CS 267: HOMEWORK #1
MATRIX MULTIPLICATION OPTIMIZATION

Professor James Demmel
April 3, 2017

Sayna Ebrahimi
Fotis Iliopoulos
Xingjie Pan
University of California, Berkeley

Introduction

The goal of this homework assignment was to improve the performance of sequential matrix multiplication algorithm using different techniques taught in class and compare the improvement upon each or a combination of them. Our final code gave us a %**59** using the methods listed below:

- Loop unrolling
- Copy optimization
- Padding
- Changing loop order
- Two level cache blocking
- Register blocking (4-by-4 SSE intrinsic multiplication)
- Compiler optimization flags (`-O3` and `-march=cpu-type`)

Details of each optimization technique is discussed in details in the following sections.

Automatic optimization flags

One of the first things that we tried was using different compiler flags for an automatic optimization of the blocked matrix multiplication (block size is chosen as 108 for consistency and please note that a full discussion on tuning cache block size will be given later). Figure 7 shows the result of each flag.

- `-O1`: This is the original code which has an average peak percentage of %8.
- `-O3`: This flag turns on all optimization flags specified by -O2 and also enables some automatic function in-lining and auto-vectorizing. It gave an average of %14. So we kept this flag throughout the next steps.
- `-funroll-loops`: Before applying a manual *loop unrolling* -to be discussed later- we used this flag along with `-O3` and saw that it actually dropped the performance by a very small amount (average of %0.5). Even though, loop unrolling is supposed to increase the performance, We think this is because compilers probably concentrate on inner loops more and that is why a manual unrolling will be a better idea.
- `-march=cpu-type`: This flag has to be modified depending on what type of processors we are using and we made use of it when we applied the SSE optimization technique (see Section). For Edison at NERSC we used `-march=ivybridge` along with the required extensions of `-sse, -sse2, -sse3`.

Changing order of loops

Instead of the regular algorithm which takes a row of matrix A and multiply it by a column of B, we transposed B and took the dot product of rows of A with rows of B. In other words we are using:

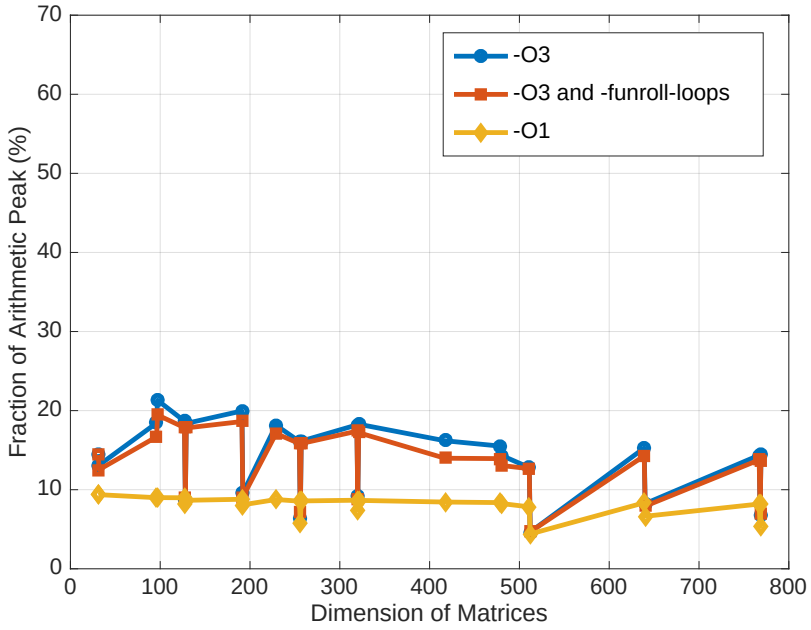

Figure 1: Comparison between different levels of automatic optimization

$\sum_{k=0}^{N} = A_{ik}B_{jk}^{T}$. This is supposed to reduce cache misses when matrices are stored in a column major format because C is a row-major format language and by each call on elements of B, we can take advantage of spatial locality. This method by itself did not increase the performance much (if not at all) as the average peak is %14 but when combined with SSE in register blocking, it made a large influence. We think the reason for that is due to the fact that B is still stored in column major anyway and therefore, the cache overhead will be high whereas in the SSE where matrices A and B will be read and stored in fast memory through one call, this method will be able to help.

