University of Illinois at Urbana-Champaign. ECE 408. October 22, 2015

Atomic Operations across GPU generations

Juan Gómez-Luna
University of Córdoba (Spain)

About me

- Juan Gómez-Luna
- Telecommunications Engineering (University of Sevilla, 2001)
- Since 2005 Lecturer at the University of Córdoba
- PhD Thesis (University of Córdoba, 2012)
  - Programming Issues for Video Analysis on Graphics Processing Units
- Research collaborations:
  - Technical University Munich (Germany)
  - Technical University Eindhoven (The Netherlands)
  - University of Illinois at Urbana-Champaign (USA)
  - University of Málaga (Spain)
  - Barcelona Supercomputing Center (Spain)
- PI of University of Córdoba GPU Education Center (supported by NVIDIA)

Outline

- Uses of atomic operations
- Atomic operations on shared memory
  - Evolution across GPU generations
  - Case studies
    - Stream compaction
    - Histogramming
    - Reduction
- Atomic operations on global memory
  - Evolution across GPU generations
  - Case studies
    - Scatter vs. gather
    - Adjacent thread block synchronization

Uses of atomic operations

- Collaboration
  - Atomics on an array that will be the output of the kernel
  - Example
    - Histogramming

- Synchronization
  - Atomics on memory locations that are used for synchronization or coordination
  - Example
    - Locks, flags…

Uses of atomic operations

- CUDA provides atomic functions on shared memory and global memory
- Arithmetic functions
  - Add, sub, max, min, exch, inc, dec, CAS

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

- Bitwise functions
  - And, or, xor
- Integer, uint, ull, and float

Outline

- Uses of atomic operations

- Atomic operations on shared memory
  - Evolution across GPU generations
  - Case studies
    - Stream compaction
    - Histogramming
    - Reduction

- Atomic operations on global memory
  - Evolution across GPU generations
  - Case studies
    - Scatter vs. gather
    - Adjacent thread block synchronization

Atomic operations on shared memory

- Code
  - CUDA: `int atomicAdd(int*, int);`
  - PTX: `atom.shared.add.u32 %r25, [%rd14], 1;`
  - 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

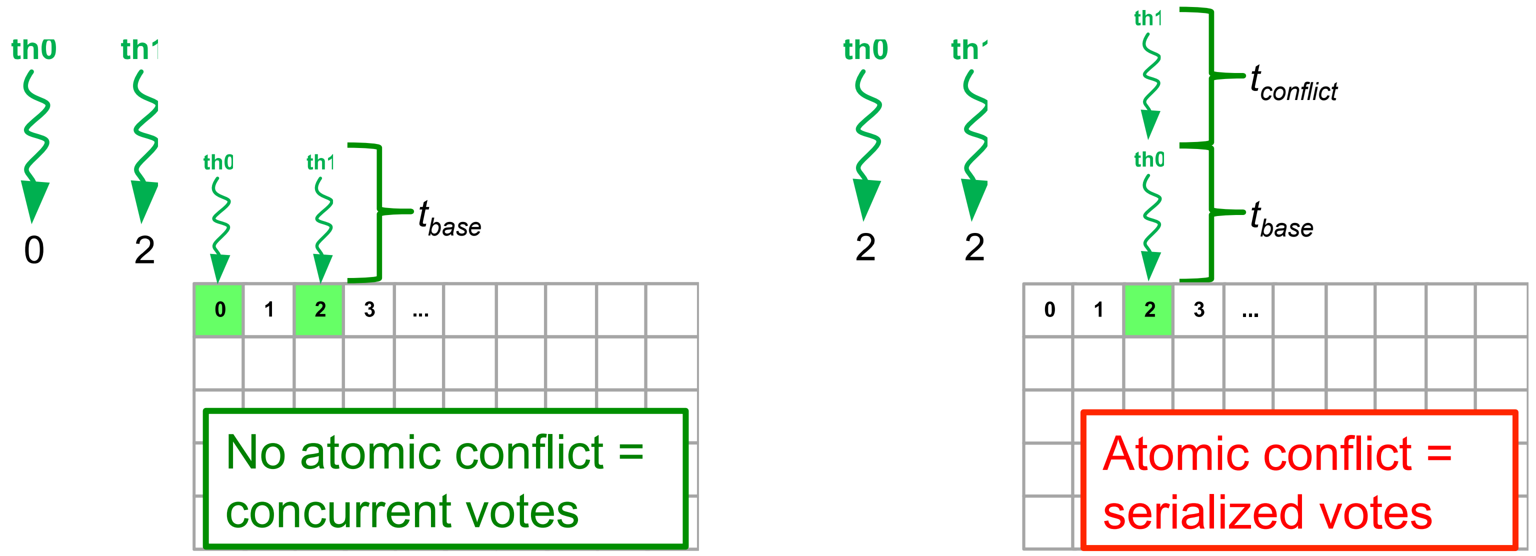
```
/*01f8*/  ATOMS.ADD RZ, [R7], R11;
```

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

  - Lock/Update/Unlock vs. Native atomic operations

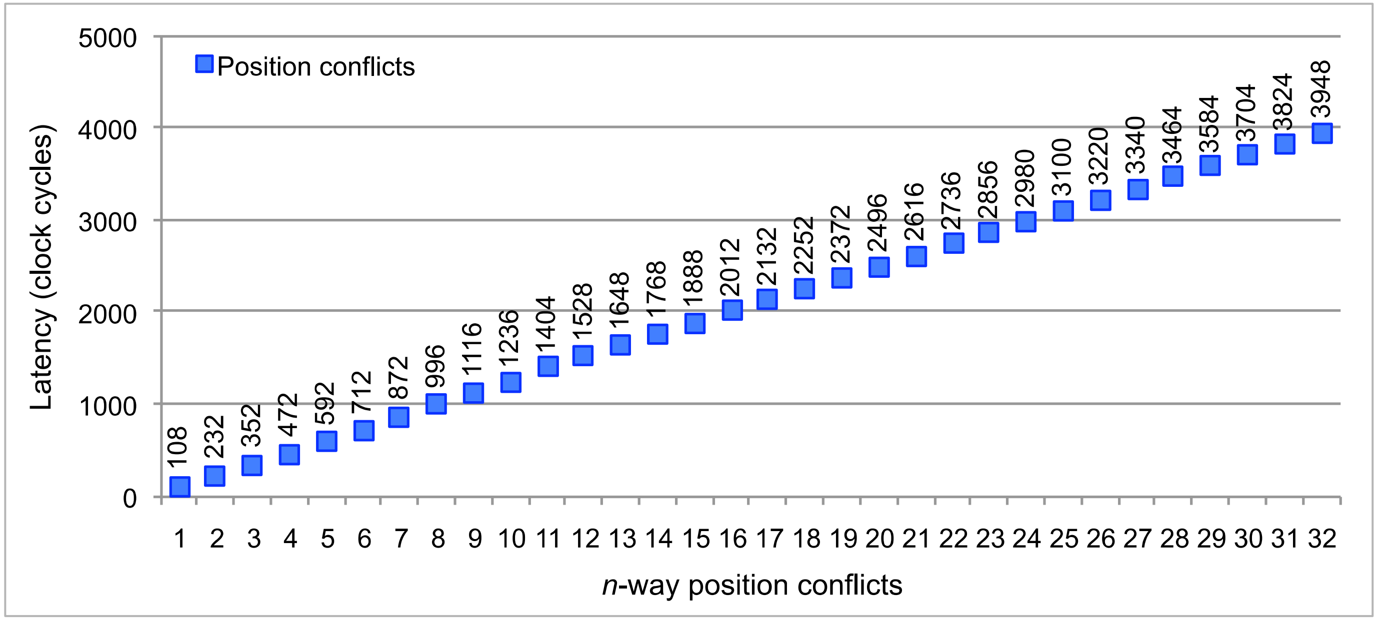
Atomic operations on shared memory

- Atomic conflict degree
  - Intra-warp conflict degree from 1 to 32



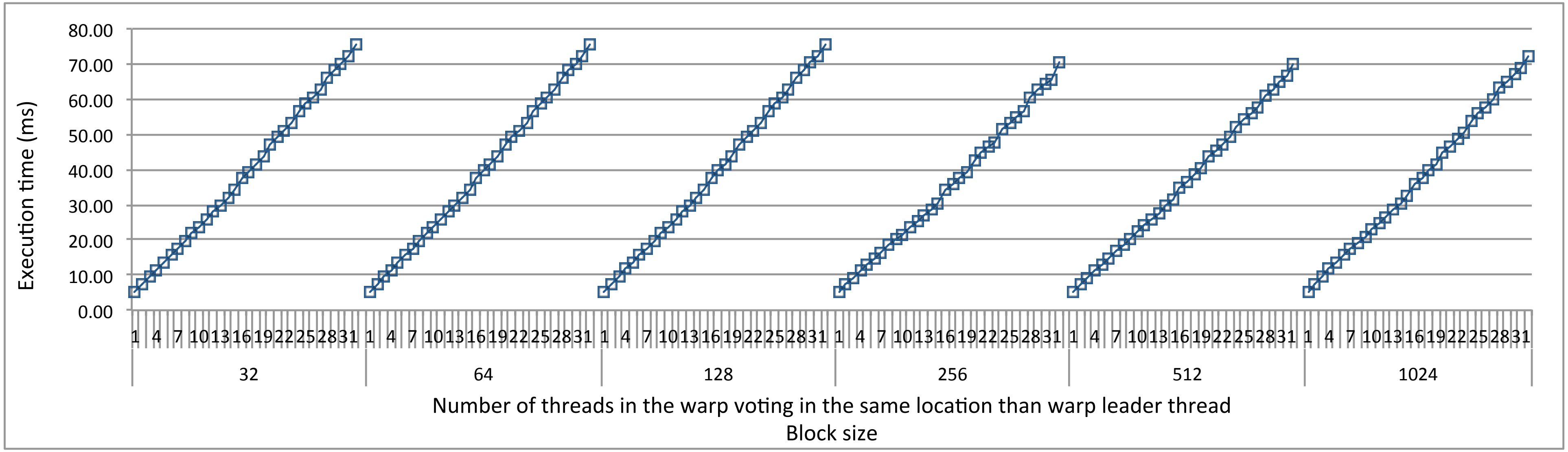
Atomic operations on shared memory

- Microbenchmarking on Tesla, Fermi and Kepler
  - Position conflicts (GTX 580 – Fermi)



