ANN. MULT. GPU PROG.

Revisiting Strassen's Matrix Multiplication for Multicore Systems

Domingo Giménez

Correspondence: domingo@um.es
Department of Computing and Systems, University of Murcia, Spain
Full list of author information is available at the end of the article

Abstract

Strassen's matrix multiplication reduces the computational cost of multiplying matrices of size $n \times n$ from the $O\left(n^3\right)$ cost of a typical three-loop implementation to approximately $O\left(n^{2.81}\right)$. The reduction is made at the expense of additional operations of cost $O\left(n^2\right)$, and additional memory is needed for temporary results of recursive calls. The advantage of Strassen's algorithm is therefore only apparent for larger matrices and it requires careful implementation. The increase in the speed of computational systems with several cores which share a common memory space also makes it more difficult for the algorithm to compete against highly optimized three-loop multiplications. This paper discusses various aspects which need to be addressed when designing Strassen multiplications for today's multicore systems.

Keywords: Strassen matrix multiplication; multicore; linear algebra libraries; batched linear algebra routines

Introduction

Strassen first introduced his matrix multiplication algorithm in 1969 [17]. Its computational cost is of the order $O\left(n^{2.8074}\right)$, while the typical naive, three-loop implementation has a cost of $O\left(n^3\right)$. Since the introduction of the Strassen algorithm, new algorithms have been proposed to further reduce the cost of matrix-matrix multiplication but implementation difficulties for achieving practical high performance for non-huge matrix sizes has led to Strassen's multiplication being known as "fast matrix multiplication".

Matrix multiplication is a widely used basic routine in the solution of scientific and engineering problems either directly or in other higher-level linear algebra routines (LU, QR, etc.). It is also used as the basic operation in linear algebra algorithms by blocks to increase the computation to memory access ratio in order to improve performance [4, 11]. It is therefore os interest to researchers, and there are now many highly-efficient implementations, some of which exploit the parallelism offered by today's multicore systems (MKL [14], MAGMA [2], etc.).

As a result, it is now more difficult to implement Strassen's multiplication in such a way that it is competitive with highly-efficient implementations. The main difficulty lies in the additional memory required by the recursive implementation of the algorithm. More memory entails a decrease in the computation to memory access ratio and, as a result, the algorithm is only competitive with naive but efficient implementations for large matrices.

There have been some recent attempts to implement Strassen's algorithm for today's computational systems [12], and we discuss certain aspects that could be taken into account for implementations optimized for multicore systems.

In the following section we will begin by summarizing the ideas behind Strassen's algorithm. We also elaborate various ideas for optimizing the algorithm for contemporary multicore systems in addition to presenting and commenting on the experimental results on a range of multicore nodes to support

these optimization ideas. We also outline our conclusions together with some possible extensions of the methodology.

Strassen's algorithm

Strassen's algorithm follows the typical divide-and-conquer recursive paradigm [6, 9] and its schema is shown in Algorithm 1. The matrices to be multiplied (A and B) and the result matrix (C) are divided into four submatrices of size $\frac{n}{2} \times \frac{n}{2}$, and the routine is called recursively seven times with matrices of this size. Ten additions and subtractions of matrices $\frac{n}{2} \times \frac{n}{2}$ are made to form the matrices to be multiplied, and the seven resulting matrices are combined with eight additional operations of the same size to obtain the final result. For small matrices, the multiplication is directly obtained with a basic matrix multiplication, which can be a naive, three-loop version or a routine dgemm from an optimized library.

strassen(A,B,C,n,$base$);
Data: Matrices A and B of size $n \times n$. Size of the base case, $base$
Result: $C = AB$
if $n \leq base$ **then**
  C=basicmultiplication(A,B);
else
  strassen($A_{11} + A_{22}$,$B_{11} + B_{22}$,P,$\frac{n}{2}$,$base$);
  strassen($A_{21} + A_{22}$,B_{11},Q,$\frac{n}{2}$,$base$);
  strassen(A_{11},$B_{12} - B_{22}$,R,$\frac{n}{2}$,$base$);
  strassen(A_{22},$B_{21} - B_{11}$,S,$\frac{n}{2}$,$base$);
  strassen($A_{11} + A_{12}$,B_{22},T,$\frac{n}{2}$,$base$);
  strassen($A_{21} - A_{11}$,$B_{11} + B_{12}$,U,$\frac{n}{2}$,$base$);
  strassen($A_{12} - A_{22}$,$B_{11} + B_{22}$,V,$\frac{n}{2}$,$base$);
  $C_{11} = P + S - T + V$;
  $C_{12} = R + T$;
  $C_{21} = Q + S$;
  $C_{22} = P - Q + R + U$;
end

Algorithm 1: Schema of the Strassen matrix-matrix multiplication algorithm.

The execution time follows the recursive equation $t(n) = 7t\left(\frac{n}{2}\right) + \frac{9}{4}n^2$, from which the cost of order $O\left(n^{2.8074}\right)$ is obtained. It is, however, necessary to determine the optimum recursion level (the value of $base$). If we unroll the recursion until size $base$, the equation of the execution time is

$$t(n) = 2k_3 n^{\log 7} base^{\log 7} + k_2 \frac{4}{3}\left(1 - \left(\frac{base}{n}\right)^2\right) \tag{1}$$

where k_3 and k_2 represent the execution time of a multiplication and an addition of two double-precision numbers, respectively. The optimum value of $base$ for a matrix size n can be estimated by deriving Equation 1 in relation to $base$ and equaling zero to obtain $base$ as a function of n. In this way, for a particular system, if the values of k_3 and k_2 are estimated, the value of $base$ can be obtained from the equation connecting it with n. However, the problem is not so simple since the values of k_3 and k_2 are not fixed and depend on the size of the operations to which they correspond and the way in which the data are stored in the memory. When working on multicore nodes, these values also depend on the number of cores working in each operation. A more sophisticated auto-tuning process is therefore needed to determine the optimum value of $base$.

The amount of memory consumed should also be taken into account. The order in which the operations in Algorithm 1 are carried out determines the required memory. The memory cost is

$$m(n) = 3n^2 + \frac{s}{3}n^2\left(1 - \left(\frac{base}{n}\right)^2\right) \quad (2)$$

where s represents the number of submatrices used to store the temporary matrices in the recursive calls to the routine. If the calls are made in the order in which they appear in Algorithm 1, only two temporary matrices are needed, and the amount of memory is $\frac{11}{3}n^2$. It may, however, be preferable to conduct the seven multiplications simultaneously so that the computation is of coarser grain and the parallelism of the system can be better exploited. In this case, the memory used for one multiplication is not reused for other multiplications, $s = 21$, and the total memory is around $10n^2$.

Auto-tuning techniques are widely used for optimizing sequential and parallel routines, and especially for linear algebra routines [5, 13, 15, 16]. In the following section, we analyze the application of some of these techniques to Strassen's algorithm for multicore systems. The order in which the recursive calls are made, the management of temporary memory and the basic routine or library used for the basic multiplications are some of the aspects to be considered.

Optimizing Strassen's algorithm for multicore systems

In this section, we discuss the application of traditional optimization techniques to Strassen's algorithm for multicore systems, before presenting our experimental results in the following section.

A routine can be optimized statically or dynamically. Static auto-tuning is traditionally applied in two ways: with empirical models (`EmpMod`) of the execution time of the routine, or with experimentation at installation (`ExpIns`) time [7, 8]. For Strassen's algorithm, the recursion level is a new parameter to be optimized, and this can additionally depend on the value of another parameter (the number of threads) when parallelism is incorporated into the routine, and this further complicates the auto-tuning process.

For `EmpMod`, the model of the execution time in Equation 1 can be adapted for a computational system with executions of the routine for small problems, and this model or the information about the optimum configuration obtained for these small problems can be used to decide the configuration (recursion level and the number of threads in a multithread version of Strassen's algorithm or the basic multiplication) used when applying the algorithm for other problem sizes.

In `ExpIn`, experiments are conducted when the routine is installed in a system, and the information generated (optimum level and number of threads in parallel routines) in the installation is incorporated into the routine, and this is compiled together with a shell that selects the parameters to be used in the multiplication for a particular problem from the parameters stored for the installation size closest to that of the multiplication being performed.

In dynamic optimization (`DynOpt`), the values of the parameters are selected with experiments which are conducted while the routine is being executed. There is some overhead due to part of the execution being carried out with non-optimum values of the parameters, but the values are selected in the conditions in which the routine is being applied and so the values obtained can be more accurate.

The preferred basic routine used for the multiplications in the basic case in Algorithm 1 varies with the problem size (the value of $base$), and the value of k_3 on Equation 1 depends on the routine used. A polyalgorithmic approach can therefore be used [3], with the basic routine a new parameter to be selected in the auto-tuning process. The basic routine can be optimized for different configurations of the node, for example, for the multicore CPU or for the joint use of CPU and GPU if there is some available (which is normal in contemporary computational nodes).

Experimental analysis

Experiments have been conducted over three multicore systems:

- **6Phenom**, with a CPU AMD Phenom II X6 1075T (6 cores) at 3 GHz, with 16 GB of shared memory
- **12Xeon**, with two hexa-cores Intel Xeon Haswell E5-2620 V3 at 3.4 GHz, with 64 GB of shared memory
- **24Xeon**, with four hexa-cores Intel Xeon E7530 with 32 GB of shared memory at 1.87 GHz

The optimization techniques referred to in the previous section were analyzed in the three nodes and displayed similar results with variations caused by the different numbers of cores and their relative speed. The most significant results are shown for some of the systems, and the variations for other systems are highlighted.

Static optimization

In order to apply the model of the execution time in Equation 1, the values of k_3 and k_2 should be estimated for the computational system we are working with. Small matrices are used for low execution time. For example, we can consider a direct multiplication (recursion level 0) of size 1000×1000 and a Strassen's multiplication of the same size and one level of recursion. In this way, in **6Phenom**, the values are $k_3 = 0.0847\ \mu sec$ and $k_2 = 5.06\ \mu sec$. There is a big difference in the values of k_2 and k_3, which might be due to the use of the optimized `dgemm` routine of MKL for the basic multiplications and the additions being directly programmed. The methodology explained here does however work the same for other implementations. By performing the experiments with larger sizes and adjusting the data to the least squares approximation, better approximations could be obtained, but the simple approach used here provides satisfactory results. The experiments with the sizes used takes less than 1 second.

For `ExpIns`, experiments are conducted in **6Phenom** at installation time with sizes $\{1000, 3000, 5000, 7000\}$, and the optimum recursion level (we consider four levels: direct multiplication, and recursion from level 1 to 3) for each size is stored to be used at execution time. The manager can decide the problem sizes used for the installation, depending on the problem sizes that will normally be solved. In order to reduce the installation time, executions are performed for each problem in the installation set beginning with recursion level 0 (direct multiplication) and increasing the level while the multiplication time decreases. In this way, the largest executions are avoided. The installation time is around 285 seconds, which is much larger than that of `EmpMod`, but affordable if the predictions are better.

In order to compare the results obtained with the two optimization methods, experiments are carried out for a validation set $\{2000, 4000, 6000, 8000\}$. For these sizes, the recursion levels with which the lowest experimental execution time is obtained are 1, 2, 3 and 3, which matches the selection with `EmpMod` and with `ExpIns`. Although satisfactory results are therefore obtained with either of the two techniques, the empirical modeling technique has a shorter installation time.

Since the mean of the improvement in the execution time in relation to that with direct multiplication is around 37%, the Strassen multiplication is preferable even when an optimized routine (`dgemm` of MKL) is used for the direct multiplication. The difference in terms of the time obtained with the greatest recursive level (level 3) is around 3.3%, which means that using this level as the default level is a good option particularly for the largest problems for which this is the optimum level.

In faster systems, the accurate prediction of the execution time is more difficult and this also hampers a satisfactory selection of the recursion level. **24Xeon** works about twice as slow as **6Phenom** when working in a matrix multiplication, and **12Xeon** works around 2.5 times faster.

For the slowest system (**24Xeon**), the recursion levels are also correctly selected for the four validation sizes (2, 3, 3 and 3), and the mean improvement in relation to the direct multiplication is 79.6%. The installation times are $0.6\ \mu sec$ and $437\ \mu sec$.

The results are slightly different for faster systems. For example, the selections in **12Xeon** are shown in Table 1. The level with the lowest execution time is labeled `Low`, and this is compared with the levels selected with `EmpMod` and `EmpIns`. Fewer than half of the selections are correct, and now the recursion level selected with the two optimization techniques is different. The difference with the lowest execution time is 4.3% for `EmpMod` and 1% for `ExpIns`. Furthermore, the installation time in this system drops to 0.2 and 60 seconds for `EmpMod` and `ExpIns`.

Table 1 Recursion level with which the lowest time is obtained (`Low`), the level selected with empirical modeling (`EmpMod`) and that with the experimental technique (`ExpIns`), for different matrix sizes in 12Xeon.

	Low	EmpoMod	ExpIns
2000	1	1	1
4000	1	**2**	1
6000	2	**3**	**1**
8000	2	**3**	**1**

Use of multithread routines

The experiments conducted so far were with the `dgemm` routine from MKL with only one thread. MKL is a multithread library, and so parallelism can be exploited implicitly merely by calling the routine with the desired number of threads, possibly with this number equal to the number of cores in the system. In this way, there is a reduction in the execution time of the direct multiplication and, as with faster systems, it will be more difficult to outperform `dgemm` with the Strassen multiplication.

Table 2 compares the evolution of the recursion level with which the lowest execution time is obtained (`Low`) when the size of the matrices increases (sizes 4000, 6000 and 8000) and with systems with different speeds and varying the number of threads used in `dgemm`. The levels selected with `EmpMod` and with `ExpIns` are also shown, as are those selected dynamically (`DynOpt`), and the selections that do not coincide with the lowest time are highlighted. Although the table shows the results until six threads, **12Xeon** comprises 12 cores and **24Xeon** 24 cores.

Table 2 Evolution of the recursion level with which the lowest time is obtained for different matrix sizes and computational systems when the number of threads varies

matrix size:		4000						6000						8000					
#threads:		1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6
24Xeon	`Low`	3	2	1	1	1	0	3	3	2	1	1	1	3	3	2	2	1	0
	`EmpMod`	3	2	1	1	**0**	**1**	3	3	2	**2**	1	1	3	3	2	2	1	**2**
	`ExpIns`	3	2	1	1	1	**1**	3	3	2	**2**	1	1	3	3	2	2	1	**1**
	`DynOpt`	**2**	2	1	1	1	**1**	3	3	2	1	1	1	**4**	3	2	2	1	**1**
6Phenom	`Low`	2	2	1	0	0	0	3	1	1	1	1	1	3	3	2	1	1	1
	`EmpMod`	2	2	1	**1**	**1**	**1**	3	**3**	**2**	1	1	**2**	3	3	2	**2**	**2**	**2**
	`ExpIns`	2	**1**	1	0	0	**1**	3	**2**	**2**	1	1	**0**	3	**2**	2	1	1	**0**
	`DynOpt`	2	**1**	1	**1**	**1**	**1**	3	**2**	1	1	1	1	3	**2**	2	1	1	1
12Xeon	`Low`	1	1	0	0	0	0	2	1	1	1	0	0	2	1	1	1	0	0
	`EmpMod`	**2**	1	**1**	**1**	**1**	**1**	**3**	**2**	**2**	**2**	**2**	**1**	**3**	**2**	**2**	**2**	**2**	**2**
	`ExpIns`	1	1	0	0	0	0	**1**	1	**0**	**0**	0	0	**1**	1	1	**0**	0	0
	`DynOpt`	**2**	**2**	**1**	**1**	**1**	**1**	2	1	1	1	**1**	**1**	2	1	1	1	**1**	**1**

Some conclusions can be drawn:

- In general, the optimum level decreases with the computational cost: it is lower for smaller problems, for faster systems or for more threads. Strassen's algorithm will only therefore be competitive with the multithread implementation of `dgemm` for very large problems.
- The quality of the prediction with `EmpMod` decreases for faster systems.

- ExpIns makes better selections, especially for faster systems. It is therefore preferable for there to be no drastic increase in its installation time.

Table 3 shows the installation time (in seconds) for EmpMod and ExpIns in the three systems considered and a limit of six cores. In EmpMod the values of k_3 and k_2 are estimated independently for numbers of threads from 1 to 6, but the installation times are very low in comparison with those with ExpIns. In any case, for the slowest system the installation time is lower than 19 minutes, which is affordable if better selections are achieved.

Table 3 Installation time (in seconds) with EmpMod and ExpIns in different computational systems with a maximum of six threads

	12Xeon	6Phenom	24Xeon
EmpMod	0.40	1.04	1.58
ExpIns	143	708	1122

Figure 1 shows the quotient of the execution time in terms of time with Strassen's multiplication for recursion levels 1, 2 and 3. The results are shown for the three systems, with matrix sizes 8000 and 12000, and varying the number of threads from 1 to the number of cores in the system. For these large sizes, the improvement achieved with Strassen's multiplication in relation to the routine dgemm is between 10% and 30% when only one thread is used. When the number of threads increases, the difference decreases and the direct method is largely preferable. For a number of threads equal to that of the cores, the advantage sometimes obtained with recursion level 1 is residual for matrices of size 8000, but the difference increases with the matrix size. These results confirm the fact that Strassen's method should only be used in large multicores for very large multiplications.

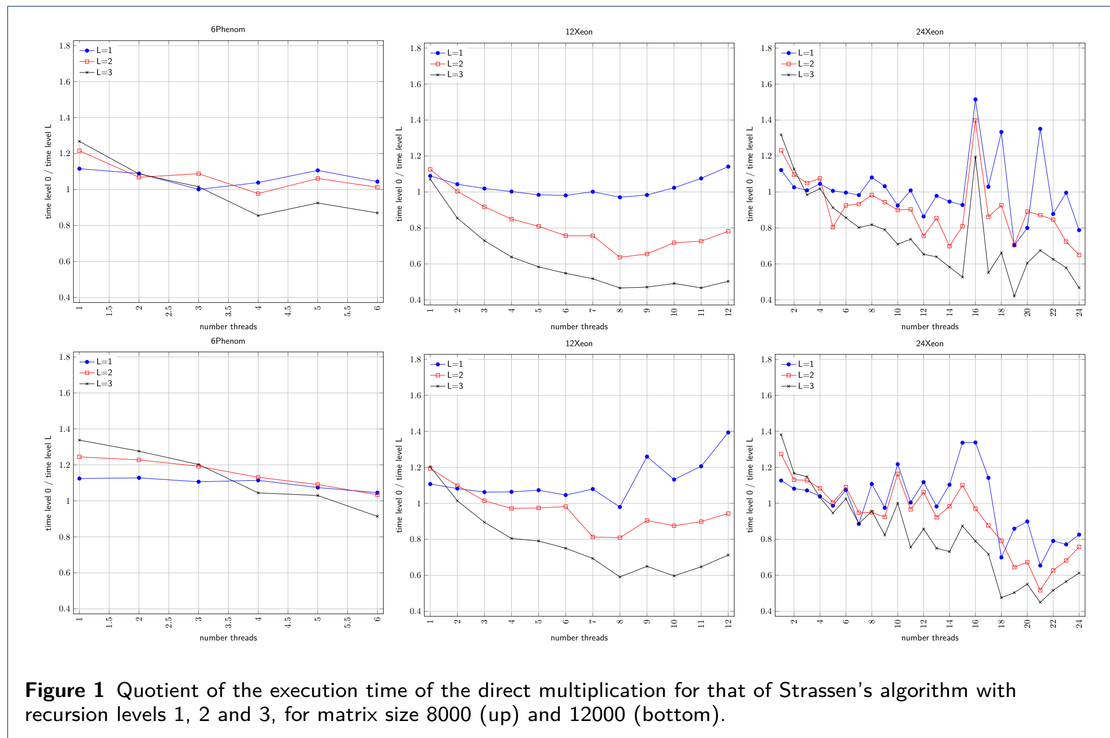

Figure 1 Quotient of the execution time of the direct multiplication for that of Strassen's algorithm with recursion levels 1, 2 and 3, for matrix size 8000 (up) and 12000 (bottom).

