P&S Heterogeneous Systems

Parallel Patterns: Histogram

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2021

18 November 2021

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

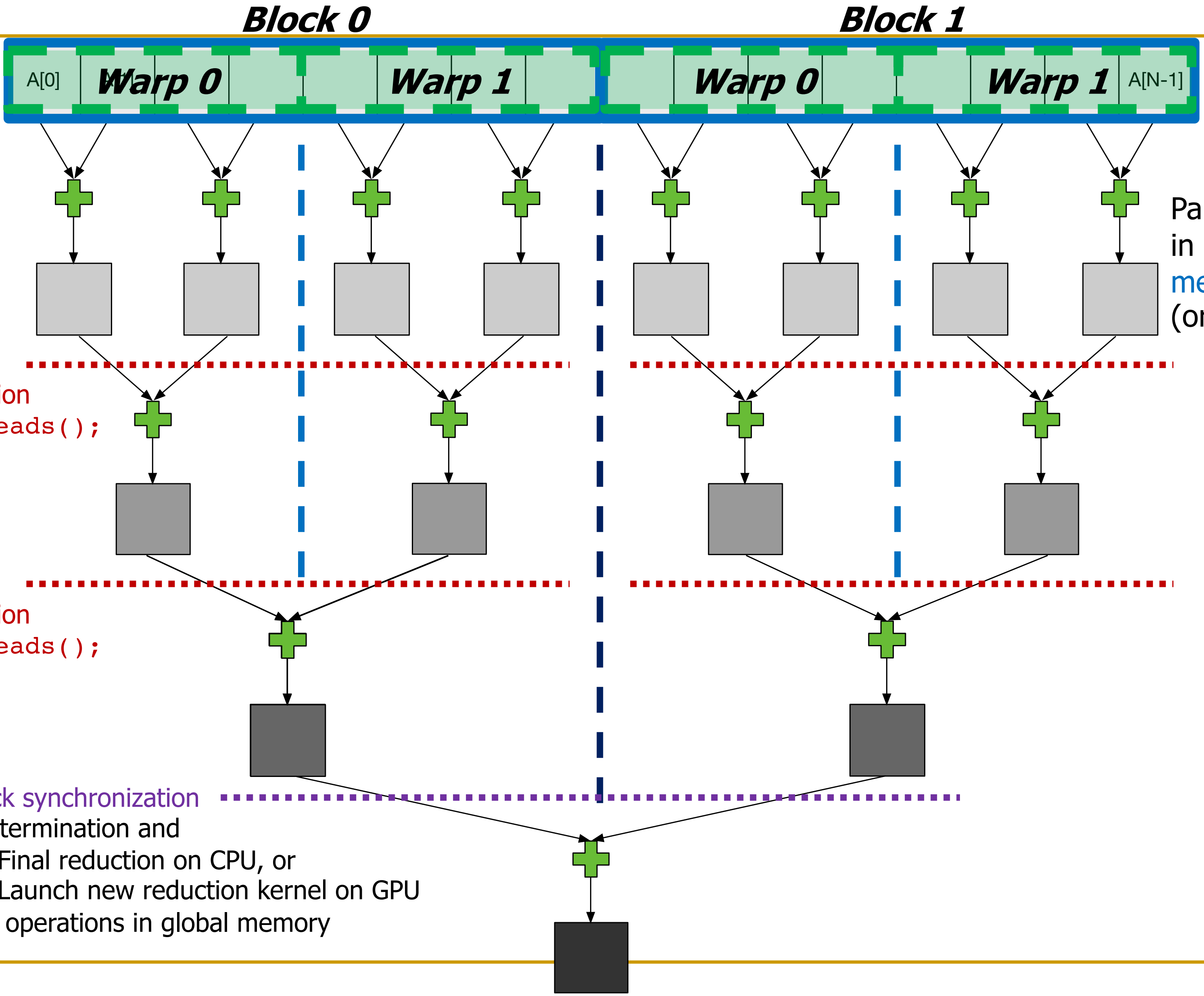
Sequential Reduction

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

```
sum = 0; // Initialize with identity value
for(i = 0; i < N; ++i) {
    sum += A[i]; // Accumulate elements of input array A[]
}
```

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

Tree-Based Reduction on GPU



Partial results in shared memory (or registers)

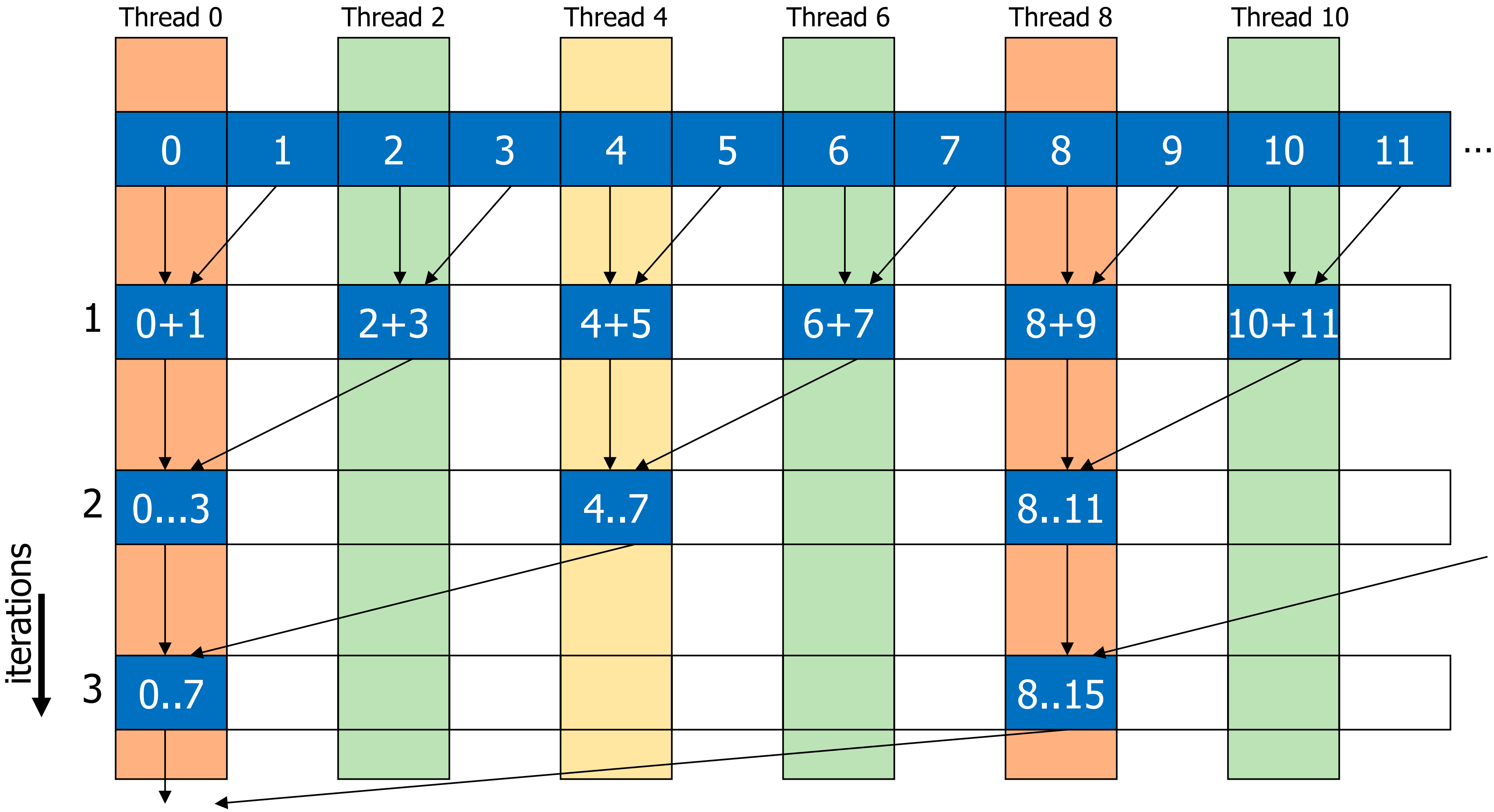
Intra-block synchronization
`__syncthreads();`

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)

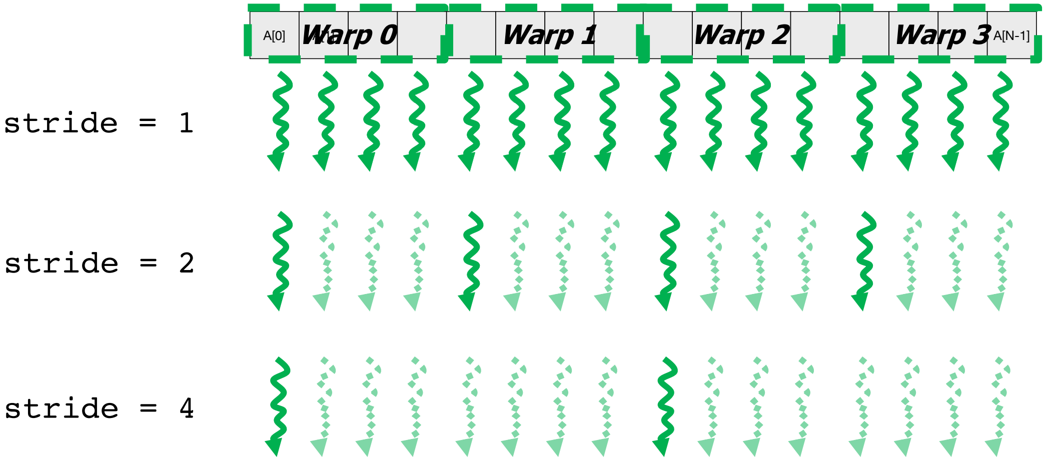


Slide credit: Hwu & Kirk

Vector Reduction: Naïve Mapping (II)

- Program with low SIMD utilization

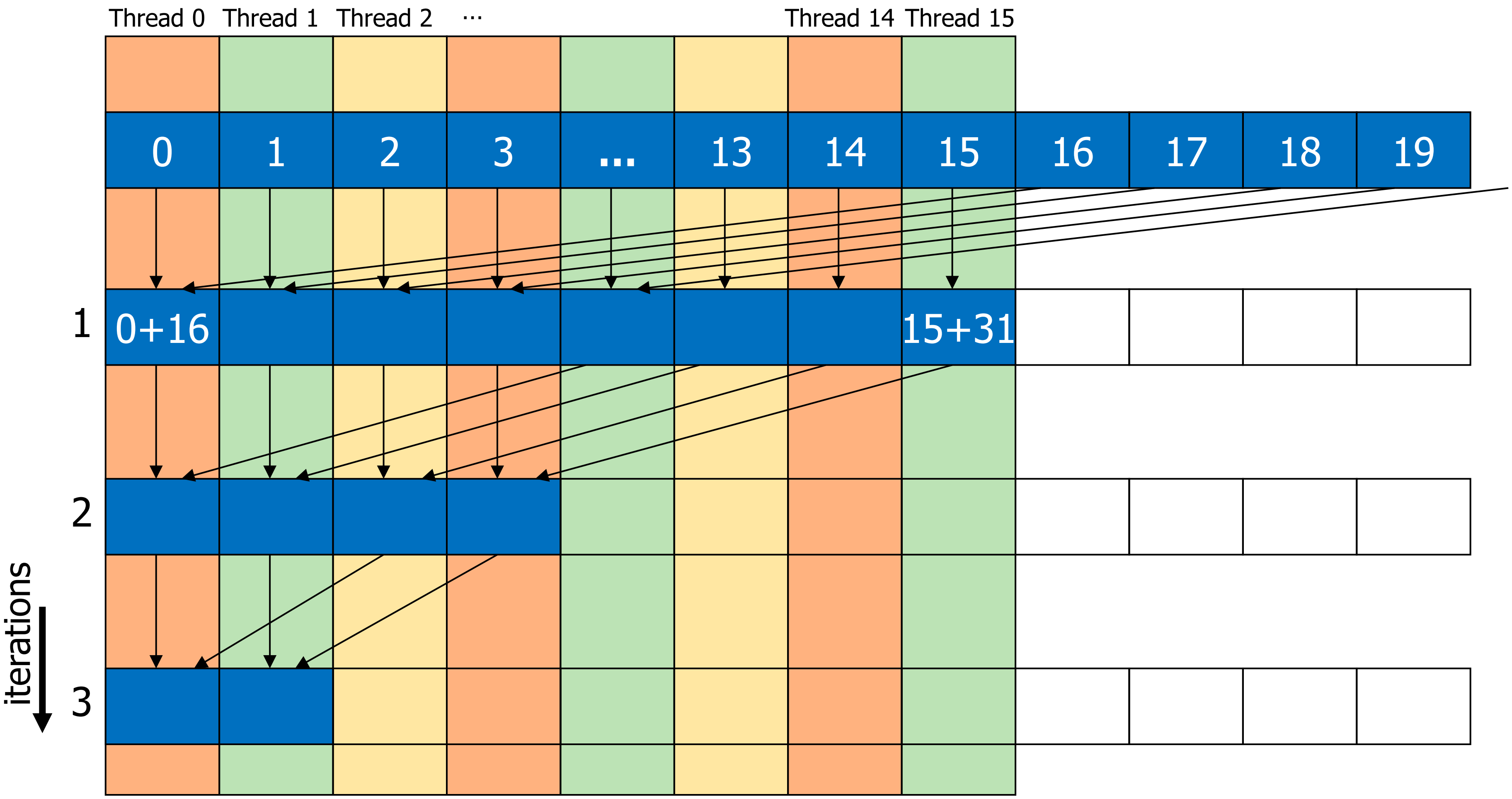
```
__shared__ float partialSum[]

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?

Divergence-Free Mapping (I)

- All active threads belong to the same warp

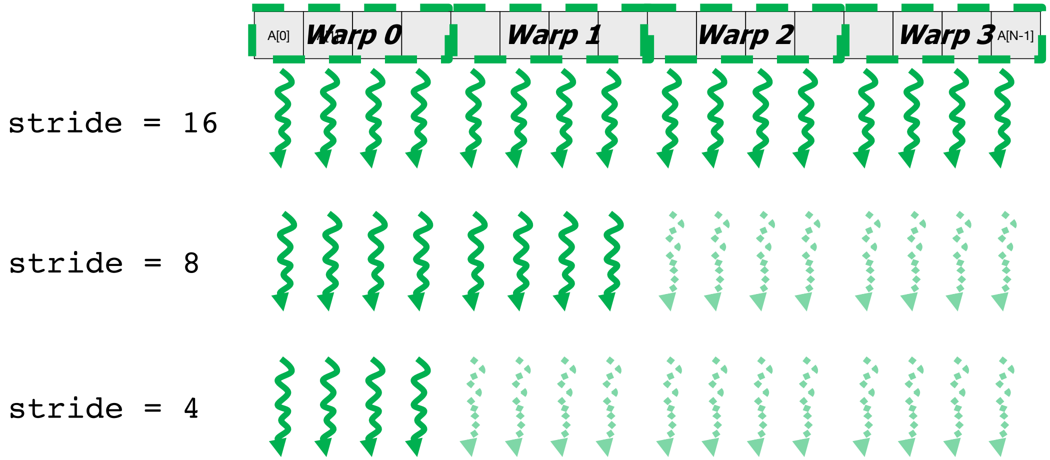


Slide credit: Hwu & Kirk

Divergence-Free Mapping (II)

- Program with high SIMD utilization

```
__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

Slide credit: Izzat El Hajj

Reduction with Warp Shuffle

```
__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;
    }
}
```

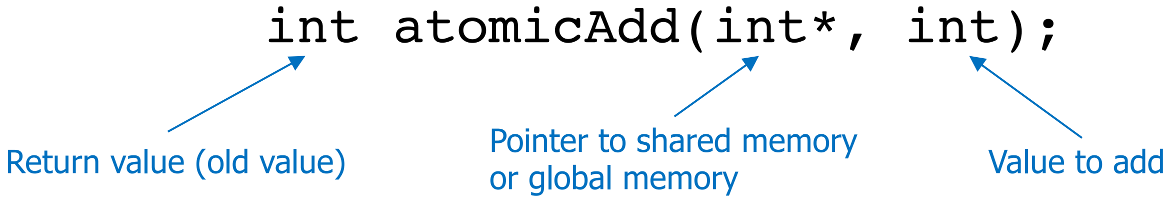
Slide credit: Izzat El Hajj

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



```
int atomicAdd(int*, int);
```

Return value (old value)

