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

© AlgoDoers

Online Anonymous Survey (BEFORE)

© AlgoDoers

GPU Hackathon

Collaborate, Innovate, Accelerate

- Format
  - Hands-on, intensive 1.5 days
  - Goal: Optimize, accelerate, and scale your codes using GPUs
  - Work directly with experts and mentors in GPU computing
- Team formation
  - Create teams
  - Group programming, brainstorming, and debugging together
- Morning check-ins and evening progress updates
- Experience first-hand GPU acceleration
- Hardware/software: remote access to Toubkal GPU partition
- Winning team
  - best performance
  - best engaging
  - best team spirit
  - best innovation

© AlgoDoers

Objectives and Plan

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

1

Discover cuBLAS

GPU memory

GEMM: dense matrix multiply

CUDA streams

Batched Algorithms

Profiling

2

Performance study: Cholesky

Apply what we learned from GEMM

3

CUDA

Write your own kernels

Understand thread parallelism

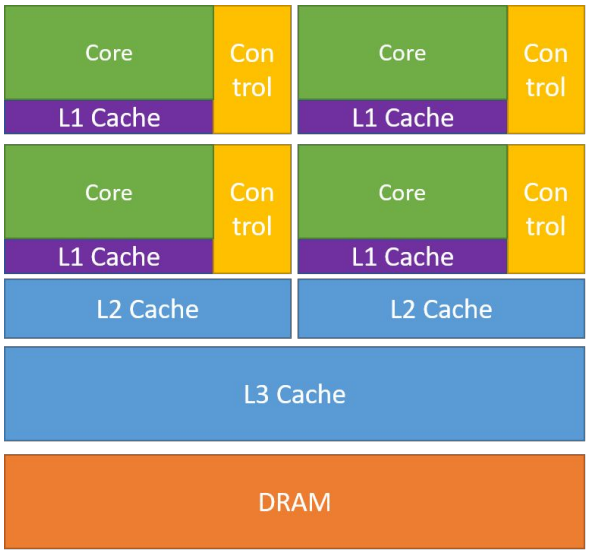
Shared memory

© AlgoDoers

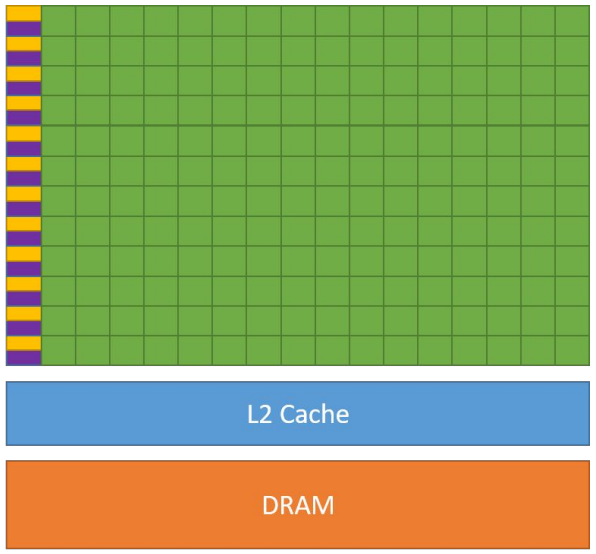
What is a GPU?