Layout of Blocks

To take advantage of spatial locality, we changed the layout of matrices before doing blocked matrix multiplication $A * B$. Elements in each block are saved contiguously. Blocks and elements of the matrix A are saved in row major and B in column major. This increased about %13 performance compared to the simple blocked method.

Manual Loop Unrolling

Calculations of the innermost loop of matrix multiplication are not independent, while those of the second innermost loop are independent. Manually unrolling the second inner most loop makes the compiler notice these parallelable computations. Manual loop unrolling increased about %13 performance to solely changing the layout of matrices.

However, this technique is not compatible with manual vectorization via SSE since they all rely

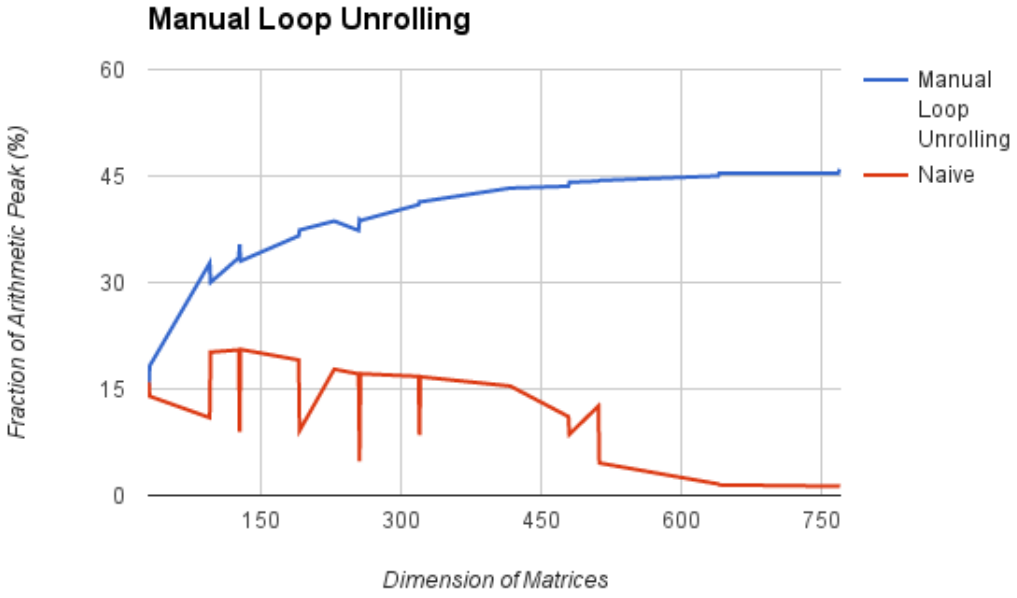

Figure 2: Performance of manual loop unrolling combined with blocking.

on the same SIMD hardware. When we tried to combine these two methods, the performance even dropped about %10.

Recursive Layout

The problem of matrix multiplification $A * B$ could be reduced into multiplification of their submatrices:

$$A * B = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} * B_{11} + A_{12} * B_{21} & A_{11} * B_{12} + A_{12} * B_{22} \\ A_{21} * B_{11} + A_{22} * B_{21} & A_{21} * B_{12} + A_{22} * B_{22} \end{bmatrix}$$

If the dimension of a matrix is a factor of two, we can solve the problem recursively by deviding matrices into equal-sized submatrices. Recursive layout is a technique to save a matix such that elements of each submatrix of any recursive level are saved contiguously. Then we can take advantage of spacial locality without knowing sizes of caches.

We rearranged matrices with recursive layout and multiplied matices recursively. However, the performance was not satisfying because of two major problems.

The first problem was that we required submatrices at each level to be equal-sized, which means we had to append zeros to matrices whose dimension were not a factor of two and the performance dipped at points where matrix sizes were factors of two. This problem could be solved by allowing submatrices to have different sizes.

