A Data-Parallel Programming Model for Reconfigurable Architectures

Steven A. Guccione
Mario J. Gonzalez
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712

Recently, several machines have been built using Field Programmable Gate Array (FPGA) technology. These reconfigurable architectures have demonstrated very high performance for a variety of problems. The configuration of these machines typically rely on some form of hardware specification. In this paper we demonstrate that a more traditional software approach may be used. A vector based data-parallel model and its mapping to a reconfigurable architecture are introduced. Included in the model are parallel prefix or scan operators. The language supporting this model is a subset of the C programming language.

1 Introduction

Recently, several high performance coprocessors have been built using FPGA technology [1] [4], [5], [8] [12]. These systems have shown a very high level of performance using a relatively small amount of hardware.

These architectures, however, differ dramatically from conventional von Neumann machines. This has made it difficult to provide high level support for development that users of traditional machines normally require.

This paper describes a programming model and a simple programming language that may be used to implement algorithms for a general purpose reconfigurable system.

2 Reconfigurable Architectures

All of the reconfigurable systems referenced have a similar architecture. A central Reconfigurable Processor Unit (RPU) is connected to a memory system. A host computer is used to configure the RPU and to provide input / output support.

This architecture is very flexible and may be used in a variety of ways. The most obvious use of such a machine is to configure the RPU to behave like an existing commercial microprocessor. The ability to emulate existing processors permits the use of a familiar software environment. Unfortunately, it is not likely to be faster than existing custom devices and will most likely require an order of magnitude more hardware.

Another approach is to use the RPU as a custom circuit implementation of a specific algorithm. This approach provides very high performance for a given algorithm. Unfortunately, the programmer must be skilled in the use of circuit design tools.

For any potential mainstream use of reconfigurable architectures, both high performance and conventional programming language support will probably be necessary.

3 A Vector-Based Data Parallel Programming Model

In a reconfigurable machine, there are two distinct phases of operation. The first is the configuration of the processor. During this phase a circuit is set up in the RPU. The second is the processing phase. Here, arithmetic and logical operations are performed on data. During the configuration phase, no actual processing of data occurs. To reduce the overhead of configuration, it is desirable to minimize the time spent in configuration and maximize the amount of time spent processing data.

In the "custom circuit" approach, configuration is done once before any processing of data begins. Reconfiguration in the middle of an algorithm is not usually performed. This style of usage of the RPU is excellent for simple applications which require high bandwidth. Unfortunately, only a few algorithms are able take advantage of this approach.

What is desirable is some method of minimizing the amount of time spent configuring the RPU, while still performing large amounts of work. One model which fits this description is a vector model of computation. Here, the circuit is configured once, and streams of data are processed. Clearly, longer vectors will minimize the reconfiguration overhead.

With this approach, a reconfigurable architecture can be used much like a traditional vector processor, but with several advantages. First, any arbitrary vector operation may be implemented in the RPU. In traditional architectures, the vector operations are limited to the those defined by the machine instruction set. Second, the circuits performing these operations can be pipelined, increasing the throughput of the system.

The use of pipelining of vector operations should provide very high raw performance. In reality, however, the structure of many algorithms makes it difficult to achieve high performance via pipelining. The most common barrier to vector performance is data dependencies. A data dependency occurs when the result of an operation currently in the pipeline is required to complete before another can begin. There are several established techniques for handling data dependencies in a pipeline. Most rely on complex scheduling and reservation schemes.

Other methods of handling dependencies such as stalling or flushing the pipeline can dramatically reduce performance. With a reconfigurable architecture, the pipelines may potentially be very deep. The penalty for a data dependency may cause an unacceptable reduction in the performance of the particular algorithm. What is desirable is a model of computation which handles data dependencies in vector calculations in a graceful manner.

One model of computation that fits this description is the *data parallel* model. in this model, all operations are performed in parallel on a vector of data. Since data items in the vectors are processed independently, algorithms developed with this model exhibit no data dependencies. This model has been successfully used in commercial SIMD systems [6] and has even been applied to MIMD multiprocessors [7].

4 Vector and SIMD Architectures

Since this model is so closely associated with SIMD architectures, the relationship between the data parallel model, SIMD architectures and pipelines should be discussed.

