Syllabus

Unit 5A: Introduction to High Performance Computing

Goals of parallel computing, Speed of parallelism, CPU vs High Performance Computing, Architecture of a modern GPU, Parallel Programming Languages and models, GPU Computing, Data parallelism

INDEX

5A.1 Parallel Computing:

"Parallel processing (Computing) is a computing technique when multiple streams of calculations or data processing tasks co-occur through numerous central processing units (CPUs) working concurrently"

It is the use of multiple processing elements simultaneously for solving any problem. Problems are broken down into instructions and are solved concurrently as each resource that has been applied to work is working at the same time.

Why parallel computing?

- The whole real-world runs in dynamic nature i.e. many things happen at a certain time but at different places concurrently. This data is extensively huge to manage.
- Real-world data needs more dynamic simulation and modeling, and for achieving the same, parallel computing is the key.
- Parallel computing provides concurrency and saves time and money.
- Complex, large datasets, and their management can be organized only and only using parallel computing's approach.
- Ensures the effective utilization of the resources. The hardware is guaranteed to be used effectively whereas in serial computation only some part of the hardware was used and the rest rendered idle.
- Also, it is impractical to implement real-time systems using serial computing.

Goals of Parallel computing

It must be remembered that parallel computing itself is not the goal. Rather, the goals are what results from parallel computing: reducing run time, performing larger calculations, or decreasing energy consumption.

1. Faster run time with more compute cores
2. Larger problem sizes with more compute nodes
3. Energy efficiency by doing more with less
4. To speed up the computer processing capability or in words. Means, it increases the computational speed.
5. Increases throughput, i.e, amount of processing that can be accomplished during a given interval of time.
6. Improves the performance of the computer for a given clock speed.
7. Two or more ALU's in CPU can work concurrently to increase throughput.
8. The system may have two or more processors operating concurrently.

Types of Parallelism:

1. **Bit-level parallelism –**
   It is the form of parallel computing which is based on the increasing processor's size. It reduces the number of instructions that the system must execute in order to perform a task on large-sized data.

   Example: Consider a scenario where an 8-bit processor must compute the sum of two 16-bit integers. It must first sum up the 8 lower-order bits, then add the 8 higher-order bits, thus requiring two instructions to perform the operation. A 16-bit processor can perform the operation with just one instruction.
2. **Instruction-level parallelism –**
   A processor can only address less than one instruction for each clock cycle phase. These instructions can be re-ordered and grouped which are later on executed concurrently without affecting the result of the program. This is called instruction-level parallelism.
3. **Task Parallelism –**
   Task parallelism employs the decomposition of a task into subtasks and then allocating each of the subtasks for execution. The processors perform the execution of sub-tasks concurrently.
4. **Data-level parallelism (DLP) –**
   Instructions from a single stream operate concurrently on several data – Limited by non-regular data manipulation patterns and by memory bandwidth

Applications of Parallel Computing:

- Databases and Data mining.
- Real-time simulation of systems.
- Science and Engineering.
- Advanced graphics, augmented reality, and virtual reality.

Advantages of Parallel Computing over Serial Computing are as follows:

1. It saves time and money as many resources working together will reduce the time and cut potential costs.
2. It can be impractical to solve larger problems on Serial Computing.
3. It can take advantage of non-local resources when the local resources are finite.
4. Serial Computing 'wastes' the potential computing power, thus Parallel Computing makes better work of the hardware.

Limitations of Parallel Computing:

- It addresses such as communication and synchronization between multiple sub-tasks and processes which is difficult to achieve.

- The algorithms must be managed in such a way that they can be handled in a parallel mechanism.
- The algorithms or programs must have low coupling and high cohesion. But it's difficult to create such programs.
- More technically skilled and expert programmers can code a parallelism-based program well.

5A.2 Speedup of parallelism

The *speedup* of a parallel algorithm *over* a corresponding sequential algorithm is the ratio of the compute time for the sequential algorithm to the time for the parallel algorithm. If the speedup factor is *n*, then we say we have *n-fold* speedup. For example, if a sequential algorithm requires 10 min of compute time and a corresponding parallel algorithm requires 2 min, we say that there is 5-fold speedup.

Amdahl's Law is a formula for estimating the maximum speedup from an algorithm that is part sequential and part parallel. The search for *2k*-digit primes illustrates this kind of problem: First, we create a list of all *k*-digit primes, using a sequential sieve strategy; then we check *2k*-digit random numbers in parallel until we find a prime.

The Amdahl's Law formula is

$$overall\ speedup = \frac{1}{(1-P) + \frac{P}{S}}$$

- P is the time proportion of the algorithm that can be parallelized.
- S is the speedup factor for that portion of the algorithm due to parallelization.

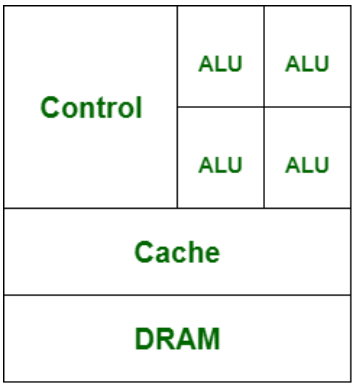
For example, suppose that we use our strategy to search for primes using 4 processors, and that 90% of the running time is spent checking 2k-digit random numbers for primality (after an initial 10% of the running time computing a list of k-digit primes). Then P = .90 and S = 4 (for 4-fold speedup). According to Amdahl's Law,

$$overall\ speedup = \frac{1}{(1-0.90) + \frac{0.90}{4}} = \frac{0.10}{0.225} = 3.077$$

This estimates that we will obtain about 3-fold speedup by using 4-fold parallelism.

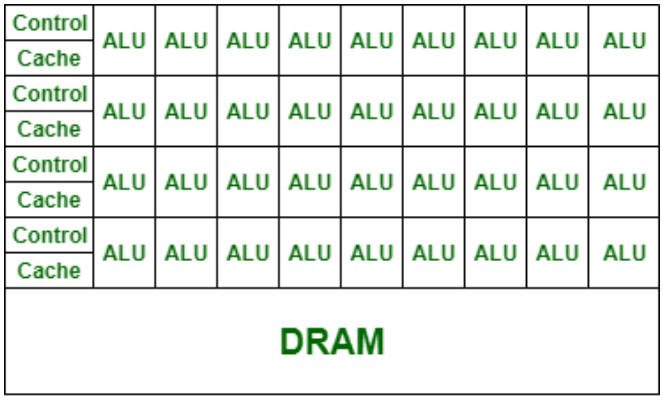
5A.3 CPU vs HPC (High Performance Computing like GPU)

- **Central Processing Unit (CPU):**
  CPU is known as brain for every ingrained system. CPU comprises the arithmetic logic unit (ALU) accustomed quickly to store the information and perform calculations and Control Unit (CU) for performing instruction sequencing as well as branching. CPU interacts with more computer components such as memory, input and output for performing instruction.



CPU

- **Graphics Processing Unit (GPU):**
  GPU is used to provide the images in computer games. GPU is faster than CPU's speed and it emphasis on high throughput. It's generally incorporated with electronic equipment for sharing RAM with electronic equipment that is nice for the foremost computing task. It contains more ALU units than CPU.



GPU

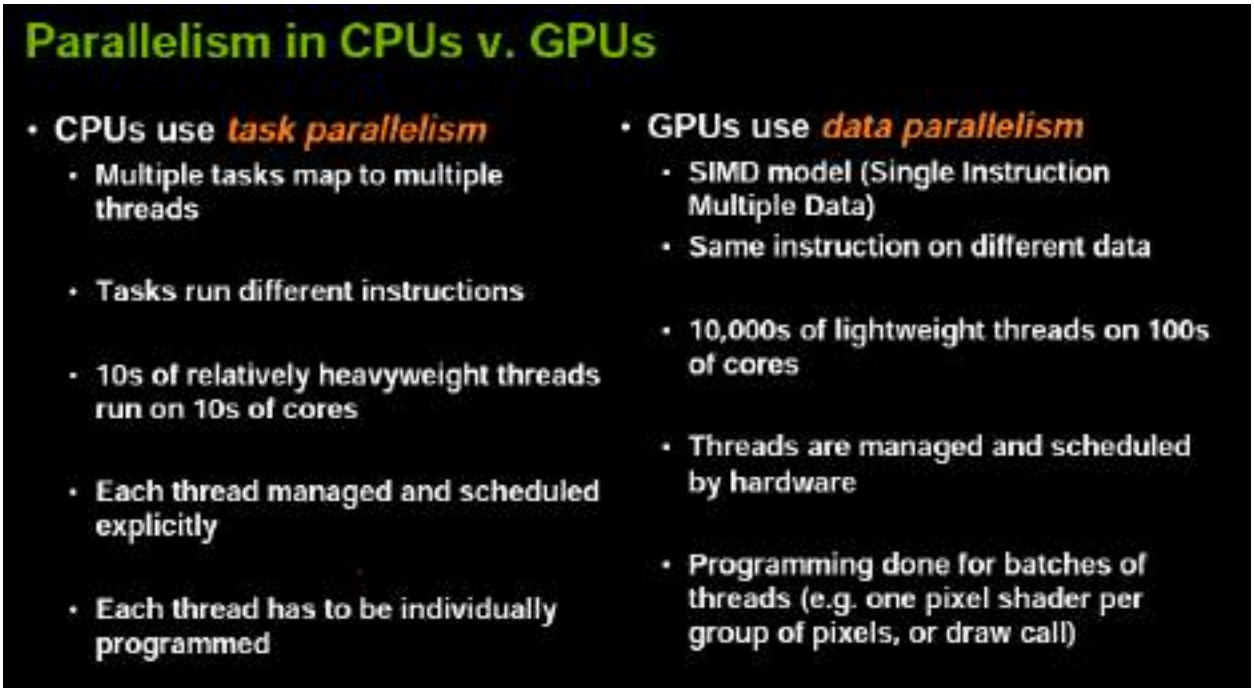
GPU acceleration is the practice of using a graphics processing unit (GPU) in addition to a central processing unit (CPU) to speed up processing-intensive operations. GPU-accelerated computing is beneficial in data-intensive applications, such as artificial intelligence and machine learning.

GPUs will be faster than CPUs on most rendering tasks. This is because the GPU is great at handling lots of information and processing it on its thousands of cores quickly in parallel.

A CPU consists of four to eight CPU cores, while the GPU consists of **hundreds of smaller cores**. Together, they operate to crunch through the data in the application. This massively parallel architecture is what gives the GPU its high compute performance.

The basic difference between CPU and GPU is that CPU emphasis on low latency. Whereas, GPU emphasis on high throughput. CPU (central processing unit) is a generalized processor that is designed to carry out a wide variety of tasks. Let's see the difference between CPU and GPU:

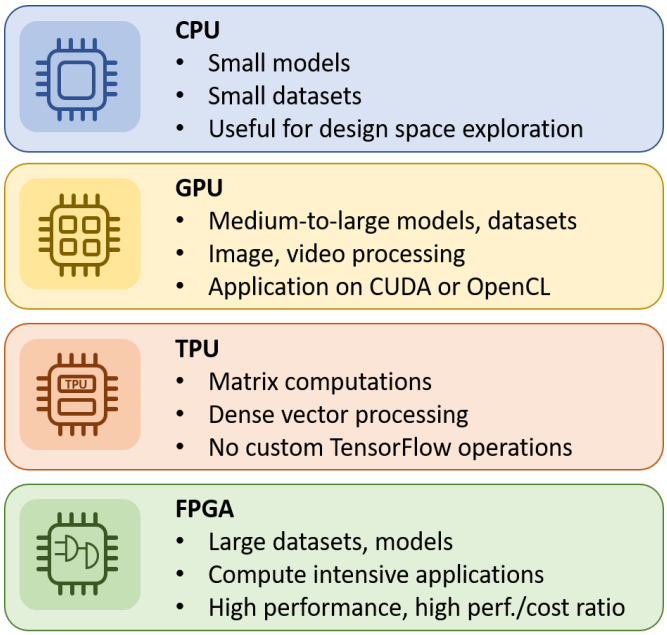
S.NO	CPU	GPU
1.	CPU stands for Central Processing Unit.	While GPU stands for Graphics Processing Unit.
2.	CPU consumes or needs more memory than GPU.	While it consumes or requires less memory than CPU.
3.	The speed of CPU is less than GPU's speed.	While GPU is faster than CPU's speed.
4.	CPU contain minute powerful cores.	While it contain more weak cores.
5.	CPU is suitable for serial instruction processing.	While GPU is not suitable for serial instruction processing.
6.	CPU is not suitable for parallel instruction processing.	While GPU is suitable for parallel instruction processing.
7.	CPU emphasis on low latency.	While GPU emphasis on high throughput.



Application developers harness the performance of the parallel GPU architecture using a parallel programming model invented by NVIDIA called "CUDA." All NVIDIA GPUs - GeForce, Quadro, and Tesla - support the NVIDIA CUDA parallel-programming model. **(CUDA – Compute Unified Device Architecture)**

GPU: Graphics Processing Unit

GPU is a specialized processing unit with enhanced mathematical computation capability, ideal for computer graphics and machine-learning tasks.



TPU: Tensor Processing Unit

The next big future move made after GPUs is Google's TPU(Tensor Processing Unit). The TPU is 15x to 30x faster than current GPUs and CPUs on production AI applications that use neural network inference

FPGA: Field Programmable Gate Arrays

FPGAs are designed to perform concurrent fixed-point operations with a close-to-hardware programming approach, while GPUs are optimised for parallel processing of floating-point operations using thousands of small cores

5A.4 Architecture of a modern GPU

A GPU uses many lightweight processing cores, leverages data parallelism, and has high memory throughput. While the specific components will vary by model, fundamentally most modern GPUs use single instruction multiple data (SIMD) stream architecture. GPUs uses single instruction multiple threads (SIMT) adds multithreading to SIMD which improves efficiency as there is less instruction fetching overhead.

Brief introduction to NVIDIA and CUDA

- **Nvidia Corporation** is an American multinational technology company incorporated in Delaware and based in Santa Clara, California.
- It is a software and fabless company which designs graphics processing units (GPUs), application programming interface (APIs) for data science and high-performance computing as well as system on a chip units (SoCs) for the mobile computing and automotive market.
- Nvidia is a global leader in artificial intelligence hard- and software. Its professional line of GPUs are used in workstations for applications in such fields as architecture, engineering and construction, media and entertainment, automotive, scientific research, and manufacturing design.
- In addition to GPU manufacturing, Nvidia provides an API called **CUDA** that allows the creation of massively parallel programs which utilize GPUs. They are deployed in supercomputing sites around the world.

History of Nvidia GPU Architecture

While Nvidia GPUs have certainly made the news more frequently in recent years, they're by no means new. In fact, there have been multiple iterations of Nvidia GPUs and advances in GPU architecture over the years. So, let's take a look back at recent history to understand how GPUs have evolved over time.

- Kelvin

  Released in 2001, Kelvin was Nvidia's first new GPU microarchitecture of the millennium. The original Xbox gaming console used an NV2A GPU with the Kelvin microarchitecture. The GeForce 3 and GeForce 4 series GPUs were released with this microarchitecture.

- Rankine

  Rankine was the follow-up to Kelvin released in 2003 and used for the GeForce 5 series of Nvidia GPUs. Rankine had support for vertex and fragment programs and increased VRAM size to 256MB.

