Embedded Supercomputing in FPGAs with the VectorBlox MXP Matrix Processor

Aaron Severance
University of British Columbia,
VectorBlox Computing Inc.
aaronsev@ece.ubc.ca, aseverance@vectorblox.com

Guy G.F. Lemieux
University of British Columbia,
VectorBlox Computing Inc.
lemieux@ece.ubc.ca, glemieux@vectorblox.com

Abstract

Embedded systems frequently use FPGAs to perform highly parallel data processing tasks. However, building such a system usually requires specialized hardware design skills with VHDL or Verilog. Instead, this paper presents the VectorBlox MXP Matrix Processor, an FPGA-based soft processor capable of highly parallel execution. Programmed entirely in C, the MXP is capable of executing data-parallel software algorithms at hardware-like speeds. For example, the MXP running at 200MHz or higher can implement a multi-tap FIR filter and output 1 element per clock cycle. MXP's parameterized design lets the user specify the amount of parallelism required, ranging from 1 to 128 or more parallel ALUs. Key features of the MXP include a parallel-access scratchpad memory to hold vector data and high-throughput DMA and scatter/gather engines. To provide extreme performance, the processor is expandable with custom vector instructions and custom DMA filters. Finally, the MXP seamlessly ties into existing Altera and Xilinx development flows, simplifying system creation and deployment.

1. Introduction

Modern embedded systems often contain high bandwidth I/O devices and require extensive computing resources. One example is smart cameras with built-in video analytics. This in-device or *edge-based* processing can reduce wireless bandwidth costs and power-hungry server farms.

Furthermore, these embedded devices are almost always built around an FPGA. The FPGA is used to control I/O, particularly image sensors or similar arrays connected to high-speed DACs and ADCs. When performance demands are fairly low, a soft processor in the FPGA is often sufficient. The fastest available soft processors are pipelined, in-order RISC processors running up to 250MHz. There are many advantages to this software-based approach, primarily the ease of developing in C and the rapid edit-compile-debug loop. Unfortunately, attempts to build even faster processors have failed due to the difficulty of implementing multiple write ports and high clock rates in FPGAs.

The next level of performance is achieved by transferring the computation to an off-chip CPU or DSP. This preserves all of the advantages of a software-based development model. However, there are downsides as well: limited performance, increased circuit board area, a new chip in the BoM and inventory with possibly limited future re-use, and device procurement. Also, if the CPU/DSP has specialized memory or voltage requirements, there may be additional cost and complexity for increased on-board memory or voltage regulators.

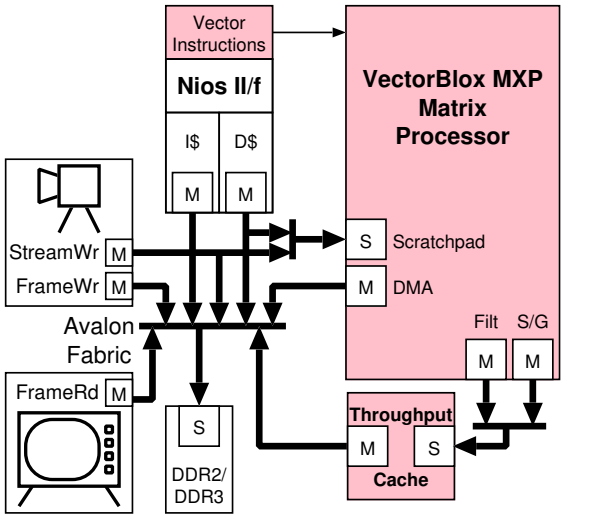

Figure 1. Typical VectorBlox MXP system (Altera version)

The final solution is designing a circuit at the RTL level to do the processing inside the FPGA. However, in addition to designing with a hardware description language such as Verilog or VHDL, this also requires performing hardware-level debug and waiting for slow FPGA place-and-route iterations after every design change.

To mitigate the need to create RTL, we have designed the VectorBlox MXP Matrix Processor as an embedded supercomputer to greatly accelerate what can be done with Nios II/f or MicroBlaze soft processors. Using a vector-oriented data-parallel programming model, it is programmed entirely in C/C++ and produces speedups of up to 1000×. The ability to add custom vector instructions offer potential for an additional 10-100× speedup. These levels of performance are not possible with any other FPGA-based processor, including FPGAs with the new hard dual-core ARM A9s.

A high-level block diagram of a VectorBlox MXP system is given in Figure 1. In the figure, an Altera-based system using the Avalon interconnect fabric is shown, but we also support Xilinx and the AXI interconnect fabric. The system is based around a fast Nios II/f processor with its direct-mapped instruction (I$) and data caches (D$). The blocks labelled 'M' and 'S' are master and

978-1-4799-1417-3/13/$31.00 (C)

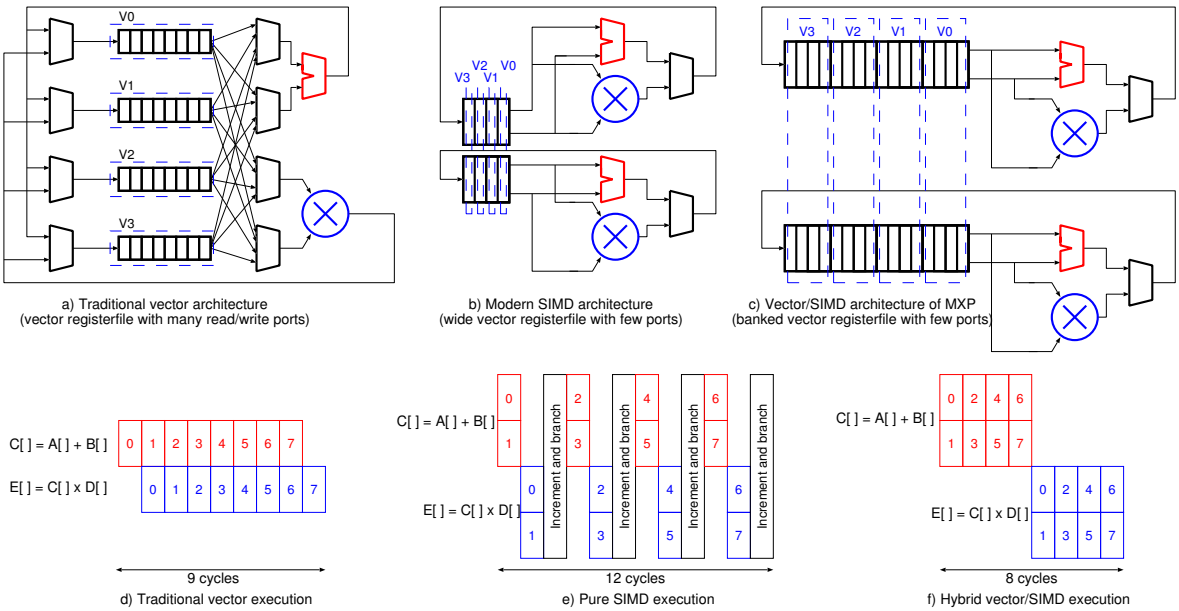

Figure 2. Vector and SIMD modes of execution

slave ports for the Avalon interconnect; a master issues an address for either reading or writing. The camera shown can inject data either using a FrameWriter or StreamWriter module. The former writes an entire frame to external DDR DRAM for processing, while the latter writes a few scanlines at a time directly to either the MXP scratchpad or to external memory. A FrameReader module continuosly reads a framebuffer from DRAM for the output display. The MXP components consist of a parameterizable vector engine and optional scatter/gather cache. A vector engine with as few as 16 parallel ALUs running at 200MHz can perform many interesting transforms in real-time on 1080p60 video (1920x1080 at 60fps).

