CS61C Spring 2014 Project 3: Digit Recognition Parallelization

TAs: Sagar Karandikar, Roger Chen

Updates

- 4/7/14 Grading scale for 128x128x64 has been updated.
- 4/7/14 Updated Optimization Details section with more tips.
- 4/7/14 Clarification: You are not required to work in groups, but it is highly recommended.
- 4/7/14 Clarification: The 7.5 Gflop/s requirement applies separately to each matrix size (it is NOT an average across sizes).

Background

Your objective is to parallelize and optimize the digit recognition code you wrote for Project 1. We've released the autograder for Project 1 (see Piazza), so that you can confirm and fix any issues your original code encounters. If you're unsure of the operation you need to perform, please refer to the Project 1 Spec.

In order to accomodate larger images and the performance metrics we've developed in class (Gflops aka Billions of floating point operations per second), you will need to slightly modify your digit-rec code from project 1 to use single-precision floating points instead of unsigned integers for your computed distance and the input images. We'll discuss this further in the converting calc-dist.c for Project 3 section below.

Architecture

What follows is information concerning the computers you will be working on. Some of it will be useful for specific parts of the project, but other elements are there to give you a real world example of the concepts you have been learning, so don't feel forced to use all of the information provided.

You will be developing on Dell Precision T5500 Workstation. They are packing not one, but two Intel Xeon E5620 microprocessors (codename Westmere). Each has 4 cores, for a total of 8 processors, and each core runs at a clock rate of 2.40 GHz.

All caches deal with block sizes of 64 bytes. Each core has an L1 instruction and L1 data cache, both 32 Kibibytes. A core also has a unified L2 cache (same cache for instructions and data) of 256 Kibibytes. The 4 cores on a microprocessor share an L3 cache of 12 Mibibytes. The L1, L2 caches are 8-way associative, while L3 is 16-way associative.

Getting Started

For this project, you should work in groups of two (however, you may work solo). You are not allowed to show your code to other students, loved ones, pets, or any other individuals.

The Usual Disclaimers

Looking at solutions from previous semesters is also strictly prohibited. We have various tools at our disposal to detect plagiarism and they are very good at what they do.

Additionally, tampering with machines in any way to gain an unfair advantage - for example spamming other students' terminal sessions - is strictly prohibited (and considered abuse of campus resources). We can track anything that happens on your `cs61c-XX` account and you will be caught if others complain. Remember that you are responsible for anything that happens on your account: "my friend did it" is not an excuse.

Prepping Your Environment

To begin, copy the contents of the `~cs61c/proj/03` directory.

```
$ mkdir ~/proj3
$ cd ~/proj3
$ cp -r ~cs61c/proj/03/* ~/proj3
[now, copy your Project 1 calc_dist.c into ~/proj3]
```

Project Files

The following files are provided:

- `Makefile`: Defines all the compilation commands.
- `benchmark.c`: Contains the benchmarking code that measures the performance of your program. Feel free to modify this to try additional sizes.
- `calc_dist.c.example`: An example of what your `calc_dist.c` should look like.
- `digit_rec.h`: Defines template width and height and function signatures related to digit recognition.
- `digit_rec.c`: Loads bitmap images and calls `calc_min_dist()` to calculate distances.
- `oracle.h`: Header file (function signatures) for the oracle. See `oracle.o` below.
- `oracle.o`: Computes the correct output for a given image and template. Allows you to check your output results by comparing against a guaranteed working solution.
- `test_digitrec.c`: Examines the calculated distances and prints out the digit recognition result, which is the digit with shortest distance to the test image.
- `utils.h`: The `Image` struct and utility function signatures.
- `utils.c`: Bitmap loading and printing functions.
- `templates/`: Contains the template images.
- `basic_tests/`: Contains test images for basic test cases (eg. only rotated, etc).

You will replace your updated project 1 code in `calc_dist.c` with a new parallelized solution that you will write for this part of the project. **This is the only file you will be permitted to submit, so you should place all of your code here. Code in other files will be overwritten or ignored.** More about your specific task is discussed in the following section.

Please post all questions about this assignment to Piazza, or ask about them during office hours.

