ECE 473 / CS 572 CUDA Extra Credit:
Numerical Integration

Forrest Briggs and Nathan Haller

December 2, 2009

1 Overview of CUDA
CUDA (Compute Unified Device Architecture) is NVIDIA’s programming language for general purpose parallel computing on a GPU. The language is essentially a subset of C. The primary limitation is that recursion is not allowed. However, CUDA makes it relatively simple to run parallel code in a very large number of simultaneously executing threads that have access to a single shared memory. In contrast with classic CPU threads, CUDA GPU threads are very light-weight, i.e. they do not require a large amount of overhead to create. For a tutorial on installing CUDA for Mac OS X, setting up XCode (Apple’s C/C++ IDE), and mixing C++ and CUDA, refer to http://web.engr.oregonstate.edu/~briggsf/cuda/.

2 Numerical Integration
In this paper, we explore parallelizing definite numerical integration using CUDA. A simple way to numerically evaluate the definite integral \( \int_a^b f(x)dx \) is to divide the range from \( a \) to \( b \) into \( n \) rectangles, evaluate \( f(x) \) once for each, then add up the results. The integral can be approximated using the Riemann sum:

\[
\int_a^b f(x)dx \approx \frac{b-a}{n} \sum_{i=0}^{n-1} f(i/n)(b-a) + a
\]

To make the computation simpler, we will assume \( a = 0 \) to \( b = 1 \), which gives:

\[
\int_0^1 f(x)dx \approx \frac{1}{n} \sum_{i=0}^{n-1} f(i/n)
\]

For the purposes of experimentation, we consider two functions to be integrated, and also vary \( n \). The first function is the probability density function for a standard normal distribution:

\[
f_1(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}
\]

This function is relatively fast and simple to compute. Note that \( \int_0^1 f_1(x)dx \) is the probability that a standard normal random variable lies in the range 0 to 1, and that it is not possible to compute this probability without numerical integration, so this is actually a useful calculation. The second function is \( \cos \) nested 16 times, which requires much more computation:

\[
f_2(x) = \cos(\cos(\cos(\cos(\cos(\cos(\cos(\cos(\cos(\cos(\cos(\cos(\cos(\cos(\cos(x)))))))))))))))
\]
3 Sequential C Code

The sequential C code to compute $\int_0^1 f_1(x)\,dx$ is listed below. It is a fairly straight-forward translation of the Riemann sum formula for the definite integral.

```c
float integrate(int N)
{
    float sum = 0;
    for(int i = 0; i < N; ++i)
    {
        float x = (float) i / (float) N;
        float fx = (1.0f / sqrt(2.0f * M_PI)) * exp(-x * x / 2.0f );
        sum += fx;
    }
    sum *= 1.0f / (float)N;
    return sum;
}
```

4 Parallel CUDA Code

The parallel CUDA code to compute $\int_0^1 f_1(x)\,dx$ is listed below.

```c
__global__ void integratorKernel(float *a, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    float x = ... add up results
    float sum = 0;
    for (int i=0; i < N; i++) sum += a_h[i];
    sum *= 1.0 / (float)N;;
    // clean up
    free(a_h); cudaFree(a_d);
    return sum;
}
```

The code is divided into two functions named `cudaIntegrate` and `integratorKernel`. Only the code in `integratorKernel` is actually executed on the GPU. The `cudaIntegrate` function is standard C which sets up the necessary arrays to perform the calculation, and spawns the CUDA threads. It allocates an array with $n$ elements on the host (CPU) and on the device (GPU). The first line of the kernel determines which element of the sum it is computing ($idx$), then computes $f(x)$, and writes the result to the corresponding element in the device array. After the threaded computation finishes, the array is copied from the device to the host. Then sequential C code is
Table 1: The results for integrating $f_1$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>C Runtime (ms)</th>
<th>CUDA Runtime (ms)</th>
<th>C Runtime / CUDA Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.021</td>
<td>0.072</td>
<td>0.305</td>
</tr>
<tr>
<td>1,000</td>
<td>0.075</td>
<td>0.068</td>
<td>1.114</td>
</tr>
<tr>
<td>10,000</td>
<td>0.588</td>
<td>0.392</td>
<td>1.499</td>
</tr>
<tr>
<td>100,000</td>
<td>5.774</td>
<td>3.586</td>
<td>1.609</td>
</tr>
<tr>
<td>1,000,000</td>
<td>77.584</td>
<td>17.262</td>
<td>4.494</td>
</tr>
<tr>
<td>10,000,000</td>
<td>586.867</td>
<td>135.617</td>
<td>4.327</td>
</tr>
</tbody>
</table>

Table 2: The results for integrating $f_2$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>C Runtime (ms)</th>
<th>CUDA Runtime (ms)</th>
<th>C Runtime / CUDA Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.074</td>
<td>0.061</td>
<td>1.228</td>
</tr>
<tr>
<td>1,000</td>
<td>0.738</td>
<td>0.074</td>
<td>9.846</td>
</tr>
<tr>
<td>10,000</td>
<td>5.625</td>
<td>0.526</td>
<td>10.693</td>
</tr>
<tr>
<td>100,000</td>
<td>55.896</td>
<td>1.897</td>
<td>29.462</td>
</tr>
<tr>
<td>1,000,000</td>
<td>565.397</td>
<td>18.614</td>
<td>30.374</td>
</tr>
<tr>
<td>10,000,000</td>
<td>5580.010</td>
<td>169.141</td>
<td>32.990</td>
</tr>
</tbody>
</table>

used to add up the elements of the array.

The code to integrate $f_2$ is almost identical (it differs only in the two lines that compute $f(x)$), so we omit it for brevity.

5 Experiments & Conclusion

To measure the speedup (or slowdown) from using CUDA to calculate these integrals, we tried varying $n$. Runtimes are measured in milliseconds. Experiments were run on a MacBook Pro with dual 2.53 Ghz Intel processors, 4Gb of RAM, and an NVIDIA GeForce 9400M graphics card. We verified that the C and CUDA code produce identical results. The sequential C code was not compiled at the maximum level of compiler optimization in gcc, which might produce a 2-4x speedup (gcc will go so far as to automatically use SIMD instructions on an Intel processor with the -03 optimization level).

Tables 1 and 2 list the results for integrating $f_1$ and $f_2$ with varying values of $n$. The speedup for using CUDA increases as $n$ increases. This is because the CUDA code incurs a large amount of overhead while calling malloc to allocate the shared memory arrays, and to copy the data between the CPU and GPU. The C code does not use any arrays, so it avoids this overhead. The cost of the overhead is only amortized when $n$ is very large. We also note that there is much more speedup for integrating $f_2$ than $f_1$. We think this is because the GPU is heavily optimized for computing cos, as it is a common operation in graphics, and because $f_2$ requires more computation, and hence the overhead is less significant by comparison. Another interesting quirk is that the first time we execute a CUDA call within a process, it runs slower than subsequent times. We speculate that CUDA is doing some setup the first time it is called.

To our knowledge, it is not possible to run this sort of calculation in CUDA without using shared memory, so some overhead is unavoidable. However, in our CUDA implementation, computing the sum is done sequentially. It is possible, but much more complicated, to compute this sum with some degree of parallelism. Further speedup could be obtained by parallelizing the sum, although available performance analysis of parallel sum computations lead us to suspect that the improvement would be marginal.