Introduction aux GPUs, à la programmation CUDA

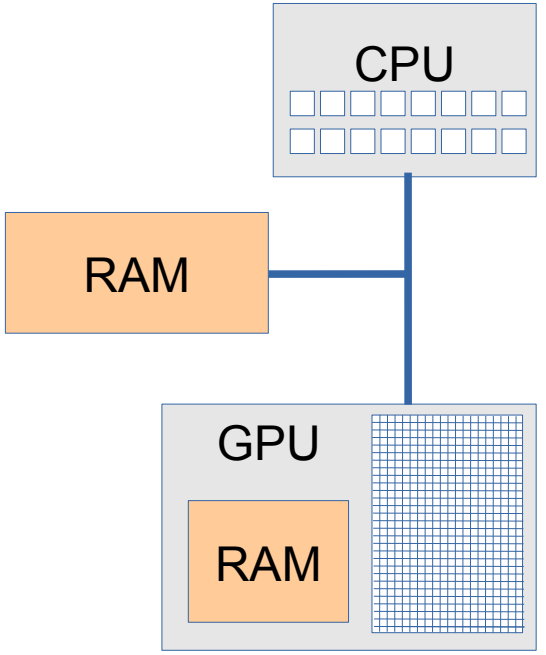
12 juin 2024

Jussieu, Paris

Samuel Thibault

GPUs

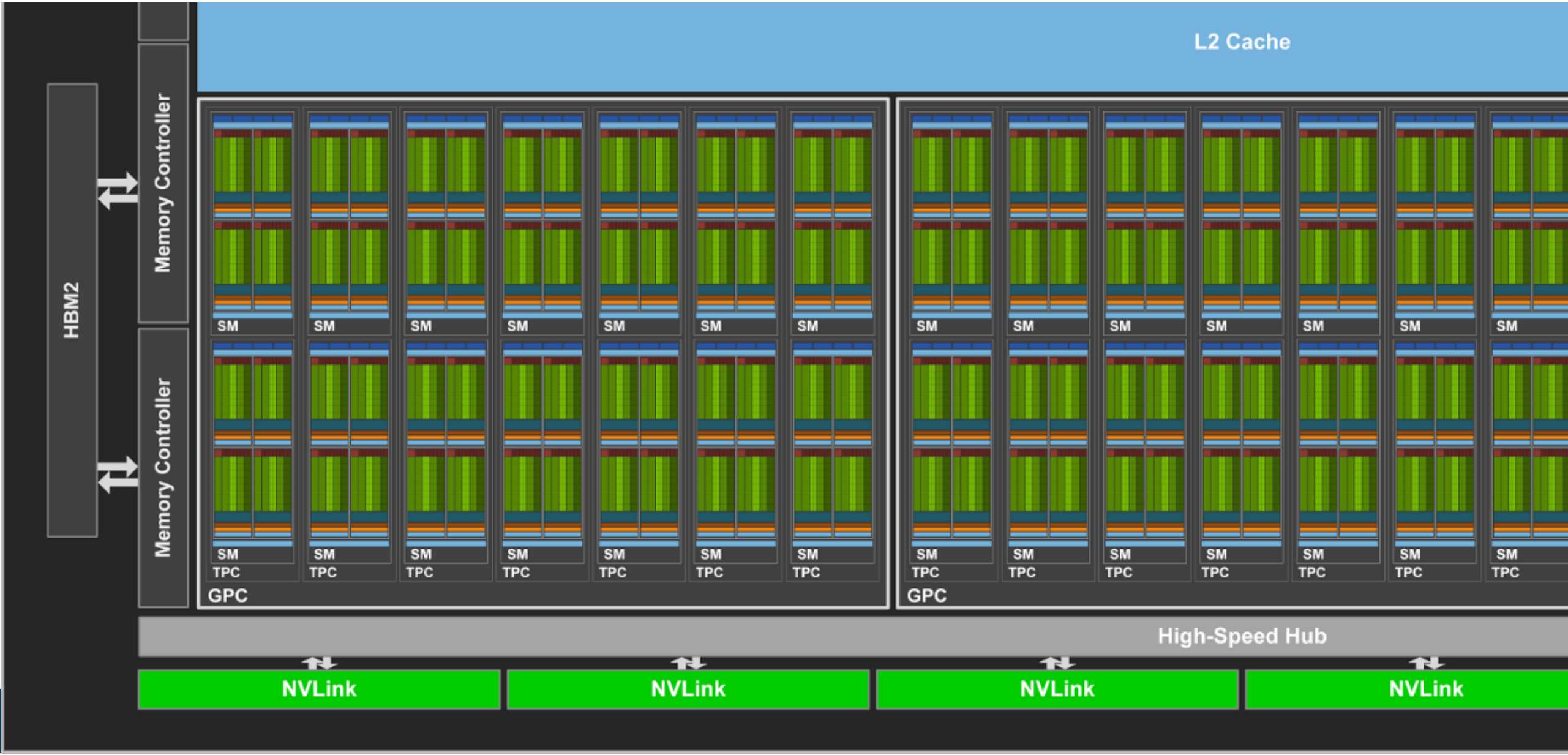
- Programmation particulière
  - Parallélisme massif !
  - Mémoire séparée (la plupart du temps)
- Développement particulier
  - Compilation différente
  - Lancement par offload logiciel
    - On ne peut pas juste recompiler
  - Outils de débuggage différents



