# Hardware Support for Irregular Control Flow in Vector Processor

Huy Vo
huytbvo@berkeley.edu
May 7, 2012

## Abstract

*As data parallel processors grow in popularity, so too does their desired range of applications. If this growth is too continue, data parallel processors must be able to support regular as well as irregular data parallelism. In light of this fact, this paper presents an implementation of a vector processor that can handle data parallel applications that exhibit irregular control flow. In particular, I start with a single threaded pending vector fragment buffer(PVFB) design which I then extend to a multithreaded PVFB design. Through an extensive design space exploration of various microarchitectural alternatives, I evaluate the efficiency and performance of my system. My results indicate that a multithreaded PVFB design with 8 threads and 64 μThreads per thread maximizes performance while minimizing the area overhead.*

## 1. Introduction

Data level parallelism(DLP) can be divided into two categories: regular DLP and irregular DLP. Regular DLP applications, such as the code in Figure 1(a), have structured control flow and data accesses. In contrast, irregular DLP, such as the programs in Figure 1(c) and Figure 1(b), can have any combination of unstructured control flow and data accesses. Because many existing DLP applications already exhibit irregular DLP, it is likely the case that future DLP applications will have significantly larger amounts of irregular DLP. In order for vector processors to continue to experience commercial and mainstream success they must be able to efficiently handle both regular and irregular DLP.

In this paper, I examine alternative methods of handling DLP applications with irregular control flow in a vector processor. I first present an overview of the vector architecture that I am working with in Section 3. In Section 4 I show how to augment the existing vector architecture to provide support for DLP applications with irregular control flow and discuss the design space associated with this implementation. Then in Section 5 I briefly present the motivation for moving from a single threaded design to a multithreaded design before showing how to extend the single threaded design to a multithreaded design. I also discuss the tradeoffs of this design point. Finally, I describe my hardware and evaluation toolchain in Section 6 and present my results in Section 7.

## 2. Related Work

In this section I introduce two vector microarchitectures that are similar to the vector microarchitecture that I am working with in my project and briefly discuss how these microarchitectures manage irregular control flow.

The Maven [2,3] vector thread data parallel accelerator targets a vector ISA for structured memory accesses and targets a scalar ISA for everything else. For branches, the compiler generates the same instructions as it would for a scalar processor. This microarchitecture uses a pending vector fragment buffer (PVFB) to manage divergence and reconvergence. In order to reduce the complexity of this structure, the Maven accelerator limited the hardware vlen to be 32.

NVIDIA GPUs [1, 5] also target a scalar ISA. Unlike Maven, the compiler provides static hints for reconvergence. This microarchitecture has a small on chip stack that is backed up by DRAM to manage divergence. Furthermore, NVIDIA GPUs group threads together into chunks of 32 called warps. They have a hardware scheduler that selects one or more warp to issue instructions from every cycle. They mostly chose to limit warps to size of 32 in order to reduce the complexity of divergent state.

## 3. Baseline Vector Architecture

In this section I introduce the baseline vector architecture, a vector thread (VT) architecture, that I modified to support irregular control flow. I compare it to a scalar architecture and a traditional vector architecture to highlight the differences. In the following discussion, I will use the term *microthread* (μT) to refer to one iteration of the body of a vectorizable loop, *appvlen* refers to the application vector length, and *vlen* refers to the hardware vector length.

### 3.1. Scalar Architecture

Figure 2(a) shows the assembly code that the example code in Figure 1(a) will compile to when mapped to a

```
for (i = 0; i < 100; i++) {
  A[i] = B[i] + C[i]
}
```

(a) Regular DLP Code

```
for (i = 0; i < 100; i++) {
  if ( B[i] > C[i] ) {
    A[i] = B[i] - C[i]
  } else {
    A[i] = B[i] + C[i]
}
```

(b) DLP Code with Irregular Control Flow

```
for (i = 0; i < 100; i++) {
  A[D[i]] = B[i] * C[i]
}
```

(c) DLP Code with Irregular Memory Access

**Figure 1: DLP Code Examples** Three examples of DLP code written in C-syntax. Example (a) has regular control flow and data accesses. Example (b) has irregular control flow while example (c) has irregular memory accesses.

scalar architecture. The scalar core fetches and executes every single instruction for every µT over the entire *appvlen*.

### 3.2. Traditional Vector Architecture

Figure 2(b) shows the assembly code generated when the example in Figure 1(a) is compiled for a traditional vector (TVEC) architecture. The key differences in the assembly code are the *v*'s prepended to the instructions and registers to indicate vector instructions and vector registers. The scalar core fetches each of those instructions *appvlen/vlen* times, recognizes that they are vector instructions and sends them to vector issue unit. The vector issue unit then executes each of those instructions once per µT over the entire *vlen*.

### 3.3. Vector Thread Architecture

Figure 2(c) shows the assembly code generated when compiling for a vector thread (VT) architecture. Unlike the TVEC architecture (which targets a vector ISA), a VT architecture targets a scalar ISA. Furthermore, the scalar CPU does not execute any of the instructions shown in Figure 2(c). Instead, the scalar CPU sends the PC (*0x0* in this example) to the vector issue unit (VIU). The VIU then starts fetching instructions from that PC and executes those instructions once per µT over the entire *vlen*. In continues fetching and executing instructions until it hits the *stop* instruction at PC *0x10*. Stripmining happens on the scalar CPU. That is, the scalar CPU will send PC *0x0* to the VIU *appvlen/vlen* times.

