

# P&S Heterogeneous Systems

## Collaborative Computing

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2021

6 January 2022

In Our Previous Lecture...

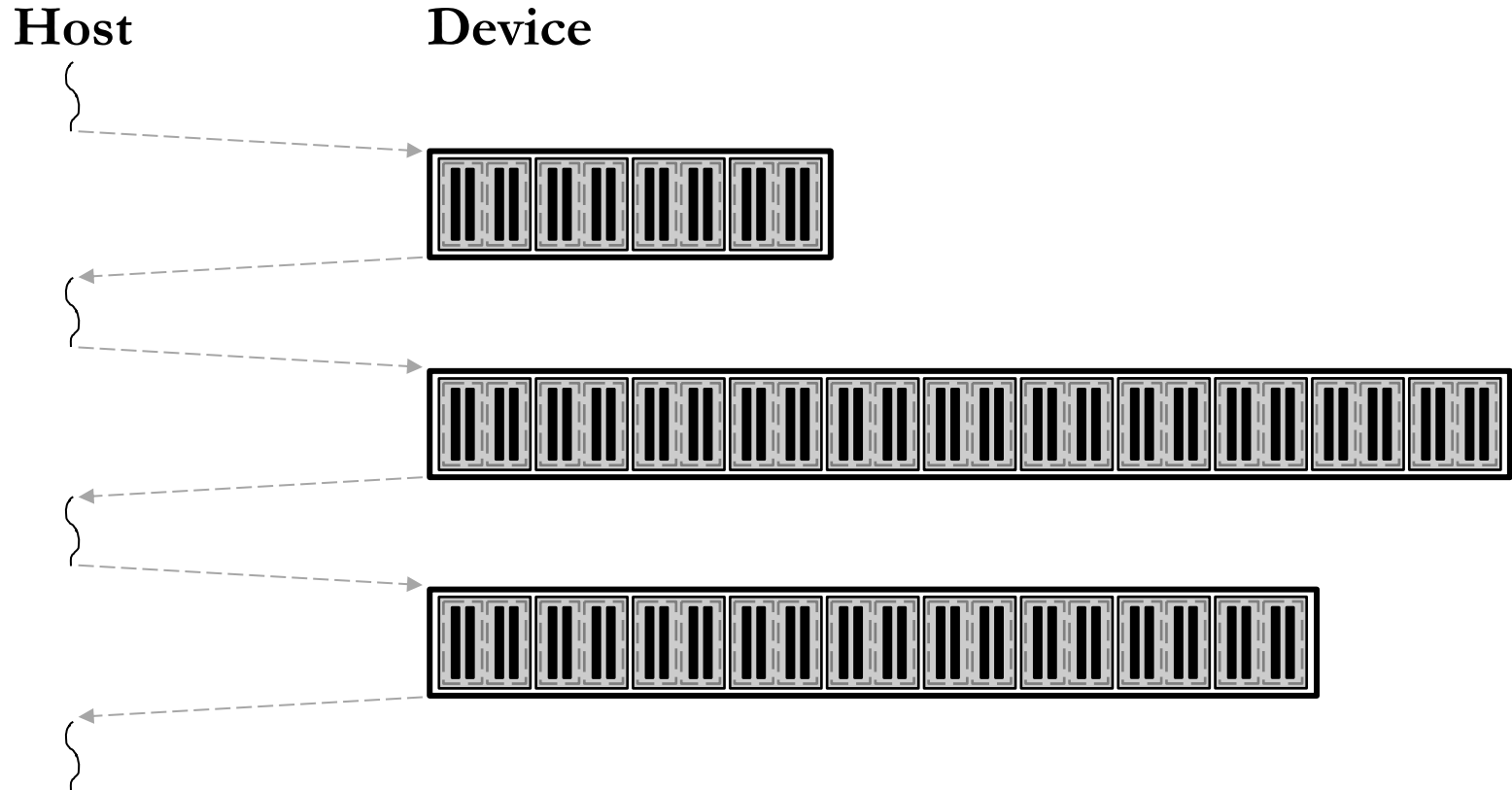
# Dynamic Parallelism

---

- GPU programming frameworks provide an interface to express **dynamic refinement algorithms** in a more natural way
  - Recall BFS
    - Each node in the frontier has a different number of neighbors
- CUDA Dynamic Parallelism
  - Important semantics when a kernel is launched from a kernel
  - Performance considerations

# Kernel Launch without Dynamic Parallelism

---

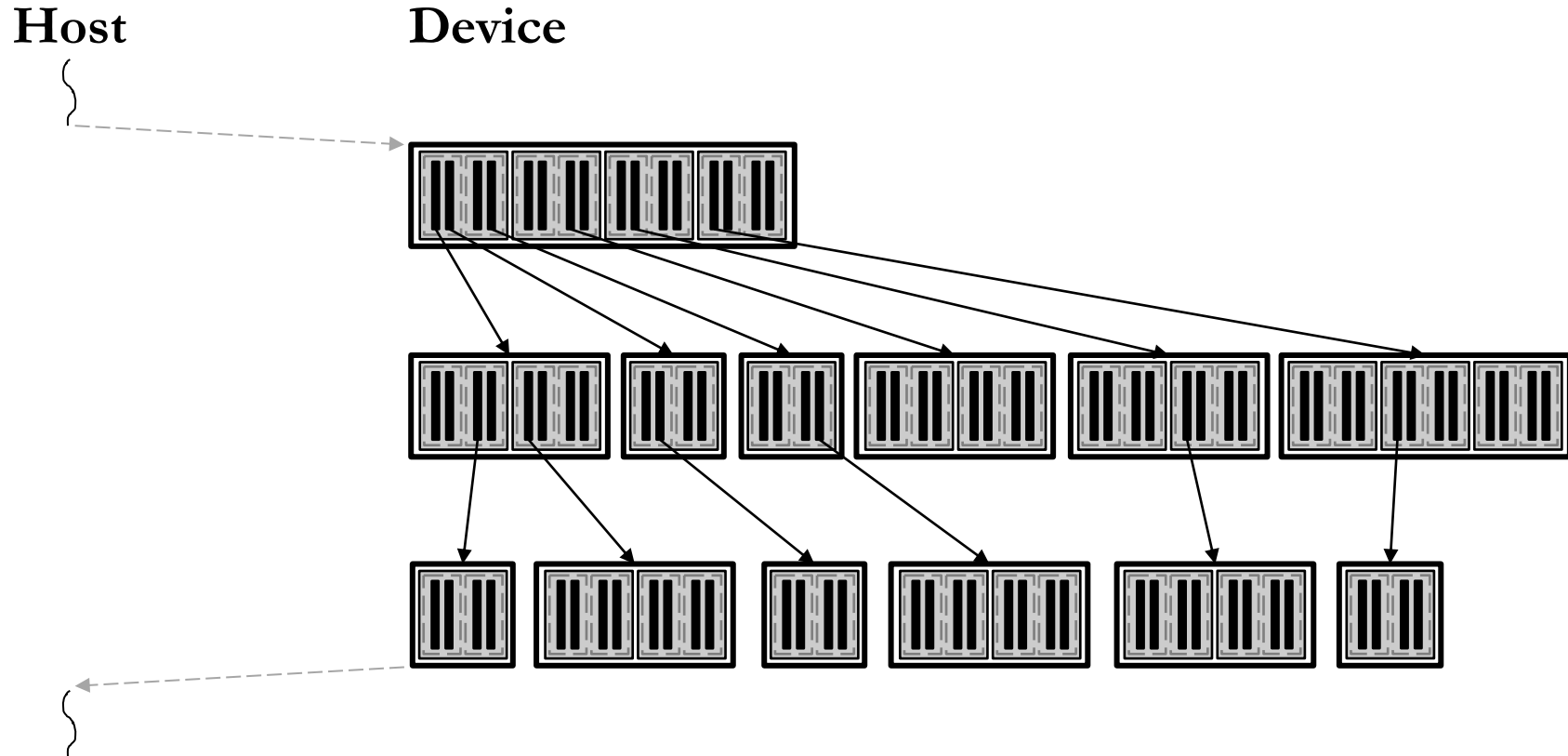


Previously, kernels could **only be launched from the host** (painful to program!)



# Kernel Launch with Dynamic Parallelism

---



Easier to write programs with **dynamically discovered parallelism**

# Lecture on Dynamic Parallelism

Block Granularity Aggregation (II)

```
__global__ void kernel(params) {
    kernel body
}
```

Original Kernel

```
__global__ void kernel_agg (param arrays, gD array, bD array) {
    calculate index of parent thread ( original kernel index) #
    load params from param arrays
    load actual gridDim/blockDim from gD/bD arrays
    calculate actual blockDim #
    if(threadIdx < actual blockDim) {
        kernel body (with kernel launches transformed and with
    }
```

Transformed Kernel  
(block-granularity aggregation example)

ETH zürich SAFARI

Juan Gomez L...

1:05:54 / 1:12:10

El Han et al., "KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism", MICRO 2021

## Heterogeneous Systems Course: Meeting 12: Dynamic Parallelism (Fall 2021)

287 views • Streamed live on Dec 22, 2021

18 DISLIKE SHARE + SAVE ...



**Onur Mutlu Lectures**  
21.5K subscribers

SUBSCRIBED



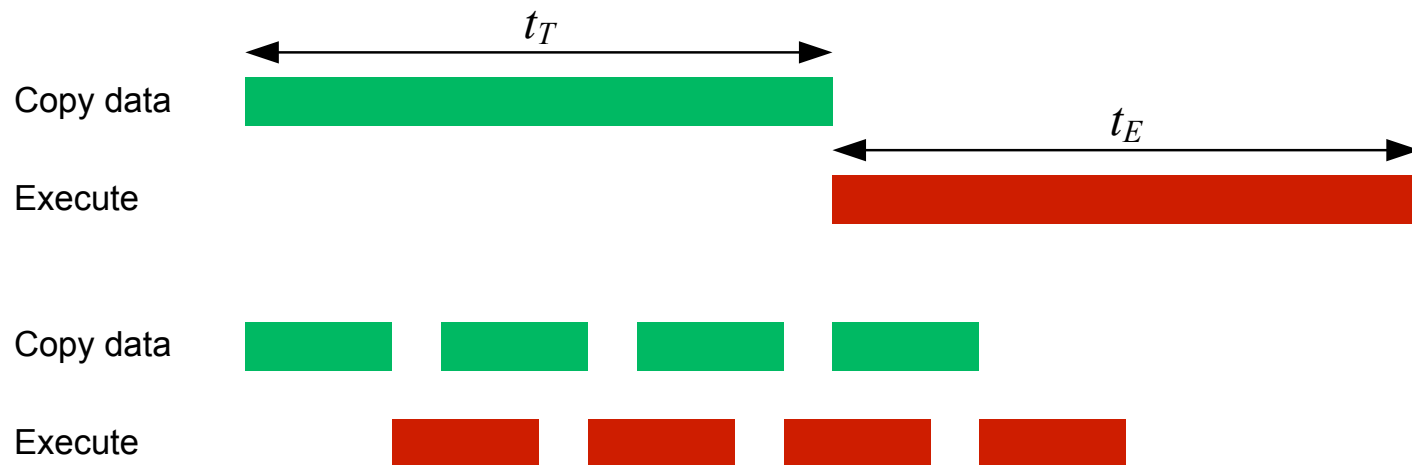
Project & Seminar, ETH Zürich, Fall 2021  
Hands-on Acceleration on Heterogeneous Computing Systems (  
[https://safari.ethz.ch/projects\\_and\\_s...](https://safari.ethz.ch/projects_and_s...))

<https://youtu.be/uzPkEQWaH4E>

# Asynchronous Data Transfers between CPU and GPU

# Recall: CUDA Streams

- **CUDA streams** (command queues in OpenCL)
- Sequence of operations that are performed in order
  - 1. Data transfer CPU-GPU
  - 2. Kernel execution
    - D input data instances, B blocks
    - #Streams:  $(D / \text{\#Streams})$  data instances,  $(B / \text{\#Streams})$  blocks
  - 3. Data transfer GPU-CPU



# Asynchronous Transfers between CPU & GPU

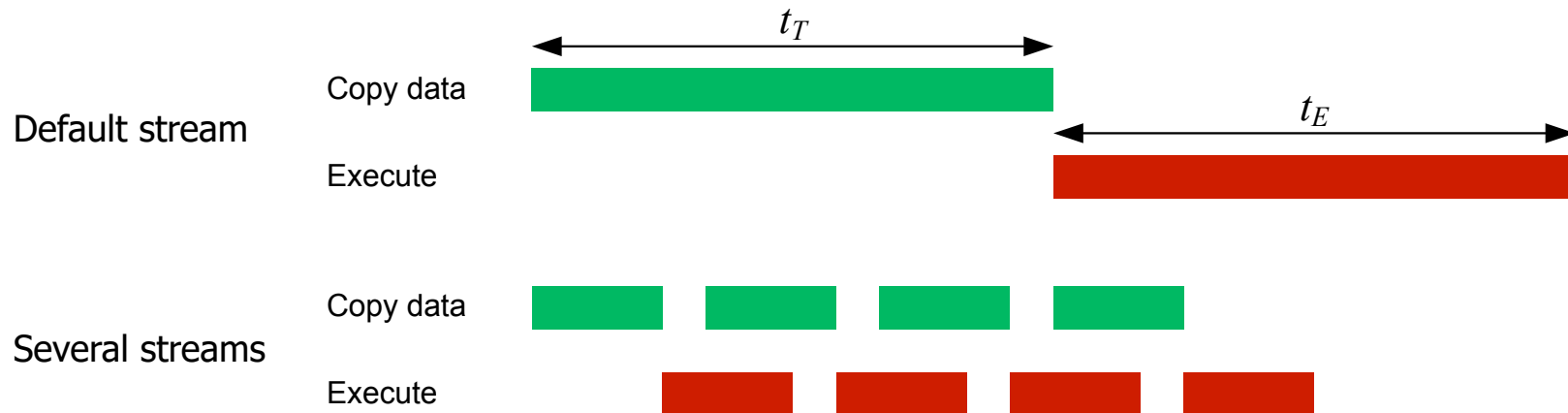
- Computation **divided into #Streams**

- D input data instances, B blocks

- #Streams

- D/#Streams data instances

- B/#Streams blocks



- Estimates

$$t_E + \frac{t_T}{\#Streams}$$

$t_E \geq t_T$  (dominant kernel)

$$t_T + \frac{t_E}{\#Streams}$$

$t_T > t_E$  (dominant transfers)

# Overlap of Data Transfers and Kernel Execution

Code for devices that do not support concurrent data transfers

```
// Create streams
int number_of_streams = 32;
cudaStream_t stream[number_of_streams]; // Stream declaration
for(int i = 0; i < number_of_streams; ++i)
    cudaStreamCreate(&stream[i]); // Stream creation

// CPU-GPU data transfers
for (int i = 0; i < number_of_streams; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size,
                    cudaMemcpyHostToDevice, stream[i]);

// Kernel launches
for (int i = 0; i < number_of_streams; ++i)
    MyKernel<<<num_blocks / number_of_streams, num_threads, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);

// GPU-CPU data transfers
for (int i = 0; i < number_of_streams; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size,
                    cudaMemcpyDeviceToHost, stream[i]);

cudaThreadSynchronize(); // Explicit synchronization

// Destroy streams
for (int i = 0; i < number_of_streams; ++i)
    cudaStreamDestroy(stream[i]); // Stream destruction
```

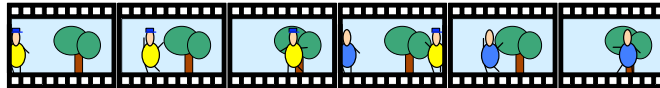
Check CUDA programming guide  
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams>

# Use Case: Video Processing

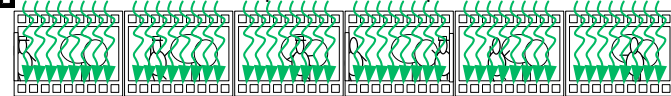
- Applications with independent computation on different data instances can benefit from asynchronous transfers
- For instance, **video processing**

Non-streamed execution

A sequence of 6 frames is transferred to device

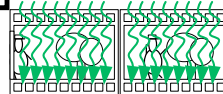
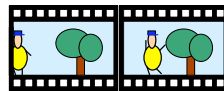


6 x  $b$  blocks compute on the sequence of frames

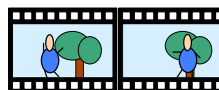
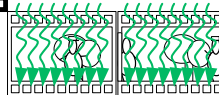
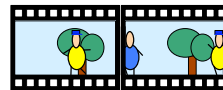


Streamed execution

A chunk of 2 frames is transferred to device



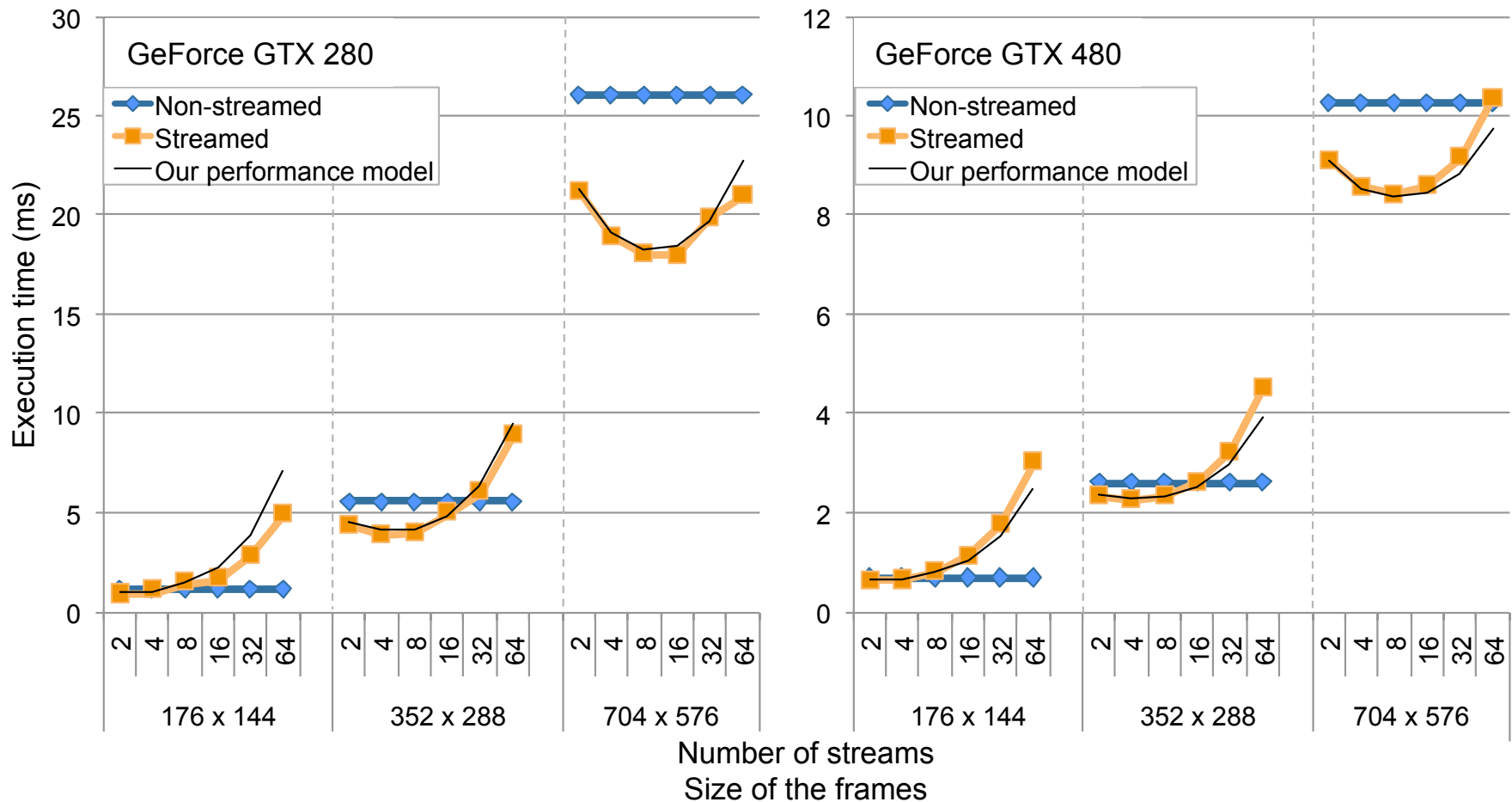
2 x  $b$  blocks compute on the chunk, while the second chunk is being transferred



