Level-3 BLAS on the TI C6678 multi-core DSP

Murtaza Ali, Eric Stotzer
Texas Instruments
{*mali,estotzer*}*@ti.com*

Francisco D. Igual
Dept. Arquitectura de Computadores y Automática
Univ. Complutense de Madrid
figual@fdi.ucm.es

Robert A. van de Geijn
Department of Computer Science
University of Texas at Austin
rvdg@cs.utexas.edu

Abstract

*Digital Signal Processors (DSP) are commonly employed in embedded systems. The increase of processing needs in cellular base-stations, radio controllers and industrial/medical imaging systems, has led to the development of multi-core DSPs as well as inclusion of floating point operations while maintaining low power dissipation. The eight-core DSP from Texas Instruments, codenamed TMS320C6678, provides a peak performance of 128 GFLOPS (single precision) and an effective 32 GFLOPS (double precision) for only 10 watts. In this paper, we present the first complete implementation and report performance of the Level-3 Basic Linear Algebra Subprograms (BLAS) routines for this DSP. These routines are first optimized for single core and then parallelized over the different cores using OpenMP constructs. The results show that we can achieve about 8 single precision GFLOPS/watt and 2.2 double precision GFLOPS/watt for General Matrix-Matrix multiplication (*GEMM*). The performance of the rest of the Level-3 BLAS routines is within 90% of the corresponding* GEMM *routines.*

1 Introduction

Power efficiency has become an important factor in High Performance Computing (HPC) applications. This has led to an interest in programmable and/or reconfigurable platforms including Graphics Processing Units (GPUs) and Field Programmable Gate Array (FPGA) devices. In this paper, we present an alternative low power architecture for HPC, the TI C66x Digital Signal Processor (DSP) that is widely used in embedded applications, focusing on the implementation of the Level 3 BLAS. We discuss how, unlike GPUs or FPGAs, standard techniques and plain C/C++ suffice to attain high performance for low power.

High performance DSPs are commonly found in cellular base-stations, radio network controllers, as well as industrial and medical imaging systems. The processing complexity of the algorithms used in these embedded systems has been growing rapidly in recent years. DSPs have responded to this growth in various ways- by adding vector instructions, by using multiple cores on the same device, by adding floating point capabilities, cache based memory hierarchy, etc. These features were added while maintaining the low power requirement needed for embedded systems, and are of wide appeal for general-purpose HPC. The most powerful DSP available in the market to date is the TMS320C6678 (we will refer to this as simply C6678) from Texas Instruments[10]. This device is based on the Keystone multi-core architecture, code named C66x. This particular device contains eight C66x cores, provides a combined 128 single precision (SP) GFLOPS (billions of floating point operations per second) and 32 double precision (DP) GFLOPS and consumes 10W when the cores are running at 1 GHz. Unlike other power efficient compute platforms like GPUs and FPGAs, DSPs can be used both as an acceleration platform where part of the compute can be off-loaded to DSPs or as a standalone system where whole applications can run.

In the HPC arena, dense linear algebra plays an important role and is an important indicator of the potential performance of novel architectures. Several high performance linear algebra libraries are available that provide scalable multi-core multi-device implementations. These libraries cast much of their computations in calls to the Basic Linear Algebra Subroutines specification (BLAS). The Level-3 BLAS routines support matrix-matrix operations and are the most computationally intensive [2]. A companion paper [5] provides early experience in implementing the single precision General Matrix-Matrix multiplication (GEMM) operation on the C6678 and integration of the routine into a modern dense linear algebra library, namely the `libflame` library [11]. In this paper, we focus on the implementation and performance of the full set of Level-3 BLAS routines, a first for a DSP. Based on single core implementations, OpenMP is used to implement multithreaded versions of these routines.

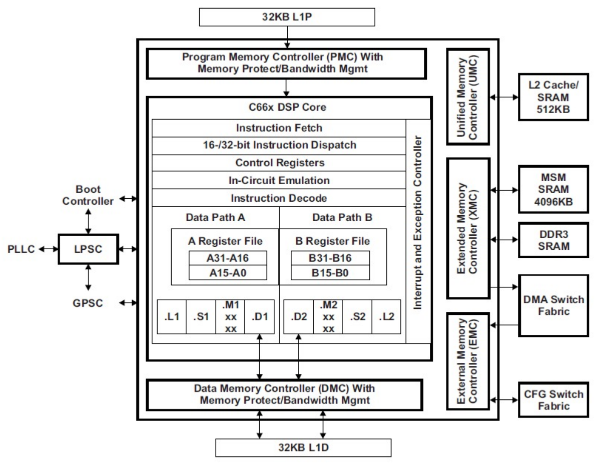


Figure 1. C66x DSP Core Block Diagram.

