Wavefront Skipping using BRAMs for Conditional Algorithms on Vector Processors

Aaron Severance
University of British Columbia
Vancouver, BC Canada
aaronsev@ece.ubc.ca

Joe Edwards
University of British Columbia
Vancouver, BC Canada
jedwards@ece.ubc.ca

Guy G.F. Lemieux
University of British Columbia
Vancouver, BC Canada
lemieux@ece.ubc.ca

VectorBlox Computing, Inc.
Vancouver, BC Canada
aseverance@vectorblox.com

VectorBlox Computing, Inc.
Vancouver, BC Canada
jedwards@vectorblox.com

VectorBlox Computing, Inc.
Vancouver, BC Canada
glemieux@vectorblox.com

ABSTRACT

Soft vector processors can accelerate data parallel algorithms on FPGAs while retaining software programmability. To handle divergent control flow, vector processors typically use mask registers and predicated instructions. These work by executing all branches and finally selecting the correct one. Our work improves FPGA based vector processors by adding wavefront skipping, where wavefronts that are completely masked off are skipped. This accelerates conditional algorithms, particularly useful where elements terminate early if simple tests fail but require extensive processing in the worst case. The difference in logic speed and RAM area for FPGA based circuits versus ASICs led us to a different implementation than used in fixed vector processors, storing wavefront offsets in on-chip BRAM rather than computing wavefronts skipped dynamically. Additionally, we allow for partitioning the wavefronts so that partial wavefronts can skip independently of one another. We show that <5% extra area can give up to 3.2× better performance on conditional algorithms. Partial wavefront skipping may not be generally useful enough to be added to a fixed vector processor; it provides up to 65% more performance for up to 27% more area. In an FGPA, however, the designer can use it to make application specific tradeoffs between area and performance.

1. INTRODUCTION

Soft vector processors (SVPs) are a particular type of overlay that creates a vector processor inside of an FPGA. Having a software-programmable model reduces design cycle time and makes programming and debugging easier than writing at the RTL level. An SVP has one or more lanes (ALUs with their own local memory) that can process data in parallel; the data processed in a single cycle is called a wavefront. If the vector length is longer than the number of lanes the instruction will sequentially process one wavefront per cycle until the whole vector instruction has completed. For algorithms that use conditional execution, i.e. branching, the vector processor must execute both paths of the branch and mask off writes for elements not on the current branch. This can result in a large portion of execution unit results that are not used, especially for algorithms that can stop processing some data elements early depending on a conditional check.

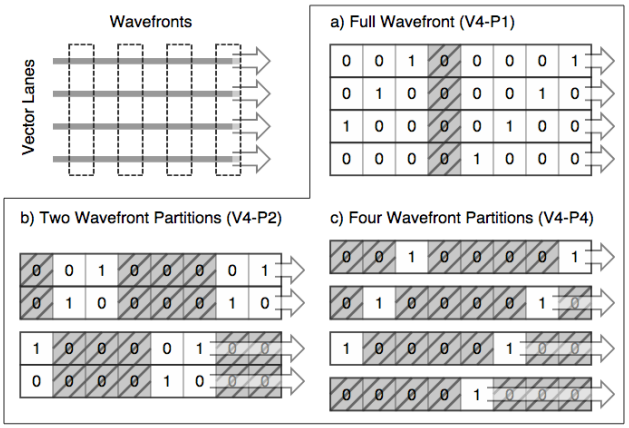

Figure 1: Wavefront Skipping on a 4 Lane MXP

In order to speed up these conditional algorithms, masked-off elements must not utilize execution slots. It is possible to compress vectors in order to remove masked-off elements, but this is costly for algorithms that use packed data such as stencil filters. Rather, we would like to skip over elements without rearranging data, which led us to implement wavefront skipping. A wavefront is one cycle of data within the parrallel vector pipelines. In wavefront skipping, wavefronts where all the elements are masked off are skipped. We also implemented partial wavefront skipping, by which the wavefront is divided into partitions that can skip independently. This finer grained skipping can lead to performance increases, at the cost of requiring more resources.

Figure 1 helps illustrate how wavefront skipping works. The values shown are the mask bits corresponding to each

FPGA'15, February 22–24, 2015, Monterey, CA, USA.

ACM 978-1-4503-3315-3/15/02 ...$15.00.
http://dx.doi.org/10.1145/2684746.2689072.

data element; a zero indicates that element is masked off. In Figure 1a one of the eight wavefronts can be skipped since all elements are masked off. Normally this instruction would take eight cycles to process, but with wavefront skipping it can complete in seven. Figure 1b shows two wavefront partitions; the finer granularity means more partial wavefronts can be skipped, and so the instruction can run in only four cycles (the partition with only three cycles ends up being idle during the last cycle). And Figure 1c shows four wavefront partitions, at which point the partition size is a single element, and the instruction can execute in two cycles.

Our work gives the first implementation of wavefront skipping on SVPs. It uses a different approach than used on fixed vector processors, where the mask register is read out in parallel and the number of leading masked-off elements is computed each cycle. Instead, we took advantage of the relative cheapness of BRAMs on FPGAs by computing wavefront offsets beforehand in a setup instruction and storing them in BRAMs. Additionally, we implement partitioned wavefront skipping, where instead of entire wavefronts skipping together, partial wavefronts (down to individual bytes) can skip by different amounts. To our knowledge this is the first implementation of this idea in any vector processor. The full wavefront skipping implementation requires no more than 5% increase in area and a single BRAM and achieves speedups of up to 3.2× for early exit algorithms. The addition of partial wavefront skipping provides additional gains but requires up to 27% additonal area, so it might not make sense to implement in a hard vector processor. However, it may make sense to add in an SVP if the design has some unused BRAMs within the device or the algorithm is particularly sensitive to wavefront partitioning.

2. BACKGROUND

Vector processors have been used in supercomputers since the 1970's [3]. By performing the same operation on multiple elements, vector processors are able to achieve high throughput on data parallel problems using only a small amount of code. In particular, the inner loops of many scientific and signal processing algorithms can be converted to vector instructions. Vector processing is a form of SIMD (single instruction, multiple data) processing which is highly efficient for simple, regular algorithms such as dense linear algebra and stencil filters. In much of the supercomputer world, vector processors have given way to GPUs, which share many concepts and structures with vector processors but are more general purpose (at the cost of more area and power). Vector processing has a few niches still; for example the Convey HC-2ex [1] uses FPGAs as a vector coprocessor for memory-intensive applications, and IBM's ViVA (Virtual Vector Architecture) [5] allows programmers to treat multiple POWER processors as a single vector processor in supercomputer applications.

