Christof Teuscher

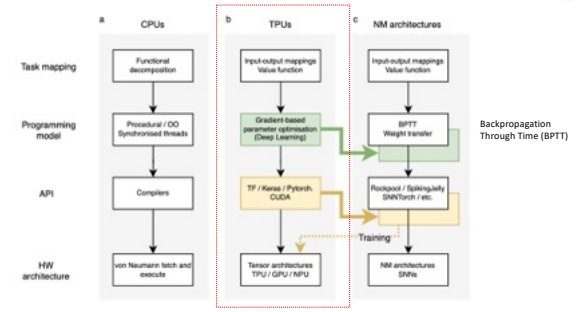
ECE 410/510: Hardware for AI and ML

GPUs + CUDA + CNN/DNN

Portland State University
Department of Electrical and Computer Engineering (ECE)
www.teuscher-lab.com
teuscher@pdx.edu

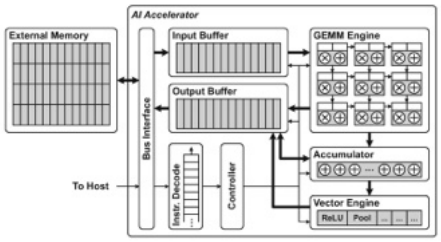
teuscher Lab teuscher-lab.com

Portland State UNIVERSITY



https://doi.org/10.1038/s41467-025-57352-1

General AI accelerator architecture



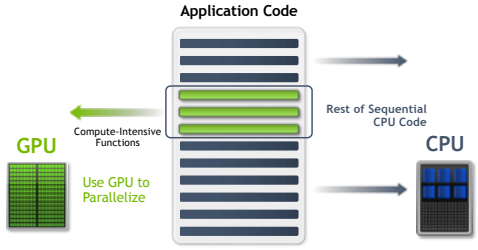
GEMM = General Matrix Multiply

Tensor cores!

A vector is a 1st-order tensor
A matrix is a 2nd-order tensor.

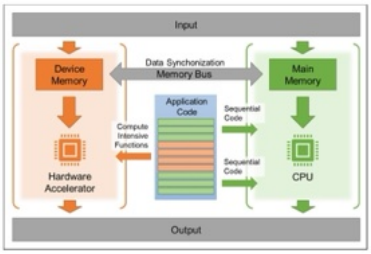
Mishra, Ashutosh; Cha, Jaekwang; Park, Hyunbin; Kim, Shiho. Artificial Intelligence and Hardware Accelerators, 2025

General co-processing mechanism



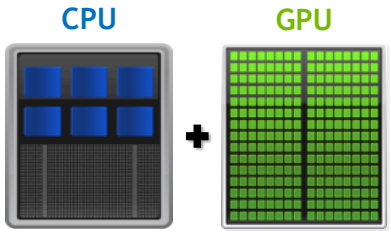
NVIDIA

General co-processing mechanism



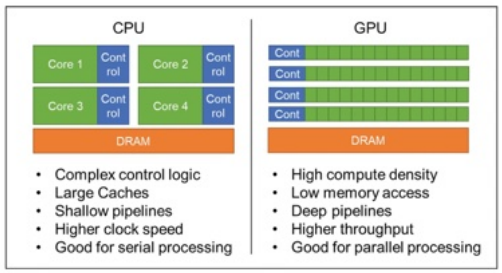
Mishra, Ashutosh; Cha, Jaekwang; Park, Hyunbin; Kim, Shiho. Artificial Intelligence and Hardware Accelerators, 2025

Multi-core vs GPU



NVIDIA

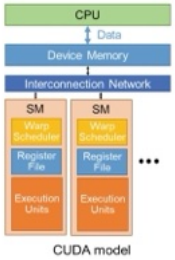
Multi-core vs GPU

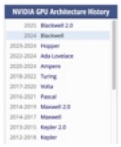


SIMT (Single Instruction, Multiple Threads) is a parallel processing model used in GPUs where multiple threads execute the same instruction simultaneously, but on different data.

Mishra, Ashutosh; Cha, Jaekwang; Park, Hyunbin; Kim, Shiho. Artificial Intelligence and Hardware Accelerators, 2025

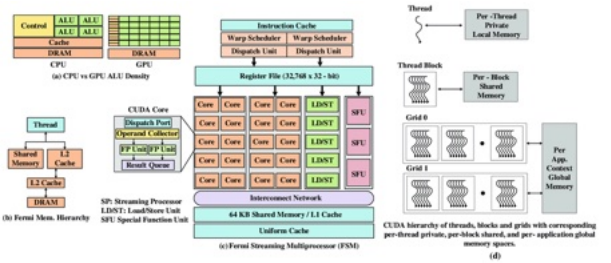
GPU/CUDA model





Mishra, Ashutosh; Cha, Jaekwang; Park, Hyunbin; Kim, Shiho. Artificial Intelligence and Hardware Accelerators, 2025

NVIDIA Fermi Microarchitecture



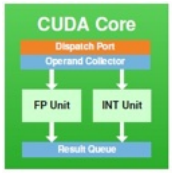
10.1109/ACCESS.2020.3008250

What is a CUDA Core?

