GotoBLAS - Anatomy of a fast matrix multiplication

High performance libraries in computational science

Alexander Krivutsenko
Computational Science and Engineering
Technical University Munich, 2008

Abstract

In modern world of high performance computing Mr. Kazushige Goto has became a legend mostly by his work on the GotoBLAS library which is currently the fastest implementation of BLAS routines. This paper is an attempt to summarize theoretical and practical approaches which were used to develop high performance BLAS code. As it is shown here such ideas as implementation analysis and efficient memory usage are useful for many real world problems although the paper is focused on matrix multiplication implementations.

1. INTRODUCTION

Linear algebra plays important role in scientific applications. Matrices and operations on them are frequently used in mathematical modeling for the wide range of physical processes and systems. Application areas include physics, electrical engineering, economics, biology, computer science, etc. Considering the importance of the efficiency of the matrix calculations many different open source and proprietary libraries were developed by the scientific community. The process of development of new approaches does not stop, since the hardware environments are changing frequently and the most effective solutions are implemented using machine dependent techniques.

Eventually the work of different researchers and teams has resulted in appearance of the BLAS application programming interface, which is known as Basic Linear Algebra Subprograms. BLAS is thought to be a general interface to linear algebra procedures. Currently there are several BLAS libraries available from various hardware manufacturers and also from the open source community. Such libraries are used on wide range of hardware, from graphics adapters to high scale supercomputers in performance critical applications thus it is critical for the library to implement linear algebra routines as efficient as possible.

One of the famous researchers in the area of implementations of linear algebra procedures is Kazushige Goto which has developed his version of BLAS library currently known as GotoBLAS. Nowadays the library outperforms most of the competitors in terms of speed because the code, written by Kazushige Goto is handcrafted for many various hardware architectures. The basic subroutines, also called “kernels” are developed in assembly language and tailored for specific mathematical applications. There are different versions of code specially made for working with symmetric, triangular, quadratic matrices. In 2003 the GotoBLAS code was used by 7 of the 10 fastest supercomputers in the world.

While first part of the paper is focused on introduction to BLAS and shows various naive implementations the second part explains theoretical and practical concepts behind the fastest linear algebra code.

2. BLAS

The idea behind this term has resulted in creation of the universal application programming interface for the developers of applications which use linear algebra [3]. Operations are divided into different sets, also called “levels” in BLAS terminology. Three different levels are chosen in such a way that higher level operations can be implemented in terms of lower level routines.

Level 1 consists of operations on vectors, such as norm and dot product.
The general representation of level 1 operations is:

$$x = ax + y$$

where x and y are vectors and a is a scalar.
The notion implies that result of the operation is placed into the location of the original x vector
Multiplication of vector by scalar is also a Level 1 operation.

Level 2 operations make basis for matrix - vector calculations in form:

$$y = aAx + by$$

The resulting vector is the product of scalar a , matrix A and vector x which is added to the product of scalar b and vector y .

Level 3 operations are matrix – matrix calculations:

$$C = aAB + bC$$

where a and b are scalars, and A , B , C are matrices. Again, the result of such operation is placed into C location, overwriting the previous data.

Such distribution of various linear algebra operations also allows to define upper bounds for the operational complexity for all three levels. Assuming that operations are performed on input data with length N the following estimations are valid.

Level	Data movements	Floating point operations	Example of the BLAS primitive
Level 1 BLAS	O(N)	O(N)	DDOT
Level 2 BLAS	O(N²)	O(N²)	DGEMV
Level 3 BLAS	O(N²)	O(N³)	DGEMM

Table 1. Three levels of BLAS operations

Note that order number of each level corresponds to the power of N in the estimation of the floating point operations needed to perform such operation. Thus it is easy to think that level 2 routines require at worse O(N²) floating point operations to complete.

All subprograms in BLAS ideology are defined using strict notation so that name unique

implies the operation and data input.
If the name of the operation is expressed in form as ABC then
A denotes input data prevision such as:

- "S" for single real data
- "D" for double real
- etc.

B part denotes the nature of input data such as

- "GE" for general matrices
- "SY" for symmetric, etc.

C part defines the operation performed on input data, ie

- "MM" is matrix multiplication
- "MV" is matrix vector multiplication.