2.1 FPGA-based Soft Vector Processors

Given that vector processing is relatively inexpensive in terms of logic gates and can give high throughput for data parallel applications, it makes sense in hindsight for it to be used in FPGAs (where gates are much more expensive than ASICs). The history of the idea can be traced back to VIRAM [6], which was an ASIC vector processor that showed vector processing to be more efficient in both power and performance than superscalar or VLIW processors for

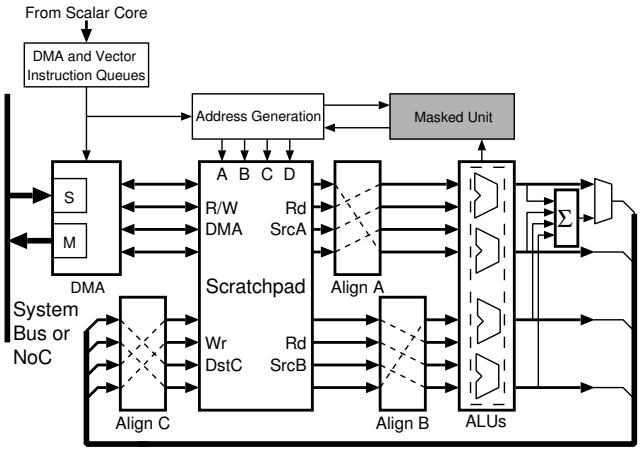

Figure 2: VectorBlox MXP with four 32-bit lanes

embedded media applications. Two parallel SVP implementations showed the feasibility of implementing a vector processor on an FPGA, VESPA from University of Toronto [13] and VIPERS from University of British Columbia [14]. Further work has been done to customize vector processors for FPGAs, with the most recent being the VectorBlox MXP [9] which has the ability to dispatch multi-operand custom instructions to the external FPGA fabric. Additional work on applications and programmability has been done at University of Cambridge [8] where a neural network simulation was implemented and a C++ library developed to encapsulate vectors as objects.

2.2 VectorBlox MXP

Our SVP, the VectorBlox MXP (MXP), is shown in Figure 2. It consists at a high level of a scalar core (Altera's Nios II/f) coupled to a vector core and DMA engine. The scalar core handles all control flow and I/O, and dispatches instructions to the vector core and DMA engine. All data processed by MXP is held in a flatly-addressable, multiple-bank scratchpad memory. Rather than have individual vector registers, a vector can start at any address in the scratchpad and be of any length. Data is initially read into the scratchpad using the DMA engine. Once it has been read into the scratchpad, a vector instruction can execute. If there are data hazards between the vector instruction and DMA transaction, hardware interlocks stall the vector core until the hazard is resolved.

When a new vector instruction is ready to issue, the address generation logic will generate the addresses for the vector operand wavefronts, then increment them appropriately each cycle until the end of the vector is reached. After the source operands are read from the scratchpad, they proceed through an alignment network. Alignment is required because vectors can start at any address within the scratchpad, hence be located in any bank of the scratchpad. The operands are then fed into the parallel ALUs to execute. An optional reduction-accumulate stage reduces the output to a single sum; a commonly needed operation in vector programs. Finally, the data is realigned to the destination bank before being written back to the scratchpad. The pipeline takes 8 stages from read to writeback, with extra stages

added for alignment and reduction in wider vector processors.

We also support subword SIMD operations where two 16-bit operations or four 8-bit operations can execute in one lane. Additionally, we have a conditional move operation (discussed in Section 3) that will only overwrite the destination vector if a condition check is true. The conditional move uses byte enables to enable or disable writing to specific scratchpad banks.

2.3 Conditional Execution

Support for conditional execution in vector processors is well researched; a good summary is [11]. For short conditionals, implementations that use predicated operations or conditional moves can give good performance. These operations are performed on all data elements, with only the elements that pass some conditional test written back; the rest of the results are discarded. For longer conditional branches, though, this can lead to a large percentage of execution time spent on unused results. In these instances, it is desirable to skip elements that are not on the current branch. Three strategies can be employed (separately or together): compress/expand, scatter/gather, and wavefront skipping (also referred to as density-time masking).

Compress vector operations are a way to take a source vector and a mask and produce a new vector that only contains the valid (unmasked) elements without gaps. The expand vector operation is the opposite of a compress; it takes a compressed source vector and a mask and fills in the unmasked slots in the destination vector with the source data. VIRAM implemented VCOMPRESS and VEXPAND operators. During a long branch, the source operands can be compressed, followed by processing only the shorter (compressed) vector operations, and finally the result expanded. The main drawback to this approach is that all source operands must be compressed, and all destination operands must be expanded. In a MxN conditional stencil filter on an image, for instance, the MxN input pixels would all have to be separately compressed. Viola-Jones face detection operates in this manner, which is why compress/expand was not considered suitable for our implementation. This will be explained further in Section 3.4.

Scatter/gather is a method of performing indexed memory accesses (indexed store is a scatter, indexed load is a gather), either to the local memory store or external memory. Along with a compress operation or special index calculating instructions, scatter/gather can be used to speed up conditional execution. For the conditional stencil filter example, the indices of pixels to be processed could be compressed, and then the pixel data needed for each location could be loaded using gather operations. The main drawback of this approach is that parallelizing scatter and gather operations requires parallel memory acceses, which are nontrivial. VESPA could perform parallel scatter gather accesses within a single cache line. A special throughput cache [10] was developed for MXP to support scatter/gather operations, but even in the best case where all data could fit in a statically allocated multiple-bank on-chip memory, speedups were modest.

Wavefront skipping, by contrast, uses knowledge of the mask register to skip elements that do not need to be processed. An earlier implementation scans the mask register during instruction execution to determine if subsequent wavefronts can be skipped. This introduces enough latency that [7] gives a way to reduce the overhead by only skipping powers of 2 elements (at the cost of doing some extra work because the skipping is not exact). Given that an FPGA implementation is already slower than a hard processor and we are double-clocking our scratchpad to provide additional ports, we wished to avoid the additional latency in our design. Our implementation uses a special mask setup instruction to store the offsets of valid wavefronts in one or more BRAMs within the FPGA. Prior work [11] suggests the idea that each lane can skip forward individually rather than lockstep as entire wavefronts. We take this one step further by allowing the wavefront to be partitioned and separately implement wavefront skipping in each partition. The number of partitions can range from 1 (the entire wavefront) to four times the number of lanes (making each byte-lane in a separate partition). To our knowledge, we are the first to propose and implement partitioned wavefront skipping in a vector processor.

A similar concept exists in GPUs. GPUs divide up a kernel (consisting of hundreds of threads) into warps (analogous to our wavefronts, usually 16 or 32 threads). Inside a kernel, branch instructions allow different threads to diverge. If no threads inside a warp are on one of the branches, that warp is not scheduled to execute that branch; this is somewhat analagous to unpartitioned wavefront skipping in vector processors. Our method of wavefront skipping keeps more of the management of testing and skipping under software control; we consider this a good tradeoff for SVPs where logic gates and power are more precious than in custom chips. Additionally [4] gives simulation results for SIMT warp compaction. Their approach works within blocks of warps and moves elements between warps to reduce the amount of unnecessary work done. This achieves a similar result as our partitioned wavefront skipping where elements from different wavefronts are executed at the same time. To our knowledge no GPU has implemented this technique yet.

3. BRAM BASED WAVEFRONT SKIPPING

Prior to this work MXP supported predicated execution through conditional move, or CMOV, instructions only. This differs from wavefront skipping in that the CMOV operation does both a condition check and move in the same instruction, and it operates on the entire vector instead of just the valid wavefronts. The CMOV instruction checks conditions using a flag bit associated with each element; each 9-bit wide scratchpad BRAM stores 8-bits of data and one flag bit. The flag bit is set by earlier vector instructions; for instance an add instruction stores overflow while a shift right stores the bit shifted out. The flag bit is used along with whether the resulting byte is zero or non-zero to perform several different CMOV operations. The most common CMOV operations first perform a subtraction-based comparison; the result is predicated based on whether the result is less than zero (LTZ), less than or equal to zero (LTE), etc. The CMOV hardware is part of the ALUs shown in Figure 2.

3.1 Full Wavefront Skipping

Figure 3 gives an example of how to use wavefront skipping to perform strided operations, such as operating on every fifth element as in Figure 1. The first loop sets every

```
#define STRIDE 5

//Toy functition to double every fifth element
//in the vector v_a
void double_every_fifth_element( int *v_a,
                                 int *v_temp
                                 int vector_length )
{
  int i;

  //Initialize every fifth element of v_temp to zero
  for( i = 0; i < vector_length; i++ ) {
    v_temp[i] = i % STRIDE;
  }

  //Set the vector length which will be processed
  vbx_set_vl( vector_length );

  //Set mask for every element equal to zero
  vbx_setup_mask( CMOV_Z, v_temp );

  //Perform the wavefront skipping operation:
  //multiply all non-masked off elements of v_a by 2
  //and store the result back in v_a
  vbx_masked( SVW, MUL, v_a, 2, v_a );
}
```

Figure 3: Code Example: Double Every Fifth Element of a Vector

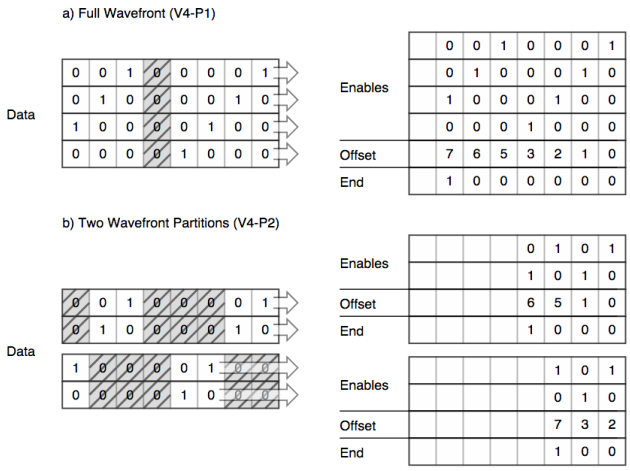

Figure 4: Data Written to Mask BRAM (Every Fifth Element Valid)

fifth element to be 0 (this could be done with vector divide or modulo instructions if the SVP supports them, but is shown in scalar code for clarity). The setup_mask instruction takes as input a comparison operation (CMOV_Z, or use the CMOV hardware to test for equal to zero) and a pointer to the vector operand (v_temp, which is a pointer to a vector of 32-bit data).

After the mask is set, the svp_masked instruction is executed. The 'SVW' type specifier indicates that it operates on a scalar (2) and vector (v_a) inputs and that the values are words (32-bit). The elements in the source and destination vector (both v_a) that are masked-off will be skipped, while the valid elements will be multiplied by 2. Although this is a trivial example and is not faster than using a conditional move, complex algorithms that reuse the mask several times and/or have sparse valid bytes can give significant speedup.

In order to support wavefront skipping, we need to alter the address generation logic of MXP. For normal vectors, the addresses of the operands are incremented by one wavefront every cycle. So, with a V2 MXP (meaning it has two 32-bit vector lanes), each address is incremented by 8 bytes (the width of one wavefront) each cycle. To support wavefront skipping, we want to add a variable number of wavefronts to the operand addresses depending on the value of a mask that has been set. We accomplish this by storing the offsets of wavefronts that have valid elements in a BRAM inside the 'masked unit'. Because not all elements in the wavefront may be valid, we also have to store a valid bit for each byte; during subsequent execution this becomes a byte enable upon writeback. Finally, the length of a wavefront skipped vector will (hopefully) be shorter than the length of the full vector, so we have to mark the last wavefront. We use an extra bit to mark the wavefront that is the end of the skipped vector. This is an implementation detail that was convenient in our design and could be replaced by storing the number of wavefronts in a register.

Block RAMs are limited in depth, however, and MXPs scratchpad can hold vectors of any length. To allow for wavefront skipping of vectors of the whole scratchpad length, the masked unit would need a BRAM as deep as the scratchpad itself. This would be wasteful for many applications, so we allow the user to configure a maximum masked vector length (MMVL). The minimum depth of BRAMs in the Stratix IV FPGAs we tested on is 256 words, so lower MMVL than 256 will result in underutilized BRAMs. The fact that wavefront skipping instructions have their own length restriction must be known by the programmer. In real applications data does not fit entirely in scratchpad memory and so is operated on in chunks (also known as strip-mining) even without an MMVL; the MMVL only affects the size of the chunks.

To generate the wavefront offsets, we added the mask setup instruction. This mask setup instruction uses the conditional move logic already in MXP to generate the valid bits set in the mask BRAM. Figure 4a shows the data written the mask BRAM during the mask setup instruction. For each wavefront processed, the offset and valid bytes are sent to the masked unit. If no valid bytes are set, as in wavefront 4 (data values 16-19), the masked unit does not write into the mask BRAM. If any of the valid bytes are set, the masked unit writes the current offset and the valid bytes to its current BRAM write address and increments the write address. The final wavefront (wavefront 7) causes the end bit to be written to the mask BRAM along with its offset and byte enables, provided there are any byte enables set. In this example the final wavefront has valid byte enables, but in instances where the final wavefront has no byte enables set we avoid writing an empty wavefront by redoing the last write to the mask BRAM with the end bit set.

