Automatic OpenCL code generation for multi-device heterogeneous architectures

Pei Li*, Elisabeth Brunet*, François Trahay*, Christian Parrot*, Gaël Thomas* and Raymond Namyst†
*Telecom SudParis, *firstname.lastname@telecom-sudparis.eu*
†University of Bordeaux 1, *raymond.namyst@inria.fr*

***Abstract*—Using multiple accelerators, such as GPUs or Xeon Phis, is attractive to improve the performance of large data parallel applications and to increase the size of their workloads. However, writing an application for multiple accelerators remains today challenging because going from a single accelerator to multiple ones indeed requires to deal with potentially non-uniform domain decomposition, inter-accelerator data movements, and dynamic load balancing. Writing such code manually is time consuming and error-prone. In this paper, we propose a new programming tool called STEPOCL along with a new domain specific language designed to simplify the development of an application for multiple accelerators. We evaluate both the performance and the usefulness of STEPOCL with three applications and show that: (i) the performance of an application written with STEPOCL scales linearly with the number of accelerators, (ii) the performance of an application written using STEPOCL competes with a handwritten version, (iii) larger workloads run on multiple devices that do not fit in the memory of a single device, (iv) thanks to STEPOCL, the number of lines of code required to write an application for multiple accelerators is roughly divided by ten.**

***Keywords*-Heterogeneous architectures, OpenCL, Code generation, Accelerators**

I. Introduction

Numerical simulations in Physics, Chemistry or Health relying on hardware accelerators such as GPUs or Xeon Phis to meet their computing needs are now commonplace. Existing programming tools for these accelerators [1]–[3] typically define APIs to deploy and execute *compute kernels*, *i.e.* single-instruction multiple-data (SIMD) parallel tasks, on the processing units of accelerators. However, these programming tools are not tailored for multi-accelerator application development, while using multiple accelerators, potentially heterogeneous, is definitely attractive to increase both the computing power and the memory capability of applications.

Designing applications able to exploit multiple heterogeneous accelerators is challenging. Beside writing compute kernels optimized for the accelerators, the developer has to cope with many non-functional aspects. First, in order to achieve the best possible performance, the developer has to identify an efficient partitioning of the workload among the accelerators. This efficient partitioning is not only related to the theoretical performance of the accelerator, but also to the actual application and its workload. Then, the developer has to write code to coordinate the execution of multiple kernels running on different devices, to actually partition the workload among them and to perform data movements and synchronization between them. Implementing these features is time-consuming and error-prone. Moreover, the developer often implements these features for a specific hardware and has to drastically modify his code on a different hardware.

To ease the development of an application for multiple heterogeneous accelerators, we propose a new programming tool called STEPOCL. STEPOCL separates the functional aspects of the code, *i.e.*, the compute kernels, from the non-functional ones. The non-functional aspects of the code are described in a domain specific language, called the STEPOCL language. It mainly consists in describing the data layout and the kernel of the application. Using a domain specific language improves the productivity of the developer by decreasing the number of lines of codes required to implement a full multi-device application. Moreover, the code written in the domain specific language is not limited to a specific hardware setting, which increases the portability of the code. Finally, it also has the advantage of avoiding many bugs caused by the use of a low-level language.

Based on the compute kernels and on the description of the non-functional aspects, STEPOCL automatically generates a full OpenCL application, but also an offline profiler for this application. The profiler computes an efficient workload partitioning for a specific set of accelerators and a specific workload. The generated application takes in charge the initialization of the OpenCL environment, which includes the discovery of the accelerators, the mapping of the data to the accelerators according to the offline profiling results, and the launch of the kernels over the accelerators. During execution, the generated application also automatically exchanges the data between the accelerators to maintain the data consistency thanks to a polyhedral data analysis, and, at the end of the execution, the generated application reduces the results by retrieving them from the accelerators.

We evaluate STEPOCL with three application kernels (a 5-point 2D-stencil, a matrix multiplication and an N-body application) on two multi-device heterogeneous machines that combine CPUs with different accelerators: CPU+GPUs and CPU+Xeon Phis. Our main results show that:

- When running on multiple heterogeneous devices [1], the performance of the code generated by STEPOCL scales linearly with the number of devices.
- As compared to the same applications written directly in

[1]A device is either a CPU or an accelerator.

OpenCL and provided with the OpenCL version of AMD, the applications written with STEPOCL have similar performance.
- Thanks to STEPOCL, we are able to run large workloads on multiple devices that do not fit in the memory of a single device.
- As compared to the generated code, STEPOCL is able to divide by ten the number of lines of code of the application. Also as compared to the native OpenCL applications, STEPOCL is able to divide by five their number of lines of code. Furthermore, while the applications shipped with the OpenCL version of AMD only run on a single device, the STEPOCL applications are able to run on multiple heterogeneous devices.

The rest of the paper is organized as follows. Section II presents an overview of OpenCL, on which STEPOCL relies. Section III describes how STEPOCL generates source code and Section IV presents the offline profiler. The results of the evaluation of STEPOCL are analyzed in Section V. Section VI discusses the articulation of our work in the state of the art. Finally, Section VII gives a conclusion of the paper.

II. OpenCL in a nutshell

This Section briefly describes the Open Computing Language [2] (OpenCL), the language used internally by STEPOCL. OpenCL combines a unified programming interface with a variant of the C language to use different parallel processing devices together (e.g. CPU, GPU and Xeon Phi).

OpenCL defines an abstract device as a set of *compute units*, each compute unit being a set of *processing elements* (PEs) with a compact high-speed local memory.

OpenCL follows the single-instruction multiple-data (SIMD) execution paradigm. The developer first defines a function, called a *kernel*. Then, he defines a *host code* that orchestrates the copy of the data used by the kernel on the device before starting multiple instances of this kernel on the PEs. An instance of the kernel is called a *work-item*. It handles only a subset of the data identified by its *specific identifier*, which is given by a specific OpenCL primitive (the `get_global_id` primitive).

Because a device does not provide global synchronization mechanisms neither global shared memory, OpenCL also defines the notion of *work-group*. A work-group contains a set of work-items and is deployed on a same compute unit. Figure 1 illustrates this notion of *work-group*: a 2D-kernel index space is split into $Gx \times Gy$ *work-groups*, each of which consists of $Sx \times Sy$ work-items. For the work-items of a same work-group, OpenCL provides efficient synchronization mechanisms and allows to share local memory as they are executed on the same compute unit.

III. STEPOCL code generator

This Section describes both the STEPOCL language and the generated OpenCL application, while the next Section describes the offline profiler.

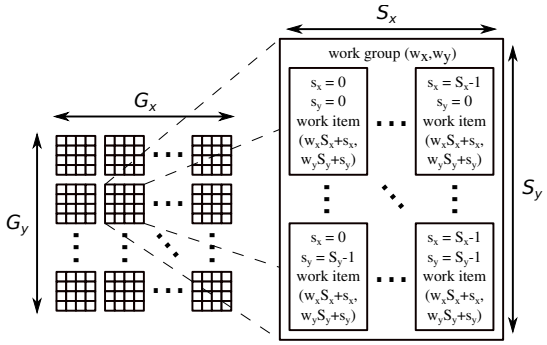


Fig. 1. OpenCL Work-Groups and Work-Items.

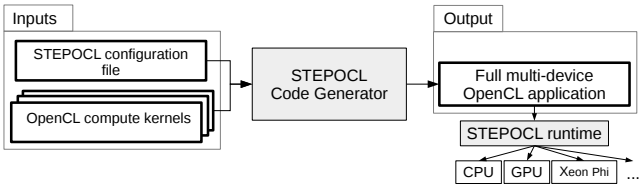


Fig. 2. Overview of the STEPOCL environment.

As illustrated in Figure 2, the core component of STEPOCL is a *code generator*, which takes as input a list of raw OpenCL kernels (see Subsection III-A) together with a configuration file (see Subsection III-B) which describes:

- the layout of the data, which expresses how the data shall be split among the devices,
- the association between a compute kernel and a specific device type (see Subsection III-B), and
- the expected control flow of the application to generate.

Based on this input, STEPOCL then generates a complete OpenCL program (see Subsection III-C) able to exploit concurrently different accelerators, e.g., CPU, GPU and Xeon Phi that runs on top of the STEPOCL *runtime*. At bootstrap, the generated code sets up a unified OpenCL multi-device environment and distributes the workload among the accelerators based on the results of the offline profiler. Then, during the run, the generated code maintains data consistency between the devices by using the result of a data flow analysis performed during compilation.

A. Compute kernels

The input kernels are regular OpenCL kernels, which express the computation to execute on a device. As a basic example for the remaining of this Section, we provide a 1D-stencil kernel written for a single generic device in Listing 1.

The developer may prepare several versions of the compute kernel to achieve the best performance on a specific device type. For instance, in order to favor data cache effects of a CPU in the 1D-stencil, it is more efficient to process data by tile, *i.e.* by block of elements instead of element-by-element, as presented in Listing 2. In this case, the computation performed

```
__kernel void generic_stencil1D(__global float *A, __global
      float *B) {
  const unsigned int i = get_global_id(0) + 1;
  B[i]= (A[i-1]+A[i]+A[i+1])/3;
}
```

Listing 1. Generic OpenCL 1D-stencil kernel.

```
__kernel void tiled_stencil1D(__global float *A, __global
     float *B) {
  const unsigned int i = get_global_id(0) + 1;
  for(int k = 0; k < 4; k ++)
    B[i+k]= (A[i+k-1]+A[i+k]+A[i+k+1])/3;
}
```

Listing 2. Tiled OpenCL 1D-stencil kernel.

by the CPU remains the same, but the amount of work-items differs.

B. STEPOCL *configuration file*

The configuration file takes the form of a tree, implemented in XML format. It defines the *arguments* of the kernel, how the *kernels* are mapped to the device type, and the *control flow* of the program.

An `argument` describes an array of values, transferred to the devices and accessed by the work-items. It is defined by three elements: (i) a name (`ID`), later used in the kernel section to refer to the argument, (ii) the type (`data_type`) of its tokens and (iii) its `size`, which further specifies the dimension and length of the array. For instance, the configuration file given in Listing 3 defines the arguments of our 1D-stencil example: arrays `A` and `B` are two 1D vectors, which contain 1026 float values each.

The `kernel` section of the configuration file describes the mapping between the kernels and the device types. It contains three elements:

- A `name` element. It references the kernel in the subsequent control flow section.
- A `data_split` element. For each argument, it gives along which dimension its related data should be split.
- At least one `implem` element. An implem element associates a compute kernel code to a specific device and defines the size of a *tile*, *i.e.* the size of the data to compute by each work-item. The default tile size is 1.

The kernel section may contain several *implem* elements in order to associate different kernel codes (`funcname`) to different devices (`device_type`). The DEFAULT `implem` is used when no other `implem` corresponds to the device.

As introduced in Section II, the programmer can also specify the size of the work-groups (`work-group`) used to design a specific kernel version.

Listing 4 presents the kernel configuration of the 1D-stencil. The user expresses that arrays `A` and `B` shall be split by column (*i.e.* split along the `x` axis). Three versions of the kernel are provided. The first version (the `tiled_stencil1D` kernel) targets CPU devices and performs its computation on tiles of four elements. The second version (the `GPU_stencil1D` kernel) is designed for GPU devices. It specifies the aggregation of work-items by groups of 16 along the x axis, which means that a group of 16 work-items can share their memory. The third kernel version (the `generic_stencil1D` kernel) is the more basic as it uses neither tile nor work-group. This version is also the more generic as it can be used on any other available device.

```
<argument>
  <ID> A </ID>
  <data_type> float </data_type>
  <arg_size>
    <dim_size axis=x> 1026 </dim_size>
  </arg_size>
</argument>
<argument>
  <ID> B </ID>
  <data_type> float </data_type>
  <arg_size>
    <dim_size axis=x> 1026 </dim_size>
  </arg_size>
</argument>
```

Listing 3. Argument description of the 1D-stencil kernel.

```
<kernel>
  <name> Stencil1D </name>
  <data_split>
    <axis ID=B> x </axis>
    <axis ID=A> x </axis>
  </data_split>
  <implem>
    <device_type> CPU </device_type>
    <funcname> tiled_stencil1D </funcname>
    <tile>
      <target_arg> B </target_arg>
      <size axis=x> 4 </size>
    </tile>
  </implem>
  <implem>
    <device_type> GPU </device_type>
    <funcname> GPU_stencil1D </funcname>
    <work_group>
      <size axis=x> 16 </size>
    </work_group>
  </implem>
  <implem>
    <device_type> DEFAULT </device_type>
    <funcname> generic_stencil1D </funcname>
  </implem>
</kernel>
```

Listing 4. 1D-stencil kernel information.

The third component of the STEPOCL configuration file describes the program `control` flow. The control flow is basically the meta-algorithm of the application and is used by the data analysis pass to ensure consistency. This component describes the number of iterations of the kernel to launch (`loop` keyword) and how data are exchanged between two iterations (`switch` keyword). For instance, in Listing 5, the STEPOCL application executes 10 times the `Stencil1D` and has to switch the arguments `A` and `B` between two iterations.

C. STEPOCL *Output OpenCL application*

Based on the kernel(s) and on the configuration file, STEPOCL generates a multi-device OpenCL program. After the initialization of the OpenCL environment, the gener-

```
<control>
  <loop iterations=10 >
    <exec> Stencil1D </exec>
    <switch>
      <arg ID=A> B </arg>
    </switch>
  </loop>
</control>
```

Listing 5. 1D-stencil kernel execution.

```
__kernel void generic_stencil1D(__global float *A, __global
    float *B) {
  const unsigned int i = get_global_id(0)+1;
  //  <A[PHI1]-R-EXACT-{PHI1==i, i==1}>
  //  <A[PHI1]-R-EXACT-{PHI1==i+1, i==1}>
  //  <A[PHI1]-R-EXACT-{PHI1==i-1, i==1}>
  //  <B[PHI1]-W-EXACT-{PHI1==i, i==1}>
  B[i]= i + (A[i-1]+A[i]+A[i+1])/3;
}
```

Listing 6. Resulting READ and WRITE array regions of the 1D-stencil kernel.

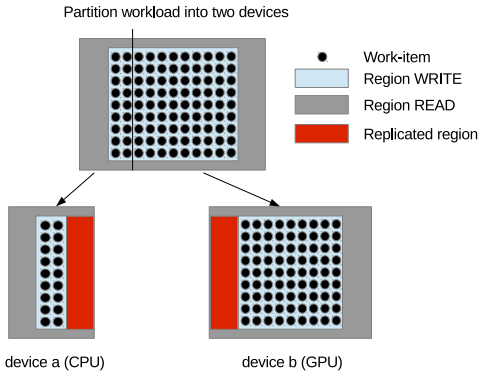


Fig. 3. Detection of data to be updated on other devices between two iterations.

ated program contains three main components: the *detection* component, the *deployment* component, and the *consistency* component. The generated code ends by collecting and aggregating the result from each device. Algorithm 1 illustrates this process.

Detection component: During the initialization (line 1 of Algorithm 1), the generated code detects the available OpenCL devices and associates a ratio to each device thanks to the offline profiler (see Section IV). It indicates the percentage of the workload that the generated code has to deploy to each device. After the initialization, the `Devices` variable thus contains a list of `(device, ratio)` pairs.

Deployment component: The deployment component (lines 2 to 5 and lines 8 to 12 in Algorithm 1) performs the initial deployment of the kernels on the devices.

First, based on the ratio (provided by the detection component) and on the tile size (provided by the configuration file, see Section III-B) of each device, the generated code computes the number of tasks for each device. For a given device, the number of tasks is simply equal to the size of the output argument multiplied by the ratio and divided by the size of a tile.

Then, with the `data_split` axis provided by the configuration file (see Subsection III-B) and on the list of tasks computed at the previous line, the generated code computes the subset of the input data accessed by each device (line 4 of Algorithm 1). This computation relies on the PIPS compiler [4]. PIPS analyzes the instructions and represents memory accesses as convex polyhedra [5]–[7]. As an illustration, Listing 6 reports the access patterns to arrays A and B in the 1D-stencil: the kernel reads A at three different positions (i, $i+1$ and $i-1$) while it writes B at the position i. From this information, the generated code computes the convex hull of the data accessed by the tasks of each device, and stores this result in the `Subdata` variable.

Finally, the generated program copies the data to the devices (lines 8 to 9 of Algorithm 1) and deploys the tasks (lines 11 to 12).

Consistency component: This component ensures data consistency and tries to minimize the data transfers between the devices between two iterations. It includes the lines from 6 to 7 and line 14 of Algorithm 1.

The generated code first identifies the regions that have to be exchanged between the iterations (line 6 of Algorithm 1). Again, the generated code uses the PIPS analysis. For each argument exchanged between two iterations, the generated code identifies which regions of the argument are replicated between at least two devices, accessed in read in the input argument and accessed in write in the output argument. Figure 3 presents an illustration with a 2D-stencil computation. In this case, the red regions are replicated between the devices. They are accessed in read during an iteration because computing the new values at the border of the blue part of the matrix relies on the values in the red part. In this example, as at the end of an iteration, the B matrix becomes the A matrix, the generated code has to reload the red regions from its neighbor devices.

For each device, the `FindDeviceNeighboring` function at line 6 thus identifies both these neighbors and their associated replicated regions. At the end of an iteration (line 15 of Algorithm 1), the generated code uses the result of this analysis to perform the communication.

IV. STEPOCL OFFLINE PROFILER

This Section presents the offline profiler used by STEPOCL to estimate the optimal workload partitioning of the generated application on a set of devices, so that further executions of the application balance the workload efficiently.

In order to generate the offline profiler, we reuse the compilation infrastructure described in the previous Section, but with three modifications. First, instead of using the ratio provided by the offline profiler (which is not yet available), the generated code estimates an initial ratio r_i^0 of data that has to be deployed on the device i by using its specification. Second, the offline profiler computes the elapsed time d_i^j to perform the iteration j on each device i. Third, after each iteration, the offline profiler copies all the output arguments that have

Algorithm 1: Generated host code.

Input:
$Data\_host$: Data location on the host
$Data\_size$: Data size
$Data\_split$: Data-splitting plan
$Kernel\_tile$: Computation size done by each kernel instance
N_iter : Number of iterations
Data:
$Devices$: List of detected computing devices
$Subtasks_i$: Workload assigned to device i
$Subdata_i$: Data chunk to distribute on device i
$Kernel\_accesses$: Data access pattern by the kernel
$Neighbors_i$: Data neighboring of device i
$Data\_device_i$: Data location on device i

```
Devices ← InitOpenCLEnvironment();
Subtasks ← PartitionWorkload(Data_size,
Kernel_tile, Devices);
Subdata ← PartitionData(Data_host, Data_size,
Data_split, Subtasks);
Neighbors ← FindDeviceNeighboring(Subdata,
Kernel_accesses);
foreach dev_i in Devices do
    Data_device_i ← InitDeviceData(dev_i, Subdata_i);
for k=1 to N_iter do
    foreach dev_i in Devices do
        InvokeKernel(Subtasks_i, Data_device_i);
    WaitForTermination();
    UpdateSubData(Subdata, Neighbors);
foreach dev_i in Devices do
    ReadBack(Data_host, dev_i, Data_device_i);
FinalizeOpenCLEnvironment();
```

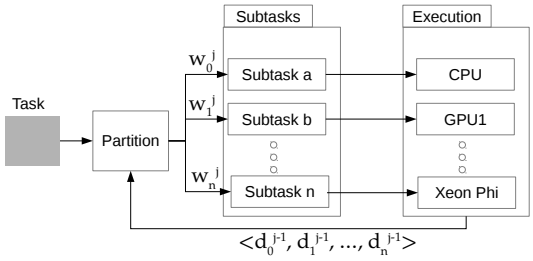


Fig. 4. Overview of the load balancing algorithm used in the STEPOCL runtime.

to be switched with an input argument in the main memory, re-estimates a new ratio r_i^{j+1} based on the elapsed time and the previous ratio r_i^j, and redeploys the input arguments on the device.

Figure 4 summarizes the mechanism. In the Figure, w_i^j represents the workload of the device i at the iteration j, *i.e.*, the size of the input data multiplied by r_i^j.

A naive way to adjust the ratios after each iteration would be to consider the ratio between $\overline{d^j}$, the mean duration at iteration j, and d_i^j, for instance, by defining r_i^{j+1} as $r_i^j \times 1 + \overline{d^j}/d_i^j$. The main drawback we may encounter by scheduling with this method is the occurrence of the ping-pong effect illustrated in Figure 5. Let us consider two homogeneous devices A and B. Let us assume, for instance, that due to the instability caused by cache misses or some other reasons, the evaluation of the duration of iteration j for CPU_a is over estimated. Then at iteration $j+1$, the naive re-balancing formulae will assign more work-items to CPU_b than to CPU_a. As a result, at the iteration j, the offline profiler considers that CPU_b works slowly and under loads it. We have observed that the naive re-balancing formulae can overload CPU_a at iteration $j+2$ and that this phenomenon may be amplified.

In order to reduce the risk of this ping-pong effect, STEPOCL adapts the changing speed of the ratio adjustment from one iteration to another. The main idea of this method is to reduce this changing speed whenever an inversion of the direction of variation of r_i^j is detected. To this end, a value Q, initialized to 1, is incremented after each such inversion.

We consider that when Q increases, *i.e.*, that after few inversions, the re-balancing factor $\overline{d^j}/d_i^j$ should become less important because the workloads are converging to an efficient configuration. For this reason, we define the ratio for iteration j and for device i as follows:

$$r_i^{j+1} := r_i^j \left(1 + \frac{1}{Q} \times \left(\frac{\overline{d^j}}{d_i^j} - 1\right)\right);$$

The self-adjustment process keeps on running until σ^j, defined as the standard deviation of execution time on each device for iteration j, becomes small enough. In STEPOCL, we consider that the workload is "calibrated" when $\sigma^j < 0.05\ \overline{d^j}$. Once the calibration is complete, the new ratios r_i^{j+1} are returned, so that the next invocations of the generated application distribute the workload efficiently among the tested devices.

V. Experiments

We evaluate STEPOCL on three test cases: a 5-point 2D-stencil, a matrix multiplication and an N-body application.

We start by describing the target application context before evaluating the generated codes according to different criteria:

- we compare the size of handwritten applications to the size of the equivalent applications generated by STEPOCL;
- we evaluate the accuracy of the offline profiler;
- we compare the performance of the handwritten applications to the performance of the equivalent applications generated by STEPOCL when running on a single device;
- we evaluate the performance of the generated applications when running on multiple devices.

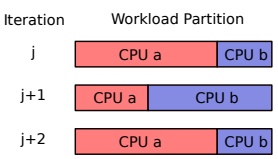


Fig. 5. Ping-pong effect of workload adjustment.

```
__kernel void stencil(__global float *B,
                      __global float *A,
                      unsigned int line_size) {
 const unsigned int x = get_global_id(0);
 const unsigned int y = get_global_id(1);
 A += line_size + 1; // OFFSET
 B += line_size + 1; // OFFSET
 for(unsigned int k=0; k<4; k++){
   B[(y*4 + k)*line_size + x]
   = 0.75 * A[y*4 + k][x]
     + 0.25 * (tile[y*4 + k+1][x-1]
             + tile[y*4 + k+1][x+1]
             + tile[y*4 + k-1][x-1]
       + tile[y*4 + k-1][x+1] );
 }
}
```

Listing 7. 2D-stencil kernel with a tile of 4 elements.

A. Test cases

In order to assess the STEPOCL usability and performance, we use three typical applications that can run on heterogeneous multi-device platforms.

Stencil computation: The first application is a home-made 5-point 2D-stencil. It takes a 2D matrix of scalar values as input and outputs, for each element, a weighted average of its 4-neighborhood. Figure 6 illustrates this computation. We compare the STEPOCL stencil with another native OpenCL version implemented by one of the author before our work on STEPOCL. We set the configuration file so that:

- It defines two kernel arguments: an *input* and an *output* matrices of float elements, which are marked to be split by lines.
- It defines two kernel versions: the first kernel (reported in Listing 7) is designed to fit CPU and Xeon Phi devices by setting work-items to work over 4-elements wide tiles while the second kernel is optimized for GPUs and takes advantage of their shared memory. Each work-group instantiates 16×4 work-items that first copy their data to their local shared memory before entering the computation phase and processing 4 elements.
- The target application iterates ten times, switching input and output matrices after each iteration. The STEPOCL code updates data in device memory as described in Subsection III-C: when a sub-matrix, corresponding to the bounded interval $[(x1, y1), (x2, y2)]$, is assigned to a device, the frontiers of its corresponding output are fetched from all its neighbor devices.

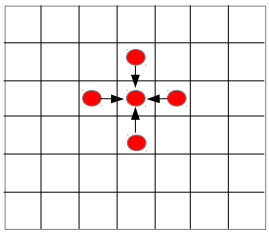


Fig. 6. 5-point 2D-stencil computation.

Matrix multiplication: The matrix multiplication application computes the $C = A \times B$ operation on 2D matrices. We compare the STEPOCL matrix multiplication with the one provided by the AMD APP SDK benchmark. We set the related configuration file so that:

- It defines three kernel arguments: the three matrices of float elements A, B and C. C and A are marked to be split by lines, and B is totally replicated on all the devices because of the data dependency analysis.
- It defines two kernel versions: the first one targets CPU and Xeon Phi devices using a tile of 4×4 elements, while the one designed for GPUs also uses a tile of 4×4, but defines work-groups of 4×4 work-items in order to work on shared local memory.
- The generated application executes only one iteration without triggering any communication between devices.

N-body: The N-body application simulates the collective motions of a large particle set under Newtonian forces in a 3D space. At each iteration, each particle, defined by a mass and a velocity, changes both its position and its velocity by using the position and the velocity of all the other particles. We compare the STEPOCL N-body with the one provided by the AMD APP SDK benchmark.

In STEPOCL, we define the following arguments:

- Three integers: the number of particles, a softening factor and an elapsed time interval. All these arguments are replicated because of the data dependency analysis;
- Two arrays of float elements containing the particle positions: one stores the positions at the previous iteration while the other, the current ones;
- Two arrays of float elements that contain the velocity of each particle: one array stores the velocities at the previous iteration while the other contains the current ones.

The application simulates ten iterations. After each iteration, it switches the input and output arrays. Considering the communication, the two input arrays are replicated on all the devices because of the data dependency analysis. Indeed, to compute the new position and a new velocity of one particle, all particle positions and velocities of the previous iteration are used. As the input and output arrays are switched after an iteration, between each iteration, each device broadcasts its output to all the other devices.

B. Volume of the generated source code

Table I reports the number of lines of code of the three native OpenCL applications as well as the size of the three configuration files that STEPOCL used for generating the tested applications. The table also contains the number of lines of code of the applications generated by STEPOCL. The kernels consist of a few tens of lines of code. However, the native OpenCL applications and generated host codes in charge of instantiating the kernels on the devices consist of several hundreds of lines of code.

The native OpenCL applications are only able to run on a single device, while the STEPOCL applications can run

Test case	Native OpenCL application (kernels included)	Kernels	STEPOCL configuration file	STEPOCL generated code (kernels included)
Stencil	490	GPU=51/others=18	75	1153
Mat. Mult.	1212	GPU=103/others=78	56	1216
N-body	1041	81	83	1116

TABLE I
GENERATED STEPOCL CODE SIZE (IN LINES).

Name	HANNIBAL	MISTRAL
CPU models	2 x Intel Xeon X5550	2 x Intel Xeon E5-2670
# CPU cores	2 x 4	2 x 10
# threads	2 x 8	2 x 10
CPU frequency	2.66 GHz	2.50GHz
OpenCL support	Intel OpenCL 1.1	Intel OpenCL 1.2
Accelerator type	NVIDIA GPU	Intel Xeon Phi
Models	3 x Quadro FX 5800	2 x Xeon Phi 7120P
# cores	3 x 240 CUDA cores	2 x 61 cores (2 x 244 threads)
Processor clock	1296 MHz	1238MHz
Memory	3 x 4096 MB GDDR3	2 x 16 GB GDDR5
OpenCL support	NVidia OpenCL 1.1	Intel OpenCL 1.2

TABLE II
EXPERIMENTAL PLATFORM OUTLINE.

on multiple devices. This difference explains that the number of lines of code of the stencil generated with STEPOCL is significantly larger than the native OpenCL stencil. For the two other applications, this comparison is unfair: the native OpenCL applications are written in C++ with a large number of templates while the generated STEPOCL applications are written in C.

To conclude, we can observe that the STEPOCL configuration files contain only a few tens of lines of code, which is roughly ten times smaller than the generated code. This result shows that STEPOCL simplifies the development of an application for multiple device.

C. Performance evaluation

We now evaluate the performance of the three applications on two heterogeneous platforms.

1) Experimental platforms: We summarize the characteristics of the two platforms in Table II: the HANNIBAL platform is a dual quad-core Intel Xeon CPU with three Quadro FX 5800 GPUs while MISTRAL is a dual 10-cores Intel Xeon CPU with two Xeon Phi accelerators.

2) Evaluation of the profiler: Before evaluating the performance of the applications, we evaluate the offline profiler by measuring how it converges towards a balanced distribution of the workload.

Figure 7 depicts the workload distribution w_i^j and the measured duration d_i^j of the CPU and GPU devices on a machine running a 7-point 3D-stencil. Artificially, for the first iteration of the application, we assign 99 % of the workload to the GPU and the remaining 1 % to the CPU. After the first iteration, the profiler detects that it assigned too much workload to the GPU. The workload partition for the second iteration thus assigns more work-items to the CPU, which results in a more balanced execution time. Yet, the difference between d_0^{t2} and d_1^{t2} leads to assigning more work-items to the CPU for the third iteration. After the third iteration, the profiler detects that the difference between d_0^{t3} and d_1^{t3} is small enough and stops the calibration process. In this experiment, we run a few more iterations in order to make sure that the execution on each device is stable when the workload distribution does not change.

3) Comparison with the reference codes: In order to ensure that the code generated by STEPOCL does not degrade the performance as compared to a native implementation written directly in OpenCL, we compare the performance of the generated application with the native OpenCL applications.

As the native OpenCL applications are only designed to run on a single device, we only use a single GPU on HANNIBAL. From Table III, we can observe that STEPOCL does not change the performance of the matrix multiplication and that STEPOCL introduces an overhead of 1% for stencil and 2% for N-body. From this result, we can thus conclude that STEPOCL does not significantly modify the performance on a single accelerator.

4) Stencil: Figure 8 presents the performance of the generated stencil code on the two machines. Our measures underline the performance scalability achieved by STEPOCL, with GPU and CPU devices adding up their computational horsepower efficiency.

On HANNIBAL (see Figure 8(a)), we observe that the peak performance achieved when using all the available devices (91.8 GFLOPS) roughly corresponds to the sum of the per-

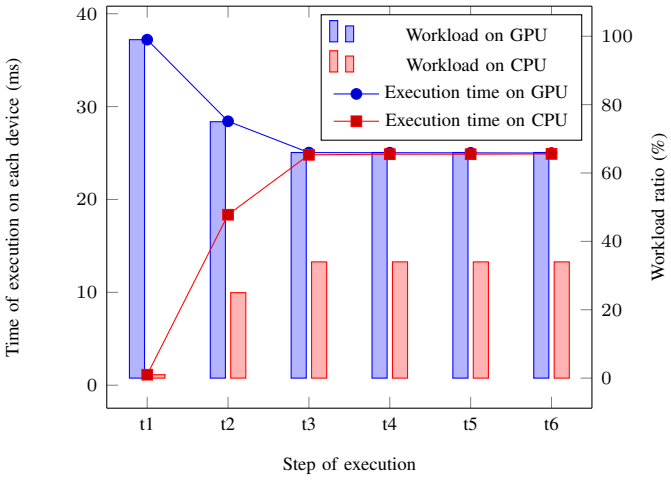
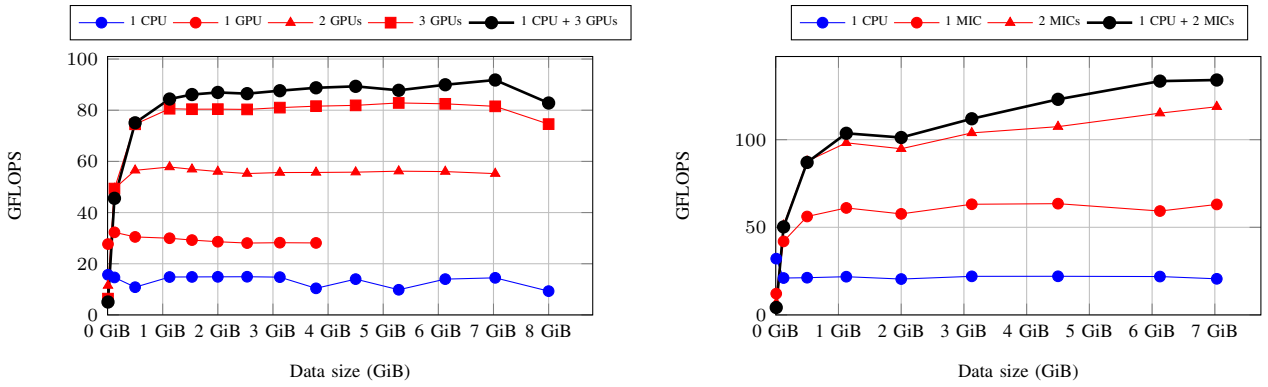


Fig. 7. Workload adjustment performance of the 3D-stencil application.



(a) Performance on HANNIBAL

(b) Performance on MISTRAL

Fig. 8. Performance of the 5-point 2D-stencil application. The horizontal axis corresponds to the size the input and output matrices required to solve the problem.

formance of each device running individually (14.5 GFLOPS for 1 CPU, 28.1 GFLOPS for 1 GPU). The efficiency is not 100 % because of the communication between the devices that are required when computing the stencil on multiple devices in order to ensure data consistency.

The results on the mistral platform have a similar trend: the peak performance when using all the devices (134.1 GFLOPS) approximately corresponds to the sum of the individual performance (20.6 GFLOPS on 1 CPU, 63.1 GFLOPS on 1 Xeon Phi).

From these two results, we can conclude that STEPOCL scales linearly with the number of devices. This result also confirms that the offline profiler seems to provide efficient ratios able to perfectly balance the load between the devices.

Moreover, STEPOCL pushes forward the memory limits by automatically distributing data sets which are too large to be processed by a single device, summing up the memory of multiple devices to handle larger problems. On the HANNIBAL platform, where GPUs are equipped with 4 GiB of memory, the 1-GPU version cannot process the test cases that require more than 4 GiB of memory. Similarly, the 2-GPUs version is limited to 8 GiB, while the versions that exploit the 3-GPUs can process larger problems.

5) Matrix multiplication: Figure 9 presents the performance of the generated matrix multiplication on the HANNIBAL and MISTRAL machines. On HANNIBAL (see Figure 9(a)), the performance achieved when using both the CPU and the 3 GPUs (184.3 GFLOPS) almost corresponds to the sum of the performance achieved by each device individually (18.36 GFLOPS on the CPU, 56.16 GFLOPS on each GPU).

Application name	2D-Stencil	Mat. Mult.	N-body
Workload	4096x4096	1024x1024	32768 particles
Relative performance	0.99	1.00	0.98

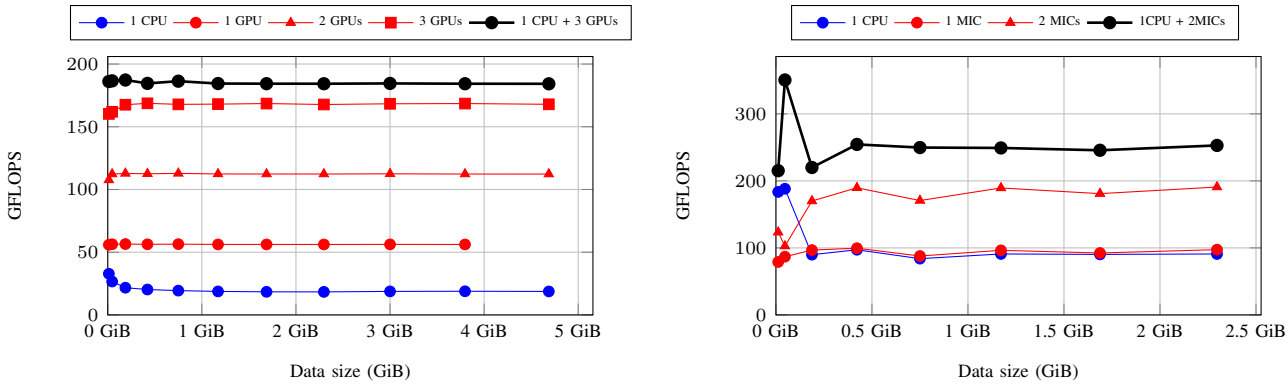
TABLE III
RELATIVE PERFORMANCE OF STEPOCL AS COMPARED TO A NATIVE OPENCL IMPLEMENTATION ON HANNIBAL.

On MISTRAL (see Figure 9(b)), when using the CPU and the 2 Xeon Phis (252.9 GFLOPS), the performance corresponds to 88 % of the one achieved by each device individually (90.9 GFLOPS on the CPU, 97.4 GFLOPS on each Xeon Phi). Thus, in terms of performance, the STEPOCL generated code scales up correctly. Moreover, as for the stencil application, the 1-GPU version is bounded by its inner memory size. Once again, STEPOCL allows to compute bigger problem than the original code thanks to its multi-device dimension.

6) N-body: Figure 10 presents the performance of the generated N-body application on HANNIBAL and MISTRAL. On HANNIBAL (see Figure 10(a)), the performance achieved when using both the CPU and the 3 GPUs (152.8 GFLOPS) corresponds to 94 % of the cumulated performance achieved by each device individually (9.38 GFLOPS on the CPU, 51.12 GFLOPS on each GPU). On MISTRAL (see Figure 10(b)), when using the CPU and the 2 Xeon Phis (90 GFLOPS), the performance corresponds to 83 % of the one achieved by each device individually (16.31 GFLOPS on the CPU, 45.68 GFLOPS on each Xeon Phi).

As described in Subsection V-A, the updated data communication between each iteration is highly critical in comparison to our two other test cases: after each iteration, the computed data is broadcasted to all devices attending the computation. This is why the generated N-body code does not scale well on small data sets, particularly on Xeon Phi as cross-device communication are very expensive on this platform. However, the computation time increases in quadratic with the workload, while the communication time only increases linearly. For this reason, with larger workloads, the N-body application scales linearly with the number of devices.

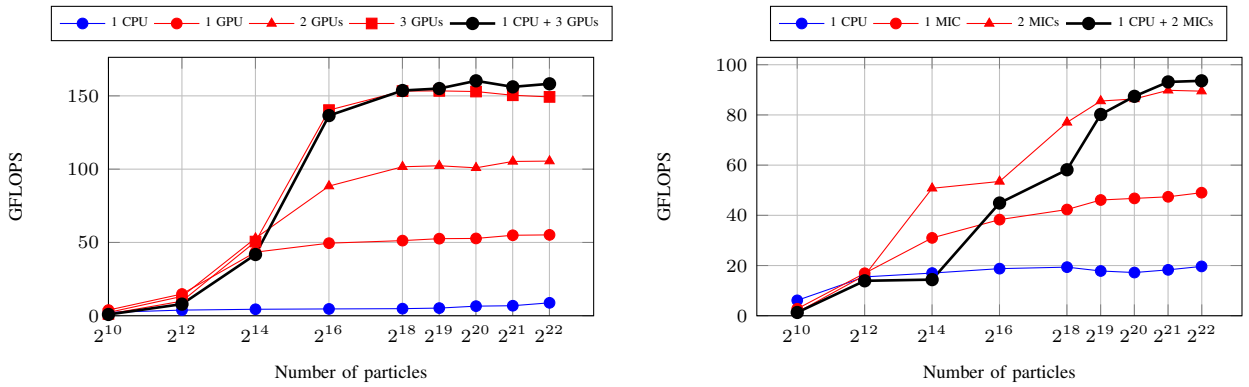
The N-body application does not deal with memory footprint as all arrays need to be duplicated. Since computation grows quadratically, it really focuses on the reduction of the makespan. STEPOCL reaches this objective with its multi-device dimension.



(a) Performance on HANNIBAL

(b) Performance on MISTRAL

Fig. 9. Performance of the matrix multiplication application. The horizontal axis corresponds to the summed size of the A, B, and C matrices.



(a) Performance on HANNIBAL

(b) Performance on MISTRAL

Fig. 10. Performance of the N-body application.

VI. STATE OF THE ART

As heterogeneous architectures are becoming ubiquitous, many studies have focused on alleviating heterogeneous systems programming. OpenCL [2], which is described in Section II, is an interface used to implement parallel programs over heterogeneous systems. While OpenCL only provides primitives to deploy a compute-kernel on a specific device, STEPOCL also automatically takes in charge the deployment on heterogeneous devices, the communication between the devices, the data consistency between the device and the retrieval of the result from the different devices at the end of the computation.

In order to maintain a software compatibility when the architecture changes and in order to introduce dynamic load balancing, several research works propose to enable applications to dynamically dispatch computation kernels over processing devices so as to maximize their utilization. SOCL [8], based on the StarPU [9] runtime system, allows the developer to submit the compute kernels to a unique virtual OpenCL device, which transparently distributes them over the physical devices at runtime. LibWater [10] also proposes a uniform approach for programming heterogeneous computing systems thanks to a *global command queue*. Furthermore, LibWater hides the distribution by managing transparently devices on remote machines. With these approaches, application developers submit tasks and describe dependencies between tasks. The runtime system then schedules each task over a single device. Instead, STEPOCL targets the uses of multiple device to execute a task. It partitions the workload among the devices and maintains their consistency during the execution, which is not handled by these projects.

Other approaches consist in using source-to-source compiler techniques for generating the source code necessary for exploiting multiple devices. For instance, Kim et al. [11] propose an OpenCL framework to manage multiple GPUs within a node. This OpenCL framework hides multiple GPUs and simulates a single GPU. It takes in charge the deployment of the code on the multiple GPUs transparently by partitioning the workload equally among the different devices. Compared to STEPOCL, Kim et al. focuses on homogeneous multiple device systems and data independent tasks. Their OpenCL framework can neither work with heterogeneous devices and the developer has still to write the code to handle data consistency and communication.

Klaus Kofler et al. [12] propose an approach to automatically optimize task partitioning for different problem sizes and different heterogeneous architectures. They use the Insieme [13] source-to-source compiler to translate a single device OpenCL program into a multi-device OpenCL program. The Insieme runtime system then performs dynamic task partitioning based on an offline-generated prediction model. The predictive model is generated by using an Artificial Neural Networks approach. It incorporates static program features with dynamic input sensitive features. While the Insieme runtime partitions the workload at runtime, it relies on predicted performance that may be imprecise. The approach that we propose partitions the workload according to the actual measured performance of the application.

Lee Janghaeng et al. also propose a tool, called SKMD, that generates a multi-device source code from a single-device kernel [14]. This system transparently orchestrates collaborative execution of a single data-parallel kernel across multiple asymmetric CPUs and GPUs. The programmer is responsible for developing a single data-parallel kernel in OpenCL, while the system automatically partitions the workload across an arbitrary set of devices, generates kernels to execute the partial workloads, dynamically adjusts the workload across devices and merges the partial outputs together. As compared to STEPOCL, SKMD duplicates all data on every device. Thus each device needs to allocate all the data, even if the device does not use it. This unnecessary memory redundancy for the whole system limits the problem size that can be treated. On the contrary, STEPOCL allocates only the necessary data on each device and automatically transfers intermediary data between devices. Thus, larger problems can be processed using multiple devices.

VII. Conclusion

This paper introduces STEPOCL, a programming tool that eases the development of applications for multiple heterogeneous devices. Based on a legacy OpenCL compute kernel and a STEPOCL configuration file provided by the user, STEPOCL generates an offline profiler to guide the partitioning of the workload and generates the OpenCL host part of the application. The generated application schedules the workload according to the profiling results, launches their execution, and performs the necessary data exchanges between devices.

We evaluated STEPOCL with three applications: a stencil application, a matrix multiplication, and an N-body application. We measured the performance of these applications on two different multi-device platforms. Our evaluation shows that, thanks to STEPOCL, the number of lines of code to write an application for multiple devices is drastically reduced. Our measurements also show that the code generated by STEPOCL can run on complex multi-device systems and that its performance scales well with the number of devices. Using multiple devices also enables to cope with problem sizes that cannot fit into a single accelerator.

Further works will focus on the research of automatic kernel optimization for heterogeneous architectures. Architectures of current accelerators are very diverse. To obtain optimal performance, developers usually write different OpenCL kernel for each type of device. We thus plan to extend STEPOCL with heuristics to rewrite the code, in order to generate accelerator-specific optimization.

References

[1] Nvidia, "Compute unified device architecture programming guide," 2007.

[2] A. Munshi, B. Gaster, T. Mattson, and D. Ginsburg, *OpenCL Programming Guide*. Pearson Education, 2011.

[3] T. Ni, "Direct compute: Bring gpu computing to the mainstream," in *GPU Technology Conference*, 2009.

[4] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization: an overview of the pips project," in *ICS*, 1991.

[5] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier, "Step: A distributed openmp for coarse-grain parallelism tool," in *OpenMP in a New Era of Parallelism*. Springer, 2008, vol. 5004, ch. 8, pp. 83–99.

[6] B. Creusillet, "Array region analyses and applications," Ph.D. dissertation, École Nationale Supérieure des Mines de Paris, 1996.

[7] B. Creusillet and F. Irigoin, "Interprocedural array region analyses," in *Languages and Compilers for Parallel Computing*. Springer, 1996, vol. 1033, pp. 46–60.

[8] S. Henry, A. Denis, D. Barthou, M. C. Counilh, and R. Namyst, "Toward OpenCL Automatic Multi-Device Support," in *Euro-Par 2014 Parallel Processing*, 2014, pp. 776–787.

[9] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[10] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer, "LibWater: Heterogeneous Distributed Computing Made Easy," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ACM, 2013, pp. 161–172.

[11] K. Jungwon, K. Honggyu, L. J. Hwan, and L. Jaejin, "Achieving a single compute device image in OpenCL for multiple GPUs," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. ACM, 2011, pp. 277–288.

[12] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer, "An automatic input-sensitive approach for heterogeneous task partitioning," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ACM, 2013, pp. 149–160.

[13] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer, "INSPIRE: The insieme parallel intermediate representation," *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, vol. 0, pp. 7–17, 2013.

[14] L. Janghaeng, S. Mehrzad, P. Yongjun, and M. Scott, "Transparent CPU-GPU Collaboration for Data-parallel Kernels on Heterogeneous Systems," in *Proceedings of PACT '13*, 2013, pp. 245–256.