On Reducing TLB Misses in Matrix Multiplication

FLAME Working Note #9

Kazushige Goto
Robert van de Geijn

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
{kgoto,rvdg}@cs.utexas.edu

November 1, 2002

Abstract

During the last decade, a number of projects have pursued the high-performance implementation of matrix multiplication. Typically, these projects organize the computation around an "inner kernel," $C = A^T B + C$, that keeps one of the operands in the L1 cache, while streaming parts of the other operands through that cache. Variants include approaches that extend this principle to multiple levels of cache or that apply the same principle to the L2 cache while essentially ignoring the L1 cache. The intent is to optimally amortize the cost of moving data between memory layers.

The approach proposed in this paper is fundamentally different. We start by observing that for current generation architectures, much of the overhead comes from Translation Look-aside Buffer (TLB) table misses. While the importance of caches is also taken into consideration, it is the minimization of such TLB misses that drives the approach. The result is a novel approach that achieves highly competitive performance on a broad spectrum of current high-performance architectures.

1 Introduction

It is somewhat surprising that after decades of research into the optimal implementation of matrix multiplication, papers on the subject still appear with great regularity. Matrix multiplication continues to be of importance because a broad range of high-performance packages that support directly or indirectly scientific computation depend, to a large degree, on the performance of the matrix multiplication kernel [3, 13, 28, 5]. New contributions continue to be made because the gap between the performance of the CPU and the bandwidth to the memory continues to widen and new architectural features are introduced into computers, which require new techniques or refinements of old techniques, for matrix multiplication.

Two observations are fundamental to our approach:

- The ratio between the rate at which floating point computation can be performed by the floating point unit(s) and the rate at which floating point numbers can be streamed from the L2 cache is typically relatively small.
- Thus, it is the cost for starting the streaming of data from the L2 cache that represents a significant overhead.
- A large component of the startup cost of the streaming of data cames from Translation Look-aside Buffer (TLB) misses since these inherently stall the CPU.

By taking these observations into account, the contribution of this paper is that by casting the matrix multiplication in terms of an inner kernel that performs the operation $C = \hat{A}^T B + C$, where $\hat{A}$ fills most of memory addressable by the TLB table and C and B are computed a few columns at a time,

- TLB misses can be largely avoided,
- the cost of the TLB misses that do occur can be amortized of a large amount of computation, and
- the cost of transposing submatrices so that the overall matrix multiplication can be cast into this inner kernel is amortized over a large amount of computation.

In practice, these observations lead to implementations that attain extremely high performance.

It can be argued that the exact nature of the new contribution of this paper is hard to identify. Much of what we present has been incorporated in one form or another in other implementations of matrix multiplication. It can also be argued that it is already known as street-wisdom and/or is incorporated in proprietary libraries that keep the details of the implementation a trade-secret. We would like to think that at the very least this paper exposes some of the issues explicitly and thereby makes a contribution the body of knowledge in this area. The fact that the method leads to consistently higher performance than achieved by competing implementations provides some support for this view.

The structure of this paper is as follows: In Section 2 we discuss research related to the high-performance implementation of matrix multiplication. Basic architectural considerations are given in Section 3. Observations that show the importance of the TLB are given in Section 4. These observations are translated to a practical implementation in Section 5. In Section 6 we report performance results from implementations on various architectures. Concluding remarks follow in the final section.

2 Related Work

The addition of a cache memory to vector architectures required library developers to reformulate linear algebra libraries that had been written in terms of vector operations.. To obtain high performance on these new machines, both vector operations and blocking to take advantage of the cache was necessary. IBM's ESSL library included block-based vector algorithms for a number of

linear equation solvers that were part of LINPACK [7], including LU and Cholesky based solvers for dense and banded matrices [21]. These implementations were based on highly optimized linear algebra routines that performed blocking together with an inner kernel that vectorized the linear algebra operation on blocks that fit in the cache memory.

