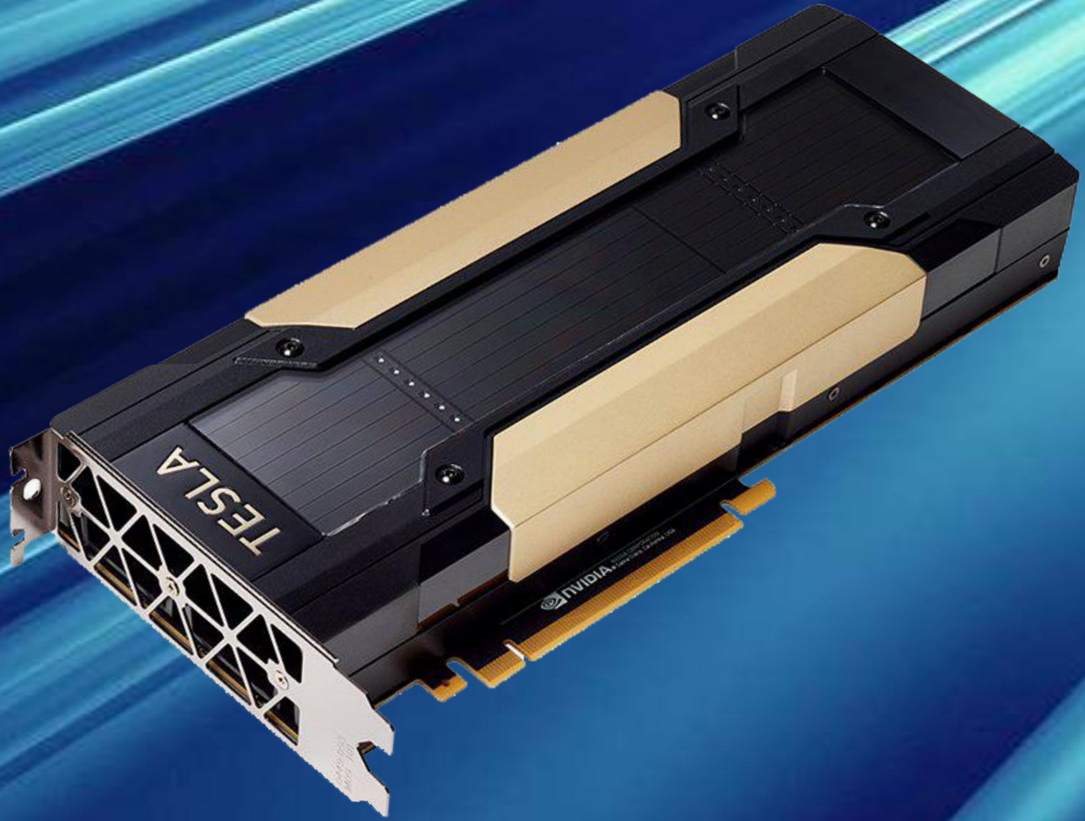
sciCORE
Center for scientific computing

Universität Basel

SIB Swiss Institute of Bioinformatics



Introduction to GPU programming

Outline

Day 1
(09:00 – 13:00)

- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Homework

Day 2
(09:00 – 13:00)

- ❑ Recap and homework solutions
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
  - ❑ Sequence alignment
- ❑ Exercise
  - ❑ Area of the Mandelbrot set
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks

Outline

Day 1
(09:00 – 13:00)

- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Homework

Day 2
(09:00 – 13:00)

- ❑ Recap and homework solutions
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
  - ❑ Sequence alignment
- ❑ Exercise
  - ❑ Area of the Mandelbrot set
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks

Outline

Day 1

(09:00 – 13:00)

- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Homework

Day 2

(09:00 – 13:00)

- ❑ Recap and homework solutions
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
  - ❑ Sequence alignment
- ❑ Exercise
  - ❑ Area of the Mandelbrot set
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks

Warp divergence (aka Execution divergence)

Threads are executed in warps of 32, with all threads in the warp executing the same instruction at the same time.

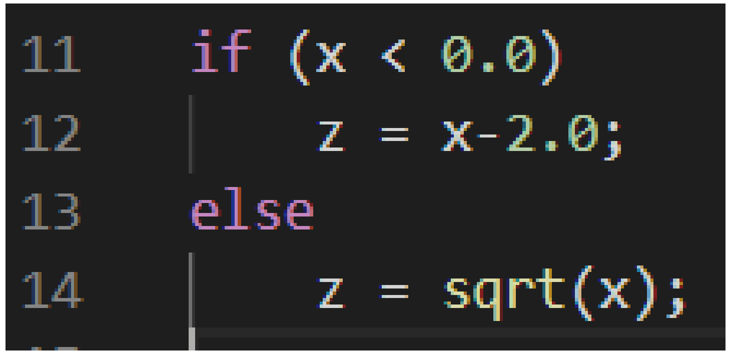
What happens if different threads in a warp need to do different things?

This is called **warp divergence**. CUDA will generate correct code to handle this, but to understand the performance you need to understand what CUDA does with it.

All the threads in a warp execute both conditional branches

If the condition evaluate to false for a given thread, it will **remain idle** and wait for the other threads to finish.

=> **potentially large loss of performance.**



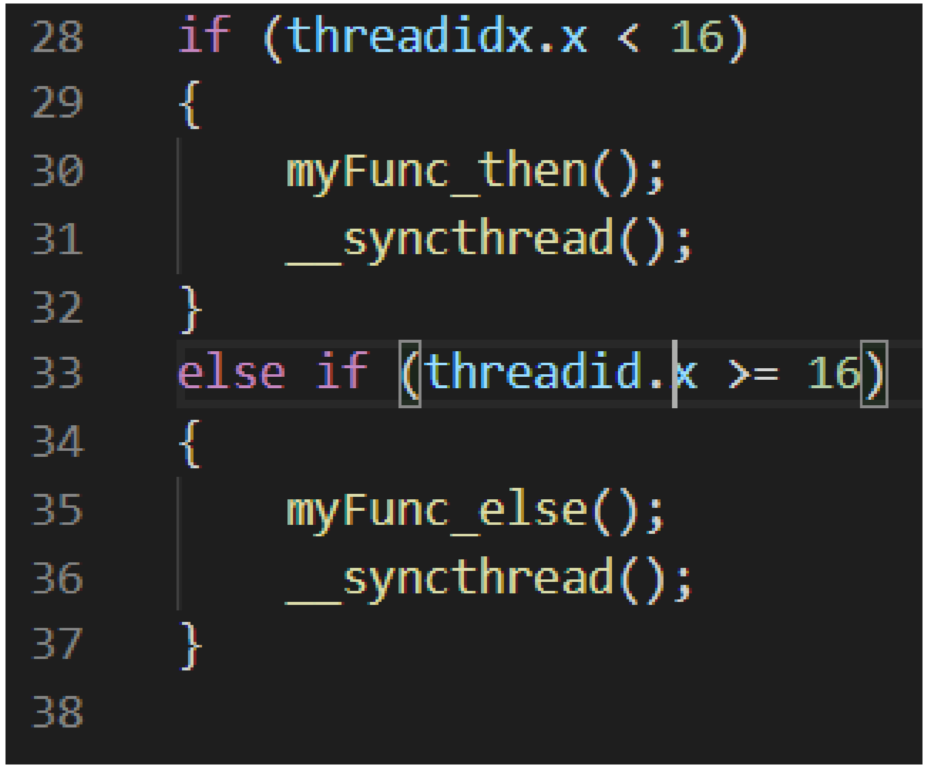
```
if (x < 0.0)
    z = x-2.0;
else
    z = sqrt(x);
```

Deadlock

Again, all the threads in a warp execute both branches of the condition.

Example: a single block of 32 threads is mapped to a single warp, where the 32 threads **will execute the same instruction**.

- 50% of the threads (with threaded.x < 16) execute the first branch, **the other 50% are idle.**
- The first 50% reach the __syncthread() instruction. It will never return because the other threads cannot execute the second branch yet.



```
if (threadidx.x < 16)
{
    myFunc_then();
    __syncthread();
}
else if (threadid.x >= 16)
{
    myFunc_else();
    __syncthread();
}

```

Outline

Day 1

(09:00 – 13:00)

- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Homework

Day 2

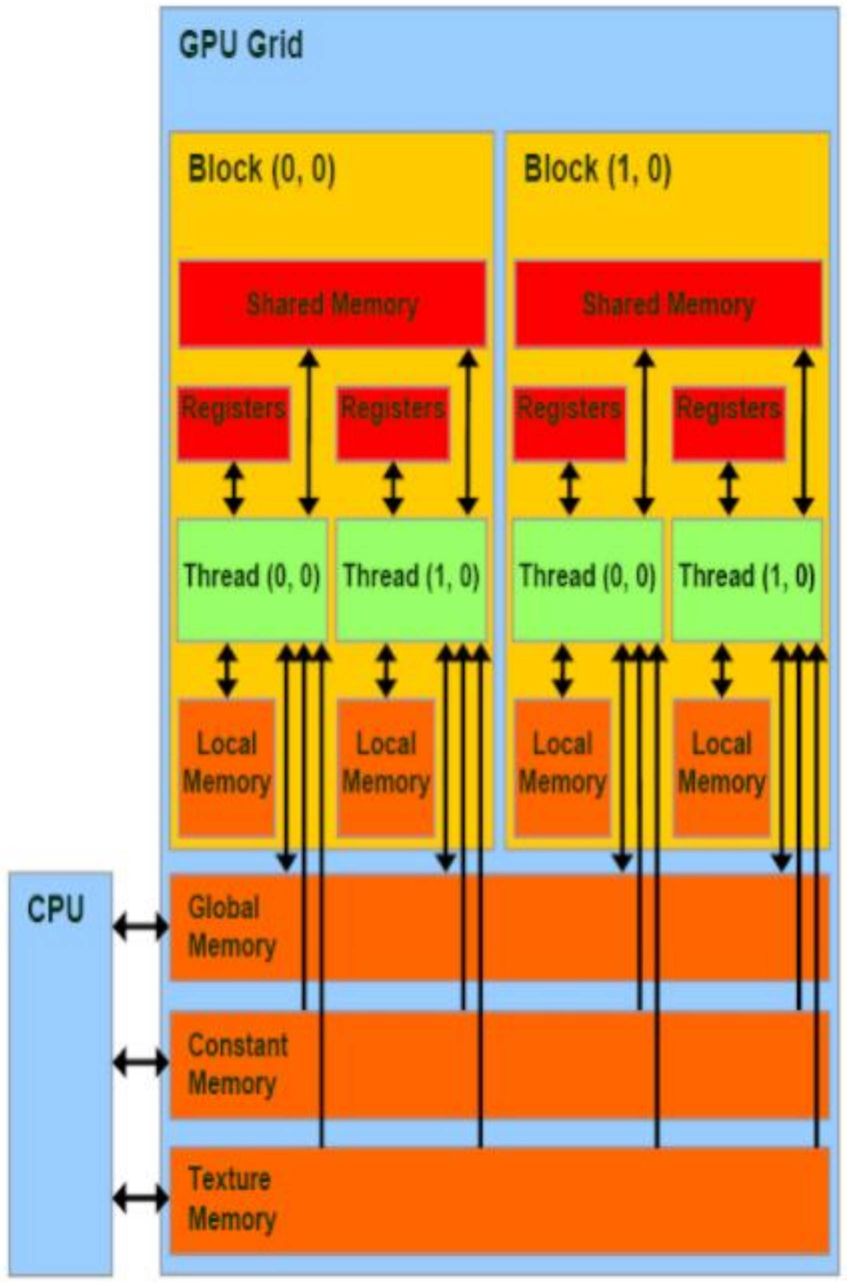
(09:00 – 13:00)

- ❑ Recap and homework solutions
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
  - ❑ Sequence alignment
- ❑ Exercise
  - ❑ Area of the Mandelbrot set
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks

Memory management

There are 6 types of memories in a GPU.

Memory type	Speed	Location	Size	Comments
Register	+++++	In-chip	256 KB/SM	Visible only to the thread that wrote it and it has the same lifespan as that thread
Shared	++++	In-chip	48 KB/SM	Visible to all threads within the block and it has the same lifespan as that block
Constant	+++	Off-chip	64 KB	≈8 KB/SM cached. Read-only. Used for data that will not change over the course of a kernel execution
Texture	++	Off-chip	≈ GB	Read-only. Useful when all reads in a warp are physically adjacent. Taken out from Global memory.
Local	+	Off-chip	512 KB/Thread	Visible only to the thread that wrote it and it has the same lifespan as that thread. Taken out from Global memory.
Global	+	Off-chip	≈ GB	Visible to all threads in the application and it has the same lifespan as the host allocation



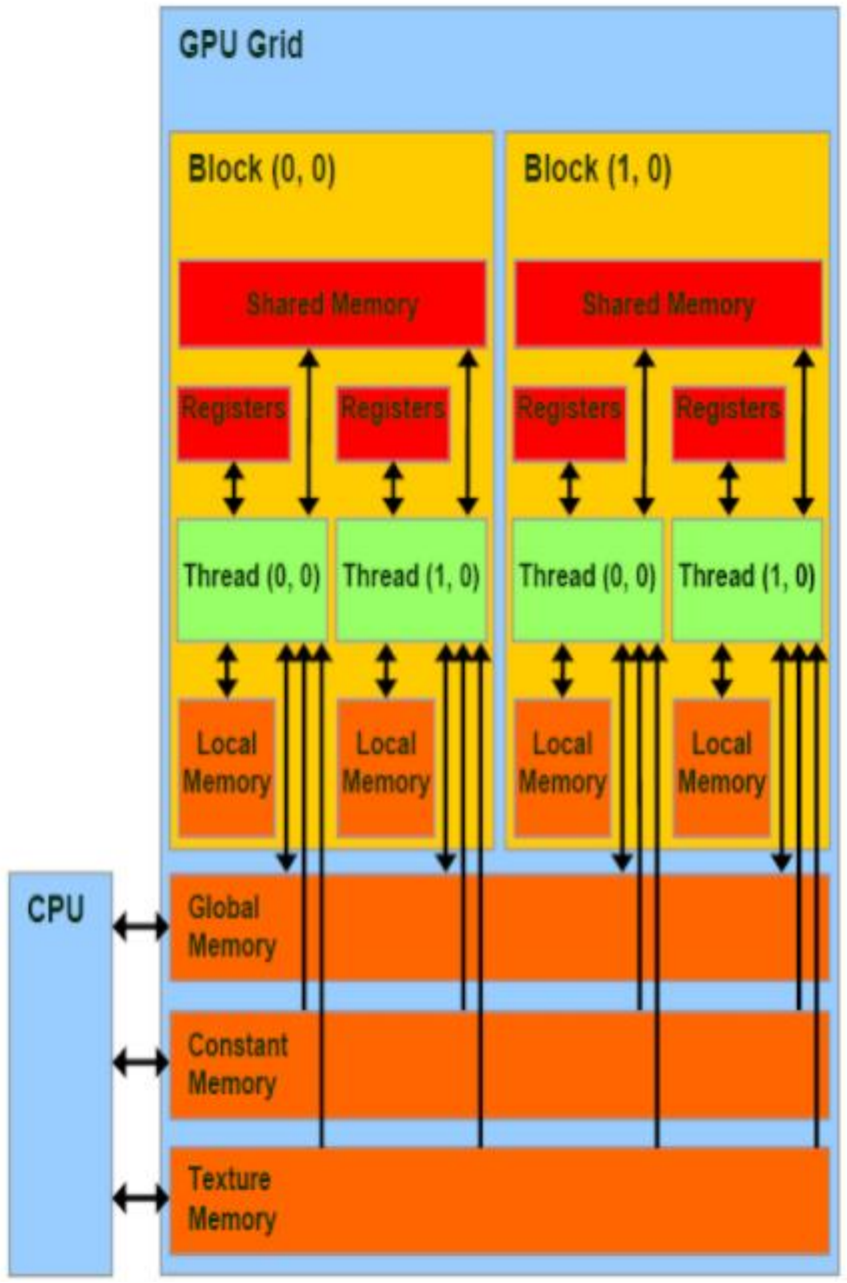
Source: cuda-programming.blogspot.com

Memory management

There are 6 types of memories in a GPU.

Memory type	Speed	Location	Size	Comments
Register	+++++	In-chip	256 KB/SM	Visible only to the thread that wrote it and it has the same lifespan as that thread
Shared	++++	In-chip	48 KB/SM	Visible to all threads within the block and it has the same lifespan as that block
Constant	+++	Off-chip	64 KB	≈8 KB/SM cached. Read-only. Used for data that will not change over the course of a kernel execution
Texture	++	Off-chip	≈ GB	Read-only. Useful when all reads in a warp are physically adjacent. Taken out from Global memory.
Local	+	Off-chip	512 KB/Thread	Visible only to the thread that wrote it and it has the same lifespan as that thread. Taken out from Global memory.
Global	+	Off-chip	≈ GB	Visible to all threads in the application and it has the same lifespan as the host allocation

