Supercomputing with Reconfigurable Architectures

Steven A. Guccione and Mario J. Gonzalez

Computer Engineering Research Center
Department of Electrical and Computer Engineering
University of Texas
Austin, Texas 78712

Abstract. Recently, several research and commercial systems based on reconfigurable logic have been implemented. These machines have demonstrated supercomputer levels of performance for a number of algorithms. While these demonstrations have been impressive, it is not clear that architectures based on reconfigurable logic will necessarily be suitable for algorithms commonly executed on supercomputers. This paper discusses the implementation of Livermore Fortran Kernels for a supercomputer class machine based on reconfigurable logic.

1 Introduction

Recently, a large number of systems based on reconfigurable logic have been designed and built [5]. Some of the largest of these machines have demonstrated supercomputer levels of performance on selected algorithms. As larger reconfigurable machines become available, it is widely believed that they will be used to implement algorithms typically found on existing supercomputers. It is not clear, however, that these architectures are well suited to these tasks. Most of the algorithms implemented to date are more typically found implemented in custom hardware rather than on large general purpose machines.

This study examines the feasibility of general purpose supercomputing using reconfigurable logic. Algorithms selected for the *Livermore FORTRAN Kernels (LFK)* [9] [2] are used to examine the performance of a reconfigurable logic based supercomputer. The LFK are chosen for several reasons.

First, the LFK suite is a widely used tool to measure CPU performance. This allows comparison to a wide range of existing architectures. Second, the LFK are composed of a number of tests which include a wide range of computational structures. While some of these structures are used to measure the peak performance of a system, others are constructed specifically to limit performance. This permits an examination of architectural weaknesses as well as strengths. Finally, the LFK are relatively compact and self contained. This allows their simulation on models of proposed hardware. Performance information gained from such simulations is valuable in guiding the design.

2 The Livermore Fortran Kernels

The LFK are a collection of 24 relatively small fragments of code. Each of these code fragments contains a CPU intensive loop, giving the test suite its informal name, "the Livermore Loops". The LFK were developed in 1970 to test the code generated by compilers. Over time, these codes have become a benchmarking tool for new supercomputer systems.

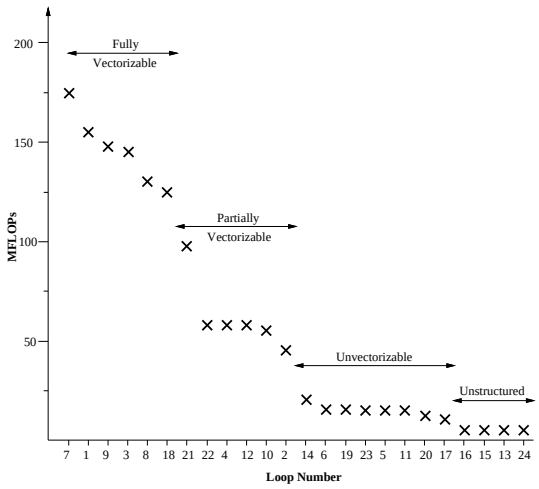

Fig. 1. Performance sorted my MFLOPs for a CRAY X-MP.

As the LFK have evolved into a benchmarking tool, new loops have been added to exercise specific features of both compilers and hardware. The number of loops has grown from the initial 12 to the current 24.

The kernels in this study were converted by hand from the original FORTRAN to a data parallel version of the *C* language. Simulations of circuits extracted from this data parallel code are compared against a standard C version of the LFK [3].

Finally, it should be noted that the LFK are specified for high accuracy floating point arithmetic. While some work is being done on the implementation of floating point arithmetic in reconfigurable logic [1], it is understood that using the technology available today, a very large RPU would be required to implement these functions. While numeric accuracy is important, it is the computational structures in the LFK which are of primary interest. It is these structures, not numeric accuracy, which has the greated impact of performance.

Figure 1 plots the performance of the 24 Livermore Fortran Kernels run on a *CRAY X-MP* using the *CFT77 3.0* compiler [10]. The numbers are listed in MFLOPs and are sorted by performance. From this sorted graph of the LFK,

four distinct performance ranges can be identified. These are: *fully vectorizable*, *partially vectorizable*, *unvectorizable* and *unstructured*.

In general, kernels in each of these regions present different computational challenges. These will be discussed in more detail as the kernels are implemented. For brevity, only representative kernels from each region are discussed. Kernels were chosen primarily for their simplicity in illustrating the particular computational structures.

The reconfigurable hardware platform used for execution of these kernels is assumed to consist of a relatively large *Reconfigurable Processing Unit*, or *RPU*, dedicated memory tightly coupled to the RPU, and a host machine. The reconfigurable portion of the system is assumed to operate at 50 MHz. The highly pipelined circuits combined with the vector data accesses make this feasible.

