Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Hybrid static-dynamic selection of implementation alternatives in heterogeneous environments

David del Rio Astorga · Manuel F. Dolz · Javier Fernandez · Javier Garcia Blas

Received: date / Accepted: date

Abstract With the emergence of heterogeneous architectures, developing parallel software has become an increasingly complex task. The ability of using multiple devices in a single application, such as CPUs, accelerators or co-processors, has turned the implementation and optimization tasks into a challenging process, which comes along with a variety of difficulties. The inherent complexities of the parallel algorithm, its multiple implementations, and the mapping possibilities onto one of the available processors are just examples of how intricate these tasks can become. To alleviate these issues, this paper proposes a hybrid static-dynamic selector to better exploit resources provided by heterogeneous systems. Specifically, this framework generates at compile-time a decision-tree based on historical information for selecting the implementation that performs best at run-time. To evaluate the benefits of this approach, we analyze the performance with two use cases: the general matrix-matrix multiplication and an image processing medical application. The experimental results demonstrate that our proposed selector enhances performance and minimizes efforts needed to tune applications. We proved that our solution improves from 10 % to 24 % the overall application performance in comparison with other similar approach.

Keywords Implementation selector · Heterogeneous platforms · Auto-tuning

1 Introduction

Recent trends in processors development, energy efficiency, and programming model design have manifested that heterogeneous computing is acquiring a great performance value in many scientific and engineering domains [15]. This is because heterogeneous systems comprise multiple processor architectures

David del Rio Astorga, Manuel F. Dolz, Javier Fernandez, Javier Garcia Blas
Department of Computer Science, Universidad Carlos III, 28911–Leganés (Madrid), Spain
E-mail: drio@pa.uc3m.es,{mdolz,jfmunoz,fjblas}@inf.uc3m.es

(such as multi-core CPUs, GPUs and FPGAs) which, effectively leveraged, can improve both performance and energy efficiency of applications. According to the needs, application developers are able to benefit from the characteristics provided by these distinct processors, e.g., SIMD capabilities of GPUs or low power consumption of FPGAs. However, programming efficiently on these environments is notoriously more difficult, since different programming frameworks and libraries have to be used for each processor. This has led to a progressive development of multiple architecture-specific algorithm implementations with the lack of an unified framework or API [1].

Focusing on this fact, a challenge is to select the most appropriate pair of processor and kernel implementation for solving a given algorithm. 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. Indeed, recent schedulers assist in improving performance, since they incrementally learn from past executions. This mechanism allows them to self-tune applications by means of selecting the most appropriate kernel version and processor [6]. Although less common, static approaches reduce the run-time related overheads by shifting the decision-making task directly at compile time. Several proposals in this line and based on analytic models, machine learning, and adaptive optimization methods can be found in the literature [4]. Given the foregoing, this paper broadens the current literature with the following contributions:

– We present a hybrid static-dynamic implementation selector that leverages an interface based on C++ attributes and generates source-code decision trees with the best combinations of processor-implementation.
– We propose an ad-hoc profile-guided optimization technique, allowing to generate decision trees in the source codes based on historical information and end-user requirements.
– We evaluate the performance of the selector by analyzing its convergence time and the time-to-solution of two use cases: the general matrix-matrix multiplication and a real medical application that computes a spherical deconvolution algorithm of human brain diffusion MRI data.
– We compare the proposed framework with respect to a runtime approach from the state-of-the-art and assess their performance and self-tuning capabilities.

In general, this paper extends the results presented in [13] with *i)* an improved implementation selector supporting a hybrid-based approach, *ii)* a real medical application as a use case, and *iii)* a comparative study with a state-of-the-art runtime scheduler from the OmpSs programming framework [6].

The rest of this document is organized as follows. Section 2 reviews related works in the area. Section 3 describes the hybrid static-dynamic selector framework along with the custom attributes. In Section 4, we evaluate our approach using a dense linear algebra kernel and a real medical application. Finally, Section 5 closes this paper with some concluding remarks and future works.

2 Related work

Since heterogeneous platforms have spread across the scientific community, different implementations of the same algorithm targeted to specific processor architectures have been developed. For example, several libraries comprising highly-tuned numeric kernels from BLAS and LAPACK, are available for different processors, e.g., cuBLAS [11] for nVidia GPUs, Intel MKL [9] for multi-/many-core processors, etc. This situation reveals as a new challenge the selection of the most suitable device and routine implementation to solve a given problem. To tackle this issue, two different approaches have been traditionally taken: *i)* runtime schedulers that are able to map kernels from multiple libraries on processors available in a heterogeneous platform, and *ii)* static tools that select at compile time the most appropriate implementation according to past knowledge.

Some research works using static approaches can be found in the literature. For instance, the work presented by Jun et al. [16] proposes an automatic system based on source code analysis, which maps user calls to optimized kernels. Additionally, Jie Shen et al. [14] propose an analytic system for determining which hybrid programming configuration is optimal for a given problem. Likewise, the work by Zhong et al. [17] proposes a solution that uses functional performance models (FPMs) of processing elements and FPM-based data partitioning algorithms to obtain ideal data partitions among the processing units in a heterogeneous platform. Our approach, however, differs from the latter by the fact that it bypasses data partitioning techniques, but selects the kernel implementation that performs best on any of the processing units.