FP:
- Addition, subtraction, multiplication, division, square roots.
- CUDA supports both single-precision (FP32) and double-precision (FP64) floating-point operations

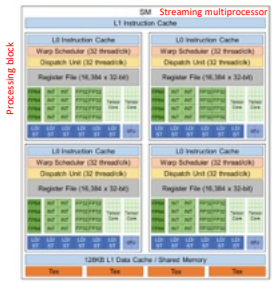
INT:
- additions, subtractions, multiplications, comparison, and bitwise operations

SFU:
- accelerates the execution of specific transcendental and trigonometric functions like sine, cosine, tangent, and exponential calculations

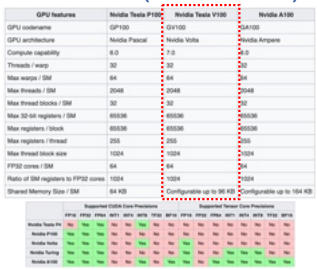


https://www.electronicshub.org/cuda-cores/

NVIDIA Volta Microarchitecture (2013-2017)



Tex = Texture units
SFU = Special Function Unit



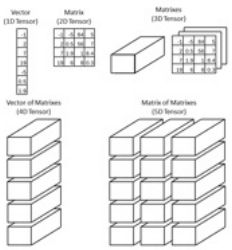
- First chip to feature tensor cores
- A GV100 SM is partitioned into four processing blocks

Warp

- A "warp" is a group of threads that execute instructions simultaneously on a GPU (Graphics Processing Unit).
- The warp scheduler is a component of the GPU architecture that manages the execution of these thread groups.
- A warp typically consists of 32 threads in NVIDIA GPUs (AMD calls similar groupings "wavefronts"). These threads execute the same instruction at the same time in a **SIMT** (Single Instruction, Multiple Thread) architecture.
- The warp scheduler is responsible for:
  - Deciding which warps are ready to execute
  - Allocating compute resources to warps
  - Managing warp context switching when threads need to wait for memory operations.
  - Hiding memory latency by switching between warps

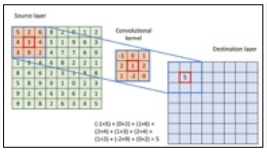
What is a tensor?

What is a tensor?

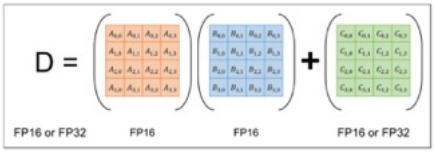


Tensor cores

- Each tensor core executes MMA (matrix multiplication and accumulation, D = A × B + C) on a 4 × 4 floating-point matrix.
- They are built to enhance deep learning by boosting matrix arithmetic.
- Low-precision operations often supported.



Convolution



MMA

Mishra et al., 2023

Tensor cores (1)

Each Tensor Core operates on a 4x4 matrix and performs the following operation:

D = A×B + C

where A, B, C, and D are 4x4 matrices (Figure 8). The matrix multiply inputs A and B are FP16 matrices, while the accumulation matrices C and D may be FP16 or FP32 matrices (see Figure 8).

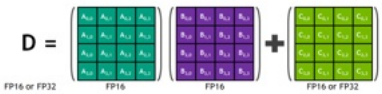


Figure 8. Tensor Core 4x4 Matrix Multiply and Accumulate

Tensor cores (2)

Tensor Cores operate on FP16 input data with FP32 accumulation. The FP16 multiply results in a full precision product that is then accumulated using FP32 addition with the other intermediate products for a 4x4x4 matrix multiply (see Figure 9). In practice, Tensor Cores are used to perform much larger 2D or higher dimensional matrix operations, built up from these smaller elements.

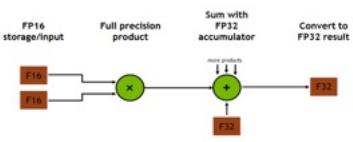


Figure 9. Mixed Precision Multiply and Accumulate in Tensor Core

Tensor cores (3)

Figure 10 shows the 4x4 matrix multiplication (using the two source 4x4 matrices outside the cube) requiring 64 operations (represented by the cube) to generate a 4x4 output matrix (shown below the cube). The Volta-based V100 accelerator with Tensor Cores can perform such calculations at 12x faster rate than Pascal-based Tesla P100.

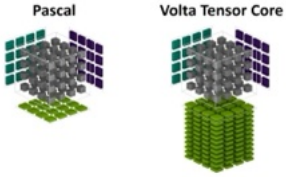
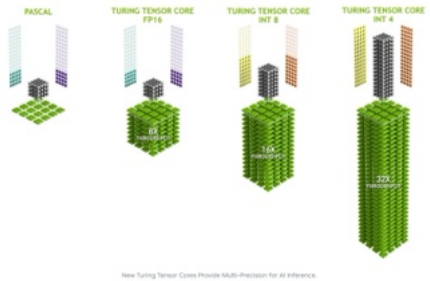


Figure 10. Pascal and Volta 4x4 Matrix Multiplication

Tensor cores (4)



https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth

CUDA vs tensor cores (1)

Metric	CUDA Cores	Tensor Cores
Primary Function	General-purpose Computing	Specialized for deep learning matrix computations
Arithmetic Operations	Executes parallel tasks efficiently	Optimized for matrix multiplication
Ideal Applications	Scientific research, machine learning, gaming, and more	Deep learning models, neural network training
Raw Power	Generally, a higher number of cores per GPU	Fewer cores, but highly optimized for specific tasks
Performance Optimization	Efficient for complex algorithms in various fields	Accelerates matrix operations frequently used in AI
>Use Case Suitability	A broad range of applications beyond deep learning	Specific tasks like training large-scale neural networks
Advantage in Deep Learning	Less efficient for matrix-heavy computations	Significant speedups and improved efficiency
Choice Based on Requirements	Suitable for diverse computing needs	Ideal for deep learning projects requiring extensive matrix computations
Future Prospects	Continuous improvements expected	Continuous improvements expected

https://acecloud.ai/resources/blog/cuda-cores-vs-tensor-cores

CUDA vs tensor cores (2)

