GP-GPU and High Performances Computing

Lecture 3
Programming on GP-GPU (1)

Example

Vector addition on CPU

```
#include <iostream>
int main(void) {
  int N = 1<<20; // 1M elements
  float *x = new float[N];
  float *y = new float[N];
  // CPU: initialize x and y
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f; y[i] = 2.0f;
  }
  // CPU: run on 1M elements
  add(N, x, y);
  // Free memory
  delete [] x; delete [] y;
  return 0;
}
```

```
// CPU: adding two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}
```

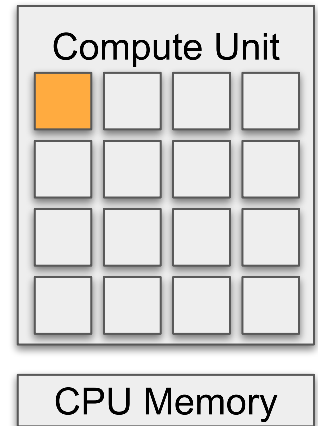
Declaration of array

Initialization of data

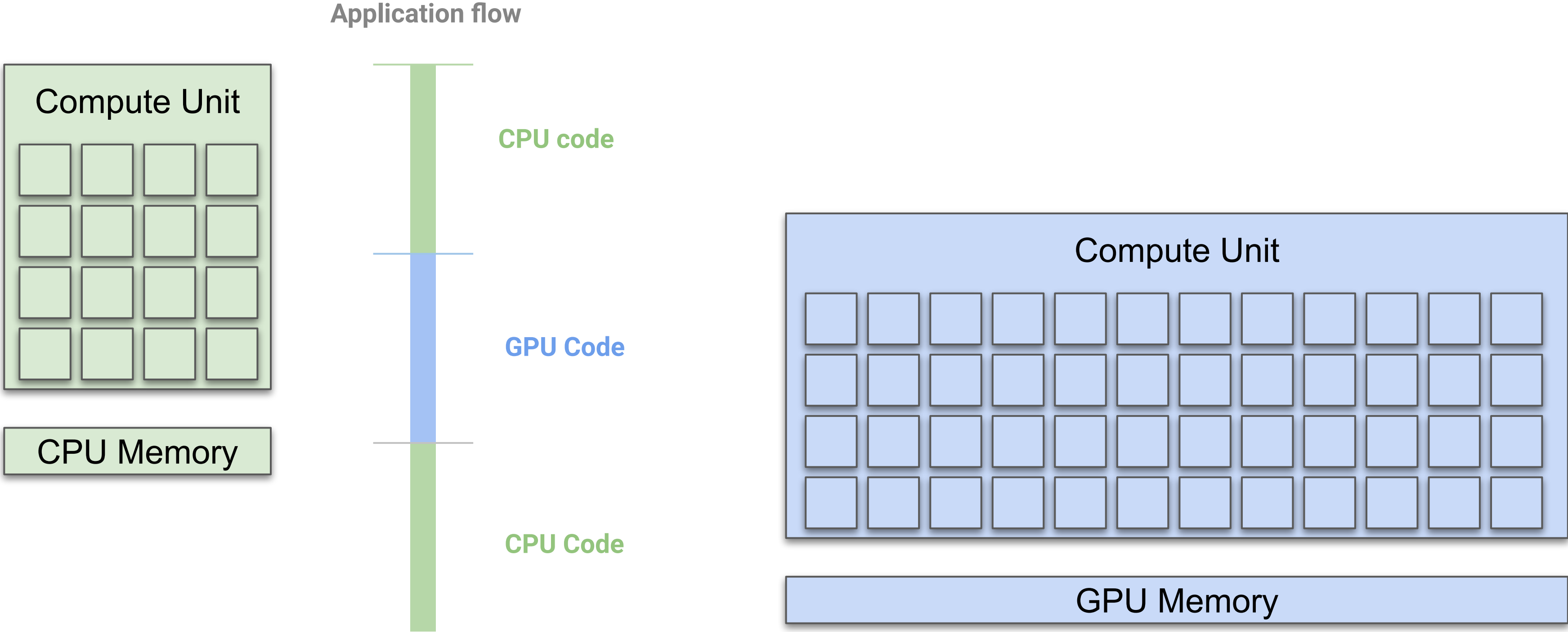
Computation

Cleanup

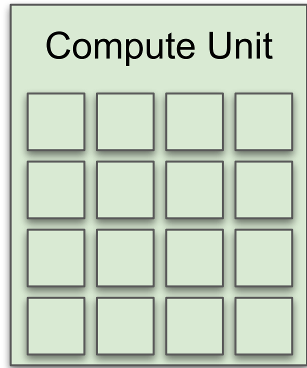
On CPU, this function will be run on one core by default.