A block diagram of the baseline VT microarchitecture is presented in Figure 3(a). Notice that the VIU is connected to the instruction cache which is not the case in a TVEC architecture. The scalar CPU interfaces to the VIU through a queue. This allows the stripmining loop to run ahead instead of waiting on VIU to finish fetching and executing instructions.

## 4. Single Pending Vector Fragment Buffer Design Space

### 4.1. Overview

Now that I have introduced the baseline vector architecture that I am working with, I will proceed to describe the modifications I made to the existing pipeline to support irregular control flow in DLP applications. In the following discussion, I will be using the code in Figure 4(b) to show how modifications interacts with the existing pipeline. It should be noted that my changes are purely microarchitectural; the RISC-V compiler compiles DLP applications with irregular control flow the same way it would compile any other DLP application.

In order to handle irregular control flow, the VT processor maintains a bit vector which I will call a *mask*. Each bit in the mask corresponds to a µT. It if the bit is 1 then the corresponding µT is active, meaning the VT processor will execute that iteration of the loop. Otherwise, if the is is 0, the corresponding µT is inactive and the VT processor will not execute that iteration of the loop.

As an example, let's assume that we have an appvlen of 4, *vlen* also of 4, and we want to execute the VT as-

```
0: ld  x1, 0(B)
4: ld  x2, 0(C)
8: add x3, x2, x1
C: sd  x3, 0(A)
```

(a) Scalar Assembly

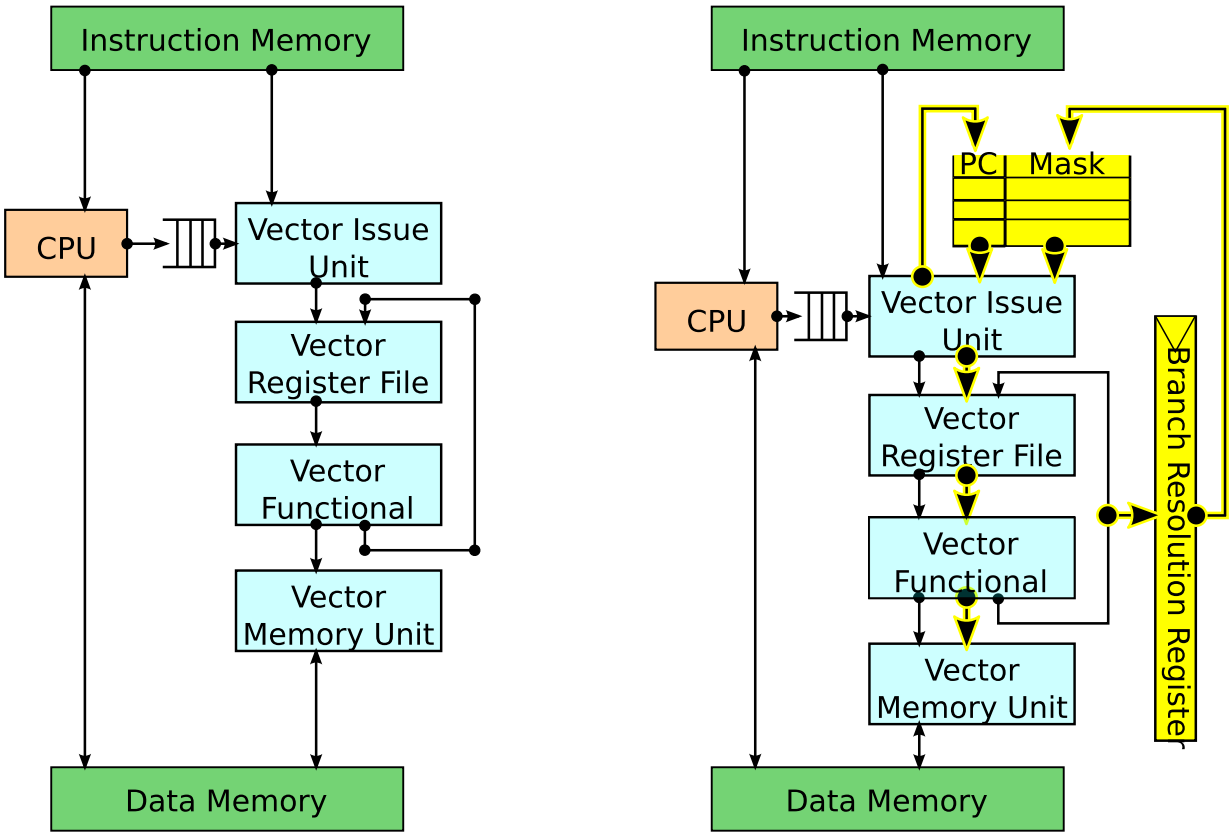
```
0: vld  vx1, B
4: vld  vx2, C
8: vadd vx3, vx2, vx1
C: vsd  vx3, A
```

(b) Traditional Vector Assembly

```
 0: ld  x1, B + utidx
 4: ld  x2, C + utidx
 8: add x3, x2, x1
 C: sd  x3, A + utidx
10: stop
```

(c) Vector Thread Assembly

**Figure 2: Vectorizable C Code Mapped to Different Architectures** The code in Figure 1(a) compiles to the scalar assembly code shown in 2(a), the traditional vector assembly in 2(b), and finally the vector thread assembly in 2(c). The assembly code only shows the main body of the loop; the stripmining overhead is ignored for brevity. *A*, *B*, and *C* hold the memory address of the application vectors.




