OPTIMIZING MATRIX-MATRIX MULTIPLICATION FOR AN EMBEDDED VLIW PROCESSOR

Roland E. Wunderlich
rolandw@cmu.edu

Final Report – 18-799b Algorithms and Computation in Signal Processing, Spring 2005
Carnegie Mellon University

ABSTRACT

The optimization of matrix-matrix multiplication (MMM) performance has been well studied on conventional general-purpose processors like the Intel Pentium 4. Fast algorithms, such as those in the Goto and ATLAS BLAS libraries, exploit common microarchitectural features including superscalar execution and the cache and TLB hierarchy to achieve near-peak performance. However, the microarchitectures of embedded processors typically use explicitly parallel in-order execution and have configurable memory hierarchies. Thus, approaches that find good MMM code for processors like the Pentium may not be as effective for embedded processors.

For this project, I investigated the methods needed to achieve high performance MMM on an embedded VLIW (very-long instruction word) processor, the Texas Instruments C6713 floating-point DSP. This processor has three distinguishing features that affect an MMM implementation: an 8-wide in-order pipeline, an L2 mapped RAM, i.e., software-controlled scratch pad, and a direct memory access (DMA) engine. I present MMM implementations obtained through search and a model-driven approach that leverage the DSP microarchitecture. By using the scratch pad and DMA, I observed a 51% performance increase over a blocked MMM implementation.

1. INTRODUCTION

The availability of a high performance MMM implementation is of critical importance for a large range of numerical computation problems. MMM is both a common standalone function and a ubiquitous kernel of more complex computations. Unfortunately, naïve implementations have poor data locality, but speedups are possible if the data reuse during the computation of a MMM is properly ordered. The data reuse is apparent since $O(n^3)$ operations are performed on just $O(n^2)$ data. As a result of MMM's importance and opportunity for high performance implementations, the development of the fastest possible code is worth the investment of time and effort.

Manufacturers of CPUs often provide hand-tuned assembly code implementations of MMM to ensure good computation performance on their product. The Intel Math Kernel Library [1] contains such a MMM implementation that achieves 78% of the peak performance [3] of the Pentium 4. An even higher level of performance, 86% of peak performance, is realized by another hand-tuned implementation, the Goto BLAS library [3]. While both of these libraries are fast, it requires extensive effort to port them to new microarchitectures without sacrificing performance.

An alternative approach to creating fast MMM code is automatic code generation for specific microarchitectures. The ATLAS library generator [4] uses search to find optimal implementation parameters for a target processor. While ATLAS produces competitive MMM code (69% of peak performance [3] on the Pentium 4), the search process cannot examine the entire parameter space in a tractable amount of time. Recent work by Yotov et al. [5] has shown that a model-driven optimization can obtain results comparable to ATLAS without the time needed for search.

Both the Goto and ATLAS BLAS libraries are intended for use with modern processors that have deep superscalar pipelines and large hardware-managed cache hierarchies. These implementations are not optimal for embedded processors that typically have short pipelines with compiler-controlled parallelism and small software-managed cache hierarchies. I have modified both the search and model-driven methods to automatically determine optimal implementation parameters for the Texas Instruments (TI) C6713 floating-point DSP [2] to determine what fast MMM requires on this platform.

I use the MMM implementation included in the TI DSP Library as the kernel of my blocked MMM implementation. The TI MMM is hand-tuned assembly code that only performs well with input matrices that fit in the L1 data cache. I perform a search, and derive an analytic model, to determine the optimal matrix block size for inputs that do not fit in cache. In addition, I modify the blocked MMM algorithm to make use of the scratch pad and DMA. I also derive an analytic model to determine the best usage of the scratch pad. I show that the use of L2

Table 1. Microarchitecture comparison DSP vs. GPP

	TI C6713	Pentium 4
Process technology	0.13 µm	0.13 µm
Clock speed	300 MHz	3.4 GHz
Price[1]	$37	$286
Floating point units	6 (32-bit)	1 (128-bit)
MFLOPS (32-bit)	1800	13,600
Peak power	1.5 W	90 W
L1 data cache size	4 KB	8 KB
Unified L2 cache size	0-64 KB	512 KB
Scratch pad SRAM	192-256 KB	none
L1 latency (cycles, ns)	4, 13.3	9, 2.6
L2 & scratch pad latency	8, 26.7	16, 4.7

scratch pad and DMA yield large performance improvements over conventional blocked MMM.

