CUDA and Applications to Task-based Programming

B. Kerbl, M. Kenzel, M. Winter, and M. Steinberger

Reims 2022

In this first part of the tutorial, we will give a quick overview of the history of the GPU, followed by an introduction to CUDA and how to set up basic CUDA applications. Afterward, we will consider the CUDA execution model and how it maps to the underlying hardware architecture, followed by a few examples for writing CUDA code and first steps towards performance optimization.

Tutorial Resources

- We have prepared course notes and code samples for this tutorial
  - Course notes include presented slides with additional notes
  - Code samples include buildable versions of the code in the tutorial
    - If you want to try them as you go along, feel free to fetch and build them now
    - Important: CMake 3.20 required!

- May also be extended in the future to stay up to date ☺

- Find them here: https://cuda-tutorial.github.io

Tutorial Presenters
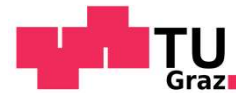
Bernhard Kerbl

TU Wien

Institute for Visual Computing
and Human-Centered Technology

Markus Steinberger

Graz University of Technology
Institute for Computer Graphics and Vision

Michael Kenzel

DFKI Saarbrücken

Martin Winter

3

History of the GPU and CUDA

From 2D blitters to pure parallel co-processors

Reims 2022

The history of the GPU, even though it started somewhat recently, describes a fast-moving stream of advancements and improvements, which turned the initial 2D blitting devices into massively parallel, general-purpose processors.

Evolution of the GPU in a Nutshell

1987 - Commodore Amiga, **2D Blitter** („bit block transfer), 4096 colors

1996 - 3dfx Voodoo1, **triangle rasterization**, 500 Mhz, 4MB RAM

1999 - NVIDIA GeForce 256, **transform-and-lighting**, 120 Mhz

2001 - NVIDIA GeForce 3, **vertex and fragment shaders**, 200 Mhz

2006 - NVIDIA GeForce 8, **compute shaders**, 1500 Mhz, 576 GFLOPs

2009 - ATI Radeon HD 5000, **tessellation**, 850 Mhz, 2720 GFLOPs

2017 - NVIDIA Titan V, **tensor cores**, 1.2 GHz, 12 TFLOPs

2018 - NVIDIA Geforce 2080, **task shaders**, **ray-tracing**, 1.5 Ghz, 14 TFLOPs

The blitter, which is a portmanteau of „bit", „block" and „transfer", was featured in the Amiga with fixed resolution and 4096 colors. These cards had no 3D functionality, only the ability to combine and output different 2D color information. The first 3D capabilities for the wider consumer market arrived with 3dfx and the Voodoo 1, which would be installed alongside already running 2D graphics cards to extend machines with 3D functionality (3D accelerators). These accelerators would take care of rasterization only, so geometry processing would still occur on the CPU. With the GeForce 256, GPUs were now capable of doing both 2D and 3D with a single piece of hardware, and the basic geometry process for 3D content, transformation and lighting, was moved from the CPU to the GPU as well. Shortly after, we saw the introduction of vertex and fragment shaders, that is, the first example of programmable consumer-grade GPUs.

These abilities to execute custom code on a parallel device were quickly exploited by crafty developers, who would compute complex simulations by feeding arbitrary „vertex" data and interpreting pixel color outputs as results with improved performance. Luckily, the vendors eventually responded to these trends and make the exploitation of the GPU's parallel processing more convenient with the introduction of the unified shader model and compute shaders. Most recently, the developments of the GPU indicate an interesting trend: developers are given more options for programmability of the graphics and processing pipeline, and some fixed functions are either removed or made configurable. At the same time, the most common operations are facilitated by specialized hardware modules that can accelerate them over pure software implementations. The GPU today is, therefore, becoming more general and more specialized at the same time.

- Ca. 1970 – 2003: The Free Performance Lunch
  - Ability to increase transistor count no longer maps to performance gain
  - Performance of already-written code no longer increases on its own

- Three walls (as defined by D. Patterson at UC Berkeley)
  - Power wall: Cooling expanses not economized by additional performance
  - Memory wall: Multiple fast cores are bottlenecked by slow main memory
  - ILP wall: There is only so much prediction and pipelining you can do

- Maintain growth with parallel architectures and programming paradigms!

These changes are strongly motivated by several roadblocks that conventional, CPU-side execution is facing. Around 2003, it became apparent that CPU performance no longer increases as time goes by since further optimizations appear to hit one of three walls: either the power wall, where raising a CPU's clock rate is no longer feasible or safe or the memory wall, which implies that even on multi-core systems, collaborative computations will be bottlenecked by slow main memory or lastly the ILP (instruction-level parallelism) wall, which tells us that branch prediction and machine code analysis can only do as much optimization as the program flow allows. Thus, in order to maintain growing performance for processing, the hardware, paradigms, and programming patterns with which we approach problems have changed in favor of massively parallel processing.

Today: GPUs Without Graphics

Reims 2022

- Pure compute power for massively-parallel co-processors

- Designed for machine learning, data centers

- E.g.: NVIDIA Tesla/Volta V100, Ampere A100

- No rasterization engines, no display output
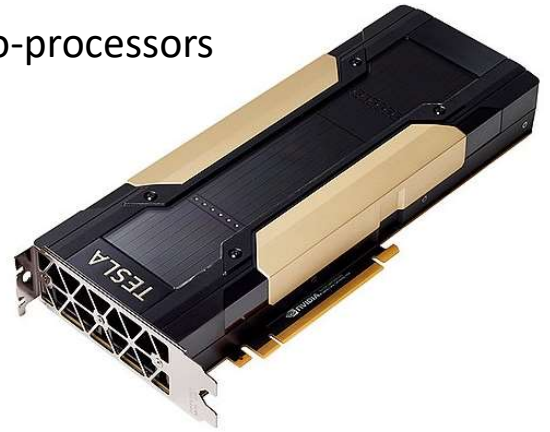
Wikimedia Commons, NVIDIA TESLA V100. CC-BY-SA-4.0:
https://creativecommons.org/licenses/by-sa/4.0/deed.en

Today's GPUs provide an answer to this demand for consumers, developers, and researchers alike. The benefits of their raw compute power for applications like machine learning, off-line rendering, data science, physics simulations and many more have given rise to extremely powerful hardware models like the V100 or the A100 which, despite being called GPUs, no longer feature a display port: these developments reflect how the ability to produce real-time graphics has in many cases become secondary.

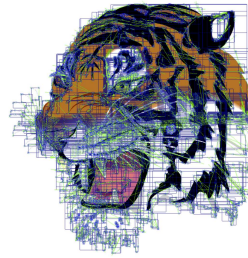# The GPU: A Parallel Powerhouse

- Vast range of applications
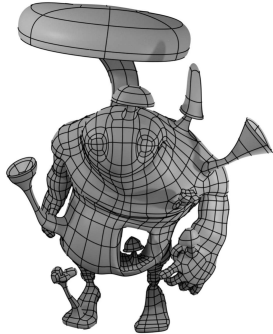
Geometry Subidivison / Manipulation

Medical Imaging

Physically-based Simulations

Rasterization

Procedural Content Generation

$$\mathbf{A} = \begin{bmatrix} \cdot & 5 & \cdot & 9 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ 2 & 3 & 6 & \cdot & 3 & 6 & \cdot & 3 & 6 & 3 \\ \cdot & \cdot & 7 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 4 & 8 & 1 & \cdot & \cdot & \cdot & 5 & 7 \end{bmatrix}$$
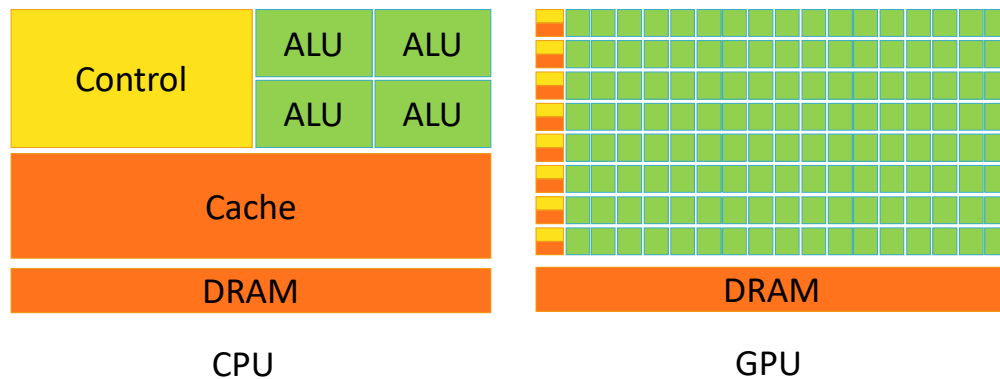
Parallel Matrix Operations / Machine Learning

CPU vs. GPU Architectural Properties

Reims 2022

- Architecture design dictates programming paradigms for both

Let us quickly compare the CPU and GPU architecture in broad strokes. The CPU is a latency-oriented design, meaning it will attempt to receive the result of computations as quickly as possible. For this purpose, it features large L1 caches to reduce the average latency of data and only requires a few, high-performance arithmetic logic units to quickly compute results. Today's models will also make heavy use of instruction-level parallelism to compute partial results ahead of time to further reduce latency. The GPU design, on the other hand, is throughput-oriented. Due to the vast number of parallel processors it contains, it cannot provide L1 caches for each of them with a size similar to the CPU. Memory accesses are therefore more likely to go to slower memory types, which incurs latency. However, if the GPU is "over-subscribed" with threads, that is, it runs significantly more threads than physical cores, it can hide these latencies by quickly switching execution between those threads.

GPU threads are in general more lightweight than CPU threads, which makes switching between them more efficient. Even though latencies may be higher, the ability to switch threads and pipeline additional instructions quickly ensures that the GPU can achieve a high throughput during the execution of a job. Hence, the payoff from using GPUs for processing can rise the more threads are being used for a given compute job.

- **C**ompute **U**nified **D**evice **A**rchitecture, first SDK in February of 2007

- Describes full architecture, encapsulates three APIs
  - Driver API
  - **Runtime API**
  - Device Runtime API

- Driver API is a superset of runtime API and can be mixed freely with it

```
// Runtime API:
int* a;
cudaMalloc(&a, 4);
cudaMemcpy(a, c, 4, cudaMemcpyHostToDevice);
cudaDeviceSynchronize();

// Driver API:
CUdeviceptr b;
cuMemAlloc(&b, 4);
cuMemcpyHtoD(b, c, 4);
cuCtxSynchronize();
```

The Compute Unified Device Architecture, or CUDA for short, defines hardware standards and several APIs to perform high-performance computing on GPUs in parallel. The three APIs it includes are the driver API, the runtime API, and the device runtime API. Since it is easiest to get used to and used in most teaching materials, we will be focussing on the runtime API in this tutorial. However, the use of the driver API is not much more difficult, and it provides a strict superset of the runtime API in terms of functionality, with a few additional advanced features.

