Agenda for Today

- GPU as an accelerator
  - Program structure
    - Bulk synchronous programming model
  - Memory hierarchy and memory management

- Performance considerations
  - Memory access
  - SIMD utilization
  - Atomic operations
  - Data transfers
Recommended Readings

- CUDA Programming Guide

An Example GPU
NVIDIA GeForce GTX 285

- NVIDIA-speak:
  - 240 stream processors
  - “SIMT execution”

- Generic speak:
  - 30 cores
  - 8 SIMD functional units per core
NVIDIA GeForce GTX 285 “core” (I)

- SIMD functional unit, control shared across 8 units
- Multiply-add
- Multiply
- Instruction stream decode
- Execution context storage

64 KB of storage for thread contexts (registers)

Slide credit: Kayvon Fatahalian
NVIDIA GeForce GTX 285 “core” (II)

- Groups of 32 threads share instruction stream (each group is a Warp): they execute the same instruction on different data.
- Up to 32 warps are interleaved in an FGMT manner.
- Up to 1024 thread contexts can be stored.

64 KB of storage for thread contexts (registers)

Slide credit: Kayvon Fatahalian
NVIDIA GeForce GTX 285

30 cores on the GTX 285: 30,720 threads

Slide credit: Kayvon Fatahalian
Evolution of NVIDIA GPUs

Evolution of NVIDIA GPUs

- GTX 285 (2009)
- GTX 480 (2010)
- GTX 780 (2013)
- GTX 980 (2014)
- P100 (2016)
- V100 (2017)

GFLOPS

Stream Processors

#Stream Processors

0 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 11000 12000 13000 14000 15000 16000

0 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 11000 12000 13000 14000 15000 16000
NVIDIA V100

NVIDIA-speak:
- 5120 stream processors
- “SIMT execution”

Generic speak:
- 80 cores
- 64 SIMD functional units per core

- Specialized Functional Units for Machine Learning (tensor ”cores” in NVIDIA-speak)
NVIDIA V100 Block Diagram

80 cores on the V100

https://devblogs.nvidia.com/inside-volta/
NVIDIA V100 Core

15.7 TFLOPS Single Precision
7.8 TFLOPS Double Precision
125 TFLOPS for Deep Learning (Tensor “cores”)

https://devblogs.nvidia.com/inside-volta/
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 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

- **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, HBM2)
    - Data layout
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)
Hardware is **free to schedule** thread blocks.

Each block can execute in any order relative to other blocks.
Memory Hierarchy

Grid (Device)

Block (0, 0)

Block (1, 0)

Shared memory

Registers

Thread (0, 0)

Thread (1, 0)

Shared memory

Registers

Thread (0, 0)

Thread (1, 0)

Global / Texture & Surface memory

Constant memory

Host
Traditional Program Structure in CUDA

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

- `main()`
  
  1. **Allocate memory** space on the device – `cudaMalloc(&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();

---

Slide credit: Hwu & Kirk
CUDA Programming Language

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

- Memory copy
  ```
cudadMemcpy(d_in, h_in, #bytes, cudaMemcpyHostToDevice);
  ```

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

- Memory deallocation
  ```
cudadFree(d_in);
  ```

- Explicit synchronization
  ```
cudadDeviceSynchronize();
  ```
Images are 2D data structures
- height x width
- Image[j][i], where 0 ≤ j < height, and 0 ≤ i < width
Image Layout in Memory

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

\[ \text{Stride} = \text{width} \]

\( \text{Image}[0][1] = \text{Image}[0 \times 8 + 1] \)

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

- One GPU thread per pixel
- Grid of Blocks of Threads
  - $\text{gridDim}.x, \text{blockDim}.x$
  - $\text{blockIdx}.x, \text{threadIdx}.x$

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

```
6 \times 4 + 1 = 25
```

```
\text{blockIdx}.x \times \text{blockDim}.x + \text{threadIdx}.x
```
Indexing and Memory Access: 2D Grid

