Unit-1

Dr.S.Nagadevi
Department of Computing Technologies
SRM Institute of Science and Technology
Chennai

GPU

- Graphics Processing Unit
- Single-Chip MicroProcessor
- Dedicated processor for 3D graphics to represent 3D world realistically
- Additional power
  - single-chip processor for mathematically-intensive tasks
  - transforms of vertices and polygons
  - Lighting, polygon clipping, texture mapping, polygon rendering

- Graphics Cards--- gaming etc..

WHY GPU?

- Manipulating and displaying computer graphics---Virtual Processing Unit(VPU)
- Reduces the burden of CPUs
- Additional computational power that is customized to perform 3D tasks

New trends of microprocessors

- Since 2003, there has been two main trajectories for microprocessor design
- *Multicore* – a relatively small number of cores per chip, each core is a full-flesh processor in the "traditional sense"
- *Many-core* – a large number of much smaller and simpler cores
  - NVIDIA GeForce GTX 280 GPU (graphics processing unit) has 240 cores, each is heavily multi-threaded, in-order, single-instruction issue processor. Eight cores share control and instruction cache.
  - As of 2009 peak performance of many-core GPUs is at around 10 fold the peak performance of multicore CPUs
  - GPUs have larger memory bandwidth (simpler memory models and fewer legacy requirements)

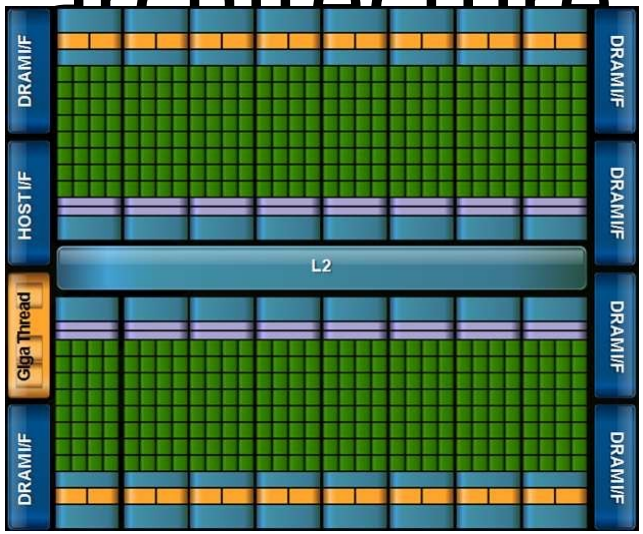
Early use of GPU for computing

- Design of GPU is shaped by video game industry, ability to perform a massive number of floating-point (single precision) calculations per video frame
- Full 3D pipeline: transform, lighting, rasterization, texturing, depth testing and display
- Computing on the earlier GPU architectures had to cast computing as graphics operations
  - GeForce 3 in 2001 – programmable pixel shading
  - Later GeForce products – separate programmable engines for vertex and geometry shading

GPGPU

- General-purpose GPU – capable of performing non-graphics processing
  - running shader code against data presented as vertex or texture information
  - computing results retrieved at later stage in the pipeline still "awkward programming" compared with CPU
- "Unified shader architecture" – each shader core can be assigned with any shader task, no need for stage-by-stage balancing
  - GeForce 8800 in 2006 (128 processing elements distributed among 8 shader cores)
  - Tesla product line – graphics cards without display outputs and drivers optimized for GPU computing instead of 3D rendering

NVIDIA's Fermi architecture



- Designed for GPU computing (graphics-specific bits largely omitted)
- 16 streaming multiprocessors (SMs)
- 32 CUDA cores (streaming processors) in each SM (512 cores in total)
- Each streaming processor is massively threaded
- 6 DRAM 64-bit memory interfaces
- GigaThread scheduler
- Peak double-precision floating-point rate: 768 GFLOPs

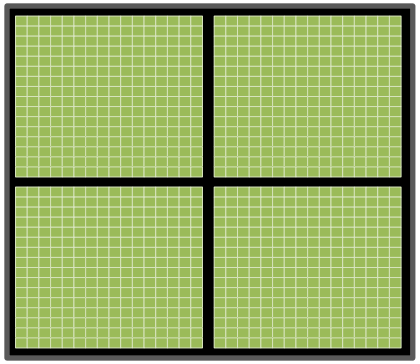
Massive (but simple) parallelism

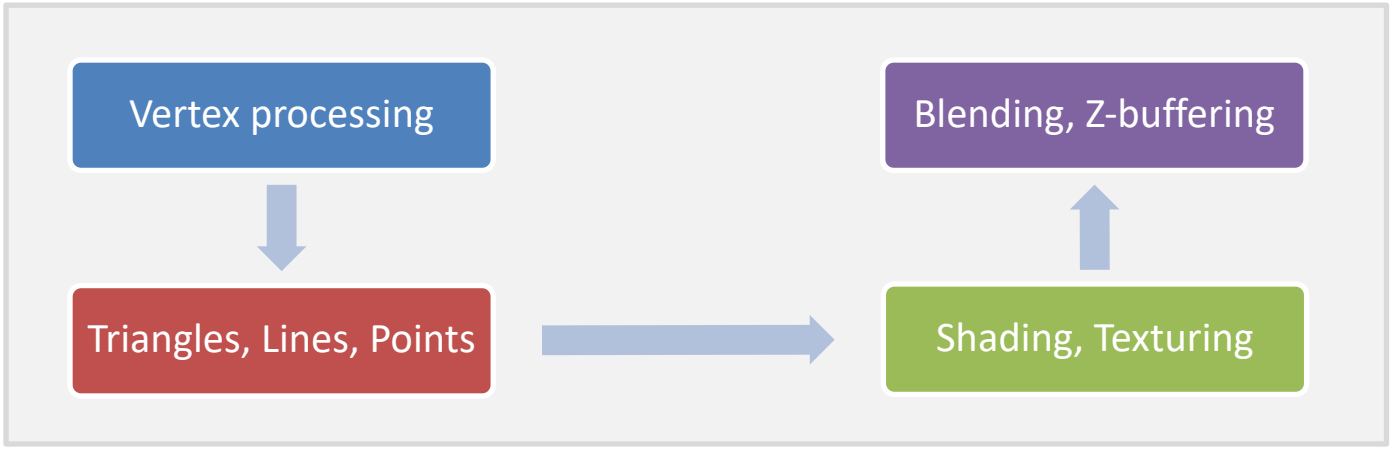
- One streaming processor is the most fundamental execution resource
- Simpler than a CPU core
- But capable of executing a large number of threads simultaneously, where the threads carry out same instruction to different data elements —- single-instruction-multiple-data (SIMD)
- A number of streaming processors constitute one streaming multiprocessor (SM)
- On Nvidia's GT200, 1024 threads per SM, up to about 30,000 threads simultaneously

CPU computing vs. GPU computing

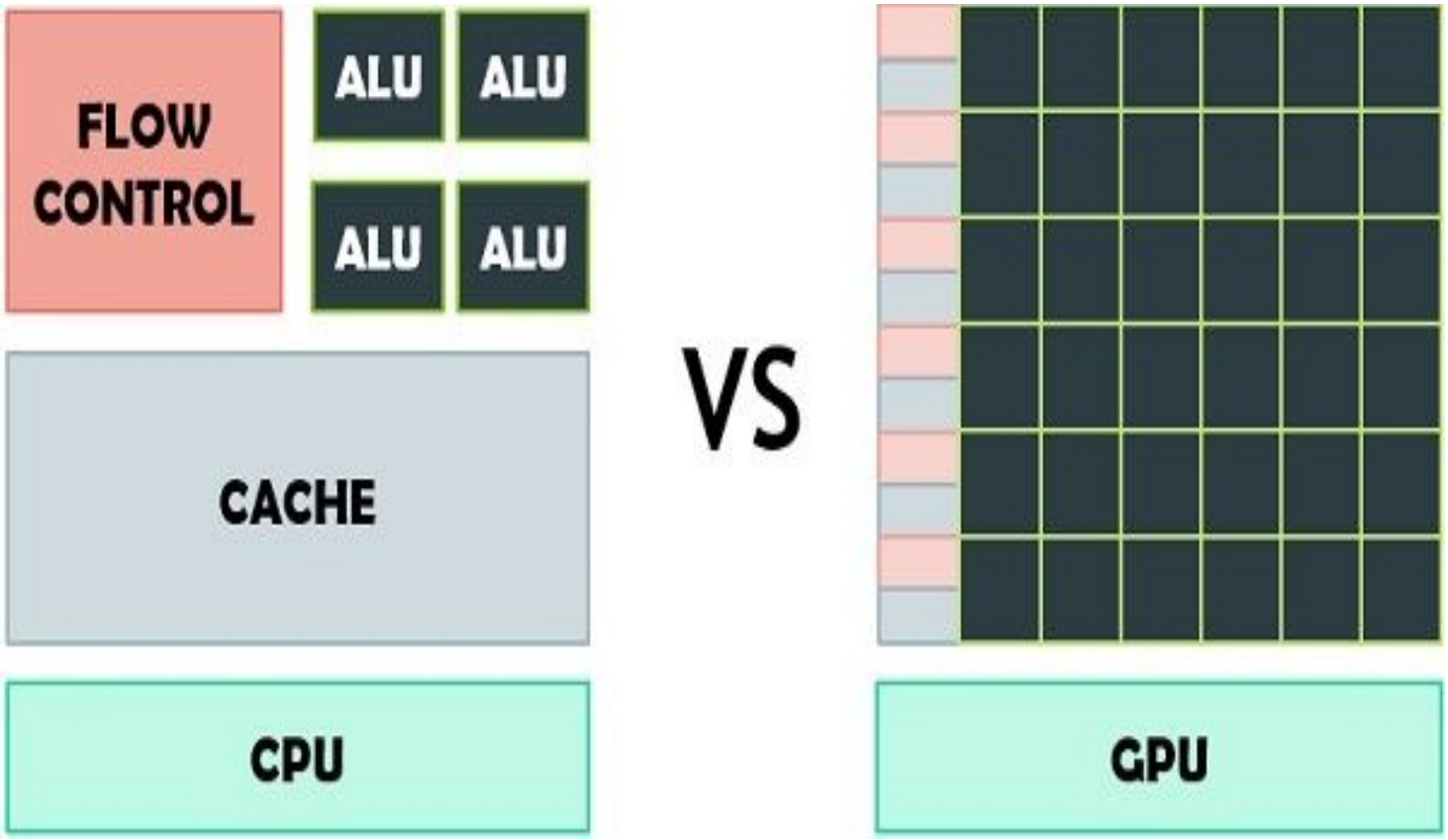
- CPUs are good for applications where most of the work is done by a small number of threads, where the threads have high data locality, a mixture of different operations and conditional branches
- GPU aims at the “other end of the spectrum”
  - data parallelism – many arithmetic operations performed on data structures in a simultaneous manner
  - applications with multiple threads that are dominated by longer sequences of computational instructions
  - computationally intensive (not control-flow intensive)
- GPU computing will not replace CPU computing

