García, J.D., Sotomayor, R., Fernández, J., Sánchez, L.M. (2015). Static partitioning and mapping of kernel-based applications over modern heterogeneous architectures. *Simulation Modelling Practice and Theory*, 25(1), pp. 79-94

DOI: 10.1016/j.simpat.2015.05.010

[1]

# Static partitioning and mapping of kernel-based applications over modern heterogeneous architectures

J. Daniel García, Rafael Sotomayor, Javier Fernández, Luis Miguel Sánchez

*Computer Architecture Group,*
*Computer Science and Engineering Departament,*
*Universidad Carlos III de Madrid,*
*28911 Leganés, Madrid, Spain*
*E-mail: josedaniel.garcia@uc3m.es*

**Abstract**

Heterogeneous architectures are being used extensively to improve system processing capabilities. Critical functions of each application (kernels) can be mapped to different computing devices (i.e. CPUs, GPGPUs, accelerators) to maximize performance. However, best performance can only be achieved if kernels are accurately mapped to the right device. Moreover, in some cases those kernels could be split and executed over several devices at the same time to maximize the use of compute resources on heterogeneous parallel architectures.

In this paper, we define a static partitioning model based on profiling information from previous executions. This model follows a quantitative model approach which computes the optimal match according to user-defined constraints.

We test different scenarios to evaluate our model: single kernel and multi-kernel applications. Experimental results show that our static partitioning model could increase performance of parallel applications by deploying not only different kernels over different devices but a single kernel over multiple devices. This allows to avoid having idle compute resources on heterogeneous platforms, as well as enhancing the overall performance.

*Keywords:* parallel computing, heterogeneous computing, kernel partitioning

## 1. Introduction

In recent years, the number of heterogeneous architectures in the top500 list [1] has been increasing. They are mainly based on a combination of one or several GPGPUs and CPU cores. Those architectures consist of several computing devices that work together, where some are better fitted than others for specific classes of algorithms. The main computing device is a multi-core CPU well fitted for task parallelism, while graphics processing units (GPU) are currently very popular for most data parallel algorithms. However, in general, many applications do not make efficient use of available computing resources, which leads to a reduction of global efficiency for those parallel heterogeneous architectures [2].

The use of accelerators to improve performance of parallel programs is widespread. However, there are still some limitations in the use of those accelerators. For example, the memory-bound problem [3] refers to a general case where a code is limited by memory access [4]. Additionally, "*a data-transfer bottleneck emerges as data to be consumed and produced by the GPU must be transferred from the host CPU memory towards the GPU memory (and vice versa). Data transfers tend to become a performance bottlneck because the computing processors (CPU, GPU*

[1]Published as Static partitioning and mapping of kernel-based applications over modern heterogeneous architectures. J. Daniel Garcia, Rafael Sotomayor, Javier Fernández, Luis M. Sánchez. Simulation Modelling Practice and Theory, 58(1):79-94. November 2015. DOI: http://dx.doi.org/10.1016/j.simpat.2015.05.010. 

*or accelerators) are increasing their throughput much faster than the bandwidth of the physical connections. This problem cancels out any gain obtained from the external accelerators (e.g. GPU, Xeon Phi)*"as noted in [22].

This problem cancels out any gain obtained from the external accelerators (e.g. GPU, Xeon Phi). Finally, not all algorithms fit well to GPGPU programming model [5], with the multi-core programming model being a better choice for certain parallel applications [6].

Open standards regarding parallel programming models have been proposed for developing new software over heterogeneous architectures, such as OpenACC[7] or the latest version of OpenMP (4.0 or higher) [8]. However, those standards are currently not fully supported by all hardware devices.

The Open Computing Language (OpenCL) [9] is a C-based programming model, used for different computing devices (e.g. CPUs, GPGPUs, DSP, FPGA, accelerators) that has become widely accepted and supported by major vendors. OpenCL is based on parallel code regions, called kernels, that could be executed on a device. OpenCL allows the development of heterogeneous parallel applications that could use more than one computing device, improving application efficiency.

Achieving an efficient mapping of the kernels onto the available computing devices is challenging due to the variation in characteristics and requirements of those kernels. The characteristics from each kernel (e.g. data input and output size, data access pattern or number of branches) and the features of each device result in different performances for different kernel-device combinations. Therefore, some applications include several versions of the same kernel in order to use the one that best fits the device at hand. Moreover, sometimes one kernel can be split into several sub-kernels that can be spread over a set of computing devices. This operation, called partitioning, can improve the performance obtained with only one computing device. This partitioning may affect different sections of the code (task-level) or one or more variables, which are split (over a loop, for instance) to obtain data-level parallelism.

The final goal is the generation of an efficient kernel mapping and partitioning based on a careful selection of the best version for each kernel, a correct management of the dependencies between the selected kernels, and the use of the profiling information about their expected performance.

This paper proposes a partitioning model that allows to describe the different schedules considering kernels, devices, input/output sizes and the relations between them. All these concepts are combined in a representation of the possible schedules for a given programs. The resulting representation is used in an algorithm that performs an off-line partitioning and scheduling of a set of kernels obtained from an application. The partitioning/scheduling is focused on maximizing the performance of the kernels executed in parallel, while observing the dependencies between them. Also, an execution algorithm is presented which allows the application to run the scheduling plan previously obtained.

The rest of the paper is organized as follows: Section 2 presents the scheduling model, the static off-line partitioning/scheduling proposed algorithm and the algorithm to run the execution plan; in Section 3 we show an evaluation of the proposal performed with single kernel and multi kernel examples; in Section 4 we review related work; and finally, in Section 5 we provide conclusions and outline future work.

## 2. Scheduling model and algorithm

### *2.1. Model overview*

Parallel heterogeneous architectures are those where different computing devices are available (e.g. CPUs, GPUs, DSPs, FPGAs, and other accelerators). In this paper the study is restricted to single node computers, explicitly leaving out multi-node architectures (e.g. clusters). The ultimate objective is to be able to schedule the set of kernels from a codebase into the computing devices in the heterogeneous platform. To this end, a model has been created that allows to represent all different possible schedules. The model is based on four key aspects: kernels, input/output size, devices and transfer rates. Each pair of kernel and input/output size takes a certain time to run. Also related to the data size is the transfer rate. Lastly, each device has its own strengths and limitations, and as such their performance will vary from kernel to kernel.

For the purposes of this study, the combination of a kernel and an specific input size is named an *execution unit*. Thus, a given kernel will be considered a different execution unit when run with an input size of 256 or 512 bytes.

There are two important relationships among execution units: *incompatibility* and *dependency*. Two execution units are *incompatible* if both cannot be executed together on the same application run. The idea behind this is that the algorithm is able to consider different versions of the source code, selecting the best one for each device. For example,

it may decide between executing one large execution unit or several smaller execution units when considering the same section of code, or whether to run the outermost or innermost loop as an execution unit. One execution unit *depends* on another if it needs to wait until the latter has finished. This categorization allows to detect independent execution units, which can be run in parallel. Ultimately, this will allow to differentiate between valid and invalid sequences of execution, improving the computation of the best schedule accordingly.

Another measure of interest is the *feasibility* of deploying an execution unit on a specific device. One execution unit is *feasible* for one device if it can be executed on that device. This is required to decide which is the best device for each execution unit considering only valid devices for each execution unit. For example, any execution unit that performs system calls shall be considered unfeasible for a GPGPU device. An execution unit with a sufficiently large input data may be unfeasible for a device with limited memory.

The previous concepts are formalized as follows:

Let $E$ be set of execution units $E = \{e_1, \ldots, e_l\}$, $D$ the set of devices $D = \{d_1, \ldots, d_n\}$ and $R$ the set of execution restrictions $R = \{r_1, \ldots, r_n\}$.

