ALGO DOERS

GPU Training Days

Hatem Ltaief

CTO AlgoDoers
Chercheur KAUST

PhD in Computer Science, 2007

University of Houston, USA

Parallel Numerical Algorithms

High-Performance Computing

Mixed-Precision Computations

Valentin Le Fèvre

Ingénieur HPC
AlgoDoers

PhD in Computer Science, 2020

Ecole Normale Supérieure, France

High-Performance Computing

Numerical Linear Algebra

Energy Efficiency

Online Anonymous Survey: Get to Know you!

GPU Training

Collaborate, Innovate, Accelerate

- Format
  - Hands-on, two intensive afternoons
  - Goal: Optimize, accelerate, and scale your codes using GPUs
  - Work directly with experts and mentors in GPU computing
- Hardware/software: remote access to Toubkal GPU partition
- Team formation (if needed)
  - Create teams
  - Group programming, brainstorming, and debugging together
- Experience first-hand GPU acceleration
- Report lively progress updates

Objectives and Plan

- Understand how a GPU works
- Handle GPU memory, parallelism
- Use of standard vendor libraries for linear algebra (cuBLAS)
- Write CUDA kernels
- Analyze performance of an application / a kernel

Discover your GPU with cuBLAS

GPU specs

GPU memory

GPU events

test case: GEMM - dense matrix multiply

Application profiling

CUDA programming

Write your **own kernels**

Understand thread parallelism

Shared memory

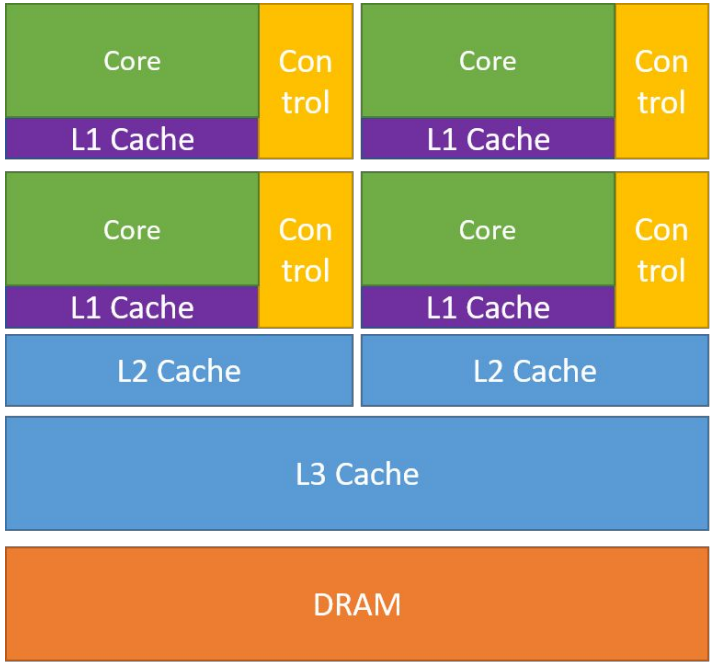
Kernel profiling

Optimization

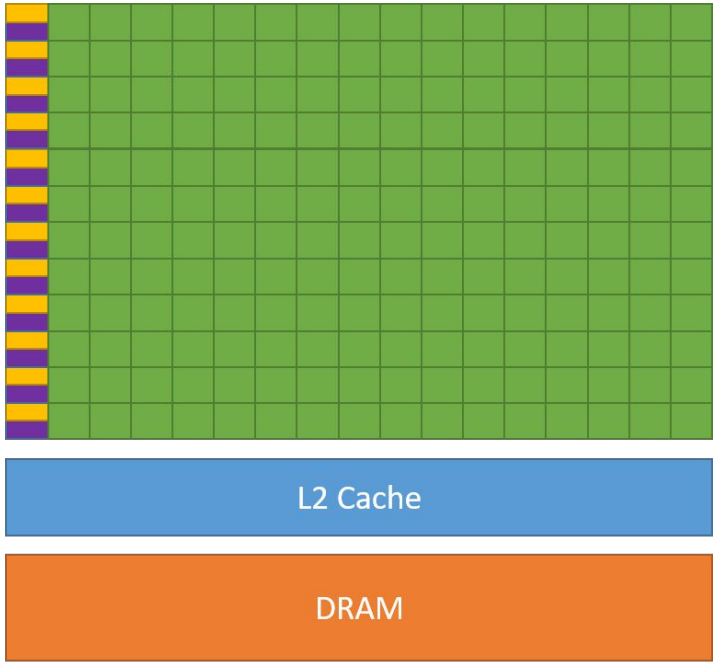
What is a GPU?

- Massive parallelism
- CPU: few threads, very fast
- GPU: many threads, slower

To sum up, a GPU is less flexible than a CPU but more efficient in **time** and **energy** to execute massively parallel task.



