In Our Previous Lecture...
Dynamic Parallelism

- GPU programming frameworks provide an interface to express **dynamic refinement algorithms** in a more natural way
  - This dynamic parallelism interface allows GPU threads to launch GPU kernels when new work is dynamically discovered
  - 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
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

A Recursive Example: Quadtree (V)

- 1 thread block is launched from host

Outline of recursive kernel

Assign block to node

points > min_points && depth < max_depth

Y

Compute center of bounding box

Count points in children

Scan for offsets

Reorder points

Launch 4 children

Depth = 0

Buffer 0: a b c d e f g h i j k l m n o p q r s t u

Buffer 1: b c e f g j i n o a d h i k l p q r s t u

https://youtu.be/X74BLPO8tT4
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

```c
// 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

Kernel Arrangement

- Creating global barriers needs frequent kernel launches
- Too much overhead
- Solutions:
  - Partially use GPU-synchronization
  - Multi-layer Kernel Arrangement
  - Dynamic Parallelism
  - Persistent threads with global barriers

https://youtu.be/cvP1aH31ar4
NVIDIA Grace Hopper Superchip

- CPU + GPU
  - Grace CPU + Hopper GPU
- 900 GB/s coherent interface (7x faster than PCIe Gen 5)

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

```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 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

- Kepler GPU
- CPU
- Unified Memory
  (Limited to GPU Memory Size)

Pascal Unified Memory

- Pascal GPU
- CPU
- Unified Memory
  (Limited to System Memory Size)
Heterogeneous System Architecture

- HSA extends the unified memory space beyond GPUs
  - DSPs, DMA engines, cryptoengines, and other accelerators

Legacy GPU compute on discrete GPU cards

Legacy GPU compute on SOCs

An HSA enabled SOC featuring multiple processors beyond CPU

Unified Memory: Memory Management

- 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();
```

- 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

<table>
<thead>
<tr>
<th>Experiment</th>
<th>No Atomics</th>
<th>Atomics for Computation</th>
<th>Atomics for Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS</td>
<td>D</td>
<td>U</td>
<td>D</td>
</tr>
<tr>
<td>HSTO</td>
<td>D</td>
<td>U</td>
<td>D</td>
</tr>
<tr>
<td>HSTI</td>
<td>D</td>
<td>U</td>
<td>D</td>
</tr>
<tr>
<td>RSCD</td>
<td>D</td>
<td>U</td>
<td>D</td>
</tr>
<tr>
<td>PAD</td>
<td>U</td>
<td>D</td>
<td>U</td>
</tr>
<tr>
<td>SC</td>
<td>D</td>
<td>U</td>
<td>D</td>
</tr>
<tr>
<td>TRNS</td>
<td>D</td>
<td>U</td>
<td>D</td>
</tr>
</tbody>
</table>

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?
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 Collaboration

- Fine-grained collaboration becomes possible with unified memory (post Kepler/Maxwell architecture)
- Pascal/Volta/Turing/Ampere Unified Memory (& HSA)
  - 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 and Later

- Unified memory
  
  ```c
  cudaMallocManaged(&h_in, in_size);
  ```

- System-wide atomics
  
  ```c
  old = atomicAdd_system(&h_out[x], inc);
  ```
OpenCL 2.0 and Later

- **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

(memory_scope_all_svm_devices)

- Platform atomics (HSA)

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

Program Structure

data-parallel tasks

sequential sub-tasks

coarse-grained synchronization

Chang+, "Collaborative Computing for Heterogeneous Integrated Systems," ICPE 2017
Collaborative Patterns: Data Partitioning

Program Structure

Data Partitioning

Chang+, "Collaborative Computing for Heterogeneous Integrated Systems," ICPE 2017
Collaborative Patterns: Task Partitioning (I)

Program Structure

Coarse-grained Task Partitioning

Chang+, "Collaborative Computing for Heterogeneous Integrated Systems," ICPE 2017
Collaborative Patterns: Task Partitioning (II)

Program Structure

Data-parallel tasks

Sequential sub-tasks

Coarse-grained synchronization

Chang+, "Collaborative Computing for Heterogeneous Integrated Systems," ICPE 2017
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/tasks executed in Device 1 and Device 2

Huang+, "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures," ICPE 2019
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

The total execution time is

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

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

---

Huang+, "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures," ICPE 2019
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)
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 tasks executed in Device 1 and Device 2

Coarse-grained task partitioning

The total execution time is

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

Huang+, "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures," ICPE 2019
Data Partitioning
Traditional approach: **Separate CPU and GPU histograms are merged at the end**

