Computing with GPU
Part 1

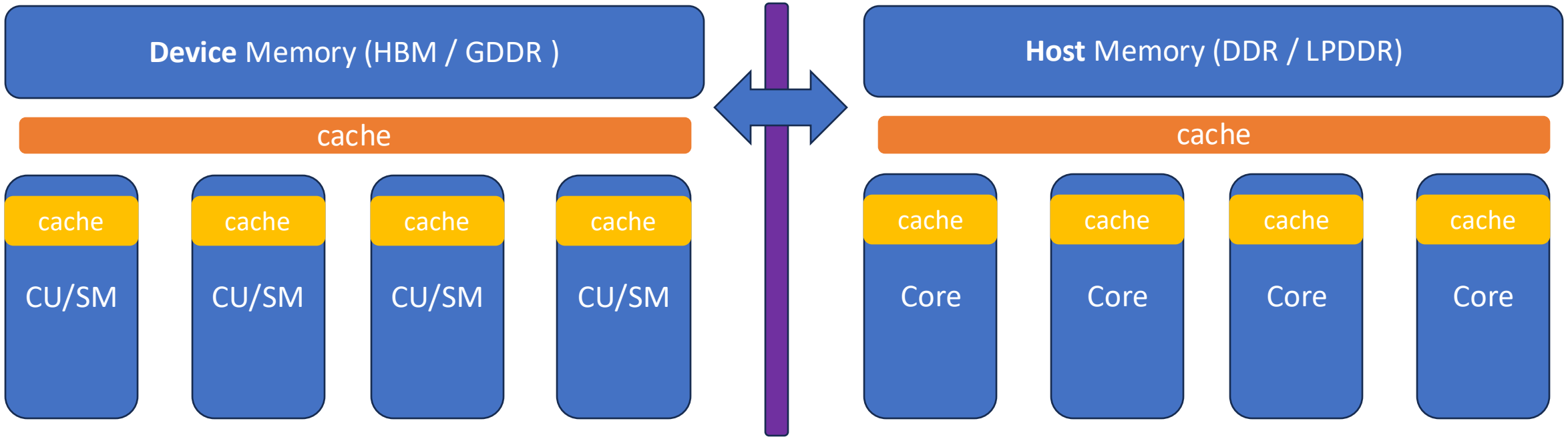
General Processing Unit

We will skip the history and discussion on how GPUs evolved from being Graphic cards to becoming Processing Units for general applications.

For us … GPU is a compute device. GPU must be used together with a CPU as GPU can not run an Operating System and perform some of the OS functionality.

GPUs are mostly seen as accelerators capable to execute some range of workloads at a speed and power efficiency much higher than CPUs.

Why GPUs provide a better performamce?



GPU's architecture demands high level of thread-parallelism !

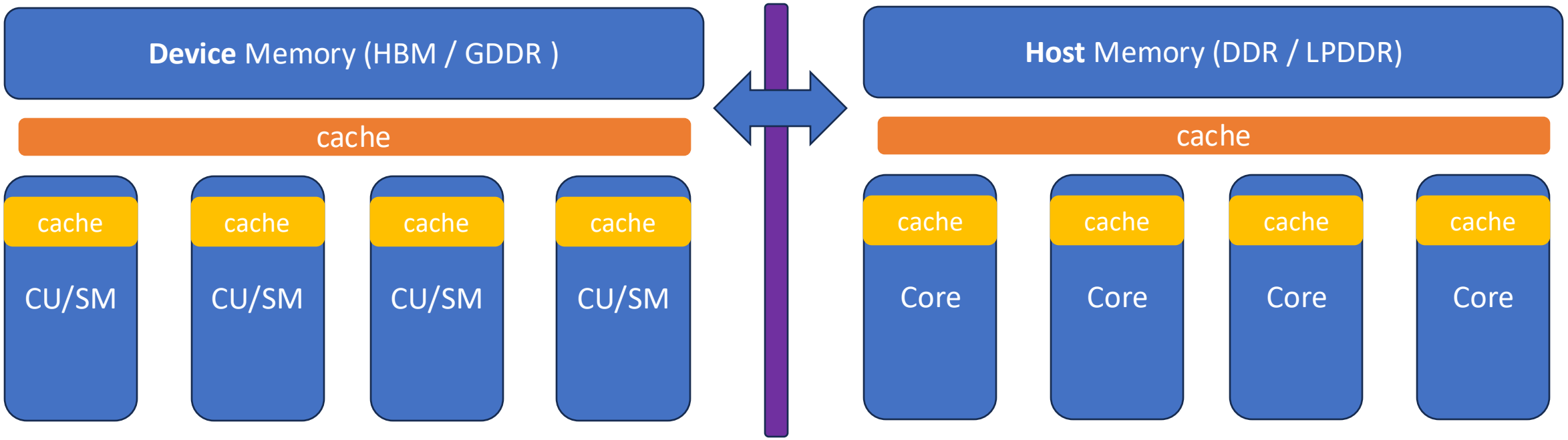
A GPU CU/SM can effectively support up to 2048 threads

