# Vector Processors for Energy-Efficient Embedded Systems

Daniel Dabbelt, Colin Schmidt, Eric Love, Howard Mao, Sagar Karandikar, and Krste Asanovic
University of California: Berkeley
{dpd,colins,ericlove,zhemao,sagark,krste}@eecs.berkeley.edu

## ABSTRACT

High-performance embedded processors are frequently designed as arrays of small, in-order scalar cores, even when their workloads exhibit high degrees of data-level parallelism (DLP). We show that these multiple instruction, multiple data (MIMD) systems can be made more efficient by instead directly exploiting DLP using a modern vector architecture. In our study, we compare arrays of scalar cores to vector machines of comparable silicon area and power consumption. Since vectors provide greater performance across the board - in some cases even with better programmability - we believe that embedded system designers should increasingly pursue vector architectures for machines at this scale.

## 1. INTRODUCTION

Modern computing platforms of all sizes must meet demands for improved energy efficiency even as they attempt to supply increasingly sophisticated functionality. Since clock frequencies can no longer increase due to the end of Dennard scaling, architects have been forced to expose some degree of parallelism to the user in order to continue to provide more computational throughput. As mobile architectures run ever more complex algorithms on larger amounts of data, energy efficiency will continue to be an important concern in the future.

The simplest and most popular way to exploit parallelism in embedded systems today is to take an energy-efficient, lightweight in-order core and replicate it many times to produce an array of processors that operate independently. This multiple instruction, multiple data (MIMD) paradigm has allowed for continued performance scaling, but is far from optimal. Most of the software being run on these devices - from DSP code in radio controllers to convolutional neural networks in more recent internet-of-things (IoT) endpoints - consists primarily of numerically-intensive, highly data-parallel kernels. As a result, the multiple cores in a MIMD array tend to execute copies of the *same* instructions across multiple data elements. In power and area terms, this is extremely wasteful, since the instruction fetch mechanism is one of the most expensive components of a general-purpose processor.

The obvious way to reduce this waste is to execute a *single instruction on multiple data* items. However SIMD, or *packed SIMD* as it is generally understood, is also an inefficient architecture for embedded devices. SIMD is normally implemented by instantiating several ALUs and providing instructions that supply multiple source operands to these ALUs simultaneously, which then execute in lock-step. SIMD ISAs usually include a different opcode for every possible length of input vector. As a result, they are difficult to program and adding more ALUs cannot improve performance unless software is rewritten to take advantage of the microarchitectural improvements. Worse, they impose a heavy area burden on embedded processors, because they require replicating compute resources for each element of the longest supported vector type. However, these copies will necessarily remain idle during instructions that use less than the full width. Furthermore, SIMD architectures cannot amortize the overhead of instruction fetch over as many elements as a vector machine since the degree of amortization is tied directly to the amount of duplication.

*Vector processors* are a much more effective design for exploiting DLP. In addition to *spatial* replication of compute resources, vector processors can provide *temporal* reuse of a decoded instruction by configuring a single ALU to execute the instruction across multiple data elements over a number of cycles. Moreover, they can hide the actual number of available compute units as a *microarchitectural* parameter, allowing software to simply specify the amount of work to be done. This often permits a single binary to execute at near-optimal efficiency on cores with varying levels of parallelism.

Though vector processors are commonly associated with Cray-sized supercomputers, their flexibility also enables them to scale down to embedded computers. The contribution of this paper is to show that the vector paradigm is an attractive alternative to MIMD in the embedded space. We compare mobile processor-sized arrays of in-order cores to single- and multi-lane vector designs of similar silicon area and power consumption.

More specifically, we compare Hwacha, a modern vector core based on the vector-fetch [12] paradigm to an array of single-issue, in-order RISC-V Rocket [8] cores. We performed cycle-accurate comparisons between all designs by simulating complete RTL for each processor. Both types of processors were instantiated in Rocket Chip [2], an SoC generator that supplies core-to-core and core-to-memory inter-

*MES '16 June 19-19, 2016, Seoul, Republic of Korea*
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4262-9/16/06.
DOI: http://dx.doi.org/10.1145/2934495.2934497

connect and I/O peripherals. Rocket, Hwacha, and Rocket Chip are all written in the Chisel [3] HDL, an object-oriented, functional hardware design language embedded in Scala and targetted at producing reusable, highly-parameterizable hardware generator libraries. We synthesized all designs in a popular, high-performance 28nm mobile-oriented process to obtain area and timing measurements. In the interest of time, no layout was done and gate-level activities were not taken into account when calculating power numbers. However, our estimates are validated by measurements taken from real silicon that has been fabricated in the past from the same RTL.

