

General Purpose Computing On GPU

Computer Architecture and Organization | 20/12/2017

Term Report

Computer Architecture And organization

# Submitted By:

Muhammad Saad Qadri (BCS02153218)

Muhammad Afzal Arshad (BCS02153259)

Muhammad Khubaib Waheed (BCS02153277)

Contents

1. [Introduction ............................................................................. page 2](#p2)
2. [History of GPGPU.................................................................... page](#p2) 3
3. [Architecture of GPU................................................................. page](#p2) 4
   1. GPU hardware architecture………..……………………… page 4
   2. Modern CPUs ………..……………..……………………….page 4
   3. No focus on single instruction stream performance …... .Page 6
   4. Share instructions between streams ……………..……... Page 8
   5. A closer look at real GPU ………………………………...page 10
      1. NVIDIA GeForce GTX 480 (Fermi)…… ……….………page 10
4. [CPU vs GPU……………....................................................... page 13](#p3)
   1. Design……………………………………………..………page 13
   2. Performance………...……………………………………page 13
5. [GPU accelerated computing................................................. page](#p4) 16

# How GPU accelerate Applications …………….……….page 16

1. [General-purpose computing with GPU...................................page 1](#p6)7
2. [Compute Unified Device Architecture (CUDA)........................page](#p9) 18
   1. Codding In CUDA ……………………………………. page 19
3. [References………………………………………........................page](#p9) 20

Abstract

GPU is a processor optimized for 2D/3D graphics, video, visual computing, and display. It is highly parallel optimized for visual computing. To better understand the differences between CPU and GPU architectures we start out with a CPU architecture and make several key changes until we have a GPU like architecture. NVIDIA GeForce GTX 480 has 15 cores and16 SIMD functional units per core.

It is important to note that GPGPU is only effective for problems that can be solved efficiently by [parallel processing](https://en.wikipedia.org/wiki/Stream_processing). CUDA allows [software developers](https://en.wikipedia.org/wiki/Software_developer) to use a CUDA-enabled  GPU for general purpose processing.

1. Introduction

**General-purpose computing on graphics processing units** (**GPGPU**) is the use of a [graphics processing unit](https://en.wikipedia.org/wiki/Graphics_processing_unit) (GPU), which typically handles computation only for [computer graphics](https://en.wikipedia.org/wiki/Computer_graphics), to perform computation in applications traditionally handled by the [central processing unit](https://en.wikipedia.org/wiki/Central_processing_unit) (CPU). The use of multiple [video cards](https://en.wikipedia.org/wiki/Video_card) in one computer, or large numbers of graphics chips, further parallelizes the already parallel nature of graphics processing. In addition, even a single GPU-CPU framework provides advantages that multiple CPUs on their own do not offer due to the specialization in each chip.

Essentially, a GPGPU [pipeline](https://en.wikipedia.org/wiki/Graphics_pipeline) is a kind of [parallel processing](https://en.wikipedia.org/wiki/Parallel_computing) between one or more GPUs and CPUs that analyzes data as if it were in image or other graphic form. While GPUs operate at lower frequencies, they typically have many times the number of [cores](https://en.wikipedia.org/wiki/Multi-core_processor). Thus, GPUs can process far more pictures and graphical data per second than a traditional CPU. Migrating data into graphical form and then using the GPU to scan and analyze it can create a large [speedup](https://en.wikipedia.org/wiki/Speedup).

GPGPU pipelines were developed at the beginning of the 21st century for [graphics processing](https://en.wikipedia.org/wiki/Graphics_processing) (e.g., for better shaders). These pipelines were found to fit [scientific computing](https://en.wikipedia.org/wiki/Scientific_computing) needs well, and have since been developed in this direction.

1. History

General-purpose computing on GPUs only became practical and popular after about 2001, with the advent [floating point](https://en.wikipedia.org/wiki/Floating_point) support on graphics processors. Notably, problems involving [matrices](https://en.wikipedia.org/wiki/Matrix_(mathematics)) and/or [vectors](https://en.wikipedia.org/wiki/Vector_(mathematics_and_physics)) – especially two-, three-, or four-dimensional vectors – were easy to translate to a GPU, which acts with native speed and support on those types. The scientific computing community's experiments with the new hardware began with a [matrix multiplication](https://en.wikipedia.org/wiki/Matrix_multiplication) routine (2001); one of the first common scientific programs to run faster on GPUs than CPUs was an implementation of [LU factorization](https://en.wikipedia.org/wiki/LU_factorization) (2005).

