A Case for MVPs: Mixed-Precision Vector Processors

Albert Ou, Quan Nguyen, Yunsup Lee, Krste Asanović
University of California, Berkeley
{aou,quannguyen,yunsup,krste}@eecs.berkeley.edu

Abstract

Mixed-precision computation presents opportunities for programmable accelerators to improve performance and energy efficiency while retaining application flexibility. We augment Hwacha, a decoupled vector-fetch data-parallel accelerator, to support dynamic configuration of architectural register widths and operations on packed data. We discuss the implications on the programming model and the microarchitectural features for maximizing register file utilization and datapath parallelism. A complete VLSI implementation quantifies the impact on area, performance, and energy consumption.

1. Introduction

To accommodate increasingly sophisticated functionality under stringent power constraints, computing platforms must meet simultaneous demands for greater efficiency and flexibility. Compared to fixed-function hardware, programmable processors contend with inefficiencies from at least two major sources: wasteful power consumption by over-provisioned datapaths and overhead of instruction delivery. We propose seamless mixed-precision computation on a vector architecture to address both aspects.

Many applications exhibit a broad variation in data value widths. Fixed-function accelerators attain significantly better energy efficiency and performance by exploiting minimal and heterogeneous word widths in their custom datapaths. By contrast, processors must support a range of conventional datatype widths to fulfill a general-purpose role. Thus, the datapath is typically fixed at the maximum precision potentially employed by any application. For certain applications (e.g., multimedia and signal processing) that do not require the highest available precision, unnecessary energy is expended obtaining an equivalent result.

Rarely is one global precision optimal throughout all stages of computation; for example, the widths of addresses and integer data often differ, and in widening arithmetic operations such as a fused multiply-add (FMA), the product of n-bit values is added to a $2n$-bit accumulator without intermediate rounding. For versatility, a processor should be able to simultaneously intermix several reduced-precision modes according to application-specific conditions.

Data-level parallelism (DLP) is the most efficient form of parallelism with respect to control overhead. Vector processors perform multiple homogeneous and independent operations with each concise instruction, thereby amortizing the cost of instruction fetch, decode, and sequencing. The inherent regularity and parallelism of vector operations promote extensive scalability in lane organization to meet varying performance/power targets. Vector data accesses also adhere to highly structured and predictable patterns, allowing for microarchitectural optimizations such as prefetching and register file banking [9] to reduce port count.

Vector architectures are readily adaptable to reduced-precision computation. Due to the intrinsic data independence, a wide datapath can be naturally partitioned into multiple narrower elements. Compaction improves the utilization of register file accesses and interconnection fabric for operand communication, and additional functional units can be introduced to leverage subword parallelism with relatively small area cost. Most importantly, increased throughput enables faster race-to-halt into a low power state.

Moreover, denser storage of elements lessens memory pressure and allows for longer vectors with the same register file capacity. The expanded buffering assists with decoupled execution in more effectively hiding memory latency.

Section 2 reviews existing approaches and their disadvantages. In response, section 3 presents a more refined solution for mixed-precision vector processing. Section 4 describes the baseline accelerator design, and section 5 elaborates on the microarchitectural changes for mixed precision. Section 6 details a preliminary evaluation based on a VLSI implementation.

2. Related Work

Commercial ISAs, including x86 and ARM, popularly feature instruction set extensions for SIMD (AVX [7] and NEON [1], respectively). While also capable of packed subword computations, these share certain architectural shortcomings that render them inconvenient for a mixed-precision environment.

- *Opcodes designate a fixed vector length.* Exposing the hardware SIMD width in the ISA hinders portability. Code must be recompiled to benefit from longer SIMD widths, and migration becomes yet more tedious if explicit intrinsics are used instead of an auto-vectorizing compiler. Such incompatibilities necessitate CPU dispatch techniques to select between different versions of code at runtime. Software is also burdened by the trailing edge case when the application vector length is not a multiple of the SIMD width, a situation further complicated by the lack of general ISA support for masking inactive elements.
- *All vector registers are uniformly fixed in size.* Consequently, the number of elements per vector register varies at different

precisions, and it is not so straightforward to chain mixed-precision operations. Equalizing vector lengths might involve splitting a vector across several architectural registers, depleting register encoding space.

