High Performance Code Generation for Stencil Computation on Heterogeneous Multi-device Architectures

Pei Li
Telecom SudParis,
9 rue Charles Fourier,
91011, Evry, France
pei.li@telecom-sudparis.eu

Elisabeth Brunet
Telecom SudParis,
9 rue Charles Fourier,
91011, Evry, France
elisabeth.brunet@telecom-sudparis.eu

Raymond Namyst
University of Bordeaux 1,
351 Cours de la Liberation,
33400 Talence, France
raymond.namyst@inria.fr

***Abstract*—Heterogeneous architectures have been widely used in the domain of high performance computing. On one hand, it allows a designer to use multiple types of computing units and each able to execute the tasks that it is best suited for to increase performance; on the other hand, it brings many challenges in programming for novice users, especially for heterogeneous systems with multi-devices.**

In this paper, we propose the code generator STEPOCL that generates OpenCL host program for heterogeneous multi-device architecture. In order to simplify the analyzing process, we ask user to provide the description of input and kernel parameters in an *XML* file, then our generator analyzes the description and generates automatically the host program. Due to the data partition and data exchange strategies, the generated host program can be executed on multi-devices without changing any kernel code. The experiment of iterative stencil loop code (ISL) shows that our tool is efficient. It guarantees the minimum data exchanges and achieves high performance on heterogeneous multi-device architecture.

***Index Terms*—GPGPUs, OpenCL, Stencil computations, Multi-device, Code generation, Heterogeneous architectures**

I. INTRODUCTION

High performance computing (HPC) is closely tied to scientific computing and industries. Because of the increasing complexity and growing amount of data for practical problem, we always demand better performance to achieve faster time to solution. Hence, there are two paths to combine: the enhancement of the algorithms and better material architectures. Recently HPC system architectures are shifting from the traditional homogeneous multi-core systems to heterogeneous systems such as GPGPU. Compared to the standard multi-core CPUs, GPGPUs offer a significantly higher floating point peak and a better power efficiency.

However, this novel architecture presents new challenges at the application developing level: time and effort are needed to exploit such kind of material. Moreover, many heterogeneous systems start having multiple computing devices. It can be more difficult and error-prone since developing programs that make best use of the characteristics of different computing devices increases the programmer's burden. Balancing the workload between several available computing devices can be also complicated, especially given that they have different performance characteristics. Besides, the communication and the exchange of intermediary results between several devices should also be considered. It is costly and difficult to design applications for heterogeneous multi-device systems. Thus, there is a huge demand for programming tools that help the novices designing applications for heterogeneous multi-device systems.

In this paper, we propose the code generator STEPOCL that automatically generates the parallel host OpenCL code for heterogeneous multi-device systems. It enables OpenCL programs written for a single compute device to run on systems with multiple devices without any modification. The architecture of the system is completely transparent to user. The information of available devices is obtained at run time and the workload is distributed to each device with optimal strategy. Thus, OpenCL kernels are executed in parallel. The host program manages automatically the communication and data exchanges between devices and results are retrieved from each device at the end of execution.

The rest of paper is organized as follows. In Section II, we present our contribution in detail. Section III discusses the evaluation of our generated code. Section IV presents related works. Finally, Section V concludes the paper.

II. STEPOCL

STEPOCL aims to facilitate programming on heterogeneous multi-GPUs systems through the open standard OpenCL [1], a language especially designed to address heterogeneous platforms consisting of multi-core CPUs, GPUs and other modern processors. Instead of writing the error-prone code for heterogeneous multi-device system, user only needs to provide the basic description of kernel argument, space information and the kernel function for one device. From these description, STEPOCL automatically generates an entire OpenCL source code for multiple devices architectures dealing with all the necessary technical aspects including not only the basic ones with the tuning of the initialization phase (library, devices declaration, etc.), or kernels launching and

retrieving of the results, but also trickier aspects in order to determine the best data and computation distribution or the exchange of intermediary results.

```
for(int t=0; t<T; ++t){
  for(int i=10;i<N-10;++i){
    for(int j=10;j<N-10;++j){
      A[i][j]=CNST * (B[i][j+1]+B[i][j-1]
          +B[i-10][j]+B[i+10][j]);
    }
  }
  swap(A,B);
}
```

Listing 1. Stencil Loop Example

Indeed, if we consider an iterative stencil computation – a case widely used in many scientific domains as depicted in Listing 1, the computation of each element needs the access to a set of neighboring elements according to a fixed pattern. Thus, in a multi-device version, the distribution may imply the allocation of some of those neighboring elements in the memory of another device. Then, some data need to be shared in some way – by exchange or by replication – and furthermore, need to be updated after each iteration in order to maintain the data coherency. All these operations make programming iterative stencil code on heterogeneous architectures more difficult and error-prone.