On the other hand, dynamic approaches are also greatly extended in the community. Particularly, the OmpSs [6] programming framework leverages an extended set of OpenMP-like pragmas to support asynchronous parallelism and exploit task-parallelism of applications via data-dependencies. Concretely, among the available pragmas, the `target` directive allows developers to select the target device in an heterogeneous platform. Together with this directive, the `implements` clause lets users to specify that the annotated code is an alternate implementation of a given function. This feature allows its *versioning* runtime scheduler to freely map the same task onto different devices. Other works in this line, like the extension for the SkePu framework presented in [5], take advantage of machine learning techniques to automatically select the most appropriate implementation of a given function. These models basically carry out a tuning phase for estimating the ranges in which different implementations perform better than others. Following a similar approach, the framework presented in this paper gets hints from the user-code C++ attributes in order to select among implementations and processors available in the heterogeneous platform. Using the dynamic mode, applications compiled with our framework are able to select the most appropriate implementation at run-time based on a decision tree that is generated at compile time. For that purpose, our approach requires a previous training phase in order to find out which implementation performs best for a given problem size.

3 The hybrid static-dynamic selector framework

This section describes the proposed hybrid static-dynamic implementation selection framework. This framework provides an interface based on C++ attributes with the objective of generating source-code decision trees with the most suitable combinations of processor-implementation. Particularly, this framework has been designed as a feedback system, i.e., data collected during an execution influences the next ones. The main goal of the framework is to improve the selection task in each compilation. Depending on the constraints used in the attribute-annotated user codes, the selector can work using a full-static or a hybrid static-dynamic approach. On the one hand, the full-static mode replaces the annotated interfaces by a single implementation at compile time. This mode is useful when the user already knows the problem size. On the other hand, the hybrid static-dynamic mode of the selector generates `if-else` decision trees at compile time, which are processed by the user application at run-time. In the following sections we introduce the mathematical foundations of the selection algorithm and describe in more detail the modules of the selection framework.

3.1 Formal definition of the selection algorithm

In order to proceed further with our rationale for selecting implementation that delivers the best performance, we formally describe the theoretical basics of the selection algorithm used in our framework. Consider $\mathcal{V}$ a set of available versions of a same routine, s the problem size, and $t_i(s)$ the execution time of the version i using the problem size s. With this, the formula

$$Best_{point}(\mathcal{V}, s) = A \quad \Leftrightarrow \quad A \in \mathcal{V} \wedge \forall i \in \mathcal{V} : t_A(s) \leq t_i(s) \tag{1}$$

determines that the implementation A has an execution time lower than any other version in $\mathcal{V}$ for a certain problem size s. Similarly, the formula

$$Best_{range}(\mathcal{V}, [s_b, s_e]) = A \quad \Leftrightarrow \quad \forall i \in \mathcal{V} : \int_{s_b}^{s_e} t_A(x)\,\mathrm{d}x \leq \int_{s_b}^{s_e} t_i(x)\,\mathrm{d}x \tag{2}$$

states that the version A has the smallest area under its function $t_A(x)$ in the range of sizes $[s_b, s_e]$. Therefore, being the one providing the lowest run time when it is executed multiple times on different problem sizes within the range.

With the above-described formulas, it is possible to obtain the best implementation in $\mathcal{V}$ for a fixed size and a range of sizes only if the execution times for any problem size are known in advance. In other words, if the functions $t_i(x)$ are defined accurately in all its domain. However, in a real scenario this is not the case. To deal with this issue, we approximate the domain of $t_i(x)$ as the union of problem sizes intervals in the set E, i.e.,

$$E = \big\{[s_0, s_1), [s_1, s_2), ..., [s_{n-1}, s_n)\big\} \quad \Leftrightarrow \quad \mathcal{Domain}(t_i(x)) = \bigcup_{I \in E} I.$$

Note that the intervals in E are defined by the problem sizes whose execution time is known in a given point in time, e.g., if the values of $t_i(s_a)$ and $t_i(s_b)$ are known, the interval $[s_a, s_b)$ would be part of E. Furthermore, we approximate the function $t_i(x)$ with the set of functions

$$\{\tau_i^I(x) = mx + c : \forall I \in E\} \tag{3}$$

being $\tau_i^I(x)$ a linear function between the endpoints of the interval I in E. With these definitions, we redeclare the function $Best_{point}$ in Eq. 1 as

$$Best'_{point}(\mathcal{V}, s) = A \;\;\Leftrightarrow\;\; \forall i \in \mathcal{V},\, \exists I \in E : s \in I \;\;\wedge\;\; \tau_A^I(s) \leq \tau_i^I(s) \tag{4}$$

for determining approximately which version provides the best performance with the current knowledge. In this case, A is the version whose function $\tau_A^I(s)$, defined in the interval I where s belongs, has the lowest value than any other version in $\mathcal{V}$. Likewise, we also define an estimation of $Best_{range}$ in Eq. 2 as

$$Best'_{range}(\mathcal{V}, [s_b, s_e]) = A \Leftrightarrow \forall i \in \mathcal{V} : area(A, [s_b, s_e]) \leq area(i, [s_b, s_e]) \tag{5}$$

where the area under its function $t_A(x)$ is calculated approximately with

$$area(A, [s_b, s_e]) = \int_{s_b}^{I_e^B} \tau_A^{I^B}(x)\,\mathrm{d}x + \sum_{r \in R} \int_{r_b}^{r_e} \tau_A^r(x)\,\mathrm{d}x + \int_{I_b^E}^{s_e} \tau_A^{I^E}(x)\,\mathrm{d}x$$

$$: \exists s_b \in I^B \in E \;\;\wedge\;\; \exists s_e \in I^E \in E \;\;\wedge\;\; \exists R \in E \Leftrightarrow \forall r \in R, r \in (s_b, s_e).$$

