Abstract—Heterogeneous platforms are mixes of different processing units. The key factor to their efficient usage is workload partitioning. Both static and dynamic partitioning strategies have been defined in previous work, but their applicability and performance differ significantly depending on the application to execute. In this paper, we propose an application-driven method to select the best partitioning strategy for a given workload. To this end, we define an application classification based on the application kernel structure—i.e., the number of kernels in the application and their execution flow. We also enable five different partitioning strategies, which mix the best features of both static and dynamic approaches. We further define the performance-driven ranking of all suitable strategies for each application class. Finally, we match the best partitioning to a given application by simply determining its class and selecting the best ranked strategy for that class. We test the matchmaking on six representative applications, and demonstrate that the defined performance ranking is correct. Moreover, by choosing the best performing partitioning strategy, we can significantly improve application performance, leading to average speedup of $3.0 \times /5.3 \times$ over the Only-GPU/Only-CPU execution, respectively.

Keywords—Heterogeneous platforms; Workload partitioning; Application classification; Applicability; Performance

I. INTRODUCTION

Heterogeneous platforms with multi-core CPUs and accelerators, such as GPUs and Xeon Phi, are becoming attractive for parallel applications [1]–[3]. To improve application performance, we need to partition the workload to best utilize each processor on the platform. However, for a given application, how to perform the partitioning to achieve the best performance is not trivial. In this paper, we follow an application-centric way to determine the most suitable partitioning strategy for a given workload.

Our work focuses on data parallel applications, where there is massive parallelism to exploit. Each parallel section of code in the program is called a kernel, and a data parallel application can have one or multiple kernels executed in certain sequences. In data parallel applications, the workload is proportional to the number of data points to be computed in the kernel (i.e., the problem size). Partitioning the workload is therefore equivalent with splitting the data and assigning partitions to different processors.

There are multiple ways to perform the partitioning. At high level, they can be categorized into static and dynamic partitioning. Static partitioning determines a fixed partitioning before runtime, dividing the data into several partitions, typically as many as processors. Dynamic partitioning divides the data into multiple partitions at runtime (usually more than the number of processors), and schedules the partitions to processors based on certain scheduling policies.

Previous studies have developed different static and dynamic partitioning strategies suitable for different data parallel applications. The work in [4]–[10] proposed static methods; the difficulty in these studies is to determine the optimal partitioning by building prior knowledge. Moreover, these methods are only designed for single-kernel applications. Dynamic partitioning strategies for both single-kernel [11]–[13] and multi-kernel applications [14]–[17] have been proposed, where the scheduling policy, the data dependency, and the locality between multiple kernels are the main research points. These strategies have wider applicability, but often lead to suboptimal performance due to the scheduling overhead. In other words, static partitioning trades applicability for performance, while dynamic partitioning trades performance for applicability.

To satisfy both requirements—the applicability and the performance—in one go, we aim to design an application analyzer that chooses the best performing strategy for different types of applications. To design the application analyzer, we must answer two important questions.

First, what is an appropriate application classification? An appropriate classification makes it possible to propose efficient partitioning strategies. We note that simply using the number of kernels to classify the applications is not sufficient. In this paper, we use the application kernel structure, i.e., the number of kernels as well as the kernel execution flow, to classify the applications into five classes.

Second, what are efficient partitioning strategies? We answer this question in two steps. First, we propose five partitioning strategies starting from existing successful static and dynamic solutions [10], [16]. Specifically, we propose new ways to apply static partitioning to multi-kernel applications, and only use dynamic partitioning as a fall-back scenario. Next, for each application class, we analyze the
performance ranking of all feasible strategies. As a result, for each class, we have a set of suitable partitioning strategies, and their ranking in terms of performance.

To evaluate the proposed partitioning strategies and the correctness of the ranking, we select six representative applications from different application classes [18]. Our experiments show that, for each application, we are able to choose the best performing strategy, and the theoretical performance ranking we propose matches the empirical ranking. By applying the best partitioning on these applications, we achieve on average 3.0×/5.3× speedup compared to the Only-GPU/CPU execution.

