IEICE TRANS. INF. & SYST., VOL.E98–D, NO.4 APRIL 2015

PAPER

Enabling a Uniform OpenCL Device View for Heterogeneous Platforms

Dafei HUANG[†a)], **Changqing XUN**[†], **Nan WU**[†], **Mei WEN**[†], **Chunyuan ZHANG**[†], **Xing CAI**[††*], ***Nonmembers***, ***and*** **Qianming YANG**[†], ***Student Member***

SUMMARY Aiming to ease the parallel programming for heterogeneous architectures, we propose and implement a high-level OpenCL runtime that conceptually merges multiple heterogeneous hardware devices into one *virtual heterogeneous compute device* (VHCD). Moreover, automated workload distribution among the devices is based on offline profiling, together with new programming directives that define the device-independent data access range per work-group. Therefore, an OpenCL program originally written for a single compute device can, after inserting a small number of programming directives, run efficiently on a platform consisting of heterogeneous compute devices. Performance is ensured by introducing the technique of virtual cache management, which minimizes the amount of host-device data transfer. Our new OpenCL runtime is evaluated by a diverse set of OpenCL benchmarks, demonstrating good performance on various configurations of a heterogeneous system.

key words: *heterogeneous devices, OpenCL, virtualized single device, automated workload distribution, data transfer minimization*

1. Introduction

Heterogeneity has become a prevailing hardware characteristic of major platforms of the computing industry today. However, it is very difficult to program such systems that consist of multiple heterogeneous compute devices. Although the OpenCL parallel programming standard has been designed with cross-platform portability in mind, user-friendliness is hampered by the many programming details of OpenCL that a user must explicitly handle, especially when the compute devices are heterogeneous.

OpenCL programming in a nutshell. The main control of an OpenCL implementation lies inside its *host program*, which uses OpenCL APIs to submit commands to the devices for performing computations. The host program also operates, via APIs, each device's memory, which is typically organized as *buffer* objects. A *kernel* is a special function written in the OpenCL C language, to be executed on the devices. The unit of concurrent OpenCL execution is a *work-item*, which executes a kernel in a single-kernel-multiple-data manner. The programmer specifies the number of work-items associated with a kernel, and these work-items are organized as an N-dimensional range (NDRange). For workload distribution, an NDRange is divided into multiple equal-sized *work-groups*, each having a unique global ID. The work-items within each work-group have unique local IDs.

Programmer's headaches. From a programmer's point of view, there are several headaches associated with OpenCL programming. First, the programmer has to query, select and initialize the compute devices, and then create computing contexts and command queues. These steps are followed by enqueuing kernel execution, memory manipulation and synchronization commands, which are submitted to the command queues on the devices. Although an OpenCL device is actually a logical concept, vendor-supplied specific OpenCL runtimes force a programmer to submit commands to each physical device specifically. The second headache is associated with kernel workload distribution, because the existing OpenCL runtimes provide no support for automating workload division between the devices. The programmer thus has to write input data to each device's buffers and decide the NDRange size on each device, etc. For heterogeneous devices, the programmer considers the specific hardware of each device before assigning to it a number of work-groups, typically relying on some possibly inaccurate early experience. The third headache concerns data management. When a distributed kernel finishes on all the involved devices, the programmer has to manipulate inter-device data transfers if needed. Last but not least, a change in the hardware configuration may require the OpenCL code to be manually re-tuned for performance.

Our solution. As seen above, OpenCL programming is not very user-friendly for heterogeneous compute devices. To improve the programmability and handle the above headaches, we propose in this paper a high-level OpenCL runtime. The rationale is that user-friendliness will arise from a uniform compute device view, despite whether the underlying devices are homogeneous or heterogeneous. We want to relieve the programmer of cumbersome details about the number and types of devices, and also the tasks of workload division and data transfer.

In addition to proposing the high-level OpenCL runtime, this paper also has contributions in (1) balanced workload distribution, (2) analysis of work-group buffer access range, and (3) data buffer management. More specifically, automated offline profiling is adopted to facilitate a fair dis-

Manuscript received July 14, 2014.
Manuscript revised November 5, 2014.
Manuscript publicized January 20, 2015.

[†]The authors are with School of Computer, National University of Defense Technology, 410073, Changsha, P. R. China.
[††]The author is with the Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, NO-0316 Oslo, Norway.
[*]Presently, with Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway.
a) E-mail: hdafei@acm.org

DOI: 10.1587/transinf.2014EDP7244

tribution of workload among heterogeneous devices. The OpenCL C syntax is extended with directives that provide device-independent buffer access pattern definitions. A virtual software-managed distributed cache is introduced between the host and the underlying devices, with dedicated replacement and write-back strategies, for the purpose of minimizing device-host-device data transfers.

The remainder of this paper is structured as follows. Section 2 gives an overview of the new OpenCL runtime and some key design issues. Section 3 explains the implementation details. The new OpenCL runtime is evaluated by a set of benchmarks in Sect. 4, while Sect. 5 discusses the related work. Finally, Sect. 6 addresses the limitations and proposes some future work.

2. Design Overview

As shown in Fig. 1, we want a new OpenCL runtime that encompasses different vendor-specific OpenCL runtimes and, at the same time, allows the programmer to operate multiple heterogeneous compute devices through a single *virtual heterogeneous compute device* (VHCD). That is, only one VHCD is visible and initialized by the programmer, providing a virtualized device view. The VHCD-supported runtime automatically distributes workload across the underlying multiple devices. Underneath the VHCD-supported runtime, vendor-provided OpenCL runtimes directly control execution of subtasks on the underlying devices. Moreover, the new OpenCL runtime manages data transfers between the multiple physical devices.

Based on the FreeOCL [1] framework, which is an open-source CPU implementation of OpenCL for Linux, we have implemented the new VHCD-supported runtime. Our OpenCL runtime is designed as an *installable client driver* (ICD), consistent with the OpenCL standard. To prepare the reader for the implementation details of the new OpenCL runtime in Sect. 3, let us first provide an overview of its VHCD-enabling design.

Balanced workload distribution. We use an offline profiling approach (similar to [2]) to workload distribution, as a tradeoff between static performance modeling [3] and a dynamically scheduled task pool [4]. To ensure good partitioning accuracy, the computing capability of each device is measured by an offline code execution, which practically incurs little overhead. The work-group index space of a kernel is then automatically partitioned into N work-group assignments, the same as the number of underlying compute devices available.

Analysis of work-group buffer access range. Another desirable feature is automated identification of the data part(s) needed by each device. This relies on the knowledge of the buffer access range of all the work-groups. Full software analysis of the buffer access range, however, may incur large overheads in form of time and/or register usage. This is especially true in the presence of a large number of fine-grain data items, for each we need to maintain and manipulate base, bound, and dependency information. Nevertheless, these overheads can be significantly reduced by aggregating all the buffer access range information into a small number of pattern template items. Therefore, we propose new compiler directives, called *buffer access pattern definition* (BAPD), to be inserted into an OpenCL code for describing the pattern of work-group buffer accesses. Our new VHCD-supported runtime parses the inserted BAPD directives to determine the buffer access range of each work-group assignment.

