Soft Vector Processors with Streaming Pipelines

A. Severance †, ‡ aaronsev@ece.ubc.ca
J. Edwards † jedwards@vectorblox.com
H. Omidian ‡ hosseino@ece.ubc.ca
G. Lemieux †, ‡ lemieux@ece.ubc.ca

† VectorBlox Computing Inc.
Vancouver, BC

‡ Univ. of British Columbia
Vancouver, BC

ABSTRACT

Soft vector processors (SVPs) achieve significant performance gains through the use of parallel ALUs. However, since ALUs are used in a time-multiplexed fashion, this does not exploit a key strength of FPGA performance: pipeline parallelism. This paper shows how streaming pipelines can be integrated into the datapath of a SVP to achieve dramatic speedups. The SVP plays an important role in supplying the pipeline with high-bandwidth input data and storing its results using on-chip memory. However, the SVP must also perform the housekeeping tasks necessary to keep the pipeline busy. In particular, it orchestrates data movement between on-chip memory and external DRAM, it pre- or post-processes the data using its own ALUs, and it controls the overall sequence of execution. Since the SVP is programmed in C, these tasks are easier to develop and debug than using a traditional HDL approach. Using the N-body problem as a case study, this paper illustrates how custom streaming pipelines are integrated into the SVP datapath and multiple techniques for generating them. Using a custom pipeline, we demonstrate speedups over 7,000 times and performance-per-ALM over 100 times better than Nios II/f. The custom pipeline is also 50 times faster than a naive Intel Core i7 processor implementation.

1. INTRODUCTION

Although capable of high performance, FPGAs are also difficult to program. Exploiting both wide and deep custom pipelines typically requires a hardware designer to design a custom system in VHDL or Verilog. Recently, the emergence of ESL tools such as Vivado HLS and Altera's OpenCL compiler allow software programmers to produce FPGA designs using C or OpenCL. However, since all ESL tools translate a high-level algorithm into an HDL, they share common drawbacks: changes to the algorithm require lengthy FPGA recompiles, recompiling may run out of resources (eg, logic blocks) or fail to meet timing, debugging support is very limited, and high-level algorithmic features such as dynamic memory allocation and recursion are unavailable. Hence, ESL tools are not the most effective way to make FPGAs accessible to software programmers.

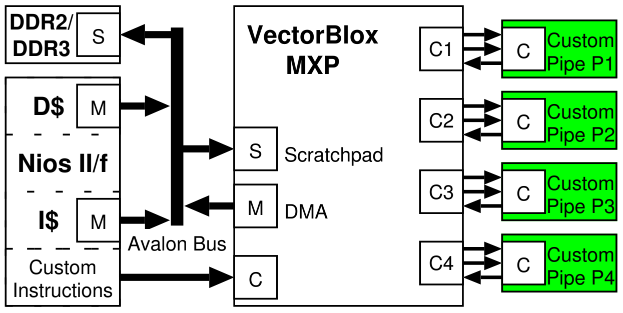


Figure 1: System view of VectorBlox MXP

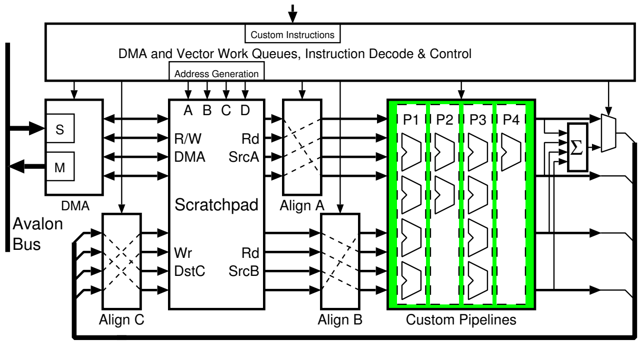


Figure 2: Internal view of VectorBlox MXP

Another approach to supporting programmers is to provide a soft vector processor (SVP). A SVP achieves high performance through wide data parallelism, efficient looping, and prefetching. The main advantages of a SVP are scalable performance and a traditional software programming model. Performance scaling is achieved by adding more ALUs, but beyond a certain point (eg, 64 ALUs) the increases in parallelism are eroded by clock frequency degradation. This ultimately limits performance scaling.

To increase performance of SVPs even further, they must also harness deep pipeline parallelism. For example, most types of encryption (such as AES) need deep pipelines to get significant speedup. Although processors (and SVPs) are not built to exploit deep pipeline parallelism, FPGAs support it very well.

The natural questions then become: How can a SVP be interfaced with deep pipelines to exploit both wide and deep parallelism? What kind of performance can be achieved? What types of problems arise, and how can they be solved?

To investigate these questions, we developed the VectorBlox MXP, shown in Figure 1 and devised a way to add deep custom pipelines to the processor, shown in Figure 2. The interface is kept as simple as possible so that software programmers can eventually develop these custom pipelines using C; we also show that a simple high-level synthesis tool can be created for this purpose. To demonstrate speedups, we selected the N-body gravity problem as case study. In this problem, each body exerts an attractive force on every other body, resulting in an $O(N^2)$ computation. The size and direction of the force between two bodies depends upon their two masses as well as the distance between them. Solving the problem requires square root and divide, neither of which are native operations to the MXP. Hence, we start by implementing simple custom instructions for the reciprocal and square root operations. Then, we implement the entire gravity equation as a deep pipeline.

It is important to note that we view the problem of adding floating-point units (FPUs) as a special case of adding deep, custom pipelines. FPUs are large in area, so many applications will not need one FPU for every integer ALU. In fact, many floating-point applications would find a single f-p divide unit or f-p square root unit to be sufficient, along with several f-p adders and f-p multipliers. Also, FPUs are deeply pipelined, so they need very long vectors to keep their pipelines utilized, especially when many units are instantiated in parallel. So, for both area and performance reasons, the programmer should control the number of integer units separately from the number of f-p adders, the number of f-p multipliers, etc. Hence, this paper is not just proposing a method for connecting specialized pipelines, but also for connecting general floating-point operators to SVPs.

The main contribution of this work is introducing a modular way of allowing users to add streaming pipelines into SVPs as *custom vector instructions* to get huge speedups. On the surface, this appears to be a simple extension of the way custom instructions are added to scalar CPUs such as Nios II. However, there are unique challenges to be able to stream data from multiple operands in a SVP. Also, scalar CPU custom instructions are often data starved, limiting their benefits. We show that SVPs can provide high-bandwidth data streaming to properly utilize custom instructions.

2. BACKGROUND

Vector processing has a long tradition in high performance computing, with designs originating in the 1960s. The canonical example (and the first commercially successful) is the Cray-1 [1]. The Cray-1 used a RISC-like load/store model, processing vector operands from a vector register file (VRF) of 8 named registers that were each 64-bits wide and 64 elements deep. Vector operations were streamed through the execution units at a rate of one per clock cycle.

2.1 FPGA-based Soft Vector Processors

The VIRAM [2] project demonstrated that vector processing could be more efficient than traditional processor architectures for embedded multimedia ASIC designs. Following this path, two FPGA-based projects implemented a VIRAM-like soft vector processor in FPGAs: VESPA [3] and VIPERS [4]. All three processors employed a hybrid vector-SIMD model, where vectors are streamed sequentially over time through replicated (parallel) execution units.