In summary, the main contributions in this work are:

- We propose a classification of data parallel applications, making it possible to design suitable partitioning strategies for different application classes.
- We extend the usability of static partitioning, and further propose a mix of static and dynamic partitioning strategies to achieve both wide applicability and high performance for using workload partitioning.
- We determine the performance-based ranking of all suitable partitioning strategies for each application class (we also validate the ranking empirically).
- We design an application analyzer that selects the best performing partitioning strategy for a given application, leading to efficient application execution on heterogeneous platforms.

The rest of the paper is organized as follows. We briefly discuss existing successful partitioning solutions in Section II. In Section III, we present our analyzer with the application classification and the proposal of partitioning strategies. The empirical evaluation is given in Section IV, followed by a discussion in Section V. We study related work in Section VI, and conclude in Section VII.

II. BACKGROUND

In this section, we discuss the usage of static partitioning and dynamic partitioning on heterogeneous platforms. Based on this introduction, we analyze the strengths and limitations of each partitioning solution and the motivation for combining them for high performance heterogeneous computing.

A. The Glinda Partitioning Approach

The Glinda partitioning approach [10] predicts the optimal workload partitioning for single-kernel applications running on heterogeneous platforms. This approach supports various platforms, with one or more accelerators, identical or non-identical. Figure 1 shows an overview of the Glinda partitioning approach.

The approach consists of three main steps. The first step is modeling the partitioning, where Glinda models the execution of the workload on the heterogeneous platform. Given a fitting criteria (e.g., the minimum execution time), it builds a partitioning model that represents the optimal partitioning. This partitioning model integrates the modeling of application workload, hardware capabilities, and data transfer overhead together, and is finally expressed as an equation with two derived metrics—(1) the relative hardware capability (the ratio of GPU throughput to CPU throughput), and (2) the GPU computation to data transfer gap (the ratio of GPU throughput to data-transfer bandwidth). The two metrics are key factors for determining the optimal partitioning, and vary depending on the platform to be used, and the application and the dataset to be computed. Next, for predicting the optimal partitioning, Glinda uses a low-cost profiling to estimate the values of the two metrics, ensuring a realistic estimation adaptive to any changes of platforms, applications, and datasets. By substituting the estimations into the partitioning model, Glinda predicts the optimal partitioning. The final step is making the decision in practice. This step determines the right hardware configuration (Only-CPU, Only-GPU, or CPU+GPU with workload partitioning) taking the actual hardware utilization into consideration. By checking if the obtained partitioning is able to efficiently use a certain amount of hardware cores of each processor, Glinda takes the decision to use either a single processor or the mix of both.

In summary, the Glinda static partitioning approach provides a systematic way to determine (1) the best performing hardware configuration and, when needed, (2) the optimal workload partitioning.

B. The OmpSs Programming Model

OmpSs [16] is a high level, task-based programming model that supports heterogeneous many-core systems. In this paper, a task is an independent, parallelizable section of code that corresponds to a kernel. The user annotates a kernel with the task construct, and defines (1) the task size and (2) the task data dependencies. When the program execution reaches a task annotation, the OmpSs runtime will create an instance of the task, and will schedule it to a compute resource (e.g., a CPU core, a GPU) for execution. The total number of task instances is proportional to the user-defined task size.

OmpSs applies a thread-pool execution model [19] (different from OpenMP’s fork-join model), where the runtime creates a team of software threads (called SMP threads in OmpSs) when the program starts. A thread manages the execution of a task instance on a compute resource. The runtime analyzes data dependencies (indicated by the user) of each
created task instance, adds it to a task dependency graph, and schedules it when all its dependencies are fulfilled and there is a free compute resource. This ensures a correct, asynchronous execution of tasks, i.e., the team of threads can process different task instances (from the same kernel and/or different kernels) concurrently. The taskwait construct provides a way to set a global synchronization point after a kernel. At this point, the program waits until all previously created task instances have completed.

OmpSs enables heterogeneous computing by using the target construct with several clauses. The device clause specifies which kind of device (i.e., compute resource) executes the task. Examples of devices are CPU cores (noted as smp) and GPUs (noted as opencl or cuda). When using CPU cores, multiple SMP threads manage multiple task instances running on different cores. When using GPUs, either OpenCL or CUDA is chosen as the device-side programming model. The OmpSs memory model supports multiple memory spaces, and the runtime ensures data consistency (i.e., manages the data transfers) between memories by analyzing the user-defined data dependencies. In heterogeneous environments, the taskwait construct not only synchronizes task execution but also flushes data in different memories to the host (the CPU) memory. The implements clause allows for multiple implementations of the same task for different kinds of compute resources, so the runtime can dynamically schedule a task instance to a compute resource based on a given scheduling policy. As a result, the OmpSs runtime dynamically partitions the application workload to use different components of the heterogeneous platform.