## 1.1 Related Work

Energy efficient mobile processors are a very hotly contested area, both in academia and industry; therefor it is be impossible to provide an exhaustive list of all related projects. A few representative projects are listed below:

- Rigel[5] explored a highly parallel MIMD architecture that might compete with high performance GPUs in the 10-100W range. Our own baseline MIMD processor, Rocket, is significantly less scalable than the Rigel architecture, but since we only evaluate the baseline for smaller core counts, these differences should be negligible. Rocket's microarchitecture is similar to Rigel's and thus is comparable to many other MIMD designs in this space.

- The Maven[9] project provides another comparison of vector-thread machines with a host of different architectural patterns, including MIMD architectures. It introduces the vector-thread architectural pattern; we later summarize in detail how traditional vector architectures differ from the Hwacha's vector-fetch model.

- A whole host of embedded processors focus on short integer performance: for example[4][6][7]. This work focuses on floating-point performance. Since floating-point operations are significantly more expensive than short integer operations, evaluations of systems focused on short integer operations cannot be extrapolated to systems focused on floating-point operations.

- ARM's Mali GPU is a commercial implementation of an explicitly data-parallel mobile processor. Since it is the most closely-related product to our Hwacha vector-fetch architecture, a detailed comparison of the two systems has been performed to demonstrate that Hwacha is competitive with modern commercial multicore embedded systems [11].

The contribution of this paper is to dispel the myth that MIMD processor arrays are always the best means of exploiting parallelism in embedded devices, and that vector- and other data-parallel architectures are too costly for such use cases.

## 2. DATA-PARALLEL PROGRAMMING MODELS

DLP is pervasive in signal processing, computer vision, and machine learning workloads that constitute much of the processing demand of modern mobile systems. In fact, many of these applications can be structured as data-flow graphs of small, data-parallel kernels such as DGEMM, FFT, stencils, filters, and others. As a result, very few computational cycles are spent in branchy integer control code, and the need to exploit instruction-level parallelism is correspondingly small. Exploiting DLP, however, is crucial to attaining the low-power operating regime required by these devices' limited battery capacities, so we now describe the way in which our data-parallel engine exposes DLP architecturally.

## 2.1 Vector-Fetch: The Hwacha Programming Model

The Hwacha programming model is best explained by contrast with other, popular data-parallel programming models. As a running example, we use a conditionalized SAXPY kernel, CSAXPY. Figure 1 shows CSAXPY expressed in C as both a vectorizable loop and as a SPMD kernel. CSAXPY takes as input an array of conditions, a scalar **a**, and vectors **x** and **y**; it computes $\mathbf{y} \mathrel{+}= \mathbf{ax}$ for the elements for which the condition is true.

The simplest way to exploit DLP is, as described previously, to use a MIMD architecture targeted by a single program, multiple data (SPMD) programming model. The SPMD implementation of CSAXPY is hardly different from the original serial loop, containing only a little more logic to divide inputs between all available threads. This is thus the easiest programming model to target, as each thread can operate as if on a general-purpose CPU.

Figure 2a shows CSAXPY kernel mapped to a hypothetical packed SIMD architecture, similar to Intel's SSE and AVX extensions. This SIMD architecture has 128-bit registers, each partitioned into four 32-bit fields. As with other packed SIMD machines, ours cannot mix scalar and vector operands, so the code begins by filling a SIMD register with copies of **a**. To map a long vector computation to this architecture, the compiler generates a *stripmine loop*, each iteration of which processes one four-element vector. In this example, the stripmine loop consists of a load from the conditions vector, which in turn is used to set a predicate register. The next four instructions, which correspond to the body of the *if*-statement in Figure 1a, are masked by the predicate register[1]. Finally, the address registers are incremented by the SIMD width, and the stripmine loop is repeated until the computation is finished—almost. Since the loop handles four elements at a time, extra code is needed to handle up to three *fringe* elements. For brevity, we omitted this code; in this case, it suffices to duplicate the loop body, predicating all of the instructions on whether their index is less than **n**.

