High-Performance Matrix Multiplication Algorithms for Architectures with Hierarchical Memories *

John A. Gunnels† Greg M. Henry‡ Robert A. van de Geijn§

June 20, 2001

Abstract

During the last half-decade, a number of research efforts have centered around the development of software for generating automatically tuned matrix multiplication kernels. These include the PHiPAC project and the ATLAS project. The software products of both projects employ brute force to search a parameter space for blockings that accommodate multiple levels of memory hierarchy. We take a different approach. Using a simple model of hierarchical memories we employ mathematics to determine a locally-optimal strategy for blocking matrices. The theoretical results show that, depending on the shape of the matrices involved, different strategies are locally-optimal. Rather than determining a blocking strategy at library generation time, the theoretical results show that ideally one should pursue a heuristic that allows the blocking strategy to be determined dynamically at run-time as a function of the shapes of the operands. When the resulting family of algorithms is combined with a highly optimized inner-kernel for a small matrix multiplication, the approach yields performance that is superior to that of methods that automatically tune such kernels. Preliminary results, for the Intel Pentium (R) III processor, support the theoretical insights.

1 Introduction

Research in the development of linear algebra libraries has recently shifted to the automatic generation and optimization of the matrix multiplication kernels. The idea is that many linear algebra operations can be implemented in terms of matrix multiplication [3, 11, 7] and that thus it is this operation that should be highly optimized on different platforms. Since the coding effort is considerable, especially when multiple layers of cache are involved, the general concensus is that this process should be automated.

In this paper, we develop a theoretical framework that (1) suggests a formula for the block sizes that should be used at each level of the memory hierarchy, and (2) restricts the possible loop orderings to a specific family of algorithms for matrix multiplication. Together, we show how to use these results to build highly optimized matrix multiplication implementations that utilize the caches in a locally-optimal fashion. The results could be equally well used to limit the search space that must be examined by packages that automatically tune such kernels.

*This work was partially performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The work was funded by the Remote Exploration and Experimentation Project (a part of the NASA High Performance Computing and Communications Program funded by the NASA Office of Space Science.)

†Department of Computer Sciences, The University of Texas, Austin, TX 78712, gunnels@cs.utexas.edu

‡Intel Corp., Bldg EY2-05, 5350 NE Elam Young Pkwy, Hillsboro, OR 97124-6461, greg.henry@intel.com

§Department of Computer Sciences, The University of Texas, Austin, TX 78712, rvdg@cs.utexas.edu

The current pursuit of highly optimized matrix kernels achieved by coding in a high-level programming language started with the implementation of the FORTRAN implementation of Basic linear Algebra Subprograms (BLAS) [5] for the IBM Power2 [1]. Subsequently, the PHiPAC project [4] at UC-Berkeley demonstrated that high-performance matrix multiplication kernels can be written in C and that code generators could be used to automatically generate many different blockings, allowing automatic tuning. Next, the Automatically Tuned Linear Algebra Software (ATLAS) project [12] at the University of Tennessee extended the ideas developed as part of the PHiPAC project by reducing the kernel that is called once matrices are massaged to be in the L1 cache into one specific case: $C = A^T B + \beta C$ for small matrices A, B, and C and reducing the space searched for optimal blockings. Furthermore it marketed the methodology allowing it to gain wide-spread acceptance and igniting the current craze in the linear algebra community towards automatically tuned libraries. Finally, there has been a considerable recent interest in recursive algorithms and recursive data structures. The idea here is that by recursively partitioning the operands blocks that fit in the different levels of the caches will automatically be encountered [9]. By storing matrices recursively, blocks that are encountered during the execution of the recursive algorithms will be in contiguous memory [2, 8, 10].

Other work closely related to this topic is discussed in other papers presented as part of this session of the conference.

2 Notation and Terminology

2.1 Special cases of matrix multiplication

The general form of a matrix multiply is $C \leftarrow \alpha AB + \beta C$ where C is $m \times n$, A is $m \times k$, and B is $k \times n$. We will use the following terminology when referring to a matrix multiply when two dimensions are large and one is small:

	Condition	Shape
Matrix-panel multiply	n is small	$\boxed{C} = \boxed{A}\,\boxed{B} + \boxed{C}$
Panel-matrix multiply	m is small	$\boxed{C} = \boxed{A}\,\boxed{B} + \boxed{C}$
Panel-panel multiply	k is small	$\boxed{C} = \boxed{A}\,\boxed{B} + \boxed{C}$

The following observation will become key to understanding concepts encountered in the rest of the paper: Partition $X = \left(\begin{array}{c|c|c} X_1 & \cdots & X_{N_X} \end{array} \right) = \left(\begin{array}{c} \hat{X}_1 \\ \hline \vdots \\ \hline \hat{X}_{M_X} \end{array} \right)$ for $X \in \{A, B, C\}$, where C_j is $m \times n_j$, $\hat{C}_i$ is $m_i \times n$, A_p is $m \times k_p$, $\hat{A}_i$ is $m_i \times k$, B_j is $k \times n_j$, and $\hat{B}_p$ is $k_p \times n$. Then $C \leftarrow AB + C$ can be achieved by

multiple matrix-panel multiplies:	$C_j \leftarrow AB_j + C_j$ for $j = 1, \ldots, N_C$	$\left(C_1 \mid C_2 \mid C_3 \right) \mathrel{+}= \left(A \right) \left(B_1 \mid B_1 \mid B_1 \right)$
multiple panel-matrix multiplies:	$\hat{C}_i \leftarrow \hat{A}_i B + \hat{C}_i$ for $i = 1, \ldots, M_C$	$\begin{pmatrix} \hat{C}_1 \\ \hat{C}_2 \\ \hat{C}_3 \end{pmatrix} \mathrel{+}= \begin{pmatrix} \hat{A}_1 \\ \hat{A}_2 \\ \hat{A}_3 \end{pmatrix} \left(B \right)$
multiple panel-panel multiplies	$C \leftarrow \sum_p^{N_A} A_p \hat{B}_p + C$	$\left(C \right) \mathrel{+}= \left(A_1 \mid A_2 \mid A_3 \right) \begin{pmatrix} \hat{B}_1 \\ \hat{B}_2 \\ \hat{B}_3 \end{pmatrix}$