Since the basic topic of this paper is matrix - matrix multiplication we will focus mostly on routine named "DGEMM" which implies matrix multiplication with general matrix input with double precision. Examples of other BLAS names are written in the Table 1.

The importance of high speed BLAS implementations is based on fact that many higher level problems can be solved using BLAS operations. Thus solution to the bigger problem depends on the performance of the lower level operations such as matrix-matrix multiplication. Furthermore, it has become a generally accepted practice to use GEMM (general matrix multiplication) for getting high performance for other matrix-matrix operations such as symmetric matrix-matrix multiply, symmetric rank-k update, symmetric rank-2k update, triangular matrix-matrix multiply, and triangular solve with multiple right-hand sides [2].

3. EVOLUTION OF MATRIX MULTIPLICATION ALGORITHMS

Traditionally matrix multiplications were used since the birth of first computers to model systems and solve linear equations. The time complexity ($O(N^3)$) for a single matrix – matrix multiplication placed great restrictions on the size of input data limiting the usage of implementations to small scientific problems.

The evolution of matrix multiplications advances to the direction of providing better data locality of the input data so that various high performance hardware features can be utilized.
A straightforward implementation approach is suffering because of low data locality (*Figure 1*).

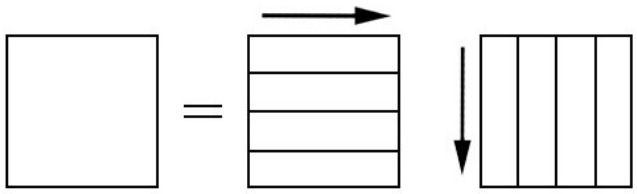

Figure 1. Naive matrix multiplication implementation

The parallelization of such code does not bring a lot of benefits since various functional units must access the same data. The cache is also not utilized properly since elements of matrix in the same column (row) are always far away from each other. The main problem of this solutions is the row major or column major placement of the elements in both matrices which requires jumping in memory for iterating elements in the same column or row.

The more efficient solution would be performing transposition of one matrix before multiplying (*Figure 2*)

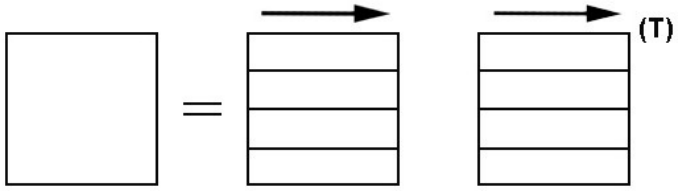

Figure 2. Improved naive matrix multiplication implementation

This approach delivers slightly better performance since a part of the row (column) is now placed into the cache line and is being accessed several times.

Another more efficient solution, which is used currently in many non-scientific application is based on the following idea. When considering a product of two matrices A and B one can see that :

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{bmatrix} = \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \\ \mathbf{B}_3 \end{bmatrix}$$

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} & a_{11}b_{14} + a_{12}b_{24} + a_{13}b_{34} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} & a_{21}b_{14} + a_{22}b_{24} + a_{23}b_{34} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} a_{11}\mathbf{B}_1 + a_{12}\mathbf{B}_2 + a_{13}\mathbf{B}_3 \\ a_{21}\mathbf{B}_1 + a_{22}\mathbf{B}_2 + a_{23}\mathbf{B}_3 \end{bmatrix}$$

Figure 3. Iterative algorithm for matrix multiplication

The sample code implementation is shown below:

```
//this sample code assumes that original matrices are
// double a[N][M];
// double b[M][N];
void dgemm(double * c[], double * a[], double b[], int N, int M)
{
        memset(c, 0, sizeof(int) * N * N);
        double * pLeft = &a[0][0];
        double * pDstLim = &c[0][0] + N * N;
        double * pRightLim = &b[0][0] + N * M;
        for(int * pDst = &c[0][0]; pDst< pDstLim; pDst += N )
        {
                double * pRight = &b[0][0];
                while(pRight < pRightLim)
                {
```

```
                    double * pTmpLim = pDst + N;
                    while(pDst < pTmpLim)
                    {
                     *(pDst++) +=  *pLeft * *(pRight++);
                    }
                    ++pLeft;
                    pDst -= N;
                }
        }
}
```

The advantage of this format is that instead of multiplying rows of A times columns of B, we multiply the elements of A times the rows of B. This means that we don't have to transpose a matrix to find the result [4].

