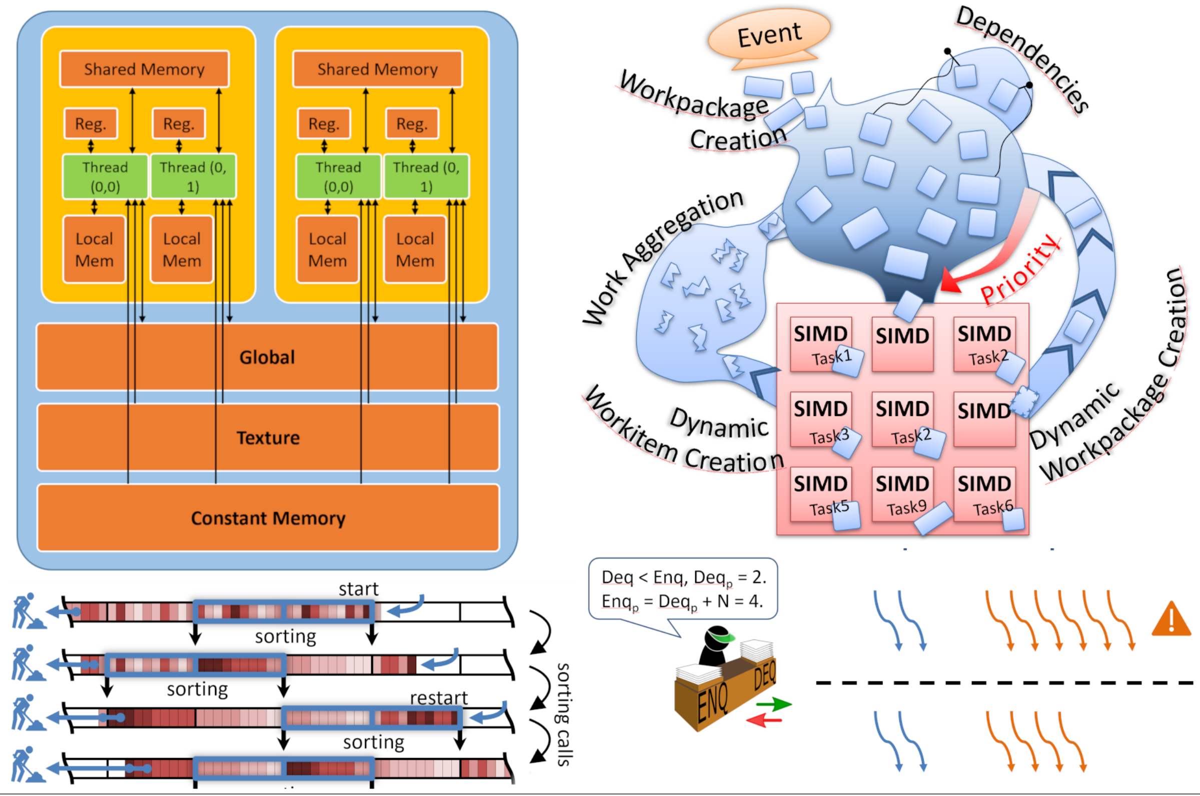
CUDA and Applications to Task-based Programming

M. Kenzel [1], B. Kerbl [2], M. Winter [3] and M. Steinberger [3]

[1]Saarland University, Computer Graphics Lab, Germany
[2]TU Wien, Institute of Visual Computing and Human-Centered Technology, Austria
[3]Graz University of Technology, Institute of Computer Graphics and Vision, Austria



Abstract

Since its inception, the CUDA programming model has been continuously evolving. Because the CUDA toolkit aims to consistently expose cutting-edge capabilities for general-purpose compute jobs to its users, the added features in each new version reflect the rapid changes that we observe in GPU architectures. Over the years, the changes in hardware, growing scope of built-in functions and libraries, as well as an advancing C++ standard compliance have expanded the design choices when coding for CUDA, and significantly altered the directives to achieve peak performance. In this tutorial, we give a thorough introduction to the CUDA toolkit, demonstrate how a contemporary application can benefit from recently introduced features and how they can be applied to task-based GPU scheduling in particular. For instance, we will provide detailed examples of use cases for independent thread scheduling, cooperative groups, and the CUDA standard library, libcu++, which are certain to become an integral part of clean coding for CUDA in the near future.