Execution time saved thanks to streams

# Video Processing: Performance Results (I)

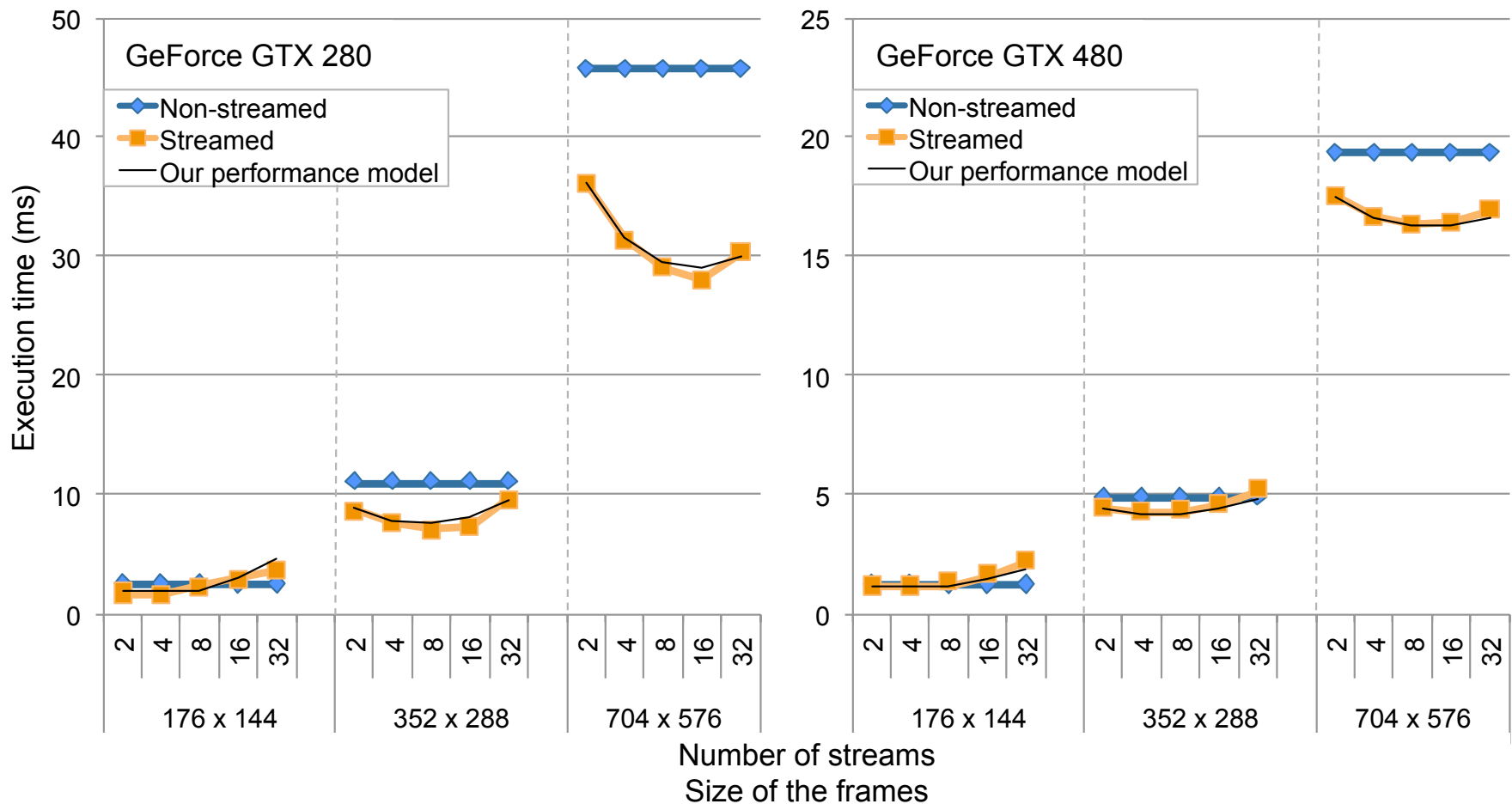
## ■ 256-bin histogram calculation





# Video Processing: Performance Results (II)

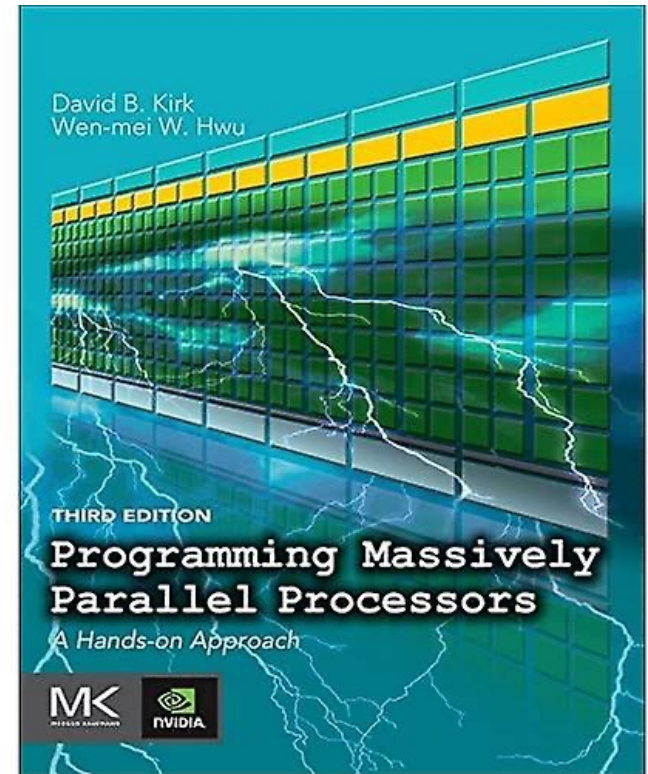
## ■ RGB-to-grayscale conversion



# Recommended Readings

---

- Hwu and Kirk, “**Programming Massively Parallel Processors,**” Third Edition, 2017
  - Chapter 18 - Programming a heterogeneous computing cluster, Section 18.5

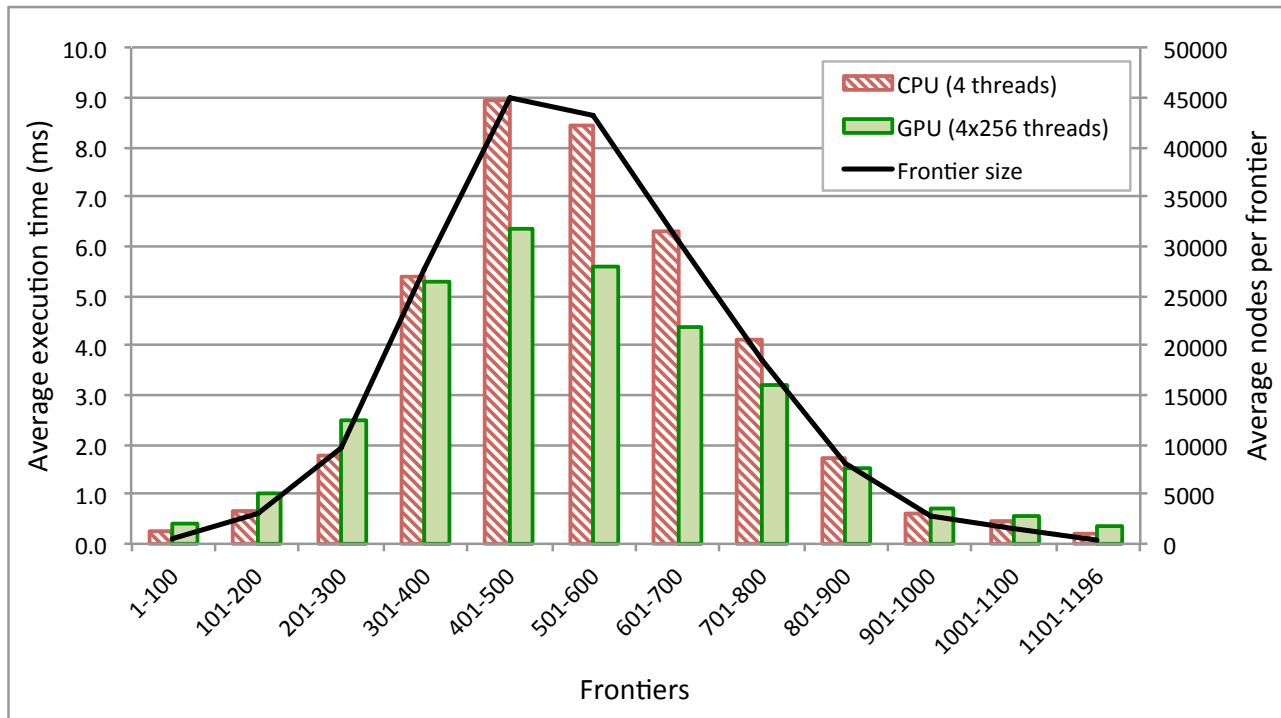


# Collaborative Computing

# Recall: BFS on CPU or GPU?

## ■ Motivation

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



# BFS: Collaborative Implementation (I)

---

- Choose CPU or GPU depending on frontier

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

    if(frontier_size < LIMIT){

        // Launch CPU threads
    }
    else{

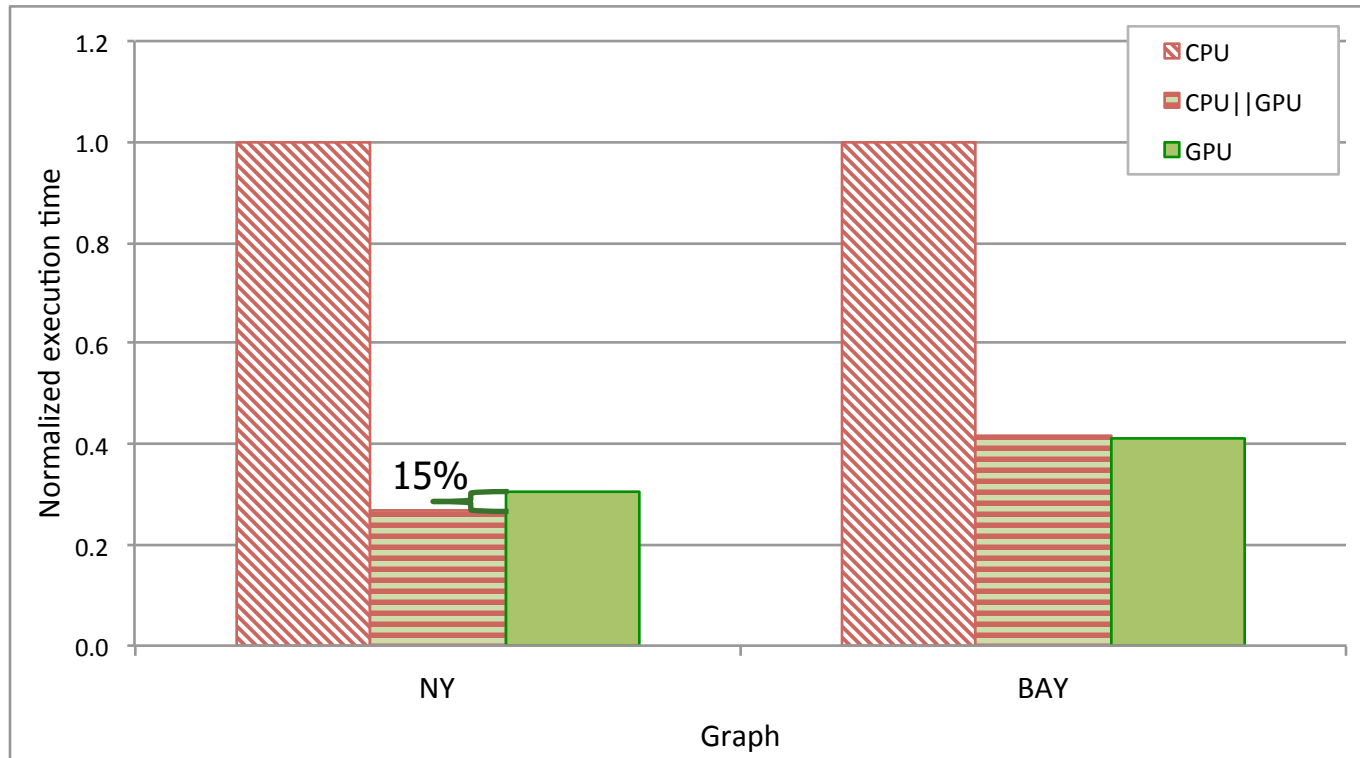
        // Launch GPU kernel
    }
}
```

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

# BFS: Collaborative Implementation (II)

## ■ Experimental results

- NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)



# Lecture on Graph Search

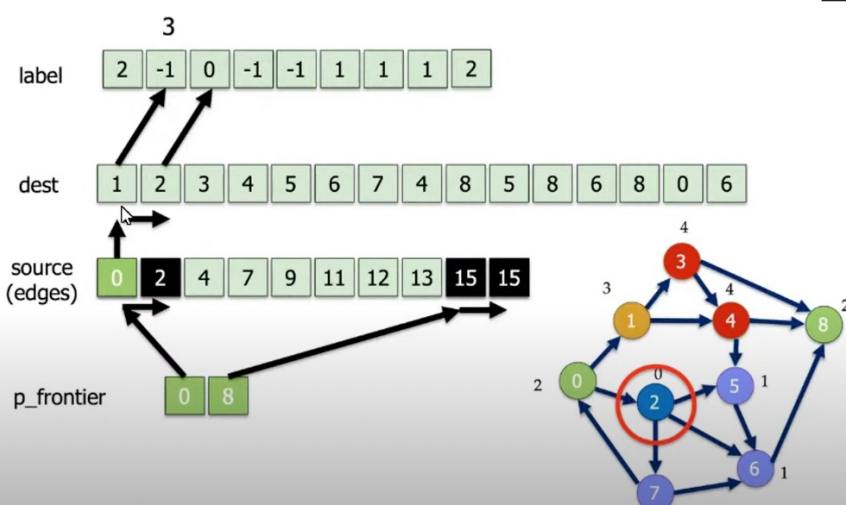
BFS: Processing the Frontier (2<sup>nd</sup> Iteration)

label: 2 -1 0 -1 -1 1 1 1 2

dest: 1 2 3 4 5 6 7 4 8 5 8 6 8 0 6

source (edges): 0 2 4 7 9 11 12 13 15 15

p\_frontier: 0 8



24:18 / 1:24:50

Slide 6 of 11: Hwu & Kirk

## Heterogeneous Systems Course: Meeting 11: Parallel Patterns: Graph Search (Fall 2021)

621 views • Streamed live on Dec 16, 2021

23 DISLIKE SHARE + SAVE ...



**Onur Mutlu Lectures**  
21.5K subscribers

SUBSCRIBED



Project & Seminar, ETH Zürich, Fall 2021  
Hands-on Acceleration on Heterogeneous Computing Systems (  
[https://safari.ethz.ch/projects\\_and\\_s...](https://safari.ethz.ch/projects_and_s...))

<https://youtu.be/95OnUeUuOGg>

# Unified Memory



# Memory Allocation and Data Transfers

- Traditional approach to **device allocation, CPU-GPU transfer, and GPU-CPU transfer**
  - ❑ `cudaMalloc()`;
  - ❑ `cudaMemcpy()`;
- Naturally matches systems with **discrete GPUs**

```
// 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();

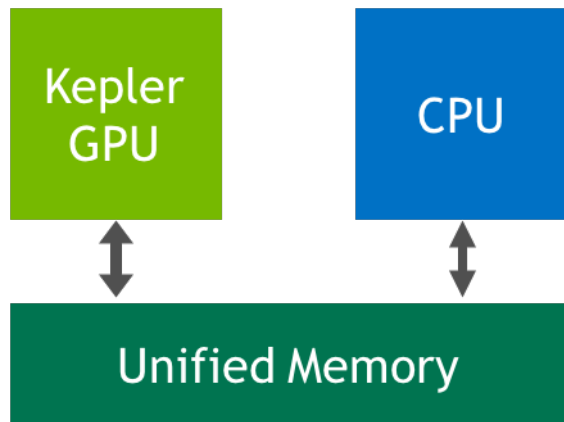
// Copy output to host memory
cudaMemcpy(output, d_output, ..., DeviceToHost);
```

# Unified Memory (I)

---

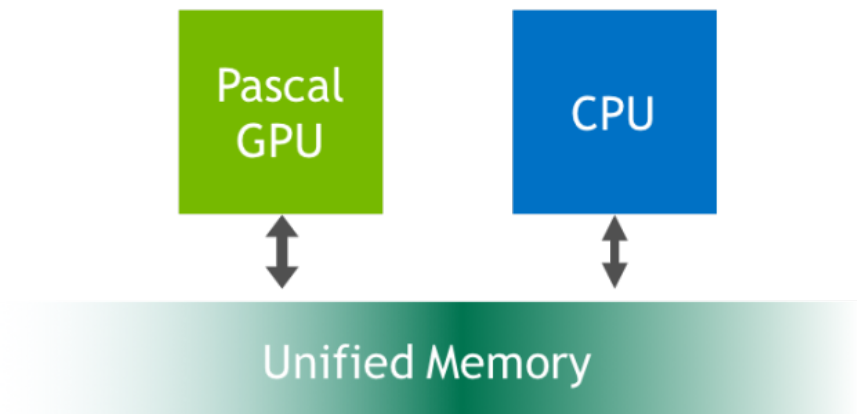
- Unified Virtual Address space
  - Same virtual address space across host and device
- CUDA 6.0: Unified memory
- CUDA 8.0 + Pascal: GPU page faults

CUDA 6 Unified Memory



(Limited to GPU Memory Size)

Pascal Unified Memory



(Limited to System Memory Size)

# Unified Memory (II)

---

- 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, ...);

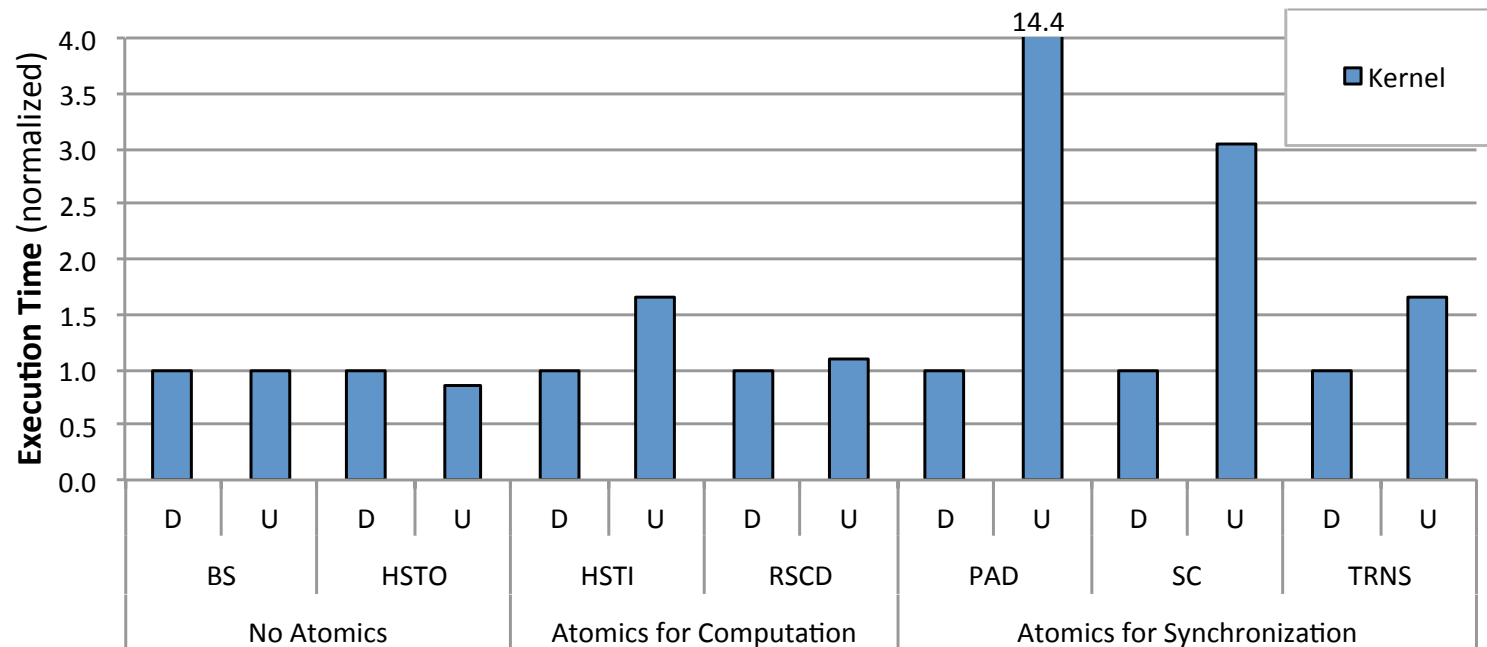
// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// Synchronize
cudaDeviceSynchronize();
```

- No need for double allocation or explicit data transfers
- Naturally matches physically integrated devices (e.g., CPU and GPU in the same chip) or devices with the same physical memory (e.g., CPU and GPU in the same package)
  - ❑ But it can also be implemented for discrete GPUs

# Unified Memory: Kernel Time

- IBM Power8 with NVIDIA Pascal GPU
  - **D**: Discrete (or traditional, without unified memory)
  - **U**: Unified memory

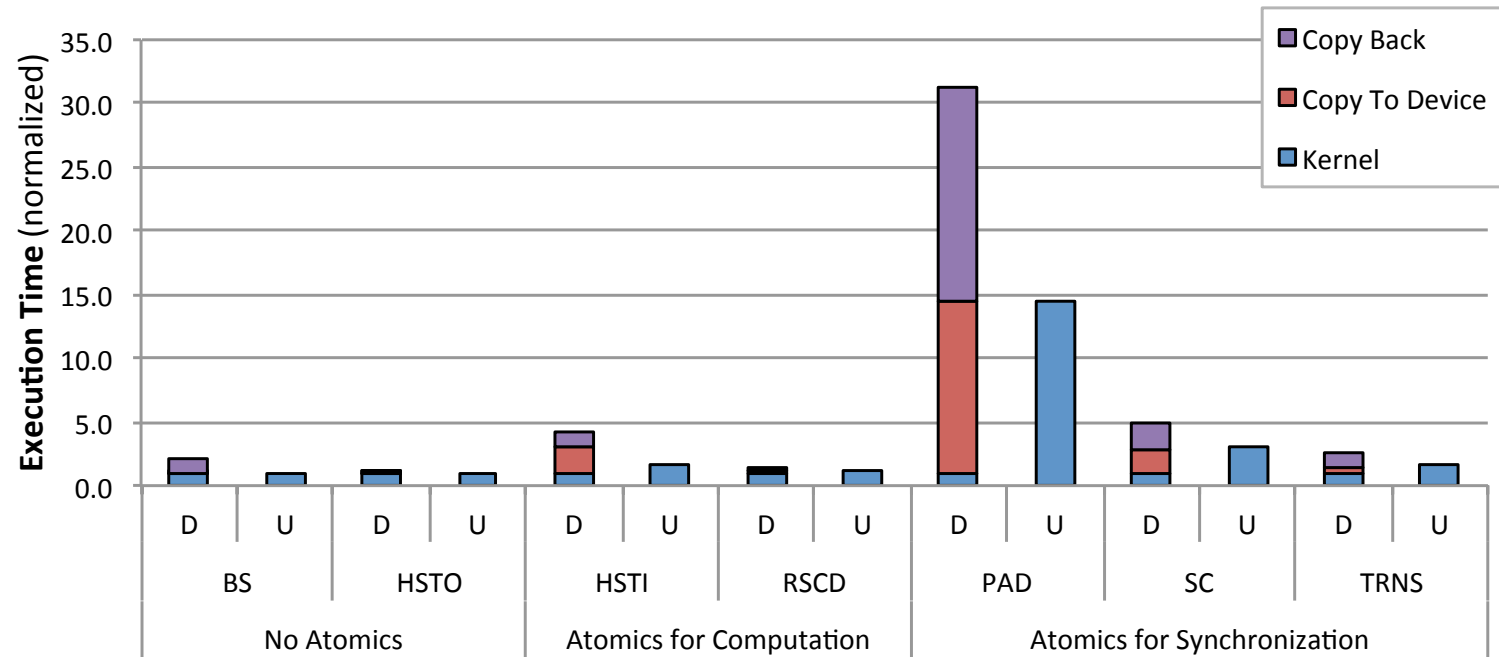


No cross-device  
communication

Cross-device communication may heavily  
burden kernel performance

# Unified Memory: Total Execution Time

- IBM Power8 with NVIDIA Pascal GPU
  - **D**: Discrete (or traditional, without unified memory)
  - **U**: Unified memory



