Introduction to Parallel Programming using CUDA

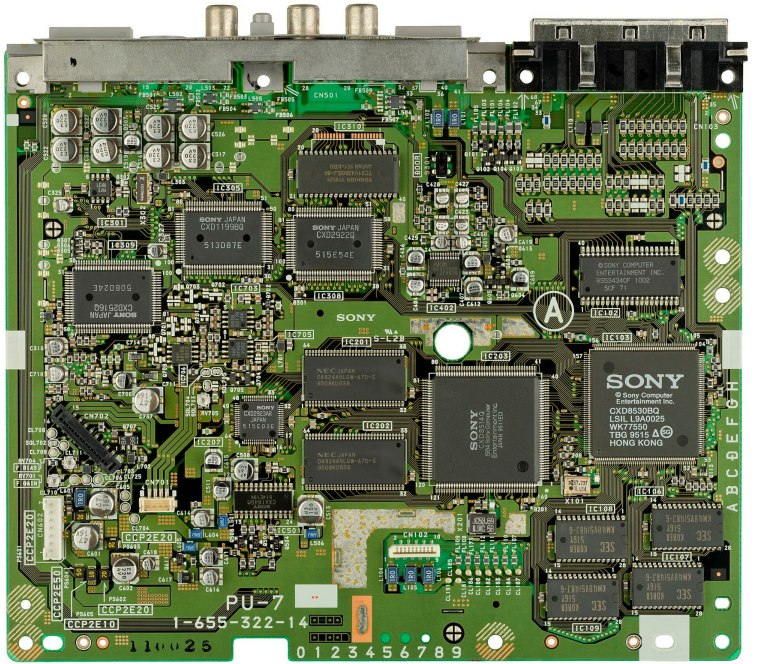
T. Ranjit
14.02.25

1. What is a GPU Exactly?

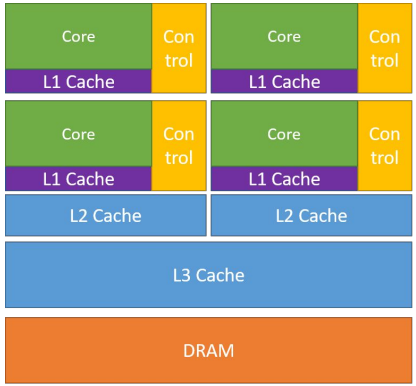
What is a GPU?

A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly render graphics and images by performing parallel mathematical calculations. Originally developed for computer graphics and image processing, GPUs have evolved to handle a wider range of applications.

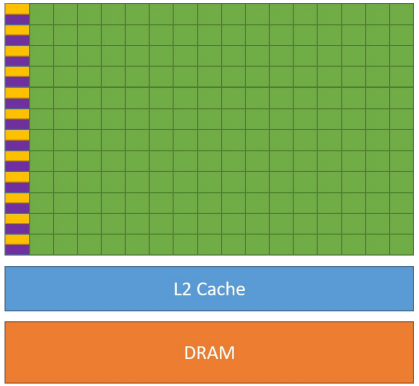
The term "GPU" was coined by Sony in reference to the 32-bit Sony GPU (designed by Toshiba) in the PlayStation video game console, released in 1994.



What makes a GPU different from a CPU



CPU



GPU

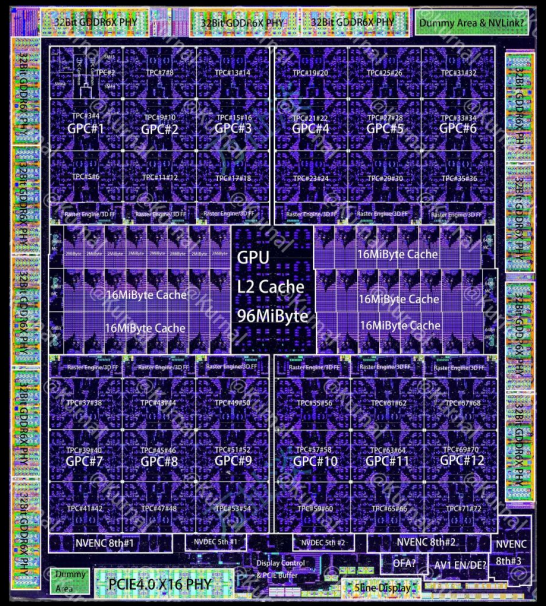
GPUs are composed of hundreds of cores that can handle thousands of threads simultaneously, while CPUs have just a few cores optimized for serial processing.

