

# A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices

Amnon Barak, Tal Ben-Nun, Ely Levy and Amnon Shiloh  
Department of Computer Science  
The Hebrew University of Jerusalem  
Jerusalem 91904, Israel

**Abstract**—Heterogeneous systems provide new opportunities to increase the performance of parallel applications on clusters with CPU and GPU architectures. Currently, applications that utilize GPU devices run their device-executable code on local devices in their respective hosting-nodes. This paper presents a package for running OpenMP, C++ and unmodified OpenCL applications on clusters with many GPU devices. This Many GPUs Package (MGP) includes an implementation of the OpenCL specifications and extensions of the OpenMP API that allow applications on one hosting-node to transparently utilize cluster-wide devices (CPUs and/or GPUs). MGP provides means for reducing the complexity of programming and running parallel applications on clusters, including scheduling based on task dependencies and buffer management. The paper presents MGP and the performance of its internals.

**Index Terms**—HPC cluster, GPGPU computing, OpenCL, OpenMP, parallel applications

## I. INTRODUCTION

Heterogeneous computing systems provide an opportunity to dramatically increase the performance of parallel and High-Performance Computing (HPC) applications on clusters with CPU and GPU architectures. This can be accomplished by combining stream based General-Purpose GPU (GPGPU), hence devices, and traditional multi-core CPUs. However, without simple programming models and adequate run-time environments, the complexity required to modify serial code so that it runs in parallel is very high. This is due to the need to partition (parallelize) and run the (CPU) program on different cluster nodes, e.g. as in MPI; then to develop and optimize the (GPU) device executable code, then to run and manage this low level code and to interface and synchronize between that code and the CPU program.

Development of parallel applications is usually simpler in OpenMP [1] than in MPI [2], mainly because OpenMP, as opposed to MPI, supports shared-memory, multi-threads and fine granularity. However, pure OpenMP is confined to a single computer, while MPI can be scaled up to a large number of nodes. Modifying serial to parallel code usually requires more changes in MPI and thus more programming efforts. Nevertheless, the ability to run HPC applications on low-cost clusters makes MPI more popular. Indeed, MPI is being used by several projects that are developing GPU clusters [3], [4] and TSUBAME [5], which uses MPI on different nodes over threaded OpenMP and Pthreads within each node.

The main challenge of GPGPU computing is the development and optimizations of device executable code, to exploit high degree of parallelism. Developers can choose between the OpenCL standard programming environment [6], which allows applications to run across GPU and CPU platforms of most vendors; or a vendor specific programming environment, such as CUDA [7], [8]. Currently, most applications, including MPI applications that utilize GPU devices, run their kernel-code on local devices where their CPU programs were started.

This paper presents the Many GPUs Package (MGP) that can run extended OpenMP, C++ and unmodified OpenCL applications transparently on clusters with many GPU devices. MGP is geared to simplify the development of parallel programs and to reduce the management complexity needed to run these program on a cluster. For convenience, programs are provided with a simple API and the impression of a single host with many devices (single-system image). To achieve these goals, we implemented an extended OpenMP API above our own C++ API and a version of OpenCL that allows applications to transparently utilize cluster-wide devices (CPUs and/or GPUs). With MGP, users can start a parallel application on a (hosting) computer, (which could be the user's workstation, a cluster's node, a server, etc.), then MGP manages and transparently runs the kernels of the application on different nodes of a cluster, just like MPI.

Two approaches exist for harnessing the benefits of both the OpenMP and MPI models for running parallel programs on clusters. One approach uses MPI on top of OpenMP: MPI distributes tasks to cluster-nodes, while OpenMP distributes the tasks further within each node. In the second approach, OpenMP establishes a cluster-wide Distributed Shared-Memory (DSM), which is implemented using MPI [9]. While the second approach is attractive due to OpenMP's ease of programming, its main disadvantage is the complexity and overhead to support DSM in large-scale configurations. MGP introduces a new, MOSIX-like [10], [11] approach. It runs the CPU part of the application on a single node, thus avoiding the problems associated with DSM; and the GPU kernels on cluster-wide devices. As such, MGP gains the reduced programming complexity of OpenMP, as well as concurrent access to devices in many nodes, as in MPI. Note that MGP can still be combined with MPI, by allowing each MPI task that runs on a satellite node to utilize cluster-wide devices.

We believe that our approach can benefit most users due to the availability of low-cost multi-core computers, with large shared-memory that can utilize cluster-wide (CPU and GPU) resources transparently. As we shall show, the overhead implied by MGP is reasonable for most non interactive applications.

