

8.9 Fallacies and pitfalls

(Original section¹)

GPUs have evolved and changed so rapidly that many fallacies and pitfalls have arisen. We cover a few here.

Fallacy: GPUs are just SIMD vector multiprocessors.

It is easy to draw the false conclusion that GPUs are simply SIMD vector multiprocessors. GPUs do have a SPMD-style programming model, in that a programmer can write a single program that is executed in multiple thread instances with multiple data. The execution of these threads is not purely SIMD or vector, however; it is *single-instruction multiple-thread* (SIMT), described in COD Section B.4 (Multithreaded multiprocessor architecture). Each GPU thread has its own scalar registers, thread private memory, thread execution state, thread ID, independent execution and branch path, and effective program counter, and can address memory independently. Although a group of threads (e.g., a warp of 32 threads) executes more efficiently when the PCs for the threads are the same, this is not necessary. So, the multiprocessors are not purely SIMD. The thread execution model is MIMD with barrier synchronization and SIMT optimizations. Execution is more efficient if individual thread load/store memory accesses can be coalesced into block accesses, as well. However, this is not strictly necessary. In a purely SIMD vector architecture, memory/register accesses for different threads must be aligned in a regular vector pattern. A GPU has no such restriction for register or memory accesses; however, execution is more efficient if warps of threads access local blocks of data.

In a further departure from a pure SIMD model, an SIMT GPU can execute more than one warp of threads concurrently. In graphics applications, there may be multiple groups of vertex programs, pixel programs, and geometry programs running in the multiprocessor array concurrently. Computing programs may also execute different programs concurrently in different warps.

Fallacy: GPU performance cannot grow faster than Moore's law.

Moore's law is simply a rate. It is not a "speed of light" limit for any other rate. Moore's law describes an expectation that, over time, as semiconductor technology advances and transistors become smaller, the manufacturing cost per transistor will decline exponentially. Put another way, given a constant manufacturing cost, the number of transistors will increase exponentially. Gordon Moore [1965] predicted that this progression would provide roughly two times the number of transistors for the same manufacturing cost every year, and later revised it to doubling every 2 years. Although Moore made the initial prediction in 1965 when there were just 50 components per integrated circuit, it has proved remarkably consistent. The reduction of transistor size has historically had other benefits, such as lower power per transistor and faster clock speeds at constant power.

This increasing bounty of transistors is used by chip architects to build processors, memory, and other components. For some time, CPU designers have used the extra transistors to increase processor performance at a rate similar to Moore's law, so much so that many people think that processor performance growth of two times every 18–24 months is Moore's law. In fact, it is not.

Microprocessor designers spend some of the new transistors on processor cores, improving the architecture and design, and pipelining for more clock speed. The rest of the new transistors are used for providing more cache, to make memory access faster. In contrast, GPU designers use almost none of the new transistors to provide more cache; most of the transistors are used for improving the processor cores and adding more processor cores.

GPUs get faster by four mechanisms. First, GPU designers reap the Moore's law bounty directly by applying exponentially more transistors to building more parallel, and thus faster, processors. Second, GPU designers can improve on the architecture over time, increasing the efficiency of the processing. Third, Moore's law assumes constant cost, so the Moore's law rate can clearly be exceeded by spending more for larger chips with more transistors. Fourth, GPU memory systems have increased their effective bandwidth at a pace nearly comparable to the processing rate, by using faster memories, wider memories, data compression, and better caches. The combination of these four approaches has historically allowed GPU performance to double regularly, roughly every 12 to 18 months. This rate, exceeding the rate of Moore's law, has been demonstrated on graphics applications for approximately 10 years and shows no sign of significant slowdown. The most challenging rate limiter appears to be the memory system, but competitive innovation is advancing that rapidly too.

Fallacy: GPUs only render 3D graphics; they can't do general computation.

GPUs are built to render 3D graphics as well as 2D graphics and video. To meet the demands of graphics software developers as expressed in the interfaces and performance/feature requirements of the graphics APIs, GPUs have become massively parallel programmable floating-point processors. In the graphics domain, these processors are programmed through the graphics APIs and with arcane graphics programming languages (GLSL, Cg, and HLSL, in OpenGL and Direct3D). However, there is nothing preventing GPU architects from exposing the parallel processor cores to programmers without the graphics API or the arcane graphics languages.