The most important drawback to packed SIMD architectures lurks in the assembly code: the SIMD width is expressly encoded in the instruction opcodes and memory addressing code. When the architects of such an ISA wish to increase performance by widening the vectors, they must add a new set of instructions to process these vectors. This consumes substantial opcode space: for example, Intel's newest AVX instructions are as long as 11 bytes. Worse, application code cannot automatically leverage the widened vectors. In order to take advantage of them, application code must be

[1]We treat packed SIMD architectures generously by assuming the support of full predication. This feature is quite uncommon. Intel's AVX architecture, for example, only supports predication as of 2015, and then only in its Xeon line of server processors.

recompiled. Conversely, code compiled for wider SIMD registers fails to execute on older machines with narrower ones. As we later show, this complexity is merely an artifact of poor design.

A key feature of Cray-style vector architectures is the *vector length register* (VLR), which represents the number of vector elements that will be processed by the vector instructions, up to the hardware vector length (HVL). Software manipulates the VLR by requesting a certain application vector length (AVL); the vector unit responds with the smaller of the AVL and the HVL. As with packed SIMD architectures, a stripmine loop iterates until the application vector has been completely processed. But, as Figure 2b shows, the difference lies in the manipulation of the VLR at the head of every loop iteration. The primary benefits of this architecture follow directly from this code generation strategy. Most importantly, the scalar software is completely oblivious to the hardware vector length: the same code executes correctly and with maximal efficiency on machines with any HVL. Second, there is no fringe code: on the final trip through the loop, the VLR is simply set to the length of the fringe.

The Hwacha baseline architecture builds on the traditional vector architecture, 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 then completes the stripmining loop faster and is able to continue doing useful work, while the vector processor is independently executing the vector instructions.

Like the traditional vector machine, Hwacha has vector data registers (`vv`*n*) and vector predicate registers (`vp`*n*), but it also has two flavors of scalar registers. These are the *shared* registers (`vs`*n*), which can be read and written within a vector fetch block, and *address* registers (`va`*n*), which are read-only within a vector fetch block.

Figure 2c shows the CSAXPY code for the Hwacha machine. The structure of the stripmine loop is similar to the traditional vector code, but instead of explicitly executing the vector instructions, this stripmine loop simply moves the array pointers to the vector unit, then executes a vector fetch instruction, causing the Hwacha unit to execute the vector fetch block. The code in the vector fetch block is equivalent to the vector code in Figure 2b, with the addition of a `vstop` instruction, signifying the end of the block.

## 3. EXPERIMENTAL METHODOLOGY

We validate our claims about the effectiveness of vector designs by configuring the Rocket Chip paramaterizable system-on-chip generator to emit two kinds of processors, both of which are described below.

### 3.1 MIMD Baseline

Our baseline MIMD microarchitecture consists of four or more instances of the Rocket single-issue, in-order pipeline, which is shown in Figure 3. Each Rocket core contains split 16KiB L1 data and instruction caches, a floating point unit, and a coherent interface to a shared multi-banked L2 cache.

### 3.2 Hwacha

Hwacha is designed as a decoupled co-processor that attaches to the Rocket in-order core. Configuration instructions and vector fetch blocks are pushed to Hwacha's command queue for execution. Hwacha consists of one or more replicated vector lanes assisted by a single scalar unit. A scheduler for the entire machine distributes the work to each of the lanes, which individually schedule their own execution, allowing lanes to slip with respect to each other. Lanes are composed of: a banked register file, for vector and predicate registers; pipelined functional units, including four double precision FMAs; and a memory interface to the L2, with a TLB for virtual memory support[1]. Extensive decoupling enables the microarchitecture to effectively tolerate long and variable memory latencies with an in-order design, which is detailed in [10].

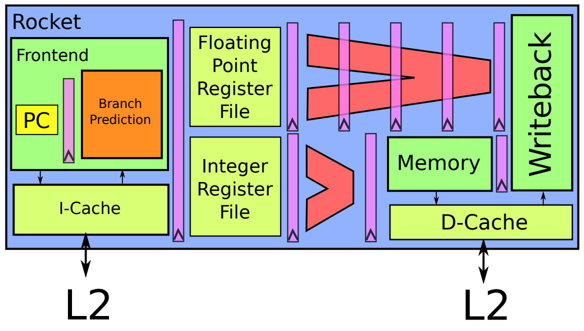


**Figure 3: Rocket 5-stage in-order pipeline**

### 3.3 Memory System

In addition to generating core arrays, Rocket Chip is capable of generating an interconnect, L2 cache, and connection to an array of memory controllers. In order to perform a fair comparison between the baseline and Hwacha, similar memory system configurations were generated for both the baseline MIMD architecture and the proposed Hwacha vector architecture. Figure 5 shows the memory system for the MIMD-8 configuration evaluated in this paper.

