Matrix Multiplication on High-Density Multi-GPU Architectures: Theoretical and Experimental Investigations

Peng Zhang[1(✉)] and Yuxiang Gao[2]

[1] Biomedical Engineering Department,
Stony Brook University, Stony Brook, NY 11794, USA
peng.zhang@stonybrook.edu
[2] Cluster Solution Department, Cray Inc., San Jose, CA 95112, USA

Abstract. Matrix multiplication (MM) is one of the core problems in the high performance computing domain and its efficiency impacts performances of almost all matrix problems. The high-density multi-GPU architecture escalates the complexities of such classical problem, though it greatly exceeds the capacities of previous homogeneous multicore architectures. In order to fully exploit the potential of such multi-accelerator architectures for multiplying matrices, we systematically evaluate the performances of two prevailing tile-based MM algorithms, standard and Strassen. We use a high-density multi-GPU server, CS-Storm which can support up to eight NVIDIA GPU cards and we test three generations of GPU cards which are K20Xm, K40m and K80. Our results show that (1) Strassen is often faster than standard method on multicore architecture but it is not beneficial for small enough matrices. (2) Strassen is more efficient than standard algorithm on low-density GPU solutions but it quickly loses its superior on high-density GPU solutions. This is a result of more additions needed in Strassen than in standard algorithm. Experimental results indicate that: though Strassen needs less arithmetic operations than standard algorithm, the heterogeneity of computing resources is a key factor of determining the best-practice algorithm.

Keywords: Matrix multiplication · Performance evaluation · Heterogeneous architectures · High-density multi-GPU architectures

1 Introduction

Since ENIAC was announced in 1946, researchers never stop to seek a faster approach for multiplying matrices. Not only one of the kernels in numerical linear algebra, the problem of matrix multiplication (MM) is also a bottleneck for almost all matrix problems such as least square problem and eigenvalue problem [1–5]. The key problem has widely been studied in computing theory and in practical implementation. Mathematicians have been looking for the possible lowest bound for multiplying matrices. The standard method for multiplying two $n \times n$ matrices is $O(n^3)$. In 1969, Strassen reduced the computing complexity to $O(n^{2.807})$ [6]. In 1987, a big breakthrough of this problem is the Coppersmith-Winograd algorithm which can do MM in $O(n^{2.376})$

J.M. Kunkel and T. Ludwig (Eds.): ISC High Performance 2015, LNCS 9137, pp. 17–30, 2015.
DOI: 10.1007/978-3-319-20119-1_2

operations [7]. More new algorithms are proposed for beating the records and approaching the true lowest bound [8].

Practical implementation is essential to exploit the proposed algorithms on the novel parallel computing facilities [5, 9–20]. Different from theoretical studies, the complex characteristics of computing facilities need to be taken into the design of parallel programs. In distributed computing, communication needs to be minimized [12, 19, 21]. Particularly the task mapping problem for the Strassen algorithm is addressed for balancing multiplications [9, 18]. In the latest heterogeneous architectures, MM needs to be optimized on special-purposed processors and accelerators, such as CELL processor [5] and graphics processing units (GPUs) [11]. Besides, many high-performance implementations of MM are developed such as in GotoBLAS [22, 23], ATLAS [10], LAPACK [24] and CUBLAS [23]. To accommodate the ever-changing computing architectures, new algorithms have been designed and developed with the birth of new technologies. Recently, high-density multi-GPU technologies are available to the community of supercomputing. For example, a 2U server node can be configurated with 8 NVIDIA GPU cards in CS-Storm [25, 26], featuring up a high-density space-efficient design for integrating multiple GPUs. This design has significantly escalated computing complexities, though it greatly improves computing performance. There is a need to investigate MM algorithms on this novel architecture.

This motivated the work to investigate the performance of the standard and Strassen tile-based algorithms on the high-density multi-GPU architecture. Our contributions in the work are: (i) to systematically compare the performances of the standard and Strassen algorithms on the high-density multi-GPU platforms; (ii) to find out the optimal algorithms through extensive experiments for a wide range of problem sizes under different system configurations; (iii) to present the performance characteristics to the researchers and the engineers for better algorithmic and engineering designs.

The paper is organized as follows: standard and Strassen algorithms are reviewed in Sect. 2. Theoretical evaluation is presented in terms of floating-point operations and execution time in Sect. 3. High-density multi-GPU architecture is described in terms of hardware specifications and software stacks in Sect. 4. Experimental results are presented and analyzed in Sect. 5. Conclusion is drawn in Sect. 7.

2 Matrix Multiplication Algorithms

We consider the tiled matrix multiplication (MM) algorithms on shared-memory multicore and multi-accelerator systems. The first method is the standard tiled MM algorithm and it is also referred to as the *Naïve* method thereafter. Naïve method partitions each input matrix into a block matrix whose tiles are submatrices of identical sizes. Based on the given partition, the computing products of submatrices are performed concurrently. The other method is the *Strassen* algorithm, which is often faster than Naïve on the multicore systems for large size matrices. Figure 1 gives the examples to show the data partition and computing flows for both methods. In the examples, the input matrices A and B are partitioned into 2 × 2 block matrices. Each tile (namely, submatrix) is referred to as atomic data module. Intermediate data modules are needed to buffer intermediate results. Finally, the resultant matrix C is computed and stored in the same manner. This case of multiplying two 2 × 2 matrices shows that Strassen saves one multiplication at the expense of 14 more additions. The cost of multiplying matrices is often highlighted; however, the cost of adding matrices is somewhat ignored in the analysis of most algorithms. This could result in the problems in practical implementations. For example, clearly the benefit of Strassen would be marginal for small enough matrices. It is observed that in practice on the multicore systems, there is a performance crosspoint between Strassen and Naïve [10, 27]. However, in this work, we'd ask one question: is there a performance crosspoint for large matrices on heterogeneous architectures?

For convenience of description, we assume that: input matrix is a square matrix of size $N \times N$; the tiled partition is $(2p) \times (2p)$; and, each tile is a square submatrix of size $n \times n$. Thus, $N = 2p \times n$. N, p and n are positive integers. p is called as partition factor. In the example (Fig. 1), the titled partition is 2 × 2 and $p = 1$.

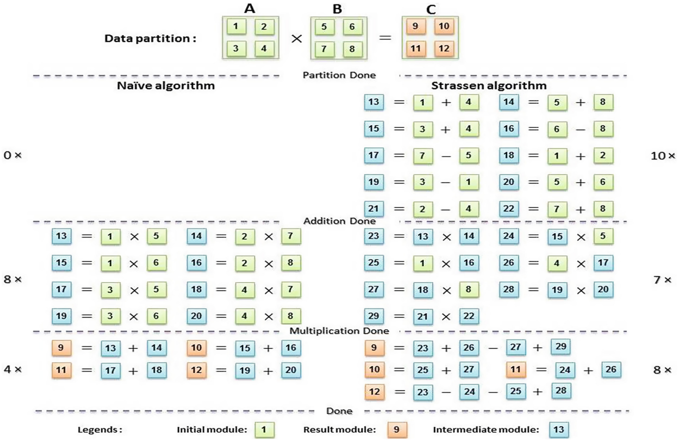


Fig. 1. Data partitions and computing procedures for Naïve and Strassen algorithms

3 Theoretical Evaluation

We conduct the theoretical evaluation for two methods in terms of floating-point operations (FLOP) and execution time.

Floating-point operations (FLOP): Let F_{mm} (p, n) and F_{st} (p, n) be the number of floating-point operations (FLOP) that is required by Naïve and Strassen, respectively. The formulas are written as:

$$F_{mm}(p,n) = 8p^3 \cdot f_m(n) + p^2(8p-4) \cdot f_a(n) \tag{1}$$

$$F_{st}(p,n) = 7p^3 \cdot f_m(n) + p^2(22p-4) \cdot f_a(n) \tag{2}$$

Here $f_m(n) = n^2(2n-1)$ and $f_a(n) = n^2$ are the FLOP for multiplying and adding $n \times n$ matrices.

Let the ratio of $F_{st}(p,n)$ over $F_{mm}(p,n)$ be $\gamma(p,n)$ written as:

$$\gamma(p,n) = \frac{F_{st}(p,n)}{F_{mm}(p,n)} = \frac{7p(2n-1)+22p-4}{8p(2n-1)+8p-4} \sim 0.875 - \frac{15}{n} \tag{3}$$

The partition factor p is often small in tiled algorithms. Then, we see: $\gamma(p,n) \rightarrow 0.875$ as $n \rightarrow \infty$. Figure 2 illustrates the evolution of ratio $\gamma(p,n)$ under certain conditions. This reaffirmed the asymptotic complexities for Naïve and Strassen and it also indicated the performances on multicore architectures [6].

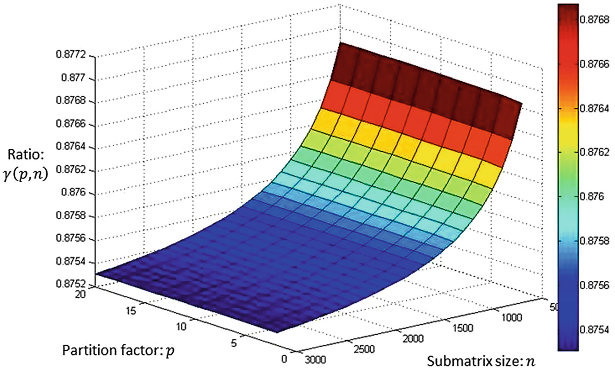


Fig. 2. Ratio of FLOP (Strassen) over FLOP (Naïve) with varied partition factor p and submatrix size n

Execution Time: The performances of these two methods are then evaluated in terms of execution time. Let $T_m(n)$ and $T_a(n)$ be the time for multiplying and adding two submatrices of size $n \times n$, respectively. Let $T_{mn}(p,n)$ and $T_{st}(p,n)$ are the total time needed for Naïve and Strassen and written as:

$$T_{mm}(p,n) = 8p^3 \cdot T_m(n) + p^2(8p-4) \cdot T_a(n) \tag{4}$$

$$T_{st}(p,n) = 7p^3 \cdot T_m(n) + p^2(22p-4) \cdot T_a(n) \tag{5}$$

On homogeneous multicore architectures, multiplication is often more time-consuming than addition for large enough matrices so we assume: $T_m(n) \gg T_a(n)$ for large n. Let n_p be the number of processor cores that process concurrently. Thus, we have the facts on homogenous multicore systems that:

- Strassen is more efficient than Naïve and its improvement is ~ 12.5 %.
- Parallel efficiency is nearly perfect when the number $7p^3$ is a multiple of n_p. Naturally, it is because $7p^3$ multiplication instances could be distributed evenly on the n_p processor cores, thus leading to perfect balanced multiplying operations [9].

However, the situation may be different on the heterogeneous multi-GPU architecture. The difference is due to the disparity of GPU and CPU performances (Table 1). We calculate the ratio of $T_{st}(p,n)$ over $T_{mm}(p,n)$ as:

Table 1. Performance comparison between GPUs and CPUs

	Peak floating point performances (TFlops)	
	Double-precision	Single-precision
Tesla K20	1.17	3.52
Tesla K40	1.43	4.29
Tesla K80	1.87	5.60
Xeon E5-2670	0.166	0.333

$$\beta(p,n) = \frac{T_{st}(p,n)}{T_{mm}(p,n)} = \frac{p(7\cdot\kappa(n)+22)-4}{p(8\cdot\kappa(n)+8)-4} \sim \frac{7\cdot\kappa(n)+22}{8\cdot\kappa(n)+8} \tag{6}$$

Here $\kappa(n) = T_m(n)/T_a(n)$ is the ratio of multiplication time over addition time for submatrices of size $n \times n$. Smaller $\beta(p,n)$ means that Strassen is more efficient than Naïve. Writing $\kappa(n)$ as a function of $\beta(p,n)$ yields:

$$\kappa(n) \sim -\frac{8\cdot\beta(p,n)-22}{8\cdot\beta(p,n)-7} \tag{7}$$

This helps find out the asymptotic trend:

$$\lim_{\beta(p,n)\to 1}\kappa(n) = 14 \tag{8}$$

Secondly, we have:

$$\frac{\partial\beta(p,n)}{\partial\kappa(n)} \sim \frac{-15}{8(\kappa(n)+1)^2} < 0 \tag{9}$$

From Eqs. (8) and (9), we find out that:

- If $\kappa(n) > 14$, Strassen could be faster than Naïve. On the other hand, if $\kappa(n) < 14$, Naïve could outperform Strassen, though it required more FLOP.
- Thus, when multiplication is much faster than addition, Strassen is greatly beneficial, compared to Naïve. However, when multiplication becomes as fast as addition, Naïve may in turn surpass Strassen.

Figure 3 shows the changes of $\beta(p,n)$ under varied p and $\kappa(n)$. Therefore, this made a possible: when/if the multiplication could be as fast as addition, Naïve could outperform Strassen. This assumption is hardly achievable in today's processors but it maybe holds on the hybrid multi-GPU architectures.

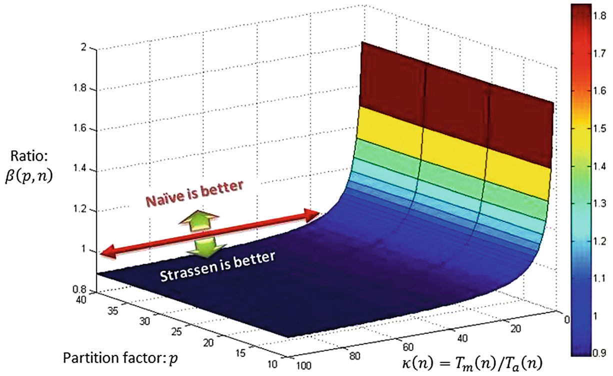


Fig. 3. Ratio of $T_{st}(p,n)$ (Strassen) over $T_{mm}(p,n)$ (naïve) with varied partition factor p and $\kappa(n)$

4 High-Density Multi-GPU Architecture

4.1 Hardware

CS-Storm [25] is used for performing all experiments and it is a 2U sever that can be equipped with up to 8 NVIDIA Tesla GPU cards and 2 Intel Xeon processors. In this work, we test three multi-GPU systems. Table 2 lists the hardware specifications. On the board, four PCIe switches are enclosed; each hooking up to 2 GPUs with the host.

Table 2. Hardware specification for multi-K20/K40/K80 server nodes

Systems	GPU hardware				CPU hardware			
	Model	# of GPUS	Total CUDA Cores	GDDR5 / GPU (GB)	Model	# of CPUS	Total CPU Cores	Total Host Memory (GB)
K20	K20Xm (1x Kepler GK110)	4	10,752	5.76	Intel Xeon E5-2670 v2	2	20	165
K40	K40m (1x Kepler GK110B)	8	23,040	11.52	Intel Xeon E5-2670 v2	2	20	165
K80	K80 (2x Kepler GK210)	16	39,936	11.52	Intel Xeon E5-2680 v3	2	24	165

4.2 Software

System software includes RHEL 6.5 and NVIDIA driver 340.32. For best performances of subprograms on CPUs and GPUs, we select two BLAS (basic linear algebra subprograms) libraries: Intel Math Kernel Library (MKL v11.2) for CPUs and CUBLAS (CUDA 6.5) for GPUs. Complier package is Intel Parallel Studio 2015.

A data-oriented mapping paradigm (DMP) is extended to distribute tasks among CPUs and GPUs [28]. Following the work [28], we describe the scheduler work. In the tiled algorithms, the tiles are treated as data modules. All the tiles associated with input matrices are treated as initial data modules. All the tiles that belong to the resultant matrix are referred to as resultant data modules. Intermediate tiles are treated as intermediate data modules. All of data modules are given by an identifier. In the method, when a function is defined as $d_s = f(d_1, d_2)$, we say d_s depends on d_1 and d_2. Here, d_s, d_1 and d_2 are the identifiers of data modules and the function f is either the multiplication or the addition. In this manner, the data dependency is defined. A function in the method is referred to as a task in the computing. In the computing, initial data modules are first loaded and ready to use. Then a dedicated scheduler checks the availability of new tasks until all tasks are done. A task is available as long as the input data modules it requires are ready to use. The scheduler sends a new task to the next available CPU core or GPU card, as long as the task is available. We further add an arbitrator layer in the scheduler, which allows the scheduler to distribute specified tasks to preferred platforms. For example, the scheduler could distribute the addition tasks only on CPUs and the multiplication tasks only on GPUs.

5 Experimental Evaluation

5.1 Performance Metrics

Wallclock time (in seconds) is used for timing. $T(A)/T(B)$ denotes the wallclock time for method A and B. $S(A,B) = T(B)/T(A)$ is the speedup for A over B. Performance improvement for A over B is defined as $P(A,B)$ and calculated as:

$$P(A,B) = \frac{T(B) - T(A)}{T(B)} = 1 - S(A,B)^{-1} \tag{10}$$

As this equation suggests, a positive $P(A,B)$ implies A is faster than B; otherwise, B is faster than A. In addition, for the clarity of showing schedulability, parallel activities trace (PAT) is proposed to illustrate the activities of concurrent computations. PAT is the 2D graphic scheme, in which the horizontal axis shows wallclock time and the vertical axis implies the device type (CPU/GPU) and thread identifiers (IDs). Different colors refer to different types of tasks (functions). Naturally, two ends of a color bar indicate starting and ending time points of data processing of a particular task so the length means the amount of time the task takes. PAT graphically helps illustrate the complexities of parallel activities of parallel programs. 64-bit and 32-bit precision floating-point formats are tested. Input matrices are partitioned as 4 × 4 tiles.

5.2 Homogeneous Multicore Architectures

Parallel efficiency is nearly perfect when the number of concurrent processes is a multiple of 7 thanks to the nature of Strassen [9, 29]. Thus, we benchmark both methods using 7 and 14 processor cores for a range of varied problem sizes. Figures 6 and 7 present the performances for Strassen and Naïve using the 64-bit (double) and 32-bit (single) precisions, respectively. Performance improvements of Strassen over

Naïve are accordingly calculated. From these results, we can find out: (i) Strassen lost its superiority for small enough sizes. With the increase of problem sizes, Strassen gains more benefits and it becomes consistently more efficient than Naïve. On seven cores, Strassen improved the performance by a factor of 13 ~ 14 %, compared with Naïve. On 14 cores, performance improvements of Strassen over Naïve increase to 16 ~ 17 %. This reaffirms: Strassen is more beneficial for large enough sizes. The precision of floating-point numbers has a dominating impact on absolute performances but it has a relatively small impact on performance improvements of Strassen over Naïve.

5.3 Heterogeneous Multi-GPU Architectures

5.3.1 Heterogeneity of Performances

When migrating from homogeneous multicore to heterogeneous multi-GPU architectures, we need to exam the performances of two key operations, multiplication and addition, on processors and accelerators. Figure 4 presents the performances for varied problem sizes. The results show that: (i) GPU multiplication (CUBLAS_DGEMM/CUBLAS_SGEMM) is dominantly faster than CPU multiplication (MKL_DGEMM/MKL_SGEMM). (ii) CPU addition (MKL_DOMATADD/MKL_SOMATADD) is even faster than GPU addition (CUBLAS_DAXPY/CUBLAS_SAXPY), due to data transfer overhead between the host and the GPU devices. Thus, CPU is still the optimal platform to perform the matrix addition but GPU becomes the best choice for the matrix multiplication for large enough size. (iii) Lastly, CPU multiplication is typically 2 ~ 3 orders of magnitude slower than CPU addition; however, GPU multiplication is merely one order of magnitude slower than CPU addition. Figure 5 shows the ratios. This trend made a possible that Naïve may outperform Strassen on multi-GPU architectures, on which multiplication is not that slower than addition.

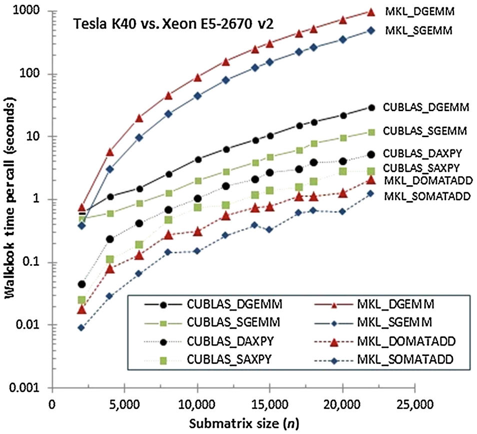


Fig. 4. Multiplication and addition performances on CPU/GPU

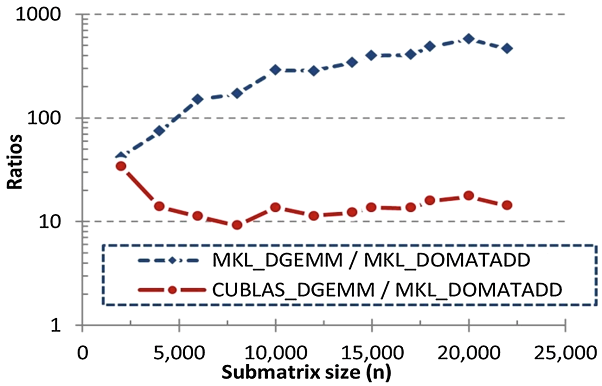
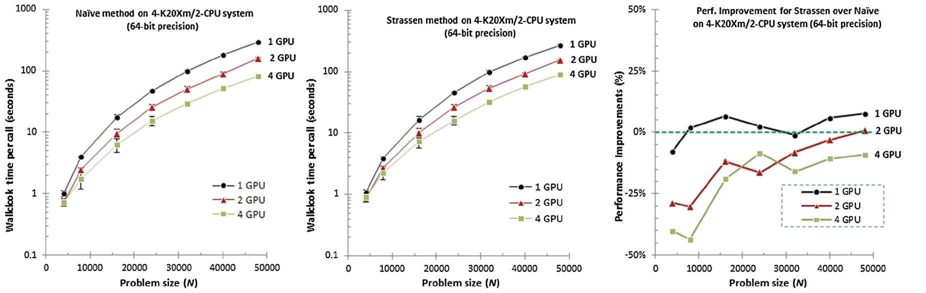


Fig. 5. Ratios of CPU-/GPU multiplication time over CPU addition time

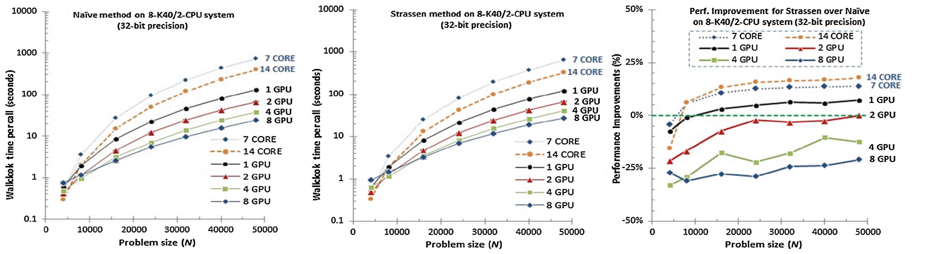
5.3.2 Strassen vs. Naïve on 8x K40m

Figures 6 and 7 present the absolute performances of two methods in (a) and (b), and then the performance improvements for Strassen over Naïve in (c), for 64-bit (double) and 32-bit (single) precision, respectively. From the results, we can find out that: (i) The GPU-solutions are often superior to the CPU-only solutions for large enough matrices. (ii) High-density GPU-solutions are always better than low-density GPU-solutions. (iii) In the single-GPU solution (i.e., 1-GPU), Strassen still retains its superiority over Naïve. This could be because single GPU per node cannot give enough competitive performance for multiplying matrices, compared with the performance for adding matrices given by the many processor cores. Under this condition, Strassen could be still more efficient than Naïve since it needs fewer multiplications than Naïve. (iv) However, with the increase of GPU cards in one system, the efficiency of Naïve is greatly improved and in turn, Naïve outperforms Strassen. For examples, in the 4-GPU and 8-GPU solutions, Naïve appears much more efficient than Strassen. The results from the high-density multi-GPU tests verified the assumption that: Naïve may surpass Strassen under the condition that the time of multiplying two matrices is approx. one order of magnitude slower than the time of adding matrices of same sizes. Currently, this condition is hardly satisfied on low-density multi-GPU configurations since the processor cores are relatively more powerful than single GPU card. However, with the capability of densely integrating accelerators, the performance gap between multiplying and adding matrices is further reduced, thus directly affecting the best practice for MM on these novel architectures. (v) Naïve is more efficient than Strassen on 4-/8-GPU solutions regardless of floating-point precision (32-bit/64-bit). Figure 8 shows the parallel activities traces (PAT) of the case for multiplying two matrices of size 48,000, partitioned as 4 × 4 tiles, on the 8-K40 m system. PAT illustrates that (i) multiplication tasks are evenly distributed on GPU cards; and (ii) Strassen needs significantly more addition tasks than Naïve. Particularly, Fig. 8 shows that, at the beginning of program, all of cores are busy with these substantial additions for Strassen. Similarly, at the finishing of program, Strassen needs more addition tasks than Naïve (Fig. 1). In the middle of program, multiple GPU cards could be more efficient for multiplication tasks, compared with traditional processor cores. In this, the results show that Naïve becomes more efficient than Strassen.