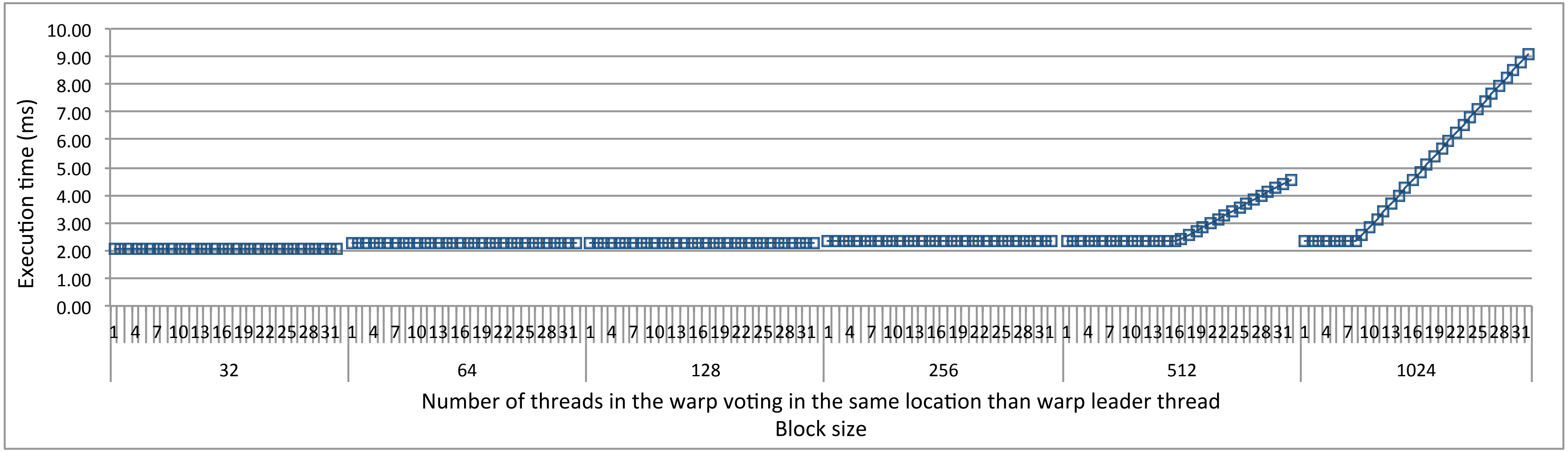
Atomic operations on shared memory

- Microbenchmarking on Tesla, Fermi and Kepler
  - Position conflicts (K20 – Kepler)



Atomic operations on shared memory

- Microbenchmarking on Maxwell
  - Position conflicts (GTX 980 – Maxwell)



Case studies: Filtering

- Filtering / Stream compaction

Stream compaction

Input	2	1	3	0	0	1	3	4	0	0	2	1

Predicate: Element > 0

Output	2	1	3	1	3	4	2	1

Case studies: Filtering

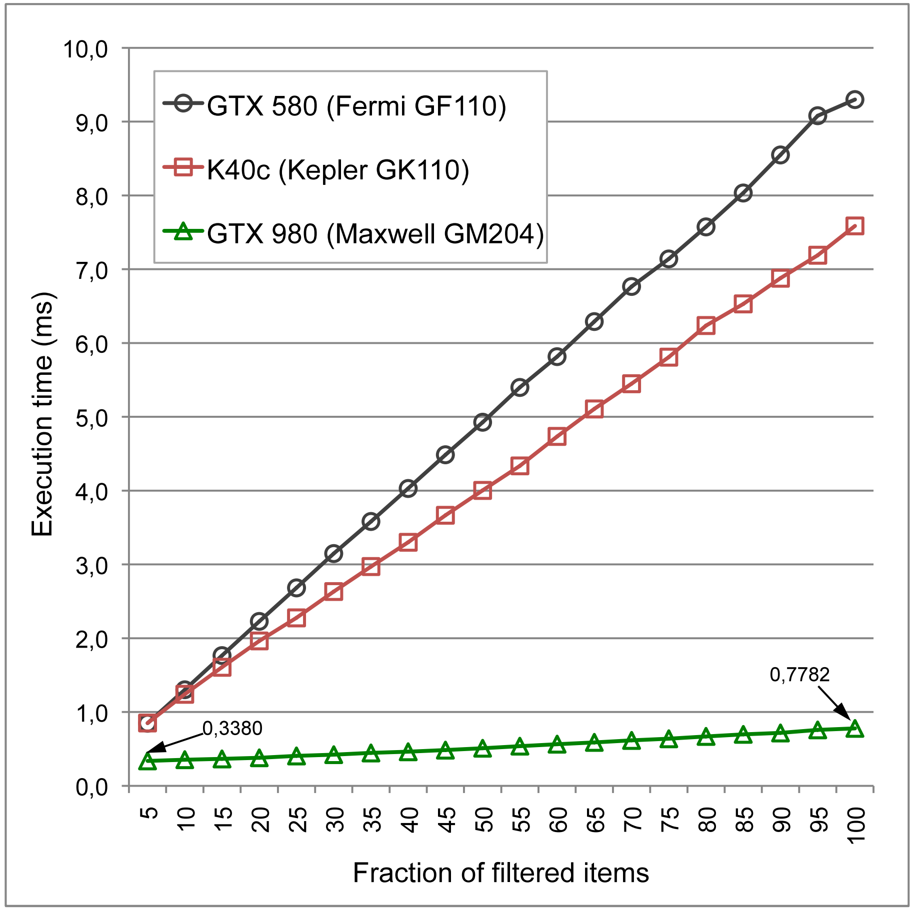
- Filtering /
  - Global m
  - Shared r

```
__global__ void
  int i = threac
  if(i < n && sı
    int index =
    dst[index] =
  }
}
```

```
__global__ void filter_shared_k(int *dst, int *nres, const int* src, int n, int value) {
  __shared__ int l_n;
  int i = blockIdx.x * (NPER_THREAD * BS) + threadIdx.x;

  for (int iter = 0; iter < NPER_THREAD; iter++) {
    // zero the counter
    if (threadIdx.x == 0)
      l_n = 0;
    __syncthreads();

    // get the value, evaluate the predicate, and
    // increment the counter if needed
    int d, pos;
    if(i < n) {
      d = src[i];
      if(d != value)
         pos = atomicAdd(&l_n, 1);
    }
    __syncthreads();

    // leader increments the global counter
    if(threadIdx.x == 0)
       l_n = atomicAdd(nres, l_n);
    __syncthreads();
    // threads with true predicates write their elements
    if(i < n && d != value) {
      pos += l_n; // increment local pos by global counter
      dst[pos] = d;
    }
    __syncthreads();
    i += BS;
  }
}
```

