



Search Creators

**Visvesvaraya Technological
University(VTU)**

Prepared By

Hanumanthu

Dedicated To.

All Engineering Students

For More Updates

 **Website:** <https://searchcreators.org/>
 **YouTube:** <https://www.youtube.com/@searchcreators7348>
 **Instagram :** <https://www.instagram.com/searchcreators/>
 **Telegram:** <https://t.me/SearchCreators>
 **WhatsApp:** +917348878215
 **Contact Number:** +91 73488 78215

Scan Us | One Stop for All



COPYRIGHT & OWNERSHIP DECLARATION

Full Ownership:

All the content in this study material is the **exclusive intellectual property of Search Creators, Tumakuru.**

It is fully **created, compiled, and owned** by Search Creators.

Source:

The material is **prepared strictly based on VTU-prescribed textbooks** under the **2022 Scheme**, ensuring syllabus accuracy.

No AI / Website Use:

-  No ChatGPT, AI tools, or online generators were used.
-  No third-party websites or digital platforms contributed content.
-  100% manually prepared by the **Search Creators academic team.**

Usage Terms:

- ❖  Minimum cost for **personal student use only** (learning, revision, internal exam prep).
- ❖  **Not allowed** for resale, upload, sharing on public platforms, or reproduction in any form without written permission.
- ❖  Strictly prohibited from being published or claimed by any other party.

Prepared & Owned By:

Search Creators, Tumakuru

Final Year Project & Academic Support Center

 **Contact: 73488 78215**

 **Email: aryahanumanthu@gmail.com**

Module -05

GPU programming with CUDA

GPUs and GPGPU

The Genesis: Graphics Demands

- **Late 1990s - Early 2000s:** A surge in demand for highly realistic computer video games and animations drove the development of specialized processors.
- **Graphics Processing Units (GPUs):** These were the result. Their initial purpose was precisely as their name suggests: to **accelerate the rendering of detailed images and video**. They excel at highly parallel computations involving vast amounts of pixel data, vertex transformations, and texture mapping.

Evolution to GPGPU

- **Temptation of Power:** Programmers not involved in graphics soon realized the immense computational power inherent in GPUs. This power, designed for visual tasks, held potential for much broader applications.
- **Early 2000s: GPGPU Emerges:** This led to the movement of **General Purpose computing on GPUs (GPGPU)**. The idea was to harness GPU power for non-graphics problems like searching, sorting, scientific simulations, and more.

Early Challenges and the Need for New APIs

Graphics-Centric Programming:

The biggest hurdle for early GPGPU developers was that GPUs could *only* be programmed using existing computer graphics APIs. This meant using libraries like **Direct3D** (Microsoft's API for Windows) and **OpenGL** (a cross-platform industry standard).

Mental Model Shift:

Programmers had to contort their general computational algorithms into graphics concepts. For example, treating data as textures, computations as shader operations on pixels, and results as rendered images. This "hack" added significant complexity and made GPGPU development very difficult and non-intuitive.

Demand for High-Level Abstractions:

The complexity quickly drove efforts to create new languages, compilers, and APIs that would allow programmers to write general-purpose code for GPUs using more conventional, high-level programming paradigms, closer to what they used for CPUs.

Modern GPGPU APIs

These efforts led to the development of dedicated GPGPU programming models:

CUDA (Compute Unified Device Architecture):

- Developed by **Nvidia** specifically for their GPUs.
- Proprietary:** It's an Nvidia-specific platform, meaning CUDA code runs only on Nvidia hardware.
- Ease of Use (Relative):** Because it targets a single vendor's hardware, it typically requires less setup and boilerplate code. This is why the text states it will be used for further discussion.

OpenCL (Open Computing Language):

- Developed by the **Khronos Group** (the same consortium behind OpenGL).
- Open Standard & Portable:** Designed to be highly portable and work across a wide range of heterogeneous computing devices, including GPUs from different vendors (Nvidia, AMD, Intel), FPGAs (Field-Programmable Gate Arrays), and DSPs (Digital Signal Processors).
- Increased Complexity (for Portability):** To achieve this broad portability, OpenCL programs generally require more explicit code for setup, device discovery, and managing execution contexts.

How GPUs Achieve Performance (Implicit Mention - SIMD)

The mention of "Execution of branch on a SIMD system" in Table 6.1 (though the table content isn't provided) implicitly refers to a key architectural principle of GPUs: **Single Instruction, Multiple Data (SIMD)**.

Table 6.1 Execution of branch on a SIMD system.

Time	Datapaths with $x[i] \geq 0$	Datapaths with $x[i] < 0$
1	Test $x[i] \geq 0$	Test $x[i] \geq 0$
2	$x[i] += 1$	Idle
3	Idle	$x[i] -= 2$

SIMD: GPUs are massive SIMD machines. They have hundreds or thousands of processing cores, but these cores often execute the *same instruction simultaneously on different pieces of data*.

Branches (Conditionals): This SIMD nature makes handling conditional statements (branches like if/else) more complex and potentially less efficient than on traditional CPUs. If threads within a SIMD group take different branches, the GPU might execute both paths serially, enabling/disabling threads as needed, which wastes computational cycles. This is a crucial consideration when writing GPGPU code.

GPU architectures

CPUs vs. GPUs in Flynn's Taxonomy

CPUs (Conventional Processors): Often characterized as **SISD (Single Instruction, Single Data)**. This means a single control unit fetches one instruction at a time and executes it on a small number of data items. This is the traditional sequential processing model.

GPUs: Are fundamentally **SIMD (Single Instruction, Multiple Data)** processors. This means a single control unit broadcasts the same instruction to multiple processing units (datapaths), and each datapath executes that instruction on its own distinct piece of data.

Understanding SIMD Architecture

Recall from Section 2.3 (presumably from the book) that a SIMD processor comprises:

- **A single control unit:** Responsible for fetching instructions.
- **Multiple datapaths:** Each datapath has its own set of registers and arithmetic logic units (ALUs) to operate on data.

How SIMD handles branches (conditionals): Consider the example:

```
if (x[i] >= 0)
    x[i] += 1;
else
    x[i] -= 2;
```

If this code runs on a SIMD system with n datapaths, where datapath i works on $x[i]$:

1. All datapaths perform the $x[i] \geq 0$ test simultaneously.
2. **Phase 1:** Datapaths for which $x[i] \geq 0$ is true execute $x[i] += 1$. All other datapaths are **idle**.
3. **Phase 2:** Datapaths for which $x[i] \geq 0$ is false (i.e., $x[i] < 0$) then execute $x[i] -= 2$. The datapaths that were active in Phase 1 are now **idle**.

This illustrates a key characteristic of SIMD: **branch divergence** (where different datapaths take different paths) can lead to serialization, as each path must be executed sequentially by the active datapaths while others wait.

Nvidia GPU Architecture (Streaming Multiprocessors - SMs)

A typical GPU can be thought of as containing one or more SIMD processors. Nvidia GPUs are structured around **Streaming Multiprocessors (SMs)**.

SMs as SIMD Processors: Each SM itself can be seen as comprising one or more SIMD processors. An SM has:

- **Several control units:** This allows for more flexibility than a single, monolithic SIMD control unit.
- **Many more datapaths (cores/SPs):** These are the actual execution units.

Asynchronous SM Operation:

A crucial aspect is that **SMs operate asynchronously**. This means there's no performance penalty if different branches of an if-else statement execute on *different* SMs.

For example, if all $x[i] \geq 0$ computations happen on SM1 and all $x[i] < 0$ computations happen on SM2, then both branches can execute concurrently across the two SMs, potentially requiring only two stages (one for each branch) for the entire GPU to complete the operation.

Nvidia Terminology:

- **Datapaths:** Nvidia calls these **cores, Streaming Processors (SPs)**. The text states it will use "SP" to avoid confusion with "CPU cores" in MIMD architectures.
- **SIMT (Single Instruction, Multiple Thread):** Nvidia's term for its SIMD architecture. This emphasizes that while threads on an SM execute the same instruction, their execution isn't necessarily perfectly simultaneous. Threads might block to hide memory access latency, allowing other threads that have their data ready to proceed. This is a form of fine-grained multithreading within the SM.

Hierarchical Memory Structure (Fig. 6.1 - conceptual):

- **Shared Memory per SM:** Each SM has a relatively small block of very fast memory that is shared among its SPs. This is often called **shared memory** in CUDA, and it's programmer-managed scratchpad memory.

- **Global Memory (Shared among all SMs):** All SMs on a single GPU chip have access to a much larger block of memory. This is the main **device memory** (often called global memory in CUDA). Accessing this memory is significantly slower than the per-SM shared memory.

