On Composing Matrix Multiplication from Kernels

Bryan Marker

May 8, 2007

A Turing Scholars Honors Thesis
Supervised by Robert van de Geijn

Abstract

Matrix multiplication is often treated as a basic unit of computation in terms of which other operations are implemented, yielding high performance. In this paper initial evidence is provided that there is a benefit gained when lower level kernels, from which matrix multiplication is composed, are exposed. In particular it is shown that matrix multiplication itself can be coded at a high level of abstraction as long as interfaces to such low-level kernels are exposed. Furthermore, it is shown that higher performance implementations of parallel matrix-multiplication can be achieved by exploiting access to these kernels. Experiments on Itanium2 and Pentium 4 servers support these insights.

1 Introduction

The focus of this paper is the General Matrix-Matrix Multiplication (GEMM) operation, which is part of the Basic Linear Algebra Subprograms (BLAS). It is a widely-used operation in linear algebra algorithms. Unfortunately, the commonly accepted means for interfacing GEMM is sub-optimal. The accepted view is that GEMM (and all BLAS routines) should be treated as atomic.

This paper will show why this is not preferable when GEMM is performed multiple times on the same data. An improved way to implement GEMM by composing it from low-level kernels will be presented. The three low-level kernels needed to compose GEMM as it is in the GotoBLAS include one for performing the actual multiplication and two for packing portions of the operand matrices (for the purpose of making the data physically contiguous in memory, which improves performance). When the low-level kernels are available, a high-level implementation can perform nearly as well as hand-tuned GEMM for both sequential and multi-threaded implementations. Figure 1, for example, demonstrates how using the low-level kernels with a high-level implementation of parallelized GEMM results in performance as good as or better than that of the GotoBLAS.

1.1 Multithreaded Systems

The portion of this thesis covering parallel issues focuses on architectures that support multiple, simultaneous threads of execution such as Symmetric Multiprocessing (SMP) systems. It will be assumed that there is a shared pool of memory and either a single processor with multiple processing cores or multiple processors accessing that pool (each of these processors could have multiple cores as well). With vendors shifting their development to multi-core and, eventually, many-core architectures, the need for efficient scalability for parallelization will become greater. The term "processing core" will be used to refer to a single processing

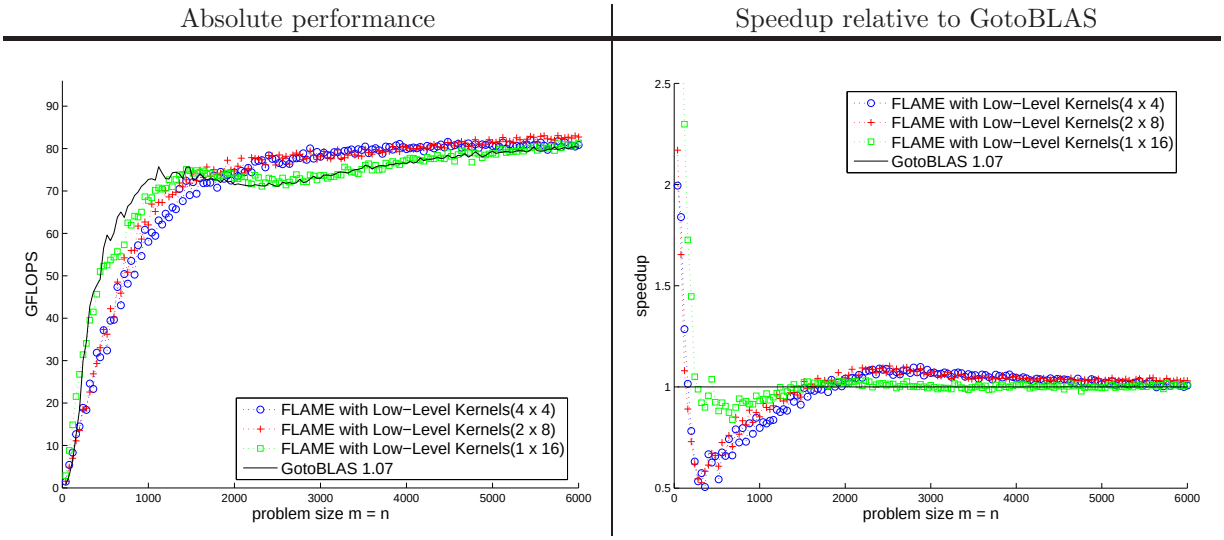


Figure 1: Results of the top-performing partitioning schemes for parallelized GEMM composed of low-level kernels compared to the performance of the multi-threaded GotoBLAS on a 16 Itanium2 based server.

element that runs a computation (e.g. a dual-core processor has a two processing cores while a system with two dual-core processors has four processing cores).

1.2 Notation

Matrices will be represented by upper case letters such as A and B. The partitioned blocks of matrices will be represented by the matrix letter and a subscript numeral for the block number such as A_0 and B_2.

The Formal Linear Algebra Methods Environment (FLAME) was used throughout the research for this thesis [1]. It is a way to systematically derive algorithms that are proven correct concurrently with derivation. It is also a way to make the transition from algorithm to code (in this case C) seamless to reduce common errors. Therefore, the FLAME notation will be used to express algorithms in this paper.

In Figure 2 a trivial example of a FLAME algorithm is shown. This example is given only to demonstrate the FLAME method for expressing algorithms and code; it has no relevance to GEMM or low-level kernels. It is a function that adds the input matrices A and B and stores the result in B. It does so by partitioning A and B into panels of rows, each if which has a maximum of b_m rows. On each iteration the partitions A_1 and B_1 are added and the result is stored in B_1. Notice that the function FLA_Add actually performs the same task as the example itself, which is why it is rather trivial. The figure illustrates the way that algorithms can be converted seemlessly to C code. It also demonstrates the way that all operations will be expressed.

1.3 Workqueuing

The FLAME API for the C language, called FLAME/C, has been extended to include a workqueuing mechanism for SMP systems [2, 6, 8]. FLAME workqueuing allows for an algorithm to be coded in a way that is only a slight modification of the sequential implementation. Furthermore, it maintains the correctness of FLAME-derived algorithms. Figure 3 shows the parallelized version of the algorithm and sequential code in Figure 2.

Algorithm: $B := \text{SAMPLE\_ADD}(A, B)$

Partition $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$

where A_T has 0 rows, B_T has 0 rows

while $m(A_T) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right)$$

where A_1 **has** b **rows,** B_1 **has** b **rows**

$B_1 := A_1 + B_1$

Continue with

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right)$$

endwhile

```
int sample_add( FLA_Obj A, FLA_Obj B, int b_m )
{
  FLA_Obj AT,              A0,
          AB,              A1,
                           A2;
  FLA_Obj BT,              B0,
          BB,              B1,
                           B2;
  int b;

  FLA_Part_2x1( A,    &AT,
                      &AB,            0, FLA_TOP );
  FLA_Part_2x1( B,    &BT,
                      &BB,            0, FLA_TOP );
  while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
    b = min( FLA_Obj_length( AB ), b_m );
    FLA_Repart_2x1_to_3x1( AT,               &A0,
                         /* ** */           /* ** */
                                             &A1,
                           AB,               &A2,        b, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( BT,               &B0,
                         /* ** */           /* ** */
                                             &B1,
                           BB,               &B2,        b, FLA_BOTTOM );
    /*--------------------------------------------------------------*/
    FLA_Add(A1, B1);
    /*--------------------------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,              A0,
                                                A1,
                            /* ** */          /* ** */
                              &AB,              A2,     FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &BT,              B0,
                                                B1,
                            /* ** */          /* ** */
                              &BB,              B2,     FLA_TOP );
  }
  return FLA_SUCCESS;
}
```

Figure 2: Left: FLAME algorithm to add the matrices A and B and store the result in B. Right: Representation in C using the FLAME/C API. The function FLA_Add adds the two operands and stores the result in the second operand.