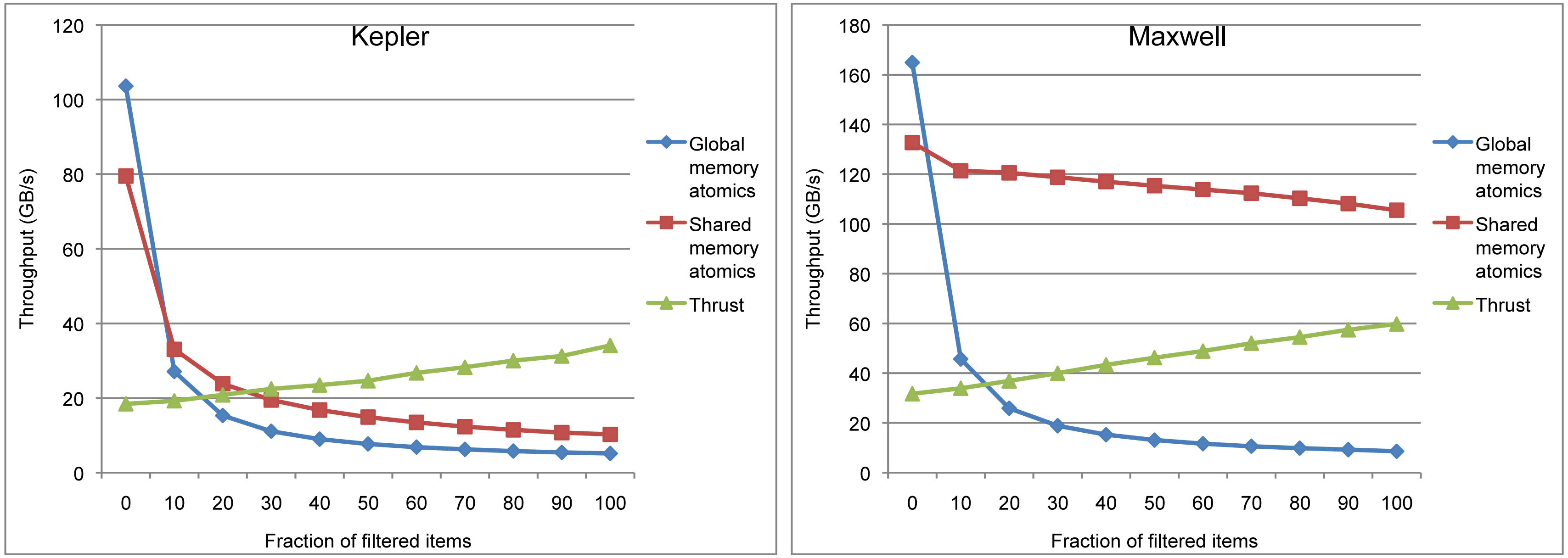
Case studies: Filtering

- Filtering / Stream compaction: Shared memory atomics



Case studies: Filtering

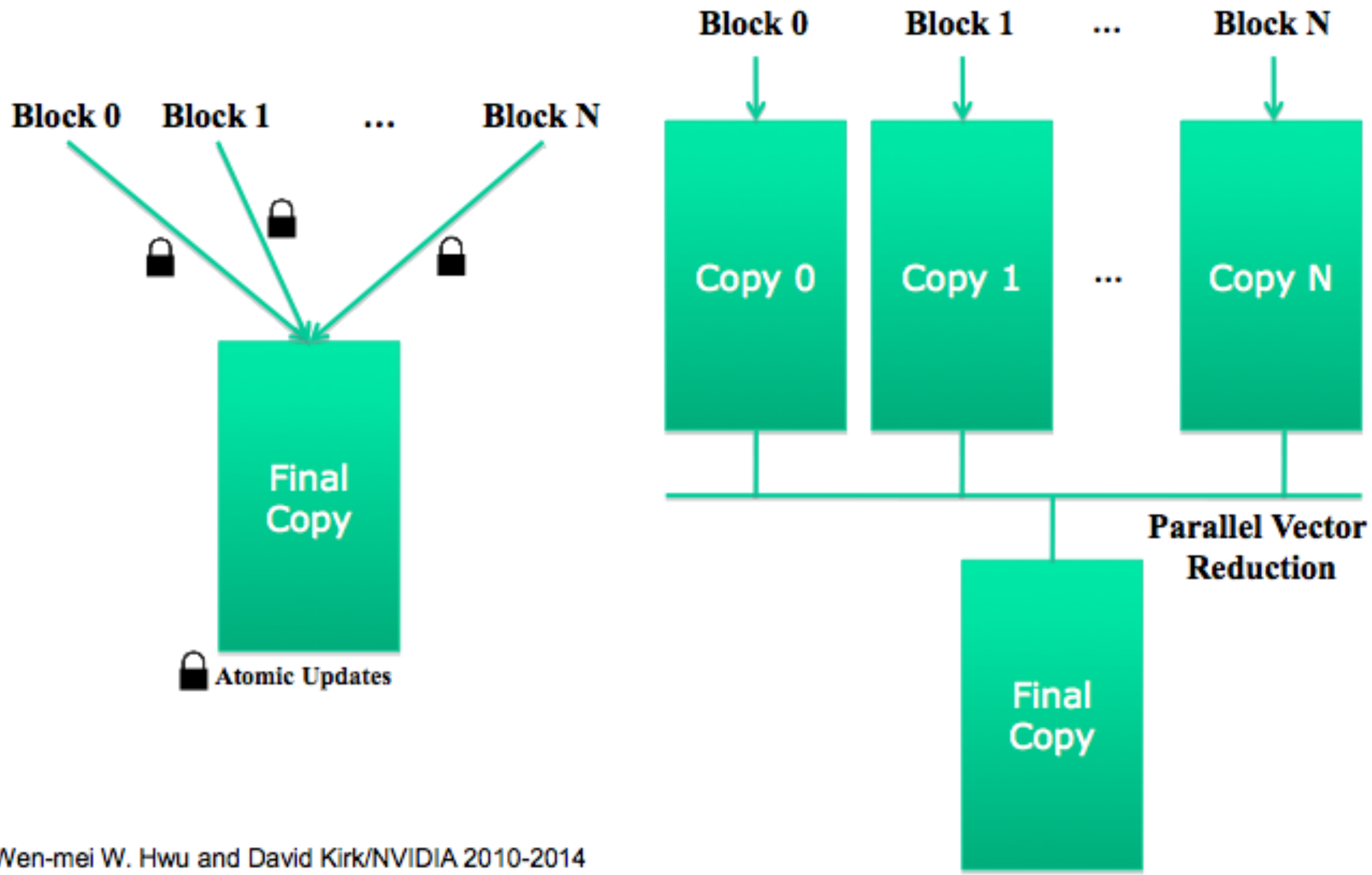
- Filtering / Stream compaction



Find more: CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics

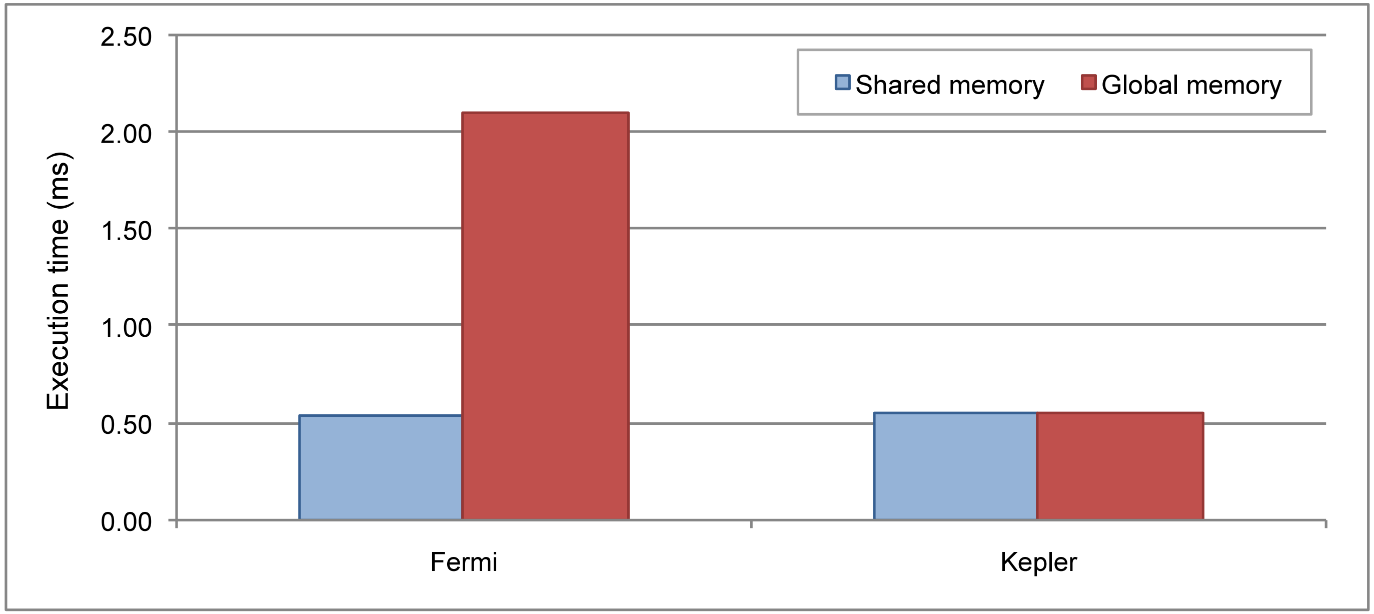
Case studies: Histogramming

- Privatization for histogram generation



Case studies: Histogramming

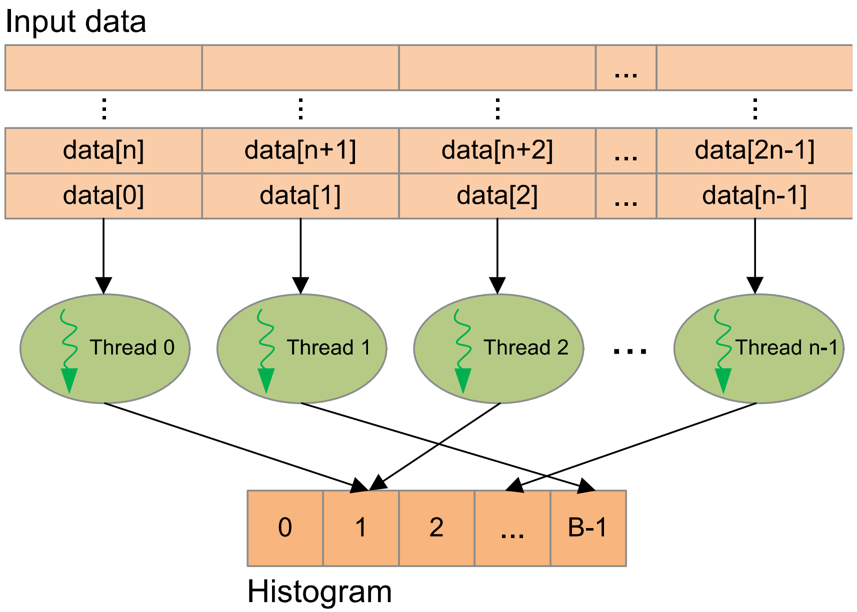
- Privatization
  - 256-bin histogram calculation for 100 real images
    - Shared memory implementation uses 1 sub-histogram per block
    - Global atomics were greatly improved in Kepler



Case studies: Histogramming

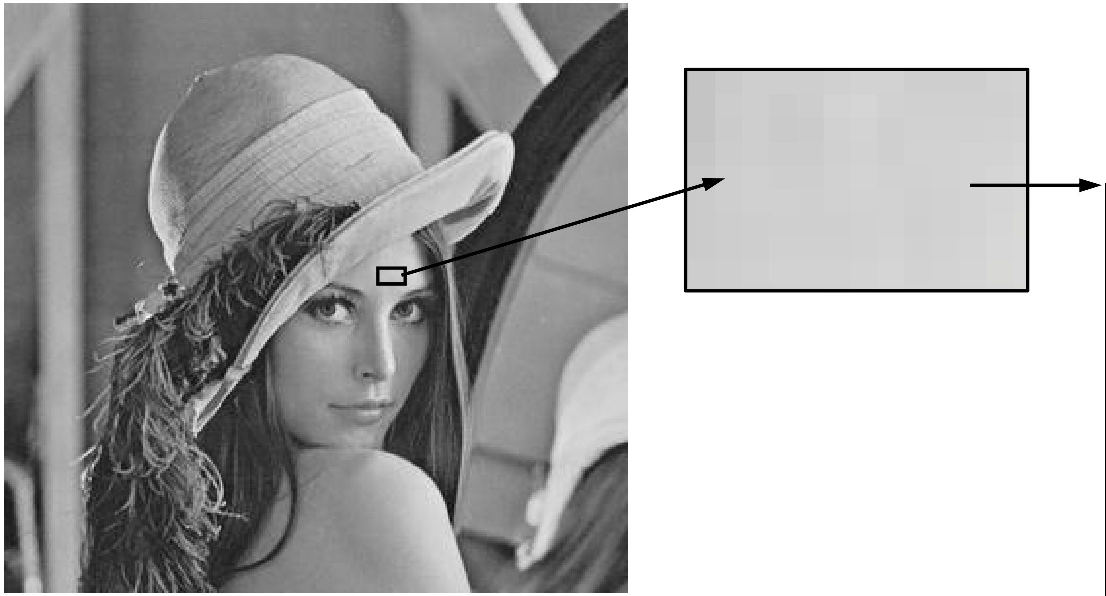
- Histogram calculation

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



Case studies: Histogramming

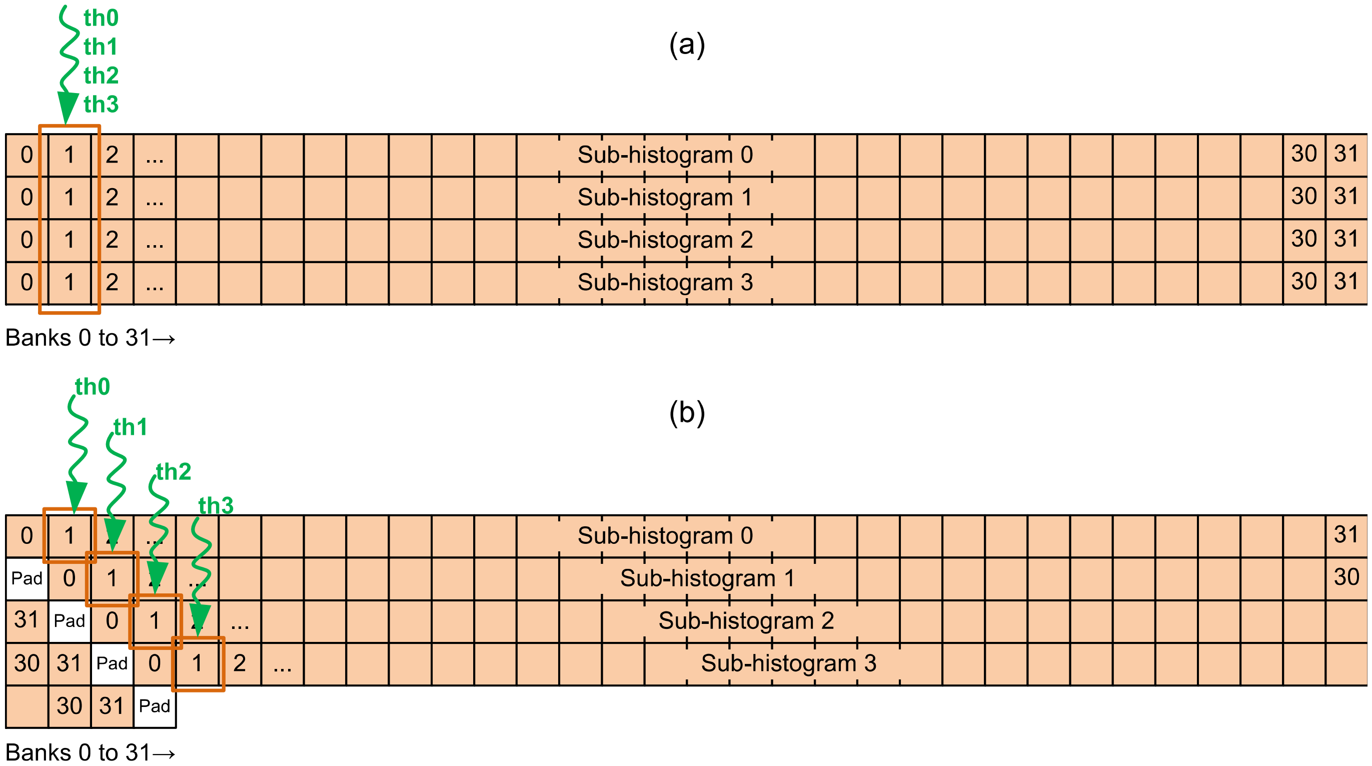
- Histogram calculation
- Natural images: spatial correlation