- Parallel execution GPUs can be performed through a variety of APIs: CUDA, OpenCL, DirectX, OpenGL, Vulkan, Mantle…

- Each define their own terminology for components and techniques
  - Easily can be confusing, attempts for vendor/API "dictionaries" exist[2]
  - Focusing on CUDA, we will employ the associated terminology

- Examples:
  - "device" for CUDA-capable parallel processor (NVIDIA GPU)
  - "host" for architecture that controls devices (usually CPU)

Before we get started with CUDA, we must note that the terminology being used in materials is often vendor-specific. This complicates things slightly when we try to communicate common concepts that you may already know from other APIs or architectures because many of them are given another name by different vendors. Some attempts at making corresponding dictionaries exist, but we will try to make an effort here to introduce each of the concepts with basic descriptions and illustrations, and hopefully you will be able to establish the connections yourself. The first piece of terminology that is common to CUDA is the separation of platforms where code is executed. This can be either the device, which represents a CUDA capable parallel graphics processing unit, or the host, which communicates with the device via the runtime or driver API, usually the CPU.

- Programming Convenience
  - Call stacks, heap memory, pointers!
  - Strong support for modern C++ features (e.g., template meta-programming)
  - Code reuse between host and device, standard library `cuda::std`
  - Vast range of well-maintained libraries for frequent use cases
  - Basic compute pipeline setup with only 5 lines of C++ code
  - …

- Ahead of the curve: cutting-edge NVIDIA hardware features are often available in CUDA first (although porting speed has been increasing)

25.04.2022   CUDA and Applications to Task-based Programming   12

A valid question is why you should care about CUDA in particular, given that by now, there is a large list of frameworks and libraries that handle processing on the GPU for you, while low-level graphics APIs can provide direct access to the GPU's compute capabilities via compute shaders or similar concepts. However, a strong point of CUDA over other low-level approaches is the combination of both. For developers, it is more convenient to write CUDA applications over computer shaders, since CUDA is continuously improving its support for the C++ standard. Furthermore, CUDA comes with a collection of ready-to-use libraries for common use cases. At the same time, low-level GPU functionality is often exposed by CUDA first, ahead of their adoption in other vendor-agnostic APIs yet. Hence, CUDA can offer you a versatile approach to GPU programming: convenient, high-level functionality with libraries, high-performance with low-level instructions, and a convenient approach to managing your codebase between different architectures.

History in the Making

- **_Volta_** marks a turning point for many aspects of GPU programming

- In the last few years, CUDA functionality has drastically expanded
  - Some changes are obvious and related to general hardware trends
  - Others are more subtle and specific to the CUDA environment

- Disclaimer: Some of our code samples today are non-optimal
  - Not because they are wrong or deprecated, but because other options exist
  - Fundamental patterns can be better realized with recent features
  - We will revisit them tomorrow when we discuss novel CUDA capabilities

The history of the GPU is not over. In the last few years, the GPU architecture has arguably undergone its most transformative era, introducing the ability to perform ray tracing and machine learning directly in hardware. However, these features may have overshadowed some of the less spectacular changes, which are nonetheless important. In this tutorial, we will try to introduce first the fundamentals of CUDA. During this part, we will adhere to the basics and the legacy commands that are also heavily featured in the CUDA programming guide. However, it should be noted that the paradigms for programming in CUDA are shifting towards a clearly defined, cleaner coding style, enabled by newly introduced features. Thus, the code samples shown today should be taken with a grain of salt: they are meant to illustrate the features and common patterns for using CUDA, but developers who are interested in writing stable and portable code should strive to replace these concepts with more recent alternatives, which we will be introducing in the third part of this tutorial, after discussing the underlying hardware details in part 2.

Getting Started

Environments, Guidelines, Compilers and Debuggers

Reims 2022

Before we can write CUDA applications, there are a few requirements that we need to fulfill first.

- CUDA Toolkit

- Classical (full control over kernel design)
  - C++ build environment
  - PyCUDA

- Python-centric
  - Numba (parallel GPU code from Python)
  - Pyculib (library bindings)

Initially, we need to decide which method of using CUDA is most suitable for us. CUDA is available in many shapes in forms, for instance, it can be accessed via a C++ build environment or via Python. Any use of CUDA will require the installation of the CUDA toolkit first. If you choose to go with Python, you may use low-level libraries like PyCUDA, which enable you to follow the instructions in the CUDA programming guide more closely, or solutions like Numba, paired with Pyculib, which abstract most of the implementation details for the purpose of number crunching.

- C++ build environment (e.g., Microsoft Visual Studio with CUDA 11)

- CUDA Toolkit/Driver: https://developer.nvidia.com/cuda-downloads

- Nsight Systems: https://developer.nvidia.com/nsight-systems

- Nsight Compute: https://developer.nvidia.com/nsight-compute

However, in order to be able to closely control GPU code generation, exploit low-level features at will and follow the most common teaching materials, we will be providing all code samples and application scenarios in a C++ environment. In order to follow along, recreate or experiment with the examples, you will need a C++ build environment. Setting up CUDA projects can be done for instance with CMake for maximum portability, but it is also easy to set up Visual Studio projects with correct linked libraries set from the project creation wizard once the CUDA toolkit and driver are installed. In addition to the toolkit, we also strongly advise that you get Nsight Systems and Nsight Compute, or equivalent solutions for debugging and profiling if you are using older hardware.

Source Files and Compilation

- CUDA/C++ source files, commonly identified by `.cu` extension

- Source can contain code for execution on both host and device

- Separate compilation performed by NVIDIA CUDA Compiler (NVCC)

- E.g., compile CUDA source file foo.cu: `nvcc foo.cu -o foo`

In general, we will be writing CUDA code in files that are considered by the NVIDIA CUDA compiler, or NVCC for short. The source files use, by convention, the extension .cu. Within these code files, it is possible to mix GPU and CPU code. The proper division of the source into host and device functions is performed by the NVCC, which compiles them separately and unites them in an executable. This behavior can, for instance, be hidden behind an IDE like Visual Studio or a make file for convenience. Furthermore, there many alternative workflows that the NVCC supports, such as producing CUDA binaries or machine code for specific architectures. If you are interested in the different ways in which compilation and linking can be performed in more complex setups, please refer to the NVCC manual for documentation.

Code Samples for CUDA with CMake

- Growing support for CUDA projects!

- Significant improvements since 3.17
  - Important flags clearly defined
    - Previously: CUDA_NVCC_FLAGS
    - Error-prone, not always portable, challenging to debug or adapt
  - Convenience features
    - Variables for CUDA Toolkit, targeted architectures
    - E.g., CMake 3.20: automatically detects default GPU architecture that NVCC builds for

```
1  cmake_minimum_required(VERSION 3.20)
2
3  set(CMAKE_CXX_STANDARD 17)
4  set(CMAKE_CXX_EXTENSIONS OFF)
5  set(CMAKE_CUDA_STANDARD 17)
6
7  project(TutorialSamples LANGUAGES CUDA CXX)
8
9  add_subdirectory(01_HelloGPU)
```

```
1  add_executable(01_HelloGPU
2      src/main.cu
3  )
4
5  target_include_directories(01_HelloGPU PRIVATE
6  ${CMAKE_CUDA_TOOLKIT_INCLUDE_DIRECTORIES})
```

18

Recommended Resources

- CUDA Programming Guide
- CUDA API Reference Manual
- PTX Instruction Set Architecture

  } Essential reading

- CUDA Compiler Driver NVCC } Building executables

- CUDA-MEMCHECK
- Nsight Documentation
- Kernel Profiling Guide

  } Debugging & profiling

- NVIDIA Developer Forums } Clarifications, explanations, intricate details

Lastly, it is vital to know where to get your information. We recommend that, if you want to obtain a detailed understanding of not only how, but why the CUDA architecture can achieve the performance that it does, you consider the resources provided on this slide. The programming guide, the API reference manual and the PTX ISA are essential reading for anybody who wants a deeper understanding of the architecture. In addition, there are detailed manuals for the most useful tools, and the information in there often complements parts that may be missing in the essential reading documents. Lastly, if things are still unclear after consulting all of these resources, the NVIDIA developer forums are a fantastic resource for getting highly specific questions answered from other members of the GPU programming community or even professionals.

The CUDA Execution Model

Reims 2022

Let us now take a first look at how the CUDA architecture handles the execution of code in parallel.

Kernel Functions and Device Functions

- Kernel functions may be called directly from host
  - Launch configuration, parameters (built-in types, structs, pointers)
  - Indicated by `__global__` qualifier for functions
  - Cannot return values, must be of type void

- Device functions may only be called from kernels or device functions
  - No launch configuration, parameters from kernels or device functions
  - Indicated by `__device__` qualifier for functions
  - Support arbitrary return types, recursion

When we write code for the GPU with CUDA, we can distinguish `__global__` and `__device__` functions. The former signify so-called kernel functions, which may be invoked straight from the host and must not have a return value other than void. The latter are functions that may only be called from functions already running on the device, such as kernels or other `__device__` functions.

