Lecture 8: GPU Programming

CSE599G1: Spring 2017
Announcements

● **Project proposal** due on Thursday (4/28) 5pm.

● **Assignment 2** will be out today, due in two weeks.
  ○ Implement GPU kernels and use cublas library
  ○ Infer output shapes and memory planning
Overview

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

CPU

input

output

ALU

Decode

Fetch

output

Write back

input
CPU vs GPU

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

Vector operations (SSE / AVX)
CPU vs GPU

- **CPU**
  - Input
  - Output
  - **Fetch**
  - **Decode**
  - **Write Back**

- **GPU: Specialized Accelerator**
  - **Decode**
    - **ALU**
  - **Vector operations (SSE / AVX)**
    - **ALU**

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

Theoretical single precision GFLOP/s at base clock

- NVIDIA GPU
- Intel CPU

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

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

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
- L2 cache: 3 MB
- GPU DRAM: 12 GB
- Price: $1,200
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
L2 cache: 3 MB
GPU DRAM: 12 GB
Price: $1,200

GPU memory hierarchy

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

More registers than L1 cache

- Paul G. Allen School of Computer Science & Engineering
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

GPU memory hierarchy

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

Theoretical Peak GB/s

- NVIDIA GPU
- Intel CPU

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

- Programmer writes code for a single thread in simple C program.
  - All threads executes 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.
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

```c
// 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`

Each thread only performs one pair-wise addition
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, cudaMemcpyDeviceToHost);
    int nbblocks = (n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
    vecAddKernel<<<nbblocks, 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);
    cudaMalloc((void **) &d_C, size);
    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));
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    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.

![Diagram showing input and output data with computed results by block 1]
__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/lecture8/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 x B
A: MxK matrix
B: KxN matrix
C: MxN matrix

Workload of a thread block
Case study: GEMM

Each thread block computes a $b \times b$ area

Suppose each thread block has $t \times t$ threads, $b_t = \frac{b}{t}$

$C = A \times B$
A: $M \times K$ matrix
B: $K \times N$ matrix
C: $M \times N$ matrix

Global memory

Shared memory

Registers

Thread 1

Thread 2

C strip

B strip

A strip

C tile

Cooperatively loaded by both thread 1 and 2
Case study: GEMM pseudocode

block_dim: <M / b, N / b>
thread_dim: <t, t>

// 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