Editors: Michael Dennin, mdennin@uci.edu
Steven Barrett, steveb@uwyo.edu

Why Computer Architecture Matters: Memory Access

By Cosmin Pancratov, Jacob M. Kurzer, Kelly A. Shaw, and Matthew L. Trawick

This three-part series shows how applying knowledge about the underlying computer hardware to the code for a simple but computationally intensive algorithm can significantly improve performance. This second segment focuses on memory accesses.

When scientists write code to implement an algorithm, we initially focus on just getting something that works. However, as we execute our code on large problem sizes, we sometimes discover that our initial implementation runs too slowly. Fortunately, there are some easy ways to speed up the code by tailoring it to how computer hardware actually works.

In the first installment of this three-part series (May/June 2008), we examined the individual instructions required by a simple but computationally intensive algorithm (an orientational correlation function) and explored how to reduce the number of clock cycles needed for each loop iteration. By avoiding long-running instructions and precalculating some values outside of the innermost loop, we successfully cut the clock cycles needed for each iteration by more than half.

But our code still executed much more slowly than we anticipated. Although we estimated that each loop iteration should require roughly 50 clock cycles on our test system, we found that each actually required more than 130 clock cycles! The reason for this large discrepancy turned out to be the delay time, or *latency*, associated with retrieving data from the computer's main memory.

In this installment of our series, we show how to rearrange code so that it accommodates the peculiarities of the memory system and consequently reduces execution time.

Main Memory and Caches

In part 1 of this series, we saw that latencies for arithmetic instructions range from one to 10 cycles (with some select instructions taking longer). In contrast, a typical main memory access can take approximately 300 cycles. All the optimizations we made to decrease the number of cycles doing computation are irrelevant if most memory accesses incur these long latencies.

Fortunately, hardware designers have developed a strategy to mask this problem. They place a small amount of faster memory, called a "cache," on or very near the CPU chip itself, where it can be accessed much more quickly than the main memory. Modern processors typically have two or more levels of cache (denoted L1, L2, and so on), each with a different trade-off of size versus latency.[1] Figure 1 depicts the memory-system configuration used in our tests. Reading from the L1 cache (the smallest cache, located closest to the processor) is very fast, whereas reading from the main memory (which is large and farthest from the processor) is much slower.

To reduce the negative impact of memory accesses, the fast L1 cache must be able to satisfy as many memory accesses as possible. We want to avoid going to the main memory at all costs. As programmers, we need to understand when the cache retains data—that way, we ensure that the data we want will be cached when we need it.

Locality, Locality, Locality!

Caches work by retaining recently used data so that it can be retrieved more quickly on future accesses. Data is retained in the cache for as long as possible and is only evicted when other, more recently accessed data must be retained. One way to reduce the frequency of having to go to the main memory is to reuse data as many times as possible within a short period of time, because the data will likely remain cached. This is known as *temporal locality*.

Caches also exploit *spatial locality*, which occurs when a program uses multiple pieces of contiguously stored data in a short time period—for instance, by accessing array elements sequentially. Caches exploit this spatial locality by retrieving and storing not just the specific bytes of data requested by the processor but an entire "line" or sequence of contiguous bytes of data. By doing so, this cached line of data can satisfy future requests for nearby memory addresses. Additionally, memory controllers usually "prefetch" multiple lines of data from the main memory into the cache when they suspect that they might

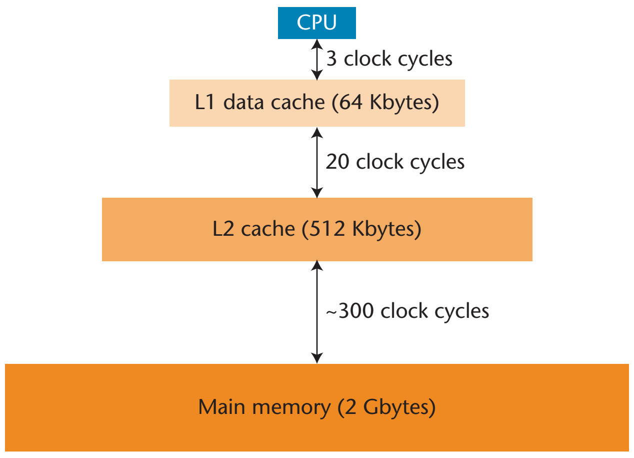

Figure 1. The relative sizes and latencies of different parts of our test platform's memory system.

