Computer Architecture

Lecture 14: New Programming Features in Heterogeneous Systems

Juan Gómez Luna
ETH Zürich

Fall 2017

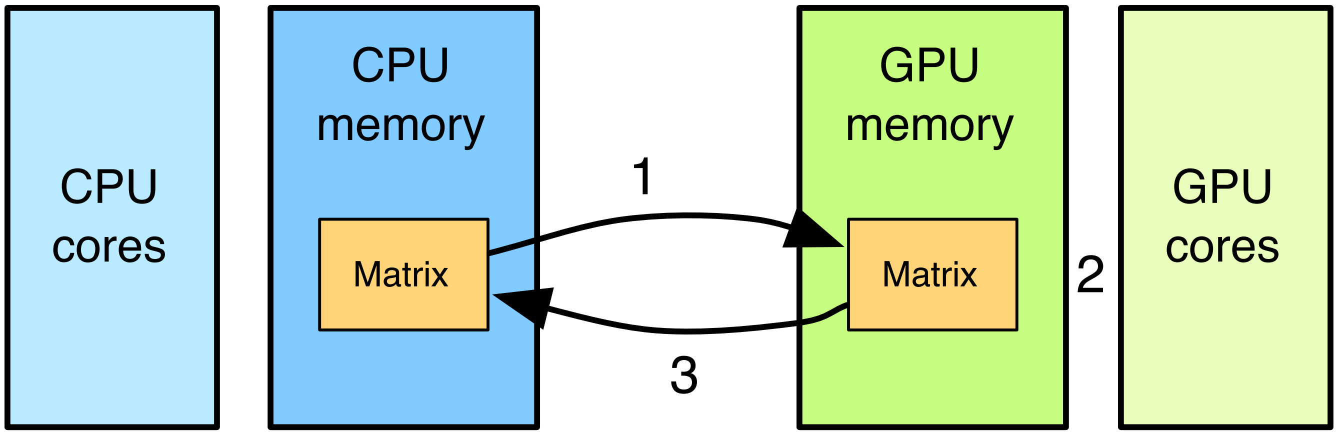
8 November 2017

Agenda for Today

- Traditional accelerator model
  - Review: Program structure
  - Review: Memory hierarchy and memory management
  - Review: Performance considerations
    - Memory access
    - SIMD utilization
  - Atomic operations
  - Data transfers
- New programming features
  - Collaborative computing
  - Dynamic parallelism

Review: GPU Computing

- Computation is offloaded to the GPU
- Three steps
  - CPU-GPU data transfer (1)
  - GPU kernel execution (2)
  - GPU-CPU data transfer (3)



Review: Traditional Program Structure

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU

Serial Code (host)

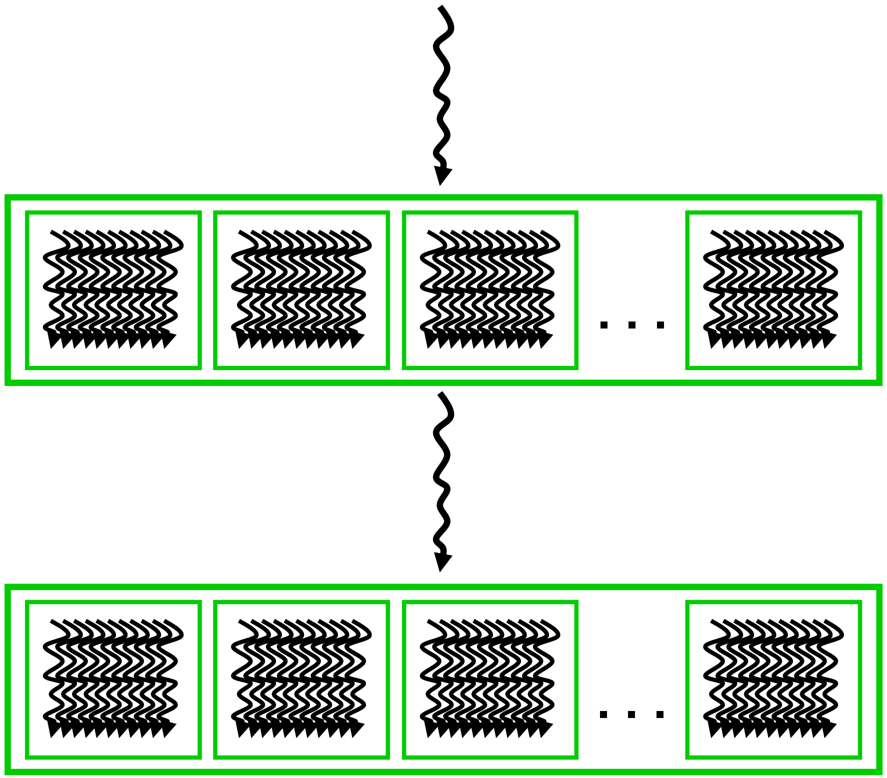
Parallel Kernel (device)

```
KernelA<<< nBlk, nThr >>>(args);
```

Serial Code (host)

Parallel Kernel (device)

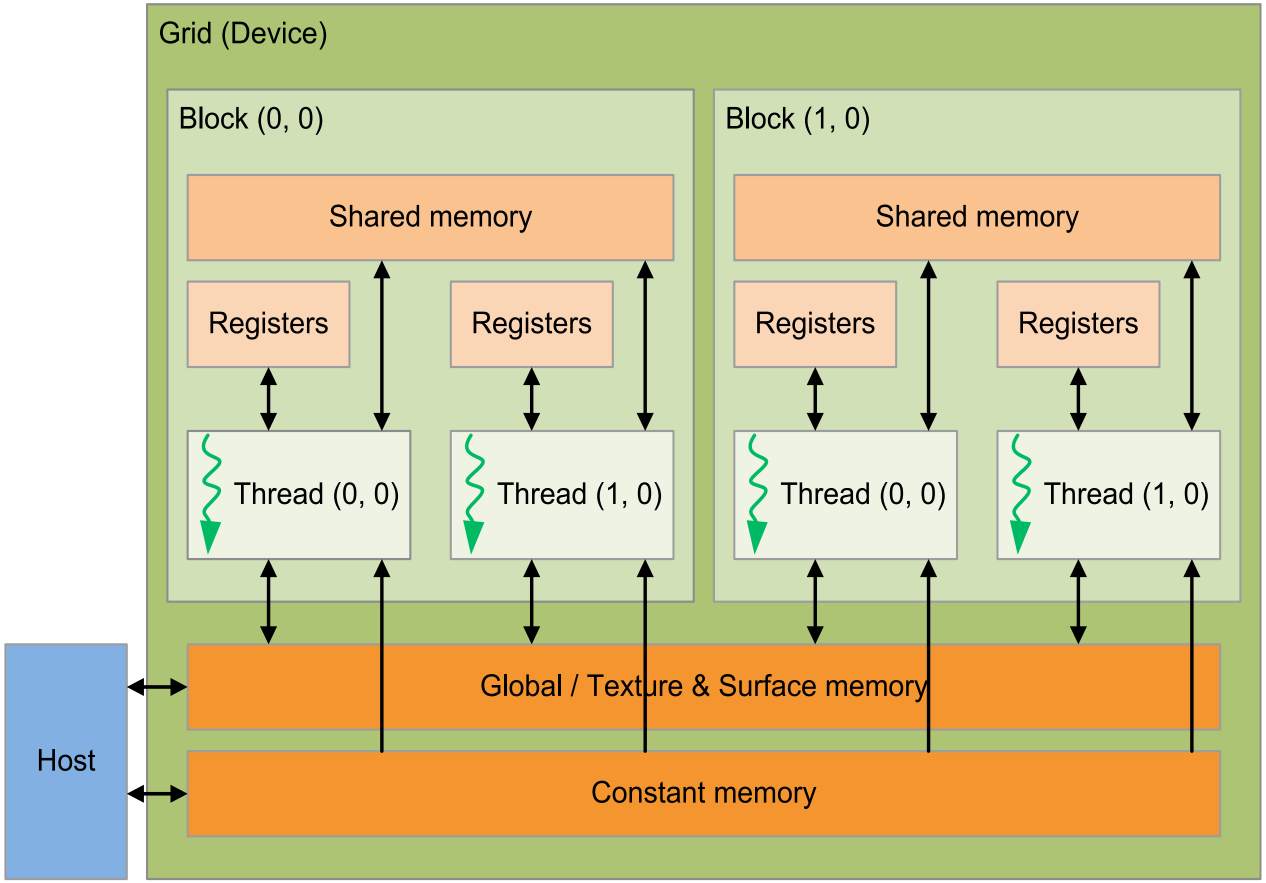
```
KernelB<<< nBlk, nThr >>>(args);
```



Slide credit: Hwu & Kirk

Review: CUDA/OpenCL Programming Model

- Memory hierarchy



Review: Traditional Program Structure

- Function prototypes

  `float serialFunction(…);`

  `__global__ void kernel(…);`

- `main()`

  1) Allocate memory space on the device – `cudaMalloc(&d_in, bytes);`
  2) Transfer data from host to device – `cudaMemCpy(d_in, h_in, …);`
  3) Execution configuration setup: #blocks and #threads
  4) Kernel call – `kernel<<<execution configuration>>>(args…);`
  5) Transfer results from device to host – `cudaMemCpy(h_out, d_out, …);`

repeat as needed

- Kernel – `__global__ void kernel(type args,…)`
  - Automatic variables transparently assigned to registers
  - Shared memory – `__shared__`
  - Intra-block synchronization – `__syncthreads();`

Slide credit: Hwu & Kirk

Review: CUDA Programming Language

- Memory allocation

```
cudaMalloc((void**)&d_in, #bytes);
```

- Memory copy

```
cudaMemcpy(d_in, h_in, #bytes,
           cudaMemcpyHostToDevice);
```

- Kernel launch

```
kernel<<< #blocks, #threads >>>(args);
```

- Memory deallocation

```
cudaFree(d_in);
```

- Explicit synchronization

```
cudaDeviceSynchronize();
```

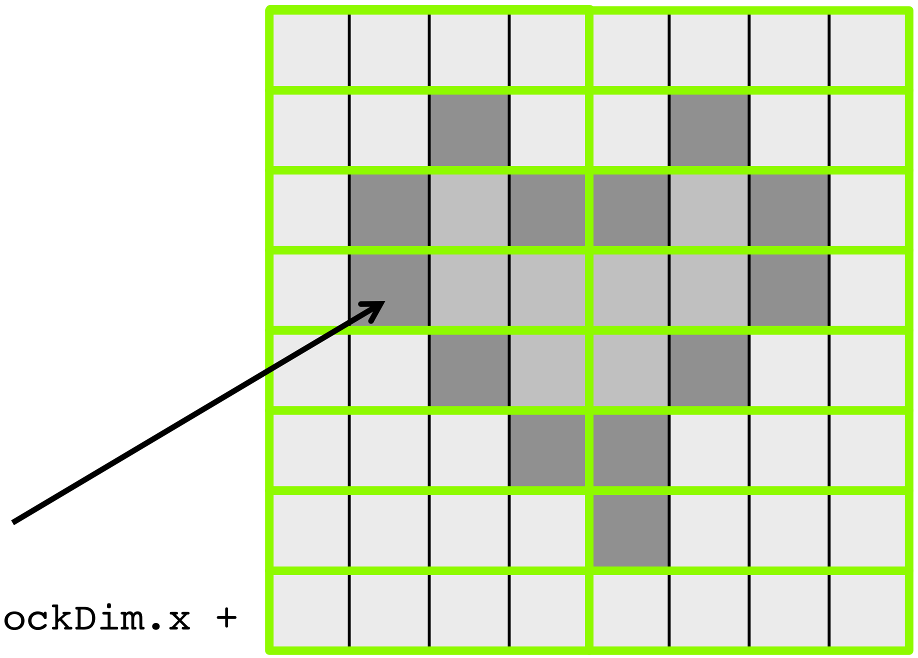
Review: Indexing and Memory Access

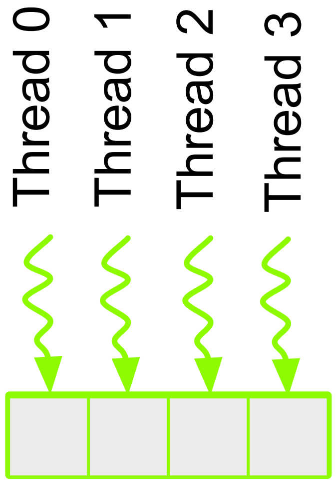
- One GPU thread per pixel
- Grid of Blocks of Threads
  - `blockIdx.x, threadIdx.x`
  - `gridDim.x, blockDim.x`

`blockIdx.x`

`threadIdx.x`

Block 0





Block 0

6 * 4 + 1 = 25

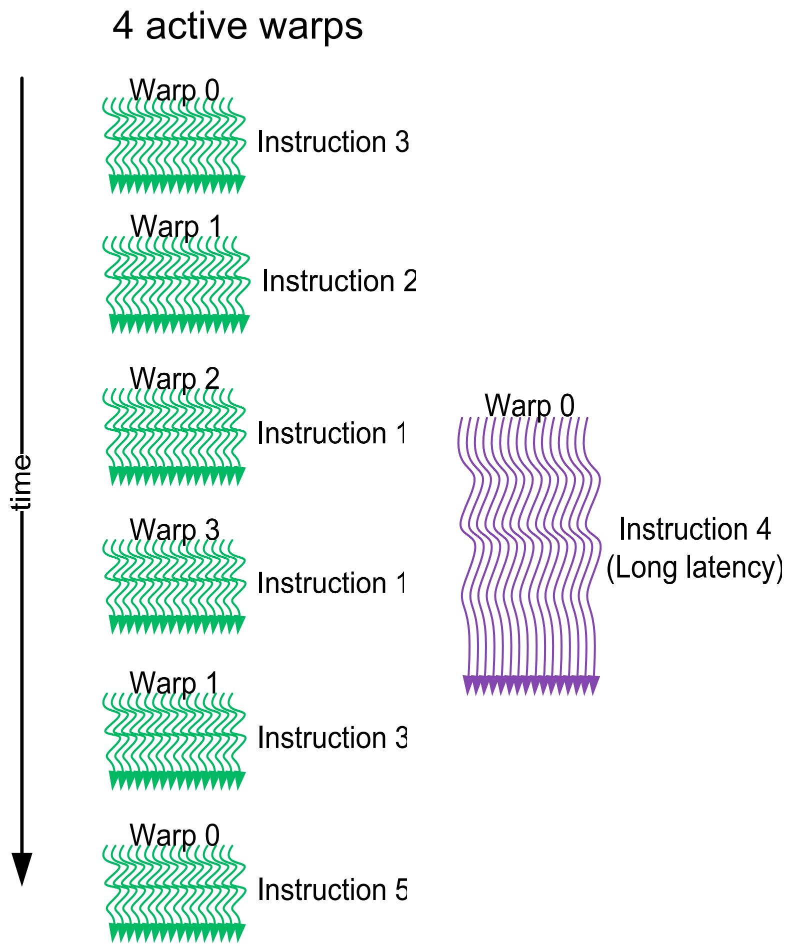
`blockIdx.x * blockDim.x + threadIdx.x`

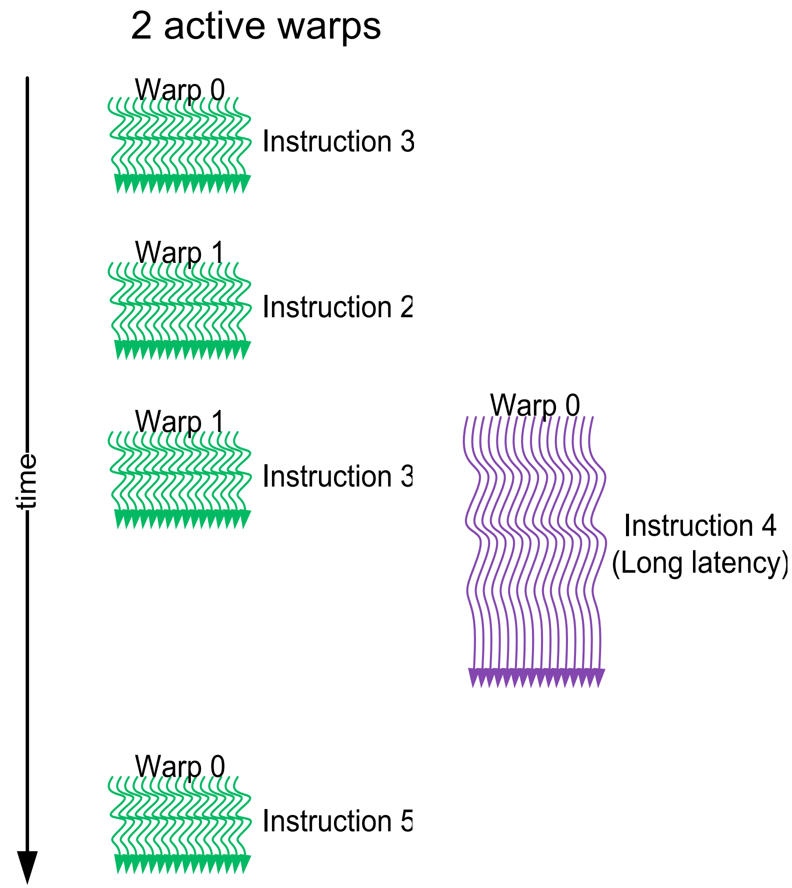
Review: Performance Considerations

- Main bottlenecks
  - Global memory access
  - CPU-GPU data transfers
- Memory access
  - Latency hiding
    - Thread Level Parallelism (TLP)
    - Occupancy
  - Memory coalescing
  - Data reuse
    - Shared memory usage
- SIMD Utilization
- Atomic operations
- Data transfers between CPU and GPU
  - Overlap of communication and computation

Review: Latency Hiding

- Occupancy: ratio of active warps
  - Not only memory accesses (e.g., SFU)



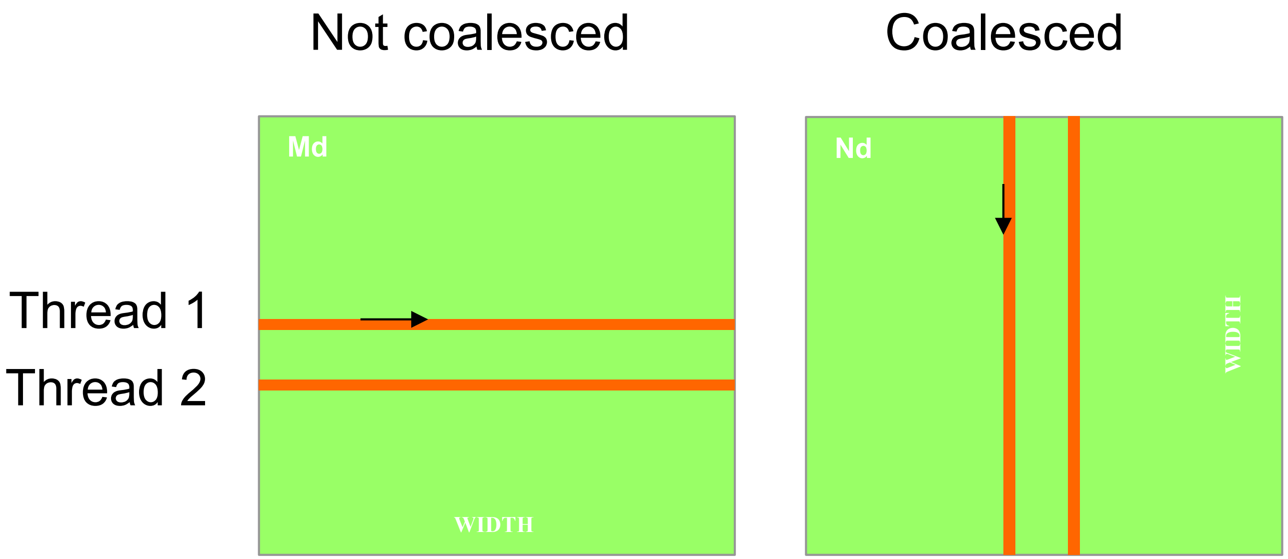


Review: Occupancy

- SM resources (typical values)
  - Maximum number of warps per SM (64)
  - Maximum number of blocks per SM (32)
  - Register usage (256KB)
  - Shared memory usage (64KB)
- Occupancy calculation
  - Number of threads per block
  - Registers per thread
  - Shared memory per block
- The number of registers per thread is known in compile time

