Optimization of Triangular Matrix Functions in BLAS Library on Loongson2F

Yun Xu, Mingzhi Shao, Da Teng

Optimization of Triangular Matrix Functions in BLAS Library on Loongson2F

Yun Xu[1,2], Mingzhi Shao[1,2], and Da Teng[1,2]

[1] School of Computer Science and Technology, University of Science and Technology of China,Hefei, China
xuyund@ustc.edu.cn, smz55@mail.ustc.edu.cn, tengda@mail.ustc.edu.cn
[2] Anhui Province Key Laboratory of High Performance Computing
Hefei, China

Abstract. BLAS (Basic Linear Algebra Subprograms) plays a very important role in scientific computing and engineering applications. ATLAS is often recommended as a way to generate an optimized BLAS library. Based on ATLAS, this paper optimizes the algorithms of triangular matrix functions on 750 MHZ Loongson 2F processor-specific architecture. Using loop unrolling, instruction scheduling and data pre-fetching techniques, computing time and memory access delay are both reduced, and thus the performance of functions is improved. Experimental results indicate that these optimization techniques can effectively reduce the running time of functions. After optimization, double-precision type function of TRSM has the speed of 1300Mflops, while single-precision type function has the speed of 1800Mflops. Compared with ATLAS, the performance of function TRSM is improved by 50% to 60%, even by 100% to 200% under small-scale input.

Keywords: BLAS; ATLAS; triangular matrix function; loop unrolling; data pre-fetching

1 Introduction

In the contemporary scientific engineering, most of the running time is spent on basic linear algebra functions. A lot of software related to matrix computing invokes functions in BLAS [1] (Basic Linear Algebra Subprograms). As a consequence, it is imperative to optimize the BLAS libraries based on a specific machine to fully utilize its hardware resource. KD-50-I is a high performance computer that employs China's Loongson 2F superscalar CPU, which has the advantage of low power, low cost and high integration. Our benchmark for high performance computer KD-50-I is HPL (High Performance Linpack), which is implemented by invoking functions in the BLAS library. Therefore, the efficiency of functions in the BLAS library can directly affect the performance of the KD-50-I system.

The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. The

Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations. In this paper we mainly aim at the optimization of functions that computes triangular-matrix and vector in level 2, and the TRMM and TRSM function that implement triangular-matrix and matrix operations in level 3. These functions take up about one third of the BLAS library.

On optimization of linear algebra library, contemporary research focuses on an algorithmic level [2, 3]. In a period of time that CPU reads one byte from memory, it can execute hundreds of instructions. Consequently, the bottleneck of optimizing functions is not computing time but memory access delay. In order to reduce memory access delay, RA Chowdhury [4] proposed a method that extends the cache-oblivious framework to solve The Gaussian Elimination Paradigm (GEP); Tze Meng Low [5] provided with a high-efficiency blocking algorithm for functions in level 3. ATLAS [6, 7] (Automatically Tuned Linear Algebra Software) is one of the matrix packages [8,9]. ATLAS is portable BLAS software which firstly tests hardware parameters and then optimizes some dense-matrix functions using basic optimizing techniques. ATLAS can optimize basic BLAS functions automatically upon the parameters of cache capacity and memory access delay that ATLAS has tested. However, there still exist unknown parameters of specific architectures, e.g. pipeline structure. Thus, there is room for optimization of codes that ATLAS generates.

Based on ATLAS, we further optimize triangular-matrix functions in BLAS from an algorithmic level to reduce access delay and to improve the performance of BLAS, using general optimizing techniques (such as matrix blocking, loop unrolling) and optimizing techniques specific on Loongson 2F (such as multiply-add instruction, instruction scheduling, data pre-fetching).

2 Triangular-matrix Functions and ATLAS Methods

There are 8 triangular-matrix functions in BLAS, however, here we only take the TRSM function to illustrate optimizing methods and results.