GPUs also higher compute density when compared to a traditional CPU.

This architectural difference allows GPUs to excel at parallel processing tasks

RTX 5090 DIE

92 Billion Transistors!!!

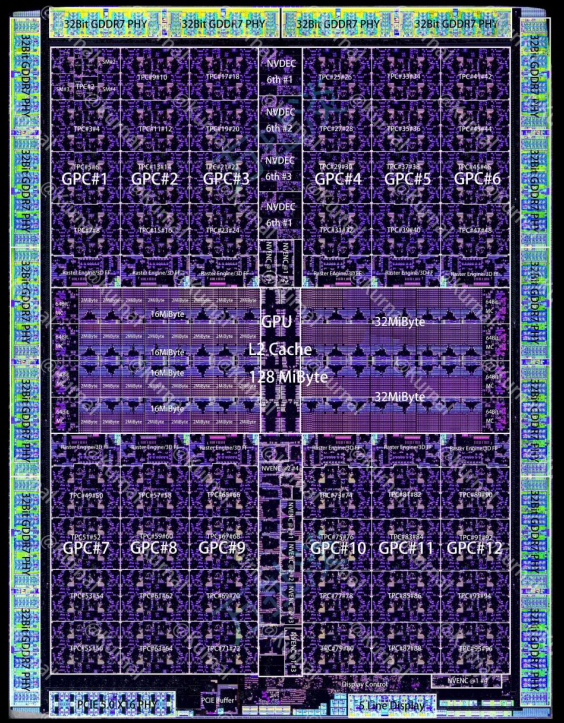


AD102 23.37mm x 26.36mm

Chip By @ASUS Tony 俞元麟

Dieshot By @万扯淡

Layout By @Kurnal



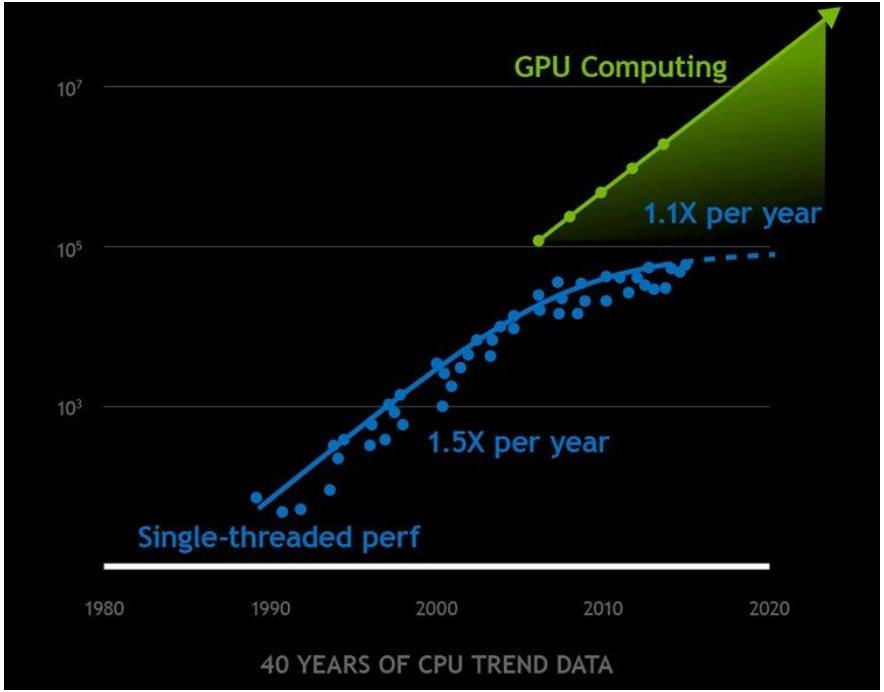
GB202 24.10mm x 31.60mm

2. What is Parallel Programming?

What is Parallel Programming?