The solution explained above delivers much better performance than its predecessors. However, it still fails to multiply big matrices efficiently because elements of the whole matrix B are traversed in the increasing order. Such solution works good on relatively small matrices but when matrix size becomes bigger the performance degrades. Also such approach is not sufficient for parallel implementations making it useless for high performance supercomputers .

4. BLOCKING

The idea behind faster solutions require understanding of how a bigger matrix can be separated into smaller inner matrices which can be multiplied independently. Such operation is called blocking.

$$\begin{matrix} A_{11}\, A_{12} \ldots A_{1k} \\ \ldots \\ A_{k1}\, A_{k2} \ldots A_{kk} \end{matrix} \quad x \quad \begin{matrix} B_{11}\, B_{12} \ldots B_{1k} \\ \ldots \\ B_{k1}\, B_{k2} \ldots B_{kk} \end{matrix} \quad = \quad \begin{matrix} C_{11}\, C_{12} \ldots C_{1k} \\ \ldots \\ C_{k1}\, C_{k2} \ldots C_{kk} \end{matrix}$$

where every block Cij of matrix C is determined with the rule:

$$C_{ij} = \sum_{t=1}^{k} A_{it}\, B_{tj}$$

The blocking is applied in all high performance implementations because it is obvious that for arbitrary sized input such mechanism is needed to use expensive memory resources efficiently.

Various algorithms have been introduced for blocking matrix multiplication. The main advantage of the blocking operation is that such approach allows to distribute work among different processors utilizing various parallel architectures. One of the most famous such algorithms is the Strassen's algorithm, which recursively repartitions the matrices into 2 by 2 blocks and then multiplies the blocks using 7 multiplications and 18 additions and subtractions. This process needs $O(N^{2.8074})$ steps which is a good improvement over the original $O(N^3)$. Another advanced algorithm by Coppersmith and Winograd's allows to improve the runtime complexity up to $O(N^{2.376})$.

While ideas for blocking algorithms have been already developed the main problem for the developers is how to perform such blocking to utilize the hardware resources most efficiently.
The author of GotoBLAS has used „divide and conquer“ algorithmic decomposition to replace one matrix matrix multiplication with subsequent matrix.- vector operations.

5. IMPLEMENTATION ANALYSIS

It has become clear for the scientific community that there is a need for uniform theoretical basis to measure the efficiency of various implementations. As a first step of developing such measurement the general hardware environment was considered.

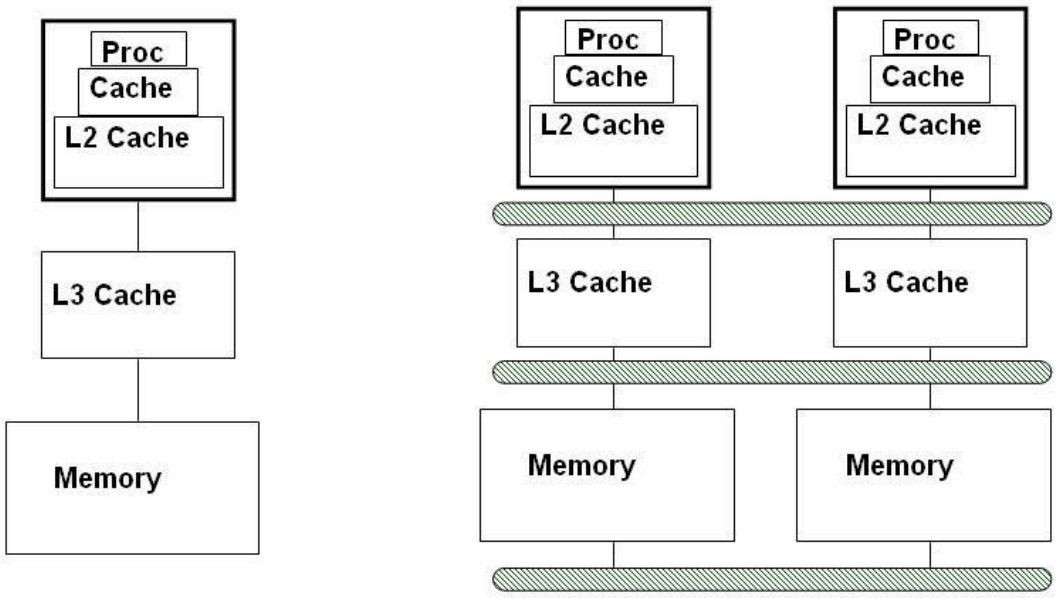

Figure 4. Data storage hierarchy in conventional and parallel architecture

As it is shown above (*Figure 4*) there are multiple interconnections between various memory layers. Soon computer scientists realized that the amount of movements of data between memory layers is as much important as the amount of arithmetic operations performed on that data. Furthermore in current hardware environments different memory units can have slower access times thus minimizing data movements between such layers is extremely important problem.