In the case of algorithms with multiple early exit tests, it is often desirable to do a wavefront skipping 'setup_mask' instruction. This is a trivial extension of the normal mask setup instruction; instead of the wavefront numbers progressing linearly, they are taken from the output of the masked unit. In this way the number of valid elements can decrease until either the algorithm finishes or no more valid elements are left. When no valid elements are left, any wavefront skipping instructions will execute as NOPs. Alternatively, the scalar core can query a mask status register and exit the algorithm if it reports no valid elements are left.

```
for stage in classifier:
    for row in image:
        vector::init-mask

        for feat in stage:
            for rect in feat:
                vector::feat.sum += vector::image[rect]

            if vector::feat.sum > feat.threshold
                vector::stage.sum += feat.pass
            else:
                vector::stage.sum += feat.fail

        if vector::stage.sum < stage.threshold:
            vector::update-mask (exit early if possible)
```

Figure 5: Pseduocode for Viola-Jones Face Detection

3.2 Wavefront Partitioning

So far we have only discussed skipping whole wavefronts. When dealing with wide SVPs, it may be of little value to skip whole wavefronts, since it will be rare (i.e., improbable) that the *entire* wavefront is skippable. As a trivial example, the code from Figure 3 will skip 4/5 of wavefronts on a V1, 1/5 of wavefronts on a V4, and no wavefronts on wider MXPs (every wavefront will contain at least one valid word). To get a speedup on this code on a wide MXP, we need to support skipping at a narrower granularity than the wavefront. We support this by partitioning the wavefront into narrower units which each get their own BRAM in the masked unit; for instance, a V4's masked unit can have 1 BRAM (whole wavefront skipping), 2 BRAMs (pairs of lanes can skip together), or 4 BRAMs (each lane can skip individually). Each BRAM stores a separate offset and end bit, but the byte-enables are split across them.

Figure 4b shows the data written to the 2 mask partition BRAMs for the doubling every fifth element code from Figure 3. This mask will take 4 cycles to execute (the maximum of the depth written to all of the partitions), compared to the 7 cycles needed for the single partition shown in Figure 4a.

One complication with wavefront partitioning in our architecture is the mapping from partitions to scratchpad BRAMs. In an architecture with a simple register file or a scratchpad that did not allow unaligned accesses, each BRAM would get its offset from a fixed partition; in a V2 with two partitions lane 1 would get its offset from partition 1and lane 2 from partition 2. However, as mentioned in Section 2.2, our scratchpad supports unaligned addresses. This means that partitions are not directly associated with a scratchpad BRAM; instead, depending on the alignment of the vector operands a scratchpad BRAM address may come from any of the partitions. This means we had to implement an offset mapping network to map wavefront offsets to scratchpad BRAMs. With a single offset (full wavefront skipping) the same offset goes to all BRAMs, which is just a broadcast of the offset requiring no additional logic.

3.3 Application Example: Viola-Jones Face Detection

Figure 5 gives high level psuedocode for one of the benchmarks we've implemented, Viola-Jones face detection. Face detection attempts to detect a face at every (x,y) pixel location in an input image. To vectorize this, we will test several possible starting locations in parallel: a 2D vector of starting

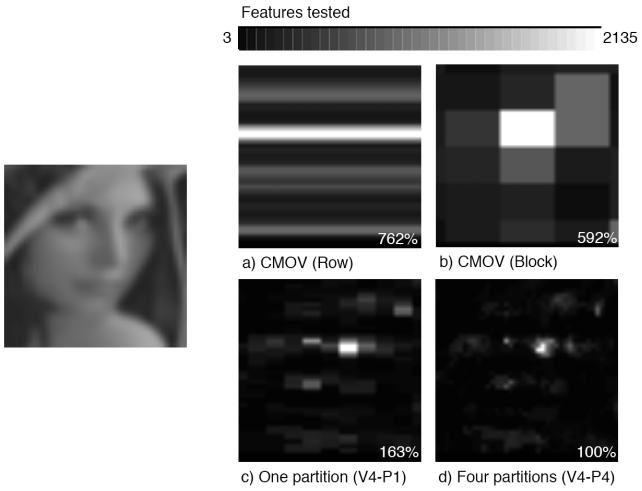

Figure 6: Haar Features Calculated for Different Groupings (Relative to Minimum)

locations characterized by (x+i,y+j). In this way, we will be testing for n^2 face locations in the vector simultaneously.

The vector/SIMD algorithm must compute several thousand values, called Haar features, for each pixel location. Features are grouped into stages. The features in a stage must pass a threshold test if a face is to be detected at that location. If any stage fails this threshold test, there is no point in testing further features at that location.

Features are calculated in-order across a vector of locations. Utilizing predicated instructions, we require processing the maximum number of stages required by any location within the vector. Only if there is no face at any of the locations we are testing, can we exit early. In this model, parallelism due to vectorization runs against the ability to exit early; longer vectors are more likely to contain locations requiring computation of many features, leading to extensive processing for all elements in the vector. By using wavefront skipping, we can avoid processing elements that are already known not to contain a face. Further features are computed only for those locations still in question, effectively shortening the vector length to the relevant elements.

The only differences between the predicated/CMOV version (without wavefront skipping) and this code are the init-mask and update-mask instructions and the low level details of the early exit test. Were we not to exit early the code would continue to run correctly, but the vector instructions would skip all wavefronts and effectively become NOPs. Porting an existing predicated algorithm to wavefront skipping is therefore straightforward and requires minimal effort.

Figure 6 shows the amount of work done for different locations; at minimum 3 Haar features need to be calculated, and at maximum 2135. The three order of magnitude difference shows the need for some smarter form of predication than simply computing the worst case on all pixels. The input face is shown on the left, with the other subfigures showing the number of Haar features calculated at each location. In order to run the application on an SVP, groups of pixels must be operated on as parallel vectors. Larger groupings provide more parallelism, but without wavefront skipping (only using CMOV instructions) every pixel in a

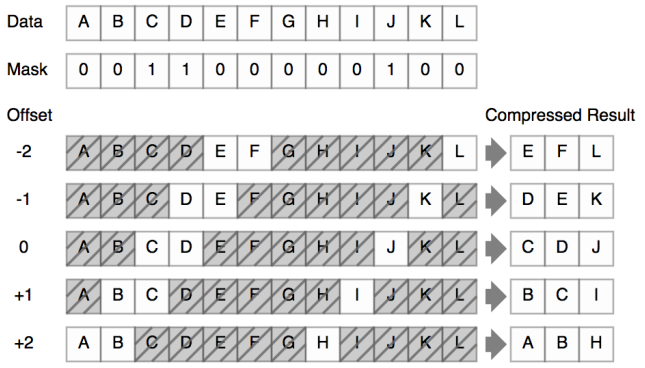

Figure 7: Vector Compress Operations Needed for 5x1 Stencil Filter

group must pass the maximum number of Haar feature tests of any pixels within the group. It can be difficult to balance the CMOV implementation; it is tempting to use the minimum vector length possible to reduce the amount of work done, but below a certain level the lack of parallelism means runtime actually increases. These results are from a 4 lane MXP implementation, and we found the runtime optimal vector length for CMOV implementations empirically.