There are work queuing initialization, executation, and finalization commands that have been added to the code in Figure 3. Additionally, the FLAME function FLA_Add has been changed to the workqueuing equivalent ENQUEUE_FLA_Add. This function has the same prototype as the sequential version, but instead it queues work tasks instead of actually executing them.

Notice that the partitioning is performed in the same way as in the sequential version. Therefore, the exact same work that is executed in the sequential version is queued in the parallel version. When FLA_Queue_exec is called, the tasks are dequeued; with each dequeued task, a function call is made on a thread bound to a single core. Thus, parallel execution is performed.

1.4 Paper Organization

In Section 2 a quick overview is given of a state-of-the-art implementation of the BLAS GEMM operation found in the GotoBLAS. Next, Section 3 discusses how a naive implementation of GEMM, built on the BLAS using FLAME/C, is sub-optimal because of redundancies that occur; an improved version is given that takes advantage of access to low-level kerels. Section 4 discusses parallelized GEMM and how redundancies can be remedied. Finally, Section 5 gives concluding comments.

```
int sample_add( FLA_Obj A, FLA_Obj B, int b_m )
{
  FLA_Obj AT,             A0,
          AB,             A1,
                          A2;
  FLA_Obj BT,             B0,
          BB,             B1,
                          B2;
  int b;

  FLA_Queue_init();
  FLA_Part_2x1( A,    &AT,
                      &AB,            0, FLA_TOP );
  FLA_Part_2x1( B,    &BT,
                      &BB,            0, FLA_TOP );
  while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
    b = min( FLA_Obj_length( AB ), b_m );
    FLA_Repart_2x1_to_3x1( AT,                &A0,
                        /* ** */            /* ** */
                                            &A1,
                           AB,              &A2,        b, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( BT,              &B0,
                        /* ** */            /* ** */
                                            &B1,
                           BB,              &B2,        b, FLA_BOTTOM );
    /*--------------------------------------------------------------*/
    ENQUEUE_FLA_Add(A1, B1);
    /*--------------------------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,              A0,
                                                A1,
                            /* ** */          /* ** */
                              &AB,              A2,     FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &BT,              B0,
                                                B1,
                            /* ** */          /* ** */
                              &BB,              B2,     FLA_TOP );
  }
  FLA_Queue_exec();
  FLA_Queue_finalize();
  return FLA_SUCCESS;
}
```

Figure 3: The parallelized version of the algorithm and code of Figure 2.

1.5 Author's Contribution

This paper builds on the research and work performed by members of the FLAME group including that featured in [7]. The contributions of the author include the high-level implementations of both the sequential and parallel GEMM operations using low-level kernels.

2 GotoBLAS GEMM

The following is a minimal discussion of how the GEMM operation is tuned for near-optimal performance in the GotoBLAS [4]. This knowledge will be used in future chapters to expose and to remedy sub-optimal uses of the BLAS.

2.1 Memory Hierarchy

On current architectures of interest, there are multiple levels of memory, as illustrated in Figure 4 (left). Access to main memory is slow, which typically holds the data. With each successive, higher level, storage capacity decreases while read and write performance increases. At the top level are the processor's registers.

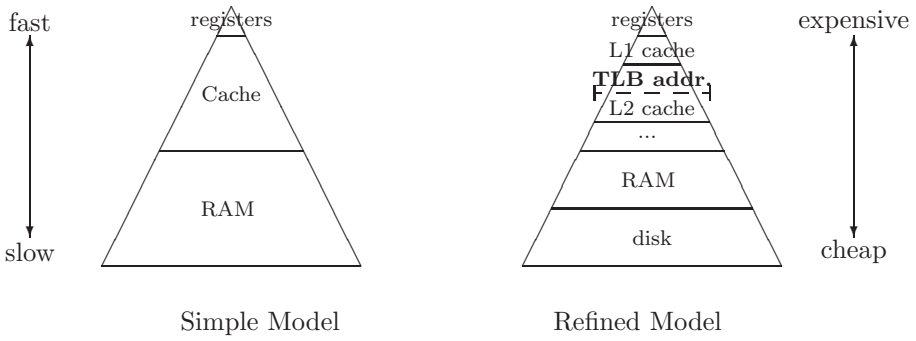


Figure 4: The hierarchical memories viewed as a pyramid. (This figure was borrowed from [4])

Here, very little storage capacity is available, but the read and write speed is effectively instantaneous.

All data is initially store in the main memory and must eventually migrate to the registers for computation to take place. This movement is costly, taking many clock cycles, so it must be orchestrated very carefully for near-optimal performance to be achieved.

To assist in reusing data, the cache provides a level of storage between main memory and the registers. It is significantly faster than main memory but does not have near the storage capacity. Its purpose is to store recently accessed data. The idea behind this is that if data has been recently read, it will be read soon in the future. Therefore, by storing data that has been recently read from main memory (a costly operation), when it is read soon in the future the read is made from the faster cache.

Since the cache is much smaller than main memory and automagically stores data, the use of it must be carefully orchestrated. When done properly, though, using portions of data stored in the cache rather than the main memory results in significant performance benefits. There are often multiple levels of cache; the L1 cache is smaller but faster than L2. On some architectures (such as the Itanium2) there is an L3 cache which is larger and slower than the L2 cache.

In the pyramid on the right in Figure 4, the Translation-Lookaside Buffer (TLB) is shown between the L1 and L2 caches. It is a special type of cache that logs the translation of virtual addresses to physical addresses for recently accessed memory. When a TLB miss occurs, a costly operation is required to read from memory the mapping and then to read from memory the requested data. Therefore, minimizing the number of TLB misses in sequential reads can significantly enhance performance. This is done when sequential reads are made from physically contiguous memory.

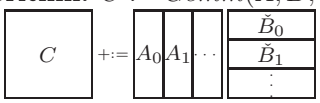
2.2 Variants of GEMM

The prototypical GEMM operation is defined for the purposes of this paper as $C := AB + C$. A, B, and C are matrices of size $m \times k$, $k \times n$, and $m \times n$, respectively, where m, k, and n are dimensions that are trivially large. A and B are multiplied together and the result is added to and then stored in C.

