CSCI-GA.3033-004
Graphics Processing Units (GPUs): Architecture and Programming

Lecture: Parallel Patterns

Most slides of this lecture are from:
• David Kirk/NVIDIA
• Wen-mei W. Hwu

EUniversity of Illinois

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Convolution
Convolution

• An Array operation

Output data element = weighted sum of a collection of neighboring input elements.

• The weights are defined by an input mask array.

• Usually used as filters to transform signals (or pixels or ...) into more desirable form.
Convolution

Mask

\[ \begin{array}{ccccccc}
\hline
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

\[ \begin{array}{cccccccc}
\hline
 & & & & & & 57 & \\
\end{array} \]

\[ \begin{array}{ccccccc}
\hline
3 & 4 & 5 & 4 & 3 \\
\end{array} \]

\[ \begin{array}{ccccccc}
 & & & & & & 3 & 8 & 15 & 16 & 15 \\
\end{array} \]
Convolution can also be 2D.
Convolution
Convolution

__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P, int Mask_Width, int Width) {

    int i = blockIdx.x*blockDim.x + threadIdx.x;

    float Pvalue = 0;
    int N_start_point = i - (Mask_Width/2);
    for (int j = 0; j < Mask_Width; j++) {
        if (N_start_point + j >= 0 && N_start_point + j < Width) {
            Pvalue += N[N_start_point + j]*M[j];
        }
    }
    P[i] = Pvalue;
}

• Thread organized as 1D grid.
• Pvalue allows intermediate values to be accumulated in registers to save DRAM bw.
• We assume ghost values are 0.
• There will be control flow divergence (due to ghost elements).
• Ratio of floating point arithmetic calculation to global memory access is ~ 1.0 → What can we do??
Regarding Mask $M$

- Size of $M$ is typically small.
- The contents of $M$ do not change during execution.
- All threads need to access $M$ and in the same order.

Doesn’t this make $M$ a good candidate for constant memory?
Constant Memory

- Constant memory variables are visible to all thread blocks.
- Constant memory variables cannot be changed during kernel execution.
- The size of constant memory can vary from device to device.
How to Use Constant Memory

• Host code allocates, initializes variables the same way as any other variables that need to be copied to the device

• Use `cudaMemcpyToSymbol(dest, src, size)` to copy the variable into the device memory

• This copy function tells the device that the variable will not be modified by the kernel and can be safely cached.
Mask M and Constant Memory

- In host:
  - ```
    #define MASK_WIDTH 10
    __constant__ float M[MASK_WIDTH]
  ```
  - Allocate and initialize a mask `h_M`
    - `cudaMemcpyToSymbol(M, h_M, MASK_WIDTH * sizeof(float), offset, kind);`

- Kernel functions
  - access constant memory variables as global variables \( \Rightarrow \) no need to pass pointers of these variables to the kernel as parameter.
Question: Isn’t the constant memory also in DRAM? Why is it assumed faster than global memory?

Answer:

• CUDA runtime knows that constant memory variables are not modified.
• It directs the hardware to aggressively cache them during kernel execution.
Reduction Trees
What? And Why?

• Arguably the most widely used parallel computation pattern.

• A commonly used strategy for processing large input data sets
  – There is no required order of processing elements in a data set (associative and commutative)
  – Partition the data set into smaller chunks
  – Have each thread to process a chunk
  – Use a reduction tree to summarize the results from each chunk into the final answer

• Google and Hadoop MapReduce frameworks are examples of this pattern
Reduction enables other techniques

- Reduction is also needed to clean up after some commonly used parallelizing transformations

Example: Privatization
- Multiple threads write into an output location
- Replicate the output location so that each thread has a private output location
- Use a reduction tree to combine the values of private locations into the original output location
What is a reduction computation

• Summarize a set of input values into one value using a “reduction operation”
  – Max
  – Min
  – Sum
  – Product
  – Often with user defined reduction operation function as long as the operation
    • Is associative and commutative
    • Has a well-defined identity value (e.g., 0 for sum)
An efficient sequential reduction algorithm performs $N$ operations in $O(N)$

- Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction

- Scan through the input and perform the reduction operation between the result value and the current input value
A parallel reduction tree algorithm performs $N-1$ Operations in $\log(N)$ steps.
A tournament is a reduction tree with “max” operation.
A Quick Analysis

• For N input values, the reduction tree performs
  – \((1/2)N + (1/4)N + (1/8)N + \ldots (1/N) = (1- (1/N))N = N-1\) operations
  – In \(\log(N)\) steps - 1,000,000 input values take 20 steps
    • Assuming that we have enough execution resources
  – Average Parallelism \((N-1)/\log(N))\)
    • For \(N = 1,000,000\), average parallelism is 50,000
    • However, peak resource requirement is 500,000!

• This is a work-efficient parallel algorithm
  – The amount of work done is comparable to sequential
  – Many parallel algorithms are not work efficient
A Sum Reduction Example

- Parallel implementation:
  - Recursively halve the # of threads, add two values per thread in each step
  - Takes log(n) steps for n elements, requires n/2 threads

- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory is used to hold a partial sum vector
  - Each step brings the partial sum vector closer to the sum
  - The final sum will be in element 0
  - Reduces global memory traffic due to partial sum values
Vector Reduction with Branch Divergence

Data

Thread 0 | Thread 1 | Thread 2 | Thread 3 | Thread 4 | Thread 5
---|---|---|---|---|---
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

0..7 | 8..15

Partial Sum elements

steps

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5

0...3 | 4...7 | 8...11

0...7 | 8...15
Simple Thread Index to Data Mapping

• Each thread is responsible of an even-index location of the partial sum vector
  – locations: 0, 2, 4, 6, ... hold sum of 0+1, 2+3, 4+5, ...

• After each step, half of the threads are no longer needed

• In each step, one of the inputs comes from an increasing distance away
Optimizing Reduction Trees

• Performance factors of a reduction kernel
  – Memory coalescing
  – Control divergence
  – Thread utilization
A Sum Example (review)

Thread 0  Thread 1  Thread 2  Thread 3

Data:

Step 1:

Step 2:

Step 3:

Active Partial Sum elements
The Reduction Steps

for (unsigned int stride = 1;
    stride <= blockDim.x; stride *= 2)
{
    __syncthreads();
    if (t % stride == 0) // t is thread ID
        partialSum[2*t] += partialSum[2*t+stride];
}

Why do we need syncthreads()?
Barrier Synchronization

• __syncthreads() are needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step

• Why not another __syncthread() at the end of the reduction loop?
Back to the Global Picture

• At the end of the kernel execution, thread 0 in each block writes the sum of the block (stored in partialSum[0]) into a vector indexed by the value of blockIdx.x.

• There can be a large number of such sums if the original vector is very large
  – The host code may iterate and launch another kernel

• If there are only a small number of sums, the host can simply transfer the data back and add them together.
Some Observations

• In each iteration, two control flow paths will be sequentially traversed for each warp
  – Threads that perform addition and threads that do not
  – Threads that do not perform addition still consume execution resources

• No more than half of threads will be executing after the first step
  – All odd-index threads are disabled after first step
  – After the 5th step, entire warps in each block will fail the if-condition, poor resource utilization but no divergence.
    • This can go on for a while, up to 5 more steps (1024/32=16= $2^5$), where each active warp only has one productive thread until all warps in a block retire
Thread Index Usage Matters

- In some algorithms, one can shift the index usage to improve the divergence behavior
  - Commutative and associative operators
  - At the end, the performance of many CUDA kernels depends on clever indexing.