Unified memory can hide data transfers with kernel execution

# How to Implement Collaborative Computing Applications?

# Collaborative Computing Applications

---

- 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-grained heterogeneity** becomes possible with unified memory (post Kepler/Maxwell architecture)
- Pascal/Volta/Turing/Ampere 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 and Later

---

- Unified memory

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

- System-wide atomics

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

# OpenCL 2.0 and Later

---

## ■ 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)
```

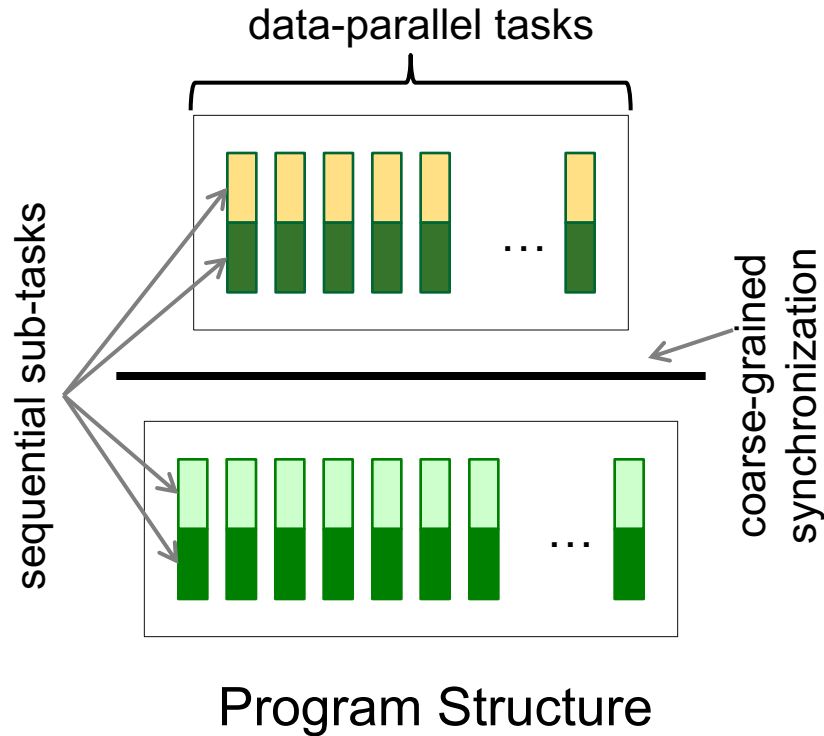
- Platform atomics (HSA)

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

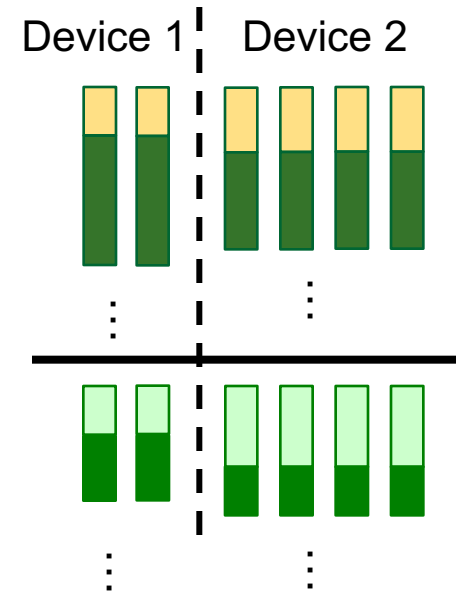
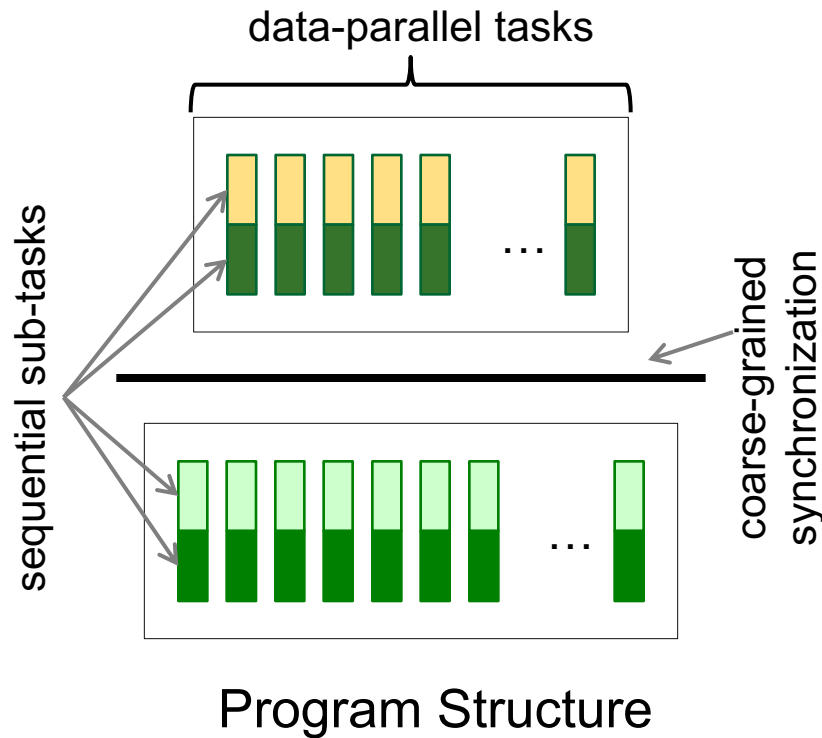
# Collaborative Patterns

# Traditional Program Structure

---

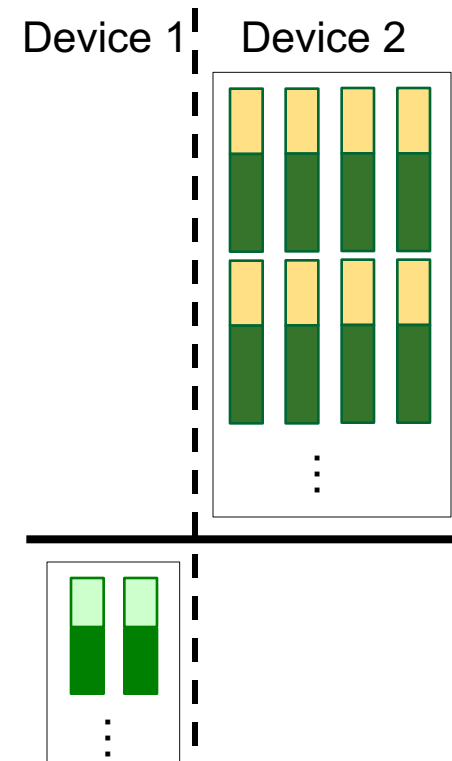
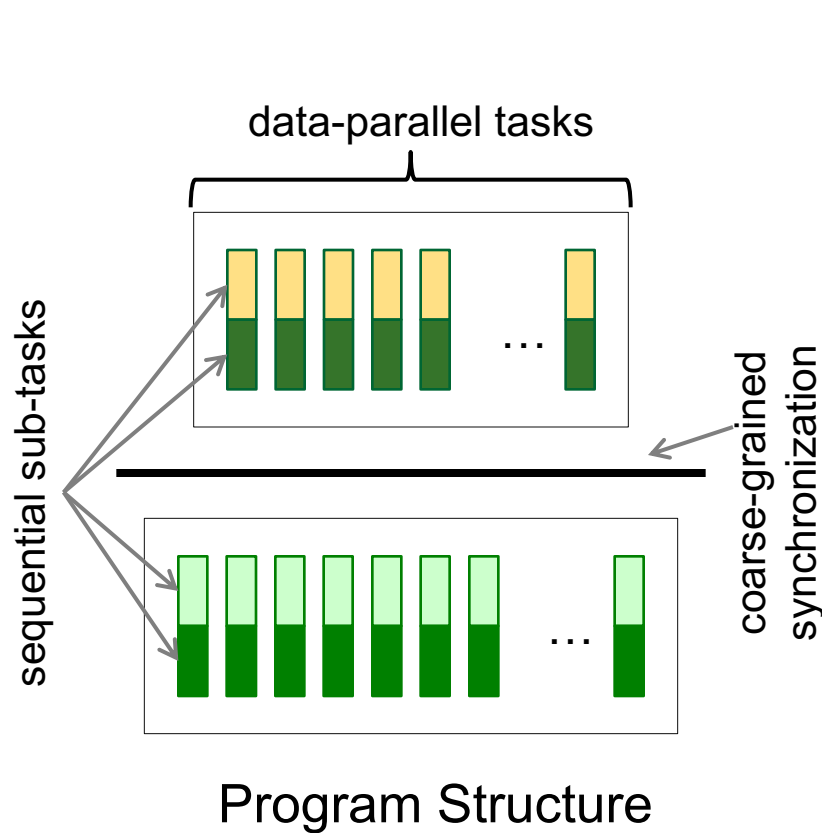


# Collaborative Patterns: Data Partitioning



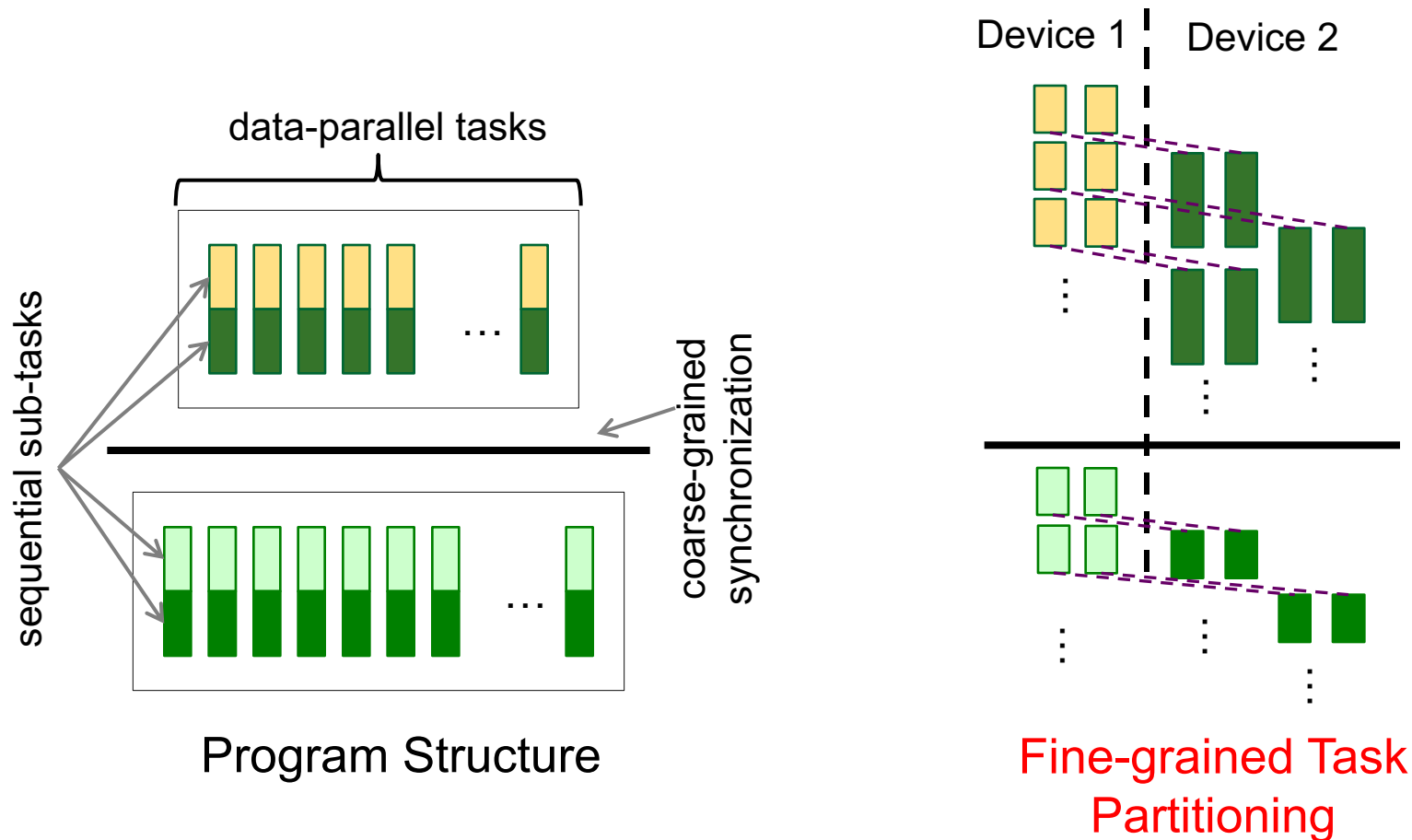
Data Partitioning

# Collaborative Patterns: Task Partitioning (I)



Coarse-grained Task Partitioning

# Collaborative Patterns: Task Partitioning (II)





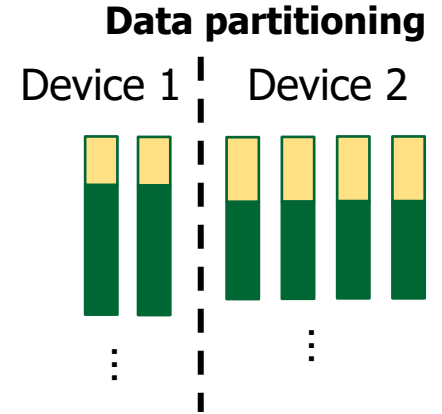
# Analytical Modeling

---

- $N$ : Number of data parallel tasks in the application
- $t_{i,D1}$ : Execution time of sub-task  $i$  by a Device 1 worker
- $t_{i,D2}$ : Execution time of sub-task  $i$  by a Device 2 worker
- $w_{D1}$ : Number of available Device 1 workers
- $w_{D2}$ : Number of available Device 2 workers
- $\beta$ : Distribution and aggregation overhead factor
- $\alpha$ : Fraction of data parallel tasks assigned to Device 1
- $S_{D1}$  and  $S_{D2}$  are, respectively, the set of subtasks executed in Device 1 and Device 2

# Analytical Model: Data Partitioning

- $N$ : Number of data parallel tasks in the application
- $t_{i,D1}$ : Execution time of sub-task  $i$  by a Device 1 worker
- $t_{i,D2}$ : Execution time of sub-task  $i$  by a Device 2 worker
- $w_{D1}$ : Number of available Device 1 workers
- $w_{D2}$ : Number of available Device 2 workers
- $\beta$ : Distribution and aggregation overhead factor
- $\alpha$ : Fraction of data parallel tasks assigned to Device 1



## Data partitioning

The total execution time is

$$t_{\text{data, total}} = \beta_{\text{data}} \cdot \max \left( \frac{\alpha N \sum_i t_{i,D1}}{w_{D1}}, \frac{(1 - \alpha) N \sum_i t_{i,D2}}{w_{D2}} \right)$$

Total D1 execution time (sequential execution)      Total D2 execution time (sequential execution)

Fixing all the variables except  $\alpha$ , the optimal  $\alpha$  (global minimum point) is

$$\alpha^* = \frac{\sum_i t_{i,D2}}{w_{D2}} / \left( \frac{\sum_i t_{i,D1}}{w_{D1}} + \frac{\sum_i t_{i,D2}}{w_{D2}} \right)$$

Workloads of Device 1 and Device 2 workers are balanced

# Analytical Model: Fine-Grained Task Part.

- $N$ : Number of data parallel tasks in the application
- $t_{i,D1}$ : Execution time of sub-task  $i$  by a Device 1 worker
- $t_{i,D2}$ : Execution time of sub-task  $i$  by a Device 2 worker
- $w_{D1}$ : Number of available Device 1 workers
- $w_{D2}$ : Number of available Device 2 workers
- $\beta$ : Distribution and aggregation overhead factor
- $S_{D1}$  and  $S_{D2}$  are, respectively, the set of subtasks executed in Device 1 and Device 2

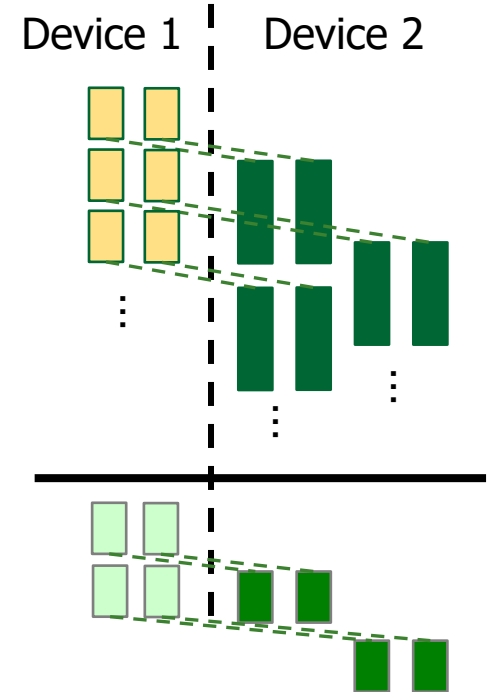
## Fine-grained task partitioning

The total execution time is

$$t_{\text{task, total}} = \beta_{\text{task}} N \cdot \max \left( \frac{\sum_{i \in S_{D1}} t_{i,D1}}{w_{D1}}, \frac{\sum_{i \in S_{D2}} t_{i,D2}}{w_{D2}} \right)$$

(Assume sub-tasks are very fine-grained)

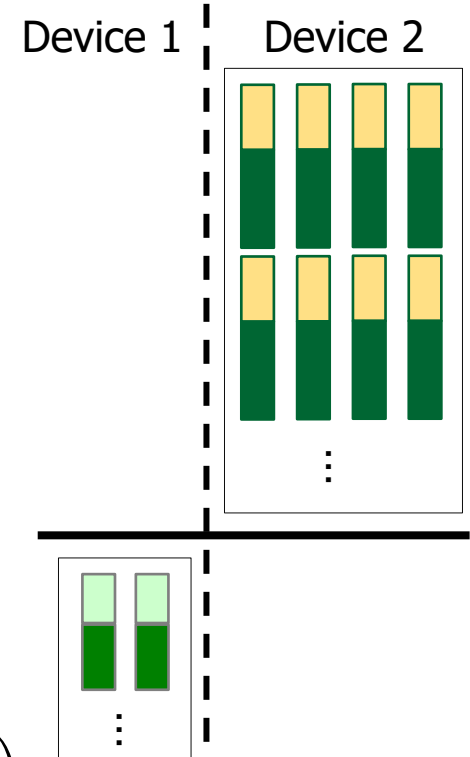
## Fine-grained task partitioning



# Analytical Model: Coarse-Grained Task Part.

- $N$ : Number of data parallel tasks in the application
- $t_{i,D1}$ : Execution time of sub-task  $i$  by a Device 1 worker
- $t_{i,D2}$ : Execution time of sub-task  $i$  by a Device 2 worker
- $w_{D1}$ : Number of available Device 1 workers
- $w_{D2}$ : Number of available Device 2 workers
- $\beta$ : Distribution and aggregation overhead factor
- $S_{D1}$  and  $S_{D2}$  are, respectively, the set of subtasks executed in Device 1 and Device 2

## Coarse-grained task partitioning



## Coarse-grained task partitioning

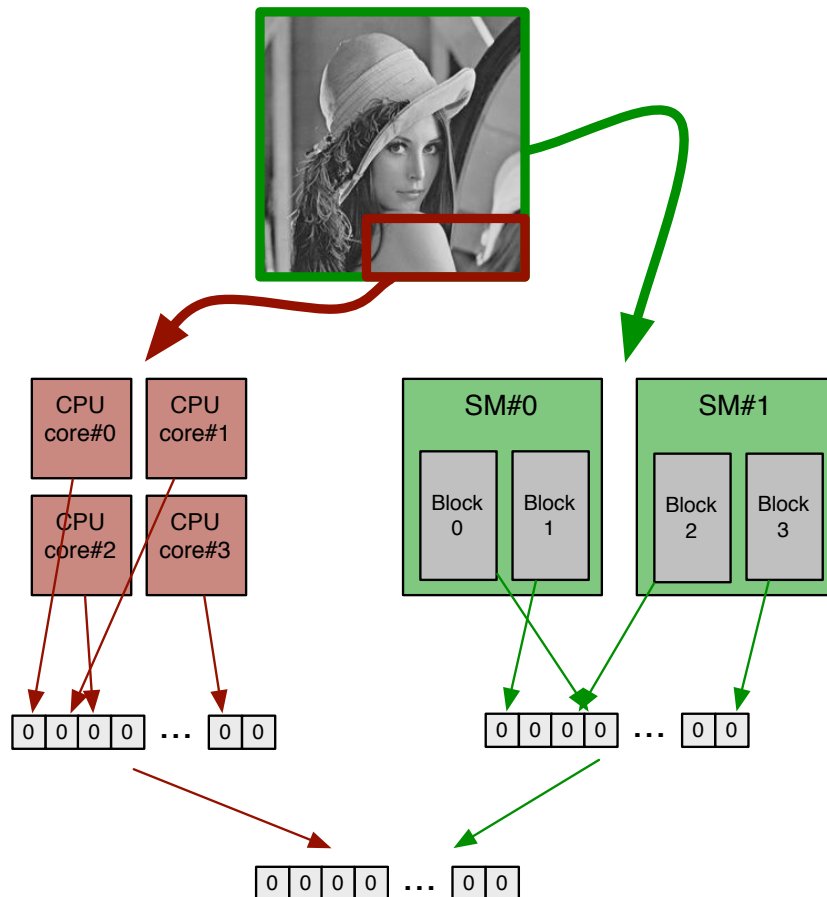
The total execution time is

$$t_{\text{task, total}} = \beta_{\text{task}} N \cdot \left( \frac{\sum_{i \in S_{D1}} t_{i,D1}}{w_{D1}} + \frac{\sum_{i \in S_{D2}} t_{i,D2}}{w_{D2}} \right)$$

# Data Partitioning

# Histogram without Unified Memory

- Traditional approach: **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);

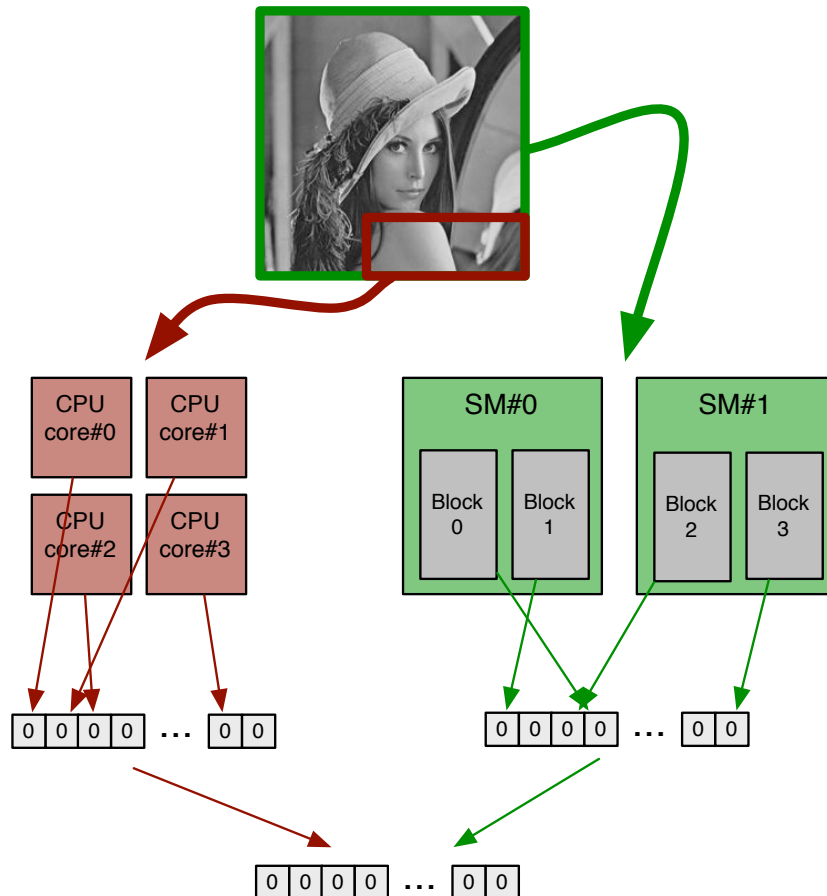
// Launch CPU threads
// Launch GPU kernel

cudaMemcpy(GPU histogram, DeviceToHost);

// Launch CPU threads for merging
```

