Special Issue of International Journal of Computer Applications (0975 – 8887)
on Advanced Computing and Communication Technologies for HPC Applications - ACCTHPCA, June 2012

Distributed OpenCL Distributing OpenCL Platform on Network Scale

Barış Eskikaya
Department of Computer Engineering
Istanbul Technical University
Istanbul, Turkey

D Turgay Altılar
Department of Computer Engineering
Istanbul Technical University
Istanbul, Turkey

ABSTRACT

This paper presents a framework that extends OpenCL by distributing computing process to many computing resources connected via network and enables the computing resources to run in parallel. Using JSON RPC (Remote Procedure Call technique relying on JavaScript Object Notation) in communication layer, Distributed OpenCL framework provides platform and operating system independency. Using this framework, a host program executed on a computer that has no OpenCL support is able to use other computing resources distributed on network in parallel. Results show that OpenCL platform model can be extended to network scale to provide a vendor, architecture and operating system independent and a parallel computing environment with reasonable communication overhead.

General Terms

Distributed computing, GPU clusters

Keywords

OpenCL, GPU, GPGPU, HPC, GPU clusters

1. INTRODUCTION

Since the architecture of Graphics Processing Units (GPU) uses pipeline model, GPUs are convenient for processing massive data using SIMD (Single Instruction Multiple Data) technique which provides data-level parallelism. Although primitive GPUs were not programmable and only used for graphics rendering, with the improvements in GPU technology, modern GPUs are evolved to provide programmable interfaces and this improvement has brought General Purpose programming on GPUs (GPGPU) concept in use. With extensive use of this concept GPUs have been a part of main computing resources of a computer along with CPUs and multi-core CPUs.

As a result of widespread use of GPGPU concept, GPU manufacturers have introduced GPGPU programming languages such as C for Graphics, Close to Metal (from AMD/ATI), CUDA (from NVIDIA) [1], BrookGPU (from Stanford University) [2] and DirectCompute (from Microsoft) [3]. However these programming languages were platform and vendor specific so they were not supporting the use of the programs in multi vendor environments and portability between devices. These languages also needed developers to be familiar with specific features of language and architectures of devices that they aim to program.

To overcome the problems above, OpenCL framework has been introduced by Khronos Group [4] with contribution and collaboration of leading manufacturers such as Apple, IBM, Intel, AMD and NVIDIA. OpenCL framework consists of a C based language and a universal API set that enables to write and execute programs between heterogeneous platforms such as GPUs, CPUs and other processors. Since OpenCL standards are being supported and accepted as common standards by CPU and GPU vendors, it provides compatibility and portability between vendors. Although different vendors have different implementations of OpenCL, they comply with the same standards for data types, method signatures and API set.

The features described above make OpenCL a convenient programming framework for heterogeneous environment and abstract technical details of vendor specific implementations and devices architectures from developers. Considering the advantages and properties of OpenCL, we claim it is possible to extend the benefits of OpenCL by designing a system which consists of distributed computing resources and hosts that use these computing resources connected via network.

In this study we propose a framework that extends OpenCL on network scale using JSON RPC communication technique between hosts and computing resources. Since the framework is based on OpenCL, it covers the advantages of OpenCL such as heterogeneous resource usage, vendor and architecture independency, moreover it increases the level of parallelism by increasing number of computing resources and by using JSON RPC for communication it enables operating system and platform independency in communication layer.

The rest of the paper is organized as follows. In Section II we present an overview of OpenCL platform and JSON RPC, in Section III we explain the design and implementation of Distributed OpenCL, in Section IV we show the experiments and test results for our study and comparison to other similar studies on this field. In section V we present our conclusions.

2. OPENCL & JSON RPC OVERVIEW

2.1 OpenCL Overview

OpenCL (Open Computing Language) is a framework for writing programs which execute across heterogeneous platforms consisting of CPUs, GPUs and other processing units. OpenCL includes a C based language (based on C99) to write functions that execute on devices which supports OpenCL standards and an API to control these platforms. Target of OpenCL is to let the developers write portable, vendor and device independent programs and abstract them from complex technical specifications and detailed architectures of computing devices. In our study we followed OpenCL v1.1 specification [5].

