Chair of Computational Mathematics

Technical University of Munich

GPU Programming using CUDA

Ilya Nyrkov ✉
Technical University of Munich
✉ ilya.nyrkov@tum.de
September 4, 2025

1 Introduction

Originally designed to handle computer graphics, GPUs have evolved into powerful tools for various computational tasks. By utilizing parallel architecture, GPUs excel at processing large amounts of data simultaneously, making them ideal for computation-heavy tasks in various fields like: artificial intelligence, computational mathematics, physics, biology and many more. This article introduces GPU programming using NVIDIA's CUDA, a popular and easy-to-use programming model widely adopted for GPU tasks. We'll explore GPU fundamentals, how CUDA simplifies GPU programming, key programming concepts, and practical applications.

2 Graphics processing units

2.1 Definition of a GPU

A Graphics Processing Unit (GPU) is a specialized electronic circuit designed to accelerate the creation and rendering of images for output to a display. It achieves that by using different primitives like: triangles(polygons), lines, pixels. Other techniques used for image rendering are: texture mapping, shading, lighting.

Initially developed to handle the computational demands of rendering graphics, GPUs became more programmable and have evolved to perform complex mathematical calculations, making them essential in various computational tasks beyond graphics rendering.

2.2 Primary purpose of GPU

The primary purpose of a GPU is to offload graphics rendering tasks from the Central Processing Unit (CPU), enabling smoother visuals and freeing up the CPU to handle other tasks. In some cases CPU is not even capable of rendering with desired frame rate, lighting and shadow effects and other characteristics. This offloading is particularly useful in applications requiring real-time image processing, such as video games and interactive media like flight simulators. Modern GPUs are designed with a highly parallel structure, allowing them to process thousands of tasks simultaneously, which is ideal for tasks involving large blocks of data that can be processed in parallel.

2.3 Practical GPU vs CPU Performance Example

Let's consider a practical scenario: rendering a 3D application: flight simulator running at 60 frames per second (fps) on a standard resolution of 1920x1080 (Full HD). A simplified formula for calculating amount of compute operations per frame:

$$\frac{operations}{frame} = pixel * \frac{operations}{pixel} * \frac{1}{frame}$$

.

With 1920x1080 pixels per frame, there are approximately 2 million pixels (2,073,600 pixels/frame). Assuming a modest 500 operations per pixel (shading, lighting calculations, transformations), the total number of operations per second is:

$$2,073,600\frac{pixel}{frame}*500\frac{op}{pixel}*60\frac{frame}{sec} \approx 62.2*10^9\frac{op}{sec}$$

Consider an Intel Core i7-9700K CPU with 8 cores and base frequency (without overclocking) 3.6 GHz (3.6 billion cycles per second), suppose that 2 graphical operations can be done in one cycle, then:

$$2\frac{op}{cycle} * 3.6 * 10^9\frac{cycle}{sec} * 8 \approx 57.6 * 10^9\frac{op}{sec}$$

$$57.6 * 10^9\frac{op}{sec} < 62.2 * 10^9\frac{op}{sec}$$

Even at full utilization, CPU is not capable of rendering image with required parameters. Aside from rendering, flight simulator will require more resources to actually run simulation itself (plane position, fuel consumption, wind stream and resistance and many more).

Now, let's compare this with NVIDIA's RTX 4090 GPU with 16384 cores and base frequency 2.52 GHz, suppose that 2 graphical operations can be done in one cycle, then:

$$2\frac{op}{cycle} * 2.52 * 10^9 \frac{cycle}{sec} * 16384 \approx 82 * 10^{12} \frac{op}{sec}$$

$$57.6 * 10^9 \frac{op}{sec} << 82 * 10^{12} \frac{op}{sec}$$

This simple calculation highlights the vast computational advantage of GPUs over CPUs in handling large-scale parallel workloads such as 3D rendering. However, while GPUs excel in massive parallelism, flight simulation involves more than just rendering; it requires complex physics calculations (wind resistance, fuel consumption and many more), real-time user input processing, where CPUs are better suited. CPUs are designed for low-latency, high-frequency operations, making them essential for real-time decision-making and managing the overall simulation logic. Therefore, rather than replacing CPUs, GPUs complement them, offloading parallelizable tasks like rendering while the CPU handles critical, latency-sensitive computations. This synergy between CPU and GPU enables a more efficient and responsive flight simulator experience.

2.4 GPU vs CPU: Key Architectural Differences

CPUs and GPUs differ significantly in their architectural design, particularly in their cache hierarchies and instruction execution patterns. These differences come from the distinct nature of the workloads each processor unit is optimized for.

A CPU is designed to efficiently execute latency-sensitive, sequential, and branching-heavy tasks. These tasks are frequent in server applications like databases, authentication servers. They are also frequent in desktop applications like web browsers, text/video/image editing software.

These applications are sensitive to delays because it directly affects user's experience. A delay of 100ms is almost unnoticeable, but a delay starting from 300ms will frustrate end user.

To achieve efficiency in latency-sensitive, sequential, and branching-heavy operations, branch prediction, instruction pre-fetching and multi-level cache hierarchy is used (Figure 1).

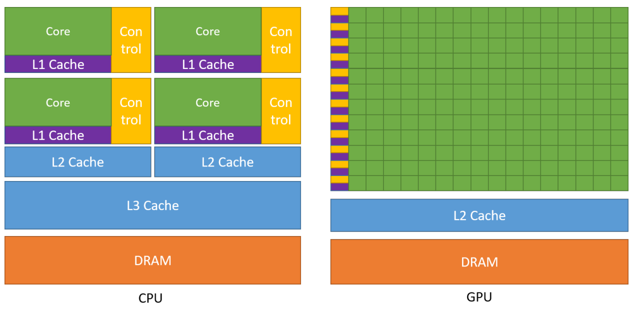


Figure 1 gpu devotes more transistors to data processing (image source: CUDA C++ programming guide)

CPU Cache hierarchy consists of:

- L1 cache (approx. size 32KB per core): Extremely fast, small-capacity cache, directly accessible by each CPU core. Split into instruction (L1i) and data (L1d) caches. Pre-fetched instructions go here for single-cycle access.
- L2 cache (approx. size in range 512KB – 2MB per core): Slightly slower than L1, larger cache used to store frequently accessed data closer to the processor and less immediate instructions (e.g., function calls after a branch).
- L3 cache (12–30 MB shared across cores): Larger and slower than L1/L2 caches. Shared across all of the cores. Used for similar purposes as L2.

CPU uses branch predictors (like L-Tage) to guess the next instruction address. For example, in a loop like:

```
for (int i=0; i<1000; i++) { func(i) }
```

the CPU predicts the loop will repeat 1,000 times and pre-fetches instructions for the loop body into L1/L2 caches.

Successful prediction and instruction pre-fetching can lead to following impact on performance:

- In case of a success: Reduces CPU pipelining stalls. For example, in a binary search, accurate prediction of the next if (key < mid) branch allows to pre-fetch the correct half of the dataset and save significant amount of cpu cycles i.e. time.
- In case of a failure: A wrong branch prediction (e.g., an unpredictable if-else based on a string that user enters) flushes the pipeline, wasting many cycles to fetch the correct instructions from RAM. We need to keep in mind that RAM is significantly slower compared to caches.

With these cache hierarchy and pre-fetching mechanisms, cpu excels at sequential tasks where temporal locality (reused instructions/data) is high. For example, a CPU parsing XML tags can keep the parseTag() function in L1i and tag data in L1d.

From the Figure 1 on the one hand we can notice that CPU devotes more transistors to caches and control logic than arithmetic logic units. On the other hand GPUs allocate most of their transistors to Arithmetic Logic Units (ALUs) rather than complex multi-level cache hierarchies and control logic. GPUs designed for compute-heavy tasks, prioritizing ALUs over caches leading to significantly more cores (e.g. Intel Core i7-9700K has 8 cores and NVIDIA RTX 4090 has 16384 cores).

GPUs don't have instruction prediction and pre-fetching in traditional CPU sense. GPUs are throughput-oriented and use SIMT (Single Instruction, Multiple Thread) type of execution. GPUs also use special primitives for execution, like threads (set of tasks/instructions for execution) and warps (set of threads which have same tasks/instructions) for NVIDIA.

There's no need for pre-fetching of divergent paths. GPUs rely on massive thread parallelism to hide latency, instead of CPU's speculative pre-fetching. For example a pixel shader applying a blur effect fetches the same texture2D() and for-loop instructions for all threads in a warp and each thread is responsible for it's pixel/subset of pixels. Here no branch prediction is needed, just all threads run same instruction.

If a shader has if (pixel.isEdge), then divergent threads serialize execution (e.g., execute all threads in the if path first, then all threads that follow the else path), causing warp divergence.

Key point here: unlike in CPUs, here pre-fetching both paths is pointless, as GPUs prioritize throughput over latency.

GPUs have simpler cache hierarchy. L2 cache is shared across all of the cores (streaming multiprocessors for NVIDIA) and focused on coalesced memory accesses for a block of threads. This type of caching is optimized for parallel tasks like matrix multiplication. When multiplying two large matrices (e.g. 10000 rows x 10000 columns), GPU threads access contiguous blocks of memory, and the cache pre-fetches entire rows/columns for reuse.

Performance Tradeoffs To conclude our CPU and GPU architecture comparsion:

- Sequential Workloads: CPUs dominate in desktop applications like editors, server applications like databases and backend, recursive algorithms with multiple branching like quicksort. GPUs struggles with these due to thread divergence and lack of pre-fetching.
- Parallel Workloads: GPUs win. For big amount of identical (e.g. matrix) operations GPUs have significantly bigger core count and can hide memory latency by switching warps while waiting for data.

3 General-Purpose GPU Computing

While we talked about GPUs computational capabilities in terms of rendering tasks like shader execution and texture mapping, their architecture's raw parallelism has unlocked a far broader domain: general-purpose computing on GPUs (GPGPU). This shift transformed GPUs from graphics-specific accelerators into versatile engines capable of solving scientific simulations, machine learning, cryptography more effective than CPUs.

3.1 Definition of GPGPU Computing

General-Purpose GPU Computing (GPGPU) is a technology used to perform general-purpose computational operations on a graphics processing unit. Unlike traditional CPU-based computing, which prioritizes low-latency sequential execution, GPGPU exploits the GPU's ability to execute thousands of threads simultaneously, focusing on throughput over individual thread speed.

3.2 SIMD and SIMT

SIMD (Single Instruction Multiple Data) is a type of a machine in Flynn's Taxonomy categorization where a single instruction is applied to multiple data elements simultaneously (Figure 2). For example, applying ADD instruction to every element in an array but each processor adds different integer value. SIMD requires lockstep execution i.e. all processors must perform the same operation at the same time. This condition leads to problems with conditional branches like if-else, switch-case.

To mitigate this issue an updated approach called SIMT (Single Instruction Multiple Threads) was introduced. Here, group of threads executes the same instruction simultaneously, but each thread operates on independent data. Crucially, SIMT allows threads to diverge, i.e. execute different branches of code. Most of the modern GPUs used for general purpose computing use SIMT approach instead of SIMD.

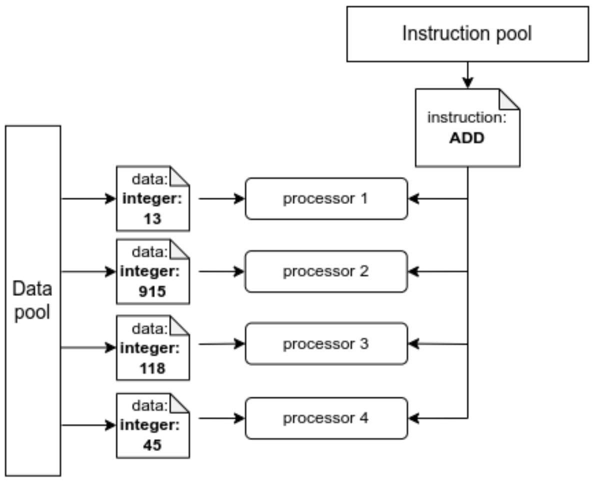


Figure 2 SIMD (single instruction multiple data)

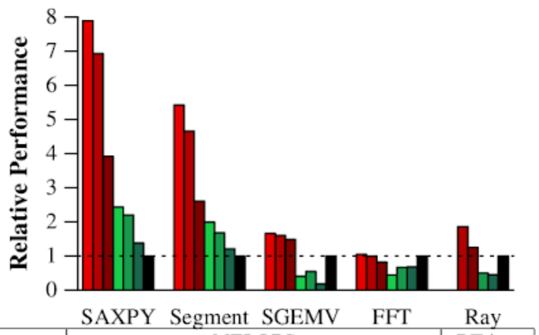
3.3 Transition from Graphics to General-Purpose Computing

Before the emergence of specialized tools, libraries, and architectures like CUDA and OpenCL, the path to general-purpose GPU computing (GPGPU) was paved by creatively reusing graphical abstractions. Early pioneers realized that the GPU's parallel architecture, designed for rendering images, could be repurposed for non-graphics tasks by reinterpreting graphics-centric concepts. Shaders, originally used to calculate lighting and textures, were adapted into general-purpose compute kernels (functions which are executed by GPU instead of CPU), while textures—traditionally storing image data—were reimagined as multidimensional arrays for scientific and mathematical computations.