Theoretically the problem of minimizing data movements can be expressed in the following form. When implementing a computational algorithm it is useful to consider such qualitative factors as:

m: number of memory elements (words) moved between fast and slow memory
t(m): time for slow memory operation
f: number of arithmetic operations
t(f): time per arithmetic operation (much smaller than t(m))

Based on such metrics for any algorithm or implementation the *Computational Intensity factor* can be derived:

Computational Intensity:

$$q = f / m$$

which means average number of flops per slow element access.
When comparing two algorithms or their practical implementation it can be realized that the higher the computational intensity, the faster is the implementation.
This fact can be also derived from the formulas. The minimum possible time for any algorithm can be expressed as

f* t(f)

when all data is located in fast memory

However, in practical situations the actual time is

f * t(f) + m * t(m) = f * t(f) * (1 + t(m)/t(f) * 1/q)

Looking at the right hand side one can see that with q going to the infinity the overall execution time is converging to the estimated minimum. Thus a good implementation has higher q.

As an example of such analysis we consider the improved naive implementation of basic DGEMM routine.

```
Implementation of C = C + A*B
(DGEMM)

for i = 1 to n
        {read row i of A into fast memory}
        for j = 1 to n
        {read C(i,j) into fast memory}
        {read column j of B into fast memory}
        for k = 1 to n
                C(i,j) = C(i,j) + A(i,k) * B(k,j)
                {write C(i,j) back to slow memory}
```

The naive implementation of matrix multiplication has
$m = n^3$ (read each column of B n times)
$+ n^2$ (read each row of A once)
$+ 2 * n^2$ (read and write each element of C once)
$= n^3 + 3 * n^2$

So computational intensity parameter for the improved naive implementation is

$q = f / m = 2 * n^3 / (n^3 + 3 * n^2) \sim= 2$

This gives us a ratio between floating point operations and slow memory access. With the common sense one can realize that approximately for any two arithmetic operation such naive implementation would require one access to the slow memory. This is factor is converging to 2 so there is no dependence to the size of the input data.

Now let us consider the blocking DGEMM implementation algorithm:

Assume that A,B,C are N by N matrices of b by b blocks where b=n / N is the block size

```
for i = 1 to N
    for j = 1 to N
        {read block C(i,j) into fast memory}
        for k = 1 to N
            {read block A(i,k) into fast memory}
            {read block B(k,j) into fast memory}
```

*C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}*
{write block C(i,j) back to slow memory}

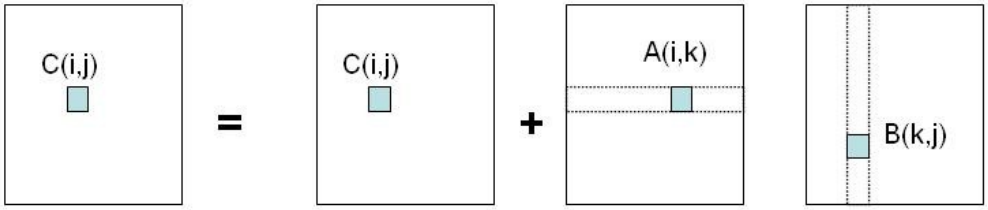

Figure 5: Blocking DGEMM implementation

For the given blocking implementation the following figures apply:

m = N* n² (read a block of B n³ times (n³ * n/N * n/N))
+ N* n² read a block of A N3 times
+ 2* n² read and write each block of C once
= (2N + 2) * n²

So computational intensity

$$q = f / m = 2 n^3 / ((2N + 2) * n^2) \sim= n / N = b \text{ for large n}$$

Since there is a clear dependence between computational intensity and the block size, in practice we can improve the performance of the algorithm by using greater block size of course if our hardware allows such increase. Now it is obvious that such blocking algorithm is faster than naive implementation.