- Basic kernel, launched with distinct `<<<grid,block>>>()` syntax

```cpp
__global__ void HelloWorldGPU()
{
    printf("Hello, world, from the GPU!\n");
}

int main()
{
    HelloWorldGPU<<<1,12>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

```
Microsoft Visual Studio-Debugging-Konsole        —    □    ×
Hello, world, from the GPU!
Hello, world, from the GPU!
Hello, world, from the GPU!
Hello, world, from the GPU!
Hello, world, from the GPU!
Hello, world, from the GPU!
Hello, world, from the GPU!
Hello, world, from the GPU!
Hello, world, from the GPU!
Hello, world, from the GPU!
Hello, world, from the GPU!
Hello, world, from the GPU!
```

- Kernel launches are *asynchronous* to host execution
  - Does that mean we always need the synchronization towards the end?

With this knowledge, and the addition that CUDA supports printing to the console, it is extremely simple to write an initial kernel that proves to us that, it is in fact, running in parallel on the GPU. Note the characteristic syntax for calling a `__global__` function from a standard C++ CPU-side function, which defines the launch configuration, or „grid" of threads that the compute job should use. This syntax will later be replaced by the NVCC with explicit function calls to run GPU code with the given parameters. Here, we launch a total of 12 threads, each of which will print a fixed message. Eventually, in this short example we also call a CUDA function before the program terminates, called `cudaDeviceSynchronize`. This may give the initial impression that, like in other APIs like Vulkan, manual synchronization is frequently required, but this is actually not the case.

CUDA Command Execution

- Some CUDA commands are asynchronous with regard to the **host**, but not concurrent to **each other** (unless explicitly requested)

- By default, CUDA will implicitly assume that consecutive operations that **could** have a dependency also **do** have a dependency, e.g.:
    - Kernel **A** followed by kernel **B** → **A** must finish before **B** starts
    - Copy memory to device before kernel → copy must finish before kernel starts
    - Copy results from device after kernel → Kernel must finish before copy starts

- But then why do we need a synchronizing command?

Some CUDA commands, like kernel calls, are asynchronous with respect to the host. However, by default, they are not asynchronous to each other. That means that, unless specified otherwise, CUDA will assume that any kernel calls or copy instructions are dependent on previous events, and order them accordingly. For instance, when two kernels are launched in succession, the second will wait for the first to end before running. On the other hand, the basic methods for memory copies will synchronize both the GPU and the CPU. Thus, a kernel, followed by a copy from device to host will ensure that the copy command can see and transfer the results that were written by the previously launched kernel back to the CPU. While it seems like synchronization is mostly implicit, functions for explicit synchronization are sometimes required, like in the previous example.

Synchronization (Host with Device)

- `cudaDeviceSynchronize()` to synchronize CPU and GPU

- `cudaEventSynchronize()` to synchronize up to certain event

- Overuse incurs performance penalty, rarely needed! Examples:
  - Wait for the implicit transfer of the `printf` buffer to CPU for displaying
  - Measuring GPU performance with CPU code (e.g., with `std::chrono`)
  - Synchronize access to `managed` memory on CPU and GPU
  - Debugging (`cudaDeviceSynchronize` returns previous launch errors)

Two commonly used synchronization functions for the host side are `cudaDeviceSynchronize` and `cudaEventSynchronize`. Both of them synchronize the GPU and GPU, with the difference that the former synchronizes the CPU will all previously submitted asynchronous commands, while the second takes an additional event parameter that marks a particular point in the GPU execution pipeline. While it may not break the program to overuse synchronization functions, it will be detrimental to performance. Hence, `cudaDeviceSynchronize` should be reserved for particular use cases and placed with care if performance is key. The use cases include, for instance debugging applications, the use of unified managed memory, which we will talk about in part 3, and in the particular case of our example, when printf is used, to make sure that the CPU will wait for the implicitly buffered console output to be transferred back to and processed on the CPU, without the use of an explicit copy instruction.

- __host__ qualifier for host functions, combines with __device__

```
__host__ __device__ float squareAnywhere(float x)
{
    return x * x;
}
```

- Architecture-agnostic code can significantly simplify your code base!

- Critical sections that require architecture-specific instructions can be implemented using the __CUDA_ARCH__ preprocessor macro

In addition to __global__ and __device__, CUDA defines an additional decoration for functions, named __host__. This is to signify functions that should be interpreted by the NVCC as functions that run on the CPU. If none of the available labels is used, NVCC will by default assume that a function is a host function. However, the addition of this label opens up a new possibility for increasing code reuse: functions that are decorated with both __host__ and __device__ labels will be compiled to run on both, the host and the device. If the code being used is generic enough to run on both, this means that developers can write architecture-agnostic code once that may be executed on both architectures. We will see that, with the introduction of recent features, the restrictions regarding what can and cannot be written in this portable manner are continuously dwindling.

- Execution occurs in a hierarchical model

- CUDA distinguishes four granularities:
  - Grid (launch configuration)
  - Block (cooperative threads)
  - Thread (isolated execution state)

- In-between: warps
  - Groups of 32 threads, enable SIMD execution
  - Implicitly defined as parts of a block

Grid
Block
Warp
Thread

multiple blocks

multiple warps

32 threads
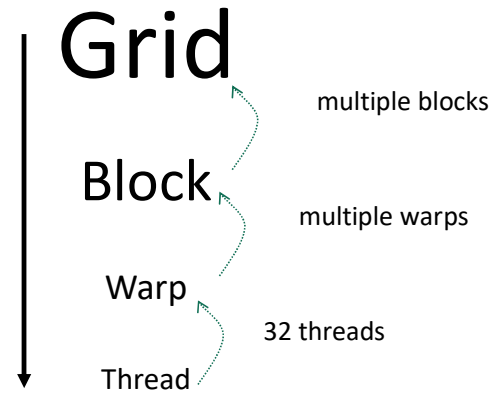
The execution hierarchy of code that is launched to run on the GPU provides several layers. For a CUDA kernel launch, a definition of a grid is required, which includes the number of cooperative thread blocks that should be started, as well as the size of each individual block. Below the threadblock granularity are individual threads, which can hold individual information and state during execution. An additional, hardware-governed layer lies between the two: the warp. Blocks will implicitly be split into warps, that is, groups of 32 threads, which may execute together on the SIMD units of the GPU.

Grids to Blocks to Threads

- Grid defines total number of launched threads
  - Indirectly, via the number of blocks
  - Complete grid defined by grid and block dimensions

- Threads within a block can synchronize

- Up to 32 threads (a warp) execute the same instruction on the same SIMD compute unit
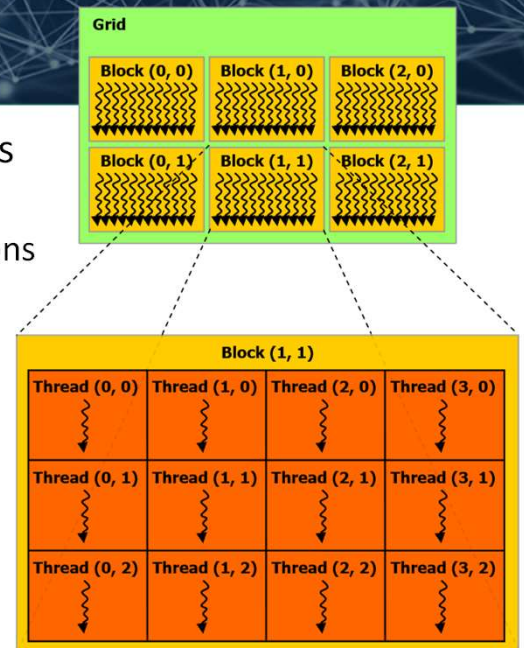
We can visualize this relationship more clearly. A grid may contain multiple blocks, each of which has a configurable size that dictates the number of threads in a block. The threads within a block have special opportunities to communicate, and may for instance synchronize at a certain point in the program. However, each thread in a block can have its own state and memory, and therefore represents its own entity. For the sake of exploiting SIMD hardware units, threads will always execute in groups of 32, regardless of the block size being used.

# Logical and Physical Hierarchy Aspects

- Logical kernel hierarchy levels: grid, block, thread
  - Allows developers to structure their solutions for particular problem sizes
  - Leaves threads with the independence to make individual decisions

- Physical kernel hierarchy level: the warp
  - Hardware SIMD-width: an arbitrary grouping that developers cannot avoid
  - Also implies that at least 32 threads will run (although some may be inactive)
  - Historically, CUDA has tried to make warps transparent to novice developers
  - **Before** Volta: not understanding warps may crash your application
  - **After** Volta: not caring about warps may make your application slower

CUDA Block Execution Model

- Grid size can be chosen, regardless of GPU model
  - Use grid configuration to complete a particular task
  - Abstracts away hardware scheduling details
  - Block queue provides processors with work
  - Adapting to hardware may raise performance

- Threads in a block can share, synchronize

- Warps of one block are assigned to single streaming multiprocessor (SM)
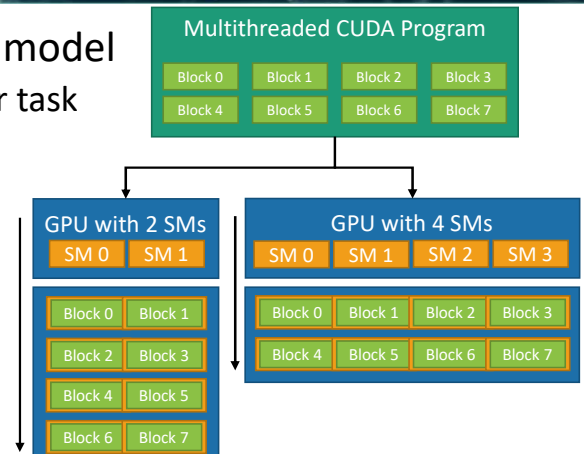
When running a kernel, the blocks that make up a grid are committed to the GPU in a block queue. The GPU will then proceed to process the blocks in parallel. The degree of parallelisms depends on the hardware being used but is transparent to the developer: only the problem size, that is, the grid configuration and how many threads should run, must be defined. The GPU will then process as many blocks as it can fit on its parallel compute units and keep fetching work from the block queue until all threads have completed execution. Each block (and the warps it is comprised of) is explicitly and fully assigned to one of several larger processing units of the GPU, the streaming multiprocessors.

Streaming Multiprocessors

- CUDA cores: basic integer/floating point arithmetic – high throughput, low latency

- Load/Store (LD/ST): issues memory accesses to appropriate controller – possibly high latency

- Special Function Unit (SFU): trigonometric math functions, etc – reduced throughput

- Since Turing and Volta, also include special **tensor cores** (not explicitly shown here)
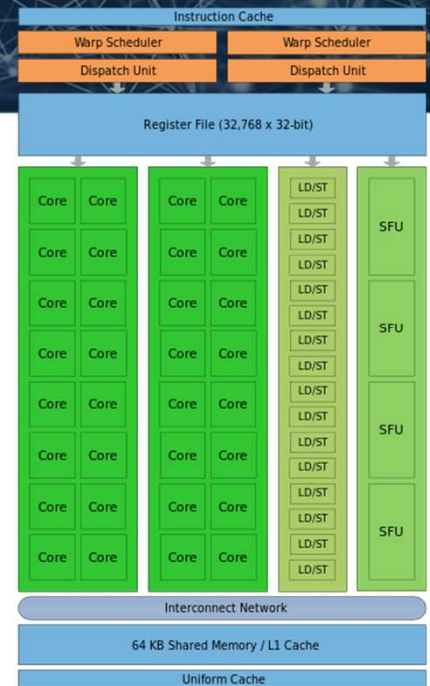
The streaming multiprocessor, or SM for short, is the powerhouse of the NVIDIA GPU. It contains the relevant, specialized units that threads can use to retrieve or compute results. We can distinguish so-called CUDA cores, which is usually a synonym for the units that perform integer or floating-point arithmetic, the load and store units, which take of communicating with different types of memory, special function units, which perform slower, more complex operations and, last but not least, the recently introduced tensor cores that have specialized matrix arithmetic capabilities.

CUDA Warp Execution Model

- When blocks are assigned to SMs, their warps are made "resident"

- In each cycle, SMs attempt to find warps to execute instruction

- If none of the resident warps are ready to run, the SM will idle

- Each warp scheduler may select a warp that is ready to proceed
  - All threads in executed warp run the same instruction → convergence
  - Different threads are at different points in the program → divergence

When we assign blocks to a particular SM, their warps are described as being resident on that SM. In each cycle, the SM will then try to schedule instructions for warps that were assigned to it. Naturally, an SM can only select warps that are ready to be executed. Hence, if a particular warp is depending on the result of a computation or a memory transfer, it may not be scheduled. This brings back the concept of oversubscription of the compute units of the GPU. The more warps an SM has to choose from, the higher the chances are that it can hide latency by switching to different warps.

Since warps execute as one, the threads in them can progress simultaneously. However, every thread is still its own entity, and may choose not to participate in a scheduled instruction. In this case, we refer to the warp as being diverged.

CUDA and Applications to Task-based Programming
25.04.2022
32
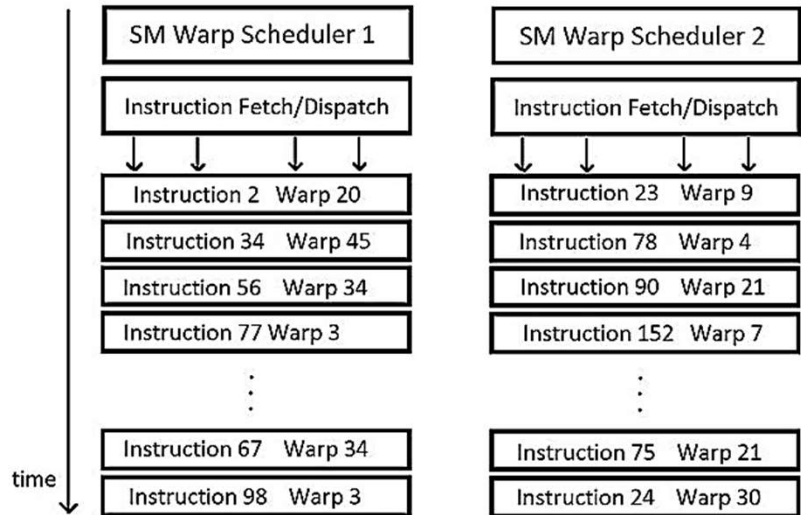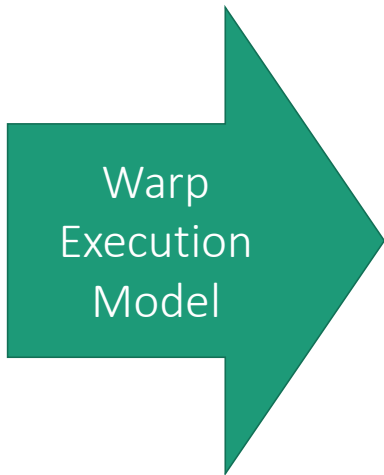
Here we can see a basic illustration of the execution model in an SM, with one potential progression over time. The SM warp schedulers will try to find ready warps, fetch instructions and dispatch them for execution. It is unlikely that a warp can immediately continue execution, hence the warp scheduler will try to find a different warp for the next cycle. As time progresses, warps eventually make progress until all warps in the block have completed their tasks.

CUDA Threads and SIMT

- Each thread may follow a different path, setting it apart from SIMD
  - Threads maintain active/inactive state information during program
  - Selectively executing instructions when **active** leads to diverging behavior

- CUDA code can be agnostic of the size and SIMD nature of warps

- New naming convention: single instruction, multiple threads (SIMT)

- Thread behavior usually governed by unique global or local launch ID

As stated, each thread in a warp has its own set of individually computed values, as well as an active flag that indicates whether or not a thread will participate in the computation within its warp. This active flag is all that is required to elicit individual behavior for threads, even when they progress as warps. By selectively enabling and disabling this flag, every thread in a warp can theoretically explore a different flow in the running program and arrive at a unique state. This is however a design choice in hardware, and transparent to the programmer. Developers can, for the most part, write CUDA code as if every individual thread was executed individually, with some exceptions. This architecture design, which enables threads to behave like individual entities, while still enabling the exploitation of efficient SIMD operations when threads are not diverged is described by the term "same-instruction-multiple-threads", or SIMT for short.

Distinguishing Threads and Blocks

- Program flow can vary depending on `threadIdx` and `blockIdx`, `gridDim` and `blockDim`

