Hwacha Preliminary Evaluation Results, Version 3.8.1

Yunsup Lee
Colin Schmidt
Sagar Karandikar
Daniel Dabbelt
Albert Ou
Krste Asanović

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-264
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-264.html

December 19, 2015

Copyright © 2015, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Hwacha Preliminary Evaluation Results
Version 3.8.1

Yunsup Lee, Colin Schmidt, Sagar Karandikar, Palmer Dabbelt, Albert Ou, Krste Asanović
CS Division, EECS Department, University of California, Berkeley
{yunsup|colins|sagark|palmer.dabbelt|aou|krste}@eecs.berkeley.edu

December 19, 2015

Contents

1 Introduction **3**

2 Evaluation Framework **4**

2.1 Microbenchmarks 5
2.2 OpenCL Compiler 6
2.3 Samsung Exynos 5422 and the ARM Mali-T628 MP6 GPU 6
2.4 RTL Development and VLSI Flow 8

3 Preliminary Evaluation Results **9**

3.1 Memory System Validation 9
3.2 Area and Cycle Time 10
3.3 Performance Comparison 13
3.4 Energy Comparison 15

4 History **17**

4.1 Funding 17

References **19**

1 Introduction

This work-in-progress document presents preliminary Hwacha evaluation results. We first discuss the details of our evaluation framework. Using this framework, we compare a baseline Hwacha design to a Hwacha design with mixed-precision support to see how it affects performance, power/energy, and area. For the discussion, we name the Hwacha machine with mixed-precision support, *high-occupancy vector lanes*, or HOV. Obviously, this is not the only experiment we can run (consult the Hwacha microarchitecture manual for Hwacha parameters that we can change), however, it serves as a good example for the purpose of this document. We also validate the Hwacha design against the ARM Mali-T628 MP6 GPU by running a suite of microbenchmarks compiled from the same OpenCL source code using our custom LLVM-based scalarizing compiler and the ARM stock compiler.

2 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 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 2.1), and developed our own LLVM-based scalarizing compiler that can generate code for the Hwacha vector-fetch assembly programming model (see Section 2.2). 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 2.3 for more details). We synthesize and place-and-route both Hwacha designs (the baseline and HOV) in a commercial 28 nm process resembling the 28 nm high-κ metal gate (HKMG) process used to fabricate the Exynos 5422, and run the compiled microbenchmarks