2.2 A cost model for hierarchical memories

The memory hierarchy of a modern microprocessor is often viewed as a pyramid: 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, while the cost of memory decreases, the amount of memory increases along with the time required to access that that memory.

We will model the above-mentioned hierarchy naively as follows: (1) The memory hierarchy consists of H levels, indexed $0, \ldots, H-1$. Level 0 corresponds to the registers. We will often denote the ith level by L_i. Notice that on a typical current architecture L_1 and L_2 correspond the level 1 and level 2 data caches and L_3 corresponds to RAM. (2) Level h of the memory hierarchy can store S_h floating point numbers. Generally $S_0 \leq S_1 \leq \cdots \leq S_{H-1}$. (3) Loading a floating point number stored in level $h+1$ to level h costs time ρ_h. We will assume that $\rho_0 < \rho_1 < \cdots < \rho_{H-1}$. (4) Storing a floating point number from level h to level $h+1$ costs time σ_h. We will assume that $\sigma_0 < \sigma_1 < \cdots < \sigma_{H-1}$. (5) If $m_h \times n_h$ matrix C, $m_h \times k_h$ matrix A, and $k_h \times n_h$ matrix B are all stored in level h of the memory hierarchy then forming $C \leftarrow AB + C$ costs time $2 m_h n_h k_h \gamma_h$. (Notice that γ_h will depend on m_h, n_h, and k_h).

3 Building-blocks for matrix multiplication

Consider the matrix multiplication $C \leftarrow AB + C$ where $m_{h+1} \times n_{h+1}$ matrix C, $m_{h+1} \times k_{h+1}$ matrix A, and $k_{h+1} \times n_{h+1}$ matrix B are all stored in L_{h+1}. Let us assume that somehow an efficient matrix multiplication kernel exists for matrices stored in L_h. In this section, we develop three distinct approaches for matrix multiplication kernels for matrices stored in L_{h+1}.

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 C_{ij} is $m_h \times n_h$, A_{ip} is $m_h \times k_h$, and B_{pj} is $k_h \times n_h$. The objective of the game will be to determine optimal m_h, n_h, and k_h.

3.1 Multiple panel-panel multiplies in L_h

Noting that $C_{ij} \leftarrow \sum_{p=1}^{K} A_{ip} B_{pj} + C_{ij}$, let us consider the algorithm in Fig. 1 for computing the matrix multiplication. In that figure the costs of the various operations are shown to the right. The order of the outer-most loops is irrelevant to the analysis.

Algorithm 1

for $j = 1, \ldots, N$	
for $i = 1, \ldots, M$	
Load C_{ij} from L_{h+1} to L_h.	$m_h n_h \rho_h$
for $p = 1, \ldots, K$	
Load A_{ip} from L_{h+1} to L_h.	$m_h k_h \rho_h$
Load B_{pj} from L_{h+1} to L_h.	$k_h n_h \rho_h$
Update $C_{ij} \leftarrow A_{ip} B_{pj} + C_{ij}$	$2 m_h n_h k_h \gamma_h$
endfor	
Store C_{ij} from L_h to L_{h+1}	$m_h n_h \sigma_h$
endfor	
endfor	

Figure 1: Multiple panel-panel multiply based blocked matrix multiplication.

The cost for updating C is given by

$$m_{h+1} n_{h+1} (\rho_h + \sigma_h) + m_{h+1} n_{h+1} k_{h+1} \frac{\rho_h}{n_h} + m_{h+1} n_{h+1} k_{h+1} \frac{\rho_h}{m_h} + 2 m_{h+1} n_{h+1} k_{h+1} \gamma_h$$

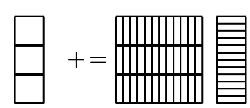
Since it also equals $2 m_{h+1} n_{h+1} k_{h+1}$, solving for γ_{h+1}, the effective cost per floating point operation at level L_{h+1}, yields

$$\gamma_{h+1}^{PP} = \frac{\rho_h + \sigma_h}{2 k_{h+1}} + \frac{\rho_h}{2 n_h} + \frac{\rho_h}{2 m_h} + \gamma_h$$

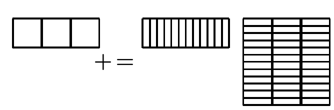
The question now is how to find the m_h, n_h, and k_h that minimize γ_{h+1} under the constraint that C_{ij}, A_{ik} and B_{kj} all fit in L_h, i.e., $m_h n_h + m_h k_h + n_h k_h \leq S_h$. The smaller k_h, the more space in L_h can be dedicated to C_{ij} and thus the smaller the fractions ρ_h / m_h and ρ_h / n_h can be made. A good strategy is thus to let essentially all of L_h be dedicated to C_{ij}, i.e., $m_h n_h \approx S_h$. The minimum is then attained when essentially $m_h \approx n_h \approx \sqrt{S_h}$.

