P&S Heterogeneous Systems
Parallel Patterns: Histogram

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Fall 2022
14 November 2022
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
Sequential Reduction

- A sequential implementation of reduction only needs a for loop to go through the whole input array
  - N elements → N iterations

- Many independent operations
  - A parallel implementation can calculate multiple partial sums, and then reduce them

```c
int A[0]; // Initialize with identity value
for(i = 0; i < N; ++i) {
    sum += A[i]; // Accumulate elements of input array A[
}
```
Tree-Based Reduction on GPU

**Block 0**
- **Warp 0**
- **Warp 1**

**Partial results in shared memory (or registers)**

**Intra-block synchronization**
- `__syncthreads();`

**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
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

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

2 0...3 4..7 8..11

3 0..7 8..15

iterations

Slide credit: Hwu & Kirk
Program with low SIMD utilization

```c
__shared__ float partialSum[];

unsigned int t = threadIdx.x;

for(int stride = 1; stride < blockDim.x; stride *= 2){
    __syncthreads();
    if (t % (2*stride) == 0)
        partialSum[t] += partialSum[t + stride];
}
```

How to avoid the warp underutilization?

<table>
<thead>
<tr>
<th>stride</th>
<th>Warps Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Warp 0</td>
</tr>
<tr>
<td>2</td>
<td>Warp 0, 1</td>
</tr>
<tr>
<td>4</td>
<td>Warp 0, 1, 2</td>
</tr>
</tbody>
</table>

![Diagram showing warp underutilization for different strides](image)

---

Vector Reduction: Naïve Mapping (II)

- Program with low SIMD utilization
- How to avoid the warp underutilization?
Divergence-Free Mapping (I)

- All active threads belong to the same warp
Divergence-Free Mapping (II)

Program with **high SIMD utilization**

```c
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for(int stride = blockDim.x; stride > 0; stride >>= 1){
    __syncthreads();

    if (t < stride)
        partialSum[t] += partialSum[t + stride];
}
```

Warp utilization is maximized
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
Reduction with Warp Shuffle

```c
__global__ void reduce_kernel(float* input, float* partialSums, unsigned int N) {
    unsigned int segment = 2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;

    // Load data to shared memory
    __shared__ float input_s[BLOCK_DIM];
    input_s[threadIdx.x] = input[i] + input[i + BLOCK_DIM];
    __syncthreads();

    // Reduction tree in shared memory
    for(unsigned int stride = BLOCK_DIM/2; stride > WARP_SIZE; stride /= 2) {
        if(threadIdx.x < stride) {
            input_s[threadIdx.x] += input_s[threadIdx.x + stride];
        }
        __syncthreads();
    }

    // Reduction tree with shuffle instructions
    float sum;
    if(threadIdx.x < WARP_SIZE) {
        sum = input_s[threadIdx.x] + input_s[threadIdx.x + WARP_SIZE];
        for(unsigned int stride = WARP_SIZE/2; stride > 0; stride /= 2) {
            sum += __shfl_down_sync(0xffffffff, sum, stride);
        }
    }

    // Store partial sum
    if(threadIdx.x == 0) {
        partialSums[blockIdx.x] = sum;
    }
}
```
Reduction with Warp Reduce

```c
__global__ void reduce_kernel(int* input, int* partialSums, unsigned int N) {

    unsigned int segment = 2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;

    // Load data to shared memory
    __shared__ int input_s[BLOCK_DIM];
    input_s[threadIdx.x] = input[i] + input[i + BLOCK_DIM];
    __syncthreads();

    // Reduction tree in shared memory
    for(unsigned int stride = BLOCK_DIM/2; stride > WARP_SIZE; stride /= 2) {
        if(threadIdx.x < stride) {
            input_s[threadIdx.x] += input_s[threadIdx.x + stride];
        }
        __syncthreads();
    }

    // Reduction tree with shuffle instructions
    int sum;
    if(threadIdx.x < WARP_SIZE) {
        sum = input_s[threadIdx.x] + input_s[threadIdx.x + WARP_SIZE];
    }
    // warp reduce intrinsic for cc 8.0 or higher
    sum = __reduce_add_sync(0xffffffff, sum);

    // Store partial sum
    if(threadIdx.x == 0) {
        partialSums[blockIdx.x] = sum;
    }
}
```
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);
    ```
  - Pointer to shared memory or global memory
  - Value to add
  - Return value (old value)

- 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
Atomic Operations (II)

- Atomic operations serialize the execution if there are atomic conflicts

No atomic conflict = concurrent updates

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
Data Races

- A **data race** occurs when multiple threads access the same memory location concurrently without ordering and at least one access is a write
  - Data races may result in unpredictable program output

