DistCL: A Framework for the Distributed Execution of OpenCL Kernels

Tahir Diop[†], Steven Gurfinkel[†], Jason Anderson, Natalie Enright Jerger
{*tahir.diop, steven.gurfinkel*}*@mail.utoronto.ca*, {*janders, enright*}*@eecg.toronto.edu*
Department of Electrical and Computer Engineering, University of Toronto, Ontario, Canada

***Abstract*—GPUs are used to speed up many scientific computations; however, to use several networked GPUs concurrently, the programmer must explicitly partition work and transmit data between devices. We propose DistCL, a novel framework that distributes the execution of OpenCL kernels across a GPU cluster. DistCL makes multiple distributed compute devices appear to be a single compute device. DistCL abstracts and manages many of the challenges associated with distributing a kernel across multiple devices including: (1) partitioning work into smaller parts, (2) scheduling these parts across the network, (3) partitioning memory so that each part of memory is written to by at most one device, and (4) tracking and transferring these parts of memory. Converting an OpenCL application to DistCL is straightforward and requires little programmer effort. This makes it a powerful and valuable tool for exploring the distributed execution of OpenCL kernels. We compare DistCL to SnuCL, which also facilitates the distribution of OpenCL kernels. We also give some insights: distributed tasks favor more compute bound problems and favour large contiguous memory accesses. DistCL achieves a maximum speedup of 29.1 and average speedups of 7.3 when distributing kernels among 32 peers over an Infiniband cluster.**

I. Introduction

Recently, there has been significant interest in using GPUs for general purpose and high performance computing. Significant speedups have been demonstrated when porting applications to a GPU [1] [2]; however, additional speedups are still possible beyond the computational capabilities afforded by a single GPU. Therefore, it is no surprise that modern computing clusters are starting to include multiple GPUs [3]. However, distributing a data-parallel task across a cluster still involves the following steps:

1) partitioning work into smaller parts,
2) scheduling these parts across the network,
3) partitioning memory so that each part of memory is written to by at most one device, and
4) tracking and transferring these parts of memory.

This paper introduces DistCL, a framework for the distribution of OpenCL kernels across a cluster. To simplify this task, DistCL takes advantage of three insights:

1) OpenCL tasks (called *kernels*) contain threads (called *work-items*) that are organized into small groups (called *work-groups*). Work-items from different work-groups cannot communicate during a kernel invocation. Therefore, work-groups only require that the memory they read be up-to-date as of the beginning of the kernel invocation. So, DistCL must know what memory a work-group reads, to ensure that the memory is up-to-date on the device that runs the work-group. DistCL must also know what memory each work-group writes, so that future reads can be satisfied. But no intra-kernel synchronization is required.

2) Most OpenCL kernels make only data-independent memory accesses; the addresses they access can be predicted using only the immediate values they are passed and the geometry they are invoked with. This means that their accesses can be efficiently determined before they run. DistCL requires that kernel writes be data-independent.

3) Kernel memory accesses are often contiguous. This is because contiguous accesses fully harness the wide memory buses of GPUs [2]. DistCL does not require contiguous accesses for correctness, but they improve distributed performance because contiguous accesses made in the same work-group can be treated like a single, large access when tracking writes and transferring data between devices.

In OpenCL (and DistCL) memory is divided into large device-resident arrays called *buffers*. DistCL introduces the concept of *meta-functions*: simple functions that describe the memory access patterns of an OpenCL kernel. Meta-functions are programmer-written kernel-specific functions that relate a range of work-groups to the parts of a buffer that those work-groups will access. When a meta-function is passed a range of work-groups and a buffer to consider, it divides the buffer into intervals, marking each interval either as *accessed* or *not accessed* by the work-groups. DistCL takes advantage of kernels with sequential access patterns, which have fewer (larger) intervals, because it can satisfy their memory accesses with fewer I/O operations. By dividing up buffers, meta-functions allow DistCL to distribute an *unmodified* kernel across a cluster. To our knowledge, DistCL is the first framework to do so.

In addition to describing DistCL, this paper evaluates the effectiveness of kernel distribution across a cluster based on the kernels' memory access patterns and their compute-to-transfer ratio. It also examines how the performance of various OpenCL and network operations affect the distribution of kernels.

†Authors contributed equally and are listed alphabetically.

This paper makes the following primary contributions:

- The introduction of DistCL and its concept of meta-functions, which allow for the distribution of unmodified kernels.
- An evaluation of how the properties of different kernels affect their performance when distributed.

This paper first describes the OpenCL execution model and why it is a good candidate for distributed execution (Section II). Then, it describes DistCL using vector addition as an example, in particular looking at how DistCL handles each step involved with distribution (Section III). Focus then shifts to analysis; the benchmarks are introduced and grouped into three categories: linear runtime benchmarks, compute intensive benchmarks, and benchmarks that involve inter-node communication (Section IV). Results for these benchmarks are presented (Section V). A comparison with SnuCL [4] is also provided. Finally, conclusions are drawn about how the properties of an OpenCL kernel and cluster hardware impact distributed performance (Section VII).

II. OpenCL

OpenCL is a popular framework for programming multi-core processors such as CPUs, GPUs, and other processors in heterogeneous computing systems. Conceptually, an OpenCL program consists of three main components: a *host* - a regular processing environment which runs a *host program*; an *OpenCL platform* which exposes a standardized interface to the host program; and a set of *compute devices*, the accelerators attached to the host, which can be programmed using OpenCL. The host program can create an *OpenCL context* where compute devices within the same context can share OpenCL objects such as data buffers, programs and kernels. Using this OpenCL context, the host program orchestrates the execution of OpenCL kernels on compute devices. When an OpenCL kernel executes on a device, it does so using a special programming model that was developed with massive parallelism in mind.

A. Kernel Programming Model

The threads of an OpenCL kernel invocation execute in a theoretical grid called an *NDRange*. The NDRange can be one-dimensional (linear), two-dimensional (rectangular), or three-dimensional (rectangular prism-shaped). An n-dimensional NDRange is specified by n positive integers, each being the NDRange's size in one dimension. The size and dimensionality of the NDRange is decided by the host program. At the finest granularity, this NDRange contains unit-sized *work-items*. These work-items are the threads of an OpenCL kernel invocation and each run their own instance of the kernel function. Each work-item has a unique n-dimensional *global ID* which are its coordinates in the NDRange. Similarly, the size of the NDRange is the invocation's *global size*.

