RC24172 (W0611-123) November 27, 2006
Computer Science

IBM Research Report

Is Cache-oblivious DGEMM Viable?

J. Gunnels, F. Gustavson
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

K. Pingali, K. Yotov
Department of Computer Science
Cornell University
Ithaca, NY 14853

Is Cache-oblivious DGEMM Viable?

J. Gunnels, F. Gustavson[1] and K. Pingali, K. Yotov[2]

[1] T. J. Watson Research Center, Yorktown Heights, NY 10598, USA,
(gunnels,fg2)@us.ibm.com
[2] Dept. of Computer Science, Cornell University, Ithaca, NY 14853, USA,
(pingali,kyotov)@cs.cornell.edu

Abstract. We present a study of implementations of DGEMM using both the cache-oblivious and cache-conscious programming styles. The cache-oblivious programs use recursion and automatically block DGEMM operands A, B, C for the memory hierarchy. The cache-conscious programs use iteration and explicitly block A, B, C for register files, all caches and memory. Our study shows that the cache-oblivious programs achieve substantially less performance than the cache-conscious programs. We discuss why this is so and suggest approaches for improving the performance of cache-oblivious programs.

1 Introduction

One recommendation of the Algorithms and Architectures (AA) approach [1, 6, 10] is that researchers from Architecture, Compilers and Algorithms communicate. In this spirit, the authors of this paper, who are from the compilers and algorithms areas, have been collaborating in a research project to build BRILA (Block Recursive Implementation of Linear Algebra), a domain-specific compiler for Dense Linear Algebra (DLA) programs. This compiler takes recursive descriptions of linear algebra problems, and produces optimized iterative or recursive programs as output.

As part of this effort, we investigated the cache-oblivious (CO) and cache-conscious (CC) approaches to implementing programs for machines with memory hierarchies. CO research introduced recursion via the divide and conquer paradigm into DLA approximately ten years ago [9, 4]. The work in [9] was inspired by earlier work [1] which first enunciated the AA approach; see also [10, 6]. [9] additionally advocated the use of new data structures (NDS) and L1 cache blocking to improve the performance of DLA recursive algorithms such as the Level 3 BLAS [6].

The results described in this paper summarize and extend a companion paper directed towards the optimizing compiler community [13]. Our main findings can be summarized as follows. As is well-known, the performance of many programs such as DGEMM is limited by the performance of the memory system in two ways. First, the latency of memory accesses can be many hundreds of cycles, so the processor may be stalled most of the time, waiting for reads to complete. Second, the bandwidth from memory is usually far less than the rate at which the processor can consume data. Our study examined these limitations for highly optimized CO and CC DGEMM programs, generated by BRILA, on four modern architectures: IBM Power 5, Sun UltraSPARC IIIi, Intel Itanium 2, and Intel Pentium 4 Xeon. We found that there is a significant gap between the performance of CO and CC algorithms for DGEMM; we provide reasons why this gap exists and how it might be overcome. We are not aware of any similar study in the literature.

The rest of this paper is organized as follows. In Section 2 we give a quantitative analysis of how blocking can reduce the latency of memory accesses as well as the bandwidth required from memory. Using this analysis, one can tell when an architecture cannot deliver a peak performing DGEMM. This was the case for the IBM PC604 computer whose L1 cache was too small and whose bandwidth between L2 and memory was too small[1]. In Section 3 we discuss the performance of naïve iterative and recursive programs. Neither program performs well on any architecture but for different reasons: the iterative program performs poorly mainly because of poor memory hierarchy behavior, and the recursive one behaves poorly mainly because of recursive overhead. In Section 4 we evaluate approaches for reducing recursive overhead. Following [4] we introduce, in [13], a *recursive microkernel* which does instruction scheduling and register blocking for a small problem of size $R_U \times R_U \times R_U$. However, even after considerable effort, we are unable to produce a recursive microkernel that performs well. In Section 5 we explore *iterative* microkernels produced by the BRILA compiler using tiling and unrolling of the standard iterative programs. We show that these iterative microkernels perform much better than the recursive microkernels. A main finding of [3, 11, 13] is that prefetching is important to obtain better performance. While prefetching is easy if the outer control structure is iterative, it is not clear how to accomplish this if the outer control structure is recursive. In Section 6 we discuss the importance of two recent architectural innovations: streaming and novel floating-point units [3, 14]. We describe a new concept called the L0/L1 cache interface for reasoning about the impact of these innovations [11]. Lastly, in Section 7, we summarize our findings on recursive and iterative approaches to matrix multiplication.

