Parallel and Distributed Computing

Alberto Paoluzzi – Lecture 18

Wed 27-04-2022

Source: Steve Lantz, Virtual Workshop: Understanding GPU Architecture, Cornell Center for Advanced Computing

1. GPU Characteristics
2. Threads and Cores Redefined
3. SIMT and Warps
4. Kernels and Streaming Multiprocessors (SMs)
5. Memory Levels
6. Memory Types – 1/2
7. Volta Block Diagram
8. Tensor Cores

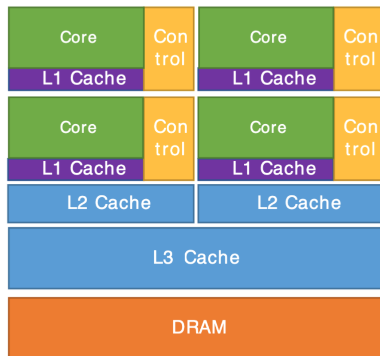
Section 1

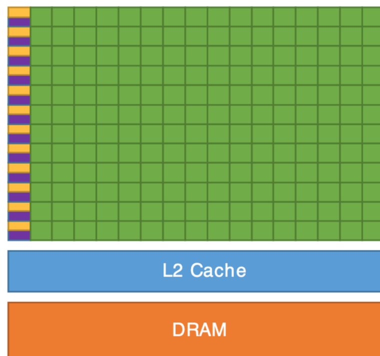
GPU Characteristics

GPU Characteristics

The transistor counts associated with various functions are represented abstractly by the relative sizes of the different shaded areas

In the figure, green corresponds to computation; gold is instruction processing; purple is L1 cache; blue is higher-level cache, and orange is memory (DRAM, which should really be thousands of times larger than the caches).





CPU vs GPU

The diagram above, which is taken from the CUDA C++ Programming Guide (v.11.2), does not depict the actual hardware design of any particular CPU or GPU

However, based on the size, color, and number of the various blocks, the figure does suggest that:

- CPUs can handle more complex workflows compared to GPUs.

This is indicated in the diagram by having just one gold control box for every row of the little green computational boxes.

CPU vs GPU

The diagram above, which is taken from the CUDA C++ Programming Guide (v.11.2), does not depict the actual hardware design of any particular CPU or GPU

However, based on the size, color, and number of the various blocks, the figure does suggest that:

- CPUs can handle more complex workflows compared to GPUs.
- CPUs don't have as many arithmetic logic units or floating point units as

This is indicated in the diagram by having just one gold control box for every row of the little green computational boxes.

CPU vs GPU

The diagram above, which is taken from the CUDA C++ Programming Guide (v.11.2), does not depict the actual hardware design of any particular CPU or GPU

However, based on the size, color, and number of the various blocks, the figure does suggest that:

- CPUs can handle more complex workflows compared to GPUs.
- CPUs don't have as many arithmetic logic units or floating point units as
- GPUs (the small green boxes above, roughly speaking), but the ALUs and FPUs in a CPU core are individually more capable.

This is indicated in the diagram by having just one gold control box for every row of the little green computational boxes.

CPU vs GPU

The diagram above, which is taken from the CUDA C++ Programming Guide (v.11.2), does not depict the actual hardware design of any particular CPU or GPU

However, based on the size, color, and number of the various blocks, the figure does suggest that:

- CPUs can handle more complex workflows compared to GPUs.
- CPUs don't have as many arithmetic logic units or floating point units as
- GPUs (the small green boxes above, roughly speaking), but the ALUs and FPUs in a CPU core are individually more capable.
- CPUs have more cache memory than GPUs.

This is indicated in the diagram by having just one gold control box for every row of the little green computational boxes.

CPU vs GPU

The diagram above, which is taken from the CUDA C++ Programming Guide (v.11.2), does not depict the actual hardware design of any particular CPU or GPU

However, based on the size, color, and number of the various blocks, the figure does suggest that:

- CPUs can handle more complex workflows compared to GPUs.
- CPUs don't have as many arithmetic logic units or floating point units as
- GPUs (the small green boxes above, roughly speaking), but the ALUs and FPUs in a CPU core are individually more capable.
- CPUs have more cache memory than GPUs.
- GPUs are really designed for workloads that can be parallelized to a significant degree

This is indicated in the diagram by having just one gold control box for every row of the little green computational boxes.