2.1 TRSM and Symbols

TRSM implements multiplication of inverse of triangular-matrix A and matrix B, as formula(1) illustrates,

$$B \leftarrow \alpha op(A)B \qquad or \qquad B \leftarrow \alpha B op(A^{-1}) \tag{1}$$

where α(ALPHA, a scalar) is an extension factor, B is an M-by-N matrix, A is an upper (or lower) triangular (or unitriangular) matrix, and $op(A)$ can be A, the transpose of A, or the conjugate transpose of A. If $op(A)$ is on the left of B (left multiplier), A is M-by-M;if not, A is N-by-N.

The declaration of TRSM function is xTRSM (ORDET, SIDE, UPLO, TRANS, DIAG, M, N, ALPHA, A, LDA, B, LDB), where x represents s, d, c or z which

respectively stands for single precision float data type, double precision float data type, single precision complex data type, and double precision complex data type.

2.2 ATLAS Methods

In terms of TRSM, ATLAS employs the solution method for linear equations. ATLAS has made some basic optimizations toward TRSM as follows:

Matrix blocking ATLAS optimizes TRSM by blocking, setting block size of real numbers as 4 and block size of complex numbers as 8, the same as coefficients of loop unrolling. After matrix blocking, triangular matrix A is divided into several smaller rectangular matrices and triangular matrices, where smaller rectangular matrices can be solved by invoking GEMM function and smaller triangular matrices can be solved directly.

Copy and Partial Matrix Transpose Through copy and partial matrix transpose technique, ATLAS transfers a matrix into a transposed or non-transposed status, which changes data storage order and further improves the performance of functions.

Loop Unrolling ATLAS has devised the trsmKL and trsmKR function that operate on real numbers, and the CtrsmK function that operates on complex numbers, where trsmKL and trsmKR unroll a loop by $8 \times 1 \times 1$ and CtrsmK unrolls all of the two inner loops.

3 General optimizing techniques

3.1 Adjusting Block Size

Matrix blocking is a widely applied optimizing technique to enhance storage availability. It reduces local data sets to avoid conflicts. The matrix blocking algorithm is to partition the sub-data blocks of a matrix, in order to reuse data that are in the cache.

We adjust the size of the blocks to a proper value so that each data set could be stored in a buffer, which reduces extra cost of blocking and ensures relatively low conflicts. The left multiplication format of dTRSM exemplifies the specific steps of matrix blocking. As Fig. 1 illustrates, triangular matrix $A_{M\times M}$ is spitted into several smaller triangular matrices $A'_{RB\times RB}$(grey region in Matrix A in Fig. 1) and several smaller rectangular matrices $A''_{RB\times M'}$(white region in Matrix A in Fig. 1, where M' is a variable and $RB \leq M' \leq M$), so each A' can be fully stored in L1 cache. As for $A'_{RB\times RB}$, $B_{M\times N}$ is split into several smaller rectangular matrices $B'_{RB\times N'}$, the counterparts of partitioned $A'_{RB\times RB}$ is solved by TRSM, and the counterparts of partitioned $A''_{RB\times RB}$ is solved by GEMM.

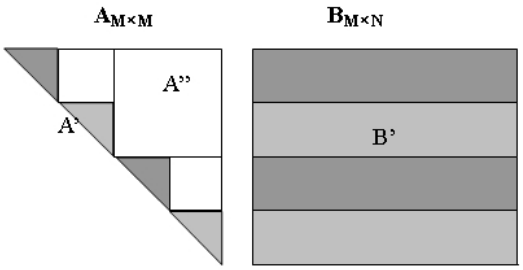

Fig. 1. Using matrix block technique to the triangular matrix A and the matrix B

As a consequence, optimization is concentrated in the multiplication operation of triangular matrix $A'_{RB\times RB}$ and $B'_{RB\times N}$. It is clear that both A' and B' can be read from cache instead of memory, which avoids conflict and enhances the performance of TRSM.

3.2 Setting Coefficients of Loop Unrolling

Loop unrolling is a common optimizing compilation technique, employed to reduce both cyclic variable operations and branch instructions. Besides, unrolling the outer loop of multi-loop can make certain data reusable. We can put these data into registers so that these data can be read directly from registers instead of memory, which lowers the requirements of communication bandwidth between register and cache as well as that between cache and memory.