OpenCL structure can be described as combination of the following models:

- Platform Model

- Execution Model
- Memory Model
- Programming Model

2.2 OpenCL Platform Model

The Platform Model consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more computing units. Each computing unit is divided into one or more processing elements. Processing elements are where the computation is done. An OpenCL device can be a GPU, CPU or another type of processor. OpenCL systems are identified with version of the platform, version of devices and version of OpenCL C language supported on the devices. OpenCL application running on a host submits commands to devices to run on the processing elements on the devices. Processing elements execute single stream of instructions as SIMD units or SPMD (Single Program Multiple Data) units depending on the type of processing element.

2.3 OpenCL Execution Model

There are two parts for execution of an OpenCL program: Kernels and host program. The host program defines context for the kernels and manage their execution, and the kernels execute one or more OpenCL devices. When a kernel is submitted for execution by the host, an index space is defined. For each point in the index space an instance of kernel is created. These kernel instances are called work-items. Each work item has a global ID which is given according to position of corresponding point in the index space. Work-items are organized into work groups. Processing elements in a computing unit can concurrently execute work-items in a work group. With this execution model, OpenCL supports data-parallel and task-parallel programming models.

2.4 OpenCL Memory Model

The host application creates memory objects on global memory using OpenCL API and enqueue memory commands operating on these memory objects. Memory blocks of host application and computing devices usually run independent. In case of interaction is needed between host application memory and computing device memory, either transferring of memory blocks occur or host application uses mapping / unmapping methods to reach memory blocks of computing devices.

2.5 JSON RPC Overview

JSON RPC is a remote procedure call protocol which sends RPC commands between server and client by encoding data into JSON format [6]. In JSON encoding format data is converted to UTF-8 JSON strings and transferred via HTTP or TCP/IP transfer protocols. In Distributed OpenCL framework, TCP/IP communication protocol is used since it is faster than HTTP protocol. Open source JSONRPC-CPP framework is used for client-server communication. Some modifications are done on the framework to decrease message size and to enable splitting large TCP messages at TCP client, which's size are greater than maximum allowed TCP package size, and remerging them at TCP server. To avoid TCP connection starting cost for each API call, TCP connection between client and server is opened at the first OpenCL API call from the host application, kept alive during the execution of program and closed at the end of the program. Native OpenCL types are converted into native C++ types and encoded to JSON format (see Table 1).

Table 1. Conversion for OpenCL data types into JSON encodable C++ data types

OpenCL Type	Equivalent C++ type	Encoding to reduce data size
Integer types eg: cl_int, cl_uint, cl_bool, cl_ulong	int, unsigned int, long unsigned int	-
Pointer types eg: cl_platform_id, cl_device_id, cl_context, cl_mem, cl_program, cl_kernel	Pointer converted to unsigned long	-
Data parameters eg: float array, integer array, char array	Converted to byte array from void pointer	Encoded as string using base64 encoding

Compared to similar RPC technologies like Microsoft RPC, XML RPC, Java RMI; JSON RPC is more suitable to use for our framework since it provides platform independency, and reasonable message size.

3. DISTRIBUTED OPENCL

3.1 Distributed OpenCL Overview

Distributed OpenCL is a framework that extends OpenCL on network scale by preserving OpenCL functionality and advantages described in the above sections. Distributed OpenCL framework has a client-server architecture that can consist of multiple clients and multiple servers. Distributed OpenCL framework consists of the following components:

- Clients
- Distribution Layer
- Servers

Clients run OpenCL host applications and OpenCL API calls from host applications are redirected to servers on the network by distribution layer running on the client machines. OpenCL functionality is provided on servers. Figure 1 represents the architecture of Distributed OpenCL framework.

When the OpenCL host application running on the client makes an API call, this call is sent to the distribution layer running on the client. Distribution layer converts API calls to JSON message packages by marshalling. These JSON messages are sent to the servers registered in the system using JSON RPC communication technique. JSON RPC server application on the server machines listens to the incoming JSON messages and converts these messages to OpenCL API calls by unmarshalling. With this method, OpenCL API calls are replicated on the server with the same parameters from the client. Server converts responses from OpenCL computing devices to JSON messages and send back result to client which made the API call. Since OpenCL computing devices keep the context and state data of program in their own memory, servers do not keep any context or state data of client programs. Pointers to the instances of OpenCL types such as *cl_mem*, *cl_context*, *cl_program*, *cl_kernel* etc. that are created by OpenCL API calls are allocated in heap section of server memory to provide access by pointer addresses of them to use in following OpenCL API calls. These allocated spaces are freed when release functions such as *clReleaseMemObject*, *clReleaseContext* etc. are called by the corresponding host application.

