CMSC 603
High-Performance Distributed Systems

Introduction to CUDA

Dr. Alberto Cano
Associate Professor
Department of Computer Science
acano@vcu.edu

The CPU vs the GPU: purposes

CPU

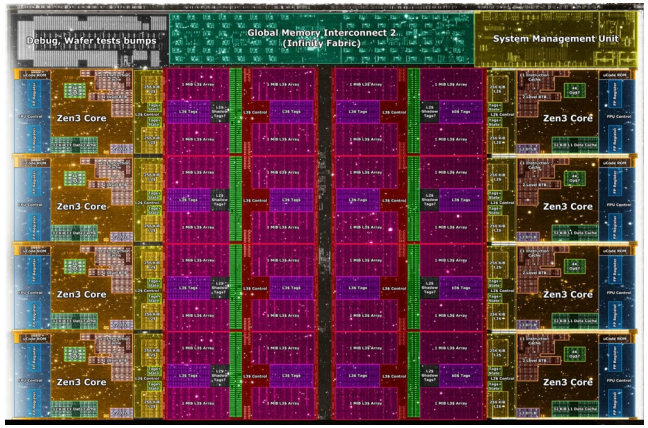
- General-purpose computation
- Small number of highly specialized complex cores
- Fast execution of a single stream of instructions (minimize latency)
- Pipelining, caching, branch prediction, our-of-order execution, interruptions
- Main DDR4 memory ~64 GB/s

GPU

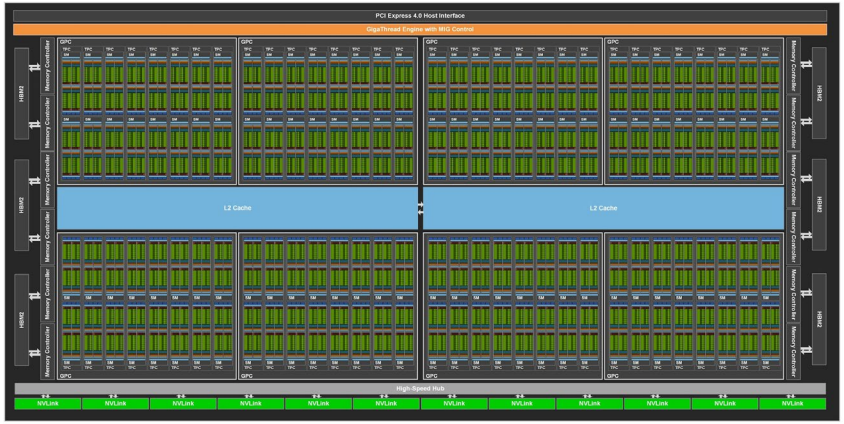
- Originally for graphics computing
- Large number of simple cores for parallel SIMD computation
- Large FP bandwidth for massive parallel number of operations
- No fancy hardware tricks except for asynchronous data / execution
- GPU GDDR6X memory > 512 GB/s
- GPU HBM3 memory > 3 TB/s

The CPU vs the GPU: architectures

CPU (AMD Zen 3)



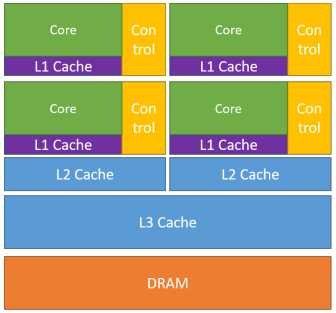
GPU (NVIDIA A100)



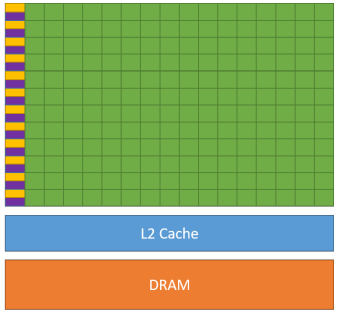
4,8,16 cores

4.15 billion transistors

967 GFlops FP32



CPU



GPU

6912 cores

432 tensor cores

54 billion transistors

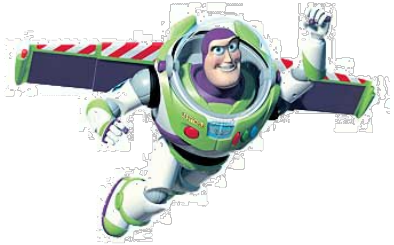
19.5 TFlops FP32 (x2 FP16)