- *The vector length is typically short, currently at most 256 bits.* These SIMD extensions, electing for the path of least resistance towards subword parallelism, re-use existing scalar datapaths and control. Superscalar complexity therefore places a practical upper bound on the SIMD width. Implementations rely nearly exclusively on spatial execution of vector operations, forcing the use of superscalar issue mechanisms to saturate functional units and hide long operational latencies. With shorter hardware vectors, more stripmine iterations are required to cover the application vector length. This redundancy counteracts the instruction bandwidth efficiency of SIMD.

Some GPUs and microcontrollers implement limited support for variable precision by storing wider datatypes across several narrower architectural registers. For example, double-precision floating-point values might occupy pairs of 32-bit registers and be referenced solely through even register specifiers, effectively halving the available register set.

Asanović [2] and Kozyrakis [8] each describe how a vector machine can be treated as an array of "virtual processors" whose datapath widths are set collectively through a virtual processor width (VPW) register. However, the precision of an individual vector cannot be configured independently.

3. Mixed-Precision Vector Processors

Traditional vector machines, descended from the Cray archetype [15], offer a cleaner abstraction than SIMD extensions. With modifications, mixed floating-point and fixed-point precisions can be expressed with minimal complexity.

Current work extends the architecture to allow fine-grained runtime configuration of the vector register set, such that the number of architectural registers can be specified as well as their individual datatype widths. The hardware automatically determines the appropriate register file mapping, subdivides physical registers as needed to hold multiple elements, and adjusts the hardware vector length to either the maximum possible or a software-requested value if shorter.

Hardware management of register allocation and vector lengths lends several advantages in terms of flexibility. Programs can exchange unused architectural registers for longer hardware vectors. Furthermore, subword packing is effectively transparent to the application. An implementation lacking full mixed-precision support can ignore configuration hints and still execute the same code, albeit at reduced efficiency. An extremely broad spectrum of possible designs is therefore feasible without compromising portability, crucial given the proliferating diversity of mobile platforms.

Hwacha is a decoupled vector-fetch data-parallel accelerator [11] which serves as the vehicle for our architectural exploration. Its microarchitecture and programming paradigm

```
1  start:
2    vsetcfg   2, 1, 1, 2  # Configure 2 integer, 1 double,
3                          # 1 single, and 2 half registers
4    la        x1, vtcode  # Load vector-fetch address
5    li        x2, vlen    # Load application vector length
6
7  loop:
8    vsetvl    x3, x2      # Set hardware vector length
9    vfld      vf0, (x4)   # Load vector from pointer x4
10   vfls      vf1, (x5)   # Load vector from pointer x5
11   vf        (x2)        # Execute vector-fetch block
12   vsd       vx1, (x6)   # Store vector to pointer x6
13                         # Pointer increment omitted
14   sub       x2, x2, x3  # Update application vector length
15   bnez      x2, loop    # Loop if more elements remain
16   j         exit
17
18 vtcode:
19   fcvt.h.d  f2, f0      # Convert vf0 (double) to half
20   fcvt.h.s  f3, f1      # Convert vf1 (single) to half
21   fadd.h    f3, f3, f2  # FP add vf2 (half) and vf3 (half)
22   fcvt.l.h  x1, f3      # Convert vf3 (half) to integer
23   stop                  # End vector-fetch block
```

Figure 1: Example mixed-precision assembly code

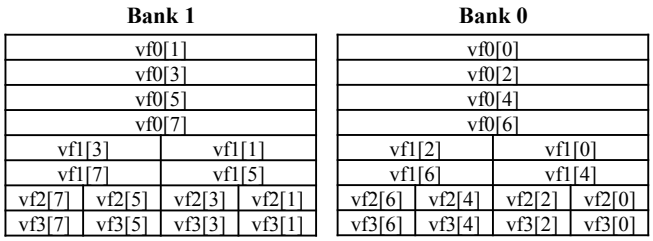

Figure 2: Resultant vector register file mapping. The numbers in brackets identify the element's index in the vector.

combines facets of predecessor vector-threaded architectures, such as Maven [4, 10], with concepts from traditional vector machines—in particular, divergence handling mechanisms in software.

An asynchronous *control thread* dispatches *traditional vector* commands to manage and configure a set of *microthreads* (μT), as well as to initiate vector memory and broadcast operations. To assign the microthreads a workload, the control thread sends a *vector-fetch* (`vf`) command indicating the initial PC of a separate instruction stream. From a logical perspective, vector operations appear as an array of microthreads, each delegated an individual element, executing the same scalar instructions of the vector-fetch code in lockstep. Hwacha supports the full complement of RISC-V integer, floating-point, load/store, and atomic memory instructions [17].

