CSE 4373/5373 - General Purpose GPU Programming

GPU Pattern: Parallel Scan

Alex Dillhoff

University of Texas at Arlington

Parallel Scan

Parallel Scan is a common pattern in parallel programming. It is also known as **prefix sum**.

It is often applied to sequential problems.

Parallel Scan

It works with computations that can be described in terms of recursion.

- Resource allocation
- Work distribution
- Polynomial evaluation

Example: Inclusive Scan

Inclusive Scan

Given an array of numbers, the inclusive scan computes the sum of all elements up to a given index.

For example, given the array [1, 2, 3, 4, 5], the inclusive scan would produce [1, 3, 6, 10, 15].

Inclusive Scan

You could solve this recursively, but it would be horribly inefficient.

A sequential solution is achievable with dynamic programming.

The parallel solution is obviously faster.

Inclusive Scan

Dynamic programming solution:

```
void sequential_scan(float *x, float *y, uint N) {
    y[0] = x[0];
    for (uint i = 1; i < N; i++) {
        y[i] = y[i - 1] + x[i];
    }
}
```

Naive Parallel Reduction

Naive Parallel Reduction

If we have n elements, we could have n threads each compute the sum of a single element.

How many operations would that take?

Naive Parallel Reduction

The first thread computes the sum of 1 element, or 0 operations.

The second thread computes the sum of 2 elements, 1 operation, and so on.

Naive Parallel Reduction

This can be described as a sum of the first n natural numbers, which is $n(n+1)/2$.

This parallel solution is worse than the sequential solution, coming in at $O(n^2)$.

Kogge-Stone Parallel Scan

Kogge-Stone

A more sophisticated parallel algorithm that relies on **reduction**.

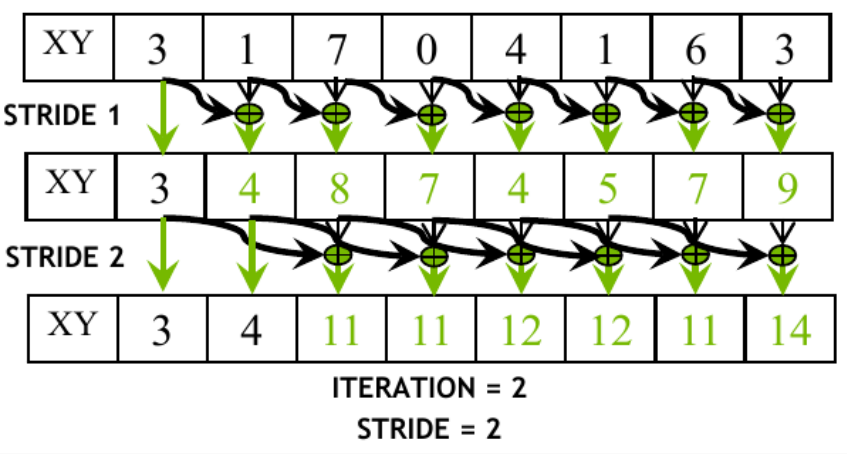
It was published in 1973 by Peter M. Kogge and Harold S. Stone at Stanford University.

Adapting the Reduction Tree

Summary

- Design reduction tree so that each thread has access to relevant inputs.
- The input matrix is modified to that input A_i contains the sum of up to 2^k elements after k iterations.
- For example, after iteration 2, A_3 contains the sum $A_0 + A_1 + A_2 + A_3$.