```c
// Allocating memory
malloc(CPU image);
cudaMalloc(GPU image);
cudaMemcpy(GPU image, CPU image, ..., HostToDevice);
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
```
Traditional approach: Separate CPU and GPU histograms are merged at the end

```c
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**

```c
malloc(CPU image);
cudaMallocManaged(GPU image);
memcpy(GPU image, CPU image, ...);
cudaMallocManaged(Histogram);
cudaMemset(Histogram, 0);
```

```c
// 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 $m = n = 3$

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

(1)

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

(2)
Bézier Surfaces: Static Distribution (I)

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

Palomar+, "High-Performance Computation of Bézier Surfaces on Parallel and Heterogeneous Platforms," IJPP, 2018
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: Static Distribution (III)**

- **Performance results** on NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)
  - Bézier surface: 300x300, 4x4 control points
  - %Tiles to CPU
  - 17% speedup over GPU only
Bézier Surfaces with Unified Memory

**With 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
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**

  (a) Static Distribution
  (b) Dynamic Distribution

- **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

      ... // Kernel body
  }
  ```

- Bézier Surfaces: Dynamic Distribution

  Palomar+, "High-Performance Computation of Bézier Surfaces on Parallel and Heterogeneous Platforms," IJPP, 2018
Benefits of Collaboration: Bézier Surfaces

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

![Graph showing execution time improvement across different configurations.](image)

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

Gómez-Luna+, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," ISPASS 2017
Matrix padding

- Use cases:
  - Memory alignment
  - Transposition of near-square matrices
  - Etc.

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

![Graph showing execution time (ms) vs %CPU workload](image)
In-Place Padding

- Pascal/Volta/Turing/Ampere Unified Memory

Coherent memory

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td></td>
</tr>
</tbody>
</table>

Flags

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

GPU temporary location

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
</tbody>
</table>

CPU temporary location

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td></td>
</tr>
</tbody>
</table>

Adjacent synchronization: CPU and GPU
In-place implementation will be possible
Benefits of Collaboration: Padding

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

Gómez-Luna+, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," ISPASS 2017
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

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 3 0 0 1 3 4 0 0 2 1</td>
<td>2 1 3 1 3 4 2 1</td>
</tr>
</tbody>
</table>

Predicate: Element > 0
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

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

Gómez-Luna+, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," ISPASS 2017
Coarse-Grained Task Partitioning
Breadth-First Search

- Small-sized and big-sized frontiers
  - Top-down approach
  - Kernel 1 and Kernel 2
- Atomic-based inter-block synchronization
  - Avoids kernel re-launch

- Very small frontiers
  - Underutilize GPU resources
- Collaborative implementation
Motivation

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

- Small frontiers processed on CPU
- Large frontiers processed on GPU
Without Unified Memory (UM)

- 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 with UM (I)

- **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 for every frontier (kernel termination and relaunch)
        cudaDeviceSynchronize();
    }
}
```
**BFS: Kernel Termination and Relaunch**

- AMD Kaveri (4 CPU cores + 8 GPU cores)
  - High overhead of kernel relaunch makes CPU+GPU collaboration impractical

---

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 (frontier_size / block_size) blocks
  - Persistent blocks: up to (number_SMs \times max_blocks_SM)
Atomic-based Block Synchronization (I)

- 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
}
```
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
...
```
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 (keep running while frontier_size >= LIMIT)
        cudaDeviceSynchronize();
    }
}
```