A number of features contribute to the novelty of the MXP architecture:

- scratchpad to hold vector data
- fracturable ALUs to operate on bytes, halfwords or words
- 1D (vector), 2D (matrix) or 3D (volume) operations
- DMA engine with 1D or 2D transfers
- automatic alignment of data
- datasize conversion in-flight with operations
- decoupled scalar/vector/DMA cores
- custom DMA filters
- double-buffering hides most memory latency
- scatter/gather support with high-throughput cache
- custom vector instructions for increased parallelism
- scalable processor size for performance
- easy-to-use parallel programmming model

The VectorBlox MXP is available as an FPGA IP core which is added to any Altera or Xilinx FPGA design using Avalon or AXI interconnects, respectively. To our knowledge, it is the only commercially oriented soft processor capable of achieving significant speedup.

2. Vector Processing and SIMD Background

Vector processing has been around since the late 1960s when it was applied to supercomputing. The first commercially successful vector supercomputer was the Cray-1 [3]. Running at 80 MHz, it supported 64-bit word size and a 64-bit, 8MB ECC memory subsystem with 16-way interleaving. With a peak throughput of 250MFLOPs, and sustained throughput of 138MFLOPs, it was faster than any other computer in the mid-1970s.

One of the strengths of the Cray-1 architecture is the load/store RISC-like architecture to vector processing. Prior vector machines stored vectors in main memory. Instead, Cray uses a fast 4kB vector register file arranged as 8 named vector registers that are 64 elements long and 64 bits wide. Since each vector instruction takes up to 64 clock cycles to execute, high throughput is achieved by pipelining. The CPU contains only one instance of each functional unit, so increased parallelism is provided by operator chaining, where the output of one pipeline is sent to another pipeline before the complete vector is written out to the register file. In addition to chaining, the ability to overlap vector-load or vector-store operations with vector arithmetic operations provided memory level parallelism for increased performance.

Recently, there has been increased attention given to SIMD-oriented instructions. These are often called multimedia instructions, or sometimes vector extensions. However, the vectors these instructions operate upon are obtained by dividing a fixed processing width into sub-units; for example, Intel's latest mainstream vector extensions (AVX) are 256 bits wide while the extensions used in the Xeon Phi device are 512 bits wide. A key aspect of SIMD extensions is the need to iterate to cover a longer vector; for best performance, the CPU must issue and execute loop iteration instructions concurrently (in the superscalar sense) along with the SIMD instructions. To execute in a streaming fashion, these processors must also execute SIMD-load and SIMD-store instructions concurrently. Finally, there is considerable instruction-level overhead to pack/unpack and manipulate byte/word orderings within a SIMD payload.

In contrast with traditional vector and modern SIMD instructions, the VectorBlox MXP takes a hybrid approach. First, it takes

the SIMD approach by creating a wide data engine that can be subdivided into words, halfwords, or bytes; we have built engines up to 4096 bits wide. This can also be described as 128 *vector lanes*, where one vector lane contains a full 32-bit datapath. Second, it takes the vector approach of adding hardware iteration to eliminate loop and control overhead. This allows us to define vectors that are long as well as wide. We illustrate the three choices taken in Figure 2. In part (a), the vector approach is bottlenecked by a need for multiple read and write ports. In part (b), the SIMD approach is shown, but significant loop overhead and the need to constantly reload SIMD registers can be a problem. In part (c), our approach with long, wide vectors is shown.

On top of the hybrid vector/SIMD model, we eliminate the rigidly defined register files from both approaches and instead use a flexible, multi-banked scratchpad; in essence, we have rejected the RISC load/store approach taken by the Cray-1 and most vector processors since then and instead regressed to the memory-based vectors architectures developed before Cray. As will be shown, our on-chip scratchpad choice provides significant freedom for programming and is often beneficial for performance while maintaining the locality and fixed latency of a register file.

The next section explores VectorBlox MXP architectural details.

3. VectorBlox MXP High-level Architecture

In Figure 1, the VectorBlox MXP is shown added to an existing Altera Nios II/f based system. Within the Altera environment it utilizes the Avalon interconnect to access memory and devices. Within the Xilinx environment, it can be added to an existing MicroBlaze based system, and utilizes the AXI interconnect. All vendor-provided tools, such as IDEs and debuggers, can be used normally by the programmer. To access accelerated features, the program must contain specific calls to invoke the MXP accelerator. This can be done by calling MXP specific library routines or by using MXP intrinsic functions placed inline with regular C/C++ code.

3.1 Overview

The VectorBlox MXP is a scalable processor that uses a blend of traditional vector processing and SIMD-style execution to operate upon vectors, arrays, and volumes of data. The user can specify the number of 32-bit ALU datapaths to create in parallel; we call this the number of lanes. We denote the size of a processor engine as $V4$, for example, when it contains 4 vector lanes; alternatively, this could be described as a 128-bit wide vector-SIMD engine.

The MXP is a fixed-point processor. It supports data sizes of 8 (byte), 16 (halfword) and 32 (word) bits. All operations have both signed and unsigned variants. In a traditional RISC processor, operations on bytes or halfwords are the same speed or slower than operations on integers because they are expanded to fill the contents of a 32-bit register. In MXP, operands remain at their original size, requiring significantly less storage when operating on vectors of 8-bit or 16-bit data. Furthermore, the 32-bit ALUs can be fractured into smaller 8-bit or 16-bit ALUs, providing increased parallelism on smaller operands. Due to area overhead, divide and modulo are not supported by default. However, they can be optionally added using a custom vector instruction (to be discussed later). As well, a full suite of logical, arithmetic, multiply, shift/rotate, and conditional move instructions are supported.

Central to the MXP is the vector data storage area, called the *vector scratchpad*. The scratchpad can be sized independently of the number of lanes, spanning a range from 4kB to the maximum memory of the FPGA device (which can 2MB or more). Practical sizes will depend on the FPGA memory blocks used, which come in a variety of sizes depending upon the device. For example, the Altera Cyclone IV device contains 9kb memory blocks. To support unaligned accesses at the byte level, we configure these to be byte wide (arranged as 1024 x 9); four are needed for each each vector lane, so 4kB of scratchpad data per vector lane is the practical minimum for these devices. We have found this amount (4kB/lane) to be a useful value in almost all of our scalable software applications; less scratchpad does not allow for long enough vectors to be stored while more has minor, diminishing gains. We make use of the 9th bit of the memory blocks as a per byte condition code, so the memory is fully utilized.

The next important subsystem is a dedicated *DMA engine* that is used to fill or empty the scratchpad storage. The DMA engine transfers blocks of data between the MXP and any other memory-mapped device. It supports long bursts as well as 2D windowed operations; a windowed operation is the repeated operation of a burst followed by a fixed gap or stride. For maximum flexibility different strides can be applied to the source and destination.