To perform the GEMM operation, A, B, and C can be partitioned in three natural ways. The first partitions along the m dimension:

$$A = \left(\begin{array}{c} \check{A}_0 \\ \hline \check{A}_1 \\ \hline \vdots \end{array}\right) \text{ and } C = \left(\begin{array}{c} \check{C}_0 \\ \hline \check{C}_1 \\ \hline \vdots \end{array}\right)$$

Algorithm: $C := Gemm(A, B, C)$

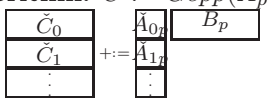


for $p = 0, \ldots, k/b_k$

$C := A_p \check{B}_p + C$ ($C := Gepp(A_p, \check{B}_p, C)$)

endfor

Algorithm: $C := Gepp(A_p, B_p, C)$



Pack B_p into work buffer $\tilde{B}$

for $i = 0, \ldots, m/b_m$

Pack and transpose $\check{A}_{ip}$ into $\tilde{A}$

$\check{C}_j := \tilde{A}\tilde{B} + \check{C}_i$ (GEBP)

endfor

Figure 5: Left: Algorithm for GEMM via GEPP. Right: Alogorithm for optimized GEPP.

such that the operation looks like

$$\left(\begin{array}{c}\check{C}_0\\ \hline \check{C}_1\\ \hline \vdots\end{array}\right) := \left(\begin{array}{c}\check{A}_0\\ \hline \check{A}_1\\ \hline \vdots\end{array}\right) B + \left(\begin{array}{c}\check{C}_0\\ \hline \check{C}_1\\ \hline \vdots\end{array}\right) = \left(\begin{array}{c}\check{A}_0 B + C_0\\ \hline \check{A}_1 B + C_1\\ \hline \vdots\end{array}\right).$$

Then, $C := AB + C$ becomes $C_i := \hat{A}_i B$.

The second partitions along the n dimension:

$$B = \left(\begin{array}{c|c|c} B_0 & B_1 & \cdots \end{array}\right) \text{ and } C = \left(\begin{array}{c|c|c} C_0 & C_1 & \cdots \end{array}\right)$$

such that the operations looks like

$$\left(\begin{array}{c|c|c} C_0 & C_1 & \cdots \end{array}\right) := A \left(\begin{array}{c|c|c} B_0 & B_1 & \cdots \end{array}\right) + \left(\begin{array}{c|c|c} C_0 & C_1 & \cdots \end{array}\right) = \left(\begin{array}{c|c|c} AB_0 + C_0 & AB_1 + C_1 & \cdots \end{array}\right)$$

or $C_p := AB_p + C_p$.

Lastly, partitioning along the k dimension, which consists of General Panel-Panel Multiplication operations (GEPP) or rank-k updates, is implemented with

$$A = \left(\begin{array}{c|c|c} A_0 & A_1 & \cdots \end{array}\right) \text{ and } B = \left(\begin{array}{c}\check{B}_0\\ \hline \check{B}_1\\ \hline \vdots\end{array}\right)$$

such that operations look like

$$C := \left(\begin{array}{c|c|c} A_0 & A_1 & \cdots \end{array}\right) \left(\begin{array}{c}\check{B}_0\\ \hline \check{B}_1\\ \hline \vdots\end{array}\right) + C = C + A_0 B_0 + A_1 B_1 + ...$$

so that $C := C + A_0 \check{B}_0 + A_1 \check{B}_1 + ...$

It is the third variant that we will study, since it is the variant used by the GotoBLAS. The algorithm which implements GEMM in terms of GEPP is illustrated in Figure 5 (Left).

2.3 GEPP Algorithm

The GotoBLAS optimizes GEMM by optimizing the individual GEPP operations of which it is composed. It is the implementation of GEPP that must be carefully tuned to attain efficiency.

Let the partitions on which GEPP operates be called A_p and $\check{B}_p$ for $p = 0, ..., k/b_k$, which are assumed to be of sizes $m \times b_k$ and $b_k \times n$, respectively. b_k is an blocking size that is chosen to be optimal for the hardware [4]. For simplicity we assume that the k dimension of A and B is evenly divisible by b_k.

The panel operand A_p is further-partitioned to cast GEPP in terms of General Block-Panel Multiply (GEBP) operations as seen on the left in Figure 5 (the lines related to packing are further explained in the next subsection). The operands of GEBP are assumed to be of size $b_m \times b_k$ and $b_k \times n$. The partitions of A_p are denoted by $\check{A}_{ip}$ for $i = 0, ..., m/b_m$. Again, for simplicity we assume that m is evenly divisible by b_m.

2.4 Optimized GEPP Implementation

As stated before data movement between memory layers must be very carefully orchestrated. Here, each GEPP is viewed as a sequence of many GEBP operations. All of these use the same matrix of B, so unnecessary (and costly) TLB misses associated with B should be avoided

To do this the partitions of B are packed into a physically contiguous work buffer so that TLB misses are minimized, as shown in Figure 5 (Right). Furthermore, the partitions of A are packed into a work buffer such that the result of packing leaves A in the L2 cache and requires few TLB entries. With the cost of packing B amortized across all of the GEBP operations performed by GEPP and the benefit of significantly reduced impact due to TLB misses, the performance hit due to slow memory is greatly reduced by using this algorithm. Thus, the GEBP operation can be performed at near peak rates [4].

As seen in Figure 5 (Right), there are three main building blocks for of the GEPP algorithm. We call these "low-level kernels":

- **Pack** A **:** Copies a block of A into a physically contiguous work buffer, $\tilde{A}$. The result is that $\tilde{A}$ is stored in the L2 cache and requires a minimal number of TLB entries. (As GEMM has been defined above, this process also transposes the data into row-major order for reasons that go beyond the scope of this paper.)
- **Pack** B **:** Copies a panel of B into a physically contiguous work buffer, $\tilde{B}$, such that all of $\tilde{B}$ can be addressed by a minimal number of TLB entries.
- **GEBP :** performs $C := \tilde{A}\tilde{B} + C$. $\tilde{A}$ is $b_m \times b_k$ and $\tilde{B}$ is $b_k \times n$ for the best performance.

2.5 Related Work

The ATLAS package [9] provides an alternative implementation of GEMM that will be discussed here to give a contrast to the previously described algorithm. A better overview can be found in [10]. While the GotoBLAS is hand-tuned for each architecture to which it is ported, the ATLAS approach is based on the automatic tuning approach pioneered by [3]. ATLAS uses parameters such as the blocking size, N_B, and the order of loops to search many possible code combinations on any particular architecture for superior selections. By running benchmarks for code generated with combinations of parameters, the idea is that the best values for the parameters can be found. As one would expect, the search space is quite large, so this process takes some time and is not guaranteed to find the optimal values for parameters. Recent work uses analytical models combined with a more limited search to find these values [10].

The ATLAS method blocks in all three dimensions, m, k, and n, by the same blocking size N_B. The $N_B \times N_B$-sized blocks of A, B, and C are multiplied in what are called mini-GEMM operations. In order to minimize costly memory operations, the parameter N_B is found such that the required data for the mini-GEMM operations can fit onto the L1 cache. Thus, each mini-GEMM is meant to run from the relatively fast L1 cache.

Much as with the GotoBLAS, ATLAS generally copies portions of the matrix (often all of A, a panel of B, and a tile of C) into a physically contiguous work buffer. Thus, while the algorithms is different than that of the GotoBLAS, there are still low-level kernels which perform the necessary steps to pack data and perform the sub-operations.

In [5] it is shown theoretically that the blocking strategy used by the GotoBLAS GEMM is superior.

3 Sequential FLAME Implementation

In this section the sequential GEMM operation will be implemented by composing the low-level GEMM kernels of the algorithm covered in Section 2. This will be shown to result in an implementation that is essentially as efficient as that of the GotoBLAS. Thus, it is shown that a high-level implementation using FLAME can be as high-performing when it is composed of low-level kernels.

3.1 Standard Implementation

In Figure 6 the GEMM operation is implemented by casting it into a number of GEPP operations. GEPP is implemented in Figure 7 in which GEBP is performed by calling the GotoBLAS implementation of GEMM. Here, the low-level kernels are not used; therefore, the advantage of having them available will be apparent when compared.

