Achieving a Single Compute Device Image in OpenCL for Multiple GPUs *

Jungwon Kim Honggyu Kim Joo Hwan Lee Jaejin Lee

Center for Manycore Programming
School of Computer Science and Engineering
Seoul National University, Seoul 151-744, Korea
http://aces.snu.ac.kr
{jungwon, honggyu, joohwan}@aces.snu.ac.kr, jlee@cse.snu.ac.kr

Abstract

In this paper, we propose an OpenCL framework that treats multiple GPUs as a single compute device. Providing the single GPU image makes an OpenCL application written for a single GPU portable to the GPGPU systems with multiple GPUs. It also makes the application exploit the full computing power of the multiple GPUs and the entire amount of GPU memories available in the system. Our OpenCL framework automatically distributes at run time an OpenCL kernel written for a single GPU into multiple CUDA kernels that execute on the multiple GPUs. It applies a run-time memory access range analysis to the kernel by performing a sampling run and identifies an optimal workload distribution for the kernel. To achieve a single compute device image, the runtime maintains a virtual device memory that is allocated in the main memory of the GPGPU system. The OpenCL runtime treats the memory as if it were the memory of a single GPU device and keeps it consistent to the memories of the multiple GPUs. Our OpenCL-C-to-C translator generates the sampling code from the OpenCL kernel code and our OpenCL-C-to-CUDA-C translator generates the CUDA kernel code for the distributed OpenCL kernel. We show the effectiveness of our OpenCL framework by implementing the OpenCL runtime and the two source-to-source translators. We evaluate its performance with a GPGU system that contains eight GPUs using eleven OpenCL benchmark applications.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming; D.3.4 [*PROGRAMMING LANGUAGES*]: Processors – Code generation, Compilers, Optimization, Run-time environments

General Terms Algorithm, Design, Experimentation, Languages, Measurement, Performance

Keywords OpenCL, Compilers, Runtime, Access range analysis, Workload distribution, Virtual device memory

* This work was supported by grant 2009-0081569 (Creative Research Initiatives: Center for Manycore Programming) from the National Research Foundation of Korea funded by the Korean government (Ministry of Education, Science and Technology). ICT at Seoul National University provided research facilities for this study.

1. Introduction

Open Computing Language (OpenCL)[7] is a standard for general-purpose, heterogeneous parallel programming. OpenCL provides a uniform programming environment for software developers to write portable code across heterogeneous parallel platforms, such as multicore CPUs, GPUs, Cell-BE processors, and DSPs. OpenCL consists of a programming language, called OpenCL C, and OpenCL runtime API functions. OpenCL C is a subset of C99 with some extensions and it is used to write computational kernels that execute on a compute device, such as a CPU and a GPU. OpenCL API functions are used to configure and manage OpenCL objects and to coordinate executions of the computational kernels. Since OpenCL has a strong CUDA[8, 15] heritage, a general-purpose GPU (GPGPU) system is a representative heterogeneous platform that OpenCL is targeting.

A recent GPU[18] contains more than 400 stream processors, capable of achieving more than 1 TFLOPS for single precision and more than 500 GFLOPS for double precision. A single GPU can deliver the supercomputing power at 1/20th of the power consumption and 1/10th of the cost when compared to the latest general-purpose, quad-core CPU. With the C-like CUDA or OpenCL language, software developers can develop GPU applications easily without using complex graphics programming APIs. High processing power, cost effectiveness, energy efficiency, and ease of programming provided by GPUs make them a natural choice for high performance computing.

Moreover, in support of this trend, many GPGPU systems start to have multiple GPU devices to solve large size problems[19, 20, 26, 28]. While GPGPU systems with multiple GPUs are widening their user base, the users still manually write code from scratch to exploit the multiple GPUs available in the system. In addition, converting an application written for a single GPU to run on multiple GPUs generally requires rewriting the code, and sometimes fairly extensive modifications are required. The programmer needs to insert code for distributing workload across multiple GPUs and managing data between the host main memory and multiple GPU device memories. Guaranteeing consistency between the multiple copies of data makes this process more difficult for the programmer.

In this paper, we propose an OpenCL framework that provides an illusion of a single compute device to the programmer for the multiple GPUs available in the system. It enables OpenCL applications written for a single compute device to run on the system with multiple GPUs without any modification. Moreover, our framework makes it possible to run the GPU applications that require a bigger size of data than the device memory size of a single GPU.

At run time, when the information about an OpenCL kernel index space is fully known to our OpenCL runtime, it distributes the kernel workload across multiple GPUs. To distribute the kernel workload across multiple GPU devices efficiently, our framework performs a sampling run just before the kernel is executed. The sampling run obtains the memory access range of each GPU for the kernel. The runtime distributes the kernel workload according to this memory access range information to minimize data transfer between the CPU and multiple GPUs.

To achieve a single compute device image, the OpenCL runtime maintains a virtual device memory that is allocated in the main memory of the GPGPU system. The runtime treats the virtual device memory as if it were the memory of a single GPU and keeps it consistent to the memories of the multiple GPUs. Our OpenCL-C-to-C translator generates the sampling code from the OpenCL kernel code and OpenCL-C-to-CUDA-C translator generates each GPU's CUDA kernel code for the distributed OpenCL kernel.

The major contributions of this paper are the following:

- We describe the design and implementation of our OpenCL framework for a GPGPU system with multiple GPU devices.
- We propose a run-time memory access range analysis technique. It detects efficiently the memory access ranges of each GPU for a given kernel work-group index space.
- We propose a run-time workload distribution technique that is optimal with regards to three optimality constraints: expressibility in a single grid of CUDA thread blocks (i.e., for the minimum number of kernel launches), workload balancing between multiple GPUs, and the minimum amount of data transfer.
- We describe translation techniques used in our two source-to-source translators: OpenCL-C-to-C and OpenCL-C-to-CUDA-C.
- We show the effectiveness of our OpenCL framework by implementing the OpenCL runtime and the two source-to-source translators. We evaluate its performance with a GPGPU system that contains 8 GPUs using 11 OpenCL benchmark applications.

The rest of the paper is organized as follows. Section 2 briefly describes OpenCL and its features. Section 3 explains the techniques of achieving a single compute device image in our OpenCL framework. Section 4 discusses and analyzes the evaluation result of our OpenCL framework. Section 5 describes related work. Finally, Section 6 concludes the paper.

2. Background

In this section, we briefly describe the OpenCL platform and its execution semantics. OpenCL inherited many features from CUDA[8, 15].

The OpenCL platform consists of a host processor connected to one or more *compute devices*, each of which contains one or more *compute units* (CUs). Each CU contains one or more *processing elements* (PEs). A PE is a *virtual* scalar processor. In addition, OpenCL defines four distinct memory regions: global, constant, local and private. *Compute device memory* consists of the global and constant memory regions. The local memory is shared by all PEs in the same compute unit. The private memory is local to a PE. Accesses to the global memory or the constant memory may be cached in the global/constant memory data cache if there is such a cache in the device. The OpenCL runtime maps each OpenCL platform component to a component in the target GPGPU system. The entire GPU card becomes an OpenCL compute device.