Adapting the Reduction Tree



Kogge-Stone Reduction Tree (Source: NVIDIA DLI)

Adapting the Reduction Tree

Start by loading the data into shared memory.

```
__global__ void Kogge_Stone_scan_kernel(float *X, float *Y, unsigned int N) {
    __shared__ float A[SECTION_SIZE];
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N) {
        A[threadIdx.x] = X[i];
    } else {
        A[threadIdx.x] = 0;
    }
```

Adapting the Reduction Tree

Reduction loop

```
for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
    __syncthreads();
    float temp;
    if (threadIdx.x >= stride) {
        temp = A[threadIdx.x] + A[threadIdx.x - stride];
    }
    __syncthreads();
    if (threadIdx.x >= stride) {
        A[threadIdx.x] = temp;
    }
}
```

Adapting the Reduction Tree

Write the data back to global memory.

```
    if (i < N) {
        Y[i] = A[threadIdx.x];
    }
}
```

Adapting the Reduction Tree

It is not possible to guarantee that a block can cover the entire input, so `SECTION_SIZE` is used to ensure that the input is covered.

This should be the same as the block size.

Adapting the Reduction Tree

Each thread starts off by loading its initial input into shared memory.

Starting with thread 2, each thread computes a sum of the value assigned to its thread as well as the one before it by a factor of `stride`.

Adapting the Reduction Tree

The loop itself is moving *down* the reduction tree, which is bounded logarithmically.

The local variable `temp` is used to store the intermediate result before barrier synchronization takes place.

Otherwise there would be a possibility of a *write-after-read* race condition.

Double Buffering

The temporary variable and second call to `__syncthreads()` are necessary since a thread may read from a location that another thread is writing to.

If the input and output arrays were represented by two different areas of shared memory, this call could be removed.

This approach is called **double-buffering**.

Double Buffering

- The input array is read from global memory into shared memory.
- At each iteration the data read from the first array is used to write new values to the second array.
- Values used in that iteration are only read from the first array.
- The second array can be used as the input array for the next iteration.
- This cycle continues until the final result is written.

Work Efficiency

Work efficiency is a measure of how efficient a particular algorithm is compared to the minimum number of operations required.

For inclusive scan, the minimum number of add operations required is $n - 1$, yielding an efficiency of $O(n)$.

Work Efficiency

During each iteration of the Kogge-Stone algorithm, each thread iterates over a loop that is logarithmic in size.

It starts with a stride of 2, then 4, 8, $\ldots \frac{n}{2}$.

Complexity: $O(n \log_2 n)$.

Work Efficiency

Due to the parallel nature of the algorithm, it still requires fewer steps than the sequential algorithm.

This takes $\frac{1}{m} n \log_2 n$ steps, where m is the number of execution units.

If we have as many execution units as we have elements, then we only need $\log_2 n$ steps.

Brent-Kung Parallel Scan

Brent-Kung Algorithm

Another approach, the Brent-Kung algorithm, is characterized by the following properties.

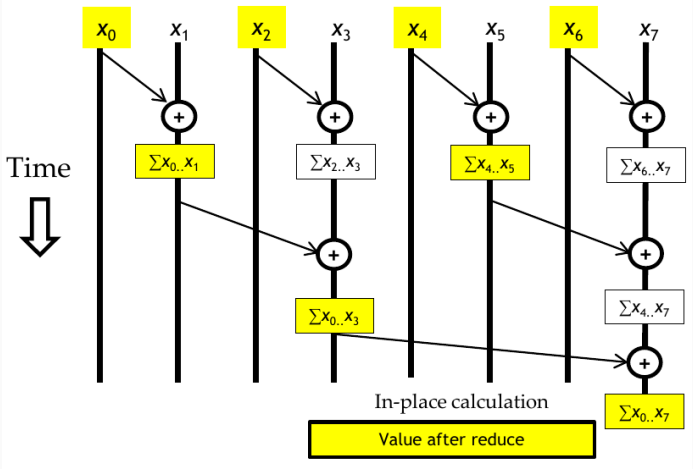
- Share intermediate results
- Distribute to different threads
- Sub-sums used to calculate some of the scan output values.

Brent-Kung Algorithm

Use reduction tree on the first $n/2$ elements, then use the results to reverse the tree.

At the start of the reverse direction, the elements with index $2^i - 1$, for $i = 1, 2, \ldots, \log_2 n$, already have the correct value.

