# Heterogeneous Sparse Matrix Computations on Hybrid GPU/CPU Platforms

Valeria CARDELLINI [a] and Alessandro FANFARILLO [a] and Salvatore FILIPPONE [b]
[a] *Dipartimento di Ingegneria Civile e Ingegneria Informatica*
[b] *Dipartimento di Ingegneria Industriale*
*Università di Roma "Tor Vergata", Roma, Italy*

**Abstract.** Hybrid GPU/CPU clusters are becoming very popular in the scientific computing community, as attested by the number of such systems present in the Top 500 list. In this paper, we address one of the key algorithms for scientific applications: the computation of sparse matrix-vector products that lies at the heart of iterative solvers for sparse linear systems.

We detail how design patterns for sparse matrix computations enable us to easily adapt to such a heterogeneous GPU/CPU platform using several sparse matrix formats in order to achieve best performance; then, we analyze static load balancing strategies for devising a suitable data decomposition and propose our approach. We discuss our experience in using different sparse matrix formats and data partitioning algorithms with a number of computational experiments executed on three different hybrid GPU/CPU platforms.

**Keywords.** Sparse Matrix Computations, Design Patterns, CUDA, GPGPU

## Introduction

Sparse matrices and related computations are one of the centerpieces of scientific computing: most mathematical models based on the discretization of Partial Differential Equations (PDEs) require the solution of linear systems with a coefficient matrix that is large and sparse. An immense amount of research has been devoted over the years to the efficient implementation of sparse matrix computations on high performance computers. In particular, the Parallel Sparse BLAS (PSBLAS) project implements a library of Basic Linear Algebra Subroutines for parallel sparse applications that facilitates the porting of complex computations on multicomputers. The current version of PSBLAS is written in Fortran 2003 applying object-oriented (OO) techniques to achieve at the same time maximal flexibility and optimal performance [1].

In [2] we have detailed how the usage of multiple Design Patterns [3] enables an efficient handling of computations on general purpose GPU (GPGPU) devices. Here we take the State design pattern and we use it to handle computations on hybrid compute nodes hosting multiple conventional CPU cores and (possibly multiple) GPUs, demonstrating that it allows us to choose the best combination of sparse formats on CPU and GPU in order to achieve optimal performance.

As a second contribution of this paper, we consider how to distribute the workload (in our case, the matrix rows) among the processing elements using static load balancing algorithms to perform an appropriate data partitioning. To choose the ratio of GPU data to CPU data, we run two small benchmarks (sparse matrix-vector product and a GPU-CPU bandwidth test) at installation time of PSBLAS to compute out the relevant performance differences between the processing elements; similar approaches have also been proposed in [4,5,6]. The data points measured by the benchmark are then used to build a regression model in the data partitioning algorithms.

We discuss our experience in using different sparse matrix formats and data partitioning algorithms with a number of computational experiments executed on three hybrid GPU/CPU platforms having different performance characteristics. Our results show that hybrid computation is not always beneficial in terms of performance, especially when there is a high degree of heterogeneity among the CPU and GPU devices.

The paper is organized as follows. In Section 1 we describe how design patters for sparse matrix computations can help to tackle the heterogeneity of hybrid GPU/CPU platforms and review related work on hybrid computing systems. In Section 2 we discuss some issues related to sparse matrix computations on hybrid GPU/CPU platforms and describe the used benchmark. In Section 3 we present the data partitioning algorithms and in Section 4 we discuss the experimental results. Finally, we conclude in Section 5.