Data buffer management. We maintain a virtual VHCD memory that maps to the host memory. The memory space of each physical device is an image of a portion of the VHCD memory. Since the devices can only operate with local data, data transfers between the VHCD and the devices are needed for maintaining data consistency. To avoid redundant data transfers, we introduce the technique of buffer cache management, which maintains data distribution information on the memory hierarchy.

3. Implementing the VHCD-Supported OpenCL Runtime

The new OpenCL runtime works in two phases: offline profiling and actual execution, as shown in Fig. 2. During the profiling phase, three main modules are in action: kernel parser, kernel profiler and workload distributor. The first module is designed based on Clang [5], and translates each kernel command from the VHCD-command queue. The

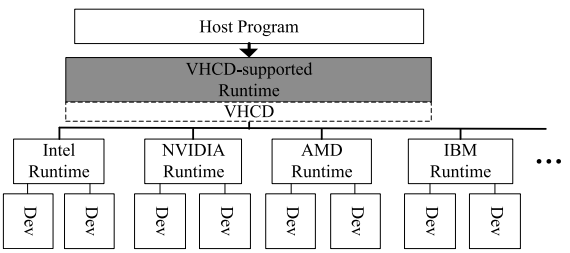


Fig. 1 A virtualized uniform device view enables the VHCD runtime to operate multiple heterogeneous compute devices.

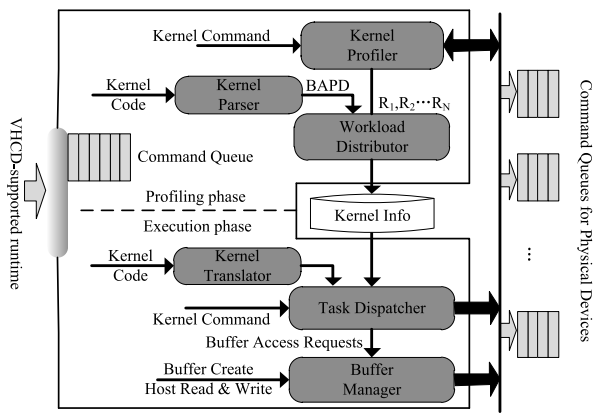


Fig. 2 The VHCD-supported OpenCL runtime has two phases: profiling and execution, each consisting of a number of modules.

second module profiles kernel execution on each device. The third module computes a workload partitioning and distributes all commands to the devices, in addition to storing the related partitioning information in a data structure called *KernelInfo*. If the system configuration remains unchanged, the KernelInfo structure can be reused, thus no need for new profiling.

During the execution phase, three more main modules are in action: kernel translator, task dispatcher and buffer manager. The first module translates an original kernel code into a distributed form (as subtasks) intended for the devices. The second module generates operations of the subtasks to be executed on the devices. The third module queues the operations to each device's command queue.

3.1 Offline Workload Profiling and Partitioning

To ensure a fair workload partitioning across N heterogeneous devices, such that the overall kernel execution time is minimized, we make the following assumption.

Assumption: All the work-groups are executed in the single-kernel-multiple-data manner. The kernel execution time on any given device is linearly proportional to the number of assigned work-groups. Moreover, the time needed by two different physical devices to complete a single work-group has a fixed ratio in between.

The key to our assumption is a linear relationship between the kernel execution time and the number of launched work-groups. That relationship is basically a step function. However, we can assume that the execution times of work-groups are even. Although some kernels with conditional branches may conflict with this assumption, these branches are avoidable (e.g. Stencil2D using "halo points" to resolve the boundary conditions), or only result in small variations in execution times (e.g. DCT8x8 ignoring the blocks outside of image). What's more, due to the massive number of work-groups, the step intervals are comparatively very small to the work-group count as well as the total execution time. So a linear relationship can apply here.

For verification, we have used an NVIDIA Tesla C2050 GPU to run both benchmark MatrixMul from NVIDIA OpenCL SDK and benchmark Stencil2D from SHOC [6]. The first test program multiplies two dense matrices (thus compute-bound), and the latter applies a 9-point stencil on a 2D grid (thus memory-bound). Kernel execution time was measured against different numbers of work-groups launched, and the results of MatrixMul and Stencil2D are shown in Fig. 3a) and Fig. 3b), respectively. It is obvious that the two plots confirm our assumption.

Our OpenCL runtime distributes the workload across heterogeneous devices based on a simple philosophy. Suppose the kernel execution time per work-group on device i is T_i, $i = 1, \ldots, N$. An ideal distribution that minimizes the overall execution time, where R_i denotes the fraction of work-groups assigned to device i, can be described by the following equations:

$$T_1 * R_1 = T_2 * R_2 = \ldots = T_N * R_N \quad (1)$$

$$R_1 + R_2 + \ldots + R_N = 1 \quad (2)$$

Therefore, we can obtain $R_i = 1/T_i * (1/T_1 + 1/T_2 + \ldots + 1/T_N)^{-1}$ as a fair fraction of the work-groups that should be assigned to device i.

We remark that offline profiling needs to be performed for each compute device and each new application, or every time the code is changed. The associated overhead is usually negligible. For a kernel that is repeatedly executed in an actual OpenCL program, a single kernel execution is sufficient for the profiling purpose. Moreover, only a relatively small number of work-groups need to participate in the single kernel execution, because what we want to measure is the time usage per work-group.

3.2 Defining Buffer Access Pattern

Most typically, an OpenCL kernel traverses a loop. Each work-item corresponds to one or multiple iterations of the loop body. The two major memory access patterns associated with looping are stride and index [7], [8]. In the stride pattern, the data blocks accessed by adjacent loop iterations have a fixed distance, and the data blocks have a uniform size. Each work-item thus accesses a data block or multiple data blocks with a fixed stride. In the index pattern, both the stride and the block size are determined by an index array, both may vary with the loop index. Since a work-group consists of contiguous work-items, they should have the same access pattern.

Observation: There generally exists a pattern for the work-items (and work-groups) to access a buffer. Moreover, for each buffer, the access range of a work-group in the index space can be denoted by four attributes, $\{P, S, D, N\}$, where P is the start address, S is the size of each data block, D is the stride between two consecutive data blocks, N is the number of data blocks.

We have extended the OpenCL C syntax to allow programmers to define a work-group's access pattern for each buffer, as shown in Fig. 4. The unit for P, S and D is byte. By default, the value of both P and D is $\{0, 0, 0\}$, the value of S is $\{1, 1, 1\}$, and the value of N is 1. Figure 4a) and Fig. 4b) show, respectively, the BAPD directives that can be inserted into the Stencil2D benchmark for specifying the access pattern of its input buffer (dataBuf1) and output buffer (dataBuf2). Each rectangle in Fig. 4 represents a

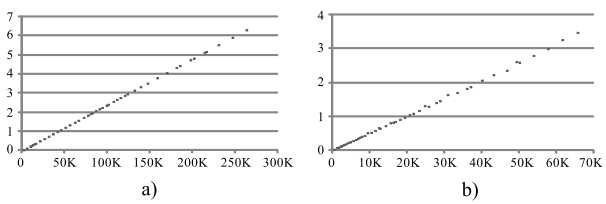


Fig. 3 Kernel execution time of benchmark MatrixMul (a, in seconds) and Stencil2D (b, in milliseconds), as function of the number of work-groups launched.

