from *SIAM News*, Volume 33, Number 5

Portable Memory Hierarchy Techniques For PDE Solvers: Part I

By Craig C. Douglas, Gundolf Haase, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, and Christian Weiss

I am taking the unusual step of writing an editorial foreword for this article. The main reason is that the article, the first of two parts, is highly pedagogical and therefore atypical for this column. The content of this first part is required for a full understanding, and appreciation, of the more application-focused second part, which will appear in the next issue of SIAM News. *Additionally, I feel that many readers will benefit from this exposition, especially those who are not familiar with the impact that caches can have on application performance.—GA*

APPLICATIONS ON ADVANCED ARCHITECTURE COMPUTERS

Greg Astfalk, Editor

In recent years, computer processing power has vastly outstripped advances in the ability of memory chips to read or write data. Superior use of memory hierarchies, in which each level's memory is referred to as a cache, is a remedy that leads to dramatically faster codes.

Cache-aware solvers for (coupled) partial differential equations (PDEs) are now absolutely essential to achieving anything other than poor performance when measured against possible peak performance figures conjured up by vendors. We describe a set of techniques for achieving a respectable percentage of peak performance. Some of these techniques are utterly invasive and should be left to software library writers. Others can be added to an existing code in a matter of minutes.

Not long ago, cache experts believed that portable, cache-aware algorithms were a myth. The folklore was that only by using every last detail of the CPU and its memory hierarchy was it possible to design a fast, cache-aware algorithm. Today, CPUs are even faster; memory subsystems still run at about the same speeds, albeit with somewhat larger caches. Users who maintain a collection of portable algorithms in their code portfolios have suddenly discovered they are cache-wealthy on a wide variety of platforms, from laptops and PCs to supercomputers.

This two-part article is an overview of the efforts of two cooperative research groups to solve partial differential equation problems using (primarily) iterative, multilevel, cache-aware algorithms. The cache-aware algorithms discussed here provide results that are bitwise identical to those of standard implementations. This means that two implementations, one cache-aware and the other in standard form, can be compared. If the norm of the difference of the two computed solutions is not identically zero, then there is a bug in the cache-aware implementation. In addition, the convergence rates of our cache-aware iterative methods are identical to those obtained with the standard methods (see [4, 7].

We begin with a tutorial on how computers work. In Part II, to appear in the next issue, we continue with some simple, constant-coefficient, matrix-free problems and methods and end up considering variable-coefficient, coupled PDEs on unstructured grids.

Hyperlinks to our individual cache-aware algorithm research projects are provided through MGNet [3] at http://www.mgnet.org/.

Processors and Memory Subsystems

When a program is executed, the computer's central processing unit performs the numerical and logical calculations. The data for the CPU is stored in main memory. When the CPU requires data that it does not already have, it makes a request to memory.

Computers have been getting faster and faster over the years, while the memory chips that are used to store data have become only marginally faster. Sadly, current CPUs can perform numerical operations much faster than memory can deliver data, which leads to fast CPUs being idle most of the time. This situation is responsible for the widespread skepticism that greets vendors' claims of peak CPU speeds.

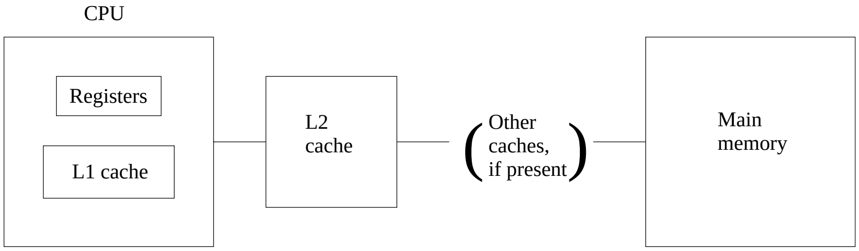
Many hardware and software strategies have been devised to reduce the time a CPU spends waiting for data. The most common hardware strategy (in recent years) is to divide computer memory into a hierarchy of layers, with the CPU linked directly to the highest level (see Figure 1(a)). These layers are referred to as the cache or the memory subsystem [5, 6].

Cache is fast, expensive memory that replicates a subset of data from main memory. The purpose of cache is to provide data to the CPU faster than main memory can. The goal is to reduce the idle time for the CPU.