An OpenCL application consists of a *host program* and *kernels*. The host program executes on the host processor and submits *commands* to perform computations on the PEs within a compute

```
void matrixMul(float* C, float* A, float* B, int WA, int HA, int WB)
{
    for (int j = 0;  j < HA; j++) {
        for (int i = 0; i < WB; i++) {
            float acc = 0.0f;
            for (int k = 0; k < WA; k++) {
                acc += A[k + j * WA] * B[i + k * WB];
            }
            C[i + j * WB] = acc;
        }
    }
}
```

(a)

```
__kernel void matrixMul( __global float* C, __global float* A,
                         __global float *B, int WA, int WB)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    float acc = 0.0f;
    for (int k = 0; k < WA; k++)
        acc += A[k + j * WA] * B[i + k * WB];
    C[i + j * WB] = acc;
}
```

(b)

```
#define BLOCK_SIZE 16
void main()
{
    ...
    cl_mem A_gpu, B_gpu, C_gpu;
    float *A_cpu, *B_cpu, *C_cpu;
    int WA, WB, WC;  // widths of matrices
    int HA, HB, HC;  // heights of matrices
                     // HB=WA, WC=WB, and HC=HA
    ...
    // Initialize the OpenCL runtime
    ...
    // Build and create the kernel
    ...
    // Allocate the main memory space for the matrices
    A_cpu = (float*) malloc(sizeA);
    B_cpu = (float*) malloc(sizeB);
    C_cpu = (float*) malloc(sizeC);
    // Initialize A_cpu, B_cpu, and C_cpu
    ...
    // Allocate the device memory for the matrices
    A_gpu = clCreateBuffer(context, CL_MEM_READ_ONLY,
                           sizeA, NULL, NULL);
    B_gpu = clCreateBuffer(context, CL_MEM_READ_ONLY,
                           sizeB, NULL, NULL);
    C_gpu = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                           sizeC, NULL, NULL);
    ...
    // Copy the matrices from the main memory to the device memory
    clEnqueueWriteBuffer(command_queue, A_gpu, CL_TRUE, 0, sizeA,
                         A_cpu, 0, NULL, NULL);
    clEnqueueWriteBuffer(command_queue, B_gpu, CL_TRUE, 0, sizeB,
                         B_cpu, 0, NULL, NULL);
    ...
    // Set up kernel arguments
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &C_gpu);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &A_gpu);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &B_gpu);
    clSetKernelArg(kernel, 3, sizeof(int), &WA);
    clSetKernelArg(kernel, 4, sizeof(int), &WB);
    // Set the size of the global index space
    size_t globalWorkSize[] = {WC, HC};
    // Set the size of the local index space
    size_t localWorkSize[] = {BLOCK_SIZE, BLOCK_SIZE};
    // Execute the kernel
    clEnqueueNDRangeKernel(command_queue, kernel, 2, 0,
                           globalWorkSize, localWorkSize, 0, NULL, NULL);
    // Copy the result from the device memory to the main memory
    clEnqueueReadBuffer(command_queue, C_gpu, CL_TRUE, 0, sizeC,
                        C_cpu, 0, NULL, NULL);
}
```

(c)

Figure 1. (a) A C function for matrix multiplication. (b) The OpenCL kernel for the C function in (a). (c) The OpenCL host program for matrix multiplication.

device or to manipulate memory objects. There are three types of commands: kernel execution, memory, and synchronization. A kernel is a function and written in OpenCL C. It executes on a single compute device. It is submitted to a *command-queue* in the form of a kernel execution command by the host program. A compute device may have one or more command-queues. Commands in a command-queue are issued in-order or out-of-order depending on the queue type.

When the host program submit a kernel execution command to a command-queue, it defines an N-dimensional abstract index space for the kernel, where $1 \leq N \leq 3$. Each point in the index space is specified by an N-tuple of integers with each dimension starting at 0. Each point is associated with an execution instance of the kernel, which is called *work-item*. Thus, the N-tuple becomes the global ID of the associated work-item. Each work-item performs a different task based on its ID in an SPMD[4] manner. Before enqueueing a kernel command, the host program defines an integer array of length N (i.e., the dimension of the index space) that specifies the number of work-items in each dimension of the index space.

A *work-group* contains one or more work-items. Each work-group has a unique ID that is also an N-tuple. An integer array of length N specifies the number of work-groups in each dimension of the index space. A work-item in a work-group is assigned to a unique local ID within the work-group, treating the entire work-group as an index space. The index space is called a local index space. The global ID of a work-item can be computed with its local ID, work-group ID, and work-group size. Work-items in a work-group execute concurrently on the PEs of a single CU.

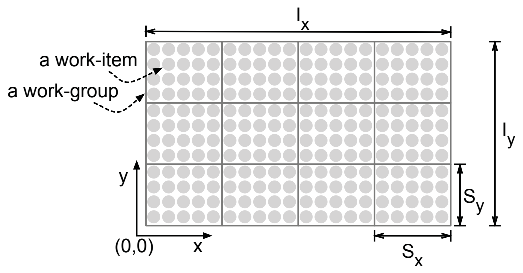


Figure 2. A two-dimensional index space.

For example, Figure 2 shows a two-dimensional index space whose sizes in dimensions x and y are I_x and I_y respectively. Suppose that the work-group size in dimension x is S_x and in dimension y is S_y. Let $(ID_x^{global}, ID_y^{global})$, $(ID_x^{local}, ID_y^{local})$, and $(ID_x^{group}, ID_y^{group})$ be the global ID, local ID, and work-group ID of a work-item in the index space, respectively. The number of work-groups in dimension x is computed by I_x/S_x and in dimension y is by I_y/S_y. Each work-group contains $S_x \times S_y$ work-items. The global ID is computed with the work-group size, work-group ID and local ID,

$$ID_x^{global} = S_x \cdot ID_x^{group} + ID_x^{local}$$

$$ID_y^{global} = S_y \cdot ID_y^{group} + ID_y^{local}$$

The host program enqueues memory commands that operate on memory objects in the device memory. Only the host program can dynamically create global or constant memory objects with OpenCL API functions. Pointers to the memory objects are passed as arguments to a kernel that accesses the objects. A memory object in the device memory is typically a *buffer object*, called in short as a *buffer*. A buffer stores a one-dimensional collection of elements that can be a scalar data type, a vector data type, or a user-defined structure.

For an example of the OpenCL kernel, consider a C function that performs matrix multiplication in Figure 1 (a). It computes $A \times B$ and stores the result in C. Figure 1 (b) shows an OpenCL kernel function that implements the C function. It is written in OpenCL C. OpenCL C has four address space qualifiers to distinguish different memory regions: `__global`, `__constant`, `__local` and `__private`. The `__global` qualifier is used in the argument declaration of the kernel. It tells the OpenCL C compiler that the buffers of matrices A, B, and C are allocated in the global memory. The kernel has a two-dimensional index space. OpenCL functions `get_global_id(0)` and `get_global_id(1)` return the first and second elements of the global ID of the work-item that executes the kernel, respectively. Each work-item computes an element of C.

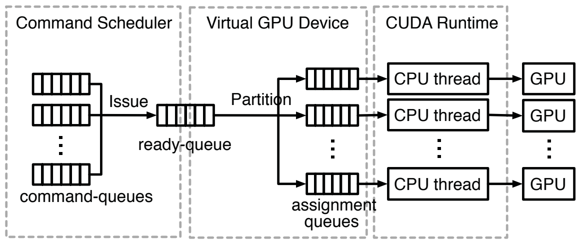


Figure 3. The organization of our OpenCL runtime

Actions taken by the host program	Tasks performed by our OpenCL runtime
1) Allocate buffers in the device memory for the input and output arrays by invoking **clCreateBuffer()**	• Allocate the requested buffers in the virtual device memory and record the information about the buffers
2) Copy the input arrays from the main memory to the buffers by invoking **clEnqueueWriteBuffer()**	• Copy the arrays to the virtual buffers
3) Set up kernel arguments by invoking **clSetKernelArg()**	• Record the information about the kernel arguments
4) Execute the kernel on the device by invoking **clEnqueueNDRangeKernel()**	• Perform a sampling run on the host to obtain access ranges of the virtual buffers by the kernel • Compute an optimal distribution of the workload in the kernel and the virtual buffers across multiple GPU devices • Allocate device buffers in each device memory based on the distribution • Copy the necessary chunks of the virtual buffers to the real buffers in each device • Execute the distributed kernel on the multiple devices
5) Copy the output arrays from the device to the host by invoking **clEnqueueReadBuffer()**	• Update the virtual buffers by comparing the updated device buffers with the twins of shared regions in the virtual buffers • Copy the requested buffers in the virtual device memory to the arrays in the host

Table 1. The tasks performed by our OpenCL runtime.

Figure 1 (C) shows the host program for the matrix multiplication. At the beginning, the host initializes the OpenCL runtime by creating an OpenCL context and a (in-order) command-queue for the compute device. Then, it builds and creates the kernel by invoking the OpenCL C compiler. The host allocates spaces for matrices A, B, and C (subscripted with _cpu) in the main memory (lines 12 – 15) and initializes them (lines 16 – 17). It also allocates buffer objects for matrices A, B, and C (subscripted with _gpu) in the device memory (lines 18 – 24). After allocating the buffers, the host copies matrices A and B from the main memory to the buffers by enqueueing memory commands to the command-queue (lines 26 – 30). After setting up kernel arguments to be passed to the kernel (lines 32 – 37), it specifies the dimension and size of the global and local index spaces (lines 38 – 41). The host executes the kernel by enqueueing a kernel command to the command-queue (lines 42 – 44). To copy matrix C from the buffer to the main memory, the host enqueues a memory command to the command-queue (lines 45 – 47). The command-queue is an in-order queue. Thus, the enqueued memory and kernel commands are issued to the compute device in order.

