

**SUPSI**

# Introduction to GPUs and CUDA

Tiziano Leidi

Fondamenti di Multimedia Processing  
Bachelor in Ingegneria Informatica

## Multimedia Processing

Today, multimedia contents are practically all in digital form. The acquisition, transfer, modification and archiving of multimedia information represents a big challenges in many application domains.

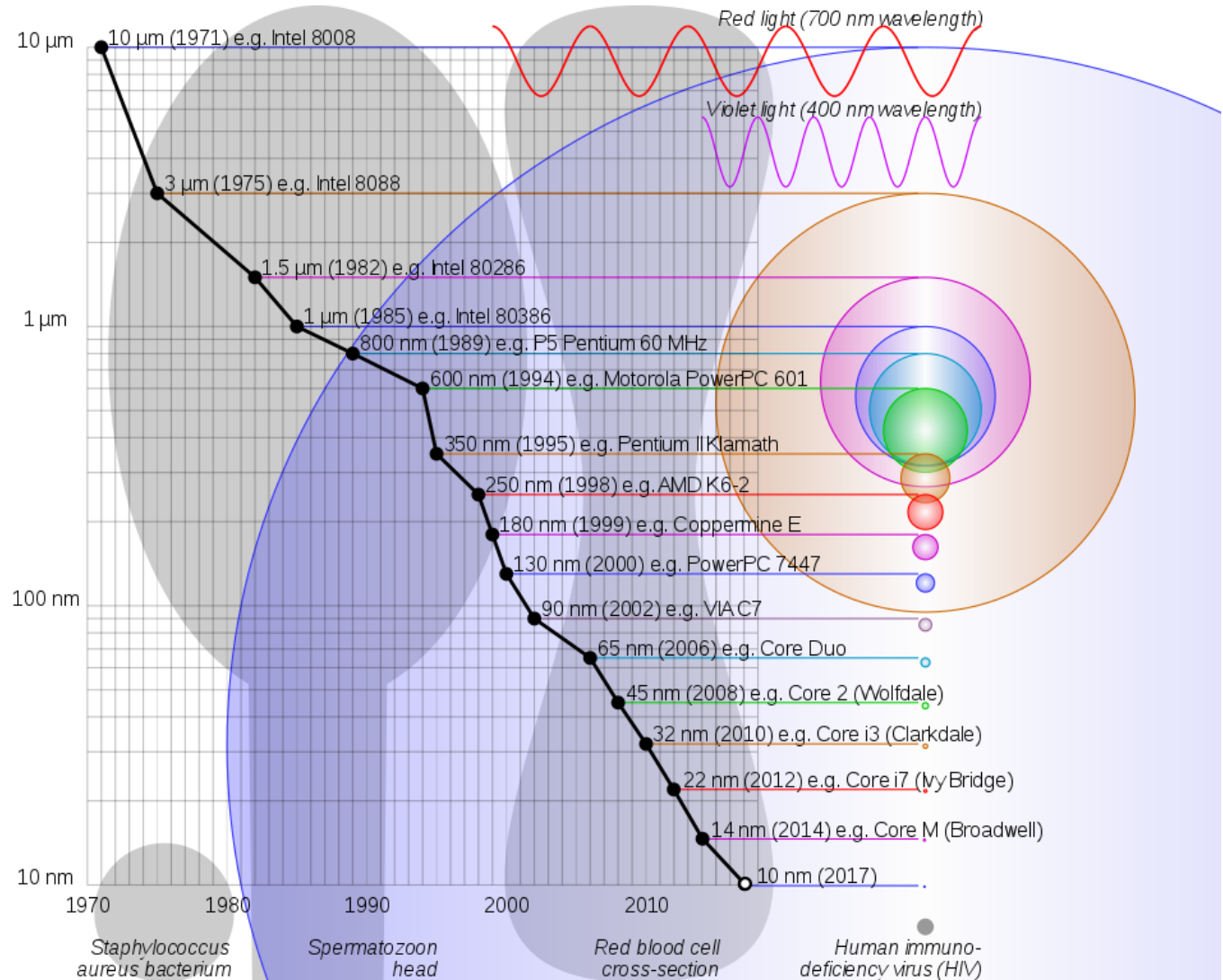
Innovative technologies are continuously required to allow **processing the growing amount of multimedia data at real-time**. Multimedia processing is constantly **hungry of computing power**.

## Microprocessors

- Miniaturization of components is now reaching the molecular level.
- The smallest "linewidth" (width of feature on the chip surface) shrank from two microns in 1980 to less than a tenth micron (100 nanometers) a quarter-century later.
- If viewed in cross section, the thickness of horizontal layers of material deposited on the silicon surface is currently less than 1 nanometers.
- *The width of a human hair is about 100 microns and the width of a molecule is about 1 nanometer.*

## Semiconductor manufacturing processes

10  $\mu\text{m}$  – 1971  
 6  $\mu\text{m}$  – 1974  
 3  $\mu\text{m}$  – 1977  
 1.5  $\mu\text{m}$  – 1982  
 1  $\mu\text{m}$  – 1985  
 800 nm – 1989  
 600 nm – 1994  
 350 nm – 1995  
 250 nm – 1997  
 180 nm – 1999  
 130 nm – 2001  
 90 nm – 2004  
 65 nm – 2006  
 45 nm – 2008  
 32 nm – 2010  
 22 nm – 2012  
 14 nm – 2014  
 10 nm – 2016  
 7 nm – 2018  
 5 nm – 2020