VESPA included support for heterogeneous vector lanes [5], e.g. there are fewer multipliers than general-purpose ALUs. Due to the mismatch between vector register file width and execution unit width, a parallel load queue was used to buffer a vector for heterogeneous operations, and a separate output queue was used to buffer results before writeback. This required additional memory and multiplexers. In contrast, this paper uses pre-existing alignment networks and requires no additional buffering to solve the width mismatch problem for 2-input/1-output custom instructions. We show how to add custom vector instructions that require more operands using minimal additional buffering.

VEGAS [6] and VENICE [7] are refinements of the VIPERS processor, further tailoring the architecture for FPGAs. Improvements include replacing the VRF with a scratchpad memory to allow for arbitrary data packing and access, removing vector length limits, enabling sub-word SIMD (four packed bytes or two packed shorts) within each lane, simplified conditionals and flags to fit within FPGA BRAMs, and a DMA-based memory interface rather than a traditional vector load/store approach.

Work by Cong et al. [8] created composable vector units. At compilation time, the DFG of a vector program was examined for clusters of operations that can be composed together to create a new streaming instruction that uses multiple operators and operands. This was done by chaining together existing functional units using an interconnection network and multi-ported register file. This is similar to traditional vector chaining, but it was resolved statically by the compiler (not dynamically the architecture) and encoded into the instruction stream. This provided pipeline parallelism, but was limited by the number of available operators and available register file ports. It is not easily extended to support wide SIMD-style parallelism. The reported speedups were less than a factor of two.

The FPVC, or floating point vector coprocessor, was developed by Kathiara and Leeser [9]. It adds a floating-point vector unit to the hard Xilinx PowerPC cores which can exploit SIMD parallelism as well as pipeline parallelism. The FPVC fetches its own VIRAM-like instructions and has its own private register file. Unlike most other vector architectures, it can also execute its own scalar operations separate from the host PowerPC.

Convey's HC-2 [10] is a vector computer built using several FPGAs. The FPGAs can adopt one of several 'personalities', each of which provides a domain-specific vector instruction set. User-developed personalities are also possible. Designed for high-performance computing, the machine includes a high bandwidth, highly interleaved, multi-bank DRAM array to reduce strided access latency.

2.2 VectorBlox MXP

The VectorBlox MXP [11], or MXP for short, is a new SVP that somewhat resembles VENICE [7]. It is designed for embedded systems that use a simple memory system. Figure 1 gives a high-level system view of MXP on Altera FPGAs; there is also a Xilinx version.

The host processor, Nios II/f, runs C code compiled with Altera's gcc. MXP instructions are inserted as Nios *custom instructions* using inline C functions. There are two types of

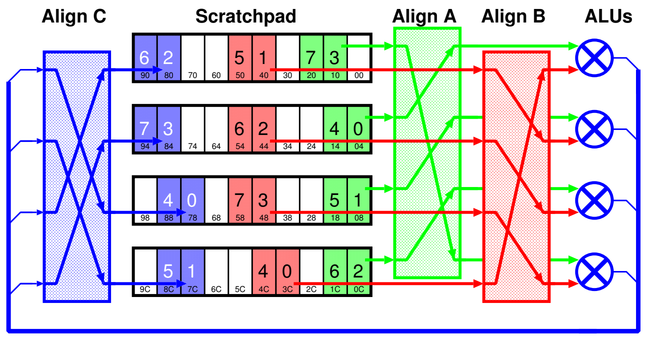


Figure 3: Alignment of data operands during vector instruction execution

MXP instructions: DMA operations and vector operations. Nios, DMA and vector operations all run concurrently, providing 3-way parallelism. Hardware interlocks resolve dependencies between the DMA and vector engines without software intervention.

In addition, the user can add custom operators or pipelines by connecting them in Altera's Qsys tool using a 'conduit' interface connection. These conduits are indicated in Figure 1 using the letter 'C', as opposed to Avalon masters (M) or slaves (S).

Figure 2 provides an internal view of the processor. The 2D DMA engine fetches data from the external Avalon system, typically from a DRAM controller. This efficiently copies data from external DRAM to a scratchpad memory built using on-chip block RAM. The scratchpad is double-clocked to provide four access ports. The DMA engine uses one of these ports; the vector engine uses the other three to fetch operands and store results. Internally, the scratchpad is organized as a wide, byte-addressable multibanked memory to provide concurrent, high-bandwidth data access.

Source and destination vectors are specified by a pointer into the scratchpad. These vectors can be unaligned with respect to each other. Data alignment networks ensure that any starting address can be issued to any port. Unaligned vectors naturally occur with sliding window algorithms such as convolutions; at each iteration the starting location of the vector to be processed will advance by one element. For these cases MXP has three separate alignment networks as shown in Figure 3: networks A and B align the source operands to the ALUs, while network C aligns the write-back results. Data is read out from the scratchpad in waves, where one wave is a full-width set of data that requires just one clock cycle. Back-to-back waves form a data stream.

Networks A and B shift the data to ensure that the first vector element of operands A and B, respectively, appears at the 'top' position of the wave, regardless of which bank it is actually stored in. Successive data elements are also shifted, forming a contiguous fully packed wave. The waves stream through the ALUs, which are followed by an optional summation stage (for accumulation reductions), and finally into network C to align the writeback. Figure 3 shows an alignment example where the first wave of 4 elements are read out from two vectors, and written back to a third vector, where all three vectors have differing alignments. Execution of a vector instruction on all 8 elements fits into two waves and fills two back-to-back pipeline cycles. A following vector instruction can issue its own operands with completely different alignments and have its waves packed back-to-back with the preceeding instruction.

The regular integer ALUs for MXP are located between the front-end alignment networks A and B and the back-end alignment network C. These are not directly shown in Figure 2, but they should be assumed to coexist with the 'custom pipelines' stage. The regular ALUs have a 3-stage pipeline, while custom pipelines can be of arbitrary length and have an arbitrary number of internal operators. The figure shows four different custom pipelines in the processor.

The number of parallel lanes (scratchpad banks/execution units) is configurable at synthesis time, but software does not need to be rewritten for different configurations thanks to the vector paradigm. Vector operations may execute over multiple cycles depending on the vector length and number of parallel lanes. Multiple instructions may be in the pipeline at a time. When hardware detects a hazard, e.g. when a vector instruction attempts to read a value currently in the pipeline, pipeline bubbles are inserted until the values are written back to the scratchpad.

The MXP natively supports integer and fixed-point data types. Every instruction can operate on bytes, shorts, or 32-bit integers in either signed or unsigned mode. Operations include traditional ALU instructions, including multiply, rotate and shift instructions, plus a move and several conditional move instructions. Also, fixed-point multiply can be done with a fixed decimal position. Note that divide or modulo operations and floating-point data types are missing, as these require complex logic.

