CS 314 Principles of Programming Languages

Zheng Zhang
Fall 2016

Dec 14 — GPU Programming

Rutgers University
Programming with CUDA

✦ Compute Unified Device Architecture (CUDA)

✦ Mapping and managing computations to GPU
  * make GPU work as a data-parallel computing device.
  * need no explicit mapping from app. to a graphics API
  * largest user base and mature performance tuning experience
  * similar to other general purpose GPU programming framework
CUDA Programming Model

- Terminology
  - Device: the GPU (worker)
    - Capable of executing a large number of threads in parallel
  - Host: the CPU (commander)
    - Send workload to and communicate with GPU
  - Kernel: a function to be run on the device
    - The actual code that runs on GPU
The Example in C

Add vector A and vector B to vector C.

```c
void add_vector (float *A, float *B, float *C, int N) {
    for (int index = 0; index < N; index++)

    // More code...
}

void main ( ) { ...
    add_vector (A1, B1, C1, N1);
    ...
}
Programming with CUDA

The Example in CUDA

Add vector A and vector B to vector C.

```c
__global__ void kernel function to run on GPU
add_vector (float *A, float *B, float *C, in N){....
    ....
}

void main ( ) { main function to run on CPU
    add_vector <<<dimGrid, dimBlock>>>(A1,B1,C1,N1);
    ...
}
```
Review: CUDA Thread View

add_vector <<<dimGrid, dimBlock>>> (A1,B1,C1,N1);

- One or more blocks are assigned to a multiprocessor.

- Threads in a block is organized in warps. (32 threads/warp)

- One (or half) warp runs in a SIMD fashion.
Review: SIMD (Single Instruction Multiple Data)

- \( C[\text{index}] = A[\text{index}] + B[\text{index}] \);

  \begin{align*}
  \text{thread 0:} & \quad C[0] = A[0] + B[0]; \\
  \text{....} & \\
  \end{align*}

Execute at the same time.
Performance Hazard 1: SIMD Divergence

- Eliminate as much control divergence as possible

**control flow** (thread divergence)

```
tid: 0 1 2 3 4 5 6 7
[ ] [ ] [ ] [ ] [ ] [ ] [ ]
  
  if (A[tid]) {...}
```

```
A[ ]: 0 0 6 0 2 4 1
```

```
for (i=0;i<A[tid]; i++) {...}
```

Degrade throughput by up to warp size times.

*(warp size = 32 in modern GPUs)*
CUDA — Memory Model

- **On-chip**
  - Registers, shared memory (scratch-pad), I cache, C cache, texture cache
  - Accessing speed are similar

- **Off-chip (all types share the GDDR memory space)**
  - **Local memory**: register spilling, activation record for function calls
  - **Constant memory**: read through a constant cache per SM, optimized for broadcast reads.
  - **Texture memory**: read through texture cache, optimized for spatially related reads, with dedicated hardware as filters (hardware cache)
CUDA Shared Memory (Scratch-pad Memory)

- Small and Fast: Can also be used as a type of cache — software cache, placing the hot/frequent data objects
- Typically 16 or 32 banks
- Successive 32-bit words are assigned to successive banks (16 bits/cycle per bank)
- A thread accesses strided instead of contiguous array elements

Performance Hazard II: Bank Conflicts
Data is fetched in a chunk. Loading/storing such a chunk is called a transaction. A transaction can be 32 bytes, 64 bytes or 128 bytes.

A thread warp needs to fetch data for all of its threads before it can run.

Make data for one thread warp fit into as few memory transactions as possible is called memory coalescing.
Example: Matrix Transpose
Matrix Transpose Code

```
__global__ void transpose_naive(float *odata, float* idata, int width, int height)
{
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in  = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
```

What is the performance hazard here?
Matrix Transpose Code

```c
__global__ void transpose_naive(float *odata, float* idata, int width, int height)
{
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in  = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;

        odata[index_out] = idata[index_in];
    }
}
```

Simple implementation.

memory write access not coalesced
Example: Matrix Transpose

A thread corresponds to a cell in the matrix
Matrix Transpose Code

```c
__global__ void transpose(float *odata, float *idata, int width, int height){
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];
    // read the matrix tile into shared memory
    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if((xIndex < width) && (yIndex < height))
    {
        unsigned int index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = idata[index_in];
    }
    __syncthreads();
    // write the transposed matrix tile to global memory
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if((xIndex < height) && (yIndex < width)){
        unsigned int index_out = yIndex * height + xIndex;
        odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

Efficient implementation.

memory access coalesced
Example: Matrix Transpose

Efficient implementation.

