CMSC 691
High Performance Distributed Systems

Introduction to CUDA

Dr. Alberto Cano
Assistant 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
- Pipelining, caching, branch prediction, our-of-order execution, interruptions
- Main DDR memory ~ 20 GB/s

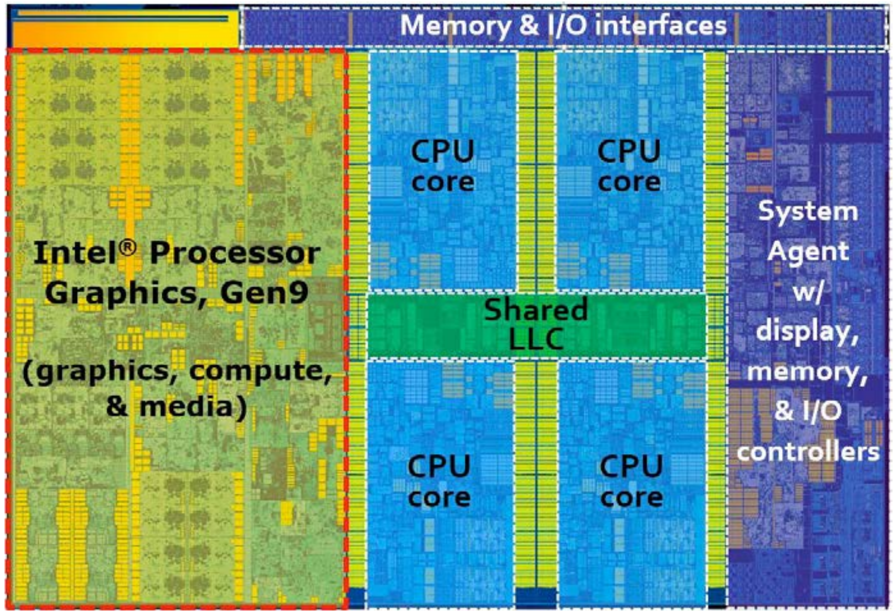
GPU

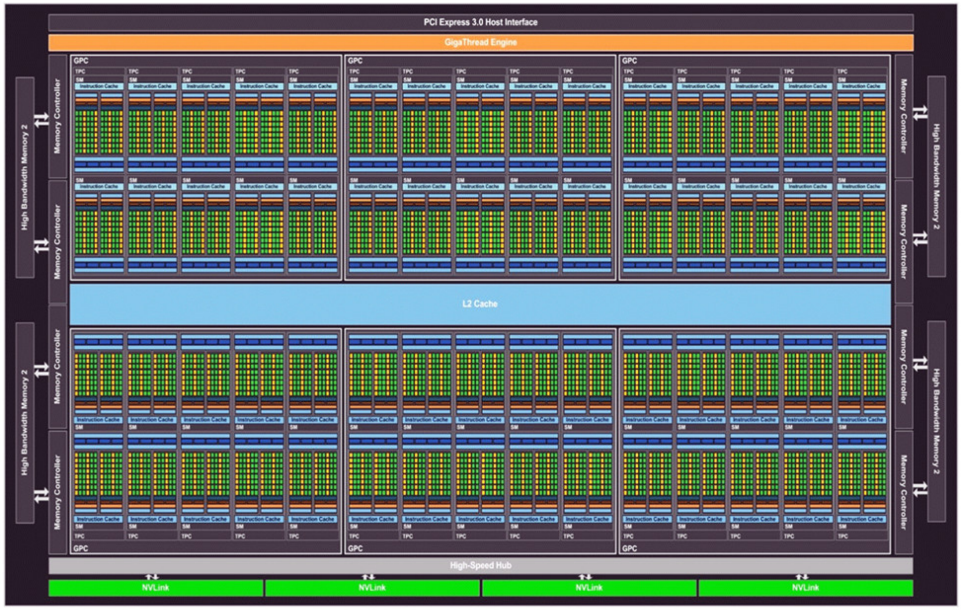
- Originally for graphics computing
- Large number of simple cores for parallel SIMD computation
- Large FP bandwidth for massive parallel number of operations
- No fancy hardware tricks except asynchronous data / execution
- GPU memory GDDR > 200 GB/s
- Future: HBM2 > 1 TB/s