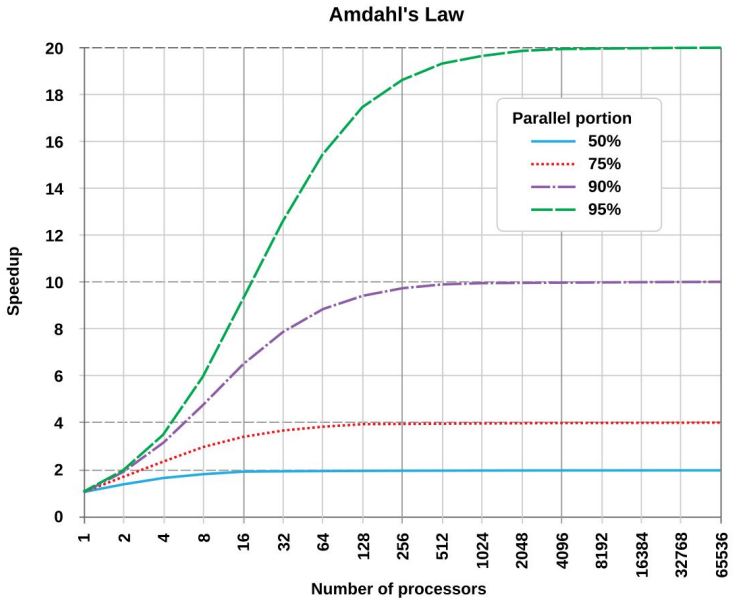
Parallel programming is the process of breaking down a computational problem into smaller tasks that can be executed simultaneously using multiple compute resources. It allows a computer to use multiple processors or cores to solve problems concurrently, improving efficiency and speed.

But why?



One answer: Moore's Law

Amdahl's Law



"the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used"

$$\text{Overall Speedup} = \frac{\text{Old execution time}}{\text{New execution time}}$$

$$= \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}} \right) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