Section 2

Threads and Cores Redefined

Threads and Cores Redefined

What is the secret to the high performance that can be achieved by a GPU?

The answer lies in the graphics pipeline that the GPU is meant to “pump”: the sequence of steps required to take a scene of geometrical objects described in 3D coordinates and render them on a 2D display.

Two key properties of the graphics pipeline permit its speed to be accelerated

1. a typical scene is composed of many independent objects (e.g., a mesh of tiny triangles approximating a surface)

By their very nature, then, GPUs must be highly capable parallel computing engines.

Threads and Cores Redefined

What is the secret to the high performance that can be achieved by a GPU?

The answer lies in the graphics pipeline that the GPU is meant to “pump”: the sequence of steps required to take a scene of geometrical objects described in 3D coordinates and render them on a 2D display.

Two key properties of the graphics pipeline permit its speed to be accelerated

1. a typical scene is composed of many independent objects (e.g., a mesh of tiny triangles approximating a surface)
2. the sequence of steps needed to render each of the objects is basically the same for all of the objects, so that the computational steps may be performed in parallel on all them at once

By their very nature, then, GPUs must be highly capable parallel computing engines.

CPUs vs GPUs

But CPUs, too, have evolved to become highly capable parallel processors in their own right—and in this evolution, they have acquired certain similarities to GPUs

Therefore, it is not surprising to find a degree of overlap in the terminology used to describe the parallelism in both kinds of processors

However, one should be careful to understand the distinctions as well, because the precise meanings of terms can differ signficantly between the two types of devices.

For example, with CPUs as well as GPUs, one may speak of threads that run on different cores

In both cases, one envisions distinct streams of instructions that are scheduled to run on different execution units

CPUs vs GPUs

Yet the ways in which threads and cores act upon data are quite different in the two cases.

It turns out that a single core in a GPU—which we'll call a CUDA core hereafter, for clarity—is much more like a single vector lane in the vector processing unit of a CPU

Why? Because CUDA cores are essentially working in teams of 32 to execute a Single Instruction on Multiple Data, a type of parallelism known as SIMD

In CPUs, SIMD operations are possible as well, but they are carried out by vector units, based on smaller data groupings (typically 8 or 16 elements).

Comparison table

The table below attempts to reduce the potential sources of confusion

It lists and defines the terms that apply to the various levels of parallelism in a GPU, and gives their rough equivalents in CPU terminology

GPU term	Quick definition for a GPU	CPU equivalent
thread	The stream of instructions and data that is assigned to one CUDA core; note, a Single Instruction applies to Multiple Threads, acting on multiple data (SIMT)	N/A
CUDA core	Unit that processes one data item after another, to execute its portion of a SIMT instruction stream	vector lane
warp	Group of 32 threads that executes the same stream of instructions together, on different data	vector
kernel	Function that runs on the device; a kernel may be subdivided into *thread blocks*	***thread(s)***
SM, streaming multiprocessor	Unit capable of executing a thread block of a kernel; multiple SMs may work together on a kernel	***core***

Figura 2: Comparison table

Section 3

SIMT and Warps

SIMT and Warps

SIMT

As you might expect, the NVIDIA term "Single Instruction Multiple Threads" (SIMT) is closely related to a better known term, Single Instruction Multiple Data (SIMD).

Single Instruction Multiple Threads" (SIMT)

What's the difference between SIMD and SIMT?

Single Instruction Multiple Threads" (SIMT)

What's the difference between SIMD and SIMT?

- In pure SIMD, a single instruction acts upon all the data in exactly the same way

Single Instruction Multiple Threads" (SIMT)

What's the difference between SIMD and SIMT?

- In pure SIMD, a single instruction acts upon all the data in exactly the same way
- In SIMT, this restriction is loosened a bit: selected threads can be activated or deactivated, so that instructions and data are processed only on the active threads, while the local data remain unchanged on inactive threads.