Brent-Kung Algorithm



Brent-Kung Algorithm (Source: NVIDIA DLI)

Brent-Kung Algorithm

- The second half is better seen as a table of values.
- The row labeled `Initial` contains the state of the array after the first half of the algorithm is completed.
- For the values that already have their correct value, no update is needed.
- The two rows following `Initial` show the state of the array after the first and second iterations of the reverse direction.

Brent-Kung Algorithm

	0	1	2	3	4	5	6	7
Initial	0	0...1	2	0...3	4	4...5	6	0...7
Iteration 1						0...5		
Iteration 2			0...2		0...4		0...6	

Implementing the Forward Half

Reduction Tree Phase of Brent-Kung

```
for (uint stride = 1; stride <= blockDim.x; stride *= 2) {
    __syncthreads();
    if ((threadIdx.x + 1) % (stride * 2) == 0) {
        A[threadIdx.x] += A[threadIdx.x - stride];
    }
}
```

Implementing the Forward Half

- From the perspective of `threadIdx.x = 7`, the first iteration would add the value from `threadIdx.x = 6` to its own value.
- On the next iteration, the stride offset would add `threadIdx.x = 5`.
- Thread 5 already has the sum from 4 and 5 before being added to 7, so 7 now has the sums from indices 4 through 7.
- On its last iteration, the value for stride is now 4, and 7 adds the value from 3 to its own value.

Implementing the Forward Half

- **There is a lot of control divergence present in this code.**
- Since fewer threads stay active as the loop goes on, it is better to organize the threads such that they are contiguous.
- We can do that with slightly more complicated indexing, so that contiguous threads use data from the active portions of the array.

Implementing the Forward Half

Reduction Tree Phase of Brent-Kung (Revised)

```
for (uint stride = 1; stride <= blockDim.x; stride *= 2) {
    __syncthreads();
    int index = (threadIdx.x + 1) * 2 * stride - 1;
    if (index < blockDim.x) {
        A[index] += A[index - stride];
    }
}
```

Implementing the Forward Half

Thread 0 maps to index, thread 1 maps to index 3, thread 2 to index 5, and so on.

Only when the number of active threads drops below the warp size does control divergence become a problem.

Implementing the Reverse Half

Reverse Half

```
for (uint stride = blockDim.x / 4; stride > 0; stride /= 2) {
    __syncthreads();
    int index = (threadIdx.x + 1) * 2 * stride - 1;
    if (index + stride < blockDim.x) {
        A[index + stride] += A[index];
    }
}
```

Implementing the Reverse Half

Continuing the example from above, the first thread `threadIdx.x = 0` maps to index 3.

This will load the value from index 3 and add it to the value at index 5.

At that point, the value at index 5 will be the sum of the values from indices 0 through 5.

Efficiency Analysis

The reduction tree of the first half of the algorithm require $n - 1$ operations.

For n elements, the reverse half requires $(2 - 1) + (4 - 1) + \cdots + (n/2 - 1)$ operations for a total of $N - 1 - \log_2 n$.

This yields a work efficiency of $O(n)$.

Efficiency Analysis

Even though the theoretical work efficiency is better than Kogge-Stone, doesn't mean its performance will always be better in practice.

The drop off in active threads for Brent-Kung is much more severe than Kogge-Stone.

Efficiency Analysis

It also requires additional steps to perform the reverse half.

In general, Kogge-Stone is a better choice when we have more execution units, owing to its better parallelism.

Adding Coarsening

Adding Coarsening

What happens if we don't have enough resources to fully utilize the parallelism of the problem?

Coarsening: Each thread will execute a *phase* of sequential scan, which is more work efficiency than other of the solutions presented previously.

Adding Coarsening

Phase 1

- Threads collaborate to load data into shared memory.
- Perform a sequential scan.