# Histogram with Unified Memory (I)

- Traditional approach: **Separate CPU and GPU histograms** are merged at the end



```
malloc(CPU image);
cudaMallocManaged(GPU image);
memcpy(GPU image, CPU image, ...);

malloc(CPU histogram);
memset(CPU histogram, 0);
cudaMallocManaged(GPU histogram);
cudaMemset(GPU histogram, 0);

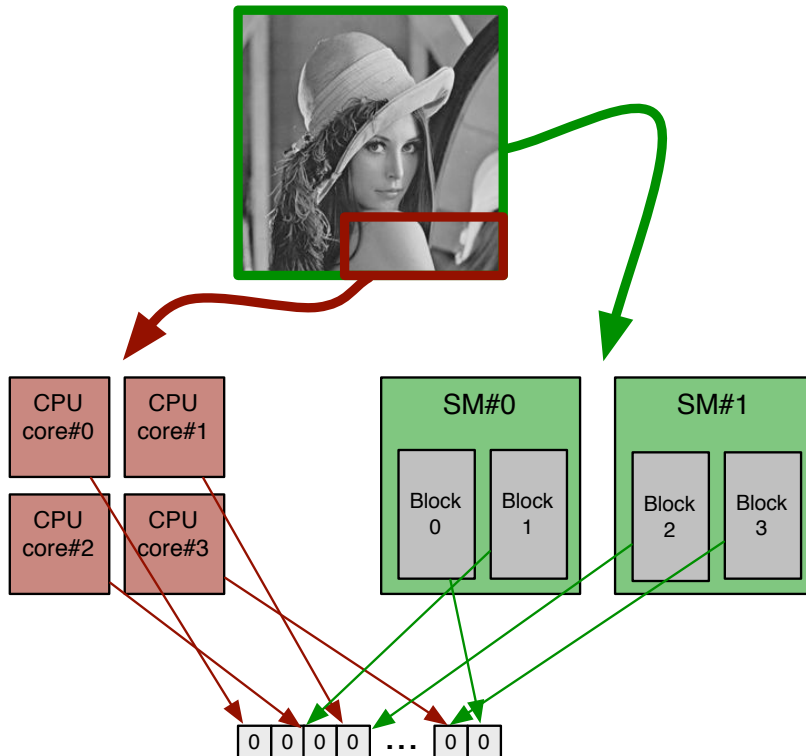
// Launch CPU threads
// Launch GPU kernel

cudaDeviceSynchronize();

// Launch CPU threads for merging
```

# Histogram with Unified Memory (II)

- System-wide atomic operations: **One single histogram**

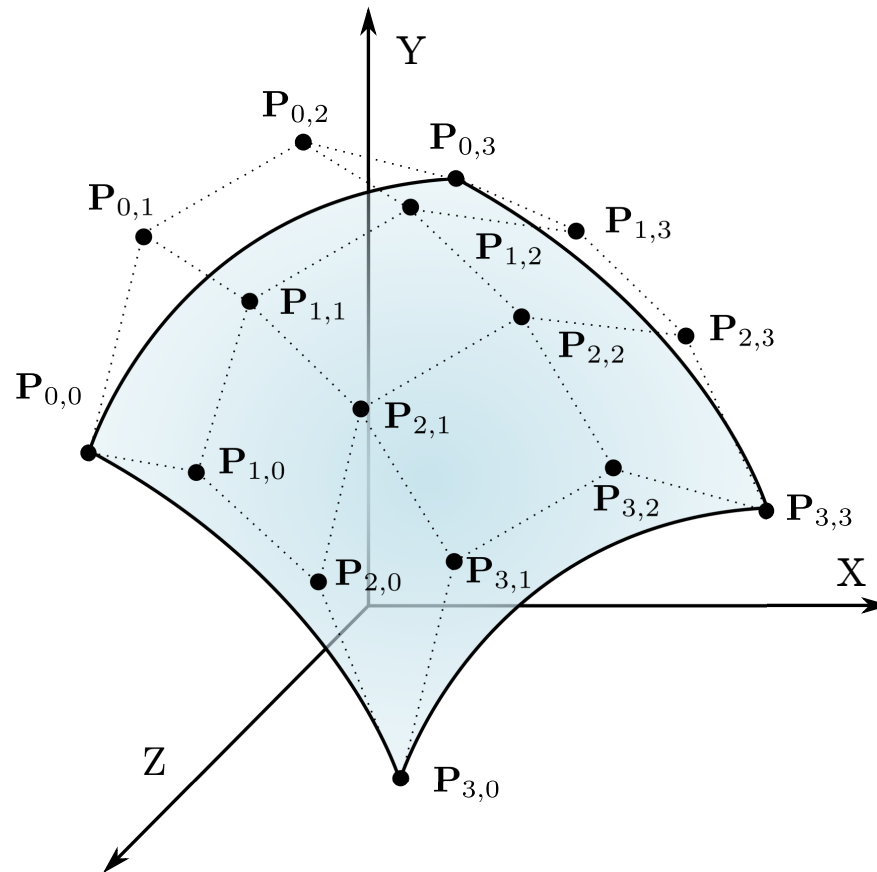


```
cudaMallocManaged(Histogram);  
cudaMemset(Histogram, 0);  
  
// Launch CPU threads  
// Launch GPU kernel (atomicAdd_system)
```



# Bézier Surfaces (I)

- Bézier surface: 4x4 net of control points



# Bézier Surfaces (II)

---

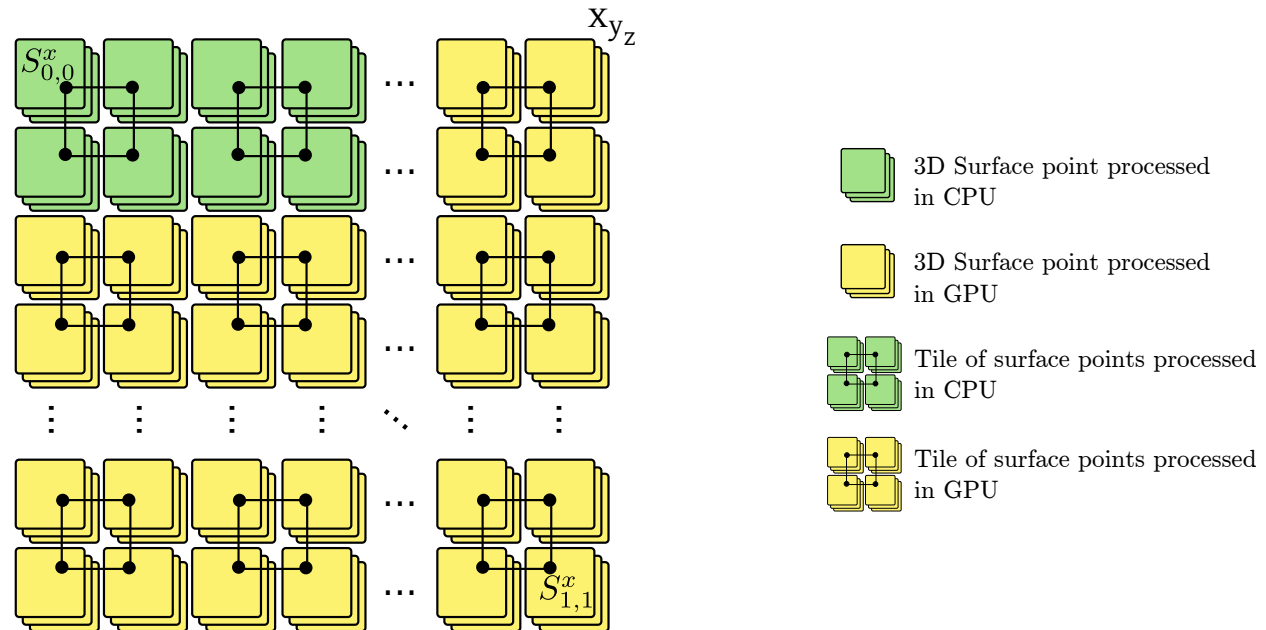
- Parametric non-rational formulation
  - Bernstein polynomials
  - Bi-cubic surface  $m = n = 3$

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

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

# Bézier Surfaces: Static Distribution (I)

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



# Bézier Surfaces: Static Distribution (II)

---

## ■ 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, ...);

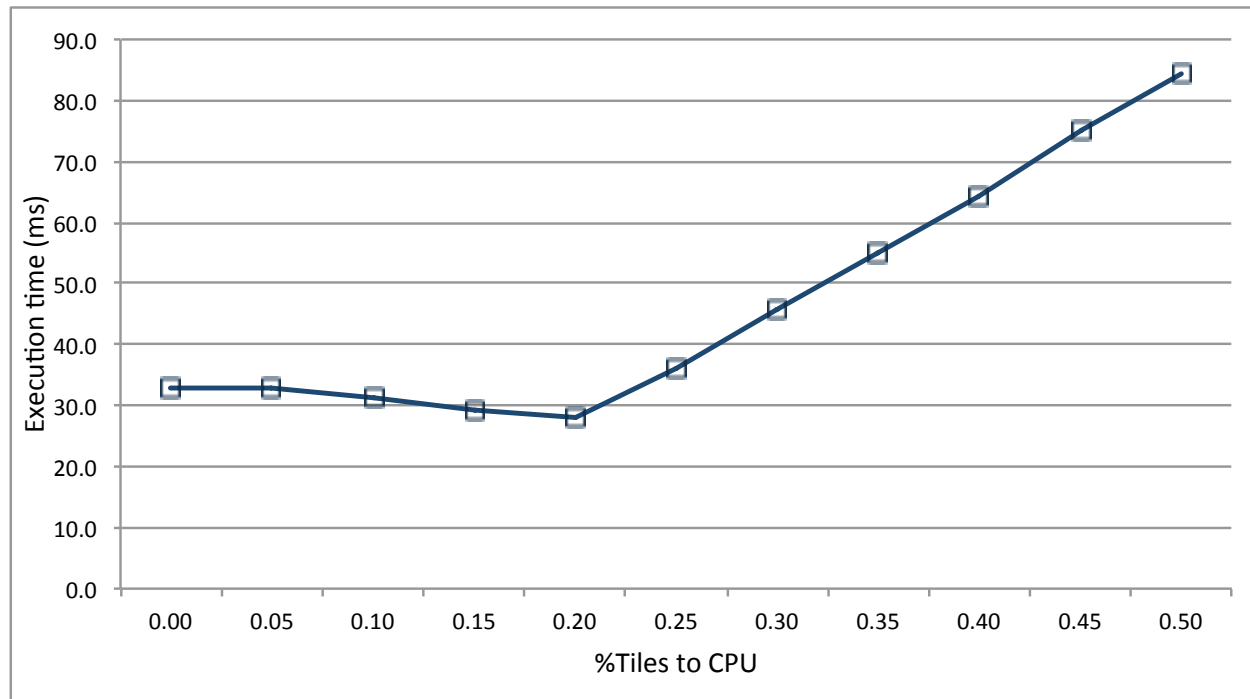
// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_surface, d_control_points, ...);

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

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

# Bézier Surfaces: Static Distribution (III)

- Performance results on NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)
  - ❑ Bezier surface: 300x300, 4x4 control points
  - ❑ %Tiles to CPU
  - ❑ 17% speedup over GPU only



# Bézier Surfaces with Unified Memory

---

## ■ With 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
cudaMallocManaged(surface, ...);

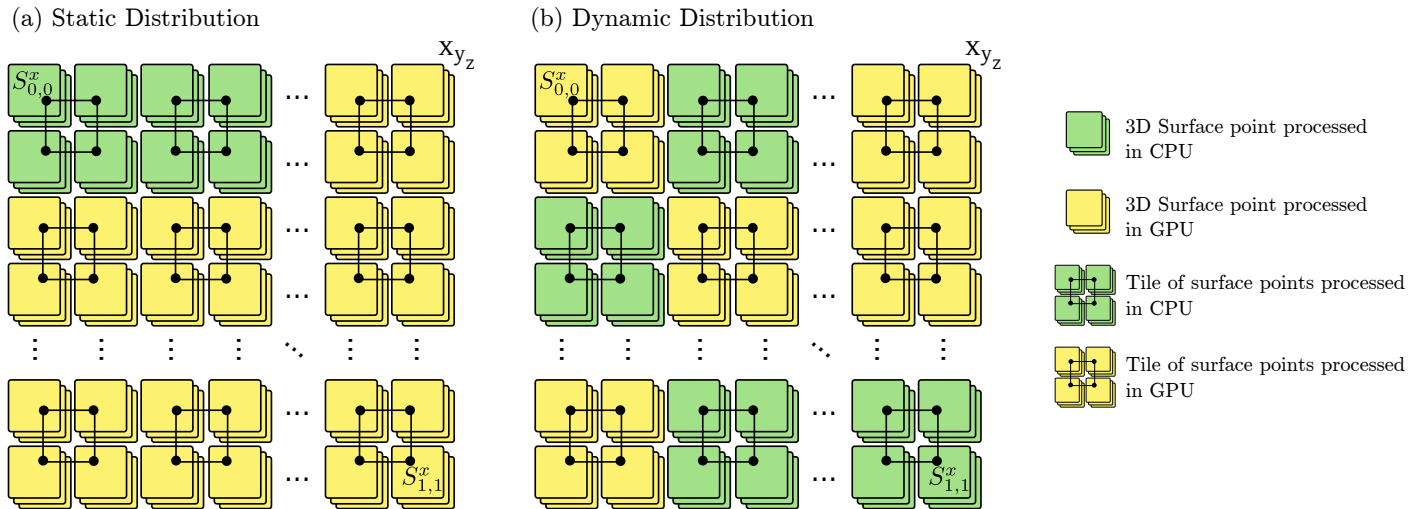
// Launch CPU threads
std::thread main_thread (run_cpu_threads, control_points, surface, ...);

