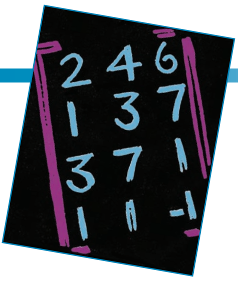
Editors: Konstantin Läufer, laufer@cs.luc.edu
Konrad Hinsen, hinsen@cnrs-orleans.fr



WHY MODERN CPUS ARE STARVING AND WHAT CAN BE DONE ABOUT IT

By Francesc Alted

CPUs spend most of their time waiting for data to arrive. Identifying low-level bottlenecks—and how to ameliorate them—can save hours of frustration over poor performance in apparently well-written programs.

A well-documented trend shows that CPU speeds are increasing at a faster rate than memory speeds.[1,2] Indeed, CPU performance has now outstripped memory performance to the point that current CPUs are starved for data, as memory I/O becomes the performance bottleneck.

This hasn't always been the case. Once upon a time, processor and memory speeds evolved in parallel. For example, memory clock access in the early 1980s was at approximately 1 MHz, and memory and CPU speeds increased in tandem to reach speeds of 16 MHz by decade's end. By the early 1990s, however, CPU and memory speeds began to drift apart: memory speed increases began to level off, while CPU clock rates continued to skyrocket to 100 MHz and beyond. It wasn't too long before CPU capabilities began to substantially outstrip memory performance. Consider this: a 100 MHz processor consumes a word from memory every 10 nanoseconds in a single clock tick. This rate is impossible to sustain even with present-day RAM, let alone with the RAM available when 100 MHz processors were state of the art. To address this mismatch, commodity chipmakers introduced the first on-chip cache.

But CPUs didn't stop at 100 MHz; by the start of the new millennium, processor speeds reached unparalleled extremes, hitting the magic 1 GHz figure. As a consequence, a huge abyss opened between the processors and the memory subsystem: CPUs had to wait up to 50 clock ticks for each memory read or write operation.

During the early and middle 2000s, the strong competition between Intel and AMD continued to drive CPU clock cycles faster and faster (up to 4 GHz). Again, the increased impedance mismatch with memory speeds forced vendors to introduce a second-level cache in CPUs. In the past five years, the size of this second-level cache grew rapidly, reaching 12 Mbytes in some instances.

Vendors started to realize that they couldn't keep raising the frequency forever, however, and thus dawned the multicore age. Programmers began scratching their heads, wondering how to take advantage of those shiny new and apparently innovative multicore machines. Today, the arrival of Intel i7 and AMD Phenom makes four-core on-chip CPUs the most common configuration. Of course, more processors means more demand for data, and vendors thus introduced a third-level cache.

So, here we are today: memory latency is still much greater than processor clock step (around 150 times greater or more) and has become an essential bottleneck over the past 20 years. Memory throughput is improving at a better rate than its latency, but it's also lagging behind processors (about 25 times slower). The result is that current CPUs are suffering from serious starvation: they're capable of consuming (much!) more data than the system can possibly deliver.

The Hierarchical Memory Model

Why, exactly, can't we improve memory latency and bandwidth to keep up with CPUs? The main reason is cost: it's prohibitively expensive to manufacture commodity SDRAM that can keep up with a modern processor. To make memory faster, we need motherboards with more wire layers, more complex ancillary logic, and (most importantly) the ability to run at higher frequencies. This additional complexity represents a much higher cost, which few are willing to pay. Moreover, raising the frequency implies pushing more voltage through the circuits. This causes the energy consumption to quickly skyrocket and more heat to be generated, which requires huge coolers in user machines. That's not practical.

To cope with memory bus limitations, computer architects introduced a hierarchy of CPU memory caches.[3] Such caches are useful because they're closer to the processor (normally in the same die), which improves both latency and bandwidth. The faster they run, however, the smaller they must be due mainly to energy dissipation problems. In response, the industry