A CPU core can effectively support 1-4 HW threads

GPU is a massively parallel device

GPUs support hundreds of thousands of threads and are designed for **shared memory programming model**.

CPUs support O(100) threads and are designed for shared memory and distributed memory programming models.



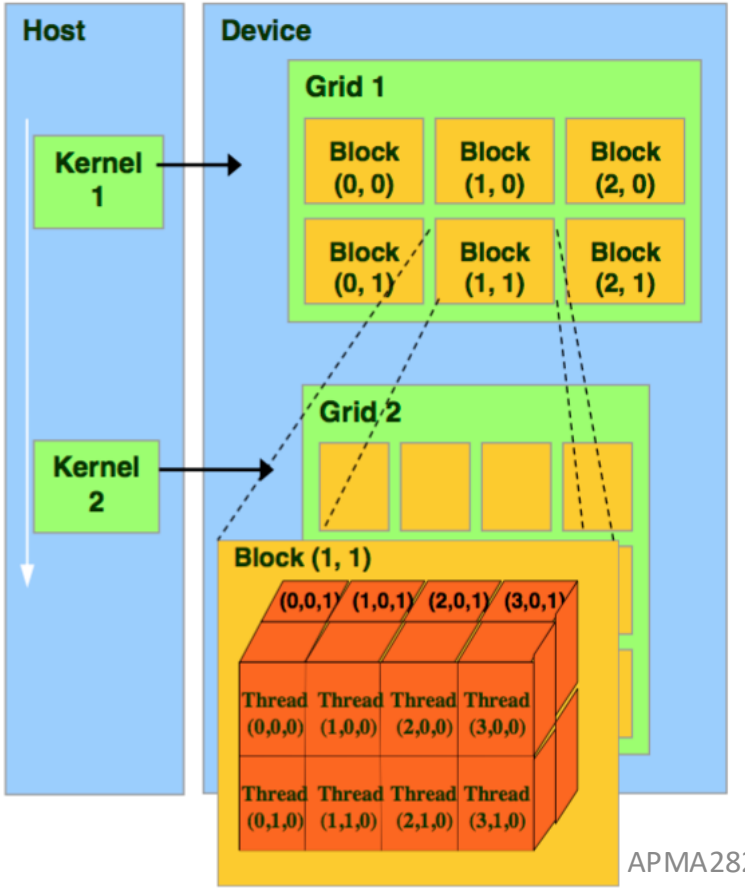
Each CU/SM has very wide vector units .

A basic set of threads executing concurrently on those vector units is called a WARP.
On AMD GPUs WARPS are either 32-lane wide or 64-lane wide, on NVIDIA GPUs warps are 32-lane wide.

The threads in a WARP execute the same instruction and can operate on different data (SIMD)

To express thread level parallelism for GPUs developers consider multiple levels of parallelism.

To express thread level parallelism for GPUs developers should consider multiple levels of parallelism.



Grid – collection of threads executing the same "kernel"
(kernel is a function executed by 1 or more threads).
Grid can run on 1 GPU.
Each GPU can support multiple Grids concurrently

Grid is subdivided to **thread-blocks**.

Maximum size of a thread-block is 1024 threads (on current AMD and NVIDIA GPUs).

Thread-blocks are numbered using a 3D indexing (coordinates):
blockIdx.x blockIdx.y blockIdx.z

Similarely, thread-blocks have dimensions in 3D
blockDim.x blockDim.y blockDim.z

Threads within a thread-block can also be indexed using 3D indexing:
threadIdx.x threadIdx.y threadIdx.z

To express thread level parallelism for GPUs developers should consider multiple levels of parallelism.

Threads in the same thread-block run on the same CU/SM

Each CU/SM can support multiple thread-blocks.