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

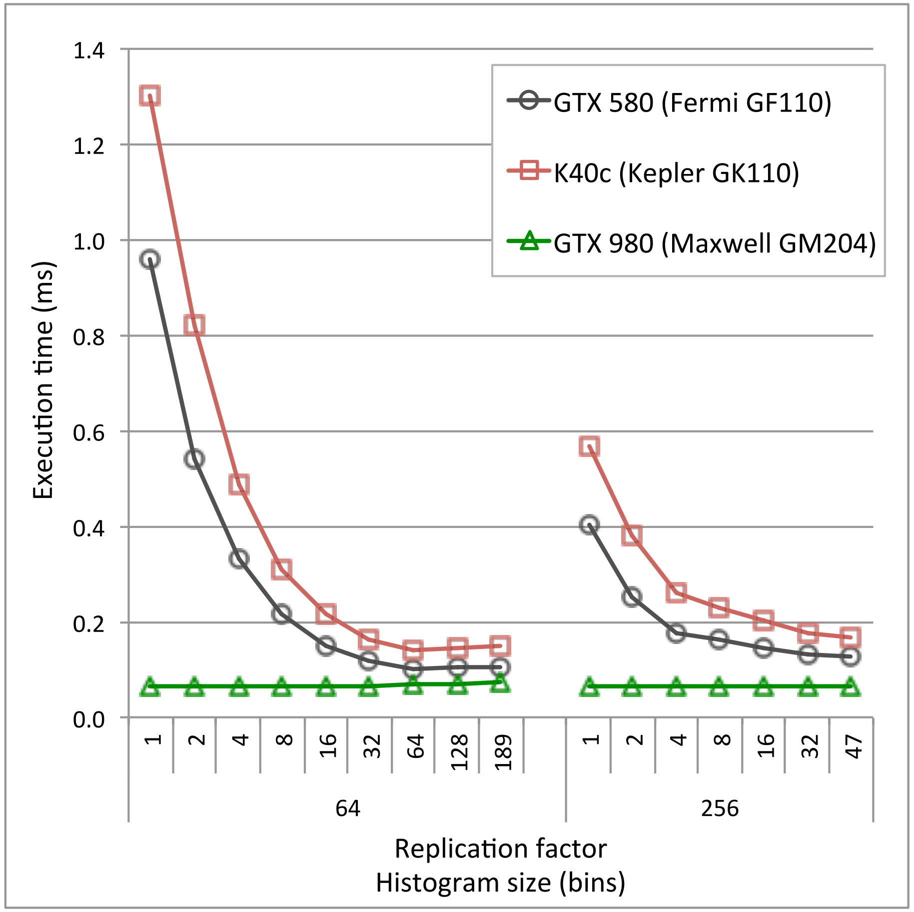
Case studies: Histogramming

- Histogram calculation
- Privatization + Replication + Padding



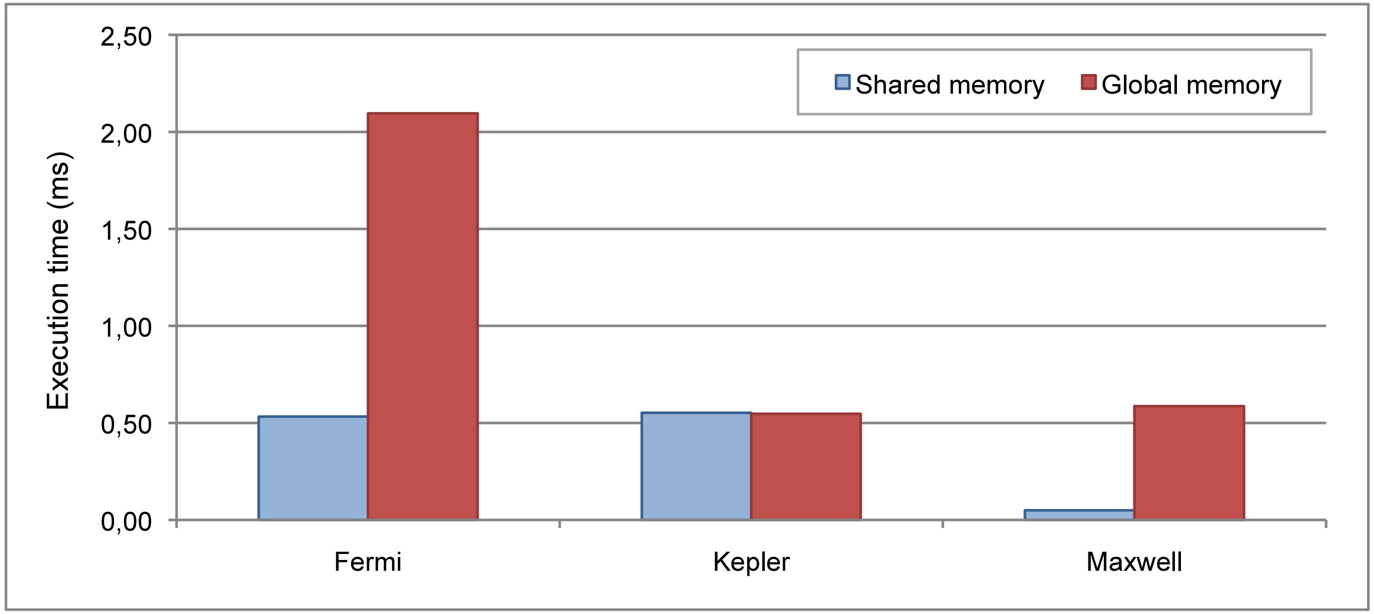
Case studies: Histogramming

- Histogram calculation: 100 real images
  - Privatization + Replication + Padding



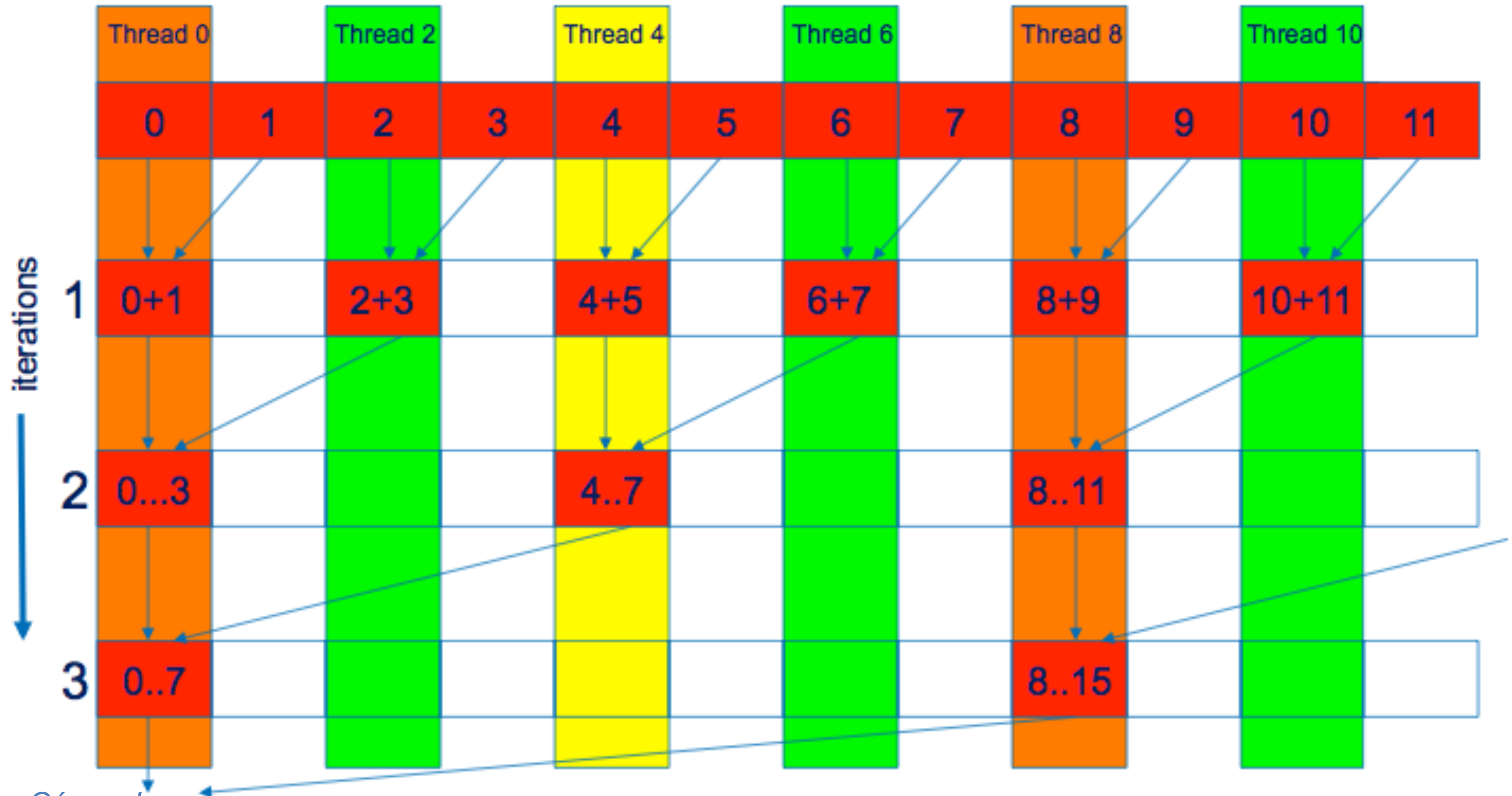
Case studies: Histogramming

- Privatization
  - 256-bin histogram calculation for 100 real images
    - Shared memory implementation uses 1 sub-histogram per block
    - Global atomics were greatly improved in Kepler



Case studies: Reduction

- Reduction
  - Tree-based algorithm is recommended (avoid scatter style)