Next, we analyze how varied coefficients affect performance of functions, and then specific coefficient will be chosen upon Loongson 2F architecture. We define the function ATL_dreftrsmLUNN(ref for short) that implements the upper triangular form, non-transposed form, and non-unimatrix form of partitioned dTRSM. As Algorithm 1 illustrates, this function shows how coefficients affect function performance.

There are 3 layers in the loop of algorithm 1, which are denoted by R, S, T. The ref function is attributable to the speed-up of memory access of TRSM under limitation.

The times of memory access is $(M^2+Md)N$ when multiplication operation of $M \times M$ triangular matrix A and $M \times N$ rectangular matrix B is implemented. If layer T is unrolled for α times, the times of memory access of A and B is respectively $NM(M+1)/2$ and $MN(M+1)/2\alpha$; if layer S is unrolled forβ times, the times for memory access of A and B is respectively $MN(M+1)/2\beta$ and $MN(M+1)/2$. So the sum of memory access for unrolling layer T for α times and for unrolling layer S for β times is $MN(M+1)/2a+MN(M+1)/2\beta$.

The computing complexity of dTRSM is $(M^2+2M)N$, and memory access speed is denoted by L(Mb/s), then the theoretical upper limit for the computing

Algorithm 1 Algorithm of ATL_dreftrsmLUNN

```
1: S    for(j=0;j<N;j++)
    {
2: T        for(i=M-1;i>=0;i–)
        {
          rC=A[i][i]*B[k][j];
3: R      for(k=i+1;k<M;k++)
          {
            rC-=A[i][k]*B[k][j];
          }
          B[i][j]=rC/A[i][i];
        }
    }
```

speed is:

$$speed = \frac{\text{computing complexity}}{\text{memory access}} \times L$$

$$= \frac{(M^2+2M)NL}{MN(M+1)(\alpha+beta)/(2\alpha\beta)} = \frac{2\alpha\beta(M+2)L}{(\alpha+\beta)(M+1)} \tag{2}$$

$$\approx \frac{2\alpha\beta L}{\alpha+\beta}$$

When we are deal with large-scaled data, and efficiency of ATL_dreftrsmLUNN is limited by memory access, increasing the value of α and β could elevate this upper limit of computing speed.

3.3 Other Optimizing Techniques

Except for transforming division to multiplication operation for ref, we have done the same to complex data type functions, because a multiplication operation only takes one cycle while a division operation takes tens of cycles. Taking the left multiplication form of zTRSM for example, computing each column of matrix B that stores complex numbers requires $2M$ times division operations, so the total number of operations is $2MN$. An array is necessary to place the elements in the diagonal of the triangular matrix, whose expression is $r/(r^2+i^2)$ and $i/(r^2+i^2)$ where r and i respectively stand for the real part and the imaginary part. It is not difficult to replace the former division operations by multiplication, owing to which there is totally $2M$ division operations and $2MN$ multiplication operations-executing time is shortened.

What's more, after unwinding the loops in zTRSM and cTRSM, we also unroll Mmls multiply-subtract instruction for complex numbers used as core of computation to separate real part and imaginary part, so that it is easier for further optimization with specific techniques on Loongson 2F architecture.

4 Optimization Techniques Based on Loongson 2F Architecture

Loongson is a family of general-purpose MIPS-compatible CPUs developed at the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS) in China . KD-50-I is a supercomputer with a total of more than 330 Loongson-2F CPUs. Loongson's instruction set is 64bit MIPS III compatible. It has separate 64/64 KB instruction and data L1 caches. According to literature[10], RB is set as 60 when we block the matrices in dTRSM function in section 3.1. There are 32 registers in Loongson 2F, so theoretically the number of times of loop unrolling for ATL_dreftrsmLUNN can be set as $4 \times 4 \times 2$, that is $\alpha = \beta = 4$, to acquire a good result $speed = 4L$. In the following sections, there are other special techniques based on Loongson 2F architecture.

4.1 Multiply-add Instruction

Because the traditional multiply and add instruction is RAW (read-after-write), it is often necessary to put other instructions between them to reduce pipeline idling. However, Loongson 2F is compatible with a specific multiply-add instruction, using which we can combine multiplication with add operations to improve our program.