Developers do not specify affinity for thread-blocks (and threads); thread-blocks and threads are scheduled by a special scheduler.

Threads in a thread-block share all the caches (L0, L1, L2 …)

Threads from different thread-blocks only share the Last Level Cache (L2 or L3 – depending on the architecture)

Grid – collection of threads executing the same "kernel" (kernel is a function executed by 1 or more threads).
Grid can run on 1 GPU.
Each GPU can support multiple Grids concurrently

Grid is subdivided to **thread-blocks**.

Maximum size of a thread-block is 1024 threads (on current AMD and NVIDIA GPUs).

Thread-blocks are numbered using a 3D indexing (coordinates):
blockIdx.x blockIdx.y blockIdx.z

Similarely, thread-blocks have dimensions in 3D
blockDim.x blockDim.y blockDim.z

Threads within a thread-block can also be indexed using 3D indexing:
threadIdx.x threadIdx.y threadIdx.z

Threads also share per block *shared memory* (in AMD GPUs this memory is called Local Device Storage, LDS) .

Threads also have their own local storage.

Grid – collection of threads executing the same "kernel" (kernel is a function executed by 1 or more threads).
Grid can run on 1 GPU.
Each GPU can support multiple Grids concurrently

Grid is subdivided to **thread-blocks**.

Maximum size of a thread-block is 1024 threads (on current AMD and NVIDIA GPUs).

Thread-blocks are numbered using a 3D indexing (coordinates):
blockIdx.x blockIdx.y blockIdx.z

Similarely, thread-blocks have dimensions in 3D
blockDim.x blockDim.y blockDim.z

Threads within a thread-block can also be indexed using 3D indexing:
threadIdx.x threadIdx.y threadIdx.z

Decent description can be found here : https://nyu-cds.github.io/python-gpu/02-cuda/

Sample code using HIP (AMD and NVIDIA GPUs) and CUDA (NVIDIA GPUs)

```
#include <hip/hip_runtime.h>

int main() {
  …
  double *X, *Y;
  X = new double[N];
  Y = new double[N];
  for (auto i=0; i<N; i++)
    {X[i] = 1.1; Y[i] = 1.3;}

  double *x_d, *y_d;
  hipMalloc(&x_d,N*sizeoff(double));
  hipMalloc(&y_d,N*sizeoff(double));
  hipMemcpy(x_d,X,…);
  hipMemcpy(y_d,Y,…);
  gpu_func<<< … >>>(x_d, y_d, N);
  hipDeviceSynchronize();
  hipMemcpy(X,x_d,…);
  hipMemcpy(Y,y_d,…);

  cpu_func(X,Y);

  hipFree(x_d); hipFree(y_d);
  delete[] X;
  delete[] Y;
}
hipcc name.cpp
```

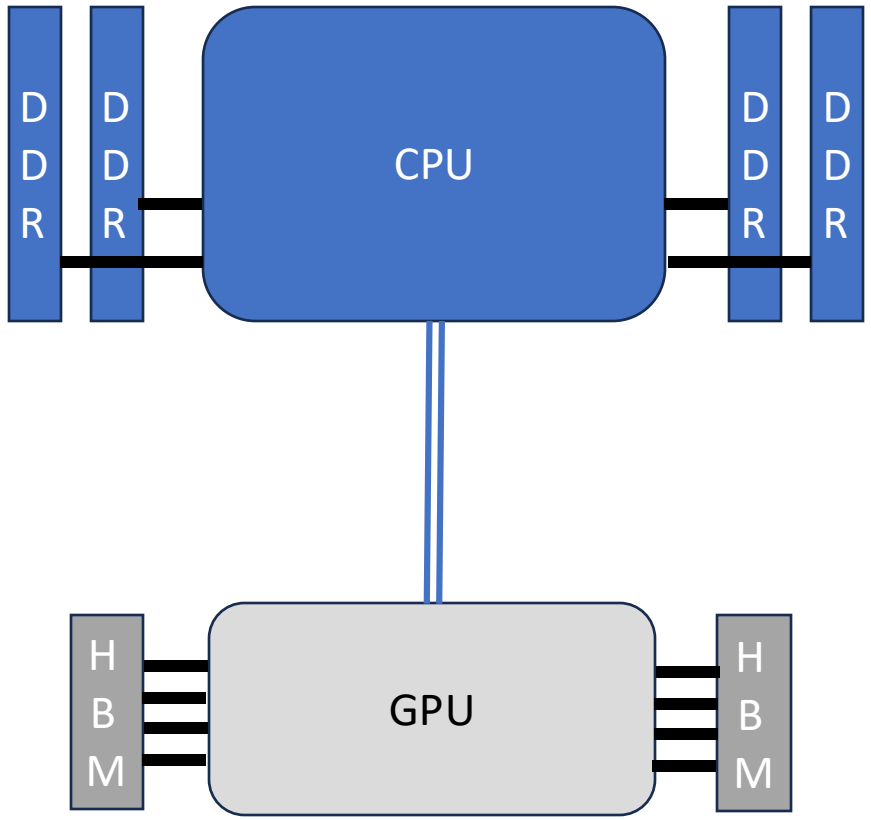
```
int main() {
  …
  double *X, *Y;
  X = new double[N];
  Y = new double[N];
  for (auto i=0; i<N; i++)
    {X[i] = 1.1; Y[i] = 1.3;}

  double *x_d, *y_d;
  cudaMalloc(&x_d,N*sizeoff(double));
  cudaMalloc(&y_d,N*sizeoff(double));
  cudaMemcpy(x_d,X,…);
  cudaMemcpy(y_d,Y,…);
  gpu_func<<< … >>>(x_d, y_d, N);
  cudaDeviceSynchronize();
  cudaMemcpy(X,x_d,…);
  cudaMemcpy(Y,y_d,…);

  cpu_func(X,Y);

  cudaFree(x_d); hipFree(y_d);
  delete[] X;
  delete[] Y;
}
nvcc name.cu
```