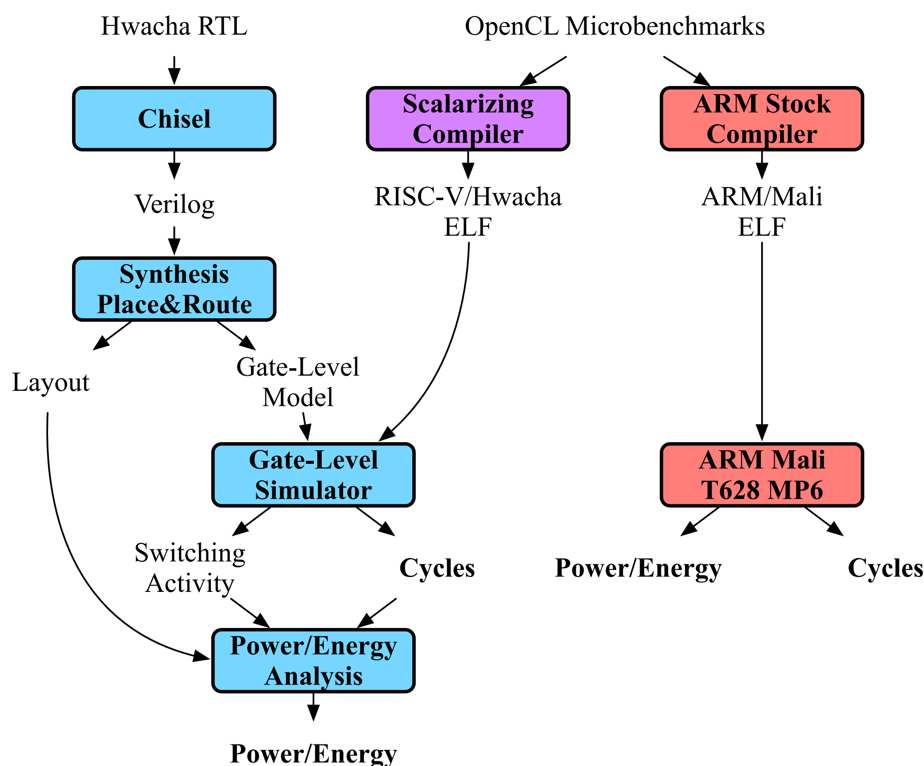


Figure 1: Hwacha Evaluation Framework.

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 1: A listing of the microbenchmarks used.

on both gate-level simulators to obtain accurate switching activities and performance numbers (see Section 2.4 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 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.

2.1 Microbenchmarks

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

2.2 OpenCL Compiler

We developed an OpenCL compiler based on the PoCL OpenCL runtime [10] and a custom LLVM [12] backend. The main challenges in generating Hwacha vector code from OpenCL kernels are moving thread-invariant values into scalar registers [17, 8, 14], identifying stylized memory access patterns, and using predication effectively [11, 9, 13]. Thread-invariance is determined using the variance analysis presented in [14], 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 [13].

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.

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

Figure 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 [3, 2]. Table 2 presents the specific Rocket Chip SoC generator parameters we chose 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, named *Mali2*, and again on the four shader core device, named *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