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

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

# Bézier Surfaces: Dynamic Distribution

## ■ Static vs. dynamic implementation



## □ Pascal/Volta/Turing/Ampere Unified Memory: system-wide atomic operations

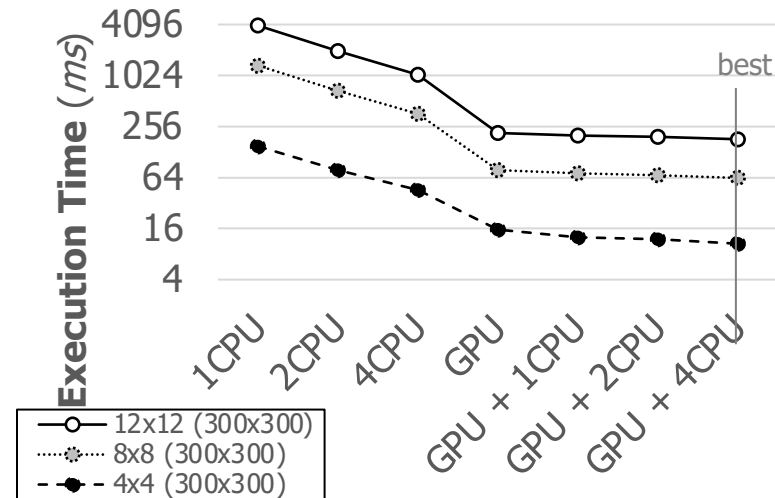
```
while(true){
    if(threadIdx.x == 0)
        my_tile = atomicAdd_system(tile_num, 1); // my_tile in shared memory; tile_num in UM

    __syncthreads(); // Synchronization

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

# Benefits of Collaboration: Bézier Surfaces

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

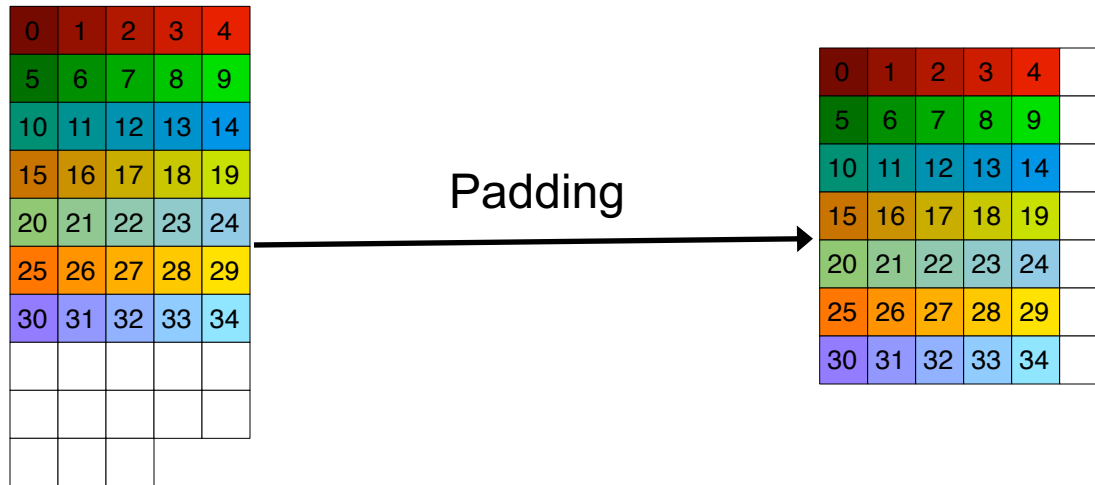


Bézier Surfaces  
(up to 47% improvement over GPU only)



# Padding (I)

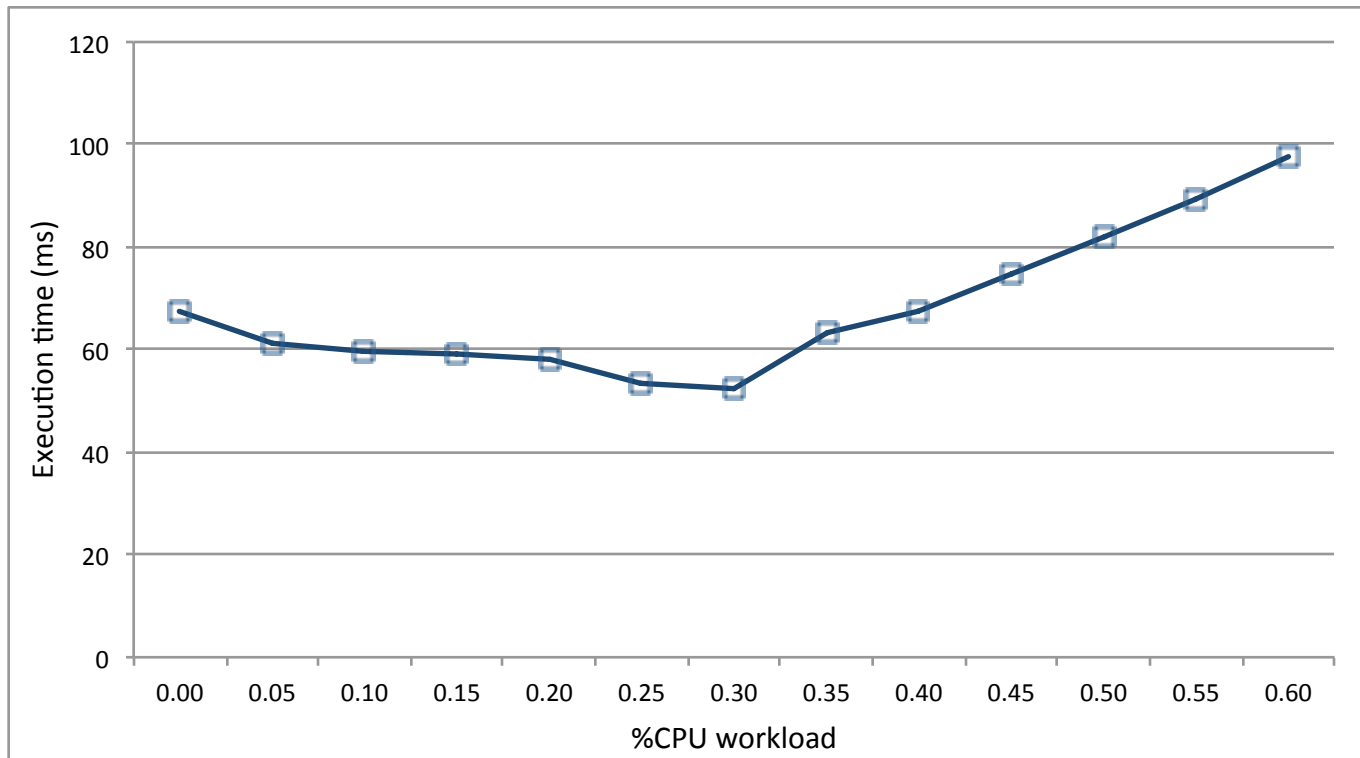
- Matrix padding
  - Memory alignment
  - Transposition of near-square matrices



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

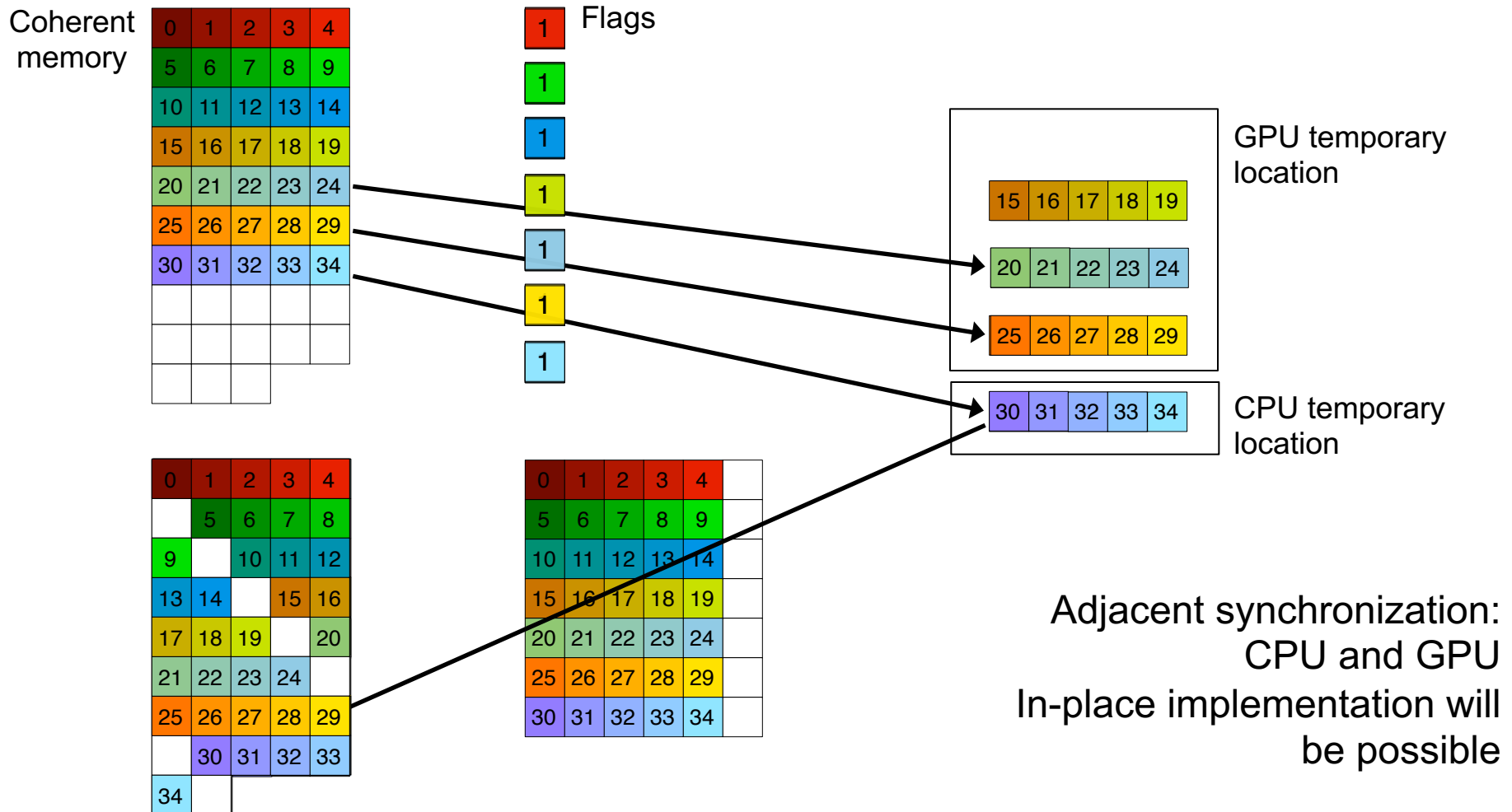
# Padding (II)

- Performance results on NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)
  - Matrix size: 4000x4000, padding = 1
  - 29% speedup over GPU only



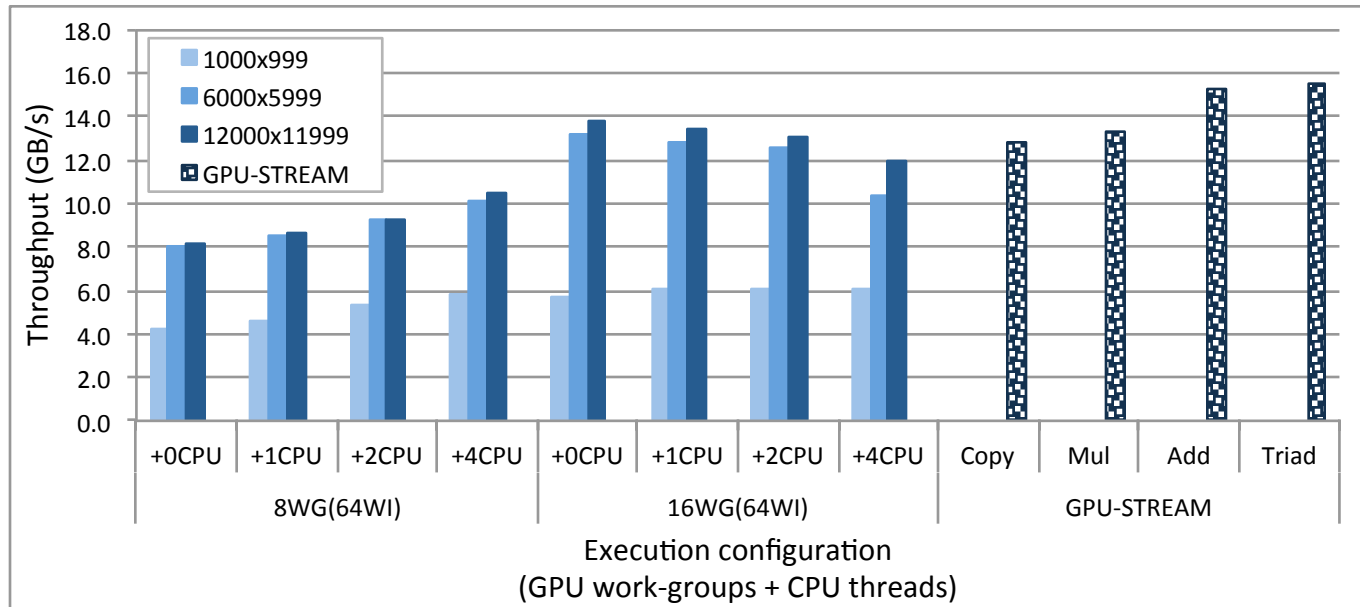
# In-Place Padding

## ■ Pascal/Volta/Turing/Ampere Unified Memory



# Benefits of Collaboration: Padding

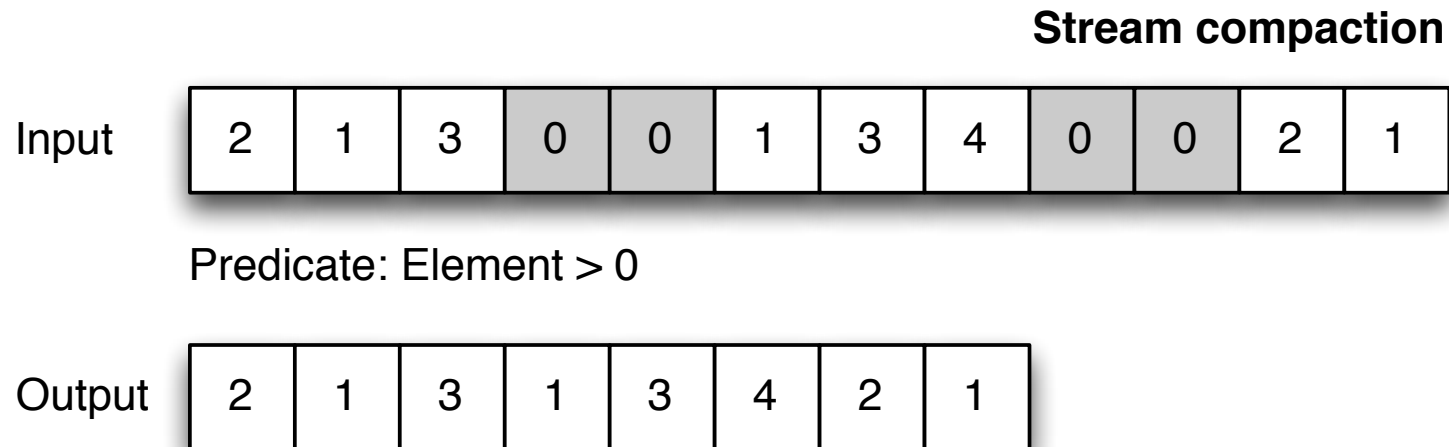
- AMD Kaveri (4 CPU cores + 8 GPU cores)
  - Optimal number of devices is not always the maximum



# Stream Compaction (I)

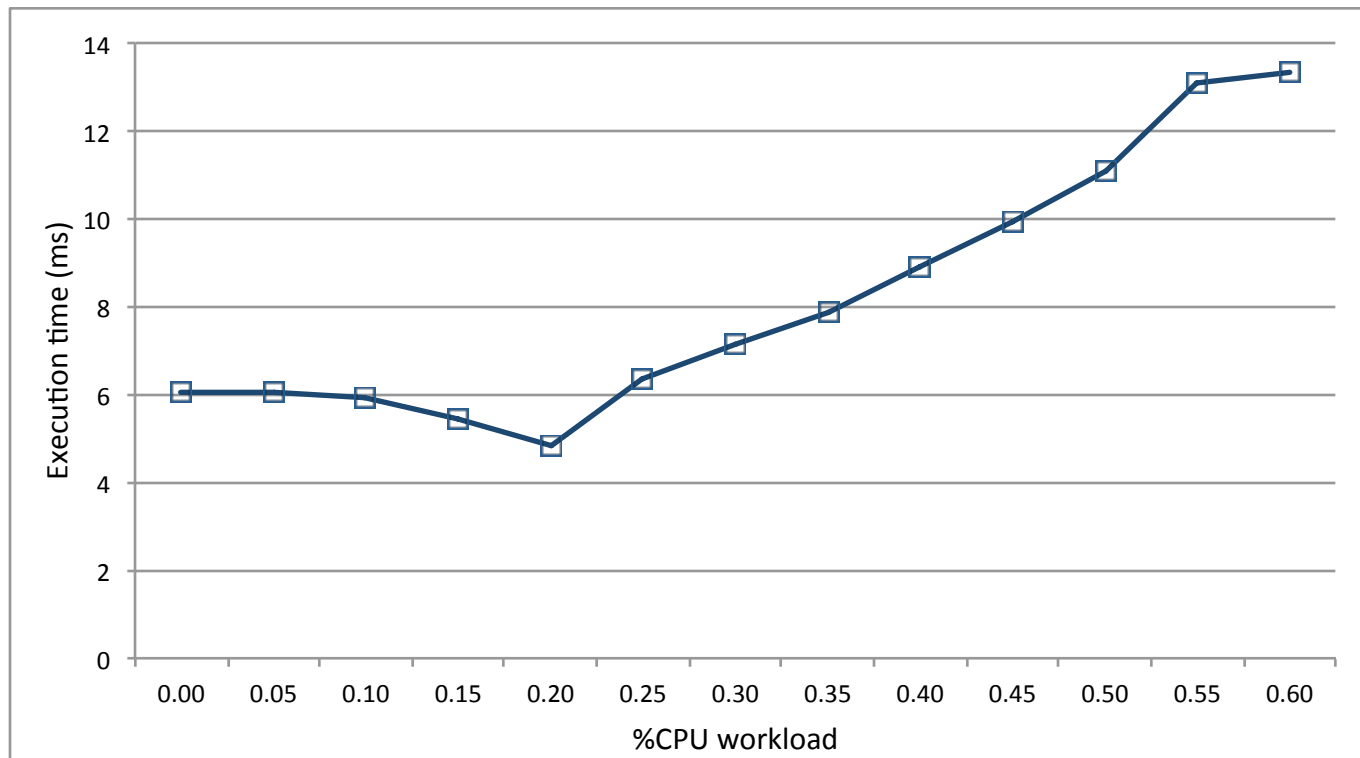
---

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



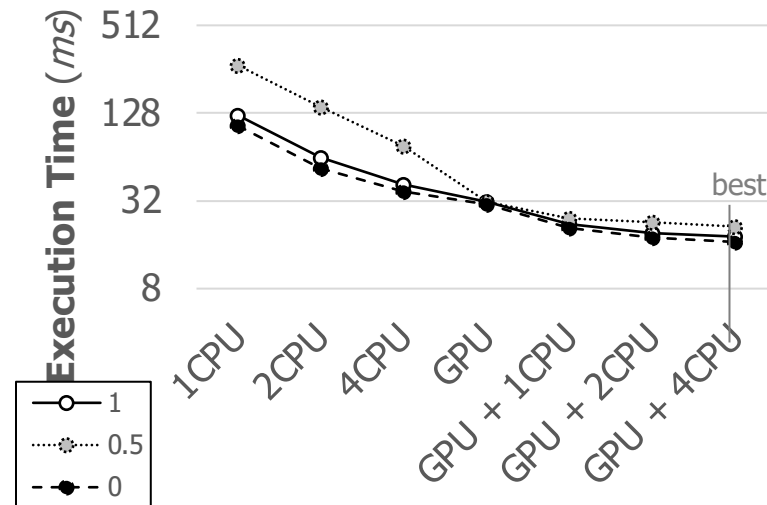
# Stream Compaction (II)

- Performance results on NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)
  - Array size: 2 MB, filtered items = 50%
  - 25% speedup over GPU only



# Benefits of Collaboration: Stream Comp.

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



# Coarse-Grained Task Partitioning



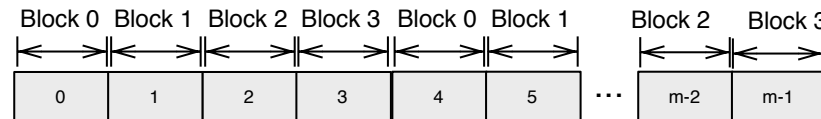
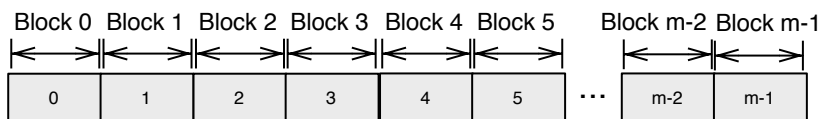
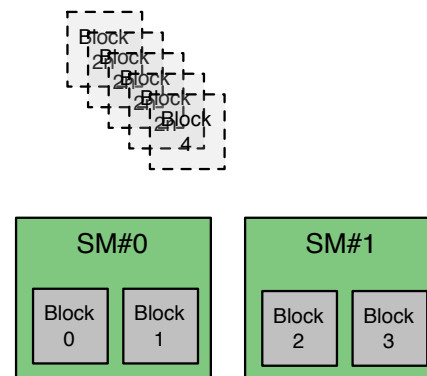
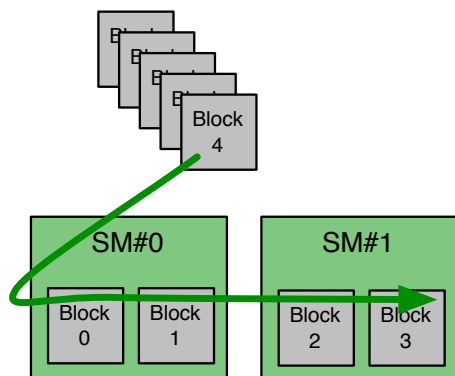
# Breadth-First Search

---

- 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

# Recall: Persistent Thread Blocks

- Combine Kernel 1 and Kernel 2
- We can **avoid kernel re-launch**
- We need to use **persistent thread blocks**
  - Kernel 2 launches ( $\text{frontier\_size} / \text{block\_size}$ ) blocks
  - Persistent blocks: up to  $(\text{number\_SMs} \times \text{max\_blocks\_SM})$



# Atomic-based Block Synchronization (I)

---

## ■ 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 (II)

---

## ■ 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
}

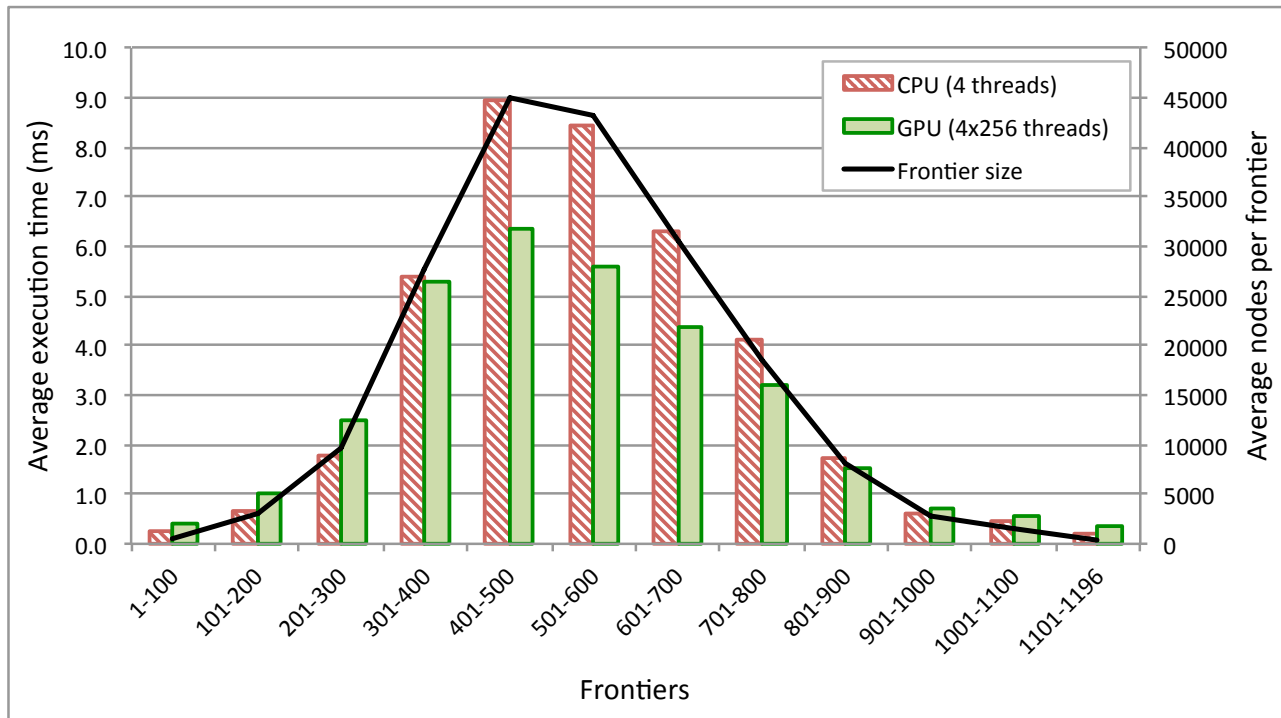
__syncthreads(); // Rest of threads wait here

...
```