The blocking sizes of this algorithm b_m and b_k are chosen to be the same as those used by the GotoBLAS (since the GotoBLAS is to be mimicked). Notice that each call to the GotoBLAS's GEMM in GEPP then operates on the same portion of B as would the GEBP low-level kernel in Figure 5 (Right); specifically, the entire call results in a single call to the GEBP low-level kernel.

Further notice that in this naive implementation, each of the m/b_m calls in Figure 7 requires B to be re-packed because direct calls to the GEBP low-level kernel are not made (B is re-packed in Figure 5 (Right)). This means that as the m dimension increases, the redundant overhead of re-packing increases linearly.

3.2 Improved Implementation

Now, consider Figure 8. When access to the low-level kernels previous described are directly available, the portions of B need not be re-packed for each of the GEBP operations. B is packed for each call to GEPP, just as in Figure 5 (Right). A is packed soon before each call to GEBP, just as in GotoBLAS.

3.3 Experimental Results

There are two main ideas that are demonstrated by the results of this section. Firstly, a high-level implementation of the GEMM operation can be as efficient as the GotoBLAS implementation. Secondly, this implementation requires access to low-level kernels to reach such efficiency.

There are two sets of results reported in this section. Each was taken from a machine with different processors to provide some variety in results.

The first set of performance plots was taken from a machine consisting of four Intel Itanium2 processors with a shared memory pool of 8 GB. Itanium2 processors execute a maximum of four FLOPs per clock cycle and each of these processors runs at 1.5GHz. Therefore, the each processor runs at a peak of 6 GFLOPS (10^9 FLOPs/sec.), which results in a machine peak of 24 GFLOPS. Only a single processor was used and the remaining three were kept idle, so the peak performance is 6 GFLOPS. This is represented at the top of the vertical axis on the graphs, which show performance. The matrices tested where squares with the dimensions ranged ranged between 200 and 6000 in increments of 200. The matrix A was embedded such that its leading dimension was equal to 6000. The leading dimension of B was equal to the matrix size k.

Algorithm: $C := \text{GEMM}(A, B, C)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_L & A_R \end{array} \right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right)$

where A_L has 0 columns, B_T has 0 rows

while $n(A_L) < n(A)$ **do**

Determine block size b

Repartition

$\left(\begin{array}{c|c} A_L & A_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array} \right)$,

$\left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array} \right)$

where A_1 **has** b **columns,** B_1 **has** b **rows**

$C := A_1 B_1 + C$ $(Gepp)$

Continue with

$\left(\begin{array}{c|c} A_L & A_R \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array} \right)$,

$\left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array} \right)$

endwhile

```
int FLA_Gemm( FLA_Obj A, FLA_Obj B, FLA_Obj C, int b_k, int b_m )
{
  FLA_Obj AL,    AR,       A0,  A1,  A2;
  FLA_Obj BT,              B0,
          BB,              B1,
                           B2;
  int b;

  FLA_Part_1x2( A,    &AL,  &AR,      0, FLA_LEFT );
  FLA_Part_2x1( B,    &BT,
                      &BB,            0, FLA_TOP );
  while ( FLA_Obj_width( AL ) < FLA_Obj_width( A ) ){
    b = min( FLA_Obj_width( AR ), b_k );
    FLA_Repart_1x2_to_1x3( AL,  /**/ AR,        &A0, /**/ &A1, &A2,
                           b, FLA_RIGHT );
    FLA_Repart_2x1_to_3x1( BT,                &B0,
                         /* ** */           /* ** */
                                             &B1,
                           BB,               &B2,        b, FLA_BOTTOM );
    /*--------------------------------------------------------------*/
    FLA_Gepp( A1, B1, C, b_m );
    /*--------------------------------------------------------------*/
    FLA_Cont_with_1x3_to_1x2( &AL,  /**/ &AR,        A0, A1, /**/ A2,
                              FLA_LEFT );
    FLA_Cont_with_3x1_to_2x1( &BT,               B0,
                                                 B1,
                            /* ** */          /* ** */
                              &BB,               B2,     FLA_TOP );
  }
  return FLA_SUCCESS;
}
```

Figure 6: Top: Sequential GEMM algorithm expressed in FLAME notation. Bottom: Representation in C using the FLAME/C API.

Algorithm: $C := \text{GEPP}(A, B, C)$

Partition $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$, $C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)$

where A_T has 0 rows, C_T has 0 rows

while $m(A_T) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right), \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$$

where A_1 **has** b **rows,** C_1 **has** b **rows**

$C := A_1 B + C_1 (Gebp)$

Continue with

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right), \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$$

endwhile

```
int FLA_Gepp( FLA_Obj A, FLA_Obj B, FLA_Obj C, int b_m )
{
  FLA_Obj AT,              A0,
          AB,              A1,
                           A2;
  FLA_Obj CT,              C0,
          CB,              C1,
                           C2;
  int b;

  FLA_Part_2x1( A,    &AT,
                      &AB,            0, FLA_TOP );
  FLA_Part_2x1( C,    &CT,
                      &CB,            0, FLA_TOP );
  while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
    b = min( FLA_Obj_length( AB ), b_m );
    FLA_Repart_2x1_to_3x1( AT,                &A0,
                         /* ** */           /* ** */
                                             &A1,
                           AB,               &A2,        b, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( CT,               &C0,
                         /* ** */           /* ** */
                                             &C1,
                           CB,               &C2,        b, FLA_BOTTOM );
    /*--------------------------------------------------------------*/
    FLA_dgemm( A1, B, C1 );
    /*--------------------------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,               A0,
                                                 A1,
                            /* ** */          /* ** */
                              &AB,               A2,     FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &CT,               C0,
                                                 C1,
                            /* ** */          /* ** */
                              &CB,               C2,     FLA_TOP );
  }
  return FLA_SUCCESS;
}
```

Figure 7: Top: Sequential GEPP algorithm expressed in FLAME notation. Bottom: Representation in C using the FLAME/C API. For now GEBP is performed by calling a wrapper to the GotoBLAS GEMM implementation for double-precision floating points.

```
int FLA_Gepp( FLA_Obj A, FLA_Obj B, FLA_Obj C, int b_m )
{
  FLA_Obj AT,              A0,
          AB,              A1,
                           A2;
  FLA_Obj CT,              C0,
          CB,              C1,
                           C2;
  FLA_Obj packed_A, packed_B;
  int b;

  FLA_Pack_B( B, packed_B );
  FLA_Part_2x1( A,    &AT,
                      &AB,            0, FLA_TOP );
  FLA_Part_2x1( C,    &CT,
                      &CB,            0, FLA_TOP );
  while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
    b = min( FLA_Obj_length( AB ), b_m );
    FLA_Repart_2x1_to_3x1( AT,                &A0,
                         /* ** */           /* ** */
                                              &A1,
                           AB,                &A2,        b, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( CT,                &C0,
                         /* ** */           /* ** */
                                              &C1,
                           CB,                &C2,        b, FLA_BOTTOM );
    /*--------------------------------------------------------------*/
    FLA_Pack_A( A, packed_A );
    FLA_Gebp( packed_A, packed_B, C1 );
    /*--------------------------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,               A0,
                                                 A1,
                            /* ** */           /* ** */
                              &AB,               A2,     FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &CT,               C0,
                                                 C1,
                            /* ** */           /* ** */
                              &CB,               C2,     FLA_TOP );
  }
  return FLA_SUCCESS;
}
```

Figure 8: Improved version of Fig. 7 using low-level kernels to prevent unnecessary re-packing operations. FLA_Pack_A, FLA_Pack_B, and FLA_Gebp are wrappers to the low-level kernels.