3 Fully Vectorizable Loops

Kernels in the fully vectorizable category typically perform the highest on vector supercomputers. These kernels are characterized by being easily vectorized as well as providing enough work to occupy multiple functional units.

3.1 Loop 1: Hydrodynamic Code

Loop 1 is a fragment from a hydrodynamic simulation. The original FORTRAN code for this loop is shown in Fig. 2. This loop is easily vectorizable and can make concurrent use of several functional units.

```
      Do 1 k = 1,n
1     X(k) = Q + (Y(k) * ((R * Z(k+10)) + (T * Z(k+11))))
```

Fig. 2. The original FORTRAN code for Loop 1.

The translation of this algorithm to data parallel form is shown in Figure 3. Because of the structure and simplicity of this loop, the similarities between the FORTRAN code, the algorithm and the data parallel code are clear.

```
Z10 = delta(Z, 10);
Z11 = delta(Z10, 1);
X = q + (Y * ((r * Z10) + (t * Z11)));
```

Fig. 3. The data parallel code for Loop 1.

One feature of this implementation which may require further explanation is the use of the *delta()* function. This function is used to provide vectors whose indices are offset some small number of units. The *delta* is literally a delay. By providing a delayed version of the vector, data can be made available as it is required without having to re-access the memory system. The translation from an indexed style of coding is faily simple.

Figure 4 shows the RPU circuit extracted from the dataflow graph of this code. This is the circuit makes use of 5 functional units, and has a latency of 5 functional units.

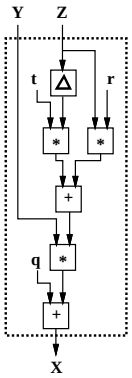

Fig. 4. The configured circuit for loop 1.

Estimating performance of this circuit is fairly simple. Assuming sufficient memory bandwidth and a clock speed of 50 MHz, the processor will produce one result per clock cycle, neglecting latency. Since all functional units are kept busy on each cycle, approximately 250 million operations per second are performed. If the functional units all perform floating point operations, this corresponds to 250 MFLOPs. Even at this modest clock speed, this exceeds the rate of computation of the CRAY X-MP.

3.2 Loop 3 - Inner Product

The second fully vectorizable loop is an inner product calculation. This is a multiply-accumulate function found in many applications, including the matrix arithmetic. Because of the widespread use of this type of calculation, most supercomputers are especially efficient at its execution. The data parallel version of the code is simply the *add-scan()* of a product, as shown in Figure 5.

The circuit extracted from this data parallel code is fairly simple, containing a multiplier and an add-scan functional unit. The memory bandwidth required is also fairly modest. Two vector inputs and a vector output are required.

```
Q = add-scan(Z * X);
```

Fig. 5. The data parallel code for Loop 3.

At a rate of 50 MHz, neglecting overhead, this circuit performs 100 million operation per second. This is somewhat less than the CRAY X-MP. The very small number of functional units indicates very little exploitable parallelism.

4 Partially Vectorizable Loops

The next group of kernels perform at a level somewhat below that of the fully vectorizable kernels. Here, these loops are referred to as *partially vectorizable loops*. In these kernels, the ability to fully use the vector units is reduced. While these loops do not have the performance levels of the fully vectorizable loops, their levels of performance are still substantial, but only a fraction of the peak performance achieved by the fully vectorizable loops.

4.1 Loop 12 - First Difference

Loop 12 is a first difference calculation. Despite its relatively low performance, this loop is fairly simple and is easily translated to data parallel code.

Figure 6 gives the translated data parallel code for the first difference calculation. The use of the *delta()* function provides the offset version of the vector Y, saving input bandwidth.

```
Y1 = delta(Y, 1);
X = Y - Y1;
```

Fig. 6. The data parallel code for Loop 12.

The circuit extracted from this code contains only a *delta* unit and a subtractor. Because this implementation requires only a single functional unit, the calculation proceeds at a rate of 50 million operation per second. This is similar to the rate achieved by the CRAY X-MP. The performance of this loop is reduced because of a lack of work to be performed on the data.

4.2 Loop 22 - Planckian Distribution

Loop 22 is from a Planckian distribution program. Here, the computation rate on traditional vector processors is slowed by conditional execution. Figure 7 gives the data parallel code for this loop.

```
if (U < (V * 20.0))
   Y = (U / V);
   else
      Y = 20.0;

W = X / (exp(Y) - 1.0);
```

Fig. 7. The data parallel code for Loop 22.

In this implementation, the conditional statement provides two alternate values for the elements in Y, depending on the result of the conditional statement. This permits a parallel computation of the two values, with the proper result being selected.