The number of resident microthreads represents the hardware vector length, which the control thread may detect through the `vsetvl` command. To handle application vectors of arbitrary length, a common pattern is to fragment processing over multiple vector-fetch sessions ("stripmining").

Microthreads may be uniformly distributed up to 32 integer and 32 floating-point registers each via the `vsetcfg` command. Each scalar register, as viewed from a microthread, correlates with one element of the vector register as viewed from the control thread.

Figure 1 presents an excerpt of mixed-precision assembly code, for which Figure 2 depicts an example mapping of vector registers to two banks of eight 64-bit physical registers. By explicitly configuring the vector unit with `vsetcfg`, the program interacts with `vf0`, `vf1`, `vf2`, and `vf3` as vectors of double-, single-, and half-precision floating-point registers, respectively. The program also has access to two vector integer registers, `vx0` and `vx1` (not shown), the former of which is hardwired to zero. Register `vf1` fits two elements per bank entry, whereas `vf2` and `vf3` fit four. Note that this packing arrangement yields a hardware vector length of eight.

4. Baseline Implementation

The Hwacha accelerator interfaces with the Rocket processor, a six-stage in-order RISC-V scalar core [11] onto which the control thread is mapped. Figure 3 illustrates a system-level block diagram. Hwacha possesses a 8 KiB direct-mapped L1 instruction cache, and Rocket possesses a 16 KiB 2-way set-associative L1 instruction cache. Both share a 32 KiB 4-way set-associative L1 data cache. A unified 256 KiB 8-way set-associative L2 cache backs the primary caches. DRAMSim2 provides the DDR3 timing model [13].

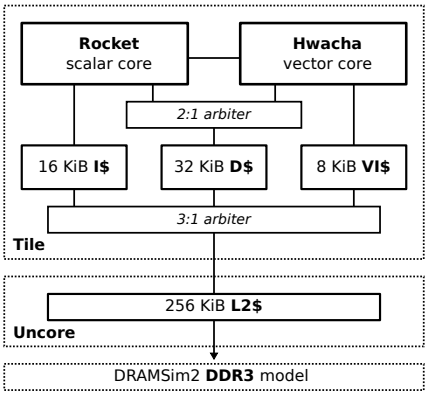

Figure 3: System integration

Figure 4 presents a general block diagram of the accelerator. Hwacha is internally decoupled into two components: the single-lane *Vector Execution Unit* (VXU), which encompasses the register file and the functional units, and the *Vector Memory Unit* (VMU), which coordinates data movement between the VXU and the memory system.

4.1. Vector Execution Unit

Traditional vector instructions arrive from the control processor through the Vector Command Queue (VCMDQ). Upon encountering a `vf` command, the vector-fetch frontend begins instruction fetch at the PC provided with the VCMDQ entry and proceeds until a `stop` instruction is reached.

The fetched instructions are decoded by the issue unit and then wait in the hazard-checking unit until all structural hazards pertaining to the occupancy of sequencer slots, bank read/write ports, and functional units are resolved.

