3. Fundamentals of GPU Programming

Gündisch Philipp, Holzinger Philipp

1. Basic GPU Architecture
    1. GPU Architecture
    2. System Integration
2. GPU Programming Model
    1. General Concept
    2. OpenCL
    3. Cuda

3.1 Basic GPU Architecture

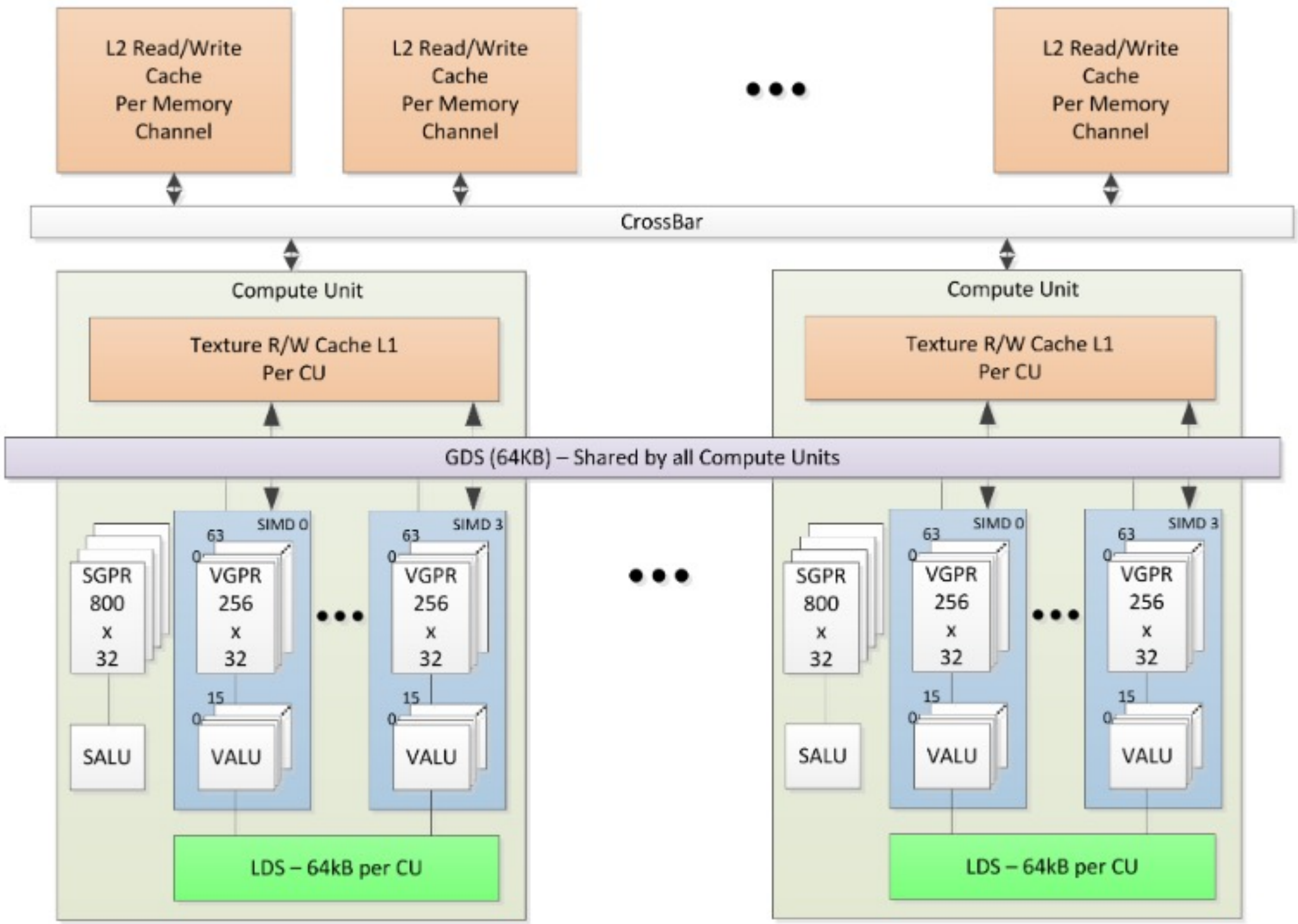
Main differences of a GPU to a CPU:

- Plenty ALUs, less control flow prediction
- GPU highly optimized for vector processing – Single Instruction, Multiple Threads (**SIMT**) execution model
- High SMT factor to hide I/O latencies
- A certain number of cores (**wavefront or warp**) ALWAYS execute the same instruction (with different data)
- In case of diverging branches both sides must be executed one after another (**thread divergence**)