While this is a more complex loop, only 5 functional units, not including the comparison or the the multiplexer, are used. This assumes that the *exp()* function is counted as a single functional unit. Depending on the implementation, this operator may be composed of other simpler arithmetic and logical operations.

Assuming a clock speed of 50 MHz, this implementation achieves approximately 250 million operations per second. This is almost four times the rate of the CRAY X-MP reference machine. This increase is attributed to the ability to efficiently perform conditional operations.

5 Unvectorizable Loops

The kernels in this performance range are typically unvectorizable and are unable to make extensive use of vector hardware. Since they are not able to make use of the vector processing facilities that helped enhance performance in the previous loops, their performance is not only considerably lower, but also more uniform. These algorithms are typically forced to used the non-vector portion of the CPU, thus testing the performance of this portion of the architecture.

Most of these loops are unvectorizable because of data dependencies introduced by recurrence equations. While completely unvectorizable using traditional fixed instruction architectures, the use of structures such as parallel prefix *scan* operators open up new possibilities for these functions.

5.1 Loop 5 - Tridiagonal Elimination

Loop 5 is a fragment of code use in tridiagonal elimination. The original FORTRAN code contains a data dependency in X that prohibits vectorization. This kernel typifies a class of equations known as first order linear recurrence equations. Several approaches to parallelizing these equations have been proposed over the years [8] [7] [4].

The approach demonstrated here makes use of the fact that the computed values are actually independent if previously computed values are substituted

into the subsequent equations. $X(5)$, for instance, can be written as in Equation 1.

$$X_5 = Z_5Y_5 - Z_5Z_4Y_4 + Z_5Z_4Z_3Y_3 - Z_5Z_4Z_3Z_2Y_2 + Z_5Z_4Z_3Z_2X_1 \quad (1)$$

This approach, while producing independent calculations, raises the computational complexity from $O(n)$ to $O(n^3)$. It is, however, possible to represent these equations using *scan* operators. While the actual construction of the *scan* based version of this code is beyond the scope of this paper, the parallel form of the equation contains easily discernible patterns amenable to *scan* operators. Figure 8 gives the data parallel code for this kernel.

```
X = mul-scan(-Z) * (add-scan((Z * Y) /
    mul-scan(-Z)) - x0)
```

Fig. 8. The data parallel code for Loop 5.

From this data parallel code, a pipelined circuit can be extracted. The ability to use non-standard operators such as *scans* has permitted a pipelined version of this kernel, greatly improving performance.

The circuit uses 7 functional units and two vector inputs and a single vector output. At 50 MHz, this circuit calculates 350 million operations per second. While the re-casting of the algorithm has added these extra functional units, thereby boosting the number of operations, the throughput of this circuit is still superior to other implementations, including those on supercomputers. While performance is increased, the new approach to this algorithm introduces some numerical stability problems not found in the original version.

5.2 Loop 11 - First Sum

Loop 11 is a first sum calculation. As in loop 5, a data dependency in the form of a recurrence is responsible for the low performance.

Unlike the recurrence equation in loop 5, the first sum in this kernel is very simple. It is essentially the definition of the *add-scan()* operator. The circuit extracted from this code is again, very simple. A single *add-scan* operator is used. A single vector input Y is used to produce a single vector output X.

While providing a vector solution for this algorithm, the first sum suffers a similar performance limitation to loop 12, the first difference kernel. Since only a single functional unit is used, the number of operations at 50 MHz is only 50 million per second. Even this low rate of calculation, however, still exceeds supercomputer levels of performance.

6 Unstructured Loops

These kernels are the lowest in performance on the CRAY X-MP reference machine. As with the unvectorizable loops, they are unable to take advantage of the special vector hardware. Additionally, these kernels contain structures that further reduce performance, even for the non-vector portion of the processor.

For lack of a better term, these loops will be referred to as *unstructured*. They are characterized primarily by the presence of unstructured control, usually in the form of *goto* statements, as well as complicated array indexing schemes.

Some of these loops actually exhibit a large amount of parallelism. It is often the way in which the algorithm is expressed, rather than any limitation in the underling algorithm, that reduces performance. For these reasons, some of these loops are better test of FORTRAN compiler optimizers than the underlying processor architecture.

6.1 Loop 24 - First Minimum

Loop 24 is selected as a representative of the unstructured loops because it has a deceptively simple implementation, while having the lowest performance of all 24 loops on a CRAY X-MP. The original FORTRAN code for this kernel is given in Figure 9.

```
      max24 = 1
      Do 24 k = 2,n
24      if (X(k) .lt. X(max24)) max24 = k
```

Fig. 9. The original FORTRAN code for Loop 24.