Let $r_e$ be the restrictions that apply to an execution unit, defined in Equation 1. Also, let $r_d$ be the restrictions that apply to a device, defined in Equation 2.

$$r_e = \{e, R_e\} \mid e \in E, R_e \subseteq R \tag{1}$$

$$r_d = \{d, R_d\} \mid d \in D, R_d \subseteq R \tag{2}$$

Let $R_e$ be the set of restrictions that apply to all the execution units $R_e = \{r_{e_1}, \ldots, r_{e_l}\}$, such that $|E| = |R_e|$, and $R_d$ be the set of restrictions that apply to all the devices $R_d = \{r_{d_1}, \ldots, r_{d_m}\}$, such that $|D| = |R_d|$

Given the previous sets, we consider the relationships of incompatibility, dependency and feasibility, defined respectively in Equations 3,4 and 5.

$$I \in M_{e\times e}(\{0,1\}) \mid I_{i,j} = \begin{cases} 1 \text{ if } e_i \text{ and } e_j \text{ cannot be executed in the same run} \\ 0 \text{ otherwise} \end{cases} \tag{3}$$

$$P \in M_{e\times e}(\{0,1\}) \mid P_{i,j} = \begin{cases} 1 \text{ if } e_i \text{ depends on } e_j \\ 0 \text{ otherwise} \end{cases} \tag{4}$$

$$F \in M_{e\times d}(\{0,1\}) \mid F_{i,j} = \begin{cases} 1 \text{ if } R_{e_i} \cap R_{d_j} = \emptyset \mid R_{e_i} \in R_e, R_{d_j} \in R_d \\ 0 \text{ otherwise} \end{cases} \tag{5}$$

The data and constraints defined so far are extended to include data partitions. Execution units can be partitioned in order to execute different parts simultaneously on at least two devices. Such partitions share the kernel code, but have only a fraction of the input and/or output datasets. Consequently, data can be processed in parallel. Splitting execution kernels also allows to run execution units with bigger input/output data sets than a device can handle. Each partition is responsible for transferring the portion of the required input/output data to and from the device. Thus, kernel code must be extended to include data slicing and transfer. However, not all cases can be partitioned. In general, partitioning is posible in most scenarios where each execution thread has no data dependencies on other threads output, which is the typical case in kernels that can be mapped to SIMD computing devices. Worst case scenario arises in the case of data inputs that cannot be partitioned as the whole input dataset must be used by every task. In this case the whole input needs to be sent to each partition device. Even when possible, partitioning is not always the most efficient solution. There are several cases when this is the case, such as those execution units with small workloads or input sizes, algorithm where the computational time is reduced compared to the transfer time, etc. Lastly, a threshold will be considered before partitioning. The purpose of this threshold is to avoid "*false positives*", where the algorithm proposes a partition that would yield a very small speedup, and because of the variance of the measures, results would be worse than not partitioning at all. For all cases, this threshold has been established to 10% of the data size. Whenever less than that amount is proposed to be moved to a device other than the CPU, the partition is ignored and everything is run in the CPU.

A wide range of possible execution unit partitions is considered: given the sets of execution units $E$ and devices $D$, a set of execution unit partitions $EP = \{ep_1, \ldots, ep_r\}$ is defined. Each execution unit partition $ep_i = (e_a, d_b, g_c, r_d)$ consists of an execution unit $e_a$, a device $d_b$, a range $g_c$ and a ratio $r_d$. The range ($g_c$) represents each of the executions of the kernel code when the data set does not fit into the device memory and, consequently, several executions are needed for different subsets of the data set. For example, a 4 GiB dataset run in a device with a memory limit of 2 GiB would lead to two ranges. This is an instance of the memory bound problem. The algorithm must ensure that two execution unit partitions with the same execution unit, device and range are not included in the same scheduling plan. The ratio ($r_d$) is the fraction of the input dataset that is assigned to an execution unit partition. For example, if the input data set is divided into 20% to the CPU, 50% to a GPU and 30% to another GPU, corresponding ratios of partitions would be 0.2, 0.5, and 0.3. A wide set of rates from 0 to the maximum rate that can be stored in the device memory are considered for each range.

### *2.2. Algorithm overview*

After creating the partitions set, the algorithm has to obtain a scheduling plan composed by a sorted list of execution unit partitions $SP = \{ep_1, \ldots, ep_m\}$. The plan should cover all the kernels required with their complete datasets and achieve the best possible performance. The incompatibility, dependency and feasibility matrices are extended to cover execution unit partitions ($IP \supseteq I, PP \supseteq P, FP \supseteq F$).

Lastly, a function to estimate the execution time of each kernel partition is required. This function then uses profiling data from previous kernel executions on those devices with different input data sizes. The algorithm requires execution time for at least two data sizes, and the transfer times from the host to all devices for at least two data sizes (not necessarily the same ones). The database does not need to be extensive, but at the very least, it requires the execution and transfer times for the largest and the smallest data sizes, so that the algorithm may apply linear interpolation. For this experiment, execution times of several use cases has been taken, with several input data sizes for each use case. Furthermore, data transfer time is stored for the different devices considered in the scheduling process.

In this work, a simple linear interpolation function is used taking profiling data to provide estimations for data sizes that are yet to be measured. As for the partitions, the function allows to estimate the execution and transfer times for partitions different to the ones that have been measured. This allows to provide a relatively high speedup with a small profiling work. The interpolation method is supposed to be incremental (only for applications that are meant to be executed more than once). The first time the interpolation is done with few values and a great margin of error. As executions are being made more values are obtained and the estimations get better.

The smallest and biggest limits for the interpolation also change as the application is run several times. Initially, the smallest and biggest values are those of the smallest and biggest input datasets provided by each of the benchmarks used in this work. However, in a general case without such information an application may start with two arbitrary sizes. As kernels are partitioned and executed, feedback provides a path to improve accuracy.

Although we use here this simple estimation function, more complex functions may be used. For example, it could be of interest to provide estimations based on code characteristics (via fuzzy logic, genetic algorithms or similar techniques), rather than using simple linear interpolation.

To ensure that when a kernel is started all the needed input data set is available at that device, the location of every parameter needs to be tracked. To this end, a list of parameter locations is kept and updated, representing the computing devices where these parameters are located. When a new execution unit partition is to be executed, all the input parameters that are not already on the device where the partition is to be run need to be be transferred there.

Our model does not take into consideration the possibility of device-to-device transfers. Therefore, all the parameters needed by a device are first looked up in the host. In the event that those are not found in the host, but have been produced in a different device, they are first transferred from this device to the host, and secondly, all the needed parameters are transferred from the host to the requesting device. Whether data will be completely transferred or only partially transferred depends on kernel partition rate.

Similarly, output parameters need to be taken into consideration. Complete output parameters are left on the device while partial output parameters should be transferred to the host to be combined with the rest of parameter data. A partial parameter mapping is used to state which parameters are partially stored on the host and their corresponding storage rate. This mapping is updated each time partial output parameters are transferred to the host. If one partial parameter is completed then it is moved from this mapping to the parameter location list.

When the parameter location list is initialized, the parameters available at start-up are stored on the host. Any output parameter that needs to be produced is unchecked until the time where an execution unit creates it. The list of parameter locations is updated after each kernel partition execution. At the end, the final output parameters need to be transferred from the device where they are stored to the host.

Once the scheduling plan is done, it can be executed using an execution algorithm. This algorithm selects the next execution unit partition to be executed on each device. Then it runs the execution unit and performs the data transfers in the background. Finally, it waits until the current execution unit partition and its transfers are finished to select and run the next one.

The whole process is divided into three phases: the partitioning phase, the scheduling phase and the execution phase. Figure 1 shows the general work-flow. Stage one takes an input source code and generates the full list of execution units and partitions to be considered. This list is passed to the scheduling phase, which in turn will find the theoretical best scheduling plan. Afterwards, this plan will be executed in the third phase. The second phase is the longest out of the three, mainly because it has to generate many possible final schedules and even more partial schedules. In the case that there is a need to decrease the execution time of this phase, the list generated in the previous phase should contain as few partitions as possible. The trade-off would be the worsening of the performance that the schedule plan may attain.