It is important to note that the scalar host processor, the vector engine, and the DMA engine all operate concurrently. This is accomplished by having the scalar host processor (Nios II or MicroBlaze) deposit vector and DMA instructions into a processing queue. If the queue is full, or if the user performs an explicit synchronization, the scalar host will stall until both the vector and DMA engines have completed their operations. Otherwise, the vector and DMA operations will be removed from the queue in-order, but completed out-of-order relative to each other and the scalar host. Hardware interlocks between the vector and DMA engine ensure that the original program order is maintained by stalling the respective engine when there is a hazard. In all cases, the vector engine executes instructions in-order, as does the DMA engine.

3.2 System Design

Since we have not modified the host Nios or MicroBlaze development environment, users do not have to learn new tools. Instead, they merely learn to use our software library system. Furthermore, since the MXP is placed in an existing Avalon or AXI computer system, which are designed using a GUI environment, virtually no hardware design skills are needed to build a system.

When first deploying a MXP in a Nios or MicroBlaze environment, it must be configured. The MXP has a few simple parameters that govern its size and operation:

- number of lanes
- size of scratchpad
- width of interconnect path to memory
- length of memory bursts
- multiplier implementation style
- fixed-point radix position

The first two parameters have already been discussed. The width to memory and the length of a memory burst, affect the interconnect that will be generated via Avalon/AXI to connect with off-chip memory; it has a direct impact on the maximum bandwidth to memory. The multiplier implementation style will be discussed later, but it allows trade-off between performance and the number of hard multiplier blocks used. Finally, the processor can accelerate fixed-point operations for a specified radix position; other radix positions can still be used at runtime but must be handled by software.

A program written for the MXP architecture can run on any size vector engine. The inability to run the same program on different processors was a key downfall of early VLIW processors, and it is a key weakness of the ever-widening SIMD instructions.[1] MXP's scratchpad size is a hard limit, but programs can be written to

[1] In addition, Intel has progressed through several generations of SIMD instructions, from MMX to SSE versions 1 through 4 (and sub-versions), and

a) Plain Nios II/f

```
for(j=0; j<DATA_SIZE; j++){
 out[j] = 0;
 for(i=0; i<NUM_COEFFS; i++){
   out[j] += in[j+i]*coeffs[i];
 }
}
```

```
;; Compiled with gcc 4.5.3 -O3
20003d4: stw  zero,0(r9)
20003d8: bne  r13,zero,2000414
20003dc: mov  r8,zero
20003e0: mov  r6,r11
20003e4: mov  r5,r15
20003e8: mov  r7,r8
20003ec: ldw  r2,0(r6)
20003f0: ldw  r3,0(r5)
20003f4: addi r8,r8,1
20003f8: addi r6,r6,4
20003fc: mul  r2,r2,r3
2000400: addi r5,r5,4
2000404: add  r2,r7,r2
2000408: mov  r7,r2
200040c: stw  r2,0(r9)
2000410: bne  r10,r8,20003ec
2000414: addi r12,r12,1
2000418: addi r9,r9,4
200041c: addi r11,r11,4
2000420: bne  r14,r12,20003d4
```

b) Intel i7 AVX

```
for(j=0; j<DATA_SIZE; j++){
 out[j] = 0;
 for(i=0; i<NUM_COEFFS; i++){
   out[j] += in[j+i]*coeffs[i];
 }
}
```

```
;; Compiled with gcc 4.6.3
;; -O2 -ftree-vectorize -m64
;; -march=corei7-avx
;; -mtune=corei7-avx
.L23:
  leaq    (%rdi,%rsi), %rcx
  xorl    %eax, %eax
  vpxor   %xmm0, %xmm0, %xmm0
.L22:
  vmovdqu coeffs(%rax), %xmm1
  vmovdqu (%rcx,%rax), %xmm2
  addq    $4, %rax
  vmovd   %xmm1, %edx
  cmpq    $64, %rax
  vmovd   %edx, %xmm3
  vpshufd $0, %xmm3, %xmm1
  vpmulld %xmm1, %xmm2, %xmm1
  vpaddd  %xmm1, %xmm0, %xmm0
  jne     .L22
  vmovdqa %xmm0, (%rsi)
  addq    $16, %rsi
  cmpq    $output+1024, %rsi
  jne     .L23
```

c) MXP using 1D instruction

```
vbx_set_vl( NUM_COEFFS );
for(j=0; j<DATA_SIZE; j++){
 vbx_acc( VVW, VMUL, v_output+i,
          v_input+i, v_coeff );
}
```

```
;; Compiled with gcc 4.5.3 -O3

;; setup instructions
2000aec: ldw     r2,36(r4)
2000af0: mov     r4,r2
2000af4: call    <vbx_set_vl>
2000af8: mov     r5,zero
2000afc: mov     r4,zero

;; kernel instructions
2000b00: add     r2,r18,r4
2000b04: custom  0,c2,r2,r16
2000b08: add     r3,r20,r4
2000b0c: custom  1,c11,r3,c12
2000b10: addi    r5,r5,1
2000b14: addi    r4,r4,4
2000b18: bne     r17,r5,2000b00
```

d) MXP using 2D instruction

```
vbx_set_vl( NUM_COEFFS );
vbx_set_2D( DATA_SIZE, sizeof(int),
            sizeof(int), 0 );
vbx_acc_2D( VVW, VMUL, v_output,
            v_input, v_coeff );
```

```
;; Compiled with gcc 4.5.3 -O3

;; setup instructions
2000b7c: ldw     r2,36(r4)
2000b80: ldw     r17,32(r4)
2000b84: sub     r17,r17,r2
2000b88: mov     r4,r2
2000b8c: call    <vbx_set_vl>
2000b90: mov     r4,r17
2000b94: movi    r5,4
2000b98: mov     r6,r5
2000b9c: mov     r7,zero
2000ba0: call    <vbx_set_2D>

;; kernel instructions
2000ba4: custom  0,c6,r21,r20
2000ba8: custom  1,c11,r22,c12
```

Table 1. Comparison of FIR filter implementations

accommodate nearly any scratchpad size. Overall, this means the program does not need to be recompiled to run on a vector engine with more ALUs or different scratchpad sizes; it automatically scales up to utilize larger processors. A key aspect of MXP is that the vector storage bandwidth is automatically scaled to match the ALU processing bandwidth each time a different system size is generated.

Furthermore, programs are fully portable across FPGA device families and vendors, requiring only a recompile to execute code on a different host processor. We provide a vendor-indpendent library to assist with portability for common system calls.

3.3 Programming Example

To illustrate how to program the MXP, Table 1 compares various FIR filter implementations. The first part (a) illustrates plain C code and Nios II/f assembly code emitted by GCC, using a total of 10 instructions per iteration in the innermost loop. In contrast, part (b) illustrates the same C code compiled for an i7 using autovectorization. The innermost loop still contains 10 instructions, but each iteration performs 4 operations in parallel. Parts (c) and (d) show the code for both nested loops using 1D and 2D instructions, respectively. The kernel instructions are taken exactly from the compiler, but the setup instructions are modified for clarity. The inner loop in part (c) consists of two custom instructions at addresses `b04` and `b0c` while the outer loop is a total of 7 instructions per iteration. In part (d), both nested loops are replaced by two custom instructions. Hence, the entire FIR filter has been passed to the vector engine.