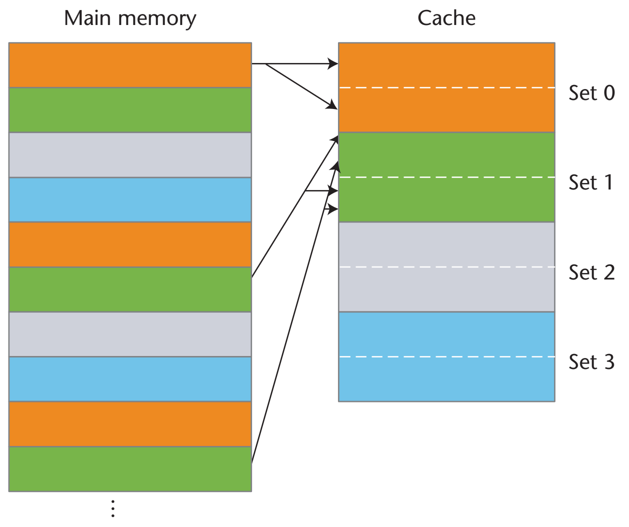

Figure 2. Mapping between main memory (divided into cache lines) and a two-way set associative cache with a capacity of eight cache lines.

be used in the near future—for example, if nearby memory locations have been recently accessed. So, even if data spans several cache lines, storing it contiguously improves the chance that it will be prefetched before it's needed.

There's another way in which a program's spatial locality can improve a cache's performance, based on a subtle aspect of how caches work. Figure 2 shows an example cache with eight cache lines. On the left, we also show an example memory space divided into cache lines. In an ideal world, any line of data could reside in any cache line. However, to limit the number of lines that must be searched and keep the cache access times short, caches generally limit the number of cache lines in which a given piece of data can reside. In our example, every piece of data maps to two different cache lines, collectively referred to as a *set*. (We refer to this configuration as *two-way set associative* because each set contains two cache lines.) In the figure, green areas in the main memory can map to only one of the two green cache lines in the cache. If more than two areas map to the same set, a conflict results, causing some data to be evicted. Spatial locality can avoid such conflicts, because contiguous areas in the main memory are guaranteed to map to different sets in the cache.

A Programming Example

Let's look back at our orientational correlation code from part 1 of this series to evaluate its spatial and temporal locality. The data to be analyzed is a series of N points, possibly from a microscope image, where each point has a location (with x and y values), as well as a local orientation. The data is stored in four separate input arrays: `x[N]`, `y[N]`, `sin6[N]`, and `cos6[N]`. (Previously, we found it computationally advantageous to precalculate the sine and cosine for each given orientation angle—hence, the two arrays.) Calculating the orientational correlation function requires calculating the distance between each possible pair of points i, j,

$$r_{i,j} = \sqrt{\left(x_i - x_j\right)^2 + \left(y_i - y_j\right)^2},$$

and accumulating correlation data for each pair in the `r`th element of the arrays `g[r]` and `count[r]`. The relevant portion of the code is as follows:

```
//Now, accumulate data for all pairs of
//points (i,j).
for(i=0; i<N; ++i)        //for each i < N
  for(j=i+1; j<N; ++j) {  //for each j < N
    Dx = x[i]-x[j];
    Dy = y[i]-y[j];
    r = sqrt(Dx*Dx + Dy*Dy);
    g[r] += cos6[i]*cos6[j]+sin6[i]*sin6[j]
    ++count[r];
  }
```

Even without thinking too hard about what the code is actually doing, it's clear that it already exhibits some degree of temporal and spatial locality, which the cache is exploiting. The temporal locality comes from repeatedly reusing the `i`th element of the input arrays, as guaranteed by the nested `for` loops. The spatial locality comes from