An attempt to translate this algorithm into data parallel code reveals some of its limitations. First, this code uses a conditional operator, which interferes with vectorization. Next, it performs an operation involving only two scalar quantities. Finally, X is indexed by a scalar quantity which changes unpredictably. All of these factors combine to dramatically reduce the performance of this kernel.

The goal of this loop. however, is to find the location of the minimum value in the vector X. Constructing a data parallel solution will require more than a simple translation from the original FORTRAN specification of the algorithm.

Figure 10 gives the data parallel code for this algorithm. In this implementation, the *min-scan()* operator is used to determine the minimum value. The conditional operator is then used to select the index for the minimum value. Since a looping index is not directly available, the *add-scan(1)* statement is used to generate these index values.

```
Min = min-scan(X);
Min1 = delta(Min, 1);
Diff = Min1 < Min;

Index = add-scan(1);
M = max-scan(Index * Diff);
```

Fig. 10. The data parallel code for Loop 24.

While a substantial modification of the original algorithm, this version is fully pipelinable and executes at approximately 200 million operations per second. While much of this figure is due to the additional functional units, this implementation still produces the desired result in approximately N clock cycles for a vector of length N.

A final note on unstructured algorithms. Many may not be suitable for reconfigurable machines. Unstructured access to vector data is a problem. Vector indexing of the form $X[Y[n]]$ is particularly difficult. Without special hardware support in the memory system, this type of calculation will almost certainly involve the host.

7 Conclusions

The table in Figure 11 gives an analysis of the performance of the seven kernels implemented. Perhaps not surprisingly, the algorithms which fared well on supercomputers also fared well on reconfigurable logic based machines. What is more surprising is the high levels of performance achieved by some of the loops which performed poorly on the CRAY X-MP, the supercomputer reference machine.

Loop Number	Vector Inputs	Vector Outputs	Latency	Functional Units	Estimated MFLOPs	CRAY X-MP MFLOPs
1	2	1	5	5	250	160
3	2	1	2	2	100	138
12	1	1	2	1	50	63
22	3	1	5	5	250	68
5	2	1	6	7	450	14
11	1	1	1	1	50	14
24	1	1	5	5	200	3

Fig. 11. The LFK performance parameters.

While many of the algorithms are easily implementable and exhibit very high performance, some structures are still problematic. First, simple recurrences can be implemented efficiently using scan circuits. Currently, however, no simple algorithm exists for translating more complex recurrences into these circuits. Secondly, unstructured algorithms, particularly those which make use of indirect array indexing, are not well suited to reconfigurable logic. Using the host to vectorize these types of array accesses before they are submitted to the RLU may be a solution for some algorithms.

This study was performed primarily to examine the feasibility of general purpose supercomputing using reconfigurable logic. While not a solution to all problems, the results for a large class of popular computational structures is promising. Furthermore, it should be noted that the algorithms in the LFK are taken from real applications, written for traditional architectures. It is possible that new classes of algorithms which exploit the unique features of reconfigurable logic will provide even higher levels of performance for a larger class of problems.

References

1. Barry Fagin and Cyril Renard. Field programmable gate arrays and floating point arithmetic. *IEEE Transactions on Very Large Scale Integrated (VLSI) Systems*, 2(3):365–367, September 1994.
2. John T. Feo. An analysis of the computational and parallel complexity of the livermore loops. *Parallel Computing*, 7(2):163–185, June 1988.
3. Martin Fouts. The Livermore Loops in C. NASA Ames Research Center memo, 1994.
4. Daniel D. Gajski. An algorithm for solving linear recurrence systems on parallel and pipelined machines. *IEEE Transactions on Computers*, C-30:190–206, March 1981.
5. Steven A. Guccione. List of FPGA-based computing machines. World Wide Web page http://www.utexas.edu/~ guccione/HW_list.html, 1994.
6. Steven A. Guccione and Mario J. Gonzalez. A data-parallel programming model for reconfigurable architectures. In Duncan A. Buell and Kenneth L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 79–87, Los Alamitos, CA, April 1993. IEEE Computer Society Press.
7. Peter M. Kogge. Parallel solution of recurrence problems. *IBM Journal of Research and Development*, 18(2):138–148, March 1974.
8. Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, August 1973.
9. Frank H. McMahon. The Livermore Fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, December 1986.
10. Wayne Pfeiffer, Arnold Alagar, Anke Kamrath, Robert H. Leary, and Jack Rogers. Benchmarking and optimization of scientific codes on the CRAY X-MP, CRAY-2 and SCS-40 vector computers. *The Journal of Supercomputing*, 4(2):131–152, June 1990.

This article was processed using the LaTeX macro package with LLNCS style