As we will see below the author of GotoBLAS has considered such theoretical way to determine the efficiency of the implementation. Unfortunately most of the real life architectures are not so simple as we considered in this chapter although the same approach can be also applied with memory architectures of a higher dimensions. The key point behind the implementation of the blocking matrix multiplication algorithm is to derive such block size which will require minimum memory access while still achieving high number of floating point operations.

6. DATA MOVEMENTS IN GOTOBLAS

Any approach to repartition the input data to the blocks of the appropriate size still needs an efficient low level processing of such blocks. When the total work can be shared among multiple functional units the second critical time consuming operation is the fetch of the data from the main memory to the system cache memory. To address this issue a layered memory model has been introduced [5].

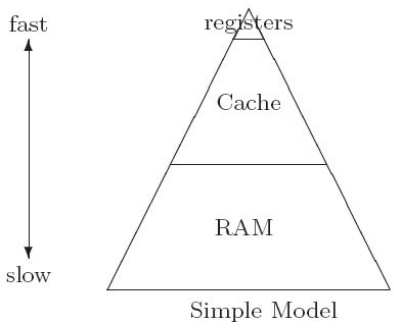

Figure 6. Layered memory model

In such diagram it is implied that the higher is the resource, the less is the amount of the available memory. In the ideal case the number of floating point operations must be significantly greater then the amount of needed data movements among the memory resources because every data movement can a lot of time especially when cache misses are involved.. As it has been shown in Table 1 the Level 3 BLAS operations need $O(N^2)$ total data movements thus enabling various data blocking approaches for the input. It is obvious that in other architectures such as NUMA the memory resources scheme can have different structure but the GotoBLAS implementation is primarily aimed for the traditional memory architectures.

The idea behind such division is that in every point of time some data can be located on one of the levels in the memory hierarchy and this data must be moved to the higher or lower level memory resource [6]. The computational gain in the performance comes from the fact that optimized code can be concentrated on data which is located on a single memory level while ignoring other levels. This requires optimal data „blocking“ mechanism to minimize data movements between adjacent memory levels. As a result to this approach a number of algorithms has been developed which allowed to utilize better the more expensive memory resources.

While accepting the memory architecture shown in the figure above the author of GotoBLAS has noticed that such approach is too simplistic for the completely successful optimization because it ignores the issues related to Translation Look-aside Buffer (TLB).
This component of the memory structure has been ignored by the developers of other high performance computing libraries thus allowing increase in the performance in the Goto BLAS code. This resulted in a refined scheme which was considered for creation of the code:

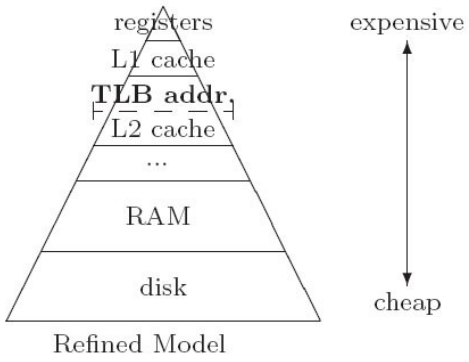

Figure 7. GotoBLAS memory model

Translation Look-aside Buffer is a fully associative cache which is specifically aimed to translate virtual memory addresses to the physical memory addresses. A TLB is used in conjunction with cache whose tags are based on virtual addresses. Each virtual address requested by the application is presented simultaneously to the TLB and to the cache so that cache access and the virtual to physical address translation can be done in parallel ("on the side"). If the requested address is not cached then the physical address is used to locate the data in main memory.
Every miss in the TLB covered address space is 10 to 60 times slower than a hit. For the average compiler generated code the probability of the miss ranges from 0.01% to 1%. Furthermore the miss in the TLB cache is more expensive than mis in the cache memory because it causes CPU to stall.

It has been observed by the developer of the GotoBLAS is that the amount of data that can be addressed by the TLB is the limiting factor for the size of the block of the input matrix. An architectural goal was to design such algorithm that leaves a part of the input matrix A in the TLB cache as long as the data is needed but then it never enters TLB cache again (here it is meant that for a single GEBP operation it is enough to store block of A only once in a cache memory).

