Module 4
A VERY simple introduction to GPU programming

To learn more: **CUDA C++ Programming Guide** (website)
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Overview

- GPUs & computing
- Principles of CUDA programming
- A couple of coding examples

New trends of microprocessors

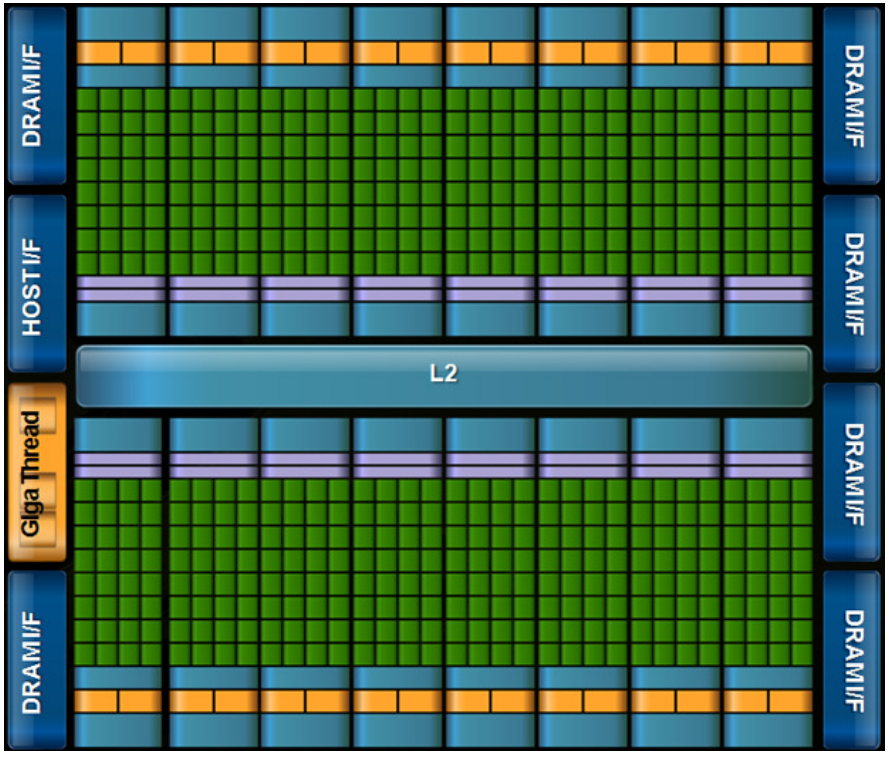
- Since 2003, there have been two main trajectories for microprocessor design
- *Multicore* – a relatively small number of cores per chip, each core is a full-flesh processor in the "traditional sense"
- *Manycore* – a large number of much smaller and simpler cores
  - A graphics processing unit (GPU) from NVIDIA can have thousands of "cores", each is heavily multi-threaded, in-order, single-instruction issue processor
  - Peak performance of many-core GPUs can be 10 fold the peak performance of multicore CPUs
  - GPUs have larger memory bandwidth
- There are also other GPU manufacturers now (such as AMD)

The top 3 supercomputers as of summer 2023

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016

No. 1 and No. 3 supercomputers rely on AMD GPUs for delivering the computing power.

NVIDIA's Fermi architecture (2010)



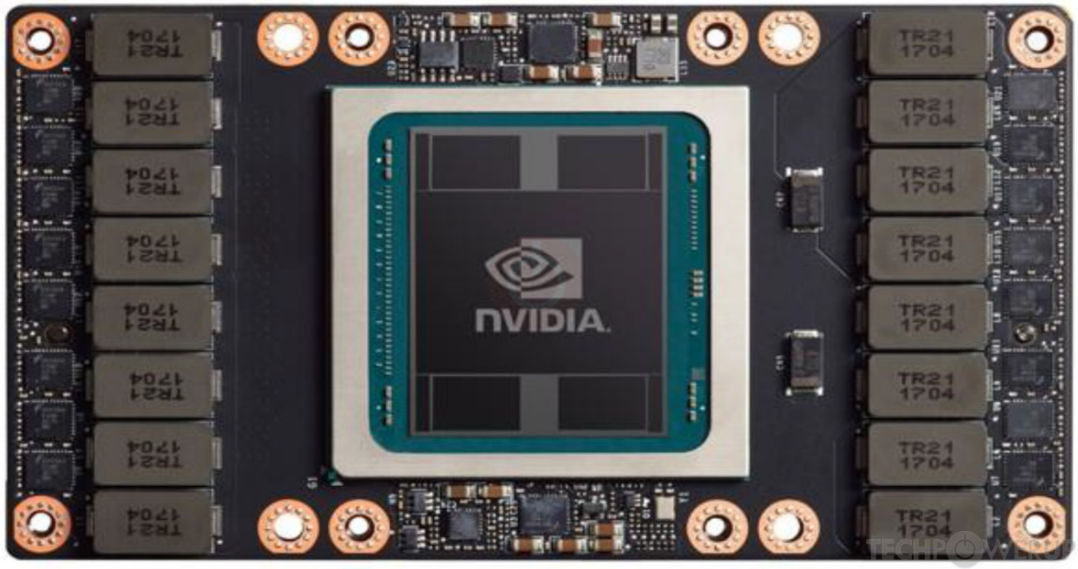
- Designed for GPU computing
- 16 streaming multiprocessors (SMs)
- 32 CUDA cores in each SM (512 cores in total)
- Each streaming processor is massively threaded
- Peak double-precision floating-point rate: 768 GFLOPs