The CPU vs the GPU architectures

CPU (Skylake)

GPU (P100)

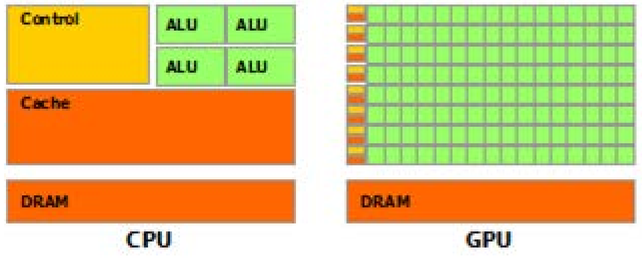

Memory & I/O interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers



4 cores

1.75 billion transistors

110 Gflops FP32

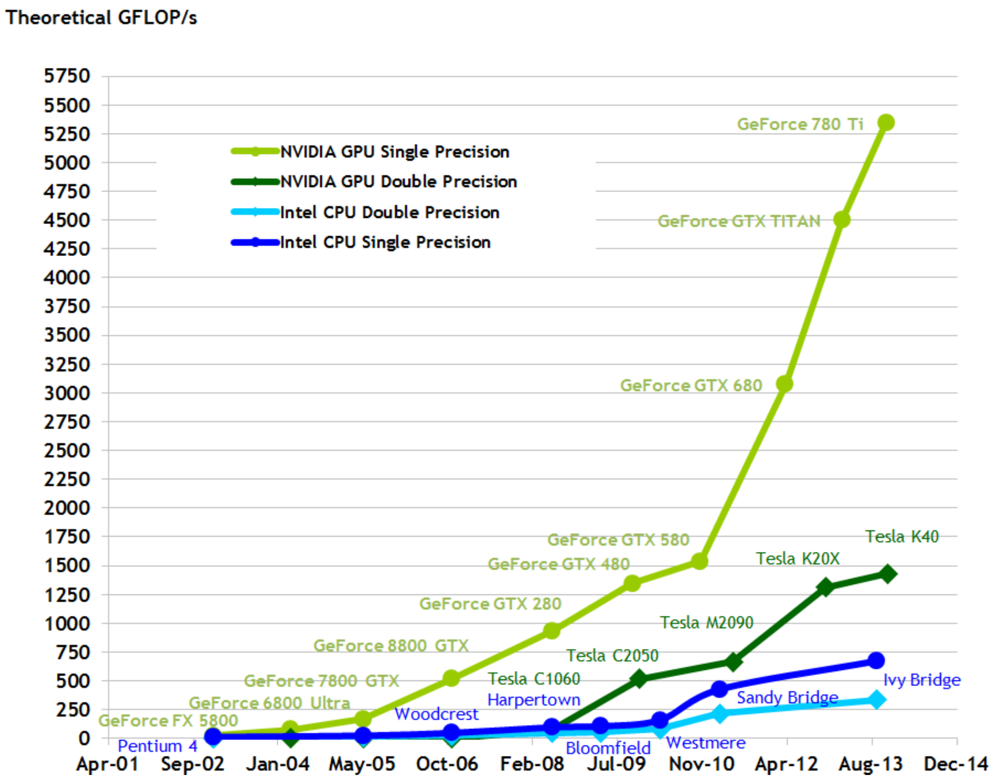


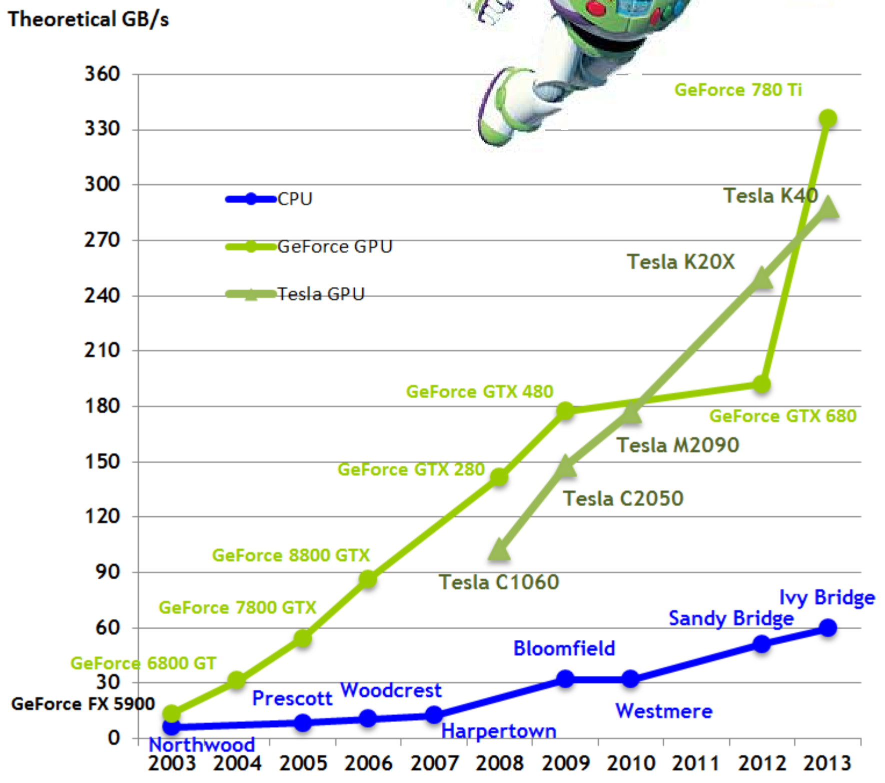
3840 cores

15 billion transistors

10 Tflops FP32

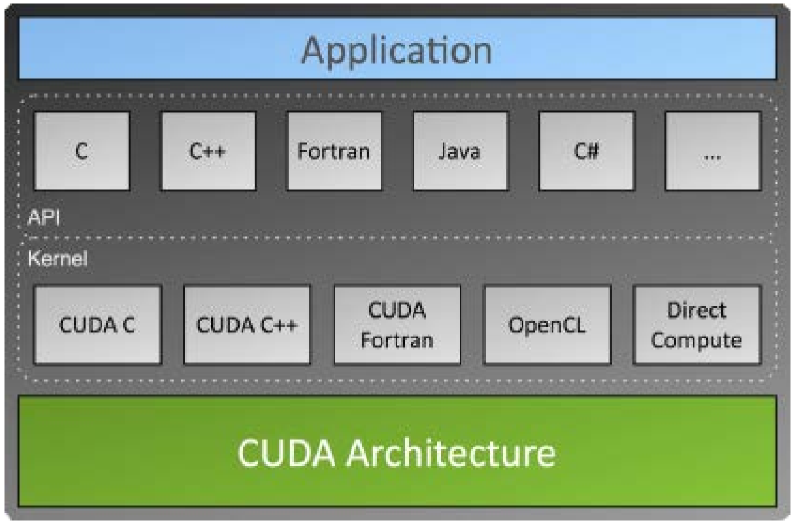