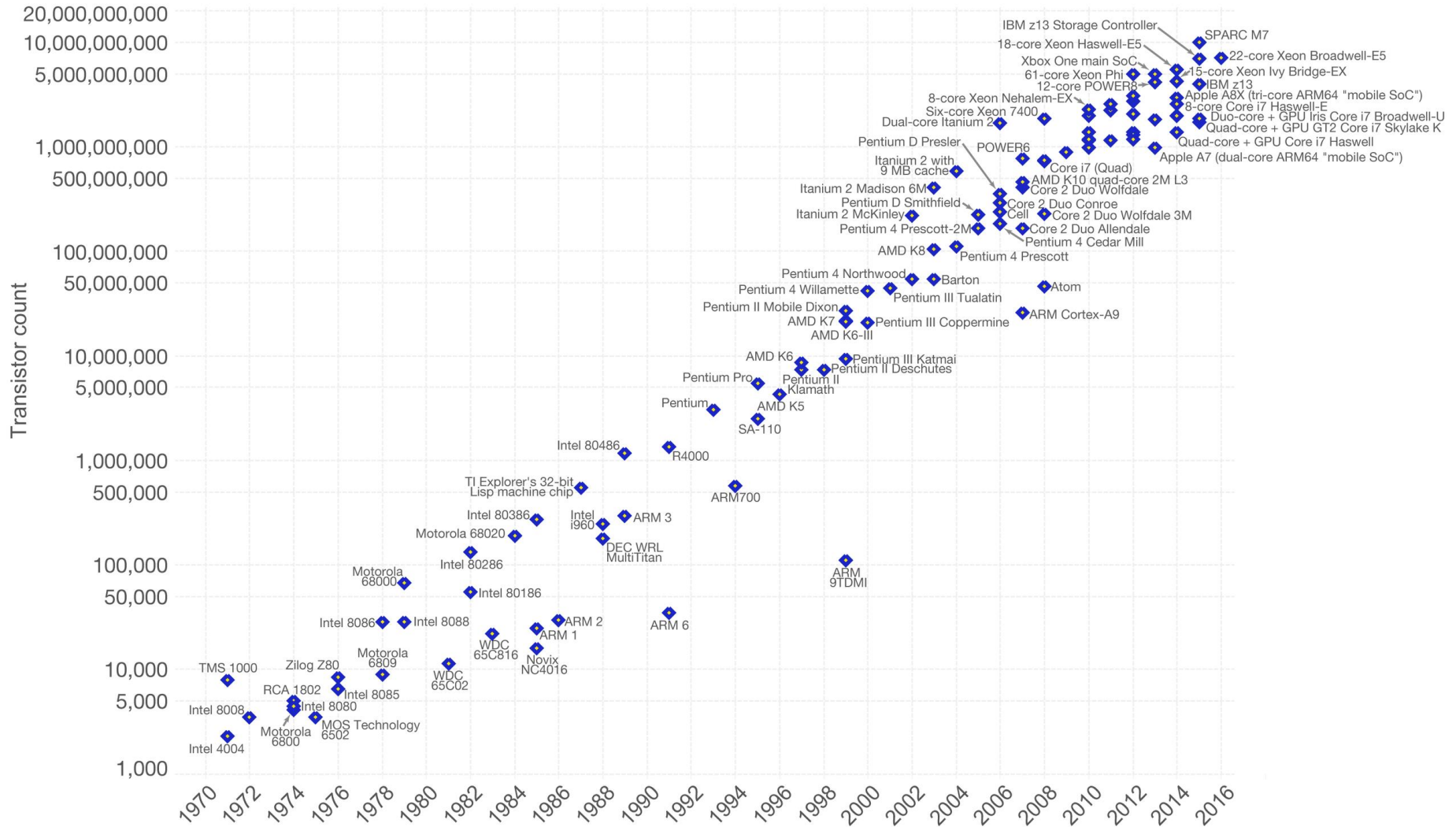


## More's Law

- “The number of transistors in integrated circuits doubles approximately every 18 months”.
- Law described by Gordon E. Moore (Intel), based on the period from 1958 to 1965. Trend “for at least 10 years”.
- Today, used by semiconductor industry for long-term planning.
- Influences features of digital devices: speed of computers, memory sizes, number of pixels, ...



# More's Law



## Technical Obstacles

- **The memory wall:** increasing gap between processor and memory speeds. Pushes cache sizes larger in order to mask the latency of memory. Then, the memory bandwidth becomes the bottleneck.
- **The ILP wall:** the increasing difficulty of finding enough parallelism in a stream of instructions to keep a single-core processor busy.
- **The power wall:** the trend of consuming exponentially increasing power with each factorial increase of operating frequency.

## Chip-Level Multiprocessors

- While manufacturing technology improves (smaller size of individual gates), **physical limits of semiconductors** have become a major design concern. These physical limits can cause significant heat dissipation, energy consumption and data synchronization problems.
- **Clock rates** increased by orders of magnitude in the decades of the late 20th century, from several megahertz in the 1980s to several gigahertz in the early 2000s.
- Then, as the rate of clock speed improvements slowed, increased use of parallel computing in the form of **multi-core and many-core processors** has been pursued to improve overall processing performance.



## Hardware Acceleration

Hardware acceleration is the use of computer hardware to **perform specific functions faster** than is possible with software on a general-purpose CPU.

Depending upon **granularity**, hardware acceleration can vary from a small functional unit to a large functional block.

Support provided **SIMD execution** (SSE, AVX) and **floating-point units (FPUs)** may also be considered a form of HW acceleration.

## Hardware Acceleration

- The hardware that performs the acceleration, when in a separate unit from the CPU, is referred to as a **hardware accelerator**.
- Hardware accelerators are designed for **computationally intensive** tasks.
- Examples of hardware accelerators are **GPUs**.

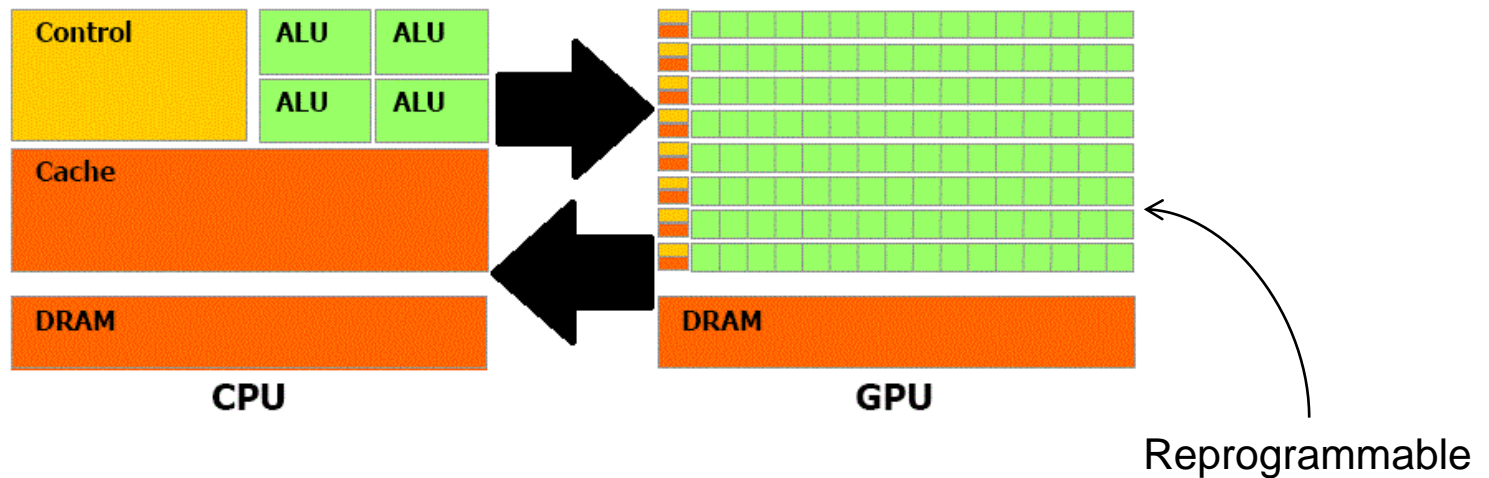


## Heterogeneous Computing

**Heterogeneous computing** refers to systems that use more than one kind of processor at the same time.

These systems usually incorporates **specialized processing capabilities** to handle particular tasks (for example CPU + GPU).

# Heterogeneous Computing



## Types of Hardware Accelerators

- **GPUs:** initially designed for manipulating images for output to a display. Today, their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel.
- **MICs:** multiprocessors developed by Intel (Xeon Phi), leverage x86 legacy by creating a x86-compatible multiprocessor architecture that can utilize existing parallelization software tools (OpenMP, Cilk).
- **DSPs:** specialized processors with an architecture optimized for digital signal processing. Large number of mathematical operations performed quickly and repeatedly on a series of data samples.
- **FPGAs:** integrated circuits designed to be configured by a customer or a designer after manufacturing.

## GPUs

GPUs have been developed as a consequence of the strong request of faster graphic cards coming from the gaming industry.

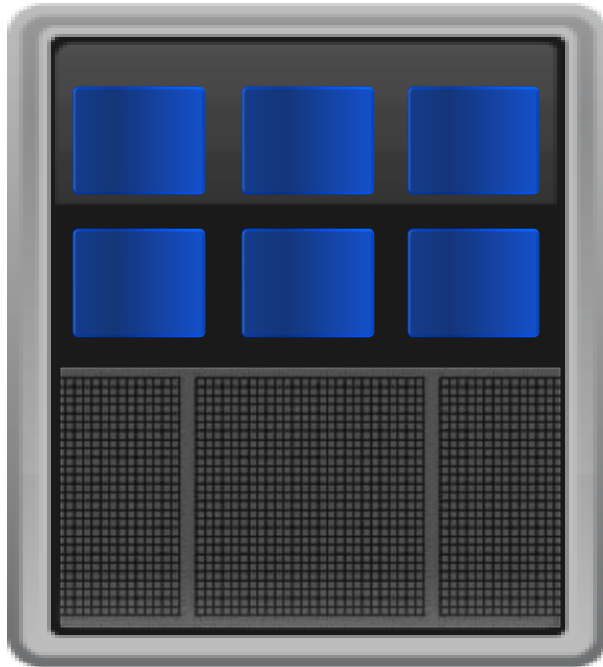
Then, people realized that the computing power provided by GPUs could also be exploited for different tasks than computer graphics: for example for scientific simulation, data mining and obviously for multimedia processing.

