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

WHY COMPUTER ARCHITECTURE MATTERS

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

Over the course of a three-part series, the authors will walk through the implementation of a simple but computationally intensive algorithm and show how a series of incremental refinements to the code yields significant performance gains. In this first installment, they concentrate on instruction selection and scheduling.

When we write computer programs for data analysis or simulations, we typically translate the relevant mathematical algorithm into computer code as directly as possible, with little regard for how the computer will actually perform the computations. We choose to ignore such details as how individual calculations are divided among the several arithmetic and floating-point units (FPUs) on the CPU, how data is shuttled between the CPU and the main memory, and how recently used data is temporarily stored in the computer's various levels of memory caches. These questions fall under the broad heading of "computer architecture," and in many cases, our intentional ignorance serves us well. A compiler can usually translate our programs into reasonably efficient machine code quite effectively—and for many short calculations, modern computers are already many times faster than they need to be. For longer computations, however, programming with a little bit of attention to the architecture can sometimes produce gains in execution speed that are significant enough to make the extra effort worthwhile.

Computer architecture has changed drastically over the past 10 to 20 years. Although it goes without saying that these developments have made computers much faster than they used to be, what is less obvious is that they've also significantly changed the relative speeds of different types of operations. Floating-point operations used to be relatively costly, for example, but now their speed is closer to being on par with integer operations. In contrast, reading and writing to a computer's main memory, although faster than ever before in absolute terms, now represents a huge bottleneck for modern processors.[1] In fact, a modern processor with two FPUs can sustain a maximum throughput of two floating-point calculations per clock cycle, whereas a single access of main memory could incur a latency of several hundred clock cycles. When execution speed is important, these architectural realities must inform our decisions as computer programmers.

Over the course of the next three issues, we present a discussion of several aspects of computer architecture that scientific computer programmers should bear in mind when execution speed is important. We frame our discussion around a single, real-world example: calculating an orientational correlation function.

A Concrete Example

Many materials consist of small crystallites, each oriented in a different random direction. For such materials, the orientational correlation function $g(r)$ tells us the characteristic length scale over which the local orientation remains uniform; comparing $g(r)$ for different material samples lets us quantitatively compare crystallite sizes. Suppose that we have a microscope image of a piece of material from which we can measure the local crystallographic orientation θ_i for N data points (x_i, y_i) in two dimensions, as shown in Figure 1. For a hexagonal lattice, in which the local orientation is defined modulo 60°, we define the orientational correlation between any pair of points (x_i, y_i, θ_i) and (x_j, y_j, θ_j) as $g_{i,j} = \cos[6(\theta_i - \theta_j)]$. Its maximum value is $g_{i,j} = +1$ if the two orientations are the same or differ by a multiple of 60°, and its minimum value is $g_{i,j} = -1$ if the two orientations differ by 30°, the greatest possible misorientation. On average, we expect $g_{i,j} = 0$ for two points that are very far apart because their orientations should be neither positively nor negatively correlated. (That is, the orientations of two very distant crystallites are equally likely to differ by any value between 0° and 60°.) We define the orientational correlation function $g(r)$ as the average value of $g_{i,j}$ for all pairs of points separated by a distance

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

that is, $g(r) = \langle \cos[6(\theta_i - \theta_j)] \rangle_r$, where "$\langle\rangle_r$" denotes the average.[2]

The following code snippet shows a straightforward implementation of the calculation. We assume that input

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

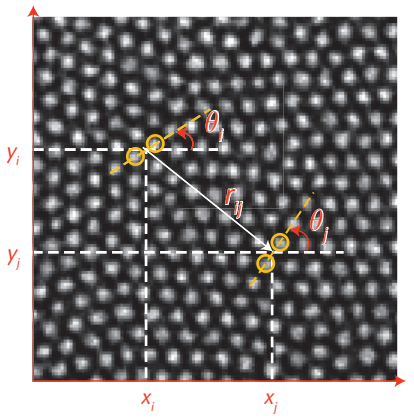

Figure 1. Material composed of two crystallites of different orientations. The local orientation θ of the lattice at a location (x, y) is defined by the orientation of a line segment joining two adjacent lattice points.