The remainder of this report is organized as follows. I present background on the C6713 DSP, the vendor supplied MMM, and the original and model-based ATLAS in Section 2. In Section 3, I determine the best block size analytically and experimentally. Section 4 presents my investigation of using the L2 mapped RAM in MMM. Finally, I discuss the conclusions of this work in Section 5.

2. BACKGROUND

2.1. DSP microarchitecture

The TMS320C6713 DSP is the latest implementation in the C67x family of high-end floating-point (FP) DSP chips from Texas Instruments. This processor is intended for demanding embedded applications that have power and cost constraints that preclude the use of a general purpose processor (GPP) such as the Pentium. The chip is implemented in a 0.13 µm process technology and is available at speeds up to 300 MHz. It has two single precision FP adders, two FP comparators, and two FP multipliers that give it a peak performance of 1800 MFLOPS, or for MMM, 1200 MFLOPS because only the adders and multipliers are used. The processor consumes a peak of ~1.5 watts, and costs about $37.[1] Table 1 compares the C6713 to the fastest Pentium 4 in the same 0.13 µm process technology, the Northwood 3.4 GHz processor.

It is not unusual for embedded processors to have some software control of their cache hierarchy. Specifically, fast but small SRAM is often available to store time-critical program and data segments. This SRAM can be used effectively when embedded processors execute only a single program at a time and have well-defined data access patterns. The C6713 has a 192 KB scratch pad SRAM, referred to by TI as L2 mapped RAM, and a 64 KB L2 cache that can be dynamically converted to scratch pad memory as well. This means that up to 256 KB of low-latency (8 cycles) SRAM is available.

Finally, many DSP implementations have DMA engines that allow for background block memory transfers. These DMA engines can perform memory copy operations between the many I/O ports of the DSP. The C6713 DSP can also perform memory copies to the scratch pad. This essentially allows data to be loaded into the "L2 cache" with minimal interaction with the CPU. When transferring data to and from the scratch pad to main memory, the C6713 can achieve transfer rates of 400 MB/s with 100 MHz SDRAM.

2.2. Blocked MMM parameters

The data access pattern of the MMM can be improved for better cache performance with better data reuse locality. This can be achieved by partitioning the computation to operate on one set of cache resident sub-matrices (called blocks) before moving on to subsequent computations. The computation that is performed on these cache resident blocks is called the mini-MMM.

I implemented a blocked MMM implementation (with block copying) as described by Yotov et al. [5]. This algorithm has nine parameters that describe the final C code. Four of the parameters determine the loop structure of the MMM and mini-MMM, while the other five describe the organization of the innermost loop body.

Loop structure parameters:

- N_B = block edge dimension (mini-MMM input)
- M_U, N_U = register block dimensions
- K_U = unroll factor for inner-most loop

Innermost loop body:

- L_S = latency for computation scheduling
- FMA = fused multiply-add availability
- F_F, I_F, N_F = scheduling of loads

2.3. TI DSP Library MMM

TI provides an optimized single precision floating point MMM implementation for C67x processors, the DSPF_sp_mat_mul() function. This function is not a blocked MMM, and thus is far from optimal for input matrices that do not fit in the L1 data cache. However, it is optimal for L1 cache resident data, its innermost loop attains 100% of the peak performance of the C6713, with only small overhead for the *i* and *j* loop control code.

The TI MMM function was hand-written in assembly code, and the innermost loop was written such that all four floating-point units are fully utilized. Data loads are made at least 4 cycles before dependent operations, resulting in stall-free execution of the innermost loop when all data is in the L1 cache.

The 8-wide pipeline allows the innermost loop control code to be executed in parallel with the FP operations, yielding 100% peak FLOPS performance within this loop. The wide pipeline means that the innermost loop does not

[1] Prices are current as of April 2005.

need to be unrolled to amortize loop bounds control code. Nevertheless, the DSPF_sp_mat_mul() function unrolls the innermost loop by a factor of 8. It must be unrolled at least this much to cover the 5 branch delay slots and schedule the operations more than 4 cycles from data loads.