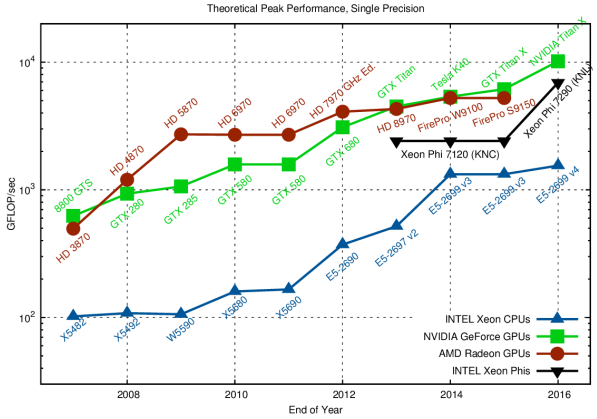
The GPU architecture (NVIDIA A100)

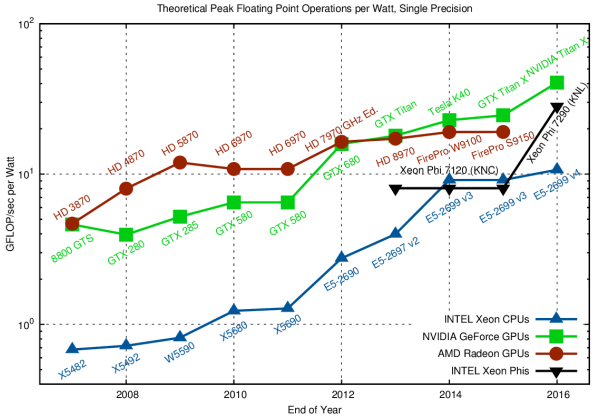
- Streaming Multiprocessors (SMs)
- FP16/FP32 and INT8/INT4 mixed-precision

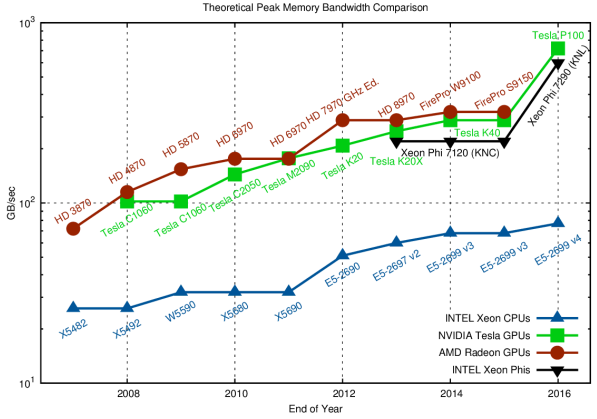
	Peak Performance
Transistor Count	54 billion
Die Size	826 mm^2
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS | 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS | 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS | 624 teraFLOPS*
INT8 Tensor Core	624 TOPS | 1,248 TOPS*
INT4 Tensor Core	1,248 TOPS | 2,496 TOPS*
GPU Memory	40 GB
GPU Memory Bandwidth	1.6 TB/s
Interconnect	NVLink 600 GB/s PCIe Gen4 64 GB/s
Multi-Instance GPUs	Various Instance sizes with up to 7MIGs @5GB
Form Factor	4/8 SXM GPUs in HGX A100
Max Power	400W (SXM)