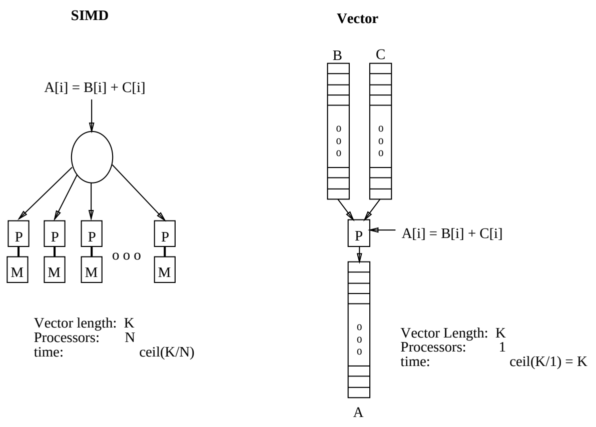

Figure 1: SIMD vs. vector parallelism.

In a SIMD machine, N (usually) identical processors perform operations in lock-step on local data. This permits N operations to be performed per instruction cycle. Data is occasionally exchanged between processors via an interconnection network.

Using a pipeline approach, a single processor can perform N operations in N cycles, plus some initial startup cost. This reduces the hardware complexity by at least a factor of N. Performance, however, may still be competitive, despite this hardware reduction. The instruction cycle time of a pipelined system may be several orders of magnitude faster than that of a SIMD machine.

Figure 1 illustrates vector and SIMD architectures. By trading computation time for hardware, the vector model of computation can be viewed as the dual of the SIMD model. Because of this relationship, the SIMD vector based data parallel programming model can be applied to a pipelined vector machine.

5 Programming Language Support

Language support for vector operations has been available for some time. APL, a language with built in vector support, has been available since the early 1970s. Vectorizing FORTRAN compilers have also been commercially available for several years. These two languages are excellent candidates for a programming language for a vector based reconfigurable architecture. Both, however, have drawbacks.

APL, a very powerful language for vector operations, has not achieved widespread popularity. This is most likely due to its somewhat unusual syntax. Vector based FORTRAN compilers have the disadvantage that they rely heavily on the traditional von Neumann model of computation. The vector additions are at best afterthoughts.

Functional programming languages are also possible candidates for reconfigurable architectures. These languages tend to be more architecture independent and provide vector support. Backus' *FP* is a good example of a functional language that includes vector support [2].

Several interesting programming languages for SIMD architectures have been defined and implemented. In particular, **LISP* and *C** for the Connection Machine [6] have achieved some popularity. These languages support vectors and data parallel programming. All of these would make excellent candidates for a reconfigurable architecture.

As a more practical concern, the popularity of languages like *C* cannot be ignored. If the goal is to provide a programming environment usable by today's programmer, *C* would be the obvious first choice for a programming language.

Some interesting work has been done using the *C* language on a reconfigurable architecture [1]. Our approach is to take features of the languages above and use them in a language which which is very similar to *C*.

6 A C-like Programming Language

The first problem in using *C* to program a reconfigurable architecture is its inherently von Neumann style. Data structures such as pointers directly correspond to memory addresses. Control structures such as loops correspond to branches. These sorts of structures will likely not translate well to a reconfigurable architecture.

Consider the simple case of adding two vectors. In *C*, the following code is used:

```
int  i;
int  a[100];
int  b[100];
int  c[100];

for (i=0; i<100; i++)
    c[i] = a[i] + b[i];
```

The declaration of the vectors is suitable for our purposes. However, the explicit control structure provided by the *FOR()* loop is redundant and can be eliminated. This results in the following "vector" *C* code:

```
int  a[100];
int  b[100];
int  c[100];

c = a + b;
```

In this simple case, we see that three vectors of integers of 100 elements each must be allocated. The RPU should be configured as a simple integer adder. Translating such a program fragment to run on a reconfigurable architecture is very straightforward.

It should be mentioned that this programming style is gaining popularity on conventional architectures. Languages such as *C++* and *Ada* permit the overloading of operators such as "+". This allows vectors to be added using simple infix notation without the explicit use of control structures.