- Reduction satisfies this criterion.
A Better Strategy

• Always compact the partial sums into the first locations in the partialSum[] array

• Keep the active threads consecutive
An Example of 16 threads

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 14</th>
<th>Thread 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>...</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>0+16</td>
<td></td>
<td>15+31</td>
</tr>
<tr>
<td>3+16</td>
<td>14+31</td>
<td>15+31</td>
<td>16+31</td>
<td>17+31</td>
</tr>
</tbody>
</table>
A Better Reduction Kernel

for (unsigned int stride = blockDim.x;
     stride >= 1; stride /= 2)
{
    __syncthreads();
    if (t < stride) // t is thread ID
        partialSum[t] +=
        partialSum[t+stride];
}
A Quick Analysis

• For a 1024 thread block
  – No divergence in the first 5 steps
  – 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
  – The final 5 steps will still have divergence
Parallel Algorithm Overhead

__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;

unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];
for (unsigned int stride = blockDim.x;
    stride >= 1;  stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];
for (unsigned int stride = blockDim.x;
    stride >= 1;  stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
Parallel Execution Overhead
Although the number of “operations” is N, each “operation involves much more complex address calculation and intermediate result manipulation.

If the parallel code is executed on a single-thread hardware, it would be significantly slower than the code based on the original sequential algorithm.
Parallel Scan (Prefix Sum)
What? Why?

• Frequently used for parallel work assignment and resource allocation
• A **key primitive** in many parallel algorithms to convert serial computation into parallel computation
  – Based on reduction tree and reverse reduction tree
(Inclusive) Scan (Prefix-Sum) Definition

**Definition:** The scan operation takes a binary associative operator \(\oplus\), and an array of \(n\) elements

\[x_0, x_1, \ldots, x_{n-1}\],

and returns the prefix-sum array

\[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})\].

**Example:** If \(\oplus\) is addition, then the scan operation on the array

\[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3\]

would return

\[3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22 \ 25\]
A Inclusive Scan Application Example

- Assume that we have a 100-inch bread to feed 10
- We know how much each person wants in inches
  - [3 5 2 7 28 4 3 0 8 1]
- How do we cut the bread quickly?
- How much will be left?

- Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.

- Method 2: calculate prefix-sum array
  - [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)
  - You can make 10 cuts in parallel at the above 10 cut points
Typical Applications of Scan

• Scan is a simple and useful parallel building block
  – Convert recurrences from sequential:
    \[
    \text{for}(j=1; j<n; j++) \quad \text{out}[j] = \text{out}[j-1] + f(j);
    \]
  – into parallel:
    \[
    \text{forall}(j) \{ \text{temp}[j] = f(j) \};
    \text{scan(out, temp)};
    \]

• Useful for many parallel algorithms:

  • radix sort
  • quicksort
  • String comparison
  • Lexical analysis
  • Stream compaction
  • Polynomial evaluation
  • Solving recurrences
  • Tree operations
  • Histograms
  • …
Other Applications

- Assigning camp slots
- Assigning farmer market space
- Allocating memory to parallel threads
- Allocating memory buffer to communication channels
- ...

An Inclusive Sequential Scan

Given a sequence \([x_0, x_1, x_2, \ldots]\)

Calculate output \([y_0, y_1, y_2, \ldots]\)

Such that

\[
\begin{align*}
y_0 &= x_0 \\
y_1 &= x_0 + x_1 \\
y_2 &= x_0 + x_1 + x_2 \\
&\quad \ldots \\
Y_i &= Y_{i-1} + x_i
\end{align*}
\]

Using a recursive definition

\[
Y_i = Y_{i-1} + x_i
\]
A Sequential C Implementation

\[ y[0] = x[0]; \]

\[
\text{for } (i = 1; i < \text{Max}_i; i++) \ y[i] = y[i-1] + x[i];
\]

Computationally efficient:
N additions needed for N elements \( \to O(N) \)
A Naïve Inclusive Parallel Scan

- Assign one thread to calculate each y element
- Have every thread to add up all x elements needed for the y element

\[
\begin{align*}
y_0 &= x_0 \\
y_1 &= x_0 + x_1 \\
y_2 &= x_0 + x_1 + x_2
\end{align*}
\]

"Parallel programming is easy as long as you do not care about performance."
Parallel Inclusive Scan using Reduction Trees