## 1. Software Techniques

The State design pattern is a behavioral pattern that allows the encapsulation of the object state behind an interface that allows the object type to vary at runtime [2,3]. In the context of sparse matrix computations, it provides a useful and natural solution to switch at runtime among different storage formats for a given sparse matrix. Therefore, the State pattern allows easy handling of heterogeneous computing platforms: the application making use of the computational kernels will see a uniform outer data type, but the inner data type can be adjusted according to the specific features of the processing element that the current process is running on. For instance, the code that sets up the matrix-vector product test is basically:

```
if (have_gpu(iam)) then
  amold => agpu
else
  amold => acpu
end if
call a%cscnv(info,mold=amold)
do i=1,ntimes
   call psb_spmm(done,a,xv,dzero,bv,desc_a,info)
end do
```

where the `have_gpu` function will choose, based on the process index `iam`, which processes will perform the computations on GPU and which ones on CPU cores.

A critical point in the management of a heterogeneous computing platform is how to distribute data to ensure best usage of resources. In a system with $g$ GPUs and $c > g$ cores we have $g$ cores acting as frontends to the GPUs, and $c-g$ free cores that can be used to increase the computational capacity. To get a good balance of computations we need to arrange the data in such a way that the serial parts of the computations will be aligned, i.e., all images will take the same time to execute their local matrix-vector product. Since GPUs are substantially faster, this means that the data distribution should *not* be uniform, but a proportionally larger percentage of the matrix rows should be assigned to the images running on the GPUs. This is allowed since in our PSPLAS library the number of rows to be assigned to each process is arbitrary.

How should we choose the ratio of GPU data to CPU data? A simple idea is to run a small benchmark at installation time to figure out the relevant performance differences between the two computing elements, similarly to the approach already exploited in [4, 5,6]. It is necessary to run the specific kernel under consideration, namely the sparse matrix-vector product, since its behavior is substantially different from other commonly available performance measurements, such as those from dense matrix multiply.

A parallel sparse matrix-vector product is often used in the field of the iterative solvers; for this kind of applications each iteration requires a data exchange between the processes involved in the computation. In a heterogeneous parallel platform this means, for each iteration, two transfers through the PCIe bus in order to deliver the needed data. That transfer is the bottleneck of the heterogeneous platforms and it should be reduced as much as possible.

Other works addressing hybrid computing systems include [7,8,9,10]. StarPU [7] is a runtime system that applies work-stealing to balance work among subsystems. However, StarPU requires the programmer to write separate CPU and GPU code. HDSS [8] is a two-phase scheduling and load balancing scheme for execution of loops on heterogeneous architectures; during the initial phase the scheme dynamically learns the computational power of each processor; then, it schedules the rest of the workload using a weighted self-scheduling algorithm. The work in [9] presents a programming model in which the best mapping of programs to processors and memories is determined empirically. Profiling-based approaches for hybrid computing systems include [10].

### *1.1. Sparse Matrix Formats*

Sparse matrix storage formats are essential to achieve best performance. In a hybrid environment we have an additional complication since a given format might be the best on a CPU and the worst on a GPU for the same kind of problem. This is an obvious consequence of the fact that the sparse matrix-vector product kernel performance is determined by the data movements between memory and processors, more than the actual floating-point computations. In our PSBLAS software library the default sparse format is CSR (Compressed Storage by Rows); to use GPUs we rely on an auxiliary CUDA kernel library, named SPGPU[1]. The SPGPU library is based on the ELLPACK format and a variant thereof called HLL; for these formats we have created a CPU implementation from which we derived an additional GPU-enabled version, as detailed in [2]. The GPU-enabled versions of the data formats are marked with a G in the name, for instance, HLL is the base CPU format while HLG is the GPU-enabled version. In the same vein, we also developed interfaces for the CSR and HYB formats provided by NVIDIA through the CuSparse library version 4.1. In Section 4 we investigate the performance of three sparse formats on the CPU side and four on the GPU side.

[1]`http://code.google.com/p/spgpu/`

## 2. Hybrid CPU/GPU Computations

On a heterogeneous node, equipped with CPUs and GPUs, several issues arise due to the fact that these computational units are designed for different scopes. Modern CPUs are essentially latency-oriented architectures which focus on the minimization of the execution time of a single sequential program; they use caches with ever increasing size and complex instructions that can be executed in few clock cycles. GPUs are throughput-oriented architectures based on the assumption that the workload has a high level of parallelism. This means that a GPU will use many “small” cores oriented toward a SPMD approach where instructions to be executed for each task stay the same while data change.

