CMSC 222: Program Optimization

Claxton, Spencer

3 Dec 2013

Abstract

In this paper I present some single threaded optimizations of integer matrix multiplication and integer sort. These optimizations hinge on the concepts of data locality, SIMD, branch optimization, and algorithms. I am using a naive implementation of matrix multiplication and a simple, unoptimized implementation of bubble sort as my baselines for optimization performance.

Contents

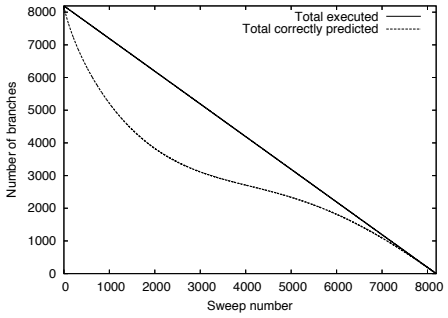
1 Introduction

All data was gathered with Instruments 4.1, a native OS X profiling application, on a mid-2012 MacBook Pro with the following specs:

- 2.5GHz dual-core Intel Core i5 processor
- 3MB L3 cache
- 4GB of 1600MHz DDR3 memory

2 Sort

The baseline sorting implementation used leaves much to be improved. Bubble sort is an in-place sorting algorithm that has time complexity $O(n^2)$ and space complexity $O(1)$. Bubble sort is slow because it makes on the order of n^2 comparisons and in the worst case has on the order of n^2 writes. It also sufferes greatly from branch misprediction latency, since it makes so many branching comparisons. In his paper, "*An Experimental Study of Sorting and Branch Prediction*", N. Nash includes the following plot of branch misprediction in naive bubble sort:



It is clear from this plot that branch mispredictions have a significant affect on performance, since if they penalty for branch misprediction is on the order of 10 cycles the overall execution time is heavily affected.

No branch

That said, my first optimization of bubble sort was removing the main branch by rewriting the loop body and using the O3 compiler optimization flag. The loop body of naive bubble sort is as follows:

```
1 if (array[i]>array[i+1]) {
      swap(i, j)
  }
```

With compler optimization O3, this gets assembled as:

```
  movl    (%rdx), %esi
2 cmpl    %esi, %ebx
  jle     switch_elmts
  movl    %esi, -4(%rdx)
  movl    %ebx, (%rdx)
  jmp     loop_increment
```

I rewrote the body of the loop to be the following:

```
  tmp = array[d];
  swap = array[d] > array[d+1];
3 array[d] = swap ? array[d+1] : tmp;
  array[d+1] = swap ? tmp : array[d+1];
```

Which assembles to:

```
1 movl    (%rdx), %ebx
  cmpl    %ebx, %eax
  movl    %eax, %esi
  cmovgl  %ebx, %esi
  movl    %esi, -4(%rdx)
6 cmovll  %ebx, %eax
```

Thus, we see that the jumps have been replaced by conditional moves, which are branchless. The result is that when sorting a list of 100,000 elements, there are roughly 33% less mispredicted branches in this optimized bsort.

Algorithm change: Quick sort

My second optimization was changing algorithms. I changed algorithms to quick sort becuase it more immediately and elegantly solves most of bubble sorts problems. Quick sort moves an element to its final position with each step, however, after each step, the algorithm is applied recursively to the two parts of the list to the right and left of the element in its final position. So quick sort makes much fewer comparisons in total than bubble sort. Quick sort also only swaps two elements once per an iteration, unlike bubble sort. Thus, with far fewer writes and reads than bubble sort, quick sort will almost certainly have much fewer cache misses.

Optimized Quick Sort

I further optimized Quick Sort per Sedgewick et al. in their landmark paper on the then novel algorithm. This optimization consisted of three major steps: (1) I made the algorithm non-rescursive and simulated the recursive steps with a stack data structure, (2) I chose the pivot element using the median-of-three methodology, (3) I set a maximum depth of recursion so that small lists in the later steps of the algorithm are sorted by insertion sort, which is more efficient for smaller inputs.