A naive vector layout would be to have every row be a vector (6a), which does almost 8× as much work as the optimal. A slightly better method uses rectangular blocks to get better spatial locality (6b); getting sufficient parallelism still requires vectors long enough that almost 6× as much work as the minimum is done. In contrast, simply by changing the implementation to use wavefront skipping, the effective grouping is the wavefront partition width (6c). Wavefront skipping removes the need to profile an application to determine the best vector length to use; the wavefront skipping implementation can use the longest vector length possible and will always achieve at least the performance of the CMOV implementation. Finally, a fully partitioned design performs the minimal number of feature tests (6d), the same as a scalar implementation would.

3.4 Comparison with Vector Compress

An alternative method mentioned in Section 2.3 is a vector compress operation. The compress operation removes masked-off elements from a vector, creating a shorter vector as a result. Figure 7 demonstrates how this would work for a simple 5x1 stencil filter on our architecture. The stencil filter takes as inputs shifted versions of the input data, from an offset of -2 to +2. Before running the stencil filter, each of the inputs must be separately compressed. The 5 offseted versions of the data must be compressed into 5 scratchpad locations, each of which uses as much storage as the original in the worst case. After compressing the inputs, subsequent vector operations can operate on the shorter vectors at full speed.

The drawbacks of requiring a compress instruction and additional storage space for each input make it impractical in many stencil filter algorithms. For instance, on the Viola-Jones face detection application the amount of work and extra space needed would be prohibitive. The Haar feature tests operate on a sliding 20x20 window; each feature selects a subset of the window. Compressing the inputs would require compressing each location, or 400 compress operations and 400 temporary vectors. The wavefront skipping method does not need to manipulate the input data, meaning MXP only needs one copy of the chunk of the image being scanned.

Compress operations also need the inverse vector expand operation to restore the output. This is less critical for algorithms like stencil filters where the number of outputs is fewer than the number of inputs. Note that the compress operation is only of use when the vector being compressed is used multiple times, because compressing the vector takes an extra instruction. Setting up a mask for wavefront skipping also takes an extra instruction, but the same mask can be used for multiple offsets in a stencil filter.

4. RESULTS

Our results were obtained using Altera Stratix IV GX230 FPGAs on the Terasic DE4-230 development board. FPGA builds were done using Quartus II 13.0sp1. We used one 64-bit DDR2 channel as our external memory.

4.1 Area Results

Table 1 shows the resources used and maximum frequency achieved for various configurations of MXP. In addition to the actual FPGA resources used we also report the area in equivalent ALMs (eALMs) [12]. By factoring the approximate silicon area of all of the resources used, eALMs are a convenient way to compare architectures that use different mixtures of logic, memory, and multipliers.

All MXP configurations shown have a maximum masked vector length (MMVL) of 256 wavefronts (the effect of MMVL on area and benchmark results will be investigated later). MXP configurations are listed as VX PY where X is the number of 32-bit vector lanes and Y is the number of mask partitions. P0 means that masked instructions are disabled and only CMOV instructions can be used for conditionals. V1s are configured with 64kB of scratchpad, V4s 128kB, and V32s 256kB. The area numbers for MXP include the Nios scalar core used for control flow and vector instruction dispatch, as well as prefix sum and square root custom vector instructions that are only used in the face detection benchmark.

The eALM area penalty is 5.0% from no wavefront skipping to wavefront skipping with full wavefront skipping on a V1, mostly in ALMs. Since the number of ALMs for full waverfront skipping is roughly constant with respect to the number of lanes, the overhead drops to 3.6% on a V4 and <1% on a V32. For multiple partitions, the area overhead is much higher (up to 27.3% more eALMs in the V32-P32 case), because of the partition to scratchpad BRAM mapping needed (as explained in Section 3.2). We tried implementing this mapping using both a switching network and multiplexers; the area results were similar but the multiplexer implementation had lower latency and so was used for our results.

4.2 BRAM Usage

BRAM usage is minimal for the V1 and V4 as each mask partition uses just 1 BRAM. For the V32, a single partition uses 4 BRAMs since it has to store 128-bits worth of byte enables, 8 bits of offset, and 1 end bit, which fits in four 36-bit wide M9Ks. With more partitions, the number of byte enables per partition is reduced, so that the V32-P32 only

Table 1: Resource Usage

MXP Configuration Vector Lanes - Wavefront Partitions	Logic ALMs	Memory M9Ks	DSP Blocks	Total Area eALMs	Area Increase % Over CMOV	fmax MHz
(CMOV) V1-P0	4,697	96	2	7,502		206
V1-P1	5,034	97	2	7,877	5.0%	200
(CMOV) V4-P0	9,732	152	5	14,243		173
V4-P1	10,210	153	5	14,750	3.6%	176
V4-P4	13,276	156	5	17,902	25.7%	183
(CMOV) V32-P0	63,334	414	33	76,198		144
V32-P1	63,027	418	33	76,005	-0.3%	144
V32-P4	78,698	422	33	91,791	20.5%	149
V32-P32	83,220	446	33	97,002	27.3%	146
Nios II/f	1,370	19	1	1,945		283
DE4230 Maximum	91,200	1,235	161	131,434		–

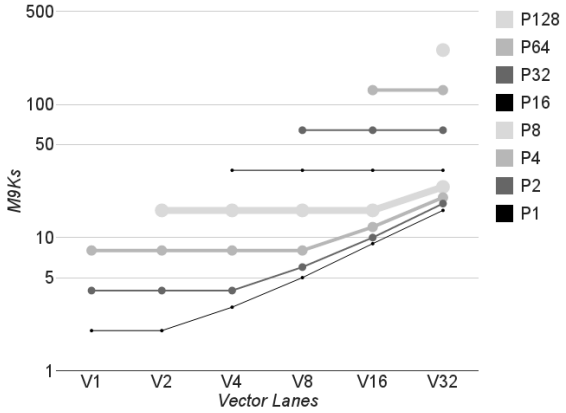

Figure 8: BRAM Usage vs Wavefront Partitions (MMVL = 1024)