Notice that it suffices to have $m_{h+1} = m_h$ or $n_{h+1} = n_h$ for the above cost of γ_{h+1} to be achieved. Thus, the above already for the special cases

$$\left(\begin{array}{c} C_{11} \\ \hline\hline \vdots \\ \hline\hline C_{M1} \end{array} \right) += \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline\hline \vdots & & \vdots \\ \hline\hline A_{M1} & \cdots & A_{MK} \end{array} \right) \left(\begin{array}{c} B_{11} \\ \hline \vdots \\ \hline B_{K1} \end{array} \right) \tag{2}$$



$$\left(\begin{array}{c||c||c} C_{11} & \cdots & C_{1N} \end{array} \right) += \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \end{array} \right) \left(\begin{array}{c||c||c} B_{11} & \cdots & B_{1N} \\ \hline \vdots & & \vdots \\ \hline B_{K1} & \cdots & B_{KN} \end{array} \right) \tag{3}$$



Here the distance between single/thin lines is k_h and between double/thick lines $m_h = n_h$, where k_h is much smaller than m_h and n_h.

The inner-most loop in Alg. 1 implements multiple panel-panel multiplies since k_h is small relative to m_h and n_h. Hence the name of this section.

3.2 Multiple matrix-panel multiplies in L_h

Moving the loops over l and i to the outside we obtain the algorithm in Fig. 2(left). Performing an

Algorithm 2

```
for p = 1, ..., K
    for i = 1, ..., M
        Load A_ip from L_{h+1} to L_h.
        for j = 1, ..., N
            Load C_ij from L_{h+1} to L_h.
            Load B_pj from L_{h+1} to L_h.
            Update C_ij ← A_ip B_pj + C_ij
            Store C_ij from L_h to L_{h+1}
        endfor
    endfor
endfor
```

Algorithm 3

```
for j = 1, ..., N
    for p = 1, ..., K
        Load B_pj from L_{h+1} to L_h.
        for i = 1, ..., M
            Load C_ij from L_{h+1} to L_h.
            Load A_ip from L_{h+1} to L_h.
            Update C_ij ← A_ip B_pj + C_ij
            Store C_ij from L_h to L_{h+1}
        endfor
    endfor
endfor
```

Figure 2: Multiple matrix-panel (left) and panel-matrix (right) multiply based blocked matrix multiplication.

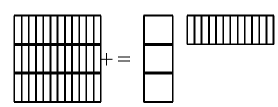
analysis similar to that given in Section 3.1 the effective cost of a floating point operation is now given by

$$\gamma_{h+1}^{MP} = \frac{\rho_h}{2n_{h+1}} + \frac{\rho_h + \sigma_h}{2k_h} + \frac{\rho_h}{2m_h} + \gamma_h \tag{4}$$

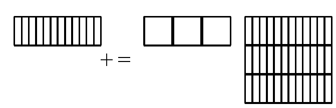
Again, the question is how to find the m_h, n_h, and k_h that minimize γ_{h+1} under the constraint that C_{ij}, A_{ik} and B_{kj} all fit in L_h, i.e., $m_h n_h + m_h k_h + n_h k_h \le S_h$. Note that the smaller n_h, the more space in L_h can be dedicated to A_{il} and thus the smaller the fractions $(\rho_h + \sigma_h)/2k_h$ and $\rho_h/2m_h$ can be made. A good strategy is thus to let essentially all of L_h be dedicated to A_{il}, i.e., $m_h k_h \approx S_h$. The minimum is then attained when essentially $m_h \approx k_h \approx \sqrt{S_h}$.

Notice that it suffices to have $m_{h+1} = m_h$ or $k_{h+1} = k_h$ for the above cost of γ_{h+1} to be achieved. In other words, the above holds for the special cases

$$\left(\begin{array}{c|c|c} C_{11} & \cdots & C_{1N} \\ \hline\hline \vdots & & \vdots \\ \hline\hline C_{M1} & \cdots & C_{MN} \end{array}\right) += \left(\begin{array}{c} A_{11} \\ \hline\hline \vdots \\ \hline\hline A_{M1} \end{array}\right) \left(\begin{array}{c|c|c} B_{11} & \cdots & B_{1N} \end{array}\right) \tag{5}$$



$$\left(\begin{array}{c||c||c} C_{11} & \cdots & C_{1N} \end{array}\right) += \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \end{array}\right) \left(\begin{array}{c||c||c} B_{11} & \cdots & B_{1N} \\ \hline \vdots & & \vdots \\ \hline B_{K1} & \cdots & B_{KN} \end{array}\right) \tag{6}$$



The inner-most loop in Alg. 2 implements multiple matrix-panel multiplies since n_h is small relative to m_h and k_h. Thus the name of this section.

3.3 Multiple panel-matrix multiplies in L_h

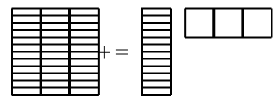
Finally, moving the loops over p and j to the outside we obtain the algorithm given in Fig. 2(right). This time, the effective cost of a floating point operation is given by

$$\gamma_{h+1}^{PM} = \frac{\rho_h}{2m_{h+1}} + \frac{\rho_h + \sigma_h}{2k_h} + \frac{\rho_h}{2n_h} + \gamma_h \tag{7}$$

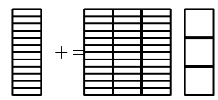
Again, the question is how to find the m_h, n_h, and k_h that minimize γ_{h+1} under the constraint that C_{ij}, A_{ik} and B_{kj} all fit in L_h, i.e., $m_h n_h + m_h k_h + n_h k_h \le S_h$. Note that the smaller m_h, the more space in L_h can be dedicated to B_{pj} and thus the smaller the fractions $(\rho_h + \sigma_h)/2k_h$ and $\rho_h/2n_h$ can be made. A good strategy in this case is to dedicate essentially all of L_h to B_{pj}, i.e., $n_h k_h \approx S_h$. The minimum is then attained when essentially $n_h \approx k_h \approx \sqrt{S_h}$.