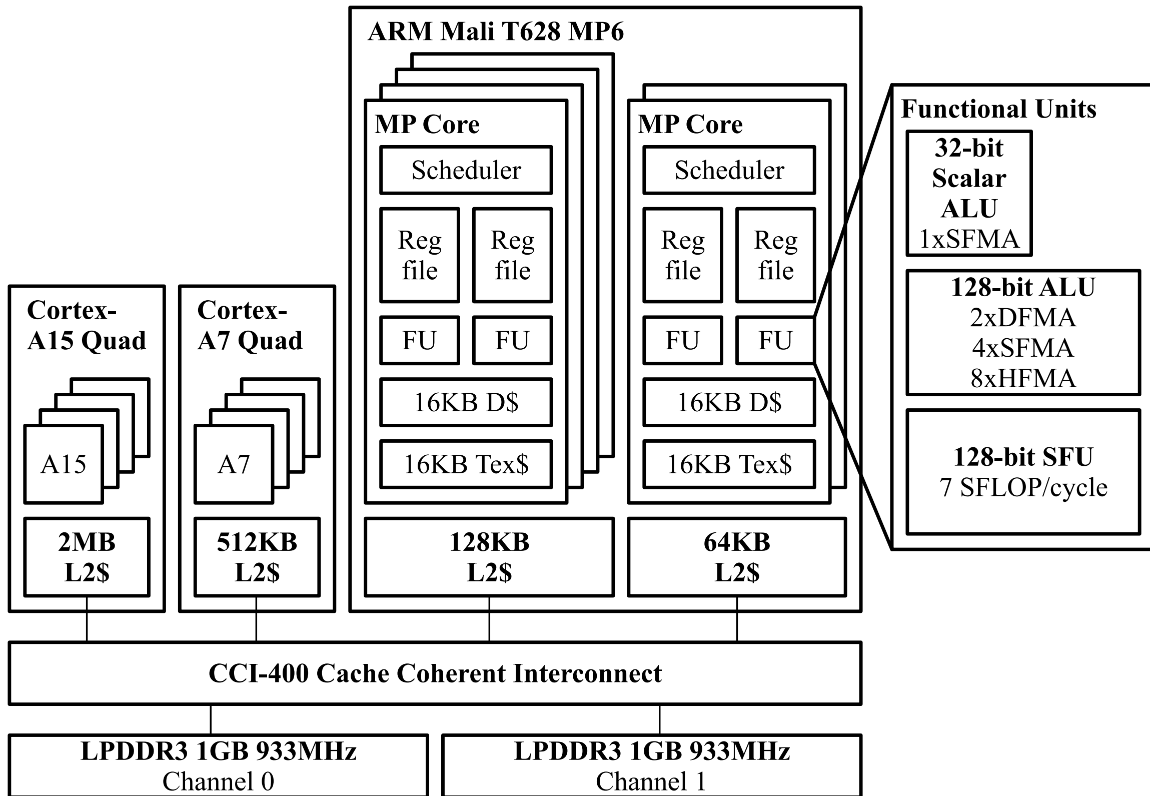


Figure 2: Exynos 5422 Block Diagram.

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

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 2: Used Rocket Chip SoC generator parameters.

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, [1] and [4] 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, 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. The Hwacha 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.

2.4 RTL Development and VLSI Flow

The Hwacha RTL is written in Chisel [6], 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).

3 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.

3.1 Memory System Validation

We configure DRAMSim2 [16] 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 [7] contains a variety of benchmarks to characterize multi-core systems. We use `ccbench`'s `caches` benchmark, which performs a pointer chase to measure latencies of each level of the memory hierarchy.

Figure 3 compares the performance of our cycle-accurate simulated memory hierarchy against the Exynos 5422. On the simulated RISC-V Rocket core, `ccbench` measures cycles, normalized to nanoseconds by assuming the 1 GHz clock frequency attained by previous silicon implementations of Rocket [15]. On the Exynos 5422, `ccbench` measures wall-clock time.

This comparison reveals that our simulated system and the Exynos 5422 match closely in terms

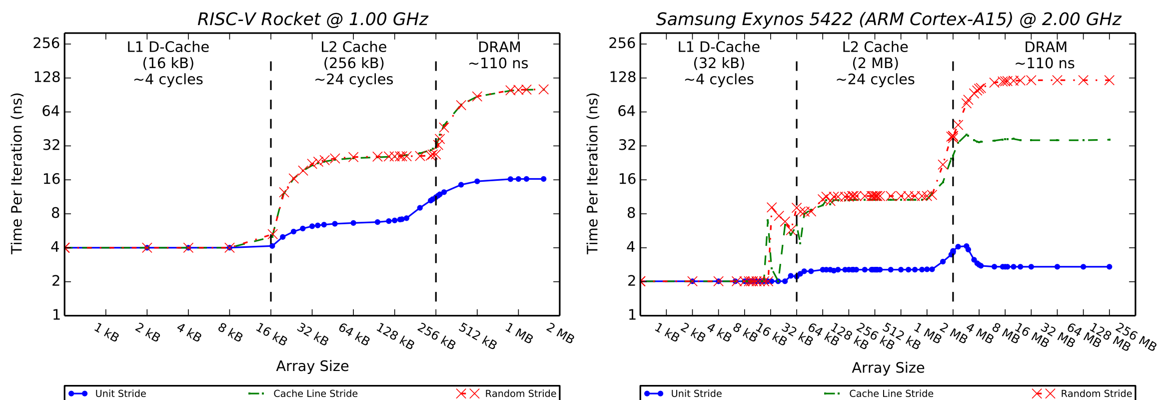


Figure 3: `ccbench` "`caches`" Memory System Benchmark.

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/stores [5].