These early efforts to use GPUs as general-purpose processors required reformulating computational problems in terms of graphics primitives, as supported by the two major APIs for graphics processors, [OpenGL](https://en.wikipedia.org/wiki/OpenGL) and [DirectX](https://en.wikipedia.org/wiki/DirectX). This cumbersome translation was obviated by the advent of general-purpose programming languages and APIs such as [Sh](https://en.wikipedia.org/wiki/Lib_Sh)/Rapid Mind, [Brook](https://en.wikipedia.org/wiki/BrookGPU) and Accelerator.

These were followed by Nvidia's [CUDA](https://en.wikipedia.org/wiki/CUDA), which allowed programmers to ignore the underlying graphical concepts in favor of more common [high performance](https://en.wikipedia.org/wiki/High-performance_computing) computing concepts. Newer, hardware vendor-independent offerings include Microsoft's [DirectCompute](https://en.wikipedia.org/wiki/DirectCompute) and Apple/Khronos Group's [OpenCL](https://en.wikipedia.org/wiki/OpenCL). This means that modern GPGPU pipelines can leverage the speed of a GPU without requiring full and explicit conversion of the data to a graphical form.

1. Architecture of GPU

# GPU hardware architecture

The hardware architecture of a graphics processing unit differs from that of a normal CPU in several key aspects. These differences originate in the special conditions in the field of real-time computer graphics:

* Many objects like pixels and vertices can be handled in isolation and are not interdependent.
* There are many independent objects (millions of pixels, thousands of vertices, …).
* Many objects require expensive computations.

The GPU architectures evolved to meet these requirements. To better understand the differences between CPU and GPU architectures we start out with a CPU architecture and make several key changes until we have a GPU like architecture.

# Modern CPUs

Most modern CPUs that are used in high performance computing as well as servers and desktop systems are much more complex than the simple calculations machines they are often programmed as. To be able to calculate something a CPU needs:

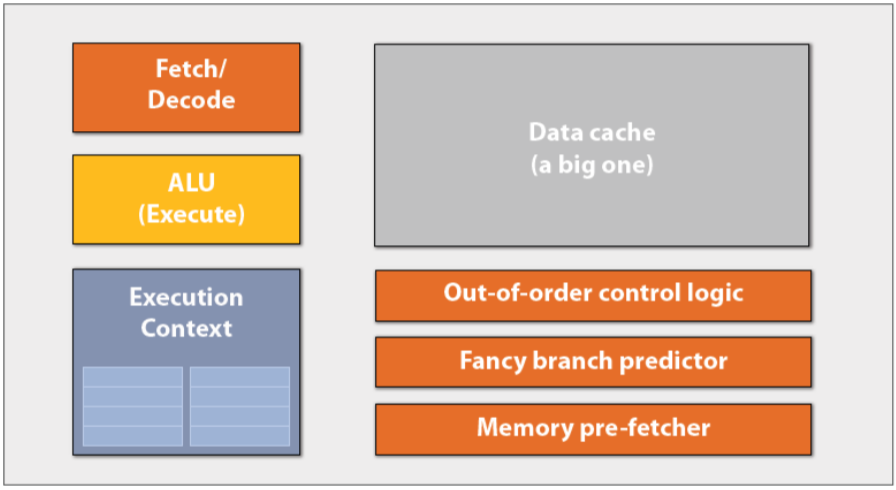
* To fetch & decode instructions from the memory
* An execution unit that does the calculations (ALU, FPU, …)
* Some kind of execution context (registers)

With that alone however a CPU would be very slow due to some effects:

* **Memory latency:** Fetching data from the off chip main memory is a very time consuming operation. If an execution unit needs some data for a calculation, it's stalled until the data is available, potentially waiting for a very long time.
* **Suboptimal program flow:** The way a program uses the execution units of the CPU may be inefficient and leaves some of the execution units idle.