MGP has two layers. The MOSIX VCL layer, described in Sec. II, is an implementation of the OpenCL specifications and a set of OpenCL extensions that allow unmodified OpenCL applications to transparently utilize many devices in a cluster. This layer also provides a run-time environment to the API layer in which all the cluster devices are seen as if they are located in the hosting-node. The API layer, described in Sec. III, provides an extended OpenMP APIs for writing applications that run kernels on many devices. The implementation of this API is based on a C++ library, which also provides a scatter-gather API. Parallel applications that are developed using the APIs are provided with an easy-to-use task-based run-time environment for transparently running tasks on many devices. Such applications are also provided with queueing, scheduling based on task dependencies and buffer management services.

To test the performance of MGP, we used various tests, mostly applications from the Scalable Heterogeneous Computing (SHOC) benchmark suite [3]. Briefly, SHOC contains OpenCL applications that test the performance and stability of GPUs and multi-core processors. The cluster version of SHOC uses MPI to distribute tasks to nodes in a cluster.

The tests were performed on a cluster with (Intel 4-way Core i7 CPU) nodes connected by a Quad Data Rate (QDR) Infiniband, each with an NVIDIA GeForce GTX 480 (Fermi) GPU; and workstations (Intel Core 2 Duo CPU) without GPUs, that were connected to the cluster by a 1Gb/s Ethernet. Each test was conducted 5 times and the average result is presented.

The paper presents the layers of MGP and their performance. Related works are described in Sec. IV and our conclusions in Sec. V.

### A. Definitions

The following terms, which are related to many devices in a cluster, are used throughout the paper:

- **GPGPU (GPU)**: a general-purpose graphic processing unit capable of running device specific code.
- **OpenCL device (device)**: any CPU or GPU that is supported by the locally installed OpenCL drivers.
- **Hosting-node (host)**: a computer, often a workstation, with or without GPUs, on which applications are started.
- **Node**: a computer with one or more OpenCL devices.
- **GPU cluster (cluster)**: a collection of nodes, usually servers, each with one or more GPU devices.
- **GPU task (task)**: an instance of device executable code (kernel), its arguments and the input and output memory-objects that it requires.
- **Heterogeneous application (application)**: an executable that includes both host and device code. It starts by running CPU code on the hosting-node, then runs tasks on local or non-local (remote) devices.

## II. THE MOSIX VCL LAYER

This section presents a MOSIX-like [10] cluster-wide Virtual implementation of OpenCL (VCL) that allows unmodified applications to transparently utilize many OpenCL devices (CPUs and/or GPUs) in a cluster. This is accomplished by a run-time environment in which all the cluster devices are seen as if they are located in each hosting-node - applications need not to be aware which nodes and devices are available and where the devices are located. The resulting platform benefits OpenCL applications that can use multiple devices concurrently.

Applications may create OpenCL “contexts” that span devices on several nodes; or multiple contexts each with the devices of a different node; or a combination of the above. Another alternative is to split job(s) into several independent processes, each running on a different set of devices. Applications may be specific about which devices they include in their contexts, but for the benefit of completely unmodified applications, we also allow environment-variables to control the device allocation policies. By default, each context that is created includes all the devices of a single node.

Remote nodes perform OpenCL functions on behalf of application(s) on the host(s). In order to support application concurrency on different devices, our implementation is fully thread-safe (even though this is not a requirement of the OpenCL specifications). VCL communicates between hosts and remote nodes using standard TCP/IP sockets. When running OpenCL on remote devices, network latency is the main limiting factor: we therefore attempt to minimize the number of network round-trips for existing applications, to the extent possible without compromising the compliance to the OpenCL-specifications. In order to improve performance beyond the limitations of the OpenCL specification, we also implemented a set of extensions that allow a fully programmable sequence of kernels (for example, running two different kernels alternately  $N$  times) to be run at once on remote devices, often with just a single network round-trip.

Our implementation supports, and can even mix, different devices from all vendors (as long as the corresponding OpenCL libraries are installed). We note that our implementation supports almost all the OpenCL library calls, with the exception of direct mapping of device memory (the “clEnqueueMapBuffer” and the “clEnqueueMapImage” functions) that is unsuitable among disjoint nodes.

### A. Raw Performance of the MOSIX VCL Layer

The differences between running kernels on local devices using the “native” OpenCL library vs. running kernels using MGP on local and remote devices, are the overhead of the MGP management and the network delay when accessing remote devices.