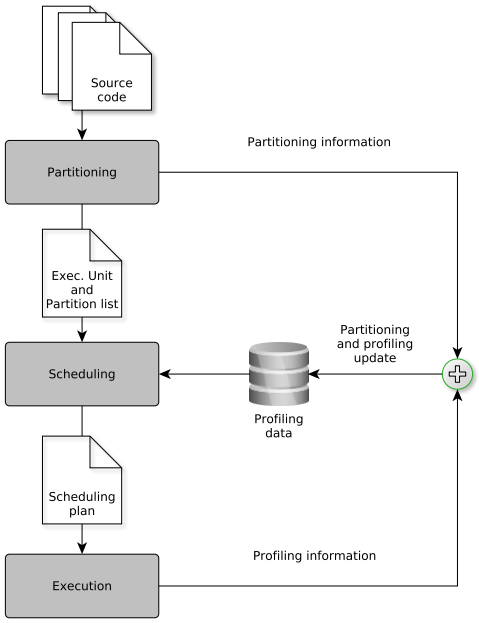


Figure 1: Algorithm work-flow.

Considering all possible parameters, that is, number of devices, number of execution units, and number of possible partitions, as the same input, the complexity of the first stage is O($n^4$). The complexity of the second stage is O($n^5$). The last stage depends entirely on the complexity of the executed partitions, so it has no fixed complexity. However, at the very least, the complexity would be O(n), assuming all the executed partitions are of complexity O(1).

### *2.2.1. Partition phase*

The procedure for the partition phase is outlined as follows:

1. The set of execution partitions $EP = \{ep_1, \ldots, kp_r\}$ is defined for each valid combination of execution unit, device, range and rate: $ep_i = (e_a, d_b, g_c, r_d)$
2. For each feasible pair of execution unit and device, at least one range is defined. The first range is associated with a set of partial rates from 0 to the maximum rate that can be stored in the device memory (called memory

bound of the device). If this maximum rate is not equal to 1, then several other ranges should be created with just two rates associated (0 and the memory bound of the device). The remaining data will be split according to the number of possible rates. The number of ranges should be enough to execute the whole dataset with one partition from each range. Details from this step are provided in Algorithm 1. Here, firstly the number of necessary ranges is computed. Afterwards, the rates are calculated as explained before. For example, in a two range dataset of 60 and 40, the first range will have two rates (0 or 60), whereas the second will be split according to the number of possible rates. Lastly, the partitions are established according to the number of ranges and rates.

3. An extended incompatibility matrix is created for partitions. Two partitions are incompatible if their execution units are incompatible or their kernels, devices and ranges are the same.
4. An extended dependency matrix is created for partitions. One partition depends on another if its execution unit also depends on the kernel from the other partition.

**Algorithm 1** Partitioning Phase: Algorithm.

```
1: let numRanges be the number of ranges required to cover the full data size of an execution unit in a device.
2: let MaxNumOfRates store either the number of rates associated with an execution unit. If the execution unit fits
   in a device in a single range, this variable will be the number of possible rates. If not, then it will store the rates
   for the last, smallest range.
3:
4: procedure executionUnitPartitionAlgorithm(EP,numEP, NumPossibleRates)
5:     numEP ← 0
6:     for i in 1..numExecutionUnits do
7:         for j in 1..numDevices do
8:             if F_{i,j} = 1 then
9:                 numRanges ← ⌈data size of execution unit e_i / memory bound of device d_j⌉
10:                if numRanges > 1 then
11:                    MaxNumOfRates ← memory bound of device d_j / (data size of execution unit e_i / NumPossibleRates)
12:                else
13:                    MaxNumOfRates ← NumPossibleRates
14:                for k in 1..numRanges do
15:                    if k = 1 then
16:                        for l in 0..MaxNumOfRates do
17:                            rate = (l / NumPossibleRates)
18:                            EP_{numEP} = (i,j,k,rate)
19:                            numEP = numEP + 1
20:                    else
21:                        EP_{numEP} = (i,j,k,0)
22:                        EP_{numEP+1} = (i,j,k, MaxNumOfRates/NumPossibleRates)
23:                        numEP = numEP + 2
```

*2.2.2. Scheduling phase*

Algorithms 2, 3, 4 and 5 show the scheduling procedure which can be outlined as follows:

1. A search tree is created to represent all the possible scheduling plans. The root node represents the original, empty scheduling plan. Intermediate nodes are partial scheduling plans. Lastly, the leaf nodes are complete scheduling plans. Each plan is associated with an execution time estimation. The parameter location array of the root node is initialized with initial parameters stored on the host.
2. A child node with a new plan is created by adding an execution unit or a partition to the scheduling plan of the parent node. Each parent node can have a child node for each execution unit or partition that:

- Is not included in the parent node plan.
- Is not incompatible with any other kernel partition within the parent scheduling plan.
- Does not depend on any other partition that is not included in or incompatible with the parent plan.

If there is none, the scheduling is complete and this is a leaf node.

3. When a new kernel partition is added, the following transfers should be obtained:
   - The transfers of all the input parameters from whatever device they are stored to the host.
   - The transfers of all the input parameters that are completely required, from the host to the selected device.
   - The transfers of all the input parameters that are partially required, from the host to the selected device.
   - The transfer of all the outputs that are partially generated from the selected device to the host.
4. The parameter location array and the partial parameters vector are updated accordingly. The transfer time required for those parameters is added to the execution time of the kernel partition.
5. The execution time associated to each node is equal to the finishing time for the latest partition plus the transfer time for the final output transfer operation. The finishing time of a new partition is computed by adding the estimated execution/transfer time of the new partition to the maximum of: 1) the finishing time of the latest partition executed on the same device. 2) the finishing time of the latest partition on which the new kernel partition depends. The time of the final transfer operation is the time required to send the final parameters from whatever device they are stored to the host.
6. Any scheduling plan where the added rates of the partitions with the same kernel is bigger than 1 is invalid, and its node must be discarded. Furthermore, if the partitions cover all the possible ranges of this kernel and the added rate is lower than 1 the plan is invalid and its node, too, must be discarded.
7. The scheduling plan with the lowest execution time will be selected for execution.

It is important to note that, as far a scheduling goes, there is very little difference between execution units and partitions. While it is true that the use of partitions allows to bypass restrictions such as the memory bounds of a device, in the end they are treated as smaller execution units that inherit the dependencies and incompatibilities of the execution unit that was split, and are also incompatible with said execution unit. In this way, the choice of splitting an execution unit, or running it completely, requires no special cases within the algorithm. In fact, an execution unit can be considered as a partition with a rate of 1 for the purpose of our algorithm.

#### *2.2.3. Execution phase*

The execution phase is the stage of the algorithm when the application is run. Algorithm 6 shows the procedure. The algorithm for the execution phase can be outlined as follows:

1. The execution of the execution units and/or partitions with their input and output transfers is done in background. A finalization event raises when it is done and an associated handle is executed.
2. The handler function is executed whenever an OpenCL finalization event raises. Firstly, it obtains which partition was finalized. Then, it audits all the free devices and check which one is the next execution unit or partition to be executed on each one. If these only depend on other execution units or partitions that have already finished, they are executed in background and the same handle is associated to the finalization event.
3. When the last execution unit or partition is completed, the handler will finish the execution.
4. The application begins executing the handler function directly in order to launch one task per device (if possible).

For the implementation of this phase we have used the OpenCL API. For example, the finalization event raised will be an OpenCL finalization event.

**Algorithm 2** Scheduling Phase: Algorithm.

```
1: let bestTime store the best execution time of a plan at any given moment.
2: let bestSP store the best scheduling plan at any given moment.
3: let queue store newNodes, which contain partial scheduling plans, except for lead nodes.
4: let ListNextEP list the next execution units elegible to be run after the point in time scheduled in node
5:
6: procedure SCHEDULINGALGORITHM(bestSP, bestTime)
7:     bestTime ← MAX
8:     queue ← INSERT(queue, emptyNode)
9:     while not EMPTY(queue) do
10:        node ← EXTRACT(queue)
11:        ListNextEP ← LISTNEXTEXECUTIONUNITPARTS(node)
12:        for i in ListNextEP do
13:            newNode ← ADDEP(node, EP_i)
14:            if ISVALIDSCHED(newNode) then
15:                if not ISFINALSCHED(newNode) then
16:                    queue ← INSERT(queue, newNode)
17:                else
18:                    if bestTime >GETTIME(newNode)then
19:                        bestSP ← SCHEDPLAN(newNode)
20:                        bestTime ← GETTIME(newNode)
```

## 3. Evaluation

The proposed algorithm has been evaluated using several use cases. They are pieces of OpenCL code calling one or more kernels. In this experiment, the number of kernels ranges from one to several (twenty-seven at most). They have been executed on an architecture with several computing devices. The selected examples, which are listed below, are considered to be representative from many kernel-based programs. Experimental results show the improvement obtained with the proposed scheduling as opposed to more typical solutions, such as running the program in a single device, or not partitioning the input and output data.