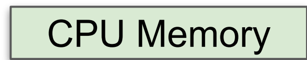


Executing flow with GP-GPU

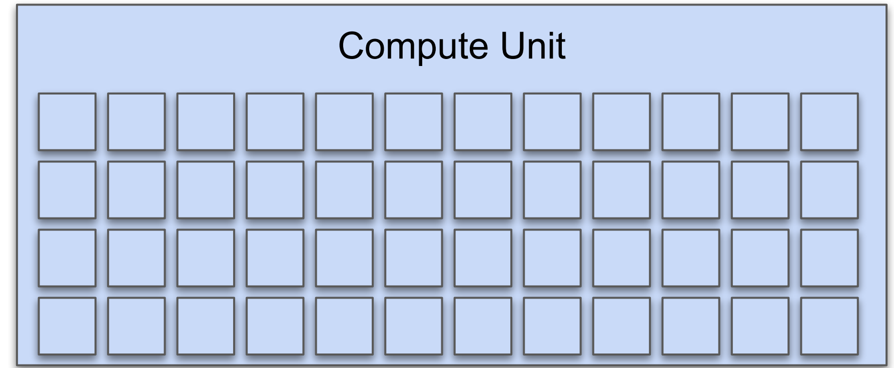


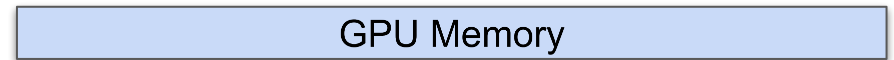
Executing code with GPU





1. Allocate memory in GPU that is "visible" to CPU
2. Load GPU program and execute
3. Wait to complete before looking at results from CPU





Memory transfer

Vector addition on GPU

```
int N = 1<<20; // 1M elements
float *x,*y;
// Allocate in GPU memory visible to GPU
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
// CPU: initialize x and y
for (int i = 0; i < N; i++) {
  x[i] = 1.0f; y[i] = 2.0f;
}
// CPU: launch kernel on GPU
add<<<1,1>>>(N, x, y);
// Wait for GPU to finish
cudaDeviceSynchronize();
// Free GPU memory
cudaFree(x); cudaFree(y);
```