- Curie

  Curie -- the microarchitecture used by GeForce 6 and 7 series GPUs -- was released as the successor to Rankine in 2004. Curie doubled the amount of VRAM to 512MB and was the first generation of Nvidia GPUs to support the PureVideo video decoding.

- Tesla

  The Tesla GPU microarchitecture, released in 2006 as Curie's successor, introduced several important changes to Nvidia's GPU product line. In addition to being the architecture used by the GeForce 8, 9, 100, 200, and 300 series GPUs, Tesla was used by the Quadro line of GPUs designed for use cases outside of graphics processing.

  Confusingly, Tesla was both the name of a GPU microarchitecture and a brand of Nvidia GPUs. In 2020, Nvidia decided to stop using the Tesla name to avoid confusion with the popular electric vehicle brand.

- Fermi

  Tesla's successor Fermi was released in 2010. Fermi introduced a number of enhancements including:

  - Support for 512 CUDA cores
  - 64KB of RAM and the ability to partition L1 cache/shared memory
  - Support for Error Correcting Code (ECC)

- Kepler

  Kepler GPU microarchitecture was released as the successor to Fermi 2012. Key improvements over Fermi were:

  - A new streaming multiprocessor architecture known as SMX
  - Support for TXAA (an anti-aliasing method)
  - An increase in CUDA cores to 1536
  - Less power consumption
  - Support for automatic overclocking via GPU boost

- Support for GPUDirect which allowed GPUs - both in the same computer or with network access to one another - to communicate without accessing the CPU

- Maxwell
  - Maxwell, released in 2014, was the successor to Fermi. According to Nvidia, The first generation of Maxwell GPUs had these advantages over Fermi:
  - More efficient multiprocessors as a result of enhancements related to control logic partitioning, clock-gating, instruction scheduling, and workload balancing
  - 64KB of dedicated shared memory on each streaming multiprocessor
  - Native shared memory atomic operations that offered performance improvements when compared to the lock/unlock paradigm used by Fermi
  - Dynamic parallelism support
- Pascal
  - Pascal succeeded Maxwell in 2016. This Nvidia GPU microarchitecture offered improvements over Maxwell such as:
  - Support for NVLink communications, which can offer a significant speed advantage over PCIe
  - High bandwidth Memory 2 (HBM2)- a 4096-bit memory bus that offered a memory bandwidth of 720 GB
  - Compute preemption
  - Dynamic load balancing to enable optimization of GPU resource utilization
- Volta

  Volta was a somewhat unique microarchitecture iteration released in 2017. While most previous microarchitectures were used in consumer GPUs, Volta GPUs were marketed strictly for professional applications. Volta was also the first microarchitecture to use Tensor Cores.

  Tensor Cores are a newer type of processing core that perform specialized math calculations. Specifically, Tensor Cores perform matrix operations that enable AI and deep learning use cases.

- Turing

  Turing was released in 2018 and in addition to supporting Tensor Cores, includes a number of consumer-focused GPUs as well. Turing is the microarchitecture used by Nvidia's popular Quadro RTX and GeForce RTX series GPUs. These GPUs support real-time ray tracing (a.k.a. RTX) which is vital to computationally heavy applications such as virtual reality (VR).

- Ampere

  The Ampere microarchitecture is just beginning to hit the market. Ampere aims to further enable high-performance computing (HPC) and AI use cases. Enhancements in Ampere including 3rd generation NVLink and Tensor cores, structural sparsity (the conversion of unneeded parameters to zeros to enable AI model training), 2nd generation ray tracing cores, multi-instance GPU (MIG) to enable partitioning of A100 GPUs into individual logically isolated and secure GPU instances.

CUDA compute hierarchy

The processing resources in CUDA are designed to help optimize performance for GPU use cases. Three of the fundamental components of the hierarchy are threads, thread blocks, and kernel grids.

1. Threads

   A thread -- or CUDA core -- is a parallel processor that computes floating point math calculations in an Nvidia GPU. All the data processed by a GPU is processed via a CUDA core. Modern GPUs have hundreds or even thousands of CUDA cores. Each CUDA core has its own memory register that is not available to other threads.

   While the relationship between compute power and CUDA cores is not perfectly linear, generally speaking -- and assuming all else is equal -- the more CUDA cores a GPU has, the more compute power it has. However, there are a variety of exceptions to this general idea. For example, different GPU microarchitectures can impact performance and make a GPU with fewer CUDA cores more powerful

2. Thread blocks

   As the name implies, a thread block -- or CUDA block -- is a grouping of CUDA cores (threads) that can be executed together in series or parallel. The logical grouping of cores enables more efficient data mapping. Thread blocks share memory on a per-block basis. Current CUDA architecture caps the amount of threads per block at 1024. Every thread in a given CUDA block can access the same shared memory.

3. Kernel grids

   The next layer of abstraction up from thread blocks is the kernel grid. Kernel grids are groupings of thread blocks on the same kernel. Grids can be used to

perform larger computations in parallel (e.g. those that require more than 1024 threads), however since different thread blocks cannot use the same shared memory, the same synchronization that occurs at the block-level does not occur at the grid-level.

CUDA memory hierarchy

Like compute resources, memory allocation follows a specific hierarchy in CUDA. While the CUDA compiler automatically handles memory allocation, CUDA developers can and do program to optimize memory usage directly. Here are the key concepts to understand about the CUDA memory hierarchy.

- Registers

  Registers are the memory that gets allocated to individual threads (CUDA cores). Because registers exist in “on-chip” memory and are dedicated to individual threads, the data stored in a register can be processed faster than any other data. The allocation of memory in registers is a complicated process and is handled by compilers as opposed to being controlled by software CUDA developers write.

- Read-only memory

  Read-only (RO) is on-chip memory on GPU streaming multiprocessors. It is used for specific tasks such as texture memory which can be accessed using CUDA texture functions. In many cases, fetching data from read-only memory can be faster and more efficient than using global memory.

- L1 Cache/shared memory

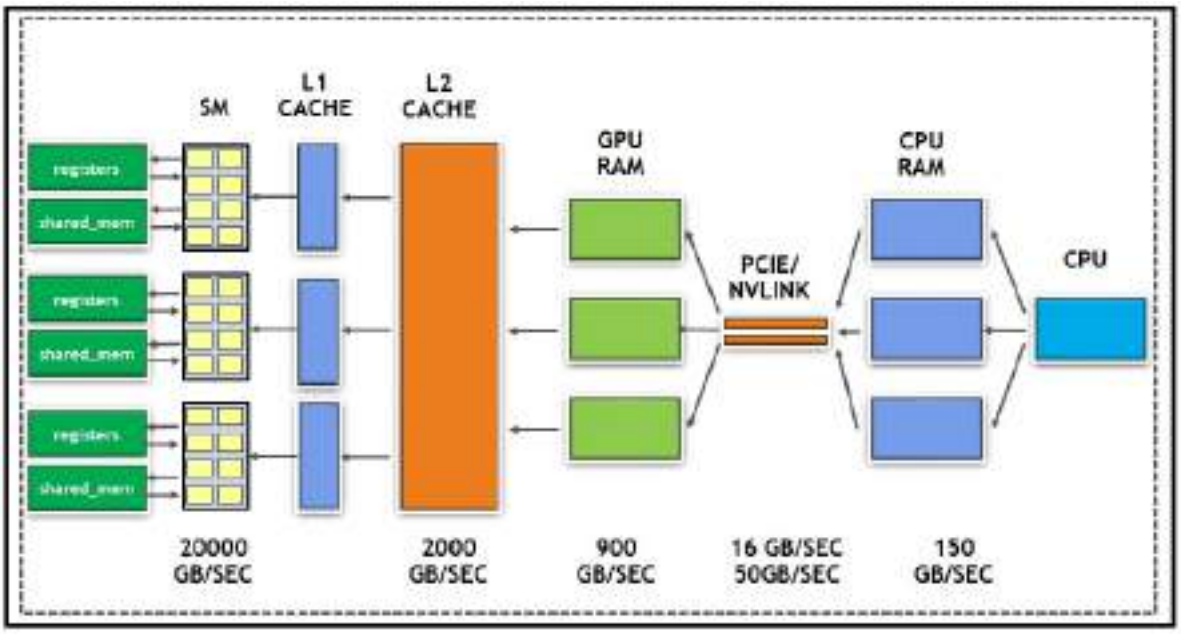
  Layer 1 (L1) cache and shared memory is on-chip memory that is shared within thread blocks (CUDA blocks). Because L1 cache and shared memory exists on-chip, it is faster than both L2 cache and global memory. The fundamental difference between L1 cache and shared memory is: shared memory usage is controlled via software while L1 cache is controlled by hardware.

- L2 Cache

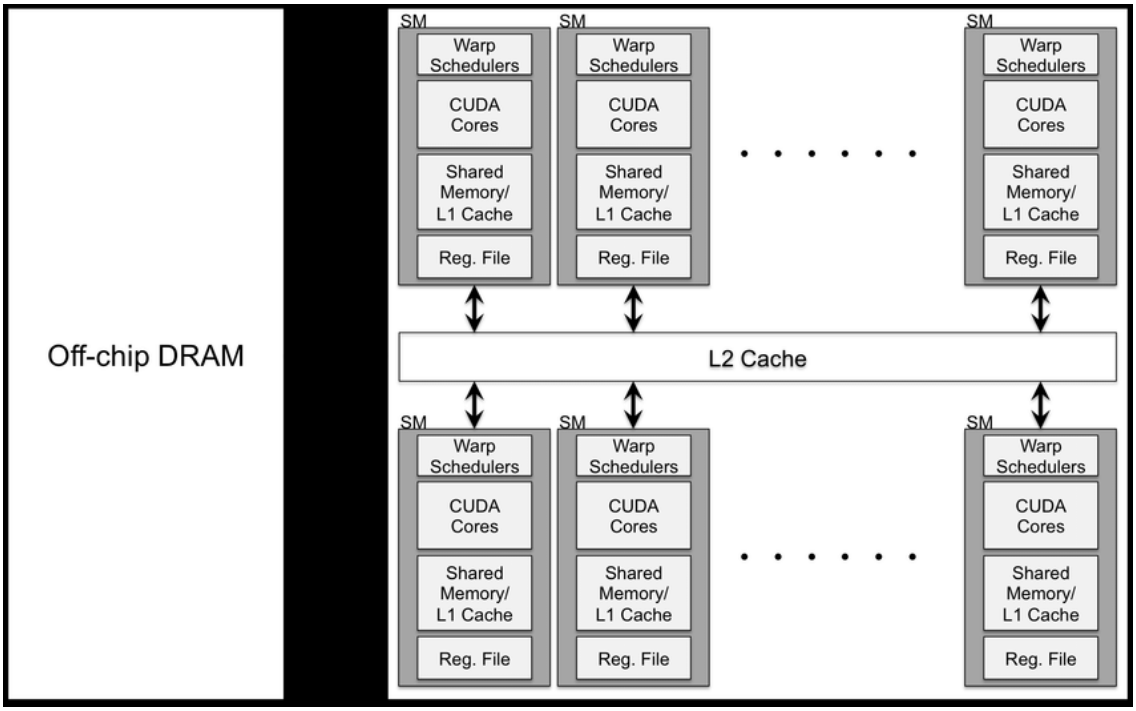
  Layer 2 cache can be accessed by all threads in all CUDA blocks. L2 cache stores both global and local memory. Retrieving data from L2 cache is faster than retrieving data from global memory.

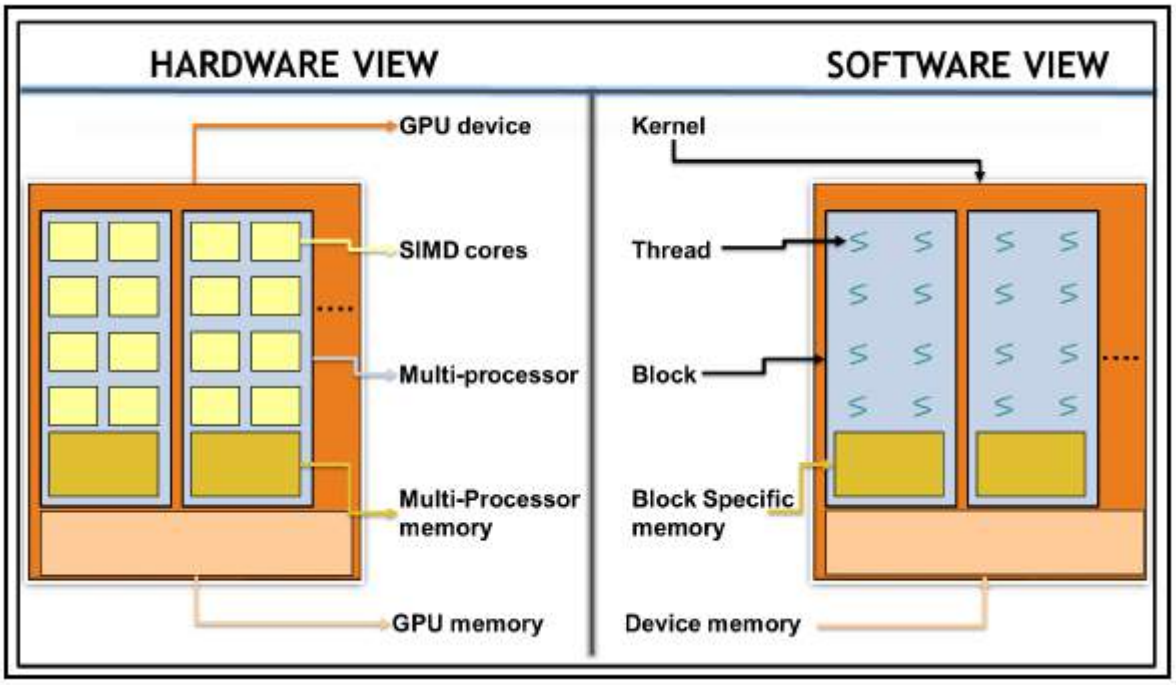
- Global memory

  Global memory is the memory that resides in a device's DRAM. Using a CPU analogy, global memory is comparable to RAM. Fetching data from global memory is inherently slower than fetching it from L2 cache.