Considering GPU architecture performs operations in SIMD model, data which will be processed can be distributed to multiple servers registered in the system. This feature reduces the workload of one server so reduces the workload of each processing unit in the server and results speedup for calculation by increasing level of parallelism.

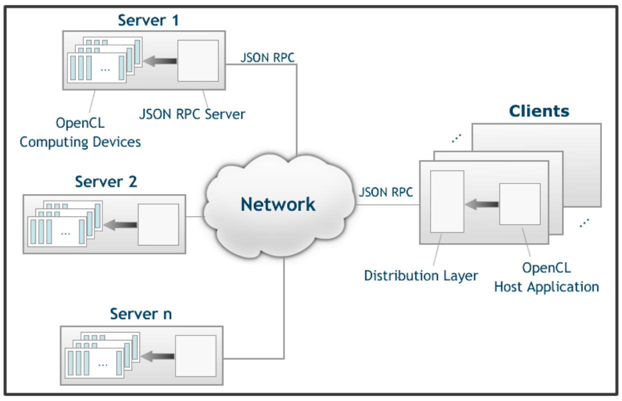

Figure 1. Distributed OpenCL framework architecture

3.2 Client

Client refers to original OpenCL host applications running on a client machine. These host applications can be compiled to use with Distributed OpenCL framework just by replacing original OpenCL include directory reference to Distributed OpenCL distribution layer include directory reference and OpenCL library reference to distribution layer library reference to use in linking step. After compilation, OpenCL API calls are handled by our distribution library instead of original OpenCL implementation.

3.3 Distribution Layer

Distribution layer is the Distributed OpenCL library running on the client. The library consists of copies of OpenCL header files that contains OpenCL type declarations and function signatures, and C++ source file which simulates function bodies for OpenCL functions declared in OpenCL v1.1 specification. OpenCL API calls are handled by our distribution library instead of original OpenCL implementation. For each API call, distribution library executes corresponding functions and these functions convert and combine parameter values into JSON messages adding function name to call. After this step, JSON messages are sent to server by TCP/IP based JSON RPC client running on the distribution layer. Functions in distribution layer also convert returned JSON messages from server into OpenCL types and send results back to the host application. Type conversion operations, binary data encoding for function parameters and binary data decoding for function results are done by the distribution library.

3.4 Server

JSON RPC server application running on the servers listens to the corresponding ports to handle incoming JSON messages. Corresponding handle method for the incoming JSON messages are called by JSON RPC Server application and parameters in the JSON message are unmarshalled into OpenCL data types and corresponding OpenCL API function which is declared in the JSON message is called via original OpenCL API with parameter values in the message. Type conversion from JSON values to OpenCL types and binary data decoding for the data parameters in the JSON message are done at this step. OpenCL computations are done by computing devices (GPUs or CPUs) on the server. Return values from the OpenCL API are marshaled into JSON format by type conversions and binary data encoding if exist, then result JSON message is sent back to the client. Results from the servers are gathered, combined, unmarshalled in the client by distribution layer and returned to host application. TCP connection from client to the server is established at the first call to the server and kept open during the execution of host program to improve the performance.

Each server in the system can serve concurrently to multiple clients or multiple host programs since OpenCL API keeps context and state information. Pointers to the OpenCL types like cl_mem, cl_context etc. are kept in hash maps in the server application memory matched with corresponding host application IDs.

3.5 Providing Parallel Execution

Since GPU performs operations in SIMD model, many OpenCL applications can be parallelized just by partitioning input data into blocks. For many problems such as convolution, edge detection etc., applications can also be parallelized by partitioning input data into overlapping blocks. By modifying host applications to run with divided partitions of input data, Distributed OpenCL framework provides parallel execution. Figure 2 shows parallel execution in Distributed OpenCL framework. After partitioning input data, for each partition, host applications can be run in a multi-thread architecture and results of computations can be gathered using shared memory or similar parallel data distribution systems.

