Efficient VLSI Implementations of Vector-Thread Architectures

by Yunsup Lee

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor K. Asanović
Research Advisor

(Date)

* * * * * * *

Professor J. Wawrzynek
Second Reader

(Date)

Abstract

We present a taxonomy and modular implementation approach for data-parallel accelerators, including the MIMD, vector-SIMD, subword-SIMD, SIMT, and vector-thread (VT) architectural design patterns. We introduce Maven, a new VT microarchitecture based on the traditional vector-SIMD microarchitecture, that is considerably simpler to implement and easier to program than previous VT designs. Using an extensive design-space exploration of full VLSI implementations of many accelerator design points, we evaluate the varying tradeoffs between programmability and implementation efficiency among the MIMD, vector-SIMD, and VT patterns on a workload of compiled microbenchmarks and application kernels. We find the vector cores provide greater efficiency than the MIMD cores, even on fairly irregular kernels. Our results suggest that the Maven VT microarchitecture is superior to the traditional vector-SIMD architecture, providing both greater efficiency and easier programmability.

Contents

List of Figures

List of Tables

1 Introduction

Data-parallel kernels dominate the computational workload in a wide variety of demanding application domains, including graphics rendering, computer vision, audio processing, physical simulation, and machine learning. Specialized data-parallel accelerators [10, 31, 8, 23, 15] have long been known to provide greater energy and area efficiency than general-purpose processors for codes with significant amounts of data-level parallelism (DLP). With continuing improvements in transistor density and an increasing emphasis on energy efficiency, there has recently been growing interest in DLP accelerators for mainstream computing environments. These accelerators are usually attached to a general-purpose host processor, either on the same die or a separate die. The host processor executes system code and non-DLP application code while distributing DLP kernels to the accelerator. Surveying the wide range of data-parallel accelerator cores in industry and academia reveals a general tradeoff between programmability (how easy is it to write software for the accelerator?) and efficiency (energy/task and tasks/second/area). In this paper, we examine multiple alternative data-parallel accelerators to quantify the efficiency impact of microarchitectural features intended to simplify programming or expand the range of code that can be executed.

We first introduce a set of five architectural design patterns for DLP cores in Section 2, qualitatively comparing their expected programmability and efficiency. The MIMD pattern [10] flexibly supports mapping data-parallel tasks to a collection of simple scalar or multithreaded cores, but lacks mechanisms for efficient execution of regular DLP. The vector-SIMD [28, 31] and subword-SIMD [8] patterns can significantly reduce the energy on regular DLP, but can require complicated programming for irregular DLP. The single-instruction multiple-thread (SIMT) [17] and vector-thread (VT) [15] patterns are hybrids between the MIMD and vector-SIMD patterns that attempt to offer alternative tradeoffs between programmability and efficiency.

When reducing these high-level patterns to an efficient VLSI design, there is a large design space to explore. In Section 3, we present a common set of parameterized synthesizable microarchitectural components and show how these can be combined to form complete RTL designs for the different architectural design patterns, thereby reducing total design effort and allowing a fairer comparison across patterns. In this section, we also introduce Maven, a new VT microarchitecture. Our modular design strategy revealed a much simpler and more efficient implementation than the earlier Scale VT design [15, 3, 13, 14]. Maven [2, 16] is based on a vector-SIMD microarchitecture with minimal changes to enable the improved programmability from VT, instead of the decoupled cluster microarchitecture of Scale. Another innovation in Maven is to use the same RISC ISA for both vector and scalar code, greatly reducing the effort required to develop an efficient VT compiler. The Scale design required a separate clustered ISA for vector code, which complicated compiler development [9].

To concretely evaluate and compare the efficiency of these patterns, we have generated and analyzed hundreds of complete VLSI layouts for the MIMD, vector-SIMD, and VT patterns using our parameterized microarchitecture components targeting a modern 65 nm technology. Section 4 describes our methodology for extracting area, energy, and performance numbers for a range of compiled microbenchmarks and application kernels. Section 5 presents and analyzes our results.

Our results show that vector cores are considerably more efficient in both energy and area-normalized performance than MIMD cores, although the MIMD cores are usually easier to program. Our results also suggest that the Maven VT microarchitecture is superior to the traditional vector-SIMD architecture, providing greater efficiency and a simpler programming model. For both VT and vector-SIMD, multi-lane implementations are usually more efficient than multi-core single-lane implementations and can be easier to program as they require less partitioning and load balancing. Although we do not implement a SIMT machine, some initial analysis indicates SIMT will be less efficient than VT but should be easier to program.

1.1 Collaboration, Previous Publications, and Funding

This thesis is a result of a collaborative group project. Other people have made direct contributions to the ideas and results that are included in this thesis. The Maven VT core was developed by Krste Asanović, Christopher Batten, myself, and a group of students from 2007 through 2011. Christopher Batten was the lead architect for Maven. He helped direct the development and evaluation of the architecture, microarchitecture, RTL, compiler, microbenchmarks, and application kernels. Christopher was responsible for the Maven instruction set architecture, C++ compiler, programming support libraries, assembly test suite, microbenchmarks, and application kernels. I started working on the project from 2008, and later took the lead on improving the Maven architecture to better support irregular DLP from 2010 when Christopher graduated. I was responsible for the microarchitecture and the RTL implementation of the Maven VTU, and several microarchitectural optimizations including banking, per-bank integer ALUs, and density-time execution. I also worked on the initial Maven C++ compiler port, developed the Maven functional simulator, and brought up the CAD toolflow. Rimas Avizienis took the lead on the microarchitecture and the RTL implementation of the control processor, multithreaded MIMD processor, and the vector memory unit. Rimas also wrote the Maven proxy kernel, integrated the cache timing model, and helped on various CAD toolflow issues. Alex Bishara took the lead on the microarchitecture and the RTL implementation of the 1-stack, and 2-stack pending vector fragment buffer, and wrote several application kernels. Richard Xia took the lead on the microarchitecture and RTL implementation of the dynamic memory coalescer, wrote several application kernels, and developed a tool that draws graphs by analyzing hundreds of data points. Both Alex and Richard developed an infrastructure to manage hundreds of machines to execute many RTL simulation, synthesis, place-and-route, gate-level simulation, and power simulation jobs simultaneously. Derek Lockhart took the lead on writing vector-SIMD versions of microbenchmarks and application kernels. Christopher Celio took the lead on the definition

of the traditional vector instruction set architecture, and wrote several application kernels. Hidetaka Aoki made significant contributions to some of the very early ideas about single-lane VTUs. Finally, Krste Asanović was integral in all aspects of the project.

Some of the figures and content in this thesis are adapted from previous publications, including "Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators" from ISCA 2011 [16], and Christopher Batten's thesis "Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators" [2]. More specifically, the contributions of this thesis over the previous publications include: in Section 2, additional discussion of each architectural pattern, and execution diagrams for irregular DLP, in Section 3, example of banked register file read port scheduling, example and more details about the convergence schemes, and more details about the memory coalescing scheme, in Section 4, more configurations in Table 1, additional microbenchmarks and discussion, and more details about the programming methodology with examples, in Section 5, more example VLSI layouts, new data points in Figure 16, and more detailed energy results for the microarchitectural optimizations.

This work was supported in part by Microsoft (Award #024263) and Intel (Award #024894, equipment donations) funding and by matching funding from U.C. Discovery (Award #DIG07-10227).

2 Architectural Design Patterns for Data-Parallel Accelerators

Data-parallel applications can be categorized in two dimensions: the regularity with which data memory is accessed and the regularity with which the control flow changes. *Regular data-level parallelism* has well structured data accesses where the addresses can be compactly encoded and are known well in advance of when the data is ready. Regular DLP also has well structured control flow where the control decisions are either known statically or well in advance of when the control flow actually occurs. *Irregular data-level parallelism* might have less structured data accesses where the addresses are more dynamic and difficult to predict, and might also have less structured control flow with data-dependent control decisions. Irregular DLP might also include a small number of inter-task dependencies that force a portion of each task to wait for previous tasks to finish. Eventually a DLP kernel might become so irregular that it is better categorized as *task-level parallelism*.

Figure 1 uses simple loops to illustrate the spectrum from regular to irregular DLP. The regular loop in Figure 1(a) includes unit-stride accesses (`A[i]`,`C[i]`), strided accesses (`B[2*i]`), and shared accesses (`x`). The loop in Figure 1(b) uses indexed accesses (`E[C[i]]`,`D[A[i]]`). The loop in Figure 1(c) includes a data-dependent conditional to choose the correct shared constant, while the irregular loop in Figure 1(d) includes conditional accesses (`B[i]`,`C[i]`) and computation. The irregular loop in Figure 1(e) includes an inner loop with a complex data-dependent exit condition.

There have been several studies which demonstrate that full DLP applications contain a mix of regular and irregular DLP [29, 15, 27, 18]. Accelerators that can handle a wider variety of DLP are more attractive than those which are restricted to just regular DLP for many reasons. First, it is possible to improve performance and energy-efficiency even on irregular DLP. Second, even if the performance and energy-efficiency on irregular DLP is similar to a general-purpose processor, by keeping the work on the accelerator we make it easier to exploit regular DLP inter-mingled with irregular DLP. Finally, a consistent way of mapping both regular and irregular DLP simplifies the programming methodology. The rest of this section presents five architectural patterns for the design of data-parallel accelerators, and describes how each pattern handles both regular and irregular DLP.

```
for ( i = 0; i < n; i++ )
  C[i] = x * A[i] + B[2*i];
```

(a) Regular DA & Regular CF

```
for ( i = 0; i < n; i++ )
  E[C[i]] = D[A[i]] + B[i];
```

(b) Irregular DA & Regular CF

```
for ( i = 0; i < n; i++ )
  x = ( A[i] > 0 ) ? y : z;
  C[i] = x * A[i] + B[i];
```

(c) Regular DA & Irregular CF

```
for ( i = 0; i < n; i++ )
  if ( A[i] > 0 )
    C[i] = x * A[i] + B[i];
```

(d) Irregular DA & Irregular CF

```
for ( i = 0; i < n; i++ )
  C[i] = false; j = 0;
  while ( !C[i] & (j < m) )
    if ( A[i] == B[j++] )
      C[i] = true;
```

(e) Irregular DA & Irregular CF

Figure 1: Different Types of Data-Level Parallelism – Examples expressed in a C-like pseudocode and are ordered from regular DLP (i.e., regular data access (DA) and control flow (CF)) to irregular DLP (i.e., irregular data access (DA) and control flow (CF)). (from [2] and [16])

2.1 MIMD Architectural Design Pattern

The **multiple-instruction multiple-data** (MIMD) pattern is perhaps the simplest approach to building a data-parallel accelerator. A large number of scalar cores are replicated across a single chip. Programmers can map each data-parallel task to a separate core, but without any dedicated DLP mechanisms, it is difficult to gain an energy-efficiency advantage when executing DLP applications. These scalar cores can be extended to support per-core multithreading which helps improve performance by hiding various latencies. Figure 2(a) shows the programmer's logical view and an example implementation for the multithreaded MIMD pattern. All of the design patterns include a *host thread* (HT) as part of the programmer's logical view. The HT runs on the general-purpose processor and is responsible for application startup, configuration, interaction with the operating system, and managing the data-parallel accelerator. We refer to the threads that run on the data-parallel accelerator as *microthreads* (µTs), since they are lighter weight than the threads which run on the general-purpose processor. The primary advantage of the MIMD pattern is the flexible programming model, and since every core can execute a fully independent task, there should be little difficulty in mapping both regular and irregular DLP applications. This can simplify parallel programming compared to the other design patterns, but the primary disadvantage is that this pattern does little to improve the energy efficiency of DLP applications.

The pseudo-assembly in Figure 3(a) illustrates how we might map a portion of a simple irregular loop in Figure 1(d) to each µT. The first ten instructions divide the work among the µTs such that each thread works on a different consecutive partition of the input and output arrays. Notice that all µTs redundantly load the shared scalar value `x` (line 11). This might seem trivial, but the lack of a specialized mechanism to handle shared loads and possibly also shared computation can adversely impact many regular DLP codes. Similarly there are no specialized mechanisms to take advantage of the regular data accesses. Figure 4(a) shows an execution diagram corresponding to the pseudo-assembly in Figure 3(a) for a 2-core, 4-µT implementation with two-way multithreading illustrated in Figure 2(a). The scalar instructions from each µT are interleaved in a fixed pattern. It is very natural to map the data-dependent conditional to a scalar branch (line 15) which simply skips over the unnecessary work when possible. It is also straight-forward to implement conditional loads and stores of the `B` and `C` arrays by simply placing them after the branch. The execution diagram shows how the µTs are *coherent* (execute in lock-step) before the branch and then *diverge* after the data-dependent conditional with µT0 and µT3 quickly moving on to the next iteration. After a few iterations the µTs will most likely be completely diverged.

The recently proposed 1000-core Illinois Rigel accelerator is a good example of the MIMD pattern with a single µT per scalar core [10]. Sun's 8-core Niagara processors exemplify the spirit of the multithreaded MIMD pattern with 4–8 threads per core for a total of 32–64 threads per chip [12, 21]. The Niagara processors are good examples of the multithreading pattern, although they are not specifically data-parallel accelerators. Niagara threads are heavier-weight than µTs,

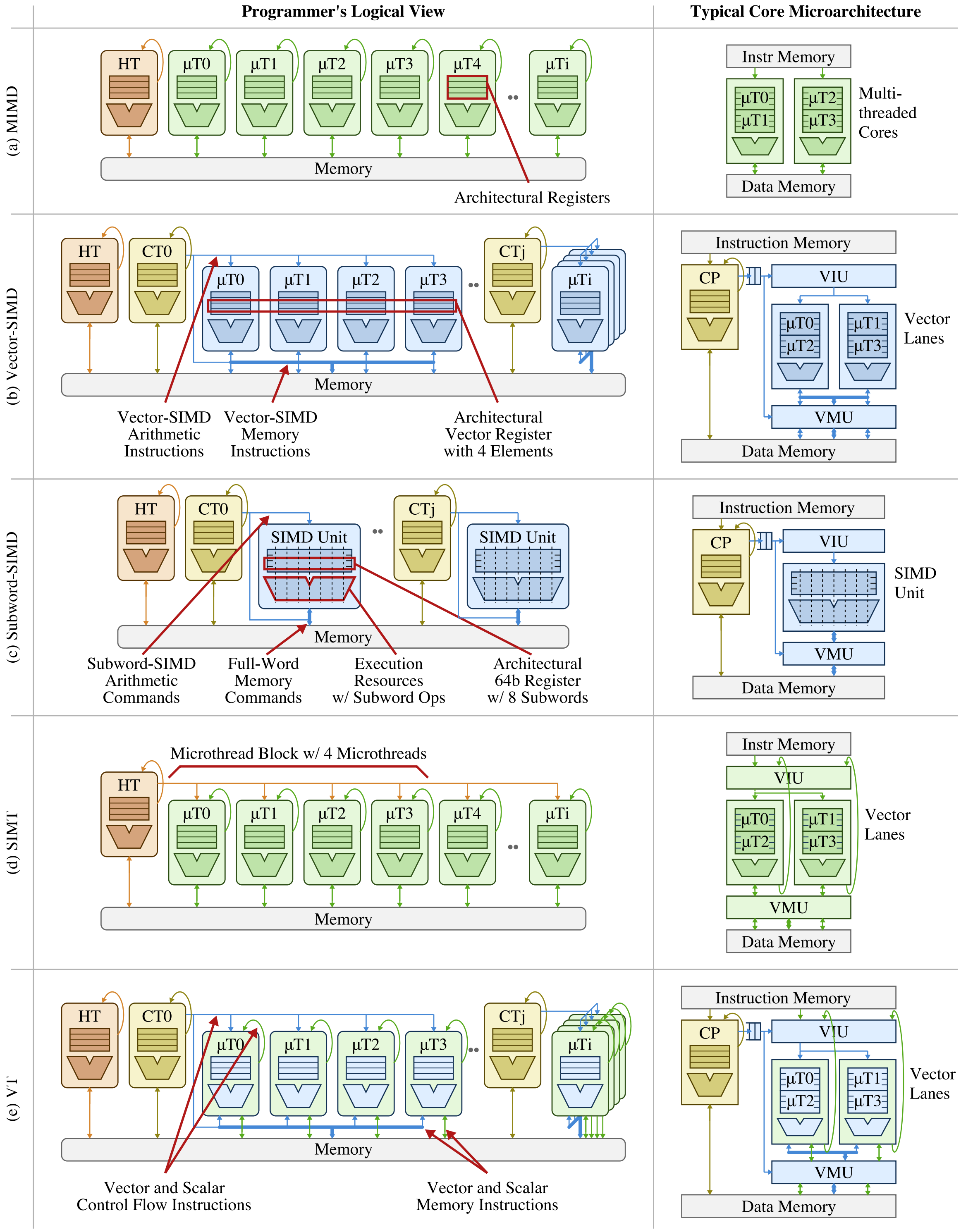


Figure 2: Architectural Design Patterns – Programmer's logical view and a typical core microarchitecture for five patterns: (a) MIMD, (b) vector-SIMD, (c) subword-SIMD, (d) SIMT, and (e) VT. HT = host thread, CT = control thread, CP = control processor, µT = microthread, VIU = vector issue unit, VMU = vector memory unit. (from [2] and [16])

```
 div    m, n, nthr
 mul    t, m, tidx
 add    a_ptr, t
 add    b_ptr, t
 add    c_ptr, t

 sub    t, nthr, 1
 br.neq t, tidx, ex
 rem    m, n, nthr
ex:
 load   x, x_ptr

loop:
 load   a, a_ptr
 br.eq  a, 0, done

 load   b, b_ptr
 mul    t, x, a
 add    c, t, b
 store  c, c_ptr

done:
 add    a_ptr, 1
 add    b_ptr, 1
 add    c_ptr, 1

 sub    m, 1
 br.neq m, 0, loop
```

