Appendix A

CUDA: *Programming Interface*

A. Introduction

CUDA C/C++ provides a little overhead for users familiar with the C/C++ programming language to easily write programs for execution by the device. It consists of a minimal set of extensions to the C language and a runtime library.

The core language extensions have been introduced in Programming Model (Chapter 2). They allow programmers to define a kernel as a C function and use some new syntax to specify the grid and block dimension each time the function is called. A description of these extensions is given in Appendix B. Any source file that contains some of these extensions must be compiled with nvcc as outlined in next section.

The runtime is introduced in Compilation Workflow. It provides C functions that execute on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc.

The runtime is built on top of a lower-level C API, the CUDA driver API, which is also accessible by the application. The driver API provides an additional level of control by exposing lower-level concepts such as CUDA contexts - the analogue of host processes for the device - and CUDA modules - the analogue of dynamically loaded libraries for the device. Most applications do not use the driver API as they do not need this additional level of control and when using the runtime, context and module management are implicit, resulting in more concise code.

A.1. Compilation with NVCC

Kernels can be written using the CUDA instruction set architecture, called *PTX*. It is however usually more effective to use a high-level programming language such as C/C++. In both cases, kernels must be compiled into binary code by **`nvcc`** to execute on the device.

`nvcc` is a compiler driver that simplifies the process of compiling CUDA C/C++ or PTX code: It provides simple and familiar command line options and executes them by invoking the collection of tools that implement the different compilation stages. This section gives an overview of `nvcc` workflow and command options.

A.1.1. Compilation Workflow

- Offline Compilation

Source files compiled with `nvcc` can include a mix of host code (i.e., code that executes on the host) and device code (i.e., code that executes on the device). `nvcc`'s basic workflow consists in separating device code from host code and then:

- compiling the device code into an assembly form (PTX code) and/or binary form (`cubin` object),
- and modifying the host code by replacing the `<<<...>>>` syntax by the necessary CUDA C runtime function calls to load and launch each compiled kernel from the *PTX* code and/or `cubin` object.

The modified host code is output either as C code that is left to be compiled using another tool or as object code directly by letting `nvcc` invoke the host compiler during the last compilation stage. Applications can then:

- Either link to the compiled host code (this is the most common case),
- Or ignore the modified host code (if any) and use the CUDA driver API to load and execute the *PTX* code or `cubin` object.

- Just-in-Time Compilation

Any *PTX* code loaded by an application at runtime is compiled further to binary code by the device driver. This is called *just-in-time compilation*. Just-in-time compilation increases application load time, but allows the application to benefit from any new compiler improvements coming with each new device driver. It is also the only way for applications to run on devices that did not exist at the time the application was compiled.

When the device driver just-in-time compiles some *PTX* code for some application, it automatically caches a copy of the generated binary code in order to avoid repeating the compilation in subsequent invocations of the application. The cache - referred to as *compute cache* - is automatically invalidated when the device driver is upgraded, so that applications can benefit from the improvements in the new just-in-time compiler built into the device driver.

Environment variables are available to control and support just-in-time compilation.

A.1.2. Compatibility

- Binary Compatibility

Binary code is architecture-specific. A `cubin` object is generated using the compiler option `-code` that specifies the targeted architecture: For example, compiling with `-code=sm_35` produces binary code for devices of compute capability 3.5. Binary compatibility is guaranteed from one minor revision to the next one, but not from one minor revision to the previous one or across major revisions. In other words, a `cubin` object generated for compute capability X.y will only execute on devices of compute capability X.z where $z \geq y$.

- PTX Compatibility

Some *PTX* instructions are only supported on devices of higher compute capabilities. For example, the warp shuffle instructions are only supported on devices of compute capability 3.0 and above. The `-arch` compiler option specifies the compute capability that is assumed when compiling C to *PTX* code. So, code that contains these type of instructions must be compiled with `-arch=sm_30` (or higher).

PTX code produced for some specific compute capability can always be compiled to binary code of greater or equal compute capability.

- Application Compatibility

To execute code on devices of specific compute capability, an application must load binary or *PTX* code that is compatible with this compute capability. In particular, to be able to execute code on future

architectures with higher compute capability (for which no binary code can be generated yet), an application must load *PTX* code that will be just-in-time compiled for these devices.

The source code can have an optimized code path that uses warp shuffle operations, for example, which are only supported in devices of compute capability 3.0 and higher. The `__CUDA_ARCH__` macro can be used to differentiate various code paths based on compute capability. It is only defined for device code. When compiling with `-arch=compute_35` for example, `__CUDA_ARCH__` is equal to 350.

