Complete the following assignments (alone or in a group), and send only your source code via e-mail to meistero@in.tum.de until Sunday, November, 2nd 2014.

**Worksheet 1: Matrix Multiplication in CUDA**

**T1.1: Basic matrix multiplication**

a) Write a simple matrix-matrix multiplication for small matrices (up to \( n = 16 \)). The following tasks are necessary:

- Compute the amount of memory required for storing an \( n \times n \) matrix with single floating point precision.
- Allocate device memory, transfer the input matrices \( \mathbf{A} \) and \( \mathbf{B} \) from host to device memory and the result matrix \( \mathbf{C} \) back to host memory, deallocate device memory.
- Define grid and block size and call the device kernel. You can assume for now, that a single block is sufficient for a matrix.
- Implement a basic matrix multiplication kernel in the function `matrixMultKernel_global`

**H1.1: Make it run**

Assignments a) and b) are mutually exclusive, complete only one of them.

a) Compile the exercise code on a local machine:

- Verify you have a CUDA-capable GPU and install the CUDA toolkit.
- Download and extract the exercise code from [http://www5.in.tum.de/wiki/index.php/HPC_-_Algorithms_and_Applications_-_Winter_14](http://www5.in.tum.de/wiki/index.php/HPC_-_Algorithms_and_Applications_-_Winter_14), i.e. into `~/HPC/Exercise1`
• You might have to export some path variables:
  export PATH=$PATH:<cuda_dir>/bin
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<cuda_dir>/lib64
• Change to the folder cd ~/HPC/Exercise1 and type make, compilation should work.

b) Compile the exercise code on the MAC cluster:

• Open an ssh connection to the MAC cluster (see slides).
• Open firefox & navigate to http://www5.in.tum.de/wiki/index.php/HPC_-_Algorithms_and_Applications_-_Winter_14, then download and extract the exercise code, i.e. into ~/HPC/Exercise1
• Load required modules: module load cuda; module switch gcc gcc/4.6
• Change to the folder: cd ~/HPC/Exercise1 and type make, compilation should work.
• Open an interactive shell on an NVidia node: salloc --ntasks=1 --partition=nvd
c) Test the code using ./out 8 1 (local) or srun ./out 8 1 (MAC-Cluster). If everything is correct, the resulting matrix is diagonal, and each diagonal entry has the value $4.50 = \frac{n+1}{2}$. You can disable (enable) output by (un)commenting the macro definition of OUTPUT.

H1.2: Increase of problem size

a) Extend the code by allowing matrices of size $n > 16$.
   • Change your grid and block size computation to handle $n \times n$ matrices for any $n > 0$. You can assume that the matrices fit into device memory.
   • Change your matrix multiplication kernel to handle the new grid and block sizes.

b) Measure 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. Find the optimal block size for a matrix of size 256 $\times$ 256.

H1.3: Tiled matrix multiplication

a) Implement a tiled matrix multiplication kernel for improved memory performance. The tile size is defined in a preprocessor macro called TILE_SIZE.
   • 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.
   • Fill the shared tiles with data from the matrices.
   • Perform a partial matrix multiplication on the shared tiles.
   • Set appropriate thread barriers in order to synchronize all threads of a block.

b) Compare the performance of CPU, basic GPU and tiled GPU matrix multiplication for a matrix of size 256 $\times$ 256. Which implementation is the fastest?