- **Example:**

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>oldVal = bins[b]</code></td>
<td><code>oldVal = bins[b]</code></td>
</tr>
<tr>
<td><code>newVal = oldVal + 1</code></td>
<td><code>newVal = oldVal + 1</code></td>
</tr>
<tr>
<td><code>bins[b] = newVal</code></td>
<td><code>bins[b] = newVal</code></td>
</tr>
</tbody>
</table>

  - If both threads have the same `b` and `bins[b]` is initially 0, the final value of `bins[b]` could be **2** or **1**
### Data Races Example (I)

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>oldVal = bins[b]</code></td>
<td><code>oldVal = bins[b]</code></td>
</tr>
<tr>
<td>2</td>
<td><code>newVal = oldVal + 1</code></td>
<td><code>newVal = oldVal + 1</code></td>
</tr>
<tr>
<td>3</td>
<td><code>bins[b] = newVal</code></td>
<td><code>bins[b] = newVal</code></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><code>oldVal = bins[b]</code></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td><code>newVal = oldVal + 1</code></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td><code>bins[b] = newVal</code></td>
</tr>
</tbody>
</table>

In these two scenarios, the final value of `bins[b]` will be 2
### Data Races Example (II)

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>oldVal = bins[b]</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>newVal = oldVal + 1</code></td>
<td><code>oldVal = bins[b]</code></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>bins[b] = newVal</code></td>
<td><code>newVal = oldVal + 1</code></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td><code>bins[b] = newVal</code></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In these two scenarios (and many others), the final value of `bins[b]` will be **1**
Mutual Exclusion

- To avoid data races, concurrent read-modify-write operations to the same memory location need to be made mutually exclusive to enforce ordering.

- One way to do this on CPUs is using locks (mutex)
  - Example:

    ```c
    mutex_lock(lock);
    ++bins[b];
    mutex_unlock(lock);
    ```

Using locks with SIMD execution may cause deadlock.
Mutexes in a FGMT Architecture

- UPMEM Processing-in-Memory cores are fine-grained multithreaded
- Threads (called *tasklets*) can use mutexes for concurrent read-modify-write operations

**DPU Pipeline**
- In-order pipeline
  - Up to 425 MHz
- Fine-grain multithreaded
  - 24 hardware threads
- 14 pipeline stages
  - **DISPATCH**: Thread selection
  - **FETCH**: Instruction fetch
  - **READOP**: Register file
  - **FORMAT**: Operand formatting
  - **ALU**: Operation and WRAM
  - **MERGE**: Result formatting

**Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture**

Juan Gómez-Luna1  Izzat El Hajj2  Ivan Fernandez1,3  Christina Giannoula1,4  Geraldo F. Oliveira1  Onur Mutlu1

1ETH Zürich  2American University of Beirut  3University of Malaga  4National Technical University of Athens


**Figure 20: Execution times (ms) of two versions of histogram (HST-L, HST-S) on 1 DPU.**

Processing-in-Memory Course
Lecture 2: Real-world PIM: UPMEM PIM (Spring 2022)
[https://youtu.be/6dwV_RBjK2c](https://youtu.be/6dwV_RBjK2c)
Atomic Operations: Architectural Support

- The GPU ISA evolves with GPU architecture generations

- CUDA: `int atomicAdd(int*, int);`
- PTX: `atom.shared.add.u32 %r25, [%rd14], %r24;`
- SASS:

Tesla, Fermi, Kepler

```c
/*00a0*/ LDSLK P0, R9, [R8];
/*00a8*/ @P0 IADD R10, R9, R7;
/*00b0*/ @P0 STSCUL P1, [R8], R10;
/*00b8*/ @!P1 BRA 0xa0;
```

Maxwell, Pascal, Volta...

```c
/*01f8*/ ATOMS.ADD RZ, [R7], R11;
```

Native atomic operations for 32-bit integer, and 32-bit and 64-bit atomicCAS
Recall: 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
Optimized Parallel Reduction

- 7 versions in CUDA samples: Tree-based reduction in shared memory
  - Version 0: No whole warps active
  - Version 1: Contiguous threads, but many bank conflicts
  - Version 2: No bank conflicts
  - Version 3: First level of reduction when reading from global memory
  - Version 4: Warp shuffle or unrolling of final warp
  - Version 5: Warp shuffle or complete unrolling
  - Version 6: Multiple elements per thread sequentially
Reduction with Atomic Operations

- 3 new versions of reduction based on 3 previous versions
  - Version 0: No whole warps active
  - Version 3: First level of reduction when reading from global memory
  - Version 6: Multiple elements per thread sequentially

- New versions 7, 8, and 9
  - Replace the for loop (tree-based reduction) with one shared memory atomic operation per thread
Search Space of Parallel Reduction

Over 85 different versions possible!