CUDA Cores	Tensor Cores
CUDA cores deliver very high accuracy but at the cost of computing speed.	Tensor cores take the hit in accuracy but deliver accelerated computational speed.
CUDA cores are adequate for typical Machine Learning training. But enterprises predominantly prefer to use these for graphics-rich visual content and accelerated Ray-Tracing tasks.	Tensor cores are perfect for both low-end and high-end enterprise-grade AI development, data-rich ML training, and Deep Learning model development with multiple layers in neural networks.
CUDA cores are general-purpose cores that serve various tasks, such as graphics rendering, video editing, data analysis, machine learning, cryptographic operations, etc.	Tensor cores are specialized cores that perform high-end computations and allow mixed-precision training.
Enterprises that require small-scale/ individual-level visual/video editing, game development tasks, Data analytics operations, and Indie research prefer CUDA-Cores GPUs.	Enterprises that demand robust, dedicated, and energy-efficient operation or research, such as managing data centers, massive AI training projects, High Performance Computing (HPC)), etc., prefer Tensor core GPUs.

https://acecloud.ai/resources/blog/cuda-cores-vs-tensor-cores

CUDA vs tensor cores (2)

CUDA Cores	Tensor Cores
CUDA cores deliver very high accuracy but at the cost of computing speed.	Tensor cores take the hit in accuracy but deliver accelerated computational speed.
CUDA cores are adequate for typical Machine Learning training. But enterprises predominantly prefer to use these for graphics-rich visual content and accelerated Ray-Tracing tasks.	Tensor cores are perfect for both low-end and high-end enterprise-grade AI development, data-rich ML training, and Deep Learning model development with multiple layers in neural networks.
CUDA cores are general-purpose cores that serve various tasks, such as graphics rendering, video editing, data analysis, machine learning, cryptographic operations, etc.	Tensor cores are specialized cores that perform high-end computations and allow mixed-precision training.
Enterprises that require small-scale/ individual-level visual/video editing, game development tasks, Data analytics operations, and Indie research prefer CUDA-Cores GPUs.	Enterprises that demand robust, dedicated, and energy-efficient operation or research, such as managing data centers, massive AI training projects, High Performance Computing (HPC)), etc., prefer Tensor core GPUs.

https://acecloud.ai/resources/blog/cuda-cores-vs-tensor-cores

CUDA vs tensor cores (3)

CUDA Cores	Tensor Cores
CUDA Cores cannot reduce the use cycles needed for mixed multiplication and addition operations.	Tensor cores can reduce the number of cycles needed for mixed multiplication and addition operations.
CUDA Cores are slower vis-a-vis Tensor cores. Thus, Data scientists and ML Engineers do not prefer them for extensive ML training.	Tensor cores have significantly high computational power and are a perfect choice for Data Science, ML development, and algorithm training.
CUDA Cores perform a single operation in one clock cycle. 1X1 per GPU clock	Tensor cores can perform multiple operations in one clock cycle. (1!1 (1!1 1!1 x 1!1 1!1) 1!1) per GPU clock
These are excellent for high-end graphics-based operations like Ray Tracing, rendering physics engines, and other editing purposes that require accuracy.	These are excellent for handling Machine Learning operations that feed on massive data collection or extensive Artificial Neural Network (ANN) layers.
Accelerated rendering of 3D graphics in Cloud-based games can leverage CUDS Cores.	Cloud services such as GPU-as-a-Service will prefer Tensor cores over Cuda cores.

https://acecloud.ai/resources/blog/cuda-cores-vs-tensor-cores

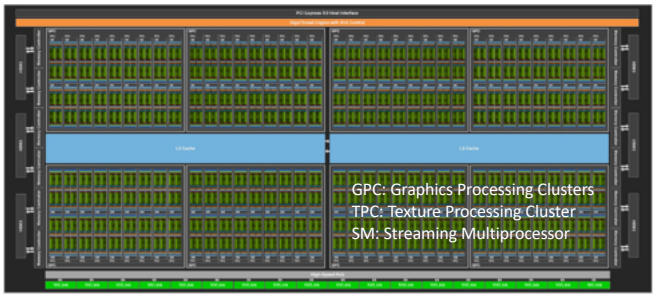
Graphics Card	GeForce RTX 3090	GeForce RTX 4090	GeForce RTX 5090
GPU Codename	GA102	AD102	GB202
GPU Architecture	NVIDIA Ampere	NVIDIA Ada Lovelace	NVIDIA Blackwell
GPCs	7	11	11
TPCs	41	64	85
SMs	82	128	170
CUDA Cores / SM	128	128	128
CUDA Cores / GPU	10496	16384	21760
Tensor Cores / SM	4 (3rd Gen)	4 (4th Gen)	4 (5th Gen)
Tensor Cores / GPU	328 (3rd Gen)	512 (4th Gen)	680 (5th Gen)
GPU Boost Clock (MHz)	1695	2520	2407
RT Cores	82 (2nd Gen)	128 (3rd Gen)	170 (4th Gen)
RT TFLOPS	69.5	191	317.5
Frame Buffer Memory Size and Type	24 GB GDDR6X	24 GB GDDR6X	32 GB GDDR7
Memory Interface	384-bit	384-bit	512-bit
Memory Clock (Data Rate)	19.5 Gbps	21 Gbps	28 Gbps
Memory Bandwidth	936 GB/sec	1008 GB/sec	1792 GB/sec
ROPs	112	176	176