CPUs contain several additional complex subsystems in order to overcome these problems and to boost the performance:

* Out-of-order execution
* Branch prediction
* Speculative Execution
* Memory pre-fetching
* Cache hierarchy

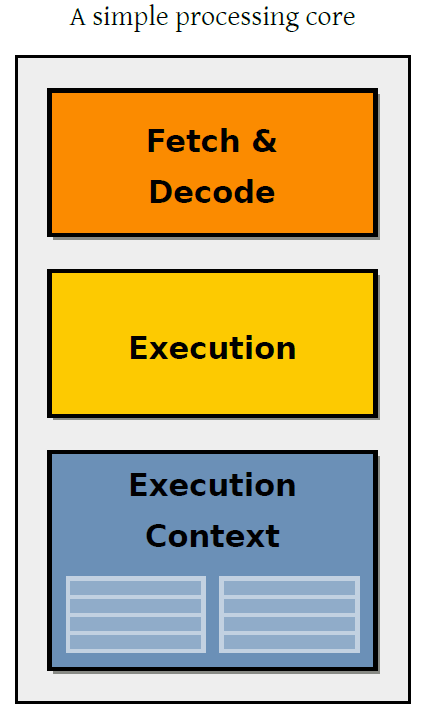


The purpose of all these optimizations is to improve the performance of a single instruction stream. Since CPUs traditionally run only one program at a time this was the only performance that mattered for a long time. However, the CPU architecture is aware of the fact that multiple programs are run in a time sliced manner, e.g. by providing a memory management unit and a translation look aside buffer to efficiently implement virtual memory.

During the last years this fact has also been used to optimize the performance by adding hardware to handle a second instruction stream (e.g. a second “Fetch & Decode” block). If one instruction stream is stalled because of a mispredicted branch the second stream can be fed to the execution units, resulting in a better overall utilization of the execution units.

# No focus on single instruction stream performance

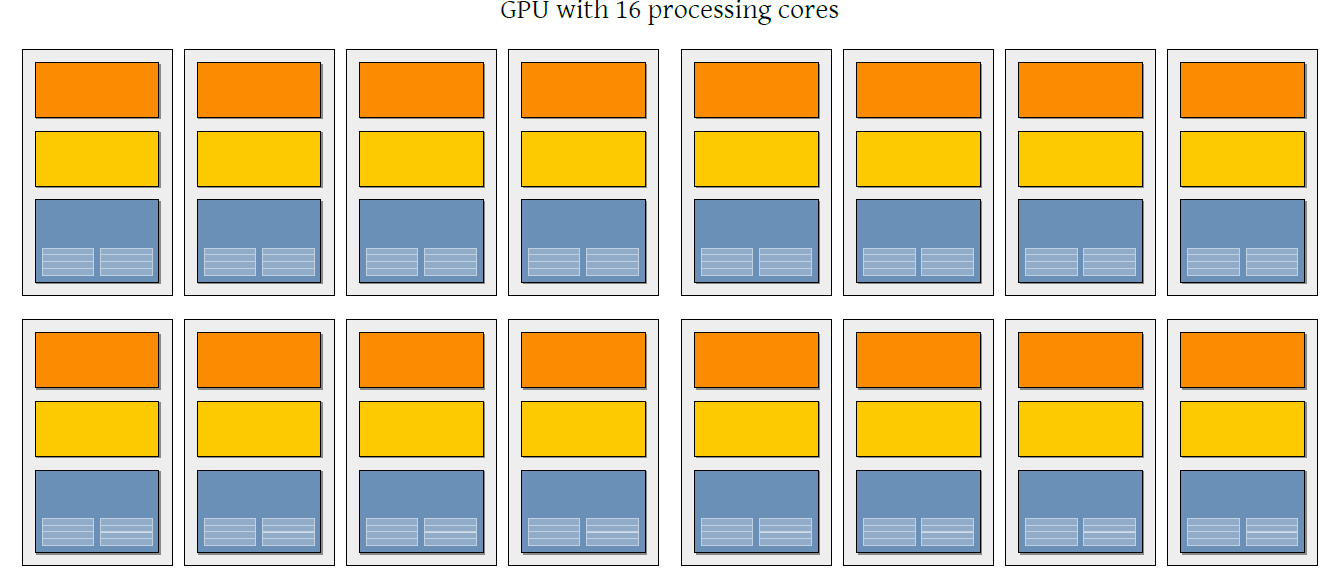
Tasks in computer graphics usually offer good parallelization potential. Many operations need to be done on all pixels of an image or on all vertices of a scene. These operations can usually be computed more or less independent of each other. From the hardware architecture point of view this kind of workload can be distributed over many cores and does not require strict sequential operation like many CPU based algorithms do.