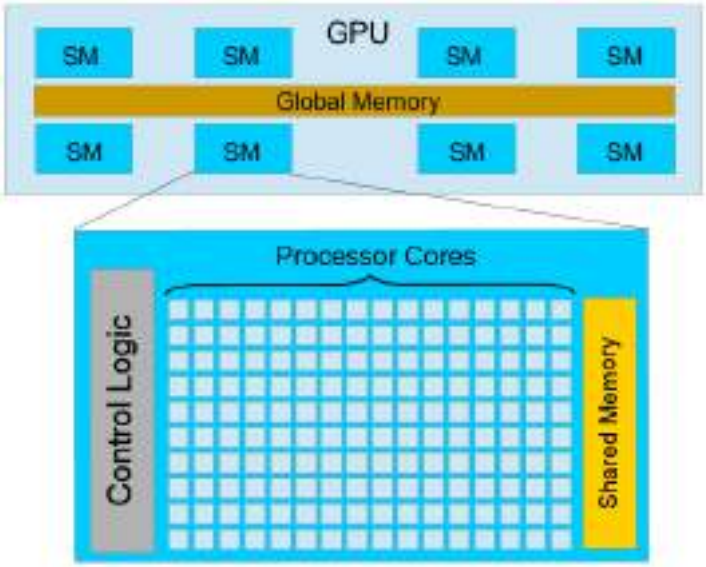


General modern GPU architecture



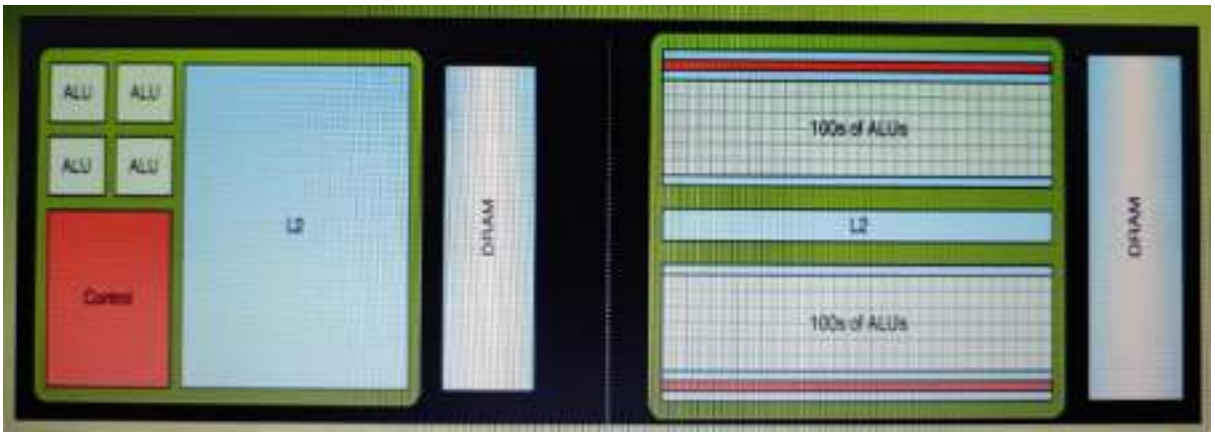


Software		Hardware
CUDA thread	Executes on/as	CUDA Core/SIMD code
CUDA block		Streaming multiprocessor
GRID/kernel		GPU device

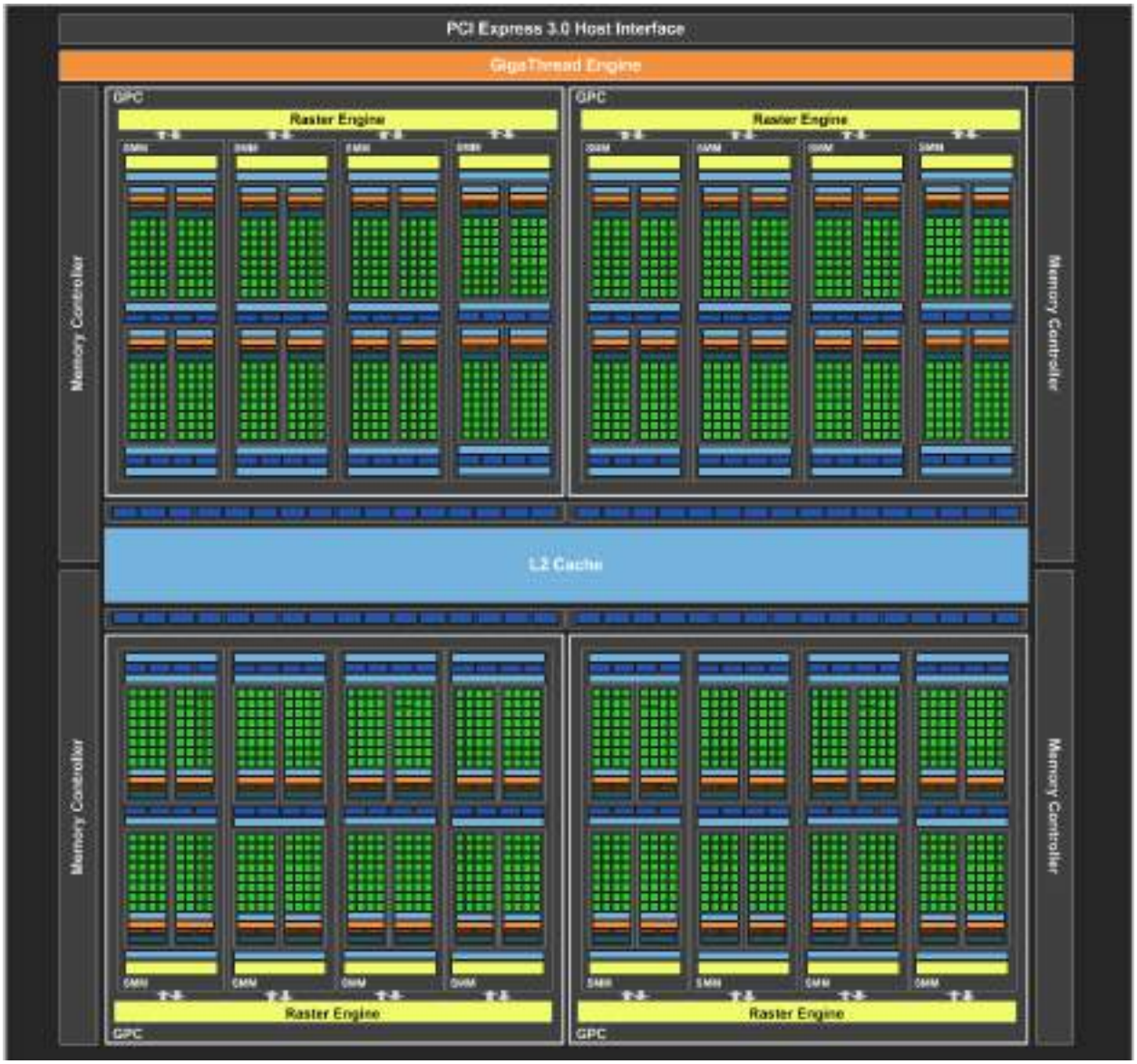


NVIDIA GeForce980 (GTX980) GPU – [Maxwell]

- CUDA uses thousands of threads executing in parallel, all these threads are executing in same function called kernel.
- Compiler organizes these threads into blocks known as thread blocks.
- Grids contains thread blocks.



- Streaming Multi Processors(SMP) are part of GPU that actually runs these kernels.
    - SMP of Maxwell version - SMM
    - SMP of Kepler version - SMX
- SMM are the heart of GPU architecture, performs all the computations, they have their own control units, execution pipelines, caches and registers.
- In NVIDIA GeForce GTX 980 -
    - GTX – Giga Texel Shader
    - 4 64-bit memory controllers
    - 4 Graphics Processing Clusters (GPC)
    - 4 SMPs per each GPC – Total 16 SMPs
    - Each SMP –
        - contains 4 blocks
        - Each block contains 32 cores (4X8)
        - Total 32 cores * 4 blocks = 128 cores in each SMM
    - So, 16 SMPs * 128 cores = 2048 cores
    - Warp Scheduler: it groups 32 parallel threads called warps.
    - Warp schedulers can run two instructions per warp every clock
    - Each Warp scheduler contains Instruction buffer.



NVIDIA GeForce GTX 980 GPU Architecture – Maxwell 2nd Version


SMM
4 x 8 = 32 cores per block
32 cores/block x 4 blocks = 128 cores
128 Cores for a single SMM


PCI Express 3.0 Host Interface
GPC
Raster Engine
16 SMM in total

The <u>most</u> modern GPU was NVIDIA GeForce RTX 3070 – [Ampere's version]

- RTX – Ray Tracing eXtreme
- 8 memory controllers
- 6 GPCs
- 8 SMM per each GPC, means 48 SMMs
- 5888 CUDA\Shader\Stream cores
- 46 RT cores
- 184 Tensor cores

5A.5 Parallel Programming languages:

APIs/ frameworks **support parallelism in host languages**	Dataflow programming	Distributed computing	Functional programming
• Apache Beam • Apache Flink • Apache Hadoop • Apache Spark • CUDA • OpenCL • OpenHMPP • OpenMP for C, C++, and Fortran (shared memory and attached GPUs)	• CAL • E (also object-oriented) • Joule (also distributed) • LabVIEW (also synchronous, also object-oriented) • Lustre (also synchronous) • Preesm (also synchronous) • Signal (also synchronous) • SISAL • BMDFM	• Bloom • Emerald • Hermes • Julia • Limbo • MPD • Oz - • Sequoia • SR	• Clojure • Concurrent ML • Elixir • Elm • Erlang • Futhark • Haskell • Id • MultiLisp • SequenceL

Logic programming	Multi-threaded		Coordination language
• Constraint Handling Rules[1] • Parlog • Prolog • Mercury	• C= • Cilk • Cilk Plus • Cind • C# • Clojure • Concurrent Pascal • Emerald •	• Fork – programming language for the PRAM model. • Go • Java • LabVIEW • ParaSail • Rust • SequenceL	• CnC (Concurrent Collections) • Glenda • Linda coordination language • Millipede

5A.6 Parallel Programming Models:

There are four types of parallel programming models:

1. Shared memory model
2. Message passing model
3. Threads model
4. Data parallel model

1. Shared Memory Model

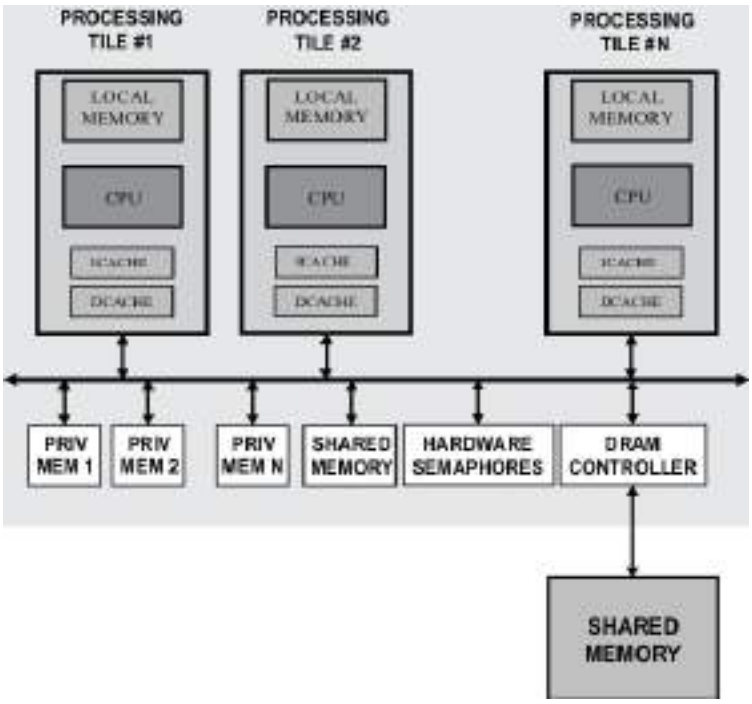
- In this type, the programmer views his program as collection of processes which use common or shared variables.
- The processor may not have a private program or data memory. A common program and data are stored in the main memory. This is accessed by all processors.
- Each processor is assigned a different part of the program and the data. The main program creates separate process for each processor.
- The process is allocated to the processor along with the required data. These process are executed indecently on different processors.
- After the execution, all the processors have to rejoin the main program

Advantages

- Program development becomes simple.
- There is no need to explicitly specify the communication between data and process.

Disadvantages

- Users will not understand where his variables use stored.
- The native compiler present at each processor translates user variables into act addresses in main memory.



2. **Message Passing Model**

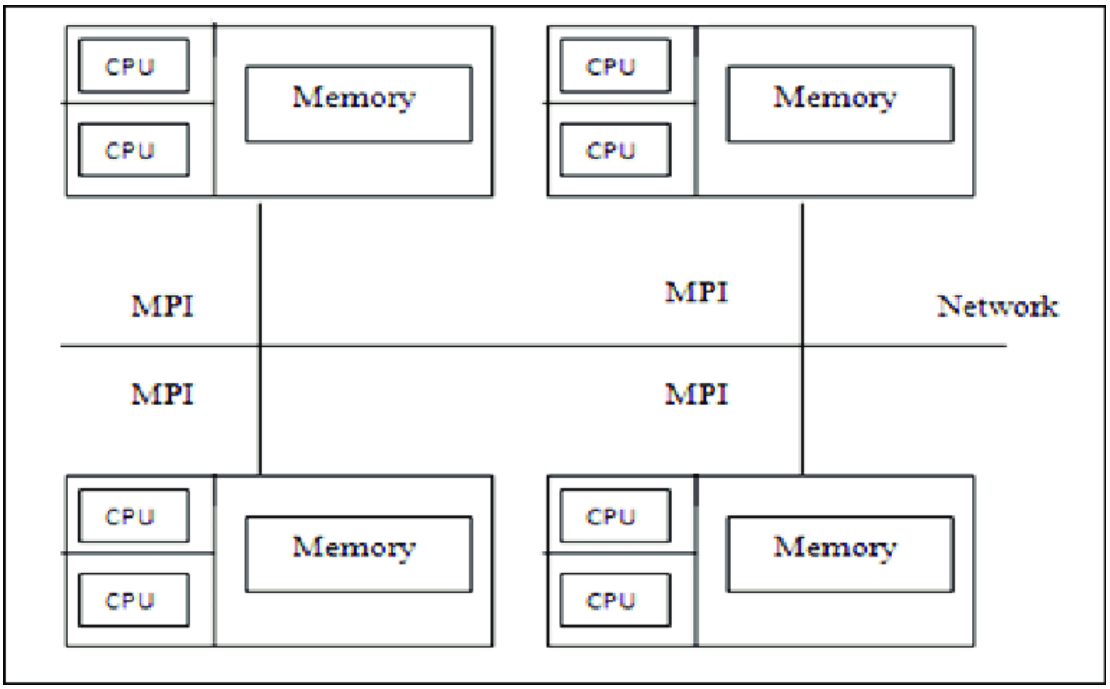
- In this type, different processes may be present on single multiple processors. Every process has its own set of data
- The data transfer between the processes is achieved by send and receive message requires co-operation between every process.
- There must be a receive operation for every send operation.

Advantages

- The data can be stored anywhere.
- Communication between processes is simple.
- Many Manage Passing Interfaces (MPS) are available.

Disadvantages

- Programmers should take care that there is a receive function for every send function.
- If any of the process, quite, others also will stop working, they are dependent.



3. **Threads Model**

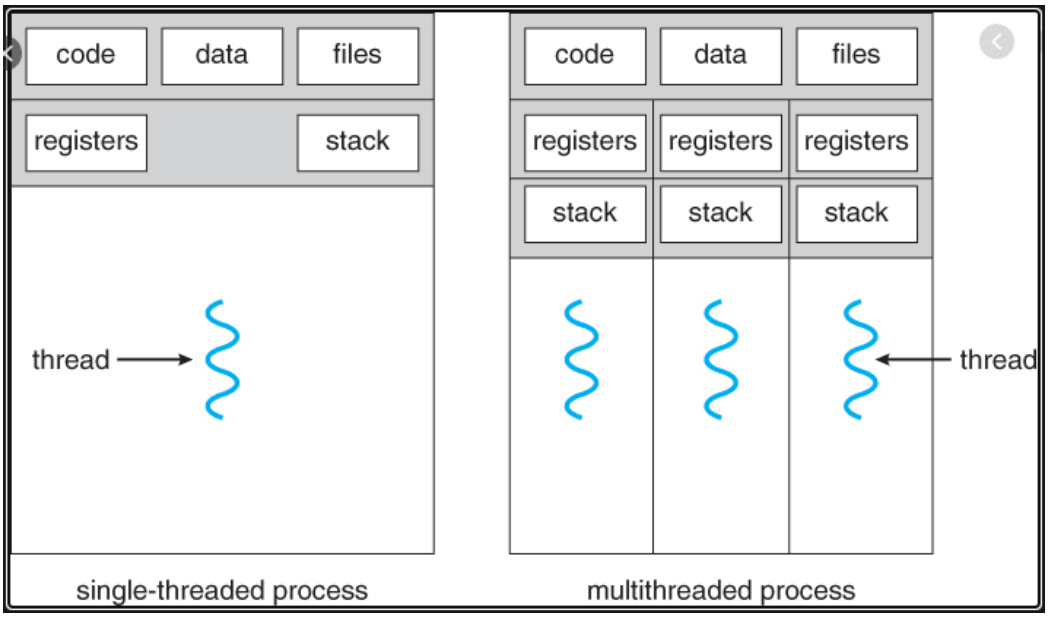
- A thread is defined as a short sequence of instructions, within a process. Different threads can be executed on same processor or on different process.
- If the threads are executed on same processor, then the processor switches between the threads in a random fashion.
- If the threads are executed on different processors, they are executed simultaneously.
- The threads communicate through global memory.

