Lecture 5: GPU Programming

CSE599W: Spring 2018
Typical Deep Learning System Stack

- **High level Packages**
  - Programming API
  - Gradient Calculation (Differentiation API)

- **System Components**
  - Computational Graph Optimization and Execution
  - Runtime Parallel Scheduling

- **Architecture**
  - GPU Kernels, Optimizing Device Code
  - Accelerators and Hardwares
Typical Deep Learning System Stack

**High level Packages**
- Programming API
- Gradient Calculation (Differentiation API)

**Computational Graph Optimization and Execution**
- Computational Graph Optimization and Execution
- Runtime Parallel Scheduling

**Architecture**
- GPU Kernels, Optimizing Device Code
- Accelerators and Hardwares
Overview

- GPU architecture
- CUDA programming model
- Case study of efficient GPU kernels
CPU vs GPU
CPU vs GPU
CPU vs GPU

Too much overhead in compute resources and energy efficiency.
CPU vs GPU

Vector operations (SSE / AVX)
CPU vs GPU

- **CPU**:
  - Input
  - ALU
  - Decode
  - Fetch
  - Write back

- **GPU**:
  - Specialized accelerator
  - ALU
  - Decode
  - Fetch
  - Write back

Vector operations (SSE / AVX)
Streaming Multiprocessor (SM)

- Decode and schedule the next instructions
- Registers
- SP float core
- DP float core
- Load/store memory
- Special function unit
- Multiple caches
GPU Architecture
Theoretical peak FLOPS comparison

Memory Hierarchy

CPU memory hierarchy

- Core
  - Reg
  - L1 cache
  - L2 cache
Memory Hierarchy

CPU memory hierarchy

Core
- Reg
- L1 cache
- L2 cache

Core
- Reg
- L1 cache
- L2 cache

L3 cache

DRAM
Memory Hierarchy

CPU memory hierarchy

- Core
  - Reg
  - L1 cache
  - L2 cache

- L3 cache

DRAM

GPU memory hierarchy

- SM
  - Reg
  - L1 cache
  - Shared memory
  - Read-only cache

- L2 cache

GPU DRAM
Memory Hierarchy

**CPU memory hierarchy**
- Intel Xeon E7-8870v4
  - Cores: 20
  - Reg / core: ??
  - L1 / core: 32KB
  - L2 / core: 256KB
  - L3 cache: 50MB
  - DRAM: 100s GB
  - Price: $12,000

**GPU memory hierarchy**
- Titan X Pascal
  - SMs: 28
  - Cores / SM: 128
  - Reg / SM: 256 KB
  - L1 / SM: 48 KB
  - Sharedmem / SM: 64 KB
  - L2 cache: 3 MB
  - GPU DRAM: 12 GB
  - Price: $1,200
Memory Hierarchy

CPU memory hierarchy:
- Intel Xeon E7-8870v4
  - Cores: 20
  - Reg / core: ??
- L1 / core: 32KB
- L2 / core: 256KB
- L3 cache: 50MB
- DRAM: 100s GB
- Price: $12,000

GPU memory hierarchy:
- Titan X Pascal
  - SMs: 28
  - Cores / SM: 128
  - Reg / SM: 256 KB
- L1 / SM: 48 KB
- Sharedmem / SM: 64 KB
- L2 cache: 3 MB
- GPU DRAM: 12 GB
- Price: $1,200

More registers than L1 cache
Memory Hierarchy

CPU memory hierarchy

- Core
  - Reg
- L1 cache
- L2 cache
- L3 cache
- DRAM

Intel Xeon E7-8870v4
- Cores: 20
- Reg / core: ??
- L1 / core: 32KB
- L2 / core: 256KB
- L3 cache: 50MB
- DRAM: 100s GB
- Price: $12,000

Titan X Pascal
- SMs: 28
- Cores / SM: 128
- Reg / SM: 256 KB
- L1 / SM: 48 KB
- Sharedmem / SM: 64 KB