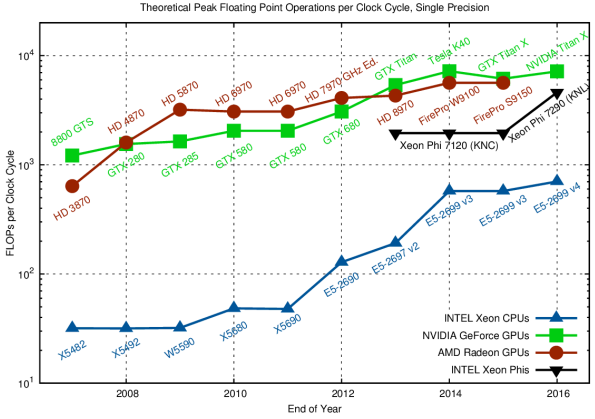
The GPU evolution, performance and bandwidth





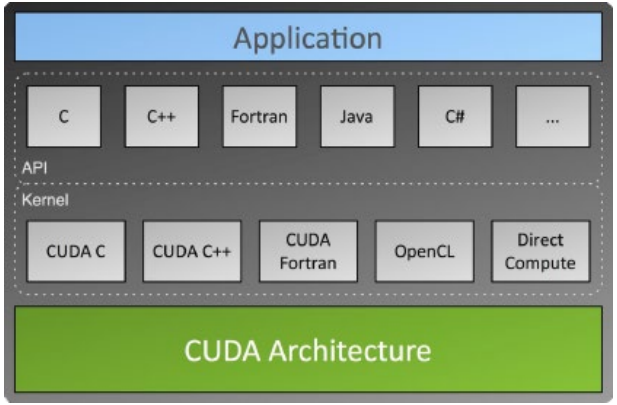






Exploiting the GPU

- CUDA C/C++
- CUDA Fortran
- OpenCL
- PyCUDA
- jCUDA
- Matlab
- TensorFlow
- OpenCV



Many Different Approaches

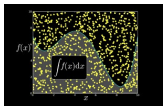
- Application level integration
- High level, implicit parallel languages
- Abstraction layers & API wrappers
- High level, explicit language integration
- Low level device APIs

Some interesting libraries for high-performance computing

- Thrust: parallel algorithms and data structures, e.g. transformation, reductions, sorting

- cuDNN: CUDA Deep Neural Network is a GPU-accelerated library of primitives for neural networks

- nvGRAPH: Graph Analytics Library for GPUs



- cuRAND: Random Number Generation library
- cuBLAS: Basic Linear Algebra Subroutines

- TensorFlow: Software Library for Machine Learning

Terminology

- **Host**: the CPU and its memory space
- **Device**: the GPU and its memory space
- Code, computation, and memory spaces managed differently
- Parallel code in a GPU is implemented in a **kernel** function

Thread hierarchy in a GPU

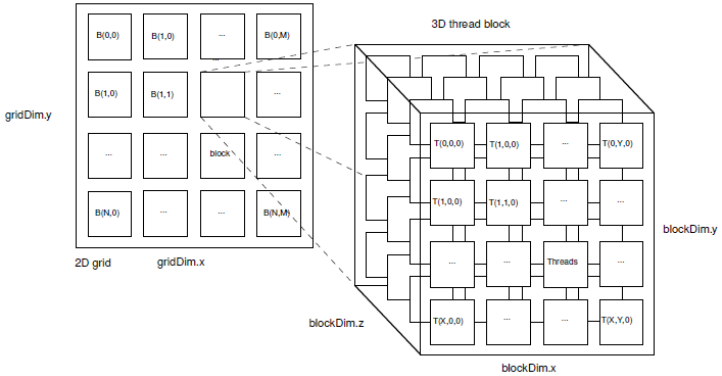
- **Thread**: executes a sequence of instructions implemented in a kernel function
- **Block**: group of threads defined by the programmer
- **Grid**: group of blocks defined by the programmer

Please read the NVIDIA CUDA programming guide

CUDA multi-dimensional thread hierarchy

- Programmer defines 1/2/3-dimensional **blocks of threads** (max 1024 threads/block)
- Programmer defines 1/2/3-dimensional **grid of blocks** (max [2^32, 64k, 64k] blocks)

