

Monte Carlo Radiative Heat Transfer Simulation on a Reconfigurable Computer

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

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 relatively little experience with mapping an application-specific floating point loop onto FPGAs. In this work, we port a supercomputer application benchmark onto Xilinx Virtex II and Virtex II Pro FPGAs and compare performance with three Pentium IV Xeon microprocessors. 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 speed up of $10.37\times$. We analyze the trade-offs between hardware and software to characterize the algorithms that will perform well on current and future FPGA architectures.

1 Introduction

Over the past decade, Reconfigurable 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 FPGAs 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 as 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

behavior, 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 an FPGA. We compare the performance of an application-specific (single precision) floating point pipeline mapped to the Virtex family of FPGAs to execution on comparable microprocessors.

Reconfigurable 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 a 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]. First, we describe other floating-point applications implemented on FPGAs and discuss floating-point libraries for 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 inner loop on three Intel Pentium IV Xeon workstations and compare that to the performance of our implementation on Xilinx Virtex II and Virtex II Pro FPGAs. Finally, we provide our conclusions.

2 Related Work

Using FPGAs for floating-point operations is not new. Past efforts exploring floating point include exploration by Virginia Tech[6], a re-evaluation at Clemson[7] and a library produced at Northeastern[8]. 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 [9] 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 [10]. 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 flexibility, 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 a general purpose processor. Third, FPGAs are based on SRAM, and thus they track trends in transistors (e.g. “Moore’s Law”) more closely than 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 [11,12] and open source [8,10] libraries are available for creating floating-point circuits. For our implementation of the radiative heat transfer algorithm, we chose the FPLibrary, a VHDL library of hardware operators for floating-point computation, developed by the Arénaire project [13]. The FPLibrary meets three important qualifications. First, it is written in 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 leverage 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 computer cluster [14] as well the Cyber 205 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.

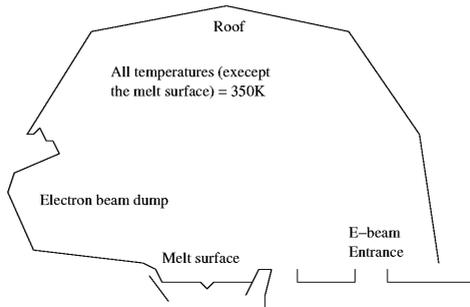


Fig. 1. Test Geometry for Radiative Heat Transfer

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 simulation records 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, N photons are emitted (with randomly chosen characteristics) from each surface of an m -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 a convex chamber. There is also a more sophisticated version which works with both convex and concave surfaces, and is the subject of future work.

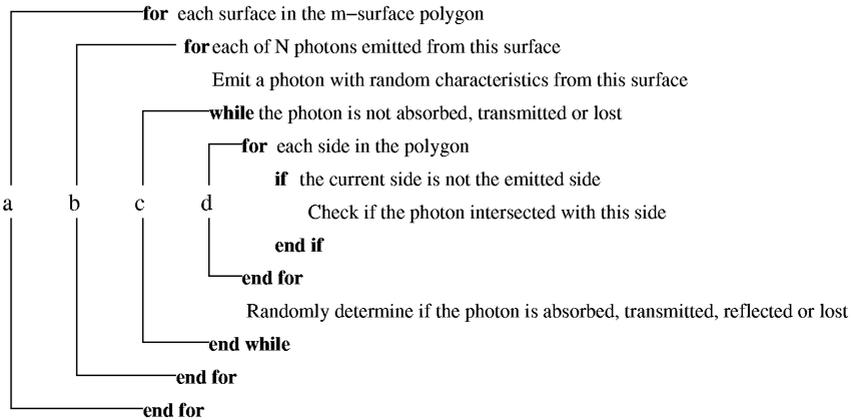


Fig. 2. Radiative Heat Transfer algorithm loop structure. Loop “d” is implemented on the FPGA.

The parallel version of the algorithm distributes at the “task” level. The pseudo-code for each task is summarized in Figure 2. In loop “a”, a task iterates through the m surfaces of the polygon and traces the N photons emitted from each surface. For each surface, a **for** loop (“b”) iterates through each photon emitted, then an inner **while** loop (“c”) checks if the photon is still active before following it to its next surface intersection. Inside the **while** loop, an inner **for** loop (“d”) computes the surface intersection, then the random number generator determines if the photon is absorbed, reflected, transmitted or lost.

When considering which part of the algorithm to implement on the FPGA, we decided that parallelism at the task or surface level was too coarse, and would not 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 between loop iterations that make 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.

```

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];
x1s = x1[s];  y1s = y1[s];  x2s = x2[s];  y2s = y2[s];  sqlns = sqln[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*/
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; /* break here in the software version */
}

```

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

In addition, this inner **for** loop is the most computationally intensive portion of the program. Using a timer described in Section 5.1, with $\mathbf{N}=5000$ and $\mathbf{m}=37$, we found that a Pentium IV Xeon 3 GHz workstation spends 86% of the algorithm time executing the inner **for** loop. The C code inside this loop is included in Figure 3. All the variables used in the arithmetic computations are floating-point.