DOI: 10.2312/egt.20211037

1. Presenter Details

Michael Kenzel ✉ is a researcher at the German Research Center for Artificial Intelligence. His research interests focus on the areas of GPU programming models, high-performance computing, and real-time graphics with numerous publications at reputable venues including Eurographics, SIGGRAPH, and SIGGRAPH Asia. He has been involved in teaching courses in the areas of GPU programming as well as computer graphics for many years at two different universities.

Bernhard Kerbl ✉ is a post-doctoral university assistant at TU Wien. He obtained his PhD at Graz University of Technology for his research into GPU scheduling, real-time rendering, parallel data structures and geometry processing. He has published papers on these topics at major computer science venues, including Eurographics, ACM CHI and SIGGRAPH. His interests include real-time rendering, parallel programming and high-performance computing. Bernhard regularly reviews technical papers for top-tier venues and has been part of the IPC for the Eurographics and High-Performance Graphics conference venues. He has taught graphics and CUDA-related courses at three Austrian universities.

Martin Winter ✉ is a PhD student at Graz University of Technology, Austria, working in the GPU Computing Group at the Institute for Computer Graphics and Vision. Since joining the group, he has published several first-author papers at conferences as master student (HPEC'17) and as PhD student (SC'18, PPoPP'19, ICS'20 and PPoPP'21), as well as a number of second-author publications, even winning a best student paper award at HPEC'17. His research interests include high-performance computing, dynamic graph / resource management and task scheduling on GPUs as well as teaching (currently teaching the introductory GPU programming course at Graz University of Technology).

Markus Steinberger ✉ is an Assistant Professor at Graz University of Technology, Austria, leading the GPU Computing Group at the Institute for Computer Graphics and Vision. His biggest honors include the promotion sub auspiciis praesidentis rei publicae in 2014, being the first Austrian to win the GI Dissertation Prize, and winning the Heinz Zemanek Prize. His research interests are reflected by the numerous awards won by his papers, including ACM CHI, IEEE Infovis, Eurographics, ACM NPAR, EG/ACM HPG, and IEEE HPEC best paper.

2. Outline

In the first part of this 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 the first steps towards performance optimization. We will focus on the basic execution hierarchy, as well as the concept of warp scheduling and latency hiding. We will discuss tools for debugging and profiling, as well as the most important CUDA libraries.

In the second part, we will consider the different types of memory that CUDA provides to developers. Furthermore, we will analyze the actual behavior of the underlying hardware when responding to memory requests, and how to optimize data layouts for peak performance. We will discuss the two different layers of compiled CUDA code: PTX and SASS. We will look at some examples of the different types of machine code, and give examples of efficient and high-overhead instructions with respect to throughput and achievable occupancy on the GPU.

In the third part, we treat advanced mechanisms of CUDA that were not covered by earlier parts, novel features of recent toolkits and architectures, as well as overall trends and caveats for future developments. The relevant features that we will discuss include managed memory, independent thread scheduling details, cooperative groups, the libcu++ standard library, tensor cores, the set-aside L2 cache. For each of them, we provide use cases and, where applicable, important factors to consider when first introducing them into existing codebases, as well as pitfalls when porting legacy code to accommodate these new mechanics. We also provide our own personal recommendations for managing new GPU features.

In the final part of the tutorial, we will cover the different levels of the GPU hierarchy and how they can be exploited for different programming patterns. We then turn to task scheduling, first detailing queues on GPUs, a core component of most task scheduling approaches. Based on such queues, we then build different schemes for task scheduling on the GPUs, controlled from the CPU or entirely from the GPU. Lastly, we will hear about some examples, which greatly benefit from task parallelism and typically exhibit mixed parallelism during execution [KKM*18, SKK*12, SKB*14, KKS*17, WMZ*18, WMPS20].

