Algorithm of Automatic Parallelization of Generalized Matrix Multiplication[⋆]

Elena N. Akimova[1,2], Roman A. Gareev[1]

[1] Ural Federal University, Ekaterinburg, Russia
[2] Krasovskii Institute of Mathematics and Mechanics, Ural Branch of RAS, Ekaterinburg, Russia
`aen15@yandex.ru, gareev.roman@urfu.ru`

Abstract. Parallelization of generalized matrix-matrix multiplication is crucial for achieving high performance required in many situations. Parallelization performed using contemporary compilers is not sufficient enough to replace expert-tuned multi-threaded implementations or to get close to their performance. All competitive solutions require previously optimized external implementations that cannot be available for a given type of data and hardware architecture. In the paper, we introduce an automatic compiler transformation that does not require an external code or automatic tuning to attain more than 85% of performance of an optimized BLAS library. Our optimization shows competitive performance across various hardware architectures and in the case of different forms of generalized matrix-matrix multiplication. We believe that availability of multi-threaded implementations of generalized matrix-matrix multiplication can save time when any optimized libraries are not available.

Keywords: high-performance, multicore, compilers, mathematical software, computations on matrices, linear algebra algorithms

1 Introduction

Let A, B, and C be appropriately sized matrices containing elements in a set S; and let S and operations $\oplus$ and $\otimes$ be contained in a closed semiring [1]. Then the formula $C \leftarrow \alpha \otimes C \oplus \beta \otimes A \otimes B$, where $\oplus$ and $\otimes$ operations from the corresponding matrix semiring, and α and β are constants that are not equal to zero, is a generalized matrix multiplication or matrix multiply-and-add operation (MMA) [2].

MMA is important in many situations. For instance, it can be used to solve algebraic path problems (APPs) such as finding shortest connecting paths, finding the least and the most reliable paths, finding paths with maximum capacity, or finding paths with maximum cost. Examples of their usage are finding directions between physical locations, such as driving directions on web mapping websites

[⋆] This work was partly performed at the Center of Excellence "Geoinformation technologies and geophysical data complex interpretation" of the Ural Federal University.

like MapQuest or Google Maps [3], applications studied in operations research, robotics, plant and facility layout, transportation, and VLSI design [4]. There are various applications of general matrix-matrix multiplication MMA with $\oplus = +$ and $\otimes = \times$. General matrix-matrix multiplication can be used for encoding the database before it goes for data mining in a centralized environment [5]. It can be also used during the computation in convolution layers of the Convolutional Neural Networks [6] applied in machine learning. Another example of its application is solving the inverse structural gravity problem of mathematical physics using the Levenberg-Marquardt algorithm [7]. Matrix-matrix multiplication can be applied in the calculation of the RI-MP2 correlation energy of a molecule studied by quantum chemistry [8]. General matrix-matrix multiplication is typically a building block for other matrix-matrix operations studied in high performance computing [9].

Contemporary compilers and compiler front ends (e.g., GCC [10], Clang [11], ICC [12], IBM XL [13]) cannot automatically transform a textbook style implementation of general matrix-matrix multiplication into a code that comes close to the performance of expert-tuned multi-threaded implementations of MMA. Specialized libraries (e.g., Intel's MKL [14], BLIS [15], OpenBLAS [16]) provide high-performance implementations of general matrix-matrix multiplication, but nevertheless these approaches require previously optimized external code and can only be used if an optimized implementation is available [17].

The work presents a new compiler optimization for MMA based on the Polly loop optimizer [18]. It is aimed to close the performance gap between compilers and expert-tuned multi-threaded libraries.

Our contributions are

- an automatic transformation for parallelizing MMA based on the computational structure proposed by the BLIS framework (Section 4.3);
- comparison of our approach to existing production compilers and vendor optimized BLAS libraries: we attain more than 85% of performance of the multi-threaded instances of general matrix-matrix multiplication that are available in optimized BLAS libraries (Section 5).

2 Background

In this section, we briefly describe the polyhedral model, i.e. a mathematical framework for loop nest optimizations, which is used to focus on modeling and optimization of the memory access behavior of a program [19,20].