Applications using the driver API must compile code to separate files and explicitly load and execute the most appropriate file at runtime.

- C/C++ Compatibility

The front end of the compiler processes CUDA source files according to C++ syntax rules. Full C++ is supported for the host code. However, only a subset of C++ is fully supported for the device code. Some restrictions are: it does not support run time type information (RTTI), exception handling, and the full C++ Standard Library.

- 64-Bit Compatibility

The 64-bit version of `nvcc` compiles device code in 64-bit mode (i.e., pointers are 64-bit). Device code compiled in 64-bit mode is only supported with host code compiled in 64-bit mode. Similarly, the 32-bit version of nvcc compiles device code in 32-bit mode and device code compiled in 32-bit mode is only supported with host code compiled in 32-bit mode.

The 32-bit version of `nvcc` can compile device code in 64-bit mode also using the `-m64` compiler option. The 64-bit version of `nvcc` can compile device code in 32-bit mode also using the `-m32` compiler option.

A.2. CUDA C Runtime

The runtime is implemented in the `cudart` library, which is linked to the application, either statically via `cudart.lib` or `libcudart.a`, or dynamically via `cudart.dll` or `libcudart.so`. Applications that require `cudart.dll` and/or `cudart.so` for dynamic linking typically include them as part of the application installation package.

A.2.1. Initialization

There is no explicit initialization function for the runtime; it initializes the first time a runtime function is called (more specifically any function other than functions from the device and version management). One needs to keep this in mind when timing runtime function calls and when interpreting the error code from the first call into the runtime.

During initialization, the runtime creates a **CUDA context** for each device in the system. This context is the primary context for this device and it is shared among all the host threads of the application. As part of this context creation, the device code is just-in-time compiled if necessary and loaded into device memory. This all happens under the hood and the runtime does not expose the primary context to the application.

When a host thread calls `cudaDeviceReset()`, this destroys the primary context of the device the host thread currently operates on (i.e., the current device). The next runtime function call made by any host thread that has this device as current will create a new primary context for this device.

A.2.2. Device Memory

As mentioned in Heterogeneous Programming (section 2.4), the CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Kernels operate out of device memory, so the runtime provides functions to allocate, deallocate, and copy device memory, as well as transfer data between host memory and device memory.

Device memory can be allocated either as linear memory or as CUDA arrays.

Linear memory exists on the device in a 40-bit address space, so separately allocated entities can reference one another via pointers, for example, in a binary tree.

Linear memory is typically allocated using `cudaMalloc()` and freed using `cudaFree()` and data transfer between host memory and device memory are typically done using `cudaMemcpy()`.

Linear memory can also be allocated through `cudaMallocPitch()` and `cudaMalloc3D()`. These functions are recommended for allocations of 2D or 3D arrays as it makes sure that the allocation is appropriately padded to meet the alignment requirements, therefore ensuring best performance when accessing the row addresses or performing copies between 2D arrays and other regions of device memory (using the `cudaMemcpy2D()` and `cudaMemcpy3D()` functions). The returned pitch (or stride) must be used to access array elements.

There are also `cudaMemcpyToSymbol()` and `cudaMemcpyFromSymbol()` to access constant and device variables that declared in global scope.

Programmer should free the allocated memory by using functions like `cudaFree()`.

CUDA arrays are opaque memory layouts optimized for texture fetching. They are referred as Texture and Surface Memory.

A.2.3. Shared memory

The shared memory is allocated using the `__shared__` qualifier. Shared memory is expected to be much faster than global memory. Any opportunity to replace global memory accesses by shared memory accesses should therefore be exploited.

The shared memory has limited size; therefore, the programmer has to decompose the task/data so it can benefit from the shared memory. One can think of this as cache memory to increase speed of computation.

A.2.4. Page-locked host memory

The runtime provides functions to allow the use of page-locked (also known as pinned) host memory (as opposed to regular pageable host memory allocated by `malloc()`)

Portable, Write Combining, Mapped are the types of page-locked memory.

A.2.5. Asynchronous Concurrent Execution

- Concurrent Execution between Host and Device

In order to facilitate concurrent execution between host and device, some function calls are asynchronous: Control is returned to the host thread before the device has completed the requested task. These are:

- Kernel launches;
- Memory copies between two addresses to the same device memory;
- Memory copies from host to device of a memory block of 64 KB or less;
- Memory copies performed by functions that are suffixed with `Async`;
- Memory set function calls.