The second problem was that the overhead of calling recursive functions was high. Each time

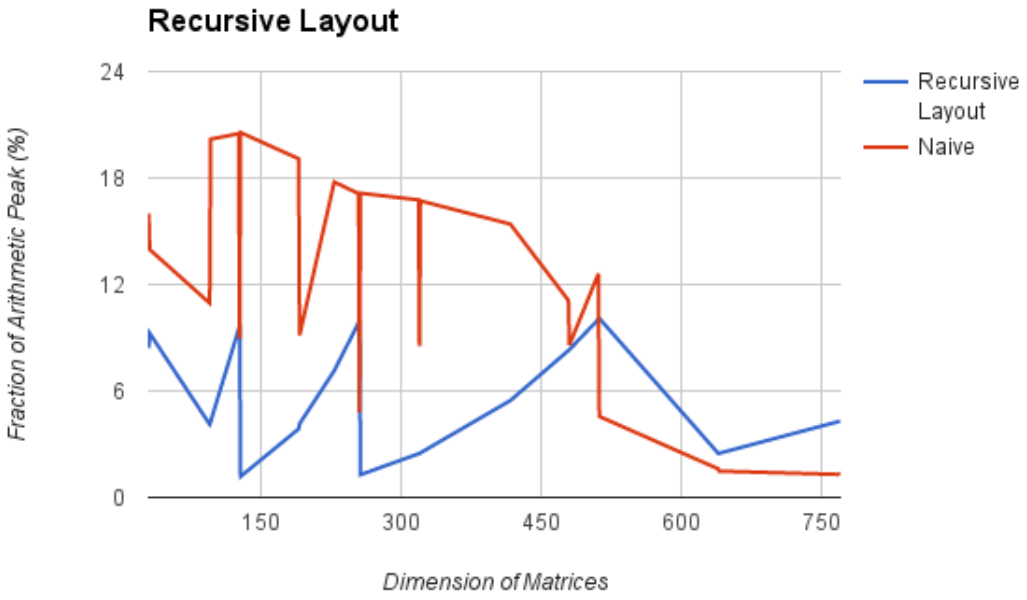

Figure 3: Performance of recursive layout

a function was called, its arguments should be pushed into the stack which slowed the program. Therefore we could only get about 10% performance even for matrices whose dimensions are factors of two.

Register Blocking

SIMD or Single Instruction, Multiple Data streams, engages the computer architecture such that it deals with multiple data streams simultaneously by a single instruction. This can be very advantageous in applications such as matrix-vector/matrix-matrix multiplication where the same arithmetic operation has to be executed on a big data stream. Different CPUs support different types of SIMD extensions such as SSE, SSE2, SSE3, SSE4, AVX, AVX2, or MMX. We have chosen SSE which offers SIMD instruction sets for XMM registers. For 64 bit system, there are 16 XMM registers (xmm0-xmm15) in the CPUs, and an XMM register has 128 bits (16 bytes) which can process two double precision floating point numbers simultaneously. With the aid of the references [1, 2, 3, 4, 5] we could implement the SSE programming.

Based on computer architecture of NERSC's supercomputer and suggested in [1], we picked 4-by-4 SSE multiplication in order to use all the available SSE registers as much as possible (we used 14 registers). Loading and storing the data of matrices A and B is performed as *unaligned* simply because we might have odd dimensions for them which is taken care by *padding* method that adds zeros to the boundary to convert the matrices into the block-size format. In this step, we did register blocking along with one level of cache blocking only to see the pure performance increase due to

register blocking. We tuned the L1 cache block size as best as we could (132). As a consequence we got 56% of performance in the average peak speed. We believed this was the most important change we could make to optimize the results and as the memory hierarchy consists of two or more levels, we tried to optimize for 2 levels which is discussed in the next section.

Cache Blocking

Performing Matrix Multiplication by breaking up the two matrices in blocks (tiles) is a way to increase computational intensity by essentially decreasing the amount of memory traffic between the different levels of memory. In L1-cache blocking, the idea is to separate the matrices into tiles so that each block by block multiplication can be done by accessing almost exclusively the L1-cache. That is, we are aiming to minimize the communication with higher levels in the memory hierarchy during the computation of each new block. Similarly, in L2-cache blocking we apply the same idea twice. That is, we tile the matrix in to (bigger) blocks, and recursively (for one step) apply block matrix multiplication within each block. The idea is that the size of big blocks should be about the size of L2 cache, while the size of the small blocks should be about the size of L1-cache.