Basically, this formula estimates the area of the version A in the interval $[s_b, s_e]$ by adding the areas under their $\tau_A^I(x)$ defined for each interval I contained in the range. Note that the areas of the intervals containing the endpoints of $[s_b, s_e]$ are only partially included.

3.2 Description of the framework

In this section we describe in detail the steps taken by the selection framework (see Fig. 1). First, the hardware information module extracts the platform information and stores it into a file (`HPP.json`). In the next step, the users should provide the annotated header files with the different implementations available for each interface. In the same way, the user annotates application function calls, candidates to be analyzed and replaced by our framework. With this information, the selector analyzes the function calls annotated in the user source code and replaces them by the most suitable implementation in the header files. Furthermore, the framework instruments these function calls to measure their execution time. Finally, after the application run, the framework stores the performance profiles of the instrumented functions into a file (`PERF.json`). Basically, this file contains, for each implementation and problem size, the average run time and the total number of samples collected. This allows to recalculate the averages run times in an incremental way, i.e., each time a new sample arrives. This file is later used to make selections using a profile-guided optimization approach.

Next, we describe the attributes of the framework used for annotating function calls and declarations.

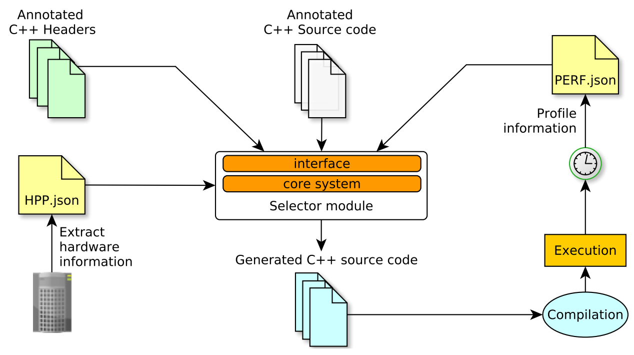


Fig. 1: The hybrid static-dynamic implementation workflow.

Header attributes. As mentioned, our selection framework requires the user intervention to declare a set of constraints for each interface and routine implementation. These restrictions specify which implementations are associated to each different function interface and target device. These requirements, in form of attributes under the `rpr` *namespace*, are the following:

– `rpr::implements`: This attribute specifies that the code under the attribute is an alternate implementation of a given interface. Basically, it receives, as the sole parameter, the function name to let the selector know which implementations are available for that interface. Consider, for instance, the general matrix-matrix multiplication (`dgemm`). In this case, the attribute contains the generic function name of the `dgemm`, although the routines actually implementing this algorithm might have different names. Note, as well, that all the implementation alternatives for an interface should specify the same name in `rpr::implements`.
– `rpr::device`: This attribute bounds a given implementation to a specific target device. Supported parameter values for this attribute are: `CPU`, `GPU`, `PHI` (for the Intel Xeon Phi co-processor), etc. [12].

Function call attributes. In this stage, the user is responsible for annotating function calls candidate to be analyzed by the selector in the user application code. This set of C++ attributes is the following:

– `rpr::interface`: This attribute indicates that the annotated function call is an interface, and should be replaced by the framework with an actual implementation during the selection process.
– `rpr::target`: It defines the preferred target device to execute the annotated function call, e.g., `rpr::target(CPU)`. Valid arguments for this attribute are those that are accepted by the `rpr::device` attribute.

In order to specify static problem constraints and to enable the hybrid static-dynamic mode in an annotated function call, the user should employ one of these options:

– `rpr::size`: This attribute is used when the user already knows the problem size during the function call. This attribute receives the problem size as a single parameter. This attribute makes the selector to work using the full-static approach, as stated before.
– `rpr::minsize` and `rpr::maxsize`: Alternatively, when the user is not able to specify the problem size, `rpr::minsize` and `rpr::maxsize` can be used to establish both lower and upper bounds of the problem size. Similar to the `rpr::size` attribute, these attributes enable the full-static selection approach.
– `rpr::dynamic`: If this attribute is set, the selector replaces the function interface by a decision tree in the source code, implemented using `if-else` statements. Therefore, the application will be able to select, at run-time, the most suitable implementation. The conditions in the `if-else` statements are evaluated using the problem size, obtained at run-time, and the intersection values where an implementation delivers better performance than other. Additionally, this attribute receives an expression producing an integer which obtains the problem size in the application context.

It is important to remark that function calls annotated by the user should match those provided in the corresponding header files.