Notice that it suffices to have $n_{h+1} = n_h$ and/or $k_{h+1} = k_h$ for the above cost of γ_{h+1} to be achieved. In other words, the above holds for the special cases

$$(8) \qquad \left(\begin{array}{c|c|c} C_{11} & \cdots & C_{1N} \\ \hline\hline \vdots & & \vdots \\ \hline\hline C_{M1} & \cdots & C_{MN} \end{array}\right) \mathrel{+}= \left(\begin{array}{c} A_{11} \\ \hline\hline \vdots \\ \hline\hline A_{M1} \end{array}\right) \left(\begin{array}{c|c|c} B_{11} & \cdots & B_{1N} \end{array}\right)$$



$$(9) \qquad \left(\begin{array}{c} C_{11} \\ \hline\hline \vdots \\ \hline\hline C_{M1} \end{array}\right) \mathrel{+}= \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline\hline \vdots & & \vdots \\ \hline\hline A_{M1} & \cdots & A_{MK} \end{array}\right) \left(\begin{array}{c} B_{11} \\ \hline\hline \vdots \\ \hline\hline B_{K1} \end{array}\right)$$



an observation that will become important later.

3.4 Summary

The conclusions to draw from Sections 2.1 and 3.1–3.3 are: (1) There are three shapes of matrix multiplication that one expects to encounter at each level of the memory hierarchy: panel-panel, matrix-panel, and panel-matrix multiplication. (2) If one such shape is encountered at L_{h+1}, a locally-optimal approach to utilizing L_h will perform multiple instances with one of the other two shapes. (3) Given that multiple instances of a given shape are to be performed, the strategy is to move a submatrix of one of the three operands into L_h (we will call this the resident matrix in L_h), filling most of that layer, and to amortize the cost of this data movement by streaming submatrices from the other operands from L_{h+1} to L_h.

Interestingly enough, the shapes discussed are exactly those that we encountered when studying a class of matrix multiplication algorithms on distributed memory architectures [6]. This is not surprising, since distributed memory is just another layer in the memory hierarchy.

4 A Family of Algorithms

We now show how to turn the observations made in the previous section into a practical implementation.

High-performance implementations of matrix multiplication typically start with an "inner-kernel". This kernel carefully orchestrates the movement of data in and out of the registers and the computation under the assumption that one or more of the operands are in the L1 cache. For our implementation on the Intel Pentium (R) III processor, the inner-kernel performs the operation $C = A^T B + \beta C$ where 64×8 matrix A is kept in the L1 cache. Matrices B and C have a large number of columns, which we view as multiple-panels, with each panel of width one. Thus, our inner-kernel performs a multiple matrix-panel multiply (MMP) with a transposed resident matrix A. The technical reasons why this particular shape was selected go beyond the scope of this paper.

While it may appear that we thus only have one of the three kernels for operation in the L1 cache, notice that for the submatrices with which we compute at that level one can instead compute $C^T = B^T A + C^T$, reversing the role of A and B. This simple observation allows us to claim that we also have an inner-kernel that performs a multiple panel-matrix multiply (MPM).

Let us introduce a naming convention for a family of algorithms that perform the discussed algorithms at different levels of the memory hierarchy:

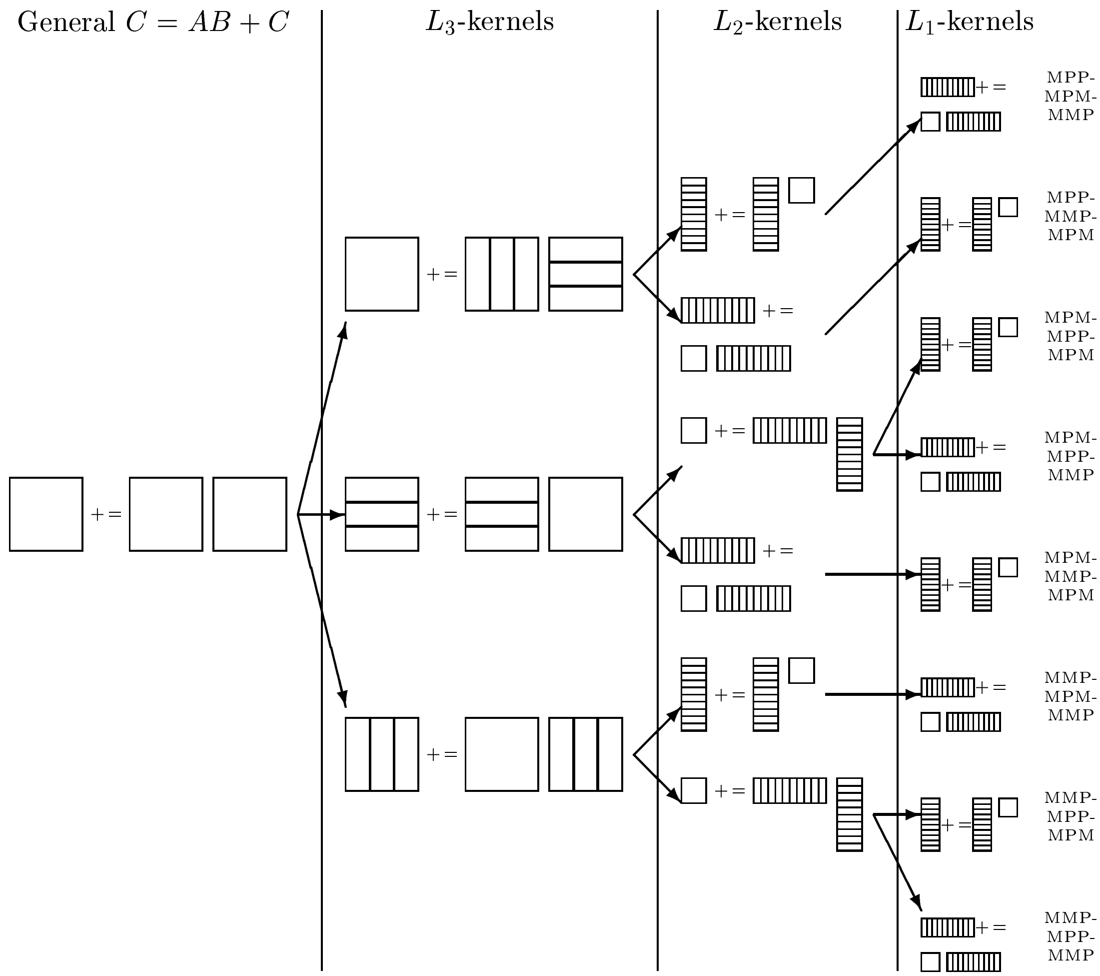


Figure 3: Possible algorithms for matrices in memory level L_3 given all L_2-kernels.

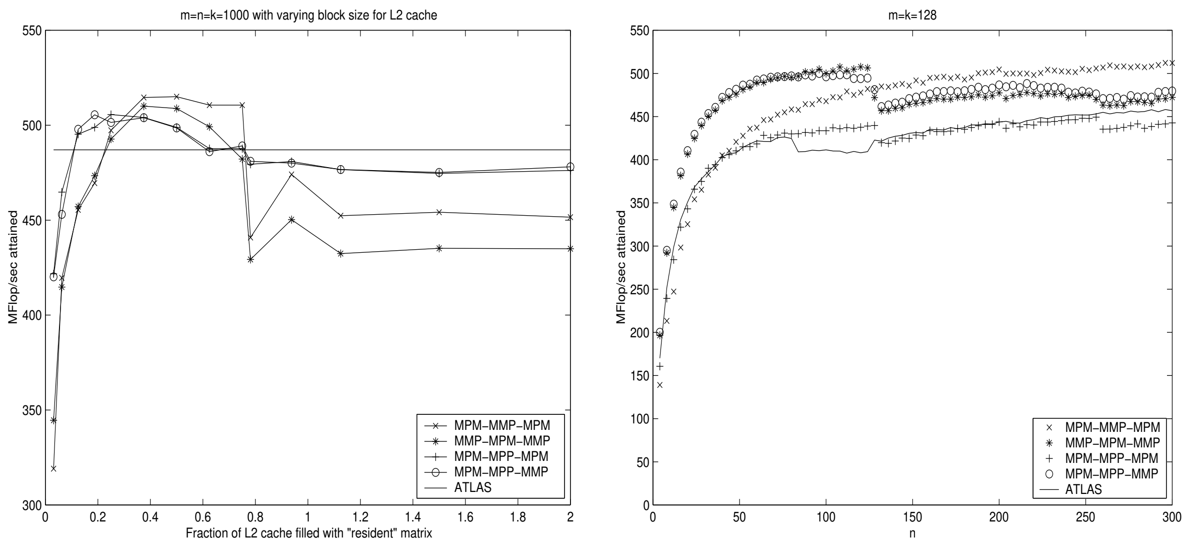
$$<\text{kernel at } L_3>\text{-}<\text{kernel at } L_2>\text{-}<\text{kernel at } L_1>.$$

For example MPP-MPM-MMP will indicate that the L_3-kernel uses multiple panel-panel multiplies, calls the L_2-kernel that uses multiple matrix-panel multiplies, which in turn calls the L_1-kernel that uses multiple panel-matrix multiplies. Given the constraint that only two of the possible three kernel algorithms are implemented at L_1, the tree of algorithms in Fig. 3 can be implemented.

5 Performance

In this section, we report performance attained by the different algorithms. Performance is reported by the rate of computations attained, in millions of floating point operations per second (MFLOPS/sec). For the usual matrix dimensions m, n, and k, we use the operation count $2mnk$ for the matrix multiplication. We tested performance of the operation $C = C - AB$ ($\alpha = -1$ and $\beta = 1$) since this is the case most frequently encountered when matrix multiplication is used in libraries such as LAPACK.

We report performance on an Intel Pentium (R) III (650 MHz) processor with a 16 Kbyte L1



(a) (b)

Figure 4: Left: Performance for fixed dimensions $m = n = k = 1000$ as a function of the size of the resident matrix in the L2 cache. Right: Performance as a function of n when $m = k = 128$ so that A fits in the L2 cache.

