Efficient Conditional Operations for Data-parallel Architectures

Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, Brucek Khailany
Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{ujk, billd, rixner, pmattson, jowens, khailany}@cva.stanford.edu

Abstract

Many data-parallel applications, including emerging media applications, have regular structures that can easily be expressed as a series of arithmetic kernels operating on data streams. Data-parallel architectures are designed to exploit this regularity by performing the same operation on many data elements concurrently. However, applications containing data-dependent control constructs perform poorly on these architectures. Conditional streams convert these constructs into data-dependent data movement. This allows data-parallel architectures to efficiently execute applications with data-dependent control flow. Essentially, conditional streams extend the range of applications that a data-parallel architecture can execute efficiently. For example, polygon rendering speeds up by a factor of 1.8 with the use of conditional streams.

1. Introduction

Many applications contain abundant data-parallelism, particularly emerging media applications such as graphics and video, image, and signal processing. Data-parallel architectures, such as vector [2][11][15][16], SIMD [3][12], and stream [9] processors, are well suited to extracting this data parallelism, achieving very high levels of performance. They utilize partitioned register files and reduced control overhead in order to support 10s to 100s of ALUs efficiently on a single chip [10]. The applicability of these architectures to current workloads is reflected by the recent emergence of SIMD ISA extensions, such as VIS [14] and MMX [8], as well as vector microprocessors [1][5][17]. However, strictly data-parallel machines are limited to executing applications that are largely free of data-dependent control constructs. Data-dependent control is typically handled by using select operations that leave computing resources idle or by using the memory system to reorder data. Either approach results in significant degradations of efficiency.

This paper introduces the concept of *conditional streams*, which extends the application range of data-parallel architectures by converting data-dependent control into data-dependent data routing. A data-parallel machine with conditional stream support more efficiently executes applications that would otherwise require data-dependent control. For instance, polygon rendering achieves a speedup of 1.8x when conditional streams are applied to it.

Many data-parallel applications have regular structures that can be represented as a series of computation *kernels* operating on data *streams* [9]. For example, consider the kernel in Figure 1 (pseudocode is provided in Figure 4), which converts pixels from the RGB color space to the YUV color space, as might be done as part of a video coding application [4]. This kernel consumes a single input stream of pixels in RGB, performs a set of identical calculations in parallel across a set of SIMD processing elements, and produces a single output stream of converted YUV pixels. In the *stream programming model*, applications are composed of a series of such kernels that consume data streams produced by previous kernels and generate data streams for use by subsequent kernels.

Figure 2(a) illustrates one type of data-dependent control, an *if* statement, that cannot be efficiently executed on a data-parallel machine. This code gets an input element, *x*, from the input stream (*in)* and applies one of two functions to it based on whether *x* is positive or negative. It then sends the result for negative inputs to one output stream (*a*), and the result for positive inputs to a different output stream (*b*).

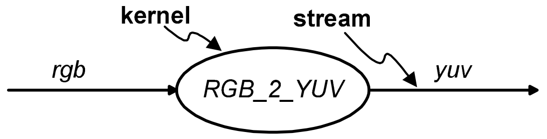
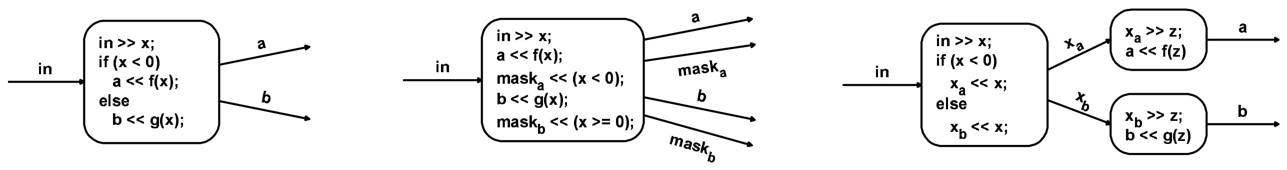


Figure 1: Kernel and streams example

The research described in this paper was supported by the Defense Advanced Research Projects Agency under ARPA order E254 and monitored by the Army Intelligence Center under contract DABT63-96-C-0037. Several of the authors are also supported by National Science Foundation and/or Stanford Graduate Fellowships.



(a) original code (b) SIMD without conditional streams (c) SIMD with conditional streams

Figure 2: ***If-then-else*** **statement on a SIMD machine**

On a data-parallel machine, where every processing element executes the same code, this example kernel would typically be implemented using mask streams as illustrated in Figure 2(b). For each input element, both outputs are calculated and associated mask streams are generated to indicate which elements of each output stream are valid. This approach leads to three significant inefficiencies. First, every processing element executes both *f* and *g* on every data value of *in*. Second, twice as much output data is generated than is necessary since there is an entry in both streams, *a* and *b,* for every element of *in*, and this will require more storage.[1] Finally, either the execution time of subsequent kernels that must scan over this unneeded data increases, or explicit memory or communication operations are required to compress the result streams so that all elements are valid.

Alternatively, a machine with conditional streams first separates the input stream to avoid computing unneeded results as illustrated in Figure 2(c). The first kernel performs a *conditional output* operation passing negative elements of the input stream (*in*) to one output stream (x_a) and positive elements of the input stream to another output stream (x_b). The kernel at the upper right of Figure 2(c) then applies function *f* to stream x_a, generating a dense output stream *a* in which every element is valid. Similarly, the lower right kernel applies *g* to the positive elements of *in* producing the dense output stream *b.* The conditional output operation allows the data-dependent control of Figure 2(a) to be converted to data-dependent routing. As a result, there is no wasted computation, the output streams contain only valid elements, and no subsequent memory or communication operations are needed to compact the resulting streams.

The remainder of this paper develops the concept of conditional streams in more detail. Section 2 lays the groundwork for conditional streams by describing data-parallel architectures and the stream programming model. Section 3 discusses conditional streams and presents several situations where they can be applied. Section 4 discusses the microarchitectural additions required to implement conditional stream functionality in hardware, as well as how to implement a software solution. The evaluation methodology used to gather results is described in Section 5, including a description of a processor implementing the conditional stream mechanism. The evaluation results and discussion are in Section 6. Finally, Section 7 describes related work.

1 The data in streams *a* and *b* could technically be stored in one stream, but the output of the following kernels would still require twice as much storage.

2. Background

Figure 3 shows a block diagram of a SIMD stream processor. The processor is organized into *n* partitions, each containing a processing element (PE) and a local memory. The local memories collectively form a stream register file (SRF) that is distributed across the partitions. Data streams are stored in this SRF as they are forwarded from one computation kernel to the next. All processing elements must be identical and may contain several functional units and local register files for temporary results. The *n* partitions are controlled by a single shared controller that broadcasts identical instructions to all of the processing elements and identical addresses to all of the local memories. This architecture is efficient for two main reasons: 1) the shared controller reduces control overhead; and 2) the partitioned organization exploits locality to provide high data bandwidth.