NVIDIA's Kepler architecture (2012)



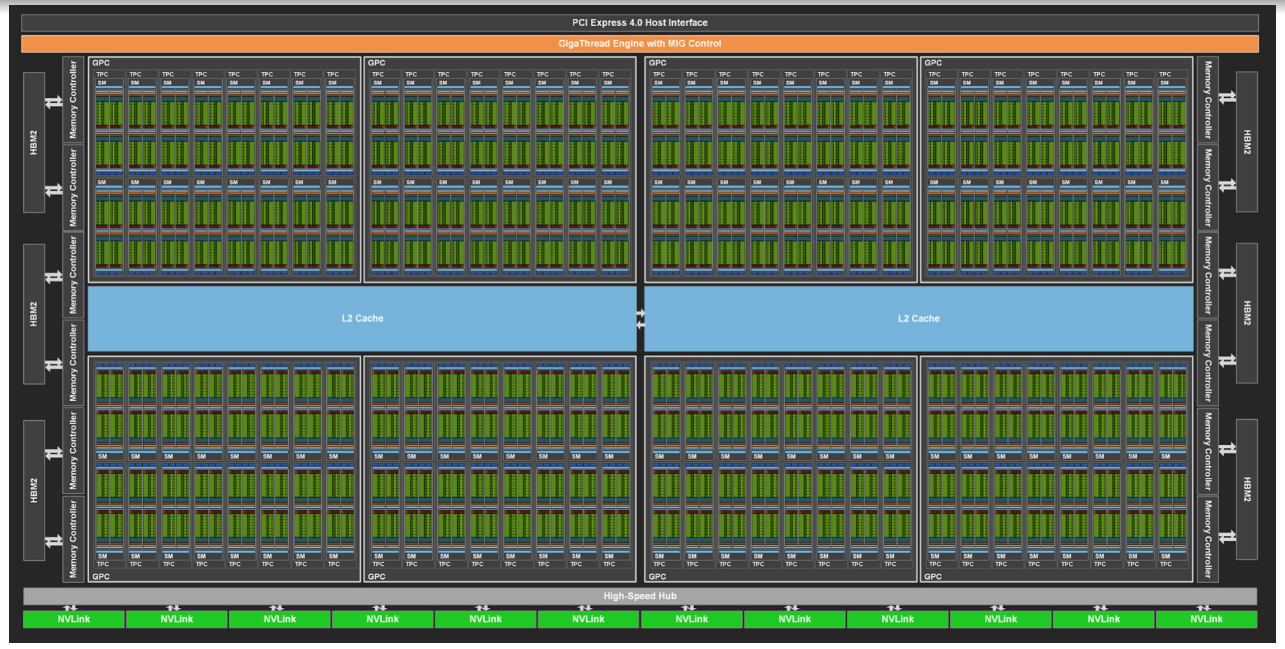
- Each streaming multiprocessor is more powerful than Fermi
- Peak double-precision floating-point rate of K40 GPU: 1430 GFLOPs

NVIDIA's Volta architecture (2017)



- 80 streaming multiprocessors (SMs)
- 5120 FP32 CUDA cores + 2560 FP64 CUDA cores in total
- 640 Tensor cores in addition
- Peak memory bandwidth 900 GB/s
- Peak double-precision floating-point rate of V100 GPU: 7500 GFLOPs