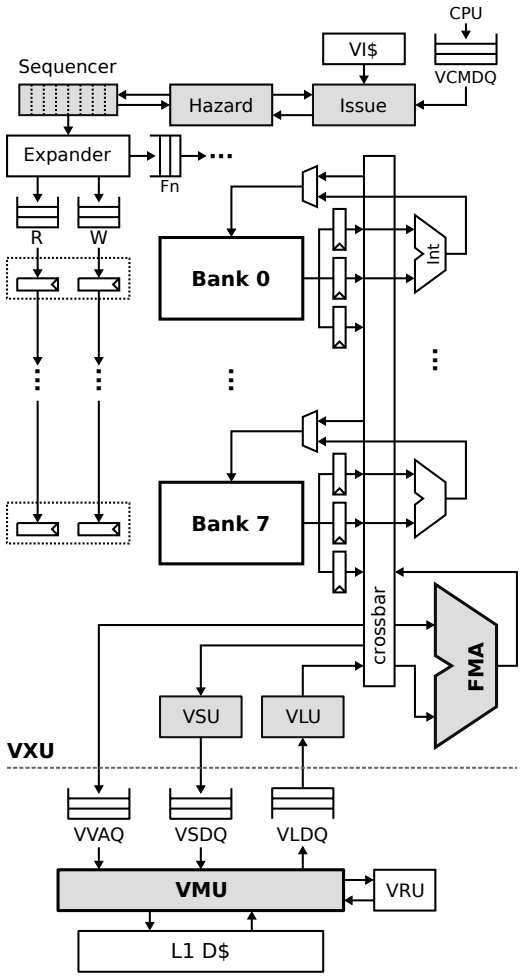

Figure 4: Block diagram of the Hwacha accelerator

Once clear of the hazard unit, instructions are placed into the sequencer, a circular array of state and control information about every active instruction. Each entry in the sequencer encapsulates an operation associated with a particular functional unit. A pointer cycles across the eight sequencer entries in a static round-robin schedule. When the pointer reaches a valid entry in the array and no relevant stall conditions are raised, the corresponding operation is conveyed to the expander.

The expander converts a sequencer operation into its constituent micro-ops (μops), which are sets of low-level control signals. These are inserted into shift registers with the displacement of the read and write μops coinciding exactly with the functional unit latency. Upon being released to the lane, the μops iterate through the microthreads as they sequentially traverse the banks cycle by cycle, as depicted in Figure 5. For vector lengths not a multiple of the bank count, the μops automatically disable themselves for elements past the end of the vector. The sequencer operation retires when it has been sequenced enough iterations to complete a hardware vector.

A crossbar switch connects the vector register file, organized into eight 256×64b SRAM banks, with the long-latency functional units.

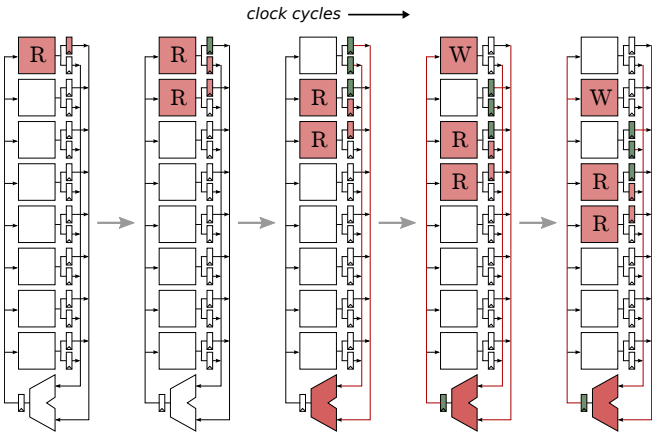

Figure 5: Systolic bank execution. The banked vector register file effectively delivers two operands per cycle from each 1R1W SRAM bank after an initial two-cycle latency.

4.2. Vector Memory Unit

The VMU exists to fulfill memory requests from the VXU. To better tolerate variable memory latency and simplify timing, the decoupled VMU interfaces with the rest of the vector unit through a set of queues [6]. Figure 6 outlines the internal organization of the VMU.

For each memory operation, the issue unit populates the VMU command queue with the operation type, data type, and vector length. A separate queue provides a base address and a stride for traditional vector operations. The Address Generation Unit (AGU) decomposes constant-stride vector accesses into an individual request for each element. Addresses for scatters and gathers instead arrive from the VXU through the Vector Virtual Address Queue (VVAQ). Virtual addresses are translated and saved in the Vector Physical Address Queue (VPAQ). Requests are throttled by the VXU to facilitate restartable exceptions [16]. The Vector Refill Unit (VRU) analyzes impending vector memory operations and generates prefetch commands for the memory hierarchy [5].

Load and store data are buffered within the Vector Load Data Queue (VLDQ) and Vector Store Data Queue (VSDQ), respectively. Each register file bank deposits store data into a shallow two-entry queue, whose contents are then multiplexed by the Vector Store Unit (VSU) into the VSDQ.

The Vector Load Unit (VLU) bears the task of routing elements from the VLDQ to their respective register file banks. As the memory interface may return responses in arbitrary order, shallow two-entry queues in front of the banks implement an opportunistic writeback mechanism to reduce latency.

5. Mixed-Precision Implementation

Our microarchitectural extensions for mixed precision focus on the modules shaded in Figures 4 and 6. We introduced reduced-precision arithmetic units, modified the issue/hazard units and the sequencer to consider the side effects of register compaction, and implement packed accesses in the VMU.