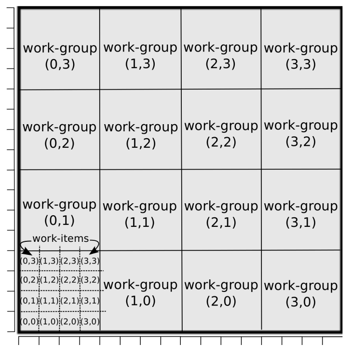

Figure 1. A 2D NDRange with a total of $4 \times 4 = 16$ work-groups and $16 \times 16 = 256$ work-items.

The NDRange is also divided up into equally-sized n-dimensional regions called *work-groups*. Each work-item belongs to exactly one work-group, and has an n-dimensional *local ID* based on its position in the work-group. Similarly, the size of the work-groups is the invocation's *local size*. OpenCL devices schedule work at the work-group granularity. Therefore, the work-items contained within a work-group are all guaranteed to be executing at the same time and on the same core (*compute unit*) of the target compute device. However, there are no guarantees on the scheduling order of work-groups. Work-items within the same work-group are guaranteed to see each other's memory accesses, but work-items in different work-groups are not. Fig. 1 shows an example two-dimensional NDRange. The NDRange consists of 16 work-groups where each work-group contains 16 work-items for a total of 256 work-items.

III. DistCL

DistCL executes on a cluster of networked computers. OpenCL host programs use DistCL by creating one context with one command queue for one device. This device represents the aggregate of all the devices in the cluster. When a program is run with DistCL, identical processes are launched on every node. When the OpenCL context is created, one of those nodes becomes the *master*. The master is the only node that is allowed to continue executing the host program. All other nodes, called *peers*, enter an event loop that services requests from the master. Nodes communicate in two ways: messages to and from the master, and raw data transfers that can happen between any pair of nodes.

To run a kernel, DistCL divides its NDRange into smaller grids called *subranges*. Kernel execution gets distributed because these subranges run on different peers. DistCL must know what memory a subrange will access in order to distribute the kernel correctly. This knowledge is provided to DistCL with *meta-functions*. Meta-functions are programmer-written, kernel-specific callbacks that DistCL uses to determine what memory a subrange will access. DistCL uses meta-functions to divide buffers into arbitrarily-sized *intervals*. Each interval of a buffer is either accessed or

```
__kernel void vector(__global int *a, __global int *b,
     __global int *out)
{
    int i = get_global_id(0);
    out[i] = a[i] + b[i];
}
```

Listing 1. OpenCL kernel for vector addition.

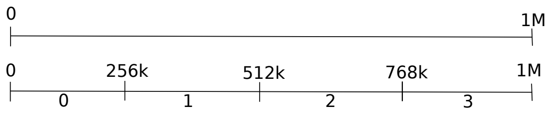

Figure 2. Vector's 1-dimensional NDRange is partitioned into 4 subranges.

not. DistCL stores the intervals calculated by meta-functions in objects called *access-sets*. Once all the access-sets have been calculated, DistCL can initiate the necessary transfers needed to allow the peers to run the subranges they have been assigned. Recall the important distinction between *subranges* which contain threads and *intervals* which contain data. The remainder of this section describes the execution process in more detail, illustrating each step with a vector addition example, whose kernel source code is given in Listing 1.

A. Partitioning

Partitioning divides the NDRange of a kernel execution into smaller grids called *subranges*. DistCL never fragments work-groups, as that would violate OpenCL's execution model and could lead to incorrect kernel execution. For linear (1D) NDRanges, if the number of work-groups is a multiple of the number of peers, each subrange will be equal in size. Otherwise, some subranges will be one work-group larger than others. DistCL partitions a multidimensional NDRange along its highest dimension first, in the same way it would partition a linear NDRange. If the subrange count is less than the peer count, DistCL will continue to partition lower dimensions.

Multidimensional arrays are often organized in row-major order, so highest-dimension-first partitioning frequently results in subranges accessing contiguous regions of memory. Transferring fragmented regions of memory requires multiple I/O operations to avoid transferring unnecessary regions, whereas large contiguous regions can be sent all at once.

Our vector addition example has a one-dimensional NDRange. Assume it runs with $1\text{M} = 2^{20}$ work-items on a cluster with 4 peers. Assuming 1 subrange per peer, the NDRange will be partitioned into 4 subranges, each with a size of 256k work-items, as shown in Figure 2.

B. Dependencies

The host program allocates OpenCL buffers and can read from or write to them through OpenCL function calls. Kernels are passed these buffers when they are invoked. For example, the three parameters, `a`, `b`, and `out` in Listing 1 are buffers.

DistCL must know what parts of each buffer a subrange will access in order to create the illusion of many compute devices with separate memories sharing a single memory. The set of addresses in a buffer that a subrange reads and writes are called its *read-set* and *write-set*, respectively. DistCL represents these *access-set*s with concrete data-structures and calculates them using meta-functions. Access-sets are calculated every kernel invocation, for every subrange-buffer combination, because the access patterns of a subrange depend on the invocation's parameters, partitioning, and NDRange. In our vector addition example with 4 subranges and 3 buffers, 24 access-sets will be calculated: 12 read-sets and 12 write sets.

An access-set is a list of intervals within a buffer. DistCL represents addresses in buffers as offsets from the beginning of the buffer; thus an interval is represented with a low and high offset into the buffer. These intervals are half open; low offsets are part of the intervals, but high offsets are not.

For instance, subrange 1 in Figure 2 contains global IDs from the interval [256k, 512k). As seen in Listing 1, each work-item produces a 4-byte (`sizeof (int)`) integer, so subrange 1 produces the data for interval [1 MB, 2 MB) of `out`. Subrange 1 will also read the same 1 MB region from buffers `a` and `b` to produce this data. The intervals [0 MB, 1 MB) and [2 MB, 4 MB) of `a`, `b` and `out` are not accessed by subrange 1.

1) Calculating Dependencies: To determine the access-sets of a subrange, DistCL uses programmer-written, kernel-specific meta-functions. Each kernel has a read meta-function to calculate read-sets and a write meta-function to calculate write-sets.

DistCL passes meta-functions information regarding the kernel invocation's geometry. This includes the invocation's global size (`global` in Listing 2), the current subrange's size (`subrange`), and the local size (`local`). DistCL also passes the immediate parameters of the kernel (`params`) to the meta-function. The subrange being considered is indicated by its starting offset in the NDRange (`subrange_offset`) and the buffer being considered is indicated by its zero-indexed position in the kernel's parameter list (`param_num`).

DistCL builds access-sets one interval at a time, progressing through the intervals in order, from the beginning of the buffer to the end. Each call to the meta-function generates a new interval. If and only if the meta-function indicates that this interval is accessed, DistCL includes it in the access-set.

