CSE 4373/5373 - General Purpose GPU Programming

GPU Pattern: Reduction

Alex Dillhoff

University of Texas at Arlington

Reduction

A **reduction** is a process that combines all the elements of an array into a single value.

For example, a histogram is a reduction that counts the number of occurrences of each value in an array.

Reduction

Many of the operations you rely on are examples of reductions.

The `sum` function is a reduction, as is the `max` function.

A reduction can be viewed as a linear combination of the input values, or transformed values, and is often used to compute a summary statistic.

Reduction

If $\phi(\cdot)$ is a binary operator, then a reduction computes the following:

$$v = \phi(v, x_i) \text{ for } i = 1, 2, \ldots, n,$$

where v is the accumulated value and x_i are the input values.

Reduction

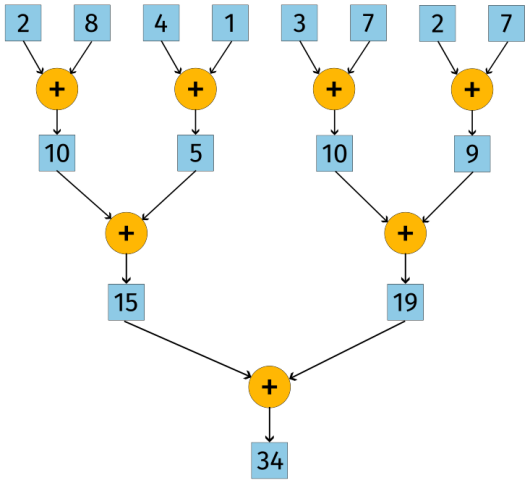
- The operator $\phi(\cdot)$ can be any associative and commutative operation, such as addition or multiplication.
- Each operator has a corresponding identity element, such as 0 for addition or 1 for multiplication.
- The identity element is used to initialize the reduction and can be represented as $v = v_0$ in the equation above.

Reduction Trees

Reduction Trees

- Reductions of any kind are well represented using trees.
- The first level of reduction maximizes the amount of parallelism.
- As the input is gradually reduced, fewer threads are needed.

Reduction Trees



Sum reduce as a reduction tree.

Reduction Trees

In order to implement a parallel reduction, the chosen operator must be associative.

For example, $a + (b + c) = (a + b) + c$.

The operator must also be commutative, such that $a + b = b + a$.

Reduction Trees

- Reduction trees reveal the logarithmic nature of parallel reductions.
- The number of threads is halved at each level of the tree.
- The number of levels in the tree is $\log_2(n)$, where n is the number of input values.

Reduction Trees

- Given an input size of $n = 1024$, the number of threads required is $\log_2(1024) = 10$.
- This is a significant reduction from the original input size.
- The sequential version of this reduction would require 1023 operations.

Reduction Kernels

A Simple Kernel

- Reduction requires communication between threads.
- Since only the threads within a single block can communicate, we will focus on a block-level reduction.
- Each block can work with a total of 2048 input values based on the limitation of 1024 threads per block.

A Simple Kernel

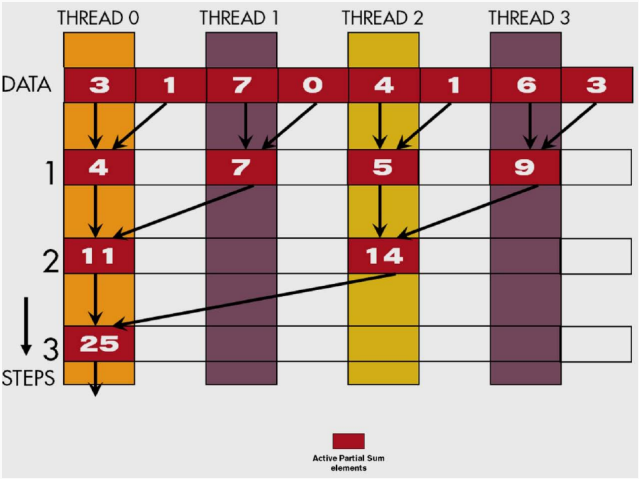
```
__global__ void sumReduceKernel(float *input, float *output) {
    uint i = 2 * threadIdx.x;
    for (uint stride = 1; stride <= blockDim.x; stride *= 2) {
        // Only threads in even positions participate
        if (threadIdx.x % stride == 0) {
            input[i] += input[i + stride];
        }
        __syncthreads();
    }

    if (threadIdx.x == 0) {
        *output = input[0];
    }
}
```

A Simple Kernel

- Each thread is assigned to a single write location `2 * threadIdx.x`.
- The stride is doubled after each iteration of the loop, effectively halving the number of active threads.
- It also determines the second value that is added to the first.
- By the last iteration, only one thread is active.

