

Using FPGAs to Supplement Ray-Tracing Computations on the Cray XD-1

Charles B. Cameron

Department of Electrical Engineering, US Naval Academy (USNA), Annapolis, MD
cameronc@usna.edu

Abstract

Optical ray tracing simulations in lens design commonly employ six major computations: the point of intersection of a ray and an optical surface, a check to see whether the ray is inside a restrictive aperture, calculation of a unit vector normal to the surface at the point of intersection, the result of reflection or refraction of the ray at the surface, and coordinate conversions of the new ray's starting position and direction to facilitate repetition of the calculations at succeeding optical elements. Because the rays are independent of one another, ray tracing can benefit greatly from parallel processing, especially when there are billions of rays to be traced. The efficiency of using 839 AMD Opteron processors for this application has been shown to be 97.9%. This means that adding additional processors is a highly effective strategy for increasing the rate of ray tracing, thus reducing the time of simulation. Using field-programmable gate arrays (FPGA) is an effective strategy for further increasing the rate of ray tracing.

In this paper we describe key aspects of implementing deeply pipelined processors within an FPGA to perform scientific computations, using them to supplement the ray tracing provided by the sequential Opteron processors in the Cray XD-1. The discussion includes how to schedule the use of the FPGA's internal resources, synchronize the interaction between the FPGA and the Opterons, and assess the fraction of time that each major computational resource within the FPGA is in use. We discuss a method that guarantees 100% efficiency of the critical resource, the resource needed most often for a specific computation. We also consider how to increase the efficiency of the non-critical computational resources within the FPGA and what the side effects of such changes can be.

1. The Ray-Tracing Problem

It is common both in lens design as well as in computer-generation of digital images to rely on the formulations of ray tracing to describe the propagation of

light. When ray tracing theory is applied to lens design, a typical objective is to discover where each ray strikes the image plane, if it does so at all. The lens systems designer can assess the acceptability of the result, modify the lens design if needed, and repeat the ray-tracing simulation.

There are six main computational tasks in this kind of ray tracing that must be performed repetitively, once for each ray as it encounters each surface in the system. For each optical surface within a lens system, we begin with a starting point and the initial direction of a particular ray. We then need to perform the following steps.

1. Find the point where the ray strikes the next optical surface.
2. Check to see whether the ray has fallen outside or inside an aperture associated with that optical surface.
3. Find the unit vector normal to the surface at the intersection point.
4. Compute the direction of the new ray (or rays) resulting from reflection or refraction when the old ray strikes this optical surface.
5. Convert the intersection point from the local coordinate system associated with the previous optical surface into that associated with the next optical surface. This entails a rotation and a translation in three dimensions.
6. Convert the direction of the new ray from the local coordinate system associated with the previous optical surface into that associated with the next optical surface. This just entails a rotation in three dimensions.

The four broad classes of surfaces encountered in typical imaging lens systems are planes, spheres, conicoids, and aspherics. While planes and spheres can be described as special cases of conicoids, the problem of determining where a ray strikes one of these surfaces can be accomplished more quickly by taking advantage of their simpler mathematical descriptions. In this paper we do not treat aspheric surfaces.

Figure 1 is a schematic drawing of a particular lens system we have used in exploring the application of parallel processing techniques to the problem of ray

tracing. It is the Moderate Resolution Imaging Spectroradiometer (MODIS), currently in the two, low-orbit, earth-observing satellites Terra and Aqua^[1]. This system incorporates a pinhole attenuator that limits the amount of solar power that enters the system during the calibration of the instrument once each orbital period, which is about 90 minutes in length. It also includes a solar diffuser, likewise used only during calibration. Its effect is to spread the solar illumination evenly on the image plane to permit the determination of a suitable level of amplification in each of the optical detectors in the image plane. This is necessary to offset variability in the earth's distance from the sun.