SIMT can accommodate branching

Given an `if-else` construct beginning with `if (condition)`, the threads for which `condition==true` will be active when running statements in the `if` clause,

and the threads for which `condition==false` will be active when running statements in the `else` clause

SIMT can accommodate branching

Given an `if-else` construct beginning with `if (condition)`, the threads for which `condition==true` will be active when running statements in the `if` clause,

and the threads for which `condition==false` will be active when running statements in the `else` clause

The results should be correct, but the inactive threads will do no useful work while they are waiting for statements in the active clause to complete

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

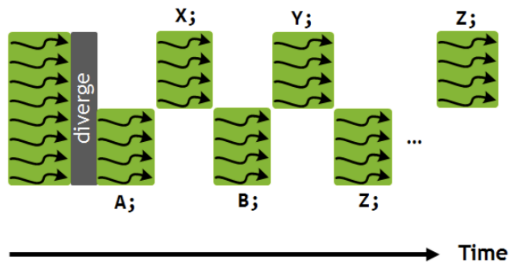


Figura 3: CBranching within SIMT

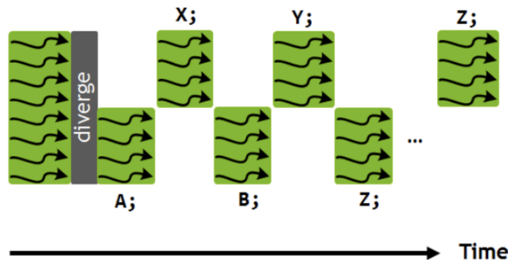
Synchronization of shared data at intermediate points

Note that in NVIDIA GPUs prior to Volta, the entire `if` clause (i.e., both statements A and B) would have to be executed by the relevant threads,

then the entire `else` clause (both statements X and Y) would have to be executed by the remainder of the threads, then all threads would have to synchronize before continuing execution (statement Z)

Volta's more flexible SIMT model permits synchronization of shared data at intermediate points (say, after A and X).

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



CPU code vs SIMT parallelism on the GPU

In contrast to how CPU code is written, SIMT parallelism on the GPU does not have to be expressed via “vectorized loops”

Instead—at least in CUDA—every GPU thread executes the kernel code as written

This somewhat justifies NVIDIA’s “thread” nomenclature

But note that GPU code can also be written using OpenMP or OpenACC directives, in which case it can end up looking very much like vectorized CPU code.

Block of threads divided into warps for SIMT execution

One warp consists of a bundle of 32 threads with consecutive thread indexes.

The threads in a warp are then processed together by a set of 32 CUDA cores.

Block of threads divided into warps for SIMT execution

One warp consists of a bundle of 32 threads with consecutive thread indexes.

The threads in a warp are then processed together by a set of 32 CUDA cores.

vectorized loop on a CPU is chunked into vectors of a fixed size

then processed by a set of vector lanes.

Origin of name Warp

The reason for bundling threads into warps of 32 is simply that in NVIDIA's hardware, CUDA cores are divided into fixed groups of 32

Each such group is analogous to a vector processing unit (VPU) in a CPU

Breaking down a large block of threads into chunks of this size simplifies the SM's task of scheduling the entire thread block on its available resources.

Apparently NVIDIA borrowed the term "warp" from weaving, where it refers to the set of vertical threads through which the weaver's shuttle passes

To quote the original paper by Lindholm et al that introduced SIMT, "The term warp originates from weaving, the first parallel-thread technology." (NVIDIA continues to use this quote in their CUDA C++ Programming Guide.)

Section 4

Kernels and Streaming Multiprocessors (SMs)

Kernels and Streaming Multiprocessors (SMs)

We continue our survey of GPU-related terminology by looking at the relationship between kernels, thread blocks, and streaming multiprocessors (SMs).

Kernels (in software)

A function that is meant to be executed in parallel on an attached GPU is called a kernel

In CUDA, a kernel is usually identified by the presence of the `__global__` specifier in front of an otherwise normal-looking C++ function declaration

The designation `__global__` means the kernel may be called from either the host or the device, but it will execute on the device.

Instead of being executed only once, a kernel is executed N times in parallel by N different threads on the GPU

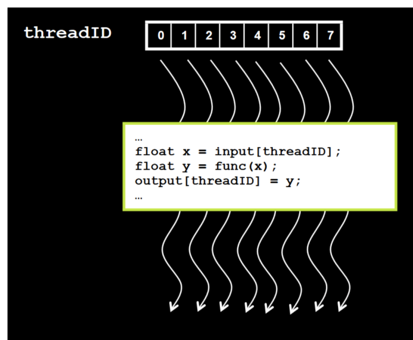
Each thread is assigned a unique ID (in effect, an index) that it can use to compute memory addresses and make control decisions.

