Geavanceerde Computerarchitectuur 4th Session

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

Thomas Feys, thomas.feys@kueluven.be
DRAMCO – WaveCore – ESAT

Academic year 2024/25

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

```
#define SIZE 16
...
int* p1 = (int*)malloc(SIZE*sizeof(int));
int* p2;
cudaMalloc(&p2, SIZE*sizeof(int));
...
free(p1);
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

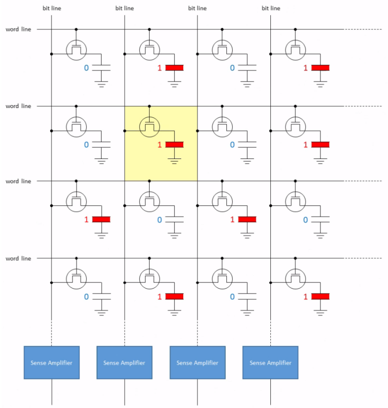
4 Appendix

5 Sources

2 Coalesced memory access

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

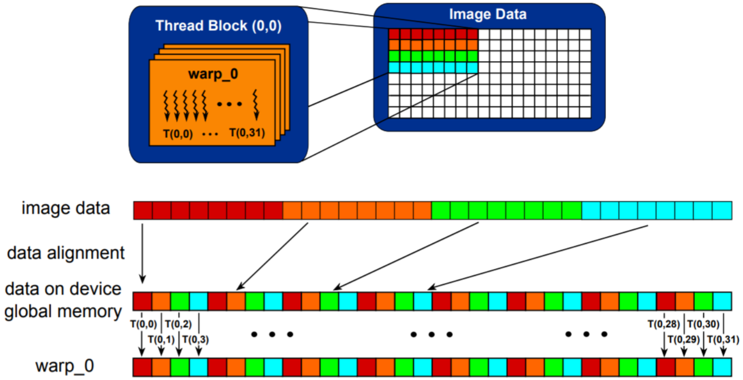
- DRAM access is done in groups (high-throughput parallel access)
- DRAM access is otherwise sequential (e.g. per request)
- Threads in a warp (32) execute the same instructions synchronously
- Group memory access reduces memory latency



2 Coalesced memory access

- Is achieved when all threads in a warp access consecutive global memory locations.
- When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations. If that's the case, the hardware combines (coalesces) all these accesses into a consolidated access to consecutive DRAM locations.
- For example, for a given load instruction of a warp, if thread 0 accesses global memory location N2, thread 1 location N+1, thread 2 location N+2, and so on, all these accesses will be coalesced into a single request for consecutive locations when accessing the DRAMs.
- Such coalesced access allows the DRAMs to deliver data as a burst.

2 Coalesced memory access in practice



2 Non-divergent threads and coalesced memory access

Non-divergent threads

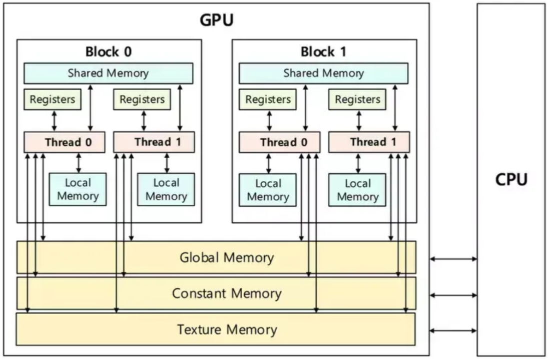
Threads in a **warp** take the same **computation** path (same complexity)

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

```
int* garr;
cudaMalloc(&garr, SIZE*sizeof(int));
// or (see analogy to constant memory in a later slide)
__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>>>(...);` $\rightarrow$ `__shared__ int smem[SIZE];`

Define size dynamically during kernel invocation

`my_kernel<<<blk, tpb, size>>>(...);` $\rightarrow$ `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:

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

2 Other memory

Registers

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

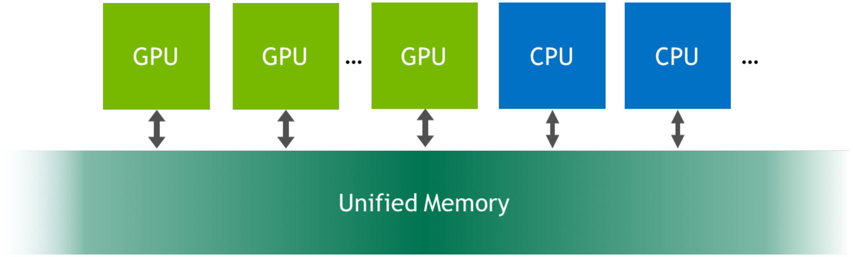
Local memory

- 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.
(Abstraction that handles memory migration.)