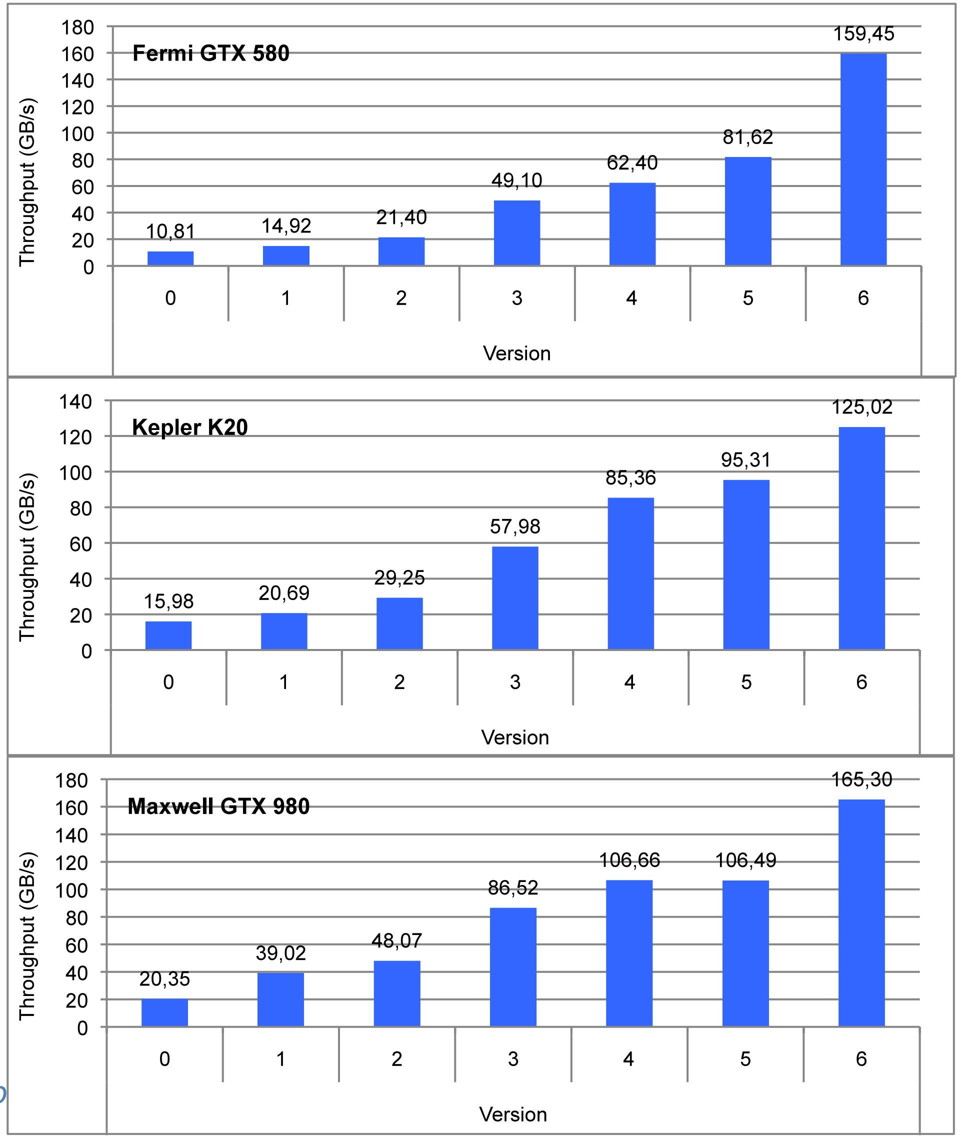


Case studies: Reduction

- 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

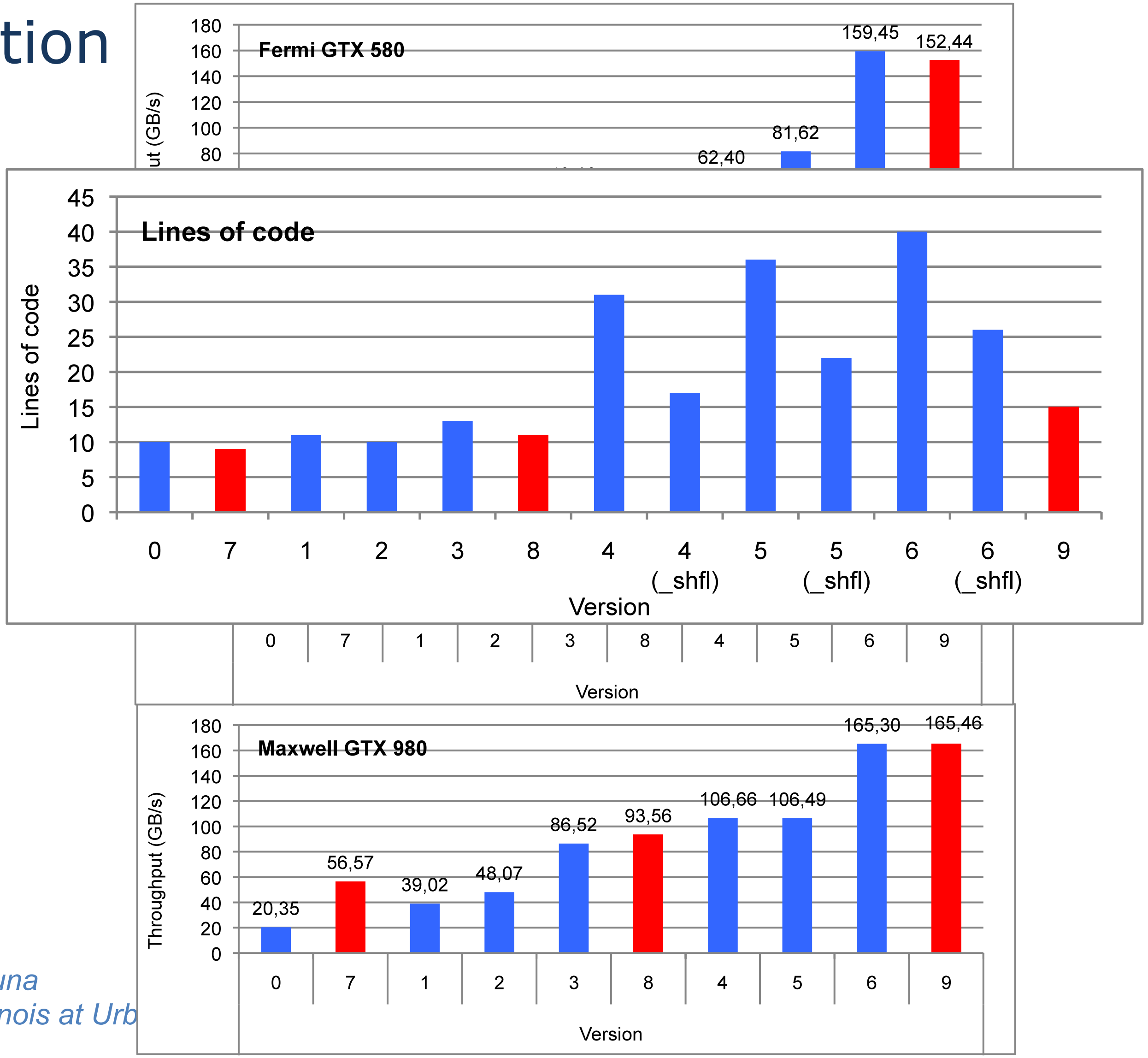
Case studies: Reduction

- Reduction



Case studies: Reduction

- Reduction



Outline

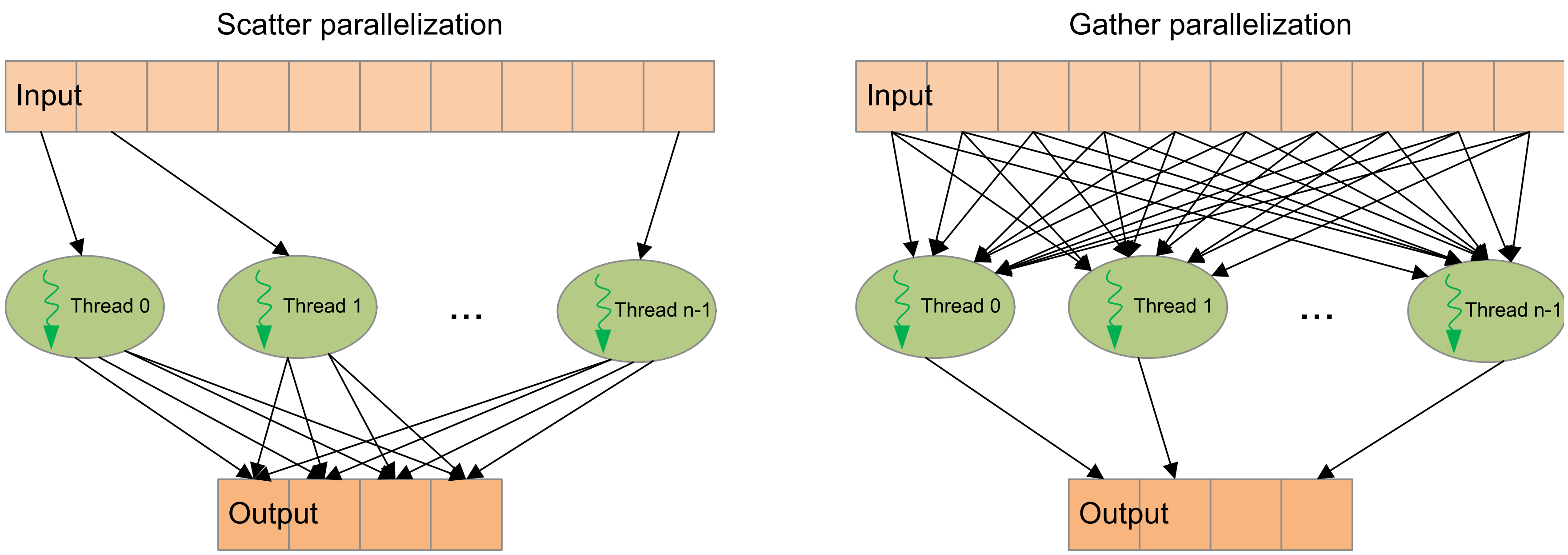
- Uses of atomic operations
- Atomic operations on shared memory
  - Evolution across GPU generations
  - Case studies
    - Stream compaction
    - Histogramming
    - Reduction
- Atomic operations on global memory
  - Evolution across GPU generations
  - Case studies
    - Scatter vs. gather
    - Adjacent thread block synchronization

Atomic operations on global memory

- Tesla:
  - Executed on DRAM
- Fermi:
  - Executed on L2
  - Atomic units near L2
- Kepler and Maxwell:
  - Atomic units near L2 now have kind of local cache

