Iterative Performance Optimization

Project 2-2

Iterative Performance Optimization

Last Updated: October 18, 2010

In the first part of this project, you learned about the performance characteristics of a number of rotation and sorting programs. In this part, you will implement your own programs to solve these problems. Be sure to keep in mind the lessons that you learned from the first part and the material that we have been covering in lecture; all of this will help you design and perfect fast solutions.

Approaching this assignment

Now that you have profiled and inspected the application binaries left behind at SnailSpeed, Ltd., you are ready to re-implement your own versions of these applications. After closely examining SnailSpeed's needs, you have decided that it is sufficient to implement rotation and sorting programs that operate on 32-bit data only. Your goal is to implement a number of versions of the two programs and present the best versions to your boss.

Getting the code

You each have been given a new `project2.2` repository with the code for this assignment. As before, you can get the project code using git:

```
% git clone /afs/csail/proj/courses/6.172/\
   student-repos/project2.2/username/ project2.2
```

Submitting your solutions

As before, submit your write-up on Stellar and your code with Git before the deadline stated above. Remember to explicitly add new files to your repository before committing and pushing your final changes.

```
git add <new-files>
git commit -a
git status
git push
```

If `git status` shows any modified files, then you probably haven't checked your code into your repository properly.

In keeping with good programming practice, you should check in incremental changes to your repository as you write and test your code. We were lenient with students who had problems pushing their changes for the first project, but we will henceforth expect you to submit your code properly by the due date.

A quick note about `git commit -a`: this command will commit all modifications to tracked files to your local repository. If you only wish to commit changes to certain files, you can `git add` them explicitly and `git commit` will commit only files that have been manually staged (use `git status` to check what's been staged). If this is confusing, please just stick to `git commit -a`.

Writeups

For the beta, we ask you to submit as small writeup with the goal of helping the staff (who understand the assignment and its general strategies but have never seen your code) to fairly grade your assignment. We don't expect beta writeups to be a large formal writeup, but we suggest you include the following:

- A brief overview of your design, particularly what improvements you made over the beta. This is not designed to replace appropriate code documentation, though.
- The general state of completeness/expected performance of your implementation, as well as any bugs/-gotchas that you are aware of.
- Any additional information that you feel would be helpful to the staff in evaluating your submission (e.g. if you spent a lot of time on approaches that didn't result in a speedup, etc)

Finally, for the final submission, we expect an even shorter writeup to accompany it with:

- An overview of changes you made to your beta submission, and what motivated you to do it (surprised with performance ranking? New revelations after master meeting? Ideas conceived before the beta deadline but ran out of time implementing it?)
- Some comments on meeting with your MITPOSSE mentor.

Code layout and Makefiles

The `main` function for both sections of this project will be located in a `testbed.c` file, which you cannot modify (even if you do, it will be overwritten with a fresh copy during our tests). You will only modify the functions implemented in `rotateN.c` and `sortN.c` as specified in the following sections. You will have to edit the Makefiles we give you by adding new targets to the top as you implement them (it will be obvious what to do when you read them).

The Makefiles will build 32-bit and 64-bit binaries as in Project 1. **While the 32-bit binaries maybe be userful for testing on machines other than the clouds, whenever a question asks about performance, we want you to run your programs on the cloud machines using PNQ** (use the `pnqsub` script to submit jobs). Type `make run` to run some simple test cases on your code.

1 Image rotation

Assume you have decided that the only way to achieve the best possible implementation of image rotation is to experiment with a number of different algorithms. In the real world, you will often find that this is the case. You will start by writing a correct but naïve implementation of image rotate. (This is also usually a good idea; premature optimization should be avoided!)

You should avoid increasing the optimization level to `-O3` for this part. This optimization level would enable automatic loop reordering, which would make it harder for you to see the effect of manually reordering loops.

1.1

Run `rotate1` (via PNQ) with the input sizes 511, 512 and 513 (and, say, 10,000 repetitions). How does the runtime scale with the number of bytes (4 per pixel) in the matrix to be rotated? How can you explain this behavior? (*Hint: The L1 data cache is 32KB in size and 8-way associative. How might rotate1's memory access pattern be causing poor cache performance?*)

1.2

For what input sizes might this problem occur, and why does it not occur in larger matrix sizes, such as 2048? Determine a way to avoid this problem, and implement your change. In your writeup, explain how the changes you made prevent the problem.

1.3

Duplicate `rotate1.c` and rename the copy `rotate2.c`. Exchange the inner and outer loops. You will also need to edit your `Makefile` so that it builds `rotate2`. We've left some comments in the Makefile to help you do this.

1.4

Create a `rotate3.c` and update the `Makefile`. Using rectangular blocks, implement the blocking access pattern you came up with in Project 2.1. Tune your block size to obtain optimal performance, and report the optimal block dimensions. Also include a table of the measurements that led you to this block size. *Hint:* For an input of size $n = 10,000$, you should be able to keep your L1 cache miss rate below 4%.

1.5

Create a fourth version of the program that performs the rotation using the following divide-and-conquer approach.

- Divide the matrix into four equally-sized submatrices.
- Recursively call rotate on each of these submatrices. Remember to pass the appropriate `src` and `dest` pointers.
- When the matrix is small enough, switch over to the algorithm you used in either `rotate1` or `rotate2`.

After you finish writing this program, tune the matrix size at which you switch over to your "base case" algorithm. Again, include a table of the measurements that you perform while doing so.