3.1.1 GPU Architecture

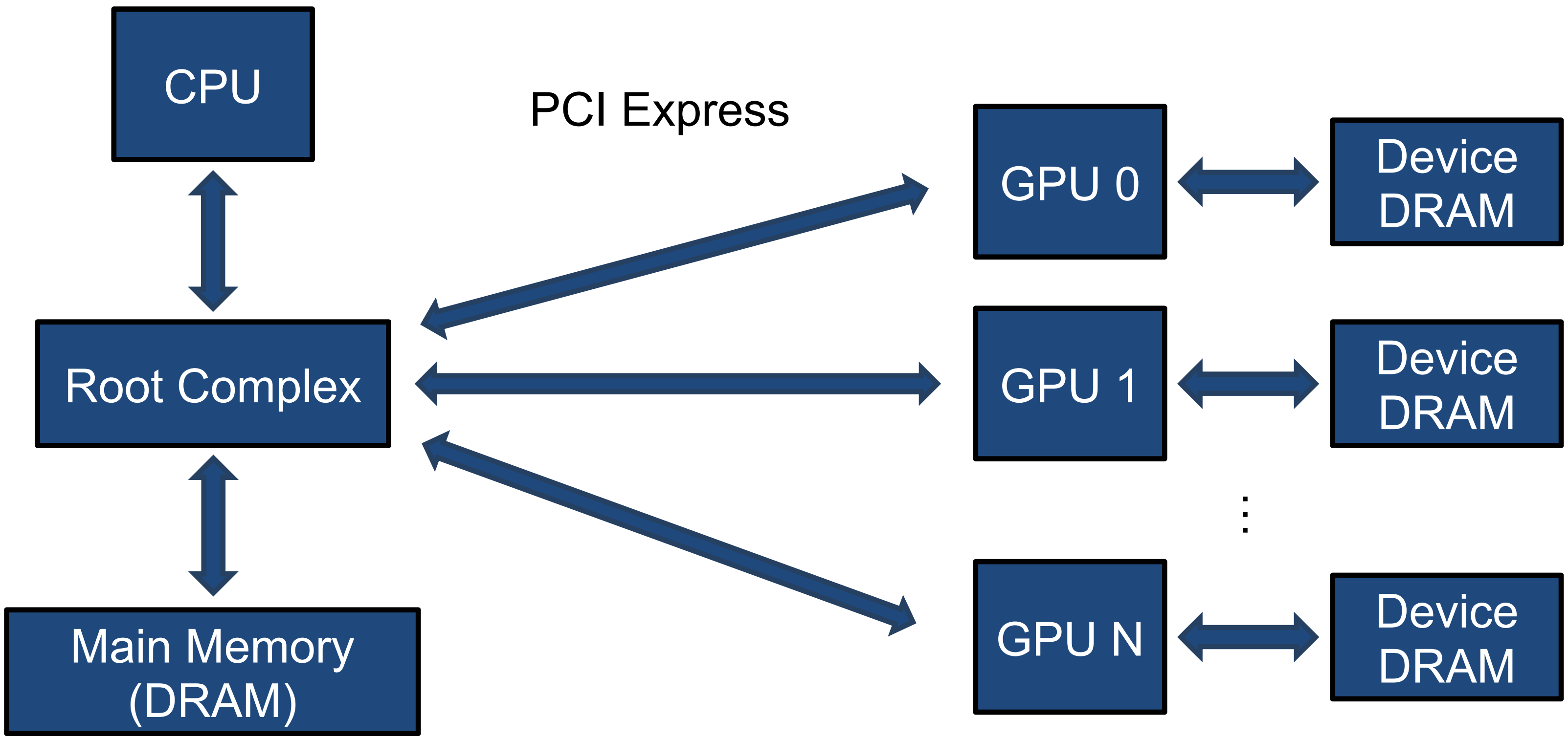
https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf

3.1.1 GPU Architecture



https://developer.amd.com/wp-content/resources/Vega_Shader_ISA_28July2017.pdf

3.1.2 System Integration



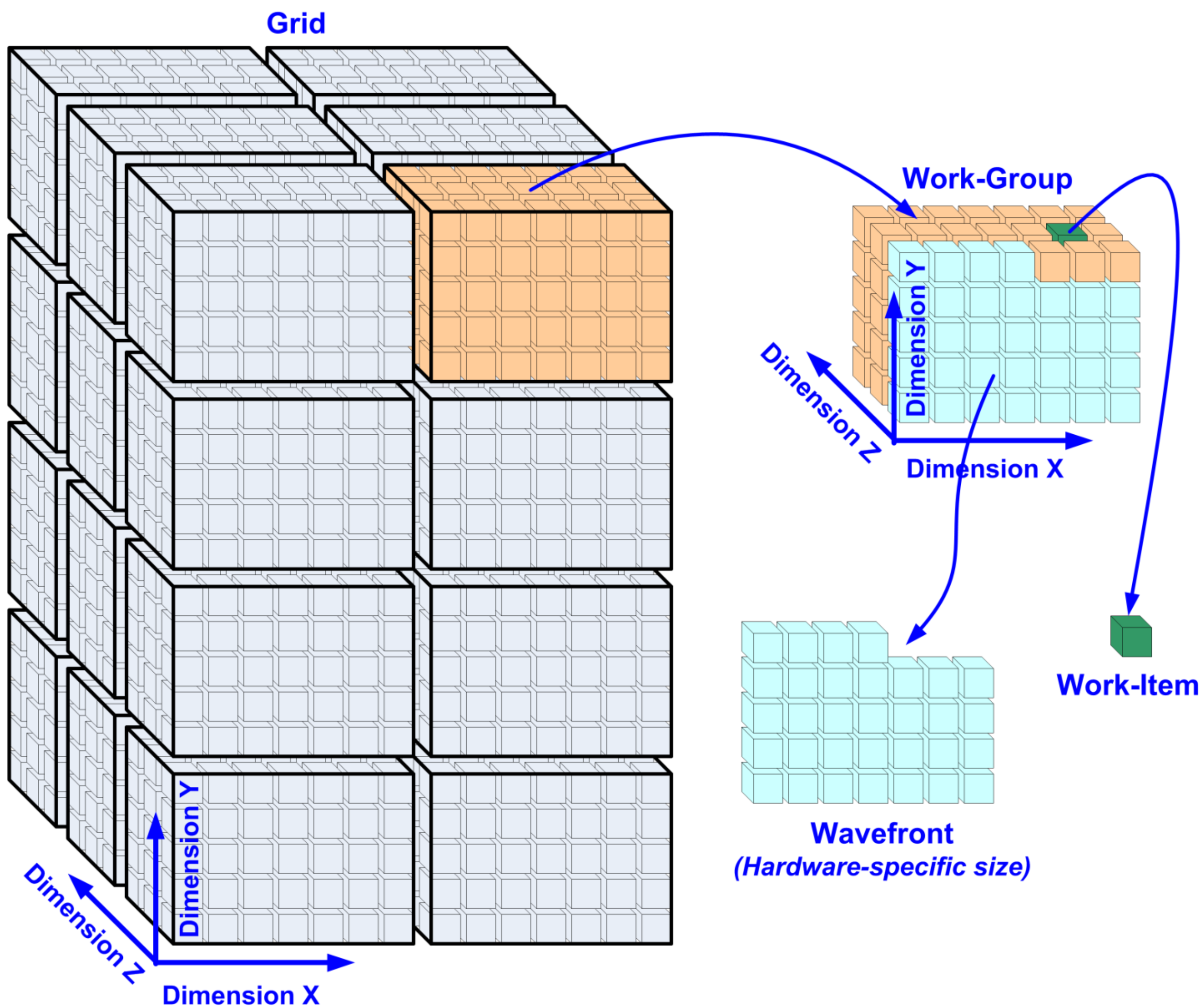
3.2 GPU Programming Model

Typical GPU programming workflow:

- Preparations:
  - Virtually divide task in a **grid** structure. Each item will be executed by a separate thread
  - Write a **kernel** which describes what a single thread will do
  - The kernel will be executed for every **work-item** in the grid

- Execution:
  - Allocate memory buffers for the data on the GPU
  - Copy data from host memory (CPU RAM) to device memory (GPU VRAM)
  - Launch kernel on the GPU (asynchronously to CPU)
  - (Wait for kernel execution)
  - Copy results from device memory back to host memory

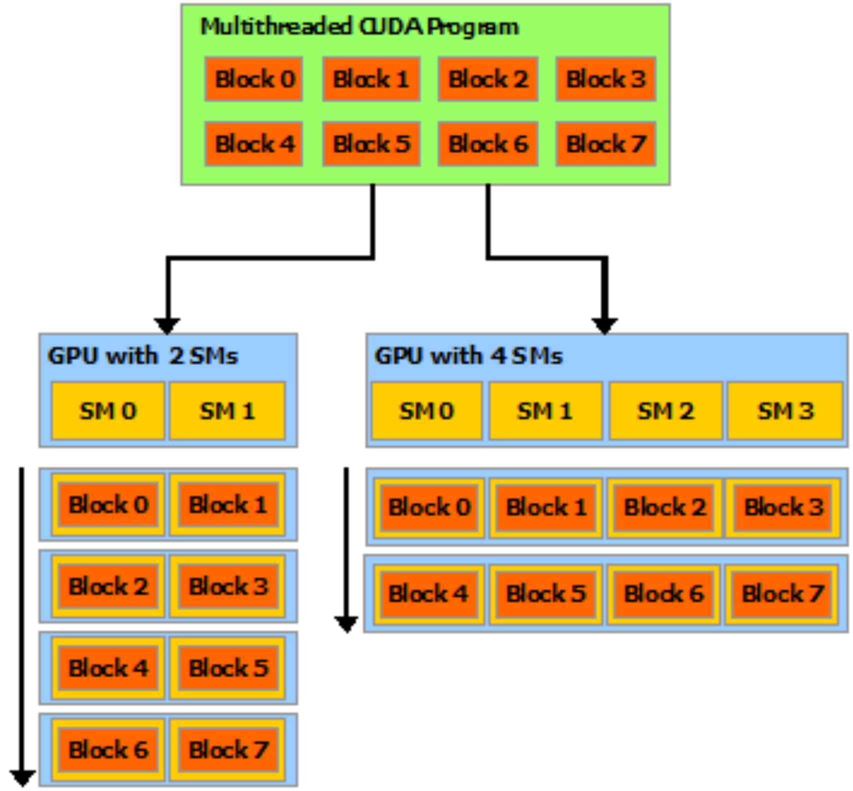
3.2.1 General Concept



- http://www.hsafoundation.com/standards/

The purpose of thread blocks

- despite the distinct indexing of threads, it is not possible to decide where it will be executed on hardware
- **blocks** always executed on the same streaming multiprocessor
→ access to common resources and exchange of data via shared memory