It wasn't until the late 1980s that, with the introduction of the Cray 2, which also combined vector processing with a cache memory, there was a strong impetus in the linear algebra library community to standardize a new interface to a set of matrix-matrix operations, the level-3 BLAS [8]. The primary purpose of this new set of routines was to support newly proposed libraries such as LAPACK [6, 2, 3]. By casting the bulk of computation in terms of matrix-matrix operations, which perform $O(n^3)$ operations on $O(n^2)$ data, blocks of data could be moved in and out of the data cache while amortizing the cost of this movement over a large number of computations. The substantial task of providing all levels of BLAS was pushed onto the vendors. The reward was that numerically stable libraries like LAPACK then provided high-performance across a large variety of architectures.

By the early 1990s, it was recognized that as architectures were becoming increasingly complex the task of providing a complete set of (especially level-3) BLAS was becoming a substantial burden on the vendors. Fortunately, it was shown that high-performance level-3 BLAS could be coded to be portable by casting these operations in terms of matrix multiplication [23, 16, 24, 13]. This reduced the cost of implementing the level-3 BLAS to the cost of implementing matrix multiplication. Next, it was recognized that by combining a blocking strategy with a carefully crafted inner kernel, which performs matrix multiplication with blocks that are roughly of a size so that they fit in the cache memory, the cost of implementing the level-3 BLAS could be reduced to the cost of implementing this inner kernel. At IBM the idea of designing the architecture for this approach to coding the matrix multiply and other algorithms, referred to as *Algorithms and Architectures*, was both expounded and applied to the development of the IBM POWER2 architecture in conjunction with the ESSL library for that architecture [1]. By also designing compilers specifically for this combination of algorithms and architecture, the implementations of the BLAS could be coded in FORTRAN rather than in assembly code.

By the late 1990s architectures with multiple levels of cache memory were being introduced. With it came a recognition that the implementation of matrix multiplication for a given architecture had become, and would remain, a formidable task [12]. Based on the work at IBM that coded such operations in FORTRAN, the PHiPAC project at Berkeley pursued the portable implementation of matrix multiplication in a high-level language, C [4]. The idea behind that project was to automatically generate code so that in combination with an exhaustive search, the optimal blocking of the operands and optimal ordering of the loops could be discovered. The different blocking schemes were intended to automatically detect optimal blockings for the different caches while the different loop orderings would automatically detect how the movement of blocks between memory layers could be best amortized over computation. In addition, the inner kernel was automatically generated so that the number of registers and depth of pipelines could be detected. At the expense of an optimization process that often took days or even weeks to complete, remarkable performance was observed.

The ATLAS [29] project at the University of Tennessee refined the techniques developed as part

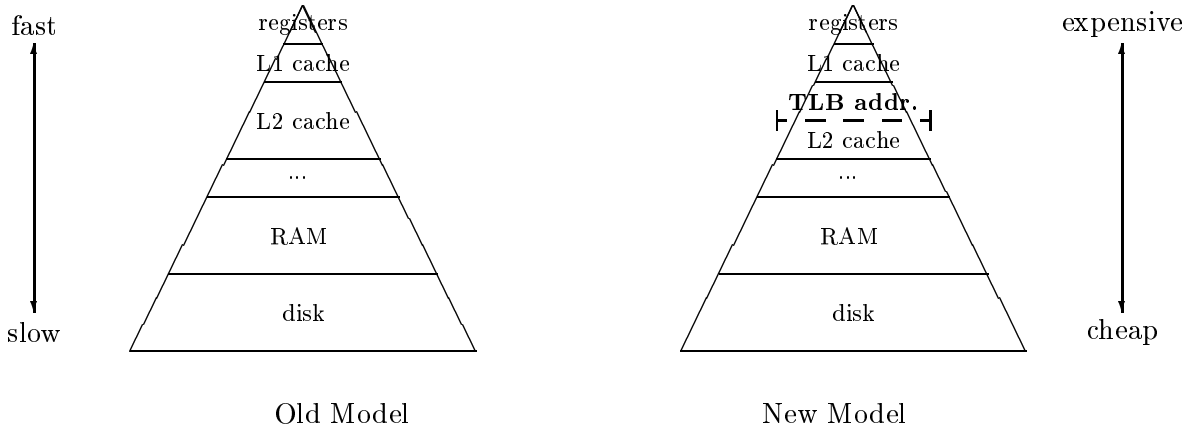

Figure 1: The hierarchical memories viewed as a pyramid. Under the new model, the memory that is addressable by the TLB is explicitly exposed.

of the PHiPAC project by constraining the number of different implementations that are generated as part of the search process. As a result, the optimization process completes more quickly, typically in a matter of hours.