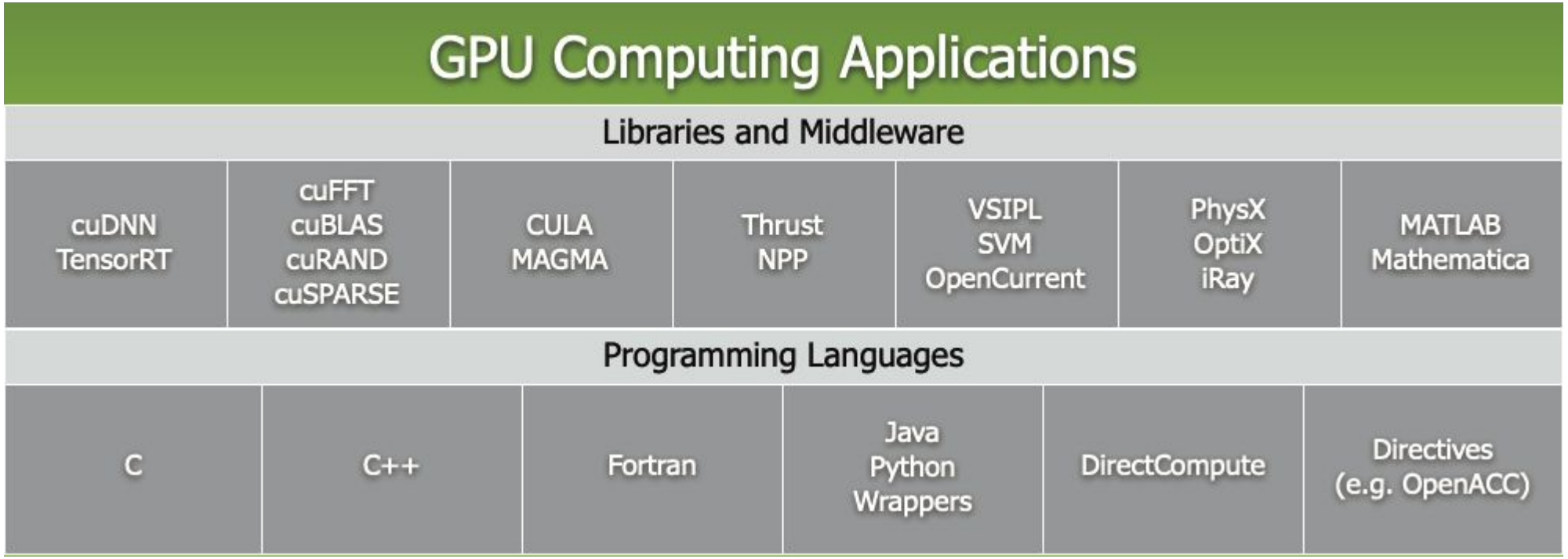
CPU



GPU

CUDA Programming Model

CUDA is a programming model designed to exploit parallelism on NVIDIA GPUs. Based on C.



Architecture GPU

A GPU embed several SM:
Streaming Multiprocessors

Each SM can execute a block of threads

L2 cache is shared among SMs

GPU Architecture

Each SM is divided into blocks

Example with *Ampere* architecture:
4 sub-blocks

Each block embeds:

- 16 INT32 processing units
- 16 FP32 processing units
- 8 FP64 processing units
- 1 tensor core (3rd generation)

- Warp scheduler
- Registers

CUDA abstraction

All GPUs are different.
CUDA will help the programmer with some abstractions, which automatically dispatches data and code execution to the SMs.

We will see CUDA programming later ;)

1st objective:

Learn how to use a GPU using vendor libraries.
Our test case will be a GEMM (GEneral Matrix-Multiply) kernel, using the cuBLAS library.

CUDA Tools

nvcc

2-step compilation:

- Source code → PTX
- PTX → SASS

cuda-gdb compute-sanitizer

Debugging tools:

- cuda-gdb works like gdb
- compute-sanitizer detects memory errors, synchronization, ...

nsys

Nsight Systems:

- Overview of the whole application
- CUDA, memory transfers...

ncu

Nsight Compute:

- Kernel profiling
- GPU occupancy, resource usage, ...

Profiling: Nsight Systems (1)

Nsight Systems is used to profile the whole application.
It can measure/report:

- calls to CUDA API
- data transfers
- GPU kernel executions
- system calls, CPU execution, multiple threads, ...
- calls to libraries (ex: cuBLAS)
- OpenMP and MPI communications

In summary, Nsight Systems allows us to have **insights at the scale of the whole application**. The trace is saved in a .nsys-rep file, that can be visualized with the GUI tool **nsys-ui**.

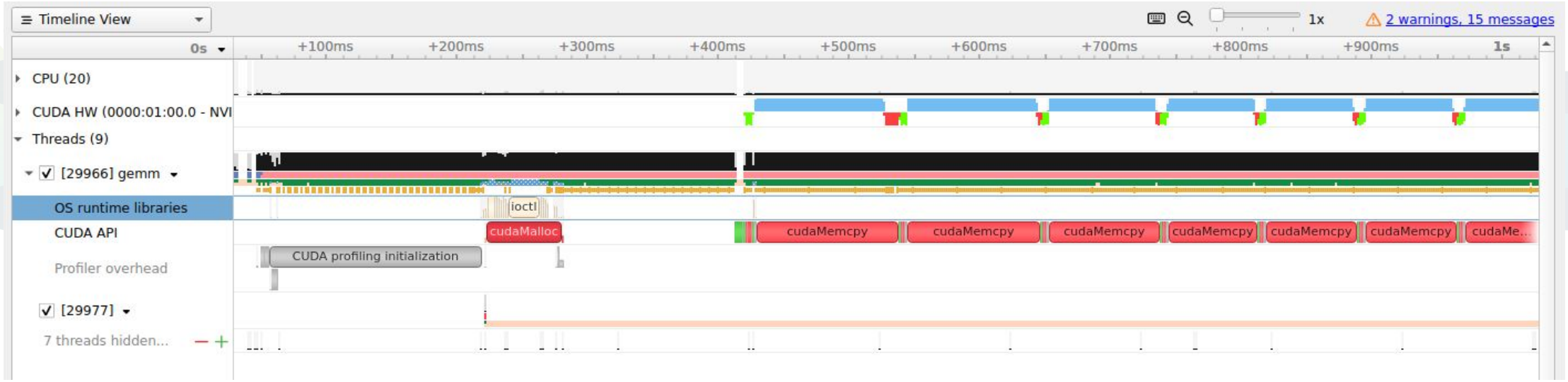
Command-line usage:

```
nsys profile -o report -t cuda ./myapp
nsys-ui report.nsys-rep
```

Profiling: Nsight Systems (2)

The GUI allows to:

