P&S Heterogeneous Systems
Collaborative Computing

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Spring 2023
16 June 2023
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

![Diagram of CUDA 6 Unified Memory and Pascal Unified Memory]

(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: 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
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

```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/Hopper 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
  cudaMemcpyManaged(&h_in, in_size);
  ```

- **System-wide atomics**
  
  ```c
  old = atomicAdd_system(&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-parallel tasks

Sequential sub-tasks

Coarse-grained synchronization

Device 1

Device 2

Data Partitioning

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

Program Structure

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

Device 1

Device 2

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

Fine-grained Task Partitioning

Chang+, "Collaborative Computing for Heterogeneous Integrated Systems," ICPE 2017
Data Partitioning
Bézier Surfaces (I)

- Bézier surface: 4x4 net of control points

Palomar+, "High-Performance Computation of Bézier Surfaces on Parallel and Heterogeneous Platforms," IJPP, 2018
Bézier Surfaces (II)

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

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

$$B_{i,m}(u) = \binom{m}{i} (1 - u)^{(m-i)} u^i, \quad (2)$$
Bézier Surfaces: Static Distribution (I)

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

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);
```
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

![Graph showing execution time vs. %Tiles to CPU]
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/Hopper Unified Memory:** system-wide atomic operations

  ```c
  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
  ```

- 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

![Execution Time Graph](image)

Bézier Surfaces
(up to 47% 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
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

![Graph showing average execution time and nodes per frontier for CPU (4 threads) and GPU (4x256 threads)]
BFS: Collaborative Implementation

- Choose the most appropriate device

- small frontiers processed on CPU
- large frontiers processed on GPU
Collaborative Implementation without UM

- **Without** Unified Memory (UM)
  - Explicit memory copies

```java
// 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)
    }
}
```
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 for every frontier (kernel termination and relaunch)
        cudaDeviceSynchronize();
    }
}
Collaborative Implementation with UM (II)

- Pascal/Volta/Turing/Ampere/Hopper 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();
```
Benefits of Collaboration: BFS

- 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
Fine-Grained Task Partitioning
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
In-Place Padding

- Pascal/Volta/Turing/Ampere Unified Memory

Adjacent synchronization:
CPU and GPU
In-place implementation will be possible
Heterogeneous Systems Course (Spring 2023)

- Short weekly lectures
- Hands-on projects

https://www.youtube.com/playlist?list=PL5Q2soXY2Zi-gSKahS4ofaEwY17_qp9mw

https://safari.ethz.ch/projects_and_seminars/spring2023/doku.php?id=heterogeneous_systems

SAFARI Project & Seminars Courses (Spring 2023)
Trace: • processing_in_memory • heterogeneous_systems

Home
Courses
- SoftMIC
- Ramulator
- Accelerating Genomics
- Mobile Genomics
- Processing-in-Memory
- Heterogeneous Systems

Table of Contents
- Programming Heterogeneous Computing Systems with GPUs and other Accelerators (227-0085-51L)
  - Course Description
  - Mentors
  - Lecture Video Playlist on YouTube
  - Spring 2023 Meetings/Schedule
  - Past Lecture Video Playlists on YouTube
  - Learning Materials
  - Assignments

Programming Heterogeneous Computing Systems with GPUs and other Accelerators (227-0085-51L)

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 a critical 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.
Processing-in-Memory Course (Spring 2023)

- Short weekly lectures
- Hands-on projects

https://www.youtube.com/playlist?list=PL5Q2soXY2Zi_EObuoAZVSq_o6UyS
WQHvZ
More P&S Courses: SSDs, Memory, Bioinformatics…

- FPGA-based Exploration of DRAM and RowHammer
- Exploration of Emerging Memory Systems
- Accelerating Genome Analysis with FPGAs, GPUs, and New Execution Paradigms
- Genome Sequencing on Mobile Devices
- Data-Centric Architectures: Fundamentally Improving Performance and Energy
- Programming Heterogeneous Computing Systems with GPUs and other Accelerators
- Understanding and Designing Modern NAND Flash-Based SSDs
- Intelligent Architectures using Hardware/Software Cooperative Techniques

https://safari.ethz.ch/projects_and_seminars/spring2023/doku.php?id=start
More Resources: Onur Mutlu Lectures

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

https://www.youtube.com/c/OnurMutluLectures/playlists
P&S Heterogeneous Systems

Collaborative Computing

Dr. Juan Gómez Luna
Prof. Onur Mutlu

ETH Zürich
Spring 2023
16 June 2023