Getting rid of recursion eliminates the need for the overhead that comes with the function stack. Since jump and link instructions are expensive, cycle time is reduced when they are eliminated. The pivot being chosen as the median of the first, last, and middle elements of the list prevents the worst case of quick sort form occuring, which happens when an extreme pivot element is chosen at every step. The last optimization is arguably the most important. By calling insertion sort on the small lists, we greatly reduce our instruction count.

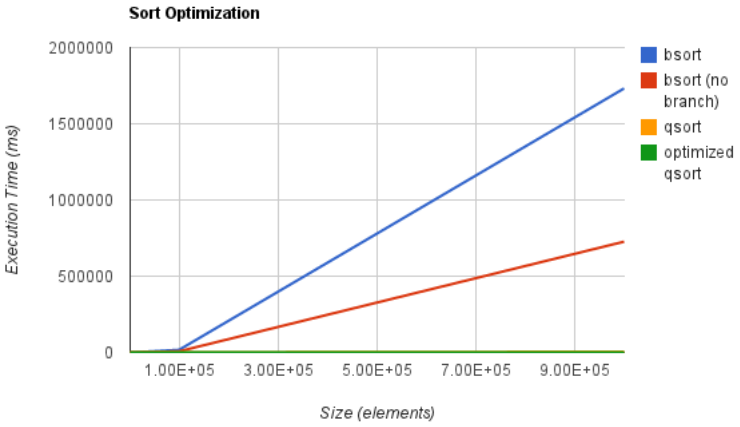
Originally, Sedgewick advised calling insertion sort only once after the algorithm had finished recursing to it's maximum depth, leaving small sections of the whole list unsorted. As noted by A. Lamarca in his paper "*The Influence of Caches on the Performance of Sorting*", though he single pass of insertion sort on the whole list is efficient from an instruction count standpoint, it is not efficient from a cache locality stand point. Lamarca suggests sorting each small section of the list as it is encountered in the recursion. This is beneficial because when such a small section is first encountered, it has just been partitioned by the algorithm and is thus guaranteed to be in the cache. I have decided to implement this more cache efficient technique, as cache awareness has become more of an immediate issue with contemporary technology since processors keep getting faster and faster.

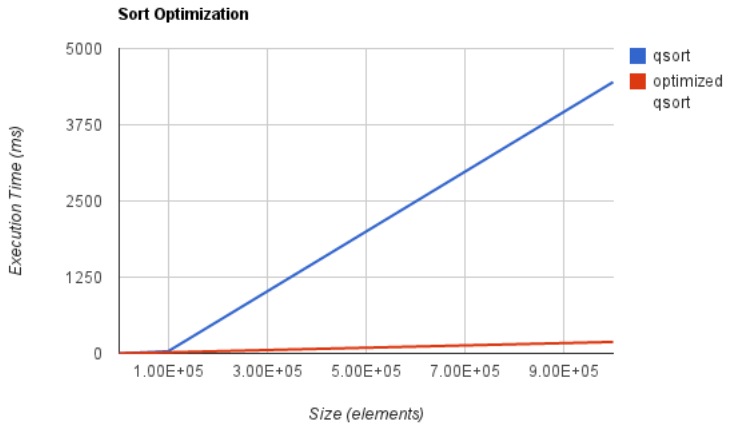
Results

The following measurements are taken from an experiment sorting an array of size 10^6 elements. These measurements are averages taken over a sample size of 5:

Optimization	Time (ms)	L1D Misses
bsort	1,732,189	50,750,218,975
bsort (no branch)	726,742	42,386,356,279
qsort	4,451	27, 85,192,323
optimal qsort	188	6,496,580

Below is a graph of execution time vs. size and then the same graph just detailing quicksort and optimized quicksort, since they are indistinguishable on the first graph:





3 Matrix Multiplication

The baseline matrix multiplication implementation is also greatly improved by data locality, but

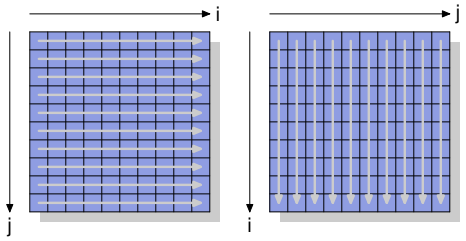
it can also be improved by vectorization with SSE instrinsics. We will discuss vectorization in greater detial in the conclusion.