3. What is CUDA?

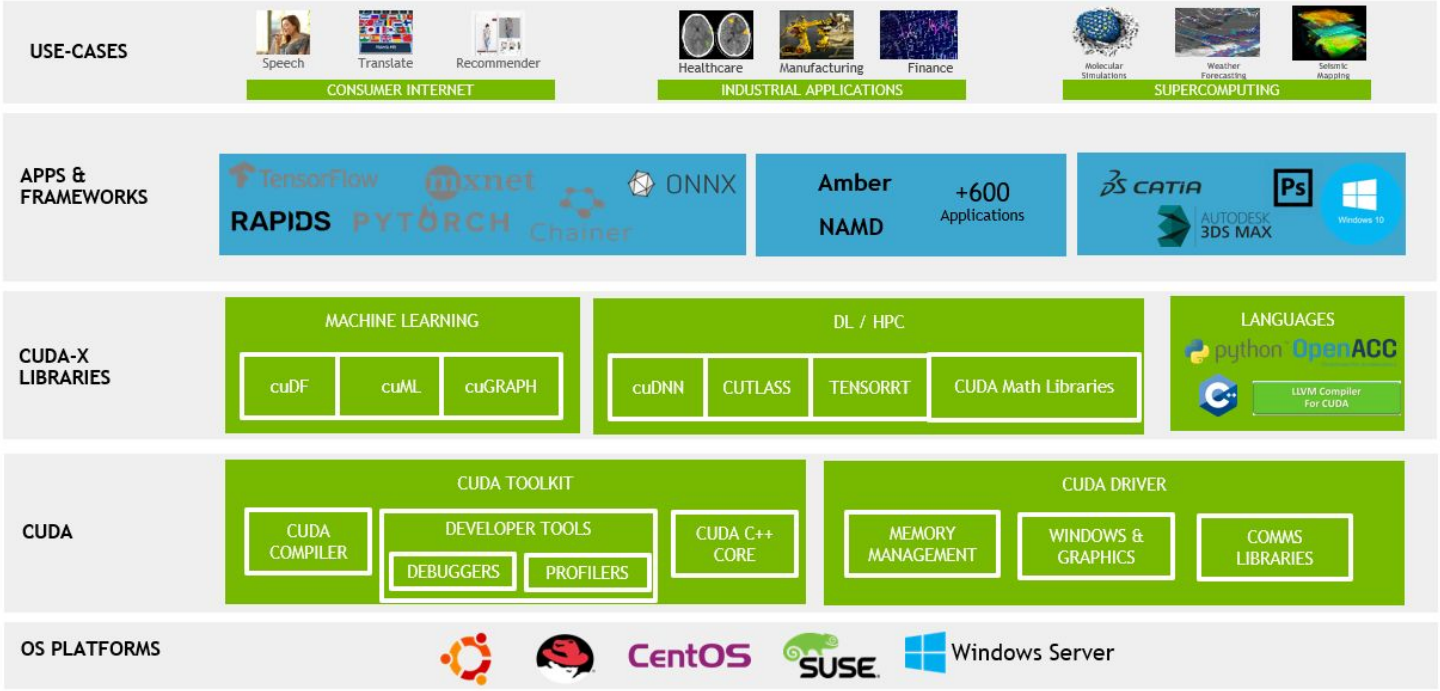
What is CUDA?

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA in 2006. It allows software developers to use NVIDIA GPUs for general-purpose processing, enabling significant speedups for computationally intensive tasks.

Key features of CUDA include:

1. Direct access to the GPU's virtual instruction set and parallel computational elements
2. Support for programming languages like C, C++, Fortran, and Python
3. A software environment that includes libraries, debugging tools, and a runtime library
4. Ability to dramatically accelerate computing applications by leveraging GPU power