Declaration of array

Initialization of data

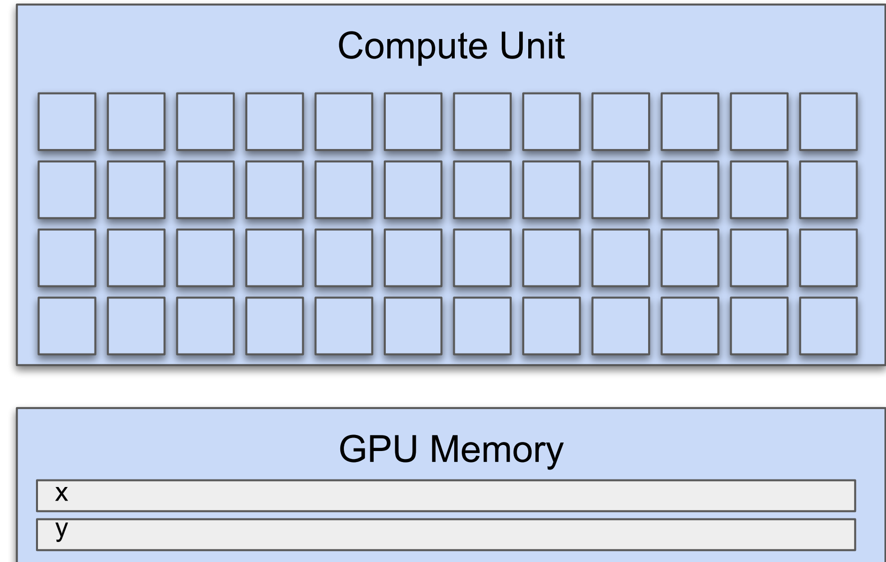
Computation

Cleanup

Vector addition with GPU

```
int N = 1<<20; // 1M elements
float *x,*y;
// Allocate in GPU memory visible to GPU
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
// CPU: initialize x and y
for (int i = 0; i < N; i++) {
  x[i] = 1.0f; y[i] = 2.0f;
}
// CPU: launch kernel on GPU
add<<<1,1>>>(N, x, y);
// Wait for GPU to finish
cudaDeviceSynchronize();
// Free GPU memory
cudaFree(x); cudaFree(y);
```

Declaration of array



Vector addition on GPU

```
// GPU: adding elements of two arrays
__global__ void add(int n, float *x, float *y) {
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}
```

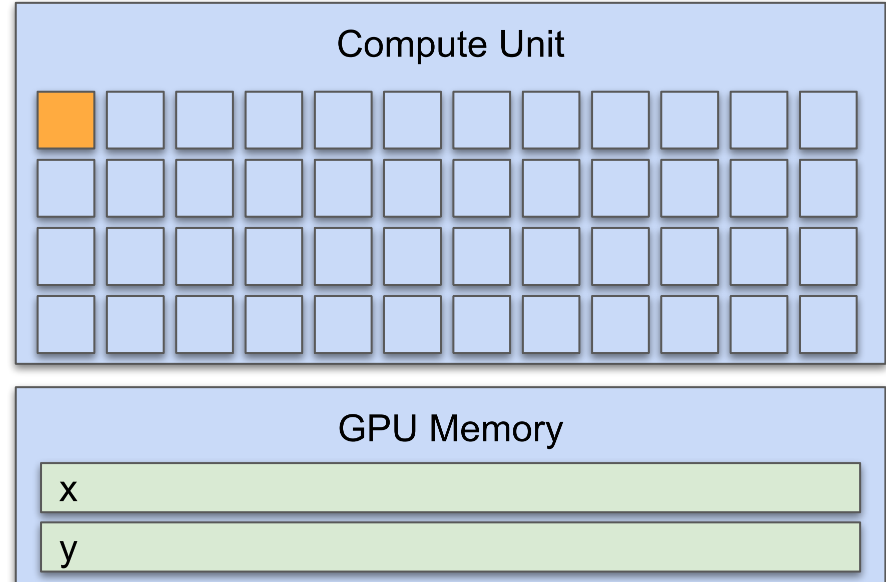
➔ Signature of function is similar
➔ Content of function is identical

➔ `__global__` defines a function that can be called from the CPU or the GP-GPU and is executed on GP-GPU

How to launch code from the CPU ?

Vector addition with GPU

```
int N = 1<<20; // 1M elements
float *x,*y;
// Allocate in GPU memory visible to GPU
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
// CPU: initialize x and y
for (int i = 0; i < N; i++) {
  x[i] = 1.0f; y[i] = 2.0f;
}
// CPU: launch kernel on GPU
add<<<1,1>>>(N, x, y);
// Wait for GPU to finish
cudaDeviceSynchronize();
// Free GPU memory
cudaFree(x); cudaFree(y);
```



`<<<1,1>>>` means only
➔ 1 compute unit is used.
➔ 1 thread is spawned.

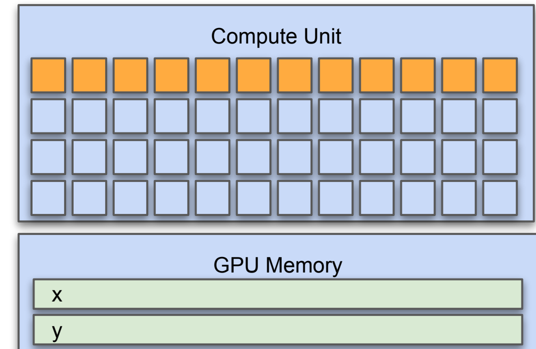
Multiple threads

```
// GPU: adding elements of two arrays
add<<<1,256>>>(N, x, y);
```