2 Overview of the C6678 DSP architecture

The C6678 DSP is an eight core high performance DSP with both fixed and floating point capabilities [10]. It is based on the C66x core described in more details later. The overall eight core architecture also provides a rich set of industry standard peripherals: PCIe interface to to communicate with a CPU host, Serial Rapid I/O (SRIO) running at 20 Gbps to communicate between DSP devices, or Direct memory Access (DMA) to transfer data between DDR3 memory and on-chip memory where the data transfer can be hidden behind compute operations. The cores are capable of running up to 1.25 GHz. However, for our analysis here, we have analyzed a 1 GHz device dissipating 10 W power.

2.1 Core architecture and memory hierarchy

The C66x core architecture [9] based on very Long Instruction Word (VLIW) architecture is shown in Figure 1. The architecture takes advantage of several levels of parallelism. The *instruction level parallelism* is due to the eight functional units arranged in two sides. Each side has four unique units: L, M, S and D units. The M units perform multiplication operations. The D unit handles load/store and address calculations. The other functionalities including additions/subtractions, logical, branch and bitwise operations are distributed between L and S units. This 8-way VLIW machine can issue eight parallel instructions every cycle. There are two sets of 32-bit registers for a total of 64 registers- one set connected to one side of units.

The instruction set includes Single Instruction Multiple Data (SIMD) operating on up to 128-bit vectors, thereby, allowing *data level parallelism* within the core. With two sets of L, S and M units, the core is capable of carrying out 8 SP multiply-add operations in one cycle. In double precision, it can carry out 2 multiply-add operations in one cycle. All floating point operations are IEEE754 compliant. With eight such cores running at 1 GHz in the C6678 DSP, we have 128 SP GFLOPS or 32 DP GFLOPS. The eight cores allow the system to take advantage of *thread level parallelism* by running different threads across different cores.

There are 32 KB of L1 program cache and 32 KB of L1 data cache. In addition there is also 512 KB of L2 cache. These caches are dedicated to each core. The L1 data and L2 memory can be configured as RAM or cache or part RAM/part cache. This provides additional capability of handling memory and is exploited in our BLAS implementations. There is an additional 4096 KB of shared on chip memory accessible by all cores (usually referred as MSMC memory). The external 64 bit DDR3 memory interface runs at 1600 MHz and also has ECC DRAM support.

2.2 Programming the DSP

TI's DSPs run a lightweight real time native operating system called SYS/BIOS. A C/C++ compiler is provided as part of the development environment. The compiler is C89/C++98 compliant and virtually every C89 code can be ported with no additional effort. To improve the efficiency of the generated code for each TI architecture, the compiler provides optimization techniques in the form of `pragmas` and intrinsic SIMD instructions to fully exploit the core architecture and extract the all the potential performance without resorting to assembly programming.

The compiler supports OpenMP 3.0 to allow rapid porting of existing multi-threaded codes to multi-core DSP. The compiler translates OpenMP into multi-threaded code with calls to a custom runtime library built on top of SYS/BIOS and inter-processor communication (IPC) protocols. The OpenMP runtime performs the appropriate cache control operations to maintain the consistency of the shared memory when required, but special precaution must be taken to keep data coherency for shared variables, as no hardware support for cache coherency across cores is provided.

3 Implementation of GEMM on a single core

Our GEMM implementation is structured much like the implementation in the GotoBLAS library [3, 4]. The GotoBLAS is one of the most widely used and highly performing BLAS implementation available today. Here, we briefly introduce the key aspects of our implementation. A more detailed description of the mapping of the GotoBLAS approach to C6678 can be found at [5]. The GotoBLAS approach decomposes the GEMM into different layers, each mapped to a different level in the memory hierarchy. The decomposition also amortizes the data movement between memory levels with computations, thereby, allowing near-optimal performance. As the general strategies adopted GEMM are the base for the rest of the BLAS-3 implementations, we review the main concepts underlying our GEMM implementation.

3.1 Mapping into memory hierarchy

The basic GEMM performs the following operation,

$$C := \alpha AB + \beta C$$

where A, B, and C are $m \times k$, $k \times n$, and $m \times n$ dense matrices, respectively. These matrices are all assumed to be resident in external DDR3 memory. The initial division of the matrices into sub-blocks is illustrated in Figure 2. First A and B are divided in column panels (of dimension $m \times k_{partition}$) and row panels (of dimension $k_{partition} \times n$) respectively. The matrix C is updated using low rank panel-panel multiplications (GEPP). The panels of A are further subdivided into blocks (of dimension $m_{partition} \times k_{partition}$) and the corresponding row panel of C is updated using block-panel multiplications (GEBP).

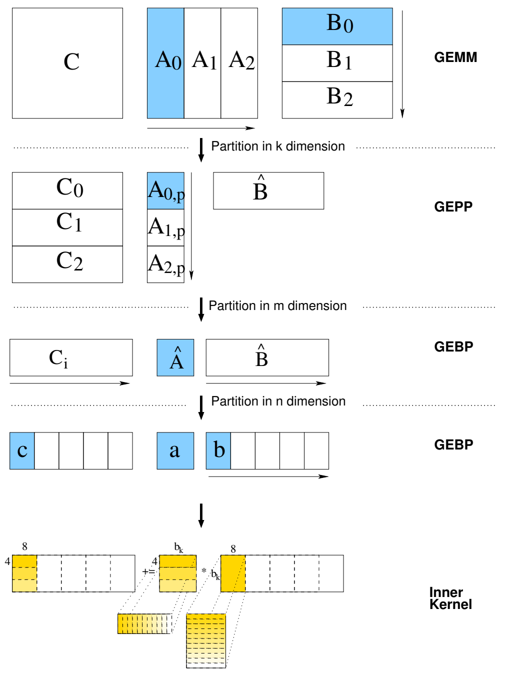


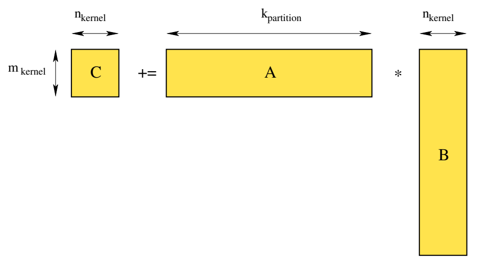
Figure 2. Blocking of matrices to fit into internal memory for single precision GEMM. $\hat{A}$ and $\hat{B}$ are the packed versions of the corresponding blocks of A and B.

One such block is shown as matrix a in Figure 2. The size of this sub-matrix is chosen so that it can fit into L2 RAM. We have used half of the L2 RAM available (256 KB) to store this sub-block of the A matrix. The panel of B matrix which could be packed beforehand is also divided into small blocks at this time. The size of the sub-panel b is chosen to fit into L1 RAM. In our implementation, we have chosen half of L1 (16 KB) as cache and the other half as L1 RAM. The latter memory is allocated to hold the block b.

3.2 Mapping into core architecture

The block a that has been brought into L2 RAM (with some necessary packing to be described later) is further subdivided into smaller sub-blocks of size $m_{kernel} \times k_{partition}$. The block b, which now resides in L1 RAM (again after necessary packing) is of size $k_{partition} \times n_{kernel}$. An optimized kernel was developed to perform matrix-matrix multiplication update of a small block c of size $m_{kernel} \times n_{kernel}$ by multiplying the sub-block of A of size $m_{kernel} \times k_{partition}$ residing in L2 RAM with the sub-block of B of size $k_{partition} \times n_{kernel}$ residing in L1 RAM. The choices of m_{kernel} and n_{kernel} depend on the number of registers available in the core. Figure 3 details these values for different data types used in our implementation.

The actual kernel implementation is done through a rank-1 update of sub-block of C being computed by multiplying one column (of size $m_{kernel} \times 1$) of the sub-block of A with one row (of size $1 \times n_{kernel}$) of the sub-block of B at a time looping over $k_{partition}$ times. This inner loop of the kernel is optimized using intrinsic vector instructions to take the best advantage of the C66x core architecture.



Data type	m_{kernel}	n_{kernel}
single precision (s)	4	8
double precision (d)	2	4
single precision complex (c)	4	4
double precision complex (z)	1	1

Figure 3. Inner kernel partitions for the C66x architecture.

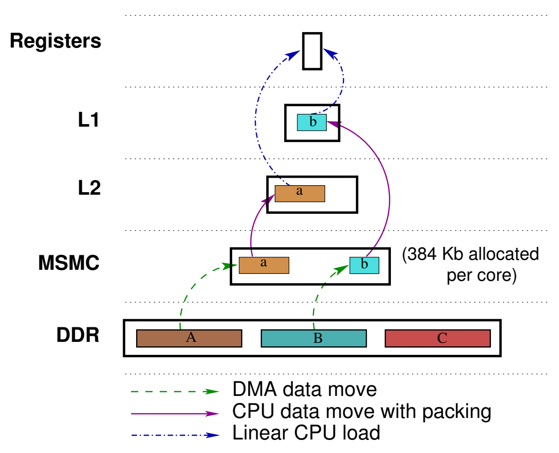


Figure 4. Data movement strategy for GEMM.

3.3 Data movement and packing

We have developed a scheme to overlap data movement with computations as much as possible, leveraging the DMA capabilities of the DSP. Blocks of A and B are first brought into on-chip shared memory from the external DDR3 memory as illustrated in Figure 4. The shared memory is partitioned into 8 equal blocks of size 384 KB to be used by each core. This way, each core can independently manage its data movement through the shared RAM. This data movement is the slowest and hence we employed DMA to perform it. Note that DMA operates independently of the CPU core and hence the CPU can continue performing operations on already loaded data.

Once the per core assigned data is loaded in the shared RAM and the CPU is exhausted operating on all the data available in its L2 and L1 RAM, the CPU will move the block of A brought into shared RAM to L2 RAM and the block of B to L1 RAM with necessary packing. The packing required of A is illustrated in Figure 5. It basically extracts sub-blocks of size $m_{kernel} \times k_{partition}$ from the block of A of size $m_{partition} \times k_{partition}$ already loaded into shared RAM and put these sub-blocks in linear memory in L2 RAM. This allows the kernel to traverse the L2 memory linearly for best cache performance.

4 Extension to other Level-3 BLAS routines

It has long been observed that the other Level-3 BLAS can be implemented in terms of GEMM [6]. We instead follow the approach in [3] to extend the strategy proposed for GEMM to the rest of Level-3 BLAS functions. In this approach, most of the customization for performing various Level-3 BLAS functions is kept in the data moving and packing part of the operations. The GEMM kernel is modified only slightly to accommodate the requisite variations as described below. Thus, only the main differences with the approach taken for GEMM are illustrated in this section for each one of the implemented routines.

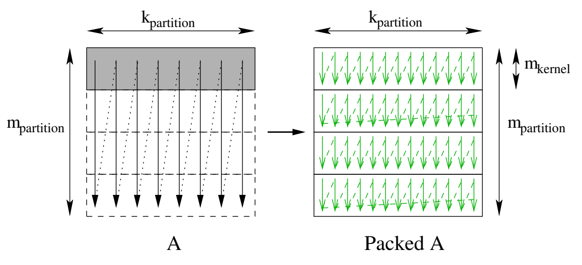


Figure 5. Packing of A for use by the inner kernel.

Table 1 summarizes the Level-3 BLAS routines operating in real arithmetic and the flop count that will be used for the experimental experiments in Section 6.

4.1 SYMM and HEMM

In SYMM operations, only the upper or lower part of the matrix is referenced. Thus, there can be three cases when the $m_{partition} \times k_{partition}$ block, a, of matrix A is brought into on-chip shared memory (the following assumes that the lower part of the matrix is to be referenced).

Case 1: Block a is to the left of the diagonal. The data movements are exactly the same as GEMM.

Case 2: Block a is to the right of the diagonal. The valid data for this case is in transposed location to the left of the diagonal. Data movement is the same as GEMM for transpose condition.

Case 3: The diagonal cuts through block a. In this case part of the block is valid and part of it needs to be read from the corresponding transpose locations. In this two blocks of data will be brought in. During data packing from shared memory to L2 memory, the region above the diagonal will be filled in.

Other then data movement and packing operations, there is no specific change necessary in the GEMM kernel. The kernel is used as is. Also note that any conjugate transpose needed for the Hermitian function (HEMM) can be easily performed during these data packing operations.

4.2 SYRK/HERK and SYR2K/HER2K

The operations of SYRK are very similar to GEMM where the first matrix is A and the second matrix is A^T. The only

Routine	Operation	Comments	Flops
xGEMM	$C := \alpha\, op(A)\, op(B) + \beta C$	$op(X) = X, X^T, X^H$, C is $m \times n$	$2mnk$
xSYMM	$C := \alpha AB + \beta C$ $C := \alpha BA + \beta C$	C is $m \times n$, $A = A^T$	$2m^2n$ $2mn^2$
xSYRK	$C := \alpha AA^T + \beta C$ $C := \alpha A^TA + \beta C$	$C = C^T$ is $n \times n$	n^2k
xSYR2K	$C := \alpha AB^T + \alpha BA^T + \beta C$ $C := \alpha A^TB + \alpha B^TA + \beta C$	$C = C^T$ is $n \times n$	$2n^2k$
xTRMM	$C := \alpha\, op(A)C$ $C := \alpha\, Cop(A)$	$op(A) = A, A^T, A^H$, C is $m \times n$	nm^2 mn^2
xTRSM	$C := \alpha\, op(A^{-1})C$ $C := \alpha\, Cop(A^{-1})$	$op(A) = A, A^T, A^H$, C is $m \times n$	nm^2 mn^2

Table 1. Functionality and number of floating point operations of the studied BLAS routines.

difference is that only the upper or the lower part of the matrix C is computed. This is achieved by simple modification of the GEBP loop by controlling the loop indices which will only compute the part below or above the diagonal element.

The GEMM kernel computes a $m_{kernel} \times n_{kernel}$ sub-block of C. A slight modification to the GEMM kernel is done to allow writing only above or below any diagonal line of this sub-block. This ensures that we do not overwrite any portion of matrix C while computing sub-blocks on the diagonals. SYR2K is simply computed by carrying out SYRK-like functions twice- first with A and B^T and then with B and A^T. An equivalent approach has been adopted for Hermitian matrices (routines HERK and HER2K).

4.3 TRMM

In TRMM, the matrix A is upper or lower triangular. The operations are performed in-place. Hence, the values are to be computed for top down rows for upper triangular case and bottom up for lower triangular case for case of left multiplication. The GEBP is modified so that loop indices only vary over non-zero data blocks that the kernel operates on. In this case also, we can have three cases when the $m_{partition} \times k_{partition}$ block, a, of matrix A is brought into on-chip shared memory (the following assumes that the lower part of the matrix is to be referenced).

Case 1: Block a is to the left of the diagonal. The data movements are exactly the same as GEMM.

Case 2: Block a is to the right of the diagonal. This case should not occur since the GEBP loop indices will ensure this case is skipped.

Case 3: The diagonal cuts through block a. In this case we still bring in the 2-D sub-block from memory. However, during data packing from shared memory to L2 memory, the region above the diagonal will be filled with zero for each $m_{kernel} \times m_{kernel}$ sub-blocks on the diagonal.

4.4 TRSM

In order to explain the TRSM, we refer to Figure 6. This illustration assumes that the matrix A is lower triangular. The operations are in-place, i.e., the elements of B will be replaced by corresponding elements of $A^{-1}B$ as the computation progresses. We focus on the update of the small block b_3 of size $m_{kernel} \times n_{kernel}$. In our blocking approach to fit matrix blocks in internal memory, we load blocks of A of size $m_{partition} \times k_{partition}$ to shared RAM. These blocks are then moved to L2 RAM with necessary data packing. In this particular illustration, two such blocks of A (namely A_1 and A_2) are necessary to update b_3. The parts of A within these blocks that are needed for updating b_3 are a_1, a_2, and a_3. a_3 is the $m_{kernel} \times m_{kernel}$ block that is at the diagonal of the original matrix A and is also lower triangular. The corresponding sub-blocks in the matrix B are b_1 and b_2. Note that these sub-blocks will already have the corresponding values $A^{-1}B$ (referred to as b'_1 and b'_2) before they are used to update b_3. The updated values of b_3 can now be computed as

$$b'_3 = a_3^{-1}(b_3 - a_2 b_2 - a_1 b'_1).$$

In this illustration, b_3 will be updated in two iterations. First, when the block A_1 is loaded to L2 RAM, the update is a simple GEMM operation

$$b_3 = -a_1 b'_1.$$

This case thus uses the data packing and internal kernel as typical GEMM. The second update occurs when the second block A_2 is loaded to L2 RAM. In this case, we first need GEMM operations

$$b_3 = b_3 - a_2 b'_2,$$

followed by a matrix multiplication

$$b'_3 = a_3^{-1} b_3.$$

A GEMM kernel is modified to create an ad-hoc TRSM kernel which performs the GEMM update followed by the

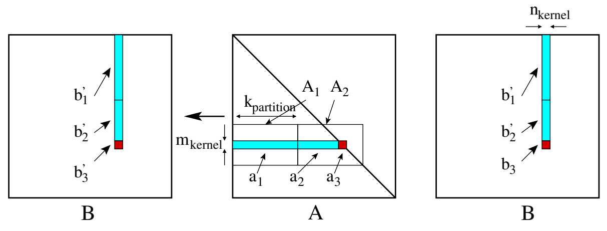


Figure 6. Updating TRSM using GEMM.

matrix multiplication. This kernel assumes that the inverse of the lower triangular block a_3 is already available in the block A_2. When this block is moved from shared RAM to L2 RAM, the portion of data corresponding to the sub-block a_2 will be packed in a similar fashion to that used for GEMM. In addition, the $m_{kernel} \times m_{kernel}$ blocks on the diagonal will be inverted. Note that since this block is lower triangular, the inverse is also lower triangular.

5 Multi-threading using OpenMP

The single core optimized routines described in previous sections are parallelized across multiple cores using an OpenMP-based multi-threading scheme. Each thread is assigned to one core of the DSP. The output matrix is divided into non-overlapping regions as follows:

- For TRMM and TRSM, the output matrix is divided into non-overlapping vertical strips. In this scenario, each thread will use the whole matrix A but will only need to use the corresponding vertical strip of B assigned to the particular thread. This division allows each thread to produce output independent of others.

- For all other routines, the output matrix is divided into non-overlapping horizontal strips. In this scenario, each core will use the whole matrix B and will only use the corresponding horizontal strip of A assigned to it. This is the preferred division over vertical strips as is done with TRMM and TRSM, since the performance improves with larger values of k assigned to each core.

In all the above cases, the horizontal/vertical strips are chosen so that the computational load in each core is approximately the same. We note the constraints that need to be maintained while blocking the output matrices to each thread. The number of rows assigned to an output block being operated on by a thread needs to be multiple of the size of the cache line (128 bytes for this architecture). This is to ensure that multiple cores operate on independent cache lines as there is no hardware cache coherency mechanism. Then number of columns assigned to an output block needs to be multiple of n_{kernel}. There is no restriction on k.

6 Experimental results

6.1 Hardware setup

Our experimental evaluation is carried out using a TMDXEVM6678LE evaluation module (EVM) that includes an on-board C6678 processor running at 1 GHz. The board has 512 MB of DDR3 RAM memory. Our tests have been developed on top of SYS/BIOS. Experimental results are reported in terms of GFLOPS for increasing problem sizes, using square matrices. For a detailed flop count of each BLAS routine, see Table 1.

The evaluation of a new full BLAS implementation is a thorough task. In this section, we report a selected subset of results that are representative of the rest of the library performance and behavior on both single and multiple cores.

6.2 Single core performance

Figure 7 reports the performance attained by our optimized routines for single and double precision operating in real (top plot) and complex (bottom plot) arithmetic.

The results show that our implementation achieves a peak performance of 10.3 GFLOPS for single precision GEMM and 2.8 GFLOPS for double precision, and 10.9 and 2.4 GFLOPS for complex GEMM. The rest of the routines are within 90% of the corresponding GEMM operations. Our GEMM performance is about 64% of peak for single precision and about 70% of peak for double precision. The performance loss is mainly dictated by the caching mechanism in the DSP, not as aggressive as that of general-purpose architectures like Intel x86. We have identified several factors related to this in the implementation of GEMM:

- Cache misses occur at every kernel call when a new set of sub-block of matrix A is needed.

- Loop overhead and loading and saving of entries of C in the inner kernel.

- Overhead in data movement across different layers (DMA and `memcpy`) and data packing.

The 2D DMA mechanism dictates some restrictions regarding the maximum stride allowed for sub-blocks of matrices. For large matrices this limit is reached, and it is necessary to switch to `memcpy`-like data copies. This effect is evident for complex matrices, in which the stride doubles that of real matrices, but we have observed similar overhead for bigger real matrices than those shown in the figures. Alternate approaches using the flexibility that DMA engine provides are currently being investigated to remove this restriction larger matrices and attain a homogeneous behavior.

There are two main reasons for the difference of performances between GEMM and the rest of routines:

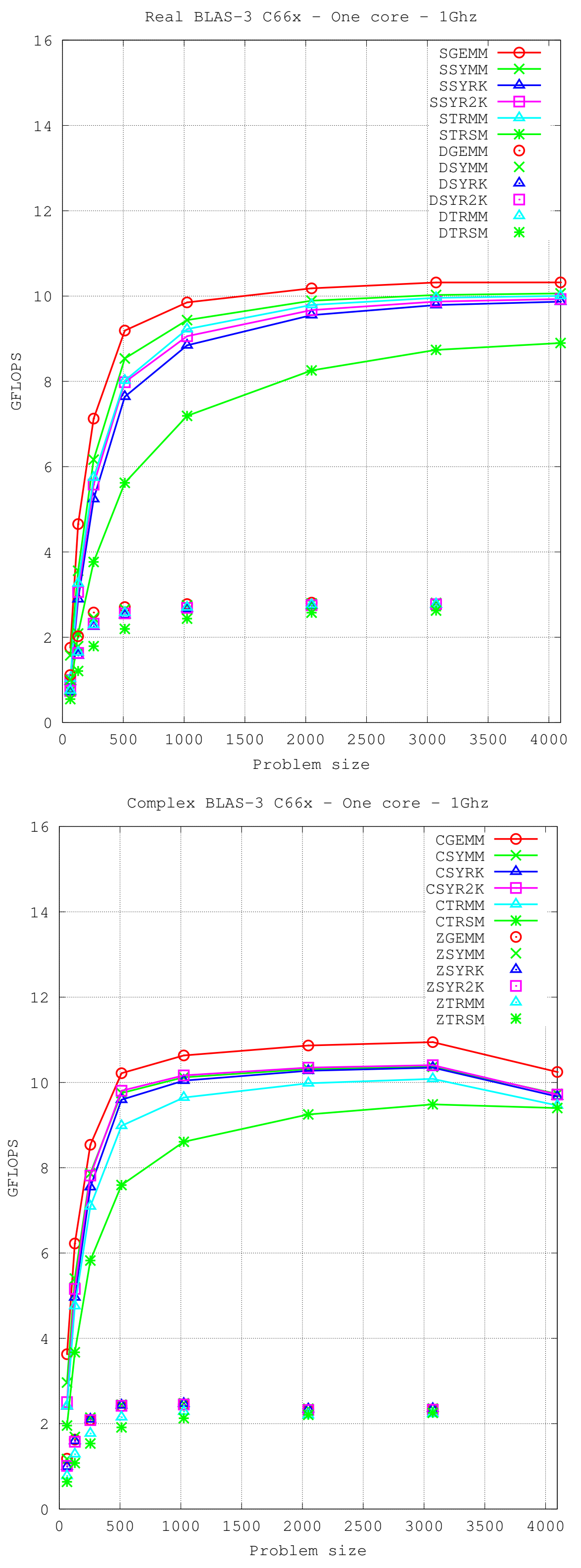


Figure 7. Performance of the single-core BLAS-3 implementation. Top: real arithmetic. Bottom: complex arithmetic.

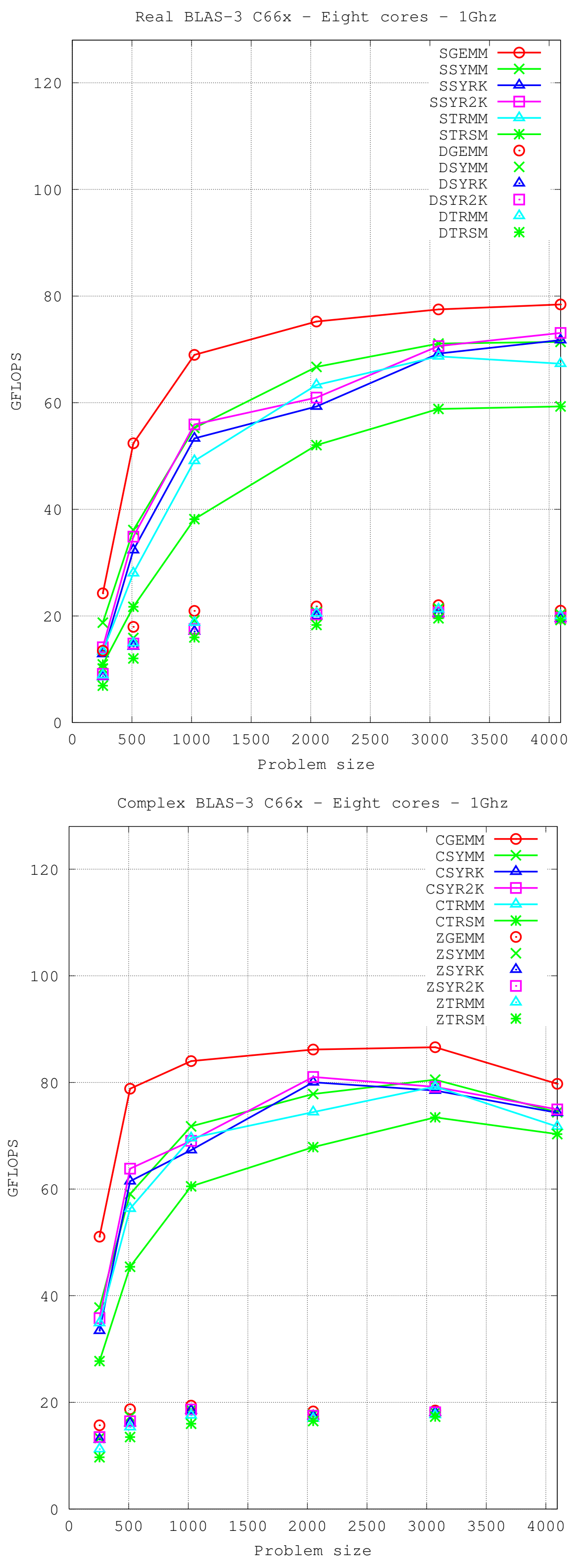


Figure 8. Performance of the multi-core BLAS-3 implementation. Top: real arithmetic. Bottom: complex arithmetic.