2 Approximate Blocking

The cache-oblivious approach can be viewed as a way of performing approximate blocking for memory hierarchies. Each step of the recursive divide-and-conquer process generates sub-problems of smaller size, and when the working set of a sub-problem fits in some level of the memory hierarchy, that sub-problem can execute without capacity misses at that level. It is known that this recursive approach is *I/O optimal* for common problems like matrix-multiplication and FFT, which means intuitively that the volume of data transfers between different cache

[1] The IBM PC architecture introduced prefetching instructions (called touches), and using them and the AA approach, one of us introduced the concept of algorithmic prefetching to improve DGEMM performance on this platform

levels is as small as it can be (to within constant factors) for any program implementing the same computation [7]. However, program performance is limited by both the latency of memory accesses and the bandwidth between different levels of the memory hierarchy. We argue in this section that minimizing the volume of data transfers between cache levels by approximate blocking may reduce bandwidth demands to an adequate level, but may not necessarily address the latency problem.

We consider blocked Matrix-Matrix Multiply (MMM) of $N \times N$ matrices in which each block computation multiplies matrices of size $N_B \times N_B$. We assume that there is no data reuse between block computations, which is a conservative assumption for both latency and bandwidth estimates. We find an upper bound on N_B by considering a simple two-level memory hierarchy model with a cache and main memory, and requiring the size of the working set of the block computation to be less than the capacity of the cache, C. Assume that the line size is L_C, and that the cache has an access latency l_C. Let the access latency of main memory be l_M. The working set is bounded above by the size of the sub-problem. Therefore, the following inequality is a conservative approximation.

$$3N_B^2 \leq C \tag{1}$$

The total number of memory accesses each block computation makes is $4N_B^3$. Each block computation brings $3N_B^2$ data into the cache, which results in $\frac{3N_B^2}{L_C}$ cold misses. If the block size is chosen so that the working set fits in the cache and there are no conflict misses, the cache miss ratio of the complete block computation is $\frac{3}{4N_B \times L_C}$. Assuming that memory accesses are not overlapped, the expected memory access latency is as follows:

$$l = \left(1 - \frac{3}{4N_B \times L_C}\right) \times l_C + \frac{3}{4N_B \times L_C} \times l_M \tag{2}$$

Equation (2) shows that the expected latency decreases with increasing N_B, so latency is minimized by choosing the largest N_B for which the working set fits in the cache. In practice, the expected memory latency computed from Equation (2) is somewhat pessimistic because loads can be overlapped with each other or with actual computations, reducing the effective values of l_C and l_M. These optimizations are extremely important in the generation of the micro-kernels, as is described in Section 4. Furthermore, hardware and software prefetching can also be used to reduce effective latency, as is discussed in Section 5 of [13] and Section 6.

Bandwidth considerations provide a lower bound for N_B. We start by considering the bandwidth between the registers and the L1 cache, which we will refer to as $B(L0, L1)$. Assume that the processor can perform one fused multiply-add per cycle. Therefore, the time to execute a matrix-multiply is bounded below by N^3 cycles. Without blocking for registers, the bandwidth required between registers and the L1 cache is therefore $4N^3 \div N^3 = 4$ doubles/cycle.