Consumer-grade GPUs

Pixel Fill-rate (Gigapixels/sec)	189.8	443.5	423.6
Texture Units	328	512	680
Texel Fill-rate (Gigatexels/sec)	555.96	1290.2	1636.76
L1 Data Cache/Shared Memory	10496 KB	16384 KB	21760 KB
L2 Cache Size	6144 KB	73728 KB	98304 KB
TGP (Total Graphics Power)	350 W	450 W	575 W
Manufacturing Process	Samsung 8 nm 8N NVIDIA Custom Process	TSMC 4nm 4N NVIDIA Custom Process	TSMC 4nm 4N NVIDIA Custom Process
PCI Express Interface	Gen 4	Gen 4	Gen 5

GPC: Graphics Processing Clusters
TPC: Texture Processing Cluster
SM: Streaming Multiprocessor
RT: Ray Tracing
ROP: Render Output Unit

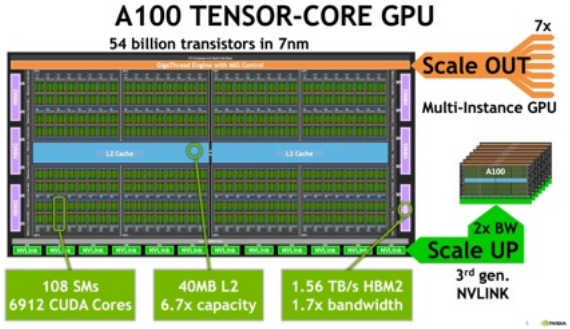
RTX 5090



A100 TENSOR-CORE GPU

54 billion transistors in 7nm

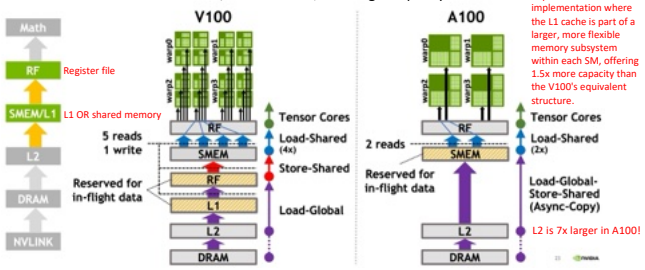
NVIDIA's **Multi-Instance GPU (MIG)** allows a single GPU to be partitioned into multiple, independent GPU instances, enabling multiple users to run workloads with isolated resources



V100 A100

A100 data movement efficiency

3x SMEM/L1 bandwidth, 2x in-flight capacity



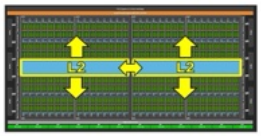
The A100 features an improved implementation where the L1 cache is part of a larger, more flexible memory subsystem within each SM, offering 1.5x more capacity than the V100's equivalent structure.

L2 is 7x larger in A100!

in-flight data = data that is actively being processed or transferred within the system but hasn't yet reached its final destination.

A100 data movement efficiency

Split L2 with hierarchical crossbar - 2.3x increase in bandwidth over V100, lower latency

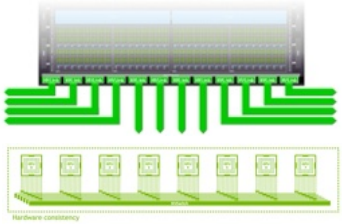


Third Generation NVLink

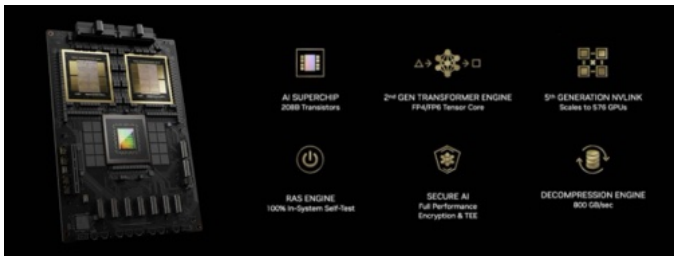
50 Gbit/sec per signal pair

12 links, 25 GB/s in/out, 600 GB/s total

2x vs. V100



GB100 $40k, GB200: $70k



RAS (Reliability, Availability, and Serviceability)

Datacenter

Die	GB100
Variant(s)	
Release date	Dec 2024
Cores: CUDA Cores	
Cores: TMUs	
Cores: ROPs	
Cores: RT Cores	
Cores: Tensor Cores	
Streaming Multiprocessors	
Cache: L1	
Cache: L2	
Memory interface	8192-bit
Die size	
Transistor count	104 bn.
Transistor density	
Package socket	SXM6
Products	B100 B200

Consumer

Die	GB202	GB203	GB205	GB206	GB207
Variant(s)	GB202-300-A1 GB202-400-A1	GB203-200-A1 GB203-300-A1 GB203-400-A1	GB205-300-A1		
Release date	Jan 30, 2025	Jan 30, 2025	Mar 4, 2025	Mar 2025	TBA
Cores: CUDA Cores	24,576	10,752	8,400	4,608	3,840
Cores: TMUs	768	336	200	144	60
Cores: ROPs	192	112	80	48	32
Cores: RT Cores	192	84	50	36	20
Cores: Tensor Cores	768	336	200	144	60
SMs	192	84	50	36	20
GPCs	12	7	5	3	2
Cache: L1	24 MB	10.5 MB	6.25 MB	4.5 MB	2.5 MB
Cache: L2	128 MB	64 MB	48 MB	32 MB	32 MB
Memory interface	512-bit	256-bit	160-bit	128-bit	128-bit
Die size	750 mm²	378 mm²	263 mm²		
Transistor count	92.2 bn.	45.6 bn.	31.1 bn.		
Transistor density	122.9 M/mm²	120.6 M/mm²	118.3 M/mm²		
Products					
Consumer: Desktop	RTX 5090 RTX 5090 D	RTX 5070 Ti RTX 5080	RTX 5070		
Consumer: Mobile	RTX 5090 Laptop RTX 5080 Laptop		RTX 5070 Ti Laptop	RTX 5070 Laptop	
Workstation: Desktop	RTX PRO 6000 RTX PRO 5000	RTX PRO 4500 RTX PRO 4000			
Workstation: Mobile	RTX PRO 5000 Mobile	RTX PRO 4000 Mobile RTX PRO 3500 Mobile	RTX PRO 3000 Mobile	RTX PRO 2000 Mobile	RTX PRO 500 Mobile RTX PRO 1000 Mobile
Server	RTX PRO 6000				