- 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

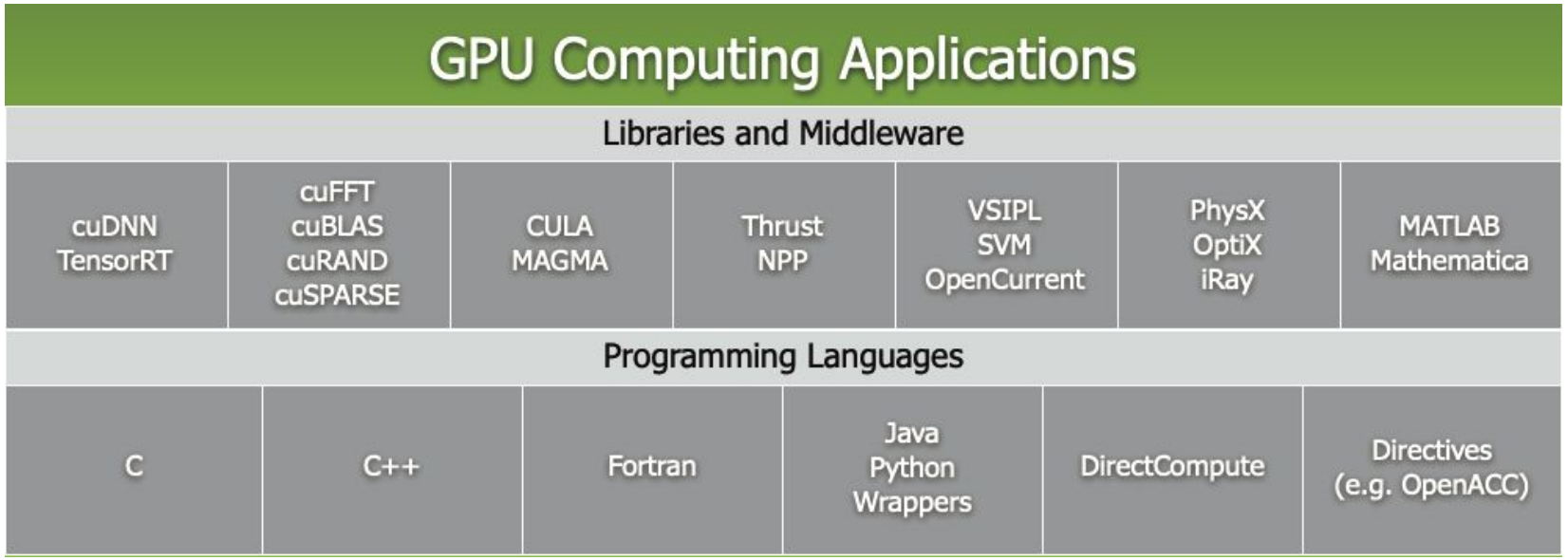


GPU

© AlgoDoers

CUDA Programming Model

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



© AlgoDoers

Architecture GPU

A GPU embed several SM:
Streaming Multiprocessors

Each SM can execute a block of threads

L2 cache is shared among SMs

© AlgoDoers

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

© AlgoDoers

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

Debugging tool:
- Like gdb

nsys

Nsight Systems:
- Overview of the whole application
- CUDA, memory transfers...

ncu

Nsight Compute:
- Kernel profiling
- GPU occupancy, resource usage, ...

© AlgoDoers

Discover your GPU

- Compile the deviceQuery.cu source file with:
  nvcc deviceQuery.cu -o deviceQuery -arch=sm_XX
- XX is the compute capability of your GPU. Example for *Ampere* (A100, A40, …) : 80
- run the executable with:
  ./deviceQuery