Thus, in order to generate a complete multi-device code, we need to face the following challenges:

- Workload and data partitioning: the kernel space and data space should be characterized in order to be efficiently partitioned by rows, columns or grids. The shared data regions [2] – called here ghost zones – should be distinguished from useful data.
- Managing the data transmission between devices: in order to maximize the reutilization of data transferred in devices memory, intermediary results need to be exchanged in a way that guarantees the minimum data transfer, that is to say only the effective data necessary to pursue the computation on the other devices.
- Code generation: it should be transparent to users. Users should not care about the number or the type of available devices. The OpenCL host code will be automatically generated from the description of kernel information for single device.

The rest of this section presents more details about each of mentioned points and how we consider to implement them in STEPOCL.

A. Workload and Data partitioning

1) Background - OpenCL kernel space: An OpenCL application consists of two distinct parts: the host program and a collection of computation kernels addressing computation potentially heterogeneous devices such as CPUs or GPUs. Kernels are typically simple functions that transform input memory objects into output memory objects. Nevertheless, the host part is the driver of the execution: it is in charge of the data transfers to and from the devices memory and of kernels launching on the devices. When the host submits a kernel, the OpenCL runtime system creates an integer index space based on two arguments: the number of the work-item and the size of work-groups. An instance of the kernel is executed for each point in this index space.

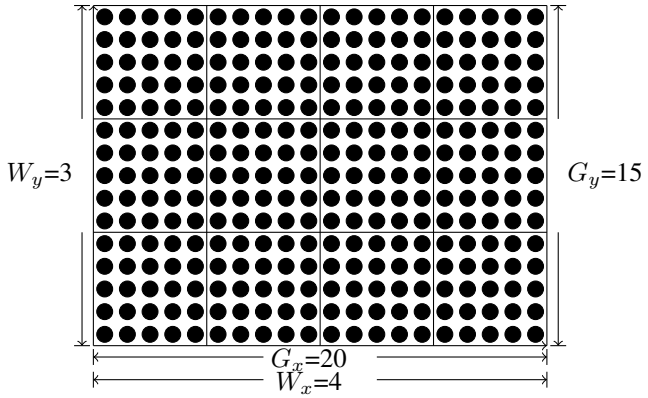


Fig. 1. The kernel index space

Each instance of an executing kernel is called a **work-item**. Work-item are organized into **work-groups**. The work-groups provide a more coarse-grained decomposition of the index space. All the work-groups are the same size in corresponding dimensions, and this size evenly divides the global size in each dimension. A unique ID is assigned to each work-group following the same dimensionality as the index space used for the work-items. Each work-item is referenced by a unique local ID within a work-group, so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. For example, Figure 1 shows a two-dimensional index space with a global size of index space of (20×15) and a size of work-group of (5×5). Hence the number of work-groups in the whole index space is (4×3)

During the execution, each work-item uses the same sequence of instructions defined by a single kernel. In order to execute the kernel on multiple devices, the number of work-item should be adjusted according to the number of device. The corresponding data space should also be split and distributed to the memory of each device.

2) Determining the kernel and data space: As we said in introduction, for the first prototype of STEPOCL, we solicit the assistance of the user to describe how the space can be split. Therefore, the kernel and data space is described in *XML* as a property of arguments as depicted in Listing 2. Still on the same example, the argument A is defined as a 2 dimension memory object with a size of 4096×1026. The number of work-item in kernel space is 4096×256. After analyzing these description with a XML parser, we can easily determine the size of kernel and data space. Nevertheless, the objective of our further work is to automatically analyze the kernel and data space from the real kernel program thanks to efficient compilers as PIPS [3] or Insieme [4].

```
<argument>
```

```
    <name>A</name>
    <property>output</property>
    <dataType>float</dataType>
    <cl_mem>1</cl_mem>
    <arg_size>
        <dim_size>4098</dim_size>
        <dim_size>1026</dim_size>
    </arg_size>
  </argument>

  <kernel>
    <name>stencil.cl</name>
    <global>
        <g_size>4096</g_size>
        <g_size>256</g_size>
    </global>
    <local>
        <l_size>64</l_size>
        <l_size>16</l_size>
    </local>
  </kernel>
```