In a recent paper [14] a family of algorithms based on a model of the memory hierarchy was introduced. The model predicts, and preliminary experiments with an implementation for the Intel Pentium (R) III processor show, that at a given level of the memory the blocking of the matrices and order of the loops is dictated by the shapes of the operands together with the size of memory layer one level above (in the pyramid).

Recently, algorithms that automatically block for caches by formulating the algorithms to be recursive have received a great deal of attention for matrix multiplication and many other important computations such as matrix factorizations [11, 17, 29, 22, 15, 26, 19]. Others have focused on (also) applying "recursion" to produce new data formats for matrices, instead of the traditional FORTRAN and C data structures [27]. Our view is that recursion is very powerful and excellent results are obtainable. The techniques presented in this paper are in some sense orthogonal to those addressed by recursion in data storage and algorithm implementation.

3 Basic Architectural Considerations

In this section we present, at a high level of abstraction, some of the architectural features of a typical modern microprocessor.

The memory hierarchy of a modern microprocessor is often viewed as the pyramid given in Fig. 1. At the top of the pyramid, there are the processor registers, with extremely fast access. At the bottom, there are disks and even slower media. As one goes down the pyramid, the amount of memory increases as does the time required to access that that memory, while the financial cost of memory decreases.

A second architectural consideration relates to the page management system. A typical modern architecture uses virtual memory so that the size of usable memory is not constrained by the size of the physical memory. Memory is partitioned in pages of some (often fixed) prescribed size. A table, referred to as the *page table* maps virtual addresses to physical addresses and keeps track of whether as page is in memory or on disk. The problem is that this table itself could be large (many Mbytes) which hampers speedy translation of virtual addresses to physical addresses. To overcome this, a smaller table, the Translation Look-aside Buffer (TLB), that stores information about the most recently used pages, is kept. Whenever a virtual address is found in the TLB, the translation is fast. Whenever it is not found (a TLB miss occurs), the page table is consulted and the resulting entry is moved from the page table to the TLB.

The most significant difference between a cache miss and a TLB miss is that a cache miss does not necessarily stall the CPU. A small number of cache misses can be tolerated by using algorithmic prefetching techniques as long as the data can be read fast enough from the memory where it does exist and arrives at the CPU by the time it is needed for computation. A TLB miss, by contrast, causes the CPU to stall until the TLB has been updated with the new address. In other words, prefetching can mask a cache miss but not a TLB miss.

4 Emphasizing the TLB

Consider the multiplication $C = AB + C$. Partition

$$(1)\quad C = \left(\begin{array}{c|c|c} C_{11} & \cdots & C_{1N} \\ \hline \vdots & & \vdots \\ \hline C_{M1} & \cdots & C_{MN} \end{array}\right), A = \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline \vdots & & \vdots \\ \hline A_{M1} & \cdots & A_{MK} \end{array}\right), \text{ and } B = \left(\begin{array}{c|c|c} B_{11} & \cdots & B_{1N} \\ \hline \vdots & & \vdots \\ \hline B_{K1} & \cdots & B_{KN} \end{array}\right)$$

where the partitionings are conformal so that

$$C_{ij} = \sum_{p=1}^{K} A_{ip}B_{pj} + C_{ij}.$$

The following loop ordering will compute the multiplication:

Algorithm 1

for $i = 1 : M$
  for $p = 1 : K$
    for $j = 1 : N$
      $C_{ij} = A_{ip}B_{pj} + C_{ij}$
    endfor
  endfor
endfor

A typical approach to optimizing matrix multiplication starts by writing an inner kernel to compute $C_{ij} = A_{ip}B_{pj} + C_{ij}$. This approach has the property that the CPU attains near-optimal

performance when A_{ip} remains in the L1 cache and elements of C_{ij} and B_{pj} are streamed for a lower level in the memory pyramid. The dimensions of A_{ip} are optimized so that this inner kernel attains the best performance. Finally, Some loop is created to compute all submatrices of C. Beyond this basic approach, there are some options. It is often beneficial, especially if A_{ip} is embedded in a matrix with a large leading dimension, to pack it into contiguous memory so that TLB misses are reduced. Also, it is often beneficial to transpose A_{ip} so that accesses to memory are contiguous when inner-products of columns of A_{ip}^T and B_{pj} are computed to update elements of C_{ij}.