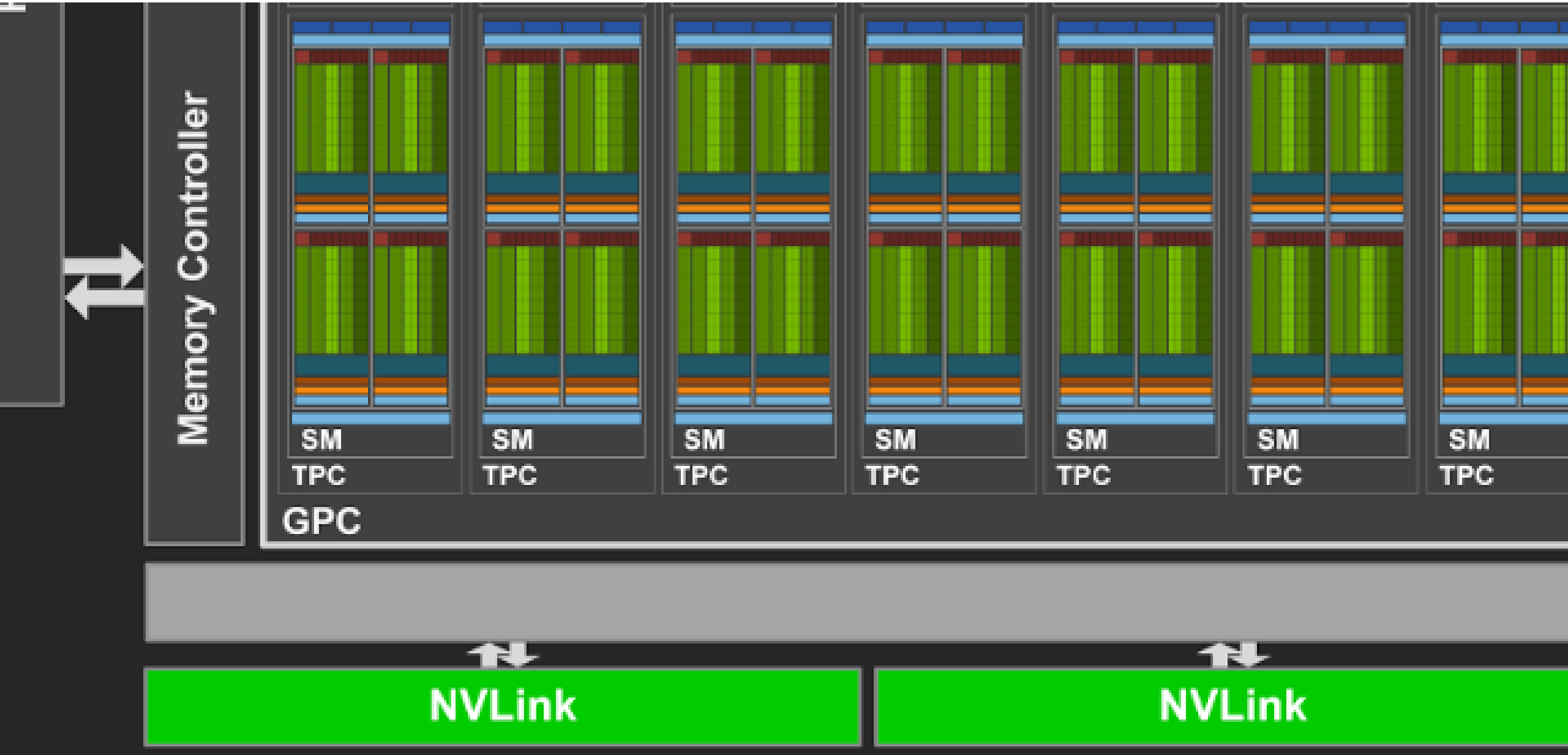
GPUs


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

GPUs


L2 Cache
HBM2
Memory Controller
Memory Controller
SM
TPC
GPC
High-Speed Hub
NVLink
NVLink
NVLink
NVLink

GPUs



GPUs


L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU

GPUs

Massive parallelism !

Example: V100