If the processor cannot sustain this bandwidth, it is necessary to perform register-blocking. If the size of the register block is N_B, we see that each block computation requires $4N_B^2$ data to be moved, so our simple memory model implies that the total data movement is $\left(\frac{N}{N_B}\right)^3 \times 4N_B^2 = \frac{4N^3}{N_B}$. The ideal execution time of the computation is still N^3, so the bandwidth required from memory is $\frac{4N^3}{N_B} \div N^3 = \frac{4}{N_B}$ doubles/cycle. Therefore, register-blocking by a factor of N_B reduces the bandwidth required from L1 cache by the same factor.

We can now write the following lower bound on the value of N_B, where $B(L0, L1)$ is the bandwidth between registers and the L1 cache.

$$\frac{4}{N_B} \leq B(L0, L1) \tag{3}$$

Inequalities (1) and (3) imply the following inequality for N_B:

$$\frac{4}{B(L0, L1)} \leq N_B \leq \sqrt{\frac{C}{3}} \tag{4}$$

This argument generalizes to a multi-level memory hierarchy. If $B(L_i, L_{i+1})$ is the bandwidth between levels i and $i+1$ in the memory hierarchy, $N_B\,(i)$ is the block size for the i^{th} cache level, and C_i is the capacity of this cache, we obtain the following inequality:

$$\frac{4}{B(L_i, L_{i+1})} \leq N_B\,(i) \leq \sqrt{\frac{C_i}{3}} \tag{5}$$

In principle, there may be no values of $N_B\,(i)$ that satisfy the inequality. This can happen if the capacity of the cache as well as the bandwidth to the next level of the memory hierarchy are small. According to this model, the bandwidth problem for such problems cannot be solved by blocking. The IBM PC604 processor is an example of such a processor.

Equation (5) shows that in general, there is a range of block sizes for which bandwidth constraints can be satisfied. In particular, if the upper bound in Equation (5) is more than twice the lower bound, the divide-and-conquer process in the cache-oblivious approach will generate a block size that lies within these bounds, and bandwidth constraints will be satisfied. However, Equation (2) suggests that latency might still be a problem, and that it may be a bigger problem for the CO approach since blocking only approximate.

3 Naïve Recursive and Iterative DGEMM Routines

As a baseline for performance comparisons, we considered naïve recursive routines that recursed down to 1×1 matrix multiplications, as well as unblocked iterative routines for MMM. We found that both performed very poorly but for different reasons. We show that recursive codes have low cache misses but high calling overhead whereas the opposite holds for iterative codes.

For these implementations we need a data structure for matrices and a control structure. We use standard Fortran and C two dimensional arrays to represent the matrix operands A, B, C of DGEMM. In [9, 6] the *all-dimensions* (AD) strategy of divide and conquer is described. Here one divides the rows and columns of A, B, C equally and thereby generates eight sub-problems of half the size. For the iterative control, we used simple *jki* loop order, [5]. This was the worst choice for iterative control as our matrices were stored in row major order. Both programs performed very poorly on all four platforms obtaining about 1% of peak. Three reasons emerged as the main causes of bad performance:

1. The overhead of recursive calling was in the hundreds of cycles whereas an independent FMA executed in one cycle.
2. Both programs made poor use of the floating point register files. Currently, compilers fail to track register values across procedure calls. For iterative codes compilers can do register blocking; however, none of our compilers were able to do so.
3. A remarkable fact emerged when we examined L2 cache misses on Itanium. Our iterative code suffered roughly *two* cache misses per FMA resulting in a miss ratio of 0.5! So, poor memory behavior limits the performance of our iterative code. For the recursive code the miss ratio was a tiny .002 misses per FMA, resulting in a miss ratio of 0.0005!. This low miss ratio is a practical manifestation of the theoretical I/O optimality of recursive programs. However, because of the calling overhead, I/O optimality alone did *not* guarantee good overall performance.