CUDA

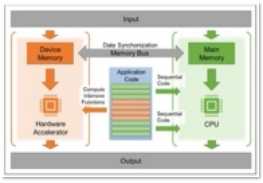
- **CUDA** (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA.
- It enables developers to use NVIDIA graphics processing units (GPUs) for general-purpose computing, a practice known as **GPGPU** (General-Purpose computing on Graphics Processing Units).
- CUDA programs to be written without knowing the exact hardware configuration in advance.

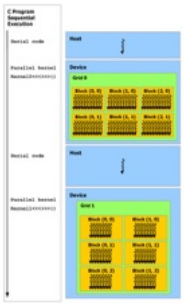
NVIDIA CUDA

- Good resource: https://docs.nvidia.com/cuda/cuda-c-programming-guide

Programming and execution model

A GPU is built around an **array of Streaming Multiprocessors (SMs). A multithreaded program partitioned into blocks of threads that execute independently from each other**, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.





CUDA programming model assumes that the **CUDA threads execute on a physically separate device** that operates as a coprocessor to the host running the C++ program.

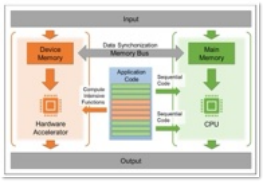
Serial code executes on the host while parallel code executes on the device.

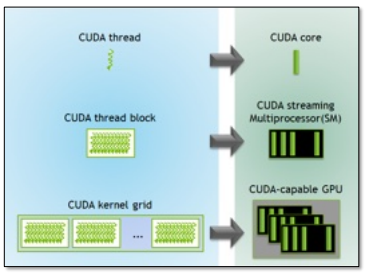
Some terminology

- A **warp** in CUDA is a fundamental execution unit consisting of 32 threads that execute in lockstep. All 32 threads in a warp execute the same instruction at the same time on different data (**SIMD**: Single Instruction, Data).
- A **CUDA kernel** is a function that runs on the GPU.
- A **block** in CUDA (also called a **thread block**) is a fundamental organizational unit of threads that execute a kernel function.
- The number **of threads per block** is limited (typically 1024 threads in modern GPUs), as they must reside on the same streaming multiprocessor (**SM**).
- **Thread blocks are composed of multiple warps**. For example, a block of 128 threads would consist of 4 warps.
- In CUDA programming, a **kernel grid** is the highest-level organization of threads when executing a CUDA kernel.

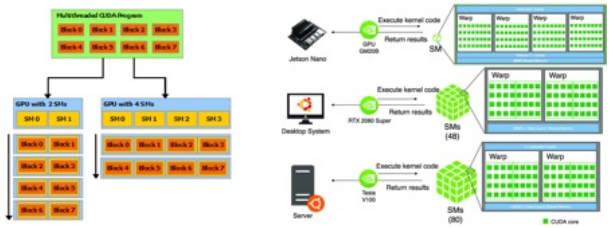
Programming and execution model

A GPU is built around an **array of Streaming Multiprocessors (SMs). A multithreaded program is partitioned into blocks of threads that execute independently from each other**, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.





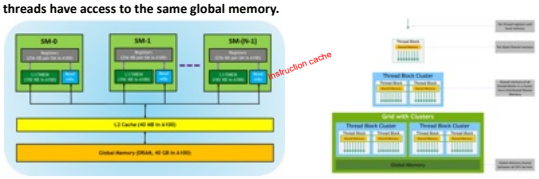
Automatic scalability



A GPU is built around an **array of Streaming Multiprocessors (SMs). A multithreaded program is partitioned into blocks of threads that execute independently from each other**, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

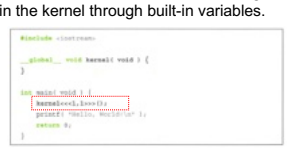
Memory hierarchy

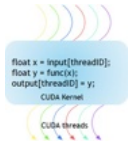
- CUDA threads may access data from multiple memory spaces during their execution.
- **Each thread has private local memory.**
- **Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block.**
- Thread blocks in a thread block cluster can perform read, write, and atomics operations on each other's shared memory.
- **All threads have access to the same global memory.**



Kernels (1)

- CUDA C++ extends C++ by allowing the programmer to define C++ functions, called **kernels**, that, when called, **are executed *N* times in parallel by *N* different CUDA threads, as opposed to only once like regular C++ functions**.
- A kernel is defined using the **__global__ declaration specifier** and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new **<<<...>>> execution configuration** syntax.
- Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through built-in variables.