For Static Control Parts (SCoPs) [21,22], i.e. program regions that are "sufficiently regular" to be modeled, each compute statement is described by three components: iteration domain, scheduling function, and access relation. An iteration domain is represented as a $\mathbb{Z}$-Polytope [23] describing the dynamic instances of the SCoP statement. A scheduling function is represented by a $\mathbb{Z}$-Polytope that relates dynamic statement instances to their execution time vectors defining the execution order. An access relation is described using a $\mathbb{Z}$-Polytope defining the relation between iteration vector and the accessed array subscript.

3 Automatic Detection of Kernel

In the paper, we consider optimization of MMA-like kernels introduced in Polly [18]. They have the following definition:

Definition 1. *A generalized matrix multiplication-like kernel (MMA-like kernel) is a perfectly nested loop nest such that*

- it satisfies the requirements of the polyhedral model;
- without loss of generality, it contains three one-dimensional for loops Loop 1, Loop 2, and Loop 3 with induction variables i, j, and p, respectively, that are incremented by one;
- the innermost loop body is representable as a statement of the form $C[i][j] = E(A[i][p], B[p][j], C[i][j])$, where $A[i][p]$, $B[p][j]$, $C[i][j]$ are accesses to matrices A, B, C, respectively, and E is an expression that contains reads from the matrices A, B, C and an arbitrary number of reads from constants with respect to Loop 1, Loop 2, and Loop 3.

To detect and subsequently optimize the SCoP statements that implement MMA-like kernels, we use the algorithm that is based on the analysis of memory accesses and memory dependencies implemented in Polly. Its description is beyond the scope of our paper. MMA is a particular case of MMA-like kernel. We consider how it affects the optimization in Section 4.3.

4 Optimizing MMA-like kernel

In this section, we present an algorithm for obtaining the code structured similar to an expert-optimized multi-threaded general matrix-matrix multiplication. Firstly, we consider the expert-designed multithreaded general matrix-matrix multiplication and then discuss our optimization.

4.1 The expert implementation of general matrix-matrix multiplication

An example of an expert implementation of the general matrix-matrix multiplication algorithm (i.e. an implementation that was tuned by dense linear algebra experts) can be found in BLIS [15]. It consists of two packing routines and five loops around the micro-kernel. Packing routings perform copying the data of matrices A and B (that are operands of the MMA-like kernel) to created arrays A_c and B_c, respectively, to ensure that their elements are aligned to cache lines boundaries, preloaded in certain cache levels, and read in-stride access [24]. The micro-kernel is a loop around an outer product that can be implemented in assembly. The micro-kernel and two surrounding loops form the macro-kernel. Pseudo-code of the described implementation can be found in Listing 1.

Determination of M_c, N_c, K_c, M_r, and N_r parameters of the described implementation can be done manually or using an analytical model [24]. We use an analytical model for determination of parameters of single-threaded MMA-like kernels presented in Polly [18]. Determination of optimal values for multi-threaded implementation is only partially considered in [25] and can be direction of the future work.

```
$Loop_1$:  for ($j$ = 0; $j$ < N; $j$ += $N_c$)
$Loop_2$:     for ($p$ = 0; $p$ < K; $p$ += $K_c$) {
                 // Pack into $B_c$
$Copy_0$:        B($p$:$p$ + $K_c$ - 1, $j$:$j$ + $N_c$ - 1) $\to$ $B_c$
$Loop_3$:        for ($i$ = 0; $i$ < M; $i$ =+ $M_c$) {
                    // Pack into $A_c$
$Copy_1$:           A($i$:$i$ + $M_c$ - 1, $p$:$p$ + $K_c$ - 1) $\to$ $A_c$
                    // Macro-kernel
$Loop_4$:           for ($j_c$ = 0; $j_c$ < $N_c$; $j_c$ += $N_r$)
$Loop_5$:              for ($i_c$ = 0; $i_c$ < $M_c$; $i_c$ += $M_r$) {
                          // Micro-kernel
$Loop_6$:                 for ($p_c$ = 0; $p_c$ < $K_c$; $p_c$++)
$S$:                         $C_c$($i_c$:$i_c$ + $M_r$ - 1, $j_c$:$j_c$ + $N_r$ - 1)
                                  += $A_c$($i_c$:$i_c$ + $M_r$ - 1, $p_c$)
                                     * $B_c$($p_c$, $j_c$:$j_c$ + $N_r$ - 1)
                       }
                 }
              }
```