# Recall: BFS on CPU or GPU?

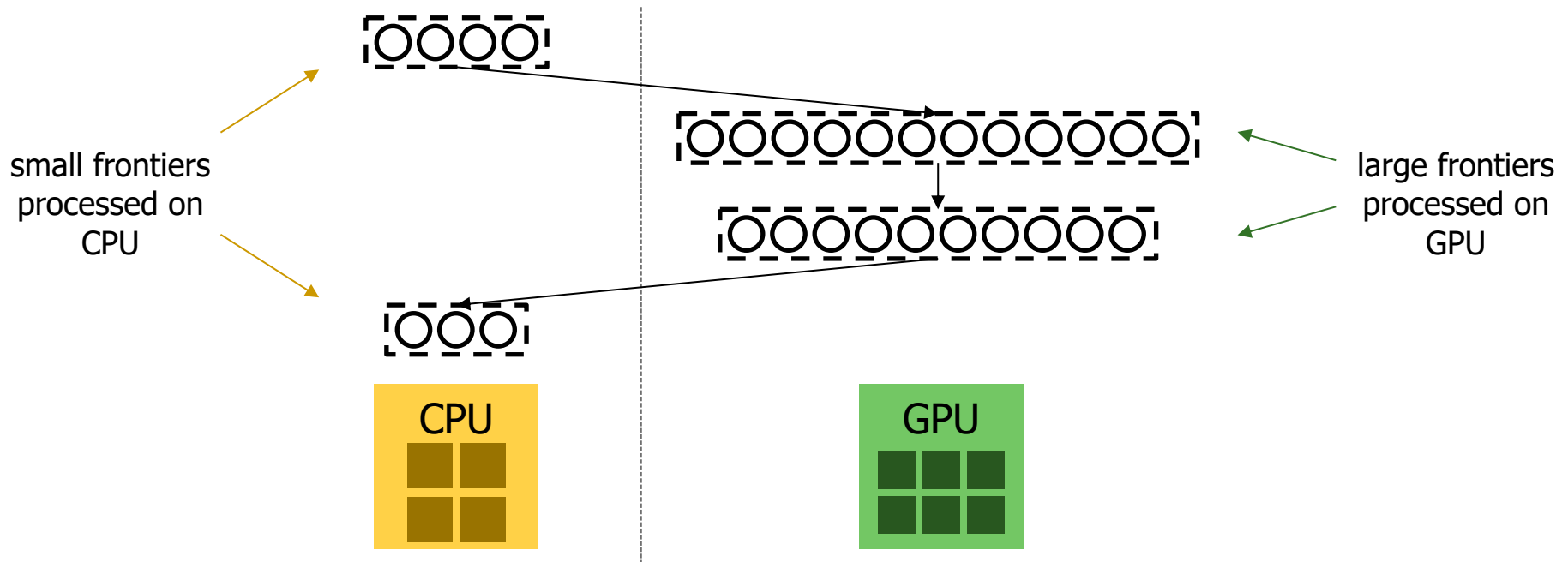
## ■ Motivation

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



# BFS: Collaborative Implementation (I)

- Choose the most appropriate device



# BFS: Collaborative Implementation (II)

---

- Choose CPU or GPU depending on frontier

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

    if(frontier_size < LIMIT){

        // Launch CPU threads
    }
    else{

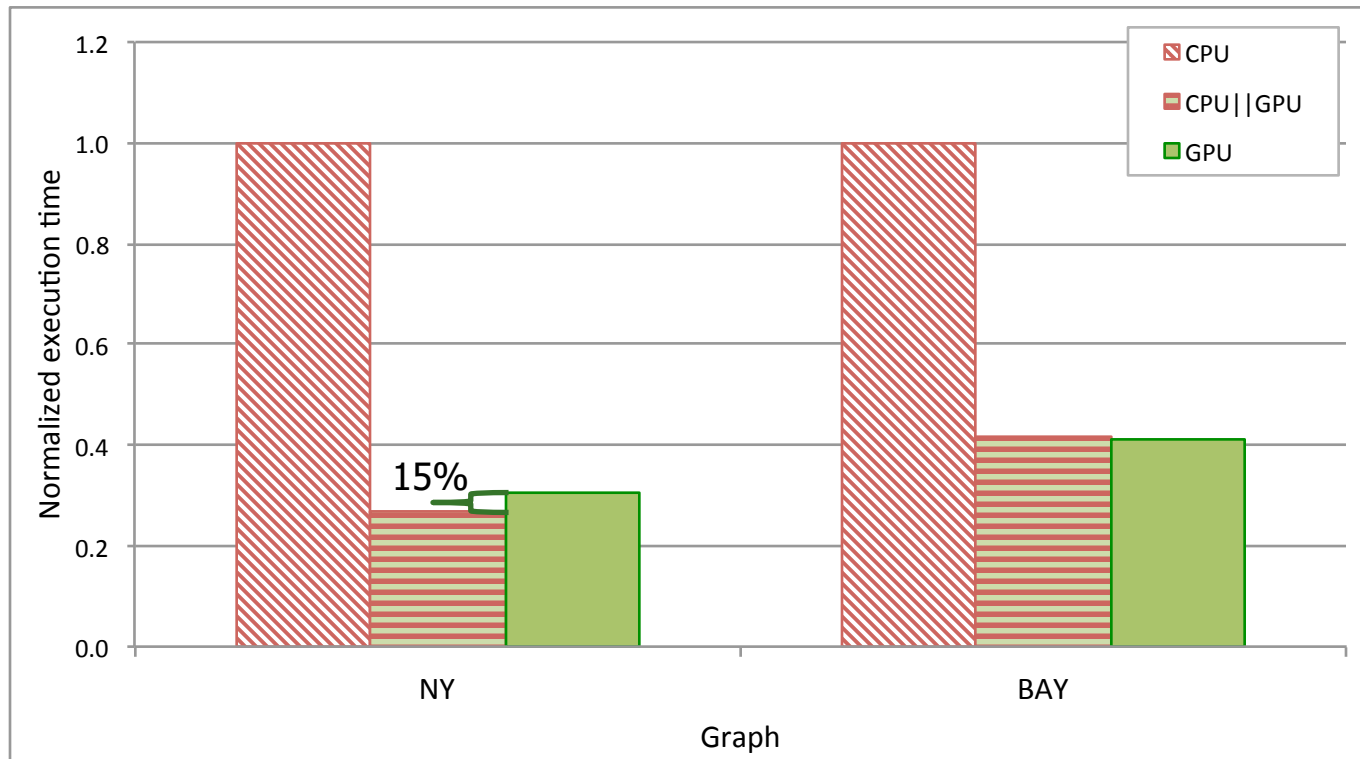
        // Launch GPU kernel
    }
}
```

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

# BFS: Collaborative Implementation (III)

## ■ Experimental results

- NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)





# Collaborative Implementation without UM

---

- **Without** Unified Memory (UM)
  - 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 with UM (I)

---

## ■ 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();

    }

}
```

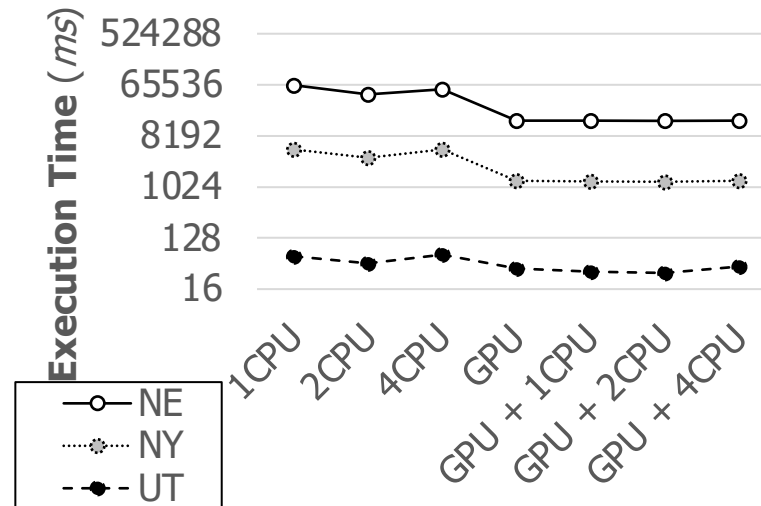
# Collaborative Implementation with UM (II)

---

- Pascal/Volta/Turing/Ampere 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

- AMD Kaveri (4 CPU cores + 8 GPU cores)
  - SSSP performs more computation than BFS

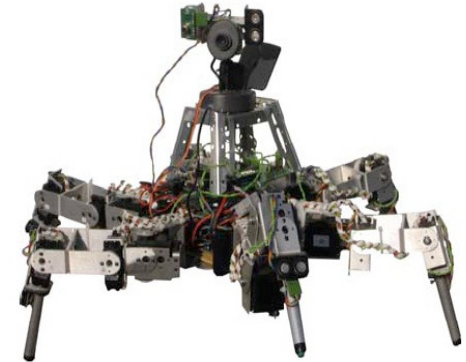


Single Source Shortest Path  
(up to 22% improvement over GPU only)

# Fine-Grained Task Partitioning

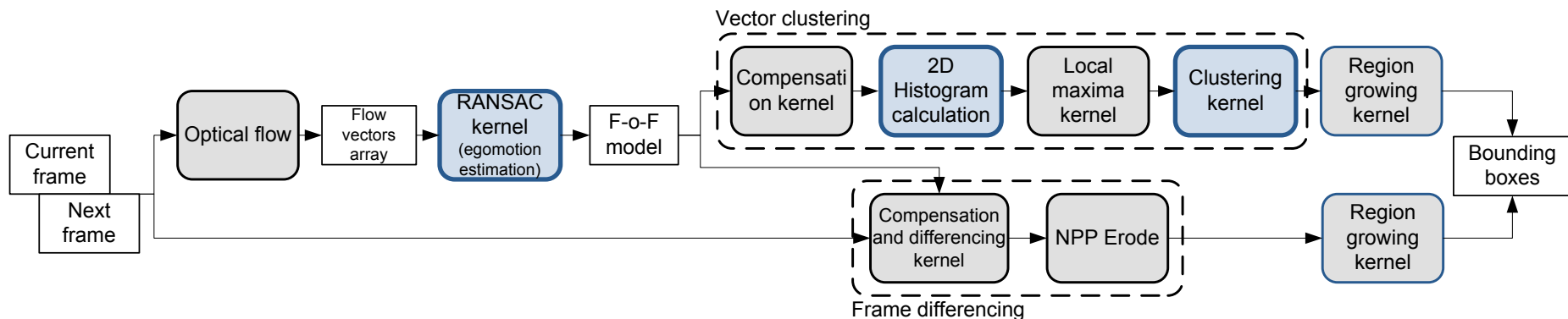
# Egomotion Compensation and Moving Objects Detection (I)

