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, ...);
- Kernel – __global__ void kernel(type args,...)
  - Automatic variables transparently assigned to registers
  - Shared memory – __shared__
  - Intra-block synchronization – __syncthreads();
Review: CUDA Programming Language

- **Memory allocation**
  
  ```c
cudaMalloc((void**)&d_in, #bytes);
  ```

- **Memory copy**
  
  ```c
cudaMemcpy(d_in, h_in, #bytes,
             cudaMemcpyHostToDevice);
  ```

- **Kernel launch**
  
  ```c
  kernel<<< #blocks, #threads >>>(args);
  ```

- **Memory deallocation**
  
  ```c
cudaFree(d_in);
  ```

- **Explicit synchronization**
  
  ```c
cudaDeviceSynchronize();
  ```
Review: Indexing and Memory Access

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

\[
\text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} = 6 \times 4 + 1 = 25
\]
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)

Array of Structures (AoS)

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

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

- Same memory locations accessed by neighboring threads

```c
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

```c
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
...
Load tile into shared memory
__syncthreads();
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];
    }
}
```
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**

```c
Compute(threadIdx.x);
if (threadIdx.x % 2 == 0){
    Do_this(threadIdx.x);
}
else{
    Do_that(threadIdx.x);
}
```
Review: SIMD Utilization

- Intra-warp divergence

```c
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

No atomic conflict = concurrent updates

Atomic conflict = serialized updates
Histogram Calculation

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

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

![Diagram showing data distribution across threads and histogram bins](image)

- Atomic additions
Histogram Calculation

- **Frequent conflicts** in natural images
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

![Diagram showing data transfer times: $t_T$ for copy data and $t_E$ for execution.](image-url)
Asynchronous Transfers

- **Computation** divided into \( n \text{Streams} \)
  - \( D \) input data instances, \( B \) blocks
  - \( n \text{Streams} \)
    - \( D/n \text{Streams} \) data instances
    - \( B/n \text{Streams} \) blocks

- **Estimates**
  \[
  t_E + \frac{t_T}{n \text{Streams}}
  \]
  \( t_E \geq t_T \) (dominant kernel)

  \[
  t_T + \frac{t_E}{n \text{Streams}}
  \]
  \( t_T > t_E \) (dominant transfers)
Asynchronous Transfers

- **Overlap of communication and computation** (e.g., 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
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
Review

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

```c
// 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

CUDA 6 Unified Memory

- Kepler GPU
- CPU
- Unified Memory

(Limited to GPU Memory Size)

Pascal Unified Memory

- Pascal GPU
- CPU
- Unified Memory

(Limited to System Memory Size)
Unified Memory

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

```c
// 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();
```
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

```c
// 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

```c
// 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

  \[
  \text{cudaMallocManaged}(&h\text{\_in},\ \text{in}\text{\_size});
  \]

- System-wide atomics

  \[
  \text{old} = \text{atomicAdd\_system}(&h\text{\_out}[x],\ \text{inc});
  \]
OpenCL 2.0

- **Shared virtual memory**
  
  ```c
  XYZ * h_in = (XYZ *)clSVMAlloc(    
      ocl.clContext, CL_MEM_SVM_FINE_GRAIN_BUFFER, in_size, 0);
  ```

- **More flags:**
  
  ```c
  CL_MEM_READ_WRITE
  CL_MEM_SVM_ATOMICS
  ```

- **C++11 atomic operations**
  
  ```c
  (memory_scope_all_svm_devices)
  old = atomic_fetch_add(&h_out[x], inc);
  ```
Unified memory space (HSA)

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

C++11 atomic operations

```c
(memory_scope_all_svm_devices)
```

Platform atomics (HSA)

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

Program Structure

- Sequential sub-tasks
- Data-parallel tasks
- Coarse-grained synchronization

Data Partitioning

Device 1

Device 2
Collaborative Patterns

Program Structure

data-parallel tasks

sequential sub-tasks

coarse-grained synchronization

Device 1

Device 2

Coarse-grained Task Partitioning
Collaborative Patterns

Program Structure

data-parallel tasks

coarse-grained synchronization

Fine-grained Task Partitioning
Histogram

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

```c
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
```
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 \( m = n = 3 \)

\[
S(u, v) = \sum_{i=0}^{m} \sum_{j=0}^{n} P_{i,j} B_{i,m}(u) B_{j,n}(v), \quad (1)
\]

\[
B_{i,m}(u) = \binom{m}{i} (1 - u)^{m-i} u^i, \quad (2)
\]
Bézier Surfaces

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

- **Without Unified Memory**

```c
// 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**
  - Bézier surface: 300x300, 4x4 control points
  - %Tiles to CPU
  - NVIDIA Jetson TX1 (4 ARMv8 + 2 SMX): 17% speedup wrt GPU only