```cpp
__global__ void PrintIDs()
{
    auto tID = threadIdx;
    auto bID = blockIdx;
    printf("Thread Id: %d,%d\n", tID.x, tID.y);
    printf("Block Id: %d,%d\n", bID.x, bID.y);
}

int main()
{
    …
    dim3 gridSize = { gridX, gridY, gridZ };
    dim3 blockSize = { blockX, blockY, blockZ };
    PrintIDs<<<gridSize, blockSize>>>();
    cudaDeviceSynchronize();
    …
```

Each thread can, for instance, adapt its behavior depending on its launch IDs. CUDA provides several built-in variables that threads can access in order to retrieve their ID in the grid or inside a block, which they can use to identify their target or source position in a given problem domain. Consider for instance an image, where each thread should be assigned to a particular 2D portion to perform, e.g., a filtering operation. In this case, the grid may be configured in a variety of ways. Grids can have up to 3 dimensions, x, y and z, and we can use 3-dimensional structs as parameters for the kernel launch. In the case of a 2D image, it makes sense to utilize 2D block and grid dimensions, for instance. After launching a particular kernel, each thread can retrieve the coordinates of the block in the grid, as well as the coordinates of the threads inside each block. The image on the right illustrates this for a simple case, where 2D block and thread IDs are illustrated for a simple block layout that uses 8 threads on its x-axis and 1 on its y-axis. The numbers that they are labeled with correspond to the output that each thread would create when running the code on the left, respectively.

In combination with another built-in variable, blockDim, threads may also easily find their unique global ID in the full grid, such as the exact pixel that they should compute in an output image.

- In-order program execution (but compiler may reorder instructions)

- Volta and later architectures support two thread execution modes
  - Legacy Thread Scheduling
  - Independent Thread Scheduling (ITS)

- On current GPUs with ITS, can select either model with compiler flag

- Can significantly change performance and correctness (!) of code

Whenever threads run on the GPU, they will follow the compiled instruction in order. As of now, there is no significant layer for ILP, however, the compiler may of course decide to reorder the coded operations to boost performance at runtime. Modern NVIDIA GPUs support two separate execution modes: one is legacy scheduling, which was the only available option until the Volta architecture arrived, and independent thread scheduling, which was introduced with Volta. Which execution mode should be used can be selected with a compiler flag. However, it is important to understand the fundamental implications of choosing either mode, since using one over the other can decide whether or not a particular code sample elicits undefined behavior or causes crashes.

Legacy Thread Scheduling

- Only one program counter per warp, i.e., entire warp can only store a single position for all threads in the executed program

- All threads that are inactive will not execute current instruction

- Threads may only progress to the next instruction in lockstep

- When branches occur, warp must execute first one, then the other

Legacy thread scheduling follows the conventional "lockstep" principle. This mode implies that there is only a single program counter per warp. That is, all threads in a warp may only ever be at the same instruction in the program. If program flow diverges, the SM must execute first one branch to completion and then the other, before the warp can proceed.
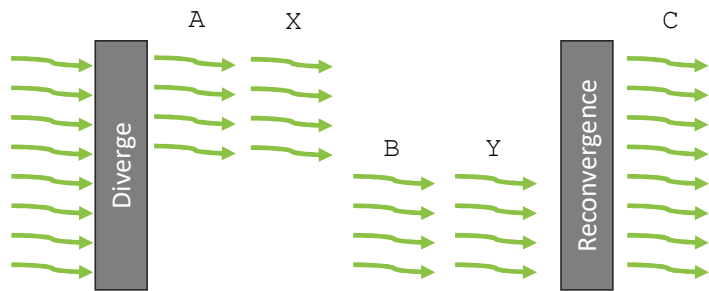
Legacy Thread Scheduling

- Diverged threads will try to reach convergence point before switching

- Cannot get past convergence point until all involved threads arrive

```
if(threadIdx.x & 0x4)
{
    A();
    X();
}
else
{
    B();
    Y();
}
C();
```

This behavior is illustrated here. Consider for instance the branch given based on the thread ID. The lower four threads will enter one branch, the remaining threads will enter the other. However, once a branch has been chosen, it must be completed before the other branch can begin because the warp only maintains a single program counter for all threads. It can, for instance, not switch to execute B directly after A, because that would imply that half of the threads are at one point in the program, while the others are at another instruction, hence both branches would need to maintain separate program counters.

Legacy Thread Scheduling

- Scheduling dictates what algorithms are and aren't possible

- Actually, quite easy to get a deadlock between threads within a warp