Many more details and experimental results about this Section are contained in Section 3 of [13].

4 Recursive Kernels for DGEMM and Reducing Recursive Overhead

A standard approach to reducing the recursive overhead is to cut the recursion off once the problem size is below some cut-off, and call a kernel routine to perform the sub-problem. For DGEMM one no longer performs single FMAs at the leaves of the recursion tree; instead, the program would call a kernel routine that would perform N_B^3 FMA's. This *microkernel* is a basic block obtained by fully unrolling the recursive code for a small problem of size $R_U \times R_U \times R_U$, as suggested by Frigo et al. [4]. This kernel routine has to be optimized carefully for the registers and the processor pipeline, so it is not "register-oblivious". We call this a *recursive* microkernel since the FMAs are performed in the same order as they were in the original recursive code.

The optimal R_U value is determined empirically for values between 1 and 15; *i.e.*, this microkernel must be done in a register file. The overhead of recursion is amortized over R_U^3 FMAs, rather than a single FMA. Furthermore, a compiler might be able to register allocate array elements used in the microkernel, which gives the effect of register blocking. Therefore, performance should improve significantly.

One also needs to worry about selecting a good data structure to represent the matrix operands A, B, C of the microkernel as the standard Fortran and C two dimensional arrays are often a poor choice [10]. Virtually all high performance BLAS libraries internally use a form of a contiguous blocked matrix, such as Row-Block-Row (RBR); *e.g.*, see [9, 10, 12, 6, 11]. An alternative is to use a recursive data layout, such as a space filling curve like Morton-Z [9, 6]. In [13] we compared the MMM performance using both these choices and we rarely saw any performance improvement using Morton-Z order over RBR. Thus [13] used RBR in all of its experiments and it chose the data block size to match the kernel block size.

In [13] we considered four different approaches to performing register allocation and scheduling for the recursive microkernel; see Sections 4.1,2,3,4 and Table 4 of [13]. The results of these four recursive microkernel experiments led us to the following four conclusions:

– The microkernel is critical to overall performance. Producing a high performance recursive microkernel is a non-trivial job, and requires substantial programming effort.
– The performance of the program obtained by following the canonical recipe (recursive outer control structure and recursive microkernel) is substantially lower than the near-peak performance of highly optimized iterative codes produced by ATLAS or in vendor BLAS. The best we were able to obtain was 63% of peak on the Itanium 2; on the UltraSPARC, performance was only 38% of peak.
– For generating the microkernel code, using Belady's algorithm [2] followed by scheduling may not be optimal. Belady's algorithm minimizes the number of loads, but minimizing loads does not necessarily maximize performance. An integrated register allocation and scheduling approach appears to perform better.
– Most compilers we used did not do a good job with register allocation and scheduling for long basic blocks. The situation is more muddied when processors perform register renaming and out-of-order instruction scheduling. The compiler research community needs to pay more attention to this problem.

5 Iterative Kernel Routines for DGEMM

Iterative microkernels are obtained by register-blocking the iterative code, and have three degrees of freedom called K_U, N_U, and M_U. The microkernel loads a block of the C matrix of size $M_U \times N_U$ into registers, and then accumulates in them the results of performing a sequence of outer products between column vectors of A of size M_U and row vectors of B of size N_U. Therefore, $M_U + N_U$ loads of A and B operands accomplish $M_U \times N_U$ FMA's. Clearly, with a large enough register file one can do cache blocking out of higher levels of cache than L1. This actually happens on Itanium and on IBM BG/L, see [3]. With this kernel, K_U can be very large.

The microkernels were generated by BRILA in [13] as follows:

1. Start with a simple kji triply-nested loop for performing an MMM with dimensions $\langle K_U, N_U, M_U \rangle$ and unroll it completely to produce a sequence of $M_U \times N_U \times K_U$ FMAs.
2. Use the algorithm described in Figure 5 of [13] for register allocation and scheduling, starting with the sequence of FMAs generated above. As in Section 4.4 of [13], Belady register allocation was used and [13] scheduled dependent instructions back-to-back on the Pentium 4 Xeon.

