GENERATING VECTOR CODE FOR MATRIX-MATRIX MULTIPLICATION

Joohoon Lee and Dongkeun Lee

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

The current state of the art Matrix-Matrix-Multiplication (MMM) kernel is known as ATLAS, which generates the best performing MMM code by search. However, today's computer architecture changes rapidly and it is hard to generate a high performance code without knowing how to use the new instruction sets. Since ATLAS does not make use of blocking for L2 cache, or SSE/SSE2 instruction, we are encouraged to improve ATLAS to obtain higher MMM performance than that of the original ATLAS. Our experiment result shows that we can obtain high performance using SSE/SSE2 which is available on the new generations of Pentium.

1. INTRODUCTION

1.1 Motivation

The matrix-matrix-multiplication (MMM) kernel has been widely used in numerical computing. There have been a lot of efforts to generate optimized code for different architectures. One distinguishable attempt is the ATLAS project. However, ATLAS does not take advantage of some of the new features of state-of-the-art processors. Thus, we want to create a better code generator that makes use of such advanced features.

1.2 Previous Work and State of the Art

The current state-of-the-art platform adaptation and code generation for the MMM is ATLAS. ATLAS uses register blocking, L1-cache blocking, instruction skewing and loop unrolling techniques to generate a fast code that are suitable for each platform. However, ATLAS does not block for L2-cache, and does not generate code that uses SSE / SSE2 instructions that are present in some of the new architectures such as Pentium 4. Since Pentium 4 is the most widely used architecture today, the lack of support for SSE instruction is a big disadvantage, because the use of SSE can dramatically increase the performance. The L2-cache blocking improves performance when the matrix size is too big to fit in L1 cache. Most of the recent architectures have multiple levels of memory hierarchy, and L2-cache is very common, and some recent architecture also has L3-cache.

1.3 What I Am Going to Do

We want to make a self platform adapting code generator for MMM, which supports the L2 cache blocking and SSE instructions on top of all of the current ATLAS features.

1.4 Organization of the Paper

In section 2, we provide the background on the matrix matrix multiplication and its cost analysis. We explain our methods in section 3, then present the results in section 4. Section 5 concludes the paper, and section 6 is the references we used.

2. NECESSARY BACKGROUND

2.1 Matrix Matrix Multiplication

The MMM can be performed by the following equation.

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj} \,,$$

where C_{ij} is each entry of result matrix and $A_{ik.}$ B_{kj} are each entry of input matrices. For each entries of the output matrix, the computation requires n additions and n multiplication, where n is the size of the matrix.

2.2 Cost Analysis

The arithmetic cost of multiplying an n x n by an n x n is $2n^3 – n^2$ by definition. For each entry in the result matrix C, a row vector of matrix A and a column vector of matrix B is multiplied together, thus it requires n multiplications. After multiplication, we need to sum up the multiplied values of the vector requiring n-1 additions. There are n^2 entries in the result matrix C, so the total cost of the MMM is $n^2(n + n – 1) = 2n^3 – n^2$.

3. METHOD

3.1 **Our version of ATLAS**

We first tried to implement most of the current features of ATLAS in our own code. The current ATLAS performs the following major optimizations to generate the optimal code. Each optimization is parametrized with the parameters specified.

• Register Blocking	(N_u, M_u)
• L-1 Cache Blocking	(N_B)
• Loop Unrolling	(K_u)
• Instruction Skewing	(L_s)

We implemented all of the above optimizations with exception of the Instruction Skewing. However, it was very hard to achieve the high performance that ATLAS can generate in limited time we had on this project. In addition, some of the optimizations done in ATLAS was not fully understood, thus it was almost impossible to implement a code that performs better than original ATLAS generated code. Since our focus was to generate a code that makes use of the SSE/SSE2, we decided to spend more time on optimizing the new code with SSE/SSE2 instead of spending time on a code that ATLAS can do much better job.

3.2 **Self-verification System**

After implementing a rough version of codes, we made a self verification system which compares the result of computation automatically, so that we can be sure of the correctness of our new code. This system enabled quick verification of the new codes. The correctness of computation is very important, because small variations in the code can lead to wrong results.