In global memory -> in shared memory -> in global memory

coalesced read

coalesced write
Parallel Primitives on GPU

- Parallel Reduction
- Parallel Scan (Prefix Sum and Segment Scan)
- Parallel Sort
What is a reduction operation? (think about map reduce)

The sequential code for “sum” of an array

```c
int sum = 0;
for ( int i = 0; i < N; i++ )
    sum = sum + A[i];
```

Parallel Reduction

```
3 1 7 0 4 1 6 3
sum: 0 3 4 11 11 15 16 22 25
```
Parallel Reduction (Summation)

Basic Idea

parallel iteration 1
parallel iteration 2
parallel iteration 3
Parallel Reduction (Summation)

- Initial Implementation

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0)  sdata[tid] += sdata[tid + s];
    __syncthreads();
}
```
Parallel Reduction (Summation)

Initial Implementation

What is the performance hazard here?
Parallel Reduction (Summation)

Initial Implementation

\[ s=1 \]
\[ s=2 \]
\[ s=4 \]
\[ s=8 \]

Values (shared memory)

\[
\begin{array}{cccccccccccccccc}
10 & 1 & 8 & -1 & 0 & -2 & 3 & 5 & -2 & -3 & 2 & 7 & 0 & 11 & 0 & 2 \\
\end{array}
\]

Step 1
Stride 1

\[
\begin{array}{cccccccccccccccc}
11 & 1 & 7 & -1 & -2 & 8 & 5 & -5 & -3 & 9 & 7 & 11 & 11 & 2 & 2 \\
\end{array}
\]

Step 2
Stride 2

\[
\begin{array}{cccccccccccccccc}
18 & 1 & 7 & -1 & 6 & -2 & 8 & 5 & 4 & -3 & 9 & 7 & 13 & 11 & 2 & 2 \\
\end{array}
\]

Step 3
Stride 4

\[
\begin{array}{cccccccccccccccc}
24 & 1 & 7 & -1 & 6 & -2 & 8 & 5 & 17 & -3 & 9 & 7 & 13 & 11 & 2 & 2 \\
\end{array}
\]

Step 4
Stride 8

\[
\begin{array}{cccccccccccccccc}
41 & 1 & 7 & -1 & 6 & -2 & 8 & 5 & 17 & -3 & 9 & 7 & 13 & 11 & 2 & 2 \\
\end{array}
\]

Control Divergence!!!
Parallel Reduction (Summation)

- Removed control divergence

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) sdata[index] += sdata[index + s];
    __syncthreads();
}
```
Parallel Reduction (Summation)

- Removed divergence

What is the performance hazard now?
Potential shared memory bank conflicts ...
Review: CUDA Shared Memory Bank Conflicts

- Typically 16 or 32 banks
- Successive 32-bit words are assigned to successive banks (16 bits/cycle per bank)
- A thread accesses strided instead of contiguous array elements
- Can also be used as a type of cache — software cache, placing the hot/frequent data objects
Parallel Reduction (Summation)

Third version -- sequential addressing

```cpp
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) sdata[tid] += sdata[tid + s];
    __syncthreads();
}
```

Any possible further improvement?
Parallel Reduction (Summation)

Fourth version
reduce number of conditional checks and thread sync for warps

for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
  if (tid < s)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}
if (tid < 32) {
  sdata[tid] += sdata[tid + 32];
  sdata[tid] += sdata[tid + 16];
  sdata[tid] += sdata[tid + 8];
  sdata[tid] += sdata[tid + 4];
  sdata[tid] += sdata[tid + 2];
  sdata[tid] += sdata[tid + 1];
}

Fifth version
halve the threads at the first iteration

unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
Parallel Reduction Summation

Sixth version -- code specialization with template

```c
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
}
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
}
if (blockSize >= 128) {
    if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
}
if (tid < 32) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
```

code in red evaluated in compile-time
Parallel Reduction (Summation)

- There is a 7-th version that further optimizes it!