## GPUs

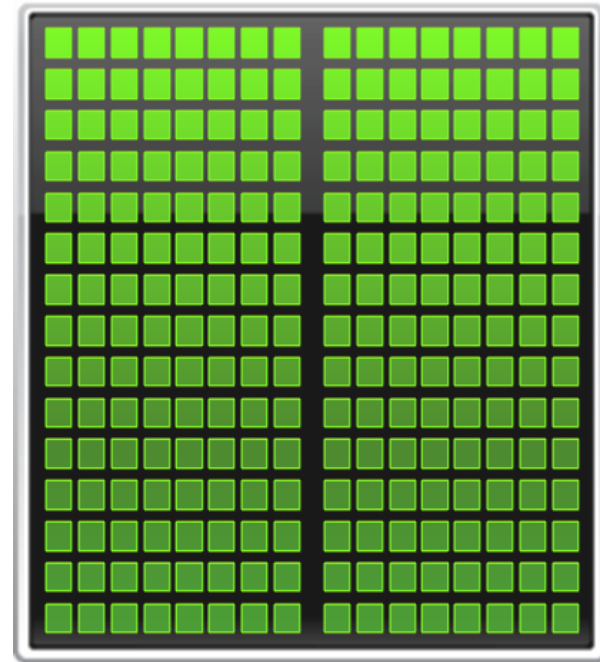
- GPUs are specialized processors with **dedicated memory** and with **many-core design optimized for SIMD floating point operations** (often required for rendering graphics).
- GPUs devote proportionally more transistors to arithmetic logic units and **less to caches and flow control** in comparison to CPUs (no branch prediction, no speculative execution, ...). GPUs also typically have **larger memory bandwidth** compared to CPUs.
- GPUs have a parallel throughput architecture that emphasizes executing **many concurrent threads slowly**, rather than executing a single thread very quickly.

# Parallel Processing with GPUs

## CPU

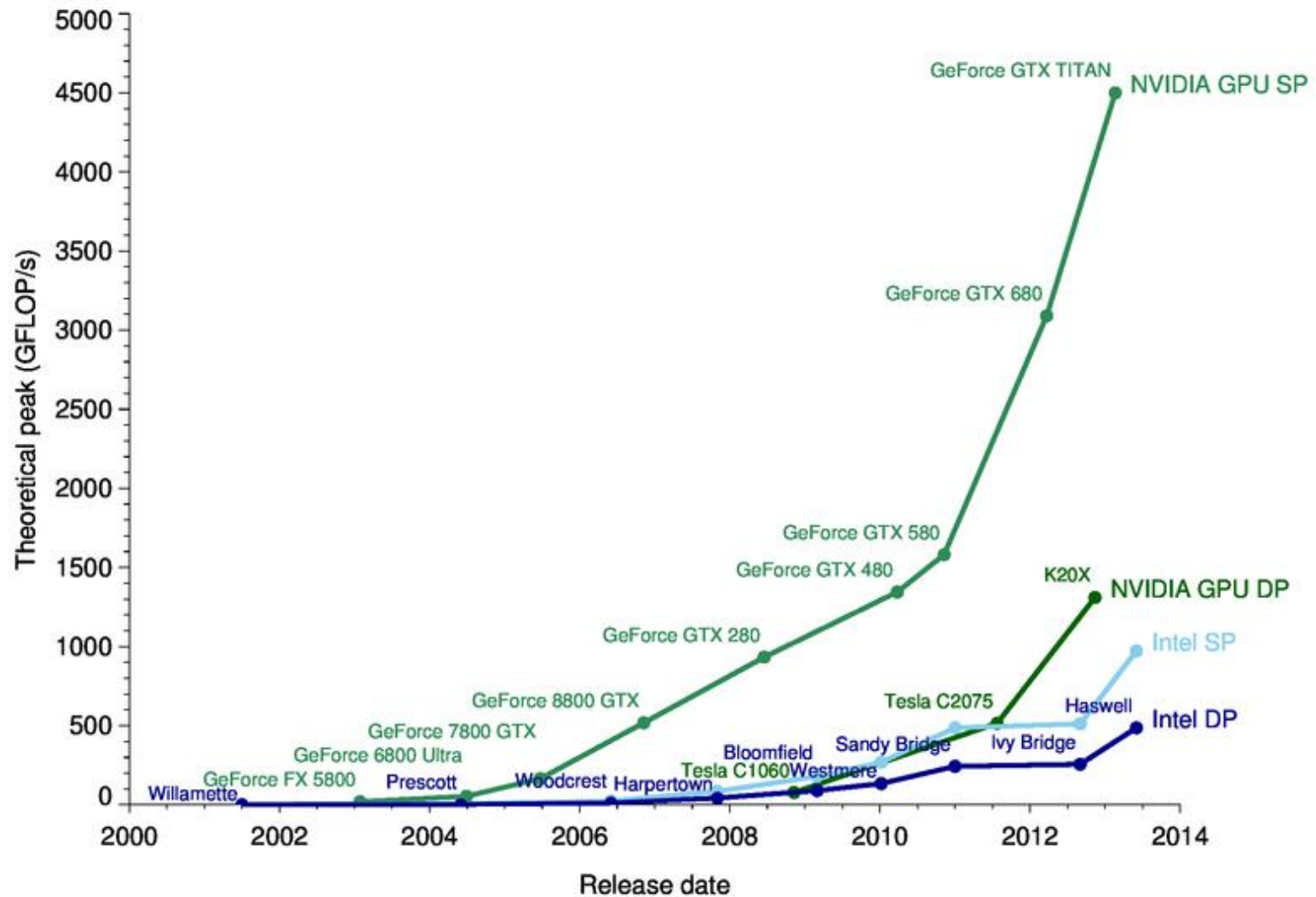


## GPU





# CPU vs. GPU – Floating Point Operations



## Examples

### *Nvidia GeForce RTX 2080 Ti – 999\$*

- 12 nm, 18.6 billion of transistors
- 4352 CUDA Cores (Turing Architecture)
- Memory: 11 GB GDDR6, 616 GB/sec
- Proc. power (GFLOPS): 13447 single, 420 double, 26895 half

### *AMD Radeon RX Vega 64 Liquid – 699\$*

- 14 nm
- 4096 Stream Processors
- Memory: 8 GB HBM2, 484 GB/sec
- Proc. power (GFLOPS): 13738 single, 859 double, 27476 half

## Examples

### *Nvidia Quadro GV100 – 9999\$*

- 12 nm
- 5120 CUDA Cores (Volta Architecture)
- Memory: 32 GB HBM2, 870 GB/sec
- Proc. power (GFLOPS): 14800 single, 7400 double

### *AMD Radeon Pro SSG – 4600\$*

- 14 nm
- 4096 Stream Processors (Vega Core)
- Memory: 16 GB HBM2, 484 GB/sec
- Proc. power (GFLOPS): 12290 single, 768 double, 24600 half

## Examples

### *Nvidia Tesla V100 – 10664\$*

- 12 nm
- 5120 CUDA Cores (Volta Architecture)
- Memory: 16 GB HBM2, 900 GB/sec
- Proc. power (GFLOPS): 14028 single, 7014 double

### *Nvidia Tegra X1*

- ARM Cortex-A57 + ARM Cortex-A53 (4 + 4 cores)
- 256 CUDA Cores (Maxwell Architecture)
- Memory: 8 GB LPDDR3/4, 26.6 GB/sec
- Proc. power (GFLOPS): 512 single, 1024 half