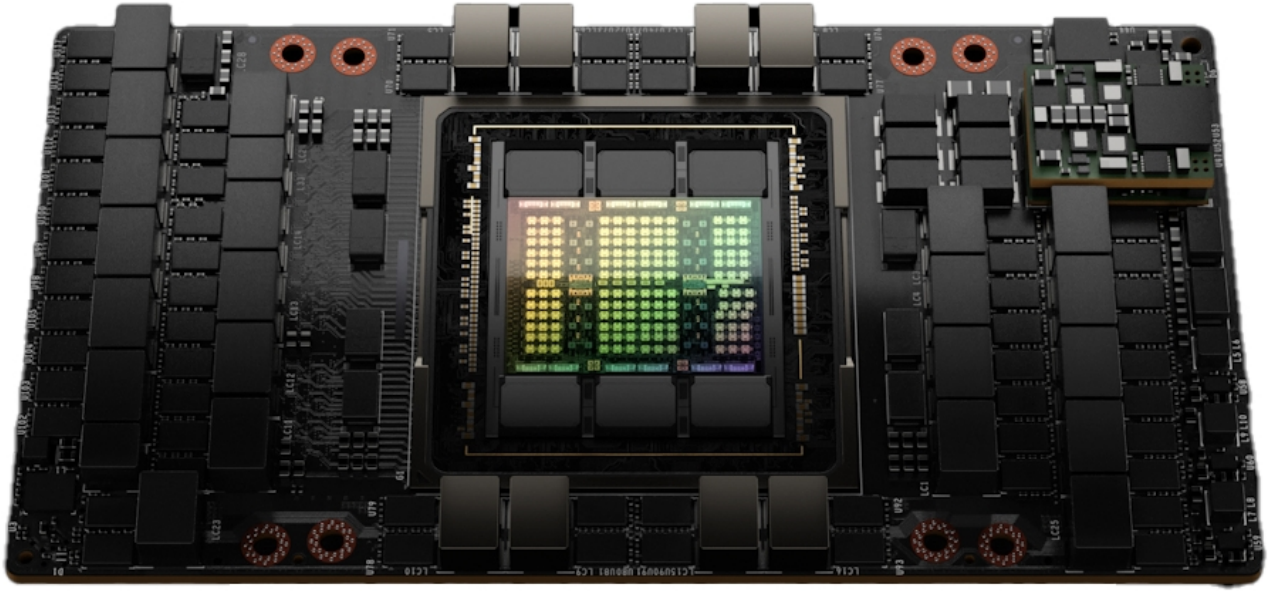
NVIDIA's Ampere architecture (2020)



- 108 streaming multiprocessors (SMs)
- 6912 FP32 CUDA cores + 3456 FP64 CUDA cores in total
- 432 third-generation Tensor cores in addition
- Peak memory bandwidth 1555 GB/s
- Peak double-precision floating-point rate of A100 GPU: 9700 GFLOPs

  Youtube video of the official release (please click the link)

NVIDIA's Hopper architecture (2022)



- (up to) 144 streaming multiprocessors (SMs)
- (up to) 576 fourth-generation Tensor cores in addition
- HBM3 memory with 3.35 TB/sec of bandwidth

GPUs can handle massive (but simple) parallelism

- A streaming processor (CUDA core) is the most fundamental execution resource
- Simpler than a CPU core
- Capable of executing a large number of threads simultaneously, where the threads carry out same instruction to different data elements — single-instruction-multiple-data (SIMD) all called single-instruction-multiple-thread (SIMT) in NVIDIA terminology
- The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.
- A number of streaming processors constitute one streaming multiprocessor (SM)
- Each SM can schedule and execute threads in groups of 32 parallel threads called *warps*.

CUDA (specific for NVIDIA GPUs)

- CUDA – Compute Unified Device Architecture
- C-based programming model for GPUs
- Introduced together with GeForce 8800
- Joint CPU/GPU execution (*host/device*)
- A CUDA program consists of one of more phases that are executed on either host or device
- User needs to manage data transfer between CPU and GPU
- A CUDA program is a unified source code encompassing both host and device code

There are other ways to do GPU programming as well! (such as OpenCL, OpenACC, AMD HIP)

More about CUDA

- To a CUDA programmer, the computing system consists of a host (CPU) and one or more devices (GPUs)
- Data can be transferred between host and device (either explicitly enforced by a programmer or automatically "managed" through *Unified Memory*)
- On each device, there is so-called *global memory*
- Device global memory tends to have long access latencies and finite bandwidth
- On-chip memories: registers and shared memory (**per SM block**) have limited capacity, but are much faster
- Registers are private for individual threads
- All threads within a thread block can access variables in the shared memory
- Shared memory (per SM block) is an efficient means for threads to cooperate, by sharing input data and intermediate results

Programming GPUs for computing (1)

- *Kernel functions* – the computational tasks on GPU
  - An application or library function may consist of one or more kernels
  - Kernels can be written in C, extended with additional keywords to express parallelism
- Once compiled, each kernel makes use of many threads that **execute the same program in parallel**
- Multiple threads are grouped into thread blocks
  - All threads of a thread block run on a single SM
  - Within a thread block, threads cooperate and share memory
  - A thread block is divided into warps of 32 threads
    - Warp is the fundamental unit of dispatch within an SM
  - Thread blocks may execute in any order