Pointer to shared memory or global memory

Value to add

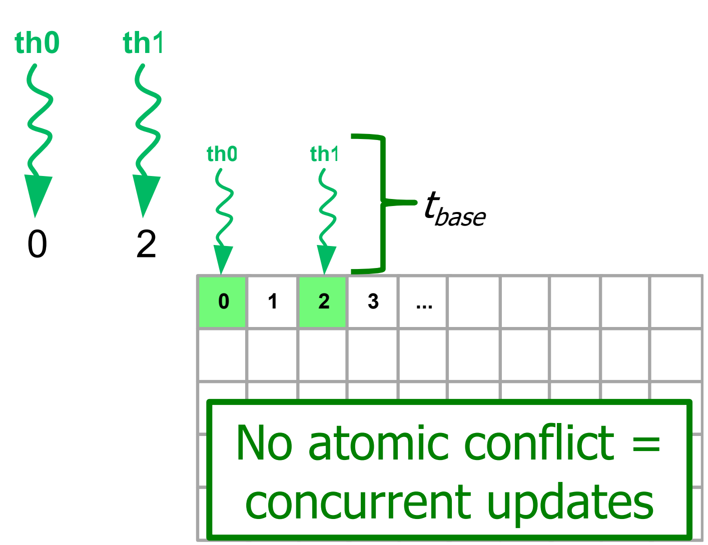
- Bitwise functions
  - And, or, xor

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

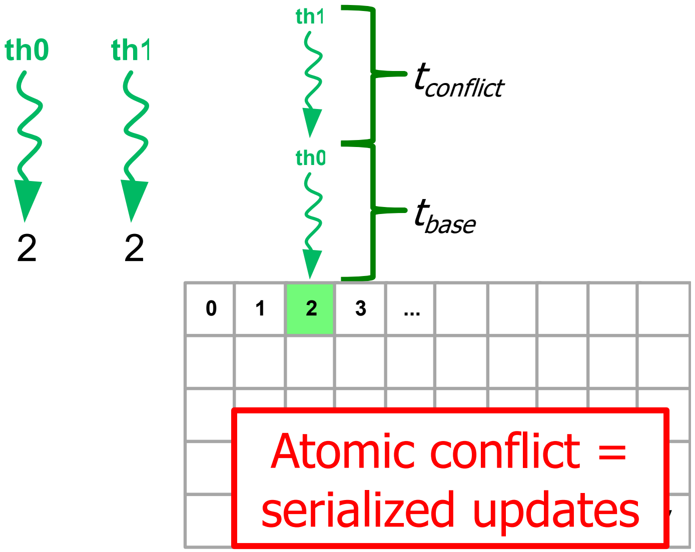
* Datatypes for different atomic operations in https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

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:

Thread A

```
oldVal = bins[b]
newVal = oldVal + 1
bins[b] = newVal
```

Thread B

```
oldVal = bins[b]
newVal = oldVal + 1
bins[b] = newVal
```

  - If both threads have the same `b` and `bins[b]` is initially 0, the final value of `bins[b]` could be 2 or 1

Slide credit: Izzat El Hajj

Data Races Example (I)

Time	Thread A	Thread B
1	`oldVal = bins[b]`	
2	`newVal = oldVal + 1`	
3	`bins[b] = newVal`	
4		`oldVal = bins[b]`
5		`newVal = oldVal + 1`
6		`bins[b] = newVal`

In these two scenarios, the final value of bins[b] will be 2

Time	Thread A	Thread B
1		`oldVal = bins[b]`
2		`newVal = oldVal + 1`
3		`bins[b] = newVal`
4	`oldVal = bins[b]`	
5	`newVal = oldVal + 1`	
6	`bins[b] = newVal`	

Slide credit: Izzat El Hajj

Data Races Example (II)

Time	Thread A	Thread B
1	oldVal = bins[b]	
2	newVal = oldVal + 1	
3		oldVal = bins[b]
4	bins[b] = newVal	
5		newVal = oldVal + 1
6		bins[b] = newVal

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

Time	Thread A	Thread B
1		oldVal = bins[b]
2		newVal = oldVal + 1
3	oldVal = bins[b]	
4		bins[b] = newVal
5	newVal = oldVal + 1	
6	bins[b] = newVal	

Slide credit: Izzat El Hajj

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:

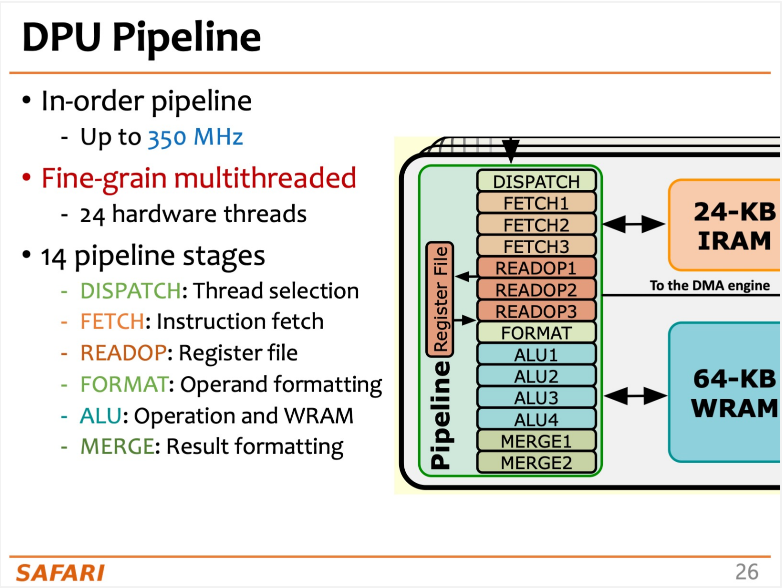
```
mutex_lock(lock);

++bins[b];

mutex_unlock(lock);
```

Using locks with SIMD execution may cause deadlock

Slide credit: Izzat El Hajj

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 350 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

SAFARI 26

Processing-in-Memory Course
Meeting 2: Real-world PIM architectures (Fall 2021)
https://youtu.be/D8Hjy2iU9l4

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

Juan Gómez-Luna[1] Izzat El Hajj[2] Ivan Fernandez[1,3] Christina Giannoula[1,4]
Geraldo F. Oliveira[1] Onur Mutlu[1]

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

https://arxiv.org/pdf/2105.03814.pdf

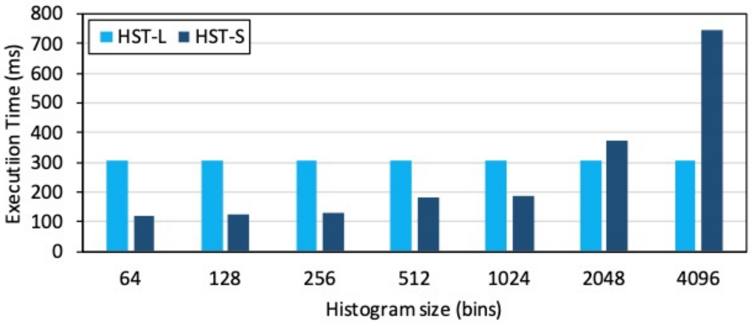


Figure 20: Execution times (ms) of two versions of histogram (HST-L, HST-S) on 1 DPU.

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

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

Maxwell, Pascal, Volta…

```
/*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

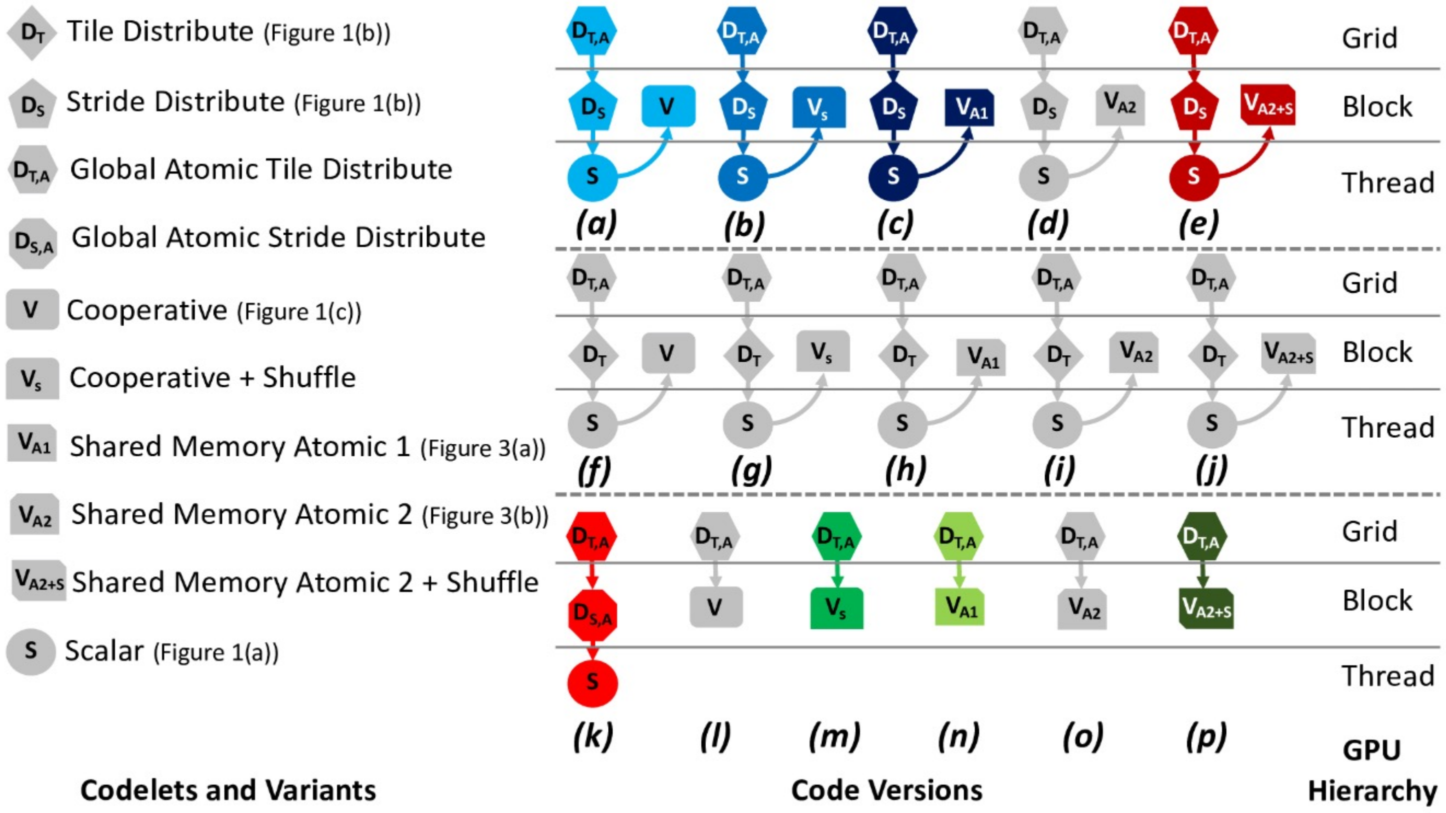
https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-parallel-reduction

Harris, "Optimizing Parallel Reduction in CUDA," https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

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

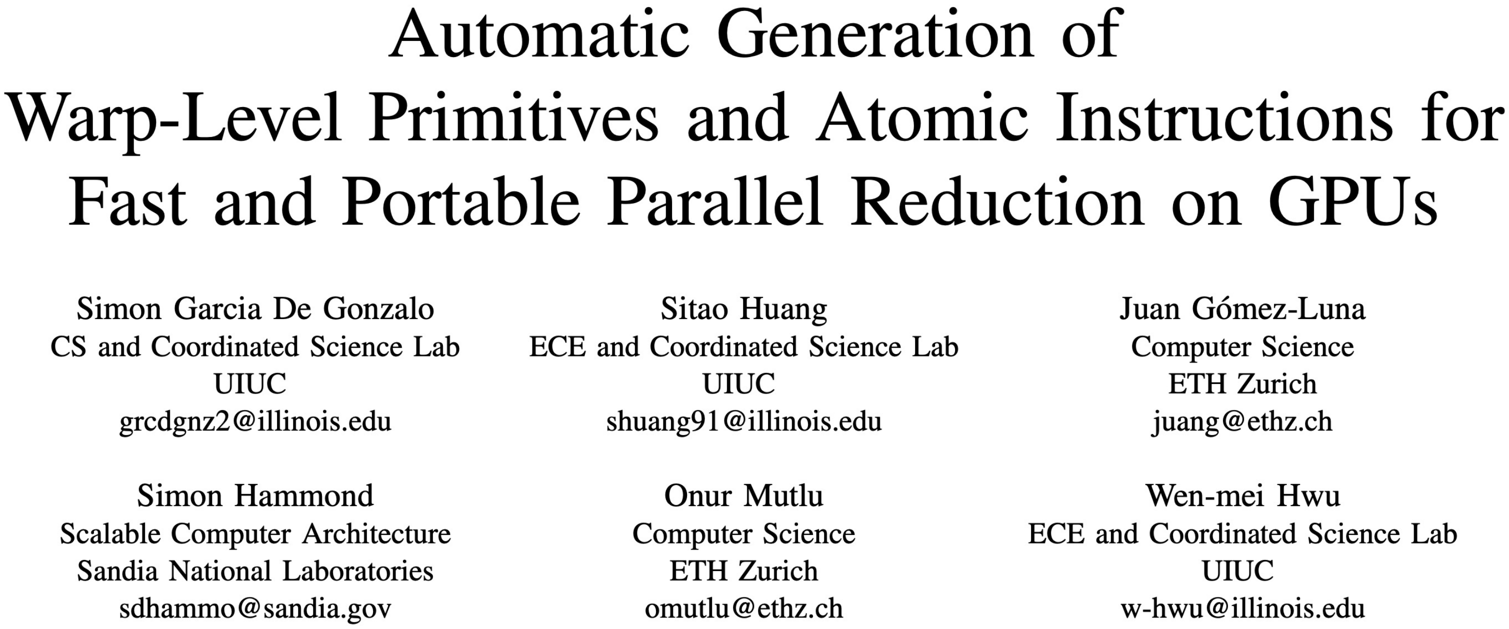

D_T Tile Distribute (Figure 1(b))
D_S Stride Distribute (Figure 1(b))
D_T,A Global Atomic Tile Distribute
D_S,A Global Atomic Stride Distribute
V Cooperative (Figure 1(c))
V_s Cooperative + Shuffle
V_A1 Shared Memory Atomic 1 (Figure 3(a))
V_A2 Shared Memory Atomic 2 (Figure 3(b))
V_A2+S Shared Memory Atomic 2 + Shuffle
S Scalar (Figure 1(a))
Codelets and Variants
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
(o)
(p)
Code Versions
Grid
Block
Thread
GPU Hierarchy

Over 85 different versions possible!

Garcia de Gonzalo et al., "Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs," CGO 2019

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"** *Proceedings of the International Symposium on Code Generation and Optimization (**CGO**)*, Washington, DC, USA, February 2019. [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

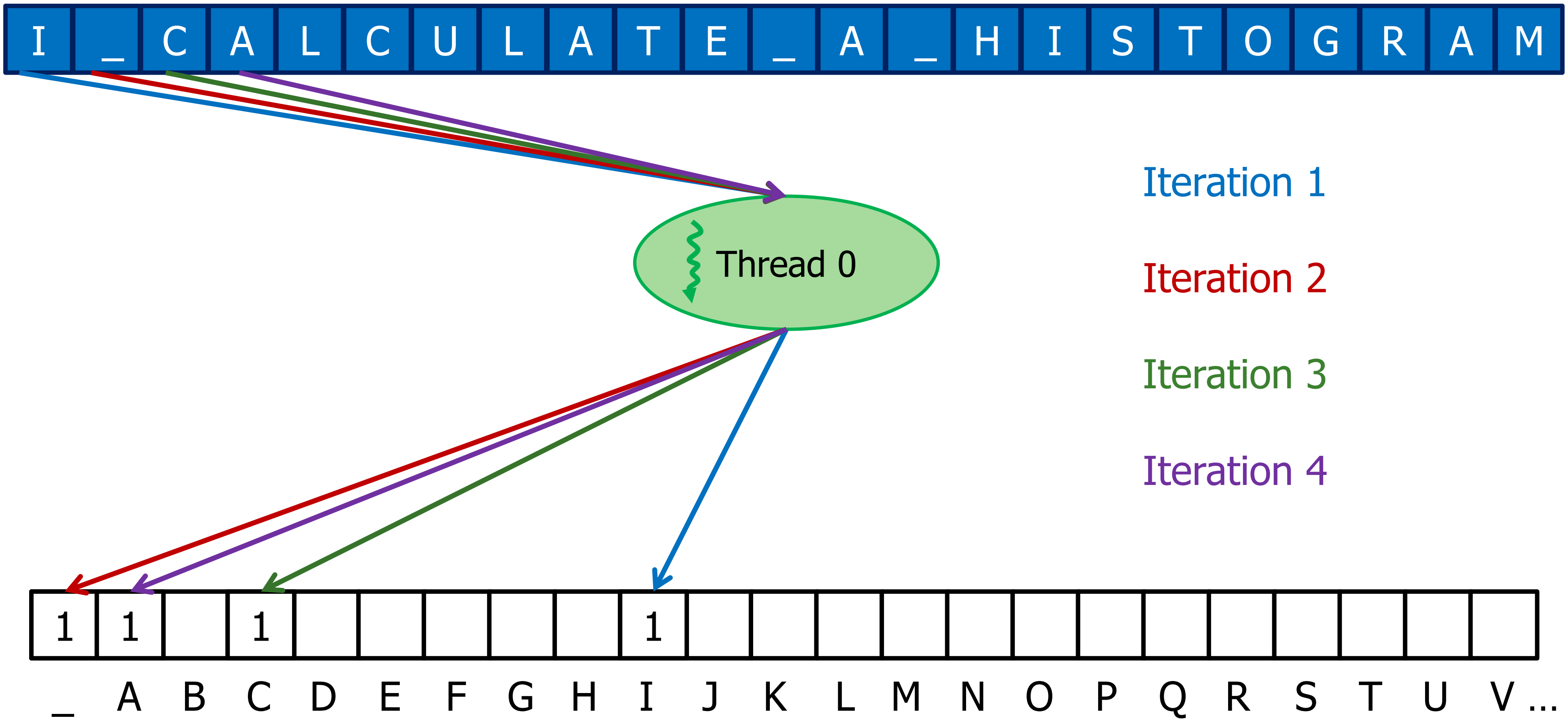
Histogram Computation

- 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

Slide credit: Hwu & Kirk

Sequential Histogram Computation

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



I	_	C	A	L	C	U	L	A	T	E	_	A	_	H	I	S	T	O	G	R	A	M

Thread 0

Iteration 1

Iteration 2

Iteration 3

Iteration 4

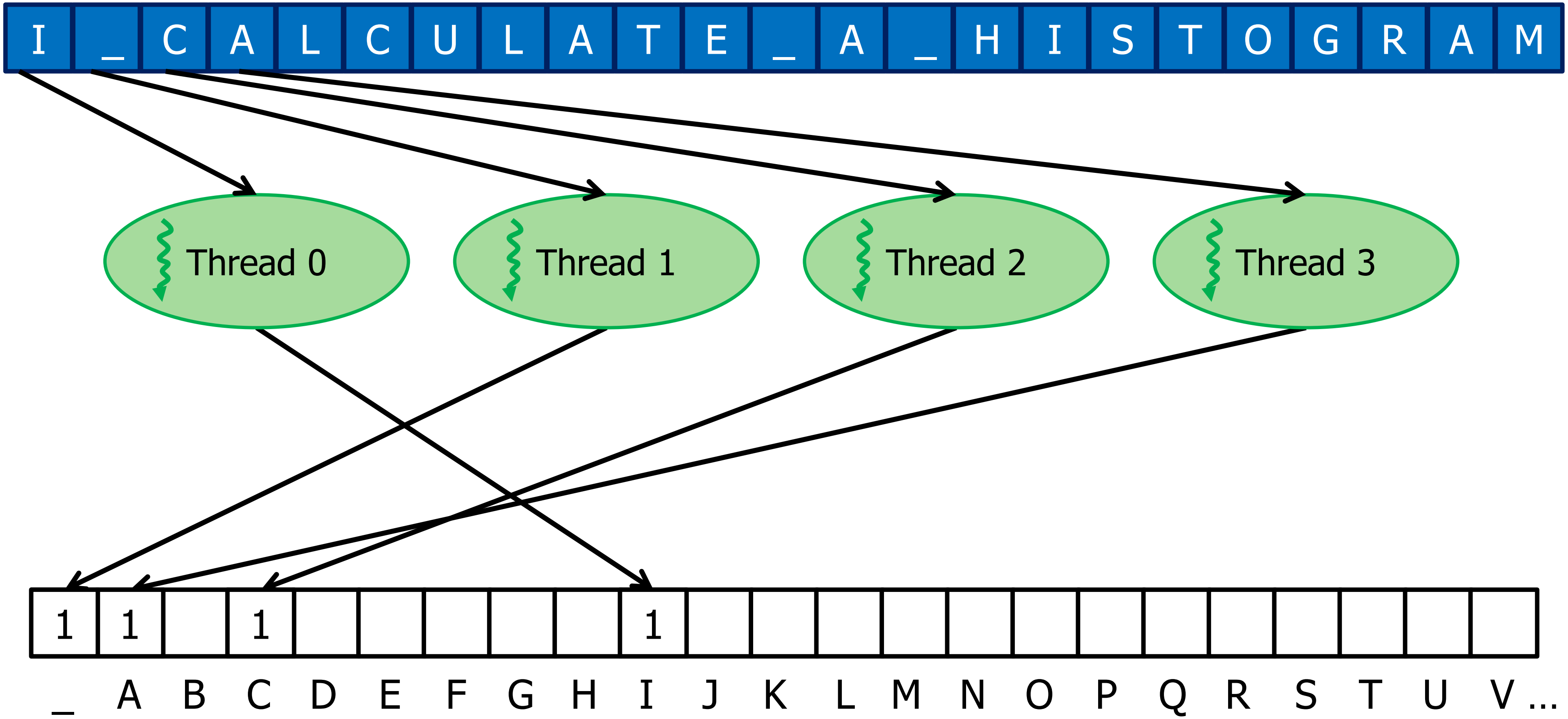
1	1		1						1													
_	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V …

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

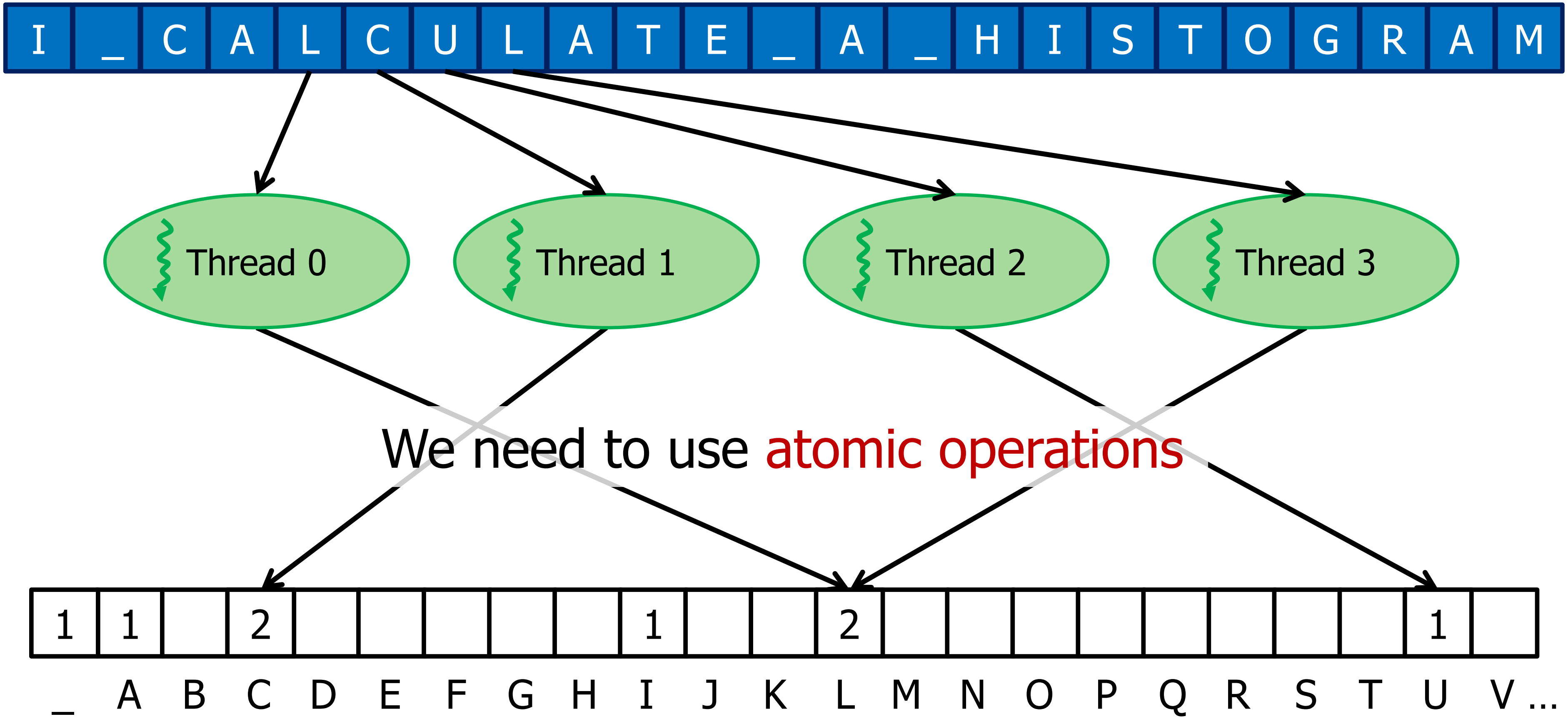

I _ C A L C U L A T E _ A _ H I S T O G R A M
Thread 0
Thread 1
Thread 2
Thread 3
1
1
1
1
_ A B C D E F G H I J K L M N O P Q R S T U V …

(Wrong) Parallel Histogram Kernel

