Hardware für Eingebettete Systeme

Lab 9: GPU Programming with CUDA (1/2)

Prof. Dr. Mario Porrmann
M.Sc. Marc Rothmann
B.Sc. Philipp Gehricke

1 Vector Addition in CUDA

In this lab you are going to program in CUDA and utilize the GPU on the Jetson Nano. The CUDA API will help you take advantage of the highly parallel architecture of the GPU. As CPUs and GPUs became parallel systems with an increasing number of cores, the challenge is to develop application software that scales its parallelism to these cores. CUDA tries to overcome this challenge by providing abstractions in form of C/C++ language extensions.
A (NVIDIA) GPU contains multiple Streaming Multiprocessors (SMs) which can independently execute blocks of threads. Each SM contains multiple Cores for execution. A GPU with a higher number of SMs can therefore execute a parallelized program faster than a GPU with fewer SMs. (Assumed the compared GPUs share the same SM architecture)
For more information about the CUDA API visit:
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

With the program deviceQuery.cpp you can show some interesting device information of the CUDA capable GPU. Compile and run the program with:

```
cd /usr/local/cuda-10.2/samples/1_Utilities/deviceQuery/
sudo make
./deviceQuery
```

How many Streaming Multiprocessors and how many CUDA Cores does the Jetson Nano have?

Copy the files `vector_addition.cu` and `memCopy_vec_add.cu` located inside the zip archive from your Host-Computer to your Jetson Nano. Run 1. and 3. on your Jetson Nano and 2. on the Host:

```
1.) mkdir jetson_lab
2.) scp jetson_lab.zip username@ipv4_address:~/jetson_lab/
3.) unzip jetson_lab/jetson_lab.zip
```

Verify that your copy was successfull.

Take a look at `vector_addition.cu`. This program executes a simple vector addition in C code on the CPU. You are now going to convert this vanilla C++ program into a CUDA C++ program (.cu suffix) which executes the vector addition on the GPU.
You can already compile and run the program with:

```
nvcc vector_addition.cu
./a.out
```

With jtop running in a sepperate (tmux) pane, you can take a look at the CPU and GPU utilization, as well as, memory usage. Start jtop followed by executing your binary.

```
jtop
```

You can let jtop running for the rest of this lab.
`nvprof` shows you a profile of the execution of your program. Start it with:

```
sudo nvprof ./a.out
```

The output should not show you anything, because you did not run anything on the GPU yet. Also keep in mind, that "Time on GPU" is still potraying the runtime on the CPU, since

you did not implement anything yet.
You are now going to implement the vector addition in CUDA for the embedded GPU on the Jetson Nano step-by-step. The libraries needed for this tasks are already imported.

1. Kernels: Kernels are special functions and a C++ extension, that, when called, are executed N times in parallel by N threads. These kernels are defined by using the `__global__` declaration specifier. The number of times a kernel is called is specified by the `<<<...>>>` execution configuration.
   Convert the `vector_add_GPU` function to a GPU kernel:

```
__global__ void vector_add_GPU(float *outGPU, float *a, float *b, int n)
{
       for(int i = 0; i < n; i++){
                 outGPU[i] = a[i] + b[i];
            }
    }
```

   And also change the call of the function in main() to a kernel call:

```
vector_add_GPU<<<1,1>>>(outGPU, a, b, N);
cudaDeviceSynchronize();
```

   With `cudaDeviceSynchronize()` you are waiting for all threads to finish computing on the GPU. This is done to ensure all data being processed before accessed by the Host CPU.

2. Memory allocation: Usually when dealing with GPGPUs you would have to allocate memory for your operations using the CPU with `malloc()` and also for your NVIDIA GPU using `cudaMalloc()`. You then would have to transfer your data from the so called host (CPU) memory to the device (GPU) memory. After this memory management, you could perform kernel tasks on the GPU SMs and memory. Since you are using a Jetson Nano Development Board, you do not have seperate memory for the CPU and GPU. The CPU and GPU on the Jetson Nano are using the same DRAM and therefore Unified Memory. While you could, without a big impact on the performance, implement the memory management the usual way, it is also possible to use the `cudaMallocManaged()` method. It automates the above described process of the memory transfer/copy. It returns a universal pointer which can be accessed on both CPU and GPU.
   There is also the `zeroCopy()` method which is similar to the managed method. You can do some research, if you want to use it.
   `memCopy_vec_add.cu` contains the solution to this exercise using manual copying and transferring memory between CPU and GPU. This will also work on the Jetson Nano.

   Change the allocation of the vectors a, b and outGPU, so that they can be accessed by the GPU and managed by the CUDA API:

```
// Allocate memory
outCPU = (float*)malloc(sizeof(float) * N);
cudaMallocManaged((void**)&a, sizeof(float) * N);
```

```
cudaMallocManaged((void**)&b, sizeof(float) * N);
cudaMallocManaged((void**)&outGPU, sizeof(float) * N);
```

And free the allocated memory with:

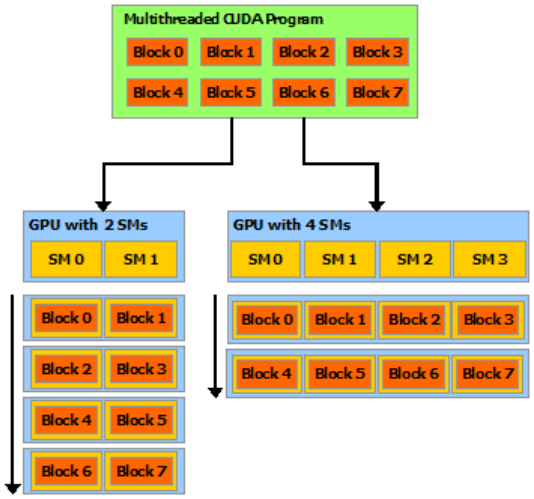
```
cudaFree(a);
cudaFree(b);
cudaFree(outGPU);
free(outCPU);
```

