CUDA-For-Clusters: A System for Efficient Execution of CUDA Kernels on Multi-core Clusters

Raghu Prabhakar[1,⋆], R. Govindarajan[2], and Matthew J. Thazhuthaveetil[2]

[1] University of California, Los Angeles
raghu@cs.ucla.edu
[2] Supercomputer Education and Research Centre,
Indian Institute of Science, Bangalore, India
{govind,mjt}@serc.iisc.ernet.in

Abstract. Rapid advancements in multi-core processor architectures coupled with low-cost, low-latency, high-bandwidth interconnects have made clusters of multi-core machines a common computing resource. Unfortunately, writing good parallel programs that efficiently utilize all the resources in such a cluster is still a major challenge. Various programming languages have been proposed as a solution to this problem, but are yet to be adopted widely to run performance-critical code mainly due to the relatively immature software framework and the effort involved in re-writing existing code in the new language. In this paper, we motivate and describe our initial study in exploring CUDA as a programming language for a cluster of multi-cores. We develop CUDA-For-Clusters (CFC), a framework that transparently orchestrates execution of CUDA kernels on a cluster of multi-core machines. The well-structured nature of a CUDA kernel, the growing popularity, support and stability of the CUDA software stack collectively make CUDA a good candidate to be considered as a programming language for a cluster. CFC uses a mixture of source-to-source compiler transformations, a work distribution runtime and a light-weight software distributed shared memory to manage parallel executions. Initial results on running several standard CUDA benchmark programs achieve impressive speedups of up to 7.5X on a cluster with 8 nodes, thereby opening up an interesting direction of research for further investigation.

Keywords: CUDA, Multi-Cores, Distributed Programming, Distributed Systems, Clusters, Software Distributed Shared Memory.

1 Introduction

Clusters of multi-core nodes have become a common HPC resource due to their scalability and attractive performance/cost ratio. Such compute clusters typically have a hierarchical design with nodes containing shared-memory multi-core

⋆ The author was affiliated with the Indian Institute of Science during this work.

processors interconnected via a network infrastructure. While they provide an enormous amount of computing power, writing parallel programs to efficiently utilize all the cluster resources remains a daunting task. For example, intra-node communication between tasks scheduled on a single node is much faster than inter-node communication, hence it is desirable to structure code in a way so that most of the communication takes place locally. Interconnect networks have large bandwidth and are suitable for heavy, bursty data transfers. This task of manually orchestrating the execution of parallel tasks efficiently and managing multiple levels of parallelism is difficult. A popular programming choice is a hybrid approach [10] using multiple programming models like OpenMP[5] (intra-node) and MPI[20] (inter-node) to explicitly manage locality and parallelism. The challenge lies in writing parallel programs that can readily scale across systems with steadily increasing numbers of both cores per node and nodes in the cluster. Various programming languages and models that have been proposed as a solution to this problem ([11], [12] etc.,) are yet to be adopted widely due to the effort involved in porting applications to the new language as well as the constantly changing software stack supporting the languages.

GPGPU computation has attracted the attention of software developers and researchers off-late, and has been facilitated mainly by NVIDIA's CUDA [3] and OpenCL [4]. In particular, CUDA has become a popular language as evident from an increasing number of users [3] and benchmarks [6] [13]. However, CUDA is a shared memory programming model designed in tandem with the CUDA architecture which consists of homogeneous cores. Therefore intuitively, CUDA does not seem to fit the bill to program distributed machines. However, the semantics of CUDA enforce a structure on parallel kernels where communication between parallel *threads* is guaranteed to take place correctly only if the communicating threads are part of the same *thread block*, through some block-level *shared memory*. From a CUDA *thread*'s perspective, the *global* memory offers a relaxed consistency that guarantees coherence only across kernel invocations, and hence no communication can reliably take place through global memory within a kernel invocation. Such a structure naturally exposes data locality information that can readily benefit from the multiple levels of hardware-managed caches found in conventional CPUs. In fact, previous works such as [21] and [22] have shown the effectiveness using CUDA to program multi-core shared memory CPUs, and similar research has been performed on OpenCL as well [16]. There has also been some recent work on using OpenCL to program heterogeneous CPU/GPU clusters [17]. More recently, a compiler that implements CUDA on multi-core x86 processors has been released commercially by the Portland Group [7]. CUDA has evolved into a very mature software stack with efficient supporting tools like debuggers and profilers, making application development and deployment easy.