3. CUSTOM VECTOR INSTRUCTIONS

3.1 Minimal Core Instructions

The VectorBlox MXP was designed to have a minimal core instruction set. It is very important to keep this instruction set minimal with a SVP because the area required by an operation will be replicated in every vector lane, thus multiplying its cost. In MXP, multiplication/shift/rotate instructions are included as core instructions because they share use of the hard multipliers in the FPGA fabric. However, divide and modulo are not included as core instructions because they require more than 3 pipeline stages and more logic than all other operators combined.

In addition to the minimal core instruction set, there are a large number of simple, stateless 1- or 2-input, single-output operators with no state that can be created. This includes arithmetic (e.g. divide, modulo, reciprocal, square root, reciprocal square root), bit manipulation (e.g. population count, leading-zero count, bit reversal), and encryption acceleration (e.g. s-box lookup, byte swapping, finite field arithmetic). Some of these operators require very little logic, while others demand a significant amount of logic. However, supporting all such operators is prohibitively expensive in FPGAs. Also, it is hard to imagine a single application that makes use of almost all of these specialized instructions. Finally, even when they are used by an application, they may not appear very frequently in the dynamic instruction mix.

For this reason, we have excluded many operations that could potentially be useful, but we felt are too specialized. Instead, we have devised a way for users to add these operations to the processor using custom vector instructions. Also, we allow the user to decide how many of these opera-

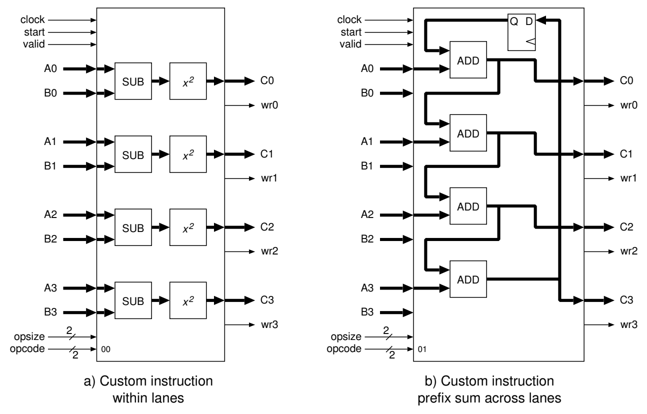


Figure 4: Examples of custom vector instructions

tors should be added to the pipeline, since it may not make sense to replicate large operators in every lane.

3.2 Custom Vector Instructions (CVIs)

For applications that require more complex operations, where software emulation is too difficult or slow, users can add their own application-specific *custom vector instructions*, also known as CVIs, into the pipeline.

Our add-on CVI approach is different from the VESPA approach [3], which allows selective instruction subsetting from a master instruction set. Subsetting allows very fine-grained control but it typically saves only a small amount of logic. Because no core set of instructions are defined, it also leads to derivative SVPs with incompatible instruction sets.

In contrast, the MXP approach defines a core set of instructions to increase software portability, while user-specified CVIs can be added to accelerate application-specific operations that are rarely needed by other applications. CVIs use an external conduit port interface to MXP, allowing the addition to be done without modifying the processor source HDL. This modularity will also make it easier to add CVIs using run-time reconfiguration. The user can create CVIs or take advantage of our current library of CVIs. This library includes count leading zeroes, compress, divide, square root, and prefix sum operations.

3.3 CVI Interface

A typical CVI is executed in MXP like a standard arithmetic instruction, with two source operands and one destination operand. The main difference is that data is sent out of MXP, through the conduit to the CVI and back again, then multiplexed back into the MXP pipeline before writeback. One individual CVI may consist of many parallel execution units, processing data in both a parallel and a streaming fashion.

In Altera's Qsys environment, CVIs are implemented as Avalon components with a specific conduit interface. Vector data and control signals are exported out of the top level of MXP and connected to the CVI automatically through the conduit. The conduit interface to a CVI designer can be seen in Figure 4. The left side (Figure 4a) shows an example of a simple CVI, the 'difference-squared' operation. This CVI does the same action on all data, so its individual execution units are simply replicated across the number of CVI lanes. In the simplest case, the number of CVI lanes will match the number of MXP lanes. This is adequate if the CVI is small, or there is plenty of area available. Later, we will consider the area-limited case when there must be fewer CVI lanes than MXP lanes.

The right side (Figure 4b) shows a more complicated example, a prefix sum, where data is communicated across lanes. The prefix sum calculates a new vector that stores the running total of the input vector appearing on operand A. Even if expressed as a tree, its complexity scales faster than $O(N)$. This makes it a very different type of operation than the difference-squared operation, which does not have communication between lanes. As a result, computing a prefix sum is a difficult operation for wide vector engines; it is best implemented in a streaming fashion. Since a vector may be longer than the width of the SVP, it is important to accumulate the value across multiple clock cycles in time. To support this, the CVI interface provides a clock and vector start signal. Furthermore, a data valid signal indicates when each wave of input data is provided, and individual data-enable input signals (not shown for clarity) are provided for each lane.

Additional signals in the CVI interface include an opsize (2 bits) that indicates byte/short/word data. Also, output byte-enable signals allow writing back only partial vector data, or to implement conditional-move operations, or when writing back a last (incomplete) wave. Finally, an opcode field is provided to allow the selection of multiple CVIs. Alternatively, the opcode can be passed to a single CVI and used as a mode-select for different functions, such as sharing logic for divide and modulo, or to implement different rounding modes. The opcode field is shown as two bits, but this can be easily extended.

We have found this interface capable of implementing a wide array of CVIs. We are considering extending this interface with a few additional control signals, such as signed/unsigned, scalar load, and pipeline status information, but we wish to keep the interface as simple as possible.

3.4 Heterogeneous Lane Support

Custom operators may be prohibitively large to add to each vector lane. For example, a fully pipelined Q16.16 fixed-point divider requires 2,652 ALMs to implement in a Stratix IV FPGA. This is more logic than an entire vector lane in the MXP. Thus, it can be desirable to use fewer dividers than the number of lanes (depending upon the portion of divides in the dynamic instruction mix). MXP supports using narrower CVIs with minimal overhead by reusing existing address generation and data alignment logic.

Figure 5 shows how CVIs with a different number of lanes are added to the existing MXP datapath. During normal operation, the address generation logic increments each address by the width of the vector processor each cycle. For the example shown, i.e. an MXP with four 32-bit lanes (written as a 'V4'), each source and destination address is incremented by 16 bytes, until the entire vector is processed. As discussed in Section 2, the input alignment networks align the start of both source vectors to lane 0 before the data is processed by the ALUs. After execution, the destination alignment network is used to align the result to the correct bank in the scratchpad for writeback.