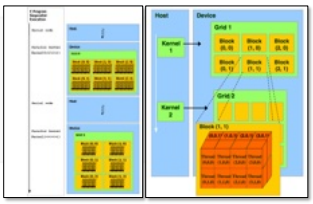
https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model

Kernels (2)

- As an illustration, the following sample code, using the built-in variable threadIdx, adds two vectors A and B of size N and stores the result into vector C.
- Single Instruction, Multiple Threads (SIMT)

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

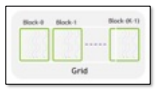
Here, each of the *N* threads that execute `VecAdd()` performs one pair-wise addition.



Kernels (3)

- **`kernel<<M,T>>`**
- The kernel launches with a grid of M thread blocks. Each thread block has T parallel threads.

- The information between the triple chevrons is the execution configuration, which dictates how many device threads execute the kernel in parallel.
- In the CUDA programming model, we speak of **launching a kernel with a grid of thread blocks**.
- The first argument in the execution configuration specifies the number of thread blocks M in the grid, and the second specifies the number of threads T in a thread block.

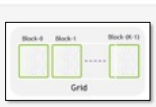


https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model

Kernels (4)

The CUDA program for adding two matrices below shows multi-dimensional `blockIdx` and `threadIdx` and other variables like `blockDim`. In the example below, a 2D block is chosen for ease of indexing and each block has 256 threads with 16 each in x and y-direction. The total number of blocks are computed using the data size divided by the size of each block.

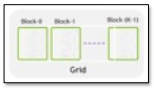
```
// Kernel - Adding two matrices MatA and MatB
__global__ void MatAdd(float MatA[N][N], float MatB[N][N],
float MatC[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        MatC[i][j] = MatA[i][j] + MatB[i][j];
}

int main()
{
    ...
    // Matrix addition kernel launch from host code
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((N + threadsPerBlock.x -1) / threadsPerBlock.x, (N+threadsPerBlock.y -1) / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(MatA, MatB, MatC);
    ...
}
```



https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model

Kernels (4)

What happens if you ask for more blocks than are physically available on the GPU?



https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model

Kernels (4)

What happens if you ask for more blocks than are physically available on the GPU?

- The GPU doesn't reject your kernel launch. Instead, it schedules the execution of blocks over time. This is a key aspect of CUDA's execution model.
- The GPU hardware scheduler automatically queues the blocks and executes them as resources become available.
- Blocks waiting in the queue don't make progress until scheduled
- The total execution time will increase as blocks must be processed sequentially
- Memory for all blocks must still be allocated, potentially creating memory pressure.
- GPUs have a metric called "**occupancy**" which represents how many threads can be active simultaneously relative to the theoretical maximum. The scheduler tries to maximize occupancy based on resource availability.
- CUDA programs can be written without knowing the exact hardware configuration.

https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model

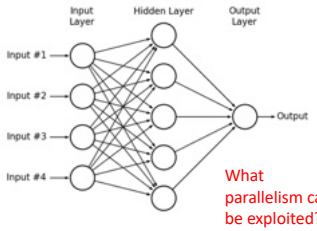
Another example

SAXPY stands for "Single-precision A*X Plus Y"

https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c

How to map a neural network on a GPU?

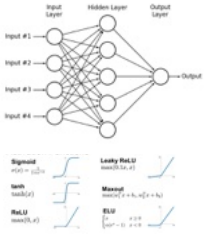
Multi-layer perceptron (MLP)



What parallelism can be exploited?

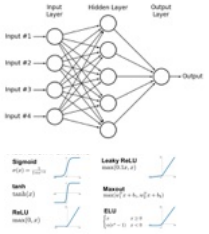
Multi-layer perceptron (MLP)

- **Data parallelism**: You can process multiple training examples simultaneously. Since each example is processed independently through the forward pass, you can distribute batches across different computing units.
- **Model parallelism**: The network itself can be partitioned, with different parts of the model assigned to different processors. This works particularly well for very large networks.
- **Layer-wise parallelism**: Within each layer, the computations can be parallelized since neurons in the same layer operate independently.

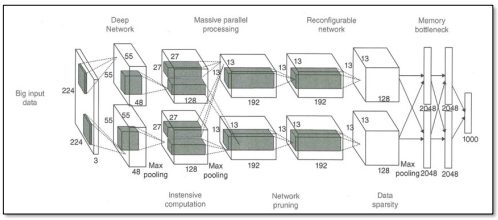


Multi-layer perceptron (MLP)

- **Matrix multiplications**: The core operation in MLPs (Wx + b) is highly parallelizable. Each output neuron's computation is independent.
- **Activation functions**: These are applied element-wise and can be computed in parallel across all neurons.
- **Backpropagation**: During training, gradient computations can be parallelized both within and across layers.



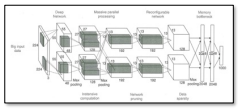
AlexNet: A classical deep convolutional neural network

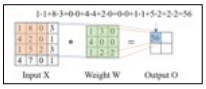


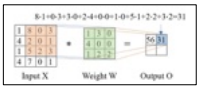
AlexNet paper: https://papers.nips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

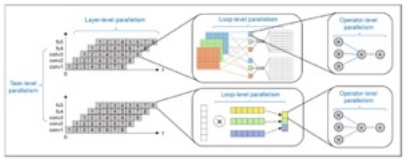
Liu & Law, Artificial Intelligence Hardware Design, 2021

Exploiting parallelism







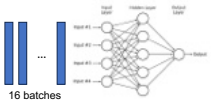


CUDA MLP (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <time.h>

// Dimensions of our network
#define INPUT_SIZE 4
#define HIDDEN_SIZE 5
#define OUTPUT_SIZE 1
#define BATCH_SIZE 16  // Process 16 samples at once