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 inverts the red color component in 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

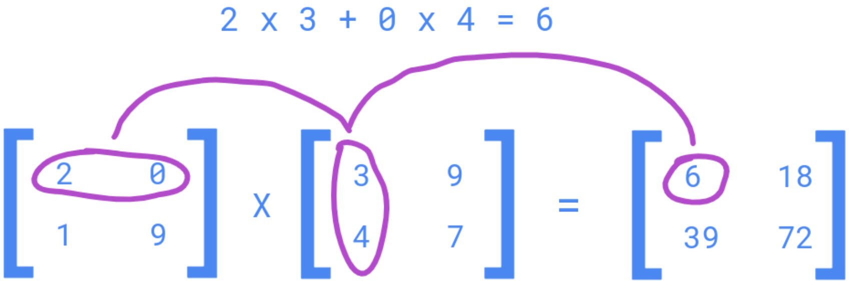
- Load up an image using last time's code or make a 1D array of your liking.
- Use one array which sorts data as `RGBRGB...RGB` and one as `RR...RGG...GBB...B`.
- Design two kernels, each for processing one of the above arrays.
- Calculate the inverse of the red color component, $\mathrm{R} = 255 - R$.
- Time the kernels for different input and/or grid sizes.

3 Guiding questions – Part 1

- What is the performance difference between the coalesced and uncoalesced kernel?
- How does the performance difference change dependent on input and/or grid size?

3 Pointers – Part 2

Recall matrix multiplication:



$$\begin{bmatrix} 2 & 0 \\ 1 & 9 \end{bmatrix} \times \begin{bmatrix} 3 & 9 \\ 4 & 7 \end{bmatrix} = \begin{bmatrix} 6 & 18 \\ 39 & 72 \end{bmatrix}$$

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 $\mathbf{C} = \mathbf{AB}$.
- Make a kernel that uses only global memory to compute $\mathbf{C} \rightarrow$ **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 and/or grid sizes.

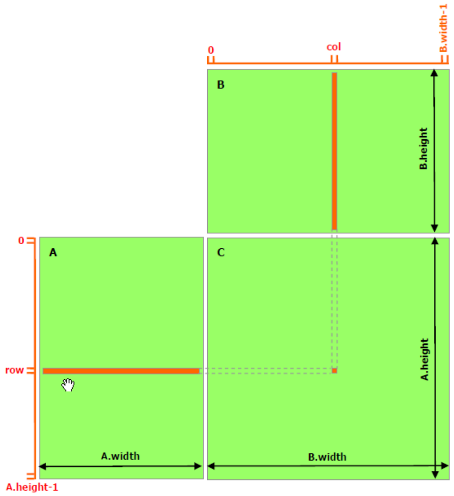
3 Guiding questions – Part 2

- How much time does memory migration take (CPU to GPU global; GPU global to GPU shared)?
- How do the execution times of matrix multiplication compare when employing the three different memory types?
- How do the times scale with input and/or grid size?

Outline

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

4 Memory access during matrix multiplication



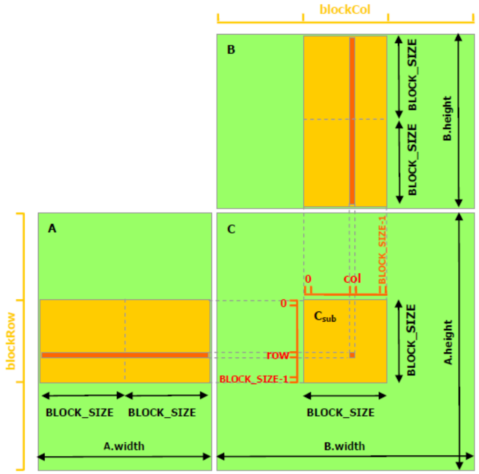
Assume:

- Matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ all NxN
- NxN grid of threads

Then:

- Each thread writes 1 element in $\mathbf{C}$
- $\mathbf{A}$-rows and $\mathbf{B}$-columns are re-read N-1 times

4 Improvement by leveraging shared memory



Assume:

- Matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ all NxN
- NxN grid of threads
- Square blocks with size BLOCK_SIZE

Then:

- Each thread writes 1 element in $\mathbf{C}$
- Each thread copies one element from both $\mathbf{A}$ and $\mathbf{B}$ to shared memory, k-times
- $\mathbf{A}$-rows and $\mathbf{B}$-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