P&S Heterogeneous Systems

GPU Software Hierarchy:
Grids, Blocks, Threads

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Fall 2022
17 October 2022
GPUs are SIMD Engines
Underneath
108 cores on the A100
(Up to 128 cores in the full-blown chip)
40MB L2 cache

https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
NVIDIA A100 Core

19.5 TFLOPS Single Precision
9.7 TFLOPS Double Precision
312 TFLOPS (FP16, Tensor Cores)

https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
Evolution of NVIDIA GPUs
Evolution of NVIDIA GPUs (Updated)
NVIDIA H100 Block Diagram

144 cores on the full GH100
60MB L2 cache
NVIDIA H100 Core

48 TFLOPS Single Precision*
24 TFLOPS Double Precision*
800 TFLOPS (FP16, Tensor Cores)*

* Preliminary performance estimates

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
Recall: Latency Hiding via Warp-Level FGMT

- **Warp**: A set of threads that execute the same instruction (on different data elements)

- **Fine-grained multithreading**
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies

- Register values of all threads stay in register file

- FGMT enables long latency tolerance
  - Millions of pixels

Slide credit: Tor Aamodt
Recall: Warp Execution

32-thread warp executing ADD A[tid],B[tid] → C[tid]

Execution using one pipelined functional unit


Execution using four pipelined functional units

A[27]  B[27]
A[22]  B[22]

Time

Space

Slide credit: Krste Asanovic
Recall: SIMD Execution Unit Structure

- Functional Unit
- Registers for each Thread
- Lane

Registers for thread IDs 0, 4, 8, ...
Registers for thread IDs 1, 5, 9, ...
Registers for thread IDs 2, 6, 10, ...
Registers for thread IDs 3, 7, 11, ...

Memory Subsystem

Slide credit: Krste Asanovic
Recall: Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle
GPU Programming
Recall: Vector Processor Disadvantages

-- Works (only) if parallelism is regular (data/SIMD parallelism)
  +++ Vector operations
-- Very inefficient if parallelism is irregular
  -- How about searching for a key in a linked list?

To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That’s hard to do in the first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is

General Purpose Processing on GPU

- Easier programming of SIMD processors with SPMD
  - GPUs have democratized High Performance Computing (HPC)
  - Great FLOPS/$, massively parallel chip on a commodity PC
- Many workloads exhibit inherent parallelism
  - Matrices
  - Image processing
  - Deep neural networks
- However, this is not for free
  - New programming model
  - Algorithms need to be re-implemented and rethought
- Still some bottlenecks
  - CPU-GPU data transfers (PCIe, NVLINK)
  - DRAM memory bandwidth (GDDR5, GDDR6, HBM2, HBM3)
    - Data layout
Recommended Readings (I)

Recommended Readings (II)

- CUDA Programming Guide

**CUDA C++ Programming Guide**

The programming guide to the CUDA model and interface.

**Changes from Version 11.3**

- Added Graph Memory Nodes.
- Formalized Asynchronous SIMT Programming Model.

1. Introduction

1.1. The Benefits of Using GPUs

The Graphics Processing Unit (GPU) provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU (see GPU Applications). Other computing devices, like FPGAs, are also very energy efficient, but offer much less programming flexibility than GPUs.

This difference in capabilities between the GPU and the CPU exists because they are designed with different goals in mind. While the CPU is designed to excel at executing a sequence of operations, called a thread, as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at executing thousands of them in parallel (amortizing the slower single-thread performance to achieve greater throughput).