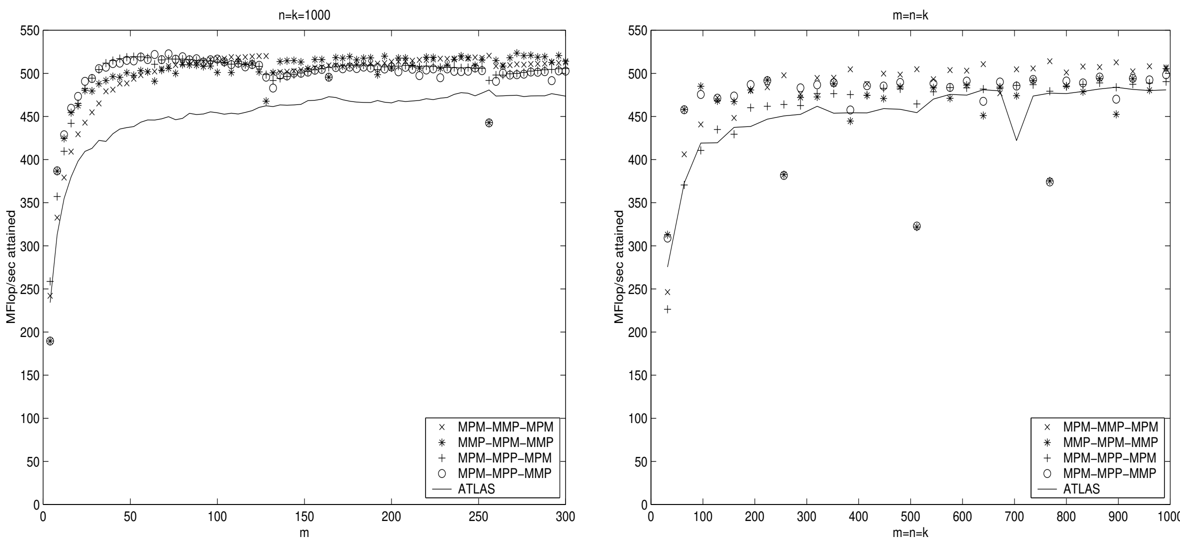
data cache and a 256 Kbyte L2 cache running RedHat Linux 6.2. The inner-kernel, which perform the operation $C \leftarrow A^T B + \beta C$ with 64×8 matrix A and $64 \times k$ matrix B, was hand-coded using Intel Streaming SIMD Extensions (TM) (SSE). In order to keep the graphs readable, we only report performance for four of the eight possible algorithms. For reference, we report performance of the matrix multiply from ATLAS R3.2 for this architecture.

Our first experiment is intended to demonstrate that the block size selected for the matrix that remains resident in the L2 cache has a clear effect on the overall performance of the matrix multiplication routine. In Fig. 4(a) we report performance attained as a function of the fraction of the L2 cache filled with the resident matrix when a matrix multiplication with $k = m = n = 1000$ is executed. This experiment tests our theory that reuse of data in the L2 cache impacts overall performance as well as our theory that the resident matrix should occupy "most" of the L2 cache. Note that performance improves as a larger fraction of the L2 cache is filled with the resident matrix. Once the resident matrix fills more than half of the L2 cache, performance starts to deminish. This is consistent with the theory which tells us that some of the cache must be used for the matrices that are being streamed from main memory. Once more than 3/4 of the L2 cache is filled with the resident matrix, performs drops significantly. This is consistent with the scenario where parts of the other matrices start evicting parts of the resident matrix from the L2 cache.

Based on the above experiment, we fix the block size for the resident matrix in the L2 cache to 128×128, which fills exactly half of this cache, for the remaining experiments.

The next experiment shows that the cost of moving a submatrix into the L2 cache and then amortizing the cost of this memory operation over as much computation as possible is indeed observable in practice. In Fig. 4(b) we fix $m = k = 128$ and vary n. Notice that the curve for `MPM-MMP-MPM`, which keeps 128×128 matrix A resident in the L2 cache, improves smoothly in performance as n increases. It is also obvious from this graph that depending on the matrix size, different algorithms attain superior performance.

In Fig. 5(a) we show performance as a function of m when n and k are fixed to be large. There is more information in this graph than we can discuss in this paper. Notice for example



(a) (b)

Figure 5: Left: Performance when $n = k = 1000$ as a function of m. Right: Performance when all operands are square.

that performance of the algorithm that performs multiple panel-matrix multiplies in the L_3 cache and multiple matrix-panel multiplies in the L_2 cache, `MPM_MMP_MPM`, increases as m increases to a multiple of 128. This is consistent with the theory.

In Fig. 5(b) we show the obligatory graph that reports performance for square matrices. Notice the interesting performance dip for multiples of 256. Whenever matrices have a leading dimension of a multiple of 256 there appears to be interference due to problems with the set associative cache or Table Look-aside Buffer (TLB) infractions.

6 Conclusion

In this paper, theoretical insight was used to motivate a family of algorithms for matrix multiplication on hierarchical memory architectures. The approach attempts to amortized the cost of moving data between memory layers in a fashion that is locally-optimal. Preliminary experimental results on the Intel Pentium (R) III processor appear to support the theoretical results.

Many questions regarding this subject are not addressed in this paper: For example, the techniques can be trivially extended to the other cases of matrix multiplication: $C \leftarrow \alpha A^T B + \beta C$, $C \leftarrow \alpha A B^T + \beta C$, and $C \leftarrow \alpha A^T B^T + \beta C$ by transposing matrices at appropriate stages in the algorithm. Also, while we claim that given different matrix dimensions, m, n, and k, a different algorithm may be best we do not address how to choose from the different algorithms. We have developed simple heuristics that yield very satisfactory results. Space limitations do not allow us to elaborate here. Finally, experiments that support the theory performed on a number of different architectures are needed to draw definitive conclusions.

Clearly, our techniques can be used to greatly reduce the set of block sizes to be searches at each level of the memory hierarchy. Our techniques could be combined with techniques for automatically generating the inner-kernel and/or an automated search for the optimal block sizes.

