CSE 4373/5373 - General Purpose GPU Programming

GPU Pattern: Parallel Histogram

Alex Dillhoff

University of Texas at Arlington

Histograms

Histograms

A **histogram** is a discretized representation of the distribution of data.

It is a useful tool for understanding the distribution of data and is used in many applications, such as image processing, data analysis, and simulations.

Histograms

Examples of histograms:

- Frequency of words in a document
- Distribution of pixel intensities in an image
- Distribution of particle velocities in a simulation
- Distribution of thread block execution times in a GPU kernel

Histograms

Consider a histogram of the letters in a string, where each character is a key and the number of occurrences is the value.

A sequential solution is trivial, since there is no risk of write-after-read hazards.

Histograms

```c
void histogram_sequential(char *data, uint length, uint *hist) {
    for (uint i = 0; i < length; i++) {
        hist[data[i] - 'a']++;
    }
}
```

Histograms

- **Parallelize:** launch a kernel which has each thread work with one character from the input.
- This presents a major problem when updating the histogram, as multiple threads may try and increment the same location simultaneously.
- This is called *output interference*.

Race Conditions

Example

- Thread 1 and thread 2 may have the letter ’a’ as their input.
- They both issue a *read-modify-write* procedure
- The current value of the histogram is read, incremented, and then written back to memory.

Race Conditions

If thread 1 reads the value of the histogram before thread 2 writes to it, the value that thread 1 writes back will be incorrect.

This is a classic example of a **race condition**.

One thread could have read the updated value from the other thread, or both threads could have read the same value and incremented it, resulting in a loss of data.

Atomic Operations

One solution to this problem is to perform **atomic operations**.

This is a special type of operation that locks a memory location while it is being updated, preventing other threads from reading or writing to the same location until the operation is complete.

Atomic Operations

CUDA provides several atomic operations:

- `atomicAdd()`
- `atomicSub()`
- `atomicExch()`
- `atomicMin()`
- `atomicMax()`
- `atomicInc()`
- `atomicDec()`
- `atomicCAS()`

Atomic Operations

These are all *intrinsic functions*, meaning they are processed in a special way by the compiler.

They are implemented as inline machine instructions.

Atomic Operations

```
void histogram_atomic(char *data, uint length, uint *hist) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < length) {
        atomicAdd(&hist[data[i] - 'a'], 1);
    }
}
```

Latency of Atomic Operations

Latency of Atomic Operations

Problem: Atomic operations are slow.

Latency of Atomic Operations

- Atomic operations prevent the hardware from maximizing DRAM bursts since they serialize memory accesses.
- This can lead to a significant performance penalty.
- For each atomic operation, there are two delays: the delay from loading an element and the delay from storing the updated values.

Latency of Atomic Operations

- Not all threads will be loading and storing to the same memory location; this is dependent on the number of bins used in the histogram.
- If the number of bins is small, the performance penalty will be greater.
- This analysis is further complicated based on the distribution of the data.

Latency of Atomic Operations

- Atomic operations can be performed on the last level of cache.
- If the value is not in the cache, it will be brought into cache for future accesses.
- Although this does provide a performance benefit, it is not enough to offset the performance penalty of atomic operations.

Privatization

Privatization

Privatization is a technique used to reduce the contention on atomic operations.

It gives each thread its own private store so that it can update without contention.

Privatization

- Each copy must be combined in some way at the end.
- The cost of merging these copies is much less than the cost of the atomic operations.
- In practice, privatization is done for groups of threads, not individual ones.

Privatization

The previous example can be privatized by making a copy of the histogram for each thread block.

- Only a single block will update their own private copy.
- All copies are allocated as one monolithic array.
- Individual block use local indices to offset the pointer.
- Private copies of values will likely still be cached in L2, so the cost of merging the copies is minimal.

Privatization

For example, if we have 256 threads per block and 26 bins, we can allocate a 26×256 array of integers.

Each thread block will have its own copy of the histogram.

Privatization

```
void histogram_privatized(char *data, uint length, uint *hist) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < length) {
        int pos = data[i] - 'a';
        atomicAdd(&hist[blockIdx.x * NUM_BINS + pos], 1);
    }
    __syncthreads();
    if (blockIdx.x > 0) {
        for (uint bin = threadIdx.x; bin < NUM_BINS; bin += blockDim.x) {
            uint binValue = hist[blockIdx.x * NUM_BINS + bin];
            if (binValue > 0) atomicAdd(&hist[bin], binValue);
        }
    }
}
```

Privatization

- Each block is free to use its section of the histogram without contention.
- After all blocks have finished, the histogram is merged by summing the values of each bin across all blocks.
- Each block add its values to the global histogram.
- `__syncthreads` is used to ensure that all blocks have finished updating their private copies before the merge.

Privatization

- Each block after index 0 will add its values to the global histogram, represented by the first block.
- Only a single thread per block will be accessing each bin, so the only contention is with other blocks.
- If the bins are small enough, shared memory can be used to store the private copies.
- Even though an atomic operation is still required, the latency for loading and storing is reduced by an order of magnitude.

Privatization

```
void histogram_privatized(char *data, uint length, uint *hist) {
    __shared__ unsigned int hist_s[NUM_BINS];
    for (uint bin = threadIdx.x; bin < NUM_BINS; bin += blockDim.x) {
        hist_s[bin] = 0;
    }
    __syncthreads();
```

Privatization

```
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < length) {
            int pos = data[i] - 'a';
            atomicAdd(&hist_s[pos], 1);
        }
        __syncthreads();
        if (blockIdx.x > 0) {
            for (uint bin = threadIdx.x; bin < NUM_BINS; bin += blockDim.x) {
                uint binValue = hist_s[bin];
                if (binValue > 0) {
                    atomicAdd(&hist[bin], binValue);
                }
            }
        }
    }
```

Coarsening

Coarsening

The bottleneck of using privatization moves from DRAM access to merging copies back to the *public* copy of the data.

This scales up based on the number of blocks used, since each thread in a block will be sharing a bin in the worst case.

Coarsening

If the problem exceeds the capacity of the hardware, the scheduler will serialize the blocks.

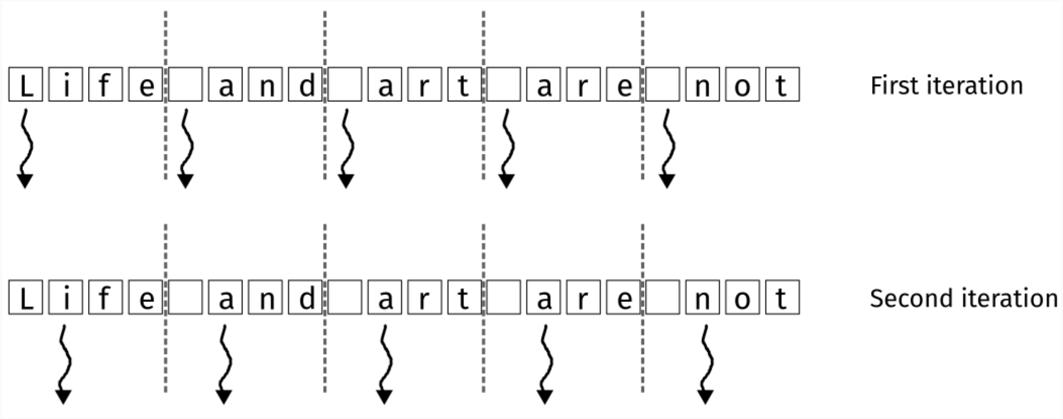
If they are serialized anyway, then the cost of privatization is not worth it.

Coarsening

Coarsening will reduce the overhead of privatization by reducing the number of private copies that are committed to the public one.

Each thread will process multiple elements.

Contiguous Partitioning



L i f e   a n d   a r t   a r e   n o t
First iteration
L i f e   a n d   a r t   a r e   n o t
Second iteration

Contiguous partitioning.

Contigious Partitioning

- Each thread is assigned a contiguous range of elements
- The kernel is a simple extension of the privatized kernel.
- This approach works better on a CPU, where there are only a small number of threads.
- On a GPU, it is less likely that the data will be in the cache since so many threads are competing.

Contiguous Partitioning

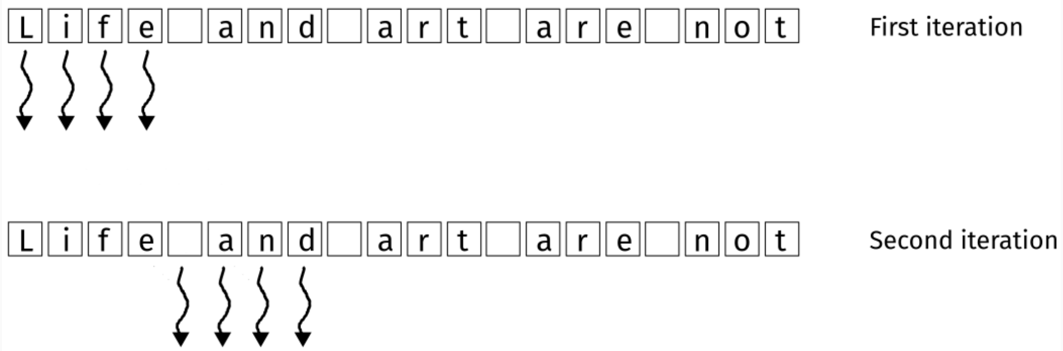
```
void histogram_privatized_cc(char *data, uint length, uint *hist) {
    __shared__ uint hist_s[NUM_BINS];
    for (uint bin = threadIdx.x; bin < NUM_BINS; bin += blockDim.x) {
        hist_s[bin] = 0;
    }
    __syncthreads();

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < length) {
        int pos = data[i] - 'a';
        atomicAdd(&hist_s[pos], 1);
    }
    __syncthreads();
```

Contiguous Partitioning

```
    if (blockIdx.x > 0) {
        for (uint bin = threadIdx.x; bin < NUM_BINS; bin += blockDim.x) {
            uint binValue = hist_s[bin];
            if (binValue > 0) {
                atomicAdd(&hist[bin], binValue);
            }
        }
    }
}
```

Interleaved Partitioning



L i f e a n d a r t a r e n o t
First iteration
L i f e a n d a r t a r e n o t
Second iteration

Interleaved partitioning.

Interleaved Partitioning

- Contiguous partitioning allowed for contiguous access to values relative to each thread.
- The memory was not contiguous with respect to other threads.
- Each individual read from memory was too far apart to take advantage of coalescing.

Interleaved Partitioning

With **interleaved partitioning**, the memory can be accessed in a single DRAM access since the memory is coalesced.

Interleaved Partitioning

The primary difference is in the following loop.

```
for (int i = tid; i < length; i += blockDim.x * gridDim.x) {
    int pos = data[i] - 'a';
    if (pos >= 0 && pos < 26) {
        atomicAdd(&hist_s[pos], 1);
    }
}
__syncthreads();
```

Interleaved Partitioning

The index `i` is incremented by `blockDim.x * gridDim.x`.

This ensures that the threads of each block access memory in a contiguous manner rather than each thread being contiguous.

Aggregation

Aggregation

- It is not uncommon that the input data will have a skewed distribution.
- There may be sections of the input that are locally dense.
- This will lead to a large number of atomic operations within a small area.
- Input can be aggregated into a larger update before being committed to the global histogram.

Aggregation

```
for (uint i = tid; i a, length; i += blockDim.x * gridDim.x) {
    int binIdx = data[i] - 'a';
    if (binIdx == prevBinIdx) {
        accumulator++;
    } else {
        if (prevBinIdx >= 0) atomicAdd(&hist_s[prevBinIdx], accumulator);
        accumulator = 1;
        prevBinIdx = binIdx;
    }
}
if (accumulator > 0) {
    atomicAdd(&hist_s[prevBinIdx], accumulator);
}
__syncthreads();
```

Aggregation

- The difference in this kernel is the histogram loop in the middle.
- The previous bin index is tracked to determine if contiguous values would be aggregated.
- As long as the values are the same, the accumulator is increased.

Aggregation

- As soon as a new value is encountered, a batch update is performed.
- This reduces the number of atomic operations by a factor of the number of contiguous values.

Aggregation

If the data is relatively uniform, the cost of aggregation exceeds the simple kernel.

If you are working with images, spatially local data will usually be aggregated.

This kernel would be beneficial in that case.

Aggregation

Another downside to the aggregated kernel is that it requires more registers and has an increased chance for control divergence.

As with all implementations, you should profile this against your use case.

Summary

Summary

- Computing histograms is a common operation in fields such as image processing, natural language processing, and physics simulations.
- A core preprocessing step for training a large language model is to compute the frequency of words in a corpus.
- This is a perfect example of a task that can be parallelized on a GPU.

Summary

In this lecture, we discussed:

- Histograms
- Race Conditions
- Atomic Operations
- Privatization
- Contiguous Partitioning
- Interleaved Partitioning
- Aggregation