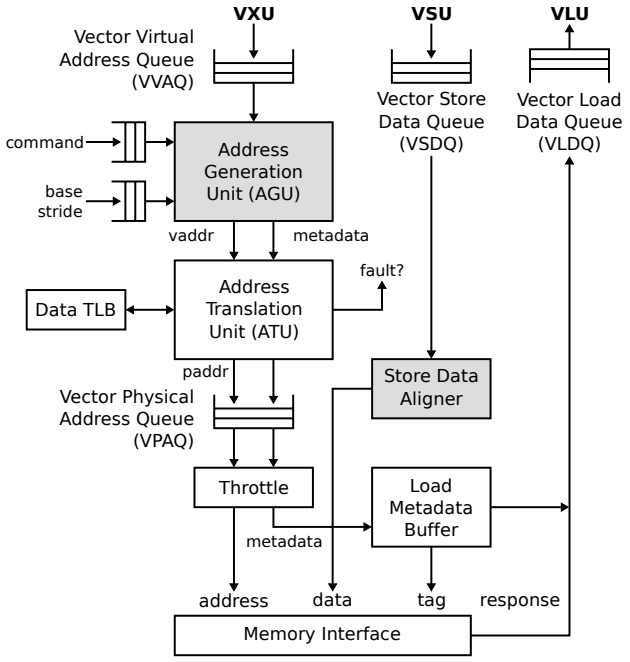

Figure 6: Block diagram of the Vector Memory Unit

Mixed-precision computation introduces new hazards arising from variations in throughput. The baseline design, for all operations, processes exactly eight elements with every pass through the lane. However, by packing two single-precision or four half-precision floating-point values into a physical register, each pass in the MVP design can potentially manage as many as 16 or 32 elements, respectively. The challenges in control logic may be summarized as follows:

1. Packed operations cannot proceed at maximum rate when subject to a data dependency on a slower predecessor that has not yet completed.
2. Source and destination registers may be read and written at unequal rates for mixed-precision operations.

5.1. Mixed-Precision Vector Chaining

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. This technique dramatically compresses the execution time with a justifiable degree of control complexity. Chaining, however, relies on balancing throughput between operations.

Fundamentally, newer and faster instructions must be prevented from overtaking older and slower instructions. Here, age pertains to program order, and speed pertains to the maximum number of elements processed per sequencing event. Since the sequencer already records the progress of each operation, it requires only minimal extra logic to stall an operation when it is poised to advance beyond a previous one. This approach is overly conservative in that it restricts progress regardless of whether actual data hazards occur, but compared with examining true register dependencies, the amount of logic is greatly diminished without sacrificing much performance.

5.2. Divergent Read/Write Rates

Operations involving unequal input and output precisions are more difficult to sequence because the rate at which operands are read differs from that at which results are written.

For example, the `fmv.x.h` instruction moves half-precision floating-point values into integer registers. It reads four times as many elements in one pass through the eight banks as it can write in one pass through the eight banks. To compensate for the mismatch, the same SRAM entries are read four times, but each pass selects a different portion of the SRAM entry and hence a different architectural register. The 32 converted results are written in four passes across the eight banks. However, unlike the read operation, the address of the destination register changes between each pass of the write operation.

As another example, the `fcvt.s.d` instruction converts from double-precision to single-precision floating-point values. In this case, each read of a whole source SRAM entry corresponds to half of a destination entry. To store partial results, a write mask is provided to the SRAM array.

5.3. Register Mapping

The register mapping scheme—how SRAM entries are organized into architectural registers—strongly affects the efficiency of microthread execution. So as to maintain a regular systolic cascade of µops through the banks, the entire state of a microthread resides within the same SRAM bank. Consecutive microthreads are striped across banks, such that microthreads execute sequentially by index as a µop descends. This avoid structural hazards that would otherwise require µops to be scheduled differently for distinct banks. Essential to the tractability of the control logic is the "fire-and-forget" principle, which stipulates that µops must never stall after being produced by the expander.

These properties must be preserved when adapting the register mapping scheme to support architectural registers of varying widths. The banks are segmented into integer, double, single, and half regions, such that registers of the same type are situated together and can be addressed by constant stride. Due to the microthread striping, elements packed contiguously within the same SRAM entry correspond with indices eight apart (the number of banks). Although the VXU/VMU interface becomes slightly more complicated in rearranging elements between register order and memory order, mixed-precision vector chaining is substantially eased by mapping a microthread's registers to the same bank regardless of width.