Host and Device (Fig. 6.2 - conceptual)

Table 6.2 Execution of branch on a system with multiple SMs.

Time	Datapaths with $x[i] \geq 0$ (on SM A)	Datapaths with $x[i] < 0$ (on SM B)
1	Test $x[i] \geq 0$	Test $x[i] \geq 0$
2	$x[i] += 1$	$x[i] -= 2$

Physical Separation: Historically, the GPU (the **device**) and its memory were physically separate from the CPU (the **host**) and its main memory.

Explicit Data Transfer: In older systems, data had to be explicitly transferred between host memory and device memory using specific function calls (e.g., `cudaMemcpy`). This was a significant programming overhead and bottleneck.

Unified Memory (Recent Nvidia Systems - compute capability `geq3.0`):

More recent Nvidia systems support "Unified Memory" (or similar concepts). This allows the programmer to conceptually access GPU memory directly from the CPU (and vice versa) without explicit transfer calls for correctness. The system handles data migration automatically.

However, the text notes that explicit transfers *may still improve overall performance* because the automatic migration might not always be optimal.

Heterogeneous computing

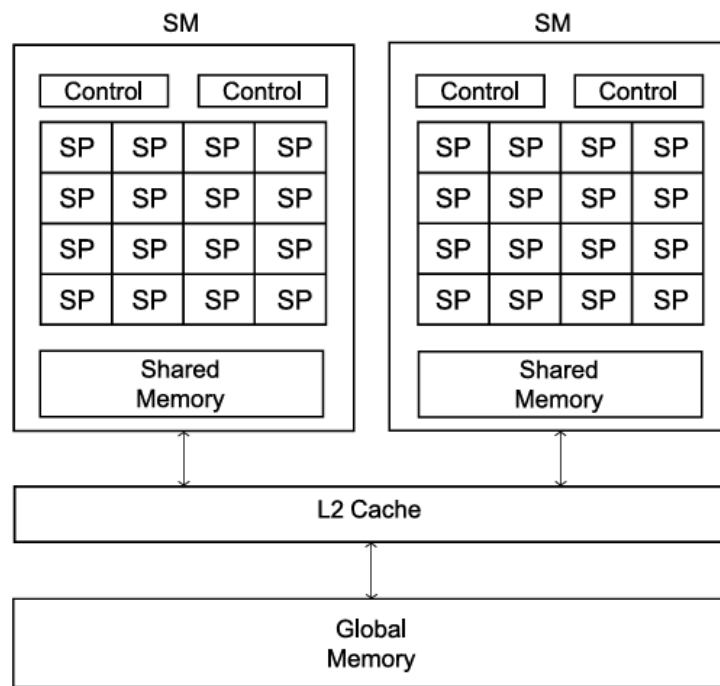
What is Heterogeneous Computing?

Definition:

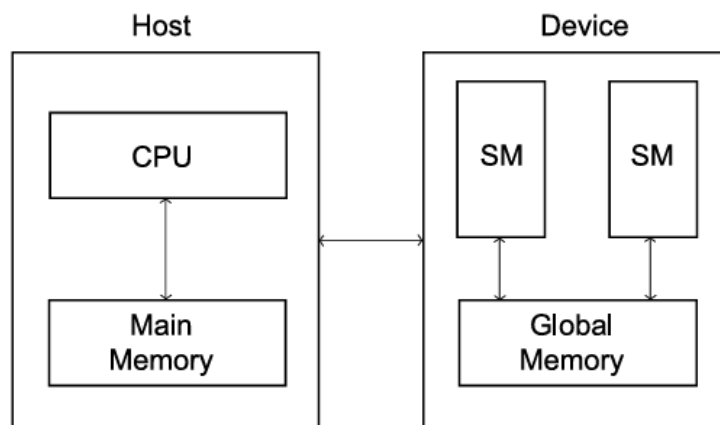
Heterogeneous computing refers to systems where different types of processors work together to execute a program. Up until this point, the discussion largely implied homogeneous systems where all processors (e.g., CPU cores in a multi-core CPU) shared the same architecture.

CPU + GPU Example:

The most common example of heterogeneous computing today is a system combining a conventional CPU (the **host processor**) with a GPU (the **device processor**).

**FIGURE 6.1**

Simplified block diagram of a GPU.

**FIGURE 6.2**

Simplified block diagram of a CPU and a GPU.

Architectural Differences:

As previously discussed, CPUs and GPUs have fundamentally different architectures (SISD/MIMD for CPUs vs. SIMD/SIMT for GPUs). This architectural difference is what makes the system "heterogeneous."

Programming Heterogeneous Systems

Single Program, Multiple Processors: Even though you're writing for different processor types, you still typically write a single overall program.

SPMD (Single Program, Multiple Data) Approach: The text notes that the SPMD approach (where the same program is run on all processors, but different parts are active on different processor types or data) is still used.

"Two Programs in One": Practically, this means you'll have:

- **Host-side code:** Functions written for and executed by the CPU. This typically handles overall program control, I/O, and sequential parts of the algorithm.
- **Device-side code (Kernels):** Functions (often called "kernels" in GPGPU contexts) written specifically for and executed by the GPU. These are the highly parallel portions of the algorithm.
- Data transfer code (explicit or implicit) between host and device memory.

Why Heterogeneous Computing is Gaining Importance

Stagnation of Single-Thread CPU Performance:

- From roughly 1986 to 2003, single-thread CPU performance increased dramatically (over 50% per year). This meant applications got faster simply by waiting for the next generation of CPUs.
- **Post-2003:** This trend significantly slowed down. Since 2003, single-thread performance improvements have fallen drastically (less than 4% per year from 2015-2017). This is often attributed to hitting physical limits (e.g., power consumption, heat dissipation) for increasing clock speeds and instruction-level parallelism.

The Search for Performance: With the "free lunch" of increasing single-core speed largely over, programmers and hardware designers are aggressively seeking alternative ways to boost performance.

Leveraging Specialized Processors: Heterogeneous computing offers a solution by offloading suitable computational tasks to specialized processors that are far more efficient at those specific tasks than a general-purpose CPU.

- **GPUs:** Excel at highly parallel, data-intensive computations, especially those that can be expressed as SIMD operations.
- **FPGAs (Field-Programmable Gate Arrays):** Contain configurable logic blocks that can be customized *prior to program execution* to implement highly optimized, application-specific hardware circuits. They offer extreme parallelism and low latency for specific algorithms but are harder to program.

- **DSPs (Digital Signal Processors):** Contain specialized circuitry optimized for manipulating and processing signals (e.g., audio, video, communications). They are very efficient at operations like filtering, compression, and Fourier transforms.

Threads, blocks, and grids

CUDA Thread Hierarchy

CUDA organizes threads in a hierarchical manner to map computations efficiently onto the GPU's hardware architecture (SMs and SPs).

Threads:

- An individual execution unit that runs the kernel code (the `__global__` function).
- Each thread executes on a **Streaming Processor (SP)**.
- A thread's unique identifier within its block is given by `threadIdx.x` (or `y`, `z` for multi-dimensional blocks).

Thread Blocks (or just "Blocks"):

- A collection of threads that execute together on a **single Streaming Multiprocessor (SM)**.
- Threads within the same block can cooperate by:
 - Sharing data through fast **shared memory** (per-SM memory).
 - Synchronizing their execution using **barriers** (e.g., `__syncthreads()`).
- The first value in the angle brackets of a kernel launch specifies the **number of thread blocks**.
- The second value in the angle brackets specifies the **number of threads per block**.
- **Independence of Blocks:** A critical requirement in CUDA is that **thread blocks must be independent**.

This means:

- A block must be able to complete its execution regardless of the state of other blocks.
- Blocks can be executed sequentially in any order or in parallel.
- This independence allows the GPU scheduler to efficiently distribute and re-distribute blocks to available SMs without needing to check dependencies between them.

Grids:

- A **grid** is the entire collection of all thread blocks launched by a single kernel call.
- It represents the total work to be done by the kernel.
- A block's unique identifier within its grid is given by `blockIdx.x` (or `y`, `z` for multi-dimensional grids).

Kernel Launch Syntax (<<<...>>>)

The triple angle brackets <<<...>>> are used in CUDA to launch a kernel onto the GPU.

KernelName<<<gridDim, blockDim>>>(arguments);

- **gridDim (First argument):** Specifies the number of thread blocks in the grid.
- **blockDim (Second argument):** Specifies the number of threads within each block.

Examples from the text:

Hello <<<1, thread_count>>>();

- Launches 1 thread block.
- This block contains thread_count threads.
- Result: All thread_count threads run on a *single* SM because a block always runs on one SM.

