# CPU vs GPU

At the start of multicore CPUs and GPUs the processor chips have become parallel systems. But speed of the program will be increased if software exploits parallelism provided by the underlying multiprocessor architecture. Hence there is a big need to design and develop the software so that it uses multithreading, each thread running concurrently on a processor, potentially increasing the speed of the program dramatically. To develop such a scalable parallel application, a parallel programming model is required that supports parallel multicore programming environment.

NVIDIA’s graphics processing units (GPUs) are very powerful and highly parallel. GPUs have hundreds of processor cores and thousands of threads running concurrently on these cores, thus because of intensive computing power they are much faster than the CPU as shown in Figure

At start, they were used for graphics purposes only. But now GPUs are becoming more and more popular for a variety of general-purpose, non-graphical applications too. For example, they are used in the fields of computational chemistry, sparse matrix solvers, physics models, sorting, and searching etc. The programs designed for GPGPU (General Purpose GPU) run on the multi processors using many threads concurrently. As a result, these programs are extremely fast.

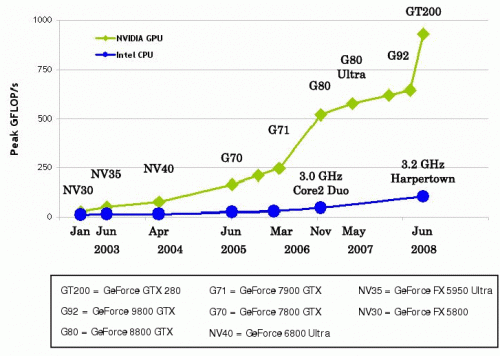


Figure 1: Performances of CPUs (circle) and GPUs (diamond) over the last few years.

# General-Purpose Computing on Graphics Processing Units

General-purpose computing on graphics processing units (GPGPU) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). The use of multiple video cards 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.

GPGPUs are used for tasks that were formerly the domain of high-power CPUs, such as physics calculations, encryption/decryption, generation of cryptocurrencies such as Bitcoin, and scientific computations. Because graphics cards are constructed for massive parallelism, they can dwarf the calculation rate of even the most powerful CPUs for many parallel processing tasks. The same shader cores that allow multiple pixels to be rendered simultaneously can similarly process multiple streams of data at the same time. Although a shader core is not nearly as complex as a CPU, a high-end GPU may have thousands of shader cores; in contrast, a multicore CPU might have eight or twelve cores.

There has been an increased focus on GPGPUs since DirectX 10 included unified shaders in its shader core specifications for Windows Vista. Higher-level languages are being developed all the time to ease programming for computations on the GPU. Both AMD/ATI and Nvidia have approaches to GPGPU with their own APIs (OpenCL and CUDA, respectively).

# What is CUDA?

CUDA stands for **Compute Unified Device Architecture**, and is an extension of the C programming language and was created by **NVIDIA**. Using CUDA allows the programmer to take advantage of the massive parallel computing power of an NVIDIA graphics card in order to do general purpose computation. CPUs like Intel Core 2 Duo and AMD Opteron are good at doing one or two tasks at a time, and doing those tasks very quickly. Graphics cards, on the other hand, are good at doing a massive number tasks at the same time, and doing those tasks relatively quickly.

To put this into perspective, suppose you have a 20-inch monitor with a standard resolution of 1920 x 1200. An NVIDIA graphics card has the computational ability to calculate the color of 2,304,000 different pixels, many times a second. In order to accomplish this feat, graphics cards use dozens, even hundreds of ALUs.

Fortunately, NVIDIA’s ALUs are fully programmable, which enables us to harness an unprecedented amount of computational power into the programs that we write.

As stated previously, CUDA lets the programmer take advantage of the hundreds of ALUs inside a graphics processor, which is much more powerful than the handful of ALUs available in any CPU. However, this does put a limit on the types of applications that are well suited to CUDA.

# When to Use CUDA?

## CUDA is only well suited for highly parallel algorithms

In order to run efficiently on a GPU, you need to have many hundreds of threads. Generally, the more threads you have, the better. If you have an algorithm that is mostly serial, then it does not make sense to use CUDA. Many serial algorithms do have parallel equivalents, but many do not. If you can’t break your problem down into at least a thousand threads, then CUDA probably is not the best solution for you.

## CUDA is extremely well suited for number crunching

If there is one thing that CUDA excels at, it’s number crunching. The GPU is fully capable of doing 32-bit integer and floating point operations. In fact, it GPUs are more suited for floating point computations, which makes CUDA an excellent for number crunching. Some of the higher end graphics cards do have double floating point units, however there is only one 64-bit floating point unit for every 16 32-bit floating point units. So, using double floating point numbers with CUDA should be avoided if they aren’t absolutely necessary for your application.

## CUDA is well suited for large datasets

Most modern CPUs have a couple megabytes of L2 cache because most programs have high data coherency. However, when working quickly across a large dataset, say 500 Megabytes, the L2 cache may not be as helpful. The memory interface for GPUs is very different from the memory interface of CPUs. GPUs use massive parallel interfaces in order to connect with its memory. This type of interface is approximately 10 times faster than a typical CPU to memory interface, which is great.