GPU memory hierarchy

- SM
  - Reg
  - L1 cache
  - Shared memory
  - Read-only cache
- L2 cache
- GPU DRAM

Intel Xeon E7-8870v4
- Cores: 20
- Reg / core: ??
- L1 / core: 32KB
- L2 / core: 256KB
- L3 cache: 50MB
- DRAM: 100s GB
- Price: $12,000

Titan X Pascal
- SMs: 28
- Cores / SM: 128
- Reg / SM: 256 KB
- L1 / SM: 48 KB
- Sharedmem / SM: 64 KB

Price: $1,200

GPU DRAM: 12 GB
Price: $1,200

L1 cache controlled by programmer
GPU Memory Latency

Registers: R 0 cycle / R-after-W ~20 cycles

L1/texture cache: 92 cycles
Shared memory: 28 cycles
Constant L1 cache: 28 cycles
L2 cache: 200 cycles

DRAM: 350 cycles

(for Nvidia Maxwell architecture)

Memory bandwidth comparison

The diagram shows the theoretical peak GB/s for NVIDIA GPUs and Intel CPUs from 2003 to 2015. The NVIDIA GPU line shows a steady increase, whereas the Intel CPU line shows a more erratic but also increasing trend. The data source is from the GitHub repository of the Oxford CS Deep NLP lectures:

## Nvidia GPU Comparison

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture</strong></td>
<td>Kepler GK110</td>
<td>Maxwell GM200</td>
<td>Pascal GP102</td>
</tr>
<tr>
<td><strong>Number of SMs</strong></td>
<td>15</td>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td><strong>CUDA cores</strong></td>
<td>2880 (192 * 15SM)</td>
<td>3072 (128 * 24SM)</td>
<td>3584 (128 * 28SM)</td>
</tr>
<tr>
<td><strong>Max clock rate</strong></td>
<td>875 MHz</td>
<td>1177 MHz</td>
<td>1531 MHz</td>
</tr>
<tr>
<td><strong>FP32 GFLOPS</strong></td>
<td>5040</td>
<td>7230</td>
<td>10970</td>
</tr>
<tr>
<td><strong>32-bit Registers / SM</strong></td>
<td>64K (256KB)</td>
<td>64K (256KB)</td>
<td>64K (256KB)</td>
</tr>
<tr>
<td><strong>Shared Memory / SM</strong></td>
<td>16 KB / 48 KB</td>
<td>96 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td><strong>L2 Cache / SM</strong></td>
<td>1.5 MB</td>
<td>3 MB</td>
<td>3 MB</td>
</tr>
<tr>
<td><strong>Global DRAM</strong></td>
<td>12 GB</td>
<td>12 GB</td>
<td>12 GB</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>235 W</td>
<td>250 W</td>
<td>250 W</td>
</tr>
</tbody>
</table>
CUDA Programming Model
Programming model: SIMT

- **SIMT**: Single Instruction, Multiple Threads
- Programmer writes code for a single thread in simple C program.
  - All threads execute the same code, but can take different paths.
Programming model: SIMT

- **SIMT**: Single Instruction, Multiple Threads
  - Programmer writes code for a single thread in simple C program.
    - All threads execute the same code, but can take different paths.
  - Threads are grouped into a block.
    - Threads within the same block can synchronize execution.
Programming model: SIMT

- **SIMT**: Single Instruction, Multiple Threads
- Programmer writes code for a single thread in simple C program.
  - All threads execute the same code, but can take different paths.
- Threads are grouped into a block.
  - Threads within the same block can synchronize execution.
- Blocks are grouped into a grid.
  - Blocks are independently scheduled on the GPU, can be executed in any order.
- A kernel is executed as a grid of blocks of threads.
Kernel Execution

- Each block is executed by one SM and does not migrate.
- Several concurrent blocks can reside on one SM depending on block’s memory requirement and the SM’s memory resources.
Kernel Execution

