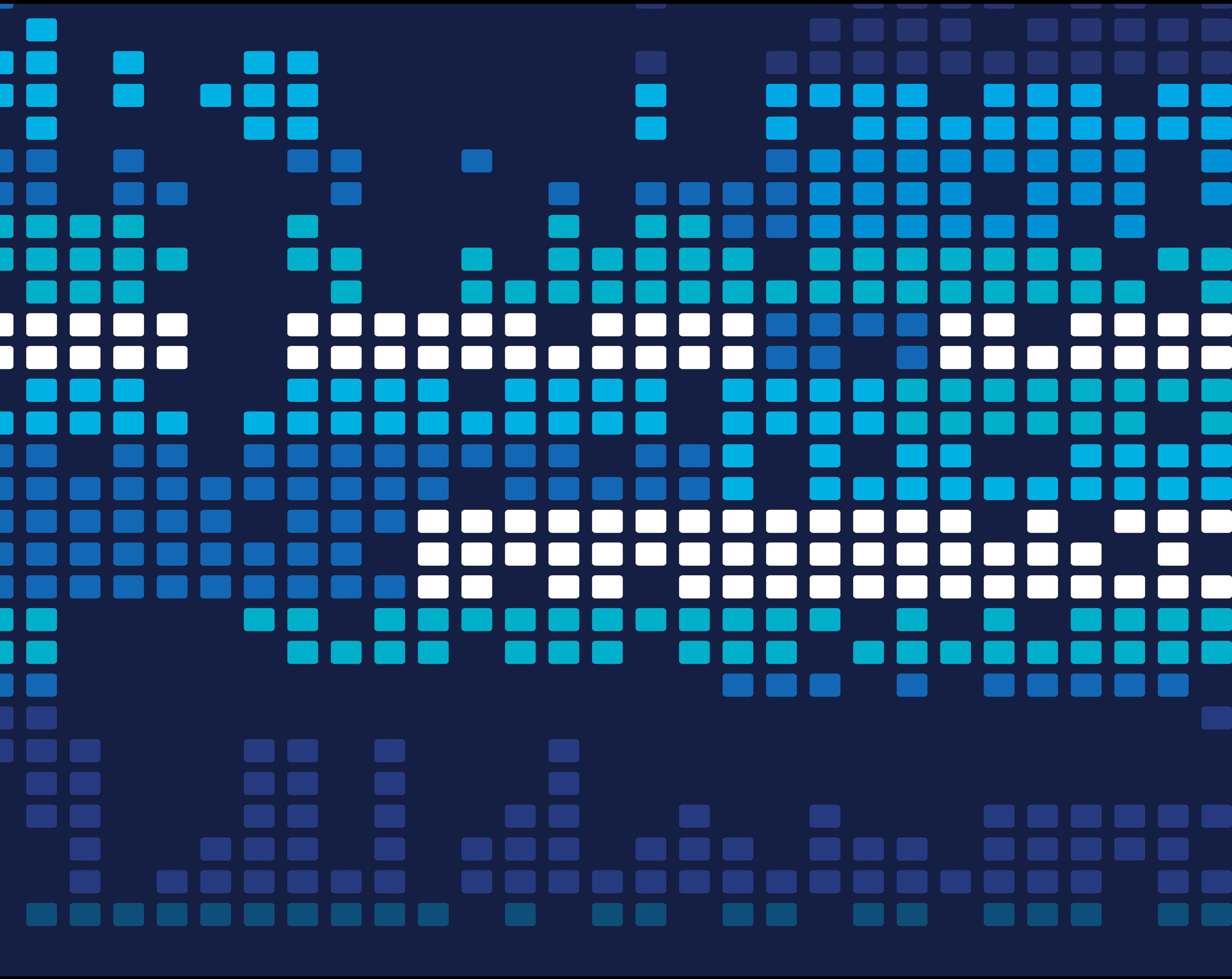
Scientific Programming

Programming Models, Languages, and Compilers for Manycore and Heterogeneous Architectures

Guest Editors: Sunita Chandrasekaran, Barbara Chapman, Xinmin Tian, and Yonghong Yan



Programming Models, Languages, and Compilers for Manycore and Heterogeneous Architectures

Programming Models, Languages, and Compilers for Manycore and Heterogeneous Architectures

Guest Editors: Sunita Chandrasekaran, Barbara Chapman, Xinmin Tian, and Yonghong Yan

Editorial Board

Contents

Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 376317, 1 page
http://dx.doi.org/10.1155/2015/376317

Editorial

Programming Models, Languages, and Compilers for Manycore and Heterogeneous Architectures