### *3.1. Benchmarks and target platform*

Table 1 describes the testbed hardware platform used in the evaluation process. AMD's driver [13] was used instead of Intel's OpenCL driver because the latter produced results with a big variance, as opposed to the AMD's more stable results. For this paper, we have used AMD's driver, though the issue with Intel's driver will be addressed in future work.

| | Intel CPU | AMD GPU | Intel Xeon Phi |
|---|---|---|---|
| Model | Intel®Xeon® CPU E5-2695 | AMD®Radeon® R9 290 series | Xeon Phi® coprocessor 3120 series |
| Core clock | 1.2 GHz | 1.0 GHz | 1.1 GHz |
| Computing units | 24 | 2816[2] | 224 |
| Memory | 128 GiB | 4 GiB | 6 GiB |
| OpenCL driver | AMD-APP-SDK-v2.8 | | OpenCL™Runtime 14.2 |
| OpenCL supported version | 1.2 | | |

Table 1: Test platform.

The benchmarks used for evaluation our algorithm are the following:

[2] `Stream` processors.

**Algorithm 3** Scheduling Phase: Auxiliary functions.

let *usedEP* store those partitions already used in the scheduling plan stored in *node*.
let *IncompEP* store those partitions that are incompatible with the scheduling plan stored in *node*.
let *RemainEP* store those partitions that can still be used after the point in the scheduling plan stored in *node*.
let *RedPP* store the list of dependencies between remaining partitions.
let *IndepEP* store those thos partitions that are have no dependencies.

**function** LISTNEXTEXECUTIONUNITPARTS (node)
    $usedEP \leftarrow usedEP_i = 1 \Leftrightarrow EP_i \in$ SCHEDPLAN(*node*)
    $IncompEP \leftarrow IncompEP_i = IP_{i,0} \wedge usedEP_0 \vee \cdots \vee IP_{i,n} \wedge usedEP_n$
    $RemainEP \leftarrow RemainEP_i = \neg(IncompEP_i \vee usedEP_i)$
    $RedPP \leftarrow RedPP_{i,j} = PP_{i,j} \wedge RemainEP_i \wedge RemainEP_j$
    $IndepEP \leftarrow IndepEP_i = \neg(RedPP_{i,0} \vee \cdots \vee RedPP_{i,n})$
    $ListNextEP \leftarrow ListNextEP_i = RemainEP_i \wedge IndepEP_i$
    **return** *ListNextEP*
**function** ISVALIDSCHED(node)
    **for** $EP_i$ in SCHEDPLAN(*node*) **do**
        $(execution\ unit, device, range, rate) \leftarrow EP_i$
        $usedRanges_{execution\ unit,device} \leftarrow usedRanges_{execution\ unit,device} + 1$
        $usedRates_{execution\ unit} \leftarrow usedRates_{execution\ unit} + rate$
    $Overflow \leftarrow \exists_i \parallel (usedRates_i > 1)$
    $Underflow \leftarrow \exists_i \forall_j \parallel (usedRates_i < 1) \wedge (usedRanges_{i,j} = \lceil data\ size_i / memory\ bound_j \rceil)$
    **return** $\neg(Overflow \vee Underflow)$
**function** ISFINALSCHED(node)
    $usedEP \leftarrow usedEP_i = 1 \Leftrightarrow EP_i \in$ SCHEDPLAN(*node*)
    $IncompEP \leftarrow IncompEP_i = IP_{i,0} \wedge usedEP_0 \vee \cdots \vee IP_{i,n} \wedge usedEP_n$
    $RemainEP \leftarrow RemainEP_i = \neg(IncompEP_i \vee usedEP_i)$
    **return** $\forall_i \parallel RemainEP_i = 0$

let *tExec* store the amount of time it would take to transfer input and output data, as well as execute, a given partition.
let *tEndDepEP* store the times it would take to meet each of a partition's dependencies.
let *tStartEP* store the worst-case starting time for a partition.

**function** GETTIME(node)
    **for** *i* in SCHEDPLAN(*node*) **do**
        $tExec \leftarrow$ GETEXECUTIONTIME($EP_i$)
        $tExec \leftarrow tExec +$ GETTRANSFERTIME($EP_i$)
        $tEndDepEP \leftarrow tEndDepEP_r = PP_{i,r} \cdot tEndEP_r$
        $tStartEP_i = max(tEndDev_{EP_i.device}, max_r(tEndDepEP_r))$
        $tEndEP_i = tStartEP_i + tExec$
        $tEndDev_{EP_i.device} = tEndEP_i$
    **return** $max_{dev}(tEndDev_{dev}) +$ GETFINALTRANSFERTIME()

**Algorithm 4** Scheduling Phase: Transfer Auxiliary Functions 1.

This procedure calculates the transfer time necessary to move input and output data from one step to the next.
**procedure** getTransferTime($EP_k$)
    ($execution\ unit, device, range, rate$) $\leftarrow EP_k$
    transfersPerUnitExec($EP_k$)
    UpdateParamLocation($EP_k$)
    **for** i **do** in 1..$numDevices$
        $aux \leftarrow aux_i = 1 \Leftrightarrow CompTransfAnyHost_i == i$
        $time = time$ + transferTime($i, aux$)
    $time = time$ + transferTime($device, compTransfHostDev$)
    $time = time$ + transferTime($device, rateTransfHostDev$)
    $time = time$ + transferTime($device, rateTransfDevHost$)
    **return** $time$

This procedure calculates just the transfer time to gather the output data in the host at the last stage.
**procedure** getFinalTransferTime()
    $aux \leftarrow aux_i = FinalHostParams_i - (paramLocation_{host,i} \wedge FinalHostParams_i)$
    **for** j **do** in 1..$numDevices$
        $paramsInDev \leftarrow paramsInDev_i = aux_i \wedge paramLocation_{j,i}$
        $aux \leftarrow aux_i = aux_i - paramsInDev_i$
        $CompFinalTransf \leftarrow CompFinalTransf_i = CompFinalTransf_i + (paramsInDev_i * i)$
        $time = time$ + transferTime($j, paramsInDev$)
    **return** $time$

This procedures calculates the devices where the input and output variables must reside in, in order to execute the next step.
**procedure** transfersPerUnitExec($EP_k$)
    ($execution\ unit, device, range, rate$) $\leftarrow EP_k$
    **if** $rate = 0$ **then**
        **return** 0
    $Inputs \leftarrow Inputs_i = CompInputs_i \vee RateInputs_i$
    $InputsReady \leftarrow InputsReady_i = paramLocation_{device,i} \wedge Inputs_i$
    $InputsMove \leftarrow InputsMove_i = inputs_i - InputsReady_i$
    **if** $rate = 1$ **then**
        $compTransfHostDev_{EP_k} \leftarrow InputsMove$
    **else**
        $compTransfHostDev_{EP_k} \leftarrow compTransfHostDev_{EP_k,i} = CompInputs_i \wedge InputsMove_i$
        $rateTransfHostDev_{EP_k} \leftarrow rateTransfHostDev_{EP_k,i} = (RateInputs_i \wedge InputsMove_i) * rate$
        $rateTransfDevHost_{EP_k} \leftarrow rateTransfDevHost_{EP_k,i} = RateOutputs_i * rate$
    $InputsMoveToHost \leftarrow InputsMoveToHost_i = InputsMove_i - (paramLocation_{host,i} \wedge InputsMove_i)$
    $aux \leftarrow InputsMoveToHost$
    **for** j in 1..$numDevices$ **do**
        $paramsInDev \leftarrow paramsInDev_i = aux_i \wedge paramLocation_{j,i}$
        $aux \leftarrow aux_i = aux_i - paramsInDev_i$
        $CompTransfAnyHost \leftarrow CompTransfAnyHost_i = CompTransfAnyHost_i + (paramsInDev_i * i)$