In order to achieve a large number of cores these cores need to be simple. Therefore, we reduce the processing core to the absolutely required minimum:

* Instruction fetch and decode
* Execution unit
* Execution context

We have effectively removed all logic that boosts single instruction stream performance but gained the ability to put more cores on a chip. This increases the potential calculation power of our architecture. For example, we can now put 16 simple processing cores together and process 16 instruction streams in parallel.



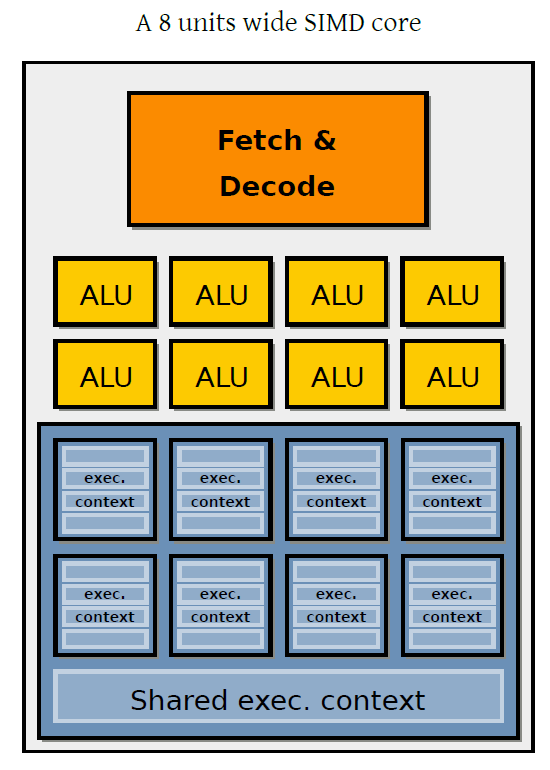
# Share instructions between streams

When executing a program that calculates a pixel the same instructions has to be applied to a very large amount of data. On a picture with the dimensions of 1280×1024 the same program has to be executed 1,310,720 times, once for each pixel. In our current 16 core design we would process all these pixels in blocks of 16 at a time.

However most of these 1,310,720 instruction streams do exactly the same. They are executing the same instruction just on different data. This gives us the opportunity for another architecture optimization.

# single instruction multiple data (SIMD) processing:

Instead of one decode & fetch unit per instruction stream we reuse the decode & fetch unit for several instruction streams. With this optimization our new SIMD processing core consists of one decode & fetch unit and a number of execution units, each with an associated execution context. For example, an 8 units wide SIMD processing core would consist of one decode & fetch unit, 8 execution units and 8 execution contexts.



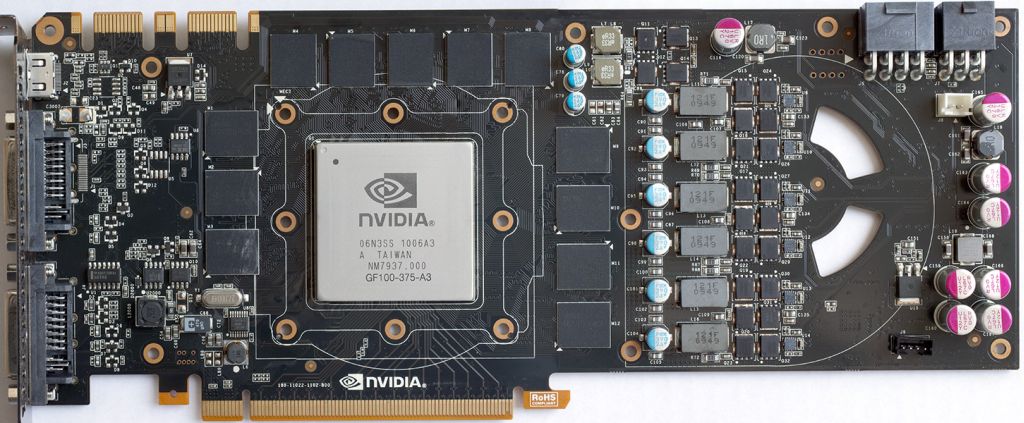
If all 8 execution units share the same instruction all works well and we get 8 times the performance without much organizational overhead. But if the shared instruction stream contains branch instructions it can happen that not all 8 execution units continue with the same instruction. A branch instruction like a conditional jump continues the control flow of the program at different positions depending on a runtime condition (e.g. the value of a variable or pixel coordinate). With such an instruction it can happen that some execution units of one SIMD core need to continue execution on another position than the remaining execution units. In that case we have to execute the instructions for both branches. The execution units not in the currently executed branch then need to ignore these instructions.