It is also worth noting that the TI MMM implementation has only 248 instructions, or 992 bytes. This is approximately one-fourth of the 4 KB L1 instruction cache, leaving 3 KB for the outer MMM loops to be implemented with no instruction cache misses for the entire blocked MMM. Assuming no instruction or data cache misses, the runtime of DSPF_sp_mat_mul() for two $n \times n$ input matrices (where n is even) is

$$\text{Runtime} = 0.5n^3 + 6n^2 + 4n + 22 \text{ cycles.}$$

When using DSPF_sp_mat_mul() as the mini-MMM in a blocked MMM, all but one parameter are fixed:

$M_U = N_U = K_U = 2$
$L_S = 4$ cycles
FMA = not available
$F_F, I_F, N_F = 4$ cycles

This leaves us to determine the optimal value of N_B.

2.4. ATLAS and model-determined optimal block size

ATLAS determines the best block size by measuring the performance mini-MMM implementations with a range of values for N_B. Specifically, ATLAS tests values for $\mathrm{N_B}$ that are multiples of four between 16 and $\min(80, \sqrt{C_1})$, where C_1 is the size of the L1 data cache.

Goto and van de Geijn state that non-square mini-MMM blocks are preferable to avoid TLB misses [3]. They conclude that the mini-MMM input matrix B and output matrix C should have relatively large row dimensions to minimize TLB miss penalties. This optimization is not required for the C6713 because there is no address translation for virtual memory.

Yotov et al. derived an analytic model to determine the highest performance block size by analyzing the MMM data access pattern and typical cache behavior. Their model for the optimal N_B is

$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3\left\lceil \frac{N_B \cdot N_U}{B_1} \right\rceil + \left\lceil \frac{M_U}{B_1} \right\rceil \cdot N_U \le \frac{C_1}{B_1},$$

where B_1 is the cache line size. This model assumes an allocate-on-write policy for the L1 cache. While this assumption is true for processors like the Pentium 4, the C6713 does not allocate an L1 data cache entry upon writes. I refine this model to match the C6713's cache behavior, and solve for the optimal N_B size in Section 3.1.

3. OPTIMAL BLOCKED MMM

In this section, I adapt Yotov et al.'s model for the C6713 and solve for the optimal block size. I confirm this result by performing an ATLAS style search across a range of possible block sizes, and present my blocked MMM.

3.1. Analytic model for N_B

The model presented by Yotov et al. assumes that when elements the output matrix are written out, they occupy space in the L1 data cache. This is not the case on the C6713; only data reads can allocate L1 cache lines. Data writes are allocated in the L2 cache unless the data already exists in the L1 data cache. The model for N_B simplifies to

$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + 2\left\lceil \frac{N_B \cdot N_U}{B_1} \right\rceil \le \frac{C_1}{B_1}$$

for any processor that does not allocate-on-write. If we substitute the values for C_1, B_1, and N_U from the C6713 and the TI MMM implementation, we can solve for N_B. The C6713 has a L1 data cache of 1024 words, and a 32 byte cache line size (8 words). Thus

$$\left\lceil \frac{N_B^2}{8} \right\rceil + 2\left\lceil \frac{N_B}{4} \right\rceil \le 128 \Rightarrow N_B \le 29$$

We expect that a block size of 28×28 (3136 bytes or 49 cache lines) will have the best performance, after rounding N_B down to an even value. Using the TI MMM runtime formula from Section 2.3 we can estimate that a 28×28 MMM will take 15,814 cycles. This is equal to 818 MFLOPS at 300 MHz, or 68% peak performance of the C6713 DSP.

3.2. Searching for the best N_B

I performed a search for the highest performance for N_B between 16 and 48 for every multiple of two. All performance results are presented as MFLOPS as measured on the TI Code Composer Studio 3.0 cycle accurate simulator. I subtracted the overhead of the input setup code that touched the input matrices in order to load them in cache as if they had just been produced by a preceding function. The L2 cache was configured at its largest possible size of 64 KB.

The TI TMS320C6x C/C++ Compiler 5.0 was used to compile my C code implementation. The relevant compiler options that used were –O3 and –mv6700, to turn on optimization and specify the microarchitecture, respectively. The compiler options had no effect on the TI MMM since it is implemented in assembly code.

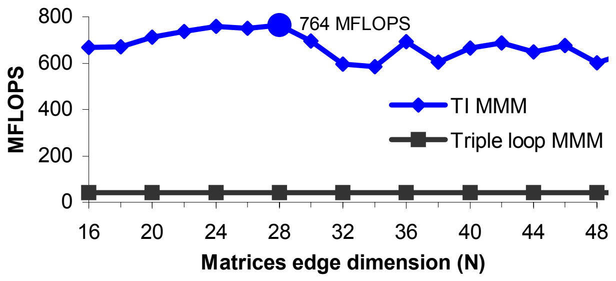

Figure 1. mini-MMM performance across N_B

The results of the search for the best block size are plotted in Figure 1. A naïve triple loop implementation is shown for comparison. The block size of 28×28 is the best value for N_B, as the analytic model predicted. The TI MMM reaches 764 MFLOPS, or 64% of peak performance at this block size. The difference from the expected 818 MFLOPS performance is explained by compulsory instruction cache misses.