**Algorithm 5** Scheduling Phase: Transfer Auxiliary Functions 2.

This procedure updates the last known location of the input and output parameters.
**procedure** UPDATEPARAMLOCATION($EP_k$)
    $(execution\ unit, device, range, rate) \leftarrow EP_k$
    $paramLocation_{host} \leftarrow paramLocation_{host,i} = paramLocation_{host,i} \vee InputsMoveToHost_i$
    $paramLocation_{device} \leftarrow paramLocation_{device,i} \vee compTransfHostDev_i \vee CompOutputs_i$
    **if** $rate = 1$ **then**
        $paramLocation_{device} \leftarrow paramLocation_{device,i} \vee RateOutputs_i$
    $rateHostParameters \leftarrow rateHostParameters_i + rateTransfDevHost_i$
    $completedParams \leftarrow completedParams_i = 1 \Leftrightarrow rateHostParameters_i == 1$
    $rateHostParameters \leftarrow rateHostParameters_i - completedParams_i$
    $paramLocation_{host} \leftarrow paramLocation_{host,i} + completedParams_i$

**Algorithm 6** Execution Phase: Algorithm.

let $EP_r$ store the next partition scheduled for a given device through procedure *nextEP*.
let $ListDep_{EP_r}$ store the list of dependencies for partition r that have yet to be met.

let procedure *FINISH* check whether or not a device is done executing a partition.
let procedure *EXTRACT* remove the finished partition from the scheduling plan.
let procedures *INPUTTRANSFERS* and *OUTPUTTRANSFERS* transfer data to and from a device, respectively.
let procedure *EXEC* launch a partition in a device.

**procedure** SCHEDULER()
    HANDLER()
    **while** $schedPlan \neq \phi$ **do**
        HANDLER()
**procedure** HANDLER()
    **for** i in 1..$numDevices$ **do**
        **if** FINISH($device_i$) **then**
            EXTRACT($schedPlan, device_i.EP$)
            $EP_r \leftarrow$ NEXTEP($schedPlan, device_i$)
            $ListDep_{EP_r} \leftarrow \forall_{EP_j \in schedPlan} || (ep_r, ep_j) \in R_{PP}$
            **if** $ListDep_{EP_r} = \phi$ **then**
                INPUTTRANSFERS($EP_r, device_i$)
                EXEC($EP_r, device_i$)
                OUTPUTTRANSFERS($EP_r, device_i$)
                $device_i.EP \leftarrow EP_r$

- Stencil [2]
- Transitive closure algorithm.
- S3D (with is a Navier-Stokes equations implementation) [3]
- LU Decomposition (LUD) [3]
- GEMM [4]
- Monte Carlo simulation [4]
- Median filter [4]
- Bitonic Sort [4]

The benchmarks Stencil, Transitive closure and S3D are explained in detail, while the rest will be presented at the end of the evaluation in order to summarize the model evaluation.

The implementation of all these examples is based on a core code composed of one or several kernels that are executed once or several times within a loop. This is the way most of the state-of-the art benchmarks for kernel-based programs are implemented. Our approach on those loop-based algorithms is to focus on scheduling one iteration in the best possible way. Then, each iteration is executed using the selected scheduling. This approach compromises between obtaining a good scheduling performance and reducing the problem complexity to get the scheduling plan on a reasonable time. Furthermore, in many cases, each iteration depends completely on the results of the previous iteration, and consequently each iteration can be analysed as a separated scheduling problem. Those applications need to have an implementation in OpenCL or other kernel-based framework. It is for this reason that these benchmarks have been chosen, since it saves the effort of coding new versions from scratch.

*Stencil benchmark.* The stencil benchmark used in this work is adapted from the 3D stencil found in the Rodinia benchmark [11, 12]. It is used to make a 2D stencil, where the program can be partitioned amongst the devices that make up the heterogeneous platform. The benchmark has a single kernel that performs the stencil algorithm. The partitioning can be achieved by splitting the input matrix at the row level. This also applies to the output matrix. In the case of partitioning, the host will send the input partitions, and receive the output ones.

*Transitive closure benchmark.* The transitive closure benchmark implements the transitive closure of a binary relationship among the elements from a set. The relationship is defined using a binary square matrix. The algorithm is a conditional loop, where each iteration performs three steps: a binary multiplication of matrices, a comparison of matrices and an union (binary OR) of matrices. The three operations are implemented as independent kernels that can be executed in parallel on each iteration. A previous study shows that the multiplication kernel is the one that takes the longest time to run. For this reason, this kernel can be partitioned across the devices, whereas the other two are executed completely on one device. The input of the multiplication kernel is split by fragmenting all the matrices at the row level. To allow this, it is necessary that the second input matrix is transposed before calling the kernel.

*S3D benchmark.* The S3D benchmark is an OpenCL/CUDA implementation of the Navier-Stokes equations for a regular 3D domain, used to simulate combustion. The benchmark is composed of 27 kernels that have several dependencies between them. Most of the computation is held by two groups of parallel kernels (8 kernel each) that are, in fact, a manual partition of two bigger kernels. Because of this reason, there is no point on performing any more partitioning. Each kernel is executed completely on one device.

*LU Decomposition (LUD).* The LU Decomposition is an algorithm to calculate the solutions of a set of linear equations. The LUD kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix.

[2] Rodinia benchmark suite [11, 12]
[3] SHOC benchmark suite [10]
[4] Intel Code OpenCL Samples [26]

*GEMM.* An optimized General Matrix Multiply (GEMM) benchmark is included in order to use efficiently the internal hardware characteristics of the Intel(R) Xeon Phi(tm). This implementation, extracted from the Intel OpenCL Code Samples [26] optimizes the trivial matrix multiplication nested loop version to utilize the memory cache more efficiently by introducing a well-known practice as tiling (or blocking), where matrices are divided into blocks, and the blocks are multiplied separately to maintain better data locality.

*Monte Carlo simulation.* This benchmark simulates European stock option pricing through Monte Carlo algorithm. The option price calculation is performed using Black-Scholes stock pricing model.

*Median filter.* Median filter is a non-linear digital filtering technique used in order to reduce noise.

*Bitonic Sort.* Bitonic sort is a parallel algorithm for sorting. It is also used as a construction method for building a sorting network.

### *3.2. Results*

The evaluation process for the stencil and the transitive closure benchmarks is performed as follows. Firstly, profiling data is gathered for each kernel from the benchmark. Each kernel is executed independently on each device using inputs of different sizes. This profiling information will be used to interpolate the expected performance for each kernel with different input sizes and different levels of partitioning. Also the information about the incompatibility, dependency and feasibility of each kernel is gathered. Secondly, the proposed algorithm is executed for a selected problem size and partition pattern. The resulting scheduling plan is implemented on the benchmark and executed to obtain its performance. Finally, all the possible schedules considered in the algorithm are also executed and their performance results are compared to the selected solution.

The evaluation process for the S3D benchmarks is almost the same as the former ones. The only difference is that the selected scheduling plan cannot be compared to all the other schedules considered because the amount of tests would make it unfeasible ($2^{27}$ combinations). Therefore, we compare the chosen schedule with the original configurations provided in the benchmark: all kernels in either device.

The stencil benchmark contains just one kernel, which is executed once. In this case the incompatibility and dependency matrices are 1x1 zero matrices. Also the feasibility matrix is a 1x2 matrix flled with ones. Profiling data is obtained for data sizes 256, 512, 1024, 2048, 4096 and 8192. These sizes correspond with the sides of the input matrices, such that base size 1024 corresponds with a matrix of real size 1024x1024 elements. We also consider all the corresponding 1/4, 2/4 and 3/4 fractions of each base size (For example 64, 128 and 192 are the fractions for the 256 data size). Then scheduling plans are obtained for data sizes 256, 512, 1024, 2048, 4096 and 8192. On each case the partition pattern used fragments the problems size in 16 slices. Therefore, the partition pattern goes from 0/16 of the total size to 16/16. Table 2 shows the scheduling plan obtained for each one of the data sizes considered. The reason for this process is twofold. Firstly, splitting in four parts seems to be the norm in similar state-of-the-art tools. The closest example is that of O'Boyle's static approach [18]. Secondly, we wanted to make sure that the partition sizes were multiples of the amount of partitions, and using interpolation up to 16 parts, we can also stress-test our tool.

| DataSize | 256 | 512 | 1024 | 2048 | 4096 | 8192 |
|---|---|---|---|---|---|---|
| CPU partition | 16/16 | 16/16 | 16/16 | 4/16 | 4/16 | 4/16 |
| GPU partition | 0/16 | 0/16 | 0/16 | 12/16 | 12/16 | 12/16 |
| ACC partition | 0/16 | 0/16 | 0/16 | 0/16 | 0/16 | 0/16 |

Table 2: Stencil benchmark: Selected scheduling plans.

Finally, each valid scheduling combination considered on previous configurations (data sizes from 256 to 8192 with a partition pattern from 0/16 to 16/16) is executed, and all the performance results are compared with the performance of the selected solution.

The results for the stencil benchmark are shown in Figure 2 and Figure 3. Figure 2 shows the percentile of the performance of the selected solution considered within the whole performance distribution of all the valid scheduling

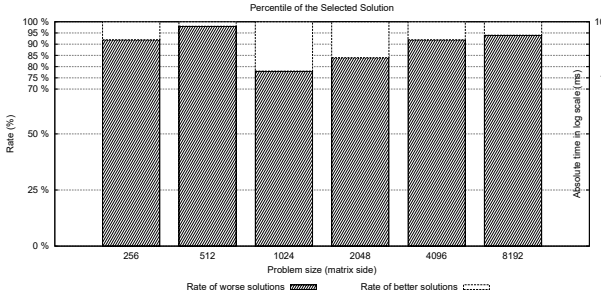


Figure 2: Stencil benchmark: percentile of the obtained solution.

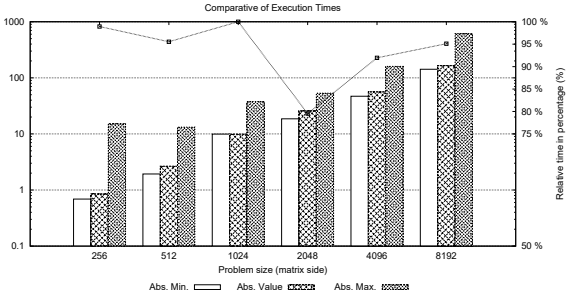


Figure 3: Stencil benchmark: performance comparison.

combinations. Figure 3 compares the performance of the best and the worst configuration to the performance of the selected solution. The comparisons are done using absolute performances (best, worst and selected plan) as well as relative performance from the selected solution compared with the best and the worst ones (the bigger the percentage, the closer to the best solution it is).

Several observations can be drawn from these results. Firstly, for small data sizes, the best option is to execute the whole kernel on the CPU. The reason behind this is that the data transfers to other devices cost outweights other benefits. Besides, the problem size is not enough to take advantage of the level of parallelism that other devices offer. Secondly, for bigger data sizes, the GPU, and to some extent, the accelerator, exhibit a better performance ratio, but not enough to execute the whole kernel on the GPU. Therefore, the execution seems to settle down on a fixed execution ratio of 1/4 on the CPU and 3/4 on the GPU (at least for considered data sizes). Figure 2 shows that the performance of the selected solution is always very near to the best one. Besides, Figure 2 shows that there are very few solution (among all the solutions considered) with a better performance. In most cases the small discrepancy is due to the performance variability of the same scheduling over several runs. However, 1024 and 2048 data sizes are slightly different because in those cases the difference between using only the CPU and a 1/4 CPU vs. 3/4 GPU usage is not as big. For example, for data size 1024 the percentile is lower but the performance is nearer to the best solution. This is because, in this case there are more solutions with a similar performance. A similar situation happens for data size 2048 but in this case performance ranking is worse due to the higher performance variability of the GPU compared to the CPU. Finally, in all tested cases the accelerator has worse results than CPU and GPU. For that, the stencil model never takes into account the accelerator to process data. The reason behind this is that, while it is true that the accelerator may attain a better performance than the CPU, the cost of transferring data from the host to the accelerator negates most benefits. Furthermore, the GPU is better oriented towards data parallelism, and this means that, in the event that some kernel or partition can be delegated to another device, the GPU will still be a better choice than the accelerator. It is possible, however, that in cases of massive parallelism, where many kernels can be executed simultaneously, the accelerator will be used.

The transitive closure benchmark consists of three kernels, executed several times on a conditional loop. Each iteration depends on the result of the previous one. Consequently, we consider only the execution of one iteration. In this case, the incompatibility and dependency matrices are 3x3 zero matrices because all kernels are independent and required for each execution at the same time. Also, the feasibility matrix is a 3x2 matrix full of ones (all kernels can be executed on all the devices). Profiling data for the three kernels is obtained for data sizes 256, 512, 1024, 2048, 4096 and 8192. The comparison and the union kernels are executed only on CPU or GPU so no fractions are considered for the profiling data. For the multiplication kernel, 1/4, 2/4 and 3/4 fractions are also collected (For example 64, 128 and 192 are the fractions for the 256 data size). Scheduling plans are obtained for data sizes 128, 256, 384, 512, 768, 1024, 1536, 2048, 3072, 4096, 6144 and 8192. For the multiplication kernel, the partition pattern used fragments the problems size in 16 slices. Therefore, the partition pattern goes from 0/16 of the total size to 16/16. Table 3 shows the scheduling plan obtained for each one of the data sizes considered.

Finally, each valid scheduling combination considered in previous configurations (data sizes from 128 to 8192 with a partition pattern form 0/16 to 16/16 on the multiplication kernel) is executed, and all the performance results are compared with the performance of the selected solution.

Results for the transitive closure benchmark are shown in Figure 4 and Figure 5. Figure 4 shows the percentile of

| DataSize | 128 | 256 | 384 | 512 | 768 | 1024 | 1536 | 2048 | 3072 | 4096 | 6144 | 8192 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Multi. Kernel | CPU | CPU | CPU | 3/4 CPU | CPU | CPU | CPU | CPU | CPU | CPU | CPU | CPU |
| Union Kernel | GPU | GPU | GPU | CPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU |
| Comp. Kernel | CPU | CPU | CPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU |

Table 3: Transitive closure: selected scheduling plans.

| Kernels | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Case 24 | CPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU |
| Case 32 | CPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU |
| Case 40 | CPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU |
| Case 48 | CPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU |
| Kernels | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | |
| Case 24 | GPU | GPU | GPU | GPU | GPU | CPU | GPU | GPU | GPU | GPU | GPU | GPU | CPU | |
| Case 32 | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | CPU | |
| Case 40 | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | CPU | |
| Case 48 | GPU | GPU | GPU | GPU | GPU | GPU | GPU | GPU | CPU | CPU | CPU | CPU | CPU | |

Table 4: S3D: Selected scheduling plans.

the performance of the selected solution considered within the whole performance distribution of all the valid scheduling combinations. Figure 5 compares the performance of the best and the worst configuration with the performance of the selected solution. The comparisons are done by absolute performance (best, worst and selected schedule), as well as the relative performance of the selected solution compared to the best and the worst ones (the bigger the percentage, the closer to the best solution).