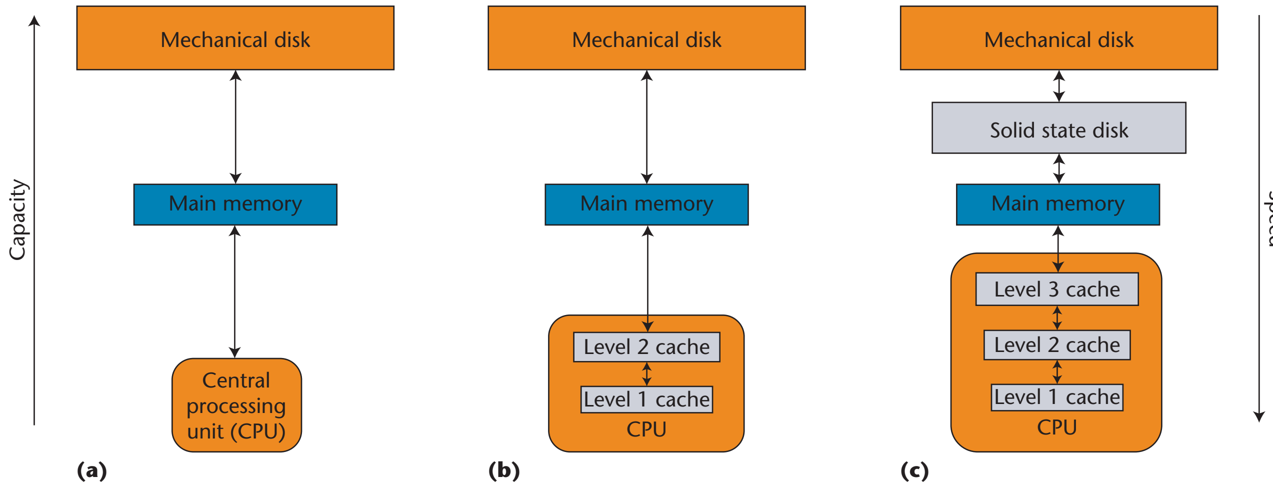


Figure 1. Evolution of the hierarchical memory model. (a) The primordial (and simplest) model; (b) the most common current implementation, which includes additional cache levels; and (c) a sensible guess at what's coming over the next decade: three levels of cache in the CPU and solid state disks lying between main memory and classical mechanical disks.

implemented several memory layers with different capabilities: lower-level caches (that is, those closer to the CPU) are faster but have reduced capacities and are best suited for performing computations; higher-level caches are slower but have higher capacity and are best suited for storage purposes.

Figure 1 shows the evolution of this hierarchical memory model over time. The forthcoming (or should I say the present?) hierarchical model includes a minimum of six memory levels. Taking advantage of such a deep hierarchy isn't trivial at all, and programmers must grasp this fact if they want their code to run at an acceptable speed.

Techniques to Fight Data Starvation

Unlike the good old days when the processor was the main bottleneck, memory organization has now become the key factor in optimization. Although learning assembly language to get direct processor access is (relatively) easy, understanding how the hierarchical memory model works—and adapting your data structures accordingly—requires considerable knowledge and experience. Until we have languages that facilitate the development of programs that are aware of memory hierarchy (for an example in progress, see the Sequoia project at www.stanford.edu/group/sequoia), programmers must learn how to deal with this problem at a fairly low level.[4]

There are some common techniques to deal with the CPU data-starvation problem in current hierarchical memory models. Most of them exploit the principles of temporal and spatial locality. In *temporal locality*, the target dataset is reused several times over a short period. The first time the dataset is accessed, the system must bring it to cache from slow memory; the next time, however, the processor will fetch it directly (and much more quickly) from the cache.

In *spatial locality*, the dataset is accessed sequentially from memory. In this case, circuits are designed to fetch memory elements that are clumped together much faster than if they're dispersed. In addition, specialized circuitry (even in current commodity hardware) offers *prefetching*—that is, it can look at memory-access patterns and predict when a certain chunk of data will be used and start to transfer it to cache before the CPU has actually asked for it. The net result is that the CPU can retrieve data much faster when spatial locality is properly used.

Programmers should exploit the optimizations inherent in temporal and spatial locality as much as possible. One generally useful technique that leverages these principles is the *blocking* technique (see Figure 2). When properly applied, the blocking technique guarantees that both spatial and temporal localities are exploited for maximum benefit.

Although the blocking technique is relatively simple in principle, it's less straightforward to implement in practice. For example, should the basic block fit in cache level one, two, or three? Or would it be better to fit it in main memory—which can be useful when computing large, disk-based datasets? Choosing from among these different possibilities is difficult, and there's no substitute for experimentation and empirical analysis.

