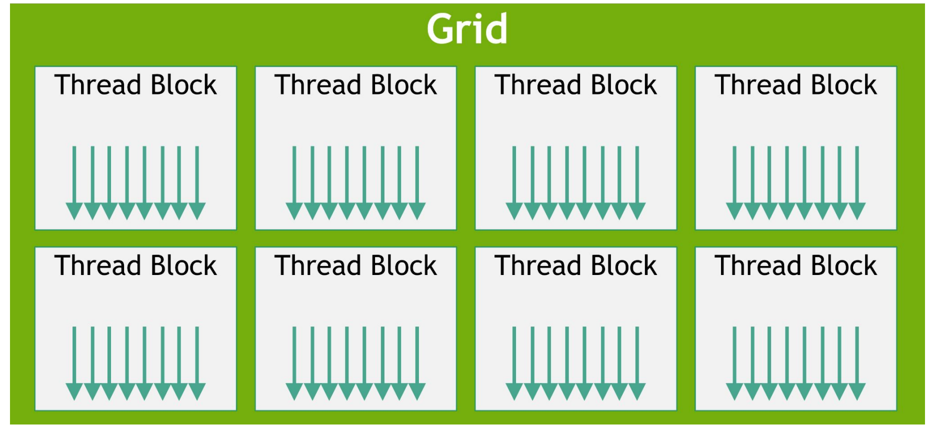
First steps in CUDA ! (1)



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!

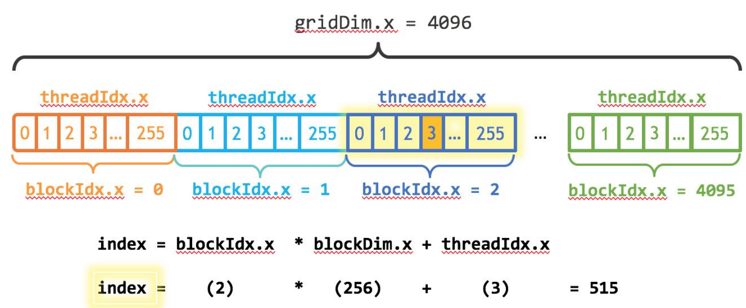
Built-in variable allows to identify the different threads:

- 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*.