Advantages

- Programmer need not have to worry about parallel processing.

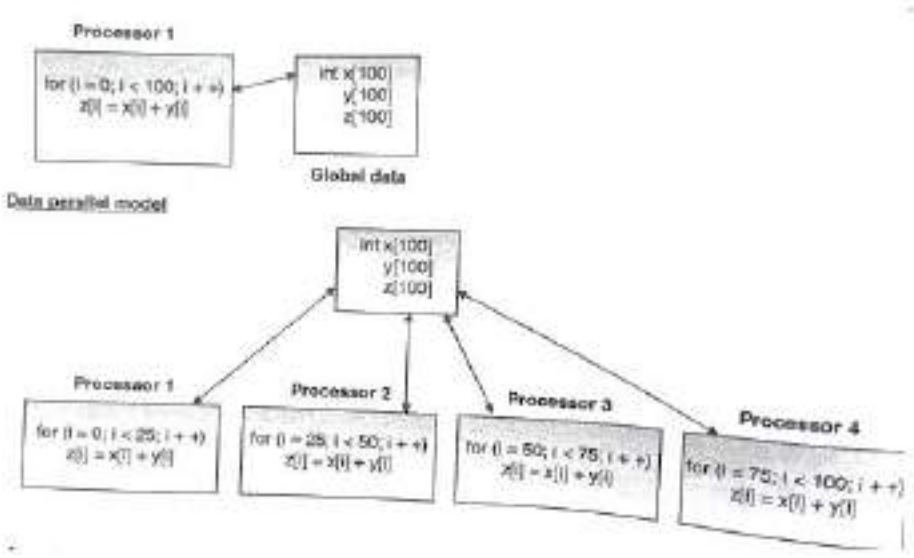
Disadvantages

- Care must be taken that no two threads update the shared resources simultaneously.



4. Data Parallel Model

- Data parallelism is one of the simplest form of parallelism. Here data set is organized into a common structure. It could be an array.
- Many programs apply the same operation on different parts of the common structure.
- Suppose the task is to add two arrays of 100 elements store the result in another array. If there are four processor then each processor can do 25 additions.



5A.7 GPU Computing

- GPU computing is the use of a GPU (graphics processing unit) as a co-processor to accelerate CPUs for general-purpose scientific and engineering computing.
- The GPU accelerates applications running on the CPU by offloading some of the compute-intensive and time consuming portions of the code. The rest of the application still runs on the CPU. From a user's perspective, the application runs faster because it's using the massively parallel processing power of the GPU to boost performance. This is known as "heterogeneous" or "hybrid" computing.
- A CPU consists of four to eight CPU cores, while the GPU consists of hundreds of smaller cores. Together, they operate to crunch through the data in the application. This massively parallel architecture is what gives the GPU its high compute performance. There are a number of GPU-accelerated applications that provide an easy way to access High-Performance Computing (HPC).

- GPU computing has become the key to optimizing deep learning, accelerating time to value (TTV), increasing processing speed during coding, enhancing data management, content creation, and product engineering, and delivering comprehensive insight into data analytics.

- This multifaceted and beneficial process happens through parallel computing. When CPUs become overwhelmed with processing massive volumes of data (i.e., Big Data), the GPU steps in and separates complex problems into millions of tasks, making it easier to find solutions all at once. The GPU runs various levels of tasks consecutively, which frees up the normal processing capabilities of the CPU and protects the integrity of both systems by allocating specific workloads to the most efficient processor for the job. Both the CPU and GPU can work together in an artificial intelligence (AI) ecosystem, supporting problem solving interchangeably.

Relation between CPU and GPU

- GPUs get the credit for leading the charge in supercomputing. In situations when graphics or content must be rendered at high speeds, GPUs are essential. Utilizing GPU computing benefits the internal CPU, allowing for processing and rendering graphics at an accelerated rate.
- This alliance between graphics processing units and central processing units promotes a smoother processing system, achieving utilization that could not be achieved by a CPU on its own. While CPUs do have much higher processing speeds, GPUs have unmatched processing capabilities thanks to parallelism.

Benefits of GPU computing

- Acting as a companion processor to CPUs, GPUs exponentially boost the speed and processing capabilities of a system. GPUs perform computing applications concerning technical and scientific data in an accelerated way, adding efficiency when they are integrated alongside CPUs.
- Another benefit to using GPUs is that they lessen the burden on the CPU by processing repetitive data in smaller chunks across several processors and enabling computing to proceed uninhibited by the limitless number of problems that it is tasked with solving.
- In addition to processing power, GPUs extend memory bandwidth. Working hundreds of times faster than CPUs, GPUs make the automation and intelligence of machine learning (ML) and Big Data analysis possible, as they process massive amounts of data via neural networks. The AI then learns deeply complex tasks that no data scientist would have the language to teach or translate.
- Additional benefits include but are not limited to:
    - Superior processing power
    - Exponentially greater memory storage/bandwidth
    - Robust data analysis and analysis of AI and ML
    - Rapid advances in gaming and graphics
    - Easy integration into data centers

Working of GPU computing

- IT's focus has shifted to reflect and support the compute demands of AI and data science. This work is done by GPUs. Applications that are being run on CPUs are accelerated using GPU compute, which optimizes performance and workload capacity.
- GPU computing enables applications to run with extreme efficiency by offloading series of computational scientific and technical tasks from the CPU. GPUs process thousands of tasks in seconds through their hundreds of cores via parallel processing. Parallel processing denotes a function where data sets are funneled into a GPU's processing cores and are all solved for simultaneously. Performance is increased as the GPU crunches and translates data while the CPU is running the remaining applications.
- Insights from data analytics lead the way in problem solving and increased functionality with the use of GPU compute. GPU's ability to quickly process and sort massive amounts of data allows for industry leaders to quickly and accurately access the insights into their data and innovate accordingly.

GPU Computing Applications

GPU computing is being used for numerous real-world applications. Many prominent science and engineering fields that we take for granted today would have not progressed so fast, if not GPU computing.

1. Deep Learning

   Deep learning is a subset of machine learning. Its implementation is based on artificial neural networks. Essentially, it mimics the brain, having neuron layers work in parallel. Since data is represented as a set of vectors, deep learning is well-suited for GPU computing. You can easily experience up to 4x performance gains when training your convolutional neural network on a Dedicated Server with a GPU accelerator. As a cherry on top, every major deep learning framework like TensorFlow and PyTorch already allows you to use GPU computing out-of-the-box with no code changes.

2. Drug Design

   The successful discovery of new drugs is hard in every respect. We have all become aware of this during the Covid-19 pandemic. Eroom's law states that the cost of discovering a new drug roughly doubles every nine years. Modern GPU computing aims to shift the trajectory of Eroom's law. Nvidia is currently building Cambridge-1 - the most powerful supercomputer in the UK - dedicated to AI research in healthcare and drug design.

3. Seismic Imaging

   Seismic imaging is used to provide the oil and gas industry with knowledge of Earth's subsurface structure and detect oil reservoirs. The algorithms used in seismic data processing are evolving rapidly, so there's a huge demand for additional computing power. For instance, the Reverse Time Migration method can be accelerated up to 14 times when using GPU computing.

4. Automotive design

   Flow field computations for transient and turbulent flow problems are highly compute-intensive and time-consuming. Traditional techniques often compromise on the underlying physics and are not very efficient. A new paradigm for computing fluid flows relies on GPU computing that can help achieve significant speed-ups over a single CPU, even up to a factor of 100.

5. Astrophysics

   GPU has dramatically changed the landscape of high performance computing in astronomy. Take an N-body simulation for instance, that numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body. You can accelerate the all-pairs N-body algorithm up to 25 times by using GPU computing rather than using a highly tuned serial CPU implementation.

6. Options pricing

   The goal of option pricing theory is to provide traders with an option's fair value that can then be incorporated into their trading strategies. Some type of Monte Carlo algorithm is often used in such simulations. GPU computing can help you achieve 27 times better performance per dollar compared to CPU-only approach.

7. Weather forecasting

   Weather forecasting has greatly benefited from exponential growth of mere computing power in recent decades, but this free ride is nearly over. Today weather forecasting is being driven by fine-grained parallelism that is based on extensive GPU computing. This approach alone can ensure 20 times faster weather forecasting models.

GPU Computing in the Cloud

- Even though GPU computing was once primarily associated with graphical rendering, it has grown into the main driving force of high performance computing in many different scientific and engineering fields.
- Most of the GPU computing work is now being done in the cloud or by using in-house GPU computing clusters. Here at Cherry Servers we are offering Dedicated GPU Servers with high-end Nvidia GPU accelerators. Our infrastructure services can be used on-demand, which makes GPU computing easy and cost-effective.
- Cloud vendors have democratized GPU computing, making it accessible for small and medium businesses world-wide. If Huang's law lasts, the performance of GPU will more than double every two years, and innovation will continue to sprout.

5A.8 Data Parallelism

Data parallelism is a form of parallelization which relies on splitting the computation by subdividing data across multiple processors in parallel computing environments. A data parallel algorithm focuses on distributing the data across different parallel computing nodes, in contrast to task parallelism which aims at subdividing the operations to perform. In a multiprocessor system, data parallelism is achieved when each processor performs the same task on different pieces of distributed data.

Applications

- Data parallelism finds its applications in a variety of fields ranging from physics, chemistry, biology, material sciences to signal processing.
- Sciences imply data parallelism for simulating models like molecular dynamics, sequence analysis of genome data and other physical phenomenon.
- Driving forces in signal processing for data parallelism are video encoding, image and graphics processing, wireless communications

Steps to parallelization

The process of parallelizing a sequential program can be broken down into four discrete steps.

Type	Description
Decomposition	The program is broken down into tasks, the smallest exploitable unit of concurrence.
Assignment	Tasks are assigned to processes.
Orchestration	Data access, communication, and synchronization of processes.
Mapping	Processes are bound to processors.

References

1. https://www.spiceworks.com/tech/iot/articles/what-is-parallel-processing/
2. https://livebook.manning.com/book/parallel-and-high-performance-computing/chapter-1/51
3. http://selkie.macalester.edu/csinparallel/modules/IntermediateIntroduction/build/html/ParallelSpeedup/ParallelSpeedup.html
4. https://www.stolaf.edu/people/rab/pdc/text/alg.htm
5. https://www.boston.co.uk/info/nvidia-kepler/what-is-gpu-computing.aspx
6. https://www.google.com/search?q=gpu+means+high+performance+computing&rlz=1C1CHBF_enIN856IN856&oq=gpu+means+high+performa&aqs=chrome.2.69i57j33i160l3j33i22i29i30.9134j0j7&sourceid=chrome&ie=UTF-8#imgrc=sqRNQ06qVJ7k0M
7. https://www.geeksforgeeks.org/difference-between-cpu-and-gpu/
8. https://www.analyticsvidhya.com/blog/2022/08/evolution-of-tpus-and-gpus-in-deep-learning-applications/#:~:text=The%20next%20big%20future%20move,that%20use%20neural%20network%20inference.
9. chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf
10. https://forum.beyond3d.com/threads/how-a-modern-gpu-works.56334/
11. https://www.google.com/imgres?imgurl=https%3A%2F%2Fslideplayer.com%2Fslide%2F5719359%2F19%2Fimages%2F11%2FDiagram%2Bof%2Ba%2Bmodern%2BGPU%2BInput%2Bfrom%2BCPU%2BHost%2Binterface.jpg&imgrefurl=https%3A%2F%2Fslideplayer.com%2Fslide%2F5719359%2F&tbnid=OAr1ZIEcwqeP1M&vet=12ahUKEwjCvaSuj_f8AhXFi9gFHcA8Bh8QMygHegUIARDHAQ..i&docid=zJ3xHCvzi62c3M&w=960&h=720&q=architecture%20of%20a%20modern%20gpu&ved=2ahUKEwjCvaSuj_f8AhXFi9gFHcA8Bh8QMygHegUIARDHAQ
12. chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://acg.cis.upenn.edu/milom/cis371-Spring12/lectures/GPU-Architecture.pdf
13. https://www.partitionwizard.com/partitionmagic/gpu-architecture.html
14. chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf
15. https://www.cherryservers.com/blog/everything-you-need-to-know-about-gpu-architecture
16. https://www.researchgate.net/figure/Illustration-of-a-modern-GPU-architecture_fig2_314092937
17. https://www.ques10.com/p/36530/explain-the-various-types-of-parallel-programming-/
18. https://www.sciencedirect.com/science/article/abs/pii/0167739X92900349#:~:text=The%20languages%20are%3A%20SR%20(based,Orca%20(logically%20shared%20data).
19. https://www.researchgate.net/figure/Shared-memory-architecture_fig2_221339822

20. https://www.researchgate.net/figure/Message-Passing-Model_fig1_281270827
21. https://pt.slideshare.net/vlbthambawita/lecture-2-more-about-parallel-computing?next_slideshow=true
22. http://scicom.esi.uclm.es/SciCom/pdfs/IEEETPDS2012.pdf
23. chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.ijcsit.com/docs/Volume%205/vol5issue04/ijcsit2014050497.pdf
24. https://www.hpe.com/us/en/what-is/gpu-computing.html
25. https://www.cherryservers.com/blog/what-is-gpu-computing
26. https://link.springer.com/referenceworkentry/10.1007/978-1-4419-9863-7_1028
27. https://en.wikipedia.org/wiki/Data_parallelism

Syllabus

Unit 5B: Introduction to CUDA Programming

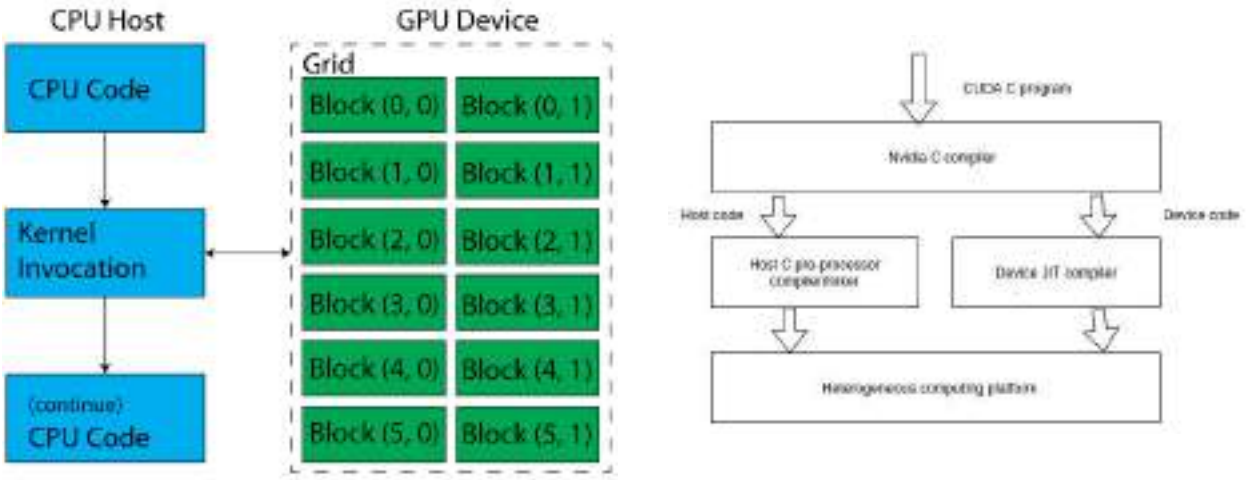
CUDA program structure, vector addition kernel, matrix multiplication kernel, device global memory and data transfer, kernel functions and threading, CUDA thread organization and synchronization, querying device properties, CUDA memory types

INDEX

5B.1 CUDA Programming Structure

A CUDA program is comprised of two primary components: a host and a GPU kernel. Host code runs locally on a CPU, while the GPU kernel codes are GPU functions that run on GPU devices. Kernel execution can be completely independent of host execution.

