Lecture 6 - GP-GPU and High Performances Computing

Parallel Pattern

Previously

- Patterns
- Avoiding memory conflicts

Objectives

- Learn parallel scan (prefix sum) algorithms
- Learn double buffering concept
- Understand work efficiency vs latency tradeoffs
- Develop hierarchical algorithms

Prefix Sum-Scan

- Frequently used for parallel work assignment and resource allocation
- Key primitive to convert serial computation into parallel
- Fundamental parallel computation pattern
- Efficient design for data intensive computations

Prefix Scan Definition

Definition: The all prefix-sums 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

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

Example: For $\oplus$ = addition

Input:

$$\{3, 1, 7, 0, 4, 1, 6, 3\}$$

Output:

$$\{3, 4, 11, 11, 15, 16, 22, 25\}$$

Real-World Example: Cutting Bread

We have 100 inches of bread to feed 10 people

Request sizes (inches):

```
{3, 5, 2, 7, 28, 4, 3, 0, 8, 1}
```

Method 1: Cut sequentially (3 first, 5 second, 2 third…)

Method 2: Calculate prefix-sum array

```
{3, 8, 10, 17, 45, 49, 52, 52, 60, 61}
```

Applications of Scan

- Histograms
- Reduction and broadcast O(log n)
- Sparse Matrix-Vector Multiply
- Adding n-bit integers in O(log n)
- Matrix multiplication O(log n)
- Triangular matrix inversion
- Dense matrix inversion
- Segmented Scan
- Parallel page layout
- Tridiagonal matrices
- Linked list traversal
- Minimal spanning trees
- Expression evaluation
- Convex hulls
- Image segmentation

Converting Serial to Parallel

Sequential:

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

Parallel:

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

Sequential Scan Algorithm

Recursive definition:

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

C Implementation:

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

✅ **Computationally efficient:** N additions for N elements - O(N)

Naïve Parallel Scan

Assign one thread per element, each calculates full sum:

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

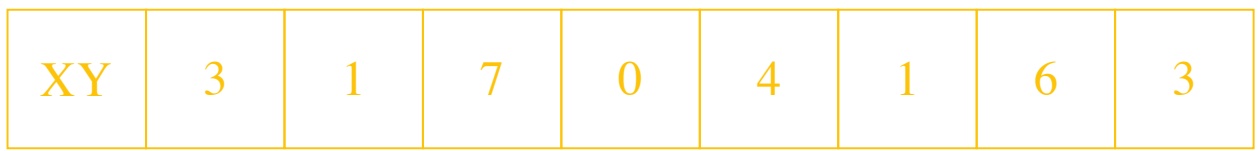
$$y_2 = x_0 + x_1 + x_2$$

⚠ **Remark:** Parallel programming is easy as long as you don’t care about performance!

Better Parallel Scan Algorithm

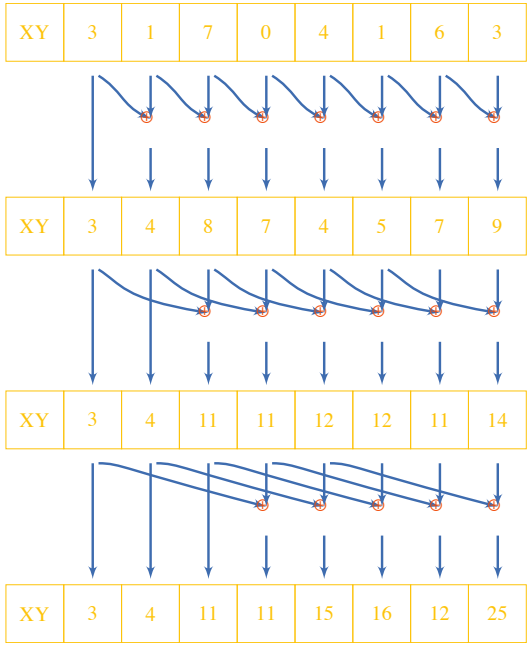
Steps:

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



   - Active threads: stride to n-1
   - Thread j adds elements j and j-stride
   - Requires barrier synchronization
3. Write output to device memory

Scan Example



Each iteration doubles the stride, building partial sums

Handling Dependencies

- Each thread can overwrite another thread's input
- **Solution:** Barrier synchronization pattern
  1. Barrier sync before read (ensure inputs ready)
  2. All threads secure their input operands
  3. Barrier sync again (ensure all secured)
  4. Perform addition and write output

Scan Kernel Code

```
__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];
  }

  for (unsigned int stride = 1; stride <= threadIdx.x; stride *= 2)
  {
    __syncthreads();
    float in1 = XY[threadIdx.x - stride];
    __syncthreads();
    XY[threadIdx.x] += in1;
  }
}
```

Work Efficiency Considerations

This scan executes log(n) parallel iterations:

- Steps do (n-1), (n-2), (n-4), … (n - n/2) adds each
- **Total adds:** n·log(n) - (n-1) → **O(n log n)** work
- ❌ **Not work efficient!**
  - Sequential scan: n adds
  - Factor of log(n) can hurt: **10× slower for 1024 elements!**
- A parallel algorithm can be slower than sequential when execution resources are saturated

Improving Efficiency: Balanced Trees

Strategy:

- Form a balanced binary tree on input data
- Sweep to and from the root
- Tree is a concept, not an actual data structure

For scan:

1. **Reduction phase:** Traverse leaves → root, building partial sums
2. Root holds sum of all leaves
3. **Reverse phase:** Traverse root → leaves, building output

Reduction Phase Kernel

```
// 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();
}
```

Each iteration halves the number of active threads, building a tree of partial sums.

Post Reduction Reverse Phase

```
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];
```

Propagates partial sums down the tree to compute final scan values.

Work Analysis

Reduction step: log(n) iterations

- Does n/2, n/4, …, 1 adds
- **Total:** (n-1) → O(n) work

Reverse step: log(n)-1 iterations

- Does 1, 3, 7, …, n/2-1 adds
- **Total:** (n-2) - (log(n)-1) → O(n) work

✅ **Both phases:** ≤ 2(n-1) adds total

✅ Only **2× the work** of sequential, but **parallelizable!**

Tradeoffs

Work efficient kernel (usually better):

- ✅ Better energy efficiency
- ✅ Less execution resource requirement

Work inefficient kernel (can be faster when):

- Sufficient execution resources available
- Single-step nature benefits absolute performance

Exclusive Scan

Definition: Returns shifted array with identity element at start

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

becomes

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

Example:

Input: `[3, 1, 7, 0, 4, 1, 6, 3]`

Output: `[0, 3, 4, 11, 11, 15, 16, 22]`

Why Exclusive Scan?

- **Primary use:** Finding beginning addresses of allocated buffers
- Inclusive and exclusive scans easily derived from each other

Comparison:

Input: `[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]`

Simple Exclusive Scan Kernel

Adapt inclusive 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 threads: load `X[blockIdx.x*blockDim.x+threadIdx.x-1]`

For work efficient version: Each thread loads two elements

- Only one zero should be loaded
- All elements shifted by one position

Dealing with Large Vectors

Build on work efficient scan kernel:

1. Assign sections of `2*blockDim.x` elements to each block
2. Each block writes its section sum to `Sum[]` array indexed by `blockIdx.x`
3. Run scan kernel on the `Sum[]` array
4. Add scanned `Sum[]` values to corresponding section elements

Similar adaptation for work inefficient kernel.

Conclusions

Key Themes:

- 📊 Scan memory pattern
- ⚡ Introduction to efficiency considerations
- 🌲 Balanced tree algorithms
- 🔄 Work vs latency tradeoffs

Conclusions

Key Themes:

- 📊 Scan memory pattern
- ⚡ Introduction to efficiency considerations
- 🌲 Balanced tree algorithms
- 🔄 Work vs latency tradeoffs