Probabilistic-based selection of alternate implementations for heterogeneous platforms

Javier Fernández(✉)*, Andrés Sánchez Cuadrado,
David del Rio Astorga, Manuel F. Dolz, and J. Daniel García

Computer Science and Engineering Department,
University Carlos III of Madrid, 28911–Leganés, Spain
{jfmunoz,andrsanc,mdolz,jdgarcia}@inf.uc3m.es,david.rio@uc3m.es

Abstract. Over the last years, heterogeneous architectures have become a *de facto* approach for improving the performance of numerous scientific and industrial applications. However, developing for these architectures is not straightforward: each processor demands its specific programming paradigm and, often, certain applications are only well-suited to run on a particular processing unit. Therefore, a major challenge arises when programming for these platforms: to select the most suitable device and routine implementation to solve a given problem. To deal with this issue, this paper proposes a novel probabilistic-based selector that uses the problem size to automatically choose the most appropriate version of a same kernel. In order to analyze this approach, we have developed this selector within the OmpSs programming framework and evaluated its accuracy and performance gains when executing different implementations of the general matrix-matrix multiplication. Finally, we also demonstrate how this solution delivers a comparable performance with respect to a runtime approach from the state-of-the-art.

Keywords: Implementation selector, Heterogeneous platforms, Autotuning, Probabilistic modeling

1 Introduction

In the recent years, the evolution of high performance computing has moved towards heterogeneous platforms comprising multiple processing units with different features and programming models [12]. Therefore, according to the needs, application developers are able to benefit from the specific characteristics provided by these architectures, e.g., SIMD capabilities of GPUs or low power consumption of FPGAs. While the benefits of using these platforms have been clearly defined, the challenges of exploiting heterogeneity have discouraged the adoption

*This work was partially supported by the EU project ICT 644235 "RePhrase: REfactoring Parallel Heterogeneous Resource-Aware Applications" and the project TIN2013-41350-P "Scalable Data Management Techniques for High-End Computing Systems" from the *Ministerio de Economía y Competitividad*, Spain.

of heterogeneous programming models. These challenges include the inherent difficulties of diverse programming paradigms and the fact that certain processors are only well-suited for applications with special demands. This has led to a progressive development of multiple architecture-specific implementations [5]. Thus, an additional challenge arises when programming for these platforms: to select the most convenient device and implementation to solve a given problem.

While a naive approach is to manually map tasks onto the underlying parallel processors, runtime schedulers have demonstrated to be a better solution in these scenarios [4]. Indeed, recent schedulers help in improving performance, since they learn incrementally from past executions. This mechanism allows them to self-tune applications by means of selecting the most appropriate kernel version and processor [7]. To pave the way, this paper extends the current literature with a novel probabilistic-based selector of alternate implementations for heterogeneous platforms (PrISe). In order to implement and evaluate this selector, we have leveraged the OmpSs programming framework instead of other solutions (such as StarPU [2]), given that OmpSs is more usable and allows to easily integrate new scheduling modules. Specifically, this work contributes with the following:

- We present an implementation selector that allows automatically choosing the most suitable implementation of a same kernel using a probabilistic and profile-guided approach.
- We incorporate the probabilistic selector as a scheduler into the OmpSs programming framework and detail which modifications have been required in its Mercurium compiler and Nanos++ runtime.
- We evaluate the proposed scheduler by analyzing the accuracy of the selections made and the performance gains using the general matrix-matrix multiplication as use case.
- We demonstrate how our scheduler self-tunes and delivers a comparable performance with respect to a runtime approach from the state-of-the-art.

The rest of this document is organized as follows. Section 2 reviews a few related works in the area. Section 3 describes the OmpSs programming framework along with its two major components: the Mercurium compiler and the Nanos++ runtime. Section 4 presents the probabilistic implementation selector as for the main contribution of this paper. In Section 5, we evaluate our approach using the general matrix-matrix multiplication and compare it with an already existing OmpSs scheduler. Finally, Section 6 closes this paper with a few concluding remarks and future works.