- A warp consists of 32 threads
  - A warp is the basic schedule unit in kernel execution.
- A thread block consists of 32-thread warps.
- Each cycle, a warp scheduler selects one ready warps and dispatches the warps to CUDA cores to execute.
Control flow

100: ...
101: if (condition) {
102:   ...
103: } else {
104:   ...
105: }

warp
Control flow

100: ...
101: if (condition) {
102:    ...
103: } else {
104:    ...
105: }

warp

pc: 100  pc: 101

time
Control flow

100: ... 
101: if (condition) {
102:    ...
103: } else {
104:    ...
105: }

warp

cpc: 100  pc: 101  pc: 102  time
Control flow

100: ...
101: if (condition) {
102:     ...
103: } else {
104:     ...
105: }

[Diagram of control flow with warp and time progression]
Control flow

100: ...
101: if (condition) {
102:     ...
103: } else {
104:     ...
105: }

warp

pc: 100  pc: 101  pc: 102  pc: 104  pc: 105

Time
Thread Hierarchy & Memory Hierarchy

- Thread
- Thread block
- Block 0, Block 1, Block 2, Block 3
- Grid
- Registers & Local memory
- Shared memory
- Global memory

GPU memory hierarchy:
- SM
- Reg
- L1 cache
- Shared memory
- Read-only cache
- L2 cache
- GPU DRAM
Example: Vector Add

// compute vector sum $C = A + B$
Void vecAdd_cpu(const float* A, const float* B, float* C, int n) {
    for (int i = 0; i < n; ++i)
        C[i] = A[i] + B[i];
}
Example: Vector Add

// compute vector sum C = A + B
Void vecAdd_cpu(const float* A, const float* B, float* C, int n) {
    for (int i = 0; i < n; ++i)
        C[i] = A[i] + B[i];
}

__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
Example: Vector Add

```c
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Suppose each block only includes 4 threads:
blockDim.x = 4
Example: Vector Add

```c
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Suppose each block only includes 4 threads: blockDim.x = 4

<table>
<thead>
<tr>
<th>global index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>threadIdx.x</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>blockIdx.x</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each thread only performs one pair-wise addition.
#define THREADS_PER_BLOCK   512

void vecAdd(const float* A, const float* B, float* C, int n) {
    float *d_A, *d_B, *d_C;
    int size = n * sizeof(float);
    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);
    cudaMemcpy(d_C, C, size, cudaMemcpyDeviceToHost);
    int nblocks = (n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
    vecAddKernel<<<nblocks, THREADS_PER_BLOCK>>>(d_A, d_B, d_C, n);
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
Example: Vector Add (Host)

```c
#define THREADS_PER_BLOCK  512
void vecAdd(const float* A, const float* B, float* C, int n) {
    float *d_A, *d_B, *d_C;
    int size = n * sizeof(float);
    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, C, size, cudaMemcpyHostToDevice);
    int nblocks = (n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
    vecAddKernel<<<nblocks, THREADS_PER_BLOCK>>>(d_A, d_B, d_C, n);
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Launch the GPU kernel asynchronously
Example: Sliding Window Sum

- Consider computing the sum of a sliding window over a vector
  - Each output element is the sum of input elements within a radius
  - Example: image blur kernel
- If radius is 3, each output element is sum of 7 input elements
A naive implementation

```c
#define RADIUS 3
__global__ void windowSumNaiveKernel(const float* A, float* B, int n) {
    int out_index = blockDim.x * blockIdx.x + threadIdx.x;
    int in_index = out_index + RADIUS;
    if (out_index < n) {
        float sum = 0.;
        for (int i = -RADIUS; i <= RADIUS; ++i) {
            sum += A[in_index + i];
        }
        B[out_index] = sum;
    }
}
```
A naive implementation