CUDA kernel execution

A CUDA kernel is executed by an array of threads

All threads run the same code

Each thread has an ID that it uses to compute memory addresses and make control decisions



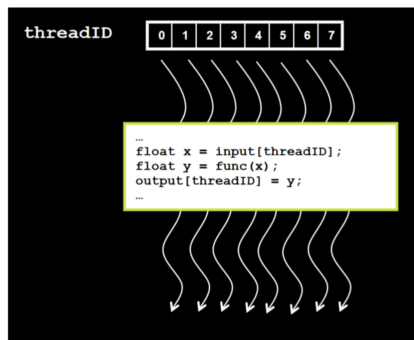
- Accordingly, kernel calls must supply special arguments specifying how many threads to use on the GPU

CUDA kernel execution

A CUDA kernel is executed by an array of threads

All threads run the same code

Each thread has an ID that it uses to compute memory addresses and make control decisions



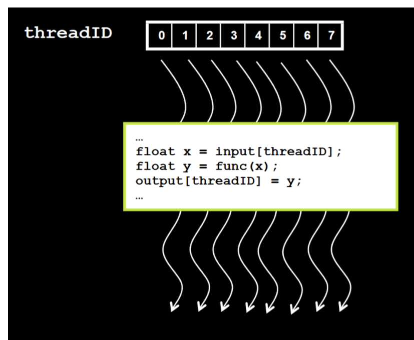
- Accordingly, kernel calls must supply special arguments specifying how many threads to use on the GPU
- They do this using CUDA's "execution configuration" syntax, which looks like this: `fun<<<1, N>>>(x, y, z)`

CUDA kernel execution

A CUDA kernel is executed by an array of threads

All threads run the same code

Each thread has an ID that it uses to compute memory addresses and make control decisions



- Accordingly, kernel calls must supply special arguments specifying how many threads to use on the GPU
- They do this using CUDA's "execution configuration" syntax, which looks like this: `fun<<<1, N>>>(x, y, z)`
- Note that the first entry in the configuration (1, in this case) gives the number of blocks of N threads that will be launched

Streaming Sultiprocessors (in hardware)

On the GPU, a kernel call is executed by one or more streaming multiprocessors, or SMs

The SMs are the hardware homes of the CUDA cores that execute the threads

The CUDA cores in each SM are always arranged in sets of 32 so that the SM can use them to execute full warps of threads

The exact number of SMs available in a device depends on its NVIDIA processor family (Volta, Turing, etc.), as well as the specific model number of the processor

Thus, the Volta chip in the Tesla V100 has 80 SMs in total, while the more recent Turing chip in the Quadro RTX 5000 has just 48.

number of SMs that the GPU will actually use

However, the number of SMs that the GPU will actually use to execute a kernel call is limited to the number of thread blocks specified in the call

Taking the call `fun<<<M, N>>>(x, y, z)` as an example, there are at most `M` blocks that can be assigned to different SMs

A thread block may not be split between different SMs

(If there are more blocks than available SMs, then more than one block may be assigned to the same SM.) By distributing blocks in this manner, the GPU can run independent blocks of threads in parallel on different SMs.

SM's schedulers and Local memory

Each SM then divides the N threads in its current block into warps of 32 threads for parallel execution internally

On every cycle, each SM's schedulers are responsible for assigning full warps of threads to run on available sets of 32 CUDA cores

(The Volta architecture has 4 such schedulers per SM.) Any leftover, partial warps in a thread block will still be assigned to run on a set of 32 CUDA cores.

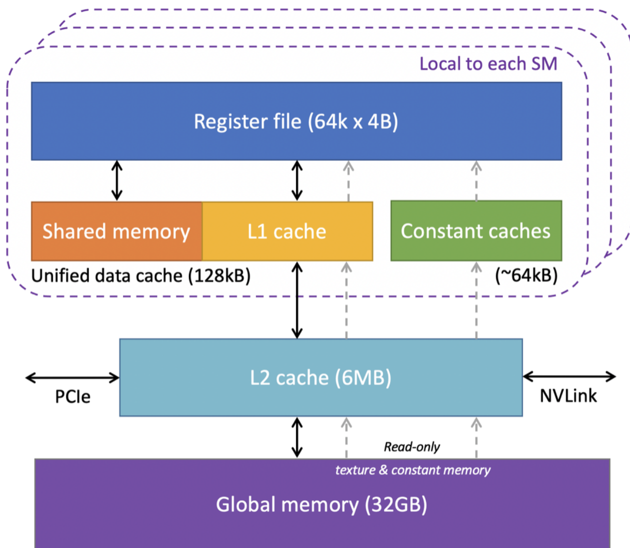
Local memory

The SM includes several levels of memory that can be accessed only by the CUDA cores of that SM: registers, L1 cache, constant caches, and shared memory
The exact properties of the per-SM and global memory available in Volta GPUs will be outlined shortly.