```
__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 threadID + #threads



I	_	C	A	L	C	U	L	A	T	E	_	A	_	H	I	S	T	O	G	R	A	M

We need to use atomic operations

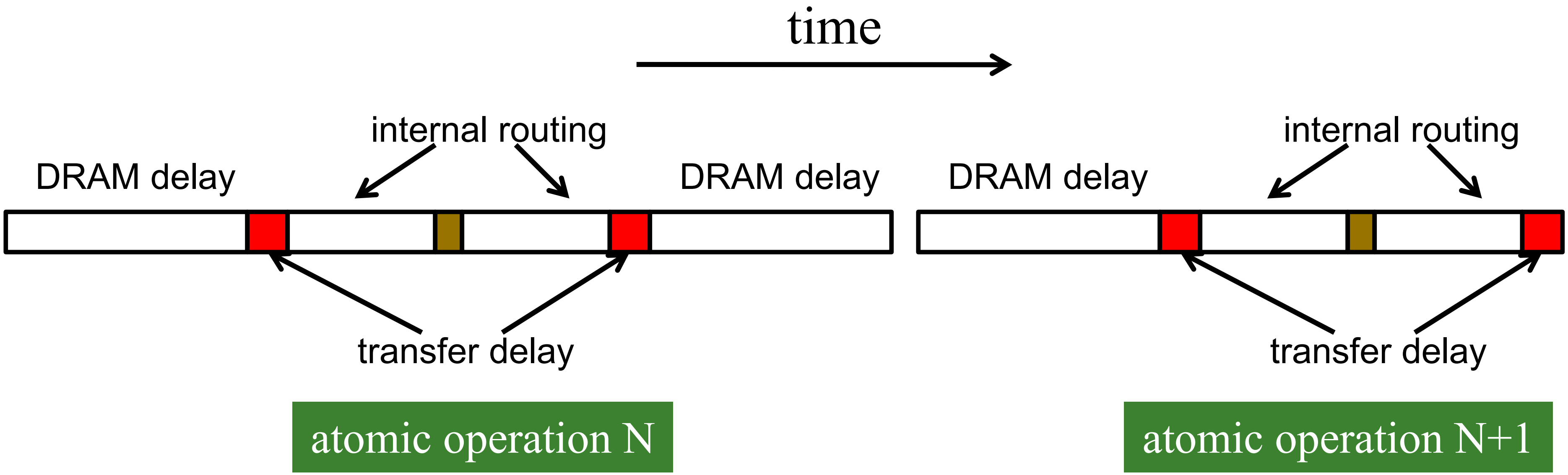
1	1		2						1			2									1	
_	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V …

