

Programming Assignment #1

Optimization of Matrix Multiplication

Due Date: Wednesday, October 29, 2008 at 5:00PM PST

For this assignment, you will use your knowledge of system architecture to improve the performance of a matrix multiplication kernel. Starting with a naïve implementation, you will use feedback from performance analysis tools to direct your work. You will learn to apply compiler tuning, code optimization, and your knowledge of memory hierarchies towards improving the performance of this important application.

SYNOPSIS

Your group's assignment is to take the application stub that we provide and reimplement its matrix multiplication kernel to maximize performance (minimize runtime). Your optimized kernel should maximize performance for all matrix sizes. This might entail you coding several implementations, each tuned for a different matrix size. After implementing your optimized matrix multiply, you will write a short report including:

1. A qualitative description of the optimizations you employed.
2. Quantitative data demonstrating the effectiveness of your optimizations.

DEVELOPMENT ENVIRONMENT

You will be coding your assignment under Linux. We will be grading your assignment under the CentOS 5.2 distribution on the Hotbox cluster of 2.33 GHz Intel Xeon 5345's. You are allowed to use any compiler we provide on the cluster.

The Intel Xeon E5345 is a quad-core chip, composed by packaging two dual-core Woodcrest (Core 2 Duo family) dice in a single multi-chip module. Each core has a 32 KB L1 data cache, and each die has 4 MB of unified L2 cache (for a total of 8 MB of L2 cache per chip). Each cluster node is populated with 32 GB of FB-DIMM memory (composed of 667 MHz DDR-II chips).

We have created accounts for every student in the class. Your username and password are the same as your SUNet account.

GETTING STARTED

As soon as possible, ssh into *soda.stanford.edu* (also known as hotbox-1) to ensure that your account is active.

NOTE TO OFF-CAMPUS STUDENTS: soda is not directly accessible off-campus. You will have to first log into Leland (e.g. elaine, bramble, myth, etc.) in order to ssh to soda. Alternatively, you can install the Stanford VPN client¹ to access soda directly.

We provide a simple Makefile and application stub, including a naïve implementation of matrix multiply. Unpack this into your home directory by typing:

```
cd
wget http://ee282.stanford.edu/pa1.tar.gz
tar zxvf pa1.tar.gz
cd pa1
```

Type "make" to build the application. Type ". /matmul" to run the application. Type ". /matmul -c" to have the driver check your matmul () routine for correctness. You may run matmul on soda (hotbox-1) only ONCE, after which you must start submitting it to the cluster. In the "Using the Cluster" section of this handout, we describe how to run matmul on the cluster.

The matmul application is already instrumented using PAPI-C, a library that facilitates direct access to the built-in performance counters in the Intel Xeon. It currently measures the instruction count and cycle count. You are free to use any other PAPI counters² you find helpful. Measurements can only be taken from the performance counters on machines with kernel support for them (i.e. the cluster, not Leland). Type "papi_avail" and "papi_native_avail" to see a list of available counters. Edit driver.c to configure the performance counters.

The source file you are to edit is "matmul.c". The interface you must implement is:

```
void matmul(int N, const double *A, const double *B, double *C);
```

N is the dimension of the matrices. You may assume that the matrices are square, and that N is a power of 2, ranging from 2 to 1024. A and B hold the matrices you will be multiplying, stored in row-major order. You are not allowed to modify the contents held in A and B. C is the output matrix, and is also stored in row-major order. The driver will run your matmul () function several times to make precise measurements of its performance. Do not initialize C each time; just accumulate the result of each multiplication into it (like the naïve implementation does).

You may edit driver.c to add additional performance measurements, but you may *not* optimize it. We will be using our own driver.c to grade your performance.