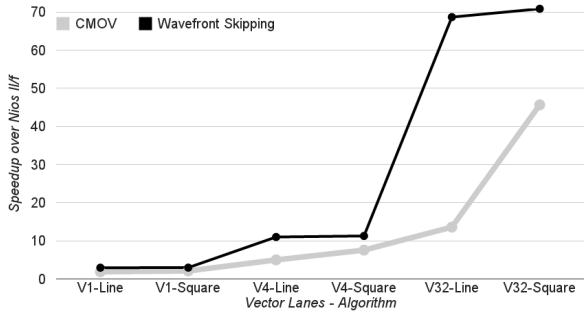

Figure 9: Mandelbrot Benchmark

uses 1 BRAM per partition. Although not shown, the V32 can use up to 128 partitions; this would require 128 more BRAMS than the 414 used in the P0 version. The masked unit has no critical timing paths; increasing the number of BRAMs used does make the placement job harder for the CAD tools, but the critical path always remains outside the masked unit. Since the contribution of this work is not affecting Fmax directly, we decided to remove this variability from our results by running all benchmarks shown here at 125MHz.

Figure 8 shows how many BRAMs are used in our partitioned wavefront skipping with a maximum masked vector length (MMVL) of 1024 wavefronts. With a small number of partitions, as the vector processor gets wider, multiple BRAMs per partition are required to store the byte enables. With a large number of partitions, the byte enables are divided up into small enough chunks that they always fit in a single BRAM per partition. This wide range in the number of BRAMs that can be used gives the designer freedom to allocate BRAMs on the FPGA device to achieve the needed performance for their algorithm.

4.3 Mandelbrot Benchmark

Figure 9 shows speedup results for computing the Mandelbrot set (geometric mean of 23 frames in a flyby demo). Results are shown compared to a scalar version run on the Nios II/f as a baseline. The Mandelbrot computation iterates a complex valued equation at each pixel until either the pixel reaches a set condition (the early exit) or else a maximum iteration count is reached. Without masked instructions (CMOV configurations) we can only exit early after all pixels in the group agree to exit early. We show two results, the 'line' implementation which naively computes pixels in raster order (row by row), and the 'square' implementation which computes a 2D block of pixels at a time. The 'line' implementation is more straightforward and is what we first used when writing this code. However, since the early exit pixels are correlated spatially, selecting a group of pixels closer together in a block results in less wasted computation and therefore higher performance.

For the CMOV configurations, the difference between the line and square implementations is vast; almost a 4× difference at V32. With the masked implementations, the difference between line and square are much less; 10% at most. The masked implementations always outperform the CMOV implementations. Partitioned wavefront skipping has little effect in Mandelbrot and so is not shown; the data is grouped together smoothly so that wavefronts exit at the same or nearly the same time. The CMOV implementation also uses a vector length determined by profiling; too long and early exits aren't helpful, too short and instruction dispatch rate and data hazards reduce performance. The masked implementation just uses the maximum vector length it can, which is either determined by the MMVL of the masked instructions or the size of the scratchpad and number of vectors needed. Between not having to profile to find the best vector length, and being able to use the naive 'line' implementation with little performance penalty, the masked implementation seems much easier for the programmer.

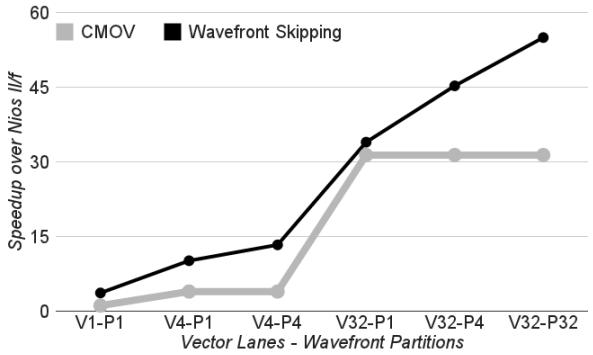

Figure 10: FAST9 Feature Detection

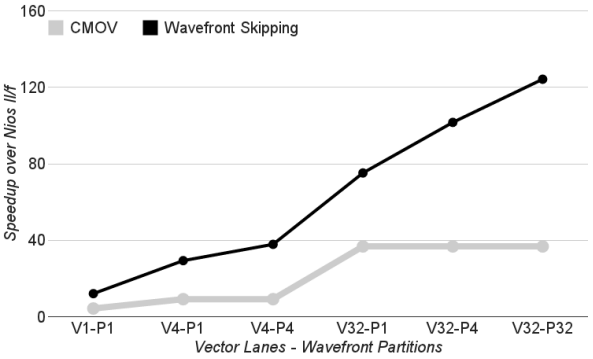

Figure 11: Viola-Jones Face Detection Speedup

4.4 FAST9 Feature Detection

Figure 10 shows the results of FAST9 feature detection. FAST9 determines if a pixel is a feature by examinining a circle of pixels around the target location and looking for a consecutive series of pixels that are darker or lighter by a threshold. An early exit can happen if certain conditions hold, such as if neither of two pixels on opposite sides are above or below the threshold. All operations are byte-wide, taking advantage of MXPs subword-SIMD which executes byte operations 4× faster than word operations. While running FAST9 on simple input data with the CMOV code, we found checking early exit (as done in Mandelbrot) to be helpful. However, on a real image (Lenna again), the early exit code only helped on a V1. Thus, the CMOV code is doing the full calculation for all pixels on V4 and V32. In contrast, the masked code is able to achieve speedup by skipping at a more granular level. However, if the mask is too coarse, such as at V32-P1 which is 128 bytes wide, very little is gained. Only with 4 or 32 partitions is speedup achieved, and it may be possible to gain even more speed by using byte-wide partitions (our test configurations had a minimum partition size of one 32-bit lane wide).