4.2 Instruction Scheduling

Another technique is to adjust the sequence of instructions to avoid pipeline idling. Loongson Pentium Pro Architecture has 5 execution units: 2 ALUs, 2 FPUs, and 1 address generation unit (AGU). Consequently, in each execution, one load instruction has to be followed by two floating point instructions and one fixed-point math instruction, so that 4-way superscalar is strictly fitted, which is contributable to acceleration of IPC (Instructions Per Clock).

4.3 Data Pre-fetching

The prefetching instruction of Loongson instruction set can cause the reorder queue blocked, so we employ a branch prediction technique as data pre-fetching technique. After unrolling loops in algorithm 1(in Fig. 2)by $4 \times 4 \times 2$, we use data pre-fetching and instruction scheduling techniques to make instructions in each row fit for 4-way superscalar. The codes of the most inner loop are as follows:

In Algorithm 2, incA and incB respectively stands for the distance of two continuous elements in matrix A and B. The load instruction in the first part has acquired data required by the second part, and the data acquired in the second part is necessary in the first part of the next iteration. It is obvious that except rA0, rB0, rA1, rB1, rA2, rB2, rA3 and rB3, each instruction can attain the data needed at least one cycle ahead, that is, data is pre-fetched. By renaming registers and using registers in turns, dependency among instructions can be

Algorithm 2 Using data pre-fetching in loop unrolling

```
1:    rA0=* pA0; rB0=* pB0;
      rA1=* pA1; rB1=* pB1;
      rA2=* pA2; rB2=* pB2;
      rA3=* pA3; rB3=* pB3;
2:    for(k=i+1; k < M-1; k+=2)
      {
        ra0=pA0[incA];   rC00-=rA0* rB0;   rC10-=rA1* rB0;   pA0+= incA2;
        rb0=pB0[incB];   rC01-=rA0* rB1;   rC11-=rA1* rB1;   pB0+= incB2;
        ra1=pA1[incA];   rC20-=rA2* rB0;   rC21-=rA2* rB1;   pA1+= incA2;
        rb1=pB1[incB];   rC02-=rA0* rB2;   rC12-=rA1* rB2;   pB1+= incB2;
        ra2=pA2[incA];   rC22-=rA2* rB2;   rC32-=rA3* rB2;   pA2+= incA2;
        rb2=pB2[incB];   rC30-=rA3* rB0;   rC31-=rA3* rB1;   pB2+= incB2;
        ra3=pA3[incA];   rC03-=rA0* rB3;   rC13-=rA1* rB3;   pA3+= incA2;
        rb3=pB3[incB];   rC23-=rA2* rB3;   rC33-=rA3* rB3;   pB3+= incB2;
        rA0=* pA0;   rC00 -= ra0* rb0;   rC10 -= ra0* rb1;
        rB0=* pB0;   rC20 -= ra0* rb2;   rC30 -= ra0* rb3;
        rA1=* pA1;   rC01 -= ra1* rb0;   rC11 -= ra1* rb1;
        rB1=* pB1;   rC21 -= ra1* rb2;   rC31 -= ra1* rb3;
        rA2=* pA2;   rC02 -= ra2* rb0;   rC12 -= ra2* rb1;
        rB2=* pB2;   rC22 -= ra2* rb2;   rC32 -= ra2* rb3;
      }
```

decreased. For example, as algorithm 2 indicates, we employ rA0, rA1, rA2, rA3 and ra0, ra1, ra2, ra3 by rotation. The following two lines states that rA0 and ra0 fetch data in turns and they are independent of each other, so that data is successfully pre-fetched.
$ra0 = pA0[incA]; rC00- = rA0*rB0; pA0+ = incA2; \quad rA0 = *pA0; rC00- = ra0 * rb0$

5 Experimental Results and Discussion

Our experiment is implemented through the combination of repeatedly testing using one case and testing in circles using a group of cases. Specifically, every function is executed repeatedly under various data scale, until the average speed is calculated as the final result. Here we use Mflops(million floating-point operations per second) as the technical criteria. For convenience, parameters "M", "N", "LDA" and "LDB" of function TRSM is replaced by the same value "length". Configurations of compilers include set open multiply-substract instructions (-DAdd_DStringSunStyle), Linux operating system(-DATL_OS_Linux), not saving frame pointer at function call (-forit-frame-pointer), optimization level (-O3), and unrolling all the loops(-funroll-all-loops).

5.1 Optimization Results of dTRSM

The outcome of optimization on dTRSM will be discussed from two aspects: influences placed on dTRSM by each techniques under small data scale and comprehensive effect by all the techniques under large data scale.

Because implementation of functions is independently done by TRSM under small data scale, the effect of optimization is reflected directly by performance of TRSM. Fig. 2 shows the performance influenced by each technique under small data scale.

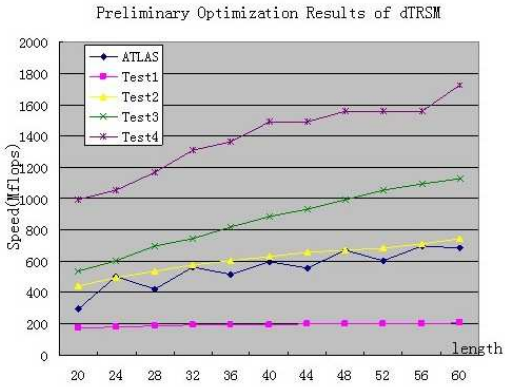

Fig. 2. Comparison of different optimization methods in terms of dTRSM function performance

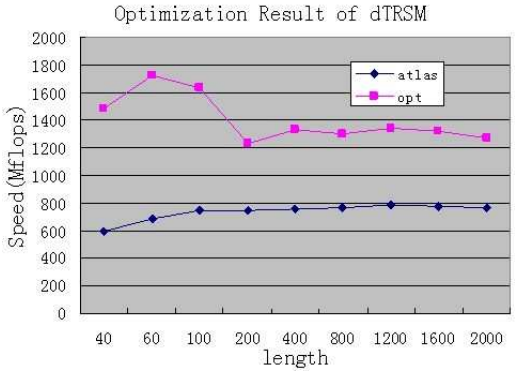

Fig. 3. Performance of optimized function dTRSM

In Fig. 2, Test1 stands for invoking dTRSM directly; Test2 stands for using loop unrolling technique; Test3 for loop unrolling and data pre-fetching; Test4 for loop unrolling, data pre-fetching, and division replced by multiplication. Test2 outperforms Test1 by almost 3 times, which is close to the theoretical value; from Test2 and Test3, it is clear that rate is doubled by instruction scheduling technique and data pre-fetching technique on the basis of loop unrolling; from Test3 and Test4, it is not hard to observe that executing time is shortened by replacing division with multiplication instruction. Thus, loop unrolling, data-prefetching, instruction scheduling, and replacing division with multiplication techniques can all accelerate the speed of execution. After Test4 optimization, the speed of dTRSM can be 1723.45Mflops when the threshold of blocking size is 60-it outperforms the algorithm 1 by a factor of 7.37 and outperforms ATLAS by a factor of 1.5.

When dealing with large-scaled data, the computation of TRSM takes up a rather low ratio. Thus, we only discuss a comprehensive result of optimization-opt in Fig.3 represents the result. The speed of dTRSM in the steady status is 1300Mflops, which outperforms the former function by 60%.

5.2 Performance of Other Optimized Functions in TRSM

In terms of double precision complex type function zTRSM, we acquire the proper optimizing techniques through testing how those techniques works under small scaled data. The function does not achieve a high performance-the speed is only 573.56Mflops when threshold of block size is 24, when loop unrolling and replacing division by multiplication techniques are employed. So we unroll the core instruction Mmls (multiply-subtract instruction for complex numbers) to respectively compute the real part and the imaginary part, and then instruction scheduling and data pre-fetching is applied. Finally, the rate has reached 1211.57Mflops, which outperforms ATLAS by 200%.

As for single float type function sTRSM, the techniques used are the same as that of dTRSM except the threshold of block size is set as 72. The optimizing methods for cTRSM are similar as that of zTRSM except the threshold of block size is set as 36. Fig. 3 presents the performance of optimized sTRSM, dTRSM and zTRSM.