Case study: Scatter vs. gather

- Scatter vs. Gather



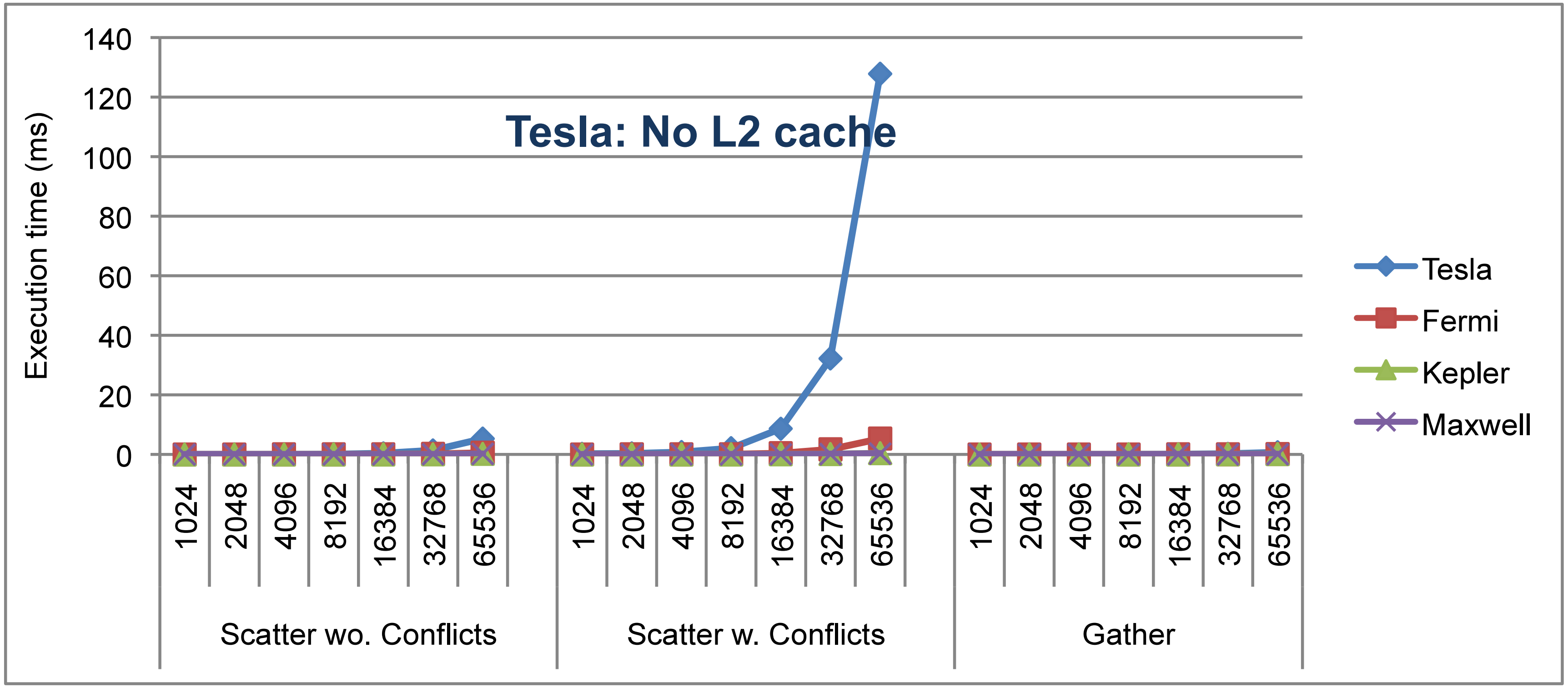
Case study: Scatter vs. gather

- Scatter vs. Gather

```
__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;
    }
}
```

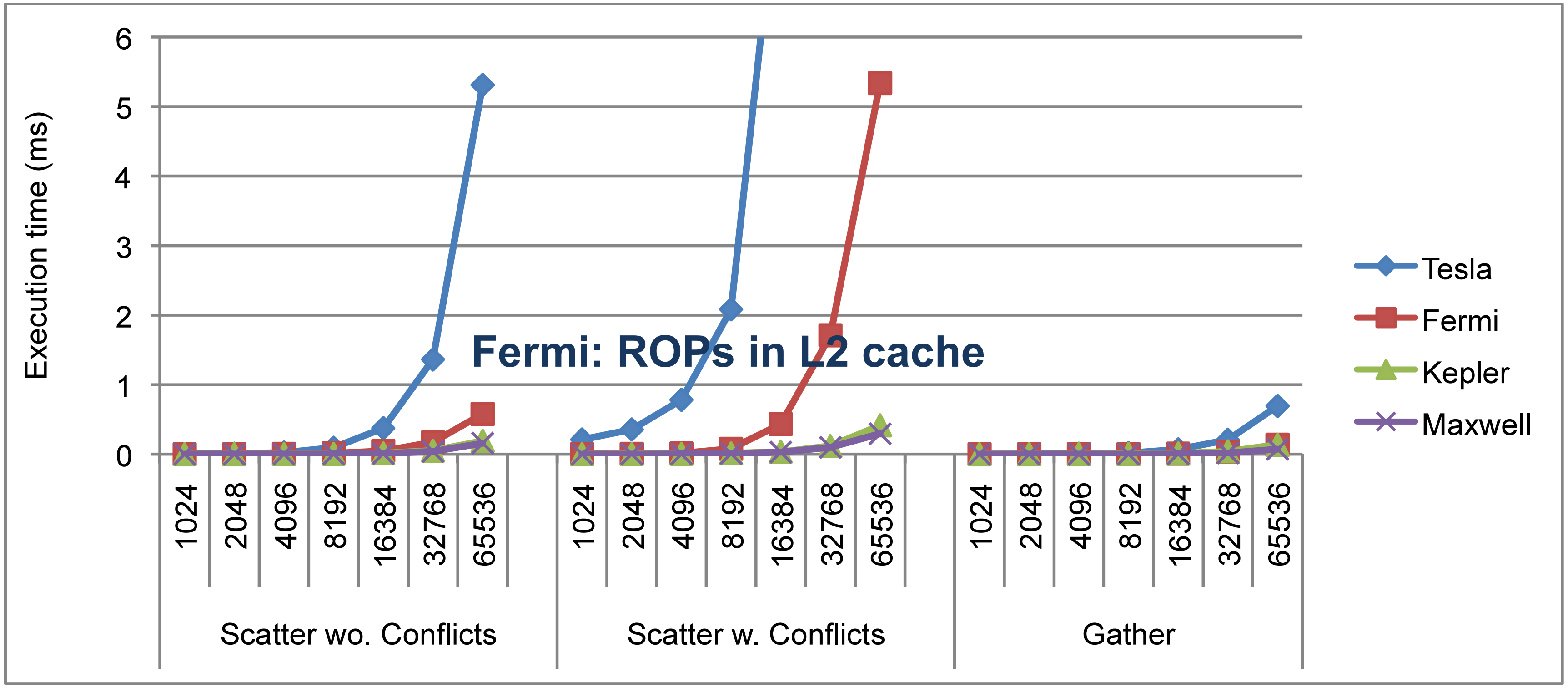
Case study: Scatter vs. gather

- Scatter vs. Gather



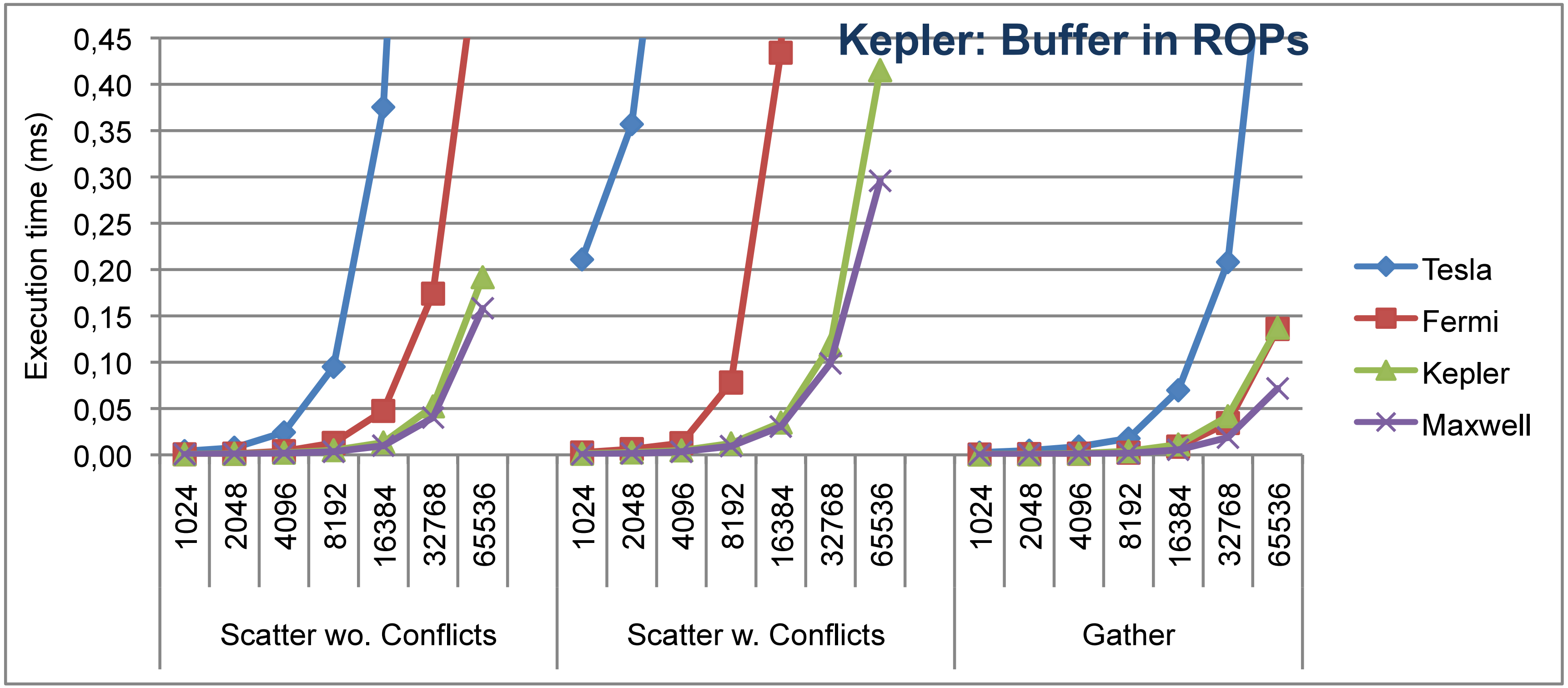
Case study: Scatter vs. gather

- Scatter vs. Gather



Case study: Scatter vs. gather

- Scatter vs. Gather

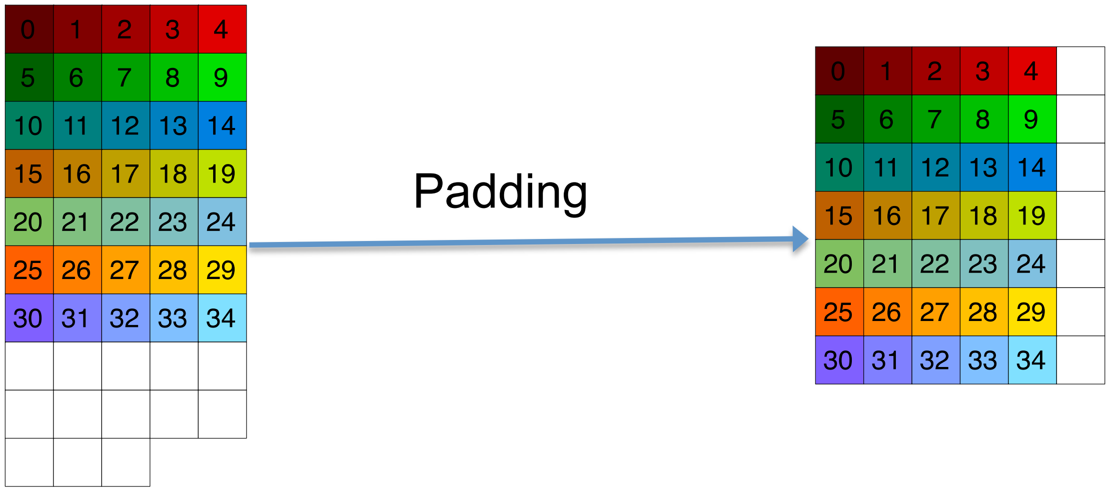


Case study: Adjacent block synchronization

- GPU programming with CUDA (or OpenCL) might not completely exploit inherent parallelism in some algorithms
  - In-place operations
    - Possible dependence between consecutive thread blocks
  - Bulk synchronous parallel programming
    - Thread block synchronization requires kernel termination and relaunch