(a) MIMD

```
 load     x, x_ptr

loop:
 setvl    vlen, n
 load.v   VA, a_ptr
 load.v   VB, b_ptr
 cmp.gt.v VF, VA, 0

 mul.sv   VT, x, VA,  VF
 add.vv   VC, VT, VB, VF
 store.v  VC, c_ptr,  VF

 add      a_ptr, vlen
 add      b_ptr, vlen
 add      c_ptr, vlen

 sub      n, vlen
 br.neq   n, 0, loop
```

(b) Vector-SIMD

```
 br.gte tidx, n, done

 add    a_ptr, tidx
 load   a, a_ptr
 br.eq  a, 0, done

 add    b_ptr, tidx
 add    c_ptr, tidx

 load   x, x_ptr
 load   b, b_ptr
 mul    t, x, a
 add    c, t, b
 store  c, c_ptr
done:
```

(c) SIMT

```
 load    x, x_ptr
 mov.sv  VZ, x

loop:
 setvl   vlen, n
 load.v  VA, a_ptr
 load.v  VB, b_ptr
 mov.sv  VD, c_ptr
 fetch.v ut_code

 add     a_ptr, vlen
 add     b_ptr, vlen
 add     c_ptr, vlen

 sub     n, vlen
 br.neq  n, 0, loop
 ...

ut_code:
 br.eq   a, 0, done
 mul     t, z, a
 add     c, t, b
 add     d, tidx
 store   c, d
done:
 stop
```

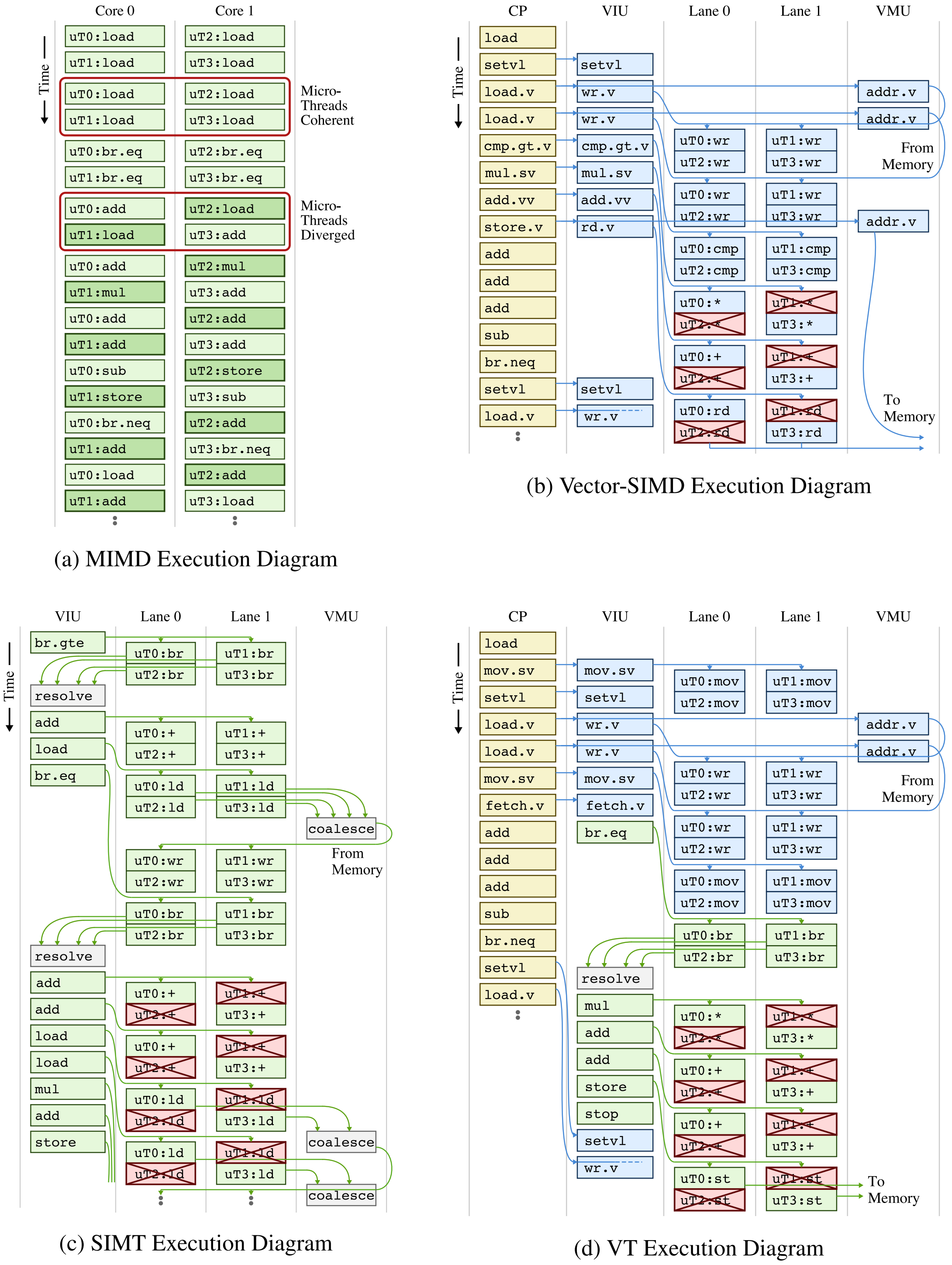
(d) VT

Figure 3: Pseudo-Assembly for Irregular DLP Example – Pseudo-assembly implements the loop in Figure 1(d) for the (a) MIMD, (b) vector-SIMD, (c) SIMT, and (d) VT patterns. Assume `*_ptr` and `n` are inputs. `V`*i* = vector register *i*, `VF` = vector flag register, `*.v` = vector command, `*.vv` = vector-vector op, `*.sv` = scalar-vector op, `nthr` = number of µTs, `tidx` = current microthread's index. (from [2] and [16])

and Niagara is meant to be a stand-alone processor as opposed to a true coprocessor. Even so, the Niagara processors are often used to execute both regular and irregular DLP codes, and their multithreading enables good performance on these kinds of codes [32]. These MIMD accelerators can be programmed using general-purpose parallel programming frameworks such as OpenMP [25] and Intel's Thread Building Blocks [26], or in the case of the Rigel accelerator, a custom task-based framework is also available [11].

2.2 Vector-SIMD Architectural Design Pattern

In the **vector single-instruction multiple-data** (vector-SIMD) pattern a *control thread* (CT) uses vector memory instructions to move data between main memory and vector registers, and vector arithmetic instructions to operate on vectors of elements at once. As shown in Figure 2(b), one way to think of this pattern is as if each CT manages an array of µTs that execute in lock-step; each µT is responsible for one element of the vector and the hardware vector length is the number of µTs (e.g., four in Figure 2(b)). In this context, µTs are sometimes referred to as virtual processors [34]. Unlike the MIMD pattern, the HT in the vector-SIMD pattern only interacts with the CTs and does not directly manage the µTs. Even though the HT and CTs must still allocate work at a coarse-grain



(a) MIMD Execution Diagram

(b) Vector-SIMD Execution Diagram

(c) SIMT Execution Diagram

(d) VT Execution Diagram

Figure 4: Execution Diagrams for Irregular DLP Example – Executions are for the loop in Figure 1(d) for the (a) MIMD, (b) vector-SIMD, (c) SIMT, and (d) VT patterns. CP = control processor, VIU = vector issue unit, VMU = vector memory unit. (from [2])

amongst themselves via software, this configuration overhead is amortized by the hardware vector length. The CT in turn distributes work to the µTs with vector instructions enabling very efficient execution of fine-grain DLP. In a typical vector-SIMD core, the CT is mapped to a *control processor* (CP) and the µTs are mapped both spatially and temporally across one or more *vector lanes* in the vector unit. The *vector memory unit* (VMU) handles executing vector memory instructions, and the *vector issue unit* (VIU) handles the dependency checking and eventual dispatch of vector arithmetic instructions.

Figures 3(b) shows the pseudo-assembly corresponding to the loop in Figure 1(d). Unit-stride vector memory instructions (lines 5–6,11) efficiently move consecutive blocks of data in and out of vector registers. A vector-vector arithmetic instruction (line 10) efficiently encode a regular arithmetic operation across the full vector of elements, and a combination of a scalar load and a scalar-vector instruction (lines 1,9) can easily handle shared accesses. In the vector-SIMD pattern the hardware vector length is not fixed by the instruction set but is instead stored in a special control register. The `setvl` instruction takes the *application vector length* (`n`) as an input and writes the minimum of the application vector length and the hardware vector length to the given destination register `vlen` (line 4). As a side-effect, the `setvl` instruction sets the *active vector length* which specifies how many of the µTs are active and should participate in a vector instruction. Software can use the `setvl` instruction to process the vectorized loop in blocks equal to the hardware vector length without knowing what the actual hardware vector length is at compile time. The `setvl` instruction will naturally handle the final iteration when the application vector length is not evenly divisible by the hardware vector length; `setvl` simply sets the active vector length to be equal to the final remaining elements. This technique is called *stripmining* and enables a single binary to handle varying application vector lengths while still running on many different implementations with varying hardware vector lengths. Note that a vector flag is used to conditionally execute the vector multiply, addition, and store instructions (lines 9–11). More complicated irregular DLP with nested conditionals can quickly require many independent flag registers and complicated flag arithmetic [30].

Figure 4(b) shows the execution diagram corresponding to the pseudo-assembly in Figure 3(b) for a two-lane, four-µT implementation depicted in Figure 2(b). The vector memory commands (lines 5–6,11) are broken into two parts: the address portion goes to the VMU which will issue the request to memory while the register write/read portion goes to the VIU. For vector loads, the register writeback waits until the data returns from memory and then controls writing the vector register file two elements per cycle over two cycles. Notice that the VIU/VMU are decoupled from the vector lanes to allow the implementation to overlap processing multiple vector loads. The vector arithmetic operations (lines 7,9–10) are also processed two elements per cycle over two cycles. Note that some µTs are *inactive* because the corresponding vector flag is false. The temporal mapping of µTs to the same lane is an important aspect of the vector-SIMD pattern. We can easily imagine using a larger vector register file to support longer vector lengths that would keep the vector unit busy for tens of cycles. The fact that one vector command can keep the vector unit busy for many cycles decreases instruction issue bandwidth pressure. So as in the MIMD pattern we can exploit instruction-level parallelism by adding support for executing multiple instructions per µT per cycle, but unlike the MIMD pattern it may not be necessary to increase the issue bandwidth, since one vector instruction occupies a vector functional unit for many cycles. Almost all vector-SIMD accelerators will take advantage of multiple functional units and also support bypassing (also called *vector chaining*) between these units. A final point to note is how the control processor decoupling and multi-cycle vector execution enables the control thread to continue executing while the vector unit is still processing older vector instructions. This decoupling means the control thread can quickly work through the loop overhead instructions (lines 13–18) so that it can start issuing the next iteration of the stripmine loop as soon as possible.

Figures 3(b) and 4(b) illustrate three ways that the vector-SIMD pattern can improve energy efficiency: (1) some instructions are executed once by the CT instead of for each µT as in the MIMD pattern (lines 1,13–18); (2) for operations that the µTs do execute (lines 5–11), the CP and VIU can amortize various overheads such as instruction fetch, decode, and dependency checking over `vlen` elements; and (3) for memory accesses which the µTs still execute (lines 5–6,11) the VMU can efficiently move the data in large blocks.

The Berkeley Spert-II system exemplifies this pattern, and its eight-lane vector-SIMD T0 processor helps accelerate neural network, multimedia, and digital signal processing applications [31]. The Spert-II system uses hand-coded assembly to vectorize critical kernels, but vectorizing compilers are also possible [5].

2.3 Subword-SIMD Architectural Design Pattern

The **subword single-instruction multiple-data** (subword-SIMD) architectural pattern shown in Figure 2(c) captures some important differences from the vector-SIMD pattern. In this pattern, the "vector-like unit" is really a full-word scalar datapath with standard scalar registers often corresponding to a double-precision floating-point unit. The pattern leverages these existing scalar datapaths and registers to execute multiple narrow-width operations in a single cycle. Some subword-SIMD variants support bitwidths larger than the widest scalar datatype, in which case the datapath can only be fully utilized with subword-SIMD instructions. Other variants unify the CT and SIMD unit such that the same datapath is used for both control, scalar arithmetic, and subword-SIMD instructions. Subword-SIMD has short vector lengths that are exposed to software as wide fixed-width datapaths, while vector-SIMD has longer vector lengths exposed to software as a true vector of elements. In vector-SIMD, the vector length is exposed in such a way that the same binary can run on many different implementations with varying hardware resources, while code for one subword-SIMD implementation is usually less portable to other implementations. Subword-SIMD often requires shuffling elements via special permute instructions, and this leads to a large amount of cross-element communication. Vector-SIMD has more flexible data-movement operations which alleviates the need for software data shuffling.

The IBM Cell processor is a good example of this pattern, with eight data-parallel cores each including a unified 128-bit subword-SIMD datapath that can execute scalar operations as well as 16 × 8-bit, 8 × 16-bit, 4 × 32-bit, or 2 × 64-bit operations [8]. In terms of programming methodology, many modern compilers include intrinsics for accessing subword-SIMD operations, and some compilers include optimization passes that can automatically vectorize regular DLP. In this work, we focus less on the subword-SIMD pattern, because the vector-SIMD pattern is better suited to exploiting large amounts of data-parallelism as opposed to a more general-purpose workload with smaller amounts of data-parallelism.

2.4 SIMT Architectural Design Pattern

The **single-instruction multiple-thread** (SIMT) pattern is a hybrid pattern with a programmer's logical view similar to the MIMD pattern but an implementation similar to the vector-SIMD pattern. As shown in Figure 2(d), the SIMT pattern supports a large number of µTs but no CTs; the HT is responsible for directly managing the µTs (usually through specialized hardware mechanisms). A *µT block* is mapped to a SIMT core which contains vector lanes similar to those found in the vector-SIMD pattern. However, since there is no CT, the VIU is responsible for amortizing overheads and executing the µT's scalar instructions in lock-step when they are coherent. The VIU also manages the case when the µTs execute a scalar branch possibly causing them to diverge. µTs can sometimes reconverge through static hints in the scalar instruction stream or dynamic hardware mechanisms. SIMT only has scalar loads and stores, but the VMU can include a memory coalescing unit to dynamically detect when these scalar accesses can be converted into vector-like memory operations. The SIMT pattern usually exposes the concept of a µT block to the programmer: barriers are sometimes provided for intra-block synchronization, and application performance depends heavily on the coherence and coalescing opportunities within a µT block.

The loop in Figure 1(d) maps to the SIMT pattern in a similar way as in the MIMD pattern except that each µT is usually only responsible for a single element as opposed to a range of elements (see Figure 3(c)). Since there are no control threads and thus nothing analogous to the vector-SIMD pattern's `setvl` instruction, a combination of dedicated hardware and software is required to manage the stripmining. The host thread tells the hardware how many µT blocks are required for the computation and the hardware manages the case when the number of requested µT blocks is greater than what is available in the actual hardware. In the common case where the application vector length is not statically guaranteed to be evenly divisible by the µT block size, each µT must use a scalar branch to verify that the computation for the corresponding element is actually necessary (line 1).

Figure 4(c) shows the execution diagram corresponding to the pseudo-assembly in Figure 3(c) for a two-lane, four-µT implementation shown in Figure 2(d). Scalar branch management corresponding to the branch at line 1 will be discussed later. Without a control thread, all four µTs redundantly perform address calculations (lines 3,7–8) and the actual scalar load instruction (lines 4,11) even though these are unit-stride accesses. The VMU dynamically checks all four addresses, and if they are consecutive, then the VMU coalesces these accesses into a single vector-like memory operation. Also notice that since there is no control thread to amortize the shared load at line 10, all four µTs must redundantly load x. The VMU may be able to dynamically coalesce this into one scalar load which is then broadcast to all four µTs. The VMU attempts to coalesce well-structured stores (line 14) as well as loads. Since the µTs are coherent when they execute the scalar multiply and addition instructions (lines 12–13), the VIU should be able to execute them with vector-like efficiencies. After issuing the scalar branch corresponding to line 5, the VIU waits for the µT block to calculate the branch resolution based on each µT's scalar data. The VIU then turns these branch resolution bits into a dynamically generated vector flag, which is used to mask off inactive elements on either side of the branch. Various SIMT implementations handle the details of µT divergence differently, but the basic idea is the same. In contrast to vector-SIMD (where the control processor is decoupled from the vector unit making it difficult to access the vector flag registers), SIMT can avoid fetching instructions when the vector flag bits are all zero. So if the entire µT block takes the branch at line 5, then the VIU can completely skip the instructions at lines 7–14 and start the µT block executing at the branch target. Also note that conditional memory accesses are naturally encoded by simply placing them after the branch (lines 10–11,14).

Figures 3(c) and 4(c) illustrate some of the issues that can prevent the SIMT pattern from achieving vector-like energy-efficiencies on regular DLP. The µTs must redundantly execute instructions that would otherwise be amortized onto the CT (lines 1–3,7–10). Regular data accesses are encoded as multiple scalar accesses (lines 4,11,14) which then must be dynamically transformed (at some energy overhead) into vector-like memory operations. In addition, the lack of a control thread necessitates per µT stripmining calculations (lines 1) and prevents access-execute decoupling which can efficiently tolerate memory latencies. Even so, the ability to achieve vector-like efficiencies on coherent µT instructions helps improve SIMT energy-efficiency compared to the MIMD pattern. The real strength of the SIMT pattern, however, is that it provides a simple way to map complex data-dependent control flow with µT scalar branches (line 5).