We applied the L1-cache blocking idea, after we searched for the optimal block size (on Edison that was 108), and got a speed up of about 8%. We also applied the L2-cache blocking idea, to get another 2% speed up (after we searched for the optimal block sizes which were 400 and 7500 for L1-cache and L2-cache, respectively). While the latter is not a big increase, we noticed that for matrices of big dimension the increase can reach up to 4-5%. This is expected because for matrices of small size, performing the multiplication by tiling them into blocks doesn't really make sense (and only introduces an overhead) since, for example, they might already fit into the cache. This also explains the dip in the results for small size matrices.

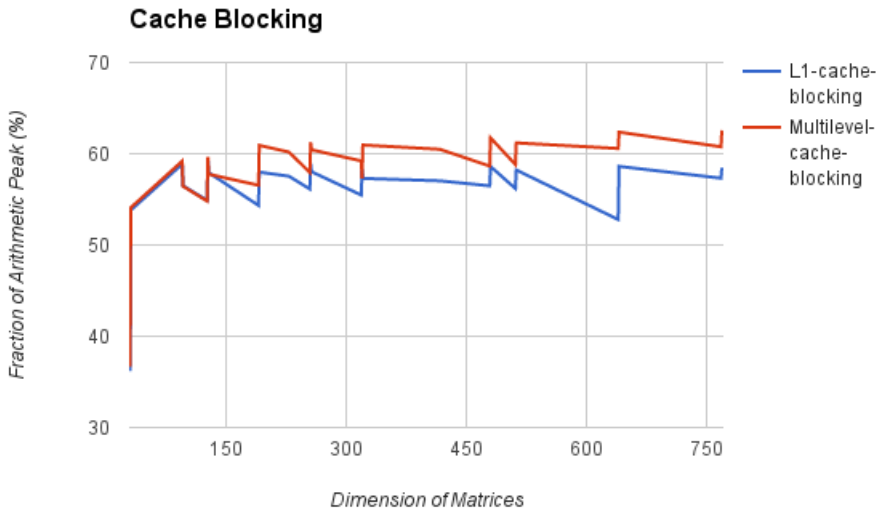

Figure 4: Comparison between levels of blocking

Results

In this section we discuss the overall increase on the performance we got on both Edison and our laptops. Here is again a summary of all the methods we used in our final optimized code:

- Loop unrolling
- Copy optimization
- Padding
- Changing loop order
- Two level cache blocking
- Register blocking (4-by-4 SSE intrinsic multiplication)
- Compiler optimization flags (`-O3` and `-march=cpu-type`)

Performance on Edison

Combining the optimization techniques described above we managed to get a %**59** average percent increase in the performance of matrix multiplication on Edison, that goes up to *63* % for matrices of big size. Figure 5 shows the percentage of average of peak speed for our final code compared to naive and naive-blocked algorithms. The overall performance is also shown for different matrix dimensions in Fig. 6 in which dips are less visible in our final code. There is a jump increase from matrix of size 31 to

