GP-GPU and High Performances Computing

Lecture 06 – Patterns

January 2, 2024

Last time

- Patterns
- Avoiding memory conflicts

Objectives

- to learn parallel scan (prefix sum) algorithms based on reductions and reverse reductions
- to learn the concept of double buffering
- to understand tradeoffs between work efficiency and latency
- to learn how to develop hierarchical algorithms (across multiple kernels)

Inclusive scan

Prefix Sum-Scan

- Frequently use for parallel work assignment and ressource allocation.
- A key primitive in numerous parallel algorithms to convert serial computation into parallel computation.
- Fundamental parallel computation pattern.
- Efficient design for data intensive computations.

Prefix scan

Definition 1

The all prefix-sums operation takes a binary associative operator $\oplus$ and an array of n elements

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

and returns the array

$$[x_0, (x_0 \oplus x_1), \dots, (x_0 \oplus x_1 \oplus \dots \oplus x_{n-1})]$$

Example

For $\oplus$ the classical addition between integer, the prefix sum operation on

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

returns

$$[3, 4, 11, 11, 15, 16, 22, 25]$$

.

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

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

$$[3, 8, 10, 17, 45, 49, 52, 52, 60, 61]$$

(39 inches left)

Typical Applications of Scan

- Scan is a simple and useful parallel building block
- Convert recurrences :

  from sequential

```
for(j=1;j<n;j++)
        out[j] = out[j-1] + f(j);
```

  into parallel:

```
forall(j) { temp[j] = f(j) };
scan(out, temp);
```

- Useful for many parallel algorithms:
  - Histograms
  - Reduction and broadcast in O(log n) time
  - Sparse-Matrix-Vector-Multiply (SpMV) using Parallel prefix (scan) in O(log n) time
  - Adding two n-bit integers in O(log n) time
  - Multiplying n-by-n matrices in O(log n) time
  - Inverting n-by-n triangular matrices in O(log2 n) time
  - Inverting n-by-n dense matrices in O(log2 n) time Segmented Scan
  - Parallel page layout in a browser (Leo Meyerovich, Ras Bodik)
  - Solving n-by-n tridiagonal matrices in O(log n) time
  - Traversing linked lists
  - Computing minimal spanning trees
  - Evaluating arbitrary expressions in O(log n) time
  - Computing convex hulls of point sets
  - Evaluating recurrences in O(log n) time
  - 2D parallel prefix, for image segmentation (Catanzaro, Keutzer)

An Inclusive Sequential Addition Scan

Algorithm 1: Inclusive scan

Data: A sequence $[x_0, x_1, x_2, \ldots]$

Result: $[y_0, y_1, y_2, \ldots]$

$y_0 = x_0$

$y_1 = x_0 + x_1$

$y_2 = x_0 + x_1 + x_2$

$\ldots$

Which translates into the recursive definition

$$y_i = y_{i-1} + x_i$$

A Work Efficient C Implementation

```
y[0] = x[0];
for (i = 1; i < Max_i; i++)
y[i] = y [i-1] + x[i];
```

Computationally efficient:

N additions needed for N elements - O(N)!

Only slightly more expensive than sequential reduction.

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

$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$

> **Remarque**
>
> Parallel programming is easy as long as you do not care about performance.

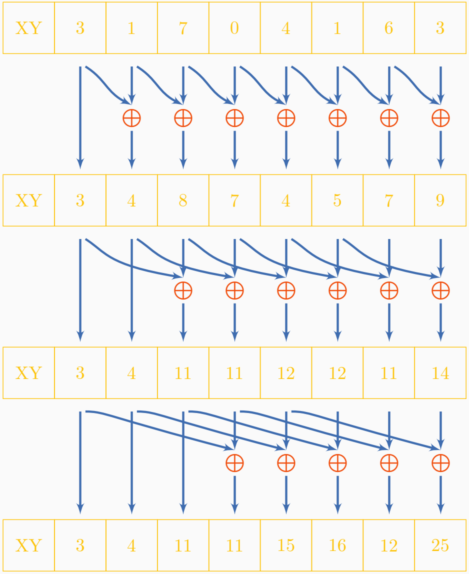
A Better Parallel Scan Algorithm

1. Read input from device global memory to shared memory.
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration.

XY	3	1	7	0	4	1	6	3

   - Active threads stride to n-1 (n-stride threads).
   - Thread j adds elements j and j-stride from shared memory and writes result into element j in shared memory.
   - Requires barrier synchronization, once before read and once before write.
3. Write output from shared memory to device memory.