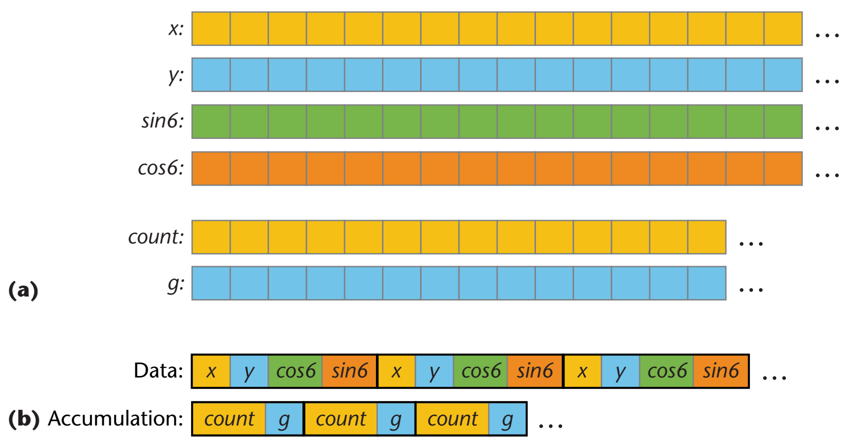
x:
y:
sin6:
cos6:
count:
g:
(a)
Data: x y cos6 sin6 x y cos6 sin6 x y cos6 sin6
(b) Accumulation: count g count g count g

Figure 3. Array merging for increased spatial locality. (a) The data structure as originally coded in six separate arrays and (b) the revised data structures, as two arrays of structures.

always accessing the four input arrays sequentially, with the index `j` incremented by one every iteration. The cache must satisfy at least some of the program's memory requests—if not, the clock cycles per pair of points in our code would be significantly higher than the observed 130 cycles.

But there's clearly room for improvement. Each of the inner loop's iterations currently requires access to six different arrays, which means that six different locations in memory must be accessed. And since the cache won't be able to hold all of the data for large values of *N*, the data will have to be read again from main memory every time the inner loop begins. Our goal as programmers is to find ways to improve spatial and temporal locality where we can, by changing either the order in which we store data or the order in which we access it. Fortunately, some broadly applicable strategies can help.

Improving Spatial Locality: Array Merging

Examining our sample program, we have declared six different large arrays of numbers. The declarations for these arrays are

```
int x[N], y[N];
float sin6[N], cos6[N];
int count[MAX_R];
float g[MAX_R];
```

As Figure 3a shows, these arrays are each stored sequentially in the main memory. Structuring the data into these parallel arrays would be ideal if our processing required accessing only the *x* or *y* values. But our program accesses the `j`th element of each of the four input arrays and the `r`th element of each of the two output arrays at the same time, so this data organization doesn't mimic how we access the data.

In our case, we can improve our program's spatial locality by storing contiguously in memory the values associated with each `j`th input point as shown in Figure 3b, a technique known as *array merging*.[2] In C, this is accomplished by defining a struct that contains all four values for each point, with our data held in a single array `data[N]` of those structs. When the cache retrieves the cache block containing the `x` value for point `j`, it will automatically prefetch the `y`, `cos6`, and `sin6` values for that point and could potentially prefetch values for several other points (such as `j + 1`, `j + 2`, and so on), depending on the cache line size. Similarly, because the `r`th element of the arrays `count` and `g` are always accessed at the same time, we can define a second struct that contains both the `g` and `count` values; all `Max_R` of these values are held in a single array of these structures `accum[Max_R]`. The revised declaration section and the code that uses these structures is as follows:

```
struct DataStruct {
  int x, y;
  float cos6, sin6; };
DataStruct data[N];

struct AccumulationStruct {
  int count;
  float g; };
AccumulationStruct accum[Max_R];

//Now, accumulate data for all pairs of
//points (i,j).
for(i=0; i<N; ++i)        //for each i < N
  for(j=i+1; j<N; ++j) {  //for each j < N
    Dx = data[i].x - data[j].x;
    Dy = data[i].y - data[j].y;
    r = sqrt(Dx*Dx + Dy*Dy);
    accum[r].g += data[i].cos6 * data[j].cos6 +
                  data[i].sin6 * data[j].sin6;
    ++accum[r].count;
  }
```

We tested this code's speed and compared it to the code without array merging; the results appear in the first two

Table 1. Effects of array merging and blocking on execution time.

	Execution time (seconds)	Clock cycles per pair
Without array merging or blocking	4,422	133.6
With array merging only	2,917	88.1
With blocking only	4,086	123.5
With array merging and blocking	2,634	79.6

rows of Table 1. (All table entries reflect the use of the trigonometric optimization discussed in the first installment.) Apparently, this simple code modification cuts the cycles needed per pair by more than one-third, getting our observed execution time closer to our theoretical estimate of about 50 clock cycles per iteration. Clearly, rearranging the data to reflect how data is accessed enhances the cache's ability to exploit spatial locality and streamlines the prefetching of data.