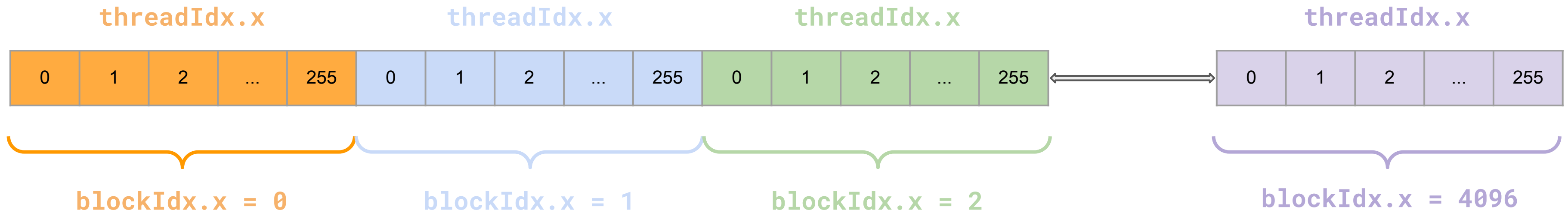
```
// GPU: adding elements of two arrays
__global__ void add(int n, float *x, float *y) {
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i+=stride)
    y[i] = x[i] + y[i];
}
```

256 threads are used simultaneously

- ➔ threadIdx.x is the index of the thread in a block in the x dimension
- ➔ blockDim.x is the side of a block in dimension x



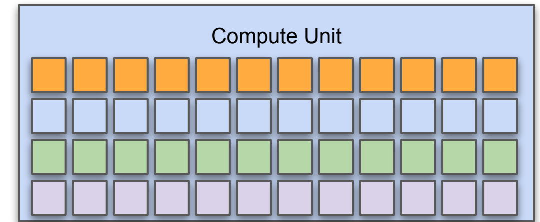
Adding more blocks



To compute the index of a thread in the index set :

```
idx = threadId.x + blockIdx.x * blockDim.x
```

The global space of index is a called a grid.

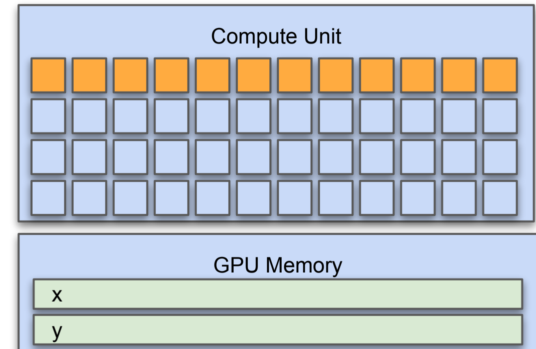


Link with the GP-GPU architecture

- ➔ 1 thread exists within a warp. A warp cannot execute one thread only. All 32 threads in a warp are simultaneously executed.
- ➔ A block is a virtual grouping of threads.
- ➔ We may customize the number of threads and the number of blocks.

```
// Customize execution size
int blockSize = 256;
int numBlocks = 1;
add<<<numBlocks, blockSize>>>(N, x, y);
```

```
// GPU: adding elements of two arrays
__global__ void add(int n, float *x, float *y) {
  int index = threadIdx.x + blockIdx * blockDim.x;
  y[i] = x[i] + y[i];
}
```



Vector addition with GPU

```
float *x = new float[N];
float *y = new float[N];
int size = N*sizeof(float);
float *d_x, *d_y; // device copies of x y
cudaMalloc((void **)&d_x, size);
cudaMalloc((void **)&d_y, size);
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
// Execute kernel on GPU
add<<<4096,256>>>(d_x, d_y);
// Copy result back to host
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
// Free memory
cudaFree(d_x); cudaFree(d_y);
delete [] x; delete [] y;
```

Declaration of array

Copy data to GPU

Computation

Copy back data

Cleanup

Executing on a GP-GPU

Some definitions

Host : hardware component that will execute the main part of the application. Traditionally, it is the CPU or the core computing elements of the CPU. It will drive the computation and manage access to the different ressources.