The NVIDIA Fermi graphics processor is a good example of this pattern with 32 SIMT cores each with 16 lanes suitable for graphics as well as more general data-parallel applications [23]. Various SIMT frameworks such as Microsoft's DirectX Compute [19], NVIDIA's CUDA [22], Stanford's Brook [4], and OpenCL [24] allow programmers to write high-level code for the host thread and to specify the scalar code for each µT as an annotated function. A combination of off-line compilation, just-in-time optimization, and hardware actually executes the data-parallel program.

2.5 VT Architectural Design Pattern

The **vector-thread** (VT) pattern is also a hybrid pattern but takes a very different approach from the SIMT pattern. As shown in Figure 2(e), the HT manages a collection of CTs and each CT in turn manages an array of µTs. Similar to the vector-SIMD pattern, this allows various overheads to be amortized onto the CT, and control threads can also execute vector memory commands to efficiently handle regular data accesses. Unlike the vector-SIMD pattern, the CT does not execute vector arithmetic instructions but instead uses a *vector fetch instruction* to indicate the start of a scalar instruction stream that should be executed by the µTs. The VIU allows the µTs to execute coherently, but as in the SIMT pattern, they can also diverge after executing scalar branches.

Figure 3(d) shows the VT pseudo-assembly corresponding to the loop in Figure 1(d). Stripmining (line 5), loop control (line 11–16), and regular data accesses (lines 6–7) are handled just as in the vector-SIMD pattern. Instead of vector arithmetic instructions, we use a vector fetch instruction (line 9) with one argument which indicates the instruction address at which all µTs should immediately start executing (e.g., the instruction at the `ut_code` label). All µTs execute these scalar instructions (lines 20–24) until they reach a `stop` instruction (line 26). An important part of the VT pattern is the interaction between vector registers as accessed by the control thread, and scalar registers as accessed by each µT. In this example, the unit-stride vector load at line 6 writes the vector register `VA` with `vlen` elements. Each µT's scalar register `a` implicitly refers to that µT's element of the vector register (e.g., µT0's scalar register `a` implicitly refers to the first element of the vector register `VA`). In other words, the vector register `VA` as seen by the control thread and the scalar register `a` as seen by the µTs are two views of the same register. The µTs cannot access the control thread's scalar registers, since this would significantly complicate control processor decoupling. Shared accesses are thus communicated with a scalar load by the control thread (line 1) and then a scalar-vector move instruction (lines 2,8) which copies the given scalar register value into each element of the given vector register. A scalar branch (line 20) is used to encode data-dependent control flow. µTs thus skip the instructions at 21–24 when the branch condition is true. The conditional store is encoded by placing the store after the branch (line 24) similar to the MIMD and SIMT examples.

Figure 4(d) illustrates how the pseudo-assembly in Figure 3(d) would execute on the implementation pictured in Figure 2(e). An explicit scalar-vector move instruction (line 2) writes the scalar value into each element of the vector register two elements per cycle over two cycles. The unit-stride vector load instructions (lines 6–7) execute as in the vector-SIMD pattern. The control processor then sends the vector fetch instruction to the VIU. The VIU fetches the branch instruction (line 20) and issues them across the µTs. Similar to the SIMT pattern, the VIU waits until all µTs resolve the scalar branch. If all µTs either take or do not take the branch, then the VIU can start fetching from the appropriate address. If some µTs take the branch while others do not, then the µT diverge and the VIU needs to keep track of which µT are executing which side of the branch.

Figures 3(d) and 4(d) illustrate how VT achieves vector-like energy-efficiency while maintaining the ability to flexibly map irregular DLP. Control instructions are executed once by the control thread per-loop (lines 1–2) or per-iteration (lines 11–16). A scalar branch (line 20) provides a convenient way to map complex data-dependent control flow. The VIU is still able to amortize instruction fetch, decode, and dependency checking for vector arithmetic instructions (lines 21–23). VT uses the same vector memory instructions to efficiently move blocks of data between memory and vector registers (lines 6–7). There are however some overheads including the extra scalar-vector move instruction (line 2), vector fetch instruction (line 9), and µT stop instruction (line 26).

The Scale VT processor is an early example of the VT pattern [15]. Scale's programming methodology uses either a combination of compiled code for the control thread and hand-coded assembly for the µTs, or a preliminary version of a vectorizing compiler written specifically for Scale [9].

3 Microarchitecture of MIMD, Vector-SIMD, and VT Tiles

In this section, we describe in detail the microarchitectures used to evaluate the various patterns. A data-parallel accelerator will usually include an array of tiles and an on-chip network to connect them to each other and an outer-level memory system, as shown in Figure 5(a). Each tile includes one or more tightly coupled cores and their caches, with examples in Figure 5(b)–(d). In this paper, we focus on comparing the various architectural design patterns with respect to a single data-parallel tile. The inter-tile interconnect and memory system are also critical components of a DLP accelerator system, but are outside the scope of this work.

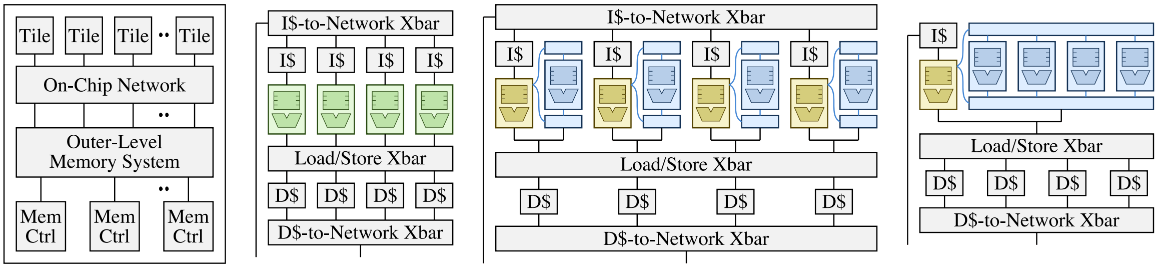
3.1 Microarchitectural Components

We developed a library of parameterized synthesizable RTL components that can be combined to construct MIMD, vector-SIMD and VT tiles. Our library includes long-latency functional units, a multi-threaded scalar integer core, vector lanes, vector memory units, vector issue units, and blocking and non-blocking caches.

A set of **long-latency functional units** provide support for integer multiplication and division, and IEEE single-precision floating-point addition, multiplication, division, and square root. These units can be flexibly retimed to meet various cycle-time constraints.

Our **scalar integer core** implements a RISC ISA, with basic integer instructions executed in a five-stage, in-order pipeline but with two sets of request/response queues for attaching the core to the memory system and long-latency functional units. A two-read-port/two-write-port (2r2w-port) 32-entry 32-bit register file holds both integer and floating-point values. One write port is for the integer pipeline and the other is shared by the memory system and long-latency functional units. The core can be multithreaded, with replicated architectural state for each thread and a dynamic thread scheduling stage at the front of the pipeline.

Figure 6 shows the microarchitectural template used for all the vector-based cores. A control



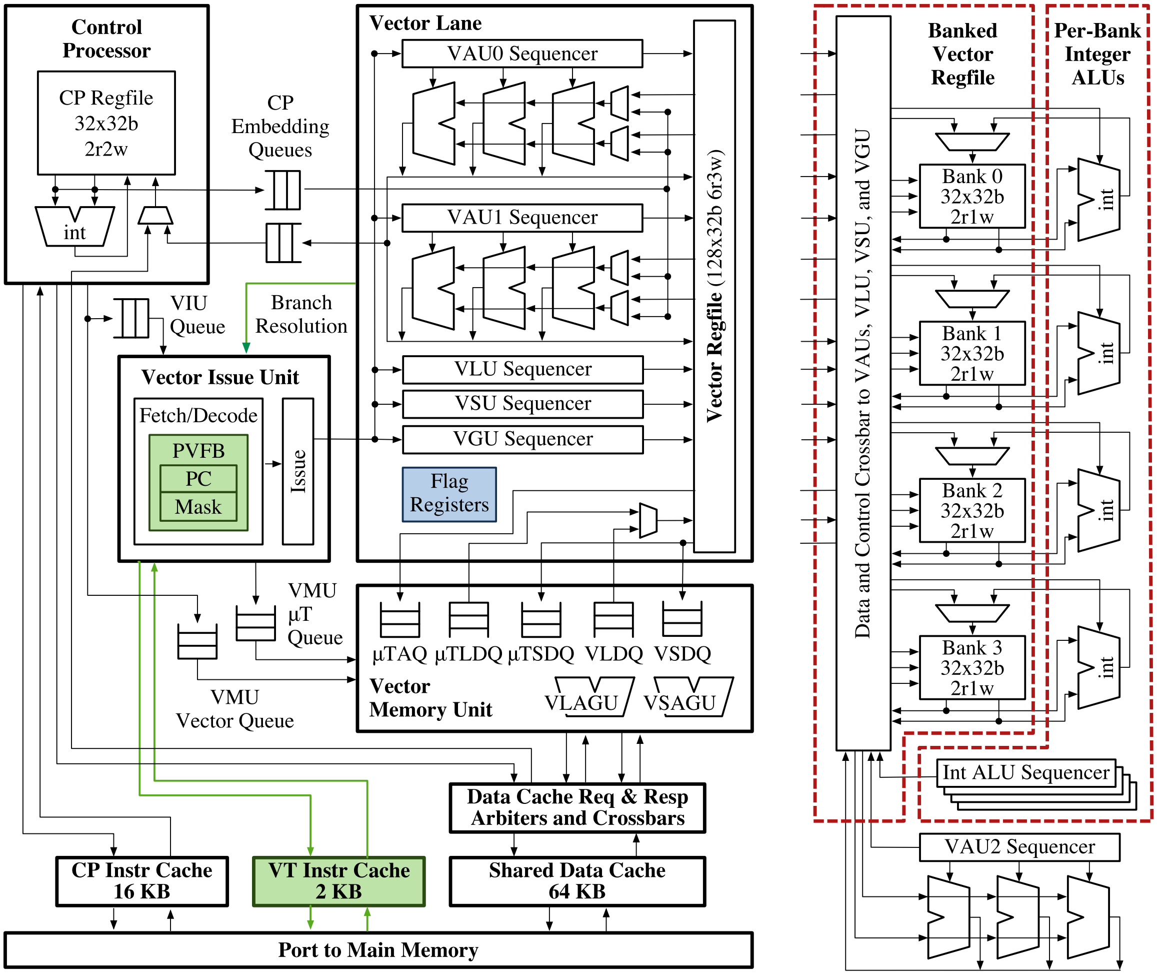
(a) Data-Parallel Accelerator

(b) MIMD Tile with Four Cores

(c) Vector-SIMD Tile with Four Single-Lane Cores

(d) Vector-SIMD Tile with One Four-Lane Core

Figure 5: Example Data-Parallel Tile Configurations (from [16])



(a) Baseline Vector-SIMD and VT Core Microarchitecture

(b) Banked Register File w/ Per-Bank Integer ALUs

Figure 6: Vector-Based Core Microarchitecture – (a) Each vector-based core includes one or more vector lanes, vector memory unit, and vector issue unit; PVFB = pending vector fragment buffer, PC = program counter, VAU = vector arithmetic unit, VLU = vector load-data writeback unit, VSU = vector store-data read unit, VGU = address generation unit for µT loads/stores, VLDQ = vector load-data queue, VSDQ = vector store-data queue, VLAGU/VSAGU = address generation unit for vector loads/stores, µTAQ = µT address queue, µTLDQ = µT load-data queue, µTSDQ = µT store-data queue. Modules specific to vector-SIMD or VT cores are highlighted. (b) Changes required to implement intra-lane vector register file banking with per-bank integer ALUs. (from [16])

processor (CP) sends vector instructions to the vector unit, which includes one or more vector lanes, a vector memory unit (VMU), and a vector issue unit (VIU). The lane and VMU components are nearly identical in all of the vector-based cores, but the VIU differs significantly between the vector-SIMD and VT cores as discussed below.

Our baseline **vector lane** consists of a unified 6r3w-port vector register file and five vector functional units (VFUs): two arithmetic units (VAUs), a load unit (VLU), a store unit (VSU), and an address-generation unit (VGU). Each VAU contains an integer ALU and a subset of the long-latency functional units. The vector register file can be dynamically reconfigured to support between 4–32 registers per µT with corresponding changes in maximum vector length (32–1). Each VFU has a sequencer to step through elements of each vector operation, generating physical register addresses.

The **vector memory unit** coordinates data movement between the memory system and the vector register file using decoupling [6]. The CP splits each vector memory instruction into a vector memory µop issued to the VMU and a vector register access µop sent to the VIU, which is eventually issued to the VLU or VSU in the vector lane. A load µop causes the VMU to issue a vector's worth of load requests to the memory system, with data returned to the vector load data queue (VLDQ). As data becomes available, the VLU copies it from the VLDQ to the vector register file. A store µop causes the VMU to retrieve a vector's worth of data from the vector store data queue (VSDQ) as it is pushed onto the queue by the VSU. Note that for single-lane configurations, the VMU still uses wide accesses between the VLDQ/VSDQ and the memory system, but moves data between the VLDQ/VSDQ and the vector lane one element at a time. Individual µT loads and stores (gathers and scatters) are handled similarly, except addresses are generated by the VGU and data flows through separate queues.

The main difference between vector-SIMD and VT cores is how the **vector issue unit** fetches instructions and handles conditional control flow. In a vector-SIMD core, the CP sends individual vector instructions to the VIU, which is responsible for ensuring that all hazards have been resolved before sending vector µops to the vector lane. Our vector-SIMD ISA supports data-dependent control flow using conventional vector masking, with eight single-bit flag registers. A µT is prevented from writing results for a vector instruction when the associated bit in a selected flag register is clear.

In our VT core, the CP sends vector-fetch instructions to the VIU. For each vector fetch, the VIU creates a new *vector fragment* consisting of a program counter, initialized to the start address specified in the vector fetch, and an active µT bit mask, initialized to all active. The VIU then fetches and executes the corresponding sequential instruction stream across all active µTs, sending a vector µop plus active µT mask to the vector lanes for each instruction. The VIU handles a branch instruction by issuing a compare µop to one of the VFUs, which then produces a branch-resolution bit mask. If the mask is all zeros or ones, the VIU continues fetching scalar instructions along the fall-through or taken path. Otherwise, the µTs have diverged and so the VIU splits the current fragment into two fragments representing the µTs on the fall-through and taken paths, and continues to execute the fall-through fragment while placing the taken fragment in a *pending vector fragment buffer* (PVFB). The µTs can repeatedly diverge, creating new fragments, until there is only one µT per fragment. The current fragment finishes when it executes a `stop` instruction. The VIU then selects another vector fragment from the PVFB for execution. Once the PVFB is empty, indicating that all the µTs have stopped executing, the VIU can begin processing the next vector-fetch instruction.

Our library also includes **blocking and non-blocking cache** components with a rich set of parameters: cache type (instruction/data), access port width, refill port width, cache line size, total capacity, and associativity. For non-blocking caches, additional parameters include the number of miss-status-handling registers (MSHR) and the number of secondary misses per MSHR.

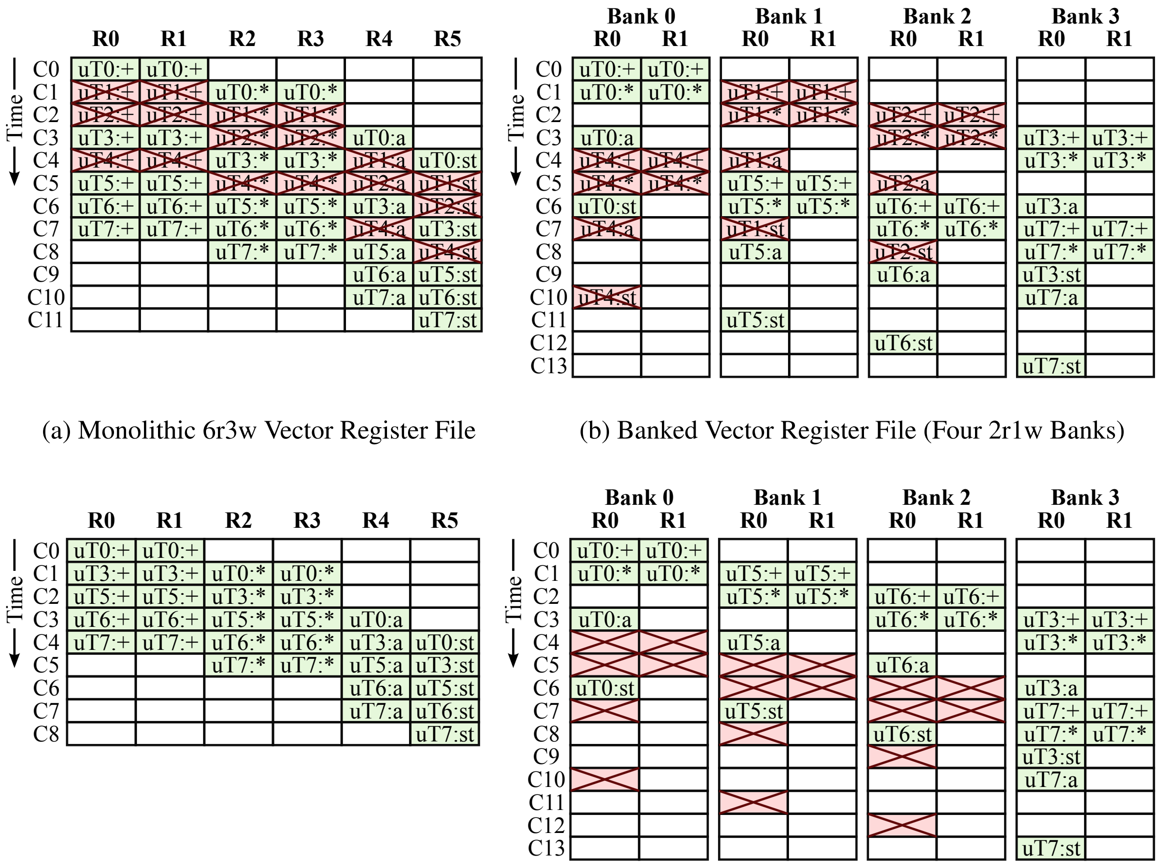
3.2 Constructing Tiles

MIMD cores combine a scalar integer core with integer and floating-point long-latency functional units, and support from one to eight µTs per core. **Vector cores** use a single-threaded scalar integer core as the CP connected to either a vector-SIMD or VT VIU, with one or more vector lanes and a VMU. To save area, the CP shares long-latency functional units with the vector lane, as in the Cray-1 [28].