Section 5

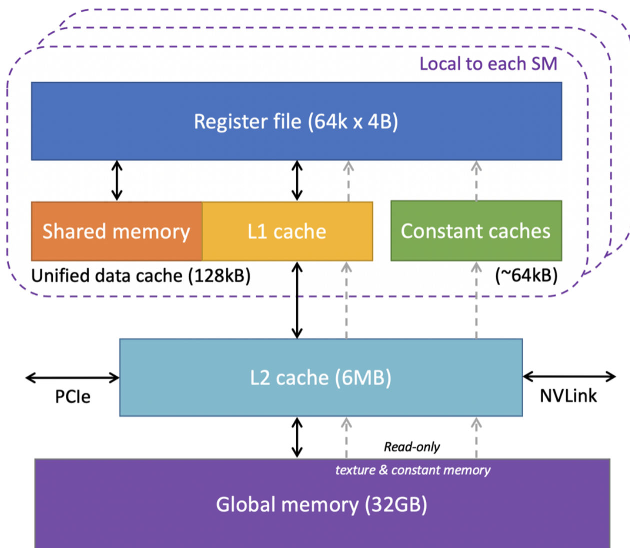
Memory Levels

Memory Levels



Like a CPU, the GPU relies on a memory hierarchy to ensure that its processing engines are kept supplied with the data they need

Memory Levels



Like a CPU, the GPU relies on a memory hierarchy to ensure that its processing engines are kept supplied with the data they need

Like cores in a CPU, the streaming multiprocessors (SMs) in a GPU require data to be in registers to be available for computations.

Memory Levels

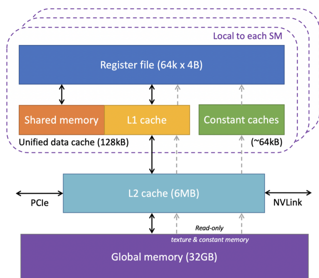


Figura 7: Memory Levels

Depending on where the data start, they may have to hop through several layers of cache to enter the registers of an SM and become accessible to the CUDA cores.

Memory Levels

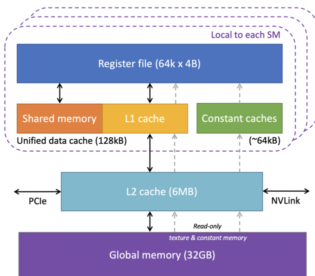


Figura 7: Memory Levels

Depending on where the data start, they may have to hop through several layers of cache to enter the registers of an SM and become accessible to the CUDA cores.

Global memory is by far the largest layer, but it is also furthest from the SMs

Clearly it would be favorable for 4-byte operands to travel together in groups of 32 as they move back and forth between caches and registers and CUDA cores

Memory Levels

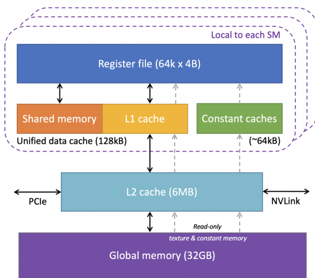


Figura 7: Memory Levels

Depending on where the data start, they may have to hop through several layers of cache to enter the registers of an SM and become accessible to the CUDA cores.

Global memory is by far the largest layer, but it is also furthest from the SMs
Clearly it would be favorable for 4-byte operands to travel together in groups of 32 as they move back and forth between caches and registers and CUDA cores

Why? A 32-wide group is exactly right to supply a warp of 32 threads, all at once. Therefore, it makes perfect sense that the size of the cache line in a GPU is 32 x (4 bytes) = 128 bytes.

Memory Levels

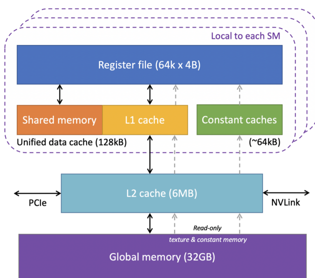


Figura 7: Memory Levels

Depending on where the data start, they may have to hop through several layers of cache to enter the registers of an SM and become accessible to the CUDA cores.

