

# Geavanceerde Computerarchitectuur

## 4<sup>th</sup> Session

Optimization techniques,  
Coalesced memory access and device (GPU) memory types

Alex Marinsek, [alexander.marinsek@kuleuven.be](mailto:alexander.marinsek@kuleuven.be)

DRAMCO – WaveCore – ESAT

Academic year 2023/24

# 0 Outline

- ① Intermezzo
- ② GPU memory access and memory types
- ③ Exercise
- ④ Appendix
- ⑤ Sources

# 1 Outline

- ① Intermezzo
- ② GPU memory access and memory types
- ③ Exercise
- ④ Appendix
- ⑤ Sources

# 1 Free allocated memory

```
1 #define SIZE 16
2 ...
3 int* p1 = (int*)malloc(SIZE*sizeof(int));
4 int* p2;
5 cudaMalloc(&p1, SIZE*sizeof(int));
6 ...
7 free(p1);
8 cudaFree(p2);
```

Otherwise memory leaks are introduced.

Can use [Valgrind](#) to detect leaks.

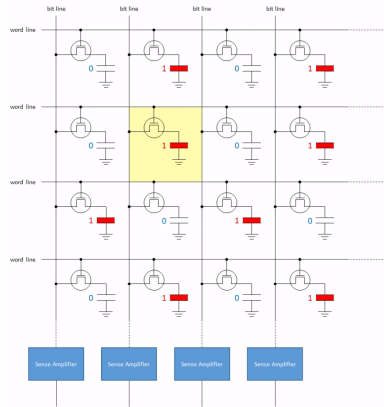
## 2 Outline

- ① Intermezzo
- ② GPU memory access and memory types
- ③ Exercise
- ④ Appendix
- ⑤ Sources

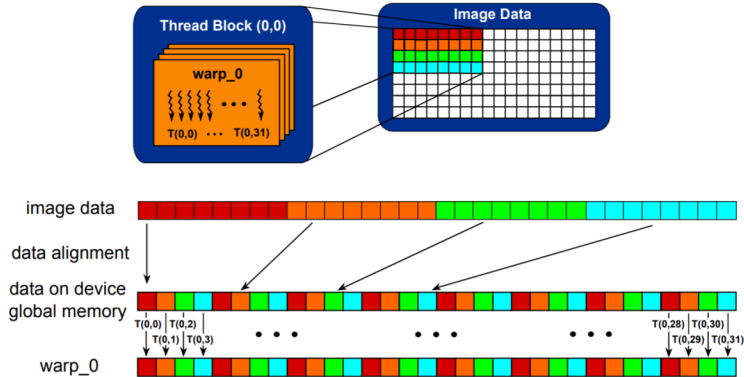
## 2 Coalesced memory access

*Coalesce: to come together to form one larger group, substance, etc.*

- ▶ DRAM access is done in groups (high-throughput parallel access)
- ▶ DRAM access is otherwise sequential (e.g. per request)
- ▶ Threads in a warp (32) execute the same instructions synchronously
- ▶ Group memory access reduces memory latency



## 2 Coalesced memory access in practice



## 2 Non-divergent threads and coalesced memory access

### Non-divergent threads

Threads in a **warp** take the same **computation** path (same complexity)

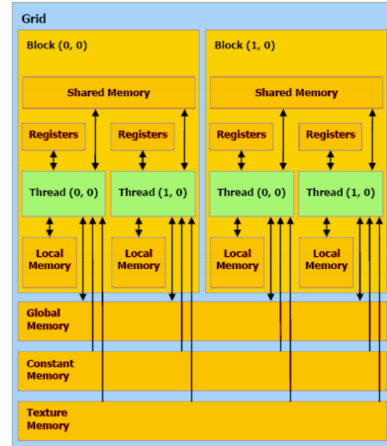
### Coalesced memory access

Threads in a **warp** access the same **memory** group (adjacent addresses)



## 2 GPU memory types

- ▶ Shared – Per block, on-chip
- ▶ Registers – Per thread, on-chip
- ▶ Local – Per thread, off-chip
- ▶ Constant – For all, off-chip
- ▶ Global – For all, off-chip
- ▶ Texture – For all, off-chip



## 2 Global memory

- ▶ Off-device – high access latency
- ▶ All threads in the grid have access, as well as the CPU
- ▶ Large size – typically several GB
- ▶ The 'default' option

As in previous labs

```
1 int* garr;  
2 cudaMalloc(&garr, SIZE*sizeof(int));  
3 // or (see analogy to constant memory in a later slide)  
4 __device__ int garr[SIZE];
```