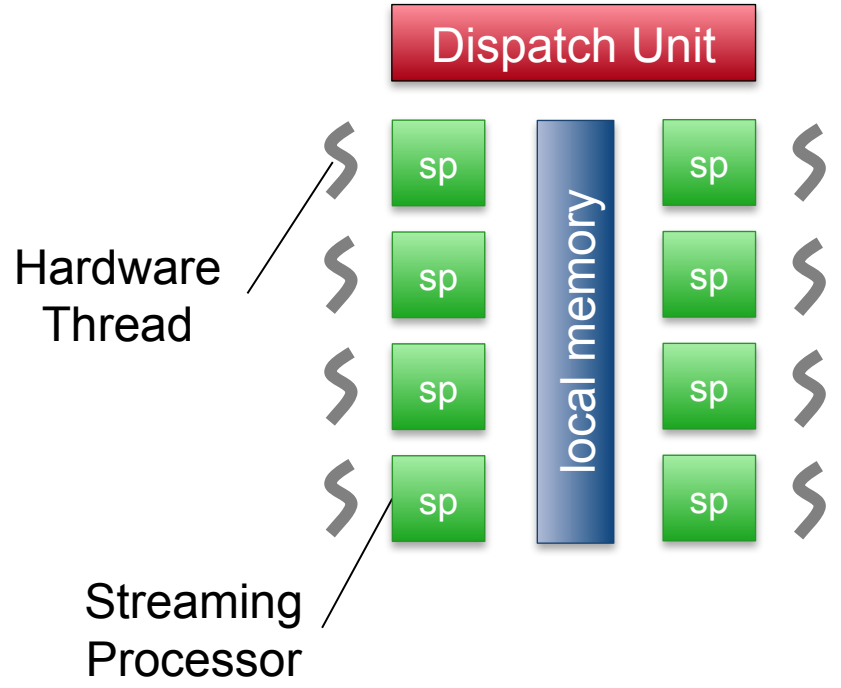
- 5,120 CUDA cores, structured in SM, TPC, GPC, …
  - Parallelism to be structured
  - And massive to overlap memory access
- SP: ~16TFlop/s
- DP: ~8TFlop/s
- HBM Memory: ~900GB/s
- (and not counting the tensor cores)

GPU execution model

Basic block: Streaming Multiprocessor (SM)

= Cluster of streaming processors (~core)