3.3. Blocked MMM implementation

Using the TI MMM as a mini-MMM function, I wrote a blocked MMM implementation. The pseudo-code of this function is shown in Figure 2 for further reference. The performance of this implementation is plotted for matrices up to size 224×224 in Figure 3.

My blocked MMM reaches is 83% faster than the TI MMM for large matrices. The TI MMM is faster for $N < 112$ (49 KB per input matrix) because it does not have the overhead of copying blocks, but for larger matrices this overhead is justified since it helps improve data locality. Also, the performance of the TI MMM drops gradually when the input matrices are resident in L2 cache because there is only a relatively small increase in latency.

4. DSP-SPECIFIC OPTIMIZATIONS

The C6713 can convert its L2 cache into software-controlled memory in 16 KB steps, allowing 64 to 0 KB of L2 cache, and 192 to 256 KB of scratch pad SRAM. In addition, the C6713 has a DMA engine that can perform efficient memory block copies from main memory to this scratch pad. In this section, I investigate the possibility of improving the blocked MMM's performance by leveraging the memory system of the C6713.

4.1. Sources of cache misses in blocked MMM

To determine the best usage of the scratch pad, I identify the potential sources of cache misses. There are four main sources of compulsory and capacity misses to both the L1 and L2 caches in my blocked MMM. I assume that stores do not incur any cost. This assumption implies that the store buffer never fills, and thus, never causes stalls.

```
 1 float[N][N] MMM(float[N][N] A, float[N][N] B) {
 2     float[NB][NB] a;
 3     float[N][NB] b;
 4     float[NB][NB] c;
 5     for j = 0:NB:N {
 6         b = copyBlock(B, (0,j));
 7         for i = 0:NB:N {
 8             fillZeroes(c);
 9             for k = 0: NB:N {
10                 a = copyBlock(A, (i,k));
11                 c += miniMMM(a, b+k·NB);
12             }
13             writeBlock(c, C, (i,j));
14         }
15     }
16 }
```

Figure 2. Blocked MMM pseudo-code

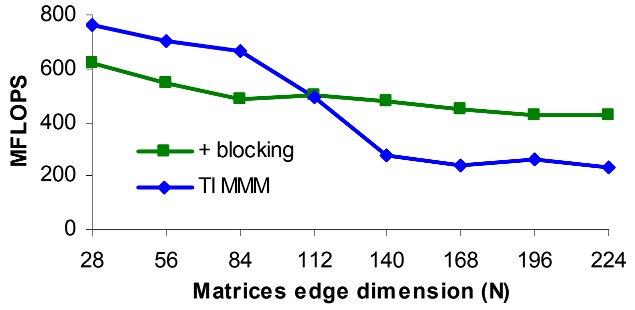

Figure 3. Blocked MMM performance vs. TI MMM

The first source of potential misses is the two calls to the copyBlock() function (see Figure 2). This function loads portions of the input matrices A and B and stores them into variables a and b. There is no optimization that can eliminate the compulsory misses incurred by the first call to copyBlock() on line 6. This is because each portion of the B matrix that is loaded is never used again. The second call to copyBlock() on line 10 incurs compulsory misses on the loop iterations where $j = 0$. On all other iterations, it will only incur capacity misses if A does not fit completely in L2 cache along with a, b, and c. These misses occur only when $N > 92$.

The second source of potential data cache misses is the writeBlock() function on line 13. This function has to read all the values of c. Since the c block is read and written to by the preceding calls to miniMMM(), it will reside entirely within the L1 and L2 caches. Thus only L1 capacity misses may occur, but these cannot be avoided since moving c into the L1 would evict useful portions of a and b.