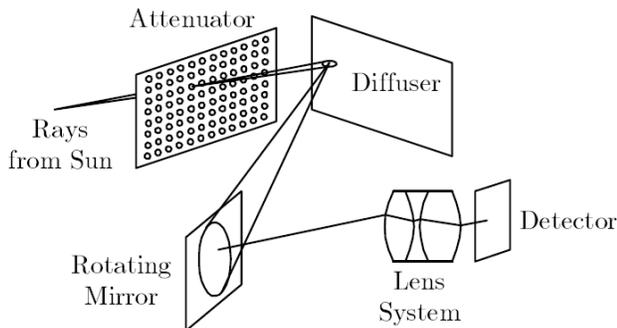


Figure 1. The Moderate Resolution Imaging Spectroradiometer performs a calibration once during every orbit. This entails attenuating the sun's direct rays with a pinhole attenuator. Multiple diffuse images of the sun then fall upon a diffuser, producing more or less even illumination on the detector and permitting adjustment of the gain of each detector element. Views of the earth use neither the pinhole attenuator nor the diffuser.

A single ray in optical ray tracing simulations of this system generates a very large number of rays when it strikes the solar diffuser. This tends to make the simulations very lengthy, about two weeks long when performed on a Digital Equipment Corporation Alpha 3000 series model 800 computer^[2].

By performing the simulations on a parallel computer system, the Naval Research Laboratory's (NRL's) Cray XD-1 computer, we managed to reduce their duration to less than 27 s. Figure 2 shows the rate of ray tracing as a function of the number of processors cooperating in the simulation. The straight line shows that adding more processors is very effective in achieving improved performance. The program uses the Message Passing Interface (MPI) to coordinate the efforts of the multiple processors assigned to it.

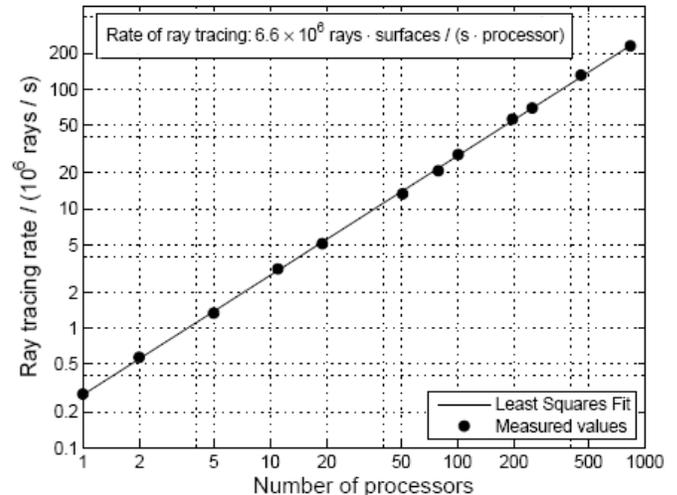


Figure 2. Ray tracing benefits greatly from the presence of additional processors because individual rays are independent of each other

The Cray XD-1 groups its processors into collections of four processors, each group known collectively as a single node. Of the 840 available nodes in the NRL Cray XD-1, 576 of them contain a Field Programmable Gate Array (FPGA). The data shown in Figure 2 were collected without using any FPGAs at all. We are interested in using the FPGAs to perform a subset of the ray-tracing computations in order to trace a higher number of rays per unit time.

Kumar et al.^[3] define the efficiency of a parallel system by

$$E = \frac{r_p}{nr_{np}}$$

where r_p is the rate of execution when a single processor tackles the problem, n is the number of processors actually assigned to work cooperatively on the problem, and nr_{np} is the rate of execution when n processors work on it. Whenever adding another processor actually gives a good return on the investment, E approaches 100%. Whenever adding another processor yields little improvement in performance, E approaches 0%. By this measure of efficiency, we achieved $E = 97.9\%$ with $n = 839$.

It would be hard to achieve much greater efficiency than this. However, it is quite feasible to consider improving both r_p and r_{np} together and, because the efficiency E is so high, it is also quite reasonable to consider assigning still more processors to the problem in order to achieve faster execution.