Considering the factors of programmability, popularity, scalability, support and expressiveness, we believe that CUDA can be used as a single language to efficiently program a cluster of multi-core machines. From a utility perspective, establishing an execution flow from CUDA to a distributed system would immediately enable many CUDA programs to achieve speedups on commodity cluster machines. In this paper, we explore this idea and describe CFC, a framework to execute CUDA kernels can be efficiently and in a scalable fashion on a cluster of multi-core machines. As the thread-level specification of a CUDA kernel is too fine grained to be profitably executed on a CPU, we employ compiler techniques described in [22] to serialize threads within a block and transform the kernel code into a *block-level* specification. The independence and granularity of thread blocks makes them an attractive schedulable unit on a CPU core. As global memory in CUDA provides only a relaxed consistency, we show it can be realized by a lightweight software distributed shared memory (DSM) that provides an abstraction of a single shared address space across the compute cluster nodes. Finally, we describe our work-partitioning runtime that distributes thread blocks across all cores in the cluster. We evaluate our framework using several standard CUDA benchmark programs from the Parboil benchmark suite [6] and the NVIDIA CUDA SDK [2] on a compute cluster with eight nodes. We achieve promising speedups ranging from 3.7X to 7.5X compared to a baseline multi-threaded execution (around 56X compared to a sequential execution). We claim that CUDA can be successfully and efficiently used to program a compute cluster and thus motivate further exploration in this area.

The rest of this paper is organized as follows: Section 2 provides the necessary background. In Section 3, we describe the CFC framework in detail. In Section 4, we describe our experimental setup and evaluate our framework. Section 5 discusses related work. In section 6 we discuss possible future directions and conclude.

2 Background

2.1 CUDA Programming Model

The CUDA programming model provides a set of extensions to the C programming language enabling programmers to execute functions on a GPU. Such functions are called *kernels*. Each kernel is executed on the GPU as a *grid* of *thread blocks*. The grid size and block size are specified by the programmer during invocation. Data transfer between the main memory and GPU DRAM is performed explicitly using CUDA APIs. Each block is scheduled to execute on one *streaming multiprocessor* (SM) on the GPU. Each SM contains a number of scalar processors (SP), a large register file and some scratch pad memory. Thread-private variables are stored in registers in each SM. Read-only GPU data that has been declared as *constant* is mapped to a different *constant* memory. Programmers can use *shared* memory - which is a low-latency, user-managed scratch pad memory - to store frequently accessed data. Shared memory data is visible to all the threads within the same block.The *syncthreads* construct provides barrier synchronization across threads within the same block.

Each thread block in a kernel grid gets scheduled independently on the SM that it is assigned to. The programmer must be aware that a race condition potentially exists if two or more thread blocks are operating on the same global memory address and at least one of them is performing a write/store operation. This is because there is no control over when the competing blocks will get scheduled. CUDA's *atomic* primitives can be used only to ensure that the accesses are serialized in some arbitrary order, but there is no mechanism to communicate globally across blocks in a single kernel invocation.