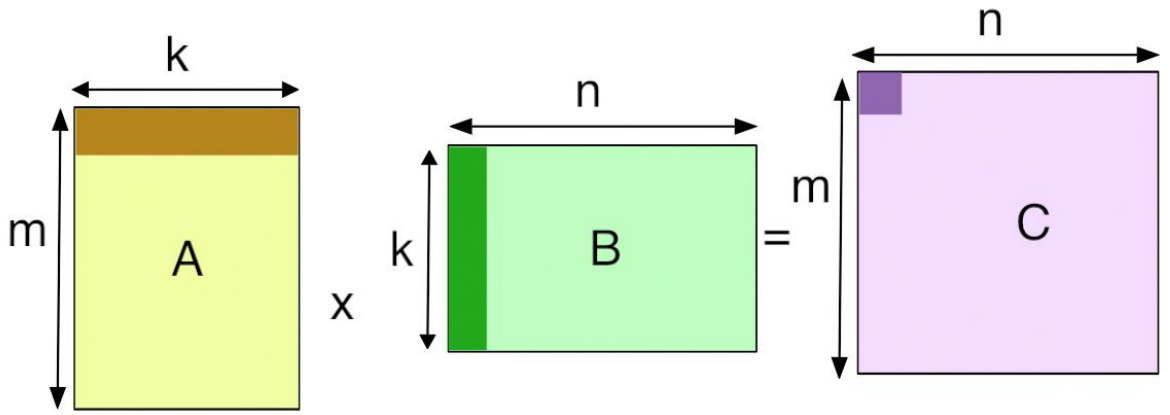
- launch profiling commands directly from there
- visualize the traces



Your first case study: Dense Matrix Multiplication

Multiply two matrices is a fundamental operation found in many applications: numeric simulations, matrix factorization, deep learning, ...

This is a compute-bound operation (i.e. essentially made of computations) which is highly parallelizable: perfect for a GPU !



GEMM

GEneral Matrix Multiplication

$$C \leftarrow \alpha * op(A) * op(B) + \beta * C$$

A, B and C are our 3 matrices: A,B as input and C as input and output.
op(A) is either A or A^T. We will focus on the standard case with no transposition.
3 integer parameters define the GEMM operation: M, N, K.

- A is a matrix of size MxK
- B is a matrix of size KxN
- C is a matrix of size MxN

α et β are two scalar parameters.

Matrix data structure

If A is a matrix of size MxN, we use an array of size MN.

```
float* ptrA = malloc(sizeof(float) * M * N);
```

2 formats are possible: Row-Major,

Column-Major.

- Row-Major: elements are stored row by row.
- **Column-Major: elements are stored column by column.**

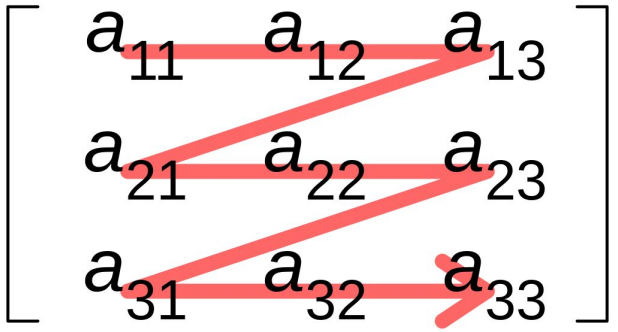
Element A(i,j) is:

```
ptrA[i * ld + j]
```

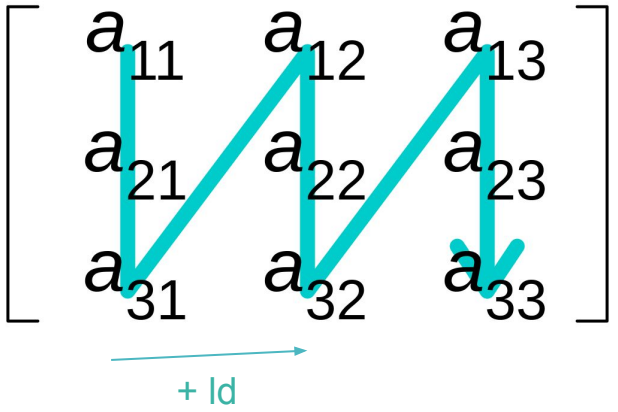
ptrA[i + j * ld]

FOR US: ld (leading dimension) is **the stride between two consecutive elements on the same row**.
Most of the time, it will be equal to the number of rows (M).

Row-major order



Column-major order



GEMM standard call

gemm(order, opA, opB, M, N, K, α, ptrA, ldA, ptrB, ldB, β, ptrC, ldC);

- order: specify row-major or column-major order. In CBLAS: CblasRowMajor/CblasColMajor. In cuBLAS: does not exist (library assumes column-major ordering).
- opA: specify if A has to be transposed or not. In CBLAS: CblasNoTrans/CblasTrans. In cuBLAS: CUBLAS_OP_N/CUBLAS_OP_T.
- opB: specify if B has to be transposed or not.
- M: number of rows of matrices A and C.
- N: number of columns of matrices B and C.
- K: number of columns of matrix A and number of rows of matrix B.
- α: scaling parameter for AB.
- ptrA: pointer to memory layout of A.
- ldA: leading dimension of A.
- ptrB: pointer to memory layout of B.
- ldB: leading dimension of B.
- β: scaling parameter for C.
- ptrC: pointer to memory layout of C.
- ldC: leading dimension of C.

Download starting files

https://drive.google.com/drive/u/0/folders/1trrn7FFMAt4_RVBnO0nsSVwjq9NY-HLd

Cluster usage

To setup the working environment:

- module load CUDA
  - gives access to the whole CUDA toolkit (nvcc, vendor libraries, …)
- module load GCC/12.3.0
  - for compatibility with nvcc
- module imkl
  - Intel MKL setup (for CPU execution)

Useful slurm commands:

- sbatch *script*: **submit a job to the cluster**, which executes *script*
- squeue: check status of pending jobs
- scancel *job_id*: cancel job with ID *job_id*

Slurm script

An example of slurm script is available:
exec.slurm

- adjust time limit if you need (--time)
- choose output file name (--output)

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --time=00:10:00
#SBATCH --partition=gpu
#SBATCH --reservation=gpu_hackathon
#SBATCH --gres=gpu:1
#SBATCH --output=slurm.out
#SBATCH --account=MANAPY-1WABCJWE938-DEFAULT-GPU

commands to execute (bash script)
```

Do not execute a program directly in the login node, **submit a job**!

Edit the script file (with vim, nano, …) when you want to execute different commands

Discover your GPU

- Compile the deviceQuery.cu source file with:
  nvcc deviceQuery.cu -o deviceQuery
- edit the file **exec.slurm** to execute:
  ./deviceQuery
- launch the job:
  sbatch exec.slurm
- check the output:
  cat slurm.out

Discover your GPU (2)

Main information:

- CUDA Compute Capability (CC): “version” of GPU architecture
- Global memory: slow memory on GPU (RAM)
- Shared Memory per block/SM: shared memory size per threadblock/SM
- Registers available per block/SM: number of registers per threadblock/SM
- Max. number of threads per block/SM: max. number of threads per SM
- Max dimension size of thread block/grid size: max. dimension of threadblocks/grid
- Warp Size: always 32 for NVIDIA