Below Figure shows the basic programming structure of a GPU application.



An application starts by executing the code on a CPU host. At a certain point in the program, the host code invokes a GPU kernel on a GPU device.

This kernel is executed on a GPU grid composed of independent groups of threads called thread blocks. The GPU executes a kernel in parallel using many threads. When the kernel completes its execution, the CPU continues to execute the original program.

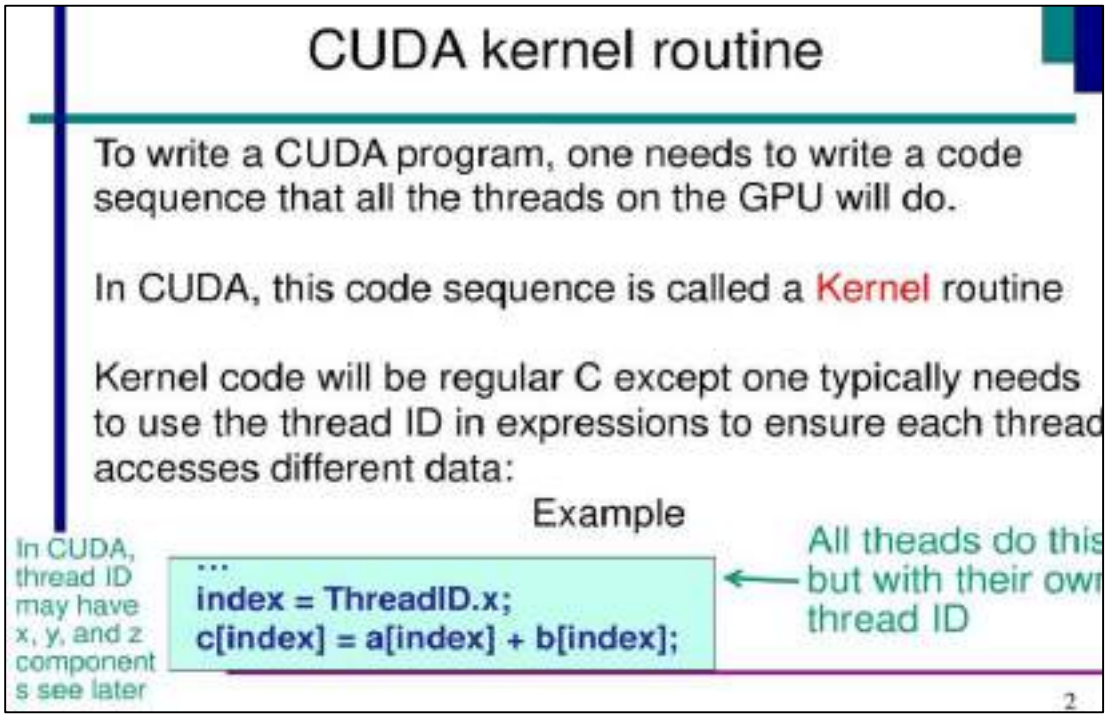
The host may also continue its execution without waiting for the GPU to complete its tasks using an asynchronous kernel.

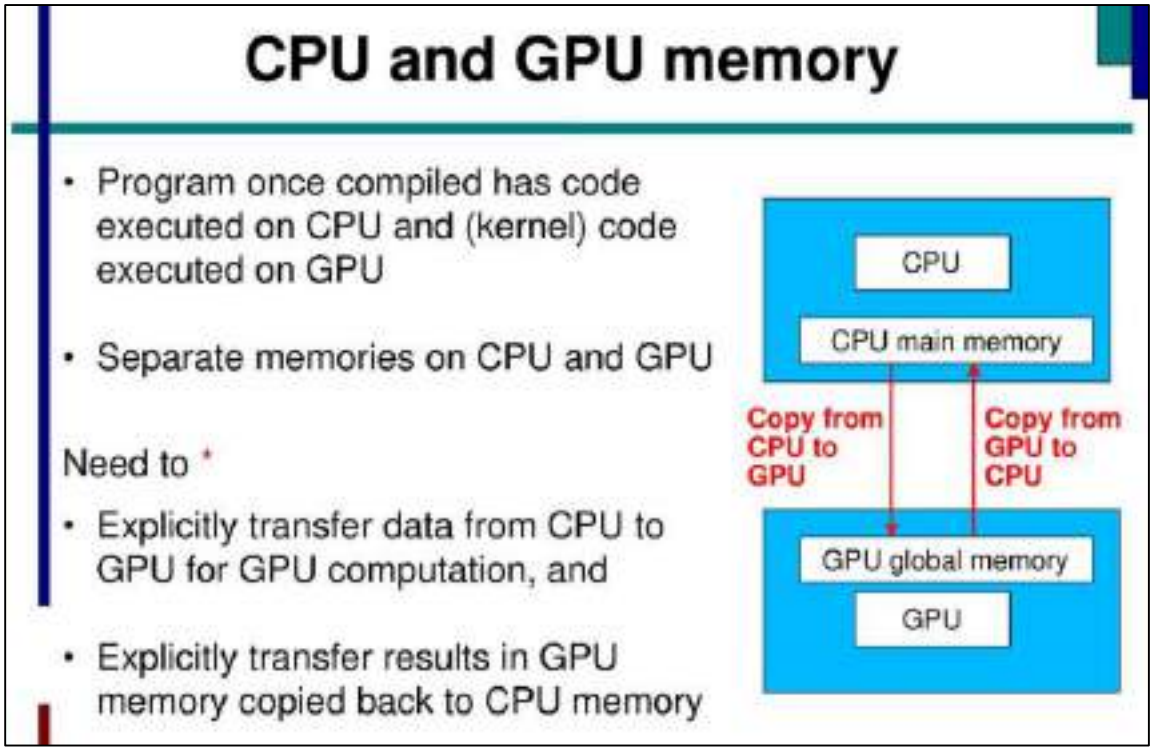
A typical CUDA program has code intended both for the GPU and the CPU.

By default, a traditional C program is a CUDA program with only the host code. **The CPU is referred to as the host, and the GPU is referred to as the device.** Whereas the host code can be compiled by a traditional C compiler as the GCC, the device code needs a special compiler to understand the api functions that are used. For Nvidia GPUs, the compiler is called the NVCC (Nvidia C Compiler).

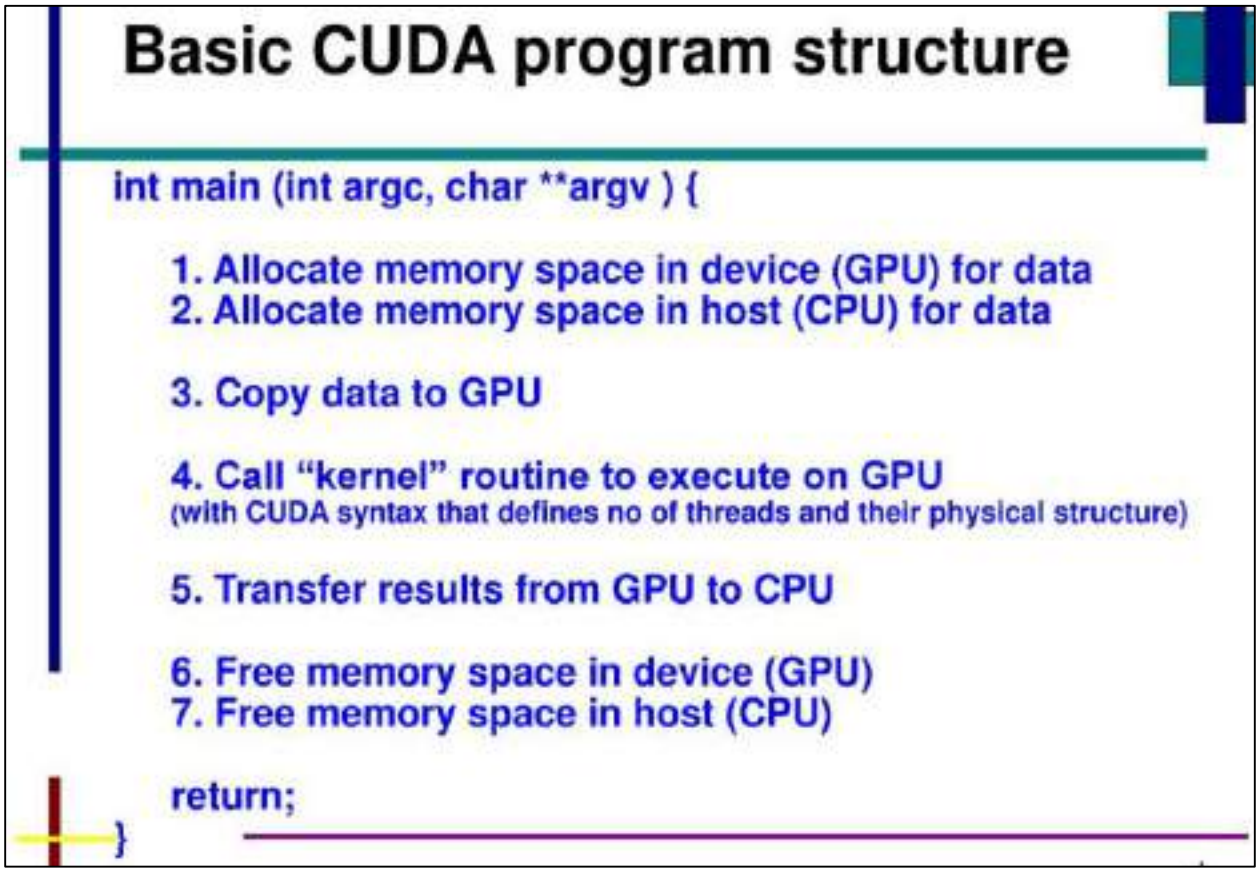
The device code runs on the GPU, and the host code runs on the CPU. The NVCC processes a CUDA program, and separates the host code from the device code. To accomplish this, special CUDA keywords are looked for. The code intended to run of the GPU (device code) is marked with special CUDA keywords for labelling

data-parallel functions, called 'Kernels'. The device code is further compiled by the NVCC and executed on the GPU.





Sequential code		CUDA code	
Step 1	Allocate memory on the CPU, that is, `malloc new`.	Step 1	Allocate memory on the CPU, that is, `malloc new`.
Step 2	Populate/initialize the CPU data.	Step 2	Allocate memory on the GPU, that is, `cudaMalloc`.
Step 3	Call the CPU function that has the crunching of data. The actual algorithm is vector addition in this case.	Step 3	Populate/initialize the CPU data.
Step 4	Consume the crunched data, which is printed in this case.	Step 4	Transfer the data from the host to the device with `cudaMemcpy`.
		Step 5	Call the GPU function with `<<<,>>>` brackets.
		Step 6	Synchronize the device and host with `cudaDeviceSynchronize`.
		Step 7	Transfer data from the device to the host with `cudaMemcpy`.
		Step 8	Consume the crunched data, which is printed in this case.



1. Allocating memory space in "device" (GPU) for data

Use CUDA malloc routines:

```
int size = N *sizeof( int);      // space for N integers

int *devA, *devB, *devC;     // devA, devB, devC ptrs

cudaMalloc( (void**)&devA, size) );
cudaMalloc( (void**)&devB, size );
cudaMalloc( (void**)&devC, size );
```

2. Allocating memory space in "host" (CPU) for data

Use regular C malloc routines:

```
int *a, *b, *c;
...
a = (int*)malloc(size);
b = (int*)malloc(size);
c = (int*)malloc(size);
```

or statically declare variables:

```
#define N 256
...
int  a[N], b[N], c[N];
```

3. Transferring data from host (CPU) to device (GPU)

Use CUDA routine cudaMemcpy

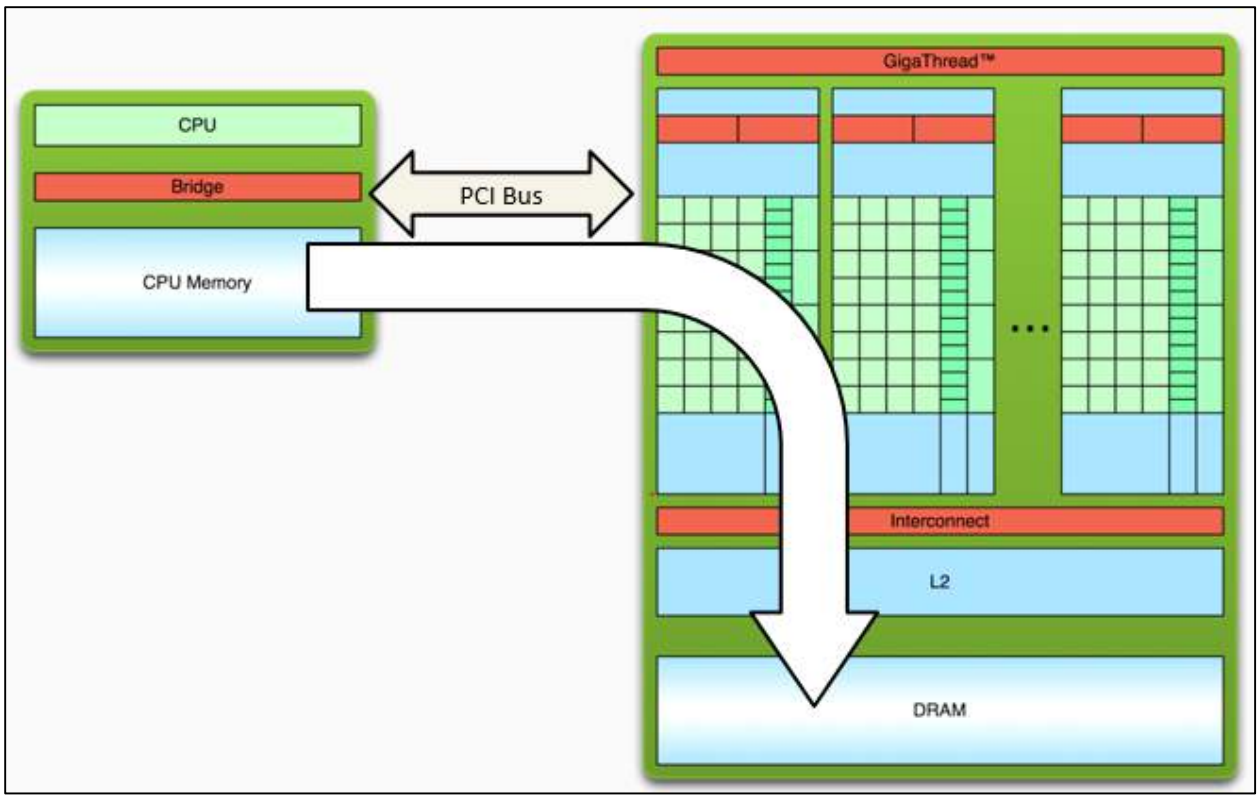
Destination Source

```
cudaMemcpy( devA, a, size, cudaMemcpyHostToDevice);
cudaMemcpy( devB, b, size, cudaMemcpyHostToDevice);
```