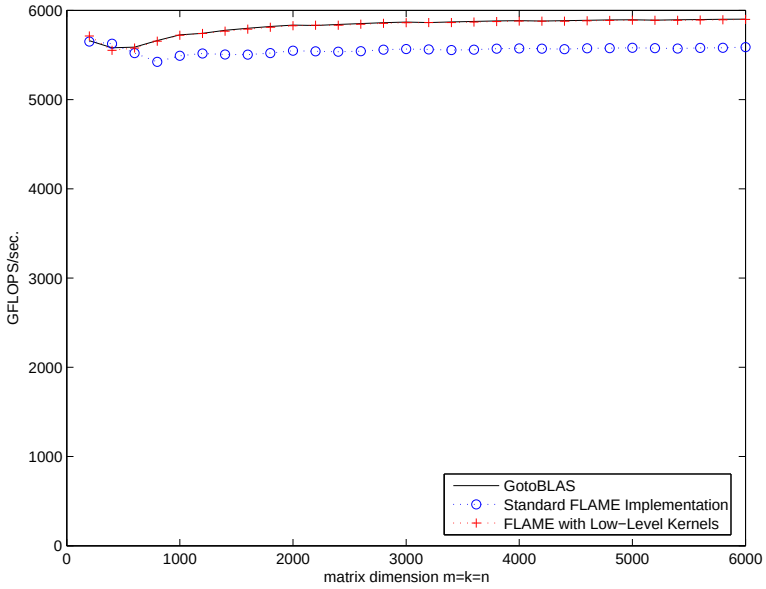


Figure 9: Results taken on Itanium2 processor of sequential GEMM.

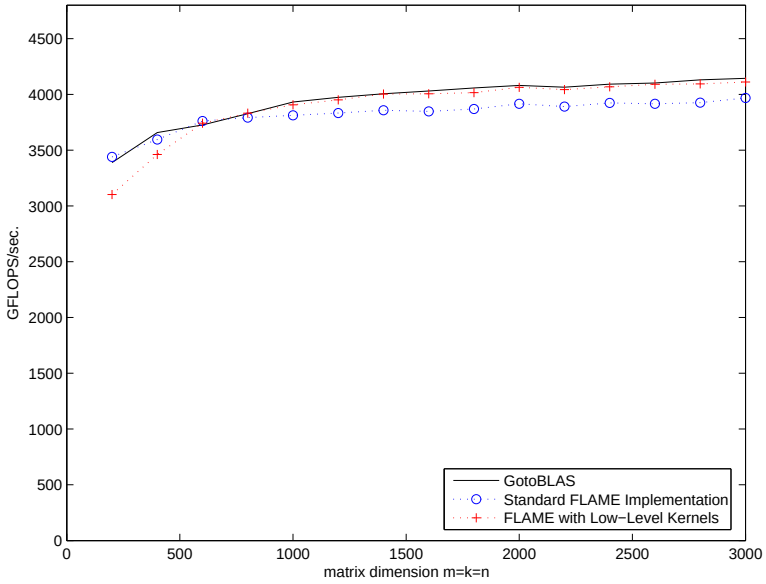


Figure 10: Results taken on Pentium 4 Northwood processor of sequential GEMM.

The second set was taken from a single Intel Pentium 4 Northwood processor (2.4 GHz) with 2 FLOPs being performed during each cycle. It was thus capable of up to 4.8 GFLOPS. The machine had 512 MB of memory. The matrices tested were again squares. The dimensions ranged between 200 and 3000 in increments of 200. The leading dimension of A was 3000, and that of B was equal to the matrix size k.

The graphs demonstrate performance as dependent on varied matrix sizes, as specified on the horizontal axis.

As seen in Figure 9, results on the Itanium2 processor confirm that there is a benefit to exposing low-level kernels so as to circumvent re-packing. In fact the implementation composed of low-level kernels showed an average of 4.8% improved performance over the standard implementation on the Itanium2.

While the results on Northwood were not as pronounced in Figure 10, they were still telling. An average improvement of 1.9% was seen.

Notice that in both implementations there are matrix sizes for which the improved implementation performs poorer than the standard implementation. In fact the GotoBLAS implementation follows the standard implementation curve at these points. The reason for this is outside the scope of this paper and only affects small matrix sizes for certain architectures such as those with the SSE2 instruction set. The previously explained issues are still valid for the majority of problem sizes.

The fact that the improved implementation roughly mimicked the results of the GotoBLAS for most problem sizes that were tested is very important. This shows that a high-level implementation of GEMM is effectively as well-performing as the hand-tuned GotoBLAS implementation as long as the low-level kernels are used.

4 Parallel FLAME Implementation

Architectures that can execute simultaneous threads, like SMPs, are quickly becoming common and will be even more prevalent in the future. Therefore, it is natural for high performance algorithms to target such systems. Unfortunately, though, the inefficiencies seen with the naive approach on sequential systems are amplified when work is partitioned for parallelism. It will be shown that a high level implementation can be given for parallelized GEMM that performs very efficiently. Furthermore, with the low-level kernels available, inefficient packing overhead can be removed. This section first covers how work is to be partitioned. Then, the naive and improved versions of parallel GEMM will be presented and results will demonstrate the efficiency of the high-level implementation.

4.1 Work Partitioning

As with distributed memory systems, a partitioning scheme for the data must be created to induce work tasks that can be distributed among the processing cores. We will refer to the number of simultaneous threads as t. The goal is to create t tasks, each of which is carried out by a separate thread, which is bound to a processing core. An ideal situation is when work can be evenly divided among the t threads so that perfect load balancing is achieved.

The partitioning schemes for GEMM presented in Section 2.2 naturally expose one-dimensional work partitioning along the m, k, or n dimension such that the result is t partitions of work. This can be achieved by taking the blocksize to be m/t, k/t, or n/t, depending on the dimension over which partitioning occurs.

Recall from Section 2.2 that partitioning along the k dimension results in an operation of the form

$$C := \left(\begin{array}{c|c|c} A_0 & A_1 & \cdots \end{array} \right) \left(\begin{array}{c} \check{B}_0 \\ \hline \check{B}_1 \\ \hline \vdots \end{array} \right) + C = C + A_0 B_0 + A_1 B_1 + ...$$

This means that if each thread calculated one of these terms, the contributions would need to be summed together. Therefore, partitioning in the k dimension for parallelism is not discussed here. In fact because any GEMM operation can be implemented efficiently as a series of rank-k updates, we will assume that the

k dimension is already small, as would be the case in each GEPP operation of a rank-k update, and we will pursue performance parallelism only with each GEPP operation.