```c
void windowSum(const float* A, float* B, int n) {
    float *d_A, *d_B;
    int size = n * sizeof(float);
    cudaMalloc((void **) &d_A, (n + 2 * RADIUS) * sizeof(float));
    cudaMemset(d_A, 0, (n + 2 * RADIUS) * sizeof(float));
    cudaMemcpy(d_A + RADIUS, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    dim3 threads(THREADS_PER_BLOCK, 1, 1);
    dim3 blocks((n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK, 1, 1);
    windowSumNaiveKernel<<<blocks, threads>>>(d_A, d_B, n);
    cudaMemcpy(B, d_B, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B);
}
```
How to improve it?

- For each element in the input, how many times it is loaded?
How to improve it?

- For each element in the input, how many times it is read?
  - Each input element is read 7 times!
  - Neighboring threads read most of the same elements

- How can we avoid redundant reading of data?
Sharing data between threads within a block

- A thread block first cooperatively loads the needed input data into the shared memory.

```
input

output

Computed by block 1
```
Kernel with shared memory

```c
__global__ void windowSumKernel(const float* A, float* B, int n) {
    __shared__ float temp[THREADS_PER_BLOCK + 2 * RADIUS];
    int out_index = blockDim.x * blockIdx.x + threadIdx.x;
    int in_index = out_index + RADIUS;
    int local_index = threadIdx.x + RADIUS;
    if (out_index < n) {
        temp[local_index] = A[in_index];
        if (threadIdx.x < RADIUS) {
            temp[local_index - RADIUS] = A[in_index - RADIUS];
            temp[local_index + THREADS_PER_BLOCK] = A[in_index+THREADS_PER_BLOCK];
        }
    }
    __syncthreads();
}
```
Kernel with shared memory

```c
float sum = 0.;
for (int i = -RADIUS; i <= RADIUS; ++i) {
    sum += temp[local_index + i];
}
B[out_index] = sum;
```
Performance comparison

Demo!

Code:
https://github.com/dlsys-course/examples/blob/master/cuda/window_sum.cu
Case study of efficient GPU kernels
Case study: GEMM

\[ C = A \times B \]
A: MxK matrix
B: KxN matrix
C: MxN matrix

Workload of a thread block
Case study: GEMM

\[ C = A \times B \]
A: MxK matrix
B: KxN matrix
C: MxN matrix

Workload of a thread block
Case study: GEMM

\[ C = A \times B \]
A: MxK matrix
B: KxN matrix
C: MxN matrix

Workload of a thread block
Case study: GEMM

A strip

B strip

C strip

Each thread block computes a b x b area

Suppose each thread block has t * t threads, b_t = b / t

C = A x B
A: MxK matrix
B: KxN matrix
C: MxN matrix

Global memory

Shared memory

Registers

Thread 1

Thread 2
Case study: GEMM pseudocode

block_dim: \langle M / b, N / b \rangle
thread_dim: \langle t, t \rangle

// thread function
__global__ void SGEMM(float *A, float *B, float *C, int b, int s) {
__shared__ float sA[2][b][s], sB[2][s][b]; // shared by a thread block
float rC[b][b] = {0};                     // thread local buffer, in the registers

Cooperative fetch first strip from A, B to sA[0], sB[0]
__sync_threads();
for (k = 0; k < K / s; k += 1) {
    Cooperative fetch next strip from A, B to sA[(k+1)%2], sB[(k+1)%2]
    __sync_threads();
    for (kk = 0; kk < s; kk += 1) {
        for (j = 0; j < b; j += 1) { // unroll loop
            for (i = 0; i < b; i += 1) { // unroll loop
                rC[j][i] += sA[k%2][threadIdx.x*b+j][kk]*sB[k%2][kk][threadIdx.y*b+i];
            }
        }
    }
}
Write rC back to C

Run in parallel
Case study: GEMM

More details see:

Case study: Reduction Sum

Tips for high performance

- Use existing libraries, which are highly optimized, e.g. cublas, cudnn.
- Use nvprof or nvvp (visual profiler) to debug the performance.
- Use high level language to write GPU kernels.
References