where:

devA and **devB** are pointers to destination in device

a and **b** are pointers to host data



4. Declaring "kernel" routine to execute on device (GPU)

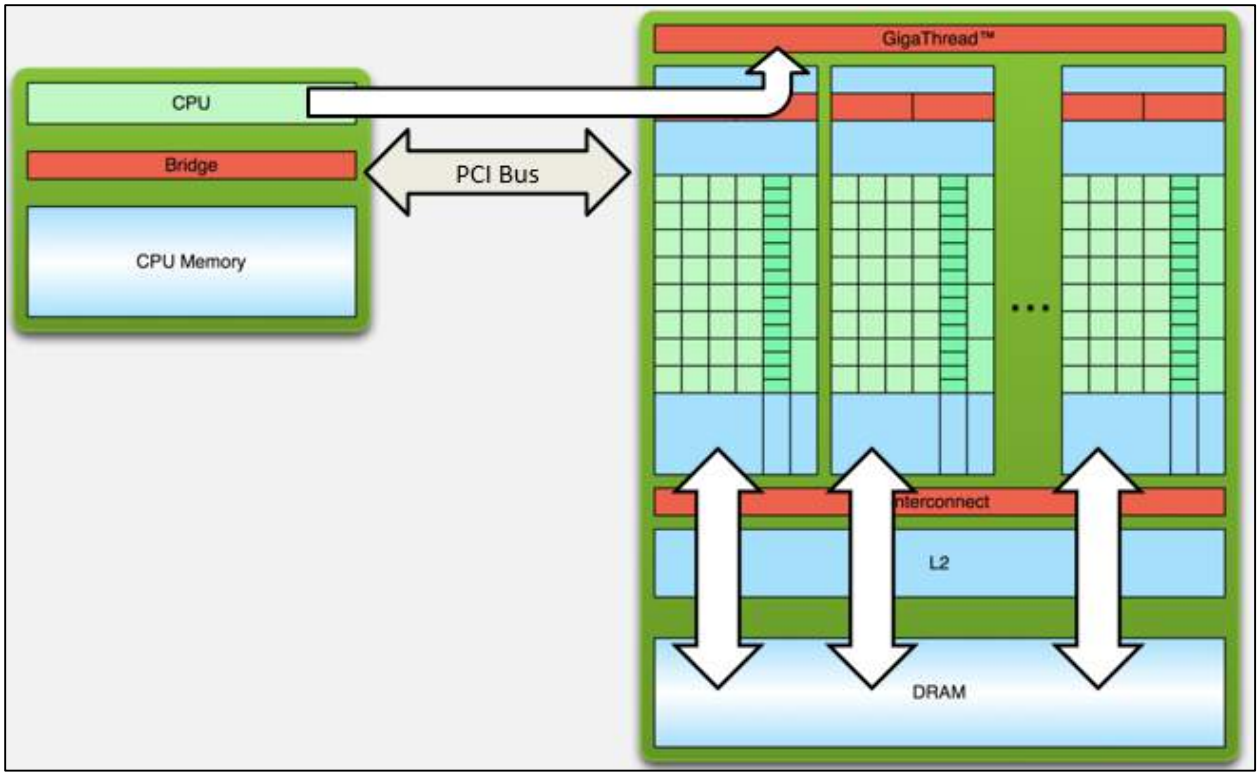
CUDA introduces a syntax addition to C:
Triple angle brackets mark call from host code to device code.
Contains organization and number of threads in two parameters:

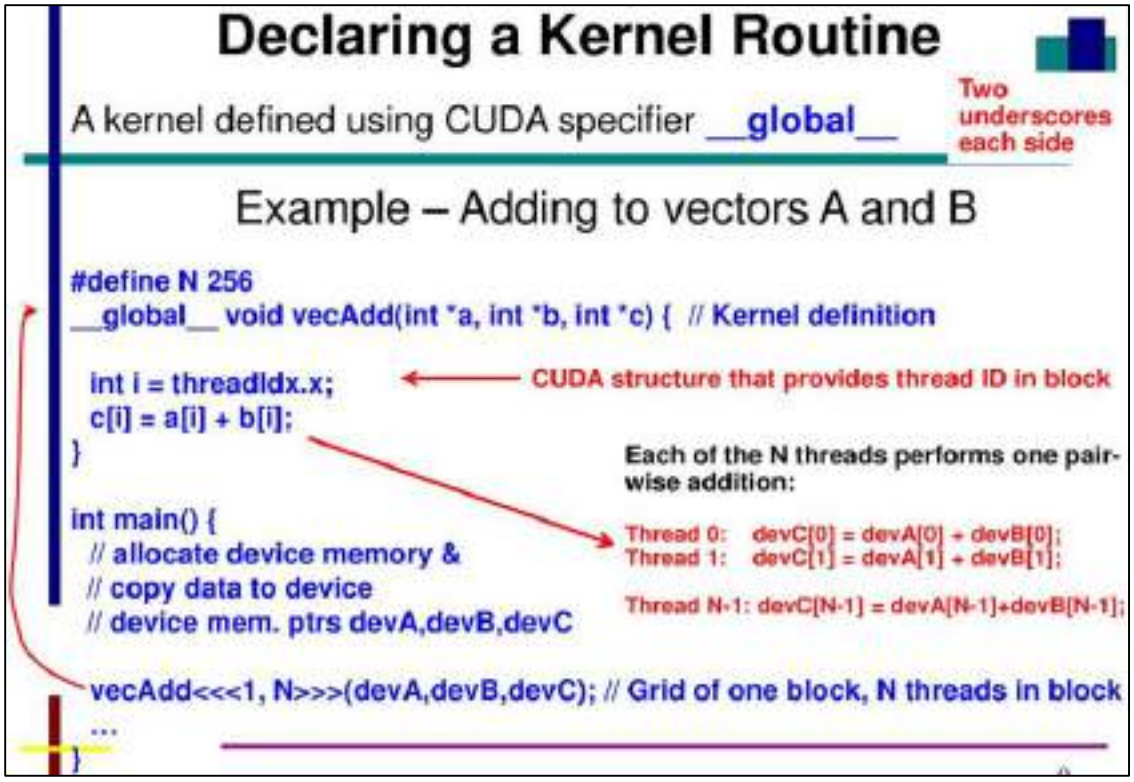
myKernel<<< n, m >>>(arg1, …);

n and **m** will define organization of thread blocks and threads in a block.

For now, we will set **n = 1**, which say one block and **m = N**, which says N threads in this block.

arg1, … , -- arguments to routine **myKernel** typically pointers to device memory obtained previously from **cudaMalloc**.




Declaring a Kernel Routine
A kernel defined using CUDA specifier __global__
Two underscores each side
Example – Adding to vectors A and B
#define N 256
__global__ void vecAdd(int *a, int *b, int *c) { // Kernel definition
int i = threadIdx.x;
CUDA structure that provides thread ID in block
c[i] = a[i] + b[i];
}
Each of the N threads performs one pair-wise addition:
int main() {
// allocate device memory &
// copy data to device
// device mem. ptrs devA,devB,devC
Thread 0: devC[0] = devA[0] + devB[0];
Thread 1: devC[1] = devA[1] + devB[1];
Thread N-1: devC[N-1] = devA[N-1]+devB[N-1];
vecAdd<<<1, N>>>(devA,devB,devC); // Grid of one block, N threads in block
...
}

5. Transferring data from device (GPU) to host (CPU)

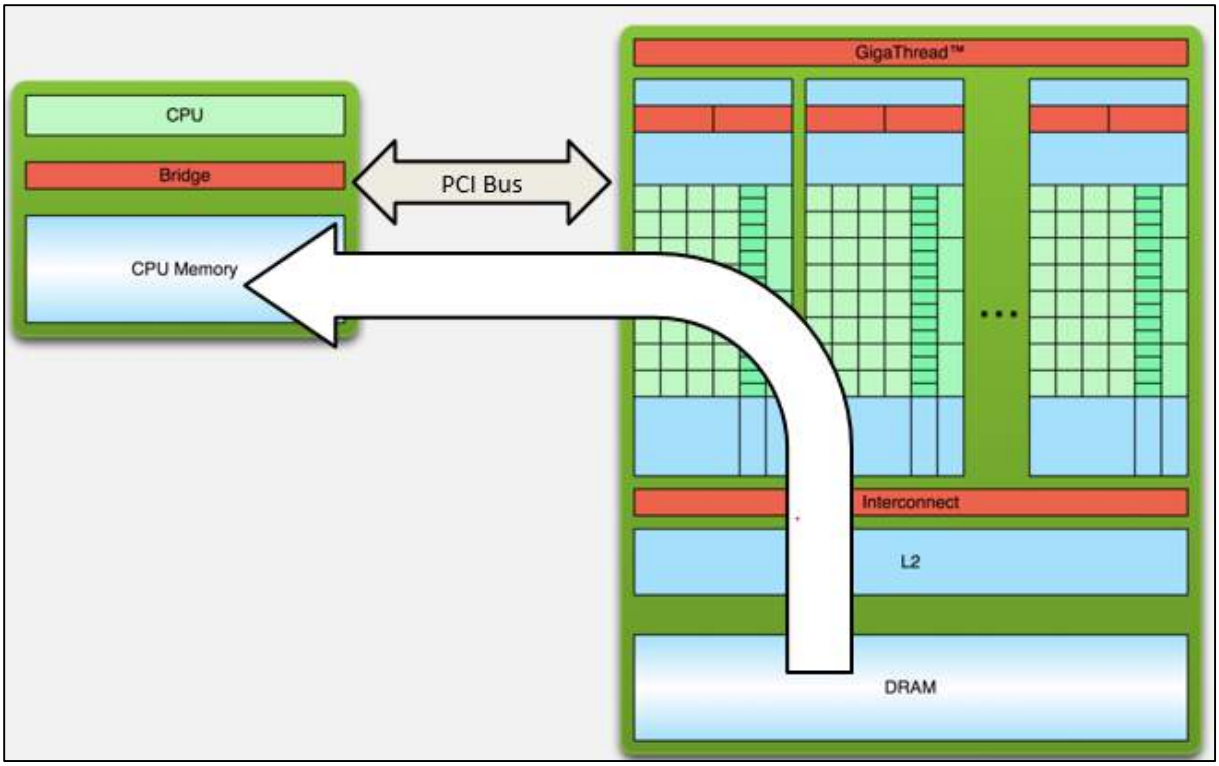
Use CUDA routine cudaMemcpy

Destination Source

cudaMemcpy(c, devC, size, cudaMemcpyDeviceToHost);

where:

devC is a pointer in device and **c** is a pointer in host.



6. Free memory space in "device" (GPU)

Use CUDA cudaFree routine:

```
cudaFree( devA);
cudaFree( devB);
cudaFree( devC);
```

7. Free memory space in (CPU) host

(if CPU memory allocated with malloc)

Use regular C free routine to deallocate memory if previously allocated with malloc:

```
free( a );
free( b );
free( c );
```

Complete CUDA program

Adding two vectors, A and B

N elements in A and B, and

N threads

(without code to load arrays with data)

```
#define N 256

__global__ void vecAdd(int *A, int *B, int *C) {
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

int main (int argc, char **argv ) {

  int size = N *sizeof( int);
  int  a[N], b[N], c[N], *devA, *devB, *devC;

  cudaMalloc( (void**)&devA, size) );
  cudaMalloc( (void**)&devB, size );
  cudaMalloc( (void**)&devC, size );

  cudaMemcpy( devA, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy( devB, b size, cudaMemcpyHostToDevice);

  vecAdd<<<1, N>>>(devA, devB, devC);

  cudaMemcpy( c, devC size, cudaMemcpyDeviceToHost);

  cudaFree( devA);
  cudaFree( devB);
  cudaFree( devC);

  return (0);
}
```

Compiling CUDA programs "nvcc"

NVIDIA provides nvcc -- the NVIDIA CUDA "compiler driver".

Will separate out code for host and for device

Regular C/C++ compiler used for host (needs to be available)

Programmer simply uses nvcc instead of gcc/cc compiler on a Linux system

Command line options include for GPU features

Compiling code - Linux

Command line:

Directories for #include files

nvcc –O3 –o <exe> <source_file> -I/usr/local/cuda/include

–L/usr/local/cuda/lib –lcuda –lcudart

Optimization level if you want optimized code

Directories for libraries

(Dynamic) CUDA libraries to be linked (core and runtime, both needed)

CUDA source file that includes device code has the extension .cu

Need regular C compiler installed for CPU.

Make file convenient – see next.

Compilation process

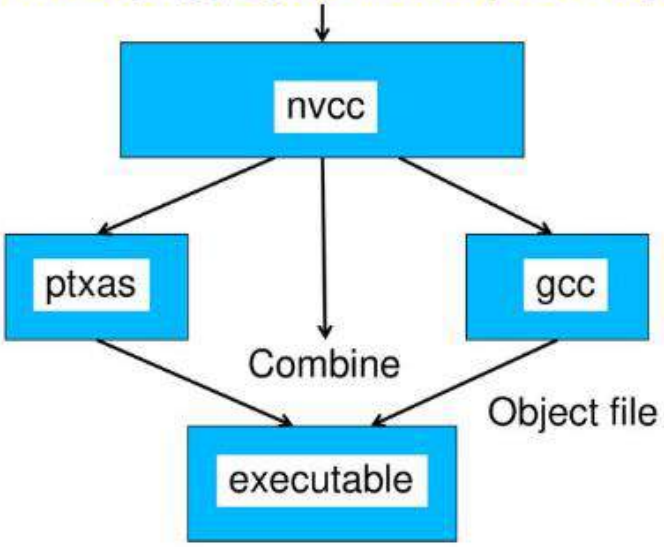
nvcc "wrapper" divides code into host and device parts.