Sunita Chandrasekaran,[1] Barbara Chapman,[1] Xinmin Tian,[2] and Yonghong Yan[3]

[1] *University of Houston, 4800 Calhoun Road, Houston, TX 77004, USA*
[2] *Intel Corporation, 2200 Mission College Boulevard, SC12-301, Santa Clara, CA 95052, USA*
[3] *Oakland University, 2200 N Squirrel Road, Rochester, MI 48309, USA*

Correspondence should be addressed to Sunita Chandrasekaran; sunita@cs.uh.edu

Received 25 March 2015; Accepted 25 March 2015

Aim and Scope. Hardware is emerging rapidly, core count is increasing, and systems consist of large cluster of nodes. These nodes are becoming more heterogeneous, that is, multicore CPUs attached to accelerators meant to address specific needs of specific computations. The type of parallelism each accelerator offers is quite different from the other. On the other hand, software for such emerging parallel and heterogeneous computing systems is still catching up. The gap between hardware and software is growing; it is a challenge for the software developers to keep up with the hardware advancements. Thus, there is an urgent need to develop and maintain sophisticated software that can not only offer performance productive solutions but also be applicable to a wide range of hardware systems. Some of the promising and widely used programming solutions include directive-based programming models such as OpenMP, vendor-specific programming models such as NVIDIA's CUDA, OpenCL, and recently emerging programming model, OpenACC. This special issue publishes papers on the evaluations of these models for parallel computing with respect to several factors including locality-aware scheduling, data transfer optimizations, SIMD vectorization on Phi coprocessors, and programming multi-GPU.

Acknowledgments

The editors would like to thank the authors for their tremendous effort and time spent on their respective manuscripts. We relied on peer-review process to uphold the quality of the manuscripts. To this effort, we would like to thank and acknowledge all external reviewers for spending their valuable time to provide the authors with valuable feedback.

Sunita Chandrasekaran
Barbara Chapman
Xinmin Tian
Yonghong Yan

Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 981759, 16 pages
http://dx.doi.org/10.1155/2015/981759

Research Article

Locality-Aware Task Scheduling and Data Distribution for OpenMP Programs on NUMA Systems and Manycore Processors

Ananya Muddukrishna,[1] Peter A. Jonsson,[2] and Mats Brorsson[1,2]

[1] *KTH Royal Institute of Technology, School of Information and Communication Technology, Electrum 229, 164 40 Kista, Sweden*
[2] *SICS Swedish ICT AB, Box 1263, 164 29 Kista, Sweden*

Correspondence should be addressed to Ananya Muddukrishna; ananya@kth.se

Received 15 May 2014; Accepted 29 September 2014

Academic Editor: Sunita Chandrasekaran

Performance degradation due to nonuniform data access latencies has worsened on NUMA systems and can now be felt on-chip in manycore processors. Distributing data across NUMA nodes and manycore processor caches is necessary to reduce the impact of nonuniform latencies. However, techniques for distributing data are error-prone and fragile and require low-level architectural knowledge. Existing task scheduling policies favor quick load-balancing at the expense of locality and ignore NUMA node/manycore cache access latencies while scheduling. Locality-aware scheduling, in conjunction with or as a replacement for existing scheduling, is necessary to minimize NUMA effects and sustain performance. We present a data distribution and locality-aware scheduling technique for task-based OpenMP programs executing on NUMA systems and manycore processors. Our technique relieves the programmer from thinking of NUMA system/manycore processor architecture details by delegating data distribution to the runtime system and uses task data dependence information to guide the scheduling of OpenMP tasks to reduce data stall times. We demonstrate our technique on a four-socket AMD Opteron machine with eight NUMA nodes and on the TILEPro64 processor and identify that data distribution and locality-aware task scheduling improve performance up to 69% for scientific benchmarks compared to default policies and yet provide an architecture-oblivious approach for programmers.

1. Introduction

NUMA systems consist of several multicore processors attached to local memory modules. Local memory can be accessed both faster and with higher bandwidth than remote memory by cores within a processor. Disparity between local and remote node access costs increases both in magnitude and nonuniformity as NUMA systems grow. Modern NUMA systems have reached such size and complexity that even simple memory-oblivious parallel executions such as the task-based Fibonacci program with work-stealing scheduling have begun to suffer from NUMA effects [1]. Careful data distribution is crucial for performance irrespective of memory footprint on modern NUMA systems.

Data distribution is also required on manycore processors which exhibit on-chip NUMA effects due to banked shared caches. Cores can access their local cache bank faster than remote banks. The latency of accessing far-off remote cache banks approaches off-chip memory access latencies. Another performance consideration is that cache coherence of manycore processors is software configurable [2]. Scheduling should adapt to remote cache bank access latencies that can change based on the configuration.

