Distributed and Parallel Computing

Lecture 06

Alan P. Sexton

University of Birmingham

Spring 2019

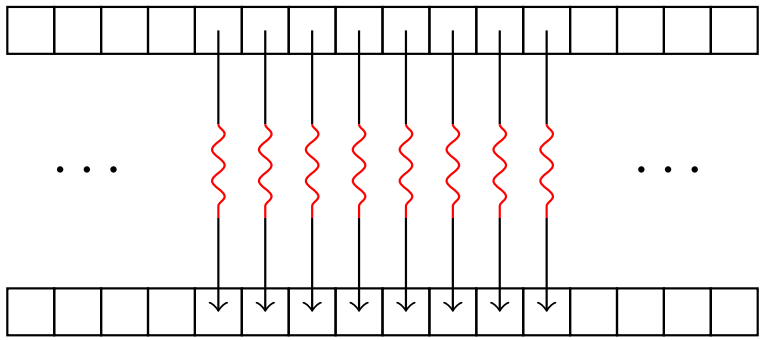
General Parallel Programming Patterns

There are a number of standard parallel programming patterns that form the basic building blocks of most GPU programs:

- Map
- Gather
- Scatter
- Stencil
- Transpose
- Reduce
- Scan/Sort
- Histogram

Map

Map uses each thread to take an input value from the location corresponding to the block/thread id and write an output value to the location corresponding to the block/thread id with no two threads reading from or writing to the same location.

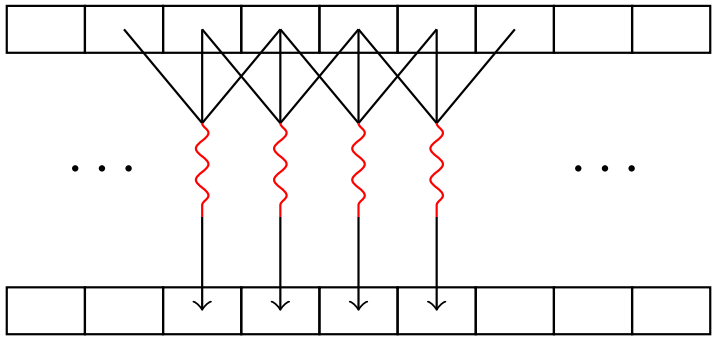


- One-to-one pattern
- No read/write race issues if source and destination are different
- Easy to ensure coalesced global memory accesses
- Easy to avoid shared memory bank collisions

Avoid double

Gather

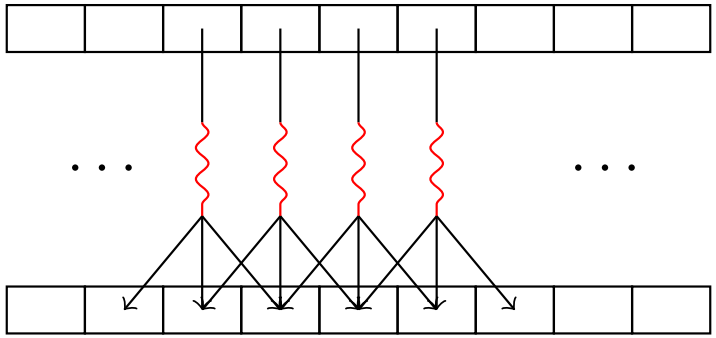
Gather uses each thread to take one or more input value from some locations (not just one from the direct location indicated by the thread and block id) and write an output value to one location with no two threads writing to the same location. Multiple different threads may share some read locations.



- Gather locations may have different patterns for each thread
- Many-to-one pattern
- No read/write race issues if source and destination are different
- Easy to ensure coalesced global memory accesses on writes
- Easy to avoid shared memory bank collisions on writes
- Care needed on read access patterns

Scatter

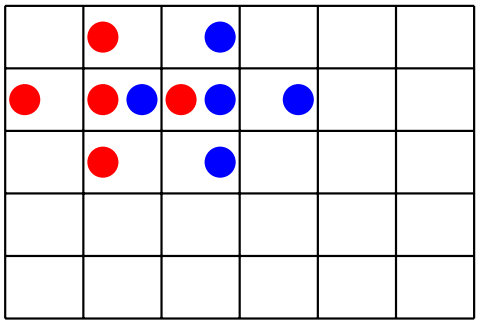
Scatter uses each thread to read the value from the location corresponding to the block/thread id and write output values to a number of locations Multiple different threads may share some write locations (e.g. using read/write operations such as increment)

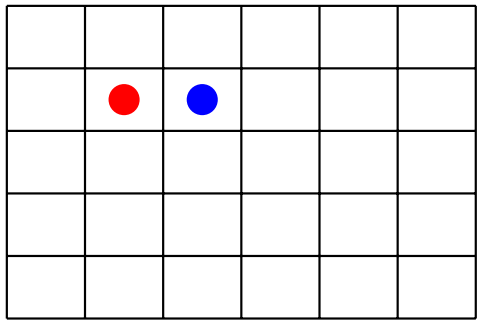


- Write locations may have different pattern for each thread
- One-to-many pattern
- Potential overwrite race issues
- Easy to ensure coalesced global memory accesses on reads
- Easy to avoid shared memory bank collisions on reads
- Care needed on write access patterns

Stencil

Stencil is a special case of Gather where the pattern of reads is constant across the threads.