Review: Memory Coalescing

- When accessing global memory, peak bandwidth utilization occurs when all threads in a warp access one cache line



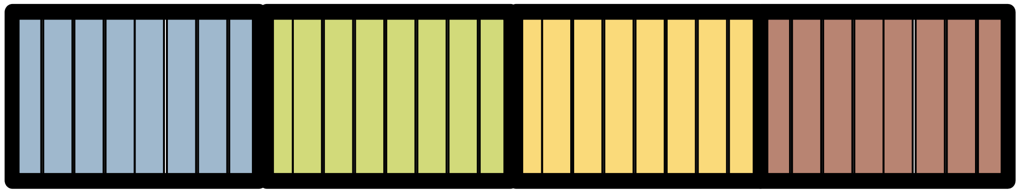
Slide credit: Hwu & Kirk

Review: Memory Coalescing

- AoS vs. SoA

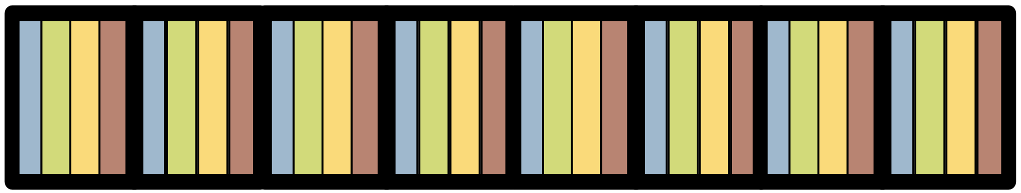
Structure of Arrays (SoA)

```
struct foo{
 float a[8];
 float b[8];
 float c[8];
 int d[8];
} A;
```



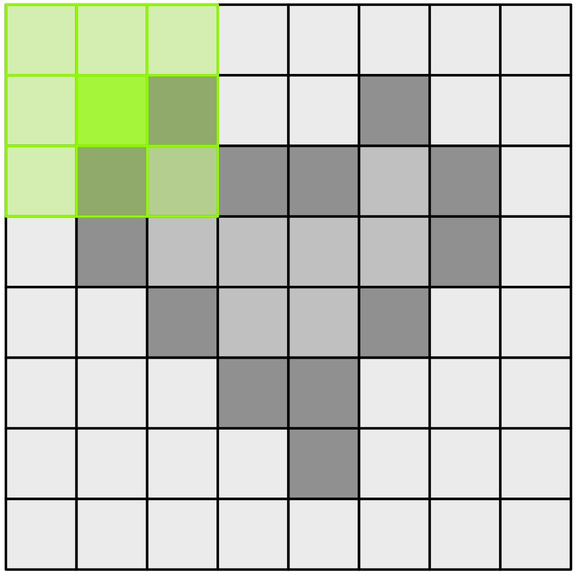
Array of Structures (AoS)

```
struct foo{
 float a;
 float b;
 float c;
 int d;
} A[8];
```



Review: Data Reuse

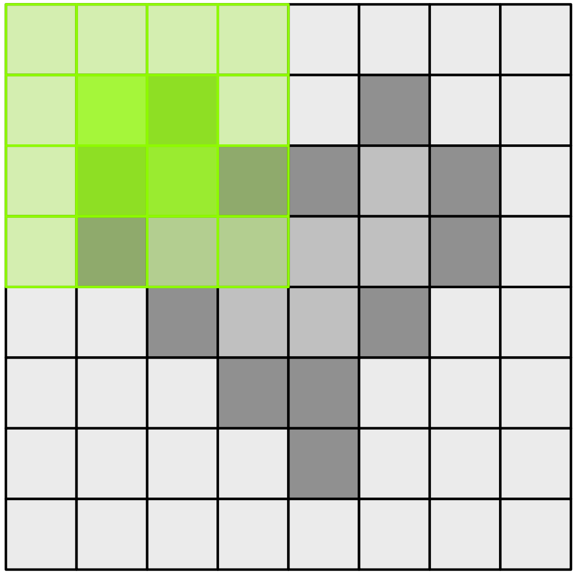
- Same memory locations accessed by neighboring threads



```
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
    }
}
```

Review: Data Reuse

- Shared memory tiling



```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
…
Load tile into shared memory
__synchthreads();
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
  }
}
```

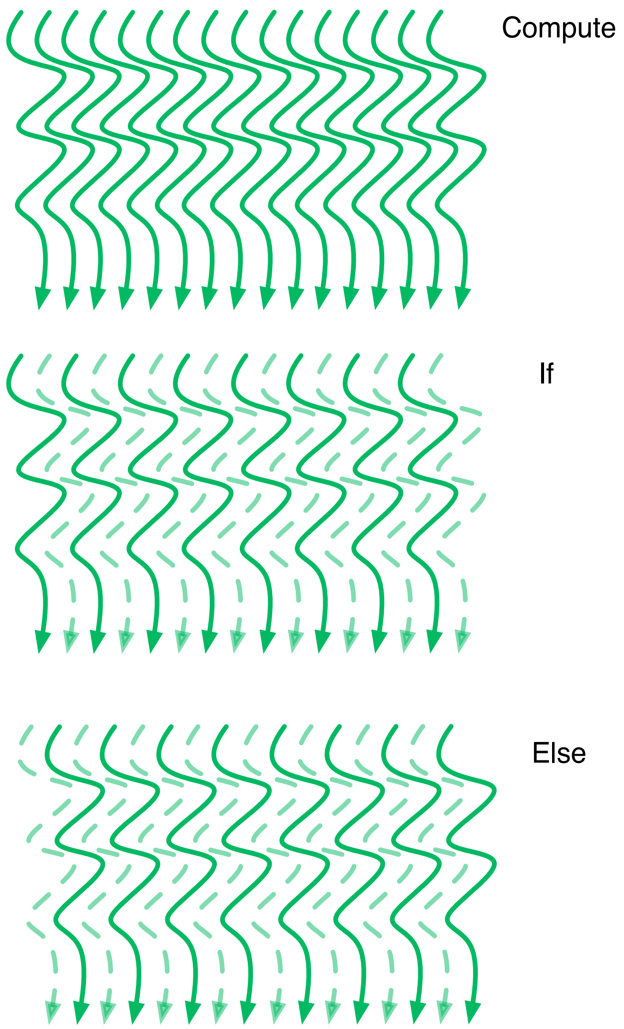
Review: Shared Memory

- Shared memory is an interleaved memory
  - Typically 32 banks
  - Each bank can service one address per cycle
  - Successive 32-bit words are assigned to successive banks
    - Bank = Address % 32
- Bank conflicts are only possible within a warp
  - No bank conflicts between different warps

Review: SIMD Utilization

- Intra-warp divergence

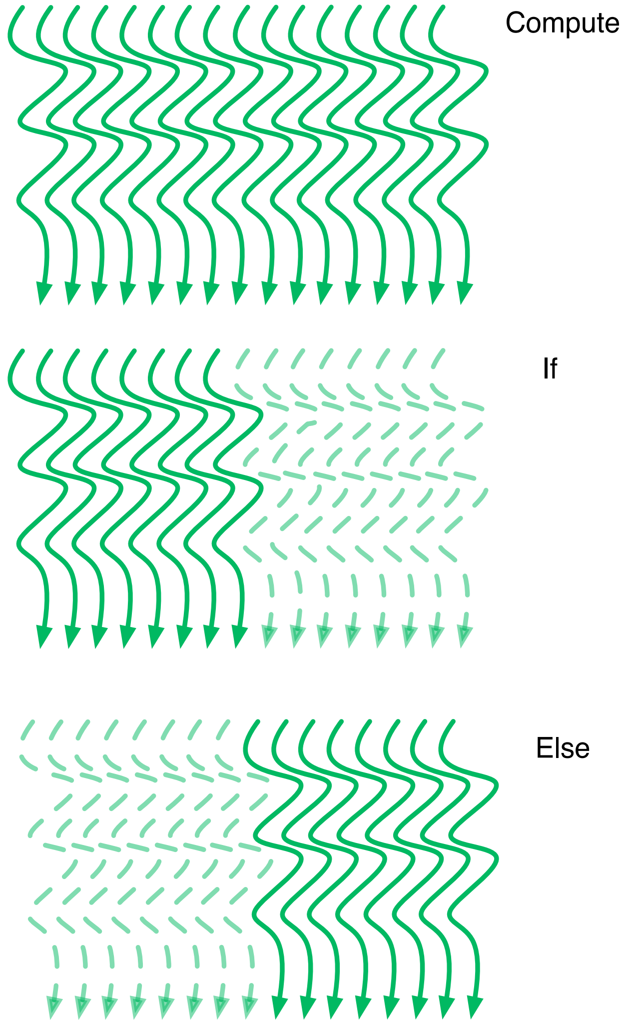
```
Compute(threadIdx.x);
if (threadIdx.x % 2 == 0){
  Do_this(threadIdx.x);
}
else{
  Do_that(threadIdx.x);
}
```



Review: SIMD Utilization

- Intra-warp divergence

```
Compute(threadIdx.x);
if (threadIdx.x < 32){
  Do_this(threadIdx.x * 2);
}
else{
  Do_that((threadIdx.x%32)*2+1);
}
```



