Short Circuit

CUDA

Programming Massively Parallel Processors: A Hands-On Approach
by David B. Kirk
Third Edition
Up until Section 5.3

Summary by Emiel Bos

1 Basics

A CPU has only a few sophisticated processors and was always meant for executing sequential programs tasks as fast as possible. This is reflected in the design; arithmetic units and operand data delivery logic minimize the latency of operation at the cost of increased use of chip area and power, and more importantly, CPUs have large cache memories are provided to reduce the instruction and data access latencies (latency-oriented design). In the beginning, single-core processors got faster and faster clock frequencies and operations per tick until 2003, when limits on energy consumption and heat dissipation slowed this down. Therefore, CPUs went multicore. Of course, this means that if software is to be sped up, it needs to be multithreaded.
A GPU takes this concept to the extreme. It is a manycore device, which has tens of thousands of very simple processors, meant for massive data or task parallelization; doing the same simple task many many times in parallel. The large number of execution throughput is the GPUs way of "hiding" memory access latency; the hardware takes advantage of the large number of threads to find work to do when some of them are waiting for long-latency memory accesses or arithmetic operations (throughput-oriented design).
For optimal performance, a program should strive to occupy both the CPU and GPU as much as possible; executing the sequential parts on the CPU and numerically intensive parts on the GPUs. However, this means that when the GPU is maxed out and the CPU is idling, it may be better to move numerically intensive parts to the CPU. Amdahl's law states that the theoretical speedup of a parallel program compared to its single-threaded counterpart is $\frac{1}{(1-p)+\frac{1}{s}}$, where p is the portion of the program that is parallelizable and s is the speedup of that portion of parallelized. As a consequence, parallelizing a program is only worth the time and effort if a significant portion of it is parallelizable.

The GPU was originally developed for graphics, hence the name, but more general uses were quickly discovered, collectively called GPGPU. In the early stages, GPGPU had to be done by using the programmable vertex and fragment shaders for these general calculations and reading back the resulting pixels. After that OpenGL introduced specialized compute shaders (which is used as the only shader in a pipeline, or shader program, in contrast with the regular graphics pipeline). Now, there are dedicated libraries and APIs for GPGPU, so we can thankfully skip the entire graphics interface and forget about doing that with OpenGL. There is the open-source OpenCL, and there is the far better CUDA, with the downside that it is owned by NVIDIA and only works on NVIDIA GPUs.

CUDA is a language extension for multiple languages, like C++, Python, Fortran, OpenCL, OpenACC, OpenMP, and more. but here we'll look at CUDA C. In CUDA terminology, the CPU is the *host* and the GPU is a *device*. A CUDA source file is a mix of host source code – which is just ANSI C – and device source code – for the data parallel functions called *kernels* that are run on the GPU. If the file contains only host code it is basically just a C source file. When device code or other CUDA-specific keywords are added, a regular C compiler can't handle it anymore. Instead, it has to be processed by the NVIDIA C Compiler (NVCC). This compiler separates the file into the host code, which is compiled with the host's standard C compiler, and device code, which is just-in-time compiled by a run-time component of NVCC and run on a GPU.

A C function is designated as a kernel using the `__global__` keyword, which indicates that the function can only be called

by the host and runs on the GPU. Other keywords are `__device__`, which are for kernels that can only be called from other kernels, and `host`, which is for a traditional C function that is called from and runs on the host (and is therefore the default if no other keyword is specified).[1] When the host calls a kernel function, this launches a one, two, or three dimensional *grid* consisting of one, two, or three dimensional equisized (thread) *blocks*[2] consisting of max 1024 threads.[3] The dimensionality of the grid and its blocks usually reflects the data on which the kernel is called (e.g. when processing a greyscale image, both the grid and blocks are two dimensional). The threads take very few clock cycles to generate and schedule due to efficient hardware support, in contrast with CPU threads that take thousands of clock cycles to generate and schedule. All threads launched by a kernel invocation run the same function, hence it is Single Program Multiple Data (SPMD).[4] Block can execute in any order, which is what makes the whole thing scalable and device agnostic.
A kernel is called like `kernelFunction<<< dim_grid, dim_block >>>(args)`, where `dim_grid` and `dim_block` are of C struct type `dim3` containing unsigned int fields `x`, `y` and `z` specifying dimensionalality. Because of how structs in C work, for both `dim_grid` and `dim_block`, we could supply only one int to make the corresponding structure one dimensional.
A kernel running on the GPU has access to some built-in variables: `gridDim`, `blockDim`, `blockIdx` and `threadIdx` are all structs with `x`, `y` and `z` fields giving the specified dimensionalities and indices, respectively. In one dimension, a globally unique index is calculated as `blockIdx.x * blockDim.x + threadIdx.x`. Unfortunately, multidimensional arrays will need to be flattened, because CUDA's ANSI C standard requires kwowing the number of columns at compile time. If a kernel needs to access a pixel index, this is done using `row * row_size + column` for row-major layout. Similarly for three dimensions.