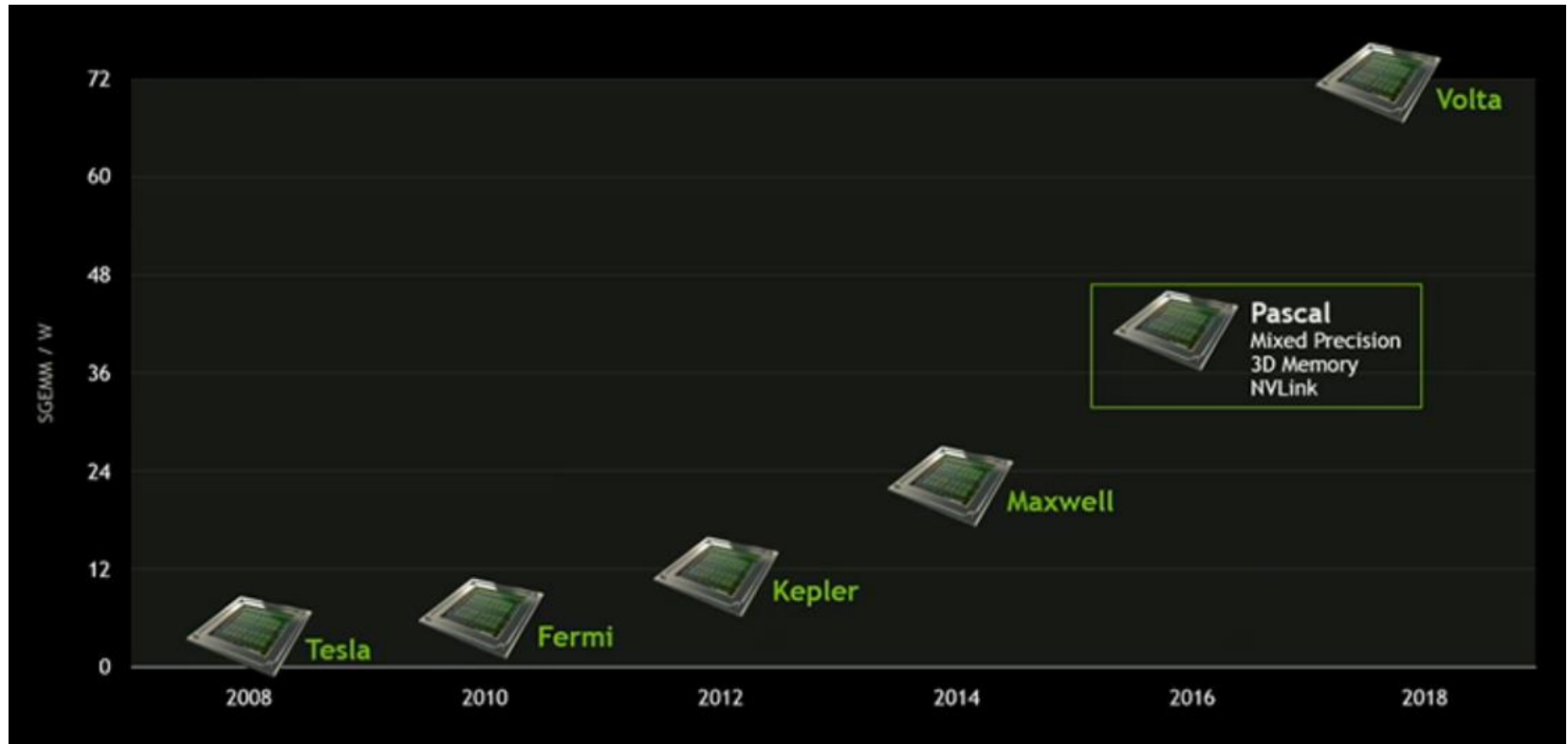
# Benchmarks

Search: 1060 etc ...	User rating %	Value %	Avg. bench %	Mkt. share %	Age months
 Compare <input type="checkbox"/> Nvidia RTX 2080 Ti Samples 3.9k	66 90 <sup>th</sup>	80 58 <sup>th</sup>	194 52 ↔ 235 100 <sup>th</sup>	0.76 96 <sup>th</sup>	1 99 <sup>th</sup>
 Compare <input type="checkbox"/> Nvidia Titan V Samples 216	70 93 <sup>rd</sup>		190 133 ↔ 215 100 <sup>th</sup>	0 24 <sup>th</sup>	11+ 97 <sup>th</sup>
 Compare <input type="checkbox"/> Nvidia Titan Xp Samples 2k	70 93 <sup>rd</sup>		156 53 ↔ 185 100 <sup>th</sup>	0.05 66 <sup>th</sup>	19+ 93 <sup>rd</sup>
 Compare <input type="checkbox"/> Nvidia GTX 1080-Ti Samples 217k	148 99 <sup>th</sup>	79.4 57 <sup>th</sup>	155 68 ↔ 177 99 <sup>th</sup>	3.71 99 <sup>th</sup>	20 93 <sup>rd</sup>
 Compare <input type="checkbox"/> Nvidia Quadro P6000 Samples 163	54 72 <sup>nd</sup>		151 127 ↔ 164 99 <sup>th</sup>	0 24 <sup>th</sup>	24+ 91 <sup>st</sup>
 Compare <input type="checkbox"/> Nvidia RTX 2080 Samples 6.8k	57 82 <sup>nd</sup>	93.1 84 <sup>th</sup>	150 61 ↔ 169 99 <sup>th</sup>	1.06 97 <sup>th</sup>	1 99 <sup>th</sup>
 Compare <input type="checkbox"/> Nvidia Titan X Pascal Samples 3.8k	60 86 <sup>th</sup>		146 44 ↔ 181 99 <sup>th</sup>	0.03 42 <sup>nd</sup>	27+ 90 <sup>th</sup>
 Compare <input type="checkbox"/> Nvidia RTX 2070 Samples 1.3k	58 83 <sup>rd</sup>	102 98 <sup>th</sup>	134 111 ↔ 145 99 <sup>th</sup>	0.57 94 <sup>th</sup>	1 100 <sup>th</sup>
 Compare <input type="checkbox"/> Nvidia Quadro GP100 Samples 28	52 56 <sup>th</sup>		132 92 ↔ 152 99 <sup>th</sup>	0 24 <sup>th</sup>	18+ 94 <sup>th</sup>
 Compare <input type="checkbox"/> AMD RX Vega 64-LC (Liquid Cooled) Samples 767	60 86 <sup>th</sup>		124 71 ↔ 139 98 <sup>th</sup>	0.01 42 <sup>nd</sup>	15+ 96 <sup>th</sup>

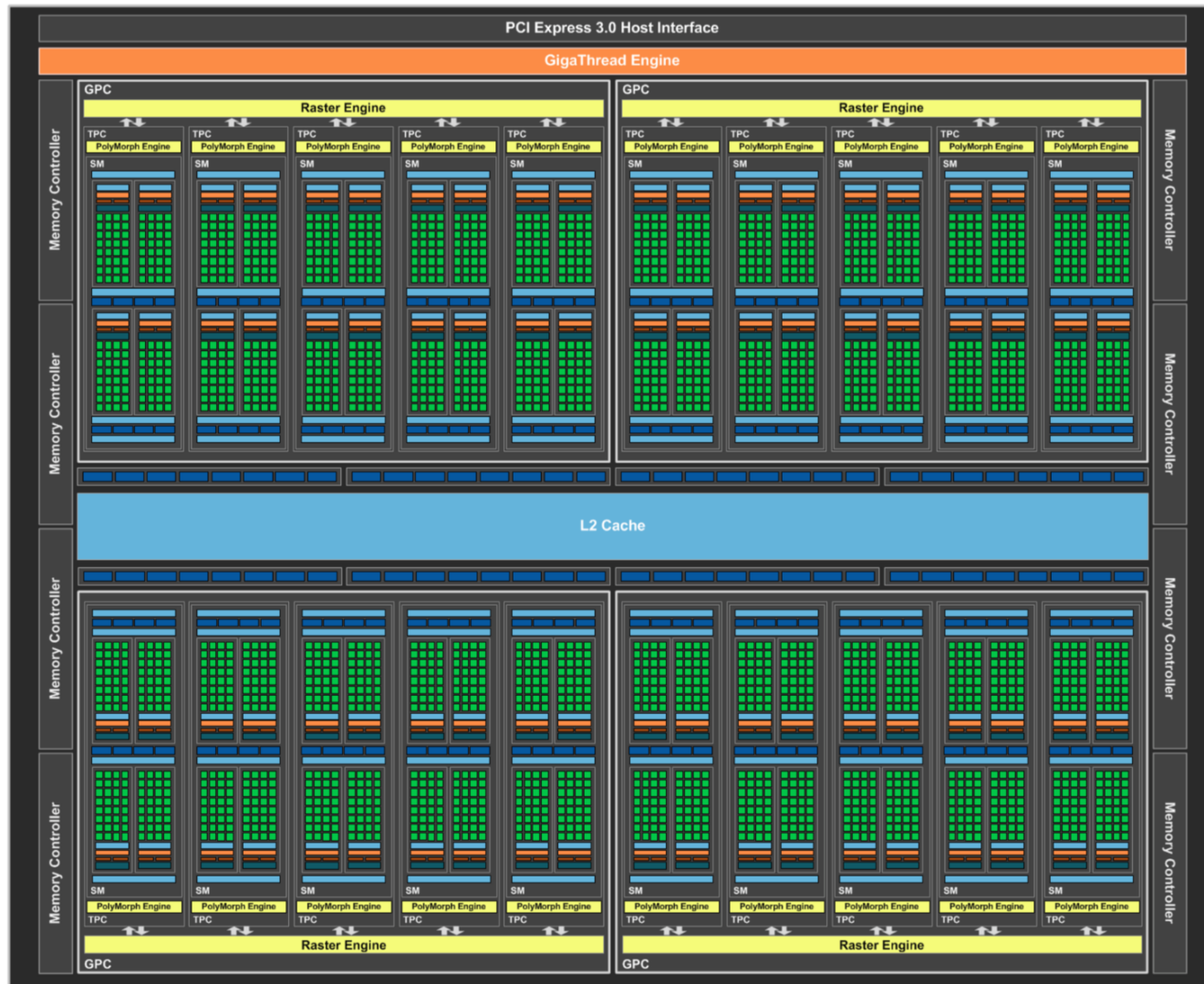
<http://gpu.userbenchmark.com/Comparison?>