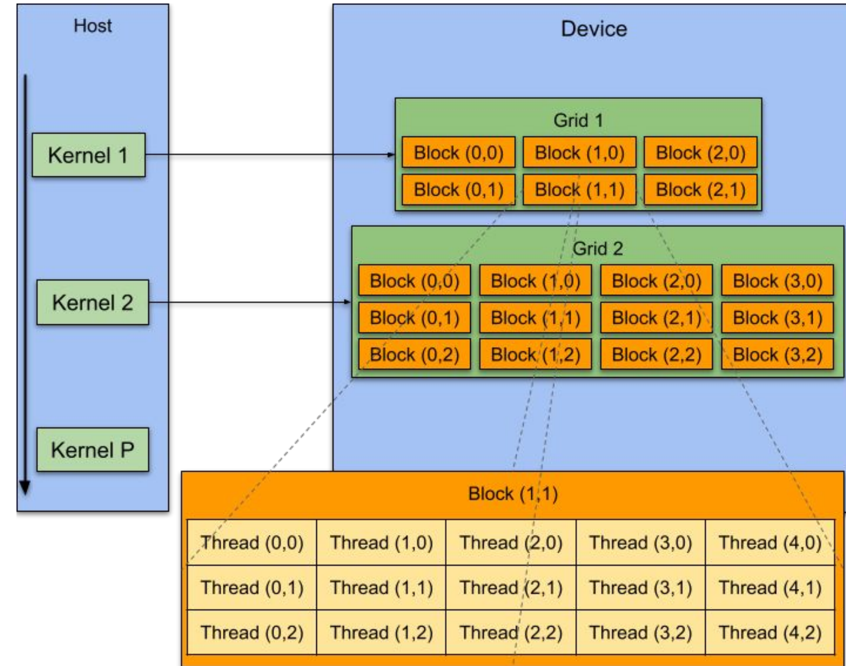
Device : hardware component that will execute the data intensive part of the application. Traditionally, it is a GP-GPU.

Kernel : set of functions that will be executed on a device. Kernel code is written in subset of C/C++. The kernel is broadcasted by the host on the different streaming multiprocessors.

Execution of thread, block, grids

The CPU program requests the execution of a grid of blocks of threads:

- ➔ assume all threads are identical.
- ➔ threads organized in blocks, each block running on a single Stream Multiprocessor (SM)
- ➔ blocks organized within a grid, which distributes its blocks on all Stream Multiprocessors
- ➔ we can define 1D, 2D and 3D grids and blocks:
  - 1D grid of 2D blocks
  - 2D grid of 1D blocks
  - 2D grid of 2D blocks
  - 3D grid of 1D blocks
  - ...



Streaming Multiprocessor

- Thousands of registers that can be partitioned among threads of execution
- Several caches:
  - Shared memory for fast data interchange between threads
  - Constant cache for fast broadcast of reads from constant memory
  - Texture cache to aggregate bandwidth from texture memory
  - L1 cache to reduce latency to local or global memory
- Warp schedulers that can quickly switch contexts between threads and issue instructions to warps that are ready to execute
- Functional units to LD/ST in memories
- Execution cores for integer and floating-point operations:
  - Integer and single-precision floating point operations
  - Double-precision floating point
  - Special Function Units (SFUs) for single-precision floating-point transcendental functions

Example - On Ampere architecture

- A classical SM is partitioned into four processing partitions each with
  - a 64 KB register file
  - an L0 instruction cache
  - one warp scheduler
  - one dispatch unit
  - sets of math and other units

- The four partitions share a combined 128 KB L1 data cache/shared memory subsystem.