### 3.4 Benchmarks

In order to compare the performance of Hwacha with the baseline MIMD microarchitecture we use a suite of 5 microbenchmarks designed to span the range of common computations that embedded processors must perform. These benchmarks were hand-tuned for each of the configurations, both MIMD and vector, under evaluation

- `vvadd`: A 1000-element, double-precision vector-vector addition. This benchmark has no contention, but has a low arithmetic intensity.
- `dgemm`: A $32 \times 32$, double-precesion matrix multiplication. Deep neural networks heavily rely on good matrix multiplication performance.
- `mask-sfilter`: A masked stencil filter. This is representative of common image processing applications, which are frequently run on high-performance embedded processors. This benchmark was written in OpenCL and compiled for each configuration under evaluation.
- `csaxpy`: A conditional SAXPY. Filters with conditional elements are common in image processing applications This serves to demonstrate the performance

```
void csaxpy(size_t n, bool cond[],
    float a, float x[], float y[])
{
  for (size_t i = 0; i < n; ++i)
    if (cond[i])
      y[i] = a*x[i] + y[i];
}
```

(a) vectorizable loop

```
void csaxpy_spmd(size_t n, size_t threads, size_t tid,
    bool cond[], float a, float x[], float y[])
{
  const size_t chunk_sz = n / threads;
  const size_t max = MIN((tid+1)*chunk_sz, n);
  for(size_t i = tid * chunk_sz; i < max; i++)
    if (cond[i])
      y[i] = a*x[i]+y[i];
}
```

(b) SPMD kernel

**Figure 1: Conditional SAXPY kernels written in C. The SPMD kernel launch code for (b) is omitted for brevity.**

```
csaxpy_simd:
     slli    a0, a0, 2
     add     a0, a0, a3
     vsplat4 vv0, a2
stripmine:
     vlb4    vv1, (a1)
     vcmpez4 vp0, vv1
!vp0 vlw4    vv1, (a3)
!vp0 vlw4    vv2, (a4)
!vp0 vfma4   vv1, vv0, vv1, vv2
!vp0 vsw4    vv1, (a4)
     addi    a1, a1, 4
     addi    a3, a3, 16
     addi    a4, a4, 16
     bleu    a3, a0, stripmine
# handle edge cases
# when (n % 4) != 0 ...
     ret
```

(a) SIMD

```
csaxpy_tvec:
stripmine:
     vsetvl  t0, a0
     vlb     vv0, (a1)
     vcmpez  vp0, vv0
!vp0 vlw     vv0, (a3)
!vp0 vlw     vv1, (a4)
!vp0 vfma    vv0, vv0, a2, vv1
!vp0 vsw     vv0, (a4)
     add     a1, a1, t0
     slli    t1, t0, 2
     add     a3, a3, t1
     add     a4, a4, t1
     sub     a0, a0, t0
     bnez    a0, stripmine
     ret
```

(b) Traditional Vector

```
csaxpy_lov:
  vsetcfg ...
  vmss vs0, a2
stripmine:
  vsetvl  t0, a0
  vmsa    va0, a1
  vmsa    va1, a3
  vmsa    va2, a4
  vf      csaxpy_work
  add     a1, a1, t0
  slli    t1, t0, 2
  add     a3, a3, t1
  add     a4, a4, t1
  sub     a0, a0, t0
  bnez    a0, stripmine
  ret

csaxpy_work:
     vlb     vv0, (va0)
     vcmpez  vp0, vv0
!vp0 vlw     vv0, (va1)
!vp0 vlw     vv1, (va2)
!vp0 vfma    vv0, vv0, vs0, vv1
!vp0 vsw     vv0, (va2)
     vstop
```

(c) Hwacha

**Figure 2: CSAXPY kernel mapped to data-parallel architectures. Pseudo-assembly implements kernel in Figure 1 for (a) packed SIMD, (b) traditional vector, and (c) Hwacha. In all implementations, `a0` holds variable `n`, `a1` holds pointer `cond`, `a2` holds scalar `a`, `a3` holds pointer `x`, and `a4` holds pointer `y`.**

degradation the various microarchitectures experience in the presence of control flow divergence.