Hello <<<2, thread_count/2>>>(); (If thread_count is even)

- Launches 2 thread blocks.
- Each block contains thread_count/2 threads.
- Result: If the GPU has at least 2 SMs, these two blocks can potentially run on different SMs concurrently, thus utilizing more of the GPU's parallel capabilities.

Built-in CUDA Variables for Thread/Block Identification

Within the __global__ kernel function, threads can access several predefined variables to determine their unique position and the dimensions of their execution environment.

These are all dim3 structs with x, y, and z fields (though often only x is used for 1D problems).

threadIdx:

- Type: dim3
- Purpose: The unique index of the **current thread within its thread block**.
- Example: threadIdx.x gives the thread's rank in its block along the x-dimension.

blockIdx:

- Type: dim3

- Purpose: The unique index of the **current thread block within the grid**.
- Example: blockIdx.x gives the block's rank in the grid along the x-dimension.

blockDim:

- Type: dim3
- Purpose: The dimensions (size) of the **thread blocks**. This value is constant for all threads within the same kernel launch.
- Example: blockDim.x will be equal to th_per_blk from the kernel launch.

gridDim:

- Type: dim3
- Purpose: The dimensions (size) of the **grid** (i.e., the total number of blocks). This value is constant for all threads within the same kernel launch.
- Example: gridDim.x will be equal to blk_ct from the kernel launch.

Default Initialization: When you use simple integer values in the kernel launch <<<blk_ct, th_per_blk>>>, gridDim.x is set to blk_ct, and blockDim.x is set to th_per_blk. The y and z fields of these dim3 variables default to 1, effectively creating 1D blocks and grids.

Multi-dimensional Launch: You can explicitly define dim3 variables for grid_dims and block_dims to create 2D or 3D thread blocks and grids, which is useful for processing multi-dimensional data (e.g., matrices, images)

```
dim3 grid_dims, block_dims;
grid_dims.x = 2; grid_dims.y = 3; grid_dims.z = 1; // 6 blocks total
block_dims.x = 4; block_dims.y = 4; block_dims.z = 4; // 64 threads per block
// ...
Kernel <<<grid_dims, block_dims>>> (. . .);
```

This launch would create 2times3times1=6 blocks, each containing 4times4times4=64 threads.

Nvidia compute capabilities and device architectures

Nvidia Compute Capability

Definition: Compute Capability (CC) is a version number (e.g., a.b) that describes the hardware features and capabilities of a specific Nvidia GPU. It determines:

Table 6.3 GPU architectures and compute capabilities.

Name	Ampere	Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing
Compute capability	8.0	1.b	2.b	3.b	5.b	6.b	7.0	7.5

- The set of instructions the GPU can execute.
- Available hardware features (e.g., specific memory types, atomic operations).
- Performance characteristics.
- Compatibility with different CUDA API versions.

Version Numbering:

- **Major Revision Number (a):** Indicates the core architecture generation (e.g., 1, 2, 3, 5, 6, 7, 8). The text notes the absence of a major revision number 4.
- **Minor Revision Number (b):** Indicates incremental improvements or variations within a major architecture (range 0-7).

CUDA Support:

- CUDA continually evolves, and older compute capabilities may be deprecated. The text states that "CUDA no longer supports devices with compute capability < 3." This means if you have a very old Nvidia GPU with CC 1.x or 2.x, you might not be able to run recent CUDA applications or compile code for them with modern CUDA toolkits.

Hardware Limits Tied to Compute Capability:

- **Maximum Threads per Block:** For devices with CC > 1, the limit is 1024 threads per block.
- **Maximum Threads per SM:**
 - CC 2.b: 1536 threads per SM.
 - CC > 2: Currently 2048 threads per SM.
- **Block/Grid Dimension Limits:**
 - CC > 1: Maximum x- or y-dimension for blocks and grids is 1024.
 - CC > 1: Maximum z-dimension for blocks and grids is 64.
- These limits are important for programmers to consider when defining their kernel launch configurations (<<<gridDim, blockDim>>>). Exceeding these limits will result in an error. The exact limits for newer architectures can be found in the CUDA C++ Programming Guide.

Nvidia Microarchitectures (Device Architectures)

Nvidia also assigns codenames to its GPU microarchitectures, which correspond to different compute capabilities.

Examples (from the provided text and general knowledge):

- Early Architectures (not explicitly listed in text but implied by CC 1,2,3): Tesla, Fermi, Kepler
- **Kepler:** CC 3.x
- **Maxwell:** CC 5.x
- **Pascal:** CC 6.x
- **Volta:** CC 7.0
- **Turing:** CC 7.5
- **Ampere:** CC 8.x
- **Hopper:** CC 9.0 (very recent as of current knowledge, not in the provided text's dated list but relevant in 2025)
- **Ada Lovelace:** CC 8.9 (also very recent)
- **Blackwell:** CC 10.x, 12.x (even more recent)

Confusing Naming: The text highlights a point of confusion: Nvidia uses "Tesla" as both an early architecture name and a product family name for GPGPU-focused products (e.g., Tesla V100, Tesla P100, etc.).

Product Families: Nvidia's product families can range from discrete graphics cards (e.g., GeForce) to "system on a chip" (SoC) solutions that integrate many hardware components, like those found in mobile phones or embedded systems.

CUDA API Versions vs. Compute Capabilities

- It's important to note that the **version of the CUDA API (e.g., CUDA 11.0, CUDA 12.0)** does **not directly correspond** to the Compute Capability of a GPU.
- A specific CUDA Toolkit version will *support* a range of compute capabilities. For example, CUDA 12.x might support GPUs from CC 5.x up to CC 9.0 (Hopper) and newer.
- Newer CUDA Toolkits generally add support for newer compute capabilities, but they also often drop support for very old ones. This is why it's crucial to check the compatibility matrix (e.g., on Nvidia's developer website) when choosing a CUDA Toolkit version for your specific GPU.

Vector Addition

This is the CUDA program for **Vector Addition** (Program 6.3), which is a fundamental example of data-parallel programming on GPUs.

```
__global__ void Vec_add(  
    const float x[] /* in */,  
    const float y[] /* in */,  
    float      z[] /* out */,  
    const int  n   /* in */) {  
    int my_elt = blockDim.x * blockIdx.x + threadIdx.x;  
  
    /* total threads = blk_ct*th_per_blk may be > n */  
    if (my_elt < n)  
        z[my_elt] = x[my_elt] + y[my_elt];  
} /* Vec_add */  
  
int main(int argc, char* argv[]) {  
    int n, th_per_blk, blk_ct;  
    char i_g; /* Are x and y user input or random? */  
    float *x, *y, *z, *cz;  
    double diff_norm;  
  
    /* Get the command line arguments, and set up vectors */  
    Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);  
    Allocate_vectors(&x, &y, &z, &cz, n);  
    Init_vectors(x, y, n, i_g);  
  
    /* Invoke kernel and wait for it to complete */  
    Vec_add <<<blk_ct, th_per_blk>>>(x, y, z, n);  
    cudaDeviceSynchronize();  
  
    /* Check for correctness */  
    Serial_vec_add(x, y, cz, n);  
    diff_norm = Two_norm_diff(z, cz, n);  
    printf("Two-norm of difference between host and ");  
    printf("device = %e\n", diff_norm);  
  
    /* Free storage and quit */  
    Free_vectors(x, y, z, cz);  
    return 0;  
} /* main */
```

Program 6.3: Kernel and main function of a CUDA program that adds two vectors.

Here's a concise summary of its key components and how it works:

Problem: Vector Addition ($z[i]=x[i]+y[i]$)

- **Embarrassingly Parallel:** Each element's calculation is independent, making it perfectly suited for parallel execution on a GPU.
- **Data Type:** Uses float (32-bit floating-point) as GPUs often have more 32-bit floating-point units.

The CUDA Kernel (`__global__ void Vec_add(...)`)

- **Purpose:** This function runs on the GPU (device) and performs the actual element-wise addition.
- **`__global__`:** A CUDA keyword indicating that this function is a kernel, callable from the CPU (host) and executed by many threads on the GPU.
- **Thread Indexing (`my_elt`):**
 - `int my_elt = blockDim.x * blockIdx.x + threadIdx.x;`
 - This formula calculates a unique, global 0-indexed ID for each thread across the entire grid of threads launched.
 - `blockDim.x`: Number of threads per block.
 - `blockIdx.x`: Index of the current block.
 - `threadIdx.x`: Index of the current thread within its block.
 - This global ID (`my_elt`) directly corresponds to the array index `i` for which the current thread is responsible.
- **Boundary Check:**
 - `if (my_elt < n)`: Essential because the total number of threads launched might be greater than the vector size `n`. This ensures that only valid threads perform computations, preventing out-of-bounds memory access.
- **Computation:** `z[my_elt] = x[my_elt] + y[my_elt];` Each thread computes one element of the result.