Atomic Operations

- Shared memory atomic operations
  - 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;
```

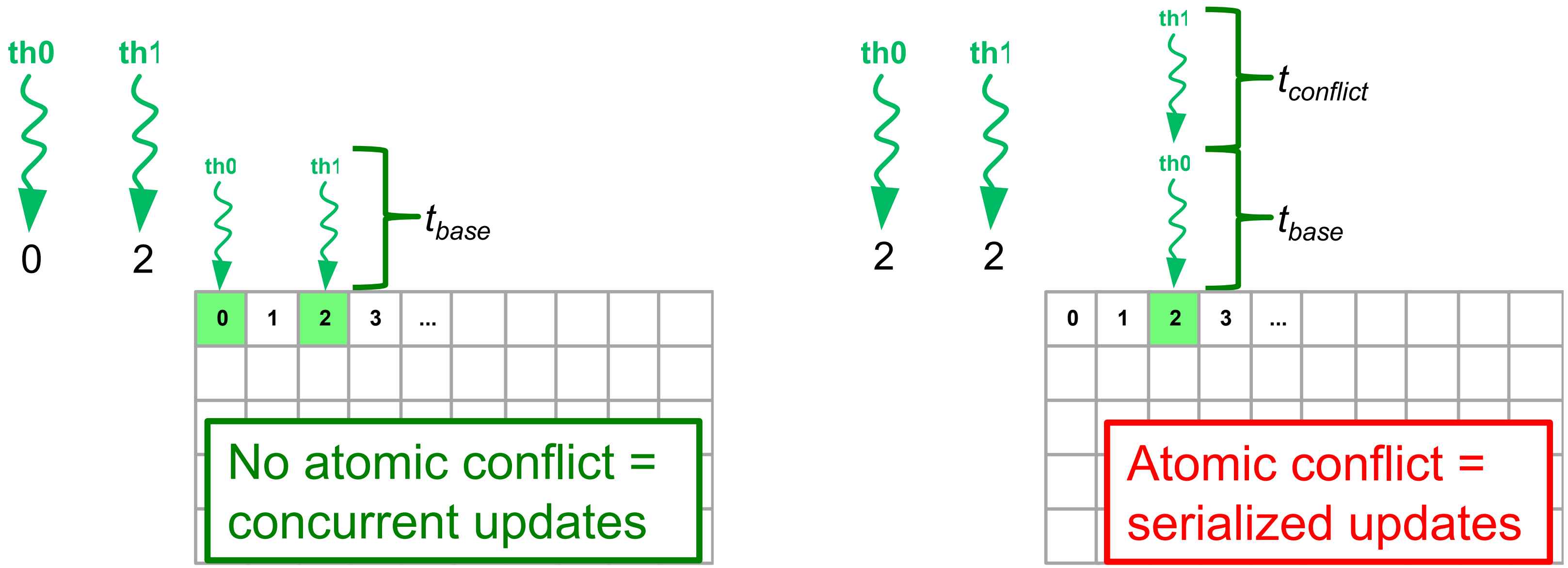
Maxwell

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

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

Atomic Operations

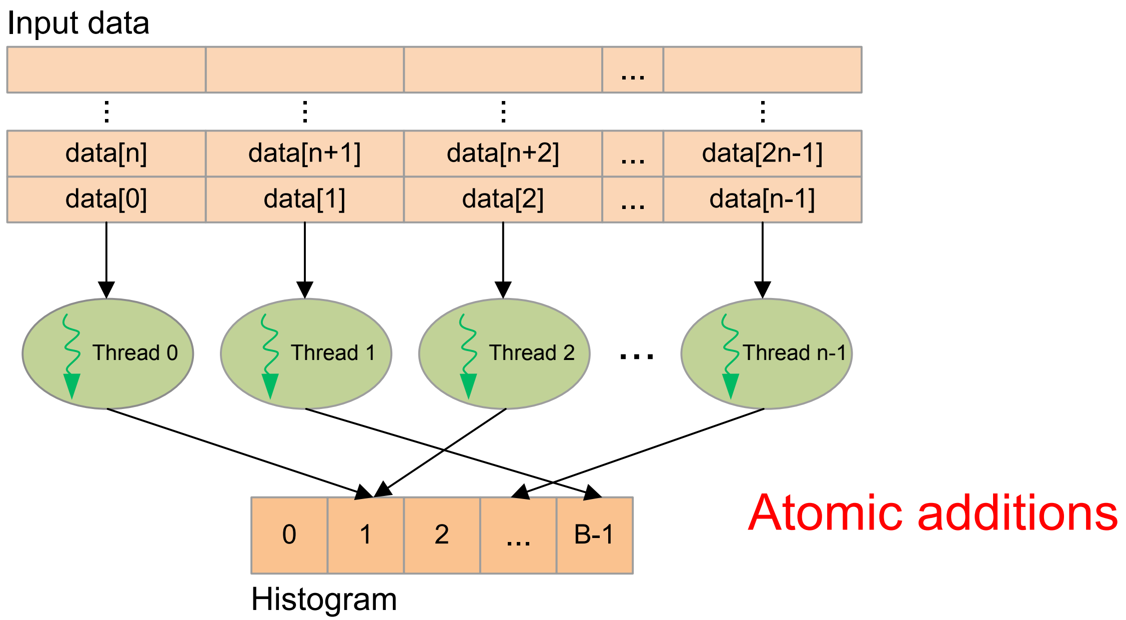
- Atomic conflicts
  - Intra-warp conflict degree from 1 to 32



Histogram Calculation

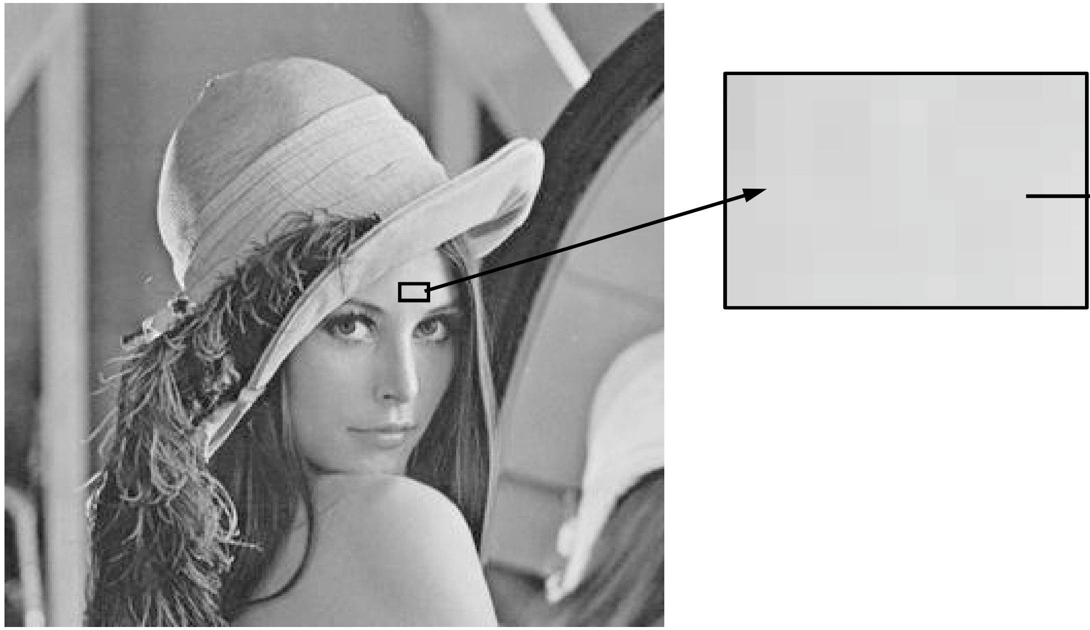
- Histograms count the number of data instances in disjoint categories (bins)

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



Histogram Calculation

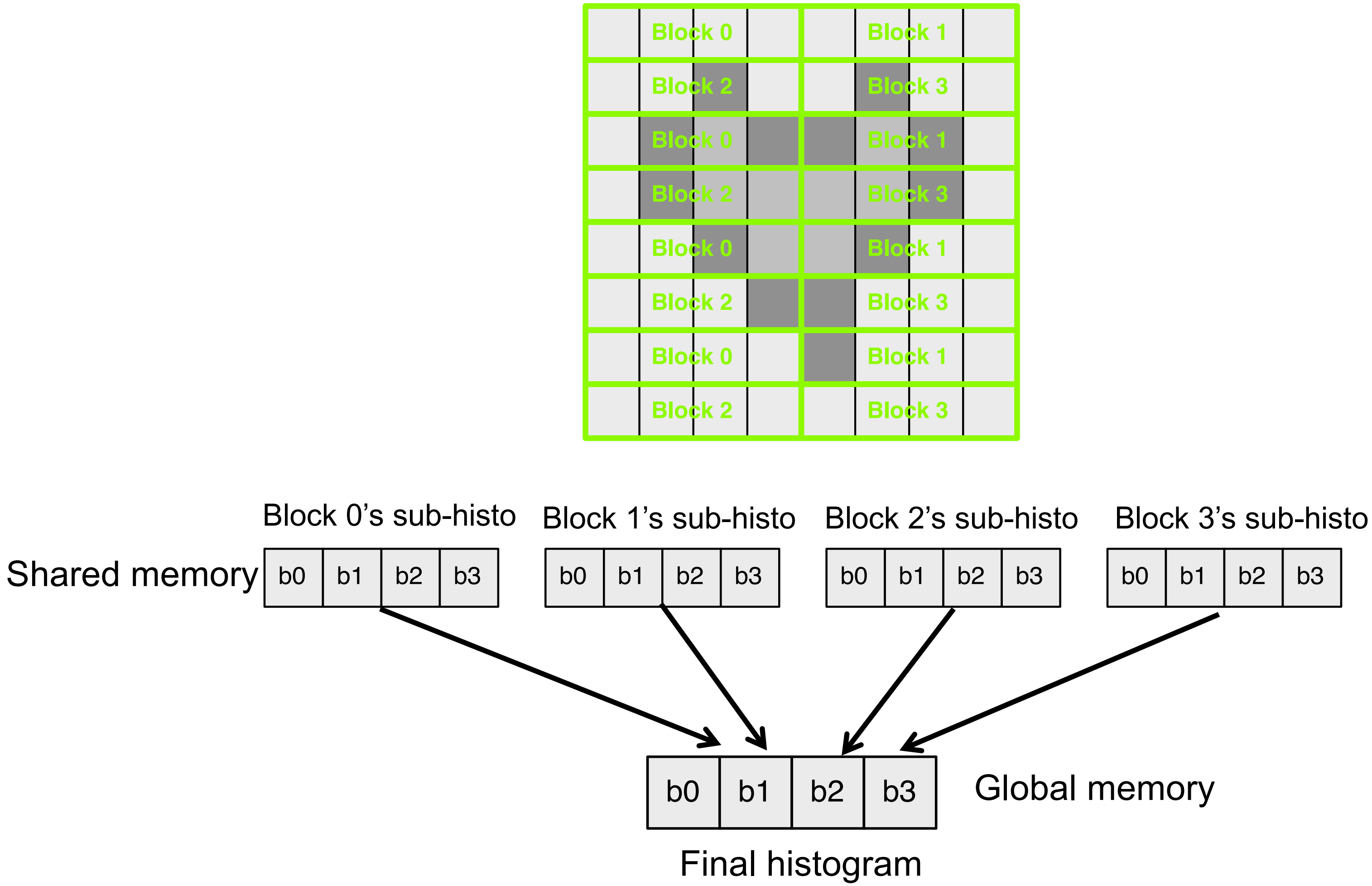
- Frequent conflicts in natural images



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

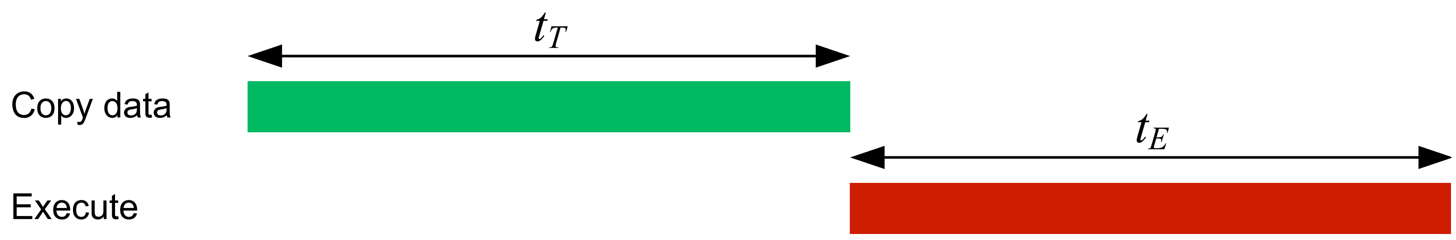
Histogram Calculation

- Privatization: Per-block sub-histograms in shared memory



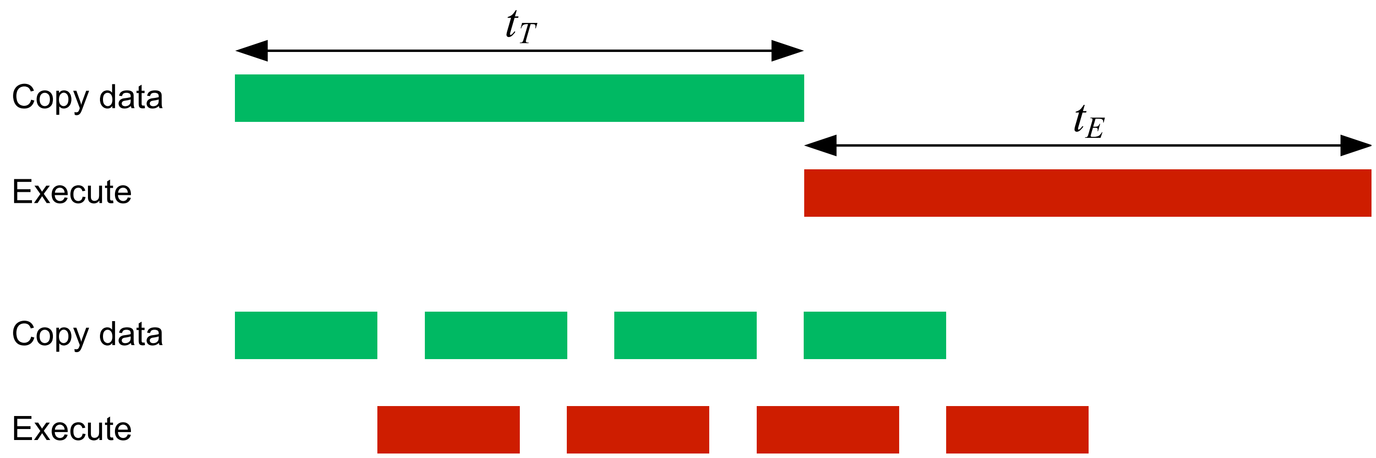
Data Transfers

- Synchronous and asynchronous transfers
- Streams (Command queues)
  - Sequence of operations that are performed in order
    - CPU-GPU data transfer
    - Kernel execution
      - D input data instances, B blocks
    - GPU-CPU data transfer
  - Default stream



Asynchronous Transfers

- Computation divided into nStreams
  - D input data instances, B blocks
  - nStreams
    - D/nStreams data instances
    - B/nStreams blocks



- Estimates

$$t_E + \frac{t_T}{nStreams}$$

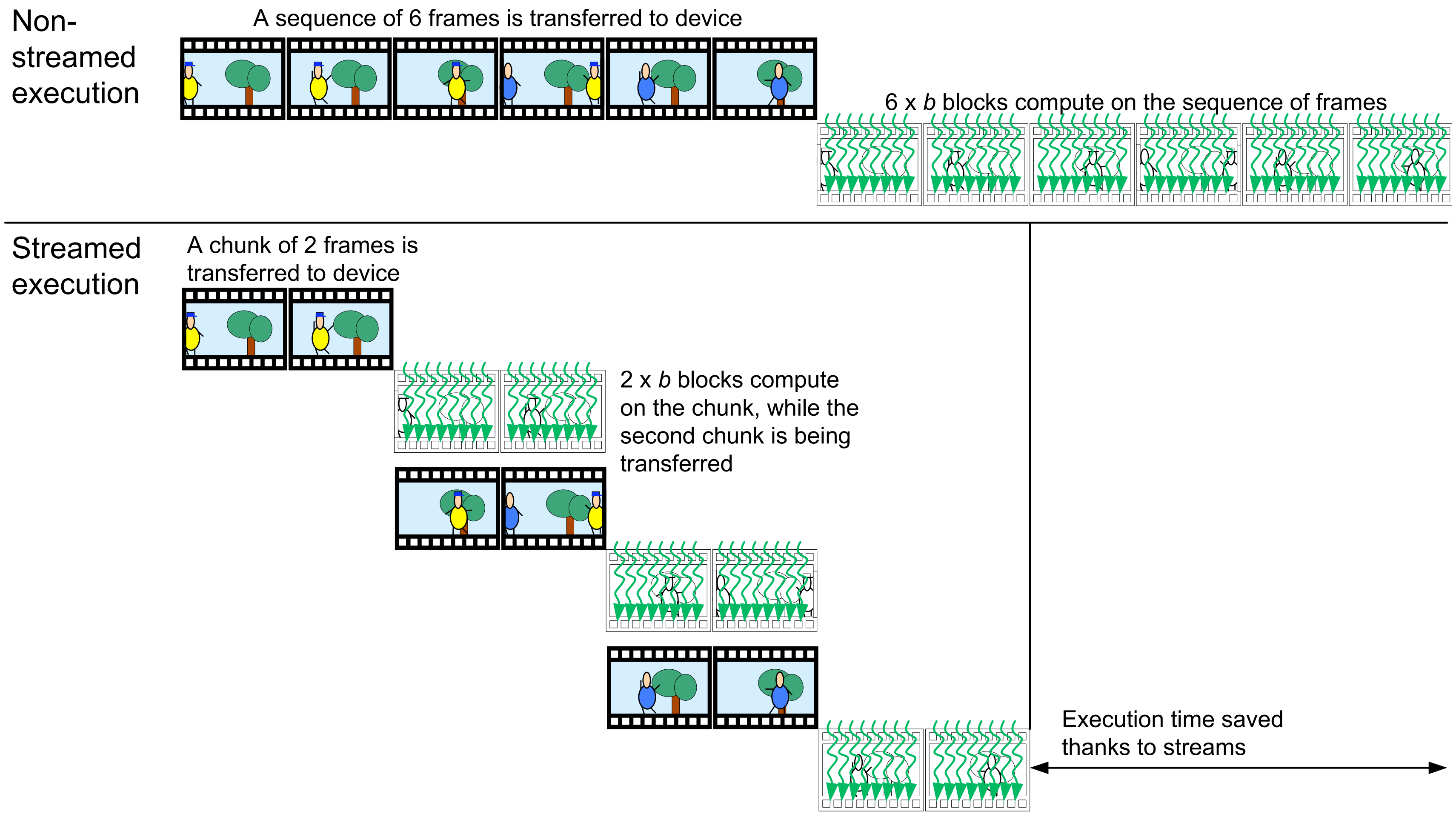
$t_E >= t_T$ (dominant kernel)

$$t_T + \frac{t_E}{nStreams}$$

$t_T > t_E$ (dominant transfers)

Asynchronous Transfers

- Overlap of communication and computation (e.g., video processing)



Summary

- Traditional accelerator model
  - Program structure
    - Bulk synchronous programming model
  - Memory hierarchy and memory management
  - Performance considerations
    - Memory access
      - Latency hiding: occupancy (TLP)
      - Memory coalescing
      - Data reuse: shared memory
    - SIMD utilization
    - Atomic operations
    - Data transfers

Collaborative Computing

Systems@ETH Zürich

SAFARI

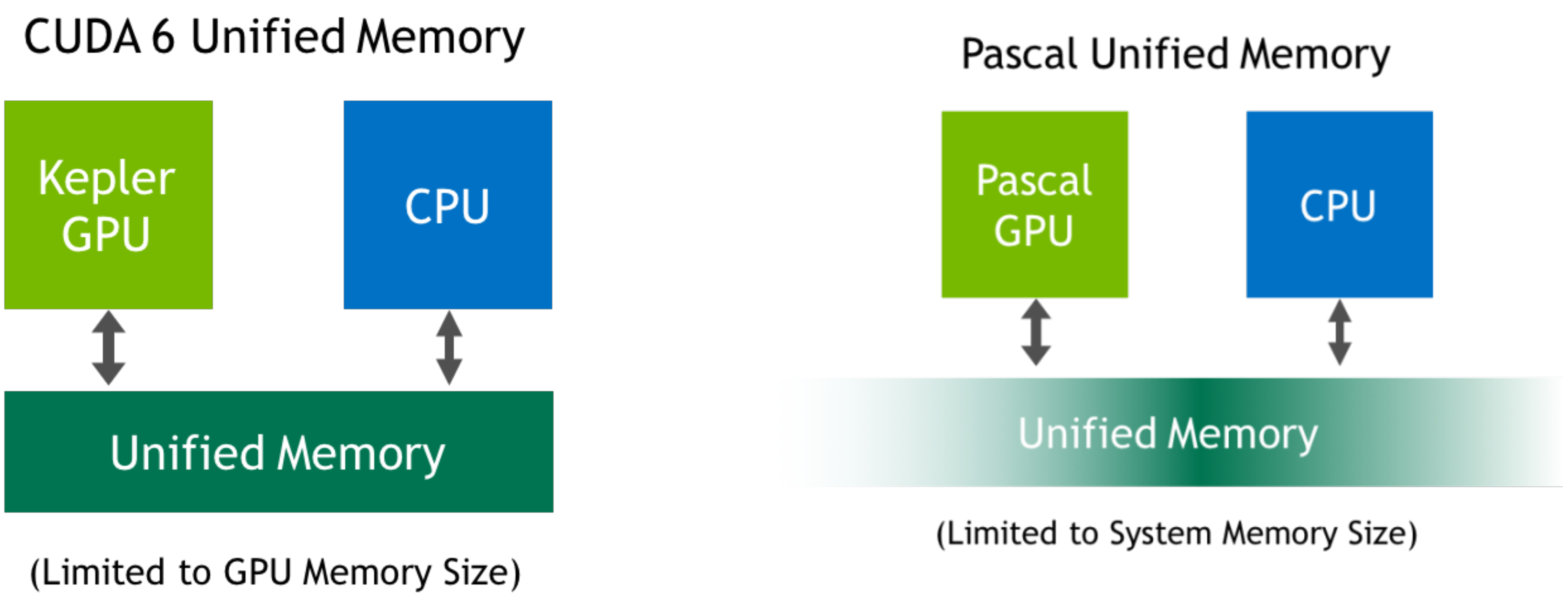
Review

- Device allocation, CPU-GPU transfer, and GPU-CPU transfer
  - `cudaMalloc();`
  - `cudaMemcpy();`

```
// Allocate input
malloc(input, ...);
cudaMalloc(d_input, ...);
cudaMemcpy(d_input, input, ..., HostToDevice); // Copy to device memory

// Allocate output
malloc(output, ...);
cudaMalloc(d_output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// Synchronize
cudaDeviceSynchronize();

// Copy output to host memory
cudaMemcpy(output, d_output, ..., DeviceToHost);
```

Unified Memory

- Unified Virtual Address
- CUDA 6.0: Unified memory
- CUDA 8.0 + Pascal: GPU page faults



Unified Memory

- Easier programming with Unified Memory
  - `cudaMallocManaged();`

```
// Allocate input
malloc(input, ...);
cudaMallocManaged(d_input, ...);
memcpy(d_input, input, ...); // Copy to managed memory

// Allocate output
cudaMallocManaged(d_output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// Synchronize
cudaDeviceSynchronize();
```

Collaborative Computing Algorithms

- Case studies using CPU and GPU
- Kernel launches are asynchronous
  - CPU can work while waits for GPU to finish
  - Traditionally, this is the most efficient way to exploit heterogeneity

```
// Allocate input
malloc(input, ...);
cudaMalloc(d_input, ...);
cudaMemcpy(d_input, input, ..., HostToDevice); // Copy to device memory

// Allocate output
malloc(output, ...);
cudaMalloc(d_output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// CPU can do things here

// Synchronize
cudaDeviceSynchronize();

// Copy output to host memory
cudaMemcpy(output, d_output, ..., DeviceToHost);
```

Fine-Grained Heterogeneity

- Fine-grain heterogeneity becomes possible with Pascal/Volta architecture
- Pascal/Volta Unified Memory
  - CPU-GPU memory coherence
  - System-wide atomic operations

```
// Allocate input
cudaMallocManaged(input, ...);

// Allocate output
cudaMallocManaged(output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (output, input, ...);

// CPU can do things here
output[x] = input[y];

output[x+1].fetch_add(1);
```

CUDA 8.0

- Unified memory

```
cudaMallocManaged(&h_in, in_size);
```

- System-wide atomics

```
old = atomicAdd_system(&h_out[x], inc);
```

OpenCL 2.0

- Shared virtual memory

```
XYZ * h_in = (XYZ *)clSVMAlloc(
           ocl.clContext, CL_MEM_SVM_FINE_GRAIN_BUFFER, in_size, 0);
```

- More flags:

```
CL_MEM_READ_WRITE
CL_MEM_SVM_ATOMICS
```

- C++11 atomic operations

  (`memory_scope_all_svm_devices`)

```
old = atomic_fetch_add(&h_out[x], inc);
```

C++AMP (HCC)

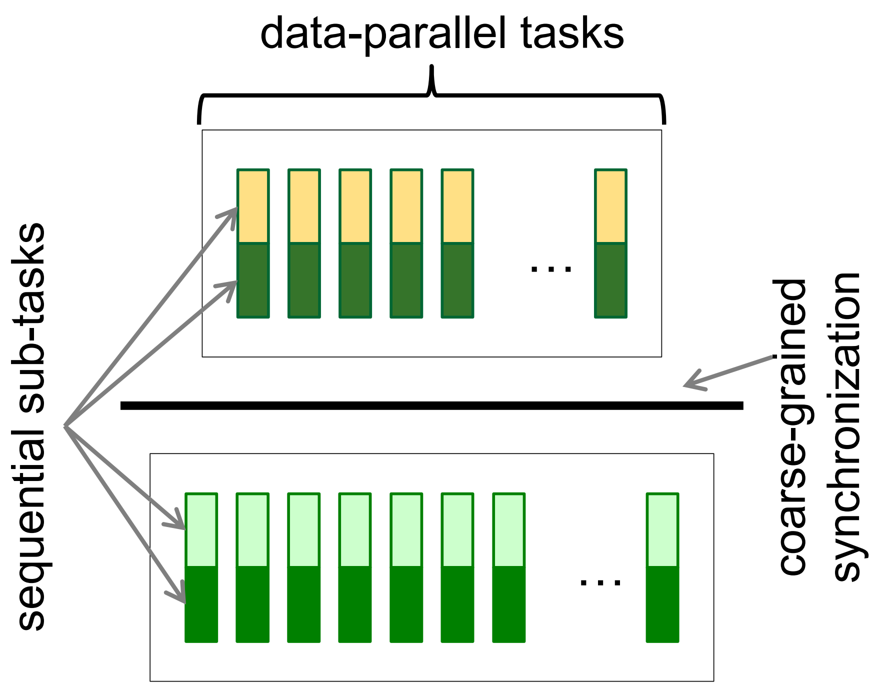
- Unified memory space (HSA)

```
XYZ *h_in = (XYZ *)malloc(in_size);
```

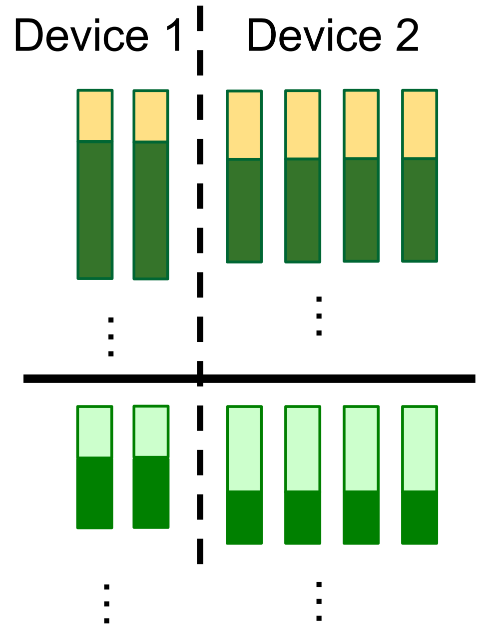
- C++11 atomic operations (`memory_scope_all_svm_devices`)
  - Platform atomics (HSA)