If simple vector operations are performed exclusively, this model should be sufficient. Unfortunately, the lack of other *C*-style control structures such as if-then statements and pointers will diminish the power of this language.

Either some way must be found to support these control structures, or new control structures must be introduced.

7 Scan Primitives

One construct popularized by SIMD architectures is the *parallel prefix* or *scan* operation. This operation is fairly simple and bridges the gap between vector and scalar operations. A simple example is the *scan-add* function. This function is denoted by *+-scan()* in the Connection Machine literature. We will also use this notation.

The *+-scan()* function takes in a vector and returns a vector. The function adds the elements in vector returning partial sums. For example, consider the code fragment below.

```
A[] = {1, 4, 7, 2, 6, 0, 3};

A = +-scan(A);
```

The initial and final values for the vector A[] will be:

```
A:          [1, 4,  7,  2,  6,  0,  3]

+-scan(A): [1, 5, 12, 14, 20, 20, 23]
```

This scan operation provides the necessary functionality to perform operations such as accumulation. Other scan primitives can include *max-scan*, *min-scan*, *and-scan*, *or-scan*, **-scan*, etc ...

These scan or parallel prefix operators have been available in *APL* for some time, and have more recently been popularized as a parallel processing construct on SIMD style machines. In SIMD machines, the implementation of scan operations make extensive use of the interprocessor communication network. Data is propagated from one processor to another, either in a serial or tree fashion. While this makes scan operations a useful communication technique, the extensive use of the network makes these operations expensive on SIMD hardware.

Interestingly, these operations can be implemented very efficiently using a reconfigurable architecture. A simple macrocell implementation of the *+-scan* primitive would be a simple adder with the sum fed back into one of the inputs. Figure 2 illustrates this macrocell. Other scan operators can be implemented in a similar manner.

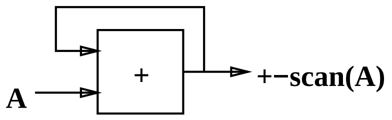

Figure 2: A macrocell for the +-scan operator.

For a more detailed look at scan primitives, see [3], [6], and [10].

8 A Simple Example: e^x

Below is an example calculation implemented using a data parallel approach and scans. This calculation is used to compute the value of e^x. The technique used is the summation of the Taylor series:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} \cdots$$

This particular example was chosen not so much for its high computation requirements, but for its simplicity. It contains a small number of operations, several of which are scans.

The calculation of the series can be performed with the following code fragment:

```
float denom[MAX];            /* Denominator */
float neum[MAX];             /* Numerator */
float series[MAX];           /* Series terms */
float sum[MAX];              /* Series sum */

denom = 1;                   /* [1, 1, 1, 1 ...] */
denom = +-scan(denom);       /* [1, 2, 3, 4 ...] */
denom = *-scan(denom);       /* [1, 2, 6, 24 ...] */
                             /* [1!, 2!, 3! ...] */

neum = x;                    /* [x, x, x, x ...] */
neum = *-scan(neum);         /* [x, x^2, x^3,...] */

series = neum / denom;
sum = +-scan(series);        /* e^x - 1 */
```

Four data vectors are declared. These contain the values in the numerator and denominator of the series, the resultant series and the final sum. The first step in the algorithm is to initialize the denominator vector to 1. Note that the constant initialization of a vector is permitted.

A *+-scan* performed on the denominator produces a vector of sequential non-negative integers. A **-scan* on these integers produces a vector of sequential factorials. At this point, the denominator of the series has been computed.

In a fashion similar to the initialization of the denominator, the numerator vector, *neum* is initialized to x. A **-scan* is then performed on this vector to produce the numerator series.

The numerator and denominator vectors are then divided, producing a vector containing the terms of the series. All that is required now is that the series be summed. This is accomplished with a final *+-scan*. This produces a vector containing the partial sums of the series. Each element in the vector will contain a better successive approximation of $e^x - 1$.

It should be noted that $e^x - 1$ rather than e^x is calculated. This is done to simplify the example. To calculate the series without ignoring the initial term of 1, the first element in the *neum* and *denom* vectors may be set to 1. Alternately, a final addition of 1 to the *sum* vector can be performed.