C. Strengths and Limitations

Glinda uses a static partitioning strategy to determine the best performing hardware configuration and workload partitioning, but its usability is limited to single-kernel applications. This limitation is due to the foundation of the partitioning model, where the optimal partitioning ensures a perfect execution overlap between processors. To achieve such an overlap, the model needs to know explicitly in execution flow when the parallelism starts and ends. Single-kernel applications fit this requirement.

OmpSs uses a dynamic partitioning strategy, where the partitioning is determined at runtime without the need to know when a kernel starts and ends. It supports both single-kernel and multi-kernel applications. The runtime keeps correct data dependencies, enabling multi-kernel asynchronous execution and inter-kernel parallelism. However, dynamic partitioning introduces runtime scheduling overhead (including multiple data transfers on heterogeneous platforms) which does not exist in static partitioning. In addition, the application performance largely depends on the choice of scheduling policy and task size. A wrong scheduling policy or an unwise task size selection can lead to suboptimal partitioning and degraded performance.

Observing that both single-kernel and multi-kernel applications exist, and each partitioning strategy (static or dynamic) has its advantages and disadvantages, it is necessary to design an application analyzer that proposes the most suitable partitioning strategy for each type of applications. This is the goal of this work.

III. THE APPLICATION ANALYZER

In this section, we present our application analyzer. We first present its requirements and overview. Next, we explain in detail the key design components: a classification of applications and a set of suitable partitioning strategies.

A. Requirements

Firstly, we need to define and classify applications according to the application kernel structure. The kernel structure shows the number of kernels an application has and the kernel execution flow. Secondly, for each application class, we need to propose feasible partitioning strategies and select the one that best improves application performance.

Figure 2 gives an overview of our application analyzer from input to output. (1) We parallelize the application, obtaining the source code. (2) From the source code, we analyze the application kernel structure, and identify the class the application belongs to. (3) We select the best performing partitioning strategy from a set of options based on the determined application class. (4) According to the decision, we enable the corresponding partitioning strategy in the source code. In our work, we use OmpSs to parallelize the application, because it enables not only dynamic partitioning but also static partitioning with only few modifications. We note that the use of our analyzer is not limited to OmpSs, and users can apply our analyzer to their own implementations.

B. Application Classification

We use the application kernel structure to classify applications. Specifically, we use two criteria: the number of kernels and the type of kernel execution flow, which can be a sequence (one kernel after another), a loop, or a full DAG (kernel execution forming a directed acyclic graph). We propose therefore five application classes, shown in Figure 3.

- **SK-One** (Class I) only has a single kernel.
- **SK-Loop** (Class II) also has a single kernel, but the kernel is iterated in a loop.

1Usually, dynamic scheduling is used when referring to the case of multi-kernel applications, and dynamic partitioning for single-kernel applications. In this paper, we use dynamic partitioning to refer to both cases.
We have examined five benchmark suites (a total of 86 applications) [18]. The study shows that the five application classes cover all 86 applications. In Classes III–V, there are some applications in which one or some of kernels can be unfolded/unrolled. Therefore, the loop structure does not affect the application’s main kernel structure and the partitioning strategies that can be applied.

C. Partitioning Strategies

We propose five partitioning strategies suitable for different classes of applications and execution scenarios (illustrated in Figure 4). The five strategies are proposed starting from existing successful partitioning solutions with minimal changes, and provide more than sufficient coverage for each application class. As previous static partitioning solutions only work for single-kernel applications, we extend the applicability of static partitioning to multi-kernel applications.

- **SP-Single** is a static partitioning strategy, based, for example, on Glinda (see Section II-A), to determine a static partitioning for a single kernel. It is used for the SK-One and SK-Loop classes. We assume that, for SK-Loop, the kernel has stable performance in the loop, and therefore the partitioning remains the same (i.e., we determine the partitioning for one iteration, and use it for all iterations). If this assumption is not true, we can regard each iteration of the kernel as a different kernel, thus turning a SK-Loop application into a MK-Seq application.