3. Schedule

Full-Day Tutorial, 4×90 minutes

To provide a profound understanding of how CUDA applications can achieve peak performance, the first half of this tutorial outlines the modern CUDA architecture. Following a basic introduction, we expose how language features are linked to—and constrained by—the underlying physical hardware components. Furthermore, we describe common applications for massively parallel programming, offer a detailed breakdown of potential issues, and list ways to mitigate performance impacts. An exemplary analysis of PTX and SASS snippets illustrates how code patterns in CUDA are mapped to actual hardware instructions.

In the second half, we will focus on novel features that were enabled by the arrival of CUDA 10+ toolkits and the Volta+ architectures, such as ITS, tensor cores, and the graph API. In addition to basic use case demonstrations, we outline our own experiences with these capabilities and their potential performance benefits. We also discuss how long-standing best practices are affected by these changes and describe common caveats for dealing with legacy code on recent GPU models. We show how these considerations can be implemented in practice by presenting state-of-the-art research into task-based GPU scheduling, and how the dynamic adjustment of thread roles and group configurations can significantly increase performance.

1. Fundamentals of CUDA
   - 1.1. History of the GPU
   - 1.2. The CUDA execution model
   - 1.3. Kernels, grids, blocks and warps
   - 1.4. Building CUDA applications
   - 1.5. Debugging and Profiling
   - 1.6. Common CUDA libraries
2. Understanding the GPU hardware
   - 2.1. The CUDA memory model
   - 2.2. Warp scheduling and latency hiding
   - 2.3. Independent thread scheduling
   - 2.4. Performance metrics and optimization
   - 2.5. Basics of PTX and SASS
3. Recent CUDA features and trends
   - 3.1. Synchronization with independent thread scheduling
   - 3.2. Graph API
   - 3.3. Arrival-wait barriers
   - 3.4. Tensor cores
   - 3.5. Set-aside L2 cache
   - 3.6. libcu++: a standard library for CUDA
   - 3.7. Global memory vs. texture memory
   - 3.8. Shared memory vs. the L1 cache
4. Task-based CUDA programming
   - 4.1. Programming on different levels of the GPU hierarchy
   - 4.2. Persistent threads and megakernels
   - 4.3. Dynamic parallelism and task-queues
   - 4.4. GPU queues
   - 4.5. Dynamic memory management
   - 4.6. Mixed-parallelism usage scenarios: image processing, software rasterization, mesh subdivision, building spatial acceleration structures and more

4. Intended Audience

The target audience possesses basic to advanced knowledge of parallel algorithms and graphics APIs. This tutorial intends to attract viewers with a strong interest for understanding and optimizing for the underlying mechanisms of parallel execution on GPU hardware. Senior developers get a chance to acquaint themselves with recent CUDA features and their impact on kernel design. Furthermore, the audience is introduced to task-based applications of CUDA beyond the classic many-kernel programming pattern.

5. Sample Course Notes

Ever since compute capability 3.0 (Kepler), CUDA has had support for the basic concept of unified memory. The methods for managing it allow for a significant amount of control, even on devices where it is not supported directly by the system allocators. The fundamental additions to the CUDA architecture that managed memory provides are the `__managed__` keyword for defining variables in memory, as well as the cudaMallocManaged method to allocate storage on the host side. The managed memory will automatically be migrated to the location where it is accessed, without explicit commands to trigger the transfer. This solution decouples the handle to a memory range from its actual physical storage, which is transient and may change multiple times during execution. Initially, there was a noticeable performance penalty associated with the use of unified memory, but recently, managed memory has experienced a significant boost, making it much more practical than it used to be in addition to simplifying the code base, so we will quickly revisit it here.

With unified or managed memory, both the CPU and GPU may try to access the same variables at the same time, since kernel launches and CPU-side execution are asynchronous. While it is now possible on some systems to have concurrent accesses, older cards with compute capability lower than 6.0 and even moderately modern ones may not support it. In this case, the CPU must ensure that its access to managed memory does not overlap with kernel execution. This can for instance be achieved with synchronization primitives.

