Portable memory hierarchy techniques for PDE solvers

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

In recent years, computer processors have vastly outstripped advances in memory chips' abilities to read or write data. Superior use of memory hierarchies, where each level's memory is referred to as a cache, leads to dramatically faster codes.

Cache aware solvers for (coupled) partial differential equations 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 to achieve 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. Folklore said that only by using every last detail of the CPU and its memory hierarchy could a fast, cache aware algorithm be designed. Times have changed: CPU's are even faster and memory subsystems still run at about the same speed, albeit caches are somewhat larger now.

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 PC's to supercomputers.

This article gives an overview of how two cooperative research groups solve partial differential equation problems using (primarily) iterative, multilevel, cache aware algorithms. The cache aware algorithms discussed here provide bitwise identical results to standard implementations. This means that two implementations, one cache aware and one using 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 standard ones (see [4] and [7]).

We begin with a tutorial on how computers work. We continue with some simple, constant coefficient, matrix free problems and methods. Finally, we end up with variable coefficient, coupled PDE's on unstructured grids. While we concentrate on two-dimensional problems in this article, we note that we have three-dimensional solvers and are in the process of creating parallel versions.

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

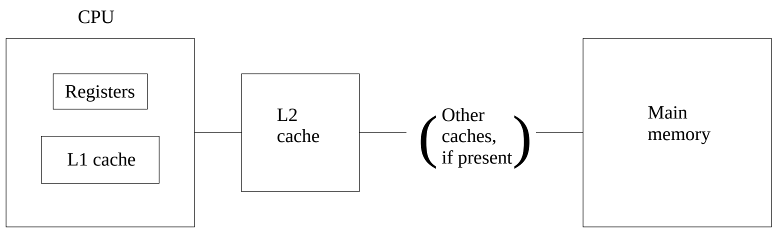
Processors and memory subsystems

A computer's central processing unit (CPU) performs the numerical and logical calculations when a program is executed. 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 CPU's can perform numerical operations much faster than memory can deliver data, which leads to fast CPU's being idle most of the time. This situation is why vendors' claims of peak CPU speeds are held in the same high regard as statements made by politicians or sellers of used cars or insurance policies.

Many hardware and software strategies exist to reduce the time that a CPU is waiting for data. The most common hardware strategy (in recent years) is to divide computer memory into a hierarchy of different 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



(a) Memory hierarchy.

Level	clocks
register	1
L1 cache	2–3
L2 cache	6–12
near main memory	60–100
far main memory	100–250
distributed main memory	$\mathcal{O}(100)$
message-passing	$\mathcal{O}(1000)$–$\mathcal{O}(10000)$

(b) Clock speeds.

Figure 1: Memory hierarchy and speeds [1].

than main memory can. The goal is to have the CPU less idle when cached data is used.

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

The cache is itself a hierarchy of levels, called L1, L2, ..., each with a different size and speed. One or two levels of cache is typical. The L1 cache is smaller and 2–6 times faster than the L2 cache. The L2 cache in turn is smaller than main memory but is 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, though there are some exceptions. The smallest block of data moved in and out of a cache is a cache line. A cache line holds data which 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 that the CPU requests 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 (i.e., a slower) of 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.

We clearly want to maximize the number of cache hits since cache is faster than main memory in fulfilling the CPU's memory requests. In designing algorithms to maximize cache hits, we must first decide which cache to optimize for. Many CPU's have a small L1 cache of only 8–96 KB, which is usually too small for 3D simulations involving coupled partial differential equations. Most computers have a larger L2 cache, which we tailor our algorithms for. An exception is the HP PA series of CPU's, 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 may be unified or split. Unified caches mix both computer instructions and data. Split caches are two separate caches with instructions in one and data in the other. Split caches are superior to unified ones since the hit ratio tends to be much better for instructions than in unified caches. Additionally, there is less possibility for cache thrashing, a concept we will define shortly.

