LA-UR- 04-1807

Approved for public release; distribution is unlimited.

Title: Monte Carlo Radiative Heat Transfer Simulation on a Reconfigurable Computer

Author(s): Maya Gokhale, ISR-3
Christine Ahrens, CCN-12
Jan Frigo, ISR-3
Ron Minnich, CCS-1
Justin Tripp, ISR-3

Submitted to: International Conference on Field Programmable Logic + Applications
Antwerp, Belgium,
August 30 - Sept 1, 2004

Monte Carlo Radiative Heat Transfer Simulation on a Reconfigurable Computer

Maya Gokhale, Christine Ahrens, Jan Frigo, Ron Minnich, and Justin L. Tripp

Los Alamos National Laboratory

Abstract. Recently, the appearance of very large (3 – 10M gate) FPGAs with embedded arithmetic units has opened the door to the possibility of floating point computation on these devices. While previous researchers have described peak performance or kernel matrix operations, there is as yet little experience with mapping an application-specific floating point pipeline onto FPGAs. In this work, we port a supercomputer application benchmark onto Xilinx Virtex II and II Pro FPGAs and compare performance with comparable microprocessor implementation. Our results show that this application-specific pipeline, with 12 multiply, 10 add/subtract, one divide, and two compare modules of single precision floating point data type, shows speedup of 1.6x – 1.7x. We analyze the trade-offs between hardware and software "sweet spots" to characterize the algorithms that will perform well on current and future FPGA architectures.

1 Introduction

Over the past decade, Re-Configurable Computing (RCC) using Field-Programmable Gate Arrays (FPGAs) has demonstrated speed-ups of one to two orders of magnitude on data- and compute-intensive processing tasks involving fixed point computation on small integers, typically in signal and image processing applications. Floating point computation was not mapped to FPGA due to the large operand size (32- or 64-bit) and excessive area consumed by floating point arithmetic units on configurable logic cells. Recently, that limitation of FPGAs appears to be receding: 3–10 million gate FPGAs with embedded processors, memories, and arithmetic units have become available, making it feasible to consider a broader range of applications than traditional signal and image processing, including those requiring floating point operations. Studies comparing floating point performance of FPGAs vs. high performance microprocessors [1] suggest that peak FPGA floating-point performance is growing significantly faster than peak floating-point performance for a CPU. Other studies ([2], [3]) also suggest that modern FPGAs may be competitive with microprocessors on dense matrix operations such matrix multiply and LU decomposition.

However, it is well-known in the supercomputing community that peak performance and dense matrix kernel operations are far from accurate predictors of realized performance of a complete application. Memory access patterns, cache

behaviour, control flow, and inter-processor communication result in actual performance that is well below peak. For example, applications run on a cluster supercomputer often realize no more than 50–80% of theoretical peak ([4]), reducing a 30 TFLOP machine to 15 TFLOPs.

The purpose of the work described below is to better quantify the performance of FPGA-based floating point computation on real applications by mapping a portion of an application (as opposed to a kernel) onto FPGA. We compare the performance of an application-specific (single precision) floating point pipeline mapped to the Virtex family of FPGA to execution on comparable microprocessors.

Re-Configurable Computing using FPGAs exploits "spatial parallelism", the ability, for example, to unroll a computational block directly onto hardware, executing the entire block in parallel. This ability is not available on a CPU, which depends on fast clock rate to increase performance. FPGAs use a significant amount of spatial parallelism to compensate for having a clock speed that is an order of magnitude slower than that of a CPU.

In this paper we describe our FPGA implementation of a floating-point intensive supercomputing application called "radiative heat transfer" [5]. For our implementation, we chose the FPLibrary, a VHDL library of hardware operators for floating-point (FP) computation, developed in the Arénaire project, at ENS Lyon, France [6]. We describe the floating-point libraries available and why we chose FPLibrary. We also describe other floating-point applications implemented on FPGAs. Next, we give an overview of the radiative heat transfer application. We describe how we parallelized the inner loop of the application, which is the most computationally intensive portion of the program. We present performance results of the computationally intensive inner loop of the application on Intel Xeon 1.76GHz and 3.06GHz workstations and compare that to the performance of our implementation on Xilinx Virtex II [7] and Virtex II Pro [8] FPGAs. Finally, we provide our conclusions and future work.