In general, it's always wise to use libraries that already leverage the blocking technique (and others) for achieving high performance; examples include Lapack (www.netlib.org/lapack) and Numexpr (http://code.google.com/p/numexpr). Numexpr is a virtual machine written in Python and C that lets you evaluate potentially complex arithmetic expressions over arbitrarily large arrays. Using the blocking technique in combination

Compression and Data Access

Over the past 10 years, it's been standard practice to use compression to accelerate the reading and writing of large datasets to and from disks. Optimizations based on compression leverage the fact that it's generally faster to read and write a small (compressed) dataset than a larger (uncompressed) one, even when accounting for (de)compression time. So, given the gap between processor and memory speed, can compression also accelerate data transfer from memory to the processor?

The new blocking, shuffling, and compression (Blosc) library project uses compression to improve memory-access speed. Blosc is a lossless compressor for binary data that is optimized for speed rather than high compression ratios. It uses the blocking technique (which I describe in the main text) to reduce activity on the memory bus as much as possible. In addition, the shuffle algorithm maximizes the compression ratio of data stored in small blocks.

As preliminary benchmarks show, for highly compressible datasets, Blosc can effectively transmit compressed data from memory to CPU faster than it can transmit uncompressed data (see www.pytables.org/docs/StarvingCPUs.pdf). However, for datasets that compress poorly, transfer speeds still lag behind those of uncompressed datasets. As the gap between CPU and memory speed continues to widen, I expect Blosc to improve memory-to-CPU data transmission rates over an increasing range of datasets. You can find more information about Blosc at http://blosc.pytables.org.

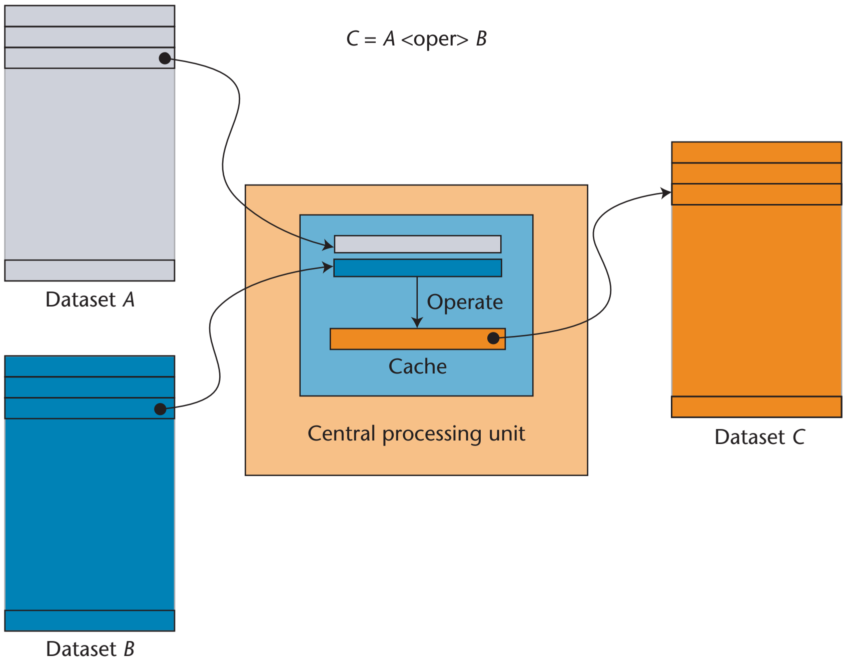


Figure 2. The blocking technique. The blocking approach exploits both spatial and temporal localities for maximum benefit by retrieving a contiguous block that fits into the CPU cache at each memory access, then operates upon it and reuses it as much as possible before writing the block back to memory.

with a specialized, just-in-time (JIT) compiler offers a good balance between cache and branch prediction, allowing optimal performance for many vector operations. However, for expressions involving matrices, Lapack is a better fit and can be used in almost any language. Also, compression is another field where the blocking technique can bring important advantages (see the sidebar, "Compression and Data Access").

Enter the Multicore Age

Ironically, even as the memory subsystem has increasingly lagged behind the processor over the past two decades, vendors have started to integrate several cores in the same CPU die, further exacerbating the problem. At this point, almost every new computer comes with several high-speed cores that share a single memory bus. Of course, this only starves the CPUs even more as several cores fight for scarce memory resources. To make this situation worse, you must synchronize the caches of different cores to ensure coherent access to memory, effectively increasing memory operations' latency even further.