The GPU evolution, performance and bandwidth





Exploiting the GPU

- CUDA C/C++
- CUDA Fortran
- OpenCL
- PyCUDA
- jCUDA
- Matlab
- Tensor flow
- OpenCV

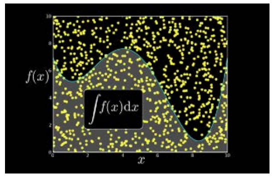


Many Different Approaches

- 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

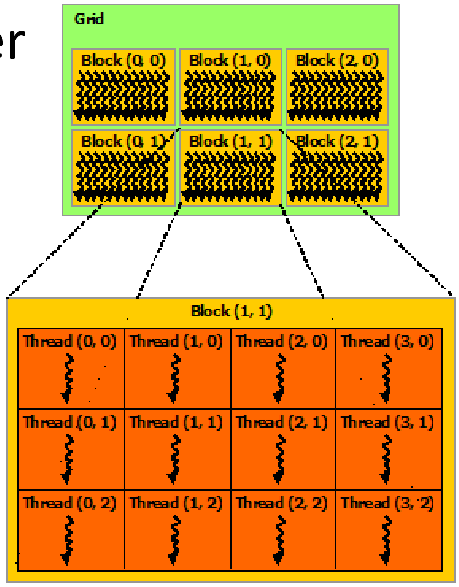


Terminology

- Host: the CPU and its memory, typically *h_variable*
- Device: the GPU and its memory, typically *d_variable*