Programmers can globally disable asynchronous kernel launches for all CUDA applications running on a system by setting the `CUDA_LAUNCH_BLOCKING` environment variable to `1`. This feature is provided for debugging purposes only and should never be used as a way to make production software run reliably.

Kernel launches are synchronous if hardware counters are collected via a profiler (Nsight, Visual Profiler).

- Overlap of Data Transfer and Kernel Execution

Some devices can perform copies between page-locked host memory and device memory concurrently with kernel execution. Applications may query this capability by checking the `asyncEngineCount` device property, which is greater than zero for devices that support it.

- Concurrent Kernel Execution

Some devices of compute capability 2.x and higher can execute multiple kernels concurrently. Applications may query this capability by checking the `concurrentKernels` device property, which is equal to `1` for devices that support it.

The maximum number of kernel launches that a device can execute concurrently is 16 on devices of compute capability 2.0 through 3.0; the maximum is 32 concurrent kernel launches on devices of compute capability 3.5 and higher. Devices of compute capability 3.2 are limited to 4 concurrent kernel launches.

A kernel from one CUDA context cannot execute concurrently with a kernel from another CUDA context.

Kernels that use many textures or a large amount of local memory are less likely to execute concurrently with other kernels.

- Concurrent Data Transfers

Some devices of compute capability 2.x and higher can perform a copy from page-locked host memory to device memory concurrently with a copy from device memory to page-locked host memory. Applications may query this capability by checking the `asyncEngineCount` device property, which is equal to 2 for devices that support it.

- Streams

Applications manage concurrency through streams. A stream is a sequence of commands (possibly issued by different host threads) that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behaviour is not guaranteed and should therefore not be relied upon for correctness (e.g., inter-kernel communication is undefined).

- Explicit Synchronization

There are various ways to explicitly synchronize streams with each other. The call `cudaDeviceSynchronize()` waits until all preceding commands in all streams of all host threads have completed. `cudaStreamSynchronize()` takes a stream as a parameter and waits until all preceding commands in the given stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device.

`cudaStreamWaitEvent()` takes a stream and an event as parameters and makes all the commands added to the given stream after the call to `cudaStreamWaitEvent()` delay their execution until the given event has completed. The stream can be 0, in which case all the commands added to any stream after the call to `cudaStreamWaitEvent()` wait on the event.

`cudaStreamQuery()` provides applications with a way to know if all preceding commands in a stream have completed.

To avoid unnecessary slowdowns, all these synchronization functions are usually best used for timing purposes or to isolate a launch or memory copy that is failing.

- Implicit Synchronization

Two commands from different streams cannot run concurrently if any one of the following operations is issued in-between them by the host thread:

- a page-locked host memory allocation,
- a device memory allocation,
- a device memory set,
- a memory copy between two addresses to the same device memory,
- any CUDA command to the NULL stream,
- a switch between the L1/shared memory configurations described in Compute Capability 2.x and Compute Capability 3.x.

Synchronization of any kind should be delayed as long as possible.

- Callbacks

The runtime provides a way to insert a callback at any point into a stream via `cudaStreamAddCallback()`. A callback is a function that is executed on the host once all commands issued to the stream before the callback have completed. Callbacks in stream 0 are executed once all preceding tasks and commands issued in all streams before the callback have completed.

A callback must not make CUDA API calls (directly or indirectly), as it might end up waiting on itself if it makes such a call leading to a deadlock.

- Stream Priorities

The relative priorities of streams can be specified at creation using `cudaStreamCreateWithPriority()`. The range of allowable priorities, ordered as [highest priority, lowest priority] can be obtained using the `cudaDeviceGetStreamPriorityRange()` function. At runtime, as blocks in low-priority schemes finish, waiting blocks in higher-priority streams are scheduled in their place.

- Events

The runtime also provides a way to closely monitor the device's progress, as well as perform accurate timing, by letting the application asynchronously record events at any point in the program and query when these events are completed. An event has completed when all tasks - or optionally, all commands in a given stream - preceding the event have completed. Events in stream zero are completed after all preceding tasks and commands in all streams are completed.

This can be used to measure elapsed time but a portion of the code.

- Synchronous Calls

When a synchronous function is called, control is not returned to the host thread before the device has completed the requested task. Whether the host thread will then yield, block, or spin can be specified by calling `cudaSetDeviceFlags()` with some specific flags before any other CUDA call is performed by the host thread.

