Today

• GPU vs CPU
  • Different architecture, different workloads

• Basics of CUDA
  • Executing code on GPU
  • Managing memory between CPU and GPU

• CUDA API
  • Quick look at the most important parts

• Basic optimization techniques
  • Memory access pattern optimization
  • Shared memory and block-wide cooperation
Workloads used to be sequential
   And still mostly are

Thus, CPU design is \textit{latency-oriented}
   Aim for \textit{low latency}
   Make sequentially dependent code run as fast as possible
   Try to avoid memory access bottleneck with large caches

Parallelism almost an afterthought
   SIMD instruction sets – explicit programmer control
   Hyperthreading and multiple cores – max speedup \( \approx 2 \times \#\text{cores} \)
Workloads have always been massively parallel and mostly coherent (vertices, pixels, etc.)

Thus, GPU design is **throughput-oriented**
- Aim for high throughput
- Finish as many instructions per clock cycle as possible
- Need a lot of computation on chip → cannot afford large caches
  - Solution: More parallelism

Parallelism deeply ingrained in the design
- Single-threaded execution is not valid use of the hardware
Multicore CPU: Run ~10 threads fast

- **Few** processors, each supporting **1–2** hardware threads
- Large on-chip caches near processors for hiding latency
- Each thread gets to run instructions close to back-to-back
Manycore GPU: Run ~10,000 threads fast

- Hundreds of processors, each supporting dozens of hardware threads
- Small on-chip memories near processors
  - Use as explicit local storage, allow thread co-operation
- Hide latency by switching between threads
Analogy

- Mostly payload
  - Threads doing computation
- High latency
- High throughput

- Mostly boat
  - Keeping few threads going fast
- Low latency
- Low throughput
When is using a GPU a good idea? (1)

- **Large workload**
  - Preferably 10,000s of threads per launch

- **Coherent execution**
  - Each thread does *approximately* the same thing
  - No gratuitous branching, pointer chasing, etc.
  - More about this in next week’s lecture

- **Coherent data access**
  - GPUs have *very small* amounts of cache per thread
  - *Spatial reuse* is captured well, *temporal reuse* is captured badly
When is using a GPU a good idea? (2)

- Not all tasks are suitable for GPUs
  - Compilers, Powerpoint, running OS kernel, etc.

- But many are
  - Image / video / audio processing
  - Deep learning, neural networks
    - AlphaGO, self-driving cars, …
    - GPUs are used by *everyone* using DNNs
  - Supercomputing applications
  - Anything with a lot of number crunching
Questions?
Basics of CUDA
Main document: Programming Guide
- Language extensions
- Pre-defined variables in kernels
- Execution model: Parallelism, synchronization points, etc.

API reference: CUDA Runtime API
- CUDA functions for memory management, etc.

Plus a lot more – Best Practices Guide, Tuning Guides, etc.
Quick note about API flavors

- We are using the **CUDA Runtime API** on this course
  - Functions are prefixed with `cuda...`, for example `cudaMalloc()`

- Don’t confuse it with **CUDA Driver API**
  - Low-level API that we will NOT use
  - Functions are prefixed with `cu...`, for example `cuMemAlloc()`

- Just something to watch out for when seeking information
Median5 walkthrough

- We will look at code snippets from a simple GPU-based Median5 program throughout this part.

- Simplified: No error checks, no file I/O, etc.
CUDA machine model

- GPU memory (on GPU board)
- GPU memory (on GPU board)
- GPU memory (on GPU board)
- GPU
- GPU
- GPU
- CPU
- PCIE Bus
- Main memory (on motherboard)
GPU memory is not directly accessible from CPU, and vice versa. They *can* be made accessible, but it’s relatively slow because of the PCIE bus. Yet sometimes this is the fastest solution, e.g., when read exactly once.
Memory management

Programs must explicitly
- Allocate memory buffers on both host and device
- Transfer data between host and device

