On optimization techniques for the matrix multiplication on hybrid CPU+GPU platforms

Luis-Pedro García
Technical University of Cartagena
Email: luis.garcia@sait.upct.es

Javier Cuenca and Domingo Giménez
University of Murcia
Email: javiercm@ditec.um.es, domingo@um.es

***Abstract*—The use of auto-tuning techniques in a matrix multiplication routine for hybrid CPU+GPU platforms is analyzed. Basic models of the execution time of the hybrid routine and information obtained during its installation are used to optimize the execution time with a balanced assignation of the computation to the computing components in the heterogeneous system. Satisfactory results are obtained, with experimental execution times close to the lowest achievable.**

I. Introduction

In most scientific and engineering problems, computations are carried out using basic BLAS type matrix routines. Level 3 BLAS collects all the matrix-matrix operations, which are the set of the most computational intensive BLAS routines. The basic kernel of BLAS and of many of the scientific codes is matrix multiplication. Therefore, the improvement in the performance of scientific codes is achieved in many cases by the efficient use of this routine.

Efforts have been devoted to the optimization of linear algebra routines (and in particular of the matrix multiplication) in computational systems of different characteristics [1], [2], [3], [4], [5]. The decisions to take depend on the type of computational system for which the routines are developed. For example, it is necessary to adapt the original ideas developed for homogeneous parallel systems to heterogeneous or dynamic systems [6], [7], [8], [9], and the omnipresence today of multicore+GPU systems makes the adaptation of previous auto-tuning techniques to these heterogeneous systems compulsory.

This paper studies empirical auto-tuning techniques to achieve optimum load balance between GPU and CPU when they are performing a matrix-matrix multiplication. The CPU part can be carried out with a multithread BLAS library. Many BLAS implementations exist, for both multicore (vendors implementations: Intel MKL [10], IBM ESSL [11], etc.; or free implementations: ATLAS [2], Goto BLAS [12], etc.) and GPU (CULA Tools [13], CUBLAS [14] and MAGMA [15]). The CPU and GPU implementations used here are MKL and CUBLAS, but the same techniques can be applied with any other basic libraries, and experiments carried out with these provided similar performance results.

The rest of the paper is organized as follows. Section 2 comments on some adaptations of linear algebra software to GPU and combinations of CPU+GPU. Section 3 shows the structure of the matrix multiplication and the auto-tuning methodology. Experimental results are shown in section 4. Section 5 concludes and outlines possible research lines.

II. Linear algebra in multicore+GPU platforms

Due to the omnipresence of multicore systems with GPU accelerators, efforts are being devoted to the development of software for these systems, and especially to the design of linear algebra routines which manage the heterogeneity of the whole system to obtain maximum performance.

In [16] a strategy is presented to perform matrix-matrix multiplications on hybrid NVIDIA GPU systems. The basic idea is to carry out a matrix multiplication $A = BC$ by splitting the data of matrices B and C between the CPUs of the multicore and a single GPU, and perform the operations simultaneously on both devices. The final result is obtained by aggregation of the results independently obtained in CPU and GPU. A similar approach is used in the core numerical kernels included as part of the NVIDIA LINPACK TOP 500 benchmark suite [17] to rank the fastest heterogeneous supercomputers in the world.

The PHIGEMM [18] library extends the basic mapping presented in [16] and uses a work-load distribution based on a pre-defined split factor and the latest capabilities of CUDA to efficiently control asynchronous data transfer and overlapping multi-device computations. Users must define this split factor manually according to the ratio of computational power of the CPU and the GPU. PHIGEMM is freely available as open-source code in QEF [19].

In [20] a hybrid programming model combining MPI, OpenMP and streaming computing is described. The LAPACK task, thread and data parallelisms are exploited. The main idea to optimize the load distribution across the CPUs and GPUs is to use a two-level adaptive method, to measure the relative performance of GPUs and CPUs at runtime, and to split the workload for the next computation accordingly. Additionally, a software pipelining technique is used to bypass the low-bandwidth communication between CPU and GPU.

In [21] a variable block size auto-tunig scheme on CPU+GPU hybrid systems for the QR factorization in MAGMA is proposed. The approach is to fit the CPU and GPU cost via two independent regression models and to define a cost objective function to balance the workloads between CPU and GPU.

In this work, we use a static approach to decide the split of the matrices between the components of the heterogeneous computing system. The dynamic selection is discarded because it would suppose high overheads when the matrix multiplication is used inside higher-level codes (for example,

LU, QR or Cholesky factorizations). Experimental and model-based empirical techniques previously used for the selection of algorithmic parameters in the installation of the routine in NUMA systems [22], [23] are studied for this new computing system.

III. AUTO-TUNING A MULTI-DEVICE MATRIX MULTIPLICATION

The matrix multiplication is computed simultaneously on CPU and GPU. The multiplication $C = \alpha AB + \beta C$ can be expressed as $C = \alpha(AB_1 + AB_2) + \beta(C_1 + C_2)$, and $AB_1 + \beta C_1$ can be performed in the GPU and $AB_2 + \beta C_2$ in the CPU (Figure 1). In the experiments, the CUBLAS library (`cublasDgemm` routine) is used for the computation in the GPU, and the MKL library (`dgemm` routine) in the CPU, but the same methodology works when other basic libraries are used.

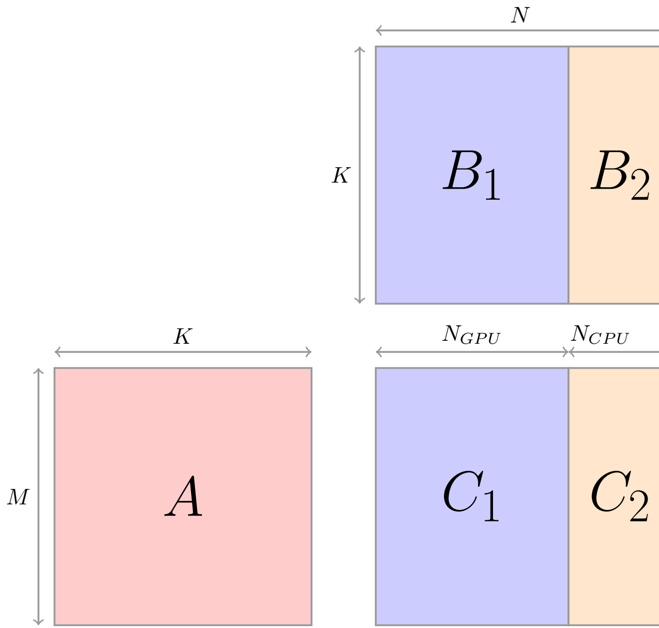


Fig. 1. Distribution of the computation of a matrix multiplication on a CPU+GPU system. The blue portion (left part) is performed in the GPU and the orange portion (right part) in the CPU.

An optimum split of the matrix would keep the time consumed by the GPU and CPU balanced [16], [24]. The multi-device (GPU and CPUs) computations are overlapped and the data transfers between GPU and CPU are performed asynchronously, which allows computation and communication overlapping, so increasing the performance. To reduce the data transfer time between CPU and GPUs, we use the pinned memory mechanism provided by CUDA.

In this work, we have considered two different auto-tuning methods to obtain a balanced distribution of the matrix between CPU and GPU according to the machine characteristics: an experimental method (guided search [22]) and a mixed theoretical-experimental method (empirical modeling [23]). In the next subsections these methods are described.

A. Method I: Guided search

The working scheme of the guided search method is shown in Figure 2. The installation of the hybrid `dgemm` routine in the system is done by executing the routine for each matrix size specified in an *Installation_Set*, varying the amount of data of matrices B and C between CPU (N_CPU) and GPU (N_GPU), with $N = N_GPU + N_CPU$. The *Installation_Set* contains significant values, from small to large sizes, so the installation gives satisfactory results for a wide range of values. For low installation time, the search begins with the smallest problem size in the *Installation_Set* and uses a value 0 for N_CPU. Then the value of N_CPU is increased (and the value of N_GPU is decreased) by a predetermined amount until the execution time exceeds by a threshold the previous lowest execution time. For other problem sizes, the search starts with the value of N_CPU that provides the best execution time for the previous problem size in the *Installation_Set*. The search is made in two directions, decreasing and increasing N_CPU, and it finishes for each problem size when the execution time exceeds the minimum for that size by an amount greater than the threshold.

B. Method II: Empirical modeling of the execution time

An alternative approach consists on the empirical modeling of the execution time of the hybrid routine. The working scheme of this method is shown in Figure 3. A theoretical model of the execution time is used to determine the optimal split of the matrices. There are two routines for which the model of the execution time must be obtained: the matrix multiplication on CPU and the matrix multiplication on GPU. Considering only square matrices of size $m \times m$ for simplicity, the time to run a matrix multiplication with the hybrid `dgemm` routine can be written $T_{dgemm}(m, n) = k_1 m^2 n + k_2 m^2 + k_3 m$, where n takes the value corresponding to the amount of data of matrices B and C between CPU ($n = n_{cpu}$) and GPU ($n = n_{gpu}$). The value of the coefficients k_i may be obtained with least-square. We obtain a set of values of the coefficients k_i for the multiplication in CPU (routine Intel MKL `dgemm`) and another set for the multiplication in GPU (routine CUDA `cublasDgemm`). In this way, the models of T_{dgemm_cpu} and T_{dgemm_gpu} are

$$T_{dgemm_gpu}(m, n) = k_{1_gpu}\ m^2\ n + k_{2_gpu}\ m^2 + k_{3_gpu}\ m \quad (1)$$

$$T_{dgemm_cpu}(m, n) = k_{1_cpu}\ m^2\ n + k_{2_cpu}\ m^2 + k_{3_cpu}\ m \quad (2)$$

On the other hand, since the GPU contains its own memory, before the execution of the GPU kernel the input data must be copied from the CPU memory to the GPU memory. Likewise, when the kernel completes its execution, the output data must be copied from the GPU memory to the CPU memory. Therefore, in the execution model of the hybrid `dgemm` routine it is necessary to consider the cost of the transfers between GPU and CPU memory. As discussed in [25], data transfers

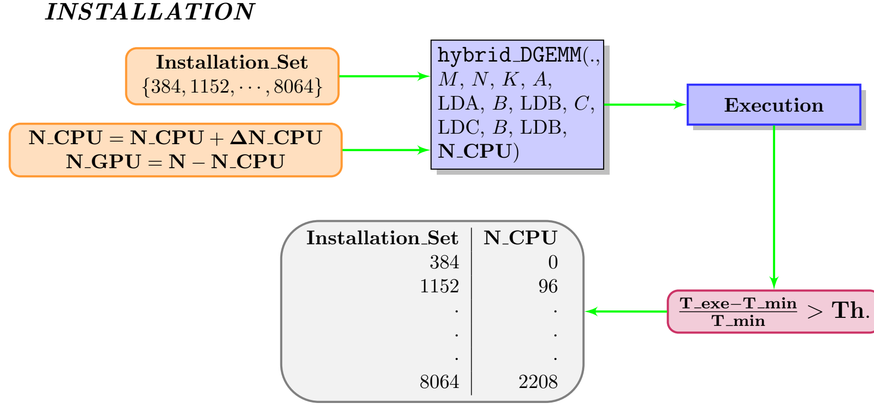


Fig. 2. General scheme of the guided search method.

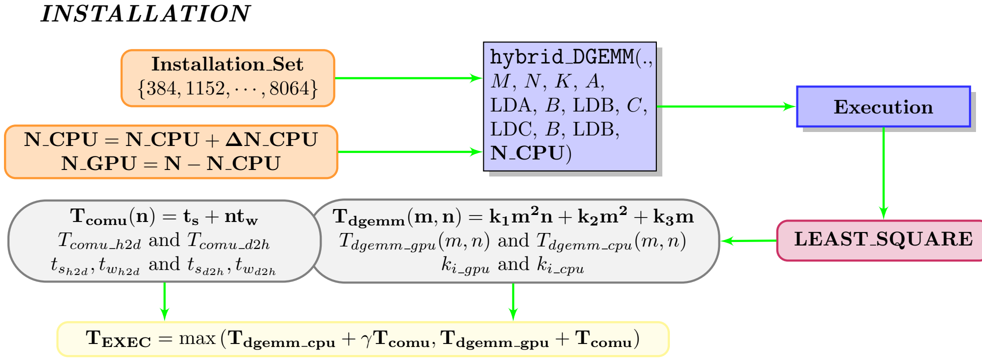


Fig. 3. General scheme of the empirical modeling method.

between the CPU and GPU when pinned memory is used can be modeled with a fixed overhead representing the latency of sending the first byte, t_s, plus the time required to send each subsequent byte, t_w. Therefore, the time to transfer n bytes can be written as $T_{comu}(n) = t_s + nt_w$, in the same way to the traditional message-passing paradigm. Since the hybrid dgemm routine copies to device memory (GPU) the entire matrix A of size $m \times m$ and the panel of matrix B of size $m \times n_{gpu}$, and the panel of C of size $m \times n_{gpu}$ is copied back to the host memory (CPU), the cost of the transfers between CPU and GPU can be written

$$T_{comu} = t_{s_{h2d}} + m^2\ t_{w_{h2d}} + t_{s_{h2d}} + m\ n_{gpu}\ t_{w_{h2d}} + t_{s_{d2h}} + m\ n_{gpu}\ t_{w_{d2h}} \quad (3)$$

where $h2d$ and $d2h$ indicate the direction of transfer (host to device, $h2d$, or device to host, $d2h$). Empirically, the values for t_s and t_w are different depending on the direction in which data are transferred. As mentioned, in the hybrid dgemm routine the CPU work is overlapped with the GPU work and the data transfers between GPU and CPU are asynchronous in order to improve performance. So, the routine can be modeled as

$$T_{exec} = \max\left(T_{dgemm_cpu}, T_{dgemm_gpu} + T_{comu}\right) \quad (4)$$

if the CPU work is overlapped with work on the GPU and the data transfers, and as

$$T_{exec} = \max\left(T_{dgemm_cpu}, T_{dgemm_gpu}\right) + T_{comu} \quad (5)$$

if the CPU work only overlaps with work on the GPU but not with data transfers. Finally, these two models can be combined, obtaining a general model for any platform:

$$T_{exec} = \max\left(T_{dgemm_cpu} + \gamma\ T_{comu}, T_{dgemm_gpu} + T_{comu}\right) \quad (6)$$

where the value of the coefficient γ for a particular system is obtained experimentally. A value 0 (corresponding to the model in equation 4) or 1 (equation 5) could be obtained, as can values between 0 and 1, which represent partial overlapping of computation on CPU with data transfers of CPU to and from GPU. The values of T_{dgemm_cpu}, T_{dgemm_gpu} and T_{comu} are different for equations 4, 5 and 6, and, once the model corresponding to our system (the value for γ) has been determined, the work distribution for CPU and GPU which minimizes T_{exec} in equation 6 is obtained.

IV. Experiments for the matrix multiplication on multicore CPU+GPU

Experiments were carried out on two platforms:

- 12CK20 is a shared-memory system with two hexa-cores (12 cores) Intel Xeon E5-2620 with a GPU device Tesla K20c (based on Kepler Architecture) with 4800 Mbytes of Global Memory and 2496 CUDA Cores (13 Streaming Multiprocessors, with 192 Streaming Processors per SM).
- 6CGTX690 is a hexa-core AMD 1075T with a GPU device GeForce GTX 590 with 1536 Mbytes of Global Memory and 512 CUDA cores (16 Streaming Multi-processors, with 32 Streaming Processors per SM).

A. Method I: Guided search

1) Guided search: installation: The *Installation_Set* was $\{384, 1152, 1920, \ldots, 9600, 10368, 11136\}$ for 12CK20 and $\{384, 1152, 1920, \ldots, 6528, 7296, 8064\}$ for 6CGTX690. The value of N_CPU is initially 0, and is then increased by 16. Figure 4 shows, for thresholds 2%, 5% and 10% and for the matrix sizes in the *Installation_Set*, the value for N_CPU (in %) with which the highest values of GFLOPS are obtained on 12CK20 and 6CGTX690. The behavior in the two systems is different, with a greater increase in the percentage of N_CPU in 6CGTX690 than in 12CK20, which is due to differences in the relative speeds of CPU and GPU. Furthermore, an unexpected behavior is observed on 12CK20, where for matrix sizes between 8064 and 10368 there is a decrease in the percentage of N_CPU, probably because MKL `dgemm` is not able to handle pinned memory efficiently for some matrix sizes.

This small difference in the partition size produces small differences in the GFLOPS obtained when using the three thresholds. The GFLOPS for different matrix sizes are shown in Figure 5. The highest values are always obtained with a threshold of 10%, but there are no significant differences with the other thresholds considered, and the 5% threshold gives similar results to those with a threshold of 10% (in 6CGTX690 the values coincide, and in 12CK20 there are differences in only 2 of the 15 sizes).

The installation times with the three thresholds in the two computational systems experimented with are shown in Table I. The installation time increases with the threshold, and is larger in 6CGTX690, which has slower computational components. In the two systems the installation times are affordable, and there are no significant differences in the installation time or in the prediction when the threshold varies.

Threshold	12CK20	6CGTX690
2%	92	319
5%	98	370
10%	229	489

TABLE I. INSTALLATION TIME (IN SECONDS) WITH DIFFERENT THRESHOLDS IN TWO COMPUTATIONAL SYSTEMS.

2) Guided search: validation: Different problem sizes are used for validation (*Validation_Set*). We recall that, given a problem to solve of size n, the value for N_CPU is selected by applying an interpolation process to the closest information stored during the installation phase (*Installation_Set*). Tables II (12CK20) and III (6CGTX690) show the results obtained with guided search and the different thresholds considered. The column “OPTIMUM” shows the highest values of GFLOPS

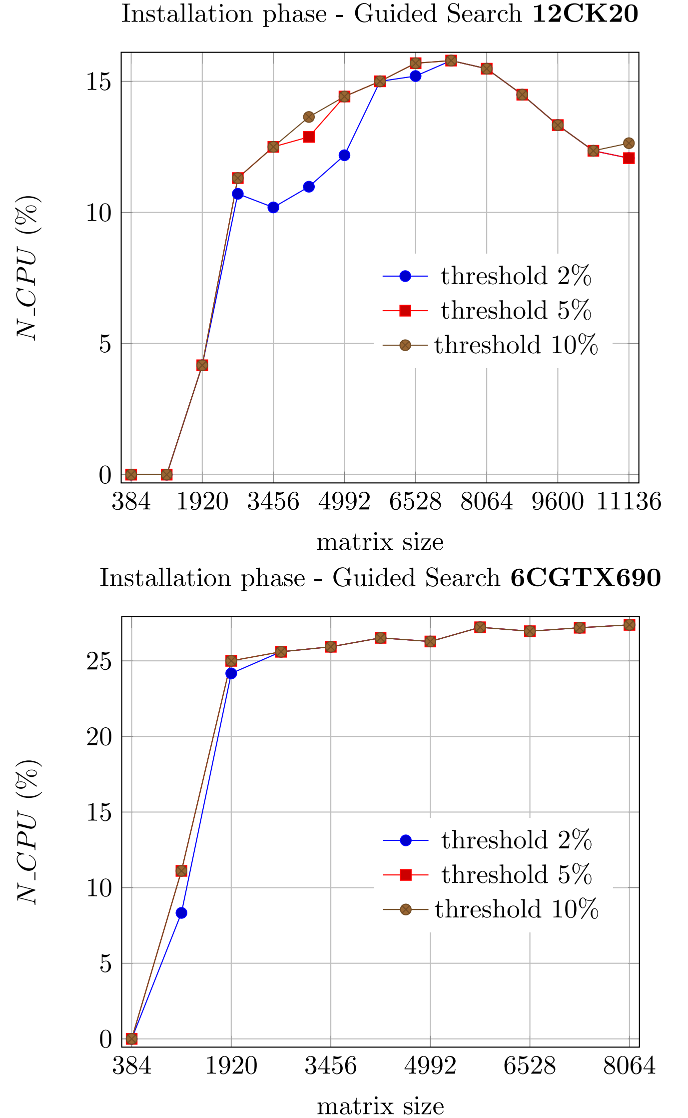
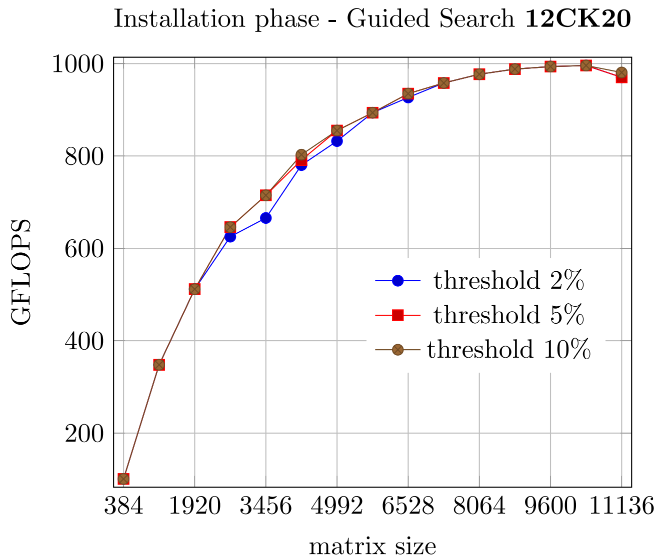
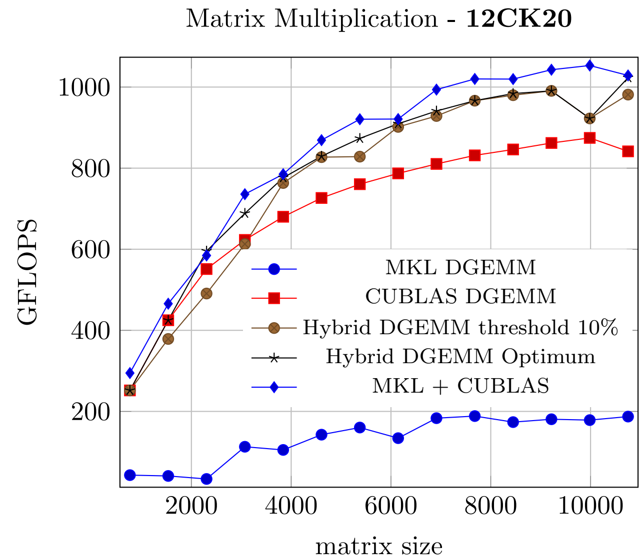
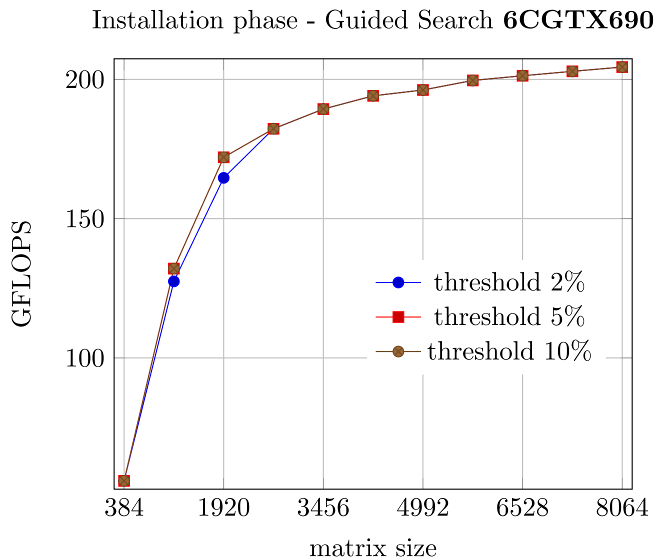


Fig. 4. Comparison of the N_CPU (in %) with which the highest values of GFLOPS are obtained with thresholds of 2%, 5% and 10% for the hybrid `dgemm` routine. In 12CK20 (top) and 6CGTX690 (bottom).

experimentally obtained and the values of N_CPU with which they are achieved. The partition given by the installation with the different thresholds is close to the optimum partition experimentally obtained, and such a small difference can be due to variations in the experiments. On 12CK20 the guided search with a threshold of 10 % gives an average deviation of 3% with respect to the optimum. With a threshold of 5% the average deviation is 3.3%, while on 6CGTX690 the average deviation for 10% and 5% is 1.3%. Finally, with a stopping criterion of 2% the average deviation increases to 3.6% in 12CK20 and to 1.4% in 6CGTX690, which are not significant values for the deviation, and the results are also satisfactory. On 12CK20 the average value for the GFLOPS achieved for the hybrid `dgemm` routine with a threshold of 2% is 767.38, with a threshold of 5% it is 770.03 and with a threshold of 10 % it is 773.18. On 6CGTX690 the average value for the GFLOPS achieved for the hybrid `dgemm` routine with a threshold of 2% is 181.39, and with a threshold of 5% and 10% it is 181.44.







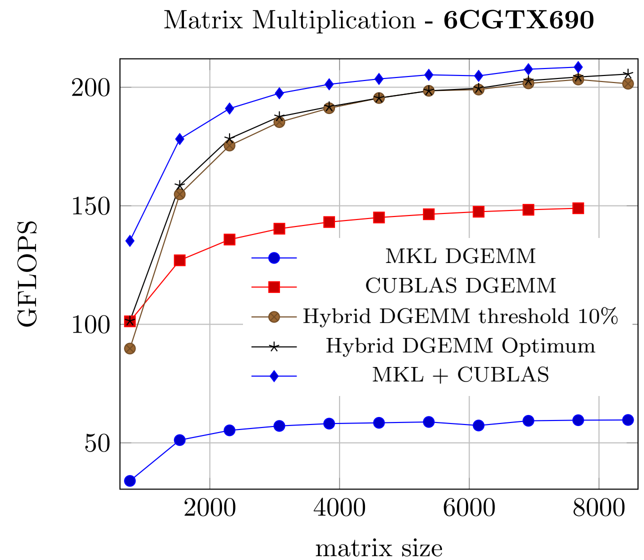


Fig. 5. Comparison of GFLOPS obtained with the value of N_CPU selected with thresholds of 2%, 5% and 10% for the hybrid `dgemm`. In 12CK20 (top) and 6CGTX690 (bottom).

Fig. 6. Comparison of the GFLOPS of the hybrid `dgemm` routine with those achieved with the CUBLAS and MKL routines, in 12CK20 (top) and 6CGTX690 (bottom).

So, the differences for the three thresholds are minimal in both systems, and satisfactory results are achieved with the guided search, regardless of the threshold used, with improved installation time for small thresholds.

Figure 6 compares the GFLOPS achieved with the hybrid `dgemm` routine with the CUBLAS and MKL versions. For the CUBLAS `dgemm` the cost of the asynchronous copies between host and device is taken into consideration. The figure illustrates that the hybrid `dgemm` routine with the guided selection of N_CPU with, for example, a threshold of 10% (**Hybrid DGEMM threshold 10%**), outperforms CUBLAS `dgemm`.

On 12CK20 the average GFLOPS achieved with the CUBLAS and the MKL routines are 705.09 and 133.12, respectively. On 6CGTX690 the average GFLOPS are 138.37 (CUBLAS) and 55.34 (MKL). Indeed, the hybrid `dgemm` routine with a threshold of 10% obtains an improvement of 10% in the GFLOPS on 12CK20 and an improvement of 31% on 6CGTX690 over the CUBLAS routine. This improvement is similar to that obtained with the optimum distribution (**Hybrid DGEMM Optimum**), and very close to the addition of GPLOPS that can be obtained ideally working with MKL `dgemm` and CUBLAS `dgemm` separately (**MKL+CUBLAS**). There is a significant difference in the relative performance of CUBLAS and MKL in the two systems, which is due to the relative speed of CPU and GPU and to the number of cores in the system; so, in systems with more cores or with another speed ratio the performance would be different, but the installation technique works in the same way.

On the other hand, if the simplest method to split the matrix is used, that is, if the partition of matrices between CPU and the GPU is implemented with a strategy based on the average GFLOPS achieved with CUBLAS and MKL, the performance improvement is worse. For example, in the system 6CGTX690

	2%		5%		10%		OPTIMUM	
n	n_cpu	GFLOPS	n_cpu	GFLOPS	n_cpu	GFLOPS	n_cpu	GFLOPS
768	0	251.78	0	251.78	0	251.78	0	251.78
1536	40	378.74	40	378.74	40	378.74	0	424.89
2304	184	505.72	192	490.78	192	490.78	240	595.42
3072	320	592.51	368	612.77	368	612.77	336	688.49
3840	408	749.46	488	750.84	504	763.37	512	776.42
4608	536	807.22	632	819.57	648	827.17	640	829.48
5376	736	820.57	792	828.41	792	828.41	800	873.33
6144	928	879.27	944	901.75	944	901.75	960	909.48
6912	1072	940.93	1088	928.64	1088	928.64	1072	940.93
7680	1200	966.40	1200	966.40	1200	966.40	1200	966.40
8448	1264	979.93	1264	979.93	1264	979.93	1280	983.98
9216	1280	990.65	1280	990.65	1280	990.65	1280	990.65
9984	1280	922.64	1280	922.64	1280	922.64	1280	922.64
10572	1297	957.47	1297	957.47	1314	981.42	1552	1023.00
11520	1344	997.14	1344	997.14	1408	995.54	1392	1005.17

TABLE II. COMPARISON OF THE HIGHEST EXPERIMENTAL GFLOPS, AND N_CPU WITH WHICH THEY ARE OBTAINED, WITH THOSE OBTAINED WITH THRESHOLDS 2%, 5% AND 10% FOR THE HYBRID DGEMM ROUTINE. IN 12CK20.

	2%		5%		10%		OPTIMUM	
n	n_cpu	GFLOPS	n_cpu	GFLOPS	n_cpu	GFLOPS	n_cpu	GFLOPS
768	64	89.77	64	89.71	64	89.77	16	101.46
1536	280	151.35	304	154.85	304	154.85	336	158.55
2304	576	178.33	584	175.41	584	175.41	576	178.33
3072	792	185.29	792	185.29	792	185.29	800	187.60
3840	1008	191.16	1008	191.16	1008	191.16	1024	191.80
4608	1216	195.45	1216	195.45	1216	195.45	1216	195.45
5376	1440	198.50	1440	198.50	1440	198.50	1440	198.50
6144	1664	199.02	1644	199.02	1644	199.02	1632	199.56
6912	1872	201.64	1872	201.64	1872	201.64	1888	202.78
7680	2096	203.26	2096	203.26	2096	203.26	2112	204.33
8448	2208	201.49	2208	201.49	2208	201.49	2336	205.56

TABLE III. COMPARISON OF THE HIGHEST EXPERIMENTAL GFLOPS, AND N_CPU WITH WHICH THEY ARE OBTAINED, WITH THOSE OBTAINED WITH THRESHOLDS 2%, 5% AND 10% FOR THE HYBRID DGEMM ROUTINE. IN 6CGTX690.

and for matrix sizes $\{384, 1152, 1920, 2688, 3456, 4224\}$ the average GFLOPS achieved are 126.88, and with the guided search the average GFLOPS achieved are 154.27. On 12CK20 the average GFLOPS achieved are 660.15 while with the guided search 773.16 GFLOPS are obtained.

Our method is a 31% improvement on 6CGTX690 and a 10% improvement on 12CK20 compared with the `dgemm` routine of CUBLAS, and the improvement is 21% on 6CGTX690 and 17% on 12CK20 compared with a selection based on the average GFLOPS.

As conclusion, the results with the guided search method are similar to the optimum obtained experimentally, and close to the ideal performance, considering the aggregation of the individual performances of MKL and CUBLAS.

B. Method II: Empirical modeling

1) Empirical modeling: installation: When the hybrid `dgemm` routine is installed in a specific platform, the values for the different t_s, t_w and the values of the coefficients k_i are experimentally obtained. To determine the value of t_s and t_w, we measure the transfer time by a simple benchmark that invokes the CUDA routines `cublasSetMatrixAsync` and `cublasGetMatrixAsync` for different sizes of data to copy. Then the values for $t_{s_{h2d}}$, $t_{w_{h2d}}$, $t_{s_{d2h}}$ and $t_{w_{d2h}}$ are estimated by a linear regression. Similarly, the coefficients k_i are estimated by least-square using the experimental results of simple benchmarks for the basic operations `dgemm` and `cublasDgemm` with previously specified data in an *Installation_Set*. Furthermore, the benchmarks obtain the running times of the basic operations with the data storage and access scheme used in the hybrid `dgemm` routine.

Experiments are guided in a similar way as in the guided search. Experiments begin with the smallest problem size in the *Installation_Set* and use a value 0 for N_CPU. Then the value of N_CPU is increased (and the value of N_GPU is decreased) by a predetermined amount until the execution time exceeds by a threshold the previous lowest execution time. In this way, the time of experimentation is reduced, and the experiments to estimate the values for the parameters are carried out with values close to the optimum, and consequently the values obtained represent better the behavior of the routine for the values with which it will be used.

Empirically, the model in equation 5 best predicts the time cost for the computational system 12CK20, that is, in the summarized model in equation 6, the value for γ is 1. The reason is that the CPU is not idle during the copy of matrices A and B from CPU to GPU. The average deviation between the modeled time and the measured time for the hybrid `dgemm` routine ranges from 4.14%, for medium and large matrix sizes, to 11.44% for small matrix sizes.

2) Empirical modeling: validation: The model of the hybrid `dgemm` routine should provide information of the value of N_CPU to use according to the problem size and on the size and relative speed (CPU/GPU performance) of the computational system. Once the routine has been installed, the model and the possible values for the N_CPU are stored. At execution time, the value of N_CPU with which the lowest time is obtained for each problem size is selected by using the information provided by the model. The possible values of

N_CPU are substituted in the model, and the one providing the lowest modeled time is used in the solution of the problem.

Table IV shows, for different matrix sizes, n, in a *Validation_Set*, the execution time (in seconds) obtained for the hybrid `dgemm` routine with optimum selection of N_CPU and the selection provided by the empirical model. The column "Deviation" shows the deviation with respect to the optimum execution time. The value of N_CPU is well predicted only in 3 of the 15 cases, but it has not a great influence on the mean of the relative deviation from the optimum, with a value of approximately 4% in 12CK20. The average value for the GFLOPS achieved with the selection provided by the empirical model is 785.12, better than guided search, which makes empirical modeling the preferred installation technique in 12CK20.

	Model			OPTIMUM		Deviation
n	n_cpu	time	GFLOPS	n_cpu	time	(%)
768	0	0.0036	251.78	0	0.0036	0.00
1536	48	0.0199	364.38	0	0.0171	16.61
2304	224	0.0424	577.29	240	0.0411	3.14
3072	384	0.0846	685.37	336	0.0842	0.46
3840	512	0.1459	776.42	512	0.1459	0.00
4608	640	0.2359	829.48	640	0.2359	0.00
5376	768	0.3562	872.47	800	0.3558	0.10
6144	896	0.5110	907.83	960	0.5100	0.18
6912	1008	0.7093	931.10	1072	0.7019	1.06
7680	1136	0.9618	941.99	1200	0.9375	2.59
8448	1264	1.2305	979.93	1280	1.2255	0.41
9216	1376	1.9682	795.41	1280	1.5803	24.55
9984	1504	2.1745	915.33	1280	2.1573	0.80
10572	1616	2.3111	1022.55	1552	2.3101	0.04
11520	1744	3.3041	925.40	1392	3.0419	8.62

TABLE IV. COMPARISON OF THE TIME (IN SECONDS) OBTAINED FOR THE HYBRID DGEMM ROUTINE WITH THE VALUE OF N_CPU SELECTED WITH THE EMPIRICAL MODEL, THE OPTIMUM EXPERIMENTAL TIME AND N_CPU WITH WHICH IT IS OBTAINED. IN 12CK20.

V. CONCLUSIONS AND FUTURE RESEARCH

Two load-balancing methods for the matrix multiplication in hybrid multicore+GPU systems are compared. One method is based on experimental analysis of the behavior of the routine. Experiments are conducted when the routine is being installed in a system, with experiments for a number of problem sizes and varying the size of the matrices partition. The search of the preferred partition is guided in order to reduce the installation time. The other method is based on a theoretical model of the execution time of the routine, with parameters representing the relative speed of the computational components in the system and the cost and overlapping of the communications. The values of the parameters are estimated in the installation through experiments with selected matrix and partition sizes, and the model is then used at running time to decide the matrices partition. The two methods provide partitions close to the optimum for the two systems where experiments were conducted and, consequently, the execution time is close the the maximum achievable with a perfect combination of the basic libraries in use for CPU and GPU, and improvements over the use of optimized libraries for CPU or GPU are obtained. Figure 7 compares the results with the two optimization methods with those with optimized routines in only one computational component and with the added performance of the two libraries.