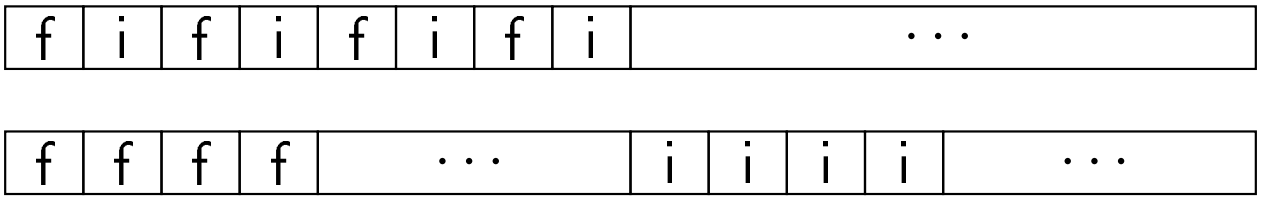


- Read locations have the same pattern for each thread
- Several-to-one pattern
- No read/write race issues if source and destination are different
- Easy to ensure coalesced global memory accesses on writes
- Easy to avoid shared memory bank collisions on writes
- Care needed on read access patterns

Transpose

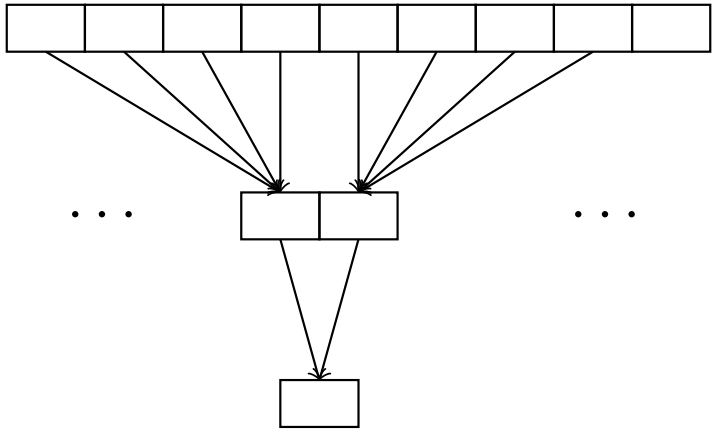
Transpose is most commonly the standard transpose operation on matrices, but is also used for restructuring datastructures for efficient memory accesses.

```
struct {
    float f;
    int i;
} X[LEN];
```



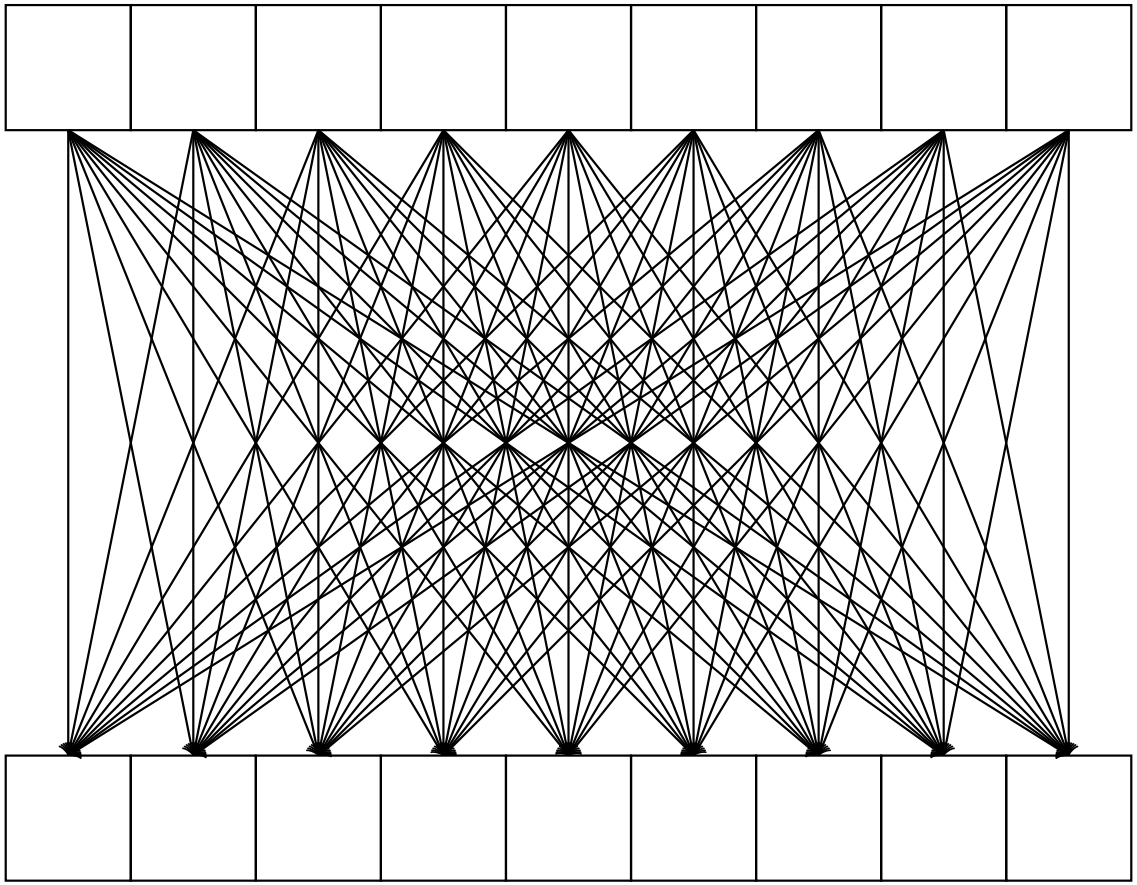
- Can be implemented with a *Scatter* or with a *Gather*