Modern FPGAs such as those in the Cray XD-1 contain a large number of hardware devices. Their interconnections can be changed on the fly, making it feasible to alter the hardware design inside an FPGA

partway through the completion of a processing job. This provides another avenue for improving the performance of a parallel task such as ray tracing. By offloading some of the computational work from the sequential processors of the Cray XD-1 into the inherently parallel hardware of the FPGA, we expect to achieve significant gains in processing speed without requiring any additional sequential processors to be included.

2. Depth-First vs. Breadth-First Ray Tracing

It is reasonable in a sequential ray-tracing program to select a single ray and trace it from beginning to end. This requires repeated calculations of intercept point, aperture, unit normal vector, interaction, and coordinate transformation as the ray transits the system, as described earlier. Such an approach could be described as a depth-first simulation: one ray is followed throughout its history as it proceeds deeper and deeper into the optical system. Because programs are stored in random-access memory, it is equally costly to access any part of the program at any time (although cache systems can affect the costs in ways that are hard to predict.)

In contrast, appreciable overhead is incurred when the hardware design within an FPGA is replaced with a different design. The space within an FPGA is limited, so it is not feasible for the FPGA to hold a large enough hardware design to perform all the calculations it needs to do. In a ray-tracing system the hardware design loaded into the FPGA needs to be replaced repeatedly as the ray advances, in order to accomplish different phases of the computation. In order to minimize the effect of this overhead on the performance of the system, it is preferable to do a breadth-first tracing of the rays. This means that rather than tracing one ray at a time through the whole system, it is preferable instead to trace a large collection of rays through a single stage of the system and *then* change the FPGA's function by loading a different design into it.

In the calculation of the point at which a ray intersects an optical surface, this would entail loading the intersection-calculating hardware design into the FPGA and then calculating the intersection point for a large number of rays at the next optical surface. (A different design would apply according to whether the surface was a plane, sphere, or general conicoid.)

Next, determine if another hardware design is needed to check if the rays are inside or outside the aperture associated with the surface.

The system would need to calculate the unit normal to the optical surface. In the case of a planar surface, this step is unnecessary because the unit normal vector is constant throughout the plane and known in advance.

The fourth phase would entail calculating the effect of refraction or reflection at the surface. Again, this would be done for a large number of rays. Since it is known in advance whether a particular surface is a reflector or a refractor, it is straightforward to load whichever design is appropriate.

Finally, the coordinates that specify the position and direction of the rays in the local coordinate system for the next surface would be calculated by loading two further hardware designs into the FPGA one at a time and converting the coordinates for a large number of rays.

3. Modulo Scheduling of FPGA Resources

Rau and Glaeser^[4] described an optimal means of scheduling generalized vector computations, of which ray tracing is an example. Because modern FPGAs can hold a large amount of hardware, it now is feasible to apply this method to the scheduling of deeply pipelined scientific computations in FPGAs. To do this we must make a suitable specification of resources and their interconnections within an FPGA.

As an example of this, we outline three approaches to scheduling the use of resources in pipelined hardware that compute the intersection of a light ray with a conicoid surface. Of the six main computations in the ray-tracing application, this calculation is the most involved. Such schedules are suitable in designs based on the floating-point hardware modules available, for example, with the Xilinx ISE software suite^[5]. These pipelined modules permit a new floating-point operation to be initiated every cycle, even though the latency of any single calculation may be many cycles.

Four equations are sufficient for calculating the distance u a ray must traverse before striking a conicoid surface^[6]:

$$\begin{aligned} f &= c(1 + kN^2) \\ g &= N - c(x_0L + y_0M + (1+k)z_0N) \\ h &= c(x_0^2 + y_0^2 + (1+k)z_0^2) - 2z_0 \\ u &= \frac{h}{g + \sqrt{g^2 - fh}} \end{aligned}$$

where the starting point of the ray is $\mathbf{P}_0 = (x_0, y_0, z_0)$; its starting direction is given by the unit vector $\hat{\omega}_0 = (L, M, N)$; k is the conic constant that specifies whether the conicoid is a sphere ($k = 0$), an oblate ellipsoid ($k > 0$), a prolate ellipsoid ($-1 < k < 0$), a paraboloid ($k = -1$), or a hyperboloid ($k < -1$); and c is the curvature of the conicoid surface.