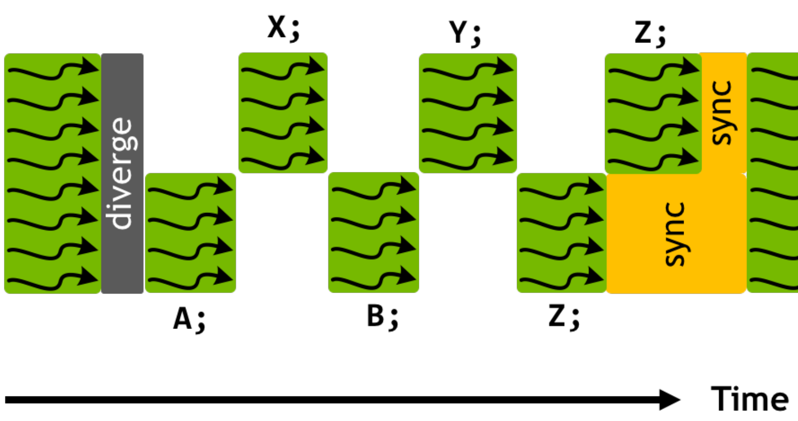


Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Warps / Wavefronts – the smallest unit

- On Nvidia GPUs groups of 32 threads are arranged into warps
- Threads inside the warp work according to SIMD model
- Branches inside kernels possible, but thread divergence affects effectiveness of parallelism
- Shared program counter!!!

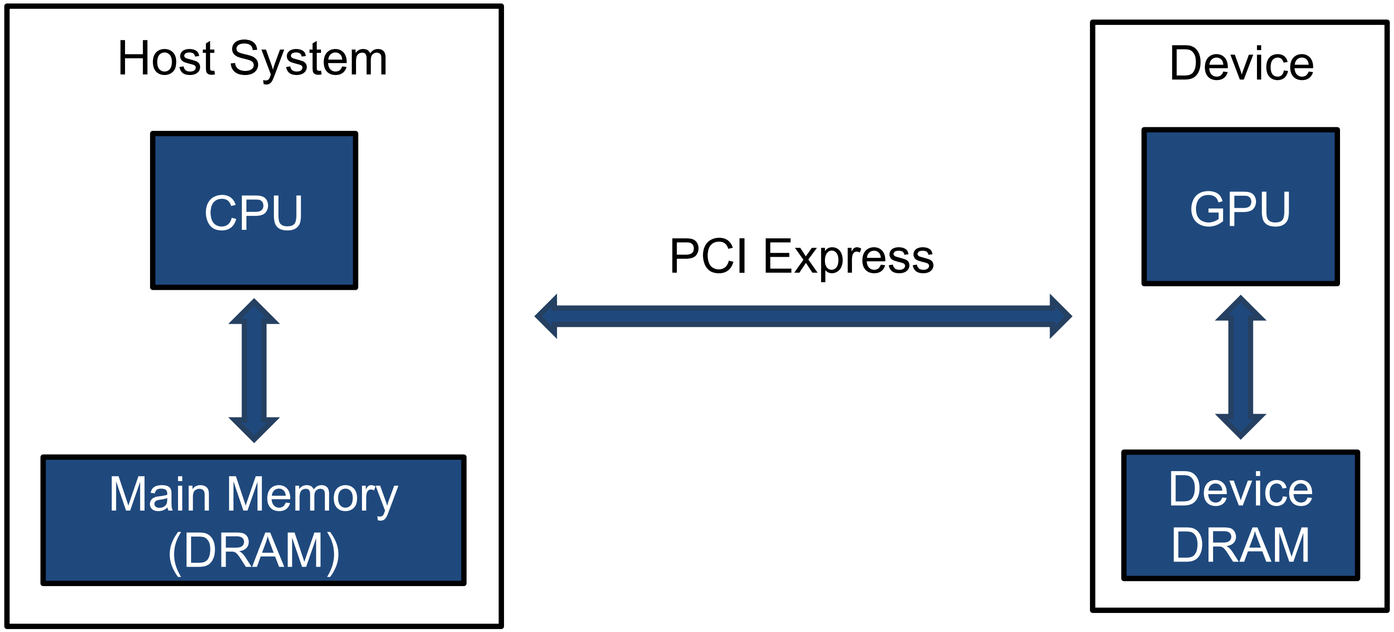
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```



Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

3.2.1 General Concept

Initial situation:

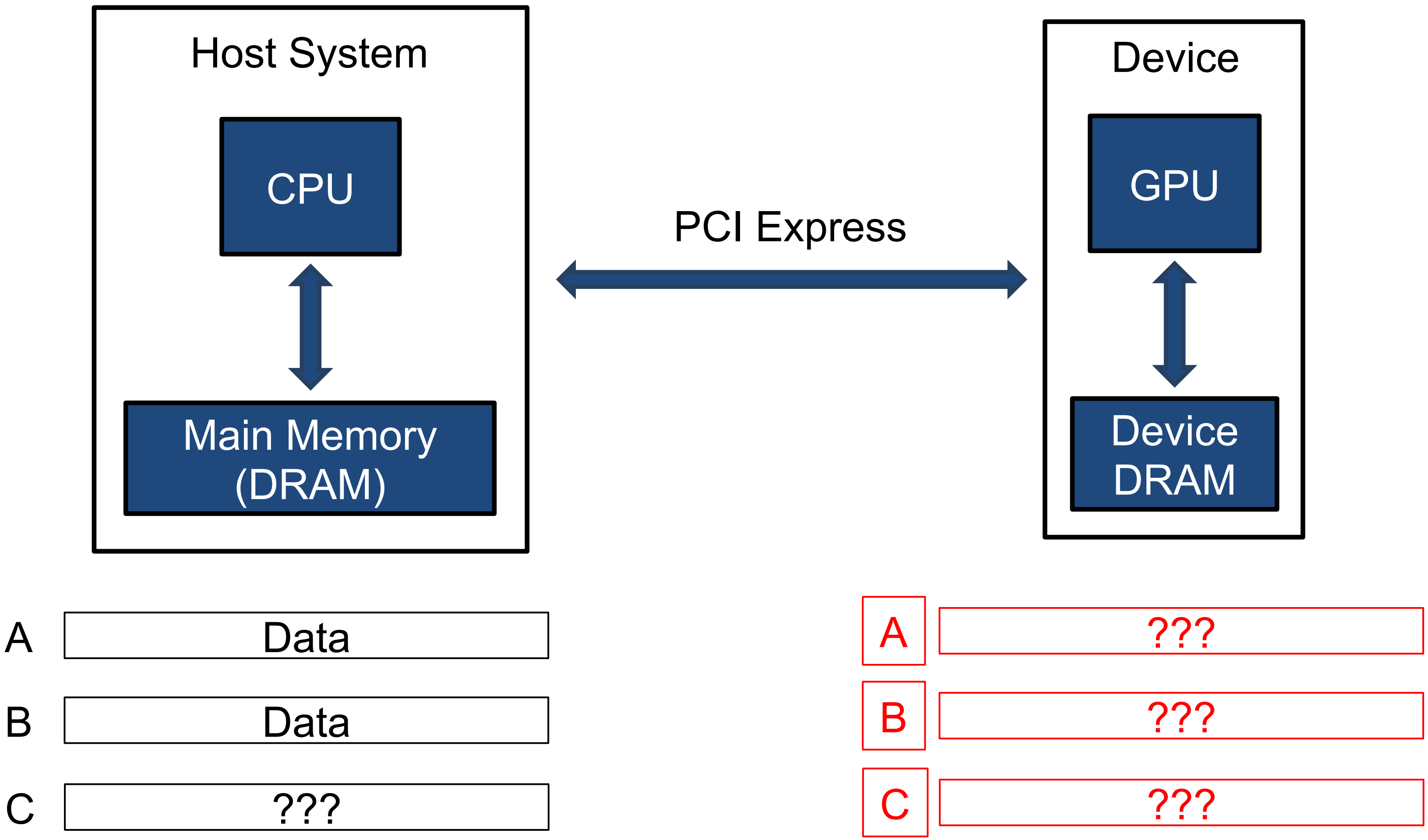


A Data

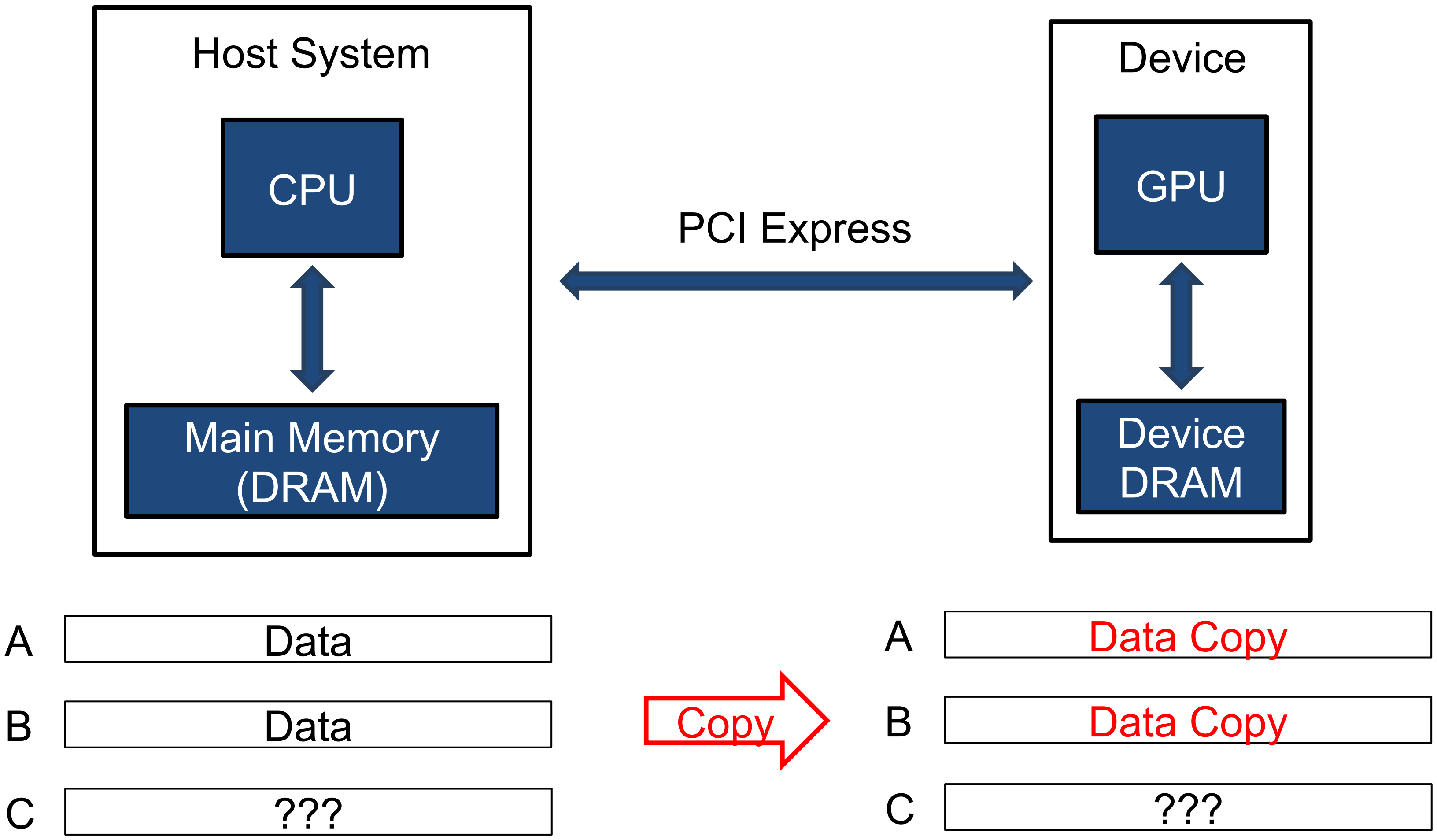