2 Related work

Heterogeneous architectures, combining different processing units, have become a very common scenario across the scientific community. Given that these processing units have inherent advantages and drawbacks, highly-tuned implementations of a same algorithm have been developed to fully exploit them. For example, several numerical libraries comprising highly tuned kernels, from BLAS and LAPACK, are available for diverse computing architectures: clBLAS [1] has support for OpenCL processors, GSL [8] is targeted to multi-/many-core processors, etc. This fact poses the need of selecting the most suitable pair device–implementation to solve a given problem. To deal with this issue, the solutions in the state-of-the-art have generally taken two directions: *i)* runtime schedulers, which are able to map and execute kernels from multiple libraries on the available processing units; and *ii)* static approaches, which allow selecting at compile time the most appropriate implementation according to historical data. In the following, we review some works adopting these approaches.

Regarding the approaches making static selections, we find the work by Jun et al. [13], which proposes an automatic system based on source code analysis that maps user calls to optimized kernels. Similarly, Jie Shen et al. [11] propose an analytic system for determining which hybrid programming configuration is optimal to solve a given problem. Alternatively, the approach by Rio et al. [10] presents an adaptive implementation selector that chooses, at compile time, the tuple device-implementation that delivers the best performance.

In contrast with static approaches, dynamic solutions are also widely extended in the community. A well-known runtime selector is the *versioning* scheduler [9] from the OmpSs programming framework [7]. This scheduler chooses the most appropriate task version among those marked as implementation alternatives. Another solution is the extension for the SkePu framework [6], which leverages machine learning techniques to decide which of the available versions of a given function offers the lowest execution time. Following a similar approach, the selector presented in this paper uses a novel technique based on probabilities and problem sizes that allows determining the best implementation at runtime.

3 The OmpSs programming model

The OmpSs programming model [3] is an effort to complement OpenMP with new directives to support asynchronous parallelism on homogeneous and heterogeneous architectures. OmpSs extends the execution and memory models of the OpenMP programming model in two main aspects. First, it leverages a runtime based on thread-pool instead of the traditional fork-join model. Second, it is designed to handle multiple physical addresses of the available processing units of a heterogeneous platform. Therefore, the runtime takes care of where the data resides and manages data transfers as tasks consume or produce them.

One of the key features of OmpSs is its support for pragma annotations in function declarations or definitions with the well-known `task` directive. With it, each time the OmpSs runtime encounters a function annotated with this directive, a worker thread will run its associated code onto one of the available processors. Furthermore, to provide heterogeneity, the `target` directive in task declaration allows specifying the processor that must run its code.

In general, the OmpSs environment is mainly built on top of two major components: the Mercurium compiler and Nanos++ runtime system. These components are described as follows:

Mercurium is a source-to-source compilation infrastructure targeted to the C, C++ and Fortran languages. The main goal of Mercurium is to detect the OmpSs pragmas and substitute them with calls to the Nanos++ runtime. The compiling phases of Mercurium are implemented as plugins, therefore new modules can be included for supporting new features. Code modifications can be performed by introducing raw source code instead of using its internal syntactic representation.

Nanos++ has been designed to serve as runtime to deal with the OmpSs programming model. Its main goal is to manage asynchronous parallelism by means of controlling data dependencies of tasks specified in the pragma-annotated source codes. A remarkable feature of Nanos++ is the multiple scheduling policies available for deciding the order of execution of tasks and the resource where the tasks will be executed. These scheduling policies are implemented as independent modules that are dynamically loaded at runtime. An example of module supporting heterogeneity is the *versioning* scheduler [9]. This module allows selecting the most appropriate implementation of a same task depending on the target device or the execution circumstances. This is enabled via the `implements` clause, which allows marking alternate implementations of a same task targeted to different processing units in a heterogeneous platform.