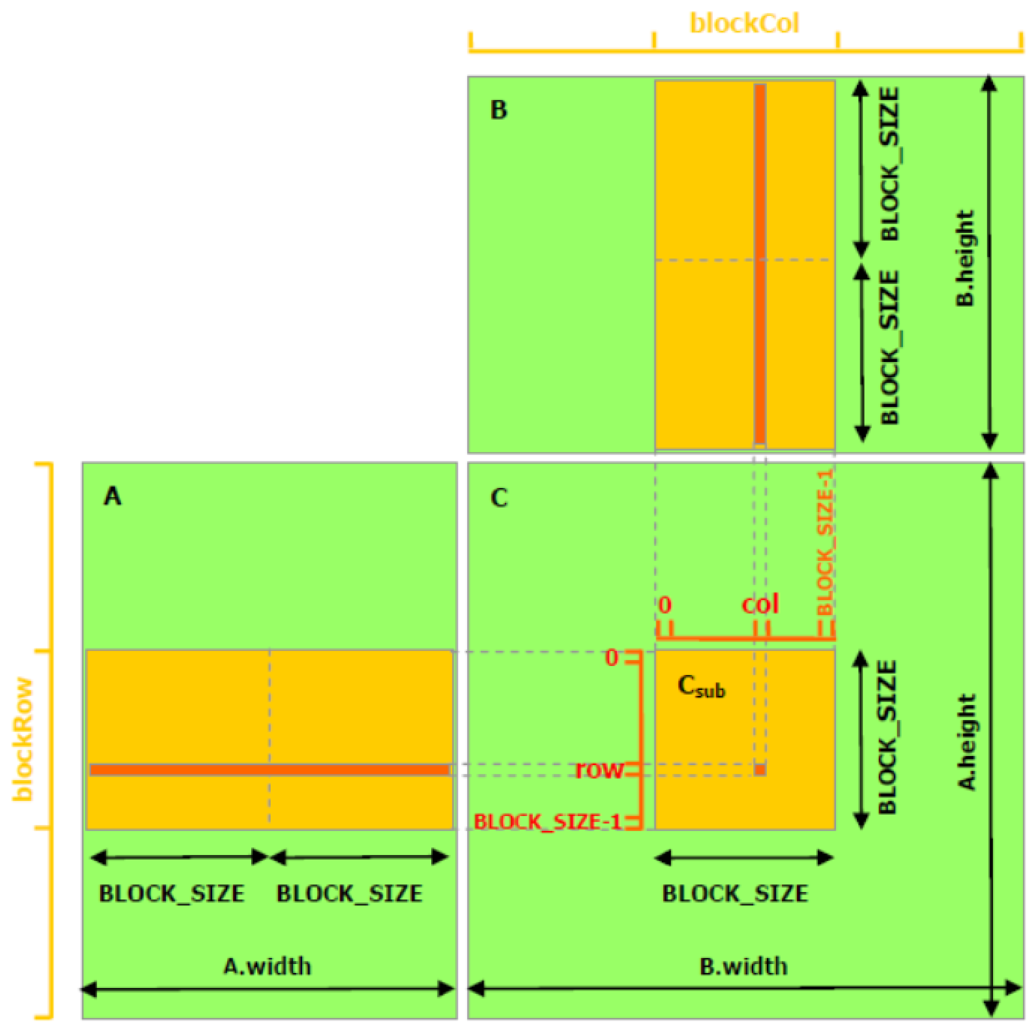
Recommendation: Try to make use of shared memory wherever possible. Global memory can be 150x slower!



Source: cuda-programming.blogspot.com

Memory management

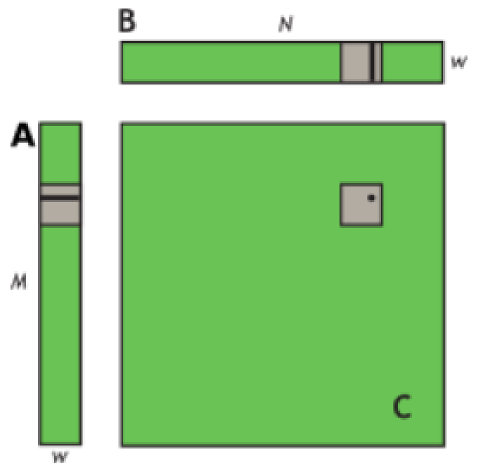
Up to now we have linearized the 2D matrices and mapped the multiplication onto the GPU.
Now we are going to perform the multiplication by blocks:



- Each threadblock computes a sub-matrix C_{sub}
- Each thread within the block computes one element of C_{sub}

BLOCK_SIZE = 16 (or 32), so that the # of threads/block is a multiple of the warp size (w) and remains below the max # of threads/block.

```
__global__ void simpleMultiply(float *a, float* b, float *c, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for (int i = 0; i < TILE_DIM; i++) {
        sum += a[row*TILE_DIM+i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```



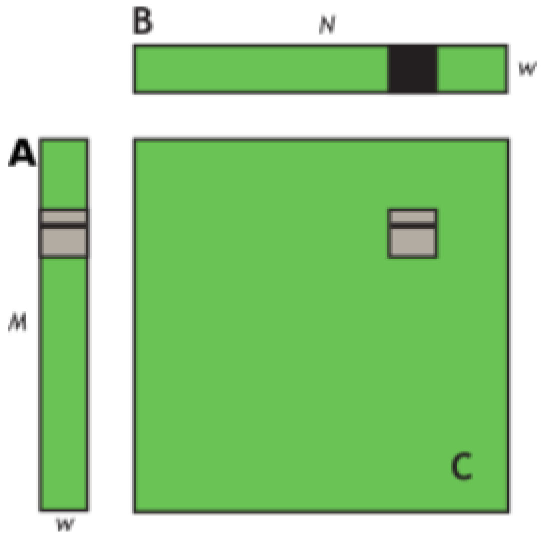
blockDim.x, blockDim.y, and TILE_DIM = w

Effective BW: 119,9 GB/s on Tesla V100

Source: Nvidia CUDA documentation

Memory management

Let's analyze how data is accessed by the warps:



Each row of C_{sub} needs a row of the A tile and all the columns of the B tile.
For each iteration of i, the warp reads a full row of the B tile (**Good**), but all threads in the warp read the same value from global memory from the A tile (**Bad**): wasted memory BW and likely cache miss.

It would be better to read the A tile from shared memory:

```
__global__ void simpleMultiply(float *a, float* b, float *c, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for (int i = 0; i < TILE_DIM; i++) {
         sum += a[row*TILE_DIM+i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

