CS61c Summer 2014 Project 2: Eigenvector Finding

Due: August 03 at 23:59:59

Our second project is to optimize a simple eigenvector finding algorithm.

1 Background

We're going to explain some mathematical background for what the project is doing. Don't fret if you don't understand all the math – as long as you understand the for-loops inside the skeleton you should be fine.

1.1 Matrix-Vector Multiplication

We say that an n element vector v is the product of a matrix A and a vector u if and only if for all $i \in [1, n]$ we have

$$v_i = \sum_{k=1}^{n} A_{ik} u_k$$

Note that this means we can write a Matrix-vector product as a pair of nested for loops.

1.2 Vector Norms

The euclidean norm of an n element vector v, written as $||v||$, is defined as

$$\sqrt{\sum_{i=1}^{n} v_i^2}$$

Once again, the computation is easily handled using for-loops.

1.3 Power Iteration and Eigenvectors

An eigenvector v of a matrix A is a nonzero vector such that for some λ the following holds

$$\lambda v = Av$$

One simple algorithm for finding an eigenvector numerically is to repeatedly "hit" a random vector with a matrix until the eigenvector with the largest eigenvalue (the λ in the above equation) becomes dominant. Such an algorithm might look something like this

```
Pick a seed vector $u_0$ at random
for k=1:max_iters,
    $v_k$ := $Au_{k-1}$
    $\mu_k$ := $||v_k||$
    $u_k$ := $v_k/\mu_k$
endfor
```

This is the algorithm which you will be optimizing, with one small modification. Instead of applying power iteration to a single seed vector at once, we'll do a batch computation and do power iteration on an array of n different seed vectors at once (Those of you with some numerical analysis/linear algrebra background might notice that this is a categorically stupid idea. Just roll with it). Our new algorithm will look something like this:

```
Pick a seed vector array $u_{0,1}$, $u_{0,2}$, ..., $u_{0,n}$ at random
for k=1:max_iters,
    for i=1:n,
        $v_{k,i} := Au_{k-1,i}$
        $\mu_{k,i} := ||v_{k,i}||$
        $u_{k,i} := v_{k,i}/\mu_{k,i}$
    endfor
endfor
```

2 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 the hive Ubuntu workstations. They are packing not one, but two Intel Xeon E5620 microprocessors (codename Gainestown). Each has 4 cores, for a total of 8 processors, and each core runs at a clock rate of 2.40 GHz.

Each core comes with 16 general purpose integer registers (though a few are reserved such as the stack pointer) and 8 floating point registers. They also have 16 XMM registers apiece for use with the SIMD intrinsics.

All caches deal with block sizes of 64 bytes. Each core has an L1 instruction and L1 data cache, both of 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 8 Mibibytes. The associativities for the L1 instruction, L1 data, unified L2, and shared L3 caches are 4-way, 8-way, 8-way, and 16-way set associative, respectively.

3 Setting Up Your Workspace

The skeletons for this semester's projects are still at `git@github.com:ucberkeley-cs61c/projects_su14.git`, so you can get a copy with the following commands

```
$ cd ~/work #Assumes you placed your working repository at ~/work
$ git remote rm projects #Only needed if you've already messed up this alias
$ git remote add projects git@github.com:ucberkeley-cs61c/projects_su14.git
$ git pull projects master
```

Once you've pulled the skeleton you should find a folder named `proj2` inside your local repository. This folder contains the skeleton for this project.

4 Running Your Solution

You may use the provided `Makefile` to build your code. Running `make` will produce two executables: `bench-fast` and `bench-naive` which will benchmark your optimized code and the naive implementation which we have provided you, respectively. E.g. If you wanted to test your code you could run

```
$ make clean #paranoia is good sometimes
$ make
$ ./bench-fast
```

5 Submission Instructions

In order to submit this project tag the commit you want to submit with the tag `proj2` and push to your git repo. For example, the following commands would submit the most recent commit in your local repository

```
$ git status #check that everything I wanted committed was committed
$ git tag proj2 -f
$ git push --tags origin master
```

Don't be afraid about submitting multiple times – you can move your tags around as often as you please, so you can choose an older commit later on if you want, or a newer commit.

There is a file named `proj2.marker` at the top of the skeleton. We will be using that file to locate your submission when grading it, so be sure not to move it. If you're worried that we won't be able to find your submission properly you are also encouraged to use the `check_submission.py` script provided in the skeleton to sanity check yourself.