Table 1. Typical latencies for integer and floating-point operations on current processors.*

	Operation	Typical latency
Integer operations	Addition/subtraction	1–4
	Multiplication	3–6
	Division	16–42
Floating-point operations	Addition/subtraction	4–6
	Multiplication	4–6
	Division	16–38
	Square root	6–70
	Trigonometric functions	92–200

**The range for each entry reflects differences depending on the sizes of the operands and where they're stored, as well as differences between architectures.*

data for N points have already been read into three parallel arrays `x[N]`, `y[N]`, and `theta[N]`:

```
//First, accumulate g(r)
//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 > i
    Dx = x[i]-x[j];
    Dy = y[i]-y[j];
    r = sqrt(Dx*Dx+Dy*Dy); //distance r_ij
    g[r] += cos(6*(theta[i]-theta[j]));
                            //add g_ij to g(r)
    ++count[r];             //increment count
  }                         //end of j loop

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

Note that because of the doubly nested loop, this code's execution time is proportional to N^2. For large images with $N \approx 10^6$ points, this code as written takes almost a day to complete on a typical single-processor workstation. Clearly, for a calculation that must be performed on several images, it's worthwhile to try to improve the speed.

Not All Operations Are Created Equal

Examining the five lines of code inside the inner loop, we can get a very rough idea of the computational load that this calculation should require. Each pair appears to have six regular additions or subtractions, three multiplications, one square root, and one cosine. (We discuss the additional computational load associated with address arithmetic later.) Table 1 shows typical latencies for some of these calculations, in which latency is the number of clock cycles required to perform a single operation. Although the wide range for each latency makes it hard to make many generalizations, it's clear that we should avoid the use of division, square roots, and especially trigonometric functions wherever possible.

Of course, we can often avoid unnecessary division operations by paying attention to order of operations—for instance, we'd expect the expression `(x/y)/z` to have a higher latency than `x/(y * z)`. Likewise, when dividing a series of numbers by π within a loop, it helps to first define a variable `inv_pi = 1.0/3.1415` outside the loop, replacing each division inside with a multiplication.

Of the instructions in our code, the cosine function is by far the most costly. In this example and in other calculations, finding some way to avoid trigonometric functions is a good place to start looking for efficiency gains. Some possible strategies include approximations using a Taylor expansion ($\cos x = 1 - x^2/2! + x^4/4! - \ldots$) or creating a small lookup table of precomputed values if we can sacrifice some precision for greater speed.

For our example case, it's possible to remove the calculation of the cosine from the inner loop with no loss of precision using the trigonometric identity

$$\cos(\theta_i - \theta_j) = \cos\theta_i \cos\theta_j + \sin\theta_i \sin\theta_j.$$

Although this would appear to replace a single trigonometric function with four of them, the advantage is that we need to calculate each of the four only N times, once for each of N individual data points, rather than an order N^2 times, once for each pair. Within the inner loop, we've now replaced the cosine operation, one subtraction, and

the multiplication by six with two multiplications and one addition. Our code snippet now looks like this:

```
//First, calculate all sines and cosines
for(ii=0; ii<N; ++ii) {
  sin6[ii] = sin(6*theta[ii]);
  cos6[ii] = cos(6*theta[ii]);
}