The Host Code (main function)

Runs on CPU: Manages the overall program flow, including data setup, kernel launch, and result verification.

Argument Parsing & Setup: Calls `Get_args` to read `n` (vector size), `blk_ct` (number of blocks), and `th_per_blk` (threads per block) from command-line arguments.

Memory Allocation (`Allocate_vectors`):

- For `x`, `y`, and `z`, it uses `cudaMallocManaged()`. This is CUDA's **Unified Memory** feature, which allows the same pointers to be accessed by both the CPU and GPU, simplifying memory management by automatically handling data transfers.

- For cz (a reference array for correctness check), it uses standard malloc().

Initialization: Init_vectors initializes the input arrays x and y on the host.

Kernel Launch: Vec_add <<>>(x, y, z, n);

- This is where the GPU computation begins. blk_ct blocks, each with th_per_blk threads, are launched.

Synchronization (cudaDeviceSynchronize()):

- Since kernel launches are asynchronous (CPU can proceed while GPU works), this call blocks the CPU until the Vec_add kernel has finished executing on the GPU. This ensures z contains the final results before the CPU attempts to read it.

Correctness Check:

- Serial_vec_add(x, y, cz, n): Computes the vector sum on the CPU for comparison.
- Two_norm_diff(z, cz, n): Calculates the difference between the GPU's result (z) and the CPU's reference (cz) to verify accuracy.

Memory Deallocation (Free_vectors):

- Uses cudaFree() for memory allocated with cudaMallocManaged().
- Uses free() for memory allocated with malloc().

This program serves as a foundational example for understanding how to leverage the parallel processing power of GPUs for simple, data-intensive tasks using CUDA's thread hierarchy and unified memory model.

Returning results from CUDA kernels

Why Direct Pass-by-Reference Fails

Problem: In standard C, you can pass a pointer to a variable (&sum) to a function, and the function can modify the variable through that pointer. This doesn't work directly with CUDA kernels.

Reason: "Addresses on the host are, in most systems, invalid on the device, and vice versa." The CPU's memory space and the GPU's memory space are typically distinct. If you pass a CPU memory address (&sum) to a kernel, the GPU will interpret that address within its own memory space, likely leading to:

- **Accessing invalid memory:** The address might point to a region the GPU cannot access, causing a crash or "hang."
- **Corrupting unrelated data:** The address might coincidentally point to valid GPU memory that's used for something else, leading to silent data corruption.

Example Provided: The first Add kernel and main function demonstrate this failure. The `&sum` address passed from the host will be meaningless or incorrect on the device.

Correct Approaches to "Returning" Results

CUDA kernels cannot return values like regular C functions (they have void return type). Instead, results are returned by writing them to memory locations that the host can eventually access.

1. Using Pointers with Unified Memory (`cudaMallocManaged`)

This is the most straightforward modern approach:

- **Memory Allocation:** Instead of declaring `sum` as a local `int` on the host, declare an `int* sum_p`; and allocate memory for a single integer using `cudaMallocManaged(&sum_p, sizeof(int));`.
- **Host/Device Access:** `cudaMallocManaged` allocates memory that is accessible by both the host and the device. CUDA's Unified Memory system automatically handles migrating the data between physical host and device memories as needed.
- **Kernel Operation:** The kernel (`Add <<<1, 1>>>(2, 3, sum_p);`) writes the result to `*sum_p`.
- **Synchronization:** `cudaDeviceSynchronize();` is **essential** after the kernel launch. This ensures that the GPU has completed writing the result to `*sum_p` and that the data has been migrated back to host memory (if necessary) before the `printf` statement on the host tries to read it.
- **Cleanup:** `cudaFree(sum_p);` is used to deallocate the managed memory.
- **Benefit:** Simplifies code by abstracting explicit data transfers.

2. Using Pointers with Explicit Memory Transfers (`cudaMalloc` and `cudaMemcpy`)

This approach is necessary for older GPUs (Compute Capability < 3.0) or when a programmer wants explicit control over data movement for performance:

Separate Memory Allocation:

- Host-side pointer: `int *hsum_p`; allocated with `malloc(sizeof(int));`
- Device-side pointer: `int *dsum_p`; allocated with `cudaMalloc(&dsum_p, sizeof(int));`

No Initial Data Transfer (for output): Unlike input data, there's no need to copy an initial value for `sum_p` from host to device before the kernel runs, as the kernel will compute it.

Kernel Operation: The kernel (`Add <<<1, 1>>>(2, 3, dsum_p);`) writes the result to `*dsum_p` (the device memory location).

Result Transfer: After the kernel launch, `cudaMemcpy(hsum_p, dsum_p, sizeof(int), cudaMemcpyDeviceToHost);` is used to explicitly copy the computed result from `dsum_p` (device memory) to `hsum_p` (host memory).

Implicit Synchronization: `cudaMemcpy` is a synchronous operation. It blocks the host thread until the data transfer (and implicitly, any preceding kernel that generated the data) is complete. Therefore, `cudaDeviceSynchronize()` is **not strictly necessary** before this specific `cudaMemcpy` call, as the copy operation itself ensures kernel completion.

Cleanup: `free(hsum_p);` and `cudaFree(dsum_p);` are used for deallocation.

Benefit: Gives the programmer fine-grained control over data movement, which can be critical for performance optimization.

3. Using a Global Managed Variable (`__managed__`)

This is another option available with Unified Memory, but it uses a global variable:

Declaration: `__managed__ int sum;`

- The `__managed__` qualifier declares `sum` as a variable that resides in Unified Memory, accessible from both host and device. It implies global scope.

Kernel Operation: The kernel (`__global__ void Add(int x, int y) { sum = x + y; }`) directly writes to the global `sum` variable.

Synchronization: `cudaDeviceSynchronize();` is **required** before the host attempts to read `sum`. Similar to `cudaMallocManaged`, the system manages data migration, and the host needs to wait for the GPU to finish writing and for the data to be coherent in host memory.

Restrictions:

- Requires Compute Capability ≥ 3.0 .

- On devices with Compute Capability < 6.0, sum cannot be accessed simultaneously by both the device (during kernel execution) and the host. The host must wait for kernel completion.

Disadvantage: Using global variables can reduce modularity and make code harder to maintain and debug compared to passing pointers as arguments.

CUDA trapezoidal rule I

The Trapezoidal Rule (Mathematical Background)

Purpose: To estimate the definite integral of a function $f(x)$ over an interval $[a,b]$.

Method: Divides the interval $[a,b]$ into n equal subintervals, each of width $h=(b-a)/n$. For each subinterval $[x_i, x_{i+1}]$, it approximates the area under the curve as the area of a trapezoid with height h and parallel sides $f(x_i)$ and $f(x_{i+1})$.

Formula for one trapezoid: $\text{Area}_i = \frac{h}{2} [f(x_i) + f(x_{i+1})]$

Total Area Approximation: Summing the areas of all trapezoids:

$$\text{Area} \approx \frac{h}{2} [f(x_0) + f(x_1)] + \frac{h}{2} [f(x_1) + f(x_2)] + \dots + \frac{h}{2} [f(x_{n-1}) + f(x_n)]$$

Simplified Formula: This can be rewritten as: $\text{Area} \approx h \left[\frac{1}{2} (f(a) + f(b)) + \sum_{i=1}^{n-1} f(x_i) \right]$

Serial Implementation (Program 6.11):

- Calculates h and initializes `trap` with $0.5 \times (f(a) + f(b))$.
- Uses a for loop from $i = 1$ to $n-1$ to sum the intermediate $f(x_i)$ values into `trap`.
- Finally, multiplies the total `trap` by h .

CUDA Implementation (Initial Thoughts & Challenges)

Goal: Parallelize the for loop, where the majority of the work lies. Each thread could calculate one $f(x_i)$ and add it to a running sum.