```
old = atomic_fetch_add(&h_out[x], inc);
```

Collaborative Patterns

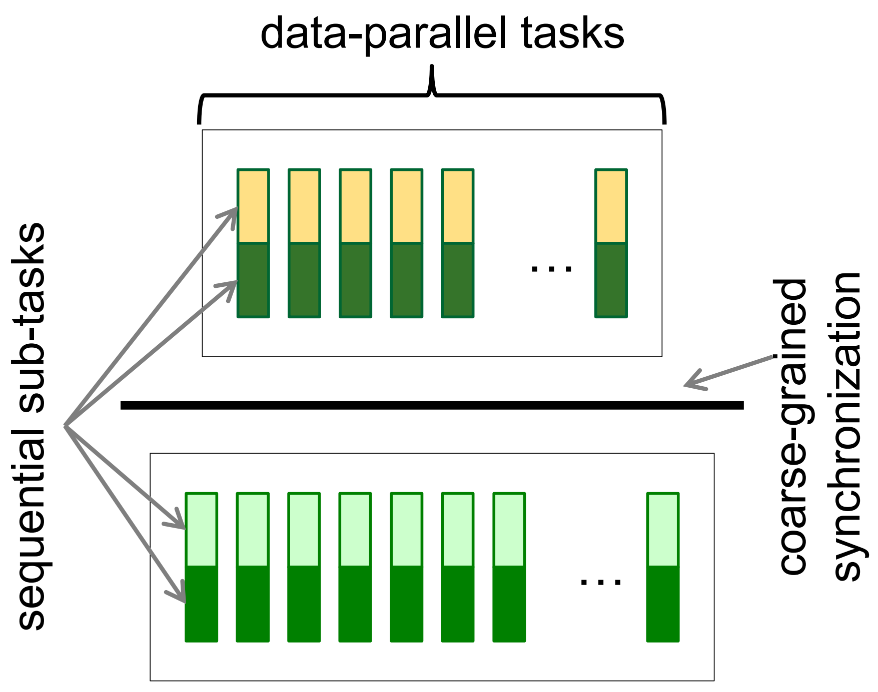


Program Structure

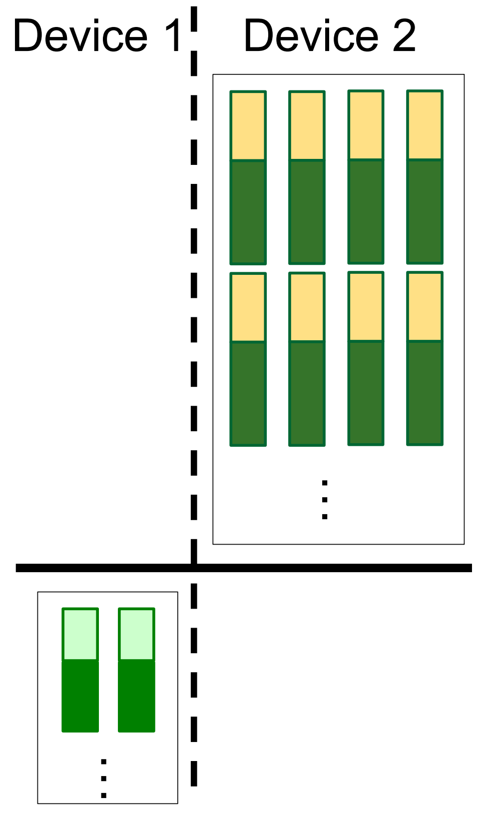


Data Partitioning

Collaborative Patterns

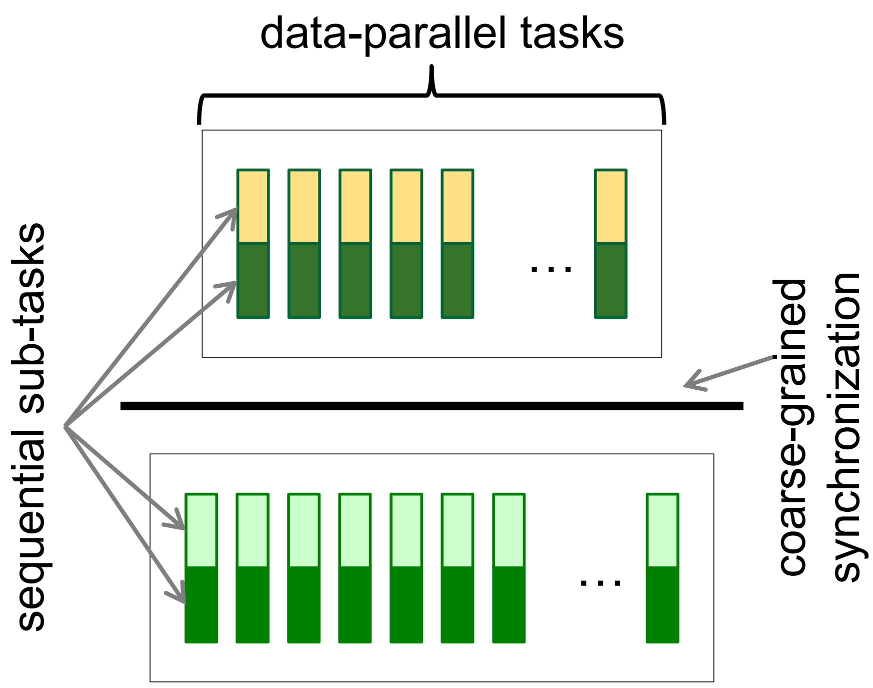


Program Structure

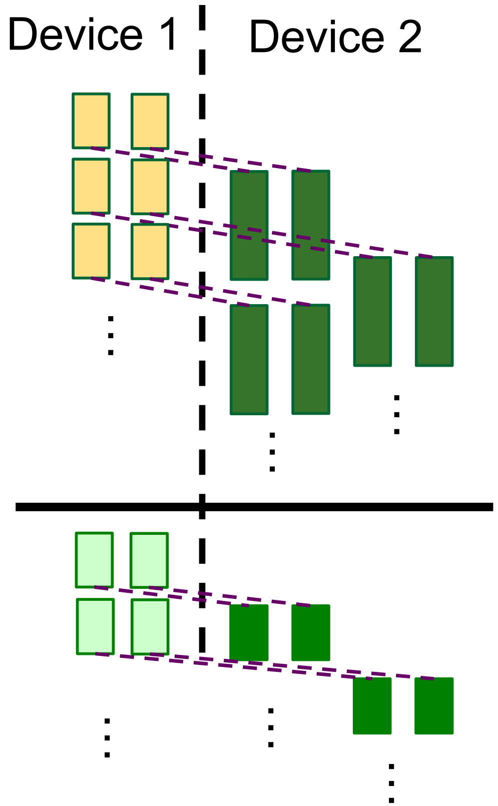


Coarse-grained Task Partitioning

Collaborative Patterns



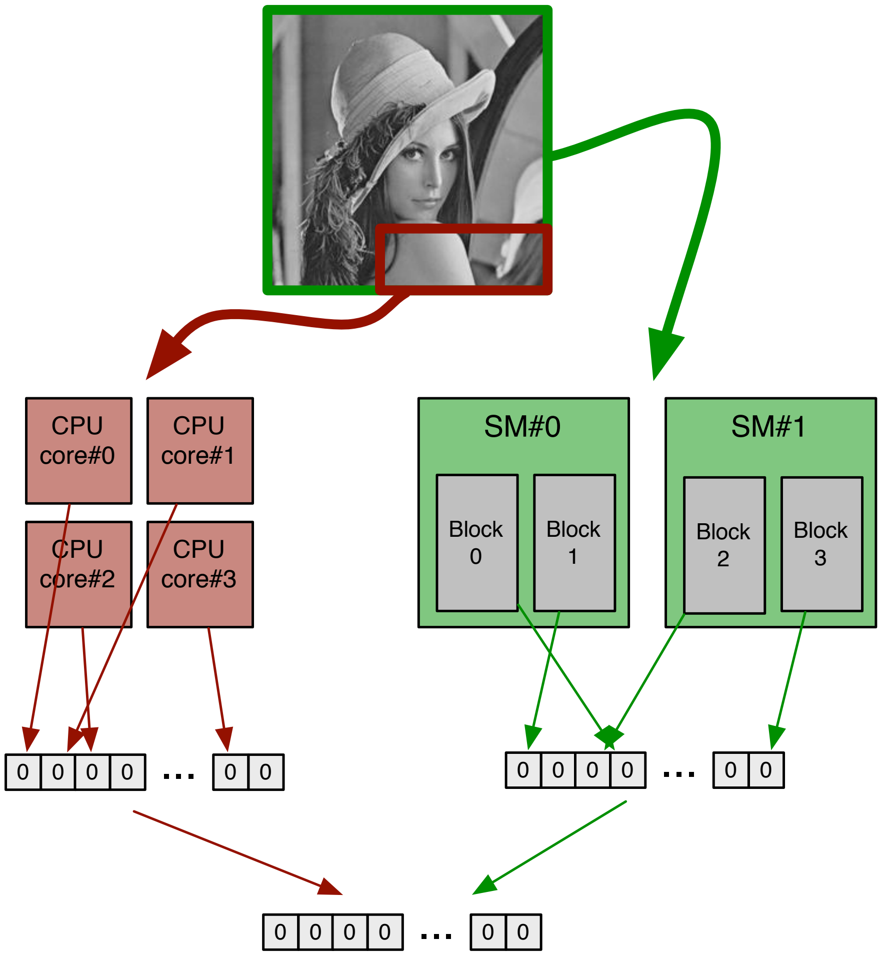
Program Structure



Fine-grained Task Partitioning

Histogram

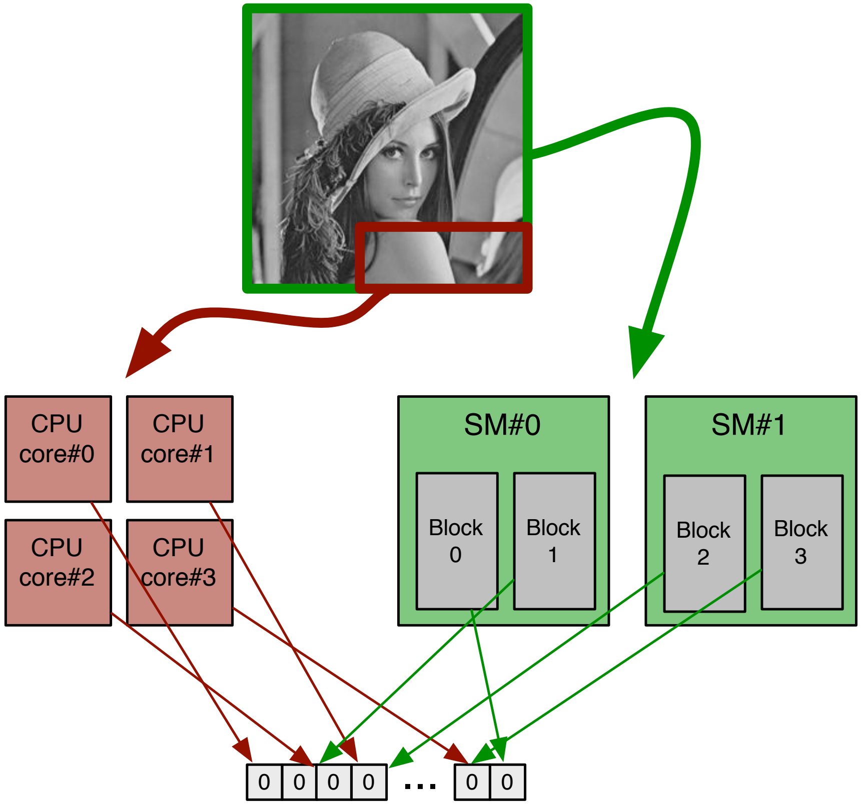
- Previous generations: separate CPU and GPU histograms are merged at the end




```
malloc(CPU image);
cudaMalloc(GPU image);
cudaMemcpy(GPU image, CPU image, ...,
           Host to Device);
malloc(CPU histogram);
memset(CPU histogram, 0);
cudaMalloc(GPU histogram);
cudaMemset(GPU histogram, 0);

// Launch CPU threads
// Launch GPU kernel

cudaMemcpy(GPU histogram, DeviceToHost);

// Launch CPU threads for merging
```

Histogram

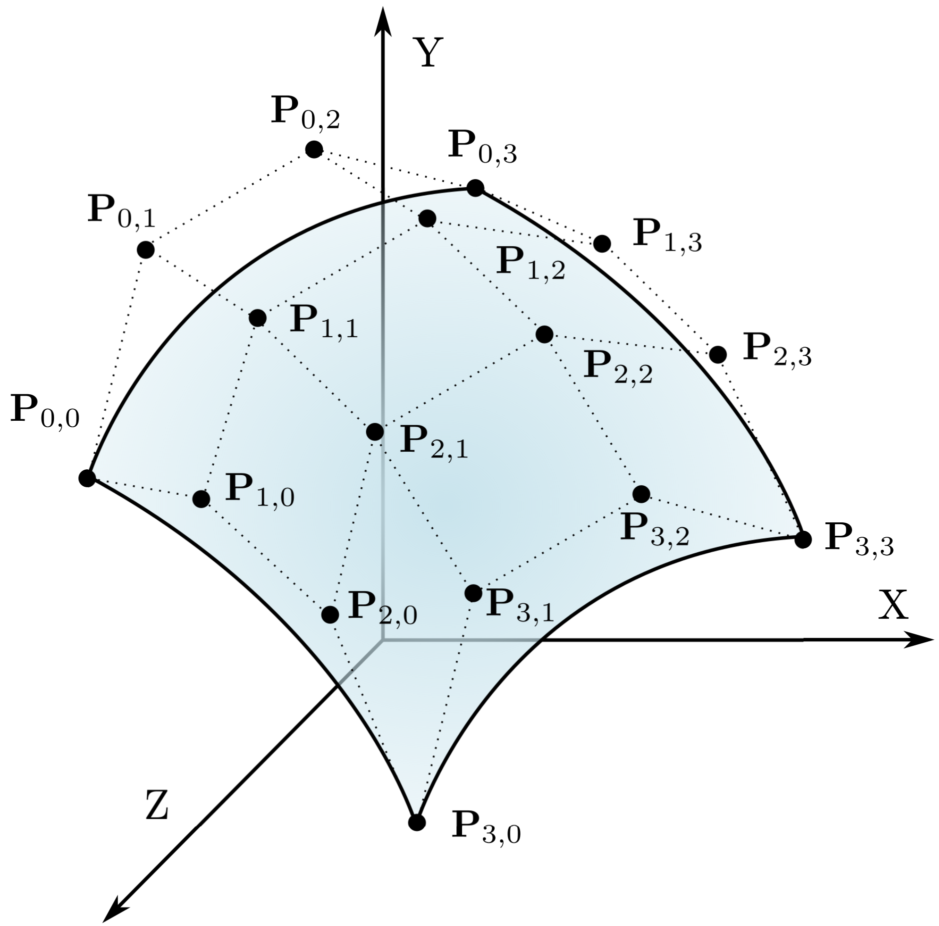
- System-wide atomic operations: one single histogram




```
cudaMallocManaged(Histogram);
cudaMemset(Histogram, 0);

// Launch CPU threads
// Launch GPU kernel (atomicAdd_system)
```

Bézier Surfaces

- Bézier surface: 4x4 net of control points



Bézier Surfaces

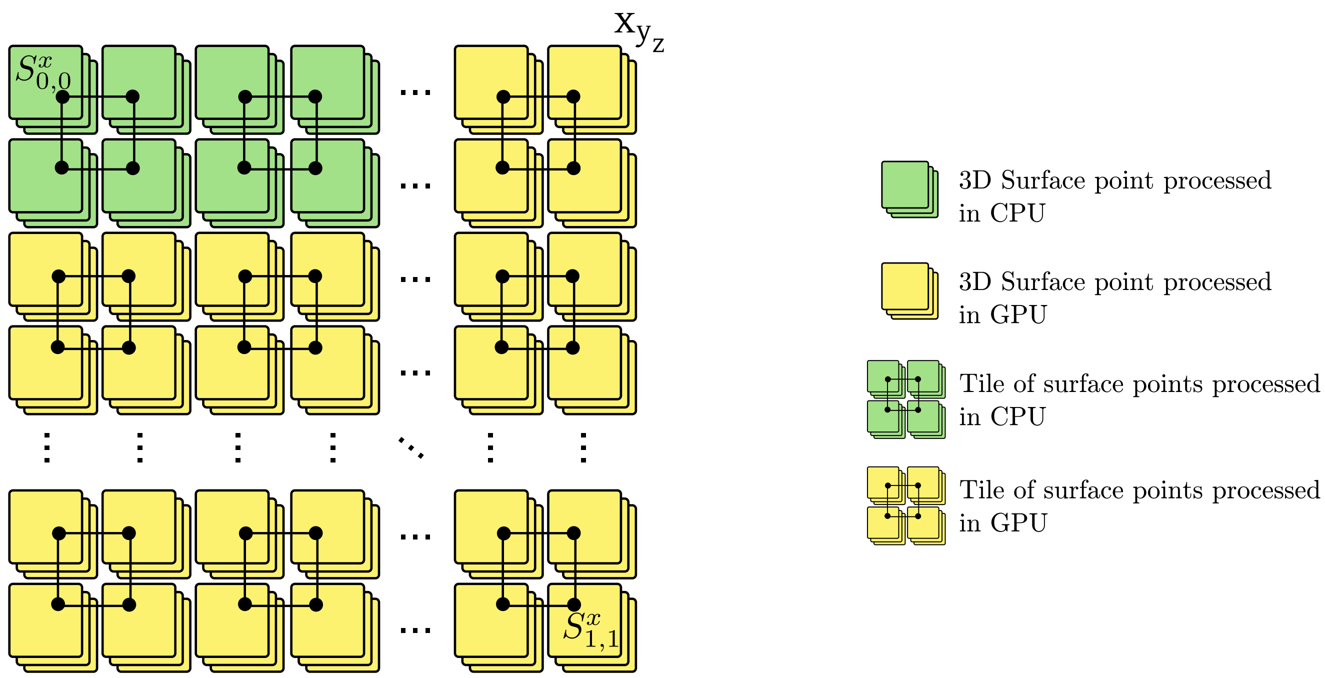
- Parametric non-rational formulation
  - Bernstein polynomials
  - Bi-cubic surface *m* = *n* = 3

$$\mathbf{S}(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{P}_{i,j} B_{i,m}(u) B_{j,n}(v), \qquad (1)$$

$$B_{i,m}(u) = \binom{m}{i} (1-u)^{(m-i)} u^{i}, \qquad (2)$$

Bézier Surfaces

- Collaborative implementation
  - Tiles calculated by GPU blocks or CPU threads
  - Static distribution



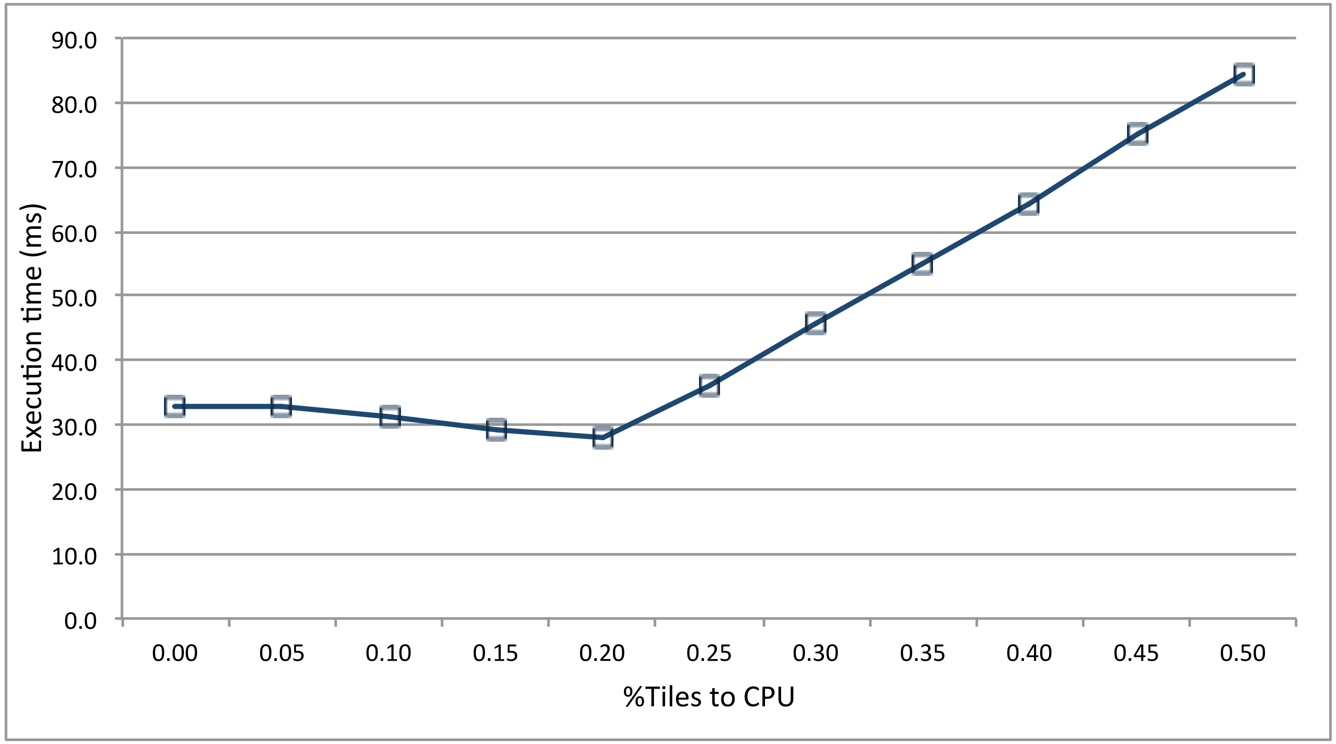
Bézier Surfaces

- Without Unified Memory