We constructed two tile types: **multi-core tiles** consist of four MIMD (Figure 5(b)) or single-lane vector cores (Figure 5(c)), while **multi-lane tiles** consist of a single CP connected to a four-lane vector unit (Figure 5(d)). All tiles have the same number of long-latency functional units. Each tile includes a shared 64-KB four-bank data cache (8-way set-associative, 8 MSHRs, 4 secondary misses per MSHR), interleaved by 64-byte cache line. Request and response arbiters and crossbars manage communication between the cache banks and cores (or lanes). Each CP has a 16-KB private instruction cache and each VT VIU has a 2-KB vector instruction cache. Hence the overall instruction cache capacity (and area) is much larger in multi-core (64–72 KB) as compared to multi-lane (16–18 KB) tiles.

3.3 Microarchitectural Optimizations: Banking and Density-Time Execution

We explored a series of microarchitectural optimizations to improve performance, area, and energy efficiency of our baseline vector-SIMD and VT cores. The first was using a conventional **banked vector register file** to reduce area and energy (see Figure 6(b)). While a monolithic 6r3w register file simplifies vector lane design by allowing each VFU to access any element on any clock cycle, the high port count is expensive. Dividing the register file into four independent banks each with one write and two read ports significantly reduces register file area while keeping capacity constant. A crossbar connects banks to VFUs. The four 2r1w banks result in a greater aggregate bandwidth of



	R0	R1	R2	R3	R4	R5
C0	uT0:+	uT0:+				
C1	~~uT1:+~~	~~uT1:+~~	uT0:*	uT0:*		
C2	~~uT2:+~~	~~uT2:+~~	~~uT1:*~~	~~uT1:*~~		
C3	uT3:+	uT3:+	~~uT2:*~~	~~uT2:*~~	uT0:a	
C4	~~uT4:+~~	~~uT4:+~~	uT3:*	uT3:*	~~uT1:a~~	uT0:st
C5	uT5:+	uT5:+	~~uT4:*~~	~~uT4:*~~	~~uT2:a~~	~~uT1:st~~
C6	uT6:+	uT6:+	uT5:*	uT5:*	uT3:a	~~uT2:st~~
C7	uT7:+	uT7:+	uT6:*	uT6:*	~~uT4:a~~	uT3:st
C8			uT7:*	uT7:*	uT5:a	~~uT4:st~~
C9					uT6:a	uT5:st
C10					uT7:a	uT6:st
C11						uT7:st

(a) Monolithic 6r3w Vector Register File

	Bank 0		Bank 1		Bank 2		Bank 3	
	R0	R1	R0	R1	R0	R1	R0	R1
C0	uT0:+	uT0:+						
C1	uT0:*	uT0:*	~~uT1:+~~	~~uT1:+~~				
C2			~~uT1:*~~	~~uT1:*~~	~~uT2:+~~	~~uT2:+~~		
C3	uT0:a				~~uT2:*~~	~~uT2:*~~	uT3:+	uT3:+
C4	~~uT4:+~~	~~uT4:+~~	~~uT1:a~~				uT3:*	uT3:*
C5	~~uT4:*~~	~~uT4:*~~	uT5:+	uT5:+	~~uT2:a~~			
C6	uT0:st		uT5:*	uT5:*	uT6:+	uT6:+	uT3:a	
C7	~~uT4:a~~		~~uT1:st~~		uT6:*	uT6:*	uT7:+	uT7:+
C8			uT5:a		~~uT2:st~~		uT7:*	uT7:*
C9					uT6:a		uT3:st	
C10	~~uT4:st~~						uT7:a	
C11			uT5:st					
C12					uT6:st			
C13							uT7:st	

(b) Banked Vector Register File (Four 2r1w Banks)

	R0	R1	R2	R3	R4	R5
C0	uT0:+	uT0:+				
C1	uT3:+	uT3:+	uT0:*	uT0:*		
C2	uT5:+	uT5:+	uT3:*	uT3:*		
C3	uT6:+	uT6:+	uT5:*	uT5:*	uT0:a	
C4	uT7:+	uT7:+	uT6:*	uT6:*	uT3:a	uT0:st
C5			uT7:*	uT7:*	uT5:a	uT3:st
C6					uT6:a	uT5:st
C7					uT7:a	uT6:st
C8						uT7:st

(c) Monolithic VRF with Density-Time Execution

	Bank 0		Bank 1		Bank 2		Bank 3	
	R0	R1	R0	R1	R0	R1	R0	R1
C0	uT0:+	uT0:+						
C1	uT0:*	uT0:*	uT5:+	uT5:+				
C2			uT5:*	uT5:*	uT6:+	uT6:+		
C3	uT0:a				uT6:*	uT6:*	uT3:+	uT3:+
C4	☒	☒	uT5:a				uT3:*	uT3:*
C5	☒	☒	☒	☒	uT6:a			
C6	uT0:st		☒	☒	☒	☒	uT3:a	
C7	☒		uT5:st		☒	☒	uT7:+	uT7:+
C8			☒		uT6:st		uT7:*	uT7:*
C9					☒		uT3:st	
C10	☒						uT7:a	
C11			☒					
C12					☒			
C13							uT7:st	

(d) Banked VRF with Density-Time Execution

Figure 7: Example Read Port Scheduling – All four examples execute an add, a multiply, a µTAQ access, and a µTSDQ access µop. A hardware vector length of eight, and an active µT mask of `11101001` is assumed. VRF = vector register file, R = read port, C = cycle, + = addition, * = multiplication, a = µTAQ access, st = µTSDQ access.

eight read and four write ports. We take advantage of by adding a third VAU (VAU2) to the vector lane and rearranging the assignment of functional units to VAUs.

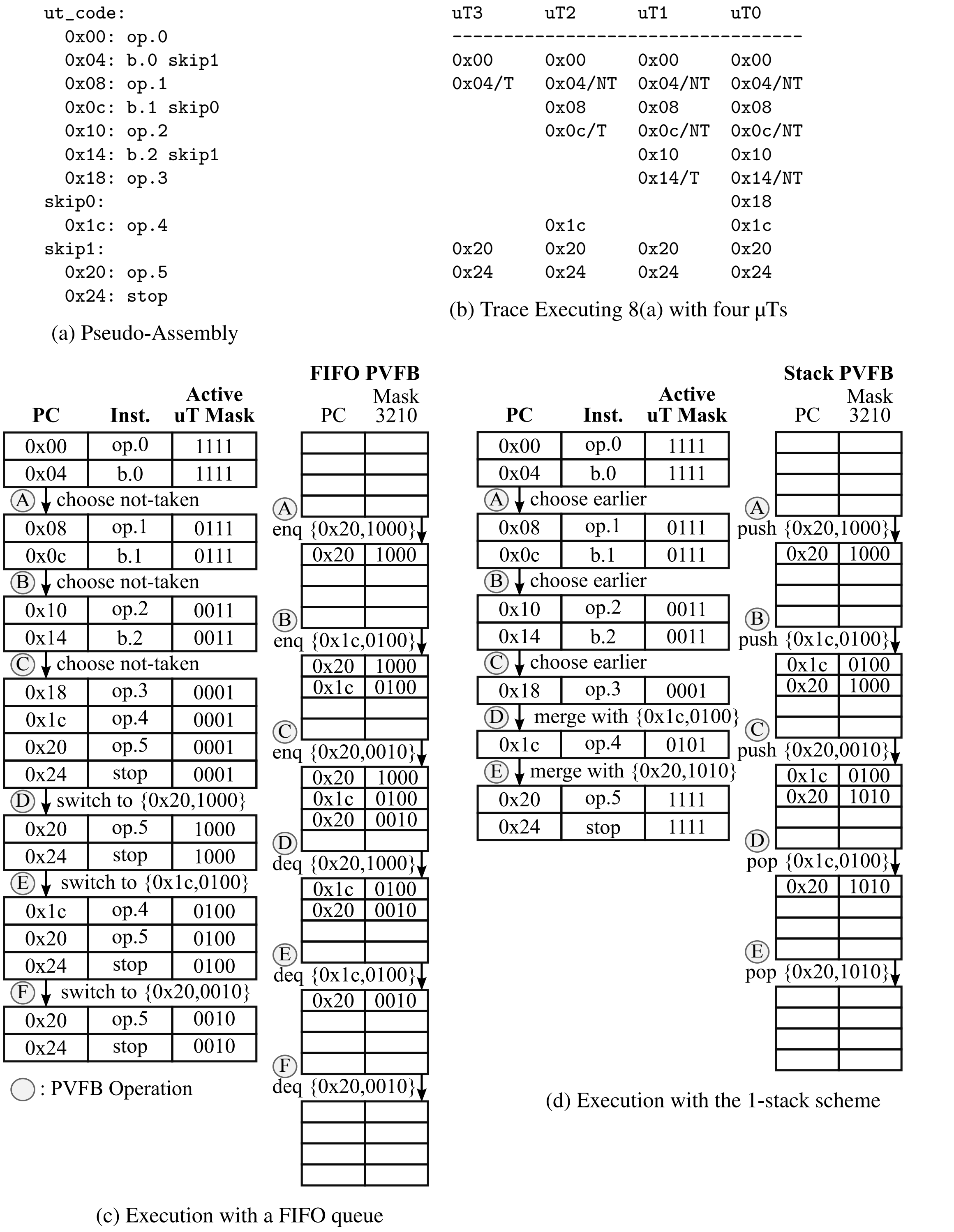
Figure 7(a) illustrates an example read port scheduling with a monolithic 6r3w vector register file. The add µop is issued at cycle 0 to use read ports R0 and R1 for eight cycles. The multiply µop is scheduled to access its operands with R2 and R3 starting at cycle 1. µTAQ and µTSDQ access µops for a µT store instruction are done through R4 and R5 at cycle 3 and 4 respectively. Figure 7(b) illustrates how the read port scheduling changes with vector register file banking. Registers within a µT are co-located within a bank, and µTs are striped across banks. As a VFU sequencer iterates through the µTs in a vector, it accesses a new bank on each clock cycle. The VIU must schedule vector µops to prevent bank conflicts, where two VFUs try to access the same bank on the same clock cycle. Note that the µTSDQ access cannot be scheduled at cycle 4 because of a bank conflict with the add operation. The operation is instead scheduled at cycle 6, resulting a total of 14 cycles to finish, which is 2 cycles longer compared to 12 cycles with a monolithic vector register file. Read ports are simply disabled for inactive µTs (µT1, µT2, and µT4) in both cases.

We developed another optimization for the banked design, which removes integer units from the VAUs and instead adds four **per-bank integer ALUs** directly connected to the read and write ports of each bank, bypassing the crossbar (see Figure 6(b)). This saves energy, and also helps performance by avoiding structural hazards and increasing peak integer throughput to four integer VAUs. The area cost of the extra ALUs is small relative to the size of the register file.

We also investigated **density-time execution** [30] to improve vector performance on irregular codes. The baseline vector machine takes time proportional to the vector length for each vector instruction, regardless of the number of inactive µTs. For example, if the hardware vector length is 8, it will take 8 cycles to execute a vector instruction, even if only five µTs are active (Figure 7(a)). Codes with highly irregular control flow often cause significant divergence between the µTs, splintering a vector into many fragments of only a few active µTs each. Density-time improves vector execution efficiency by "compressing" the vector fragment and only spending cycles on active µTs. With density-time execution in Figure 7(c), it only takes 5 cycles per vector fragment rather than 8 cycles. As illustrated in Figure 7(d), bank scheduling constraints reduce the effectiveness of density-time execution in banked register files. Rather than compressing inactive µTs from the whole vector, only inactive µTs from the same bank can be compressed. In Figure 7(d), µT3 and µT7 from the same bank are both active, resulting no actual cycle savings with density-time execution. Multi-lane machines have even greater constraints, as lanes must remain synchronized, so we only added density-time to single-lane machines.

3.4 Microarchitectural Optimizations: Dynamic Fragment Convergence

The PVFB in our baseline VT machine is a FIFO queue with no means to merge vector fragments. Figure 8(c) shows the execution of code in Figure 8(a) with the baseline FIFO queue. We assume a hardware vector length of four, and the outcome of branches `b.0`, `b.1`, and `b.2` for all four µTs are shown as part of the execution trace in Figure 8(b). The execution starts at the vector-fetched PC (`0x00`) with an active µT bit mask, initialized to all active (`1111`). Since `op.0` is not a branch instruction, the FIFO vector fragment selection policy chooses the PC+4 fragment, which consists of a PC (`0x04`) and the same µT mask (`1111`), for execution. Once branch `b.0` is resolved, the selection policy chooses to stash the taken fragment {`0x20,1000`} to the PVFB for later execution and execute the not-taken fragment {`0x08,0111`}. Vector fragments {`0x1c,0100`} and {`0x20,0010`} are next saved as a result of branches `b.1` and `b.2`. Once the current fragment {`0x24,0001`} encounters a `stop` instruction, the selection policy chooses to dequeue a fragment from the PVFB to execute. Note that once a vector becomes fragmented, those fragments will execute independently until all µTs execute a `stop` instruction, even when fragments have the same PC (`0x20`).



```
ut_code:
  0x00: op.0
  0x04: b.0 skip1
  0x08: op.1
  0x0c: b.1 skip0
  0x10: op.2
  0x14: b.2 skip1
  0x18: op.3
skip0:
  0x1c: op.4
skip1:
  0x20: op.5
  0x24: stop
```

(a) Pseudo-Assembly

```
uT3      uT2      uT1      uT0
----------------------------------
0x00     0x00     0x00     0x00
0x04/T   0x04/NT  0x04/NT  0x04/NT
         0x08     0x08     0x08
         0x0c/T   0x0c/NT  0x0c/NT
                  0x10     0x10
                  0x14/T   0x14/NT
                           0x18
         0x1c              0x1c
0x20     0x20     0x20     0x20
0x24     0x24     0x24     0x24
```

(b) Trace Executing 8(a) with four µTs

(c) Execution with a FIFO queue

(d) Execution with the 1-stack scheme

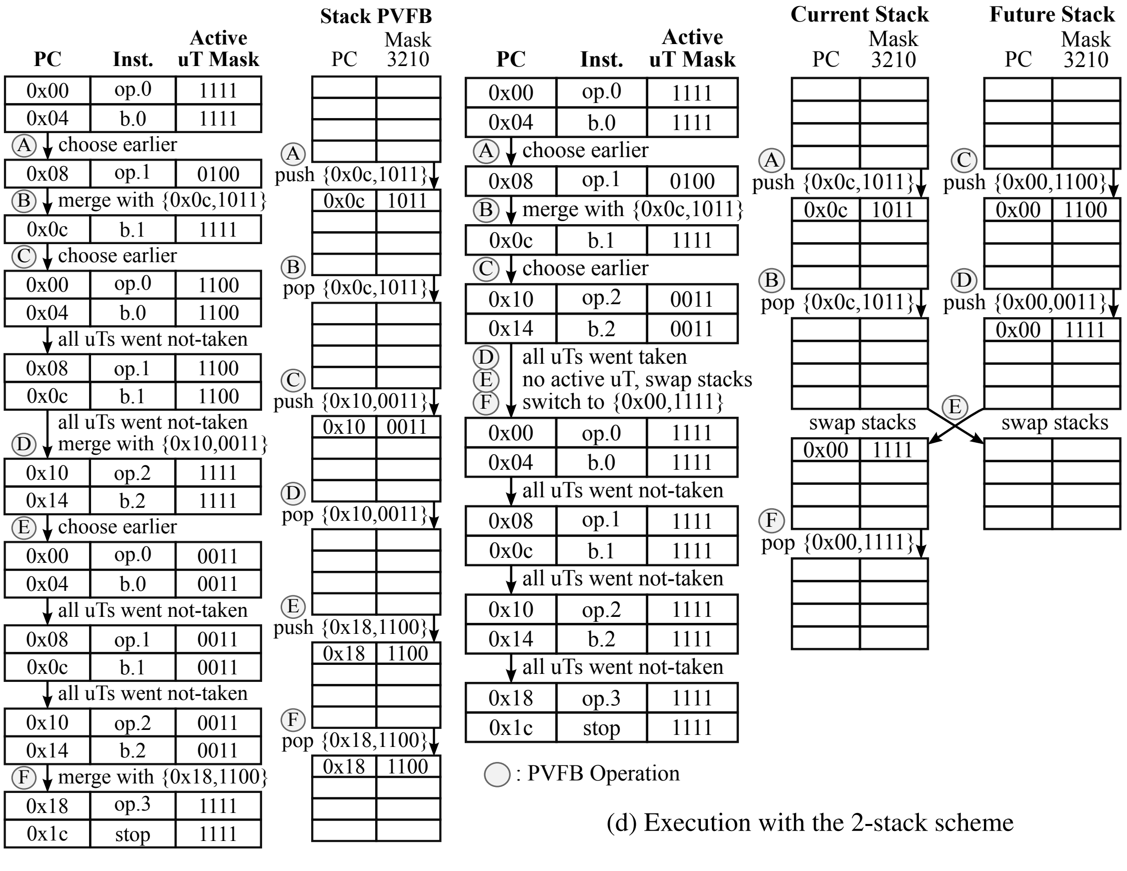
Figure 8: Executing Irregular DLP Code with Forward Branches Only – Example (a) pseudo-assembly, (b) trace (PCs are aligned to match the 1-stack scheduling), (c) execution diagram illustrating how the FIFO queue manages divergence, (d) execution diagram illustrating how the 1-stack scheme manages divergence. T = taken, NT = not-taken, PVFB = pending vector fragment buffer.

```
loop:
  0x00: op.0
  0x04: b.0 skip
  0x08: op.1
skip:
  0x0c: b.1 loop
  0x10: op.2
  0x14: b.2 loop
  0x18: op.3
  0x1c: stop
```

(a) Pseudo-Assembly

```
uT3      uT2      uT1      uT0
----------------------------------
0x00     0x00     0x00     0x00
0x04/T   0x04/NT  0x04/T   0x04/T
         0x08
0x0c/T   0x0c/T   0x0c/NT  0x0c/NT
                  0x10     0x10
                  0x14/T   0x14/T
0x00     0x00     0x00     0x00
0x04/NT  0x04/NT  0x04/NT  0x04/NT
0x08     0x08     0x08     0x08
0x0c/NT  0x0c/NT  0x0c/NT  0x0c/NT
0x10     0x10     0x10     0x10
0x14/NT  0x14/NT  0x14/NT  0x14/NT
0x18     0x18     0x18     0x18
0x1c     0x1c     0x1c     0x1c
```

(b) Trace Executing 9(a) with four µTs



(d) Execution with the 2-stack scheme

(c) Execution with the 1-stack scheme

Figure 9: Executing Irregular DLP Code with Backward Branches – Example (a) pseudo-assembly, (b) trace (PCs are aligned to match the 2-stack scheduling), (c) execution diagram illustrating how the 1-stack scheme manages divergence, (d) execution diagram illustrating how the 2-stack scheme manages divergence. T = taken, NT = not-taken, PVFB = pending vector fragment buffer.