## 2 Shared memory

- ▶ On-device – low access latency
- ▶ Shared among the threads of a single block
- ▶ User-managed cache, by default 48 kB ( `cudaFuncSetCacheConfig(...)` )
- ▶ Two options in CUDA, see below

Hardcode `SIZE` (or use `#define` )

```
my_kernel<<<blk, tpb>>>(...); → extern __shared__ int smem[SIZE];
```

Define size dynamically during kernel invocation

```
my_kernel<<<blk, tpb, size>>>(...); → extern __shared__ int smem[];
```

## 2 Workflow with shared memory

*Threads migrate data from global to shared memory*

- 1 (on CPU) Copy data **to global memory** and run the kernel
- 2 (on GPU, per thread) Migrate one or more elements **to shared memory**
- 3 **Process** the data in **shared memory**
- 4 Migrate the data **back to global memory**
- 5 (on CPU) Copy data from global memory

## 2 Workflow with shared memory

*Threads migrate data from global to shared memory*

- 1 (on CPU) Copy data **to global memory** and run the kernel
- 2 (on GPU, per thread) Migrate one or more elements **to shared memory**
- 3 **Process** the data in **shared memory**
- 4 Migrate the data **back to global memory**
- 5 (on CPU) Copy data from global memory

Where should we put `__syncthreads()` ?

## 2 Constant memory

- ▶ Off-device, optimized for low-latency read
- ▶ All threads in the grid have access, as well as the CPU
- ▶ Small size – 64 KB

From [the documentation](#):

```
1 __constant__ int cmem[SIZE]; // GPU
2 ...
3 int* arr; // CPU
4 ...
5 cudaMemcpyToSymbol(cmem, arr, SIZE * sizeof(int)); // From CPU to GPU
```

## 2 Other memory

### Registers

- ▶ On-device – Low access latency
- ▶ Access per thread
- ▶ `threadIdx.x`

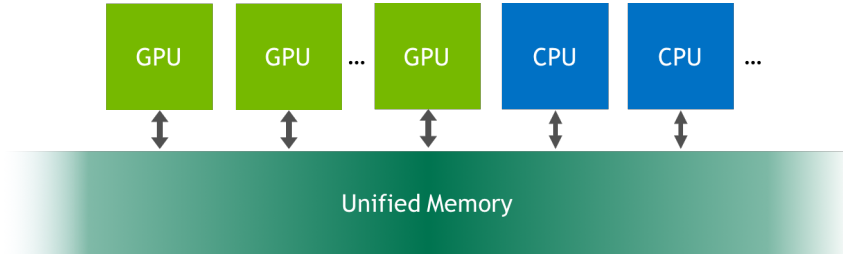
### Local memory

- ▶ Off-device – High(er) access latency
- ▶ Access per thread
- ▶ `int arr[128];`

### Texture memory

- ▶ Off-device with optimized read access
- ▶ Access for all threads
- ▶ For 2D graphic data

## 2 Regarding a question



*Unified memory* allows one to allocate once and use on both devices.  
More [in the docs](#) and [a blog post](#).



### 3 Outline

- ① Intermezzo
- ② GPU memory access and memory types
- ③ Exercise
- ④ Appendix
- ⑤ Sources

### 3 Assignment

#### Part 1 – Coalesced memory access

- ▶ Design a kernel that calculates the average grayscale of an image.
- ▶ Evaluate execution time for coalesced and un-coalesced memory access.

#### Part 2 – Global, shared, and constant memory

- ▶ Design a kernel that multiplies two matrices.
- ▶ Assess the processing time for a kernel that utilizes global-only, global and shared, and global and constant memory.

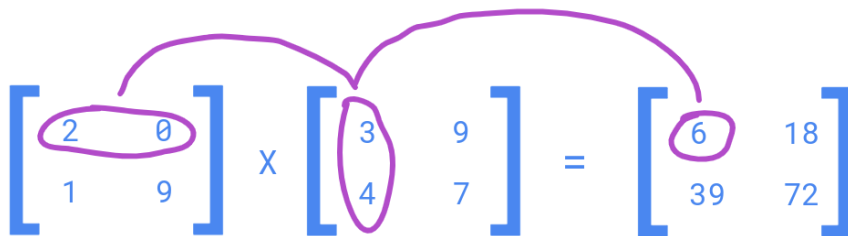
*Evaluations should consider changing the grid and/or input size.*