```
// Allocate control points
malloc(control_points, ...);
generate_cp(control_points);
cudaMalloc(d_control_points, ...);
cudaMemcpy(d_control_points, control_points, ..., HostToDevice); // Copy to device memory

// Allocate surface
malloc(surface, ...);
cudaMalloc(d_surface, ...);

// Launch CPU threads
std::thread main_thread (run_cpu_threads, control_points, surface, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_surface, d_control_points, ...);

// Synchronize
main_thread.join();
cudaDeviceSynchronize();

// Copy gpu part of surface to host memory
cudaMemcpy(&surface[end_of_cpu_part], d_surface, ..., DeviceToHost);
```

Bézier Surfaces

- Execution results
  - Bezier surface: 300x300, 4x4 control points
  - %Tiles to CPU
  - NVIDIA Jetson TX1 (4 ARMv8 + 2 SMX): 17% speedup wrt GPU only



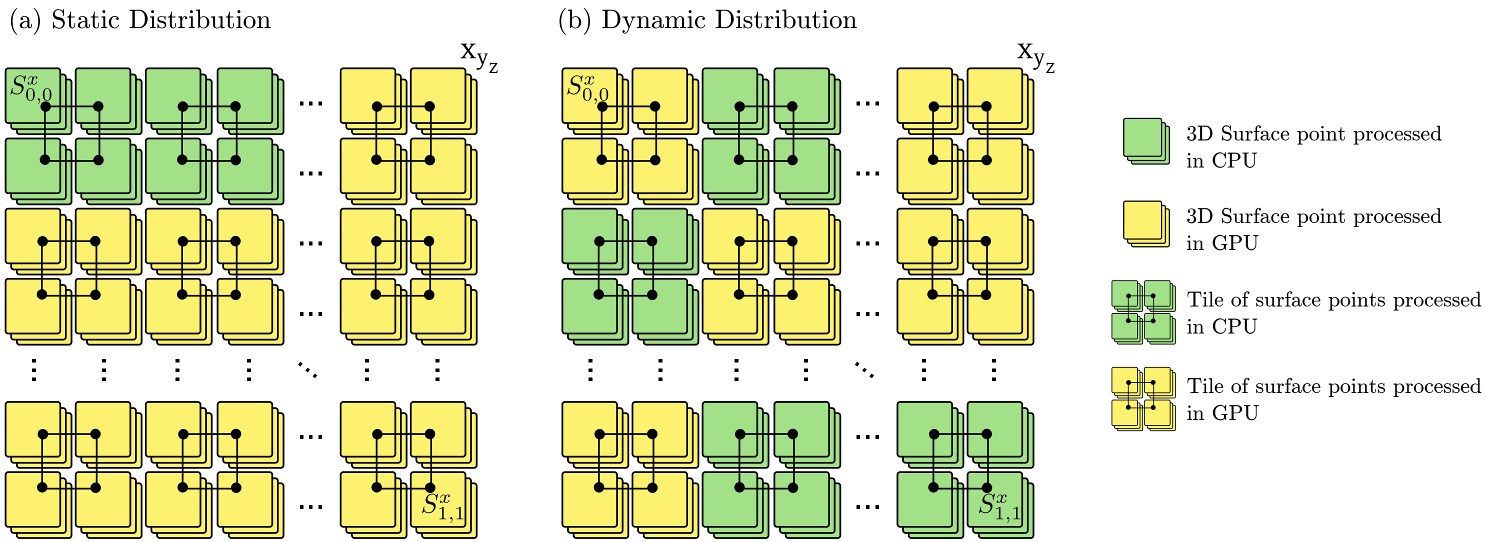
Bézier Surfaces

- With Unified Memory (Pascal/Volta)

```
// Allocate control points
malloc(control_points, ...);
generate_cp(control_points);
cudaMalloc(d_control_points, ...);
cudaMemcpy(d_control_points, control_points, ..., HostToDevice); // Copy to device memory

// Allocate surface
cudaMallocManaged(surface, ...);

// Launch CPU threads
std::thread main_thread (run_cpu_threads, control_points, surface, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (surface, d_control_points, ...);

// Synchronize
main_thread.join();
cudaDeviceSynchronize();
```

Bézier Surfaces

- Static vs. dynamic implementation

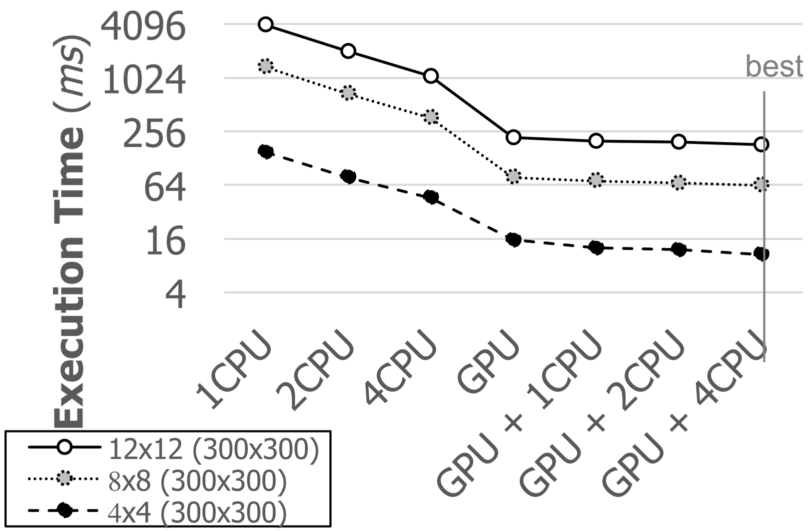


- Pascal/Volta Unified Memory: system-wide atomic operations

```
while(true){
    if(threadIdx.x == 0)
        my_tile = atomicAdd_system(tile_num, 1);  // my_tile in shared memory; tile_num in UM

