CS 61C Spring 2013 Project 3: Matrix Decryption

TA: Sung Roa Yoon

Updates

- 4/9/13: Added an additional picture to help explain the multiplication process.
- 4/15/13: Modified the benchmark to the new benchmark with a bug resolved and upgraded for new features.
- 4/21/13: Project3-2 extended to 4/22/13 11:59pm.

Background (If this is confusing, read the next part! The picture helps though!)

The Mediocre University CS department decided that they should try making their own form of complicated security system, and made encrypted matrix data stream to send to the students.

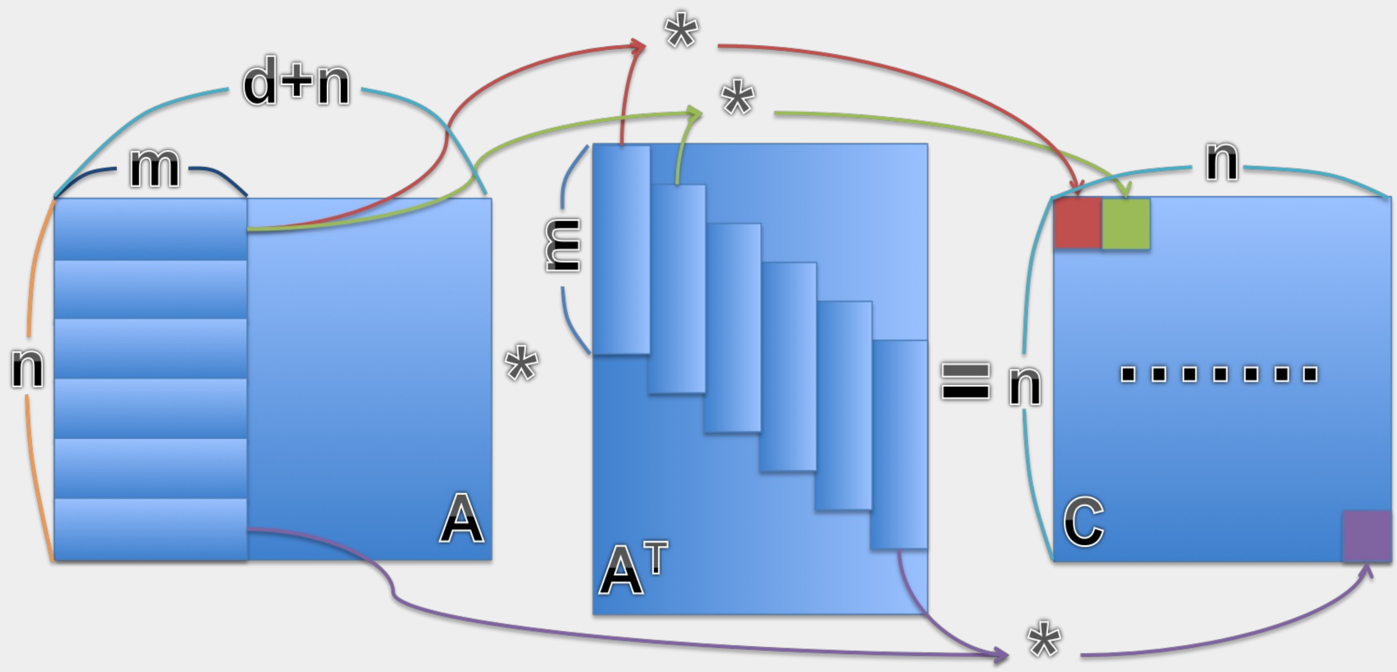
The students can decode these encrypted matrix data into readable decrypted matrix data, by knowing the size of the decrypted matrix and the size of the data strip, as well as the size of the padding (to hide how large the data strip is). The students use these information to find the rectangular chunk of the matrix and the diagonal chunk of the matrix.

If the outsider does not know the size of the strip or size of the decrypted matrix, the outsider cannot obtain the correct chunks of data to actually decrypt this message, as the outsider will not be able to determine the size of the decrypted matrix or the size of the strip.

To look at it another way, the messages are decrypted by getting correct strips of information from the encrypted matrix and multiplying them like normal matrices.

However, due to the nature of these gigantic matrix messages, it can take a really long time to decrypt any non-trivial message. Therefore, the faculty of the University hired you to build a faster decrypter, given the length of the sides of encripted and decrypted matrix, as well as the decrypted matrix itself. (As the students SHOULD have all of these given information)

The matrix is column major. The following diagram labels the matrices with their number of rows and columns, and represents a good way to visualize the problem:



Here are overview of all of the variables again:

- n = the height of the matrix
- m = size of the data strip on a given line
- d = padding of the matrix, will always be larger than or equal to m

Your objective is to parallelize and optimize the matrix decryption as described above!!! Awesome, right?