5.4. Vector Memory Unit

Although orthogonal to subject of mixed precision, maintaining a balance between compute and memory performance is generally desirable per Ahmdahl's law. With subword packing now enabling the VXU to deliver load and store data at increased rates, the VMU must be similarly augmented to sustain higher throughput.

For unit-stride vector accesses to fully utilize the available memory bandwidth, the AGU now supports coalescing as many adjacent elements into each request as the width of the memory interface permits, which for the L1 data cache is 64 bits. The VMU correctly handles edge cases involving base addresses not 64 bit-aligned and irregular vector lengths not a multiple of the packing density. An aligner module, interposed between the VSDQ and the memory interface, shifts the store data appropriately for unaligned vector and scatter operations.

As mentioned in the prior subsection, elements are arranged non-contiguously in the packed register format to aid the implementation of mixed-precision vector chaining. The VLU repacks elements from memory to register order, using a rotation-based permutation network to route elements to multiple banks in parallel. The VSU performs the inverse maneuver, repacking elements from register to memory order.

6. Evaluation

We rely on direct simulation of a VLSI implementation to evaluate the microarchitecture in terms of area, performance, and energy dissipation. Preliminary results are encouraging.

6.1. Physical Design

The RTL description for Hwacha, along with Rocket and the uncore, is written in the Chisel hardware construction language [3]. The Chisel-generated Verilog code underwent a Synopsys-based ASIC design flow using Design Compiler for logic synthesis and IC Compiler for place-and-route, targeting the TSMC40GPLUS technology at a clock period of 1 ns. Without a memory compiler for the same process available to us, the SRAMs were modeled as black-box modules with area, timing, and power characterized by CACTI 6.5 [12].

Figure 7 displays the final VLSI layout. The critical paths involve the uncore and L1 data cache, with worst negative slacks of -0.18 ns for baseline and -0.20 ns for MVP.

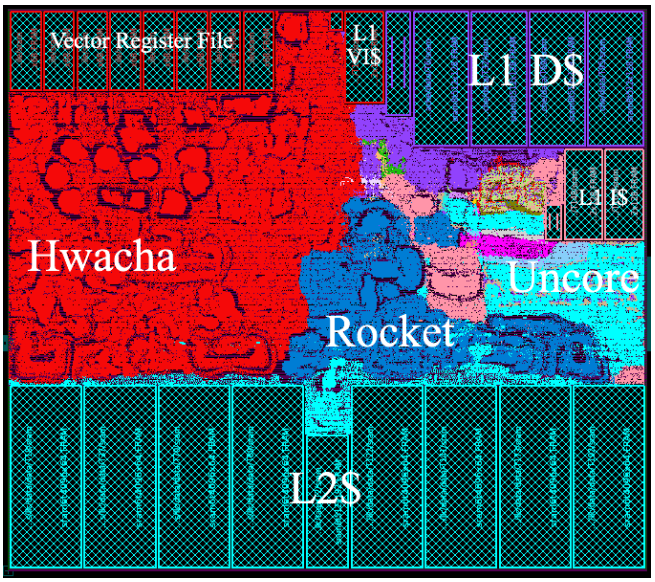

Figure 7: Layout of the reference chip

Table 1: Area distribution

Module Hierarchy	baseline mm²	baseline %	mvp mm²	mvp %	Δ %
Top	3.583	100.0	3.792	100.0	5.8
└ Hwacha	0.906	25.3	1.063	28.0	17.3
├ Issue	0.009	0.2	0.012	0.3	35.5
├ Seq	0.033	0.9	0.039	1.0	19.3
├ Lane	0.605	16.9	0.701	18.5	15.9
│ ├ FMA	0.149	4.2	0.235	6.2	57.7
│ └ Conv	0.030	0.8	0.030	0.8	1.6
├ VSU	0.014	0.4	0.017	0.5	21.9
├ VLU	0.024	0.7	0.036	0.9	49.4
└ VMU	0.051	1.4	0.082	2.2	60.9

6.2. Benchmarking

Power estimates were obtained with the Synopsys PrimeTime suite from simulation of the post-layout gate-level netlist, back-annotated with parasitic information extracted from ICC.