4.5 Viola-Jones Face Detection

For Viola-Jones face detection, we used the settings of [2], a custom FPGA implementation using the same DE4 development board we are using. No reference image was specified in the publication, so we used the standard 'Lenna' image. Their hardware implementation with 32 PEs was able to achieve 30fps. Our SVP implementation has an inner loop

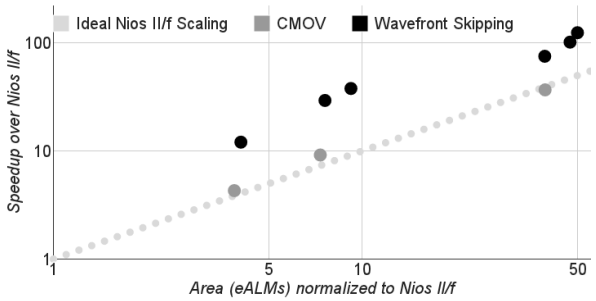

Figure 12: Viola-Jones Face Detection Speedup Vs Area

of 9 or 13 instructions long (depending on the Haar classifier), instead of being fully pipelined as in a hardware implementation, so it is expected to be somewhat slower. Also, our implementation includes full data movement, from input image to framebuffers driving a DVI display.

Figure 11 shows the results for CMOV only and masked implementations including partial wavefront skipping. The masked implementations with a single partition are up to 3× faster than CMOV, and partitioning increases it to up to 4.1×. At V4, partitioning gives a 29% improvement from P1 to P4, and at V32 a 35% improvement from P1 to P4 and an additional 22% improvement from P4 to P32. While the impact of partitioning is not as dramatic as the impact of switching from CMOV instructions to wavefront skipping, it provides a performance increase without having to rewrite software.

Figure 12 presents the results as speedup versus area (eALMs). MXP without wavefront skipping is only able to achieve about the performance per area of a Nios II/f (which in practice can't scale ideally to achieve higher performance). Wavefront skipping provides significantly higher performance per area, since the area impact is minimal compared to the performance gained. The highest performance per area is 4.1× that of Nios II/f (37.8× faster with 9.2× more area for the V4-P4 configuration).

With our fastest configuration we achieve 3 frames per second, or about 1/10th that of the previous hardware result, on the Lenna input image. A blank image takes 60 frames per second; even though all locations exit early, the Viola-Jones algorithm requires multiple steps of resizing and calculating integral images.

4.6 Maximum Masked Vector Length (MMVL) Tradeoffs

So far we have examined tradeoffs in additional logic and BRAMs when using multiple partitions, but it's also possible to use more RAM to allow for longer masked vector instructions as mentioned in Section 3.1. Each BRAM needs to store an offset of width $log_2(MMVL)$ as well 4 byte enables per lane and an end bit; when partitioning, the byte enables are split between BRAMs but the other data is replicated. As explained in Section 3, we default to a MMVL of 256 wavefronts since that is the minimum depth of M9Ks. Figure 13 shows the BRAM usage for two configurations of MXP as the MMVL is varied. A MMVL of 128 or 256 wavefronts almost always has the same resource usage, except on a V16 with one partition where the width of the BRAM data

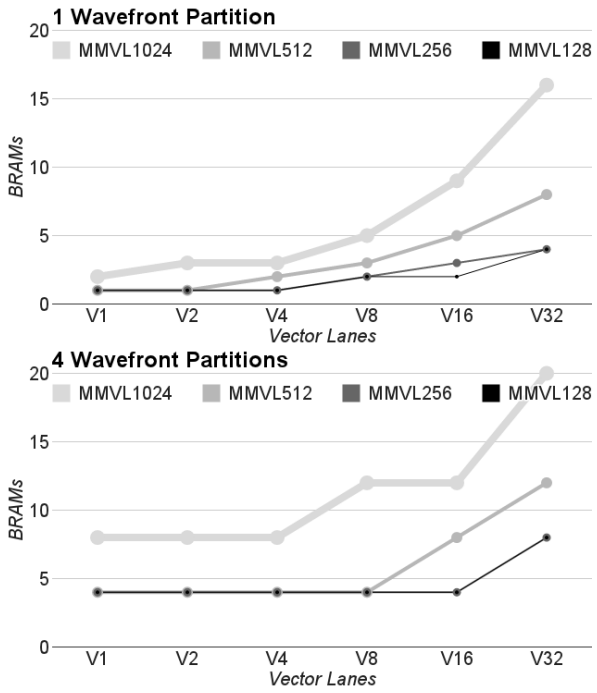

Figure 13: BRAM Usage for Masks When Varying Maximum Masked Vector Length (MMVL) for 1 (top) and 4 (bottom) Partitions

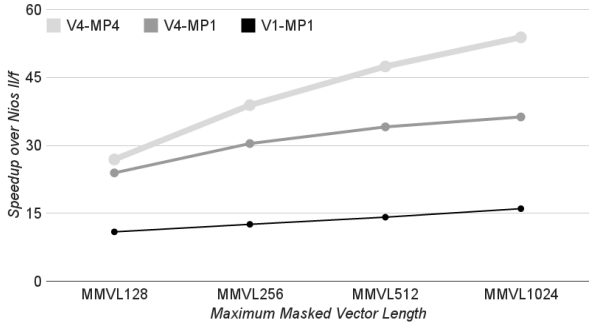

Figure 14: Effect of Changing MMVL on Viola-Jones Face Detection

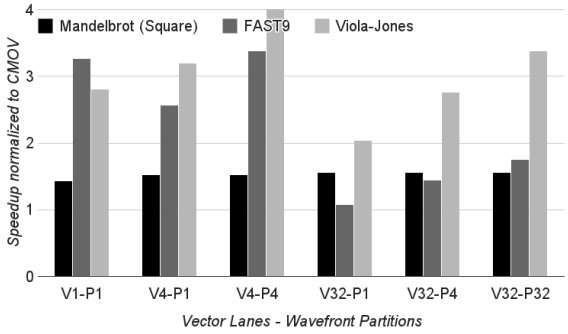

Figure 15: Speedup from Wavefront Skipping

goes from 72 and 73 bits and can no longer fit in two 36-bit wide M9Ks. Increasing MMVL to 512 does have a cost in BRAMs though once the width of the BRAM data gets past 18-bits, with an MMVL of 1024 requiring even more as BRAMs are only 9-bits wide at that depth.

Figure 14 shows the results of changing MMVL on the face detection benchmark. For V1, there is a reasonable gain to be had in increasing MMVL; 16% better performance from 128 wavefronts to 256, with an additional 13% improvement going from 256 wavefronts to 512. The results are even more dramatic on a V4: 100% faster from MMVL of 128 to 1024 with 4 mask partitions. The fact that a V4-P4 with MMVL 128 is actually slower than a V4-P1 with MMVL 256+ suggests that using BRAM for increased depth is better than using it for more partitions. The reason that increased MMVL makes such a large difference is that as masked instructions have fewer and fewer valid wavefronts, they start taking so few cycles that efficiency drops (either data hazards or instruction dispatch rates dominate). Longer MMVL means that fewer masked instructions are needed, and less time is spent in this low efficiency regime. However, by V32, masked vector length is no longer limited by MMVL but by the number of vectors needed and the size of the scratchpad. In this case, changing MMVL had no effect on the results so they are not shown. The trend for V1 and V4 does suggest that increased vector length (in this case by having a larger scratchpad) would provide higher performance on this application, though.