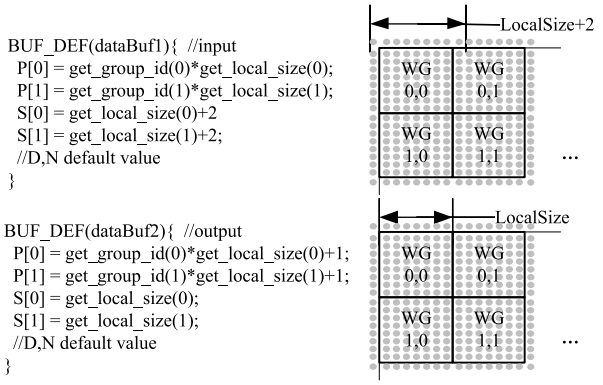


Fig. 4 BAPD directives used for the input and output buffers of the Stencil2D benchmark.

work-group.

3.3 Distributing Workload among Multiple Devices

For the efficiency of kernel execution, we have adopted two principles for workload distribution:

Principle 1: Each work-group assignment should be expressed in a single OpenCL NDRange. That is, the work-group IDs are contiguous in each dimension.

Principle 2: Subtask partitioning is done in the highest dimension. That is, for a 2D index space, partitioning is performed in rows; for a 3D index space, partitioning is performed in horizontal planes.

The size of the partitions may not be constant, dependent on the underlying physical devices. If a partitioning does not satisfy Principle 1, the distributed kernel needs to be launched multiple times, causing excessive overhead of kernel launching. Principle 2 concerns the efficiency of host-device data transfers. It ensures that the buffer access range of each subtask is a continuous memory region, which can be transferred between host and devices by a single command. Otherwise, multiple commands may be enqueued for each subtask, which can significantly reduce the overall effective bandwidth.

During the profiling phase, after the BAPD directives are read by the kernel parser, the R_i values will be computed by the kernel profiler. Using the above two principles, the workload distributor determines the partitioning and records it into the KernelInfo data structure, to be used during the execution phase. The information stored inside KernelInfo has three parts. In addition to the kernel name, the other two parts are as follows:

- The workload partitioning description stored as the offset and size of each work-group assignment.
- The buffer access range description of each device for each involved buffer. In particular, the four attributes $\{P, S, D, N\}$, given by the BAPD directives, enable the workload distributor to transform 2D/3D data blocks to a series of {offset, size}, denoting the buffer access range in 1D flat memory space.

```
1 KernelName: Stencil2D
2 WorkLoadDistribution:
3  Dev0:(0,0,0), (4096,1024,1)      Dev1:(0,1024,0), (4096,3072,1)
4 BufferList:
5  dataBuf1: Dev0:Read 0,16818192          Dev1: Read 16805898,50389008
6  dataBuf2: Dev0:Write 16392,16809996 Dev1: Write 16822290,50356224
```

Fig. 5 An example of the auto-generated KernelInfo data structure associated with the Stencil2D benchmark, which uses a 4096×4096 grid.

Figure 5 shows an example of KernelInfo associated with Stencil2D, where the global size is (4096,4096,1) and the size of each work-group is (16,16,1). Two heterogeneous devices are assumed, and the partitioning ratio is 1:3.

3.4 Executing Distributed Kernels

Using information stored in KernelInfo, the task dispatcher sends a series of commands to each device to execute the distributed kernels. There are four steps corresponding to clEnqueueNDRangeKernel().

1. Use KernelInfo to generate buffer access requests to the buffer manager;
2. Use the buffer segment table to locate requested input data; If needed, carry out device-host-device data transfer and update the segment table;
3. Execute the distributed kernel on devices;
4. Use KernelInfo to update the segment table of each output buffer.

The first, second and the fourth steps are mainly completed by the buffer manager to be described in Sect. 3.5. The task dispatcher sends buffer access requests to the buffer manager according to the BufferList description of KernelInfo. In the following text of Sect. 3.4, we will describe the third step, which executes a distributed kernel on multiple devices.

The behavior of each work-item executed on a single device should be consistent with that when executed on multiple devices. After the workload partitioning, the index space of a work-group assignment may not be consistent with the original programmer-visible kernel index space. It can cause execution errors of the distributed kernel. For example, the code segment of the kernel for a vector addition with memory coalescing is shown in Fig. 6a), while Fig. 6b) shows the situation when the kernel is executed on a single device, assuming global size is 16 and work-group size is 1. The work-item with global ID 12, for instance, will iterate twice, referencing a[12] and a[28]. Figure 6c) shows the situation of two devices, where the five last work-items are executed on physical device 1 as a subtask. The global ID of the previous work-item is 1, and the global size is 5. This means that the work-item will iterate seven times, referencing a[1], a[6], a[11], a[16], a[21], a[26] and a[31].

It can be seen from the above example that the inconsistency in index space affects not only the buffer locations accessed by a work-item, but also the number of inner iterations per work-item. To solve the problem, we modify the kernel command which sends the distributed kernel to the command queues of the physical devices, as shown in

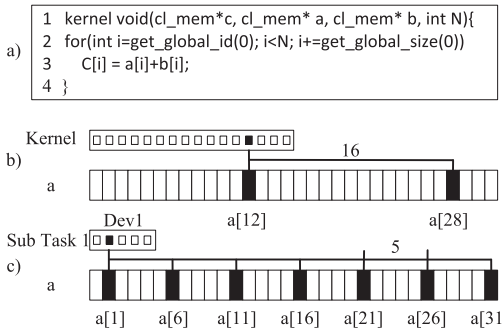


Fig. 6 Example execution error of a distributed vector add kernel.

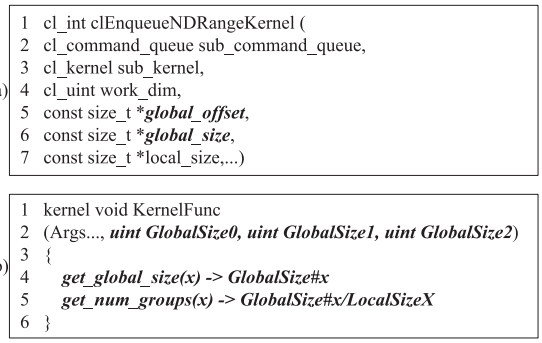


Fig. 7 Example kernel code translated by the kernel translator.

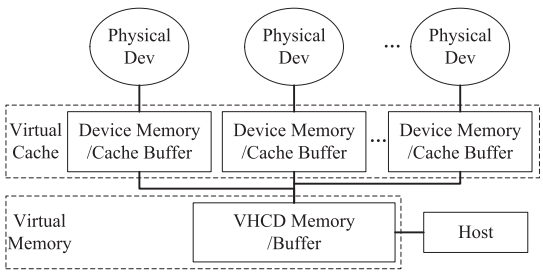


Fig. 8 Two-level memory hierarchy of VHCD.

Fig. 7a). The parts in bold italic denote the differences with the original kernel command. We set global_size as the size of the work-group assignment for the device, and global_offset is set to shift the start address of the NDRange in the distributed kernel, ensuring correct values of global ID and work-group ID.

In addition, based on Clang [5], we have designed a source-to-source translator to transform the original kernel code into the distributed kernel code. As shown in Fig. 7b), after the transformation, extra global size arguments are inserted. These three arguments denote different dimensions respectively. In other words, the original global size is passed to the distributed kernel as arguments, and functions get_global_size() and get_num_groups() are transformed to use the corresponding arguments. These three arguments are initiated by the task dispatcher when invoking clEnqueueNDRangeKernel() during the execution phase.