- **SP-Unified** is designed for MK-Seq and MK-Loop based on the SP-Single partitioning strategy. In SP-Unified, we regard all the kernels as a single, fused kernel, and determine a unified partitioning. The condition to use this strategy is that the application does not need global synchronization between two consecutive kernels (e.g., a taskwait construct in OmpSs). Each device processes its partition (of each kernel) without inter-kernel synchronization with the host. As a result, the data locality is preserved on each device, and there is only a single host-to-device data transfer before the first kernel starts, and one device-to-host data transfer after the last kernel finishes.

- **SP-Varied** is also designed for MK-Seq and MK-Loop, where we apply the SP-Single partitioning strategy kernel by kernel, resulting in varied partitioning points. SP-Varied is used for the cases where applications need inter-kernel synchronization: (1) applications originally use synchronization to flush the data to the host for post-processing, or (2) due to the partitioning, applications need synchronization to assemble the output data of one kernel produced on different processors for the correct input of the next kernel. The use of global synchronization incurs multiple data transfers between the host and the devices.

- **DP-Dep** is a dynamic partitioning strategy usable for all application types. It schedules partitions (task instances) to devices in a breadth first order. If the application falls in Classes II–V, DP-Dep keeps tracking the data dependency chain to assign partitions that belong to the same chain to the same device, minimizing the data transfers.

- **DP-Perf** is a dynamic partitioning strategy also usable for all application types. It also tracks data dependency as DP-Dep, and implements a performance-aware scheduling policy. For each kernel, the runtime profiles how fast each device processes a partition, and estimates the device busy time (when a device becomes free for use as it finishes all the partitions that have been already assigned to it). The information is kept and updated to determine which device will be the earliest executor for the next coming partition, and the runtime will schedule the coming partition to that device. DP-Dep and DP-Perf are provided by the OmpSs runtime, and further details can be found in [20].

Next, we summarize suitable partitioning strategies for each application class in Table I, and theoretically analyze their performance ranking.

**Proposition 1:** For all classes, $\text{DP-Perf} \geq \text{DP-Dep}$ (where “$\geq$” means “outperforms or equals”).

**Discussion:** DP-Perf uses performance information to determine the scheduling. It is able to distinguish different devices and balance the workload. DP-Dep cannot distinguish such a difference, and may assign too much work to one device, leading to worse performance compared to DP-Perf. This proposition shows that it is necessary to apply a
According to Proposition 2, it outperforms DP-Dep and determines the partitioning separately. Proposition 3: For MK-Seq and MK-Loop, (1) if the application does not need inter-kernel synchronization, SP-Unified > DP-Perf ≥ DP-Dep ≥ SP-Varied; (2) if the application originally uses or needs inter-kernel synchronization, SP-Varied > DP-Perf ≥ DP-Dep ≥ SP-Unified.

Discussion: (1) For the application that does not need inter-kernel synchronization, SP-Unified regards all the kernels as a single kernel, and determines a single, unified partitioning. According to Proposition 2, it outperforms DP-Perf and DP-Dep. SP-Varied regards each kernel as an independent kernel, and determines the partitioning separately. To enable SP-Varied, we need to know explicitly when each kernel starts and ends, which means we need to add extra global synchronization points between kernels. This leads to multiple inter-kernel data transfers, an unnecessary, expensive overhead for the application. Therefore, SP-Varied performs the worst. (2) For the application that originally uses or needs inter-kernel synchronization, the execution flow is segmented by the synchronization points. SP-Varied ensures the optimal partitioning for each kernel, so it ensures the overall best performance that none of the other strategies can achieve. SP-Unified fixes a unified partitioning regardless of kernel differences, so it may result in severe workload imbalance and worse performance compared to DP-Perf or even DP-Dep.

For MK-DAG, because the execution flow is too dynamic, feasible partitioning strategies are DP-Perf and DP-Dep. It may be possible to apply static partitioning to certain kernel(s), but this requires adding extra synchronization point(s), and may or may not bring in performance improvement (which is application-specific). Thus, we recommend the use of dynamic partitioning strategies for this class.

IV. EXPERIMENTAL EVALUATION

In this section, we present the evaluation of the proposed partitioning strategies. We select a set of applications, and for each application, we test all partitioning strategies, compare their performance, and select the best performing one. Next, we verify that the empirical ranking indeed fits with the theoretical ranking we have proposed in Section III-C. In addition, we also measure the performance of Only-CPU and Only-GPU executions\(^3\) to see how much performance is gained by using the best partitioning strategy on the heterogeneous platform.