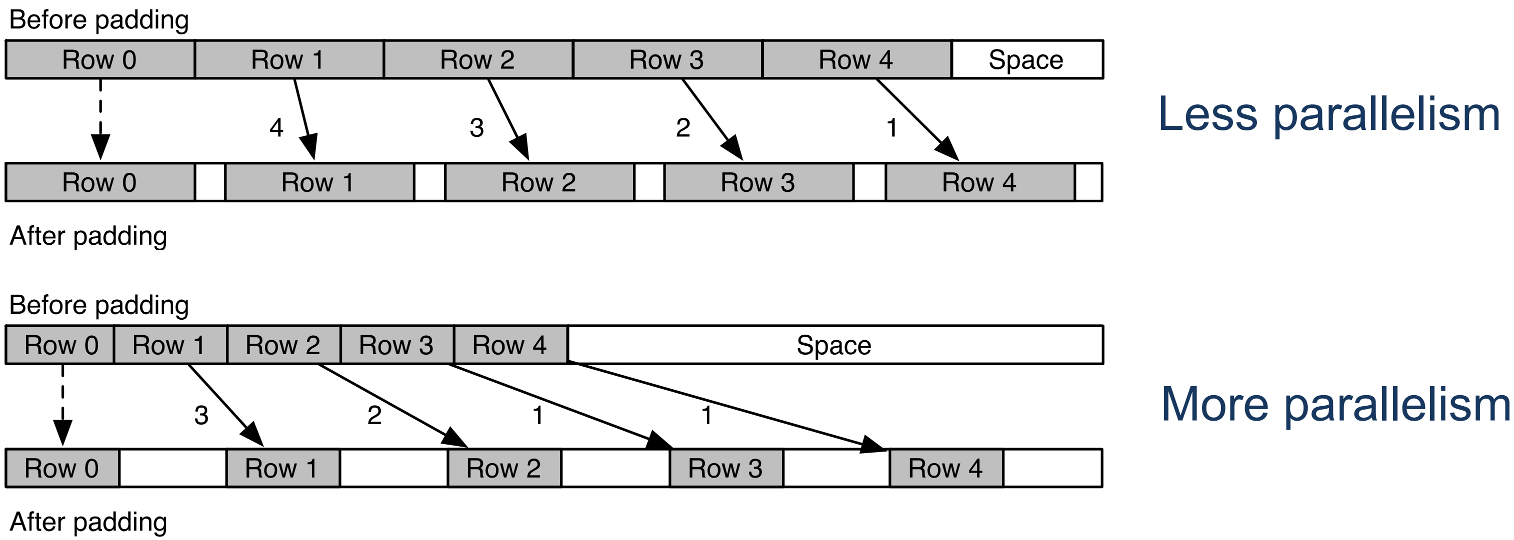
Case study: Adjacent block synchronization

- In-place matrix padding
  - Limited GPU memory makes it desirable



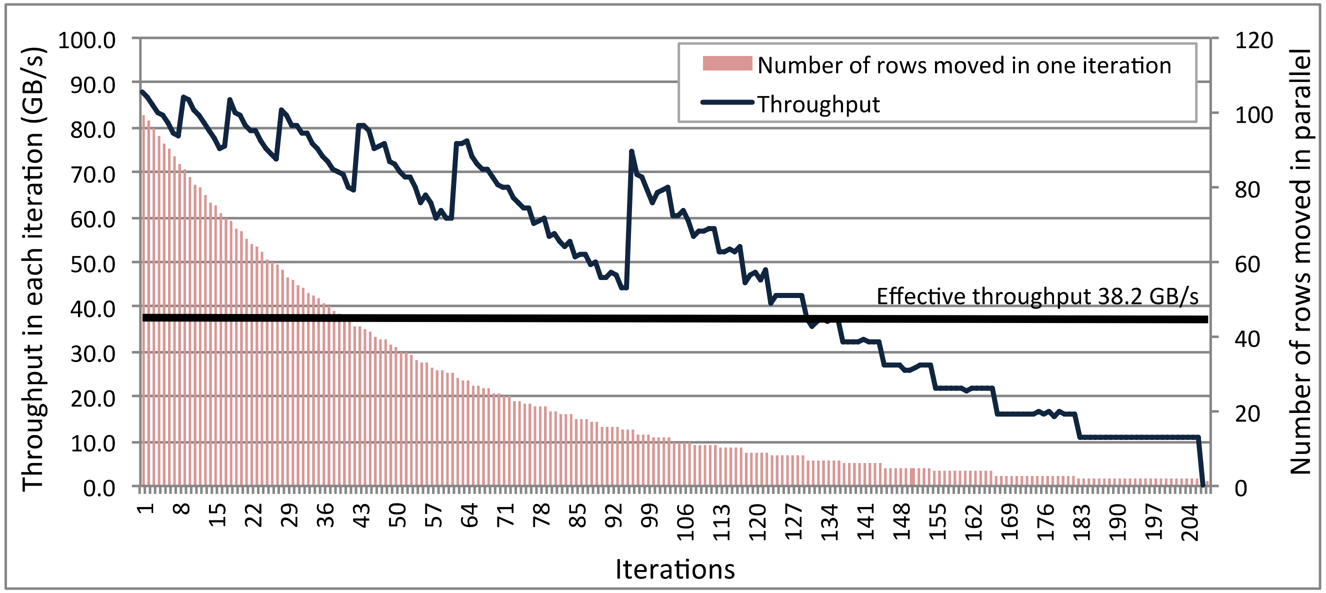
Case study: Adjacent block synchronization

- In-place matrix padding
  - Temporary storage into on-chip memory
  - Bulk synchronous programming
    - Global synchronization = kernel termination



Case study: Adjacent block synchronization

- Motivation: In-place matrix padding
  - 5000x4900 -> 5000x5000
    - Almost 100 rows moved in first iteration
    - 181 iterations with some parallelism
    - Last 99 iterations moved sequentially
  - Effective throughput only less than 20% peak bw.

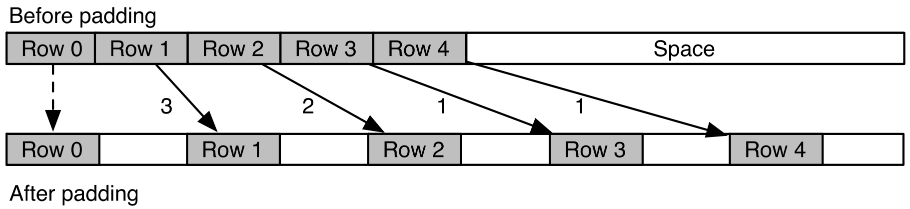


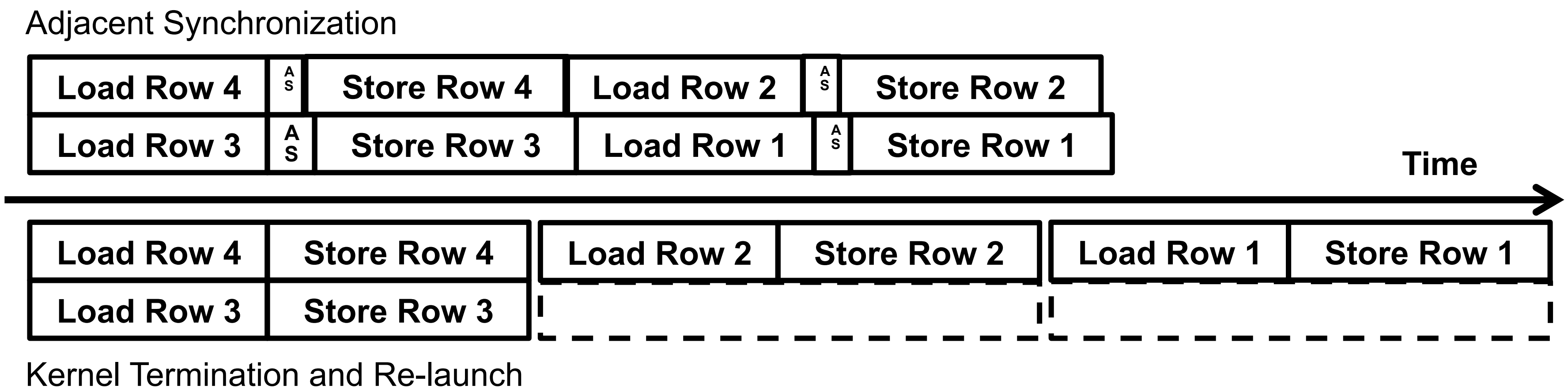
Case study: Adjacent block synchronization

- Regular Data Sliding
  - Dynamic thread block id allocation
    - Avoids deadlocks
  - Loading stage
    - Coarsening factor
  - Adjacent thread block synchronization
    - Avoids kernel termination and relaunch
  - Storing stage

Case study: Adjacent block synchronization

- Timing comparison of the two approaches





Case study: Adjacent block synchronization

- Regular Data Sliding
  - Adjacent block synchronization (Yan *et al.*, 2013)
    - Leader thread waits for previous block flag set
    - Avoids kernel termination and relaunch

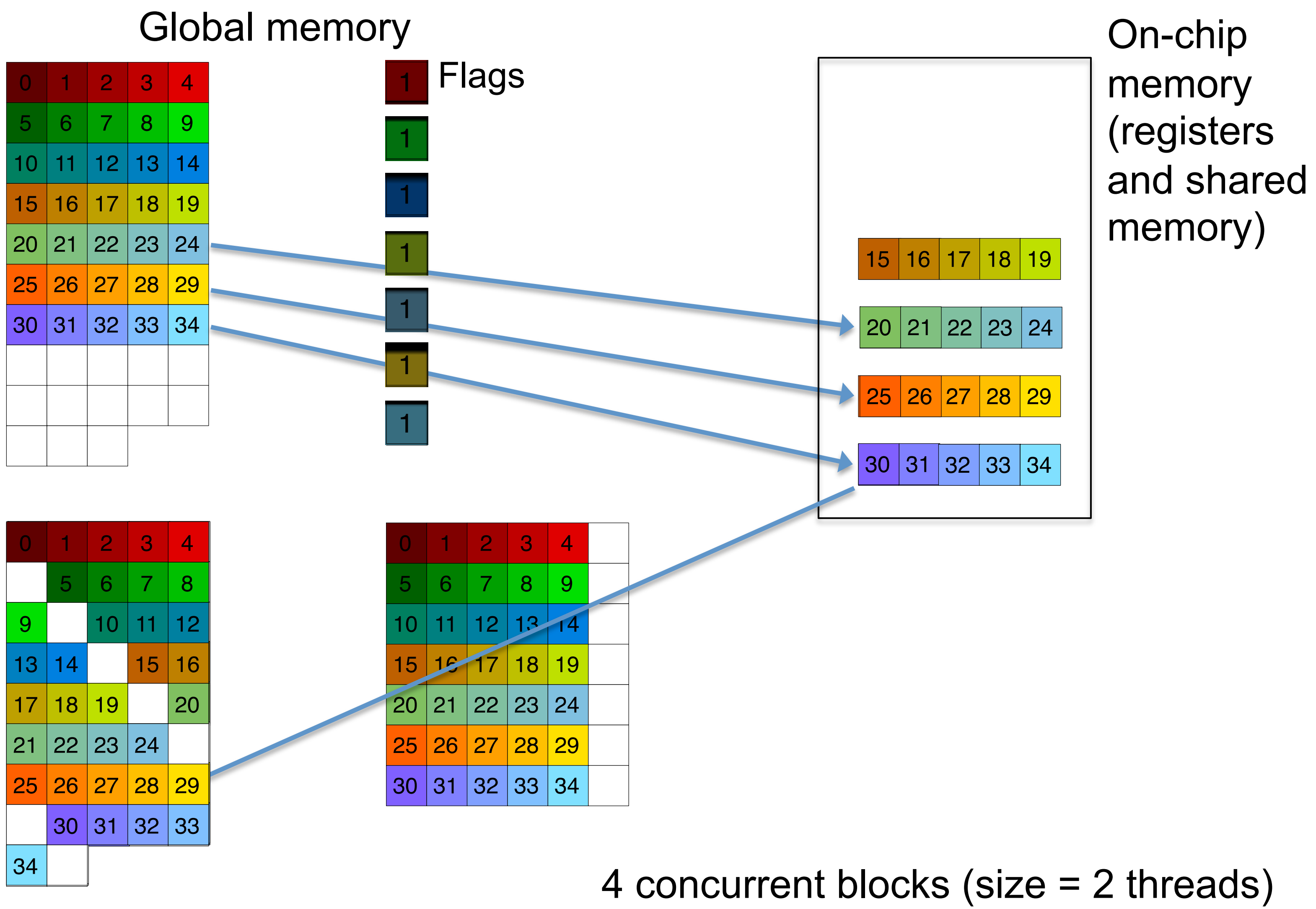
```
__synchthreads();
if (tid == 0){
    // Wait
    while(atomicOr(&flags[bid_ - 1], 0) == 0){;}
    // Set flag
    atomicOr(&flags[bid_], 1);
}
__synchthreads();
```