//Now, accumulate data for all pairs of
//points (i,j).
for(i=0; i<N; ++i)
  for(j=i+1; j<N; ++j) {
    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];
  }
```

To test this optimization's effect on speed, we ran both versions of our code on a test system. (Our test system consisted of an AMD Athlon 64 3500+ processor [at 2.2 GHz] on a Gigabyte GA-K8NXP-SLI motherboard, with 2 Gbytes of RAM. We used the same system on all tests throughout this series.) Table 2 shows the results of applying the cosine precalculation; in this case, it resulted in a speed increase of more than a factor of two. (When applied in conjunction with other optimizations described later in this series, the speed increase yielded by the cosine precalculation varies somewhat, typically reducing execution time by between 50 and 150 clock cycles per pair.)

Table 2. Effects of cosine precalculation on execution time.*

	Execution time (*N* = 382,000 points)	Clock cycles per pair (inner loop only)
Without cosine precalculation	10,976 seconds	331.7
With cosine precalculation	4,422 seconds	133.6

**In this and subsequent tables, "clock cycles per pair" includes only the time for processing the inner loop.*

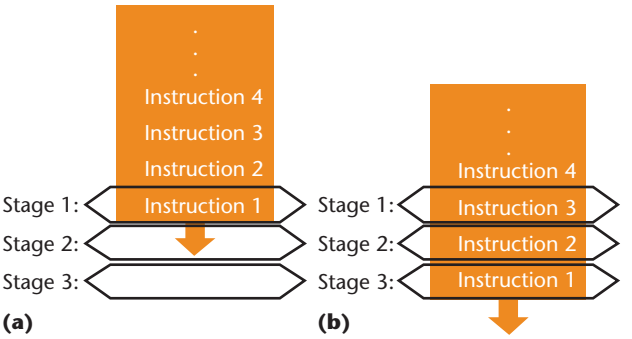

Figure 2. Instructions executed in a pipelined instructional unit with three stages. (a) Instruction 1 has just entered the first stage of the pipeline. (b) Two clock cycles later, instruction 1 is in the last stage, and instruction 3 has just begun.

Logic Units and Pipelines

Turning our attention to the other operations in the inner loop, multiplications and additions are all handled by arithmetic logic units (ALUs) for fixed-point (integer) calculations and FPUs for floating-point calculations. Our AMD Athlon CPU has three ALUs and two FPUs (one for addition/subtraction/shifts and one for multiplication/division/square root); the CPU schedules tasks on these functional units based on which is free. Although each arithmetic or floating-point operation takes several clock cycles to run from start to finish, *pipelining* lets a new calculation start and a previous calculation complete every clock cycle. With pipelining, each functional unit takes the form of a miniature assembly line, as Figure 2 shows. Like an object in an assembly line, each instruction must have a series of operations performed on it in a specific order. When building the functional unit, the chip designer creates stages in the hardware corresponding to these operations and then connects them in a series. At any given time, an instruction can be in exactly one stage, allowing other instructions to simultaneously use other stages. Thus, the maximum CPU throughput is one calculation per clock cycle for each functional unit (although whether this can be maintained depends on the order of operations a particular computation requires).[3] In some cases, operations that don't directly depend on each other can also be processed out of order, allowing the functional units to be scheduled more efficiently.

Because the pipeline for floating-point operations is longer than that for fixed-point operations (meaning each individual floating-point operation takes longer to complete than each integer operation), we might suspect we could improve performance by converting floating-point numbers to integers where feasible. For example, we could multiply our precalculated sines and cosines currently stored as floating-point numbers by, say, 1,000, and then store them as integers, with an acceptable loss of precision. In our example, applying this "optimization" yields mixed results, sometimes speeding up the code by 10 to 20 percent and

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

Code line	Explicit operations		Address arithmetic	
	Integer	Float	Array	Likely operations used
`for(j=i+1; j<N; ++j) {`	`2 (+,<)`			
`Dx = x[i]-x[j];`	`1(-)`		`x[j]`	1
`Dy = y[i]-y[j];`	`1(-)`		`y[j]`	1
`r = sqrt(Dx*Dx + Dy*Dy);`	`3(*, +, *)`	`1(sqrt)`		

`g[r] += cos6[i]*cos6[j]+ sin6[i]*sin6[j];`		`4(+, *, +, *)`	`g[r]`
			`sin6[j]`
			`cos6[j]`
				2
				1
				1

`++count[r];`	`1(+)`		`count[r]`	2
Totals	**8**	**5**		**8**

sometimes slowing it down by roughly the same amount, depending on what other optimizations we use. Apparently, one can negate the shorter ALU pipeline's advantage if the ALUs are more heavily scheduled than the FPUs (for instance, for address arithmetic), a condition known as a *structural hazard*.[1]

Counting Operations in the Inner Loop

In considering further optimizations, it helps to make a rough count of the operations currently used in our code's inner loop (see Table 3). The columns in the table under "Explicit operations" show the specific arithmetic operations, such as additions and multiplications, required in each line of the code. (Note that Table 3 doesn't count any of the "load" or "store" operations that many architectures require. Our table is meant to serve as a very rough estimate of the computational load, a full accounting of which would require careful examination of the compiled assembly code.)