We note that for applications in the MK-DAG class, we cannot obtain valuable performance comparison between static and dynamic partitioning strategies, so we exclude the experiments for this class\(^3\).

A. EXPERIMENTAL SETUP

1) APPLICATIONS: We present in Table II the applications we used for all the experiments. We note that all these applications are selected after our study in [18], and are representative for the classes of applications we focus on. MatrixMul is a dense matrix-matrix multiplication. BlackScholes is a financial model to calculate European option prices. These two applications fall in the SK-One class, as

\(^2\)Only-CPU is the parallel execution that only uses OmpSs on the CPU. Only-GPU is the parallel execution that only uses OpenCL on the GPU.

\(^3\)We refer interested readers for the performance comparison of the two dynamic partitioning strategies in [20].
they have a single kernel for parallelization. Nbody is a scientific application that simulates interactions of individual bodies over time. HotSpot is a thermal modeling application that computes the temperature of a grid of cells over time. These two applications belong to the SK-Loop class, because both of them execute a single kernel iteratively. The STREAM benchmark is used to test memory bandwidth with four different kernels. We use STREAM in two forms: STREAM-Loop (the original form) executes the four kernels multiple times, and STREAM-Seq executes the four kernels one time (by limiting the number of iterations). They fall in the MK-Loop class and the MK-Seq class, respectively.

2) Implementation: The selected applications originate from different benchmarks, so we port the code⁴ to use OmpSs. The implementation can be summarized in two steps.

In the task implementation step, we prepare, for each kernel, a CPU implementation and a GPU implementation. The CPU implementation is the sequential implementation, and the GPU implementation is the kernel in OpenCL or CUDA (we use OpenCL in this work). By using the task and target constructs in OmpSs, we annotate the two implementations as the CPU task and the GPU task, respectively.

In the partitioning implementation step, we prepare two code versions: one for static partitioning and one for dynamic partitioning. In static partitioning, the GPU task is invoked once, and the CPU task is invoked \( m \) times to create \( m \) task instances mapping to \( m \) threads. Assuming \( n \) is the full problem size, and \( n_g \) and \( n_c \) are the GPU and CPU problem sizes obtained using static partitioning strategies (\( n = n_g + n_c \)), we set the GPU task size as \( n_g \), and the CPU task size as \( n_c/m \). In dynamic partitioning, we use the implements clause to notify the runtime that the GPU task and the CPU task are two implementations of the same task. To create \( m \) task instances, we need to set the task size as \( n/m \). The runtime decides to which device each task instance is scheduled.

3) Hardware and Software: We perform our evaluation on a heterogeneous platform which integrates an Intel Xeon E5-2620 CPU (Hyper-Threading enabled) and an Nvidia Tesla K20 GPU. Table III lists the hardware information.

The OmpSs version we use is OmpSs-14.10 (runtime version 0.7.4, compiler version 1.99.4). The applications are compiled with OmpSs compiler with -O3 option. The backend compilers for the CPU and the GPU are Intel ICC 13.3 and Nvidia OpenCL 5.5, respectively.

### Table II

<table>
<thead>
<tr>
<th>Application</th>
<th>Application class</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatrixMul</td>
<td>SK-One</td>
<td>Nvidia OpenCL SDK [21]</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>SK-One</td>
<td>Nvidia OpenCL SDK</td>
</tr>
<tr>
<td>Nbody</td>
<td>SK-Loop</td>
<td>Mont-Blanc benchmark suite*</td>
</tr>
<tr>
<td>HotSpot</td>
<td>MK-Seq</td>
<td>The STREAM benchmark [22]</td>
</tr>
<tr>
<td>STREAM-Seq</td>
<td>MK-Seq</td>
<td>The STREAM benchmark</td>
</tr>
<tr>
<td>STREAM-Loop</td>
<td>MK-Loop</td>
<td>The STREAM benchmark</td>
</tr>
</tbody>
</table>

*Mont-Blanc is implemented in OmpSs by Barcelona Supercomputing Center.

<table>
<thead>
<tr>
<th>Application</th>
<th>Application class</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatrixMul</td>
<td>SK-One</td>
<td>Nvidia OpenCL SDK [21]</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>SK-One</td>
<td>Nvidia OpenCL SDK</td>
</tr>
<tr>
<td>Nbody</td>
<td>SK-Loop</td>
<td>Mont-Blanc benchmark suite*</td>
</tr>
<tr>
<td>HotSpot</td>
<td>MK-Seq</td>
<td>The STREAM benchmark [22]</td>
</tr>
<tr>
<td>STREAM-Seq</td>
<td>MK-Seq</td>
<td>The STREAM benchmark</td>
</tr>
<tr>
<td>STREAM-Loop</td>
<td>MK-Loop</td>
<td>The STREAM benchmark</td>
</tr>
</tbody>
</table>

⁴Code available at http://www.pds.ewi.tudelft.nl/jieshen/code/
and the GPU data transfer overhead only takes a small proportion of the GPU execution time. The SP-Single strategy detects this fact, and decides to assign approximately 90% of the data to the GPU and the remaining 10% to the CPU\(^5\), leading to the best performance. As most of the work is on the GPU, the performance of SP-Single is close to that of Only-GPU. DP-Perf uses performance-aware scheduling. It actually assigns all the task instances to the GPU, so it performs slightly worse SP-Single (i.e., DP-Perf overestimates the GPU capability). DP-Dep performs much worse than DP-Perf and SP-Single, because DP-Dep by default uses all the CPU cores and the GPU, but does not take into account their different hardware capabilities. As a result, only one task instance is assigned to the GPU and the rest to the CPU, leading to workload imbalance.

\(^5\)In static partitioning, the final \(n_y\) is rounded up to a multiple of GPU warp size, and the final \(n_c\) is calculated as \(n - n_y\).

Figure 5. The execution time (ms) of different strategies in SK-One.

Figure 6. The partitioning ratio of different strategies in SK-One.

We see similar performance behavior in BlackScholes. SP-Single performs the best out of all. This application has a large data transfer overhead (the data transfer takes 37.5× more time than the kernel computation on the GPU), and SP-Single calculates a 41%/59% assignment to the CPU/GPU heterogeneous platform. DP-Perf also detects the performance change but still overestimates the GPU capability. As a result, the number of task instances assigned to the GPU exceeds the optimal, degrading the overall performance. Again, DP-Dep performs the worst because it assigns too much work to the CPU.

2) The SK-Loop Class: We evaluate Nbody and Hotspot for this class. Both applications perform a kernel computation in a loop, and the computation output of one iteration is the input of the next iteration. There is a global synchronization point after each iteration to ensure the outputs from different processors are combined at the host and updated to the input buffer before the next iteration. In Nbody, we computes the status of 1,048,576 bodies stored in 1D arrays (64 MB). In HotSpot, the grid size is 8192×8192 (0.75 GB), and we apply row-wise partitioning as for MatrixMul.

Figure 7. The execution time (ms) of different strategies in SK-Loop.

The partitioning strategies we compare are SP-Single, DP-Perf, and DP-Dep. In SP-Single, we determine the partitioning for one iteration, and use it for all iterations. Figure 7 shows the performance comparison, and Figure 8 shows the partitioning ratios. We see that in both applications, SP-Single gets the best performance. For Nbody, the GPU performs much better than the CPU, so SP-Single assigns most of the work to the GPU. On the contrary, HotSpot has better performance on the CPU, and SP-Single assigns a large partition to the CPU (the GPU performs worse mainly due to the data transfer overhead). DP-Perf detects similar partitioning as SP-Single in both applications, but its performance is worse than SP-Single (even worse than Only-GPU in Nbody) because of the dynamic partitioning overhead, which includes multiple times of taking scheduling decisions, OpenCL kernel invocations, and data transfers. DP-Dep does not distinguish different processors, thus resulting in the worst performance.

Figure 8. The partitioning ratio of different strategies in SK-Loop.

Summary 1: SP-Single is the best performing partitioning strategy for applications in the SK-One and SK-Loop classes. If we choose DP-Perf or DP-Dep, we observe sub-optimal performance.

3) The MK-Seq Class: We evaluate STREAM-Seq for this class. The application performs 4 different kernels (copy, scale, add, and triad) on 1D arrays. The number of array elements is 62,914,560 (0.7 GB). We further consider two cases of executions, with and without inter-kernel synchronization.
We note that inter-kernel synchronization is originally not needed, but we manually add it to mimic applications that need synchronization (see Section III-C).