Listing 1: The implementation of general matrix-matrix multiplication of BLIS

4.2 Expert implementation of multi-threaded general matrix-matrix multiplication

The implementation of general matrix-matrix multiplication in BLIS consists of five loops that can be parallelized. To find the loop that should be parallelized to obtain the best performance gain, the parameters of the target architecture (e.g., the availability of the L3 cache, the ability to share L2 and L3 caches between threads, the support of the cache coherency) along with values of parameters of the implementation need to be considered [25](i.e., M, M_c, M_r, N, N_c, N_r). In the paper, we discuss the automatic parallelization of the second loop around the micro-kernel (i.e., indexed by j_c). In case the ratio of N_c to N_r is large, which is usually the case, it gives good opportunities for parallelism [25] helping to amortize the cost of transferring of the block of A_c from the main memory into the L2 cache and to reduce the execution time of general matrix-matrix multiplication.

4.3 An automatic parallelization of MMA-like kernel

In this subsection, we consider the algorithm for automatic parallelization implemented in Polly along with its modifications helping to apply it in the case of MMA-like kernels and, in particular, general matrix-matrix multiplication.

Polly can detect the outermost loop of the loop nest that can be executed in parallel to generate the OpenMP [26] code and to take advantage of shared memory parallelism in a SCoP [18]. To do it, Polly uses the dependence analysis [18] to discover data dependencies among program statements. A data dependency is a situation, in which a program statement refers to the data of a preceding statement [27]. In the case of Polly, data dependencies between program statements of SCoP statements are considered.

For example, let us consider the matrix-matrix multiplication of the form $C \mathrel{+}= A \times B$, where the sizes of the matrices A, B, and C are $M \times K$, $K \times N$, and $M \times N$, respectively. Listing 2 contains an example of a program that implements the described matrix-matrix multiplication. In this case, we have only one true dependency, only one anti dependency, and only one output dependency. The dependencies have the form $S(i, j, p) \rightarrow S(i, j, p + 1)$.

```
      for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
          for (p = 0; p < K; p++)
S:          C[i][j] += A[i][p] * B[p][j];
```

Listing 2: Example of matrix-matrix multiplication

We apply Polly to get the code, which is a generalization of the expert implementation of the general matrix-matrix multiplication in terms of the outer product defined in the semiring representing the multiplication. Subsequently, we add two output dependencies and two flow dependencies introduced by the packing routines (Section 4.1). The output dependencies have the following form: $Copy_0(j, p, i, j_c, i_c, p_c) \rightarrow Copy_0(j+N_c, p+K_c, i, j_c, i_c, p_c)$, $Copy_1(j, p, i, j_c, i_c, p_c) \rightarrow Copy_1(j, p + K_c, i + M_c, j_c, i_c, p_c)$. The flow dependencies have the following form: $Copy_0(j, p, i, j_c, i_c, p_c) \rightarrow S(j, p, i, j_c, i_c, p_c)$, $Copy_1(j, p, i, j_c, i_c, p_c) \rightarrow S(j, p, i, j_c, i_c, p_c)$. Since j_c becomes the outermost parallel loop, it is automatically parallelized by Polly.

5 Experimental Results

We compare performance of the code generated by the presented optimization to the multi-threaded instances of MMA. The experimental setup is presented in (Table 1, and Table 2). Throughput and latency of processor floating-point arithmetic units are denoted by N_{VFMA} and L_{VFMA}, respectively. Sizes of the first two cache levels of all CPUs under evaluation are 32 Kbytes and 256 Kbytes, respectively. Their associativity degrees are equal to 8. Results of measurements considered in the section are the corresponding arithmetic means collected until 95% confidence intervals to be within 10% of reported means.

Table 1. Experimental setup

Nickname	CPU	Clock speed (GHz)	Memory (GB)	L_{VFMA}[1]	N_{VFMA}[1]	Cores per socket	Sockets
Intel Sandy Bridge	Intel Core i7-3820 Sandy Bridge	3.6	16	8	1	4	1
IBM Power 7	POWER7	3.55	64	12	2	16	1
ARM	APM883208-X1	2.4	32	18	1	1	8