We developed two schemes for VT machines to implement **dynamic fragment convergence** in the PVFB. When a new fragment is inserted into the PVFB, both schemes will attempt to dynamically merge the fragment with an existing fragment if their PCs match, OR-ing their active µT masks together. The challenge is to construct a fragment scheduling heuristic that maximizes opportunities for convergence by avoiding the execution of fragments which are later able to merge with other fragments in the PVFB.

Our first convergence scheme, called *1-stack*, organizes the PVFB as a stack with fragments sorted by PC address, with newly created fragments systolically insertion-sorted into the stack. The stack vector fragment selection policy always picks the fragment with the numerically smallest (earliest) PC among the taken and not-taken fragments, merging with the fragment at the top of the stack PVFB when possible. The intuition behind 1-stack is to favor fragments trailing behind in execution, giving them more chance to meet up with faster-moving fragments at a convergence point. The execution diagram in Figure 8(d) illustrates this efficient fragment scheduling. Note that when fragment {`0x1c,0100`} is pushed to the stack, the stack enforces PC ordering and keeps that fragment on the top. Also note fragment {`0x20,0010`} is merged with an existing fragment in the stack, which has the same PC `0x20`. Once PC `0x1c` is reached, the current fragment is merged with the fragment at the top of the stack {`0x1c,0100`}, resulting fragment {`0x1c,0101`} to execute. Note that the operation at PC `0x20` are now executed with all µTs active.

The 1-stack scheme performs reasonably well, but is sub-optimal for loops with multiple backwards branches. Fragments which first branch back for another loop iteration are treated as if they are behind slower fragments in the same iteration and race ahead. Suppose four µTs execute pseudo-assembly in Figure 9(a) and four µTs pick branch directions as shown in Figure 9(b). An execution diagram in Figure 9(c) illustrates this phenomenon. As shown in the diagram, µT2 and µT3 satisfies a loop condition such that `b.1` is taken. Since the 1-stack scheme favors the smallest PC, these µT fragments are executed to completion, only converging with the fragment at PC `0x10` on the last iteration. This scheduling reduces the number of active µTs per instruction execution (instructions at PC `0x00` to `0x14` are executed with an active µT mask `0011`) and does not yield the optimal execution for this type of code.

To solve this problem, our second scheme, called *2-stack*, divides the PVFB into two virtual stacks, one for fragments on the current iteration of a loop and another for fragments on a future iteration of a loop (Figure 9(d)). Fragments created from the forward branch ({`0x0c,1011`}) are pushed onto the current stack, while the fragments from backwards branches ({`0x00,1100`} and {`0x00,0011`}) are pushed onto the future stack. Note that the selection policy only pops fragments from the current stack. When the current stack empties, the current and future stacks are swapped (PVFB operation E). The 2-stack implementation is similar to the 1-stack implementation, but PCs saved in the PVFB have an extra bit in the MSB used in comparisons for ordering. This bit is set if the fragment being inserted into the PVFB was a backwards branch to prevent this fragment from being chosen until all fragments on the current iteration are executed. Once no fragments of the current iteration remain, this bit is toggled so that the next iteration's fragments become active candidates for selection. Implementing the 2-stack scheme in this way allows us to physically use only one stack, exploiting the fact that we will only ever use as many entries as there are µTs.

Note that the Maven VT design only uses dynamic information such as the PC of a fragment with no explicit static hints to aid fragment convergence as are believed to be used in SIMT architectures [23, 7]. The stack-based convergence scheme proposed in [33] and described in [7] uses immediate postdominators as explicit yield points to guide convergence in the warp scheduler. To know the immediate postdominator of a diverging branch, however, the control flow graph needs to be analyzed. [7] proposes dynamic warp formation to increase the utilization of the SIMD pipeline. Five scheduling polices were considered to maximize the number of active threads when dynamically forming a warp. Among the five schemes, the program counter (DPC) policy is similar to the 1-stack convergence scheme. The intuition behind two schemes are the same: the program counter is a good indicator of progress.

3.5 Microarchitectural Optimizations: Dynamic Memory Coalescer

The final optimization we considered is a **dynamic memory coalescer** for multi-lane VT vector units (Figure 10). During the execution of a µT load instruction, each lane may generate an individual memory address on each cycle. The memory coalescer compares the high-order bits of each memory address and combines matching requests. The low-order bits are stored as word and byte select bits alongside the load data buffer. The number of low-order bits should match the maximum size of the memory response in bytes. When memory responses arrive from the cache we use the

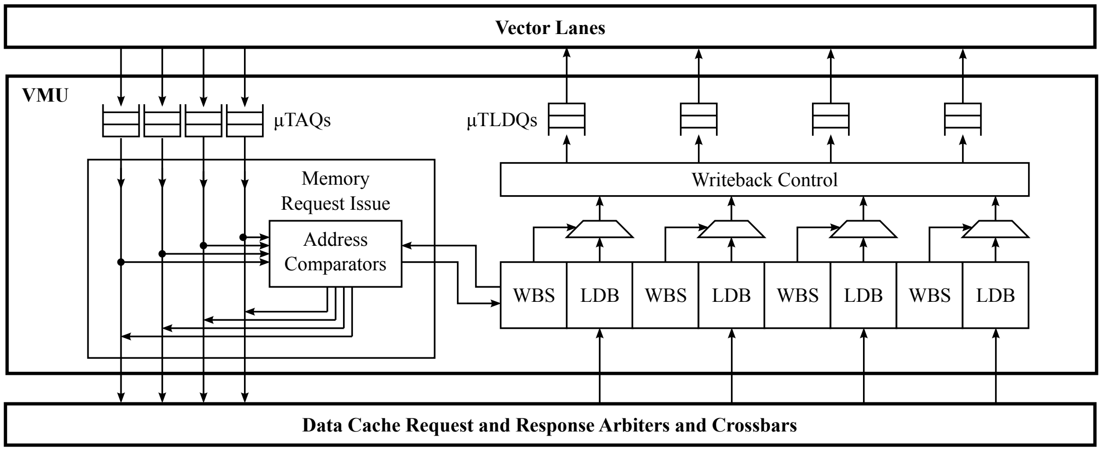


Figure 10: Memory Coalescer Microarchitecture – A memory coalescer for multi-lane VT tiles with four lanes. VMU = vector memory unit, µTAQ = µT address queue, µTLDQ = µT load data queue, WBS = word/byte select buffer, LDB = load data buffer.

word and byte select bits to select the correct portion of the response to write to each µT load data queue, which feed back to the vector lanes.

Suppose each address contains four low-order bits, and each data cache response may contain up to sixteen bytes. The four µT address queues issue the following load word requests: 0x001c, 0x0014, 0x0008, 0x0014. The address comparators identify that the first, second, and fourth requests may be coalesced. The second and fourth requests are disabled, while the first request is annotated with extra information that indicates that it is a coalesced request for the first, second, and fourth lanes. The third request is left untouched. At the same time, the address comparators write the word and byte select information to the word and byte select buffers. In this case the first, second, and fourth requests select the fourth, second, and second words of the sixteen-byte coalesced response, respectively. Notice how addresses need not be in ascending order and may even match in the low-order bits.

Additional control logic must be added to the data cache arbiter to correctly route memory responses. The arbiter must be able to write a response to multiple load data buffers but only if all buffers are ready to receive data. A conflict occurs when a coalesced response arrives at the same time a non-coalesced response and both wish to write to the same load data buffer. This introduces more complex control dependencies between vector lanes.

Dynamic memory coalescing can significantly help performance on codes that use µT loads to access memory addresses with a unit stride, as these would otherwise generate cache bank conflicts. Similarly, codes which use µT loads to access the same memory address also benefit. This effect diminishes with larger strides, as requests no longer occupy the same cache bank. Compared to vector memory operations, however, µT memory operations are still less efficient even with dynamic memory coalescing for code with regular memory access patterns. Vector memory operations have the benefit of access-execute decoupling because addresses can be computed independently of non-memory operations. Vector memory operations may also statically determine whether accesses can be coalesced into a single response if the stride is known statically. On the other hand, dynamic memory coalescing for µT memory coalescing still improves the efficiency of irregular access patterns with high spatial locality. We compare the effectiveness of purely using µT memory operations with memory coalescing to using vector memory operations in Section 5.3.

4 Evaluation Framework

This section describes the hardware and software infrastructure used to evaluate the various microarchitectural options introduced in the previous section, and also outlines the specific configurations, microbenchmarks, and application kernels used in our evaluation. Figure 11 illustrates the overall process of compiling C++ application code into a binary, generating a VLSI layout from an RTL description of a particular machine configuration, and simulating the execution of the application to extract area, performance, and power statistics.

4.1 Hardware Toolflow

We use our own machine definition files to instantiate and compose the parameterized Verilog RTL into a full model for each tile configuration. We targeted TSMC's 65-nm GPLUSTC process using a Synposys-based ASIC toolflow: VCS for simulation, Design Compiler for synthesis, and IC Compiler for place-and-route (PAR). RTL simulation produces cycle counts. PAR produces cycle time and area estimates. The steps in this process are depicted on the right hand side of Figure 11. Table 1 lists IC Compiler post-PAR power estimates based on a uniform statistical probability of

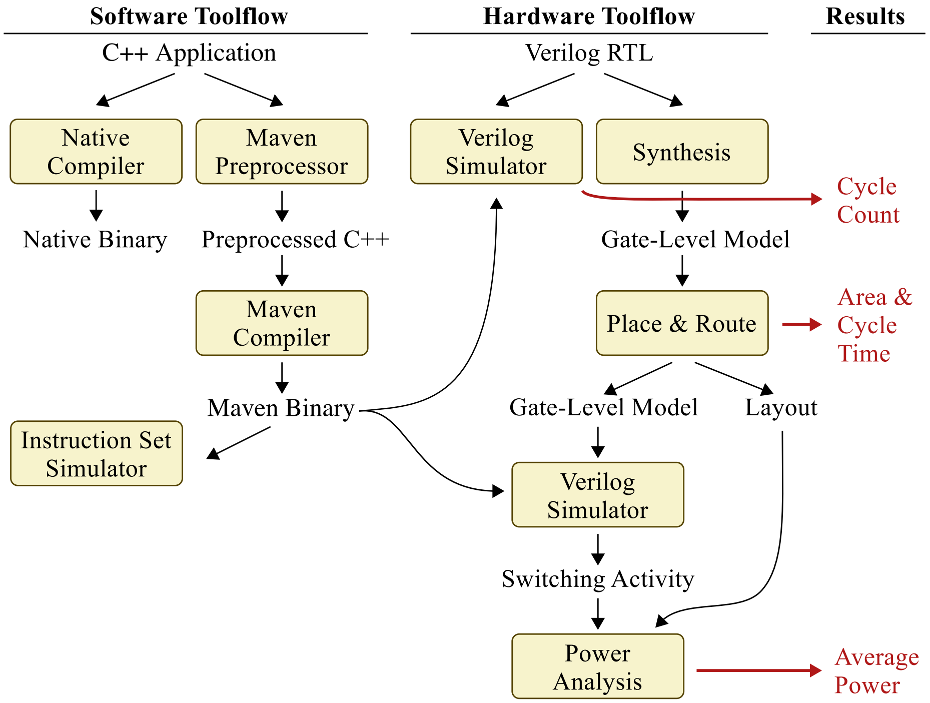


Figure 11: Evaluation Framework – The software toolflow allows C++ applications to be compiled either natively or for Maven, while the hardware toolflow transforms the Verilog RTL for a data-parallel tile into actual layout. From this toolflow we can accurately measure area, performance (1/cycle count × cycle time), and energy (average power × cycle count × cycle time). (from [2])

		Per Core		Peak Throughput		Power		Total	Cycle
	Num	Num	Num	Arith	Mem	Statistical	Simulated	Area	Time
Configuration	Cores	Regs	µTs	(ops/cyc)	(elm/cyc)	(mW)	(mW)	(mm^2)	(ns)
mimd-c4r32§	4	32	4	4	4	149	137 – 181	3.7	1.10
mimd-c4r64§	4	64	8	4	4	216	130 – 247	4.0	1.13
mimd-c4r128§	4	128	16	4	4	242	124 – 261	4.2	1.19
mimd-c4r256§	4	256	32	4	4	299	221 – 298	4.7	1.27

		Per Core		Per Lane	Peak Throughput		Power		Total	Cycle
	Num	Num	Max vlen	Num	Arith	Mem	Statistical	Simulated	Area	Time
Configuration	Cores	Lanes	Range	Regs	(ops/cyc)	(elm/cyc)	(mW)	(mW)	(mm^2)	(ns)
vsimd-c4v1r32	4	1	1 – 8	32	4c + 8v	4l + 4s	349	154 – 273	4.8	1.23
vsimd-c4v1r64	4	1	2 – 16	64	4c + 8v	4l + 4s	352	176 – 278	4.9	1.28
vsimd-c4v1r128	4	1	4 – 32	128	4c + 8v	4l + 4s	367	194 – 283	5.2	1.30
vsimd-c4v1r256	4	1	8 – 32	256	4c + 8v	4l + 4s	384	207 – 302	6.0	1.49
vsimd-c4v1r256+bi§	4	1	8 – 32	256	4c + 16v	4l + 4s	396	213 – 331	5.6	1.37
vsimd-c1v4r256+bi§	1	4	32 – 128	256	1c + 16v	4l + 4s	224	137 – 252	3.9	1.46
vt-c4v1r32	4	1	1 – 8	32	4c + 8v	4l + 4s	384	136 – 248	5.1	1.27
vt-c4v1r64	4	1	2 – 16	64	4c + 8v	4l + 4s	391	151 – 252	5.3	1.32
vt-c4v1r128	4	1	4 – 32	128	4c + 8v	4l + 4s	401	152 – 274	5.6	1.30
vt-c4v1r256	4	1	8 – 32	256	4c + 8v	4l + 4s	428	162 – 318	6.3	1.47
vt-c4v1r128+b	4	1	4 – 32	128	4c + 8v	4l + 4s	396	148 – 254	5.3	1.27
vt-c4v1r256+b	4	1	8 – 32	256	4c + 8v	4l + 4s	404	147 – 271	5.6	1.31
vt-c4v1r128+bi	4	1	4 – 32	128	4c + 16v	4l + 4s	439	174 – 278	5.6	1.31
vt-c4v1r256+bi	4	1	8 – 32	256	4c + 16v	4l + 4s	445	172 – 298	5.9	1.32
vt-c4v1r256+bi	4	1	8 – 32	256	4c + 16v	4l + 4s	445	172 – 298	5.9	1.32
vt-c4v1r256+bi+d	4	1	8 – 32	256	4c + 16v	4l + 4s	449	196 – 297	6.0	1.41
vt-c4v1r256+bi+1s	4	1	8 – 32	256	4c + 16v	4l + 4s	408	193 – 289	5.8	1.39
vt-c4v1r256+bi+1s+d	4	1	8 – 32	256	4c + 16v	4l + 4s	409	213 – 293	5.8	1.41
vt-c4v1r256+bi+2s	4	1	8 – 32	256	4c + 16v	4l + 4s	409	225 – 304	5.9	1.32
vt-c4v1r256+bi+2s+d§	4	1	8 – 32	256	4c + 16v	4l + 4s	410	168 – 300	5.9	1.36
vt-c1v4r256+bi+2s§	1	4	32	256	1c + 16v	4l + 4s	205	111 – 167	3.9	1.42
vt-c1v4r256+bi+2s+mc	1	4	32	256	1c + 16v	4l + 4s	223	118 – 173	4.0	1.42

Table 1: Subset of Evaluated Tile Configurations – Multi-core and multi-lane tiles for MIMD, vector-SIMD, and VT patterns. Configurations with § are used in Section 5.4. *statistical power* column is from post-PAR; *simulated power* column shows min/max across all gate-level simulations; *configuration* column: b = banked, bi = banked+int, 2s = 2-stack, d = density-time, mc = memory coalescing; *num µTs* column is the number of µTs supported with the default of 32 registers/µT; *arith* column: xc + yv = x CP ops and y vector unit ops per cycle; *mem* column: xl + ys = x load elements and y store elements per cycle. (adapted from [16])

bit transitions, and the range of powers reported via PrimeTime across all benchmarks when using bit-accurate activity for every net simulated on a back-annotated post-PAR gate-level model. The inaccuracy of the IC Compiler estimates and the large variance in power across benchmarks motivated us to use only detailed gate-level simulation for energy estimates.