As with distributed memory systems, two-dimensional work partitioning can generally achieve better performance than one-dimensional partitioning. Therefore, the two flavors of one-dimensional work partitioning (in the m and n dimensions) can be combined to form a two-dimensional partitioning scheme.

$$\left(\begin{array}{c|c|c} C_{00} & C_{01} & \cdots \\ \hline C_{10} & C_{11} & \cdots \\ \hline \vdots & \vdots & \ddots \end{array}\right) := \left(\begin{array}{c} \check{A}_0 \\ \hline \check{A}_1 \\ \hline \vdots \end{array}\right) \left(\begin{array}{c|c|c} B_0 & B_1 & \cdots \end{array}\right) + \left(\begin{array}{c|c|c} C_{00} & C_{01} & \cdots \\ \hline C_{10} & C_{11} & \cdots \\ \hline \vdots & \vdots & \ddots \end{array}\right)$$

$$= \left(\begin{array}{c|c|c} A_0B_0 + C_{00} & A_0B_1 + C_{01} & \cdots \\ \hline A_1B_0 + C_{10} & A_1B_1 + C_{11} & \cdots \\ \hline \vdots & \vdots & \ddots \end{array}\right)$$

In this scheme the m and n dimensions are broken into t_r and t_c partitions, respectively, to form a grid such that $t = t_r \times t_c$. This leads to the situation of t even tasks to be distributed. If the t processing cores were viewed as a $t_r \times t_c$ grid, the processor t_{ip}, for $i = 0, \ldots, t_r$ and $p = 0, \ldots, t_c$, would calculate $C_{ip} := A_i B_p + C_{ip}$. If either $t_r = 1$ or $t_c = 1$, the two-dimensional scheme reduces to one-dimensional partitioning.

4.2 Standard Implementation

Figure 11 shows an initial implementation of GEMM with two-dimensional work partitioning that uses the GotoBLAS implementation of GEMM for the individual tasks that are induced. In this figure the matrix is partioned in the m dimension by a blocksize determined by FLA_Task_compute_blocksize, which effectively returns m/t_r. FLA_Gemm_n, shown in Figure 12, blocks in the n dimension by n/t_c. The $t = t_r \times t_c$ tasks are enqueued with each call of ENQUEUE_FLA_Gemm. When FLA_Queue_exec is called, these tasks are dequeued and executed in parallel. Each of the t tasks is bound to one of the t simultaneously executing threads.

Just as with the intial approach to the sequential implementation, there are redundant packing operations that occur. There are t_r threads that use each of the t_c partitions of B. Thus, on each of the threads, the partitions of B are re-packed as shown in Figure 5 (Right). Therefore, as t_r increases, the performance hit due to redundant packing increases linearly.

4.3 Improved Implementation

With access to the low-level kernels, the performance hit incurred due to redundant packing can be avoided while still using FLAME code and the high-level abstraction of two-dimensional data partitioning. In Figures 13 and 14 the improved code is shown.

In Figure 13 FLA_Parallel_pack_B refers to the parallelized packing of B, similar to what occurred in Figure 8. As before the purpose of packing partitions of B into contiguous work buffers is to minimize the number of TLB misses that occur when reading data. Since the portions of B are used by multiple threads, they will redundantly pack B to contiguous work buffers. This redundancy serves no purpose, so the packing of B can performed at the beginning, in parallel, and all of the threads can use the pre-packed partitions of B. They still benefit from reduced TLB misses but do not have to re-pack B as in the standard implementation.

After B is completely packed to a work buffer, A and B are partitioned into t_r and t_c almost equal blocks, respectively. This results in a two-dimensional grid of $t = t_r \times t_c$ GEPP operations that are enqueued.

```
FLA_Error FLA_Gemm_mn( FLA_Obj A, FLA_Obj B, FLA_Obj C )
{
  FLA_Obj AT,             A0,
          AB,             A1,
                          A2;
  FLA_Obj CT,             C0,
          CB,             C1,
                          C2;
  int b;

  FLA_Queue_init();
  FLA_Part_2x1( A,    &AT,
                      &AB,            0, FLA_TOP );
  FLA_Part_2x1( C,    &CT,
                      &CB,            0, FLA_TOP );
  while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) )
  {
    b = FLA_Task_compute_blocksize( 0, A, AT, FLA_TOP );
    FLA_Repart_2x1_to_3x1( AT,               &A0,
                        /* ** */           /* ** */
                                             &A1,
                           AB,               &A2,        b, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( CT,               &C0,
                        /* ** */           /* ** */
                                             &C1,
                           CB,               &C2,        b, FLA_BOTTOM );
    /*-------------------------------------------------------------*/
    FLA_Gemm_n( A1, B, C1 );
    /*-------------------------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,              A0,
                                                A1,
                            /* ** */          /* ** */
                              &AB,              A2,     FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &CT,              C0,
                                                C1,
                            /* ** */          /* ** */
                              &CB,              C2,     FLA_TOP );
  }
  FLA_Queue_exec();
  FLA_Queue_finalize();
  return FLA_SUCCESS;
}
```

Figure 11: The FLAME/C code for parallelized GEMM, which blocks in the m and n dimensions (and thus called FLA_GEMM_mn). It partitions A and C in the m dimension and calls FLA_Gemm_n (Figure 12). FLA_Task_compute_blocksize chooses the optimal blocksize for a $t = t_r x t_c$ grid of roughly equal tasks.

```
FLA_Error FLA_Gemm_n( FLA_Obj A1, FLA_Obj B, FLA_Obj C1 )
{
  FLA_Obj BL,    BR,       B0,  B1,  B2;
  FLA_Obj C1L,   C1R,      C10,  C11,  C12;
  int b;

  FLA_Part_1x2( B,    &BL,  &BR,      0, FLA_LEFT );
  FLA_Part_1x2( C1,    &C1L,  &C1R,      0, FLA_LEFT );
  while ( FLA_Obj_width( BL ) < FLA_Obj_width( B ) ){
    b = FLA_Task_compute_blocksize( 2, B, BL, FLA_LEFT );
    FLA_Repart_1x2_to_1x3( BL,  /**/ BR,        &B0, /**/ &B1, &B2,
                           b, FLA_RIGHT );
    FLA_Repart_1x2_to_1x3( C1L,  /**/ C1R,        &C10, /**/ &C11, &C12,
                           b, FLA_RIGHT );
    /*------------------------------------------------------------*/
    /*    C11 =  A1 * B1 + C11;    */
    ENQUEUE_FLA_Gemm( A1, B1, C11 );
    /*------------------------------------------------------------*/
    FLA_Cont_with_1x3_to_1x2( &BL,  /**/ &BR,        B0, B1, /**/ B2,
                              FLA_LEFT );
    FLA_Cont_with_1x3_to_1x2( &C1L,  /**/ &C1R,        C10, C11, /**/ C12,
                              FLA_LEFT );
  }
  return FLA_SUCCESS;
}
```

Figure 12: The code for FLA_Gemm_n partitions in the n dimension (see Figure 11). Each of the t tasks is enqueued in the loop with the function ENQUEUE_FLA_Gemm The function FLA_Queue_exec distributes all of the tasks to threads and runs them in parallel using the standard call to sequential dgemm in GotoBLAS.

When the FLA_Queue_exec is called, the tasks are dequeued. Each of these tasks execute in parallel. They proceed in a manner very similar to what is seen in Figures 6 and 8 except the partitions of B are already packed. The partitions of A must be re-packed on each thread because this serves the dual-purpose of reducing TLB misses and bringing the block into the L2 cache.

4.4 Packing Issues

In the standard implementation, the GotoBLAS implementation of GEMM is called in each of the threads. When this is the case, the packing of B is handled automagically by the implementation. This can be sub-optimal for some architectures with limited memory bandwidth because having all t threads reading from and writing to memory at the same time could cause a memory bottleneck. In Figure 13, the packing of B is performed explicitly at the beginning of the implementation using low-level kernels. Thus, with some experimentation, the packing of B can be performed such that bottlenecks are reduced.