You will probably want to begin by only handling cases where the input size n is a power of two. Then, generalize so that your program can handle any n. What special case appears when n is not a power of two, and how can you deal with it? *Hint:* How will your recursive function get a particular element of a submatrix from memory? *Hint:* It may be useful for your recursive function to know what the size of the original matrix was.

1.6

In `ssetranspose.c`, we have provided a code fragment that uses Intel's SSE instructions, which are a widely-supported extension to the x86 instruction set, to perform a very fast 4x4 matrix transpose. We use SSE intrinsics to ask the compiler to emit SIMD instrictions. This function takes four input arrays (each representing a four-element row) and four output arrays (each, again, a four-element row) as parameters.

Think about how elements are moved when a 4×4 matrix is transposed, and compare this to how they are moved when a 4×4 matrix is rotated. (You should probably get a pen and paper and walk through an example.) You should see a simple relationship between the transposed matrix and the rotated one. What is it? Make some minor tweaks to the provided code so that it computes a rotated matrix instead of a transposed one, and rename the function accordingly.

1.7

Create a fifth program, `rotate5`, which extends your blocking rotate program (`rotate3`) to take advantage of this new SSE rotation function. You may assume that this program will only be given inputs whose size is a multiple of four.

The SIMD instructions in this function require that the memory they operate on be aligned to 16-byte boundaries. As you may have noticed, the code which we provided allocates memory so that the array as a whole is aligned. However, if you choose to invoke your SSE rotation function on a subarray, you must make sure that the subarray is also aligned. If you do not, your program will crash with a segmentation fault.

1.8

Compare and contrast the performance of each of your implementations. Can you explain why each version performs the way it does? Provide well-presented data to substantiate your claims.

Pick whichever of your rotate programs you would like us to time, copy it to `rotate.c` and add the `rotate` target to the Makefile. We'll test all six of these programs for correctness (although two of them should be identical), but only the one you select will be tested for performance.

Correctness

The testbed we have given you already checks for correctness and handles timing. If you add your own output statements (for example, calls to printf) you should make sure that the correctness result, which is printed after your function returns, remains on its own line. A submission will be judged correct if it passes this correctness test for any input size n such that $n \geq 0$. (Your last program, `rotate5`, is only required to handle cases where n is a multiple of four.) You may assume that n will fit in a 32-bit signed integer.

2 Sorting

2.1

We have provided a simple implementation of insertion sort in `sort1.c`. Improve its performance by modifying the inner loop, which shifts data elements to make room for insertions. You may want to try using a sentinel test, as described in lecture; this will reduce the number of operations necessary to test the loop bound. *Hint:* You may wind up needing to duplicate the body of the loop.

2.2

We have provided a simple implementation of quick sort in `sort2.c`. Improve its performance. At the very least, you should implement the following ideas.

- Hybridize the algorithm by switching to insertion sort when the size of the array is small. Determine and report the size at which it is best to make this switch.
- Optionally, try switching to other sorting methods. (You saw several interesting options in Project 2.1.)
- Eliminate one of the recursive calls by introducing a loop that allows quick sort to continue sorting one of the two subarrays within the same function invocation. Which of the two arrays should be passed to the recursive call? Why?
- Quick sort's worst-case $O(n^2)$ time complexity occurs when it sorts arrays which are already sorted. Can you protect yourself from this case by changing how you select a pivot? You might also consider looking at more than one possible candidate and selecting your pivot from among them. (Your solution will still be prone to the worst-case complexity, just not in the (possibly common) case that the array is already sorted.)

Comment on the effect of the optimizations and the performance improvements you see. Provide data to support your claims.

2.3

In `sort3.c` you will find an empty sorting function. Implement a least significant digit radix sort. You may want to read the Wikipedia article on radix sort.

For simplicity, you can use the statically allocated 2D arrays already created for you to implement your bucket queues. Experiment with different radix sizes. Which size performs the best?

2.4

As you saw in Project 2.1, radix sort has poor cache behavior because it inserts data into each queue in an unpredictable manner. You can reduce the number of cache misses by prefetching the tail of a queue as soon as you compute the bucket that an element must be moved into. Use the prefetching function presented to you in the code to perform such prefetches.

Comment on whether you see any improvements in performance.

2.5

Compare and contrast the performance of each of your sorting algorithms. Can you explain why each version performs the way it does? Provide well-presented data to substantiate your claims.

Pick whichever of your sorting programs you would like us to time, copy it to `sort.c` and add the `sort` target to the Makefile. We'll test all four of these programs for correctness, but only the one you select will be tested for performance.

Please note that the sorting program you submit for performance testing does not have to be one of the three described above. You may use any and all techniques at your disposal to implement the fastest sorting algorithm possible. Several concepts covered in recent lectures will probably be of use. Good luck!

Correctness

The testbed we have given you already checks for correctness and handles timing. If you add your own output statements (for example, calls to printf) you should make sure that the correctness result, which is printed after your function returns, remains on its own line. A submission will be judged correct if it passes this correctness test for any input size n such that $n \geq 0$. You may assume that n will fit in a 32-bit signed integer.

For this assignment, we will not ask that you submit additional test cases; there is no grading component for test coverage. Still, you are highly advised to write unit tests to test individual parts of your implementation, and to test your implementation with data which is not randomly distributed. We will likely include some non-random data in the test suite used to measure your performance grade.

6.172 Performance Engineering of Software Systems

Fall 2010