A Simple Kernel



Parallel reduction (Source: NVIDIA DLI)

A Simple Kernel

You can see that the kernel is simple, but it is also inefficient.

There is a great deal of control divergence, and the number of active threads is not maximized.

Minimizing Control Divergence

Minimizing Control Divergence

How can we minimize control divergence?

Keep as many threads active as possible.

Minimizing Control Divergence

- A warp of 32 threads would consume the execution resources even if half of them are inactive
- As each stage of the reduction tree is completed, the amount of wasted resources increases.
- Depending on the input size, entire warps could be launched and then immediately become inactive.

Minimizing Control Divergence

The number of execution resources consumes is proportional to the number of active warps across all iterations.

$$(\frac{5N}{64} + \frac{N}{128} + \frac{N}{256} + \cdots + 1) * 32$$

where N is the number of input values.

Minimizing Control Divergence

- Each thread operates on 2 values, so $\frac{N}{2}$ are launched in total.
- A total of $\frac{N}{64}$ warps are launched.
- For the first 5 iterations, all warps will be active.
- The 5th iteration only has 1 active thread in each warp.
- On the 6th iteration, the number of active warps is halved, and so on.

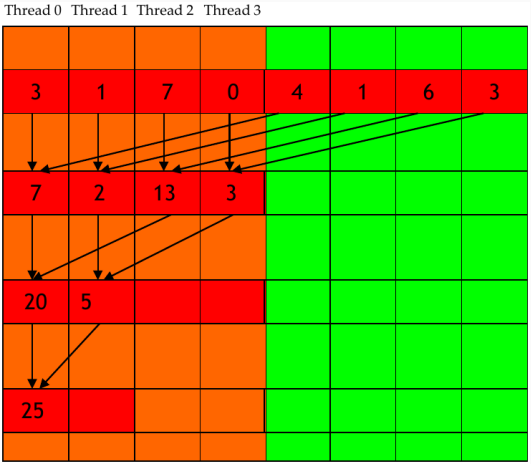
Minimizing Control Divergence

- For an input of size $N = 1024$, the number of resources consumed is $(80 + 8 + 4 + 2 + 1) * 32 = 3040$.
- The total number of results committed by the active threads is equal to the number of operations performed, which is $N - 1 = 1023$.
- The efficiency of the kernel is then $\frac{1023}{3040} = 0.34$.
- Only around 34% of the resources are used to perform the reduction.

Rearranging the Threads

- A simple rearrangement of where the active results are stored can improve the efficiency of the kernel by reducing control divergence.
- The idea is to keep the threads that own the results of the reduction close together.
- Instead of increasing the stride, it should be decreased.

Rearranging the Threads



Parallel reduction rearranged (Source: NVIDIA DLI)

Rearranging the Threads

- The rearrangement of the threads ensures that each warp has less control divergence.
- Warps that drop off after each iteration are no longer consuming execution resources.
- For an input of 256, the first 4 warps are fully utilized (barring the last thread of the last warp).

Rearranging the Threads

- After the first iteration, the number of active warps is halved.
- Warps 3 and 4 are now fully inactive, leaving warps 1 and 2 to perform the reduction operation on all threads.
- We can compute the number of resources consumed under this new arrangement as follows:

$$(\frac{N}{64} + \frac{N}{128} + \frac{N}{256} + \cdots + 1 + 5) * 32$$

Rearranging the Threads

- At each iteration, half of warps become inactive and no longer consume resources.
- The last warp will consume execution resources for all 32 threads, even though the number of active threads is less than 32.
- For $N = 1024$, the number of resources consumed is $(16 + 8 + 4 + 2 + 1 + 5) * 32 = 1152$, resulting in an efficiency of $\frac{1023}{1152} = 0.89$.