In order to have the threads actually operate on data, we need to dynamically allocate memory on the DDRAM *global/device memory* on the GPU. The host does this by calling `cudaMalloc(void** pointer, int size)`, where the first argument is an address/reference to a pointer (which should be cast to `void**` because the function expects a generic pointer not tied to any type), used to return the location of the allocated memory.[5] Use `cudaMemcpy(*void pointer_dest, *void pointer_src, int size, cudaMemcpyKind direction)` to copy data to the allocated memory (and vice versa), where `direction` is either `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`. We can free the memory on the device again by calling `cudaFree(*void pointer)` with the same pointer.

A `__syncthreads()` in a kernel's code serves as synchronization barrier, at which point threads will wait for all other threads in its block to arrive there. Each `__syncthreads()` is a different barrier. Be cautious with putting a `__syncthreads()` in an if-block, or putting two different `__syncthreads()` in different branches, or else some threads might wait forever. If you do, make sure all threads in a block follow the same path. In order to avoid long waiting times, CUDA runtime systems assign execution resources to all threads in a block as a unit, so that all threads have access to the same resources and are therefore in close temporal proximity of each other.
On host code, call `cudaDeviceSynchronize()` to wait for all blocks of all preceding tasks to finish. It returns an error if one of the tasks failed.

Modern GPUs partition their execution resources into Streaming Multiprocessors (SMs). Each SM is a core of the GPU, and each SM has functional units, including ALUs, memory load/store units, and various special instruction units. Each SM can get a limited number (e.g. 8) of blocks assigned to it, constrained by hardware. Because of resource shortage (e.g. built-in registers for maintaining thread and block indices and tracking their execution status), the total number of threads (e.g. 1536) and the total number of blocks (e.g. 8) that can be assigned to an SM are limited, whichever becomes a limitation first. Often, there are much more blocks in a grid than spots available in SMs. CUDA dynamically partitions and assigns the available thread slots in an SM to as many thread blocks as possible given the block size (and given the number of registers used by the kernel, but that's for later).
A block assigned to an SM is further subdivided into *warps*, which are groups of 32 threads (at the time of writing, this is the number all CUDA devices use) of consecutive `threadIdx` values. Thread blocks are partitioned into warps based on their (row-major linearized) `threadIdx`. The last warp is padded if the number of threads in a block is not a multiple of 32. Warps are the smallest unit of execution/thread scheduling within SMs.[6] An SM executes all threads in a warp according

[1]Both `host` and `device` can be specified for the same function, in which case two object functions will be compiled, one for the host and one for the device.

[2]Block are sometimes also called co-operative thread arrays (CTA).

[3]The thread count should also be a multiples of 32 in each dimensions for hardware efficiency. It is likely that the thread count exceeds the data to be processed, in which case threads need to check whether they're need operation outside the data range, e.g. by wrapping everything in a `if(i < n)` block.

[4]This is not the same as SIMD, where multiple processing units (ALUs) share one control unit (which consists of a program counter (PC) and a instruction register (IR)) and for which all threads need to run the same *instruction* at each tick. So while the instruction is the same, the operand values in the register files differ. This sharing of the control unit saves enormously in manufacturing cost and power consumption, because control units are more expensive than ALUs in those regards.

[5]The C `malloc` function only takes the `size` argument and return the pointer. Using CUDA's two parameter version allows for returning error codes of type `cudaError_t`, which should of course be caught (in case `error != cudaSuccess`).