3.2 Area and Cycle Time

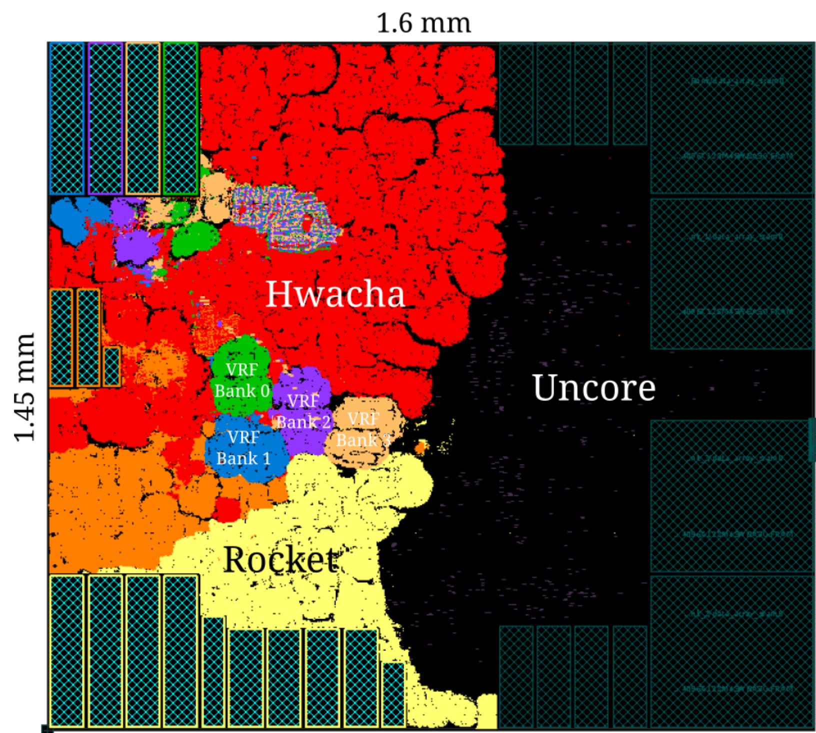
Table 3 lists the clock frequencies and total area numbers obtained for a variety of Hwacha configurations. Figure 4 shows the layout for the single-lane HOV design.

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, they are still generally above 1 GHz. However, the aggressive physical design does involve a trade-off in area.

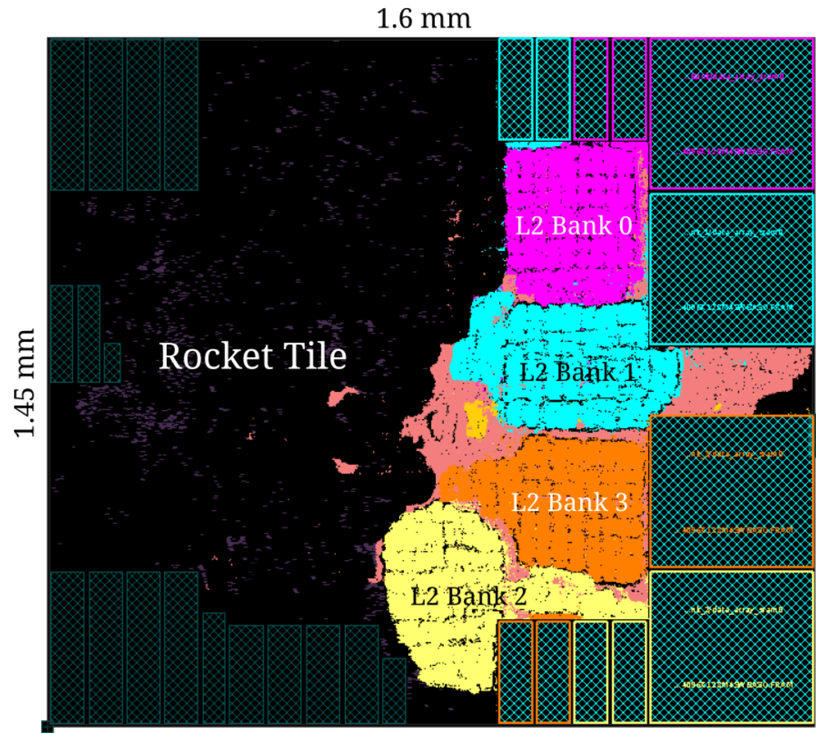
As seen in Figure 5, the area overhead for the HOV extensions spans from 4.0% in the single-lane design to 12.5% in the four-lane design. The additional functional units account for a large portion of the increase. The banks also become somewhat larger from the widening of the predicate register file, as does the control due to the non-trivial amount of comparators needed to implement the expander hazard check in the sequencers.