3.3 The profile-guided selection algorithm

This section describes the internal workings of the selector, as the core module of the framework. Basically this module analyzes function calls annotated with the `rpr::interface` attribute. Then, it starts a selection process that replaces interfaces by actual implementations (using the full-static mode) or by decision trees (using the hybrid mode) complying with the restrictions stated by the user according to available implementations and processors. For that, the selector leverages a profile-guided optimization approach that takes advantage of the information gathered in the file `PERF.json` (introduced in Section 3). Note that the entire selector has been implemented using the Clang 3.8.0 compiler API that is used to analyze C++ attributes [10]. Specifically, the selector module performs the following steps:

1. First, the selector analyzes the annotated header files and the implementations provided by each function interface used in the application code.
2. Next, it checks for annotated functions in the application user code using the attribute `rpr::interface`. Simultaneously, it examines whether the user has marked the interfaces with the attribute `rpr::target` or not, i.e., to use a preferred target processor. In this case, the selector considers only the implementations for such interface that can be executed on the processor specified by the user. Other implementations are automatically discarded. If there are no implementations than can run on the preferred processor, regarding the knowledge about the platform in `HPP.json`, the implementation delivering the best performance on any available processor is taken instead.

3. Finally, if the function interface has been annotated either with the `rpr::size` or `rpr::minsize-rpr::maxsize`, the selector performs a static decision to determine which implementation, among the candidate ones, offers the best performance. Otherwise, if the `rpr::dynamic` attribute has been used, the module will calculate the intersection values where an implementation delivers better performance than other. Next, it will generate an `if-else` decision tree, which is processed by the user application at run-time in order to decide which implementation should be executed. All decisions are made according to the information stored in the `PERF.json` file.

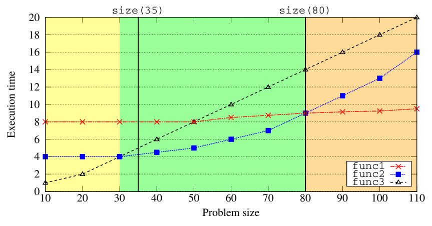
The full-static selection mode. The full-static selection mode implemented is entirely based on the problem size and boundaries specified by the user. Depending on the attributes used, the algorithm proceeds as follows.

– If the attribute `size` is set, the selector takes the implementation offering the minimum execution time according to the function $Best'_{point}$ in Eq. 4. For this purpose, the selector performs a linear interpolation for the requested problem size for all implementations available in such a function interface, in case it is not present in the performance file. (Note that to smooth extreme performance values stored in `PERF.json`, the selector only considers average execution times entries that have been computed with, at least, three samples.) Otherwise, if multiple implementations deliver the same minimum performance, the selector randomly picks one of them. However, this random policy can be eventually replaced by another that takes into account the lower maximum performance in order to avoid extreme behaviors. Consider the scenario in Fig. 2a that shows the behavior of a given function interface offering three different implementations. For instance, if the user sets the `size` attribute to 35, the selector will consider `func2`, while if the problem size is fixed to 80, the framework will randomly select `func1` or `func2`.
– On the contrary, if the developer has used both `minsize` and `maxsize` attributes to indicate a range of possible problem sizes, the selector computes the area under the performance curve (or integral) for the available implementations using the function $Best'_{range}$ in Eq. 5. With it, the framework selects the implementation that has the smallest area in the range. As shown in Fig. 2b, if the user selects a range between 25 and 50 as for the minimum and maximum problem sizes, the selector module will compute the integrals for the three implementations available. Afterwards, it will compare the areas below the curves and take that having the smallest one, i.e., `func2`. Note that if there are no performance values in the boundaries of the range, the values that intersect the boundaries are computed via linear interpolation. As in the previous option using the `size` attribute, if there are two or more implementations whose area value is equal, the selector will pick one randomly. Also, only average execution times entries with, at least, three samples are considered.

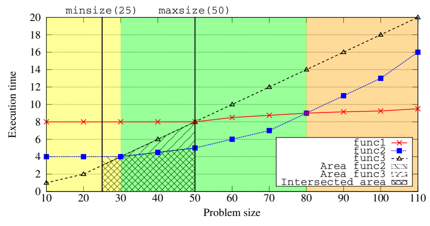
The hybrid selection mode. This mode generates, at compile time, a decision tree that is based on the performance data collected from previous executions. This tree is generated when the `dynamic` attribute is set with the following algorithm.

First, the selector calculates the intersection points among all the functions estimating the execution time, as defined in Eq. 3, for the available implementations. Following the aforementioned scenario, Fig. 2c shows the problem size intervals and the intersection points highlighted with circles. Next, for each two consecutive intersections, the best version for this interval is obtained using the function $Best'_{range}$ in Eq. 5. Fig. 2d shows the different intersection intervals with their minimum areas denoting the fastest implementation in the range. (Note that the intervals 2 and 3 are merged together in the end, as the best implementation for both is the same.) With this, the selector is able to generate a tree whose decision nodes correspond with the boundaries of all the obtained intervals and the leafs represent the implementations. Therefore, the function responsible for computing the problem size in the application context allows to walk the tree until reaching a leaf node.

