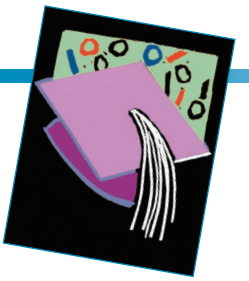
Editors: Michael Dennin, mdennin@uci.edu
Steven Barrett, steveb@uwyo.edu



WHY COMPUTER ARCHITECTURE MATTERS: THINKING THROUGH TRADE-OFFS IN YOUR CODE

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. The final installment focuses on modifying a specific algorithm by applying general principles of efficient programming.

In the previous two installments of this department, we discussed four principles we can use to leverage our understanding of computer architecture to create faster programs:

- *Choose low-latency instructions:* avoid costly instructions such as floating point division and trigonometric functions in favor of fast integer instructions.
- *Precalculate values*: perform computations outside of a loop to avoid repeated calculations.
- *Use temporal locality*: change the order of calculations to re-use data soon after its initial use, when it's likely to still be held in cache.
- *Use spatial locality*: reorganize data structures and change the order of calculations so that data stored together are accessed at nearly the same time.

We showed how to apply these principles using several specific techniques, such as array merging and blocking, that are broadly applicable to many programming tasks. We incrementally applied these individual techniques to improve performance.

Sometimes, however, further performance optimization requires larger-scale changes, which we can't achieve by simply applying a series of independent techniques. In these cases, it pays to think carefully about how a specific algorithm functions to discover new ways to apply these principles to the situation at hand. Resulting improvements might require simultaneous use of a combination of principles or involve a trade-off of benefits and penalties between conflicting principles.

Here we continue to work with our example from the earlier installments of this series, but because the best combination of principles isn't easily generalizable from problem to problem, this time we focus on the actual thought process involved in making decisions about these larger-scale modifications.

Execution Speed: Finding the Bottleneck

Let's look back at our orientational correlation program from previous installments. The data to be analyzed is a series of N points, possibly from a microscope image, stored in an array `data[N]` of structures. Each point has a location, stored in `data[].x` and `data[].y`, as well as a local orientation, stored in `data[].cos6` and `data[].sin6`. (Previously, we found it computationally advantageous to precalculate the sine and cosine for each given orientation—hence, the two separate fields.) Calculating the orientational correlation function $g(r)$ for the data 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 information for all pairs of points as a function of that distance. Eventually, we calculate an average correlation value g for all pairs of points separated by each particular distance r. The heart of our implementation of the algorithm, which follows, requires calculating the distance `r` between each pair of points, and using it as the index of an array `accum` to which we add the results of the computation for that pair:

```
//The first four lines choose each possible
//pair of points i,j in an efficient order to
//maximize locality.
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) {
        //For each ij pair, calculate r
        Dx = data[i].x - data[j].x;
        Dy = data[i].y - data[j].y;
        r = sqrt(Dx*Dx + Dy*Dy);
```

 Copublished by the IEEE CS and the AIP 1521-9615/08/$25.00 © 2008 IEEE

Table 1. Three strategies for accumulating results.

Accumulation array	Size	Index calculation	Operations
accum[r]	MAX_r	r = sqrt(Dx*Dx + Dy*Dy)	SQRT,*,*,+
accum[r2]	MAX_r*MAX_r	r2 = Dx*Dx + Dy*Dy	*,*,+
accum2d[y_ind][x_ind]	MAX_y*MAX_x	y_ind = ABS(Dy) x_ind = ABS(Dx)	ABS,ABS,*,+

```
        //calculate cos(6 *(t_i - t_j))
        //using precalculated sines and
        //cosines. Add the result to accum[r].
        accum[r].g += data[i].cos6 *
                      data[j].cos6 +
                      data[i].sin6 *
                      data[j].sin6;
        ++accum[r].count;
      }

//Finally, divide through by # of pairs at
//each r to get average.
for(r=0;r<MAX_r;++r)
  accum[r].g = accum[r].g / accum[r].count;
```

When we measured the execution speed of this code, which we optimized by applying a set of independent techniques, we found that while faster than our earliest attempts, each iteration of the innermost loop still took a disappointing 79.6 clock cycles to complete. How can we make this faster?