```
if(threadIdx.x & 0x4)
{
    A();
    waitOnB();
}
else
{
    B();
    waitOnA();
}
C();
```


A   waitOnB…

Diverge
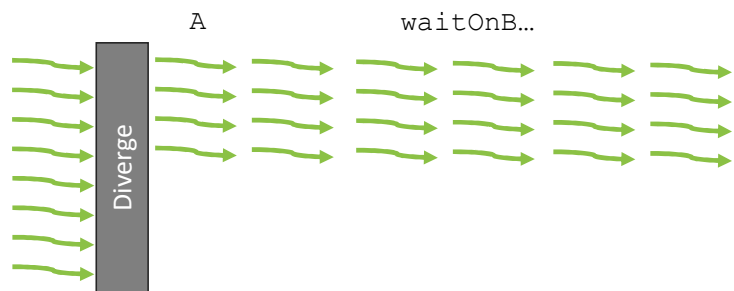
This has several implications that programmers must respect when they program for individual threads. For instance, consider the case where half of the threads in a warp are waiting on the other half. This is illustrated in this code sample. Because with the legacy thread scheduling model, threads cannot execute a different branch until the first chosen branch is complete, this program will hang since either A or B will never be executed, but each branch is waiting on an event that occurs in the other.

Independent Thread Scheduling (ITS)

- Two registers reserved, each thread gets its own program counter

- Individual threads can now be at different points in the program

- Warp scheduler can (and does) advance warps on all possible fronts
  - Guaranteed progress for all resident threads
  - Enables thread-safe implementation of spinlocks, starvation-free algorithms

- Threads in a warp still can only do one instruction at a time

With independent thread scheduling, situations like this are no longer an issue. Each thread is given its own, individual program counter, meaning that theoretically, each thread can store its own unique instruction that it wants to perform next. The execution of threads still happens in warps, this has not changed. It is not possible for threads in a warp to perform different instructions in the same cycle. However, a warp may now be scheduled to progress at any of the different program counters that the threads within it are currently holding. Furthermore, ITS provides a "progress guarantee": eventually, over a number of cycles, all individual program counters that the threads in a warp maintain will be visited. This means that if, for instance, the execution has diverged and two branches, both are guaranteed to be executed sooner or later.
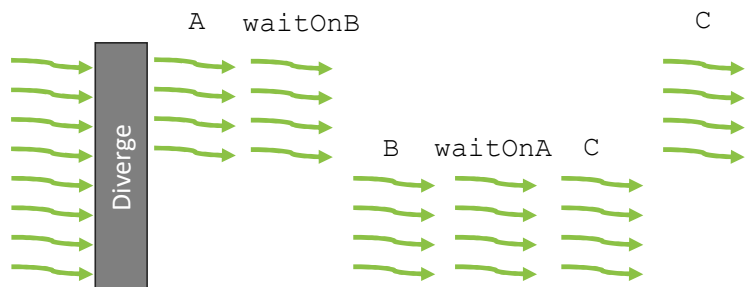
Independent Thread Scheduling (ITS)

- Guaranteed progress, one branch can wait on another branch

- Diverged threads may not reconverge, should be explicitly requested!

```
if(threadIdx.x & 0x4)
{
    A();
    waitOnB();
}
else
{
    B();
    waitOnA();
}
C();
```

With ITS enabled, the previous code sample no longer poses a problem. A branch may be chosen as before start waiting on the other branch. Due to the progress guarantee, sooner or later, the other branch will be scheduled and its threads will proceed, which is possible because every thread has a program counter to maintain its own unique position in the program code. A side effect of the new design, however, is that program code can no longer make any assumptions about threads moving in lockstep since they are free to stay diverged until the program finishes. The GPU will try to make threads reconverge at opportune times, but if it is desired that threads are guaranteed to perform given instructions in groups of 32, e.g., to exploit SIMD behavior, this must now be explicitly requested with a synchronization command.

- `__syncwarp()` synchronizes active threads in a warp
  - Volta and later architectures only, before that no threads with different PCs
  - Additional `mask` parameter enables synchronizing a subset only
  - May be called from different points in the program, as long as masks match

- `__syncthreads()` synchronizes active threads in block at a point
  - All active threads must reach the same instruction in the program
  - Undefined behavior if some threads in block do not reach it (likely hang!)

- `this_grid().sync()` can busy-wait to synchronize entire kernel

In addition to the host-side functions that synchronize between CPU and GPU, which we saw before, synchronization may of course also be performed between the threads running on the device itself. The primitive to use to force a warp or parts of a warp to reconverge is the `__syncwarp` function. `__syncwarp` only really makes sense on systems that support ITS, because earlier models would have warps advance in lockstep. `__syncwarp` takes an additional mask parameter, which can be used to define only a subset of the threads in a warp that should synchronize. This is conveniently done via a 32bit integer, where each bit indicates whether or not a thread with the corresponding ID should participate in the synchronization. Interestingly, `__syncwarp` may be called from different points in a program, e.g., it is possible for threads in a warp to synchronize while they are executing different branches. However, according to documentation, it is an error to have threads reach a `__syncwarp` they don't participate in. One level above is the `__syncthreads`, which is not so forgiving and applies to all threads in a block.

A `__syncthreads` will make sure that all active threads in a block arrive at the same point in the program where the synchronization happens. In contrast to `__syncwarp`, it may NOT be called from different branches in the same block, since this may cause the program to hang. Lastly, it is also possible to synchronize the entire kernel launch grid, that is, wait for all threads to arrive at a certain point in the program, however, this method has several restrictions and requires a special setup, as well as the cooperative groups programming model, which we will see only in part 3.

Warp-Level Primitives

- Initially, CUDA programming paradigm stopped at block level
  - Developers were not meant to assume specific properties about warps
    - But performance benefits were too great, so they did anyway (e.g., warp voting)

- **Warp-level primitives** are instructions where threads in one warp exploit the fact that they run together to quickly share information

- Most instructions available since compute capability 3.0 (Kepler)
  - Since CUDA Toolkit 9.0, must include synchronization to comply with ITS

Now that we have a basic understanding of what grids, blocks and threads are, we should point out the special role of warps. The fact that threads are scheduled in warps is independent of the grid-block-thread design. Initially, developers were not meant to assume particular behavioral properties of warps and the official programming paradigms would not include them. However, as it turns out, the benefits of exploiting the knowledge of which threads are scheduled together for an instruction is much too important for performance to be ignored. The CUDA programming model has since committed itself to expose and encourage the use of knowledge about warps during execution. In particular, NVIDIA hast started to introduce so-called warp level primitives. These include special instructions that provide a fast lane for threads that are scheduled together for execution to exchange information with a single, fast instruction.

These warp-level primitives have been enabled starting with architectures that have compute capability of 3.0 or higher. In order to comply with the CUDA standard in the toolkit 9.0 or newer, they have been updated to enforce synchronization on devices with ITS. If you are not familiar with these terms, however, you may be wondering what exactly a compute capability is, how it associates with the CUDA version, and why those numbers are at times so dissimilar?

Compute Capability ≠ Toolkit Version

- One ensures availability of explicit hardware capabilities, the other the toolkit's support for building applications that can exploit them

- Although not directly associated, restrictions do apply
  - E.g., cannot use tensor core instructions on Turing card if toolkit is outdated

- Highest compute capability currently at 8.6

- Latest CUDA Toolkit currently at version 11.6

It is important to note that those two signify very different things, although they are related. The compute capability of a given GPU ensures its ability to perform certain operations, expose features or adhere to particular hardware specifications, such as the number of available CUDA cores or tensor cores per SM. On the other hand, the CUDA toolkit version will govern whether your development environment is capable of translating code that makes use of new hardware-accelerated instructions and features. For instance, you cannot use an outdated CUDA toolkit to compile code that makes use of tensor cores, even if you are running the compiled code on a Turing card.

Architectures and Compute Capabilities

Architecture	Exemplary GPU Model	Compute Capability	Important Features
Tesla	GeForce 8800 GTX	1.0 – 1.3	Basic
Fermi	GeForce GTX 480	2.0 – 2.1	Ballots, 32-bit floating point atomics, 3D grids
Kepler	GeForce GTX 780	3.0 – 3.7	Shuffle, unified memory, dynamic parallelism
Maxwell	GeForce GTX 980	5.0 – 5.3	Half-precision floating point operations
Pascal	GeForce GTX 1080	6.0 – 6.2	64-bit floating point atomics
Volta	TITAN V	7.0 – 7.2	Tensor cores
Turing	GeForce RTX 2080	7.5	More concurrency, RTX cores (not compute)
Ampere	GeForce RTX 3090	8.0 – 8.6	L2 Cache Residency Management
Lovelace	?	9.0 – ?	?

Here, we provide a rough summary for orientation of how compute capabilities map to different architecture generations and some of the most important features that they introduced to GPU models of that era.

Example: Parallel Reduction

Reims 2022

Let us now consider a concrete example where we exploit the parallel processing power of the GPU with CUDA to accelerate a very common operation: data reduction.
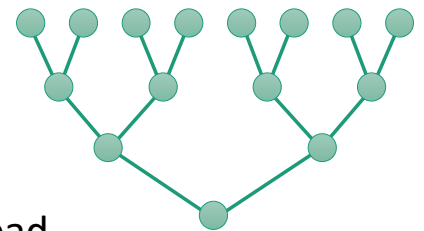
Parallel Reduction

- Common and important data-parallel primitive
  - Many data elements → single output (e.g., sum)
  - Easy to implement in CUDA, tree-based approach

$$\sum \quad \boxed{12} \; \boxed{42} \; \boxed{99} \; \boxed{99} \; \ldots \; \boxed{5}$$
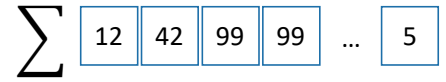
- To beat CPU, need to use multiple thread blocks
  - A large grid to process large arrays
  - More parallelism can better utilize the GPU

- Partition the array, map each entry to a single thread
 - Where and how do we combine them to calculate the result?

For parallel reduction, our aim is to exploit the parallel nature of the GPU in order to compute some sort of reduced result from a large number of original inputs, such as their sum, the minimum value, or their average. Reduction is a general and useful operation and is also rather effective to compute on the CPU since it can usually be performed with a single pass over the full input length. However, we will try to show how parallel computing on the GPU can exploit the knowledge about the different levels of the execution hierarchy and collaborate across them can yield significantly improved performance for this type of operation. However, before we do so, we must first find a way to receive inputs and store the final result. Hence, let's have a quick preview of the different memory types we have at our disposal.

Types of Memory

- Registers
  - Per-thread, fast, automatically allocated for variables
- Local Memory
  - Per-thread, slow, used when registers are unavailable
- Shared Memory
  - Per-block, fast, allocated by host or `__shared__`
- Global Memory
  - Per-device, slow, allocated by host or `__device__`
- Constant Memory
  - Per-device, fast uniform access, via `__constant__`
- Texture Memory
  - Per-device, slow, with texture reading functionality
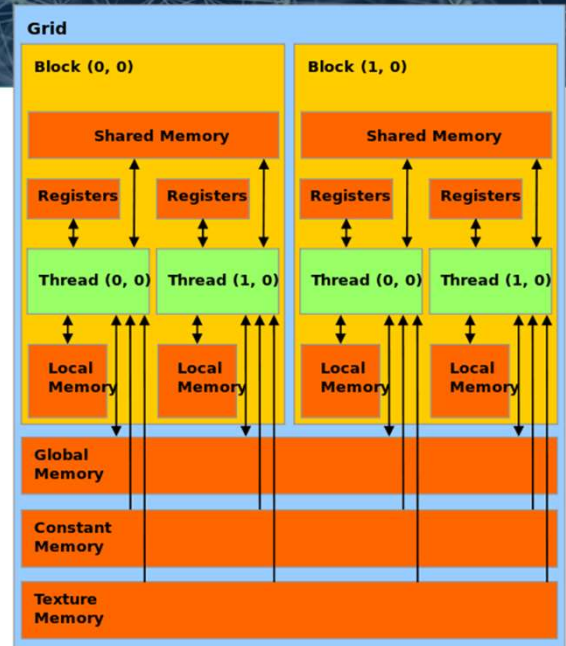
Registers are the fastest type of memory. Similar to registers on the CPU, they are allocated automatically for basic variables in computations. However, they are only visible per-thread, hence they are not suited for device-wide communication. Local memory, too, is memory that is only visible per-thread, and is used when it is not possible to use the faster registers. Shared memory is somewhat slower than registers and visible to all threads within a block. However, this is not sufficient, since we are considering a potentially vast number of inputs, which may be much more than the maximum size of a block, that is, 1024 threads. Global memory, on the other hand, is visible to every thread in the device, but also significantly slower, since it is not directly located on the SM. It can also be allocated and written to by the host. Constant memory describes a limited amount of read-only global memory with a particularly fast cache for uniform reads, and texture memory has additional capabilities that mirror those of texture and image variables in common graphics APIs.

Since we want to read potentially large input arrays to reduce and write the result where we can later retrieve, we will therefore choose to place both of them in global memory.

- Result must be updatable from every thread → use global memory
  - When thousands of threads simply write to memory, results are lost
  - First solution: use **atomic operations** to update single global variable