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

  **Row:**
  \[
  \text{Row} = \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y}
  \]
  \[
  \text{Row} = 1 \times 2 + 1 = 3
  \]

  **Col:**
  \[
  \text{Col} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}
  \]
  \[
  \text{Col} = 0 \times 2 + 1 = 1
  \]

  **Image[3][1] = Image[3 \times 8 + 1]**
Brief Review of GPU Architecture (I)

- Streaming Processor Array
  - Tesla architecture (G80/GT200)
Brief Review of GPU Architecture (II)

- **Streaming Multiprocessors (SM)**
  - Streaming Processors (SP)

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

---

**NVIDIA Fermi architecture**

<table>
<thead>
<tr>
<th>Streaming Multiprocessor</th>
<th>Instruction Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Warp Scheduler</td>
</tr>
<tr>
<td></td>
<td>Warp Scheduler</td>
</tr>
<tr>
<td>Dispatch Unit</td>
<td>Dispatch Unit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register File</th>
<th>LD/ST</th>
<th>SFU</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>LD/ST</td>
<td>SFU</td>
</tr>
<tr>
<td>SP</td>
<td>LD/ST</td>
<td>SFU</td>
</tr>
<tr>
<td>SP</td>
<td>LD/ST</td>
<td>SFU</td>
</tr>
<tr>
<td>SP</td>
<td>LD/ST</td>
<td>SFU</td>
</tr>
<tr>
<td>SP</td>
<td>LD/ST</td>
<td>SFU</td>
</tr>
<tr>
<td>SP</td>
<td>LD/ST</td>
<td>SFU</td>
</tr>
<tr>
<td>SP</td>
<td>LD/ST</td>
<td>SFU</td>
</tr>
<tr>
<td>SP</td>
<td>LD/ST</td>
<td>SFU</td>
</tr>
<tr>
<td>SP</td>
<td>LD/ST</td>
<td>SFU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shared Memory / L1 Cache</th>
<th>LD/ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Cache</td>
<td></td>
</tr>
</tbody>
</table>
Brief Review of GPU Architecture (III)

- **Streaming Multiprocessors (SM) or Compute Units (CU)**
  - SIMD pipelines

- **Streaming Processors (SP) or CUDA “cores”**
  - Vector lanes

- **Number of SMs x SPs across generations**
  - Tesla (2007): 30 x 8
  - Fermi (2010): 16 x 32
  - Kepler (2012): 15 x 192
  - Maxwell (2014): 24 x 128
  - Pascal (2016): 56 x 64
  - Volta (2017): 80 x 64
Performance Considerations
Performance Considerations

- Main bottlenecks
  - Global memory access
  - CPU-GPU data transfers

- Memory access
  - Latency hiding
    - Occupancy
  - Memory coalescing
  - Data reuse
    - Shared memory usage

- SIMD (Warp) Utilization: Divergence

- Atomic operations: Serialization

- Data transfers between CPU and GPU
  - Overlap of communication and computation
Memory Access
Latency Hiding

- **FGMT** can hide **long latency operations** (e.g., memory accesses)
- **Occupancy**: ratio of active warps

![Diagram showing warps and instructions with latency hiding examples]
Occupancy

- **SM resources (typical values)**
  - Maximum number of warps per SM (64)
  - Maximum number of blocks per SM (32)
  - Register usage (256KB)
  - Shared memory usage (64KB)

- **Occupancy calculation**
  - Number of threads per block (defined by the programmer)
  - Registers per thread (known at compile time)
  - Shared memory per block (defined by the programmer)
Memory Coalescing

- When accessing global memory, we want to make sure that concurrent threads access nearby memory locations.
- **Peak bandwidth** utilization occurs when all threads in a warp access one cache line.