Host part compiled by regular C compiler

Device part compiled by NVIDIA "ptxas" assembler

Two compiled parts combined into one executable

nvcc –o prog prog.cu –I/includepath -L/libpath



Executable file a "fat" binary" with both host and device code

17

Executing Program

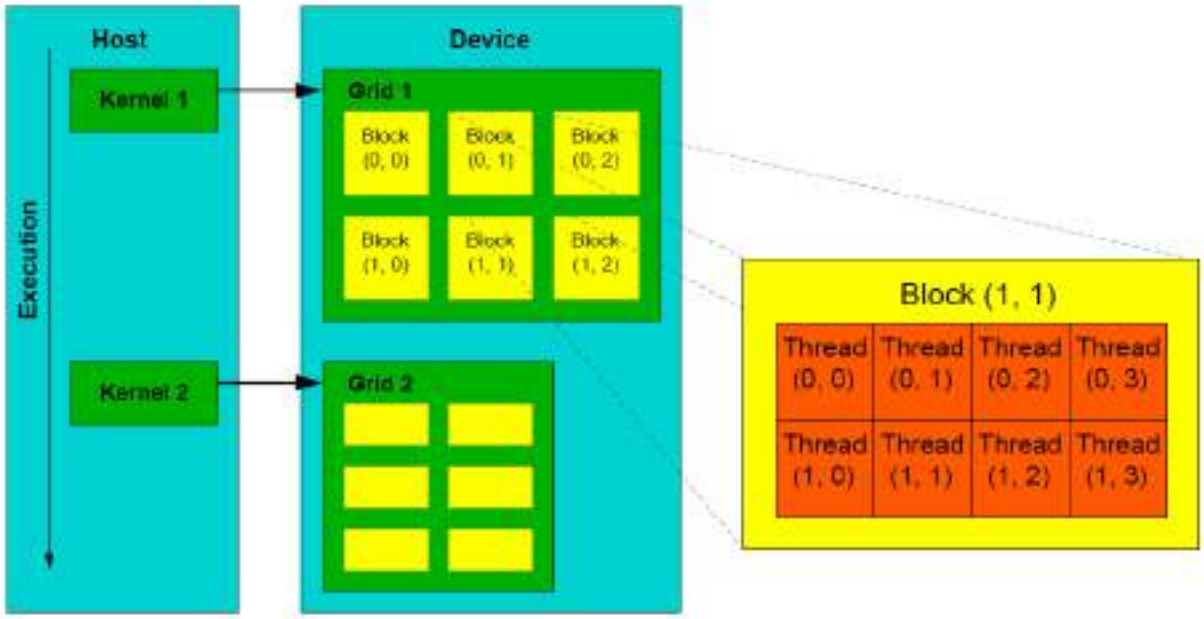
Simple type name of executable created by nvcc:

./prog1

File includes all the code for host and for device in a "fat binary" file.

Host code starts running

When first encounter device kernel, GPU code physically sent to GPU and function launched on GPU.



Compiling and executing on a Windows system

Can use Microsoft Visual Studio and a PC with a NVIDIA GPU card.

Basic set up described in
"Configuring Microsoft Visual Studio 2008 for CUDA Tookit Version 3.2," B. Wilkinson and Brian Nacey, Feb 24, 2012, found at

http://coitweb.uncc.edu/~abw/SIGCSE2011Workshop/ConfiguringVSforCUDA.pdf

but NVIDA now provides a fully configured NVIDIA Nsight Visual Studio Edition found at

http://developer.nvidia.com/nvidia-nsight-visual-studio-edition

and Eclipse version found at

http://developer.nvidia.com/nsight-eclipse-edition

- **__global__:** This keyword, when added before the function, tells the compiler that this is a function that will run on the device and not on the host. However, note that it is called by the host. Another important thing to note here is that the return type of the device function is always "void". Data-parallel portions of an algorithm are executed on the device as kernels.

- **<<<,>>>:** This keyword tells the compiler that this is a call to the device function and not the host function. Additionally, the 1,1 parameter basically dictates the number of threads to launch in the kernel. We will cover the parameters inside angle brackets later. For now, the 1,1 parameter basically means we are launching the kernel with only one thread, that is, sequential code with a thread since we are not doing anything important in the code apart from printing.
- **ThreadIdx.x, blockIdx.x:** This is a unique ID that's given to all threads
- **cudaDeviceSynchronize():** All of the kernel calls in CUDA are asynchronous in nature. The host becomes free after calling the kernel and starts executing the next instruction afterward. This should come as no big surprise since this is a heterogeneous environment and hence both the host and device can run in parallel to make use of the types of processors that are available. In case the host needs to wait for the device to finish, APIs have been provided as part of CUDA programming that make the host code wait for the device function to finish. One such API is cudaDeviceSynchronize, which waits until all of the previous calls to the device have finished.

CUDA processing flow has some additional steps that need to be added to the sequential code. These are as follows:

1. **Memory allocation on GPU:** CPU memory and GPU memory are physically separate memory. *malloc* allocates memory on the CPU's RAM. The GPU kernel/device function can only access memory that's allocated/pointing to the device memory. To allocate memory on the GPU, we need to use the *cudaMalloc* API. Unlike the *malloc* command, *cudaMalloc* does not return a pointer to allocated memory; instead, it takes a pointer reference as a parameter and updates the same with the allocated memory.
2. **Transfer data from host memory to device memory:** The host data is then copied to the device's memory, which was allocated using the *cudaMalloc* command used in the previous step. The API that's used to copy the data between the host and device and vice versa is cudaMemcpy. Like other *memcopy* commands, this API requires the destination pointer, source pointer, and size. One additional parameter it takes is the direction of copy, that is, whether we are copying from the host to the device or from the device

to the host. In the latest version of CUDA, this is optional since the driver is capable of understanding whether the pointer points to the host memory or device memory. Note that there is an asynchronous alternative to *cudaMemcpy*.

3. **Call and execute a CUDA function:** As shown in the Hello World CUDA program, we call a kernel by using <<<,>>> brackets, which provide parameters for the block and thread size, respectively.
4. **Synchronize:** As we mentioned in the Hello World program, kernel calls are asynchronous in nature. In order for the host to make sure that kernel execution has finished, the host calls the *cudaDeviceSynchronize* function. This makes sure that all of the previously launched device calls have finished.
5. **Transfer data from host memory to device memory:** Use the same *cudaMemcpy* API to copy the data back from the device to the host for post-processing or validation duties such as printing. The only change here, compared to the first step, is that we reverse the direction of the copy, that is, the destination pointer points to the host while the source pointer points to the device allocated in memory.
6. **Free the allocated GPU memory:** Finally, free the allocated GPU memory using *cudaFree* API.

5B.3 Matrix multiplication kernel

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

__global__ void matmul(const float *A, const float *B, float *C, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    float sum = 0;
    if (i < n && j < n) {
        for (int k = 0; k < n; k++) {
            sum += A[i * n + k] * B[k * n + j];
        }
        C[i * n + j] = sum;
    }
}

int main(void) {
    int n = 1024;
    size_t size = n * n * sizeof(float);
    float *A, *B, *C;
    cudaMalloc((void **)&A, size);
    cudaMalloc((void **)&B, size);
    cudaMalloc((void **)&C, size);
    dim3 block(32, 32);
    dim3 grid((n + block.x - 1) / block.x, (n + block.y - 1) / block.y);
    matmul<<<grid, block>>>(A, B, C, n);
    cudaFree(A);
    cudaFree(B);
    cudaFree(C);
    return 0;
}
```

This code performs matrix multiplication of two n x n matrices A and B, storing the result in C. The computation is performed by the matmul CUDA kernel, which is launched on the GPU with grid and block dimensions specified. The memory for matrices A, B, and C is dynamically allocated on the GPU using cudaMalloc( ) and freed using cudaFree( )

5B. 4 Data transfer

After memory is allocated, we need to copy data from host to device buffer and back. This is done using the cudaMemcpy(..) function

__host__ cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)

Both copy to and from the device buffer are done using the same function and the direction of the copy is specifies by the last argument, which is cudaMemcpyKind enumeration. The enumeration can take values cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice or cudaMemcpyDefault. All but the last are self-explanatory. Passing the cudaMemcpyDefault will make the API to deduce the direction of the transfer from pointer values, but require unified virtual addressing. Second to last argument is the size of the data to be copied in bytes. The first two arguments can be either host or device pointers, depending on the directionality of the transfer. This is where using h_ and d_ prefixes come handy: this way we should only remember the order in which the destination and the source arguments are specified. For instance, host to device copy call should look something like that:

```
cudaMemcpy(d_x, h_x, numElements*sizeof(float), cudaMemcpyHostToDevice);
```

The names of the buffers suggest that the first argument (destination) is the device buffer and the second argument is the host buffer (source). This means that we are executing host to device copy, which is specified byt the last argument. After the execution on the device is done, we have the data in the device memory and the results can be copied back to the host memory using:

```
cudaMemcpy(h_x, d_x, numElements*sizeof(float), cudaMemcpyDeviceToHost);
```

.

5B. 5 Kernel functions and threading

CUDA kernel is a function that gets executed on GPU. The parallel portion of your applications is executed *K* times in parallel by *K* different CUDA threads, as opposed to only one time like regular C/C++ functions.

Every CUDA kernel starts with a `__global__` declaration specifier. Programmers provide a unique global ID to each thread by using built-in variables.

A group of threads is called a CUDA block. CUDA blocks are grouped into a grid. A kernel is executed as a grid of blocks of threads

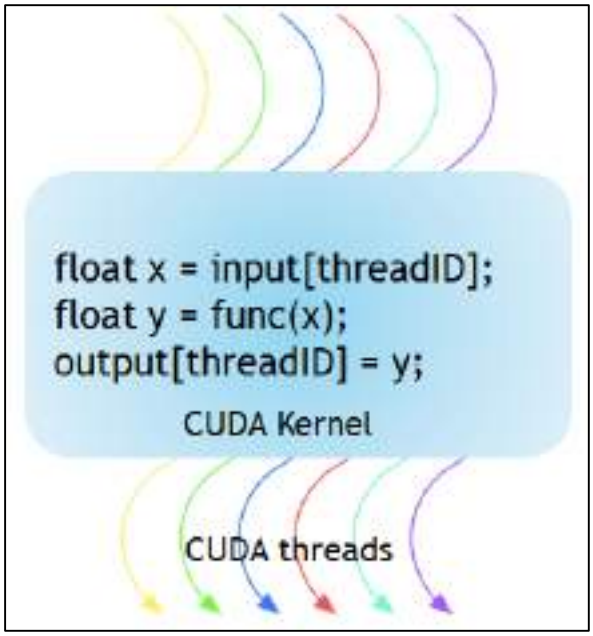
Each CUDA block is executed by one streaming multiprocessor (SM) and cannot be migrated to other SMs in GPU (except during preemption, debugging, or CUDA dynamic parallelism). One SM can run several concurrent CUDA blocks depending on the resources needed by CUDA blocks. Each kernel is executed on one device and CUDA supports running multiple kernels on a device at one time.

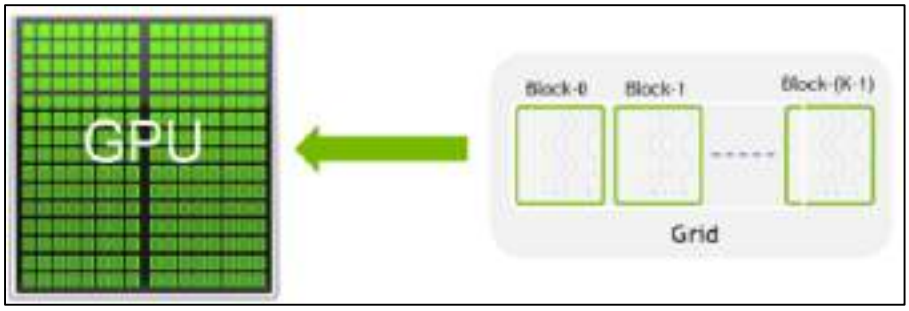
CUDA defines built-in 3D variables for threads and blocks. Threads are indexed using the built-in 3D variable `threadIdx`. Three-dimensional indexing provides a natural way to index elements in vectors, matrix, and volume and makes CUDA programming easier. Similarly, blocks are also indexed using the in-built 3D variable called `blockIdx`.

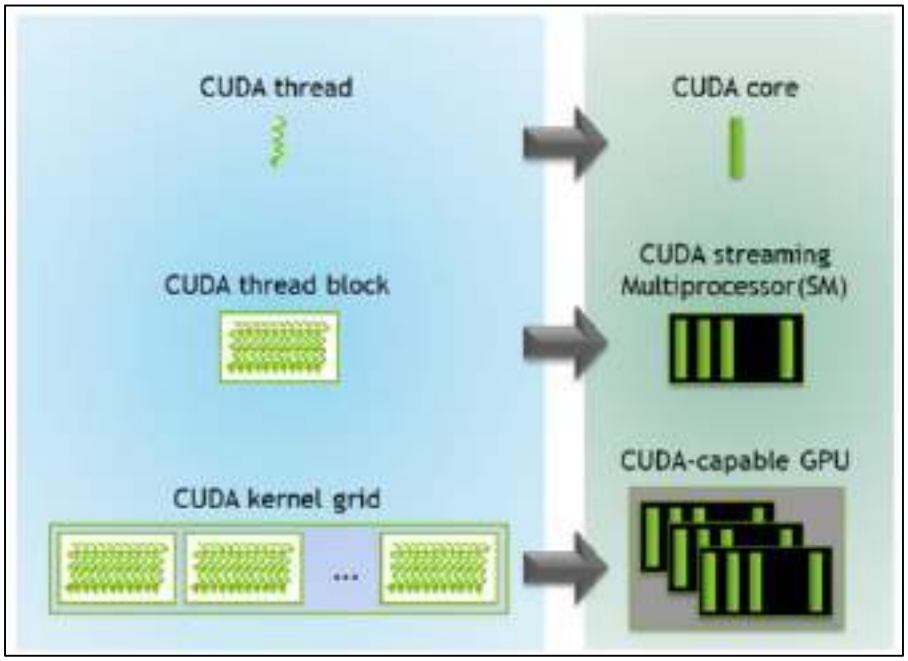
Here are a few noticeable points:

- CUDA architecture limits the numbers of threads per block (1024 threads per block limit).
- The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.
- All threads within a block can be synchronized using an intrinsic function `__syncthreads`. With `__syncthreads`, all threads in the block must wait before anyone can proceed.
- The number of threads per block and the number of blocks per grid specified in the <<<…>>> syntax can be of type int or dim3. These triple angle brackets mark a call from host code to device code. It is also called a kernel launch.

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.


float x = input[threadID];
float y = func(x);
output[threadID] = y;
CUDA Kernel
CUDA threads


GPU
Block-0
Block-1
Block-(K-1)
Grid


CUDA thread
CUDA core
CUDA thread block
CUDA streaming Multiprocessor(SM)
CUDA kernel grid
CUDA-capable GPU

5B.6 CUDA Thread organization and synchronization

Threads in a grid execute the same kernel function. They have specific coordinates to distinguish themselves from each other and identify the relevant portion of data to process. In CUDA, they are organized in a two-level hierarchy: a grid comprises blocks, and each block comprises threads.

For all threads in a block, the block index is the same. The block index parameter can be accessed using the **blockIdx** variable inside a kernel. Each thread also has an associated index, and it can be accessed by using **threadIdx** variable inside the kernel. Note that blockIdx and threadIdx are built-in CUDA variables that are only accessible from inside the kernel.

In a similar fashion, CUDA also has **gridDim** and **blockDim** variables that are also built-in. They return the dimensions of the grid and block along a particular axis respectively. As an example, **blockDim**. x can be used to find how many threads a particular block has along the x axis.

e image from sRGB to grayscale. We can calculate the total number of pixels by multiplying the number of pixels along the x axis with the total number along the y axis that comes out to be 4712 pixels. Since we are mapping each thread with each pixel, we need a minimum of 4712 pixels. Let us take number of threads in each direction to be a multiple of 4. So, along the x axis, we will need at least 80 threads, and along the y axis, we will need at least 64 threads to process the complete image. We will ensure that the extra threads are not assigned any work.

Thus, we are launching 5120 threads to process a 4712 pixels image. You may ask, why the extra threads? The answer to this question is that keeping the dimensions as multiple of 4 has many benefits that largely offsets any disadvantages that result from launching extra threads.

Now, we have to divide these 5120 threads into grids and blocks. Let each block have 256 threads. If so, then one possibility that of the dimensions each block are: (16,16,1). This means, there are 16 threads in the x direction, 16 in the y direction, and 1 in the z direction. We will be needing 5 blocks in the x direction (since there are 80 threads in total along the x axis), and 4 blocks in y direction (64 threads along the y axis in total), and 1 block in z direction. So, in total, we need 20 blocks. In a nutshell, the grid dimensions are (5,4,1) and the block dimensions are (16,16,1). The programmer needs to specify these values in the program. This is shown in the figure above.

- dim3 dimBlock(5,4,1) − To specify the grid dimensions
- dim3 dimGrid(ceil(n/16.0),ceil(m/16.0),1) − To specify the block dimensions.
- kernelName<<<dimGrid,dimBlock>>>(parameter1, parameter2, ...) − Launch the actual kernel.

n is the number of pixels in the x direction, and m is the number of pixels in the y direction. **'ceil'** is the regular ceiling function. We use it because we never want to end up with less number of blocks than required. **dim3** is a data structure, just like an int or a float. **dimBlock** and **dimGrid** are variables names. The third statement is the kernel launch statement. 'kernelName' is the name of the kernel function, to which we pass the parameters: parameter1, parameter2, and so on. <<<>>> contain the dimensions of the grid and the block.

CUDA Thread Synchronization

The CUDA API has a method, __**syncthreads()** to synchronize threads. When the method is encountered in the kernel, all threads in a block will be blocked at the calling location until each of them reaches the location.

What is the need for it? It ensure phase synchronization. That is, all the threads of a block will now start executing their next phase only after they have finished the previous one. There are certain nuances to this method. For example, if a __syncthreads statement, is present in the kernel, it must be executed by all threads of a block. If it is present inside an if statement, then either all the threads in the block go through the if statement, or none of them does.

If an if-then-else statement is present inside the kernel, then either all the threads will take the **if** path, or all the threads will take the else path. This is implied. As all the threads of a block have to execute the sync method call, if threads took different paths, then they will be blocked forever.

It is the duty of the programmer to be wary of such conditions that may arise.

5B.7 Querying Device properties

The following CUDA code demonstrates a more general approach, calculating the theoretical peak bandwidth by querying the attached device (or devices) for the needed information.

This code uses the function **cudaGetDeviceCount()** which returns in the argument nDevices the number of CUDA-capable devices attached to this system. Then in a loop we calculate the theoretical peak bandwidth for each device.The body of the loop uses **cudaGetDeviceProperties()** to populate the fields of the variable prop, which is an instance of the struct **cudaDeviceProp**. The program uses only three of cudaDeviceProp's many members: name, memoryClockRate, and memoryBusWidth.