	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?		✓				✓		

Table 3: VLSI quality of results. Columns not marked as “PAR” are results from synthesis.



(a) Tile: Rocket and Hwacha



(b) Uncore: L2 cache and interconnect

Figure 4: Layout of single-lane Hwacha design with HOV.

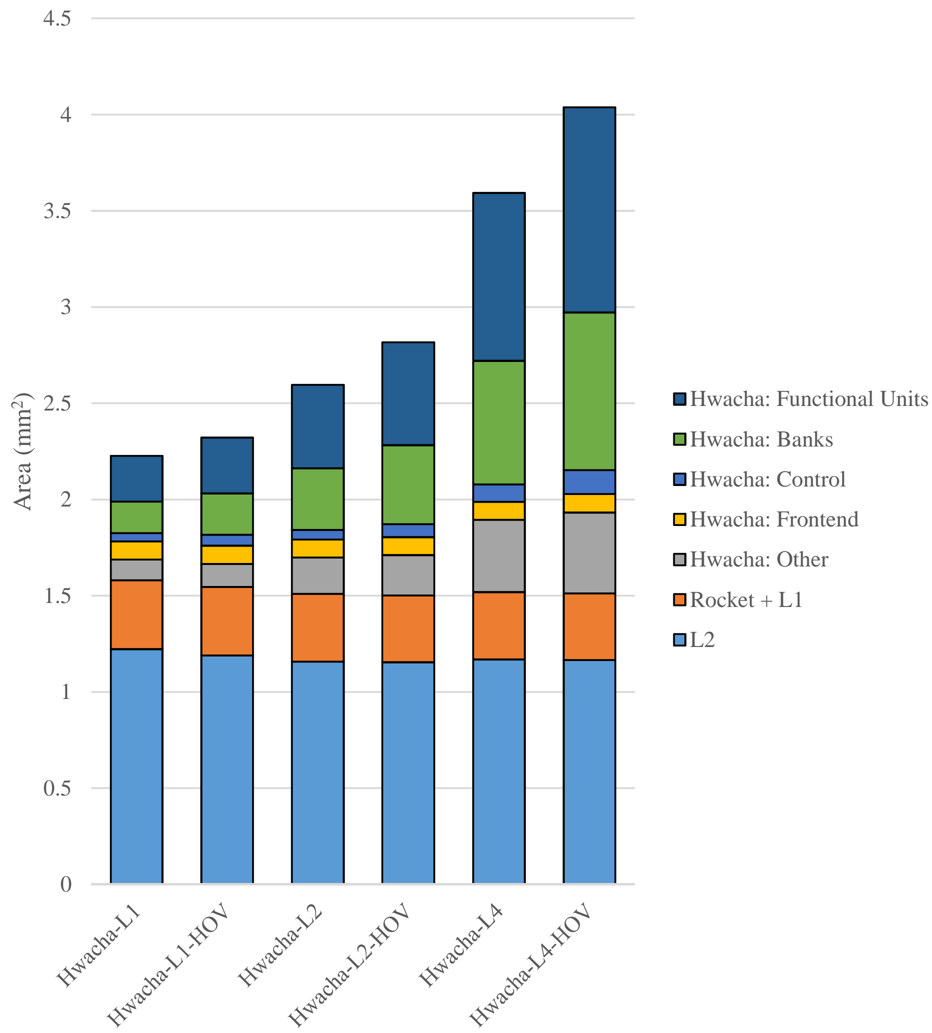


Figure 5: Area distribution for the 1/2/4-lane baseline and HOV designs. Data for the 2-lane and 4-lane designs are from synthesis only.