In terms of performance, the Nios II/f uses two cycles to execute one custom instruction. These two instructions form a single vector instruction, which the MXP can execute in as little as one cycle. If there are 8 coefficients and 8 vector lanes, part (c) will execute the innermost loop in 1 cycle and part (d) will execute both loops to produce one output data point per clock cycle. Most hardware implementations of FIR filters also achieve a speed of 1 cycle per output data point, so the MXP executes as fast as hardware in this case.

In the example above, the MXP assumed all data was already placed in the scratchpad. This is often the case in real programs because DMA runs in the background — it is used to prefetch the next vector of input data and write the output data back to external memory concurrently with the FIR filter execution. In streaming applications such as a camera, a stream writer can be used to write input data directly into the scratchpad.

4. VectorBlox MXP Architectural Details

Figure 3 provides a detailed view of the internal VectorBlox MXP architecture. The scalar host processor executes code written in C/C++, and places any vector or DMA operation into a work queue for the MXP. The precise mechanism to represent vector or DMA operations is different for each host: the Nios II/f implementation uses inline custom Nios instructions, while the MicroBlaze implementation uses inline put instructions for its FSL interface.

During execution, the front of queue is examined and dispatched to either the DMA or vector engine. If the dispatched instruction depends upon the result of an earlier instruction, the dispatched instruction will stall until the data hazard is cleared. Structural hazards can also cause a dispatched instruction to stall.

As mentioned earlier, DMA operations will transfer blocks of data between the scratchpad and an external memory-mapped region, which is usually external DRAM but could be a large block of on-chip SRAM, for example. Likewise, 2D windowed transfers are possible where several blocks have a constant stride between each block.

The DMA has an advanced mode where it can also filter the data by passing it through a user-defined pipeline. This allows advanced users to write a small amount of VHDL code that does custom data manipulation, such as interpolation between data points or colorspace conversion. For system flexibility, regular and filtered DMA operations are sent to different Avalon master ports. As

now AVX, each promising higher performance. Each time, programs have to be re-written to take advantage of the new family of SIMD instructions.

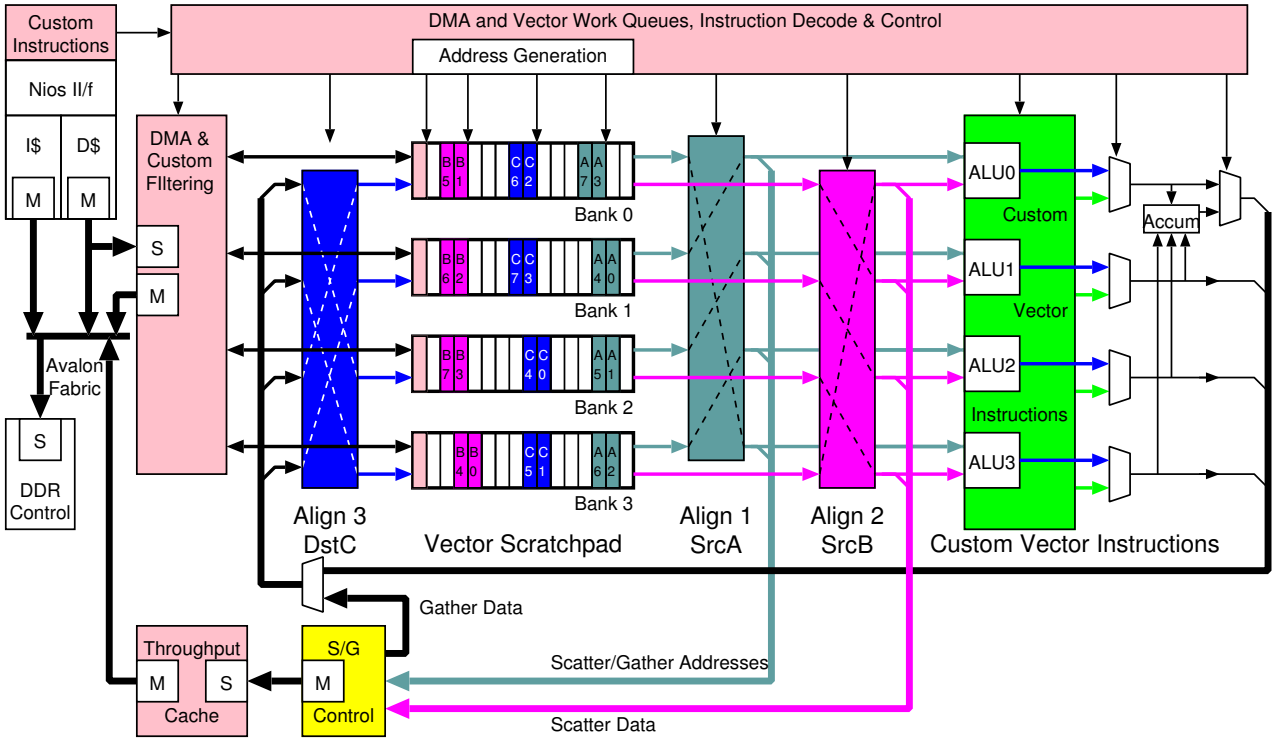

Figure 3. VectorBlox MXP detailed architecture (Altera version)

shown in Figure 1, this allows filtered DMA operations to be sent through a cache, for example.[2]

4.1 Vector Operations

For any vector operation, there are one or two source operands and one destination operand. Vector operands are specified by a pointer into the scratchpad; these pointers are passed along with the opcode to the vector engine to give the starting address of each vector. The pointers are manipulated only by the scalar host processor; copies of their values are used by the vector engine. However, the vector engine has several addressing modes that manipulate the pointer copies while executing a vector instruction.

Each of the source operands can also be specialized. The first operand can also be a scalar value in order to broadcast an immediate or scalar host register value to all vector lanes. The second operand can generate an enumerated value, which produces a vector where each element contains its positional value (0, 1, 2, etc) which is useful for indexing.

All operations can operate on either byte, halfword or word data sizes. Data size conversion can be done during the instruction — for example, two vectors of bytes can be multiplied to produce a vector of halfwords. All conversions, either up or down in size, are supported with all instructions.

4.2 Alignment Networks

An operation on a vector of length 8 is shown in Figure 3. Vectors A and B are the source operands, and vector C is the destination. Data in a vector is striped across the banks of the scratchpad, similar to a RAID array. This allows parallel access to multiple elements in the vector each clock cycle. In this case, 4 elements per cycle are accessed. However, notice that vectors A, B and C all have different alignments; their first element is located in a different bank. To correct this, three alignment networks exist in the pipeline. The first two ensure the first element of each source operand percolates to the top of the vector engine; the third ensures the destination is aligned correctly. Even when misaligned, the full width of the scratchpad can be used – this requires different addresses to reach each bank.

The first two alignment networks also perform data expansion, where the source vectors are increased in size from a byte to halfword, for example. The third alignment network performs data truncation, where a word result may be truncated to a halfword or byte.

4.3 ALU and Multiplier Operation

Aligned data are sent to the parallel ALUs for execution. The entire ALU pipeline has 4 stages, and always executes on each lane independently. When byte or halfword operands are used, each sub-lane executes independently.

One complex aspect is supporting integer multiply, fixed-point multiply, shifting and rotating for both signed and unsigned, and all data sizes of bytes, halfwords and words. To save on logic resoures, both shifts and rotates share the hardware multiplier with the multiply and fixed-point multiply instructions. For this reason, fixed-point multiply consist of a fixed shift amount determined at system build-time; this reduces it to just wiring rather than requiring an additional barrel shifter.