The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. The schematic Figure 1 shows an example distribution of chip resources for a CPU versus a GPU.
CPU vs. GPU

- Different design philosophies
  - CPU: A few out-of-order cores
  - GPU: Many in-order FGMT cores
GPU Computing

- Computation is **offloaded to the GPU**
- Three steps
  - CPU-GPU data transfer (1)
  - GPU kernel execution (2)
  - GPU-CPU data transfer (3)
Traditional Program Structure

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nThr >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nThr >>>(args);
Recall: SPMD

- Single procedure/program, multiple data
  - This is a programming model rather than computer organization

- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers

- Essentially, multiple instruction streams execute the same program
  - Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD hardware
CUDA/OpenCL Programming Model

- SIMT or SPMD

- Bulk synchronous programming
  - Global (coarse-grain) synchronization between kernels

- The host (typically CPU) allocates memory, copies data, and launches kernels

- The device (typically GPU) executes kernels
  - Grid (NDRange)
  - Block (work-group)
    - Within a block, shared memory, and synchronization
  - Thread (work-item)
Traditional Program Structure in CUDA

- Function prototypes
  ```
  float serialFunction(...);
  __global__ void kernel(...);
  ```

- main()
  1) Allocate memory space on the device – cudaMemcpy(&d_in, bytes);
  2) Transfer data from host to device – cudaMemcpy(d_in, h_in, ...);
  3) Execution configuration setup: #blocks and #threads
  4) Kernel call – kernel<<<execution configuration>>>(args...);
  5) Transfer results from device to host – cudaMemcpy(h_out, d_out, ...);

- Kernel – __global__ void kernel(type args,...)
  ```
  Automatic variables transparently assigned to registers
  Shared memory: __shared__
  Intra-block synchronization: __syncthreads();
  ```
CUDA Programming Language

- **Memory allocation**
  
  ```c
cudaMalloc((void**)&d_in, #bytes);
  ```

- **Memory copy**
  
  ```c
cudaMemcpy(d_in, h_in, #bytes, cudaMemcpyHostToDevice);
  ```

- **Kernel launch**
  
  ```c
  kernel<<< #blocks, #threads >>>(args);
  ```

- **Memory deallocation**
  
  ```c
  cudaFree(d_in);
  ```

- **Explicit synchronization**
  
  ```c
  cudaDeviceSynchronize();
  ```
Host Code Example: Vector Addition

```c
void vecadd(float* A, float* B, float* C, int N) {

    // Allocate GPU memory
    float *A_d, *B_d, *C_d;
    cudaMalloc((void**) &A_d, N*sizeof(float));
    cudaMalloc((void**) &B_d, N*sizeof(float));
    cudaMalloc((void**) &C_d, N*sizeof(float));

    // Copy data to GPU memory
    cudaMemcpy(A_d, A, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform computation on GPU
    ...

    // Copy data from GPU memory
    cudaMemcpy(C, C_d, N*sizeof(float), cudaMemcpyDeviceToHost);

    // Deallocate GPU memory
    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```
Vector Addition (I)

- Our first GPU programming example
- We assign one GPU thread to each element-wise addition
Vector Addition (II)

- The whole set of threads is called a grid
- We need a way to assign threads to GPU cores
We group threads into blocks
Hardware is free to schedule thread blocks

Each block can execute in any order relative to other blocks.
Launching a Grid

- Threads in the same grid execute the same function known as a kernel

- A grid can be launched by calling a kernel and configuring it with appropriate grid and block sizes

```c
const unsigned int numThreadsPerBlock = 512;
const unsigned int numBlocks = N/numThreadsPerBlock;

vecadd_kernel<<<numBlocks, numThreadsPerBlock>>>(A_d, B_d, C_d, N);
```
Host Code Example: Vector Addition

```c
void vecadd(float* A, float* B, float* C, int N) {

    // Allocate GPU memory
    float *A_d, *B_d, *C_d;
    cudaMalloc((void**)&A_d, N*sizeof(float));
    cudaMalloc((void**)&B_d, N*sizeof(float));
    cudaMalloc((void**)&C_d, N*sizeof(float));

    // Copy data to GPU memory
    cudaMemcpy(A_d, A, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform computation on GPU
    const unsigned int numThreadsPerBlock = 512;
    const unsigned int numBlocks = N/numThreadsPerBlock;

    vecadd_kernel<<<numBlocks, numThreadsPerBlock>>>(A_d, B_d, C_d, N);

    // Copy data from GPU memory
    cudaMemcpy(C, C_d, N*sizeof(float), cudaMemcpyDeviceToHost);

    // Deallocate GPU memory
    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```
Sample GPU SIMT Code (Simplified)

CPU code

```c
for (ii = 0; ii < 100000; ++ii) {
}
```

CUDA code

```c
// there are 100000 threads
__global__ void KernelFunction(...) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int varA = aa[tid];
    int varB = bb[tid];
    C[tid] = varA + varB;
}
```
Vector Addition Kernel

- It is preceded by the keyword `__global__` to indicate that it is a GPU kernel.