3.5 Managing a Virtual Software-Managed Distributed Cache

Our OpenCL runtime uses a two-level memory hierarchy as shown in Fig. 8. To distinguish a buffer object that is created in the virtual VHCD memory from another object created in a device's memory, we call the former a *buffer object*, in short *buffer*, and the latter a *cache buffer object*, in short *cache buffer*. The runtime allocates the VHCD memory on the host and treats it as the main memory. The on-device memory parts are treated as a virtual software-managed distributed cache. Whenever a clCreateBuffer() function is invoked during the execution phase, a buffer in VHCD memory, the corresponding on-device cache buffers, and an initial segment table are created simultaneously. The buffer manager considers the virtual cache as a write-back cache, and adopts a cache coherency protocol similar to the MSI (modified, shared, invalid) protocol. The virtual cache has three new features:

1. The host may directly access buffers in the VHCD memory without touching the virtual cache. However, data in the cache buffers may become invalid.
2. Both buffers and cache buffers are referenced by macro instructions, e.g., clEnqueueWriteBuffer() and clEnqueueNDRangeKernel(). Each memory reference is a bulk reference, enabling an efficient software management of cache coherency.
3. The virtual cache actually corresponds to the device memory of the devices. Furthermore, the buffer in the VHCD memory and the corresponding cache buffers in the virtual distributed cache are of the same size. Thus capacity miss will not happen in the virtual cache.

To record the data distribution state across all the devices, the buffer manager maintains a buffer segment table for each buffer. Each item of this table is a buffer segment representing a portion of the corresponding buffer with the same state, described by three arguments {offset, size, tag}. Offset and size denote a segment's start address and size (in bytes). Tag is a bitfield variable, revealing on which device the valid data is stored. Value 1 means valid, 0 means invalid. The length of a segment can be as short as one single byte, which ensures that no redundant data transfers will happen. On the other hand, feature 3 from above ensures that the maintenance overhead is acceptable.

Figure 9 shows the state machine of each byte in a buffer. Buffer read/write requests may be invoked by the host program (called in short as host read/write) or by the distributed kernel executed on any physical device (called in short as $Pdevice_i$ read/write), as shown in Table 1. The three states are defined as follows:

1. Modified: This byte has been modified in one cache buffer, thus inconsistent with the VHCD memory. The byte has to be written back to the VHCD memory, before other devices or the host can read it.
2. Shared: This byte is unmodified and exists in at least one cache buffer.

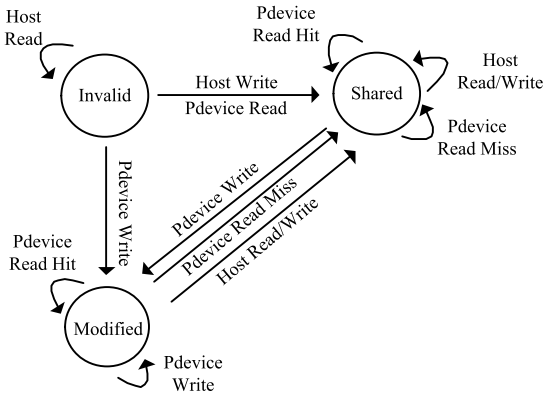


Fig. 9 State machine of each byte in the memory system of VHCD.

Table 1 An overview of actions that need to be carried out by the buffer manager, in the case of different read/write requests.

Requests	Actions
$Pdevice_i$ write	**Modified/Shared/Invalid**: Nothing.
$Pdevice_i$ read	**Modified**: If read miss, write the segment back and then transfer it to the requesting Pdevice. **Shared/Invalid**: If read miss, transfer the segment to the requesting Pdevice.
Host write	**Modified/Shared/Invalid**: Copy the data segment from user space into the VHCD memory and transfer it to all Pdevices.
Host read	**Modified**: Write the data segment from $Pdevice_i$ back to VHCD memory, and then copy it to the user space. **Shared/Invalid**: Copy the data segment from VHCD memory to user space.

3. Invalid: This byte must be fetched from the VHCD memory if it is to be stored in this cache buffer.

The buffer access requests from the host or $Pdevice_i$ are described as {offset, size, mode(write/read), source(host/$Pdevice_i$)}. The two first arguments are used to locate a data segment in a buffer as in the segment table, whereas mode denotes a read or write data segment in the buffer or its copy in a cache buffer, and source denotes the requester. Table 1 lists the actions (except state updating) that can be taken by the buffer manager. Buffer access requests to different segments within different states may cause different actions. After the actions are completed, the state of each segment will be updated according to the state machine shown in Fig. 9. A segment accessed by different requests may be split to multiple segments, and adjacent segments with the same state will be merged. It can be seen that data transfer is only needed when $Pdevice_i$ read miss happens.

3.6 Putting Everything Together

Let us use benchmark Stencil2D as an example, which has a single kernel invoked iteratively. Figure 10 shows, during the kernel's iterative execution, the state changes of the segment tables for buffers dataBuf1 and dataBuf2, plus the inter-device data transfers needed. In the figure, grey bars with symbols of {} denote buffer access operations. Operations for buffer dataBuf1 or cache buffer dataBuf1 are found in the left, operations for buffer dataBuf2 or cache buffer dataBuf2 are in the right; {offset,size} in these buffer operations and the segment tables are expressed by a bar's location and length. W and R denote write and read operations, while Host, Dev0 or Dev1 indicates who performs the operation. Every symbol of → denotes actual inter-device data transfer caused by buffer access operations. A black bar denotes transferred data location. Note that communication between Dev0 and Dev1 is actually Dev0-Host-Dev1. Other bars denote segment tables, where the tag of each segment has 3 bits, corresponding to Dev0, Dev1, Host. The shading in a segment table reveals the owners of valid data. Slash denotes Dev0, backslash denotes Dev1, and vertical bar denotes Host. Shared segments are marked with all owners' flags.

As shown in Fig. 10, Step 1 is HostWrite(dataBuf1). The whole dataBuf1 array is written to Host, Dev0, and Dev1. The segment table of dataBuf1 thus contains one item, with tag '111', denoting that all the devices have valid data, whereas the segment table of dataBuf2 contains one item with tag '000'. According to the KernelInfo structure shown in Fig. 5, the partitioning gives two subtasks, 25% of work on Dev0, 75% on Dev1. Using KernelInfo, the buffer manager knows that the input to the subtask on Dev0 consists of the upper 1/4 of the input array and one additional row as its bottom boundary, while the input to the subtask on Dev1 consists of the lower 3/4 of the input array and one additional row as its top boundary. The outputs of the subtasks are similar to the inputs except the boundary rows.

Step 2 does one execution of the kernel, with array dataBuf1 as input and array dataBuf2 as output. There arise two $Pdevice_i$ read requests and two $Pdevice_i$ write requests as shown in Fig. 10. The $Pdevice_i$ write request changes the segment table of dataBuf2 into two items, denoting that Dev0 owns valid the upper part of dataBuf2, while Dev1 owns valid the lower part of dataBuf2. No actual data transfers are needed no matter read or write request.