2 Related Work

Using FPGAs for floating-point operations is not new. Past efforts exploring floating point include exploration by Virigia Tech [9], re-evaluation at Clemson [10] and a library produced at Northeastern [11]. These efforts demonstrate the viability of using floating-point on FPGAs. FPGAs are viable targets because they can be programmed to include many concurrent floating-point operations [1]. Earlier work [12] found that FPGAs were not fast enough to be competitive with general purpose processors for floating point. However, current generations have increased performance with faster logic and embedded multipliers [13]. This increased performance may allow FPGAs to be used for floating-point in areas normally reserved for supercomputers.

FPGAs offer several advantages when used to calculate floating-point operations. First, FPGAs offer a high degree of flexibilty, where they can provide a customized solution for a given floating-point algorithm. Second, due to the

available concurrency, an FPGA can provide a floating-point solution that is faster than what is possible with a general purpose processor. Third, FPGAs are based on SRAM, and thus they track trends in transistors (e.g. "Moore's Law") more closely that general purpose processors. FPGAs take advantage of transistor density to provide high levels of concurrency. Offsetting those advantages are the slow clock speed relative to microprocessors and the relatively large area required by floating point operands and operations, which limits spatial parallelism opportunities.

Several commercial [14, 15] and open source [11, 13] libraries are available for creating floating-point circuits. The FPLibrary [6] was chosen for use with the radiant heat transfer algorithm, because it met three important qualifications. First, it was written VHDL in a platform-independent manner. This allows designs to be easily targeted to different FPGA architectures. Second, the library implements add, multiply and divide floating point operations which are required for this algorithm. Third, the modules and floating-point types have parameterizable bitwidths, so that we can easily program the library for single, double or arbitrary sized floating point types. FPLibrary is used to leverages the advantages of FPGAs to implement the core of a supercomputing application.

3 Description of the Monte Carlo Radiative Heat Transfer Simulation

Monte Carlo radiative heat transfer simulation was chosen for implementation on an FPGA, because it contains computationally intensive floating-point operations. It has been run on a SPARCStation farm [16] as well the Cyber 206 supercomputer [5]. It is a real world problem, because it models the geometry of a laser isotope separation (LIS) unit to accurately determine the radiant exchange factors among the surfaces. This is an important component of the isotope separation process simulation.

The radiative heat transfer simulation is a Monte Carlo application that traces a large number of photons emitted from the surfaces of a 2-D enclosure (Figure 1). The result of the application is a record of how many photons emitted from each surface i were absorbed at surface j. This information is used to compute a heat transfer coefficient between each pair of surfaces, i and j. It is a Monte Carlo application because it uses random values to determine characteristics of an emitted photon's path and because it traces a large number of photons.

In the algorithm, 5000 photons are emitted (with randomly chosen characteristics) from each surface of a 37-sided polygon. The algorithm follows the path of each emitted photon. It identifies the surface of intersection, which is the most computationally intensive portion of the algorithm. Next, a random number determines whether the photon is absorbed into the surface, reflected off of it, or transmitted through it. The photon is followed until it is transmitted, absorbed or lost. This algorithm is designed to calculate intersections assuming

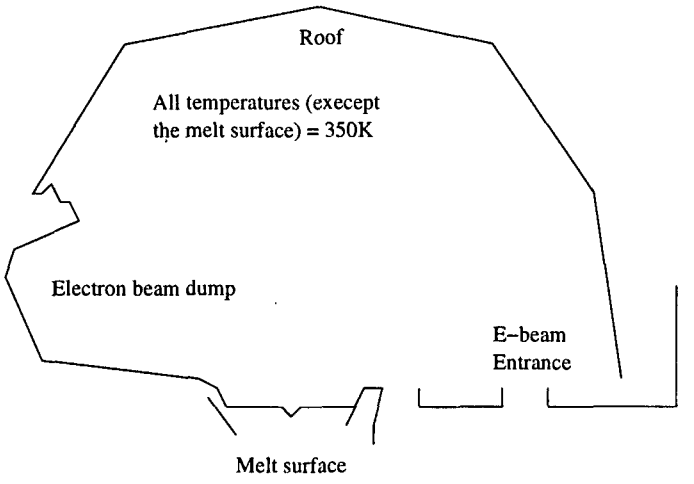

Fig. 1. Test Geometry for Radiative Heat Transfer

a convex chamber. A more sophisticated version which works with both convex and concave surfaces, and is the subject of future work.