![Graph showing execution time vs. %Tiles to CPU](Image)
Bézier Surfaces

- **With Unified Memory (Pascal/Volta)**

```c
// 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**

(a) Static Distribution

(b) Dynamic Distribution

**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)

![Diagram showing execution time for different configurations and Bézier Surfaces with up to 47% improvement over GPU only.](image-url)
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
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

Predicate: Element > 0

Input

```
2 1 3 0 0 1 3 4 0 0 2 1
```

Output

```
2 1 3 1 3 4 2 1
```
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)

![Graph showing execution time for Stream Compaction with and without data partitioning. The graph indicates an 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 \((\text{frontier}\_\text{size} / \text{block}\_\text{size})\) blocks
  - Persistent blocks: up to \((\text{number}\_\text{SMs} \times \text{max}\_\text{blocks}\_\text{SM})\)
Atomic-Based Block Synchronization

- Code (simplified)

```c
// 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

```c
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

- Small frontiers processed on CPU
- Large frontiers processed on GPU
Choose **CPU or GPU depending on frontier size**

```cpp
// 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

![Normalized execution time comparison graph](image-url)
Collaborative Implementation

- **Without** Unified Memory
  - Explicit memory copies

```c
// 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

```c
// 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

![Graph showing Execution Time (ms) for Single Source Shortest Path, with NE, NY, UT, and 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

![Diagram of the motion detection algorithm](image-url)
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: \textbf{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 Benchmark Suite

<table>
<thead>
<tr>
<th>Collaboration Pattern</th>
<th>Short Name</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Partitioning</td>
<td>BS</td>
<td>Bézier Surface</td>
</tr>
<tr>
<td></td>
<td>CEDD</td>
<td>Canny Edge Detection</td>
</tr>
<tr>
<td></td>
<td>HSTI</td>
<td>Image Histogram (Input Partitioning)</td>
</tr>
<tr>
<td></td>
<td>HSTO</td>
<td>Image Histogram (Output Partitioning)</td>
</tr>
<tr>
<td></td>
<td>PAD</td>
<td>Padding</td>
</tr>
<tr>
<td></td>
<td>RSCD</td>
<td>Random Sample Consensus</td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td>Stream Compaction</td>
</tr>
<tr>
<td></td>
<td>TRNS</td>
<td>In-place Transposition</td>
</tr>
<tr>
<td>Task Partitioning</td>
<td>RSCT</td>
<td>Random Sample Consensus</td>
</tr>
<tr>
<td></td>
<td>TQ</td>
<td>Task Queue System (Synthetic)</td>
</tr>
<tr>
<td></td>
<td>TQH</td>
<td>Task Queue System (Histogram)</td>
</tr>
<tr>
<td>Coarse-grain</td>
<td>BFS</td>
<td>Breadth-First Search</td>
</tr>
<tr>
<td></td>
<td>CEDT</td>
<td>Canny Edge Detection</td>
</tr>
<tr>
<td></td>
<td>SSSP</td>
<td>Single-Source Shortest Path</td>
</tr>
</tbody>
</table>
**Benefits of Unified Memory**

- **Comparable (same kernels, system-wide atomics make Unified sometimes slower)**
- **Unified kernels can exploit more parallelism**
- **Unified kernels avoid kernel launch overhead**

The chart shows execution time (normalized) for various kernels and data/task partitioning methods. The kernels are labeled as follows:
- BS
- CEDD
- HSTI
- HSTO
- PAD
- RSCD
- SC
- TRNS
- RSCT
- TQ
- TQH
- BFS
- CEDT
- SSSP

The partitioning methods are divided into data and task partitioning:
- **Data Partitioning**: Fine-grain
- **Task Partitioning**: Coarse-grain

The chart indicates that unified kernels can sometimes be slower due to system-wide atomics, but they exploit more parallelism and avoid kernel launch overhead.
Benefits of Unified Memory

Unified versions avoid copy overhead

Execution Time (normalized)

- Kernel
- Copy Back & Merge
- Copy To Device

Data Partitioning
- BS
- CEDHSTI
- HSTO
- PAD
- RSCD
- SC
- TRNS

Task Partitioning
- RSCT
- TQ
- TQH
- BFS
- CEDTSSSP

Fine-grain
Coarse-grain
Benefits of Unified Memory

The chart shows execution times for different tasks, normalized across different partitioning strategies: data partitioning and task partitioning. The chart highlights the execution time (normalized) for Kernel, Copy Back & Merge, Copy To Device, and SVM allocation. SVM allocation seems to take longer.

Data Partitioning: BS, CEDD, HSTI, HSTO, PAD, RSCD, SC, TRNS
Task Partitioning: RSCT, TQ, TQH, BFS, CEDT, SSSP

Coarse-grain and Fine-grain tasks are differentiated in the chart.
Benefits of Collaboration on FPGA

Case Study: Canny Edge Detection

Source: 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