¹ Available at <https://www.stanford.edu/services/vpn/>

² See documentation at http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI_USER_GUIDE.htm

OPTIMIZATION STRATEGIES

At the end of this document is the result of one of the better submissions we received the last time we assigned this problem. You can expect to receive a high grade if you achieve similar performance. Your `matmul()` should be as fast as possible for **all** matrix sizes, not just 2x2 or 32x32. We calculate MFLOPS as follows:

$$\text{Average FLOPS} = 2 * N^3 / (\text{average runtime})$$

You have a number of optimizations that you may try. You will likely achieve maximum performance through a clever combination of several strategies. Here are some example strategies:

1. **Blocking.** Your optimized matrix multiply ought to be able to make exquisite use of the Xeon's large L1 and L2 caches.
2. **Compiler optimizations.** Enable as many compiler options as you wish! You may even attempt to use auto-vectorization with GCC 4.1.
3. **Hand-coding.** Sometimes, the compiler will not perform code optimizations as aggressively as you could by hand. Consequently, you will want to consider changing the `matmul()` code to improve performance. This could include, among other things, rearranging code sequences, manually unrolling loops, or software pipelining the loops. You may even use compiler hints in the code, such as `register`, `__restrict`, `__builtin_prefetch`, and SIMD intrinsics.

Note that some optimizations may result in lower performance. It is your job to figure out what does and does not work.

The following “optimizations” are not allowed. You will get no credit for this assignment if we find violations.

1. You are NOT allowed to use any compiler flags that relax IEEE floating-point compliance (e.g. `-ffast-math`).
2. You are NOT allowed to call any library routines such as those from the Intel Math Kernel Library (LAPACK, BLAS etc.). However, feel free to study the documentation for these libraries and "borrow" some of the ideas.
3. You are not allowed to do cross file “optimizations,” i.e. trying to take advantage of how `driver.c` works. The `driver.c` file we use in the grading script will be different. So if your submission relies on “features” of `driver.c` beyond the interface, your code may not even compile when we grade your submission.

USING THE CLUSTER

In order to have dependable performance measurements, your program needs to run in isolation. The Hotbox Cluster runs Torque & Maui, a job queuing system which allows you to dispatch jobs to unladen compute nodes. If no nodes are available, it will hold your job and run it as soon as a node becomes available.

To submit a job to the cluster, type:

```
jsub -- <command>
```

`jsub` will enqueue the job to be run on the cluster and immediately return you to the command line so that you can continue working.

For example, to run `matmul`, type:

```
jsub -- ./matmul
```

`jsub` will confirm that you submitted the job to the cluster by printing out a line that looks like:

```
XXXXX.cyclades-master.tendot.stanford.edu
```

When your job completes, it will create a file called `matmul.oXXXXX` that contains the standard output from `matmul` and a file called `matmul.eXXXXX` that contains the standard error. If you want to watch your job run instead of letting it spawn asynchronously, pass the `-I` flag to `jsub`:

```
jsub -I -- ./matmul
```

Since several people will be using the cluster, it might be helpful to occasionally look at the job queue to see how many jobs are waiting to run. Use “`showq`” to see the current contents of the job queue. Use “`diagnose -n`” to see the current status of the cluster nodes.

If you submit a job on accident or a job never terminates, use “`showq`” to figure out the job's jobname, and “`qdel <jobname>`” to terminate the job.

You may find it useful to use CVS or Subversion to coordinate development between your group members and machines.

Usage restrictions:

1. A job may run for at most 5 minutes. If it runs for longer, it will be automatically terminated.
2. Each user can only run a limited number of jobs simultaneously.
3. Do not run “`./matmul`” on `hotbox-1`!!! Always submit it to a cluster node. `hotbox-1` can get crowded very quickly.

If the cluster crashes or becomes inoperable for any reason, please send e-mail to ee282-aut0809-staff@lists.stanford.edu ASAP. For other issues, please use the bulletin board.