(a) **Block Diagram of Baseline Vector Microarchitecture**

(b) **Block Diagram of Augmented Vector Microarchitecture**

**Figure 3: Comparison of Baseline Vector Microarchitecture to Single Threaded PVFB Microarchitecture** Figure (b) shows the modifications I made to the existing pipeline in order to support irregular control flow. It contains a *pending vector fragment buffer* which is a queue of PCs and masks generated from branch instructions. The *Branch Resolution Register* holds the intermediate result of an in flight branch. Finally, the extra highlighted arrows between the *Vector Issue Unit*, *Vector Register File*, *Vector Functional Unit*, and the *Vector Memory Unit* corresponds to the mask that is passed down the pipeline to enable/dis-enable the various hardware modules.

sembly code shown in Figure 4(b). The scalar CPU will send to the VIU the PC of the first instruction, which in this example is 0x0. The VIU will then start executing at address 0x0 with a mask initialized to all 1s (because initially all µT are active). The VIU executes each instruction once per active µT by sending the relevant operation and mask to the vector register file, the vector functional unit, and the vector memory unit. Execution continues normally until the branch instruction at PC 0x8. On a branch instruction, the VIU will issue a comparison operation along with the mask to the vector functional unit. The vector functional unit computes the mask for the taken path. So in this example, for each µT, the vector functional unit will produce a 1 if x1 is less than or equal to x2. If the new mask is all 1s or all 0s, then the VIU continues execution along either the taken or non-taken path respectively. Otherwise, the µTs have diverged. So let's say in this example the µTs have diverged, that is µT0 and µT1 took the branch while µT2 and µT3 did not. So the VIU will save PC else and mask 0011 (I refer to this PC and mask pair as a *vector fragment*) before continuing execution at PC 0xC with mask 1100 until it hits the stop instruction. On the stop, the VIU sees that it still has a vector fragment so it restarts execution at PC *else* with mask 0011.

### 4.2. Microarchitectural Modifications

Figure 3(b) highlights the modifications that I made to the existing pipeline. The PC and mask queue (I call this structure a *pending vector fragment buffer*(PVFB)) holds vector fragments generated from branches. The vector functional unit writes the result of a comparison operation to the branch resolution register which is then used to update the pending vector fragment buffer as appropriate. Finally, I modified the interface and datapath of the vector register file, the vector functional unit, and the

vector memory unit to use the mask sent by the VIU.

### 4.3. Design Space

In the worst case scenario, it is possible to diverge in such a way that every μT could be at a different PC. So if we have a vlen of 4, then the PVFB must have 4 entries and the mask has to be 4 bits wide. The size of the PVFB is then $4 * (32 \text{ bits} + 4 \text{ bits}) = 144$ bits. In general, the size of the PVFB is

$$\text{vlen} \cdot (32 + \text{ vlen}) = 32\text{vlen} + \text{vlen}^2.$$

In other words, the size of the PVFB scales quadratically in vlen. This fact puts a restriction on the vlen because for large vlen, this structure becomes unbuildable. However, a larger vlen is desirable because it reduces the stripmining loop (which frees up the scalar CPU for other tasks) and better amortizes the cost of instruction fetch and control.

## 5. Multi-Threaded PVFB

### 5.1. Overview

The main disadvantage of the implementation presented in Section 4.3 is that it uses a monolithic PVFB which grows quadratically in the *vlen*. In light of this fact, this section presents an implementation that uses multiple smaller PVFB. As before, my changes are purely microarchitectural; the design of the PVFB is invisible to the user.

Let's say that I have a *vlen* of 16. I could use a single PVFB with a mask queue that is 16 bits wide and 16 entries deep, as demonstrated in the Section 4. However, this design does not scale well. Instead, I can use 4 smaller PVFB where each PVFB has its own PC and mask queue. Furthermore, each mask queue is now 4 bits wide and 4 entries deep.

To see how this work, I will refer back to the example program in Figure 4(b). When the VIU receives PC 0x0 from the scalar CPU, it initializes 4 vector fragments. Each vector fragment starts at PC 0x0 with mask 1111. In other words, μT0 - μT3 are grouped in the $0^{th}$ vector fragment and stores vector fragments generated from branches in the $0^{th}$ PVFB. The other μT's are grouped in a similar fashion. Every cycle, the VIU selects from 1 of 4 vector fragments to issue instructions. It executes instructions from each of these vector fragments until all of their respective μTs have executed the stop instruction.

I call this implementation a Multithreaded PVFB Design because of the way the μTs are grouped and scheduled. The groups of μTs, which I refer to as a hardware threads, have no input and output dependencies on each other. This means that the VIU can interleave instructions from multiple threads and keep this invisible to the programmer. It should be noted that this is different from multithreading on a scalar processor where a thread can be a different instruction stream. Here, threads have the same instruction stream (i.e. the same vectorizable loop), but represent exclusive sets of iterations of that loop.

### 5.2. Microarchitectural Modifications

Figure 5 shows the modifications I made. That design shows 4 hardware threads indicated by the 4 PVFB. They are connected to the VIU through a round robin arbiter. Every cycle, the round robin arbiter gives a separate PC and mask pair to the VIU. The VIU fetches and executes the instruction at that PC over all active μTs in that vector fragment.