Transferring data takes time
- There has to be enough work per transferred data to justify transfer time
- However, transfers can be overlapped with computation
- Advanced topic for next week
int main()
{
    const int width = 16000;
    const int height = 16000;
    const int size = width * height;

    // Allocate CPU buffers for input and output. Using cudaHostAlloc() gives
    // page-locked memory that is faster to transfer between CPU and GPU than
    // memory allocated with malloc().

    int* inputCPU = 0;
    int* outputCPU = 0;
    cudaHostAlloc((void**)&inputCPU, size * sizeof(int), cudaHostAllocMapped);
    cudaHostAlloc((void**)&outputCPU, size * sizeof(int), cudaHostAllocMapped);

    ... Fill inputCPU[] with something ...
}
Median5: Allocating device memory

```c
int main()
{
    ...

    // Allocate GPU buffers for input and output.
    int* inputGPU = 0;
    int* outputGPU = 0;
    cudaMalloc((void**)&inputGPU, size * sizeof(int));
    cudaMalloc((void**)&outputGPU, size * sizeof(int));
}
int main()
{
...

// Copy input buffer from CPU to GPU.

cudaMemcpy(inputGPU, inputCPU, size * sizeof(int), cudaMemcpyHostToDevice);

// Now we have:
// inputCPU = Host-side buffer containing the input.
// inputGPU = Device-side buffer containing the input.
// outputCPU = Host-side buffer containing garbage.
// outputGPU = Device-side buffer containing garbage.
Launching a kernel

- Work is always **launched** for a number of **threads** at a time
- The function being executed on GPU is called the **kernel**
Median5: Kernel specification

```c
__global__ void medianKernel(int* output, const int* input,
                              const int width, const int height, const int size)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x; // Who am I?
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if (x >= width || y >= height) // Exit if outside the image.
        return;

    int p0 = x + width * y; // Index of center pixel.
    int a0 = input[p0];     // Read input value at center pixel.

    ... Read other input pixels and compute median ...

    output[p0] = result;   // Write result into output buffer.
}
```
Thread blocks

- **Thread block** is the basic unit when launching work on GPU
  - Block is a 1D, 2D or 3D collection of threads

- All threads of one block run concurrently, and on the same core
  - Can access block-wide *shared memory*, do block-wide synchronization
    - More about these in a moment

- Each block has to be **independent**
  - No guarantees about execution order or concurrency between blocks

- In many cases you don’t have to care much about this
  - If all threads working independently, block shape is just a perf parameter
Example launch configuration

- Blocks of $8 \times 8$ threads, grid of $10 \times 5$ blocks
- Total = $8 \times 8 \times 10 \times 5 = 3200$ threads

Different in each thread:
- threadIdx.x = 1
- threadIdx.y = 1
- blockIdx.x = 9
- blockIdx.y = 0

Same in every thread:
- blockDim.x = 8
- blockDim.y = 8
- gridSize.x = 10
- gridSize.y = 5
Median5: Kernel specification revisit

```c
__global__ void medianKernel(int* output, const int* input,
                             const int width, const int height, const int size)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x; // Who am I?
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if (x >= width || y >= height) // Exit if outside the image.
        return;

    int p0 = x + width * y; // Index of center pixel.
    int a0 = input[p0];     // Read input value at center pixel.

    ... Read other input pixels and compute median ...

    output[p0] = result; // Write result into output buffer.
}
```
Launching a kernel