Dynamic optimization

The recursion level can be selected dynamically during execution of the routine. The routine strassen in Algorithm 1 is called without a base case, and the seven multiplications in the algorithm are called

initially with an increasing level of recursion (with $base = N/2, N/4, \ldots$). The execution time of each call is obtained and the lowest is stored. When the time of one call is higher than the lowest, the recursion level for the remaining calls is fixed to the level which provided the lowest time. Because there are only seven multiplications, this method can only give satisfactory results if the optimum recursion level is not high, which is the case according to the results in Table 2. As we can see in the table, one of the disadvantages of this method is that it is not possible to select level 0. Nevertheless, the method adapts well to the speed of the systems and the size of the matrices. This is confirmed in Figure 2, which shows the quotient of the execution time of Strassen's algorithm with the optimum recursion level obtained experimentally in terms of time with `DynOpt`. The times are very close and differences might arise from different runnings. The obtained approximation is only unsatisfactory when a large number of threads is used (more than 14) and this is due to the optimum level being 0 (direct multiplication) although the experiments are conducted by `DynOpt` with levels 1 and 2.

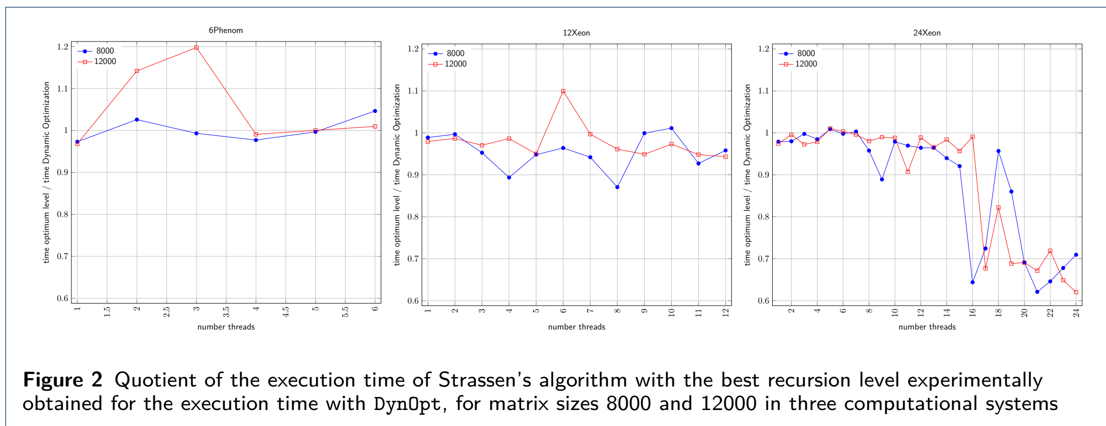

Figure 2 Quotient of the execution time of Strassen's algorithm with the best recursion level experimentally obtained for the execution time with `DynOpt`, for matrix sizes 8000 and 12000 in three computational systems

Conclusions and future work

Strassen's matrix multiplication is referred to as "fast matrix multiplication" because of its cost of order $O\left(n^{2.8074}\right)$ which outperforms the $O\left(n^3\right)$ of the naive, three-loop multiplication. It does, however, incur memory and computation penalizations. Multicore nodes are the standard basic components of today's computational systems, and there are highly optimized, multithread implementations of the matrix multiplication for these. In order, therefore, to make Strassen's multiplication competitive with these implementations, it is necessary to combine its divide-and-conquer approach with parallelism and optimization techniques for parallel routines. This paper analyzes the application to Strassen's algorithm of some of these techniques: static or dynamic optimization, with the use of a model of the execution time or with experimental installation. In order to avoid system-dependent conclusions, experiments were conducted in three multicore systems with a different number of cores and at different speeds.

The conclusions are partially satisfactory. The static optimization techniques adequately select the recursion level for certain systems and when the number of cores is not very large, which makes Strassen's algorithm only competitive for very large matrices. The dynamic version of the algorithm gives results which are close to the optimum of Strassen's algorithm so that it is not necessary to install the routine for each system.