2.2 Compiler Transformations

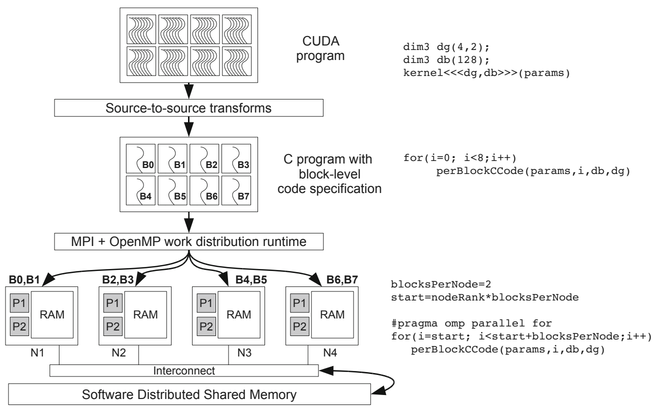
As the per-thread code specification of a CUDA kernel is too fine grained to be scheduled profitably on a CPU, we first transform the kernel into a per-block code specification using transformations described in the MCUDA framework [22]. Logical threads within a thread block are serialized, i.e., the kernel code is executed in a loop with one iteration for each thread in the block. Loop boundaries provide implicit barrier synchronization. Hence, *__syncthreads()* is implemented using a technique called *deep fission*. The single thread loop nest is *split* into two separate loops at the point of invocation of *__syncthreads()*, thereby preserving CUDA's execution semantics. Thread-local variables that are live across such synchronization boundaries are expanded into an array so that each logical thread can maintain its state correctly. Thread-private variables are replicated selectively, avoiding unnecessary duplication while preserving each thread's instance of the variable. The end result of all transformations is a block-level specification of the CUDA kernel that can be compiled and executed on a CPU. [22] has further details on these transformations. A CUDA kernel is composed of several blocks, and is executed by calling the above function several times in a loop. The next section describes how we distribute this execution across nodes using MPI and OpenMP.

3 CUDA for Clusters (CFC)

In this section, we describe CFC in detail. Section 3.1 describes CFC's work partitioning runtime scheme. Section 3.2 describes CFC-SDSM, the Software DSM that used to realize CUDA global memory in a compute cluster.

3.1 Work Distribution

Executing a kernel involves executing the per-block code fragment for all block indices, as specified in the kernel's execution configuration. In this initial work, we employ a simple work distribution scheme that divides the set of block indices into contiguous, disjoint subsets called *block index intervals*. The number of blocks assigned to each node is determined by the number of executing nodes, which is specified as a parameter during execution. If there are more blocks than nodes (as is usually the case), each node gets assigned more than one block. For the example in Fig. 1, the set of block indices $0 - 7$ has been split into four contiguous, disjoint subsets $\{0, 1\}$, $\{2, 3\}$, $\{4, 5\}$ and $\{6, 7\}$, which are scheduled to be executed by nodes N1, N2, N3 and N4 respectively. OpenMP is


CUDA program
Source-to-source transforms
B0
B1
B2
B3
B4
B5
B6
B7
C program with block-level code specification
MPI + OpenMP work distribution runtime
B0,B1
B2,B3
B4,B5
B6,B7
P1
P2
RAM
N1
N2
N3
N4
Interconnect
Software Distributed Shared Memory

```
dim3 dg(4,2);
dim3 db(128);
kernel<<<dg,db>>>(params)
```

```
for(i=0; i<8;i++)
      perBlockCCode(params,i,db,dg)
```

```
blocksPerNode=2
start=nodeRank*blocksPerNode

#pragma omp parallel for
for(i=start; i<start+blocksPerNode;i++)
   perBlockCCode(params,i,db,dg)
```

Fig. 1. Structure of the CFC framework. The pseudo-code for kernel invocation at each stage is shown on the right for clarity.

used within each node to execute the assigned work units in parallel on multiple cores. For example, in Fig. 1, within each node the assigned blocks are executed in parallel using multiple threads on cores P1 and P2. The thread blocks are thus distributed uniformly irrespective of the size of the cluster or number of cores in each cluster node.

3.2 CFC-SDSM

CFC supports CUDA kernel execution on a cluster by providing the global CUDA address space through a software abstraction layer, called CFC-SDSM. We begin by noting CUDA kernels with data races produce unpredictable results on a GPU. However, global data is coherent at kernel boundaries; all thread blocks see the same global data when a kernel commences execution. We therefore enforce a relaxed consistency semantics[8] in CFC-SDSM that ensures coherence of global data at kernel boundaries. Thus, for a data-race free CUDA program, CFC-SDSM guarantees correct execution, but provides no such guarantees for racy programs. *Constant* memory is maintained as separate local copies on every node.