Part 1: Due 4/13/2014 @ 23:59:59 (50pt)

Your task is to optimize your `calc_min_dist` function. You should optimize your code for an image of size 128x128 and a 64x64 template (N=128,M=128,T=64), but your code should also work for images/templates of any size.

- `NxM` is the size of your image. N is the width and M is the height.
- `TxT` is the size of your template. Templates are always square, so only one variable is required.

Your code must compute the minimum distance as required in Project 1, however we will now be storing distance as a float to allow larger images (as discussed above). You will also need to add an appropriate `#include` directive before you can use SSE intrinsics.

You can develop on any machine that you choose, but your solution will be compiled using the given Makefile and will be graded on the Hive machines. Therefore, you should perform all of your testing on a Hive machine, since performance is critical. **If your code does not compile on Hive, you will receive no credit.**

Converting calc_dist.c for use in Project 3

First, you'll need to modify function signatures and any variables/data structures you created in `calc_dist.c` for Project 1 to use floats. You can take a look at the example `calc_dist.c.example` to determine what needs to be changed. **The type of the inputs have been changed from `unsigned int` to `float`.** You'll need to propagate these changes throughout your Project 1 code. Once you've done so, do the following to ensure that your solution still works:

```
$ make check
```

If the `make check` tests pass, you can move on to the rest of this project. Otherwise, there is either a bug in your project 1 `calc_dist.c` or you have not correctly changed from `unsigned int`s to `float`s.

For your convenience, here are all of the changes between the skeleton `calc_dist.c` from Project 1 and the updated `calc_dist.c.example` that is included in the Project 3 skeleton:

Click here to show the diff

You can also rewrite `calc_dist.c` from scratch by copying our skeleton:

```
$ cp calc_dist.c.example calc_dist.c
```

Checking Performance

To compile/run your improved program, do the following:

```
$ make
$ ./benchmark
```

You should tune your solution to maximize performance for an image size of 128x128 with a template size of 64x64. To receive full credit, your program needs to achieve a performance level of 15.5Gflops for images/templates of this size. This requirement is worth 35pt. See the table below for details.

128x128 Image, 64x64 Template Grading

Gflop/s	1	3	5	7	9	11	13	15	15.5
Point Value	1	3	6	9	13	19	25	31	35

Intermediate Gflop/s point values are linearly interpolated based on the point values of the two determined neighbors and then rounded to nearest integer.

Your code must be able to handle images/templates other than 128x128/64x64 in a reasonably efficient manner. The benchmark will also test your code on inputs of sizes `[N=512,M=511,T=510]`, `[N=384,M=360,T=355]`, and `[N=171,M=171,T=143]`. You must achieve an average of at least 7.5 Gflop/s on **EACH** of these sizes. You can assume that your code will be graded using both the 128x128x64 matrix size as well as matrix sizes that are similar to the other three listed sizes. This requirement is worth 15pt.

For this project, you are not allowed to use any optimizations not discussed below. In particular, you are not permitted to use the GPU. Remember that aligned loads are prohibited for both parts of this project. Also, attempting to optimize by changing compile-time parameters is not allowed - we will be replacing your `Makefile` with our own when we grade your project.

Checking Correctness

You should make sure that your optimized code still produces the correct output. The `./benchmark` will compare the result of your `calc_min_dist` to a known working version in `oracle.o`, but **the benchmark does not guarantee correctness**. You should check correctness with the basic tests provided in the skeleton.

```
$ make check
```

Tips

We recommend that you implement your optimizations in the following order (for more, see the Optimization Details section below):

1. Use OpenMP (see lab 8)
2. Use SSE Instructions (see lab 7)
3. Optimize loop ordering (see lab 5)
4. Implement Register Blocking (load data into a register once and then use it several times)
5. Implement Loop Unrolling (see lab 7)
6. Compiler Tricks (minor modifications to your source code can cause the compiler to produce a faster program)

Extra Credit

This project will have an extra-credit competition, which we will discuss in part 2.

Optimization Details

What follows are some useful details for Part 1 of this project.