9 Compilation

In the e^x example above, two *+-scan*, two **-scan* and a vector divide are required. The first approach to implementing this algorithm on a reconfigurable machine would be to configure the RPU once for each operation.

For instance, the first operation is the initialization of the vector *denom* to 1. The RPU would be configured as a circuit that produces a constant “1” for any input. The *denom* vector would be processed by the RPU, with the results being stored in a temporary vector.

The RPU would next be reconfigured to perform a *+-scan* operation. The temporary vector would be processed by the *+-scan* circuit and stored in a second temporary vector. This process would continue until the final *sum* output vector is produced.

The alternating of reconfiguration and processing is well suited to vector processing. This approach, however, does have some drawbacks. The reconfiguration of the RPU for each line of code is reminiscent of interpreted languages. This interpreted mode of operation will make use of such simple operations that the utilization of the RPU may be very low. This approach also ignores any parallelism in the algorithm.

It would be desirable to combine operations and allow multiple functional units (FUs) to be configured to operate in parallel. These functional units could operate in parallel horizontally, providing superscalar style processing.

Inspection of the code for the e^x calculation reveals that initialization of the *denom* and *neum* vectors are independent and can be performed in parallel. It is possible to combine these operations into a single step. Two circuits, one initializing *denom* to 1 and one initializing *neum* to x can be configured in the RPU. From this combined circuit, the *neum* and *denom* vectors can be initialized simultaneously. This exploitation of spatial parallelism may be limited by the memory bandwidth of the underlying system. Here it is assumed that the bus connecting the RPU and the memory system is wide enough to support the simultaneous input and output of two vectors. Figure 3 illustrates this RPU configuration.

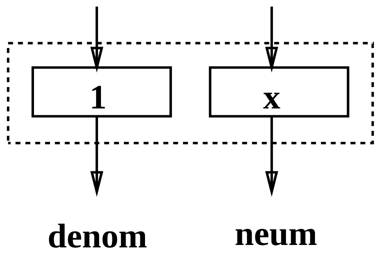

Figure 3: Functional units operating in parallel.

The functional units could also be cascaded vertically, with the output of one unit feeding the input of another. Consider the last two calculations in the algorithm for e^x. The result of the vector division is stored to the *series* vector. This vector is then used as the input to an *+-scan* operation. Rather than perform these two operations in two passes, a composite circuit can be configured that contains a vector divide FU with its output feeding a *+-scan* FU. This permits two operations to be performed in a single pass. Figure 4 illustrates this RPU configuration.

There are two effects of this cascading of functions. First, the *series* vector was originally stored to memory as a result of the vector divide. This was then input to the a *+-scan* FU in the subsequent operation. Now, since the output of the vector divide directly feeds the input of the *+-scan*, there is no longer a need to allocate the *sum* vector. The results are passed directly in the pipeline without ever accessing system memory. Not only does this cascading of operations eliminate a reconfiguration and a processing step in the calculation, but it also eliminates the memory storage required for a vector.

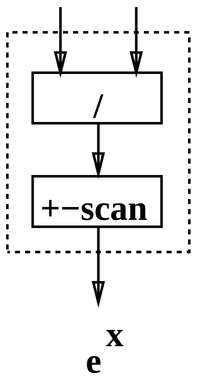

Figure 4: Cascaded functional units.

The second result of this cascading of functions is an increase in the delay of the circuit. Initially, results were available after some delay Δ. Assuming that all functional units have an equal delay, the result is now available at some time 2Δ.

This increase in delay may impact performance by reducing the maximum cycle time of the system. As more functions are cascaded, this delay could potentially grow large. The alternatives to accommodate this variable delay at the hardware level are to either provide a settable system clock or to limit the depth of cascaded FUs.

Another alternative that does not sacrifice performance is to pipeline the functional units. If each FU is implemented with registered outputs and all FUs are driven by a common clock, the clock speed of the original system can be maintained, even in the presence of multiple pipelined functional units. The only penalty paid for the pipelining will be an increased latency. For a configured N-deep pipeline of FUs, the latency will be $N\Delta$. For long vectors, however, this will have a minimal effect on overall performance.