### 3 Pointers – Part 1

- ▶ Load up an image using last time's code or make a 1D array of your liking
- ▶ Use one array which sorts data as `RGBRGB...RGB` and one as `RR...RGG...GGB...B`
- ▶ Design two kernels, each for processing one of the above arrays (coalesced and uncoalesced)
- ▶ Calculate the grayscale ( $\text{gray} = \frac{1}{3}\text{RGB}$ ) using the data at the corresponding elements
- ▶ Derive gray of a pixel by a single thread (iterate over the three `RGB` values)
- ▶ Scale up the grid size (small number of threads  $\rightarrow$  large number of threads and compare the timing results)

### 3 Pointers – Part 2

Recall matrix multiplication:

$$2 \times 3 + 0 \times 4 = 6$$


The diagram shows the multiplication of two 2x2 matrices. The first matrix is  $\begin{bmatrix} 2 & 0 \\ 1 & 9 \end{bmatrix}$  and the second is  $\begin{bmatrix} 3 & 9 \\ 4 & 7 \end{bmatrix}$ . The result is  $\begin{bmatrix} 6 & 18 \\ 39 & 72 \end{bmatrix}$ . Purple annotations highlight the calculation of the top-left element: the elements 2 and 0 from the first row of the first matrix are circled together; the elements 3 and 4 from the first column of the second matrix are circled together; and a purple arc connects the 2 to the 3 and the 0 to the 4. In the resulting matrix, the element 6 is circled, representing the sum of the products 2\*3 and 0\*4.

Image source: <https://www.codingem.com/numpy-at-operator/>

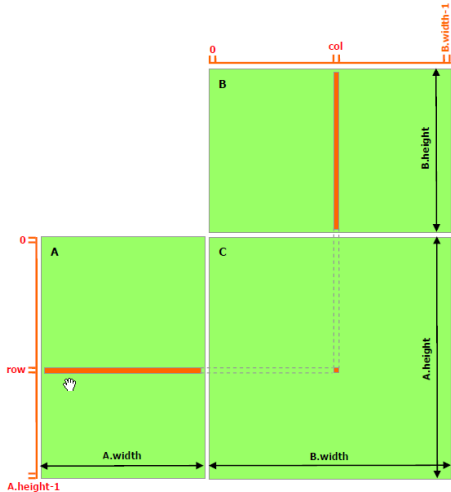
### 3 Pointers – Part 2

- ▶ Define an array that represents a matrix - in 1D or 2D form (start off with small square matrices and simple numbers for easier debugging)
- ▶ In this way define/allocate memory for three matrices, which will serve for  $C = AB$
- ▶ Make a kernel that uses only global memory to compute  $C \rightarrow$  **verify the results**
- ▶ Make a second and third matrix multiplication kernel, which use shared and constant memory, respectively, in addition to global memory
- ▶ Time the kernels for different input sizes

## 4 Outline

- ① Intermezzo
- ② GPU memory access and memory types
- ③ Exercise
- ④ **Appendix**
- ⑤ Sources

## 4 Memory access during matrix multiplication



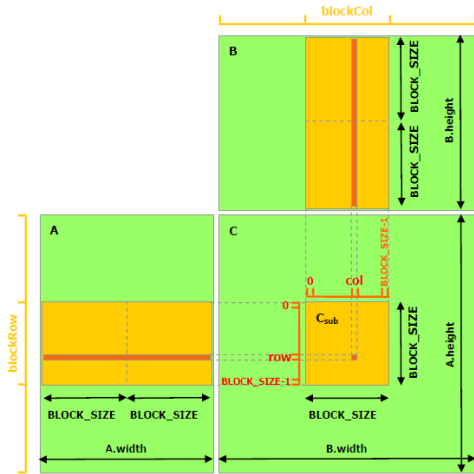
Assume:

- ▶ Matrices A, B, and C all  $N \times N$
- ▶  $N \times N$  grid of threads

Then:

- ▶ Each thread writes 1 element in C
- ▶ A-rows and B-columns are re-read  $N-1$  times

## 4 Improvement by leveraging shared memory



Assume:

- ▶ Matrices A, B, and C all  $N \times N$
- ▶  $N \times N$  grid of threads
- ▶ Square blocks with size  $BLOCK\_SIZE$

Then:

- ▶ Each thread writes 1 element in C
- ▶ Each thread copies one element from both A and B to shared memory,  $k$ -times
- ▶ A-rows and B-columns are re-read  $k$ -times
- ▶  $k = \frac{N}{BLOCK\_SIZE}$



## 5 Outline

- ① Intermezzo
- ② GPU memory access and memory types
- ③ Exercise
- ④ Appendix
- ⑤ Sources

## 5 Sources

- ▶ Coalesced meaning
- ▶ DRAM intro
- ▶ Coalesced access figure
- ▶ Blog post about unified access
- ▶ Blog post about shared memory
- ▶ GPU memory figure