```
Device 0: "NVIDIA GeForce RTX 4070 Laptop GPU"
  CUDA Driver Version / Runtime Version          12.2 / 12.2
  CUDA Capability Major/Minor version number:    8.9
  Total amount of global memory:                 7943 MBytes (8328511488 bytes)
  (036) Multiprocessors, (128) CUDA Cores/MP:    4608 CUDA Cores
  GPU Max Clock rate:                            1230 MHz (1.23 GHz)
  Memory Clock rate:                             8001 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 33554432 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        102400 bytes
  Total number of registers available per block: 65536
  Total number of registers available per SM:    65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

Task #1: run GEMM on CPU

blasGemm(order, opA, opB, M, N, K, α, ptrA, ldA, ptrB, ldB, β, ptrC, ldC);

Open your file gemm.cu and find the “TODO: TASK 1” parts. This is where you should be writing your code.
You need to:

- allocate 3 matrices A, B and C (type elem_t*) of sizes MxK, KxN, MxN;
- initialize the matrices at random;
- call the function blasGemm (from CBLAS) with A,B and C;
- free the matrices;
- measure the average execution time/performance of the gemm call (use var *timesCPU*);
- play with parameters M,N,K: the execution time should be proportional to MNK.

```
make gemm
vim exec.slurm
sbatch exec.slurm
```

The file **utils.h** contains many helper functions that you should use: allocateMatrixCPU, freeMatrixCPU, initMatrixCPU, initMatrixRandomCPU, computeTime. A GEMM represents 2MNK floating-point operations (flops).

https://www.netlib.org/lapack//explore-html/de/da0/cblas_8h_a1446cddceb275e7cd299157a5d61d5e4.html

GPU memory handling

A GPU kernel only has access to GPU memory !

Allocate/Free memory on GPU:

```
cudaMalloc(void **mem_ptr, size_t size);        cudaFree(void *mem_ptr);
```

Send data from CPU to GPU:

```
cudaMemcpy(void *dest, void *src, size_t size, cudaMemcpyHostToDevice);
```

Send data from GPU to CPU:

```
cudaMemcpy(void *dest, void *src, size_t size, cudaMemcpyDeviceToHost);
```

GPU memory handling

Example:

```
float *h_array, *d_array;
cudaMallocHost(&h_array, N*sizeof(float));
//Initialize h_array
cudaMalloc(&d_array, N*sizeof(float));
cudaMemcpy(d_array, h_array, N*sizeof(float), cudaMemcpyHostToDevice);
//…
//Process on GPU
//…
cudaMemcpy(h_array, d_array, N*sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(d_array);
cudaFreeHost(h_array);
```

CUDA Events

clock_gettime() can be used on CPU to measure GPU execution times, but **synchronization is mandatory**

CUDA has a useful tool called **events**.
CUDA kernel launches are asynchronous with respect to the CPU, but they still get executed in the order they are issued. We can insert events on the GPU and measure the time between two events.

```
float time;
cudaEvent_t begin, end;
cudaEventCreate(&begin);
cudaEventCreate(&end);

cudaEventRecord(begin);
mykernel(params); //function that we measure
cudaEventRecord(end);

cudaEventSynchronize(end); //Wait for event end to be recorded
cudaEventElapsedTime(&time, begin, end); //milliseconds
cudaEventDestroy(begin);
cudaEventDestroy(end);
```

Error handling

In case of a problem:

- for CUDA errors (cudaMemcpy, cudaMalloc, ...):
  - cudaError_t error
  - cudaGetErrorString(error)
  - error = cudaDeviceSynchronize(); error = cudaMemcpy(...); ...
  - value is cudaSuccess without error
- for cuBLAS errors (cublasGemm, ...):
  - cublasStatus_t status
  - cublasGetStatusString(status)
  - status = cublasSgemm(...)
  - value is CUBLAS_STATUS_SUCCESS without error

Task #2.1: run GEMM on GPU

cublasGemm(handle, opA, opB, M, N, K, α, ptrA, ldA, ptrB, ldB, β, ptrC, ldC);

In utils.h and gemm.cu, find the “TODO: TASK 2.1” parts.
You need to:

- write the functions allocateMatrixGPU, freeMatrixGPU;
- allocate A, B and C on the GPU (using d_A, d_B, d_C);
- copy the matrix elements from CPU to GPU (from A to d_A, …);
- free the matrices on the GPU.

```
make gemm
sbatch exec.slurm
```

Task #2.2: run GEMM on GPU

cublasGemm(handle, opA, opB, M, N, K, α, ptrA, ldA, ptrB, ldB, β, ptrC, ldC);

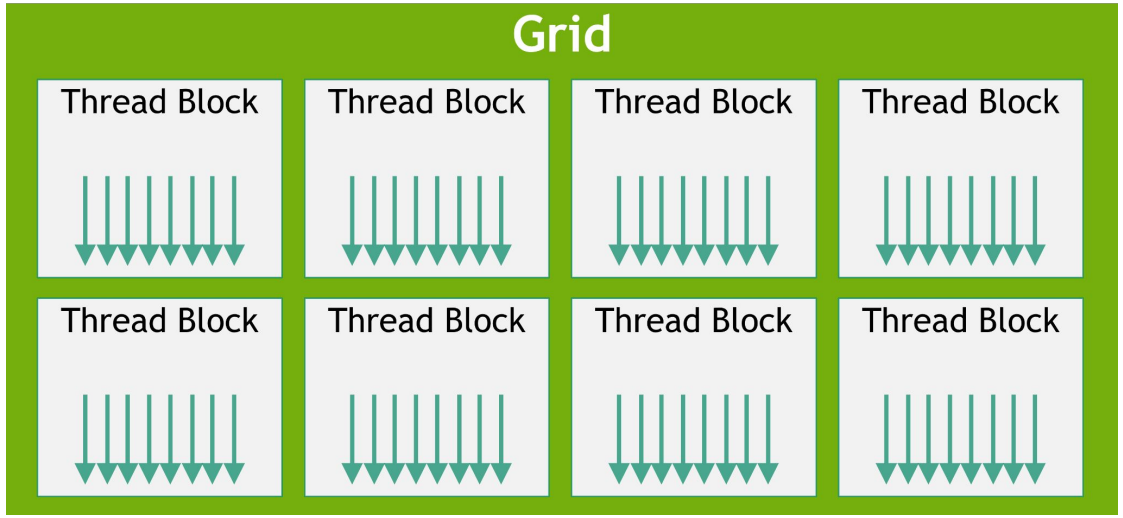
In gemm.cu, find the “TODO: TASK 2.2” parts.
You need to:

- call the function cublasGemm (from cuBLAS); you need call it **inside the two loops**: one form warmup (the GPU is slow for the first executions) and one for real execution. The first parameter is the “handle”, which already exists in your base code.
- measure the execution times with CUDA events (var timesGPU);
- compare the result from the GPU to the result from the CPU: be careful, you first need to transfer the result from the GPU onto the CPU! You can use the function compareMatrices from utils.h and variable Cgpu;
- compare the performance of CPU and GPU for different matrix sizes.

```
make gemm
sbatch exec.slurm
```

https://docs.nvidia.com/cuda/cublas/index.html#cublas-t-gemm

First steps in CUDA ! (1)



We have seen how to use libraries to use a GPU.
What if we want to **write our own functions** ?

1. define a GPU kernel
2. set the number of threads to run it
3. use thread index to parallelize the code

1. CUDA introduces 3 keywords: __host__, __kernel__ and __global__.
   - __host__ a function running on CPU
   - __device__ a function running on GPU called from GPU
   - __global__ a function running on GPU called from GPU or CPU

Example:

```
__global__ void myFunc(float *myParam)
{
        …
}
```

2. CUDA uses a hierarchical structure:
   - a grid is composed of threadblocks
   - a threadblock is composed of threads

The number of threadblocks in the grid and number of threads in each threadblock is set when launching the kernel to the GPU:

- myFunc<<<4,32>>>(myParams): we launch the kernel myFunc with the parameters myParams using a grid of 4 threadblocks of 32 threads each: 128 threads will execute the function.

First steps in CUDA ! (2)

3. All threads launched will execute the same code!

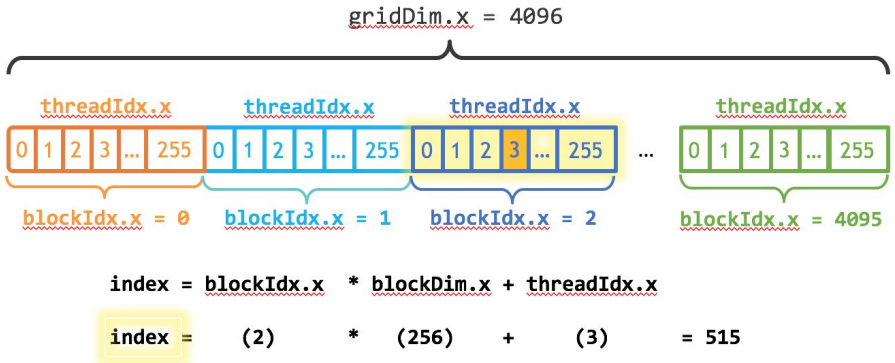
Built-in variable allows to identify the different threads:

- threadIdx.x: index of thread in a threadblock
- blockDim.x: number of threads in a threadblock
- blockIdx.x: index of threadblock in the grid
- gridDim.x: number of threadblocks in the grid

Example: multiply each element of an array T with a constant C

```
__global__ void multiply(int N, float *T, float C) {
      int index = blockIdx.x * blockDim.x + threadIdx.x;
      if (index >= N)
            return;
      T[index] = T[index]*C;
}
```

multiply<<<4096,256>>>(N, T, C);



index is unique to each thread ! Be careful with the bounds ! How many blocks to launch?

Profiling: Nsight Compute (1)

Nsight Compute is a tool to profile **GPU kernels**.
Several metrics can be gathered using sections and sets. By default, set *basic* is selected.

In summary, Nsight Compute can be used to have a detailed analysis of each GPU kernel, and the trace is stored in a .ncu-rep file. It can then be visualized with ncu-ui.

Some useful commands:

- ncu –list-sets : displays the list of available sets.
- ncu –set name : selects the set called *name*.
- ncu –list-sections : displays the list of available sections.
- ncu –section name : selects the section called *name*.

