19] R. M. Russell, "The Cray-1 computer system," *Commun. ACM*, vol. 21, no. 1, pp. 63–72, 1978. DOI: 10.1145/359327.359336.

[20] C. Batten, "Simplified vector-thread architectures for flexible and efficient data-parallel accelerators," PhD thesis, Massachusetts Institute of Technology, Feb. 2010.

[21] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, p. 6, 2013. DOI: 10.1145/2491464.

[22] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovic, and K. Asanović, "A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in *40th European Solid State Circuits Conference*, (Sep. 22–26, 2014), IEEE, 2014, pp. 199–202. DOI: 10.1109/ESSCIRC.2014.6942056.

[23] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *The 49th Annual Design Automation Conference 2012*, (San Francisco, CA, USA, Jun. 3–7, 2012), P. Groeneveld, D. Sciuto, and S. Hassoun, Eds., ACM, 2012, pp. 1216–1225. DOI: 10.1145/2228360.2228584.

[24] H. M. Cook, A. S. Waterman, and Y. Lee. (2015). TileLink cache coherence protocol implementation, [Online]. Available: https://github.com/ucb-bar/uncore.

[25] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011. DOI: 10.1109/L-CA.2011.4.

[26] J. E. Smith, "Decoupled access/execute computer architectures," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 289–308, 1984. DOI: 10.1145/357401.357403.

[27] R. Espasa and M. Valero, "A simulation study of decoupled vector architectures," *The Journal of Supercomputing*, vol. 14, no. 2, pp. 124–152, 1999. DOI: 10.1023/A:1008158808410.

[28] C. Batten, R. Krashinsky, S. Gerding, and K. Asanović, "Cache refill/access decoupling for vector machines," in *37th Annual International Symposium on Microarchitecture*, (Portland, OR, USA, Dec. 4–8, 2004), IEEE Computer Society, 2004, pp. 331–342. DOI: 10.1109/MICRO.2004.9.

[29] J. E. Smith, G. Faanes, and R. A. Sugumar, "Vector instruction set support for conditional operations," in *27th International Symposium on Computer Architecture*, (Vancouver, BC, Canada, Jun. 10–14, 2000), A. D. Berenbaum and J. S. Emer, Eds., IEEE Computer Society, 2000, pp. 260–269. DOI: 10.1109/ISCA.2000.854396.

[30] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "Pocl: A performance-portable OpenCL implementation," *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015. DOI: 10.1007/s10766-014-0320-y.

[31] C. Lattner and V. S. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization*, (San Jose, CA, USA, Mar. 20–24, 2004), IEEE Computer Society, 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665.

[32] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. W. Hwu, "Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs," in *Proceedings of the 8th International Symposium on Code Generation and Optimization*, (Toronto, Ontario, Canada, Apr. 24–28, 2010), A. Moshovos, J. G. Steffan, K. M. Hazelwood, and D. R. Kaeli, Eds., ACM, 2010, pp. 111–119. DOI: 10.1145/1772954.1772971.

[33] S. Collange, "Identifying scalar behavior in CUDA kernels," Université de Lyon, Tech. Rep. hal-00555134, Jan. 2011.

[34] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanović, "Convergence and scalarization for data-parallel architectures," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, (Shenzhen, China, Feb. 23–27, 2013), IEEE Computer Society, 2013, 32:1–32:11. DOI: 10.1109/CGO.2013.6494995.

[35] R. Karrenberg and S. Hack, "Whole-function vectorization," in *Proceedings of the 9th International Symposium on Code Generation and Optimization*, (Chamonix, France, Apr. 2–6, 2011), IEEE Computer Society, 2011, pp. 141–150. DOI: 10.1109/CGO.2011.5764682.

[36] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. M. Jr., "Divergence analysis and optimizations," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, (Galveston, TX, USA, Oct. 10–14, 2011), L. Rauchwerger and V. Sarkar, Eds., IEEE Computer Society, 2011, pp. 320–329. DOI: 10.1109/PACT.2011.63.

[37] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, and K. Asanović, "Exploring the design space of SPMD divergence management on data-parallel architectures," in *47th Annual IEEE/ACM International Symposium on Microarchitecture*, (Cambridge, United Kingdom, Dec. 13–17, 2014), IEEE, 2014, pp. 101–113. DOI: 10.1109/MICRO.2014.48.

[38] "big.LITTLE technology: The future of mobile," ARM Limited, 2013. [Online]. Available: https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf.

[39] *ODROID-XU3 block diagram*, Hardkernel Co., Ltd. [Online]. Available: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127&tab_idx=2.

[40] R. Smith. (Jul. 3, 2014). ARM's Mali Midgard architecture explored, AnandTech, [Online]. Available: http://www.anandtech.com/show/8234/arms-mali-midgard-architecture-explored.

[41] P. Galatsopoulos and M. Björge. (Oct. 28, 2014). Midgard GPU architecture, ARM Limited, [Online]. Available: http://malideveloper.arm.com/downloads/ARM_Game_Developer_Days/PDFs/2-Mali-GPU-architecture-overview-and-tile-local-storage.pdf.

[42] C. Celio. (2012). Characterizing multi-core processors using micro-benchmarks, [Online]. Available: https://github.com/ucb-bar/ccbench/wiki.

[43] *ARM Cortex-A15 MPCore processor technical reference manual*, ARM Limited, Jun. 24, 2013.