For additional information visit `http://www.cs.utexas.edu/users/flame/ITXGEMM/`.

Acknowledgments: We thank Dr. Fred Gustavson for valuable feedback regarding this project.

References

[1] R.C. Agarwal, F.G. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38(5), Sept. 1994.

[2] Bjarne S. Andersen, Fred G. Gustavson, and Jerzy Wasniewski. A recursive formalation of Cholesky factorization of a matrix in packed storage. LAPACK Working Note 146 CS-00-441, University of Tennessee, Knoxville, May 2000.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide - Release 2.0.* SIAM, 1994.

[4] J. Bilmes, K. Asanovic, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing.* ACM SIGARC, July 1997.

[5] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[6] John Gunnels, Calvin Lin, Greg Morrow, and Robert van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 110–116, 1998.

[7] John A. Gunnels and Robert A. van de Geijn. Formal methods for high-performance linear algebra libraries. In Ronald F. Boisvert and Ping Tak Peter Tang, editors, *The Architecture of Scientific Software.* Kluwer Academic Press, 2001.

[8] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. In B. Kågström et al., editor, *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, volume 1541 of *Lecture Notes in Computer Science*, pages 195–206. Springer-Verlag, 1998.

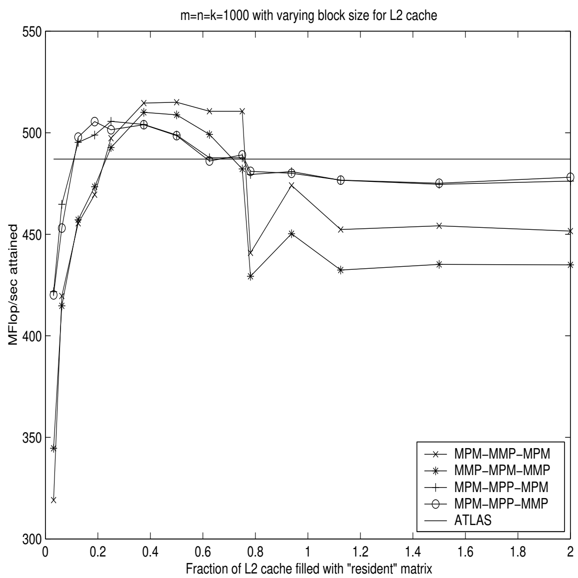
[9] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[10] Greg Henry. BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Cornell University, Feb. 1992.

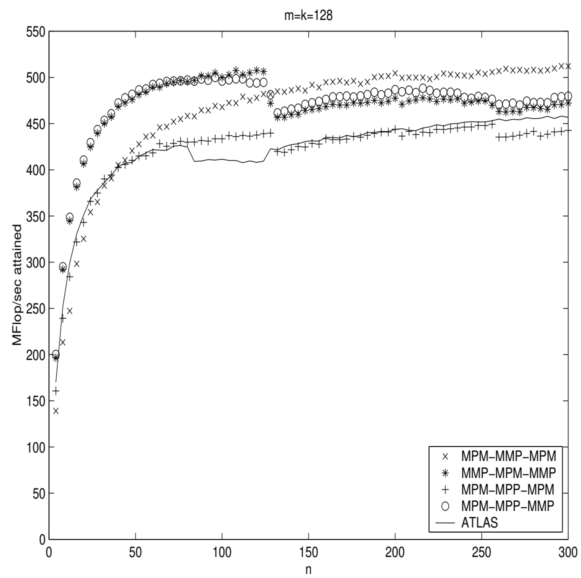
[11] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. Technical Report CS-95-315, Univ. of Tennessee, Nov. 1995.

[12] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC98*, Nov. 1998.

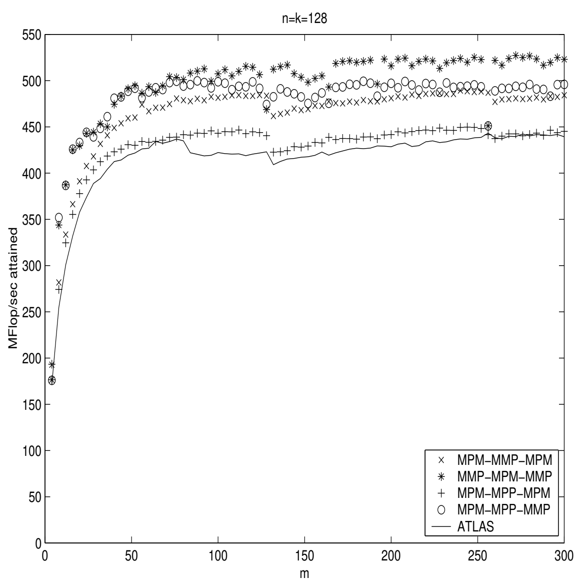
A Additional Performance Graphs for the Pentium (R) III



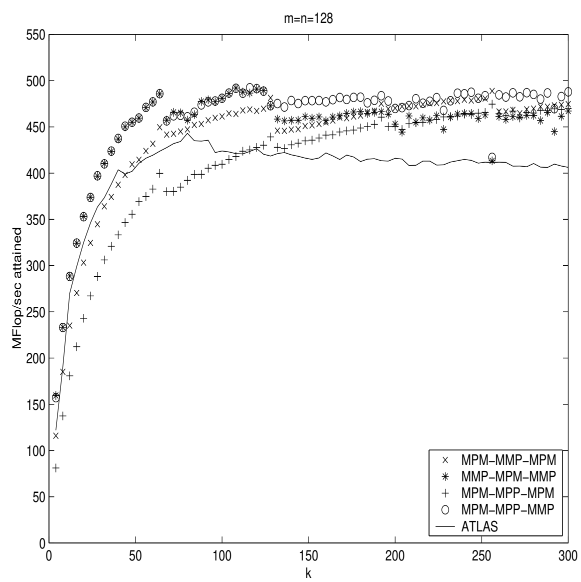
(a)