The vectorized version of the algorithm parallelizes it at the "task" level. The pseudocode for this algorithm is summarized in Figure 2. A task loops through the 37 surfaces of the polygon and traces the 5000 photons emitted from each surface. For each surface, a **for** loop iterates through each photon emitted, then an inner **while** loop checks if the photon is still active before following it to its next surface intersection. Inside the **while** loop, an inner **for** loop computes the surface intersection, then the random number generator determines interaction with the surface (absorbed, reflected, transmitted or lost).

When considering which part of the algorithm to implement on the FPGA, we decided that parallelism at the task level was too coarse because it would trace 185,000 photons. Implementing surface level parallelism was also too coarse, since it would need to trace 5000 photons emitting from one surface. Neither implementation would fit on currently available FPGAs. At the **while** loop level, tracing one photon's path until it is not active may be possible in terms of fitting on an FPGA, but there are dependencies carried from loop iteration to loop iteration, which makes the implementation more complex and limit parallelism. At the inner **for** loop level, where the algorithm checks for the surface of intersection, the code is straightforward to realize on an FPGA, since the loop iterations are independent of each other and can be spatially replicated on the FPGA.

In addition, this inner **for** loop is the most computationally intensive portion of the program. We used the Portland Group Profiler to measure the percentage of time spent in this inner **for** loop. The profiler shows that it spends 78% of its time executing the inner **for** loop. The C code inside this loop is included in

```
For each surface in the 37-surface polygon
  For each of 5000 photons emitted from this surface
    Emit a photon with random characteristics from this surface
    While the photon is not absorbed, transmitted or lost
      For each side i in the polygon
        If the current side is not the emitted side
          Check if the photon intersected with this side
            (This is implemented on the FPGA)
        End if
      End for
      Randomly determine if the photon is absorbed, transmitted,
            reflected or lost
    End while
  End for
End for
```

Fig. 2. Radiative Heat Transfer algorithm loop structure

Figure 3. All the variables used in the arithmetic computations are floating-point types.

Originally the program was written for double precision floating-point, but for our purposes, we changed that to single precision floating-point. We found that there is not a significant difference in the results when we use single versus double precision floating-point. There was only a .0025% difference in the number of photons absorbed in the single precision version compared to the double precision version.

4 Hardware Implementation

We target the hardware implementation to the Virtex II and Virtex II Pro FPGAs. These devices have small embedded memories called Block RAMs as well as embedded 18-bit multipliers. An initial approximation of the pipeline was generated from the Streams-C compiler [17] on an integer version of the code. The generated pipeline was then converted to use floating point modules, and manually optimized to maximize pipelining.

Figure 3 shows the C code for the compute-intensive **for** loop of the radiative heat transfer algorithm. In each iteration of this loop, the calculations are performed relative to one of the surfaces of the convex shape. Some variables are invariant across loop iterations (e.g., **epsdet0**) while others assume unique values for each loop iteration, as shown by the array index **s**, for example, **delxs**, **delys**, and **rhss**. The latter variables are assumed to reside in Block RAMs.