- **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 with UM (II)

- Pascal/Volta/Turing/Ampere Unified Memory & HSA
  - 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

```c
// Host code
while(frontier_size != 0){
    if(frontier_size < LIMIT){
        // Launch CPU threads (compute when frontier_size < LIMIT)
    } else{
        // Launch GPU kernel (compute when frontier_size >= LIMIT)
    }
}
cudaDeviceSynchronize();
```
■ **AMD Kaveri (4 CPU cores + 8 GPU cores)**
  - The collaborative implementation (with system-wide atomics) is up to 39% faster than the GPU only version
Benefits of Collaboration: SSSP

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

![Graph showing execution time for SSSP with different configurations.](image)

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

Gómez-Luna+, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," ISPASS 2017
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
RANSAC: 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
Collaborative Patterns

Program Structure

Data Partitioning

Coarse-grained Task Partitioning

Fine-grained Task Partitioning

Chang+, "Collaborative Computing for Heterogeneous Integrated Systems," ICPE 2017
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

Gómez-Luna+, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," ISPASS 2017
# Chai Benchmarks

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

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

---

Gómez-Luna+, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," ISPASS 2017
Chai: Diversity of Benchmarks (I)

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

### DATA PARTITIONING

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Partitioning Granularity</th>
<th>Partitioned Data</th>
<th>System-wide Atomics</th>
<th>Load Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS</td>
<td>Fine</td>
<td>Output</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>CEDD</td>
<td>Coarse</td>
<td>Input, Output</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>HSTI</td>
<td>Fine</td>
<td>Input</td>
<td>Compute</td>
<td>No</td>
</tr>
<tr>
<td>HSTO</td>
<td>Fine</td>
<td>Output</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>PAD</td>
<td>Fine</td>
<td>Input, Output</td>
<td>Sync</td>
<td>Yes</td>
</tr>
<tr>
<td>RSCD</td>
<td>Medium</td>
<td>Output</td>
<td>Compute</td>
<td>Yes</td>
</tr>
<tr>
<td>SC</td>
<td>Fine</td>
<td>Input, Output</td>
<td>Sync</td>
<td>No</td>
</tr>
<tr>
<td>TRNS</td>
<td>Medium</td>
<td>Input, Output</td>
<td>Sync</td>
<td>No</td>
</tr>
</tbody>
</table>

### FINE-GRANULAR TASK PARTITIONING

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>System-wide Atomics</th>
<th>Load Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSCT</td>
<td>Sync, Compute</td>
<td>Yes</td>
</tr>
<tr>
<td>TQ</td>
<td>Sync</td>
<td>No</td>
</tr>
<tr>
<td>TQH</td>
<td>Sync</td>
<td>No</td>
</tr>
</tbody>
</table>

### COARSE-GRANULAR TASK PARTITIONING

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>System-wide Atomics</th>
<th>Partitioning</th>
<th>Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Sync, Compute</td>
<td>Iterative</td>
<td>No</td>
</tr>
<tr>
<td>CEDT</td>
<td>Sync</td>
<td>Non-iterative</td>
<td>Yes</td>
</tr>
<tr>
<td>SSSP</td>
<td>Sync, Compute</td>
<td>Iterative</td>
<td>No</td>
</tr>
</tbody>
</table>
Chai: Diversity of Benchmarks (II)

Varying intensity in use of system-wide atomics

Diverse execution profiles

Gómez-Luna+, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," ISPASS 2017
Benefits of Unified Memory: Kernel Time