3.4 Early Approaches (Brook for GPUs)

Brook framework originally introduced in PHD thesis by Ian Buck abstracted ideas further, enabling developers to write GPU-accelerated programs without relying on graphics APIs. This framework was revolutionary because it was compatible not only with top GPU vendors at that time like NVIDIA and ATI, but also with most used graphics APIs: DirectX and OpenGL. Later on after publishing his thesis Ian Buck was hired by NVIDIA to develop CUDA.

Brook introduced key abstractions like streams (collections of data elements processed in parallel) and kernels (functions which run GPUs) which simplified the process of writing GPU-accelerated programs. By abstracting the GPU as a general-purpose processor, Brook allowed developers to bypass the complexities of graphics APIs and directly express parallel com-



	MFLOPS				RT/sec
ATI					
🟥 Reference	4923	14171	2335	1278	-
🟥 BrookDX	4324	12163	2251	1212	186
🟥 BrookGL	2444	6800	2086	1003	125
NVIDIA					
🟩 Reference	1518	5200	567	541	-
🟩 BrookDX	1374	4387	765	814	50
🟩 BrookGL	861	3152	255	897	45
■ CPU	624	2616	1407	1224	100

Figure 3 Comparing the relative performance of our test applications between a reference GPU version, a Brook DirectX and OpenGL version, and an optimized CPU version. (image source: Brook for GPUs: Stream Computing on Graphics Hardware)

putations. It also made benchmarking of GPUs and graphics APIs easier, because it was no longer required to rewrite code for each of platforms.

Ian Buck showed in his paper performance of Brook on different platforms using popular computational operations as benchmarks (Figure 3).

- SAXPY: Widely used vector operation $Y := A * X + Y$, where Y, X - vectors, A is scalar value.
- Segment: This benchmark usually refers to segmented reductions or segmented scan operations, which apply computations like summation or prefix operations independently within segments of an array.
- SGEMV (Single-precision General Matrix-Vector Multiplication): $y := vAx + uy$ where u and v are scalars, x and y are vectors and A is a m x n matrix. Widely used matrix-vector operation.
- FFT (Fast Fourier Transform): an algorithm for computing the discrete Fourier transform (DFT), transforming data between the time domain and frequency domain.
- Ray (Ray tracing): technique used for computer graphics to simulate the path of light rays in 3D scenes to generate realistic images through ray tracing.

4 CUDA

CUDA - Compute Unified Device Architecture is a software and hardware environment for parallel computing and a programming model (i.e. special sets of keywords for each of programming languages C, C++, Fortran, Python and MATLAB are provided.) created by nvidia. CUDA implements general purpose gpu (gpgpu) computing and allows to significantly speed up applications by harnessing power of nvidia gpus.

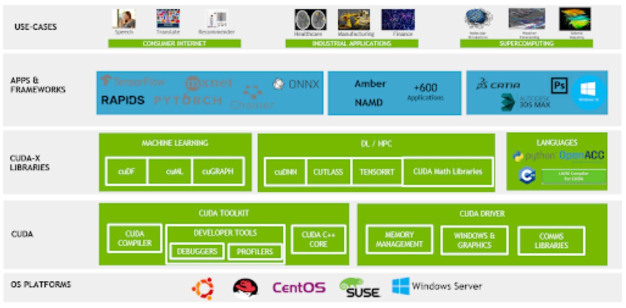


Figure 4 CUDA Tools

4.1 Overview of CUDA

CUDA comprises a rich set of tools, including over 150 CUDA-based libraries, SDKs, and powerful profiling and optimization utilities (Figure 4). These tools can be characterized in a diagram by layers:

- Operating System (OS) Platforms: CUDA is compatible with several distributions Linux and Windows. This compatibility ensures that developers can utilize CUDA across different platforms.
- CUDA: At it's core, CUDA comprises the driver (GPU hardware to OS communication) and toolkit (tools for software development).
- CUDA-X Libraries: Building on the toolkit, CUDA-X offers specialized libraries optimized for GPU performance.
  - cuDNN: Accelerates deep neural network training and inference.
  - cuBLAS: Provides GPU-optimized implementations of Basic Linear Algebra operations.
  - cuFFT: Provides fast Fourier transform capabilities.
  - cuML: Offers machine learning algorithms optimized for GPUs.
- Applications and Frameworks: At a higher abstraction level, various applications and frameworks leverage CUDA and CUDA-X libraries to deliver specialized functionalities
  - TensorFlow and PyTorch: Popular deep learning frameworks that utilize CUDA for accelerated neural network computations.
  - RAPIDS: A suite of data science and analytics tools designed for GPU acceleration, streamlining data processing workflows.
  - Visualization Platforms: Software like Autodesk and ANSYS integrate CUDA to enhance rendering and simulation performance, benefiting industries such as design and engineering.
- Use Cases: CUDA's versatility translates into applications across various sectors:
  - Healthcare: Tools like NVIDIA Clara, utilize CUDA for tasks like organ segmentation and cancer detection, improving diagnostic accuracy and treatment planning.
  - Finance: High-frequency trading firms use CUDA and NVIDIA GPUs to process vast amounts of financial data in real-time, enabling rapid decision-making.

4.2 CUDA Toolkit

CUDA Toolkit (Figure 5) is a development environment, designed to simplify GPU programming and enable developers to leverage GPU parallelism efficiently. It includes following tools:

- NVCC (NVIDIA CUDA Compiler): A compiler for CUDA programs that separates and compiles GPU and CPU code, generating optimized binaries for execution on GPUs.
- CUDA Libraries: Pre-optimized GPU libraries that allow developers to utilize common parallel operations efficiently without writing custom kernels.
- CUDA-GDB: A debugging tool for CUDA applications, facilitating debugging and performance analysis directly on GPU kernels.
- Documentation: Detailed resources, tutorials, and best practices provided by NVIDIA to assist developers in writing efficient CUDA programs (e.g. CUDA Developer Guide).

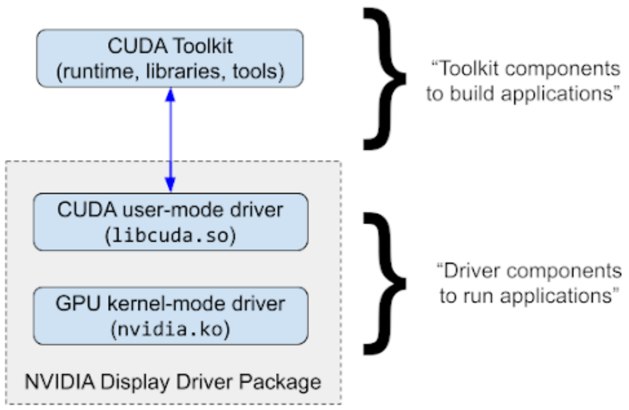

CUDA Toolkit
(runtime, libraries, tools)
"Toolkit components to build applications"
CUDA user-mode driver
(libcuda.so)
GPU kernel-mode driver
(nvidia.ko)
"Driver components to run applications"
NVIDIA Display Driver Package

Figure 5 CUDA Toolkit and Drivers (image source: NVIDIA Tesla Drivers)

4.3 CUDA Driver

GPU drivers provided by NVIDIA (Figure 5) operate in two distinct modes, each performing specific tasks:

- GPU Kernel-Mode Driver: Operates directly at the operating system kernel level, handling low-level interactions with GPU hardware. It manages critical operations, including memory allocation on the GPU, hardware resource management, kernel scheduling, and ensuring GPU hardware functions correctly within system constraints. This driver directly interacts with GPU hardware and is privileged, managing tasks inaccessible to regular user applications.
- GPU User-Mode Driver: This component exposes low-level GPU functionalities to user applications through APIs, analogous to system calls. Unlike the kernel-mode driver, the user-mode driver operates without elevated privileges, providing user-level applications controlled access to GPU features, memory management, kernel launches, and task scheduling parameters.

4.4 How to build and run CUDA program

Building and running CUDA applications involves several clearly defined stages, illustrated by the provided compilation pipeline diagram (Figure 6):

- **Preprocessing** The .cu file undergoes preprocessing to resolve macros, include headers, and expand directives. This step is handled by the CUDA compiler driver nvcc, which uses the host compiler (e.g., GCC, MSVC) for preprocessing. The output is a unified source file containing both host (CPU) and device (GPU) code.
- **CPU/GPU Code Separation** nvcc splits the preprocessed code into two streams:
  - Host Code: Standard C++ code for the CPU.
  - Device Code: Annotated GPU functions (kernels) and device-specific code (e.g. device's memory allocations).

  The separation isn't syntactic only. nvcc uses internal logic to extract device code into intermediate representations (e.g., PTX) while forwarding host code to the host compiler (e.g. gcc or Microsoft Visual C++). This is managed via the CUDA frontend.
- **Compilation of Host and Device Code**
  - **Host Code Compilation**The host code is compiled into object files (e.g., .o or .obj) using the host compiler. This generates x86/ARM/RISCV assembly optimized for sequential execution.
  - **Device Code Compilation**
    - **NVCC and libNVVM**: Device code is compiled into PTX (Parallel Thread Execution), a low-level virtual assembly language, using NVIDIA's libnvvm library. PTX is architecture-agnostic and designed for JIT compilation
    - **PTX to SASS**: During runtime or offline compilation, PTX is translated to SASS (Streaming ASSembly), the actual binary code for a specific GPU architecture (e.g., Ampere, Hopper). This step is performed by the GPU driver.
- **Fatbinary Embedding**: The compiled device code (PTX and/or cubin files) is bundled into a fatbinary, a container format that includes multiple GPU code versions for compatibility across architectures. This fatbinary is embedded into the host object file
- **Linking**: The host object files (with embedded fatbinary) are linked with CUDA runtime libraries (e.g., cudart) to resolve GPU-specific symbols (e.g., memory allocation, kernel launches). The linker produces a final executable that coordinates CPU/GPU execution
- **Runtime Execution and JIT**: At runtime, the CUDA driver dynamically compiles PTX to SASS for the specific GPU architecture. This

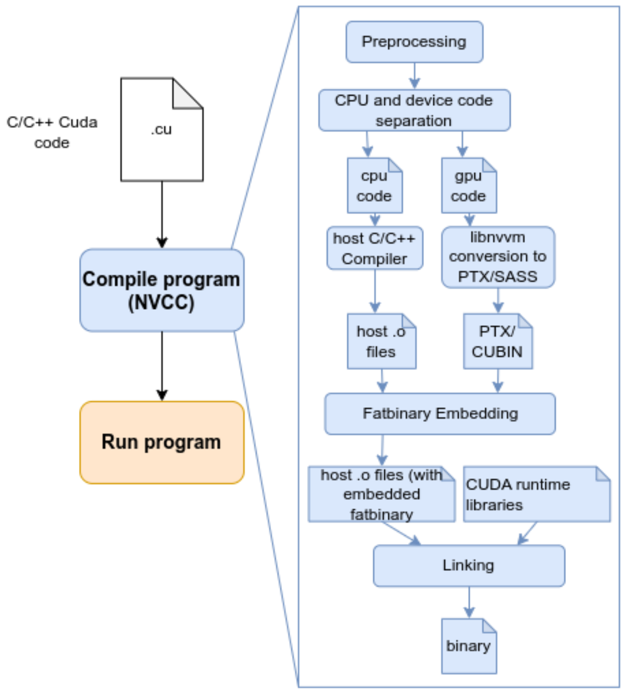


Figure 6 Building CUDA program

Just-In-Time (JIT) compilation ensures compatibility but adds a one-time overhead.

5 CUDA Terms

To effectively solve problems using CUDA, we need to understand multiple fundamental execution and memory concepts and terms. CUDA introduces specific logical abstractions such as threads, blocks, grids, warps, and streaming multiprocessors, which allow developers to structure computations efficiently on massively parallel GPU hardware. These abstractions simplify managing thousands of concurrent operations and provide intuitive ways to distribute computational tasks across GPU resources.

