Rosenheim
Technical University
of Applied Sciences

Exercise sheet 11 – GPU

*Powered by Dr. Christoph Riesinger*

Goals:

- Get your first CUDA program compiled, linked, and executed on the GPU
- Utilize shared memory to pass data between different threads of a block
- Understand the persistence of data in different GPU memory types

Exercise 11.1: Processor architecture: GPU programming

In this exercise, we will implement the dot product – a basic mathematical operation having numerous applications – with CUDA. The dot product is defined as
$a \cdot b = (a_1, ..., a_n)^T \cdot (b_1, ..., b_n)^T = a_1 * b_1 + ... + a_n * b_n = c \in R; a, b \in R^n$.
In the following, you can **expect n is a power of 2**.
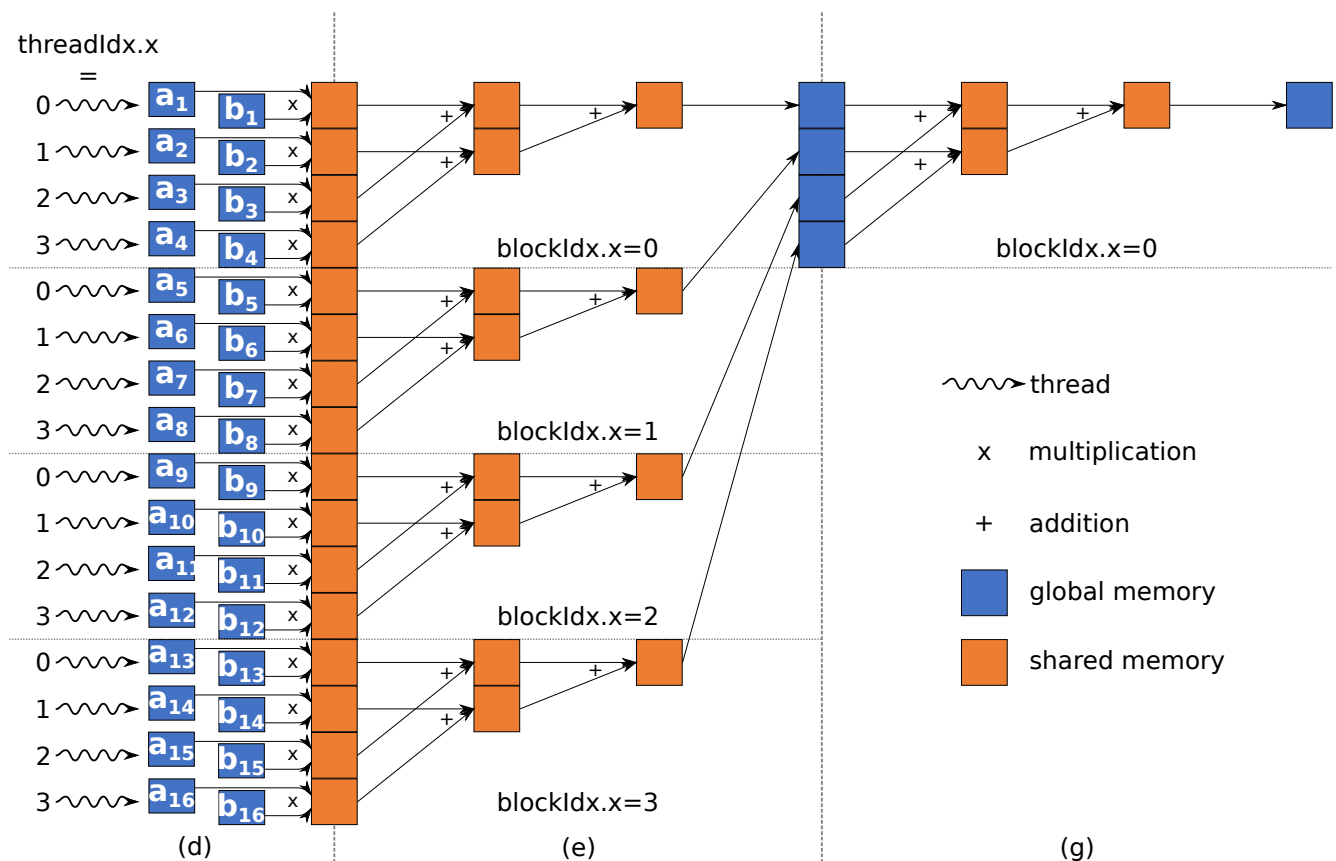
(a) Open VS code and install the *Remote - SSH* and the *vscode-cudacpp* extension.