Global memory is by far the largest layer, but it is also furthest from the SMs
Clearly it would be favorable for 4-byte operands to travel together in groups of 32 as they move back and forth between caches and registers and CUDA cores

Why? A 32-wide group is exactly right to supply a warp of 32 threads, all at once. Therefore, it makes perfect sense that the size of the cache line in a GPU is 32 x (4 bytes) = 128 bytes.

Notice that data transfers onto and off of the device are mediated by the L2 cache. In most cases, the incoming data will proceed from the L2 into the large global memory of the device.

Section 6

Memory Types – 1/2

Memory Types – 1/2

The first list covers the on-chip memory areas that are closest to the CUDA cores. They are part of every SM

Memory Types – 1/2

The first list covers the on-chip memory areas that are closest to the CUDA cores. They are part of every SM

Register File - denotes the area of memory that feeds directly into the CUDA cores. It is organized into 32 banks, matching the 32 threads in a warp. Think of the register file as a big matrix of 4-byte elements, having many rows and 32 columns. A warp operates on full rows; within a given row, each thread (CUDA core) operates on a different column (bank)

Memory Types – 1/2

The first list covers the on-chip memory areas that are closest to the CUDA cores. They are part of every SM

Register File - denotes the area of memory that feeds directly into the CUDA cores. It is organized into 32 banks, matching the 32 threads in a warp. Think of the register file as a big matrix of 4-byte elements, having many rows and 32 columns. A warp operates on full rows; within a given row, each thread (CUDA core) operates on a different column (bank)

L1 Cache - refers to the usual on-chip storage location providing fast access to data that are recently read from, or written to, main memory (RAM). L1 serves as the overflow region when the amount of active data exceeeds what an SM's register file can hold, a condition which is termed "register spilling". In L1, the cache lines and spilled registers are organized into banks, just as in the register file

Memory Types – 1/2

The first list covers the on-chip memory areas that are closest to the CUDA cores. They are part of every SM

Register File - denotes the area of memory that feeds directly into the CUDA cores. It is organized into 32 banks, matching the 32 threads in a warp. Think of the register file as a big matrix of 4-byte elements, having many rows and 32 columns. A warp operates on full rows; within a given row, each thread (CUDA core) operates on a different column (bank)

L1 Cache - refers to the usual on-chip storage location providing fast access to data that are recently read from, or written to, main memory (RAM). L1 serves as the overflow region when the amount of active data exceeeds what an SM's register file can hold, a condition which is termed "register spilling". In L1, the cache lines and spilled registers are organized into banks, just as in the register file

Shared Memory - is a memory area that physically resides in the same memory as the L1 cache, but differs from L1 in that all its data may be accessed by any thread in a thread block. This allows threads to communicate and share data with each other. Variables that occupy it must be declared explicitly by an application. The application can also set the dividing line between L1 and shared memory

Memory Types – 1/2

The first list covers the on-chip memory areas that are closest to the CUDA cores. They are part of every SM

Register File - denotes the area of memory that feeds directly into the CUDA cores. It is organized into 32 banks, matching the 32 threads in a warp. Think of the register file as a big matrix of 4-byte elements, having many rows and 32 columns. A warp operates on full rows; within a given row, each thread (CUDA core) operates on a different column (bank)

L1 Cache - refers to the usual on-chip storage location providing fast access to data that are recently read from, or written to, main memory (RAM). L1 serves as the overflow region when the amount of active data exceeeds what an SM's register file can hold, a condition which is termed "register spilling". In L1, the cache lines and spilled registers are organized into banks, just as in the register file

Shared Memory - is a memory area that physically resides in the same memory as the L1 cache, but differs from L1 in that all its data may be accessed by any thread in a thread block. This allows threads to communicate and share data with each other. Variables that occupy it must be declared explicitly by an application. The application can also set the dividing line between L1 and shared memory

Constant Caches - are special caches pertaining to variables declared as read-only constants in global memory. Such variables can be read by any thread in a thread block. The main and best use of these caches is to broadcast a single constant value to all the threads in a warp

Memory Types – 2/2

The second list pertains to the more distant, larger memory areas that are shared by all the SMs