B Data

C ???

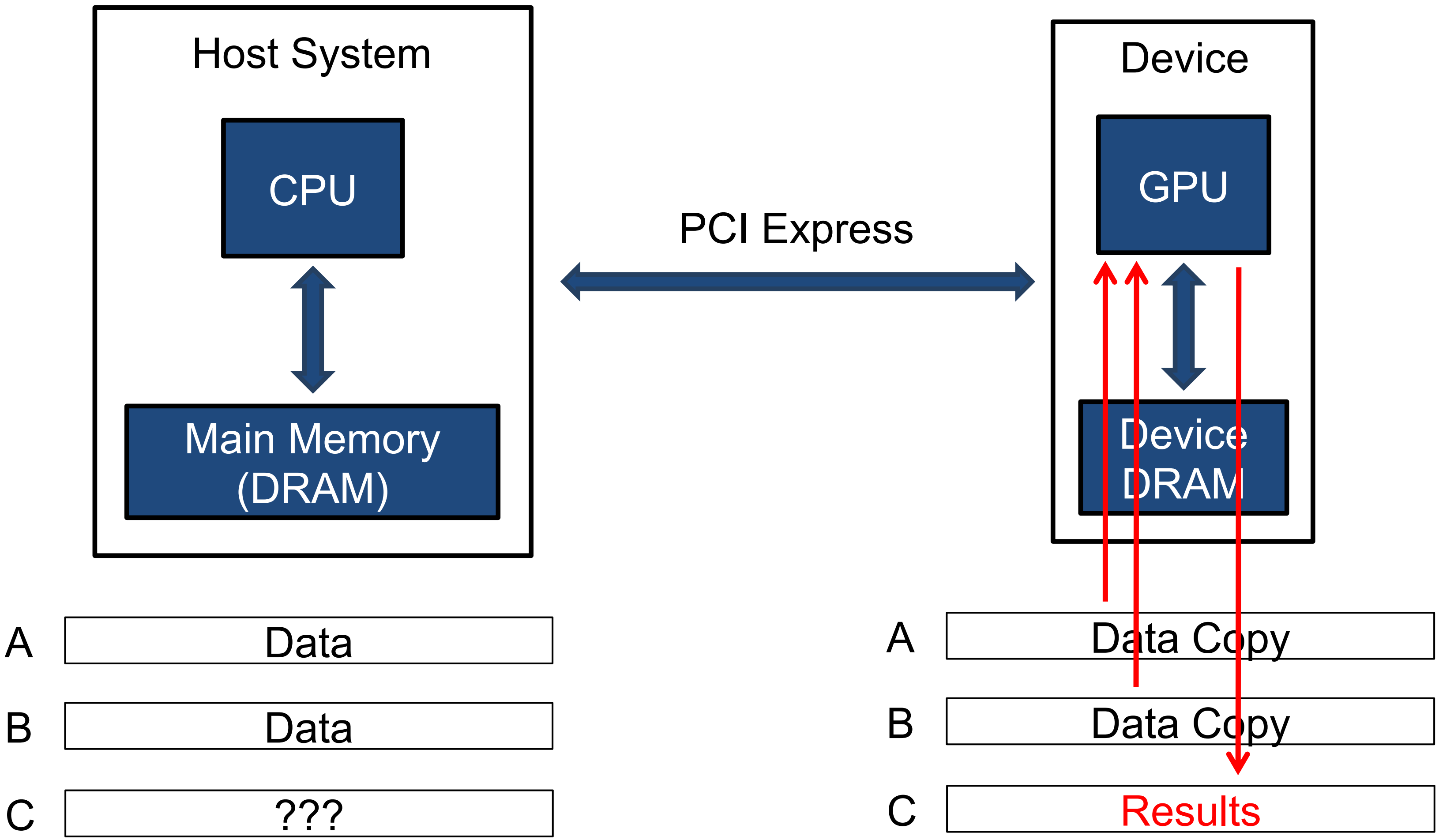
1. Allocate memory buffers:



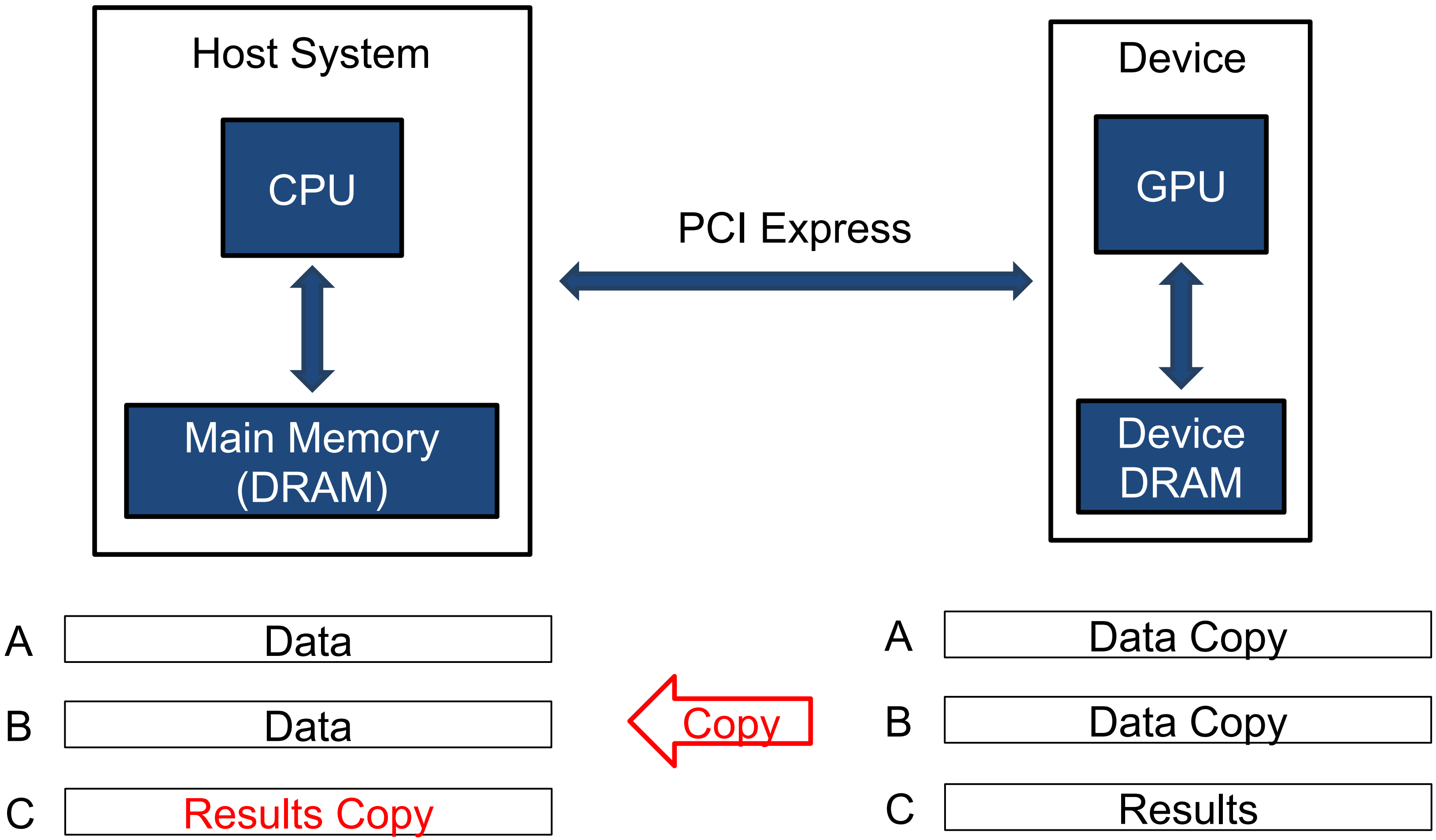
2. Copy data to GPU:



3. Calculate results on GPU:



4. Copy results to CPU:



Simple OpenCL Vector Add kernel:

```
// this is a kernel function which uses the float buffers a, b and c
__kernel void vadd(__global float *a, __global float *b, __global const float *c){

    // get index in X dimension of this work-item (thread)
    const unsigned int idx = get_global_id(0);

    // compute addition for this work-item in the grid
    c[idx] = a[idx] + b[idx];
}
```

OpenCL kernels can mostly be written in standard C, but:

- Some special memory annotations (__global, __local, __private, …)
- no main function, __kernel describes entry functions
- No function pointers or recursion
- No access to C standard library, however there are built-in math functions

Query platform and device information:

```
std::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);

// iterate over all platforms (e.g. AMD)
for(auto &platform : platforms){
    std::cout << platform.getInfo<CL_PLATFORM_VERSION>() << std::endl;
    std::cout << platform.getInfo<CL_PLATFORM_NAME>() << std::endl;
    std::cout << platform.getInfo<CL_PLATFORM_VENDOR>() << std::endl << std::endl;

    std::vector<cl::Device> devices;
    platform.getDevices(CL_DEVICE_TYPE_ALL, &devices);

    // iterate over all devices of a platform (e.g. 4 GPUs)
    for(auto &device : devices){
        // with this information store the device and platform you want to use
        std::cout << device.getInfo<CL_DEVICE_NAME>() << std::endl;
        std::cout << device.getInfo<CL_DEVICE_VENDOR>() << std::endl;
        std::cout << device.getInfo<CL_DEVICE_VERSION>() << std::endl << std::endl;
    }
}
```

Setup GPU on host CPU:

```
cl_int err;
cl::Context context(devices);

// read .cl file with the kernel sources and compile them for the device
std::ifstream filein(„vadd_kernel.cl");
std::istreambuf_iterator<char> begin(filein), end;
std::string code(begin,end);

cl::Program::Sources sources;
sources.push_back(std::make_pair(code.c_str(), code.length()+1));

program = cl::Program(context, sources, &err);
program.build(devices);

// create a command queue for the GPU
cl::CommandQueue cmdqueue(context, devices[0], 0, &err);

// create memory buffers in the GPU DRAM
cl::Buffer gpu_a(context, CL_MEM_READ_ONLY, sizeof(float)*vector_size);
cl::Buffer gpu_b(context, CL_MEM_READ_ONLY, sizeof(float)*vector_size);
cl::Buffer gpu_c(context, CL_MEM_WRITE_ONLY, sizeof(float)*vector_size);
```

Copy data and execute the kernel on the GPU:

```
// copy data to GPU
cmdqueue.enqueueWriteBuffer(gpu_a, CL_TRUE, 0, sizeof(float)*vector_size, cpu_a);
cmdqueue.enqueueWriteBuffer(gpu_b, CL_TRUE, 0, sizeof(float)*vectot_size, cpu_b);

// prepare kernel execution by setting kernel arguments
cl::Kernel kernel(program, „vadd", &err);
kernel.setArg(0, gpu_a);
kernel.setArg(1, gpu_b);
kernel.setArg(2, gpu_c);

// execute the kernel
cmdqueue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(vector_size), cl::NullRange);
cmdqueue.finish();

// copy results back to main memory
cmdqueue.enqueueReadBuffer(gpu_c, CL_TRUE, 0, sizeof(float)*vector_size, cpu_c);
```

- https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf

CUDA Example – Vector Add

Create host vectors and initialize them

```
int main(void) {
    // vector size
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    // Allocate the host input vectors A, B and C
    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i)
    {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }
```

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

CUDA Example – Vector Add

Create allocate device memory and copy the data from host to device

```
// Allocate the device input vectors A, B and C
float *d_A = nullptr;
cudaMalloc((void **)&d_A, size);
float *d_B = nullptr;
cudaMalloc((void **)&d_B, size);
float *d_C = NULL;
cudaMalloc((void **)&d_C, size);

// Copy the host input vectors A and B in host memory to
// the device input vectors in device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

CUDA Example – Vector Add

launch the vector-add kernel and copy the result to host

```
// Launch the Vector Add CUDA Kernel
int threadsPerBlock = 256;
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

// Copy the device result vector in device memory to the host result vector
// in host memory.
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

CUDA Example – Vector Add

vector-add kernel

```
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html