To address this problem, the industry introduced the nonuniform memory access (NUMA) architecture in the 1990s. In NUMA, different memory banks are dedicated to different processors (typically in different physical sockets), thereby avoiding the performance hit that results when several processors attempt to address the same memory at the same time. Here, processors can access local memory quickly and remote memory more slowly. This can dramatically improve memory throughput as long as the data is localized to specific processes (and thus processors). However, NUMA makes the cost of moving data from one processor to another significantly more expensive. Its benefits are therefore limited to particular workloads—most notably on servers where the data is often associated with certain task groups. NUMA is less useful for accelerating parallel processes (unless memory access is optimized). Given this, multicore/NUMA programmers should realize that software techniques for improving memory access are even more important in this environment than in single-core processing.

CPUs and GPUs: Bound to Collaborate

Multicores aren't the latest challenge to memory access. In the last few years, Nvidia suddenly realized that its graphics cards (also called graphics processing units, or GPUs) were in effect small parallel supercomputers. The company therefore took the opportunity to improve the GPUs' existing shaders—used primarily to calculate rendering effects—and convert them into "stream" or thread processors. They also created the Compute Unified Device Architecture (CUDA),[5] a parallel programming model for GPUs that has proven to be so popular that a new standard, OpenCL (www.khronos.org/opencl), quickly emerged to help any heterogeneous mix of CPUs and GPUs in a system work together to solve problems faster.

You can seamlessly integrate the current generation of GPUs to run tens of thousands of threads simultaneously. Regarding the starving cores problem, you might wonder whether GPUs have any practical utility whatsoever given the memory bottleneck or whether they're mostly marketing hype.

Fortunately, GPUs are radically different beasts than CPUs and traditional motherboards. One of the critical differences is that GPUs access memory over much better bandwidth (up to 10 times faster in some cases). This is because a GPU is designed to access memory in parallel over independent paths. One problem with this design, however, is that to take advantage of the tremendous bandwidth, all the available memory has to run at very high frequencies—effectively limiting the total amount of high-speed memory (up to 1 Gbyte on most current cards). In contrast, current commodity motherboards are designed to address a much greater amount of memory (up to 16 Gbytes or more).

You can compensate for the fact that GPUs access less memory (albeit at much faster speeds) by using new programming paradigms—such as OpenCL—that effectively combine GPUs with CPUs (which access much more memory at lower speeds). Of course, many problems remain to be solved before this combination is truly effective, from implementing specific hardware to enhance the CPU-to-GPU communication latency and bandwidth to developing new software that allows programmers to take advantage of this new computational model. Only time will tell whether this approach becomes mainstream, but both hardware and software developers are investing considerable effort in exploring it (see www.nvidia.com/object/cuda_home.html#).

The gap between CPU and memory speeds is enormous and will continue to widen for the foreseeable future. And, over time, an increasing number of applications will be limited by memory access. The good news is that chip manufacturers and software developers are creating novel solutions to CPU starvation.

But vendors can't do this work alone. If computational scientists want to squeeze the most performance from their systems, they'll need more than just better hardware and more powerful compilers or profilers—they'll need a new way to look at their machines. In the new world order, data arrangement, not the code itself, will be central to program design.

Acknowledgments

I'm grateful to my friend, Scott Prater, for turning my writing into something that a reader might actually want to look at.

References

1. K. Dowd, "Memory Reference Optimizations," *High Performance Computing*, O'Reilly & Associates, 1993, pp. 213–233.
2. C. Dunphy, "RAM-o-Rama—Your Field Guide to Memory Types," *Maximum PC*, Mar. 2000, pp. 67–69.
3. R. van der Pas, *Memory Hierarchy in Cache-Based Systems*, tech. report, Sun Microsystems, Nov. 2002; www.sun.com/blueprints/1102/817-0742.pdf.
4. U. Drepper, *What Every Programmer Should Know About Memory*, tech. report, RedHat Inc., Nov. 2007; http://people.redhat.com/drepper/cpumemory.pdf.
5. T. Halfhill, "Parallel Processing with CUDA: Nvidia's High-Performance Computing Platform Uses Massive Multithreading," *Microprocessor Report*, vol. 22, no. 1, 2008; www.mdronline.com/mpr/h/2008/0128/220401.html.

Francesc Alted is a freelance developer and consultant in the high-performance databases field. He is also creator of the PyTables project (www.pytables.org), where he puts into practice lessons learned during years of fighting with computers. Contact him at faltet@pytables.org.

Top articles, podcasts, and more.
cn
computingnow.computer.org