Memory Divergence of Reduction

Memory Divergence of Reduction

Does this kernel take advantage of memory coalescing?

Memory Divergence of Reduction

Does this kernel take advantage of memory coalescing?

- Each thread reads and writes from and to its *assigned* location.
- It also makes a read from a location that is a stride away.
- These locations are certainly not adjacent and will not be coalesced.

Memory Divergence of Reduction

- Adjacent threads do not access adjacent locations.
- The warp itself is unable to coalesce the thread requests into a single global memory request.
- Since each of the 32 threads in a warp are accessing their assigned locations with a separation of `stride`, the `64 * 4` bytes will require two 128 byte memory requests to access the data.

Memory Divergence of Reduction

With each iteration, the assigned locations will always be separated such that two 128 byte memory requests will need to be made.

Only on the last iteration, where only a single thread accesses a single assigned location, will a single memory request be made.

Reducing the Number of Global Memory Requests

Reducing Global Memory Requests

- Tiling moves data from global memory to shared memory.
- In reduction, threads write their results to global memory, which is read again in the next iteration.
- By keeping the intermediate results in shared memory, we can reduce the number of global memory requests.

Reducing Global Memory Requests

```
__global__ void sumReduceSharedKernel(float *input, float *output) {
    __shared__ float input_s[BLOCK_DIM];
    uint i = threadIdx.x;
    input_s[i] = input[i] + input[i + BLOCK_DIM];

    for (uint stride = blockDim.x / 2; stride >= 1; stride /= 2) {
        __syncthreads();
        if (i < stride) {
            input_s[i] += input_s[i + stride];
        }
    }

    if (i == 0) *output = input_s[0];
}
```

Reducing Global Memory Requests

- Input is loaded from global memory, added, and written to shared memory.
- This is the only time global memory is accessed, with the exception of the final write to the output.
- The call to `syncthreads()` moves to the top so that the shared memory is guaranteed before the next update.

Hierarchical Reduction

Hierarchical Reduction

A major assumption is that the kernels are running on a single block.

Thread synchronization is critical for the reduction to work correctly.

Hierarchical Reduction

- The kernel should allow for independent execution.
- This is achieved by segmenting the input and performing a reduction by segment.
- The final reduction is performed on the results of the individual reductions.

Reducing Global Memory Requests

```
void sumReduceHierarchicalKernel(float *input, float *output) {
    __shared__ float input_s[BLOCK_DIM];
    uint segment = 2 * blockDim.x * blockIdx.x;
    uint i = segment + threadIdx.x;
    uint t = threadIdx.x;
    input_s[t] = input[i] + input[i + BLOCK_DIM];

    for (uint stride = blockDim.x / 2; stride >= 1; stride /= 2) {
        __syncthreads();
        if (t < stride) input_s[t] += input_s[t + stride];
    }

    if (t == 0) atomicAdd(output, input_s[0]);
}
```

Reducing Global Memory Requests

Each block has its own shared memory and can independently perform the reduction.

Depending on the completion order, an atomic operation to add the local result is necessary.

Thread Coarsening... Again

Thread Coarsening

- Thread coarsening was first analyzed in the context of matrix multiplication.
- Whenever the device does not have enough resources to execute the number of threads requested, it is forced to serialize the execution.
- We can serialize the work done by each thread so that no extra overhead is incurred.
- Another benefit to thread coarsening is improved data locality.

Thread Coarsening

- Successive iterations increase the amount of inactive warps.
- For reduction, thread coarsening can be applied by increasing the number of elements that each one processes.
- If the time to perform the arithmetic is much faster than the time to load the data, then thread coarsening can be beneficial.

Reducing Global Memory Requests

```
__shared__ float input_s[BLOCK_DIM];
uint segment = COARSE_FACTOR * 2 * blockDim.x * blockIdx.x;
uint i = segment + threadIdx.x;
uint t = threadIdx.x;
float sum = input[i];
for (uint tile = 1; tile < COARSE_FACTOR * 2; tile++) {
    sum += input[i + tile * BLOCK_DIM];
}
input_s[t] = sum;
for (uint stride = blockDim.x / 2; stride >= 1; stride /= 2) {
    __syncthreads();
    if (t < stride) input_s[t] += input_s[t + stride];
}
if (t == 0) atomicAdd(output, input_s[0]);
```