5.1 Thread

Thread (Figure 6) is the smallest unit of computation executed on a GPU. Each instruction given to gpu is computed by some thread within a warp. Unlike CPU threads, GPU threads are lightweight and function more like individual compute tasks or "jobs," designed to efficiently perform arithmetic and logical operations in parallel. Threads are organized into a block. Blocks of threads are organized into a grid.

Each GPU thread executes the same instruction but typically operates on different data elements—a concept known as Single Instruction Multiple Thread (SIMT). Threads are managed by Streaming Multiprocessors (SMs), hardware units within GPUs responsible for scheduling and execution. Unlike in CPU, where thread is assigned to a core, in GPU threads are assigned to SM which contain a set of cores (amount depends on a model and architecture).

The backbone of parallel programming in CUDA is the fact that each thread has built-in unique set of constants identifying its position within the overall computation hierarchy:

- threadIdx.(x,y,z) - integer identifying a thread withing a block.
- blockDim.(x,y,z) - block dimensions sizes (amount of threads per each dimension in a block).
- blockIdx.(x,y,z) - block id within a grid.
- gridDim.(x,y,z) - grid dimensions sizes (amount of blocks per each dimension in a grid).

These variables help in defining and managing parallel execution effectively by assigning each thread a unique portion of a problem to solve.

5.2 Block

Block (Figure 6) is a logical grouping of threads. Each block consists of multiple threads. Blocks can have up three dimensions. Threads are scheduled for execution in blocks. Each block is assigned to a single streaming multiprocessor (SM).

Blocks group threads logically during scheduling; however, during actual execution, threads are further organized into physical groupings called "warp". All threads within a block execute on the same streaming multiprocessor (SM). A Block cannot "migrate" to another SM during execution. Multiple SMs cannot "split" a block between them for execution.

5.3 Grid

Grid (Figure 6) is a high-level organizational structure composed of multiple thread blocks assigned to a CUDA-enabled GPU for execution. Each grid represents the entirety of parallel computation tasks executed during a single kernel (GPU function) invocation. Importantly, at any given moment, only one grid can execute per kernel (Exception is dynamic parallelism). Grid is assigned to a GPU, Blocks within this grid are assigned to multiple SMs within a GPU.

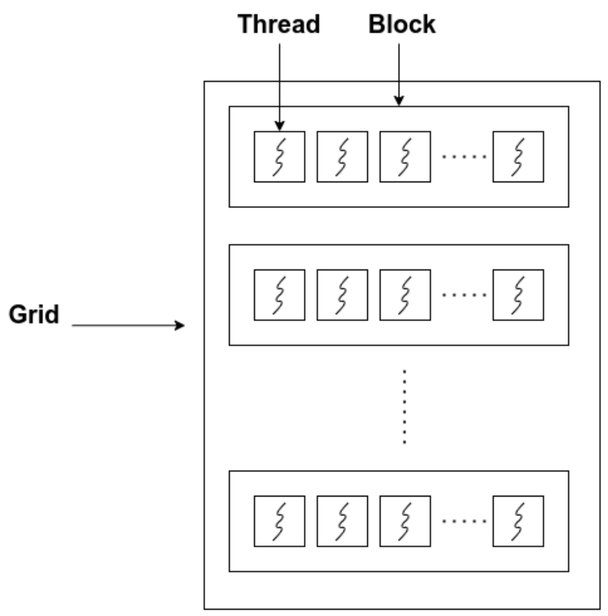


Figure 7 CUDA (Block, Grid, Thread)

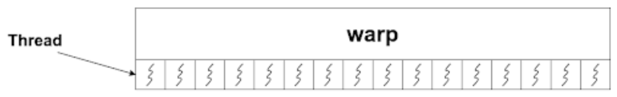


Figure 8 CUDA warp

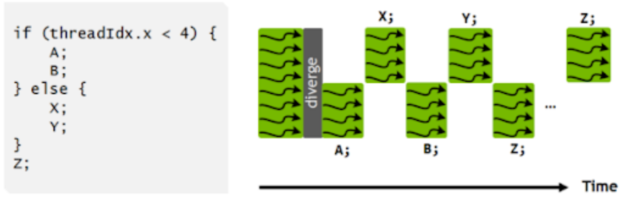
Grids serve as the logical container of thread blocks, enabling efficient distribution of tasks across GPU resources. This hierarchical structure - threads grouped into blocks, and blocks arranged into grids—allows CUDA to manage resource allocation, scheduling, and parallel execution systematically.

5.4 Warp

A warp (Figure 7) in CUDA is a fundamental execution unit comprising a fixed group of threads (typically 32) that execute the same instruction simultaneously on different data elements. Warps operate using NVIDIA's Single Instruction, Multiple Threads (SIMT) model.

Warps cannot migrate between streaming multiprocessors (SMs); each warp runs entirely within a single SM. The warp-based execution model allows GPUs to achieve remarkable parallel performance by efficiently managing thread execution and data throughput.

To manage and distribute workloads efficiently, each SM includes specialized hardware called warp schedulers. These schedulers dynamically select ready warps and dispatch their instructions to execution units (cores).



```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

Figure 9 Thread divergence

5.5 SIMT vs SIMD

CUDA uses a programming model known as Single Instruction, Multiple Threads (SIMT), an evolution from the older SIMD (Single Instruction Multiple Data) paradigm. SIMD executes a single instruction simultaneously across multiple data elements in strict lockstep model, making it efficient for simple, uniform operations without branching.

In contrast, Single Instruction, Multiple Threads (SIMT), utilized by modern GPUs, enhances flexibility by allowing threads within a warp to follow different execution paths, known as thread divergence.

5.6 Thread divergence

Thread divergence occurs when threads within a warp follow different execution paths due to conditional statements (branches). When divergence occurs, the GPU handles branches sequentially, temporarily disabling threads not following the current execution path.

Divergence does not create new threads; it only causes them to execute sequentially, causing potential performance penalties due to reduced parallel efficiency.

Consider a conditional statement (Figure 9) where threads take different branches: execution paths (A, B or X, Y) are processed sequentially (taking more time than in-parallel), and threads inactive for a given path become idle ("masked") until paths converge again. Developers typically aim to minimize divergence to maximize GPU throughput.

5.7 Kernel

Kernel is a special type of function executed on an NVIDIA GPU. Kernels are defined by adding the keyword __global__ in front of the function definition, explicitly indicating that this function will run on the GPU device (instead of CPU). For example, a kernel which runs print statement in each of the threads:

```
__global__ void hello(){
    printf("block id: %d,", blockIdx.x);
    printf("thread id: %d\n", threadId
}
```

Each print statement in a final output will have unique pair of block and thread ids, because each running thread has a unique set of these id constants (blockIdx and etc.)

When launching a kernel, developers specify its configuration parameters, including the number of blocks and the number of threads per block, allowing precise control over parallel execution.

```
int main(){
    const int BLOCKS = 2;
    const int THREADS = 4;
    hello<<<BLOCKS, THREADS>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

After kernel invocation, the GPU asynchronously executes the computation, requiring explicit synchronization (cudaDeviceSynchronize()) if the host (CPU) needs to ensure that kernel execution is complete before proceeding to the code in the main CPU thread. Result of running code from the example:

```
block id: 1, thread id: 0
block id: 1, thread id: 1
block id: 1, thread id: 2
block id: 1, thread id: 3
block id: 0, thread id: 0
block id: 0, thread id: 1
block id: 0, thread id: 2
block id: 0, thread id: 3
```

We can see $BLOCKS * THREADS = 2 * 4 = 8$ print statements. It is important to note, that the order of executing threads is not guaranteed, blocks are executed in parallel on different streaming multiprocessors within a gpu.

5.8 Multidimensional blocks/grids

CUDA supports multidimensional organization of blocks and grids (Figure 10) to simplify programming tasks that naturally fit into two or three dimensions. Threads within blocks, and blocks within grids, can be arranged in one, two, or three dimensions, making it intuitive to map parallel computations onto structures such as matrices (2D), volumes or 3D tensors, and data OLAP cubes.

Processing a 2D image or matrix multiplication naturally aligns with two-dimensional block and grid arrangements, whereas simulations involving 3D structures (like fluid dynamics or medical segmentation) benefit from 3D arrangements.

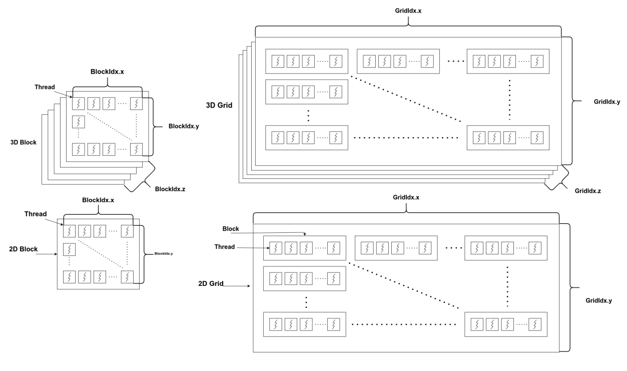


Figure 10 Multidimensional blocks and grids

Finding a sum $X := X + Y$ of matrices (Figure 11 can benefit from multidimensional blocks. Matrix can be imagined as two dimensional array, but CUDA does not allow to allocate multidimensional arrays on GPU's memory, so we need to "flatten" this array to 1D.

```
    const int M;
    // instead of
    int X[M][M];
    // use one dimensional array
    int* X;
    cudaMallocManaged(&X, M*M*sizeof(int));
```

This constrain can make understanding an algorithm more difficult. To make the job easier for us, we can make our block of threads multidimensional. In this case we use two dimensions just like in a matrix.

```
    const int M = 4;
    dim3 blockSize(M, M);
```

Then we define a kernel which will perform our operation $X := X + Y$.

```
__global__ void sum_matr(int* X,
int* Y, int N){
    int id = threadIdx.x;
    id += threadIdx.y * blockDim.x;
    X[id] = X[id] + Y[id];
}
```

Here we use row and column identifiers to find indices for each of threads. We still need to calculate identifier because arrays in video memory are flat.

Now we invoke a kernel, by assigning our 2D block size. In this case we need only one block in a grid, because we have only one matrix.

```
    dim3 blockSize(M, M);
    sum_matr<<<1, blockSize>>>(X, Y, M*M);
```

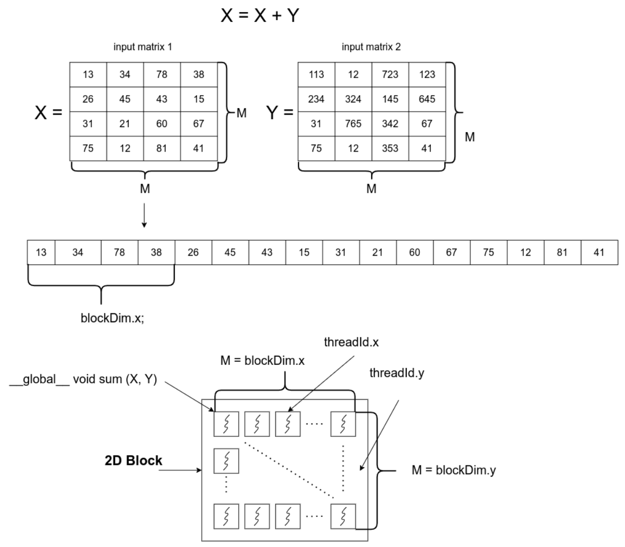


Figure 11 Sum of matrices

```
cudaDeviceSynchronize();
```

This syntax clearly indicates the dimensions of thread blocks and grids, directly reflecting the underlying data structures to simplify the parallel execution of complex problem. Worth to note that "under the hood" these are same 1D blocks and warps sent for execution. Multidimensional grids and blocks are just syntactic sugar.

5.9 Streaming multiprocessor

Streaming Multiprocessor (SM) (Figure 12) is the core hardware execution unit within NVIDIA GPUs responsible for executing CUDA kernels. Each GPU consists of multiple SMs, each capable of independently executing hundreds or thousands of threads concurrently.

An SM includes several critical hardware components (Figure 12(a)):

- **CUDA Cores (Arithmetic Logic Units - ALUs** CUDA cores are responsible for performing arithmetic and logic operations. Each SM typically includes numerous CUDA cores that simultaneously execute parallel threads. Modern GPUs contain thousands of CUDA cores distributed across many SMs.

- **Warp Schedulers** Warp schedulers manage and control warp execution within an SM. They dynamically select warps that are ready for execution and dispatch their instructions to CUDA cores, optimizing resource usage and efficiency. Warp schedulers mitigate latency by rapidly switching between warps to keep cores as utilized as possible.