```
kernelFunc<<<gridSize, blockSize>>>(param1, param2, ...)
```

- `kernelFunc` is a function annotated with keyword `__global__`
- `gridSize` and `blockSize` are variables of type `dim3`
  - Just a struct with fields `x, y, z`
  - Constructors with 1, 2, 3 parameters
    - E.g., `dim3(64, 64)` is valid
    - Sets unspecified dimensions to 1
  - Also implicit type cast from `int`
    - Allows things like `KernelFunc<<<100, 512>>>(...)`
__global__ void medianKernel(int* output, const int* input,
                         const int width, const int height, const int size)
{
    ...
}

int main()
{
    ...

    // Determine block and grid sizes for the kernel launch.
    dim3 dimBlock(64, 1);
    dim3 dimGrid((width + dimBlock.x - 1) / dimBlock.x, (height + dimBlock.y - 1) / dimBlock.y);

    // Launch the kernel on GPU.
    medianKernel<<<dimGrid, dimBlock>>>(outputGPU, inputGPU, width, height, size);
Thread block sizes

Say you want to launch $400 \times 400$ threads, i.e., $160000$ in total.

Can you put all threads in the same block? No.
- Block has a maximum size of 1024 threads

Can you create 160000 blocks with 1 thread in each? Don’t do it!
- Technically possible, but hideously slow
- Block has a practical minimum size of 64 threads

So what then?
- Start with blocks of 64 threads, set grid size as needed
- Try increasing block size (multiples of 64 work best), benchmark
Thread block shapes

Is a $1 \times 64$ block the same as a $64 \times 1$ block? Yes and no.
- Hardware launches threads, uses resources just the same.
- However, threads in a block will receive different indices.

This can have a dramatic effect on memory coherence!
Median5: Copying results device $\rightarrow$ host

```c
int main()
{
  ...

  // Copy output buffer from GPU to CPU.
  cudaMemcpy(outputCPU, outputGPU, size * sizeof(int), cudaMemcpyDeviceToHost);
  ...

  // Consume the output on CPU, free the buffers ...