3.3 Performance Comparison

For the set of hand-optimized assembly and OpenCL benchmarks, Figure 6 graphs the speedup normalized to baseline Hwacha running the OpenCL version. 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.

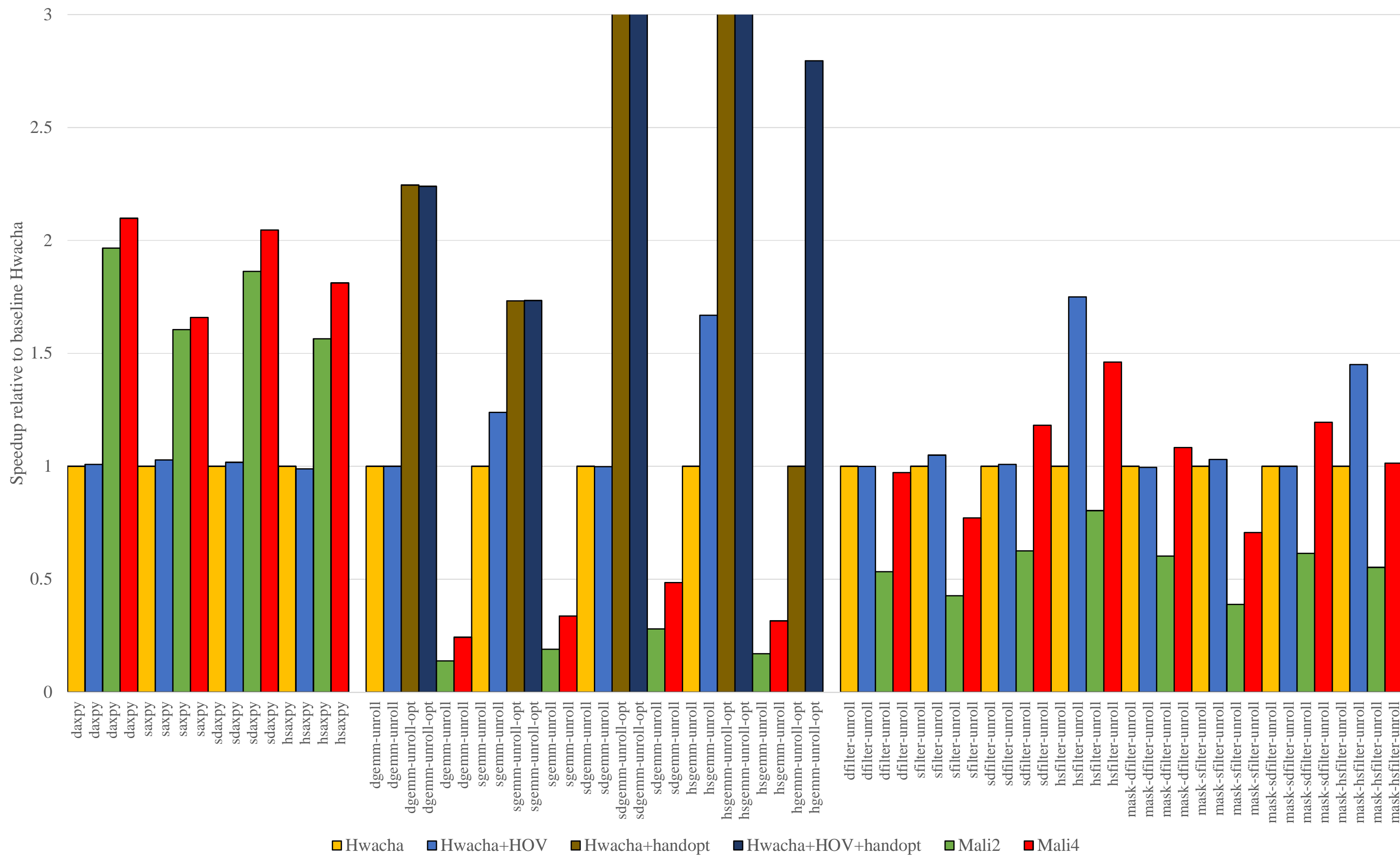


Figure 6: Preliminary performance results. (Higher is better.) Due to scale, bars for certain benchmarks have been truncated. *sdgemm-unroll-opt* has speedups 14.0× on the baseline and 13.8× on HOV. *hsgemm-unroll-opt* has speedups 12.0× on the baseline and 19.0× on HOV.

3.4 Energy Comparison

Figure 7 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.