[6]Warps are not part of the CUDA specification, but it is helpful to know about how they can affect performance.

to the Single Instruction Multiple Data (SIMD) model (whereas the whole block and grid are SPMD).[7] This means that multiple passes are needed when threads *diverge* in their execution path, e.g. one pass for the threads taking the `if` branch and a next pass for all the threads taking the `else` branch. In case of a for-loop, extra passes are needed if some threads loop more often than others. These different passes are sequential and will thus add to the execution; it is therefore best if all threads follow the same path. Diverging execution between threads often occurs due to boundary checking of data, to make sure threads operate on wrong or nonexistent data. The larger this data, the less warps experience divergence among their threads, and the less the performance hit.
Each SM can execute (different) instructions for a small number of warps at any point in time, but there generally are fewer Streaming Processors (SPs) than threads in an SM and an SM can therefore only run a small subset of its warps simultaneously. The benefit of this is that, when a warp is waiting for a long-latency operation (such as a memory access, pipelined floating-point arithmetic, or branch instructions), it gets swapped with another warp that is no longer waiting. This is called *latency tolerence* or *latency hiding*. In other words, while each individual thread's instruction throughput is low and latency high, but the aggregate arithmetic throughput of all SMs together is 5 – 10× higher than typical CPUs. This does need a sufficient number of warps, but doesn't require as much chip area dedicated to cache memories and branch prediction mechanisms as CPUs, allowing more for floating point execution resources. If multiple warps are waiting, a priority mechanism is used.

To query device properties, use `cudaGetDeviceCount(int* count)` to get the number of devices available (including shitty integrated graphics). Each device has an index, and supplying that to `cudaGetDeviceProperties (cudaDeviceProp* properties, int index)` returns the device properties via the `cudaDeviceProp` struct, which contains a bunch of fields holding properties. A few examples are `maxThreadsPerBlock` (and the array `maxThreadsDim` for the max threads per dimension), the array `maxGridSize`, `regsPerBlock` for the number of registers available per block, `sharedMemPer Block`, `multiProcessorCount` for the number of SMs, which gives a good indication of performance together with `clockRate`, and `warpSize`, which should be 32 (at the time of writing).
Some hardware memory constraints that limit the number of threads/warps per SM (meaning less latency hiding): the number of registers (divide `regsPerBlock` by the targeted number of threads per block to determine the registers per kernel available) and the amount of shared memory (divide shared memory per SM by the max number of blocks per SM to determine max shared memory per block to reach max number of blocks per SM (`sharedMemPerBlock`)) available. At the SM level, each SM has a certain number of registers; if this constraint cannot be met, the number of threads need to be reduced at the block granularity.
CUDA dynamically allocates as many blocks as possible to an SM of which the combined register usage (block size × number of registers used by kernel) is lower than the SM's number of registers and shared memory usage is lower than its shared memory.[8]

A kernel can access global memory by pointers passed to them that are obtained by `cudaMalloc` and used in `cudaMemcpy`. The *compute-to-global-memory-access ratio* is the number of floating point operation for each global memory access (in some region of the kernel code), which constrains performance in *memory-bound* programs. If the bandwidth access of the global memory DDRAM is 1 TB/s (which is normal at the time of writing), then a kernel with a compute-to-global-memory-access ratio of 1 will achieve no more than 250 *GFLOPS* (250 billion (giga) floating point operations per second) if single-precision floating point values take four bytes, which is not a lot of you consider that the peak is 12 TFLOPS. In order to increase compute-to-global-memory-access ratio, the GPU has other memory resources that avoid access congestion to global memory and have lower latencies. The following lists which code has what kind of access to which memory using what keyword:

[7]Because a warp is analogously closer to a CPU hardware thread, GPU threads are also sometimes referred to as *lanes*.
[8]NVIDIA offers the CUDA Occupancy Calculator as an Excel sheet: `https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html`