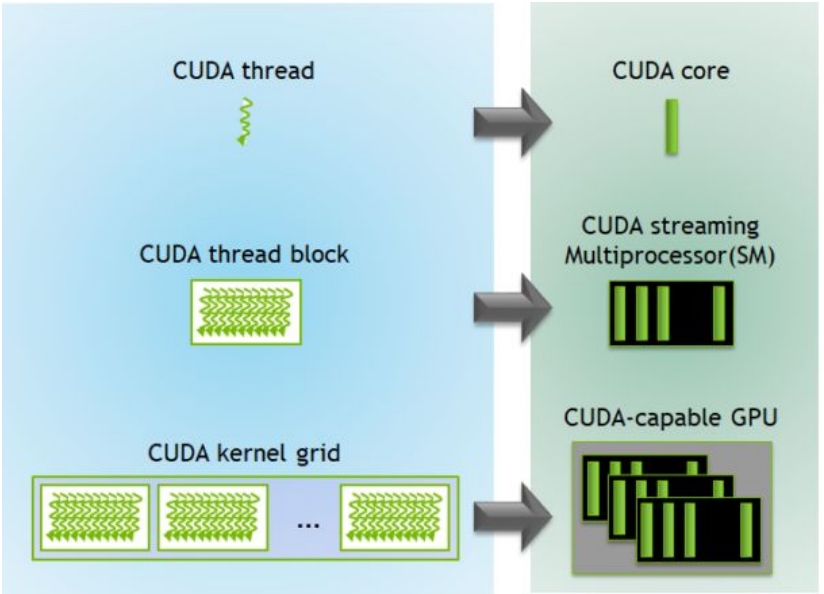
What is CUDA


USE-CASES
Speech
Translate
Recommender
CONSUMER INTERNET
Healthcare
Manufacturing
Finance
INDUSTRIAL APPLICATIONS
Molecular Simulations
Weather Forecasting
Seismic Mapping
SUPERCOMPUTING
APPS & FRAMEWORKS
TensorFlow
mxnet
ONNX
RAPIDS
PYTORCH
Chainer
Amber
NAMD
+600 Applications
CATIA
Ps
AUTODESK 3DS MAX
Windows 10
CUDA-X LIBRARIES
MACHINE LEARNING
cuDF
cuML
cuGRAPH
DL / HPC
cuDNN
CUTLASS
TENSORRT
CUDA Math Libraries
LANGUAGES
python
OpenACC
LLVM Compiler For CUDA
CUDA
CUDA TOOLKIT
CUDA COMPILER
DEVELOPER TOOLS
DEBUGGERS
PROFILERS
CUDA C++ CORE
CUDA DRIVER
MEMORY MANAGEMENT
WINDOWS & GRAPHICS
COMMS LIBRARIES
OS PLATFORMS
CentOS
SUSE
Windows Server

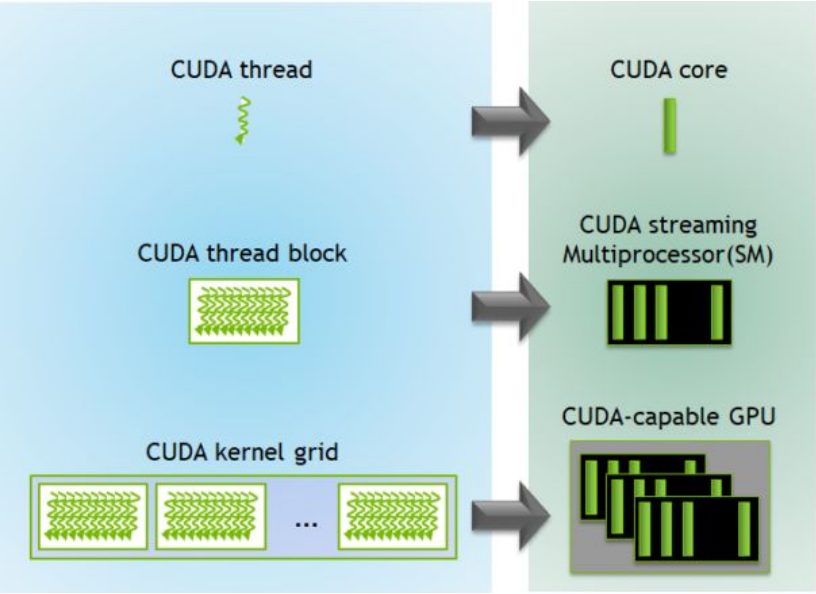
Threads, Blocks and Grids

The basic unit of parallel execution in CUDA. Each thread runs the same kernel function independently.

Threads are organized into groups called blocks. Number of threads per block can be defined by the user. A group of threads that can cooperate and share memory.