Scheduling decisions of the runtime system are key to task-based program performance. Scheduling decisions are made according to scheduling policies which until now have focused mainly on load-balancing—distributing computation evenly across threads. Load-balancing is a simple decision requiring little information from task abstractions used by the programmer and has been effective for several generations of multicore processors.

However, scheduling policies need to minimize memory access costs in addition to load-balancing for performance on NUMA systems and manycore processors. Strict load-balancing policies lose performance since they neglect data locality exposed by tasks. Neglecting data locality violates design principles of the complex memory subsystems that support NUMA systems and manycore processors. The subsystems require scheduling to keep cores running uninterrupted and provide peak performance by exploiting data locality.

Despite rising importance of data distribution and scheduling, OpenMP—a popular and widely available task-based programming paradigm—neither specifies data distribution mechanisms for programmers nor provides scheduling guidelines for NUMA systems and manycore processors even in the latest version 4.0.

Current data distribution practices on NUMA systems are to either use third-party tools and APIs [3–5] or repurpose the OpenMP `for` work-sharing construct to allocate and distribute data to different NUMA nodes. The third-party tools are fragile and might not be available on all machines and the clever use of the `parallel for` work-sharing construct [6] relies on a particular OS page management policy and requires the programmer to know about the NUMA node topology on the target machine.

Similar data distribution effort is required on manycore processors. For example, programmers directly use system API to distribute data on shared cache banks on the TILEPro64. There are no third-party tools to simplify data distribution effort. Programmers additionally have to match data distribution choice with numerous configurations available for the cache hierarchy for performance.

Expert programmers can still work around existing data distribution difficulties, but even for experts the process can be described as fragile and error-prone. Average programmers who do not manage to cope with all the complexity at once pay a performance penalty when running their programs, a penalty that might be partially mitigated from clever caching by the hardware. The current situation will get increasingly worse for everybody since NUMA effects are exacerbated by growing network diameters and increased cache coherence complexity [7] that inevitably follow from increasing sizes of NUMA systems and manycore processors.

We present a runtime system assisted data distribution scheme that allows programmers to control data distribution in a portable fashion without forcing them to understand low-level system details of NUMA systems and manycore processors. The scheme relies on the programmer to provide high-level hints on the granularity of the data distribution in calls to `malloc`. Programs without hints will work and have the same performance as before, which allows gradual addition of hints to programs to get partial performance benefits. Our runtime system assisted distribution scheme requires nearly the same programmer effort as regular calls to `malloc` and yet doubles the performance for some scientific workloads on NUMA systems.

We also present a locality-aware scheduling algorithm for OpenMP tasks which reduces memory access times by leveraging locality information gained from data distribution and task data footprint information from the programmer. Our scheduling algorithm improves performance over existing schedulers by up to 50% on our test NUMA system and 88% on our test manycore processor in programs where NUMA effects degrade program performance and remains

Table 1: Simple data distribution policies for the programmer.

Policy	Behavior
Standard	Delegate data distribution to the OS.
Fine	Distribute data, unit-wise round-robin, across all locations.
Coarse	Distribute data units, per-allocation round-robin, across all locations.

Table 2: Data distribution policy abstractions.

System	Unit	Location
NUMA system	Page	NUMA node
TILEPro64	Cache line	Home cache

competitive for other programs. Performance of scientific programs—blocked matrix multiplication and vector cross product—improves by 14% and 69%, respectively, when the locality-aware scheduler is used.

The paper is an extension of our previous work on NUMA systems [8] and manycore processors [9]. We provide common data distribution mechanisms (Tables 1 and 2) and unify the presentation of locality-aware scheduling mechanisms (Algorithms 1, 2, and 3) for both NUMA systems and manycore processors. The new experimental setup for manycore processors enables L1 caching (Section 5.2) for a more realistic scenario. We disabled L1 caching in previous work to isolate locality-aware scheduling effects. We provide new measurements for manycore processors with a work-stealing scheduler as the common baseline (Figures 9 and 10). Previous work used a central queue-based scheduler as the baseline for manycore processors. We demonstrate the impact of vicinity sizes while stealing tasks (Figure 11), which is not done in previous work.

2. Potential for Performance Improvements

We quantify the performance improvement from data distribution by means of an experiment conducted on an eight-NUMA node system with four AMD Opteron 6172 processors. The topology of the system is shown in Figure 1. The maximum NUMA distance of the system according to the OS is 22, which is an approximation of the maximum latency between two nodes. NUMA interconnects of the system are configured for maximum performance with an average NUMA factor of 1.19 [11]. Latencies to access 4 MB of memory from different NUMA nodes measured using the BenchIT tool are shown in Figure 2. Detailed memory latencies of a similar system are reported by Molka et al. [12].