Important performance guidelines for managed memory is the avoidance of excessive faulting since this negatively impacts performance. Furthermore, it should be ensured that data is always close to the processor that accesses it. Lastly, when memory is often migrated between host and device, this can quickly lead to thrashing, which is detrimental to performance as well. Managed memory has recently been made significantly more effective, insofar as the migration of data can now occur with a fine-granular page faulting algorithm, which somewhat alleviates these problems. However, developers can additionally provide hints that make memory management easier at runtime. In order to do so, they can „prefetch“ memory to a certain location ahead of it being used. Furthermore, developers can define general advice on the utilization of memory to indicate the preferred location of physical storage, the devices where it should remain mapped, and whether or not the access is governed by reading rather than writing.

Next up, we will take another look at some of the details of Independent Thread Scheduling, which was introduced with the Volta architecture. We previously discussed the behavior of ITS, and how it enables for instance use cases where threads in the same warp may wait on each other, which would have caused a deadlock with legacy scheduling. However, with guaranteed progress, such algorithms are now safe to implement in CUDA.

The switches to disable or enable ITS are listed here. Currently, GPU models still support both modes, so it is possible to run the previous example on newer GPUs with ITS enabled/disabled to see the results. It is not yet certain if legacy scheduling will eventually be abandoned in favor of ITS, however, other GPU compute APIs, like OpenGL‘s compute shader, appear to default to legacy scheduling for compatibility reasons.

There are of course a few limitations to ITS. First of all, ITS cannot absolve developers of improper parallel coding. While it can in fact take care of deadlocks, it is still very much required of developers to be aware of the scheduling model of GPUs to make sure they can avoid live locks as well. Second, ITS can only provide a progress guarantee for threads and warps that are resident at any point in time. That is, in case of a large launched grid, if the progress of threads depends on a thread that was not launched until all SMs were filled up, the system cannot progress and will hang, since resident warps are not switched out until they complete execution. Lastly, ITS, due to the fact that it is not guaranteed to reconverge, may break several assumptions regarding warp level programming. In order to ensure a fully or partially reconverged warp, developers must make proper use of `__syncwarp` and can no longer assume lockstep progress at warp level, which is a hard habit to break.

`__syncwarp` may, at first glance, seem like a smaller version of syncthreads, however, it has a number of interesting peculiarities that make it more versatile. Most importantly, `__syncwarp` is parameterized by a mask that indicates the threads that should participate in synchronization, in contrast to syncthreads, which must always include all non-exited threads in the block. `__syncwarp` may be executed from different points in the program, enabling for instance a warp to synchronize across two different branches, as long as the masks match. If optimizations at warp-level are made by developers, in order to write correct code, they will need to make generous use of `__syncwarp` in many common patterns.

In the next section, we will consider the CUDA graph API. Many applications consist of not one, but a larger number of kernels that are in some way pipelined or processed iteratively. Usually, the nature of the computations that must occur does not change significantly, and a program performs the same steps in the same order for a number of iterations. A good example would for instance be the simulation of game physics, where in each frame, several small, incremental updates are made to achieve adequate precision. These applications can often easily be expressed in the form of a graph, where each step represents a node and edges indicate dependencies. CUDA graphs enable the definition of applications with this graph structure, in order to separate the definition of program flow and execution.

When one places a kernel into a stream, the host driver performs a sequence of operations in preparation for the execution of the kernel. These operations are what are typically called "kernel overhead". If the driver, however, is aware of the program structure and the operations that will be repeatedly launched, it can make optimizations in preparation for this particular workload. In order to enable the driver to exploit this additional knowledge, developers can construct these graphs either from scratch or existing code. CUDA Graphs support fundamental node types that suffice to build arbitrary applications from their combinations. It is possible to create, attach and parameterize nodes at any point before the graphs are made final.