Automatic Generation of Parallel Reduction

Simon Garcia De Gonzalo, Sitao Huang, Juan Gomez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu,
"Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs"
[Slides (pptx) (pdf)]

Automatic Generation of
Warp-Level Primitives and Atomic Instructions for
Fast and Portable Parallel Reduction on GPUs

Simon Garcia De Gonzalo
CS and Coordinated Science Lab
UIUC
grcdgnz2@illinois.edu

Sitao Huang
ECE and Coordinated Science Lab
UIUC
shuang91@illinois.edu

Juan Gómez-Luna
Computer Science
ETH Zurich
juang@ethz.ch

Simon Hammond
Scalable Computer Architecture
Sandia National Laboratories
sdhammo@sandia.gov

Onur Mutlu
Computer Science
ETH Zurich
omutlu@ethz.ch

Wen-mei Hwu
ECE and Coordinated Science Lab
UIUC
w-hwu@illinois.edu
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.
Sequential Histogram Function

```c
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 * gridSize.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 threadID + #threads

We need to use 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 * gridDim.x;  // Total number of threads

    while(i < input_size){

        unsigned int val = input[i];

        atomicAdd(&histo[val], 1);

        i += stride;
    }
}

Atomic Operations on DRAM

- Each Load-Modify-Store has two full memory access delays
  - All atomic operations on the same variable (RAM location) are serialized
Hardware Improvements

- Atomic operations on Fermi L2 cache
  - Medium latency, but still serialized
  - Global to all blocks
  - “Free improvement” on Global Memory atomics
Atomic operations on Shared Memory

- Very short latency, but still serialized
- Private to each thread block
- Need algorithm work by programmers (more later)
- Histograms are widely used in image processing

  - Some computation before voting in the histogram may be needed

    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 of privatization](image.png)
**Histogram Privatization + Coarsening**

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

![Diagram showing the process of histogram privatization and coarsening]

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

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

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

4. **Block 3**
   - Block 3’s sub-histo
   - Shared memory: b0 b1 b2 b3

- **Parallel reduction**
- **Final histogram**
- **Global memory**
Parallel Histogram Kernel with Privatization (+ Coarsening)

```c
__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

Slide credit: Izzat El Hajj
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
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;
}
```
Evolution of the Architectural Support for Atomic Operations
The architectural support for atomic operations evolves across GPU generations

CUDA: int atomicAdd(int*, int);

PTX: atom.shared.add.u32 %r25, [%rd14], %r24;

SASS:

```
/*00a0*/ LDSLK P0, R9, [R8];
/*00a8*/ @P0 IADD R10, R9, R7;
/*00b0*/ @P0 STSCUL P1, [R8], R10;
/*00b8*/ @!P1 BRA 0xa0;
```

```
/*01f8*/ ATOMS.ADD RZ, [R7], R11;
```

Tesla, Fermi, Kepler

Maxwell, Pascal, Volta...

Native atomic operations for 32-bit integer, and 32-bit and 64-bit atomicCAS
Lock-Free Mechanism for Shared Memory Atomics

- Tesla, Fermi, and Kepler architectures
  - Parallel processing unit (PPU) = GPU core

---

```
/*00a0*/ LDSLK P0, R9, [R8];
/*00a8*/ @P0 IADD R10, R9, R7;
/*00b0*/ @P0 STSCUL P1, [R8], R10;
/*00b8*/ @!P1 BRA 0xa0;
```

Coon et al., “Lock Mechanism to Enable Atomic Updates to Shared Memory,” US8055856
Shared Memory Organization

Address 1056
0001 00001 00000 00
Page  Row  Bank

Address 0
0000 00000 00000 00
Page  Row  Bank

Address 32
0000 00000 00000 00
Page  Row  Bank

Page 0
0 1 2 3 32 33 34 35 1123 2211 12258 12259
Page 1
1024 1025 1026 1027 2115 ...
Page 2
1056 1057 1058 1059 2147 ...
Page 11

Lock addresses

Storage resource

Memory lock unit

Assembly Code for Shared Memory Atomics
(pre Maxwell)

- **Lock-free mechanism**
  - Predicated execution
- **LDSLK** loads from shared memory and sets one lock bit
  - Predication register $P_0$ set if lock succeeds
- **STSCUL** stores and releases the lock
- **BRA** jumps to start a new attempt to acquire the lock

