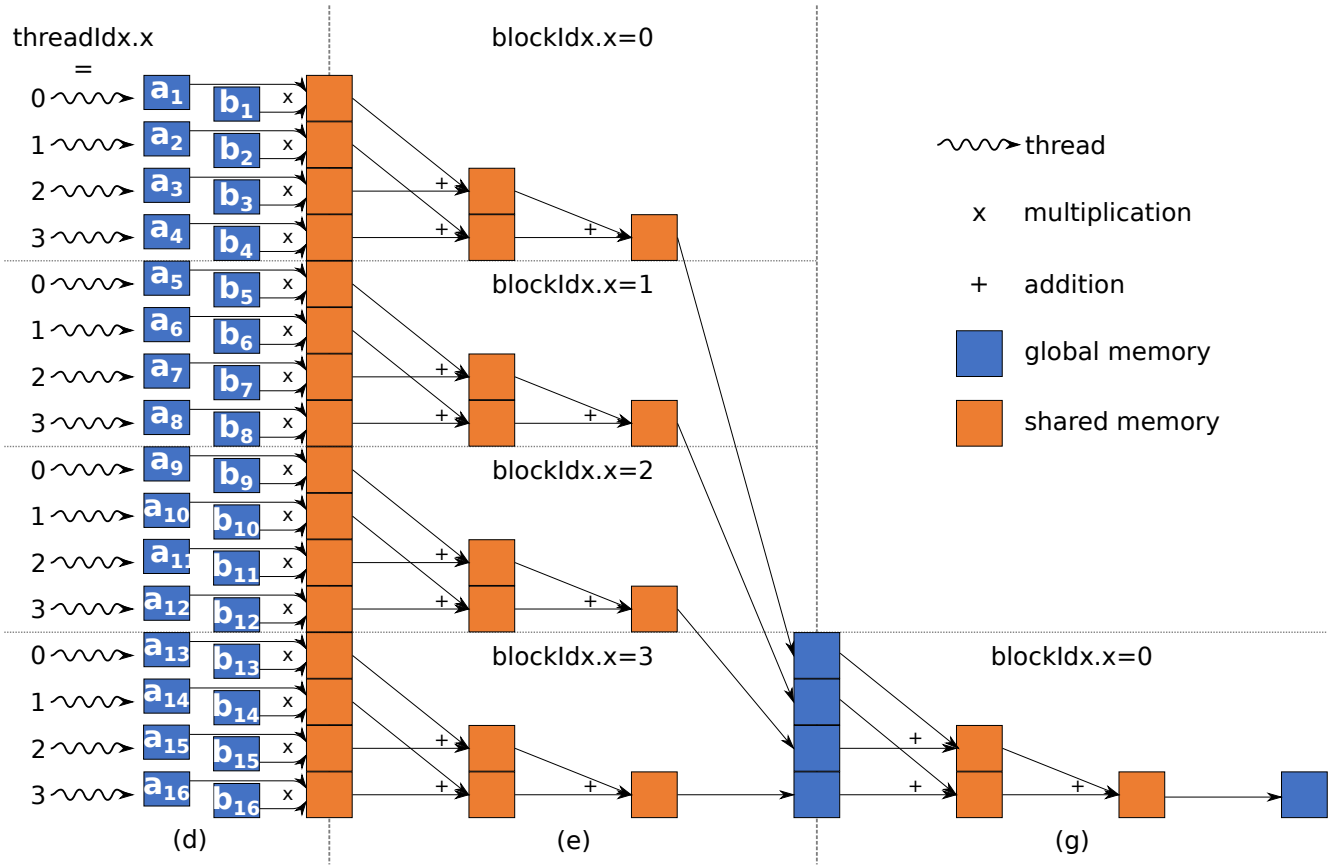
Exercise sheet 13 – GPU

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 13.1: 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 3.66.106.138 and your received user name and password.

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

- `touch ~/.ssh/config`
- Then, do the connection step with: `ssh <username>@3.66.106.138`
- 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):

```
mkdir build
cd build
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.

Proposal for solution:

```
// (d) Compute thread-local product
scratchpad[threadIdx.x] = in_a[globalThreadIdx] * in_b[globalThreadIdx];
```

(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.

Proposal for solution:

```
// (e) Perform block-local reduction step
int numOfActiveThreads = blockDim.x>>1;

while (numOfActiveThreads > 0)
{
    __syncthreads();

    if (threadIdx.x < numOfActiveThreads)
    {
```

```
47          scratchpad[threadIdx.x] += scratchpad[threadIdx.x + numOfActiveThreads];
48      }
49
50      numOfActiveThreads = numOfActiveThreads>>1;
51  }
```

(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`.

(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.

Proposal for solution: The problem so far is that we have „block-local" dot products: Since we decided each thread processes one element of the input vectors, we require multiple blocks if the vector length n is larger than the maximum number of threads per block.
Hence, the execution of a second kernel `accumulatePartialDotProducts()` is necessary to sum up the block-local dot products. This is done in a tree-like manner as before.

(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)?

Proposal for solution: There's no mechanism in device code to synchronize threads among multiple blocks. `__syncthreads()` only synchronizes the threads within the same block. Without such a global synchronization mechanism, it can't be guaranteed that all block-local dot products are ready before they are accumulated across multiple blocks.
Such a global synchronization is achieved by finishing one kernel and invoking another kernel. Since data in shared memory persists only during the execution of one and the same kernel, intermediate results for the next kernel execution have to be stored to global memory.