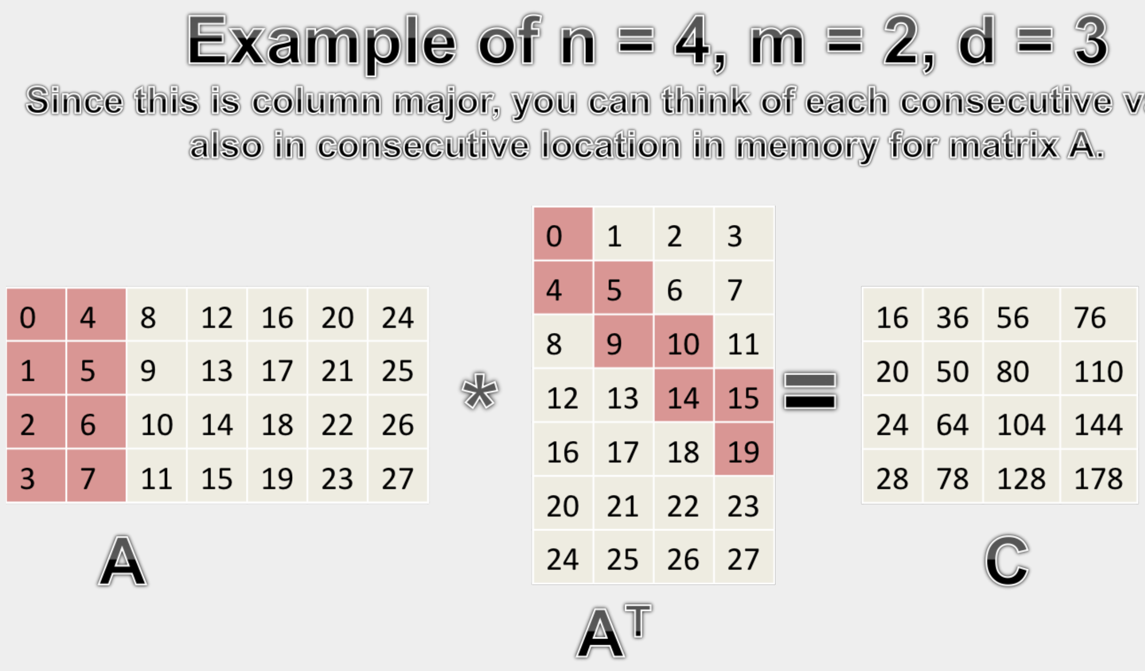
What you are supposed to do (without the story)

Essentially, what you are doing in this project is optimizing this unique matrix multiplication process, that only multiplies a specific subset of the matrix.

```
void sgemm( int m, int n, int d, float *A, float *C )
{
    for( int i = 0; i < n; i++ )
        for( int k = 0; k < m; k++ )
            for( int j = 0; j < n; j++ )
                C[i+j*n] += A[i+k*(n)] * A[j*(n+1)+k*(n)];
}
```

Where m is the size of the strip you are multiplying by, n is the height of the matrix, A is the encrypted matrix, and C is the matrix you put the decryption into. You DON'T need to come up with your own naive formula, just try your best to optimize this one!

d is the padding of the matrix in the end, but it is not really necessary for decrypting the matrix.



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 will be required to work in groups of two. You are not allowed to show your code to other students. You are to write and debug your own code in groups of 2. Looking at solutions from previous semesters is also strictly prohibited.

To begin, copy the contents of `~cs61c/proj/03` to a suitable location in your home directory. The following files are provided:

- Makefile: allows you to generate the required files using make
- benchmark.c: runs a basic performance and correctness test using a random matrix
- sgemm-naive.c: a sample program that multiplies a matrix by its transpose in an inefficient manner

You can compile the sample code by running `make bench-naive` in the project directory. You do not need to modify or submit any of the provided files. Instead, you will be submitting two new files called sgemm-small.c (for part 1) and sgemm-openmp.c (for part 2).

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)
- Optimizations specific to the fact that the product is of the form AA', like trying to reduce the actual number of squares you must compute. (because our benchmark program is set to judge Gflop/s peformance under that

assumption)

Please post all questions about this assignment to Piazza, or ask about them during office hours. Extra office hours will be held for the project.

Part 1: Due Sunday, April 14, 2013 at 11:59PM (55pt)

Optimize a single precision AA' matrix decrypt for a matrix of n = 40 and m = 48. Place your solution in a file called sgemm-small.c. You can create a template file by typing `cp sgemm-naive.c sgemm-small.c`. You will also need to add an appropriate `#include` directive before you can use SSE intrinsics. To compile your improved program, type `make bench-small`. 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. When you are done, type `submit proj3-1`.