// Activation function (ReLU)
__device__ float relu(float x) {
    return x > 0 ? x : 0;
}
```

ReLU

$\max(0, x)$



16 batches

CUDA C++ Language Extensions

7.1. Function Execution Space Specifiers

Function execution space specifiers denote whether a function executes on the host or on the device and whether it is callable from the host or from the device.

7.1.1. __global__

The __global__ execution space specifier declares a function as being a kernel. Such a function is:

- Executed on the device,
- Callable from the host,
- Callable from the device for devices of compute capability 5.0 or higher (see CUDA Dynamic Parallelism for more details).

A __global__ function must have void return type, and cannot be a member of a class.

Any call to a __global__ function must specify its execution configuration as described in Execution Configuration.

A call to a __global__ function is asynchronous, meaning it returns before the device has completed its execution.

7.1.2. __device__

The __device__ execution space specifier declares a function that is:

- Executed on the device,
- Callable from the device only.

The __global__ and __device__ execution space specifiers cannot be used together.

7.1.3. __host__

The __host__ execution space specifier declares a function that is:

- Executed on the host,
- Callable from the host only.

https://docs.nvidia.com/cuda/cuda-c-programming-guide

CUDA MLP (2)

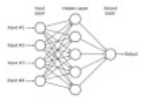
```
// Forward pass kernel for first layer (input -> hidden)
__global__ void layer1_forward(float *input, float *weights, float *bias, float *output, int batch_size) {
    int batch_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int neuron_idx = blockIdx.y * blockDim.y + threadIdx.y;

    if (batch_idx < batch_size && neuron_idx < HIDDEN_SIZE) {
        float sum = bias[neuron_idx];
        for (int i = 0; i < INPUT_SIZE; i++) {
            sum += input[batch_idx * INPUT_SIZE + i] * weights[i * HIDDEN_SIZE + neuron_idx];
        }
        output[batch_idx * HIDDEN_SIZE + neuron_idx] = relu(sum);
    }
}

// Forward pass kernel for second layer (hidden -> output)
__global__ void layer2_forward(float *input, float *weights, float *bias, float *output, int batch_size) {
    int batch_idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (batch_idx < batch_size) {
        float sum = bias[0];
        for (int i = 0; i < HIDDEN_SIZE; i++) {
            sum += input[batch_idx * HIDDEN_SIZE + i] * weights[i];
        }
        output[batch_idx] = sum;  // No activation on output layer in this example
    }
}
```

Global thread ID (x coordinate)
Global y coordinate

CUDA MLP (3)

Why separate kernels for each layer?

- **Memory optimization**: Each layer has different memory access patterns and working set sizes. Separate kernels allow you to optimize memory usage specifically for each layer's requirements rather than trying to find a one-size-fits-all approach.
- **Parallelism control**: Different layers may benefit from different thread configurations. For example, the first layer in our implementation handles 4 inputs to 5 neurons, while the output layer handles 5 inputs to 1 neuron - these naturally map to different parallelization strategies.
- **Resource utilization**: GPU resources like shared memory, registers, and thread counts can be tailored specifically to each layer's computational needs, which improves overall efficiency.
- **Kernel fusion limitations**: While fusing operations into a single kernel can reduce launch overhead, there are practical limits to how much computation can be packed into one kernel before it becomes inefficient. Separate kernels help manage this complexity.
- **Synchronization points**: Having distinct kernels creates natural synchronization points between layers, ensuring all computations from one layer are complete before the next layer begins processing.
- **Debugging and profiling**: With separate kernels, it's easier to identify performance bottlenecks or errors in specific parts of the network.
- **Specialized optimizations**: Different mathematical operations may benefit from specialized implementations - convolutions, fully-connected layers, and activation functions all have different optimal implementations on GPUs.



CUDA MLP (4)

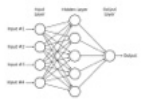
How much should be in a kernel?

The optimal kernel size involves balancing several trade-offs:

- **Kernel launch overhead**: Each kernel launch incurs some overhead. Too many small kernels can result in performance degradation due to this overhead.
- **Register pressure**: Larger kernels with more operations typically require more registers per thread, which can reduce occupancy (the number of threads that can run simultaneously on the GPU).
- **Instruction cache**: GPUs have limited instruction cache. Very large kernels might not fit well in the cache, causing additional latency.
- **Memory bandwidth utilization**: Kernels that load data, perform a small operation, and then store results can be memory-bound. Combining multiple operations in one kernel can improve arithmetic intensity and make better use of compute resources.
- **Thread divergence**: If your kernel contains many conditional branches that cause different threads to follow different execution paths, performance will suffer.

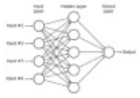
As a general guideline:

- **Good candidates for fusion**: Operations that work on the same data and have similar parallelization patterns (like a matrix multiplication followed by an element-wise activation function).
- **Better kept separate**: Operations with fundamentally different access patterns or parallelization strategies.



CUDA MLP (5)

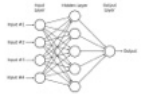
```cpp
// Initialize weights with random values between -0.5 and 0.5
void initialize_weights(float *h_weights, int size) {
    for (int i = 0; i < size; i++) {
        h_weights[i] = ((float)rand() / RAND_MAX) - 0.5f;
    }
}

int main() {
    // Seed random number generator
    srand(time(NULL));

    // Host memory allocation
    float h_input[BATCH_SIZE * INPUT_SIZE];
    float h_hidden[BATCH_SIZE * HIDDEN_SIZE];
    float h_output[BATCH_SIZE * OUTPUT_SIZE];
    float h_weights1[INPUT_SIZE * HIDDEN_SIZE];
    float h_bias1[HIDDEN_SIZE];
    float h_weights2[HIDDEN_SIZE * OUTPUT_SIZE];
    float h_bias2[OUTPUT_SIZE];