Even with this sharing hardware multipliers on the chip are a scarce resource. Large vector processors can quickly run out of multipliers, and embedded system designers may want to dedicate the multipliers for other FPGA-based logic (not the processor). To save on multiplier usage when necessary, or for code that is not multiply/shift/rotate limited, the MXP offers three implementation styles shown in Figure 4. Part (a) shows the full mode where four multipliers of different sizes are used; collectively, they can

[2] It doesn't make sense to cache regular DMA transactions, since they tend to be long streaming transfers.

C version:

```
for( i=0; i<VL; i++ ) {
  if( v_data[i] > 100 ) {
    v_data[i] = 100;
  }
}
```

MXP code:

```
vbx_set_vl( VL );
vbx( SVBU, VSUB,     v_flag,  100, v_data );
vbx( SVBU, VCMV_LTZ, v_data,  100, v_flag );
```

Figure 5. Conditional execution example

produce one word, two halfword, or four byte multiplication results each cycle. All of this fits in 1 Stratix IV DSP block, which contains two full 36×36 multipliers. Part (b) saves one-quarter of a DSP block by eliminating two 9×9 multipliers; for this to work, byte multiplications take two clock cycles each. Part (c) saves one-half of a DSP block by also eliminating the 18×18 multiplier, taking two cycles per halfword multiply and four per byte multiply.

Following ALU execution, the entire vector can be summed using a dedicated accumulate stage. This reduces the entire vector to a single element. The example FIR code in Table 1 uses the accumulator to sum the innermost loop. Hence, a dot product is possible in one instruction.

4.4 Conditional Execution

In addition to the ordinary data bits for each element, there is also a flag bit for every byte. This flag is set or cleared by DMA and ALU operations, and is used to store arithmetic condition codes and perform data-dependent conditional execution. For example, saturation, where a vector of data values are compared to a maximum value, and set equal the maximum. Additionally, overflow and carry/borrow conditions can be caught, and rounding can be performed after fixed-point multiply or right-shift operations.

An example showing the use of condition codes is given in Figure 5. This example saturates a vector of unsigned byte values to 100. The MXP version does this using two instructions, a subtract followed by a conditional move. The first instruction computes $100 - v\_data[i]$ and stores the 8 regular bits to $v\_flag[i]$, as well as the condition code for unsigned subtraction (a borrow) in the 9th bit position.[3] The second instruction performs a conditional move, which may assign the new value to $v\_data[i]$ if the predicate is true. The new value to be assigned is the scalar value 100. The last argument in the `vbx()` call is the location of the predicate vector itself. In this case, it is a vector of values, and the LTZ predicate inspects those 9-bit values to determine if the subtract result is less than zero. In this case, the result was less than zero only if the borrow bit is set, so the MXP will only inspect the value of the 9th flag bit from each element in the predicate vector.

4.5 Custom Vector Instructions

Advanced users of FPGAs will want to extract maximum performance from the hardware. These users will often be tempted to do *all* of the processing in custom logic using VHDL or Verilog. However, just like writing software, there is a large gap in effort between creating a compute kernel and developing a full application.

For these users, the MXP allows the user to define up to four custom instructions. By leveraging all of the MXP infrastructure, an advanced user can develop an entire application while developing only the compute kernel at the RTL level. Basic data processing can be done in C using the scalar host and MXP, and the performance-intensive kernel can then be implemented as a custom instruction.

[3] This is also an example of an instruction with one scalar operand.

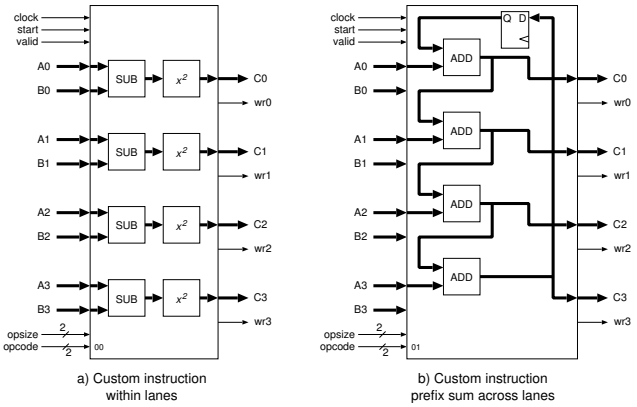

Figure 6. Custom vector instruction examples

Two examples of custom instructions are shown in Figure 6. In part (a), each lane produces a separate calculation of $(a-b)^2$. This simple example combines two instructions into one, but more complicated operations such as divide, square root, bit reversal, byte swap, population count, and find leading zero can be implemented for algorithms that need them and can save 10's of instructions.

In part (b), a more advanced operation is shown where a prefix sum is calculated; the output is a vector of sums, starting from the first element to the current element. This represents a much broader class of custom instructions, were values can be communicated between vector lanes. We have used this feature to implement an H.264 deblocking filter. It inspects up to 8 adjacent pixels and performs inter-pixel blending, resulting in up to 6 new pixel values. The filter is quite complex because the actual blending function applied depends upon precise pixel values. For this application, a single custom vector instruction replaces nearly 100 ordinary vector instructions, resulting in a significant speedup.

Using custom vector instructions, the MXP will be able to offer floating-point operations as well as the examples mentioned earlier. To assist the user, a VHDL template will be provided so only the processing engine needs to be implemented.

4.6 Scatter/Gather and Throughput Cache

Indexed memory operations are frequently needed by software. This is an addressing mode where an offset or index is added to a base address. For example, memory could hold an array of counters, where each counter represents a different type of event occuring. The index, or event number, represents which counter should be incremented. This is a *histogram* computational pattern.

In vector operations, the base address is usually the same, but every element in a vector requires access to an arbitrary index. This may be a permutation, where the indices are all unique, or it may involve collisions where an index value appears more than once. Performing an indexed write operation is called a *scatter*, while an indexed read operation is called a *gather*.

A vector of addresses is used for both scatter and gather operations. The gather operation can also be used to collect data spread across memory using any arbitrary values specified in the address vector. Likewise, the scatter operation can write values across memory at any arbitrary locations specified in the address vector.

Since gather and scatter operations will 'pick' and 'place' individual elements, only narrow memory transactions can be issued rather than wide and/or burst block transfers which make more effective use of external memory bandwidth. As a result, external DRAM quickly becomes a bottleneck when accessing random lo-

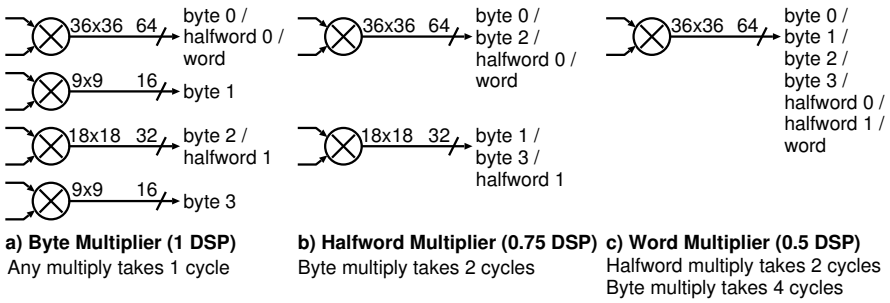

Figure 4. Fracturable multiplier (Altera version)

cations. To alleviate this, the MXP employes a custom *throughput-oriented* cache. The degree of associativity and size are both parameters set at system build time. The cache is very fast and is designed to match the speed of the underlying FPGA block RAMs; we have built a 256kB cache with 16 ways in a Cyclone IV at 210MHZ.

