High Performance Computing for Engineers
Coursework 2 - Numerical Integration

Mohammad Mirza

March 22, 2013

The Math

Integration approach

The numerical integration was done using the rectangule rule described in the coursework specifications.

Error estimation approach

The approach implemented was as follows

- Lauch Kernel for integration at sampling rates N and 2N.
- Compare sum from sampling rates N and 2N
- Loop until sum comparison less than error tolerance, eps

Other approaches investigated include Adaptive Quadrature. Where the solution adapts to the shape of the curve by eliminating all regions where the integrand is well behaved(ie, satisfies the error criterion). At the same time, the resolution of the regions that do not conform to the tolerence are eliminated and the sampling resolution is modified by splitting the range and performing the integration again. This divide and conquer approach maximises the GPU's parallelism. This method was not implemented.

Pseudo-code

Algorithm 1 Kernel Psuedo-code

```
__DEVICE_SIDE___
KERNEL integrate_FX:
        Initialize_Kernel()
        determine x array corresponding to this thread
        store to y[i] which corresponds to each thread index
        Decommission Kernel()

__HOST_SIDE__
FUNCTION integrate(grain_size):
        Copy from host memery to device memory
        while |actual_area - area| > tau:
                <<kernel>>y = integrate_FX(grain_size) // FX determined byfunction to be inte
                <<kernel>>actual_y = integrate_FX(grain_size * 2)
                // copy array dy from device to y
                area = sum(y) // accumulate the threads
                actual_area = sum(actual_y)
```

CPU

- 2 Intel Xeon X5570, quad-core were used on the CPU side for the tests.

CPU Optimisations

- Possible optimisations included adding a parallel reduce using kernel dcompostion on the GPU side to reduce the sequential accumulation of sum. This involves recursive kernel invocation and was not attempted due a time contraint.

GPU

Development was done on the CUDA platform with a Tesla M2050 (see Appendix for Device Specifications)

GPU Specifications

- 32 thread warp size
- 1024 x 1024 x 64 with max number of threads per block of 1024
- Floating point operations were performed within the kernel to allow for a backwards compatibility with older architectures

GPU Optimisations

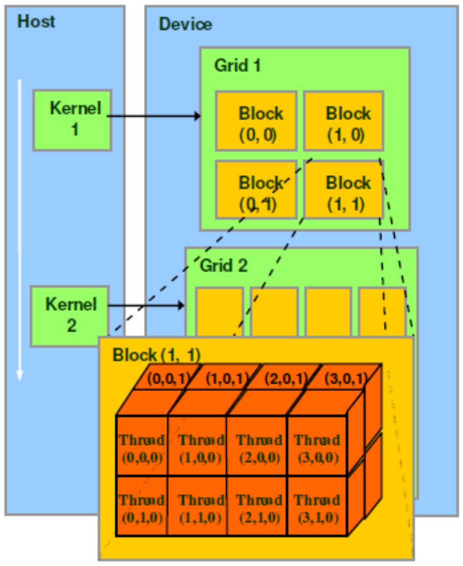

Figure 1: Grid and Block Size Architecture for CUDA Devices

Since the GPU only allows a maximum of 1024 threads per block the following block size configurations were chosen:

- 1D: A block size of 32 x 1 x 1 was used
- 2D: A block size of 32 x 32 x 1 was used
- 3D: A block size of 8 x 8 x 8 was used - Size chosen for symmetry

When the value of n is larger than the block size, additional blocks are launched within the kernal grid. This is referred to as the grid size of the grid.

Possible but not implement enhancements

- Make the variable, **base,** part of shared memory to prevent calling to Global memory every time. Global memory is large but slow, whereas shared memory is small buy fast. A common strategy is to partition the data into subsets and fit these subsets into the shared memory. Compution on these tiles can be performed independent to other kernel invocations. This requires a **__syncthreads()** so that all threads within the warp have this available before execution continues.
- Agglomerate the addition of the 32 threads within the kernel. This allows us to reduce the time taken to sum over all threads sequentially on the host side. The sum loop is reduced by a factor of 32.
- Initially, the function pointers were used to call the correct function from within the kernel. This approach was abandoned and a seperate kernel for each function was used instead to allow for backwards compatibility. *NB: Function pointers are not supported before CUDA Capability Version 2.0*

Analysis

Trade-offs between execution time and error

In this implementation, error estimation is done via the parallel execution of multiple kernels. This process repeats until the result converges to the tolerance specified. The overhead of this is repeated kernel invocation and copy new values for the base into the device for each sampling rate.