Figure 4 shows the pipelined hardware implementation of the innermost loop. The design is an 11 stage pipeline utilizing the floating point libraries from [6]. It consists of 12 multiply, 3 addition, 7 subtraction, 1 divide and 2 comparison modules. The breakdown of the latency is as follows: 4 cycles for multiplica-

```
float
x1      [NSM      ] ,x2    ,  [NSM      ] ,
y1      [NSM      ] ,y2      [NSM      ] ,
delx    [NSM      ] ,dely    [NSM      ] ,
sqln    [NSM      ] ,rhs     [NSM      ];

delxs   = delx   [s];
delys   = dely   [s];
rhss    = rhs    [s];

/*  compute intersection points*/
det  = ex*delys - ey*delxs;
absdt= fabs(det);
if(absdt <= epsdet0)
  det=  epsdet0;
dtinv= 1.0/det;
xi     = dtinv * (delxi*rhse - ex*rhss);
yi     = dtinv * (delyi*rhse - ey*rhss);

/*  test for intersection between surface endpoints*/
x1s     = x1     [s];
y1s     = y1     [s];
x2s     = x2     [s];
y2s     = y2     [s];
sqlns   = sqln   [s];
ssq  = (xi - x1s)*(xi - x1s) + (xi - x2s)*(xi - x2s)
  + (yi - y1s)*(yi - y1s) + (yi - y2s)*(yi - y2s);
if(ssq <= sqlns) {
  intersect_side[s] = 1;  /* s is the intersected side */
  else intersect_side[s] = 0;
}
```

Fig. 3. Radiative Heat Transfer code implemented on the FPGA

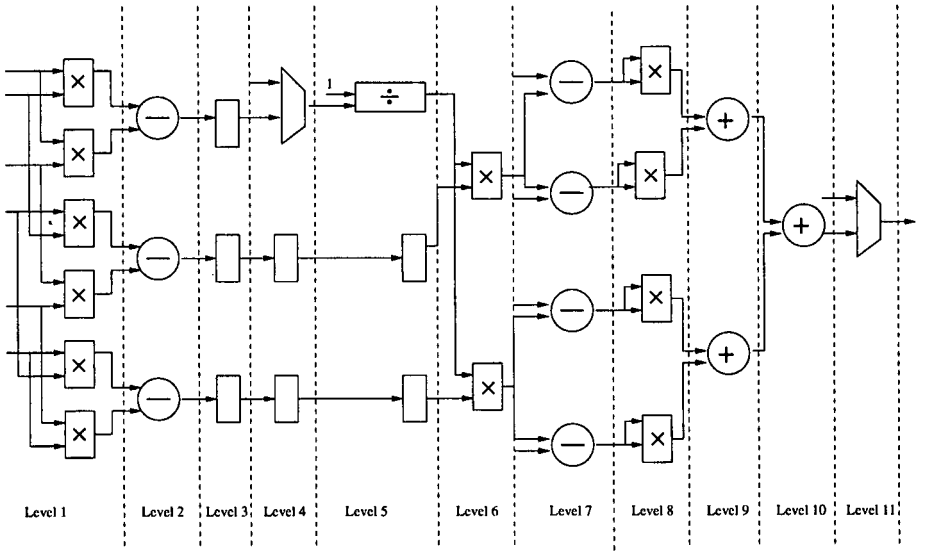

Fig. 4. FPGA Implementation

tion, 3 cycles for addition or subtraction, 15 cycles for division, and 1 cycle for comparison. The total latency of the 11 stage pipeline is 41 cycles. There are 2 intermediate registers that need pipelining from Level 4 through Level 5. This data synchronization requirement introduces 32 additional 34-bit registers into the design.[1] For clarity, only two registers are shown in Level 5; actually there are 15 registers for each operand, for a total of 30 34-bit registers at Level 5;

For this implementation there are 11 inputs to the pipeline – 6 inputs are consumed in the first Level, 4 inputs are consumed at Level 7 and 1 input is consumed at Level 10. The data is stored in 2 204-bit by 512 deep, dual-port Block RAMs. Memory reads are scheduled so that values arrive at Level 7 and at Level 10 at exactly the cycle they are consumed. By scheduling the reads in this way, we avoid the overhead of fully pipelining the 5 inputs that are needed at Level 7 and Level 10. This would introduce an extra 27 cycles $\times$ 4 registers (Level 7) plus 40 cycles $\times$ 1 register (Level 10), or $112 + 40 = 152$ 34-bit registers into the design. These 152 registers correlate to a 1% increase in area utilization on the Virtex II.

5 Performance

This section gives a performance analysis of the application running on several P4 systems versus the the Virtex II and Virtex II Pro hardware platforms.

[1] The FPLibrary adds a 2-bit tag to each floating point register.