Furthermore, there are differences in the accuracy of the floating point operations executed on CPU and GPU due to the fact that the GPU uses by default the fused multiply-add (FMA) operation [11]. The latter computes the product of two numbers and adds that product to a number (i.e., $x \times y + z$) with only one rounding step; thus, the result will in general be different from a product followed by a sum executed with two rounding steps, being FMA more accurate than performing the operations separately. The numerical behavior of these operations is essentially equivalent for the simple matrix-vector product kernel, that is the error bounds are of the same quality, but it is not possible in general to have bit-identical results when we move from the CPU to the GPU.

The architectural difference between CPUs and GPUs implies that on a heterogeneous platform we have to tackle several issues, including the selection of a sparse matrix format and the load balancing method used to distribute the computations between the processing elements. However, the critical factor that may affect the performance is the PCIe bus, which is a bottleneck between CPU and GPU: the PCIe throughput is currently the main limiting factor of heterogeneous computations and the gap between GPU performance and communication speed of PCIe is ever increasing.

At first glance, one may suppose that the hybrid use of CPUs with GPU is always a good thing to improve performance. Unfortunately, this is not true; in fact, as shown later, processing elements that are too heterogeneous may not benefit from a hybrid computation. In our experiments we used three hybrid architectures which represent different hardware combinations; their description is in Section 4.

### *2.1. Benchmark*

To model the behaviour of a compute node we consider two factors that impact over the performance of our application: the execution time spent on CPU/GPU and the transfer time on the PCIe bus. During the installation time of PSBLAS we execute two different benchmarks that address separately the two factors. As in [6], we consider groups of CPU cores as a single computational unit; in fact, on multicore platforms, parallel processes interfere with each other through shared memory so that the speed of individual cores cannot be measured independently.

The first benchmark executes 10000 sparse matrix-vector products and returns the total elapsed time, the time spent for each iteration, and the throughput.That benchmark is executed independently on CPU (4 or 6 cores) and GPU by varying the matrix size, which is expressed in terms of number of matrix rows. The second benchmark is the *bandwidthTest* provided by the CUDA SDK; we run it twice in order to get the bandwidth from host-to-device and from device-to-host. For our investigation over the data partitioning algorithms, we consider both the cases where the first benchmark produces a lot of data points (40 data points) as well as few data points (10 data points). The data points are used to build a regression model which will be exploited by the data partitioning algorithms to predict the behavior of the computing elements.

## 3. Data Partitioning Algorithms

In this section we first analyze the linear and iterative algorithms for data partitioning which are based on different kinds of regression models, pointing out their strengths and weaknesses; we then present a hybrid solution. In all the algorithms, the PCIe bus effect is modeled by adding to the GPU computation time the time needed to transmit the data to host/device. The amount of transmitted data is estimated and depends on the largest partition of matrix rows computed by a processing element (usually the GPU).

### *3.1. Linear Algorithm*

The first algorithm we consider has a pure algebraic nature; it is based on the assumption that the time needed by the CPU and GPU to solve a problem varies linearly with respect to the problem size [5]. This assumption is true for the GPU in a lot of cases, but it does not hold at all for the CPU. Indeed, due to the cache effects, the Error Sum of Squares (SSE) of a linear regression on the CPU trend is almost 2 order of magnitude greater than the GPU model. The real strength of this algorithm is that, for very "good" CPU trends, the complexity of the algorithm is reduced to a single formula. Approximating the CPU and GPU times by a line expressed in function of the problem size, we obtain $t_1 = a_1 \cdot x_1 + b_1$ and $t_2 = a_2 \cdot x_2 + b_2$, where $x_1$ and $x_2$ are the amount of data (number of rows of the sparse matrix) assigned to the CPU and GPU, respectively; we denote this total amount of rows as $r = x_1 + x_2$. The optimal data partition is obtained when $t_1 = t_2$; with some simple algebraic manipulation, we easily obtain $x_1 = \frac{(a_2 \cdot r + b_2 - b_1)}{(a_1 + a_2)}$