Reduce



- Many-to-one

Scan/Sort



- All-to-All (or at least Many-to-All)

Histogram — Serial Version

```
for (i=0; i < NUM_BINS; i++)
    bin[i] = 0;
for (i=0; i < NUM_VALS; i++)
    bin[computeBin(vals[i])] ++;
```

Histogram — Parallel Version (with error)

```
__global__ void hist(int *d_bins, const int *d_vals,
                     const int NUM_BINS)
{
    int id =  blockDim.x * blockIdx.x + threadIdx.x ;
    int val = d_vals[id] ;
    int bin = val / NUM_BINS; // computeBin fn: int divide

    d_bins[bin]++ ;
}
```

→ atomic

Histogram — Parallel Version (corrected)

```
__global__ void hist(int *d_bins, const int *d_vals,
                     const int NUM_BINS)
{
    int id =  blockDim.x * blockIdx.x + threadIdx.x ;
    int val = d_vals[id] ;
    int bin = val / NUM_BINS; // computeBin fn: int divide

    atomic_add(&d_bins[bin], 1) ;
}
```

Histogram — Parallel Version (corrected)

```
__global__ void hist(int *d_bins, const int *d_vals,
                     const int NUM_BINS)
{
    int id =  blockDim.x * blockIdx.x + threadIdx.x ;
    int val = d_vals[id] ;
    int bin = val / NUM_BINS; // computeBin fn: int divide

    atomic_add(&d_bins[bin], 1) ;
}
```

- Scalability?

Histogram — Parallel Version (corrected)

```
__global__ void hist(int *d_bins, const int *d_vals,
                     const int NUM_BINS)
{
    int id =  blockDim.x * blockIdx.x + threadIdx.x ;
    int val = d_vals[id] ;
    int bin = val / NUM_BINS; // computeBin fn: int divide

    atomic_add(&d_bins[bin], 1) ;
}
```

- Scalability?
- 1M values $\Rightarrow$ 1M atomic adds

Improving Parallel Histogram

- Have each thread calculate its own histogram on a subset of the data and then add to final result

Improving Parallel Histogram

- Have each thread calculate its own histogram on a subset of the data and then add to final result
  - No atomic adds necessary to build each thread's histogram
  - 1M values, 1024 threads 16 bins $\Rightarrow$ 1024 values per thread

Improving Parallel Histogram

- Have each thread calculate its own histogram on a subset of the data and then add to final result
  - No atomic adds necessary to build each thread's histogram
  - 1M values, 1024 threads 16 bins $\Rightarrow$ 1024 values per thread
  - 1024 histograms of 16 bins $\Rightarrow$ 16 * 1024 = 16284 atomic adds

Improving Parallel Histogram

- Have each thread calculate its own histogram on a subset of the data and then add to final result
  - No atomic adds necessary to build each thread's histogram
  - 1M values, 1024 threads 16 bins $\Rightarrow$ 1024 values per thread
  - 1024 histograms of 16 bins $\Rightarrow$ 16 * 1024 = 16284 atomic adds
- Better: Have each thread calculate its own histogram on a subset of the data and then **REDUCE** to final result

Further Improving Parallel Histogram

A more sophisticated, scalable approach:

1. Calculate bin numbers (map)
2. Sort bin numbers array (parallel-sort)
3. Reduce-by-key of 1-array using bin numbers as key

Example: `computeBin(val)= val / 3; //integer division`:

reduce everytime key changes

Reduce

Reduce-by-key