Additional optimizations are, however, needed to better adapt Strassen's algorithm to multicore systems. We have not considered memory issues, and the affinity in the assignation of threads to the cores in the system should be considered. We are also working on the application of mutilevel

parallelism, which is obtained by combining explicit OpenMP parallelism with implicit parallelism in the underlying multiplication used in the base case. The number of threads to be used at each level should be selected.

The development and optimization of batched linear algebra routines has recently attracted attention [1, 10] due to their presence in many scientific problems. We are working on the application of batched computation. For example, the level before the last level of the recursion can be implemented in such a way that seven basic multiplications are performed in parallel. These multiplications can be performed in different ways, and the arrangement of the multiplications must be decided. For example, if the multicore comprises six cores, the seven multiplications can be performed consecutively, working with the six cores in each of the multiplications. The configuration would be represented as $7(1 \times 6)$, which means that seven groups of one multiplication are performed with six cores. It is also possible to perform three consecutive pairs of simultaneous multiplications, each using three cores followed by one multiplication with six cores $(3(2 \times 3) + 1(1 \times 6))$, or two consecutive triples of multiplications with two cores each, followed by one multiplication with six cores $(2(3 \times 2) + 1(1 \times 6))$. Once the preferred configuration has been determined, it can be used for each group of seven multiplications, but the best configuration will depend on the problem size, and so there is influence between the recursion level, the basic routine and the arrangement selected to perform the operations.

Acknowledgments

This work was supported by the Spanish MINECO and also by European Commission FEDER funds, under grant TIN2015-66972-C5-3-R.

References

1. A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra. Performance, design, and autotuning of batched GEMM for GPUs. In *Proceedings of 31st International Conference on High Performance Computing*, pages 21–38, 2016.
2. E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1), 2009.
3. P. Alberti, P. Alonso, A. M. Vidal, J. Cuenca, and D. Giménez. Designing polylibraries to speed up linear algebra computations. *International Journal of High Performance Computing and Networking*, 1(1/2/3):75–84, 2004.
4. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. J. Dongarra, J. D. Croz, A. Grenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1995.
5. G. Bernabé, J. Cuenca, L. García, and D. Giménez. Auto-tuning techniques for linear algebra routines on hybrid platforms. *J. Comput. Science*, 10:299–310, 2015.
6. G. Brassard and P. Bratley. *Fundamentals of Algorithms*. Prentice-Hall, 1996.
7. J. Cámara, J. Cuenca, L. García, and D. Giménez. Auto-tuned nested parallelism: A way to reduce the execution time of scientific software in NUMA systems. *Parallel Computing*, 40(7):309–327, 2014.
8. J. Cámara, J. Cuenca, D. Giménez, L. García, and A. M. Vidal. Empirical installation of linear algebra shared-memory subroutines for auto-tuning. *International Journal of Parallel Programming*, 42(3):408–434, 2014.
9. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
10. J. J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, and M. Zounon. Optimized batched linear algebra for modern architectures. In *Proceedings of Euro-Par 2017*, pages 511–522, 2017.
11. G. Golub and C. F. V. Loan. *Matrix Computations*. The John Hopkins University Press, fourth edition, 2013.
12. J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn. Strassen's algorithm reloaded. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pages 690–701, 2016.
13. S. Hunold and T. Rauber. Automatic tuning of PDGEMM towards optimal performance. In *11th International Euro-Par Conference, Lecture Notes in Computer Science*, volume 3648, pages 837–846, 2005.
14. Intel MKL web page. `http://software.intel.com/en-us/intel-mkl/`.
15. J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, 22(1):15–44, 2010.
16. T. Sakurai, T. Katagiri, H. Kuroda, K. Naono, M. Igai, and S. Ohshima. A sparse matrix library with automatic selection of iterative solvers and preconditioners. In *Proceedings of the International Conference on Computational Science (ICCS)*, LNCS, pages 1332–1341, Barcelona, Spain, June 2013.
17. V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 3(14):354–356, 1969.