In CUDA without graph APIs, we rely on streams in order to define the dependencies between different CUDA operations. By sorting commands into different streams, we indicate that they are not dependent on one another and can be concurrently scheduled. When using the graph API to build graphs from scratch, by default no dependencies are assumed. That is, if multiple kernel execution nodes are added to a graph without the definition of a dependency, they will execute as if they were all launched into separate streams.

When code is recorded into a graph, the conventional dependency model is assumed. For instance, if a single stream is recorded, all commands that may have potential dependencies on one another are treated as such. If multiple streams are being recorded, the commands in different streams may run concurrently. Capturing multiple streams into a graph takes a little extra care. Each captured graph must have an origin stream, and other captures streams must somehow be associated with the origin. Simply starting a capture in one stream before commands are executed in another will not suffice. In order to establish this association, one stream may for instance wait on an empty event from the origin stream. This way, the dependency of one stream on the other is made explicit and captured in the graph as well.

A highly popular topic of GPUs today is the introduction of tensor cores and their crucial part in many machine learning algorithms. For those of you who wondered what exactly it is that tensor cores do, we will now take a short look under the hood and describe what makes them tick. With the arrival of the Volta architecture, NVIDIA GPUs have added a new function unit to the streaming multiprocessors, that is, the tensor core. The number and capability of tensor cores is rising quickly, and they are one of the most popular features currently. A tensor core and its abilities are easily defined: each tensor core can perform a particular fused matrix operation based on 3 inputs: a 4×4 matrix A, a 4×4 matrix B, and a third 4×4 matrix for accumulation, let's call it C. The result that a single tensor core can compute is $A \times B + C$, which on its own does not seem too helpful. However, the strength of tensor cores originates from its collaboration with other cores to process larger constructs.

This collaboration can be achieved in one or two ways. The first is by using one of the readily-available libraries that make use of these capabilities in highly-optimized kernels, such as TensorRT, cuDNN or cuBLAS. For general purpose applications, it is recommended to use these solutions for higher performance.

However, the access to tensor cores is also exposed in CUDA directly via a separate header for matrix multiplication and accumulation of small matrices, which are usually only a part of the total input. These matrix tiles, or „fragments", can be larger than 4×4 if threads in a warp cooperate. The MMA headers define warp-level primitives, that is, tensor cores must be utilized collaboratively by all the threads in a given warp.

The performance of these computations is significant since the tensor core is optimized for this very specific operation. A tensor core can achieve 64 fused-multiply-add operations per clocks. With 8 tensor cores per SM, this leads to a vast 1024 operations performed in each cycle. However, restrictions do apply in their utilization. A common assumption is that tensor cores work directly on single-precision floating point values, however, this is only true for the accumulation part of the operation. So far, the input fragments A and B may not be 32-bit wide, but rather 16-bit half-precision or the more adaptive tf32 type, which has a bigger range than half-precision types. The choice of what data types are used as input directly affects the maximum size of the fragments that can be collaboratively computed. A common configuration, with half-precision for input fragments A and B, enables warps to compute MMA operations on 16×16 fragments. When using, e.g., tf32 for A and B instead, one of the dimensions must be halved.

Although knowing the exact functionality of tensor cores is interesting, a much more practical approach for the most common use cases, like machine learning, is to use the available libraries, like

TensorRT. The corresponding solutions support the loading and inference with network layouts in common machine learning formats, such as ONNX, and can compute results with unprecedented performance.

Let us now turn to the warp-level primitives that we haven't discussed so far. In addition to shuffling and voting, recent architectures have introduced additional primitives that provide interesting use cases for optimization. Two new exciting operations can now occur with high efficiency within a warp. One is the `__match_sync` operation, which has been enabled since Volta. Previously, we had the `__ballot` operation, which enabled us to find out for which threads in a warp a certain predicated evaluates to true. However, now threads can individually identify the threads whose value in a given register matches their own. Additionally, it is now possible to reduce results from registers to a single result with a single instruction. This functionality is accelerated in hardware with the Ampere architecture. For the first of the two, we can easily find interesting use cases. Consider for instance the task of processing a mesh. For rendering and many other geometry tasks, meshes are split into triangle batches with a given number of indices. When processing must be performed per vertex, e.g., for vertex shading, in order to exploit significant reuse of vertices in a mesh, duplicate vertices can be identified, and each unique vertex can only be shaded once. This was for instance realized in our previous work on enabling vertex reuse on the GPU in software. Previously, we addressed this by shuffling vertex indices and recording duplicates among threads. However, with the Volta architecture, this task maps to a single hardware-accelerated instruction. For the latter reduce operation, the application is more straightforward. Consider for instance the implementation of a reduction, where we used shuffling in the later stages to exploit intra-warp communication. The aggregate of different shuffle instructions can now be replaced with a single reduce instruction for the entire warp.