(Correct) Parallel Histogram Kernel

```
__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

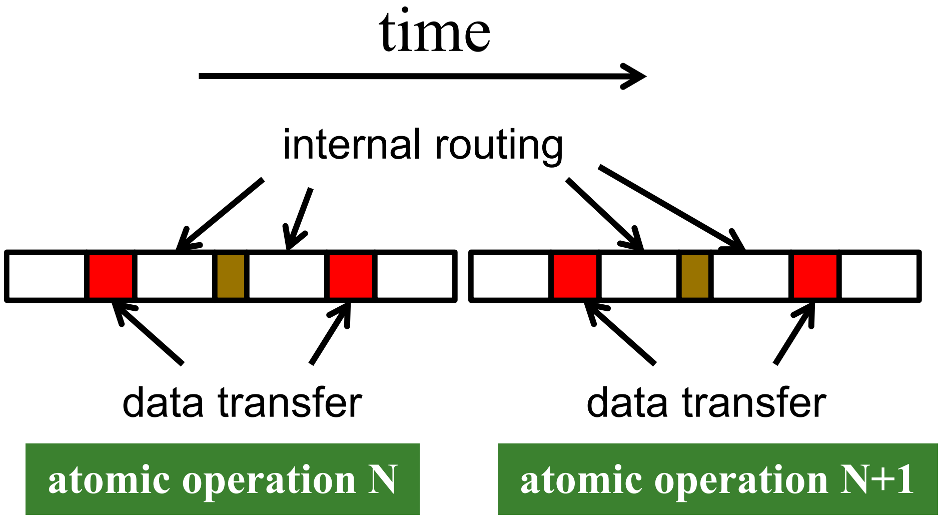
- Each Load-Modify-Store has two full memory access delays
  - All atomic operations on the same variable (RAM location) are serialized



Slide credit: Hwu & Kirk

Hardware Improvements

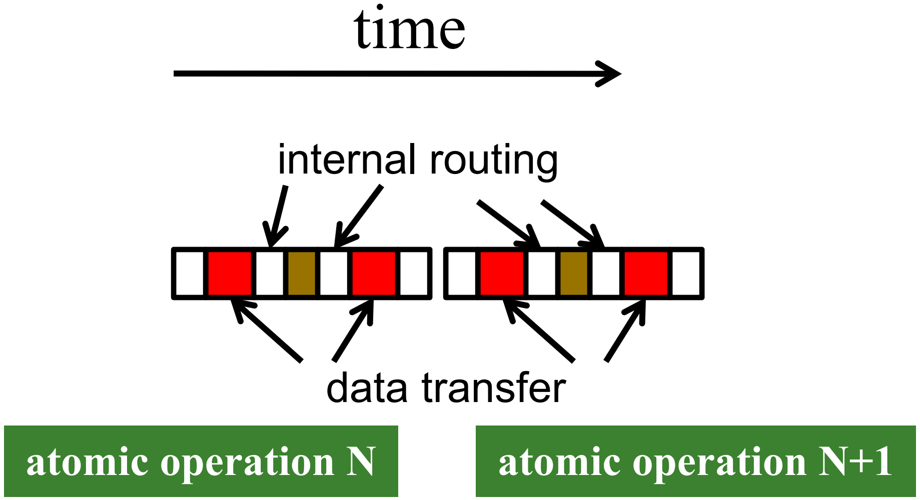
- Atomic operations on Fermi L2 cache
  - Medium latency, but still serialized
  - Global to all blocks
  - "Free improvement" on Global Memory atomics



Slide credit: Hwu & Kirk

Hardware Improvements (Cont.)

- Atomic operations on Shared Memory
  - Very short latency, but still serialized
  - Private to each thread block
  - Need algorithm work by programmers (more later)



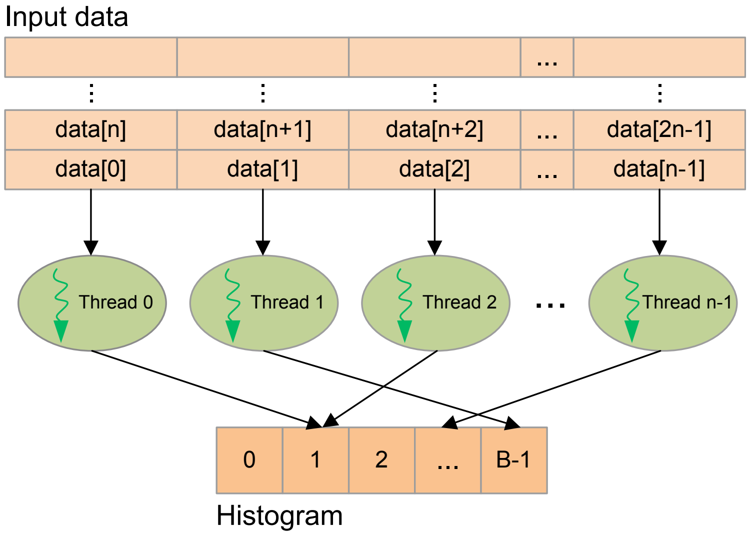
Slide credit: Hwu & Kirk

Image Histogram

- Histograms are widely used in image processing
  - Some computation before voting in the histogram may be needed

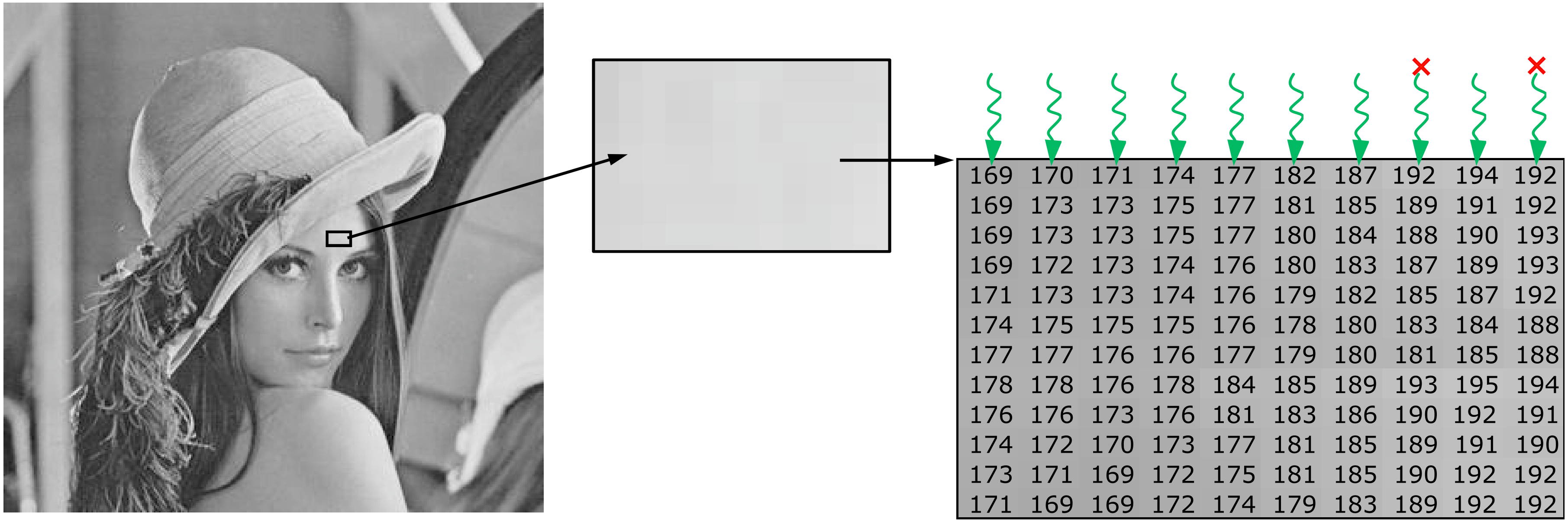
```
For (each pixel i in image I){
    Pixel = I[i]                      // Read pixel
    Pixel’ = Computation(Pixel)       // Optional computation
    Histogram[Pixel’]++               // Vote in histogram bin
}
```

  - Parallel threads frequently incur atomic conflicts in image histogram computation



Histogram Computation of Natural Images

- Frequent atomic conflicts due to the spatial similarity of the pixel value distribution in natural images



169	170	171	174	177	182	187	192	194	192
169	173	173	175	177	181	185	189	191	192
169	173	173	175	177	180	184	188	190	193
169	172	173	174	176	180	183	187	189	193
171	173	173	174	176	179	182	185	187	192
174	175	175	175	176	178	180	183	184	188
177	177	176	176	177	179	180	181	185	188
178	178	176	178	184	185	189	193	195	194
176	176	173	176	181	183	186	190	192	191
174	172	170	173	177	181	185	189	191	190
173	171	169	172	175	181	185	190	192	192
171	169	169	172	174	179	183	189	192	192

- 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

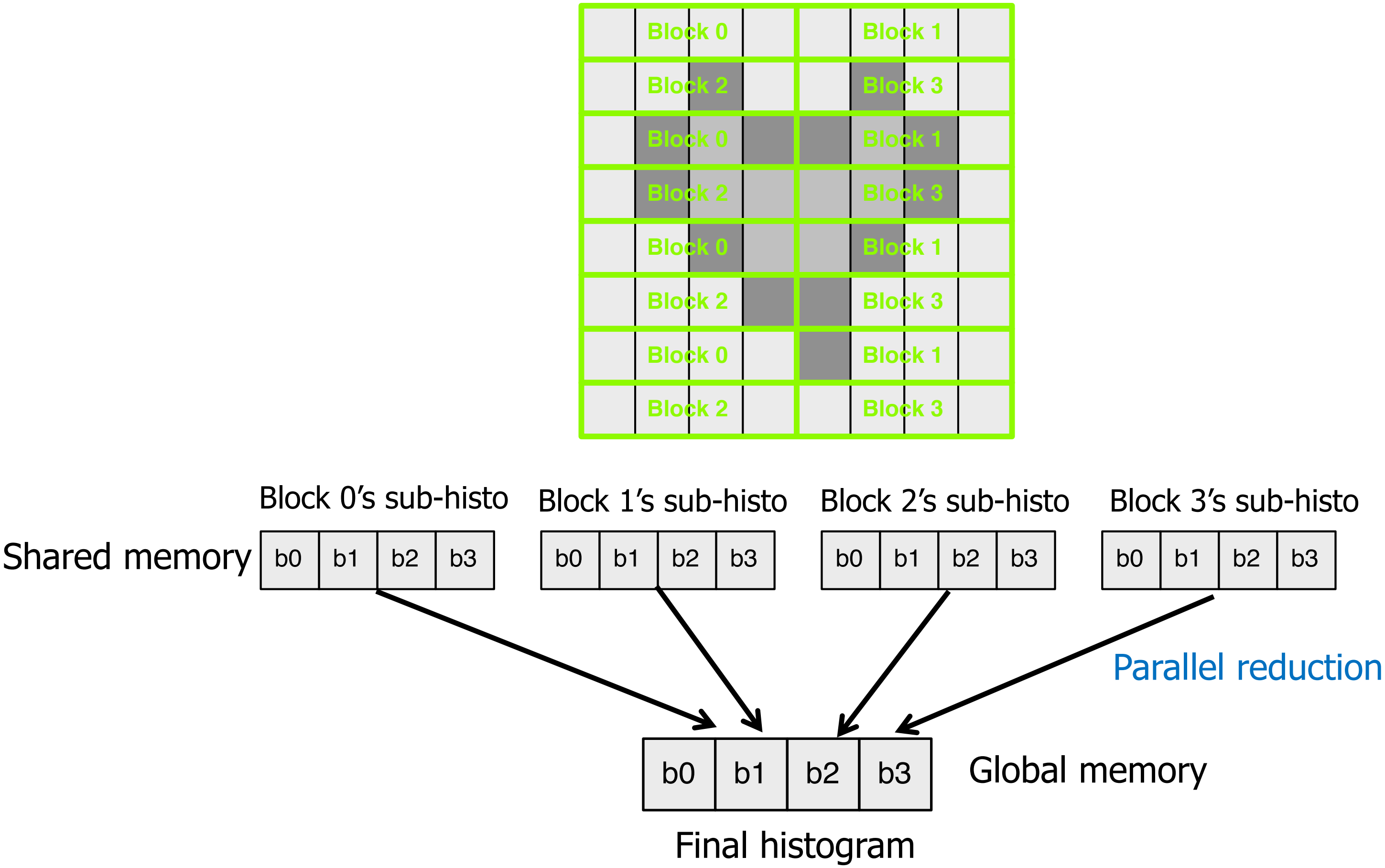
- Privatization is an optimization technique where multiple private copies of an output are maintained, then the global copy is updated on completion
    - Operations on the output must be associative and commutative because the order of updates has changed

- 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

Slide credit: Izzat El Hajj

Histogram Privatization

- Privatization: Per-block sub-histograms in shared memory
  - Threads use atomic operations in shared memory



Block 0	Block 1
Block 2	Block 3
Block 0	Block 1
Block 2	Block 3
Block 0	Block 1
Block 2	Block 3
Block 0	Block 1
Block 2	Block 3

Block 0's sub-histo

Block 1's sub-histo

Block 2's sub-histo

Block 3's sub-histo

Shared memory

b0 b1 b2 b3 | b0 b1 b2 b3 | b0 b1 b2 b3 | b0 b1 b2 b3

Parallel reduction

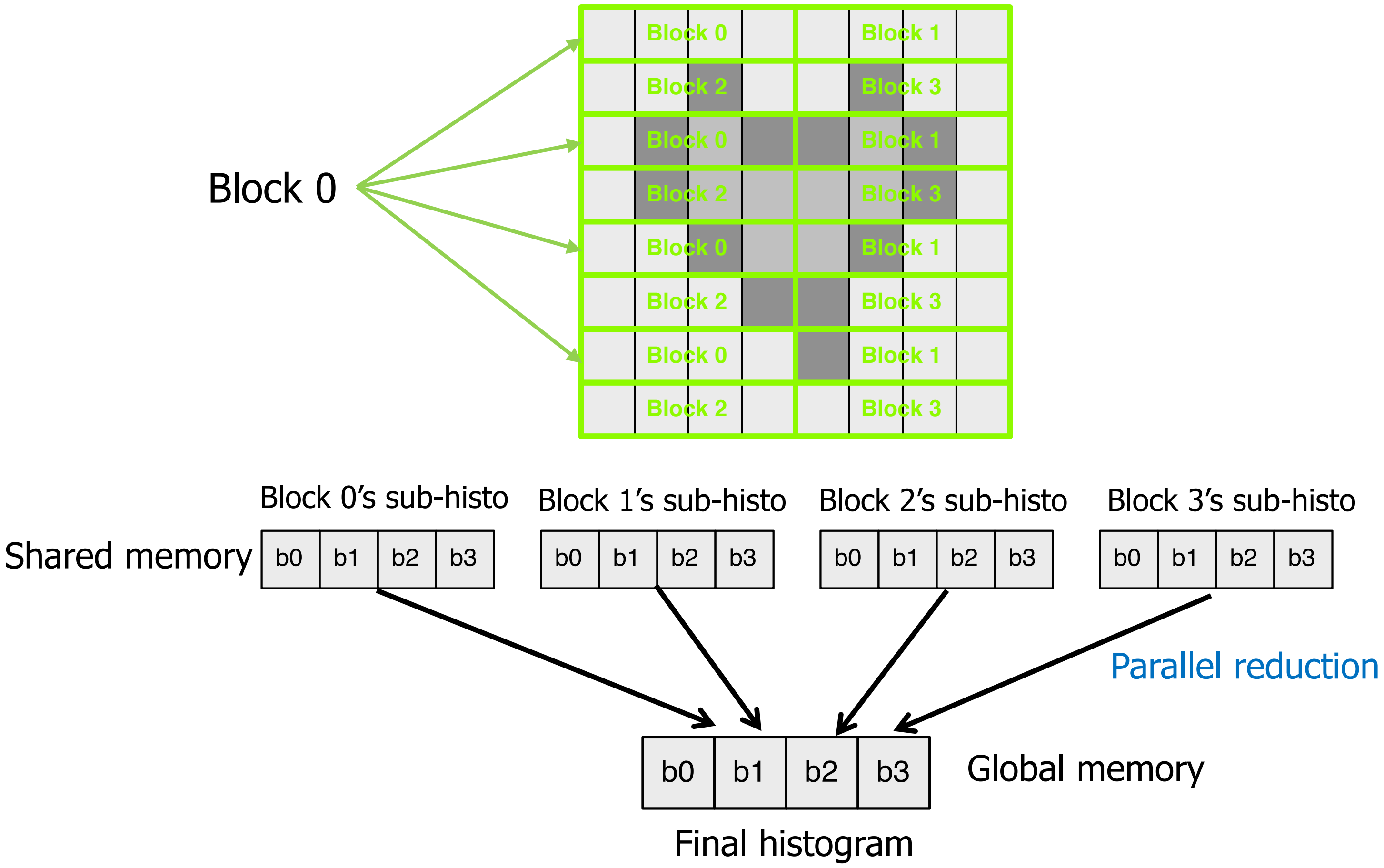
b0 b1 b2 b3

Global memory

Final histogram

Histogram Privatization + Coarsening

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



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

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

Slide credit: Hwu & Kirk (PUMPS 2021)

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

```
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;
}
```

Slide credit: Hwu & Kirk (PUMPS 2021)

Evolution of the Architectural Support for Atomic Operations

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:

Tesla, Fermi, Kepler

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

Maxwell, Pascal, Volta…

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

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

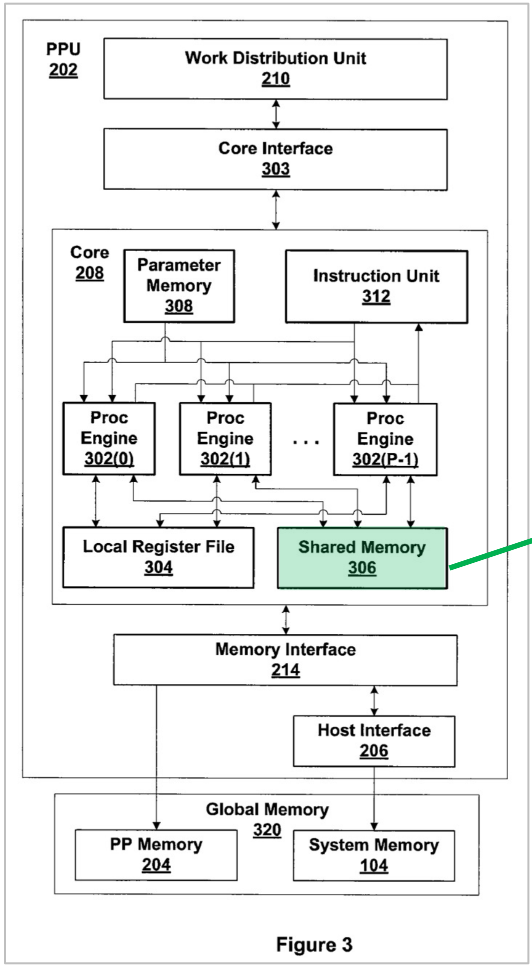


Figure 3

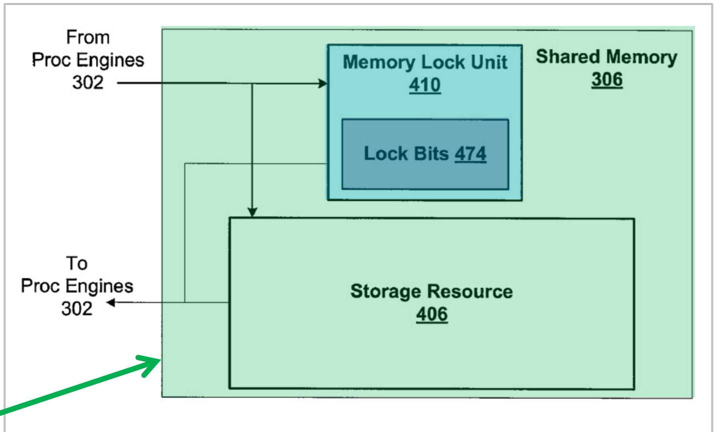
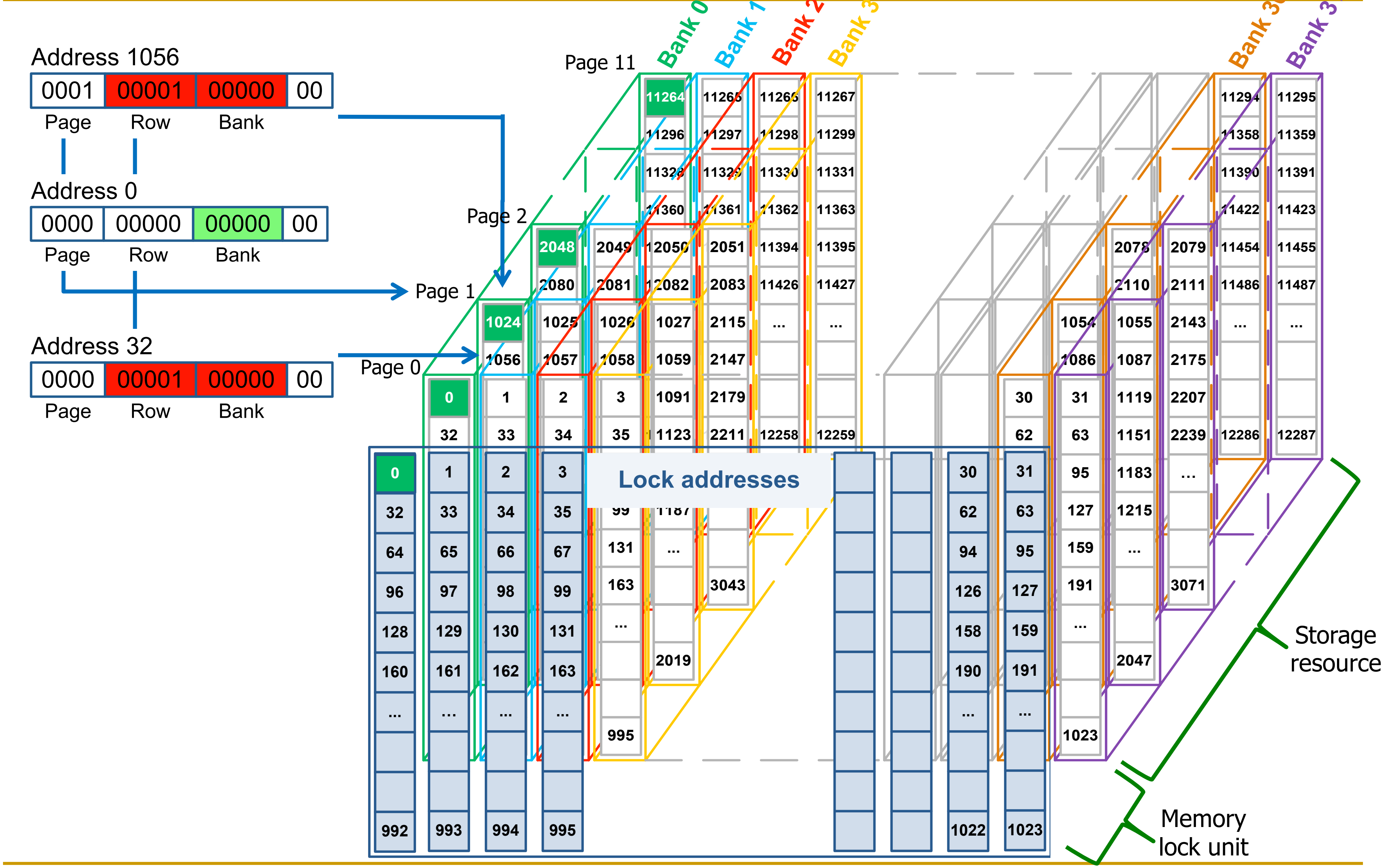


Figure 4B

