CUDA Introduction Exercises

Dorothea vom Bruch

Requirement: Access to a server with an Nvidia GPU and with CUDA installed on the server.
(For people with a CERN account, it is possible to interactively use GPUs on the grid, as desribed here: https://batchdocs.web.cern.ch/tutorial/exercise10.html)

Environment setup:
The CUDA installation location must be added to your PATH variable. Typically like this (although different server environments can vary):

```
export PATH=/usr/local/cuda-xx.v/bin${PATH:+:${PATH}}
```

Here xx.v is the CUDA version available on your server.

How to compile:
Run the following command in the directory where the source file (device_properties.cu) is located:

```
nvcc device_properties.cu -o device_properties
```

This will run the compiler nvcc on the source file (device_properties.cu) and create an executable (device_properties).

How to execute the program:

```
./device_properties
```

General remarks

All of the following examples take as one of their input arguments the device to use. This is an index ranging from 0 to N-1, where N is the number of GPUs in your system. If you only have one GPU, the device id is 0. You can run

```
nvidia-smi
```

on your server to know how many and which GPUs are available.

Hello World

You will write your first own CUDA program! Start from the skeleton provided in `hello_world_start.cu`. Write a kernel (`__global__` function) that prints the thread and block index using `printf`. You can follow this example usage of printf:

```
printf("Hello World from block %u, thread %u \n", block_index, thread_index);
```

Call the kernel with the dimensions specified by the input arguments.
Output from this program could look like this:

```
Hello World from block 2, thread 0
Hello World from block 2, thread 1
Hello World from block 2, thread 2
Hello World from block 1, thread 0
Hello World from block 1, thread 1
Hello World from block 1, thread 2
Hello World from block 0, thread 0
Hello World from block 0, thread 1
Hello World from block 0, thread 2
```

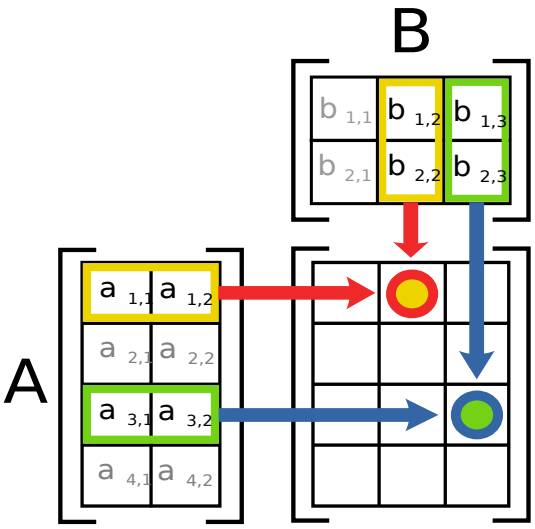
What do you observe about the order and grouping of the blocks and threads?

Vector addition

In this exercise you will add two vectors of the same size and store the result in a third vector. The operation should be done in parallel.
Start from the skeleton code in `vector_addition_start.cu`. Every thread should do the addition of one element of each input vector and store it in the corresponding element of the output vector.
Try different vector sizes and different numbers of threads per block.

Matrix multiplication

The multiplication of two matrices is an inherently parallel problem since all entries in the result matrix are independent from one another. Therefore, they can all be computed in parallel.



By File:Matrix multiplication diagram.svg:User:BilouSee below. - This file was derived from: Matrix multiplication diagram.svg, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=15175268

When describing a matrix in CUDA, it should be done in a 1d array, since the number of columns has to be known at compile time. For this, you can refer to the indexing example in the introduction slides.

1) Single Thread, single block

Code to do the matrix multiplication with a single thread and a single block is already provided in `matrix_multiply.cu`. The two input matrices are `host_matrix[0]` and `host_matrix[1]`, the output matrix is `host_matrix[3]`. Take a look at the code and make sure you understand what is happening.

2) Multiple threads
Copy the previous code to a new file. Now use `n_threads x n_threads` per block, this means that you are defining a 2d block, so you use `threadIdx.x` and `threadIdx.y`. Your block dimension is now defined like this: `dim3 block(n_threads, n_threads)`. You can capture `n_threads` from the argument list as you did for the vector addition. Keep using only one block per grid for now. Modify your kernel (the `__global__` function), so that entries of the destination matrix are calculated in parallel by the threads. Note that if not enough threads are available for the size of the matrix, one thread has to calculate several entries, so a loop is required.