Thread Hierarchy

- Thread: smallest sequence of programmed instructions
- Warp: group of 32 threads scheduled/executed in a multi-processor
- Block: group of threads defined by the programmer
- Grid: group of blocks defined by the programmer
- Multi-processor: group of physical CUDA cores



CUDA threads are extremely lightweight, almost no creation overhead, context switching is essentially free

Memory Hierarchy

- Thread local memory

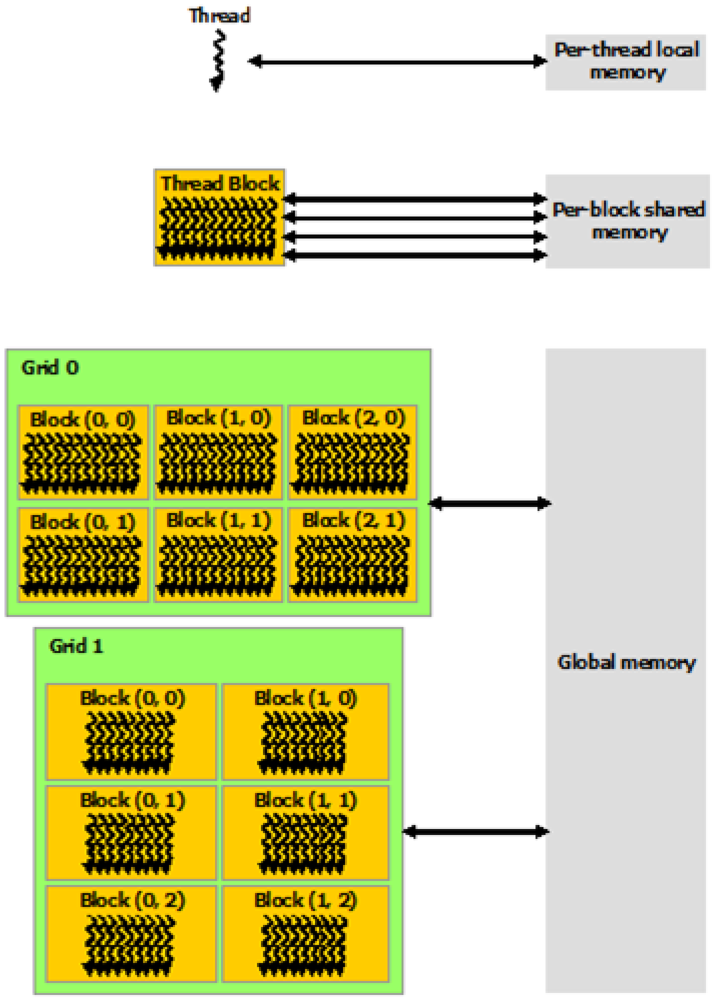
Each thread has private local memory

- Block shared memory

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

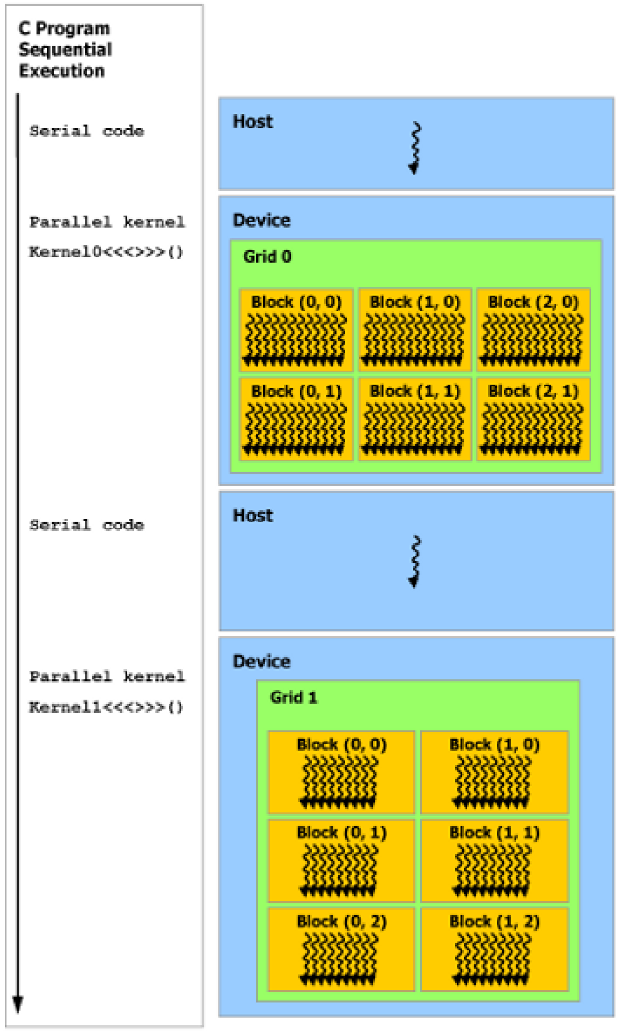
- Device global memory

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



Heterogeneous Programming

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



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