The Accelerator model : discrete CPU(s) + discrete GPU(s)



Initialize workload on CPU.
Optionally, prepare data on CPU then

copy the data to GPU's memory for computation on GPU

copy results back

Timing execution of GPU kernels

```
float time;
cudaEvent_t start, stop;

cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

my_daxpy<<<dimGrid, dimBlock, 0>>>(kernel arguments);

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);

fprintf(stderr,”my_daxpy execution time: %g [ms]\n", time);
```

```
gettimeofday(&t_start, 0);

my_daxpy <<<dimGrid, dimBlock, 0>>>(kernel arguments);
cudaThreadSynchronize();

gettimeofday(&t_stop, 0);
```

Memory allocators

cudaMalloc	Allocates DEVICE memory (GPU memory, page-locked)
cudaMallocManaged	Allocates memory that will be automatically managed by the Unified Memory system. Accessible by HOST and DEVICE.
cudaMallocHost	Allocates HOST memory (CPU memory, page-locked) accessible from HOST and DEVICE.
etc.	

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1g37d37965bfb4803b6d4e59ff26856356

Kernel and kernel launch

```
__global__
void kernel_name( kernel arguments){
  printf("Hello\n");
}
```

```
int main(){
     ...

   kernel_name<<<gridDimension,blockDimension,0,0>>>(kernel arguments);
   cudaDeviceSynchronize();
    ...
}
```

kernel launch

synchronize with Host

shared memory per thread-block (bytes)
cudastream

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#

Kernel and kernel launch

```
__global__
void kernel_name( kernel arguments){
  printf("Hello\n");
}
```

All threads in the Grid execute the same function

```
int main(){
     ...

   kernel_name<<<gridDimension,blockDimension,0,0>>>(kernel arguments);
   cudaDeviceSynchronize();
    ...
}
```

kernel launch

synchronize with Host

shared memory per thread-block (bytes)
cudastream

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#

Examples

```
__global__
  void VecAdd(float* A, float* B, float* C, size_t N) {

  size_t i = threadIdx.x+blockIdx.x*blockDim.x;
  size_t offset = blockDim.x * gridDim.x;
  for (size_t k = i; k < N; k+=offset) C[i] = A[i]+B[i];
 // if ( i < N)
  //   C[i] = A[i] + B[i];

}

int main() {
   //allocate memory for A,B,C and initialize
   dim3 nthreads = dim3(256,1,1);
   dim3 nblocks = dim3( (N+ nthreads.x – 1)/ nthreads.x,1,1);

   VecAdd<<<nblocks, nthreads>>>(A, B, C, N);

... }
```

Requesting interactive job with GPU on Oscar

GPU computing on Oscar:
https://docs.ccv.brown.edu/oscar/gpu-computing/gpus

interact -n 1 -g 1 -t 00:10:00

Useful links

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

https://cuda-tutorial.readthedocs.io/en/latest/