2^74 threads



- **Warp**: group of 32 threads scheduled/executed in a multi-processor
- **Multi-processor**: group of physical CUDA cores executing the kernel function
- CUDA threads are lightweight, no creation overhead, context switching is essentially free and allows to easily hide data latencies

Memory hierarchy

- **Thread local** memory

Each thread has its private local memory

- **Block shared** memory

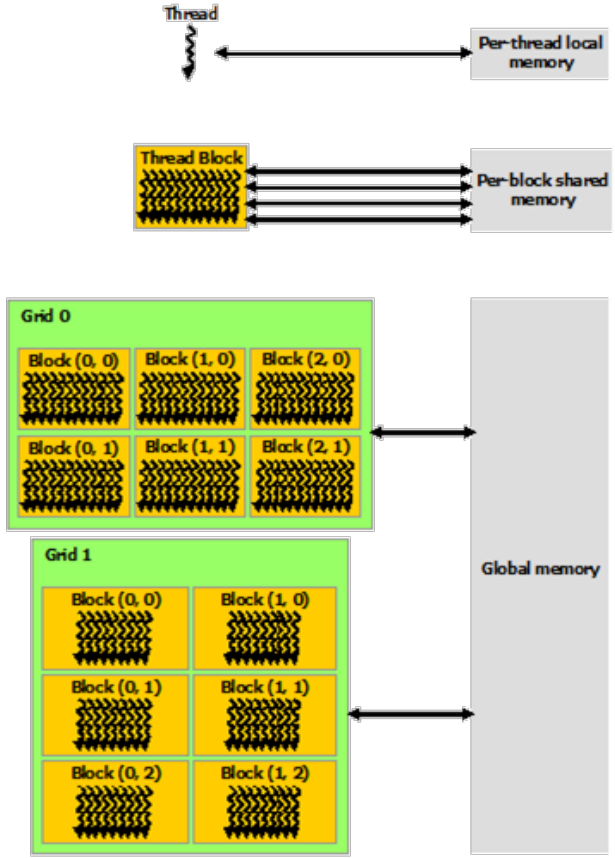
Each thread block has its shared memory visible to all threads of the block and with the same lifetime as the active block

- **Device global** memory

All threads have access to the same global memory. Lifetime of the program

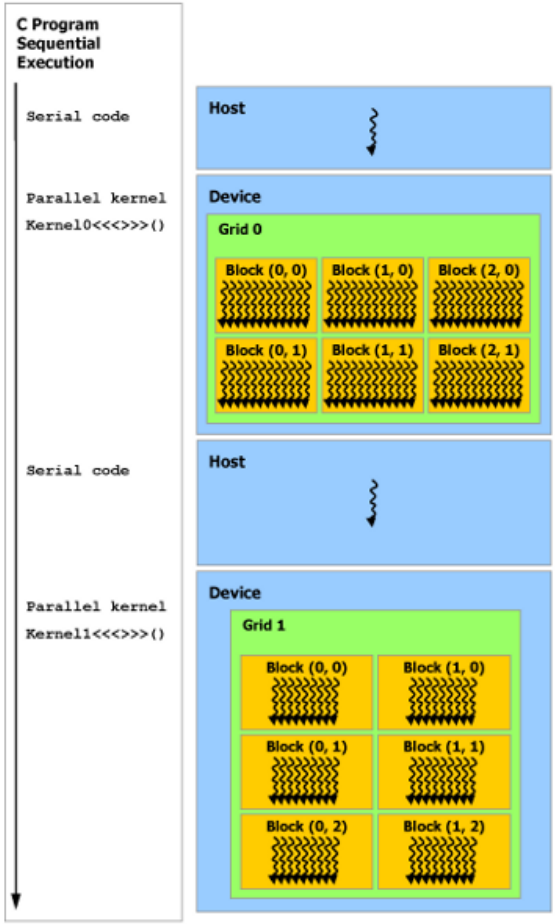
- **Host global** memory

Data on main memory (CPU side)