### 5.3. Design Space

The example that I used in presenting this design point has 4 PVFB where each PVFB has its PC and mask queue. It is easy to see that this PVFB design is much smaller than using a monolithic PVFB to realize the same vlen. In general, let's say I only want to have a mask queue that is 32 bits wide and 32 bits deep. I would then need vlen/32 PVFB. The size of this PVFB is then

$$\text{vlen}/32 \cdot (32 \text{ bits} + 32 \text{ bits}) \cdot 32 \text{ entries} = 64\text{vlen bits}.$$

```
for (i = 0; i < 100; i++) {
  if ( B[i] > C[i] ) {
    A[i] = B[i] - C[i]
  } else {
    A[i] = B[i] + C[i]
}
```

(a) DLP Code with Irregular Control Flow Written in C

```
 0: ld  x1, B + utidx
 4: ld  x2, C + utidx
 8: ble x1, x2, else
 C: sub x3, x1, x2
10: j exit
else:
14: add x3, x1, x2
exit:
1C: sd  x3, A + utidx
20: stop
```

(b) Vector Thread Assembly Code

**Figure 4: DLP Code With Irregular Control Flow Mapped to VT Architecture** Figure (b) shows what the code in Figure 4(a) looks like when compiled for a VT architecture. Again only the body of the loop is shown for brevity. Note that it looks very similar to assembly code that compiled for a scalar architecture.

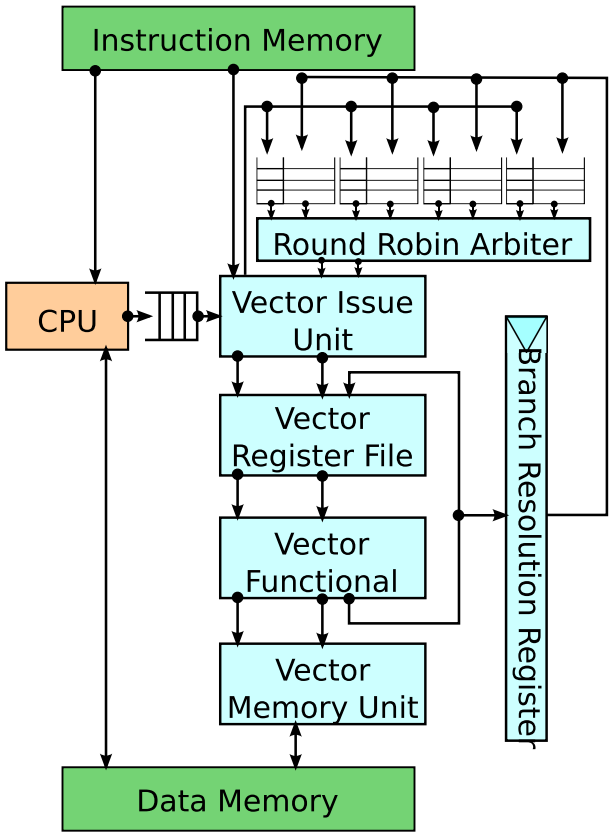



**Figure 5: Block Diagram of Multithreaded PVFB Microarchitecture** A block diagram of the multithreaded PVFB design point. Here 4 threads are shown. Every cycle, the VIU chooses one thread to issue instructions from.

As you can see, the size of the PVFB is now linear in the vlen. However, this equation assumes that I have an arbitrary number of threads which is also impractical because the path through the arbiter as well as the logic to manage the separate threads does not scale well. This design does have another upside in that because I am introducing multithreading, I can better hide the latencies from memory accesses and branch computations.

## 6. Evaluation Framework

The section describes the infrastructure used to evaluate the microarchitecture presented in the previous sections. I describe the various microarchitectural configurations I look at as well as how I generated those configurations. I also describe the application kernels I used to measure performance.

### 6.1. Hardware Toolflow

The RTL is written in the Chisel HDL developed at UC Berkeley. I have parameters that control the number of threads, the width and depth of each PVFB, and even the type of scheduler. These parameters are passed to the Chisel compiler which then produces the corresponding Verilog.

I targeted TSMC's 45-nm GSBWP processing using a Synopsys ASIC toolflow which involves using VCS for RTL simulation and Design Compiler for synthesis. RTL simulation produces cycle counts. Synthesis provides timing, area, and power numbers. I get energy numbers by multiplying cycle count by power by critical path length.

The SRAMs used to build the PVFB are generated from memory compiler that uses area, timing and power estimates provided by CACTI [4].

### 6.2. Microarchitecture Configurations

For this paper, I evaluated 42 different configurations. Each configuration has a different combination of number of threads, maximum supportable vlen, and scheduling. More specifically, I evaluated configurations that used 1, 2, 4, 8, and 16 threads. For each separate thread count, I varied the vlen from 32 to 2048 in powers of 2. It should be noted that I did not evaluate a 2048 vlen with 1 and 2 threads because I was unable to generate an SRAM large enough for the PVFB's mask queue. Furthermore, I was unable to generate SRAMs smaller than 32 bits wide with 32 entries. This means that at 2 threads, I did not have a vlen smaller than 64, at 4 threads, I did not have a vlen

smaller than 128, at 8 threads I did not have a vlen smaller than 256, and at 16 threads I did not have a vlen smaller than 512. Finally, I also looked at fine grained multithreading and coarse grained multithreading for designs with more than 1 thread.

### 6.3. Benchmarks

I used 4 different benchmarks to evalute the performance of each configuration.

The *binary search* benchmark uses a binary search algorithm to lookup 1000 keys from a sorted array of 1000 key-value pairs. This benchmark has the largest amount of irregular control flow among the four benchmarks that I used. It has an outer for-loop that iterates over the input keys and an inner while loop that performs binary search on each of the keys.