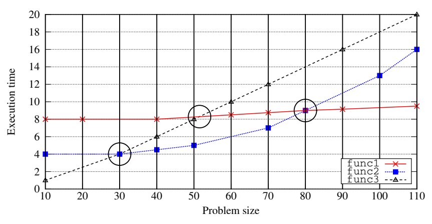
It is important to highlight that, at present, the current version of the selector only considers the problem size to select the fastest implementation. In the future, we plan to extend the set of user C++ attributes to allow users to specify other kinds of restrictions, such as memory usage or energy consumption.



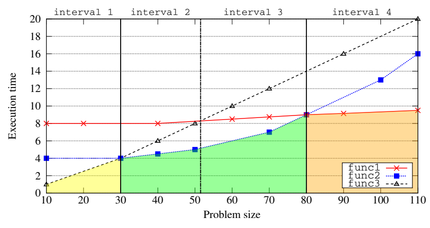
(a) Using the `size` attribute.



(b) Using the `minsize-maxsize` attributes.



(c) Using the `dynamic` attribute (step 1).



(d) Using the `dynamic` attribute (step 2).

Fig. 2: Example of the behavior of an hypothetical function interface `func` offering three different implementations (`func1`, `func2` and `func3`). Note the cutoffs for problem sizes 30 and 80 between the implementations 1–2 and 2–3, respectively.

3.4 Working example

In this section, we illustrate the workings of the framework. Listing 1 shows a header file with the attributes set by the users for the function interface `func`. In the same way, Listing 2 contains an example of user code with different attribute-annotated functions matching the interface `func` defined in the previous header file. As can be seen in the header file, three different implementations are interfacing function `func`. In their attributes, the user has defined some restrictions. For example, implementation 3 requires a GPU. Looking at the application code, the user has invoked four times this function using different attribute parameters. Finally, the selector processes and replaces these function calls with the implementations selected for each case. Listing 3 shows the code generated by the framework.

As observed, the first call has been replaced by `ns2::func2` as it is the most suitable implementation for the attribute parameters given by the user. To make this decision, the selector computes the area for the range given and picked that having the smallest value. Next, the second call is replaced by `ns1::func1` because it has the smallest minimum size for problem size 20. The third call has been substituted by `func3`, as the user specified, via `target`, the GPU as the preferred device, and any other implementation targeted to CPUs has been discarded. Finally, the fourth call has been replaced by the corresponding decision tree, as it was annotated with the `dynamic` attribute. Thus, depending on the expression `density_func(...)` provided by the user through the `dynamic` attribute and evaluated at run time, different implementations are executed.

Listing 1: Annotated header file.

```
namespace ns1 {  //Implementation 1
  [[rpr::implements("func"), rpr::device(CPU)]]
  void func1(...);
}
namespace ns2 {  //Implementation 2
  [[rpr::implements("func"), rpr::device(CPU)]]
  void func2(...);
}
//Implementation 3
[[rpr::implements("func"), rpr::device(GPU)]]
void func3(...);
```

Listing 2: Annotated user code.

```
#include <header.hpp>
int main(){
  //function call 1
  [[rpr::interface,rpr::minsize(25),rpr::maxsize(50)]]
  func(...);
  //function call 2
  [[rpr::interface,rpr::size(20)]]
  func(...);
  //function call 3
  [[rpr::interface,rpr::size(50),rpr::target(GPU)]]
  func(...);
  //function call 4
  [[rpr::interface,rpr::dynamic(density_func(...))]]
  func(...);
  return 0;
}
```

Listing 3: Processed user code.

```
#include <header.h>
int main(){
  //function call 1
  ns2::func2(...);
  //function call 2
  ns1::func1(...);
  //function call 3
  func3(...);
  //function call 4
  auto rpr_dens = density_func(...);
  if ( rpr_dens < 30 ) func3(...);
  else if ( rpr_dens >= 30 && rpr_dens < 80 )
       ns2::func2(...);
  else ns1::func1(...);
  return 0;
}
```

4 Experimental evaluation

We evaluate the presented framework along with its selector using two use cases: the general matrix-matrix multiplication (Gemm) and a real medical application that computes a spherical deconvolution algorithm of diffusion MRI data (Hardi) of human brains [8,7]. First, we perform an evaluation of the accuracy and convergence of the selector algorithm of the framework using the Gemm case. Next, we demonstrate how a real use case (Hardi) can benefit from our framework. Finally, we compare our framework with the runtime-based *versioning* scheduler from the OmpSs programming model.

We evaluate the Gemm and Hardi use cases using the Arch1 and Arch2 machines, respectively. These heterogeneous platforms are described as follows:

– Arch1 consists of two multi-core Intel Xeon E5-2695 processor with a total of 24 physical cores running at 2.40 GHz, equipped with 128 GB of RAM. This machine 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).
– Arch2 is comprised of two multi-core Intel Xeon E5-2630 v3 processor with a total of 8 physical cores running at 2.40 GHz, equipped with 128 GB of RAM. This machine also has with a NVidia Tesla K40 and a GTX 680 under CUDA version 7.5.

In both platforms, the OS used is Linux Ubuntu 14.04 x64 and the compiler employed is GCC 5.1 with the flag `-O3` set. The results presented in the next sections are the average of 5 consecutive executions.