Effective BW: 119,9 GB/s on Tesla V100

```
__global__ void coalescedMultiply(float *a, float* b, float *c, int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;

    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];

    __syncwarp();

    for (int i = 0; i < TILE_DIM; i++) {
         sum += aTile[threadIdx.y][i]* b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

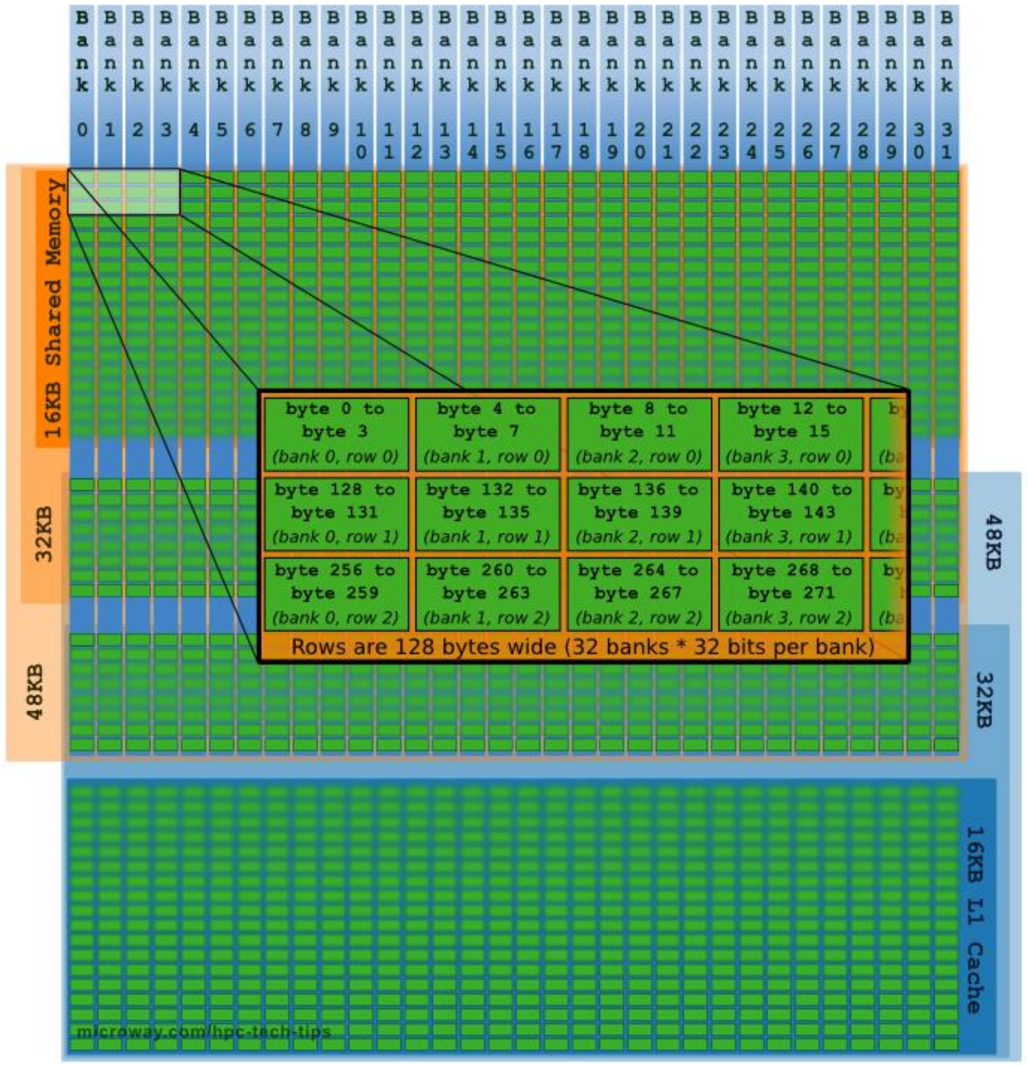
Effective BW: 144,4 GB/s on Tesla V100

Source: Nvidia CUDA documentation

Each element of tile A is read once w/o BW waste to shared memory.

Memory management

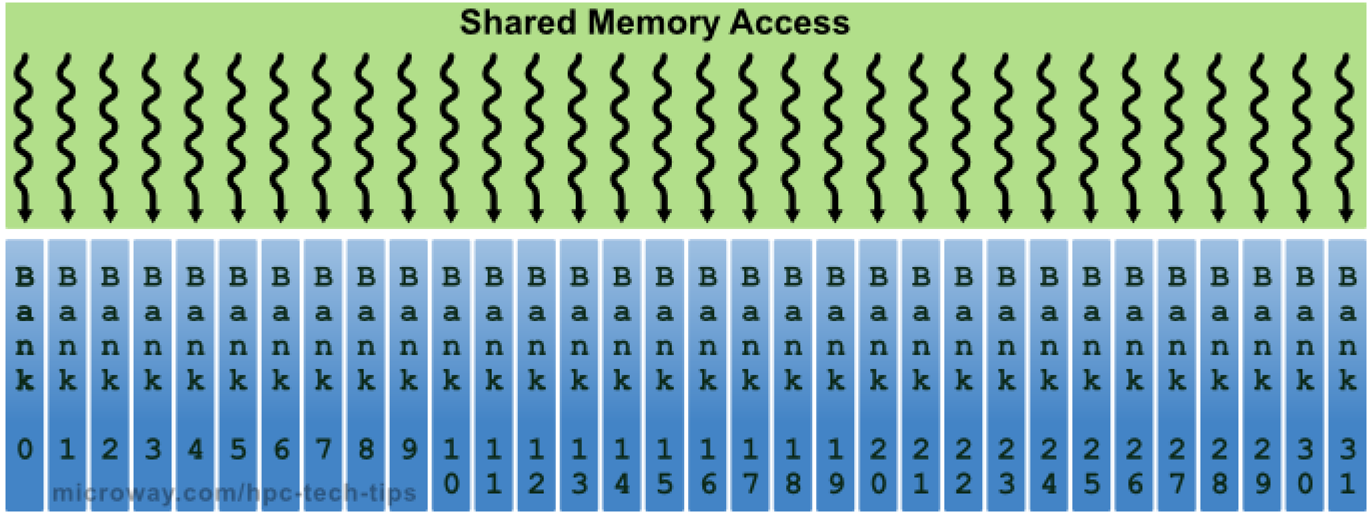
Because shared memory is in-chip, it has much higher bandwidth and lower latency than accessing global memory. But the speed gain depends on not having bank conflicts between threads.



Bank 0 … Bank 31
16KB Shared Memory
32KB
48KB
byte 0 to byte 3 (bank 0, row 0)
byte 4 to byte 7 (bank 1, row 0)
byte 8 to byte 11 (bank 2, row 0)
byte 12 to byte 15 (bank 3, row 0)
byte 128 to byte 131 (bank 0, row 1)
byte 132 to byte 135 (bank 1, row 1)
byte 136 to byte 139 (bank 2, row 1)
byte 140 to byte 143 (bank 3, row 1)
byte 256 to byte 259 (bank 0, row 2)
byte 260 to byte 263 (bank 1, row 2)
byte 264 to byte 267 (bank 2, row 2)
byte 268 to byte 271 (bank 3, row 2)
Rows are 128 bytes wide (32 banks * 32 bits per bank)
48KB
32KB
16KB L1 Cache
microway.com/hpc-tech-tips

Figure 1: Shared Memory and L1 Cache

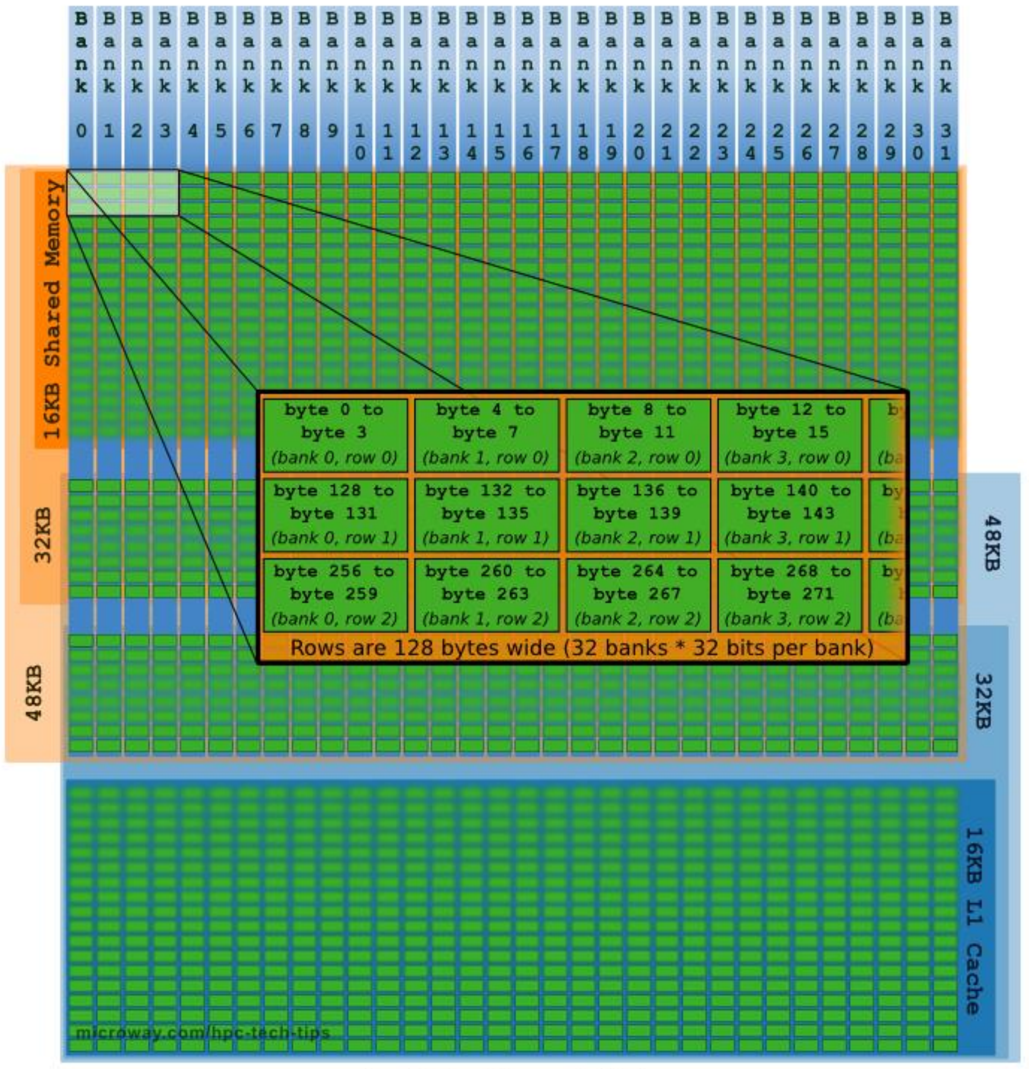
In order to effectively service concurrent memory access, shared memory is divided into 32 equal memory Banks.



Source: microway.com

Memory management

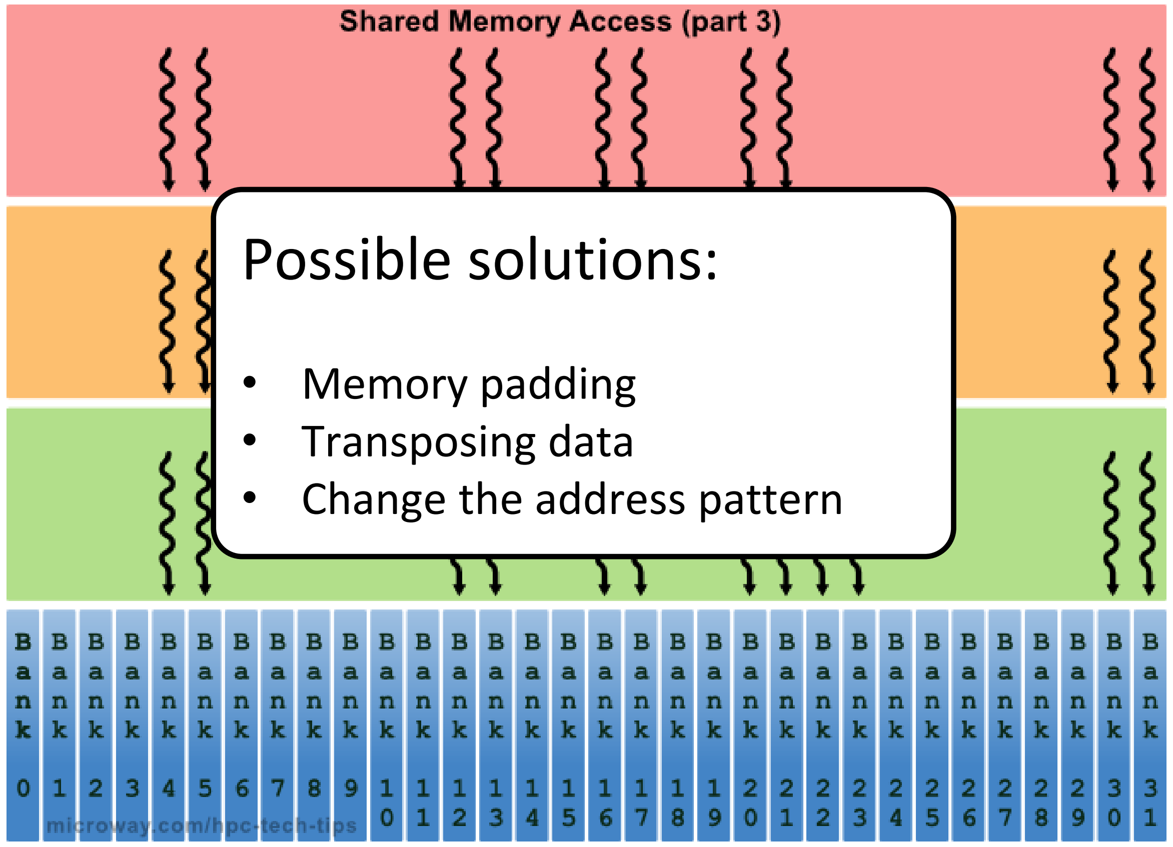
Because shared memory is in-chip, it has much higher bandwidth and lower latency than accessing global memory. But the speed gain depends on not having bank conflicts between threads.



Bank 0 … Bank 31
16KB Shared Memory
32KB
48KB
48KB
32KB
16KB L1 Cache
byte 0 to byte 3 (bank 0, row 0)
byte 4 to byte 7 (bank 1, row 0)
byte 8 to byte 11 (bank 2, row 0)
byte 12 to byte 15 (bank 3, row 0)
byte 128 to byte 131 (bank 0, row 1)
byte 132 to byte 135 (bank 1, row 1)
byte 136 to byte 139 (bank 2, row 1)
byte 140 to byte 143 (bank 3, row 1)
byte 256 to byte 259 (bank 0, row 2)
byte 260 to byte 263 (bank 1, row 2)
byte 264 to byte 267 (bank 2, row 2)