A method for extracting the parallelism in an algorithm is desired. One approach for exposing parallelism is to construct the dataflow graph of the program. The left hand side of figure 5 shows the dataflow graph for the code fragment for the computation of e^x.

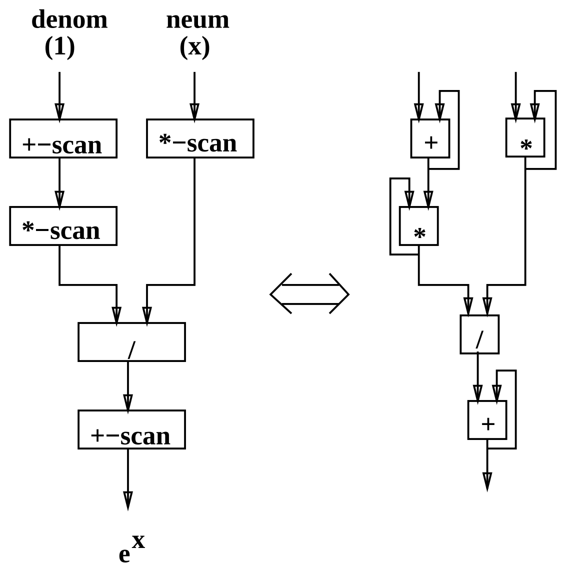

Figure 5: Dataflow graph and circuit for e^x.

The dataflow graph of the algorithm provides all of the information necessary to configure a circuit to implement the algorithm. First, the nodes in the dataflow graph correspond directly to the FUs in the circuit. Next, the edges in the dataflow graph define the bus interconnections between the functional units. Finally, it is possible that the topology of the graph may provide direction in the placement of the FUs and the routing of the interconnect. This is still under investigation.

Using the dataflow graph, the right side of figure 5 shows a circuit diagram for the e^x circuit. This circuit can be configured into the RPU and used to perform the calculation specified by the software.

Two further features of this circuit should be mentioned. First, it is typical that data from vectors in memory will be input to the RPU, with results produced at the RPU output. In this case, it is possible that the *denom* vector can consist of an array of memory locations initialized to 1. This, however, is unnecessary. Since a constant (1) is being input to the circuit, this may be "hardwired", thus saving memory space and memory bandwidth. A similar situation exists for the *neum* vector.

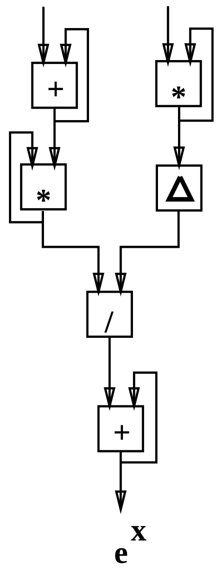

Figure 6: The circuit for e^x with delay stage inserted.

A second notable feature of this circuit is the joining of the two numerator and denominator calculation pipelines into a single pipeline. When constructing this circuit, it is crucial that the delay in the two upper branches of the dataflow circuit be balanced. If all FUs have an equal delay, it would be necessary to insert a delay stage in the right branch to balance the circuit. Figure 6 shows the circuit for e^x with the delay stage inserted.

10 Optimizations

Several standard compiler optimizations may be performed on the dataflow graph before generating the circuit. First, the elimination of common subexpressions is possible. In traditional architectures, once an expression is computed, the result can be stored and reused. The reuse of the previously calculated value reduces the time of computation.

In a reconfigurable architecture, common subexpressions do not increase the calculation time, but they do unnecessarily use RPU resources. Figure 7 illustrates this in the calculation of a simple polynomial. Here, the expression $2x^2+6x$ is reduced to $(2x)(x+3)$. This results in the elimination of a multiply FU. The depth of the pipeline is also reduced from 3 to 2. Other optimizations may still be possible on the reduced circuit.