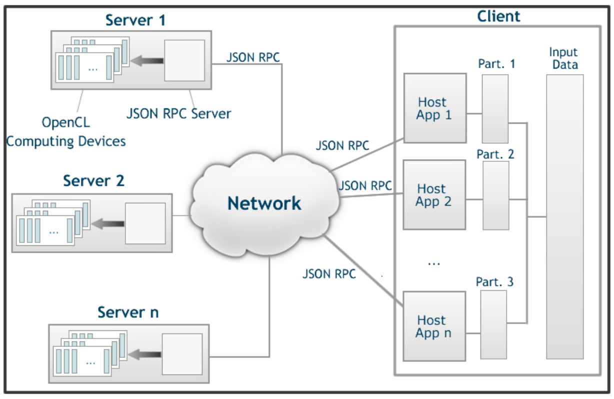

Figure 2. Distributed OpenCL parallel execution

4. EXPERIMENTS

To measure performance of Distributed OpenCL Framework, two sets of experiments are performed. As the first set of experiments (Distributed OpenCL Overhead Experiments) we compared execution times between native OpenCL (NCL), Distributed OpenCL with client program and server program running on the same machine (DCL Local), client program and server program running on different machines (DCL Remote). As the second set of experiments (Parallel Execution Experiments) we compared execution times for client program running on one remote server; two remote servers and four remote servers by partitioning input data and running host applications in parallel. We used three different host applications executing three different kernel functions: vector addition, 2D convolution and N-Body simulation.

During the experiments we have seen that transferred data size is the main determinative factor for performance difference between native OpenCL and our system. Therefore the results we present belong to the experiments with "vector addition" application since it sufficiently represents the characteristic differences between two systems. Experiment results are presented and compared considering execution times by excluding OpenCL kernel function dynamic compilation time. Tests are performed for various work item sizes from 32 to 1M. Since the test application adds two float arrays and returns a result float array, data transfer size in bytes is calculated with formula "(1)"

D.T. = 3 x sizeof(float) x Work Item Size (1)

D.T. = 12 x Work Item Size

Each test was run 10 times, median of execution times are presented as the results. For remote tests, client and servers are connected with 100 Mbit Ethernet (LAN). Specifications for the computers that we used in experiments can be seen in Table 2. Detailed specifications for the GPUs can be obtained from web sites of vendors [7] [8].

Table 2. Test Computers

Comp ID	OS	CPU	RAM	GPU
A	Windows 7 64 bit	Intel Core i7	4 GB DDR 3	NVIDIA 525M
B	Windows 7 64 bit	Intel Core i7	4 GB DDR 3	ATI RADEON HD 6370M
C	Ubuntu Linux 64 bit	Intel Core 2 Duo	4 GB DDR 3	NVIDIA 240M
D	Ubuntu Linux 64 bit	Intel Core i7	4 GB DDR 3	NVIDIA 540M
E	Windows 7 64 bit	Intel Core i7	8 GB DDR 3	NVIDIA 555M
F	Windows 7 64 bit	Intel Core i5	3 GB DDR 3	NVIDIA 520M

4.1 Distributed OpenCL Overhead Experiments

Overhead experiments are done to compare performance of native OpenCL and Distributed OpenCL framework.

- Test 1: Test application is run with native OpenCL on the computers. Results are presented in Table 3 for comp. A, B and C (NCL).
- Test 2: Test application is run with Distributed OpenCL on the computers as client and server application running on the same machine. Results are presented in Table 3 for comp. A, C, and D (DCL Local).
- Test 3: Test application is run with Distributed OpenCL on the computers as client and server application running on different machines. Tests are performed with combinations of two machines. Results are presented in Table 3 for comp. C (client) to A (server), A (client) to E (server), A (client) to C (server) and C (client) to D (server) (DCL Remote).

4.2 Parallel Execution Experiments

Parallel execution experiments are done to measure Distributed OpenCL framework performance when the work load is distributed to more than one computing nodes running parallel.

- Test 1: Test application is run with Distributed OpenCL on combinations of two machines as one client and one server. Results are presented in Table 4 for comp. C (client) to A (server), A (client) to E (server) (DCL Remote 1 Comp).
- Test 2: Test application is run with Distributed OpenCL on combinations of three machines as one client and two servers. Results are presented in Table 4 for comp. C (client) to A and E (servers), A (client) to D and E (servers) (DCL Remote 2 Comp).
- Test 3: Test application is run with Distributed OpenCL on combinations of five machines as one client and four servers. Results are presented in Table 4 for comp. C (client) to A, B, D and E (servers); F (client) to A, B, D and E (servers) (DCL Remote 4 Comp).

4.3 Evaluation of Results

In the first set of experiments (see Table 3), for Test 1 we made the following inferences:

- Execution time does not increase linearly or one-to-one by the increase in work item size since GPU architecture executes the work items in parallel.
- Native OpenCL performance between NVIDIA GPU (NCL A) and an equivalent ATI GPU (NCL B) does not differ too much.
- However for an earlier model NVIDIA GPU (NCL C), native OpenCL execution time may differ dramatically when we increase work item size since number of cores and buffer size is smaller.