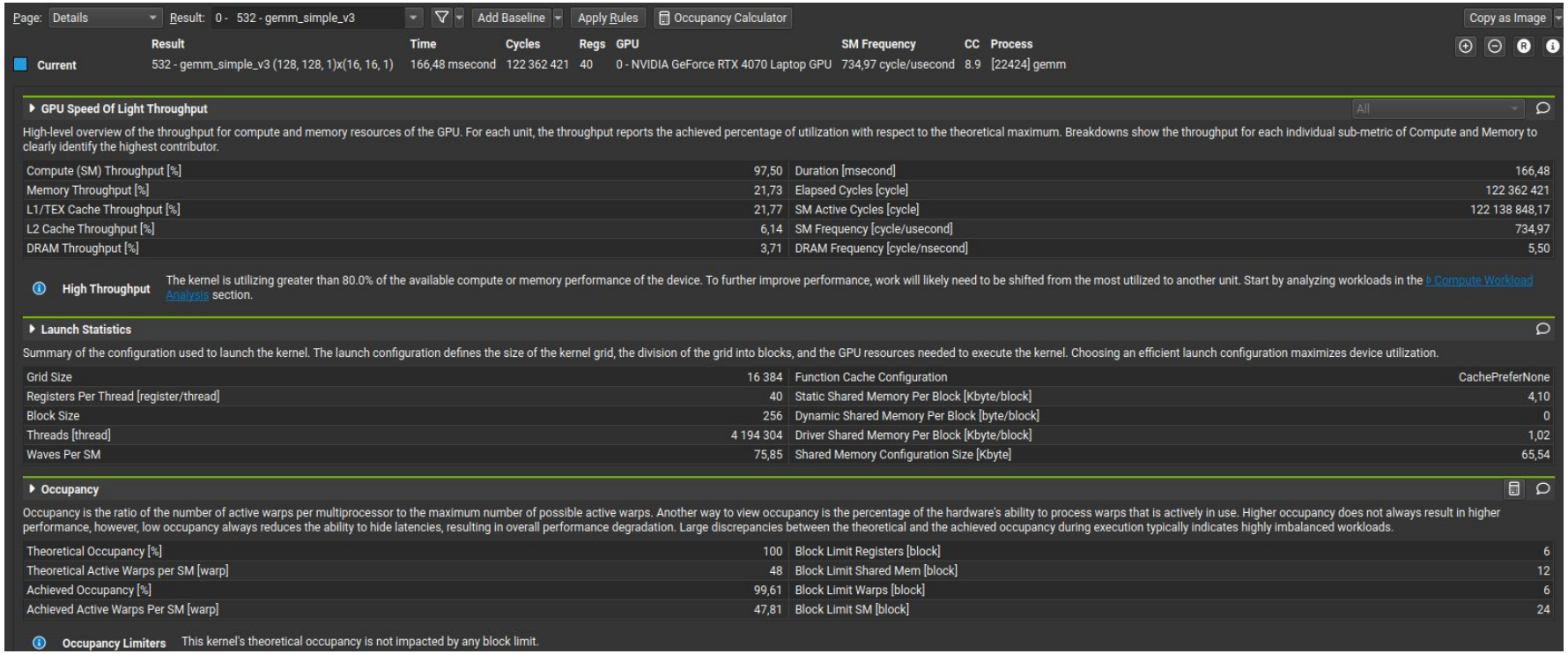
To profile, overwrite the output, with all metrics available and then visualize:

```
ncu -o profile -f --set full ./myApp
ncu-ui profile.ncu-rep
```

TMPDIR should be set in multi-users sessions to avoid concurrency.

Profiling: Nsight Compute (2)

Nsight Compute also has a GUI tool that sums up the different metrics collected and also gives hints about performance improvement for a kernel.



ROOFLINE PERFORMANCE MODEL (1)

- Performance model (simple)
- Performance assessment
- Sets up the performance expectation
- Identify performance bottlenecks
- Performance upper-bounds
- Architecture-oriented model
- Three ingredients:
  - Communication
  - Computation
  - Locality

ROOFLINE PERFORMANCE MODEL (2)

- At the level of a kernel, determine:
  - Floating-point operations per seconds (FLOPS/s)
  - Arithmetic intensity (AI)

- AI: kernel's ratio of computation to traffic
  - FLOPS per byte

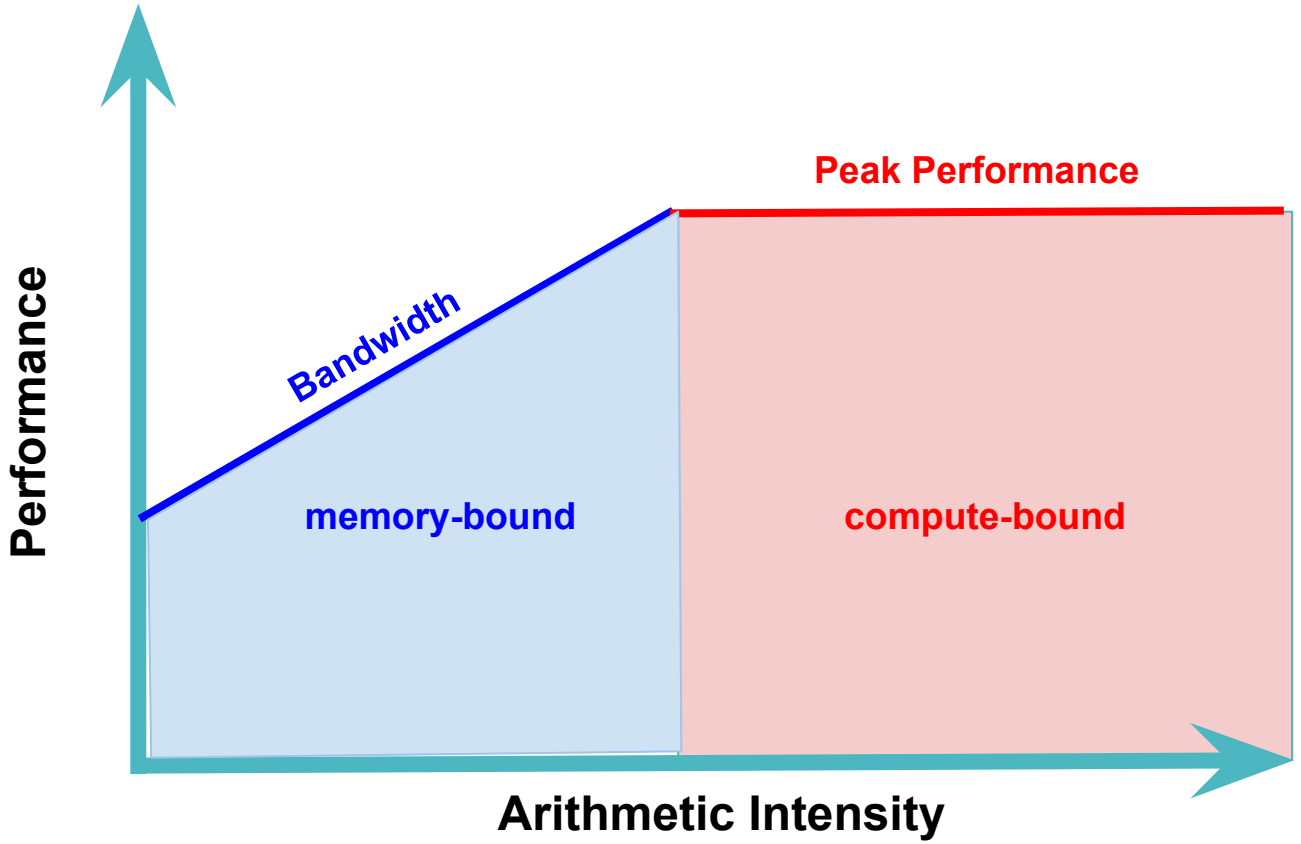
- Traffic is the volume of data to/from memory