5.1 Iterative Minikernels

By blocking explicitly for L1 and other cache levels, we get iterative *minikernels*. These minikernels usually deal with square blocks, and invoke the iterative microkernel to perform the matrix multiplications. We also considered combining recursive outer control structures with the iterative microkernel [13].

Not surprisingly, we found that blocking the iterative code for all levels of cache gave roughly the same performance as using a recursive outer control structure with the iterative microkernel. However, the iterative outer control offers many more blocking choices as there are $3! =$ six choices for each higher cache level, although this large number of choices can be reduced to only four choices [8]. Furthermore, to get the highest level of performance, it is necessary to implement prefetching from the memory, as is done in hand-tuned libraries. Prefetching for iterative control structures is well understood, but appears to be more difficult for recursive control structures because they access the matrices in a relatively unstructured fashion. We believe more research into prefetching for recursive codes is needed.

6 Pipelined Streaming and Novel Floating Point Units

Finally, we discuss the impact of novel features of the processor pipeline.

Many processors now support streaming but there are a limited number of streaming units on a given platform [14]. In Section 5.1, we discussed minikernels that block explicitly for L1 and other cache levels [1, 15, 11]. In [13] the data structure of the minikernel was RBR, which is a special case of the Square Block Format (SBF) [9, 10, 12]. In [12] the authors show that this data format minimizes L1, L2, and TLB misses for common matrix operations that involve both row and column operations. Typically, the order NB of a SBF matrix is chosen so it will reside in L1 cache. Unfortunately, streaming makes the use of SBF suboptimal since the iterative microkernel requires $2N_U + M_U$ streams with RBR storage. Usually, M_U and N_U is around four so about twelve streams are required. Unfortunately, twelve is too large. With NDS, we show that these streams can be reduced to three, one each for the A, B, C operands of DGEMM. We note that three is minimal for DGEMM as one stream is required for each of DGEMM's operands.

Next, we discuss novel floating point units. These are SIMD vector like and several platforms now have them. For example, on Intel one has SSE units which are capable of delivering 4,2 multiplies or 4,2 adds for single,double precision operands in a single cycle. The peak MFlop rate is thus quadrupled/doubled with this floating point unit. On IBM Blue Gene, BG/L, there is a pair of double SIMD Floating point units with associated double SIMD floating point register files [3]. It turns out that these units are not fully integrated with the CPU. Thus, there is an interface problem that exists in getting data from the L1 cache into the register files of these novel units. In [11] we discuss the matters in more detail and in particular, we define a new concept called the L0/L1 cache interface; also see [3]. Note that in this context the L0 cache is the register file of a given processor.

Now we briefly describe how these problems can be handled. An answer lies in changing the internal data structure [11]. Instead of using a standard Fortran or C two dimensional array, it is necessary to use a four-dimensional array. The inner two dimensions constitute register blocks. These register blocks are transposes of the register blocks described in Section 5. However, data in this new format can be addressed with memory stride of one so that twelve streams mentioned above become three. Furthermore, the L0/L1 interface problem disappears as data will now enter L1 (or bypass it completely) in a way that is optimal for its entrance into the L0 cache; see [11].

7 Conclusion/Summary

We summarize the results of [13] for the recursive and iterative programming styles. Our recursive microkernel work led us to the following conclusions.