For each of the following evaluation kernels, we assessed several incarnations varying by floating-point precision (double, single, half) and by register width configuration, which either enables or disables packed execution (two single-precision or four half-precision values per bank entry). The recorded tests all commenced with a cold cache.

The **FFT** kernel performs a radix-2 one-dimensional fast Fourier transform across $n = 1024$ data points. The butterfly's bit-reversal permutation is realized through scatter/gather memory operations, which tend to limit the performance of the VMU and dominate execution time. The benchmark therefore gauges the improvement due solely to increased FMA throughput and smaller memory footprint from narrower datatypes.

The **AXPY** kernel calculates $Ax + y$, where A is a square matrix and x and y are vectors of dimension $n = 128$. The unit-stride vector memory operations exercise the VMU under favorable conditions, in order to determine the maximum effectiveness of coalesced accesses and cache prefetching by the VRU.

6.3. Comparative Analysis

Area The modifications for mixed-precision processing in Hwacha result in a 0.2 mm² expansion in area, or a 17.3% increase in the accelerator itself. This manifests as an overhead of 5.8% in terms of the overall chip area. Table 1 enumerates a breakdown by hierarchy. Introducing two single-precision and six half-precision float-point units, distributed equally among two independent FMA clusters, contributes 0.086 mm². Growth of the VMU, VLU, and VSU accounts for 0.046 mm².

Performance Table 2 summarizes the benchmarking results. If half-precision suffices, subword packing in the mixed-precision design enhances performance by 22.1% for FFT and 62.1% for AXPY over baseline double-precision. Leveraging coalesced accesses alone still effects gains of 14.1% for FFT and 58.8% for AXPY. Both benchmarks ultimately become memory-bounded, and so the speedup remains sublinear relative to the packing density, owing largely to cache misses.

Power All of the supplementary logic in Hwacha exacts an inevitable rise in power consumption. The most pronounced increases, as detailed in Table 3, ensue from the lane, sequencer, and VMU. Forthcoming optimizations for clock gating should significantly curtail the power dissipation of idle floating-point units. Aside from the expected improvement of the functional units at reduced precisions, the overall power efficiency of the vector lane also benefits considerably from diminished SRAM read and write activity with denser register packing.

Energy With lesser precisions, the penalty in power is sufficiently offset by the increased throughput to yield an ample recovery in total energy expenditure. Single-precision generally represents a break-even point; packed half-precision operations can conserve as much as 11% overall.

Table 2: Performance and energy

	precision	packed?	baseline cycles	baseline µJ	mvp cycles	mvp µJ	Δ % cycles	Δ % µJ
FFT	double	no	135919	12.4	135789	14.0	−0.1	12.9
	single	no	119275	10.4	119601	11.9	0.3	14.4
	single	yes	—		109683	10.5	−8.0	1.0
	half	no	116767	9.9	117093	11.4	0.3	15.2
	half	yes	—		105921	9.6	−9.3	−3.0
AXPY	double	no	100687	8.4	100443	9.4	−0.2	11.9
	single	no	60673	5.1	57579	5.3	−5.1	3.9
	single	yes	—		53817	4.8	−11.3	−5.9
	half	no	41521	3.5	38427	3.5	−7.5	0.0
	half	yes	—		38199	3.1	−8.0	−11.4

Note: Numbers from packed kernels on **mvp** are compared to those from unpacked kernels of the same precision on **baseline**.

7. Conclusions and Future Work

In future design iterations, we plan to implement the full range of floating-point and integer conversion operations, many of which exercise the newly-implemented rate control logic in the sequencer. We aim to support subword packing of integer registers as we have done for floating-point values.

Furthermore, we intend to examine the benefits of mixed-precision vector processing on wider applications such as the Smith-Waterman local alignment algorithm, a staple of computational biology. Because the genomic sequences used by the algorithm select from an alphabet of four to five characters, using the full 64 bits of a modern architectural register would be wasteful where 8 bits or fewer would suffice. Molecular dynamics simulation offers another avenue to explore—convergence and accuracy concerns aside, reducing the precision of these computations stand to greatly improve the speed of simulation.

To relieve programmers from the task of manually configuring Hwacha for mixed-precision computation, we would like to integrate it with support for Precimonious, a tool that automatically determines the appropriate floating-point precision based on values observed during runtime [14].