4.1 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 [3] and GSL libraries on the Arch1 machine. Fig. 3 plots the execution times using square matrix sizes ranging from 4 to 4,096 and double-precision numbers. As can be observed, depending on the problem size, a kernel implementation delivers better performance than others. For instance, using the size range 4–504, the GSL version is the preferred option, while for the ranges 504–1,990 and 1,990–4,096, the clBLAS implementation running respectively on Xeon and Amd1 are the optimal alternatives. Thus, it becomes essential to select the best implementation depending on the matrix input size, in our case using the automatic approach presented in this paper.

Additionally, we evaluate the selector accuracy and the `dgemm` kernel performance rates by increasing the number of training iterations, using the attribute `size`, `minsize-maxsize`, and `dynamic`, indistinguishably. Note that the performance rates were obtained dividing the execution time of the fastest between the selected implementation. For each of these iterations, we train the system running an instance of the `dgemm` kernel using matrices of random

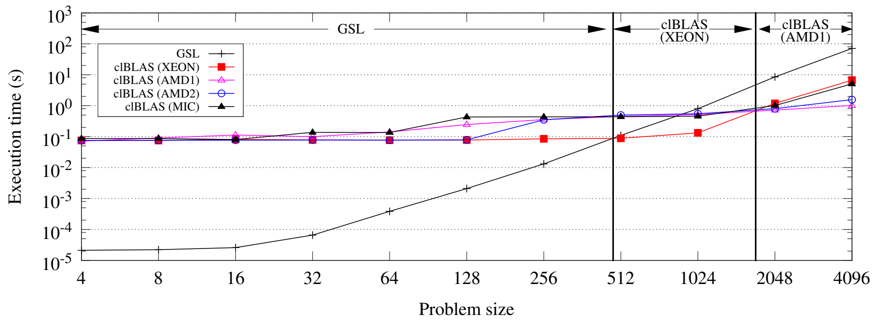
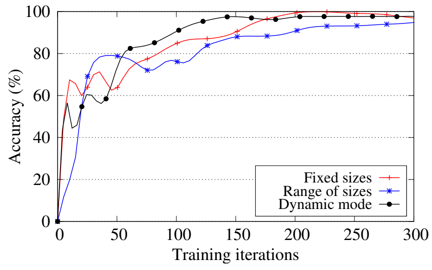


Fig. 3: Execution time of the `dgemm` kernel for different square matrix sizes and implementations.

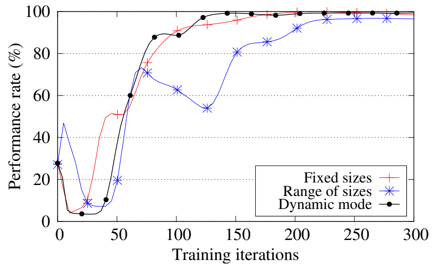
size. Afterwards, we evaluate the knowledge gained by the selector performing 100 runs of the same kernel also with random sizes. In Fig. 4a, we show the accuracy progress when using the static, i.e. using fixed and range of sizes attributes, and the dynamic modes. As can be seen, these percentages increase in a smooth curve until reaching, after 300 training iterations, roughly 97 % of the total accuracy. This behavior is mainly because the selector has already gained enough knowledge about the performance delivered by the different implementations. Looking at the performance in Fig 4b, using both static and dynamic-related attributes, the performance rates after 300 iterations reach almost 100 %. Therefore, all selections made from that point on will provide a good performance. An interesting remark is that the drops on the accuracy appearing during the first training iterations are not proportionally reflected in the performance progress. This is because a wrong selection has different consequences on the performance, and thus, depending on the implementation chosen and problem size, it might cause a lower or a higher decrease on the performance rate. In general, we find out that both static and dynamic modes provide similar performance gain. Nevertheless, there are differences between these modes: using the static approach the applications have to be recompiled whenever the problem size changes, while in the dynamic mode the applications are able to adapt themselves without recompiling them.

4.2 Analysis with the Hardi use case

We leverage the Hardi use case, responsible for executing the Robust and Unbiased Model-Based Spherical Deconvolution (Rumba-SD) method, to demonstrate the benefits of our framework. This algorithm is, up to date, one of the most advanced algorithms for detecting crossing fibers in white matter [2]. For our experiments, we use the parallel Rumba-SD method part of Hardi. We execute this parallel algorithm using different linear algebra implementations on the multi-core CPU (via Intel MKL) and the GPUs (via ArrayFire) of Arch2 with single-precision floating-point numbers.



(a) Accuracy through training iterations.



(b) Performance through training iterations.

Fig. 4: Progress of the accuracy of the selector and `dgemm` performance through training iterations.

Regarding the HARDI input data, we use a real diffusion MRI dataset acquired from healthy subject. Specifically, the whole-brain HARDI data was acquired in a 3T Philips Achieva scanner with a 8-channel head coil along 100 different gradient directions on the sphere in q-space with constant $b = 2000\,\mathrm{s/mm}^2$. Additionally, $1 b = 0$ volume was acquired with in-plane resolution of $2.0 \times 2.0\,\mathrm{mm}^2$ and slice thickness of $2\,\mathrm{mm}$. The acquisition was carried out without undersampling in the k-space (i.e., $R = 1$). The final dimension of this dataset is $128 \times 128 \times 60 \times 100$ voxels, being 60 the number of slices of 128×128 voxels, each of them comprising 100 directions.