3.3 **SSE / SSE2**

SSE(Streaming SIMD Extensions) is SIMD instruction set designed by Intel. SSE allows floating point arithmetic operations in a set of 4 single precision number which are stored in 128bit special registers. In theory, SSE can achieve a performance gain by a factor of 4. SSE2 is an extension to the basic SSE instruction set with the feature of support for double-precision (64bit) floating point numbers. However, SSE2 uses the same 128bt registers, therefore, SSE2 only supports 2-way floating point operations. This means that SSE2 can achieve performance gain by factor of 2.

We wanted to verify the correctness of our SSE / SSE2 codes before we make code generator. Thus, we made our first version of code that uses SSE / SSE2 instructions with fixed parameters instead of automatically generating corresponding code according to the parameters. After we verified the correctness using the verification system we made earlier, then we made an automatic code generator that creates a code using parameterized inputs.

3.4 **L-2 Cache Blocking**

We added the L-2 cache blocking to our code generator. We used parameter N_{B2} to represent the level 2 blocking. This optimization is aimed for big matrices that cannot fit in L-1 cache.

3.5 **Search / Code generation**

The search technique used in ATLAS is orthogonal line search. What ATLAS does is fix all but one of the parameters then conducts a search over all possible values of that specific parameter. After ATLAS obtains the best performing parameter, that information is fed back to optimize another parameter. We followed the ATLAS and made our own search program. The parameter N_{B2} is added to the search because our code generator uses L-2 cache blocking. Our code generator also generates SSE / SSE2 codes if the architecture supports them.

3.6 **Possible TLB miss optimization**

There are more interesting attempts to achieve high performance MMM. The Goto paper mentions how one can optimize the performance of MMM by reducing the TLB misses. Most of today's architectures have multiple levels of memory hierarchy to buttress the CPU. However, as you go down the memory hierarchy, more and more penalties are unavoidable because lower level cache memory generally has poor performance compared to registers or L1 cache. However, L2 cache is generally much larger than L1 cache, so it compensates for its slow speed. Computer architects usually design the L2 cache work in size of page size, which is a commonly used for virtual memory translation. However, this translation takes a quite considerable amount of time, so there is dedicated cache structure to reduce the translation penalty. This structure is known as Table Look-aside Buffer (TLB). If we can align the memory access so that everything we perform fits in a page size, then we can minimize the TLB cache misses and gain more performance. If we have more time to work on this project, we would like to try implementing this into our code. Due to lack of time, we simply aligned all the matrices to page boundary when we allocate the memory in hope to reduce some TLB misses if possible.

4. EXPERIMENTAL RESULTS

4.1 **Overview**

We first made triple loop MMM code and normally generated MMM code, which searches the best parameters and gives us the better result than that of triple loop MMM. However, our goal is to improve normally generated MMM code using SSE/SSE2 instruction. Our improved MMM code using SSE/SSE2 instruction gives us remarkable high performance compare to the normally generated MMM or nested triple loop MMM. The SSE code achieves peak performance that is almost six times faster than the non-SSE code. The SSE2 code is faster by factor of two compared to non-SSE code. The experimental result clearly shows that the use of SSE/SSE2 instructions can dramatically increase the performance of MMM.

4.2 **Experimental Setup**

To test performance, we used a Pentium4 (Hyper Threading), 3.0 GHz under linux. The machine has 16KB L1 cache and 1024KB L2 cache. For compilation we used GCC 3.34 with following flag:
-03 -Wall -I. -march=pentium4 -mfpmath=sse -fomit -lm.

4.3 **Search Result**

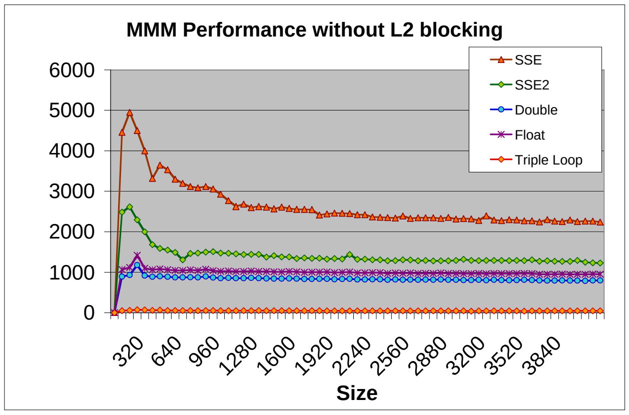
Our search program determined the following parameters for the optimal performance on our test machine.