4.7 Results Summary

Figure 15 summarizes the speedup of wavefront skipping over the previous CMOV implementation on our three benchmarks, while Figure 16 shows the speedup per area (eALMs). The maximum speedups for a single partition is 3.3× on FAST9 (V1-P1) and 3.2× on Viola-Jones (V4-P1). The biggest speedup from partitioning over full wavefront skipping is 65% on Viola-Jones (V32-P32 vs V32-P1). All of the benchmarks gain at least as much in speedup as the cost in area of the wavefront skipping implementation, with the minimum gain being 1.1× better performance per area for FAST9 on a V32 with one wavefront partition. Though increased partitions cause a large increase in the area of MXP, for FAST9 and Viola-Jones the performance per area continues to increase as the number of partitions increases. Wavefront skipping is therefore useful for all the benchmarks we tested, and partitioning useful for two of the three.

For a programmer implementing a conditional algorithm, the configuration of BRAM usage for scratchpad size, wavefront partitions, and maximum masked vector length will depend on the application. The designer will generally want to devote as much BRAM as possible to scratchpad data, as longer vectors will benefit all parts of an application. But, for heavily conditional applications like Viola-Jones, having long enough masked vectors to keep the vector core busy as the number of valid wavefronts shrink is also important. Few BRAMs are needed for just a single partition; using multiple partitions adds several BRAMs, but this may be justified when performance is absolutely necessary or the targetted FPGA has leftover BRAMs with a given design.

5. FUTURE WORK

We mentioned in Section 4.6 that it would be possible to support multiple masks at the same time. Multiple masks would allow for a greater degree of freedom in control flow

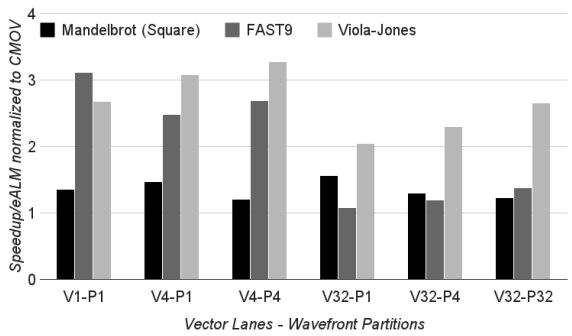

Figure 16: Speedup per Area (eALMs) from Wavefront Skipping

before having to save and restore the mask contents. With a short enough MMVL, multiple masks could share a BRAM.

We may also be able to repurpose our multiple partition addressing logic to increase the speed of transposed matrix accesses. Strided accesses already run at full speed provided they do not cause bank conflicts. We could similarly do transposed accesses for matrices of the correct dimensions ($2^N \pm 1$) or that were padded to the correct dimensions. However, in this case, we would want to write out the destination at different offsets than we read the input, requiring a different addressing mode or separate offsets for each operand.

6. CONCLUSIONS

This work has shown that integrating wavefront skipping into soft vector processors (SVPs) can be done efficiently in terms of logic and BRAM usage. Wavefront skipping not only allows for higher performance due to skipping masked off elements, it is much easier to use than checking early exit conditions on blocks of elements using predicated/CMOV instructions. Our implementation stores offsets in BRAMs, which are relatively cheaply available and high-performance in FPGAs. The alternative, which uses a count-leading-zeros operation, would have higher latency and also limit the number of wavefronts skipped in one cycle. Our approach keeps the mask logic simple and off the critical path, and can also skip an arbitrary number of wavefronts.

When not partitioned, our wavefront skipping implementation uses <5% extra area and can give up to 3.2× better performance on Viola-Jones face detection. Extra logic and BRAMs can be used to gain additional performance by partitioning, allowing parts of a wavefront to have different offsets. Though costly in terms of area, partitioning can give up to 65% extra performance. Partitioned wavefront skipping may not be a reasonable design tradeoff in a fixed vector processor. In an FPGA, partitioned wavefront skipping gives a designer an extra tool to tradeoff additional logic and BRAMs for application specific performance.

7. ACKNOWLEDGMENTS

The authors would like to thank Altera for donating hardware and software licenses used in this research, and MITACS and NSERC for providing funding.

8. REFERENCES

[1] The Convey HC-2 architectural overview. http://www.conveycomputer.com/index.php/download_file/view/143/142/.

[2] B. Brousseau and J. Rose. An energy-efficient, fast FPGA hardware architecture for OpenCV-compatible object detection. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 166–173, Dec 2012.

[3] R. Espasa, M. Valero, and J. E. Smith. Vector architectures: Past, present and future. In *Proceedings of the 12th International Conference on Supercomputing*, pages 425–432, 1998.

[4] W. Fung and T. Aamodt. Thread block compaction for efficient SIMT control flow. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 25–36, Feb 2011.

[5] J. Gebis, L. Oliker, J. Shalf, S. Williams, and K. Yelick. Improving memory subsystem performance using ViVA: Virtual vector architecture. In *Architecture of Computing Systems*, pages 146–158. 2009.

[6] C. Kozyrakis and D. Patterson. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. In *Microarchitecture*, pages 283–293, 2002.

[7] R. Lorie and H. Strong. Method for conditional branch execution in SIMD vector processors, Mar. 6 1984. US Patent 4,435,758.

[8] M. Naylor, P. Fox, A. Markettos, and S. Moore. Managing the FPGA memory wall: Custom computing or vector processing? In *Field Programmable Logic and Applications (FPL), 23rd International Conference on*, 2013.

[9] A. Severance, J. Edwards, H. Omidian, and G. Lemieux. Soft vector processors with streaming pipelines. In *The 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2014.

[10] A. Severance and G. Lemieux. TputCache: High-frequency, multi-way cache for high-throughput FPGA applications. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, 2013.

[11] J. E. Smith, G. Faanes, and R. Sugumar. Vector instruction set support for conditional operations. *SIGARCH Comput. Archit. News*, pages 260–269, 2000.

[12] H. Wong, V. Betz, and J. Rose. Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 5–14, New York, NY, USA, 2011. ACM.

[13] P. Yiannacouras, J. G. Steffan, and J. Rose. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *CASES*, 2008.

[14] J. Yu, C. Eagleston, C. H. Chou, M. Perreault, and G. Lemieux. Vector processing as a soft processor accelerator. *ACM TRETS*, 2(2):1–34, 2009.