Read: “Optimization Parallel Reduction in CUDA” by Mark Harris, NVIDIA
A Question for You

Parallel Segment Sum

```
A[ ]
```

```
SegSum[] 1 3 4
```

```
threads
```

```
tid 0 1 2 3 4 5 6 7
```

```
ind[ ] 0 1 1 1 3 3 3 3
```

```
```
Parallel Prefix Sum

Sequential Prefix Sum

```c
int PrefixSum[N];
PrefixSum[0] = 0;
for ( i = 1; i < N; i++ )
    PrefixSum[i] = PrefixSum[i-1] + A[i-1];
```

PrefixSum: [0 3 4 11 11 15 16 22]
Parallel Prefix Sum

Initial parallel implementation

Complexity: $O(n \log(n))$

$n$ is the total number of elements in this array

[ Hillis and Steele 1986 ]
Parallel Prefix Sum

- Work efficient implementation

(A). The Up-Sweep (Reduce) Phase

Complexity: $O(n)$

[Blelloch 1990]
Parallel Prefix Sum

- Work efficient implementation

(B). The Down-Sweep Phase

Complexity: $O(n)$
Parallel Prefix Sum

- Shared memory bank conflicts

```
int ai = offset*(2*thid+1)-1;
int bi = offset*(2*thid+2)-1;
ai += ai / NUM_BANKS;
bi += bi / NUM_BANKS;
temp[bi] += temp[ai]
```

Offset = 2: Address (ai) stride is 4, resulting in 4-way bank conflicts

Offset = 2: Padding addresses every 16 elements removes bank conflicts
A Question for You

- Parallel Segment Scan

Use similar algorithm as parallel prefix sum except that we add condition guards: only add if the target thread has the same key value
Parallel Sort

Bitonic sort

A *bitonic sequence* is defined as a list with no more than one local maximum and no more than one local minimum.

*Binary split:* Divide the bitonic list into two equal halves. Compare-exchange each item on the first half with the corresponding item in the second half.

*Result:* Two bitonic sequences where the numbers in one sequence are all less than the numbers in the other sequence.
Parallel Sort

Bitonic sort: many steps of bitonic split

Complexity: $O(n \log(n)^2)$
Parallel steps: $O(\log(n)^2)$
Parallel Sort

Bitonic sort code in CUDA

```c
__global__ static void bitonicSort(int * values) {
    extern __shared__ int shared[];
    const unsigned int tid = threadIdx.x;
    // Copy input to shared mem.
    shared[tid] = values[tid];
    __syncthreads();
    // Parallel bitonic sort.
    for (unsigned int k = 2; k <= NUM; k *= 2) {
        // Bitonic merge:
        for (unsigned int j = k / 2; j>0; j /= 2) {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if (tid & k) == 0)  {
                    if (shared[tid] > shared[ixj])
                        swap(shared[tid], shared[ixj]);
                } else {
                    if (shared[tid] < shared[ixj])
                        swap(shared[tid], shared[ixj]);
                }
            }
            __syncthreads();
        }
        // Write result.
        values[tid] = shared[tid];
    }
}
```
Parallel Sort

- Merge Sort (can also utilize bitonic split)

Every node may correspond to one processor/core
Parallel Sort

- Radix Sort
  1. Sort for every digit from least significant to most significant bit or the other way around. No order change for the data elements in previous sorted groups during the shuffling phase.
  2. Parallel prefix sum fits nicely in this framework. Every block of threads count the frequency of different digit values and their corresponding location in the same digit value group.

Designing efficient sorting algorithms for manycore GPUs (IPDPS’09)
(1). Designing efficient sorting algorithms for manycore GPUs (IPDPS’09)
   Radix sort and merge sort. Used prefix sum extensively.

(2). Fast parallel GPU-sorting using a hybrid algorithm (JPDC’08).
   Combination of bucket sort and vector merge sort

(3). GPU-ABiSort: optimal parallel sorting on stream architectures (IPDPS’06)
   Complexity: $O((n \log(n))/p)$ on with up to $p=n/\log(n)$ streaming processors