```
#include <stdio.h>

int main() {
  int nDevices;

  cudaGetDeviceCount(&nDevices);
  for (int i = 0; i < nDevices; i++) {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, i);
    printf("Device Number: %d\n", i);
    printf("  Device name: %s\n", prop.name);
    printf("  Memory Clock Rate (KHz): %d\n",
           prop.memoryClockRate);
    printf("  Memory Bus Width (bits): %d\n",
           prop.memoryBusWidth);
    printf("  Peak Memory Bandwidth (GB/s): %f\n\n",
           2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
  }
}
```

Output:

```
Device Number: 0
  Device name: Tesla C2050
  Memory Clock Rate (KHz): 1500000
  Memory Bus Width (bits): 384
  Peak Memory Bandwidth (GB/s): 144.00
```

There are many other fields in the cudaDeviceProp struct which describe the amounts of various types of memory, limits on thread block sizes, and many other characteristics of the GPU.

5B.8 CUDA Memory types

A CUDA device has a number of different memory components that are available to programmers - register, shared memory, local memory, global memory and constant memory. Figure 6 illustrates how threads in the CUDA device can access the different memory components. In CUDA only threads and the host can access memory.

Memory Access

We use one and two-way arrows to indicate read (R) and write (W) capability. An arrow pointing toward a memory component indicates write capability; an arrow pointing away from a memory component indicates read capability. For example, global memory and constant memory can be read (R) or write (W).

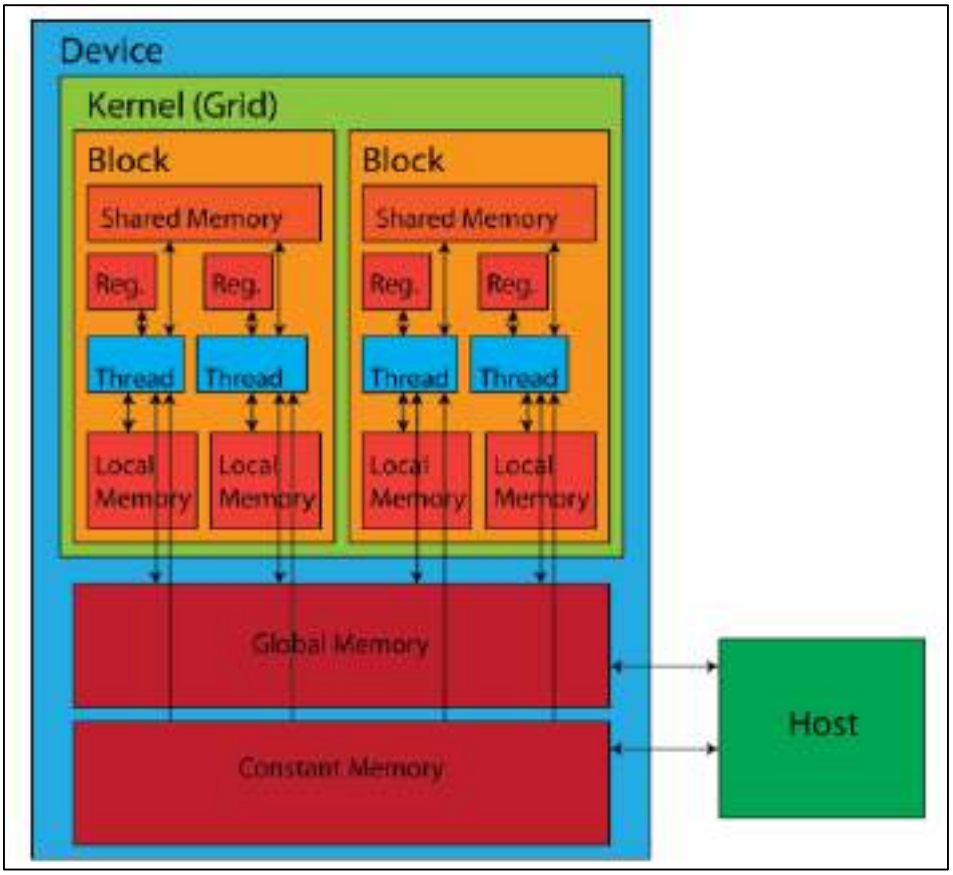
On a CUDA device, multiple kernels can be invoked. Each kernel is an independent grid consisting of one or more blocks. Each block has its own per-block shared memory, which is shared among the threads within that block. All threads can access (R/W) different parts of memory on the device that are summarized below:

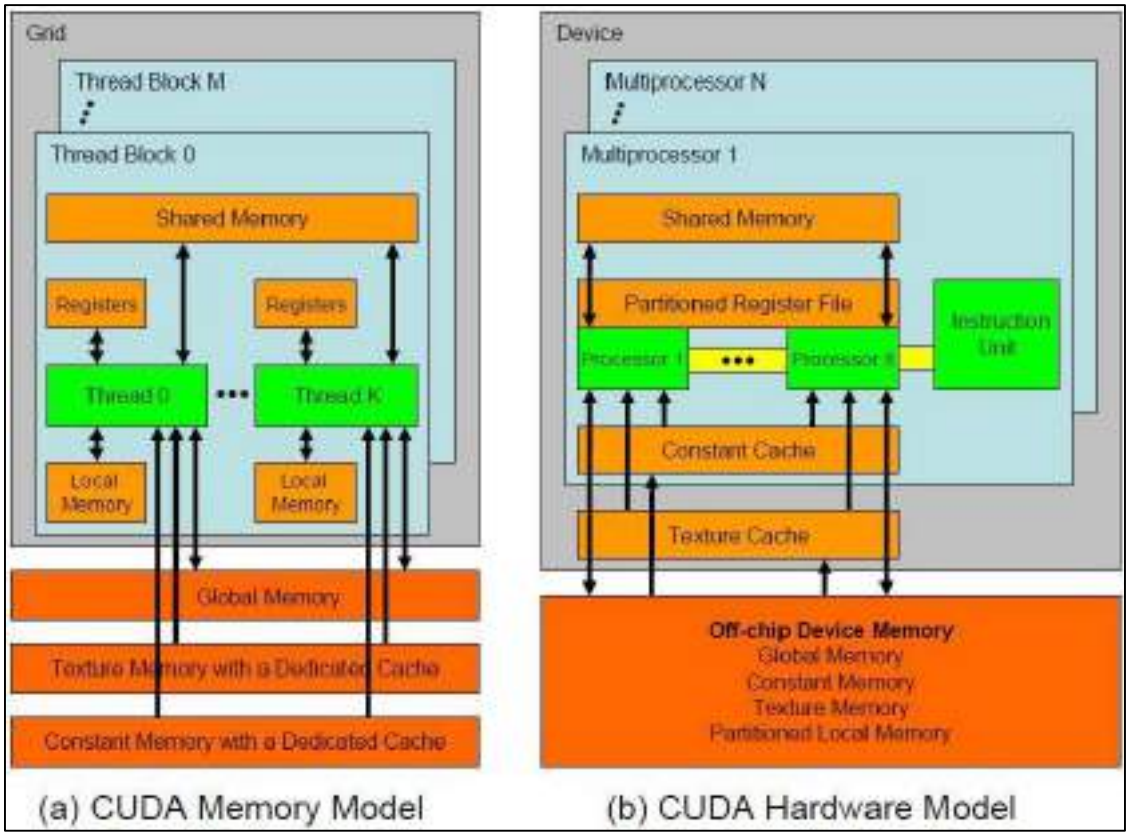
In general, we use the host to:

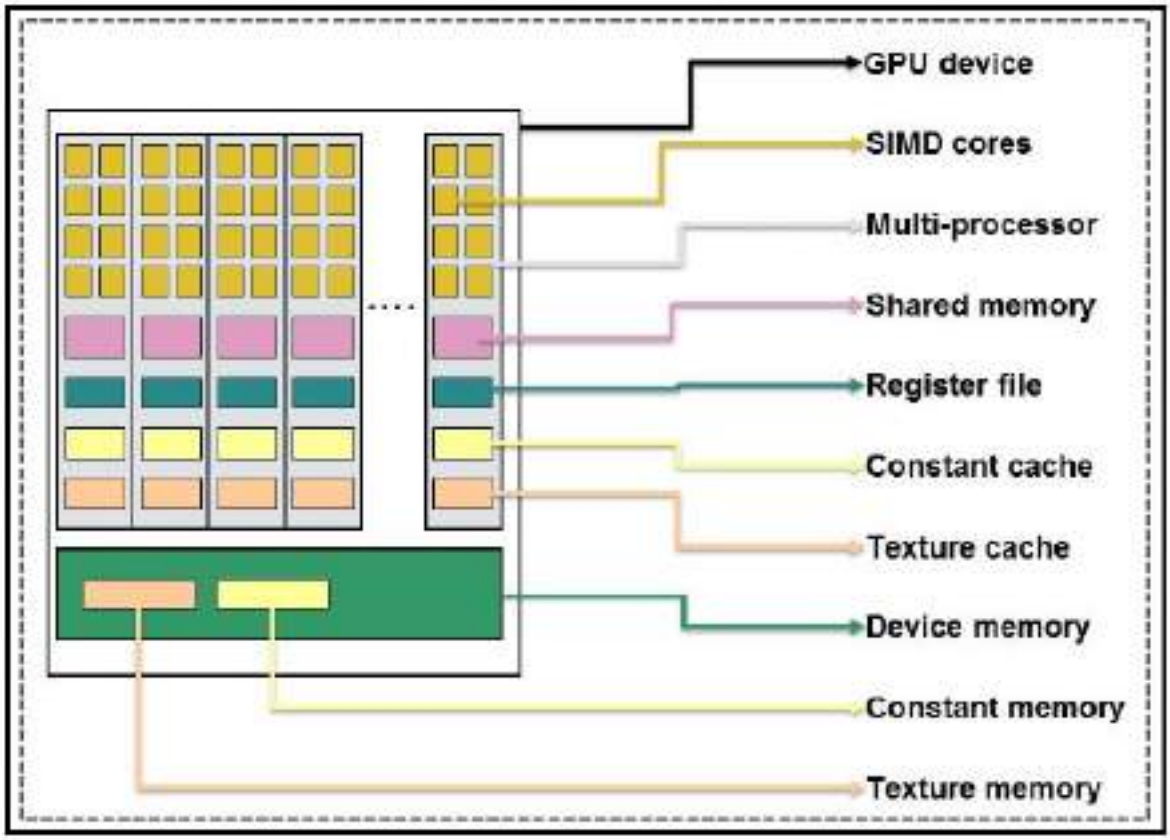
- Transfer data to and from global memory
- Transfer data to and from constant memory

Once the data is in the device memory, our threads can read and write (R/W) different parts of memory:

- R/W per-thread register
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- R per-grid constant memory


Device
Kernel (Grid)
Block
Block
Shared Memory
Shared Memory
Reg.
Reg.
Reg.
Reg.
Thread
Thread
Thread
Thread
Local Memory
Local Memory
Local Memory
Local Memory
Global Memory
Constant Memory
Host


Grid
Thread Block M
Thread Block 0
Shared Memory
Registers
Registers
Thread 0
Thread K
Local Memory
Local Memory
Global Memory
Texture Memory with a Dedicated Cache
Constant Memory with a Dedicated Cache
(a) CUDA Memory Model
Device
Multiprocessor N
Multiprocessor 1
Shared Memory
Partitioned Register File
Instruction Unit
Processor 1
Processor K
Constant Cache
Texture Cache
Off-chip Device Memory
Global Memory
Constant Memory
Texture Memory
Partitioned Local Memory
(b) CUDA Hardware Model


GPU device
SIMD cores
Multi-processor
Shared memory
Register file
Constant cache
Texture cache
Device memory
Constant memory
Texture memory

References

- https://cvw.cac.cornell.edu/gpu/structure#:~:text=A%20CUDA%20program%20is%20comprised,completely%20independent%20of%20host%20execution.
- https://www.tutorialspoint.com/cuda/cuda_key_concepts.htm
- CUDA Programming Model - ppt download (slideplayer.com)
- https://cvw.cac.cornell.edu/gpu/memory_arch
- CUDA - Keywords and Thread Organization (tutorialspoint.com)
- https://www.youtube.com/watch?v=yvSm9a8LYX0
- https://www.youtube.com/watch?v=OSpy-HoR0ac
- https://www.tutorialspoint.com/cuda/cuda_threads.htm
- Han, Jaegeun, and Bharatkumar Sharma. *Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10. x and C/C++*. Packt Publishing Ltd, 2019.