```
__device__ float result = 0;

__global__ void reduceAtomicGlobal(const float* input, int N)
{
    int id = threadIdx.x + blockIdx.x*blockDim.x;
    if (id < N)
        atomicAdd(&result, input[id]);
}
```

For the variable in which we store the result of our reduction, we can do this by defining a __device__ variable in the CUDA source file directly. Our first attempt at a reduction kernel can then add its entry to the result variable. In this case, we are performing a reduction with addition to compute the sum of all entries in the input array. We first identify each thread's unique ID, using the built-in threadIdx, blockIdx, and blockDim variables and assuming that all of them are specified with a single dimension on the x-axis. This is reasonable since the input is a 1D array and there is no added benefit from using more dimensions in the grid configuration. Note however that if we were to launch our kernel with a 2D grid instead, we would have to consider the .y coordinates in the computation as well. Each thread first checks whether its ID is lower than the number of entries to sum up. This is because thread blocks have a fixed size, hence, when we launch this kernel, in order to sum up all results, we need to make sure that we launch enough blocks.

But since the number of entries in the array N may not be a multiple of the block size, some of the threads in the last block may not want to participate in the reduction to avoid access violations. Next, we retrieve the corresponding entry from the input array and add it to the result variable. However, we are using a new function, atomicAdd, to access the `__device__` variable instead of updating it directly. Why?

Atomic Operations

- Updates to the same memory problematic with many threads
  - Read/write may occur in arbitrary order, simultaneously, overlap, be stale?
  - Atomic operations are indivisible, visible and occur in some sequential order
  - Atomic operations where return value is not used are termed *reductions*

```
foo += bar;
```

?

foo

```
atomicAdd(&foo, bar);
```

Ordered | Updates

foo
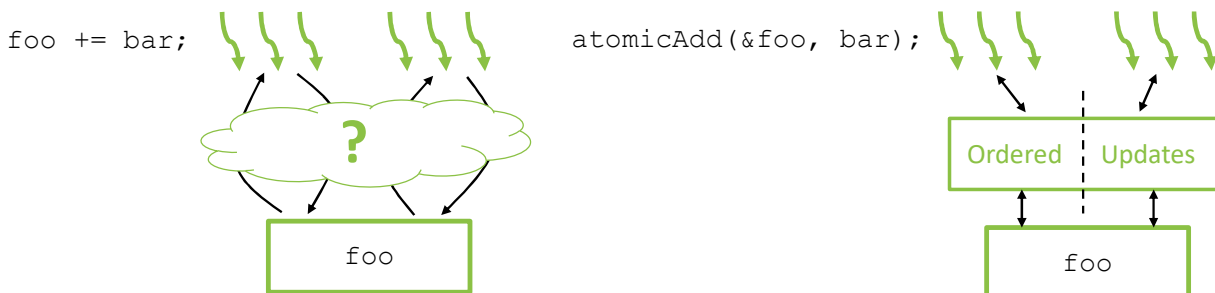
As in all multi-threaded applications, it is necessary to protect against data races to obtain coherent results. If we were to simply add values to a variable, there is no guarantee that the updates will produce the correct final result. First, each addition can be broken into two memory operations for every thread: fetching the current value and writing the new one. Fetching and writing by threads of the global variable may occur in any order, hence the result of performing these operations simultaneously with thousands of threads is undefined. Atomic operations in CUDA, as most other architectures, provide us with means to perform updates atomically, i.e., they cannot be interrupted since they are indivisible. Furthermore, atomic operations are guaranteed to produce the same effect as if all accesses to the variable had occurred in some strictly sequential order. Hence, with atomic operations, thousands of threads can add entries to the same global variable and obtain the correct result.

Performance 268M Float Reduction

- Initial version is slower than CPU implementation, which is linear

- GPU version has maximal contention on slow, global memory

	Time (TITAN V)	Bandwidth	Step speedup	Cumulative Speedup	Speedup to CPU
AtomicGlobal	635.355ms	1.520GB/s	1.000	1.000	0.3
	CPU Baseline: 283.989 ms				
	CPU Parallel Baseline: 85.393 ms				

Unfortunately, this guarantee comes at a price. Our initial implementation performs significantly poorer on a Titan V than even a single-threaded CPU implementation, let alone a multi-threaded CPU implementation. However, this is only where we begin to apply our knowledge of the GPU architecture.

- Compute block results in fast, shared memory, update global at end

```cpp
__global__ void reduceAtomicShared(const float* input, int N)
{
    int id = threadIdx.x + blockIdx.x*blockDim.x;

    __shared__ float x;
    if (threadIdx.x == 0) x = 0.0f;
    __syncthreads();

    if (id < N) atomicAdd(&x, input[id]);
    __syncthreads();

    if (threadIdx.x == 0) atomicAdd(&result, x);
}
```

As we just mentioned, there is another type of memory, which is found directly on the SM that a block runs on, and which is supposedly much faster than global memory. Hence, we can split our reduction into two stages: first, we perform updates atomically in faster, shared memory, and then only write the partial results out to global memory. Consider, for instance, a setup where each block has 256 threads. In this case, we just reduced the number of atomic updates to slow global memory by a factor of 256. The main contention was moved from a single, global variable to multiple variables, one per block, that is held in shared memory on each SM. Observe that both the initialization and the final addition of the shared variable are performed only by the first thread in the block. Before and after the accumulation in shared memory, the entire block synchronizes. This is to ensure, for one, that the first thread correctly initializes the shared variable to zero before threads start to accumulate on it.

The second __syncthreads is to ensure that all threads in the block have finished with their accumulation before the first thread in the block performs a single update to global memory, otherwise, it could update it with an incomplete result.

Performance 268M Float Reduction

- Switching from global to shared for most atomics outperforms CPU

- Contention is still high, but must only serialize on fast shared memory
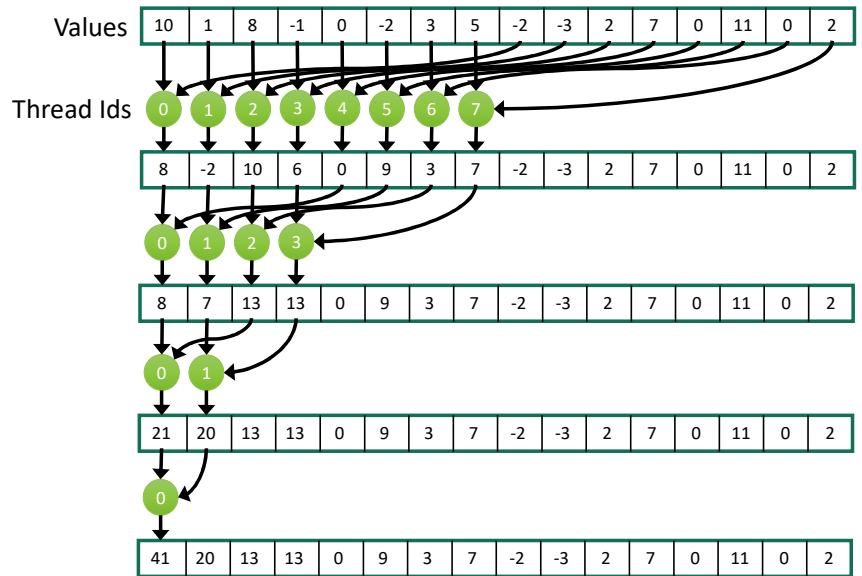
	Time (TITAN V)	Bandwidth	Step speedup	Cumulative Speedup	Speedup to CPU
AtomicGlobal	635.355ms	1.520GB/s	1.000	1.000	0.3x
AtomicShared	26.911ms	35.674GB/s	23.61	23.61	3.7x

This second version already performs significantly better than our first attempt. Furthermore, it puts us over the bar for improvement over the parallel CPU method and is now almost 4 times faster. However, there is still room for improvement. So far, we simply ported an approach that would work well on the CPU and reduced the amount of memory contention it causes. Let us return to the drawing board and consider if perhaps a different, inherently parallel algorithm can give us better results.

- Multiple iterations to reduce full input data

- In each iteration, add two values per thread

- Exclusive access, just $\log_2 N$ serialized steps

Values

| 10 | 1 | 8 | -1 | 0 | -2 | 3 | 5 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |

Thread Ids 0 1 2 3 4 5 6 7

| 8 | -2 | 10 | 6 | 0 | 9 | 3 | 7 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |

0 1 2 3

| 8 | 7 | 13 | 13 | 0 | 9 | 3 | 7 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |

0 1

| 21 | 20 | 13 | 13 | 0 | 9 | 3 | 7 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |

0

| 41 | 20 | 13 | 13 | 0 | 9 | 3 | 7 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |

Our best solution so far must still enforce sequential updates of a common variable, even though it occurs in faster shared memory. However, if we are aware of the existence of shared memory, we can come up with an elegant solution that can achieve the same result with a sublinear runtime. Consider the illustration given above. Starting with the original input, we can run multiple iterations in which each thread combines its current value with that of another thread, yielding a partially reduced intermediate result. By continuously summing up these partially reduced results, due to the transitive nature of the operation, we can eventually obtain the result of the full reduction over the inputs for all threads in the block in log2(N) iterations.

53