The kernel is executed again in Step 3, with dataBuf2 as input, dataBuf1 as output. Dev0 and Dev1 have the same input requirement for dataBuf2 as the previous kernel execution for dataBuf1. There arise two $Pdevice_i$ read requests and two $Pdevice_i$ write requests as shown in Fig. 10. However, Dev0 only has the upper part of array dataBuf2, while Dev1 only has the lower part. $Pdevice_i$ read operations will cause cache misses. In other words, the boundary needed by Dev0 belongs to Dev1, and vice versa. Inter-device data transfer is thus invoked.

Suppose array dataBuf1 is the output of the final kernel execution. There is one host read operation as shown in Step 4. The upper part of dataBuf1 belongs to Dev0, while the lower part belongs to Dev1. The buffer manager invokes the needed device-host data transfers to read data from buffers Dev0 and Dev1 to the host.

We can see for Stencil2D, there is only boundary data exchange between the devices, as efficient as manually scheduled. Note that a data transfer caused by host read/write does not count due to unavoidable communica-

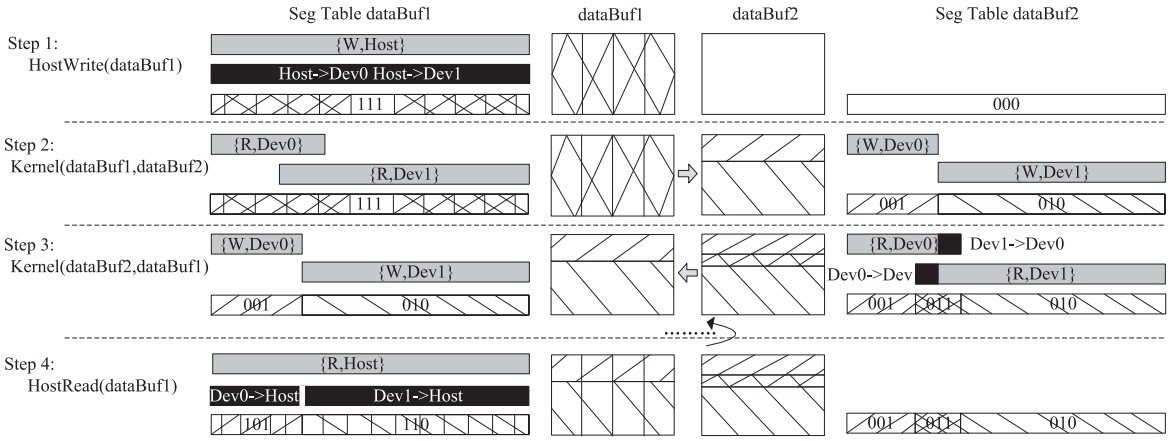


Fig. 10 State changes of the segment tables during execution of benchmark Stencil2D on a VHCD with two compute devices.

tion also in the case of manual scheduling. Introducing the virtual cache into the VHCD system is very important for real applications, because it minimizes inter-device data transfers between kernel executions (e.g., invoking a single kernel several times or invoking multiple kernels), without having to use manual scheduling.

4. Evaluation

We tested the performance of our VHCD-supported OpenCL runtime using a heterogeneous system that consists of two Intel Xeon E5620 quad-core CPUs, one NVIDIA Tesla C2050 GPU, and one NVIDIA GTX 460 GPU, plus 16GB DDR2 main memory. The operating system is Red Hat Enterprise Linux 5. The GPUs communicate with the CPUs via a PCI-E Gen.2 x16 bus that uses point-to-point serial links. The associated data transfer rate is 8GBps (full duplex). We used four different hardware configurations denoted as C1–C4, as defined in Table 2. The two E5620 CPUs are treated as a single OpenCL compute device, denoted as E5620 in this paper.

Eight OpenCL benchmarks were chosen from several sources: Parboil [9], SNU NPB [10], NVIDIA, and SHOC [6]. Some details are given in Table 3. The benchmarks were selected to cover a wide spectrum of computation types and execution features. MatrixMul implements a dense matrix multiplication. SAD computes the sum of absolute differences kernel, used in MPEG video encoders. EP generates pairs of Gaussian random deviates using the Marsaglia polar method. DCT8x8 implements the discrete cosine transform (DCT) for an 8x8 block. NBODY simulates the evolution of a system of bodies. Stencil2D applies a 9-point stencil operation to a 2D grid. FDTD3D applies a finite differences time domain progression stencil on a 3D surface. FFT3D solves a 3D partial differential equation using the fast Fourier transform. In Table 3, "single kernel" means that there is only one kernel in the test benchmark, whereas "single execution" means the kernel is executed once. Take MatrixMul for instance. There is only one kernel in this benchmark and it is executed only one time, although it may be executed multiple times for performance measuring. For the Stencil2D benchmark, however, it needs to be executed multiple times and there is data dependency between the different executions.

Table 2 Four heterogeneous hardware configurations used for testing the new OpenCL runtime. (E5620 means two E5620 CPUs combined.)

Name	Hardware Configuration
C1	E5620 + G460
C2	E5620 + C2050
C3	G460 + C2050
C4	E5620 + G460 + C2050

Although the original source codes of all the benchmarks were designed for a single OpenCL device, we easily ported them to a heterogeneous platform without any modification (except inserting a few BAPD directives). The correctness of the VHCD runtime was verified by checking the numerical results, and wall-clock timing was used for the time measurements. The overhead associated with data load (clEnqueueWriteBuffer) and store (clEnqueuReadBuffer) was ignored.

4.1 Results

Figure 11 shows that our VHCD-supported runtime gave a performance boost for almost all the benchmarks on all the four heterogeneous hardware configurations. The results also confirm that the VHCD runtime can, in this truly heterogeneous scenario, balance the workload distribution and automate the required data transfers. Speedup of the first four benchmarks is very close to be perfect, due to the absence of inter-device data transfer and good load balance between the devices. Speedup of the last four benchmarks is less perfect due to unavoidable data transfers. When the ratio of computation versus data transfer is large, the speedup is very good (such as for NBODY and Stencil2D). The worst case occurred with FDTD3D, especially the speedup obtained on configuration C4 is less than that on C2 and C3.

Table 3 Benchmarks used for evaluating the VHCD-supported OpenCL runtime. Column A shows the number of kernels, column B shows the number of executions for each kernel, column C shows whether there is inter-device data transfer through the main memory.

Benchmark	Source	Remarks	A	B	C	NDRange	Execution
MatrixMul	NVIDIA	Matrix dimension: 8192x8192	1	1		(8192,8192,1)	Single kernel, single kernel execution
SAD	Parboil	4x4,search range 33x33,input image 1920x1080	1	1		(1920,1080,1)	Single kernel, single kernel execution
EP	SNU NPB	Class C	1	1		(32768,1,1)	Single kernel, single kernel execution
DCT8x8	NVIDIA	Input image 4096x8192	1	1		(4096,8192,1)	Single kernel, single kernel execution
NBODY	NVIDIA	32768 bodies, 100 iters, single precision	1	100	X	(32768,1,1)	Single kernel, multiple kernel executions
Stencil2D	SHOC	Size 4(4096x4096:16x16), 1000 iters	1	1000	X	(4096,4096,1)	Single kernel, multiple kernel executions
FDTD3D	NVIDIA	376x376x376, radius 4, 1000 timesteps	1	1000	X	(376,376,376)	Single kernel, multiple kernel executions
FFT3D	SNU NPB	256x256x256, double precision	3	1;1;1	X	(256,256,1)	Multiple kernels

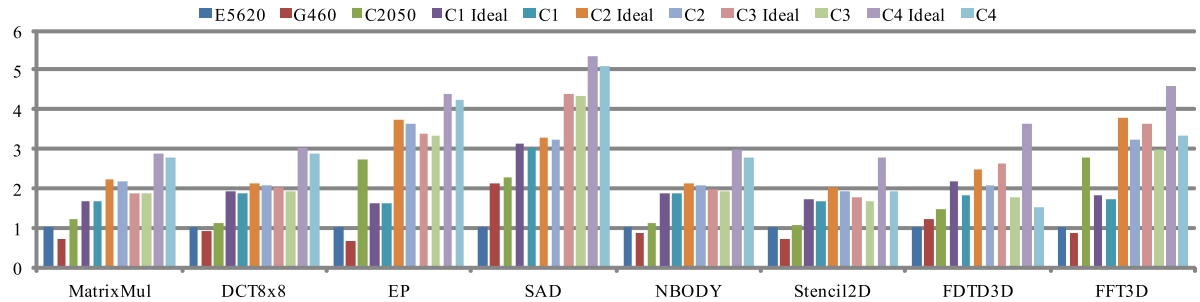


Fig. 11 Speedup obtained by the VHCD-supported runtime, in comparison with only using E5620. The value of "Cx Ideal" is calculated by summing up the individual speedup of all the devices found in one heterogeneous configuration.

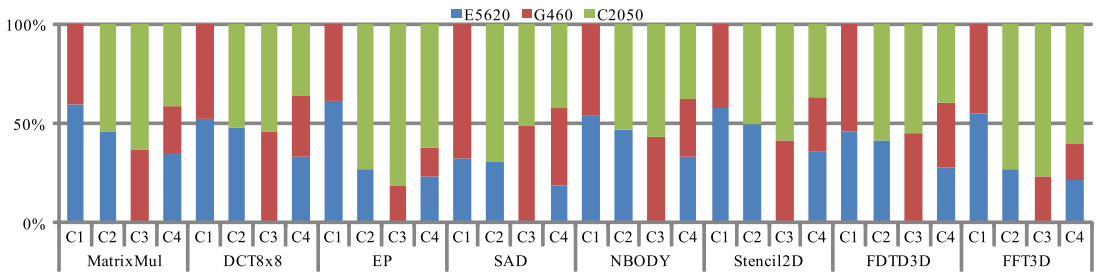


Fig. 12 Workload partitionings (between heterogeneous devices) that are automatically derived by the VHCD-supported runtime.

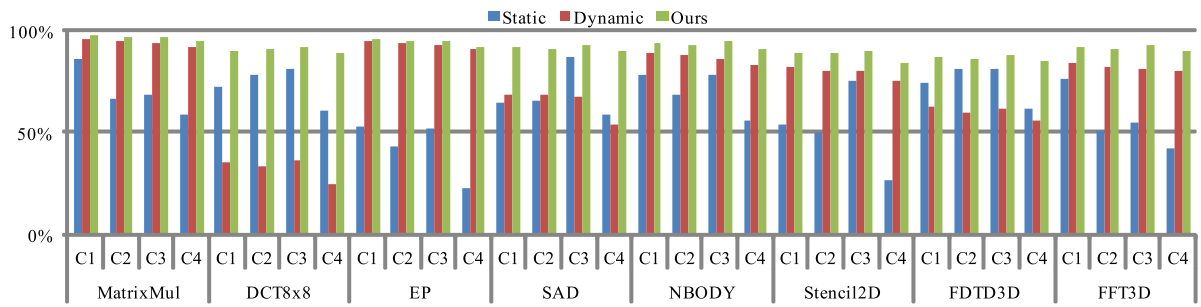


Fig. 13 A comparison of three workload distribution schemes with respect to the utilization of the heterogeneous devices.

Figures 12 and 13 investigate the workload distribution that is automatically performed by the VHCD-supported runtime. Figure 12 shows, for all the eight benchmarks, the fractions of workload that are distributed to the heterogeneous devices in each of the four hardware configurations. This partitioning information was generated by the offline profiling. It can be seen that the new OpenCL runtime can distribute the workload accordingly, when the benchmark or heterogeneous hardware configuration varies. Figure 13 shows the utilization level of heterogeneous devices using three different load balancing approaches, where "static" means the workload is equally divided among the devices,

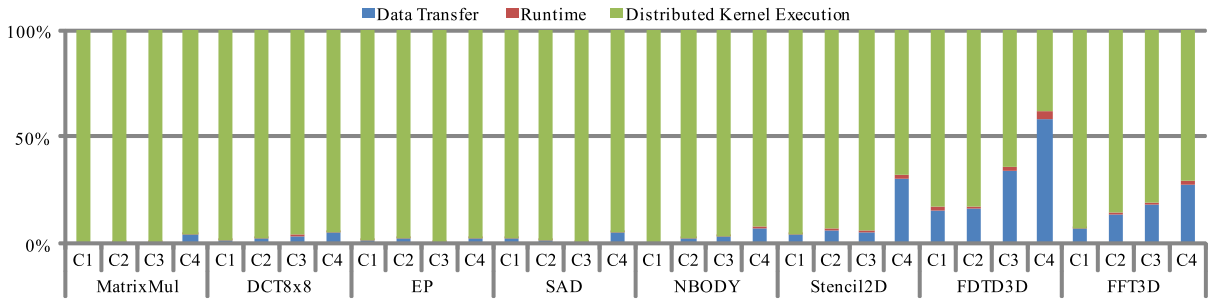


Fig. 14 Time usage breakdown of the eight benchmarks on the four heterogeneous hardware configurations.

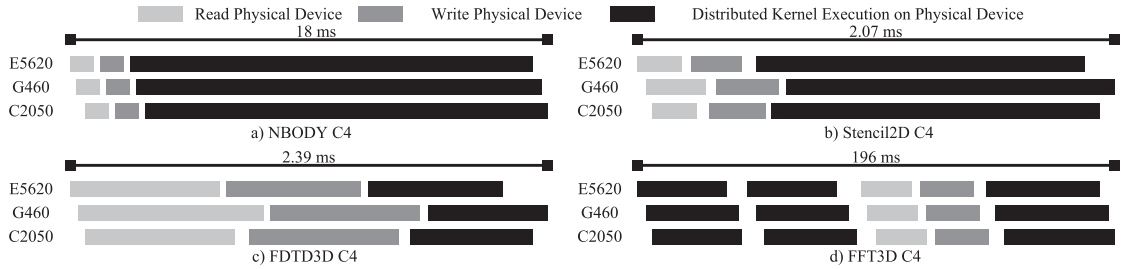


Fig. 15 Execution flows of four benchmarks on configuration C4.

"dynamic" means subtasks are dynamically dispatched from a task pool consisting of 64 subtasks, whereas "ours" refers to the approach adopted by the VHCD-supported runtime. The results in Fig. 13 show that our offline-profiling strategy can achieve a high level of device utilization. For compute-intensive benchmarks, such as MatrixMul, EP and NBODY, the result of the dynamic workload distribution is close to our method, because the runtime scheduling overhead is relatively low. Otherwise, the dynamic workload distribution performs poorly. In short, a fair distribution of the workload among heterogeneous devices is facilitated by our VHCD-supported runtime.