Complex functional units (e.g., floating-point) are implemented using Synopsys DesignWare library components, with automatic register retiming to generate pipelined units satisfying our cycle-time constraint. The resulting latencies were: integer multiplier (3) and divider (12), floating-point adder (3), multiplier (3), divider (7), and square-root unit (10).

We did not have access to a memory compiler for our target process, so we model SRAMs and caches by creating abstracted "black-box" modules, with area, timing, and power models suitable for use by the CAD tools. We used CACTI [20] to explore a range of possible implementations and chose one that satisfied our cycle-time requirement while consuming minimal power and area. We compared CACTI's predicted parameter values to the SRAM datasheet for our target process and found them to be reasonably close. Cache behavior is modeled by a cache simulator (written in C++) that interfaces with the ports of the cache modules. The latency between a cache-line refill request and response was set at 50 cycles. We specify the dimensions of the target ASIC and the placement and orientation of the large black-box modules. The rest of the design (including register files) was implemented using standard cells, all automatically placed.

4.2 Tile Configurations

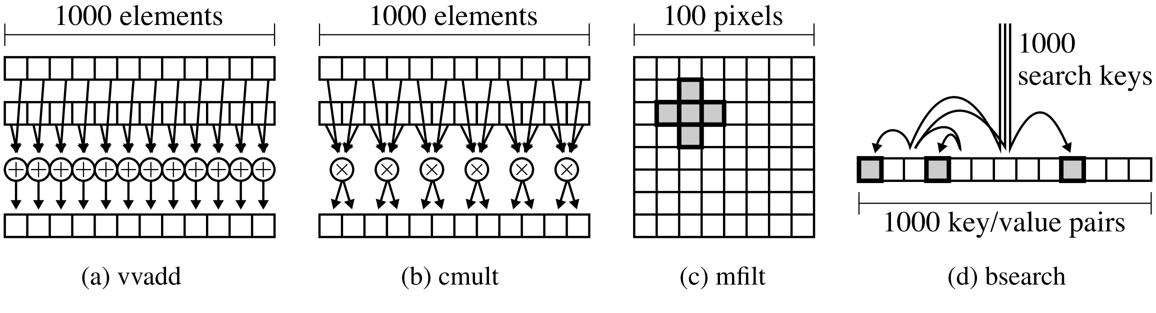
We evaluated hundreds of tile configurations using our hardware toolflow. For this paper, we focus on 25 representative configurations (see Table 1). We name configurations beginning with a prefix designating the style of machine, followed by the number of cores (c), the number of lanes (v), and physical registers (r) per core or lane. The suffix denotes various microarchitectural optimizations: b = banked register file, bi = banked register file with extra integer ALUs, 1s = 1-stack convergence scheme, 2s = 2-stack convergence scheme, d = density-time execution, mc = memory coalescing. Each type of core is implemented with 32, 64, 128, and 256 physical registers. For the MIMD cores, this corresponds to 1, 2, 4, and 8 µTs respectively. For the vector cores, the maximum hardware vector length is determined by the size of the vector register file and the number of registers assigned to each µT (4–32). The vector length is capped at 32 for all VT designs, even though some configurations (i.e., 256 physical registers with 4 registers per µT) could theoretically support longer vector lengths. We imposed this limitation because some structures in the VT machines (such as the PVFB) scale quadratically in area with respect to the maximum number of active µTs. Banked vector register file designs are only implemented for 128 and 256 physical registers.

4.3 Microbenchmarks & Application Kernels

We selected four microbenchmarks and six larger application kernels to represent the spectrum from regular to irregular DLP.

Figure 12 illustrates the four microbenchmarks. The *vvadd* microbenchmark performs a 1000-element vector-vector floating-point addition and is the simplest example of regular DLP. The *cmult* microbenchmark performs a 1000-element vector-vector floating-point complex multiplication, and illustrates regular DLP with additional computational density and strided accesses to the arrays of complex number objects. The *mfilt* microbenchmark convolves a five-element filter kernel with a 100×100 gray-scale image under a separate mask image. It uses a regular memory access pattern and irregular control flow. Each iteration of the loop checks whether a pixel in a mask image is selected and only performs the convolution for selected pixels. The *bsearch* microbenchmark uses a binary search algorithm to perform 1000 look-ups into a sorted array of 1000 key-value pairs. This microbenchmark exhibits highly irregular DLP with two nested loops: an outer `for` loop over the search keys and an inner `while` loop implementing a binary search for finding the key. We include two VT implementations: one (*bsearch*) uses branches to handle intra-iteration control flow, while the second (*bsearch-cmv*) uses conditional move assembly instructions explicitly inserted by the programmer.

The *viterbi* kernel decodes frames of convolutionally encoded data using the Viterbi algorithm. Iterative calculation of survivor paths and their accumulated error are parallelized across paths. Each µT performs an add-compare-select butterfly operation to compute the error for two paths simultaneously, which requires unpredictable accesses to a lookup table. The *rsort* kernel performs an incremental radix sort on an array of integers. During each iteration, individual µTs build local histograms of the data, and then a parallel reduction is performed to determine the mapping to a global destination array. Atomic memory operations are necessary to build the global histogram structure. The *kmeans* kernel implements the k-means clustering algorithm. It classifies a collection of objects, each with some number of features, into a set of clusters through an iterative process.



(a) vvadd (b) cmult (c) mfilt (d) bsearch

Figure 12: Microbenchmarks – Four microbenchmarks are used to evaluate the various architectural design patterns. (from [2])

		Control Thread			Microthread									
	Name	**vf**	**vec ld**	**vec st**	**int**	**fp**	**ld**	**st**	**amo**	**br**	**cmv**	**tot**	**loop**	**nregs**
µbmarks	vvadd	1	2u	2u		1						2		4
	cmult	1	4s	2s	1	6						8		4
	mfilt	1	6u	1u	10					1		12		13
	bsearch-cmv	1	1u	1u	17		2			1	4	25	×	13
	bsearch	1	1u	1u	15		3			5	1	26	×	10
App Kernels	viterbi	3	3u	1u, 4s	21						3	35		8
	rsort	3	3u, 2s	3u	14		2	3	1			25		11
	kmeans	9	7u, 3s	5u, 1s	12	6	2	2	1	1	2	40		8
	dither	1	4u, 1s	5u, 1s	13					1	2	24		8
	physics	4	6u, 12s	1u, 9s	5	56	24	4		16		132	×	32
	strsearch	3	5u	1u	35		9	5		15	2	96	×	14

Table 2: Microbenchmark and Application Kernel Statistics for VT Implementation – Number of instructions listed by type. *vec ld/st* columns indicate numbers of both unit-stride (u) and strided (s) accesses; *loop* column indicates an inner loop within the vector-fetched block; *nregs* column indicates number of registers a vector-fetched block requires. (adapted from [16])

		App Vlen Quartiles				Active µT Distribution (%)			
	Name	**1q**	**2q**	**3q**	**max**	**1–25**	**26–50**	**51–75**	**76–100**
µbmarks	vvadd	1000	1000	1000	1000				100.0
	cmult	1000	1000	1000	1000				100.0
	mfilt	1000	1000	1000	1000	3.6	4.1	9.4	82.9
	bsearch-cmv	1000	1000	1000	1000	1.0	3.3	5.8	89.9
	bsearch	1000	1000	1000	1000	77.6	12.4	5.1	4.8
	bsearch (w/ 1-stack)					23.8	23.4	11.7	41.0
	bsearch (w/ 2-stack)					10.1	26.8	49.2	13.9
App Kernels	viterbi	32	32	32	32				100.0
	rsort	1000	1000	1000	1000				100.0
	kmeans	100	100	100	100				100.0
	dither	72	143	185	386	0.2	0.4	0.7	98.7
	physics	7	16	44	917	6.9	15.0	28.7	49.3
	physics (w/ 2-stack)					4.7	13.1	28.3	53.9
	strsearch	57	57	57	57	57.5	25.5	16.9	0.1
	strsearch (w/ 2-stack)					14.8	30.5	54.7	0.1

Table 3: Microbenchmark and Application Kernel Data-Dependent Statistics – Application vector length distribution indicates number of µTs used per stripmine loop assuming infinite resources. Distribution of active µTs with a FIFO PVFB unless otherwise specified in *name* column. Each section sorted from most regular to most irregular. (adapted from [16])

Assignment of objects to clusters is parallelized across objects. The minimum distance between an object and each cluster is computed independently by each µT and an atomic memory operation updates a shared data structure. Cluster centers are recomputed in parallel using one µT per cluster. The *dither* kernel generates a black and white image from a gray-scale image using Floyd-Steinberg dithering. Work is parallelized across the diagonals of the image, so that each µT works on a subset of the diagonal. A data-dependent conditional allows µTs to skip work if an input pixel is white.

The *physics* kernel performs a simple Newtonian physics simulation with object collision detection. Each µT is responsible for intersection detection, motion variable computation, and location calculation for a single object. Oct-trees are also generated in parallel. The *strsearch* kernel implements the Knuth-Morris-Pratt algorithm to search a collection of strings for the presence of substrings. The search is parallelized by having all µTs search for the same substrings in different streams. The DFAs used to model substring-matching state machines are also generated in parallel.

Table 2 reports the instruction counts and Table 3 shows the application vector length and distribution of active µTs for the VT implementations of two representative microbenchmarks and the six application kernels. *viterbi* is an example of regular DLP with known memory access patterns. *rsort*, *kmeans*, and *dither* all exhibit mild control flow conditionals with more irregular memory access patterns. *physics* and *strsearch* exhibits characteristics of highly irregular DLP code: loops with data-dependent exit conditionals, highly irregular data access patterns, and many conditional branches.

4.4 Programming Methodology

Past accelerators usually relied on hand-coded assembly or compilers that automatically extract DLP from high-level programming languages [1, 5, 9]. Recently there has been a renewed interest in explicitly data-parallel programming methodologies [22, 24, 4], where the programmer writes code for the HT and annotates data-parallel tasks to be executed in parallel on all µTs. We developed a similar explicit-DLP C++ programming environment for Maven. Supporting such a programming style for our VT implementation was made relatively easy by the use of a single ISA for the CT and µTs. The software toolflow is illustrated on the left hand side of Figure 11. Note that to aid in debugging, we produce a program binary that runs natively on our development platform along with the binary for our target machine architecture. For all systems, a simple proxy kernel running on the cores supports basic system calls by communicating with an application server running on the host. More details about the programming methodology can be found in [2].

To bring up a reasonable compiler infrastructure with limited resources, we attempted to leverage a standard scalar compiler as much as possible. We started with a recent version of the GNU assembler, linker, and C++/newlib compiler (version 4.4.1), which all contain support for the basic MIPS32 instruction set. We then modified the assembler to support the new Maven scalar and vector instructions.

Most of our efforts went into modifying the compiler back-end. We first unified the integer and floating-point register space. Instruction templates were added for the new divide and remainder instructions, since the Maven ISA lacks high and low registers. Branch delay slots were also removed. A new vector register space and the corresponding instruction templates required for register allocation were added. Some of these modifications were able to leverage the GNU C++ compiler's built-in support for fixed-length subword-SIMD instructions. Compiler intrinsics were added for some of the vector instructions to enable software to explicitly generate these instructions and for the compiler to understand their semantics. The control thread and the µTs have different performance characteristics, so we leveraged the compiler instruction scheduling framework to create two pipeline models for Maven: one optimized for control threads and the other optimized for µTs.

There were relatively few modifications necessary to the compiler front-end. We used the GNU C++ compiler's function attribute framework to add new attributes denoting functions meant to run on the µTs for performance tuning. We were able to leverage the GNU C++ compiler's built-in support for fixed-length subword-SIMD instructions to create true C++ vector types.

Figure 13 illustrates how the irregular DLP loop in Figure 1(d) might be coded for various architectural patterns. Figure 13(a) illustrates how the MIMD architectural pattern is programmed. The VT architectural pattern can be programmed much like a SIMT machine (Figure 13(b)), in which case programming is relatively easy but execution is less efficient. The VT pattern also allows programmers to expend more effort in optimizing their code to hoist structured memory accesses out as vector memory operations, and to use scalar operations (Figure 13(c)), which provides more efficiency than is possible with a pure SIMT machine. Figure 13(d) shows how irregular DLP is mapped to the Vector-SIMD architectural pattern. Finally, Figure 13(e) describes how we leverage the MIMD programming model to target a multi-core VT machine.

For MIMD, a "master" µT on the multithreaded core is responsible for spawning the work on the other remaining "worker" µTs. To support this, we first modify the proxy kernel to support multiple threads of execution and then build a lightweight user-level threading library called *bthreads*, which stands for "bare threads", on top of the proxy-kernel threads. There is one bthread for each underlying hardware µT context. The application is responsible for managing scheduling. Spawning work is done with a `BTHREAD_PARALLEL_RANGE` macro as shown in Figure 13(a). This macro automatically partitions the input dataset's linear index range, creates a separate function, spawns the function onto each µT, passes in arguments through memory, and waits for the threads to finish. Each thread does the work from `range.begin()` to `range.end()` where `range` is defined by the preprocessor macro to be different for each thread. Line 4 specifies the total number of elements to be distributed to the worker µTs and a list of C++ variables that should be marshalled for each worker µT. The final argument to the macro is the work to be done by each µT (lines 5–11). The work can contain any of the other architectural design pattern programming methodologies to enable mapping an application to multiple cores. As illustrated in Figure 13(e), by calling the `idlp_vt` function inside the body of a `BTHREAD_PARALLEL_RANGE` macro we can use the bthreads library to distribute work amongst multiple VT cores. This programming model is similar to the OpenMP programming framework [25], where the programmer explicitly annotates the source code with pragmas to mark parallel loops.

Figure 13(b) illustrates the Maven VT programming environment used in SIMT fashion. The `config` function on line 4 takes two arguments: the number of required µT registers and the ap-

```
void idlp_mimd( int c[], int a[], int b[],
                int n, int x )
{
  BTHREAD_PARALLEL_RANGE( n, (c,a,b,x),
  ({
    for ( int i = range.begin();
              i < range.end(); i++ ) {
      if ( a[i] > 0 )
        c[i] = x * a[i] + b[i];
    }
  }));
}
```

(a) MIMD

```
void idlp_vt_simt( int c[], int a[], int b[],
                   int n, int x )
{
  int blocksz = vt::config( 11, n );
  int nblocks = ( size + blocksz - 1 ) / blocksz;

  vt::HardwareVector<int*> vap(a),vbp(b),vcp(c);
  vt::HardwareVector<int*> vxp(&x);
  vt::HardwareVector<int> vsize(size);
  vt::HardwareVector<int> vbsz(blocksz);

  for ( int bidx = 0; bidx < nblocks; bidx++ ) {
    vt::HardwareVector<int> vbidx(bidx);

    VT_VFETCH( (vxp,vap,vbp,vcp,vsize,vbsz,vbidx),
    ({
      int idx = vbidx * vbsz + vt::get_utidx();
      if ( idx < vsize ) {
        if ( vap[idx] > 0 )
          vcp[idx] = (*vxp) * vap[idx] + vbp[idx];
      }
    }));
  }
  vt::sync_cv(); // vector memory fence
}
```

(b) VT in SIMT fashion

```
void idlp_vt( int c[], int a[], int b[],
              int n, int x )
{
  int vlen = vt::config( 7, n );
  vt::HardwareVector<int> vx(x);

  for ( int i = 0; i < n; i += vlen ) {
    vlen = vt::set_vlen(n-i); // stripmining

    vt::HardwareVector<int*> vcp(&c[i]);
    vt::HardwareVector<int> va, vb;

    va.load(&a[i]); // unit-stride vector load
    vb.load(&b[i]); // unit-stride vector load

    VT_VFETCH( (vcp,vx,va,vb),
    ({
      if ( va > 0 )
        vcp[vt::get_utidx()] = vx * va + vb;
    }));
  }
  vt::sync_cv(); // vector memory fence
}
```

(c) VT

```
void idlp_vsimd( int c[], int a[], int b[],
                 int n, int x )
{
  int vlen = vt::config( 9, n );
  vt::HardwareVector<int> vx(x);

  for ( int i = 0; i < n; i += vlen ) {
    vlen = vt::set_vlen(n-i); // stripmining

    vt::HardwareVector<int> vctmp;
    vt::HardwareVector<int> va, vb, vc;

    va.load(&a[i]); // unit-stride vector load
    vb.load(&b[i]); // unit-stride vector load
    vc.load(&c[i]); // unit-stride vector load

    asm (
      "slt.f.vv $flag1, $vzero, %[va]    \n"
      "mul.vv  %[vctmp], %[vx], %[va]    \n"
      "add.vv  %[vctmp], %[vctmp], %[vb] \n"
      "mov.vv  %[vc], %[vctmp], $flag1   \n"
      : [vctmp] "=&Z"(vctmp)        // outputs
      : [va] "Z"(va), [vx] "Z"(vx), // inputs
        [vb] "Z"(vb), [vc] "Z"(vc)
    );

    vc.store(&c[i]); // unit-stride vector store
  }
  vt::sync_cv(); // vector memory fence
}
```

(d) Vector-SIMD

```
void idlp_mc_vt( int c[], int a[], int b[],
                 int n, int x )
{
  BTHREAD_PARALLEL_RANGE( n, (c,a,b,x),
  ({
    idlp_vt( &c[range.begin()],
             &a[range.begin()],
             &b[range.begin()],
             range.size(), x );
  }));
}
```

(e) Multi-core VT

Figure 13: Irregular DLP Example using Maven Programming Methodology – Code corresponds to the loop in Figure 1(d). Roughly, code (a) compiles to assembly in Figure 3(a), code (c) compiles to assembly in Figure 3(d), and code (d) compiles to assembly in Figure 3(b). (adapted from [16] and [2])

plication vector. This function returns the actual number of µTs supported by the hardware (block size), which is used to calculate the number of µT blocks on line 5. Lines 7–8 copy array base pointers and address of x, and lines 9–10,13 copy size, block size, and block index into all µTs. A for loop on line 12 emulates multiple µT blocks mapped to the same core. The VT_VFETCH macro on lines 15–22 takes two arguments: a list of hardware vectors, and the actual code, which should be executed on each µT. The code within the vector-fetched block specifies what operations to perform on each element of the hardware vectors. This means that the C++ type of a hardware vector is different inside versus outside the vector-fetched block. Outside the block, a hardware vector represents a vector of elements and has type HardwareVector<T> (e.g., vsize on line 9 has type HardwareVector<int>), but inside the block, a hardware "vector" now actually represents a *single* element and has type T (e.g., vsize on lines 18 has type int). Code within a vector-fetched block can include almost any C++ language feature including stack allocated variables, object instantiation, templates, conditionals (if, switch), and loops (for, while). The primary restrictions are that a vector-fetched block cannot use C++ exceptions nor make any system calls. After calculating its own index on line 15, a conditional on line 16 checks to make sure the index is not greater than the array size. Lines 19–20 contain the actual work. Line 24 performs a memory fence to ensure that all results are visible in memory before returning from the function. Note that this is similar to the CUDA programming methodology [22]. The address calculation on line 17 (think vbidx as blockIdx.x, vbsz as blockDim.x, and vt::get_utidx() as threadIdx.x) and the conditional branch on line 18 closely follows the CUDA programming practice.