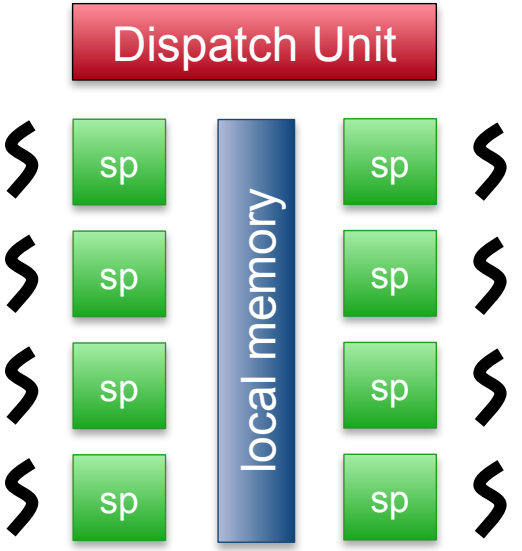
- Local memory sharing
- Synchronization
- 64KB of registers
- hardware threads
  - Creation/destruction is free !



GPU execution model

Only one instruction dispatch unit per SM

- All cores execute the same instruction at the same clock cycle
- On different data ⇒ SIMD
- Takes several cycles to fetch&decode



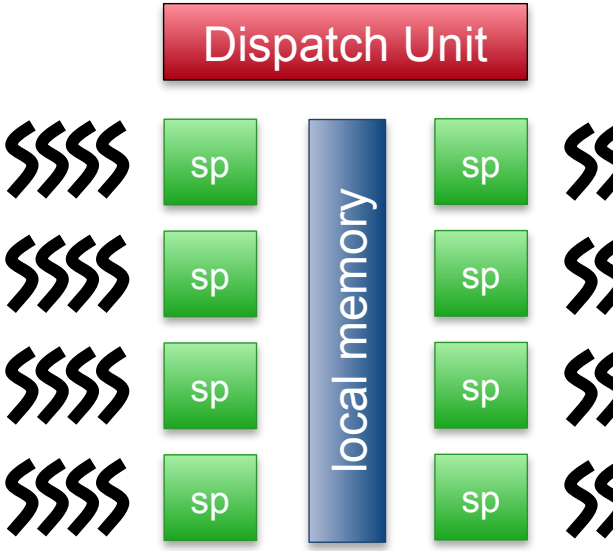
GPU execution model

Only one instruction dispatch unit per SM

- All cores execute the same instruction at the same clock cycle
- On different data ⇒ SIMD
- Takes several cycles to fetch&decode

⇒ overlap by running yet more threads running the same instruction

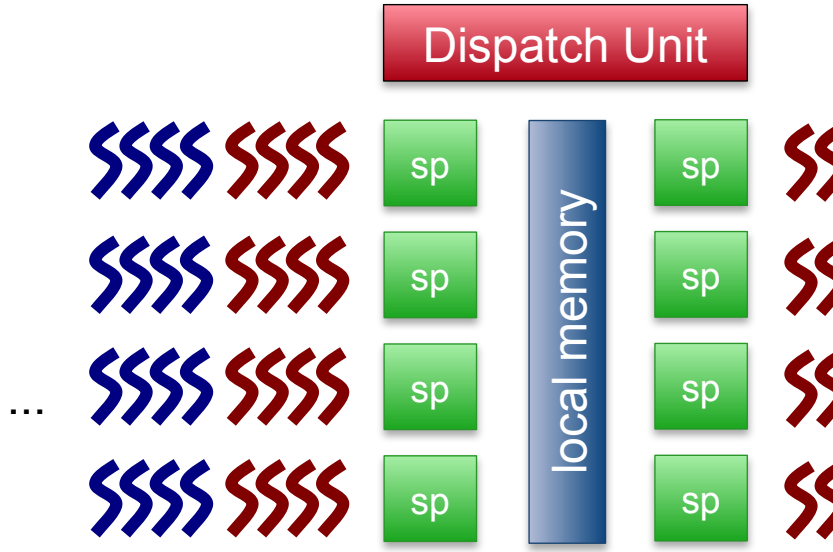
  - Warp = 32 threads



GPU execution model

Only one instruction dispatch unit per SM

- Loading data from global memory takes cycles
- ⇒ overlap by running yet more threads
  - 128 are enough to hide memory latency