Let us present Algorithm 1 as

Algorithm 2

for $i = 1 : M$
  for $p = 1 : K$
    $\left(\begin{array}{c|c|c} C_{i1} & \cdots & C_{iN} \end{array} \right) = A_{ip} \left(\begin{array}{c|c|c} B_{p1} & \cdots & B_{pN} \end{array} \right) + \left(\begin{array}{c|c|c} C_{i1} & \cdots & C_{iN} \end{array} \right)$
  endfor
endfor

Let us make the following assumptions and observations. Notice that we do not proclaim these assumptions and observations to reflect the absolute truth. They will provide a point of departure for discussion.

1. If we can optimize the individual computation

   $$\left(\begin{array}{c|c|c} C_{i1} & \cdots & C_{iN} \end{array} \right) = A_{ip} \left(\begin{array}{c|c|c} B_{p1} & \cdots & B_{pN} \end{array} \right) + \left(\begin{array}{c|c|c} C_{i1} & \cdots & C_{iN} \end{array} \right), \tag{2}$$

   we are in good shape.

2. In order to optimize (2) it is beneficial to transpose $\hat{A} = A_{ip}^T$ and compute

   $$\left(\begin{array}{c|c|c} C_{i1} & \cdots & C_{iN} \end{array} \right) = \hat{A}^T \left(\begin{array}{c|c|c} B_{p1} & \cdots & B_{pN} \end{array} \right) + \left(\begin{array}{c|c|c} C_{i1} & \cdots & C_{iN} \end{array} \right)$$

   instead. This observation comes from the fact that this allows inner products of columns of A_{ip} and B_{pj} to be computed while accessing memory contiguously. It also prevents severe thrashing of the L1 cache.

3. It is important to be able to complete a loop through all entries of $\hat{A}$ without creating a major bubble in the stream of data and computation. One way to satisfy this assumption is to store $\hat{A}$ contiguously while ensuring that accessing $\hat{A}$ does not create a TLB miss.

4. A prominent overhead comes from the cost of accessing C_{ij} and B_{pj} the first time as part of the computation $C_{ij} = \hat{A}^T B_{pj} + C_{ij}$. We will assume that this cost includes a startup (latency) cost as well as a cost proportional to the size of B_{pj}. A large part of the latency cost lies with the cost of the TLB misses associated with the first time that C_{ij} and B_{pj} are accessed. By picking B_{pj} and C_{ij} to have a relatively large row dimension, this startup cost is amortized over many elements of C_{ij} and B_{pj}. However, it is important to ensure that B_{pj} fits in the L1 cache so that the streaming of data from

5. If data is streamed so that the CPU does not stall, a second overhead that reduces performance comes from transposing (or packing) A_{ip} into $\hat{A}$.

The conclusion is that $\hat{A}$ should be relatively square and fill most of the L2 cache. Submatrices C_{ij} and B_{pj} should be relatively narrow since this means fewer entries of the TLB are devoted to those submatrices.

5 A Practical Approach

Let us examine how the above considerations affect the implementation of matrix multiplication on a current generation microprocessor like the Intel Pentium (R) 4.

We observe that on such architecture the bandwidth between the L2 cache and the registers is such that in the time it takes to load a floating point number from the L2 cache into a register, only a few floating point operations (often only a single one) can be performed once a pipeline has been established. Let us, for the sake of argument, assume that once pipelines are filled, the ratio between the cost of such a load and computation is actually one. Now, provided pipelines can be kept full, the following approach will attain high performance:

1. Partition C, A, and B as in (1), but pick C_{ij} and B_{pj} to be comprised of only a single column:

$$(3) C = \left(\begin{array}{c|c|c} c_{11} & \cdots & c_{1n} \\ \hline \vdots & & \vdots \\ \hline c_{M1} & \cdots & c_{Mn} \end{array}\right), A = \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline \vdots & & \vdots \\ \hline A_{M1} & \cdots & A_{MK} \end{array}\right), \text{ and } B = \left(\begin{array}{c|c|c} b_{11} & \cdots & b_{1n} \\ \hline \vdots & & \vdots \\ \hline b_{K1} & \cdots & b_{Kn} \end{array}\right).$$

   Notice that elements of c_{ij} and b_{pj} will be contiguous in memory.

2. Consider the computation

$$\left(\begin{array}{c|c|c} c_{i1} & \cdots & c_{in} \end{array}\right) = A_{ip} \left(\begin{array}{c|c|c} b_{p1} & \cdots & b_{pn} \end{array}\right) + \left(\begin{array}{c|c|c} c_{i1} & \cdots & c_{in} \end{array}\right).$$

   Let us implement this by first transposing $\hat{A} = A_{ip}^T$ and then computing

$$(4) \qquad \left(\begin{array}{c|c|c} c_{i1} & \cdots & c_{in} \end{array}\right) = \hat{A}^T \left(\begin{array}{c|c|c} b_{p1} & \cdots & b_{pn} \end{array}\right) + \left(\begin{array}{c|c|c} c_{i1} & \cdots & c_{in} \end{array}\right).$$

3. *If*
   (a) $\hat{A}$ is packed to be in contiguous memory,
   (b) The transposition of A_{ip}, $\hat{A} = A_{ip}^T$, is carefully ordered,
   (c) The first element of $\hat{A}$ is aligned to a page,
   (d) $\hat{A}$ and, for all j, c_{ij}, $c_{i(j+1)}$, b_{pj}, and $b_{p(j+1)}$ together do not overflow the TLB table,
   (e) $\hat{A}$ fits in the L2 cache, and
   (f) (4) is computed by the loop

for $j = 1 : n$
$\quad c_{ij} = \hat{A}^T b_{pj} + c_{ij}$
endfor

then, **in principle**,

- $\hat{A}$ will be loaded into the L2 cache, and the TLB, during the transposition $\hat{A} = A_{ip}^T$.
- Once the pages corresponding to $\hat{A}$ have been loaded into the L2 cache and TLB they will remain there during the duration of the computation in (4).
- *The streaming of the data should allow the computation of each individual* $\hat{A}^T b_{pj} + c_{ij}$ *to achieve optimal performance.*

In practice, a few modification may have be made to the approach. For example, some TLB entries may be used by data associated with indexing or the code being executed.

Note 1 *If the number of floating point operations that can be performed during the loading of a floating point operation (once streaming is established) is greater than two, the bandwidth between the L2 cache and the registers becomes a bottleneck. Let us assume the ratio equals the integer R. Then the above scheme must be modified so that* b_{pj} *and* c_{ij} *consist of R columns. In this case, for every element of* $\hat{A}$ *that is loaded,* $2 * R$ *flops can be performed once that element reaches the registers. Notice that as R increases, the number of TLB entries devoted to* B_{pj} *and* C_{ij} *increases, which means that the size of* $\hat{A}$ *may have to be reduced. More specifically, the R should be chosen so that*

$$R \geq \frac{\textit{Rate in flops per cycle}}{2 \times \textit{Bandwidth in double words per cycle between L2 and registers}} \tag{5}$$

Note 2 *The number of registers that can be used for computation and prefetching play an important role in the proposed scheme. If there aren't enough registers to support pipeline streaming the scheme breaks down.*

Note 3 *The following steps can be used to determine approximations for the various parameters:*

1. *Determine the size of the TLB table, T.*
2. *Determine R of Note 1.*
3. *The number of TLB entries used for* $\hat{A}$ *should not exceed* $T - 4R$*. The reason for this is that generally,* C_{ij} *and* B_{pj} *have, together,* $2R$ *columns, which typically require* $2R$ *TLB entries (provided a column isn't split between two pages). In order to not corrupt any TLB entries devoted to* $\hat{A}$ *another* $2R$ *entries are required for when* $C_{i(j+1)}$ *and* $B_{p(j+1)}$ *are first accessed.*
4. *The size (footprint) of* $\hat{A}$ *is now picked to not exceed* $T - 4R$ *pages of memory.*
5. *Under the constraint given in Item 4, the row and column dimensions of* $\hat{A}$ *are determined experimentally.*