Our previous estimates for this code generously suggested each iteration should take 50 or fewer clock cycles, so something here is taking longer than it should. Examining the inner loop closely, one place that might be responsible for the delay is the access of the array `accum`, which along with the input data is probably too large to fit entirely within the lowest level of our CPU's cache. Because we index `accum` with a random element `r` (which might be either big or small), there is virtually no spatial locality; each new element probably has to be loaded from a level of cache far from the processor, incurring a significant latency.

But a second place that might be just as significant a source of delay is the calculation of the square root. The square root is a high-latency instruction, requiring roughly 35 clock cycles on our system. That's a pretty big chunk of the 50 or so clock cycles we think each loop should take. It's difficult to know which of these is the greater bottleneck without an awful lot of testing, so a reasonable strategy is to try to fix one of them and see if the situation improves.

Strategies for Eliminating the Square Root

Let's consider applying the principle of choosing low-latency instructions to somehow replace the square-root function. Unfortunately, there's no obvious way to replace the square-root function with low-latency instructions that achieve the same functionality, so we can't simply apply a single principle in this situation. Instead, we have to consider multiple principles together, trading off the benefits of one for the penalty of another. Specifically, we can consider reducing the frequency of executing the square root by storing data in larger accumulation arrays, even though that will result in poorer spatial locality.

Table 1 summarizes three different strategies we might use to accumulate results in this calculation. The first option is the status quo: calculating `r` and accumulating the results in an array of size `MAX_r`. A second option would be to avoid the costly `SQRT` operation within the inner loop by accumulating results in an array indexed by r^2 instead of by `r`. The square root would eventually have to be performed, of course, but only for every nonzero value of `accum[r2]`, instead of for every pair of data points. A potential disadvantage of this option is the much larger size of the accumulation array, which might not fit in the processor's cache. Accessing this array randomly, without any benefit of spatial locality, would likely incur very high latencies.

A third option would be to store the result in a 2D array `accum2d[y_ind][x_ind]` of dimensions `MAX_y * MAX_x`. Again, this avoids the costly `SQRT` operation within the inner loop but at the cost of high latencies for randomly accessing the much larger accumulation array. Note that while this option avoids the two multiplication operations for squaring `Dx` and `Dy`, the array indexing arithmetic still requires an addition and a multiply to calculate the memory address for `y_ind * MAX_x + x_ind`. Additionally, the two absolute value operations are needed to avoid negative array indices. (Note that we've ordered our indices as `[y][x]` instead of `[x][y]`. Later, we diagram 2D arrays using the convention that horizontally adjacent elements represent neighboring elements in physical memory. The C programming language uses row-major order for multidimensional arrays, where array elements differing by one value in the *last* dimension are contiguous in physical memory. Our ordering of `[y][x]` lets the diagram's horizontal axis correspond to `[x]`, as on a standard Cartesian plane.)

The trade-offs in speed between these three options aren't at all obvious. The only real way to know whether one of the other two options will run faster is to pick one, write the code, and test it on a typical data set. Because the array size is smaller in the third option than in the second, we chose to test the third option.

Table 2. Effects on execution time of using a 2D accumulation array (as well as cosine precalculation, array merging, and blocking, described in previous installments).

	Execution time (sec)	Clock cycles per pair
With square root	2,634	79.6
With 2D output, not sorted	4,920	148.7
With 2D output, sorted	553	16.7
With all tricks	347	10.5

