GPU Programming Using CUDA

Samuli Laine
Today

- Brief history of CUDA
- Machine and execution model
- Writing CUDA kernels
- Using the API

- Will not look at the hardware yet
  - That’s for next week
Motivation for GPU programming

- This thing packs a lot FLOPs and memory bandwidth
- How to tap into that?
Early days: GPGPU

- **General-Purpose GPU** programming
  - The craze around 2004 – 2006
- Trick the GPU into general-purpose computing by casting problem as graphics
  - Turn data into images (textures)
  - Turn algorithms into image synthesis (rendering passes)
- Many attempts to handle these automatically
  - Brook, Sh, PeakStream, MS Accelerator, …
  - Take a “program”, somehow convert to shaders
Problems with GPGPU

- Constrained memory access model
  - No scattered writes, no generic read/write

- Split computation into multiple passes
  - Limited by what shaders can do
  - Must understand graphics HW to understand the limitations
  - Trickery to circumvent the rigidity of hardware

- Overhead of graphics API
GPGPU: An illustrated guide
The road to CUDA

Okay, this GPGPU thing has potential
  The only problem is that it sucks
Let’s design the right tool for the job
Need new hardware capabilities? → Build it
  We are a hardware company, after all
Develop a better API for poking the GPU
  Extend C++, don’t invent a new language
CUDA 1.0 released in June 2007
  An established and mature platform by now
What is CUDA

- A set of C/C++ language extensions that allow writing programs that run on GPUs
- C/C++ API for configuring and managing GPU execution, memory, etc.

- Tools for compiling, profiling and debugging your code
- Libraries for common tasks (FFT, BLAS, …)
- Documentation
Machine and execution model
CUDA machine model
"CUDA device"

Device memory

GPU

Device memory

GPU

PCIE bus

"Host"

CPU

Host memory
CUDA execution model

- **Kernel** ≈ A function executed over a large number of threads on a GPU
- A kernel is launched over a grid of blocks
  - Blocks and grid can be 1D, 2D or 3D
  - Extra dimensions are really just syntactic sugar but convenient if the data lives in a 2D or 3D domain
- Every thread can query its
  - Thread location within the block (**threadIdx**)
  - Block location within the grid (**blockIdx**)
  - Block and grid dimensions (**blockDim**, **gridDim**)
Example

- Use blocks with $8 \times 8$ threads
- Launch a grid of $10 \times 5$ blocks
- Total = $8 \times 8 \times 10 \times 5 = 3200$ threads
Why blocks?

Why do we have blocks, instead of just a flat gigantic grid of threads?
- Block is guaranteed to be localized
  - All threads of a block run at the same time

Threads of a block can use fast shared memory
- Load common data together and work on it

Threads of a block can synchronize efficiently

Individual blocks must be truly independent
- No guarantees about execution order or parallelism
Writing CUDA kernels
Digression: CUDA API flavors

- **Driver API** (function prefix `cu`, e.g., `cuMemcpy`)
  - Low-level → lots of control for programmer
  - E.g., explicit uploading of kernel binaries

- **Runtime API** (function prefix `cuda`, e.g., `cudaMemcpy`)
  - High-level → easy to use
  - Automatic mixed C/C++/CUDA compilation
  - Automatic management of kernel binaries
  - Syntactic sugar for ease of use
  - **We will be using this**
Code organization

- Source files may contain a mixture of host and device code
  - Recommended to use extension `.cu` for these
- These are compiled with Nvidia’s NVCC
  - Host code gets compiled with `gcc`
  - Device code gets compiled with Nvidia tools
    - Produce device binaries for multiple platforms
  - CUDA kernel launches from host code are patched to upload the relevant binaries to GPU, etc.
- Everything is linked together in a single `.o` file
CUDA language extensions

- These are understood by NVCC

Function and variable decorators
- `__device__`: Executes on GPU, callable from GPU
- `__global__`: Executes on GPU, callable from CPU (= kernel)
  - Callable from GPU as well on modern hardware
- `__host__`: Ordinary CPU function (default, can be omitted)

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

Special variables in device code
- `threadIdx, blockIdx, blockDim, gridDim`, etc.
Kernel launch

```
kernelFunc<<<Dg, Db>>>(parameters)
```

- `kernelFunc` has to be a `__global__` function
- `Dg` and `Db` are grid and block dimensions
  - Type either `int` or `dim3`
    - Which is just a struct of `unsigned int x, y, z;`
    - Components initialized to 1 by default
Example kernel and launch

// Device code
__global__ void my_kernel(int size_x, int size_y,
                           const float* input, float* output)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if (x >= size_x || y >= size_y)
        return;
    output[x + size_x * y] = 2.0 * input[x + size_x * y];
}

// Host code
...
    dim3 szBlock(16, 16);
    dim3 szGrid((size_x + szBlock.x - 1) / szBlock.x,
                 (size_y + szBlock.y - 1) / szBlock.y);
    my_kernel<<<szGrid, szBlock>>>(size_x, size_y, ...);
    ...