Improving Temporal Locality: Blocking

As mentioned earlier, our code already exhibits some temporal locality by keeping the index constant on the outer loop, resulting in each `i`th array element being repeatedly reused. During each outer loop iteration, however, it also cycles through the data for all possible values of `j`. Consequently, the time between subsequent uses of the same data is the time it takes to process all of the data for one iteration of the outer loop. For large numbers of points, the data for all values of `j` might be far too big to fit in either the L1 or L2 cache, so later values overwrite the first values of `j` in the cache. Consequently, the cache can't exploit this data's eventual reuse.

A solution to this problem is to group the input data together in small blocks (of size `blocksize`) that will easily fit in the lowest level of cache. Then we can process each possible pair of points from those two blocks before moving on to the next pair of blocks, as Figure 4 shows. This technique, called *blocking*, lets us perform more computations for every new data point that we read from the main memory. For example, if we store in the cache two blocks of data, each of `blocksize` = 100 points, we could process all 10,000 possible pairs of points before having to access the next block of 100 points. That's a big improvement from processing only a single pair for every point read. The modified portion of our code snippet is as follows:

```
for(A=0; A+2*blocksize<N; A+=blocksize)
  for(B=A+blocksize; B+blocksize<N;
      B+=blocksize)
    for(i=A; i<A+blocksize; ++i)
      for(j=B; j<B+blocksize; ++j) {
      .
      .
      .
```

The technique's disadvantage, of course, is that it makes the code more complex. First, it increases the number of nested loops from two to four. Second, it introduces several special cases (not handled in the sample code) for pairs of points within the same block and for any leftover points if the number of points N isn't an integer multiple of `blocksize`.

Table 1 shows that using this blocking technique, whether in conjunction with structs or individual arrays, increases the speed by about 10 clock cycles per pair. This modest gain underscores that the hardware was already doing a pretty good job of prefetching the input array data. However, the fact that we're still above the estimated 50 clock cycles required for the calculations in each inner loop iteration indicates that some memory latency is still affecting our performance. Apparently, the L1 cache still isn't satisfying some of the data requests.

The only way to know for sure what causes the additional latency would be to run a detailed simulation of our calculation based on our memory system's known behavior. Fortunately, we can make some educated guesses based on the sizes and latencies of each level of cache on our CPU. It happens that for the set of input data we used to test our code, the required size of the accumulation array (with or without structs) is just over 64 Kbytes, which is just a bit too big for our L1 data cache. Worse, each pair of points from the input data could be any distance apart, so the accumulation array is accessed randomly, with neither spatial nor temporal locality. Given that the L1 cache is too small to hold the entire accumulation array and the two input data blocks at the same time, data is frequently evicted from the L1 cache and must subsequently be reloaded—most likely from the L2 cache, which, at 512 Kbytes, is large enough to hold everything. Each L2 cache access takes 20 clock cycles, which would be just about right to explain the additional latencies we observe.

The array-merging and blocking techniques, which are applicable to a variety of problems, improved our code's performance by approximately 40 percent. Coupled with the improvements we made to the code in the first installment of this series, we have reduced execution time dramatically—by more than a factor of four. But we're not done yet! There are additional ways we can improve our code that also use the general strategies of reducing instructions (both explicitly and implicitly, for address arithmetic) and increasing locality (both spatial and temporal). In the next installment, we will see how paying attention to computer architecture for a particular algorithm can yield even more dramatic performance improvements.

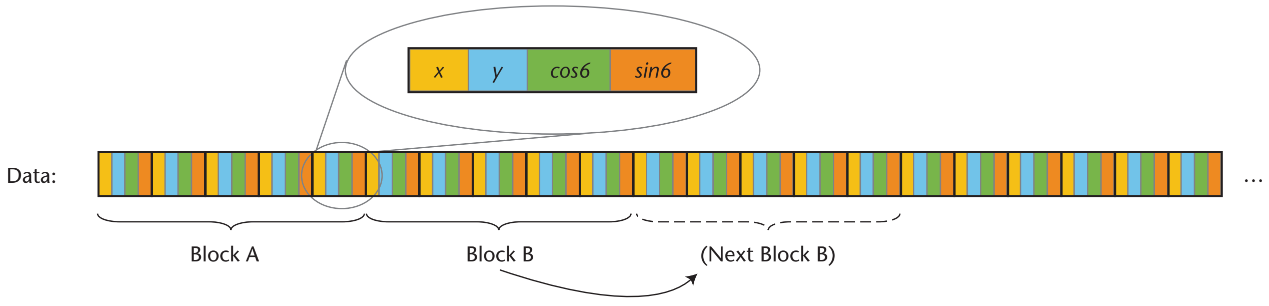

Figure 4. Increasing spatial locality using blocking. This technique lets us perform more computations for every new data point that we read from the main memory.