byte 268 to byte 271 (bank 3, row 2)
Rows are 128 bytes wide (32 banks * 32 bits per bank)
microway.com/hpc-tech-tips

Figure 1: Shared Memory and L1 Cache

But if multiple thread requests map to the same bank, the accesses are serialized with as many accesses as to ensure there a no conflicts.



Possible solutions:

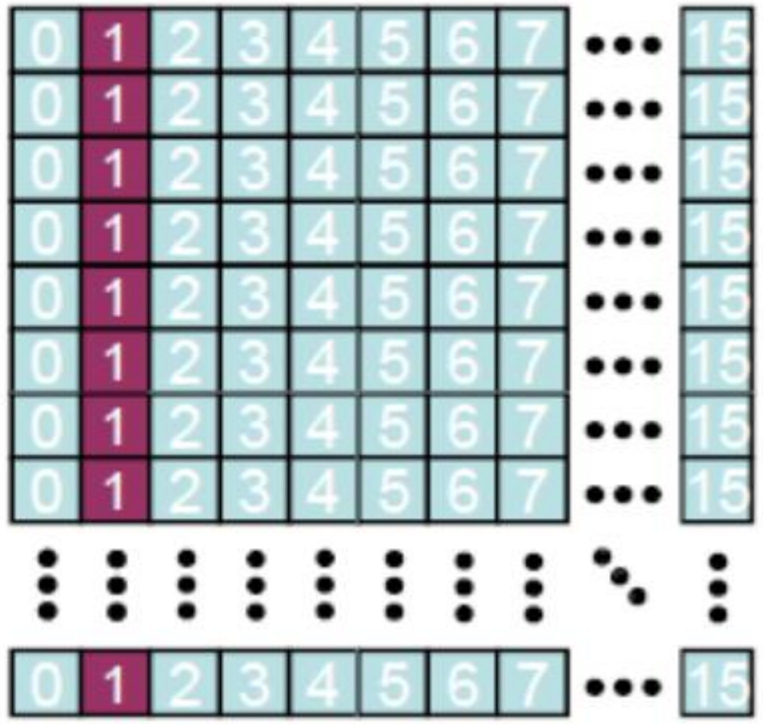
- Memory padding
- Transposing data
- Change the address pattern

Source: microway.com

Memory management

Example of bank conflict:

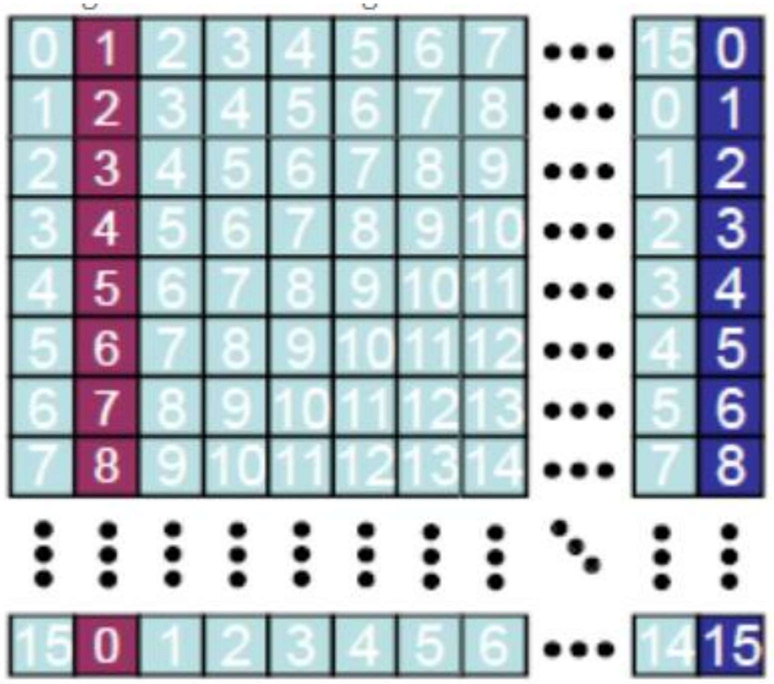
Let’s assume we want to process a 2D array (16x16) and we assign each thread to process one row.



The number is the bank in which data is stored.
All threads move to the right (like the purple cells) having a 16-way bank conflict.

With memory padding we add an extra column with zeros.
Now there is no bank conflicts!

Instead of: _shared_ int shared[TILE_WIDTH][TILE_HEIGHT];
use: _shared_ int shared[TILE_WIDTH+1][TILE_HEIGHT];



Source: cuda-programming-blogspot.com

Outline

Day 1

(09:00 – 13:00)

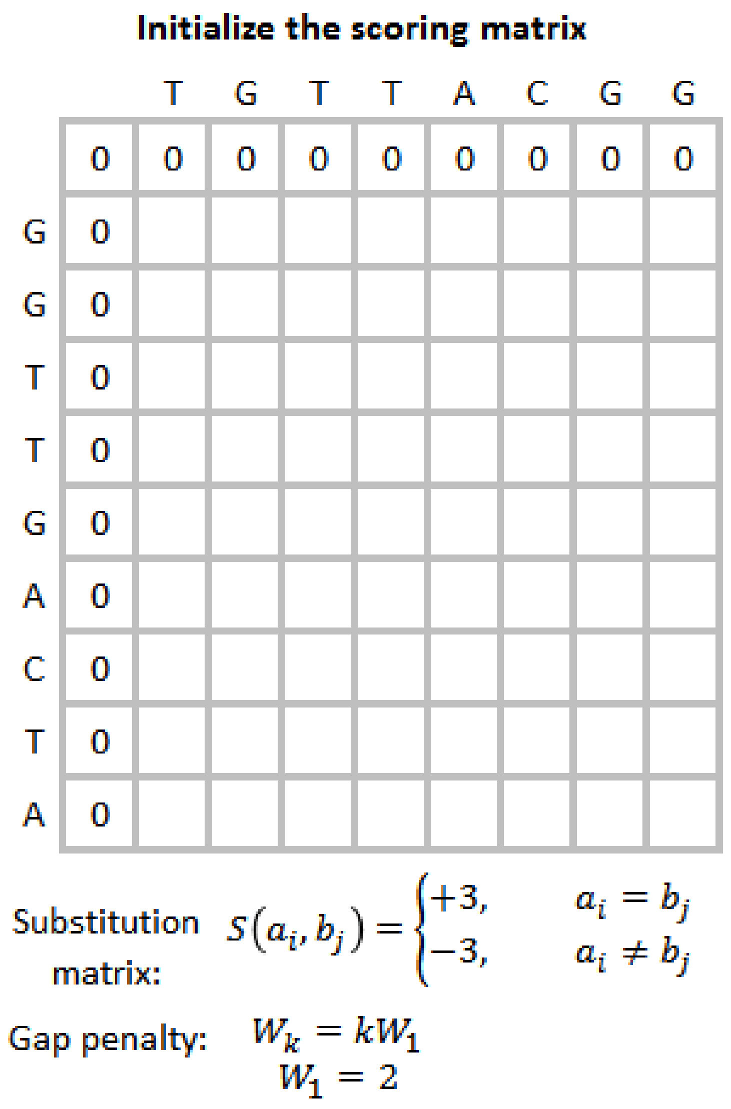
- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
    - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
    - ❑ Genomes simulation
    - ❑ Random numbers generation
- ❑ Homework

Day 2

(09:00 – 13:00)

- ❑ Recap and homework solutions
- ❑ Common pitfalls
- ❑ Memory management
    - ❑ Image processing example
    - ❑ Sequence alignment
- ❑ Exercise
    - ❑ Area of the Mandelbrot set
- ❑ Monitoring and asynchronicity
    - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks

GPUs in bioinformatics : sequence alignment



Initialize the scoring matrix

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0								
G	0								
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

Substitution matrix: $s(a_i, b_j) = \begin{cases} +3, & a_i = b_j \\ -3, & a_i \neq b_j \end{cases}$

Gap penalty: $W_k = kW_1$
$W_1 = 2$

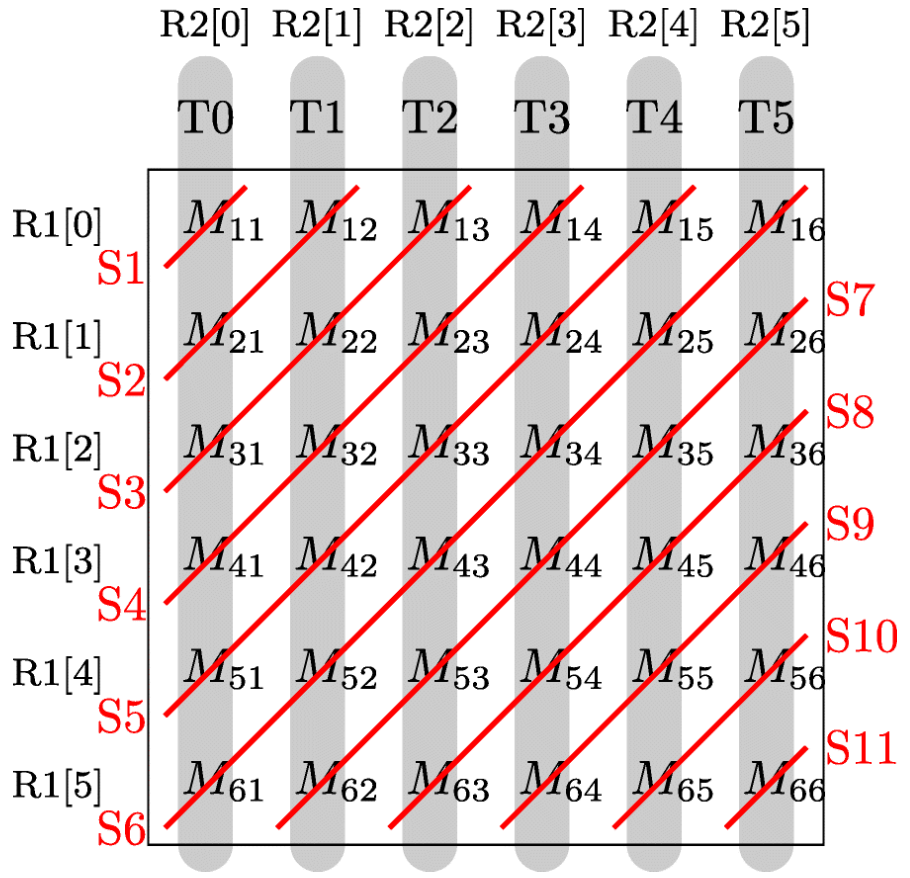
There are many variations, but sequence alignment algorithm often depends on filling a score matrix and then backtracking through it

Problem: matrix cells values depend on values of cells above and left. Parallelization is not trivial!

Solution: many methods optimize memory needed to compute a matrix at the scale of the SM, or even the thread.