- `histogram`: A histogram with 1000 bins. This kernel is bound by contention in the cache, and serves to characterize the performance of each design in the presence of irregular memory access patterns.

## 3.5 Simulated Design Configurations

We simulated Hwacha and the baseline MIMD architecture at several different design points, and compared configurations that have the same functional unit bandwidth. Since one Hwacha lane can perform up to 8 double-precision operations per cycle (as 4 FMAs), we compare it against a 4-core MIMD machine, which offers the same peak throughput. In both cases, we also use the same memory system, consisting of a 256KiB L2 cache (arranged into 4 banks) and a single channel of LPDDR3 memory.

We maintained the equality of functional unit bandwidth between MIMD and vector designs as we scaled to larger configurations. Thus, we compare 2-lane Hwacha against an 8-tile MIMD machine, and 4 vector lanes against 16 scalar cores. We also chose to scale memory bandwidth upwards, adding an additional LPDDR3 channel for each vector lane or group of four cores. This is consistent with the trend in mobile devices to scale memory bandwidth aggressively with area via a combination of flip chip and die stacking. A separate paper provides a more detailed comparison of these designs to extant mobile SoCs [11].

## 3.6 Performance

The Rocket Chip system-on-chip generator was extended to generate all of the configurations under evaluation in this paper. Rocket Chip generates working, synthesizeable RTL, which greatly simplifies the evaluation methodology: cycle-accurate simulation data can be obtained for the benchmarks from the same RTL as was used to generate VLSI synthesis results. All benchmarks were run to completion on all evaluated platforms under cycle-accurate RTL simulation.

In addition to producing cycle-accurate runtime numbers, the RTL generated by Rocket Chip for each configuration was synthesized Synopsys' Design Compiler to produce gate-level clock speed, area, and power numbers. Results were produced using Synopsys' reference methodology for a popular commercial 28nm high-performance mobile process, with memories compiled using the foundry's SRAM compiler.

While the authors acknowledge that measurements taken from synthesis and before place-and-route can be wildly inaccurate, we believe that the clock frequency and area numbers are reasonable for the presented Hwacha configurations and are optimistic for the baseline MIMD architecture. Similar configurations to those listed in the paper have been implemented in silicon at similar area and clock frequency targets by both the authors and others on both this and similar processes[13].

## 4. RESULTS

Table 1 reports the execution times in cycles for each benchmark, while Table 2 gives the VLSI synthesis results

| Design | MIMD-4 | Vector-1 | MIMD-8 | Vector-2 | MIMD-16 | Vector-4 |
|---|---|---|---|---|---|---|
| vvadd [cycles] | 4573 | 2354 | 2803 | 1714 | 2500 | 1331 |
| dgemm [cycles] | 51139 | 16411 | 37987 | 11499 | 26074 | 9147 |
| mask-sfilter [cycles] | 38441 | 22195 | 21373 | 18401 | 13324 | 18306 |
| csaxpy [cycles] | 47552 | 18213 | 23353 | 11586 | 14358 | 6543 |
| histogram [cycles] | 528983 | 544913 | 230787 | 245010 | 238078 | 104090 |

**Table 1: Cycle Time Results**

| Design | MIMD-4 | Vector-1 | MIMD-8 | Vector-2 | MIMD-16 | Vector-4 |
|---|---|---|---|---|---|---|
| Clock Frequency [GHz] | 1.09 | 0.95 | 1.08 | 0.90 | 0.92 | 0.85 |
| Logic Area [mm$^2$] | 0.237 | 0.354 | 0.481 | 0.642 | 1.026 | 1.226 |
| Macro Area [mm$^2$] | 1.164 | 1.290 | 2.322 | 2.387 | 4.780 | 4.530 |
| Full Design Area [mm$^2$] | 1.565 | 1.861 | 3.163 | 3.395 | 6.523 | 6.414 |
| Full Design Power [mW] | 162 | 172 | 332 | 275 | 738 | 481 |