OpenMP

Now that you are comfortable with writing parallel code in OpenMP, it's time to wield the formidable power offered to you by the machine's 8 cores. Use at least one OpenMP pragma statement to parallelize your computations.

SSE Instructions

Your code will need to use SSE instructions to get good performance on the processors you are using. Your code should definitely use the _mm_add_ps, _mm_mul_ps, _mm_loadu_ps, _mm_storeu_ps intrinsics as well as some others.

You will need to figure out how to handle images/templates whose dimensions aren't divisible by 4 (the SSE register width in 32-bit floats). To use the SSE instruction sets supported by the architecture we are on (MMX, SSE, SSE2, SSE3, SSE4.1, SSE4.2) you need to include the header `<nmmintrin.h>` in your program. You may also find the following intrinsics useful:

- the `_MM_TRANSPOSE4_PS(row0, row1, row2, row3)` macro, which takes 4 SIMD vectors (rows) and transposes them into 4 SIMD vectors (columns).
- `_mm_set_ps(float z, float y, float x, float w)` creates a SIMD vector from 4 floats
- `_mm_shuffle_ps` rearranges the floats inside a SIMD vector

Please note that you MAY NOT use aligned loads and stores (e.g. `_mm_load_ps`).

Register Blocking

Your code should re-use values once they have been loaded into registers (both MMX and regular) as much as possible. Your code should also try to use SIMD loads as much as possible, since they load 128 bits, as opposed to normal loads, which only load one 32-bit float.

Loop Unrolling

Unroll the loops in your code to reduce the amount of branches you have to make and improve pipelining of adder-subtractor and multiplier units. You can also reduce the amount of address calculation necessary in the inner loop of your code by accessing memory using constant offsets, whenever possible.

Transpose and Rotation

You may find that transposing or rotating a large matrix is very slow, especially on large template sizes like the `512x511x510`, which has a `510x510` template. This is why your `512x511x510` performance might be much slower than the performance of your other matrices. You can speed up the performance of your code on large matrices by avoiding transposing or rotating a matrix. Instead, you can achieve the same effect by accessing matrix elements in reverse order. (This is tricky to do with SIMD!)

For example, if you want to transpose a matrix, you can instead treat the matrix as column-major instead of row-major. That way, you never actually have to explicitly transpose the matrix in memory. Transposing a small matrix (like a `64x64` template) is fast, but if you want to efficiently transpose a large matrix, you must either implement cache blocking in your transpose (like in lab 5) or adopt a way of accessing array elements in a non-standard order (reversed, column-major, reversed column-major, etc).

You will want to take care of this case (large matrices) last, since it is not worth many points as part of the whole project.

You are not permitted to use the following optimizations for either part of the project:

- Aligned Loads / Stores (you are allowed to use mm_loadu, but not mm_load)
- AVX intrinsics or anything not supported by the Hive machines

Submission/Early Submission Incentive

We are offering an early submission incentive. We will give one bonus point to anyone that submits a `calc_dist.c` that compiles and yields a correct answer under the new framework by **Thursday night (4/10/2014 11:59PM)**. This early submission does not need to meet any speed requirements. See the section converting calc_dist.c for Project 3 for details. As this is for bonus points, slip days will not apply.

To submit for the early bonus, enter in the following commands. You will only need to turn in `calc_dist.c`.

```
$ cd ~/proj3
$ submit proj3-1-early
```

The full proj3-1 is due Sunday. To submit the full proj3-1, enter in the following commands. You will only need to turn in `calc_dist.c`.

```
$ cd ~/proj3
$ submit proj3-1
```

References

1. Chellappa, S., Franchetti, F., and Püschel, M. 2008.How To Write Fast Numerical Code: A Small Introduction,Lecture Notes in Computer Science 5235, 196-259.
2. Lam, M. S., Rothberg, E. E, and Wolf, M. E. 1991.The Cache Performance and Optimization of Blocked Algorithms,ASPLOS'91, 63-74.
3. Intel Instrinsics Guide (see Lab 7)
4. Intel® 64 and IA-32 Architectures Optimization Reference Manual
5. OpenMP reference sheet
6. OpenMP Reference Page from LLNL