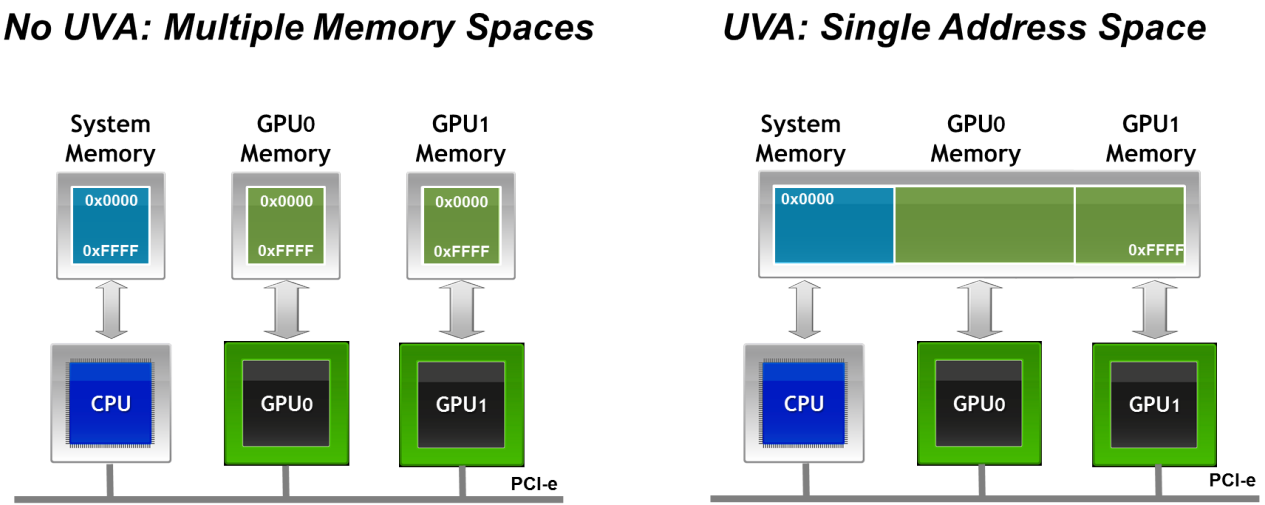
Heterogeneous programming

- Programs interleave code executed on the host (CPU) and GPU (device)
- GPU code is written in ***kernel*** functions
- Typically, the sequence of a program is:
  1. Allocate CPU memory inputs/outputs
  2. Allocate GPU memory inputs/outputs
  3. Copy inputs from host to device
  4. Execute GPU code
  5. Copy outputs from device to host



Multiple memory spaces vs unified memory

- CPU -> GPU -> CPU data transfers on isolated memory spaces
- *h_variableName* and *d_variableName* to reference memory spaces
- Single virtual address space for unified memory



Structure of a CUDA program

```
__global__ void vectorAdd(float* a, float* b, float* c) { // GPU kernel
    c[tid] = a[tid] + b[tid];
}

void main(void) {
    // Allocate CPU and GPU memory                          d_a, d_b, d_c, h_a, h_b, h_c
    // Copy host data to device memory                      h_a -> d_a, h_b -> d_b
    // Execute GPU kernel
    vectorAdd <<< gridSize,blockSize >>> (d_a, d_b, d_c);
    // Copy device results to host memory                   d_c -> h_c
}
```

- NVIDIA compiler nvcc (can be used for programs with no GPU code)

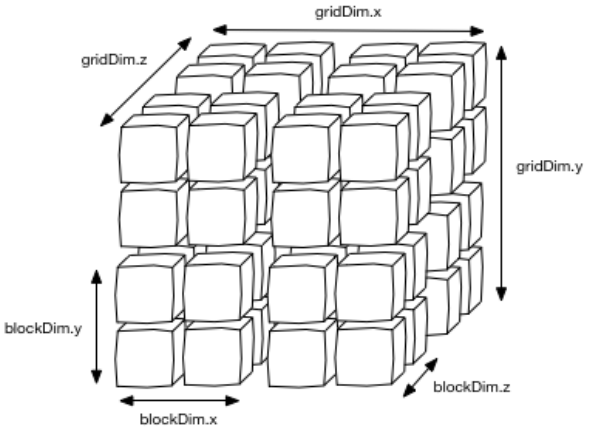
  $ nvcc –o myprogram mycode.cu

- Threads are grouped into multi-dimensional blocks
- Blocks are grouped into a multi-dimensional grid