3. Achieving a Single Compute Device Image

In this section, we describe our technique that achieves a single compute device image from multiple GPUs.

3.1 Overview of Our OpenCL Runtime

Figure 3 shows the organization of our OpenCL runtime. It consists of a command scheduler thread, a virtual GPU device thread, and a CUDA runtime. The host program thread and the command scheduler thread share command-queues. The command scheduler dequeues each command-queue one by one and schedules the dequeued commands in the ready-queue. The command scheduler honors the type (in-order or out-of-order) of each command-queue and synchronization enforced by the host program. Our OpenCL runtime can handle OpenCL applications that have multiple host threads. In this case, each command-queue may contain commands from different host threads.

The virtual GPU device thread provides an illusion of a single GPU to the user. It dequeues the ready-queue. If the dequeued command is a kernel command, it partitions the work-group index space of the kernel into N *work-group assignments* to the N GPUs available in the system. A set of work-groups that is executed by a GPU is called a *work-group assignment*. The virtual GPU device thread encapsulates proper CUDA driver API functions[17] (e.g., cuLaunchGrid(), cuParamSetv(), cuMemcpyHtoD, cuMemAlloc(), etc.) and their arguments in an *assignment object* for each work-group assignment. Then it enqueues the objects in N *assignment queues* one by one.

An assignment queue is associated with a CUDA CPU thread. Each CUDA CPU thread is in charge of one GPU. Each CUDA CPU thread in the CUDA runtime dequeues its assignment queue. By extracting CUDA driver API functions and their arguments from the dequeued assignment object, the CUDA CPU thread sends appropriate CUDA commands to the associated GPU.

To achieve a single device image, the virtual GPU device maintains *virtual device memory* that is allocated in the main memory of the system. The virtual GPU device treats the memory as if it were the device memory of a single GPU and keeps it consistent to memories of the multiple GPUs. To distinguish a buffer object that is allocated in the virtual device memory from that allocated in a real GPU's memory, we call the former as a *virtual buffer object*, in short *virtual buffer*, and the latter as a *device buffer object*, in short *device buffer*.

In our OpenCL framework, an OpenCL kernel is translated into a parameterized CUDA kernel by our OpenCL-C-to-CUDA-C translator. When an OpenCL kernel is built in an OpenCL host program, our OpenCL-C-to-CUDA-C translator is invoked. When the work-group index space of the kernel is known at run time, our OpenCL runtime partitions the index space into N disjoint sets of work-groups (i.e., work-group assignments), where N is the number of GPUs available in the system. Then, each assignment is translated into a grid of CUDA thread blocks. Since CUDA does not allow a three-dimensional grid of thread blocks, our OpenCL runtime flattens a three-dimensional OpenCL work-group index space to a two-dimensional CUDA grid.

An overview of tasks performed by our OpenCL runtime is shown in Table 1. The first column in the table shows a typical sequence of actions taken by the host program to execute an OpenCL kernel on a single compute device (e.g., the sequence in the host program for matrix multiplication shown in Figure 1 (c)). The second column shows the tasks performed by our OpenCL runtime to enable each action taken by the host program.

3.2 Workload Distribution

Our OpenCL framework automatically distributes the workload contained in a kernel across multiple GPUs at run time when the associated kernel command is issued and the kernel index space is known. The unit of workload distribution is a work-group in the index space. Each buffer accessed by the kernel is also distributed over the multiple GPUs. Our workload distribution technique is based on the following two observations:

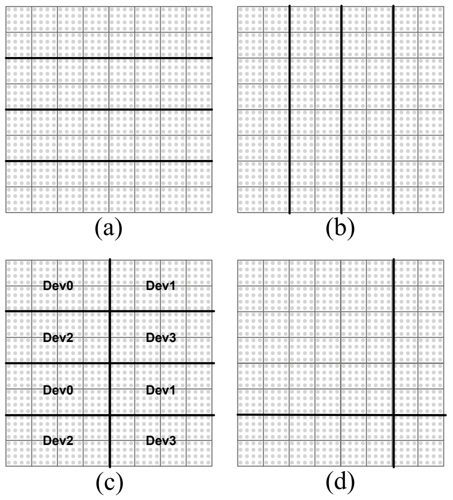


Figure 4. Different ways of partitioning the work-group index space for four GPUs.

Observation 1. *In OpenCL, there is no synchronization mechanism available between work-groups in the same kernel index space[7]. Thus, all work-groups in the same kernel index space are totally independent with each other.*

In other words, there is no dependence between work-groups in the same kernel index space. Moreover,

Observation 2. *According to the OpenCL memory consistency model[7], an update to a global memory location by a work-group does not need to be visible to other work-groups in the same kernel index space during the kernel execution.*

These two observations imply that we can freely distribute work-groups in the same index space across multiple GPUs and still preserve the original semantics of the OpenCL kernel.

We assume that there are enough work-items in a single work-group to make all processor (scalar) cores in the streaming multiprocessor of the GPU busy. We also assume that there are enough work-groups in the work-group index space to make all streaming multiprocessors in the GPU busy. There are three constraints on partitioning the work-group index space. One is about the expressibility of each work-group assignment in the partition with a single grid of CUDA thread blocks.

Constraint 1. *Each work-group assignment in the partition should be expressed in a single grid of CUDA thread blocks. In other words, the work-group IDs in each work-group assignment need to be contiguous in each dimension.*

For example, assume that there are four GPU devices and the kernel has a two-dimensional work-group index space. We can partition the work-group index space into disjoint work-group assignments in many ways. Figure 4 shows some of the partitions. A gray square in Figure 4 represents a work-group and a dot represents a work-item. There are four work-group assignments in the partition shown in Figure 4 (c). Since each assignment consists of two non-adjacent sets of eight work-groups, we cannot express the assignment with a single CUDA grid of thread blocks. For each set of eight work-groups in the work-group assignment, a single CUDA grid is required.

If a partition does not meet this constraint, the translated CUDA kernel needs to be launched multiple times to execute the work-group assignment. This introduces an unnecessary kernel launching overhead. Furthermore, a large number of thread blocks in a grid is a primary condition for keeping the entire GPU busy.

Another constraint is about the size balance between different work-group assignments in the partition. It is desirable to equally distribute all the work-groups in the index space across different work-group assignments.

Constraint 2. *The number of work-groups in each assignment should be well-balanced.*

Satisfying this condition makes all the available GPUs busy. For example, each assignment in the partition shown in Figure 4 (d) has a different number of work-groups. This is not desirable.

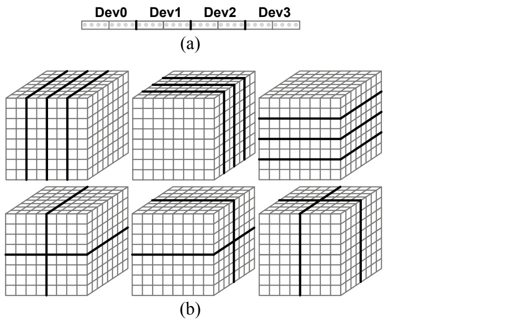


Figure 5. (a) The optimal partition with respect to Constraint 1 and Constraint 2 for one-dimensional index spaces with four GPUs. Each rectangle represents a work-group. (b) The optimal partitions with respect to Constraints 1 and Constraints 2 for three-dimensional index spaces with four GPUs. Each cube represents a work-group.

With respect to Constraint 1 and Constraint 2, the partitions in Figure 4 (a) and (b) are optimal for the two-dimensional index space. For one-dimensional index spaces, there is only one way of optimally partitioning the index space (Figure 5 (a)). For three-dimensional index spaces with four GPU devices, there are six different ways of optimally partitioning an index space as shown in Figure 5 (b).

Our OpenCL runtime has a built-in partition table. The table is indexed by a pair that consists of the dimension of the work-group index space and the number of available GPUs in the system. Each partition table entry records all optimal partitions with respect to Constraint 1 and Constraint 2. We call an optimal partition with respect to Constraint 1 and Constraint 2 as a *work-group optimal partition*.

The last constraint is about the data transfer overhead between the host memory and the GPU device memories.

Constraint 3. *The partition should guarantee a minimum amount of data transfer between the host memory and the GPU device memories.*

To find a partition that minimizes the data transfer overhead, our OpenCL runtime performs a buffer access range analysis (in the sampling run) for each candidate partition in the selected partition table entry.

3.3 Run-Time Buffer Access Range Analysis

An OpenCL kernel accesses the global memory in a compute device using a pointer (e.g., A_gpu, B_gpu, and C_gpu in Figure 1 (c)) that is passed by the host program. It typically refers to a buffer. That is, a global array in the kernel is contained in a buffer. After performing the buffer access range analysis on a buffer with a given partition, our runtime obtains the access range of the buffer by each work-group assignment in the partition. To do so, the runtime performs a sampling run on the host when the host enqueues a kernel execution command. At this point of time, the runtime has obtained all the information about the kernel arguments and the kernel index space.

If the subscript of a reference to a global array is in an affine form of the global ID, work-group ID, and local ID of the work-item that accesses the array, then the address of the array reference can be computed with the value of the pointer to the global array and an affine function of the global ID, work-group ID, and local ID of the work-item. That is, the address of the array reference is an affine function of the global ID, work-group ID, and local ID. In addition, as mentioned in Section 2, the global ID of a work-item can be computed with its work-group ID and local ID. Thus, the address of the array reference can be represented with an affine function of the work-group ID and local ID only.

For example, suppose that we have a three-dimensional index space and the address of a reference to a global array in the kernel is an affine function of global ID, work-group ID (w_x, w_y, w_z), and local ID (l_x, l_y, l_z). Then, the function can be represented only with the work-group ID and local ID:

$$\begin{aligned} f(l_x, l_y, l_z, w_x, w_y, w_z) = \quad & a_1 l_x + a_2 l_y + a_3 l_z + a_4 w_x \\ & + a_5 w_y + a_6 w_z + a_7, \end{aligned}$$

where a_i is a constant for all $i = 1, 2, ..., 7$.

The simplest way to figure out the access range of a buffer by a work-group assignment is obtaining the lower bound and upper bound of addresses accessed by each work-item in the assignment. However, performing this at run-time is a very time consuming process because a typical OpenCL kernel has a large index space that contains many work-items. Instead, we sample some representative work-items in the work-group assignment and use them to compute the assignment's buffer access range. To select the representative work-items, we use the following theorem:

Theorem 1. *Suppose that a function $f_N : [L_1, U_1] \times [L_2, U_2] \times ... \times [L_N, U_N] \to \mathbb{Z}$ is affine in N variables x_1, x_2, ..., and x_N, where $L_i, U_i \in \mathbb{Z}$ for all $i = 1, 2, ..., N$. Then the maximum and minimum of f_N occur among 2^N points $(x_1, x_2, ..., x_N)$ such that $x_i = L_i$ or $x_i = U_i$ for all $i = 1, 2, ..., N$.*

Proof. We prove this theorem using induction on N. When $N = 1$, either f_1 has the maximum at L_1 and the minimum at U_1 or *vice versa* because f_1 is a linear function. Assume that for some integer $k \geq 1$, the maximum and minimum of an affine function f_k occur among 2^k points $(x_1, x_2, ..., x_k)$ such that $x_i = L_i$ or $x_i = U_i$ for all $i = 1, 2, ..., k$. An affine function f_{k+1} can be represented with the sum of two affine function f_k and f_1,

$$\begin{aligned} f_{k+1}(x_1, x_2, ..., x_{k+1}) = \quad & a_1 x_1 + a_2 x_2 + ... + a_k x_k \\ & + a_{k+1} x_{k+1} + a_{k+2} \\ = \quad & f_k(x_1, x_2, ..., x_k) + f_1(x_{k+1}), \end{aligned}$$

where a_i is a constant for all $i = 1, 2, ..., k + 2$. Since f_k has the maximum, say max_k, and the minimum, say min_k, at some points among 2^k points $(x_1, x_2, ..., x_k)$, where $x_i = L_i$ or $x_i = U_i$ for all $i = 1, 2, ..., k$, and either f_1 has the maximum, say max_1, at L_1 and the minimum, say min_1, at U_1 or *vice versa*, f_{k+1} has the maximum $max_k + max_1$ and the minimum $min_k + min_1$ at some points among 2^{k+1} points $(x_1, x_2, ..., x_k, x_{k+1})$ such that $x_i = L_i$ or $x_i = U_i$ for all $i = 1, 2, ..., k, k + 1$. ■

For example, consider Figure 6 (a). Assume that we have a one-dimensional index space and an array reference in the kernel whose subscripts are affine functions of the global ID, work-group ID, and local ID of a work-item. To find the access range of the array reference in a work-group, it is sufficient to check two work-items located in 0 and $S_x - 1$ in the work-group, where S_x is the size of the work-group. That is, the lower bound and upper bound of the addresses accessed by the reference in the work-group can be obtained by sampling $2^1 = 2$ work-items (the two black circles in Figure 6 (a)) in the work-group. Similarly, it is sufficient to

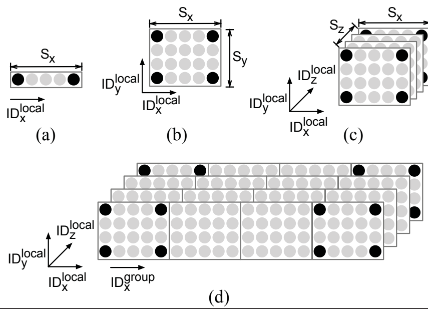


Figure 6. Representative work-items to measure the range of a buffer (array) that is accessed by a work-group assignment in the partition.

sample $2^2 = 4$ for a two-dimensional work-group (Figure 6 (b)) and $2^3 = 8$ for a three-dimensional work-group (Figure 6 (c)). An assignment in the given work-group partition typically consists of multiple work-groups as shown in Figure 6 (d). In this case, we have a three-dimensional local index space and a one-dimensional work-group index space. Thus, sampling $2^4 = 16$ work-items in the assignment is sufficient to find the access range of an affine array reference by all the work-items in the assignment.

In the worst case, it is sufficient to sample $2^6 = 64$ work-items for each work-group assignment in the partition when the assignment has a three-dimensional local index space and a three-dimensional work-group index space. The number of work-group assignments in the partition increases as the number of GPUs, say N_{device}, increases. In addition, the number of sampling is proportional to the number of work-group optimal partitions, say $N_{partition}$. Let D_{group} and D_{local} be the dimensions of the work-group index space and local index space. Then the number of work-items to be sampled by our runtime for each kernel is,

$$N_{partition} \cdot N_{device} \cdot 2^{(D_{group}+D_{local})}$$

However, to sample access ranges of a global array reference with our method, the reference needs to satisfy some more conditions with regards to the control flow. The conservative conditions that need to be simultaneously satisfied by a global array reference for sampling are as follows:

Sampling Condition

- Each index of the array reference is an affine function in global ID, work-group ID, and local ID.
- The array reference is not contained in a conditional statement.
- When the array reference is not contained in a loop, any variable used in its indices does not have more than one reaching definition.
- If the array reference is contained in well-behaved affine loops, each of its indices is an affine function in induction variables of the loops, global ID, work-group ID, and local ID.

A well-behaved loop[11] is a `for` loop in the form `for(e1; e2; e3) stmt`, in which `e1` assigns a value to an integer variable `i`, `e2` compares `i` to a loop invariant expression, `e3` increments or decrements `i` by a loop invariant expression, and `stmt` contains no assignment to `i`. We say a well-behaved loop is *affine in the global ID, work-group ID, and local ID* if the expression assigned to `i` in `e1` and the expression that is compared to `i` in `e2` are some affine functions in the global ID, work-group ID, and local ID. That is, the lower bound and upper bound of the loop index variable of a well-behaved loop are affine functions in the global ID, work-group ID, and local ID.

If the array reference is contained in well-behaved affine loops and each of its indices is an affine function in induction variables of the loops, global ID, work-group ID, and local ID, then the lower bound and upper bound of the index in the loop are also an affine function in the global ID, work-group ID, and local ID. This is because the lower bounds and upper bounds of the induction variables in the well-behaved affine loop are all an affine function in the global ID, work-group ID, and local ID. Based on Theorem 1, our runtime executes the loops with only the lower bounds and upper bounds of the loop indices.

After a sampling run, our runtime obtains buffer access ranges for each work-group assignment in all work-group optimal partitions. For an array reference that does not satisfy the sampling condition, our runtime conservatively assumes that every work-item accesses every element of the array. That is, the buffer access range of the array reference is the entire buffer. In this case, the runtime does not perform the sampling run for the array reference.

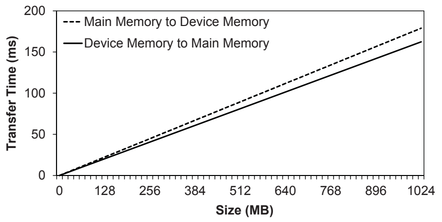


Figure 7. The data transfer time between the main memory and the GPU device memory through the PCI express bus.

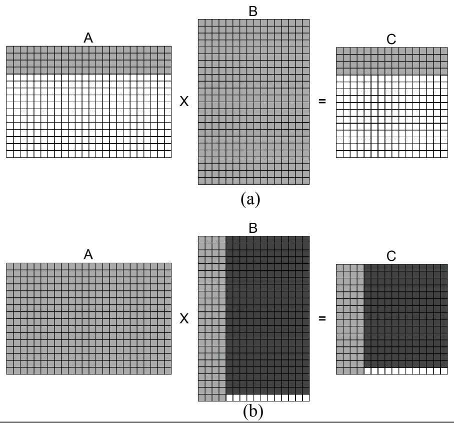


Figure 8. (a) The access range of arrays by a work-group assignment in the partition shown in Figure 4 (a) for the matrix multiplication kernel in Figure 1 (b). (b) The access range of arrays by a work-group assignment in the partition in Figure 4 (b).

3.4 Minimizing the Amount of Data Transfer

Figure 7 shows the data transfer time between the main memory (DDR3 72GB) and the GPU (NVIDIA GTX 480[13]) memory for different amounts of data transferred (from 16MB to 1024MB) through the PCI express bus. We see that the transfer time is strictly proportional to the amount of data transferred. In addition, if the OpenCL kernel accesses a buffer, then most probably all work-group assignments in a work-group optimal partition access the same buffer. This implies that the number of data transfers is most probably the same for all work-group optimal partitions. Thus, we can exclude the transfer initiation overhead of the PCI express bus from our consideration.

Since the data transfer time is strictly proportional to the amount of data transferred and the transfer is serialized by the PCI express bus, after the buffer access range analysis, the runtime computes the total amount of data transferred between the host and the devices for each work-group optimal partition that is recorded in the partition table entry. Then it chooses the partition that has the minimum amount of data transfer as an optimal partition with respect to Constraint 3.

```
void matrixMul(float *C, float *A, float *B, int WA, int WB)
{
   int i = get_global_id(0);
   int j = get_global_id(1);
   for ( int k = 0; k < WA; k += (WA-1) ) {
      GLOBAL_MEM_READ_float(&A[k + j * WA]);
      GLOBAL_MEM_READ_float(&B[i + k * WB]);
   }
   GLOBAL_MEM_WRITE_float(&C[i + j * WB]);
}
```

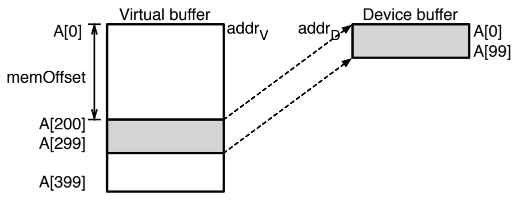
(a)

```
__global__ void matrixMul(float* C, float* A, float *B, int WA, int WB)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  int j = blockDim.y * blockIdx.y + threadIdx.y;
  float acc = 0.0f;
  for (int k = 0; k < WA; k++)
     acc  += A[k + j * WA] * B[i + k * WB];
  C[i + j * WB] = acc;
}
```

(b)

```
__global__ void matrixMul(float* C, float* A, float *B, int WA, int WB,
                                uint memOffsetC, uint memOffsetA,
                                uint memOffsetB, uint3 blockOffset)
{
   int i = blockDim.x * (blockIdx.x + blockOffset.x) + threadIdx.x;
   int j = blockDim.y * (blockIdx.y + blockOffset.y) + threadIdx.y;
   float acc = 0.0f;
   for (int k = 0; k < WA; k++)
      acc += (A-memOffsetA)[k + j * WA]
               * (B-memOffsetB)[i + k * WB];
   (C-memOffsetC)[i + j * WB] = acc;
}
```

(c)



(d)

Figure 9. (a) The sampling code generated by our OpenCL-C-to-C translator for the matrix multiplication kernel in Figure 1 (b). (b) An equivalent CUDA kernel code to the matrix multiplication kernel in Figure 1 (b) for a single GPU. (c) The CUDA kernel code generated by our OpenCL-C-to-CUDA-C translator for multiple GPUs. (d) Address translation mechanism.

Figure 8 shows the buffer access ranges of the matrix multiplication kernel in Figure 1 (b). The ranges are found by our sampling run. Figure 8 (a) shows the ranges by a work-group assignment in the partition shown in Figure 4 (a). Figure 8 (b) shows the ranges by a work-group assignment in the partition shown in Figure 4 (b). Both of the partitions in Figure 8 (a) and Figure 8 (b) are work-group optimal. However, the partition in Figure 4 (b) requires more data transfer (the gray area in Figure 4 (b)) between the host and the devices. The elements in the darker gray areas in matrices B and C in Figure 8 (b) do not actually need to be transferred to the device buffer before the kernel execution or to the main memory after the kernel execution because they are not accessed by the device at all. Our method includes these areas because a buffer (array) is a one-dimensional collection of contiguous memory elements and it computes only the lower bound and upper bound of the buffer addresses accessed by each work-group assignment. Thus, the partition in Figure 4 (a) is better than that in Figure 4 (b).

3.5 Kernel Code Transformation

There are two source-to-source translators in our OpenCL framework. One is an OpenCL-C-to-CUDA-C translator that translates the OpenCL kernel code to the CUDA kernel code. The other is an OpenCL-C-to-C translator that translates the OpenCL kernel code to the sampling code.

Figure 9 (a) shows the sampling code generated by our OpenCL-C-to-C translator for the matrix multiplication kernel in Figure 1 (b). This code will be executed on the host by our runtime. In this code, the address of each affine global array reference is wrapped with a runtime API function. For example, GLOBAL_MEM_READ_float(&A[k+j*WA]); makes the runtime record the address of A[k+j*WA]. After recording the addresses for all work-items that are sampled, the runtime obtains the lower bound and upper bound of the addresses.

To generate the sampling code, our OpenCL-C-to-C translator uses program slicing techniques[25, 27]. First, it inlines user-defined function calls in the kernel. Then it performs constant propagation, loop invariant and induction variable recognition, and reaching definitions analysis for the kernel. The translator extracts the index computation slice for each array reference and iteratively applies forward substitution to the slice until there is no more forward-substitutable variable. The translator checks if it satisfies the sampling condition mentioned before. If the array reference satisfies the sampling condition, it replaces the reference in the slice with an appropriate API function (e.g., GLOBAL_MEM_READ_float()). The address of the reference (e.g., &A[k+j*WA]) is the argument to this function. After processing index computation slices of all affine array references that satisfy the sampling condition, the translator generate the sampling code. If the generated code contains a well-behaved affine loop, then the translator transforms the loop to iterate only for the lower bound and upper bound of the loop index (e.g., from for (int k = 0; k < WA; k++) to for (int k = 0; k < WA; k += (WA - 1)) in Figure 9 (a)).

An exception to the second item of the sampling condition is the case when an array reference is contained in a sole if statement with a condition of the form (e = 0), where e is a variable that becomes get_local_id(0). This case occurs when there is a reduction computation in the kernel, and the translator recognizes this idiom.

Figure 9 (b) shows an equivalent CUDA kernel code to the matrix multiplication kernel in Figure 1 (b). The CUDA reserved variable blockDim specifies the size of the local index space in OpenCL. Similarly, blockIdx and threadIdx are equivalent to the work-group ID and local ID in OpenCL, respectively. Figure 9 (c) shows the CUDA kernel code generated by our OpenCL-C-to-CUDA-C translator for the matrix multiplication kernel in Figure 1 (b) for multiple GPUs.

Since each dimension of blockIdx always starts with 0 in CUDA, we need to add an offset (blockOffset) to each dimension of blockIdx to preserve the original location of each work-item in the index space. In Figure 9 (d), the gray area in the virtual buffer is the access range of the work-group assignment that is assigned to a GPU. This portion of the virtual buffer is copied to the device buffer of the GPU. Then, an array reference A[200] in the virtual buffer becomes a reference A[0] in the device buffer. To make a reference A[200] in the CUDA kernel access the location of A[0] on the device, we need to shift the start address of the array in the CUDA kernel by -memOffset. Thus, a reference A[i] in the original OpenCL kernel becomes (A-memOffset)[i] in the CUDA kernel. Both of blockOffset and memOffset are passed to the CUDA kernel as arguments.

3.6 Managing the Virtual Device Memory

As mentioned before, our OpenCL runtime maintains the virtual device memory that is allocated in the host main memory. The

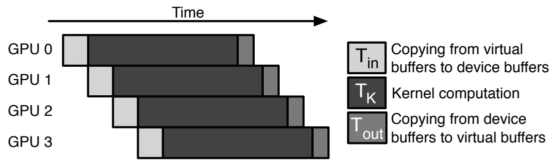

Time
GPU 0
GPU 1
GPU 2
GPU 3
T_{in} Copying from virtual buffers to device buffers
T_K Kernel computation
T_{out} Copying from device buffers to virtual buffers

Figure 10. Overlapping the kernel computation and data transfer through the PCI express bus.

runtime treats it as if it were the device memory of a single GPU device and keeps it consistent to the memories of the multiple GPUs. Each buffer object is treated as a virtual buffer allocated in the virtual device memory. The virtual device memory is organized as a list of buffers. At the beginning, the list is empty.

When the host allocates a buffer object by invoking `clCreateBuffer()`, our virtual GPU device thread allocates the requested buffer in the virtual device memory. That is, it allocates a buffer in the host main memory and inserts the buffer into the list. When the host requests copying an array in the host program to a buffer object by invoking `clEnqueueWriteBuffer()`, it copies the array in the corresponding virtual buffer.

When the host enqueues a kernel execution command by invoking `clEnqueueNDRangeKernel()`, the runtime runs a sampling code to obtain virtual buffer access ranges and distributes the kernel workload. After obtaining the work-group optimal partition, the runtime assigns each work-group assignment in the partition to a GPU device. Using the access range information of each virtual buffer by the work-group assignment, the runtime allocates a device buffer, whose size is exactly the same as that of the access range, in the associated GPU device memory. Then, the runtime copies the virtual buffer area in the access range to the device buffer.

Our OpenCL runtime makes the virtual buffer consistent to the device buffers in the following two cases:

- When the host requests copying a buffer object to an array in the host program by invoking `clEnqueueReadBuffer()`.
- When a new kernel is launched and a GPU requires the data that has been modified by another GPU in the prior kernel execution.

The runtime creates in the main memory a twin (copy) of each shared and modified region of the virtual buffers between different GPUs. This sharing information is obtained from the buffer access range information. When a GPU has updated a device buffer and if the device buffer overlaps with one of the shared and modified regions, the runtime copies the device buffer to a temporary buffer space allocated in the main memory. The runtime updates the original virtual buffer with the result of comparison between the twin of the shared/modified region and the region in the temporary buffer. The non-shared regions in the temporary buffer are copied to the virtual buffer without any comparison. If the device buffer does not overlap with one of the shared/modified regions, the updated device buffer is directly copied to the corresponding area of the virtual buffer.

Sharing of virtual buffer access ranges between different GPU devices (i.e., between different work-group assignments in the partition) is mostly due to false sharing. Even though their access ranges overlap with each other, one GPU device typically does not modify the location that is modified by another device in the shared region.

3.7 Overlapping Data Transfer and Kernel Computation

Our method naturally overlaps data transfer and kernel computation as shown in Figure 10. Suppose that the work-group assignments and buffer access ranges are symmetric between N GPUs. Let T_K, T_{in}, and T_{out} be the kernel execution time on each GPU, the time taken to copy the virtual buffers to the device buffers before the kernel execution, and the time taken to copy the device buffers to the virtual buffers after kernel execution on each GPU, respectively. Then, the overlap is maximized when $(N-1) \cdot T_{in} \leq T_K$ and $T_{out} \leq T_{in}$. The total elapsed time will be $N \cdot T_{in} + T_K + T_{out}$.

4. Evaluation

In this section, we present the evaluation methodology and result for our OpenCL framework.

4.1 Methodology

Target machine. We evaluate the performance of our OpenCL framework using a system that consists of two Intel Xeon X5680 hexa-core CPUs, eight NVIDIA GeForce GTX 480 GPUs[13], and 72GB DDR3 main memory. It runs Ubuntu Linux 9.10. The CPU supports simultaneous multithreading (SMT) that enables two threads per core. The GPU has 1.6GB device memory and 15 streaming multiprocessors. Each streaming multiprocessor has 32 CUDA cores, and therefore each GPU has 480 CUDA cores. The GPUs communicate with the CPUs via a PCI-E Gen. 2 x16 bus that uses point-to-point serial links. The PCI-E Gen. 2 x16 bus has a data transfer rate of 8GBps (full duplex).

Benchmark applications. We run 11 OpenCL applications that are from various sources: AMD[1], PARSEC[3], CUDA Zone[16, 21], Parboil[24], NAS[12], and NVIDIA[17]. The applications from PARSEC, CUDA Zone, Parboil, and NAS are translated into OpenCL applications manually. Their details are described in Table 2. Column A shows the global memory size that is required to run each application, column B shows the number of kernels, column C shows the size of the work-group index space for each kernel in the application (separated by comma for each kernel, separated by a colon for the size in each dimension in the multi-dimensional work-group index space), column D shows the number of executions for each kernel (separated by comma), column E shows % execution time taken for the kernel to run on a single GPU in the total execution time, column F shows if inter-GPU communication through the main memory is required or not to run the application, column G indicates that if kernels in the application contains array references that do not satisfy the sampling condition as described in Section 3.3, column H shows if the references identified in column G are non-affine form of global ID, work-group ID, and local ID, and finally column I shows if the references identified in column G are contained in a conditional branch. Especially, all array references in Correlator and TPACF do not satisfy the sampling condition. This implies that we do not perform sampling runs for kernels in Correlator and TPACF and assume that each work-group assignment in the partition accesses the entire array.

Runtime and source-to-source translator. We have implemented our OpenCL runtime for multiple GPUs. We use NVIDIA CUDA Toolkit 3.1[17] to implement the runtime. We have implemented our OpenCL-C-to-C and OpenCL-C-to-CUDA-C translators by modifying **clang** that is a C front-end for the LLVM compiler infrastructure[9].

4.2 Results

Figure 11 shows the speedup numbers of the ideal case and our OpenCL framework over a single GPU for each application. We vary the number of GPUs (2, 4, and 8). The ideal speedup is obtained with the Amdahl's law[2]. We treat the OpenCL kernel execution time on a single GPU as the parallel fraction. The serial fraction is the time spent on a CPU by the application.

All applications but Correlator, EigenValue, MRI-Q, and RPES show a speedup that is close to the ideal speedup. Correlator has a single kernel in which each work-item has a different workload. Thus, for this kernel, an equal distribution of work-groups does not guarantee an equal distribution of workload in the kernel. Figure 12 shows the normalized average kernel execution time per GPU.

Application	Source	Description	Input	A	B	C	D	E	F	G	H	I
BinomialOption	AMD	Binomial option pricing	65520 samples, 100 iters.	2MB	1	65520	1	99.6%				
Blackscholes	PARSEC	Black-Scholes PDE	16773120 options, 1000 iters.	447.9MB	1	65520	1	92.7%				
Correlator	CUDA Zone	Radio astronomy signal correlator	512 stations, 128 channels	1282MB	1	512	1	83.3%		X	X	
CP	Parboil	Coulombic potential	8192x8192, 100000 atoms	256MB	1	512:512	25	99.7%				
EigenValue	AMD	Eigenvalue computation	65536x65536, 10 iters.	1.7MB	2	256, 256	6, 6	99.9%	X	X	X	
EP	NAS	Embarrassingly parallel	Class D	99MB	1	32768	1	99.9%		X		X
MatrixMul	NVIDIA	Matrix multiplication	8192x8192	768MB	1	512:512	1	68.7%				
MRI-Q	Parboil	Magnetic resonance imaging Q	Large	5MB	2	4, 1024	1, 2	50.3%		X		X
Nbody	NVIDIA	N-Body simulation	524288 bodies, 10 iters.	32MB	1	2048:1	1	99.5%		X		X
RPES	Parboil	Rys polynomial equation solver	Default	79.4MB	2	65528, 65528	54, 17	64.2%	X	X	X	X
TPACF	Parboil	Two-point angular correlation function	Default	9.5MB	1	201	1	63.8%		X		X

A: global memory size, B: # of kernels, C: the size of work-group index space, D: # of kernel executions, E: % execution time of kernels on a single GPU, F: inter-GPU communication, G: contains array references that do not satisfy the sampling condition, H: contains non-affine array references, I: contains array references in a conditional branch

Table 2. Applications used for the evaluation.

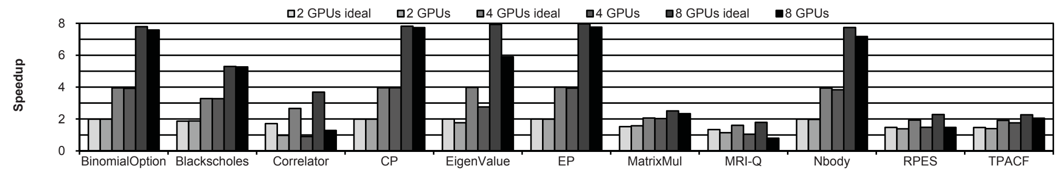


Figure 11. Speedup over a single GPU.

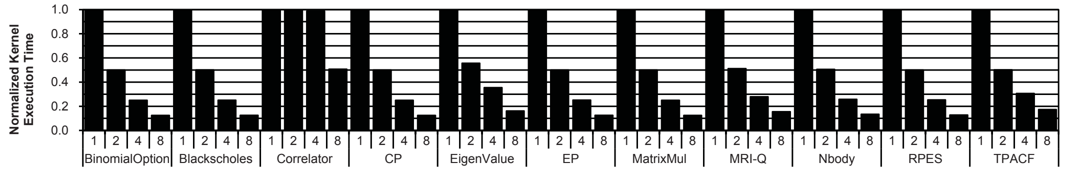


Figure 12. Average kernel execution time per GPU.

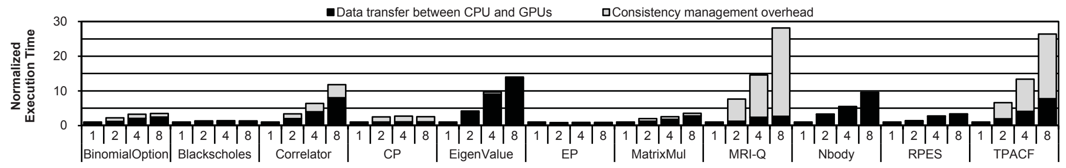


Figure 13. Data transfer time between the CPU and GPUs and consistency management overhead by the runtime.

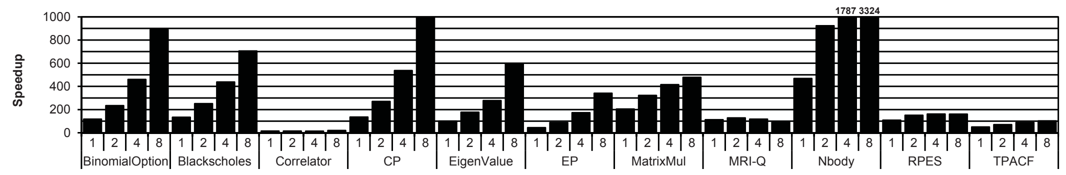


Figure 14. Speedup over a single CPU.

The average kernel execution time per GPU decreases linearly for all applications but Correlator. This accounts for the uneven distribution of the kernel workload in Correlator.

Another reason for the poor performance of Correlator is a high degree of read sharing and write sharing between buffer access ranges of different work-group assignments. Figure 13 shows the normalized data transfer time between the CPU and GPUs and consistency management overhead by the runtime for different numbers of GPUs. Data transfers are serialized by the PCI express bus. The consistency management overhead includes the overhead introduced by virtual buffer management (by the virtual GPU device thread), such as twin creation, CPU-to-CPU memory copy, and comparison for updates. The time is normalized to that of a single GPU. Since the data transfer time with multiple GPUs increases when the number of GPUs increases, a high degree of sharing exists in buffer access ranges between different work-group assignments in Correlator. Moreover, increasing the number of GPUs also increases the consistency management overhead. This also means that Correlator has a high degree of overlap (i.e., false sharing) in buffer access ranges between different work-group assignments. There is no true sharing in Correlator.

EigenValue has large data transfer time between the CPU and GPUs for more than one GPU in Figure 13. It has two kernels that are consecutively executed. The second kernel reads data that has been written by the first. In addition, a GPU executing the second kernel reads data that has been written by another GPU in the first kernel (inter-GPU communication shown in Table 2). This accounts for the poor performance of EigenValue when compared with the ideal case. The case of RPES is similar to that of EigenValue. However, as the number of GPUs increases, RPES has a relatively small increment in the data transfer time between the CPU and GPUs in Figure 13. Since its kernel execution time is much smaller than its data transfer time, the data transfer time dominates the total execution time of RPES. TPACF has a significant amount of the consistency management overhead and data transfer time with more than one GPU. Unlike the case of RPES, the kernel execution time of TPACF is much bigger than the sum of data transfer time and consistency management time. Thus, the reduced kernel execution time due to multiple GPUs dominates TPACF's total execution time.

The poor performance of MRI-Q when compared with the ideal case is due to the high degree of false sharing between buffer access ranges from different work-group assignments. In Figure 13, it has a large amount of consistency management overhead due to the false sharing for more than one GPU. When the number of GPUs increases, the amount of overhead also increases.

Nbody has a relatively poor speedup with 8 GPUs. The reason is that the number of work-groups in Nbody is not big enough to make all streaming multiprocessors in the 8 GPUs busy. It also has data sharing between two consecutive execution instances of a single kernel (due to a loop that encloses the kernel) for more than one GPU. This results in a large amount of data transfer between the CPU and the GPUs in Figure 13. Not enough number of work-groups in the work-group index space (Table 2) also accounts for the poor performance of Correlator, EigenValue, MRI-Q, and TPACF with 8 GPUs.

As shown in Table 2, seven applications contain buffer references that do not satisfy the sampling condition. Our runtime conservatively assumes that the access range of these references is the entire buffer. The overheads of data transfer and consistent management in Correlator, RPES, and TPACF mainly come from the buffer references that do not satisfy the sampling condition.

Sampling overhead. The sampling overhead of our OpenCL runtime is negligible. The sampling overhead often increases as the number of GPUs, the dimension of the kernel index space, and the number of kernel executions increase. RPES has the biggest sampling overhead among the 11 benchmark applications. The work-group index space in RPES is two-dimensional and the number of total kernel executions is 71 as shown in Table 2. The sampling overhead in RPES with 8 GPUs accounts for 0.48% of its total execution time.

For your reference, Figure 14 shows the speedup of each application over a single CPU for different number of GPUs. The sequential CPU version of each application is obtained from the same source in Table 2. For MatrixMul, we optimize the CPU version using tiling; this is because the OpenCL version of MatrixMul exploits the same tiling optimization.

Figure 15. Speedup of applications that have a bigger data size than the memory size of a single GPU. The speedup is obtained with 8 GPUs over a single CPU.