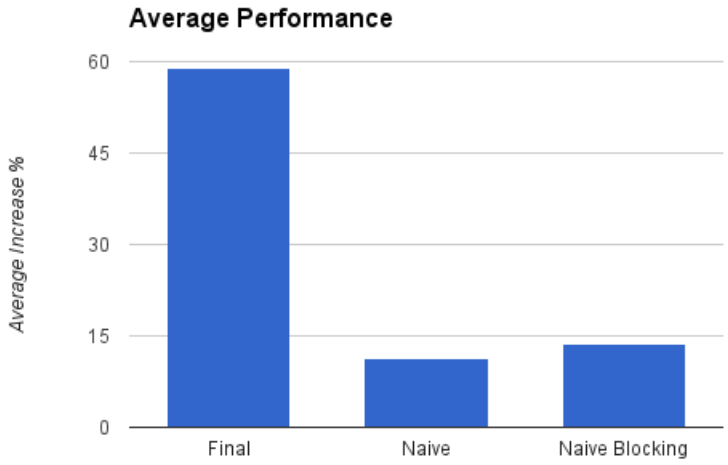

Figure 5: Average of peak performance on Edison

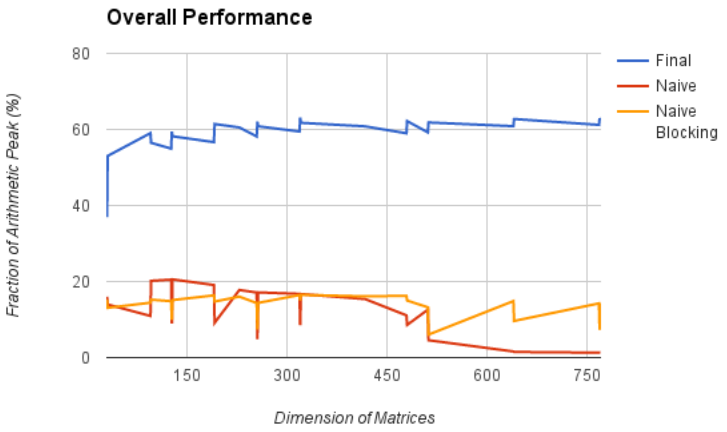

Figure 6: Overall peak performance on Edison

Performance on other machines

The following plot shows the performance of our code in a MacBook Pro - 2GHz Intel Core i7 machine and a MacBook Pro -2.8 GHz Intel Core i7 machine. The respective average performances were 56% and 51%, respectively. While the performance seems to have slightly increased, we are assuming that this is because the block sizes for the cache blocking method were not again optimized for each machine (we used the ones for Edison).

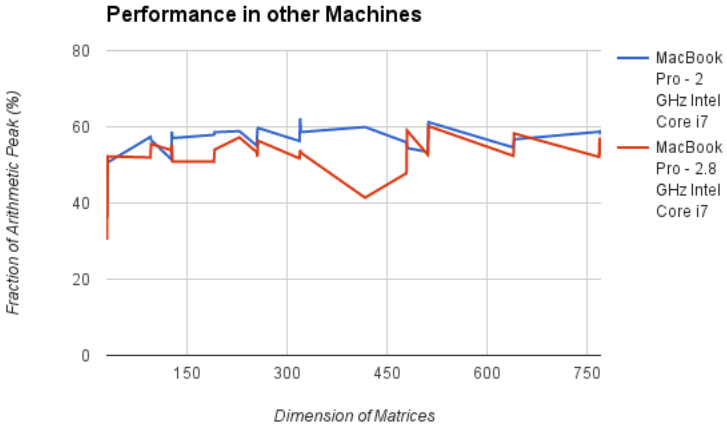

Figure 7: Overall performance on other machines

Submitted Codes

Together with the report we submit two folders:

- The first folder named “Final" contains our final submission files:
  - benchmark.c
  - dgemm-final.c
  - SSE.H required by dgemm-final.c
  - Makefile
- The second folder named “Misc” contains methods that are not integrated into the final design and code for finding the best block sizes for cache blocking.

Contribution

All three of us worked on the report. Xingjie mainly focused on cache oblivious optimization methods, changing the layout of matrices and manual loop unrolling, Sayna did the SSE multiplication along with padding, copy optimization and changing loop ordering , while Fotis worked on cache blocking, a failed attempt to use AVX, and performing experiments. We also had regular meetings throughout our cooperation, in order to coordinate, merge our codes, and write the report.

Bibliography

[1] K. Goto and R. A. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, p. 12, 2008.

[2] C. Lomont, "Introduction to intel advanced vector extensions," *Intel White Paper*, 2011.

[3] U. Drepper, "What every programmer should know about memory," *Red Hat, Inc*, vol. 11, p. 2007, 2007.

[4] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," *ACM SIGOPS Operating Systems Review*, vol. 25, no. Special Issue, pp. 63–74, 1991.

[5] S. Chellappa, F. Franchetti, and M. Püschel, "How to write fast numerical code: A small introduction," in *Generative and Transformational Techniques in Software Engineering II*, pp. 196–259, Springer, 2008.