Jan 29 — GPU Programming II

Rutgers University
Review: 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
Review: 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, in N){......
    for ( index = 0; index < N; index++ )
    .......
}

void main ( ) { ...
    add_vector (A1,B1,C1,N1);
    ...}
```
Add vector A and vector B to vector C.

```c
_global_ void add_vector (float *A, float *B, float *C, in N){
    ...
}
```

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

\[ \text{add\_vector} \lll<\lll \text{dimGrid}, \text{dimBlock}\rrr>\rrr (A1,B1,C1,N1); \]

- One or more **blocks** are assigned to a multiprocessor.

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

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


  thread 0:  C[0] = A[0] + B[0];
  ....

  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
        1 1 1 1 1 1 1 1

if (A[tid]) {...}
```

```
A[i]:   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 Memory View

- r/w registers/thread (compiler)
- r/w local mem/thread (compiler)
- r/w shared mem/block (programmer and/or compiler)
- r/w global mem/grid (programmer and/or compiler)
- r constant mem/grid (programmer and/or compiler)
- r texture mem/grid (programmer and/or compiler)
CUDA Shared Memory (Scratch-pad Memory)

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

Performance Hazard II: Bank Conflicts
CUDA Global Memory (GDDR memory)

- 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.

### Performance Hazard III: Non-coalesced Memory Access

```
... = A[P[tid]];  
P[] = { 0, 5, 2, 3, 4, 6, 7}  
tid:  0  1  2  3  4  5  6  7  
A[]:  
```
Example: Matrix Transpose
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.

What is the performance hazard here?
Matrix Transpose Code

```c
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];
    }
}

memory write access not coalesced
```

Simple implementation.
Example: Matrix Transpose

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

__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**
Application Programming Interface

• Four-fold extensions to C language
  • Function type qualifiers
    • __device__: run on device, callable from device
    • __global__: run on device, callable from host
    • __host__: run on host, callable from host. (default)
  • Variable type qualifiers
    • __constant__: in constant memory
    • __shared__: in shared memory
    • __device__: in device memory (default on global mem.)
  • Execution configuration for __global__ functions
    • __global__ void Func (float* parameter);
    • Func<<<Dg, Db, Ns>>>(param);
  • Built-in variables
    • blockDim, blockIdx, threadIdx
Built-in Vector Types

- char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, float1, float2, float3, float4
  - e.g. float4 pos = make_float4(a, b, c, d);
  - v = pos.x + pos.y + pos.z + pos.w;

- dim3 type
  - e.g. dim3 blockSize(XSIZE, YSIZE);
  - unspecified dimensions are 1.
# The Matrix Transpose Code on Host Side

```c
#define BLOCK_DIM 16
int main(){
    int size_x=32; int size_y=128; int mem_size = size_x*size_y;
    float* h_idata, *h_odata, * d_idata, * d_odata;
    h_idata = (float*) malloc(mem_size); h_odata = (float*) malloc(mem_size);
    cudaMalloc( (void**) &d_idata, mem_size); //allocate on device
    cudaMalloc( (void**) &d_odata, mem_size); //allocate on device
    //copy data from host to device
    cudaMemcpy( d_idata, h_idata, mem_size, cudaMemcpyHostToDevice);
    //set up execution configuration
    dim3 threads(BLOCK_DIM, BLOCK_DIM, 1);
    dim3 grid(size_x / BLOCK_DIM, size_y / BLOCK_DIM, 1);
    //call kernel function
    transpose_naive<<< grid, threads >>>(d_odata, d_idata, size_x, size_y);
    cudaThreadSynchronize();
    printf(“Transpose kernel is done.\n”);
    //copy results from device to host
    cudaMemcpy( h_odata, d_odata, mem_size, cudaMemcpyDeviceToHost);}
```
Compile and Run

- Compiler: nvcc
  - nvcc -c main.cu -o kernel.cu_o -I/usr/local/cuda/include -I...
  - g++ -o foo kernel.cu_o -L/usr/local/cuda/lib -L...

- Run
  - foo arg1 arg2 ...

- Debug
  - CUDA-GDB
  - printf() — the classical approach!
  - nvcc -deviceemu ...
    - generate code to run on emulator

Lots of code samples at:

/ilab/users/zz124/cs516_2015/samples/
20 min break ...
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
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?
Review: SIMD (Single Instruction Multiple Data)

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

- \text{thread 0: } C[0] = A[0] + B[0];
  ...

Execute at the same time.
Parallel Reduction (Summation)

Initial Implementation

$s=1$

$s=2$

$s=4$

$s=8$

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 (16bits/cycle per bank)

- A thread accesses strided instead of contiguous array elements

- Can also be used a type of cache — software cache, placing the hot/frequent data objects
Parallel Reduction (Summation)

- Third version -- sequential addressing

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

```c
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];
}
```
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

tid | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
---|---|---|---|---|---|---|---|---
threads | S | S | S | S | S | S | S | S
ind[ ] | 0 | 1 | 1 | 3 | 3 | 3 | 3 | 3
A[ ] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1
SegSum[ ] | 1 | 3 | 4

The diagram illustrates the parallel segment sum operation for an array `A[]` and an index array `ind[]`. Each thread processes segments of the array `A[]` starting from the index specified in `ind[]`. The `SegSum[]` array stores the sum of the segments processed by each thread.
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

```c
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

Bank

<table>
<thead>
<tr>
<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>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>17</td>
<td>21</td>
<td>25</td>
<td>29</td>
<td>33</td>
<td>37</td>
<td>41</td>
<td>45</td>
<td>49</td>
<td>53</td>
<td>57</td>
<td>61</td>
</tr>
</tbody>
</table>

thid

<table>
<thead>
<tr>
<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>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Offset = 2: Padding addresses every 16 elements removes bank conflicts

Bank

<table>
<thead>
<tr>
<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>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>18</td>
<td>22</td>
<td>26</td>
<td>30</td>
<td>35</td>
<td>39</td>
<td>43</td>
<td>47</td>
<td>52</td>
<td>56</td>
<td>60</td>
<td>64</td>
</tr>
</tbody>
</table>

thid

<table>
<thead>
<tr>
<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>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
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)
Reference

(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


Next Class

Resource Allocation (Compiler Middle-end and Back-end):

* SSA

* Register and Memory Coloring