In this case observations from results are also relevant. Firstly, the multiplication kernel is almost always executed solely on the CPU. This is because the algorithm used is much more efficient on the CPU than on the GPU. Another kernel implementation would probably yield different results. As a consequence, the other two kernels are executed on the GPU in most cases. Only when the data size is small, one of these kernels is also executed on the CPU, as in those cases, avoiding the data transfers to the GPU is better than a greater parallelism degree. Figure 4 shows that there are very few solutions (amongst all the solutions considered) that exhibit a better performance. Besides, Figure 5 shows that the performance of the selected solution is always very near to the best one. In most cases, the small discrepancy is due to the performance variability of the same scheduling over several executions. However 384 and 512 data sizes are slightly different because in those cases the difference between using mostly the CPU, or using the CPU only for the multiplication kernel is not as big. Therefore, there are more solutions with a similar performance and the corresponding percentile and execution time is a little worse.

The S3D benchmark consists of 27 kernels executed several times on a conditional loop. As with the transitive

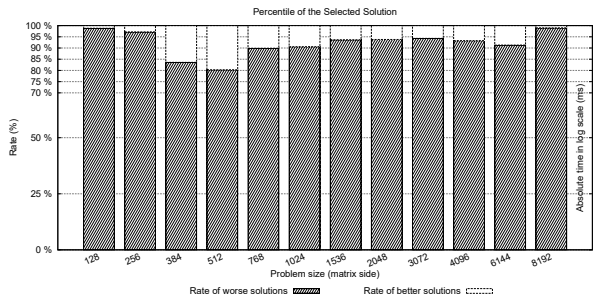


Figure 4: Transitive closure benchmark: percentile of the obtained solution.

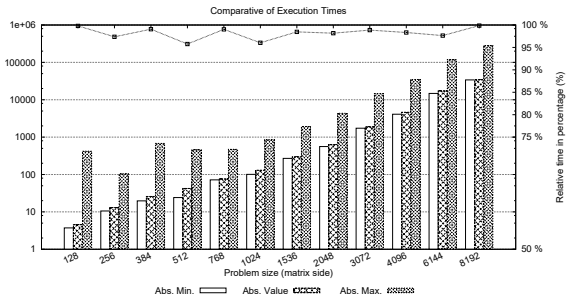


Figure 5: Transitive closure benchmark: performance comparison.

closure benchmark, we only consider the execution of a single iteration. The dependency matrix for S3D benchmark is by far the most complex of the three, since there are 27 kernels, separated in two distinct phases, and there are many dependencies between them. On the other hand, the incompatibility matrix, while bigger than the previous ones (27x27), is still a zero matrix because all kernels are compatible and required for each execution at the same time. The same happens with the feasibility matrix, because all kernels can be executed on all the devices. Profiling data for all kernels is obtained for base data sizes 24, 32, 40 and 48. These sizes represent the basic size factor that is applied to all input and output parameters. All kernels are executed only on CPU or GPU so no fractions are considered for the profiling data. Then, scheduling plans are obtained for the same data sizes, in hope that the scheduling tool will distribute the independent kernels between devices. Table 4 shows the scheduling plan obtained for each one of the data sizes considered. It is worth to notice that most of the kernels are executed on the GPU and only a few of them are executed on the CPU.

Results show the comparison between the schedule obtained with the proposed algorithm, the execution of all the kernels on CPU and the execution of all the kernels on GPU (see Figure 6). Results suggest executing everything in GPU is, probably, one of the options with a better performance, while executing everything in CPU yields much worse results. However, sharing the computation tasks between both devices improves the results in most cases even though sharing the computation involves many more data transfers than just using the GPU. It is remarkable that the GPU-only solution is a very good one, so the improvement is not very significant and can be lost due to variability of the executions and the error range of the algorithm. The obtained scheduling outperforms the GPU-only execution for cases with base data size 24, 32 and 48, although they are very close. However, in one case, with base data size of 40, the result obtained is worse than the GPU-only execution.

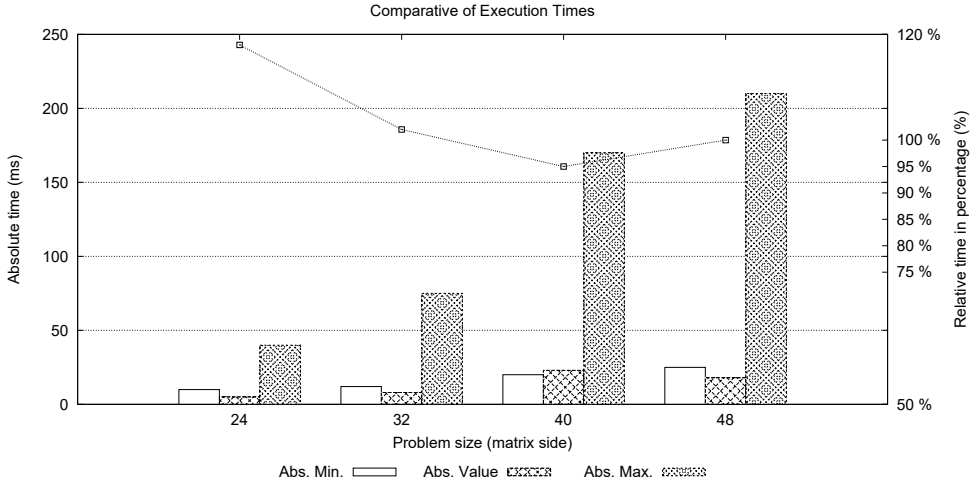


Figure 6: S3D benchmark: performance comparison.

Figure 7 shows a summary that compares the results running a benchmark on a concrete device with the prediction model result. It includes all the benchmarks presented before. It shows that the model results are close or better that the best compared result.

As a rule of the thumb, the accelerator outperforms the other two devices in those cases where the execution units make use of vector instructions. Such examples are Intel's benchmarks, such as GEMM and Bitonic Sort. If the kernels don't make use of vector instructions, then either the GPU or the CPU yield better results, depending on the kind of problem. GPU yields better results in purely SIMD problems, specially for bigger input data. For the rest of the cases, it is the CPU that gives the best performance. Our model manages to outperform these three separately because it considers execution and transfer times, as well as execution unit partitioning and multi-device scheduling.

One particular case where the model actually worsens the results is the GEMM benchmark for a base size of 256 elements. The reason for this is that, with small data sizes, and because of the linear interpolation, it is possible that the model predicts a partition much too small to actually give some benefit. For this reason, a threshold is considered

in small cases. This threshold cannot be too high, in order to avoid discarding too many possibilities. The GEMM benchmark was the single case where the prediction model fell outside the threshold, and still give a bad prediction.

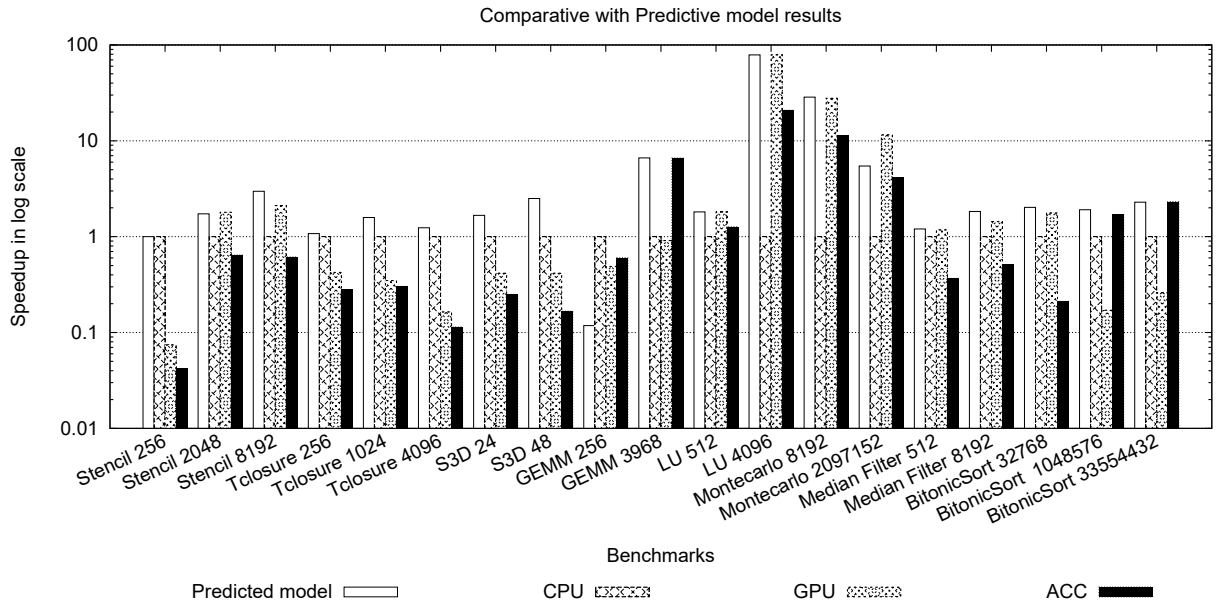


Figure 7: Summary of model prediction results.

## 4. Related Work

Some frameworks have been developed to enhance the performance of parallel applications over heterogeneous architectures. In [14], authors propose an OpenCL framework that treats multiple GPUs as a single computing device. This single computing device allows an OpenCL application written for a single GPU portable to a multi GPU system. A similar approach is used by *Load Balancing for OpenCL* library `lbcl` [15]. SkePU [16] is an open-source skeleton programming framework for multi-core CPUs and multi-GPU systems. It is a C++ template library with six data-parallel and one task-parallel skeletons, two generic container types, and support for execution on multi-GPU systems both with CUDA and OpenCL. `LibWater` [17] is another framework that proposes an uniform approach, based on OpenCL, for programming distributed heterogeneous computing systems. These libraries allow to execute over multiple-devices, but these solutions are more difficult to use on legacy code, where a complete transformation of code is needed.

Static partitioning of heterogeneous parallel applications are used to deploy the kernels on the computing devices. Usually, it is hand-made operation made by developers. To make it automatically there are two main options to perform this operation: qualitative model, where the partitioning is based on code characteristics or quantitative model, where performance results from previous executions are used to predict the way to perform the partitioning.

In [18] authors use a qualitative static partitioning model based on characteristics extracted from OpenCL kernels (i.e. memory access patterns, McCabe's complexity number, number of numerical operations). This static partitioning model uses a machine-learning approach to predict the best device for an OpenCL kernel. The model could make the decision to execute a kernel in one device (GPGPU or CPU) or split the kernel in two parts to execute in parallel. Other works [19] use a similar solution including more dynamic features. However, the models are not easily scalable and adaptable in case of heterogeneous architecture changes, because the predictive model needs to be trained again. This increase the system initialization overhead.

Other approaches are based in quantitative models. Qilin compiler [20] divides parallel loops between CPUs and GPUs using regression-based prediction at run-time through a specific API supported by the authors. This approach

needs to transform the legacy code to a new one. Other static quantitative model is presented in [21], but it does not consider to split kernel into different devices. In [23], authors present a quantitative approach based on the computation of a threshold to evaluate the best GPU/CPU mapping. However, this method does not consider more than two compute devices.

## 5. Conclusions

In this paper we have proposed an algorithm to obtain a sub-optimal partitioning and scheduling for a set of kernels that compose an application. The execution plan is obtained using profiling information from the kernels. The proposed algorithm creates a search tree with all the possible plans and then selected the one with the best performance. The paper also proposes a generic algorithm to execute the kernels using the scheduling plan obtained before.

The evaluation is done using several benchmarks of different complexity: single kernel benchmarks, multi-kernel benchmarks and many-kernel benchmarks. The obtained results show that the partitioning and scheduling plans predicted by the proposed algorithm is one of the best among a huge set of representative scheduling plans. Besides, their performance is very near to the best performance obtained within this set.

Future works will include the evaluation of the algorithm with more and different computing devices. Besides, searching for the lower power consumption could be included in the algorithm. With such improvements we would moving towards a multi-objective algorithm. Finally, profiling could be substituted by estimations performed from qualitative measurements from the code. These would allow to obtain initial results without any experimental data from the desired platform.

We are also working on automated transformations to different components of a parallel heterogeneous architecture in the context of the European Project REPARA [24]. An integrated development environment is available in the form of an Eclipse based tool called cevelop [25], plus a range of transformation plug-ins.

## Acknowledgment

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n. 609666 [24].

## References

[1] Top500, Supercomputer sites, http://top500.org (Sep. 2014).
[2] V. Kindratenko, P. Trancoso, Trends in high-performance computing, Computing in Science and Engineering 13 (2011) 92–95.
[3] Lawrence Latif, Acceleware says most CUDA applications are memory bound, http://www.theinquirer.net/inquirer/news/2255592/acceleware-says-most-cuda-applications-are-memory-bound (Mar. 2013).
[4] C. Gregg, K. Hazelwood, Where is the data? why you cannot debate cpu vs. gpu performance without the answer, in: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 134–144.
[5] Sanchez, L.M., Fernandez, J., Sotomayor, R., Escolar, S., Garcia, J.D.: A comparative study and evaluation of parallel programming models for shared-memory parallel architectures. New Generation Computing 31(3), 139–161 (2013)
[6] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. Dubey, Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu, SIGARCH Comput. Archit. News 38 (3) (2010) 451–460.
[7] OpenACC, Open Accelerators, http://www.openacc-standard.org/ (Sep. 2014).
[8] OpenMP, Open Multi-Processing, http://openmp.org/wp/ (Sep. 2014).
[9] The Khronos Group, The OpenCL Specification, http://www.khronos.org/ (Sep. 2014).
[10] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, J. S. Vetter, The scalable heterogeneous computing (shoc) benchmark suite, in: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU '10, ACM, New York, NY, USA, 2010, pp. 63–74.
[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, 2009, pp. 44–54.
[12] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, K. Skadron, A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads, in: Workload Characterization (IISWC), 2010 IEEE International Symposium on, 2010, pp. 1–11.
[13] AMD, AMD-APP driver 2.8, http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/.

[14] J. Kim, H. Kim, J. H. Lee, J. Lee, Achieving a single compute device image in opencl for multiple gpus, SIGPLAN Not. 46 (8) (2011) 277–288.
[15] C. de la Lama, P. Toharia, J. Bosque, O. Robles, Static multi-device load balancing for opencl, in: Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on, 2012, pp. 675–682.
[16] U. Dastgeer, J. Enmyren, C. W. Kessler, Auto-tuning skepu: A multi-backend skeleton programming framework for multi-gpu systems, in: Proceedings of the 4th International Workshop on Multicore Software Engineering, IWMSE '11, ACM, New York, NY, USA, 2011, pp. 25–32.
[17] I. Grasso, S. Pellegrini, B. Cosenza, T. Fahringer, Libwater: Heterogeneous distributed computing made easy, in: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, ACM, New York, NY, USA, 2013, pp. 161–172.
[18] D. Grewe, M. F. P. O'Boyle, A static task partitioning approach for heterogeneous systems using opencl, in: Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software, CC'11/ETAPS'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 286–305.
[19] K. Kofler, I. Grasso, B. Cosenza, T. Fahringer, An automatic input-sensitive approach for heterogeneous task partitioning, in: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, ACM, New York, NY, USA, 2013, pp. 149–160.
[20] C.-K. Luk, S. Hong, H. Kim, Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, in: Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, ACM, New York, NY, USA, 2009, pp. 45–55.
[21] O. E. Albayrak, I. Akturk, O. Ozturk, Improving application behavior on heterogeneous manycore systems through kernel mapping, Parallel Computing 39 (12) (2013) 867–878.
[22] J. Shen, A. L. Varbanescu, H. Sips, Look before you leap: Using the right hardware resources to accelerate applications, in: Proceedings of IEEE 16th International Conference on High Performance Computing and Communications (HPCC 2014), 2014, p. 9.
[23] J. Shen, A. L. Varbanescu, P. Zou, Y. Lu, H. J. Sips, Improving performance by matching imbalanced workloads with heterogeneous platforms, in: 2014 International Conference on Supercomputing, ICS'14, Muenchen, Germany, June 10-13, 2014, 2014, pp. 241–250.
[24] REPARA website (2015), `http://repara-project.eu/`
[25] cevelop website (2015), `http://cevelop.com/`
[26] Intel OpenCL Code Samples (2015), `https://software.intel.com/en-us/intel-opencl-support/code-samples`