Threads, Blocks and Grids

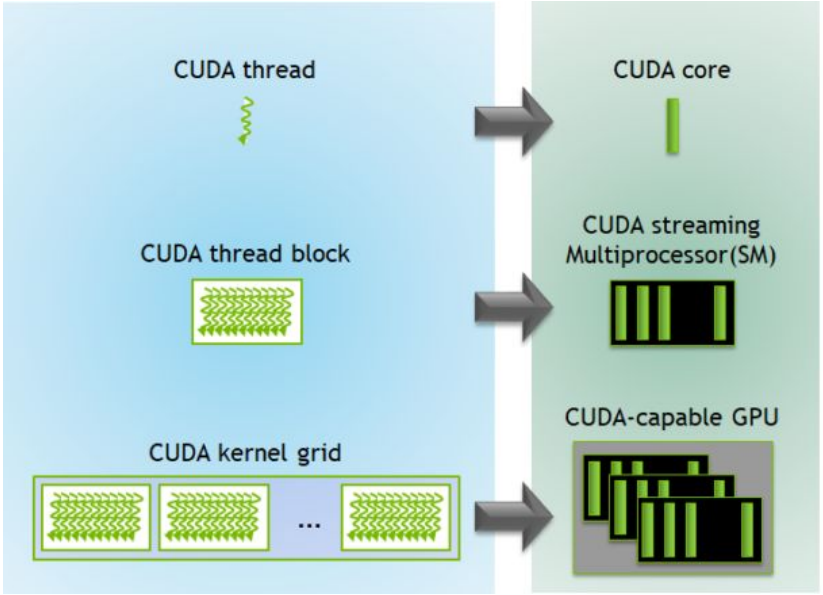


Each thread is assigned to a SM and cannot migrate. Each SM has 128 cuda cores, therefore, at any time 128 threads can run in parallel per block. So what if you have more threadblocks than fit on the SMs (as is frequently the case). They won't all run simultaneously. When a thread block completes the hardware will put on that's ready to execute onto the SM where the completed one was running.

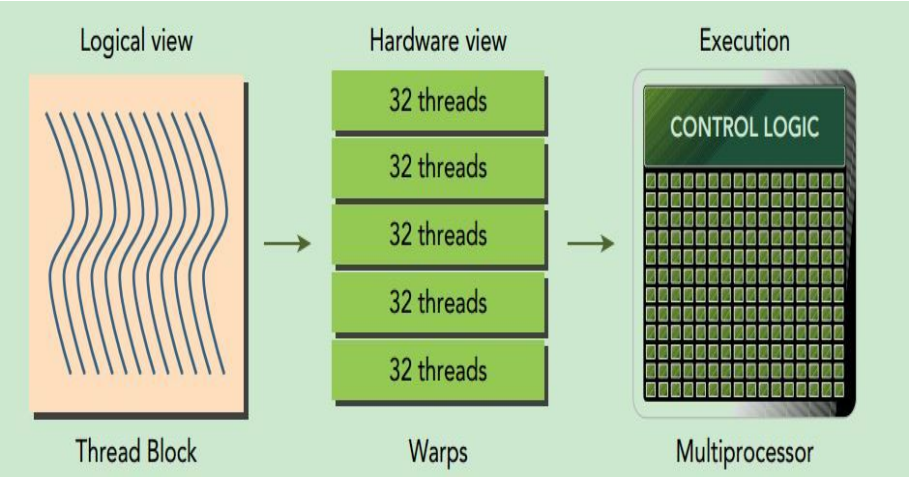
Threads, Blocks and Grids

Grids in CUDA are an essential part of the thread hierarchy, organizing blocks of threads for parallel execution on GPUs. They are essentially a superset of all the Thread Blocks. They allow you to scale your program to fit any GPU.

All threads in a grid have access to the same global memory.



Warps



A Warp is a group of 32 threads that execute instructions simultaneously. All threads in a warp execute the same instruction at the same time, following the SIMD (Single Instruction, Multiple Data) model. Each SM has a specific number of CUDA cores, allowing multiple warps to run simultaneously. For example, an SM with 128 CUDA cores can run 4 warps simultaneously.

4. Programming in CUDA

Kernel functions: global, device and host functions

Kernel functions: global, device, host keywords:

- global: Defines a kernel function that runs on the GPU and is callable from the CPU
- device: Defines a function that runs on the GPU and is only callable from the GPU
- host: Defines a function that runs on the CPU (default for functions)