Listing 2. Kernel arguments information

3) Distinguishing the useful data from ghost zones [5] and keeping them up-to-date : If we manage an ISL distribution, the ghost zones need to be exchanged among different processing elements at the beginning of the execution and after each iteration involving significant overhead in terms of communication and synchronization. Hence, larger ghost zones may be created to replicate stencil operations, reducing communication and synchronization costs at the expense of redundantly computing some values on multiple processing elements [2]. The optimal size of ghost zone can improve the performance for ISL on GPUs. Nevertheless, the objective of our work is to maximize the memory utilization to allow the kernels scaling. In this way, in our model, we always consider the minimal ghost zone size –even if it may be the case for the data space, computation space will not be replicated–. Just as in the previous paragraph, in the current STEPOCL prototype, we ask the programmer to give the relative information but the goal is to take advantage of the data dependency analysis from compilers to do it automatically.

We extract the domain of ghost zone and useful data region from the data description in the XML file given by the programmer. Then we split the useful data region into sub-region, and allocate relative ghost sub-regions for each data sub-regions. Still on the same example, ghost zone is required and may be described in the file as depicted below in Listing 3.

```
    <shadow>
        <point>{-10,0}</point>
        <point>{10,0}</point>
        <point>{0,1}</point>
        <point>{0,-1}</point>
    </shadow>
```

Listing 3. Ghost zone information

It indicates that the offset of data region from ghost zone in X dimension is (-10,10) and in Y dimension is (-1,1). The data region and ghost zone are presented in Figure 2.

Meanwhile, by analyzing the shape of ghost zone, we propose a simple partitioning strategy that guarantees a minimum

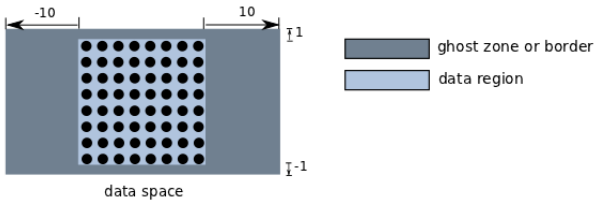


Fig. 2. Data space

amount of data transfer between the host memory and the other device memories.

For example, the Listing 1 above presents a part of typical stencil loop code. In this case, updating the matrix A depends more on the data which are relatively allocated on X axes. If we partitioning the kernel space by row (Fig. 3) instead of partitioning by column (Fig. 4), we can reduce significantly the communication and data transfer.

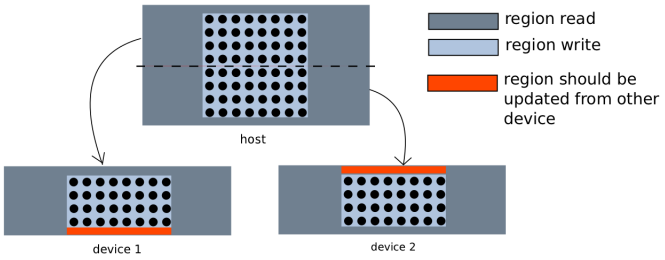


Fig. 3. Partition the data by row

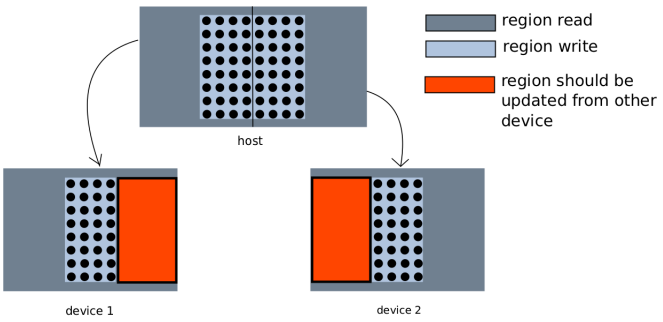


Fig. 4. Partition the data by column

4) Partitioning with abstract number of available devices: The number of available devices is unknown until the runtime in order to be able to exploit correctly the actual devices involved in the execution. Thus at compilation time, the number can only be presented with variable num_device. At first, we analyze the kernel space to evaluate the maximum parallel capacity. In Figure 1, the number of work-group in this kernel space is 12. If we want to parallelize stencil code without modifying kernel code, we need at most 12 devices. And the number of regions in dimension X should not exceed 4 (num_group_x), equally the number of regions in dimension