Figure 14 focuses on the runtime and data transfer overhead, by showing the breakdown of kernel execution time, which consists of three parts: data transfer between devices, overhead of the VHCD runtime (including task dispatching, maintaining buffer segment tables), and distributed kernel execution time. Since using three devices results in more inter-device data transfers than using two devices, configuration C4 always spends a larger portion of the time on data transfers. However, compared with C1–C3, C4 employs more compute devices thus requiring the least overall execution time. For memory-intensive applications, such as the last four benchmarks, data transfers between many devices can potentially be a scaling problem. We want to remark, though, that the same scaling problem still exists if the inter-device communications are hand-coded (instead of automatically handled by the OpenCL runtime). Another observation is that the overhead due to the VHCD-supported runtime itself is very low, almost invisible in Fig. 14.

Furthermore, Fig. 15 shows the kernel execution flows of four benchmarks on configuration C4, with inter-device data transfers automatically handled by the VHCD-supported runtime. The execution times of NBODY, Stencil2D and FDTD3D are nearly the average costs of one kernel iteration. How data transfers account for a significant percentage of the total execution time is also shown. The inherent computation-communication feature of FDTD3D makes its data transfers relatively the most expensive. Please note that with our runtime, data transfers cannot start until all devices have finished their subtasks, which is shown clearly in the flow of FFT3D. This results in a relatively poor overlap of kernel execution and data transfers, while carefully hand-tuned programs may avoid that.

Table 4 The relative amount of data transfer with the virtual software-managed distributed cache, in comparison with a "master-slave" approach

Benchmark	C1&C2&C3	C4
NBODY	60%	75%
Stencil2D	1%	2%
FDTD3D	2%	4%
FFT3D	33%	50%

To see the effectiveness of the virtual software-managed distributed cache, which minimizes the inter-device data transfers, we have made a comparison with a "master-slave" approach where the devices always send/retrieve their entire local data to/from a host. Table 4 thus shows, for four benchmarks, the relative amount of data transfer induced by the virtual cache, in comparison with that of the "master-slave" approach. (The smaller values the better.) It can be seen that, for a given benchmark, the relative amount of data transfer is the same for C1, C2 and C3. This is because the three hardware configurations all involve two devices. The advantage of using the virtual cache

is especially large for benchmarks that only require boundary data exchanges, such as Stencil2D and FDTD3D. Note also that, if direct inter-device data transfer is supported by the underlying hardware, the data transfer cost will decrease further.

5. Related Work

Although computing systems with accelerators have become mainstream, there still is a lack of support for automatic task scheduling and data consistency when multiple (heterogeneous) devices exist in the system.

The idea of using a single image that encompasses multiple devices was first introduced in [11], which is an important inspiration to our work. However, there are three different aspects between [11] and the present paper. First, workload distribution among heterogeneous devices was not considered in [11]. Second, we have presented the novel technique of buffer management, which was given little attention in [11]. Third, [11] adopted an automated sampling method for finding the minimum and maximum addresses of the work-groups. This method is potentially problematic because one needs to assume that the addresses are linear functions of the local and global IDs, which is often not true. Moreover, only considering the minimum and maximum addresses can be inappropriate for work-groups that operate with multi-dimensional data and/or use coalesced memory accesses.

Most recently, Pandit et al. [12] designed Fluidic Kernels which shares very similar idea with our work. Their proposed runtime utilizes a dynamic workload distribution scheme, so that the distribution, data transfer and data coherence are managed dynamically according to the intermediate execution status. Comparing to our method, neither user-specified memory access range nor profiling is needed, but extra runtime overhead is induced due to dynamic management which is discussed in Fig. 13.

Sun et al. [13] designed a task queuing extension for OpenCL that provides a high-level, unified execution model tightly coupled with a resource management facility. In multi-GPU environments, the work pool-based task queuing extension allows the programmer to easily adapt the scheduling policy of OpenCL kernels to fit the environment. Kim et al. [14] proposed an OpenCL framework for heterogeneous CPU/GPU clusters. The framework can achieve both high performance and ease of programming. The framework also provides an illusion of a single system for the user. Application developers can thus utilize multiple heterogeneous compute devices, such as multicore CPUs and GPUs, in a remote node as if they were in a local node. No communication API, such as the MPI library, is required in the application source.

In addition to OpenCL, other models have also been proposed to program heterogeneous platforms. Qilin [2] is a proposed programming model based on filters. However, users of Qilin have to write two versions of each filter for CPU and GPU, respectively. Offline profiling is also adopted by Qilin to obtain information about each task on each compute device. Huynh et al. [15] described an efficient and scalable code generation framework that can map general-purpose streaming applications onto a multi-GPU system. They proposed an efficient stream graph partitioning algorithm that partitions the complex applications to achieve the best performance under a given shared-memory constraint. Kunzman et al. [16] are developing a unified programming model that can be used for all cores, host and accelerator alike. They discussed the modifications they have made to the runtime system, along with discussing future modifications. They developed a simple molecular dynamics program executing on a mixture of x86 and Cell processors without requiring hardware-specific code within the application code.

For domain specific computing, MINT [17] and PHYSIS [18] use compiler directives or mathematical formulas to assist automated generation of stencil computing GPU code. A programming model and runtime were proposed in [19] to target solving dense linear algebra problems on multiple accelerators. Dynamic scheduling of workload distribution was adopted in that work.

6. Discussion and Future Work

The OpenCL standard and implementations from various vendors provide researchers with an opportunity to program heterogeneous platforms using the same programming language. Our VHCD-supported OpenCL runtime pushes this one step further, allowing an OpenCL program originally written for a single compute device to run seamlessly on a heterogeneous system that has multiple compute devices. We believe that our work is a timely contribution, because of the increasing availability of CPU-GPU heterogeneous systems. When a *same* OpenCL program wants to target different platforms, our VHCD-supported OpenCL runtime can be a useful tool. Here, it indeed makes sense to view a collection of heterogeneous devices as a virtualized single device.

The only required manual effort is to insert a few compiler directives. To the best of our knowledge, this VHCD-supported runtime is the first to automatically exploit multiple heterogeneous devices from different vendors. Experiments with a diverse set of benchmarks have confirmed the usability and performance of the new OpenCL runtime.

However, there are six limitations with the new OpenCL runtime:

- The linear assumption in the profiling phase. Violations will lead to unbalanced workload distribution, which lowers the overall performance.
- The BAPD directives need to be inserted manually.
- BAPD can only describe memory access patterns that can be represented by linear functions. For irregular memory accesses which are seldom used, programmers cannot represent the memory access pattern with BAPD at all, which is still an unsolved problem in this

research area to the best of our knowledge.

- The size of the VHCD (our proposed virtual device) global memory must be equal to the smallest global memory that physical devices have. Because the global memory of each device is regarded as a full-size cache of the VHCD global memory. The runtime will report an error when try to create memory objects if there is a violation.
- Performance portability of the OpenCL program is left untouched. A device-specific program, when executed directly on another device, typically cannot achieve good performance [20], [21]. With poor performance portability, the use of multiple heterogeneous devices will not be beneficial but cause performance penalties.
- The inherent scaling problem with regard to relatively large number of heterogeneous devices still exists.