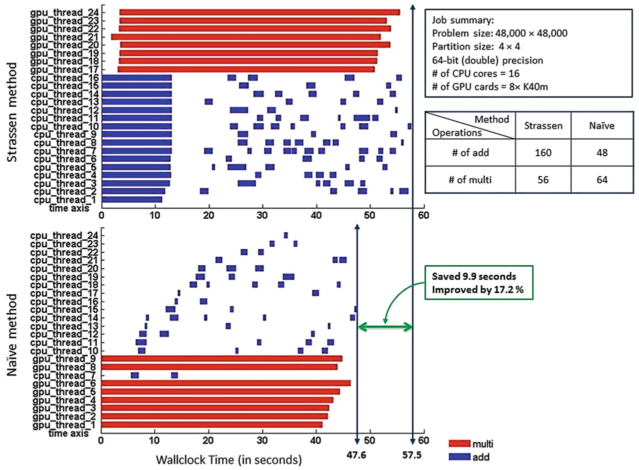
(a) Wallclock time of Naïve (b) Wallclock time of Strassen (c) Perf. Improvement of Stassen vs. Naïve

Fig. 6. Experimental results for Naïve and Strassen on 8-K40 m (double precision): (a) and (b) present the wallclock time in seconds for Naïve and Strassen, respectively. (c) shows the performance improvements for Strassen over Naïve. In the legend, 7 CORE (14 CORE) means a CPU-only solution using 7 (14) processor cores. The rest of tests are GPU solutions where 16 CPU cores used. Figures 7, 9, 10, 11 and 12 use the same legends.



(a) Wallclock time of Naïve (b) Wallclock time of Strassen (c) Perf. Improvement of Stassen vs. Naïve

Fig. 7. Experimental investigation for Naïve and Strassen on 8-K40 m (single precision).



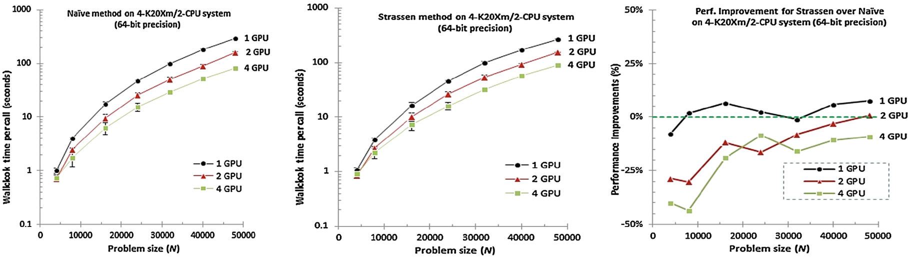
Job summary:
Problem size: 48,000 × 48,000
Partition size: 4 × 4
64-bit (double) precision
# of CPU cores = 16
# of GPU cards = 8× K40m

Method Operations	Strassen	Naïve
# of add	160	48
# of multi	56	64

Fig. 8. Parallel activities traces for Naïve and Strassen for problem size 48,000 on 8x K40 m

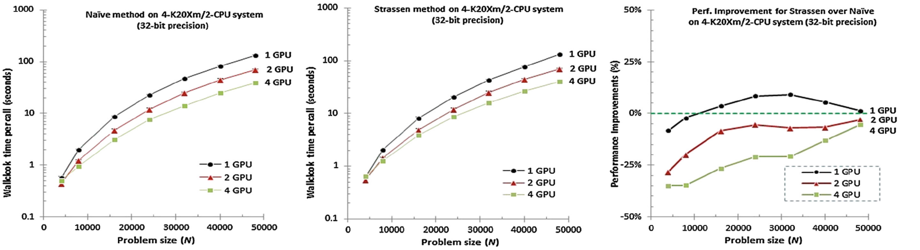
5.3.3 Strassen vs. Naïve on 4x K20Xm and 16x K80

In the literature of NVIDA GPU cards, K20Xm and K80 are the predecessor and successor of K40m. Figures 9 and 10 show the absolute performances and performance comparisons between Strassen and Naïve on 4-K20Xm, using double and single precisions, respectively. Similarly, Figs. 11 and 12 show the results on 16-K80 system. Furthermore, Fig. 13 presents the best performances of three multi-GPU platforms, which undoubtedly shows that 16-K80 is the optimal. The results on 4-K20Xm system reaffirm previous discoveries: on 1-GPU configuration, Strassen is the optimal algorithm while on 2-/4-GPU configurations, Naïve appears more efficient than Strassen. The same results appear in the 16-K80 tests.