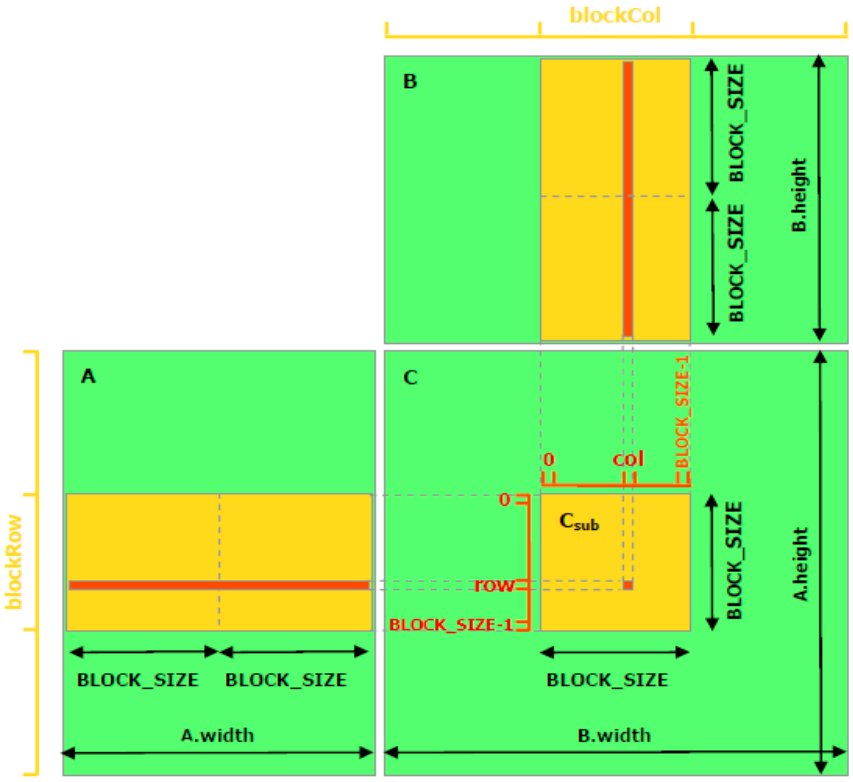
3) Multiple threads and blocks
Copy the previous code to a new file.
Add more parallelization by using several blocks in your grid. Use so many blocks that every thread only calculates one element of the destination matrix. The easiest way to do this is to use a 2d grid of blocks, so you now use `blockIdx.x` and `blockIdx.y` and define the grid size by `dim3 grid(n_blocks, n_blocks)`, where
`int n_blocks = size / (n_threads) + (size % (n_threads) != 0);`
Understand why `n_blocks` is defined like this!

4) Single vs. double precision
Use your previous example and study the effect of single and double precision floating point variables on your results. You can easily switch between the two by using a global typedef: `typedef float my_float_t` or `typedef double my_float_t`. This is already defined at the top of the code.
Which changes do you observe? What should you consider when choosing the precision of your variables? In which cases is double precision important?

5) Shared memory
Loading data from global memory is expensive, therefore it can be advantageous to first load data that is used by multiple threads in a block into shared memory and then read the data from there.
For the matrix multiplication, we can cache sub-matrices of the input matrices in shared memory and calculate a sub-matrix of the result matrix with every block. Every thread in a block calculates one element of the result sub-matrix. This is illustrated in the picture on the next page.
If we choose the size of the sub-matrix to be `n_threads x n_threads`, several such input sub-matrices are required to calculate the resulting sub-matrix. We can load the first input sub-matrices into shared memory, multiply them and store the resulting sum in a register, since one thread only calculates one element of the output sub-matrix. Subsequently, the next sub-matrices can be loaded into shared memory and the result of the multiplication added to the previous one, and so on.

When using shared memory, you should use all available threads in a block to load the data from global memory into shared memory. Remember that thread synchronization is required now before accessing data in shared memory to make sure that all threads have finished loading the data from global memory.

Start your work from the file `matrix_multiply_shared_memory_start.cu`



From nvidia programming guide