<https://www.futuremark.com/hardware/gpu>

# Nvidia Architecture Evolution



# Pascal Architecture



# Turnig Architecture





## Tensor Cores

NVIDIA Tesla GPUs are powered by **tensor cores**, a revolutionary technology that delivers groundbreaking deep learning performance.

Tensor cores can accelerate **large matrix operations**, used when training neural networks, and perform mixed-precision matrix multiply and accumulate calculations in a single operation.

For example, a tensor core multiplies two  $4 \times 4$  FP16 matrices, and then adds a third FP16 or FP32 matrix to the result by using fused multiply–add operations, and obtains an FP32 result that could be optionally demoted to an FP16 result.

## Ray Tracing

The Turing architecture introduced the first consumer products capable of **real-time ray-tracing**, which has been a longstanding goal of the computer graphics industry.

The ray-tracing performed by the cores can be used to produce **reflections, refractions and shadows**, replacing traditional raster techniques such as cube maps and depth maps.

Instead of replacing rasterization entirely, however, the information gathered from ray-tracing can be used to **augment the shading with information that is much more photo-realistic**, especially in regards to off-camera action.

## GPUs

GPUs have been developed as a consequence of the strong request of faster graphic cards coming from the gaming industry.

Then, people realized that the computing power provided by GPUs could also be exploited for different tasks than computer graphics: for example for scientific simulation, data mining and obviously for multimedia processing.

## GPUs

Use **massively parallel approach** to hide latency:

- A lot more threads than cores
- Very low task switching overhead
- Threads are not swapped in/out, registers are mapped to threads for entire lifetime

## GPUs

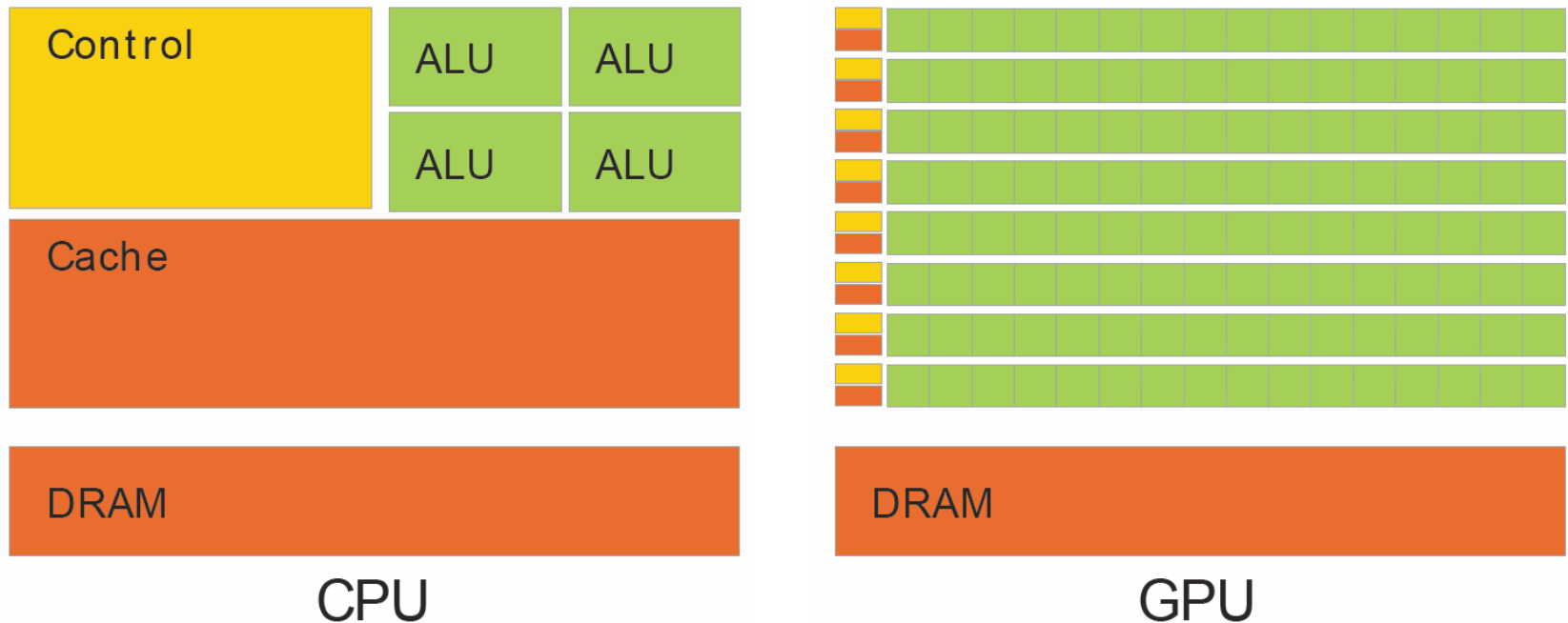
Even with massive bandwidth, memory is very slow relative to compute.

GPUs hide memory latency through fast thread context switch.

As a result of the parallel architecture:

- Memory transfers and computes overlap
- Hide memory latency with compute and compute with memory transfers

## CPU vs. GPU



**Host:** the CPU and its memory (host memory)

**Device:** the GPU and its memory (device memory)

## GPU Details

### Main hardware features:

- Multi-core design
- SIMD core optimized for floating point
- Dozens of multi cores per card
- User controlled caches per multi core (shared memory)
- Read only 2D optimized caches (textures)
- Real L1/L2 caches on from the Fermi generation

## Parallel Processing with GPUs

### Programming solutions:

- **CUDA Toolkit** (C/C++, Fortran, Python): Nvidia
- **OpenCL**: open standard, AMD, Nvidia, Intel, Apple, ...
- **Vulkan API**: open standard, AMD, Nvidia, Intel, Apple, ...
- **DirectCompute**: part of the Microsoft DirectX APIs
- **OpenACC**: collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator
- **C++ AMP** (Accelerated Massive Parallelism): library implemented on DirectCompute, with language extensions
- **OpenMP 4.5**: adds support for hardware accelerators



# CUDA Tools Ecosystem



## Accelerated Solutions

GPUs are accelerating many applications across numerous industries.

[Learn more >](#)



## Numerical Analysis Tools

GPU acceleration for applications with high arithmetic density.

[Learn more >](#)



## GPU-Accelerated Libraries

Application accelerating can be as easy as calling a library function.

[Learn more >](#)



## Language and APIs

GPU acceleration can be accessed from most popular programming languages.

[Learn more >](#)



## Performance Analysis Tools

Find the best solutions for analyzing your application's performance profile.

[Learn more >](#)



## Debugging Solutions

Powerful tools can help debug complex parallel applications in intuitive ways.

[Learn more >](#)



## Key Technologies

Learn more about parallel computing technologies and architectures.

[Learn more >](#)



## Accelerated Web Services

Micro services with visual and intelligent capabilities using deep learning.