    __syncthreads();  // Synchronization

    if(my_tile >= number_of_tiles) break;  // Break when all tiles processed
...
}
```

Benefits of Collaboration

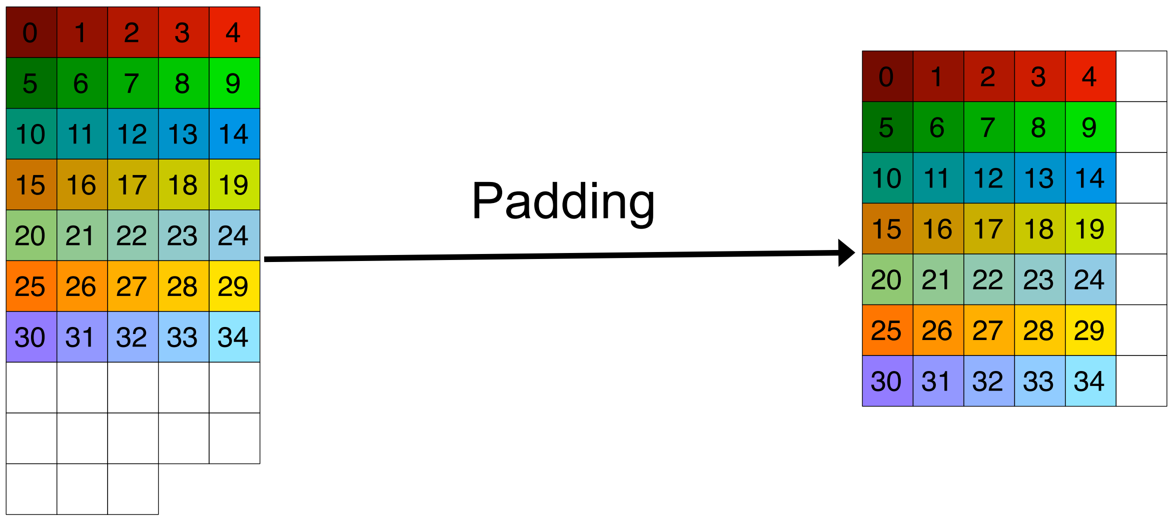
- Data partitioning improves performance
  - AMD Kaveri (4 CPU cores + 8 GPU CUs)



Bézier Surfaces
(up to 47% improvement over GPU only)

Padding

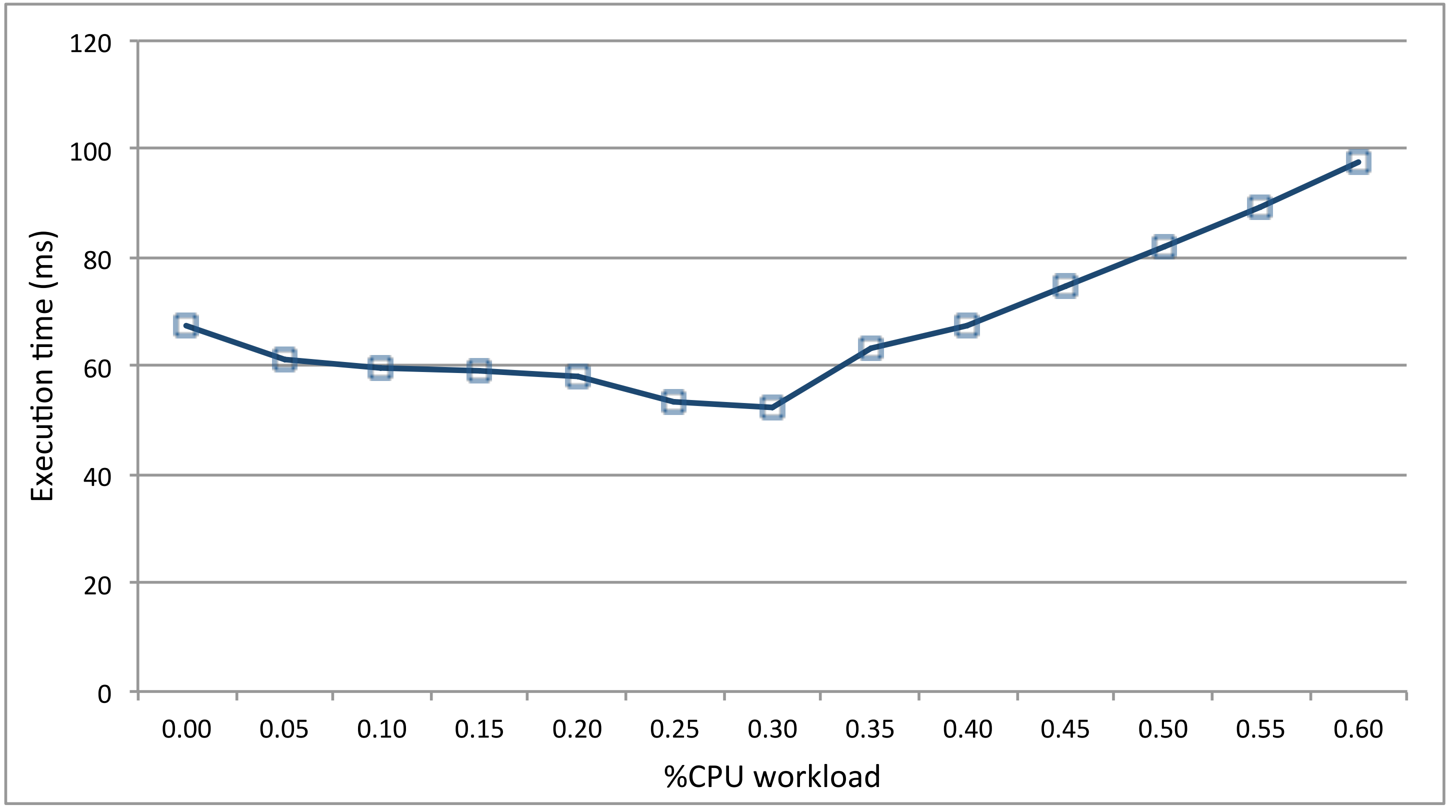
- Matrix padding
  - Memory alignment
  - Transposition of near-square matrices



- Traditionally, it can only be performed out-of-place

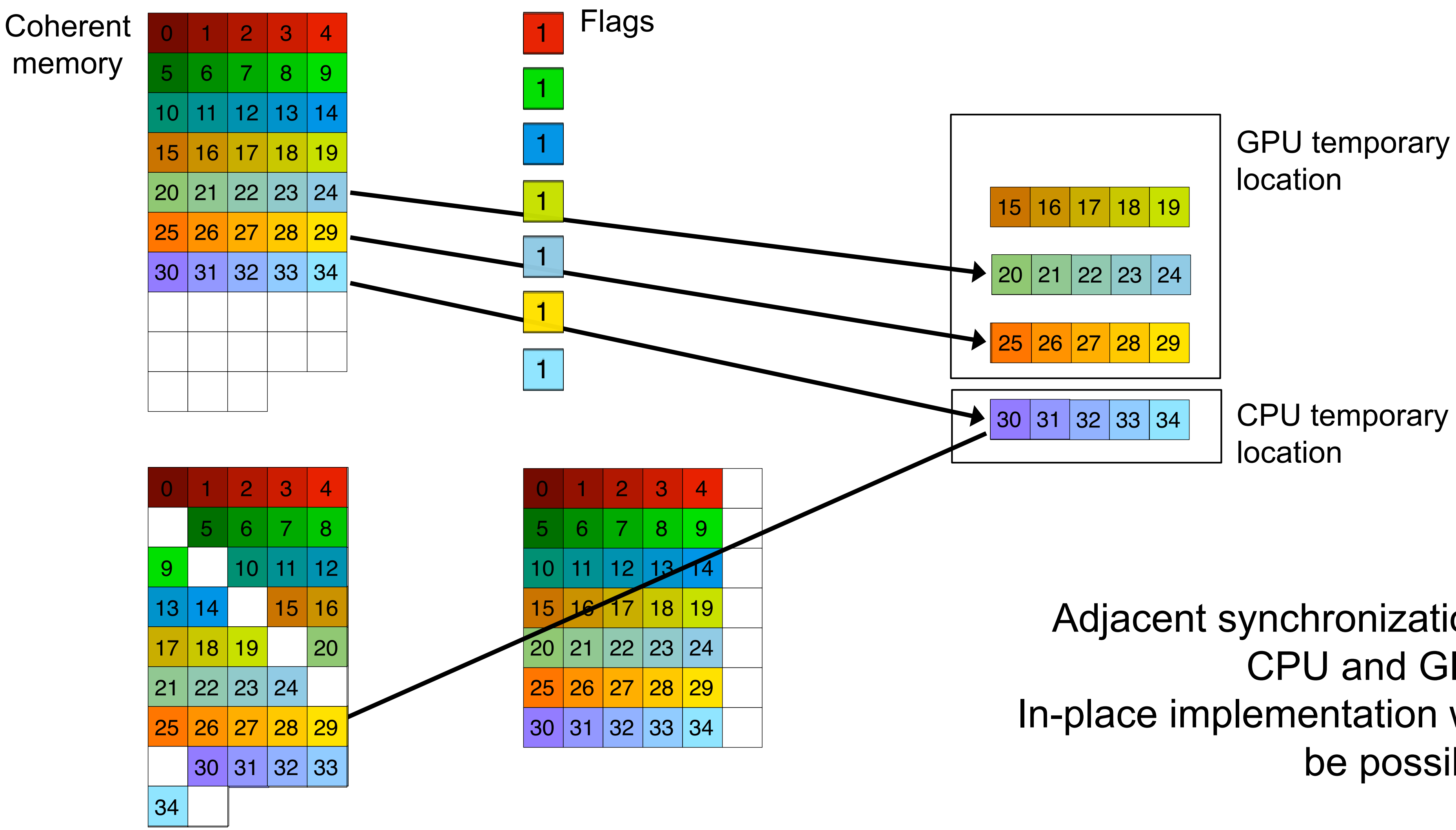
Padding

- Execution results
  - Matrix size: 4000x4000, padding = 1
  - NVIDIA Jetson TX1 (4 ARMv8 + 2 SMX): 29% speedup wrt GPU only



In-Place Padding

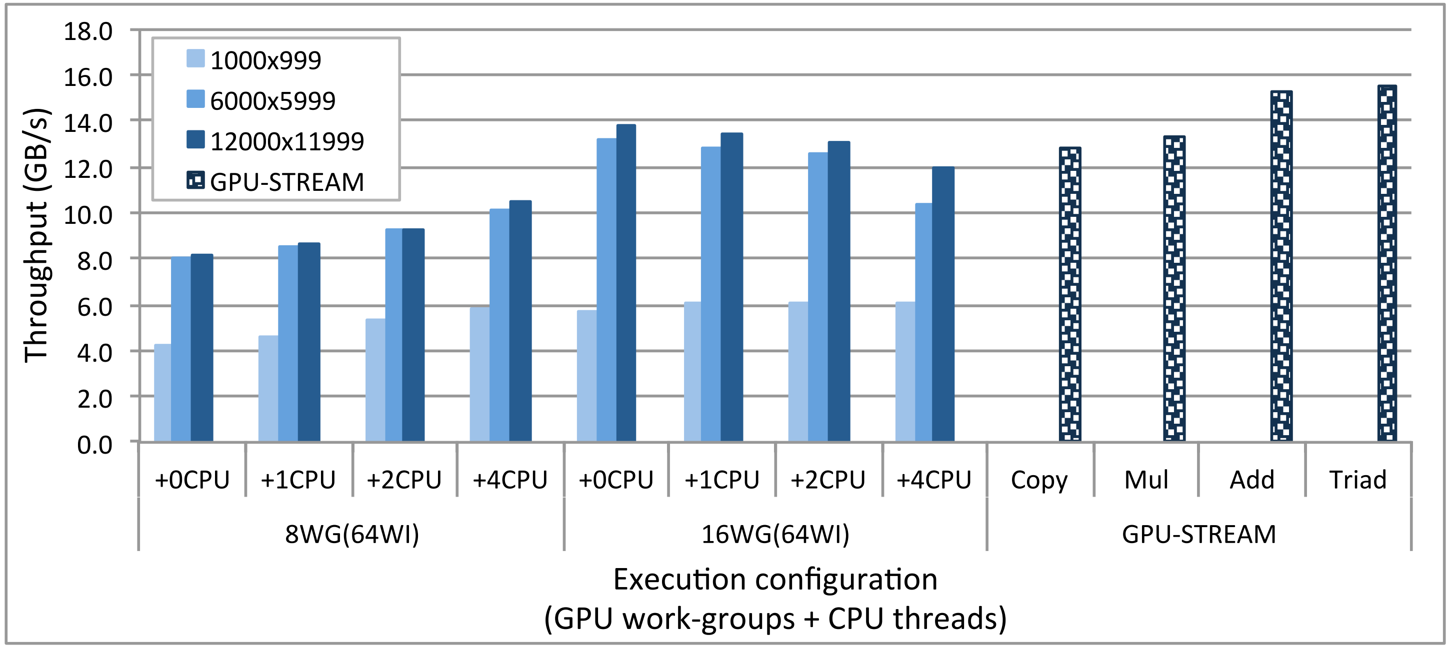
- Pascal/Volta Unified Memory



Adjacent synchronization: CPU and GPU
In-place implementation will be possible

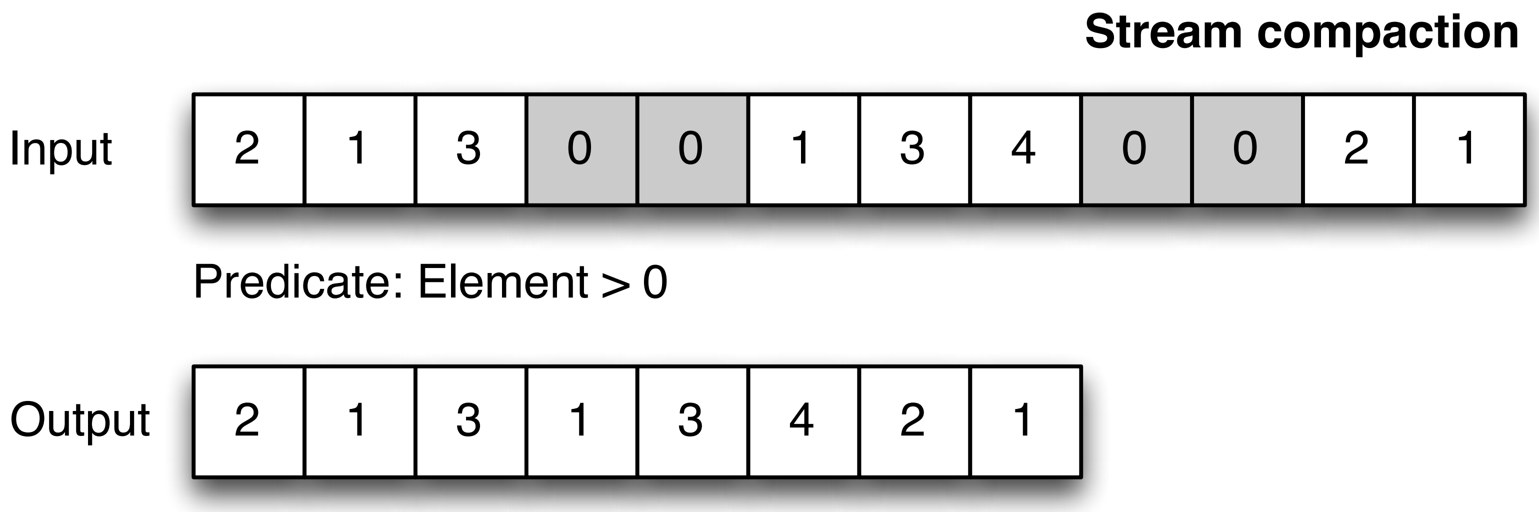
Benefits of Collaboration

- Optimal number of devices is not always max
  - AMD Kaveri (4 CPU cores + 8 GPU CUs)



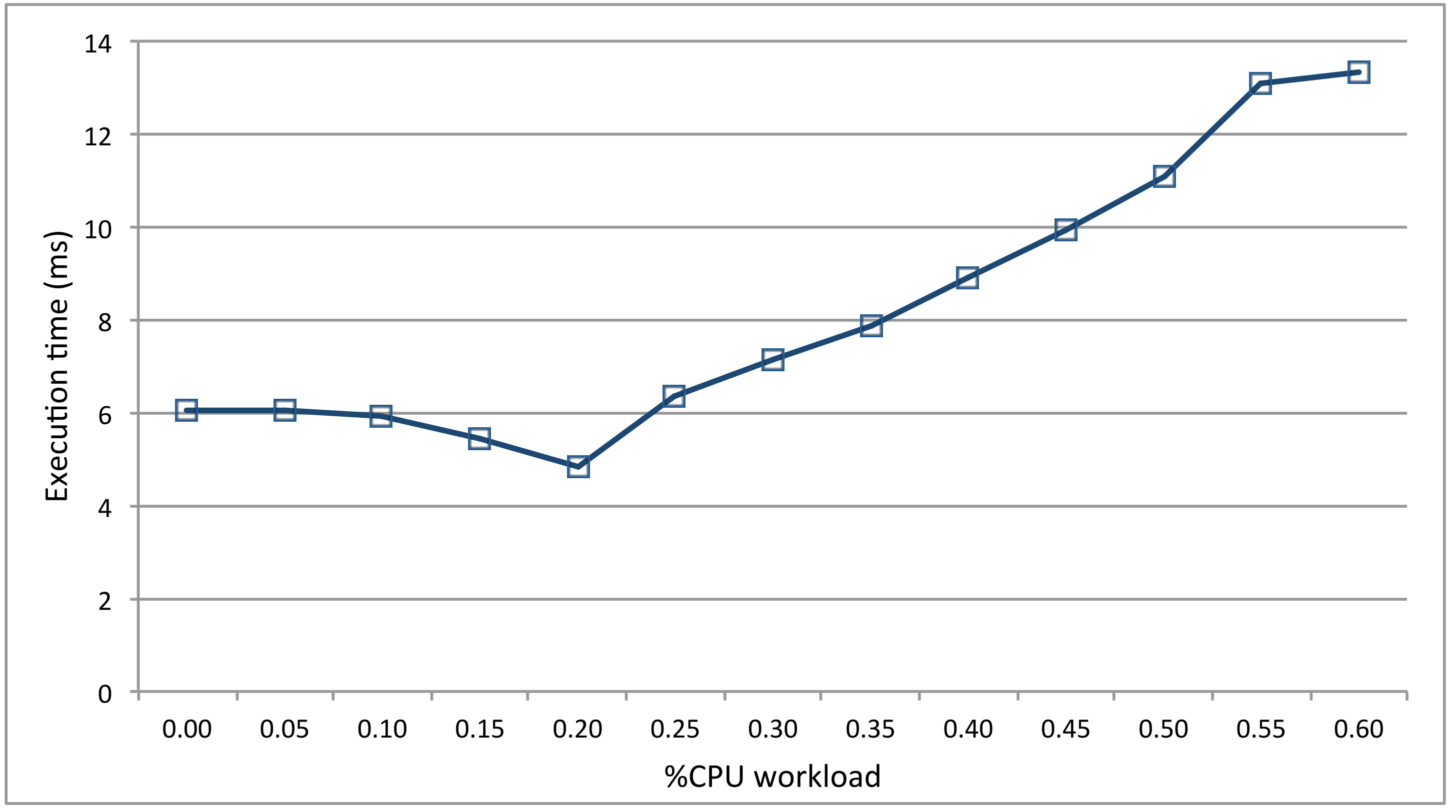
Stream Compaction

- Stream compaction
  - Saving memory storage in sparse data
  - Similar to padding, but local reduction result (non-zero element count) is propagated



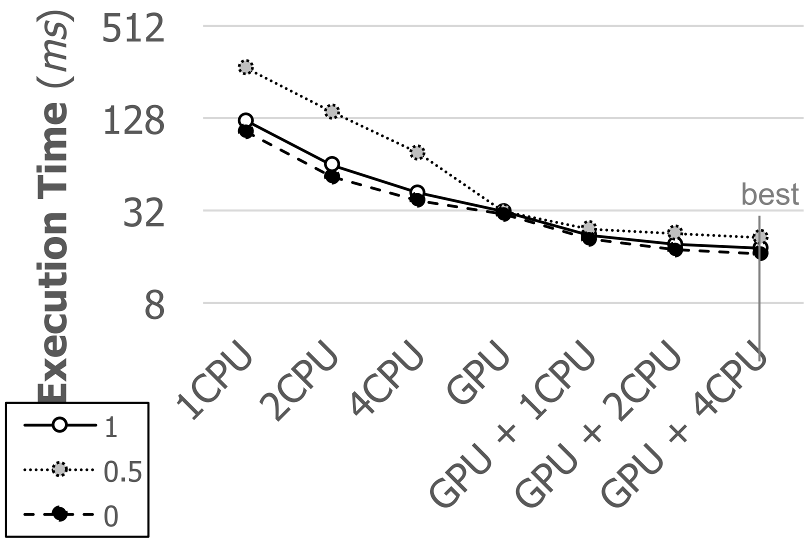
Stream Compaction

- Execution results
  - Array size: 2 MB, Filtered items = 50%
  - NVIDIA Jetson TX1 (4 ARMv8 + 2 SMX): 25% speedup wrt GPU only



Benefits of Collaboration

- Data partitioning improves performance
  - AMD Kaveri (4 CPU cores + 8 GPU CUs)



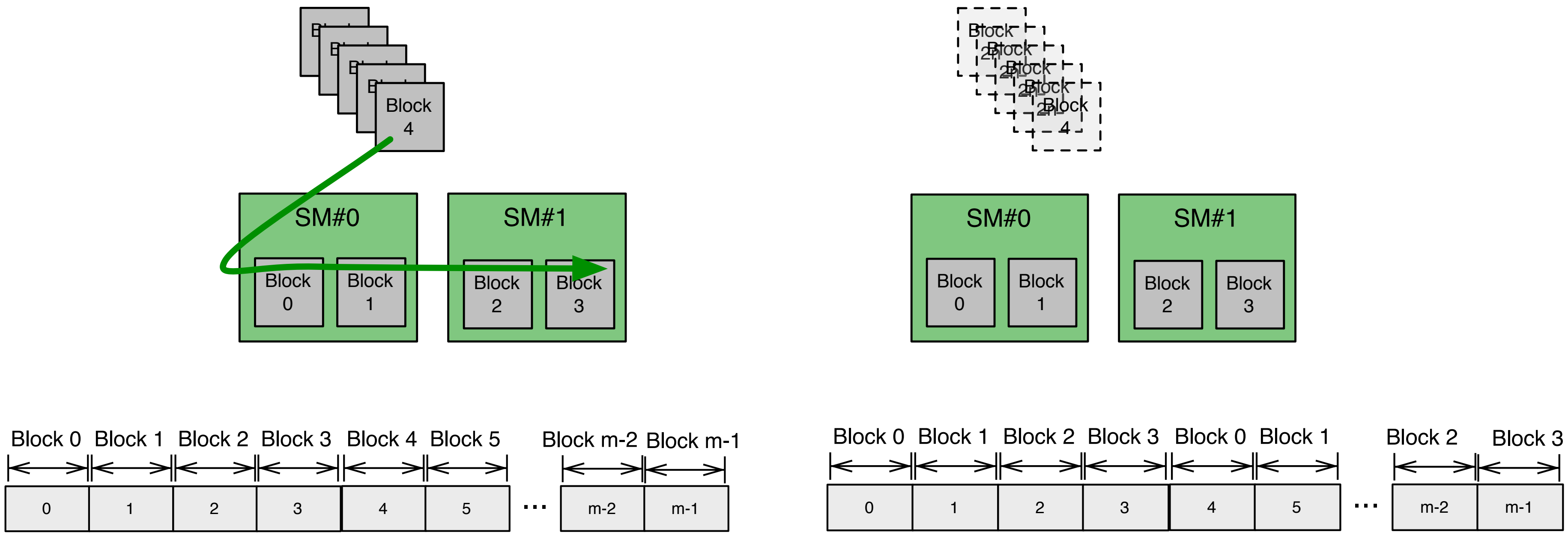
Stream Compaction
(up to 82% improvement over GPU only)

Breadth-First Search

- Small-sized and big-sized frontiers
  - Top-down approach
  - Kernel 1 and Kernel 2
- Atomic-based block synchronization
  - Avoids kernel re-launch
- Very small frontiers
  - Underutilize GPU resources
- Collaborative implementation

Atomic-Based Block Synchronization

- Combine Kernel 1 and Kernel 2
- We can avoid kernel re-launch
- We need to use persistent thread blocks
  - Kernel 2 launches (frontier_size / block_size) blocks
  - Persistent blocks: up to (number_SMs x max_blocks_SM)



Atomic-Based Block Synchronization

- Code (simplified)

```
// GPU kernel
const int gtid = blockIdx.x * blockDim.x + threadIdx.x;

while(frontier_size != 0){

    for(node = gtid; node < frontier_size; node += blockDim.x*gridDim.x){

      // Visit neighbors
      // Enqueue in output queue if needed (global or local queue)

    }

    // Update frontier_size

    // Global synchronization
}
```

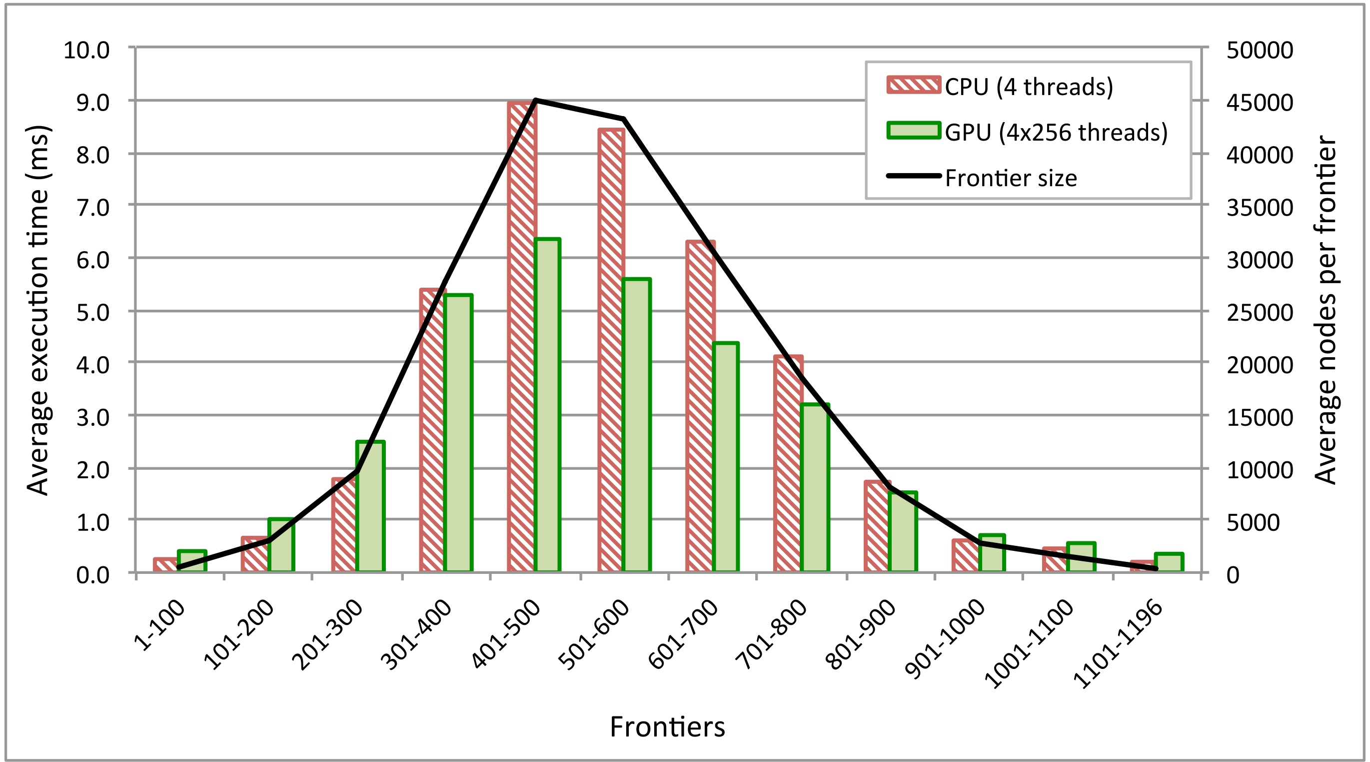
Atomic-Based Block Synchronization

- Global synchronization (simplified)
  - At the end of each iteration

```
const int tid = threadIdx.x;
const int gtid = blockIdx.x * blockDim.x + threadIdx.x;
atomicExch(ptr_threads_run, 0);
atomicExch(ptr_threads_end, 0);
int frontier = 0;
 ...

frontier++;

if(tid == 0){
    atomicAdd(ptr_threads_end, 1);  // Thread block finishes iteration
}

if(gtid == 0){
    while(atomicAdd(ptr_threads_end, 0) != gridDim.x){;}  // Wait until all blocks finish

    atomicExch(ptr_threads_end, 0);  // Reset
    atomicAdd(ptr_threads_run, 1);  // Count iteration
}

if(tid == 0 && gtid != 0){
    while(atomicAdd(ptr_threads_run, 0) < frontier){;}  // Wait until ptr_threads_run is updated
}

__syncthreads();  // Rest of threads wait here

...
```

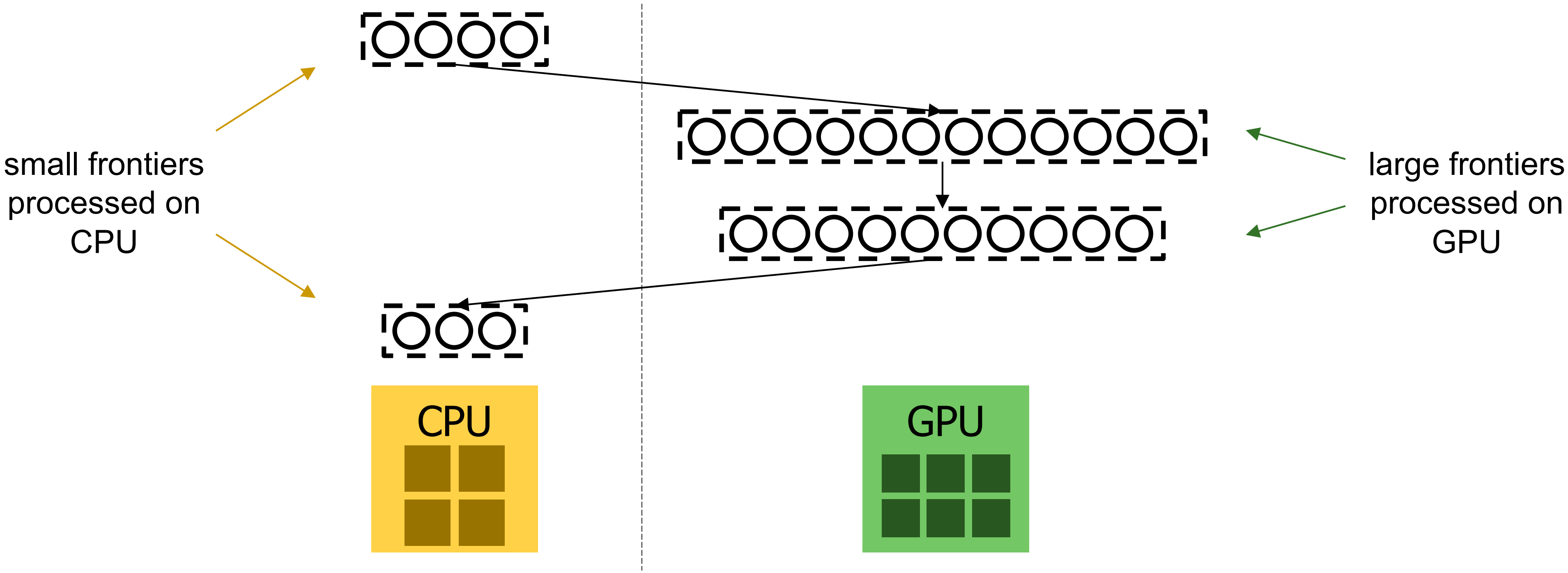
Collaborative Implementation

- Motivation
  - Small-sized frontiers underutilize GPU resources
    - NVIDIA Jetson TX1 (4 ARMv8 CPUs + 2 SMXs)
    - New York City roads



Collaborative Implementation

- Choose the most appropriate device



Collaborative Implementation

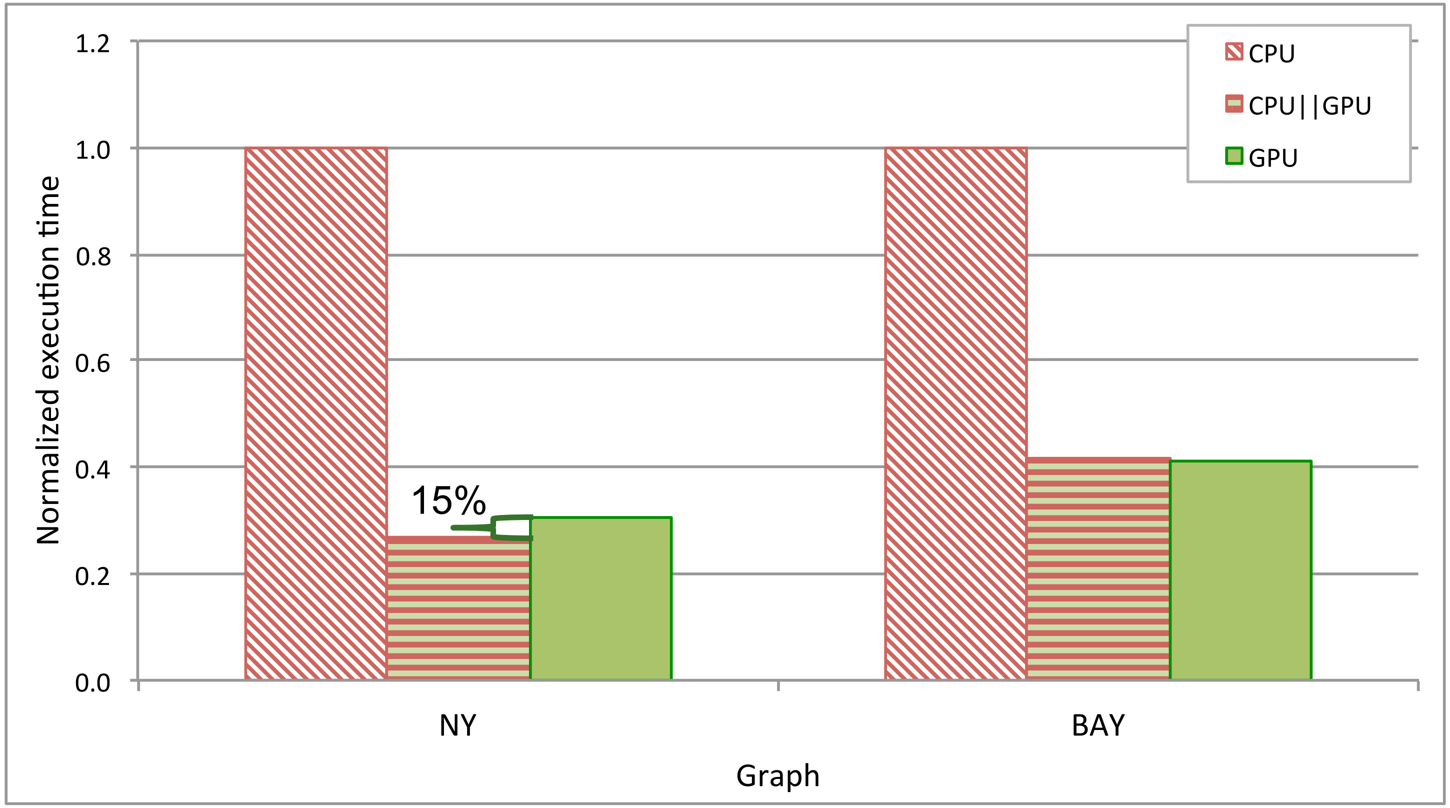
- Choose CPU or GPU depending on frontier size

```
// Host code
while(frontier_size != 0){

    if(frontier_size < LIMIT){

        // Launch CPU threads

    }
    else{

        // Launch GPU kernel

    }

}
```

- CPU threads or GPU kernel keep running while the condition is satisfied

Collaborative Implementation

- Execution results



Collaborative Implementation

- Without Unified Memory
  - Explicit memory copies

```
// Host code
while(frontier_size != 0){

    if(frontier_size < LIMIT){

        // Launch CPU threads

    }
    else{

        // Copy from host to device (queues and synchronization variables)

        // Launch GPU kernel

        // Copy from device to host (queues and synchronization variables)

    }

}
```

Collaborative Implementation

- Unified Memory
  - `cudaMallocManaged();`
  - Easier programming
  - No explicit memory copies

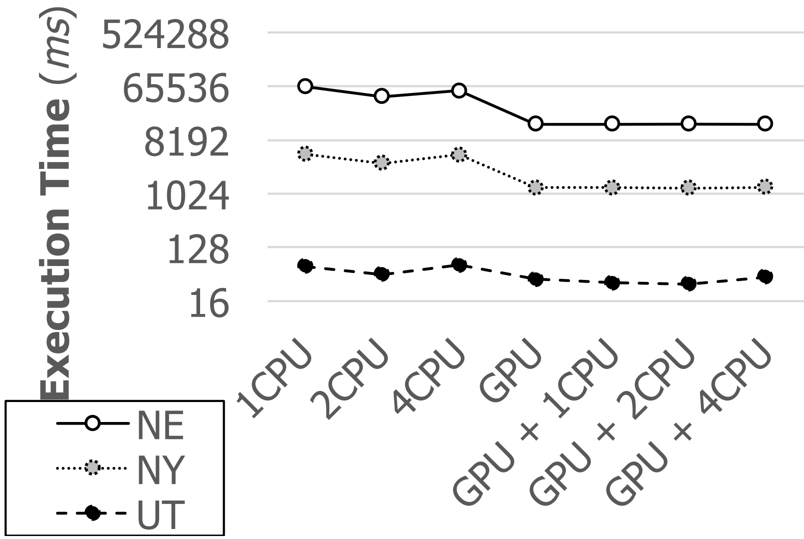
```
// Host code
while(frontier_size != 0){

    if(frontier_size < LIMIT){

        // Launch CPU threads

    }
    else{

        // Launch GPU kernel

        cudaDeviceSynchronize();

    }

}
```

Collaborative Implementation

- Pascal/Volta Unified Memory
  - CPU/GPU coherence
  - System-wide atomic operations
  - No need to re-launch kernel or CPU threads
  - Possibility of CPU and GPU working on the same frontier

Benefits of Collaboration

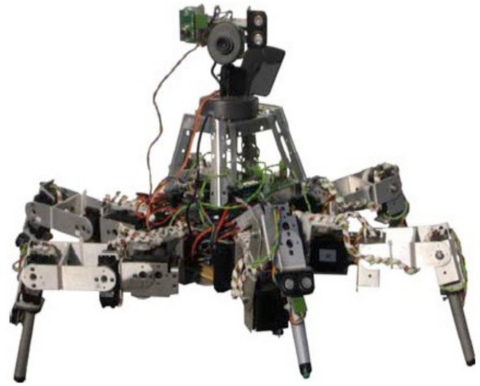
- SSSP performs more computation than BFS



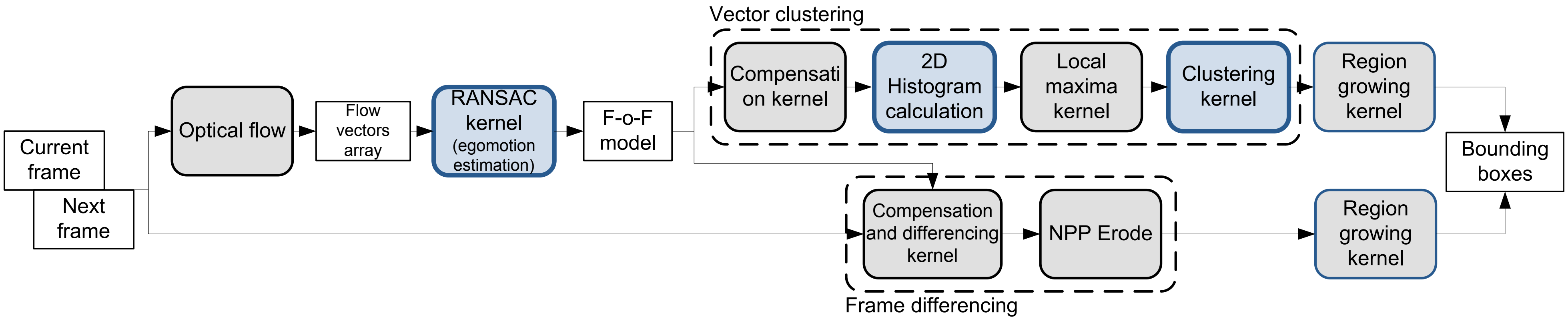
Single Source Shortest Path
(up to 22% improvement over GPU only)

Egomotion Compensation and Moving Objects Detection

- Hexapod robot OSCAR
  - Rescue scenarios
  - Strong egomotion on uneven terrains

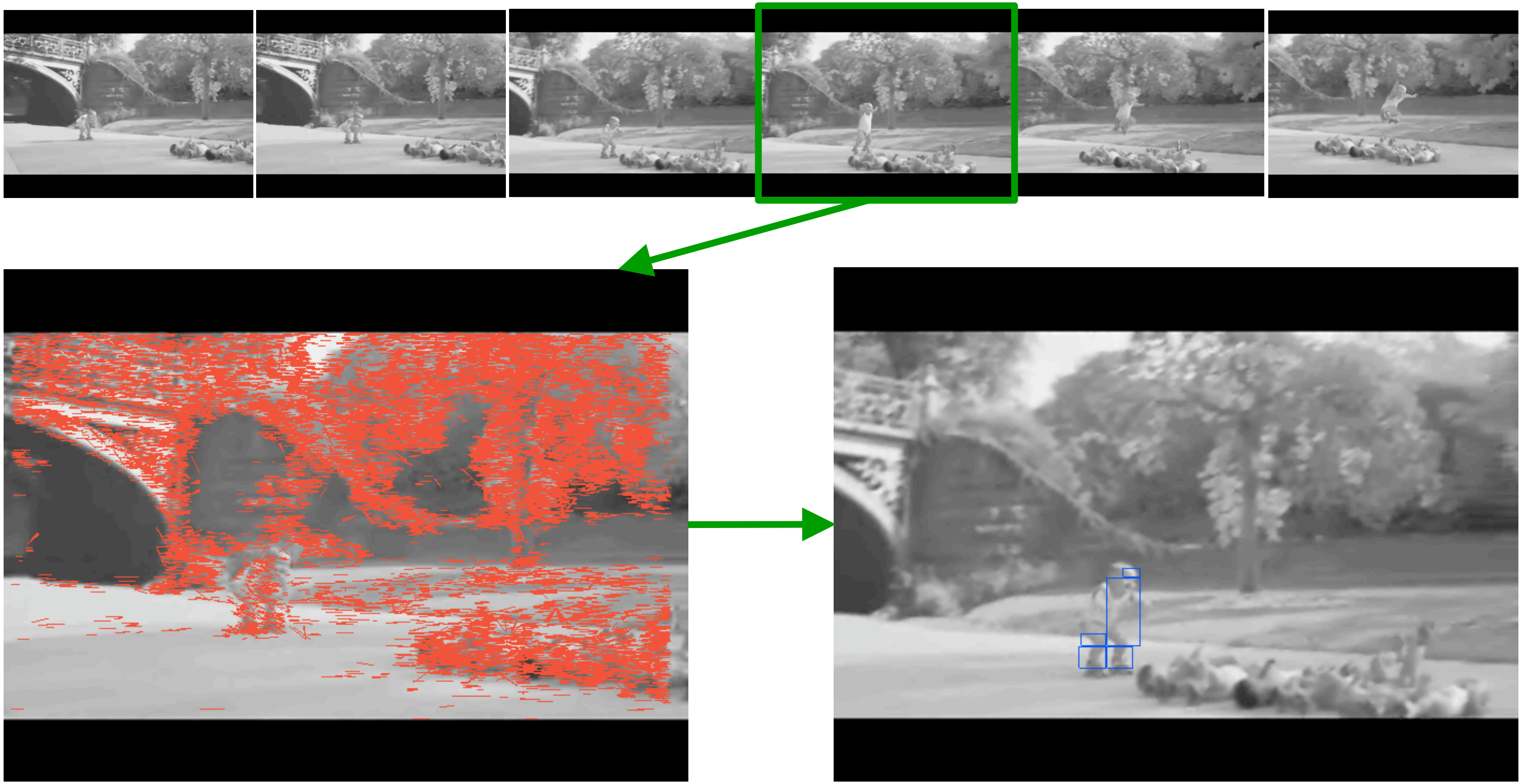


- Algorithm
  - Random Sample Consensus (RANSAC): F-o-F model



Egomotion Compensation and Moving Objects Detection

Fast moving object in strong egomotion scenario detected by vector clustering

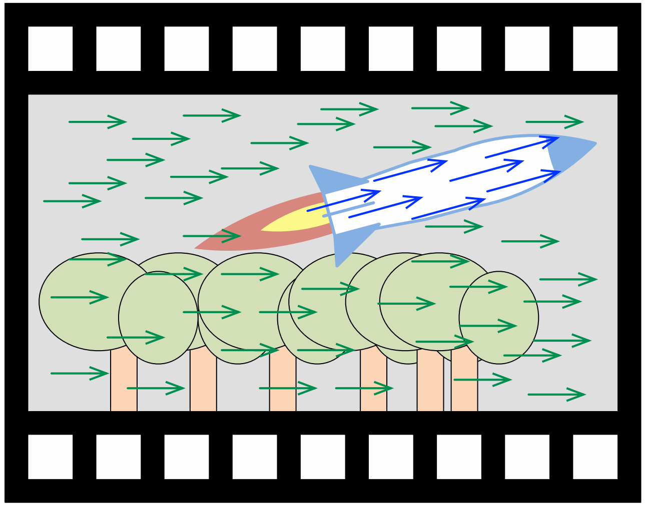


SISD and SIMD phases

- RANSAC (Fischler *et al.* 1981)

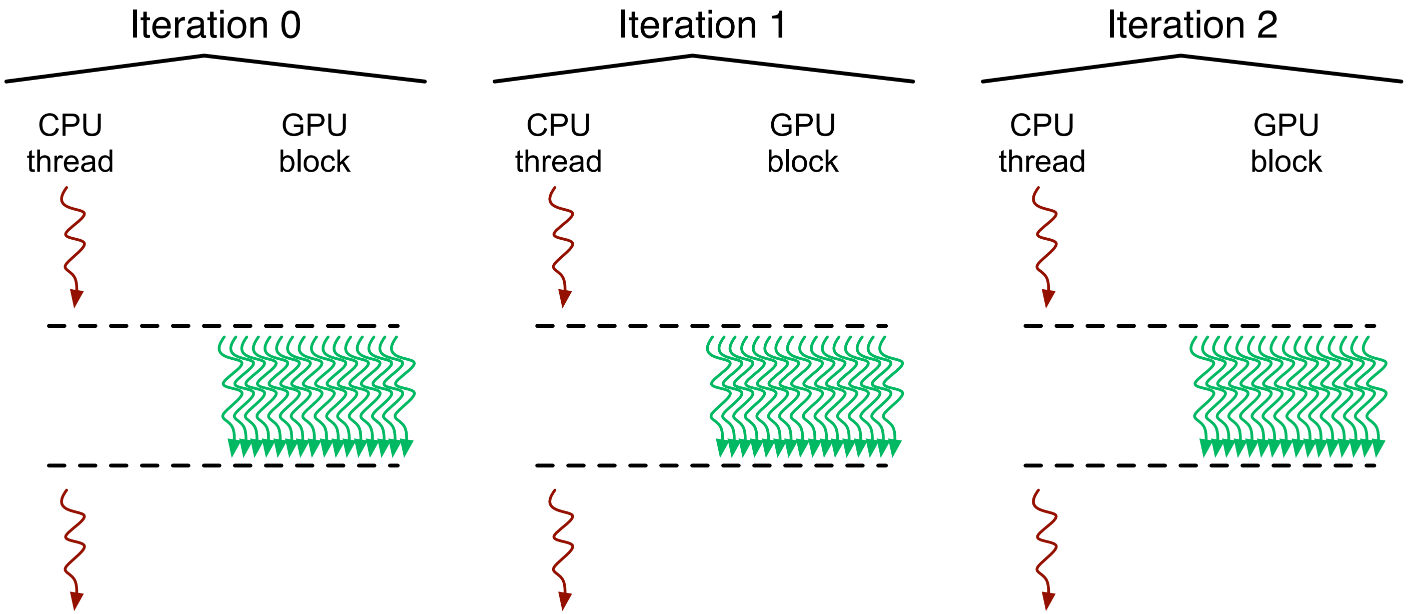
```
While (iteration < MAX_ITER){
     Fitting stage (Compute F-o-F model)           // SISD phase

