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

Raghu Prabhakar, R. Govindarajan, and Matthew J. Thazhuthaveetil

Supercomputer Education and Research Centre,
Indian Institute of Science, Bangalore, India
raghup@hpc.serc.iisc.ernet.in
{govind, mjt}@serc.iisc.ernet.in

Abstract. Rapid advancements in multi-core processor architectures along with low-cost, low-latency, high-bandwidth interconnects have made clusters of multi-core machines a common computing resource. Unfortunately, writing good parallel programs to efficiently utilize all the resources in such a cluster is still a major challenge. Programmers have to manually deal with low-level details that should ideally be the responsibility of an intelligent compiler or a run-time layer. 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 number of CUDA developers and benchmarks along with the 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, Clusters, Software DSM.

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 processors interconnected via a network infrastructure. While such clusters 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 [13][27] using multiple programming models like OpenMP[8] (intra-node) and MPI[24] (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. Various programming languages and models that have been proposed as a solution to this problem [15][14] 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.

The use of GPUs for general purpose computing applications, often called GPGPU[2] (General Purpose computing on GPUs), has been facilitated mainly by NVIDIAs CUDA (Compute Unified Device Architecture [6]) and OpenCL [7]. In particular, CUDA has become a popular language as evident from an increasing number of users [6] and benchmarks [9] [16]. 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 [26][25] have shown the effectiveness using CUDA to program multi-core shared memory CPUs, and similar research has been performed on OpenCL as well [20][19]. More recently, a compiler that implements CUDA on multi-core x86 processors has been released commercially by the Portland Group [10]. CUDA has evolved into a very mature software stack with efficient supporting tools like debuggers and profilers, making application development and deployment easy. We believe that CUDA provides a natural and convenient way of exposing data locality information and expressing multi-level parallelism in a kernel.

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. 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 [26] 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 [9] and the NVIDIA CUDA SDK [5] on an experimental 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 on CUDA programming model and the compiler transformations employed. 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 sizes of which are specified by the programmer during invocation. Data transfer between the main memory and GPU DRAM is performed explicitly using CUDA APIs. Thread-private variables are stored in registers in each SM. Programmers can declare read-only GPU data as *constant* if it is read-only. 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.

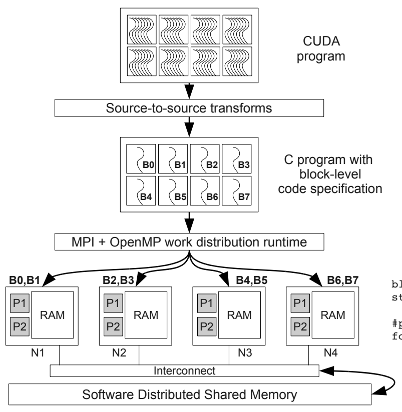
Each thread block in a kernel grid gets scheduled independently on the streaming multiprocessor (SM) that it is assigned to. The programmer must be aware that a race condition potentially exists if two or more thread blocks are operating on the same global memory address and at least one of them is performing a write 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.

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 [26]. 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. Barrier synchronization across threads is implemented using a technique called *deep fission*, where the single thread loop is *split* into two separate loops, thereby preserving CUDA's execution semantics. Thread-private variables are replicated selectively, avoiding unnecessary duplication while preserving each thread's instance of the variable. [26] has further details on each transformation.

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.


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.

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 (*dg*, in Figure 1). 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. For the example in Figure 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 used within each node to execute the set of work units within the block index interval in parallel on multiple cores. For example, in Figure 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

In the recent past, software DSMs have re-emerged as an popular choice to exploit parallelism on distributed and heterogeneous parallel systems [23][18]. This trend suggests that an efficient implementation of a software DSM which enforces appropriate consistency semantics as required by the target environment can actually provide the desired performance and scalability.

CFC supports CUDA kernel execution on a cluster by providing the global CUDA address space through a software abstraction layer, called CFC-SDSM. Recall that under CUDA semantics on a GPU, if two or more thread blocks operate on the same global memory address with at least one of them performing a write, then there is a potential race as there is no guarantee on the order in which thread blocks get scheduled for execution. 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[11] 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. For programs with data races inside a kernel, CFC-SDSM only ensures that the writes by the multiple competing thread blocks are ordered in some order. *Constant* memory is read-only, and hence is maintained as separate local copies on every node.

As the size of objects allocated in global memory can be large (we have seen arrays running to hundreds of pages in some programs), 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.

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

Field	Description
pageAddr	Starting address[1] 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*.

CFC-SDSM Operation Global data is identified at allocation time. 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 sets the *read* permission and resets the *write* permission for each global memory page. 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 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 SEGV 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 which global pages have been modified within a kernel is known within a single node of the cluster, and has to be made globally known across the cluster by communication 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 with 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*[24]. 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 is implemented using MPI_Bcast.

Let us now consider the case where a page has been modified by more than one node. Each modifier must communicate its modifications to other cluster nodes. CFC-SDSM accomplishes this by *diff*ing the modified page with its *twin* page. Recall that a twin would have been created in each of the modifying cluster nodes by the SIGSEGV handler. 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. Node 0 then broadcasts the up-to-date page to every other node. The coherence operation ends by 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 are relatively less. For pages with multiple modifiers, CFC-SDSM assumes that the nodes modified disjoint chunks of the page. This is true for most of the CUDA programs we analyzed. 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.

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 block/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 Figure 2 shows the structure the CFC compiler framework. We use optimization level O3 in all our experiments.