---

**Slide credit:** Hwu & Kirk
Uncoalesced Memory Accesses

Access direction in Kernel code

Time Period 1

Time Period 2

...
Coalesced Memory Accesses

Access direction in Kernel code

Time Period 1
T₁ T₂ T₃ T₄

Time Period 2
T₁ T₂ T₃ T₄

...
Array of Structures vs. Structure of Arrays

La granularidad en ASTA, es decir, el ancho del tile, estará relacionado con la granularidad de acceso a la memoria (warp_size = 32, por ejemplo).

Convertir entre los distintos layouts, en realidad es un tipo de transposición. Por ejemplo, AoS a ASTA:

Transposición es un tipo de permutación. Una permutación se puede descomponer en ciclos independientes de desplazamientos.

```
struct foo{
    float a[8];
    float b[8];
    float c[8];
    int d[8];
} A;
```

```
struct foo{
    float a;
    float b;
    float c;
    int d;
} A[8];
```
CPUs Prefer AoS, GPUs Prefer SoA

- Linear and strided accesses

**Throughput (GB/s)**

**Stride (Structure size)**

**Throughput (GB/s)**

**Stride (Structure size)**

AMD Kaveri A10-7850K

Sung+, “DL: A data layout transformation system for heterogeneous computing,” INPAR 2012
Data Reuse

- Same memory locations accessed by neighboring threads

```c
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
    }
}
```
Data Reuse: Tiling

- To take advantage of data reuse, we divide the input into tiles that can be loaded into shared memory

```c
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
...
Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
    }
}
```
Shared Memory

- Shared memory is an **interleaved (banked) memory**
  - Each bank can service one address per cycle

- Typically, 32 banks in NVIDIA GPUs
  - Successive 32-bit words are assigned to successive banks
    - Bank = Address % 32

- Bank conflicts are **only possible within a warp**
  - No bank conflicts between different warps
Shared Memory Bank Conflicts (I)

- Bank conflict free

Linear addressing: stride = 1

Random addressing 1:1

Slide credit: Hwu & Kirk
N-way bank conflicts

2-way bank conflict: stride = 2

8-way bank conflict: stride = 8
Reducing Shared Memory Bank Conflicts

- Bank conflicts are only possible within a warp
  - No bank conflicts between different warps

- If strided accesses are needed, some optimization techniques can help
  - Padding
  - Randomized mapping
  - Hash functions
SIMD Utilization
Intra-warp divergence

Compute(threadIdx.x);
if (threadIdx.x % 2 == 0){
   Do_this(threadIdx.x);
}
else{
   Do_that(threadIdx.x);
}
Increasing SIMD Utilization

- **Divergence-free execution**

```
Compute(threadIdx.x);
if (threadIdx.x < 32){
    Do_this(threadIdx.x * 2);
}
else{
    Do_that((threadIdx.x%32)*2+1);
}
```
Vector Reduction: Naïve Mapping (I)

Thread 0  Thread 2  Thread 4  Thread 6  Thread 8  Thread 10

0  1  2  3  4  5  6  7  8  9  10  11
0+1  2+3  4+5  6+7  8+9  10+11
0...3  4..7  8..11  8..15
0..7  4..7  8..11

Slide credit: Hwu & Kirk
Vector Reduction: Naïve Mapping (II)

- Program with low SIMD utilization

```c
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for (int stride = 1; stride < blockDim.x; stride *= 2) {
    __syncthreads();

    if (t % (2*stride) == 0)
        partialSum[t] += partialSum[t + stride];
}
```
Divergence-Free Mapping (I)

- All active threads belong to the same warp
Divergence-Free Mapping (II)

- Program with high SIMD utilization

```c
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for (int stride = blockDim.x; stride > 1; stride >>= 1){

    __syncthreads();

    if (t < stride)
        partialSum[t] += partialSum[t + stride];

}
```
Atomic Operations
Shared Memory Atomic Operations

- Atomic Operations are needed when threads might update the same memory locations at the same time.