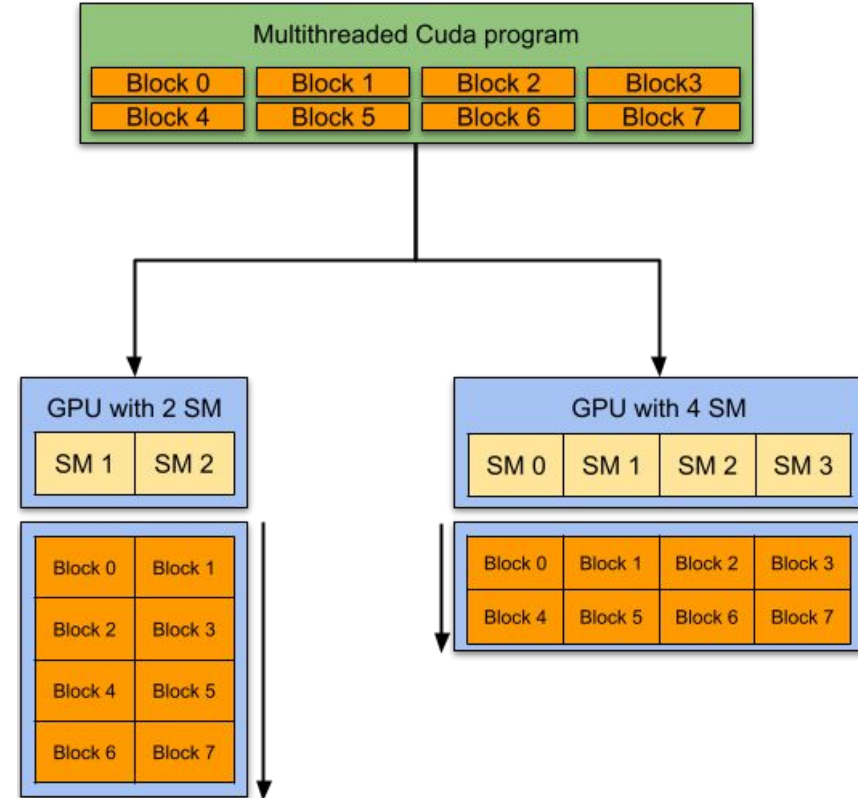
- A SM contains
  - 128 cores
  - 4 textures units
  - 1 tensor core

- There are 84 SM on A6000

Execution of threads on SM

The CPU program requests the execution of a grid of blocks of CUDA threads:

- ➔ the block scheduler distributes the blocks on different Streams Multiprocessors (SMs).
- ➔ different architectures will execute the same grid of blocks at different rate and with a different distribution.
- ➔ there is no indication on the order of execution of the blocks.



Some more definition

Core: functional unit that support the single precision floating point add, multiply, and multiply-add instructions. It does not perform other instructions (LD/ST,...). A core is also called an execute unit, a streaming processor, execution resources, compute unite.

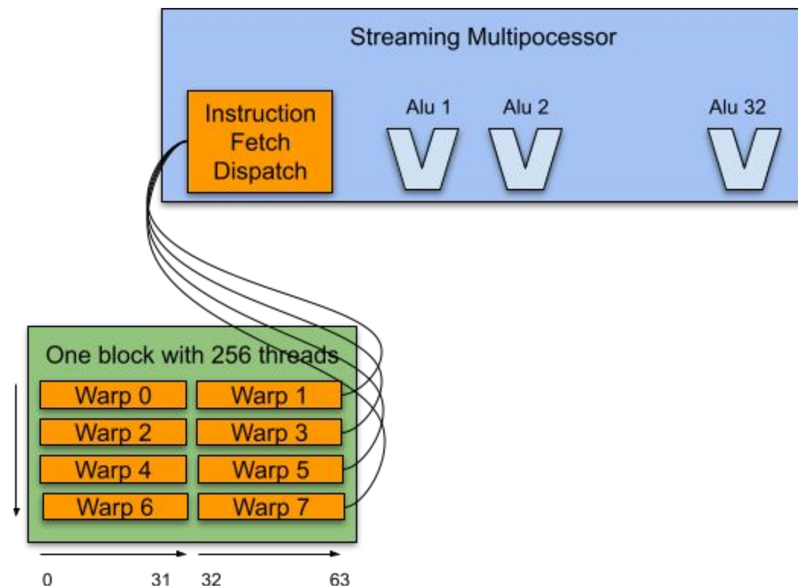
Thread: On GP-GPU, a thread is the smaller subdivision of task to be performed. It may contains floating point instructions, loading, storing.

Warp: a warp is a set of 32 consecutive threads that follow the same execution path on a SM. A warp cannot be shared among SM.

Block: a group of thread. A block is associated with a single SM. It cannot migrate from an other SM. A SM can run several blocks concurrently.