Source: wikipedia user Yz cs5160, distributed under the CC BY-SA 4.0 license

GPUs in bioinformatics : sequence alignment



R2[0] R2[1] R2[2] R2[3] R2[4] R2[5]
T0 T1 T2 T3 T4 T5
R1[0] R1[1] R1[2] R1[3] R1[4] R1[5]
S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11
M_{11} M_{12} M_{13} M_{14} M_{15} M_{16}
M_{21} M_{22} M_{23} M_{24} M_{25} M_{26}
M_{31} M_{32} M_{33} M_{34} M_{35} M_{36}
M_{41} M_{42} M_{43} M_{44} M_{45} M_{46}
M_{51} M_{52} M_{53} M_{54} M_{55} M_{56}
M_{61} M_{62} M_{63} M_{64} M_{65} M_{66}

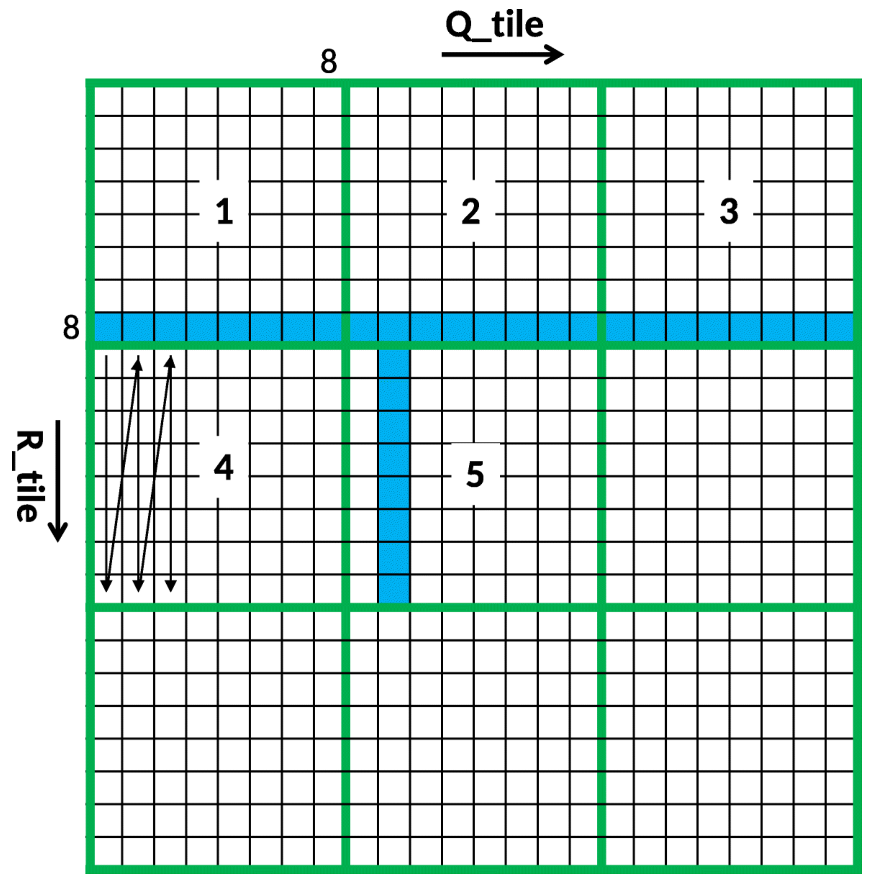
There are many variations, but sequence alignment algorithm often depends on filling a score matrix and then backtracking through it

Eg. GPU accelerated GATK haplotypeCaller

- Each SM computes a DP matrix
- At each call each thread computes 1 cell of an anti-diagonal
- Only need to remember at 2 previous anti-diagonal + backtracking info

Source: Figure from "GPU accelerated sequence alignment with traceback for GATK HaplotypeCaller" by Shanshan et al. BMC bioinformatics 2019

GPUs in bioinformatics : sequence alignment



There are many variations, but sequence alignment algorithm often depends on filling a score matrix and then backtracking through it

Eg. [GPU accelerated Darwin : de novo assembly of long reads]

- Each thread compute a tile of size TxT per GPU-invocation
- Only need to remember 1 row and a tile column
- all thread in a SM write their traceback to the same matrix in a coalesced fashion (additional x10 speedup)

Source: Figure from "GPU accelerated sequence alignment with traceback for GATK HaplotypeCaller" by Shanshan et al. BMC bioinformatics 2019

GPUs in bioinformatics

- Shift in thinking to optimize memory requirements
- Lots of new algorithms every year, not always easy to compare
- Mix between ready to use tools and code libraries

A few review articles:

- https://academic.oup.com/bib/article/18/5/870/2562773 (many application domains, but 2017)
- https://academic.oup.com/bib/article/22/5/bbab070/6210355 (different kind of parallelism, more limited)
- https://link.springer.com/chapter/10.1007/978-3-030-29407-6_15 (more generic while still focused on bio application)

Note : sequence alignment and bioinformatics-specifics algorithms are not the only everything