SP-Unified, SP-Varied, DP-Perf, and DP-Dep are the partitioning strategies to be considered. Figure 9 shows the performance comparison results. When there is no inter-kernel synchronization, SP-Unified performs the best. It regards all the kernels as a single, fused kernel, and keeps the partitioning at 44% of the elements on the GPU and 56% of the elements on the CPU. The GPU gets less work mainly because its data transfer takes too much time (around 88% of the overall execution time). DP-Perf and DP-Dep rank second. Both strategies allow for asynchronous executions of task instances from different kernels. We note that there is no visible performance difference between the two strategies. This is because DP-Dep assigns most of the task instances to the CPU, which coincidentally matches the partitioning obtained by DP-Perf. SP-Varied performs the worst, because the use of this strategy requires extra global synchronization points, leading to extra data transfers between kernels. Compared to SP-Unified, the partitioning is skewed towards the CPU in SP-Varied. Figure 10 shows the partitioning ratio of different strategies.

In dynamic partitioning, the synchronization serializes the kernel execution flow, leading to 35% performance degradation compared to that without synchronization. In SP-Unified, we use the partitioning obtained in the case without synchronization. This partitioning makes the GPU get too much work for each kernel, and therefore gets the worst performance.

4) The MK-Loop Class: We evaluate STREAM-Loop for this class. Similar to STREAM-Seq, we compare the execution with and without inter-kernel synchronization. The problem size is the same as STREAM-Seq.

Figure 11 shows the performance comparison. When there is no inter-kernel synchronization, Only-GPU outperforms Only-CPU (different from STREAM-Seq), because the 4 kernels are iterated multiple times, increasing the computation workload. The other performance results are similar to those in STREAM-Seq: (1) SP-Unified obtains the best performance in the case without synchronization. We determine a unified partitioning for one iteration, and use it for all iterations. We note that the data transfer is not profiled, because all the iterations except the first and the last ones do not have any data transfer. (2) SP-Varied performs the best in the case with synchronization. As the partitioning is obtained by profiling one iteration, the partitioning ratio per kernel is the same as that in STREAM-Seq (thus, the partitioning ratio figure is not shown to avoid repetitiveness). (3) DP-Perf and DP-Dep take second place. The benefit of asynchronous execution (in the case without synchronization) increases as the number of iterations increases.

Summary 2: For applications in the MK-Seq and MK-Loop classes, the choice of the best partitioning strategy depends on whether the application needs inter-kernel synchronization: for applications with no synchronization, SP-Unified performs best, while for applications with synchronization, SP-Varied performs best.

5) Overall: The performance ranking of different partitioning strategies in our empirical evaluation matches the theoretical ranking we have proposed in Table I.

For each application, we compare the performance of the best partitioning strategy to that of the Only-GPU and
Only-CPU executions, and present the speedup in Figure 12. We see that the performance improvement is application dependent. The speedup ranges from as much as 22× to close to 1×, mainly because each processor’s capability is application dependent. The average performance improvements for our six applications are 3.0× and 5.3× compared to Only-GPU and Only-CPU, respectively.

V. DISCUSSION

From our experimental evaluation, we see that using partitioning on heterogeneous platforms improves application performance. The best performance is achieved by static partitioning strategies in 4 out of 5 application classes, as static partitioning, as long as it is applicable and optimal, ensures a perfect execution overlap between processors. Dynamic partitioning introduces scheduling overhead at runtime, and therefore gets less performance improvement even when it achieves the same partitioning ratio as static partitioning.

The scheduling policy has a significant influence on the performance of dynamic partitioning. A good policy on heterogeneous platforms should take into account the processors’ difference in hardware capability. The task size (the granularity of partitioning) impacts performance as well. In our experiments, we have also varied the task size in dynamic partitioning, and found that the task size variation leads to performance variation. Thus, auto-tuning is recommended to find the best performing one. But even so, static partitioning outperforms dynamic partitioning for the first four classes of applications.