```
/*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	00001	00000	00
Page	Row	Bank	

Lock addresses

Storage resource

Memory lock unit

Gomez-Luna+, "Performance Modeling of Atomic Additions on GPU Scratchpad Memory," IEEE TPDS, 2013

Assembly Code for Shared Memory Atomics

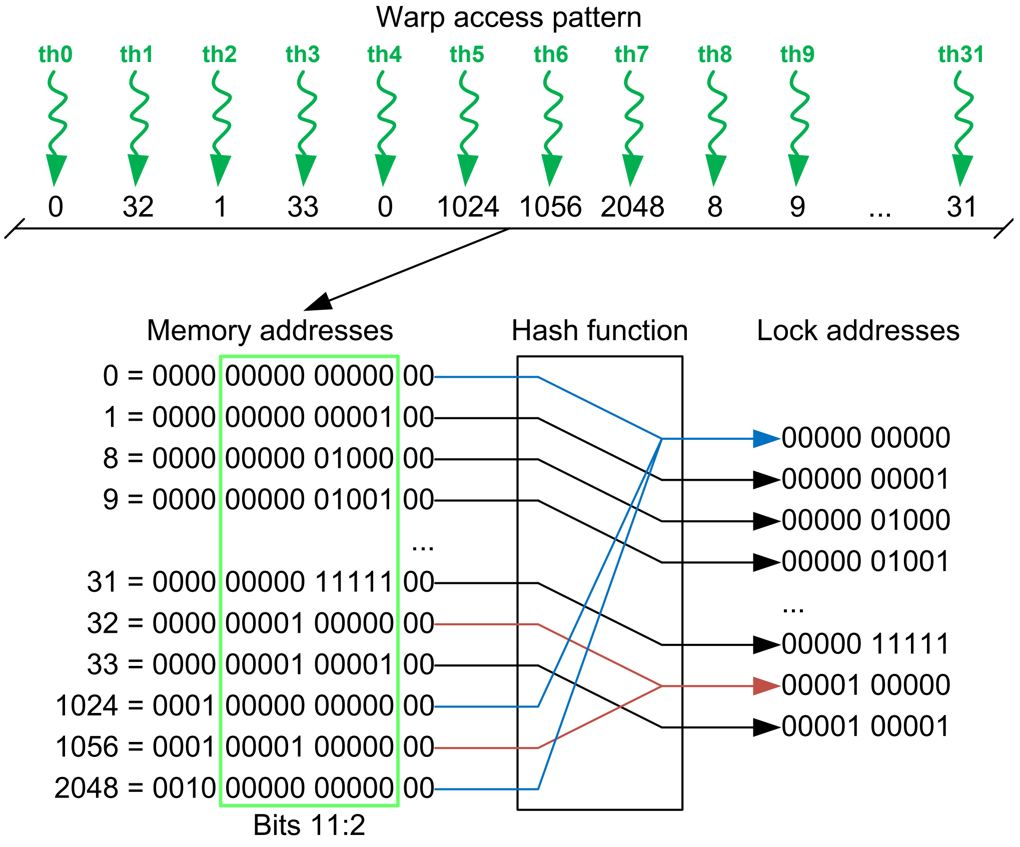
(pre Maxwell)

- Lock-free mechanism
  - Predicated execution
- LDSLK loads from shared memory and sets one lock bit
  - Predication register P0 set if lock succeeds
- STSCUL stores and releases the lock
- BRA jumps to start a new attempt to acquire the lock

```
/*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*/          ...
```

Limited Number of Lock Bits

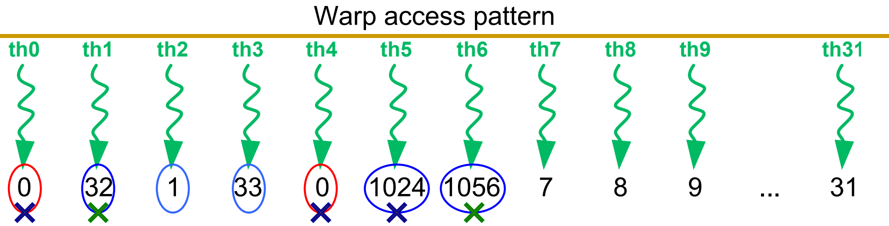
- The limited number of locks may cause high contention
  - 256 lock bits in Tesla, 1024 lock bits in Fermi and Kepler

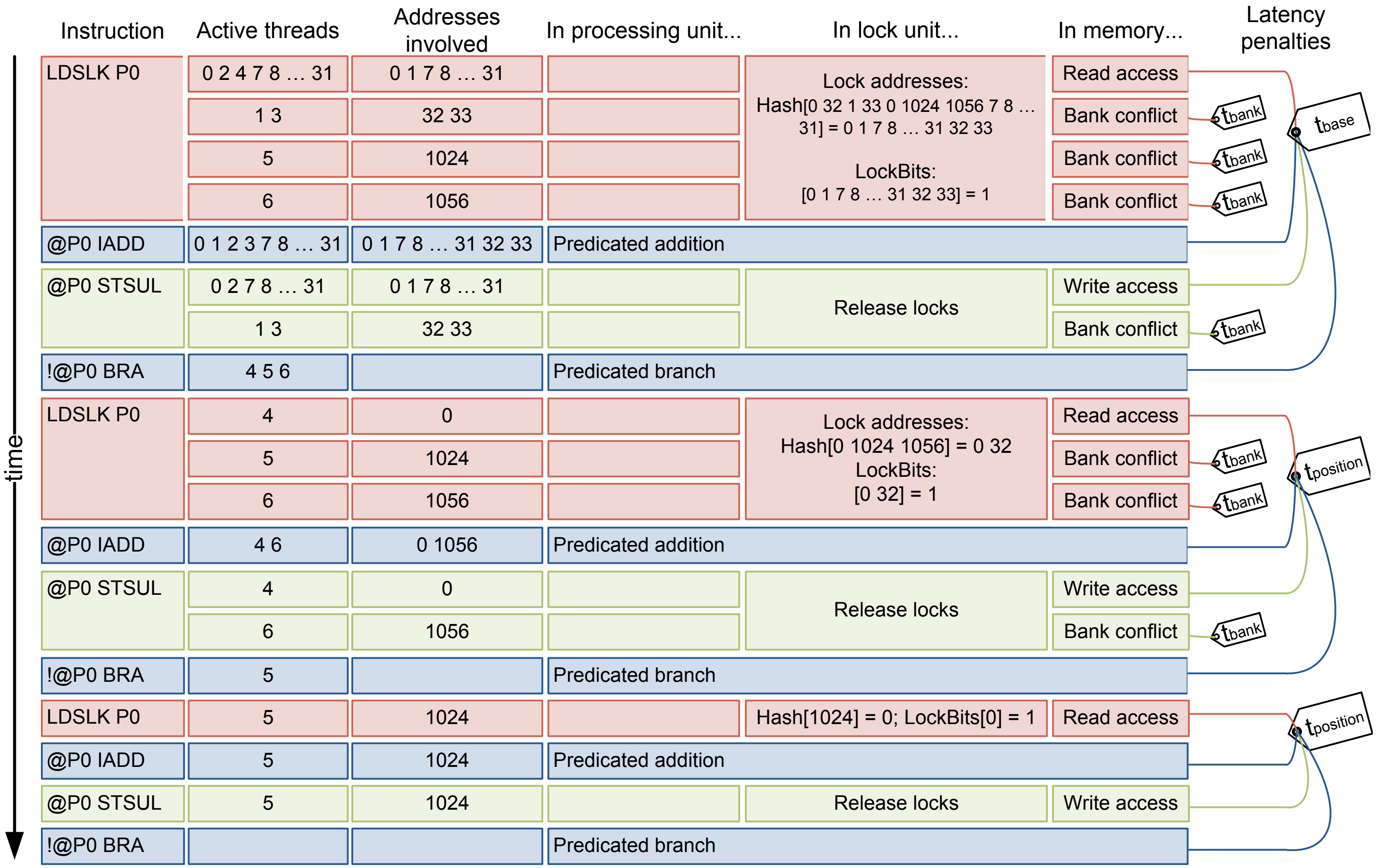


Gomez-Luna+, "Performance Modeling of Atomic Additions on GPU Scratchpad Memory," IEEE TPDS, 2013 (supplemental material)

Example Execution Timeline

```
/*0210*/   LDSLK P0, R7, [R9];
/*0218*/   @P0 IADD R10, R7, 0x1;
/*0220*/   @P0 STSUL [R9], R10;
/*0228*/   @!P0 BRA 0x210;
```





Instruction	Active threads	Addresses involved	In processing unit...	In lock unit...	In memory...	Latency penalties
LDSLK P0	0 2 4 7 8 … 31	0 1 7 8 … 31		Lock addresses: Hash[0 32 1 33 0 1024 1056 7 8 … 31] = 0 1 7 8 … 31 32 33 LockBits: [0 1 7 8 … 31 32 33] = 1	Read access	t_{base}
	1 3	32 33			Bank conflict	t_{bank}
	5	1024			Bank conflict	t_{bank}
	6	1056			Bank conflict	t_{bank}
@P0 IADD	0 1 2 3 7 8 … 31	0 1 7 8 … 31 32 33	Predicated addition			
@P0 STSUL	0 2 7 8 … 31	0 1 7 8 … 31		Release locks	Write access	
	1 3	32 33			Bank conflict	t_{bank}
!@P0 BRA	4 5 6		Predicated branch			
LDSLK P0	4	0		Lock addresses: Hash[0 1024 1056] = 0 32 LockBits: [0 32] = 1	Read access	$t_{position}$
	5	1024			Bank conflict	t_{bank}
	6	1056			Bank conflict	t_{bank}
@P0 IADD	4 6	0 1056	Predicated addition			
@P0 STSUL	4	0		Release locks	Write access	
	6	1056			Bank conflict	t_{bank}
!@P0 BRA	5		Predicated branch			
LDSLK P0	5	1024		Hash[1024] = 0; LockBits[0] = 1	Read access	$t_{position}$
@P0 IADD	5	1024	Predicated addition			
@P0 STSUL	5	1024		Release locks	Write access	
!@P0 BRA			Predicated branch			

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

Microbenchmarking Atomic Operations

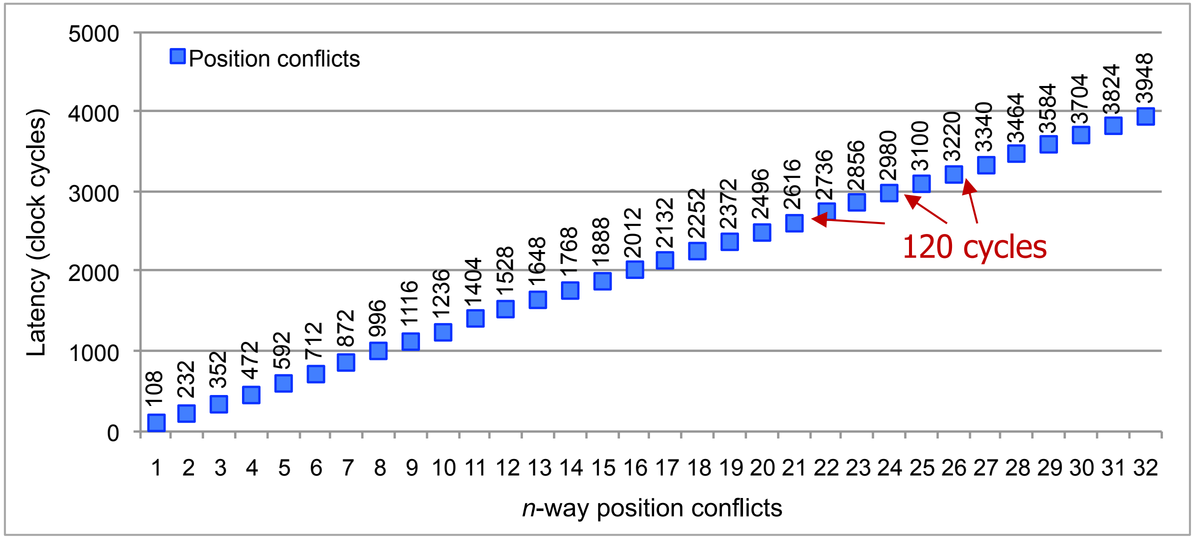
- Microbenchmark
  - `Hs` is a shared memory array
  - `clock()` returns the cycle count

