

Tuning Basic Linear Algebra Routines for Hybrid CPU+GPU Platforms

Gregorio Bernabé¹, Javier Cuenca¹, Luis-Pedro García², and Domingo Giménez¹

¹ University of Murcia, Spain

{gbernabe,jcuenca,domingo}@um.es

² Technical University of Cartagena, Murcia, Spain

luis.garcia@sait.upct.es

Abstract

The introduction of auto-tuning techniques in linear algebra routines using hybrid combinations of multiple CPU and GPU computing resources is analyzed. Basic models of the execution time and information obtained during the installation of the routines are used to optimize the execution time with a balanced assignation of the work to the computing components in the system. The study is carried out with a basic kernel (matrix-matrix multiplication) and a higher level routine (LU factorization) using GPUs and the host multicore processor. Satisfactory results are obtained, with experimental execution times close to the lowest experimentally achievable.

Keywords: Auto-tuning, Models of the execution time, Parallel linear algebra, Hybrid CPU+GPU computing

1 Introduction

In most scientific and engineering problems, computations are carried out with 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. Therefore, the improvement in the performance of scientific codes is achieved in many cases by the efficient use of these routines.

Efforts have been devoted to the optimization of linear algebra routines in computational systems of different characteristics [3, 10, 16, 19, 24]. The decisions to take depend on the type of the 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 [2, 7, 8, 9], and the omnipresence today of multicore+GPU systems makes the adaptation of previous auto-tuning techniques to these systems compulsory.

This paper studies an empirical auto-tuning technique to achieve optimum load balance between GPUs and CPUs when they are performing linear algebra routines. The CPU part can

be carried out with a multithread BLAS library. Many BLAS implementations exist, for both multicore (vendors implementations: Intel MKL [18], IBM ESSL [17], etc.; or free implementations: ATLAS [24], Goto BLAS [14], etc.) and GPUs (CULA Tools [11] and MAGMA [1]). The CPU and GPU implementations used here are MKL and CUBLAS, but the same techniques can be applied with all other basic libraries. Our study focuses on these two libraries, since preliminary experiments carried out with others 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 introduces the auto-tuning methodology used for a basic kernel (matrix-matrix multiplication), and the experimental results for this kernel are shown in section 4. Section 5 describes how to use the basic auto-tuned kernel in a LU factorization in order to improve this higher-level routine. The experimental results with the LU factorization are shown in section 6. Finally, section 7 concludes the paper and outlines possible research directions.

2 Linear Algebra in Multicore+GPU

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 the maximum achievable performance.

In [12] 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 performing the operations simultaneously on both devices. 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 [15] to rank the fastest heterogeneous supercomputers in the world.

The PHIGEMM [21] library uses a workload 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.

In [23] 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. Additionally, a software pipelining technique is used to bypass the low-bandwidth communication between CPU and GPU.

[22] proposes a variable block size auto-tuning scheme on CPU-GPU hybrid systems for the QR factorization in MAGMA. 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 our proposal 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. Two techniques previously used for tuning linear algebra routines in NUMA systems according to the machine's characteristics can be applied for this new computing system: an experimental method (guided search [6]) and a mixed theoretical-experimental method (empirical modeling [5]). The guided search method has been used in [13], whereas, in this work, the model-based

technique is used to tune a matrix-matrix multiplication kernel which is then included inside a matrix factorization by reusing the tuned information obtained for the kernel to improve the higher level routine.

3 Auto-tuning a Multi-device Matrix Multiplication

The matrix multiplication is computed simultaneously on both GPU and CPU cores. 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. 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.

An optimum split of the matrix would keep the time consumed by the GPU and CPUs balanced [12, 20]. The multi-device (GPU and CPUs) computations are overlapped and the data transfers between GPU and CPU are performed asynchronously in order to achieve the maximum performance. To reduce the data transfer time between CPU and GPU, we use the pinned memory mechanism provided by CUDA.

There are two routines for which we need to obtain a model of the execution time: 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 in the hybrid `dgemm` routine can be written as $T_{dgemm}(m, n) = k_1 m^2 n + k_2 m^2 + k_3 m$, where n takes the value corresponding to the amount of data of matrices B and C in CPU ($n = n_{cpu}$) and GPU ($n = n_{gpu}$). The values of the coefficients k_i are obtained with the standard least-square formulation. Obviously, we obtain one set of values of the coefficients k_i for the multiplication in CPU and another set for the multiplication in GPU. In this way, we get $T_{dgemm-cpu}$ and $T_{dgemm-gpu}$:

$$T_{dgemm-gpu}(m, n) = k_{1-gpu} m^2 n + k_{2-gpu} m^2 + k_{3-gpu} m \quad (1)$$

$$T_{dgemm-cpu}(m, n) = k_{1-cpu} m^2 n + k_{2-cpu} m^2 + k_{3-cpu} m \quad (2)$$

On the other hand, since the GPU contains its own memory, before the execution of the GPU kernel the input data are copied from the CPU memory to the GPU memory, and, when the kernel completes its execution, the output data are copied from the GPU memory to the CPU memory. Therefore, it is necessary to consider the cost of the transfers between GPU and CPU. As discussed in [4], data transfers 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 $T_{comu}(n) = t_s + n t_w$, in the same way as in 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, the cost of the transfers between CPU and GPU can be written as:

$$T_{comu} = t_{sh2d} + m^2 t_{wh2d} + t_{sh2d} + m n_{gpu} t_{wh2d} + t_{sd2h} + m n_{gpu} t_{wd2h} \quad (3)$$

where $h2d$ and $d2h$ indicate the direction of transfer (host to device, $h2d$, or device to host, $d2h$). It has been empirically tested that the values for t_s and t_w are different depending on the direction in which data are transferred. As mentioned earlier, in the hybrid `dgemm` routine

the CPU work overlaps with the work on the GPU and the data transfers between GPU and CPU are asynchronous in order to achieve the maximum performance. So, the routine can be modeled as:

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

if the CPU work overlaps with work on the GPU and the data transfers, and as:

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

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

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

where the value of the coefficient γ may be obtained experimentally. The value of this coefficient can be 0, corresponding to the model of equation 4, or 1, corresponding to the model of equation 5. The γ value could also be between 0 and 1, representing a partial overlap of CPU computation and data transfer between CPU and GPU.

4 Experiments for the Matrix Multiplication on Multi-core CPU+GPU

Experiments were carried out in two platforms:

- 12CK20 is a shared-memory system with two hexa-cores (12 cores) Intel Xeon E5-2620 and a GPU device Tesla K20c (based on Kepler Architecture) with 4800 MBytes of Global Memory and 2496 cores (13 Streaming Multiprocessors and 192 Streaming Processors).
- 12CC2075 is a shared-memory system with two hexa-cores (12 cores) Intel Xeon E5-2620 and a GPU device Nvidia Fermi Tesla C2075 with 5375 MBytes of Global Memory and 448 cores (14 Streaming Multiprocessors and 32 Streaming Processors).

In these platforms similar empirical behavior of the proposed auto-tuning method for the matrix multiplication has been obtained. Below, a summary of the most significant results is shown.

4.1 Installation

When the hybrid **dgemm** routine is installed in a specific platform the value for the different t_s , t_w and 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 number of data, and the values for t_{sh2d} , t_{wh2d} , t_{sd2h} and t_{wd2h} 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** over a previously specified data in an *Installation.Set*. So, the benchmarks obtain the running times of the basic operations with the data storage and access scheme used in the hybrid **dgemm** routine.

The search begins using the model to estimate the execution time with the smallest problem size in the *Installation.Set* and uses a value 0 for N_CPU . Then the value of N_CPU is

		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 1: Comparison of the time obtained for the hybrid **dgemm** routine with the value of N_CPU selected with the experimental model, the optimum experimental time and N_CPU with which is obtained. In 12CK20, times in seconds.

increased (and the value of N_GPU is decreased) by a predetermined amount until the modelled execution time exceeds by a threshold the previous lowest modeled execution time.

It has been empirically tested that 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 size, to 11.44%, for small matrix sizes.

4.2 Validation

The model of the hybrid **dgemm** routine should provide information of the value of N_CPU to use depending on the problem size and on the size of the computational system (CPU/GPU performance). 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 a problem size is selected by using the information provided by the model.

Table 1 shows, for different matrix size, 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 15 cases, but the N_CPU value selected by the model is always very close to the optimum and, so, it has not a great influence on the mean of the relative deviation from the optimum times, with a value of approximately 4% in 12CK20.

5 Auto-tuning a Multi-device LU Factorization by Blocks

In this section, the application of the methodology to a higher level routine that uses an auto-tuned multi-device kernel is described. An LU factorization is used to illustrate the methodology, but the same technique can be applied with other higher level routines. The implemen-

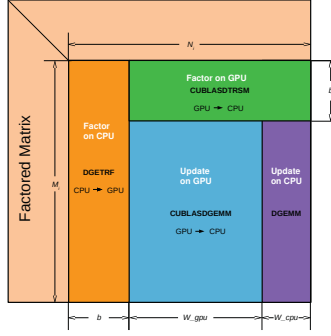


Figure 1: A schema for the multi-device LU factorization.

tation of this multi-device LU is straightforward and has the same scheme as the LAPACK right-looking block LU algorithm (routine `dgetrf`), where the matrix is partitioned into blocks of size b (called blocking factor). In the LAPACK algorithm, initially a panel (set of b columns) is factorized by the CPU kernel `dgetf2` and a permutation vector is generated according to the pivot selection: in each step the top b rows of the trailing submatrix are processed for the application of a triangular solve with multiple right-hand-sides (CPU kernel `dtrsm`). Finally, the trailing submatrix without the top b rows is updated with the application of a matrix multiplication of the form $C = C - AB$ (CPU kernel `dgemm`), where A is the panel without the top b rows, B is the top b rows of the trailing submatrix, and C is the trailing submatrix without the top b rows. Then, the same procedure is applied again, descending the diagonal of the matrix.

In the multi-device LU implementation, the CPU kernel `dgetf2` used for the panel factorization has been replaced by the `dgetrf` CPU kernel, the triangular solve `dtrsm` CPU kernel has been replaced by the `cublasDtrsm` GPU kernel, and the matrix multiplication `dgemm` CPU kernel by a hybrid `dgemm` matrix multiplication routine, where CPU and GPU compute a matrix multiplication of the form $C = C - AB$, where the width of A and the height of B are equal to the blocking factor b . Figure 1 illustrates the multi-device LU.

Regarding the communications, in the multi-device LU implementation the whole input matrix is in the CPU memory. As the LU factorization progresses, we need to send data back and forth between the CPU memory and the GPU memory. When the panel factorization is done (CPU kernel `dgetrf`), the GPU will need the result in order to process the top b rows of the trailing submatrix and to update the trailing submatrix without the top b rows, so the factorized panel needs to be transferred from the CPU memory to the GPU memory. When the top b rows are processed (GPU kernel `cublasDtrsm`), these are transferred back to the CPU memory. Additionally, the trailing submatrix, i.e. the panel of C of size $m \times n_{gpu}$, is copied to the host memory (CPU). Then the hybrid `dgemm` routine computes the trailing submatrix. Finally, the trailing submatrix updated in GPU is sent back to the CPU memory.

At this point, the auto-tuning method based on the empirical modelling of the execution time is used to search for the best distribution of the work in the hybrid `dgemm` routine at each step of the multi-device LU factorization by blocks. Three versions of the LU are compared: a version that calls to the CPU kernels from the BLAS implementation in the Intel Math Kernel Library (cpuLU), a version that follows the same schema presented for the multi-device LU, but which calls to the `cublasDgemm` GPU kernel (gpuLU), and the multi-device LU version that calls to the hybrid matrix multiplication routine (cpugpuLU).

The hybrid `dgemm` routine at each step of the LU factorization computes a matrix multi-

plication of the form $C = C - A \times B$, where A is $m \times k$, B is $k \times n$ and C is $m \times n$ and in most cases $m = n \gg k$. The value of k corresponds to the width of the panel or blocking factor (referred to b). Therefore, the execution time of this matrix multiplication can be written as:

$$T_{dgemm-gpu}(m, n, b) = k_{1-gpu}mbn + k_{2-gpu}mb + k_{3-gpu}m \quad (7)$$

$$T_{dgemm-cpu}(m, n, b) = k_{1-cpu}mbn + k_{2-cpu}mb + k_{3-cpu}m \quad (8)$$

The values of the coefficients k_i for the multiplication on GPU and for the multiplication on CPU are obtained as described in section 3, but taking into account that now $m = n \gg b$. Using this new empirical model, the performance improvement is greater than considering $m = n = k$, as is further discussed in the experimental results section.

Furthermore, in order to obtain a balanced distribution of the work between CPU and GPU when the trailing submatrix is updated by the hybrid **dgemm** routine, the cost of the communications must be taken into account. In this case, the communications to perform will be: the copy back to the CPU memory of the top b rows, the copy to the GPU memory of the trailing submatrix, i.e., the panel of C of size $m \times n_{gpu}$, and finally, the copy back to the CPU memory of the trailing submatrix updated in GPU. Hence, the cost of the transfers between CPU and GPU can be written as:

$$T_{comu-cpu} = t_{sd2h} + bmt_{wd2h} \quad (9)$$

$$T_{comu-gpu} = t_{sh2d} + mn_{gpu}t_{wh2d} + t_{sd2h} + mn_{gpu}t_{wd2h} \quad (10)$$

In order to achieve the maximum performance, when the trailing submatrix is updated, the CPU work is overlapped with the work on the GPU and the data transfers between GPU and CPU are asynchronous (using the **cublasSetMatrixAsync** and **cublasGetMatrixAsync** routines and pinned CPU memory allocation). The model is:

$$T_{exec} = \max(T_{dgemm-cpu} + T_{comu-cpu}, T_{dgemm-gpu} + T_{comu-gpu}) \quad (11)$$

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

$$T_{exec} = \max(T_{dgemm-cpu}, T_{dgemm-gpu}) + T_{comu-cpu} + T_{comu-gpu} \quad (12)$$

if the CPU work is only overlapped with work on the GPU.

6 Experiments for the LU Factorization on Multicore CPU+GPU

The experiments have been performed on the same platforms used for matrix-matrix multiplications, obtaining similar behaviours for the different LU factorization versions. In the next subsections, a summary of the most significant results is shown.

6.1 Installation

The values of the coefficients k_i in the models for the multiplication on CPU and for the multiplication on GPU are estimated by executing **dgemm** and **cublasDgemm**, for each matrix size in an *Installation_Set*, which was $\{1500, 2500, \dots, 8500, 9500\}$. As described previously, the experiments begin with a value 0 for N_CPU and, the value of N_CPU is increased (and the

n	12CC2075						12CK20					
	cpuLU	gpuLU	GPU/CPU	cpugpuLU $m = n = k$	$m = n \gg k$		cpuLU	gpuLU	GPU/CPU	cpugpuLU $m = n = k$	$m = n \gg k$	
1000	0.0458	0.0321	0.0349	0.0352	0.0317		0.0887	0.2825	0.2912	0.2844	0.2717	
2000	0.1647	0.0893	0.0976	0.0951	0.0821		0.1246	0.3302	0.4029	0.3641	0.2922	
3000	0.4089	0.1917	0.1981	0.1885	0.1800		0.3144	0.3647	0.3705	0.3562	0.3518	
4000	0.8184	0.3870	0.3696	0.3730	0.3315		0.5983	0.4362	0.4165	0.4170	0.4084	
5000	1.2470	0.7186	0.6088	0.6718	0.5779		0.9119	0.6296	0.6610	0.5908	0.5482	
6000	1.9332	1.1083	0.9665	0.9591	0.9169		1.4328	0.7822	0.7514	0.7521	0.7342	
7000	3.0254	1.6620	1.5060	1.4251	1.3335		2.1061	1.1015	1.0183	1.0367	0.9983	
8000	4.0039	2.3873	2.1811	2.0249	1.9039		2.9567	1.4395	1.3571	1.3600	1.3115	
9000	5.4579	3.3901	2.7471	2.7708	2.6254		4.0046	1.8749	1.7729	1.7582	1.7403	
10000	7.0255	4.5811	3.7258	3.6710	3.5240		5.2981	2.4139	2.2693	2.2681	2.2477	

Table 2: Comparison of the time obtained with the three versions of the LU factorization by blocks and with different methods for the selection of the value for N_CPU . Times in seconds.

value of N_GPU is decreased) by a predetermined amount until the modeled execution time exceeds by a threshold the previous lowest modeled execution time. The value for the different t_s , t_w are the same as those used in section 4.1.

It has been empirically tested that the model in equation 11 best predict the time cost for the computational system 12CC2075. The reason is that the GPU device allows concurrent copy and kernel execution. Two copy engines are also available for the transfers between CPU and GPU.

6.2 Validation

Table 2 summarizes the experimental results with the LU factorization. The execution times (in seconds) are shown for different problem sizes in a *Validation.Set* (set of some problem sizes used to validate the model in each system). The different columns correspond to the time obtained with a version that calls to the CPU kernels from the BLAS implementation in the Intel Math Kernel Library (cpuLU), a version that follows the same schema presented for the multi-device LU, but with calls to the `cublasDgemm` GPU kernel (gpuLU), and the multi-device LU version that calls to the hybrid matrix multiplication routine (cpugpuLU) with three selection methods of N_CPU : one based on the average GFLOPS achieved with CUBLAS and MKL (GPU-CPU FLOPS), another based on the model obtained for the multiplication of square matrices ($m = n = k$) and finally, a method based on the model for the matrix multiplication used in the LU factorization ($m = n \gg k$). It is clear that the time obtained with the last model is always lower than with the other methods.

Figure 2 compares the deviation in % of the GFLOPS achieved for the multi-device LU factorization with respect to the CUBLAS implementation (gpuLU) when different strategies are used for selecting the value of N_CPU . The figure illustrates that the multi-device LU routine with the selection of N_CPU based on the model for the matrix multiplication used in the LU factorization ($m = n \gg k$) outperforms always the other methods.

In 12CC2075 the cpuLU version that uses CPUs achieved an average value for the GFLOPS of 63.54, while the gpuLU version achieved a value of 110.16. Finally, the average value for the GFLOPS achieved with the cpugpuLU was 133.02. In 12CK20 the average GFLOPS are 85.01 (cpuLU), 146.59 (gpuLU) and 159.02 (cpugpuLU). It can be appreciated that, in general, using the auto-tuning methodology to use the CPUs in conjunction with the GPUs improves the overall performance of linear algebra routines.

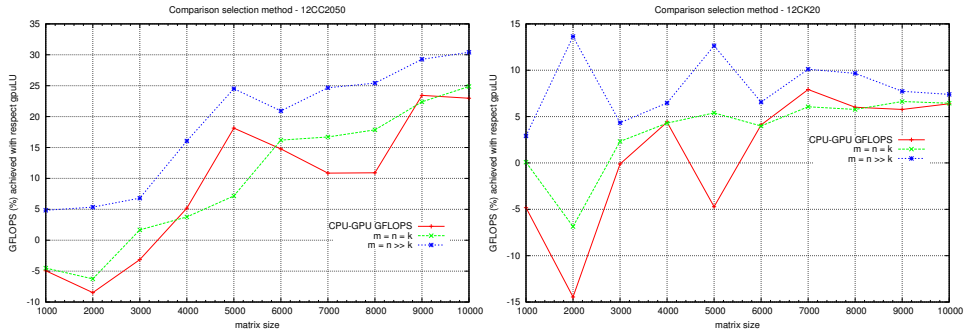


Figure 2: Deviation in % of the GFLOPS achieved for the multi-device LU factorization (cpugpuLU) with each selection method with respect to the CUBLAS implementation (gpuLU). In 12CC2075 (left) and 12CK20 (right).

7 Conclusions and Future Works

In this work, an auto-tuning method is considered to obtain balanced distributions of the work to execute linear algebra routines in CPU+GPU systems. This method is a mixed theoretical-experimental proposal based on the empirical modeling of the execution time of the routines on these hybrid platforms. It uses the model to search the best distribution of the work between the CPU and the GPU for the problem size to solve.

First, the methodology is applied to a basic kernel: a matrix-matrix multiplication. In general, the method achieves a distribution of the work very close to the optimum for any problem size. After that, the proposal is studied for a higher level routine, an LU factorization, that uses this kernel. A comparison of three versions of this routine is performed: a version that calls to a CPU kernel, a version that calls to a GPU kernel, and the multi-device LU version, which calls to the hybrid matrix multiplication routine proposed. The experimental times obtained with the multi-device version, using our proposal to balance the workload, are lower than with the other versions. Therefore, the auto-tuning methodology based on modeling the whole GPU and CPU times jointly with the inter-communication times inside CPU-GPU platforms seems to be an appropriate approach to lead to an optimum utilization of these hybrid platforms.

Nowadays, we are applying the same technique to other high level routines, like QR and Cholesky factorizations, and, also, in more complex platforms, like clusters of hybrid nodes formed with multicore CPUs and multiGPU devices.

7.1 Acknowledgements

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

References

- [1] 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.

- [2] 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*, pages 263–270, 2010.
- [3] J. Bilmes, K. Asanovic, C. W. Chin, and J. Demmel. Optimizing Matrix Multiply using PHIPAC: a Portable, High- Performance, ANSI C Coding Methodology. In *Proc. Int. Conf. on Supercomputing*, pages 340–347, 1997.
- [4] M. Boyer, J. Meng, and K. Kumaran. Improving GPU performance prediction with data transfer modeling. In *IPDPS Workshops*, pages 1097–1106, 2013.
- [5] J. Cámara, J. Cuenca, L.-P. García, and D. Giménez. Empirical modelling of linear algebra shared-memory routines. In *ICCS*, 2013.
- [6] J. Cámara, J. Cuenca, L.-P. García, D. Giménez, and A. M. Vidal. Installation of linear algebra shared-memory subroutines. *Journal of Parallel Programming (admitted)*, 2013.
- [7] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self Adapting Software for Numerical Linear Algebra and LAPACK for Clusters. *Parallel Computing*, 29:1723–1743, 2003.
- [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*, 2005.
- [9] 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*, pages 409–416, 2003.
- [10] J. Cuenca Muñoz. *Optimización Automática de Software Paralelo de Álgebra Lineal, (in Spanish)*. Ph. D. Thesis, University of Murcia, 2005.
- [11] CULA GPU Accelerated Linear Algebra. <http://www.culatools.com/dense/performance/>.
- [12] M. Fatica. Accelerating Linpack with CUDA on heterogenous clusters. In *Proc. of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 46–51, 2009.
- [13] L.-P. García, J. Cuenca, and D. Giménez. On optimization techniques for the matrix multiplication on hybrid CPU+GPU platforms. *Annals of Multicore and GPU Programming (admitted)*, 2014.
- [14] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), 2008.
- [15] GTC 2010 conference, <http://www.nvidia.com/content/gtc-2010/pdfs/2057-gtc2010.pdf>.
- [16] S. Hunold and T. Rauber. Automatic tuning of PDGEMM towards optimal performance. In *11th Int. Euro-Par Conf., LNCS*, volume 3648, pages 837–846, 2005.
- [17] IBM ESSL web page. <http://www-03.ibm.com/systems/software/essl/index.html>.
- [18] Intel MKL web page. <http://software.intel.com/en-us/intel-mkl/>.
- [19] T. Katagiri, K. Kise, H. Honda, and T. Yuba. Fiber: A generalized framework for auto-tuning software. *Springer LNCS*, 2858:146–159, 2003.
- [20] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In *VECPAR’06*, pages 305–318, 2007.
- [21] F. Spiga and I. Girotto. phiGEMM: A CPU-GPU Library for Porting Quantum ESPRESSO on Hybrid Systems. In *PDP*, pages 368–375, 2012.
- [22] Y. M. Tsai, W. Wang, and R.-B. Chen. Tuning block size for QR factorization on CPU-GPU hybrid systems. In *IEEE 6th Int. Symp. on Embedded Multicore SoCs*, pages 205–211, 2012.
- [23] Feng Wang, Can-Qun Yang, Yun-Fei Du, Juan Chen, Hui-Zhan Yi, and Wei-Xia Xu. Optimizing LINPACK benchmark on GPU-accelerated petascale supercomputer. *Journal of Computer Science and Technology*, 26:854–865, 2011.
- [24] R. Clinton Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.