- **Tensor Cores** Specialized processing units optimized specifically for matrix computations, used extensively in AI workload.

- **Shared Memory / L1 Cache** Shared memory is a high-speed on-chip memory accessible by threads within a single block running on the SM. It allows threads within a block to quickly exchange data, greatly reducing memory latency and bandwidth usage. The size of shared memory is typically limited (tens to hundreds of kilobytes per SM). Typically used for subtotal results of big computational tasks like sum of elements in a array;

- **Register File** Like in CPU, there are register files in SMs. Each SM has a large register file that provides fast, thread-local storage for temporary data required during computations. Registers are allocated to each thread, enabling rapid access to data without frequent trips to slower memory.

Threads grouped in warps execute on a single SM and utilize its hardware resources directly, making SMs the fundamental execution units in CUDA-enabled GPUs.

5.10 Counting warps per SMs

To better understand how warps, streaming multiprocessor and warp scheduler are connected, let's look at an example. Suppose we have NVIDIA RTX 4090 with 16384 cuda cores with 128 streaming multiprocessors. Older architectures run 16 threads in warp, modern ones (like our GPU) run 32 threads in one warp. First we can find a core count in each SM: $16384/128 = 128$. 128 cores per SM. Because each warp runs instruction in a group of 32 threads, we can find how many warps can run on one SM simultaneously $128/32 = 4$. 4 warps per SM. If we check tech specs of RTX 4090 we also can find that there are 4 warp schedulers per streaming multiprocessor.

5.11 Typical kernel workflow

Let's take look at a typical CUDA kernel execution workflow to better understand each of the CUDA terms introduced earlier. Take a look at Figure 13 where each step is indicated by a number.

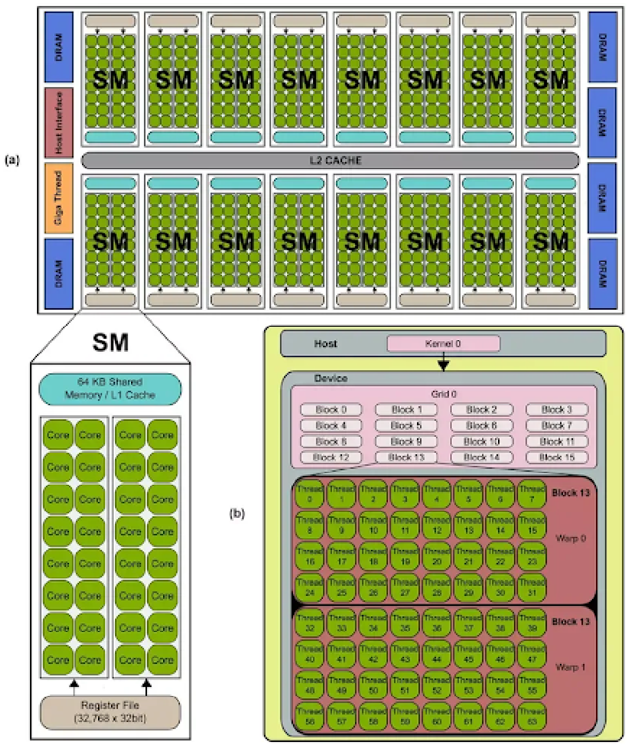


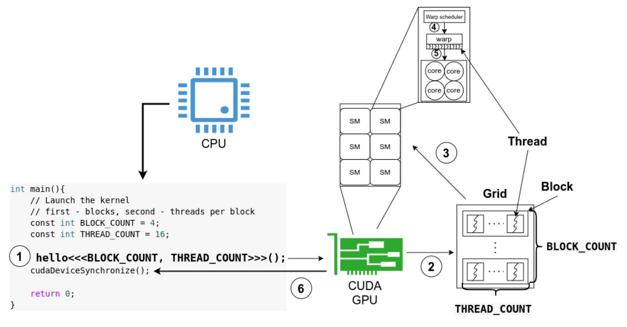
Figure 12 Typical NVIDIA GPU architecture (image source: paper "Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs")

- **Step 0: Pre-Kernel Preparations** Before the kernel is launched, the CPU performs several setup tasks to prepare the GPU for execution. Key tasks:
  - **CUDA Runtime Initialization:** The CUDA runtime initializes the GPU context and checks device availability. Device properties (e.g., compute capability, memory size) are queried.
  - **Memory Allocation on GPU:** The CPU allocates memory on the GPU for input/output data using cudaMalloc() or cudaMallocManaged() for unified memory (GPU-CPU. In our simplified example memory allocation is omitted.
  - **Data Transfer (Host → Device):** Input data is copied from CPU (host) memory to GPU (device) memory via cudaMemcpy().
- **Step 1: Kernel Launch** The CPU launches the CUDA kernel using the «<grid_dim, block_dim»> syntax. In a Figure 13 we run "hello" kernel from previous example with print statements. In CUDA kernel launches are asynchronous; the CPU continues executing code unless explicitly synchronized. We use cudaDeviceSynchronize() function which makes CPU to wait in main thread until kernel is finished. Parameters BLOCK_COUNT and THREAD_COUNT in kernel invocation define the grid (blocks) and block (threads) structure.
- **Step 2: Grid and Block Creation** The GPU creates a grid of thread blocks based on the kernel launch parameters. A grid is a 1D/2D/3D array of blocks. Each block is 1D/2D/3D array of blocks of threads. From previous sections we remember that, in the end 3D or 2D blocks/grids will be "flattened" to standard 1D on scheduling to SM stage.
- **Step 3: Block Assignment to Streaming Multiprocessors (SMs)** The GPU's Global Work Distributor assigns blocks to SMs based on resource availability (registers, shared memory). Blocks are distributed to maximize SM occupancy. Blocks do not migrate once assigned to an SM.
- **Step 4: Warp Formation and Scheduling** Threads within a block are grouped into warps (32 threads). Each warp scheduler in SM selects warps for execution and manages their execution keeping hardware as utilized as possible.
- **Step 5: Instruction Execution** The SM's execution units (CUDA cores, Tensor Cores) process instructions from active warps.
- **Step 6: Synchronization** The CPU calls cudaDeviceSynchronize() to wait for GPU kernel completion. Explicit synchronization ensures CPU-GPU data consistency. It is especially important when we copy back results from gpu to cpu and print them. If we do not synchronize devices we can print wrong results.

After kernel finished running we transfer data from gpu to host memory (if needed) and do cleanup using functions like CudaFree(). Failure to free memory causes leaks, critical in long-running applications.

5.12 Synchronization is important

Synchronization is crucial for maintaining order in inherently parallel execution and ensure both computing efficiency and correct results. Synchronization prevents race conditions by coordinating access to shared resources, such as memory, and guarantees that threads


```
int main(){
    // Launch the kernel
    // first - blocks, second - threads per block
    const int BLOCK_COUNT = 4;
    const int THREAD_COUNT = 16;

    hello<<<BLOCK_COUNT, THREAD_COUNT>>>();
    cudaDeviceSynchronize();

    return 0;
}
```

Figure 13 Typical kernel workflow

or thread groups reach a consistent state before proceeding.

CUDA offers synchronization at multiple levels:

- **Block-level: "__syncthreads()"** ensures all threads in a block reach the same point before continuing, vital for safe access to shared memory or cooperative tasks (e.g., reductions like sum or product of all elements in array). This function also acts as a memory fence, ensuring all prior writes are visible to other threads in the block.

- **Host-Device: cudaDeviceSynchronize()** blocks the CPU until all GPU operations complete, avoiding premature access to incomplete results. It's important because kernel invocation is asynchronous. CPU just goes to the next line of code after calling a kernel.

5.13 Correct synchronization is important because Deadlocks can happen

Deadlock occurs when multiple threads or groups of threads wait indefinitely for each other due to improper use of synchronization primitives like __syncthreads() or improper kernel configurations. Typically, deadlocks arise when threads within a block do not uniformly reach synchronization points.

Most common case is of deadlock is conditional branching which causes some threads to skip a synchronization primitive that others execute. Because synchronization operations require participation from every thread in a block, any divergence leading to threads waiting indefinitely results in stalled computations and severe performance degradation, potentially causing the entire GPU application to hang.

Here is an example of a code that causes deadlock.

```
// Code snippet made by
// user CygnusX1 on Jun 21 2011
// source: https://stackoverflow.com/
// questions/6426793/
// realistic-deadlock-example
// -in-cuda-opencl

__global__ void deadlockExample(){
  __shared__ int semaphore;
  semaphore=0;
  __syncthreads();
  while (true) {
    int prev=atomicCAS(&semaphore,0,1);
    if (prev==0) {
      //critical section
      semaphore=0;
      break;
    }
  }
}
```

```
int atomicCAS(int* addr, int comp, int val);
```

is an atomic CUDA function used to safely perform read-modify-write operations on shared or global memory. The term CAS stands for "Compare-And-Swap," reflecting the operation's logic:

1. The function reads the current value stored at the memory location pointed to by (int* addr).
2. It compares this value with the provided compare value (int comp).
3. If both values are equal, it replaces the current value at the memory location with the new value (int val).
4. The original value at int* addr (before any modification) is returned to the calling thread.

Term "Atomic" here comes from actual atom definition from physics which (in simple terms) is a indivisible particle. This means that the entire compare-and-swap (CAS) operation occurs as one indivisible step. Once an atomic operation starts, no other thread can interrupt or access that memory location or change it until the operation finishes.

Example: In the code,

```
atomicCAS(&semaphore, 0, 1)
```

checks if semaphore is 0. If so, it sets semaphore to 1 and returns 0 (old value located by address). Otherwise, it returns 1.

Let's go back to our code snippet of a deadlock, take a look at Figure 14 displaying timeline of execution of threads within deadlockExample() kernel.

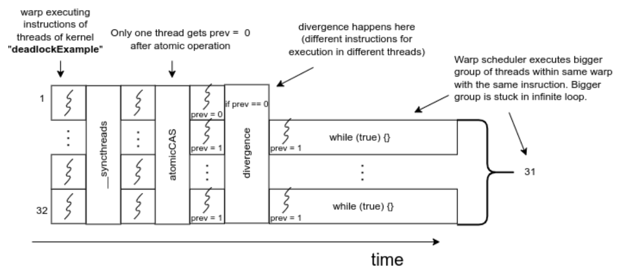

warp executing instructions of threads of kernel "deadlockExample"
Only one thread gets prev = 0 after atomic operation
divergence happens here (different instructions for execution in different threads)
Warp scheduler executes bigger group of threads within same warp with the same insruction. Bigger group is stuck in infinite loop.
1
32
__syncthreads
atomicCAS
prev = 0
prev = 1
if prev == 0
divergence
while (true) {}
31
time

Figure 14 Deadlock caused by AtomicCAS

- **Start the kernel** When kernel is invoked, blocks assigned to SM, warp scheduler in SM assigns a warp, warp starts executing instructions. Each thread stops at __syncthreads until everyone reaches this instruction.

- **AtomicCAS** Because AtomicCAS is an atomic operation, only one thread gets "prev == 0". From Figure 14 we see that it is a thread 1.

- **if prev == 0 divergence** Condition "if prev == 0" is true only for the first thread and that's where the thread divergence occur. Warp can execute only one instruction at a time. If there are different paths and different instructions, warp executes threads with different instructions sequentially.

- **while (true) infinite loop** Warp chooses the majority of threads that will be executed first. In our case it's 31 threads that have "prev == 1" and because of that stuck in a infinite loop "while (true)". This causes deadlock and our code gets stuck forever (or until we manually kill the process).

Preventing deadlocks involves ensuring synchronization primitives are used consistently and all threads within a block execute synchronization instructions identically.

In our example we can avoid the deadlock, caused by warp scheduler by omitting the break instruction and adding more operations after the if-block (where the divergence occur) that are supposed to be executed by all threads.

6 CUDA Memory models

GPUs are equipped with their own dedicated RAM, often referred to as video memory (VRAM), separate from the system's main memory (RAM). This specialized memory is designed to handle the massive parallel workloads GPUs perform, providing much high throughput compared to the limited bandwidth of system RAM. The reasoning behind having dedicated video memory is to store large datasets and textures for rendering or computation directly on the GPU, enabling much faster data access and manipulation than if it had to be fetched from RAM. This significantly reduces amount and time for data transfers between the CPU and GPU. It is a well-know performance problem for computing devices, called "The von Neumann bottleneck".

CUDA memory models are categorized into global, shared, local, and unified memory, each serving different purposes and offering specific advantages and limitations. The memory hierarchy is designed to ensure the fastest access (least amount of cycles) for threads within a block, while also supporting larger-scale data sharing between threads in different blocks.