Table 3: Average power (mW)

		baseline			mvp				
	precision	double	single	half	double	single	single	half	half
	packed?	no	no	no	no	no	yes	no	yes
FFT	Top	91.4	86.9	84.5	103.0	99.9	95.5	97.5	90.5
	└ Hwacha	54.5	51.7	50.5	65.5	64.3	59.6	63.0	55.2
	├ Issue	0.1	0.2	0.2	0.2	0.2	0.2	0.2	0.2
	├ Seq	2.6	2.7	2.7	3.5	3.6	3.6	3.6	3.5
	├ Lane	39.4	36.6	35.5	48.1	46.6	41.8	46.0	38.0
	├ FMA	9.6	7.2	6.5	13.6	11.9	10.5	11.2	9.4
	└ Conv	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8
	├ VSU	0.9	0.9	0.9	0.9	0.8	0.7	0.7	0.5
	├ VLU	1.5	1.3	1.2	2.1	2.2	2.2	1.8	1.9
	└ VMU	2.3	2.5	2.3	3.1	3.4	3.3	3.2	3.3
AXPY	Top	83.9	83.3	83.4	93.7	92.3	90.1	90.3	81.4
	└ Hwacha	37.1	38.3	41.2	46.2	48.5	45.0	52.2	43.4
	├ Issue	0.1	0.2	0.2	0.2	0.2	0.2	0.2	0.2
	├ Seq	2.3	2.4	2.3	3.0	2.9	3.1	2.9	3.1
	├ Lane	22.8	23.2	25.5	30.4	32.8	29.3	36.5	28.3
	├ FMA	5.0	5.0	5.0	8.4	8.4	8.4	8.4	8.4
	└ Conv	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8
	├ VSU	0.3	0.4	0.4	0.4	0.4	0.4	0.4	0.3
	├ VLU	1.3	1.5	1.6	1.7	1.7	1.8	1.7	1.7
	└ VMU	1.3	1.7	2.1	1.9	1.8	1.8	1.6	1.6

To conclude, we augmented Hwacha to support mixed-precision vector processing while retaining architectural simplicity. The performance and energy efficiency gains from reduced-precision configurations, as demonstrated by our preliminary VLSI implementation, affirm the viability of mixed-precision computation in programmable accelerators.

8. Acknowledgements

We would like to thank Stephen Twigg for contributing the FFT software implementation and for assisting with floorplanning.

Research is partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture 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 does not necessarily reflect the position or the policy of the sponsors.

References

[1] *ARM Architecture Reference Manual: ARMv8, for ARMv8-A architectural profile*, ARM Limited, Dec. 2013.

[2] K. Asanović, "Vector Microprocessors," Ph.D. dissertation, University of California, Berkeley, May 1998.

[3] 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*, June 2012, pp. 1212–1221.

[4] C. Batten, "Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators," Ph.D. dissertation, Massachusetts Institute of Technology, Feb. 2010.

[5] C. Batten, R. Krashinsky, S. Gerding, and K. Asanović, "Cache Refill/Access Decoupling for Vector Machines," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37. Washington, DC, USA: IEEE Computer Society, 2004, pp. 331–342.

[6] R. Espasa and M. Valero, "A Simulation Study of Decoupled Vector Architectures," *The Journal of Supercomputing*, vol. 14, no. 2, pp. 124–152, Sep. 1999.

[7] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, Intel Corporation, Feb. 2014.

[8] C. Kozyrakis, "Scalable Vector Media Processors for Embedded Systems," Ph.D. dissertation, University of California, Berkeley, May 2002.

[9] A. E. Lahti, "Scientific processor vector file organization," Oct. 1989, US Patent 4,875,161.

[10] Y. Lee, R. Avižienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the Tradeoffs Between Programmability and Efficiency in Data-Parallel Accelerators," *ACM Transactions on Computer Systems*, vol. 31, no. 3, pp. 6:1–6:38, Aug. 2013.

[11] 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.

[12] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007. MICRO 2007.* IEEE Computer Society, Dec. 2007, pp. 3–14.

[13] 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, Jan. 2011.

[14] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning Assistant for Floating-point Precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 27:1–27:12.

[15] R. M. Russell, "The CRAY-1 Computer System," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.

[16] H. Vo, Y. Lee, A. Waterman, and K. Asanović, "A Case for OS-Friendly Hardware Accelerators," in *7th Annual Workshop on the Interaction between Operating System and Computer Architecture (WIVOSCA), at ISCA*, 2013.

[17] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA Version 2.0," EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014.