Once u is known, the coordinates of the intersection point $P_1 = (x_1, y_1, z_1)$ can be calculated using the vector expression $P_1 = P_0 + u\hat{\omega}_0$.

These calculations entail 19 multiplications, 13 additions, a division, and the extraction of one square-root.

In considering our first schedule we include four floating-point units: an adder, a multiplier, a divider, and a square-root extractor. The latency of these units—the number of cycles it takes for the result to be calculated—is 11, 6, 27, and 27, respectively.

When performing modulo scheduling, the modulus is equal to the highest number of operations required of any functional unit. In the first design we discuss below, this number is 19. Using the chain of dependences of one calculation on the results of preceding calculations, the cardinal rule is to make sure that the starting cycle taken modulo 19 does not correspond to any cycle (also taken modulo 19) when that unit has already been scheduled. In any case where this condition is violated, we add enough delays to force it to be met. These delays correspond to the insertion of buffers in the pipeline to hold the result from the time it is available until the time it is needed. To determine the schedule, we start with the last calculation performed and work backwards to the earliest calculation. Once the pipeline is full, a new result appears after every interval $t = m\tau$, where m is the modulus and τ is the clock period. With large numbers of rays available, the pipeline is full most of the time.

The multiplier is the critical unit in the first schedule we consider because there are more multiplications than any other operation. Modulo scheduling permits 100% utilization of the critical unit. The utilization of the adder/subtractor in this example is $13/19 = 68\%$, that of the divider and the square-root extractor is $1/19 = 5\%$. The critical unit also determines the throughput. One computation is completed every 19 cycles in this case.

Consider the result if we add a second multiplier to the system and arrange for one of the multipliers to do 10 multiplications and for the other to do the remaining nine multiplications. This will cause the adder/subtractor to become the critical unit because it will have 13 additions to do, more than any other unit. A new suitable schedule will yield 100% utilization for the adder. The utilization of the two multipliers will be $10/13 = 77\%$ and $9/13 = 69\%$, respectively; that of the divider and the square-root extractor will climb to $1/13 = 8\%$. The throughput now will be one computation every 13 cycles. This will represent an increase in throughput of 46% due to the introduction of a second multiplier unit.

If we could include both a second adder as well as a second multiplier, we could schedule seven additions on one adder and six on the other. The multiplier with 10 multiplications scheduled on it would then become the critical unit. The utilization of the multiplier with nine

scheduled multiplications would be $9/10 = 90\%$. The two adders would have utilization $7/10 = 70\%$ and $6/10 = 60\%$, respectively. The utilization of the divider and of the square-root extractor would climb to $1/10 = 10\%$. The throughput now would be one result every 10 cycles, an improvement of 90% over the first design and of 30% over the second design.

4. Interaction Between the Opteron and the FPGAs

There are 220 nodes of four Opteron processors within NRL's Cray XD-1. There is one Xilinx Virtex II Pro FPGA in each of 144 of these nodes, making a total of 576 of this type of FPGA. In a single node, any one of the four Opteron processors can take control of the FPGA at a time.

In our design a single node can communicate with any other node using MPI. Each node runs an identical program, using OpenMP to implement multithreading. The `main()` program uses MPI to discover the set of ray-tracing tasks for which it has responsibility and establishes a queue of such tasks. The `main()` program also controls the node's FPGA. One of the threads running in parallel with `main()` loads an appropriate hardware design into the FPGA whenever needed. It also repeatedly selects a subset of the waiting ray-tracing computations and dispatches that subset to the FPGA for processing. This thread is then suspended until the FPGA has finished the assigned task. At this point the FPGA interrupts the program, reactivating the dormant thread.

Meanwhile, additional threads spawned and dispatched by `main()` using the capabilities of OpenMP execute on one of the four sequential instruction processors available in the node. Just as the thread controlling the FPGA does, these threads also select waiting ray-tracing computations, performing them with a sequential program.