[Learn more >](#)



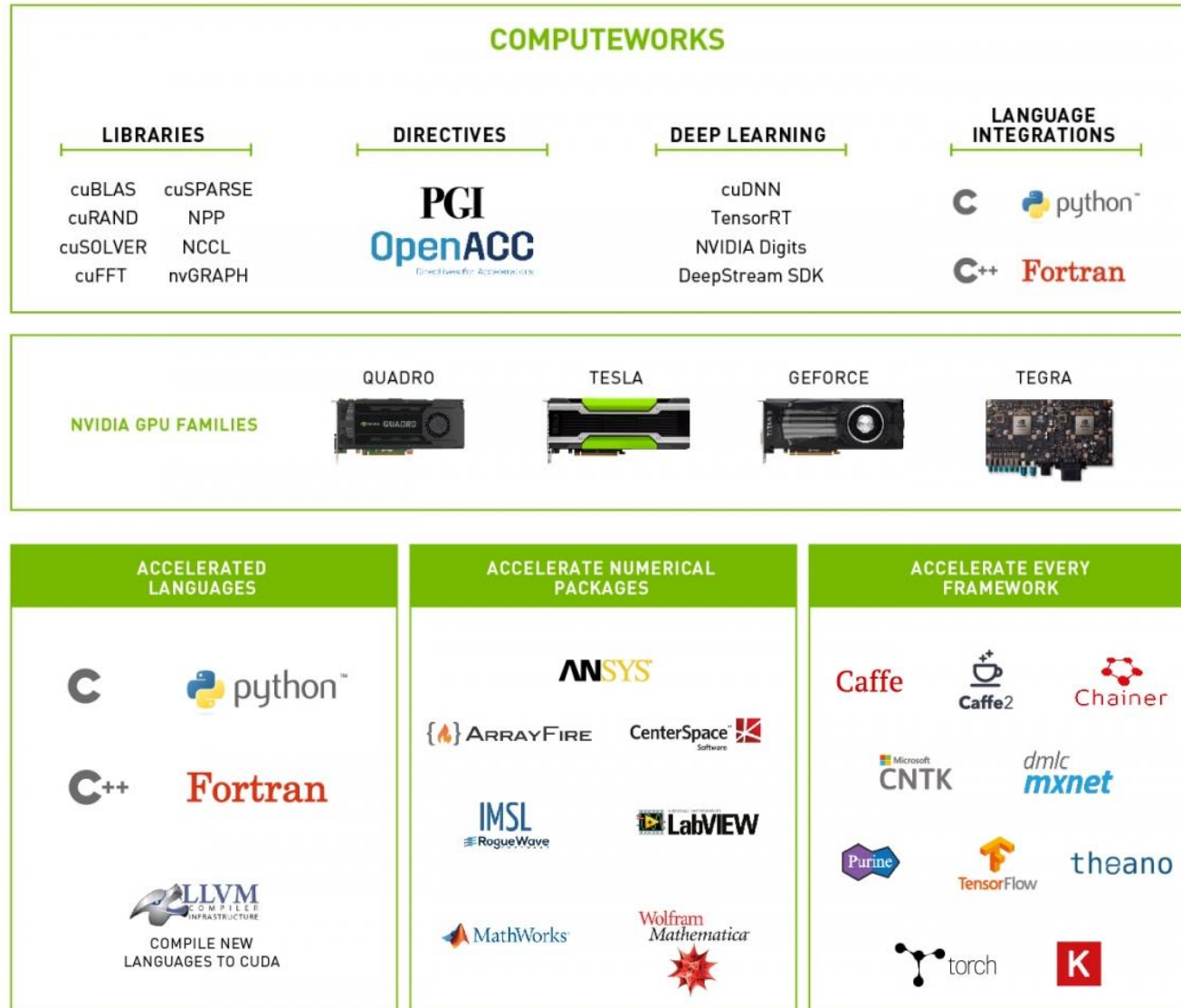
## Cluster Management

Managing your cluster and job scheduling can be simple and intuitive.

[Learn more >](#)

<https://developer.nvidia.com/tools-ecosystem>

# Nvidia ComputeWorks



# CUDA Parallel Computing Platform

Programming  
Approaches

Libraries

“Drop-in”  
Acceleration

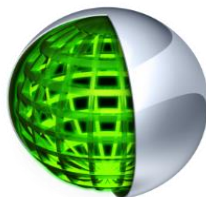
OpenACC  
Directives

Easily Accelerate  
Apps

Programming  
Languages

Maximum Flexibility

Development  
Environment



Nsight IDE (Eclipse, VS)  
Linux, Mac and Windows  
GPU Debugging and  
Profiling

CUDA-GDB  
debugger  
NVIDIA Visual  
Profiler

Open Compiler  
Tool Chain



Enables compiling new languages to CUDA  
platform, and CUDA languages to other  
architectures

## Libraries: Easy Acceleration

- **Ease of use:** using libraries enables GPU acceleration without in-depth knowledge of GPU programming.
- **“Drop-in”:** many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes.
- **Quality:** libraries offer high-quality implementations of functions encountered in a broad range of applications.
- **Performance:** libraries are tuned by experts.

## Use of Libraries

Step 1: Substitute library calls with equivalent in library:

```
saxpy ( ... ) -> cublasSaxpy ( ... )
```

Step 2: Manage data locality

- with CUDA: `cudaMalloc()`, `cudaMemcpy()`, etc.

- with CUBLAS: `cublasAlloc()`, `cublasSetVector()`, etc.

Step 3: Rebuild and link the CUDA-accelerated library

```
nvcc myobj.o -l cublas
```

## OpenACC


- Simple compiler hints
- Compiler parallelizes code
- Works on many-core GPUs & multi-core CPUs
- **Easy:** Directives are the easy path to accelerate compute intensive applications.
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors.
- **Powerful:** GPU directives allow complete access to the massive parallel power of a GPU.

## C Version (Sequential)

```
while (error > tol && iter < iter_max) {  
    error = 0.0;  
  
    for(int j = 1; j < n-1; j++) {  
        for(int i = 1; i < m-1; i++) {  
  
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +  
                                A[j-1][i] + A[j+1][i]);  
  
            error = max(error, abs(Anew[j][i] - A[j][i]));  
        }  
    }  
  
    for(int j = 1; j < n-1; j++) {  
        for(int i = 1; i < m-1; i++) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
  
    iter++;  
}
```



Iterate until  
converged



Iterate across matrix  
elements



Calculate new value  
from neighbors



Compute max error  
for convergence



Swap input/output  
arrays

# OpenACC Version

```

#pragma acc data copy(A), create(Anew)
while (error > tol && iter < iter_max) {
    error = 0.0;

    #pragma acc kernels
    for(int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                A[j-1][i] + A[j+1][i]);

            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc kernels
    for(int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}

```

Copy A in and out at start/end. Allocate Anew on accelerator

Iterate until converged

Execute GPU Kernel for loop

Calculate new value from neighbors

Compute max error for convergence

Execute GPU Kernel for loop



## What is CUDA

The **Compute Unified Device Architecture** (CUDA) is a **parallel computing platform and programming model** invented by NVIDIA that enables harnessing the power of graphics processing units (GPUs).

From its introduction in 2006, CUDA has been widely deployed, supported by an installed base of over **300 million CUDA-enabled GPUs** in notebooks, workstations, compute clusters and supercomputers.

Today, GPU-accelerated applications are available for **astronomy, biology, chemistry, physics, data mining, manufacturing, finance, ...**

## GPU Details

The NVIDIA GPU architecture is built around a **scalable array of multithreaded *Streaming Multiprocessors (SMs)***.

A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called **SIMT** (Single-Instruction, Multiple-Thread).

The instructions are pipelined to leverage **instruction-level parallelism within a single thread, as well as thread-level parallelism** extensively through simultaneous hardware multithreading.