In the first implementation of parallel GEMM using low-level kernels, the packing of B was handled sequentially by a single thread. As one would expect, performance was poorer than that of the standard implementation as predicted by Amdahl's Law. The second implementation, on the other hand, used t_c threads to pack the t_c partitions of B. This resulted in performance better than that of the first implementation. The third implementation used all t threads to pack the t_c partitions of B. The architecture of the test machine was such that using all t threads was superior to using less than t threads for all problem sizes tested. On some machines, less than t threads might be superior because of the performance hit due to memory bottlenecks. Thus, testing should be performed to find the best balance.

4.5 Experimental Results

The experimental results of this section are meant to demonstrate that a high-level implementation of parallel GEMM is as efficient as the multithreaded GotoBLAS implementation. Furthermore, they will

```
FLA_Error FLA_Gemm_mn( FLA_Obj A, FLA_Obj B, FLA_Obj C )
{
  FLA_Obj AT,              A0,
          AB,              A1,
                           A2;
  FLA_Obj CT,              C0,
          CB,              C1,
                           C2;
  FLA_Obj packed_B;
  int b;

  FLA_Parallel_pack_B ( B, packed_B );
  FLA_Queue_init();
  FLA_Part_2x1( A,    &AT,
                      &AB,            0, FLA_TOP );
  FLA_Part_2x1( C,    &CT,
                      &CB,            0, FLA_TOP );
  while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) )
  {
    b = FLA_Task_compute_blocksize( 0, A, AT, FLA_TOP );
    FLA_Repart_2x1_to_3x1( AT,                &A0,
                         /* ** */           /* ** */
                                              &A1,
                           AB,                &A2,        b, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( CT,                &C0,
                         /* ** */           /* ** */
                                              &C1,
                           CB,                &C2,        b, FLA_BOTTOM );
    /*--------------------------------------------------------------*/
    FLA_Gemm_n( A1, packed_B, C1 );
    /*--------------------------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,                A0,
                                                  A1,
                            /* ** */           /* ** */
                              &AB,                A2,     FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &CT,                C0,
                                                  C1,
                            /* ** */           /* ** */
                              &CB,                C2,     FLA_TOP );
  }
  FLA_Queue_exec();
  FLA_Queue_finalize();
  return FLA_SUCCESS;
}
```

Figure 13: The code for the improved version of parallel GEMM (Figure 11), which blocks in the m and n dimensions (and thus called FLA_GEMM_mn). FLA_Parallel_pack_B (Figure 14) pre-packs B.

```
FLA_Error FLA_Gemm_n( FLA_Obj A1, FLA_Obj packed_B, FLA_Obj C1 )
{
  FLA_Obj BL,    BR,       B0,  B1,  B2;
  FLA_Obj C1L,   C1R,      C10,  C11,  C12;
  int b;

  FLA_Part_1x2( packed_B,    &BL,  &BR,      0, FLA_LEFT );
  FLA_Part_1x2( C1,    &C1L,  &C1R,      0, FLA_LEFT );
  while ( FLA_Obj_width( C1L ) < FLA_Obj_width( C1 ) ){
    b = FLA_Task_compute_blocksize( 2, C1, C1L, FLA_LEFT );
    FLA_Repart_1x2_to_1x3( BL,  /**/ BR,        &B0, /**/ &B1, &B2,
                           1, FLA_RIGHT );
    FLA_Repart_1x2_to_1x3( C1L,  /**/ C1R,        &C10, /**/ &C11, &C12,
                           b, FLA_RIGHT );
    /*--------------------------------------------------------------*/
    /*    C11 =  A1 * B1 + C11;    */
    ENQUEUE_FLA_Gemm( A1, B1, C11 );
    /*--------------------------------------------------------------*/
    FLA_Cont_with_1x3_to_1x2( &BL,  /**/ &BR,        B0, B1, /**/ B2,
                              FLA_LEFT );
    FLA_Cont_with_1x3_to_1x2( &C1L,  /**/ &C1R,        C10, C11, /**/ C12,
                              FLA_LEFT );
  }
  return FLA_SUCCESS;
}
```

Figure 14: The code for the improved implementation of FLA_Gemm_n (Figure 13). When the tasks are dequeued to threads, an implementation similar to Figure 6 and 8 is run, which uses the pre-packed portions of B.

show that the low-level kernels should be accessible to obtain higher performance. Lastly, by using multiple partitioning schemes, two-dimensional work partitioning can be be shown to have superior performance than one-dimensional partitioning.

The machine from which the following performance data was obtained is a 16 CPU Itanium2 server. It is an SGI Altix ccNUMA system with eight dual-processor nodes. Each of processor shares a pool of 2GB of local memory with the other processors to form a logically contiguous pool of 32GB. Each processor runs at 1.5 GHz with 4 FLOPS being performed per cycle, so an optimal performance of 6 GFLOPS is obtainable per processor. Hence, the peak attainable performance of the machine is 96 GFLOPS, which is shown at the top of the vertical axis.

All experiments use $k = 256$, which is an optimal choice for the GotoBLAS on an Itanium2 architecture. m and n are varied from 0 to 6000. The leading dimensions of A and C are made to be 6000 by embedding them in larger matrices. The leading dimension of B is equal to the k dimension. Thus, the m and n dimension are varied to be equal to the dimension shown on the horizontal axis.

The data partitioning schemes shown are chosen because they are higher-performing for some range of matrices. All partitioning schemes were tested, but some performed poorly for all matrix sizes and are, therefore, of no concern here.

The multithreaded GotoBLAS implementation of GEMM was also tested for all problem sizes and the comparison between it and the two implementations discussed above is shown.

Comparing the results seen in Figure 15, it can be seen that using the low-level kernels to reduce packing results in a relatively significant performance gain over the performance of the standard implementation.

In fact in Figure 1, the performance of the improved version is shown to be roughly equal to if not greater than that of the GotoBLAS for the three optimal partitioning schemes. This shows that composing GEMM from low-level kernels is as efficient as the GotoBLAS in not only sequential implementations but also in parallel implementations. This also shows that using two-dimensional work partitioning is superior to one-dimensional work partitioning for a range of problem sizes.