In Fig. 4, s_atlas, z_atlas, c_atlas individually represent performance of sTRSM, zTRSM, cTRSM optimized by ATLAS, and s_opt, z_opt, c_opt stands for performance of sTRSM, zTRSM, cTRSM optimized by us. It is obvious in Fig. 3 that the curves titled s_opt, z_opt, c_opt are relatively smooth, and the final rate of sTRSM, zTRSM, cTRSM are respectively 1800, 1400, 1800Mflops, which outperforms that of ATLAS at least 70%.

6 Conclusion and Further Research

Every specific high performance computer has the necessity to be equipped with a specifically optimized BLAS, in order to have the hardware resources fully

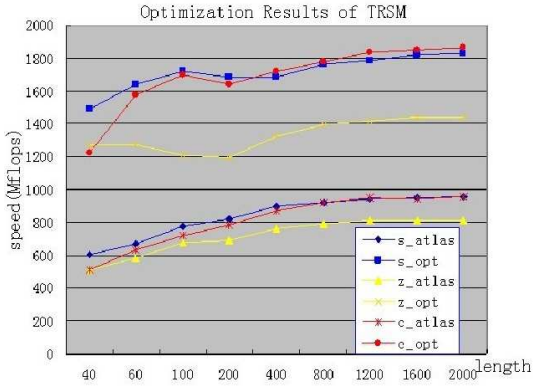

Fig. 4. Performance of optimized functions xTRSM

utilized. In this paper, we have optimized triangular matrix functions from an algorithmic level, based on the optimization of GEMM function and the Loongson 2F architecture. Here we have also employed optimizing techniques such as loop unrolling, data pre-fetching and instruction scheduling to elevate the performance of functions. The rate of double float functions dTRSM and zTRSM has reached 1300Mflops, and the rate of single float functions sTRSM and cTRSM has reached 1800Mflops. In comparison with ATLAS, our optimization has elevated by 50-60%, even 100-200% when dealing with small-scaled data.

At present, our research is concentrated in optimization of BLAS in single core environment. In the future, we will start a new program researching the optimization of BLAS in parallelism when the multi-core CPU, Loongson 3, is put into use.

Acknowledgment

We thank Bo Chen, Haitao Jiang, who provided many helpful suggestions. This paper is supported by the Key Subproject of the National High Technology Research and Development Program of China, under the grant No. 2008AA010902 and No. 2009AA01A134.

References

1. C. Lawson, R. Hanson, D. Kincaid and F. Krogh:Basic Linear Algebra Subprograms for FORTRAN usage. ACM Transaction on Mathematical Software, 5(3):308-323, 1979
2. J. G. Dumas, T. Gautier, C. Pernet. Finite Field Linear Algebra Subroutines. Proceedings of the 2002 international symposium on Symbolic on Symbolic and algebraic computation

3. E. Elmroth, F. Gustavson, I. Jonsson, B. Kagstrom. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. SIAM REVIEW, 2004
4. R. A. Chowdhury, V. Ramachandran. The Cache-oblivious Gaussian Elimination Paradigm: Theoretical Framework, Parallelization and Experimental Evaluation. Proceedings of the nineteenth annual ACM symposium on Algorithms and Computation Theory: 71-80, 2007
5. T. M. Low, A. Robert et al. API for Manipulating Matrices Stored by Blocks. Department of Computer Sciences, the University of Texas at Austin, 2004. http://www.cs.utexas.edu/users/flame/pubs/flash.ps
6. R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," Parallel Computing, 2001, 27: 3-35
7. J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes et al. Self adapting linear algebra algorithms and software. Proceedings of the IEEE, 93(2), 2005 special issue on "Program Generation, Optimization, and Adaptation"
8. K. Goto and R. van de Geijn. On reducing tlb misses in matrix multiplication. Technical Report TR02-55, Department of Computer Sciences, U. of Texas at Austin, 2002
9. R. Koenker and N. G. Pin. SparseM: A sparse matrix package for R. J. of Statistical Software, 8(6), 2003
10. N. J. Gu , K. Li et al. Optimization for BLAS on Loongson 2F architecture, Journal of University of Science and Technology of China, 2008:38(7)