The remaining benchmarks are fairly regular. The *vvadd* adds two 1000-element vector of floating point numbers. The *complex multiply* benchmarks multiplies two arrays of complex numbers. Finally, the *masked filter* benchmark applies a mask to an input set of pixels.

## 7. Results

### 7.1. Cycle Time

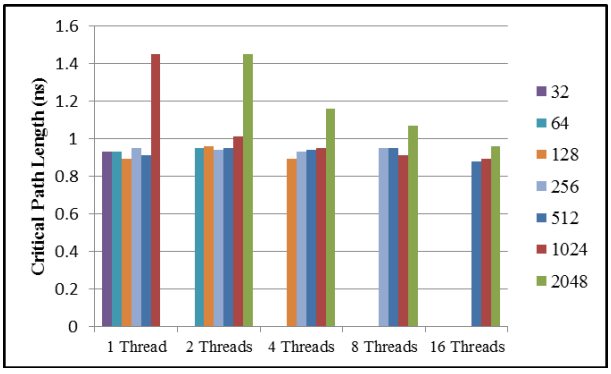



**Figure 6: Cycle Time Across Different Configurations** The length of the critical path in ns is on the y-axis. Along the x-axis I have vary the number of threads. For each thread, I vary the maximum vector length from 32 to 2048.

After generating the Verilog and SRAMs as detailed in Section 6, I pushed it through synthesis with a 1 ns cycle time constraints. Figure 6 shows the post-synthesis critical path length which is the effective cycle time. The results shows that most configurations are able to meet the 1 ns constraint. The 1 thread and the 2 thread configurations are unable to meet the 1 ns constraint with a maximum vlen of 1024 and 2048 respectively. This is due to the fact that there is a spike in the address setup time for an SRAM of 1024 bits wide and 1024 entries deep (which is used in these configurations). The 4 thread and 8 thread design point are unable to meet a 1ns constraint for a maximum vlen of 2048 because the path from the data out pin through the arbiter to the input the icache appears on the critical path. After moving to 16 threads, which uses a shallower PC queue, that critical path goes away.

### 7.2. Area

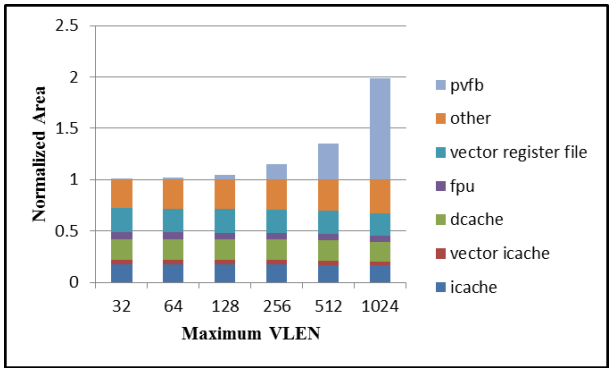


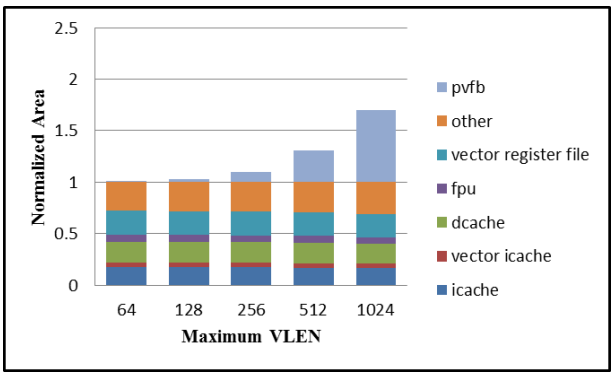

**Figure 7: Area Breakdown Across Different VLEN for 1 Thread** Area of the PVFB normalized to the rest of the system. The maximum vlen is varied from 32 to 1024.

Figure 7 compares post synthesis area usage. Notice that with a vlen of 32 to 128 the size of the PVFB is negligible. However, after that, the size of the PVFB quickly grows to become as large as the rest of the system, clearly demonstrating the quadratic dependency on the vlen. This result shows that a single threaded PVFB is simply undesirable. Ideally, I want to have a large vlen but have the size of the PVFB to be much smaller than the vector register file.

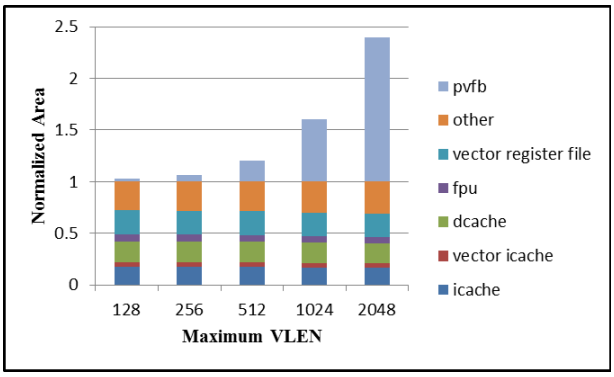
The post synthesis area results for different number of threads is summed up in Figure 8. The results shows that, for a fixed number of threads, the size of the PVFB still scales quadratically with the vlen. In fact, the PVFB still takes up approximately half of the system with a vlen of 2048. The good news is that the percentage of area devoted to the PVFB does decrease as you increase the number of threads but fix the vlen. For example, building a PVFB large enough to handle a vlen of 1024 is much more feasible at 16 threads than 2 threads.