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

**Execution Time (normalized)**

- **Kernel**

<table>
<thead>
<tr>
<th></th>
<th>BS</th>
<th>CEDD</th>
<th>HSTI</th>
<th>HSTO</th>
<th>PAD</th>
<th>RSCD</th>
<th>SC</th>
<th>TRNS</th>
<th>RSCT</th>
<th>TQ</th>
<th>TQH</th>
<th>BFS</th>
<th>CEDT</th>
<th>SSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>U</td>
<td>D</td>
<td>U</td>
<td>D</td>
<td>U</td>
<td>D</td>
<td>U</td>
<td>D</td>
<td>U</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

**Data Partitioning**

- Fine-grain

**Task Partitioning**

- Coarse-grain

AMD Kaveri (4 CPU cores + 8 GPU cores), OpenCL

Gómez-Luna+, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," ISPASS 2017
Benefits of Unified Memory: Data Transfers

Unified versions avoid copy overhead

Execution Time (normalized)

Data Partitioning

Task Partitioning

Fine-grain

Coarse-grain

Kernel | Copy Back & Merge | Copy To Device

AMD Kaveri (4 CPU cores + 8 GPU cores), OpenCL

Gómez-Luna+, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," ISPASS 2017
Benefits of Unified Memory: Allocation

SVM allocation seems to take longer

AMD Kaveri (4 CPU cores + 8 GPU cores), OpenCL

Gómez-Luna+, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," ISPASS 2017
Comparison C++AMP vs. OpenCL-U

Gómez-Luna+, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," ISPASS 2017

- Chapter 8 – Application use cases: Platform atomics
Background: Traditional I/O Technology

Typical I/O Model Flow: Total ~13µs for data prep

- DD Call: 300 Instructions, 7.9µs
- Copy or Pin Source Data: 10,000 Instructions
- MMIO Notify Accelerator
- Application Dependent, but Equal to below: 3,000 Instructions, 4.9µs
- Poll / Interrupt Completion: 1,000 Instructions
- Copy or Unpin Result Data: 1,000 Instructions
- Ret. From DD Completion

Dionysios Diamantopoulos, IBM Research – Zurich, COOL Chips 2018
CAPI/OpenCAPI Overview

- CAPI/CAPI2 (Coherent Accelerator Processor Interface)
- OpenCAPI
Collaborative Computing on CPU+FPGA

- Traditionally, accelerators (GPUs, FPGAs, etc.) have been used as *offload* engines
- Heterogeneous architectures moving towards tighter integration
  - Unified memory
  - System-wide atomics
- Tighter integration allows fine-grained collaboration

**Key challenge**: identify the best CPU-FPGA collaboration strategy

Huang+, "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures," ICPE 2019
Intel OpenCL SDK for FPGA

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

Huang+, "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures," ICPE 2019
### CPU+FPGA Evaluation Platforms

<table>
<thead>
<tr>
<th></th>
<th>Platform A</th>
<th>Platform B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FPGA Board</strong></td>
<td>Terasic DE5-Net</td>
<td>Nallatech 510T</td>
</tr>
<tr>
<td><strong>FPGA Chip</strong></td>
<td><strong>Intel Stratix V GX</strong></td>
<td><strong>Intel Arria 10 GX</strong></td>
</tr>
<tr>
<td><strong>On-Board Memory</strong></td>
<td>4 GB (DDR3)</td>
<td>8 GB (DDR4)</td>
</tr>
<tr>
<td><strong>Host CPU</strong></td>
<td>Intel Xeon E3-1240 v3</td>
<td>Intel Xeon E5-2650 v3</td>
</tr>
<tr>
<td><strong>Host Memory</strong></td>
<td>8 GB (DDR3)</td>
<td>96 GB (DDR4)</td>
</tr>
<tr>
<td><strong>Interface</strong></td>
<td>PCIe gen3.0 x8</td>
<td>PCIe gen3.0 x8</td>
</tr>
</tbody>
</table>

Huang+, "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures," ICPE 2019
Benefits of Collaboration on FPGA (I)