There is a final issue with caches that is quite subtle and leads to caches exhibiting nondeterministic 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 translation scheme is used from virtual memory addresses to physical memory addresses using hardware. Almost all caches use the physical address, not the virtual address, to determine where in a cache a piece of memory should be copied. Since running a program twice usually means that it is 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:

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

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

Hence, there is only one location in cache that a given main memory address can occupy. Traditionally, the number of lines, N, in 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 cache technique is n-way set associativity, which maps memory to n "ways," or segments, inside a cache. A given memory address could be placed in any of the n segments of the cache. (A direct mapped cache corresponds to a 1-way set associative cache.) A policy for deciding in which "way" a piece of data should be stored is required for set associative caches. There are three common algorithms for choosing in which "way" a cache line should be placed: least recently used (LRU), least frequently used (LFU), and random replacement. These are called replacement policies since when a cache line is placed into cache it replaces a cache line that is already there.

Accessing memory on a memory chip is determined by passing an address through a small collection of AND, OR, and XOR gates. A cache can be considerably more complex since 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 tend to 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 that it is almost equivalent to LRU (see [5] and its references). One can only wonder what problems were studied since they obviously did not involve solving PDE's.

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 to the other types of caches already discussed [6].

One drawback of a direct mapped cache is the possibility that several commonly used data items will map onto 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 map to the same cache locations. Then each reference to $A(i)$, $B(i)$, or $C(i)$ causes 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.

There are two simple fixes 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 portable, however, since the correct size of a padding is very machine dependent. The second fix is to change the data structure to combine 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 reduces the number of cache misses by approximately a factor of three on any of the cache designs discussed, and it is highly portable.

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

For example, suppose 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 $4L$, 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 4 banks, and 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 are known as stride-1 algorithms. They 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 to enhance cache performance

There are several widely used strategies for enhancing cache performance. All strategies try to avoid CPU stalls. Stalling is defined as periods when the CPU is idle because necessary data is not immediately available.

Prefetching brings data into cache before the CPU actually needs it. Thus, prefetching is a method for reducing the effect of memory latency. Cache lines longer than one word are one example of prefetching. When data is brought into cache, surrounding data accompanies it. If the data exhibits spatial locality, then the surrounding data has effectively been prefetched.

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

Prefetching is particularly effective within loops. It is, unfortunately, highly machine dependent and therefore not portable. For example, consider the following code fragment:

```
(intervening code)
for i = 1, n do
    A(i) = A(i) + β * B(i)
    (more intervening code)
    D(i) = D(i) + γ * 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 β
(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. Pipelining breaks complex calculations into simpler steps. A common analogy is that of an assembly line. The result of a simple calculation is input for the next simple calculation. Although this does not speed up one 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 unrolled and complex calculations are broken into very simple steps. Pipelining also helps hide memory latency since calculations occur at the same time as data is fetched from main memory.

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 pipelining (i.e., it has several independent arithmetic units), a good compiler can generate pipelined code. In executing 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 CPU's can perform out-of-order execution. The processor tracks 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, then the latter is executed immediately. This is particularly useful in short loops on vectors.

Padding has already been mentioned as one method for avoiding cache thrashing. An array is lengthened by adding a small number of extra elements, usually the amount in one cache line. This forces arrays that originally thrashed the cache to map to different cache addresses. Stride-1 access to arrays enhances spatial locality, since bringing one array element into cache also brings in neighboring array elements that are on the same cache line.

Loops are blocked by reordering data access so that the number of accesses is minimized. The reordering depends on both the data and cache sizes. A beneficial result is reusing data in cache 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 which relies extensively on blocking to tune the Basic Linear Algebra Subroutines (BLAS) automatically.

Epilogue

In the second part of this article, that will appear in the next issue of *SIAM News*, we will cover the 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 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; A truly international effort.

References

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

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

[3] C.C. Douglas, *MGNet: A multigrid and domain decomposition network*, ACM SIGNUM Newsletter, 27(1992), pp: 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), pp: 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, CA, 1996.

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

Craig C. Douglas `douglas@ccs.uky.edu` is a Professor at the University of Kentucky and at 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 Laboratory 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.