Memory Types – 2/2

The second list pertains to the more distant, larger memory areas that are shared by all the SMs

L2 Cache - is a further on-chip cache for retaining copies of the data that travel back and forth between the SMs and main memory. Like the L1, the L2 cache is intended to speed up subsequent reloads. But unlike the L1 cache(s), there is just one L2 that is shared by all the SMs. The L2 cache is also situated in the path of data moving on or off the device via PCIe or NVLink

Memory Types – 2/2

The second list pertains to the more distant, larger memory areas that are shared by all the SMs

L2 Cache - is a further on-chip cache for retaining copies of the data that travel back and forth between the SMs and main memory. Like the L1, the L2 cache is intended to speed up subsequent reloads. But unlike the L1 cache(s), there is just one L2 that is shared by all the SMs. The L2 cache is also situated in the path of data moving on or off the device via PCIe or NVLink

Global Memory - represents the bulk of the main memory of the device, equivalent to RAM in a CPU-based processor. For performance reasons, the Tesla V100 has special HBM2 high-bandwidth memory, while the Quadro RTX 5000 has fast GDDR6 graphics memory

Memory Types – 2/2

The second list pertains to the more distant, larger memory areas that are shared by all the SMs

L2 Cache - is a further on-chip cache for retaining copies of the data that travel back and forth between the SMs and main memory. Like the L1, the L2 cache is intended to speed up subsequent reloads. But unlike the L1 cache(s), there is just one L2 that is shared by all the SMs. The L2 cache is also situated in the path of data moving on or off the device via PCIe or NVLink

Global Memory - represents the bulk of the main memory of the device, equivalent to RAM in a CPU-based processor. For performance reasons, the Tesla V100 has special HBM2 high-bandwidth memory, while the Quadro RTX 5000 has fast GDDR6 graphics memory

Local Memory - corresponds to specially mapped regions of main memory that are assigned to each SM. Whenever "register spilling" overflows the L1 cache on a particular SM, the excess data are further offloaded to L2, then to "local memory". The performance penalty for reloading a spilled register becomes steeper for every memory level that must be traversed in order to retrieve it

Memory Types – 2/2

The second list pertains to the more distant, larger memory areas that are shared by all the SMs

L2 Cache - is a further on-chip cache for retaining copies of the data that travel back and forth between the SMs and main memory. Like the L1, the L2 cache is intended to speed up subsequent reloads. But unlike the L1 cache(s), there is just one L2 that is shared by all the SMs. The L2 cache is also situated in the path of data moving on or off the device via PCIe or NVLink

Global Memory - represents the bulk of the main memory of the device, equivalent to RAM in a CPU-based processor. For performance reasons, the Tesla V100 has special HBM2 high-bandwidth memory, while the Quadro RTX 5000 has fast GDDR6 graphics memory

Local Memory - corresponds to specially mapped regions of main memory that are assigned to each SM. Whenever "register spilling" overflows the L1 cache on a particular SM, the excess data are further offloaded to L2, then to "local memory". The performance penalty for reloading a spilled register becomes steeper for every memory level that must be traversed in order to retrieve it

Texture and Constant Memory - are regions of main memory that are treated as read-only by the device. When fetched to an SM, variables with a "texture" or "constant" declaration can be read by any thread in a thread block, much like shared memory. Texture memory is cached in L1, while constant memory is cached in the constant caches

Section 7

Volta Block Diagram

Volta Block Diagram

The NVIDIA Tesla V100 accelerator is built around the Volta GV100 GPU.

Section 8

Tensor Cores

Tensor Cores

The basic role of a tensor core is to perform the following operation on 4x4 matrices:

$$D = A \times B + C$$

In this formula, the inputs A and B are FP16 matrices, while the input and accumulation matrices C and D may be FP16 or FP32 matrices

Figure 0: How an NVIDIA tensor core operates on 4x4 matrices

Tensor Cores

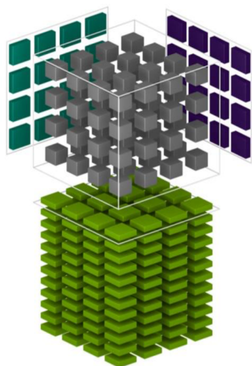


Figura 10: 3D illustration of the action of a tensor core

The two matrices to be multiplied, `A` and `B`, are depicted outside the central cube (note, matrix `A` on the left is transposed)