Case study: Adjacent block synchronization

- Regular Data Sliding
  - Dynamic block id allocation
    - Avoids deadlocks

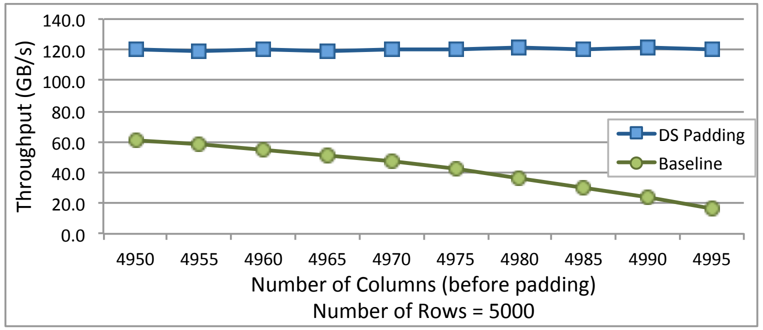
```
__shared__ int bid_;
if (tid == 0)
    bid_ = atomicAdd(&S, 1);
__syncthreads;
```

Case study: Adjacent block synchronization

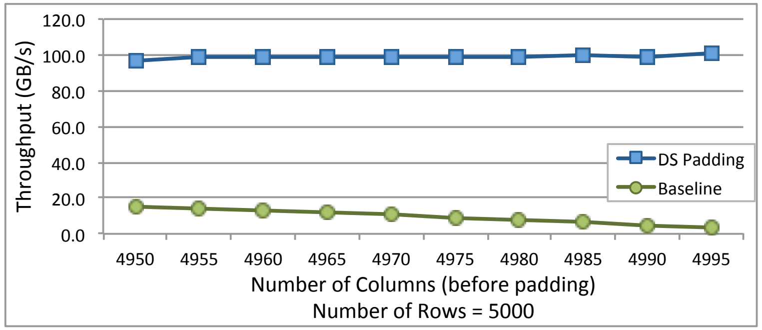


Case study: Adjacent block synchronization

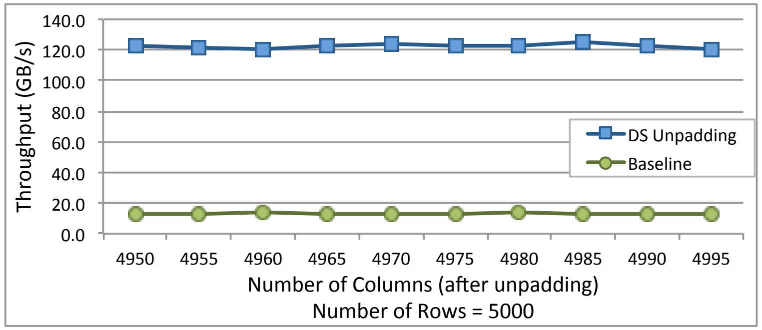
- Regular Data Sliding: Padding and Unpadding
  - Baseline = bulk synchronous implementation (Motivation)
  - Up to 9.11x (Maxwell) and up to 73.25x (Hawaii)



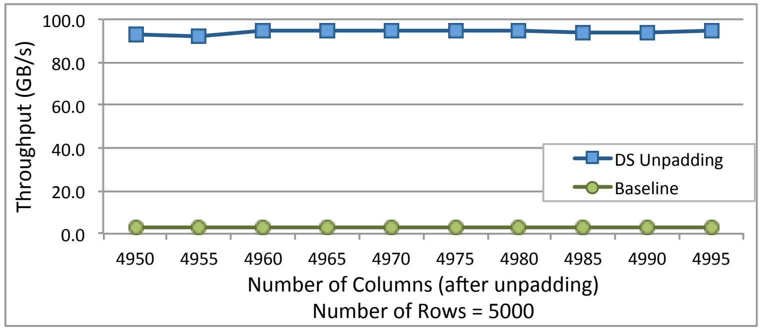
Padding on NVIDIA Maxwell



Padding on AMD Hawaii



Unpadding on NVIDIA Maxwell



Unpadding on AMD Hawaii

Case study: Adjacent block synchronization

- Irregular Data Sliding
  - Dynamic block id allocation
  - Loading stage
    - **Local counter**
  - **Reduction**
  - Adjacent block synchronization
  - Storing stage
    - **Binary prefix-sum within the thread block**

Case study: Adjacent block synchronization

- Irregular Data Sliding
  - Adjacent block synchronization
    - `count` (shared memory variable) contains reduction result
    - `flag+count` is prefix sum of blocks' reductions
    - `count = flag` makes it visible to all threads in block

```
__syncthreads();
if (tid == 0){
    // Wait
    while(atomicOr(&flags[bid_ - 1], 0) == 0){;}
    // Set flag
    int flag = flags[bid_ - 1];
    atomicAdd(&flags[bid_], flag + count);
    count = flag;
}
__syncthreads();
```

Case study: Adjacent block synchronization

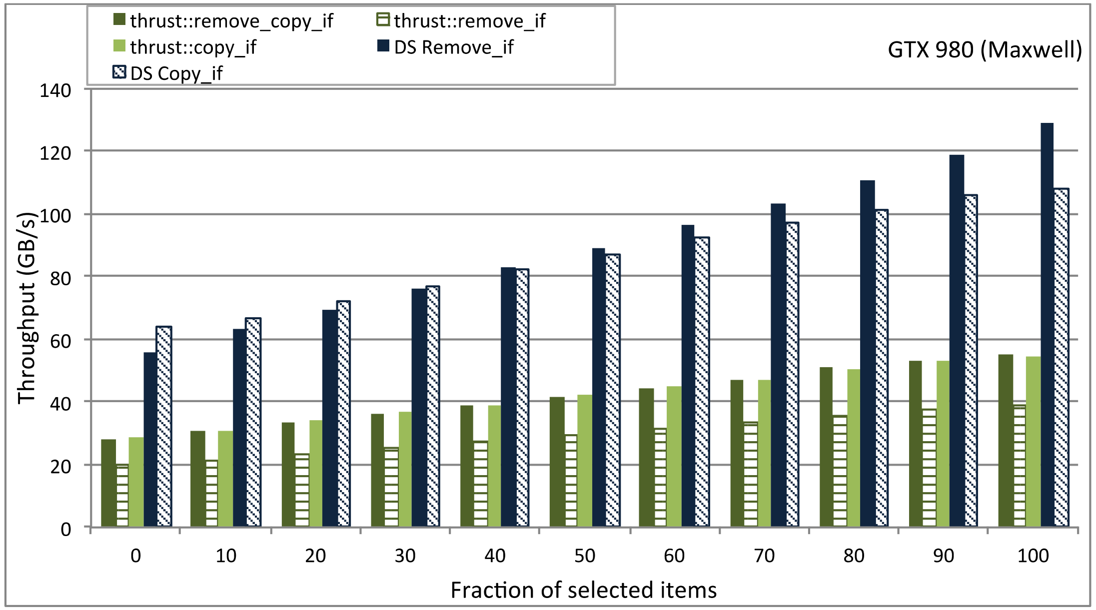
- Irregular Data Sliding: Select

Select (remove)

Input												
Input	2	1	3	0	0	1	3	4	0	0	2	1

Predicate: True if it is even

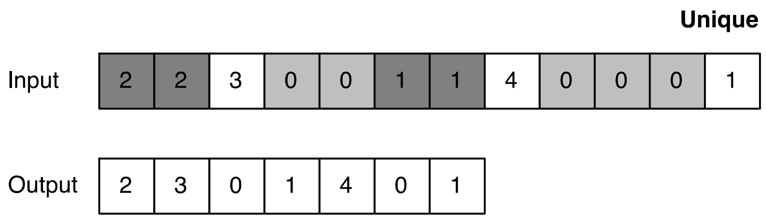
Output	1	3	1	3	1



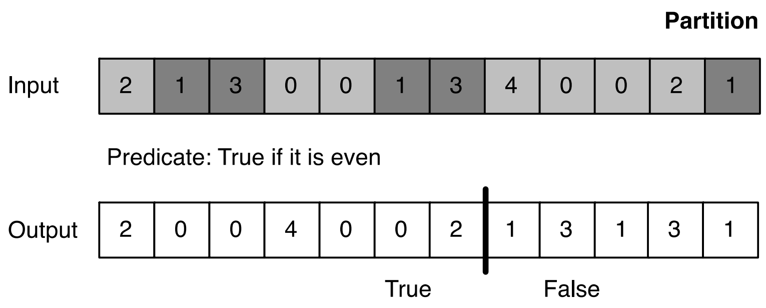
Up to 3.05x Thrust on Maxwell
2.80x on Kepler
1.78x on Fermi

Case study: Adjacent block synchronization

- Irregular Data Sliding
  - Stream compaction
    - Our Ip stable implementation is 68% of the fastest Oop unstable kernel
  - Unique
    - Up to 3.24x Thrust on Maxwell
    - 2.73x on Kepler
    - 1.66x on Fermi



  - Partition
    - Up to 2.84x Thrust on Maxwell
    - 2.88x on Kepler
    - 1.64x on Fermi



Summary

- Significant hardware improvements for atomic operations
  - Shared memory: Native integer atomics
  - Global memory: L2 + Buffer in ROPs
- They can free programmers from applying software optimization
  - Histogramming
- They may allow a more natural way of coding, saving many lines of code
  - Reduction
- They may allow using new, faster algorithms
  - Filtering
  - Adjacent synchronization

University of Illinois at Urbana-Champaign. ECE 408. October 22, 2015

Atomic Operations across GPU generations

Juan Gómez-Luna
University of Córdoba (Spain)
el1goluj@uco.es gomezlun@illinois.edu