Caches are motivated by two principles of locality: in time and in space. The principle of temporal locality states that data required now by the CPU will also be necessary again in the near future. The principle of spatial locality states that if specific data is required, its neighboring data will also be referenced soon.

The cache is itself a hierarchy of levels, called L1, L2, . . . , each with a different size and speed. A typical cache has one or two levels. The L1 cache is smaller and 2–6 times faster than the L2 cache. The L2 cache, in turn, is smaller but 10–40 times faster than main memory (see Figure 1(b)).

The data held in any level of cache is a subset of the data held in the next larger level on all processors that we consider in this article, although there are some exceptions. The smallest block of data moved in and out of a cache is a cache line. A cache line



Level	Clocks
register	1
L1 cache	2–3
L2 cache	6–12
near main memory	6–100
far main memory	100–250
distributed main memory	$O(100)$
message-passing	$O(1000) - O(10000)$

Figure 1. *Memory hierarchy (a, left) and speeds (b, right) [1].*

holds data that is stored contiguously in main memory. A typical cache line is 32, 128, or 256 bytes in length, with 32 being most common.

If the data requested by the CPU is in a cache, it is called a cache hit. Otherwise, it is a cache miss and data must be copied from a lower level of (i.e., slower) memory into a cache. The hit rate is the ratio of cache hits to total memory requests. The miss rate is one minus the hit rate.

Given that cache is faster than main memory in fulfilling the CPU's memory requests, we clearly want to maximize the number of cache hits. In designing algorithms to maximize cache hits, we must first decide which cache to optimize for. Many CPUs have a small L1 cache (8–96 Kbyte), which is not sufficient for 3D simulations involving coupled PDEs. A larger L2 cache, found on most computers, is the cache for which we tailor our algorithms. An exception is the HP PA series of CPUs, which now have a large on-CPU L1 cache (e.g., 1.5 MB on the PA-8500), but no L2 cache.

Caches on any level can be unified or split. Unified caches mix computer instructions and data. A split cache consists of two separate caches, with instructions in one and data in the other. Split caches, because of their much better hit ratios for instructions, are superior to unified caches. The possibility of cache thrashing (defined below) is also reduced with split caches.

A final, quite subtle issue needs to be considered, as it leads to nondeter-ministic cache behavior. Programs are loaded into memory using a virtual addressing scheme. A programmer thinks of memory as a linear space, with addressing beginning at zero. In reality, the program is loaded into a set of physical memory pages that are neither contiguous nor linear. A hardware-based scheme is used to translate virtual to physical memory addresses. Almost all caches use the physical address, not the virtual address, to determine where in a cache a piece of memory should be copied. Because a program run more than once will usually be loaded into different physical memory pages, the cache effects are rarely reproducible.

Memory to Cache Mappings

In order to service CPU requests efficiently, the memory subsystem must be able to determine whether requested data is present in a cache and where in the cache that data resides.

The oldest method is the direct-mapped cache. The location in cache is determined using the remainder of the (physical) main memory address divided by the cache size:

$$\text{cache address} = (\text{main memory address})\ \mathit{mod}\ (\text{size of cache})$$

Several addresses in the cache are located in the same cache line.

Hence, a given main memory address can occupy only one location in cache. Traditionally, the number of lines, N, in a cache is always a power of two, so that the remainder operation is equivalent to taking the last $\log_2(N)$ bits of the main memory address. The set bits are the last bits of the main memory address corresponding to the cache address. The tag bits are the remaining bits of the main memory address. The set bits of a request determine where in cache to look, and the tag bits determine whether a match occurs.

Another hardware technique is the n-way set-associative cache, which maps memory to n "ways," or segments, inside a cache. A given memory address can be placed in any of the n segments of the cache. (A direct-mapped cache can be considered a one-way set-associative cache.) A policy for choosing the "way" in which a piece of data should be stored is required for set-associative caches. The three algorithms commonly used are: least recently used (LRU), least frequently used (LFU), and random replacement. These are denoted replacement policies because a cache line placed in cache replaces a cache line that is already there.

For memory access on a memory chip, an address is passed through a small collection of AND, OR, and XOR gates. A cache can be considerably more complex; it must have logic to copy or replace a collection of memory locations from main memory, all of which takes time.

LRU caches track how long ago cache lines were referenced. LFU caches track how often cache lines have been referenced. For many PDE algorithms, LRU caches generally provide the best performance. However, both of these algorithms are expensive to implement in hardware.