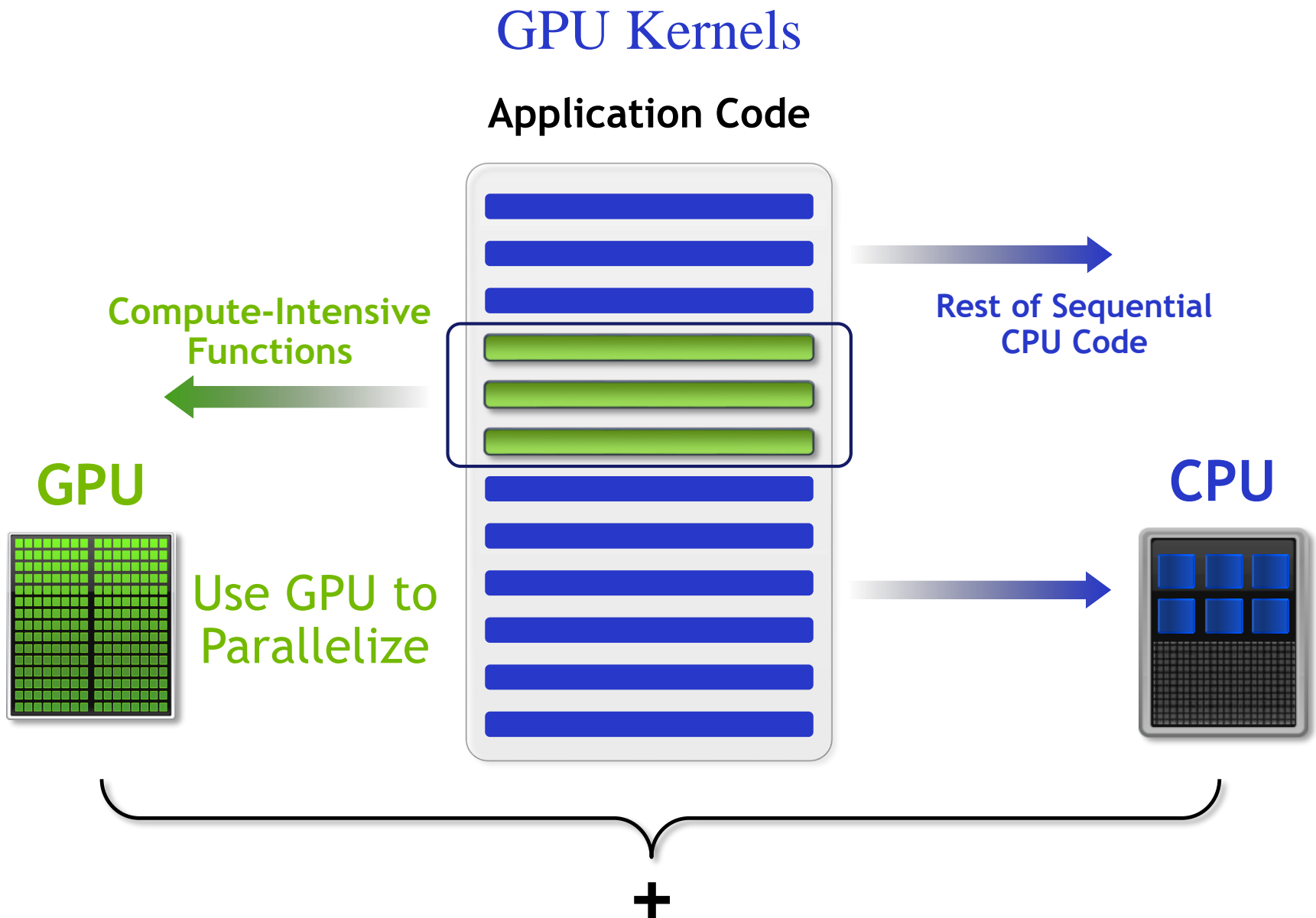
However, unlike CPU cores, **instructions are issued in order** and there is no branch prediction and no speculative execution.

## CUDA Programming Model

The GPU is seen as a compute device to execute a portion of an application that:

- Has to be executed many times
- Can be isolated as a function
- Works independently on different data

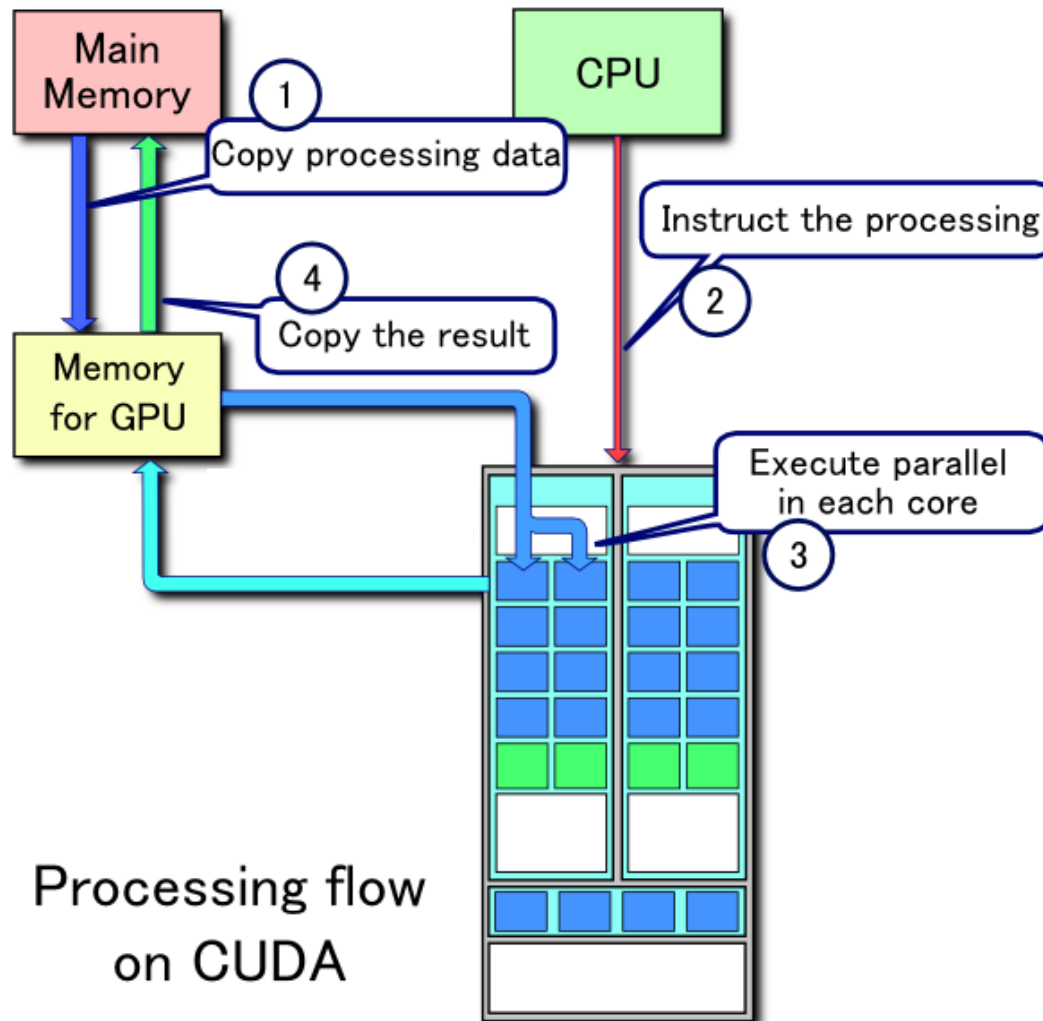
Such a function can be compiled to run on the device. The resulting program is called a **kernel**.



## What is a Kernel

- Calculation on the GPU is done via **kernels**.
- A kernel is a function callable from the host and executed on the CUDA device.
- **Run simultaneously by many threads in parallel.**
- Each kernel does as little work as possible.

# GPU Processing Flow



## Unified Memory

Unified memory is a memory management system that simplifies GPU development by providing a **single memory space directly accessible by all GPUs and CPUs** in the system, with automatic page migration for data locality.

Migration of pages allows the accessing processor to benefit from L2 caching and the lower latency of local memory.

Moreover, migrating pages to GPU memory ensures GPU kernels take advantage of the very high bandwidth of GPU memory (e.g. 720 GB/s on a Tesla P100).