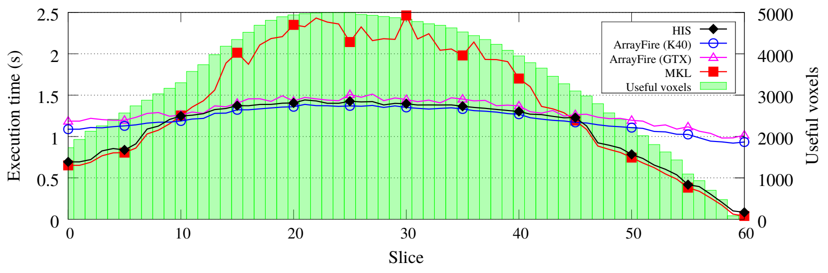


Fig. 5: Progress of the accuracy using a range of sizes.

Fig. 5 depicts the execution time and the number of useful voxels for each slice of the dataset, using lines and bars, respectively. As observed, using the MKL library, the execution times are fairly correlated with the number of useful voxels. On the contrary, the ArrayFire versions, using the GPUs (K40 and GTX 780) obtain a flatter curve. In this case, the use of ArrayFire for GPUs is only compensated for slices containing a high number of useful voxels, as the data transfers pay off the computational load. Focusing on our hybrid implementation selector (HIS), the selector takes the MKL and the ArrayFire implementations for slices comprising low and high number of useful voxels, respectively. Specifically, we configured the selector using the dynamic mode, so that, the decisions are made at run-time. We observe some negligible overheads (2 %) using our approach mainly caused by the density function run time. Note that this density function, responsible for calculating the number of useful voxels in each slice, is used each time by the decision tree in order to select the most suitable implementation. Finally, to evaluate the benefits of our approach, we computed the total execution time of HARDI using the aforementioned implementations (including HIS) in order to obtain speedup figures. We find out that using our approach with respect to MKL reduces the execution time by 24 %, while compared with ArrayFire, HIS decreases the execution about 10 %. Therefore, our approach in this case helps improving the overall performance of applications.

4.3 Comparison with alternative approaches

In this section, we validate the performance benefits of our hybrid static-dynamic implementation selector (HIS) and compare it with an existing run-time scheduler. Concretely, we compare our approach with the *versioning* run-time scheduler counterpart from the OmpSs programming model [6], as it offers a similar implementation selector to our static solution.

To compare our solution with OmpSs, we developed a microbenchmark composed of two consecutive 30-iteration loops computing the matrix-matrix product (`dgemm` kernel) using square matrices of size 256 × 256 and random sizes, respectively, in each iteration of the loops. For HIS, we annotate the `dgemm` kernel calls using the attribute `size` for the first loop and with `dynamic` for the second one. In contrast, for OmpSs we define different tasks for the available implementations that are annotated with the `implements` and `target` directives. Take into account that the multiplication is performed using the same `dgemm` implementations as in the previous experiments.

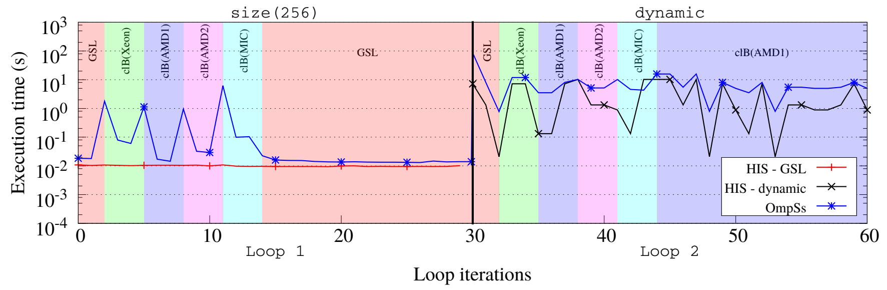


Fig. 6: Execution progress of two 30-iteration loops computing the `dgemm` kernel using HIS and OmpSs.

Fig. 6 depicts the execution progress of this microbenchmark. As can be seen, HIS starts from the first loop iteration selecting the implementations that perform best for the different matrix sizes. It is important to note that HIS was previously trained performing 100 executions of the `dgemm` kernel with random matrix sizes and the measured profiling overhead was not higher than 1 %. On the contrary, OmpSs cannot be trained offline, so it makes a few trial runs of the different implementations until it finds, at runtime, the fastest one. In these cases, the training phase of HIS pays off the OmpSs trial runs and the runtime scheduler overhead. Specifically, the OmpSs measured overhead ranges between 2 % and 40 % for the large and small matrix sizes, respectively. However, when the matrix size varies among iterations, OmpSs is not able to self-adapt and continues selecting an implementation that is not the optimal. In contrast, HIS relies on the problem size to select in each iteration the most suitable implementation, and thus, improving the overall performance. On the other hand, we have calculated the number of application executions that our approach requires (assuming that there is no previous performance data in the `PERF.json` file) in order to improve the execution time of OmpSs. We found out that, using our approach, 40 executions of the user application are necessary to compensate the training phase overheads and overtake the performance of the OmpSs `versioning` scheduler. In general, HIS offers a hybrid implementation selector that is adaptive and learns among executions, while OmpSs is a runtime alternative that has non-negligible overheads but does not require a training phase.