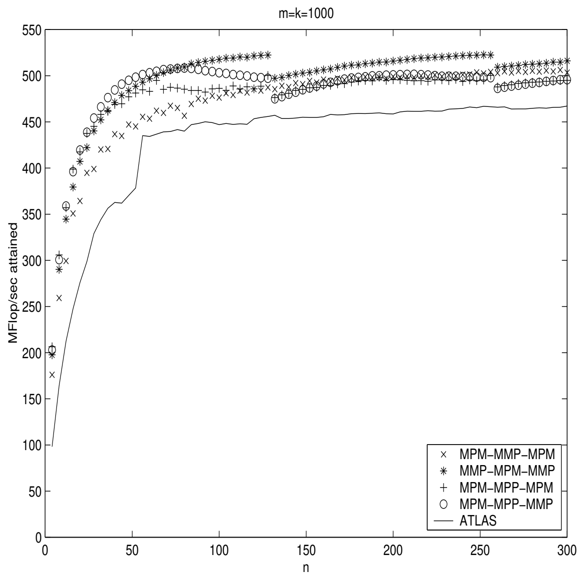
(b)



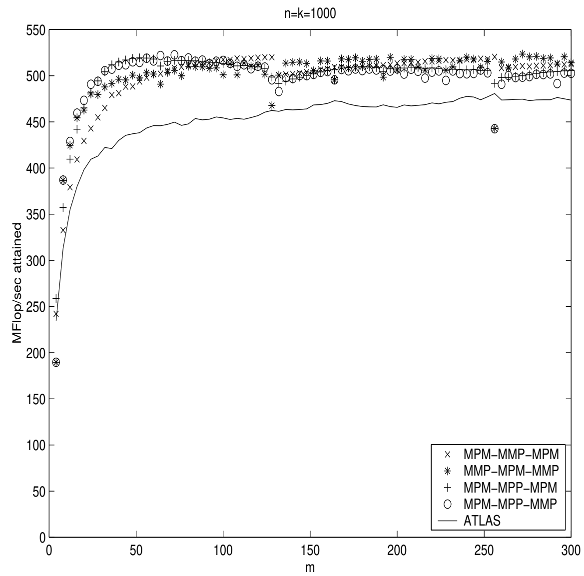
(c)



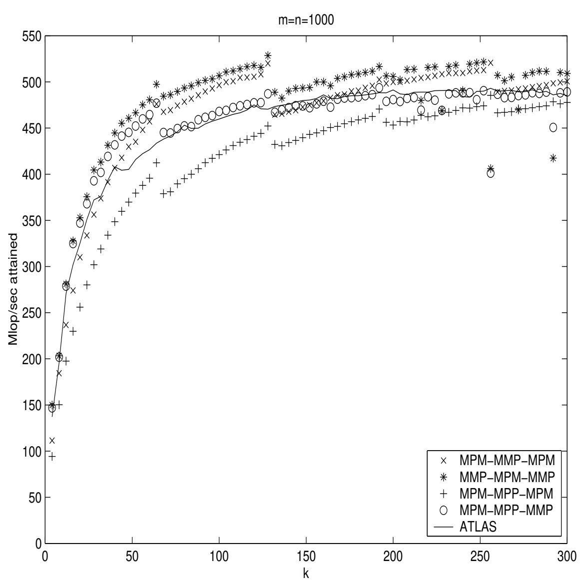
(d)


m=k=1000
MFlop/sec attained
n
MPM–MMP–MPM
MMP–MPM–MMP
MPM–MPP–MPM
MPM–MPP–MMP
ATLAS

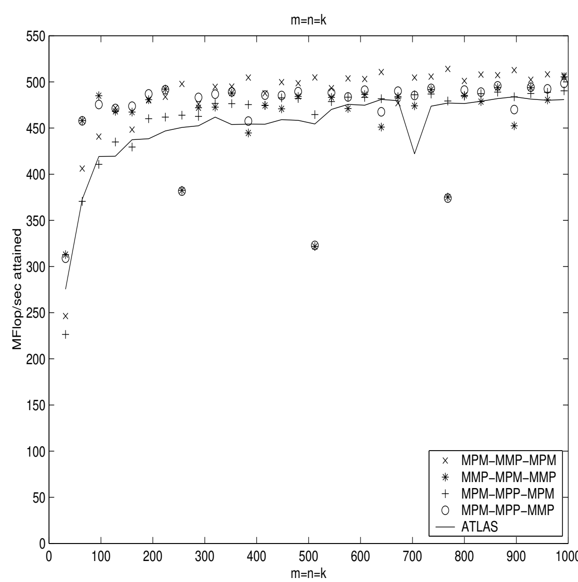
(a)


n=k=1000
MFlop/sec attained
m
MPM–MMP–MPM
MMP–MPM–MMP
MPM–MPP–MPM
MPM–MPP–MMP
ATLAS

(b)


m=n=1000
Mflop/sec attained
k
MPM–MMP–MPM
MMP–MPM–MMP
MPM–MPP–MPM
MPM–MPP–MMP
ATLAS

(c)


m=n=k
MFlop/sec attained
m=n=k
MPM–MMP–MPM
MMP–MPM–MMP
MPM–MPP–MPM
MPM–MPP–MMP
ATLAS

(d)