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

Function call example code

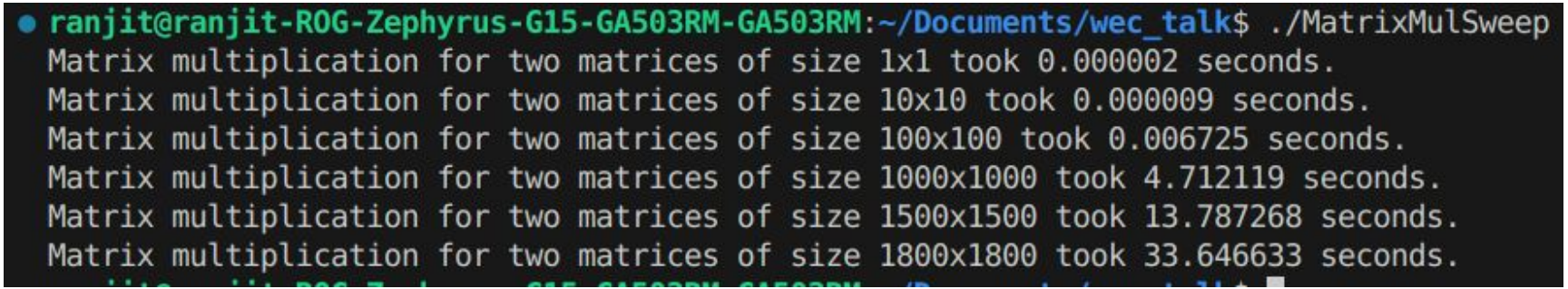
```
// Calculating required no of Blocks
int THREADS = 1024;
int BLOCKS = (N+THREADS-1)/THREADS;

// Calling the device function
ArrayAdd <<<BLOCKS,THREADS>>> (gpu1,gpu2,gpuresult,N);
cudaDeviceSynchronize();
```

Code Example: ArrayAdd

```
Array addition of 2 arrays of size 1 took 0.000003 seconds.
Array addition of 2 arrays of size 10 took 0.000001 seconds.
Array addition of 2 arrays of size 100 took 0.000002 seconds.
Array addition of 2 arrays of size 1000 took 0.000005 seconds.
Array addition of 2 arrays of size 10000 took 0.000064 seconds.
Array addition of 2 arrays of size 100000 took 0.000899 seconds.
Array addition of 2 arrays of size 1000000 took 0.007813 seconds.
Array addition of 2 arrays of size 10000000 took 0.075194 seconds.
Array addition of 2 arrays of size 100000000 took 0.402498 seconds.
Killed
```