Starting at a low sampling rate reduces the chance of passing the tolerance test and therefore requires additional kernel invocations. On the other hand, starting at a high sampling rate might cause overhead in terms of too many threads created on the device. We could agglomerate and create less threads in return for each thread performing a bigger chunk of computing power. This requires performance tuning for specific devices. This agglomeration also reduces the memory overhead on the device as smaller arrays for output need to be allocated.

Comparisons with Sequential

The sequential version was generally faster for large n, since the parallism capabilites of the GPU are maximised. For very small n, we are better off using a sequential version. The following table shows a comparison between a slightly modified dumb_impl.c and integ.cu (this file was created for comparison and does not do error estimation, it simply calculates area for the given n).

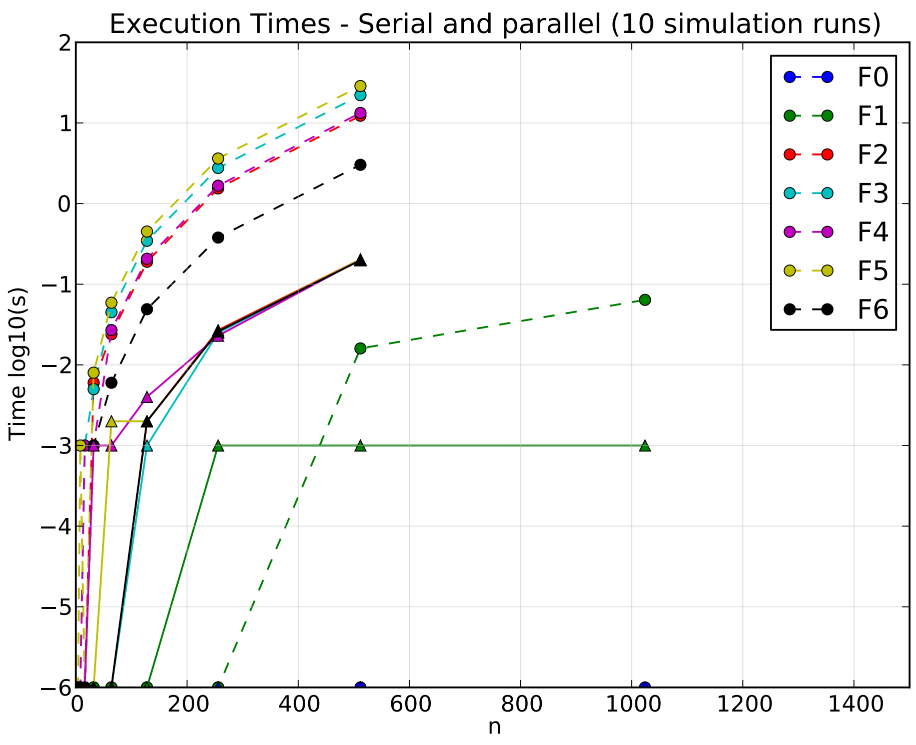

Figure 2: Compariton with serial CPU version - Circles represent CPU and Triangles represent GPU implementation

Appendix

Device 0: "Tesla M2050"
CUDA Driver Version / Runtime Version 5.0 / 5.0
CUDA Capability Major/Minor version number: 2.0
Total amount of global memory: 3072 MBytes (3220897792 bytes)
(14) Multiprocessors x (32) CUDA Cores/MP: 448 CUDA Cores
GPU Clock rate: 1147 MHz (1.15 GHz)
Memory Clock rate: 1546 Mhz
Memory Bus Width: 384-bit
L2 Cache Size: 786432 bytes
Max Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers 1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 32768
Warp size: 32
Maximum number of threads per multiprocessor: 1536
Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 2 copy engine(s)
Run time limit on kernels: No
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes

```
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
Device supports Unified Addressing (UVA): Yes
Device PCI Bus ID / PCI location ID: 0 / 3
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```