     Evaluation stage (Count outliers)             // SIMD phase

     Comparison to best model                      // SISD phase

     Check if best model is good enough and iteration >= MIN_ITER     // SISD phase
}
```

  - Fitting stage picks two flow vectors randomly
  - Evaluation generates motion vectors from F-o-F model, and compares them to real flow vectors



Collaborative Implementation

- Randomly picked vectors: Iterations are independent
  - We assign one iteration to one CPU thread and one GPU block



Chai Benchmark Suite

- Collaboration patterns
  - 8 data partitioning benchmarks
  - 3 coarse-grain task partitioning benchmarks
  - 3 fine-grain task partitioning benchmarks

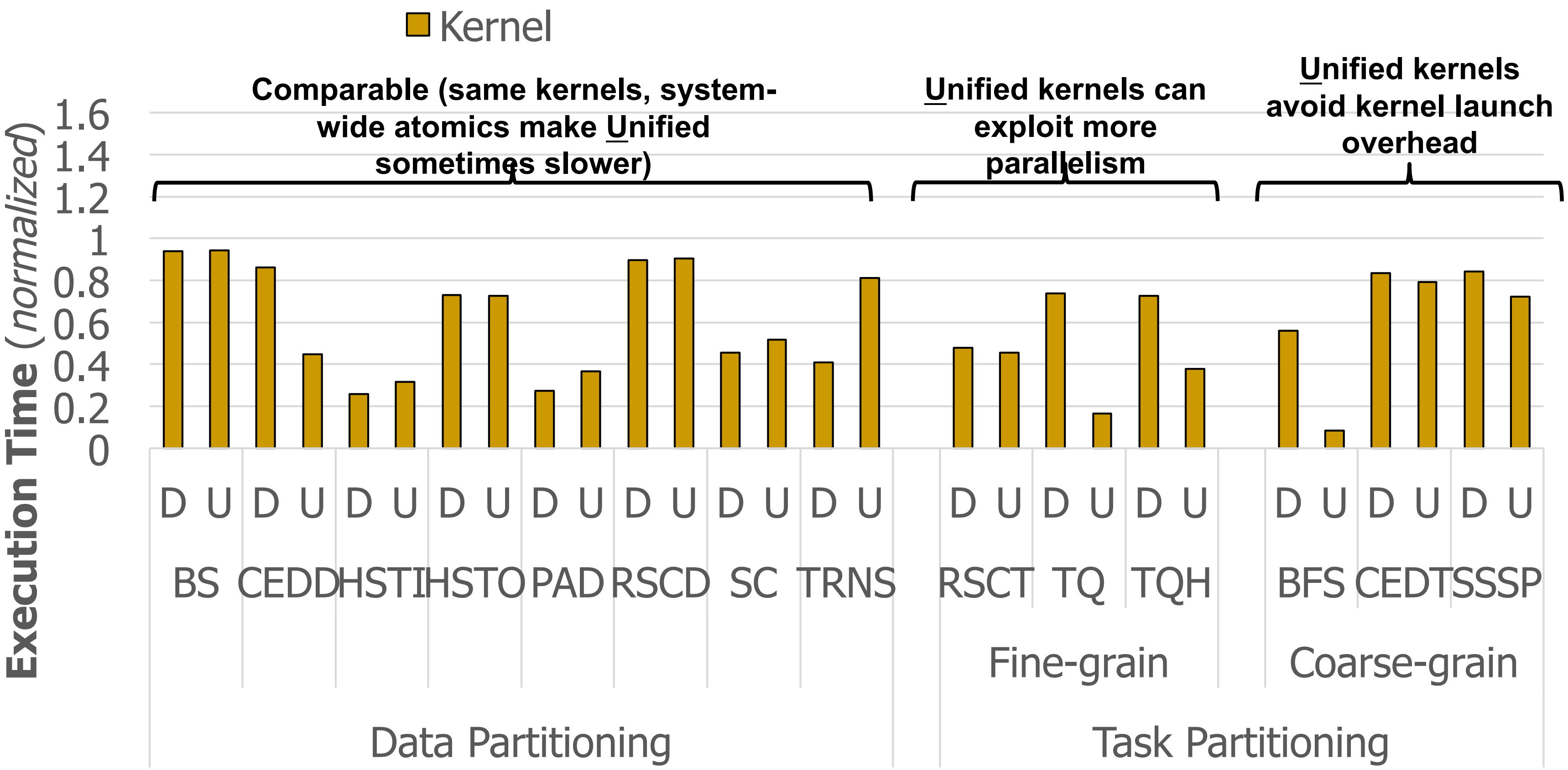
https://chai-benchmarks.github.io

CHAI

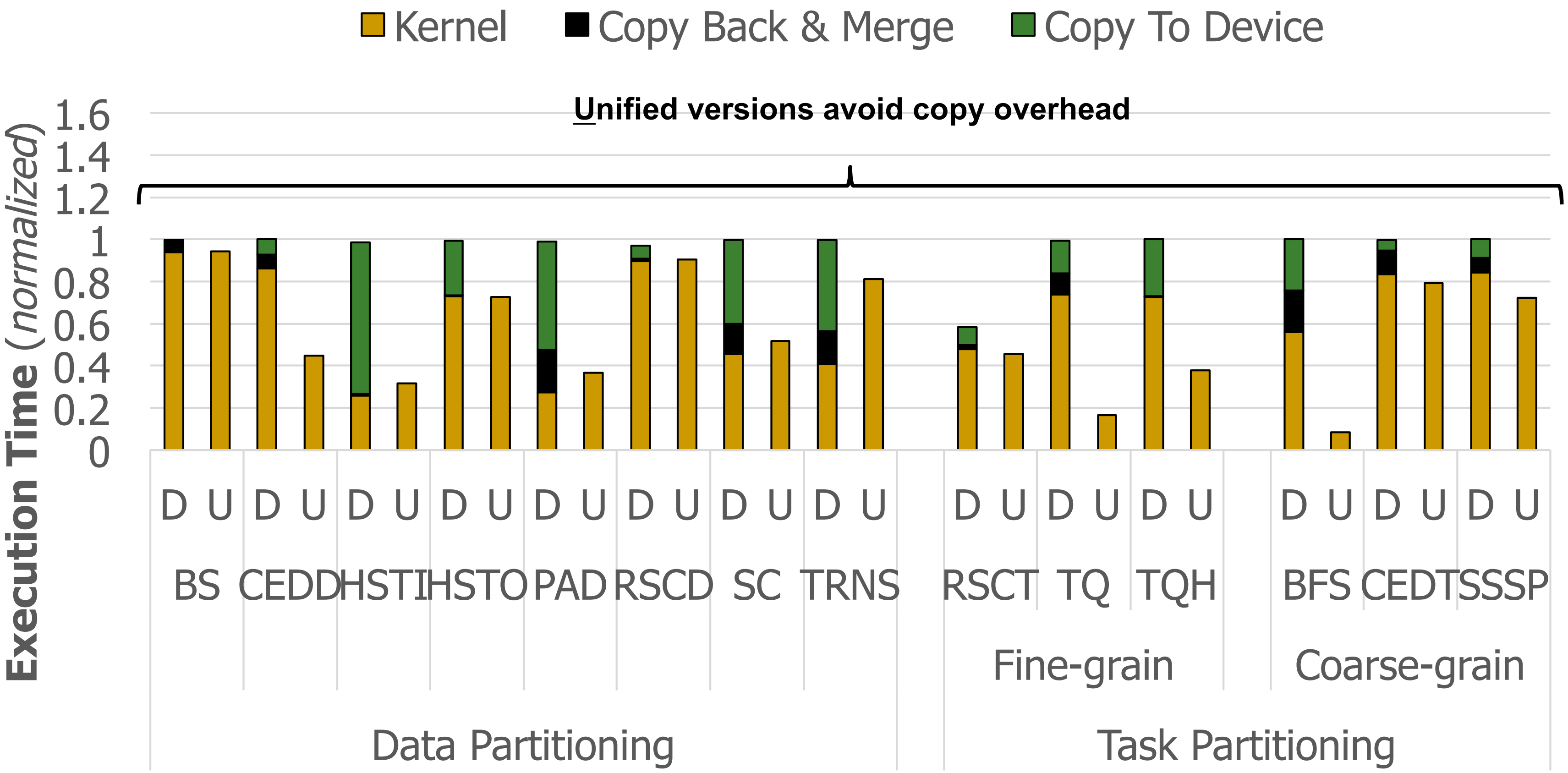
Chai Benchmark Suite

Collaboration Pattern		Short Name	Benchmark
Data Partitioning		BS	Bézier Surface
		CEDD	Canny Edge Detection
		HSTI	Image Histogram (Input Partitioning)
		HSTO	Image Histogram (Output Partitioning)
		PAD	Padding
		RSCD	Random Sample Consensus
		SC	Stream Compaction
		TRNS	In-place Transposition
Task Partitioning	Fine-grain	RSCT	Random Sample Consensus
		TQ	Task Queue System (Synthetic)
		TQH	Task Queue System (Histogram)
	Coarse-grain	BFS	Breadth-First Search
		CEDT	Canny Edge Detection
		SSSP	Single-Source Shortest Path

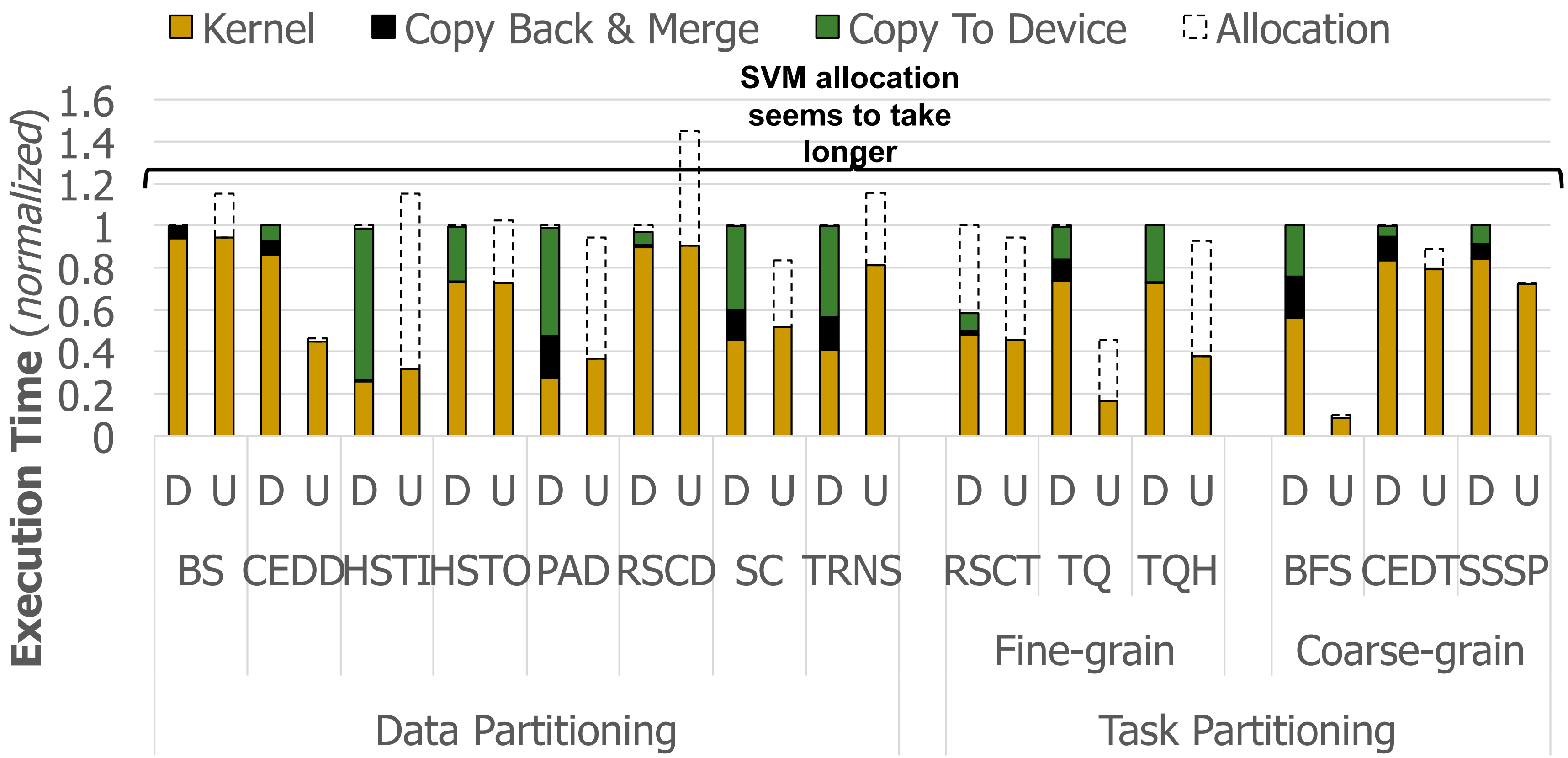
Benefits of Unified Memory



Benefits of Unified Memory

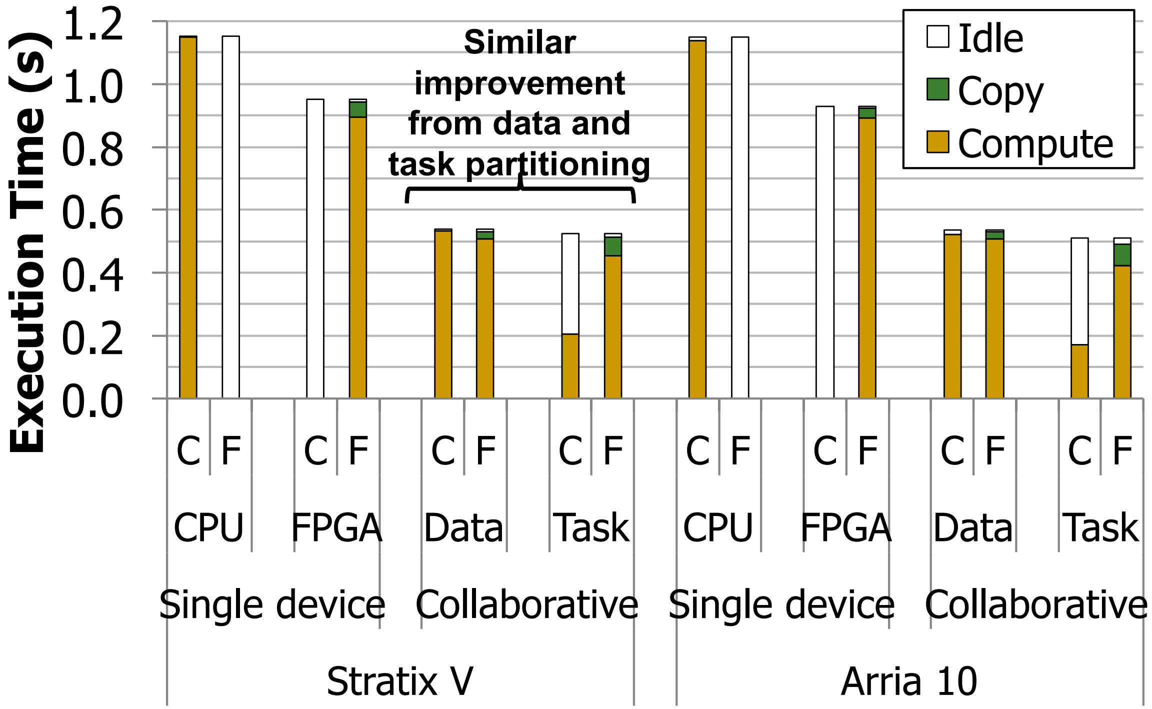


Benefits of Unified Memory



Benefits of Collaboration on FPGA

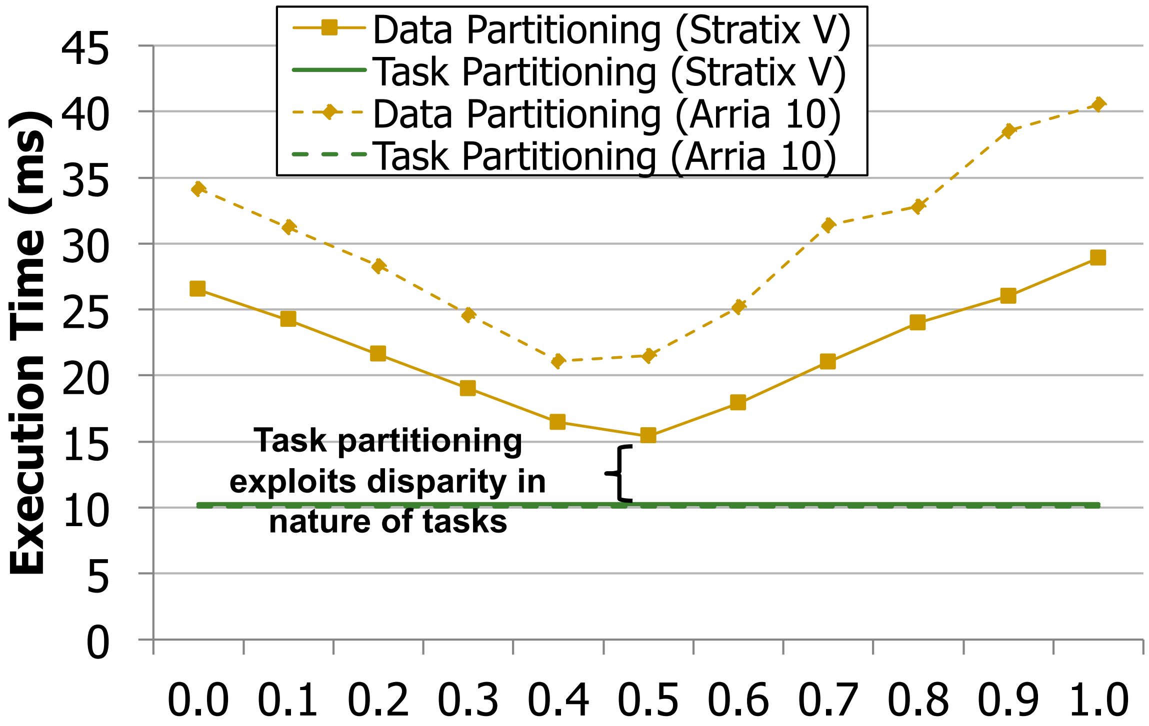
<u>Case Study:</u>
Canny Edge Detection



<u>Source:</u> Collaborative Computing for Heterogeneous Integrated Systems. *ICPE'17 Vision Track.*

Benefits of Collaboration on FPGA

Case Study:
Random
Sample
Consensus



Source: Collaborative Computing for Heterogeneous Integrated Systems. *ICPE'17 Vision Track.*

Conclusions

- Possibility of having CPU threads and GPU blocks collaborating on the same workload
- Or having the most appropriate cores for each workload
- Easier programming with Unified Memory or Shared Virtual Memory
- System-wide atomic operations in NVIDIA Pascal/Volta and HSA
    - Fine-grain collaboration

Computer Architecture

Lecture 14: New Programming Features in Heterogeneous Systems

Juan Gómez Luna
ETH Zürich

Fall 2017

8 November 2017