Traditional GPU workflow

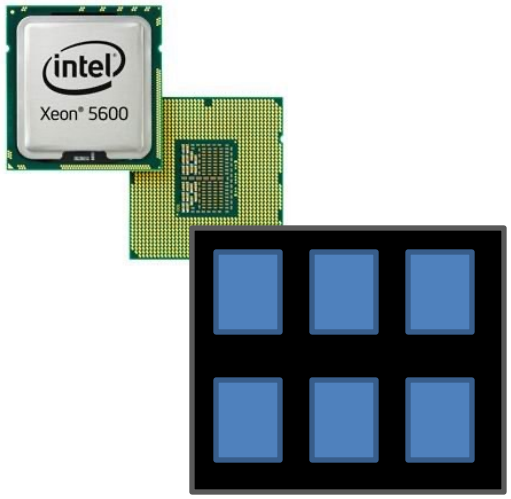




Difference Between CPU and GPU

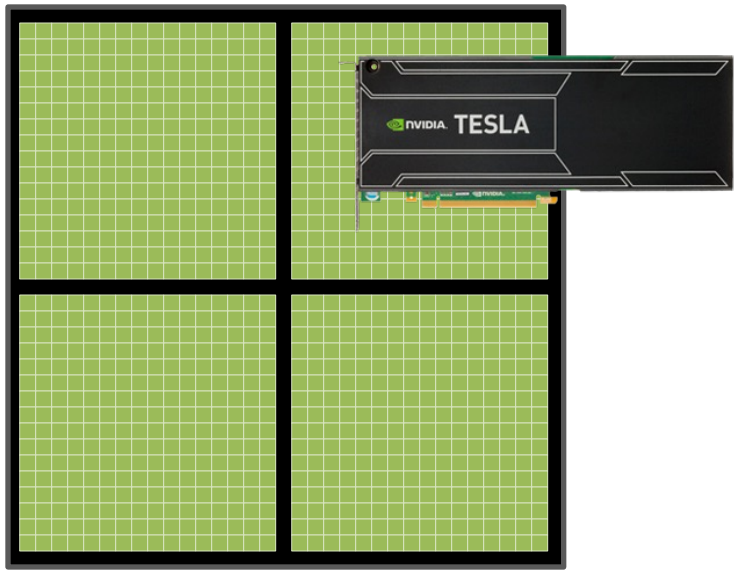


CPU



CPUs consist of a few cores optimized for serial processing

GPU



GPUs consist of hundreds or thousands of smaller, efficient cores designed for parallel performance

Difference Between CPU and GPU

CPU	GPU
A smaller number of larger cores (up to 24)	A larger number (thousands) of smaller cores
Low latency	High throughput
Optimized for serial processing	Optimized for parallel processing
Designed for running complex programs	Designed for simple and repetitive calculations
Performs fewer instructions per clock	Performs more instructions per clock
Automatic cache management	Allows for manual memory management
Cost-efficient for smaller workloads	Cost-efficient for bigger workloads

Why GPUs are faster?

- GPU originally specialized for math-intensive, highly parallel computation
- So, more transistors can be devoted to data processing rather than data caching and flow control

Will Execution on a GPU Accelerate My Application?

Computationally intensive—The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory.

Massively parallel—The computations can be broken down into hundreds or thousands of independent units of work.

Components of GPU

- Graphics Processor
- Graphics co-processor
- Graphics accelerator
- Frame buffer
- Memory
- Graphics BIOS
- Digital-to-Analog Converter (DAC)
- Display Connector
- Computer (Bus) Connector

GPU Architecture

How many processing units?

– Lots.

How many ALUs?

– Hundreds.

Do you need a cache?

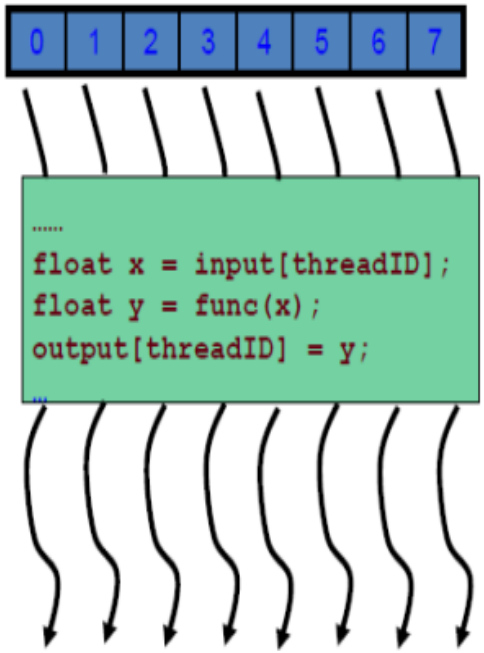
– Sort of.

What kind of memory?

– very fast.

GPU Computing: Think in Parallel

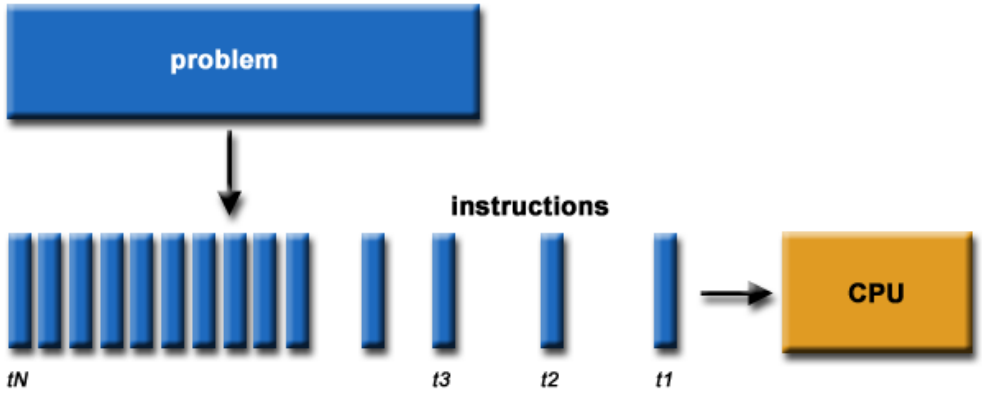
- Scale to 100's of cores, 1000's of parallel threads
- Let programmers focus on parallel algorithms & Re-writing the Code
- Not on the mechanics of a parallel programming language
- Enable heterogeneous systems (i.e. CPU + GPU)
- CPU and GPU are separate devices with separate DRAMs



Traditional Computing

Von Neumann architecture: instructions are sent from memory to the CPU

Serial execution: Instructions are executed one after another on a single Central Processing Unit (CPU)



Problems:

- More expensive to produce
- More expensive to run
- Bus speed limitation

Parallel Computing

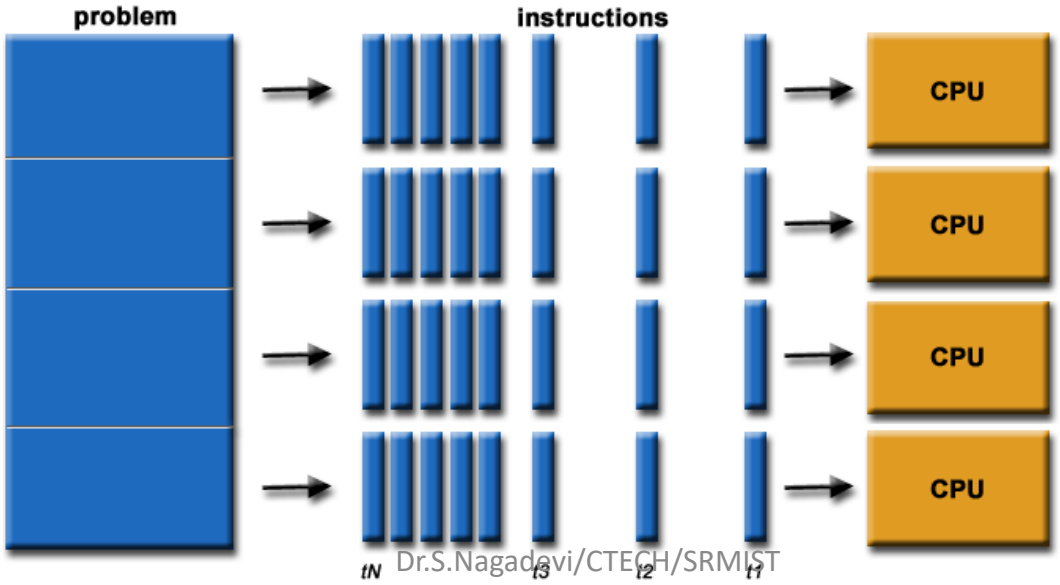
Official-sounding definition: The simultaneous use of multiple compute resources to solve a computational problem.