a lot of tasks in computational involve the matrix computations we have already seen
(computing distances/likelihoods, image analysis, ML , DL, ...)

Outline

Day 1
(09:00 – 13:00)

- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Homework

Day 2
(09:00 – 13:00)

- ❑ Recap and homework solutions
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
  - ❑ Sequence alignment
- ❑ Exercise
  - ❑ Area of the Mandelbrot set
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks

Outline

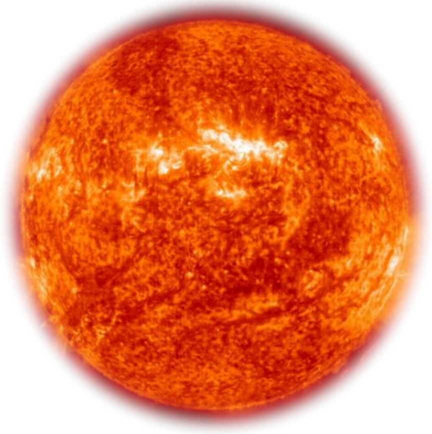
Day 1
(09:00 – 13:00)

- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Homework

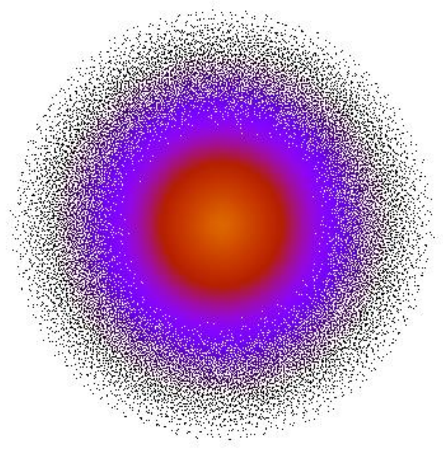
Day 2
(09:00 – 13:00)

- ❑ Recap and homework solutions
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
  - ❑ Sequence alignment
- ❑ Exercise
  - ❑ Area of the Mandelbrot set
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks

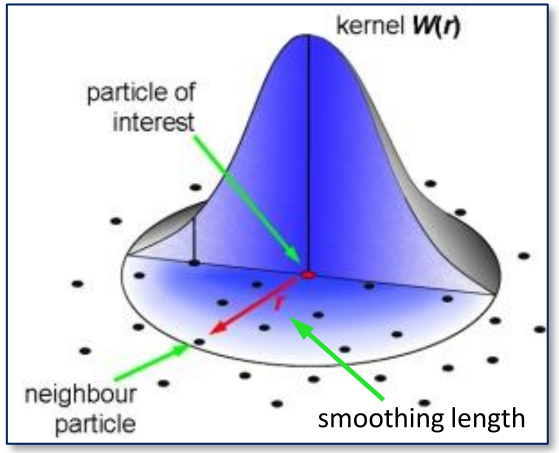
Smoothed Particle Hydrodynamics



Fluid



SPH particles



$$< f(\boldsymbol{r}) > = \int f(\boldsymbol{r}')W(\boldsymbol{r}'-\boldsymbol{r})d\boldsymbol{r}'$$

$$f_a(\boldsymbol{r}) = \sum_{b=1}^{n_v} \frac{m_b}{\rho_b} f_b(\boldsymbol{r}) W(\boldsymbol{r}_{ab}, h)$$

$$f(\boldsymbol{r}) = \rho$$

Loop over particles (*a*):

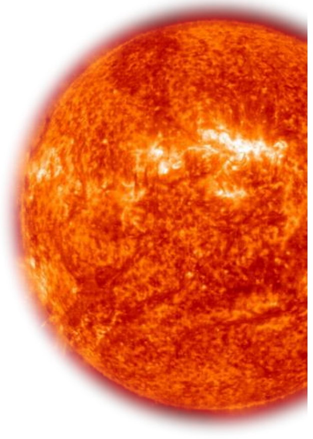
Loop over neighbors (*b*):

$$\rho_a = \sum_b m_b W_{ab}$$

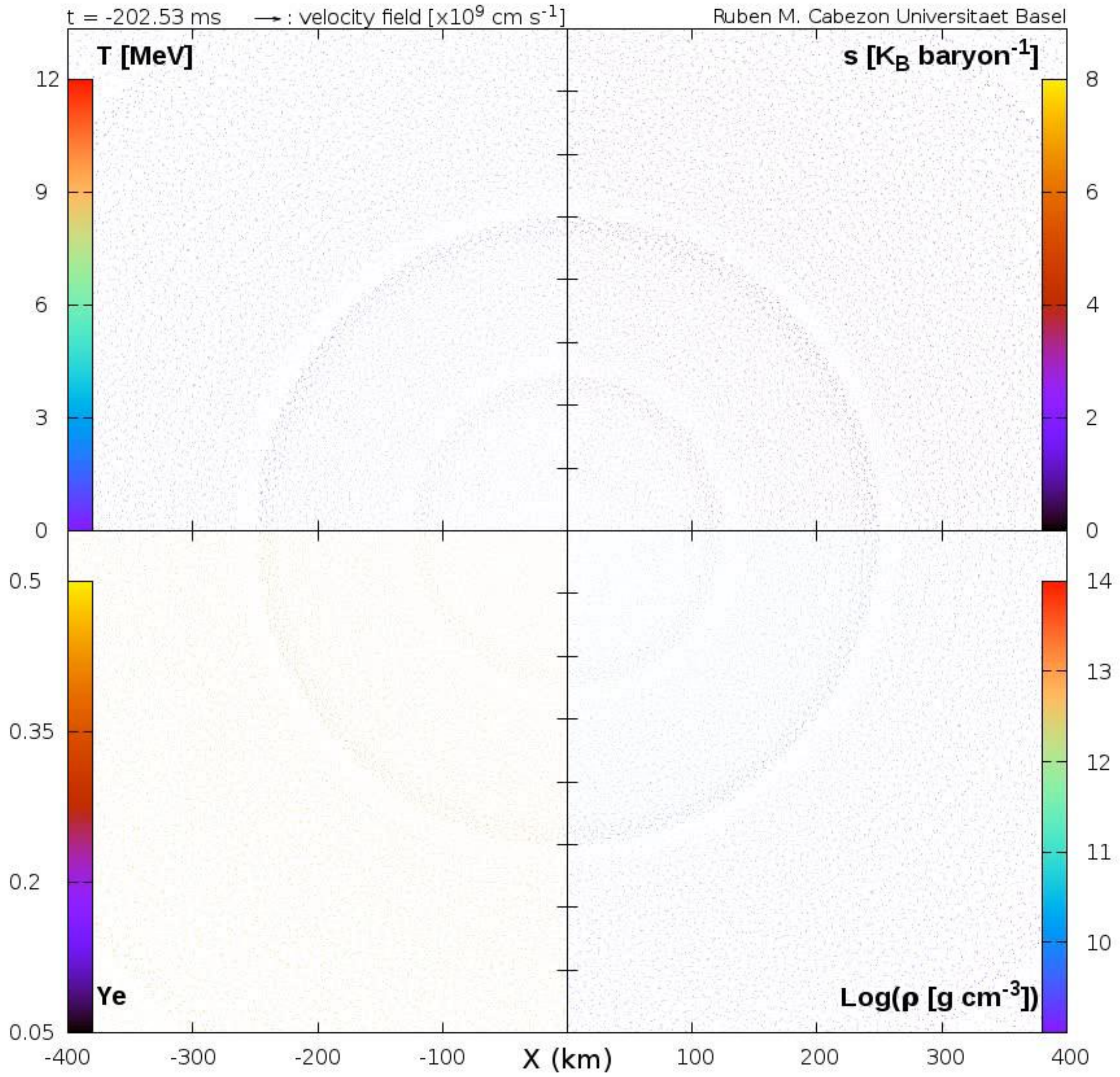
SPH interpolation kernel

$$W_n^H(v,h) = B_n(h)\begin{cases} 1 & , v = 0 \\ \left\{\mathrm{sinc}\left(\frac{\pi}{2}v\right)\right\}^n & , 0 < v \le 2 \\ 0 & , v > 2 \end{cases}$$

Smoothed Particle



Fluid



$)d\boldsymbol{r}'$

SPH inter

$W_n^H(v,h) = B_n$

$h)$

Outline

Day 1
(09:00 – 13:00)

- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Homework

Day 2
(09:00 – 13:00)

- ❑ Recap and homework solutions
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
  - ❑ Sequence alignment
- ❑ Exercise
  - ❑ Area of the Mandelbrot set
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks

GPUs at sciCORE

GPU	Nodes	CPUS/node	GPUs/node	RAM/node (GB)	Slurm Partition
Titanx	sgh[01-04] sgh[01-02]	28	6	512	titan
RTX4090	sgd[01-03]	128	8	1024	rtx4090
A100	sga[01-06] sgc[05-06]	128	4	1024	a100
A100 (80GB)	sgb01 sgj[01-02]	128	4	1024	A100-80g

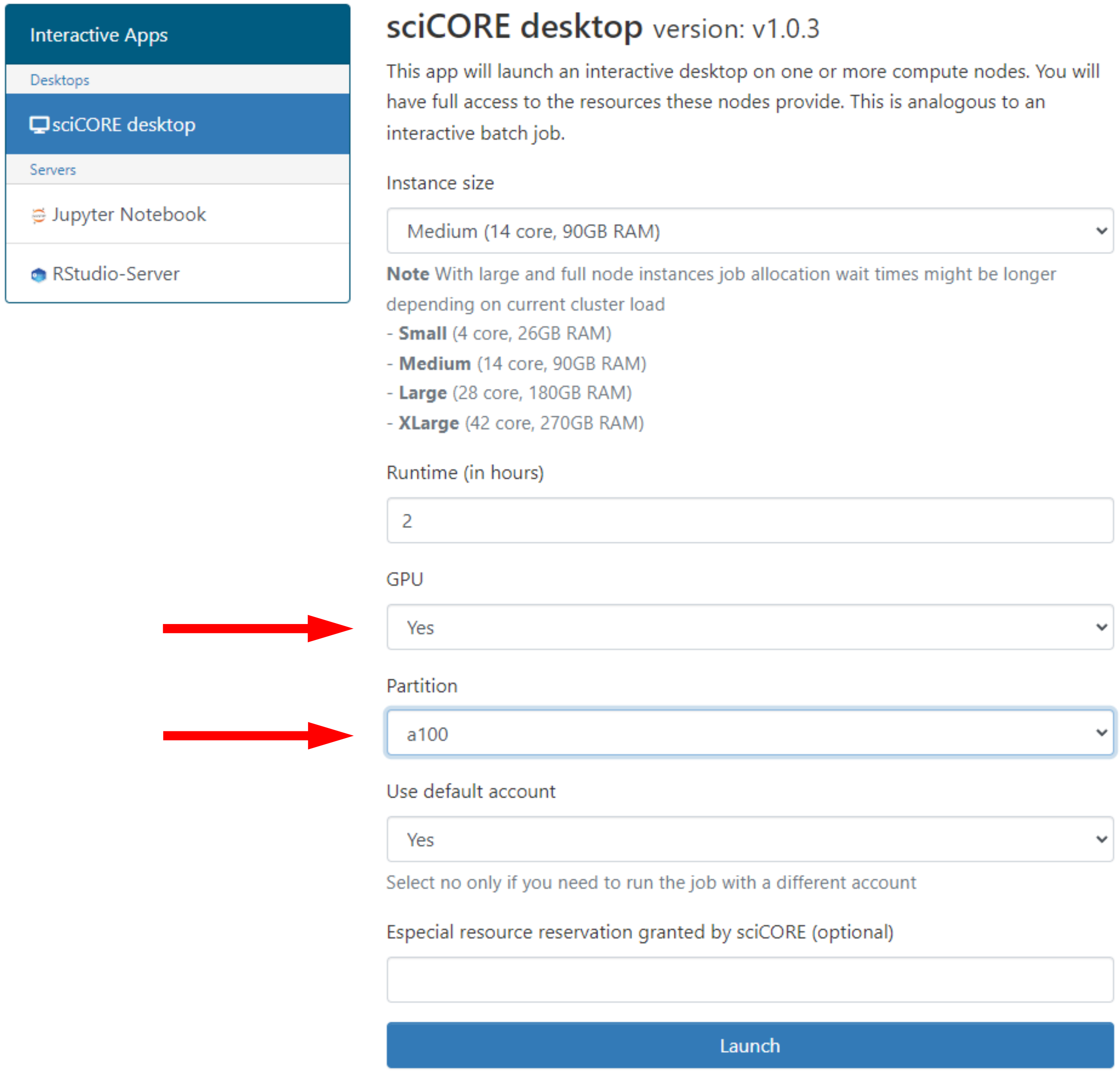
```
#!/bin/bash