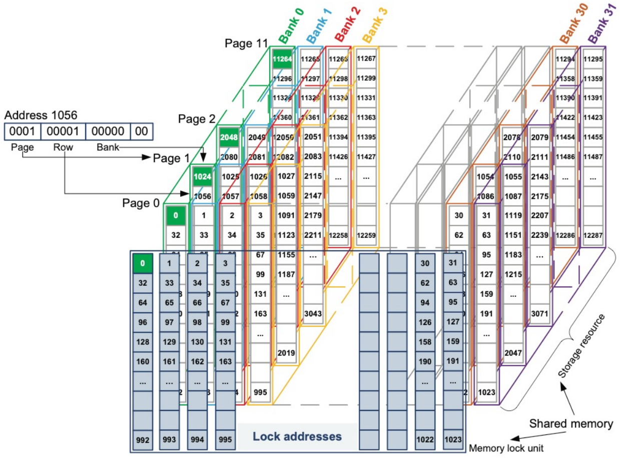
```
for(int i = begin; i < size; i += num_threads){

    // Read from global memory
    vote = input_data[i];

    start_time = clock(); // Start timing

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

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

Gomez-Luna+, "Performance Modeling of Atomic Additions on GPU Scratchpad Memory," IEEE TPDS, 2013

Microbenchmarking Results (I)

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

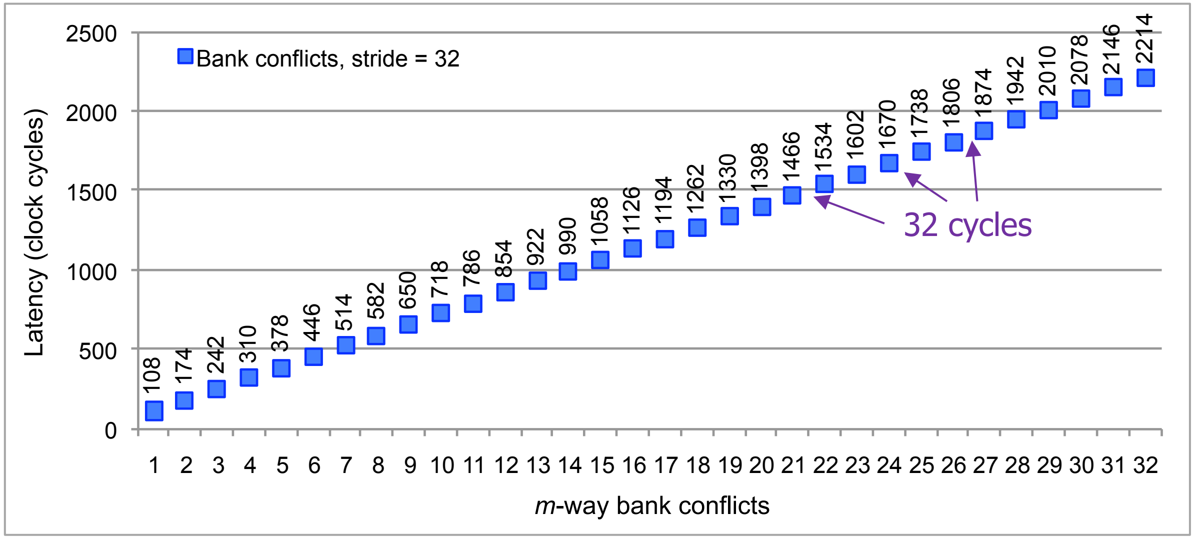


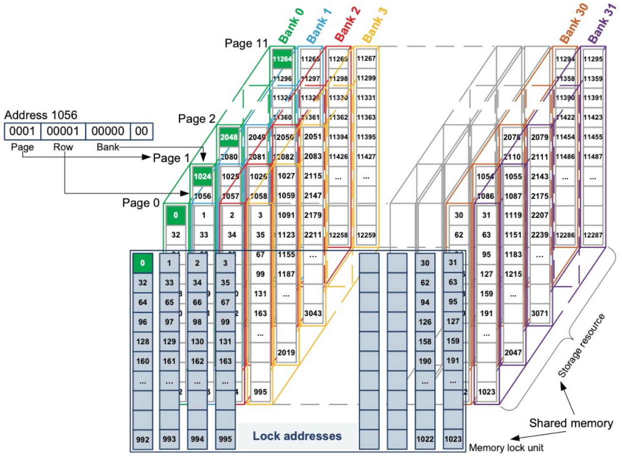


Gomez-Luna+, "Performance Modeling of Atomic Additions on GPU Scratchpad Memory," IEEE TPDS, 2013

Microbenchmarking Results (II)

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

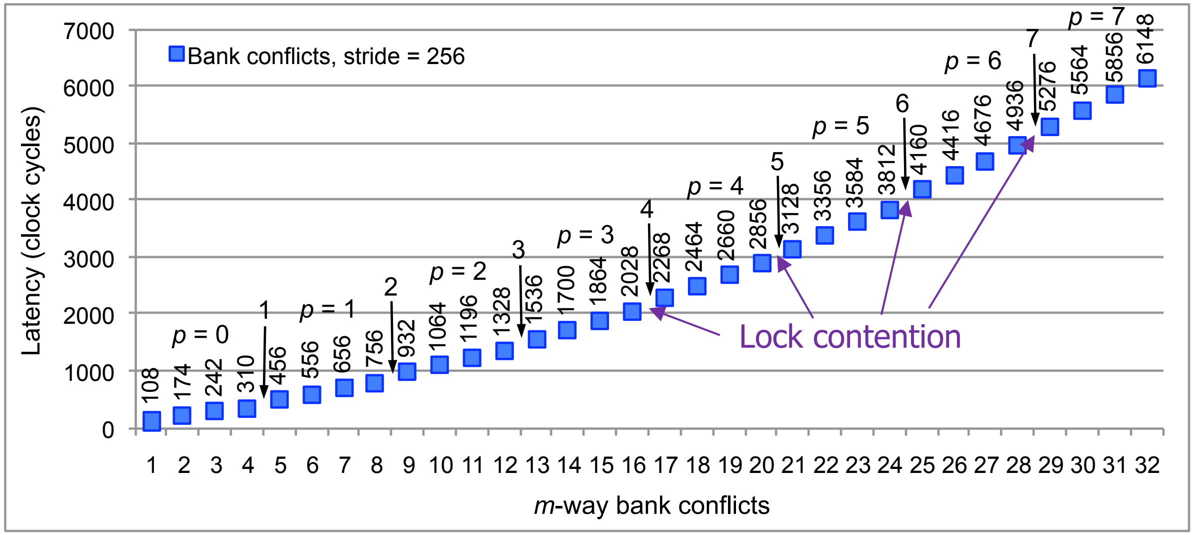


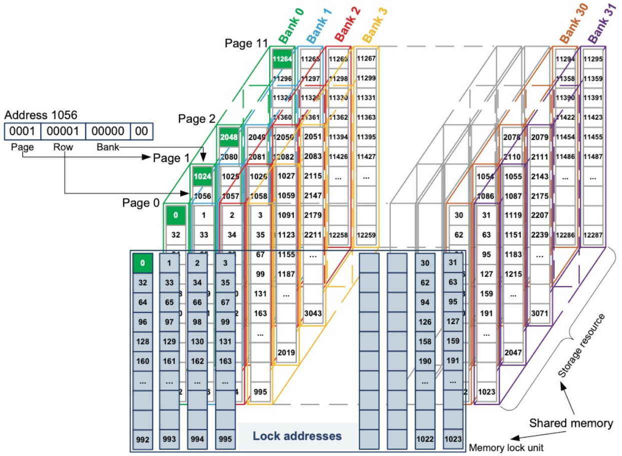


Gomez-Luna+, "Performance Modeling of Atomic Additions on GPU Scratchpad Memory," IEEE TPDS, 2013

Microbenchmarking Results (III)

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

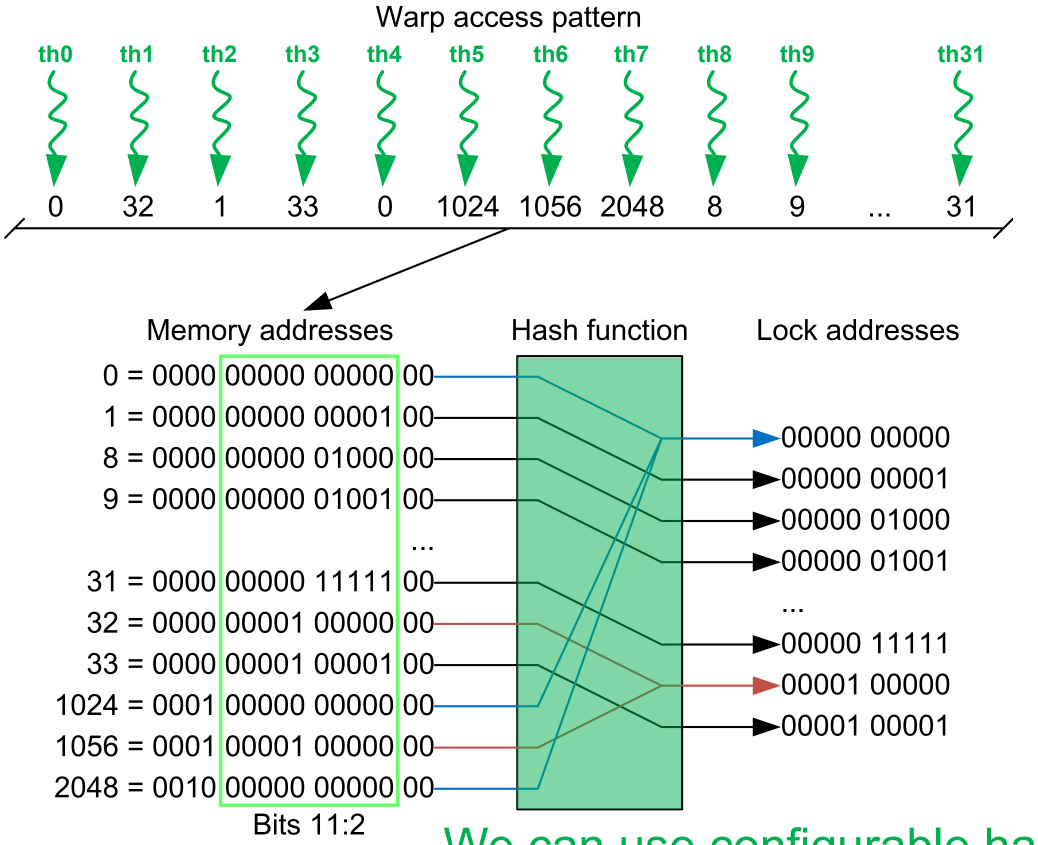




Gomez-Luna+, "Performance Modeling of Atomic Additions on GPU Scratchpad Memory," IEEE TPDS, 2013

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



Warp access pattern

th0 th1 th2 th3 th4 th5 th6 th7 th8 th9 th31

0 32 1 33 0 1024 1056 2048 8 9 ... 31

Memory addresses

0 = 0000 00000 00000 00
1 = 0000 00000 00001 00
8 = 0000 00000 01000 00
9 = 0000 00000 01001 00
...
31 = 0000 00000 11111 00
32 = 0000 00001 00000 00
33 = 0000 00001 00001 00
1024 = 0001 00000 00000 00
1056 = 0001 00001 00000 00
2048 = 0010 00000 00000 00

Bits 11:2

Hash function

Lock addresses

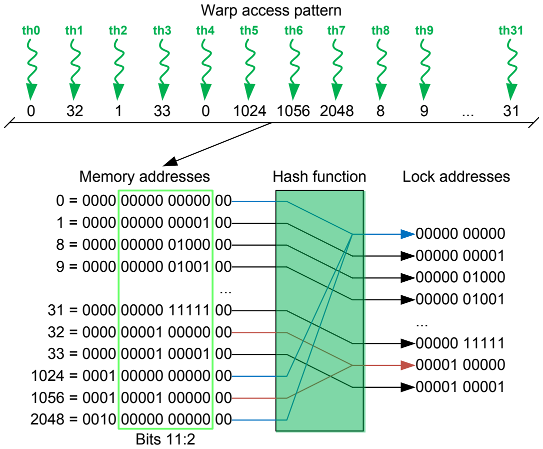
00000 00000
00000 00001
00000 01000
00000 01001
...
00000 11111
00001 00000
00001 00001

We can use configurable hash functions

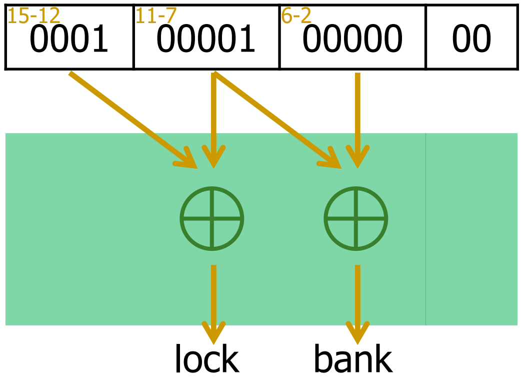
Gomez-Luna+, "Performance Modeling of Atomic Additions on GPU Scratchpad Memory," IEEE TPDS, 2013 (supplemental material)

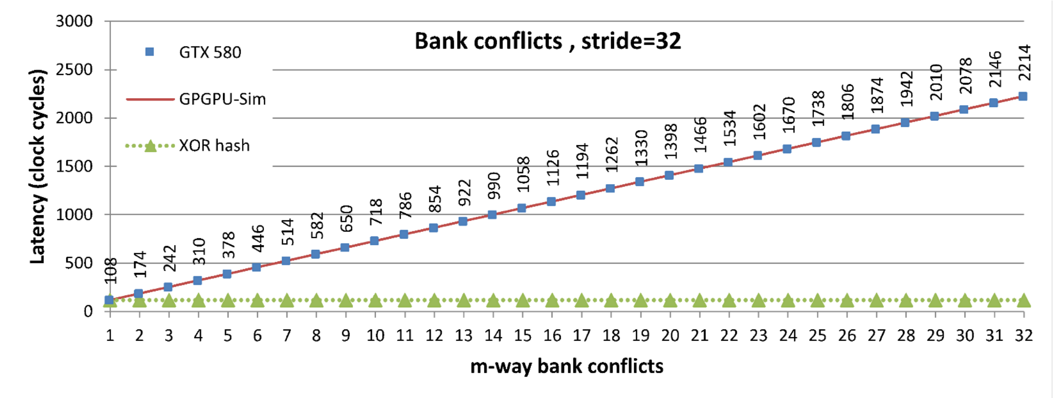
Configurable Hash Functions

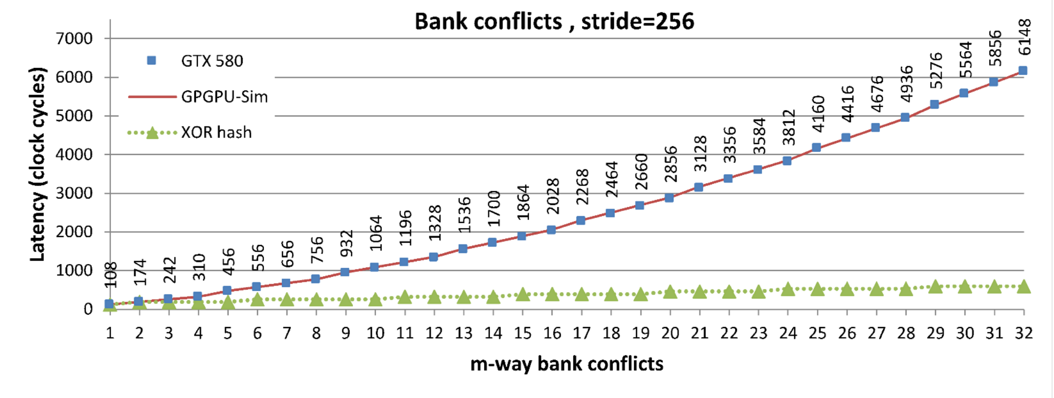
- Configurable hash functions can reduce the number of bank conflicts and lock conflicts



Address: 0x1080





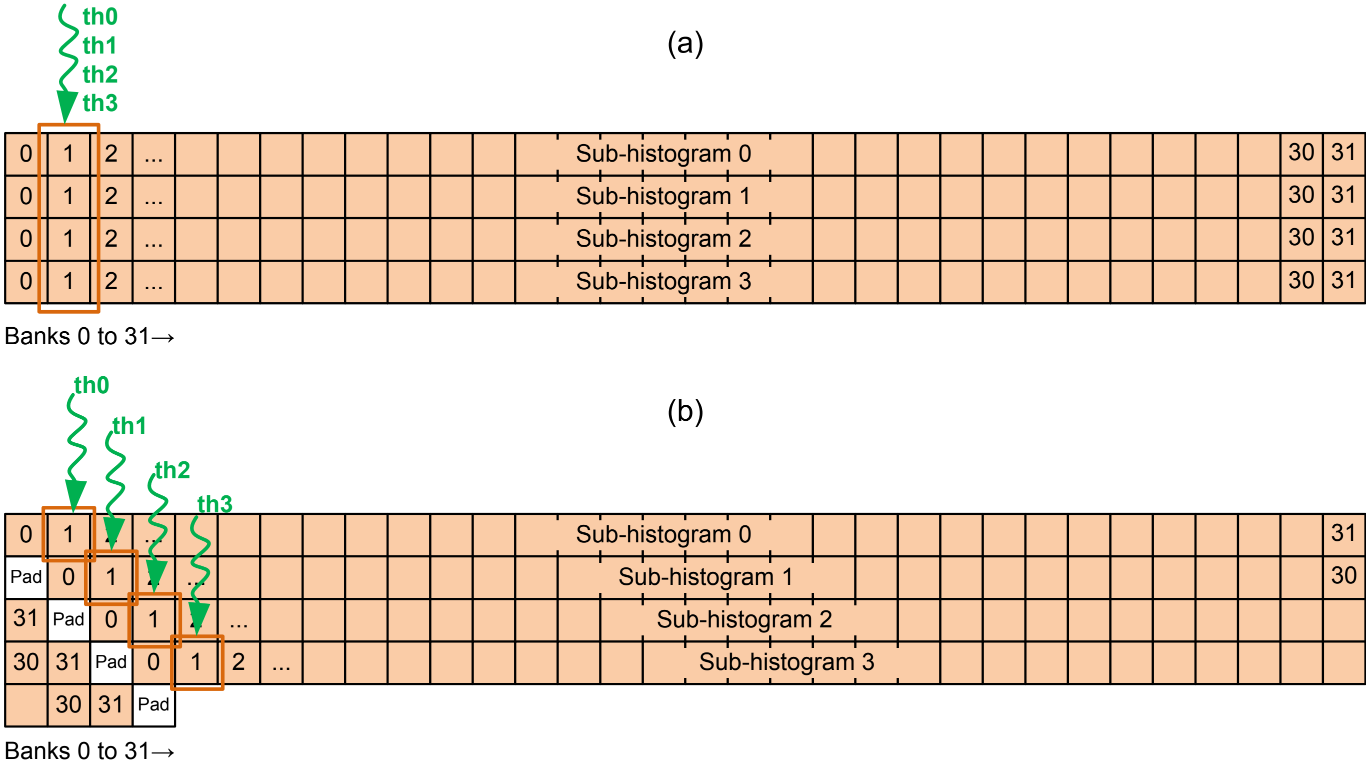


Bank and lock conflicts are greatly reduced with an XOR hash function

V.d.Braak et al., "Simulation and Architecture Improvements of Atomic Operations on GPU Scratchpad Memory," ICCD 2013