It is important to notice on the graphs in Figure 15 is that the redundant copying of partitions of B

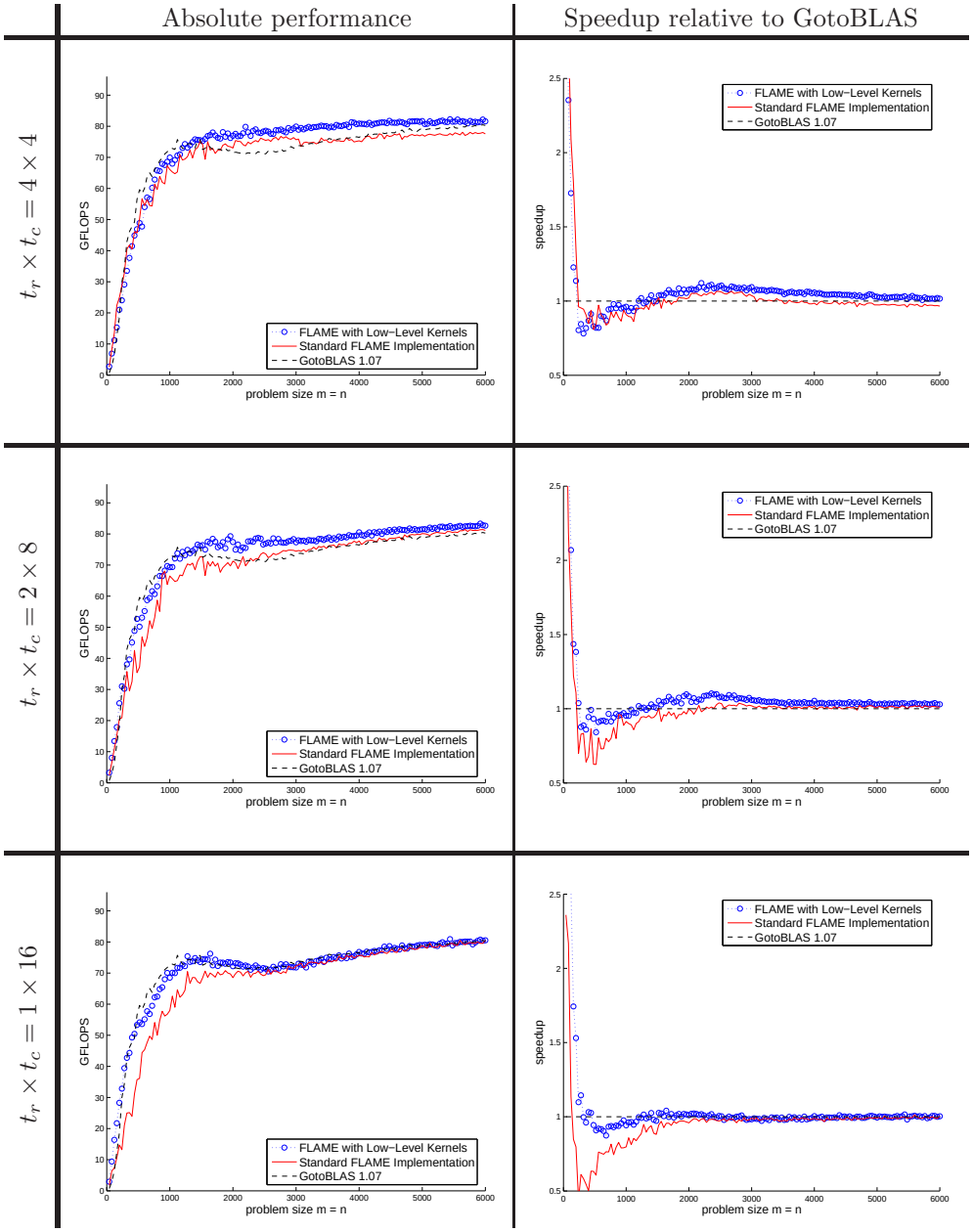


Figure 15: The results of the top-performing partitioning schemes. The standard and improved implementations are shown as well as the multi-threaded GotoBLAS implementation. The left column shows absolute performance. The right column shows performance relative to the GotoBLAS.

results in an unnecessary performance hit. As stated above as t_r increases, the performance hit due to re-packing increases. Hence, low-level kernels must be used to prevent this hit.

In these graphs, the GotoBLAS implementation is (rougly) equal in performance to the $t = 1 \times 16$ partition. This is due to the fact that GotoBLAS actually blocks with this algorithm. Therefore, as seen in the sequential version (Figure 9 and Figure 10), the performance of GEMM implemented using FLAME/C is roughly the same as GotoBLAS when using low-level kernels.

Notice in Figure 1 that for some matrix sizes, the two-dimensional partitioning schemes are superior to the one-dimensional scheme. Therefore, we can see that two-dimensional partitioning must be used to achieve optimal performance for the breadth of possible matrix sizes. This is important to note because some implementations do not consider this.

5 Conclusion

As seen in Section 3, the highly-optimized GotoBLAS GEMM can be mimicked with near-equal performance using high level algorithms and code developed in FLAME. Doing so requires access to the low-level kernels of the GotoBLAS, though, to control how data is packed. The GotoBLAS implementation includes three low-level kernels, two for packing portions of the operand matrices into physically contiguous memory and one for multiplying the packed matrices. Without access to the kernels, redundant packing overhead is incurred with calls to GEMM, which significantly diminished performance.

Similarly, Section 4, shows that a high level algorithm for data partitioning can be implemented using FLAME workqueuing to efficiently utilize systems that run multiple threads at once. Again, access to the low-level kernels must be available to prevent data from being needlessly re-packed accross threads.

Thus, the commonly-accepted view of the BLAS as atomic can lead to unnecessary overhead. By using the low-level kernels of the GotoBLAS, this overhead can be prevented, which allows high-level implementations to perform as well as the highly-tuned implementations of the GotoBLAS. For these reasons it is clear that a standardized interface to the low-level kernels of BLAS operations should be created. Much as the interfaces to the BLAS functions are standardized, so, too, should the low-level interfaces be standardized. Not only should they be made for the GEMM operation, but also for other BLAS operations since they are all generally composed of low-level kernels as well.

6 Acknowledgments

I would like to thank Professor Robert van de Geijn for the tremendous amount of help he has given me throughout my research experience. Furthermore, I thank Field Van Zee for everything he has done. My relative ignorance of Linux would have made my experience very rough if he had not been so helpful. Thanks also go to Kazushige Goto for explaining so many of the undocumented features and behaviors of the GotoBLAS. I also thank Gregorio Quintana-Ortí and Universidad Jaume I for the use of and help with the 16 CPU server on which the parallel code was tested. Additionally, thanks go to Hewlett Packard for its hadward donation that allowed me to perform much of the testing for my research and to Intel for its additional funding used to purchase hardware used for testing. This research was partially sponsored by NSF grant CCF-0540926. Any opinions, findings, and conclusions of recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF). Finally, I would give my gratitude to the Department of Computer Sciences for the Undergraduate Reasearch Opportunities Program award which helped me to perform the research in this paper.

References

[1] Paolo Bientinesi, Kazushige Goto, Tze Meng Low, Enrique Quintana-Orti, Robert van de Geijn, and Field Van Zee. Flame 2005 prospectus: Towards the final generation of dense linear algebra libraries. Technical Report FLAME Working Note 15, CS-TR-05-15, Department of Computer Sciences, The University of Texas at Austin, December 2005. `http://www.cs.utexas.edu/users/flame/pubs/`.

[2] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[3] Jeff Bilmes, Krste Asanović, Chee Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[4] Kazushige Goto and Robert van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.* to appear.

[5] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001.

[6] Tze Meng Low, Kent Milfeld, Robert van de Geijn, and Field Van Zee. Parallelizing FLAME code with OpenMP task queues. Technical Report FLAME Working Note 15, CS-TR-04-50, Department of Computer Sciences, The University of Texas at Austin, December 2005. `http://www.cs.utexas.edu/users/flame/pubs/`.

[7] Bryan Marker, Field G. Van Zee, Kazushige Goto, Gregorio Quintant-Ortí, and Robert A. van de Geijn. Toward scalable matrix multiply on multithreaded architectures. 2007. Accepted to Euro-Par.

[8] Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME API. *ACM Trans. Math. Soft.* submitted.

[9] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

[10] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Think globally, search locally. In *International Conference on Supercomputing*, 2005.