To call a meta-function, DistCL passes the low offset of the current interval through `start` and the meta-function sets `next_start` to its end. The meta-function's return value specifies whether the interval is accessed. Initially setting `start` to zero, DistCL advances through the buffer by setting the `start` of subsequent calls to the previous value of `next_start`. When the meta-function sets `next_start` to the size of the buffer, the buffer has been

```
1  int is_buffer_range_read_vector(
2      const void **params, const size_t *global,
3      const size_t *subrange, const size_t *local,
4      const size_t *subrange_offset, unsigned int param_num,
5      size_t start, size_t *next_start)
6  {
7      int ret = 0;
8      *next_start = sizeof (int) * global[0];
9      if (param_num != 2) {
10         start /= sizeof (int);
11         ret = require_region(1, global, subrange_offset,
               subrange, start, next_start);
12         *next_start *= sizeof (int);
13     }
14     return ret;
15 }
```

Listing 2. Read meta-function.

```
1  int require_region(int dim, const size_t *total_size,
       const size_t *required_start, const size_t
       *required_size, size_t start, size_t *next_start);
```

Listing 3. require_region helper function.

fully explored and the access-set is complete.

2) Rectangular Regions: Many OpenCL kernels structure multidimensional arrays into linear buffers using row-major order. When these kernels run, their subranges typically access one or more linear, rectangular, or prism-shaped areas of the array. Though these areas are contiguous in multidimensional space, they are typically made up of many disjoint intervals in the linear buffer. Recognizing this, DistCL has a helper function, called `require_region`, that meta-functions can use to identify which linear intervals of a buffer constitute any such area.

`require_region`, whose prototype is shown in Listing 3, operates over a hypothetical `dim`-dimensional grid. Typically, each element of this grid represents an element in a DistCL buffer. The size of this grid in each dimension is specified by the `dim`-element array `total_size`. `require_region` considers a rectangular region of that grid whose size and offset into the grid are specified by the `dim`-element arrays `required_size` and `required_start`, respectively. Given this, `require_region` calculates the linear intervals that correspond to that region if the elements of this `dim`-dimensional grid were arranged linearly, in row-major-order. Because there may be many such intervals, the return value, `start` parameter and `next_start` parameter of `require_region` work the same way as in a meta-function, allowing the caller to move linearly through the intervals, one at a time. If a kernel does not access memory in rectangular regions, it does not have to use the helper function.

Even though vector has a one-dimensional NDRange, `require_region` is still used for its read meta-function in Listing 2. This is because `require_region` not only identifies the interval that will be used, but also identifies the intervals on either side that will not be used. `require_region` is passed `global` as the hypothetical grid's size, making each grid element correspond to an integer, the datatype changed by a single work-item. Therefore, lines 10 and 12 of Listing 2 translate between elements and bytes, which differ by a factor of `sizeof (int)`.

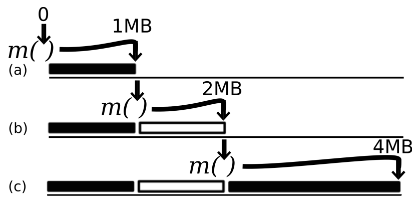

Figure 3. The read meta-function is called for buffer a in subrange 1 of vector.

Figure 3 shows what actually happens when the meta-function is called on buffer `a` for subrange 1. In Figure 3a, the first time the meta-function is called, DistCL passes in 0 as the start of the interval and the meta-function calculates that the current interval is not in the read set, and that the next interval starts at an offset of 1MB. Next, in Figure 3b, DistCL passes in 1MB as the start of the interval. The meta-function calculates that this interval is in the read-set and that the next interval starts at 2MB. Finally, in Figure 3c, DistCL passes in 2MB as the start of the interval. The meta-function calculates that this interval is not in the read-set and that it extends to the end of the buffer which has a size of 4 MB.

C. Scheduling Work

The scheduler is responsible for deciding when to run subranges and on which peer to run them. The scheduler runs on the master and broadcasts messages to the peers when it assigns work. DistCL uses a simple scheme for determining where to run subranges. If the number of subranges equals the number of peers, each peer gets one subrange; however, if the number of subranges is fewer, some peers are never assigned work.

D. Transferring Buffers

When DistCL executes a kernel, the data produced by the kernel is distributed across the peers in the cluster. The way this data is distributed depends on how the kernel was partitioned into subranges, how these subranges were scheduled, and the write-sets of these subranges. DistCL must keep track of how the data in a buffer is distributed, so that it knows when it needs to transfer data between nodes to satisfy subsequent reads - which may not occur on the same peer that originally produced the data.

DistCL represents the distribution of a buffer in a similar way to how it represents dependency information. The buffer is again divided into a set of intervals, but this time each interval is associated with the ID of the node that has last written to it. This node is referred to as the owner of that interval.

1) Buffers: Every time the host program creates a buffer, the master allocates a region of host (not device) memory, equal in size to the buffer, which DistCL uses to cache writes that the host program makes to the buffer. Whether the host program initializes the buffer or not, the buffer's dependency information specifies the master as the sole owner of the buffer. Additionally, each peer allocates, but does not initialize an OpenCL buffer of the specified size. Generally, most peers will never initialize the entire contents of their buffers because each subrange only accesses a limited portion of each buffer.

2) Satisfying Dependencies: When a subrange is assigned to a peer, before the subrange can execute, DistCL must ensure that the peer has an up-to-date copy of all the memory in the subrange's read-set. For every buffer, DistCL compares the ownership information to the subrange's read-set. If data in the read-set is owned by another node, DistCL initiates a transfer between that node and the assigned node. Once all the transfers have completed, the assigned peer can execute the subrange. When the kernel completes, DistCL also updates the ownership information to reflect the fact that the assigned peer now has the up-to-date copy of the data in the subrange's write-set. DistCL also implements host-enqueued buffer reads and writes using this mechanism.

3) Transfer Mechanisms: Peer-to-peer data transfers involve both intra-peer and inter-peer operations. For memory reads, data must first be transferred from the GPU into a host buffer. Then, a network operation can transfer that host buffer. For writes, the host buffer is copied back to the GPU. DistCL uses an OpenCL mechanism called mapping to transfer between the host and GPU.