© AlgoDoers

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) >
```

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:

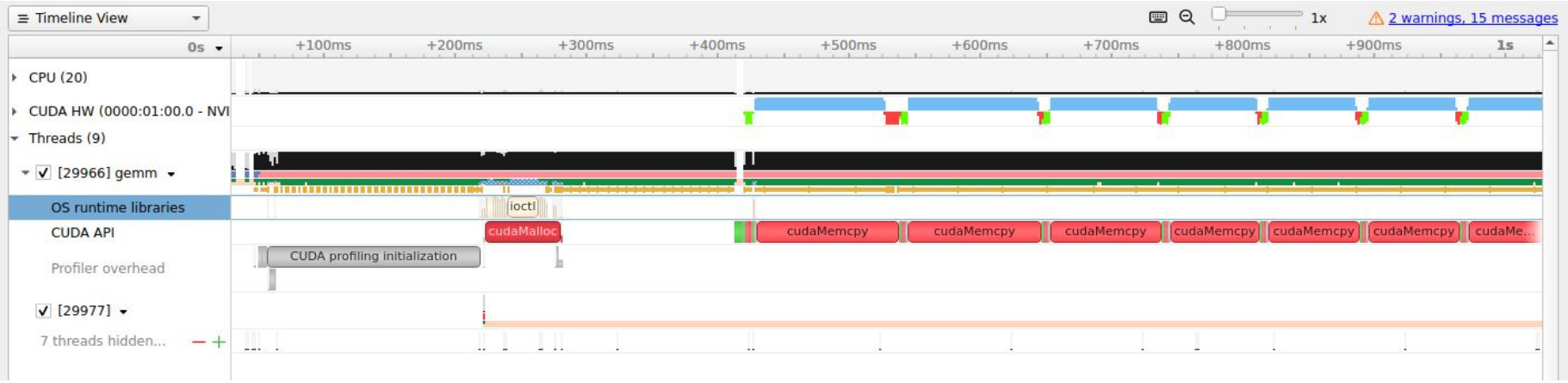
```
nsys profile -o report -t cuda ./myapp
nsys-ui report.nsys-rep
```


© AlgoDoers

Profiling: Nsight Systems (2)

The GUI allows to:

- launch profiling commands directly from there
- visualize the traces

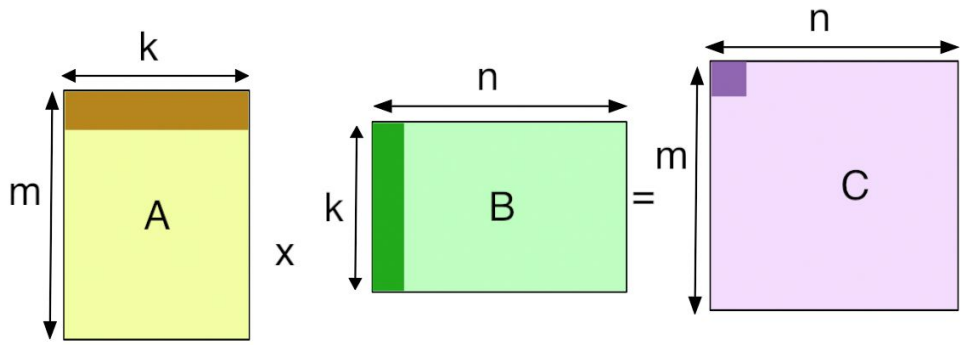


© AlgoDoers

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 !



© AlgoDoers

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.

© AlgoDoers

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.**

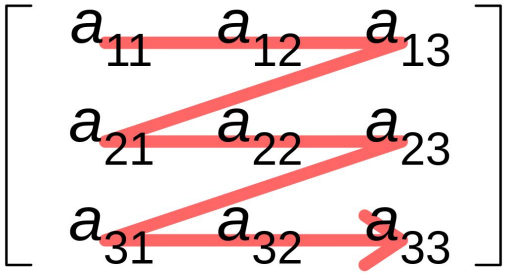
Element A(i,j) is:

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

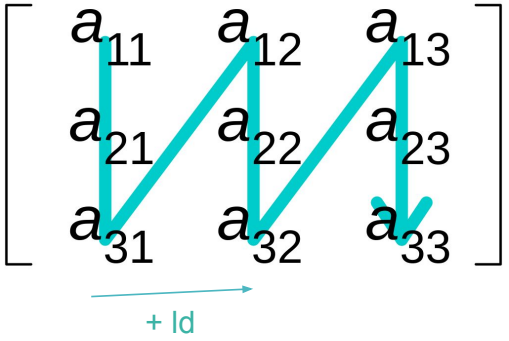
```
ptrA[i + j * ld]
```

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



© AlgoDoers

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.

© AlgoDoers

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*

An example of slurm script is available: *exec.slurm*

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

© AlgoDoers

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.

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).

make gemm && ./gemm

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);
```


© AlgoDoers

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);
```


© AlgoDoers

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);
```


© AlgoDoers

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

© AlgoDoers

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 && ./gemm

© AlgoDoers

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 && ./gemm

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

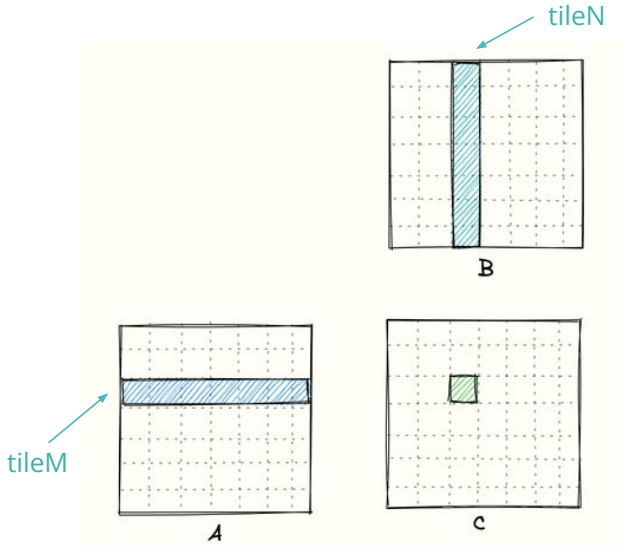
© AlgoDoers

Task #3: tiled GEMM

tileGemm(handle, M, N, K, α, ptrA, ldA, ptrB, ldB, β, ptrC, ldC, tileM, tileN);

In gemm.cu, find the "TODO: TASK 3" parts.
You need to:

- write the function tileGemm. This function has to compute the global GEMM αAB+βC by computing sequentially the different tiles of C of size tileMxtileN;
- call the function in the two main GPU loops;
- look at the execution times/performance when varying the parameters



make gemm && ./gemm

© AlgoDoers

CUDA Streams

CUDA streams can be seen as different queues of execution:

