Lab Guide 11

Program Optimisations for GPUs

Objective:

- get acquainted with technics to improve performance of GPU programs

Introduction

This lab session aims to introduce more advanced concepts of GPU device programming [using CUDA], through the improvements of the parallelisation of an one-dimensional stencil and the implementation of a matrix multiplication.

The skeleton of a CUDA base code and `Makefile` is available from lab session 10 (`/share/cpar/PL10_Codigo` folder in the SeARCH cluster).

Setup the CUDA environment with a compatible GNU compiler using the following commands on the cluster:

```
module load gcc/7.2.0
module load cuda/11.3.1
```

Compile the program in the cluster frontend using the provided `Makefile`, by executing the `make` command. Use the `sbatch run.sh` command to submit a job to the cluster to run the application on a node with a NVIDIA Kepler k20 GPU.

Exercise 1 – 1D stencil with shared memory

Consider the following GPU kernel of the 1D stencil from the lab session 10:

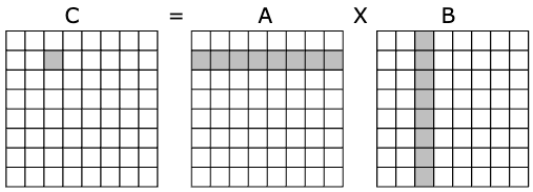
```
__global__
void stencilKernel (float *a, float *c) {
      int id = blockIdx.x * blockDim.x + threadIdx.x;
      // int lid = threadIdx.x;  // local thread id within a block

      // __shared__ float temp[NUM_THREADS_PER_BLOCK + ??];
      // temp[lid ???] = a[id ???];
      // if ...

      //__syncthreads();  // wait for all threads within a block
      c[id] = 0;

      for (int n = -2; n <= 2; n++) {
            if ((id + n>= 0) && (id + n < SIZE))
                  c[id] += a[id + n];  // c[id] += temp[??? + n]
      }
}
```

Adapt the `stencilKernel` to use CUDA shared memory by changing the lines given as comments. Initially, the full data required by a thread block should be loaded into shared memory (`temp` array). The computations in the main loop of the kernel can then be performed by reading this faster memory.

(*) Exercise 2 – Matrix multiplication (MM) on GPU

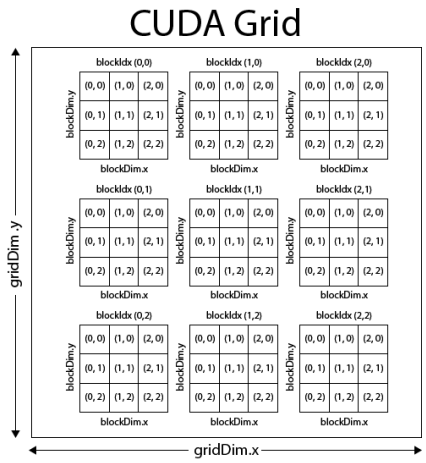
Consider the DOT version of the MM code from lab session 1.

```
/* Cij =0 */
for i=0 to N-1
  for j=0 to N-1
    for k=0 to N-1
       C[i][j] += A[i][k] * B[k][j]
```



a) Develop a (simple) CUDA based implementation of this code. In this implementation each thread computes an element of the resulting C matrix. A 2D thread structure can be specified by (using a 2D grid of thread blocks of 16x16):

```
dim3 threadsPerBlock (16, 16);   // block of 16x16 threads
dim3 blocksPerGrid (N/16, N/16); // grid of N/16xN/16 blocks
mmKernel<<< blocksPerGrid, threadsPerBlock >>> (da, db, dc);
```



With this thread organization the MM kernel becomes (almost) trivial:

```
__global__
void mmKernel (float *a, float *b, float *c) {
   int i = blockIdx.y*blockDim.y+threadIdx.y;
   int j = blockIdx.x*blockDim.x+threadIdx.x;
     ...
   c[i*N+j] = ...
}
```

b) Change the previous implementation to use thread blocks of size 256x1. The MM performance improves with this thread organization? Why?

c) Develop a version using shared memory, based on the implementation in 2.b). Load into shared memory the data that is most reused within a thread block.