To find these differences, we compared between the time required by an application to run a sequence of identical kernels using the native OpenCL library and the times to run the same kernels using the MGP VCL layer on local and remote devices.

### Overhead to Start Kernels

The first test measured the net time of running 1000 minimal kernels on a designated (local and remote) device. The time to compile the program and to copy buffers to/from the device was deliberately excluded.

The results, for buffer sizes ranging from 4KB - 512MB are shown in Table I: Column 1 lists the size of the buffer that is passed to the OpenCL kernel. Column 2 shows the native OpenCL library times. Column 3 shows the times to run the kernels with MGP on a local device and Column 4 the difference (Diff.) between Column 3 and Column 2; Column 5 shows the times to run on a remote device and Column 6 the difference between Column 5 and Column 2.

TABLE I  
NATIVE VS. LOCAL VS. REMOTE RUN-TIMES

Buffer Size	Native Time (ms)	MGP Times			
		Local (ms)	Diff. (ms)	Remote (ms)	Diff. (ms)
4KB	26	56	30	107	81
16KB	26	56	30	107	81
64KB	27	57	30	108	81
256KB	29	59	30	110	81
1MB	45	75	30	126	81
4MB	100	131	31	181	81
16MB	321	354	33	407	86
64MB	1207	1244	37	1298	91
256MB	4762	4800	38	4854	92
512MB	9498	9535	37	9590	92

From Table I it can be seen that the difference between the local/remote times and the corresponding native times are consistent, small constants for all buffer sizes. Since this constant is the net time to start 1000 kernels, it follows that starting a kernel by the VCL layer on a local device takes on average  $\sim 33\mu s$  longer than by the native OpenCL library; and  $\sim 85\mu s$  longer on a remote device, a reasonable overhead for most parallel, including HPC applications.

### Multiple Devices vs. Many Devices

We repeated the above test with different combinations of 2 - 4 devices, some combinations on a single remote node and others on several remote nodes simultaneously. The results showed no significant differences compared to a single remote device.

### B. Performance of the MOSIX VCL Layer

To test the performance of the VCL layer, we used the two dimensional Fast Fourier Transform (FFT) from the SHOC benchmark. This test consists of three-phase iterations on a 100MB buffer: FFT, inverse FFT and data comparison [3]. Due to the relatively short run-time of each iteration, we ran the program with 1,000 - 8,000 iterations.

Fig. 1 shows the run-time overhead (in %'s) of the FFT program using the VCL layer on a local and a remote device, compared to the "native" OpenCL library run-time on a local device. Note that the remote times include a one time buffer copy. From the obtained results it follows that for this test, the

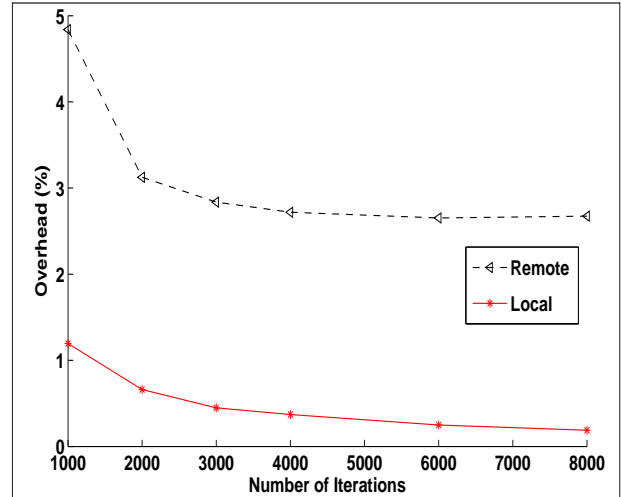


Fig. 1. Overhead of the MOSIX VCL layer (%)

average overhead of the VCL layer on local/remote devices are 0.31%/2.73% respectively.

## III. THE API LAYER

This section describes a simple C++ API for writing parallel applications that can transparently run kernels on many devices. This API is implemented by the Many-GPU Library (MGL). Devices are automatically detected so that applications need not refer to devices at all. We also describe a slightly modified extended OpenMP API based on the suggestion by Ayguadé et al. [12]. These extensions are implemented using the C++ API. Both the C++ and the OpenMP APIs are task-based, where tasks are currently written as OpenCL kernels [6]. Support for CUDA [7] and DirectCompute [13] can also be added. The API layer also includes a run-time environment that provides queuing, scheduling, buffer management, handling dependencies between tasks and other services.