Originally the program was written for double precision floating-point. In this work, we evaluate single and double precision floating-point. We found that there is not a significant difference in the scientific results from the algorithm when using single versus double precision. The number of photons absorbed differed by only .0025% in the single precision version as 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 [15] 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 loop. The design is an 11 stage pipeline utilizing the floating point libraries from [13]. It consists of 12 multiply, 3 addition, 7 subtraction, 1 divide and 2 comparison

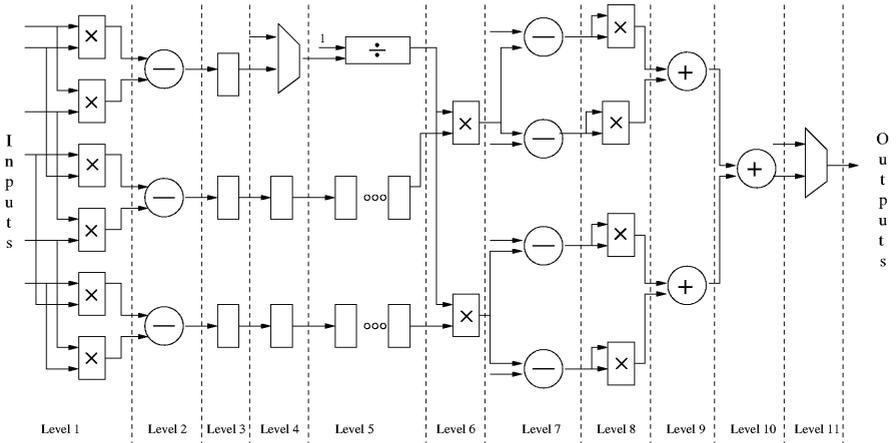


Fig. 4. FPGA Implementation

modules. The breakdown of the latency is as follows: 4 cycles for multiplication, 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 in the Figure 4 in Level 5, but there are 15 registers for each operand, for a total of thirty 34-bit registers at Level 5.

For this implementation there are eleven inputs to the pipeline – six inputs are consumed in Level 1, four at Level 7 and one at Level 10. The data is stored in two 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. The latter approach introduces an extra $27 \text{ cycles} \times 4 \text{ registers (Level 7)} + 40 \text{ cycles} \times 1 \text{ register (Level 10)}$, 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 analyzes the performance of the application running on several Pentium IV Xeon (P4) systems versus the Virtex II (V2-6k) and Virtex II Pro (V2p100,V2p125)² hardware platforms. Note that the V2p125 is not yet available.

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

² The V2-6k is speed grade -4 and the V2p100 and V2p124 are speed grade -6.

5.1 Workstation Performance

For performance comparisons with the FPGA we examined the innermost loop of the application, which is the iteration over \mathbf{m} surfaces for a single photon, searching for an intersection. The static instruction count of this loop count is 130 instructions: 61 floating point instructions, 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 into the Level 1 cache (the fastest cache level), and hence could be expected to run at maximum speed on the CPU.

Timing measurements of the inner loop are easily perturbed due to the small instruction count of 130 instructions. Obtaining an accurate measure of this loop represents a challenge, since traditional profiling tools such as gprof are only acceptable for function-level timing, and do not provide an extremely accurate measure of the inner loop. However, 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 1.7 GHz P4 the TSC runs at 1.68 GHz and has a resolution of 595 picoseconds; on a 3 GHz P4 the TSC has a resolution of 333 picoseconds.

We used the TSC to measure the inner loop of the application. C code was added using the gcc asm statement, which produces inline assembly code to read the TSC at the start and end of the loop code. 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 P4.

We tested both the Intel compiler v7.0 and gcc v3.2. The gcc compiler provided the best performance results with `-O3` optimization level. Timing for one iteration of the inner loop, shown in Figure 5, ranges from 60ns to 104ns. It is important to note that the time is an average, since in the sequential version of the loop body, there is opportunity for early exit from the loop.

5.2 FPGA Performance

Synplicity was used to synthesize the inner loop to Xilinx FPGAs. Placement and area results were obtained using Xilinx ISE 6.1. The results for one iteration of the inner loop on the Virtex II and Virtex II Pro FPGAs are shown in Figure 5. On the V2-6k, only 20% of the Look Up Tables (LUT) are 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 three pipelines of the inner loop, resulting in a higher degree of spatial parallelism. The speed up row calculates the speed up relative to the 3 GHz P4. The hardware calculation assumes a steady state pipeline in which a result is delivered every clock cycle. With three pipelines three results are delivered every clock cycle, effectively reduce the execution time by one third. These results do not include the time to write the parameters into Block RAM.