We execute task-based OpenMP programs using Intel's OpenMP implementation with two different memory allocation strategies: the first strategy uses malloc with the *first-touch* policy and the second distributes memory pages evenly across NUMA nodes using the numactl tool [5]. We use first-touch as a short hand for malloc with first-touch policy in the rest of the paper. We measure execution time of the parallel section of each program and quantify the amount of time

```
(1) Procedure deal-work(task T, queues $Q_1, \ldots, Q_N$, current node n, cores per node C)
(2)     Populate $D[1:N]$ with bytes in T.depend_list;
(3)     if $sum(D) > sizeof(LLC)/C$ and $Standard\_Deviation(D) > 0$ then
(4)         find $Q_l$ with least NUMA distance-weighted cost to $D$;
(5)         enqueue($Q_l$, T);
(6)     else
(7)         enqueue($Q_n$, T);
(8)     end
(9) end
```

Algorithm 1: Work-dealing algorithm for NUMA systems.

```
(1) Procedure find-work(queues $Q_1, \ldots, Q_N$, current node n, cores per node C)
(2)     if empty $Q_n$ then
(3)         for $Q_i$ in (Sort $Q_1, \ldots, Q_N$ by NUMA distance from n) do
(4)             if $sizeof(Q_i) > distance(i, n) * C$ then
(5)                 Run dequeue($Q_i$);
(6)                 break;
(7)             end
(8)         end
(9)     else
(10)        Run dequeue($Q_n$);
(11)    end
(12) end
```

Algorithm 2: Work-finding algorithm for NUMA systems.

```
(1) Procedure deal-work(task T, queues $Q_1, \ldots, Q_N$, current home cache n,
    current data distribution policy p, access-intensive dependence index a)
(2)     if $p == coarse$ then
(3)         if exists $a$ then
(4)             find $Q_a$ containing T.depend_list[$a$];
(5)             enqueue($Q_a$, T);
(6)         else
(7)             Populate $D[1:N]$ with bytes in T.depend_list;
(8)             if $sum(D) > sizeof(L1)$ then
(9)                 find $Q_l$ with least home cache latency cost to $D$;
(10)                enqueue($Q_l$, T);
(11)            else
(12)                enqueue($Q_n$, T);
(13)            end
(14)        end
(15)    else
(16)        enqueue($Q_n$, T);
(17)    end
(18) end
```

Algorithm 3: Work-dealing algorithm for TILEPro64.

spent waiting for memory by counting dispatch stall cycles which includes load/store unit stall cycles [13].

Several programs show a reduction in execution time when data is distributed across NUMA nodes as shown in Figure 3. The reduction in dispatch stall cycles contributes to the reduction in execution time. Performance is maintained with data distribution for all remaining programs except Strassen.

We can explain why benchmarks maintain or lose performance with data distribution. Alignment scales linearly which implies low communication. Data distribution does not relieve the memory subsystem for FFT, Health, SparseLU,