As the size of objects allocated in global memory can be large, CFC-SDSM operates at page-level granularity. Table 1 describes the meta information stored by CFC-SDSM for each page of global data in its page table.

CFC-SDSM Operation CFC-SDSM treats all memory allocated using *cudaMalloc* as global data. Each allocation call typically populates several entries in the CFC-SDSM table. Every memory allocation is performed starting at a page boundary using *mmap*. At the beginning of any kernel invocation, CFC-SDSM marks every global memory page to be *read-only*. Thus, any write to a global page within the kernel results in a segmentation fault which is handled by CFC-SDSM's SIGSEGV handler. The segmentation fault handler first examines

Table 1. Structure of a CFC-SDSM page table entry

Field	Description
pageAddr	Starting address of the page.
pnum	A unique number (index) given to each page, used during synchronization.
written	1 if the corresponding page was written, else 0.
twinAddr	Starting address of the page's *twin*.

the address causing the fault. The fault could either be due to (i) a valid write access to a global memory page that is write-protected, or (ii) an illegal address caused by an error in the source program. In the latter case, the handler prints a stack trace onto standard error and aborts execution. If the fault is due to the former, the handler performs the following actions:

– Set the *written* field of the corresponding CFC-SDSM table entry to 1.
– Create a replica of the current page, called its *twin*. Store the *twin's* address in the corresponding CFC-SDSM table entry.
– Grant write access to the corresponding page and return.

In this way, at the end of the kernel's execution, each node is aware of the global pages it has modified. Note that within each node, the global memory pages and CFC-SDSM table are shared by all executing threads, and hence all cores. So, the SIGSEGV handler overhead is incurred only once for each global page in a kernel, irrespective of the number of threads/cores writing to it. Writes by a CPU thread/thread block are made visible to other CPU threads/thread blocks executing in the same node by the underlying hardware cache coherence mechanism, which holds across multiple sockets of a node. Therefore, no special treatment is needed to handle *shared* memory.

The information of global pages that have been modified within a kernel has to be communicated globally to all other nodes at kernel boundaries. To accomplish this, each node constructs a vector called *writeVector* specifying the set of global pages written by the node during the last kernel invocation. The *writeVector*s are communicated to other nodes using an all-to-all broadcast. Every node then computes the summation of all *writeVectors*. We perform this vector collection-summation operation using `MPI_Allreduce`[20]. At the end of this operation, each node knows the number of modifiers of each global page. For instance, $writeVector[p] == 0$ means that the page having $pnum = p$ has not been modified, and hence can be excluded from the synchronization operation.

Pages having *writeVector[pnum] == 1* have just one modifier. For such pages, the modifying node broadcasts the up-to-date page to every other cluster node To reduce broadcast overheads, all the modified global pages at a node are grouped together in a single broadcast from that node. The actual page broadcast operation is implemented using `MPI_Bcast`.

For pages that have more than one modifier, each modifier must communicate its modifications to other cluster nodes. CFC-SDSM accomplishes this by *diff*ing the modified page with its *twin* page created by the SIGSEGV handler in each modifier node. In CFC-SDSM, each modifier node other than node 0 computes the *diff*s and sends them to node 0, which collects all the *diff*s and applies them to the page in question. *Diff*ing is an inexpensive operation that is easily performed using a bitwise *xor* operation. Node 0 then broadcasts the up-to-date page to every other node. The coherence operation ends with each node receiving the modified pages and updating the respective pages locally.

We show in section 4 that centralizing the *diff*ing process at node 0 does not cause much of a performance bottleneck mainly because the number of pages with multiple modifiers is relatively less. For pages with multiple modifiers, CFC-SDSM assumes that the nodes modified disjoint chunks of the page. If multiple nodes have modified overlapping regions in a global page the program has a data race, and under CUDA semantics the results are unpredictable. CFC-SDSM does not guarantee correctness for such programs.

3.3 Lazy Update

Broadcasting every modified page to every other node creates a high volume of network traffic, which is unnecessary most of the times. We therefore implement a *lazy update* optimization in CFC-SDSM where modified pages are sent to nodes *lazily* on demand. CFC-SDSM uses lazy update if the total number of modified pages across all nodes exceeds a certain threshold. We have found that a threshold of 2048 works reasonably well for many benchmarks (see section 4). In lazy update, global data is updated only on node 0 and no broadcast is performed. Instead, in each node n, read permission is set for all pages p that were modified only by n (since the copy of page p is up-to-date in node n), and the write permission is reset as usual. If a page p has been modified by some other node(s), node n's copy of page p is stale. Hence, CFC-SDSM *invalidates* p by removing all access rights to p in n. Pages which have not been modified by any node are left untouched (with read-only access rights). At the same time, on node 0, a *server thread* is forked to receive and service lazy update requests from other nodes. In subsequent kernel executions, if a node tries to read from an invalidated page (i.e. a page modified by some other node in the previous kernel call), a request is sent to the daemon on node 0 with the required page's *pnum*. In section 4, we show that the *lazy update* scheme offers appreciable performance gains for a benchmark with a large number of global pages.

4 Performance Evaluation

In this section, we evaluate CFC using several representative benchmarks from standard benchmark suites.

4.1 Experimental Setup

For this study, we performed all experiments on an eight-node cluster, where each node is running Debian Lenny Linux. Nodes are interconnected by a high-bandwidth Infiniband network. Each node is comprised of two quad-core Intel Xeon processors running at 2.83GHz, thereby having eight cores.

Compiler Framework. Fig. 2 shows the structure the CFC compiler framework. We use optimization level O3 in all our experiments.

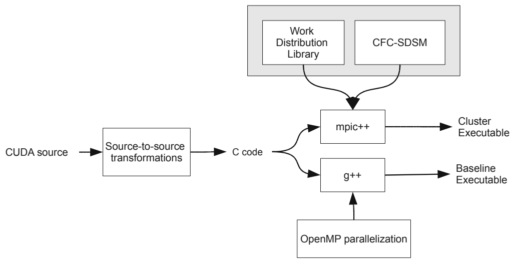


Fig. 2. Structure of the compiler framework

Benchmarks. We used five benchmark applications and one kernel. Four are from the Parboil Benchmark suite [6]. *Blackscholes* and the *Scan* kernel are applications from the NVIDIA CUDA SDK[2]. The benchmarks are from different computing disciplines, and are representative of present day workloads, all of which have mature CUDA implementations in standard benchmark suites. Table 2 briefly describes each benchmark.

Table 2. Benchmarks and description

Benchmark	Description
cp	Coulombic potential computation over one plane in a 3D grid, 100000 atoms
mri-fhd	$F^H d$ computation using in 3D MRI reconstruction, 40 iterations
tpacf	Two point angular correlation function
blackscholes	Call and put prices using Black-Scholes formula, 50000000 options, 20 iterations
scan	Parallel prefix sum, 25600 integers, 1000 iterations
mri-q	Q computation in 3D MRI reconstruction, 40 iterations

Performance Metrics. In all our experiments, we keep the number of threads equal to the number of cores on each node (eight threads per node in our cluster). We haven't explored variable number of threads per node. We define speedup of an *n node* execution as:

$$speedup = \frac{t_{baseline}}{t_{CLUSTER}} \tag{1}$$

, where $t_{baseline}$ represents the baseline multi-threaded execution time on one node, and $t_{CLUSTER}$ represents execution time in the CFC framework on n nodes. The baseline uses a single node and hence requires only OpenMP (and not MPI). The baseline can only gain because of this, thereby ensuring fairness in comparison. Observe that the speedup is computed for a cluster of n nodes (i.e., $8n$ cores) relative to performance on one node (i.e., 8 cores). In effect, for $n = 8$, the maximum obtainable speedup would be 8. Each benchmark has been run 10 times, and the median value is reported.

4.2 Results

Table 3 shows the number of pages of global memory as well as the number of modified pages. Our benchmark set has a mixture of large and small working sets along with varying percentages of modified global data, thus covering a range of GPGPU behavior suitable for studying an implementation such as ours.

Benchmark speedups are shown in Fig. 3. Fig. 3(*a*) shows speedups with the lazy

Table 3. Number of pages of global memory declared and modified in each benchmark

Benchmark	Global pages	Modified	% Unmodified
Cp	1024	1024	0
Mri-fhd	1298	510	60.7
Tpacf	1220	8	99.3
BlackScholes	244145	97658	60
Mri-q	1286	508	60.49
Scan	50	25	50

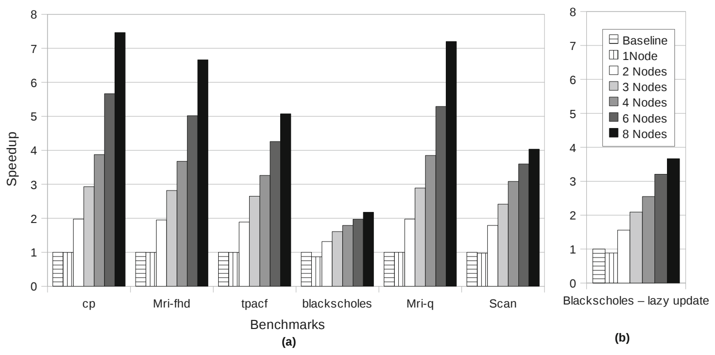


Fig. 3. Comparison of execution times of various benchmark applications on our system. (*a*) shows normalized speedups on a cluster with 8 nodes without lazy update. (*b*) shows the performance of *BlackScholes* with the *lazy update* optimization.

update optimization disabled for all the benchmarks, while 3(*b*) shows speedups for the *BlackScholes* benchmark when the lazy update optimization is enabled. As we have set a threshold of at-least 2048 global pages to trigger CFC-SDSM to operate in lazy mode, only blackscholes triggers this operation. Any number of pages less than this can easily be handled by CFC-SDSM in the normal mode, and the experimental results demonstrate it. Hence we study the lazy update effect only on *Blackscholes*. We make the following observations:

– Our implementation has low runtime overhead. Observe the speedups for $n = 1$, i.e., the second bar. In almost all cases, this value is close to the baseline. *BlackScholes* slows down by about 14% due to its large global data working set.

- The *Cp* benchmark shows very high speedups in spite of having a high percentage of global data pages being modified. *Cp* is a large benchmark with lots of computations that can utilize many nodes efficiently.
- The *Scan* benchmark illustrates the effect of a CUDA kernel design on its performance on a cluster. Originally, the *Scan* kernel is small where only 512 elements are processed per kernel. Spreading such a small kernel's execution over many nodes was an overkill and provided marginal performance gains comparable to *Blackscholes* in Fig. 3(a). However, after the kernel was modified (*coarsened* or *fattened*) to processes 25600 elements per kernel, we achieve the speedups shown in 3(a).
- The *BlackScholes* benchmark shows scalability, but low speedups. Due to the large volume of network traffic it generates, this benchmark benefits from lazy update. On a cluster with 8 nodes, we obtain a speedup of 3.7X with lazy update, compared to 2.17X without lazy update. This suggests that the performance gained by reducing interconnect traffic compensates for the overheads incurred by creating the daemon thread. We have observed that for this application, invalidated pages are never read in any node.
- We can observe network overhead specifically only in *BlackScholes* where we've overloaded CFC-SDSM with many pages. The lazy update scheme seems to work pretty well even for large memory sizes. We would like to explore potential problems when we scale this to hundreds of nodes in the future.
- Across the benchmarks, our runtime approach to extend CUDA programs to clusters has achieved speedups ranging from 3.7X to 7.5X on an 8 node cluster.

In summary, we are able to achieve appreciable speedup and a good scaling efficiency (upto 95%) with number of nodes in the cluster. While an 8-node cluster is not a very big cluster, it serves as a reasonable platform to demonstrate the effectiveness of CFC. Future work will deal with studying larger clusters and problems arising from that.

5 Related Work

We briefly discuss a few previous works related to programming models, using CUDA on non-GPU platforms and software DSMs. The *Partitioned Global Address Space* family of languages (Chapel[11], X10[12] etc.) aims to combine the advantages of both message-passing and shared-memory models. Intel's Concurrent collections [1] is another shared memory programming model that aims to abstract the description of parallel tasks.