The amount of branching in the shared instruction stream limits the usefulness of SIMD optimizations. The more branching occurs within the same SIMD core the lower the SIMD performance gain becomes.

If we update our current 16 core architecture that way we get 16 SIMD processing cores, each able to do the same operation on 8 data streams. In the optimal case we can now process 16 × 8 = 128 programs in parallel. Note that we can handle one unique instruction stream per SIMD core without performance loss since the SIMD cores are independent of each other. If however the control flow within one SIMD core varies we lose performance since the SIMD core has to execute the instructions of each branch sequentially.

# A closer look at real GPU

# NVIDIA GeForce GTX 480 (Fermi)



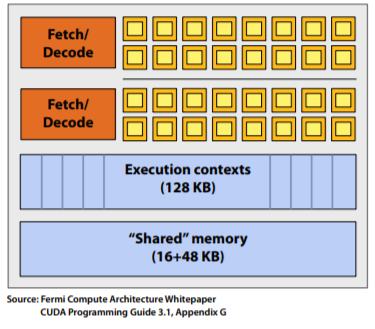
**• NVIDIA-speak:**

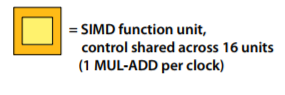
* 480 stream processors
* “SIMT execution”

**• Generic speak:**

* 15 cores
* 2 groups of 16 SIMD functional units per core

**NVIDIA GeForce GTX 480 “core”:**



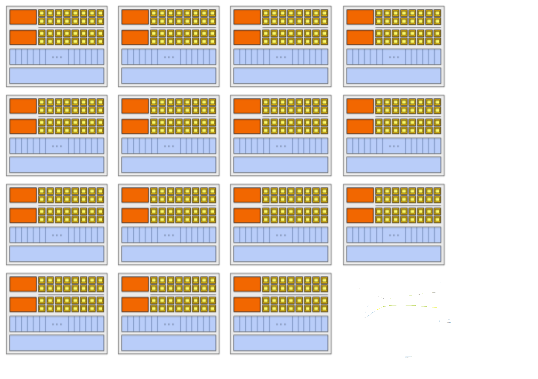


* The core contains 32 functional units
* Two groups are selected each clock (decode, fetch, and execute two instruction streams in parallel)

**There are 15 of these things on the GTX 480:**

That’s 23,000 fragments!

Or 23,000 CUDA threads!



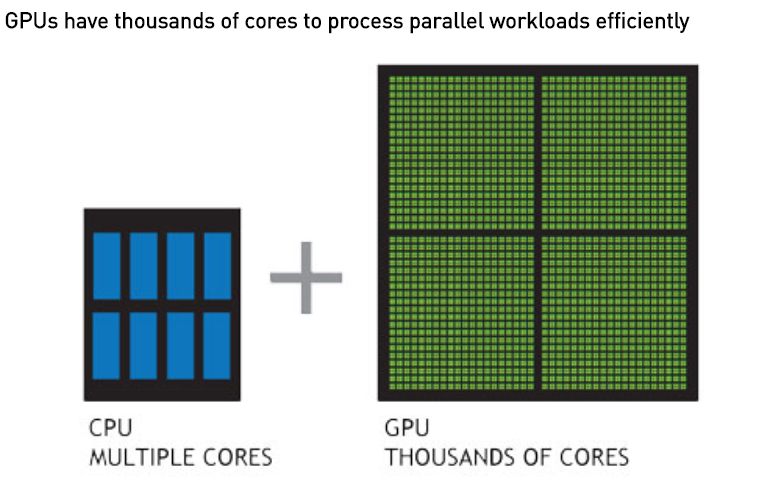
1. CPU vs GPU

# Design

CPU is designed to run 1 thread at a time and to perform very complex and general computations with a lot of branching. A CPU with 4 hex-core processors can run only 24 threads concurrently (or 48 if Hyper-Threading is supported). By comparison, the smallest executable unit of parallelism on a CUDA device comprises 32 threads. Latest NVIDIA GPUs support up to 2048 active threads concurrently per multiprocessor. On GPUs with 16 multiprocessors, this leads to more than 32,000 concurrently active threads.