6.1 Global memory

Global memory in CUDA is the main memory accessible by all threads across all blocks in a kernel. It has a high capacity, typically ranging from several gigabytes (e.g. NVIDIA Tesla H100 has 96gb VRAM), making it suitable for storing large datasets.

Accessing global memory comes with a significant latency (about 400–600 cycles), which can slow down computation if not managed effectively. The reason is the fact that global memory resides off-chip and requires time for data transfer between the GPU and memory through a data bus.

Example code snippet to allocate and use global memory:

```
__global__ void fill_arr(int *arr) {
    int idx = threadIdx.x;
    // accessing value in global memory
    arr[idx] = idx * 2;
}

int main() {
    int *X;
    const int N = 1024;
    // alocate global memory on GPU's VRAM
    cudaMallocManaged(&X, N*sizeof(int));
    fill_arr <<<1, N>>>(X);
    cudaDeviceSynchronize();
    cudaFree(d_array);
}
```

In the code above, we allocate a global memory buffer "X" and pass it to the kernel "fill_arr "for processing. Each thread performs operations on its corresponding memory location in the global array using threadId constants.

CUDA doesn't support multidimensional arrays like this:

```
int main() {
    const int N = 10;
    int arr[N][N];
}
```

To work with multidimensional structures like matrices, we need to "flatten" the array and calculate indices in kernels using treadIdx, blockIdx, multidimensional blocks/grids and etc.

```
__global__ void matr(int *a, int N) {
  // row number in matrix
  int row = blockIdx.y * blockDim.y;
  row += threadIdx.y;
  // collumn number in matrix
  int col = blockIdx.x * blockDim.x;
  col += threadIdx.x;
  a[row * N + col] = row * N + col;
}

int main() {
    const int N = 200;
    const int M = 300;
    // Matrix A size N x N
    int *A;
    int size = N * M * sizeof(int);
    cudaMallocManaged(&A, size);
    int blockCount = (N * M+1023);
    blockCount /= 1024 + 1;
    matr<<<blockCount, 1024>>>(X, N);
    cudaDeviceSynchronize();
}
```

6.2 Global memory coalescing

Global memory coalescing refers to the process where multiple memory accesses by threads within a warp (a group of 32 threads) are combined into a single transaction, thereby improving memory throughput.

Coalescing is important because non-coalesced accesses result in multiple memory transactions, which increases memory latency and reduces overall performance.

In CUDA, to achieve coalesced access, the following conditions should be met:

1. Memory accesses by threads in a warp must be to contiguous addresses.
2. The access pattern must align with the memory transaction size (typically 128 bytes for modern GPUs).

Example of coalesced memory access:

```
__global__ void coalesced(float *arr) {
  int idx = threadIdx.x;
  // 32 threads within the warp
  // continious indicies from 0 to 31
  arr[idx] = idx * 2.0f;
}
```

Example of non-coalesced access:

```
__global__ void nonCoalesced(int *arr) {
  int idx = threadIdx.x;
  // non continious indicies 0 2 4 ...
  arr[idx * 2] = idx * 2.0f;
}
```

6.3 Shared Memory

Shared memory in CUDA is a fast, low-latency memory region located within each Streaming Multiprocessor (SM) and is shared among all threads in a thread block. Unlike global memory, which has high access latency (400–600 cycles), shared memory offers much lower latency (10–20 cycles), making it an essential optimization tool for improving memory efficiency and reducing redundant global memory accesses. Shared memory is commonly used in reduction operations, tiling in matrix multiplication and many more.

There are also some drawbacks from using shared memory, that can occur. The total shared memory per SM is small (up to 227KB on H100), and excessive usage can lead to register spilling (fallback to global memory). Another problem are bank conflicts. Bank conflicts happen when multiple threads access the same memory bank, which leads to serialization and reduced performance.

Example of shared memory usage. Here we load part of a bigger data set "in" to each of the thread block's shared memory.

```
#include <stdio.h>

__global__ void shMemEx(int *in, int *out) {
  // Declare shared memory
  __shared__ int subArray[256];
  // Global index (index within main array)
  int idx = threadIdx.x;
  idx += blockIdx.x * blockDim.x;
```

```
// Thread index within the block
int tid = threadIdx.x;
 // Load data from global memory
 // to shared memory
subArray[tid] = input[idx];
// Ensure all threads finish loading
__syncthreads();
...
// Perform computation (double the values)
subArray[tid] *= 2;
```

6.4 Local Memory

Local memory in CUDA refers to thread-private memory that is used when a thread's register usage exceeds the available register space in a Streaming Multiprocessor (SM). While the name "local" might imply fast access, local memory is physically backed by global memory (GPU DRAM), making access to it much slower than registers or shared memory. The compiler handles register allocation and spill management, automatically moving excess variables from registers to local memory when necessary.

Each Streaming Multiprocessor (SM) in modern GPUs, such as the NVIDIA H100, has 256 KB of register file. This register file is divided among all active threads within the SM. Since each register in CUDA is 4 bytes, the total number of registers per SM is 256 KB / 4 bytes = 65536 registers. However, each individual thread can use a maximum of 255 registers (due to hardware limits), and if a kernel requires more than this, the compiler stores excess variables into local memory, which resides in global DRAM.

Since local memory resides in global memory, accessing it has a high latency (400-600 cycles), which can negatively impact performance.

6.5 Unified Memory

Unified Memory in CUDA provides a shared memory space between the CPU (host) and GPU (device), simplifying memory management by eliminating the need for explicit memory transfers. Unlike traditional global memory, which requires manual allocation, copying, and deallocation, Unified Memory allows seamless access to data across both host and device using cudaMallocManaged(). This makes programming more convenient, especially when developing applications that require frequent CPU-GPU interactions.

However, Unified Memory has performance trade-offs. Unlike global memory, where developers have full control over data placement and movement, Unified Memory relies on an automatic page migration mechanism. This means that the GPU driver dynamically moves memory pages between CPU and GPU memory, which can cause unexpected slowdowns, especially when large memory regions are transferred at unpredictable times. These memory migrations occur on-demand, leading to increased latency when data is accessed from a location where it is not currently resident. This is most problematic for benchmark application because it adds uncertainty to results (each new run can show significantly different results).

Example of unified memory usage:

```
...
cudaMallocManaged(&data,
size * sizeof(int));
const int size = 1000;
int *data;
// Allocate Unified Memory
// Initialize data on the host
for (int i = 0; i < size; i++) {
data[i] = i;
}
...
// can be used on device right away
// without moving from host
// to device memory
kern<<<(size+255)/256,256>>>(data, size);
...
```

7 Advanced CUDA Terms

In this section, we introduce advanced CUDA concepts that enable sophisticated parallel programming strategies and enhance computational performance on GPUs

7.1 Atomic operations

Atomic operations in CUDA allow threads to safely perform read-modify-write operations on shared or global memory without race conditions. These operations ensure that memory accesses occur atomically, meaning that the operations are executed without interruption by other threads.

We already explored AtomicCAS operation in deadlock example section, but there are also common atomic operations. These operations include

- addition (atomicAdd)
- subtraction (atomicSub)

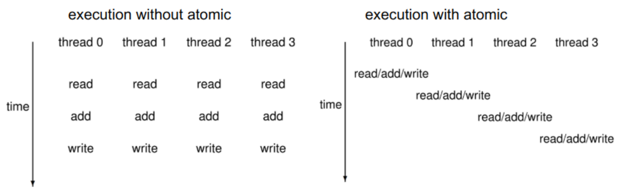


Figure 15 operations: without/with atomic

- exchange (atomicExch)
- comparison-based updates (atomicMin, atomicMax)

Without atomic operations, concurrent threads modifying the same data can lead to race conditions, causing incorrect results due to simultaneous memory reads and writes. However, atomic operations prevent such issues by ensuring that threads execute their memory operations one at a time, preserving data integrity.

Figure 15 shows that excessive usage of atomic operations can degrade performance due to serialization of memory access. For example if we need to calculate sum of elements in array we can use this code:

```
__global__ void sum(int *counter) {
    atomicAdd(counter, 1);
}
```

but this approach is extremely ineffective because add operations will be serialized instead of running at the same time. This is the main reason why it is better to use these kind of operations only in calculation of a final result. We calculate sub-sums using shared memory and regular add operations and sum all sub-sums using atomicAdd() to get a final result.

Atomic operations can also may lead to contention, limiting parallel throughput significantly if numerous threads contend frequently for the same resource.

7.2 Dynamic parallelism

Dynamic Parallelism is a feature in CUDA that allows GPU kernels to launch other kernels directly from the device, without requiring intervention from the CPU. This enables a more flexible and hierarchical execution model, where computations can be broken down into smaller tasks and managed dynamically at runtime.

In a typical CUDA program, the CPU (host) is responsible for launching kernels on the GPU (device). However, with dynamic parallelism, a kernel running on the GPU can spawn additional "child" kernels, which can execute independently.

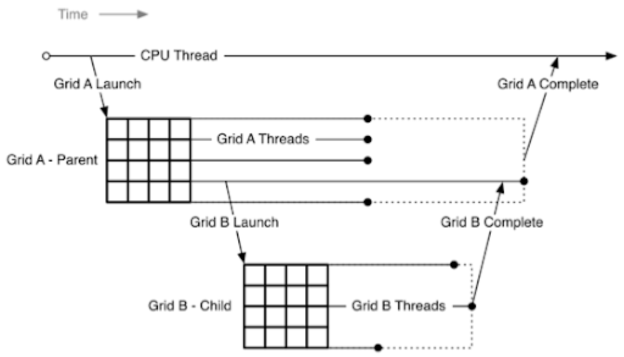


Figure 16 Parent-Child Launch Nesting (image source: CUDA C Programming Guide: Dynamic Parallelism)

Normally, in CUDA, only one grid per kernel can be launched, meaning that all thread blocks in a kernel execute as part of a single grid. However, dynamic parallelism is an exception to this rule—when a kernel launches a child kernel, it creates a new, separate grid, allowing multiple grids to exist at runtime. This hierarchical execution pattern can be seen in Figure 16, where Grid A (parent) launches Grid B (child). Execution of Grid A won't be finished until child B is finished too.

There are several benefits from using dynamic parallelism:

1. **Recursive algorithm support** Recursive functions, which were previously difficult to implement efficiently on GPUs, can now be executed directly using kernel launches from within kernels. It's a convenient feature, but there can be problems when recursion depth gets to big.

2. **Hierarchical Data Processing** Large datasets can be broken into smaller parts and processed independently within different child kernels.

3. **Reduced CPU Involvement** The CPU does not need to invoke intermediate kernels, removing time spent on CPU to GPU communication.

Kernel launches from __device__ or __global__ functions require separate compilation mode when using dynamic parallelism in CUDA. By default, CUDA expects kernel launches to be made from host (CPU) code, and device-to-device kernel launches (where a kernel launches another kernel) require separate device linking.

To compile code with device code invocation from device's kernels, a special option "-rdc=true" (Relocatable Device Code) is required.

8 Quicksort algorithm using dynamic parallelism

Quicksort is an algorithm which is used to sort elements in array in O(n * log(n)) in average case. Easiest way to implement this algorithm is by using recursion:

```
__device__ int qs_part(int *x,
int l, int r) {
   int pivot_val = x[l];
   int left = l;
   for (int i = l + 1; i < r; i++) {
      if (x[i] < pivot_val) {
         gpu_swap(&x[i], &x[left]);
         left++;
      }
}
   x[left] = pivot_val;
   return left;
}
__global__ void qs_gpu(int* x,
int l, int r) {
   if (l < r) {
      int piv_id = qs_part(x, l, r);
      qs_gpu<<<1,1>>>(x,l,piv_id);
      qs_gpu<<<1,1>>>(x,piv_id+1,r);
   }
}
```

Key components of quicksort CUDA implementation:

1. **Partitioning** The function partition() selects a pivot and rearranges elements such that smaller ones are on the left and larger ones are on the right.

2. **Recursive kernel launch** The qs_gpu kernel is launched recursively using dynamic parallelism (qs_gpu«<1, 1»>()). Each child kernel sorts a partition independently of others.

3. **Synchronization** cudaDeviceSynchronize() is needed after first kernel invokation (in CPU) code to ensure correct execution.

This implementation can be improved in many ways e.g. using multiple threads per kernel, implement shared memory optimizations for better data locality and etc. The point here is to show how easily cpu implementation can be transferred to gpu but also the problems with dynamic parallelism.

Each kernel lauch is not cheap and in our example every recursive quicksort call launches a new kernel, which means that instead of leveraging massive parallel execution, the GPU is repeatedly launching thousands of small kernels, each responsible for sorting only a fraction of the data. GPU kernel launches involve scheduling overhead, which is significant when launching many tiny kernels.