Thread and Block scheduling

- ➔ A Cuda thread is associated to a core and its environment.
- ➔ The thread scheduler organize warps execution.
- ➔ For GP-GPU, a group of threads in a block will execute the same set of instructions synchronously.
- ➔ We call this SIMT (Single Instruction, Multiple Threads). This grouping may go beyond a single groupe into the grid.
- ➔ SIMT is enforced by software, i.e. by the size of the group.
- ➔ A thread in a warp execute the same set of instructions than all the other threads.
- ➔ Each threads may execute different instructions but it will lost cycles.



Execution of a block

- An instruction fetch and dispatch unit drives 32 "hardware threads" (32 ALUs)
- The thread scheduler enables warps of 32 thread
- if possible, warps are made of threads consecutive in x (privileged) dimension
  - Create blocks of at least 32 threads otherwise part of the ALU will always be unused
  - Create blocks with a multiple thread count of 32 otherwise the last warp will be incomplete and ALUs will be unused
  - Create blocks with an x-dimension multiple of 32 otherwise the "coalescence" will be defective
  - It may still works very well with a block x-dimension multiple of half warp or quarter warp...!

Block thread optimisation (example with Ampere)

There are some constraints on the different sizes

- ➔ Max threads per SM : 2048
- ➔ Max threads per block : 1024
- ➔ Max warps per SM : 64

If 2 blocks are assigned to an SM and each block has 1024 threads, how many warps are there in an SM?

- ➔ Each block is divided into 1024/32 = 32 warps
- ➔ There are 32 * 2 = 64 Warps

At any point in time, only 4 of the 64 warps will be selected for instruction fetch and execution.

- ➔ One instruction is issued for 1 warp at every cycle (by design).
- ➔ 16 cycles are needed to execute 1 instruction on all threads of the block.

SM will interleaved warps to optimize execution.

Cuda thread block

- Thread block = virtualized multiprocessor
  - freely choose processors to fit data
  - freely customize for each kernel launch

- Thread block = a (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want

- Thread blocks of kernel must be independent tasks
  - program valid for any interleaving of block executions

Blocks

- ➔ Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially

- ➔ Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock

- ➔ Independence requirement gives scalability

Synchronization

➔ Threads within a block may synchronize with barriers

… Step 1 …

```
__syncthreads();
```

… Step 2 …

➔ Blocks coordinate via atomic memory operations, e.g., increment shared queue pointer with atomicInc()
  - This is a coordination not a synchronization!

➔ Implicit barrier between dependent kernels

```
add<<<nblocks, blksize>>>(a, b);
add<<<nblocks, blksize>>>(b, c);
```

Grid

A grid can be seen from two point of view

- How to organize the blocks, threads and instructions to cover the workflow of the application?
- How to design a grid that will cover the data (Input, Output or both)?

A grid has limitations. On Ampere

- Maximum x-dimension of a grid of thread blocks : 2^31-1
- Maximum y-, or z-dimension of a grid of thread blocks : 65535 = 2^16
- Maximum number of threads per block : 1024
- Maximum number of threads per SM : 2048
- The maximum number of threads that can be launched simultaneously is 10752

How to apply a filter on this image (1170x540) ?

Execution flow

Host execution flow

Steps for executing an application

- ➔ CPU: launch a program (main function) in Python, C, C++
- ➔ CPU: initialize variables, light calculations, sequential computations
- ➔ CPU: compile code that will be executed on GP-GPU
- ➔ CPU: data transfer from CPU memory to GP-GPU memory
- ➔ CPU: launch of remote calculations on the GP-GPU
  - ◆ CPU: transfers code
  - ◆ GPU: execute massively parallel “kernels“
- ➔ CPU: transfer results from GP-GPU memory to CPU memory

CPU/GP-GPU transfers are time consuming !

Conclusion

Parallelism

- Thread parallelism
  - each thread is an independent thread of execution

- Data parallelism
  - across threads in a block
  - across blocks in a kernel

- A grid of blocks deploys warps of threads of a CUDA kernel which access data structures on the GPU.

Conclusions

Hierarchy of a GPU

➔ A GP-GPU is a highly structured set of compute unit, with several level of organization than can be optimized by the application specialist.

Themes of this class

➔ Organization of a GP-GPU with respect to data
➔ Terminology of GP-GPU