  return 0; // Done, exit program.
}
```
Questions?
CUDA language extensions and API
Code organization

- Source files may contain a mixture of host and device code
  - Recommended to use extension `.cu` for these

- These source files are compiled with Nvidia’s NVCC toolchain
  - Host code gets compiled with native C++ compiler (gcc, msvc, …)
  - Device code gets compiled with Nvidia tools
  - NVCC patches CUDA kernel launches in host code into standard C++
    - Detect GPU architecture, upload relevant binaries to GPU, etc.
  - Everything in a `.cu` file is linked together in a single `.o` file
Most important language extensions

- **Function and variable decorators**
  - `__global__`: Executes on GPU, callable from CPU (= kernel)
  - `__device__`: Executes on GPU, callable from GPU

- **Storage specifiers**
  - `__shared__`: Block-wide shared memory
  - `__constant__`: Constant memory

- **Special variables in device code**
  - `threadIdx`, `blockIdx`, `blockDim`, `gridDim`, ...

- **Kernel launch syntax** `<<<, >>>`
API function error codes

- Every CUDA API function returns an error code
  - Type `cudaError_t`
  - Value is `cudaSuccess` (= 0) if no error occurred

- Helpers to convert error code to string
  - `const char* cudaGetErrorName(cudaError_t error)`
  - `const char* cudaGetErrorString(cudaError_t error)`

- To get last error (e.g., from kernel launch)
  - `cudaError_t cudaGetLastError(void)`

- If something doesn’t work, check errors first!
Device management

cudaGetDeviceCount(int* count)
cudaGetDeviceProperties(cudaDeviceProp* prop, int device)
cudaSetDevice(int device)

- Device 0 is selected by default
  - Driver always places the ”best” device to slot 0
  - So, often no need to call cudaSetDevice at all
- Device selection is a low-overhead call
  - Using multiple GPUs from same CPU thread is possible
Device memory allocation

cudaMalloc(void** devPtr, size_t size)
cudaFree(void* devPtr)

- **cudaMalloc** gives a C/C++ pointer that *can’t* be dereferenced on the host
  - Mixing up host and device pointers $\rightarrow$ crash

- Valid uses for the device pointer:
  - CUDA API calls (e.g. `cudaMemcpy`)
  - Using it in kernel, maybe passed as a parameter to it
Host memory allocation

cudaMallocHost(void** ptr, size_t size)
cudaFreeHost(void* ptr)

- Returns host pointer to page-locked memory
  - If copying from / to memory allocated with malloc or new[], the driver will page-lock and unlock at every copy (works fine, but takes time)
  - With cudaMallocHost, page-locking is done only once

- Page-locked memory cannot be swapped out by the operating system, so make sure your application behaves nicely
Zero-copy memory

- CPU memory can be accessed directly by the GPU
  - Slow due to every access going through the PCIE bus

- Makes sense only in specific circumstances
  - Data is read / written once, with good access pattern
  - Sparse access into a large buffer
  - Asynchronous communication
Mapping host memory on device

cudaHostAlloc(void** pHost, size_t size, unsigned int flags)
cudaHostRegister(void* ptr, size_t size, unsigned int flags)
cudaHostGetDevicePointer(void** pDevice, void* pHost, unsigned int flags)
cudaHostUnregister(void* ptr)
cudaFreeHost(void* ptr)

- Set flags=cudaHostAllocMapped to get zero-copy -eligible memory
- Allocate mapped, page-locked memory
  - Or register (i.e., page-lock + map) an existing range
- Query a device pointer to that range
Copying memory

cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)

One function for all directions

- Plain memcpy is of course fine for host → host copy
- dst and src must be either host or device pointers depending on usage
- kind is cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToDevice, or cudaMemcpyDeviceToHost
Note: CUDA API is mostly asynchronous

- Many CUDA calls, all kernel launches are asynchronous
  - Place a request in GPU command stream
  - Host thread continues running immediately

- Kernels actually execute when the device frees up

- Implicit synchronization at certain points
  - E.g., memory transfers

- Explicit synchronization is also possible
  - Matters when benchmarking
  - Also, when checking for errors!
Explicit synchronization

cudaDeviceSynchronize(void)

- Returns after all kernel launches and API calls have been completed
- Checking errors after this will detect if kernel did something bad
Benchmarking pitfalls, part 1

1. Start CPU timer
2. Copy data from host to device
3. Launch kernel
4. Copy results from device to host
5. Stop CPU timer

Problem: step 2 may need to wait until a previous kernel launch is complete
Benchmarking pitfalls, part 2

1. Copy data from host to device
2. Start CPU timer
3. Launch kernel
4. Stop CPU timer
5. Copy results from device to host

Problem: Measures only the API overhead of a kernel launch
- Close to constant time regardless of what the kernel does!
Benchmarking pitfalls, part 3

1. Copy data from host to device
2. Call `cudaDeviceSynchronize()`
3. **Start CPU timer**
4. Launch kernel
5. Call `cudaDeviceSynchronize()`
6. **Stop CPU timer**
7. Copy results from device to host

Problem: Low accuracy for short launches, includes sync overhead of step 5
Benchmarking done right

- For accurate GPU benchmarking, must use CUDA events
  - Provides exact time when GPU passed a given point in command stream
  - More about this next week, maybe

- Spent time on this only because asynchronous API often surprises programmers trying to see where the time is spent
  - Using an interactive profiler is also an option, but not always feasible
Questions?
Basic optimization techniques

Optimizing memory access patterns
Shared memory and block-wide cooperation
Memory access patterns

- Just as important in GPU programming as in CPU programming
  - But the rules are slightly different

- Threads in the same block should access nearby memory locations at about the same time

- This can have a huge effect on performance
Access patterns example

Example: We use blocks of 64 threads, and want each thread to process 10 elements of a linear array.

Thus, each block processes 640 elements.

It is best to have each block process a contiguous 640-element chunk of the array, just like on CPU.

Question: Which elements should be processed by each thread in the block?
Improving access patterns with striding

**No striding**

```c
for (i = 0; i < 10; i++) {
    arrayIdx = blockIdx.x * 640 + threadIdx.x * 10 + i
    ...
}
```

<table>
<thead>
<tr>
<th>Thread 0:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1:</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 63:</td>
<td>630</td>
<td>631</td>
<td>632</td>
<td>633</td>
<td>634</td>
<td>635</td>
<td>636</td>
<td>637</td>
<td>638</td>
<td>639</td>
</tr>
</tbody>
</table>

Bad access pattern

**Stride of 64**

```c
for (i = 0; i < 10; i++) {
    arrayIdx = blockIdx.x * 640 + threadIdx.x + i * 64
    ...
}
```

<table>
<thead>
<tr>
<th>Thread 0:</th>
<th>0</th>
<th>64</th>
<th>128</th>
<th>192</th>
<th>256</th>
<th>320</th>
<th>384</th>
<th>448</th>
<th>512</th>
<th>576</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1:</td>
<td>1</td>
<td>65</td>
<td>129</td>
<td>193</td>
<td>257</td>
<td>321</td>
<td>385</td>
<td>449</td>
<td>513</td>
<td>577</td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 63:</td>
<td>63</td>
<td>127</td>
<td>191</td>
<td>255</td>
<td>319</td>
<td>383</td>
<td>447</td>
<td>511</td>
<td>575</td>
<td>639</td>
</tr>
</tbody>
</table>

Optimal access pattern
Access patterns, GPU vs CPU

Note how the optimal access pattern differs from CPU

CPU: Each thread should process contiguous chunk of memory
   - Because threads run on different cores, on separate L1 caches

GPU: Threads in the same block should access contiguous chunks of memory at the same time
   - In reality, many threads in the same block do run in lock-step
   - Memory accesses can be coalesced if they land on the same cache line
   - Potential for $32 \times$ cost difference for each memory fetch!
Address space view

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>20</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>i=0</td>
<td>i=1</td>
<td>i=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T0</td>
<td>i=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T0</td>
<td>i=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>i=0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>i=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>i=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Good for CPU
Bad for GPU

Address | 0   | 1   | 2   | 64  | 65  | 66  | 128 | 129 | 130 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>i=0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>i=0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>i=0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T0</td>
<td>i=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>i=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>i=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T0</td>
<td>i=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>i=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>i=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bad for CPU
Good for GPU
Shared memory

- Specified using the __shared__ keyword

- **Not a per-thread resource** but a per-block resource
  - Completely unlike ordinary variables
  - All threads in a block see the same variable / array

- To avoid race conditions, threads in a block must often synchronize between accesses to shared memory
  - There are no guarantees that all threads in a block progress at exactly the same speed
Shared memory, continued

Shared memory is physically close to the execution units
  - Much faster than ordinary device memory

It is also a limited resource
  - One limiter for how many threads the GPU can have resident at once

If you use way too much shared memory (>48 KB per thread block), the kernel cannot be launched at all
Shared memory, typical scenario

1. Each thread moves a piece of data from device memory into shared memory

2. Threads in the block synchronize to ensure data is valid

3. Each thread looks at a window of data in shared memory
   - Read-only access poses no hazards, so no need to synchronize

4. Each thread writes its result into its own location in device memory
Block-wide synchronization

- Achieved with a call to `__syncthreads()` in the kernel

  Calling threads will stall until all non-terminated threads have reached that call. After that all threads in the thread block are released.

  - If a thread has been terminated by `return`; it is not waited for

  - If some threads call `__syncthreads()` and some do not, the kernel will almost certainly hang

  Conditional synchronization is allowed only if all threads in the block agree on the condition.
Shared memory is a block-wide construct. It is useful and necessary only when doing cooperation between threads in a thread block.

If all your threads are truly independent, you probably don’t want to use shared memory. Hence no need for block-wide synchronization either.

Useful for some applications, useless for some.

Programming guide has an example of matrix multiplication that takes advantage of shared memory.
Next week

- GPU architecture
- How GPU actually executes code
  - Spoiler: It’s almost like SIMD but better in every way
- More optimization techniques
- Open to requests
Thank you!

Questions?