– The performance of the program obtained by following the canonical recipe (recursive outer control structure and recursive microkernel) is substantially lower than the near-peak performance of highly optimized iterative codes produced by ATLAS or in vendor BLAS. The best we were able to obtain was 63% of peak on the Itanium 2; on the UltraSPARC, performance was only 38% of peak.
– The microkernel is critical to overall performance. Producing a high performance recursive microkernel is a non-trivial job, and requires substantial programming effort.
– For generating code for the microkernel, using Belady's algorithm [2] followed by scheduling may not be optimal. Belady's algorithm minimizes the number of loads, but minimizing loads does not necessarily correlate to performance. An integrated register allocation and scheduling approach appears to perform better.
– Most compilers we used did not do a good job with register allocation and scheduling for long basic blocks. This problem has been investigated before. The situation is more muddied when processors perform register renaming and out-of-order instruction scheduling. The compiler research community needs to pay more attention to this problem.

Our iterative microkernel and blocking work led us to the following conclusions.

– Using the techniques in the BRILA compiler, we can generate iterative microkernels giving close to peak performance. They perform significantly better than recursive microkernels.
– Wrapping a recursive control structure around the iterative microkernel gives a program that performs reasonably well since it is able to block approximately for all levels of cache and block exactly for registers. If an iterative outer control structure is used, it is necessary to block for relevant levels of the memory hierarchy.
– To achieve performance competitive with hand-tuned kernels, minikernels need to do data prefetching. It is clear how to do this for an iterative outer control structure but we do not know how to do this for a recursive outer control structure.

References

1. R. C. Agarwal, F. G. Gustavson, M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, Vol. 38, No. 5, Sep. 1994, pp. 563,576.
2. Les A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* Vol. 5, No. 2, 1966, pp. 78,101.
3. S. Chatterjee et. al. Design and Exploitation of a High-performance SIMD Floating-point Unit for Blue Gene/L. *IBM Journal of Research and Development*, Vol. 49, No. 2-3, March-May 2005, pp. 377-391.
4. Matteo Frigo, Charles Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious Algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285. IEEE Computer Society 1999.
5. J. J. Dongarra, F. G. Gustavson, A. Karp. Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine. *SIAM Review*, Vol. 26, No. 1, Jan. 1984, pp. 91,112.
6. E. Elmroth, F. G. Gustavson, B. Kagstrom, and I. Jonsson. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. *SIAM Review*, Vol. 46, No. 1, Mar. 2004, pp. 3,45.
7. Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333, 1981.
8. J. A. Gunnels, F. G. Gustavson, G. M. Henry, R. A. van de Geijn. A Family of High-Performance Matrix Multiplication Algorithms. *Computational Science - Para 2004*, J. J. Dongarra, K. A. Madsen, J. Wasniewski, eds., Lecture Notes in Computer Science 3732. Springer-Verlag, pp. 256-265, 2004.
9. F. G. Gustavson. Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. *IBM Journal of Research and Development*, Vol. 41, No. 6, Nov. 1997, pp. 737,755.
10. F. G. Gustavson High Performance Linear Algebra Algorithms using New Generalized Data Structures for Matrices. *IBM Journal of Research and Development*, Vol. 47, No. 1, Jan. 2003, pp. 31,55.
11. F. G. Gustavson. Minimal Data Copy for Dense Linear Algebra Factorization. *Computational Science - Para 2006*, Bo Kagstom, E. Elmroth, eds., Lecture Notes in Computer Science xxxx. Springer-Verlag, pp. aaa-bbb, 200y.

12. N. Park, B. Hong, V. K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Trans. Parallel and Distributed Systems*, 14(7):640-654, 2003.
13. T. Roeder, K. Yotov, K. Pingali, J. Gunnels, F. Gustavson. The Price of Cache Obliviousness. *Department of Computer Science, University of Texas, Austin Technical Report CS-TR-06-43*, September 2006.
14. B. Sinharoy, R.N. Kalla, J.M Tendler, R.G. Kovacs, R.J. Eickemeyer, J.B. Joyner. POWER5 System Microarchitecture *IBM Journal of Research and Development*, Vol. 49, No. 4/5, July/Sep. 2005, pp. 505-521.
15. R. C. Whaley, A. Petitet, J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 2001(1-2), pp. 3-35.