Caller	*Access*	*Memory*	*Scope*	*Lifetime*	*Declaration*
Device code	**R/W**	**Registers**	**Per-thread**	**Kernel**	Automatic scalar variables
Used for frequent, thread-specific variables. On chip. Access is low latency and highly parallel. Most processors have "built-in" register operands, so ALU execute instructions that directly reference register locations. Be careful not to exceed the limited capacity of the registers, which may negatively affect the number of active threads assigned to each SM. A typical register file is 14 MB, in contrast with a few tens of KB on CPU. Threads in a warp can exchange register data using the warp shuffle instruction, enabling warp-wide parallelism and storage.					
Device code	**R/W**	**Local**	**Per-thread**	**Kernel**	Automatic array variables
Automatic array variables are stored in global memory, though their scope is still limited to individual threads.					
Device code	**R/W**	**Shared**	**Per-block**	**Kernel**	`__shared__int sharedVar;`
Efficient for sharing data among threads in a block. On chip, but still requires additional instructions to make operand values available. Access is low latency and highly parallel. Basically a cache for global memory, and analogous to a CPU's L1 cache in terms of speed. A form of *scratchpad memory*. Up to 48 KB per block.					
Device code	**R/W**	**Global**	**Per-grid**	**Application**	`__device__int globalVar;`
Very high latency and low bandwidth, though improved with caches in recent devices, designed to automatically combine accesses to the global memory that are temporally sufficiently close together. Often used to pass data from one kernel invocation to the next,					
Device code	**R**	**Constant**	**Per-grid**	**Application**	
Host code	**R/W**	**Constant**	**Per-grid**	**Application**	`__constant__int constVar;`
Often used for input data to kernel function. Declaration of constant variables must be outside any function body. They are stored in the global memory but cached for efficient access. Limited to 65,537 total bytes per application. Typically 4 – 32 GB in size, with 5 – 10× higher bandwidth than CPU's RAM.					
Host code	**R/W**	**Global**	**Per-grid**	**Application**	`cudaMalloc()`

The `__device__` keyword can also be prepended to the `__shared__` or `__constant__` keywords, but it won't do anything. The scope indicates at what granularity a variable is maintained, e.g. a separate `__shared__` variable is created for and private to each block.

To dynamically allocate a variable amount of shared memory (using e.g. `sharedMemPerBlock`) at runtime, declare this amount as a third configuration parameter of type `size_t` to the kernel launch (e.g. `kernel<<<dimGrid, dimBlock, size>>>(params);`), and use the C `extern` keyword in front of the shared memory declaration and omit the size of the array in the declaration (e.g. `extern __shared__arr[];`).

The global memory's DRAM is super slow, because the bits are stored in small capacitors, where the presence or absence of a tiny electrical charge distinguishes between `0` and `1`, and reading data from a DRAM requires it to drive a highly capacitive line to a sensor and set off its detection mechanism. (Like trying to smell a cup of coffee at the other end of a hallway.) *Tiling* is a program transformation technique that partitions the data into subsets called tiles so that each tile fits into the shared memory and thereby localizes the memory locations accessed among threads and the timing of their accesses. It divides the long access sequences of each thread into phases and uses barrier synchronization to keep the timing of accesses to each section at close intervals. (Of course, this requires the tiles to be able to be processed independently.) A technique to achieve this is called *strip-mining*, which takes a long-running loop and breaks it into phases. Each phase consists of an inner loop that executes a number of consecutive iterations of the original loop. The original loop becomes an outer loop whose role is to iteratively invoke the inner loop so that all the iterations of the original loop are executed in their original order. By adding barrier synchronizations before and after the inner loop, we force all threads in the same block to focus their work entirely on a section of their input data. An example for this is Figure 4.16 on page 91.
Many parallel sensors are in each DRAM chip, so that when a memory location is accessed, a range of its consecutive locations is accessed as well, called DRAM *bursts*. Another technique for ameliorating global memory access is the pattern in which all threads in a warp access consecutive memory locations[9], which the hardware will recognize and *coalesce* all these accesses into one consolidated access to consecutive locations.[10]

[9]Older CUDA devices may impose alignment requirements on the starting location, e.g. the first address needs to be a aligned at a 16-bit word (i.e. the first 6 bits are `0`).

[10]When working with 2D matrices, for example, because multidimensional arrays are stored row-major in C, we have coalesced access when each thread processes elements in one column row-by-row, since all threads process consecutive elements from one row per step. If an algorithm intrinsically