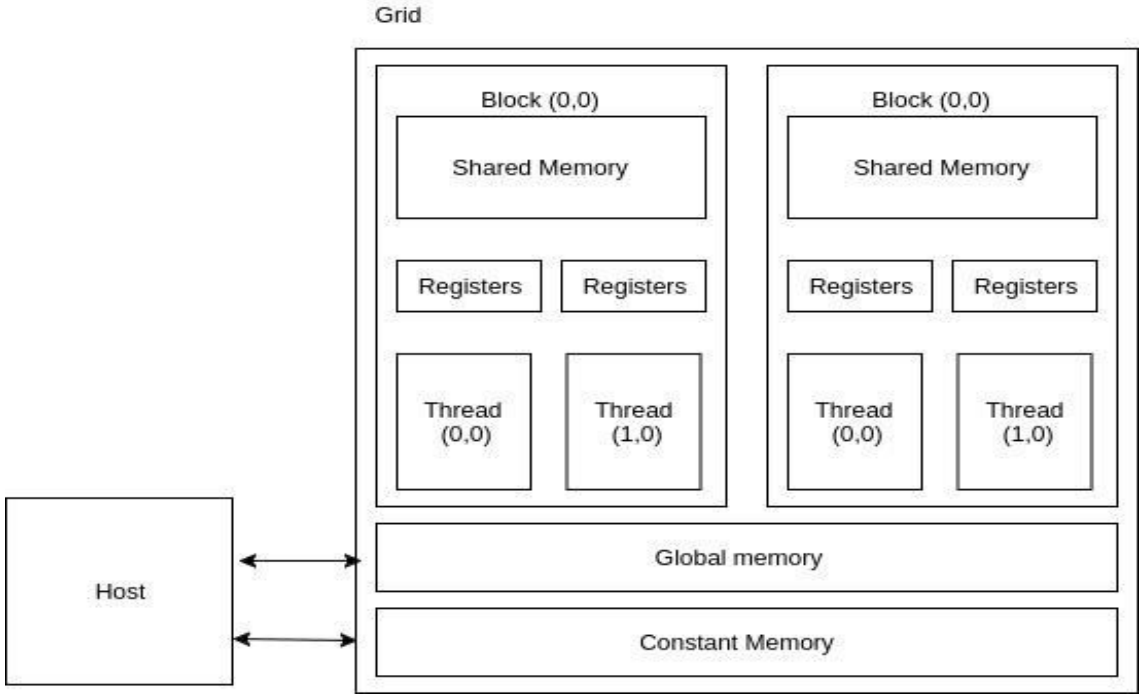
Code Example: Matrix Multiplication



Code Example: Parallel Reductions and Race conditions

5. Memory Management in CUDA

Memory in Cuda



Variables in CUDA

Table 5.1 CUDA Variable Type Qualifiers

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__, __shared__, int SharedVar;`	Shared	Block	Kernel
`__device__, int GlobalVar;`	Global	Grid	Application
`__device__, __constant__, int ConstVar;`	Constant	Grid	Application

cudaMalloc()

This function allocates memory on the GPU device.

cudaError_t cudaMalloc(void** pointer, int bytes)

It takes two parameters:

1. A pointer to a pointer where the allocated memory address will be stored
2. The number of bytes to allocate

cudaMalloc() creates memory on the GPU and returns a handle to it in the provided pointer

cudaMemcpy()

This function copies data between host and device memory.

cudaError_t cudaMemcpy(void* destination, void* source, int bytes, cudaMemcpyKind kind)

It takes four parameters:

1. Destination pointer
2. Source pointer
3. Number of bytes to copy
4. Direction of the copy (e.g., cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost)

6. Limitations and Alternatives

Limitations

1. Memory constraints: GPUs have limited memory compared to CPUs, which can be problematic for large datasets.
2. Hardware dependency: CUDA only works on NVIDIA GPUs, limiting its portability.
3. Programming complexity: CUDA requires a steep learning curve and specific programming approaches, making it challenging for developers new to GPU computing.
4. Limited exception handling: While exceptions can be thrown on the device, try-catch blocks cannot be used, and backtrace printing is costly.
5. Data transfer bottlenecks: Moving data between CPU and GPU memory can be slow, impacting overall performance.

Alternatives: CPU Parallelism

OpenMP:

- Shared-memory parallelism
- Uses compiler directives (pragmas)
- Fork-join model
- Easier to implement
- Limited to single multi-core system

MPI:

- Distributed-memory parallelism
- Message passing between processes
- Explicit communication programming
- More complex, but highly scalable
- Can run across multiple networked computers

Alternatives: OpenCL

OpenCL (Open Computing Language) is a framework for writing parallel programs that can execute across heterogeneous computing platforms, including CPUs, GPUs, DSPs, and FPGAs. It provides a standard interface for parallel computing using both task-based and data-based parallelism.

It is an open-source framework that was developed by collaboration between industry behemoths like Apple, AMD, Intel, IBM, Qualcomm and Nvidia.

CUDA often provides better performance on NVIDIA GPUs due to its closer integration with the hardware architecture.

CUDA also has a more established ecosystem with comprehensive tools and libraries specifically for NVIDIA GPUs, while OpenCL's ecosystem is broader but may not be as specialized.

Alternatives: ROCm

AMD's open source alternative to CUDA.

Open source but only runs on AMD.

Launched in 2016.

Currently integrated into a few libraries like pyTorch and Tensorflow, but hasn't gained much traction due to the rarity of AMD GPUs in the B2B sector. However, it is slowly gaining popularity now along with AMD GPUs

7. CUDA in AI

Questions?

Thank you.