	V2-6k	V2p100			V2p125			P4 1.7	P4 2.4	P4 3
# Pipelines	1	1	2	3	1	2	3			
Execution Time (ns)	29.9	16.7	7.89	5.78	15.7	7.72	6.12	104	74	60
%Area (LUTs)	20	15	33	50	12	26	40			
%Multipliers	100	32	64	97	25	51	77			
Latency (cycles)	41	41	41	41	41	41	41			
Speed up	2.01	3.59	7.61	10.37	3.82	7.77	9.81	0.58	0.81	1

Fig. 5. FPGA vs. Workstation performance for Inner Loop. Speed up compared to the P4 3 GHz System.

In terms of technology generation, the V2-6k and P4 1.7 GHz are comparable. The V2-6k hardware implementation outperforms the 1.7 GHz Pentium by a factor of 3.48. For the newer generations of FPGA and microprocessor (V2p100 and 3 GHz), the single pipeline speed up is slightly better – $3.59\times$. However, with this newer generation Virtex Pro it is possible to fit three pipelines on the V2p100 which allows a speed up of 10.37.

As noted above, the hardware design is highly pipelined. The pipelining allows a relatively high clock frequency for the design, at the cost of high latency – 41 clock cycles before the first result appears. For a large number of surfaces, the effect of pipeline latency diminishes. For example, with 10,000 surfaces the speedup for three pipelines is $10.25\times$. 150,000 surfaces are desirable for this particular simulation, so the pipeline latency effect is negligible.

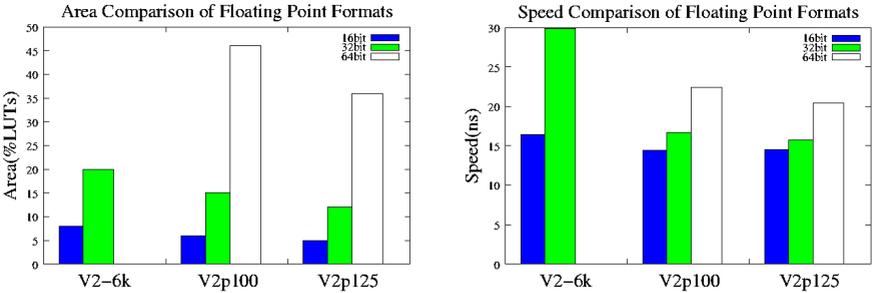


Fig. 6. Placement results for a single pipeline 16, 32, and 64-bit implementations of the inner loop.

Lastly, if we analyze the granularity of the input data width as shown in Figure 6, the placement results show that for a 16-bit word width, the area utilization across the Virtex Family is 5% to 8% which allows 10 to 20 instantiations of the inner loop to run concurrently on the FPGA. For larger bit widths, fewer parallel versions of the loop can fit onto hardware, for example with 32-bits 3 pipelines fit. As expected, the run-time clock speeds are faster for smaller bit widths. The results show that on the Virtex II Pro family, 32-bit operations are only slightly more expensive than 16-bit, while 64-bit incur a much higher penalty both in area and clock speed. As the graphs show, the 64-bit version of the application does not fit on the V2-6k.

5.3 Discussion

Our results show that the FPGA hardware outperforms a comparable generation of microprocessor by up to $10.37\times$ 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. If all loop iterations were executed, the FPGA speed up 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 tractability of an application kernel to pipelining, especially long pipelines, is crucial to get performance. The highest performance floating point operators are heavily pipelined, so there is substantial cost in starting up and breaking up the pipeline. Like vector processors, the application-specific pipeline on the FPGA shows the best performance when the algorithm has many iterations with minimal data-dependent branching. In this application, the vector length is very large, and thus the latency is negligible. This application also has the advantage of little data-dependent branching. Although predication can be used to reduce the impact of branching, area costs increase by having both the **then** and **else** bodies instantiated on the chip.

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, a floating point intensive supercomputing application gives us more accurate performance results.

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 $10.37\times$ speed up with three single precision floating point pipelines running on a Virtex II Pro hardware platform versus running the application on a 3 GHz workstation.

Acknowledgments. Thanks to Jérémie Detrey for his open source floating point library and his help with simulation and synthesis of the library modules, and to 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. N. Shirazi, A. Walters, and P. Athanas, “Quantitative analysis of floating point arithmetic 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.
7. Walter B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood, “A re-evaluation of the practicality of floating-point operations on fpgas,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, (Napa, CA), pp. 206–215, IEEE Computer Society Press, April 1998.
8. 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.
9. K. R. Nichols, M. A. Moussa, and S. M. Areibi, “Feasibility of floating-point arithmetic in FPGA based artificial neural networks,” CAINE02, Nov. 2002.
10. 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.
11. QinetiQ Holdings Ltd., “Real time systems lab.” <http://www.quixelica.com/products.htm>, 2002.
12. Nallatech, “Floating point IP cores for virtex-II.” http://www.nallatech.com/solutions/products/software_fpga_ip/fpga_ip/fpc/, 2003.
13. J. Detrey and F. de Dinechin, “FPLibrary, a VHDL library of parametrizable floating-point and LNS operators for FPGA.” <http://perso.ens-lyon.fr/jeremie.detrey/FPLibrary/>, 2004.
14. 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.
15. 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), 2000.