Adding Coarsening

Phase 2

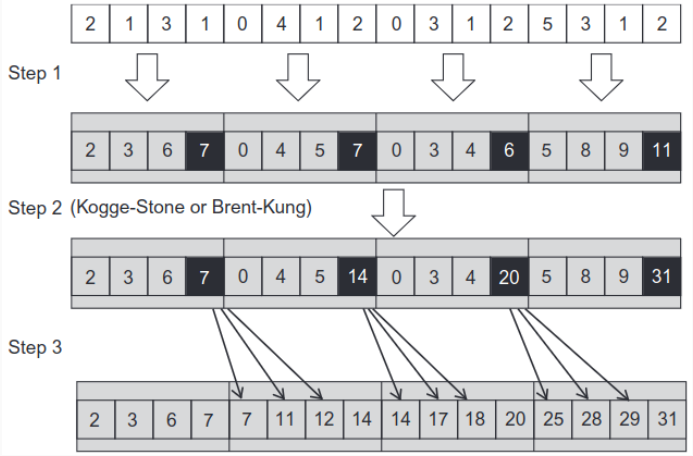
- Execute Kogge-Stone or Brent-Kung.
- Each thread has already performed a sequential scan, so it only needs to add the value from the last thread in the previous phase.

Adding Coarsening

Phase 3

- Each thread adds its last value to the first $n - 1$ elements of the next section.
- n is the number of elements assigned to each thread.

Adding Coarsening



Coarsening for parallel scan (Source: NVIDIA DLI)

Segmented Parallel Scan

Segmented Parallel Scan

What if our data cannot fit in memory?

Segmented Parallel Scan

What if our data cannot fit in memory?

None of the solutions we have presented so far can handle this.

Segmented Parallel Scan

Hierarchical scan can do this!

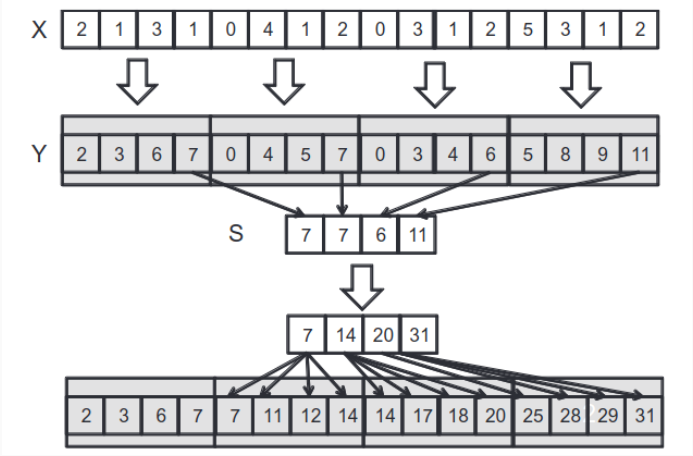
The data is partitioned into segments that can fit into each SM.

An additional kernel launches later to consolidate the results.

Segmented Parallel Scan

- Each scan block is treated as an individual application of one of the previous kernels.
- Each successive scan block initially does not contain the sums of the preceding blocks.
- Those will be added in a separate kernel.

Segmented Parallel Scan



Segmented Parallel Scan (Source: NVIDIA DLI)

Segmented Parallel Scan

After each scan block has computed the scan for its partition, the **last** elements are then processed in the next step.

These elements are all from different blocks: **we must write them to a global space.**

Segmented Parallel Scan

The resulting array has the final values of the scan corresponding to the indices from the original scan blocks.

These values can then be used to update the preceding elements in each scan block to complete the scan.

Segmented Parallel Scan

Kernel 1: Section Scan

The first kernel implements one of the previously discussed parallel scan kernels.

The accumulation array is given so that the blocks can write their output element values.