Tune your solution for matrices of n = 40 and m = 48. To receive full credit, your program needs to achieve a performance level of 11.5Gflop/s for matrices of this size. This requirement is worth 35pt. See the table below for details.

Your code must be able to handle small matrices of size other than n = 40 and m = 48 in a reasonably efficient manner. Your code will be tested on matrices with m=[32,100] and n=[32,100], and you will lose points if it fails to achieve an average of 5Gflop/s on such inputs. This requirement is worth 20pt.

For this part, you are not allowed to use any other optimizations. In particular, you are not permitted to use OpenMP directives. Cache blocking will not help you at this stage, because n = 40 and m = 48 matrices fit entirely in the cache anyway (along with all other sizes we are testing in Part 1). Remember that aligned loads and AA' optimizations are prohibited for both parts.

We recommend that you implement your optimizations in the following order:

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

n=40, m=48 Matrix Grading

Gflop/s	2	3	4	5	6	7	8	9	10	11	11.5
Point Value	1	4	7	10	15	20	25	30	32	34	35

Intermediate Glop/s point values are linearly interpolated based on the point values of the two determined neighbors and then rounded to nearest integer. Rounding modes are to be determined. Note that heavy emphasis is placed on the middle Gflop/s, with the last couple being worth very few points. So don't sweat it if you are having trouble squeezing those last few floating point operations out of the machine!

Part 2: Due Monday, April 22, 2013 at 11:59PM (55pt)

The Objective

Now that you have proved your valor on the eternal battlefield that is optimization it is time to test your mettle on a new beast of a problem. The stakes have increased but so has your arsenal. We will be dealing with much larger n from here on out, and in order to work with them efficiently you will need to use cache blocking. You will also need to parallelize your computation across multiple cores in order to meet the required goals.

Submission and Grading

Place your solution in a file called sgemm-openmp.c. Test it using `make bench-openmp`. Submit using `submit proj3-2`.

For Part 2, we will grade your average performance over a range of large matrices with n=[400,1500] and m=[32,100]. We will try strange matrix sizes, such of powers-of-2 and primes...so be prepared. Any valid solution that achieves an average performance of 45Gflop/s without using prohibited optimizations (aligned loads and stores, the fact that we are doing AA') will receive full credit (55pt).

Matrix Range Grading

Average Gflop/s	15	20	25	30	35	40	45
Point Value	0	10	20	30	40	50	55

Extra Credit

For every 2 Gflop/s the average is above the 45 Gflop/s goal, one point of extra credit will awarded. This value is rounded down. So if you get 50 Gflop/s you get a total of 57/55 points for Part 2.

Optimization Details

What follows are some useful details for Part 1 and Part 2 of this project.

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 intrinsics as well as some subset of:

_mm_load1_ps, _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 (such as _mm_hadd_ps or _mm_shuffle_ps). There are multiple ways to implement matrix multiply 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 how to handle matrices for which N isn't divisible by 4 (the SSE register width in 32-bit floats), either by using fringes cases (which will probably need optimizing) or by padding the matrices.

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.

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 loaded from the A or B matrices in multiple calculations, you can reduce the total number of times each value is loaded from memory in the course of computing all the entries in C. 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.

Loop Unrolling

Unroll the inner loop of your code to improve your utilization of the pipelined SSE multiplier and adder units, and also 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.

Padding Matrices (NOT THE SAME AS SECURITY PADDING DONE WITH D)

As mentioned previously it is generally a good idea to pad your matrices to avoid having to deal with the fringe cases. This implies copying the entire matrix into a re-formatted buffer with no fringes (e.g. copy 3x4 matrice into 4x4 buffer). Be sure to fill the padded elements with zeros.

Cache Blocking

The first step is to implement cache blocking, i.e. loading chunks of the matrices and doing all the necessary work on them before moving on to the next chunks. The cache sizes listed in the Architecture section should come in handy here.

OpenMP

Once you have code that performs well for large matrices, it will be 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.

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 in a precompute step in order to improve spatial locality.

References

1. Goto, K., and van de Geijn, R. A. 2008.Anatomy of High-Performance Matrix Multiplication,ACM Transactions on Mathematical Software 34, 3, Article 12.
2. 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.
3. Bilmes, et al.The PHiPAC (Portable High Performance ANSI C) Page for BLAS3 Compatible Fast Matrix Matrix Multiply.
4. Lam, M. S., Rothberg, E. E, and Wolf, M. E. 1991.The Cache Performance and Optimization of Blocked Algorithms,ASPLOS'91, 63–74.
5. Intel Instrinsics Guide (see Lab 7)
6. Intel® 64 and IA-32 Architectures Optimization Reference Manual
7. OpenMP reference sheet
8. OpenMP Reference Page from LLNL