5. Data-Parallel Benchmarks

To evalute MXP, there are a variety of data-parallel benchmarks. We have chosen some benchmarks from DSP and media processing benchmarks that usually to operate on blocks of data. Some operations, like FIR, are very straight-forward to vectorize. However, some of them are not as simple and require more interesting approaches. This section discusses some of the more interesting benchmarks that have been vectorized.

To exemplify the operation of the scatter/gather and cache capabilities, motion compensation and histogram applications will also be discussed.

5.1 Exemplary Benchmarks

What type of benchmarks can be vectorized? Some applications are too simple and quickly become memory-bound rather than compute-bound. For example, taking the average between two images is memory-bound. In these cases, the MXP processor is actually an ideal way to perform the computation because custom-designed hardware cannot operate any faster than the memory bandwidth. However, we also want the MXP to run fast on applications that are not memory-bound.

The best speedup is obtained when a benchmark performs repeated operations upon a small data set that fits in the scratchpad. For large data sets, it is best if the data set can be divided into tiles. Operations on halfwords or bytes are preferred, as they benefit from double or quadruple the number of ALUs. Ideally, the same operation should be applied to every member of the data set, and data sets must be stored or arranged in contiguous memory locations; care should be taken at the outset to prearrange this, as significant performance can be lost just rearranging data.

Our best speedup to date is $918\times$ faster than a Nios II/f on integer matrix multiply using 64 vector lanes on large (4096 x 4096) matrices. Matrix multiply has good compute intensity: $O(N^3)$ operations applied to $O(N^2)$ data elements with only $O(N^2)$ data transfers. Furthermore, it can be easily tiled to fit in the scratchpad; double-buffering and concurrent DMA hides all memory latency even on systems with slow DRAM. Also, the same elementwise multiply is done to every vector element, and the built-in accumulator avoids the need to iterate over the data twice. Although increased performance using halfwords or bytes may be possible, we have not investigated that option. As a rough comparison, the MXP processor is $3.2\times$ faster than a tiled and loop-interchanged version of integer matrix multiply we wrote for an Intel i7-2600 processor (not using AVX or multiple cores).

Median filter and motion estimation are two other exemplary applications. Median filter works well because it can apply to small pixels (bytes), work with very long vectors (the image width), and has high compute intensity ($O(N^2 \times K^2)$ computation for an $N \times N$ image using a $K \times K$ window versus $O(N^2)$ data transfers). Motion estimation can also apply to bytes, work with fairly long vectors, and has similar compute intensity.

5.2 Motion Estimation

Motion estimation is an image encoding algorithm where a block to be compressed is compared against candidate positions in a reference frame. The position that has the lowest sum of absolute difference (SAD) of pixel values with the block is used as a reference for compression. Calculating these SAD values requires sweeping a 2D block across a 2D search window; it requires four nested loops to implement in scalar code:

$$SAD_{y,x} = \sum_r \sum_c |block_{r,c} - image_{y+r,x+c}|$$

The vectorized code in Figure 7 uses 2D and 3D vectors to offload the innermost loop bounds and address calculations, as well as taking advantage of the reduction accumulators to compute SAD values.

The innermost loop is done by a VABSDIFF vector instruction fed into the reduction accumulators to produce the SAD of one row of the image. The input data is 8-bit but is summed across multiple pixels, so to prevent overflow a 16-bit accumulated value is written out. The next level of loop is summing the SAD from each row together to produce a SAD for the entire block comparison. A 2D vector instruction can be used to run the row SAD calculation for each row, with a final (1D) vector operation to accumulate the row SADs into a block SAD. Finally, another level of loop can be moved into MXP by using a 3D vector instruction to compute block SADs across the width of the search space.

5.3 Motion Compensation

Motion compensation is the process of constructing a new or successive image in a video stream using an old reference image and a set of motion vectors. For each macroblock tile in the new image, starting at position <x,y>, it copies a tile of data from an arbitrary <mx,my> starting location in the old image. The precise starting location is determined by a set of motion vectors, which are delta-encoded from the original <x,y> values. These motion vectors are carefully chosen to represent the macroblock in the old image that closest matches the new image. Simplified pseudocode for this is given in Figure 8.

This function is complementary to motion estimation, but it is memory intensive rather than compute intensive. For the very small

```
for( y = 0; y < SEARCH_HEIGHT; y++ ) {

  // Set vector parameters to compute the SAD for each
  // row in a block across the search space width
  vbx_set_vl( BLOCK_WIDTH );
  vbx_set_2D( BLOCK_HEIGHT,
              sizeof(uint16_t),
              (BLOCK_WIDTH+SEARCH_WIDTH)*sizeof(uint8),
              BLOCK_WIDTH*sizeof(uint8) );
  vbx_set_3D( SEARCH_WIDTH,
              BLOCK_HEIGHT*sizeof(uint16_t),
              sizeof(uint8_t),
              0 );

  // 3D vector operation with reduction accumulate
  vbx_acc_3D( VVBHU, VABSDIFF,
              v_row_sad,
              v_img+y*(BLOCK_WIDTH+SEARCH_WIDTH),
              v_block );

  //Set vector parameters for accumulating the final SAD
  vbx_set_vl( BLOCK_HEIGHT/2 );
  vbx_set_2D( SEARCH_WIDTH,
              sizeof(uint8_t),
              BLOCK_HEIGHT*sizeof(uint16_t),
              BLOCK_HEIGHT*sizeof(uint16_t) );

  //2D vector operation to produces final SAD values
  vbx_acc_2D( VVHWU, VADD,
              v_result+y*SEARCH_WIDTH,
              v_row_sad,
              v_row_sad+(BLOCK_HEIGHT/2) );

  //Transfer the line back to the host
  vbx_dma_to_host( result+y*SEARCH_WIDTH,
                   v_result+y*SEARCH_WIDTH,
                   SEARCH_WIDTH*sizeof(output_type) );
 }
```

Figure 7. Vectorized motion estimation

```
for( y=0; y<HEIGHT; y+= TILE ) {
  for( x=0; x<WIDTH; x+= TILE ) {
    my = y + motvec[x/TILE][y/TILE].dy;
    mx = x + motvec[x/TILE][y/TILE].dx;
    for( r=0; r<TILE; r++ ) {
      for( c=0; c<TILE; c++ ) {
        img_new[y+r][x+c] = img_old[my+r][mx+c];
      }
    }
  }
}
```

Figure 8. Simplified H.264 full-pixel motion compensation

4×4 tile sizes in H.264, the four pixels across fit into a single word, offering very limited spatial locality. Across the rows, there is no locality at all. Furthermore, if motion vectors are large, accesses can be essentially random.

To perform this calculation on MXP, one can use a gather instruction or a custom DMA filter. To implement H.264 motion compensation, we chose a custom DMA filter because H.264 requires pixel interpolation to be done using quarter-pixel motion vectors. Our present implementation, which is limited to full-pixel motion vectors without interpolations, performs as shown in Figure 9. Results are shown for various image sizes as well as with and without the Throughput Cache connected to the DMA Filter.