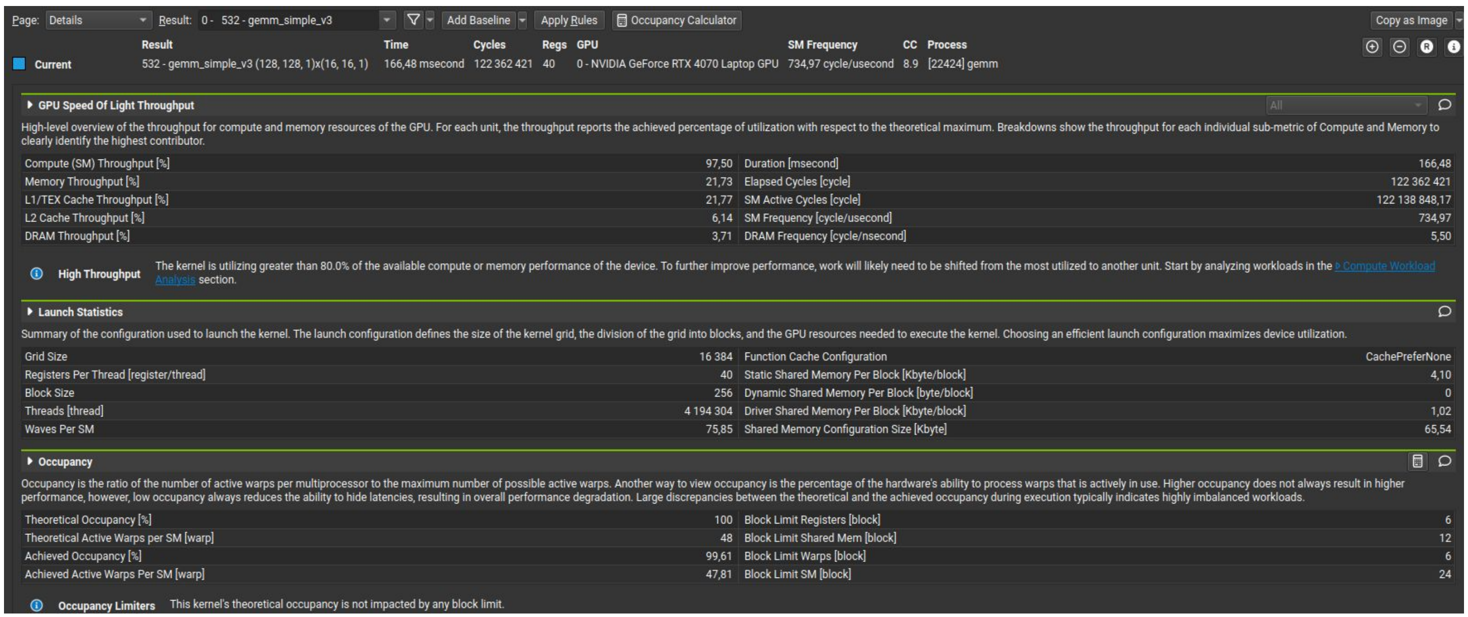
To profile, overwrite the output, with all metrics available and then visualize:

```
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

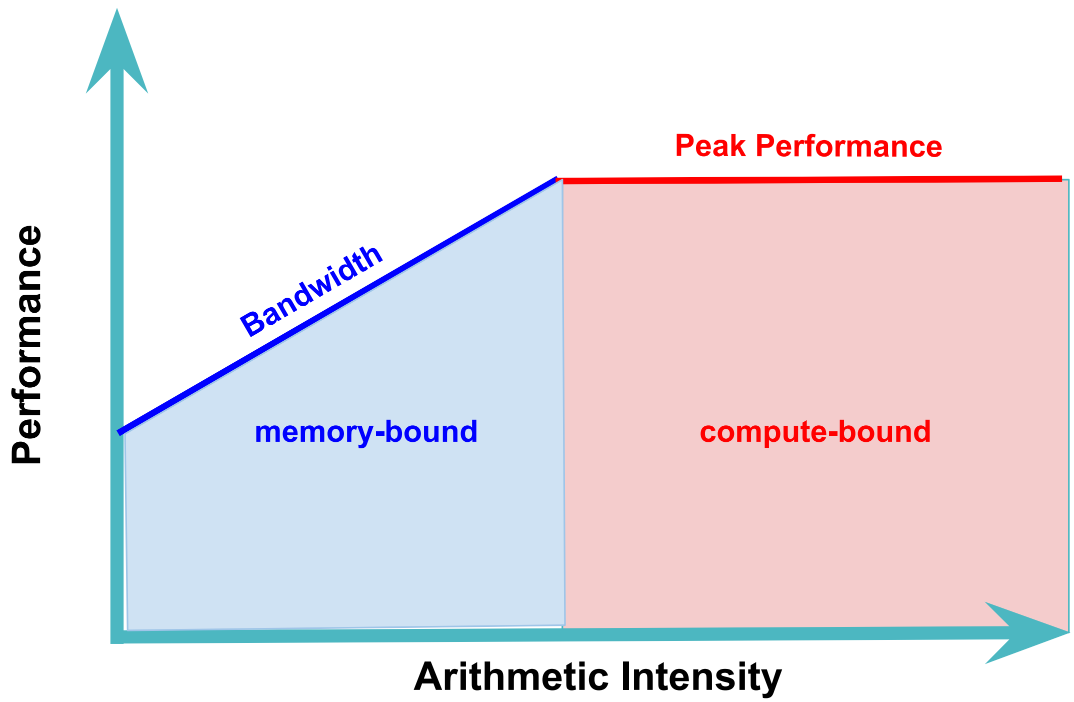
- Traffic is the volume of data to/from memory

- Compute-bound Vs Memory-bound

ROOFLINE PERFORMANCE MODEL (3)

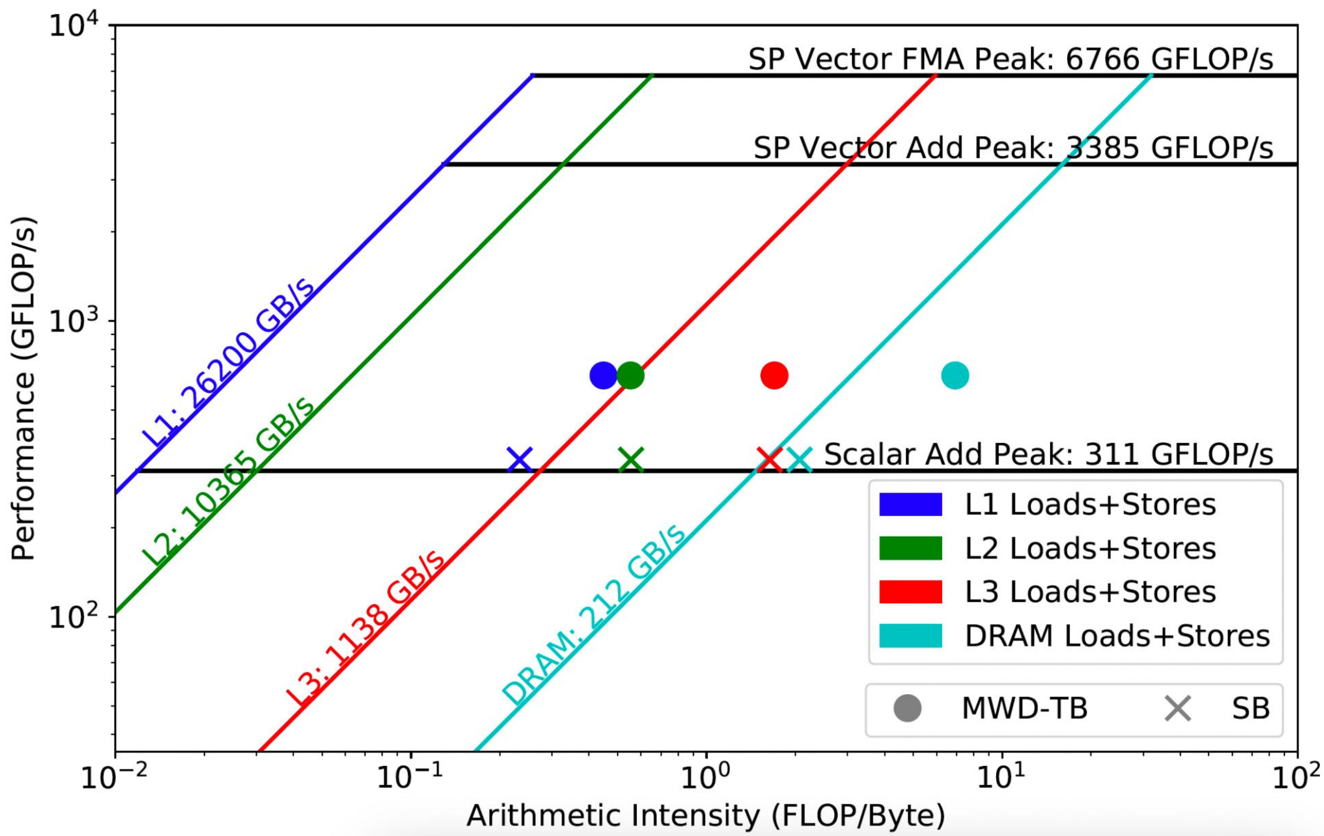
- Theoretical peak performance
  - Number of floating-point operations per seconds (*X*flops/s)
- Theoretical peak bandwidth
  - Number of byte transferred from main memory per seconds (*X*bytes/s)
- Vendor-defined from hardware specifications
- Many tools available to determine the roofline of the underlying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

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 && ./cuda_gemm

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 && ./cuda_gemm
```

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 && ./cuda_gemm
```