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



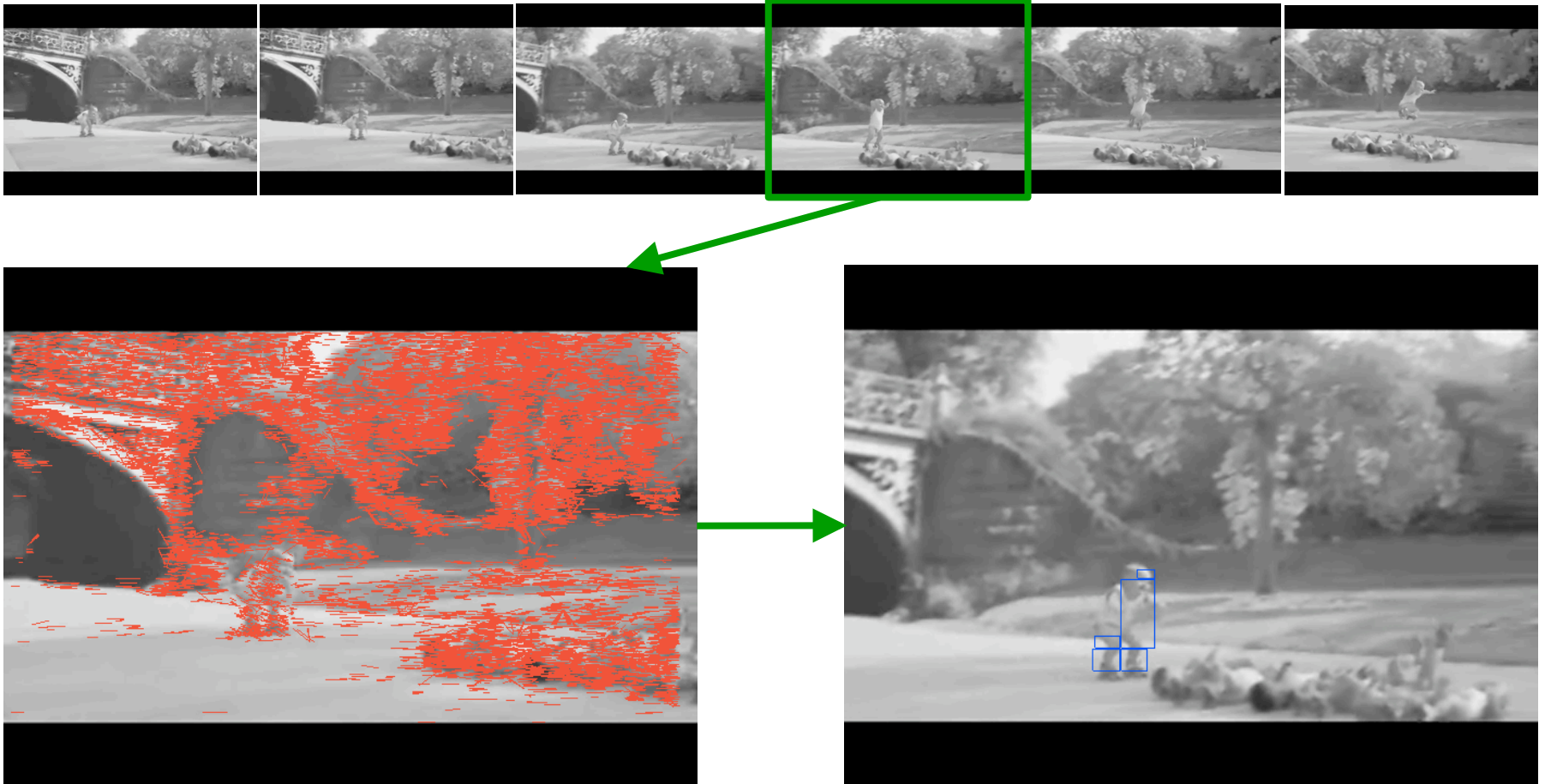
- Algorithm

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



# Egomotion Compensation and Moving Objects Detection (II)

Fast moving object in strong egomotion scenario detected by vector clustering

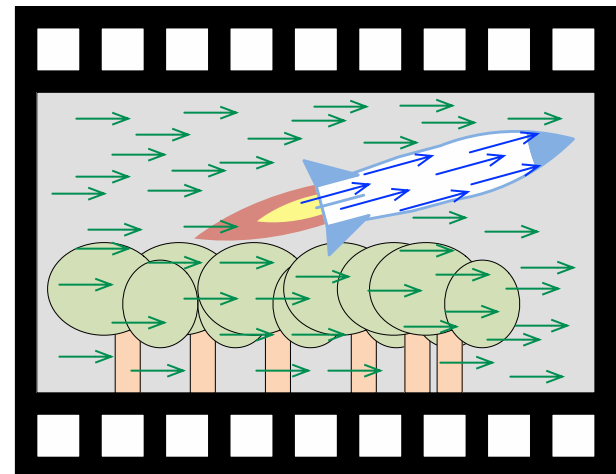


# SISD and SIMD phases

## ■ RANSAC (Fischler+, 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  
}
```

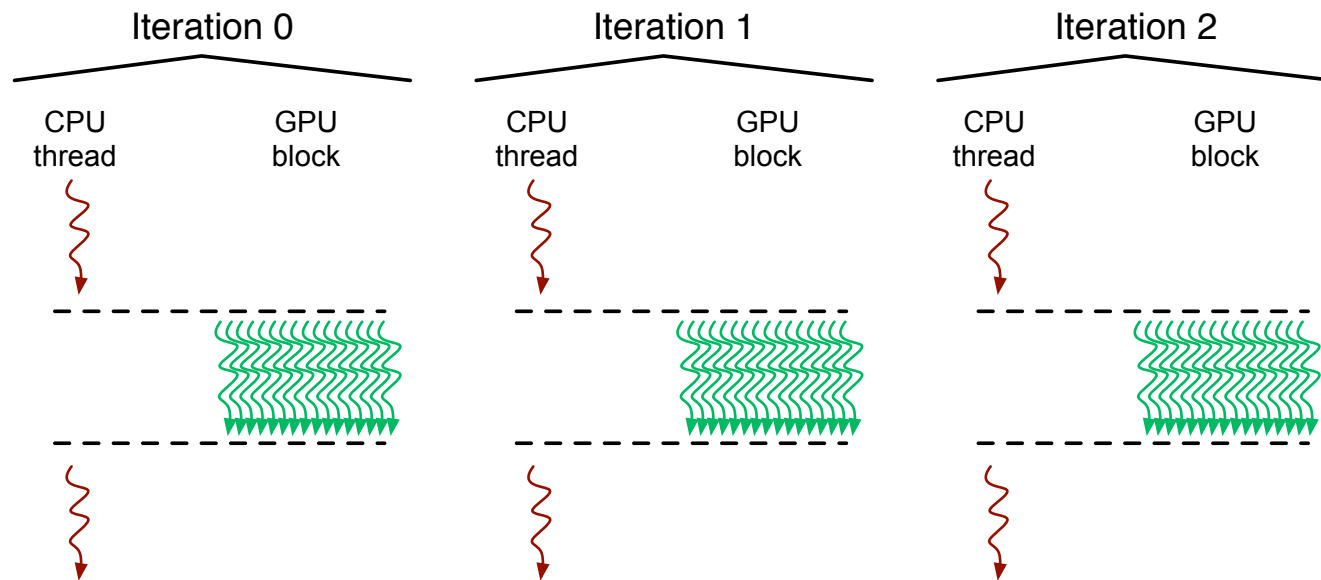
- ❑ Fitting stage picks two flow vectors randomly
- ❑ Evaluation generates motion vectors from F-o-F model, and compares them to real flow vectors





# Collaborative Implementation

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



# Chai Benchmark Suite

- Collaborative Heterogeneous Applications for Integrated architectures
- Heterogeneous execution on CPU, GPU, FPGA
- Collaboration patterns
  - 8 data partitioning benchmarks
  - 3 coarse-grain task partitioning benchmarks
  - 3 fine-grain task partitioning benchmarks
- Discrete (D) and Unified (U) versions
  - CUDA, OpenCL, and C++AMP for CPU+GPU
  - OpenCL for CPU+FPGA
  - CUDA-Sim for Gem5-GPU



<https://chai-benchmarks.github.io>



# Chai Benchmarks

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

## Versions:

- OpenCL-**U**
- OpenCL-**D**
- CUDA-**U**
- CUDA-**D**
- CUDA-**U**-Sim
- CUDA-**D**-Sim
- C++AMP

# Chai: Diversity of Benchmarks (I)

- Diversity of partitioning, usage of system-wide atomics, load balancing, and concurrency

DATA PARTITIONING

Benchmark	Partitioning Granularity	Partitioned Data	System-wide Atomics	Load Balance
BS	Fine	Output	None	Yes
CEDD	Coarse	Input, Output	None	Yes
HSTI	Fine	Input	Compute	No
HSTO	Fine	Output	None	No
PAD	Fine	Input, Output	Sync	Yes
RSCD	Medium	Output	Compute	Yes
SC	Fine	Input, Output	Sync	No
TRNS	Medium	Input, Output	Sync	No

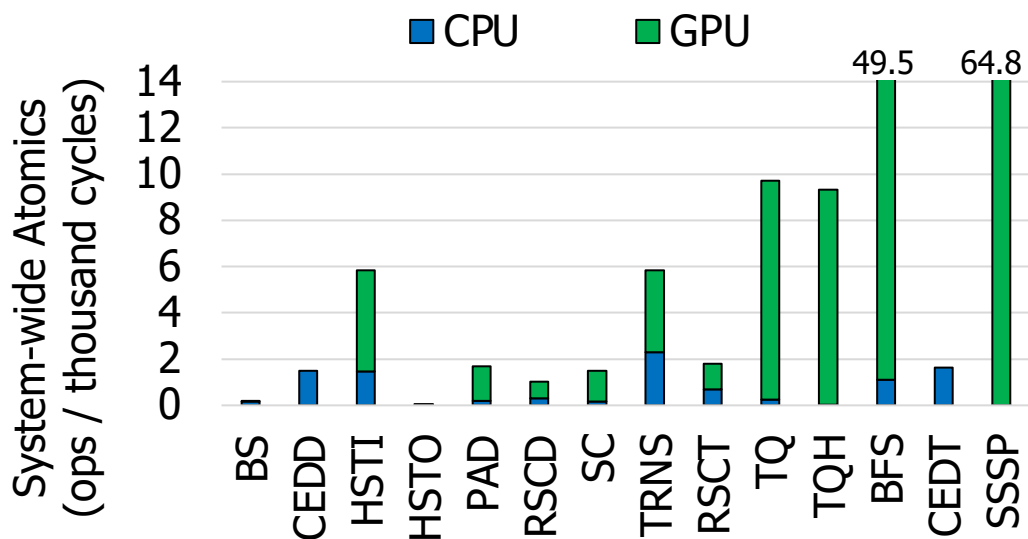
FINE-GRAIN TASK PARTITIONING

Benchmark	System-wide Atomics	Load Balance
RSCT	Sync, Compute	Yes
TQ	Sync	No
TQH	Sync	No

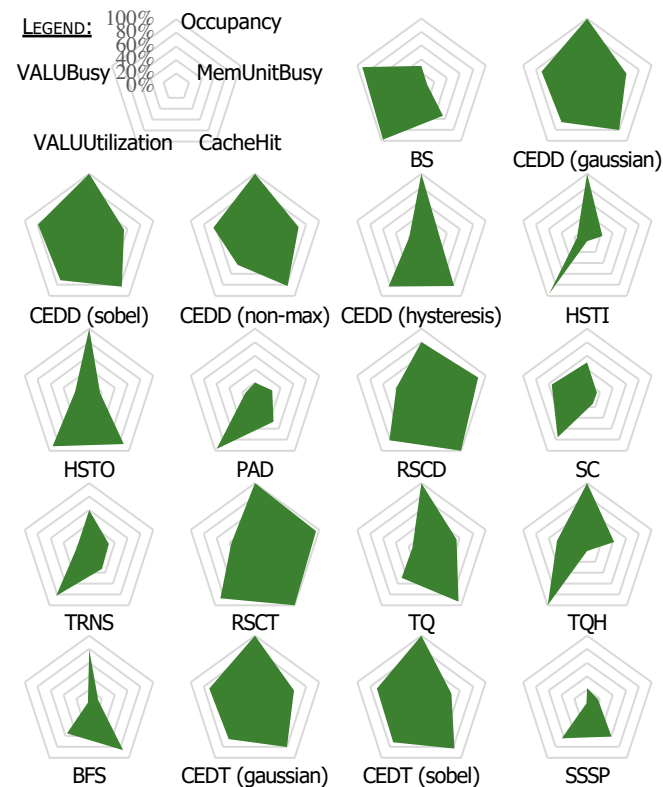
COARSE-GRAIN TASK PARTITIONING

Benchmark	System-wide Atomics	Partitioning	Concurrency
BFS	Sync, Compute	Iterative	No
CEDT	Sync	Non-iterative	Yes
SSSP	Sync, Compute	Iterative	No

# Chai: Diversity of Benchmarks (II)

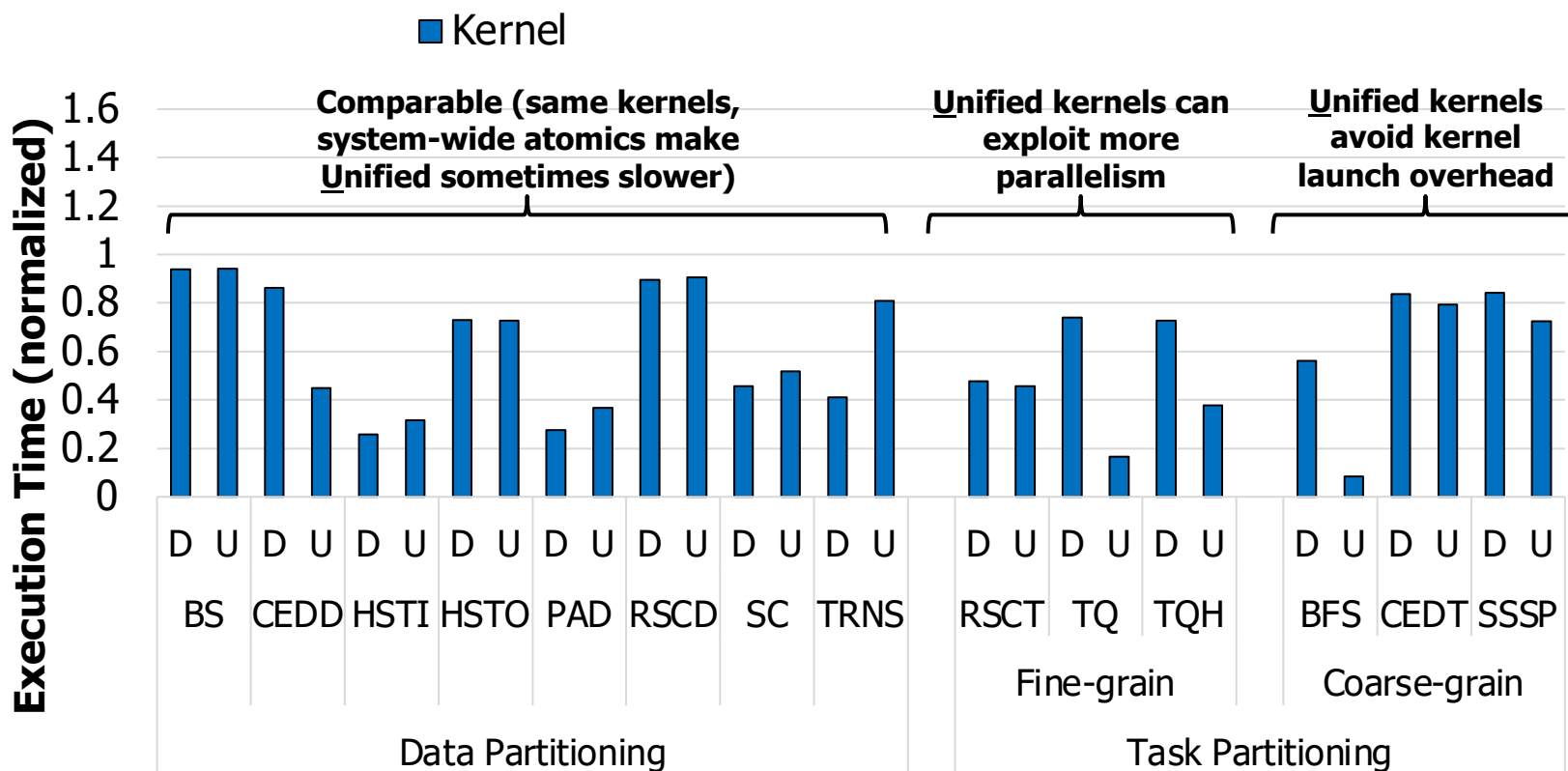


Varying intensity in use of system-wide atomics

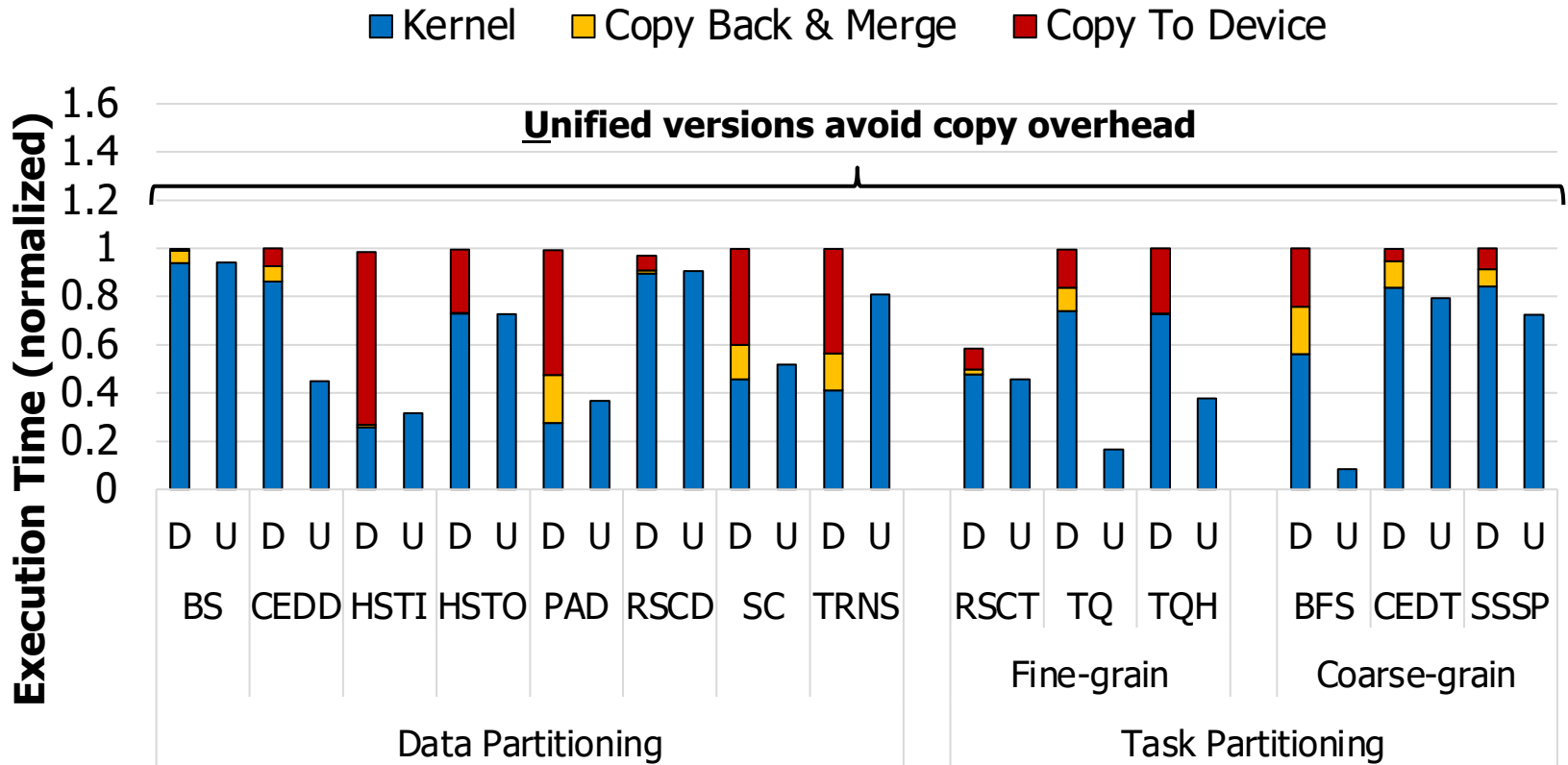


Diverse execution profiles

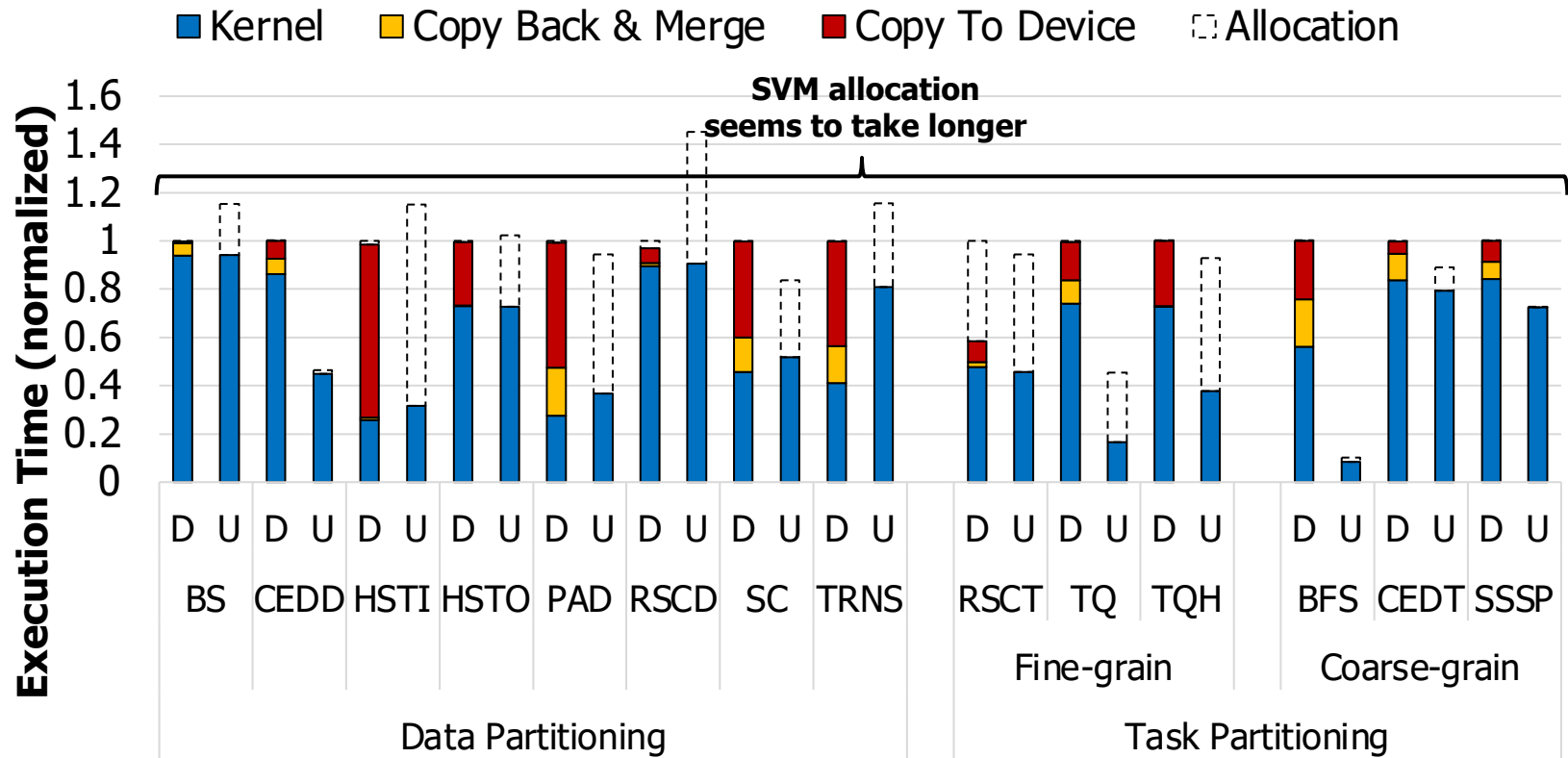
# Benefits of Unified Memory: Kernel Time



# Benefits of Unified Memory: Data Transfers

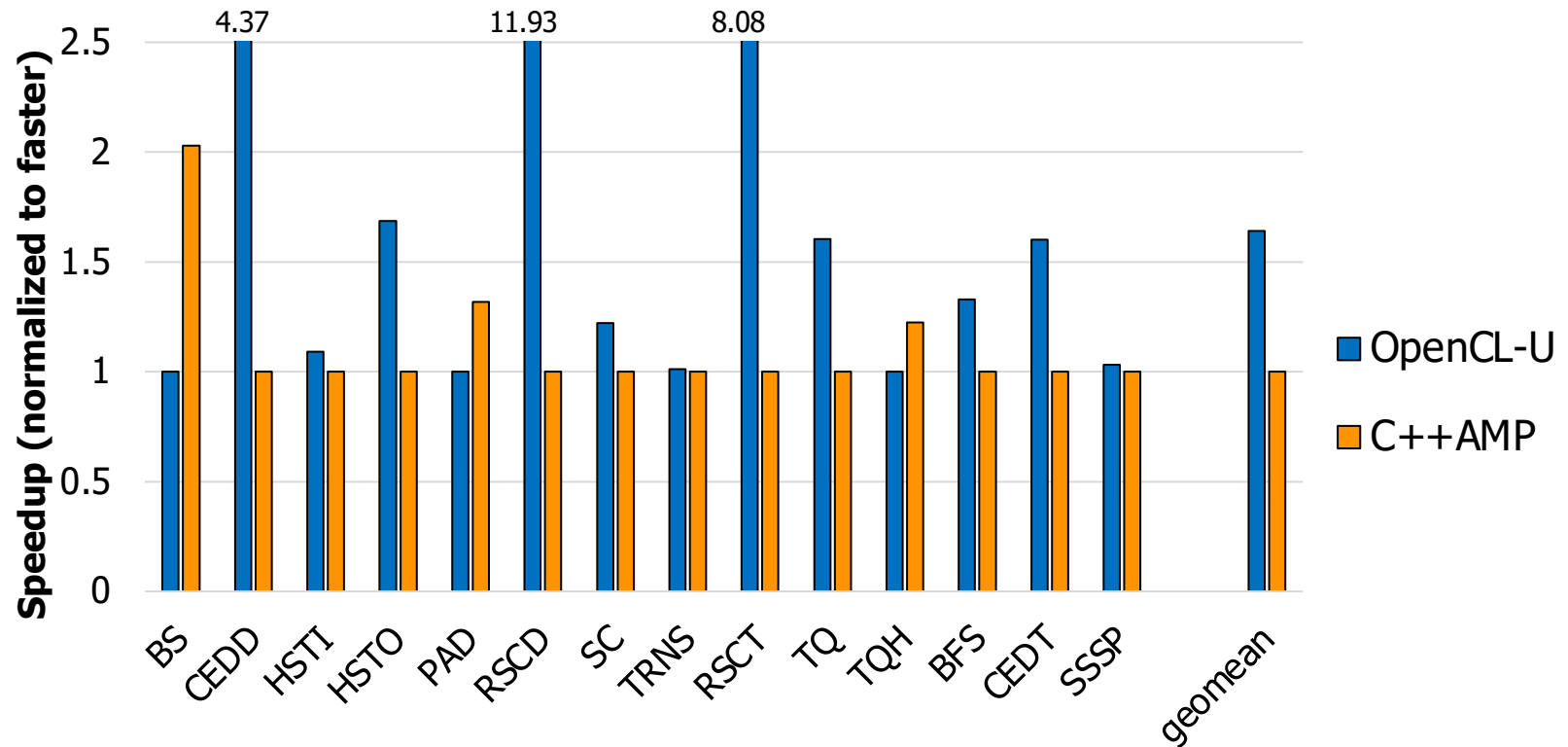


# Benefits of Unified Memory: Allocation





# Comparison C++AMP vs. OpenCL-U



# Heterogeneous System Architecture

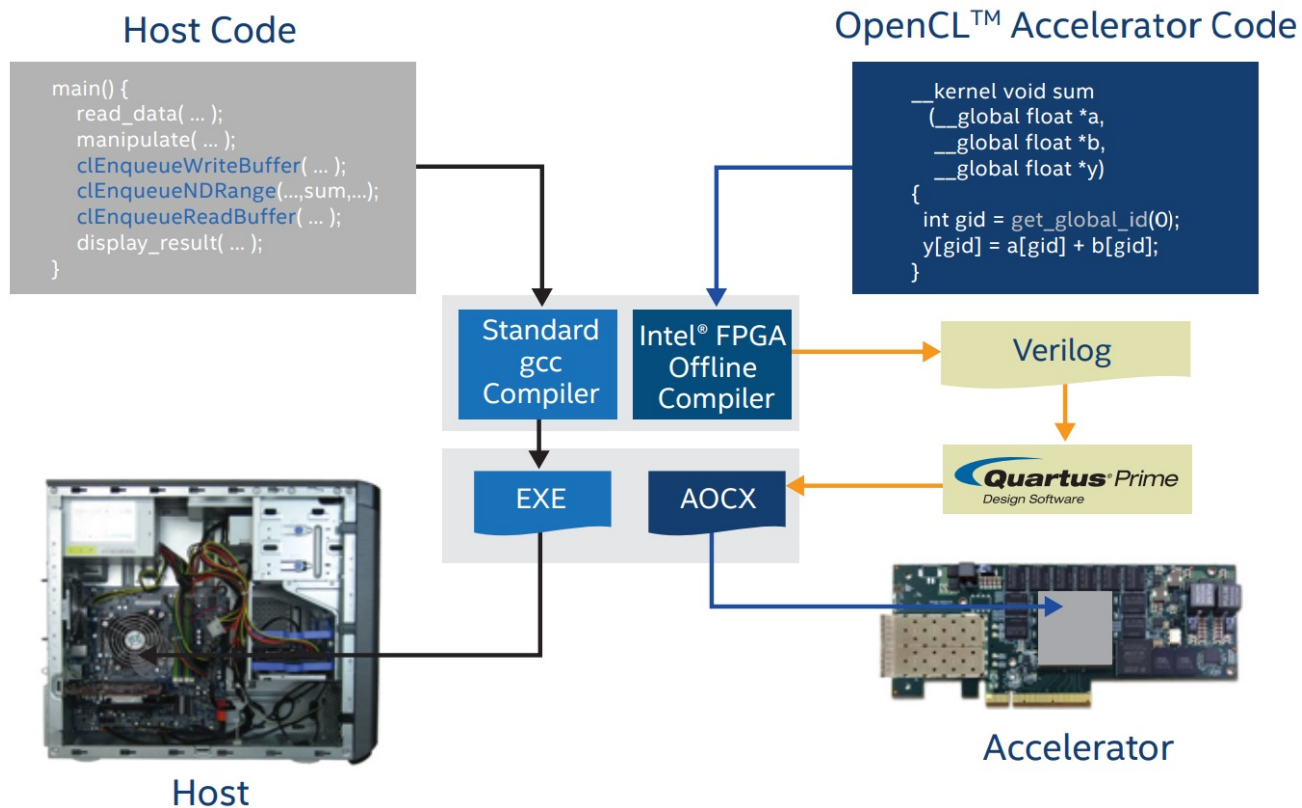
---

- Wen-mei W. Hwu (editor), “**Heterogeneous System Architecture: A New Compute Platform Infrastructure,**” 2016
  - Chapter 8 – Application use cases: Platform atomics



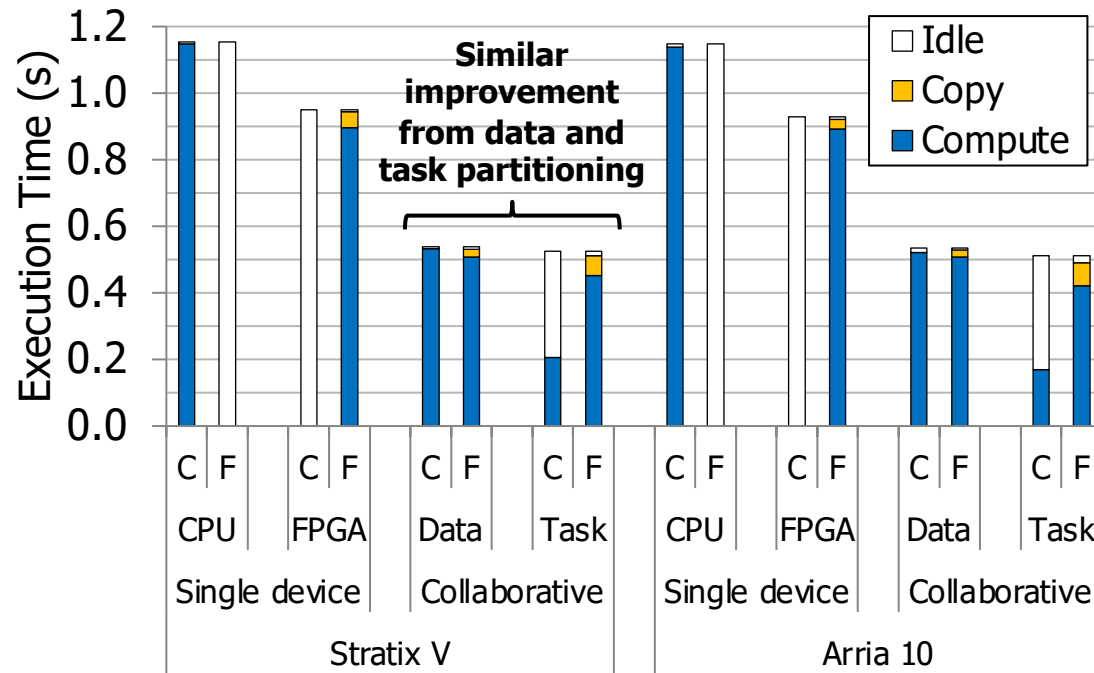
# Intel OpenCL SDK for FPGA

- Intel OpenCL SDK for FPGA is used to compile and synthesize host executable and FPGA design



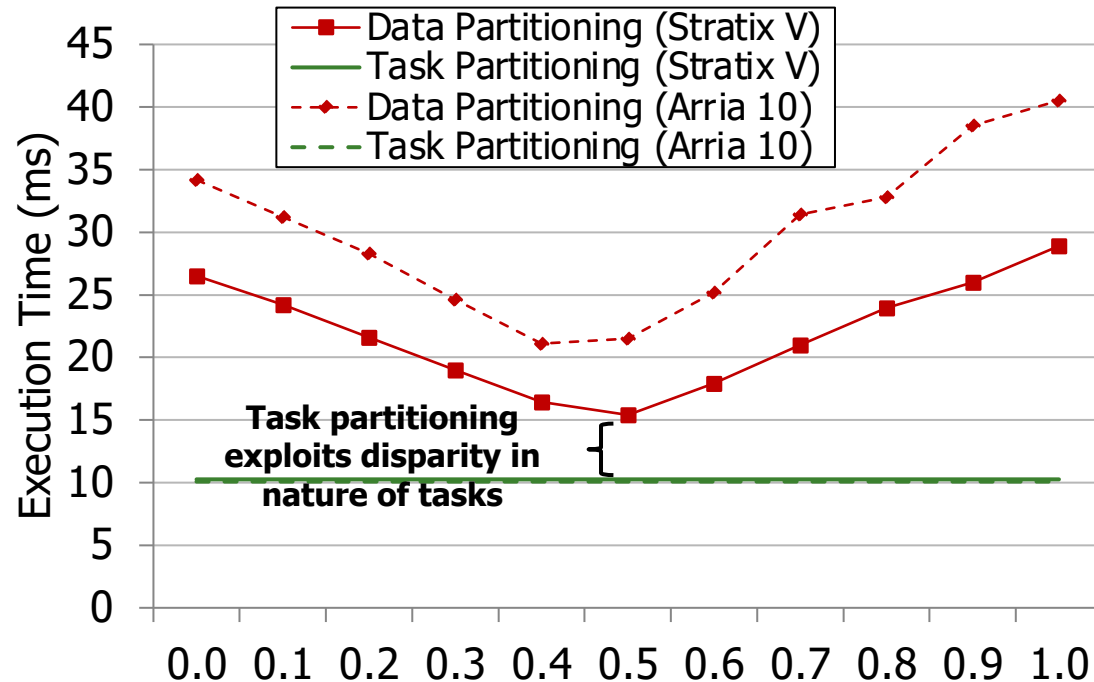
# Benefits of Collaboration on FPGA (I)

## Case Study: Canny Edge Detection



# Benefits of Collaboration on FPGA (II)

Case Study:  
Random  
Sample  
Consensus



# Chai on CPU-FPGA Systems (I)

---

- Sitao Huang, Li-Wen Chang, Izzat El Hajj, Simon Garcia De Gonzalo, Juan Gomez-Luna, Sai Rahul Chalamalasetti, Mohamed El-Hadedy, Dejan Milojicic, Onur Mutlu, Deming Chen, and Wen-mei Hwu,  
**"Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures"**  
*Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE)*, Mumbai, India, April 2019.  
[Slides (pptx)] [pdf]  
[Chai CPU-FPGA Benchmark Suite]

## Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures

Sitao Huang  
ECE, UIUC  
shuang91@illinois.edu

Li-Wen Chang\*  
Microsoft  
liwen.chang@microsoft.com

Izzat El Hajj  
ECE, UIUC  
elhajj2@illinois.edu

Simon Garcia De Gonzalo  
CS, UIUC  
grcdgnz2@illinois.edu

Juan Gómez-Luna  
CS, ETH Zurich  
juang@ethz.ch

Sai Rahul Chalamalasetti  
Hewlett Packard Labs  
sairahul.chalamalasetti@hpe.com

Mohamed El-Hadedy  
ECE, Cal Poly Pomona  
mealy@cpp.edu

Dejan Milojicic  
Hewlett Packard Labs  
dejan.milojicic@hpe.com

Onur Mutlu  
CS, ETH Zurich  
omutlu@ethz.ch

Deming Chen  
ECE, UIUC  
dchen@illinois.edu

Wen-mei Hwu  
ECE, UIUC  
w-hwu@illinois.edu

# Chai on CPU-FPGA Systems (II)

---

- Jiantong Jiang, Zeke Wang, Xue Liu, Juan Gómez-Luna, Nan Guan, Qingxu Deng, Wei Zhang, and Onur Mutlu,  
**"Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs"**  
*Proceedings of the 28th International Symposium on Field-Programmable Gate Arrays (FPGA)*, Seaside, CA, USA, February 2020.  
[[Slides \(pptx\)](#)] [[pdf](#)]

## Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs

Jiantong Jiang<sup>1★</sup>

Nan Guan<sup>3</sup>

Zeke Wang<sup>2★</sup>

Qingxu Deng<sup>1</sup>

Xue Liu<sup>1\*</sup>

Wei Zhang<sup>4</sup>

Juan Gómez-Luna<sup>2</sup>

Onur Mutlu<sup>2</sup>

<sup>1</sup> Department of Computer Science and Engineering, Northeastern University, China

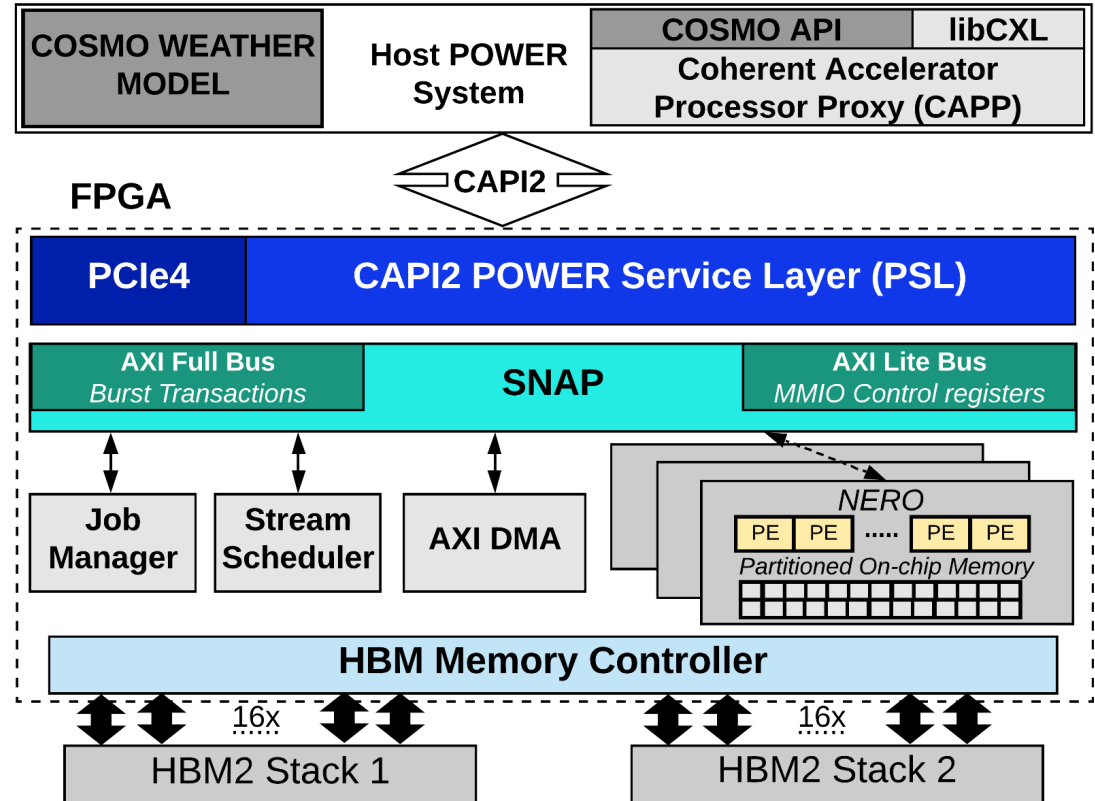
<sup>2</sup> ETH Zürich, Switzerland

<sup>3</sup> Department of Computing, Hong Kong Polytechnic University, Hong Kong

<sup>4</sup> Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong

# NERO Application Framework

- NERO communicates to Host over **CAPI2** (Coherent Accelerator Processor Interface)
- COSMO API handles offloading jobs to NERO
- SNAP (Storage, Network, and Analytics Programming) allows for seamless integration of the COSMO API



<https://github.com/open-power/snap>



# Accelerating Climate Modeling

---

- Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gómez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal,  
**"NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling"**  
*Proceedings of the 30th International Conference on Field-Programmable Logic and Applications (FPL)*, Gothenburg, Sweden, September 2020.  
[[Slides \(pptx\)](#)] [[pdf](#)]  
[[Lightning Talk Slides \(pptx\)](#)] [[pdf](#)]  
[[Talk Video](#) (23 minutes)]  
***Nominated for the Stamatis Vassiliadis Memorial Award.***

## NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling

Gagandeep Singh<sup>a,b,c</sup>    Dionysios Diamantopoulos<sup>c</sup>    Christoph Hagleitner<sup>c</sup>    Juan Gómez-Luna<sup>b</sup>  
Sander Stuijk<sup>a</sup>    Onur Mutlu<sup>b</sup>    Henk Corporaal<sup>a</sup>  
<sup>a</sup>Eindhoven University of Technology    <sup>b</sup>ETH Zürich    <sup>c</sup>IBM Research Europe, Zurich

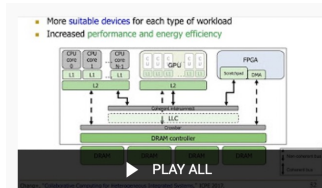
# Collaborative Computing: Key Takeaways

---

- Possibility of having **several devices collaborating** on the same workload
- Or having **the most appropriate cores** for each workload
- Easier programming with Unified Memory or Shared Virtual Memory
- CPU-GPU memory coherence and system-wide atomic operations since NVIDIA Pascal and HSA
  - Fine-grain collaboration

# Heterogeneous Systems Course (Fall 2021)

- Short weekly lectures
- Hands-on projects



## Livestream - P&S Hands-on Acceleration on Heterogeneous Computing Systems (Fall 2021)

10 videos • 566 views • Updated 6 days ago



Onur Mutlu  
Lectures

SUBSCRIBED



1

**Heterogeneous Systems Course: Meeting 1: Hands-on Acceleration on Hetero. Computing Systems (Fall21)**  
Onur Mutlu Lectures  
1:15:44

2

**Heterogeneous Systems Course: Meeting 2: SIMD processors and GPU architecture (Fall21)**  
Onur Mutlu Lectures  
1:34:00

3

**Heterogeneous Systems Course: Meeting 3: GPU Software Hierarchy (Fall 2021)**  
Onur Mutlu Lectures  
53:25

4

**Heterogeneous Systems Course: Meeting 4: GPU Memory Hierarchy (Fall 2021)**  
Onur Mutlu Lectures  
59:55

5

**Heterogeneous Systems Course: Meeting 5: GPU Performance Considerations (Fall 2021)**  
Onur Mutlu Lectures  
1:20:25

6

**Heterogeneous Systems Course: Meeting 6: Parallel Patterns: Reduction (Fall 2021)**  
Onur Mutlu Lectures  
1:15:55

7

**Heterogeneous Systems Course: Meeting 7: Parallel Patterns: Histogram (Fall 2021)**  
Onur Mutlu Lectures  
1:35:10

8

**Heterogeneous Systems Course: Meeting 8: Parallel Patterns: Convolution (Fall 2021)**  
Onur Mutlu Lectures  
1:35:10

**SAFARI Project & Seminars Courses (Fall 2021)**

Recent ChangesMedia ManagerSitemap

Trace: start • processing\_in\_memory • heterogeneous\_systems

Home

Projects

- SoftMC
- Ramulator
- Accelerating Genomics
- Mobile Genomics
- Processing-in-Memory
- Heterogeneous Systems**
- SSD Simulator

**heterogeneous\_systems**

**Hands-on Acceleration on Heterogeneous Computing Systems**

Edit

**Course Description**

The increasing difficulty of scaling the performance and efficiency of CPUs every year has created the need for turning computers into heterogeneous systems, i.e., systems composed of multiple types of processors that can suit better different types of workloads or parts of them. More than a decade ago, Graphics Processing Units (GPUs) became general-purpose parallel processors, in order to make their outstanding processing capabilities available to many workloads beyond graphics. GPUs have been critical key to the recent rise of Machine Learning and Artificial Intelligence, which took unrealistic training times before the use of GPUs. Field-Programmable Gate Arrays (FPGAs) are another example computing device that can deliver impressive benefits in terms of performance and energy efficiency. More specific examples are (1) a plethora of specialized accelerators (e.g., Tensor Processing Units for neural networks), and (2) near-data processing architectures (i.e., placing compute capabilities near or inside memory/storage).

Despite the great advances in the adoption of heterogeneous systems in recent years, there are still many challenges to tackle, for example:

- Heterogeneous implementations (using GPUs, FPGAs, TPUs) of modern applications from important fields such as bioinformatics, machine learning, graph processing, medical imaging, personalized medicine, robotics, virtual reality, etc.
- Scheduling techniques for heterogeneous systems with different general-purpose processors and accelerators, e.g., kernel offloading, memory scheduling, etc.
- Workload characterization and programming tools that enable easier and more efficient use of heterogeneous systems.