GPU execution model

Threads, threads, threads…

- But they are trivial to create
- Example: vector scaling

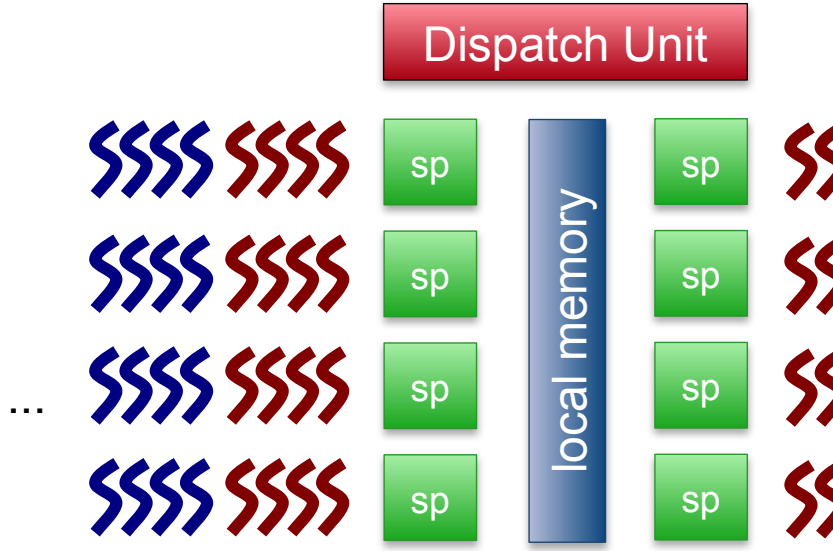
GPU function

CPU function

GPU pointer !

```
__global__ void
vector_mult_cuda(unsigned n, float *val, float factor)
{
        val[threadIdx.x] *= factor;
}

void
vector_mult_func(unsigned n, float *val, float factor)
{
        vector_mult_cuda<<<1,n,0>>>(n, val, factor);
}
```



GPU execution model

But hierarchy of SM

- Thread scheduling needs structure : blocks

```
static __global__ void vector_mult_cuda(unsigned n, float *val, float factor)
{
        unsigned i =  blockIdx.x*blockDim.x + threadIdx.x;

        if (i < n) val[i] *= factor;
}

extern "C" voi                                signed n, float *val, float factor)
{
        unsigned threads_per_block = 64;
        unsigned nblocks = (n + threads_per_block-1) / threads_per_block;

        vector_mult_cuda<<<nblocks,threads_per_block,0>>>(n, val, *factor);
}
```

Two-dimension indexing (can be three)

Compensate round-up

GPU execution model

Threads, threads, threads…

- Trivial to create
  - Massive parallelism
  - Forget about for loops
- Thread scheduling needs structure : blocks
  - How large? Depends on
    - the GPU generation,
    - available parallelism, …

GPU execution model

Thread divergence

```
static __global__ void
vector_mult_cuda(unsigned n, float *val, float factor)
{
        unsigned i =  blockIdx.x*blockDim.x + threadIdx.x;

        if (i < n && i % 2 == 0) val[i] *= factor;
}
```

Only half the threads actually work

⇒ 2x parallelism loss

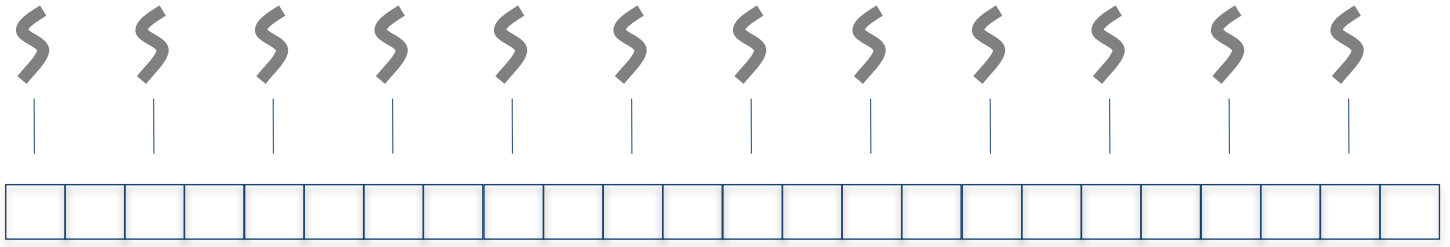
Related to warps ; depends on GPU generation