Shared memory

- Specified using the `__shared__` keyword
- Unlike normal variables, not a per-thread resource but a per-block resource
  - All threads in a block see the same variable / array
- To cooperate, threads usually must synchronize between shared writes and reads
  - `__syncthreads()`
  - Does not proceed until all threads in block have reached this point
- Synchronizing in conditional code → trouble
Shared memory, continued

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

- **It is also a limited resource**
  - Basically limits how many threads the GPU can have resident
    - Somewhat complex issue, will go into more details in the next lecture
  - If you use *way* too much (over 48 KB per block), the kernel cannot be launched at all
Shared memory, example

1. Each thread moves a piece of data from device memory into shared memory
2. All threads call \texttt{__syncthreads()}
3. Each thread looks at a window of data in shared memory (read-only)
4. Each thread writes its result into its own location in device memory
Runtime API
Calling CUDA API functions

In addition to declaring and launching kernels, a program typically performs CUDA API calls to, e.g.,
- Select the CUDA device
- Allocate / free / copy memory
- Manage events for benchmarking
- Synchronize between host and device

These are plain old library function calls
- If a file does not contain device code, can be compiled with gcc
Error codes

- Every 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!
Parts of the API covered

- Device management
- Memory management
- Basic synchronization
- Events

This is only a small subset of the API, but should be enough for this course
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 cudaMemcpyDevice at all

- Device selection is a low-overhead call
  - Using multiple GPUs from same CPU thread
Memory management

- Host and device memories are separate
- But device can access host memory directly
  - So-called ”zero-copy” memory
  - Slow, but sometimes the best choice

Basic pattern: Allocate memory on device and copy data between host and device explicitly
  - This is usually the fastest option
Memory allocation on device

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

- **cudaMalloc** gives a C/C++ pointer that CANNOT be dereferenced on the host
  - Mixing up host and device pointers → crash

- Valid uses for the device pointer:
  - CUDA API calls (e.g. cudaMemcpy)
  - Passing it as parameter in a kernel launch
Memory allocation on host

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

- Returns host pointer to *page-locked* memory
  - Whenever copying data between host and device, the memory has to be page-locked
  - If copying from/to ordinarily allocated memory (*from malloc / new[]*), must page-lock and unlock at every copy, which takes time
  - Page-locked memory cannot be swapped out by the operating system
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)

- Allocate mapped, page-locked memory or register (page-lock + map) an existing range
  - Set flags = cudaHostAllocMapped
- 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\rightarrow host copy
  - `dst` and `src` must be either host or device pointers depending on usage
- `kind` is `cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice`
- Fully synchronous: Waits until device is idle, returns after copy is complete *
Note on API asynchronicity

- Many CUDA calls are in reality asynchronous
  - Place a request in command stream, return immediately
- Also true for kernel launches
  - Control returns to host thread immediately
  - Kernel starts running when the device frees up
- Implicit synchronization at certain points
  - E.g., memory transfers
- Explicit synchronization also possible
- This matters when benchmarking
API asynchronicity: Example 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

- Stop works, because step 4 forces synchronization between host and device
- Start does not: timer may include previously issued GPU operations
API asynchronicity: Example 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

- This does not give the kernel execution time
- For correct results, must either synchronize explicitly, or use events
Explicit synchronization

cudaDeviceSynchronize(void)

- Returns after all kernel launches and API calls issued so far have been completed
Synchronization example

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

Gives a rough estimate of kernel execution time
Events

cudaEventCreate (cudaEvent_t* event)
cudaEventDestroy (cudaEvent_t event)
cudaEventRecord (cudaEvent_t event)
cudaEventElapsedTime (float* ms, cudaEvent_t start, cudaEvent_t end)
cudaEventSynchronize (cudaEvent_t event)

- Best way to benchmark what happens in the GPU
- When GPU *records* an event, internal clock is stored
- Time between two recorded events can be queried
Events, example

1. Create two events: `evStart` and `evEnd`
2. Record `evStart`
3. Launch kernel
4. Record `evEnd`
5. Call `cudaDeviceSynchronize()`
   • None of the preceding calls were synchronous
6. Query time between `evStart` and `evEnd`
7. Destroy the events
Asynchronous programming

- Advanced technique
- Asynchronous memory copies (cudaMemcpyAsync)
  - Do not wait until device idle, return immediately
- Kernel launches are already asynchronous with host code
  - Kernel launches can be overlapped with each other and with async memory transfers by using streams
- Best case: Memory transfers can be hidden completely
Asynchronous example

- Double-buffering on GPU side
- Streams to allow concurrent kernel execution and memory transfers
  - Indicates which operations are independent
  - Will skip details now
Digging deeper
What’s in the .o

- Final binaries for multiple architectures
  - In SASS = architecture-specific assembly
  - Known architectures: pick the compiled SASS
- Intermediate-language code for the latest architecture
  - In PTX = architecture-independent assembly
  - Future architectures: driver compiles into SASS at load time

- Both PTX and SASS can be extracted from the object file and examined
PTX vs SASS

PTX is a very simple translation from source
- No optimizations, no register allocation
- Stays fixed between architectures
- Not useful when trying to optimize code

SASS is compiled for a specific architecture
- Exactly what the GPU will run
- Register allocation and all optimizations done
- All ”smartness” in the compilation is in PTX→SASS phase
Extracting PTX and SASS

\texttt{cuobjdump -ptx foo.o}

- Typically only a single PTX representation for each kernel is found

\texttt{cuobjdump -sass foo.o}

- Find the correct SASS for your device
  - Can also extract SASS for a specific architecture using the \texttt{-arch} option
  - E.g., for Quadro K2000, specify \texttt{-arch sm_30}
Documentation

- docs.nvidia.com/cuda
- Main document: Programming Guide
  - Language extensions, special functions you can call in kernels, execution model, etc.
- API reference: CUDA Runtime API
- PTX documentation: PTX ISA
- SASS instruction set: CUDA Binary Utilities → Instruction Set Reference
- Plus a lot more – Best Practices Guide, Tuning Guides, etc.
Thank you

- Questions