5.1 Workstation Performance

For performance comparisons with the FPGA we needed to look at the innermost loop of the task, which is the iteration over 37 surfaces for a single photon, looking for an intersection.

The static instruction count shows 130 instructions: 61 floating point instruction, 9 branches, 73 instructions which reference the stack (including floating load/store to stack for locals), and only one integer instruction (the loop counter). All the instructions and data for this loop fit neatly into the Level 1 cache (the fastest cache level), and hence could be expected to run at maximum speed on the CPU.

Needless to say, this is a very small instruction count, and measurements can easily perturb it. Obtaining an accurate measure of this loop represents a challenge. Traditional profiling tools such as gprof are acceptable for function-level timing, but we needed an extremely accurate measure of the inner loop.

On the Pentium and later processors there is a timer register, called the Time Stamp Counter (TSC), which measures processor ticks at the processor clock rate. This 64-bit read-only counter is extremely accurate, as it is implemented as a Model-Specific Register inside the CPU. The overhead of using this register is extremely low. On a 1.8 Ghz Pentium the TSC runs at 1.8 Ghz and has a resolution of 555 picoseconds; on a 3 Ghz Pentium the TSC has a resolution of 333 picoseconds.

We used the TSC to measure the inner loop of the application. We performed measurements both in the application itself, and by extracting the inner loop and running it many times. As expected for this loop, the performance varied with the CPU being used, with the fastest CPU being the 3 Ghz Pentium.

We tested both the Intel compiler (newest version) and gcc. Gcc provided the best performance, which was somewhat surprising. Timing for one iteration of the inner loop, shown in Figure 5, ranges from 60ns to 104ns. Note that the time is an average, as in the sequential version of the loop body, there is opportunity for early exit from the loop.

	Virtex2 6000	Virtex2p100	Virtex2p125	P4 1.6	P4 2.4	P4 3.0
Speed (ns)	29.9	16.7	15.7	104	74	60
%Area (LUTs)	20	15	12			
%Multiplers	100	32	25			
Latency (cycles)	41	41	41			
Speed up	1.7	2.4	2.6	0.4	0.8	1

Fig. 5. FPGA vs. Workstation performance for Inner Loop. Speed up compared to the P4 3.0 GHz System.

5.2 FPGA Performance

Placement results for one iteration of the inner loop on the Virtex II and Virtex II Pro FPGAs are shown in Figure 5. On the Virtex II 6000, only 20% of the Look Up Tables (LUT) is used by the loop body. However, all the multipliers are used, and therefore only one instance of the loop body can fit on this part. In contrast, the larger Virtex II Pro parts can fit 2–3 instances of the inner loop, resulting in a higher degree of spatial parallelism.

As noted above, the hardware design is highly pipelined. The pipelining allows a relatively high clock frequency for the design. However, the cost is a high latency – 41 clock cycles to seeing the first result. Also, in contrast to the software implementation, in the hardware, the entire loop body is executed, and the timing results reflect the cost of executing all 37 iterations.

In terms of technology generation, the Virtex II 6000 and P4 1.6GHz are comparable. The results show that the V26K hardware implementation outperforms the microprocessor by a factor of 1.67. Encouragingly, for the newer generations of FPGA and microprocessor (V2Pro100 and 3.0GHz) , the speedup is slightly better – 1.73x.

5.3 Discussion

Our results show that the FPGA hardware outperforms comparable generation of microprocessor by 1.6 – 1.7x on an application-specific single-precision floating point pipeline. There are several points to note.

First, the FPGA implementation must execute all loop iterations of the inner **for** loop. The software timing is an average number: many times the software breaks out of the loop without completing all iterations, as the last **if** statement of Figure 3 contains a **break** in the software version of the loop. In a scenario in which all loop iterations were required, the FPGA speedup would be much greater.

Second, this application fits well in the L1 cache of the microprocessor. A more data-intensive application would better use the strengths of the FPGA (greater memory bandwidth and better performance on data intensive computation).

Third, the number of iterations of the **for** loop is quite small. In fact, the pipeline latency is greater than the number of iterations. Like vector processors, the application-specific pipeline on the FPGA shows the best performance when the algorithm has many iterations with little data-dependent branching.