Y should not exceed 3 (num_group_y). Afterwards with the dependent points which are provided by user, we can determine a priority list. The list presents the splitting order of dimensions. For instance, the dependent points in Listing 1 are (0,1), (0,-1), (10,0) and (-10,0). The projection of ghost region size on X is $dx = (|10| + | - 10|) = 20$ and the projection on Y is $dy = (1 + | - 1|) = 2$. By comparing the size of projection, we can decide which axis should be split first. In this example, we should split the data in row first, which means splitting dimension Y. And, if there are still available devices, we can split the data on dimension X afterwards. So, the list of priorities is $list = \{y, x\}$.

Following the list of priority, we calculate the greatest common divisor (gcd) of $num\_workgroup_i$ and the number of available devices num_device for each dimension. num_device is updated with $num\_device/ = gcd(num\_workgroup_i, num\_device)$ after each gcd operation. The list of gcd presents the data distribution, and the result of $\prod_{i=0}^{N} gcd(i)$ is the number of devices that will be used for computing. In our application, we may decrease the number of total devices in a certain range (default value is 100) to achieve the maximum parallelization. In the example of Listing 1, with a priority list $list = \{y, x\}$, if OpenCL detected 2 available devices, the list of gcd should be {1,2}. This means that the data in dimension X remain in one part, but the data in dimension Y will be split into two parts, as shown in Figure 3. After the data partition, each segment (we call it **local zone**) should keep the following information: the global zone ID, the relative ID in each dimension and the range of indexes in each dimension. Each local zone is composed of written data region (we call it the **write-zone**) and ghost zone (corresponding to shared data). Since we know the size of local region and all the dependence points, the index domain of write-zone can also be determined. Thereby, local regions represent $EXACTREAD$ data regions and write-regions, $EXACTWRITE$ ones. As ghost zones are partial write-zone projection of neigbors, after each iteration of stencil loops, they need to be updated to prepare the next iteration. Thus, we need to determine the neighbors global ID list for each local zone.

The global ID of zone region can be represented with relative ID (re_ID_x, re_ID_y) and device partitioning information (the number of devices in the first dimension *numdev_x*, the number of devices in the second dimension *numdev_y*).

$$\begin{aligned} global\_ID &= re\_ID\_y \times numdev\_y + \\ & \quad re\_ID\_x(2D : global\_ID) \end{aligned} \tag{1}$$

$$\begin{aligned} global\_ID &= re\_ID\_z \times numdev\_y \times numdev\_x + \\ & \quad re\_ID\_y \times numdev\_y + \\ & \quad re\_ID\_x(3D : global\_ID) \end{aligned} \tag{2}$$

A dependent point indicates a direction of dependent zone. For instance, on Figure 5: the dependent point of device_3 is (-1,-1). We suppose that the length and the height of zone is M and N which are larger than 1. So the offset of relative ID is still (-1,-1). The ID of dependent zone is $(1, 1) + (-1, -1) = (0, 0)$. So the global ID can be calculated as follows:

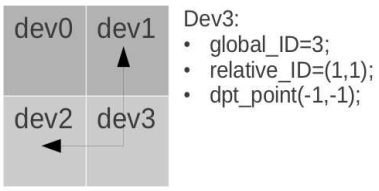


Fig. 5. The dependent points of device 3

$$global\_ID = 0 \times 2 + 0 \times 2 = 0; \tag{3}$$

In this way, we can find all the dependent zones with the list of dependent points. Then, the list of dependent zones (we call it **neighbor_list**) will be used during the data transmission or communication process.

B. Managing the data transmission between devices

1) Read/write memory objects in OpenCL: OpenCL lets users create three kinds of memory objects: buffers, 2D images and 3D images. These memory objects are stored in the host memory (typically, in RAM) or in the device memory (typically, in GRAM directly on the graphic card). There are several functions that can be used to read and write memory object. The Table I presents five functions that read and write buffer object [6].