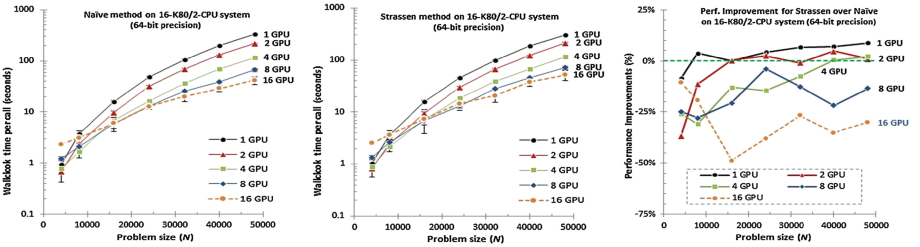
(a) Wallclock time of Naïve (b) Wallclock time of Strassen (c) Perf. Improvement of Stassen vs. Naïve

Fig. 9. Experimental investigation for Naïve and Strassen on 4-K20Xm (double precision)



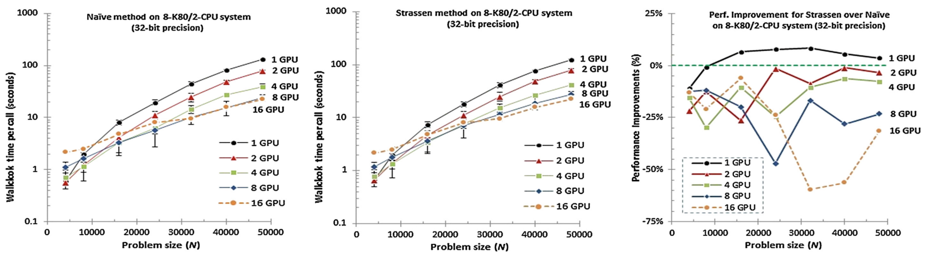
(a) Wallclock time of Naïve (b) Wallclock time of Strassen (c) Perf. Improvement of Stassen vs. Naïve

Fig. 10. Experimental investigation for Naïve and Strassen on 4-K20Xm (single precision).



(a) Wallclock time of Naïve (b) Wallclock time of Strassen (c) Perf. Improvement of Stassen vs. Naïve

Fig. 11. Experimental investigation for Naïve and Strassen on 16-K80 (double precision)



(a) Wallclock time of Naïve

(b) Wallclock time of Strassen

(c) Perf. Improvement of Stassen vs. Naïve

Fig. 12. Experimental investigation for Naïve and Strassen on 16-K80 (single precision)

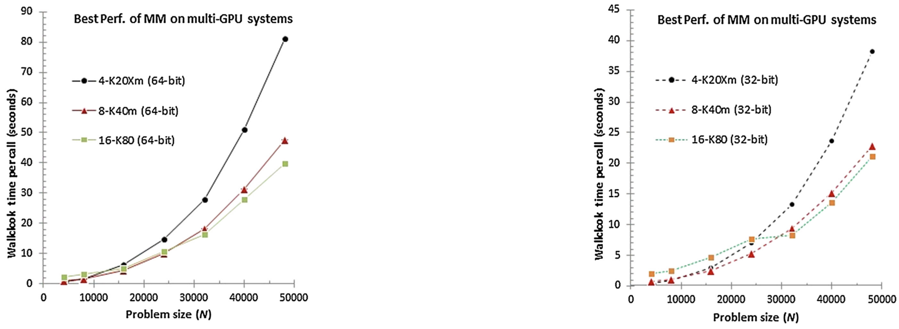


Fig. 13. Best performances of MM on multi-GPU platforms (double/single precisions: left and right plots)

6 Discussions

Through extensive experiments, we have demonstrated the big performance disparities for MM on different architectures. The greatest advantage of Strassen is that Strassen needs fewer multiplication operations than the naïve method. Thus on the classical multicore architectures and single-GPU architecture, Strassen is often more efficient than the naïve method for large enough problem size. However, on the novel high-density multi-GPU architectures, the efficiency of multiplying two matrices is significantly improved but the efficiency of adding two matrices is still constrained by the overhead of data transfer between the host and multiple accelerator devices. This made processors as the optimal platform for additions and accelerators as the optimal platform for multiplications. In this scenario, the naïve method could outperform the Strassen method. This indicates that the performance difference between the multiplication and addition operations would finally determine which method would be the best-practice solution. In this regard, the high-density multi-GPU architecture is widely different from the homogenous multi-core systems and low-density systems.

7 Conclusion

In this work, we test the standard (Naïve) and Strassen tile-based MM methods on novel high-density multi-GPU systems. Three generations of NVIDIA GPU cards, K20Xm, K40m and K80 are benchmarked on the systems. Both 64-bit double and 32-bit single precisions are tested. The results show that multi-GPU solutions can significantly improve the performances, in comparison with CPU-only solutions. The Strassen method is often beneficial on the multicore and the low-density GPU solutions; however it is beaten by the Naïve method on the high-density multi-GPU solutions. The reason is that the Strassen needs more additions than the Naïve method but GPU is not efficient enough for these additions thanks to the host-device overhead. The results in the work give a handy guide for the practitioners to use the methods for multiplying matrices on heterogeneous systems.

With the birth of new technologies, it is undoubted that the intra-chip and the inter-chip communication capability could and should be improved. By then, performance comparisons between different MM methods should be re-evaluated to find out the best-practice algorithm on novel architectures.