SUBMISSION

The final submission will be electronic. You must submit your code (“`matmul.c`”), your `Makefile` (with your compiler settings), a `GROUP` file, and your written report (as a PDF named

“report.pdf”). Note that you do not submit `driver.c`; we will be grading with our own `driver.c`. Make **sure** you check the correctness of your application (by running “`./matmul -c`”) before turning in your assignment.

The `GROUP` file (named “`GROUP`”) will list the SUNet IDs of your group’s members, one per line. This is the login name that you use for Axxess and other Stanford web sites, and not your numeric ID number.

The report should be short (3-4 pages), and include:

1. The performance data you collected on your best implementation (runtimes and MFLOPS for each matrix size). This should be a “copy and paste” from the output obtained from running your code on the cluster.
2. A table summarizing each of your matrix optimizations. Your table should include:
 - a. the optimization attempted
 - b. the performance gain or loss due to this optimization (could vary across matrix sizes!)
 - c. whether you decided to include this optimization in your final submission
3. A plot of performance (MFLOPS) versus matrix size, for the naïve and optimized cases.
4. A qualitative description of all of the optimizations you employed. Additionally, answer the following questions:
 - a. What optimizations had the biggest impact on performance?
 - b. What optimizations did you try that did not improve performance?
 - c. What effect did matrix size have on your optimizations?
5. A list of the compiler flags you tried and their impact (positive or negative) on performance.
6. Any additional comments.

You are to submit your assignment on the *Leland* cluster (elaine, myth, bramble, hedge, etc.).

First, copy your assignment, report, and `GROUP` file to your Leland account. For example, if you have all of the necessary files in the subdirectory “`pal`”, type:

```
cd
scp -r pal myth:
```

Then log into Leland and use the EE282 submit script:

```
ssh myth
cd pal
/afs/ir/class/ee282/bin/submit
```

Follow the instructions in the submit script. If all goes well, you will get a “SUCCESS” message. You may submit your assignment multiple times before the deadline. We will grade the last submission. If you have problems, please post a message on the bulletin board or send e-mail to us.

ADDITIONAL RESOURCES

1. Google!
2. GCC 4.1 Docs: <http://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/>
3. PAPI User Guide:
http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI_USER_GUIDE.htm

BASELINE OUTPUT

Example output when running `matmul()` on the cluster, compiled with GCC-4.1.2:

```
$ ./matmul
```

Each measurement is average per iteration. Runtime is given in milliseconds.
MFLOPS is estimated assuming a naive `matmul()`.

Dim.	MFLOPS	Runtime	Tot. Instr.	Tot. Cycles
2	156.764	0.0001	359	237
4	183.853	0.0007	2445	1617
8	192.903	0.0053	18593	12334
16	192.972	0.0425	145833	98621
32	197.477	0.3319	1156409	770829
64	190.723	2.7489	9212509	6380057
128	168.152	24.9436	73548983	57944531
256	156.236	214.7673	587794949	498740503
512	145.162	1849.2188	4699986314	4295919875
1024	44.185	48602.6110	37590459286	112942694149

OPTIMIZED OUTPUT

This is the output from one of the better submissions from the last time this problem was assigned. We took their code (which had been optimized for an Opteron 244) and ran it on one of the Xeon 5345s. *You may be able to do better than this by optimizing your code specifically for the Xeon 5345.*

```
$ ./matmul
```

Each measurement is average per iteration. Runtime is given in milliseconds.
MFLOPS is estimated assuming a naive `matmul()`.

Dim.	MFLOPS	Runtime	Tot. Instr.	Tot. Cycles
2	1550.629	0.0000	39	24
4	2532.046	0.0001	154	118
8	2632.123	0.0004	1017	901
16	3923.703	0.0021	11808	4812
32	5279.931	0.0124	72353	29047
64	5541.347	0.0946	506867	221005
128	5693.083	0.7367	3827673	1703692
256	5471.664	6.1324	29958488	14205693
512	5253.932	51.0923	236149461	118797783
1024	5056.051	424.7354	1882180692	986745646