The effect of these combined operations is that the FPGA supplements the processing power of the Opteron processors, accelerating the ray-tracing process.

Each node of four sequential Opteron processors operates largely independently of the others. One of the nodes is designated as the master node. It is responsible for reading the lens-description file at the outset and broadcasting the description to all the other nodes sharing in the work.

The sequential threads use a depth-first ray-tracing strategy. However, in order to minimize the overhead incurred when a node replaces its FPGA design, the FPGA thread forces the FPGA to trace rays using a breadth-first approach.

Once each node has finished its assigned tasks, it reports its results independently to the designated master

processor, which collates the results as each node completes its assigned ray-tracing tasks.

5. Simulated Performance

Preliminary simulations show that one floating-point operation can be initiated every 4.931 ns. The corresponding average ray-tracing rate is 5.350×10^6 rays-surfaces/s in the MODIS system. This is somewhat less than the 6.95×10^6 rays surfaces/s we obtained using the Cray XD-1 alone^[7]. Our experience shows, though, that the overhead associated with ray tracing is very small. Using the assumption that the overhead associated with assigning work to the FPGA also is negligible in comparison to the computational work required, we can make a rough estimate of the combined performance by adding the two rates together, leading to a ray-tracing rate of 11.88×10^6 rays·surfaces/s. This represents a 77 % speedup.

We can achieve still better performance, however, by increasing the number of adders and multipliers in the system. In a system with one adder, one multiplier, one divider, and one square-root extractor, these devices use up less than 11% of the “slices” in a Xilinx Virtex II Pro Model 2vp50ff1148-7 FPGA. Adding two adders and three multipliers (a 125% increase in floating-point units on the chip) would greatly increase the speed at which the ray tracing could proceed without exhausting the on-chip resources. Our prediction is that such a system would yield a 267 % speedup compared to one that uses the Opteron processors alone. There is a possibility of even greater improvements if we can fit additional floating-point units on the FPGA.

We are currently engaged in completing the combined design, which includes a view to experimentally determine the actual speedup we can achieve.

Acknowledgements

This work was supported by the National Aeronautics and Space Administration Goddard Space Flight Center (Code 586), the Applied Optics Branch (Code 5630) at the NRL, the United States Naval Academy, and a grant of computer time from the Department of Defense High Performance Computing Modernization Program at NRL (Code 5593). The author would also like to thank Xilinx, Inc. for their generous donation of development software used in the research described in this paper.

References

1. Waluschka, E., X. Xiong, B. Guenther, W. Barnes, and V. Salomonson, “Modeling studies for the MODIS solar diffuser attenuation screen and comparison with on-orbit measurements.” *Proceedings of SPIE*, vol. 5542, no. 47, pp. 342–353, 2004.
2. Cameron, C. B., R.N. Rodriguez, N. Padgett, E. Waluschka, S. Kizhner, G. Colon, and C. Weeks, “Fast optical ray tracing using parallel DSPs.” *IEEE Transactions on Instrumentation and Measurement*, vol. 55, no. 3, pp. 801–808, June 2006.
3. Kumar, V., *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms*, Redwood City, CA: Benjamin/Cummings Publishing Company, Inc., 1994.
4. Rau, B.R. and C.D. Glaeser, “Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing.” *SIGMICRO NewsL.*, vol. 12, no. 4, pp. 183–198, 1981.
5. *Floating-Point Operator V3.0*, Available online at http://www.xilinx.com/bvdocs/ipcenter/data_sheet/floating_point_ds335.pdf, Xilinx, September 28 2006, product specification DS335.
6. Mouroulis, P. and J. Macdonald, *Geometrical Optics and Optical Design*, New York, New York: Oxford University Press, Inc., 1997.
7. Cameron, C.B., “Parallel ray tracing using the message passing interface (MPI).” 2006, submitted for publication to *IEEE Transactions on Instrumentation and Measurement* on 19 September 2006.