V.d.Braak et al., "Configurable XOR Hash Functions for Banked Scratchpad Memories in GPUs," IEEE TC, 2016

Optimizing Histogram Computation (I)

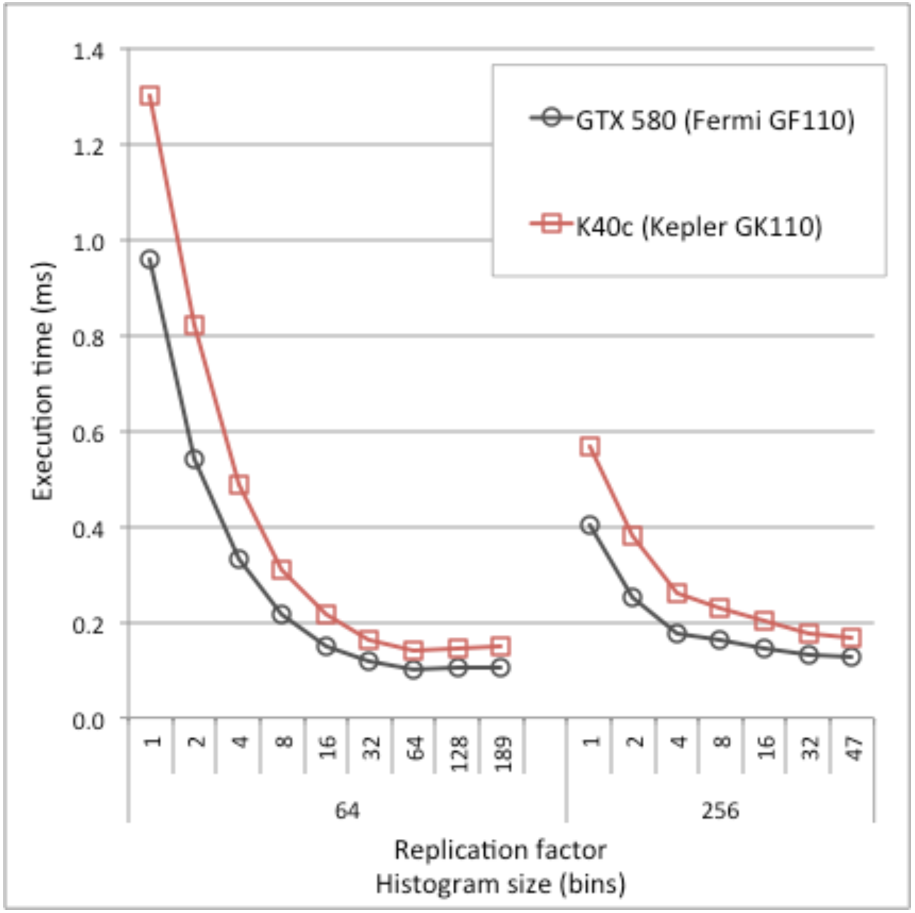
- Multiple private sub-histograms per block
- + padding to avoid conflicts on banks and lock bits


(a)
th0
th1
th2
th3
0 1 2 ... Sub-histogram 0 30 31
0 1 2 ... Sub-histogram 1 30 31
0 1 2 ... Sub-histogram 2 30 31
0 1 2 ... Sub-histogram 3 30 31
Banks 0 to 31→
(b)
th0
th1
th2
th3
0 1 2 ... Sub-histogram 0 31
Pad 0 1 2 ... Sub-histogram 1 30
31 Pad 0 1 2 ... Sub-histogram 2
30 31 Pad 0 1 2 ... Sub-histogram 3
30 31 Pad
Banks 0 to 31→

Gomez-Luna+, "An Optimized Approach to Histogram Computation on GPU," MVAP, 2013

Optimizing Histogram Computation (II)

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



Gomez-Luna+, "An Optimized Approach to Histogram Computation on GPU," MVAP, 2013

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:

Tesla, Fermi, Kepler

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

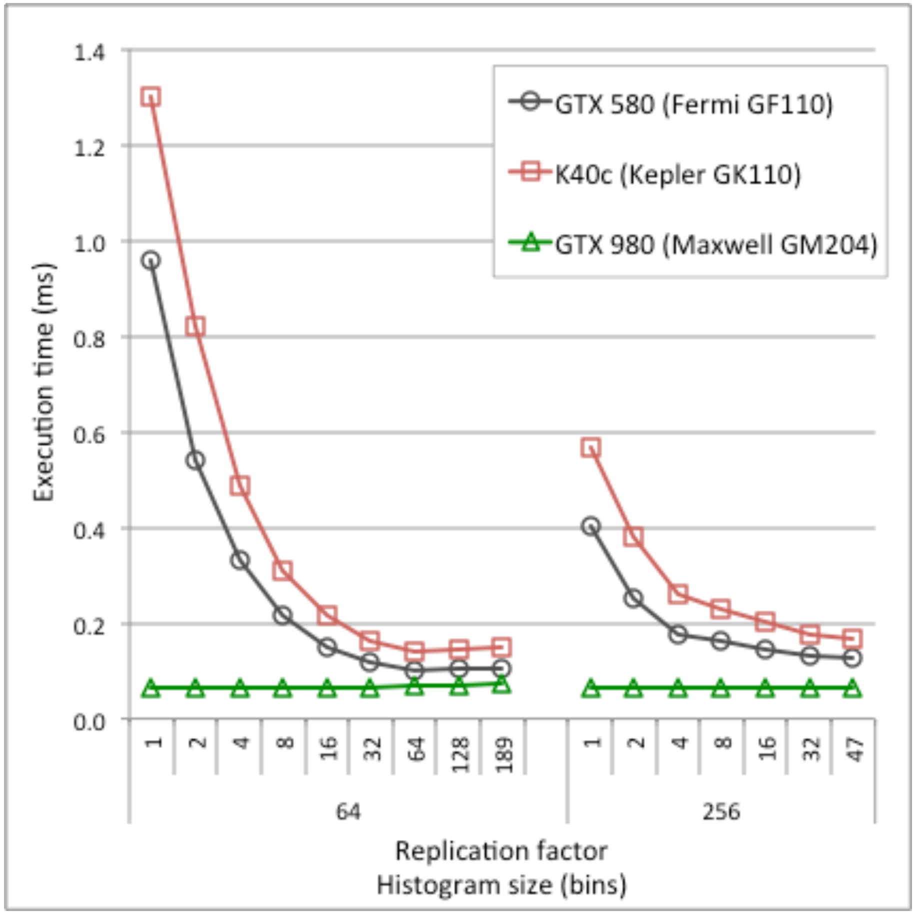
Maxwell, Pascal, Volta…

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

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

Optimizing Histogram Computation (III)

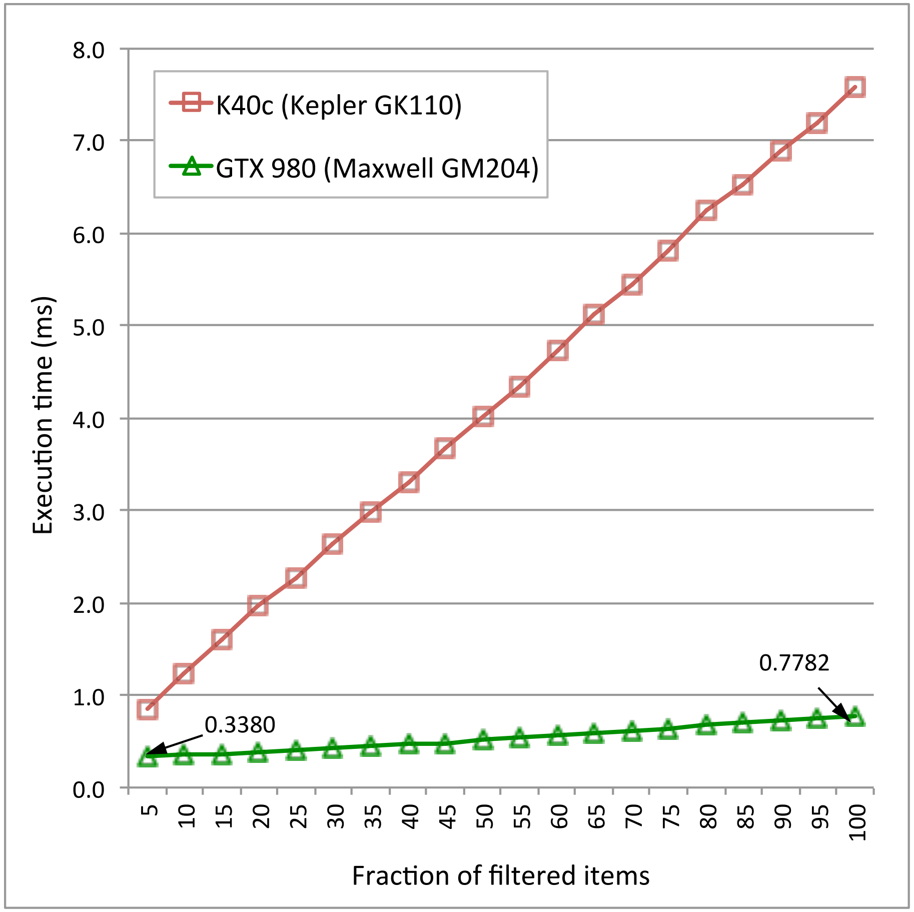
- Improved hardware (since Maxwell) saves programmers' effort



Gomez-Luna+, "An Optimized Approach to Histogram Computation on GPU," MVAP, 2013

Another Example: Stream Compaction

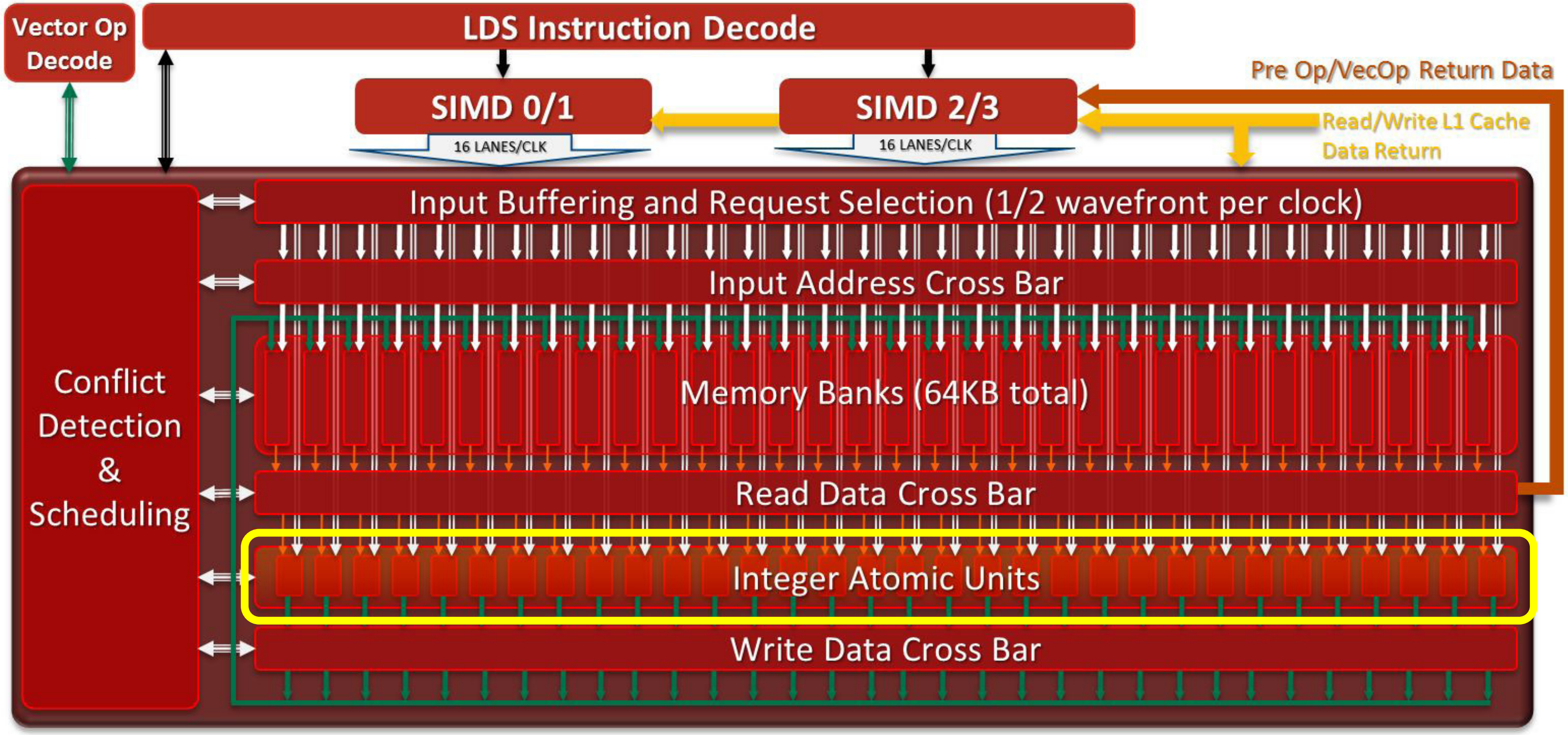
- 1 single counter per block updated via shared memory atomics



CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics
https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

Atomic Units Near Memory Banks

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



"AMD Graphics Cores Next (GCN) Architecture," White paper, June 2012

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

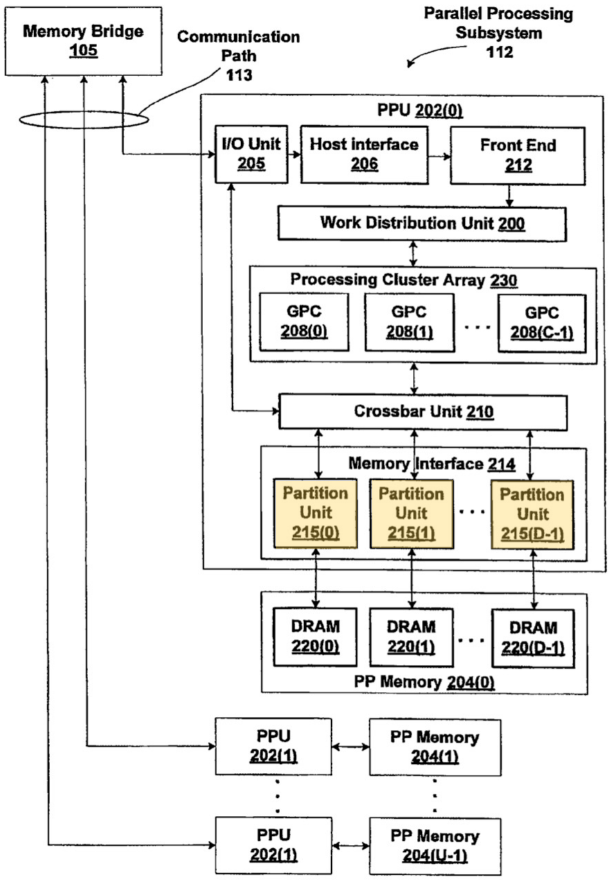
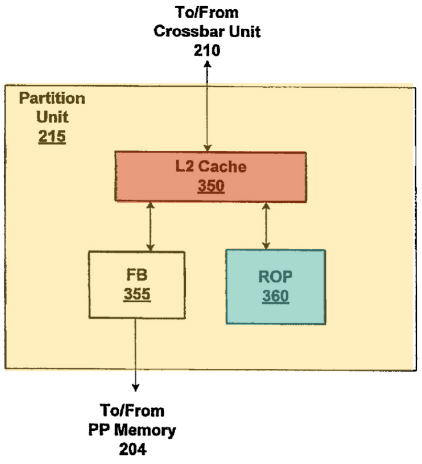
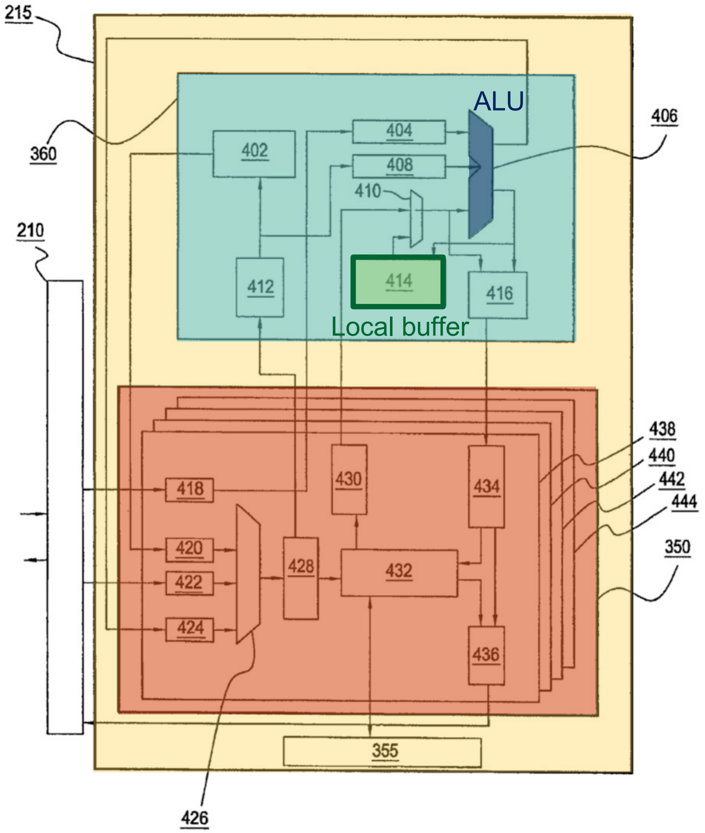


Figure 2

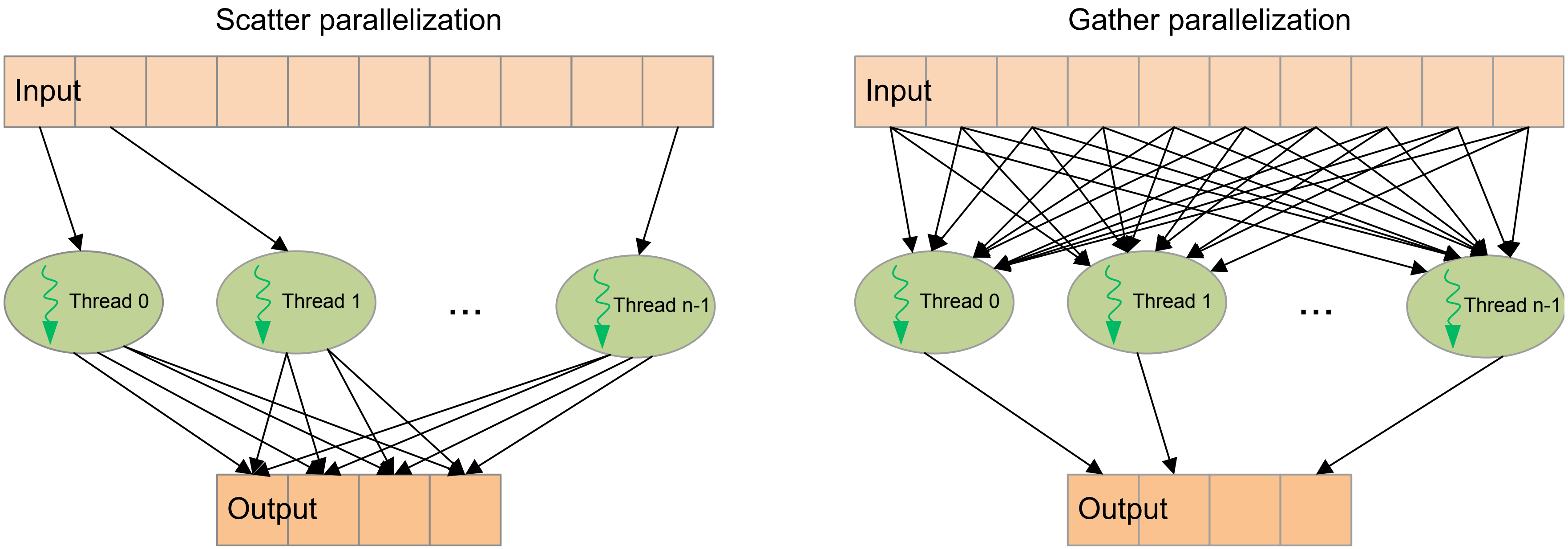




Glasco et al., "Cache-based Control of Atomic Operations in Conjunction with an External ALU Block," US8135926B1

Scatter vs. Gather

- Scatter assigns input elements
- Gather assigns output elements

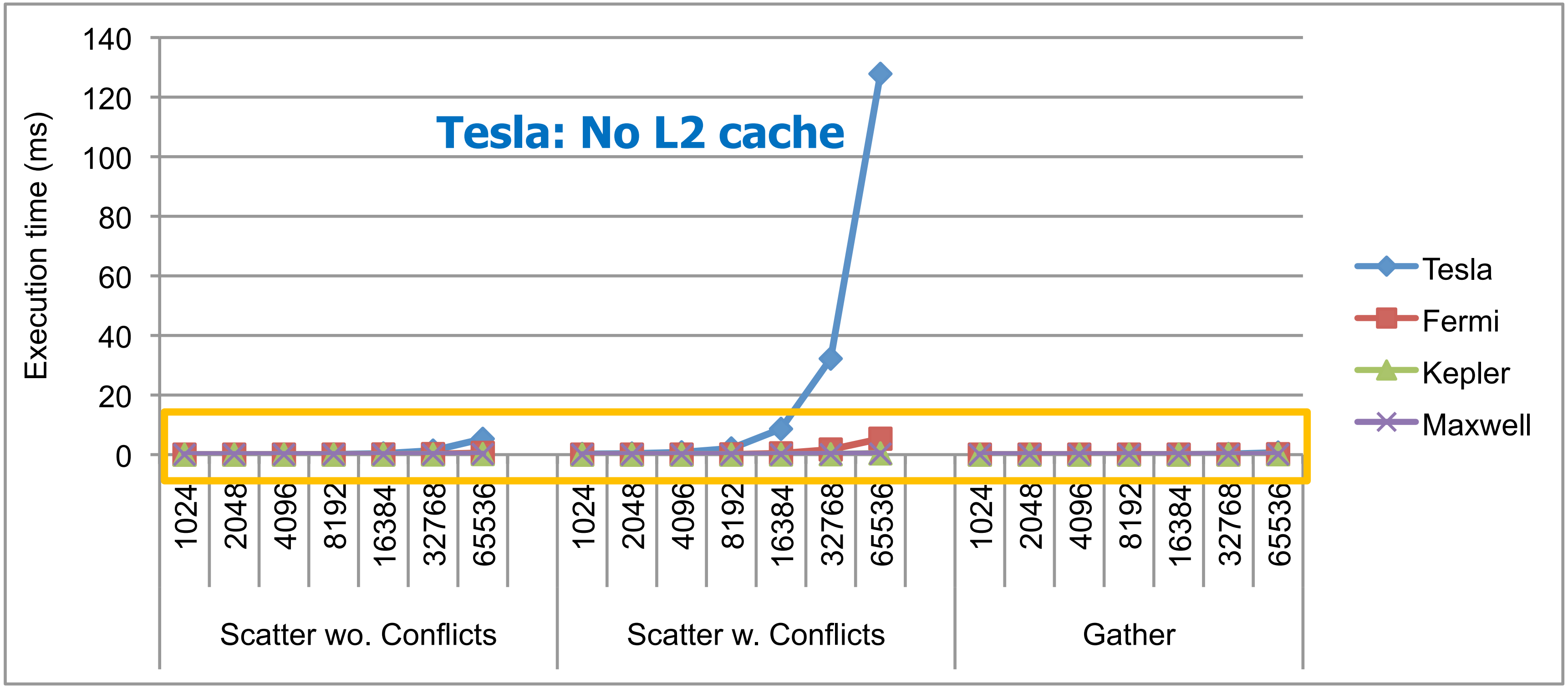


Scatter vs. Gather: Example Codes

