GCA Session 2 - Lab Report

Maxime Rombaut
Master of Electronics and ICT Engineering Technology

Abstract: In this report, I present my work on developing a CUDA-based application where an array of a specified amount is generated. I then implement three different ways to calculate the maximum value in that array. The first approach uses the CPU and is sequential. The second and third approach make use of parallelisation. **Keywords:** CUDA, GPU, CPU, C++

1 Sequential CPU approach

The first thing that happens in the program is the creation of an array with n random values. The idea is that we find the maximum value in this array with three different approaches. In this first approach, we use a standard for loop that loops over all the elements in the array and saves the value of an element if it is larger than the previous largest value.

2 Atomic approach

In this approach we make use of the parallelization that the GPU can give us. We first allocate the required memory for the parameters we will give to the kernels and then copy the parameters of the host to the device.
After this the kernelfunction will be called with the parameters and with an amount of blocks and threads. We use 1024 threads if the amount of items in the array is larger then 1024, else we use the amount of elements. The amount of blocks we use is calculated via the formula: $(int)fmax(ceil(numElements/1024), 1)$.
In the kernelfunction we calculate the id of the current thread via $blockDim.x * blockIdx.x + threadIdx.x$ and use this to get the appropriate value from the array that was given as a parameter.
Via the $atomicMax(max, a[id])$ function we compare the values of max and a[id] and save the maximum value on the address of max. This happens atomically so no other threads can access the max value on the same time. This avoids race conditions. If this has happened on every thread, we can copy the value out of the max address from the device to the host.

3 Reduction approach

As in the previous approach, we allocate some memory for the parameters and copy the input from the host to the device.
We now want to try to divide the work of the threads as follows: Imagine the input array has 8 elements. Thread 0 should calculate the max of the values on index 0 and 1 of the array, thread 1 of 2 and 3 and so on. This will be done by an amount of threads that is equal to half the size of input array.
We repeat this process but the "useful" elements are now only in the first half of the array, so we only need to do this on the first half of the array. This repeats until there is only 1 interesting value left, the maximum value. This value is copied to the maxValue variable.
To make sure that the threads don't disturb each other the $\_\_syncthreads$ function is used at the end of every process mentioned above. This way, the threads need to wait on the others before going on with the calculations.
This, however, only works when the maximum amount of elements in the array is 2048. If it is bigger, we will need more than one block to do the calculations. We could use an atomicMax to try and solve this, but we have chosen to work with an offset.
We call the kernel multiple times, depending on the amount of elements. If the amount is 2048 or lower we only need to call it once and calculate the appropriate amount of threads needed (half of the amount). If it is larger, we use 1024. If the amount of elements is larger than 2048 not a multiple of that, for example 2056, we call the kernel once with 1024 threads and then a second time with 4 threads. The offset that will be given in this case is 0 for the first call and 2048 for the second call.
This approach works most of the time but in some cases the returned value isn't the correct one. To try and solve this we have place a $cudaDeviceSynchronize()$ after each kernel call to synchronize the device but this didn't solve it completely. After many different solution attempts we still haven't found the reason this has happened and because it doesn't happen often we have decided that it works good enough for this lab.

4 Comparison

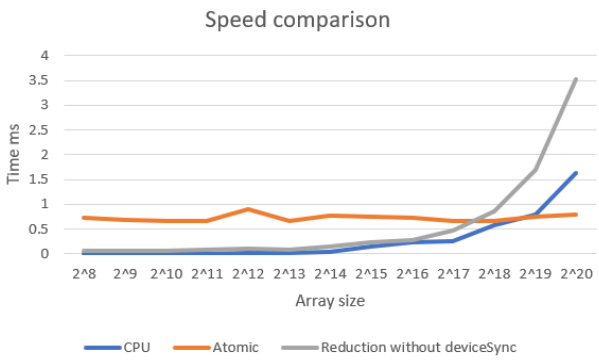

Figure 1: *Execution speed comparison between the CPU, atomic and reduction approach*

We can see that the CPU is in most cases the fastest. When reaching around 2$\hat{1}$9 items in the array, the atomic approach will start to get faster than the CPU. The difference between these 2 increases when using bigger arrays, with the atomic

approach being the faster one. The reduction approach is around the same speed as the CPU for smaller sizes, but when the array size is around the 2$\hat{1}$6 it will start taking more time. And this approach will eventually be slower than the other two. This is probably because the kernel is called multiply times. The idea of using offsets sounded great at the beginning, but it was probably wiser, and maybe easier too, to implement the atomicMax when there were more than 2048 elements.

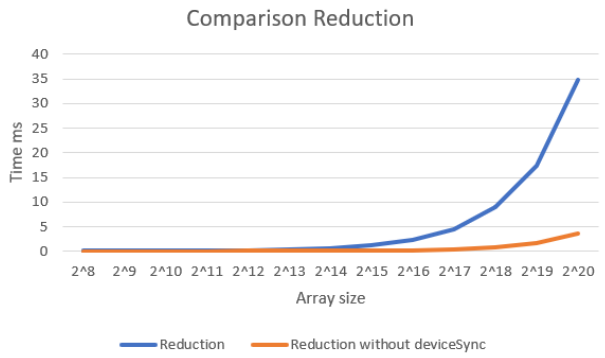

Figure 2: *Execution speed comparison between the different reduction versions*

We have also compared the execution time difference between the reduction with and without $cudaDeviceSynchronize()$. This takes a lot of time and didn't change the results all that much.

5 Github link

`https://github.com/MaximeRombaut1412/GCA_labs.git`