Benefits:

- Economical – requires less power and cheaper to produce
- Better performance – bus/bottleneck issue

Limitations:

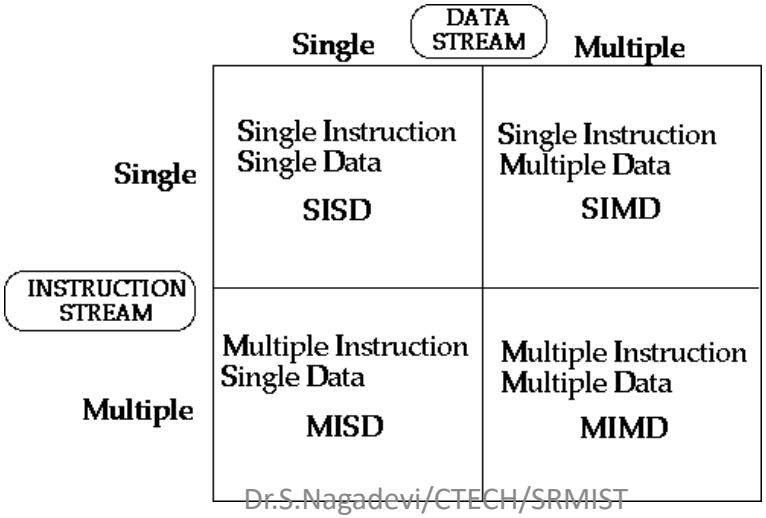
- New architecture – Von Neumann is all we know!
- New debugging difficulties – cache consistency issue



Flynn's Taxonomy

Classification of computer architectures, proposed by Michael J. Flynn

- SISD – traditional serial architecture in computers.
- SIMD – parallel computer. One instruction is executed many times with different data (think of a for loop indexing through an array)
- MISD - Each processing unit operates on the data independently via independent instruction streams. Not really used in parallel
- MIMD – Fully parallel and the most common form of parallel computing.



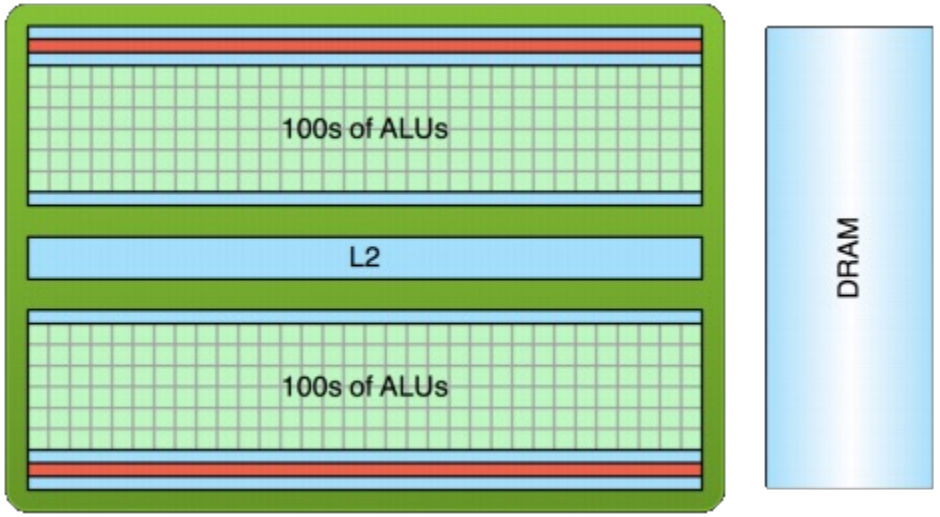
GPU computing

GPU: Graphics Processing Unit

Traditionally used for real-time rendering of computer graphics

High Computational density and memory bandwidth

Throughput processor: 1000s of concurrent threads to hide latency



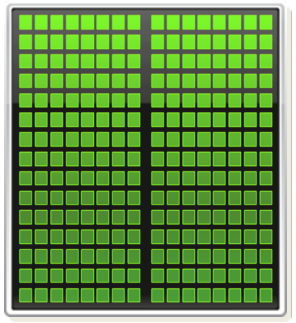
- GPU – graphics processing unit
- Originally designed as a graphics processor
- NVIDIA GeForce 256 (1999) – first GPU
  - single-chip processor for mathematically-intensive tasks
  - transforms of vertices and polygons
  - lighting
  - polygon clipping
  - texture mapping
  - polygon rendering

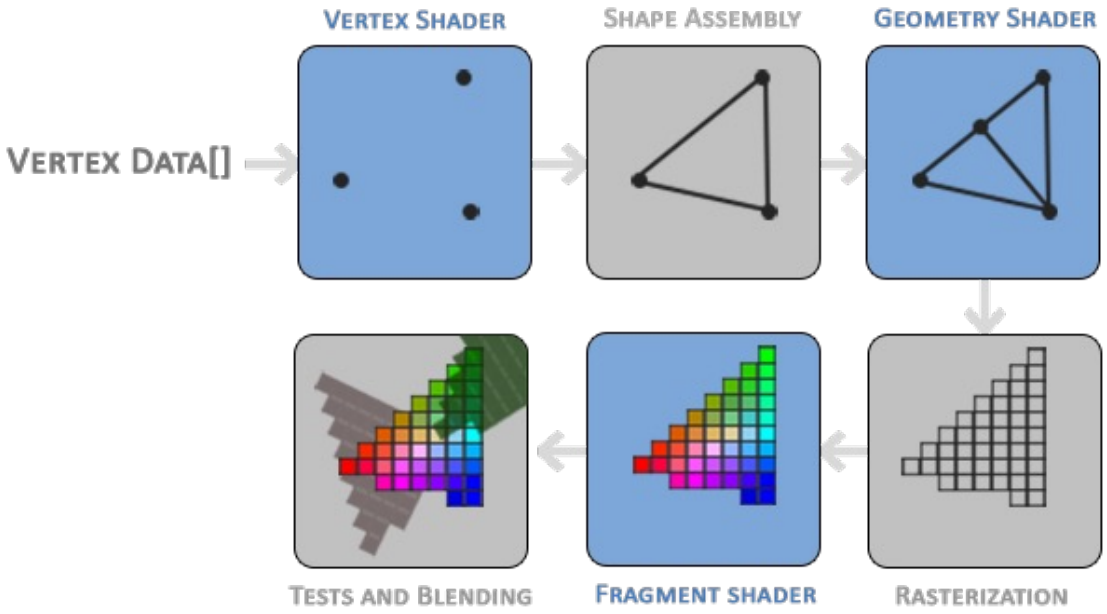
- NVIDIA Geforce 3, ATI Radeon 9700 – early 2000's
  - Now programmable!

Modern GPUs are present in

- ✓ Embedded systems
- ✓ Personal Computers
- ✓ Game consoles
- ✓ Mobile Phones
- ✓ Workstations

Traditional GPU workflow





https://learnopengl.com/#!Getting-started/Hello-Triangle

GPGPU

1999-2000 computer scientists from various fields started using GPUs to accelerate a range of scientific applications.

GPU programming required the use of graphics APIs such as OpenGL and Cg.

2001 – LU factorization implemented using GPUs

2002 James Fung (University of Toronto) developed OpenVIDIA.

NVIDIA greatly invested in GPGPU movement and offered a number of options and libraries for a seamless experience for C, C++ and Fortran programmers.

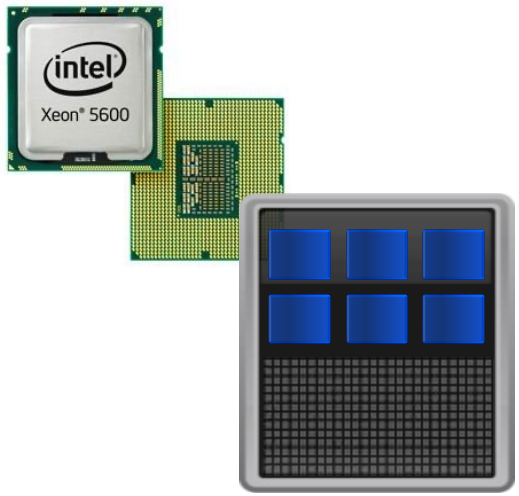
GPGPU timeline

In November 2006 NVIDIA launched CUDA, an API that allows to code algorithms for execution on GeForce GPUs using the C programming language.

Khronus Group defined OpenCL in 2008 supported on AMD, NVIDIA and ARM platforms.