### A. API for C++ Applications

The C++ API is designed to be simple, yet sufficiently powerful to develop parallel applications for many devices. It also supports other advanced features such as scatter-gather, profiling and debugging.

The object-oriented API for C++ programs includes three basic classes: context, task and buffer. Their methods are shown in Table II.

The `MGL::Context` class initializes and destroys the library run-time environment. One parameter of `CreateContext` is the type of backend, e.g. OpenCL or CUDA. The `HasBackend` function checks whether a specific library backend is available. The `Enqueue` method queues a task, without requiring the program to specify a device.

The `MGL::Task` class is a logical representation of a (GPU) task. The basic task class includes information about

TABLE II  
THE C++ API CLASSES AND METHODS

class MGL::Context	Description
<b>CreateContext</b>	Initialize a new context
<b>GetContext</b>	Return the current context
<b>Enqueue</b>	Enqueue one or more tasks
<b>HasBackend</b>	Return true if backend is supported
<b>Destroy</b>	Destroy the context
class MGL::Task	Description
<b>Task</b>	Initialize a new task
<b>GetState</b>	Return current task state
<b>GetError</b>	Return error code, if applicable
<b>GetRuntime</b>	Return task run-time
<b>Wait</b>	Block until this task is done
class MGL::Buffer	Description
<b>Buffer</b>	Initialize a new buffer
<b>GetData</b>	Read buffer data if not locked
<b>SetData</b>	Write buffer data if not locked
<b>SetAccess</b>	Set buffer access type (input/output)

the kernel and its I/O buffers. The `MGL::Task` class may be extended to include multiple kernels or implement advanced algorithms, such as scatter-gather.

The `MGL::Buffer` class is a backend-independent container for the task buffers that handles data locking. The `Buffer` class makes it unnecessary to manually allocate, copy and release memory buffers on devices.

### B. OpenMP Extensions for Many-GPUs

This section describes an implementation of OpenMP-like directives, based on the heterogeneous architecture extensions of OpenMP suggested by Ayguadé et al. [12].

The syntax of the first directive is:

```
#pragma mgl kernel target(lang)
implements(name)
```

This directive appears before a kernel written in `lang`, e.g., OpenCL, where the kernel is identified by name.

The syntax of the second directive is:

```
#pragma mgl task input(...) output(...)
global(x[,y,z]) [local(x2,y2,z2)]
```

This directive runs the task specified in the subsequent line. The global dimensions of the kernel are  $(x, y, z)$  and its local dimensions are  $(x_2, y_2, z_2)$ . Note that the `local` argument was added to the syntax suggested by Ayguadé et al. [12], to support work-groups.

The syntax of the third directive is:

```
#pragma mgl taskwait
```

This directive waits for all the currently running tasks in this OpenMP scope to finish.

Our implementation of the OpenMP extensions statically analyzes the code for the above directives. The code is then parsed into valid OpenMP code and the corresponding library calls. This code is then compiled by an OpenMP supporting compiler, such as GCC or ICC.

### C. The Scatter-Gather API

The Scatter-gather feature allows buffers to be divided into disjoint segments that can be transparently scattered to multiple devices, along with the same kernel. The underlying mechanism provides subdivision; boundary exchange and memory I/O optimizations.

Briefly, a *segment* is a contiguous portion of a buffer. The *scatter* operation splits a task into several sub-tasks, so that each sub-task operates on *segments* of one or more buffers using the same kernel. The reverse operation, *gather*, combines all the *segments* of scattered buffers after completion of the sub-tasks. A *boundary* is a type of *segment* that represents data required by other (typically adjacent) sub-tasks. *Exchange* is an operation performed when a scatter-gather task with multiple steps (iterations) swaps pre-defined *boundaries* among sub-tasks, between iterations. The scatter-gather API is described in Table III.

TABLE III  
THE SCATTER-GATHER CLASSES AND METHODS

class MGL::ScatterTask	Description
<b>ScatterTask</b>	Initialize a new scatter-gather task
<b>Exchange</b>	Exchange boundaries between sub-tasks
<b>Gather</b>	Gather task results
class MGL::ScatterHandler	Description
<b>ScatterHandler</b>	Initialize a new buffer scatter handler
<b>GetReadSegment</b>	Return read offsets of a sub-task
<b>GetWriteSegment</b>	Return write offsets of a sub-task
<b>GetBoundary</b>	Return a specific boundary required by a sub-task for Exchange
<b>GetNumBoundaries</b>	Return number of boundaries required by a sub-task for Exchange
<b>GetSubTaskSize</b>	Return total size of a sub-task buffer
<b>GetSubTaskGlobalDim</b>	Return the global dimensions of a sub-task
<b>GetNumSubTasks</b>	Return the number of sub-tasks

The `ScatterHandler` interface was designed to provide programs with fine control over scatter-gather features such as boundary exchange. This abstract class can be used for defining various scatter-gather patterns of data structures. For example, we implemented an `ImageScatter` class that splits an image to several two-dimensional blocks.