    // Initialize weights and biases
    initialize_weights(h_weights1, INPUT_SIZE * HIDDEN_SIZE);
    initialize_weights(h_bias1, HIDDEN_SIZE);
    initialize_weights(h_weights2, HIDDEN_SIZE * OUTPUT_SIZE);
    initialize_weights(h_bias2, OUTPUT_SIZE);

    // Initialize some example input data
    for (int i = 0; i < BATCH_SIZE * INPUT_SIZE; i++) {
        h_input[i] = ((float)rand() / RAND_MAX);
    }
```



CUDA MLP (6)

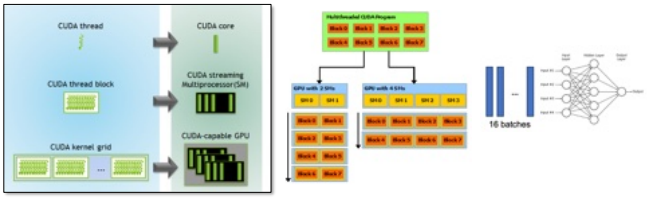
```cpp
// Host memory allocation
float h_input[BATCH_SIZE * INPUT_SIZE];
float h_hidden[BATCH_SIZE * HIDDEN_SIZE];
float h_output[BATCH_SIZE * OUTPUT_SIZE];
float h_weights1[INPUT_SIZE * HIDDEN_SIZE];
float h_bias1[HIDDEN_SIZE];
float h_weights2[HIDDEN_SIZE * OUTPUT_SIZE];
float h_bias2[OUTPUT_SIZE];
```

```cpp
// Device memory allocation
float *d_input, *d_hidden, *d_output;
float *d_weights1, *d_bias1, *d_weights2, *d_bias2;

cudaMalloc(&d_input, BATCH_SIZE * INPUT_SIZE * sizeof(float));
cudaMalloc(&d_hidden, BATCH_SIZE * HIDDEN_SIZE * sizeof(float));
cudaMalloc(&d_output, BATCH_SIZE * OUTPUT_SIZE * sizeof(float));
cudaMalloc(&d_weights1, INPUT_SIZE * HIDDEN_SIZE * sizeof(float));
cudaMalloc(&d_bias1, HIDDEN_SIZE * sizeof(float));
cudaMalloc(&d_weights2, HIDDEN_SIZE * OUTPUT_SIZE * sizeof(float));
cudaMalloc(&d_bias2, OUTPUT_SIZE * sizeof(float));

// Copy data from host to device
cudaMemcpy(d_input, h_input, BATCH_SIZE * INPUT_SIZE * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_weights1, h_weights1, INPUT_SIZE * HIDDEN_SIZE * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_bias1, h_bias1, HIDDEN_SIZE * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_weights2, h_weights2, HIDDEN_SIZE * OUTPUT_SIZE * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_bias2, h_bias2, OUTPUT_SIZE * sizeof(float), cudaMemcpyHostToDevice);
```



CUDA MLP (7)

How will we use threads for this MLP?



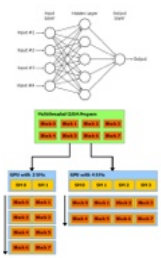
A GPU is built around an **array of Streaming Multiprocessors (SMs). A multithreaded program is partitioned into blocks of threads that execute independently from each other**.

CUDA MLP (8)

```cpp
// Set up execution configuration for first layer
dim3 threadsPerBlock1(8, 5);  // 8 batches, all 5 hidden neurons
dim3 blocksPerGrid1((BATCH_SIZE + threadsPerBlock1.x - 1) / threadsPerBlock1.x,
                    (HIDDEN_SIZE + threadsPerBlock1.y - 1) / threadsPerBlock1.y);

// Set up execution configuration for second layer
dim3 threadsPerBlock2(16, 1);  // 16 batches
dim3 blocksPerGrid2((BATCH_SIZE + threadsPerBlock2.x - 1) / threadsPerBlock2.x, 1);

// Launch kernels
layer1_forward<<<blocksPerGrid1, threadsPerBlock1>>>(d_input, d_weights1, d_bias1, d_hidden, BATCH_SIZE);
layer2_forward<<<blocksPerGrid2, threadsPerBlock2>>>(d_hidden, d_weights2, d_bias2, d_output, BATCH_SIZE);

// Check for errors
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
    printf("CUDA Error: %s\n", cudaGetErrorString(err));
    return -1;
}

// Copy results back to host
cudaMemcpy(h_output, d_output, BATCH_SIZE * OUTPUT_SIZE * sizeof(float), cudaMemcpyDeviceToHost);

// Print some outputs
printf("Sample outputs from MLP:\n");
for (int i = 0; i < 5; i++) {  // Print first 5 results
    printf("Sample %d: %.6f\n", i, h_output[i]);
}
```

Why?

kernel<<M,T>>
The kernel launches with a grid of M thread blocks.
Each thread block has T parallel threads.



A GPU is built around an **array of Streaming Multiprocessors (SMs). A multithreaded program is partitioned into blocks of threads that execute independently from each other**.

CUDA MLP (9)

```cpp
// Free GPU memory
cudaFree(d_input);
cudaFree(d_hidden);
cudaFree(d_output);
cudaFree(d_weights1);
cudaFree(d_bias1);
cudaFree(d_weights2);
cudaFree(d_bias2);
```