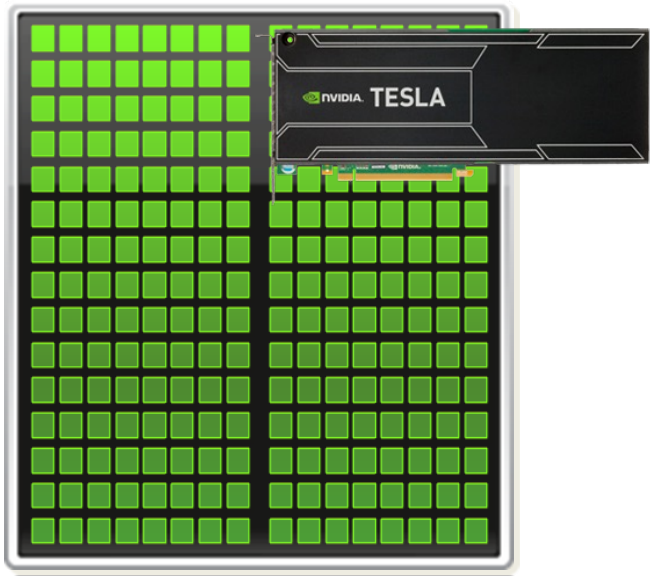
In 2012 NVIDIA presented and demonstrated OpenACC - a set of directives that greatly simplify parallel programming of heterogeneous systems.

CPU



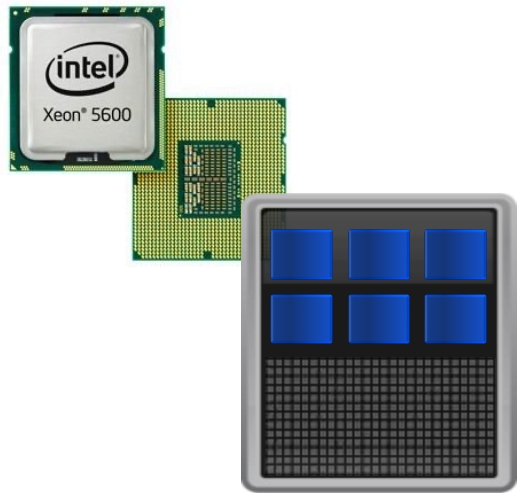
CPUs consist of a few cores optimized for serial processing and general purpose calculations.

GPU

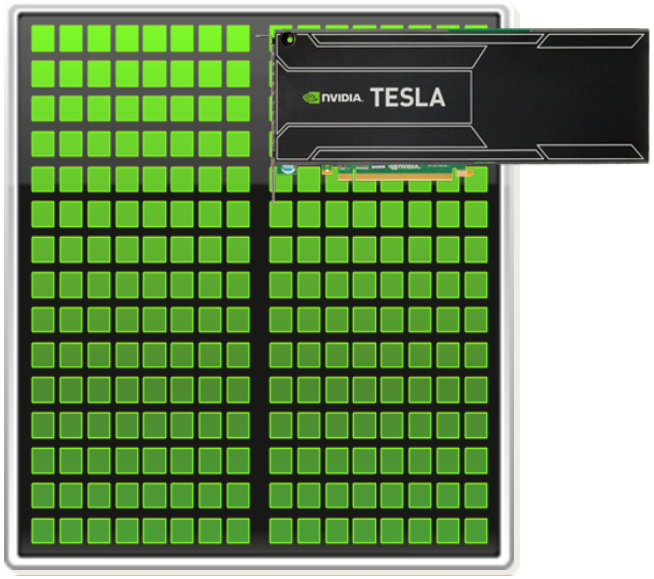


GPUs consist of hundreds or thousands of smaller, efficient cores designed for parallel performance. The hardware is designed for specific calculations.

SCC CPU



SCC GPU



Intel Xeon E5-2680v4:

Clock speed: **2.4** GHz
4 instructions per cycle with AVX2
CPU - 28 cores

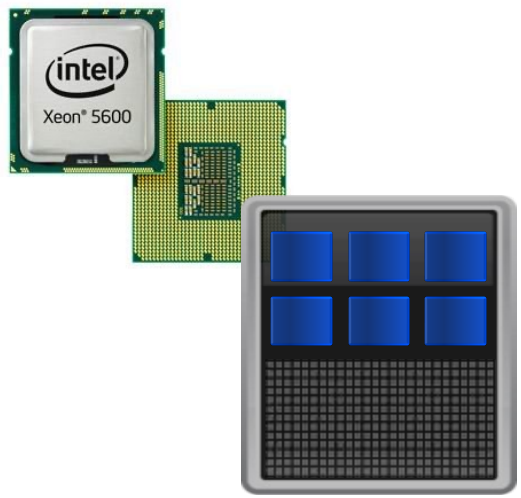
2.4 x 4 x 28 =
268.8 Gigaflops double precision

NVIDIA Tesla P100:

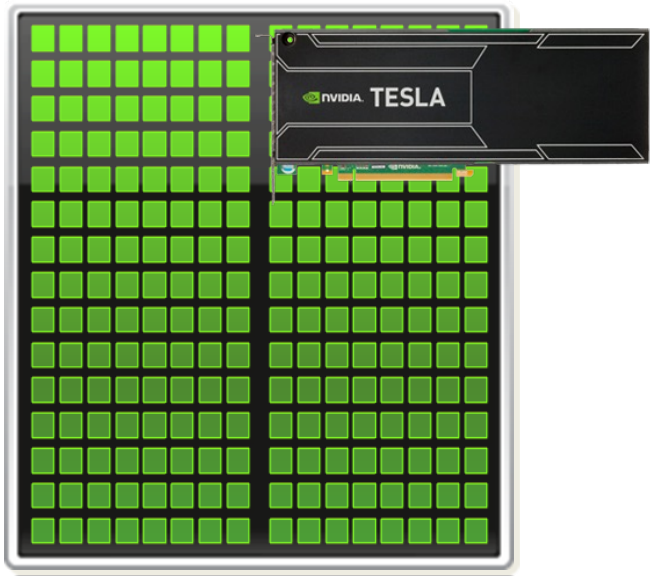
Single instruction per cycle
3584 CUDA cores

4.7 Teraflops double precision

SCC CPU



SCC GPU



Intel Xeon E5-2680v4 :

Memory size: **256** GB
Bandwidth: **76.8** GB/sec

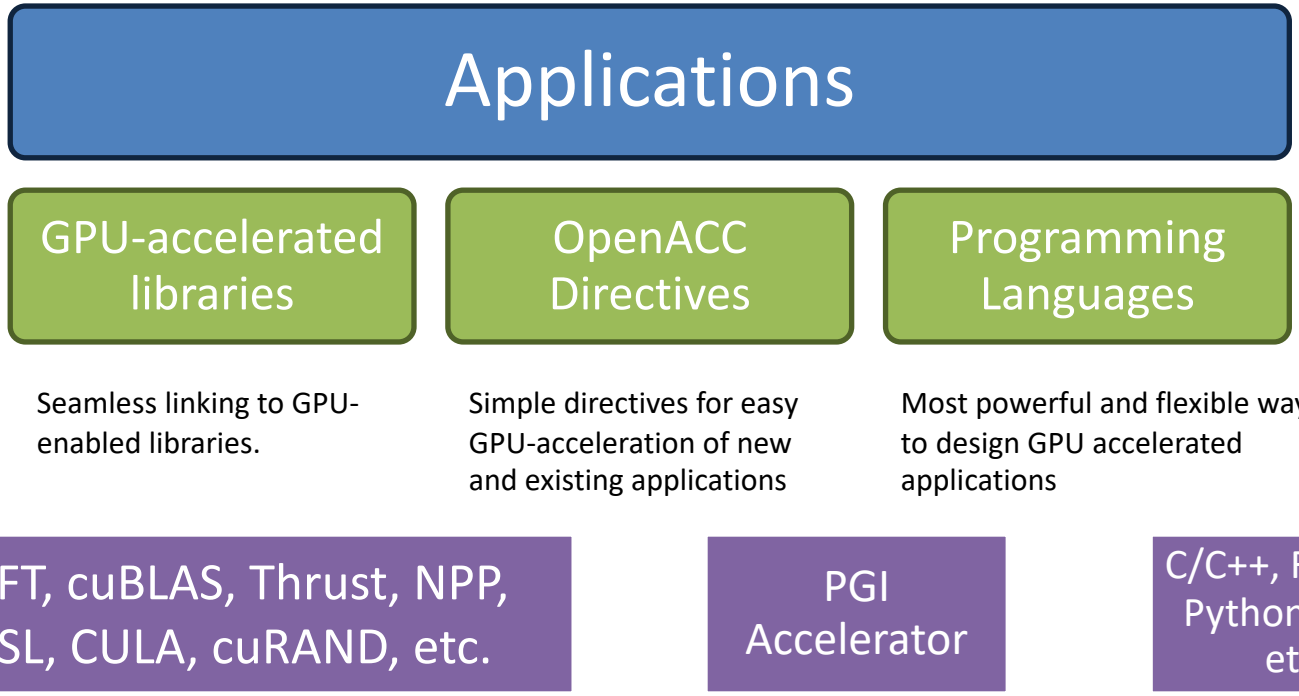
NVIDIA Tesla P100:

Memory size: **12GB** total
Bandwidth: **549** GB/sec

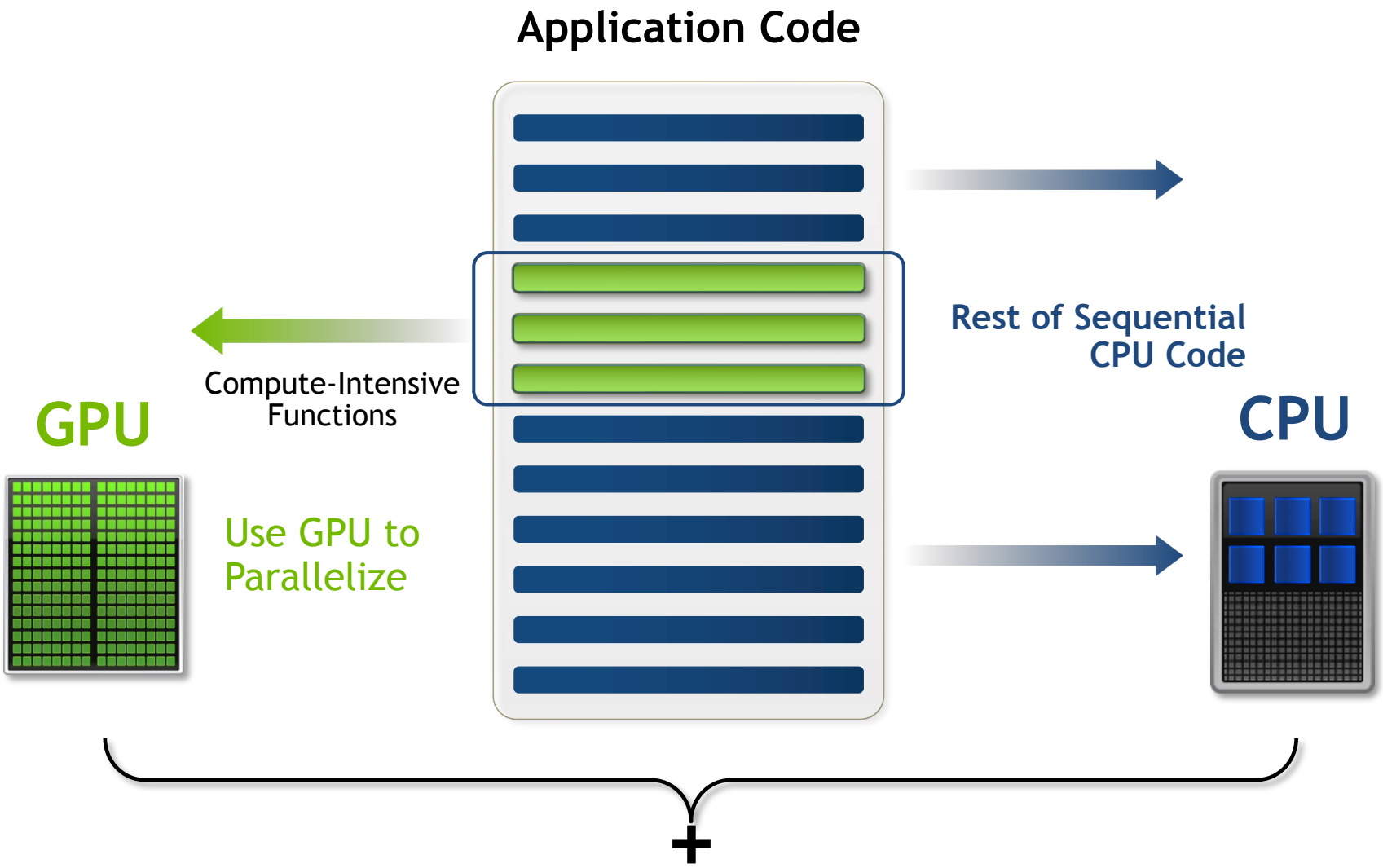
10x GPU Computing Growth

2008	2015
6,000 Tesla GPUs	450,000 Tesla GPUs
150K CUDA downloads	3M CUDA downloads
77 Supercomputing Teraflops	54,000 Supercomputing Teraflops
60 University Courses	800 University Courses
4,000 Academic Papers	60,000 Academic Papers

GPU Acceleration



Minimum Change, Big Speed-up



Will Execution on a GPU Accelerate My Application?

Computationally intensive—The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory.

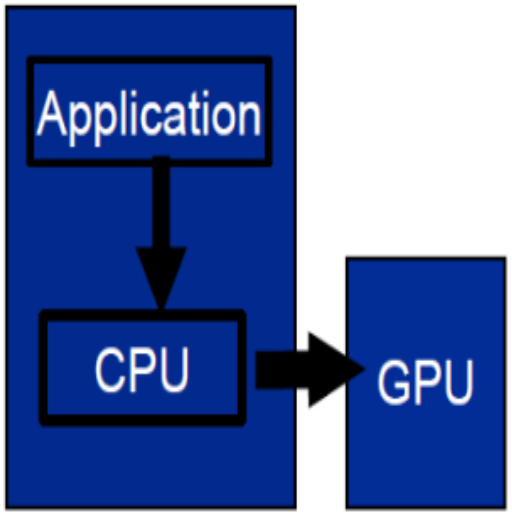
Massively parallel—The computations can be broken down into hundreds or thousands of independent units of work.

Well suited to GPU architectures – some algorithms or implementations will not perform well on the GPU.

C	OpenACC, CUDA
C++	Thrust, CUDA C++
Fortran	OpenACC, CUDA Fortran
Python	PyCUDA, PyOpenCL
Numerical analytics	MATLAB, Mathematica
Machine Learning	Theano, Tensorflow, Caffe, Torch, etc.

GPU Computing : Think in Parallel

- Speedups of 8 x to 30x are quite common for certain class of applications
- The GPU is a data-parallel processor
  - Thousands of parallel threads
  - Thousands of data elements to process
  - All data processed by the same program
  - SPMD computation model
  - Contrast with task parallelism and ILP
- Best results when you "**Think Data Parallel**"
  - Design your algorithm for data-parallelism
  - Understand parallel algorithmic complexity and efficiency
  - Use data-parallel algorithmic primitives as building blocks



GPU Computing : Think in Parallel

- Optimized for structured parallel execution
  - Extensive ALU counts & Memory Bandwidth
  - Cooperative multi-threading hides latency
- Shared Instructions Resources
- Fixed function units for parallel workloads dispatch
- Extensive exploitations of Locality
- Performance /(Cost/Watt); Power for Core
- Structured Parallelism enables more flops less watts

GPU Computing : Think in Parallel

GPU Computing : Optimise Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it's better to recompute than to cache
  - GPU spends its translators on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host

GPU Computing : Think in Parallel

- GPU Computing : Use Parallelism Efficiently
- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks
- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory

What is GPGPU ?

- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

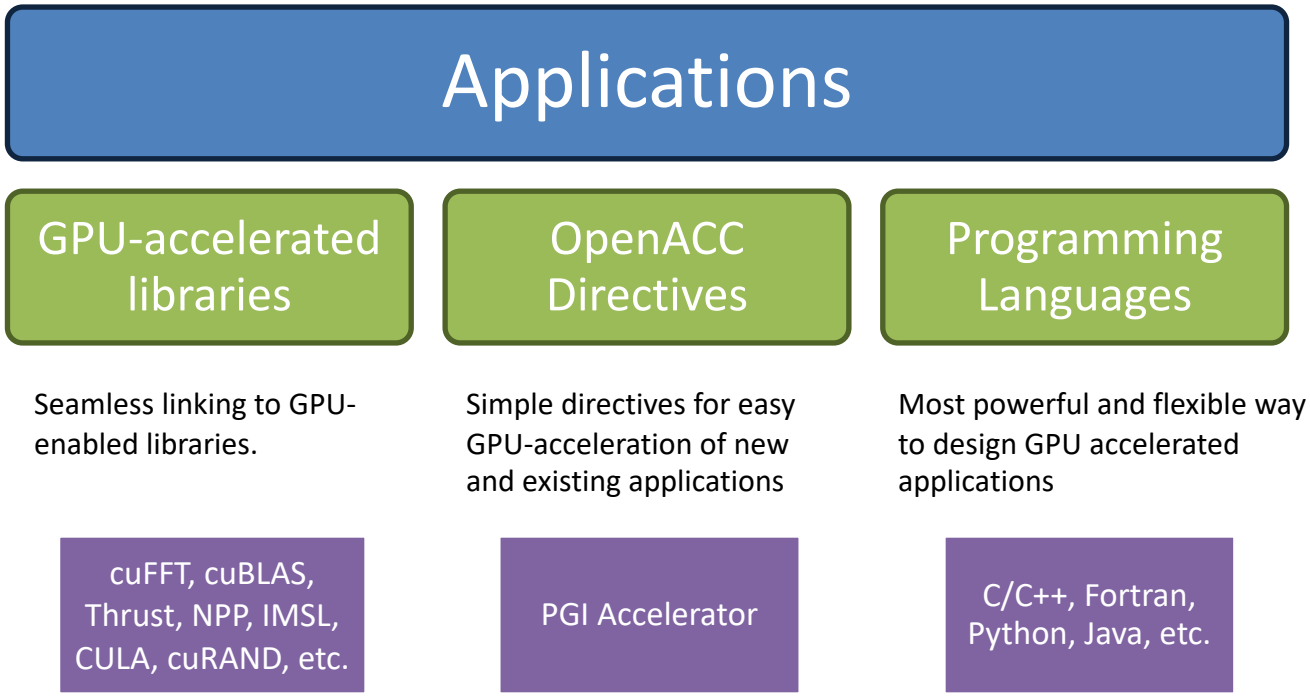
GPU Programming : Main Challenges

- Challenge 1 : Programmability
  - Example : Matrix Computations
    - To port an existing scientific application to a GPU
- GPU memory exists on the card itself
  - Must send matrix array over PCI-Express Bus
  - Send **A**, **B, C** to **GPU** over PCIe
  - Perform GPU-based computations on **A,B, C**
  - Read result **C** from **GPU** over PCIe
- The user must focus considerable effort on optimizing performance by manually orchestrating data movement and managing thread level parallelism on GPU.