Another interesting fact is that the ratio between the rate of floating point operations and the rate at which floating point numbers can be streamed from the level-2 (L2) cache to registers is

typically relatively small. This means that executing operations on data which resides in L2 cache can be a major algorithmic goal. To achieve this goal one must put the block of the input data directly into the L2 cache. The author of GotoBLAS has developed a scheme were a bigger part of the input matrix A is placed into the L2 cache thus allowing to utilize the observed ratio between floating point operations and data movements between L2 cache and registers.

7. DECOMPOSITION IN GOTOBLAS

Assuming the standard matrix multiplication is performed using GotoBLAS code:

$$C = AxB$$

where A and B are input matrices with no restrictions on their size.

It has been shown in [1] that if the input matrices are stored in column major order then the most efficient way to calculate their product is to decompose original problem into block – panel multiplications.
For the general matrices A and B and their product C the following steps are applied.

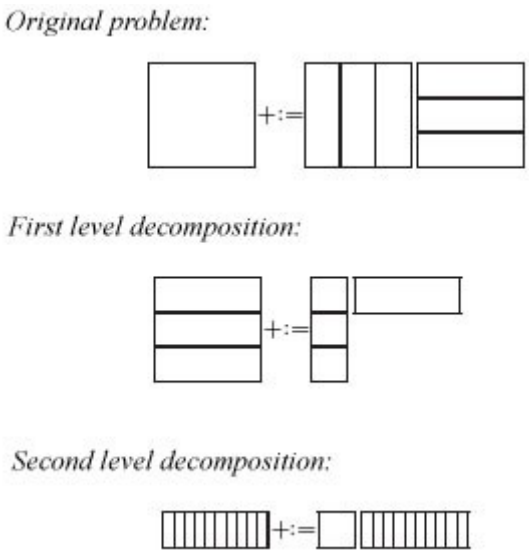

Figure 8. Operational decomposition in GotoBLAS

On the first level of decomposition the operations are performed on submatrices, with one dimension dominating the other. Such matrices are called „panels“ and respective operations are called general panel panel multiplication (GEPP).
On the second level of decomposition the data is separated into smallest blocks of such size which guarantees efficient in-cache location of the operands. There are panels (column and row vectors of the original submatrices) and there is a block (complete submatrix of original A matrix). The multiplication of block and panel is called GEBP.

The packing algorithm can be described in the following way:

1) Original matrix C is divided in such a way that submatrices (panels) with dominant row size are created in consecutive main memory locations.

2) Original matrix A is packed into square submatrices (blocks) which are placed into consecutive locations in main memory.

3) Original matrix B is packed to a set of submatrices, which have dominant row size.

4) On the second level decomposition the panel of C is again divided into column vectors in such a way that each column vector fits into the cache among with data from A and B matrices.

5) The entire square block of A is placed into the cache memory. This requires some transposition of the original data. This step is important to utilize the TLB cache since the entire block of A is used in the cycle for the GEBP operation and it must stay in TLB cache as long as possible.

6) Column vector of the panel of the original B matrix is placed into the cache memory.

When such packing is performed this guarantees the optimal memory usage.
In the main cycle only GEBP operations are performed on the packed buffers..The floating point calculations access only those parts of the matrices which were placed into the cache memory.

It has been proven in [1] that to utilize the resources in efficient way the input matrix A should reside in cache as much as possible and should be roughly square.
At the same time there should be enough space left in cache for at least one column of B and C.
In such approach the overhead for memory operations adds only about 10% to the entire computational costs.

It is obvious that the entire process is based on the performance of the lower operations on panels and blocks (GEPP and GEBP) since such these are executed most frequently. Primitives GEPP and GEBP are optimized in such a way that register memory is utilized as much as possible. For this purposes the handcrafted assembly code is written.

8. BEYOND THE GENERAL MATRIX MULTIPLICATION

It became a general practice to implement other Level 3 operations in terms of GEMM.
Furthermore it was shown in [2] that all Level 3 operations can be decomposed to a set of GEBP, GEPP, DOT operations. Lower level GEPP and GEBP and other similar operations are the building blocks for higher level matrix calculations.
However traditional implementations have some complications which are based on the fact that data is being copied unnecessary between memory resources. For example, recursive blocking algorithms pack the panels of original data multiple times as part of the individual calls to GEMM, which itself is cast in terms of GEPP operations.