Initial Idea (simplified):

```
int my_i = blockDim.x * blockIdx.x + threadIdx.x;
float my_x = a + my_i * h;
float my_trap = f(my_x);
float trap += my_trap; // This is problematic!
```

Identified Problems:

1. **Initialization of h and trap:** These are kernel arguments. If they were local variables, each thread would have its own, preventing shared modification/initialization.
2. **my_i range:** The serial loop covers $i=1, \dots, n-1$. my_i ranges from 0 to total_threads - 1, so a boundary check is needed. Threads with my_i=0 or my_i $\geq n$ should not contribute to the sum of $f(x_i)$ terms.
3. **Race Condition on trap:** If multiple threads simultaneously try to update trap (a shared variable), their updates can overwrite each other, leading to an incorrect final result. This is a classic **race condition**.
4. **Kernel Return Type:** Kernels must have void return types. trap needs to be returned to the host.
5. **Final Update:** The final multiplication by h needs to happen *after* all threads have finished their additions.

Initialization, Return Value, and Final Update (Solutions)

Formal Arguments are Private: CUDA threads, like Pthreads/OpenMP threads, have their own stacks. Kernel arguments are allocated on the thread's stack, meaning each thread gets its own private copy. Changes made by one thread to its h or trap argument are not visible to other threads.

Solution for h: h can be calculated on the host once and passed as an argument to the kernel. Each thread receives a copy of this constant value.

Solution for trap (Shared Variable):

- Allocate trap in global memory on the device (or using Unified Memory) **before** the kernel launch. This memory location will be shared among all threads.
- Pass a pointer to this shared memory location (float* trap_p) to the kernel. Each thread will get a copy of trap_p, but all copies will point to the *same* shared memory.
- The initial term ($0.5 * (f(a) + f(b))$) is computed on the **host** and stored in *trap_p before the kernel launch. This addresses initialization.
- The final multiplication by h is also done on the **host** after the kernel completes execution (and results are synchronized). This addresses the final update.

Wrapper Function (Trap_wrapper - Program 6.12):

```

__global__ void Dev_trap(
    const float a      /* in      */,
    const float b      /* in      */,
    const float h      /* in      */,
    const int  n       /* in      */,
    float*     trap_p  /* in/out */) {
    int my_i = blockDim.x * blockIdx.x + threadIdx.x;

    /* f(x_0) and f(x_n) were computed on the host. So */
    /* compute f(x_1), f(x_2), ..., f(x_(n-1))          */
    if (0 < my_i && my_i < n) {
        float my_x = a + my_i*h;
        float my_trap = f(my_x);
        atomicAdd(trap_p, my_trap);
    }
} /* Dev_trap */

/* Host code */
void Trap_wrapper(
    const float a      /* in      */,
    const float b      /* in      */,
    const int  n       /* in      */,
    float*     trap_p  /* out     */,
    const int  blk_ct  /* in      */,
    const int  th_per_blk /* in      */) {

    /* trap_p storage allocated in main with
     * cudaMallocManaged */
    *trap_p = 0.5*(f(a) + f(b));
    float h = (b-a)/n;

    Dev_trap<<<blk_ct, th_per_blk>>>(a, b, h, n, trap_p);
    cudaDeviceSynchronize();

    *trap_p = h>(*trap_p);
} /* Trap_wrapper */

```

Program 6.12: CUDA kernel and wrapper implementing trapezoidal rule.

- A common practice is to encapsulate the host-side setup, kernel launch, and post-kernel cleanup/finalization into a "wrapper" function.
- Trap_wrapper handles:
 - Initial calculation of $0.5 * (f(a) + f(b))$ on `*trap_p`.
 - Calculation of `h`.
 - Launching the `Dev_trap` kernel.
 - `cudaDeviceSynchronize()`: Essential to wait for kernel completion.
 - Final multiplication `*trap_p = h * (*trap_p)`.
- This also implicitly solves the problem of "returning a value" from the kernel, as `trap_p` points to memory accessible by the host after synchronization.

Using the Correct Threads (if $(0 < \text{my_i} \ \&\& \ \text{my_i} < n)$)

Problem: The serial loop sums $f(x_i)$ for $i=1, \dots, n-1$.

Solution: The kernel (Dev_trap) includes the condition if $(0 < \text{my_i} \ \&\& \ \text{my_i} < n)$ (Line 11).

- This ensures that only threads responsible for indices 1 through $n-1$ (which correspond to the intermediate $f(x_i)$ terms) perform the main computation and contribute to the sum.
- Threads with $\text{my_i} = 0$ (for $f(x_0)=f(a)$) or $\text{my_i} \geq n$ (for $f(x_n)=f(b)$ or excess threads) skip the addition part. The terms $f(a)$ and $f(b)$ are handled by the host-side initialization.

Updating the Return Value and atomicAdd

Race Condition: The core problem is that multiple threads concurrently try to add their my_trap value to the shared *trap_p . A simple $\text{*trap_p} += \text{my_trap}$; is not safe because addition is not an atomic (indivisible) operation at the hardware level. Threads could read the old value, then one thread's write could be overwritten by another.

Solution: Atomic Operations: CUDA provides **atomic functions** to guarantee that an operation on a shared memory location appears as a single, indivisible instruction to all threads.

atomicAdd(trap_p, my_trap); (Line 14):

- This specific atomic function is used for floating-point addition.
- It ensures that when multiple threads call atomicAdd on the same trap_p address, their additions are serialized, preventing race conditions and ensuring the final sum is correct.
- `__device__`: atomicAdd can only be called from device code (kernels).
- Returns the value of *float_p *before* the addition.

Performance Analysis

Timing: The run-time of the CUDA trapezoidal rule is measured by timing the `Trap_wrapper` function on the host, as it encompasses all relevant computations (host setup, kernel execution, host finalization).

Results (Table 6.5, $n=220$ trapezoids, $f(x)=x^2+1$):

- **ARM Cortex-A15 (CPU):** 33.6 ms
- **Nvidia GK20A (GPU, 1 SM, 192 SPs):** 20.7 ms (Faster than ARM CPU, but still relatively slow compared to other options)

- **Intel Core i7 (CPU, 1 core):** 4.48 ms (Significantly faster than GK20A, indicating single-core CPU performance can still be strong for some tasks)
- **Nvidia GeForce GTX Titan X (GPU, 24 SMs, 3072 SPs):** 3.08 ms (Fastest, 45% faster than Intel i7 single core).

Table 6.5 Mean run-times for serial and CUDA trapezoidal rule (times are in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Run-time	33.6	20.7	4.48	3.08

Observation & Question:

While the Titan X is the fastest, the text notes that with 3072 SPs, one might expect even greater performance compared to a single Core i7 core.

This suggests there might be bottlenecks or inefficiencies in this particular CUDA implementation that prevent it from fully utilizing the GPU's potential, despite using atomics for correctness.

This often points to issues like global memory access patterns or the cost of atomic operations themselves.

CUDA trapezoidal rule II: improving performance

Tree-Structured Communication (Improving Global Sums)

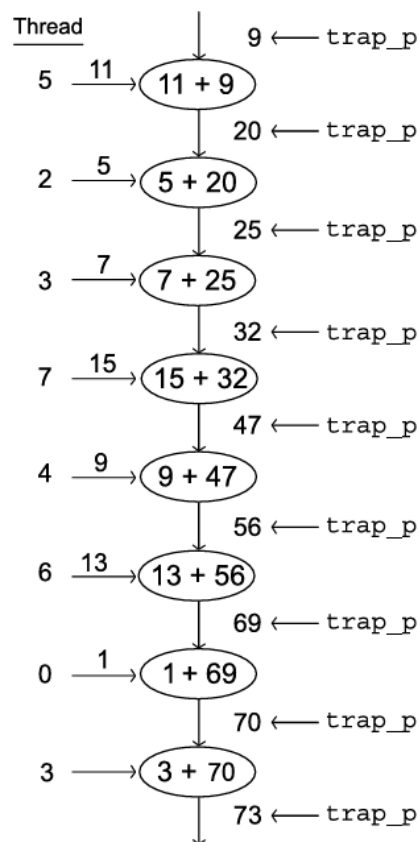
Problem with atomicAdd: While atomicAdd ensures correctness by preventing race conditions, it inherently *serializes* the additions to the shared `*trap_p` variable. Only one thread can update it at a time. For N threads, this can lead to N sequential additions, significantly limiting parallelism, especially for large N .

Visualizing the Serialization (Referring to Figure 6.3: Basic sum):

- Imagine a series of boxes, each representing a thread ($t_0, t_1, \dots, t_7$ for 8 threads).
- All these threads have a value (`my_trap`) they want to add to a single global sum location (`*trap_p`).
- **Figure 6.3** would show arrows from each thread pointing to a *single central box* representing `*trap_p`. The arrows would likely be stacked sequentially or indicate a funnel, emphasizing that even though many threads want to update, they must do so one at a time.
- Table 6.6 (provided in the text) shows a hypothetical sequence: thread 5 updates, then thread 2, then thread 3, etc. This sequential nature is the bottleneck.