Applications targeted for a stream processor are written using the *stream programming model*, which structures applications as a series of kernels that operate on streams of data records. Each kernel consumes data records in sequence from its input streams and generates data records in sequence to its output streams. Kernels are restricted to accessing only their internal local variables, the head records of input streams, and the tail records of output streams. They cannot make arbitrary memory references.

Figure 4 shows the pseudocode for the simple kernel *RGB_2_YUV* presented in the Introduction and contains a graphical depiction of its execution on a stream processor with four SIMD partitions. The dotted lines across the figure show how data and processing are allocated across the partitions. On the first iteration of the loop, PE0 through PE3 access the input stream (*rgb*) and fetch RGB_0 through RGB_3 from their associated SRF banks. Each record access is atomic in that a PE must receive all the components of a particular record. The RGB pixels are then converted to YUV_0 through YUV_3 concurrently within the four PEs, and these computed values are sent to SRF0 through SRF3. Each subsequent loop iteration will process

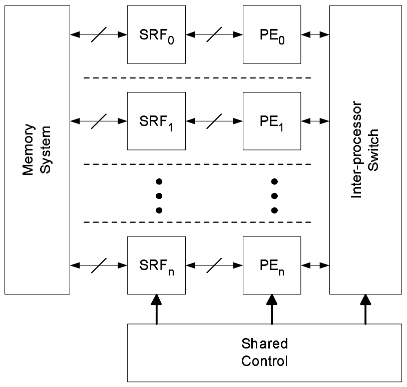


Figure 3: Stream Processor Architecture

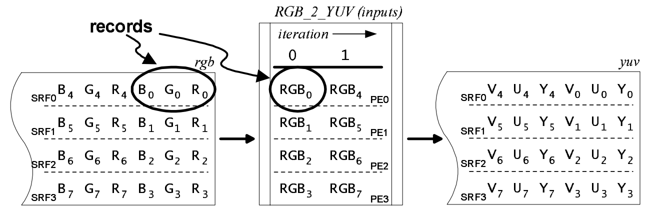
four pixels in the same manner. In practice, kernels can be much more complicated. They often contain considerable computation, interaction among stream elements, and multiple input streams and output streams that are consumed and produced at different rates. However, *RGB_2_YUV* is representative of most streaming kernels in that it loops over the data records of its input stream, performs some computation on each record, and produces successive values of its output stream. This code structure can be seen in Figure 4.

A SIMD processor efficiently executes perfectly data-parallel code, such as the *RGB_2_YUV* kernel, because every PE contributes useful computation every cycle. However, other computations require data-dependent control flow. Though, even if the data in each PE requires different types of processing, the PEs still must execute the same code since they share a single controller. Thus, data-dependent conditionals present a challenge. A simple example of such a computation, an *if statement*, is presented in the Introduction. In the following section, more complicated and realistic examples of data-dependent processing will be discussed. Furthermore, the concept of conditional streams will be introduced as a mechanism that can extend the efficiency and range of applications that can be executed on SIMD processors.

3. Conditional Streams

A *conditional stream* is a data stream that is accessed conditionally based on a case value local to a PE. Conditional access allows arbitrary stream expansion and stream compression in space (across hardware partitions) and time (across loop iterations). As will be shown, this property allows efficient execution of applications with data-dependent control on a SIMD architecture.

Figure 5 shows the data movement performed by a conditional input stream operating on a SIMD machine with four partitions. The dotted lines separate data and hardware associated with each partition. Each PE



```
record RGB { char r, g, b; };
record YUV { char y, u, v; };

kernel RGB_2_YUV( istream<RGB> rgb,
                  ostream<YUV> yuv )
{
   // loops until all data in rgb is read
   loop_until ( rgb.empty() ) {
      rgb >> in;     // Get next el. from rgb stream
      out.y = C1*in.r + C2*in.g + C3*in.b;
      out.u = C4*(in.b - out.y);
      out.v = C5*(in.r - out.y);
      yuv << out;    // Append out onto the yuv stream
   }
}
```

Figure 4: RGB_2_YUV kernel

independently decides whether to read a record from the conditional input stream each iteration based upon values from the *case* stream. On cycle 0, only PE0 and PE2 have a TRUE case value. Thus, the first two elements of the stream, the values *A* and *B*, are transferred to these PEs. The data value *B* is transferred from SRF1 to PE2, requiring communication across the partitions. (Section 4 discusses how the interprocessor switch is used to do this.) During the second iteration, PEs 1, 2, and 3 read from the input stream, receiving data values *C*, *D*, and *E* from SRFs 2, 3, and 0 respectively. Each PE that reads a value from the input stream receives the next value in sequence regardless of which SRF that value is located in. In effect, the stream is expanded in space (across hardware partitions) and in time (across loop iterations) according to the values in the case stream. In contrast, a conventional SIMD machine can only decide on each loop iteration whether or not all PEs should collectively read the next four values from the stream. On these machines, cross-partition communication requires cycling data through the memory system or coordinating communication through the interprocessor switch with software.

The programmer has access to conditional stream functionality via the simple primitive used in Line 6 of the code in Figure 5. This primitive can be used in a variety of modes to enable SIMD processors to efficiently execute applications with data-dependent control. All of the following modes can be classified as space-time expansions or compressions of data streams:

- *switch* — uses conditional output to compress
  In this mode, data is routed into one or more streams such that each stream will consist of homogeneous data. This guarantees that even though different control flow

```
1   kernel example( istream<bool> case,          // An input stream to this kernel
2                   cistream<int> input_stream )  // A conditional input stream to this kernel
3   {
4      loop_until ( case.empty() ) {
5         case >> sel;                // sel determines which PEs will access the stream
6         input_stream(sel) >> data;  // A PE receives an element of the stream only if sel
7      }                              //   is true in that PE
8   }
```

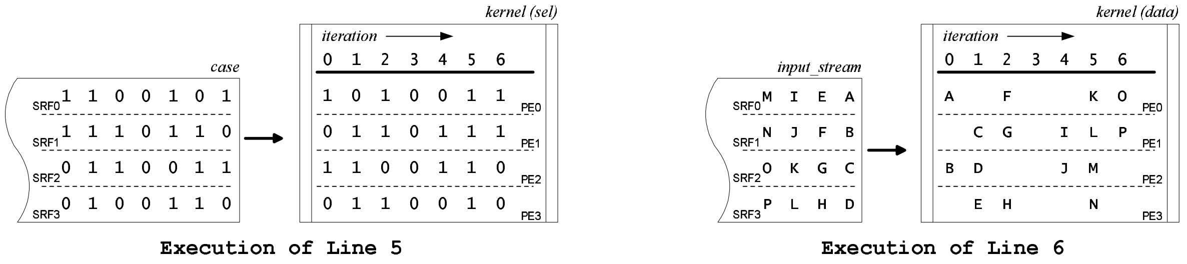


Figure 5: Conditional input stream execution with four PEs. Note that the convention adopted here is that the elements in a stream are ordered from top to bottom then right to left, and that time in a kernel goes from left to right.

and different computations may be required to process each of these resulting streams, every data element within a particular stream can be processed identically.