GPU execution model

Memory coalescing

```
static __global__ void vector_mult_cuda(unsigned n, float *val, float factor)
{
        unsigned i =  blockIdx.x*blockDim.x + threadIdx.x;

        val[i*2  ] *= factor;
        val[i*2+1] *= factor;
}
```



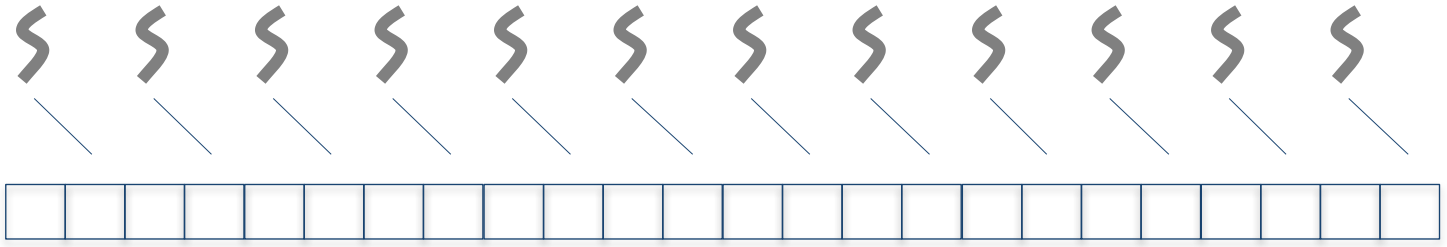
Performance is weak : accesses are non-coalesced

GPU execution model

Memory coalescing

```
static __global__ void vector_mult_cuda(unsigned n, float *val, float factor)
{
        unsigned i =  blockIdx.x*blockDim.x + threadIdx.x;

        val[i*2  ] *= factor;
        val[i*2+1] *= factor;
}
```

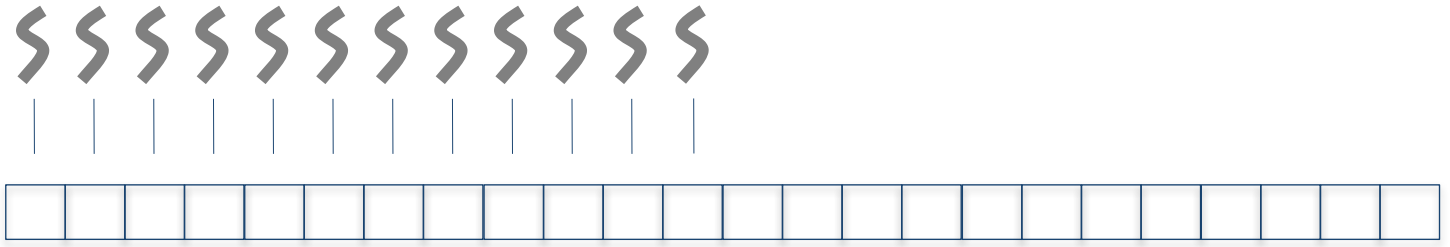


Performance is weak : accesses are non-coalesced

GPU execution model

Memory coalescing

```
static __global__ void vector_mult_cuda(unsigned n, float *val, float factor)
{
        unsigned i =  blockIdx.x*blockDim.x + threadIdx.x;
        unsigned nthreads = blockDim.x * blockDim.y;
        val[           i] *= factor;
        val[nthreads + i] *= factor;
}
```