Table 6.6 Basic global sum with eight threads.

Time	Thread	my_trap	*trap_p
Start	—	—	9
t_0	5	11	20
t_1	2	5	25
t_2	3	7	32
t_3	7	15	47
t_4	4	9	56
t_5	6	13	69
t_6	0	1	70
t_7	1	3	73

**FIGURE 6.3**

Basic sum.

Solution: Tree-Structured Sum (Reduction Tree): Instead of all threads adding directly to a single global sum, threads are paired up in stages. In each stage, half of the "active" threads add their partial sum to their partner's partial sum. This process continues until only one thread holds the final sum.

Visualizing the Tree Structure (Referring to Figure 6.4: Tree-structured sum):

- Imagine the threads arranged in a line at the bottom ($t_0, t_1, \dots, t_7$). Each thread has its `my_trap` value.

- **Stage 1 (e.g., diff = 4 or offset = 1 for pairs):**
 - Threads are paired up (e.g., t0 and t1, t2 and t3, etc.).
 - In **Figure 6.4**, this would be shown by t1 adding its value to t0, t3 adding to t2, t5 adding to t4, and t7 adding to t6. Arrows would go from the "partner" thread to the "receiving" thread. Now, t0, t2, t4, t6 hold partial sums.
- **Stage 2 (e.g., diff = 2 or offset = 2 for pairs of pairs):**
 - The remaining active threads are again paired (e.g., t2 adds its sum to t0, t6 adds to t4).
 - **Figure 6.4** would show arrows from t2 to t0, and t6 to t4. Now, t0 and t4 hold larger partial sums.
- **Stage 3 (e.g., diff = 1 or offset = 4 for final pair):**
 - The last two active threads are paired (t4 adds to t0).
 - **Figure 6.4** shows an arrow from t4 to t0.
- **Result:** Finally, only t0 holds the total sum. The diagram resembles an inverted binary tree, where each node above the base layer represents a sum of its children's values.
 - **Performance Improvement:** This drastically reduces the number of sequential additions from T to $\log_2(T)+1$.

CUDA Implementations:

- **Shared Memory:** Best for devices with compute capability < 3 .
- **Warp Shuffles:** Available for devices with compute capability ≥ 3 . These are generally preferred for their speed.

Local Variables, Registers, Shared, and Global Memory

This section provides a crucial digression into the CUDA memory hierarchy, essential for understanding performance optimization.

Memory Hierarchy (Fastest to Slowest, Smallest to Largest): This hierarchy (registers > shared memory > global memory) is depicted in Table 6.7 in the text, showing typical sizes.

1. Registers:

- **Fastest:** ~1 cycle access.
- **Smallest:** Very limited per thread/SM.
- **Usage:** Used for local variables if sufficient space is available.

2. Shared Memory:

- **Faster:** ~1 order of magnitude slower than registers (e.g., 10 cycles).
- **Smaller:** Relatively small, but larger than registers (e.g., tens of KB per SM).
- **Scope:** Accessible by all threads *within the same thread block*. Private to each SM.

3. Global Memory:

- **Slowest:** 2-3 orders of magnitude slower than registers (e.g., hundreds of cycles).
- **Largest:** Device's main memory (GBs).
- **Scope:** Accessible by all threads across the entire GPU (and by the host if unified memory is used).

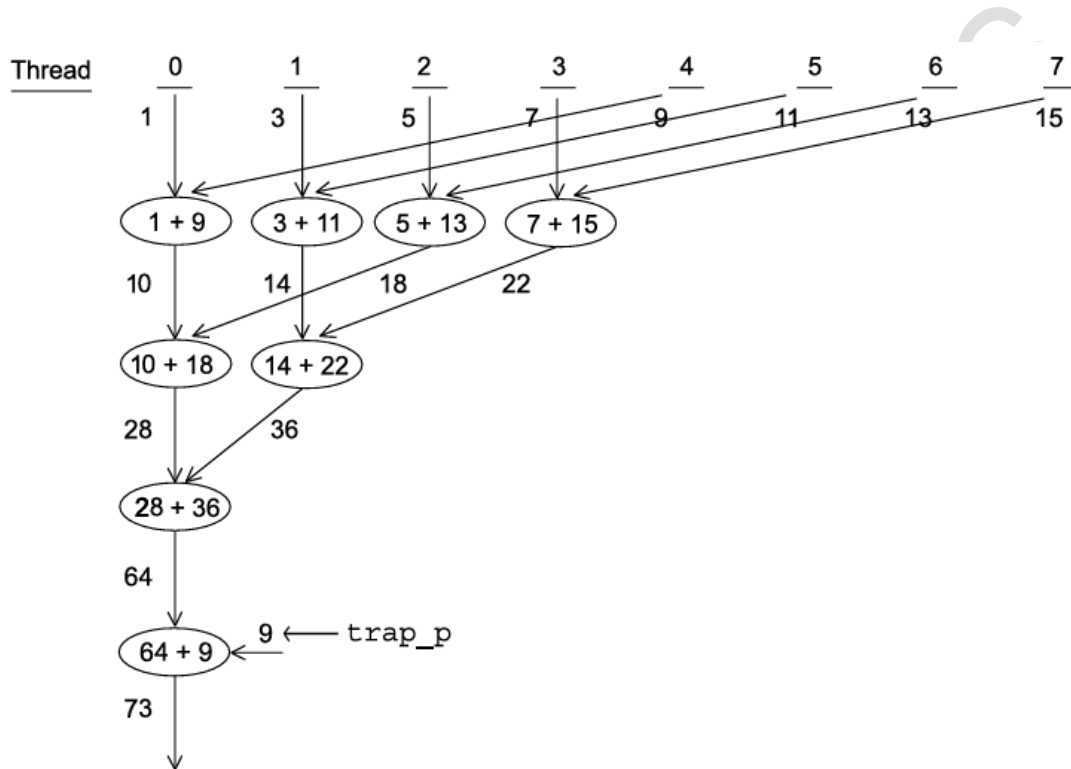


FIGURE 6.4

Tree-structured sum.

Table 6.7 Memory statistics for some Nvidia GPUs.

GPU	Compute Capability	Registers: Bytes per Thread	Shared Mem: Bytes per Block	Global Mem: Bytes per GPU
Quadro 600	2.1	504	48K	1G
GK20A (Jetson TK1)	3.2	504	48K	2G
GeForce GTX Titan X	5.2	504	48K	12G

Local Variables:

- Ideally stored in **registers** for fastest access.
- If registers are exhausted, they "spill" to a thread-private region of **global memory** (much slower).

Performance Implication: Maximizing the use of faster memory (registers, then shared memory) and minimizing global memory access is a primary optimization strategy in CUDA.

Warps and Warp Shuffles

Warp: A fundamental scheduling unit on an Nvidia GPU. It's a group of 32 threads with consecutive ranks within a thread block.

- warpSize variable holds this value (currently 32).
- Threads within a warp execute in **SIMD (Single Instruction, Multiple Data) fashion**. They execute the same instruction at the same time.
- **Divergence:** If threads within a warp take different branches (e.g., in an if-else), they "diverge." The hardware serializes their execution, running each branch and masking out the threads that aren't taking that path. This can be a significant performance penalty.
- **Convergence:** When divergent threads come back to executing the same instruction.
- **Lane:** A thread's rank within its warp (lane = threadIdx.x % warpSize).

Warp Shuffle Functions: Introduced in CUDA Compute Capability ≥ 3.0 . These functions allow threads within a warp to directly read values from the **registers** of other threads in the same warp. This is extremely fast because it bypasses shared or global memory.