# Performance

A simple way to understand the difference between a GPU and a CPU is to compare how they process tasks. A CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously.



# Comparing performance

# Intel 3.0 GHz Pentium 4

* 12 GFLOPs peak (MAD)
* 5.96 GB/s to main memory

# ATI Radeon X1800XT

* 120 GFLOPs peak (fragment engine)
* 42 GB/s to video memory

# Testing – Matrices

* Test the multiplication of two matrices.
* Two matrices with random floating point values.

# Results

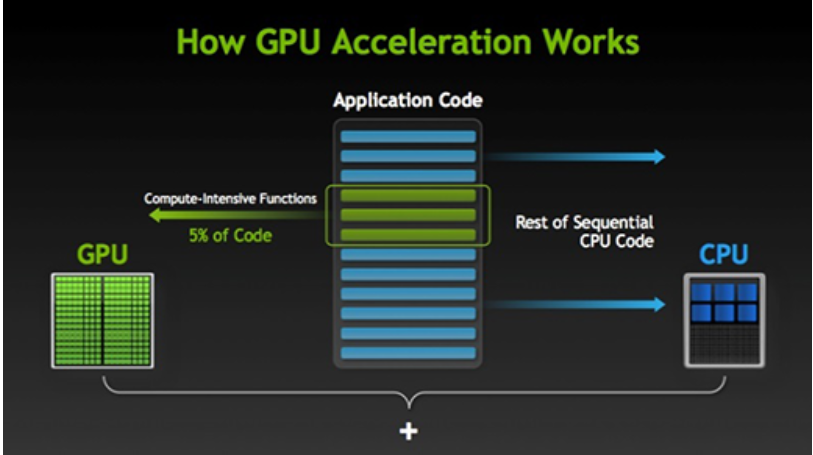
|  |  |  |
| --- | --- | --- |
| Matrix | GPU | CPU |
| 64x64 | 0.417465 ms | 18.0876 ms |
| 128x128 | 0.41691 ms | 18.3007 ms |
| 256x256 | 2.146367 ms | 145.6302 ms |
| 512x512 | 8.093004 ms | 1494.7275 ms |
| 768x768 | 25.97624 ms | 4866.3246 ms |
| 1024x1024 | 52.42811 ms | 66097.1688 ms |
| 2048x2048 | 407.648 ms | Didn’t finish |
| 4096x4096 | 3.1 seconds | Didn’t finish |

1. GPU accelerated computing

GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate [deep learning](http://www.nvidia.com/object/deep-learning.html), [analytics](http://www.nvidia.com/object/data-science-analytics-database.html), and [engineering](http://www.nvidia.com/object/computational-structural-mechanics.html) applications. Pioneered in 2007 by NVIDIA, GPU accelerators now power energy-efficient data centers in government labs, universities, enterprises, and small-and-medium businesses around the world. They play a huge role in accelerating applications in platforms ranging from artificial intelligence to cars, drones, and robots.

# HOW GPUs ACCELERATE SOFTWARE APPLICATIONS

GPU-accelerated computing offloads compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. From a user's perspective, applications simply run much faster.



1. General-purpose computing with GPU

Most of the calculations used in 3D graphics are based on matrices and vectors, and involve doing a great many similar, simple calculations simultaneously, so modern GPUs are highly optimized for these demands. However, 3D graphics is far from the only field where highly parallel matrix and vector math is used. Using a GPU for non-graphics applications is called [general-purpose computing on graphics processing units](https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units) (GPGPU). The data is usually provided in the form of a 2D grid, because GPUs are optimized for working with data in this form. However, the GPU performs the operation in parallel on multiple (perhaps all of the) elements in the grid simultaneously.