IV. Experimental Setup

Eleven applications, from the Rodinia benchmark suite v2.3 [5] [6], AMD APPSDK [7], and GNU libgcrypt [8], were used to evaluate our framework. Each benchmark was run three times, and the median time was taken. This time starts when the host initializes the first buffer and ends when it reads back the last buffer containing the results, thereby including all buffer transfers and computations required to make it seem as if the cluster were one GPU with a single memory. The time for each benchmark is normalized against a standard OpenCL implementation using the same kernel running on a single GPU, including all transfers between the host and device. We group the benchmarks into three categories:

1) Linear compute and memory characteristics: nearest neighbor, hash, Mandelbrot;
2) Compute-intensive: binomial option, Monte Carlo;
3) Inter-node communication: n-body, bitonic sort, back propagation, HotSpot, k-means, LU decomposition.

These benchmarks were chosen to represent a wide range of data-parallel applications. They will provide insight into what type of workloads benefit from distributed execution. The three categories of problems give a spread of asymptotic complexities. This allows the effect of distribution, which primarily affects memory transfers, to be studied with tasks of varying compute-to-transfer ratios. The important characteristics of the benchmarks are summarized in Table I, and each is described below. We excluded Rodinia benchmarks that required image support, contained data-dependent writes, or contained too few work-groups. From the remaining benchmarks, five were chosen. For the Rodinia benchmarks, many of the problem sizes are quite small, but they were all run with the largest possible problem size, given the input data distributed with the suite. The worst relative standard deviation in runtime for any benchmark is 11%, with the Rodinia benchmarks' runtime varying the most due to their smaller problem sizes. For the non Rodinia benchmarks, it was usually under 1%.

A. Linear Compute and Memory

All linear benchmarks consist of n work-items and a single kernel invocation. For these benchmarks the amount of data transfered scales linearly with the problem size. The compute-to-transfer ratio remains constant regardless of problem size.

Nearest neighbor. This benchmark determines the nearest locations to a specified point from a list of available locations. Each work-item calculates the Euclidean distance between a single location and the specified point. The input buffer consists of n coordinates and the output is a n-element buffer of distances. Since 12 bytes are transferred per distance calculation, this benchmark has a very low compute-to-transfer ratio and is therefore poorly suited to distribution.

Hash. The hash benchmark attempts to find the hash collision of a sha-256 hash, similar to Bitcoin [9] miners. Each kernel hashes its global ID and compares it to the provided hash. Hash is well-suited to distribution because the only data transmitted is the input hash and a single byte from each work-item that indicates whether a collision was found.

Mandelbrot. This benchmark uses an iterative function to determine whether or not a point is a member of the Mandelbrot set. Each work-item iterates over a single point which it determines using its global ID. This benchmark is well suited to distribution because it has similar characteristics to the hash benchmark. There are no input buffers and only an n-element buffer that is written back after the kernel execution, giving it a high compute-to-transfer ratio.

B. Compute-Intensive

Binomial option. Binomial Option is used to value American options and is common in the financial industry. It involves creating a recombinant binomial tree that is n levels deep, where n is the number of iterations. This creates a tree with $n+1$ leaf nodes and one work-item calculates each leaf. The $n+1$ work-items take the same input, and only produce one result. Therefore, as the number of iterations is increased the amount of computation grows quadratically, as the tree gets both taller and wider, while the amount of data that needs to be transferred remains constant. This benchmark

Table I
BENCHMARK DESCRIPTION

Benchmark	Description	Source	Inputs	Complexity	Work-Items per Kernel	Kernels	Problem Size (bytes)
Nearest neighbor	Nearest neighbor search	Rodinia	42764 locations(n)	$O(n)$	n	1	$12n$
Mandelbrot	Mandelbrot set calculation	AMD	24M points(n) x: 0 to 0.5, y: -0.25 to 0.25 max iterations(k) 1000	$O(kn)$	n	1	$4n$
Hash	sha-256 cracking	Libgcrypt	24M hashes(n)	$O(n)$	n	1	$32 + n$
Bitonic	Parallel sort	AMD	32M elem.(n)	$O(n \lg^2 n)$	$\frac{n}{2}$	$\lg^2 n$	$4n$
Binomial	Binomial American option pricing	AMD	786432 samp.(k) 767 iterations(n)	$O(kn^2)$	$k(n+1)$	1	$32k$
Monte Carlo	Monte Carlo Asian option pricing	AMD	4k sums(n) 1536 samp.(m), 10 steps(k)	$O(knm^2)$	$\frac{m^2}{8n}$	k	$2m^2(2n+1)$
n-body	n-body simulation	AMD	768k bodies(n), 1,8 iter.(k)	$O(kn^2)$	n	k	$16n$
k-means	k-means clustering	Rodinia	819200 points(n) 34 features(k), 5 clusters(c)	kern. 1 $O(nk)$ kern. 2 $O(nck)$	n	1 variable	$8nk$ $4(2nk + c)$
Back propagation	neural network training	Rodinia	4M input nodes(n) 16 hidden nodes(k)	kern. 1 $O(nk)$ kern. 2 $O(nk)$	nk	2	$4(kn + 3n + 2k + 4)$ $4(kn + 2n + k + 3)$
HotSpot	Heat transfer	Rodinia	chip dim.(n) 1k, time-steps: per-kernel(x) 5, total(k) 60	$O(n^2)$	$256\lceil \frac{n}{16-2x} \rceil^2$	$\lceil \frac{k}{x} \rceil$	$4\left(\lceil \frac{n}{16-2x} \rceil (\frac{32}{8-x})\right)^2)$ $+4n^2$
LUD	LU-decomposition	Rodinia	matrix dim.(n) 2k $\frac{n}{16} - 1$ iterations (k) current iter. denoted(i)	kern. 1 $O(n)$ kern. 2 $O(n^2)$ kern. 3 $O(n^3)$	16 $2n - 32(i+1)$ $(n - 16(i+1))^2$	$k+1$ k k	$4n^2$ $4n^2$ $4n^2$

is very well suited to distribution. Since all samples can be valued independently, only a single kernel invocation is required.

Monte Carlo. This benchmark uses the Monte Carlo method to value Asian options. Asian options are far more challenging to value than American options, so a stochastic approach is employed. This benchmark requires $\frac{mn^2}{8}$ work-items and m kernel invocations, where n is the number of options, and m the number of steps used.

C. Inter-node communication

These benchmarks all have inter-node communication between kernel invocations, as opposed to the other benchmarks where nodes only need to communicate with the master. The inter-node communication allows the full path diversity of the network to be used when data is being updated between kernels. These benchmarks, like the others, require a high compute-to-transfer ratios to see a benefit from distribution.

n-body. This benchmark models the movement of bodies as they are influenced by each other's gravity. For n bodies and k iterations, this benchmark runs k kernels with n work-items each. Each work-item is responsible for updating the position and velocity of a single body, using the position and mass of all other bodies in the problem. Data transfers occur initially (when each peer receives the initial position, mass, and velocity for the bodies it is responsible for), between kernel invocations (when position information must be updated globally), and at the end (when the final positions are sent back to the host). As the number of bodies increases, the amount of computation required increases quadratically, while the amount of data to transfer only increases linearly, meaning that larger problems are better suited to distribution.