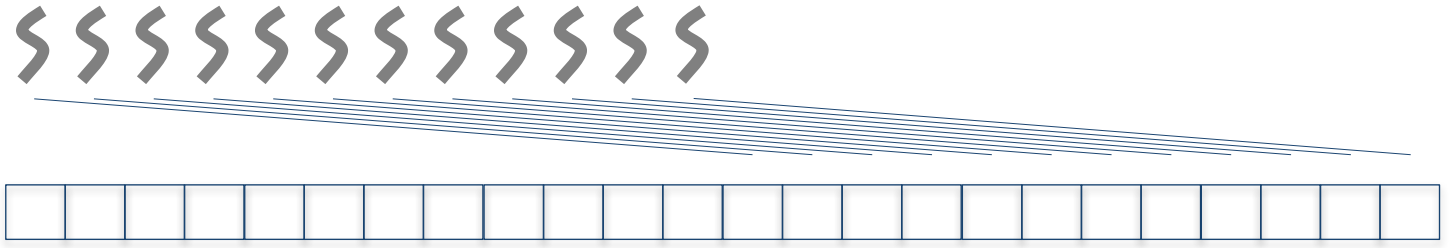


Accesses are coalesced

GPU execution model

Memory coalescing

```
static __global__ void vector_mult_cuda(unsigned n, float *val, float factor)
{
        unsigned i =  blockIdx.x*blockDim.x + threadIdx.x;
        unsigned nthreads = blockDim.x * blockDim.y;
        val[           i] *= factor;
        val[nthreads + i] *= factor;
}
```



Accesses are coalesced

GPU execution model

Memory coalescing

- Depends on the GPU generation
- Better use structures of arrays
  - E.g. all threads get the temperature, then the pression, etc.
- For the global memory
- Not for the local shared memory
  - Has to be allocated explicitly

GPU execution model

Summary : requires

- Massive parallelism
- Taking care of divergence
- Taking care of memory coalescing
  - Or taking care of using the local shared memory
- And other details I don’t have time to explain :)

If you can avoid writing kernels it’s best…

But we need to get data onto the GPU...

Data management

- Allocate memory on the GPU
- Copy between GPU and CPU
  - For copy to be efficient, use cudaMallocHost for CPU memory
    - Pinned memory for DMA

```
void scal_func(unsigned n, float *val_cpu, float factor)
{
        float *val_gpu;
        cudaMalloc(&val_gpu, n * sizeof(float));
        cudaMemcpy(val_gpu, val_cpu, n * sizeof(float), cudaMemcpyHostToDevice);
        scal_cuda_func(n, val_gpu, factor);
        cudaMemcpy(val_cpu, val_gpu, n * sizeof(float), cudaMemcpyDeviceToHost);
}
```

Data management

But transfer takes time

- Overlap cudaMemcpy with another kernel execution
- Async variants of cuda operations, and poll or wait
- Use Streams for coordination

```
void scal_func(unsigned n, float *val_cpu, float factor)
{
        float *val_gpu;
        cudaMalloc(&val_gpu, n * sizeof(float));
        cudaMemcpy(val_gpu, val_cpu, n * sizeof(float), cudaMemcpyHostToDevice);
        scal_cuda_func(n, val_gpu, factor);
        cudaMemcpy(val_cpu, val_gpu, n * sizeof(float), cudaMemcpyDeviceToHost);
}
```