All in all, thanks to the flexible design and implementation of OmpSs programming framework, it is very easy to extend any of its features, like adding new directives and clauses to the OmpSs pragma annotations in the Mercurium compiler or extending the Nanos++ runtime modules with a scheduler. In the following section, we detail how we have leveraged these features to implement PrISe within this framework.

4 The probabilistic implementation selector

This section introduces the probabilistic implementation selector (PrISe) as for the main contribution of this paper. Specifically, we describe how we have integrated this selector as a scheduling module into the OmpSs programming framework and detail which modifications have been required in its Mercurium compiler and Nanos++ runtime.

Fig. 1 depicts the compilation and execution framework of an OmpSs application that is executed using the Nanos++ runtime along with PrISe. In a first step, the Mercurium compiler performs a source-to-source transformation of the pragma-annotated source codes and introduces the corresponding calls to the Nanos++ runtime. Then, the resulting source code is compiled with a regular C/C++ compiler, which finally generates the binary of the OmpSs application. Afterwards, during the execution, the PrISe scheduler selects an implementation each time the task is run depending on the probabilities that are calculated using historical data. These probabilities are accordingly updated by the corresponding module at the end of the application run. Finally, the probabilities and a summary of the historical data is dumped onto disk to guide future executions.

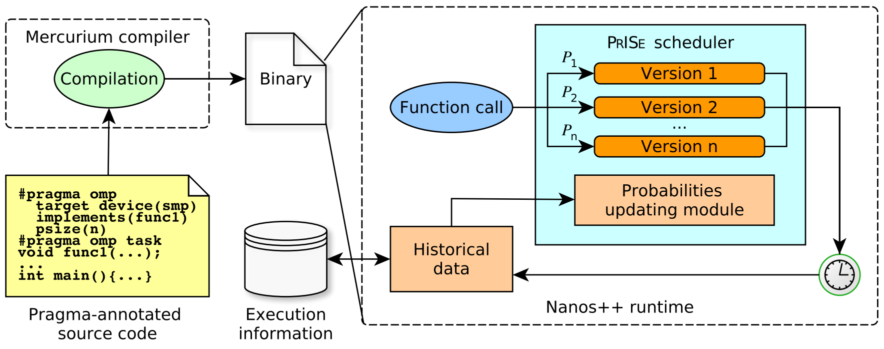


Fig. 1: Workflow from source code to execution.

In the following sections, we explain in detail the implementation selection algorithm and the probabilities updating module that have been included in the original OmpSs framework to support the new PrISE scheduler.

4.1 Implementation selector algorithm

The PrISE scheduler has been developed as a new module in the Nanos++ runtime. As stated in Section 3, the OmpSs pragmas allow specifying alternate implementations for a same task which can be selected internally by the supported scheduling modules. Specifically, this algorithm selects an alternate implementation based on the probabilities calculated for each of them. It is important to note that the weights of these probabilities depend on the version execution time and problem size. Particularly, the algorithm divides the range of problem sizes into intervals of the same length, where each might have different probabilities. With that, it uses the probabilities assigned to the interval where the input problem size belongs to.

Listing 1.1: Example of OmpSs application using different implementations.

```c
#pragma omp target device (smp) psize (2) // The second parameter contains the problem size
#pragma omp task
void func(int **m, int problemSize);

#pragma omp target device (smp) implements(func)
#pragma omp task
void func_v2(int **m, int problemSize);

#pragma omp target device (smp) implements(func)
#pragma omp task
void func_v3(int **m, int problemSize);

int main() {
  ...
  for(int i=0; i<10; i++) {
    func(matrix, problemSize);
    #pragma omp taskwait
  }
  return 0;
}
```

Concretely, the algorithm takes the following steps. First, it retrieves the input problem size and obtains the probabilities of the corresponding size interval. To obtain the problem size, we have modified the Mercurium compiler in order to implement the new `psize` clause, which extends the supported clauses of the OmpSs `target` directive. This clause is basically leveraged to indicate which parameter in the function call should be used as for the problem size. Listing 1.1 shows an example of an OmpSs application where three different implementations of the function `func` are annotated as tasks using the `implements` and `psize` clauses on the `target` directive. In this code, `psize(2)` indicates the scheduler that the second parameter contains the problem size.

Next, the algorithm chooses a candidate implementation using a roulette-wheel selection approach. This approach basically divides a line segment of length $\sum_{i=0}^{N} P_i$ in subsegments whose size correspond to the probability P_i calculated for the i-th implementation. Finally, it selects an implementation depending on the location in the segment of a previously generated pseudo-random number between 0 and 1.

4.2 Probabilities updating module

In this section, we describe the probabilities updating module, which is in charge of recalculating, after the application run, the degree of certainty that each version provides the best performance. The computation of these probabilities is mainly based on the average execution time of the different versions. Therefore, the version having the lowest execution time will lead to a higher probability and be finally preferred by the scheduler. In order to support further explanations, Eq. 1 defines that a version A provides the best performance when its expected execution time $\mathbb{E}(A)$ is lower than any other available version in the set $\mathcal{S}$.

$$Best(A, \mathcal{S}) = \forall i \in \mathcal{S} : \mathbb{E}(A) \leq \mathbb{E}(i). \tag{1}$$

The methodology to calculate the probabilities is as follows. First of all, the confidence intervals of the available implementations are computed using the averages and standard deviations of their execution time. Next, these confidence intervals are compared among them in order to determine their probabilities. For instance, if two intervals are disjointed, the option providing the best performance has a probability of 100 % of being selected. On the contrary, the probability is split between both versions. If this occurs, these versions are accordingly executed until their confidence intervals become narrow enough to avoid the overlapping. This methodology makes two general assumptions when calculating the expected execution time of a version: *i)* it is always within the confidence interval, and *ii)* it is distributed equally along the confidence interval, i.e., following a uniform distribution. For these reasons, the results obtained are not exact but accurate enough for our purposes.

Fig. 2 shows an example of three versions (X, Y and Z) with their corresponding confidence intervals along the time axis. As observed, the three confidence intervals overlap among them in some degree. In a first step, the time

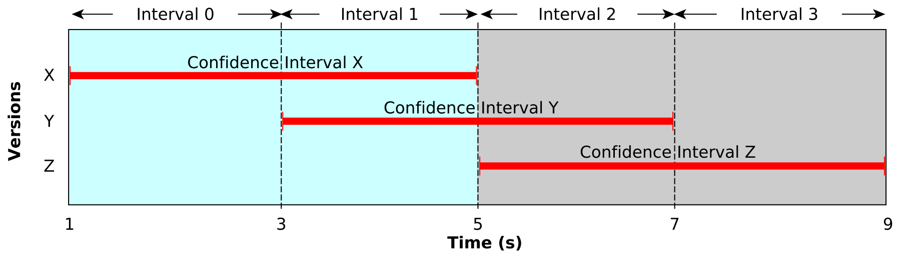


Fig. 2: Example of overlapping confidence intervals for different versions.

axis is divided into intervals that begin each time a confidence interval starts or ends. Note that in the example we obtained 4 time intervals. With this, we select those time intervals finishing at the same time or before any confidence interval, i.e., time intervals 0 and 1. This way, we can ensure that the expected execution time of the best version is within those intervals. Next, we apply the *law of total probability* in order to compute the versions probabilities by accumulating their marginal probabilities on the selected time intervals. As can be seen, the version Z does not contain any selected interval, hence, its probability is zero.