Tensor Cores

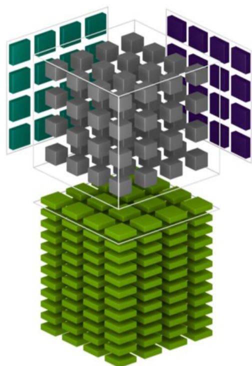


Figura 10: 3D illustration of the action of a tensor core

The two matrices to be multiplied, A and B, are depicted outside the central cube (note, matrix A on the left is transposed)

The cube itself represents the 64 element-wise products required to generate the full 4x4 product matrix

Tensor Cores

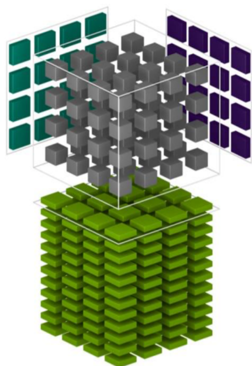


Figura 10: 3D illustration of the action of a tensor core

The two matrices to be multiplied, `A` and `B`, are depicted outside the central cube (note, matrix `A` on the left is transposed)

The cube itself represents the 64 element-wise products required to generate the full 4x4 product matrix

Imagine all 64 blocks within the cube "lighting up" at once, as pairs of input elements are instantaneously multiplied together along horizontal layers, then instantaneously summed along vertical lines

Tensor Cores

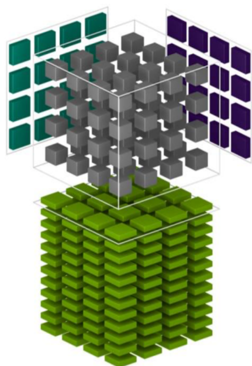


Figura 10: 3D illustration of the action of a tensor core

The two matrices to be multiplied, `A` and `B`, are depicted outside the central cube (note, matrix `A` on the left is transposed)

The cube itself represents the 64 element-wise products required to generate the full 4x4 product matrix

Imagine all 64 blocks within the cube "lighting up" at once, as pairs of input elements are instantaneously multiplied together along horizontal layers, then instantaneously summed along vertical lines

As a result, a whole product matrix ($A \times B$, transposed) drops down onto the top of the pile, where it is summed with matrix C (transposed), outlined in white

Tensor Cores

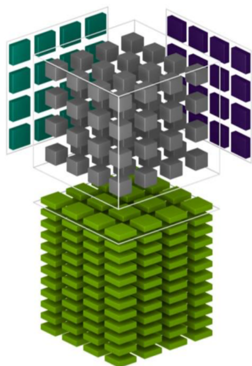


Figura 10: 3D illustration of the action of a tensor core

The two matrices to be multiplied, `A` and `B`, are depicted outside the central cube (note, matrix `A` on the left is transposed)

The cube itself represents the 64 element-wise products required to generate the full 4x4 product matrix

Imagine all 64 blocks within the cube "lighting up" at once, as pairs of input elements are instantaneously multiplied together along horizontal layers, then instantaneously summed along vertical lines

As a result, a whole product matrix ($A \times B$, transposed) drops down onto the top of the pile, where it is summed with matrix C (transposed), outlined in white

Upon summation it becomes the next output matrix D and is pushed down onto the stack of results

Tensor Cores

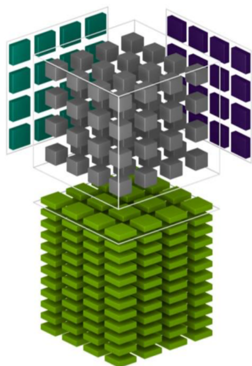


Figura 10: 3D illustration of the action of a tensor core

The two matrices to be multiplied, `A` and `B`, are depicted outside the central cube (note, matrix `A` on the left is transposed)

The cube itself represents the 64 element-wise products required to generate the full 4x4 product matrix

Imagine all 64 blocks within the cube "lighting up" at once, as pairs of input elements are instantaneously multiplied together along horizontal layers, then instantaneously summed along vertical lines

As a result, a whole product matrix ($A \times B$, transposed) drops down onto the top of the pile, where it is summed with matrix C (transposed), outlined in white

Upon summation it becomes the next output matrix D and is pushed down onto the stack of results

Prior output matrices are shown piling up below the cube, beneath the latest output matrix D (all transposed).