- operations in the same stream are executed sequentially
- operations in two different streams **can** be executed simultaneously
- “up to 32 streams at the same time”

We were already using a special stream: **the default stream**.
ANY OPERATION SUBMITTED IN THE DEFAULT STREAM WILL WAIT FOR ALL STREAMS TO FINISH AND PREVENT ANY STREAM TO EXECUTE IN PARALLEL

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cublasSetStream(handle, stream1);
cublasFunc(params);
cublasSetStream(handle, stream2);
cublasFunc(params);
cudaEventRecord(endKernel2,stream2);
cudaStreamWaitEvent(stream1, endKernel2);
cublasSetStream(handle, stream1);
cublasFunc(params);
cudaStreamSynchronize(stream1);
cublasSetStream(handle, NULL); //Default stream

cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```

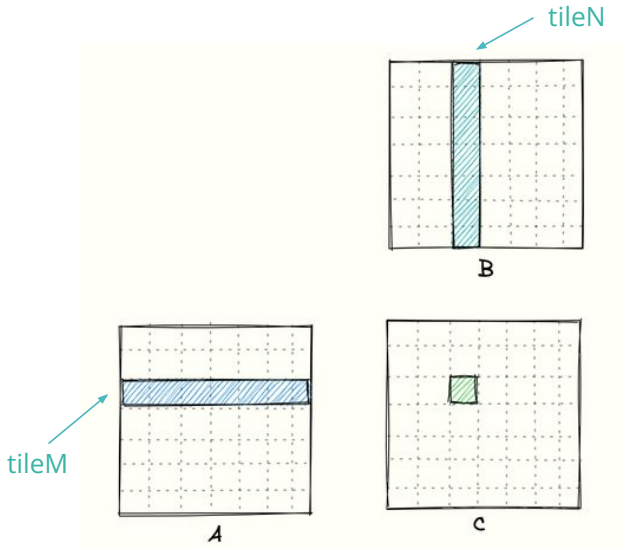

© AlgoDoers

Task #4: tiled GEMM with Streams

tileGemmStreams(handle, M, N, K, α, ptrA, ldA, ptrB, ldB, β, ptrC, ldC, tileM, tileN, int nb_streams, cudaStream_t *streams);

In gemm.cu, find the "TODO: TASK 4" parts.
You need to:

- write the function tileGemmStreams. This function is equivalent to tileGemm but you have to parallelize the different tasks that can be executed simultaneously!
- call the function in the two main GPU loops;
- look at the execution times/performance when varying the parameters



make gemm && ./gemm

© AlgoDoers

Task #5: tiled GEMM with Batch

tileGemmBatch(handle, M, N, K, α, ptrA, ldA, ptrB, ldB, β, ptrC, ldC, tileM, tileN);

Batched GEMM algorithm: parameter "batch count" is the number of GEMM to do (they must have same sizes!). Only one kernel call to the GPU !

https://docs.nvidia.com/cuda/cublas/#cublas-t-gemmbatched

In gemm.cu, find the "TODO: TASK 5" parts.
You need to:

- write the function tileGemmBatch. This function is equivalent to tileGemm but you have to parallelize the different tasks that can be executed simultaneously with a batch algorithm! **You have to call cublasGemmBatched instead cublasGemm**.
- call the function in the two main GPU loops;
- look at the execution times/performance when varying the parameters

make gemm && ./gemm

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

© AlgoDoers

Your second case study: Cholesky factorization

The goal of Cholesky factorization is to determine, for a hermitian definite-positive matrix M, a *lower* triangular matrix L such that **LL^T = M**

Your goal will be to benchmark this operation on CPU (using LAPACKE/MKL), on GPU (using cuSOLVER and Magma).

You should help yourselves with the code for GEMM, online documentations and the following plan:

- The cholesky operation is denoted by "**POTRF**" in linear algebra libraries
- First, how do you create a matrix M which symmetric definite-positive ?
- Like GEMM, start with a CPU execution, then try to make it run with cuSOLVER. cuSOLVER is very similar to cuBLAS: you have to create a cusolverDnHandle_t object, and errors can be caught with a cusolverStatus_t object (value != CUSOLVER_STATUS_SUCCESS)
- Measure the execution times and performance (number of Flops = $n^3/3$ with M of size nxn)
- Compute multiple cholesky factorizations with batched algorithms
- Compare to Magma library when everything is working with cuSOLVER, good luck !