Function	Purpose
clEnqueueReadBuffer	Reads data from a buffer object to host memory
clEnqueueWriteBuffer	Writes data from host memory to a buffer object
clEnqueueReadBufferRect	Reads a rectangular portion of data from a buffer object to host memory
clEnqueueWriteBufferRect	Writes a rectangular portion of data from host memory to a buffer object
clEnqueueCopyBuffer	Enqueues a command to copy a buffer object to another buffer object

TABLE I
READ AND WRITE BUFFER OBJECTS

2) Data transmission between multiple devices: After each iteration, the data that needs to be transferred from device **A** to device **B** can be calculated in the following way:

$$region\_aTob = exact\_write\_A \cap exact\_read\_B \tag{4}$$

The data region is an EXACT_WRITE region; the zone region is an EXACT_READ region. Thus, the formula [4] can be redefined as:

$$region\_aTob = data\_region\_A \cap zone\_region\_B \tag{5}$$

Then all the data can be transferred with the neighbor_list following the process presented in Listing 4.

```
foreach device_A
  load(data_region_a) from device_A
  foreach device_B in neighbor_list_A
    load(zone_region_b) from device_B
    transferData(region_aTob)
    wait();
```

Listing 4. The data transmission between all devices

OpenCL does not assume that data can be transferred directly between devices, so commands only exist to move from a host to device, or from a device to host. Copying data from one device to another requires an intermediate transfer to the host.

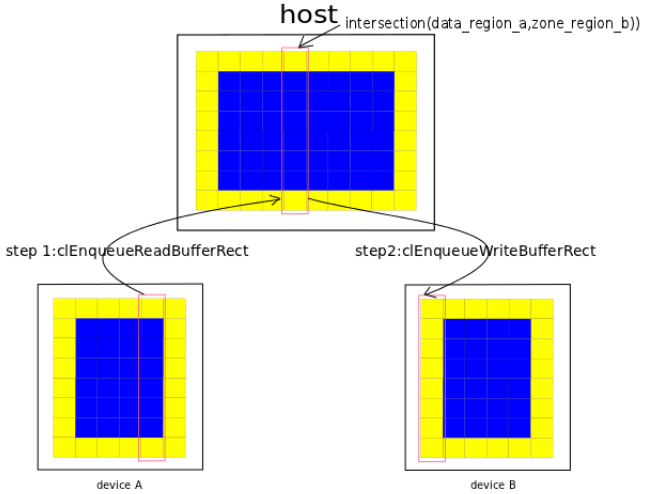


Fig. 6. Data transmission between two devices

The pointer of host memory cannot be simultaneously possessed by several devices, one device cannot communicate with host until the data transmissions of other devices have finished. This sequential process degrades the overall performance. Increasing additional ghost zone may reduce the frequency of communication, and this option is also provided in our generator.

C. Code generation

The general procedure is presented in Algorithm 1. In fact, the inputs of Algorithm 1 are described in an XML file. STEPOCL reads the script, and generates the host code without changing kernel code.

The generation procedure is composed of 3 parts: initialization of OpenCL environment, data partition and data transmission. OpenCL initialization identifies all available devices by creating corresponding Command Queues.

In the second part, subroutine $DataPartition()$ split the data space into several regions with the partitioning strategy which are mentioned in Section II. According to the information of ghost zone and relative ID, we can predict with which region the data exchange will take place. Thus, subroutine $findNeighborRegion()$ creates a $neighbor\_list$ for each region. After allocating the memory objects for each region

Algorithm 1: Host code generation

Input: NWI: Number of Work Item, WS: Work group size, DS: Data Size, DPL: Dependent Points List, DRL: Dependent Regions List, TI: Total Iteration

Output: P_{host} :Host Program

```
1  P_host ← NewHostProgram();
2  Num_Device ← clGetDeviceIDs;
3  RegionList ← NewRegionList;
4  RegionList ← DataPartition(WS, DS, DL);
5  DRL ← findNeighborRegion(RegionList, DPL);
6  foreach dev_i in deviceList do
7      dev_i ← AllocateDeviceBuffer_i_withSizeOf(RegionList(i));
8      P_host ← CopyHostToDevice(P_host, dev_i, RegionList(i));
9  foreach i in TI do
10     NWI ← KernelSpacePartition(NWI, RegionList);
11     P_host ← InvokeKernel(NWI, WS, OtherArguments);
12     synchronization();
13     foreach region_i in RegionList do
14         foreach region_b in DRL_i do
15             P_host ← transferData(Region_iTob);
16     ∀device, swap(inputBuffer, outputBuffer);
17     synchronization();
18 foreach dev_i in deviceList do
19     P_host ← CopyDeviceToHost(P_host, dev_i, RegionList(i));
```

on corresponding devices and adjusting the kernel arguments, the kernels are launched simultaneously.

In the last part, the data transmission happens after each iteration. By following the $neighbor\_list$ of each region, subroutine $transferData()$ transfers and updates the data for preparing the next execution. If all data is updated, the next execution of kernel is launched. If there are only one device, the $neighbor\_list$ will be empty, and the data transmission is not permitted.

Two synchronizations are used in this algorithm. The first ensures that the data will not be transferred during writing process while the second ensures that all data are ready for the next iteration.

The generated host code can directly be executed with the original kernel.

III. Evaluation

In order to evaluate our implementation, we generated a 4-points Jacobi 2D stencil with 50 iterations. We executed it on a machine with 4 NVIDIA GTX-460 devices. The system is Red Hat Enterprise Linux Server release 5.5. We varied the

size of the input data to highlight the new scalibity offered by our generated program.

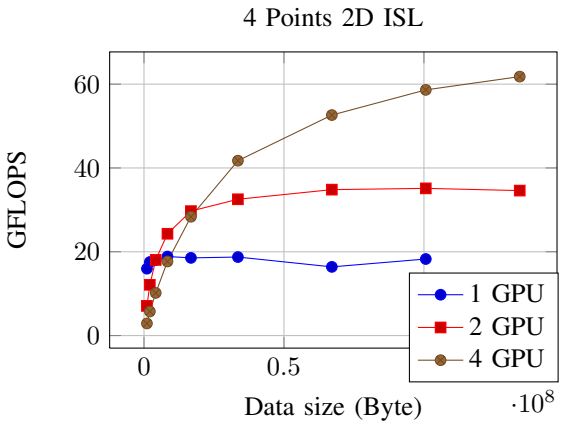
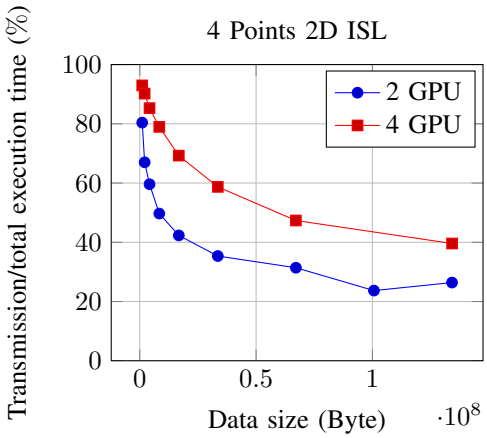


Fig. 7. Performance of stencils across architectures



;

Fig. 8. Normalized data transfer between multi-GPUs

Figure 7 evaluates the performance of the generated 2D stencil code by scaling the data size. It is measured in Floating-point Operations Per Second (FLOPS) which can be calculated as bellow:

$$Performance = Num\_iter \times Size\_date \times Flt\_Opt/Tn \quad (6)$$

The Num_iter presents the number of iterations, while Flt_Opt means the number of operations in the inner loop and Tn means the total time used for executing stencil code which also includes the time of communication between each computing device. On one hand, the results show that single GPU gains better performance with small data sizes as the number of work-groups in the kernel is not big enough to make all devices busy. In this case, the overhead of data transmission is significant in the whole execution time. Figure 8 helps to picture exactly the impact of the communication on the overall time execution in percentage (**PTE**): PTE decreases as the size of data growth. Though we use our partition strategy to avoid unnecessary extra data transmission, the PTE value is still very high. Thus, even if we will investigate in the short term how to take care of the communication – by overlapping them in order to avoid occupying CPU communication by several GPUs at the same time for example –, we above all want to take in consideration this saturation threshold as a parameter of a dynamic scheduling strategy with the purpose of determining the best compromise between data size and number of devices to use. The goal is to use only the necessary resources and not to just occupy them and avoid their exploitation for another computation. On another hand, as expected, the performance curves show aggressive growth with the increase of data. With four GPU devices, we achieved 61 GFLOPS which is 3.5 times faster than using only one GPU device and almost 1.8 times faster than using two GPU devices. The generated program even achieves to process a data input size which is impossible to treat on a single device program.

IV. Related work

A number of recent studies have focused on the parallelization of OpenCL code for multi-devices GPGPU systems. The Amdahl Software provides similar application – OpenCL CodeBench [7] . It enables developers to rapidly generate and optimize OpenCL code. The main difference between STEPOCL and OpenCL CodeBench is that the host code generated by OpenCL CodeBench is more general and users need to define their own ways of communication between several accelerators. Jungwon Kim and Honggyu Kim [8] propose an OpenCL framework that treats multiple GPUs as a single compute device. This framework analyzes the OpenCL kernel index space at run time and it performs a sampling run just before the kernel is executed. The sampling run obtains buffer access ranges of each affine array references for different GPUs. Using this information, the runtime distributes the kernel work-group index space efficiently. Sylvain Henry provides an OpenCL implementation which is called SOCL [9]. It is based on StarPU [10]. It gives a unified access to every available OpenCL device: applications can now share entities such as Events, Contexts or Command Queues between several OpenCL implementations. In addition, the Command Queues that are created without specifying a device provide automatic scheduling of the submitted commands on OpenCL devices contained in the context to which the command queue is attached. On the other hand, this implementation use dynamic analysis and scheduling the available devices at runtime. Considering our data partitioning also happens at runtime, it will be very promising to combine our research achievement.

Data transfer between the CPU and GPUs can degrade the performance. Overlapping [8], [11] the data transfer and GPU computation is a solution to reduce the overhead of data transmission.

V. Conclusion

In this paper, we introduced the design and implementation of the new OpenCL code generator STEPOCL which

provides the facilities for developing code on heterogeneous multi-device systems. Instead of developing a tedious code implementation, user only needs to provide the basic description of kernel argument, space information and the kernel function for one device. Then, STEPOCL automatically generates OpenCL code for multi-device without changing kernel functions. STEPOCL builds model of kernel and data space from the description then partition the workload with best strategy. Communication management and exchange of intermediary results between several devices are also generated. Preliminary experiments on an iterative stencil show that the generated code achieved high performance on multi-device architectures.

Further works are planned at different levels. First, we will massively enhance our tool by relaxing the information demanded to the user thanks to a cooperation with static compilation techniques: kernel and data space and ghost zones will be automatically detected when possible. Next, as STEPOCL is able to generate parametric kernel, it is able to generate non uniform distribution. We plan to collect information on available heterogeneous devices at runtime with the purpose of applying dynamic scheduling strategies and performing an accurate partitioning.

REFERENCES

[1] A. Munshi, B. Gaster, T. Mattson, and D. Ginsburg, *OpenCL Programming Guide*, ser. OpenGL. Pearson Education, 2011. [Online]. Available: http://books.google.fr/books?id=M-Sve_KItQwC

[2] J. Meng and K. Skadron, "A performance study for iterative stencil loops on gpus with ghost zone optimizations," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 115–142, 2011.

[3] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization: an overview of the pips project," in *ICS*, 1991, pp. 244–251.

[4] Insieme compiler project. [Online]. Available: http://www.dps.uibk.ac.at/insieme/

[5] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," in *ICS 09: Proceedings of the 23rd international conference on Supercomputing*, 2009, pp. 256–265.

[6] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*, D. W. Cauley, M. Thurston, and J. Fuller, Eds. Addison-Wesley, 2012.

[7] OpenCL Codebench by Amdahl Software. [Online]. Available: http://www.amdahlsoftware.com/multi-core-products/overview/

[8] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in opencl for multiple gpus," in *PPOPP*, 2011, pp. 277–288.

[9] S. Henry, A. Denis, and D. Barthou, "Programmation unifiée multi-accélérateur OpenCL," *Techniques et Sciences Informatiques*, no. 8-9-10, pp. 1233–1249, 2012. [Online]. Available: http://hal.inria.fr/hal-00772742

[10] 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. [Online]. Available: http://hal.inria.fr/inria-00550877

[11] I. Gelado, J. H. Kelm, S. Ryoo, S. S. Lumetta, N. Navarro, and W. mei W. Hwu, "Cuba: an architecture for efficient cpu/co-processor data communication," in *ICS*, 2008, pp. 299–308.