Figure 13(c) illustrates how the code looks like after spending more effort optimizing for the VT architectural pattern. The output of the config function is now used to stripmine across the application vector via the for loop on line 7. The call to set_vlen on line 8 allows the stripmine loop to naturally handle cases where size is not evenly divisible by the hardware vector length. This eliminates the first conditional branch in Figure 13(b) to check whether the index is in bounds. Line 5 instantiates a hardware vector containing elements of type int and initializes all elements in the vector with the scalar value x. This shared variable will be kept in the same hardware vector across all iterations of the stripmine loop. The structured memory accesses are turned into unit-strided loads (line 11,13–14); vlen consecutive elements of arrays a and b are moved into the appropriate hardware vector with the load member function. Note that the conditional store is implemented similarly to Figure 13(b); the base pointer for the array c is copied into all elements of the hardware vector vcp on line 10 and then a scalar store (line 19) is executed inside a conditional.

For vector-SIMD, we were able to leverage the built-in GCC vectorizer for mapping very simple regular DLP microbenchmarks, but the GCC vectorizer could not automatically compile the larger application kernels for the vector-SIMD tiles. For these more complicated vector-SIMD kernels, we use a subset of our VT C++ library for stripmining and vector memory operations along with GCC's inline assembly extensions for the actual computation. Figure 13(d) shows how the existing

VT programming environment is used with hand-coded assembly to map an irregular DLP loop to a vector-SIMD machine. The same stripmining process is used on lines 4–8. Lines 13-14 are the same unit-strided loads used in the VT programming model. The actual computation is expressed with an inline assembly extension found on lines 17-25. Note that a conditional store is implemented with a unit-strided load (line 15), a conditional move (line 21), and a unit-strided store (line 27).

The big reduction in programmability is shown in the progression of Figure 13. Figure 13(a) needs explicit "coarse grain parallelization". Explicit "data-parallelization" is required for Figure 13(b), 13(c), and 13(d). Figure 13(c) and 13(d) require factoring out, vector loads and stores, shared data, and common work to run on the control thread. Finally, Figure 13(d) requires handling irregular control flow with vector flags. As you go down the list, the level of programmer effort was substantially higher. For example, our struggle to find a suitable way to program more interesting codes for the vector-SIMD pattern is anecdotal evidence of the broader challenge of programming such accelerators. These qualitative big steps impact the high-level programming models and ability of compilers to generate quality code for each style.

5 Evaluation Results

In this section, we first compare tile configurations based on their cycle time and area before running four microbenchmarks on the baseline MIMD, vector-SIMD, and VT tiles with no microarchitectural optimizations. We then explore the impact of various microarchitectural optimizations, and compare implementation efficiency and area-normalized performance of the MIMD, vector-SIMD, and VT patterns for the six application kernels.

5.1 Cycle Time and Area Comparison

Tile cycle times vary from 1.10–1.49 ns (see Table 1), with critical paths usually passing through the crossbar that connects cores to individual data cache banks. Figure 14 shows the area breakdown of the tiles normalized to a *mimd-c4r32* tile, and Figure 15 depicts three example VLSI layouts. The caches contribute the most to the area of each tile. Note that a multi-core vector-SIMD tile (*vsimd-c4v1r256+bi*) is 20% larger than a multi-core MIMD tile with the same number of long-latency functional units and the same total number of physical registers (*mimd-c4r256*) due to the sophisticated VMU and the extra integer ALUs per bank. A multi-lane vector-SIMD tile (*vsimd-c1v4r256+bi*) is actually 16% smaller than the *mimd-c4r256* tile because the increased area overheads are amortized across four lanes. Note that we added additional buffer space to the multi-lane tiles to balance the performance across vector tiles, resulting in similar area usage of the memory unit for both multi-core and multi-lane vector tiles. Across all vector tiles, the overhead of the embedded control processor is less than 5%, since it shares long-latency functional units with the vector unit.

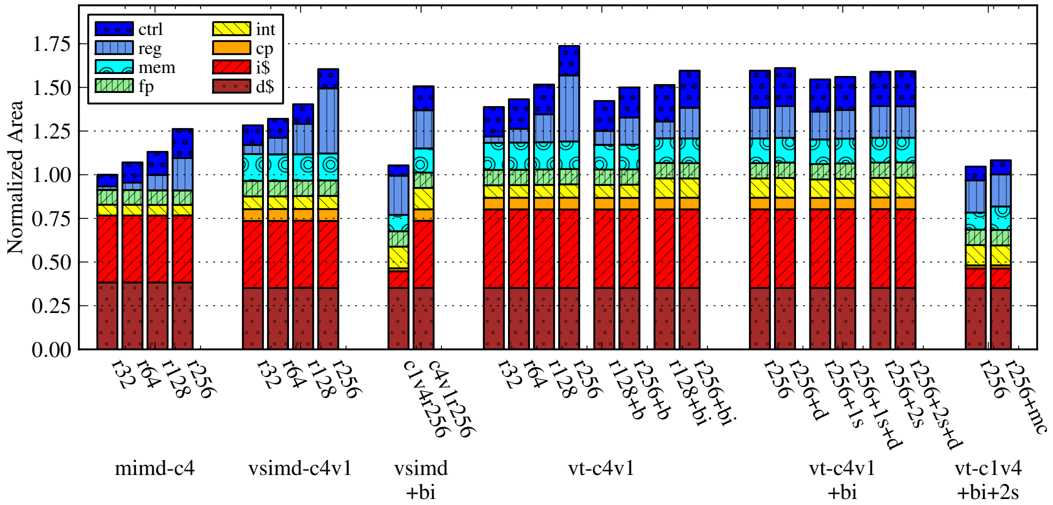
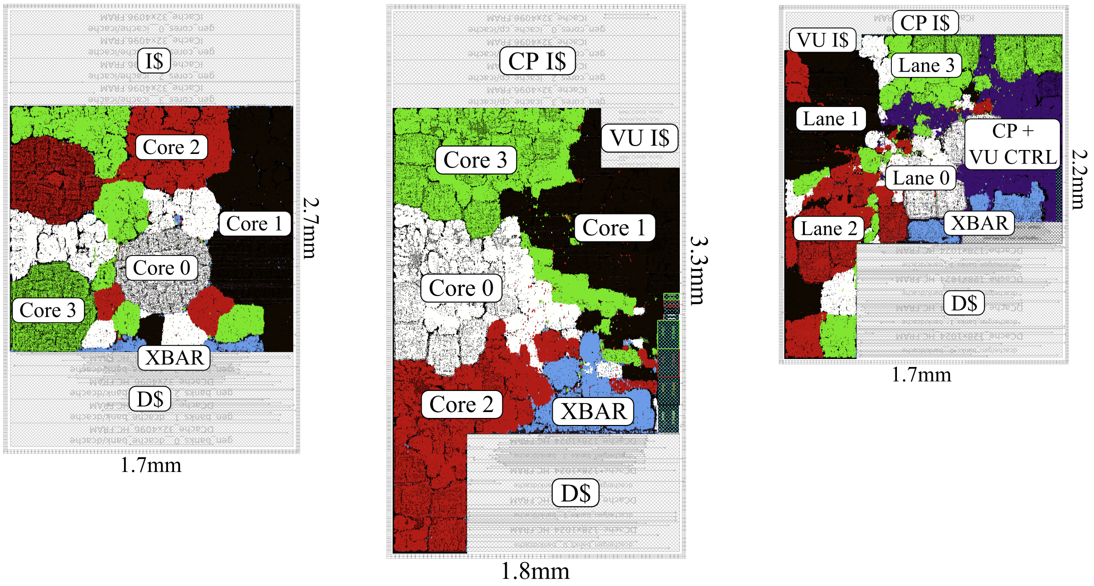


Figure 14: Area for Tile Configurations – Area breakdown for each of the 25 tile configurations normalized to the *mimd-c4r32* tile. (adapted from [16])



(a) *mimd-c4r256* (b) *vt-c4v1r256+bi+2s+d* (c) *vt-c1v4r256+bi+2s*

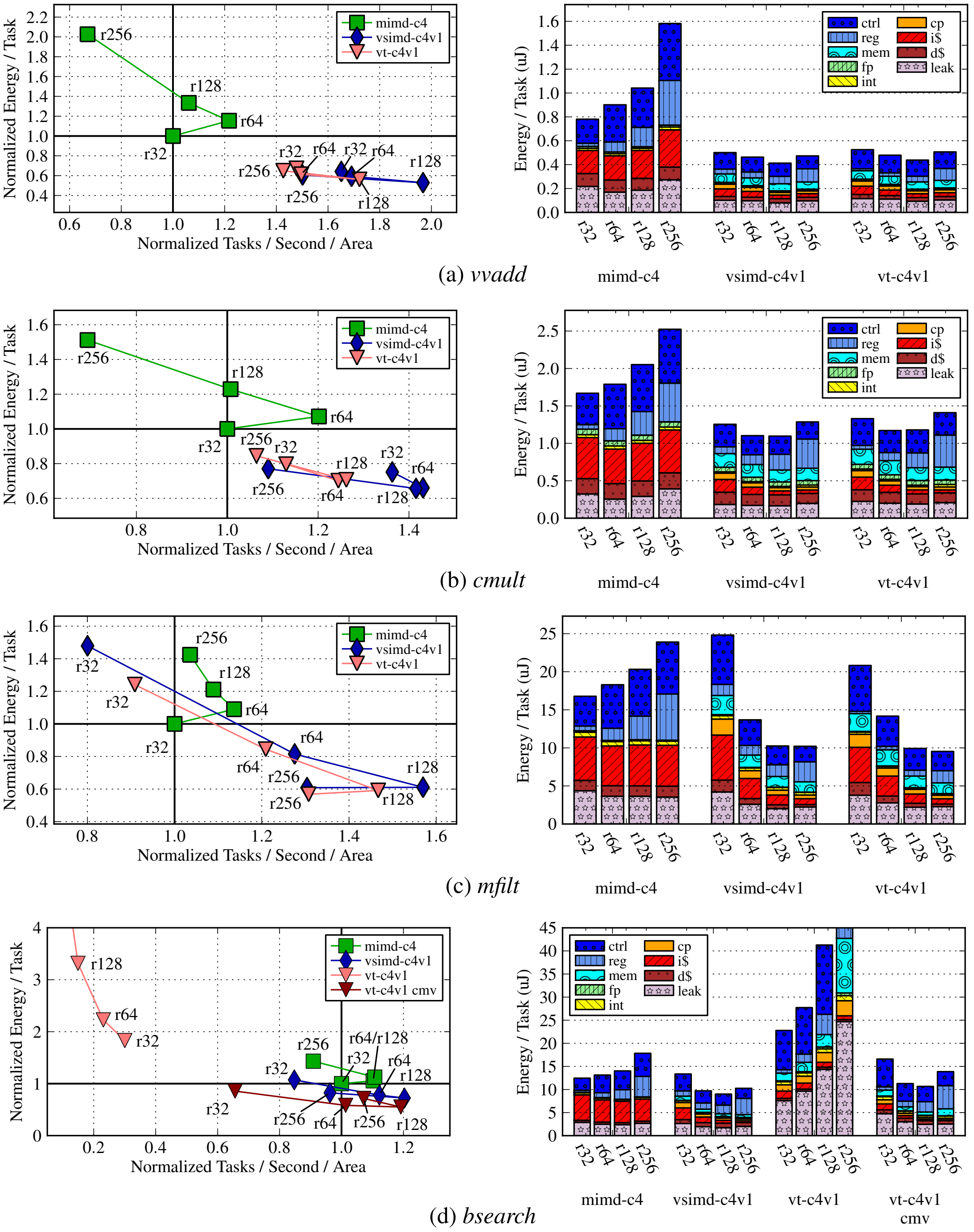
Figure 15: Example VLSI Layouts – ASIC layout designs for *mimd-c4*, *vt-c4v1r256+bi+2s+d*, and *vt-c1v4r256+bi+2s* with individual cores/lanes and the memory crossbar highlighted.

Comparing a multi-core VT tile (*vt-c4v1r256+bi*) to a multi-core vector-SIMD tile (*vsimd-c4v1r256+bi*) shows the area overhead of the extra VT mechanisms is only ≈6%. The VT tile includes a PVFB instead of a vector flag register file, causing the register file area to decrease and the control area to increase. There is also a small area overhead due to the extra VT instruction cache. For multi-lane tiles, these VT overheads are amortized across four lanes making them negligible (compare *vt-c1v4r256+bi+2s* vs. *vsimd-c1v4r256+bi*).

5.2 Microbenchmark Results without Microarchitectural Optimizations

Figure 16 compares the microbenchmark results between the baseline MIMD, vector-SIMD, and VT tiles. Note that these tiles do not implement any microarchitectural optimizations described in Section 3.3, 3.4, and 3.5. The microbenchmarks are sorted by irregularity. As you go down the rows, the microbenchmark gets more irregular (see active µT distribution in Table 3).

Figure 16(a), 16(b), and 16(c) show the impact of increasing the number of physical registers per core or lane. For *mimd-c4r**, increasing the number of µTs from 1 to 2 improves area-normalized performance but at an energy cost. The energy increase is due to a larger register file (now 64 registers per core) and more control overhead. Supporting more than two µTs reduces performance due to the non-trivial start-up overhead required to spawn and join the additional µTs and a longer cycle time. In the *vsimd-c4v1* tile and the *vt-c4v1* tile with a unified vector register file, adding more vector register elements increases hardware vector length and improves temporal amortization of



(a) *vvadd*

(b) *cmult*

(c) *mfilt*

(d) *bsearch*

Figure 16: Implementation Efficiency and Area-Normalized Performance for Baseline MIMD, vector-SIMD, and VT Tiles Running Microbenchmarks – Results for the *mimd-c4**, *vsimd-c4v1**, and *vt-c4v1** tiles running four microbenchmarks. Energy vs. performance / area results are shown on the left. Energy and performance/area are normalized to the *mimd-c4r32* configuration. Energy breakdowns are shown on the right. In (d), *vt-c4v1r256* (outside figure) uses approximately 6× as much energy (78μJ per task) and has 11× poorer performance normalized by area than *mimd-c4r32*.

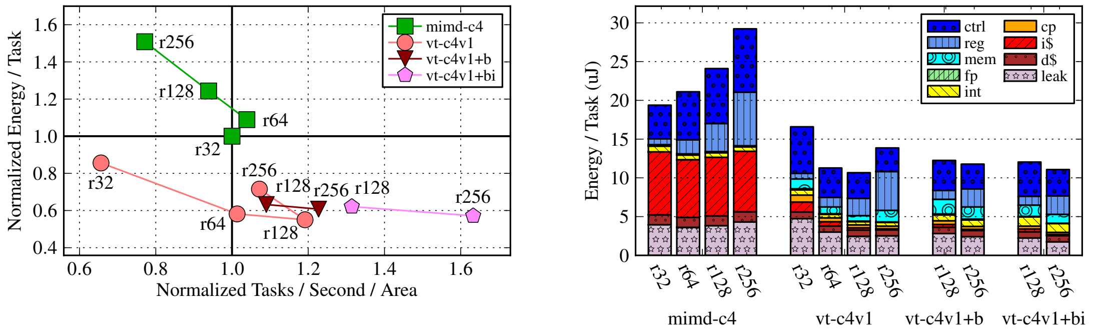
the CP, instruction cache, and control energy. At 256 registers, however, the larger access energy of the unified register file outweighs the benefits of increased vector length. The performance also decreases since the access time of the register file becomes critical. To mitigate these overheads, we consider banking the vector register file and adding per-bank integer ALUs (Section 3.3), and the results are presented in the next section (Figure 17).

