KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

GPU Memory

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

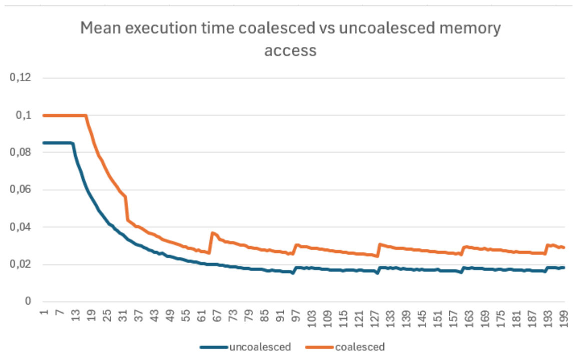
** There are no code examples in this text, please look in github repository under session1: https://github.com/Benjamin4-23/Geavanc_comp_arch/tree/main/session4 **

Introduction:
This exercise consists of 2 parts where we explore different types of memory on the GPU. In the first part of the exercise we greyscale an image using coalesced and uncoalesced memory access to see which one is faster. For the second part of the exercise we multiply 2 matrixes using global, shared and constant memory.

Part 1
For the first part of the exercise we greyscale an image. This is done 1 time with coalesced memory access and 1 time with uncoalescced memory access. With coalesced memory access threads access adjacent addresses simultaniously which performs less data reads (multiple threads use data from one read). There are 2 kernels, one gets the image in the form of [R,G,B,R,.....,B] and the second one in the form [R,R,R,...,G,G,G,...,B,B,...,B].

The first kernel accesses the R value in the first iteration of it's loop, but reads more than just the R value. Since that kernel will in the next iteration use the G value, it is now not used and will need to be read again.

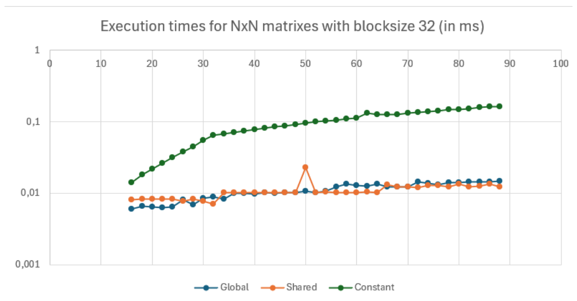
The second kernel optimizes this. It's first thread (thread_id 0) will read the R value of the first pixel, which now is next to the second pixel's R value. Now because 1 read gets more than 1 value, the second thread can get it's needed R value (for its first iteration of its loop) from the executed read from the first thread. This approach will result in less reads, saving time. In practice this way comes out slower.



Figuur 1: Execution times coalesced vs uncoalesced memory access

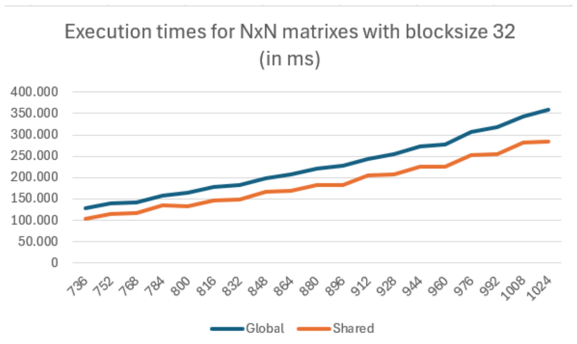
Part 2

This code tests each type of memory with a number of matrix sizes, and blocksizes. (See top of code) It makes the necessary arays, and initializes them with random values. It executes the kernel for the used matrix-and blocksize and times the execution using cuda events. Each time is also printed, so that it can be exported and plotted later. All calls to kernels are available in the code. To test one, you can simply uncomment the one you want to test. On the picture below you can see execution times for matrix sizes 16-88 in increments of 2. Blocksize was kept unchanged at 32.



Figuur 2: Execution times global vs shared vs constant

You can see that constant memory is the slowest. (the implementation where matrix A and B are stored in constant memory) Shared and global are somewhat the same until matrix sizes become bigger. Then you can see shared memory exectution is faster. We can increase matrix size further, but then we need to comment out the constants and the kernel that tests with those constants. This is because constant memory is limited to 64kb, which matrixes larger then 89x89 don't fit in. matrix sizes were now chosen bigger and the blocksize for this test was kept unchanged at 32.



Figuur 3: Comparison execution times global vs shared