Fourth, the floating point library we used in this experiment is technology-independent. In fact we were able to synthesize it to several different families, including the Altera Stratix. Technology-specific floating point cores such as Quixilica yield smaller area and faster clock rate. On the minus side, other floating point libraries, including Quixilica, have even higher operation latencies. For best performance, embedded hard floating point units in a fabric of reconfigurable logic would, of course, be desirable.

Finally, it is important to compare the performance of the application-specific pipeline, with a mix of different floating point operators and branching constructs, to peak performance results cited by others. While theoretical peak numbers are useful to gauge feasibility, using floating point in a supercomputing application gives us more accurate performance results – which, of course, are much reduced from theoretical peak.

6 Conclusions

We have presented hardware implementation of a floating point Monte Carlo radiative heat transfer simulation application on the Virtex II and Virtex II Pro families of FPGA. In contrast to previous work that presented peak performance or performance results on small kernels, we have implemented an application-specific pipeline on the FPGA. We have presented detailed timing results comparing FPGA speed to high performance workstations, realizing a 1.73x speed up of the single precision floating point pipeline running on a VirtexII Pro hardware platform versus running the application on a 3.0GHz workstation.

7 Acknowledgements

We would like to thank Jeremie Detrey for his open source floating point library and his help with simulation and synthesis of the library modules. We thank Sung-eun Choi for her assistance in obtaining workstation timings.

References

1. K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," in *ACM/SIGDA Twelfth ACM International Symposium on Field-Programmable Gate Arrays (FPGA 2004)*, 2004.
2. Seventh Annual Workshop on High Performance Embedded Computing (HPEC 2003), *Area and Power Performance Analysis of Floating-point based Application on FPGAs*, (Lexington, MA), September 2003.
3. S. Choi and V. Prasanna, "Time and energy efficient matrix factorization using fpgas," in *FPL 03: 13th International Conference on Field Programmable Logic and Applications*, Sept. 2003.
4. Top 500, "Top 500 supercomputer sites." http://www.top500.org, 2004.
5. P. J. Burns and D. V. Pryor, "Vector and parallel monte carlo radiative heat transfer simulation," *Numerical Heat Transfer*, vol. 16, 1989.
6. J. Detrey and F. de Dinechin, "FPLibrary, a VHDL library of parametrisable floating-point and LNS operators for FPGA." http://perso.ens-lyon.fr/jeremie.detrey/FPLibrary/, 2004.
7. Xilinx Corporation, "Virtex II platform FPGAs: Introduction and overview," *http://www.xilinx.com*, 2003.
8. Xilinx Corporation, "Virtex II Pro platform FPGAs: Introduction and overview," *http://www.xilinx.com.*

9. N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arihtmetic of FPGA based custom computing machines," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, (Napa, CA), pp. 155–162, IEEE Computer Society Press, 1995.
10. Walter B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood, "A re-evaluation of the practicality of floatin-point operations on fpgas," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, (Napa, CA), pp. 206–215, IEEE Computer Society Press, April 1998.
11. P. Belanović and M. Leeser, "A library of parameterized floating-point modules and their use," in *FPL 2002: The 12th International Conference on Field-Programmable Logic and Applications*, pp. 657–666, Springer-Verlag, 2002.
12. CAINE, *Feasibility of Floating-Point Arithmetic in FPGA based artificail Neural Networks*, (San Diego, CA), Nov. 2002.
13. E. Roesler and B. Nelson, "Novel optimizations for hardware floating-point units," in *FPL 2002: The 12th International Conference on Field-Programmable Logic and Applications*, pp. 637–646, Springer-Verlag, 2002.
14. QinetiQ Holdings Ltd., "Real time systems lab." http://www.quixilica.com/products.htm, 2002.
15. Nallatech, "Floating point IP cores for virtex-II." http://www.nallatech.com/solutions/products/software_fpga_ip/fpga_ip/fpc/, 2003.
16. R. Minnich and D. V. Pryor, "A radiative heat transfer simulation on a SPARC-Station farm," in *First International Symposium on High Performance Distributed Computing (HPDC '92)*, 1992.
17. M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, (Napa, CA), p. n/a, IEEE, 2000.