GPU Programming : Main Challenges

- Challenge 2 : Accuracy
- Example : Non-Scientific Computation - Video Games (Frames) (A single bit difference in a rendered pixel in a real-time graphics program may be discarded when generating subsequence frames)
- Scientific Computing : Single bit error - Propagates overall error
- **Past History :** Most GPUs support single/double precision, 32 bit /64-bit floating point operation, - all GPUs have necessarily implemented the full IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754)

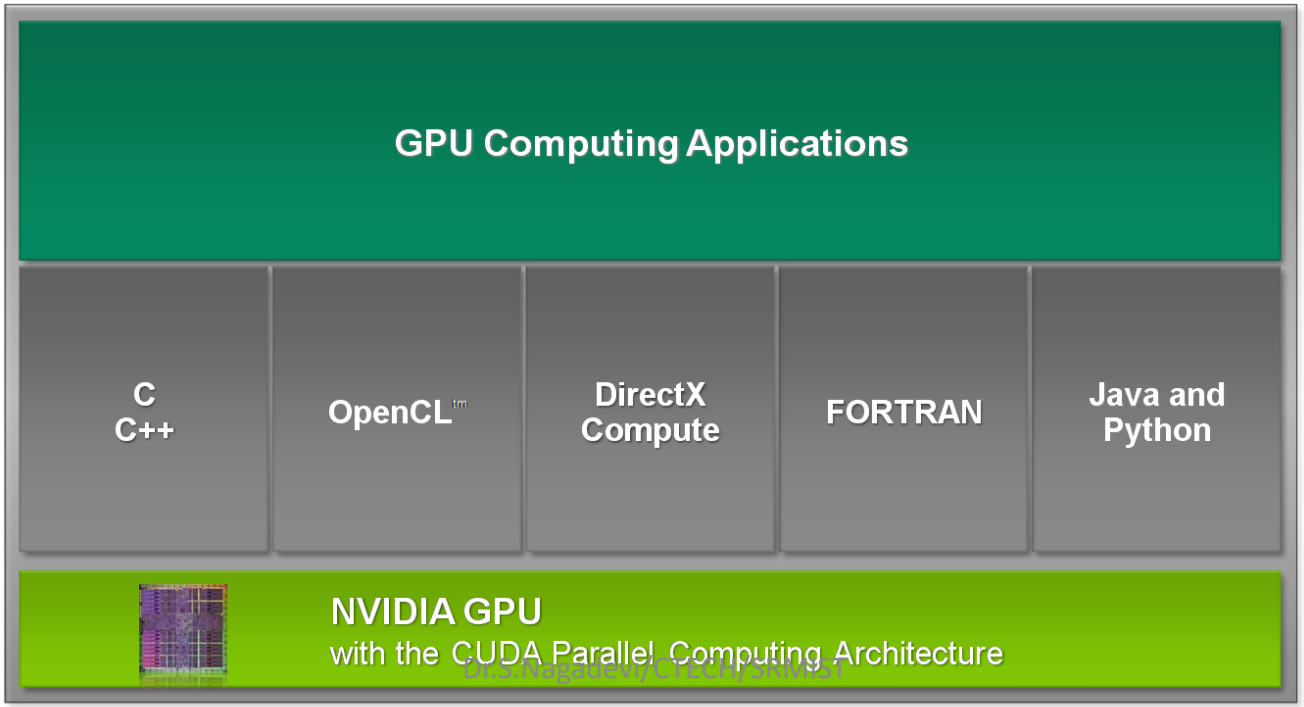
GPU Acceleration



Enter CUDA

CUDA is NVIDIA's general purpose parallel computing architecture .

- designed for calculation-intensive computation on GPU hardware
- CUDA is not a language, it is an API



GPU : Architecture

- Several multiprocessors (MP), each with: several simple cores - small shared memory
- The threads executing in the same MP must execute the same instruction
- Shared memory must be used to prevent the high latency of the global device memory

Glance at NVIDIA GPU's

- NVIDIA GPU Computing Architecture is a separate HW interface that can be plugged into the desktops / workstations / servers with little effort.
- G80 series GPUs /Tesla deliver FEW HUNDRED to TERAFLOPS on compiled parallel C ap

GeForce 8800 TeslaD870 Tesla S870

GPU Thread Organization

- Reflects the memory hierarchy of the device
- All threads from a single block are executed in the same MP
- Shared memory: Used for communication and synchronization of thread of the same block

NVIDA : CUDA – Data Parallelism

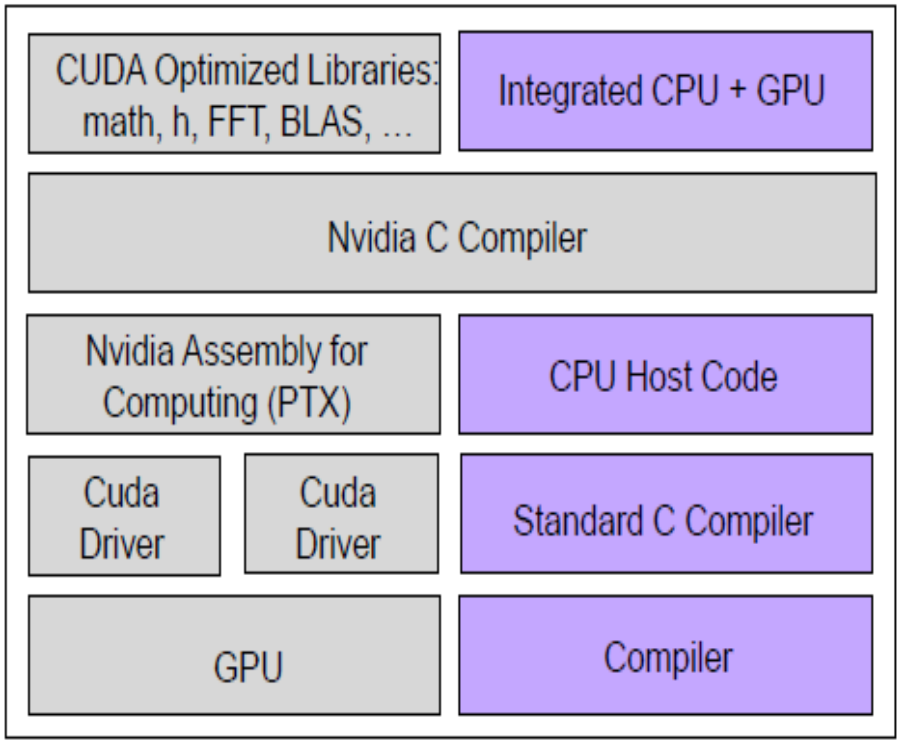
- To a CUDA Developer,
  - The computing system consists of a host, which is a traditional central processing unit (CPU) such as Intel, AMD, IBM, Cray multi-core architecture and one more devices, which are massively parallel processors equipped with a large number of arithmetic execution units.
- Computing depends upon the concept of Data Parallelism - Image Processing, Video Frames, Physics, Aero dynamics, Chemistry, Bio-Informatics
- Regular Computations and Irregular Computations.