Case Study: Canny Edge Detection

Chang+, "Collaborative Computing for Heterogeneous Integrated Systems," ICPE 2017
Huang+, "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures," ICPE 2019
Benefits of Collaboration on FPGA (II)

Case Study: Random Sample Consensus

Chang+, "Collaborative Computing for Heterogeneous Integrated Systems," ICPE 2017
Huang+, "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures," ICPE 2019
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"


[Slides (pptx) (pdf)]
[Chai CPU-FPGA Benchmark Suite]
Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs

Jiantong Jiang\(^1\)*, Zeke Wang\(^2\)*, Xue Liu\(^1\)*, Juan Gómez-Luna\(^2\), Nan Guan\(^3\), Qingxu Deng\(^1\), Wei Zhang\(^4\), and Onur Mutlu\(^2\)

1 Department of Computer Science and Engineering, Northeastern University, China
2 ETH Zürich, Switzerland
3 Department of Computing, Hong Kong Polytechnic University, Hong Kong
4 Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong
CAPI/OpenCAPI Overview

- CAPI/CAPI2 (Coherent Accelerator Processor Interface)
- OpenCAPI

Flow with a CAPI Model:

- Total 0.36µs

Shared Mem.
Notify Accelerator

Acceleration

Shared Memory Completion

400 Instructions

0.3µs

Application Dependent, but Equal to above

100 Instructions

0.06µs

Dionysios Diamantopoulos, IBM Research – Zurich, COOL Chips 2018
Evaluation Setup for Weather Acceleration

- **Host System**
  IBM POWER9-16 core (64-threads)

- **FPGA board**
  Xilinx Virtex® Ultrascale+™ XCVU37P-2

Source: IBM

Source: AlphaData
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

Singh+, "NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling," FPL 2020
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\textsuperscript{a,b,c} Dionysios Diamantopoulos\textsuperscript{c} Christoph Hagleitner\textsuperscript{c} Juan Gómez-Luna\textsuperscript{b}
Sander Stuijk\textsuperscript{a} Onur Mutlu\textsuperscript{b} Henk Corporaal\textsuperscript{a}
\textsuperscript{a}Eindhoven University of Technology \textsuperscript{b}ETH Zürich \textsuperscript{c}IBM Research Europe, Zurich

FPGA-based Near-Memory Acceleration of Modern Data-Intensive Applications

Gagandeep Singh◊ Mohammed Alser◊ Damla Senol Cali✦
Dionysios Diamantopoulos▽ Juan Gómez-Luna◊
Henk Corporaal* Onur Mutlu◊✦

◊ETH Zürich ○ Carnegie Mellon University
*Eindhoven University of Technology ▽ IBM Research Europe
Compute Express Link (CXL)

- Compute Express Link (CXL) is an open industry standard interconnect offering high-bandwidth, low-latency connectivity between host processor and devices such as accelerators, memory buffers, and smart I/O devices.
NVIDIA Grace CPU and Grace Hopper

Top Global Systems Makers Accelerate Adoption of NVIDIA Grace and Grace Hopper

Atos, Dell Technologies, GIGABYTE, Hewlett Packard Enterprise, Inspur, Lenovo and Supermicro Join First Wave Planning NVIDIA Grace-Powered HGX Systems for HPC and AI

Monday, May 30, 2022

NVIDIA today announced that a range of the world’s leading computer makers are adopting the new NVIDIA Grace™ superchips to create the next generation of servers turbocharging AI and HPC workloads for the exascale era.

Atos, Dell Technologies, GIGABYTE, HPE, Inspur, Lenovo and Supermicro are planning to deploy servers built with the NVIDIA Grace CPU Superchip and NVIDIA Grace Hopper™ Superchip.

All these new systems benefit from the just-announced Grace and Grace Hopper designs in the NVIDIA HGX™ platform, which provide manufacturers the blueprints needed to build systems that offer the highest performance and twice the memory bandwidth and energy efficiency of today’s leading data center CPU.