The third source of potential cache misses is the miniMMM() function when it loads a and b throughout its execution. Before the miniMMM() function executes, block a will be in the L2 cache due to the copyBlock() call on line 10. The b block will also reside in L2 cache, except for iterations where $i = k = 0$. On these iterations, the relevant portion of the b block will be in L2 cache only if the entire b block fits in L2 cache along with a, c, and one

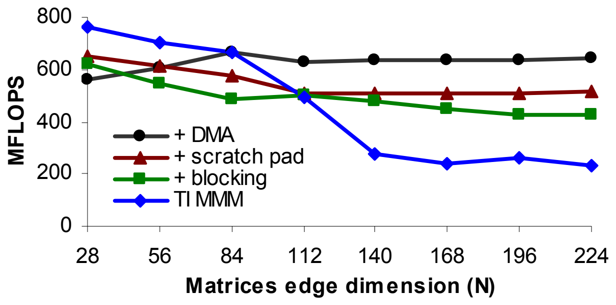

Figure 4. Performance using the scratch pad and DMA

block of A. Thus, the b block will not be in L2 cache when $i = k = 0$ for $N > 464$.

The final source of potential cache misses is the size of the blocked MMM code. If the blocked MMM and the mini-MMM functions are more than 4 KB of instructions, then instruction cache capacity misses will occur during the blocked MMM.

4.2. Model for scratch pad usage

Using the scratch pad SRAM to store data and instructions reduces L2 cache capacity misses, assuming the L2 cache size is not reduced. By placing all instructions, and the a, b, and c blocks, in L2 mapped RAM, more space in the L2 cache is available. This decreases capacity misses in two of the cases described in Section 4.1.

The first case where capacity misses are reduced is the call to copyBlock() on line 10. The threshold where A fits within the L2 cache increases to $N < 114$. The second case where misses are reduced is the miniMMM() function for iterations where $i = k = 0$. The miss latency reduces when the b block fits in the scratch pad. This is possible only when $N < 1662$. I do not further consider the case when $N > 1662$ and the b block does not completely fit in the scratch pad.

In summary, L2 cache capacity misses are reduced when all instructions, and the a, b, and c blocks, are placed in the scratch pad for $92 < N < 114$. Also, a more complex implementation is required for $N > 1662$ because the b block no longer fits in the scratch pad.

4.3. Scratch pad and DMA MMM performance

The placement of instructions and data into the scratch pad can be controlled at compile time during the linking phase. Pragmas in the C code define which memory sections variables and functions will reside in at runtime. I modified the blocked MMM to store the MMM and miniMMM functions in the scratch pad. In addition, the a, b, and c blocks are also statically allocated in the scratch pad. The performance of this implementation versus the original blocked MMM is plotted for matrices up to size 224×224 in Figure 4. The use of the scratch pad has a small performance benefit, mostly from a reduction in instruction fetch stalls.

The second optimization specific to the C6713 DSP was using the DMA engine to perform the copyBlock(), fillZeroes(), and writeBlock() functions more efficiently than possible with the CPU. The reason that these operations can be performed faster by DMA is that the CPU must operate on one word of data at a time, but the DMA engine can operate on entire cache lines at once.

The performance of my final implementation that uses blocking, the sctach pad, and DMA is shown in Figure 4. The DMA code is the worst performer at small input sizes because of the setup overhead of DMA transfers. Once larger DMA transfers are performed for large matrices, its performance advantage grows. For large matricies, the implementation using DMA achieves performance only 15% slower than the TI MMM at $N = 28$.

5. CONCLUSIONS

The blocked MMM is an effective algorithm for high performance on general purpose processors like the Pentium 4. However, when the scratch pad SRAM and DMA of the C6713 DSP are used, there are further performance gains. In fact, if the DMA operations are performed in parallel with the mini-MMM, I expect to achieve performance only limited by the mini-MMM.

6. REFERENCES

[1] Intel Math Kernel Library web site [Online]. Available: http://www.intel.com/software/products/mkl/

[2] Texas Instruments DSP products web site [Online]. Available: http://dspvillage.ti.com/

[3] K. Goto and R. van de Geijn. "On reducing TLB misses in matrix multiplication." Dept. Comput. Sci., Univ. Texas, Austin, Tech. Rep. TR-2002-55, 2002.

[4] R. C. Whaley, A. Petitet, and J. J. Dongarra. "Automated empirical optimization of software and the ATLAS project." *Parallel Comput.*, vol. 27, no. 1-2, pp. 3-35,, 2001.

[5] K. Yotov, X. Li, G. Ren, M. J. Garzarán, D. Padua, K. Pingali, and P. Stodghill. "Is search really necessary to generate high-performance BLAS?" *Proc. IEEE*, vol. 93, no. 2, pp. 358-386, 2005.