NVIDA : CUDA – Data Parallelism

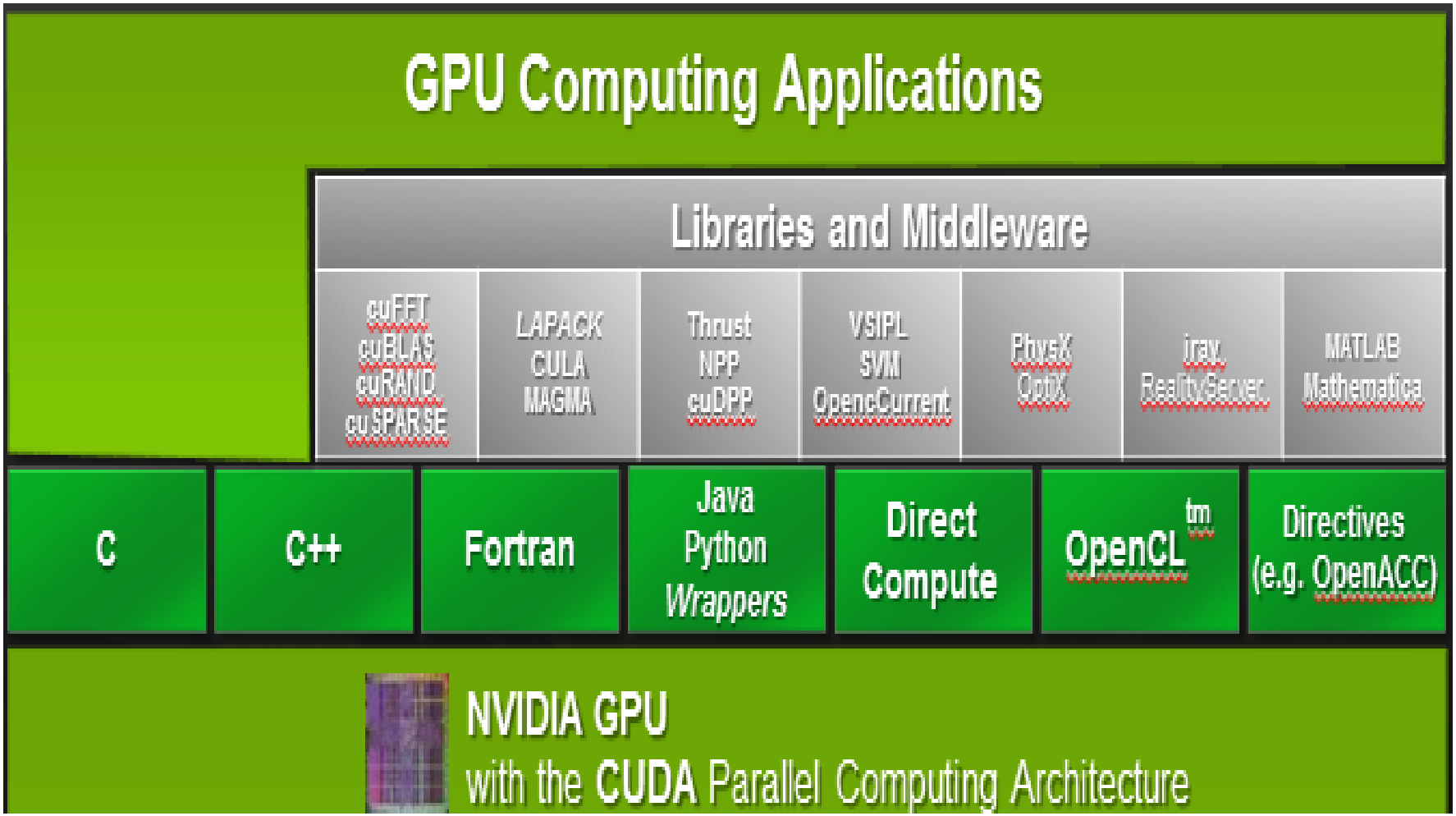
- Data Parallelism
- It refers to the program property whereby many arithmetic operations can be safely performed on the data structure in a simultaneous manner.
- The concept of Data Parallelism is applied to typical matrix-matrix computation.

CUDA Software Development

NVIDIA CUDA platform for parallel processing on Nvidia GPUs. Key elements are common C/C++ source code with different compiler forks for CPUs and GPUs; function libraries that simplify programming; and a hardware-abstraction mechanism that hides the details of the GPU architecture from programmers.



NVIDIA GPU Computing – Languages and Libraries



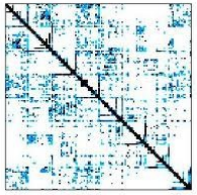
An approach to Writing CUDA Kernels

- Use algorithms that can expose substantial parallelism, you'll need thousands of threads…
- Identify ideal GPU memory system to use for kernel data for best performance
- Minimize host/GPU DMA transfers, use pinned memory buffers when appropriate
- Optimal kernels involve many trade-offs, easier to explore through experimentation with micro benchmarks based key components of the real science code, without the baggage
- Analyze the real-world use cases and select the kernel(s) that best match, by size, parameters, etc.

Processor Terminology

- SPA - Streaming Processor Array (variable across GeForce 8-series, 8 in GeForce8800)
- TPC - Texture Processor Cluster (2 SM + TEX)
- SM - Streaming Multiprocessor (8 SP); Multi-threaded processor core; Fundamental processing unit for CUDA thread block
- SP - Streaming Processor; Scalar ALU for a single CUDA thread

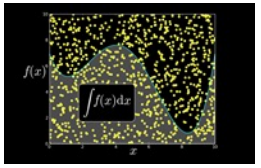
GPU Accelerated Libraries



cuSPARSE

NPP

cuFFT



cuRAND

Why GPUs for Machine Learning?

- Speed-ups over CPUs when it comes to deep neural network tasks like complex models, huge datasets, and a large number of images.
- Deep Learning model training is based on simple matrix operations
- Execute many parallel computations and increase the quality of images on the screen
- GPUs assemble many specialized cores that deal with huge data sets and deliver massive performance.
- A GPU devotes more transistors to arithmetic logic than a CPU
- Deep-learning GPUs provide high-performance computing power on a single chip while supporting modern machine-learning frameworks like TensorFlow and PyTorch
- Video graphics are made up of polygonal coordinates translated into bitmaps and then into signals shown on a screen. This translation necessitates massive processing power from the Graphics Processing Unit (GPU)

Deep Learning Tasks

- Complex computing tasks such as
  - training deep neural networks,
  - mathematical modeling using matrix calculations,
  - and working with 3D graphics.
- A GPU receives graphic information such as
  - image geometry,
  - color,
  - and textures

  from the CPU and executes them to draw images on the screen.

Algorithm Factors Affecting GPU Use for Machine Learning

- Data Parallelism
- Memory Use
- GPU Performance

what makes GPU ideal for machine learning?

- GPUs are designed to do multiple computations in parallel,
- which is excellent for deep learning algorithms' highly parallel nature.
- They also contain a large amount of memory, which is useful for deep-learning models that require a lot of data.

GPU Market Players - Nvidia and AMD

- There are two major players in the GPU market: AMD and Nvidia.
- There are a large number of GPUs used for deep learning. However, Nvidia makes the majority of the best ones.
- Nvidia dominates the market of GPUs, especially
  - for deep learning and complex neural networks, because of their substantial support in the forum, software, drivers, CUDA, and cuDNN.

NVIDIA Titan RTX

- NVIDIA Titan RTX is a high-end gaming GPU that is also great for deep learning tasks. Built for data scientists and AI researchers, this GPU is powered by NVIDIA Turing™ architecture to offer unbeatable performance. The TITAN RTX is the best PC GPU for training neural networks, processing massive datasets, and creating ultra-high-resolution videos and 3D graphics. Additionally, it is supported by NVIDIA drivers and SDKs, enabling developers, researchers, and creators to work more effectively to deliver better results.
- **Technical Features**
- CUDA cores: 4608
- Tensor cores: 576
- GPU memory: 24 GB GDDR6
- Memory Bandwidth: 673GB/s
- Compute APIs: CUDA, DirectCompute, OpenCL™

- **NVIDIA Tesla V100**
- NVIDIA Tesla is the first tensor core GPU built to accelerate artificial intelligence, high-performance computing (HPC), Deep learning, and machine learning tasks. Powered by NVIDIA Volta architecture, Tesla V100 delivers 125TFLOPS of deep learning performance for training and inference. In addition, it consumes less power than other GPUs. NVIDIA Tesla is one of the market's best GPUs for deep learning due to its outstanding performance in AI and machine learning applications. With this GPU, data scientists and engineers may now focus on building the next AI breakthrough rather than optimizing memory usage.
- **Technical Features**
- CUDA cores: 5120
- Tensor cores: 640
- Memory Bandwidth: 900 GB/s
- GPU memory: 16GB
- Clock Speed: 1246 MHz
- Compute APIs: CUDA, DirectCompute, OpenCL™, OpenACC®

- **NVIDIA Quadro RTX 8000**
- NVIDIA Quadro RTX 8000 is the world's most powerful graphics card built by PNY for deep learning matrix multiplications. A single Quadro RTX 8000 card can render complex professional models with realistically accurate shadows, reflections, and refractions, providing users with rapid insight. Powered by the NVIDIA TuringTM architecture and NVIDIA RTXTM platform, Quadro provides professionals with the latest hardware-accelerated real-time ray tracing, deep learning, and advanced shading. When used with NVLink, its memory may be expanded to 96 GB.
- **Technical Features**
- CUDA cores: 4608
- Tensor cores: 576
- GPU memory: 48 GB GDDR6
- Memory Bandwidth: 672 GB/s
- Compute APIs: CUDA, DirectCompute, OpenCL™

- **NVIDIA Tesla P100**
- Based on NVIDIA Pascal architecture, the Nvidia Tesla p100 is a GPU built for machine learning and HPC. Tesla P100 with NVIDIA NVLink technology provides lightning-fast nodes to reduce time to solution for large-scale applications significantly. With NVLink, a server node may link up to eight Tesla P100s at 5X the bandwidth of PCIe.
- **Technical Features**
- CUDA cores:
- Tensor cores:
- GPU memory:
- Memory Bandwidth:
- Compute APIs:

- **NVIDIA RTX A6000**
- One of the latest GPUs is the NVIDIA RTX A6000, which is excellent for deep learning. Based on the Turing architecture, it can execute both deep learning algorithms and conventional graphics processing tasks. The RTX A6000 also has Deep Learning Super Sampling as a feature (DLSS). This feature can render images at higher resolutions while maintaining quality and speed. A geometry processor, texture mapper core, rasterizer core, and video engine core are some of the other features of this GPU.
- **Technical Features**
- CUDA cores: 10,752
- Tensor cores: 336
- GPU memory: 48GB

Design Factors

- The design of a CPU is optimized for sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution.
- More importantly, large cache memories are provided to reduce the instruction and data access latencies of large complex applications.
- Neither control logic nor cache memories contribute to the peak calculation speed. As of 2012, the high-end general-purpose multicore microprocessors typically have six to eight large processor cores and multiple megabytes of on-chip c

- Memory bandwidth is another important issue. The speed of many applications is limited by the rate at which data can be delivered from the memory system into the processors. Graphics chips have been operating at approximately six times the memory bandwidth of contemporaneously available CPU chips.