Kazushige Goto has proposed more efficient way to perform other matrix operations such as SYRK, SYR2K, TRMM, SYMM, TRSM. The suggestion is to make packing routines as primitives for high level libraries so that similar packing can work on different types of input data. At the same time kernel calculation routines will work on the same packed data enabling the highest optimization for the entire process.

Such implementation shows better performance on various platforms. When compared with other libraries GotoBLAS delivers strong results which outperforms other implementations.
For example, when modified GEMM algorithm is used to implement triangular matrix multiplication the following performance figures are obtained [2]:

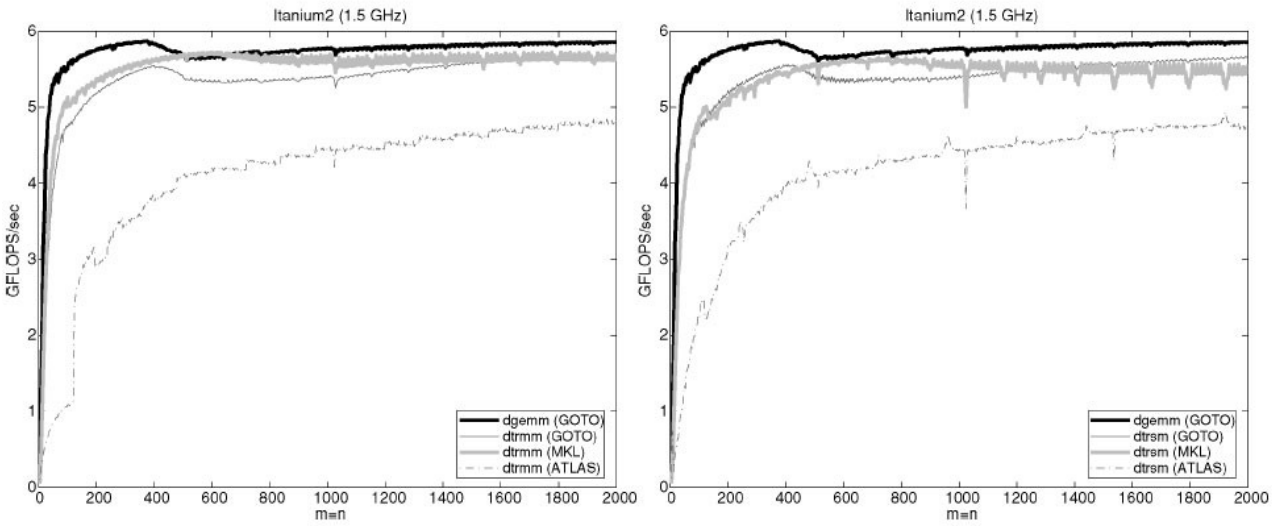

Figure 9. Performance figures

9. CONCLUSION

Matrix multiplication algorithms have long evolution history. Currently the most efficient implementations are developed with the help of various hardware mechanisms. With the fast changing hardware environments it is expected to have new, faster implementations for almost every matrix operation.

Author of GotoBLAS has obtained impressive results not only by using hardware specific methods but also by developing the optimal algorithm for data movements between memory resources. Such approach will be useful on most hardware architectures until the memory model is redesigned. This makes GotoBLAS implementation one of the leading high performance libraries for the nearest years for matrix related computations.

REFERENCES

[1] Kazushige Goto, Robert A. Van De Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. ACM Transactions on Mathematical Software, Vol. 34

[2] Kazushige Goto, Robert A. Van De Geijn. 2008. High-Performance Implementation of the Level-3 BLAS. ACM Transactions on Mathematical Software, Vol. 35

[3] BLAS support. The GNU Scientific Library
http://www.gnu.org/software/gsl/manual/html_node/BLAS-Support.html

[4] Algorithms and Special Topics
http://developer.apple.com/hardwaredrivers/ve/algorithms.html#Matrix_Multiplication
2005 Apple Computer, Inc.

[5] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. June 2001. High-Performance Matrix Multiplication Algorithms for Architectures with Hierarchical Memories, FLAME Working Note #4,.

[6] G.H. Golub and C. Van Loan, The Johns Hopkins. 1989. Matrix Computations. University Press, Baltimore MD, 2nd Ed..