“As supercomputing enters the era of exascale AI, NVIDIA is teaming up with our OEM partners to enable researchers to tackle massive challenges previously out of reach,” said Ian Buck, vice president of Hyperscale and HPC at NVIDIA. “Across climate science, energy research, space exploration, digital biology, quantum computing and more, the NVIDIA Grace CPU Superchip and Grace Hopper Superchip form the foundation of the world’s most advanced platform for HPC and AI.”

Taiwan’s Tech Titans Adopt World’s First NVIDIA Grace CPU-Powered System Designs

New Class of Data Center Systems for Digital Twins, AI, High Performance Computing, Cloud Graphics and Gaming to Come From ASUS, Foxconn Industrial Internet, GIGABYTE, QCT, Supermicro, Wiwynn

Monday, May 23, 2022

COMPUTEX — NVIDIA today announced that Taiwan’s leading computer makers are set to release the first wave of systems powered by the NVIDIA Grace™ CPU Superchip and Grace Hopper Superchip for a wide range of workloads spanning digital twins, AI, high performance computing, cloud graphics and gaming.

Dozens of server models from ASUS, Foxconn Industrial Internet, GIGABYTE, QCT, Supermicro and Wiwynn are expected starting in the first half of 2023. The Grace-powered systems will join AMD and other Arm-based servers to offer customers a broad range of choice for achieving high performance and efficiency in their data centers.

“A new type of data center is emerging — AI factories that process and refine mountains of data to produce intelligence — and NVIDIA is working closely with our Taiwan partners to build the systems that enable this transformation,” said Ian Buck, vice president of Hyperscale and HPC at NVIDIA. “These new systems from our partners, powered by our Grace Superchips, will bring the power of accelerated computing to new markets and industries globally.”

The coming servers are based on four new system designs featuring the Grace CPU Superchip and Grace Hopper Superchip, which NVIDIA announced at its two recent OTC conferences. The 2U form factor designs provide the blueprints and server baseboards for original design manufacturers and original equipment manufacturers to quickly bring to market systems for the NVIDIA CDX™ cloud gaming, NVIDIA OXX™ digital twin and the NVIDIA H10™ AI and HPC platforms.
NVIDIA Grace CPU Superchip

- 144 ARM v9 CPU cores
- LPDDR5x memory with ECC, 1 TB/s total bandwidth
- 900 GB/s coherent interface (7x faster than PCIe Gen 5)

NVIDIA Grace Hopper Superchip

- CPU + GPU
  - Grace CPU + Hopper GPU
- 900 GB/s coherent interface (7x faster than PCIe Gen 5)

Collaborative Computing: Key Takeaways

- Possibility of having several devices collaborating on the same workload
- And having the most appropriate cores for each workload, exploiting heterogeneity
- 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
Processing-in-Memory Course (Spring 2022)

- Short weekly lectures
- Hands-on projects

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

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.

https://safari.ethz.ch/projects_and_seminars/spring2022/doku.php?id=processing_in_memory

https://youtube.com/playlist?list=PL5Q2soXY2Zi-0NK1C5vi2Zx9nmE_3-cKN
Heterogeneous Systems Course (Spring 2022)

- Short weekly lectures
- Hands-on projects

Hands-on Acceleration on Heterogeneous Computing Systems

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.

https://youtube.com/playlist?list=PL5Q2soXY2Zl9XrqXR38IM_FTjmY6h7Gzm

https://safari.ethz.ch/projects_and_seminars/spring2022/doku.php?id=heterogeneous_systems
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)
- Intelligent Architectures using Hardware/Software Cooperative Techniques

https://safari.ethz.ch/projects_and_seminars/spring2022/doku.php?id=start
More Resources: Onur Mutlu Lectures

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

More resources available:
https://www.youtube.com/c/OnurMutluLectures/playlists