At present, we are working on extending the methodology

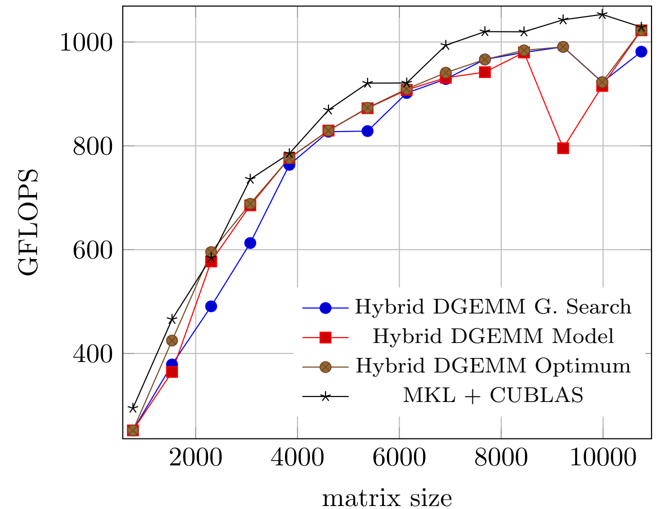


Fig. 7. Comparison of the GFLOPS of the hybrid `dgemm` routine when using guided search and empirical modeling with those achieved with the CUBLAS and MKL routines and with the added performance of the two libraries, in 12CK20.

to higher-level routines, for example, LU, QR and Cholesky factorizations. The use of the information generated for the matrix multiplication when it is used inside a routine is being analyzed, as is the direct application of the methodology to higher-level routines. The same techniques are being analyzed for the Intel Many-Integrated-Core Xeon Phi, and the work can be extended to more heterogeneous systems: multiGPU, multiMIC, hybrid CPU+GPU+MIC systems, and, in general, clusters with those components.

ACKNOWLEDGEMENTS

This work was supported by the Spanish MINECO, as well as European Commission FEDER funds, under grant TIN2012-38341-C04-03.

REFERENCES

[1] J. Bilmes, K. Asanovic, C. W. Chin, and J. Demmel, "Optimizing Matrix Multiply using PHIPAC: a Portable, High- Performance, ANSI C Coding Methodology," in *Proceedings of the International Conference on Supercomputing, ACM*, 1997, pp. 340–347.

[2] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, 2001.

[3] T. Katagiri, K. Kise, H. Honda, and T. Yuba, "Fiber: A generalized framework for auto-tuning software," *Springer LNCS*, vol. 2858, pp. 146–159, 2003.

[4] S. Hunold and T. Rauber, "Automatic tuning of PDGEMM towards optimal performance," in *11th International Euro-Par Conference, Lecture Notes in Computer Science*, vol. 3648, 2005, pp. 837–846.

[5] J. Cuenca Muñoz, "Optimización automática de software paralelo de álgebra lineal," (in Spanish), Ph. D., Departamento de Ingeniería y Tecnología de los Computadores, University of Murcia, 2005.

[6] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche, "Self Adapting Software for Numerical Linear Algebra and LAPACK for Clusters," *Parallel Computing*, vol. 29, pp. 1723–1743, 2003.

[7] J. Cuenca, D. Giménez, J. González, J. Dongarra, and K. Roche, "Automatic optimisation of parallel linear algebra routines in systems with variable load," in *PDP*, 2003, pp. 409–416.

[8] J. Cuenca, L. P. García, D. Giménez, and J. Dongarra, "Processes distribution of homogeneous parallel linear algebra routines on heterogeneous clusters," in *Proc. IEEE Int. Conf. on Cluster Computing*. IEEE Computer Society, September 2005.

[9] P. Alonso, R. Reddy, and A. L. Lastovetsky, "Experimental study of six different implementations of parallel matrix multiplication on heterogeneous computational clusters of multicore processors," in *PDP*, 2010, pp. 263–270.

[10] Intel MKL web page, `http://software.intel.com/en-us/intel-mkl/`.

[11] IBM ESSL web page, `http://www-03.ibm.com/systems/software/essl/index.html`.

[12] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, 2008.

[13] CULA GPU Accelerated Linear Algebra, `http://www.culatools.com/dense/performance/`.

[14] CUBLAS, `http://docs.nvidia.com/cuda/cublas/`.

[15] 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*, vol. 180, no. 1, 2009.

[16] M. Fatica, "Accelerating Linpack with CUDA on heterogenous clusters," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. ACM, 2009, pp. 46–51.

[17] GTC 2010 conference, `http://www.nvidia.com/content/gtc-2010/pdfs/2057_gtc2010.pdf`.

[18] F. Spiga and I. Girotto, "phiGEMM: A CPU-GPU Library for Porting Quantum ESPRESSO on Hybrid Systems," *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pp. 368–375, 2012.

[19] QE-FORGE, `http://qe-forge.org/gf/`.

[20] F. Wang, C.-Q. Yang, Y.-F. Du, J. Chen, H.-Z. Yi, and W.-X. Xu, "Optimizing LINPACK benchmark on GPU-accelerated petascale supercomputer," *Journal of Computer Science and Technology*, vol. 26, pp. 854–865, 2011.

[21] Y. Tsai, W. Wang, and R.-B. Chen, "Tuning block size for QR factorization on CPU-GPU hybrid systems," in *IEEE 6th International Symposium on Embedded Multicore Socs (MCSoC)*, 2012, pp. 205–211.

[22] J. Cámara, J. Cuenca, L.-P. García, D. Giménez, and A. M. Vidal, "Installation of linear algebra shared-memory subroutines," *Journal of Parallel Programming (admitido)*, 2013.

[23] J. Cámara, J. Cuenca, L.-P. García, and D. Giménez, "Empirical modelling of linear algebra shared-memory routines," in *ICCS*, 2013.

[24] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba, "Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment," in *Proceedings of the 7th international conference on High performance computing for computational science*, ser. VECPAR'06. Springer-Verlag, 2007, pp. 305–318.

[25] M. Boyer, J. Meng, and K. Kumaran, "Improving GPU performance prediction with data transfer modeling," in *IPDPS Workshops*, 2013, pp. 1097–1106.