The class `ScatterTask` implements all of the methods defined in `Task`. Internally, queueing a `ScatterTask` (using `Context::Enqueue`) split it into sub-tasks, with appropriate dependencies. The sub-tasks are tasks in themselves and are placed in the queue. The `Exchange` method causes each sub-task to receive its *boundaries* from other sub-tasks. `Gather` waits for all the sub-tasks to complete, then collects the resulting buffers. `GetRuntime` returns the sum of all the sub-task run-times.

The `ScatterHandler` interface includes the methods required to define the *scatter* and *exchange* operations. It is designed to be as flexible as possible and it utilizes two data structures: `Segment` and `Boundary`. A `Segment` contains the global offset in the buffer, its local offset in the sub-buffer and its size. The `GetReadSegment/GetWriteSegment` methods return the read/write *segment* of a given sub-task.

C++ API	OpenMP API
<pre>const char *PROGRAM = " kernel void mul2(global float *dst) {     int id = get_global_id(0);     dst[id] *= 2.0f; }";</pre>	<pre>#pragma mgl kernel target(opencl) implements(mul2) kernel void mul2(global float *dst) {     int id = get_global_id(0);     dst[id] *= 2.0f; }</pre>
<pre>using namespace MGL; Context::CreateContext("OpenCL"); int dims = 256; // Multiplying x by 2 Buffer *gX = new Buffer(x, sizeof(float) * dims,     INOUT, MEMORY); Task *task = new Task(PROGRAM, "", "mul2", &amp;gX,     1, 1, &amp;dims, NULL); Context::GetContext().Enqueue(task);</pre>	<pre>int dims = 256; // Multiplying x by 2 #pragma mgl task input(x) output(x) global(dims) mul2(x);</pre>
<pre>task-&gt;Wait(STATE_FINISHED); delete gX; delete task; Context::Destroy();</pre>	<pre>#pragma mgl taskwait</pre>

Fig. 2. Example of code in C++ vs. OpenMP

A *boundary* required for an *exchange* is represented by the `Boundary` class. Note that a sub-task may require more than one *boundary* per *exchange*. For example, the `ImageScatter` class requires four *boundaries*, one for each edge.

#### D. Example of C++ vs. OpenMP

An example of code that uses MGL is shown in Fig. 2. The top block shows how a kernel is defined in C++ (on the left) and OpenMP (on the right). Observe that in C++, the kernel code is a string, while in OpenMP it is not.

The mid-left block shows how to queue a simple task in C++. To create a `Task` object, the user provides the constructor with a kernel, compile arguments and I/O buffers. The task is then queued using `Enqueue`; and automatically assigned to an available (free) device, on which it runs asynchronously. In the corresponding OpenMP version, the context, buffers and tasks are automatically created. The programmer only has to specify the “task” directive.

The lower block shows how to wait for a task and destroy the MGL context. In the C++ version (on the left), the program can either `Wait` for the task, or run it in the background and check its state (using `GetState`). Note that the `Buffer` data can only be read by the program after the task has finished and the data was received (using the `GetData` method). In the corresponding OpenMP version, the “taskwait” directive must be called before the program can read the output buffers.

#### E. Performance of API Layer on a Cluster

To show the performance of the API layer on a cluster, we ported the SHOC FFT program to the MGP environment, without modifying the OpenCL kernel. The layout of the program is shown in Fig. 3. Its implementation in C++ takes 159 lines.

The program was started by creating a 256MB buffer in one node, then it used the single-queue to launch independent sets of kernels on (local and remote) nodes, implicitly copying the buffer to each device along with the first kernel.

```
// Create context and initialize buffers
work->SetAccess(MGL::INOUT);
fft = new Task(...);
Context::GetInstance().Enqueue(fft);

ifft = new Task(...);
Context::GetInstance().Enqueue(ifft);

ifft->Wait();

work->SetAccess(MGL::INPUT);
verification = new Task(...);
Context::GetInstance().Enqueue(verification);
verification->Wait();

while(!checkBuffer->GetData((char *)&failed, 0,
    sizeof(int)));

// Destroy context, tasks and buffers
return failed;
```

Fig. 3. Layout of FFT in MGP

TABLE IV  
FFT RUN-TIMES ON A CLUSTER

Number of Iterations	Native Time (Sec.)	4-Nodes		8-Nodes	
		Time (Sec.)	Speedup	Time (Sec.)	Speedup
1000	42.340	19.272	2.19	16.296	2.60
2000	82.250	30.108	2.73	22.026	3.73
4000	162.170	52.578	3.08	33.372	4.86
8000	321.910	97.532	3.29	55.950	5.74

The results are shown in Table IV. Column 2 shows the native FFT run-times on a local device. Column 3 is the total run-times, including the buffer copies, on 4 nodes; Column 4 is the speedup, obtained by the ratio between Column 2 and Column 3; The corresponding results for 8 nodes are shown in Columns 5 and 6. From the results we can see a gradual increase in the speedups along with the increase in the size of the cluster and the number of iterations.

In comparison, the MPI run-times of the original SHOC

FFT program were nearly identical to the run-times on a local device (shown in Column 2). However, the SHOC FFT test uses random values, avoiding any buffer transfer over the network. We note that the MPI implementation takes 325 lines.

#### F. Performance of the Stencil2D on a Cluster

This section presents the ease of programming and the performance of the scatter-gather API using the Stencil2D parallel application from the SHOC benchmark. Stencil2D runs a standard two-dimensional 9-point stencil calculation, performing a weighted average with varying cardinal, diagonal and center weights on each matrix element [3]. The SHOC kernel computes the stencil for one matrix element per work-item; buffer transfers are optimized by swapping the source and destination matrices after each iteration.

The SHOC MPI implementation divides the matrix into a grid of square blocks. Each node then runs one iteration on its block and exchanges the boundaries. To simplify the boundary handling, the current implementation copies the entire block to/from the host for each iteration. Upon completion, the matrix is gathered from all the satellites. We note that the MPI implementation takes 655 C++ lines.

```
// Create context, buffers and scatterhandlers
for (int i = 0; i < iters; i++) {
    stenciltask = new ScatterTask(...);

    Context::Enqueue(stenciltask);
    if (i == iters - 1)
        stenciltask->Gather();
    else
        stenciltask->Exchange();

    delete stenciltask;

    swap(source, dest);
}

// Return the resulting matrix
source->GetData(...);

// Delete context, buffers and scatterhandlers
```

Fig. 4. Layout of Stencil2D in MGP

Stencil2D was implemented in MGP, without modifying the SHOC kernel. In this implementation, the matrix is divided into contiguous horizontal stripes. Initially, the application scatters the buffers using stripe ScatterHandlers. Each sub-task runs one iteration on its stripe, then exchanges its boundaries with its neighboring sub-tasks (using Exchange). After the last iteration, Gather collects the resulting matrix. The layout of the MGP program is shown in Fig. 4. Its implementation in C++ takes 64 lines of code.

We ran Stencil2D on a 8Kx8K (64 million elements, 256MB) matrix for 1,000 – 8,000 iterations. Due to overhead of the boundary handling in the SHOC MPI implementation, we ran only the non-MPI (single-node) SHOC implementation, which does not include buffer copies over the network nor boundary exchanges. We then ran the MGP implementation

with 2, 4 and 8 stripes, where each stripe ran on a different node, including the local node, and boundary exchanges were conducted by scatter-gather. The results are presented in Fig. 5.