```
__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;
    }
}
```

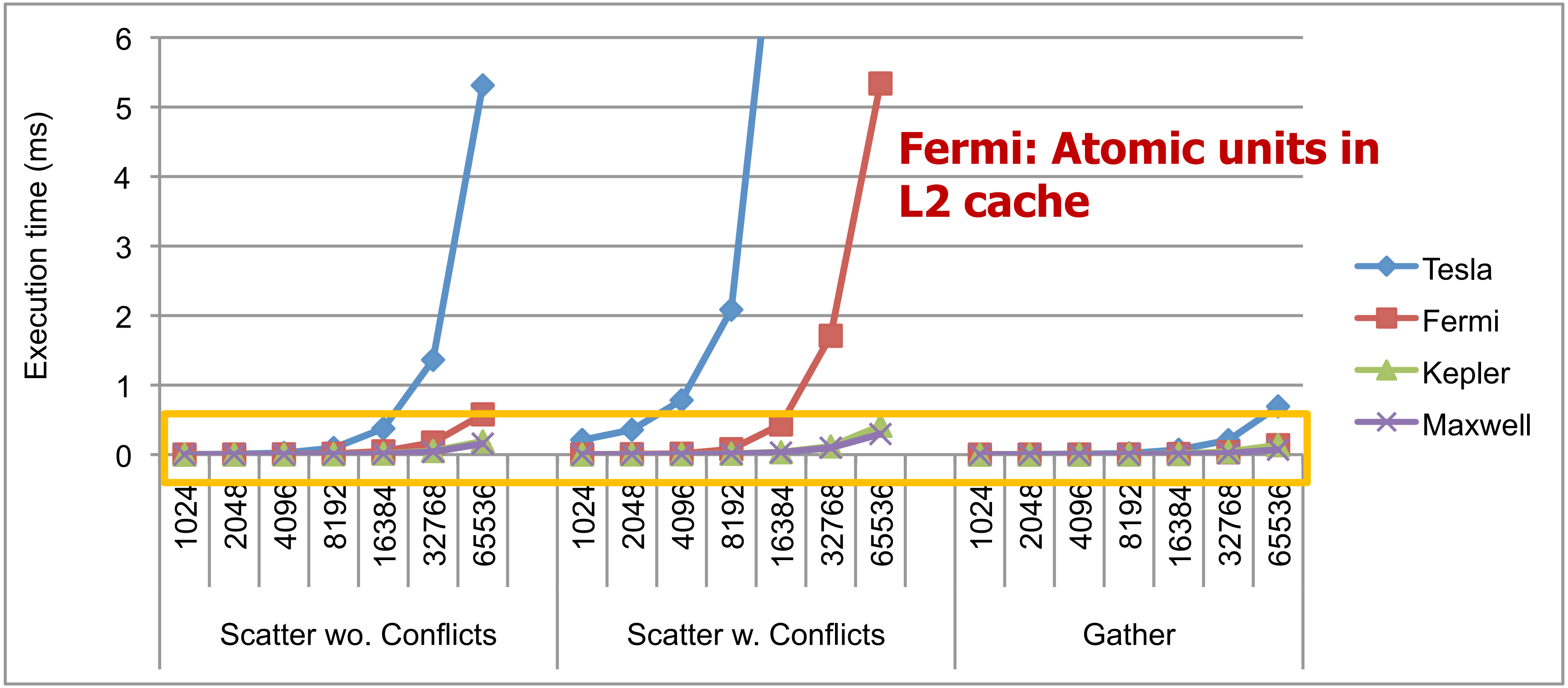
Scatter vs. Gather: Evaluation (I)

- Scatter: Large penalty due to atomic conflicts in DRAM



Scatter vs. Gather: Evaluation (II)

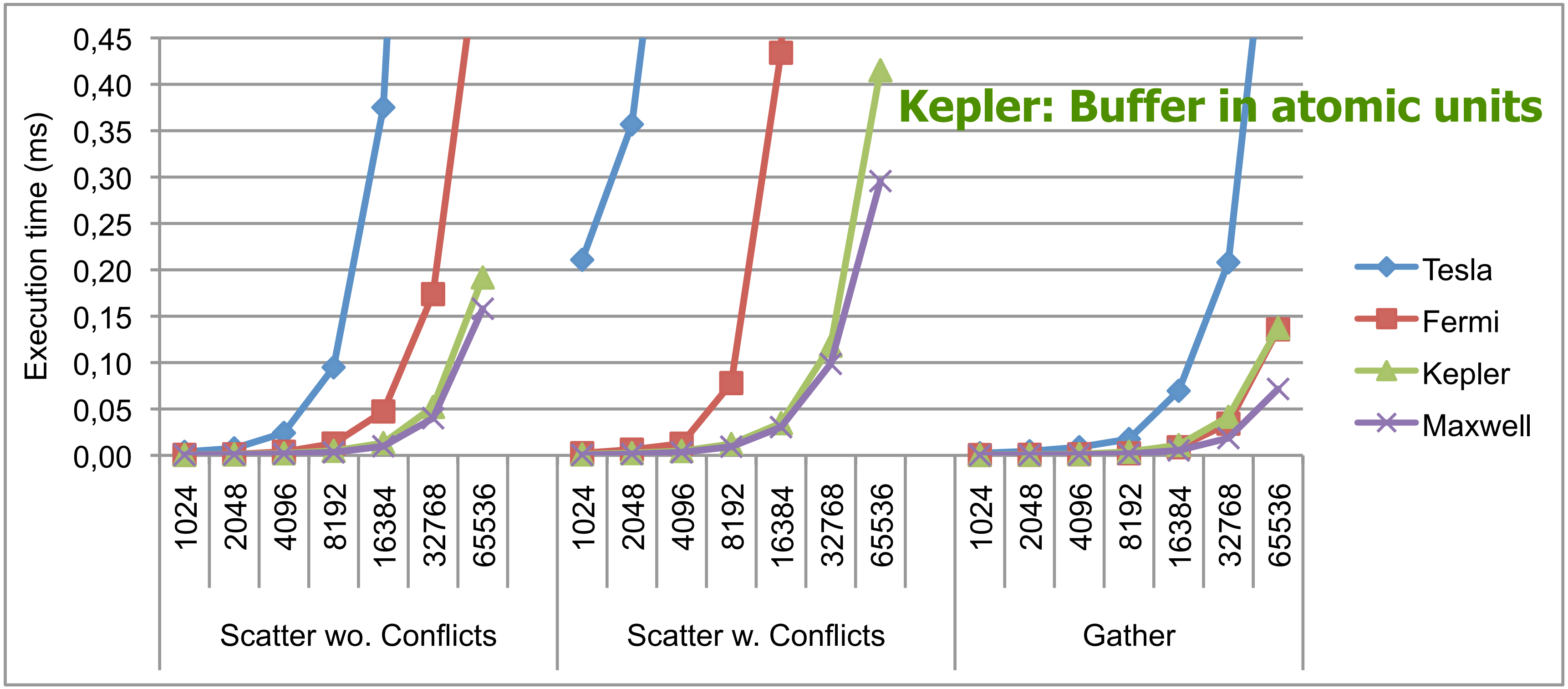
- L2 atomics improve the performance by an order of magnitude



Fermi: Atomic units in L2 cache

Scatter vs. Gather: Evaluation (III)

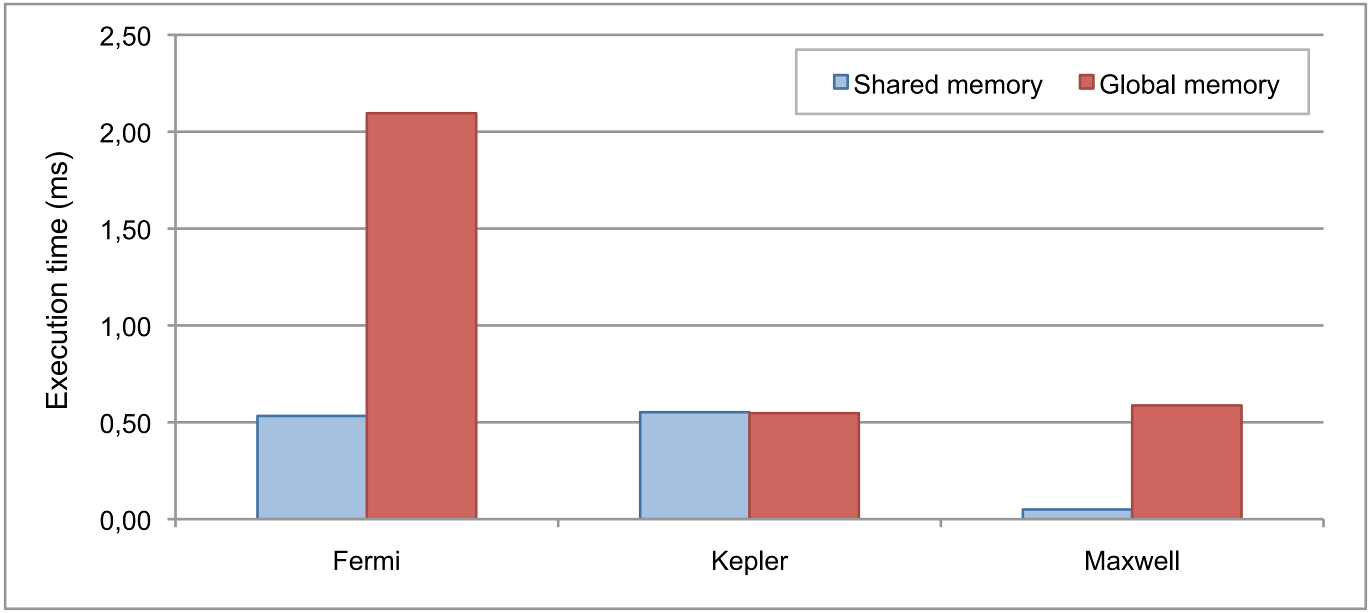
- Another 10x speedup with local buffers



Kepler: Buffer in atomic units

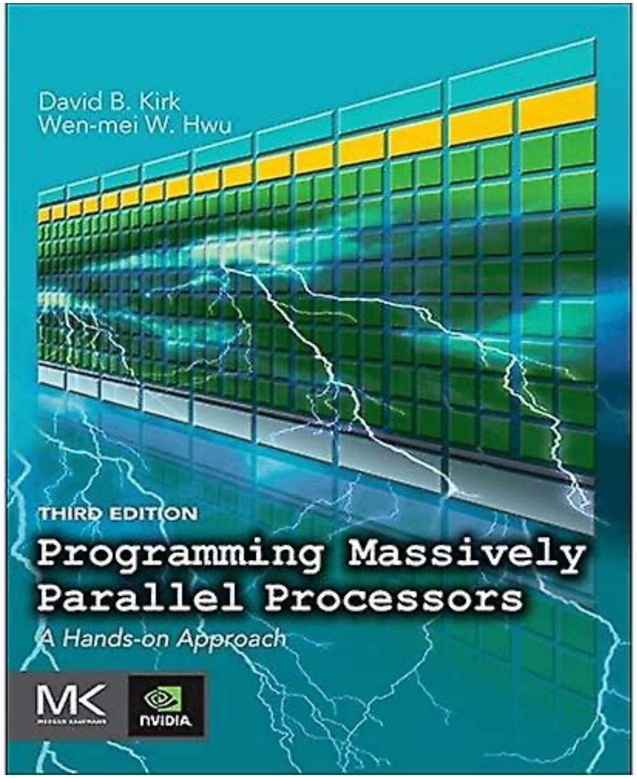
Effect of Hardware Improvements

- 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



Recommended Readings

- Hwu and Kirk, "Programming Massively Parallel Processors," Third Edition, 2017
  - Chapter 9 - Parallel patterns — parallel histogram computation: An introduction to atomic operations and privatization



P&S Heterogeneous Systems

Parallel Patterns: Histogram

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2021

18 November 2021