The random replacement policy is very popular with computer architects: It is relatively inexpensive, and studies show it to be almost equivalent to LRU (see [5] and its references). The problems studied obviously did not involve the solution of PDEs.

In a fully associative cache, any data can occupy any location in cache. This is an n-way set-associative cache, where n is the number of lines in cache, i.e., each way is exactly one line of cache. The hardware required for this type of cache is prohibitively expensive in comparison with other cache types discussed here [6].

One drawback of a direct-mapped cache is the possibility that several commonly used data items will map to the same cache address. Consider the following code fragment:

```
for $i = 1, n$ do
    $C(i) = A(i) + B(i)$
end for
```

Suppose that *A*, *B*, and *C* each maps to the same cache location. Each reference to *A*(*i*), *B*(*i*), or *C*(*i*) will then cause the same cache line to be used. The cache line is constantly emptied and refilled, causing cache thrashing. Hence, in terms of the memory latency time, *L*, the cost of running the code fragment is at least *nL*.

Two simple fixes can be applied to this example. First, the vectors can be padded by a small multiple of the cache line length (common paddings are 128 or 256 bytes, independent of whether or not they are good values). This is hardly a portable solution, however, since the correct size of a padding is very machine dependent.

The second fix is to change the data structure, combining *A*, *B*, and *C* into a single array. Then the code fragment becomes:

```
for $i = 1, n$ do
    $r(3, i) = r(1, i) + r(2, i)$
end for
```

In this case, we assume that the vector elements are in adjacent memory locations. This fix reduces the number of cache misses by approximately three on any of the cache designs discussed, and it is highly portable.

Memory can be arranged into memory banks, with each bank containing many memory chips and an addressing mechanism. All banks can be accessed simultaneously, providing parallel access to memory. Many computers, from the 1960s on, have had this feature, which allows for memory to be accessed in a "vector style."

Suppose, for example, that the CPU requests four words of data that are stored in one bank of memory. The penalty just for retrieving the data from memory is 4*L*, where *L* is the response time (or latency) of the memory. This latency occurs because the words are retrieved one at a time. Now suppose the memory is divided into four banks, with one word is in each bank. The delay due to memory latency is now *L* because the banks retrieve the data simultaneously.

Many algorithms using vectors of data operate on sequential vector elements. These algorithms, known as stride-1 algorithms, work well with memory banks. However, some algorithms work with every *m*th vector element (stride-*m* algorithms). If there are *m* memory banks, the stride-*m* algorithms once again produce slow memory access, with *NL* latency, where *N* is the number of vector accesses, since all accessed vector elements are in the same memory bank.

Common Strategies for Enhancing Cache Performance

Several strategies for enhancing cache performance are in wide use. All try to avoid CPU stalls, defined as periods dur-ing which the CPU is idle because necessary data is not immediately available.

Prefetching—bringing data into cache before the CPU actually needs it—is a method for reducing the effect of mem-ory latency. Having cache lines longer than one word is one example of pre-fetching. Data brought into cache is accom-panied by surrounding data; if the data ex-hibits spatial locality, then the surrounding data has effectively been prefetched.

Prefetching satisfies CPU requests at the speed of cache rather than of slower memory. Prefetching can be accomplished with compiler flags, via programmer intervention, or by hardware.

Prefetching is particularly effective within loops. Unfortunately, it is highly machine dependent and, therefore, not portable. Consider, for example, the following code fragment:

```
(intervening code)
for $i = 1, n$ do
    $A(i) = A(i) + \beta * B(i)$
    (more intervening code)
    $D(i) = D(i) + \gamma * C(i)$
    (yet more intervening code)
end for
```

Assume that the intervening code depends on data already in cache. On iteration *i*, it may be possible to bring *A*(*i* + 1), *B*(*i* + 1), *C*(*i* + 1), and *D*(*i* + 1) into cache before the CPU needs them by means of explicit prefetch instructions:

```
PREFETCH $A(1)$, $B(1)$, and $\beta$
(intervening code)
for $i = 1, n$ do
    $A(i) = A(i) + \beta * B(i)$
    PREFETCH $C(i)$ and $D(i)$
    (more intervening code)
   $C(i) = C(i) + \gamma * D(i)$
```

```
    PREFETCH $A(i + 1)$ and $B(i + 1)$
    (yet more intervening code)
end for
```

Prefetching is often used with software pipelining, which breaks complex calculations into simpler steps, analogous to an assembly line. The result of a simple calculation is input for the next simple calculation. Although this scheme does not affect the speed of a single calculation, it can speed up the same calculation repeated many times.