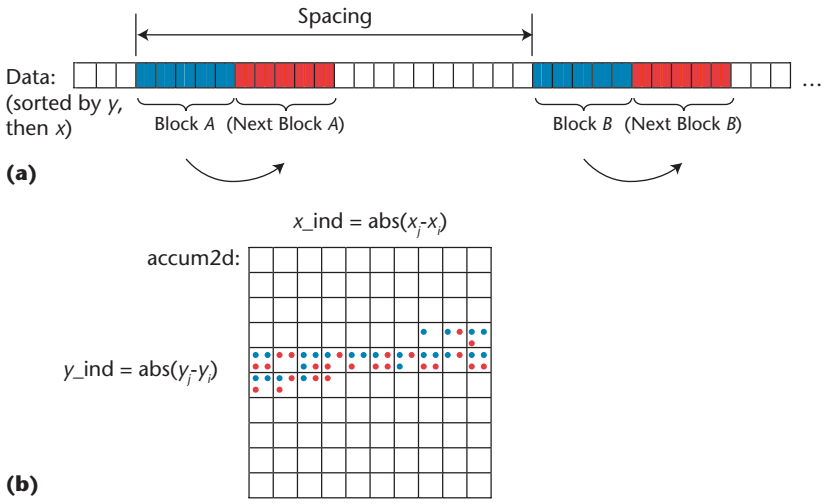


Figure 1. A revised blocking scheme for use with sorted data, showing (a) the sorted input array `data`, and (b) the 2D output array `accum2d`. The pair of blocks *A* and *B* in the input array shown in blue yield *i*,*j* pairs whose *x* and *y* displacements are localized in a region of `accum2d`, shown as blue dots. For the next pair of blocks *A* and *B*, shown in red, the spacing between the two blocks is held constant, yielding a set of pairs with similar *x* and *y* displacements, shown as red dots.

The first two lines of Table 2 shows the results of our modification. Where our original strategy of using a 1D accumulation array `accum[r]` yielded a speed of 79.6 clock cycles per pair, switching to the 2D accumulation array `accum2d[y_ind][x_ind]` actually took almost twice as long, 148.7 clock cycles. Apparently, the latency of randomly accessing the accumulation array is a more important consideration than the latency of the `SQRT` instruction. Clearly, the cost of trading poorer data locality exceeded the benefit gained from reducing the square root's frequency. Before giving up on this entire approach, however, it makes sense to try applying another of our principles (improving spatial locality) to this larger accumulation array. If we can ameliorate the cost of the increased memory usage, we might be able to still gain benefits from using the principle of removing high-latency instructions.

Improving Spatial Locality with Sorting

One strategy we haven't yet explored is sorting the data prior to processing to improve spatial locality. Because the data deals with spatial coordinates and we're calculating distances between the data, we might be able to use this fact to our advantage. Perhaps there might be some way to reorder the data so that pairs of points that fall within a particular region, or that are a particular distance apart, are all processed at nearly the same time. In this way, the accumulation array, whether indexed by `[r]`, `[r²]`, or `[Dy][Dx]`, might be accessed with some spatial locality, reducing the latency. Although it's difficult to imagine a scheme in which we could arrange the data to yield pairs neatly sorted by `r`, using the 2D accumulation array `accum2d` offers a fairly straightforward way to achieve a similar effect.

As Figure 1 illustrates, we start by sorting the input data according to the value of `data[].y`, and by `data[].x` within each value of `y`. (The sorting can be accomplished with a standard library routine requiring order *N*log*N* calculations, a negligible cost compared to the order N^2 calculations of our main program.) By sorting the data in this way, for each pair of points *i*,*j*, a small change in either *i* or *j* yields small changes in `Dy` and `Dx`. Using our blocking technique on the input data, each set of blocks *A* and *B* are guaranteed to yield pairs of points with little or no variation in `Dy`, and minimal variation in `Dx` as well. This greatly improves the spatial locality in the accumulation array—and all with no square-root calculation in the inner loop! As an additional small bonus, we no longer need an absolute value operation for `y_ind`, because `Dy` is now guaranteed to be positive or zero.

We can further improve spatial locality for the new accumulation array by making a subtle change in the blocking technique introduced in the second installment. In our original implementation, we held block *A* constant as we incremented block *B* through the entire data set. But by doing so now, data in each new block *B* would have larger *y* values than data from the previous block *B*, causing an increase in `Dy` for the resulting pairs of points. As a result, a

Table 3. Operations required for modified code, both for explicit instructions and address arithmetic.

Code line	Explicit operations		Address arithmetic	
	Integer	Float	Array	Likely operations used
for(j=B; j<B+blocksize; ++j) {	2 (+,<)			
x_ind = abs(data[j].x - data[i].x);	2(ABS,-)		`x[j]`	1
y_ind = data[j].y - data[i].y;	1(-)		`y[j]`	1

accum2d[y_ind][x_ind].g += data[i].cos6 * data[j].cos6 + data[i].sin6 * data[j].sin6;		4 (+,*,+,*)	`accum2d[][]`
			`data[j].cos`
			`data[j].sin`
				4
				1
				1

++accum2d[y_ind][x_ind].count;	1(+)		`accum2d[][]`	2
Totals	**6**	**4**		**10**

new portion of the array `accum2d` would be accessed, with some penalty in increased latency.