- It uses special keywords to distinguish different threads from each other:
  - Block index (`blockIdx.x`), block size (`blockDim.x`), thread index (`threadIdx.x`)

```c
__global__ void vecadd_kernel(float* A, float* B, float* C, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```
Boundary Conditions

- What if the size of the input is not a multiple of the number of threads per block?
  - Solution: use the ceiling to launch extra threads then omit the threads after the boundary

```c
const unsigned int numBlocks = (N + numThreadsPerBlock - 1)/numThreadsPerBlock;

_kernel__ void vecadd_kernel(float* A, float* B, float* C, int N) {
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if(i < N) {
        C[i] = A[i] + B[i];
    }
}
```
Compilation

NVIDIA CUDA Compiler (NVCC)

Host C/C++ Code

Device Just-in-Time Compiler

PTX (Virtual) ISA code

CPU

GPU

Slide credit: Izzat El Hajj
Indexing and Memory Access

- **Images are 2D data structures**
  - height \times width
  - \(\text{Image}[j][i]\), where \(0 \leq j < \text{height}\), and \(0 \leq i < \text{width}\)

![Diagram of 2D image indexing](image.png)
Image Layout in Memory

- **Row-major layout**
- $\text{Image}[j][i] = \text{Image}[j \times \text{width} + i]$

Stride = width

- $\text{Image}[0][1] = \text{Image}[0 \times 8 + 1]$
- $\text{Image}[1][2] = \text{Image}[1 \times 8 + 2]$
Indexing and Memory Access: 1D Grid

- One GPU thread per pixel
- Grid of Blocks of Threads
  - `gridDim.x, blockDim.x`
  - `blockIdx.x, threadIdx.x`

```
blockIdx.x
threadIdx.x
```

6 * 4 + 1 = 25

```
blockIdx.x * blockDim.x + threadIdx.x
```
Indexing and Memory Access: 2D Grid

- **2D blocks**
  - `gridDim.x, gridDim.y`

Row = `blockIdx.y * blockDim.y + threadIdx.y`
Row = 1 * 2 + 1 = 3

Col = `blockIdx.x * blockDim.x + threadIdx.x`
Col = 0 * 2 + 1 = 1

Image[3][1] = Image[3 * 8 + 1]
Recall: From Blocks to Warps

- **GPU cores:** SIMD pipelines
  - Streaming Multiprocessors (SM)
  - Streaming Processors (SP)

- **Blocks are divided into warps**
  - SIMD unit (32 threads)

---

![Diagram of NVIDIA Fermi architecture showing streaming multiprocessors (SM) and streaming processors (SP)].

- Blocks are divided into warps, with each warp containing 32 threads.
- Each warp is scheduled and executed by the streaming multiprocessors (SM) and streaming processors (SP).
- The architecture includes a register file, instruction cache, dispatch unit, and warp scheduler.
- Shared memory/L1 cache and constant cache are also part of the architecture.

NVIDIA Fermi architecture
Recommended Readings (I)

  - Chapter 1: Introduction
  - Chapter 2: Data parallel computing
Recommended Readings (II)

  - Chapter 1 - Introduction
  - Chapter 2 - Heterogeneous data parallel computing
  - Chapter 3 - Multidimensional grids and data
Memory Hierarchy
P&S Heterogeneous Systems

GPU Software Hierarchy:
Grids, Blocks, Threads

Dr. Juan Gómez Luna
Prof. Onur Mutlu

ETH Zürich
Fall 2022
17 October 2022