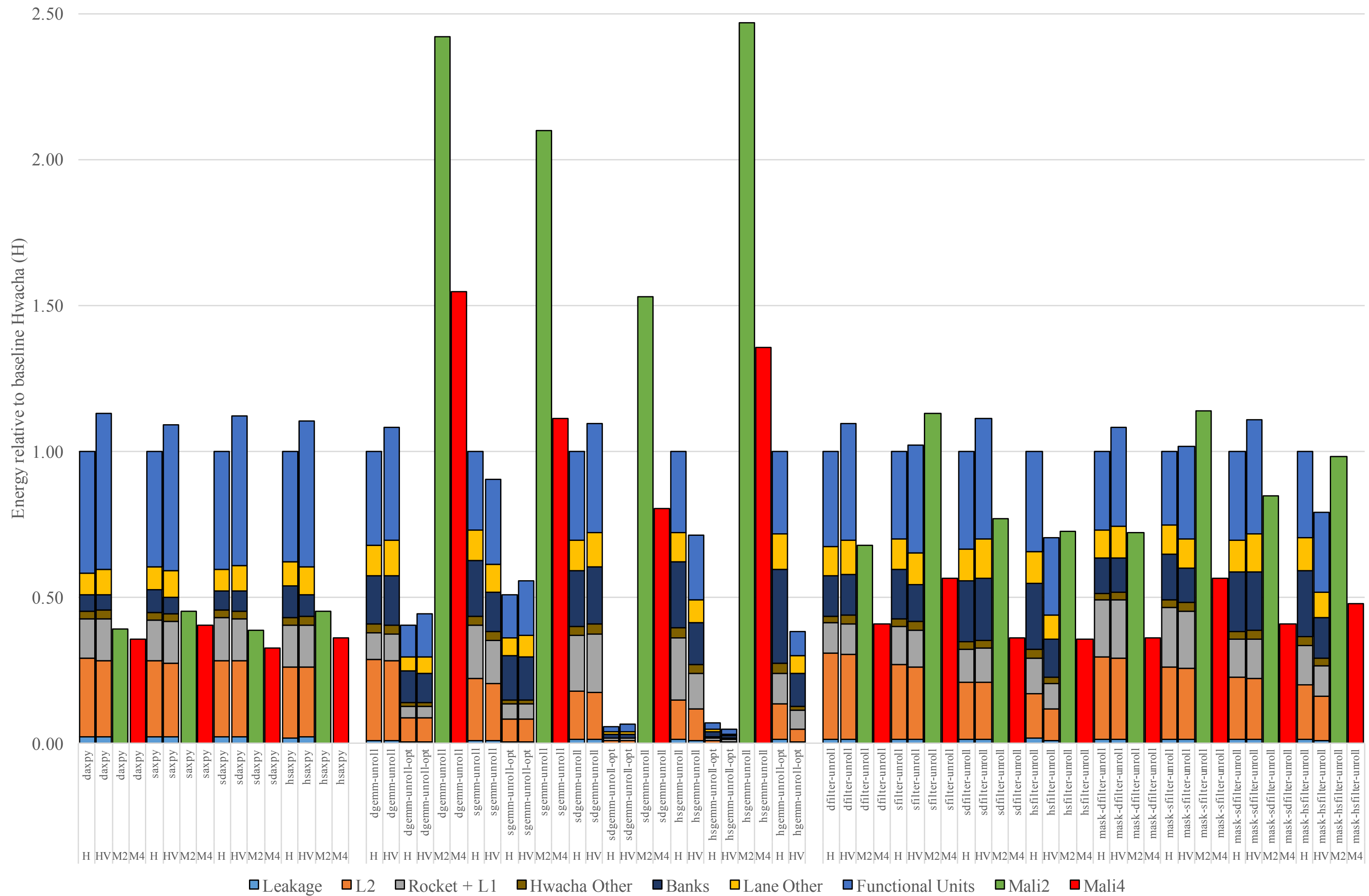


Figure 7: Preliminary energy results. (Lower is better.) H = Hwacha baseline, HV = Hwacha+HOV, M2 = Mali2, M4 = Mali4.

4 History

The detailed project history is described in the history section of the Hwacha vector-fetch architecture manual.

4.1 Funding

The Hwacha project has been partially funded by the following sponsors.

- **Par Lab**: Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support came from Par Lab affiliates: Nokia, NVIDIA, Oracle, and Samsung.
- **Silicon Photonics**: DARPA POEM program, Award HR0011-11-C-0100.
- **ASPIRE Lab**: DARPA PERFECT program, Award HR0011-12-2-0016. The Center for Future Architectures Research (C-FAR), a STARnet center funded by the Semiconductor Research Corporation. Additional support came from ASPIRE Lab industrial sponsors and affiliates: Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung.
- **NVIDIA graduate fellowship**

Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

References

[1] ARM's Mali Midgard Architecture Explored. http://www.anandtech.com/show/8234/arms-mali-midgard-architecture-explored.

[2] Samsung Exynos 5422 SoC Block Diagram. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127&tab_idx=2.

[3] big.LITTLE Technology: The Future of Mobile. https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf, 2013.

[4] Midgard GPU Architecture. http://malideveloper.arm.com/downloads/ARM_Game_Developer_Days/PDFs/2-Mali-GPU-architecture-overview-and-tile-local-storage.pdf, Oct 2014.

[5] ARM Holdings. *ARM Cortex-A15 MPCore Processor, Technical Reference Manual*, Jun 2013.

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[7] C. Celio. Characterizing Multi-Core Processors Using Micro-benchmarks. https://github.com/ucb-bar/ccbench/wiki, 2012. Github Wiki.