Previous works like [7], [14] and [22] use either compiler techniques or binary translation to execute kernels on x86 CPUs. In all the works mentioned here, CUDA kernels have been executed on single shared-memory hardware.

Various kinds of software DSMs have been suggested in literature like [9], [15], [18], and [19], to name a few. CFC-SDSM differs from the above works in the sense that locks need not be acquired and released explicitly by the programmer.

All global memory data is ‘locked’ just before kernel execution and ‘released’ immediately after, by definition. Also, synchronization operation proceeds either eagerly or lazily, depending on the total size of global memory allocated. This makes our DSM very lightweight and simple.

6 Conclusions and Future Work

In this paper, we have presented an initial study in exploring CUDA as a language to program clusters of multi-core machines. We have implemented CFC, a framework that uses a mixture of compiler transformations, work distribution runtime and a lightweight software DSM to collectively implement CUDA’s semantics on a multi-core cluster. We have evaluated our implementation by running six standard CUDA benchmark applications to show that there are indeed promising gains that can be achieved.

Many interesting directions can be pursued in the future. One direction could be towards optimizing network usage by building a static communication cost estimation model or tracking global memory access patterns that can be used by the runtime to schedule blocks across nodes appropriately. Another interesting and useful extension to this work would be to consider GPUs on multiple nodes as well, along with multi-cores. Automatic compile-time kernel coarsening and automatic kernel execution configuration tuning are other interesting areas.

References

1. Intel concurrent collections for c++, `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/`
2. Nvidia cuda c sdk, `http://developer.download.nvidia.com/compute/cuda/sdk`
3. Nvidia cuda zone, `http://www.nvidia.com/cuda`
4. Opencl overview, `http://www.khronos.org/developers/library/overview/opencl_overview.pdf`
5. Openmp specifications, version 3.0, `http://openmp.org/wp/openmp-specifications/`
6. The parboil benchmark suite, `http://impact.crhc.illinois.edu/parboil.php`
7. The portland group, `http://www.pgroup.com`
8. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29, 66–76 (1995)
9. Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: Treadmarks: Shared memory computing on networks of workstations. Computer 29(2), 18–28 (1996)
10. Cappello, F., Etiemble, D.: Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM), Supercomputing 2000. IEEE Computer Society, Washington, DC (2000)
11. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. 21(3), 291–312 (2007)

12. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA 2005: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 519–538. ACM, New York (2005)
13. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (shoc) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU 2010, pp. 63–74. ACM, New York (2010)
14. Diamos, G.F., Kerr, A.R., Yalamanchili, S., Clark, N.: Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In: PACT 2010: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, pp. 353–364. ACM, New York (2010)
15. Gelado, I., Stone, J.E., Cabezas, J., Patel, S., Navarro, N., Hwu, W.M.W.: An asymmetric distributed shared memory model for heterogeneous parallel systems. SIGARCH Comput. Archit. News 38(1), 347–358 (2010)
16. Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B.R., Zheng, B.: Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, pp. 205–216. ACM, New York (2010)
17. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: Opencl as a programming model for gpu clusters. In: LCPC 2011: Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing, (2011)
18. Li, K., Hudak, P.: Memory coherence in shared virtual memory systems. ACM Trans. Comput. Syst. 7(4), 321–359 (1989)
19. Manoj, N.P., Manjunath, K.V., Govindarajan, R.: Cas-dsm: a compiler assisted software distributed shared memory. Int. J. Parallel Program. 32(2), 77–122 (2004)
20. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI-The Complete Reference, Volume 1: The MPI Core. MIT Press, Cambridge (1998)
21. Stratton, J.A., Grover, V., Marathe, J., Aarts, B., Murphy, M., Hu, Z., Hwu, W.M.W.: Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In: CGO 2010: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 111–119. ACM, New York (2010)
22. Stratton, J.A., Stone, S.S., Hwu, W.-M.W.: Mcuda: An efficient implementation of cuda kernels for multi-core cpus, pp. 16–30 (2008)