It is important to note that GPGPU is only effective for problems that can be solved efficiently by [stream processing](https://en.wikipedia.org/wiki/Stream_processing). While a GPU can solve problems where the sub-problems are heavily interconnected, it will be much slower at that problem than a CPU, because the parallelism doesn't help with that type of problem.

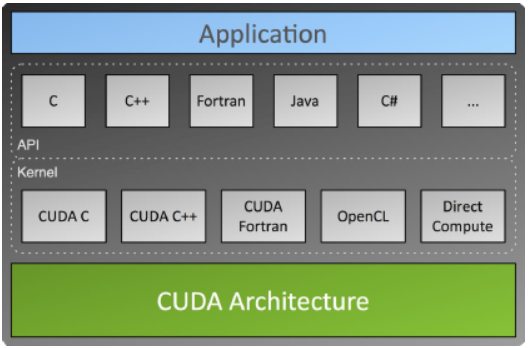
When designing algorithms for GPGPU, you should try to maximize the arithmetic intensity (the number of arithmetic operations performed per unit of data), to keep the processing from being slowed down by memory access.

While you can do GPGPU using graphics APIs or direct access to the GPU, APIs also exist that allow easier use of GPUs for GPGPU, including [OpenCL](https://en.wikipedia.org/wiki/OpenCL) and [OpenMP](https://en.wikipedia.org/wiki/OpenMP) (both of which are not GPGPU-specific; they can run on many kinds of processors).

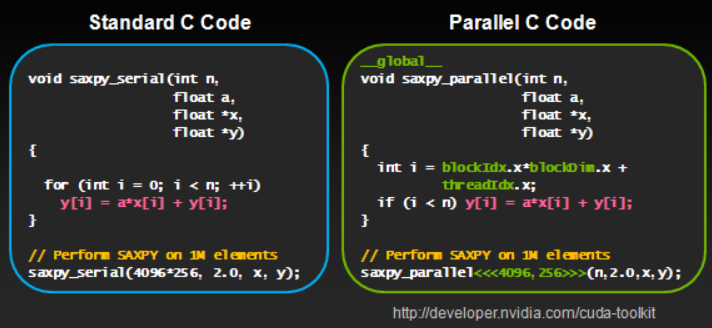
GPGPU is useful for many scientific applications such as weather forecasting and climate research, bioinformatics, and astrophysics, as well as non-scientific (more immediately profitable) applications including audio, video, and still image processing, machine learning, data mining, cryptography/cryptanalysis, and computer-aided design and engineering. GPUs and GPGPU are commonly used in supercomputers.

1. Compute Unified Device Architecture (CUDA)

CUDA is a [parallel computing](https://en.wikipedia.org/wiki/Parallel_computing) platform and [application programming interface](https://en.wikipedia.org/wiki/Application_programming_interface) (API) model created by [Nvidia](https://en.wikipedia.org/wiki/Nvidia). It allows [software developers](https://en.wikipedia.org/wiki/Software_developer) and [software engineers](https://en.wikipedia.org/wiki/Software_engineer) to use a CUDA-enabled [graphics processing unit](https://en.wikipedia.org/wiki/Graphics_processing_unit) (GPU) for general purpose processing.

The CUDA platform is designed to work with programming languages such as [C](https://en.wikipedia.org/wiki/C_(programming_language)), [C++](https://en.wikipedia.org/wiki/C%2B%2B), and [Fortran](https://en.wikipedia.org/wiki/Fortran). This accessibility makes it easier for specialists in parallel programming to use GPU resources, in contrast to prior APIs like [Direct3D](https://en.wikipedia.org/wiki/Direct3D) and [OpenGL](https://en.wikipedia.org/wiki/OpenGL), which required advanced skills in graphics programming. Also, CUDA supports programming frameworks such as [OpenACC](https://en.wikipedia.org/wiki/OpenACC) and [OpenCL](https://en.wikipedia.org/wiki/OpenCL). 

# Coding in CUDA



References

1. <http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf>
2. <http://www.nvidia.com/content/gtc-2010/pdfs/2275_gtc2010.pdf>
3. <http://www.nvidia.com/object/what-is-gpu-computing.html>
4. <https://www.tacc.utexas.edu/documents/13601/88790/8Things.pdf>
5. <https://tatourian.blog/2013/09/03/nvidia-gpu-architecture-cuda-programming-environment/>