Data management

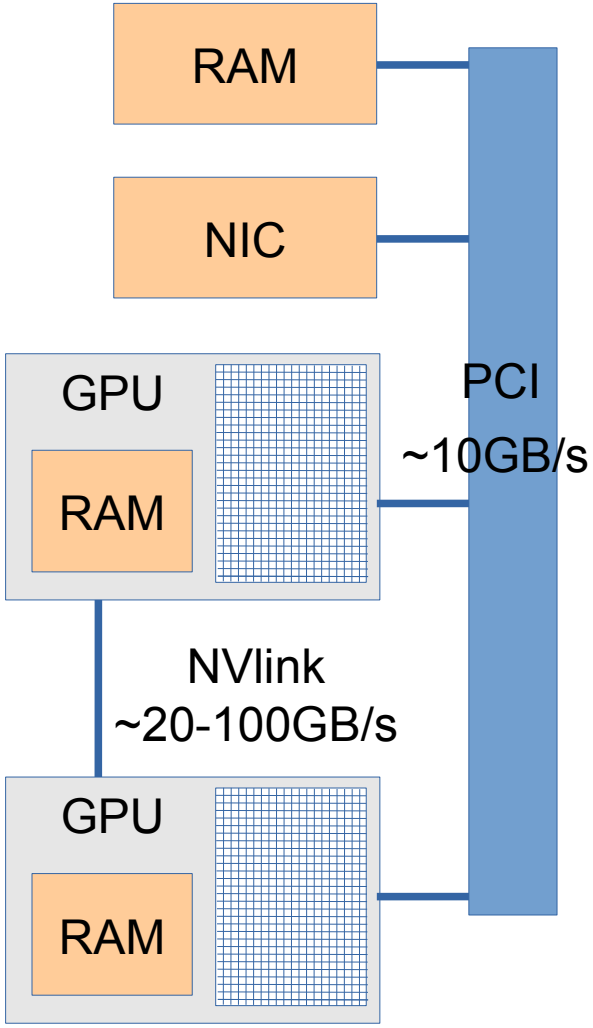
Or use cudaGraphs

- Express dependencies explicitly between cudaMemcpyAsync and kernel calls
- Let the GPU process the task graph

Data management

Topology matters

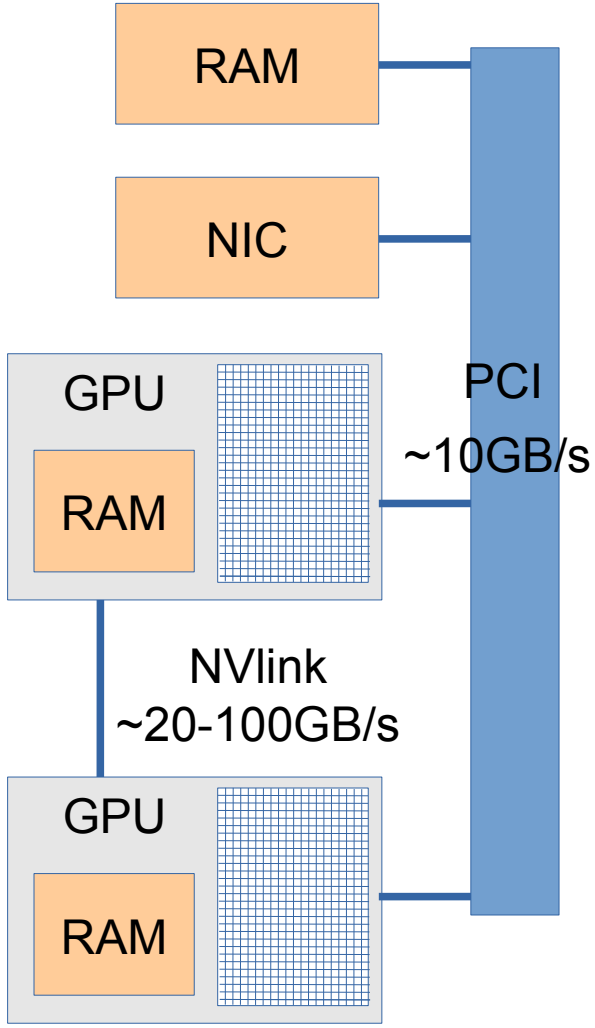
- NVlink provides fast GPU-GPU transfers
- GPU-Direct provides fast NIC-GPU transfers
  - With CUDA-aware MPI implementation
- Beware of PCI topology
- Beware of NUMA topology



Data management

Or use unified memory

- Pass managed pointers to kernels
- CUDA runtime handles transfers on the fly
- But CUDA runtime is mostly blind
  - On-demand migration, introduces latency
  - With cudaMemAdvise / cudaMemPrefetch hints, can be efficient



Data management

Summary

- HBM memory embedded in GPU
- Requires transfers
- To be pipelined with computation kernels

If you can avoid managing this it’s best…

Conclusion

GPUs are beasts

- Require massive parallelism
  - With massive data accesses
- Optimizing is very involved
- Better delegate it to another software layer