- In late 2006, GeForce 8800 GTX, or simply G80, was capable of moving data at about 85 gigabytes per second (GB/s) in and out of its main dynamic random-access memory (DRAM) because of graphics frame buffer requirements and the relaxed memory model (the way various system software, applications, and input/output (I/O) devices expect how their memory accesses work).
- The more recent GTX680 chip supports about 200 GB/s. In contrast, general-purpose processors have to satisfy requirements from legacy operating systems, applications, and I/O devices that make memory bandwidth more difficult to increase. As a result, CPUs will continue to be at a disadvantage in terms of memory bandwidth for some time

- The design philosophy of GPUs is shaped by the fast-growing video game industry that exerts tremendous economic pressure for the ability to perform a **massive number of floating-point calculations** per video frame in advanced games.
- This demand motivates GPU vendors to look for ways to maximize the chip area and power budget dedicated to floating-point calculations.
- The prevailing solution is to optimize for the execution throughput of massive numbers of threads. The design saves chip area and power by allowing pipelined memory channels and arithmetic operations to have long latency.
- The reduced area and power of the memory access hardware and arithmetic units allows the designers to have more of them on a chip and thus increase the total execution throughput.

- **Small cache memories** are provided to help control the bandwidth requirements of these applications so that multiple threads that access the same memory data do not need to all go to the DRAM.
- This design style is commonly referred to **as throughput-oriented design** since it strives to maximize the total execution throughput of a large number of threads while allowing individual threads to take a potentially much longer time to execute.

- The CPUs, on the other hand, are designed to minimize the execution latency of a single thread.
- Large last-level on-chip caches are designed to capture frequently accessed data and convert some of the long-latency memory accesses into short-latency cache accesses.
- The arithmetic units and operand data delivery logic are also designed to minimize the effective latency of operation at the cost of increased use of chip area and power.
- By reducing the latency of operations within the same thread, the CPU hardware reduces the execution latency of each individual thread. However, the large cache memory, low-latency arithmetic units, and sophisticated operand delivery logic consume chip area and power that could be otherwise used to provide more arithmetic execution units and memory access channels.
- This design style is commonly referred to as **latency-oriented design**.

- GPUs are designed as parallel, throughput oriented computing engines and they will not perform well on some tasks on which CPUs are designed to perform well.
- For programs that have one or very few threads, CPUs with lower operation latencies can achieve much higher performance than GPUs.
- When a program has a large number of threads, GPUs with higher execution throughput can achieve much higher performance than CPUs.
- Therefore, one should expect that many applications use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.

- This is why the CUDA programming model, introduced by NVIDIA in 2007, is designed to support joint CPU-GPU execution of an application.
- The demand for supporting joint CPUGPU execution is further reflected in more recent programming models such as OpenCL , OpenACC , and C++ AMP.

Decision Factors

- **Several other factors can** be even more important.
- First and foremost, the processors of choice must have a very large presence in the marketplace, referred to as the installed base of the processor.
- The reason is very simple. The cost of software development is best justified by a very large customer population.
- Applications that run on a processor with a small market presence will not have a large customer base.
- This has been a major problem with traditional parallel computing systems that have negligible market presence compared to general-purpose microprocessors.
- Only a few elite applications funded by government and large corporations have been successfully developed on these traditional parallel computing systems.

- This has changed with many-core GPUs. Due to their popularity in the PC market, GPUs have been sold by the hundreds of millions. Virtually all PCs have GPUs in them.
- There are more than 400 million CUDA-enabled GPUs in use to date. This is the first time that massively parallel computing is feasible with a mass-market product.
- Such a large market presence has made these GPUs economically attractive targets for application developers.

- **Another important decision factor is practical form factors and easy accessibility.**
- Until 2006, parallel software applications usually ran on data center servers or departmental clusters. But such execution environments tend to limit the use of these applications. For example, in an application such as medical imaging, it is fine to publish a paper based on a 64-node cluster machine.
- But actual clinical applications on magnetic resonance imaging (MRI) machines have been based on some combination of a PC and special hardware accelerators.

- . The simple reason is that manufacturers such as GE and Siemens cannot sell MRIs with racks of compute server boxes into clinical settings, while this is common in academic departmental settings.
- In fact, National Institutes of Health (NIH) refused to fund parallel programming projects for some time: they felt that the impact of parallel software would be limited because huge cluster-based machines would not work in the clinical setting.
- Today, GE ships MRI products with GPUs and NIH funds research using GPU computing

- Yet another important consideration in selecting a processor for executing numeric computing applications is the level of support for the Institute of Electrical and Electronic Engineers' (IEEE) floating-point standard. The standard makes it possible to have predictable results across processors from different vendors. While the support for the IEEE floating-point standard was not strong in early GPUs, this has also changed for new generations of GPUs since the introduction of the G80. As we will discuss

- in Chapter 7, GPU support for the IEEE floating-point standard has become comparable with that of the CPUs. As a result, one can expect that more numerical applications will be ported to GPUs and yield comparable result values as the CPUs. Up to 2009, a major remaining issue was that the GPUs' floating-point arithmetic units were primarily single precision. Applications that truly require double-precision floating-point arithmetic units were not suitable for GPU execution. However, this has changed with the recent GPUs of which the double-precision execution speed approaches about half of that of single precision, a level that high-end CPU cores achieve. This makes the GPUs suitable for even more numerical applications.

- Until 2006, graphics chips were very difficult to use because programmers had to use the equivalent of graphics API (application programming interface) functions to access the processor cores, meaning that OpenGL or Direct3D techniques were needed to program these chips. Stated more simply, a computation must be expressed as a function that paints a pixel in some way to execute on these early GPUs. This technique was called GPGPU (general-purpose programming using a graphics processing unit). Even with a higher-level programming environment, the underlying code still needs to fit into the APIs that are designed to paint pixels. These APIs limit the kinds of applications that one can actually write for early GPGPUs. Consequently, it did not become a widespread programming phenomenon. Nonetheless, this technology was sufficiently exciting to inspire some heroic efforts and excellent research results

- But everything changed in 2007 with the release of CUDA [NVIDIA2007]. NVIDIA stared to devote silicon areas on their GPU chips to facilitate the ease of parallel programming.This did not represent software changes alone; additional hardware was added to the chips. In the G80 and its successor chips for parallel computing, CUDA programs no longer go through the graphics interface at all. Instead, a new generalpurpose parallel programming interface on the silicon chip serves the requests of CUDA programs. The general-purpose programming interface greatly expands the types of applications that one can easily develop for GPUs. Moreover, all the other software layers were redone as well, so that the programmers can use the familiar C/C11 programming tools. Some of our students tried to do their lab assignments using the old OpenGL-based programming interface, and their experience helped them to greatly appreciate the improvements that eliminated the need for using the graphics APIs for computing applications.

ARCHITECTURE OF A MODERN GPU

- Figure shows the architecture of a typical CUDA-capable GPU
- It is organized into an array of highly threaded streaming multiprocessors (SMs).
- In Figure two SMs form a building block.
- However, the number of SMs in a building block can vary from one generation of CUDA GPUs to another generation.
- Also, in Figure , each SM has a number of streaming processors (SPs) that share control logic and an instruction cache.
- Each GPU currently comes with multiple gigabytes of Graphic Double Data Rate (GDDR) DRAM, referred to as global memory in Figure

CUDA-capable GPU

- . These GDDR DRAMs differ from the system DRAMs on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics.
- For graphics applications, they hold video images and texture information for 3D rendering.
- But for computing, they function as very high bandwidth off-chip memory, though with somewhat longer latency than typical system memory.
- For massively parallel applications, the higher bandwidth makes up for the longer latency

CUDA-capable GPU

- The G80 introduced the CUDA architecture and had 86.4 GB/s of memory bandwidth, plus a communication link to the CPU core logic over a PCI-Express Generation 2 (Gen2) interface.
- Over PCI-E Gen2, a CUDA application can transfer data from the system memory to the global memory at 4 GB/s, and at the same time upload data back to the system memory at 4 GB/s.

CUDA-capable GPU

- Altogether, there is a combined total of 8 GB/s. More recent GPUs use PCI-E Gen3, which supports 8 GB/s in each direction.
- As the size of GPU memory grows, applications increasingly keep their data in the global memory and only occasionally use the PCI-E to communicate with the CPU system memory if there is need for using a library that is only available on the CPUs.
- The communication bandwidth is also expected to grow as the CPU bus bandwidth of the system memory grows in the future.

CUDA-capable GPU

- With 16,384 threads, the GTX680 exceeds 1.5 teraflops in double precision.
- A good application typically runs 5,000-12,000 threads simultaneously on this chip. For those who are used to multithreading in CPUs, note that Intel CPUs support two or four threads, depending on the machine model, per core

CUDA-capable GPU

- . CPUs, however, are increasingly used with SIMD (single instruction, multiple data) instructions for high numerical performance.
- The level of parallelism supported by both GPU hardware and CPU hardware is increasing quickly.
- It is therefore very important to strive for high levels of parallelism when developing computing applications.