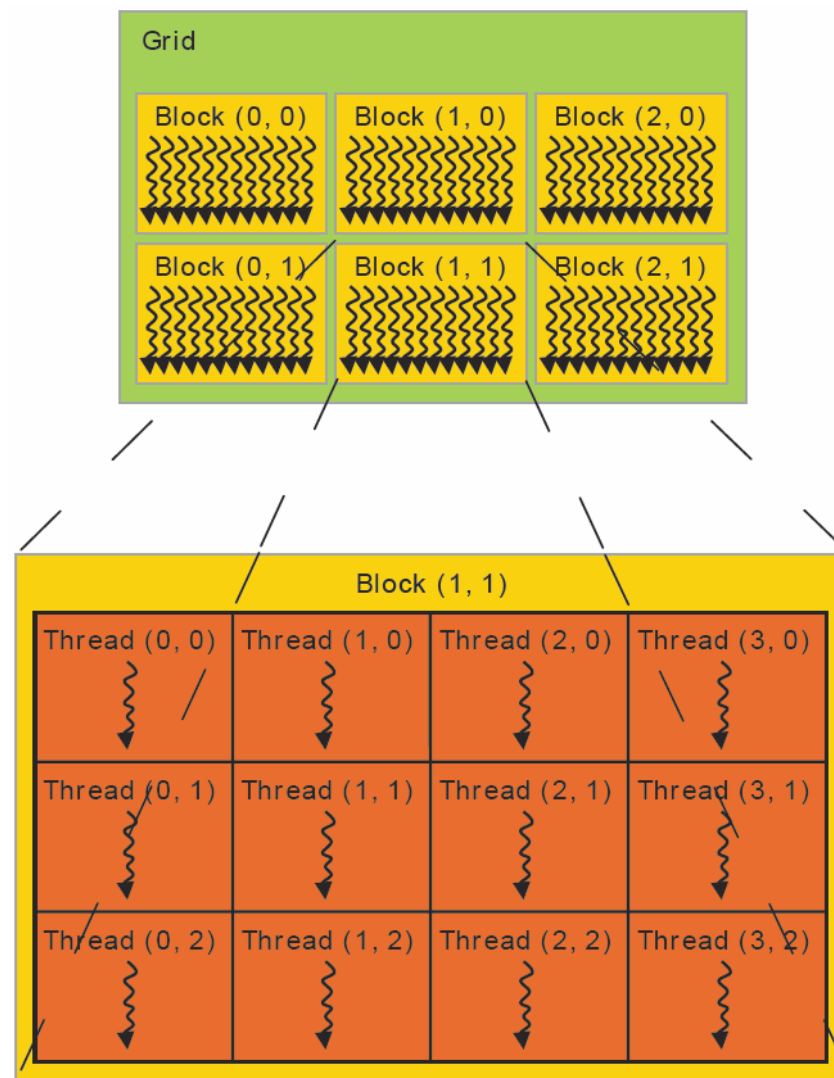
## Unified Memory

Page migration is all **completely invisible** to the developer: the system automatically manages all data movement for you.

Starting from the Pascal GPU architecture unified memory is more powerful, thanks to larger virtual memory address space and page migration engine, enabling true virtual memory demand paging.



# Kernel Execution



## Kernel Execution

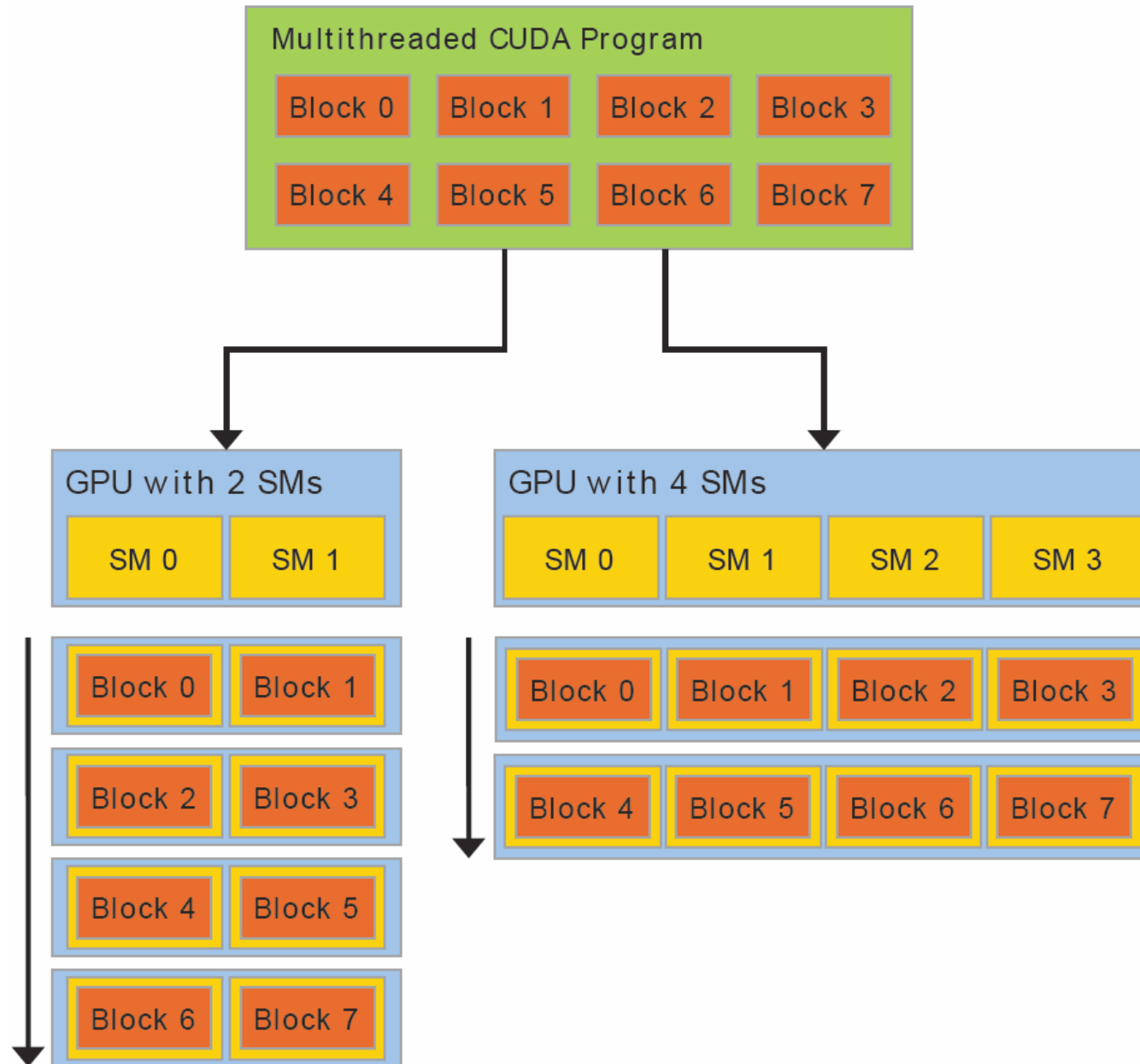
When a CUDA program on the host CPU invokes a **kernel grid**, the blocks of the grid are distributed to multiprocessors with available execution capacity. **The threads of a thread block execute concurrently on one multiprocessor**, and multiple thread blocks can execute concurrently on one multiprocessor.

As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

The batch of threads that executes a kernel is organized as a **grid of thread blocks**.

## Grids of Thread Blocks

- Limited number of threads in a block.
- Allows larger numbers of thread to execute the same kernel with one invocation.
- Blocks identifiable via block ID.
- Blocks can be one- or two-dimensional arrays.



## Kernel Execution

The multiprocessor creates, manages, schedules, and **executes threads in groups of 32 parallel threads called warps.**

Individual threads composing a warp start together at the same program address, but they have **their own instruction address counter and register state** and are therefore free to branch and execute independently.

The way a block is partitioned into warps is always the same; **each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.**

Blocks are not swapped out until they finish all their work.

## Kernel Execution

A warp executes one common instruction at a time, so **full efficiency is realized when all 32 threads of a warp agree on their execution path.**

If threads of a warp diverge via a **data-dependent conditional branch**, the warp **serially executes each branch path** taken, disabling threads that are not on that path.

Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

## Kernel Execution

If an **atomic instruction** executed by a warp reads, modifies, and writes to the same location in global memory for more than one of the threads of the warp, **each read/modify/write to that location occurs and they are all serialized, but the order in which they occur is undefined.**

## SIMT vs. SIMD

Compared to SIMD, SIMT enables programmers to write **thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads.**

For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, **substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge.**