### 7.3. Power

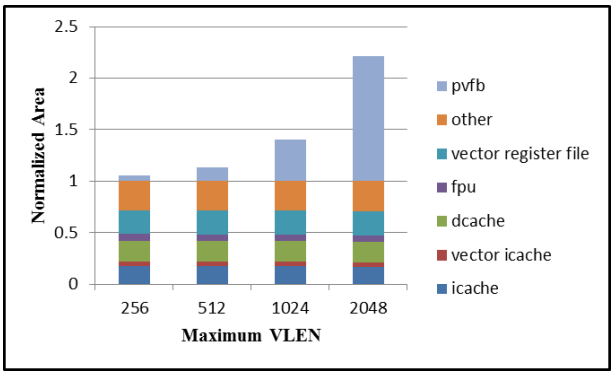
Figure 9 shows the post synthesis power numbers for different components for 1 thread with varying vlen. This result shows that the power consumption of the PVFB also grows quadratically in the vlen like the size of the PVFB. However, unlike the size, the power consumption does not jump as dramatically as the maximum vlen increases to 2048.



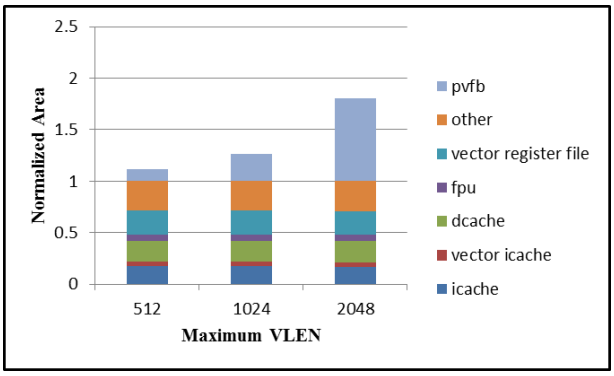

(a) 2 Threads




(b) 4 Threads




(c) 8 Threads




(d) 16 Threads

**Figure 8: Area Breakdown Across Different VLEN for All Multithreaded Design Points** Area of the PVFB normalized to the rest of the system for (a) 2 Threads, (b) 4 Threads, (c) 8 Threads, and (d) 16 Threads. The vlen is then varied for each thread size. Note that some design points are not missing because I was either not able to build a large enough or small enough SRAM.

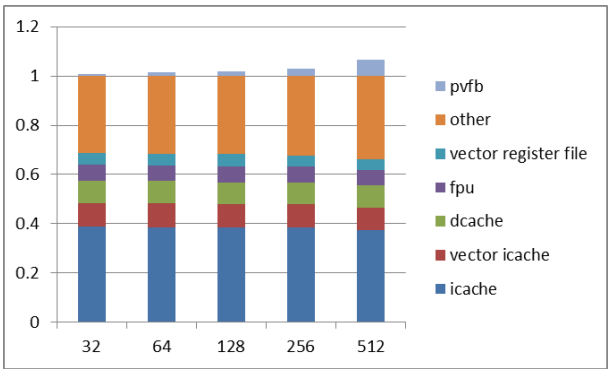



**Figure 9: Power Breakdown Across Different VLEN for 1 Thread** The power consumption of the PVFB is normalized to the rest of the system. The maximum vlen is varied from 32 to 1024.

Figure 10 shows the post synthesis power consumption numbers for the multithreaded PVFB design points. The results show that the multithreaded PVFB design points also follows a similar quadratic trend as the single threaded design. Notice that there is very little change in power consumption across different number of threads for the same vlen.