3. Your implementation can now run on the GPU. Test the performance of your compiled program. Again, use `nvprof` to show the time spent on CPU and GPU for the operations you are using. Run it with:

```
sudo nvprof ./a.out
```

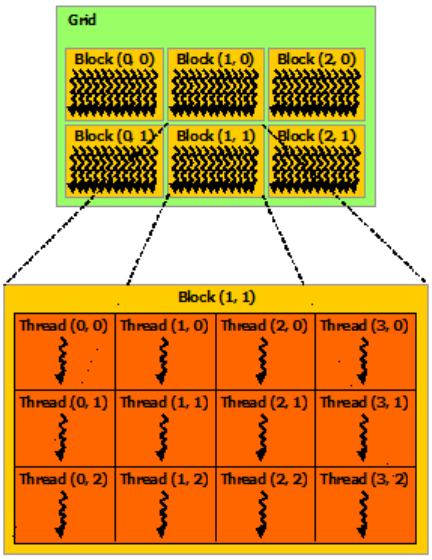
How well does your implementation perform? You might notice a bad performance regarding the execution time on the GPU. This is, because we did not add any parallelism to the execution yet.

4. ThreadBlocks and Grids: Your execution configuration with `<<<1,1>>>` denotes, that you are running the vector addition with only one thread, which isn't parallel at all. You need to modify the execution configuration and also the kernel to run the addition in parallel threads.



When it comes to CUDA programming you are provided with extensions and methods which provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data paralellism and task parallelism. With those, you can partition your task into coarse sub-problems that can then be solved independently in parallel by blocks of threads. Each sub-problem can then be partitioned into finer pieces that are solved cooperatively in parallel by all threads in the block. The CUDA API allows each block to be scheduled on any of the Streaming Multiprocessors in any order, concurrently or sequentially. Compiled CUDA code can therefore automatically scale depending on the utilized (NVIDIA) GPU platform.

The programmer divides the problem into threads, threads into thread blocks and thread blocks into grids. The compute work distributor spreads the thread blocks onto the Streaming Multiprocessors (SM). The resources for the thread block are allocated and the threads are divided into warps (usually 32 Threads = 1 Warp) after a thread block is distributed to a SM. The warps will then be dispatched to execution units.
A clarification: A thread block will be assigned to only one SM. But one SM can execute multiple blocks at the same time. Since the blocks are divided into warps which contain 32 threads, it is adviseable to choose a block size which is divisible by 32. Otherwise the scheduler would handle a warp with less than 32 threads the same way as if it would have 32 threads. The available hardware could then not be fully utilized.



You need a way to tell a thread what it should do. Or more specifically, assign all sub-task to unique threads. `threadIdx` is a vector with three components. This vector can be accessed in a Kernel function. It represents a one-, two- or three-dimensional thread index. This means you can work with one-, two- or three-dimensional block of threads (thread block). In this lab you only need to care about one-dimensional thread blocks, but you should consider higher dimensional thread blocks in the next lab exercise. There is a limit for the threads per block, since all threads of a block are expected to work on the same processor core and must share the limited ressources. But a kernel can be executed by multiple thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. A thread block itself is also organized into a one-, two- or three-dimensional way, called grid. The size of the grid depends on the size of the data being processed. Similar to the thread vectors, each block in the grid can be identified by a one-, two- or three-dimensional index through the `blockIdx` variable. The dimension of the thread block is accessible through the `blockDim` variable.

Change your kernel to the following:

```
__global__ void vector_add_GPU(float *outGPU, float *a, float *b, int n)
{


    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (thread_id < n)
    {
        outGPU[thread_id] = a[thread_id] + b[thread_id];
    }

}
```

`blockIdx.x * blockDim.x + threadIdx.x` gives you a unique thread ID. Each thread entering the kernel will compute its intended addition.
Now that you've prepared your kernel for parallel execution, the next step is to adjust the exuction configuration which is denoted by `<<<M, T>>>`. M indicates that a kernel launches with M thread blocks and each thread block has T threads. Since the amount of threads per block and the amount of blocks is dictated by the size of the given data, you need write an execution configuration which considers this characteristic.
In your code replace the execution configuration appropiately by the following lines:

```
// Call the Kernel function
int block_size = 32;
int grid_size = ((N + block_size -1) / block_size);
std::chrono::steady_clock::time_point begin2 = std::chrono::steady_clock::now();
vector_add_GPU<<<grid_size,block_size>>>(outGPU, a, b, N);
cudaDeviceSynchronize();
std::chrono::steady_clock::time_point end2 = std::chrono::steady_clock::now();
```

With the block size you set the amount of threads per block. The grid size is then calculated accordingly. This is done to make sure that enough blocks are available for the data being processed.

5. You may found out in the beginning of this lab that the Jetson Nano features only one Streaming Multiprocessor which contains 128 CUDA Cores. This concludes that big thread blocks will be executed sequentially after one another. But still, threads within a thread block (or warp) can be executed in parallel on the Streaming Multiprocessor.

   Compile your CUDA program and do some testing and benchmarks. Use `nvprof` and tegrastats as well as jtop to take a quick look at energy consumption, performance regarding time and memory management. Change the amount of threads per block and try to find a good or even optimal solution. How many threads per block are possible? Try various sizes. You can also set the sizes of the vectors to some higher value e.g. 100000000.

6. (Optional) You now know how to add two vectors on the GPU using CUDA. Perform a vector multiplication on the Jetson Nano.