CMSC 22200 November 28, 2011

cudatrace — A parallel ray tracer written in CUDA/C

Dominick LoBraico and Laura Macaddino

Our application, cudatrace, is the conversion of a sequentially implemented ray tracer to the CUDA general purpose graphics processing unit programming paradigm. While CUDA dictated that changes we made be more significant than simple in-place modifications, since adopting a more readily parallel structure, our programs performance has flourished.

In the parallel version of the ray tracer, the source of parallelism comes from the fact that each thread in each block in each grid will have the ability to simultaneously trace rays. In that manner, each thread would be working on tracing rays in parallel.

Applications written in CUDA execute code on the graphics processing unit (GPU) by initializing so-called CUDA kernels. Within each kernel is a one, two, or three dimensional matrix of blocks. Each of those blocks contains a similarly-dimensioned group of threads, each executing the code referenced by the kernel call. When many threads are created in the same block, they execute simultaneously, therein serving as the root of parallelism in CUDA programming.

One of the major changes we had to implement was caused by the fact that CUDA does not support recursive function calling. In the original sequential version of the ray tracer, reflection rays are traced recursively. In that manner, we had to ensure that reflection rays could not be traced until the ray currently being traced has finished tracing. In order for the program to know when it should continue tracing reflection rays, we created a variable which would be affirmatively set if a given ray being traced needed to have a reflection ray traced as well. If this variable was affirmatively set, another ray would be traced using the arguments saved in the previous trace. Finally, the color data obtained from the output of all of the traces would be summed after there are no more rays to be traced.

We also found that it was not convenient to incorporate the linked lists found in the sequential version. In the sequential version, a linked list is used to store the objects in a scene. In CUDA programming, lists of data are passed into the kernel as arrays. These arrays are first initialized in the host and are then are allocated with memory only visible by the kernel and device functions. Because a linked list is a conglomeration of possibly non-sequential elements in memory which contain pointers to each other, each element must separately have memory allocated. We figured it would be much more simple to allocate memory visible to the kernel and devices only one time as opposed to a separate time for each element in the linked list. With that mentality, we created a function to flatten the linked list. That function would know how many elements would be in the flattened array and thus how much memory to allocate.

All of the aforementioned changes can work on a sequential version, so we tested to make sure that a sequential C version containing these modifications would function before porting them over the parallel version. This was also helpful because we had access to a working debugger in the sequential version.

When porting the code over to the parallel version, we were required to allocate memory visible to the kernel and device functions for each array which would be accessed during the

ray tracing. These arrays include light data, object data, and the frame buffer output data. Filled host arrays such as light data and object data would then have to be copied from the host array to the device array in order to be read by the kernel and device functions. Similarly, the frame buffer data would have to be copied from the kernel memory back to the host, in order for the host to store the output in an image file. After allocating device memory for the appropriate arrays, we passed them on as arguments to the kernel function.

Finally, we had to decide how the threads were going to divide the ray tracing. We initially thought that each thread could trace multiple rays and sequentially write the output out to an indexed location (based on the thread and block coordinates) in the frame buffer array. However, many difficulties arose from this implementation, including the necessity to create of a function called in the host which would partition the pixel data between the total number of threads which would be computing in parallel. The overhead created from the partitioning of this data was also of concern. Finally, there did not seem to be one successful method we implemented which could transfer all of the frame buffer data back to the host and present acceptable output on machines with 1.x CUDA GPUs. The methods we tried included implementing a 2D array which could be passed into the kernel, a u_int128_t frame buffer which could hold the output data from four computations, and multiplying the generated index in each thread by a given number such that each thread would have a given number of output data to write into the frame buffer. Implementing a 2D array proved to be very complex due to CUDA's lack of support for simple memory allocation of device 2D arrays. If a 2D array can be conceived as an array of arrays, then there would have to be a separate device memory allocation call for each array element. Implementing a u_int128_t frame buffer proved unsuccessful because u_int128_t was not supported in our compiling environments.