Pipelining can be implemented either by hardware (as in many RISC processors) or by software. In the latter case, loops are un-rolled and complex calculations are broken into very simple steps. Because calculations and the fetching of data from main memory are done at the same time, pipe-lining also helps hide memory latency.

Consider the following code fragment illustrating a scalar multiply and add (SAXPY):

```
for $i = 1, n$ do
    $A(i) = A(i) + \gamma * B(i)$
end for
```

Provided that the CPU supports pipe-lining (i.e., has several independent arithmetic units), a good compiler can generate pipelined code. In the execution of such code, one arithmetic unit performs the scalar multiply while the other unit performs an addition. Once the pipeline has been filled, a SAXPY operation is completed at each cycle. Hence, the time to complete the operation is the time to load the pipeline (arithmetic units) plus n, the length of vector A. The programmer can provide a hint to the compiler by breaking the SAXPY into two simpler operations:

```
for $i = 1, n$ do
    $t = \gamma * B(i)$
    $A(i) = A(i) + t$
end for
```

Some CPUs can perform out-of-order execution. The processor tracks the data dependencies of the instruction queue and marks which instructions are ready to run. If an instruction is blocked but another is ready to run, the latter is executed immediately. This is particularly useful in short loops on vectors.

Padding, mentioned earlier as a way to avoid cache thrashing, involves the lengthening of an array by the addition of a small number of extra elements, usually the amount in one cache line. In this way, arrays that originally thrashed the cache are forced to map to different cache addresses. Stride-1 access to arrays enhances spatial locality—bringing one array element into cache also brings in neighboring elements in the same cache line.

Loops are blocked by a reordering of data access in a way that minimizes the number of accesses. The reordering depends on both the data and the cache sizes. A beneficial result is that data in cache can be reused many times. For example, matrix–matrix multiplication is blocked to enhance performance in vendor-supplied libraries. The ATLAS project [2] is a numerical linear algebra package that relies extensively on blocking to tune the Basic Linear Algebra Subroutines (BLAS) automatically.

Epilogue

In the second part of this article, which will appear in the next issue of *SIAM News*, we will cover techniques that will exploit cache for both logically tensor product grid-based problems as well as the more challenging variable-coefficient, unstructured grid problems.

Acknowledgments

The work presented here—a truly international effort—was supported in part by the DFG Ru 422/7–1,2, NATO grant CRG 971574, and NSF grants DMS–9707040, ACR–9721388, and CCR–9902022.

References

[1] G. Astfalk, *Relative memory speeds*, private communication, 1999.

[2] J. Dongarra and R.C. Whaley, *Automatically tuned linear algebra software*, http://www.netlib.org/atlas, 1999.

[3] C.C. Douglas, *MGNet: A multigrid and domain decomposition network*, ACM SIGNUM Newsletter, 27 (1992), 2–8.

[4] C.C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, *Cache optimization for structured and unstructured grid multigrid*, ETNA, 10 (2000), 21–40.

[5] J. Handy, *The Cache Memory Book*, Academic Press, New York, 1998.

[6] D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, California, 1996.

[7] C. Weiss, W. Karl, M. Kowarschik, and U. Rüde, *Memory characteristics of iterative methods*, Proceedings of the Supercomputing Conference, Portland, Oregon, 1999.

Craig C. Douglas (douglas@ccs.uky.edu) is a professor at the University of Kentucky and is affiliated with Yale University. Gundolf Haase (ghaase@numa.uni-linz.ac.at) is an assistant professor at Johannes Kepler Universität Linz and a visiting professor at the University of

Kentucky. Jonathan Hu (jhu@ccs.uky.edu) is a graduate student at the University of Kentucky (and is moving to Sandia National Laboratories shortly). Markus Kowarschik (kowarschik@cs.fau.de) and Ulrich Rüde (ruede@cs.fau.de) are a research assistant and a professor, respectively, at the Universität Erlangen–Nürnberg. Christian Weiss (weissc@in.tum.de) is a research assistant at the Technische Universität München.