During a CVI, the address generation logic only increments the source and destination addresses by the number

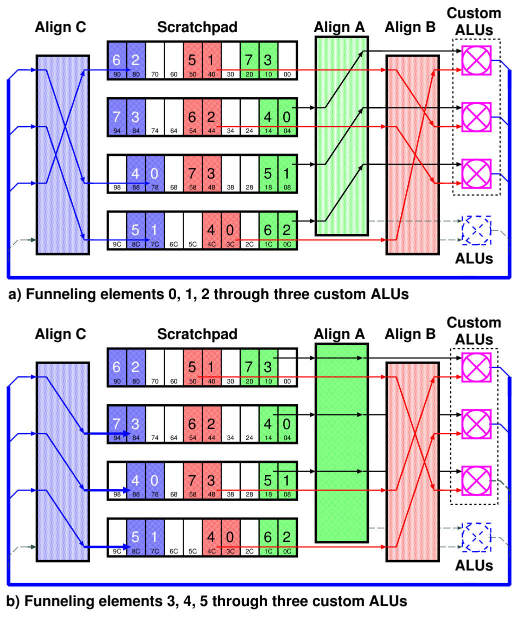

Align C
Scratchpad
Align A
Align B
Custom ALUs
ALUs
a) Funneling elements 0, 1, 2 through three custom ALUs
b) Funneling elements 3, 4, 5 through three custom ALUs

Figure 5: Custom vector instructions with fewer lanes than the SVP

of CVI lanes times the 4-byte width of each lane. In the example shown, the addresses are incremented by 12 bytes for each wave, regardless of the SVP width. As in normal execution, the alignment networks still align source data to start at lane 0 before data is processed in the custom ALUs. In this case, the fourth lane would not contain any data, so its data-enable input would be inactive. After execution, the CVI result is muxed back into the main MXP pipeline, and finally the resulting data is aligned for writeback into the scratchpad. On CVI writeback, the output byte enables are then used to write out data for only the first 12 bytes of each wave; the destination alignment network then repositions the wave to the correct target address.

3.5 CVIs with Deep Pipelines

MXP uses an in-order, stall-free backend for execution and writeback to achieve high frequencies. The CVIs are inserted in parallel to the regular 3-stage execution pipeline of MXP, which means they can also have 3 internal register stages. If fewer stages are needed, it must be padded to 3 stages. This is usually sufficient for combinations of small operators, bit twiddling, and reductions.

Some operations, such as divide or floating point, require much deeper pipelines. If the user naively creates a pipeline that is longer than 3 cycles, the first wave of data would appear to the writeback stage later than the writeback address, and the last wave of data would not reach the writeback stage at all.

To address the latter problem, we have devised a very simple strategy for inserting long pipelines. In software, we extend the vector length to account for the additional pipeline stages (minus the 3 normal stages). This solves part of the problem, allowing the last wave of vector data to get flushed out of the pipeline and appear at the writeback stage. During the last cycles, the pipeline will read data past the end of the input operands, but their results will never be written back. However, the beginning of the output vector will have garbage results.

To eliminate this waste of space, the MXP could simply delay the writeback address by the appropriate number of clock cycles. However, another way is to allow the CVI itself to specify its destination address for each wave. This requires the MXP to inform the CVI of the destination addresses, and rely upon the CVI to delay them appropriately. We have chosen this latter technique, as it allows for more complex operations where the write address needs to be controlled by the CVI, such as vector compression. Because this can write to arbitrary addresses, any CVI using this mode must set a flag which tells the SVP to flush its pipeline after the CVI has completed.

4. MULTI-OPERAND CVI

The CVIs described in the previous section are intended for 1 or 2 input operands, and 1 destination operand. This can be useful for some applications, but it is not very flexible. Certainly, the DAGs of most large compute kernels require multiple inputs and outputs, and require both scalar and vector operands. In this section, we describe how to to support multiple-input, multiple-output CVIs. As a motivating example, we have chosen the N-body gravitational problem. We modified the problem slightly to produce a pleasing visual demonstration: we restrict calculations to only 2 dimensions, we use a repelling force rather than an attracting force, and we allow collisions with the screen boundary.

4.1 N-Body Problem

The traditional N-body problem simulates a 3D universe, where each celestial object is a body, or particle, with a fixed mass. Over time, the velocity and position of each particle is updated according to interactions with other particles and the environment. In particular, each particle exerts a net force (i.e., gravity) on every other particle. The computational complexity of the basic all-pairs approach we use is $O(N^2)$. Although advanced methods exist to reduce this time complexity, we do not explore them here.

In our modified version, we consider a 2D screen rather than a 3D universe. The screen is easier to render than a 3D universe, but it also has boundaries. Also, we change the sign of gravity so that objects repel each other, rather than attract. (Attractive forces with screen boundaries would result in the eventual collapse into a moving black hole, which is not visually appealing.) Like the traditional N-body problem, we also treat particles as point masses, i.e. there are no collisions between particles. We have also adjusted the gravitational constant to produce visually pleasing results.

The run-time of the N-body is dominated by the gravity force calculation, shown below:

$$\vec{F_{i,j}} = G\frac{M_i M_j}{r^2} = 0.0625\frac{M_i M_j}{|\vec{P_i} - \vec{P_j}|^3}(\vec{P_i} - \vec{P_j})$$

where $\vec{F_{i,j}}$ is the force particle i imposes on particle j, $\vec{P_i}$ is the position of particle i, and M_i is the size or 'mass' of particle i. When computing these forces, we chose a fixed-point Q16.16 fixed-point representation, where the integer component of $\vec{P}$ represents a pixel location on the screen.

When a particle reaches the display boundary, its position and velocity are adjusted to reflect off the edge (towards the center) after removing some energy from the particle. These checks do not dominate the run-time as they are only $O(N)$.

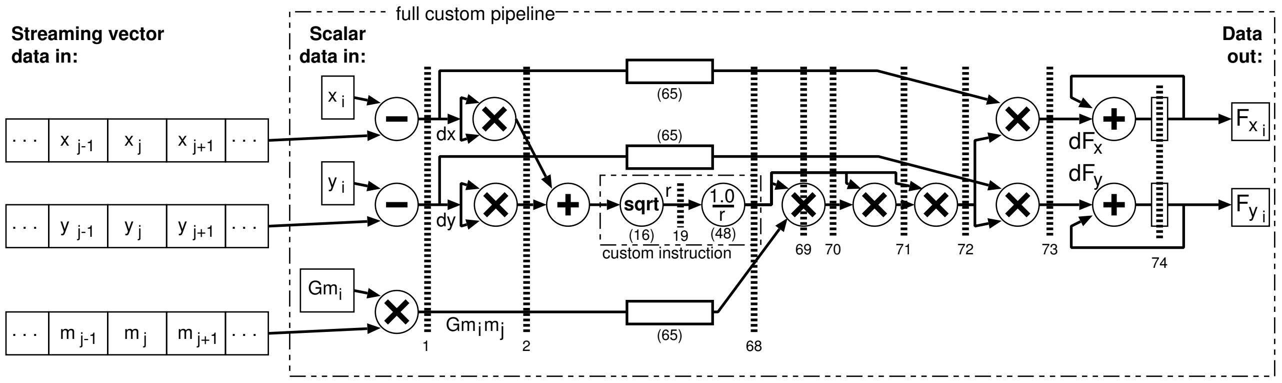


Figure 6: Force summation pipeline

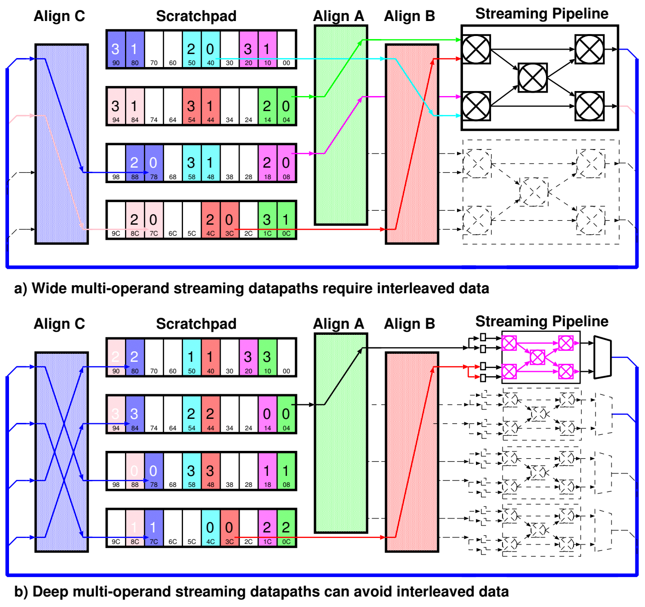

Align C
Scratchpad
Align A
Align B
Streaming Pipeline
a) Wide multi-operand streaming datapaths require interleaved data
b) Deep multi-operand streaming datapaths can avoid interleaved data

Figure 7: Multi-operand custom vector instructions

An implementation of the gravity computation as a streaming pipeline is shown in Figure 6. This is a fixed-point pipeline with 74 stages; the depth is dominated by the fixed-point square root and division operators at 16 and 48 cycles, respectively.[1] For each particle, its x position, y position, and mass (premultiplied by the gravitational constant) is loaded into scalar data registers within the instruction. This is the reference particle. Then, three vectors representing the x position, y position and mass of all particles are streamed through the vector pipeline. The pipeline integrates the forces exerted by all these particles, and computes a net force on the reference particle.

Overall, the pipeline requires 3 scalar inputs (reference particle properties) and 3 vector inputs (all other particles). It also produces 2 vector outputs (an x vector and a y vector), although the output vectors are of length 1 because of the accumulators at the end of the pipeline. Hence, this gravity pipeline is a 3-input, 2-output CVI.

All of the MXP vector instructions, including the custom type, only have 2 inputs and 1 output. This is a limitation in the software API, where only 2 inputs and 1 output can be specified, as well as the hardware dispatch, where only two source vector addresses and one destination vector address can be issued. Hence, programming a CVI with an arbitrary number of inputs and outputs requires a different way of looking at things.

Loading of scalar data can be accomplished by using vector operations with length 1, and either using an opcode bit to select scalar loading versus vector execution, or by fixed ping-ponging between scalar loading and vector execution. We use the ping-pong approach to save opcodes.

Supporting multiple vector operands is not as simple, however, and will be discussed below.

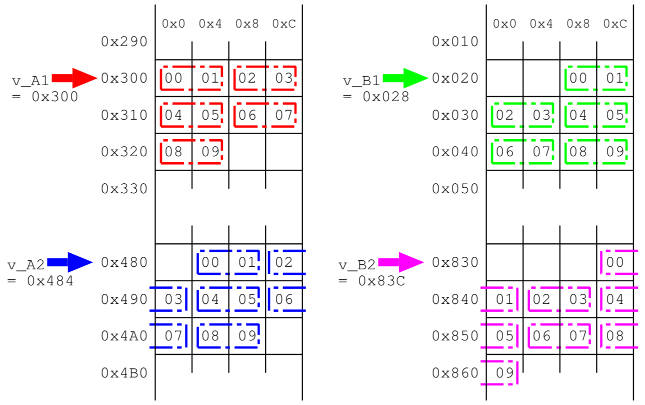
4.2 Multi-Operand CVI Dispatch

The wide approach requires data to be laid out spatially, such that operand A appears as vector element 0, operand B appears as vector element 1, and so forth. This is shown in Figure 7a. In other words, the operands are laid out consecutively in memory as if packed into a C structure. To stream these operands as vectors, an array-of-structs (AoS) is created. Ideally, the input operands would precisely fit into the first wave; with two read ports, the amount of input data would be twice the vector engine width. If more input data is required, then multiple waves will be required, which will be similar to the depth approach below. If less input data is required, then the CVI does not need to span the entire width of the SVP. In this case, it may be possible to provide multiple copies of the pipeline to add SIMD-level parallelism to the CVI.

The main drawback of the wide approach is that the data must be interleaved into an AoS. In our experience, SVPs work better when data is arranged into a struct-of-arrays (SoA). The SoA layout assures that each data item is in its own array, so SVP instructions can operate on contiguouly packed vectors.

For example, suppose image data is interleaved into an AoS as {r,g,b} triplets. With this organization, it is difficult to convert the data to {y,u,v} triplets because each output data item requires a different equation. When the image data is blocked as in a SoA, it is easy to compute the {y}

[1]We used Altera's LPM primitives for these operators. The pipeline would benefit from a combined reciprocal square root operator, but it does not exist in the Altera library.



```
VL = 2 (number of elements to keep together)
num1 = (number of arrays to interleave)
     = 2
num2 = (number of elements/VL)
     = 10/VL = 5

srcAStride1 = (v_A2 - v_A1)
            = 0x184
srcAStride2 = (v_A1[VL] - v_A1[0])
            = 0x08

srcBStride1 = (v_B2 - v_B1)
            = 0x814
srcBStride2 = (v_B1[VL] - v_B1[0])
            = 0x08

vbx_set_vl( VL );
vbx_set_2D( num1, dstStride1, srcAStride1, srcBStride1 );
vbx_set_3D( num2, dstStride2, srcAStride2, srcBStride2 );
vbx_3D( VVW, VCUSTOM1, v_dest, v_A1, v_B1 );
```

Interleaved read-out order:

Cycle	Operand A			Operand B		
1	A1	00	01	B1	00	01
2	A2	00	01	B2	00	01
3	A1	02	03	B1	02	03
4	A2	02	03	B2	02	03

```
vbx_interleave_4_2( VVW, VCUSTOM1, num_elem, VL,
   v_D1, v_D2, v_A1, v_A2, v_B1, v_B2 );
```

```
vbx_interleave_4_2( int TYPE, int INSTR, int NE, int VL,
 int8 *v_D1, int8 *v_D2,
 int8 *v_A1, int8 *v_A2, int8 *v_B1, int8 *v_B2 )
{
  vbx_set_vl( VL );
  vbx_set_2D( 2, v_D2-v_D1, v_A2-v_A1, v_B2-v_B1 );
  vbx_set_3D( NE/VL, v_D1[VL]-v_D1[0],
              v_A1[VL]-v_A1[0], v_B1[VL]-v_B1[0] );
  vbx_3D( TYPE, VINSTR, v_D1, v_A1, v_B1 );
}
```

Figure 8: Using 3D vector operations for multi-operand dispatch

matrix based upon the {r}, {g}, and {b} matrices. Furthermore, converting between AoS and SoA on the fly requires data copying and can take a long time. Hence, it is better for regular SVP instructions to use SoA format.

An alternative depth approach to multiple-operand CVIs requires data to be interleaved in time. This is shown in Figure 7b, where a streaming datapath only has access to two physical ports, operands A and B of one vector lane. This can be combined with wide parallelism by replicating the deep pipeline. It is not desirable to simply fully read two input vectors and then read the third input, though, as the CVI would have to buffer the full length of the instruction. In MXP, vector lengths are limited only by the size of the scratchpad, so the buffering could be costly. Rather, it is desirable to only buffer a single cycle's worth of inputs.

We accomplish this in MXP by using its 2D and 3D instruction dispatch to issue a single wavefront of data from each input on alternating cycles. The 2D instructions work by first executing a normal (1D) vector instruction, then applying a different stride to each of the input addresses and output address and repeating this operation multiple times. The strides and repetitions can be set at runtime using a separate set_2D instruction. The 3D instructions are an extension of this, where 2D instructions are repeated using another set of strides.

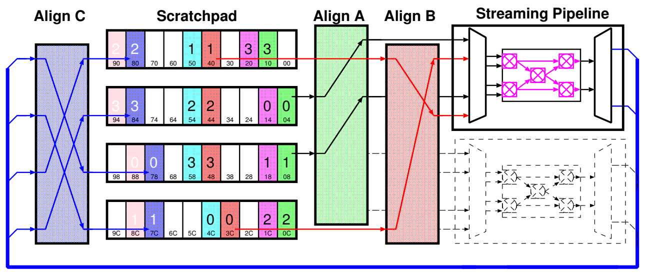


Figure 9: Multi-operand custom vector instructions with funnel adapters

Figure 8 illustrates how these 2D/3D ops are used to dispatch CVIs with multiple operands. In this example, a CVI with 4 inputs (A1, A2, B1, and B2) and 2 outputs (D1 and D2) is to be executed. The desired result is that the CVI will alternate A1/B1 and A2/B2 inputs each cycle, and alternate D1/D2 outputs each cycle.

To get this outcome, first the 1D vector length (VL) is set to the number of CVI lanes, and the 2D strides are set to the difference between input addresses (A2-A1, B2-B1) and output addresses (D2-D1). Since the inner vector length is the same as the number of custom instruction lanes, each row is dispatched as one wave in a single cycle, followed by a stride to the next input. The 2D vector length is set to the total number of cycles required (max(inputs/2, outputs/1)). Note that if more than 2 cycles (4 inputs or 2 outputs) are needed, sets of additional inputs and outputs will need to be laid out with a constant stride from each other.

Since a 2D operation merely alternates between sets of inputs (and outputs), a 3D instruction is used to stream through the arrays of data. Each 2D instruction processes one wavefront (of CVI lanes) worth of data, so the 3D instruction is set to stride by the number of CVI lanes. The number of these iterations (the 3D length) is set to the data length divided by the number of CVI lanes.

In Figure 8, the complex setup routine (top) can be abstracted away to a single function call, `vbx_interleave_4_2()` (middle). One possible implementation of this call is shown at the bottom of the figure.

On the hardware side, data is presented in wavefronts and needs to be multiplexed into a pipeline. Because a new set of inputs only arrives every max(inputs/2, outputs) cycles, the pipeline would be idle part of the time if it had the same width and clockrate as the CVI interface. We can recover the lost performance, and save area, by interleaving two or more logical streams into one physical pipeline. To do this, we have created 'funnel adapters' which are used to accept the spatially distributed wave and feed it to the pipeline over time. This is illustrated in Figure 9.

The funnel adapter for our 3-input, 2 output particle physics pipeline, which has inputs arriving every 2 cycles (and outputs leaving every 2 cycles), allows two MXP lanes worth of data to share a single physical streaming pipeline.

4.3 Face Detection CVI Example

As another example, we have also designed a multiple-input/output CVI for Viola-Jones face detection. The face detection pipeline is shown in Figure 10. Unlike the gravity

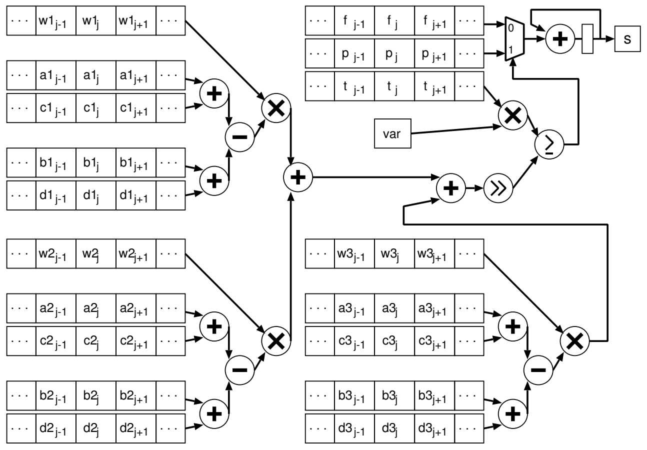


Figure 10: Face detection pipeline

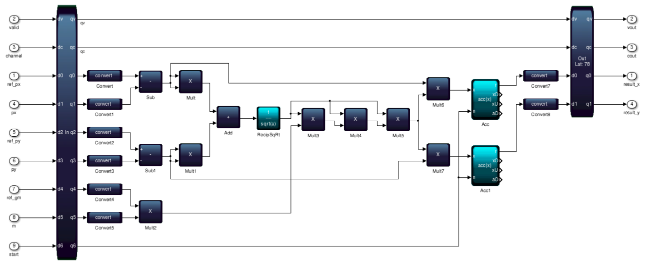


Figure 11: FLOAT Custom Vector Pipeline in Altera's DSP Builder

pipeline, the face detection requires far more inputs – a total of 18 vector inputs and 1 scalar input. It produces a single vector output.

Using regular SVP instructions, this face detection requires a total of 19 instructions, requiring 19 clock cycles per wave of data. In contrast, due to the large number of vector input operands, the face detection pipeline takes 9 clock cycles per wave of data. Hence, the best-case speedup expected from this custom pipeline is $\frac{19}{9}$, or roughly 2 times. Even though face detection contains a large number of operators, the number of input operands limits the overall speedup. Hence, not all applications will benefit significantly from custom pipelines.

5. CVI DESIGN METHODOLOGIES

While implementing a CVI to accelerate a SVP program is much easier than writing a complete accelerator, implementing them in HDL is not desirable for our target users, software programmers. Hence, we have explored two alternatives for generating CVI pipelines.

5.1 Altera's DSP Builder Pipelines

Altera's DSP Builder [12] (ADSPB) is a block-based toolset integrating into Matlab and Simulink to allow for push-button generation of RTL code. Figure 11 shows a floating-point version of our physics pipeline implemented in ADSPB. ADSPB was able to create the entire pipeline, including accumulation units, and design was significantly faster than manually building the fixed-point version in VHDL. Although we were not able to create a fixed-point version of

```
#define CVI_LANES 8 /* number of physical lanes */
typedef int32_t f16_t
f16_t ref_px, ref_py, ref_gm;
f16_t px[CVI_LANES], py[CVI_LANES], m[CVI_LANES];
f16_t result_x[CVI_LANES], result_y[CVI_LANES];

void force_calc()
{
  for( int glane = 0 ; glane < CVI_LANES ; glane++ ) {
    f16_t gmm = f16_mul( ref_gm, m[glane] );
    f16_t dx  = f16_sub( ref_px, px[glane] );
    f16_t dy  = f16_sub( ref_py, py[glane] );
    f16_t dx2 = f16_mul(dx,dx);
    f16_t dy2 = f16_mul(dy,dy);
    f16_t r2  = f16_add(dx2,dy2);
    f16_t r   = f16_sqrt(r2);
    f16_t rr  = f16_div(F16(1.0),r);
    f16_t gmm_rr    = f16_mul(rr,gmm_68);
    f16_t gmm_rr2   = f16_mul(rr,gmm_rr);
    f16_t gmm_rr3   = f16_mul(rr,gmm_rr2);
    f16_t dfx       = f16_mul(dx,gmm_rr3);
    f16_t dfy       = f16_mul(dy,gmm_rr3);
    f16_t result_x  = f16_add(result_x[glane],dfx);
    f16_t result_y  = f16_add(result_y[glane],dfy);
    result_x[glane] = result_x;
    result_y[glane] = result_y;
  }
}
```

Figure 12: Gravity pipeline C code for HLS (retiming registers omitted for clarity)

our pipeline in ADSPB because it lacks fixed-point reciprocal and square root, we were happy to generate a floating-point version as an additional data point of interest.

Some glue logic was needed to integrate the pipeline into a CVI, however, because our CVI pipelines require a clock enable signal, which ADSPB generated logic does not have. Rather than attempt to modify the generated code (including libraries used), we built a FIFO buffer to retime data appropriately, which adds minimal logic and uses one additional M9K memory per lane. This glue logic is sufficiently generic to allow any ADSPB-generated pipeline to be integrated into a CVI.

5.2 High Level Synthesis

Additionally, we have started to develop a High-level Synthesis (HLS) tool to implement CVIs in C. The goal is to show that only very simple HLS features are required to produce functional CVIs. For this reason, we started with bare LLVM rather than a more advanced HLS tool such as LegUp [13].

The user writes a function in C that produces the desired dataflow behaviour. This function has a standard API interface that matches the physical CVI interface shown in Figure 4. The input and output data are presented as global variables, and the user reads and writes to these variables to achieve the desired streaming behaviour. At the moment, our compiler recognizes specific global variable names, but this can be modified to use pragmas.

Like LegUp, our compiler converts C code into Verilog RTL output. However, since we are only attempting to generate dataflows, not complex sequential behaviour, only simple translation steps are required. For example, we unroll all loops as if they are for-generate hardware statements.

In our current implementation, the user manually inserts pipeline registers for retiming using a special function, `reg()`. For example, `a_3 = reg(a,3);` tells our LLVM compiler to

Table 1: Results with MXP compared to Nios II/f, Intel, and ARM Processors

Processor	Area (ALMs)	DSP 18-bit	F_{max} (MHz)	s/frame	GigaOp/s	pairs/s	Speedup
Nios II/f (fixed)	1,223	4	283	231.6	0.004	0.3M	1.0
Cortex A9 (zedboard) (fixed)	–	–	667	52.1	0.02	1.3M	4.5
Cortex A9 (zedboard) (float)	–	–	667	14.0	0.07	4.8M	16.6
Intel Core i7-2600 (fixed)	–	–	3400	6.5	0.15	10.3M	35.6
Intel Core i7-2600 (float)	–	–	3400	1.6	0.63	41.9M	144.8
MXP V32 (fixed)	46,250	132	193	73.8	0.14	9.1M	31.4
MXP V32+16FLOAT	115,142	644	122	0.041	24.6	1,326M	5,669
MXP V32+16FIXED	86,642	740	153	0.032	31.3	2,087M	7,203

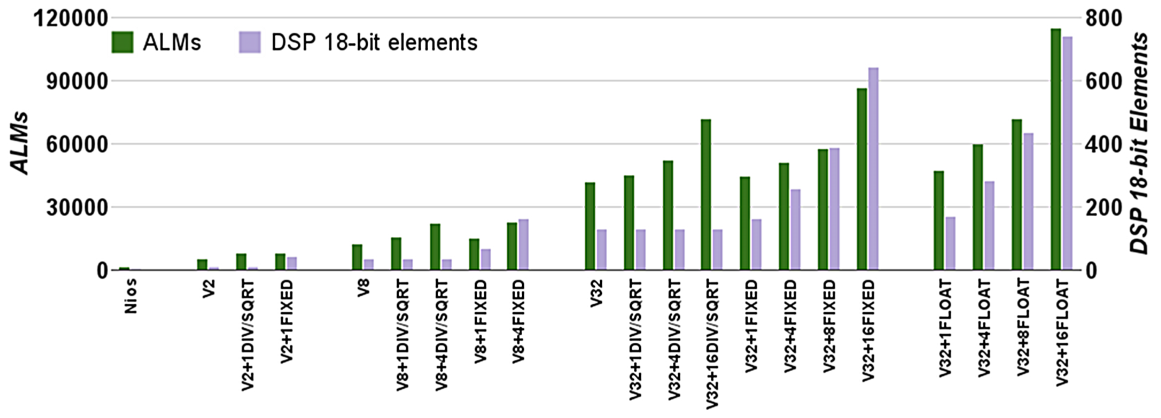


Figure 13: Area of gravity pipeline systems

create signal a_3 after adding 3 pipeline stages. Apart from this function, our C code is naturally readable by a software programmer. Later, we plan to add automatic retiming heuristics.

Although our compiler is limited in scope, we are able generate Verilog that is cycle-accurate with the fixed-point gravity pipeline described earlier in this paper. A portion of our C code using a Q16.16 fixed-point data type is shown in Figure 12. For clarity, we have removed the retiming registers and not shown the fixed-point function definitions.

6. RESULTS

All FPGA results are obtained using Quartus II 13.0 and a Terasic DE4 development board which has a Stratix IV GX530 FPGA and a 64-bit DDR2 interface. For comparison, Intel Core i7-2600 and ARM Cortex-A9 (from a Xilinx Zynq-based ZedBoard) performance results are shown. Both fixed-point (fixed) and floating-point (float) implementations were used. MXP natively supports fixed-point multiplication in all lanes. The Nios II/f contains an integer hardware multiplier and hardware divider; additional instructions are to operate on fixed-point data. The Intel, ARM and Nios II versions are written with the same C source using libfixmath [14]. We developed a vectorized version of this library for use with MXP. Nios II/f and MXP results use gcc-4.1.2 with '-O2'. The Core i7 results use gcc-4.6.3 and '-O2 -ftree-vectorize -m64 -march=corei7-avx'. ARM results use gcc-4.7.2 and reports the best runtime among '-O2' and '-O3'.