The linear algorithm is very simple and works pretty well when the assumption of linear trend holds. Unfortunately, problems arise when we try to model the CPU's trend on a wide range of data, since the regression is less accurate and produces relevant errors, especially on small values of the problem size.

Unlike [5], our algorithm does not consider the chance to assign to the GPU all the entire computation in order to investigate the behavior of hybrid computations. However, as shown later, in some cases the use of a hybrid approach can be counter-productive.

### *3.2. Iterative Algorithm*

The second algorithm we consider uses an iterative approach and allows to use different models (besides lines) to represent the CPU behavior. The main idea is to keep a linear equation of the GPU trend and uses a piecewise line to represent the CPU trend. At limit, the piecewise linear function can be the entire set of data gathered during the installation phase, which uses a linear interpolation to return values which fall between two data points. The iterative algorithm produces better results than the linear one but requires to keep in memory (some) data points retrieved during the installation phase. It converges very quickly (about 20 steps) but needs the trend of the GPU time to be lower than the

CPU one. Figure 1 shows the convergence toward the optimal solution. This algorithm was proposed in [12,6] but our implementation differs since we uses the execution time of every processing element rather than its speed.

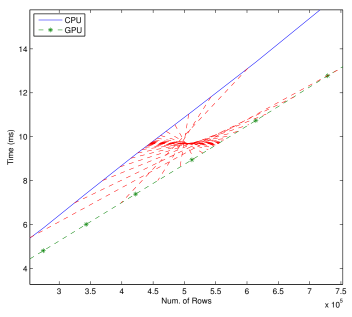


**Figure 1.** Convergence of the iterative algorithm

### *3.3. Hybrid Algorithm*

The algorithm we propose makes use of a hybrid approach to get good accuracy and require few computations during the runtime, thus combining the benefits of the two previous algorithms. The main problem of the linear algorithm is its poor accuracy related to the linear model of the CPU; on the other hand, the iterative algorithm is more accurate, but it needs to keep in memory the CPU data points and requires a certain amount of iterations that may impact negatively on the performance.

The idea of the hybrid algorithm is to use the iterative algorithm during the installation phase to find the optimal execution time and to define a regression model of the latter. The GPU is still modeled by a single line which is fairly accurate. With the optimal time values we do not need the CPU model anymore and we can get accurate results during the runtime phase just using two equations (optimal time and GPU equations).

## 4. Experimental Results

In this section we first describe the three hybrid CPU/GPU platforms we used in our experiments; then, we discuss the performance results obtained by using the different sparse matrix formats and the data partitioning algorithms over the three platforms.

### *4.1. Hybrid CPU/GPU Platforms*

Table 1 summarized the most relevant characteristics of the three hybrid CPU/GPU platforms on which we performed our tests. The AWS platform consists in a single Amazon Web Service (AWS) cluster GPU instance of type CG1; it is equipped with 2 Intel Xeon X5570, quad-core Nehalem architecture with hyperthread, plus 2 NVIDIA Tesla M2050 GPUs and 22 GB of RAM. On such platform we observed an unstable behaviour during the execution of the CPU benchmark; such instability has already been observed in [13]. The PLX platform is a single node of the PLX cluster provided by the Italian Cineca consortium, which is the largest Italian computing centre. It is equipped with 2 six-cores Intel Westmere 2.40 GHz, plus 2 NVIDIA Tesla M2070 and 48 GB of RAM. The PLX cluster is ranked at the 266th position in the Top 500 list (June 2013) and at

| **Platform** | **CPU** | **GPU** |
|---|---|---|
| AWS | Intel Xeon X5570 (quad-core) | NVIDIA Tesla M2050 |
| PLX | Intel Xeon E5645 (esa-core) | NVIDIA Tesla M2070 |
| Desktop | Intel quad-core Q6600 | NVIDIA Geforce GT 520 |

**Table 1.** Hybrid CPU/GPU platforms

the 76th position in the Green 500 list. These two platforms are the most widely used ones during the tests and show quite well how the same data partitioning algorithm can perform differently on similar architectures.

The last platform, named Desktop, represents an unusual solution in the field of scientific computing, since the performances achieved by its CPU and GPU are comparable, that is the GPU behaves almost as another quad-core socket. While all the three platforms are equipped with Fermi-based cards, the Desktop platform has the lowest performing GPU (with 48 CUDA cores), while the AWS and PLX platforms have very similar GPU cards (with 448 CUDA cores), having AWS a slightly better performing card than PLX.

In the experiments, we used the benchmark described in Section 2.1. As performance metric, we report the average execution time for matrix-vector product over 10000 runs.

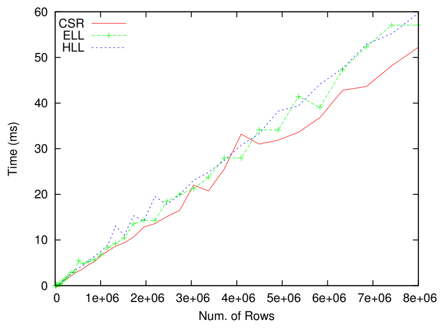
### 4.2. *Performance Analysis*

We first analyze the performance of a variety of combinations of sparse matrix formats on the most powerful AWS platform.With Figures 2(a) and 2(b) we establish the baseline performance of the various sparse matrix formats on the CPU and GPU processing elements. We observe that in our test cases the CSR format is the best one on the CPU; however, it turns out that at the same time it is the *worst* format on the GPU, whereas HLG is the best one on the GPU but not on the CPU. Given this result, in the following we will use the CSR format on the CPU and HLG format on the GPU; again, we point out that such hybrid usage is easily supported by our PSBLAS framework.

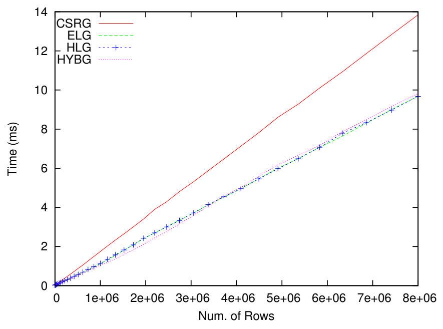
We now turn to analyze the performance of the data partitioning algorithms in distributing the workload. Figure 3(a) shows the results of the three algorithms on a typical PLX cluster node with an accurate benchmark execution having 40 data points. With a high sampling rate during the benchmark, the algorithms behavior is practically the same when executing over a stable platform, as PLX is. Figure 3(b) represents the same test case with less data points (only 10) gathered during the installation time. We see that, for a stable architecture, we do not get benefits from an accurate preliminary analysis.

However, the results change on the less stable AWS platform, as shown in Figures 4(a) and 4(b). On the latter, the iterative algorithm fits the unstable behavior of the CPU, thus producing wrong results. On the other hand, the linear algorithm is shielded from such instability and achieves better results.