- *combine* — uses conditional input to expand
  Conditional input streams are used in this mode to combine two or more input streams into one output stream. The relative rates at which the input streams are processed are usually data-dependent.
- *load-balance* — uses conditional input to expand and conditional output to compress
  When the results of a computation require a variable (data-dependent) amount of time to generate, conditional streams can be used in this mode. A PE reads data from an input stream only when it is ready to start processing new data and writes to an output stream only when a valid result has been generated.

3.1. Conditional Switching

Figure 6(a) illustrates an example application snippet that reads a stream of values, filters out values that are greater than four, performs a non-trivial computation on the remaining values, and outputs the result. Without conditional streams, the kernel, *filter_process*, must produce a mask stream as shown in the figure. Each PE must perform the computation and write a result every loop iteration, even if it operates on a data element that is to be filtered out. A separate output stream, *mask*, indicates which elements of the *processed* stream are valid. When run on a SIMD machine, this code is inefficient in three ways. First, the function *compute* will be evaluated for all input elements, valid or invalid. Second, the invalid entries in the output stream will decrease the duty factor of subsequent kernels. Further filtering may exponentially decrease that duty factor until the stream is explicitly compressed through the main memory or inter-processor switch. Third, the final stream will occupy more space than necessary in the register file since it contains many unnecessary invalid values.

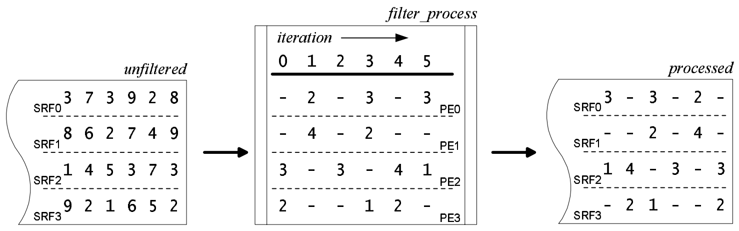
A SIMD processor with conditional streams performs the same function using two kernels as shown in Figure 6(b). When executing the first kernel, *filter*, each PE performs the test on its input element and conditionally outputs the element to an intermediate stream, *filtered*, which is now compressed, containing only valid data. In the second kernel, *process*, each PE reads a datum from the *filtered* stream, performs the computation, and appends the result to the output stream. There is no unnecessary computation as the PEs operate only on valid data, and there is no reduction in duty factor downstream because the output stream contains only valid data. In practice, the filtering operation can often be appended to the end of a previous computation kernel, eliminating the need for an additional kernel and its associated overhead.

Conditional switching is applicable to *case statement* types of data-dependent control where the computation and output performed for a data element is dependent on the value of the data element. For example, different computation may be performed depending on the state of finite elements or the type of geometry primitives. This application of conditional streams is especially useful when a rare case, such as an exception, requires a lot of processing. Normally, most PEs would idle while the exception case is processed. Conditional switching extracts only these exception cases to be dealt with independently. This works well if ordering is not important; otherwise, a separate mask stream can be generated and used to restore order at a later stage.

3.2. Conditional Combining

Figure 7 shows an example kernel, *interleave*, that produces an ordered stream (*out*) from two input streams

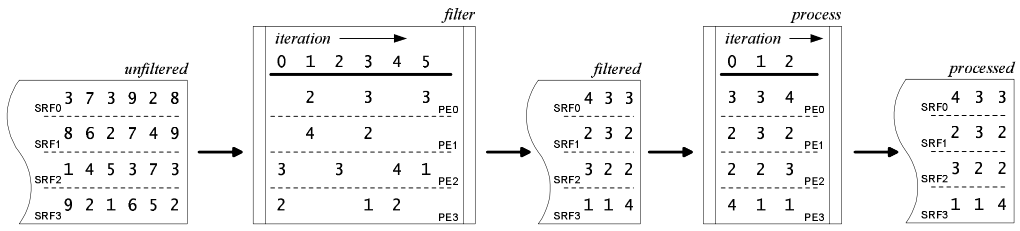
```
kernel filter_process( istream<int> unfiltered,
                       ostream<bool> mask,
                       ostream<int> processed )
{
   loop_until ( unfiltered.empty() ) {
      unfiltered >> curr;
      valid = (curr <= 4);
      mask << valid;
      processed << compute(curr);
   }
}
```



(a) without conditional streams

```
kernel filter( istream<int> unfiltered,
               costream<int> filtered  )
{
   loop_until ( unfiltered.empty() ) {
      unfiltered >> curr;
      valid = (curr <= 4);
      filtered(valid) << curr;
   }
}
```

```
kernel process( istream<int> filtered,
                ostream<int> processed )
{
   loop_until ( filtered.empty() ) {
      filtered >> curr;
      processed << compute(curr);
   }
}
```



(b) with conditional streams

Figure 6: Kernel pseudocode for the filter/process operation (*compute(x) = x*)

```
kernel interleave( istream<bool> case,
                   int addrA, int addrB,
                   ostream<unsigned int> loadIdx )
{
   loop_until ( case.empty() ) {
      case >> sel;

      // ACnt = # of PEs below you in which sel==1
      // BCnt =     "     "     "     "    sel==0
      // Note: PE_i is 'below' PE_j if (i < j)
      ACnt = numBelow(sel);  BCnt = MY_ID - ACnt;
      myAddr = sel ? (ACnt + addrA) : (BCnt + addrB);

      // numA calc. by broadcasting highest PE's val
      numA = broadcast(NUM_PE-1, ACnt + (sel ? 1 : 0) );
      addrA += numA; addrB += NUM_PE - numA;

      loadIdx << myAddr;
   }
}
```

(a) without conditional streams

```
kernel interleave(istream<bool> case,
                  cistream<int> inA,
                  cistream<int> inB,
                  ostream<int> out )
{
   // assume case.len == inA.len + inB.len
   loop_until ( case.empty() ) {
      case >> sel;
      inA(sel) >> a;
      inB(!sel) >> b;
      out << (sel ? a : b);
   }
}
```