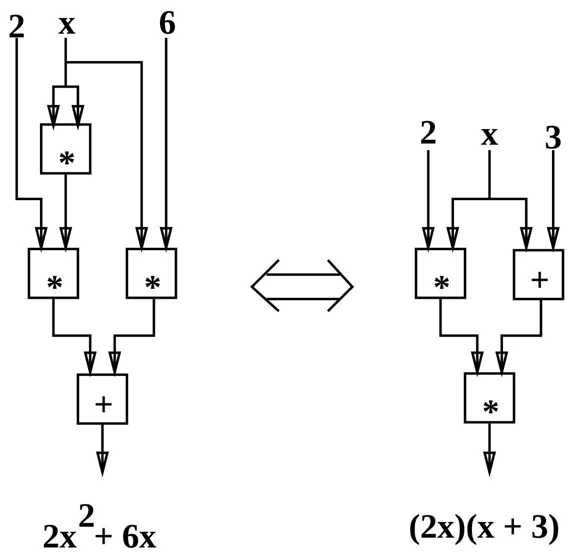

Figure 7: Optimization by common subexpression elimination.

Strength reduction of operators is also possible. The calculation $(2 * x)$ in the software description of an algorithm results in the use of a multiplier FU. This can be replaced by the equivalent expression $(x + x)$, which uses only an addition FU. The conversion from a multiply FU to an adder FU can result in large reduction in RPU usage. Figure 8 illustrates an example of a strength reduction optimization.

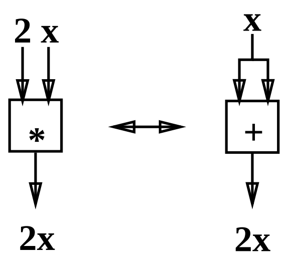

Figure 8: Optimization by strength reduction.

In general, most optimizations used by traditional compilers to reduce calculation time can also be used to reduce the required hardware resources in a reconfigurable architecture. Optimizations which are based on modifying the control flow of a program are usually not applicable, however.

In addition to these standard optimizations, new optimizations may be used with a reconfigurable architecture. For instance, variable datapath widths may be implemented. If it can be guaranteed that the range of values in the vector can be represented by N bits, it is possible to use N-bit wide hardware. Conventional systems typically limit datapath widths to even multiples of eight bits. This is no longer necessary.

In addition to using only the necessary datapath width, internal datapaths within the RPU may grow and shrink depending on the required accuracy of the calculation. A unit multiplying two N bit numbers may produce a $2N$ bit result. The inputs to this multiplier unit can be N bits, with the output and subsequent processing stages being $2N$. This technique can eliminate certain exception conditions such as overflow.

11 Architectural Issues

The use of a vector based data parallel programming model will influence the hardware implementation of the system. In general, the system will consist of three major components, the host, the RPU and the memory system.

The host is responsible for providing the user interface and the I/O facilities. The compiler and other tools reside on the host. In general, any commercially available workstation of personal computer can be used as a host. Perhaps the feature of the host that will impact performance the most is the system bus bandwidth. Since large amounts of data may be transferred from the host to reconfigurable coprocessor, a high bus bandwidth is preferred.

The RPU is an array of several FPGA devices. The width of the array is matched to the bus width of the memory system. Since the software uses a dataflow graph to configure the RPU, a top to bottom flow through the RPU is desirable. While most FPGA devices supply configurable I/O pins at the perimeter of the device, input of data at the top of the array and output at the bottom will be the standard configuration.

While the width of the RPU array will be determined by the RPU-memory bus, the depth of the RPU is arbitrary. Since some algorithms can utilize a very deep pipeline, it is desirable to make the RPU array as deep as practical.

The last, and perhaps most important part of the system, is the memory. The use of a vector based programming model will effect the design of the memory system. Since the RPU is pipelined, the memory

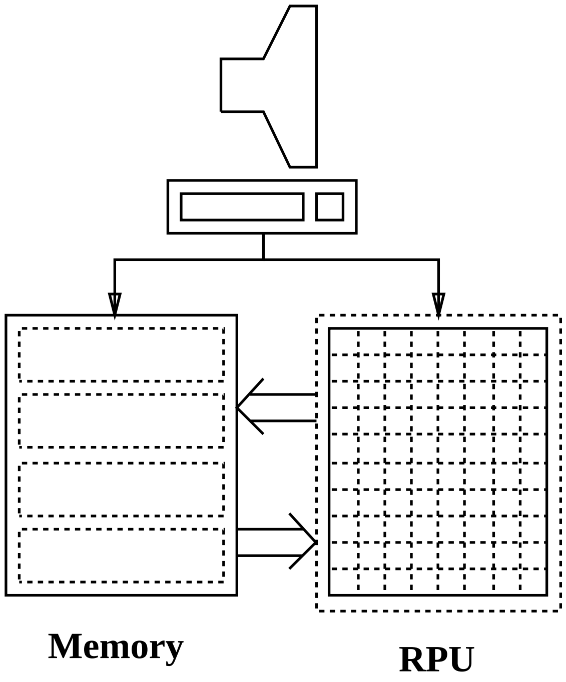

Figure 9: A reconfigurable system.

bandwidth requirements for the system can be very high.

Data reference patterns, however, will be very deterministic. Since the vector is the primary data type, data will be processed from sequential memory locations. Because of this simple memory addressing pattern, it is possible to provide a high-performance memory system at a modest cost.

One approach is to use memory that provides fast serial access. Video RAM is an ideal candidate for this application. The serialized video outputs can be used to feed sequentially stored vectors to the input of the RPU, while the second port of the video RAM can be used to store the output results.

Another approach is to use interleaved memory. Since all accesses to memory within a vector operation are sequential, the use of several banks of interleaved memory can provide the sustained bandwidth necessary to supply data to the RPU. Interleaving should allow slower and denser DRAM to be used in the place of SRAM.

12 Summary and Conclusions

Several other algorithms have been coded using the data parallel model, but space prohibits their discussion here. The literature for SIMD processors offers many examples of large, real-world problems that can be solved with this model of computation.

The vector based data-parallel model of computation is well suited to reconfigurable architectures. The use of vector operations permits large amounts of calculation to be performed with occasional reconfiguration. In addition, the parallel prefix or scan operators used by this model have a particularly efficient implementation.

Translations of the data parallel algorithms to custom circuits has been shown to be a straightforward mapping of the algorithm dataflow graph to RPU hardware. The use of the dataflow graph in the algorithm implementation permits both temporal and spatial parallelism to be exploited. In addition, several established optimizations can be used to reduce the complexity of the circuit.

Finally, the vector model permits extensive use of pipelining to increase performance. An added benefit of the data parallel model is that it eliminates the data dependency problems that often restrict the performance of pipelined systems. The vector model also allows simple and inexpensive memory systems to be used to provide the high bandwidth required by the system.

Reconfigurable architectures promise very high performance at a very low cost. Languages based on the data parallel model should make these architectures easy to program without sacrificing performance.

References

[1] Peter M. Athanas. *An Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration.* Technical Report LEMS-101, Brown University, Division of Engineering, February 1992.

[2] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs (1977 ACM Turing Award lecture). *Communications of the ACM*, 21(8):613–641, August 1978.

[3] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, Cambridge, MA, 1990.

[4] Patrice Bertin, Didier Roncin, and Jean Vuillemin. *Introduction to Programmable Active Memories.* Technical Report 3, DEC Paris Research Laboratory, 1989.

[5] Maya Gokhale, William Holmes, Andrew Kosper, Dick Kunze, Dan Lopresti, Sara Lucas, Ronald Minnich, and Peter Olsen. SPLASH: a reconfigurable linear logic array. In *International Conference on Parallel Processing*, pages I–526–I–532, 1990.

[6] W. Daniel Hillis. *The Connection Machine*. The MIT Press, Cambridge, MA, 1985.

[7] Philip J. Hatcher and Michael J. Quinn. *Data–Parallel Programming on MIMD Computers*. The MIT Press, Cambridge, MA, 1991.

[8] T. A. Kean. *Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation*. PhD thesis, University of Edinburgh, Department of Computer Science, January 1989.

[9] Peter M. Kogge. *The Architecture of Pipelined Computers*. McGraw–Hill, 1981.

[10] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. The power of parallel prefix. In *Proceedings of the International Conference on Parallel Processing*, pages 180–185, August 1985.

[11] H. T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, January 1982.

[12] Jouko Viitanen, Tapio Korpiharju, and Hannu Kiminkinen. Mapping algorithms onto the TUT cellular array processor. In *International Conference on Application Specific Array Processors*, 1990.