After implementing each of these methods on machines with CUDA 1.x GPUs, we tested to see the pixel data transferred to the host frame buffer for various coordinates. This data would always equal zero or junk data previously left in memory. We believe there was a problem in transferring the memory from the device frame buffer back to the host frame buffer. However, since we were not able to use cuda-gdb when testing these functions, we cannot confirm that statement.

Many of our difficulties were the result of the original tracer having a design contrary to CUDA and parallel paradigms. Whereas in our initial iteration, we were attempting to loop through some given set of pixels within a device function and trace the ray located at those coordinates, we came to the conclusion that a truly parallel application would eliminate the loops altogether, replacing them with logic based on thread, grid, and block indices.

To this end, cudatrace now divides each image up into two-dimensional blocks of 256 (16 by 16) threads, a number we chose to ensure relatively broad cross-platform compatibility when dealing with GPUs that have varying maximum thread per block limits. Each thread handles the tracing and rendering of one single pixel. These blocks in turn make up a two-dimensional grid of dimensions (image width divided by the number of threads) by (image height divided by the number of threads). Thus, the full grid encompasses the full resolution of the image by way of maximized parallelism. In future iterations we may assess the benefits

of more or fewer threads per block.

This led to at least one complexity in our render functions, as we now needed to store all of these pixels, already logically oriented in a grid as they should be, in a one-dimensional array. This proved to be a relatively simple task, given CUDAs support for thread, block, and grid indices. Once we calculated the x and y coordinates of the pixel currently being operated on, storing it was simply a matter of using a flattened index of the form x times grid width + y. The final remaining snag was dealing with those image resolutions that were not an even multiple of 16 (the number of threads per block in each dimension). In those cases, we simply add one more partially filled block to the total number of blocks in that dimension. As a result, in our rendering functions we have to check to ensure that our x and y coordinates are less than the x and y dimensions of the image we are rendering to make sure we do not go out of bounds or overflow.

We were able to tap the most prominent source of parallelism in a ray tracing program– that is, tracing the rays in parallel. The main variable in this implementation is the number of threads executed per block. Increasing this would potentially lead to more parallelism, but may also lead to increased execution time as a result of CUDA overhead. We will be investigating this for the final version of our application.

To test the improvement between the original version of our tracer and our CUDA implementation, we ran a test suite that rendered images of varying resolutions with each version. The attached data set and graphs illustrate the results of our testing. In sum, for lower resolutions (roughly less than 2400 by 2400 pixels), we found that the non-CUDA version of the ray tracer was more efficient (by three orders of magnitude). For each of the resolutions tested below this limit, the CUDA version took roughly 3 seconds to render. We hypothesize that this is due simply to the overhead involved with copying the scene data to and from the GPU, and as an extension of this, the time actually spent rendering is much lower.

Above that 5.5 megapixel cutoff, however, the data changes drastically. The CUDA version of the tracer far outpaces its non-parallel counterpart. At 576 megapixels, the maximum resolution we were able to test out without running into memory allocation issues due to hardware limitations, the CUDA version was able to render the image in around 90 seconds, a stark comparison to the non-CUDA version, which took over seven minutes to complete.

We are satisfied with this performance gain. For the next version of the application, we intend to optimize the CUDA-related overhead and bring the parallel version of the tracer into closer competition with the sequential version and lower resolutions. This is a necessity for the CUDA tracer to be practical, as few users will need a ray traced image that is a half of a gigapixel in resolution!

The limitations of CUDA, such as the inability to call functions recursively or easily utilize linked lists, negated our ability to only implement minor or modest changes to the original code in order to generate a simple form of parallelism. We were forced to put much of the programming effort into restructuring a portion of the code to eliminate recursive calls and linked list usage.

Because of the necessity to implement dramatic changes to the structure of the code, we believe we have achieved a majority of the maximum possible performance increase with

this ray tracer. Thus, it seems unlikely at present that we will be able to obtain anywhere near an equivalent performance increase in our final version of the code. However, we will attempt to examine the overhead time caused by calls to CUDA related memory functions and optimize accordingly. Further, experiments into the effect that modifying the number of threads per block has on this overhead may dictate tweaks there.