As future work, we will first focus on the second limitation. We have started to use a linear function with a set of linear constraints, which is known to enable precise analysis of data dependence and control dependence, to represent the array access pattern for each global array reference in a kernel [22]. We will try to integrate an array access analysis tool based on this representation to discover buffer access range automatically, instead of manually inserting BAPD directives.

Performance portability of OpenCL program is an unavoidable issue. We have started to build a translator to enable performance portability between devices by translating a single OpenCL source code to optimized device-specific code [22]. We will further improve the translator and integrate it into VHCD. What's more, the translator will also perceive the performance and figure out how to distribute the workload, which resolves the last limitation.

Acknowledgments

The authors gratefully acknowledge the support from National Natural Science Foundation of China under No. 61033008, 61103080 and 61272145, 863 Program of China under No. 2012AA012706.

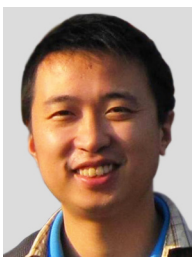
References

[1] R. Brochard and N. Nikolaev, "Multi-platform implementation of OpenCL 1.2 targeting CPUs," http://code.google.com/p/freeocl/, accessed June 5. 2013.
[2] C.K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," Proc. 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pp.45–55, New York, NY, USA, 2009.
[3] D. Grewe and M.F.P. O'Boyle, "A static task partitioning approach for heterogeneous systems using OpenCL," Compiler Construction, ed. J. Knoop, Lecture Notes in Computer Science, vol.6601, pp.286–305, Springer, 2011.
[4] M. Boyer, S. Che, K. Skadron, J. Gummaraju, and N. Jayasena, "Automatic intra-application load balancing for heterogeneous systems," AMD Fusion Developer Summit, 2011.
[5] C. Lattner, "LLVM and Clang: Advancing compiler technology," FOSDEM '11: Free and Open Source Developers' European Meeting, Brussels, Belgium, 2011.
[6] A. Danalis, G. Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, and J.S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU '10, pp.63–74, New York, NY, USA, 2010.
[7] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory access scheduling," Proc. 27th Annual International Symposium on Computer Architecture, ISCA '00, pp.128–138, Vancouver, BC, Canada, June 2000.
[8] N. Wu, M. Wen, J. Ren, Y. He, C. Xun, W. Wu, and C. Zhang, "Cache streamization for high performance stream processor," International Conference on High Performance Computing (HiPC), pp.140–149, Kochi, India, Dec. 2009.
[9] J.A. Stratton, C. Rodrigues, I.J. Sung, N. Obeid, L.W. Chang, N. Anssari, G.D. Liu, and W.W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Tech. Rep. IMPACT-12-01, Center for Reliable and High-Performance Computing, 2012.
[10] S. Seo, G. Jo, and J. Lee, "Performance characterization of the NAS parallel benchmarks in OpenCL," IEEE International Symposium on Workload Characterization (IISWC), pp.137–148, Austin, TX, USA, 2011.
[11] J. Kim, H. Kim, J.H. Lee, and J. Lee, "Achieving a single compute device image in OpenCL for multiple GPUs," Proc. 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '11, pp.277–288, New York, NY, USA, 2011.
[12] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices," Proc. Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14, pp.273–283, Orlando, FL, USA, 2014.
[13] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli, "Enabling task-level scheduling on heterogeneous platforms," Proc. 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU-5, pp.84–93, New York, NY, USA, 2012.
[14] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "OpenCL as a unified programming model for heterogeneous CPU/GPU clusters," Proc. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, pp.299–300, New Orleans, LA, USA, 2012.
[15] H.P. Huynh, A. Hagiescu, W.F. Wong, and R.S.M. Goh, "Scalable framework for mapping streaming applications onto multi-GPU systems," Proc. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, pp.1–10, New Orleans, LA, USA, Feb. 2012.
[16] D. Kunzman and L. Kale, "Programming heterogeneous systems," IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp.2061–2064, Anchorage, AK, USA, 2011.
[17] D. Unat, X. Cai, and S.B. Baden, "Mint: realizing CUDA performance in 3D stencil methods with annotated C," Proc. International Conference on Supercomputing, ICS '11, pp.214–224, New York, NY, USA, 2011.
[18] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers," Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pp.11:1–11:12, Seattle, WA, USA, 2011.
[19] G. Quintana-Ortí, E.S. Quintana-Ortí, A. Remón, and R.A. van de Geijn, "An algorithm-by-blocks for supermatrix band Cholesky factorization," in High Performance Computing for Computational Science - VECPAR 2008, ed. J.M.L.M. Palma, P.R. Amestoy, M. Daydé, M. Mattoso, and J.C. Lopes, Lecture Notes in Computer Science, vol.5336, pp.228–239, Springer-Verlag, 2008.

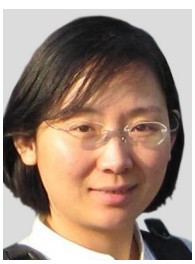
[20] S. Pennycook, S. Hammond, S. Wright, J. Herdman, I. Miller, and S.A. Jarvis, "An investigation of the performance portability of OpenCL," Journal of Parallel and Distributed Computing, vol.73, pp.1439–1450, Elsevier, Nov. 2013.
[21] H. Dong, D. Ghosh, F. Zafar, and S. Zhou, "Cross-platform OpenCL code and performance portability for CPU and GPU architectures investigated with a climate and weather physics model," Proc. 2012 41st International Conference on Parallel Processing Workshops, pp.126–134, Pittsburgh, PA, USA, Sept. 2012.
[22] D. Huang, M. Wen, C. Xun, D. Chen, X. Cai, Y Qiao, N. Wu, and C. Zhang, "Automated transformation of GPU-specific OpenCL kernels targeting performance portability on multi-Core/Many-core CPU," Proc. 20th International European Conference on Parallel and Distributed Computing, pp.210–211, Porto, Portugal, Aug. 2014.

Dafei Huang was born in 1987. He is a Ph.D. candidate in School of Computer at National University of Defense Technology. His research interests include parallel programming, compiler optimization and runtime design.

Changqing Xun was born in 1983. He is a research assistant in School of Computer at National University of Defense Technology. His research interests include parallel programming, compiler optimization and runtime design.



Nan Wu was born in 1980. He is an associate professor in School of Computer at National University of Defense Technology. His research interests include computer architecture, parallel programming and computer network.



Mei Wen was born in 1975. She is an associate professor in School of Computer at National University of Defense Technology. Her research interests include computer architecture, parallel programming and scientific computing.

Chunyuan Zhang was born in 1964. He is a professor in School of Computer at National University of Defense Technology. His research interests include computer architecture, parallel programming, low power design, embedded systems, media processing and scientific computing.

Xing Cai was born in 1968. He is a professor in Department of Informatics at University of Oslo and Simula Research Laboratory. His research interests include parallel programming, high-performance scientific computing, numerical methods and generic PDE software.

Qianming Yang was born in 1984. He is a Ph.D. candidate in School of Computer at National University of Defense Technology. His research interests include computer architecture, parallel programming, advanced memory design and embedded system.