- $N_u = 2$, $M_u = 2$, $N_B = 32$, $N_{B2} = 256$, $K_u = 32$

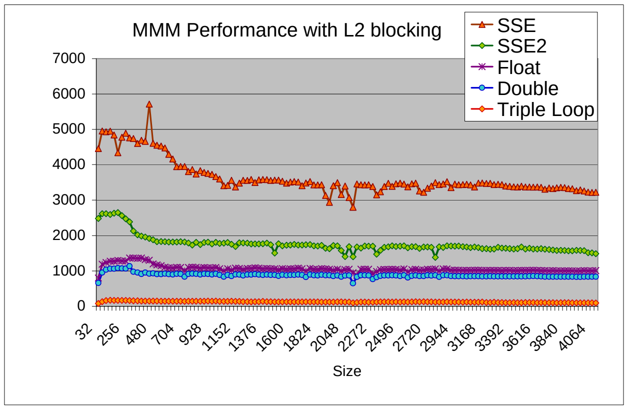
The Pentium 4 has 8 registers, thus N_u and M_u value of 2 is perfect size for register blocking. The L-1 and L-2 cache blocking size is determined by searching for the best level 2 blocking size N_{B2} first, then searching level one block size. The unrolling factor K_u is then determined.

4.4 **One level of blocking**

The Plot 1 shows the performance of various codes without the level 2 cache blocking. The performance of SSE code is 50 times as high as that of triple loop MMM. The graph shows that since we only block for one level of cache, our codes cannot sustain the high performance as the test matrix size gets larger. Our experimental data shows that without L2 cache blocking, our code suffers significant performance penalty if the test matrix size is larger than 512. We plotted both a single precision (Float) and double precision (Double) MMM performance on the graph because SSE only supports single precision calculation, so we need to compare the performance with the same data type. If we compare the performance of SSE2 code with the double precision code, we can see that SSE2 code is approximately 1.5 times faster. In theory we should have gotten twice faster, since SSE2 supports two way floating point operations, but there is overhead associated with the SSE2 instruction that reduces the ideal performance gain.



Plot 1: MMM Performance without L2 Blocking



Plot 2: MMM Performance with L2 Blocking

4.4 **Two levels of blocking**

Plot 2 shows the performance after we implemented the level 2 cache blocking optimizations. With L-2 blocking, our code sustains high performance throughout larger range of matrix sizes. The peak performance of SSE code reached almost 6000 MFLOPS, which is six times faster than our original generated code. It does not make sense to gain factor of 6, because SSE instruction supports 4-way vector calculations, but if we think about the sub-optimality of our normal code, then it makes more sense. On a 3Ghz Pentium 4, the theoretical maximum performance without the use of SSE/SSE2 instructions is approximately 3GFLOPS, and ATLAS reaches about 75% of the peak performance, so in this case, ATLAS code would have reached 2250MFLOPS. If we compare our SSE code to ATLAS, then we can see that we only gained about a factor of three. This is due to the overhead of using SSE instructions.

5. CONCLUSIONS

The speed of computer has been growing exponentially. However, the computer architects will soon face the end of the Moore's law. When the architects cannot achieve higher performance by just scaling, they will come out with new techniques to keep improving the current processors. One remarkable approach is dual core systems that are beginning to appear just now. ATLAS had the right idea when they created an automatic code generator using search to achieve high performance on new architectures, but in order to optimally use all the potentials of the new hardware, it is important to understand and make use of the new features. If a software developer does not use the new features, then developing new hardware will be pointless.

6. REFERENCES

[1] R. Whaley, A. Petitet, J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS Project", Sep. 2000

[2] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, "Is Search Really Necessary to Generate High-Performance BLAS?".

[3] K. Goto, R. Geijn, "On Reducing TLB Misses in Matrix Multiplication" FLAME Working Note #9, The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2002-55. Nov. 2002