Thread indexing

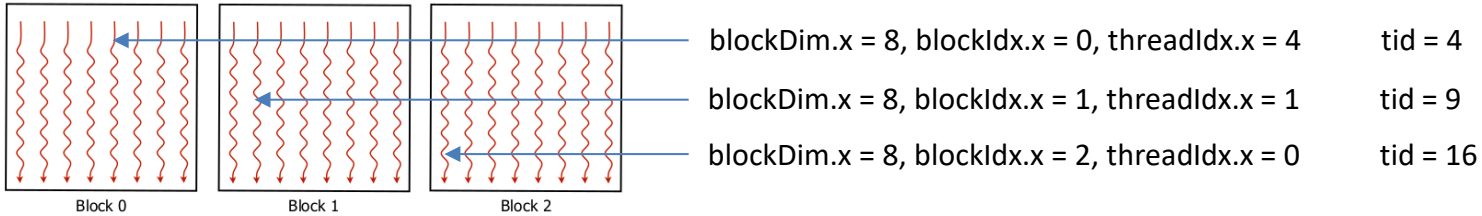
- Thread space of up to 3D grid of 3D blocks
- Built-in variables within the kernel function

Grid size [x,y,z]	*gridDim.x*	*gridDim.y*	*gridDim.z*
Block size [x,y,z]	*blockDim.x*	*blockDim.y*	*blockDim.z*
Block ID [x,y,z]	blockIdx.*x*	*blockIdx.y*	*blockIdx.z*
Thread ID [x,y,z]	*threadIdx.x*	*threadIdx.y*	*threadIdx.z*



- Indexing thread ID in a 1D grid of 1D blocks:

*int tid = blockIdx.x * blockDim.x + threadIdx.x;*



Memory allocation and transfer

```
cudaMallocHost(&h_ptr, count * sizeof(datatype));
cudaMalloc(&d_ptr, count * sizeof(datatype));

cudaMemcpy(dst_ptr, src_ptr, count * sizeof(datatype), cudaMemcpyKind);

cudaFreeHost(h_ptr);
cudaFree(d_ptr);
```

CUDA memory copy types (`cudaMemcpyKind`)

cudaMemcpyHostToDevice	Host -> Device	(+ HostToHost alternative)
cudaMemcpyDeviceToHost	Device -> Host	(+ DeviceToDevice alternative)

Kernel setup

```
dim3 gridDim(16,16,1);      // 2D grid with 16x16x1 = 256 blocks [x,y,z]
dim3 blockDim(16,8,4);      // 3D block with 16x8x4 = 512 threads [x,y,z]

MyKernel <<< gridDim, blockDim >>> (d_data ...);
```

VectorAdd example

```
__global__ void vectorAdd(float *A, float *B, float *C) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    C[tid] = A[tid] + B[tid];
}

void main(void)
{
    float *h_A = (float *)malloc(numElements * sizeof(float));
    …
    cudaMalloc(&d_A, numElements * sizeof(float));
    …
    cudaMemcpy(d_A, h_A, numElements*sizeof(float), cudaMemcpyHostToDevice);
    …
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);
    cudaMemcpy(h_C, d_C, numElements*sizeof(float), cudaMemcpyDeviceToHost);
}
```

- Careful! Let's see the full working code: see **vectorAdd.cu**
- Use *cuda-memcheck yourprogram* to find memory errors

CMSC 603
High-Performance Distributed Systems

Introduction to CUDA

Dr. Alberto Cano
Associate Professor
Department of Computer Science
acano@vcu.edu