Scan example



Handling Dependencies

- During every iteration, each thread can overwrite the input of another thread
- Barrier synchronization to ensure all inputs have been properly generated
- All threads secure input operand that can be overwritten by another thread
- Barrier synchronization to ensure that all threads have secured their inputs
- All threads perform Addition and write output

```
__global__ void scan_kernel_v1(float *X, float *Y, int InputSize)
{
  __shared__ float XY[SECTION_SIZE];
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < InputSize) {
  XY[threadIdx.x] = X[i];
}
// the code below performs iterative scan on XY
for (unsigned int stride = 1; stride <= threadIdx.x; stride *= 2)
{
  __syncthreads();
  float in1 = XY[threadIdx.xstride];
  __syncthreads();
  XY[threadIdx.x] += in1;
}
}
```

Work efficiencies considerations

- This **scan** executes $\log(n)$ parallel iterations
  - The steps do $(n-1)$, $(n-2)$, $(n-4)$,...$(n-n/2)$ adds each
  - Total adds: $n\log(n)-(n-1) \rightarrow O(n\log(n))$ work
- This scan algorithm is not work efficient
  - Sequential scan algorithm does n adds
  - A factor of $\log(n)$ can hurt: $10\times$ for 1024 elements!
- A parallel algorithm can be slower than a sequential one when execution resources are saturated from low work efficiency

Improving efficiency

Improving Efficiency

- Balanced Trees
  - Form 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:
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
  - Root holds sum of all leaves
  - Traverse back up the tree building the output from the partial sums

Reduction Phase Kernel Code

```
// XY[2*BLOCK_SIZE] is in shared memory
for (int stride = 1;stride <= BLOCK_SIZE; stride *= 2) {
  int index = (threadIdx.x+1)*stride*2 - 1;
  if(index < 2*BLOCK_SIZE)
  XY[index] += XY[index-stride];
  __syncthreads();
}
```

Post Reduction Reverse Phase Kernel

```
for (int stride = BLOCK_SIZE/2; stride > 0; stride /= 2) {
  __syncthreads();
  int index = (threadIdx.x+1)*stride*2 - 1;
  if(index+stride < 2*BLOCK_SIZE) {
    XY[index + stride] += XY[index];
  }
}
__syncthreads();
if (i < InputSize) Y[i] = XY[threadIdx.x];
```

Work Analysis of the Work Efficient Kernel

- The work efficient kernel executes log(n) parallel iterations in the reduction step
  - The iterations do $n/2, n/4, \ldots 1$ adds
  - Total adds: $(n-1) \to O(n)$ work
- It executes $log(n) - 1$ parallel iterations in the post reduction reverse step
  - The iterations do $2-1$, $4-1$,$\ldots n/2-1$ adds
  - Total adds: $(n-2)$–$(log(n)-1) \to O(n)$ work
- Both phases perform up to no more than 2*(n1) adds
- 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 2X work when there is sufficient hardware

Tradeoffs

- The work efficient scan kernel is normally more desirable
  - Better Energy efficiency
  - Less execution resource requirement
- However, the work inefficient kernel could be better for absolute performance due to its single-step nature if
- There is sufficient execution resource

Exclusive scan

Exclusive scan

Definition 2

The all 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

For $\oplus$ the classical addition between integer, the exclusive scan operation on

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

returns

$$[0, 3, 4, 11, 11, 15, 16, 22]$$

.

Why?

- 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

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

- Exclusive $[0, 3, 4, 11, 11, 15, 16, 22]$
- Inclusive $[3, 4, 11, 11, 15, 16, 22, 25]$

A simple exclusive scan kernel

- Adapt an inclusive, work in-efficient scan kernel
- Block 0:
  - Thread 0 loads 0 into `XY[0]`
  - Other threads load `X[threadIdx.x-1]` into `XY[threadIdx.x]`
- All other blocks:
  - All thread load `X[blockIdx.x*blockDim.x+threadIdx.x-1]` into `XY[threadIdex.x]`
- Similar adaption for work efficient scan kernel but pay attention that each thread loads two elements
  - Only one zero should be loaded
  - All elements should be shifted by only one position

Dealing with large vectors

- Build on the work efficient scan kernel
- Have each section of `2*blockDim.x` elements assigned to a block
- Have each block write the sum of its section into a `Sum[]` array indexed by `blockIdx.x`
- Run the scan kernel on the `Sum[]` array
- Add the scanned `Sum[]` array values to the elements of corresponding sections
- Adaptation of work inefficient kernel is similar.

Conclusion

Conclusions

- Themes of this class
  - Scan memory pattern
  - Introduction to efficiency