Table 2. The software version and additional options

Nickname	Version	Additional options
polly (original)	6.0.0	-O3 -march=native -mllvm -polly -mllvm -polly-parallel -lgomp
polly (opt)	6.0.0	-O3 -march=native -mllvm -polly -mllvm -polly-target-throughput-vector-fma=N_{VMMA} -mllvm -polly-target-latency-vector-fma=L_{VMMA} -mllvm -polly-target-1st-cache-level-associativity=W_{L_1} -mllvm -polly-target-2nd-cache-level-associativity=W_{L_2} -mllvm -polly-target-1st-cache-level-size=S_{L_1} -mllvm -polly-target-2nd-cache-level-size=S_{L_2} -ffp-contract=fast -mllvm -polly-parallel -lgomp
gcc	4.9.2	-O3 -march=native -ftree-parallelize-loops=n
icc	16.0.2	-O3 -march=native -parallel
IBM XLC	13.1.2.0	-O3 -qarch=auto -qtune=auto -qsmp=auto
Intel MKL	11.3.2	
BLIS	0.2.2	
OpenBLAS	0.2.20	

We consider the matrix-matrix multiplication of the following form $C \leftarrow \alpha \otimes C \oplus \beta \otimes A \otimes B$ implemented in Polybench 3.2 benchmark suite [28] along

[1] Values of these parameters that are not publicly available from the vendors' instruction set/optimization manuals were determined empirically and can be different from the real values.

with the maximum reliability path problem. We evaluate the matrix-matrix for four data sets defined in Polybench 3.2 benchmark suite: small, standard, large, and extra large.

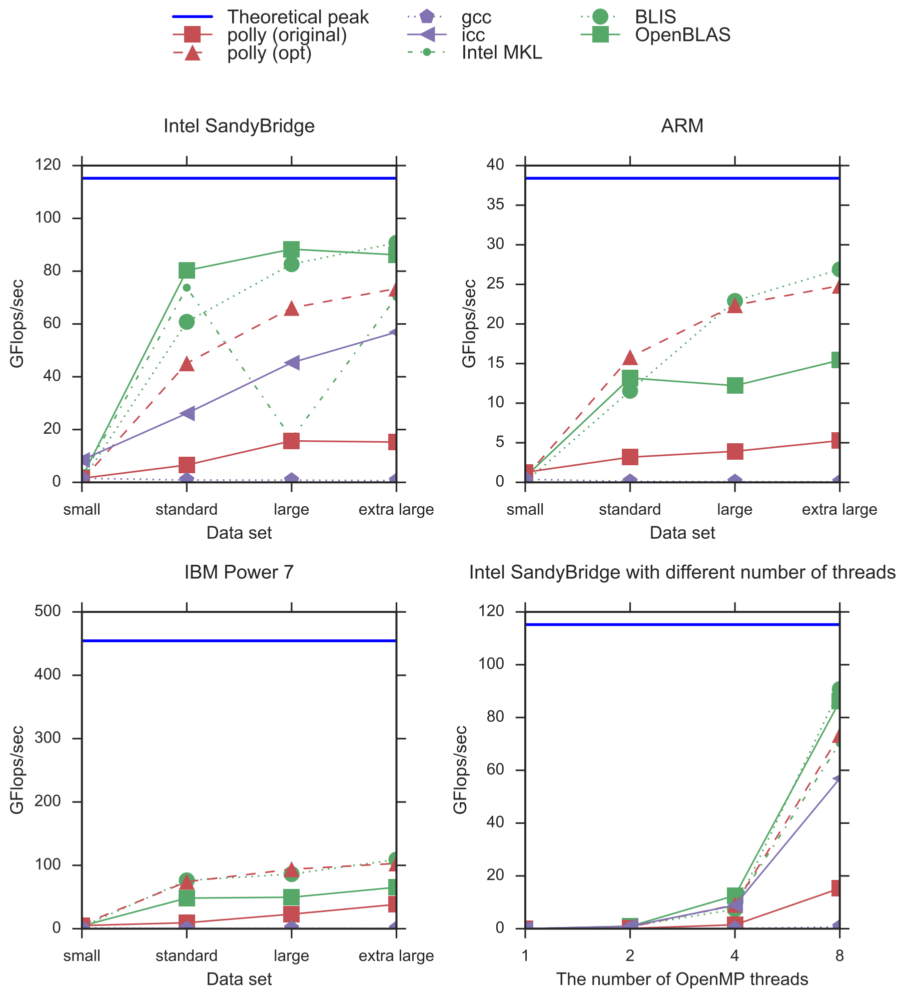

Fig. 1. Matrix-matrix multiplication of matrices that contain elements of type double in the case of Intel Sandy Bridge, ARM, and Power7 platforms

Figure 1 presents the results of performance evaluation of the implementation of the matrix-matrix multiplication provided by PolyBench 3.2 benchmark suite [28] for matrices that contain elements of type double. The maximum number of threads in the OpenMP parallel region is equal to the number of cores specified in Table 1. In the case of Intel SandyBridge with different number of threads, the extra large data set and different values of the maximum number of threads in the OpenMP parallel region are evaluated. We can conclude that the optimization attains more than 85% of performance of the multi-threaded instances of the matrix-matrix multiplication that are available in the optimized BLAS libraries. Figure 1 shows that additional optimizations of the generated assembly code and, in particular, selection of an appropriate ABI may be required to attain high performance in the case of IBM Power 7.

Figure 2 presents the result of performance evaluation for the maximum reliability path problem. We can conclude that the optimization attains more than 66% of theoretical peak performance.

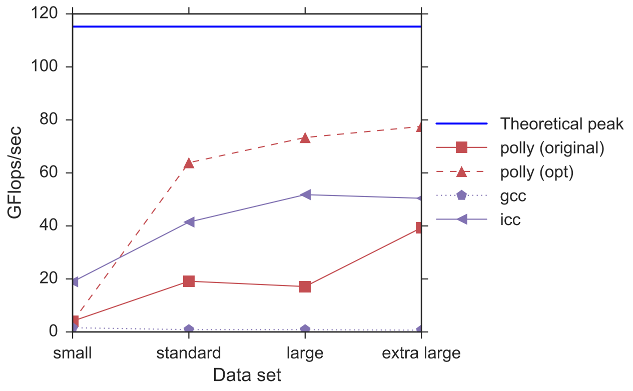

Fig. 2. The maximum reliability path problem in the case of the Intel Sandy Bridge platform

6 Related work

In this section, we describe the work related to automatic parallization of MMA and how our algorithm contributes.

Automatic parallization can be performed by compilers (e.g., ICC [12], IBM XL [13], GCC [10]) to translate a serial program into a multithreaded code. Automatic parallelization attempts to justify the parallelization effort of loops, performs the dataflow analysis (to determine whether each iteration of the loop can be executed independently of the others), and use different technologies (e.g., OpenMP [26]) to take advantage of it. Such approach can suffer from the lack of domain-specific knowledge about algorithms (e.g., the information about use of the cache memory and SIMD registers [25]). Moreover, the extra overhead that is associated with using multiple processors can negate speedup of parallelized code.

To obtain highly optimized multithreaded implementations, autotuning can be used. Automatically Tuned Linear Algebra Software (ATLAS) [29] introduces autotuning to empirically determine optimal parameters of BLAS routines (e.g., blocking and unrolling factors). It can be difficult to apply it in the case of production compilers since autotuning can consume a lot of time.

7 Conclusion and future directions of work

This work presents a new compiler optimization that helps to obtain automatically highly optimized multi-threaded instances of MMA without external code. Our optimization is based on the automatic optimization of MMA implemented in Polly along with approaches used in expert multi-threaded implementations. This allows one to provide the domain-specific knowledge and reuse automatic parallelization implemented in Polly.

In this paper, we compare the execution time of the code produced by the optimization of MMA with the code produced using contemporary compilers (GCC[10], ICC [12], IBM XL [13]), and Clang [11], the compiler front end, as well as multi-threaded instances that are available in Intel's MKL [14], BLIS [15], and OpenBLAS [16]. In the case of GEMM, we attain more than 85% of vendor optimized BLAS library performance. In the case of APPs, we can attain more than 66% of theoretical peak performance.

We believe that higher performance can be achieved with better automatic vectorization used by Polly to produce single-threaded MMA. Determination of optimal values of the analytical model for a multithreaded implementation can be another direction to improve the presented algorithm.

References

1. Lehmann, D. J.: Algebraic structures for transitive closure. Theoretical Computer Science. 4(1), 59–76 (1977)
2. Sedukhin, S. G., Paprzycki, M.: Generalizing Matrix Multiplication for Efficient Computations on Modern Computers. Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics Volume Part I. PPAM'11, Springer-Verlag, Berlin, Heidelberg. 225–234 (2012)
3. Sanders, P.: Fast route planning. Google Tech Talk. (2009)
4. Chen, D. Z.: Developing algorithms and software for geometric path planning problems. ACM Comput. Surv. 28(4) (1996)
5. Jayachandran, S., Venkatachalam, T.: A Secure Scheme for Privacy Preserving Data Mining Using Matrix Encoding. World Engineering & Applied Sciences Journal. 7(3), 190–193 (2016)
6. Cong, J., Xiao, B.: Minimizing Computation in Convolutional Neural Networks. Springer International Publishing, Cham. 281–290 (2014)
7. Akimova, E., Skurydina, A.: On solving the three-dimensional structural gravity problem for the case of a multilayered medium by the componentwize Levenberg-Marquardt method. In: 15th EAGE International Conference on Geoinformatics - Theoretical and Applied Aspects. EAGE (2016). http://earthdoc.eage.org/publication/publicationdetails/?publication=84606
8. Watson, M., Olivares-Amaya, R., Edgar, R. G., Aspuru-Guzik, A.: Accelerating correlated quantum chemistry calculations using graphical processing units. Computing in Science Engineering. 12(4), 40–51 (2010)
9. Goto, K., van de Geijn, R. A.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. 34(3), 1–25 (2008)
10. Stallman, R.: Using and Porting the GNU Compiler Collection: For Gcc-2.95. Free Software Foundation (1999)

11. A C language family frontend for LLVM. https://clang.llvm.org/
12. Intel C++ Compiler 16.0 Update 4 User and Reference Guide. https://software.intel.com/en-us/intel-cplusplus-compiler-16.0-user-and-reference-guide-pdf
13. XL C/C++: Compiler Reference - IBM. http://www-01.ibm.com/support /docview.wss
14. Intel Math Kernel Library (Intel MKL). https://software.intel.com/ru-ru/intel-mkl/?cid=sem43700011401059448&intel_term=intel+mkl
15. Van Zee, F. G., van de Geijn, R. A.: BLIS: A Framework for Rapidly Instantiating BLAS Functionality. ACM Trans. Math. Softw. 41(3), 1–33 (2015)
16. Xianyi, Z., Qian, W., Yunquan, Z.: Model-driven level 3 BLAS performance optimization on Loongson 3A processor. Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on. 684–691. IEEE (2012)
17. Menon, V., Pingali, K.: High-level semantic optimization of numerical codes. Proceedings of the 13th International Conference on Supercomputing. ICS '99, ACM, New York, NY, USA. 434–443 (1999)
18. Grosser, T., Zheng, H., Aloor, R., Simburger, A., Grolinger, A., Pouchet, L. N.: Polly – Polyhedral Optimization in LLVM. 1st International Workshop on Polyhedral Compilation Techniques (IMPACT). Chamonix, France. (2011)
19. Polyhedral Compilation without Polyhedra. http://polycomp.gforge.inria.fr
20. Feautrier, P., Lengauer, C.: Polyhedron Model. Springer US, Boston, MA. 1581–1592 (2011)
21. Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., Temam, O.: Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. Int. J. Parallel Program. 34(3), 261–317 (2006)
22. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. SIGPLAN Not. 43(6), 101–113 (2008)
23. Loechner, V., Wilde, D. K.: Parameterized polyhedra and their vertices. International Journal of Parallel Programming. 25(6), 525–549 (1997)
24. Low, T. M., Igual, F. D., Smith, T. M., Quintana-Orti, E. S.: Analytical Modeling Is Enough for High-Performance BLIS. ACM Trans. Math. Softw. 43(2), 1–18 (2016)
25. Smith, T. M., van de Geijn, R., Smelyanskiy, M., Hammond, J. R., Van Zee, F. G.: Anatomy of High-Performance Many-Threaded Matrix Multiplication. Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium. IPDPS '14, IEEE Computer Society, Washington, DC, USA. 1049–1059 (2014)
26. OpenMP Application Programming Interface. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
27. Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques, and Tools (2Nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (2006)
28. PolyBench/C the Polyhedral Benchmark suite. http://web.cse.ohio-state.edu/ pouchet/software/polybench/
29. Whaley, R. C., Petitet, A., Dongarra, J. J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing. 27(1), 3–35 (2001)