Bitonic sort. Bitonic sort is a type of merge sort well suited to parallelization. For an n-element array, it requires $\frac{n}{2} \lg^2 n$ comparisons, each of which are performed by a work-item, through $\lg^2 n$ kernel invocations. Each kernel invocation is a global synchronization point and could potentially involve data transfers. Bitonic sort divides its input into blocks which it operates on independently. While there are more blocks than peers, no inter-node communication takes place; only when the block are split between peers does communication begin.

k-means. This benchmark clusters n points into c clusters using k features. This benchmark contains two kernels: The first, which is only executed once, simply transposes the features matrix. The second kernel is responsible for the clustering. This kernel is executed until the result converges, which varies depending on the input data. For the largest input set available it took 20 kernel invocations before convergence. Both kernels consist of n work-items. For the first kernel, each work-item reads a row of the input array and writes it to a column of the output array. This results in a non-ideal memory access pattern for the writes. The second kernel reads columns of the features matrix, and the entirety of the cluster matrix which contains the centroid coordinates of each of the existing clusters. The writes of this kernel are contiguous because each work-item uses its one-dimensional global ID as an index into the array where it writes its answer.

Back propagation. This benchmark consists of the training of a two-layer neural network and contains two kernels. For a network with n input nodes and k hidden nodes, each kernel requires nk work-items. The work is divided such that each work-item is responsible for the connection between an input node and one of the hidden nodes. As the number of input nodes grows, the amount of computation required increases linearly since the number of hidden nodes is fixed for this benchmark.

HotSpot. This benchmark models processor temperature based on a simulated power dissipation profile. The chip is divided into a grid and there is a work-item responsible for calculating the temperature in each cell of the grid. The temperature depends on power produced by the chip at that cell, as well as the temperature of the four neighboring cells. To avoid having to transfer data between work-groups at each time-step this benchmark uses a “pyramid” approach. Since we need the temperature of all neighboring cells when updating the temperature, we will always read the temperature for a larger region than we will write. If we read an extra x cells in each direction we can find the temperature after x time-steps without any memory transfers. For each time-step, we calculate the updated temperature for a region that is smaller by one in each direction and that region then becomes the input for the next time-step. This creates a “pyramid” of concentric input regions of height x. While this results in less memory transfers it does mean that some work will be duplicated as there will be multiple work-groups calculating the temperature of overlapping regions during intermediate time-steps. The total amount of computation performed increases with x, while the amount of memory transferred decreases.

LU decomposition. This benchmarks factors a square matrix into unit lower triangular, unit upper triangular and diagonal matrices. LU decomposition consists of three kernels that calculate the diagonal, perimeter, and remaining values, respectively. These kernels operate over a square region of the matrix, called the area of interest. The problem is solved in 16×16 element blocks so for a matrix of size $n \times n$, LU decomposition requires $\frac{n}{16} - 1$ iterations. At each iteration the region of interest shrinks, losing 16 rows from the top and 16 columns from the left. Each iteration, the diagonal kernel updates a single block; the perimeter kernel updates the top 16 rows and left-most 16 columns of the area of interest; and the internal kernel updates the entire area of interest. After all the iterations, the diagonal kernel is run again to cover the bottom right block. While the perimeter and internal kernels can scale well, performance is limited by the diagonal kernel which consists of a single work-group and cannot be parallelized. This benchmark is not well suited to DistCL because of its inter-node communication, complex access pattern, and lack of parallelism.

D. Cluster

Our framework is evaluated using a cluster with an Infiniband interconnect [10]. The configurations and theoretical performance are summarized in Table II. The cluster consists of 49 nodes. Though there are two GPUs per node, we use only one to focus on distribution between machines. We present results for 1, 2, 4, 8, 16 and 32 nodes.

We use three microbenchmarks to test the cluster and to aid in understanding the overall performance of our framework. The results of the microbenchmarks are reported in Table III. We first test the performance of `memcpy()`, by copying a 64 MB array between two points in host memory. We initialize both arrays to insure that all the memory was paged-in before running the timed portion of the code. The measured memory bandwidth was 20.3 Gbps.

Table II
CLUSTER SPECIFICATIONS

Number of Nodes	49
GPUs Per Node	2 (1 used)
GPU	NVIDIA Tesla M2090
GPU memory	6 GB
Shader / Memory clock	1301 / 1848 MHz
Compute units	16
Processing elements	512
Network	4× QDR Infiniband (4 × 10 Gbps)
CPU	Intel E5-2620
CPU clock	2.0 GHz
System memory	32 GB

Table III
MEASURED CLUSTER PERFORMANCE

Transfer type	Test	64MB Latency ms (Gbps)	8B Latency ms (Mbps)
In-memory	Single thread `memcpy()`	26.5 (20.3)	0.0030 (21)
Inter-device	OpenCL map for reading	36.1 (14.9)	0.62 (0.10)
Inter-node	Infiniband round trip time	102 (10.5)	0.086 (3.0)

To test OpenCL map performance, a program was written that allocates a buffer, executes a GPU kernel that increments each element of that buffer, and then reads that buffer back with a map. The program executes a kernel to ensure that the GPU is the only device with an up-to-date version of the buffer. Every time the host program maps a portion of the buffer back, it reads that portion, to force it to be paged into host memory. The program reports the total time it took to map and read the updated buffer. To test the throughput of the map operation, the mapping program reads a 64MB buffer with a single map operation. Only the portion of the program after the kernel execution completes gets timed. We measured 14.9 Gbps of bandwidth between the host and the GPU. The performance of an 8-byte map was measured to determine its overhead. An 8-byte map takes 620 μs, equivalent to 100 kbps. This shows that small fragmented maps lower DistCL’s performance.

The third program tests network performance. It sends a 64MB message from one node to another and back. The round trip time for Infiniband took 102 ms and each one-way trip took only 51 ms on average, yielding a transfer rate of 10.5 Gbps. Since Infiniband uses 8b/10b encoding this corresponds to a signalling rate of 13.1 Gbps. This still fall short of the maximum signalling rate of 40 Gbps. Even using a high-performance Infiniband, network transfers are slower than maps and memory copies. For this reason it is important to keep network communication to a minimum to achieve good performance when distributing work. Infiniband is designed to be low-latency, and as such its invocation overhead is lower than that of maps.

E. SnuCL

We compare the performance of DistCL with SnuCL[1] [4], another framework that allows OpenCL code to be dis-

[1]Version 1.2 beta, downloaded November 15^{th} 2012.

tributed across a cluster. SnuCL can create the illusion that all OpenCL devices on the cluster belong to a single local context. To distribute a task with SnuCL, the programmer must partition the work into many kernels, ensuring that no two kernels write to the same buffer. SnuCL transfers memory between nodes automatically, but requires the programmer to divide their dataset into many buffers to ensure that each buffer is written to by only one node. For efficiency, the programmer should also divide up the buffers that are being read, to avoid unnecessary data transfers. The more regular a kernel's access pattern, the larger each buffer can be, and the fewer buffers there will be in total. With SnuCL, the programmer uses OpenCL buffer copies to transfer data. SnuCL will determine if a buffer copy is internal to a node, in which case it uses a normal OpenCL copy, or if it is between nodes, in which case it uses MPI. If subsequent kernel invocations require a different memory division, this task again falls to the programmer.

The buffers in SnuCL are analogous to the intervals generated by meta-functions in DistCL. However, in SnuCL, buffers must be explicitly created, resized and redistributed when access patterns change, whereas DistCL manages changes to intervals automatically. SnuCL does not abstract the fact that there are multiple devices, it only automates transfers and keeps track of memory placement. When using SnuCL, the programmer is presented with a single OpenCL platform that contains as many compute devices as are available on the entire cluster. The programmer is then responsible for dividing up work between the compute devices. Existing OpenCL code can be linked to DistCL without any algorithmic modification, reducing the likelihood of introducing new bugs. If SnuCL is linked to existing code all computation would simply happen on a single compute-device, as the code must be modified before SnuCL can distribute it.

We ported five of our benchmarks to SnuCL. We did not compare DistCL and SnuCL using the Rodinia benchmarks. While porting the Rodinia benchmarks to DistCL involved only the inclusion of meta-functions, porting the Rodinia benchmarks to SnuCL is a much more involved process involving modifications that would alter the characteristics of the Rodinia benchmarks. We were also unable to compare against inter-node communication benchmarks due a (presumably unintentional) limitation in SnuCL's buffer transfer mechanism. The one exception was n-body, which ran correctly for a single iteration, removing the need for inter-node communication and turning it into a compute-intensive benchmark.

Since kernels and buffers are subdivided to run using SnuCL, sometimes kernel arguments or the kernel code itself must be modified to preserve correctness. For example Mandelbrot requires two arguments that specify the initial x and y values used by the kernel. To ensure work is not duplicated, no two kernels can be passed the same initial coordinates. Kernels such as n-body require an additional offset parameter because per-peer buffers can be accessed using the new global ID, but globally shared buffers must be accessed with what would be the global ID if the problem were solved using a single NDRange. Hash also required an offset parameter since it used a work-item's global ID as the preimage. Similar changes must be made for any kernel that uses the value of its global ID or an input parameter to determine what part of a problem it is working on, rather than data from an input buffer.

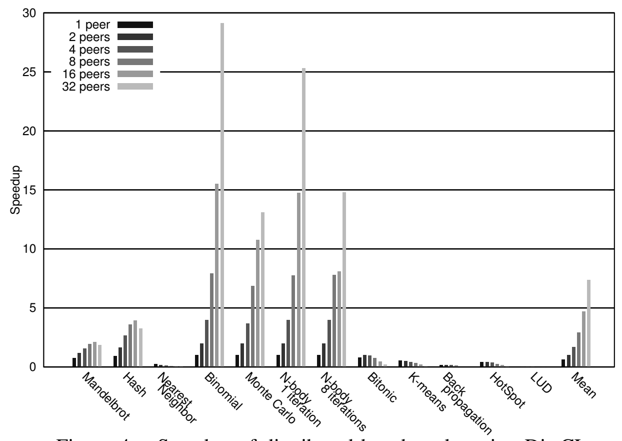

Figure 4. Speedup of distributed benchmarks using DistCL.

V. Results and Discussion

Figure 4 shows the speedups obtained by distributing the benchmarks using DistCL, compared to using normal OpenCL on a single node. Compute-intensive benchmarks see significant benefit from being distributed, with binomial achieving a speedup of over 29x when run on 32 peers. The more compute-intensive linear benchmarks, hash and Mandelbrot, also see speedup when distributed. Of the inter-node communication benchmarks, only n-body benefits from distribution, but it does see almost perfect scaling from 1-8 peers and speedup of just under 15x on 32 peers. For the above benchmarks, we see better scaling when the number of peers is low. While the amount of data transferred remains constant, the amount of work per peer decreases, so communication begins to dominate the runtime.

The remaining inter-node communication and linear benchmarks actually run slower when distributed versus using a single machine. These benchmarks all have very low compute-to-transfer ratios, so they are not good candidates for distribution. For the Rodinia benchmarks in particular, the problem sizes are very small. Aside from LU decomposition, they took less than three seconds to run. Thus, there is not enough work to amortize the overheads.

Figure 5 shows the speedups attained, by running the benchmarks on DistCL and SnuCL, relative to normal OpenCL run on a single node. Overall, the speedups are comparable, especially for the more compute-intense benchmarks, which saw the most speed up. Thus, when DistCL

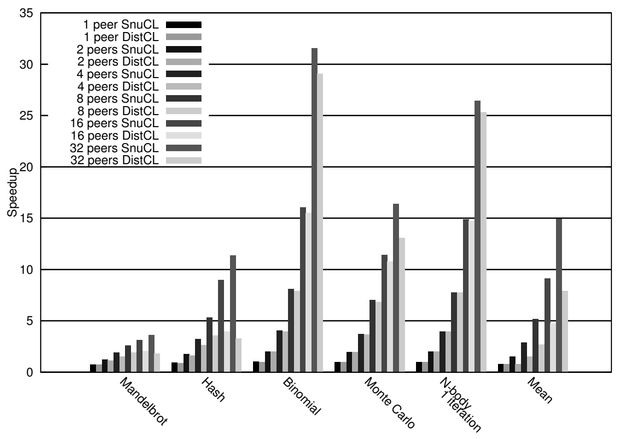

Figure 5. DistCL and SnuCL speedups.