    deviceKernel <<< 1,size >>> (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 thread blocks
- Thread blocks are grouped into a multi-dimensional grid

Thread indexing

- Thread space 3D grid of 3D blocks

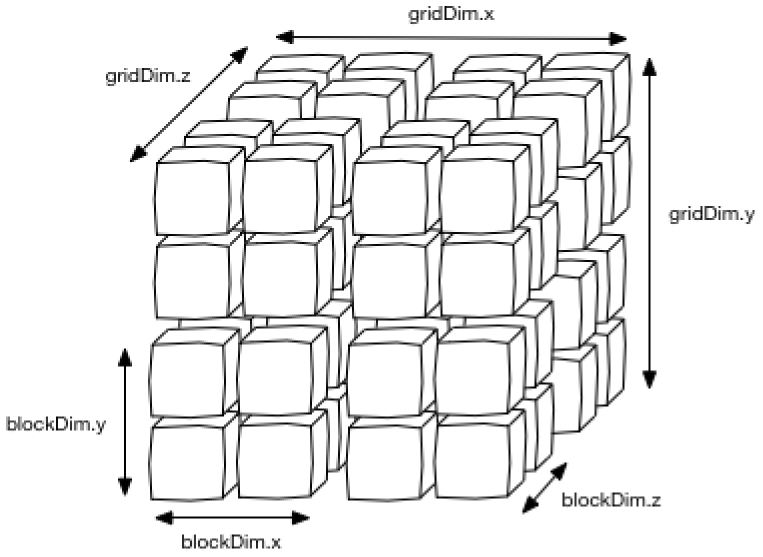
- Built-in variables

gridDrim.x gridDim.y gridDim.z

blockDim.x blockDim.y blockDim.z

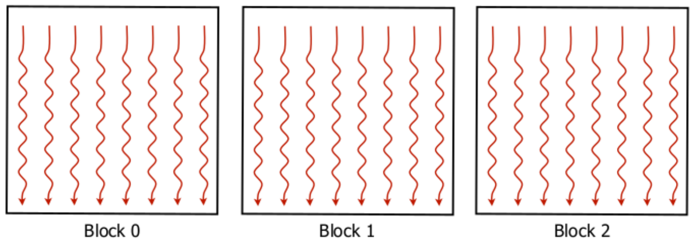
blockIdx.x blockIdx.y blockIdx.z

threadIdx.x threadIdx.y threadIdx.z



- Indexing 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

cudaMemcpyHostToDevice	Host -> Device
cudaMemcpyDeviceToHost	Device -> Host

Kernel setup

```
dim2 gridDim(1,1);  // 1D grid
dim3 blockDim(256,1,1); // 1D block with 256 threads

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
- Use *cuda-memcheck yourprogram* to find memory errors

CMSC 691
High Performance Distributed Systems

Introduction to CUDA

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