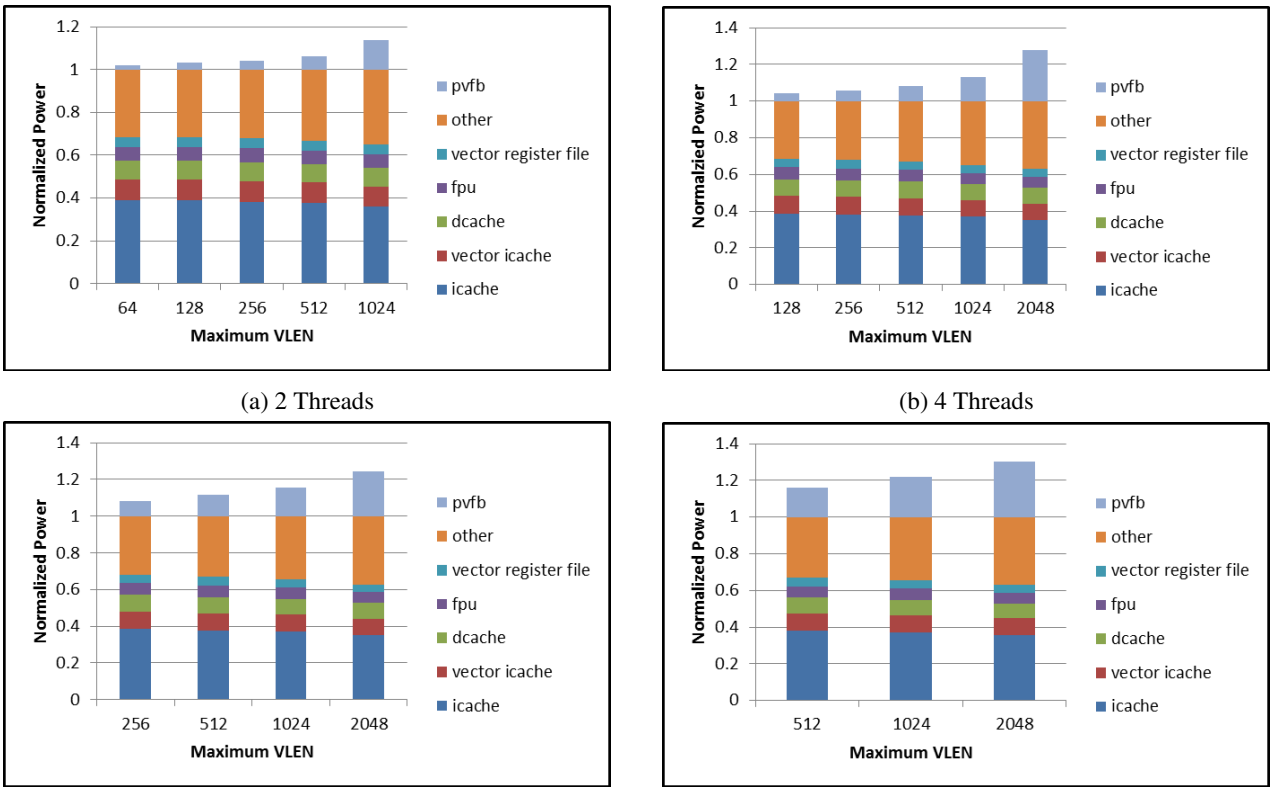
## 7.4. Energy

The energy estimate for all the benchmarks are summarized in Figure 11. Energy is estimated using the following equation:

$$\text{power} \cdot \text{cycles} \cdot \text{cycle time}.$$

The power numbers are post-synthesis power estimated which are calculated using a statistical model of bit transitions. The number of cycles is the cycle count produced from RTL simulation while the cycle time is the critical path length of respective design point.

Binary search has the most dramatic changes in energy usage across different configurations. Figure 11(a) shows that energy usage increases dramatically as the vlen is increased for a fixed vlen. This is due to the fact that, for this input data set to binary search, the μTs tend to diverge in small clustered clumps. So when a branch instruction is executed, it is possible (with a long mask) to end up with vector fragment in which only a few μTs are active. The vector processor then ends up spending many cycles not doing anything useful when executing




(a) 2 Threads

(b) 4 Threads

(c) 8 Threads

(d) 16 Threads

**Figure 10: Power Breakdown Across Different VLEN for All Multithreaded Design Points** The power consumption of the PVFB is normalized to the rest of the system for (a) 2 Threads, (b) 4 Threads, (c) 8 Threads, and (d) 16 Threads. The vlen is then varied for each thread size.

that vector fragment. Therefore, for this benchmark, it is best to use a large number of threads if a large vlen is desired. In fact, a close examination of drops in energy usage across different threads for a fixed vlen indicates that this benchmark responds well to a mask length of 128. For example, energy usage drops for a vlen for 256 and 512 as the number of threads increases from 1 to 2 and 2 to 4 respectively.

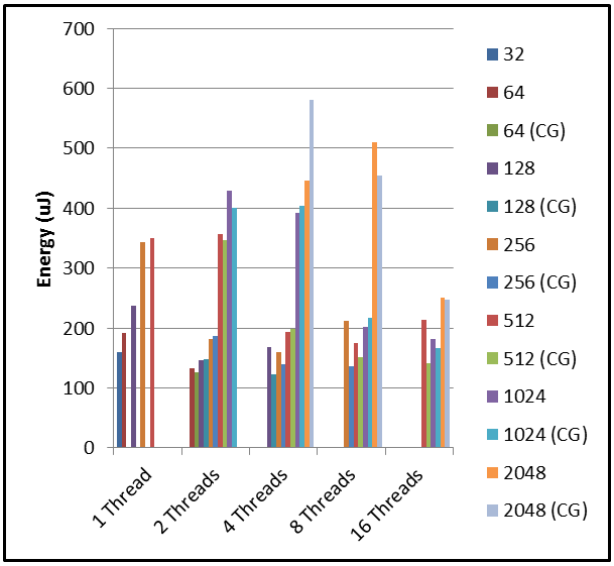
The vvadd and complex multiply benchmarks do not have irregular control flow and so do not have the same behavior as the binary search benchmark. Figure 11(b) and Figure 11(c) shows that energy usage increases with that maximum vlen no matter what size thread is used. This is due to the fact that these benchmarks only have a small decrease in in cycle count as the maximum vlen is increased but must pay the cost of a larger increase in power consumption due to larger PVFBs.

Finally, the masked filter benchmark also exhibits irregular control flow but does not diverge as much as the binary search benchmark. Figure 11(d) confirms this fact, that is there are no large consistent drops in energy usage as the number of threads or maximum vlen is increased.

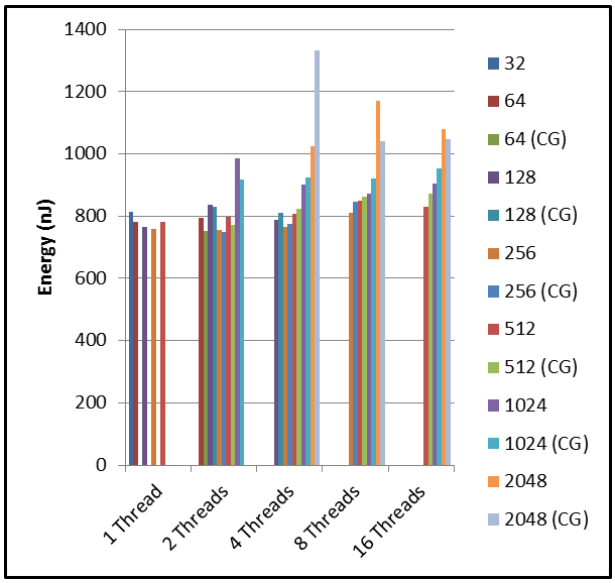
Note that there is no consistent behavior in choosing coarse grained over fine grained multithreading. However, coarse grained multithreading has the worst performance across all benchmarks for a configuration of 4 threads with a 2048 vlen.