Since our OpenCL runtime manages multiple device memories available in multiple GPUs through the virtual device memory, OpenCL applications with our OpenCL framework can process data whose size is much bigger than the size of the available memory in a single GPU. We choose some applications whose input is controllable to increase the size of data used. They are Blackscholes and MatrixMul. Figure 15 shows the speedups of Blackscholes and MatrixMul with 8 GPUs over a single CPU. Blackscholes processes 268,369,920 options and this requires 7GB of device memory. MatrixMul performs a matrix multiplication of two 16,384 x 16,384 floating point matrices. This requires a total of 3GB device memory. These two applications cannot be executed with a single GPU because the data size is bigger than the device memory size. For this reason, we obtain the speedup over a single CPU instead of a single GPU.

5. Related Work

Even though GPGPU systems with multiple GPUs are widening their user base, there have not been many proposals for exploiting those multiple GPUs. Most of the previous proposals require manual coding to exploit them. Unlike previous proposals, our approach is fully automatic and transparent to the user. With our approach, the unmodified OpenCL source code for a single GPU system runs directly on a multiple GPU system. To the best of our knowledge, our framework is the first OpenCL framework that automatically exploits multiple GPUs available in the system.

Schaa and Kaeli[22] categorize the GPU systems into distributed GPUs (a networked group of distributed systems each containing a single GPU) and shared-system GPUs (a single system containing multiple GPUs). They define performance estimation models to predict the performance of applications on these two categories. They show that their modeling framework works effectively on multiple GPUs. Quintana-Ortí *et al.*[20] propose a programming model and a runtime that solves dense linear algebra problems, such as matrix multiplication and Cholesky factorization, for multiple accelerators. Their approach is different from ours in that their runtime requires the detailed knowledge of an application before executing it, and the runtime schedules the tasks using a pre-built-in scheduling policy for the application. Phillips *et al.*[19] build a GPU-accelerated cluster for a molecular dynamics simulation of large bio-molecular systems and define API functions for programming the cluster. With 15 nodes (each node contains four GPUs), they achieve performance that nearly matches the performance of 330 CPU cores. Strengert *et al.*[23] introduce CUDASA, an extension to CUDA. CUDASA presents additional language elements that enable workload distribution capability for multiple GPUs. However, manual CUDASA coding is required to exploit multiple GPUs.

Data transfer between the CPU and GPUs can be a performance bottleneck for GPU applications[5, 8, 14, 29]. Thus, minimizing the data transfer overhead is a challenging problem. One solution is overlapping the data transfer and GPU computation. Gelado *et al.*[5] present a hardware support that overlaps the data transfer and GPU computation with a single GPU. It allows the CPU to transfer data for the next invocation of a CUDA kernel while the GPU is executing the current instance of the same kernel. Our framework enables overlapping the data transfer and GPU kernel computation for multiple GPUs without any hardware support.

There are some proposals for OpenCL frameworks[6, 10]. Gummaraju *et al.*[6] propose an OpenCL framework named Twin Peaks. It handles both CPUs and GPUs. Twin Peaks executes SPMD style OpenCL kernels on a CPU core by using routines that save/restore work-item contexts. They use their own light-weight `setjmp()` and `longjmp()` system calls for low overhead context switch. Lee *et al.*[10] propose an OpenCL framework for heterogeneous multicores with local memory, such as Cell BE processors. Their OpenCL framework has two optimization techniques for OpenCL programs running on the Cell BE architecture, namely work-item coalescing and preload-poststore buffering. Work-item coalescing is a technique that significantly reduces context switching overhead of executing an OpenCL kernel on multiple SPEs in the Cell BE processor. Preload-poststore buffering is a technique that hides memory latencies introduced by data transfers between the PPE and an SPE to execute the kernel.

6. Conclusions

We introduce the design and implementation of an OpenCL framework that provides an illusion of a single compute device to the programmer for multiple GPUs available in a GPGPU system. Providing a single virtual compute device image to the user makes an OpenCL application for a single GPU device portable to the system that has multiple GPUs. It also makes the application exploit full computing power of the multiple GPUs and all available GPU memories in the system.

Our OpenCL framework distributes the workload in an OpenCL kernel across multiple GPUs at run time when the information about the kernel work-group index space is known. To distribute the kernel workload across multiple GPUs efficiently, our framework 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. For non-affine array references, the runtime does not perform the sampling run and the access range becomes the entire buffer. Using this information, the runtime distributes the kernel work-group index space according to the three optimality constraints: expressibility in a single grid of CUDA thread blocks, load balancing between multiple GPUs, and the minimum amount of data transfer between the CPU and GPUs. To achieve a single compute device image, our runtime maintains the virtual device memory that is allocated in the main memory. The runtime treats the memory as if it were the memory of a single GPU and keeps it consistent to the memories of the multiple GPUs.

We have implemented our OpenCL runtime and compilers. Our OpenCL-C-to-C translator generates the sampling code from the OpenCL kernel code and OpenCL-C-to-CUDA-C translator generates the CUDA kernel code for the distributed OpenCL kernel. The experimental results with 11 OpenCL benchmark applications and a GPGPU system with 8 GPUs indicate that our approach is practical and promising.

References

[1] *ATI Stream Software Development Ket (SDK) v2.1*. AMD, 2010. `http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx`.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, October 2008.

[4] F. Darema. The SPMD Model: Past, Present and Future. *Lecture Notes in Computer Science*, 2131(1):1–1, January 2001.

[5] I. Gelado, J. H. Kelm, S. Ryoo, S. S. Lumetta, N. Navarro, and W.-m. W. Hwu. CUBA: an architecture for efficient CPU/co-processor data communication. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 299–308. ACM, June 2008.

[6] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 205–216. ACM, 2010.

[7] Khronos OpenCL Working Group. *The OpenCL Specification Version 1.0*. Khronos Group, 2009. `http://www.khronos.org/opencl`.

[8] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010. ISBN 0123814723, 9780123814722.

[9] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, pages 75–86, Washington, DC, USA, March 2004. IEEE Computer Society.

[10] J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, and J.-D. Choi. An OpenCL framework for heterogeneous multicores with local memory. In *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 193–204. ACM, 2010.

[11] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4.

[12] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[13] *NVIDIA Fermi Compute Architecture White Paper*. NVIDIA, 2009. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[14] *NVIDIA CUDA C Best Practices Guide 3.1*. NVIDIA, May 2010.

[15] *NVIDIA CUDA C Programming Guide 3.1.1*. NVIDIA, July 2010.

[16] *NVIDIA CUDA Zone*. NVIDIA, July 2010. `http://www.nvidia.com/object/cuda_home_new.html`.

[17] *NVIDIA GPU Computing Software Development Kit*. NVIDIA, June 2010. `http://developer.nvidia.com/object/cuda_3_1_downloads.html`.

[18] *Tesla M2050/M2070 GPU Computing Module*. NVIDIA, 2010. `http://www.nvidia.com/object/product_tesla_M2050_M2070_us.html`.

[19] J. C. Phillips, J. E. Stone, and K. Schulten. Adapting a message-driven parallel application to GPU-accelerated clusters. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–9, Piscataway, NJ, USA, November 2008. IEEE Press.

[20] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 121–130. ACM, 2009.

[21] J. W. Romein, P. C. Broekema, J. D. Mol, and R. V. van Nieuwpoort. The LOFAR correlator: implementation and performance analysis. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 169–178. ACM, 2010.

[22] D. Schaa and D. Kaeli. Exploring the multiple-GPU design space. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, May 2009.

[23] M. Strengert, C. Müler, C. Dachsbacher, and T. Ertl. CUDASA: Compute Unified Device and Systems Architecture. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV08)*, pages 49–56. Eurographics Association, April 2008.

[24] The IMPACT Research Group. Parboil Benchmark suite. `http://impact.crhc.illinois.edu/parboil.php`, 2009.

[25] F. Tip. A Survey of Program Slicing Techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[26] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[27] M. Weiser. Program Slicing. In *ICSE '81: Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[28] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu. Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing. In *IEEE Cluster '10: Proceedings of IEEE International Conference on Cluster Computing*, pages 19–28, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[29] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 86–97. ACM, June 2010.