(b) Use VS code (*REMOTE EXPLORER*) to connect to the GPU aware virtual machine with the IP 3.65.153.170 and your received user name and password.

- First, we have to ensure that all files and folders exist:
- `mkdir -p ~/.ssh`

Computer architecture
Exercise sheet 11
SoSe 2022   Prof. Dr. Florian Künzner

Rosenheim
Technical University
of Applied Sciences

- `touch ~/.ssh/config`
- Then, do the connection step with: `ssh <username>@3.65.153.170`
- Choose: `/home/<username>/.ssh/config` as the destination for the config settings
- Then you can connect to the new ssh target with: `Connect to host in current view` (right click on ssh target)
- Enter your PW
- After that, open the folder `/home/<username>/dotproduct` within VS code.

(c) Inspect and build the CUDA template code for the dot product.

- The CUDA template code is located in: `/home/<your_user>/dotproduct`
- Because `cmake` is used as the build system, you have to prepare the build with (only has to be executed once):

```
1  mkdir build
2  cd build
3  cmake ..
```

  This generates a `Makefile` that can be used from now on.
- Build the code with `make`
- You can execute the dot product executable with: `../bin/dotproduct`

(d) As a first step, implement the **element-wise multiplication** $a_i * b_i$ in kernel `dotProduct()`. Each thread should perform one multiplication, e.g. the i-th thread performs the i-th product. Store the result to shared memory.

   If you did a proper implementation, the application prints -0.0491419 as intermediate result to the console.

(e) As a second step, **sum up all products** of a block by extending kernel `dotProduct()`. Do this in a tree-like manner as depicted in the figure: A thread takes two elements from shared memory to build a thread-local partial sum and store this result to shared memory.
   With each step, more and more threads become „idle". Store the final result (this is one single element per block) to global memory.
   Reminder 1: Each thread of a block has access to each element of the shared memory of a block.
   Reminder 2: `__syncthreads()` in kernel code synchronizes all threads of a block.

   If you did a proper implementation, the application prints 7.27795 as intermediate result to the console.

(f) This sub-task does not require any coding. It serves as a check if the implementation you did so far works properly. Furthermore, it demonstrates an application of the dot product.

   A matrix-vector product can be considered as a subsequent execution of dot products. The dot product of the i-th row of the matrix with the vector gives the i-th element of the result vector. One application of the matrix-vector product is the rotation of vectors:
   If multiplied with a rotation matrix (see `https://en.wikipedia.org/wiki/Rotation_matrix` for more information), a vector gets rotated in space. It's **neither** necessary to understand how rotation matrices work **nor** do they have to be implemented.

   The final piece of code of the skeleton utilizes the `dotProduct()` kernel to implement a matrix-vector product to rotate a vector in 3D space around the x-axis. Check if your implementation so far works properly by altering the rotation angle `alpha` (in radians) and or the vector to rotate `alpha`.

Computer architecture
Exercise sheet 11
SoSe 2022   Prof. Dr. Florian Künzner

Rosenheim
Technical University
of Applied Sciences

(g) This sub-task does not require any coding. This time, you should conclude from a piece of kernel code to its functionality. Uncomment the body of kernel `accumulatePartialDotProducts()`. The application prints -129.573 as final result to the console if you did this properly.

Read the uncommented body of kernel `accumulatePartialDotProducts()` and try to follow what it's doing. Consider how this kernel is called from host code (number of threads per block, number of blocks per grid). The right part of the figure illustrates the behavior of the code to analyze. Explain the functionality added by the uncommented code.

(h) Explain why the execution of a second kernel is necessary to implement the functionality of the previous sub-task. Why can't this be done by `dotProduct()` even the block-local dot products are in global memory (randomly accessible by each thread of each block)?