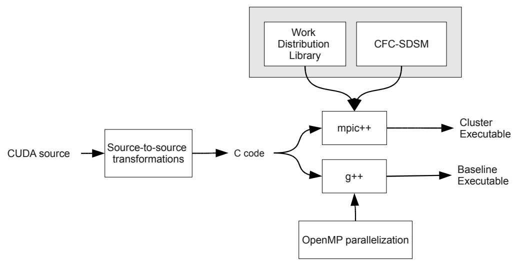


Fig. 2. Structure of the compiler framework.

Benchmarks We used five benchmark applications and one kernel. Four are from the Parboil Benchmark suite [9]. *Blackscholes* and the *Scan* kernel are applications from the NVIDIA CUDA SDK[5]. 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 (8 threads per node in our cluster). We define speedup of an *n node* execution as:

$$speedup = \frac{t_{baseline}}{t_{CLUSTER}}, \qquad (1)$$

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. Observe that the speedup is computed for a cluster of n nodes (i.e., $8n$ cores) relative to performance on 1 node (i.e., 8 cores). In effect, for $n = 8$, the maximum obtainable speedup would be 8.

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.

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

Benchmark	Global pages	Modified
Cp	1024	1024
Mri-fhd	1298	510
Tpacf	1220	8
BlackScholes	244145	97658
Mri-q	1286	508
Scan	50	25

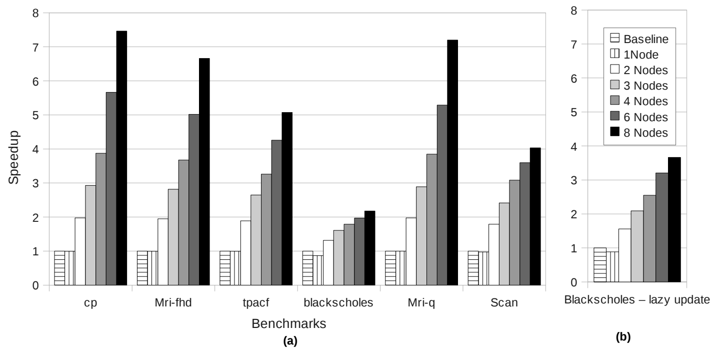


Fig. 3. Comparison of execution times of various benchmark applications on our system. (a) shows normalized speedups on a cluster with 8 nodes without lazy update. (b) shows the performance of *BlackScholes* with the *lazy update* optimization.

Benchmark speedups are shown in Figure 3. Figure 3(a) shows speedups with the lazy update optimization disabled for all the benchmarks, while 3(b) shows speedups for the *BlackScholes* benchmark when the lazy update optimization is enabled. 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 figure 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. However, this application suffers from numerous broadcasts during synchronization and hence gains from the lazy update optimization. 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.
- 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.

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 (X10[15], Chapel[14] etc.) aims to combine the advantages of both message-passing and shared-memory models. Intel's Cluster OpenMP[1] extended the OpenMP programming language to make it usable on clusters, but has been deprecated [3]. Intel's Concurrent collections [4] is another shared memory programming model that aims to abstract the description of parallel tasks.

Previous works like [26], [10] and [17] 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 [21] [12] [22] [18] etc. 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

With the rise of clusters of multi-core machines as a common and powerful HPC resource, there exists a necessity for a unified language to write efficient, scalable parallel programs easily. Here, we have presented an initial study in exploring CUDA as a language to program such clusters. 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 promising gains that can be achieved.

Many interesting directions can be pursued in the future. One direction could be in building a static communication cost estimation model 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. Also, kernel coarsening and automatic kernel execution configuration tuning could be performed. Further reduction the DSM interconnect traffic could also be achieved by smarter methods that track global memory access patterns.

References

1. Cluster openmp* for intel compilers. `http://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers/`.
2. General purpose computation using graphics hardware.
3. Intel c++ compiler 11.1 release notes. `http://software.intel.com/en-us/articles/intel-c-compiler-111-release-notes/`.
4. Intel concurrent collections for c++. `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/`.
5. Nvidia cuda c sdk. `http://developer.download.nvidia.com/compute/cuda/sdk`.
6. Nvidia cuda zone. `http://www.nvidia.com/cuda`.
7. Opencl overview. `http://www.khronos.org/developers/library/overview/opencl_overview.pdf`.
8. Openmp specifications, version 3.0. `http://openmp.org/wp/openmp-specifications/`.
9. The parboil benchmark suite. `http://impact.crhc.illinois.edu/parboil.php`.
10. The portland group. `http://www.pgroup.com`.
11. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.
12. C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.
13. F. Cappello and D. Etiemble. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.
14. B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

15. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.
16. 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 *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 63–74, New York, NY, USA, 2010. ACM.
17. G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 353–364, New York, NY, USA, 2010. ACM.
18. I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. *SIGARCH Comput. Archit. News*, 38(1):347–358, 2010.
19. J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 205–216, New York, NY, USA, 2010. ACM.
20. J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, and J.-D. Choi. An opencl framework for heterogeneous multicores with local memory. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 193–204, New York, NY, USA, 2010. ACM.
21. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.
22. N. P. Manoj, K. V. Manjunath, and R. Govindarajan. Cas-dsm: a compiler assisted software distributed shared memory. *Int. J. Parallel Program.*, 32(2):77–122, 2004.
23. B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson. Programming model for a heterogeneous x86 platform. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 431–440, New York, NY, USA, 2009. ACM.
24. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core.* MIT Press, Cambridge, MA, USA, 1998.
25. J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. mei W. Hwu. Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 111–119, New York, NY, USA, 2010. ACM.
26. J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. pages 16–30, 2008.
27. X. Wu and V. Taylor. Performance characteristics of hybrid mpi/openmp implementations of nas parallel benchmarks sp and bt on large-scale multicore supercomputers. *SIGMETRICS Perform. Eval. Rev.*, 38:56–62, March 2011.