- CUDA: int atomicAdd(int*, int);
- PTX: atom.shared.add.u32 %r25, [%rd14], %r24;
- SASS:

  **Tesla, Fermi, Kepler**
  
  /*00a0*/ LDSLK P0, R9, [R8];
  /*00a8*/ @P0 IADD R10, R9, R7;
  /*00b0*/ @P0 STSCUL P1, [R8], R10;
  /*00b8*/ @!P1 BRA 0xa0;

  **Maxwell, Pascal, Volta**
  
  /*01f8*/ ATOMS.ADD RZ, [R7], R11;
  
  Native atomic operations for 32-bit integer, and 32-bit and 64-bit atomicCAS
We define the intra-warp conflict degree as the number of threads in a warp that update the same memory position. The conflict degree can be between 1 and 32.
**Histogram Calculation**

- Histograms count the number of data instances in disjoint categories (bins)

```plaintext
for (each pixel i in image I){
    Pixel = I[i] // Read pixel
    Pixel' = Computation(Pixel) // Optional computation
    Histogram[Pixel']++ // Vote in histogram bin
}
```

Input data

```
<table>
<thead>
<tr>
<th>data[0]</th>
<th>data[1]</th>
<th>data[2]</th>
<th>...</th>
<th>data[n-1]</th>
</tr>
</thead>
</table>
```

Atomic additions

```
Thread 0
 Thread 1
   Thread 2
    ...  Thread n-1
```

Histogram

```
| 0 | 1 | 2 | ... | B-1 |
```
- Frequent conflicts in natural images
Optimizing Histogram Calculation

- **Privatization**: Per-block sub-histograms in shared memory

Data Transfers between CPU and GPU
Data Transfers

- Synchronous and asynchronous transfers
- Streams (Command queues)
  - **Sequence of operations that are performed in order**
    - CPU-GPU data transfer
    - Kernel execution
      - D input data instances, B blocks
    - GPU-CPU data transfer
  - Default stream

![Diagram of data transfer sequence](image)
Asynchronous Transfers

- **Computation** divided into \( n \text{Streams} \)
  - \( D \) input data instances, \( B \) blocks
  - \( n \text{Streams} \)
    - \( D/n\text{Streams} \) data instances
    - \( B/n\text{Streams} \) blocks

- **Estimates**
  \[
  t_E + \frac{t_T}{n\text{Streams}} \quad \quad t_T + \frac{t_E}{n\text{Streams}}
  \]
  \( t_E \geq t_T \) (dominant kernel) \( t_T > t_E \) (dominant transfers)
Applications with independent computation on different data instances can benefit from asynchronous transfers. For instance, video processing.

![Diagram showing non-streamed and streamed execution of video processing with overlap of communication and computation.](image-url)

Summary

- **GPU as an accelerator**
  - Program structure
    - Bulk synchronous programming model
  - Memory hierarchy and memory management

- **Performance considerations**
  - Memory access
    - Latency hiding: occupancy (TLP)
    - Memory coalescing
    - Data reuse: shared memory
  - SIMD utilization
  - Atomic operations
  - Data transfers
PUMPS+AI Summer School, 2018, July 16-20

The Barcelona Supercomputing Center (BSC) in association with Universitat Politècnica de Catalunya (UPC) has been awarded by NVIDIA as a GPU Center of Excellence. BSC and UPC currently offer a number of courses covering CUDA architecture and programming languages for parallel computing. Please contact us for possible collaborations.

The ninth edition of the Programming and Tuning Massively Parallel Systems + Artificial Intelligence summer school (PUMPS+AI) is aimed at enriching the skills of researchers, graduate students and teachers with cutting-edge technique and hands-on experience in developing applications for many-core processors with massively parallel computing resources like GPU accelerators.

- **Summer School Co-Directors**: Mateo Valero (BSC and UPC) and Wen-mei Hwu (University of Illinois at Urbana-Champaign)
- **Local Organizers**: Antonio J. Peña (responsible, BSC and UPC), and Pau Farre (BSC)
- **Dates**: Applications due: May 31, 2018