[8] S. Collange. Identifying Scalar Behavior in CUDA Kernels. Technical Report hal-00555134, Université de Lyon, January 2011.

[9] B. Coutinho, D. Sampaio, F. Pereira, and W. Meira. Divergence Analysis and Optimizations. In *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 320–329, October 2011.

[10] 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*, 2014.

[11] R. Karrenberg and S. Hack. Whole-function Vectorization. In *Int'l Symp. on Code Generation and Optimization (CGO)*, pages 141–150, April 2011.

[12] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program lysis and Transformation. In *Int'l Symp. on Code Generation and Optimization (CGO)*, pages 75–88, San Jose, CA, USA, Mar 2004.

[13] 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 *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 101–113, Washington, DC, USA, 2014. IEEE Computer Society.

[14] Y. Lee, R. Krashinsky, V. Grover, S. Keckler, and K. Asanovic. Convergence and scalarization for data-parallel architectures. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–11, Feb 2013.

[15] Y. Lee, A. Waterman, R. Avižienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović. A 45nm 1.3GHz 16.7 Double-Precision GFLOPS/W RISC-V Processor with Vector Accelerators. In *2014 European Solid-State Circuits Conference (ESSCIRC-2014)*, Venice, Italy, Sep 2014.

[16] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters*, 10(1):16–19, Jan 2011.

[17] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. mei W. Hwu. Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs. In *Int'l Symp. on Code Generation and Optimization (CGO)*, pages 111–119, April 2010.