Besides bursts, a processor typically has one to eight *channels*, which is a memory controller that connects with a bus[11] to a set of DRAM *banks*, which contain an array of DRAM cells, the sensing amplifiers for accessing these cells, and the interface for delivering bursts to the bus. The number of banks connected to each channel is the number required to fully utilize its bandwidth. Because the latency of the decoder's cell sensing is much larger than the time for transferring the burst data through the bus, a channel needs a lot of banks to hide this latency, e.g. if the latency-to-transfer-time ratio is r, we need at least $r+1$ banks to fully utilize the bus, so that banks can take turns churning bursts through the bus. However, to reduce the chance of bank conflicts (multiple accesses to the same bank), and because each bank can only provide a limited number of cells (for reasons of reasonable latency and manufacturability), the number of banks often is much higher than r. In order to optimally utilize access bandwidth, a sufficient number of threads must make simultaneously accesses that are evenly spread across channels and banks. *Interleaved data distribution* distributes just enough bytes to each bank to fully utilize its DRAM burst, after which the next bytes are assigned to the same bank in the next channel, i.e. it loops first through channel indices and then through bank indices.

2 Differences between CUDA and OpenGL's compute shaders

CUDA is similar to OpenGL's compute shaders, with some minor differences in terminology and functionality:

CUDA	**OpenGL**
(Thread) block	Work group
Thread	(Compute shader) invocation
Block dimensions defined in host's kernel call	Local size of work group defined in the shader itself
`kernel<<<dim_grid, dim_block>>>(args);`	`glDispatchCompute(num_groups_x, y, z);`
`gridDim`	`in uvec3 gl_NumWorkGroups`
`blockDim`	`const uvec3 gl_WorkGroupSize`
`blockIdx`	`in uvec3 gl_WorkGroupID`
`threadIdx`	`in uvec3 gl_LocalInvocationID`
`blockIdx * blockDim + threadIdx`	`in uvec3 gl_GlobalInvocationID`
Linearized (1D) `threadIdx`	`in uint gl_LocalInvocationIndex`
`__syncthreads()`	`barrier()`
`__shared__`	`shared`
Warp	Subgroup
`cudaDeviceProp.warpSize`	`gl_SubgroupSize`

The dimensions of the number of work groups are given as `GLuint` parameters to the `glDispatchCompute()` function. Alternatively, one could use `glDispatchComputeIndirect(GLintptr indirect)` function, which instead expects the byte-offset to the buffer currently bound to the `GL_DISPATCH_INDIRECT _BUFFER` target containing the work group counts. This indirect dispatch bypasses OpenGL's usual error checking. Compute shaders are not part of the graphics pipeline, so when executing a drawing command, the compute shader linked into the current program or pipeline is not involved. The biggest difference is that the local size of work groups is defined in the compute shader itself, using `layout(local_size_x = X, local_size_y = Y, local_size_z = Z)in;`. By default, the local sizes are 1, so e.g. for a 1D work group size, just omit the Y and Z coordinates. Input data is given via texture access, arbitrary image load, shader storage blocks, or other forms of interface. Similarly, outputs are returned by explicitly writing to an image or shader storage block. As with CUDA's `__syncthreads()`, the evaluations of a `barrier()` must be dynamically uniform. Besides `barrier()`, there is also `memoryBarrierShared()`, which is for `shared` variable ordering, and `groupMemoryBarrier()`, which is a barrier for all incoherent memory operations, but only within a work group [why would you want to sync between work groups anyways?]. A bunch of atomic operations are provided as functions, e.g. `atomicAdd()` and `atomicAnd()`. The maximum number of work groups in dimensions is given by `GL_MAX_COMPUTE_WORK_GROUP_COUNT`, the maximum work group dimensions in terms of invocations is given by `GL_MAX_COMPUTE_WORK_GROUP_SIZE`, the total maximum number of invocations per work group is given by `GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS`, and the maximum size of all `shared` variables if given by `GL_MAX_COMPUTE_SHARED_MEMORY_SIZE`. These must be queried with `glGetIntegeri_v(GLenum target, GLuint axis, GLint* data)`. To do anything related to subgroups in OpenGL, you need to enable one of the relevant GLSL extensions.

requires row-wise operation, a tiling technique called *corner turning* can be used, which has threads first cooperatively load a row into shared memory in a coalesced manner, and then have much faster access to the data in the shared memory.

[11]The data transfer bandwidth of a bus is defined by its width and clock frequency. Modern *double data rate* (DDR) busses perform two data transfers per clock cycle, e.g. a 64-bit DDR bus with a clock frequency of 1 GHz has a bandwidth of $8\text{B} \cdot 2 \cdot 1\text{GHz} = 16\text{GB/sec}$, which is actually too small for modern CPUs and GPUs, which is why more channels are typically needed.