References

1. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.
2. A.R. Lebeck and D.A. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study," *Computer*, vol. 27, no. 10, 1994, pp. 15–26.

Cosmin Pancratov is a research assistant and undergraduate student at the University of Richmond. His research interests include condensed matter physics and computer science. Contact him at cosmin.pancratov@richmond.edu.

Jacob M. Kurzer is a research assistant and undergraduate student at the University of Richmond. His research interests include algorithms and performance optimization. Contact him at jacob.kurzer@richmond.edu.

Kelly A. Shaw is an assistant professor of computer science at the University of Richmond. Her research interests include the interaction of hardware and software in chip multiprocessors. Shaw has a PhD in computer science from Stanford University. Contact her at kshaw@richmond.edu.

Matthew L. Trawick is an assistant professor of physics at the University of Richmond. His research interests include the physics of block copolymer materials and their applications in nanotechnology, as well as atomic force microscopy. Trawick has a PhD in physics from the Ohio State University. Contact him at mtrawick@richmond.edu.

Advertiser Index
July/August 2008

Advertiser	Page
The Portland Group	9
Princeton University Press	5

Advertising Personnel

Marion Delaney
EEE Media, Advertising Dir.
Phone: +1 415 863 4717
Email: md.ieeemedia@ieee.org
Marian Anderson
Sr. Advertising Coordinator
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: manderson@computer.org
Sandy Brown
Sr. Business Development Mgr.
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: sb.ieeemedia@ieee.org

Advertising Sales Representatives

Mid Atlantic
(product/recruitment)
Dawn Becker
Phone: +1 732 772 0160
Fax: +1 732 772 0164
Email: db.ieeemedia@ieee.org

New England (product)
Jody Estabrook
Phone: +1 978 244 0192
Fax: +1 978 244 0103v
Email: je.ieeemedia@ieee.org

New England (recruitment)
John Restchack
Phone: +1 212 419 7578
Fax: +1 212 419 7589
Email: j.restchack@ieee.org

Connecticut (product)
Stan Greenfield
Phone: +1 203 938 2418
Fax: +1 203 938 3211
Email: greenco@
optonline.net

Midwest (product)
Dave Jones
Phone: +1 708 442 5633
Fax: +1 708 442 7620
Email: dj.ieeemedia@ieee.org
Will Hamilton
Phone: +1 269 381 2156
Fax: +1 269 381 2556
Email: wh.ieeemedia@ieee.org

Joe DiNardo
Phone: +1 440 248 2456
Fax: +1 440 248 2594
Email: jd.ieeemedia@ieee.org

Southeast (recruitment)
Thomas M. Flynn
Phone: +1 770 645 2944
Fax: +1 770 993 4423
Email: flynntom@
mindspring.com

Southeast (product)
Bill Holland
Phone: +1 770 435 6549
Fax: +1 770 435 0243
Email: hollandwfh@yahoo.com

Midwest/Southwest (recruitment)
Darcy Giovingo
Phone: +1 847 498-4520
Fax: +1 847 498-5911
Email: dg.ieeemedia@ieee.org

Southwest (product)
Steve Loerch
Phone: +1 847 498 4520
Fax: +1 847 498 5911
Email: steve@didierandbroderick.com

Northwest (product)
Lori Kehoe
Phone: +1 650 458 3051
Fax: +1 650 458 3052
Email: l.kehoe@ieee.org

Southern CA (product)
Marshall Rubin
Phone: +1 818 888 2407
Fax: +1 818 888 4907
Email: mr.ieeemedia@ieee.org

Northwest/Southern CA
(recruitment)
Tim Matteson
Phone: +1 310 836 4064
Fax: +1 310 836 4067
Email: tm.ieeemedia@ieee.org

Japan
Tim Matteson
Phone: +1 310 836 4064
Fax: +1 310 836 4067
Email: tm.ieeemedia@ieee.org

Europe (product)
Hilary Turnbull
Phone: +44 1875 825700
Fax: +44 1875 825701
Email: impress@impressmedia.com