References

1. Robinson, S.: Toward an optimal algorithm for matrix multiplication. SIAM News **38**, 1–3 (2005)
2. Lancaster, P., Tismenetsky, M.: The Theory of Matrices: with Applications. Academic Press, Waltham (1985)
3. Dorn, F.: Dynamic programming and fast matrix multiplication. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 280–291. Springer, Heidelberg (2006)
4. Gunnels, J.A., Henry, G.M., Van De Geijn, R.A.: A Family of high-performance matrix multiplication algorithms. In: Alexandrov, V.N., Dongarra, J.J., Juliano, B.A., Renner, R.S., Kenneth Tan, C.J. (eds.) ICCS 2001. LNCS, vol. 2073, pp. 51–60. Springer, Heidelberg (2001)
5. Kurzak, J., Alvaro, W., Dongarra, J.: Optimizing matrix multiplication for a short-vector SIMD architecture–CELL processor. Parallel Comput. **35**, 138–150 (2009)
6. Strassen, V.: Gaussian elimination is not optimal. Numer. Math. **13**, 354–356 (1969)
7. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, pp. 1–6 (2004)
8. Williams, V.V.: Multiplying matrices faster than Coppersmith-Winograd. In: Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing, pp. 887–898 (2012)
9. Chou, C.C., Deng, Y.F., Li, G., Wang, Y.: Parallelizing strassens method for matrix multiplication on distributed-memory mimd architectures. Comput. Math. Appl. **30**, 49–69 (1995)
10. D'Alberto, P., Nicolau, A.: Using recursion to boost ATLAS's performance. In: Labarta, J., Joe, K., Sato, T. (eds.) ISHPC 2006 and ALPS 2006. LNCS, vol. 4759, pp. 142–151. Springer, Heidelberg (2008)

11. Ohshima, S., Kise, K., Katagiri, T., Yuba, T.: Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In: Daydé, M., Palma, J.M.L.M., Coutinho, A. L.G.A., Pacitti, E., Lopes, J.C. (eds.) VECPAR 2006. LNCS, vol. 4395, pp. 305–318. Springer, Heidelberg (2007)
12. Irony, D., Toledo, S., Tiskin, A.: Communication lower bounds for distributed-memory matrix multiplication. J. Parallel Distrib. Comput. **64**, 1017–1026 (2004)
13. Fatahalian, K., Sugerman, J., Hanrahan, P.: Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 133–137 (2004)
14. Beaumont, O., Boudet, V., Rastello, F., Robert, Y.: Matrix multiplication on heterogeneous platforms. IEEE Trans. Parallel Distrib. Syst. **12**, 1033–1051 (2001)
15. Thottethodi, M., Chatterjee, S., Lebeck, A.R.: Tuning Strassen's matrix multiplication for memory efficiency. In: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (CDROM), pp. 1–14 (1998)
16. Luo, Q., Drake, J.B.: A scalable parallel Strassen's matrix multiplication algorithm for distributed-memory computers. In: Proceedings of the 1995 ACM Symposium on Applied Computing, pp. 221–226 (1995)
17. Choi, J., Walker, D.W., Dongarra, J.J.: PUMMA: parallel universal matrix multiplication algorithms on distributed memory concurrent computers. Concurrency: Pract. Experience **6**, 543–570 (1994)
18. Zhang, P., Gao, Y., Fierson, J., Deng, Y.: Eigenanalysis-based task mapping on parallel computers with cellular networks. Math. Comput. **83**, 1727–1756 (2014)
19. Zhang, P., Powell, R., Deng, Y.: Interlacing bypass rings to torus networks for more efficient networks. IEEE Trans. Parallel Distrib. Syst. **22**, 287–295 (2011)
20. Zhang, P., Deng, Y., Feng, R., Luo, X., Wu, J.: Evaluation of various networks configurated by adding bypass or torus links. IEEE Trans. Parallel Distrib. Syst. **26**, 984–996 (2015)
21. Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., Schwartz, O.: Communication-optimal parallel algorithm for strassen's matrix multiplication. In: Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 193–204 (2012)
22. Goto, K., Geijn, R.A.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. (TOMS) **34**, 12 (2008)
23. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Orti, E.S.: Evaluation and tuning of the level 3 CUBLAS for graphics processors. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–8 (2008)
24. Demmel, J.: LAPACK: a portable linear algebra library for supercomputers. In: IEEE Control Systems Society Workshop on Computer-Aided Control System Design, pp. 1–7 (1989)
25. CS-Storm specification. (2014). http://www.cray.com/sites/default/files/CrayCS-Storm.pdf
26. Fang, Y.-C., Gao, Y., Stap, C.: Future enterprise computing looking into 2020. In: Park, J.J., Zomaya, A., Jeong, H.-Y., Obaidat, M. (eds.) Frontier and Innovation in Future Computing and Communications. LNEE, vol. 301, pp. 127–134. Springer, Heidelberg (2014)
27. Skiena, S.S.: The Algorithm Design Manual, vol. 1. Springer, Heidelberg (1998)
28. Zhang, P., Ling, L., Deng, Y.: A data-driven paradigm for mapping problems. Parallel Comput. (2015). doi: 10.1016/j.parco.2015.05.002 (In press)
29. Huss-Lederman, S., Jacobson, E.M., Johnson, J.R., Tsao, A., Turnbull, T.: Implementation of Strassen's algorithm for matrix multiplication. In: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, pp. 32–32 (1996)

http://www.springer.com/978-3-319-20118-4

High Performance Computing
30th International Conference, ISC High Performance
2015, Frankfurt, Germany, July 12-16, 2015,
Proceedings
Kunkel, J.; Ludwig, Th. (Eds.)

2015, XII, 530 p. 237 illus., Softcover
ISBN: 978-3-319-20118-4