To avoid this, we can instead hold the spacing between blocks *A* and *B* constant, incrementing both *A* and *B* simultaneously, as Figure 1 shows. Assuming the data are uniformly distributed in *x* and *y*, the same portion of `accum2d` will be accessed for each set of blocks *A*, *B* of a given spacing. The trade-off for the increased spatial locality in `accum2d` is a decrease in spatial locality in the `data` array, but since the active portion of `accum2d` is larger than the size of a block in the `data` array, this seems like a good trade.

The code snippet that follows shows the implementation of the 2D accumulation array and the revised blocking scheme:

```
//Sorting of input data already done
//previously.
//The first four lines choose each possible
//pair of points i,j in an efficient order to
//maximize locality.
for(spacing = blocksize;
    spacing < (N-2)*blocksize;
    spacing += blocksize)
  for(A=0; A+2*blocksize<N; A+=blocksize) {
    B=A+spacing;
    for(i=A; i<A+blocksize; ++i)
      for(j=B; j<B+blocksize; ++j) {
        //For each ij pair, calculate r
        x_ind = abs(data[j].x - data[i].x);
        y_ind =     data[j].y - data[i].y;
        accum2d[y_ind][x_ind].g +=
           data[i].cos6 * data[j].cos6 +
           data[i].sin6 * data[j].sin6;
        ++accum2d[y_ind][x_ind].count;
      }
  }

//Finally, divide through by # of pairs at
//each r to get average.
for(y=0;y<MAX_y;++y)
for(x=0;x<MAX_x;++x) {
  r = sqrt(x*x +y*y);
  g[r]= accum2d[y][x].g /
     accum2d[y][x].count;
}
```

The third line of Table 2 shows the tremendous speed increase we gained by sorting the data. Each iteration of the inner loop now requires only 16.7 clock cycles, a whopping nine-fold speed increase from the unsorted version and almost five times faster than the version with the 1D accumulation array. Now that we've removed the bottlenecks due to calculating the square root and accessing the accumulation array, the spatial and temporal locality we achieved through blocking is truly paying dividends! Thus, making a trade-off between computation and memory, applying a combination of performance improving techniques, and incorporating application-specific knowledge about the data, we were able to make a set of modifications that significantly enhanced our already optimized code.

When Do We Stop?

At this point, having achieved some spectacular speed gains for only a small amount of work, it's worthwhile to take stock in our progress. Table 3 provides an estimate of the instructions executed in one iteration of the innermost loop of our code. Examining the table carefully, we see that it's composed entirely of relatively fast instructions like multiplication and addition that have a low latency and can often be pipelined. If we're looking for additional targets of opportunity for significant latency reductions—high latency instructions like square roots, division, or trigonometric functions—there's no obvious low-hanging fruit here.

Of the 20 instructions per loop iteration listed in the table, 16 are integer operations handled by our processor's three arithmetic logic units (ALUs). In the best possible theoretical limit, all instructions could be pipelined, and the most heavily scheduled ALU would handle six of these instructions, completing one loop iteration in just six clock cycles.

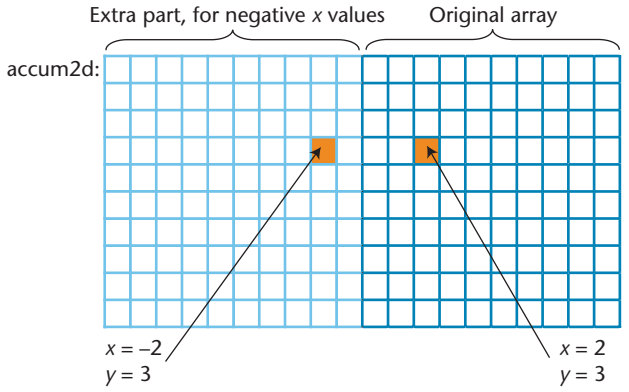


Figure 2. A modification to the accumulation array for avoiding an ABS calculation. With the original 2D array, shown in dark blue, an ABS calculation was required to keep the *x* part of the index from being negative. By extending the width of the array in *x*, and adjusting indices accordingly, this constraint is no longer necessary.

In reality, there will certainly be some overhead for "load" and "store" operations, as well as several three-clock-cycle latencies to retrieve items from the L1 cache. So, an actual time of two to four times that many clock cycles per loop would be a reasonable expectation. Since our measured time of 16.7 clock cycles per loop falls easily within that range, we can conclude that the program is now executing reasonably efficiently, no longer hampered by any particularly long latencies from memory access. As a practical matter, we shouldn't expect further code refinements to produce more than an additional 10 to 30 percent speed improvement. For most applications, this is fast enough.