Lastly, another operation is made available that is strongly motivated by the introduction of ITS, and how it affects thread scheduling. With ITS, threads may no longer progress in lockstep, diverge and reconverge somewhat arbitrarily. `__activemask` is a special warp primitive, since it does not include synchronization and no mask must be provided. This means that it can be called without knowing which threads will be calling it. `__activemask` returns a set of threads about which it makes no concrete guarantees, other than that these threads are converged at the point where `__activemask` is called. If the result of this function is used as a mask, other warp-level primitives can use it to opportunistically form groups of threads that are currently converged to optimize particular computations. All of these new instructions are helpful, but they also illustrate something else: getting optimal performance out of the GPU is getting more and more intricate. Comparably simple goals, like the one realized in the example we just gave, require a lot of careful design, correct handling and interpreting of bitmasks, and remembering the individual optimizations that can be done in hardware. This may seem discouraging, especially for newcomers to CUDA. However, in addition to exposing these new low-level operations, CUDA also now provides developers with a helpful new library called cooperative groups, which encapsulates these behaviors but abstracts the low-level details for improved usability.

References

[KKM*18] Kerbl B., Kenzel M., Mueller J. H., Schmalstieg D., Steinberger M.: The broker queue: A fast, linearizable fifo queue for fine-granular work distribution on the gpu. In *Proceedings of the 2018 International Conference on Supercomputing* (New York, NY, USA, 2018), ICS '18, Association for Computing Machinery, p. 76–85. URL: https://doi.org/10.1145/3205289.3205291, doi:10.1145/3205289.3205291. 2

[KKS*17] Kerbl B., Kenzel M., Schmalstieg D., Seidel H.-P., Steinberger M.: Hierarchical bucket queuing for fine-grained priority scheduling on the gpu. *Computer Graphics Forum 36*, 8 (2017), 232–246. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13075, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13075, doi:https://doi.org/10.1111/cgf.13075. 2

[SKB*14] Steinberger M., Kenzel M., Boechat P., Kerbl B., Dokter M., Schmalstieg D.: Whippletree: Task-based scheduling of dynamic workloads on the gpu. *ACM Trans. Graph. 33*, 6 (Nov. 2014). URL: https://doi.org/10.1145/2661229.2661250, doi:10.1145/2661229.2661250. 2

[SKK*12] Steinberger M., Kainz B., Kerbl B., Hauswiesner S., Kenzel M., Schmalstieg D.: Softshell: Dynamic scheduling on gpus. *ACM Trans. Graph. 31*, 6 (Nov. 2012). URL: https://doi.org/10.1145/2366145.2366180, doi:10.1145/2366145.2366180. 2

[WMPS20] Winter M., Mlakar D., Parger M., Steinberger M.: Ouroboros: Virtualized queues for dynamic memory management on gpus. In *Proceedings of the 34th ACM International Conference on Supercomputing* (New York, NY, USA, 2020), ICS '20, Association for Computing Machinery. URL: https://doi.org/10.1145/3392717.3392742, doi:10.1145/3392717.3392742. 2

[WMZ*18] Winter M., Mlakar D., Zayer R., Seidel H.-P., Steinberger M.: faimgraph: High performance management of fully-dynamic graphs under tight memory constraints on the gpu. In *High Performance Computing, Networking, Storage and Analysis* (2018), SC '18. 2