**Table 2: Synthesis Results**

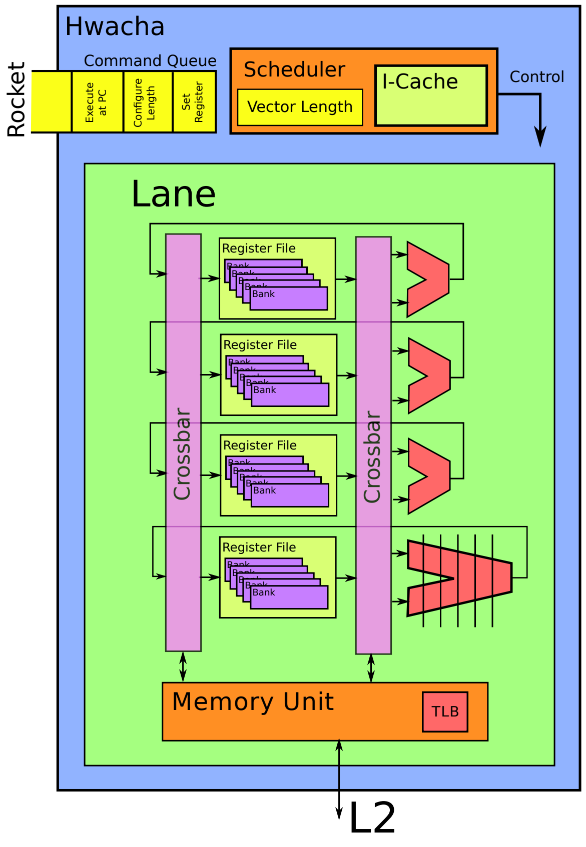


**Figure 4: A 1 lane, 4 bank Hwacha vector unit**

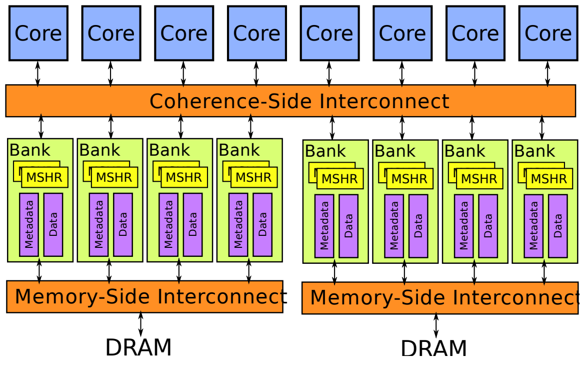
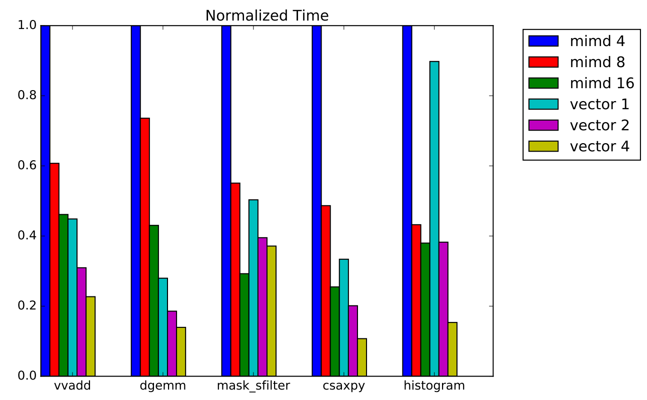


**Figure 5: The MIMD-8 configuration's memory system**

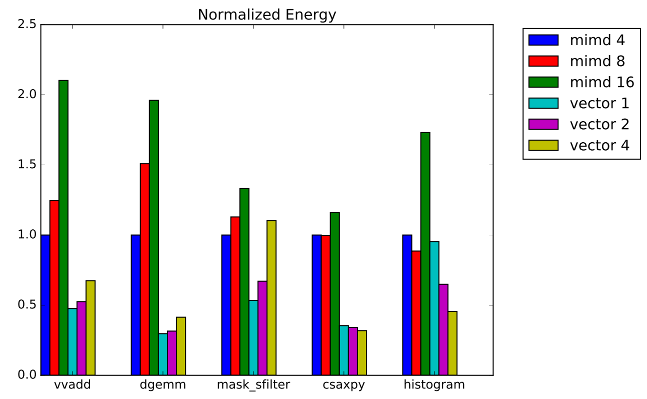
for each reference platform. To compare our design points, Figure 6a depicts the normalized absolute execution time for each of our benchmarks on each reference design point relative to the baseline 4-core MIMD machine. Recall that the MIMD-4 configuration has roughly equivalent hardware resources to the single-lane Vector-1 design, while MIMD-8 matches Vector-2, and MIMD-16 matches Vector-4. Across the board, resource-matched vector designs outperform their MIMD counterparts. In fact, for highly data-parallel kernels like VVADD and DGEMM, even the single-lane vector unit outperforms *all* MIMD designs. Figure 6b shows the total energy consumption of each benchmark normalized to the energy usage of MIMD-4. Although the larger MIMD designs provide performance scaling as compared to MIMD-4, they all *increase* overall energy use. The single-lane vector design, on the other hand, nearly always uses less energy than all of the MIMD designs.