In Test 2 it can be seen:

- When we compare DCL Local to the NCL, execution time increases because of communication overhead. Communication overhead is especially effective for OpenCL functions such as *clEnqueueWriteBuffer*, *clEnqueueReadBuffer*, *clCreateBuffer*, which send or receive data between host application and GPU device. Communication overhead increases near linear by the increase in data transfer size at large data sizes.
- Depending on the operating system configurations, TCP overhead increases faster after a certain data transfer size because data cannot be transferred in one TCP package.
- For smaller data sizes sending data from Linux operating systems to Linux or Windows operating systems is faster because of operating system characteristics. As the transfer size increases, this major difference cannot be seen.

Table 3. Results for the first experiment set (HOST ID: ID of client computer, GPU ID: ID of server computer, D.T.: Data Transfer size, E.T.: Execution Time)

				NCL	NCL	NCL	DCL Local	DCL Local	DCL Local	DCL Remote	DCL Remote	DCL Remote	DCL Remote
			Test #	**T 1**	**T 1**	**T 1**	**T 2**	**T 2**	**T 2**	**T 3**	**T 3**	**T 3**	**T 3**
			HOST ID							**C**	**A**	**A**	**C**
			GPU ID	**A**	**B**	**C**	**A**	**C**	**D**	**A**	**E**	**C**	**D**
Work Item Size	**D.T. (B)**	**D.T. (KB)**	**D.T. (MB)**	**E.T. (ms)**	**E.T. (ms)**	**E.T. (ms)**	**E.T. (ms)**	**E.T. (ms)**	**E.T. (ms)**	**E.T. (ms)**	**E.T. (ms)**	**E.T. (ms)**	**E.T. (ms)**
32	384	0	0	2	4	2	22	2	2	8	46	44	8
64	768	1	0	2	4	2	22	3	3	8	44	45	8
128	1536	2	0	3	5	2	23	3	3	8	44	46	9
256	3072	3	0	5	8	2	25	4	4	11	49	48	12
1024	12288	12	0	6	9	3	23	13	9	15	52	50	15
4096	49152	48	0	9	11	12	50	48	32	52	53	52	49
16384	196608	192	0	10	13	44	142	156	104	164	111	158	125
65536	786432	768	1	11	16	170	325	355	298	330	325	423	328
262144	3145728	3072	3	14	18	702	507	986	504	896	892	1325	878
524288	6291456	6144	6	15	19	1356	979	2016	965	1572	1746	2896	1753
1048576	12582912	12288	12	16	22	2749	1941	3457	1906	3167	3309	5268	3468

Table 4. Results for the second experiment set (HOST ID: ID of client computer, GPU IDs: ID of server computers, D.T.: Data Transfer size, E.T.: Execution Time)

				DCL Remote 1 Comp	DCL Remote 1 Comp	DCL Remote 2 Comp	DCL Remote 2 Comp	DCL Remote 4 Comp	DCL Remote 4 Comp
			Test #	**Test 1**	**Test 1**	**Test 2**	**Test 2**	**Test 3**	**Test 3**
			HOST ID	**C**	**A**	**C**	**A**	**C**	**F**
			GPU IDs	**A**	**E**	**A - E**	**D - E**	**A - B - D - E**	**A - B - D - E**
Work Item Size	**D.T. (B)**	**D.T. (KB)**	**D.T. (MB)**	**E.T. (ms)**	**E.T. (ms)**	**E.T. (ms)**	**E.T. (ms)**	**E.T. (ms)**	**E.T. (ms)**
32	384	0	0	8	46	9	44	8	42
64	768	1	0	8	44	9	44	9	43
128	1536	2	0	8	44	9	46	9	44
256	3072	3	0	11	49	11	46	9	45
1024	12288	12	0	15	52	14	48	10	47
4096	49152	48	0	52	53	29	53	16	52
16384	196608	192	0	164	111	89	75	54	52
65536	786432	768	1	330	325	181	182	165	123
262144	3145728	3072	3	896	892	452	448	334	330
524288	6291456	6144	6	1572	1746	833	854	487	489
1048576	12582912	12288	12	3167	3309	1637	1622	923	918

- While communication overhead is independent from the GPU models, impact of communication overhead to the total execution time may change since execution time for the OpenCL functions on the server machine changes (NCL A vs DCL Local A, NCL C vs, DCL Local C).