#SBATCH --job-name=GPU_JOB
#SBATCH --time=01:00:00
#SBATCH --qos=gpu6hours
#SBATCH --mem-per-cpu=1G
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --partition=a100     # or titanx / rtx3090 / ...
#SBATCH --gres=gpu:1           # --gres=gpu:2 for two GPU,...

module load CUDA
.....
```

GPUs at sciCORE

You can also use OpenOnDemand to reserve one GPU at sciCORE.

https://ood.scicore.unibas.ch/

Useful for developing and debugging, or runnning short jobs interactively.



sciCORE desktop version: v1.0.3

This app will launch an interactive desktop on one or more compute nodes. You will have full access to the resources these nodes provide. This is analogous to an interactive batch job.

Instance size: Medium (14 core, 90GB RAM)

Note With large and full node instances job allocation wait times might be longer depending on current cluster load
- **Small** (4 core, 26GB RAM)
- **Medium** (14 core, 90GB RAM)
- **Large** (28 core, 180GB RAM)
- **XLarge** (42 core, 270GB RAM)

Runtime (in hours): 2

GPU: Yes

Partition: a100

Use default account: Yes

Select no only if you need to run the job with a different account

Especial resource reservation granted by sciCORE (optional):

Launch

Where to go from here?

Official documentation:

CUDA: https://docs.nvidia.com/
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
Numba: https://numba.readthedocs.io/en/stable/index.html
PyCUDA: https://documen.tician.de/pycuda/
OpenMP: https://www.openmp.org/specifications/
OpenCL: https://github.com/KhronosGroup/OpenCL-Guide
OpenACC: https://www.openacc.org/resources

OpenMP:

Offloading support in GCC: https://gcc.gnu.org/wiki/Offloading
OpenMP offloading talk: https://www.youtube.com/watch?v=m0MqPYmlGbM

Optimization in GPUs:

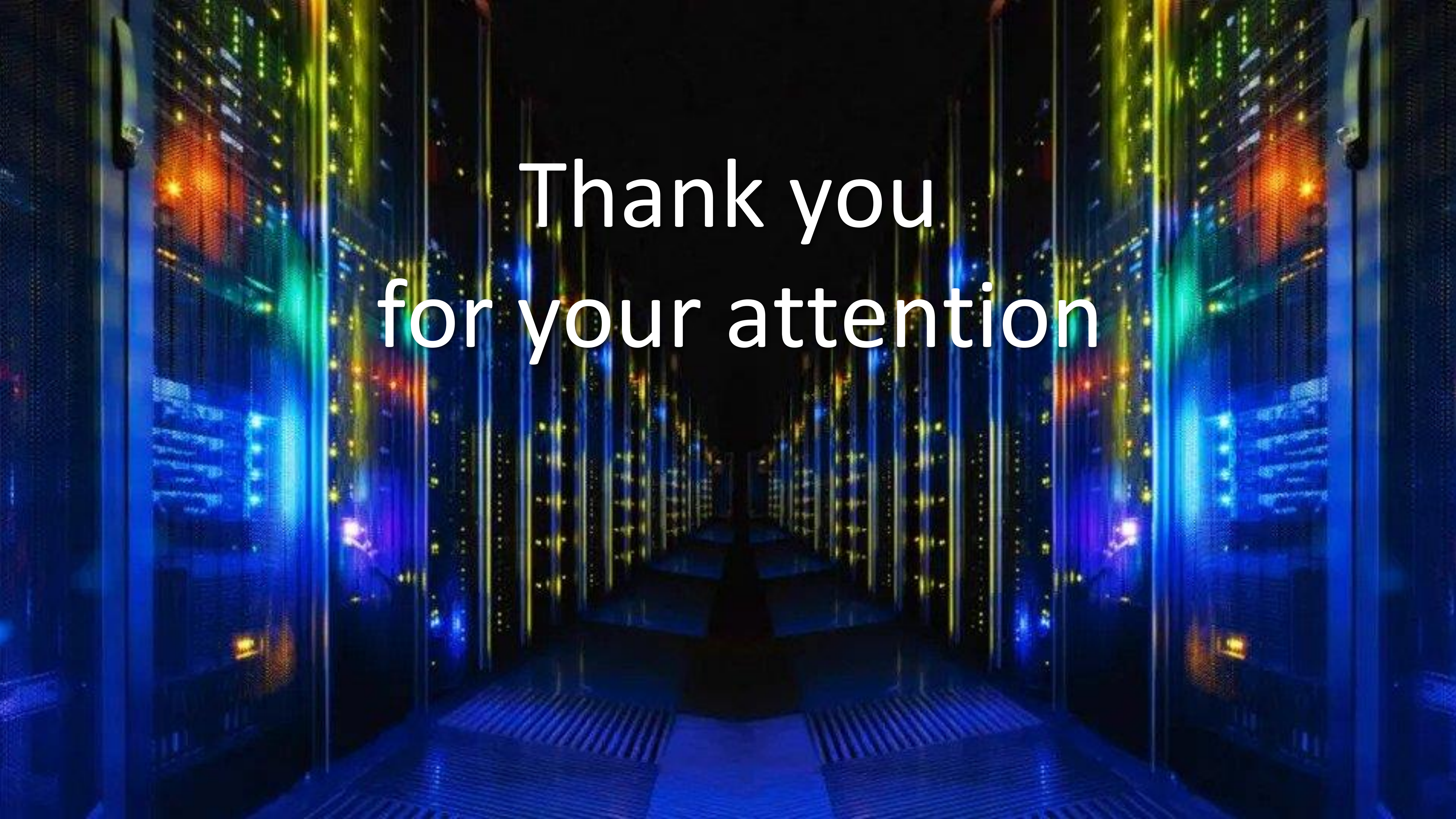
GPU optimization strategies: https://www.paranumal.com/single-post/2018/02/26/basic-gpu-optimization-strategies
Memory types: https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/
Using shared memory: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

CUDA:

nVidia CUDA training : https://developer.nvidia.com/accelerated-computing-training
3rd-party CUDA training: https://developer.nvidia.com/educators/existing-courses
Numba CUDA: https://nyu-cds.github.io/python-numba/05-cuda/

Numba:

Numba intro: https://www.youtube.com/watch?v=6oXedk2tGfk
Numba pitfalls: https://www.youtube.com/watch?v=x58W9A2lnQc



Thank you
for your attention