SHARCNET Summer School - Programming GPUs with CUDA - Tutorial Day 1

Pawel Pomorski

May 9, 2013

The files needed for this tutorial can be found in
/home/ppomorsk/CUDA_day1

The access to directories with solved problems is disabled for now.
These exercises can be done in any combination, but a person new to CUDA should consider doing them in order given, as they progressively increase in difficulty.
Please ask the instructor for advice if you get stuck.

1 SAXPY

A number of example implementations of SAXPY have been provided in directory saxpy. Experiment with compiling and running these for different vector sizes. Practice timing the code. Estimate how much time is spent transferring data to/from CPU vs the actual computation time on the GPU.

Modify the timed version of the saxpy code to use cudaMallocHost instead of malloc to allocate host memory (uses page-locked memory for higher performance). Use timing to determine the improvement in performance this gives.

Instructions for compiling are given in the initial comments of the program files. The CUDA and CUBLAS code provided here are good starting templates for problems 2 and 3.

2 Visualizing the Julia set

The serial CPU program julia_cpu.cu generates a visualization of the Julia set. Using this program as a starting point, write a CUDA program which accomplishes the same thing.

The Julia set is obtained by generating a sequence starting from a complex number Z_0

$$Z_{n+1} = Z_n^2 + C \tag{1}$$

where C is a complex parameter. If this sequence converges then the starting point Z_0 is in the Julia set. The set can be visualized in the complex plane, producing interesting patterns. The starting point is set up to produce an output file which can be viewed in gnuplot.

Performance is not an issue in this exercise, so you do not need to time this code.

3 Accelerating matrix multiplication

Develop a matrix multiplication code that is accelerated on the GPU. A starting program with matrix multiplication implemented on the CPU has been provided. Use that as your starting point, with the CPU calculation used as a check on the correctness of the GPU calculation.

For simplicity, consider only the case of A x B = C where matrix A is Nx32 and matrix B is 32xN, and N is a multiple of 32. (We have chosen 32 because then the matrix lends itself very well for decomposition into optimal 32 by 32 blocks)

A CPU only matrix multiplication benchmark implemented with MKL library is also provided. It can be run in serial and threaded fashion.

3.1 Accelerating with CUDA

Implement a GPU-accelerated matrix multiplication via CUDA. Use the CPU code provided as a starting point. Develop your GPU kernel by altering the CPU matrix multiplication function.

3.2 Accelerating with CUBLAS

Take your CUDA program and replace the kernel you wrote with CUBLAS library function cublasSgemm. Use the calling scheme in MKL benchmark.

3.3 Accelerating further with shared memory

(Optional) Once both your CUDA code and CUBLAS code are working, attempt to accelerate performance of your CUDA kernel to get a result closer to the highly optimized library. You can attempt to do this by using shared memory. In your matrix multiplication kernel, define a shared block of size 32×32, copy data from global memory to it (don't forget to synchronize therads after copying), and then use it in the kernel loop doing row times column multiplication.

3.4 Measuring performance

The goal is to acquire a quantitative feel for the acceleration capability of the GPU so timing your code is important. Try different problem sizes to observe how speedup varies with size.