__shfl_down_sync: The specific shuffle function discussed for tree reduction.

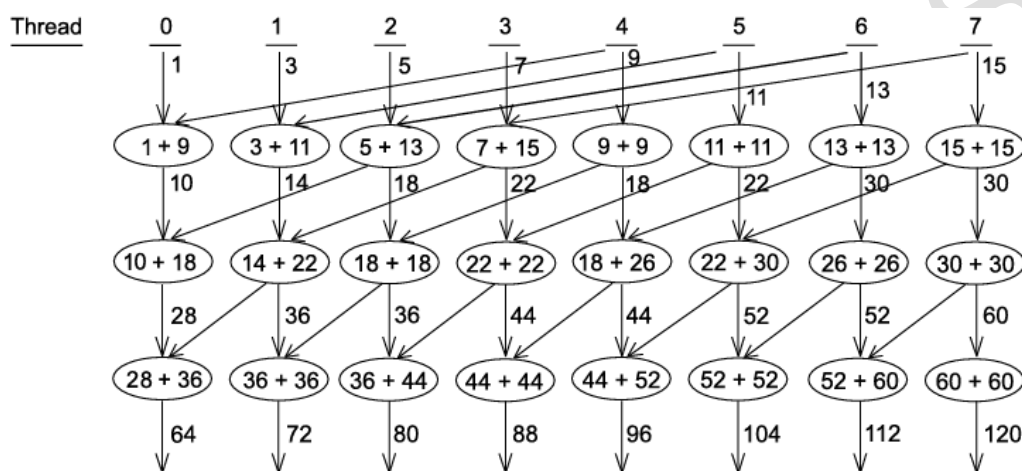
- `__device__ float __shfl_down_sync(unsigned mask, float var, unsigned diff, int width)`
- mask: A bitmask indicating participating threads (usually 0xffffffff for all threads in the warp). Ensures synchronization (`_sync`).
- var: The variable in the calling thread's register whose partner's value will be read.
- diff: Specifies which thread's value to read. If called by thread `l`, it reads from thread `l + diff`.
- **"Shuffle Down":** Reads from a higher-ranked thread.
- **Behavior when `l + diff >= warpSize`:** The call returns the *caller's own* var value. This is crucial for handling partial warps or threads at the end of a warp.
- **Returns:** The value of var from the source thread (`l + diff`).

Warp_sum Function (Program 6.13)

This function demonstrates a tree-structured reduction within a single warp using `__shfl_down_sync`.

Logic:

- It iterates with diff starting from warpSize/2 and halving in each step (diff = diff/2).
- In each iteration, `var += __shfl_down_sync(mask, var, diff);` adds the value from a partner thread (`l + diff`) to the current thread's var.
- **Result:** After the loop, only `threadIdx.x == 0` (lane 0) holds the correct sum of all values in the warp. Other threads will hold partial sums or incorrect values.

Visualizing Warp_sum (Referring to Figure 6.5: Tree-structured sum using warp shuffle):**FIGURE 6.5**

Tree-structured sum using warp shuffle.

This diagram specifically illustrates the `__shfl_down_sync` process for a warpSize of 8 (though warpSize is typically 32, this smaller example is used for clarity).

Input Layer: Shows 8 ovals, labeled with thread lane IDs (0 to 7) and their initial var values (e.g., `var_0`, `var_1`, ..., `var_7`).

First Iteration (diff = 4):

- For thread `l=0`, it attempts to read from `l+diff = 4`. So `var_0` receives `var_4`.
- For `l=1`, it receives `var_5`.
- For `l=2`, it receives `var_6`.
- For `l=3`, it receives `var_7`.
- Crucially, for `l=4, 5, 6, 7`, since `l+diff >= warpSize` (i.e., `4+4 >= 8`, `5+4 >= 8`, etc.), `__shfl_down_sync` returns *their own* current var value. The diagram would show this by an arrow looping back onto the same oval, causing var to effectively double.

- The diagram would show ovals for $l=0,1,2,3$ receiving values from higher lanes, while ovals for $l=4,5,6,7$ would show values being added to themselves.

Subsequent Iterations (diff = 2, diff = 1): The process repeats, with fewer threads actively receiving values from others as diff decreases. Threads for whom $l+diff$ is out of bounds (within the current width or warpSize) continue to "add to themselves."

Final Result: The diagram clearly shows that only the oval for lane 0 contains the sum of all initial values.

Shared Memory and an Alternative to Warp Shuffle (Shared_mem_sum)

For CC < 3.0: Warp shuffles are not available. Shared memory is the next best option for fast intra-block communication.

__device__ float Shared_mem_sum(float shared_vals[]):

- **__shared__ Memory:** For optimal performance, the shared_vals array should be declared as `__shared__` in the kernel (e.g., `__shared__ float shared_vals[32];`). This places it in the fast shared memory of the SM.
- **Dissemination Sum/Reduction:** This implementation is a variant of a tree reduction where all threads read from a partner and update their own sum.
- **Synchronization-free within Warp:** Since threads in a warp operate in lockstep (SIMD), no explicit `__syncthreads()` is needed *within* this function if it's strictly operating on a single warp. All threads read their `shared_vals[source]` before any thread writes to its `shared_vals[my_lane]`.
- **Result:** Unlike `Warp_sum`, **every thread** in the warp will hold the correct total sum after this function completes. This is useful for applications where all threads need the final result.

Visualizing Shared_mem_sum (Referring to Figure 6.6: Dissemination sum using shared memory):

- Similar to Fig 6.5, but emphasizes a "dissemination" or "butterfly" pattern.
- **Input Layer:** Ovals representing threads (e.g., 0 to 7) with their initial values in a shared array `shared_vals`.
- **Iteration (diff = warpSize/2, then halving):**
 - Each thread `my_lane` adds `shared_vals[source]` to `shared_vals[my_lane]`. The source is typically $(my_lane + diff) \% warpSize$ or similar, ensuring a partner.
 - **Figure 6.6** would show a criss-crossing pattern of arrows. For $diff=4$, thread 0 adds from thread 4, thread 1 from thread 5, etc.

- The key visual difference from `__shfl_down_sync` is that *all* threads are actively adding values from different partners in each pass, and after the process, all threads have the correct final sum. The diagram would illustrate how values from all initial positions spread out and accumulate in every thread's `shared_vals[my_lane]` location.

Dynamic Shared Memory:

- `extern __shared__ float shared_vals[];` If the shared memory size isn't known at compile time.
- Kernel launch syntax: `Dev_trap <<>>(..., th_per_blk * sizeof(float))` (third argument in triple angle brackets specifies dynamic shared memory size).

CUDA trapezoidal rule III: blocks with more than one warp

`__syncthreads()` and Inter-Warp Synchronization

Problem with Single-Warp Blocks: Limiting thread blocks to only 32 threads (one warp) underutilizes modern GPUs, which can support blocks with up to 1024 threads (32 warps). Larger blocks allow for more threads to cooperate and synchronize efficiently.

Strategy for Larger Blocks:

1. Each thread computes its contribution (`f(xi)`).
2. Each warp performs a local sum of its threads' contributions (using `Warp_sum` or `Shared_mem_sum` from the previous section). This results in 32 partial warp sums if there are 32 warps.
3. A dedicated warp (e.g., Warp 0, threads 0-31) collects and sums these 32 warp sums to get the total block sum.

The Race Condition: When Warp 0 tries to sum the warp results, it needs to ensure all other warps have finished their individual warp sums. Since warps operate independently, Warp 0 might try to read a warp sum before the contributing warp has finished writing it. This is a race condition.

Solution: `__syncthreads()`:

- `__device__ void __syncthreads(void);`
- This is a **block-level barrier synchronization**. It causes all threads *within the calling thread block* to pause at that point until every thread in that *same block* has reached and executed the `__syncthreads()` call.
- Once all threads have arrived, they are released to continue execution.

- **Corrected Pseudocode:**
- Each thread computes its contribution;
- Each warp adds its threads' contributions;
- `__syncthreads();` // Ensures all warp sums are complete
- Warp 0 in block adds warp sums;

Important Caveats of `__syncthreads()`:

1. **All Threads Must Call It:** If even one thread in a block skips the `__syncthreads()` call (e.g., due to an if condition), the threads that *do* call it will wait forever, leading to a deadlock.
2. **Block-Local Only:** `__syncthreads()` *only* synchronizes threads within the *same* thread block. Threads in different blocks operate independently and cannot be synchronized using `__syncthreads()`. For synchronization across blocks, `cudaDeviceSynchronize()` on the host is typically used, or more advanced techniques like cooperative groups (CUDA 9+) for grid-wide synchronization.

More Shared Memory

Problem: After `__syncthreads()`, how does Warp 0 access the sums from other warps? Warp shuffles only work *within* a warp. Direct register access across warps is not possible.

Solution: Shared Memory. Shared memory is accessible by all threads within the *same thread block*.

Implementation Strategy:

1. Declare a shared array: `__shared__ float warp_sum_arr[WARPSZ];` (WARPSZ being 32, the number of warps a block can have if it's 1024 threads).
2. After each warp computes its sum (`my_result = Warp_sum(my_trap)`), the thread with `lane == 0` within that warp writes its `my_result` to a specific index in `warp_sum_arr` corresponding to its warp ID (`my_warp`).

```
// ... (thread calculation and warp sum) ...  
float my_result = Warp_sum(my_trap); // Or Shared_mem_sum  
if (my_lane == 0) warp_sum_arr[my_warp] = my_result;  
__syncthreads(); // Synchronize all warps  
// Now, warp_sum_arr contains the sum from each warp, accessible by all threads in the block.  
// Warp 0 can then sum these values.
```

Shared Memory Warp Sums (Detailed Implementation)

This section discusses the finer points of managing shared memory for reductions in larger blocks, especially when *all* threads' initial contributions are also stored in shared memory.

Large Shared Array for Thread Contributions:

- `#define MAX_BLK SZ 1024`
- `__shared__ float thread_calcs[MAX_BLK SZ];`
- Each warp gets a subarray: `float* shared_vals = thread_calcs + my_warp * warpSize;`
- Each thread stores its `f(x_i)` result in `shared_vals[my_lane]`.
- Then, `Shared_mem_sum(shared_vals)` is called by each warp to get its `my_result`.

Avoiding Race Conditions on thread_calcs for Warp Sums:

- **Problem:** If other warps try to store their `my_result` directly back into `thread_calcs[my_warp]` *before* Warp 0 has finished using the original `thread_calcs` data to calculate its own sum.
- **Solution (with two `__syncthreads()` calls):**

```
float my_result = Shared_mem_sum(shared_vals); // Each warp computes its sum
__syncthreads(); // 1st sync: all warps finish their sums

if (my_lane == 0) thread_calcs[my_warp] = my_result; // Warp 0 stores its sum in
thread_calcs[0], warp 1 in thread_calcs[1], etc.
__syncthreads(); // 2nd sync: all warps have written their sums to thread_calcs

// Now it's safe for Warp 0 to collect these sums
if (my_warp == 0) {
    // Some adjustment for partially filled warp_sum_arr if blockDim.x is not a multiple of
    warpSize
    blk_result = Shared_mem_sum(thread_calcs); // Warp 0 sums the warp results
}
```

- **Cost of Multiple `__syncthreads()`:** Each `__syncthreads()` call incurs overhead as it forces all threads to wait. This can be costly, especially if the number of threads in the block is very large and not all threads can execute concurrently (some will be delayed reaching the barrier).

Alternative for storing Warp Sums (Preferred):

- Instead of overwriting `thread_calcs`, use a *separate* shared array specifically for warp sums, as discussed in 6.13.2: `__shared__ float warp_sum_arr[WARPSZ];`. This avoids the need for a second `__syncthreads()` for the data transfer, only needing one before Warp 0 starts its summation.
- `if (my_lane == 0) warp_sum_arr[my_warp] = my_result;` (Safe update without race, as each `my_warp` has a unique index).
- *Why is if (my_lane == 0) shared_vals[0] = my_result; (mentioned in the text) safe?* Because each warp is operating on its own `shared_vals` subarray. So, `shared_vals[0]` for `my_warp=0` is `thread_calcs[0]`, `shared_vals[0]` for `my_warp=1` is `thread_calcs[32]`, etc.
- These are distinct memory locations, so there's no race condition here. However, the `warp_sum_arr` approach is cleaner for the final sum.

Shared Memory Banks

Shared Memory Banking: Nvidia GPUs divide shared memory into 32 (or 16 for older cards) independent "banks."

Concurrent Access:

- If 32 threads in a warp simultaneously access 32 *different* banks, the access is fully parallel (no serialization).
- **Table 6.9 (Shared memory banks):** This table illustrates how `thread_calcs` elements map to banks. Columns are banks (0-31), rows are consecutive elements. E.g., `thread_calcs[0]`, `thread_calcs[32]`, `thread_calcs[64]` all fall into **Bank 0**.

Table 6.9 Shared memory banks: Columns are memory banks. The entries in the body of the table show subscripts of elements of `thread_calcs`.

	Bank					
	0	1	2	...	30	31
Subscripts	0	1	2	...	30	31
	32	33	34	...	62	63
	64	65	66	...	94	95
	96	97	98	...	126	127
	⋮	⋮	⋮	⋱	⋮	⋮
	992	993	994	...	1022	1023

Bank Conflicts:

- If multiple threads (e.g., from the same warp) access different memory locations that reside in the *same bank*, these accesses must be **serialized**. This causes a **bank conflict**, significantly slowing down the operation.
- **Problem Example:** If we stored warp sums in `warp_sum_arr[0]`, `warp_sum_arr[32]`, `warp_sum_arr[64]`, etc. (using indices that are multiples of `warpSize`), all these would map to Bank 0, causing serialization. This would make the final warp 0 summation very slow.

Solution: Store the warp sums in a **contiguous subarray** of shared memory, like `warp_sum_arr[0]`, `warp_sum_arr[1]`, ..., `warp_sum_arr[31]`. This ensures that each `warp_sum_arr[my_warp]` element maps to a *different* bank when accessed by different threads of Warp 0, enabling parallel access.

- `__shared__ float warp_sum_arr[WARPSZ];` (where `WARPSZ` is 32) is designed to allocate 32 contiguous floats. When Warp 0's threads (0-31) access `warp_sum_arr[0]`, `warp_sum_arr[1]`, ..., `warp_sum_arr[31]` respectively, they are accessing different banks, avoiding conflicts.

Finishing Up (Program 6.15)**Final Block Reduction:**

- After the `__syncthreads()` call, Warp 0 (specifically if `(my_warp == 0)`) takes over.
- It handles potential uninitialized elements in `warp_sum_arr` if `blockDim.x` is not a multiple of `warpSize`. Threads in Warp 0 for indices higher than the actual number of active warps set their `warp_sum_arr` element to 0.0.
- Then, Warp 0 performs a *second* `Shared_mem_sum` (or `Warp_sum` if using shuffles) on `warp_sum_arr` to compute the `blk_result` (the total sum for the entire block).
- Finally, **only thread 0 of the entire block** (if `(threadIdx.x == 0)`) uses `atomicAdd(trap_p, blk_result);` to add the block's total sum to the global `trap_p` variable. This reduces the number of `atomicAdd` calls from `N` (total threads) to `num_blocks` (number of blocks), which is a huge optimization.

Program 6.15: Provides the full kernel code for the shared memory version, incorporating all these strategies. The warp shuffle version would be similar, just calling `Warp_sum` instead of `Shared_mem_sum` and potentially not needing `thread_calcs` if input values are handled differently.

```

__global__ void Dev_trap(
    const float a      /* in */,
    const float b      /* in */,
    const float h      /* in */,
    const int  n       /* in */,
    float* trap_p      /* out */) {
    __shared__ float thread_calcs[MAX_BKLSZ];
    __shared__ float warp_sum_arr[WARPSZ];
    int my_i = blockDim.x * blockIdx.x + threadIdx.x;
    int my_warp = threadIdx.x / warpSize;
    int my_lane = threadIdx.x % warpSize;
    float* shared_vals = thread_calcs + my_warp*warpSize;
    float blk_result = 0.0;

    shared_vals[my_lane] = 0.0f;
    if (0 < my_i && my_i < n) {
        float my_x = a + my_i*h;
        shared_vals[my_lane] = f(my_x);
    }

    float my_result = Shared_mem_sum(shared_vals);
    if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
    __syncthreads();

    if (my_warp == 0) {
        if (threadIdx.x >= blockDim.x/warpSize)
            warp_sum_arr[threadIdx.x] = 0.0;
        blk_result = Shared_mem_sum(warp_sum_arr);
    }

    if (threadIdx.x == 0) atomicAdd(trap_p, blk_result);
} /* Dev_trap */

```

Program 6.15: CUDA kernel implementing trapezoidal rule and using shared memory. This version can use large thread blocks.

Performance (Table 6.10)

Table 6.10 Mean run-times for trapezoidal rule using arbitrary block size (times in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Original	33.6	20.7	4.48	3.08
Warp Shuffle, 32 ths/blk		14.4		0.210
Shared Memory, 32 ths/blk		15.0		0.206
Warp Shuffle		12.8		0.141
Shared Memory		14.3		0.150

Significant Improvement: Table 6.10 shows the mean run-times for the optimized CUDA trapezoidal rule using large thread blocks (1024 threads/block).

- **Nvidia GK20A:**
 - Original (basic atomicAdd, 32 ths/blk): 20.7 ms

- Warp Shuffle (1024 ths/blk): **12.8 ms** (a >38% improvement)
- Shared Memory (1024 ths/blk): **14.3 ms** (a ~30% improvement)
- **Nvidia GeForce GTX Titan X:**
 - Original (basic atomicAdd, 32 ths/blk): 3.08 ms
 - Warp Shuffle (1024 ths/blk): **0.141 ms** (a massive >95% improvement!)
 - Shared Memory (1024 ths/blk): **0.150 ms** (also a huge >95% improvement!)

SEARCH CREATORS