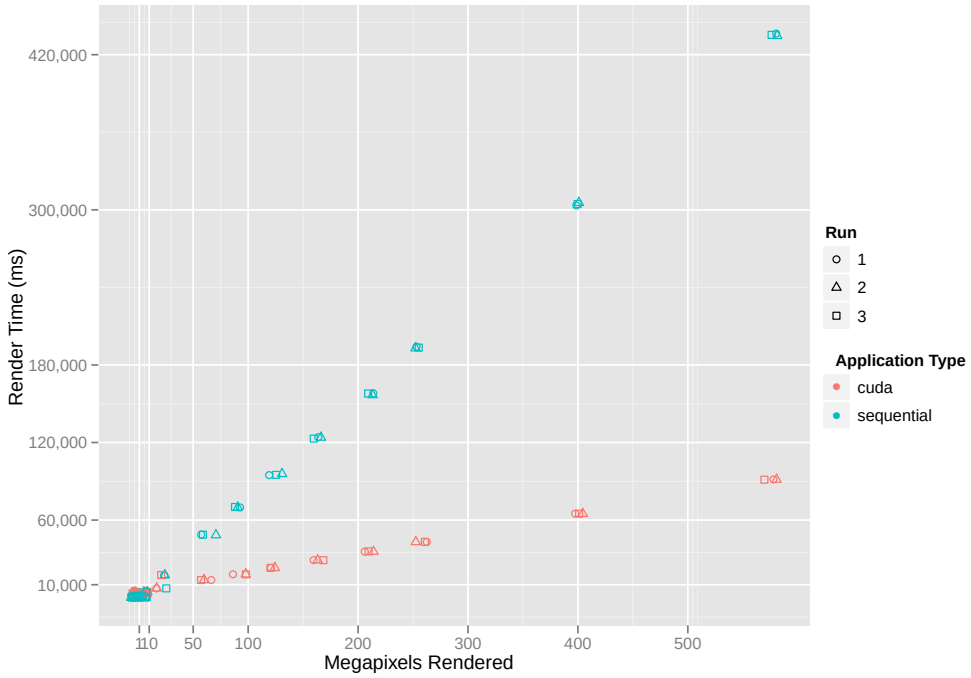
The decision to perform GPU programming using CUDA greatly affected our ability to determine a stable source of program output. We were only able to compile and run our program on one of our laptops, the GPU nodes on the PADS cluster, and certain MacLab machines. Each computer had varying Nvidia CUDA 1.x compute capability graphics cards, which may have influenced our initial varying results between computers. We later learned that if we set the -arch sm_13 or -arch sm_21 flags during compilation, we could force the compiler to act as if it was compiling with either a 1.3 card or a 2.1 card. Setting this flag allowed us to obtain more consistent results between machines, but we were limited to compiling with the -arch sm_13 flag to ensure functionality on the machines we had access to. This older version of the CUDA framework imposes several limitations not present in the 2.x capability cards: no implementation of double precision floats, no host functions called from device functions, lower limits on threads per core etc. Debugging with cuda-gdb also proved to be extremely limited in each environment. On both the MacLab and the PADS cluster, we were not able to debug unless X11/gnome was forcibly disabled or through console mode. We could access console mode on the MacLab machines, but were not able to run cuda-gdb due to errors in linking libraries. On the PADS cluster, we were never able to access any of the GPU nodes in console mode. One of our laptops had the ability to debug to some degree, but because the Geforce GT 330M in the machine only supports compute capability 1.2, we were limited to only debugging host functions. Any kernel or device function was stepped over in spite of breakpoints.

Because of these limitations, we spent a good portion of time guessing which parts of the program could be leading toward inappropriate host frame buffer output. After we had confirmed that all of our code modifications, when executed sequentially in a C program, produced correct output, we narrowed the culprit down to a CUDA-related peculiarity in transferring data between the host and device.

However, very late into the project, we were able to gain access to a high powered computing node on Amazons EC2 cloud service. The GPU computing nodes in Amazons cluster have top of the line Fermi class Nvidia GPU capable of executing CUDA 2.1 code. With this GPU, we were able to use cuda-gdb to step through the kernel and device functions on this machine. The program ran successfully and produced correct output without any further modification. For the next iteration of the project we intend to discern what differences there are between CUDA 2.1 and 1.3 that prevent our code from running correctly on the latter.