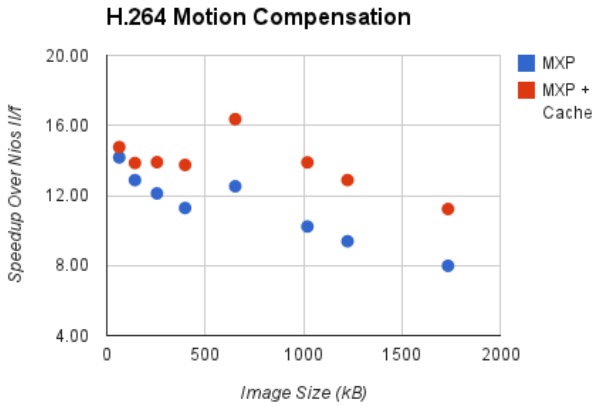

Figure 9. Speedup of H.264 full-pixel motion compensation

```
vbx_set_vl(NUM_VPS);

for(i = 0; i < NUM_KEYS; i+=NUM_VPS) {
  vbx_dma_to_vector(v_key, key+i,
                    NUM_VPS*sizeof(uint32_t));

  //Mask off index into bins (low 8 bits)
  vbx(SVWU, VAND, v_key, 0xFF, v_key);

  //Compute address of bin for given key and VP
  vbx(SVWU, VMUL, v_key, NUM_VPS,          v_key);
  vbx(VEWU, VADD, v_key, v_key,            UNUSED);
  vbx(SVWU, VMUL, v_key, sizeof(uint32_t), v_key);
  vbx(SVWU, VADD, v_key, bin_ptr,          v_key);

  //Increment the counter value
  vbx(VVWU, VGTHR, v_ctr,  v_key, UNUSED);
  vbx(SVWU, VADD,  v_ctr,  1,     v_ctr);
  vbx(VVWU, VSCTR, UNUSED, v_key, v_ctr);
}

//Accumulate the partial counts from each VP
sum_counters(bin_ptr);
```

Figure 10. Histogramming benchmark kernel

5.4 Histogram

A simple example of a scatter/gather benchmark is histogramming. A typical scalar version reads in an element, calculates the address of the counter it corresponds to, then loads, increments, and stores back the counter:

```
for(i = 0; i < NUM_KEYS; i++){
  counter[key[i] & 0xFF]++;
}
```

Simply rewriting this loop as a vector and using scatter/gather operations to perform the memory accesses leads to race conditions: it is possible that several parallel vector lanes may attempt to concurrently read-modify-write the same counter element. To work around this, a separate set of counters can be created, one for each virtual processor (VP). One virtual processor can be assigned to each element in the vector, allowing them to accumulate their data in separate counters. Then, after the main loop is completed, the counters for each VP must be summed to produced the final histogram.

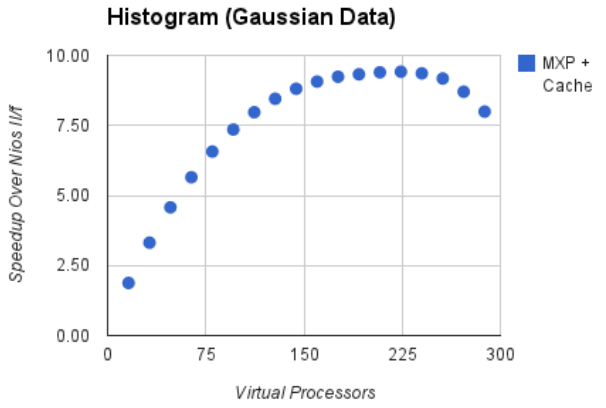

Figure 11. Speedup of histogram with Gaussian-distributed data

The vector code shown in Figure 10 performs a histogram of 32-bit input data into 256 VPs. The code is conceptually executed on multiple virtual processors (VPs); each VP then processes a fraction of the input data using its own set of counters. If more VPs are used than the number of actual vector lanes in an instance of MXP, the VPs are effectively time-multiplexed onto the vector execution units.

Speedup of histogramming is shown in Figure 11. Performance increases as longer vectors (more virtual processors) are used, until the capacity of the Throughput Cache begins to be compromised. In this result, the histogram data is randomly determined by a Gaussian distribution, which provides some temporal locality.

5.5 Difficult Benchmarks

Not all applications can be easily vectorized. For example, simply reversing the elements of a vector is not something that can be easily done using vector parallelism. Likewise, transposing a matrix cannot be done easily under ordinary circumstances.

Algorithms that perform different operations upon each vector element, particularly if it depends upon the value of the element or nearby elements, are also very difficult. One example of this is the H.264 deblocking filter, where the blending function that must be applied to pixels depends upon the pixel position, nearby pixel values, and run-time parameters. In this particular case, we were able to overcome this difficulty using a custom vector instruction.

Also, algorithms that perform sequential updates to data are difficult to parallelize. For example, summing all values from the first element to the current element is difficult. For this case, called a prefix sum problem, we have also implemented a custom vector instruction to overcome the sequential dependence. As another example, the H.264 deblocking filter has a built-in sequential dependence: macroblocks must be filtered in a predefined order, because the latest (new) pixel values for the previous macroblock are used as input values for the next macroblock. Wherever possible, algorithms that contain such a sequential dependence should be redesigned to expose more data-level parallelism.

Finally, algorithms that require values from adjacent vector lanes can be difficult to vectorize. Some cases are easy, for example computing the difference with a neighbour element can be done with a single instruction by using a pointer to a vector, as well as an incremented value of the same pointer. However, FFT-like communication patterns can be difficult to implement.[4]

[4] Large FFTs, for example 1024 points or larger, can be easily vectorized, but not small FFTs.

Device or CPU	VectorBlox MXP on Stratix IV 4GX530			
	ALMs	DSP 18×18	M9Ks	MHz
FPGA Device Maximum	212,480	1,024	1,280	–
Nios II/f (no vectors)	1,223	4	14	283
Nios II/f + V1 (4kB)	3,433	12	29	221
Nios II/f + V4 (16kB)	7,181	36	39	242
Nios II/f + V16 (64kB)	23,293	132	112	220
Nios II/f + V32 (128kB)	46,411	260	200	188
Nios II/f + V64 (256kB)	80,720	516	384	122

Table 2. Area resources and maximum clock frequency

Processor	MHz	Memory (kB)	3×3 Sobel (fps)
TI OMAP L138	456	64/256 (L1/L2)	11.0
MXP V1	100	32	19.7
MXP V2	100	32	34.2
MXP V4	100	32	54.0
MXP V8	100	32	76.0
MXP V16	100	32	94.9

Table 3. Performance of Sobel edge detection on 752x480 pixels

6. Benchmark Results

MXP has been implemented on a variety of FPGAs, including those from Altera (Cyclone II, IV and V, Stratix III and IV) and Xilinx (Spartan 6, Virtex 6, Artix 7, and Kintex 7). The results for mapping it to the largest available Stratix IV device are given in Table 2. When mapped to Cyclone, the clock speeds are lower and the logic count goes up because units are 4-input LUTs instead of 8-input ALMs. Also, Cyclone is based upon 18×18 multipliers versus the 36×36 DSP Blocks found in Stratix, but overall mapping efficiency is similar. The number of hard multipliers can be roughly cut in half using the control parameter described earlier.