Table 3 also includes two operations associated with the "for" loop itself: one integer addition to increment the value of `j` and one "less than" operation to compare the value of `j` to `N` and determine whether to return to the loop code or continue to the code following the loop. (Depending on the architecture, this comparison and branch to the appropriate next instruction might take one or two instructions per loop iteration.) To avoid this branch overhead on every loop iteration, compilers frequently use a technique called *loop unrolling*.[1] With this technique, the compiler schedules multiple loop iterations for each branch determination.

Address Arithmetic: A Hidden Drain

The final two columns in Table 3 provide an estimate of the additional computational load required for address arithmetic, such as calculating the actual memory location associated with an array's `i`th or `j`th element. As Figure 3 shows, for example, accessing the `j`th element of array `x[]` typically implies multiplying index `j` by the size of each array element (4 bytes, for a long integer) and adding this to the address of the array's first element. Thus, to a first approximation, each reference to a one-dimensional array element would require two integer operations—similarly each reference to a two-dimensional array element would require four integer operations.

However, a good compiler can minimize the load for address arithmetic in various ways. For one, it can save recently calculated addresses in registers to avoid unnecessary calculations—in our code, for example, the value of `x[i]` remains the same throughout all the inner loop's iterations, so the value would likely be held in a register rather than be recalculated with each iteration. Therefore, Table 3 doesn't show any operations as being required for `x[i]`.

The compiler can also reduce the load for calculating `x[j]`. Because the inner loop code only uses the variable `j` as an array index and increments it sequentially with each loop iteration, a good compiler will probably avoid the cost of multiplication by keeping an address of the current array element and simply adding (or subtracting) the array element's size on each loop iteration. Thus, only one integer operation persists, but the processor uses a register to retain the current array element address. With multidimensional arrays in which only one dimension changes in a given loop, the compiler will store the current address of array elements in the current row. Table 3 estimates only one operation will be required for referencing `x[j]`, `y[j]`, `cos6[j]`, and `sin6[j]`. Even assuming the compiler can take all available shortcuts, we still estimate that address arithmetic will re-

quire eight integer operations, equal to the eight integer operations that are explicitly a part of our code's inner loop.

Estimating Clock Cycles

By our estimation in Table 3, our inner loop should require roughly 16 integer operations. In the very best theoretical limit, these operations could all be distributed efficiently on the three ALUs so that the most heavily scheduled unit would have six pipelined instructions per loop, all optimally pipelined with a throughput of one operation per clock cycle, for a total time of six clock cycles per iteration. Given the number of complications we've swept under the rug, an actual number of two to four times that would be a reasonable expectation for the integer instructions alone.

The five floating-point instructions in our inner loop are more likely to be the limiting factor because they include a costly square root that, on our test system, requires 35 clock cycles and can't be pipelined. Again, accounting for some amount of overhead in our calculations, and taking into consideration that it might not be possible to schedule all integer operations simultaneously with the floating-point operations, we might reasonably guess that each iteration of the inner loop might require roughly 50 clock cycles.

However, Table 2 shows a different story: in fact, after precalculating the sines and cosines, each iteration actually requires more than 130 clock cycles. This discrepancy indicates that significant inefficiencies remain in our code, and that further scrutiny is warranted.

As it turns out, the major reason for this large discrepancy is the latency associated with reading a value from the computer's main memory into the CPU. In part two of this series, we will examine some of the architectural intricacies of the memory system on a typical computer, and examine ways in which data can be organized to take advantage of it (or at least avoid being heavily penalized by it). In the third and final part, we look at how some additional minor tweaks to an algorithm can yield substantial gains in performance.

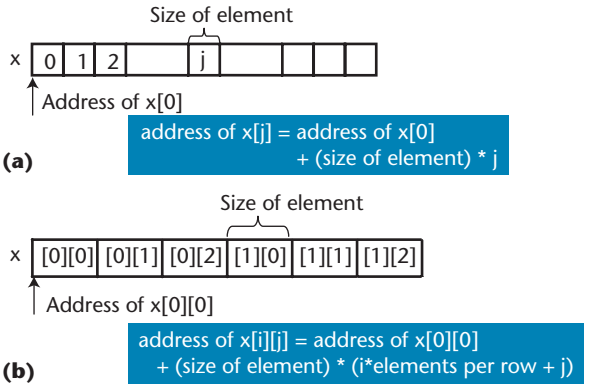

Figure 3. Address arithmetic for arrays. (a) The storage arrangement of a one-dimensional array and how it's referenced, and (b) the storage arrangement of a two-dimensional (2 × 3) array and how it's referenced.

References

1. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.
2. P.M.Chaikin and T.C. Lubensky, *Principles of Condensed Matter Physics*, Cambridge Univ. Press, 1995.
3. J.P. Shen and M.H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processing*, McGraw Hill Higher Education, 2005.

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.

How to Contact *CiSE*

Writers
Visit www.computer.org/cise/author.htm.

Subscribe
Visit https://www.aip.org/forms/journal_catalog/order_form_fs.html or www.computer.org/subscribe/.

Missing or Damaged Copies
For CS subscribers, email help@computer.org.
For AIP subscribers, email claims@aip.org.