To calculate the marginal probabilities of each time interval and version, we apply again the *law of total probability* for other versions involved in the same time interval. We decompose the marginal probability into three different addends: when the expected execution time of the compared versions is lower, within or greater than the considered time interval. To illustrate the aforementioned explanation, Eq. 2, 3, 4 calculate respectively the probability of the versions X, Y and Z, shown in Fig. 2, to be better than the rest. In these formulas, I_i denotes the i-th time interval, CI_j the confidence interval of the version j, and B_t and E_t represent the begin and the end of a given interval t. Applying these equations, we get that the highest probability is assigned to version X.

$$\begin{gathered}
\mathbf{P}\big(Best(X,\{X,Y,Z\})\big) = \mathbf{P}\big(Best(X,\emptyset) \mid \mathbb{E}(X) \in I_0\big)\mathbf{P}\big(\mathbb{E}(X) \in I_0\big) + \\
\mathbf{P}\big(Best(X,\{Y\}) \mid \mathbb{E}(X) \in I_1, \mathbb{E}(Y) \in I_1\big)\mathbf{P}\big(\mathbb{E}(X) \in I_1\big)\mathbf{P}\big(\mathbb{E}(Y) \in I_1\big) + \\
\mathbf{P}\big(Best(X,\{Y\}) \mid \mathbb{E}(X) \in I_1, \mathbb{E}(Y) > I_1\big)\mathbf{P}\big(\mathbb{E}(X) \in I_1\big)\mathbf{P}\big(\mathbb{E}(Y) > I_1\big) = \\
1 \cdot \frac{E_{I_0} - B_{I_0}}{E_{CI_X} - B_{CI_X}} + \frac{1}{2} \cdot \frac{E_{I_1} - B_{I_1}}{E_{CI_X} - B_{CI_X}} \cdot \frac{E_{I_1} - B_{I_1}}{E_{CI_Y} - B_{CI_Y}} + 1 \cdot \frac{E_{I_1} - B_{I_1}}{E_{CI_X} - B_{CI_X}} \cdot \frac{E_{CI_Y} - E_{I_1}}{E_{CI_Y} - B_{CI_Y}} = \\
1 \cdot \frac{3-1}{5-1} + \frac{1}{2} \cdot \frac{5-3}{5-1} \cdot \frac{5-3}{7-3} + 1 \cdot \frac{5-3}{5-1} \cdot \frac{7-5}{7-3} = \frac{28}{32} = 0.875.
\end{gathered} \tag{2}$$

$$\begin{gathered}
\mathbf{P}\big(Best(Y,\{X,Y,Z\})\big) = \\
\mathbf{P}\big(Best(Y,\{X\}) \mid \mathbb{E}(Y) \in I_1, \mathbb{E}(X) \in I_1\big)\mathbf{P}\big(\mathbb{E}(Y) \in I_1\big)\mathbf{P}\big(\mathbb{E}(X) \in I_1\big) = \\
\frac{1}{2} \cdot \frac{E_{I_1} - B_{I_1}}{E_{CI_Y} - B_{CI_Y}} \cdot \frac{E_{I_1} - B_{I_1}}{E_{CI_X} - B_{CI_X}} = \frac{1}{2} \cdot \frac{5-3}{7-3} \cdot \frac{5-3}{5-1} = \frac{4}{32} = 0.125.
\end{gathered} \tag{3}$$

$$\mathbf{P}\Big(Best(Z,\{X,Y,Z\})\Big) = \mathbf{P}\Big(Best(Y,\emptyset) \Bigm| \mathbb{E}(Z) \in \emptyset\Big) = 0. \tag{4}$$

5 Evaluation

In this section, we evaluate the behavior of the PrISe scheduler using the general matrix-matrix multiplication (Gemm) as for the use case. First, we perform an evaluation of the accuracy and convergence of the selector algorithm using the Gemm case. Finally, we compare the performance of the PrISe and the OmpSs *versioning* schedulers.

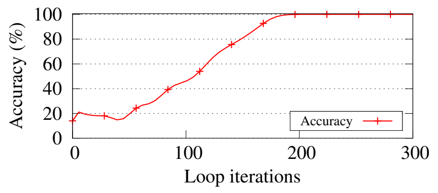
As for the heterogeneous platform, we employ a machine consisting of two multi-core Intel Xeon E5-2695 processor (Xeon) with a total of 24 physical cores running at 2.40 GHz and equipped with 128 GB of RAM. This platform is also equipped with two AMD Radeon GPUs, R9 290X (Amd1) and R9 285 series (Amd2), and an Intel Xeon Phi 3120 co-processor (Mic). On the other hand, the PrISe scheduler has been developed into the Mercurium compiler v2.0 and the Nanos++ runtime v0.12a, part of the OmpSs programming framework. Additionally, the source codes generated by Mercurium have been compiled with GCC 5.1 using the `-O3` flag.

5.1 Analysis with the Gemm use case

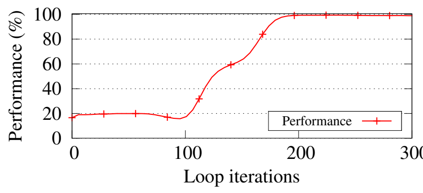
In this section, we analyze the `dgemm` kernel performance and the selector accuracy using the implementations from the clBLAS [1] and MKL libraries on the target machine. While the clBLAS `dgemm` implementation runs on all the platform processors, the MKL implementation only runs on the Xeon processor.

Fig. 3 shows the accuracy progress of PrISe and the `dgemm` kernel performance rates for increasing number of training iterations. Note that the performance rates were obtained dividing the execution time of the fastest and the selected implementation. For each of these iterations, we train the system running an instance of the `dgemm` kernel using square matrices of random sizes, ranging between 64×64 and 4,096×4,096. Afterwards, we evaluate the knowledge gained by the selector performing 100 runs of the same kernel.

As can be seen in Fig. 3a, these percentages increase in a smooth curve until reaching, after 170 training iterations, roughly 99.8 % of the total accuracy. This behavior is mainly because the confidence intervals in that iteration are narrow enough, so that, on average, the selections made are already adequate. Focusing on the progress of performance rate, shown in Fig 3b, we notice that it grows in a similar fashion than the accuracy progress. Nevertheless, each time that PrISe does not make an accurate selection, the impact in the run time is more notorious than that represented by the accuracy rate.



(a) Accuracy over training iterations.



(b) Performance over training iterations.

Fig. 3: Progress of the selector accuracy and performance over training iterations using the `dgemm` kernel.

5.2 Comparison with an alternative scheduler

In this section, we compare the performance benefits of both PrISe and *versioning* OmpSs schedulers. To assess them, we developed a synthetic benchmark consisting of two consecutive 30-iteration loops that run, in each iteration, the `dgemm` kernel using square matrices of size 1,024 and random sizes, respectively.

Fig. 4 depicts the execution progress of this application. As can be seen, PrISe starts from the first iteration of each loop selecting the implementations that perform best. This is because our scheduler uses an external file of historical data, which was collected during previous executions. (It is important to note that PrISe was previously trained performing 300 executions of the `dgemm` kernel with random matrix sizes.) On the contrary, we detect that the *versioning* scheduler does not keep any performance data among executions, so it needs a few trial runs of the different implementations until it finds the fastest one. Afterwards, the *versioning* scheduler keeps selecting the same implementation, regardless of the problem size, even if it is not the optimal. Therefore, when the matrix size varies among iterations, this scheduler is not able to self-adapt. In contrast, PrISe relies on the problem size to select the most suitable version, and thus, improves the overall performance. All in all, the presented PrISe scheduler is more adaptive and gains knowledge within application runs, while the *versioning* counterpart does not keep historical data and, therefore, needs to adapt in each execution.

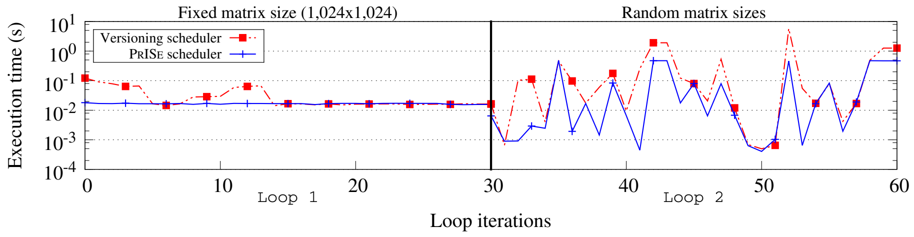


Fig. 4: Execution progress of two 30-iteration loops computing the `dgemm` kernel and using both PrISe and *versioning* schedulers.