- Error Checking

All runtime functions return an error code, but for an asynchronous function, this error code cannot possibly report any of the asynchronous errors that could occur on the device since the function returns before the device has completed the task. The error code only reports errors that occur on the host prior to executing the task, typically related to parameter validation. If an asynchronous error occurs, it will be reported by some subsequent unrelated runtime function call.

The only way to check for asynchronous errors just after some asynchronous function call is therefore to synchronize just after the call by calling `cudaDeviceSynchronize()` (or by using any other synchronization mechanisms described above) and checking the error code returned by `cudaDeviceSynchronize()`.

The runtime maintains an error variable for each host thread that is initialized to `cudaSuccess` and is overwritten by the error code every time an error occurs (be it a parameter validation error or an asynchronous error). `cudaPeekAtLastError()` returns this variable. `cudaGetLastError()` returns this variable and resets it to `cudaSuccess`.

Kernel launches do not return any error code, so one of the above functions must be called just after the kernel launch to retrieve any pre-launch errors. To ensure that any error returned by `cudaPeekAtLastError()` or `cudaGetLastError()` does not originate from calls prior to the kernel launch, one has to make sure that the runtime error variable is set to `cudaSuccess` just before the kernel launch, for example, by calling `cudaGetLastError()` just before the kernel launch. Kernel launches are asynchronous, so to check for asynchronous errors, the application must synchronize in-between the kernel launch and the call to `cudaPeekAtLastError()` or `cudaGetLastError()`.

Note that `cudaErrorNotReady` that may be returned by `cudaStreamQuery()` and `cudaEventQuery()` is not considered an error and is therefore not reported by `cudaPeekAtLastError()` or `cudaGetLastError()`.

- Call Stack

On devices of compute capability 2.x and higher, the size of the call stack can be queried using `cudaDeviceGetLimit()` and set using `cudaDeviceSetLimit()`.

When the call stack overflows, the kernel call fails with a stack overflow error if the application is run via a CUDA debugger (cuda-gdb, Nsight) or an unspecified launch error, otherwise.

A.3. Hardware Implementation

The NVIDIA GPU architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called **SIMT** (Single-Instruction, Multiple-Thread). The instructions are pipelined to leverage instruction-level parallelism within a single thread, as well as thread-level parallelism extensively through simultaneous hardware multithreading. Unlike CPU cores, they are issued in order however and there is no branch prediction and no speculative execution.

The NVIDIA GPU architecture uses a little-endian representation.

A.3.1. SIMT Architecture

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called **warps**. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. The term warp originates from weaving, the first parallel thread technology. A *half-warp* is either the first or second half of a warp. A *quarter-warp* is either the first, second, third, or fourth quarter of a warp.

When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

The SIMT architecture is akin to SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behaviour of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behaviour; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: Cache line size can be safely ignored when

designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

The threads of a warp that are on that warp's current execution path are called the active threads, whereas threads not on the current path are inactive (disabled). Threads can be inactive because they have exited earlier than other threads of their warp, or because they are on a different branch path than the branch path currently executed by the warp, or because they are the last threads of a block whose number of threads is not a multiple of the warp size.

If a non-atomic instruction executed by a warp writes to the same location in global or shared memory for more than one of the threads of the warp, the number of serialized writes that occur to that location varies depending on the compute capability of the device and which thread performs the final write is undefined.

If an atomic instruction executed by a warp reads, modifies, and writes to the same location in global memory for more than one of the threads of the warp, each read/modify/write to that location occurs and they are all serialized, but the order in which they occur is undefined.

A.3.2. Hardware Multithreading

The execution context (program counters, registers, etc.) for each warp processed by a multiprocessor is maintained on-chip during the entire lifetime of the warp. Therefore, switching from one execution context to another has no cost, and at every instruction issue time, a warp scheduler selects a warp that has threads ready to execute its next instruction (the active threads of the warp) and issues the instruction to those threads.

In particular, each multiprocessor has a set of 32-bit registers that are partitioned among the warps, and a parallel data cache or shared memory that is partitioned among the thread blocks.

The number of blocks and warps that can reside and be processed together on the multiprocessor for a given kernel depends on the amount of registers and shared memory used by the kernel and the amount of registers and shared memory available on the multiprocessor. There are also a maximum number of resident blocks and a maximum number of resident warps per multiprocessor. These limits as well the amount of registers and shared memory available on the multiprocessor are a function of the compute capability of the device and are given in Appendix C. If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch.