Reduction Operation
Reduction Operation

- A **reduction** operation reduces a set of values to a single value
  - Sum, Product, Minimum, Maximum are examples

**Properties of reduction**
- Associativity
- Commutativity
- Identity value

- Reduction is a key primitive for parallel computing
  - E.g., MapReduce programming model

Dean and Ghemawat, “MapReduce: Simplified Data Processing of Large Clusters,” OSDI 2004
Tree-Based Reduction on GPU

Intra-block synchronization

\_syncthreads();

Inter-block synchronization

- Kernel termination and
  - Final reduction on CPU, or
  - Launch new reduction kernel on GPU
- Atomic operations in global memory

Partial results in shared memory (or registers)
Vector Reduction: Naïve Mapping (I)

Thread 0  Thread 2  Thread 4  Thread 6  Thread 8  Thread 10

0  1  2  3  4  5  6  7  8  9  10  11

0+1  2+3  4+5  6+7  8+9  10+11

0...3  4..7  8..11

0..7  8..15

iterations

Slide credit: Hwu & Kirk
Divergence-Free Mapping (I)

- All active threads belong to the same warp

Slide credit: Hwu & Kirk
Atomic Operations
Atomic Operations (I)

- CUDA provides atomic instructions on shared memory and global memory
  - They perform read-modify-write operations atomically

- Arithmetic functions
  - Add, sub, max, min, exch, inc, dec, CAS
    ```c
    int atomicAdd(int*, int);
    ```
  - Return value (old value)
  - Pointer to shared memory or global memory
  - Value to add

- Bitwise functions
  - And, or, xor

- Datatypes: int, uint, ull, float (half, single, double)*

* Datatypes for different atomic operations in [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions)
Atomic Operations (II)

- Atomic operations serialize the execution if there are atomic conflicts

<table>
<thead>
<tr>
<th>th0</th>
<th>th1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ t_{base} \]

No atomic conflict = concurrent updates

<table>
<thead>
<tr>
<th>th0</th>
<th>th1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ t_{conflict} \]

Atomic conflict = serialized updates
Uses of Atomic Operations

- **Computation**
  - Atomics on an array that will be the output of the kernel
  - Example
    - Histogram, reduction

- **Synchronization**
  - Atomics on memory locations that are used for synchronization or coordination
  - Example
    - Counters, locks, flags...

- Use them to prevent **data races** when more than one thread need to update the same memory location
Histogram Computation
Histogram Computation

- Histogram is a frequently used computation for reducing the dimensionality and extracting notable features and patterns from large data sets
  - Feature extraction for object recognition in images
  - Fraud detection in credit card transactions
  - Correlating heavenly object movements in astrophysics
  - ...

- Basic histograms - for each element in the data set, use the value to identify a “bin” to increment
  - Divide possible input value range into “bins”
  - Associate a counter to each bin
  - For each input element, examine its value and determine the bin it falls into and increment the counter for that bin
Sequential Histogram Computation

- A sequential implementation of histogram computation reads all input elements one by one and updates the corresponding histogram bins.

Thread 0

Iteration 1
Iteration 2
Iteration 3
Iteration 4
void histogram_calculation(unsigned int *histo, 
    unsigned int *input, 
    unsigned int input_size){

    int i = 0; // Loop index

    while(i < input_size){

        unsigned int val = input[i];

        histo[val] += 1;

        i++;
    }
}
Parallel Histogram Computation: Iteration 1

- Adjacent threads read adjacent input characters
  - Reads from the input array are coalesced
(Wrong) Parallel Histogram Kernel

```c
__global__ void histogram_kernel(unsigned int *histo,
                                  unsigned int *input,
                                  unsigned int input_size){

  int i = blockIdx.x * blockDim.x + threadIdx.x; // Thread index

  int stride = blockDim.x * gridDim.x; // Total number of threads

  while(i < input_size){
    unsigned int val = input[i];

    histo[val] += 1;

    i += stride;
  }
}
```
Parallel Histogram Computation: Iteration 2

- All threads move to the next section of the input
  - Each thread moves to element \text{threadID} + \#threads

We need to use \textit{atomic operations}
__global__ void histogram_kernel(unsigned int *histo,
    unsigned int *input,
    unsigned int input_size){

    int i = blockIdx.x * blockDim.x + threadIdx.x; // Thread index

    int stride = blockDim.x * gridSize.x; // Total number of threads

    while (i < input_size){

        unsigned int val = input[i];

        atomicAdd(&histo[val], 1);

        i += stride;
    }
}
Histograms are widely used in **image processing**

- Some **computation before voting** in the histogram may be needed

```plaintext
For (each pixel i in image I) {
    Pixel = I[i] // Read pixel
    Pixel' = Computation(Pixel) // Optional computation
    Histogram[Pixel']++ // Vote in histogram bin
}
```

- Parallel threads frequently incur **atomic conflicts** in image histogram computation
Frequent atomic conflicts due to the spatial similarity of the pixel value distribution in natural images

By using multiple sub-histograms (which are merged at the end), we can reduce the frequency of atomic conflicts.

This optimization technique is called privatization.
Privatization

- **Privatization** is an optimization technique where multiple private copies of an output are maintained, then the global copy is updated on completion
  - Operations on the output must be **associative and commutative** because the order of updates has changed

- **Advantages:**
  - Reduces contention on the global copy
  - If the output is small enough, the **private copy can be placed in shared memory** reducing access latency
**Privatization**: Per-block sub-histograms in shared memory

- Threads use atomic operations in shared memory

![Diagram showing privatization of histograms]

- **Shared memory**: b0, b1, b2, b3
- **Block 0’s sub-histo**: b0, b1, b2, b3
- **Block 1’s sub-histo**: b0, b1, b2, b3
- **Block 2’s sub-histo**: b0, b1, b2, b3
- **Block 3’s sub-histo**: b0, b1, b2, b3

![Global memory: b0, b1, b2, b3]

**Parallel reduction**
**Histogram Privatization + Coarsening**

- **Coarsening**: Each block processes several image chunks
  - Fewer sub-histograms to initialize and to merge at the end

![Diagram showing coarsening process](image)

- **Block 0’s sub-histo**: b0 b1 b2 b3
- **Block 1’s sub-histo**: b0 b1 b2 b3
- **Block 2’s sub-histo**: b0 b1 b2 b3
- **Block 3’s sub-histo**: b0 b1 b2 b3

**Shared memory**

**Global memory**

**Parallel reduction**
Parallel Histogram Kernel with Privatization (+ Coarsening)

__global__ void histogram_kernel(unsigned int *histo, unsigned int *input, unsigned int input_size) {

    int tid = blockIdx.x * blockDim.x + threadIdx.x; // Thread index
    int stride = blockDim.x * gridDim.x; // Total number of threads

    __shared__ unsigned int histo_s[BINS]; // Private per-block sub-histogram

    // Sub-histogram initialization
    for(int i = threadIdx.x; i < BINS; i += blockDim.x) {
        histo_s[i] = 0;
    }
    __syncthreads(); // Intra-block synchronization

    // Main loop to compute per-block sub-histograms
    for(int i = tid; i < input_size; i += stride) {

        unsigned int val = input[i]; // Global memory read (coalesced)

        atomicAdd(&histo_s[val], 1); // Atomic addition in shared memory
    }
    __syncthreads(); // Intra-block synchronization

    // Merge per-block sub-histograms and write to global memory
    for(int i = threadIdx.x; i < BINS; i += blockDim.x) {

        // Atomic addition in global memory
        atomicAdd(histo + i, histo_s[i]);
    }
}
Warp-Synchronous Programming for Atomic Operations
Warp Shuffle Functions

- Built-in **warp shuffle functions** enable threads to share data with other threads in the same warp
  - Faster than using shared memory and \_\_syncthreads() to share across threads in the same block

- Variants:
  - \_\_shfl_sync(mask, var, srcLane)
    - Direct copy from indexed lane
  - \_\_shfl_up_sync(mask, var, delta)
    - Copy from a lane with lower ID relative to caller
  - \_\_shfl_down_sync(mask, var, delta)
    - Copy from a lane with higher ID relative to caller
  - \_\_shfl_xor_sync(mask, var, laneMask)
    - Copy from a lane based on bitwise XOR of own lane ID
Other Warp-Synchronous Primitives

- `__syncwarp(unsigned)`
  Forces the reconvergence of the threads in the mask

- `__activemask()`
  Returns the mask of converged threads

- `__all_sync(unsigned, bool) and __any_sync(unsigned, bool)`
  Returns true if all or any of the participating threads pass true
Other Warp-Synchronous Primitives

- `__ballot_sync(unsigned, bool)`
  Returns the mask of threads that passed true

- `__match_all_sync(unsigned, _T)`
  Returns true if all participating threads pass the same value

- `__match_any_sync(unsigned, _T)`
  Returns the mask of participating threads passing the same value

Slide credit: Hwu & Kirk (PUMPS 2021)
Coalesced Atomic Operations

- Identify threads operating on the same atomic and use a reduction

```c
int atomic_add(int * ptr, int value){

    unsigned active_mask = __activemask();
    unsigned active_mask = __match_any_sync(active_mask, ptr);

    int value = reduce_warp(active_mask, value);

    if((__ffs(active_mask) - 1) == lane) {
        value = atomicAdd(ptr, value);
    }

    value = __shfl_sync(active_mask, value, __ffs(active_mask) - 1);
    return value;
}
```
Recommended Readings (II)

  - Chapter 9 - Parallel histogram: An introduction to atomic operations and privatization
Longer Lecture

Shared Memory Organization

Address 1056
0001 00001 00000 00
Page Row Bank

Address 0
0000 00001 00000 00
Page Row Bank

Address 32
0000 00001 00000 00
Page Row Bank

HetSys Course: Lecture 7: Parallel Patterns: Histogram (Fall 2022)

Onur Mutlu Lectures 32.4K subscribers

835 views 5 months ago Livestream - P&S Programming Heterogeneous Computing Systems with GPUs and other Accelerators (Fall 2022)
Project & Seminar, ETH Zürich, Fall 2022
Programming Heterogeneous Computing Systems with GPUs and other Accelerators (https://safari.ethz.ch/projects_and_s...)
Show more

https://youtu.be/A-q9AFntJF8
P&S Heterogeneous Systems

Parallel Patterns: Histogram

Dr. Juan Gómez Luna
Prof. Onur Mutlu

ETH Zürich
Spring 2023
21 April 2023