If you are enthusiastic about working **hands-on** with different software, hardware, and architecture projects for heterogeneous systems, this is your P&S. You will have the opportunity to program heterogeneous systems with different types of devices (CPUs, GPUs, FPGAs, TPUs), propose algorithmic changes to important applications to better leverage the compute power of heterogeneous systems, understand different workloads and identify the most suitable device for their execution, design optimized scheduling techniques, etc. In general, the goal will be to reach the highest performance reported for a given important application.

**Prerequisites of the course:**

- Digital Design and Computer Architecture (or equivalent course).
- Familiarity with C/C++ programming and strong coding skills.
- Interest in future computer architectures and computing paradigms.
- Interest in discovering why things do or do not work and solving problems
- Interest in making systems efficient and usable

**The course is conducted in English.**

The course has two main parts:

- Short weekly lectures on GPU and heterogeneous programming.
- Hands-on project: Each student develops his/her own project.

**Table of Contents**


- Hands-on Acceleration on Heterogeneous Computing Systems
- Course Description
- Mentors
- Lecture Video Playlist on YouTube
- Fall 2021 Meetings/Schedule
- Learning Materials
- Assignments

[https://youtube.com/playlist?list=PL5Q2soXY2Zi\\_OwkTqEyA6tk3UsoPBH737](https://youtube.com/playlist?list=PL5Q2soXY2Zi_OwkTqEyA6tk3UsoPBH737)

[https://safari.ethz.ch/projects\\_and\\_seminars/fall2021/doku.php?id=heterogeneous\\_systems](https://safari.ethz.ch/projects_and_seminars/fall2021/doku.php?id=heterogeneous_systems)

# Processing-in-Memory Course (Fall 2021)

- Short weekly lectures
- Hands-on projects

SAFARI Project & Seminars Courses (Fall 2021)

Search

Recent ChangesMedia ManagerSitemap

Trace: • heterogeneous\_systems • processing\_in\_memory

Home

Projects

- SoftMC
- Ramulator
- Accelerating Genomics
- Mobile Genomics

processing\_in\_memory

### Exploring the Processing-in-Memory Paradigm for Future Computing Systems

Edit

#### Course Description

Data movement between the memory units and the compute units of current computing systems is a major performance and energy bottleneck. From large-scale servers to mobile devices, data movement costs dominate computation costs in terms of both performance and energy consumption. For example, data movement between the main memory and the processing cores accounts for 62% of the total system energy in consumer applications. As a result, the data movement bottleneck is a huge burden that greatly limits the energy efficiency and performance of modern computing systems. This phenomenon is an undesired effect of the dichotomy between memory and the processor, which leads to the data movement bottleneck.

Many modern and important workloads such as machine learning, computational biology, graph processing, databases, video analytics, and real-time data analytics suffer greatly from the data movement bottleneck. These workloads are exemplified by irregular memory accesses, relatively low data reuse, low cache line utilization, low arithmetic intensity (i.e., ratio of operations per accessed byte), and large datasets that greatly exceed the main memory size. The computation in these workloads cannot usually compensate for the data movement costs. In order to alleviate this data movement bottleneck, we need a paradigm shift from the traditional processor-centric design, where all computation takes place in the compute units, to a more data-centric design where processing elements are placed closer to or inside where the data resides. This paradigm of computing is known as Processing-in-Memory (PIM).

This is your perfect P&S if you want to become familiar with the main PIM technologies, which represent "the next big thing" in Computer Architecture. You will work hands-on with the first real-world PIM architecture, will explore different PIM architecture designs for important workloads, and will develop tools to enable research of future PIM systems. Projects in this course span software and hardware as well as the software/hardware interface. You can potentially work on developing and optimizing new workloads for the first real-world PIM hardware or explore new PIM designs in simulators, or do something else that can forward our understanding of the PIM paradigm.

#### Prerequisites of the course:

- Digital Design and Computer Architecture (or equivalent course).
- Familiarity with C/C++ programming.
- Interest in future computer architectures and computing paradigms.
- Interest in discovering why things do or do not work and solving problems
- Interest in making systems efficient and usable

#### The course is conducted in English.

The course has two main parts:

- Short lectures on different aspects of processing-in-memory.
- Hands-on project: Each student develops his/her own project.

#### Table of Contents

- Exploring the Processing-in-Memory Paradigm for Future Computing Systems
- Course Description
- Mentors
- Lecture Video Playlist on YouTube
- Fall 2021 Meetings/Schedule
- Learning Materials
- Assignments

### A Modern Primer on Processing in Memory

Onur Mutlu<sup>a,b</sup>, Sangmin Cho<sup>a,c</sup>, Juan Gómez-Liñán<sup>a</sup>, Rachata Assavanigatana<sup>a</sup>  
SAFARI Research Group  
aETH Zurich  
bComputer Science University  
cKing Mongkut's University of Technology North Bangkok

PLAY ALL

### Processing in Memory Course: Meeting 1: Exploring the PIM Paradigm for Future Systems - Fall'21

Onur Mutlu Lectures

1:31:01

### Processing in Memory Course: Meeting 2: Real-world PIM architectures - Fall'21

Onur Mutlu Lectures

47:35

### Processing in Memory Course: Meeting 3: Real-world PIM architectures II - Fall'21

Onur Mutlu Lectures

1:00:41

### Processing in Memory Course: Meeting 4: Real-world PIM architectures III - Fall'21

Onur Mutlu Lectures

1:08:36

### Processing in Memory Course: Meeting 5: Real-world PIM architectures IV - Fall'21

Onur Mutlu Lectures

59:05

### Processing in Memory Course: Meeting 6: End-to-end Framework for Processing-using-Memory - Fall'21

Onur Mutlu Lectures

1:33:40

### Processing in Memory Course: Meeting 7: How to Evaluate Data Movement Bottlenecks - Fall'21

Onur Mutlu Lectures

1:22:10

### Processing in Memory Course: Meeting 8: Programming PIM Architectures - Fall'21

Onur Mutlu Lectures

1:07:41

### Processing in Memory Course: Meeting 9: Benchmarking and Workload

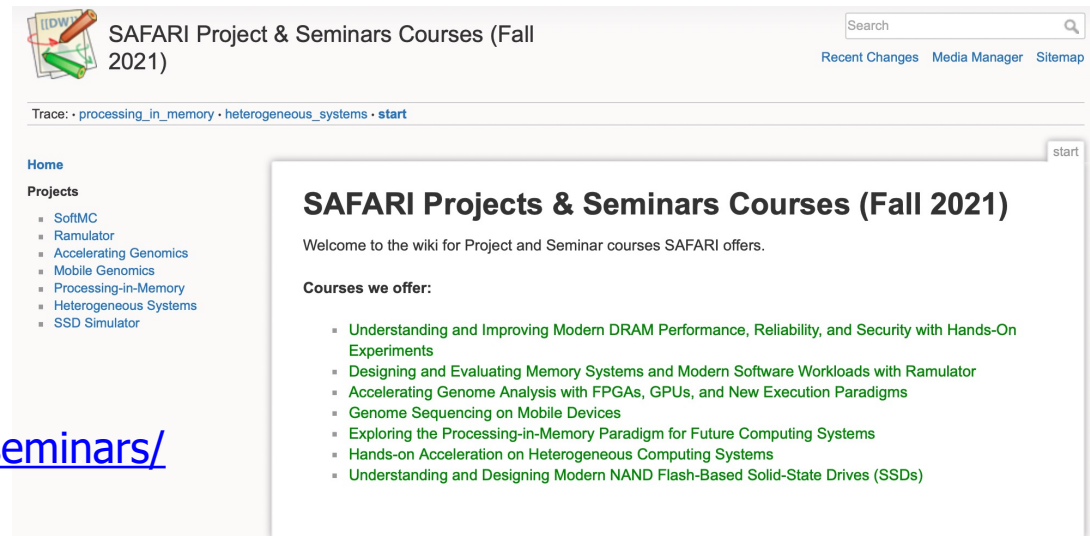
<https://youtube.com/playlist?list=PL5Q2soXY2Zi-841fUYUYK9EsXKhQKRPyX>

[https://safari.ethz.ch/projects\\_and\\_seminars/fall2021/doku.php?id=processing\\_in\\_memory](https://safari.ethz.ch/projects_and_seminars/fall2021/doku.php?id=processing_in_memory)

# More P&S Courses: SSDs, Memory, Bioinformatics

- Understanding and Improving Modern DRAM Performance, Reliability, and Security with Hands-On Experiments
- Designing and Evaluating Memory Systems and Modern Software Workloads with Ramulator
- Accelerating Genome Analysis with FPGAs, GPUs, and New Execution Paradigms
- Genome Sequencing on Mobile Devices
- Understanding and Designing Modern NAND Flash-Based Solid-State Drives (SSDs)

[https://safari.ethz.ch/projects\\_and\\_seminars/fall2021/doku.php?id=start](https://safari.ethz.ch/projects_and_seminars/fall2021/doku.php?id=start)



SAFARI Project & Seminars Courses (Fall 2021)

Trace: · processing\_in\_memory · heterogeneous\_systems · start

Home

Projects

- SoftMC
- Ramulator
- Accelerating Genomics
- Mobile Genomics
- Processing-in-Memory
- Heterogeneous Systems
- SSD Simulator

**SAFARI Projects & Seminars Courses (Fall 2021)**

Welcome to the wiki for Project and Seminar courses SAFARI offers.

**Courses we offer:**

- Understanding and Improving Modern DRAM Performance, Reliability, and Security with Hands-On Experiments
- Designing and Evaluating Memory Systems and Modern Software Workloads with Ramulator
- Accelerating Genome Analysis with FPGAs, GPUs, and New Execution Paradigms
- Genome Sequencing on Mobile Devices
- Exploring the Processing-in-Memory Paradigm for Future Computing Systems
- Hands-on Acceleration on Heterogeneous Computing Systems
- Understanding and Designing Modern NAND Flash-Based Solid-State Drives (SSDs)



# More Resources: Onur Mutlu Lectures

- All P&S courses
- Digital Design and CompArch course
- Advanced CompArch course
- Seminar in CompArch

The screenshot displays the YouTube channel page for Onur Mutlu Lectures, which has 21.5K subscribers. The page is organized into several sections of playlists:

- Created playlists:** This section features a horizontal row of six playlist thumbnails. From left to right, they are: 'Livestream - P&S Ramulator (Fall 2021)', 'Livestream - P&S Hands-on Acceleration on Heterogeneous...', 'Livestream - P&S Exploring the Processing-in-Memory Paradig...', 'Livestream - P&S Accelerating Genome Analysis with FPGAs...', 'Livestream - P&S Genome Sequencing on Mobile Devices...', and 'P&S Modern SSDs (Fall 2021)'. Each thumbnail includes a video preview and a 'VIEW FULL PLAYLIST' link.
- First Course in Computer Architecture & Digital Design 2021-2013:** This section contains a horizontal row of six playlist thumbnails. From left to right, they are: 'Livestream - Digital Design and Computer Architecture - ETH...', 'Digital Design & Computer Architecture - ETH Zürich (Spri...', 'Design of Digital Circuits - ETH Zürich - Spring 2019', 'Design of Digital Circuits - ETH Zürich - Spring 2018', 'Digital Circuits and Computer Architecture - ETH Zurich - ...', and 'Spring 2015 - Computer Architecture Lectures - Carneg...'. Each thumbnail includes a video preview and a 'VIEW FULL PLAYLIST' link.
- Advanced Computer Architecture Courses 2021-2012:** This section contains a horizontal row of six playlist thumbnails. From left to right, they are: 'Livestream - Computer...', 'Computer Architecture - ETH...', 'Computer Architecture - ETH...', 'Computer Architecture - ETH...', 'Computer Architecture - ETH...', and 'Fall 2015 - 740 Computer...'. Each thumbnail includes a video preview and a 'VIEW FULL PLAYLIST' link.

# P&S Heterogeneous Systems

## Collaborative Computing

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2021

6 January 2022