Unlike a CPU function call (which is cheap), each kernel launch incurs latency and occupies scheduler resources. If the dataset is large (e.g., 10,000 elements -> ceil(log(10000)) = 14), the sheer number of kernel launches can dominate execution time.

Each recursive partitioning step spawns a new grid (separate grid per each kernel), leading to many small grids running concurrently. The GPU scheduler is optimized for launching large workloads in parallel, but with tiny recursive partitions, the scheduler spends too much time launching grids instead of executing useful computation. This can also lead to warp underutilization which occurs because the GPU is waiting for the next batch of tiny kernel launches instead of doing bulk computations.

While dynamic parallelism provides a powerful mechanism for implementing recursive algorithms on the GPU, it must be used with caution. GPUs are optimized for bulk parallel processing, meaning that launching too many small kernels leads to scheduler overhead and poor resource utilization. Recursive algorithms, like quicksort, must be carefully implemented to avoid excessive kernel launches, deep recursion issues, and synchronization bottlenecks. Instead, use hybrid approaches that leverage iterative partitioning, parallel workloads, and warp-wide processing.

9 Mathematical functions in CUDA

CUDA provides a comprehensive set of mathematical functions optimized for high-performance computations on the GPU. These include:

- Basic arithmetic operations: add, sub, mul, div.
- Trigonometric functions: sin, cos, tan.
- Logarithmic and power functions.
- Rounding operations.

CUDA supports specialized vector types (e.g., float2, int4) that enable efficient parallel processing of multiple elements in a single operation.

Example of vector dot product using built-in cuda vector types:

```
__global__ void vectorDotProduct(
  float3 *a,float3 *b,
  float *result,int n) {
  int id = threadIdx.x;
  if (id < n) {
    // Perform elem-wise dot product
    float3 va = a[id];
    float3 vb = b[id];
    result[id] = va.x * vb.x;
    result[id] += va.y * vb.y;
    result[id] += va.z * vb.z;
    }
}
```

Half-precision floating-point (half) and complex number data types are available to optimize memory usage and performance for deep learning and scientific computing.

10 CUDA Code examples

In this section, we will explore CUDA programming through practical examples, demonstrating how different computational tasks can be parallelized efficiently on a GPU

10.1 Sum of elements

Sum of elements in array is a typical example of "reduction operation". Reduction operation reduces a set of elements to one element. Aside from sum there are other examples like: max, min, avg.

The easiest (and least effective) way to perform a reduction was mentioned in the section about atomic operations:

```
__global__ void naiveSum(int *input,
int *output, int n) {
  int idx = threadIdx.x;
  idx += blockIdx.x * blockDim.x;
  if (idx < n) {
    atomicAdd(output, input[idx]);
  }
}
```

We run atomic version of operation per each thread and each thread works with it's element from input array. The problem with this approach is the way atomic works, only one atomic operation on the same data can be executed. We are calculating sum and storing it in the same variable, which is can be accessed only be one thread at a time. This leads to basically

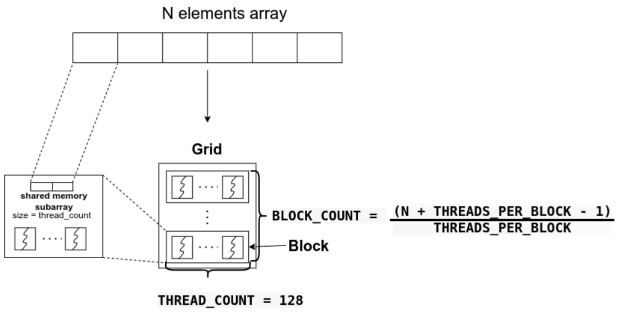


Figure 17 Processing array in multiple blocks

sequential execution of a kernel, threads execute one after another.

Reduction is common and important data parallel primitive that's easy to implement but hard to get right. The good news are, optimization techniques remain almost the same regardless of operation used in reduction: sum, multiplication, comparison (max, min) and etc.

First, we need to use multiple thread blocks in our solution to be able to process large arrays (more elements than threads in the block). By splitting the work across different blocks GPU can assign them to different streaming multiprocessors, keeping them as utilized as possible. Each block and corresponding sub-array will be reduced in parallel (Figure 17).

We instantiate shared memory array for storing subarray and calculating sum in each block:

```
// kernel declaration
__global__ void blockReduction(int *array,
int *output, int n) {
// data shared between
// threads within one block
// extern keyword allows to
// assign array length at kernel launch
extern __shared__ int subArray[];
  int tidInBlock = threadIdx.x;
  int tidGlobal = blockIdx.x*blockDim.x;
  tidGlobal += threadIdx.x;
  subArray[tidInBlock] = array[tidGlobal];
  __syncthreads();
...

// kernel invocation
int THREADS = 128;
int BLOCKS = (n+THREADS -1);
BLOCKS /= THREADS;
// third parameter is the size of shared
// memory array with extern key word
blockReduction<<<BLOCKS,
```

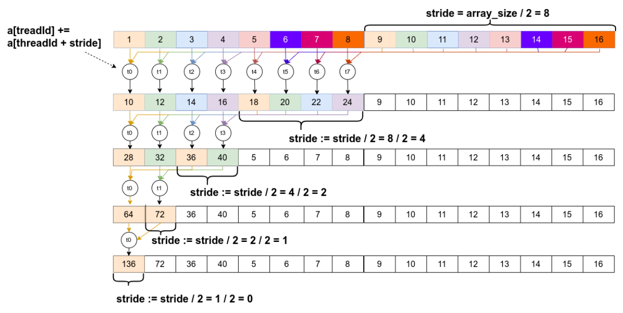

stride = array_size / 2 = 8
a[treadId] += a[threadId + stride]
stride := stride / 2 = 8 / 2 = 4
stride := stride / 2 = 4 / 2 = 2
stride := stride / 2 = 2 / 2 = 1
stride := stride / 2 = 1 / 2 = 0

Figure 18 sequential array addressing

```
THREADS,THREADS>>>(...);
```

It's also worth noting that reductions are not arithmetically intensive (1 flop per element loaded). The key bottleneck here is memory access.

Now we need actual logic how to sum elements in sub-arrays within each block. The key point here is addressing of each pair of elements that we are adding. Take a look at Figure 18, here we define a variable "stride" which is half of sub-array's length. We use this variable as an offset for each of the threads sums.

```
for (int stride = blockDim.x / 2;
stride > 0; stride /= 2) {
    if (tidInBlock < stride) {
        subArray[tidInBlock] +=
        subArray[tidInBlock + stride];
    }
    __syncthreads();
}
```

After each iteration we reduce stride value and amount of sub-sums in half. When stride becomes equal to 0 we will have final sum at index 0 in sub-array.

Finally to count the result we need to sum all sub-sums in all blocks. For this we use Atomic in first thread in each of the blocks.

```
if (tidInBlock == 0) {
  atomicAdd(output, subArray[0]);
}
```

This approach for choosing pairs of elements works but it's still not optimal solution. If we look at results, we see that starting from 1 billion naive atomic version performs faster. There are multiple reasons for this:

1. shared memory bank conflicts
2. warp under-utilization: after each iteration we reduce amount of threads doing actual work in half (even at first iteration we already cut off half of threads)
3. many __syncthreads calls introduce overhead
4. tid < stride condition causes thread divergence.

We can solve issue of half of threads staying idle in first iteration, by replacing single load to shared memory with two loads and addition:

```
extern __shared__ int subArray[];
  int tidInBlock = threadIdx.x;
  int tidGlobal = blockIdx.x*blockDim.x* 2;
  tidGlobal += threadIdx.x;
  subArray[tidInBlock] = array[tidGlobal];
  int sElem = array[tidGlobal+blockDim.x];
  subArray[tidInBlock] += sElem;
__syncthreads();
```

Also we need to half amount of blocks used in kernel invocation, because now we load two elements in shared memory load stage:

```
// kernel invocation
int THREADS = 128;
int BLOCKS = (n + THREADS − 1);
BLOCKS = (BLOCKS / THREADS) / 2;
// third parameter is the size of shared
// memory array with extern key word
kernel<<<BLOCKS,THREADS,THREADS>>>(...);
```

This code yields better results than naive atomic implementation. For 1 billion elements it takes:

- atomic: 0.066568 seconds
- new method: 0.056204 seconds

There is still room for improvements. From previous sections we remember that modern NVIDIA GPU architectures support 32 thread-warps. We can notice that in main loop, when stride variable is less than 32 only one warp is active. Instructions execute in SIMT within one warp, which means that we don't need __syncthreads() within one warp and we don't need "if stride < 32" condition too.

Last log(32) + 1 = 6 iterations can be just unwrapped as sequential instructions instead of being run within a loop. By unwrapping the loop the compiler can optimize register access patterns, reduce amount of fewer instructions that are executed overall (jump instructions used in loops and if statements).

```
__device__ void warpReduce(volatile int* s,
int tid) {
  s[tid] += s[tid + 32];
  s[tid] += s[tid + 16];
  s[tid] += s[tid + 8];
```

```
    s[tid] += s[tid + 4];
    s[tid] += s[tid + 2];
    s[tid] += s[tid + 1];
}
```

"volatile" keyword is important here, because it prevents compiler optimizations that could cache values in registers rather than reading them from memory. Using volatile in the warp reduction ensures that each thread always reads up-to-date values from shared memory, rather than using possibly outdated register-cached values.

Now we integrate it in sum loop:

```
for (int stride=blockDim.x/2;
 stride>32;stride/=2) {
 if (tidInBlock < stride) {
  int sId=tidInBlk+stride;
  subArray[tidInBlk]+=subArray[sId];
  __syncthreads();
 }
}
if (tidInBlock < 32) {
  warpReduce(subArray, tidInBlock);
}
```

This saves useless work in all warps, because now execution happens in one step for threads in the last warp. Without unwrapping every warp executes all loop iterations, even if only a few threads are contributing. This means inactive threads still perform unnecessary loop condition checks and branching which leads to thread divergence.

New computational time for 1 billion elements is 0.028024 seconds. Half of our previous compute time.

Compiler can get block sizes at a compile time by using C++ templates which also work for device code.

```
template <unsigned int blkSize>
__global__ void kern(int *in,int *o)
```

Using templates, modify function warpReduce:

```
__device__ void warpReduce(
volatile int* s, int tid) {
  if (blkSize>=64) s[tid]+=s[tid+32];
  if (blkSize>=32) s[tid]+=s[tid+16];
  if (blkSize>=16) s[tid]+=s[tid+8];
  if (blkSize>=8) s[tid]+=s[tid+4];
  if (blkSize>=4) s[tid]+=s[tid+2];
  if (blkSize>=2) s[tid]+=s[tid+ 1];
}
```

By using templates we compiler can evaluate condition "if blkSize >= N" during compilation time.

Also modify "blockReduction" i.e. kernel for sum calculation. "template" keyword needs to be added before kernel declaration:

```
template <unsigned int blockSize>
__global__ void blkReduct(int *arr, int *o)
```

In kernel invocation template parameter is stated before kernel parameters:

```
blockReduction <THREADS_PERBLOCK>
<<< BLOCKS, THREADS_PER_BLOCK,
THREADS_PER_BLOCK >>>(input, output);
```

We can get rid of loop inside "blockReduction" kernel:

```
for (int stride = blockDim.x / 2;
stride > 0; stride /= 2) {
    if (tidInBlock < stride) {
        subArray[tidInBlock] +=
        subArray[tidInBlock + stride];
    }
    __syncthreads();
}
```

And replace it with set of conditions (block size >= $2^9 \ldots 2^6$ (powers of 2):

```
int id = tidInBlock;
if (blockSize >= 512) {
  if (id < 256) {
    subArray[id] += subArray[id+256];
  }
  __syncthreads();
}
```

Because we use templates in "If (blockSize >= N)" condition, it can be evaluated at compile time, but "tidInBlock < M" condition will be evaluated at runtime affecting performance. It's still more effective than previous version with a for loop, because we get rid of extra jmp instruction used in each iteration of a for loop.

One of the possible ways to define a block size is to divide amount of cores by amount of streaming multiprocessors. This information can be looked up in vendor's tech specs for requred GPU. If we know architecture of gpu, for which our code will be launched, we can start this set of conditions from the recommended block size. For example, if we use tesla H100 there are 128 cores per SM, we can use this number as a block size and comment out useless conditions "if (blockSize >= 512)", "if (blockSize >= 256)", and "if (tidInBlock < N)" inside each of them. This approach makes our code less versatile, but key point here to show techniques that can improve performance even further. New computational time for 1 billion elements is 0.026024 seconds for block size of 128.

After implementing all of these improvements we notice that execution time has significantly reduced, but to be sure that reduction algorithm is effective we need to find it's complexity.

Let N be amount of elements in input array, then D = log(N) is an amount of parallel steps, where each step halves the number of elements that need to be processed. This results in a step complexity of O(log N).

We can check effectiveness of a new algorithm by calculating amount of operations it needs to get a result to sequential. For an array of size $N = 2^D$, the total number of operations performed is

$$\sum_{s=1}^{D} 2^{D-s} = N - 1$$

. Sequential algorithm requires same amount of operations as a new one, which means that new algorithm doesn't do any extra work and it's efficient.

Let P be amount of parallel processors, then time complexity of a reduction is $O(\frac{N}{P} + log(N))$. In case of CUDA we have blocks which can run in parallel and amount of elements N that one block processes is equal to P (amount of threads), which result to O(log(N)) which is faster than sequential O(N).

We can check compute cost of an algorithm by multiplying the number of processors used and the time complexity. If we allocate O(N) threads, the time complexity remains O(log N), making the total cost O(N log N). This is not cost-efficient compared to a sequential reduction which is O(N). We faced this problem when tried to address pairs of elements sequentially which to a lot of idle threads.

According to Brent's theorem using O(N/log N) threads is more efficient. Each thread does O(log N) sequential work, then all threads cooperate in O(log N) steps, resulting in a total cost of O(N), which is optimal.

It's worth noticing that our implementations of reduction aren't fully parallel, we used a combination of sequential and parallel reduction. It is also called algorithm cascading. Instead of having each thread sum only one element, each thread loads and sums multiple elements into shared memory.

These optimizations can be pushed even further by:

- Having ave more work per thread which will improve latency hiding.

- More threads per block reduces the number of kernel invocations.

- Avoiding high kernel launch overhead in later reduction levels prevents performance drops. There is no point in running many small kernels (like with quick sort algorithm example discussed in earlier sections).

One final reduction that can be done is by using Grid-Stride Loops.

We replace loading and summing two elements:

```
extern __shared__ int subArray[];
  int tidInBlock = threadIdx.x;
  int tidGlobal = blockIdx.x*blockDim.x* 2;
  tidGlobal += threadIdx.x;
  subArray[tidInBlock] = array[tidGlobal];
  int sElem = array[tidGlobal+blockDim.x];
  subArray[tidInBlock] += sElem;
__syncthreads();
```

With a while loop adding as many elements as necessary during loading to shared memory:

```
template <unsigned int blockSize>
__global__ void block_reduction(
int *array, int *output, int N) {
extern __shared__ int subArray[];
  int tidInBlock = threadIdx.x;
  int tidGlobal = blockIdx.x * (blockSize *
  tidGlobal += tidInBlock;
  int gridSize = blockSize*2*gridDim.x;
  subArray[tidInBlock] = 0;
  while (tidGlobal < N) {
    subArray[tidInBlock]+=array[tidGlobal];
    int id = tidGlobal+blockSize;
    subArray[tidInBlock]+=array[id];
    tidGlobal += gridSize;
  }
}
```

Lastly, we can take a look at performance graph for each reduction technique (source code) from Mark Harisses "Optimizing Parallel Reduction in CUDA" webinar (Figure 19).

- **Sequential Addressing (yellow)** shows the worst performance due to inefficient memory access patterns, warp divergence and thread underutilization.

- **First Add During Global Load (cyan)** improves performance by optimizing first iteration and reducing memory accesses, but it still under performs compared to more advanced techniques.

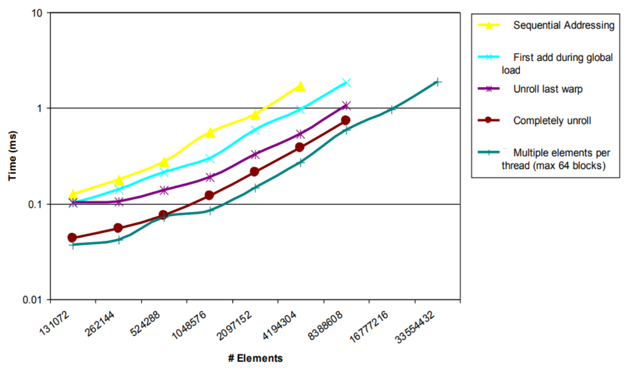


Figure 19 Performance Comparison (image source: Mark Harris: Optimizing Parallel Reduction in CUDA)

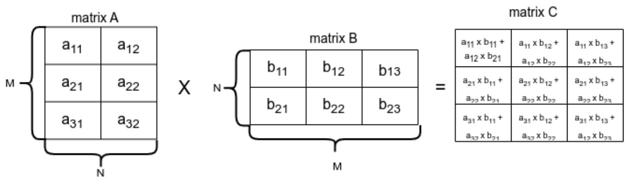


Figure 20 Matrix multiplication

- **Unroll Last Warp technique (purple)** further optimizes reduction by eliminating unnecessary synchronization in the final steps.
- **Complete Unrolling (red)** gets rid of stride loop replacing it with regular additions and conditional statements depending on block size and thread id. Block size conditions are evaluated during compile time because values are set using c++ templates.
- **Multiple Elements Per Thread (teal)** best performance is achieved by replacing two element addition with a grid stride loop.

10.2 Matrix multiplication

Matrix multiplication is a fundamental operation in scientific computing, deep learning, and numerical simulations. It involves computing the dot product of rows from the first matrix A with columns from the second matrix B, resulting in a third matrix C, as illustrated in Figure 20. This operation is computationally intensive and benefits greatly from GPU acceleration.

The easiest way to implement matrix multiplication is by using thread-per-element approach (Figure 21):

```
// Cuda doesn't support
// multidimensional arrays, so
// we "flattened" A, B, C to 1D
// N: row=column count in C
```

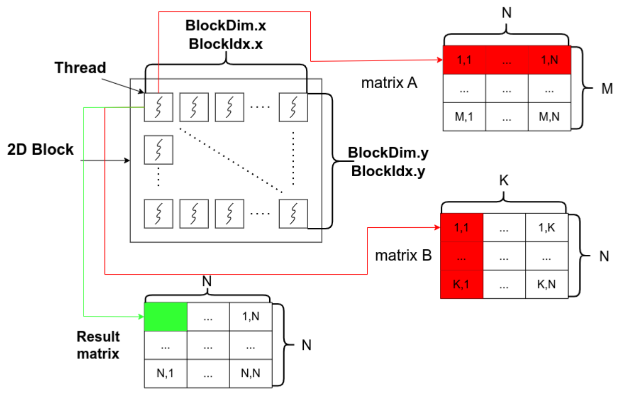


Figure 21 Matrix multiplication relative to thread blocks

```
__global__ void naiveMatMul(int *A,
  int *B, int *C, int N) {
  int row = threadIdx.y;
  row += blockIdx.y * blockDim.y;
  int col = threadIdx.x;
  col +=  blockIdx.x * blockDim.x;
  if (row < N && col < N) {
    int sum = 0;
    for (int k = 0; k < N; k++) {
      sum += A[row * N+k]*B[k * N+col];
    }
    C[row * N+col] = sum;
  }
...
// for kernel launch we use 2d blocks
// for easier access
dim3 threadsPerBlock(16, 16);
dim3 blocksPerGrid((N+15)/16,(N+15)/16);
}
```

The biggest bottleneck in this approach is global memory access overhead (which requires 600 cycles to retrieve a value).

We can improve algorithm by using shared memory for faster data access. We split the matrix in submatrices (tiles) and assign each of them to 2D thread blocks (Figure 21). Threads will collaboratively compute intermediate sums, reducing global memory accesses.

Each thread is assigned to compute a single element in result matrix C[row][col]. The thread's location in the block is identified using x and y axes:

```
int tx=threadIdx.x; // Thread x-index
int ty=threadIdx.y; // Thread y-index
int row=ty; // Row in global matrix
row+=blockIdx.y * blockDim.y
int col=tx; // Column in global matrix
col +=blockIdx.x * blockDim.x;
```

Instead of accessing global memory multiple times, each thread block loads a 16×16 tile from matrix A and matrix B into shared memory (don't forget to check if thread's row or index within the tile is out of range of original matrix's):

```
int tiles = (N+15)/16;
for (int tile=0;tile<tiles;tile++){
  if (row < N && tile * 16 + tx < N) {
    if (row<N && tile * 16+tx<N) {
      int aId = row * N+tile * 16+tx;
      tileA[ty][tx]=A[aId];
    } else {
      tileA[ty][tx] = 0;
    }

    if (col<N && tile*16+ty<N) {
      int bId = (tile*16+ty) *N+col;
      tileB[ty][tx] = B[bId];
    } else {
      tileB[ty][tx] = 0;
    }
// all threads in the block have
// finished loading their sub-matrix
// before any thread proceeds
// with calculations.
    __syncthreads();
}
```

Each thread computes the sum C[row][col] using data from the shared memory tiles:

```
for (int tile=0;tile<tiles;tile++) {
...
  for (int k=0;k<16;k++) {
    sum += tileA[ty][k]*tileB[k][tx];
  }
}
```

Instead of repeatedly accessing slow global memory, all necessary values are already stored in shared memory, making computation much faster.

After processing all of the tiles, the computed result is stored in matrix C (outside of for tile=0 loop):

```
if (row < N && col < N) {
  C[row * N+col] = sum;
}
```

This method is faster than naive because now:

- Instead of accessing 2N elements per thread using global memory, we only load small tiles into shared memory, making memory access more efficient.

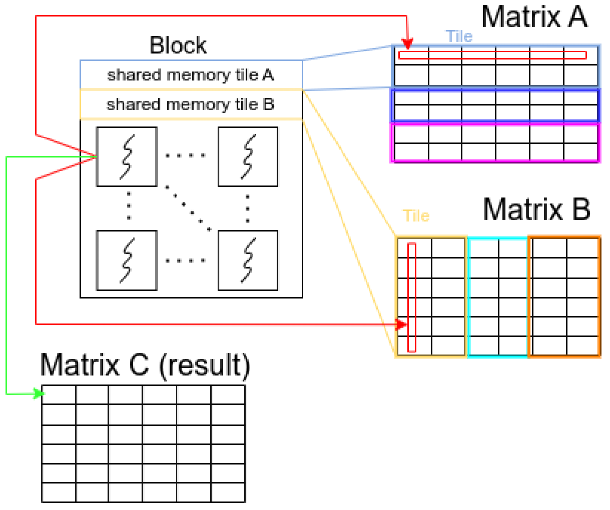


Figure 22 Matrix multiplication using tiles stored in shared memory

- Threads in a block work together to load and compute a tile, rather than acting independently like in the naive method.
- Threads access memory in a structured and coalesced manner.

10.3 Riemann Sum

The Riemann sum is a numerical method in computational mathematics used to approximate the integral of a given function over a specified interval [a, b]. This method provides a close approximation of the integral, with accuracy improving as the number of subdivisions (partitions) increases.

While the Riemann sum is unnecessary for integrals that can be solved analytically using standard calculus formulas (e.g., $\int_b^a x^2\,dx = \frac{b^3}{3} - \frac{a^3}{3}$), there are also impossible integrals e.g. $\int sin(x^2)\,dx$. Using Riemann sum we can still compute an approximate integral value over the interval [a, b].

The Riemann sum is a generalization of the rectangle method and comes in four main types: Left Riemann Sum, Right Riemann Sum, Midpoint Rule, and Trapezoidal Rule. Each of these methods follows the same core principle but differs in how sample points are chosen for function evaluation. We will take a look at left Riemann sum, as the fundamental concept remains the same across all variations.

Left Riemann sum algorithm works as follows (Figure 23):

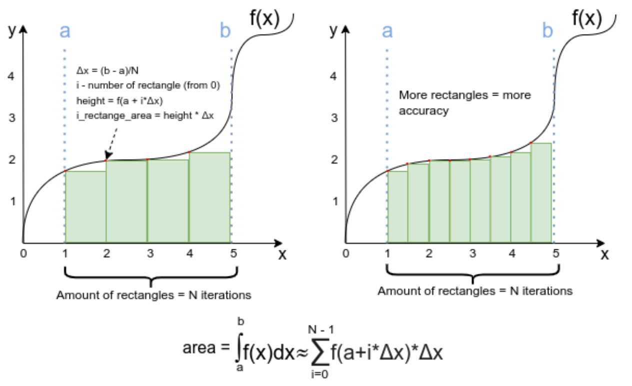

f(x)
y
a
b
Δx = (b - a)/N
i - number of rectangle (from 0)
height = f(a + i*Δx)
i_rectange_area = height * Δx
4
3
2
1
0
1
2
3
4
5
x
Amount of rectangles = N iterations
More rectangles = more accuracy
Amount of rectangles = N iterations
area = ∫ₐᵇf(x)dx≈∑ᵢ₌₀ᴺ⁻¹f(a+i*Δx)*Δx

Figure 23 Left Riemann sum

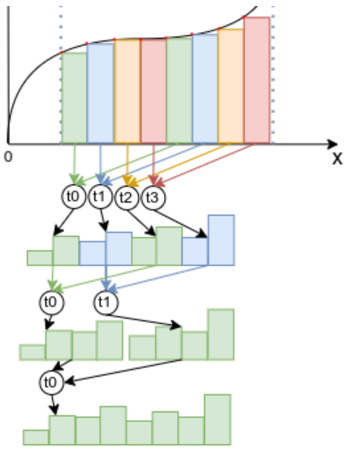


Figure 24 Left Riemann sum as a reduction operation

1. **Define the Interval and step size Δx.** The interval [a, b] is divided in N parts. The step size

$$\Delta x = \frac{b-a}{N}$$

   This represents the width of each rectangle used to calculate Riemann sum.

2. **Calculate the Height of Each Rectangle** In the Left Riemann Sum, the height of each rectangle is determined by evaluating the function at the left endpoint of each sub-interval:

$$f(a + i * \Delta x), i \in [0, N-1]$$

3. **Compute the Area of Each Rectangle** The area of each rectangle is calculated as:

$$area_i = height * width = f(a + i * \Delta x) * \Delta x$$

4. **Summing All Rectangles** The final approximation of the integral is given by summing the areas of all rectangles:

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{N-1} f(a + i * \Delta x) * \Delta x$$

   The more rectangles (larger N), the closer the approximation to the actual integral.

The computation of the Riemann sum shares many similarities with parallel reduction operations like sum, that we discussed in previous section (Figure 24), making it possible to apply the same optimization techniques with little to no modifications. The primary distinction is that, in the case of the Riemann sum, there is no explicit interaction with memory when computing the area of each rectangle; however, the subsequent summation of these areas follows the same structure as a reduction operation.

The computationally intensive part of the Riemann sum is in the initial stage, where each thread evaluates the function value and computes the area of its respective rectangle.

In a naive CUDA implementation of Left Riemann sum, we distribute the iterations among many GPU threads where each thread computes a partial sum over one or more rectangles and adds it to a global memory result variable using atomic operations.

```
__global__ void naiveRiemannSum(double a,
double dx, int N, double *result) {
    int idx = blockIdx.x * blockDim.x;
    idx += threadIdx.x;

    double sum = 0.0;
    // Each thread processes
    // multiple intervals using
    for (int i = idx; i < N;
    i += blockDim.x * gridDim.x) {
        double x = a + i * dx;
        sum += f(x) * dx;
    }

    atomicAdd(result, sum);
}
```

The main bottleneck in naive implementation is atomic operation which is required to get final sum of all rectangles. To improve performance and get rid of bottlenecks, we can use same optimization techniques we used in sum reduction operation (source code) and look at performance analysis (Tesla H100) showed on Figure 25. For the analysis function $e^x * sin(x)$ and interval [0, 10] are used.

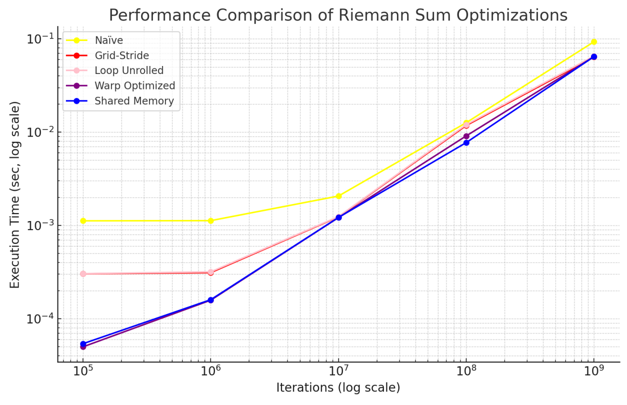


Figure 25 Performance Comparison Of Riemann Sum Optimizations.

- **Naive Implementation (Yellow)** The slowest approach as it relies on atomic operations without any optimization.
- **Grid-Stride Optimization (Red)** Improves performance slightly by allowing better load balancing among threads.
- **Loop Unrolling (Pink)** Provides a minor speedup by reducing loop overhead.
- **Warp Optimized (Purple)** Uses warp-level reductions, reducing memory accesses and improving execution speed.
- **Shared Memory (Blue)** The most optimized approach, as it avoids atomic operations for each of the rectangles and instead reduces within a block before final summation.

11 Tensor cores

Tensor Cores are specialized hardware units within Streaming Multiprocessors (SMs) on NVIDIA GPUs, designed to significantly accelerate matrix operations, particularly tensor-based computations that are fundamental in machine learning and scientific simulations. Unlike CUDA cores, which perform general-purpose computations, Tensor Cores are optimized for matrix-multiply-accumulate or Fused Multiply-Add (FMA) operations (Figure 26), allowing them to perform multiple floating-point operations per clock cycle, drastically improving performance for deep learning workloads and high-performance computing (HPC) applications.

Each Tensor Core can perform matrix multiplications such as D = A × B + C efficiently, supporting

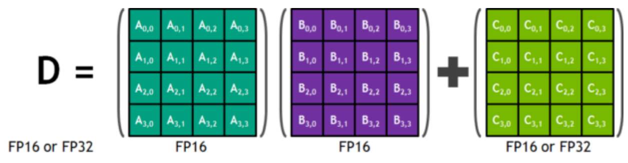


Figure 26 Tensor Core 4x4x4 matrix multiply and accumulate (image source: Programming Tensor Cores in CUDA 9)

FP16, TF32, INT8, and BF16 formats while accumulating results in FP32 precision. This specialization enables massive throughput gains, particularly in neural network training and inference tasks. For example, the NVIDIA H100 GPU 96gb (based on the Hopper architecture) features 16896 CUDA cores and 528 Tensor Cores (4 per streaming multiprocessor).

The reason why Tensor Cores are designed for FMA instead of normal matrix multiplication D = A × B is for use cases like:

- **Neural Networks (Deep Learning)** In deep learning, tensor operations frequently involve multiplying weight matrices (A × B) and adding a bias term (C). This is a fundamental step in operations like fully connected layers and convolutional layers.
- **Accumulation of Partial Sums** any algorithms require iterative updates, where the result of a multiplication is accumulated over time. The addition in FMA allows partial sums to be stored and updated efficiently without extra memory transfers.
- **Numerical Stability** MA ensures a single rounding step instead of two separate floating-point rounding operations (one for multiplication and another for addition), improving numerical precision.

Instead of performing matrix multiplication and addition operations separately (which would require storing the intermediate result of A × B before adding C), FMA fuses them into a single step, reducing memory bandwidth usage and improving computational efficiency.

Tensor Cores provide orders of magnitude higher performance compared to CUDA cores for matrix-based operations. For instance, in FP16 tensor operations, Tensor Cores can deliver over 10x higher throughput than using CUDA cores alone. This is because CUDA cores handle computations one floating-point operation per cycle per core, while Tensor Cores can compute multiple fused operations in a single step.

Despite their specialization, Tensor Cores do not operate in isolation—they work alongside CUDA cores within the SMs, allowing the GPU to execute mixed workloads efficiently. CUDA cores handle general-purpose computations and memory operations, while Tensor Cores accelerate matrix multiplications. NVIDIA's software stack, including cuBLAS, cuDNN, and TensorRT, ensures seamless utilization of Tensor Cores without requiring explicit low-level programming.

Code snippet of matrix multiplication using cuBLAS

```
// cuBLAS handle
cublasHandle_t handle;
// Creates a cuBLAS handle,
// which is required for
// all cuBLAS operations
CHECK_CUBLAS(cublasCreate(&handle));
// Alpha and beta coefficients
float alpha = 1.0f;
float beta = 0.0f;
// Tensor Core GEMM
// cublasGemmEx performs the
// generalized matrix multiplication
// C=alpha*(A*B)+beta*C
// FMA operation with coefficients
/*
CUBLAS_OP_N: Indicates no transpose
for matrices A and B.
M, N, K: Dimensions of the matrices.
alpha,beta: Scalars for the computation.
d_A,d_B,d_C: Pointers to matrices.
CUDA_R_16F: A/B are in FP16 format.
CUDA_R_32F: C is in FP32 format.
CUBLAS_GEMM_DEFAULT_TENSOR_OP:
Enables Tensor Core acceleration.
*/
CHECK_CUBLAS(cublasGemmEx(
        handle, CUBLAS_OP_N, CUBLAS_OP_N,
        M, N, K,
        &alpha,
        d_A, CUDA_R_16F, M,
        d_B, CUDA_R_16F, K,
        &beta,
        d_C, CUDA_R_32F, M,
        CUDA_R_32F,
        CUBLAS_GEMM_DEFAULT_TENSOR_OP));
```

References

[1] Jeremy Appleyard and Scott Yokim. Programming tensor cores in cuda 9, 2017. Accessed: 2025-03-07. URL: https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/.

[2] Hari Sandanagobalane Mike Murphy Mukesh Kapoor Xiaohua Zhang Arthy Sundaram, Jaydeep Marathe and Girish Bharambe. Reducing application build times using cuda c++ compilation aids, 2021. Accessed: 2025-03-07. URL: https://developer.nvidia.com/blog/reducing-application-build-times-using-cuda-c-compilation-

[3] NVIDA corp. Cuda binary utilities, 2025. Accessed: 2025-03-07. URL: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

[4] NVIDIA corp. Clara for medical imaging, 2025. Accessed: 2025-03-07. URL: https://www.nvidia.com/en-us/clara/medical-imaging/.

[5] NVIDIA corp. Cuda c++ programming guide: Introduction, 2025. Accessed: 2025-03-07. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[6] NVIDIA corp. Cuda math api reference manual, 2025. Accessed: 2025-03-07. URL: https://docs.nvidia.com/cuda/cuda-math-api/index.html.

[7] NVIDIA corp. Nvidia cuda compiler driver nvcc, 2025. Accessed: 2025-03-07. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[8] NVIDIA corp. Nvidia tesla drivers, 2025. Accessed: 2025-03-07. URL: https://docs.nvidia.com/datacenter/tesla/drivers/index.html.

[9] Wikipedia foundation's editors. Graphics processing unit, 2025. Accessed: 2025-03-07. URL: https://en.wikipedia.org/wiki/Graphics_processing_unit.

[10] Wikipedia foundation's editors. Riemann sum, 2025. Accessed: 2025-03-07. URL: https://en.wikipedia.org/wiki/Riemann_sum.

[11] Ian Buck Tim Foley Daniel Horn Jeremy Sugerman Kayvon Fatahalian Mike Houston Pat Hanrahan. Brook for gpus: stream computing on graphics hardware, 2004. Accessed: 2025-03-07. URL: https://graphics.stanford.edu/papers/brookgpu/brookgpu.pdf.

[12] Mark Harris. Cuda pro tip: Write flexible kernels with grid-stride loops, 2013. Accessed: 2025-03-07. URL: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/.

[13] Mark Harris. An efficient matrix transpose in cuda c/c++, 2013. Accessed: 2025-03-07. URL: https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/.

[14] Mark Harris. An even easier introduction to cuda, 2013. Accessed: 2025-03-07. URL: https://developer.nvidia.com/blog/even-easier-introduction-cuda/.

[15] Mark harris. Finite difference methods in cuda c/c++, part 1, 2013. Accessed: 2025-03-07. URL: https://developer.nvidia.com/blog/finite-difference-methods-cuda-cc-part-1/.

[16] Mark harris. Finite difference methods in cuda c/c++, part2, 2013. Accessed: 2025-03-07. URL: https://developer.nvidia.com/blog/finite-difference-methods-cuda-c-part-2/.

[17] Mark Harris. Optimizing parallel reduction in cuda, 2013. Accessed: 2025-03-07. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[18] Mark Harris. Using shared memory in cuda c/c++, 2013. Accessed: 2025-03-07. URL: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[19] Lara Callender Hogan. Designing for performance: Performance is user experience, 2014. Accessed: 2025-03-07. URL: https://designingforperformance.com/performance-is-ux.

[20] Yuan Lin and Vinod Grover. Using cuda warp-level primitives, 2018. Accessed: 2025-03-07. URL: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[21] Fred Oh. What is cuda, 2012. Accessed: 2025-03-07. URL: https://blogs.nvidia.com/blog/what-is-cuda-2.