```
__global__ void reduceShared(const float* input, int N)
{
    __shared__ float data[BLOCKSIZE];
    int id = threadIdx.x + blockIdx.x*blockDim.x;
    data[threadIdx.x] = (id < N ? input[id] : 0);

    for (int s = blockDim.x/2; s > 0; s/=2)
    {
        __syncthreads();
        if (threadIdx.x < s)
            data[threadIdx.x] += data[threadIdx.x + s];
    }
    if (threadIdx.x == 0)
        atomicAdd(&result, data[0]);
}
```

Changing our previous implementation to this new algorithm is not too difficult, since the majority of it can be implemented with standard C++ instructions. Note that in this case, we have found a new way to deal with the problem of potentially redundant threads in the last block that is started. In order to keep our implementation simple, we implicitly pad the read data to a multiple of the block size by having threads with an ID beyond N act as if they read a zero value. This way, they can safely participate in the reduction without changing the final result and altering our code to handle this special case. Next, we implement the previously described algorithm with a simple loop structure. However, we have to make sure that each iteration is secured by a call to __syncthreads to make sure that all threads have finished their updates before we continue with the next iteration. This is because, in each iteration, some threads are dependent on the results that other threads produced in the last iteration.

Note that there is no __syncthreads before the update to global memory is made, due to the fact that in the very last iteration, only thread 0 participates in the loop, and it may immediately use the result that it computed itself without synchronizing.

54

- The improved algorithm has a significant impact on performance

- Now even significantly reduced contention on shared memory

	Time (TITAN V)	Bandwidth	Step Speedup	Cumulative Speedup	Speedup to CPU
AtomicGlobal	635.355ms	1.520GB/s	1.0	1.00	0.3x
AtomicShared	26.911ms	35.674GB/s	23.6	23.61	3.7x
ReduceShared	2.903ms	333.536GB/s	9.2	218.86	29.3x

We can easily see that choosing a more suitable algorithm has had the biggest impact on performance so far. Exploiting both the best available memory types and inherently parallel algorithms are fundamental principles for obtaining optimal GPU performance. But we can still go a little farther.

Exchanging Registers

- Registers are by far the fastest type of memory to access

- Threads that run together as a warp can exploit warp-level primitives

- Exchange register data with another thread in warp: `__shfl_sync`
  - Returns the value that another thread has in a particular register
  - Must include synchronization, because threads may have diverged due to ITS
  - Like `__syncwarp`, threads may shuffle at different points in code on Volta+
    - Works like a messaging system – threads can put different registers on the line

Before, we mentioned that registers are the fastest type of memory available. We also mentioned that ever since compute capability 3.0, it is advised and encouraged to exploit knowledge about warps executing simultaneously with warp-level primitives. The shuffle instruction gives threads in a warp a fast lane to exchange information in registers, without having to write them out to shared or global memory. This operation which, if ITS is enabled, must of course synchronize that the desired threads are converged before it exchanges values will be exploited by us for the final stage of the reduction.

```
__global__ void reduceSharedShuffle(const float* input, int N)
{
        …
        for (int s = blockDim.x/2; s > 16; s/=2)
        …
        float x = data[threadIdx.x];
        if (threadIdx.x < 32)
        {
                x += __shfl_sync(0xFFFFFFFF, x, threadIdx.x + 16);
                x += __shfl_sync(0xFFFFFFFF, x, threadIdx.x + 8);
                x += __shfl_sync(0xFFFFFFFF, x, threadIdx.x + 4);
                x += __shfl_sync(0xFFFFFFFF, x, threadIdx.x + 2);
                x += __shfl_sync(0xFFFFFFFF, x, 1);
        }
        if (threadIdx.x == 0)
                atomicAdd(&result, x);
}
```

Here we see how this can be applied to optimize our current implementation for parallel reduction. The reduction in shared memory stops at 32 partial results. Afterward, we only let the threads in a warp exchange their accumulated results with each other. In the first iteration, each thread in the warp will try to read the value of the thread with an ID that is 16 higher than its own. Note that the 16 higher threads will not obtain meaningful results from this operation, nor do we need them to. However, they are still participating in the shuffle, because the lower 16 need to access their registers. In the following iterations, this procedure is repeated until finally thread 0 receives the accumulated register of thread 1. Having obtained a completely reduced sum, it then performs the sole update per block to global memory, as before.

57

Performance 268M Float Reduction

- We will stop at this point, but this could still be taken further

- Note: results can (and do) vary significantly between GPU models

	Time (TITAN V)	Bandwidth	Step Speedup	Cumulative Speedup	Speedup to CPU
AtomicGlobal	635.355ms	1.520GB/s	1.0	1.00	0.3x
AtomicShared	26.911ms	35.674GB/s	23.6	23.61	3.7x
ReduceShared	2.903ms	333.536GB/s	9.2	218.86	29.3x
SharedShuffle	2.101ms	460.501GB/s	1.3	302.38	40.4x

Depending on the architecture you are using, the additional use of warp-level primitives can make a significant difference, although in this case, it is comparably minor. However, the final achieved speedup relative to our initial version of a factor larger than 300 shows how important it is to know how collaborative processing can affect performance on the GPU.

Parallel Programming for the GPU

- Many algorithms are embarrassingly parallel (e.g., ray tracing)
  - Each thread can work completely independently, no communication
  - Even a direct port to the GPU may accelerate processing out of the box

- If developers know how threads collaborate, more opportunities
  - The GPU is at its most powerful when it can reuse partial results
  - Cache utilization, shared memory and warp primitives play important role
  - Competitive algorithms to reorder, reduce, analyze or filter large data sets
    - Sorting and scanning
    - Building and traversing hierarchical data structures
    - Even prioritized task scheduling

There is a large range of algorithms that can benefit directly from being ported to a parallel processor. These algorithms, which are usually classified as embarrassingly parallel, usually have no interdependencies and their efficiency rises with the number of simultaneously executing threads. However, if developers are aware of the opportunities to exploit collaboration by threads at different levels of the execution hierarchy, it significantly increases the range of algorithms that can be run on parallel architectures with significant performance gain compared to the CPU. As we have seen, even a comparably well-suited algorithm with linear runtime can be executed significantly faster on the GPU if these concepts are applied.

Profiling and Debugging

Reims 2022

But before you dive into the porting of highly complex algorithms and ambitious projects to see how much faster they can run on a GPU, we would like to give an overview of the tools that will allow you to evaluate and quantify your performance gains in a reproducible manner. It is also advisable to become familiar with the available tools and methods for detecting and fixing errors, in short, debugging parallel programs on the GPU.

- Possible solution: synchronize CPU and GPU and use `std::chrono`

- Better: use `cudaEvent_t` to mark measuring points in execution
  - Create events with `cudaEventCreate`
  - Start recording events with `cudaEventRecord`
  - Synchronize only to latest event with `cudaEventSynchronize`
  - Find the duration between two events with `cudaEventElapsedTime`
  - Eventually, clean up with `cudaEventDestroy`

- Events yield elapsed time in milliseconds, as measured by GPU clock

Initially, you may try to measure time the way it is commonly done, by using libraries like std::chrono that access the system clock. However, a cleaner method is provided by the CUDA toolkit, which can measure the GPU clock time elapsed between two events that are submitted to the stream of CUDA commands. Events can be created and recorded at arbitrary points during your program. For instance, to measure the runtime of a kernel with events, you can create two events and record the first just before and the other just after the kernel launch. You may synchronize on the second event to make sure that it has passed. After synchronization, the elapsed time between the two can be computed via `cudaEventElapsedTime`, which gives the elapsed time in milliseconds as measured by the GPU with microsecond resolution.

Reims 2022

- Timeline breakdown of application, identifies provoking architectures

- Allows to analyze the application's performance bottleneck
  - Identify specific routines, kernels or memory transfer that cause latency
  - Detect if execution is GPU bound, find opportunities for improvement

- Example: executing an expensive kernel 5 times in a row
  - Capture timeline with Nsight Systems and focus on GPU activity

Beyond simple timing measurements, a complete suite for profiling CUDA applications is given by the various tools in the Nsight family. With Nsight Systems, you can get a high-level overview of the events that occur in your application to identify, for instance, whether your application is CPU or GPU bound and which kernels are taking a particularly long time during execution. In the following, we will look at a short example that launches 5 consecutive, particularly slow kernels.
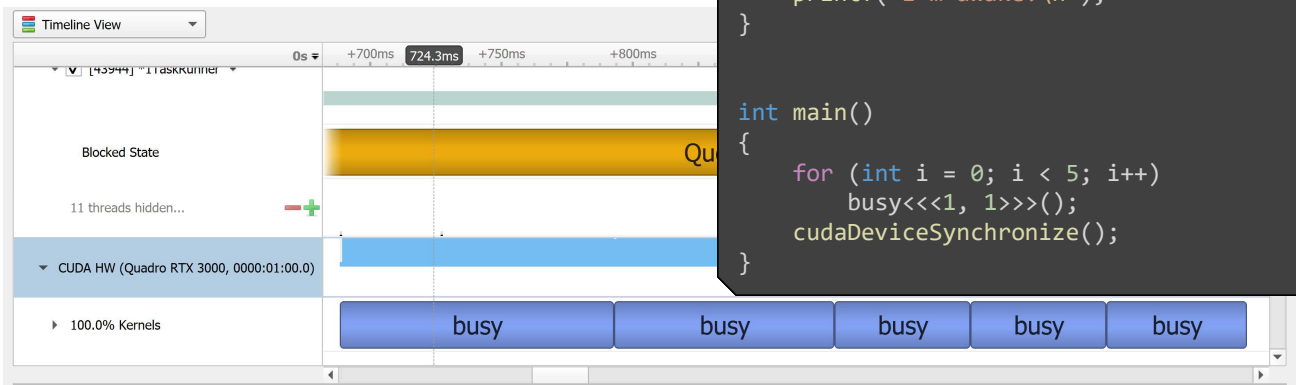
The complete code for this setup is provided on the right-hand side. The kernel will simply sleep for a given number of cycles before printing a single message. After sampling the application execution with Nsight Systems, we can use it to analyze the timeline for the program execution. Clearly, we can see that the five kernels that were launched in a loop execute one after another. We know that this is the case by default since CUDA will assume that kernels depend on each other unless indicated otherwise. However, in this example, it is evident there is no implicit dependency between kernels and they may just as well execute simultaneously. We can demonstrate how this can be achieved and confirm the change in the application timeline by introducing the concept of streams.

CUDA Streams

- A single, small kernel may not be enough to occupy the entire GPU

- CUDA is capable of running multiple jobs simultaneously

- Implicit dependencies apply, but we can separate them into streams
  - Streams are created at runtime and operations are associated with them
  - Developer uses streams to separate operations that have no dependency
  - Stream that a kernel should be launched in is 4th parameter (we skipped 3rd)
  - If no stream specified, default "Null" stream is used

CUDA enables developers to define independent streams of commands, where it is assumed that commands in different streams do not depend on each other. This becomes relevant in cases where, for instance, multiple smaller kernels should be launched to properly occupy the available processing units of the GPU, which may not be achieved by a single simple kernel. The stream can be passed to corresponding CUDA runtime API calls, such as `cudaMemcpyAsync`, or can be defined for kernel launches as the fourth parameter in the <<<>>> syntax.

- No dependencies assumed, parallel
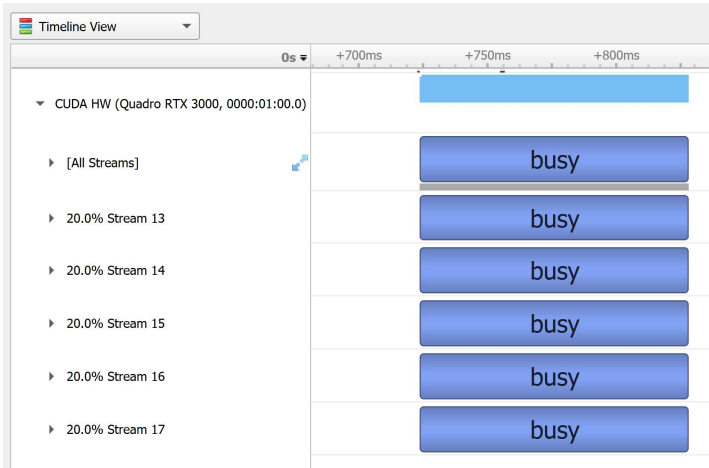