## 5. CONCLUSIONS

Despite their popular association with high-area, high-power devices like GPUs and out-of-order superscalar SIMD extensions, data parallel accelerators are also suitable and even optimal for power-constrained, limited-area embedded processor designs. In particular, classic Cray-style temporal



(a) Execution Time Normalized to MIMD-4



(b) Total Energy Normalized to MIMD-4

vector processors can be built at small sizes, and substantially outperform multicore MIMD arrays of equivalent area on workloads of interest to mobile application developers. They do this in spite of their flexible programming model, which can accommodate a much wider variety of codes than fixed-function or even SIMD-based accelerators. We have shown, through synthesis and simulation of RTL for complete MIMD and vector machines validated against fabricated silicon, that vector extensions really are feasible in tight area and power budgets, and that they are an attractive competitor to arrays of in-order cores. We hope designers will consider this approach for their future products.

## 6. ACKNOWLEDGEMENTS

Research 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, Hewlett-Packard, Huawei, LGE, NVIDIA, Oracle, and Samsung. Earlier work was supported in part by Intel Corporation and DARPA awards W911NF-08-1-0134, W911NF-08-1-0139, and W911NF-09-1-0342. Research also supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). 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.

## 7. REFERENCES

[1] A case for os-friendly hardware accelerators. 7th Workshop on the Interaction between Operating System and Computer Architecture (WIVOSCA-2013), at the 40th International Symposium on Computer Architecture (ISCA-40), 2013.

[2] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[4] J. Balfour, W. Dally, D. Black-Schaffer, V. Parikh, and J. Park. An energy-efficient processor architecture for embedded systems. *IEEE Computer Architecture Letters*, 7(1):29–32, Jan 2008.

[5] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 140–151, New York, NY, USA, 2009. ACM.

[6] C. E. Kozyrakis and D. A. Patterson. Scalable, vector processors for embedded systems. *IEEE Micro*, 23(6):36–45, Nov 2003.

[7] Christoforos Kozyrakis. *Scalable Vector Media-processors for Embedded Systems*. PhD thesis, 2002. AAI3063439.

[8] Y. Lee, A. Waterman, R. Avizienis, 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 *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014 - 40th*, pages 199–202, Sept 2014.

[9] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. *SIGARCH Comput. Archit. News*, 39(3):129–140, June 2011.

[10] Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, and Krste Asanović. The hwacha microarchitecture manual, version 3.8.1. Technical Report UCB/EECS-2015-263, EECS Department, University of California, Berkeley, Dec 2015.

[11] Yunsup Lee, Colin Schmidt, Sagar Karandikar, Daniel Dabbelt, Albert Ou, and Krste Asanović. Hwacha

preliminary evaluation results, version 3.8.1. Technical Report UCB/EECS-2015-264, EECS Department, University of California, Berkeley, Dec 2015.

[12] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and Krste Asanović. The hwacha vector-fetch architecture manual, version 3.8.1. Technical Report UCB/EECS-2015-262, EECS Department, University of California, Berkeley, Dec 2015.

[13] Yunsup Lee, Brian Zimmer, Andrew Waterman, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Ben Keller, Stevo Bailey, Milovan Blagojevic, Pi-Feng Chiu, Henry Cook, Rimas Avizienis, Brian Richards, Elad Alon, Borivoje Nikolic, and Krste Asanovic. Raven: A 28nm risc-v vector processor with integrated switched-capacitor dc-dc converters and adaptive clocking. HotChips, 2015.