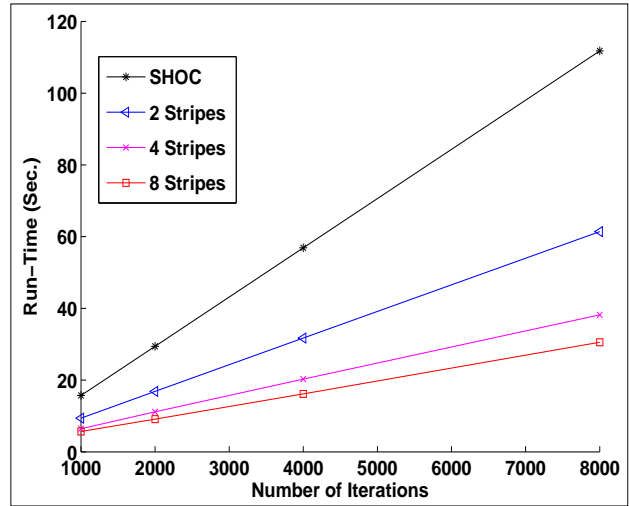


Fig. 5. Stencil2D 8Kx8K Matrix Run-Times on a Cluster

From Fig. 5 it can be seen that MGP can efficiently handle a large number of kernels, with a fixed run-time per kernel for each number of stripes, regardless of the number of iterations. Also, from the obtained results it follows that the MGP speedups for 2, 4 and 8 stripes vs. the single-node SHOC, gradually increase from 1.43, 2.09 and 2.37 respectively for 1000 iterations to 1.66, 2.67 and 3.34 respectively for 8000 iterations.

In a followup test, we ran the same test on a 16Kx16K (256 million elements, 1GB) matrix, where the memory allocation for the source and destination matrices exceeded the available memory in any one device, so the test could not be performed on a single device. The obtained results showed the same patterns as in Fig. 5.

#### IV. RELATED WORK

A survey of parallel programming models, including OpenMP, CUDA and MPI, was presented by Kasim et al. [14]. MGP is an extension of the OpenMP programming model, which allows running of GPU code in a simple, OpenMP-like manner. Another set of extensions to OpenMP were proposed in OpenMPD [15], a directive-based data-parallel extension for distributed memory systems.

Running OpenCL programs on clusters was done by CLuMPI [16] by using MPI as a networking back-end. CLuMPI runs unmodified OpenCL programs on a cluster (CPU only), turning each cluster node into a device. It is also able to spread OpenCL work-groups of one kernel to an entire cluster. Running CUDA programs on clusters was done by CUDASA [17] and rCUDA [18].

To test MGP, we used the Scalable Heterogeneous Computing (SHOC) benchmark suite [3], which tests the stability

and performance of GPU devices and other architectures using OpenCL.

GPU clusters for HPC are now increasingly being used, most are based on MPI and run CUDA applications on local devices [4]. Others, such as MITHRA [19] uses Hadoop [20] to distribute CUDA applications. Mixed MPI/OpenMP cluster packages, such as TSubame [5], use MPI to distribute tasks among different nodes, then OpenMP within each node.

An outline of parallel programming models for hybrid MPI/OpenMP on clusters was presented by Rabenseifner et al. [21].

## V. CONCLUSIONS AND FUTURE WORK

Advancements in GPGPU processors offer new opportunities to increase the performance of parallel applications on large-scale clusters. Users are provided with programming environments and SDKs that can ease the use of GPU devices on a single node, but were not specifically designed to run parallel applications on clusters.

The paper presented the Many GPUs Package (MGP) that allows parallel OpenMP, C++ and unmodified OpenCL applications to transparently run on many devices in a cluster. MGP combines the ease of OpenMP programming with task level services such as queuing, scheduling a scatter-gather, for efficient utilization of cluster-wide GPU devices (and possibly also of OpenCL enabled CPUs). Evaluating the performance of parallel applications with MGP shows that running parallel kernels efficiently on remote devices in a cluster is quite feasible. MGP should be able to support large-scale high-end parallel computing applications.

Based on our experience, an ideal cluster for running massive parallel and HPC applications with MGP would be a collection of nodes, each with several GPUs, connected by a low-latency, high-speed network. The hosting nodes should be high-end with a large memory.

The work described in this paper could be extended in several directions. First, it would be interesting to include the extended OpenMP API in existing compilers. It would also be interesting to add support for scatter-gather in the OpenMP standard itself.

For better support of GPU computing on clusters, it is important to develop MOSIX-like algorithms for dynamic resource management, load-balancing, task priorities, fair-share and a method for choosing the “best” device [10], [22]. It is also important to add network optimizations, such as support for direct exchange of boundaries among different devices in the scatter-gather mechanism.

Followup projects that we are considering include the use of Map/Reduce [23] to extend our API; porting of MGP to CUDA [7] and DirectCompute [13] and upgrade to the latest OpenCL specifications.

MGP is currently implemented for Linux platforms. A production version will be incorporated with the MOSIX cluster management distribution [11].

## ACKNOWLEDGMENT

The authors would like to thank Eri Rubin for his professional advice and our system group for their excellent technical assistance. We thank the ORNL Future Technologies Group for providing a copy of the SHOC benchmark suite and Intel-Israel (74) Ltd. for the equipment donation.