- The additional data re-arrangements needed in these routines. This is especially pronounced for TRSM, where the inversion requires division operations. TRSM also requires maintaining data coherency among various layers of the memory hierarchy.
- The processing is most efficient when the internal kernel is looping over largest possible values for k, i.e., $k_{partition}$. For triangular matrices, the actual number of loop iterations will be less than $k_{partition}$ (in some cases much less) for diagonal blocks.

6.3 Multi-core performance

Figure 8 reports the performance of the implementations running on the 8 cores of the DSP. The parallel BLAS implementation on 8 cores achieves a peak performance of 79.4 GFLOPS for single precision GEMM and 21 GFLOPS for double precision real GEMM, and 86.6 and 19.4 GFLOPS for complex GEMM. As for the single-core version, the rest of the routines are within 90% of the corresponding GEMM operations. The behavior for big matrices is directly inherited from the sequential implementation. Regarding the attained speedup of the parallel version, comparing the single precision implementation on 8 cores with the sequential version, the attained speedup values vary between $7.5\times$ for GEMM to $6.6\times$ for TRSM.

Considering 10 W as the power dissipated by the chip, our implementation attains a peak GFLOP/Watt ratio of roughly 8 GLOPS/Watt for single precision GEMM and 2.1 GFLOPS/Watt for double precision. As of today, and specially for single precision, this is one of the most efficient BLAS implementation. Moreover, DSP can be seen as standalone processors, without the need of an attached host. This increases the appeal of the architecture for highly efficiency-demanding environments and applications.

7 Conclusion and future work

We have presented results for a complete, highly-tuned Level-3 BLAS for a multi-core DSP. This implementation achieves 8 single precision GFLOPS/W and 2.2 double precision GFLOPS/W for GEMM operations. The rest of the BLAS 3 routines achieve within 90% of this GEMM performance. These results appear to be among the best achieved performance per watt numbers for current HPC platforms. This implementation is the first Level-3 BLAS implementation targeting a DSP, and will form the basis for efficient implementations of LAPACK [1] and `libflame`.

So far we have only presented results for performance on a single DSP multicore processor. Future developments will target cards with many DSP chips, which are expected to have four to eight such devices, and multiple cards attached to a host following a multi-accelerator paradigm. We are currently working on supporting the Message-Passing Interface (MPI) [8] on the C6678 device to allow standard communication across devices and to map distributed memory dense linear algebra libraries, such as Elemental [7], to it. At the DSP level, the necessity of a tuned BLAS library like the one presented in this paper that exploits efficiently the potential of the chip is fundamental to such efforts.

References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

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

[3] K. Goto and R. van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.*, 35(1):1–14, 2008.

[4] K. Goto and R. A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3), 2008.

[5] F. D. Igual, M. Ali, A. Friedmann, E. Stotzer, T. Wentz, and R. van de Geijn. Unleashing DSPs for general-purpose HPC. FLAME Working Note #61. Technical Report TR-12-02, The University of Texas at Austin, Department of Computer Sciences, February 2012.

[6] B. Kågström, P. Ling, and C. V. Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft.*, 24(3):268–302, 1998.

[7] J. Poulson, B. Marker, and R. van de Geijn. Elemental: A new framework for distributed memory dense matrix computations. FLAME Working Note #44. Technical Report TR-10-20, The University of Texas at Austin, Department of Computer Sciences, June 2010.

[8] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[9] TMS320C66x DSP CPU and Instruction Set Reference Guide. `http://www.ti.com/lit/ug/sprugh7/sprugh7.pdf`, November 2010. Texas Instruments Literature Number: SPRUGH7.

[10] TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor. `http://www.ti.com/lit/ds/sprs691c/sprs691c.pdf`, February 2012. Texas Instruments Literature Number: SPRS691C.

[11] F. G. Van Zee, E. Chan, R. A. van de Geijn, E. S. Quintana-Ortí, and G. Quintana-Ortí. The libflame library for dense matrix computations. *Computing in Science and Engineering*, 11:56–63, 2009.