Address Arithmetic: Taking Over the Controls

For some applications, pursuing an additional 10 to 30 percent is still worthwhile, and there are a few more tactics we can employ. Looking at Table 3, we see that 10 of the 16 integer operations are due to *address arithmetic* required for calculating the memory addresses of particular array elements. If we really want to squeeze the last few extra clock cycles out of the program, a good place to start is by thinking through how the processor handles address arithmetic. It might be possible to reduce the number of computations needed for this implicit code if we're willing to trade less readable code for improved performance.

For example, the output array `accum2d` is a 2D array of structures of type `OutputStruct`, each with two fields, `g` and `count`. The processor finds the memory address of `accum2d[y_ind][x_ind].g` by calculating

```
address of accum2d[0][0].g +
   (size of OutputStruct) *
   (x_ind + y_ind*MAX_x)
```

Notably, the multiplication by `MAX_x` takes place each iteration, and we can move it outside of the inner loop if we take over some of this address arithmetic from the compiler and handle it ourselves explicitly.

We start by defining `accum2d` as a 1D array of size `MAX_y * MAX_x`. Instead of addressing it as `accum2d[y_ind][x_ind]`, we use `accum2d[x_ind + MAX_x * y_ind]`. Because `y_ind` comes directly from the `y` values in the data array, we can simply multiply all values of `data[].y` by `MAX_x` at the start of the program. From that point on, we simply address the accumulation array as `accum2d[x_ind + y_ind].g`, having eliminated the need for a multiplication by `MAX_x` inside the inner loop. Thus, we can remove computation on every iteration through precomputation and by modifying our algorithm and its data storage based on application-specific information. However, the code does become less readable.

(The astute reader will note that the implicit multiplication by the size of `OutputStruct` could also be precalculated in much the same way, removing yet another instruction from the inner loop. However, the C code for doing so looks so ghastly that we can't bring ourselves to include it here. Instead, we'll leave it as an exercise for the reader who enjoys that sort of thing.)

A final tactic we can employ to remove some computation requires a willingness to change data structures in a way that makes them less intuitive and less directly expressive of an algorithm's logic. In this vein, Figure 2 illustrates one final bit of trickery to remove the last remaining `ABS` operation, which is there to prevent the *x* part of the array index from ever being negative. Because we're handling the array arithmetic for the array `accum2d` ourselves anyway, we can handle negative *x*-values with no trouble, provided we create an array that's twice as big as before, and we scale all of the *y*-indices appropriately. Furthermore, once we remove the absolute value from the calculation of the index, it's clear that we can eliminate another operation by grouping the `data[i]` terms together. Our index for `accum2d` becomes `data[j].x + data[j].y + d_i`, where we calculate `d_i = - data[i].x - data[i].y` outside of the innermost loop. The trade-off here is that the array `accum2d` is now twice as large as before, and there is a chance that we could suffer from reduced spatial locality as a result. In fact, the trade works in our favor this time. The last line of Table 2 shows that with these last changes, including our takeover of address arithmetic from the compiler, our program's execution time is now down to 10.5 clock cycles per iteration.

Many introductory programming textbooks stress the importance of using algorithms that scale efficiently—for instance, with execution times proportional to $N\log N$ instead of N^2. By contrast, most textbooks pay little if any attention to the actual proportionality constant itself. But computer architecture has evolved in recent years to a point where CPUs can perform some calculations at a rate of several operations per clock cycle, roughly 1,000 times faster than the rate at which individual bytes of data can be loaded from main memory. In this environment, that proportionality constant can make the difference between "fast enough" and "too slow to be useful."

In this three-part series, we started with a simple algorithm consisting of roughly six lines of code, and we used a few broadly applicable general principles to increase its speed by more than a factor of 30. Although we're pleased at the near optimal efficiency of our final program, the huge improvement we achieved is ultimately due to having begun from a starting point of maximum stupidity. In fact, the main lesson to be learned from this example is that most of the stupidity could have been easily avoided from the start if we had initially written our program with some attention to how our computer hardware would actually execute it. The real take-home message of this series is that even for casual programmers, computer architecture really does matter!

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.