## REFERENCES

- [1] “The OpenMP API specification for parallel programming.” [Online]. Available: <http://www.OpenMP.org>
- [2] “The Message Passing Interface (MPI) standard.” [Online]. Available: <http://www.mcs.anl.gov/research/projects/mpl/>
- [3] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, “The Scalable Heterogeneous Computing (SHOC) benchmark suite,” in *Proc. 3-rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*, March 2010, pp. 63–74.
- [4] V. V. Kindratenko, J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. mei W. Hwu, “GPU clusters for high-performance computing,” in *Proc. Workshop on Parallel Programming on Accelerator Clusters, IEEE Cluster 2009*, Aug. 2009, pp. 1–8.
- [5] *TSubame User’s Guide*, Global Scientific Information and Computing Center, Tokyo Institute of Technology, 2009. [Online]. Available: <http://www.gsic.titech.ac.jp/ccwww/tebiki/tebiki-eng.pdf>
- [6] OpenCL, *The OpenCL Specification*, A. Munshi (Ed.). Khronos Group, 2010. [Online]. Available: <http://www.khronos.org/opencl>
- [7] “CUDA.” [Online]. Available: <http://www.nvidia.com/cuda>
- [8] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.
- [9] M. Hess, G. Jost, M. S. Müller, and R. Rühle, “Experiences using OpenMP based on compiler directed software DSM on a PC cluster,” in *Proc. Int. Workshop on OpenMP Applications and Tools (WOMPAT 2002)*, ser. LNCS Vol. 2716. Springer, 2002, pp. 211–226.
- [10] A. Barak and A. Shiloh, *The MOSIX Management System for Linux Clusters, Multi-Clusters and Clouds*, 2010. [Online]. Available: [http://www.MOSIX.org/pub/MOSIX\\_wp.pdf](http://www.MOSIX.org/pub/MOSIX_wp.pdf)
- [11] “MOSIX.” [Online]. Available: <http://www.MOSIX.org>
- [12] E. Ayguadé, R. M. Badia, D. Cabrera, A. Duran, M. González, F. D. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, J. M. Pérez, and E. S. Quintana-Ortíz, “A proposal to extend the OpenMP tasking model for heterogeneous architectures,” in *Proc. 5-th Int. Workshop on OpenMP (IWOMP 2009)*, ser. LNCS Vol. 5568, M.S. Müller et al. (Eds.). Springer-Verlag, 2009, pp. 154–167.
- [13] “Directcompute.” [Online]. Available: <http://msdn.microsoft.com/directx>
- [14] H. Kasim, V. March, R. Zhang, and S. See, “Survey on parallel programming model,” in *Proc. IFIP Int. Conference on Network and Parallel Computing (NPC ’08)*, ser. LNCS Vol. 5245. Springer-Verlag, 2008, pp. 266–275.
- [15] J. Lee, M. Sato, and T. Boku, “OpenMPD: A directive-based data parallel language extension for distributed memory systems,” in *Proc. 2008 Int. Conference on Parallel Processing Workshops*. IEEE Computer Society, Sept. 2008, pp. 121–128.
- [16] “CLuMPI.” [Online]. Available: <http://clumpi.sourceforge.net>
- [17] C. Muller, S. Frey, M. Strengert, C. Dachsbacher, and T. Ertl, “A compute unified system architecture for graphics clusters incorporating data locality,” *IEEE Tran. on Visualization and Computer Graphics*, vol. 15, pp. 605–617, 2009.
- [18] “rCUDA.” [Online]. Available: <http://www.hpca.uji.es/?q=node/36>
- [19] R. Farivar, A. Verma, M. Chan, and R. H. Campbell, “MITHRA: Multiple data independent tasks on a heterogeneous resource architecture,” in *Proc. IEEE Cluster 2009*, Aug. 2009.
- [20] “Apache Hadoop.” [Online]. Available: <http://hadoop.apache.org>
- [21] R. Rabenseifner, G. Hager, and G. Jost, “Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes,” in *Proc. Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, Feb. 2009, pp. 427–436.
- [22] L. Amar, A. Barak, E. Levy, and M. Okun, “An on-line algorithm for fair-share node allocations in a cluster,” in *Proc. 7-th IEEE Int. Symp. on Cluster Computing and the Grid (CCGrid’07)*, May 2007, pp. 83–91.
- [23] J. Dean and S. Ghemawat, “Map/Reduce: A flexible data processing tool,” *Communications ACM*, vol. 53, no. 1, pp. 72–77, 2010.