In Test 3 we have seen that when we compare DCL Local to DCL Remote, execution times increases to 1.5 to 2 times with 100 Mbit Ethernet connection.

The second set of experiments (see Table 4) shows us distributing workload to multiple computing nodes running parallel increases performance. In Figure 3 it can also be seen, execution times decreases almost by the same ratio with the increase in number of computing nodes. For very small data sizes execution time does not change by increasing number of nodes because in this range rather than OpenCL input - output data transfer operations, other OpenCL function calls are also effective on execution time.

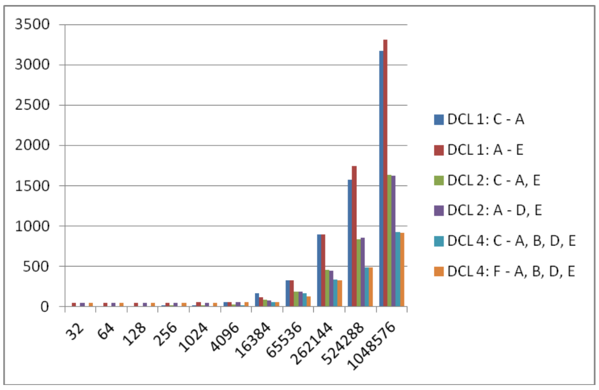

Figure 3. Distributed OpenCL parallel execution

4.4 Comparison to the Previous Studies

During our research we have come across several studies such as Many GPUs Package (MPG) [9], Virtual OpenCL Cluster Platform (VCL) [10], Remote CUDA (rCUDA) [11], Hybrid OpenCL [12], CLuMPI [13] and GPU Clusters for High-Performance Computing [14] on distributed GPGPU computing. These implementations are either vendor specific or operating system dependent. In our study, we have designed and implemented a both vendor independent and operating system independent framework by using OpenCL as GPGPU computing platform and TCP / IP protocol based JSON RPC communication technique.

5. CONCLUSIONS

OpenCL exposes a framework with a universal API set that enables to write and execute programs between heterogeneous platforms such as GPUs, CPUs and other processors. It also provides platform – vendor independency and portability. By increasing acceptance of OpenCL standards, and also contribution and collaboration of leading manufacturers its usage also increases day by day compared to other GPGPU languages / frameworks.

Considering features and architecture of OpenCL it also allows distribution of computing nodes on network scale. In this study we presented a framework that consists of clients where host applications run, servers where computing devices run and communication between them is supplied using JSON RPC technique. By this approach we extend OpenCL on network scale and improve level of parallelism by increasing number of computing nodes running parallel. As we have seen in the experiments, especially by using multi GPU servers, low latency – high bandwidth networks and configuring maximum allowed TCP package size to higher rates it would be possible to acquire a vendor - operating system independent, parallel and scalable GPU computing platform resulting speed up in overall computing performance.

6. REFERENCES

[1] "CUDA." [Online]. Available: http://www.nvidia.com/cuda

[2] "BrookGPU" [Online]. Available: http://graphics.stanford.edu/projects/brookgpu/

[3] "Directcompute." [Online]. Available: http://msdn.microsoft.com/directx

[4] "OpenCL - The open standard for parallel programming of heterogeneous systems" [Online]. Available: http://www.khronos.org/opencl/

[5] "The OpenCL Specification" [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[6] "JsonRpc-Cpp - OpenSource JSON-RPC implementation for C++" [Online]. Available: http://jsonrpc-cpp.sourceforge.net/

[7] "NVIDIA GPU Models" [Online]. Available: http://www.geforce.com/hardware/notebook-gpus

[8] "ATI GPU Models" [Online]. Available: http://www.amd.com/us/products/notebook/graphics/Pages/notebook-graphics.aspx

[9] 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 International Conference on Cluster Computing.

[10] "The Virtual OpenCL (VCL) Cluster Platform" [Online]. Available: http://www.mosix.org/vcl/VCL_wp.pdf

[11] "rCUDA." [Online]. Available: http://www.rcuda.net/.

[12] Ryo Aoki, Shuichi Oikava, Ryoji Tsuchiyama, Takashi Nakamura "Improving Hybrid OpenCL Performance by High Speed Networks" 2010 First IEEE international Conference on Networking and Computing

[13] "CLuMPI." [Online]. Available: http://clumpi.sourceforge.net

[14] V. Kindratenko et al. "GPU Clusters for High-Performance Computing"