• Calculate each output element as the reduction of all previous elements
  – Some reduction partial sums will be shared among the calculation of output elements
  – Based on design by Peter Kogge and Harold Stone at IBM in the 1970s – Kogge-Stone Trees
A Slightly Better Parallel Inclusive Scan Algorithm

1. Load input from global memory into shared memory array $T$

| T | 3 | 1 | 7 | 0 | 4 | 1 | 6 | 3 |

Each thread loads one value from the input (global memory) array into shared memory array $T$. 
A Kogge-Stone Parallel Scan Algorithm

1. (previous slide)

2. Iterate log(n) times, stride from 1 to ceil(n/2.0). Threads from stride to n-1 are active: add pairs of elements that are stride elements apart.

- Active threads: stride to n-1 (n-stride threads)
- Thread j adds elements j and j-stride from T and writes result into shared memory buffer T
- Each iteration requires two syncthreads
  - syncthreads(); // make sure that input is in place
  - float temp = T[j] + T[j - stride];
  - syncthreads(); // make sure that previous output has been consumed
  - T[j] = temp;

Iteration #1
Stride = 1
1. ...

2. Iterate \( \log(n) \) times, stride from 1 to \( \lceil n/2.0 \rceil \). Threads \textit{stride} to \( n-1 \) active: add pairs of elements that are \textit{stride} elements apart.

### Iteration #2

**Stride = 2**

<table>
<thead>
<tr>
<th>T</th>
<th>3</th>
<th>1</th>
<th>7</th>
<th>0</th>
<th>4</th>
<th>1</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>T</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td>14</td>
</tr>
</tbody>
</table>
A Kogge-Stone Parallel Scan Algorithm

1. Load input from global memory to shared memory.

2. Iterate log(n) times, stride from 1 to ceil(n/2.0). Threads \textit{stride} to \( n-1 \) active: add pairs of elements that are \textit{stride} elements apart.

3. Write output from shared memory to device memory.
Enhancement: Double Buffering

- Use two copies of data T0 and T1
- Start by using T0 as input and T1 as output
- Switch input/output roles after each iteration
  - Iteration 0: T0 as input and T1 as output
  - Iteration 1: T1 as input and T0 and output
  - Iteration 2: T0 as input and T1 as output
- This is typically implemented with two pointers, source and destination that swap their contents from one iteration to the next
- This eliminates the need for the second syncthreads
Work Efficiency Analysis

• A Kogge-Stone scan kernel executes \( \log(n) \) parallel iterations
  – The steps do \( (n-1), (n-2), (n-4), \ldots (n-n/2) \) add operations each
  – Total \# of add operations: \( n \times \log(n) - (n-1) \Rightarrow O(n \times \log(n)) \) work

• This scan algorithm is not very work efficient
  – Sequential scan algorithm does \( n \) adds
  – A factor of \( \log(n) \) hurts: 20x for 1,000,000 elements!
  – Typically used within each block, where \( n \leq 1,024 \)

• A parallel algorithm can be slow when execution resources are saturated due to low work efficiency
Improving Efficiency

• A common parallel algorithm pattern: *Balanced Trees*
  – Build a balanced binary tree on the input data and sweep it to and from the root
  – Tree is not an actual data structure, but a concept to determine what each thread does at each step

• For scan:
  1. Traverse down from leaves to root building partial sums at internal nodes in the tree
     • Root holds sum of all leaves
  2. Traverse back up the tree building the scan from the partial sums
Brent-Kung Parallel Scan - Reduction Step

Time

\[ \sum_{x_0 \ldots x_1} \]
\[ \sum_{x_2 \ldots x_3} \]
\[ \sum_{x_4 \ldots x_5} \]
\[ \sum_{x_6 \ldots x_7} \]