In fact, the Tesla architecture family of GPUs exposes the processors through a software environment known as CUDA, which allows programmers to develop general application programs using the C language and soon C++. GPUs are Turing-complete processors, so they can run any program that a CPU can run, although perhaps less well. And perhaps faster.

Fallacy: GPUs cannot run double-precision floating-point programs fast.

In the past, GPUs could not run double-precision floating-point programs at all, except through software emulation. And that's not very fast at all. GPUs have made the progression from indexed arithmetic representation (lookup tables for colors) to 8-bit integers per color component, to fixed-point arithmetic, to single-precision floating-point, and recently added double precision. Modern GPUs perform virtually all calculations in single-precision IEEE floating-point arithmetic, and are beginning to use double precision in addition.

For a small additional cost, a GPU can support double-precision floating-point as well as single-precision floating-point. Today, double-precision runs more slowly than the single-precision speed, about five to ten times slower. For incremental additional cost, double-precision performance can be increased relative to single precision in stages, as more applications demand it.

Fallacy: GPUs don't do floating-point correctly.

GPUs, at least in the Tesla architecture family of processors, perform single-precision floating-point processing at a level prescribed by the IEEE 754 floating-point standard. So, in terms of accuracy, GPUs are the equal of any other IEEE 754-compliant processors.

Today, GPUs do not implement some of the specific features described in the standard, such as handling denormalized numbers and providing precise floating-point exceptions. However, the recently introduced Tesla T10P GPU provides full IEEE rounding, fused-multiply-add, and denormalized number support for double precision.

Pitfall: Just use more threads to cover longer memory latencies.

CPU cores are typically designed to run a single thread at full speed. To run at full speed, every instruction and its data need to be available when it is time for that instruction to run. If the next instruction is not ready or the data required for that instruction is not available, the instruction cannot run and the processor stalls. External memory is distant from the processor, so it takes many cycles of wasted execution to fetch data from memory. Consequently, CPUs require large local caches to keep running without stalling. Memory latency is long, so it is

avoided by striving to run in the cache. At some point, program working set demands may be larger than any cache. Some CPUs have used multithreading to tolerate latency, but the number of threads per core has generally been limited to a small number.

The GPU strategy is different. GPU cores are designed to run many threads concurrently, but only one instruction from any thread at a time. Another way to say this is that a GPU runs each thread slowly, but in aggregate runs the threads efficiently. Each thread can tolerate some amount of memory latency, because other threads can run.

The downside of this is that multiple—many multiple threads—are required to cover the memory latency. In addition, if memory accesses are scattered or not correlated among threads, the memory system will get progressively slower in responding to each individual request. Eventually, even the multiple threads will not be able to cover the latency. So, the pitfall is that for the “just use more threads” strategy to work for covering latency, you have to have enough threads, and the threads have to be well-behaved in terms of locality of memory access.

Fallacy: $O(n)$ algorithms are difficult to speed up.

No matter how fast the GPU is at processing data, the steps of transferring data to and from the device may limit the performance of algorithms with $O(n)$ complexity (with a small amount of work per datum). The highest transfer rate over the PCIe bus is approximately 48 GB/second when DMA transfers are used, and slightly less for nonDMA transfers. The CPU, in contrast, has typical access speeds of 8–12 GB/second to system memory. Example problems, such as vector addition, will be limited by the transfer of the inputs to the GPU and the returning output from the computation.

There are three ways to overcome the cost of transferring data. First, try to leave the data on the GPU for as long as possible, instead of moving the data back and forth for different steps of a complicated algorithm. CUDA deliberately leaves data alone in the GPU between launches to support this.

Second, the GPU supports the concurrent operations of copy-in, copy-out and computation, so data can be streamed in and out of the device while it is computing. This model is useful for any data stream that can be processed as it arrives. Examples are video processing, network routing, data compression/decompression, and even simpler computations such as large vector mathematics.

The third suggestion is to use the CPU and GPU together, improving performance by assigning a subset of the work to each, treating the system as a heterogeneous computing platform. The CUDA programming model supports allocation of work to one or more GPUs along with continued use of the CPU without the use of threads (via asynchronous GPU functions), so it is relatively simple to keep all GPUs and a CPU working concurrently to solve problems even faster.

(*1) This section is in original form.

 [Provide feedback on this section](#)