6 Conclusions

In this paper, we have presented PrISe, a novel implementation selector that uses a probabilistic and profile-guided approach to choose the most appropriate implementation of a same kernel. To develop this selector we have leveraged the two main components of the OmpSs programming framework: the Mercurium compiler, to interpret a new pragma clause, and the Nanos++ runtime, to introduce a new scheduling module that implements this approach. To assess the proposed scheduler, we have evaluated its accuracy and performance using different versions of the general matrix-matrix multiplication.

Through the experimental results, we demonstrated that PrISe is able to select the fastest implementation of the `dgemm` kernel for varying square matrix sizes. We observed that the selector probabilities converges in roughly 170 training iterations and leads to sufficient accuracy and performance figures. Finally, we proved that our PRISE scheduler outperforms, in some cases, the performance delivered by the OmpSs *versioning* scheduler.

As future work, we plan to extend this approach for supporting high-level parallel patterns, such as the Pipeline and Farm constructions. Also, we intend to introduce a mechanism to update the probabilities during application run.

References

1. clBLAS. https://github.com/clMathLibraries/clBLAS (Apr 2015)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurr. Comput. : Pract. Exper. 23(2), 187–198 (Feb 2011)
3. Ayguadé, E., Badia, R.M., Bellens, P., Cabrera, D., Duran, A., Ferrer, R., Gonzàlez, M., Igual, F., Jiménez-González, D., Labarta, J., Martinell, L., Martorell, X., Mayo, R., Pérez, J.M., Planas, J., Quintana-Ortí, E.S.: Extending OpenMP to Survive the Heterogeneous Multi-Core Era. International Journal of Parallel Programming 38(5), 440–459 (2010)
4. Belikov, E., Deligiannis, P., Totoo, P., Aljabri, M., Loidl, H.W.: A survey of high-level parallel programming models. Tech. Rep. HW-MACS-TR-0103, Department of Computer Science, Heriot-Watt University (December 2013)
5. Brodtkorb, A.R., Dyken, C., Hagen, T.R., Hjelmervik, J.M., Storaasli, O.O.: State-of-the-art in heterogeneous computing. Sci. Program. 18(1), 1–33 (Jan 2010)
6. Dastgeer, U., Li, L., Kessler, C.: Advanced Parallel Processing Technologies: 10th International Symposium, APPT 2013, Stockholm, Sweden, August 27-28, 2013, Revised Selected Papers, chap. Adaptive Implementation Selection in the SkePU Skeleton Programming Library, pp. 170–183 (2013)
7. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures. Parallel Processing Letters 21, 173–193 (2011-03-01 2011)
8. Gough, B.: GNU Scientific Library Reference Manual - Third Edition. Network Theory Ltd., 3rd edn. (2009)
9. Planas, J., Badia, R.M., Ayguad, E., Labarta, J.: Self-adaptive ompss tasks in heterogeneous environments. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. pp. 138–149 (May 2013)
10. del Rio Astorga, D., Dolz, M.F., Sanchez, L.M., Fernández, J., García, J.D.: An adaptive offline implementation selector for heterogeneous parallel platforms. The International Journal of High Performance Computing Applications (2017)
11. Shen, J., Varbanescu, A., Sips, H.: Look before you leap: Using the right hardware resources to accelerate applications. In: IEEE Intl Conf on High Performance Computing and Communications. pp. 383–391 (Aug 2014)
12. Su, L.T.: Architecting the future through heterogeneous computing. In: 2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers. pp. 8–11 (Feb 2013)
13. Tan, W.J., Tang, W.T., Goh, R., Turner, S., Wong, W.F.: A code generation framework for targeting optimized library calls for multiple platforms. Parallel and Distributed Systems, IEEE Transactions on 26(7), 1789–1799 (July 2015)