We'll be testing your submissions on the hive machines (the computers in 330 SODA, not the computers in 200 SDH). In addition to testing that your submission is in the right place on GitHub, be sure that your code works in the environment in which it will be tested. "It worked on my home machine" is not a valid excuse for a broken program, and will earn you no sympathy. Additionally, we'll be grading your code using an autograder. It is particularly important that you test on the hive machines for this project, since performance optimization is much more finicky than general code portability.

6 Grading

In order to test your solutions we will disable remote logins to a couple of hive machines and then run our benchmark code on your solution to measure performance. If you compute incorrect results for a test case then it will be treated as 0 GFlop/s for grading purposes. Full credit will be awarded to any legal solution achieving at least 30 GFlop/s average performance. Solutions which do not achieve a full 30 GFlop/s will be granted partial credit in proportion to how close to 30 GFlop/s they come (e.g. a 15 GFlop/s solution would receive 50% credit).

Note that, as always, there is no guarantee that we will exhaustively test every possible size within this set. Just as in any project, there are a finite number of test cases that are exercised in grading; for better or for worse, your score must reflect the average performance obtained with the set of sizes chosen for grading. Therefore, the only way to guarantee that you make the average performance specification is to ensure that no matrix size results in a performance noticably slower than the required average; relying on specific sizes that result in faster performance to boost your average will not work (at least not reliably). We will try strange matrix sizes, such of powers-of-2 and primes, so be prepared. The set of sizes listed in `benchmark.c` is not the final list to be used for grading.

7 Partners

You are free to complete this project with one partner. Make sure that only one of you has a submission on file come the end of the project though – we will not guarantee which of your submissions will be graded if you both have submissions in the system.

8 Optimization Techniques

You are free to apply nearly any optimizations you see fit to achieve speedup, but we do require that you be responsible for writing the the actual optimized code (e.g. you may not use any BLAS libraries, even if you find a way to make them applicable), and that you implement the algorithm presented to you (i.e. you're not allowed to make algorithmic improvements to cut flops). We've enumerated a couple of different paths which you might wish to consider in order to help you to get started with your optimization.

8.1 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`, and `_mm_loadu_ps` as well as some subset of: `_mm_load1_ps`, `_mm_load_ss`, `_mm_storeu_ps`, `_mm_store_ss`, `_mm_shuffle_ps`, `_mm_hadd_ps`

Depending on how you choose to arrange data in registers and structure your computation, you may not need to use all of these intrinsics. There are multiple ways to implement power iteration using SSE instructions that perform acceptably well. You probably don't want to use some of the newer SSE instructions, such as those that calculate dot products, though you are welcome to try. You will need to figure out cases for which n is not a multiple of 4, either by using fringe cases ore by padding hte matrices.

To use the SSE instruction sets supported by the architecture used on the hive machines you need to include the header `<nmmintrin.h>` in your program. OpenMP requires use of `<omp.h>`.

8.2 Register Blocking

Your code should re-use values once they have been loaded into registers (both MMX and regular) as much as possible. By reusing values from in multiple calculations you can reduce the total number of times each value is loaded from memory in the course of your computation. To ensure that a value gets loaded into a register and reused instead of being loaded from memory repeatedly, you should assign it to a local variable and refer to it using that variable.

8.3 Loop Unrolling

You may find it helpful to unroll the inner loop of you code to expose opportunities for SIMD-izing as well as to reduce the overhead of address calculation and loop condition checking. You can reduce the amount of address calculation necessary in the inner loop of your code by accessing memory using constant offsets, whenever possible.

8.4 Padding Matrices

As mentioned previously it is sometimes a good idea to pad your matrices to avoid having to deal with fringe cases. This implies copying the entire matrix into a re-formatted buffer with no fringes (e.g. a 3x4 matrix into a 4x4 buffer). You should allocate this buffer on the heap if you decide to pad matrices. Be sure to fill the padded elements with zeroes.

8.5 Cache Blocking

Cache blocking and loop re-ordering can be used to reduce the number of misses in your cache by improving the locality (spatial and temporal) with which accesses are made.

8.6 OpenMP

Once you've produced fast serial code you will want to take advantage of the 8 cores available on the hive machines. You'll want to use at least one openMP pragma to parallelize your computations.

8.7 Miscellaneous Tips

You may also wish to try copying small blocks of your matrix into contiguous chunks of memory or taking the transpose of the matrix as a precomputation in order to improve spatial locality.

8.8 Aligned Loads

We haven't forbidden the use of aligned loads, but make sure that if you do use them that you can guarantee that the address being used is in fact aligned properly. We won't be very forgiving if you don't and your program crashes when we're benchmarking it.

9 Change Log

Any changes to the project specification will be listed here, for your convenience.