5 Conclusions

In this paper we have presented a hybrid static-dynamic implementation selector that uses an interface based on C++ attributes and is able to select the best combinations of processor-implementation. Its profile-guided optimization technique allows the framework to statically select implementations, i.e. by the replacing the annotated interfaces, or dynamically, i.e. by generating decision trees that are evaluated at run-time.

Through the experimental results, we demonstrated that the proposed framework is able to select the fastest implementation using fixed and variable problem sizes, and the hybrid approach. For the GEMM use case, we observed that the framework requires only 100 iterations to have an acceptable accuracy and performance gains. On the other hand, using a medical imaging use case, we proved that our approach improves from 10 % to 24 % the overall application performance. Finally, we also compared our framework with the OmpSs runtime *versioning* scheduler and confirmed that a hybrid compile time-runtime selection neglects the traditional scheduler overheads at the expense of a training phase.

As future work, we plan to extend the set of C++ attributes in order to allow users to specify other kinds of restrictions, such as memory usage or energy consumption. Furthermore, we also aim to incorporate a static partitioning module for supporting multiple devices in shared and distributed memory platforms.

Acknowledgements This work has been partially supported by the EU project ICT 644235 "REPHRASE: REfactoring Parallel Heterogeneous Resource-Aware Applications" and the project TIN2016-79637-P "Towards Unification of HPC and Big Data Paradigms" from the Spanish "Ministerio de Economía y Competitividad".

References

1. 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 (2010). DOI 10.1155/2010/540159
2. Canales-Rodríguez, E.J., Daducci, A., Sotiropoulos, S.N., Caruyer, E., Aja-Fernández, S., Radua, J., Mendizabal, J.M.Y., Iturria-Medina, Y., Melie-García, L., Alemán-Gómez, Y., et al.: Spherical deconvolution of multichannel diffusion MRI data with non-Gaussian noise models and spatial regularization. PloS one **10**(10), e0138,910 (2015)
3. clMathLibraries: clBLAS. `https://github.com/clMathLibraries/clBLAS` (2015)
4. Daoud, M.I., Kharma, N.: Efficient compile-time task scheduling for heterogeneous distributed computing systems. In: 12th International Conference on Parallel and Distributed Systems - (ICPADS'06), vol. 1, pp. 9 pp.– (2006)
5. 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. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
6. 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). DOI http://dx.doi.org/10.1142/S0129626411000151
7. Garcia-Blas, J.: Parallel high angular resolution diffusion imaging toolbox. `https://bitbucket.org/fjblas/phardi` (2016)
8. Garcia-Blas, J., Dolz, M.F., García, J.D., Carretero, J., Daducci, A., Alemán-Gómez, Y., Canales-Rodríguez, E.J.: Porting Matlab Applications to High-Performance C++ Codes: CPU/GPU-Accelerated Spherical Deconvolution of Diffusion MRI Data. In: Algorithms and Architectures for Parallel Processing - 16th International Conference, ICA3PP 2016, Granada, Spain, December 14-16, 2016, Proceedings, pp. 630–643 (2016). DOI 10.1007/978-3-319-49583-5_49
9. Intel: MKL - Math Kernel Library. `https://software.intel.com/en-us/intel-mkl` (2015)
10. Maurer, J., Wong, M.: Towards support for attributes in C++ (Revision 6). In: JTC1/SC22/WG21 - The C++ Standards Committee (2008). N2761=08-0271
11. nVidia: cuBLAS Library User Guide. nVidia, v5.0 edn. (2012)
12. R. Sotomayor, L. M. Sanchez, J. Garcia-Blas, A. Calderon, J. Fernandez: AKI: Automatic Kernel Identification and Annotation Tool Based on C++ Attributes. In: Proceedings of the IEEE TrustCom-BigDataSE-ISPA 2015, pp. 148–156 (2015)
13. Sanchez, L.M., d. R. Astorga, D., Dolz, M.F., Fernández, J.: CID: A Compile-Time Implementation Decider for Heterogeneous Platforms Based on C++ Attributes. In: 2016 Intl IEEE Conference on Scalable Computing and Communications (ScalCom), pp. 1149–1156 (2016). DOI 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0177
14. Shen, J., Varbanescu, A., Sips, H.: Look before you leap: Using the right hardware resources to accelerate applications. In: High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on, pp. 383–391 (2014)
15. Su, L.T.: Architecting the future through heterogeneous computing. In: 2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers, pp. 8–11 (2013). DOI 10.1109/ISSCC.2013.6487618
16. 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. IEEE Transactions on Parallel and Distributed Systems **26**(7), 1789–1799 (2015)
17. Zhong, Z., Rychkov, V., Lastovetsky, A.: Data partitioning on multicore and multi-gpu platforms using functional performance models. IEEE Transactions on Computers **64**(9), 2506–2518 (2015). DOI 10.1109/TC.2014.2375202