The MXP results vary the number of SVP lanes (V2, V8, and V32) and the number of CVI lanes. Three types of CVIs are generated: one containing separate fixed-point divide and square root instructions (DIV/SQRT), one containing a manually generated fixed-point gravity pipe (FIXED), and an ADSPB pipe (FLOAT). The LLVM pipeline results are omitted because they are nearly identical to (FIXED).

Figure 13 shows the area, in Adaptive Logic Modules (ALMs) on the left and DSP Block 18-bit elements on the right. The DIV/SQRT configurations take roughly the same area (in ALMs) as the FIXED pipeline. However, FIXED requires more multipliers. The FLOAT pipelines require about 5,500 ALMs and 38 DSP elements per lane versus 3,000 and 32 per lane for FIXED.

Running the N-body problem with 8,192 particles, Figure 14 shows the speedup relative to a Nios II/f soft processor for the various MXP configurations as well as a 3.4GHz Intel Core i7-2600 and a 667MHz ARM Cortex A9. The processor implementations are naive, but typical of what a C programmer might start with. A highly optimized single-core AVX implementation for i7-2600 matches our best MXP performance at 2×10^9 pairs per second [15]. However, that code was painstakingly hand-written in assembly language.[2] Overall, this is over 7,200 times faster than Nios II/f.

7. CONCLUSIONS AND FUTURE WORK

This work has demonstrated a way to reuse existing structures in SVPs to attach a variable number of streaming pipelines with minimal resource overhead. These can be accessed in software via *custom vector instructions* (CVIs). Logic-intensive operators, such as fixed-point divide, should not be simply replicated across all vector lanes. Doing so wastes FPGA area unnecessarily. Instead, it is important

[2]The AVX-optimized version also solves the 3D problem. Our 2D pipeline can be converted into a 3D version, with no expected loss in performance, with just 3 more additions and 2 more multiplications.

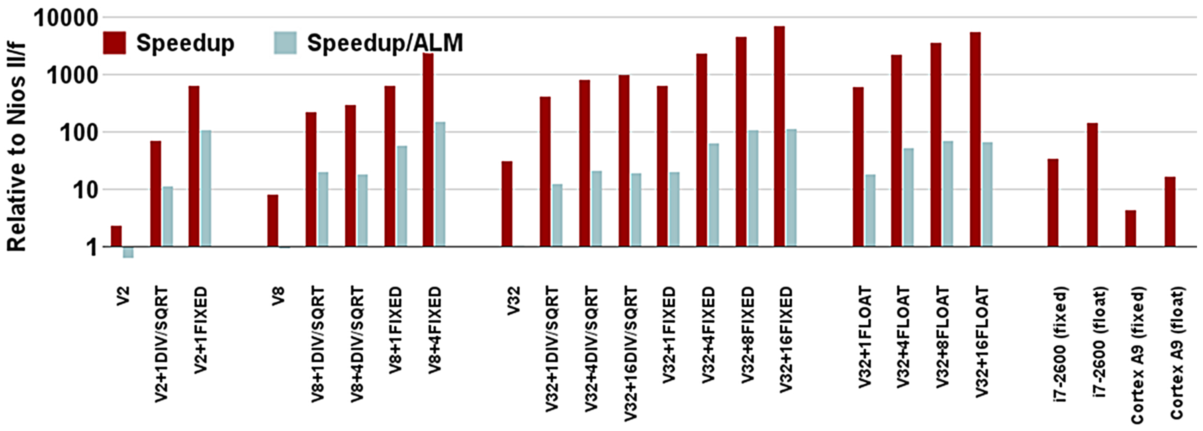


Figure 14: Performance and performance-per-area of gravity pipeline

to consider the frequency of use of the specialized pipeline, and add enough copies to get the most speed-up with minimal area overhead. Methods for dispatching complex CVIs were presented, including a time-interleaved method that allows an arbitrary number of inputs and outputs using funnel adapters.

The performance results achieve speedups far beyod what a plain SVP can accomplish. For example, a 32-lane SVP achieves a speedup of 31.4, whereas a CVI-optimized version is another 230 times faster, with a net speedup of 7,200 versus Nios II/f. This puts the MXP at par with an AVX-optimized Intel Core i7 implementation.

One area of future work is to make a repository of common operations and data types, such as min, max, power, and log for fixed-point, floating-point and even complex numbers. Also, we plan to continue exploring high-level synthesis options so that programmers can easily generate custom CVIs. Finally, allowing the programmer to dynamically reconfigure the CVIs will help when an FPGA must run several different applications.

As limitations of our work, some read and write bandwidth of the scratchpad is not utilized during the execution of our CVIs. Also, more advanced instruction dispatch strategies could be used to overlap execution of multiple CVIs. Alternatively, regular vector instructions could also overlap with a CVI.

8. ACKNOWLEDGMENTS

The authors would like to thank Martin Langhammer for his encouragement and advice on using Altera's ADSPB. We would also like to thank Altera for donating hardware and software licenses used in this research.

9. REFERENCES

[1] R. M. Russel, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[2] C. Kozyrakis and D. Patterson, "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks," in *Microarchitecture*, 2002, pp. 283–293.

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

[4] J. Yu, C. Eagleston, C. H. Chou, M. Perreault, and G. Lemieux, "Vector processing as a soft processor accelerator," *ACM TRETS*, vol. 2, no. 2, pp. 1–34, 2009.

[5] P. Yiannacouras, J. G. Steffan, and J. Rose, "Fine-grain performance scaling of soft vector processors," in *CASES*, 2009, pp. 97–106.

[6] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. Lemieux, "VEGAS: Soft vector processor with scratchpad memory," in *FPGA*, 2011, pp. 15–24.

[7] A. Severance and G. Lemieux, "VENICE: A compact vector processor for FPGA applications," in *FPT*, 2012, pp. 261–268.

[8] J. Cong, M. A. Ghodrat, M. Gill, H. Huang, B. Liu, R. Prabhakar, G. Reinman, and M. Vitanza, "Compilation and architecture support for customized vector instruction extension," in *ASP-DAC*, 2012.

[9] J. Kathiara and M. Leeser, "An autonomous vector/scalar floating point coprocessor for FPGAs," in *FCCM*, 2011, pp. 33–36.

[10] "The Convey HC-2 computer: Architectural overview," http://www.conveycomputer.com/files/4113/5394/7097/Convey_HC2_Architectual_Overview.pdf.

[11] "Embedded supercomputing in FPGAs with the vectorblox MXP matrix processor," in *ACM International Conference on Hardware/Software Codesign and System Synthesis*, Sept. 2013.

[12] "DSP builder," http://www.altera.com/technology/dsp/advanced-blockset/dsp-advanced-blockset.html.

[13] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *FPGA*, 2011, pp. 33–36.

[14] "libfixmath - cross platform fixed point maths library," http://code.google.com/p/libfixmath/.

[15] A. Tanikawa, K. Yoshikawa, K. Nitadori, and T. Okamoto, "Phantom-GRAPE: numerical software library to accelerate collisionless n-body simulation with SIMD instruction set on x86 architecture," *arXiv.org*, Oct. 2012.