```assembly
/*0090*/  PSETP.AND.AND P1, PT, !PT, PT, PT; // Predicate set predicate
/*0098*/  SSY 0xd0;  // Set synchronization point
/*00a0*/  LDSLK P0, R9, [R8];  // Load and lock
/*00a8*/  @P0 IADD R10, R9, R7;  // Integer addition
/*00b0*/  @P0 STSCUL P1, [R8], R10;  // Store conditionally and unlock
/*00b8*/  @!P1 BRA 0xa0;  // Predicated unconditional branch
/*00c8*/  Instruction.S
/*00d0*/
...```
The limited number of locks may cause high contention
- 256 lock bits in Tesla, 1024 lock bits in Fermi and Kepler

Gomez-Luna+, “Performance Modeling of Atomic Additions on GPU Scratchpad Memory,” IEEE TPDS, 2013 (supplemental material)
### Example Execution Timeline

**Warp access pattern**

```plaintext
LDSLK P0, R7, [R9];
@P0 IADD R10, R7, 0x1;
@P0 STSUL [R9], R10;
@P0 BRA 0x210;
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Active threads</th>
<th>Addresses involved</th>
<th>In processing unit...</th>
<th>In lock unit...</th>
<th>In memory...</th>
<th>Latency penalties</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDSLK P0</td>
<td>0 1 2 3 7 8 ... 31</td>
<td>0 1 7 8 ... 31</td>
<td></td>
<td></td>
<td></td>
<td>Read access</td>
</tr>
<tr>
<td></td>
<td>1 3</td>
<td>32 33</td>
<td>Bank conflict</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1024</td>
<td>Bank conflict</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1056</td>
<td>Bank conflict</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@P0 IADD</td>
<td>0 1 2 3 7 8 ... 31</td>
<td>0 1 7 8 ... 31</td>
<td></td>
<td></td>
<td></td>
<td>Predicated addition</td>
</tr>
<tr>
<td></td>
<td>1 3</td>
<td>32 33</td>
<td>Release locks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@P0 STSUL</td>
<td>0 2 7 8 ... 31</td>
<td>0 1 7 8 ... 31</td>
<td></td>
<td></td>
<td></td>
<td>Release locks</td>
</tr>
<tr>
<td></td>
<td>1 3</td>
<td>32 33</td>
<td>Write access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!@P0 BRA</td>
<td>4 5 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Predicated branch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDSLK P0</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Read access</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1024</td>
<td>Bank conflict</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1056</td>
<td>Bank conflict</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@P0 IADD</td>
<td>4 6</td>
<td>0 1056</td>
<td></td>
<td></td>
<td></td>
<td>Predicated addition</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Release locks</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1056</td>
<td>Write access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!@P0 BRA</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Predicated branch</td>
</tr>
<tr>
<td>LDSLK P0</td>
<td>5</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
<td>Hash[1024] = 0; LockBits[0] = 1</td>
</tr>
<tr>
<td>@P0 IADD</td>
<td>5</td>
<td>1024</td>
<td>Predicated addition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@P0 STSUL</td>
<td>5</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
<td>Release locks</td>
</tr>
<tr>
<td>!@P0 BRA</td>
<td>5</td>
<td>1024</td>
<td>Predicated branch</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Gomez-Luna+, “Performance Modeling of Atomic Additions on GPU Scratchpad Memory,” IEEE TPDS, 2013 (supplemental material)
Microbenchmarking Atomic Operations

- Microbenchmark
  - \( H_s \) is a shared memory array
  - \texttt{clock()} returns the cycle count

```c
for(int i = begin; i < size; i += num_threads){
    // Read from global memory
    vote = input_data[i];

    start_time = clock(); // Start timing

    // Macro repeats \texttt{atomicAdd} 256 times
    repeat256(atomicAdd(&Hs[vote], 1));

    stop_time = clock(); // Stop timing
}
```

Microbenchmarking Results (I)

- **Position conflict**: $n$ is the number of threads accessing the same address

Microbenchmarking Results (II)

- **Bank conflicts**: $m$ is the number of threads accessing the same bank (stride 32)

---

Microbenchmarking Results (III)

- **Bank conflicts**: $m$ is the number of threads accessing the same bank (stride 256)

Limited Number of Lock Bits

- The limited number of locks may cause high contention
  - 256 lock bits in Tesla, 1024 lock bits in Fermi and Kepler

We can use configurable hash functions
Configurable Hash Functions

- Configurable hash functions can reduce the number of bank conflicts and lock conflicts.

Bank and lock conflicts are greatly reduced with an XOR hash function.

Optimizing Histogram Computation (I)

- Multiple private sub-histograms per block
- + padding to avoid conflicts on banks and lock bits
Optimizing Histogram Computation (II)

- Significant execution time reduction on Fermi and Kepler
  - 100 natural images

Atomic Operations on Shared Memory

- The architectural support for atomic operations evolves across GPU generations
  
- **CUDA**: `int atomicAdd(int*, int);`
  
- **PTX**: `atom.shared.add.u32 %r25, [%rd14], %r24;`
  
- **SASS**:

  ```
  /*00a0*/ LDSLK P0, R9, [R8];
  /*00a8*/ @P0 IADD R10, R9, R7;
  /*00b0*/ @P0 STSCUL P1, [R8], R10;
  /*00b8*/ @!P1 BRA 0xa0;
  ```

  **Tesla, Fermi, Kepler**

  Native atomic operations for 32-bit integer, and 32-bit and 64-bit atomicCAS

  ```
  /*01f8*/ ATOMS.ADD RZ, [R7], R11;
  ```

  **Maxwell, Pascal, Volta...**
Optimizing Histogram Computation (III)

- Improved hardware (since Maxwell) saves programmers’ effort

Another Example: Stream Compaction

- 1 single counter per block updated via shared memory atomics

CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics
Atomic Units Near Memory Banks

- AMD GCN architecture: Atomic units near local data share (LDS)

![Diagram of AMD GCN architecture](image-url)

Atomic Operations on Global Memory

- **Tesla**
  - Atomic operations **executed on DRAM**

- **Fermi**
  - Executed on L2
  - **Atomic units near L2**

- **Kepler**
  - Atomic units near L2 incorporate a **local buffer**

- **Pascal**
  - 64-bit FP **atomicAdd()**
Atomic Units near L2 with Local Buffer

- L2 is divided into partition units

Glasco et al., "Cache-based Control of Atomic Operations in Conjunction with an External ALU Block," US8135926B1
Scatter vs. Gather

- Scatter assigns input elements
- Gather assigns output elements
Scatter vs. Gather: Example Codes

```c
__global__ void s2g_gpu_scatter_kernel(unsigned int* in, unsigned int* out,
              unsigned int num_in, unsigned int num_out) {

    unsigned int inIdx = blockIdx.x*blockDim.x + threadIdx.x;

    if(inIdx < num_in) {
        unsigned int intermediate = outInvariant(in[inIdx]);
        for(unsigned int outIdx = 0; outIdx < num_out; ++outIdx) {
            atomicAdd(&(out[outIdx]), outDependent(intermediate, inIdx, outIdx));
        }
    }
}

__global__ void s2g_gpu_gather_kernel(unsigned int* in, unsigned int* out,
              unsigned int num_in, unsigned int num_out) {

    unsigned int outIdx = blockIdx.x*blockDim.x + threadIdx.x;

    if(outIdx < num_out) {
        unsigned int out_reg = 0;
        for(unsigned int inIdx = 0; inIdx < num_in; ++inIdx) {
            unsigned int intermediate = outInvariant(in[inIdx]);
            out_reg += outDependent(intermediate, inIdx, outIdx);
        }
        out[outIdx] += out_reg;
    }
}
```
Scatter vs. Gather: Evaluation (I)

- Scatter: Large penalty due to atomic conflicts in DRAM

![Diagram showing execution time for differentmemory sizes for scatter and gather operations on Tesla, Fermi, Kepler, and Maxwell architectures.](image-url)
Scatter vs. Gather: Evaluation (II)

- L2 atomics improve the performance by an order of magnitude
Scatter vs. Gather: Evaluation (III)

- Another 10x speedup with local buffers

![Graph showing execution time (ms) for Scatter without Conflicts, Scatter with Conflicts, and Gather for Tesla, Fermi, Kepler, and Maxwell architectures. The graph indicates Kepler: Buffer in atomic units.](image-url)
Effect of Hardware Improvements

- 256-bin histogram computation for 100 natural images
  - Shared memory implementation uses 1 private histogram per block
  - Global atomics greatly improved in Kepler
  - Native shared memory atomics since Maxwell

![Histogram Execution Time Graph]

- Fermi
- Kepler
- Maxwell
Recommended Readings (I)

  - Chapter 9 - Parallel patterns — parallel histogram computation:
    An introduction to atomic operations and privatization
Recommended Readings (II)

  - Chapter 9 - Parallel histogram: An introduction to atomic operations and privatization
Parallel Patterns: Histogram

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Fall 2022
14 November 2022