In place calculation

Final value after reduce
Inclusive Post Scan Step

\[ x_0 \sum x_0 \ldots x_1 \quad x_2 \quad \sum x_0 \ldots x_3 \quad x_4 \quad \sum x_4 \ldots x_5 \quad x_6 \quad \sum x_0 \ldots x_7 \]

\[ \sum x_0 \ldots x_5 \]
Inclusive Post Scan Step

\[ x_0 + \sum_{x_1}^x + x_2 + \sum_{x_3}^x + x_4 + \sum_{x_5}^x + x_6 + \sum_{x_7}^x \]
Reduction Step Kernel Code

// float T[BLOCK_SIZE] is in shared memory

int stride = 1;
while(stride < BLOCK_SIZE)
{
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < BLOCK_SIZE)
        T[index] += T[index-stride];
    stride = stride*2;

    __syncthreads();
}
Post Scan Step

```c
int stride = BLOCK_SIZE/2;
while(stride > 0)
{
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < BLOCK_SIZE)
    {
        T[index+stride] += T[index];
    }
    stride = stride / 2;
    __syncthreads();
}
```
Work Analysis

- The parallel Inclusive Scan executes $2\times \log(n)$ parallel iterations
  - $\log(n)$ in reduction and $\log(n)$ in post scan
  - The iterations do $n/2$, $n/4$, $1, 1, ..., n/4$. $n/2$ adds
  - Total adds: $2\times (n-1) \rightarrow O(n)$ work

- The total number of adds is no more than twice of that done in the efficient sequential algorithm
  - The benefit of parallelism can easily overcome the $2\times$ work when there is sufficient hardware
A couple of details

- Brent-Kung uses half the number of threads compared to Kogge-Stone
  - Each thread should load two elements into the shared memory
- Brent-Kung takes twice the number of steps compared to Kogge-Stone
  - Kogge-Stone is more popular for parallel scan with blocks in GPUs
Overall Flow of Complete Scan
A Hierarchical Approach

Initial Array of Arbitrary Values

Scan Block 0  Scan Block 1  Scan Block 2  Scan Block 3

Store Block Sum to Auxiliary Array

Scan Block Sums

Add Scanned Block Sum $i$ to All Values of Scanned Block $i + 1$

Final Array of Scanned Values
Using Global Memory Contents in CUDA

• Data in registers and shared memory of one thread block are not visible to other blocks.
• To make data visible, the data has to be written into global memory.
• However, any data written to the global memory are not visible until a memory fence. This is typically done by terminating the kernel execution.
• Launch another kernel to continue the execution. The global memory writes done by the terminated kernels are visible to all thread blocks.
Overall Flow of Complete Scan
A Hierarchical Approach

Initial Array of Arbitrary Values

Scan Block 0
Scan Block 1
Scan Block 2
Scan Block 3

Store Block Sum to Auxiliary Array

Scan Block Sums

Add Scanned Block Sum $i$ to All
Values of Scanned Block $i+1$

Final Array of Scanned Values
**Definition:** The exclusive scan operation takes a binary associative operator $\oplus$, and an array of $n$ elements $[x_0, x_1, \ldots, x_{n-1}]$ and returns the array

$$[0, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2})].$$

**Example:** If $\oplus$ is addition, then the exclusive scan operation on

$$[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]$$

would return

$$[0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22]$$
Why Exclusive Scan

- To find the beginning address of allocated buffers

- Inclusive and Exclusive scans can be easily derived from each other; it is a matter of convenience

\[ \begin{array}{c}
3 & 1 & 7 & 0 & 4 & 1 & 6 & 3 \\
\end{array} \]

Exclusive \[ \begin{array}{c}
0 & 3 & 4 & 11 & 11 & 15 & 16 & 22 \\
\end{array} \]

Inclusive \[ \begin{array}{c}
3 & 4 & 11 & 11 & 15 & 16 & 22 & 25 \\
\end{array} \]
Conclusions

• We have reviewed several useful parallel patterns that you can use in your own GPU programming:
  – Convolution and tiled convolution
  – Reduction trees
  – Prefix scan (inclusive and exclusive)
• Parallel version must be work efficient
• Then we apply different GPU optimizations from our bag of tricks (coalescing, shared memory usage, ...).