Tutorial: HPC - Algorithms and Applications
WS 13/14

Complete the following assignments (alone or in a group), and send your source code via e-mail to meistero@in.tum.de until Sunday, November, 10th 2013.

Worksheet 1: Matrix Multiplication in CUDA

Assignment 1: Make it run

Download and run the first exercise code either on your machine or a remote machine.

a) Use SSH to remote login on the chair: ssh -X atsccs30.informatik.tu-muenchen.de

b) Open firefox &, navigate to http://www5.in.tum.de/wiki/index.php/HPC_-_Algorithms_and_Applications_-_Winter_\_13, then download and extract the exercise code, i.e. into ~/HPC/Exercise1

c) Change to the folder: cd ~/HPC/Exercise1 and export the path variables:

   export PATH=$PATH:/usr/local/cuda/bin
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib64

d) Type make, compilation should work.

e) Run the code using for example out 16 1. An error out: malloc(): memory corruption (fast) should occur. This is expected and due to missing code that you will have to implement in Assignment 2.

Assignment 2: Basic matrix multiplication

Write a simple matrix-matrix multiplication program for small matrices (up to \( n = 16 \)) using CUDA. Thus, implement the TODOs in cuda_mmult.cu and cuda_mmult_kernels.cu of the exercise code. The following tasks are necessary:

a) Compute the amount of memory required for storing an \( n \times n \) matrix with single floating point precision.
b) Allocate and deallocate device memory, and transfer the input matrices \( A \) and \( B \) from host to device memory, and the result matrix \( C \) back from device memory to host memory.

c) Define grid and block size and call the device kernel. You can assume for now, that a single block is sufficient for a matrix.

d) Implement a basic matrix multiplication kernel in the function `matrixMultKernel_global`.

e) Run the code for \( n = 16 \). If everything is correct, the resulting matrix is diagonal, and each diagonal entry has the value \( 8.50 = \frac{n+1}{2} \). You can disable (enable) output by (un)commenting the macro definition of `OUTPUT`.

**Assignment 3: Increase of problem size**

Extend the code by allowing matrices of size \( n > 16 \).

a) Change your grid and block size computation to handle \( n \times n \) matrices for any \( n > 0 \) (assuming the matrices fit into device memory).

b) Change your matrix multiplication kernel to handle the new grid and block sizes.

c) Do a measurement of the execution time using different block sizes. Compare the execution time with the execution time of a CPU code by replacing the call to `CUDA_matrixMult` with `CPU_matrixMult`. What do you observe? What is the optimal block size for a matrix of size \( 256 \times 256 \)?

**Assignment 4: Tiled matrix multiplication**

Implement a tiled matrix multiplication kernel for improved memory performance. The tile size is defined in a preprocessor macro called `TILE_SIZE`.

a) In the function `matrixMultKernel_tiled`, allocate shared memory that holds matrix tiles of size `TILE_SIZE \times TILE_SIZE` for the matrices \( A \) and \( B \).

b) Fill the shared tiles with data from the matrices.

c) Perform a partial matrix multiplication on the shared tiles.

d) Do not forget to set appropriate thread barriers where necessary in order to synchronize all threads of a block.

e) Again, compare the performance of CPU, basic GPU and tiled GPU matrix multiplication. Which implementation is the fastest?