```cpp
int main()
{
    cudaStream_t streams[5];
    for (int i = 0; i < 5; i++)
    {
        cudaStreamCreate(&streams[i]);
        busy<<<1, 1, 0, streams[i]>>>();
    }
    cudaDeviceSynchronize();
    for (int i = 0; i < 5; i++)
        cudaStreamDestroy(streams[i]);
}
```

In our example, our five waiting kernels are small and simple enough to run simultaneously. The example on the right shows how this can be realized with streams. First, we create a stream for each kernel and then submit it to the corresponding stream. After the GPU has finished execution, we eventually take care of destroying the created streams. The analysis by Nsight Systems proves to us that, in fact, the execution flow of the program has changed: the five kernels no longer execute one after another, but instead, run concurrently on the same GPU.

Profiling Kernels with Nsight Compute

- Nsight launches target application, injects itself in API calls

- Will automatically replay kernels and applications to collect results

- Complete performance report including suggestions for improvement

- Lets users inspect and compare extensive set of performance metrics
  - Provides readouts and stats from hardware counters
  - Periodically samples and keeps track of code lines that cause stalls

Beyond the high-level overview and timeline for an application, we can also obtain a much more detailed performance report for individual kernels that we launch with Nsight Compute. Nsight Compute will produce reliable measurements by injecting itself into the program during its launch and replaying kernel calls multiple times to obtain readouts for different performance metrics. The result of this analysis can be a complete report, including suggestions for performance optimization by avoiding common issues and bottlenecks. By collecting samples during kernels of the program state when execution is stalled, it can even indicate individual lines of code that most likely hurt your performance and should be revised.
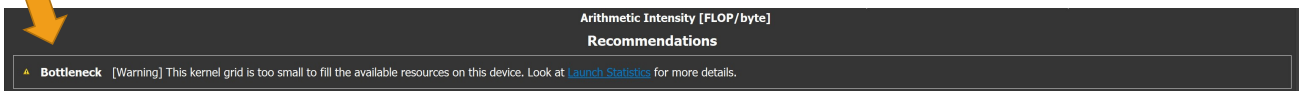
Profiling Kernels with Nsight Compute

Reims 2022

Page: Details ▾	Launch: 0 - 136 - load2	Add Baseline Apply Rules
Current	136 - load2 (1, 1, 1)x(1, 1, 1) **Time:** 72.92 msecond **Cycles:** 48,731,059 **Regs:** 16 **GPU:** Quadro RTX 3000 **SM Frequency:** 663.62 cycle/usecond **CC:** 7.5 **Process:** [23484] CudaTest2.exe	

▾ GPU Speed Of Light

High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

SOL SM [%]	0.10	Duration [msecond]	72.92
SOL Memory [%]	0.01	Elapsed Cycles [cycle]	48,731,059
SOL L1/TEX Cache [%]	0.27	SM Active Cycles [cycle]	1,611,755.97
SOL L2 Cache [%]	0.00	SM Frequency [cycle/usecond]	663.62
SOL DRAM [%]	0.01	DRAM Frequency [cycle/usecond]	140.75

GPU Utilization

- Each report section contains brief description of analyzed metrics

Arithmetic Intensity [FLOP/byte]

Recommendations

⚠ **Bottleneck** [Warning] This kernel grid is too small to fill the available resources on this device. Look at Launch Statistics for more details.

- Identifies apparent issues and suggests possible solutions

The report that is produced by Nsight contains multiple sections, each of which is concerned with a particular performance aspect. Nsight will provide a short explanation for what a particular metric is trying to measure and, in case there are apparent issues, will suggest further resources or steps to resolve bottlenecks and alleviate performance penalties.

Catching Errors

- For synchronous CUDA functions (not kernel calls), check return value
  - Return value should always equal `cudaSuccess`
  - If not, use `cudaGetErrorString` for comprehensive description

- For asynchronous functions and kernels, synchronize to retrieve error
  - After kernel, call `cudaDeviceSynchronize` and check its return value
  - Can always get and clear last reported error via `cudaGetLastError`

```cpp
kernel<<<gridDim, blockDim>>>();
cudaError_t err = cudaDeviceSynchronize();
if (err != cudaSuccess)
{
    const char* errorMessage = cudaGetErrorString(err); /*Handle error*/
}
```

While it is important to know whether or not programs are efficient, it is more important still to ensure that they are correct. Most functions that the CUDA runtime API provides return an error code that either reports errors of the called function itself if it executes synchronously or errors of asynchronous, previously executed functions. This makes reacting to a particular error slightly tricky if asynchronous functions, like kernel launches, are involved. However, when an application is known to have errors, they can be easily pinpointed by securing suspicious sections with synchronization commands, which will always return errors caused by previous asynchronous commands. Alternatively, at any point during the program, the functions `cudaGetLastError` or `cudaPeekLastError` may be used to check whether or not an error has previously occurred.

Debugging Kernels with Nsight

- Edition for Eclipse + CUDA GDB, or Microsoft Visual Studio debugging

- Supports (conditional) breakpoints, code stepping, variable watch

- When stepping, a **focus** warp is chosen manually or automatically
  - If execution is paused, can inspect states of all resident warps and threads
  - Can choose to advance only one warp or block at a time
  - Warps that, e.g., cause a memory access violation may grab focus

However, a much more convenient way of debugging CUDA applications is by using Visual Studio Nsight or Eclipse edition. These plug-ins provide mechanisms for detailed debugging of host and device code. With Nsight, developers may use many of the tools that they already utilize for debugging on the CPU, such as breakpoints, memory watches, and local variable view for all running threads. Nsight enables code stepping as well. To do this, a focus warp must be selected, and the stepping occurs either at warp or block level, one instruction at a time. When errors or exceptions occur, other warps may automatically grab the focus to draw attention to this event.

The overview of the active threads can list them all, warp by warp, and indicate which thread is currently the focus thread. Developers are free to switch between threads and warps and inspect the local results for any of them. As shown here, threads are color-coded to indicate their state, e.g., in this case, the entire warp is an exceptional state due to a read from an illegal address. Selecting the respective warp and analyzing the content of its threads' variables should enable developers to identify what caused this error.

Sanitizing with CUDA-Memcheck Suite

- **Run as** `cuda-memcheck --tool <tool> <application>`

- `memcheck`: memory errors (access violations, leaks, API errors)

- `synccheck`: misuse of synchronization (invalid masks, conditions)

- `racecheck`: data races (read-after-write, write-after-read hazards)

- `initcheck`: evaluation of uninitialized values (global memory only)

Lastly, developers may use the `memcheck` suite to sanitize their kernels. These command-line-based tools are capable of identifying fundamental issues that may lead to faulty results, such as memory access errors, invalid use of synchronization primitives, race conditions, and failure to initialize memory.

Helpful Libraries and Tools

Reims 2022

Finally, we want to provide the aspiring CUDA developer with a short, exemplary list of libraries and tools that may be helpful for the creation of larger projects.

Examples of Commonly used Libraries

- CUB/Thrust: additional primitives and functions similar to standard library
    - Algorithms: e.g., prefix sum, scan, sort
    - Data structures and containers: e.g., vectors

- cuBLAS: basic linear algebra subprograms (BLAS) on top of CUDA

- cuFFT: efficient implementation of discrete fourier transform on the GPU

- cuSparse: algorithms and optimizations for working with sparse matrices

- TensorRT: interface to learning and inference capabilities with tensor cores

- CUTLASS: provides a range of templates for tensor core matrix computations

Compiler Explorer

- Online compiler and assembly viewer: https://godbolt.org

- Currently runs several versions of NVCC 9 through 11

- Allows for inspection of PTX and SASS machine code from C++ input

- Useful for exploring, sharing and discussing the resulting low-level instructions and effectiveness of given C++ code snippets
  - We used it a lot during the preparation of this tutorial!

References

[1] Sutter, H. (2005). "The free lunch is over: A fundamental turn toward concurrency in software".
Dr. Dobb's Journal. Vol. 30 no. 3.

[2] Difference between CUDA and OpenCL 2010

[3] NVIDIA, CUDA Programming Guide

[4] NVIDIA, Kernel Profiling Guide

[5] NVIDIA, PTX ISA

[6] NVIDIA, Nsight Systems Documentation

[7] NVIDIA, Nsight Compute Documentation

[8] NVIDIA, Developer Forums

More?

- [More on Better Performance with CPUs and GPUs](#) (Talk)
- [More on Independent Thread Scheduling](#) (Volta Architecture, Blog)
- [More on Streams](#) (Blog)
- [More on Nsight Visual Studio (Code) Edition](#) (Demo)
- [More on Nsight Systems](#) (Talk)
- [More on Nsight Compute](#) (Talk)
- [More Code!](#)

https://cuda-tutorial.github.io