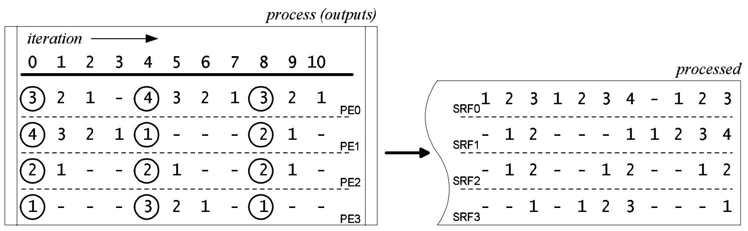
(b) with conditional streams

Figure 7: The *interleave* operation

and a third stream of case values. Each case value specifies from which input stream the next element of the output stream should originate: from *inA* if the case value is true, from *inB* otherwise. The code in Figure 7(a), which does not employ conditional streams, uses the values in the stream *case* to generate an index stream (*loadIdx*) that will be used to gather the elements of *inA* and *inB* from main memory. The index stream is generated by keeping explicit track of the running address of streams *inA* and *inB* in variables *addrA* and *addrB* respectively. After the index stream is complete, the output stream is generated by storing the *inA* and *inB* streams to memory and then performing an indexed load using the addresses in the index stream (not shown). As in this example, the PEs in a traditional SIMD processor cannot arbitrarily control the consumption rate of an input stream without a memory operation. Since the consumption rates of the two input streams are not known *a priori* for the *interleave* operation, the PEs can only control the expansion of the *inA* and *inB* streams indirectly via the indices in *loadIdx*.

The code in Figure 7(b), which employs conditional combining, eliminates the extra memory operations and the explicit operations required for the address calculations. Based on the case value *sel*, each PE simply requests a value from the appropriate stream and appends it to the output stream. Essentially, the conditional input stream correctly expands the data to the PEs so that the

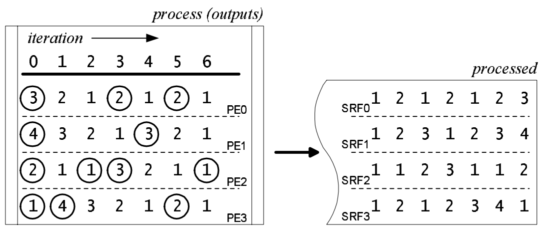
```
kernel process( istream<int> in,
                ostream<bool> mask
                ostream<int> processed )
{
   loop_until ( in.empty() ) {
      in >> curr;
      // loop ends when (curr <= 0) in all PEs
      loop_until (curr <= 0) {
         mask << (curr > 0);
         processed << curr--;
      }
   }
}
```



process (outputs)
iteration
0 1 2 3 4 5 6 7 8 9 10
3 2 1 - 4 3 2 1 3 2 1 PE0
4 3 2 1 1 - - - 2 1 - PE1
2 1 - - 2 1 - - 2 1 - PE2
1 - - - 3 2 1 - 1 - - PE3
processed
SRF0 1 2 3 1 2 3 4 - 1 2 3
SRF1 - 1 2 - - - 1 1 2 3 4
SRF2 - 1 2 - - 1 2 - - 1 2
SRF3 - - 1 - 1 2 3 - - - 1

(a) without conditional streams

```
// uses conditional input stream to load balance
kernel process( cistream<int> in,
                ostream<int> processed )
{
   in(TRUE) >> curr;
   loop_until ( in.empty() ) {
      processed << curr--;
      // curr only updated if (curr == 0)
      in(curr == 0) >> curr;
   }
   // process final elements, if necessary
   cleanup();
}
```



(b) with conditional streams

Figure 8: Pseudocode illustrating conditional streams used in the *load-balancing* mode

actual data can be interleaved while executing the kernel, obviating the need for extra memory transfers.

3.3. Conditional Load-Balancing

Load-imbalance often occurs on SIMD processors when PEs with short computations idle while one or more PEs with long computations perform additional iterations. Furthermore, the idle PEs may also generate NULL outputs during these idle cycles. Figure 8 illustrates how conditional streams can eliminate both the idle cycles and NULL outputs due to load-imbalance. For each input *curr* from the input stream, the kernels in the figure output the sequence of numbers {*curr, curr-1, ... , 1*}. The figure also shows the sequencing of the input and output streams during kernel execution.

Figure 8(a) illustrates an implementation without conditional streams. Two nested loops are used; data is read from the input stream by the outer loop, while the inner loop iterates until every PE completes processing its element. PEs with smaller values of *curr* finish earlier but are forced by the SIMD control to continue executing loop iterations and generating NULL outputs.

Figure 8(b) shows the same kernel using conditional input for load-balancing. This code only requires a single loop. On each iteration of the loop each PE generates an output value and reads a new element from the input stream only if it has completed processing the previous element, (i.e., if *curr* == 0). Thus, as soon as a PE finishes processing a data element, it requests and receives another one. The PEs perform neither idle iterations nor generate NULL outputs. PEs only remain idle when the input stream has been exhausted and while other PEs finish processing their final elements.

A concrete example is provided in the figure, where a circle around a datum indicates the cycle it was received by the PE. PE3 only needs 1 iteration to process the data element it initially receives. Since conditional stream access is not used in the code in Figure 8(a), PE3 must continue executing, outputting NULLs until the PE with the largest value of *curr* is finished three loop iterations later. The result is an output stream containing several NULLs. Figure 8(b) shows PE3 processing a new data element in the second iteration using a conditional input operation. As the variance of the processing times for the data elements increases, the savings provided by conditional load balancing improves. However, the order of the outputs produced with conditional load-balancing differs from that produced with the traditional implementation, neither of which are the same order as would result from a strictly serial implementation. In this example, if the order of the outputs was a concern, a sort would be performed on the output data (assuming additional ordering information was carried through the kernel). This issue, as it pertains to a polygon rendering pipeline, is discussed further in Section 6.3.

4. Microarchitecture

Implementing conditional streams requires both buffering and inter-partition communication of stream elements. The register file in the stream architecture presented in Section 3 operates under SIMD control; that is, each PE accesses the same location within its own

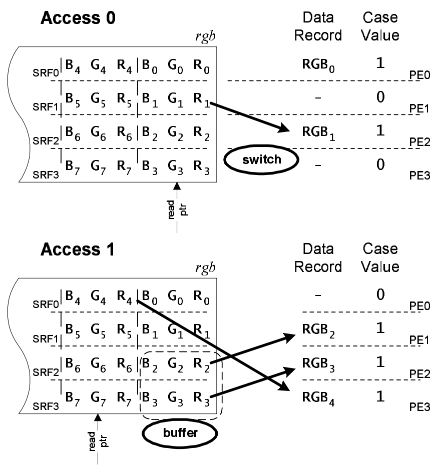


Figure 9: Sample conditional input access sequence

register file partition. Also, every PE can only read from its associated register file partition. Conditional streams require additional functionality, as shown in Figure 9. The figure shows two conditional stream accesses to an input stream of RGB data records and indicates the communication and buffering needed.