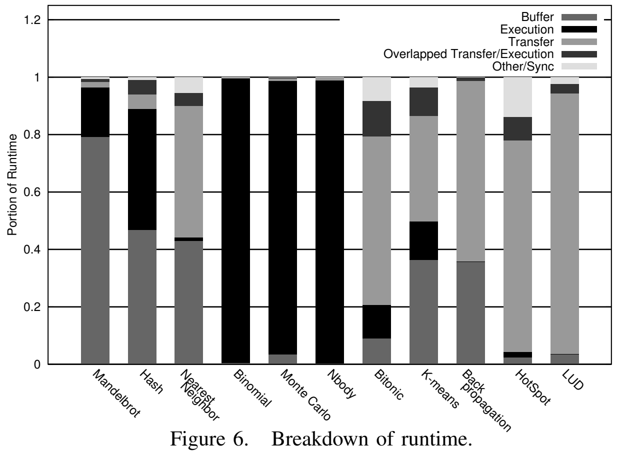

Figure 6. Breakdown of runtime.

and SnuCL are both capable of distributing workloads, those that distribute well see good performance using either framework. When SnuCL runs faster than DistCL, it is because SnuCL forces the task of tracking buffers onto the programmer, whereas DistCL uses extra synchronization in its dependency tracking algorithms.

Figure 6 show a run-time breakdown of the benchmarks for the 8 peer case. Each run is broken down into five parts: *buffer*, the time taken by host program buffer reads and writes; *execution*, the time during which there was at least one subrange execution but no inter-node transfers; *transfer*, the time during which there was at least one inter-node transfer but no subrange executions; *overlapped transfer/execution*, the time during which both subrange execution and memory transfers took place; and *other/sync*, the average time the master waited for other nodes to update their dependency information.

The benchmarks which saw the most speedup in Figure 4 also have the highest proportion of time spent in execution. The breakdowns for binomial, Monte Carlo, and n-body are dominated by execution time; whereas, the breakdowns for nearest neighbor, back propagation and LU decomposition are dominated by transfers and buffer operations, which is why they did not see a speedup. One might wonder why Mandelbrot sees a speedup, but bitonic and k-means do not, despite the proportion of time they spent in execution being similar. This is because Mandelbrot and hash are dominated by host buffer operations, which also account for a significant portion of execution with a single GPU. In contrast, Bitonic and k-means have higher proportions of inter-node communication, which map to much faster intra-device communication on a single GPU.

Table IV show the amount of time spent managing dependencies. This includes running meta-functions, building access-sets and updating buffer information. Table IV also shows the time spent per kernel invocation, and the time as a proportion of the total runtime. Benchmarks that have fewer buffers like Mandelbrot and Bitonic Sort spend less time applying dependency information per kernel invocation than benchmarks with more buffers. LU decomposition has the most complex access pattern of any benchmark. Its kernels operate over non-coalescable regions that constantly change shape. Further, the fact that none of LU decomposition's kernels update the whole array means that ownership information from previous kernels is passed forward, forcing the ownership information to become more fragmented, and take longer to process. With the exception of LU decomposition, the time spent managing dependencies is low, demonstrating that the meta-function based approach is intrinsically efficient.

An interesting characteristic of HotSpot is that the compute-to-transfer ratio can be altered by changing the pyramid height. The taller the pyramid the higher the compute-to-transfer ratio. However, this comes at the price of doing more computation than necessary. Figure 7 shows the speedup of HotSpot run with a pyramid height of 1, 2, 3, 4, 5, and 6. The distributed results are for 8 peers. Single GPU results were acquired using conventional OpenCL. In both cases, the speedups are relative to that framework's performance using a pyramid height of 1. The number of time-steps used was 60 to ensure that each height was a divisor of the number of time-steps. We can see that for a single GPU, the preferred pyramid height is 2. However, when distributed the preferred size is 5. This is because with 8 peers we have more compute available but the cost of memory transfers is much greater, which shifts the sweet spot toward a configuration that does less transfer per computation.

Benchmarks like Hotspot and LU decomposition that write to rectangular areas of two-dimensional arrays need special attention when being distributed. While the rectangular regions appear contiguous in two-dimensional space, in a linear buffer, a square region is, in general, not a single interval. This means that multiple OpenCL map and network operations need to be performed every time one of these areas is transferred.

We modified the DistCL scheduler to divide work along the y-axis to fragment the buffer regions transfered between

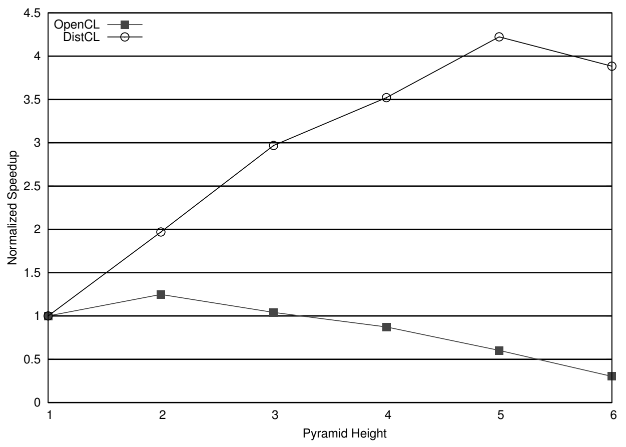

Figure 7. HotSpot with various pyramid heights.

Table IV
EXECUTION TIME SPENT MANAGING DEPENDENCIES

Benchmark	Total Time (μs)	Per Kernel Invocation Time (μs)	Percent of Runtime
Mandelbrot	109	109	0.097
Hash	112	112	0.15
Nearest neighbor	120	120	0.62
Binomial	126	126	0.0043
Monte Carlo	1500	150	0.028
n-body	166	166	0.00045
Bitonic Sort	29900	91.9	2.9
k-means	30400	1450	4.6
Back propagation	434	217	0.017
HotSpot	23500	981	5.9
LUD	2.31×10^7	60400	30

peers. This results in performance that is 204× slower on average across all pyramid heights, for 8 peers. This demonstrates that the overhead of invoking I/O operations on a cluster is a significant performance consideration.

In summary, not only does DistCL have similar performance to SnuCL, but it is easier to use. DistCL also exposed important characteristics regarding distributed OpenCL execution. Distribution amplifies the performance characteristics of GPUs. Global memory reads become even more expensive compared to computation, and the aggregate compute power is increased. Further, the performance gain seen by coalesced accesses is not only realized in the GPU's bus, but across the network as well. Synchronization - now a whole-cluster operation - becomes even higher latency. There are also aspects of distributed programming not seen with a single GPU. Sometimes, it is better to transfer more data with few transfers than it is to transfer little data with many transfers.

VI. RELATED WORK

As described in IV-E, SnuCL is another framework that distributes OpenCL kernels across a cluster. SnuCL can create the illusion that all the OpenCL devices in a cluster belong to a local context, and can automatically copy buffers between nodes based on the programmer's placement of kernels. As opposed to SnuCL, DistCL not only abstracts inter-node communication, but also the fact that there are multiple devices in the cluster.