Data Structure

It is worth noting that when computing with 2-d rectangular matrix, keeping the 1-d C array is worthwhile, since it stores the matrix as one block of contiguous memeory. This is good since if the access to the array (matrix) is regular, the hardware will prefetch data from further down in the array for when we need it later.

Transpose

A straightforward way to increase data locality in the matrix multiplication implementation given is to just take the transpose of the matrix being multiplied. This increases spatial locality immensely, since both arrays are being traversed sequentially by row, i.e. memory in the array is being accessed sequentially. The following graphic is included in U. Drepper's " *What Every Programmer Should Know Memory*" and illustrates this point nicely:



However, this regular access pattern does not come without a cost. There is significant overhead in transposing the matirx before multiplication. As a consequence, for smaller values, this method is not mo

Block Multiplying

Another way to increase data locality is to multiply the matrix in blocks. That is, one can multiply the elements of two matrices in such a way that blocks of the result matrix are computed one after the other. This technique allows for elements that are in the same block computation to be accessed closer together spatially and temporally. In particular, if the block is the proportional to the size of the cache line, this technique allows for elements in both matrices being multiplied to be accessed in such a way that maximizes chache hits.

Indeed, the reason the naive implementation is bad in the first place is because there is no data locality to be had in that implementation. Blocking solves the problems of data locality, without the overhead of transposing the matrix.

Vectorizing

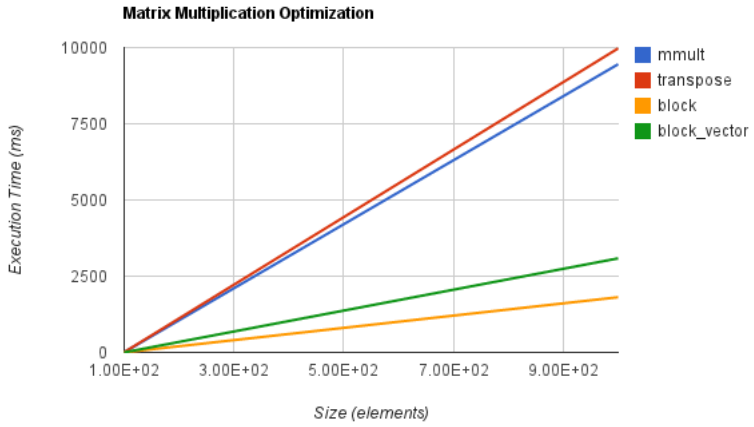
For the final optimization of matrix multiplication, I put vectorized operations on top of the block multiplication implementation. The vectors allow multiple data to be computed over simultaneously. This optimization did not prove to be better than just block multiplication for smaller values. This can probably be explained by the overhead involved with SSE instrinsics.

Results

The following measurements are taken from an experiment a 1000 x 1000 matrix. These measurements are averages taken over a sample size of 5:

Optimization	Time (ms)	L1D Misses
mmult	9,454	1,278,606,853
transpose	9,974	1,267,717,533
block	1,810	27, 34,519,715
vector block	3,085	41,917,439

Below is a graph of execution time vs. size of the martrix being multiplied:



4 Conclusion

Not taking into account the generating of the list, my best sorting optimization is roughly 1000x faster than the original bsort code. My best matrix multiplication optimization is roughly 5x faster.

Vectorization did not work for me particularly wellin this instance, though many things oculd be changed to further optimize the SSE code. One issue with my SSE code might be that it is not efficiently loading and storing data. I could possibly store the data with write combining so that it is written back asynchronously. Software prefetching is a particularly interesting avenue for more efficiently loading vectors. If vectors were longer, this could also increase my gains. For instance, if we somehow had vectors the length of the cache line, loading and storing vectors could be potentially optimal.