To compare performance results to a typical embedded platform, Table 3 gives the performance of a 3×3 Sobel edge detection on a Texas Instruments OMAP L138 DSP and various sizes of MXP. The MXP is implemented in a low-end Cyclone IV device and limited to 100MHz, although higher clock speeds are possible. In both cases, the image size is 752x480 pixels. Processing times include accurate (16-bit) RGBA-to-LUMA conversion as well as both horizontal and vertical gradients. Despite running at less than 1/4 of the clock speed, the smallest MXP with a single lane is 70% faster than the TI DSP; with 16 vector lanes it is $8.6\times$ faster. At 95 fps and 32 bits per pixel, the MXP version achieves roughly 275 MB/s throughput and nearly saturates the 400 MB/s DRAM. On a Stratix IV FPGA, with a faster memory subsystem, significantly higher rates are possible. This example shows that an external DSP is not always required when MXP can do the job just as well.

Figure 12 shows the speedup of MXP over Nios II/f on a larger suite of embedded data parallel benchmarks. Speedups range up to $918\times$ on a 64-lane MXP (V64) on matrix multiplication. The geometric mean of benchmark results ranges from $8.8\times$ for a V1 MXP to $116\times$ for a V64 MXP.

Matrix multiplication and motion estimation both perform particularly well as they can take advantage of 2D/3D vectors and the reduction accumulators. Motion estimation fails to scale past V16, however, due to limited vector length. Other benchmarks such as imgblend and filt3x3 fail to scale well to 64 lanes due to memory bandwidth saturation even with the 64-bit DDR2 memory controller running at twice the system frequency on our Terasic DE4 (Stratix IV) test board.

The benchmarks rgbyiq and rgbcmyk do not get as high a speedup on a V1 as other benchmarks due to packing and unpack-

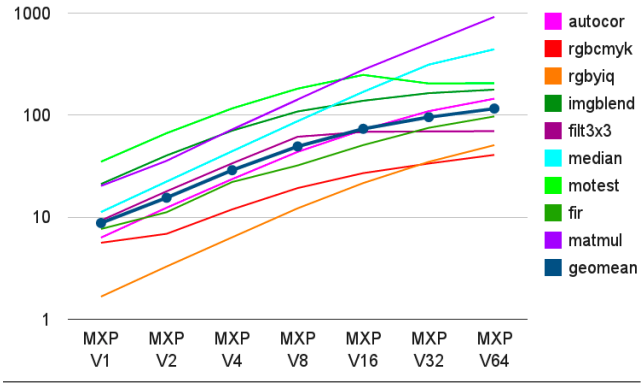

Figure 12. Cycle count speedup (y-axis) compared to Nios II/f

ing RGB triplets. This is a special case of the problem of dealing with data in array-of-structures (AoS) format on vector processors; it is better to arrange data as structures-of-arrays (SoA). For larger AoS data structures, the 2D DMA engine or scatter/gather units can be used to reorganize the data into the SoA format that is more readily handled by the vector processor. However, in this case, the overhead of fetching individual bytes is greater than that of packing/unpacking the RGB triplets in the scratchpad. In practice, this is not a severe limitation, because these types of colorspace conversions can be easily performed using a custom DMA filter, leaving all of the MXP ALUs available for other computation.

7. Related Work

VIRAM [2] demonstrated that vector archictectures could outperform VLIW and superscalar processors for embedded systems. VIRAM was an ASIC implementation, but it inspired the first generation of FPGA-based soft vector processors VIPERS [7] and VESPA [6]. These SVPs were traditional load/store vector architectures. The first scratchpad-based SVP was VEGAS [1] followed by VENICE [4]. VENICE added 2D/3D vector instructions (but only 1D DMA), as well as condition codes using the 9th bit of FPGA memory blocks.

MXP inherits some features from each of these SVPs, but adds several novel features, most notably custom vector instructions and custom DMA filtering support, 2D DMA, enumerated vector operands, and scatter/gather operations with a throughput oriented cache for memory access coalescing. MXP also adds interlocks to DMA operations to simplify the programming model, and includes fixed-point support.

Examinations of the impact of FPGA fabric on processor design [5] show that many techniques used to implement VLIW, wide superscalar, and out-of-order processors are prohibitively expensive on FPGAs. SVPs are the only purely software option shown to achieve significant speedups over the base RISC soft processors provided by FPGA vendors.

8. Conclusions and Future Work

The VectorBlox MXP is a drop-in addition to any FPGA-based computing system. It allows data-parallel processing to be done in an FPGA using a software development model. Algorithms can be programmed in C/C++ with vector intrinsics to accelerate data processing over 900× faster than the fastest currently-available FPGA-based soft processor.

MXP provides additional flexibility over previous solutions by supporting scatter/gather instructions, custom vector instructions and custom DMA filters. Together this allows programmers to implement many applications entirely in software on the FPGA while being able to accelerate those applications that need even more performance by implementing only a small amount of RTL.

The performance of MXP is sufficient to displace off-chip DSPs, yielding a more streamlined embedded system with a smaller board design footprint. This can also lead to extrinsic benefits such as streamlined inventory management due to lower BoM counts.

For the simplicity of extending an existing design environment, the MXP currently relies upon integration with the scalar host processors provided by Altera and Xilinx. However, these RISC processors were not designed to be attached to a high-performance vector engine, and suffer many inefficiencies. For example, neither Nios II/f nor MicroBlaze support external cache invalidations, so all coherence needs to be manually maintained. Also, for small vector processors, significant resources could be shared between the scalar front-end host processor and the vector engine. Overall, significant gains in performance could be achieved by redesigning the scalar host front-end to better suit the vector backend. This would also allow for better integration of the MXP ISA and scalar host ISA.

Work on producing a high-level compiler is underway. This will ease some of the burden of explicit memory management, eg double-buffered DMA can be automated. However, we do not recommend producing a full autovectorizing compiler; such an approach has had limited success with GCC, and we note that a significant source of MXP speedups come from reorganizing data to better suite the processor. Typically, significant data organization transformations fall outside of the scope of most compilers.

References

[1] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. Lemieux. VEGAS: Soft vector processor with scratchpad memory. In *FPGA*, pages 15–24, 2011. ISBN 978-1-4503-0554-9. doi: 10.1145/1950413.1950420. URL `http://portal.acm.org/citation.cfm?id=1950420`.

[2] C. Kozyrakis and D. Patterson. Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 283–293, 2002. doi: 10.1109/MICRO.2002.1176257.

[3] R. M. Russel. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[4] A. Severance and G. Lemieux. Venice: A compact vector processor for FPGA applications. In *International Conference on Field-Programmable Technology (FPT)*, pages 261–268, Dec.

[5] 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. ISBN 978-1-4503-0554-9. doi: 10.1145/1950413.1950419. URL `http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/1950413.1950419`.

[6] P. Yiannacouras, J. G. Steffan, and J. Rose. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *CASES*, pages 61–70. ACM, 2008. ISBN 978-1-60558-469-0. doi: 10.1145/1450095.1450107. URL `http://portal.acm.org/citation.cfm?id=1450107`.

[7] 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. doi: 10.1145/1534916.1534922. URL `http://portal.acm.org/citation.cfm?id=1534916.1534922`.