The third set of experiments is on the Desktop platform, which represents the lowest performance architecture. As shown in Figure 5(a), the stability of the Desktop platform

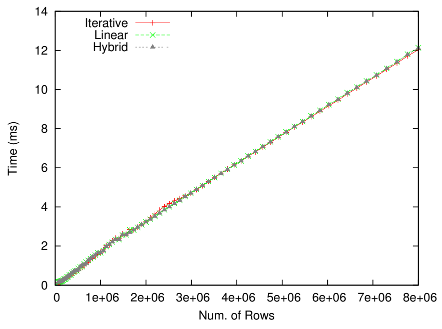


(a) CPU

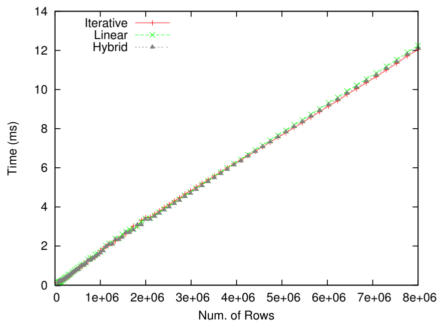


(b) GPU

**Figure 2.** Performance of sparse matrix formats on AWS platform

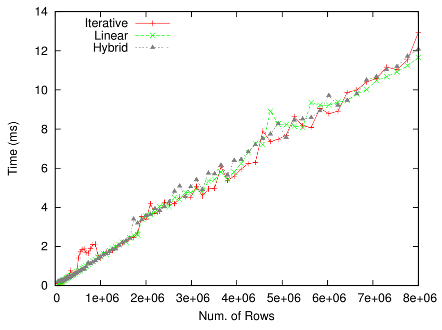


(a) 40 data points

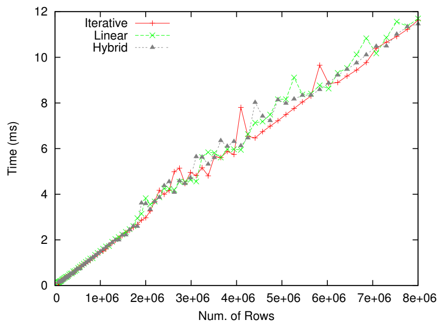


(b) 10 data points

**Figure 3.** Performance of data partitioning algorithms on PLX platform



(a) 40 data points

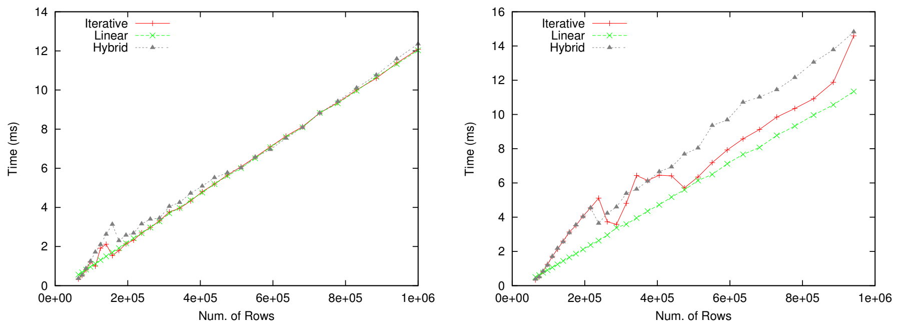


(b) 10 data points

**Figure 4.** Performance of data partitioning algorithms on AWS platform

in terms of performance variations makes the iterative and hybrid algorithms the worst. The linear algorithm does not encounter any difficulty due to the linear behavior of CPU and GPU. Even with an inaccurate sampling (see Figure 5(b)), the same result holds.

To prove the effectiveness of the iterative algorithm, we compare it towards the ex-



(a) 40 data points (b) 5 data points

**Figure 5.** Performance of data partitioning algorithms on Desktop platform

haustive optimum search executed over PLX for a matrix with 1000000 rows. As shown in Figure 6 we are really close to the real optimum.

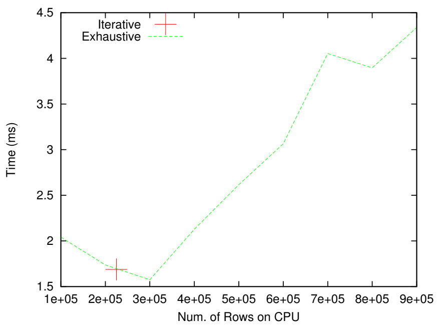


**Figure 6.** Comparison of the iterative algorithm with the exhaustive search

A final note regards the presence of multiple processing elements, in particular CPUs. Taking advantage of more computational units is strictly related to the architecture and the load balancing algorithm. In most experiments, we observed that there is no significant gain when using more CPU cores. This is mainly due to the impact of the MPI communication overhead amplified by unbalances among cores. Furthermore, the most important limit to scalability is the PCIe bandwidth. For example, with a matrix having 8000000 rows, we have to transmit, for each iteration, about 320000 bytes (with double precision); we know that on PLX the PCIe can reach at most 5840 MB/sec. With a simple formula we can estimate the matrix size that saturates the PCIe bandwidth: on the PLX platform, this limit is 10648000 rows which require to transmit about 387200 bytes for each iteration. For that amount of data, the bus reaches 5440 MB/sec that is very close to its maximum. During our experiments, PSBLAS assigns rows to processes by using a block decomposition strategy which may cause harmful communication overheads. Therefore, our idea is that multiple computational units may produce significant improvements when a more intelligent load distribution is used. A better strategy is that based on graph partitioning distribution (provided by PSBLAS as well) which aims to create domain partitions that require the minimal number of communications.

During the experiments we observed that the use of hybrid computation is not always beneficial. In fact, we get significant benefits only for the Desktop platform whose CPU and GPU are very similar in terms of performance. On AWS, which has the most powerful GPU among the 3 architectures, it is more convenient to use only the GPU. On PLX we get a small improvement for small matrices, where the CPUs and GPU throughput are almost the same. As a future development, it may be useful to define an index which expresses the affinity to hybridization of a specific platform.

## 5. Conclusions

We demonstrate how the usage of object-oriented techniques and design patterns allows an efficient utilization of heterogeneous computing platforms which are very popular in the high performance computing community. The optimization of the matrix-vector product is however just a part of a complete application. We will realize an OpenMP plugin which removes the MPI overhead and investigate preconditioners that are suitable on GPU in the context of linear solvers employing Krylov methods.

### Acknowledgments

We wish to acknowledge Amazon Web Services and the CINECA supercomputing center for awarding us with a grant for access to the AWS and PLX platforms, respectively.

## References

[1] S. Filippone and Buttari A. Object-oriented techniques for sparse matrix computations in Fortran 2003. *ACM Trans. Math. Softw.*, 38(4), 2012.

[2] V. Cardellini, S. Filippone, and D. Rouson. Design patterns for sparse-matrix computations on hybrid CPU/GPU platforms. *Scientific Computing*, 2013. doi: 10.3233/SPR-130363. To appear.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone. Performance optimization and modeling of sparse kernels. *Int'l J. of High Performance Computing Applications*, 21(4):467–484, November 2007.

[5] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. of 42nd IEEE/ACM Int'l Symp. on Microarchitecture*, pages 45–55, 2009.

[6] Z. Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on heterogeneous multicore and multi-gpu systems using functional performance models of data-parallel applications. In *Proc. of IEEE Cluster '12*, pages 191–199, September 2012.

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[8] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, January 2013.

[9] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. In *Proc. of ASPLOS '13*. ACM, 2013.

[10] Z. Wang, L. Zheng, Q. Chen, and M. Guo. CAP: co-scheduling based on asymptotic profiling in CPU+GPU hybrid systems. In *Proc. of PMAM '13*, pages 107–114. ACM, 2013.

[11] N. Whitehead and A. Fit-Florea. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. Technical report, NVIDIA Corporation, 2011.

[12] A. Lastovetsky and R. Reddy. Data partitioning with a functional performance model of heterogeneous processors. *Int'l J. of High Performance Computing Applications*, 21(1):76–90, 2007.

[13] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1-2):460–471, September 2010.