Additionally, both our tracer and the original tracer fail at very high resolutions (over approximately 600 megapixels) because we attempt to store all of the pixels in memory at the same time. For next version we will attempt to overcome this by writing to the file in blocks as the rendering occurs, rather than all at once after all rays are traced. Assuming we are able to accomplish this, our next hardware-related ceiling is the GPUs limitation on

Figure 1: Performance of Raytracer Implementations at Various Resolutions



number of thread blocks allowed in the grid. For modern hardware, this will not quickly become an issue, as CUDA supports 65,535 times 65,535 = 4,294,836,225 blocks. With our conservative implementation of 256 threads per block, each handling one pixel, we are left with a rough theoretical maximum of 1.09 trillion pixels.

The significance of the results that we were able to obtain without any major optimization to the original code is indicative of the fact that ray tracing lends itself very well to CUDA parallelization. In particular, the multi-dimensionality of threads within CUDA blocks and kernels is in itself a clear representation of the pixels in a two dimensional image. With further testing and the implementation of various lower level optimizations, we are confident that our cudatrace application will be an excellent example of CUDA's power.

Figure 2: Inset—Performance of Raytracer Implementations at Various Resolutions

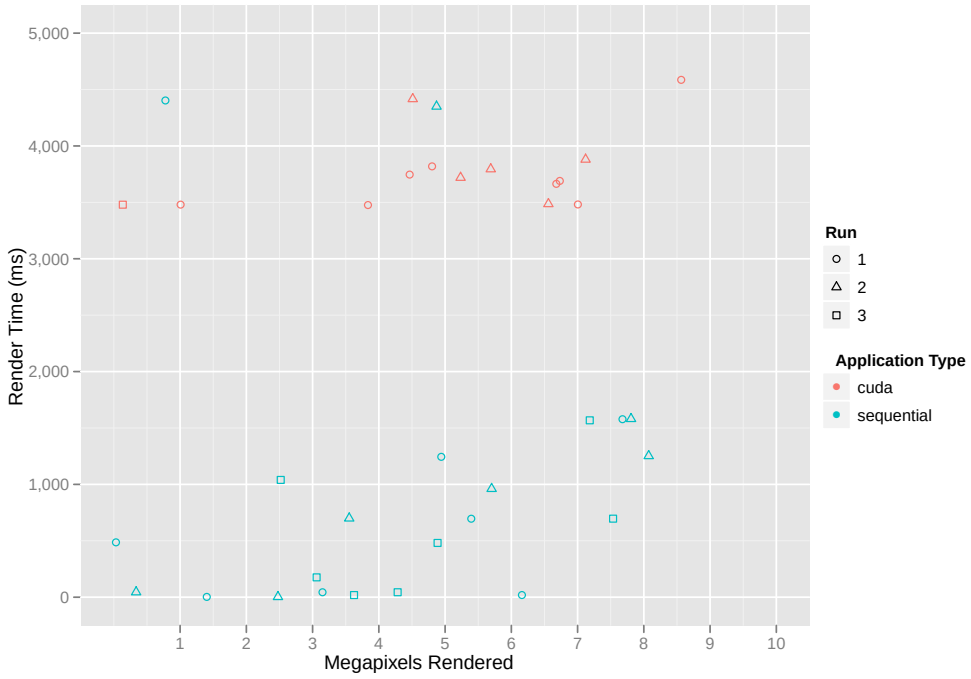


Figure 3: Performance of Raytracer Implementations at Various Resolutions—Run 1

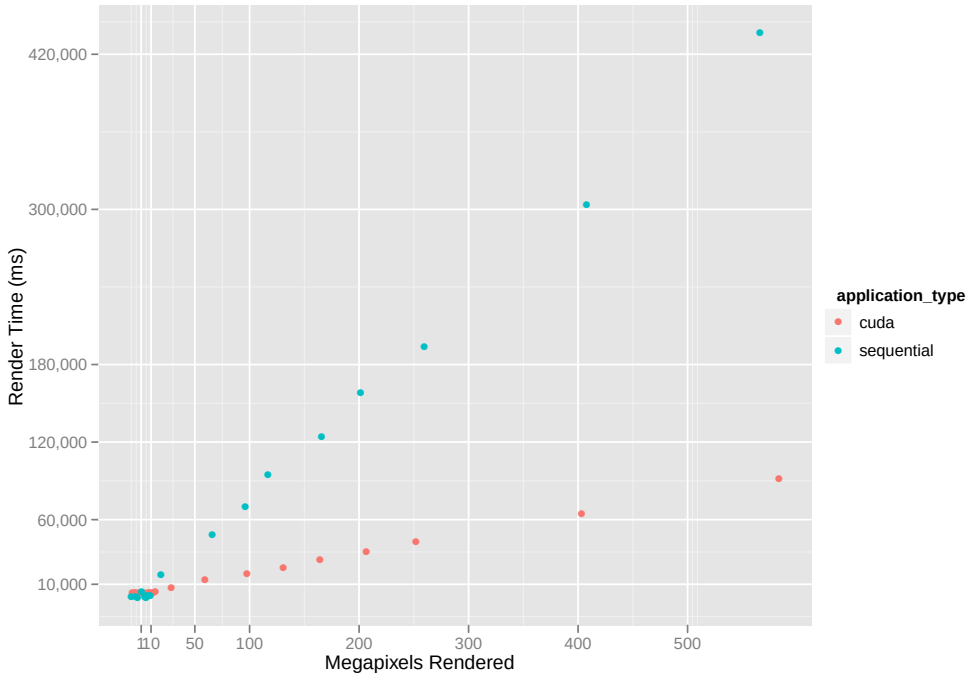


Figure 4: Performance of Raytracer Implementations at Various Resolutions—Run 2

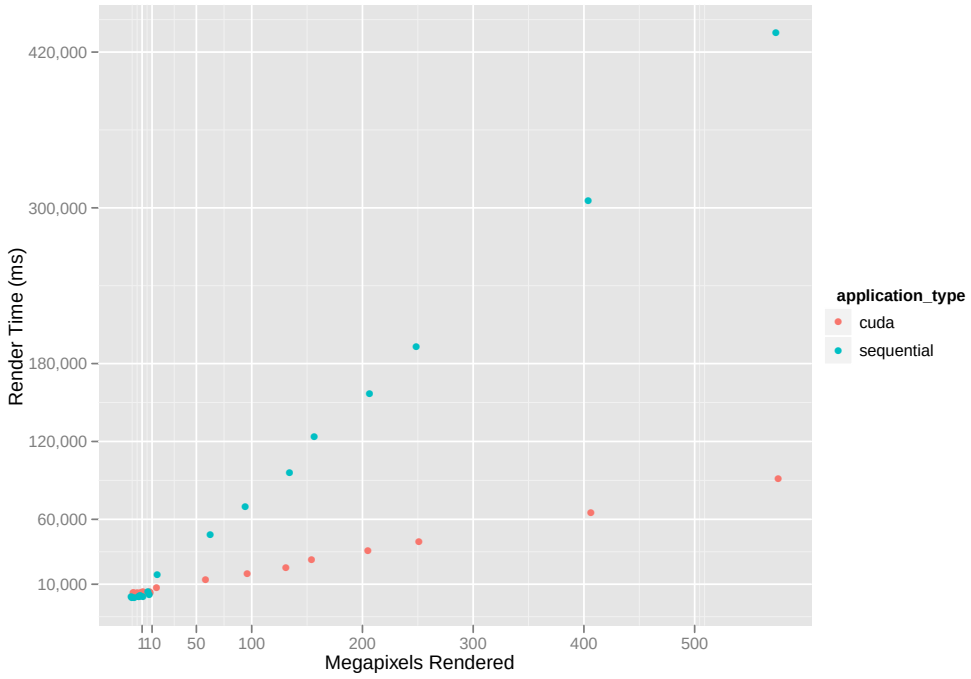


Figure 5: Performance of Raytracer Implementations at Various Resolutions—Run 3