Segmented Parallel Scan

Kernel 1: Section Scan

```
__syncthreads();
if (threadIdx.x == blockDim.x - 1) {
    accumulation[blockIdx.x] = A[threadIdx.x];
}
```

Segmented Parallel Scan

Kernel 2: Update Scan

Run a kernel that updates the scan blocks with the values from the accumulation array.

Segmented Parallel Scan

Kernel 3: Update Elements

The final kernel takes the accumulated values and updates the original array so that all the elements have their correct scan value.

Optimizing Memory Efficiency

Optimizing Memory Efficiency

Segmented Scan is not memory efficient because of the need to store the accumulation array.

Data is transferred back and forth between global memory and host memory in between kernel calls.

Optimizing Memory Efficiency

Stream-based Scan

As before, all blocks begin by executing a local scan.

The first scan block has all the information it needs to compute the full scan for its partition.

The second block needs the final output value from the first before it can update its own values.

Optimizing Memory Efficiency

Stream-based Scan

It can then add that value to its own elements before sending its final value to the next block.

This process continues until all the blocks have been processed.

Optimizing Memory Efficiency

Is this fully sequential?

Optimizing Memory Efficiency

Is this fully sequential?

Not completely: the first scan on each block is parallel.

Optimizing Memory Efficiency

The second phase is sequential.

Once each block receives the final value from the previous block, it can then update its own values.

Optimizing Memory Efficiency

What tool do we have for block-wide communication?

Optimizing Memory Efficiency

What tool do we have for block-wide communication?

Atomic operations!

Optimizing Memory Efficiency

```
__shared__ float previous_sum;
if (threadIdx.x == 0) {
    // Wait for previous flag
    while(atomicAdd(&flags[bid], 0) == 0);
    // Read previous partial sum
    previous_sum = scan_value[bid];
    // Propagate partial sum
    scan_value[bid + 1] = previous_sum + local_sum;
    // Memory fence
    __threadfence();
    // Update flag
    atomicAdd(&flags[bid + 1], 1);
}
__syncthreads();
```

Optimizing Memory Efficiency

`flags` is a global array used to store lock values.

Once a flag is set to 1, the value is read and used to update the local sum.

Optimizing Memory Efficiency

What is `__threadfence()`?

Optimizing Memory Efficiency

What is `__threadfence()`?

It ensures that the thread has written to `scan_value` before the flag is set.

Optimizing Memory Efficiency

What does this code do often?

Optimizing Memory Efficiency

What does this code do often?

Global memory access.

Optimizing Memory Efficiency

Many of these values are overlapping and were accessed by previous blocks.

They are likely to be in the cache.

Optimizing Memory Efficiency

This only requires a single kernel.

The previous approach required 3 kernel calls, leading to a lot of data transfer.

Preventing Deadlocks

Depending on how the blocks are scheduled, **a deadlock can occur.**

The second block could be scheduled before the first block.

Preventing Deadlocks

The second block could be scheduled before the first block.

When would this cause a deadlock?

Preventing Deadlocks

The second block could be scheduled before the first block.

When would this cause a deadlock?

If there aren't enough SMs left.

Preventing Deadlocks

Solution: Dynamic block index assignment.

The block index assignment will not be dependent on `blockIdx.x`.

It is assigned as blocks are processed.

Preventing Deadlocks

```
__shared__ uint bid_s;
if (threadIdx.x == 0) {
    bid_s = atomicAdd(&block_index, 1);
}
__syncthreads();
uint bid = bid_s;
```

Preventing Deadlocks

The main call is to `atomicAdd()`, which is used to increment the block index.

This ensures that each block gets a unique index and that the blocks are processed in the order they are ready.

Recap

Recap

- Parallel scan is a powerful tool for solving problems that can be described in terms of a recursion.
- The Kogge-Stone and Brent-Kung algorithms are two ways of parallelizing the scan operation.
- This problem presents a unique look at how tradeoffs must be made when parallelizing a problem.
- At the end of the day, we must work within the constraints of the hardware and framework made available to us.