- Compute-bound Vs Memory-bound

ROOFLINE PERFORMANCE MODEL (3)

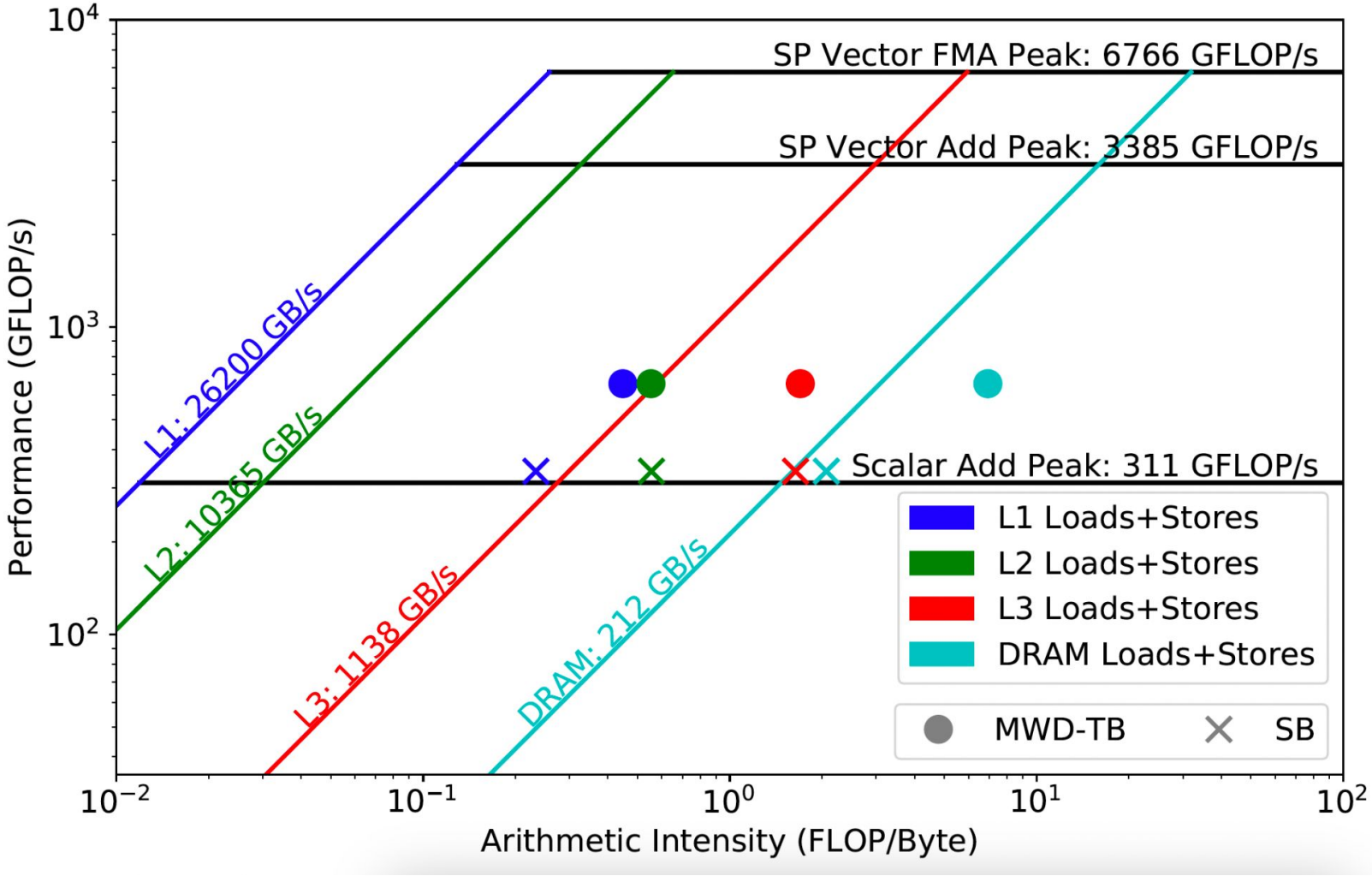
- Theoretical peak performance
  - Number of floating-point operations per seconds (*X*flops/s)
- Theoretical peak bandwidth
  - Number of byte transferred from main memory per seconds (*X*bytes/s)
- Vendor-defined from hardware specifications
- Many tools available to determine the roofline of the underlying hardware

ROOFLINE PERFORMANCE MODEL (4)



ROOFLINE PERFORMANCE MODEL (5)

two-socket 26-core Intel Skylake
Intel advisor / Likwid



Occupancy

An important factor for performance is called Occupancy.
An occupancy of 100% means that each SM is used at maximum capability.
The theoretical occupancy can be computed with:

Th-Occ = (max nb of threadblocks per SM * nb of threads per threadblock / 32) / max nb of warps per SM

Max number of warps per SM is equal to the max number of threads per SM divided by 32, e.g.
max 1536 threads → max 48 warps.

The number of threads per threadblock is set by the user.

The max number of blocks per SM depends on several factors: number of registers that the kernel needs, the amount of shared memory per threadblock, and the number of threads per threadblock.

Nsight Compute will give you all this information ;) It can also show the roofline model.

Task #1: launch your custom implementation of GEMM on a GPU

Open the file cuda_gemm.cu. You will find already implemented: allocation of the matrices, memory transfers, a call to CPU blas_gemm as reference. All you will need to do is: write the function gemmV1 and call it !