The figure shows that the data record RGB_1 is required by PE2 but is located in the register file partition associated with PE1. Therefore, the value must be communicated over the interprocessor switch. The second access in Figure 9 requires reading the first record of certain register file partitions (RGB_2 and RGB_3) and the second record in another partition (RGB_4). RGB_2 and RGB_3 must be buffered in order to provide the necessary data.

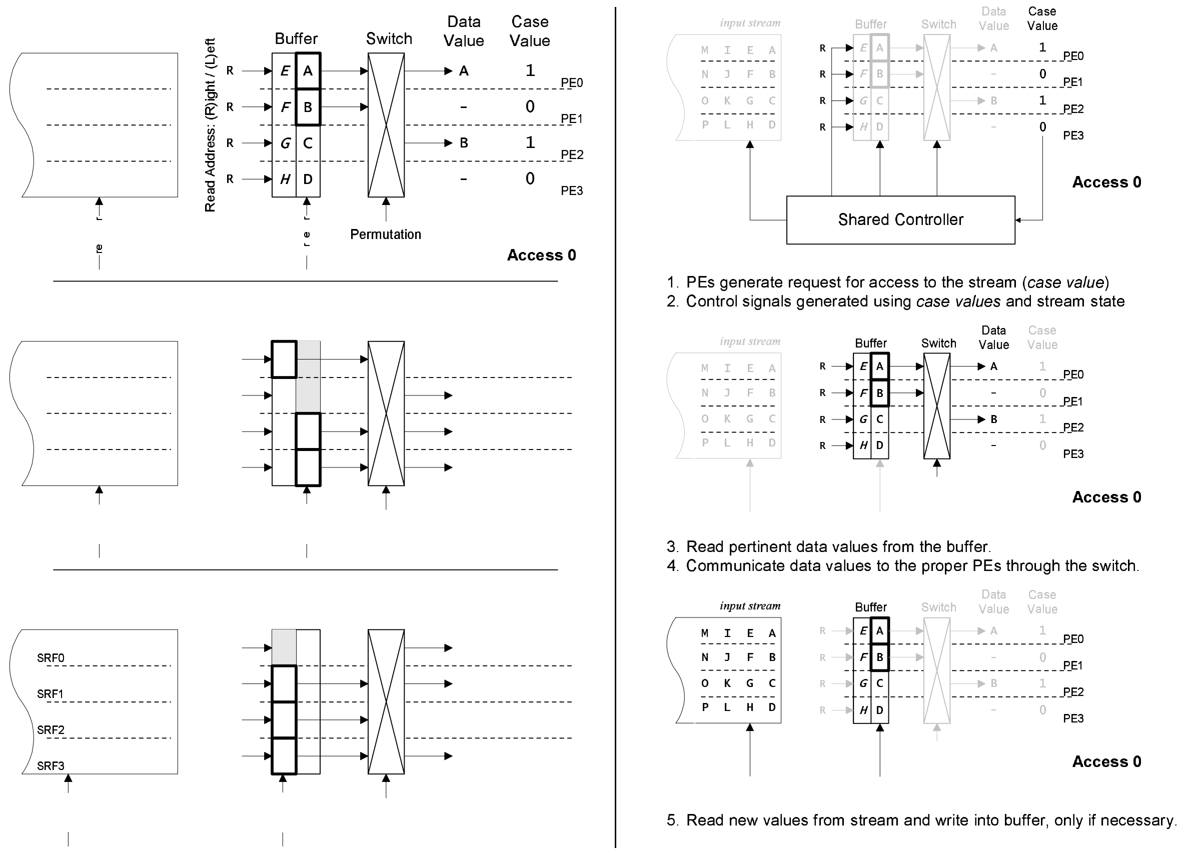
Figure 10 illustrates how a small buffer that is indexed using a local pointer in each PE in concert with an inter-processor switch can be used to implement conditional stream operations. Figure 10(a) illustrates the usage of the register file, buffer, and switch for a sequence of three conditional input operations.

Figure 10(b) shows the five steps required for the first access of Figure 10(a). First, the case values are examined to determine which PEs require input data. Second, control signals are generated for the switch, buffer, and register file by the shared controller (only shown in the top diagram in Figure 10(b)) to be used in the next three steps. Next, the appropriate buffer entry in each PE is accessed. For this first access, both values come from the right-hand side of the buffer. In step four, the data read from the buffer is communicated through the switch to the requesting PEs. Finally, if one side of the buffer has been completely emptied by the operation, as occurs after the second access, the empty side of the buffer is refilled by reading the next stream elements from the register file.

The controller determines whether or not to read new data values into the buffer, from which side of the buffer each PE should read, and the switch configuration. For example, the first access in Figure 10(a) only reads two values from right-hand side buffer entries, hence new values are not required. Then, when the second access reads three values and empties the right-hand side entries in the buffer, the controller causes four new values to be read from the input stream and written into these empty entries in the buffers. Since both data values for the first access are in right-hand side entries in the buffer, all read addresses are identical. However, the second access requires data values which reside in different sides of the buffer in the PEs. To account for this, the controller sets the read addresses of the buffer differently in each PE.

Dealing with output conditional streams, with the final values of input and output conditional streams, and with record lengths greater than one are all relatively straightfoward. Output streams are supported by sending data in the opposite direction. Data flows from the PEs through the switch into the buffer and eventually into the register file. At the end of an input stream, there may not be enough data in the buffers to satisfy all requests. An extra value, not shown in the figure, must be generated by the controller indicating to each PE whether or not valid data was received. An output conditional stream may not have received enough data to fill the final buffer entries. A user supplied NULL must be used to fill those empty entries if necessary. Finally, note that the accesses depicted in Figure 10 are for non-record data types. In order to keep transfers of records atomic, entries in the buffer in each PE are allocated for each record component, and steps 3-5 in Figure 10(b) must be iterated for each record component.

The implementation discussed so far requires a change only to the shared controller depicted in Figure 3. However, conditional streams can also be implemented without any dedicated control, assuming that a switch and a method for executing a hardware select (to perform the buffering) are available in each PE. In this case, the controller's functionality can be completely duplicated in software by storing the necessary conditional stream state, albeit redundantly in some cases, in every PE. The case values are broadcast over the switch, and each PE uses this information in conjunction with the stream state it has stored to determine which of the two buffer entries to read from. The switch permutation is calculated by the PEs and used to route the data from the buffer through the switch as before. Finally, since the access to the input stream for new data and the write into the buffer are either performed by every PE or by none at all, they can be enclosed in a branch. The code within the branch is only executed by every PE once all the entries in one of the two buffer sides have been emptied. This software approach is less efficient than the above hardware support, and will be explored further in Section 6.



1. PEs generate request for access to the stream (*case value*)
2. Control signals generated using *case values* and stream state

3. Read pertinent data values from the buffer.
4. Communicate data values to the proper PEs through the switch.

5. Read new values from stream and write into buffer, only if necessary.

(a) conditional accesses

(b) steps to execute one access

Figure 10: Buffer and switch usage and control for a conditional input stream. In (a), locations in the buffer that are being read are in bold outlined boxes, and values which have already been read are in grayed boxes. Italicized entries indicate values which were written into the buffer by the previous access (or that were just initialized).

5. Experimental Setup

To evaluate the performance advantages of conditional streams, a set of benchmarks was executed on a cycle-accurate simulator of the Imagine Stream Processor [9]. Imagine is similar in concept to the generic SIMD stream processor shown in Figure 3. Each processing element contains six 32-bit arithmetic units (three adders, two multipliers, and a divider) interconnected via a distributed register file structure, as shown in Figure 11. Each processing element has a communication unit which interfaces that processing element to the inter-PE switch. Imagine contains eight identical PEs controlled by a single microcontroller that broadcasts instructions. The only control flow operation supported by the microcontroller is a loop. Loop termination can be controlled by the value of a boolean in all of the PEs, by a counter, or by exhausting an input stream in the stream register file (SRF).

Data-dependent conditionals represent a programming challenge on typical SIMD machines that contain no control flow instructions other than loops. Imagine provides three mechanisms for dealing with data dependent conditionals: the select operation, a register offset addressable scratch-pad memory, and hardware conditional streams. The select operation can execute on any arithmetic unit in a single cycle. Based on a boolean value, the output of the select operation is set to the value of either its left or right input, similar to the C "?:" operator. Each PE in Imagine contains a 256 entry scratch-pad memory that is addressed by using a common base address, supplied by the microcontroller, added to an offset computed locally in each PE. This allows each PE to reference different locations inside its local scratch-pad, simplifying functions such as adaptive histogramming or matrix transposition. Conditional streams on Imagine are implemented as described in Section 4. The scratch-pad memories in the PEs are used as the buffer storage and the communication units provide the interface to the interprocessor switch in order to shuffle data records between hardware partitions for conditional stream accesses. When used for conditional streams, both of these units execute special instructions that supply the indices to the scratch-pads and the communication pattern directly from the conditional stream hardware. Every conditional

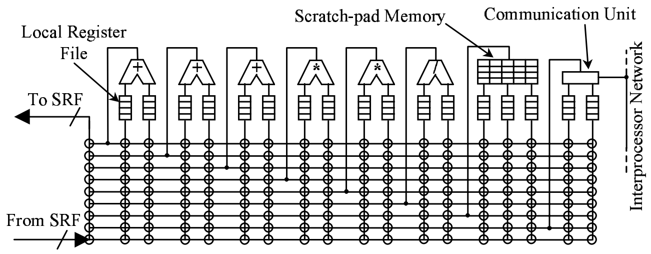


Figure 11: Processing element of Imagine

stream access requires a read from and a write to the scratch-pad memories, a communication, and a stream access. All four of these operations take one cycle each on Imagine. Additional cycles are required to generate the control signals from the case values; two cycles for conditional input streams and one cycle for conditional output streams. When implemented in software, 26 and 51 extra operations are required for each conditional input and output access respectively (for record lengths of one).

Each of the eight processing elements in Imagine has access to a 16KB bank of the stream register file. The eight banks of the SRF are connected to a streaming memory system that supplies a peak bandwidth of 4GB/s to off-chip double data rate SDRAM. Imagine is expected to operate at 500MHz, yielding a peak computation rate of 20GFLOPS and a peak SRF bandwidth of 32GB/s.

6. Results and Discussion

The performance of three configurations are evaluated: TRADITIONAL, a stream processor without conditional stream support; HARDWARE, a stream processor with hardware conditional stream support; and SOFTWARE, a stream processor with only software support for conditional stream accesses. These configurations are evaluated on two microbenchmarks based on the examples of Section 3. Two realistic applications are also considered: merge sort and polygon rendering.

6.1. Microbenchmarks

Figure 12 shows the results of running the *conditional switching* microbenchmark, which is largely based on the example illustrated in Figure 6. The graph shows the execution time as the number of arithmetic operations in the function *compute* increases from 10 to 100 for the three implementations. Since filtering would most likely occur at the end of another kernel in a real application, an additional 30 arithmetic operations are performed before the filtering in order to simulate a more realistic workload. This processing occurs in the kernel *filter_process* for TRADITIONAL and in *filter* for the other two. Results are shown for a 256 element data set where 0%, 50%, and 100% of the data elements are valid. TRADITIONAL executes for the maximum number of cycles regardless of how many elements are valid because it uses a mask stream. Hence the TRADITIONAL curves in all three graphs are identical. The 0% percent curves show maximum performance gains possible with conditional streams for this benchmark. Since the stream of compressed valid values in the conditional stream implementations will be empty for the 0% case, execution time is constant regardless of how long it takes to process valid values. Finally, the 100% curves show the overhead of each of the conditional stream methods. Most of the overhead in HARDWARE is due to splitting the computation into two kernels. SOFTWARE has additional overhead since operations are added to the inner loop of the *filter* kernel for emulating conditional stream accesses. Despite this overhead, SOFTWARE still offers better performance than TRADITIONAL when handling infrequent cases that require large amounts of processing. Finally, HARDWARE and SOFTWARE are initially flat in the 50% and 100% curves because 20 operations are not enough to completely fill the branch delay slots in a minimum length loop on Imagine.

The other microbenchmark, *load-balance*, is based on the example presented in Section 3.3, except instead of a trivial decrement each iteration, 60 and 100 arithmetic operations per output element are simulated. The results are in Figure 13 for 10 different input datasets that consist of 256 randomly generated elements. Within a dataset, the number of outputs produced when processing its elements is uniformly distributed with a mean of 10. The x-axis of the graph is the standard deviation of the number of outputs produced by the elements within a dataset. The total number of outputs generated by each dataset differ by less than 1%. Each dataset will thus require roughly the same number of cycles to execute in the ideally load-balanced case.

Both conditional streams implementations offer similar performance regardless of the standard deviation in the processing time for each dataset. In contrast, TRADITIONAL incurs a synchronization penalty between elements, thus requiring more time as the standard deviation of the dataset increases. The results for the dataset with a standard deviation of zero indicate that the overhead of dealing with load-imbalance is larger in TRADITIONAL than in HARDWARE. The overhead in TRADITIONAL is due to synchronizing the PEs, while the overhead in HARDWARE is due to extra control, buffering, and communication required for conditional streams.

6.2. Merge Sort

The first application considered is a merge sort, which sorts two smaller pre-sorted streams into one larger combined sorted stream, and which can be used as a building block for a larger full sort. Merge sort is an

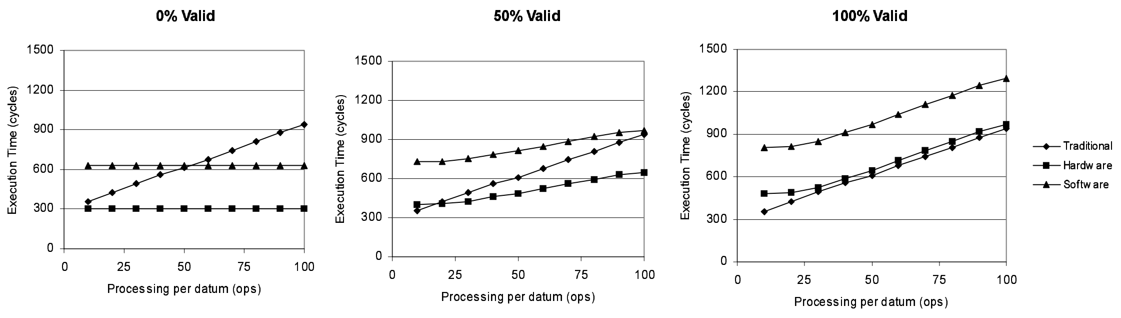


Figure 12: Conditional switching microbenchmark results

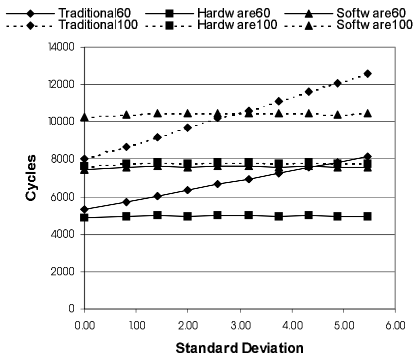


Figure 13: Conditional load-balance microbenchmark results

important example because it cannot be implemented in any reasonable fashion on a stream processor without conditional streams. This is because the rate of expansion of the two input streams must be determined while analyzing the data in the two streams. A stream of case values cannot be generated separately as in the *interleave* example of Section 3.2, because the two input streams would have to be expanded in order to generate the case stream, which was the initial problem.

Conditional streams, though, do allow an elegant solution on a stream processor. Initially, each of the n PEs reads one element from each input stream ($2n$ elements total). Software coordinates PE communication in order to determine the smallest n elements out of the $2n$ elements read. These elements are then rearranged in increasing order and output. Each PE then reads data from an input stream only if the value it previously read from that stream was small enough to be sent to the output stream. Now, the situation is the same as the initial case, and this process repeats until both streams are exhausted.

By allowing the expansion of the input streams to occur between the PE array and the register file instead of in the memory system, a merge sort can be efficiently implemented on a stream processor with a straightforward kernel. Without this ability, either a full sort or an algorithm with multiple passes is required.

6.3. Polygon Rendering

A typical polygon rendering pipeline [6] was implemented on the TRADITIONAL and HARDWARE configurations. The pipeline has three stages: geometry, rasterization, and composition. The application is strip-mined such that batches of triangles are loaded into the SRF and carried through the whole pipeline together. The batch size was chosen to be as large as possible while still allowing all temporary data to fit fully in the SRF without requiring memory spills. This size was 192 triangles for HARDWARE and 40 triangles for TRADITIONAL.

The geometry portion of the pipeline, which is perfectly data-parallel, transforms a stream of triangles from model space to screen space and performs lighting calculations. The rasterization stage generates spans from triangles, and then rasterizes each span to produce several fragments. At this point, a hash kernel separates out the fragments within the stream which are at a unique screen location from those that fall on the same screen location as another fragment within the batch.

A z-comparison test is performed on the stream of unique fragments using the old z-buffer values, and the fragments that pass are used to update the z- and frame-buffers. The stream of conflicting fragments is then sorted according to screen location to ensure all fragments at the same screen location are near each other in the stream. This allows the rest of the fragment composition stage to easily handle these exception cases. The fastest sort methods in this particular situation are a full merge sort for HARDWARE and a bitonic sort for TRADITIONAL. A complete description of the implementation on Imagine can be found in [7].

Note that the hash and sort are required because the application does not process one triangle at a time. Instead it amortizes control and memory access overheads over a whole stream of triangles (a batch). Also, the application requires that all fragments must be composited in the same order as the originating triangles in the source data. Since both implementations in the rasterization stage can potentially reorder the data in a way that violates this requirement, a small overhead is incurred. In particular, each triangle is assigned an ID that is carried through the pipeline until the sort, where it is used as a secondary sort key. However, note that HARDWARE does not incur any additonal penalty over TRADITIONAL. While the hash kernel splits all fragments into two streams, these two streams do not need to be composited in order or together. Also, the sort is required for the conflicting fragments for both configurations, not just HARDWARE.

Finally, intermediate streams in the TRADITIONAL configuration may contain invalid values. For example, one particular kernel marks all backward facing triangles as invalid. Also, the rasterization kernels produce invalid output as some PEs idle waiting for others to finish processing taller triangles or wider spans. These invalid values affect the duty factor of kernels later in the pipeline. To mitigate this, streams are compressed through memory at opportune points in the pipeline. These streams are stored using a mask, so that the stream will reside in a compressed form in memory, and will return to the SRF via a regular load operation. The final z-buffer and frame-buffer stores are also predicated so that only those fragments passing the z-test will generate writes to external memory.

Table 1: Polygon Rendering Results

Configuration	Batch Size (Triangles)	Execution (Cycles)	Mem. Accesses
TRADITIONAL	40	1,068,777	348,992
HARDWARE	40	836,881	131,080
HARDWARE	192	583,457	131,488
HARDWARE4	192	593,049	131,488

Table 1 shows the results of running the application on the two configurations. Comparing the results for a batch size of 40 on both configurations shows a speedup of 1.3x when using conditional streams. Two factors are responsible for this speedup. First, conditional streams increase the duty factor of PEs by using conditional switching and load-balancing. Second, TRADITIONAL has to periodically make extra memory references to compress streams that are diluted with NULLs.

The largest batch size TRADITIONAL can process without spilling any intermediate streams to memory is 40 triangles. In comparison, HARDWARE can process 192 triangles per batch since all streams are compressed while in the SRF. This reduces overhead incurred for loop prologues and epilogues. Since kernels are usually heavily software pipelined, this overhead can be significant, and a larger batch size considerably reduces the execution time of the application. For this scene a 1.8x speedup over TRADITIONAL was achieved.

The final configuration evaluated was HARDWARE4. This configuration has an inter-processor switch that is pipelined, but that has a latency four times that of the switch in the HARDWARE configuration. In all other respects it is similar to HARDWARE. The switch latency may increase as the number of PEs increase, and this may affect the performance of applications that use conditional streams since all data in a conditional stream is routed through the switch. However, this increase in switch latency has minimal effect on performace for this application, showing only a 9.8% slowdown when the latency is increased by a factor of four.

7. Related Work

A common approach to handling conditionals, as found in the early Solomon machine [12] and other data-parallel architectures, is to idle some PEs while others compute. Many vector processors, such as the Cray-1 [11] and the VPP500 [15], take a similar approach. On almost all vector processors, the only way to compress or expand a vector is through gather/scatter instructions that cycle data through memory. A notable exception is the NEC SX Supercomputer [16], which provides vector register-register compress and expand instructions. While these instruction don't actually touch any memory locations, they still occupy the global register file port(s), as well as the memory system port(s) and switch, potentially interfering with other critical memory loads and stores.

Smith, et al. provide a good summary of support for conditionals on vector processors to date and also conclude that solutions whose performance depends on the fraction of valid values in a vector (*density-time*) are preferred to those whose performance depends simply on the length of the vector (*VL-time*) [13]. For example, register-register primitives allow *density-time* processing. They, however, propose a new density-time solution that modifies the register file ports. Their implementation increments the vector register address to point to the next valid input entry based on the VM (a vector mask register). Memory addresses are also adjusted, requiring a multiplier for strided accesses. In addition, limited load-balancing is possible with their approach.

Conditional streams also achieve *density-time* processing. However, conditional streams are applicable to a larger variety of situations, such as merge sorting and optimal load-balancing. Also, in contrast, conditional streams use global register file bandwidth and storage more efficiently since streams diluted with invalid entries are never stored in the global register file.

8. Conclusions

Data-parallel architectures effectively exploit the parallelism in applications expressed in the stream programming model. However, data-dependent control constructs reduce efficiency as they do not map well to these architectures. Conditional streams are a mechanism to convert these control constructs into data-dependent data routing, and can result in significant speedups on media processing applications such as polygon rendering.

Data-dependent control flow requires each PE to perform different operations on its data, which is not possible with SIMD control. Instead, conditional streams convert control decisions into routing decisions. Data streams are transferred to and from (expanded and compressed) the PEs in such a way that all PEs execute the same control flow while still computing useful data. This functionality is exposed to the programmer by allowing stream accesses to be conditional upon a case value local to each PE. This simple abstraction compresses and expands data streams for the three modes presented in this paper: conditional switching, conditional combining, and conditional load-balancing. All of these modes replace data-dependent control constructs with the expansion or compression of data streams without additional memory accesses.

Conditional streams may be used in other modes of operation and are applicable to data-parallel applications with data-dependent control. The concept of conditional streams can also be implemented in software on existing SIMD architectures, but the full performance advantage is realized with additional hardware control. Existing infrastructure, such as the interprocessor switch, is leveraged to reduce the amount of additional hardware required. Hardware-based conditional streams on the Imagine stream processor enable a 1.8x speedup on a polygon rendering application. Furthermore, conditional streams enable efficient merge sorting, which is not possible otherwise.

Performance advantages of SIMD machines can often be offset by their inability to execute data-dependent control flow on many applications. Conditional streams extend the range and efficiency of applications possible on data-parallel architectures. The range of problems for which conditional streams are applicable can be expanded by further investigation of other modes of operation. Also, this paper presented an hardware implementation on the Imagine stream processor. Implementations on other architectures and configurations will potentially require different tradeoffs on how to leverage already existing hardware in order to get the best overall application performance.

Acknowledgements

The authors would like to thank Brian Towles for his help with the experiments, as well as all the other Imagine project members for their contributions to this paper and the project. Finally, we would like to thank everyone who read early drafts of the paper for providing helpful and insightful comments.

References

[1] ESPASA, ROGER, VALERO, MATEO, AND SMITH, JAMES E. Vector Architectures: Past, Present and Future. In *Proceedings of the 1998 International Conference on Supercomputing*, pp. 425-432.

[2] HENNESSY, JOHN L., AND PATTERSON, DAVID A. *Computer Architecture: A Quantitative Approach*, Appendix B, Morgan Kaufmann Publishers, Inc: San Francisco, California, 1996.

[3] HWANG, KAI. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, Chapter 8, McGraw-Hill, Inc: New York, New York, 1993.

[4] JACK, KEITH. *Video Demystified: A Handbook for the Digital Engineer*, LLH Technology Publishing: Eagle Rock, Virginia, 1996.

[5] LEE, CORINNA G., AND STOODLEY, MARK G. Simple Vector Microprocessors for Multimedia Applications. In *Proceedings of the 31st International Symposium on Microarchitecture* (November, 1998), pp. 25-36.

[6] OPENGL ARCHITECTURE REVIEW BOARD. OpenGL Reference Manual: the Official Reference Document to OpenGL, Version 1.1, Addison-Wesley Developers Press, Reading, Massachusetts, 1997.

[7] OWENS, JOHN D., ET AL. Polygon Rendering on a Stream Architecture. *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (August 2000), pp 23-32.

[8] PELEG, ALEX AND WEISER, URI. MMX Technology Extension to the Intel Architecture. In *IEEE Micro* (August, 1996), pp. 42-50.

[9] RIXNER, SCOTT, ET AL. A Bandwidth-Efficient Architecture for Media Processing. In *Proceedings of the 31st International Symposium on Microarchitecture* (November, 1998), pp. 3-13.

[10] RIXNER, SCOTT, ET AL. Register Organization for Media Processing. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture* (January, 2000), pp. 375-387.

[11] RUSSELL, RICHARD M. The Cray-1 Computer System. In *Communications of the ACM* (January 1978), pp. 63-72.

[12] SLOTNICK, DANIEL L., ET AL. The Solomon Computer. In *Proceedings of the Fall Joint Computer Conference* (1962), pp. 97-107.

[13] SMITH, J. E., FAANES, GREG, AND SUGUMAR, RABIN. Vector Instruction Set Support for Conditional Operations. In *Proceedings International Symposium on Computer Architecture* (June, 2000), pp. 260-269.

[14] TREMBLAY, MARC, ET AL. VIS Speeds New Media Processing. In *IEEE Micro* (August, 1996), pp. 10-20.

[15] UTSUMI, TERUO, ET AL. Architecture of the VPP500 Parallel Supercomputer. In *Proceedings of the Conference on Supercomputing* (November 1994), pp. 478-487.

[16] WATANABE, T., ET AL. The Supercomputer SX System: An Overview. In *Proceedings of the Second International Conference on Supercomputing* (November, 1987), pp. 51-56.

[17] WAWRZYNEK, JOHN, ET AL. Spert-II: A Vector Microprocessor System. In *IEEE Computer (March, 1996)*, pp. 79-86.