Pragmatically speaking, for a given parallel application which is not yet partitioned, we recommend using our application analyzer (see Section III-A) to find the best partitioning strategy. If it is already partitioned in a dynamic way, but the best strategy is static partitioning (which is likely to happen), we can make dynamic partitioning “behave” like static partitioning: (1) set the task size to the full problem size, and determine the static partitioning ratio; (2) convert the static partitioning ratio to the task assignment ratio (e.g., k task instances on the CPU and l task instances on the GPU); (3) assign the determined numbers of task instances to the CPU and the GPU, respectively. Using this approach, the application gets a close-to-optimal partitioning with minimal manual effort.

VI. RELATED WORK

Static workload partitioning on CPU+GPU heterogeneous platforms has attracted quite some research recent years. Our previous work [9], [10] builds a partitioning model that uses modeling, profiling, and prediction techniques to determine the optimal workload partitioning for balanced and imbalanced applications. Luk et al. proposed Qilin [4] which builds an analytical model based on curve-fitting to determine the best workload distribution. Insieme [5] uses offline training and machine learning to build a prediction model that derives the partitioning based on program features and problem size dependent features. Grewe et al. applied a similar machine learning approach [6], and they also considered the partitioning in the presence of GPU contention [7]. SKMD from Lee et al. [8] utilizes a decision tree heuristic to search the best workload partitioning taking into account the performance variation of each processor. All these approaches determine a fixed, static partitioning between heterogeneous components of the platform, but their usability is limited to single-kernel applications. In this paper, we extend the usability of static partitioning to multi-kernel applications.

Apart from that, single-kernel dynamic partitioning schemes have been developed to achieve load balancing at runtime. Boyer [11] proposed a scheduling algorithm based on varied chunk size. The chunk size is increased by a factor at each scheduling time and the execution times of the scheduled chunks are used to partition the remaining work. Scogland et al. [12] proposed a runtime system that divides an accelerated OpenMP region across CPUs and GPUs based on four optional scheduling policies suitable for different execution scenarios. Ravi et al. [13] proposed both uniform- and non-uniform-chunk distribution schemes, where the latter assigns larger chunks to the GPU and smaller chunks to the CPU. These schemes efficiently reduce scheduling overhead, but still cannot outperform the optimal partitioning determined by the static partitioning approaches.

Dynamic scheduling for multi-kernel applications has been proposed at two different scheduling granularities: per kernel and per chunk (of each kernel). Becchi et al. [14] proposed a method to determine the location of each kernel taking the processor disparity and the data locality into consideration. The work in [15] determines the kernel-processor mapping based on code and runtime features. Unlike the OmpSs dynamic partitioning strategies [16], [20] used in this paper, which schedules at chunk (task instance) granularity, these approaches cannot utilize inter-kernel parallelism to further improve application performance. StarPU [17] uses the same scheduling granularity as that in OmpSs, and therefore also enables inter-kernel asynchronous execution while maintaining correct data dependencies at runtime.

Compared to related work, this paper proposes a set of five partitioning strategies, combining both static and
dynamic features, for both single- and multi-kernel applications. Moreover, we propose an application classification and a performance ranking of different strategies for each application class. Thus, for a given application, we are able to determine the best strategy to partition the workload, maximizing the performance improvement.

VII. CONCLUSION

Workload partitioning is mandatory to improve application performance on heterogeneous platforms. To achieve both high performance and wide applicability for workload partitioning, matchmaking applications and partitioning strategies is desirable. In this paper, we classify data parallel applications into five classes by analyzing the application kernel structure, and we propose a set of suitable partitioning strategies which combine static and dynamic approaches to cover all application classes. We further design an application analyzer that uses performance ranking to select the best performing strategy for a given application. Our work improves the applicability of static partitioning, and demonstrates its superiority, in many cases, over dynamic partitioning. By combining both static and dynamic partitioning, our analyzer applies a unified method enabling a large variety of applications to be executed efficiently on heterogeneous platforms. In the future, we plan to apply our analyzer to heterogeneous platforms with other types of accelerators, and further extend its usability. We also want to investigate the possibility to refine the classification of MK-DAG applications for a better selection of their preferred partitioning.

ACKNOWLEDGMENT

This work is supported by the HiPEAC collaboration grant (EU ICT-287759), the CSC (China Scholarship Council) Scholarship, the Spanish Ministry of Education (TIN2012-34557), and the Generalitat de Catalunya (MPEXPAR-2014-SGR-1051). The authors would like to thank Judit Planas, the author of the OmpSs performance-aware scheduler, for her help to our questions on the scheduler details.

REFERENCES