## 8. Conclusion

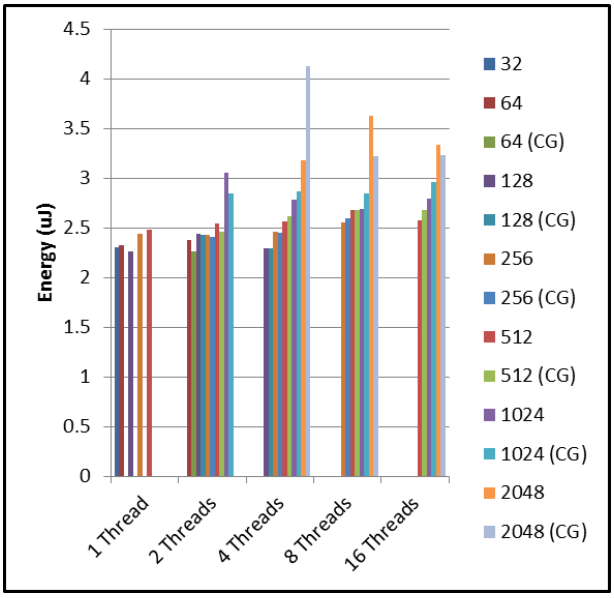
The results from Section 7.2 suggest that a multithreaded design is best because the single threaded implementation does not scale well. Although the baseline VT architecture can support a 2048 vlen, the multithreaded area results suggest that I can support at most a 512 vlen. The area breakdown in Figure 8 shows that supporting a 2048 vlen is infeasible even at 16 threads. A vlen of 1024 might appear to be buildable with 16 threads, but this is not the case. In that configuration, the PVFB is as large as the vector register file which is not a good use of resources. The area results suggest that using 8 threads to support a maximum vlen of 512 has optimal area usage. The cycle time and energy usage results confirms that this hardware configuration will perform well.



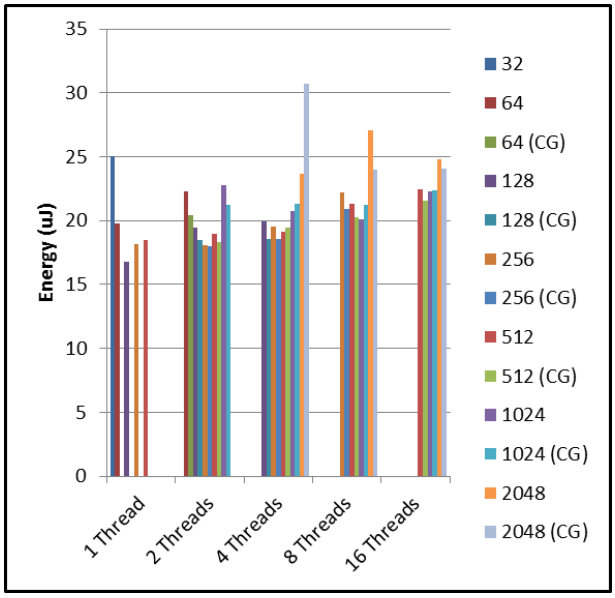

(a) Binary Search




(b) vvadd




(c) Complex Multiply




(d) Masked Filter

**Figure 11: Energy Usage for Different Benchmarks** Energy usage is shown for different number of threads for (a) binary search, (b) vvadd, (c) complex multiply, and (d). At each thread count the vector length and scheduling is varied. Fine grained multithreading is used by default and as applicable. CG is short for coarse grained.

## 9. Future Work

The reason that binary search has the behavior discussed in Section 7.4 is due to the fact that there is no reconvergence. If the hardware did provide support for reconvergence then it would be possible to merge vector fragments with very few active μTs together, thus preventing the vector processor from spending many cycles doing nothing. However, hardware reconvergence mechanisms tend to be power hungry.

Figure 11 shows that different thread scheduling algorithms will lead to different energy usages. So there is merit to exploring more sophisticated scheduling algorithms as well as examining why there is no consistent behavior between choosing coarse grained multithreading over fine grained multitreading.

I briefly mentioned in Section 4 that a larger vlen is desirable because it minimizes the stripmining loop, freeing the scalar CPU for other tasks. However, the benchmarks that I used in my analysis were unable to take advantage of the larger vlen. Furthermore, in these benchmarks, the scalar CPU did not have anything to do other than send PCs to the VIU. So in order to better evaluate my system, I would use more complicated benchmarks and applications that can benefit from larger vlen.

Finally, my results in Section 7.2 showed that supporting a 2048 vlen is still infeasible with a multithreaded PVFB design. One possible idea is to use a hybrid design in which I have a width but shallow PVFB working alongside the standard PVFB design. The idea is that applications that can use a larger vlen but have little divergence can use the wide and shallow PVFB. Applications with a lot of divergence, on the other hand, would have to use a smaller vlen in order to use the standard multithreaded PVFBs.

## References


[1] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

[2] Y. Lee. Efficient VLSI Implementations of Vector-Thread Architectures. Master's thesis, UC Berkeley, 2011.

[3] 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.

[4] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches, 2009.

[5] Nvidia's next gen cuda compute architecture: Fermi. NVIDIA White Paper, 2009.