As shown in Figure 16(d), adding more registers to the VT tile when running *bsearch* results in worse area-normalized performance and worse energy consumption. This is due to the high irregularity of the microbenchmark. According to the active µT distribution statistics in Table 3, only 1-25% of the µTs were active 77.6% of the time. Without any microarchitectural optimizations such as density-time execution (Section 3.3) and dynamic fragment convergence (Section 3.4), increase in the vector length means that more time is spent on inactive µTs, since the execution time is proportional to the vector length rather than the number of active µTs. On the other hand, vector flags used in the hand-coded *bsearch* for vector-SIMD, and conditional move assembly instructions used in the hand-optimized *bsearch-cmv*, which encode data-dependent conditionals, do not splinter a vector into many fragments of only a few active µTs each. As a result, the trend looks more similar to the microbenchmarks above. See how microarchitectural optimizations such as density-time execution and dynamic fragment convergence help to achieve better energy efficiency and area-normalized performance in the next section (Figure 18).

For regular DLP (Figure 16(a) and 16(b)) and mild irregular DLP (Figure 16(c)), vector tiles surpass the MIMD tiles in both energy efficiency and area-normalized performance. For highly irregular DLP (Figure 16(d)), the VT tile without any microarchitectural optimizations performs worse than the MIMD tile. The area overhead, and some other overheads including the vector fetch instruction and the µT stop instruction of the VT tile when compared to the vector-SIMD tile are exposed because the microbenchmarks only execute 1–10 µT instructions (see Table 2) for a short period of time. A fairer comparison among vector tiles with microarchitectural optimizations are presented in Section 5.4.

5.3 Microarchitectural Tradeoffs

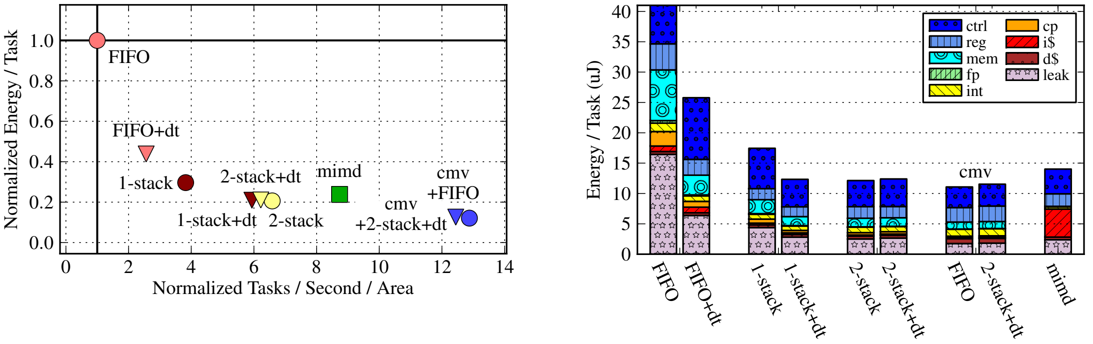
Figure 17 shows the impact of register file banking and adding per-bank integer ALUs when executing *bsearch-cmv*. Banking a register file with 128 entries reduces register file access energy but decreases area-normalized performance due to bank conflicts (see *vt-c4v1+b* configuration). Adding per-bank integer ALUs partially offsets this performance loss (see *vt-c4v1+bi* configuration). With the additional ALUs, a VT tile with a banked register file improves both performance and energy versus a VT tile with a unified register file. Figure 14 shows that banking the vector register file reduces the register file area by a factor of 2×, while adding local integer ALUs in a banked design only modestly increases the integer and control logic area. Based on analyzing results across many



(a) Energy vs. Performance / Area for *bsearch-cmv*

(b) Energy Breakdown for *bsearch-cmv*

Figure 17: Impact of Additional Physical Registers, Intra-Lane Register File Banking, and Additional Per-Bank Integer ALUs – Results for multi-core MIMD and VT tiles running the *bsearch-cmv* microbenchmark.



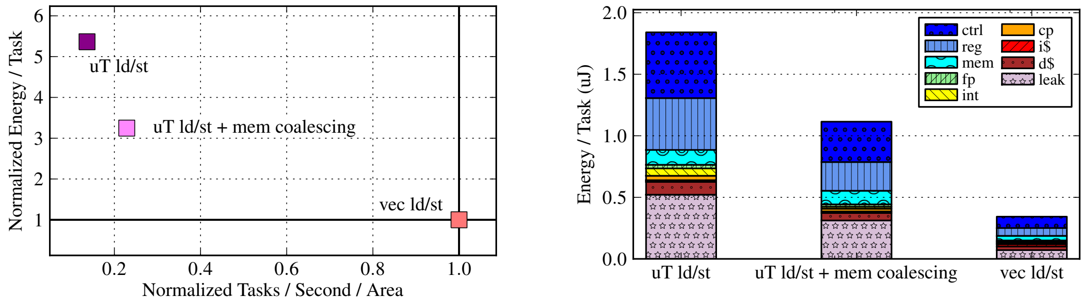
(a) Energy vs. Performance / Area for *bsearch*

(b) Energy Breakdown for *bsearch*

Figure 18: Impact of Density-Time Execution and Stack-Based Convergence Schemes – Results for the *mimd-c4r128* and *vt-c4v1r256+bi* tiles running *bsearch* and *bsearch-cmv*. 2-stack VT tiles have better performance and energy efficiency than the MIMD tile. In (b), FIFO (extends outside figure) uses approximately 59µJ per task.

tile configurations and applications, we determined that banking the vector register file and adding per-bank integer ALUs was the optimal choice for all vector tiles.

Figure 18 shows the impact of adding density-time execution and dynamic fragment convergence to a multi-core VT tile running *bsearch*. Adding just density-time execution eliminates significant wasted work after divergence, improving area-normalized performance by 2.5× and reducing energy by 2×. Density-time execution is less useful on multi-lane configurations due to the additional constraints required for compression. Our stack-based convergence schemes are a different way of mitigating divergence by converging µTs when possible. For *bsearch*, the 2-stack PVFB forces µTs to stay on the same loop iteration, improving performance by 6× and reducing energy by 5× as compared to the baseline FIFO PVFB. Combining density-time and a 2-stack PVFB has little impact here as the 2-stack scheme already removes most divergence (see Table 3). Our experience with other microbenchmarks and application kernels suggest that for codes where convergence is



(a) Energy vs. Performance / Area for *vvadd*

(b) Energy Breakdown for *vvadd*

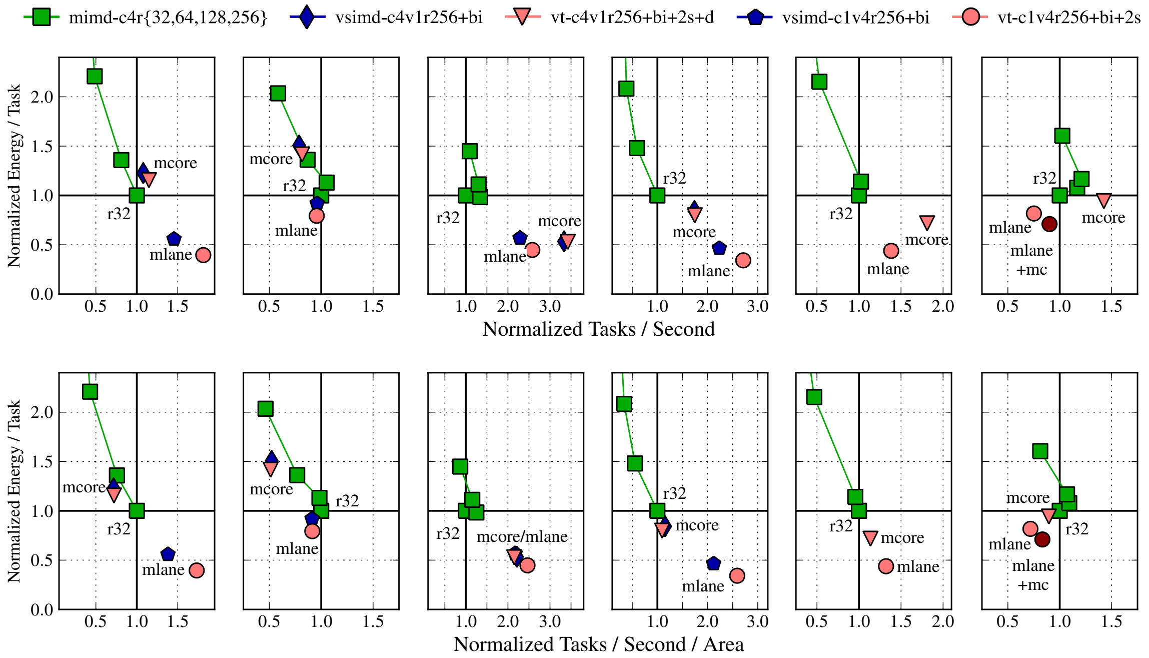
Figure 19: Impact of Memory Coalescing – Results for multi-lane VT tile running *vvadd*.

simply not possible the addition of density-time execution can have significant impact. Note that replacing branches with explicit conditional moves (*bsearch-cmv*) performs better than dynamic optimizations for µT branches, but µT branches are more general and simpler to program for irregular DLP codes. Table 1 and Figure 14 show that the 2-stack PVFB and density-time execution have little impact on area and cycle time. Based on our analysis, the 2-stack PVFB is used for both multi-core and multi-lane VT tiles, while density-time execution is only used on multi-core VT tiles.

Figure 19 illustrates the benefit of vector memory accesses versus µT memory accesses on a multi-lane VT tile running *vvadd*. Using µT memory accesses limits opportunities for access-execute decoupling and requires six additional µT instructions for address generation, resulting in over 5× worse energy and 7× worse performance for *vvadd*. Memory coalescing recoups some of the lost performance and energy efficiency, but is still far behind vector instructions. This small example hints at key differences between SIMT and VT. Current SIMT implementations use a very large number of µTs (and large register files) to hide memory latency instead of a decoupled control thread, and rely on dynamic coalescing instead of true vector memory instructions. However, exploiting these VT features requires software to factor out the common work from the µTs. Also note that memory coalescing can still help µT memory accesses used for non-structured data accesses in VT implementations (see Figure 20(f)).

5.4 Application Kernel Results

Figure 20 compares the application kernel results between the MIMD, vector-SIMD, and VT tiles. All vector tiles include a banked vector register file with per-bank integer ALUs. Both VT tiles (multi-core and multi-lane) use the 2-stack dynamic fragment convergence scheme. On top of these microarchitectural optimizations, the multi-core VT tile implements density-time execution, and one of the multi-lane VT tile includes a dynamic memory coalescer. The upper row plots overall energy/task against performance, while the lower row plots energy/task against area-normalized performance to indicate expected throughput from a given silicon budget for a highly parallel work-



(a) *viterbi* (b) *rsort* (c) *kmeans* (d) *dither* (e) *physics* (f) *strsearch*

Figure 20: Implementation Efficiency and Performance for MIMD, vector-SIMD, and VT Patterns Running Application Kernels – Each column is for different kernel. Legend at top. *mimd-c4r256* is significantly worse and lies outside the axes for some graphs. There are no vector-SIMD implementations for *strsearch* and *physics* due to difficulty of implementing complex irregular DLP in hand-coded assembly. mcore = multi-core vector-SIMD/VT tiles, mlane = multi-lane vector-SIMD/VT tiles, mlane+mc = multi-lane VT tile with a dynamic memory coalescer, r32 = MIMD tile with 32 registers (i.e., one µT). (adapted from [16])

load. Kernels are ordered to have increasing irregularity from left to right. We draw several broad insights from these results.

First, we observed that adding more µTs to a multi-core MIMD tile is not particularly effective, especially when area is considered. We found parallelization and load-balancing become more challenging for the complex application kernels, and adding µTs can hurt performance in some cases due to increased cycle time and non-trivial interactions with the memory system.

Second, we observed that the best vector-based machines are generally faster and/or more energy-efficient than the MIMD cores though normalizing for area reduces the relative advantage, and for some irregular codes the MIMD cores perform slightly better (e.g., *strsearch*) though at a greater energy cost.

Third, comparing vector-SIMD and VT on the first four kernels, we see VT is more efficient than vector-SIMD for both multi-core single-lane (*c4v1*) and single-core multi-lane (*c1v4*) design points. Note we used hand-optimized vector-SIMD code but compiled VT code for these four kernels. One reason VT performs better than vector-SIMD, particularly on multi-lane *viterbi* and *kmeans*, is that vector-fetch instructions more compactly encode work than vector instructions, reducing pressure on the VIU queue and allowing the CT to run ahead faster.

Fourth, comparing *c4v1* versus *c1v4* vector machines, we see that the multi-lane vector designs are generally more energy-efficient than multi-core vector designs as they amortize control overhead over more datapaths. Another advantage we observed for multi-lane machines was that we did not have to partition and load-balance work across multiple cores. Multi-core vector machines sometimes have a raw performance advantage over multi-lane vector machines. Our multi-lane tiles have less address bandwidth to the shared data cache, making code with many vector loads and stores perform worse (*kmeans* and *physics*). Lack of density-time execution and no ability to run independent control threads also reduces efficiency of multi-lane machines on irregular DLP code. However, these performance advantages for multi-core vector machines usually disappear once area is considered, except for the most irregular kernel *strsearch*. The area difference is mostly due to the disparity in aggregate instruction cache capacity.

Overall, our results suggest a single-core multi-lane VT tile with the 2-stack PVFB and a banked register file with per-bank integer ALUs (*vt-c1v4r256+bi+2s*) is a good design point for Maven.

6 Conclusions

Effective data-parallel accelerators must handle regular and irregular DLP efficiently and still retain programmability. Our detailed VLSI results confirm that vector-based microarchitectures are more area and energy efficient than scalar-based microarchitectures, even for fairly irregular data-level parallelism. We introduced Maven, a new simpler vector-thread microarchitecture based on the traditional vector-SIMD microarchitecture, and showed that it is superior to traditional vector-SIMD architectures by providing both greater efficiency and easier programmability. Maven's efficiency is improved with several new microarchitectural optimizations, including efficient dynamic convergence for microthreads and ALUs distributed close to the banks within a banked vector register file.

In future work, we are interested in a more detailed comparison of VT to the popular SIMT design pattern. Our initial results suggest that SIMT will be less efficient though easier to program than VT. We are also interested in exploring whether programming environment improvements can reduce the need of VT or SIMT mechanisms to further improve efficiency while maintaining programmability, and whether hybrid machines containing both pure MIMD and pure SIMD might be more efficient than attempting to execute very irregular code on SIMD hardware.

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, Dec 1994.

[2] C. Batten. Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators. PhD Thesis, MIT, 2010.

[3] C. Batten, R. Krashinsky, S. Gerding, and K. Asanović. Cache Refill/Access Decoupling for Vector Machines. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2004.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics (TOG)*, 23(3):777–786, Aug 2004.

[5] D. DeVries and C. G. Lee. A Vectorizing SUIF Compiler. *SUIF Compiler Workshop*, Jan 1995.

[6] R. Espasa and M. Valero. Decoupled Vector Architectures. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 1996.

[7] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(2):1–35, Jun 2009.

[8] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, Mar 2006.

[9] M. Hampton and K. Asanović. Compiling for Vector-Thread Architectures. *Int'l Symp. on Code Generation and Optimization (CGO)*, Apr 2008.

[10] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2009.

[11] J. H. Kelm, D. R. Johnson, S. S. Lumetta, M. I. Frank, and S. J. Patel. A Task-Centric Memory Model for Scalable Accelerator Architectures. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2009.

[12] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25:21–29, Mar/Apr 2005.

[13] R. Krashinsky. Vector-Thread Architecture and Implementation. PhD Thesis, MIT, 2007.

[14] R. Krashinsky, C. Batten, and K. Asanović. Implementing the Scale Vector-Thread Processor. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 13(3), Jul 2008.

[15] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[16] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computer Architecture. *IEEE Micro*, 28(2):39–55, Mar/Apr 2008.

[18] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel. Tradeoffs in Designing Accelerator Architectures for Visual Computing. *Int'l Symp. on Microarchitecture (MICRO)*, Nov 2008.

[19] Graphics Guide for Windows 7: A Guide for Hardware and System Manufacturers. Microsoft White Paper, 2009.

[20] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches, 2009.

[21] U. G. Nawathe, M. Hassan, L. Warriner, K. Yen, B. Upputuri, D. Greenhill, and A. Kumar. An 8-Core 64-Thread 64 b Power-Efficient SPARC SoC. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2007.

[22] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, Mar/Apr 2008.

[23] NVIDIA's Next Gen CUDA Compute Architecture: Fermi. NVIDIA White Paper, 2009.

[24] The OpenCL Specification. Khronos OpenCL Working Group, 2008.

[25] OpenMP Application Program Interface. OpenMP Architecture Review Board, 2008.

[26] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[27] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. Vector Lane Threading. *Int'l Conf. on Parallel Processing (ICPP)*, Aug 2006.

[28] R. M. Russel. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, Jan 1978.

[29] K. Sankaralingam, S. W. Keckler, W. R. Mark, and D. Burger. Universal Mechanisms for Data-Parallel Architectures. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2003.

[30] J. E. Smith, G. Faanes, and R. Sugumar. Vector Instruction Set Support for Conditional Operations. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2000.

[31] J. Wawrzynek, K. Asanović, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan. Spert-II: A Vector Microprocessor System. *IEEE Computer*, 29(3):79–86, Mar 1996.

[32] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. *Communications of the ACM (CACM)*, 52(4):65–76, Apr 2009.

[33] S. Woop, J. Schmittler, and P. Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (TOG)*, 24:434–444, Jul 2005.

[34] M. Zagha and G. E. Blelloch. Radix Sort for Vector Multiprocessors. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, 1991.