Programming GPUs for computing (2)

- When a kernel is invoked by the CPU host, a *grid* of parallel threads are generated on the GPU device
- Threads in a grid are organized in a two-level hierarchy
  - each grid consists of one or more thread blocks
  - all blocks in a grid have the same number of threads
  - each block has a unique 3D coordinate `blockIdx.x`, `blockIdx.y` and `blockIdx.z`
  - each thread block is organized as a 3D array of threads `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
- The grid and thread block dimensions are set when a kernel is invoked

Programming GPUs for computing (3)

- Syntax for invoking a kernel

```
dim3 dimGrid(64,32,1)
dim3 dimBlock(4,2,2);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

- `gridDim` and `blockDim` contain the dimension info
- All threads in a block share the same `blockIdx`
- Each thread has its unique `threadIdx` within a block
- Values of `blockIdx` and `threadIdx` can be used to determine which data element(s) that a thread is to work on
- Often, one thread is used to compute one data element (fine-grained parallelism)

Thread execution

- Launching a CUDA kernel will generate a 1D or 2D or 3D array of thread blocks, each having a 1D or 2D or 3D array of threads
- The thread blocks can execute in any order relative to each other
- The CUDA runtime system bundles several threads for simultaneous execution, by partitioning each thread block into *warps* (32 threads)
- Scheduling of warps is taken care by CUDA runtime system
- The hardware executes same instruction for all threads in the same warp

A simple example of CUDA program

Use GPU to calculate the square of each element in an array

```
// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
  // 1D thread blocks and 1D thread array inside each block
  // N is the length of array a, stored in device memory

  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

A simple example of CUDA program (cont'd)

```
// main routine that executes on the CPU host
int main(void)
{
  float *a_h, *a_d;  // Pointer to host & device arrays
  const int N = 10;  // Number of elements in arrays
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);       // Allocate array on host
  cudaMalloc((void **) &a_d, size); // Allocate array on device
  // Initialize host array and copy it to CUDA device
  for (int i=0; i<N; i++) a_h[i] = (float)i;
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  // Do calculation on device:
  int block_size = 4;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  square_array <<< n_blocks, block_size >>> (a_d, N);
  // Retrieve result from device and store it in host array
  cudaMemcpy(a_h, a_d, size, cudaMemcpyDeviceToHost);
  // Print results
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  // Cleanup
  free(a_h); cudaFree(a_d);
}
```

Matrix multiplication, version 1

- We want to compute $Q = M \times N$, assuming Q, M, N are all square matrices of same size `Width`×`Width`
- Each matrix has a 1D contiguous data storage
- A naive kernel implementation is as follows:

```
__global__ void MatrixMulKernel(float* Md,
                                 float* Nd,
                                 float* Qd,
                                 int Width)
{
  int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
  int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;
  float Qvalue = 0;
  for (int k=0; k<Width; ++k)
    Qvalue += Md[Row*Width+k] * Nd[k*Width+Col];
  Qd[Row*Width+Col] = Qvalue;
}
```

Matrix multiplication, version 2

- The previous implementation is not memory efficient
- Each thread reads 2×`Width` values from the device's global memory
- A better approach is to let the threads of each thread block (`TILE_WIDTH`×`TILE_WIDTH` threads) share the 2×`TILE_WIDTH`×`Width` data reads
- That is, each thread reads 2×`Width`/`TILE_WIDTH` values from global memory
- Shared memory can be important for performance (otherwise we need to rely on L1/L2 caches)
- Also beware that size of shared memory is limited

Matrix multiplication, version 2 (cont'd)

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Qd,
                                int Width)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Qvalue = 0;
  for (int m = 0; m<Width/TILE_WIDTH; ++m) {
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
    __syncthreads();

    for (int k=0; k<TILE_WIDTH; ++k)
      Qvalue += Mds[ty][k] * Nds[k][tx];
    __syncthreads();
  }

  Qd[Row*Width+Col] = Qvalue;
}
```

Some useful YouTube videos about CUDA programming

- CUDA Crash Course: Vector Addition (please click the link)
- CUDA Crash Course: Matrix Multiplication (please click the link)
- ... and more

About HPC & Python (not about CUDA!!!)

Python has the reputation of being slow for numerical computation. However, there are tools and libraries that can help with getting close to the speed of C or other compiled languages

- Introduction to High-Performance Computing in Python (please click the link)
- Mixed programming with Python (please click the link)
- ... and more