KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Finding max value in array

GPU vs CPU

Benjamin Rübenkamp
R0793577
MEIsw
Master industrieel ingenieur Elektronica-ICT
Academiejaar 2023-2024

** There are no code examples in this text, please look in github repository under session1: https://github.com/Benjamin4-23/Geavanc_comp_arch/tree/main/session2 **

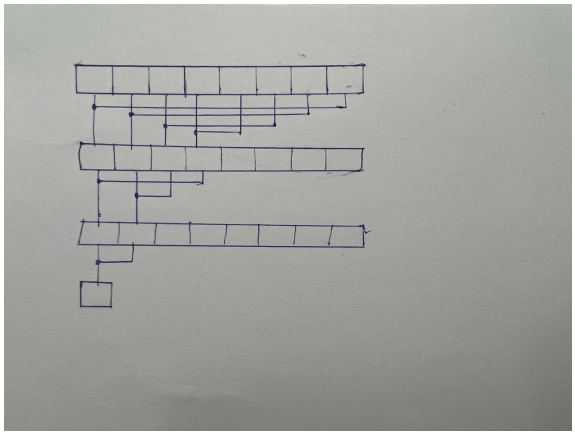
Introduction:
For this exercise, an array was created from which the program must extract the maximum value. This has to be done 1 time through use of the CPU, 1 time with the GPU using atomic instructions and 1 time with the GPU using reduction. At the end we look at how long these executions take for different lengths of arrays.

First, the intitial array "h_arrïs created and populated with random values.

CPU:
"cpu_max"Is the function that is called to have the cpu calculate the maximum of the array. The function will return the maximum value. The function simply iterates over all values with a loop and if the value of the array at index ïïs greater than the current maximum then that value is kept as the new maximum. This maximum value is initialized on the first value of the array. At the end of the loop, the maximum value is known.

GPU Atomic:
The GPU can easily calculate the maximum value of the array in a similar way. In this implementation, 1 thread will be started for each element and it will compare the number to the current maximum at its position in the array. However, these operations must be synchronized otherwise data races will occur. This is done by using atomic operations that are not executed simultaneously.

GPU Reduction:
The maximum is also calculated by reduction. This involves running over half the array each time (first half). A thread is started for each element. Each thread will look in the array at the position of the thread's id to see if the number on the other side of the array (mirrored relative to center of the array) is greater than the current element. If the other number is larger, the number at the index of the executing thread's Id is set to the other element so that the largest number of the 2 will be in the first half of the array. Here is an example of which elements are compared.



Figuur 1: Element comparison

After 1 iteration, all the highest elements of the array are in the first half. We can now perform the same operation on half of the new array (new array is the first half of the previous array), after which the largest values will be in the first half of that. Eventually, the largest element will be in the first place in the array. In order for half the threads to perform the comparison each time, the threadId is checked and has to be smaller than half the number of interesting elements. For example in the first iteration for an array of 8 numbers, the number of interesting numbers is equal to 8 and so the threads with threadId smaller than 4 must perform the comparison. (first half with index 0,1,2,3). The second iteration there are 4 interesting numbers so threadId's smaller than 2 (0 and 1) will run.

Threads are synchronized after each iteration.

How many iterations there must be is determined by the number of times the number of elements must be divided by 2 until 1 element remains. For example with 8 elements, 3 times can be divided by 2 before one gets to 1. (also log of 2 of that number) This of course only works if the number is a power of 2. Otherwise the array should be extended to the next power of 2 and padded with any number from the array. (Must not be larger) efficiency does go down in this process. (Unnecessary comparisons performed)

After all iterations, the largest number (in array at index 0) is plugged into result.

The CPU processes an array of 8 elements in 0.0004 miliseconds, the GPU with atomic instructions in 0.10 miliseconds and the GPU with reduction in 0.008 miliseconds. This is inline with what I expected. The cpu is the fastest (no thread spawning overhead), then reduction (needs only half as much threads) and in last place atomic instructions. If we use 512 elements, timings jump to 0.004, 0.10 and 0.009 respectively. I assumed the atomic instructions would increase the least with growing array size, which is true.

Notes:

- you can only go up to length 1024 for the array, that's how many threads can max be run at once, and thus synchronized. Going above this creates race conditions.