Indications:

- remember that threads work in parallel so we need independent writes. **Make one thread compute one single element of the output matrix C**.
- number of threads and number of threadblocks can be 2D (threadIdx.**x/y**, blockDim.**x/y**, blockIdx.**x/y**, ...) with dim3 gridSize(M,N); dim3 blockSize(m,n); kernel<<<gridSize, blockSize>>> will launch MN blocks with mn threads.
- play with the different parameters (size of matrices, number of threads per block, ...) and look at the performance / profile with Nsight Compute.
- compute-sanitizer --tool=initcheck ./cuda_gemm
- compute-sanitizer --tool=memcheck ./cuda_gemm

GEMM pseudo-code for CPU:

```
for i=0,...,M-1 do
    for j=0,...,N-1 do
        sum = 0
        for k=0,...,K-1 do
            sum += A[i,k] * B[k,j]
        C[i,j] = α * sum + β * C[i,j]
```

```
make cuda_gemm
sbatch exec.slurm
```

Coalesced memory accesses

Accessing memory is costly: it should be accessed as efficiently as possible

SIMT execution model: a whole warp executes simultaneously.
Memory can be issued multiple times to fetch the whole data.
Data is loaded by sectors of 32+ bytes.

> "as a general rule, the more scattered the addresses are, the more reduced the throughput is"
>
> CUDA Programming Guide

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Slow

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Fast

Task #1.2: check your global memory accesses

Indications:

- check your global memory accesses in gemmV1
  - if they are not coalesced, change the mapping of your threads to the data layout and check the performance
  - if they are coalesced, try “transposing” your mapping of threads to the data layout to uncoalesce the accesses
- Compare the performance boost (or drop) between the two versions. You can also check with Nsight Compute that accesses are coalesced or not.

```
make cuda_gemm
sbatch exec.slurm
```

Thread communication through Shared Memory

One of the most critical part when writing high performance GPU kernels is the data transfers:

- global memory (RAM): high latency, high storage
- shared memory: average latency, limited storage
- registers: quick memory, very limited storage

Elements from the global memory have to be loaded to registers and/or shared memory, but the bandwidth is slower.
Goal: reduce number of accesses to global memory!

Shared memory can be declared with **__shared__** inside a GPU kernel: one allocation per threadblock.
All threads in the same threadblock can read/write into the same shared memory: communication is possible.
Synchronization will be necessary in most cases.

```
__shared__ int smem[256];

//Threads can write into smem

__syncthreads();

//Threads can read into smem
```

Shared memory can also be configured through a 3rd parameter in kernel launch and adding the keyword *extern* in the declaration inside the GPU kernel code.

```
myFunc<<<gridSize, blockSize, 256*sizeof(int)>>>(myParams);
```

Task #2: improve performance with Shared Memory

Open the file cuda_gemm.cu. The goal is to write the function gemmV2, making use of the shared memory to reduce the number of global memory accesses.

Indications:

- shared memory is very limited! You won't be able to store a whole row of A (or a whole column of B) if K is large.
- a threadblock should compute a small rectangle of C: if you compute a row or a column you will re-use less data. The good idea is to iteratively load tiles of A and B of size tileMxtileK and tileKxtileN to compute a tile of C of size tileMxtileN (for each threadblock).
- play with the different parameters (size of matrices, number of threads per block, ...) and look at the performance / profile with Nsight Compute.

```
make cuda_gemm
sbatch exec.slurm
```

Task #3: Higher arithmetic intensity

Open the file cuda_gemm.cu. The goal is to write the function gemmV3, increasing the arithmetic intensity compared to gemmV2.

Remarks:

- increasing TILE_M and/or TILE_N reduces the number of *global* memory loads for computing one sub matrix of size TILE_MxTILE_N
- extreme case: TILE_M=M and TILE_N=N. We have no parallelism between blocks and limited number of threads → each thread must compute several results
- we can adopt a 2-level blocking strategy: each threadblock computes a tile of the result (higher tile size = less *global* memory loads) and each thread computes a sub-tile of each tile (higher sub-tile size = less *shared* memory loads)
- currently a thread: two SM loads for 1 multiplication/addition, quite bad

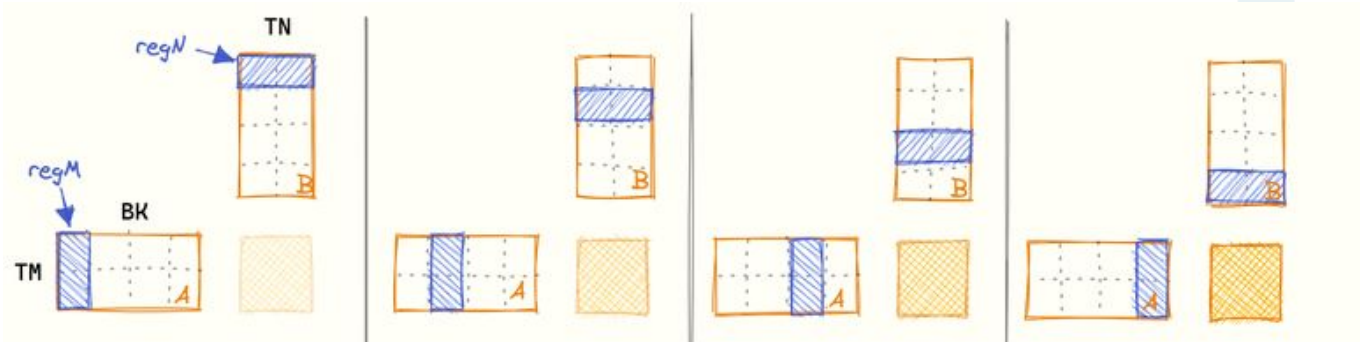
```
make cuda_gemm
sbatch exec.slurm
```

Task #3: Higher arithmetic intensity

Open the file cuda_gemm.cu. The goal is to write the function gemmV3, increasing the arithmetic intensity compared to gemmV2.

Indications:

- one thread will compute a sub-tile THREAD_MxTHREAD_N of the result, stored in registers
- one threadblock will compute a sub-tile TILE_MxTILE_N of the result
- adapt TILE_K wisely
- choose the right number of threads to launch per thread block
- try it and analyze with Nsight Compute !
- outer-loop algorithm to reduce SM loads



Online Anonymous Survey (AFTER)