Other prior work also proposes to dispatch work to remote GPUs in a cluster environment as if they were local. rCUDA [11] provides this functionality for Nvidia devices and Mosix VCL [12] provides this functionality for OpenCL on Linux-based systems. Hybrid OpenCL [13], dOpenCL [14], and Distributed OpenCL [15] provide this functionality cross vendor and cross platform. In clOpenCL [16] each remote node is represented locally as a separate platform. All these works, as well as SnuCL, require the programmer to create a separate set of buffers and kernel invocations for each device and rely on explicit commands from the host program for communication between peers. libWater [17] is also similar to the above works, but takes advantage of task dependencies inferred using the OpenCL event model to automatically schedule tasks efficiently.

Partitioning of a single kernel invocation has been addressed in a limited fashion. CUDASA [18] extends the CUDA [19] programming model to include *network* and *bus* levels on top of the pre-existing *kernel*, *block* and *thread* levels. Although CUDASA does provide a mechanism for the programmer to distribute a kernel between computers, it requires them to make explicit function calls to move memory between computers. The kernel code must also be modified in order to be distributed.

Work by Kim et al. [20] transparently distributes OpenCL kernels between multiple GPUs on the same PCIe bus. Instead of using meta-functions, it uses compiler analysis and sample runs of certain work-items to determine the memory accesses that will be performed by a subset of a kernel. Since the kernel is only divided among local GPUs there is no need to worry about network delays or manage multiple processes. Our approach to dividing up kernels falls somewhere between these two approaches. It allows programmers to direct the framework using relevant information about a kernel's memory accesses, but relieves them of the burden of having to manually partition the work or buffers making the code more portable.

VII. CONCLUSION

We present DistCL, a framework for distributing the execution of an OpenCL kernel across a cluster, causing that cluster to appear as if it were a single OpenCL device. DistCL shows that it is possible to efficiently run kernels across a cluster while preserving the OpenCL execution model. To do this, DistCL uses meta-functions that abstract away the details of the cluster and allow the programmer to focus on the algorithm being distributed. We believe the meta-function approach imposes less of a burden than any other OpenCL distribution system to date. Speedups of up to 29 on 32 peers are demonstrated.

With a cluster, transfers take longer than they do with a single GPU, so more compute-intense approaches perform better. Also, certain access patterns generate fragmented memory accesses. The overhead of doing many fragmented

I/O operations is profound and can be impacted by partitioning.

By introducing meta-functions, DistCL opens the door to distributing unmodified OpenCL kernels. DistCL allows a cluster with 2^{14} processing elements to be accessed as if it were a single GPU. Using this novel framework, we gain insight into both the challenges and potential of unmodified kernel distribution. In the future, DistCL can be extended with new partitioning and scheduling algorithms to further exploit locality and more-aggressively schedule subranges. DistCL is available at http://www.eecg.toronto.edu/~enright/downloads.html.

Acknowledgments

We thank the reviewers for their feedback, Prof. Andreas Moshovos for his guidance, and the members of Prof. Enright Jerger's research group for their suggestions. This work was supported by NSERC, OCE, and AMD. Computations were performed on the Gravity supercomputer at the SciNet HPC Consortium. SciNet is funded by: the Canada Foundation for Innovation under the auspices of Compute Canada, the Government of Ontario, Ontario Research Fund - Research Excellence, and the University of Toronto.

References

[1] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Int'l Symp. on Performance Analysis of Systems and Software*, 2009, pp. 163–174.

[2] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. of the Symp. on Principles and practice of parallel programming*, 2008, pp. 73–82.

[3] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, and W. Hwu, "GPU clusters for high-performance computing," in *IEEE Int'l Conf. on Cluster Computing and Workshops*, 2009, pp. 1–8.

[4] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," in *Proc. of the Int'l Conf. on Supercomputing*, 2012, pp. 341–352.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE Int'l Symp. on Workload Characterization*. IEEE, 2009, pp. 44–54.

[6] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *Int'l Symp. on Workload Characterization*. IEEE, 2010, pp. 1–11.

[7] AMD, "AMD accelerated parallel processing (APP) SDK," 2011, http://developer.amd.com/sdks/amdappsdk/pages/default.aspx.

[8] Free Software Foundation, "GNU libgcrypt," 2011, http://www.gnu.org/software/libgcrypt.

[9] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009, www. bitcoin. org.

[10] C. Loken, D. Gruner, L. Groer, R. Peltier, N. Bunn, M. Craig, T. Henriques, J. Dempsey, C.-H. Yu, J. Chen *et al.*, "Scinet: Lessons learned from building a power-efficient top-20 system and data centre," in *Journal of Physics: Conference Series*, vol. 256, no. 1. IOP Publishing, 2010, p. 012026.

[11] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Orti, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *Int'l Conf. on High Performance Computing and Simulation*. IEEE, 2010, pp. 224–231.

[12] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices," in *IEEE Int'l Conf. on Cluster Computing*, 2010, pp. 1–7.

[13] R. Aoki, S. Oikawa, T. Nakamura, and S. Miki, "Hybrid OpenCL: Enhancing OpenCL for distributed processing," in *Int'l Symp. on Parallel and Distributed Processing with Applications*, 2011, pp. 149–154.

[14] P. Kegel, M. Steuwer, and S. Gorlatch, "dopencl: Towards a uniform programming approach for distributed heterogeneous multi-/many-core systems," in *Int'l Symp. on Parallel and Distributed Processing Workshops & PhD Forum*. IEEE, 2012, pp. 174–186.

[15] B. Eskikaya and D. Altılar, "Distributed OpenCL distributing OpenCL platform on network scale," *Int'l Journal of Computer Applications*, vol. ACCTHPCA, no. 2, pp. 25–30, July 2012.

[16] A. Alves, J. Rufino, A. Pina, and L. P. Santos, "clopencl-supporting distributed heterogeneous computing in hpc clusters," in *Euro-Par 2012: Parallel Processing Workshops*. Springer, 2013, pp. 112–122.

[17] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer, "Libwater: heterogeneous distributed computing made easy," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 161–172.

[18] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl, "CUDASA: Compute unified device and systems architecture," in *Eurographics Symp. on Parallel Graphics and Visualization*, 2008, pp. 49–56.

[19] Nvidia Corporation, "Programming guide," 2008.

[20] J. Kim, H. Kim, J. Lee, and J. Lee, "Achieving a single compute device image in OpenCL for multiple GPUs," in *Proc. of the 16th Symp. on Principles and practice of parallel programming*, 2011, pp. 277–288.