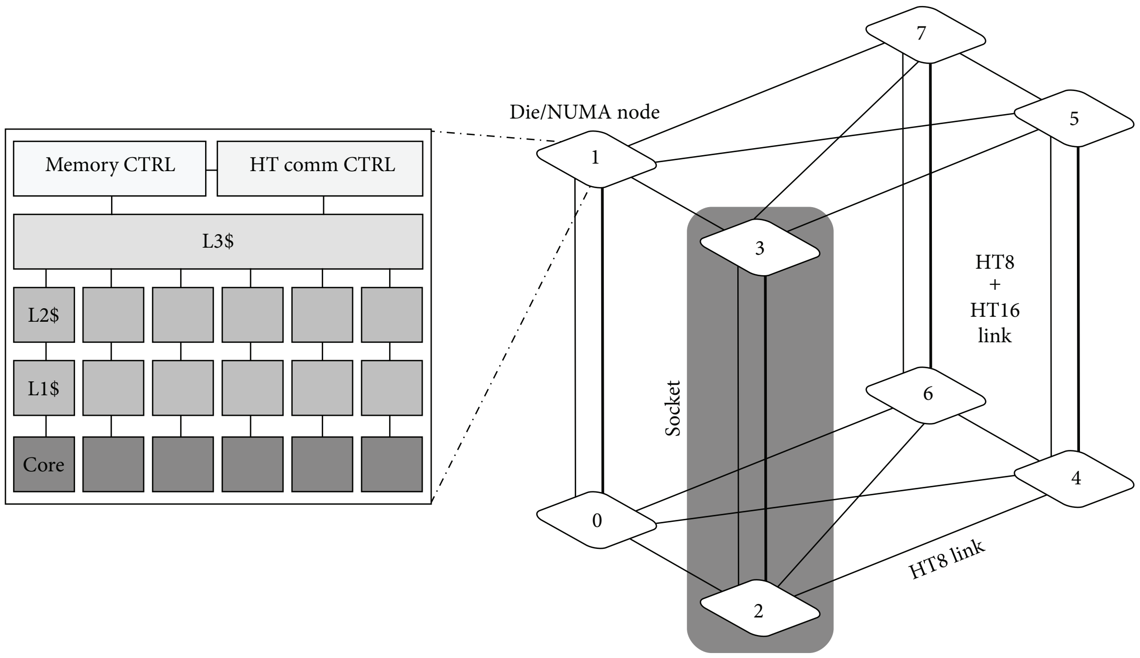


Figure 1: Topology of eight NUMA node, 48-core system with four AMD Opteron 6172 processors. Each processor has a 64 KB DL1 cache, a 512 KB L2 cache and a 5 MB L3 cache.

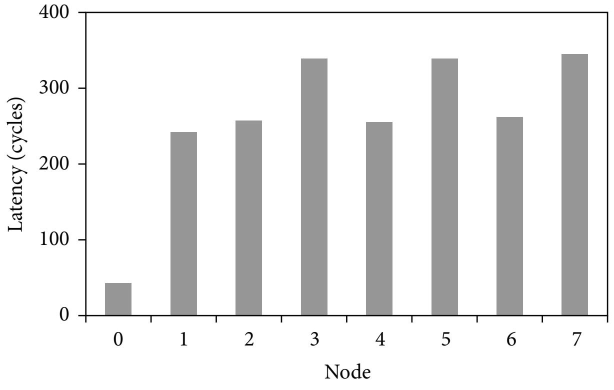


Figure 2: Latencies measured while accessing 4 MB of data allocated on different NUMA nodes from node 0 of the eight-node Opteron system. Remote node access is expensive.

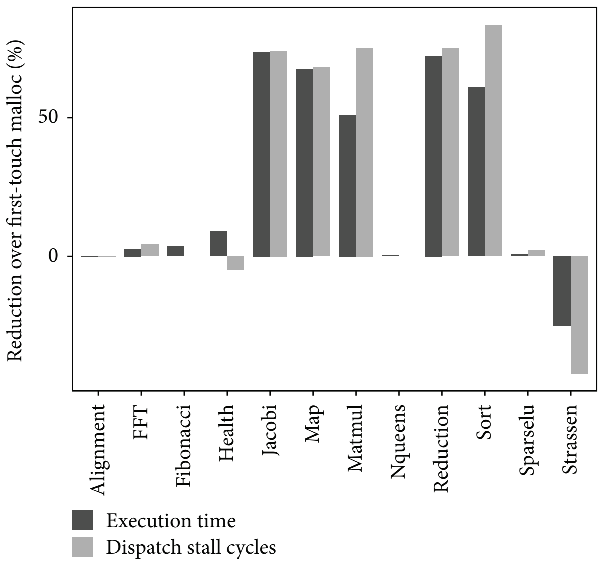


Figure 3: Performance impact of data distribution compared to first-touch in programs taken or derived from the Barcelona OpenMP Task Suite (BOTS) [10] and executed on the eight-node Opteron system. Execution time corresponds to the critical path of parallel section. Dispatch stall cycles are aggregated over all program tasks. Most programs improve or maintain performance when data is distributed across NUMA nodes.

and Strassen benchmarks. Execution time of Health surprisingly improves despite increased dispatch stall cycles implying bandwidth improvements with data distribution. Strassen is a counter-example whose performance degrades from data distribution. Strassen allocates memory inside tasks. Distributing the memory incurs higher access latencies than first-touch.

We demonstrate how locality-aware task scheduling used in conjunction with data distribution can further improve performance by means of an experiment on the TILEPro64 manycore processor. We explain the experiment after introducing key locality features of the TILEPro64 architecture.

The TILEPro64 is a 64-core tiled architecture processor with a configurable directory-based cache coherence protocol and topology as shown in Figure 4. Load and store misses from cores are handled by a specific L2 bank called the *home cache*. A cache line supplied by the home cache can be allocated selectively in the local L2 bank (inclusive) and the L1 cache depending on software configuration. Stores in a tile are

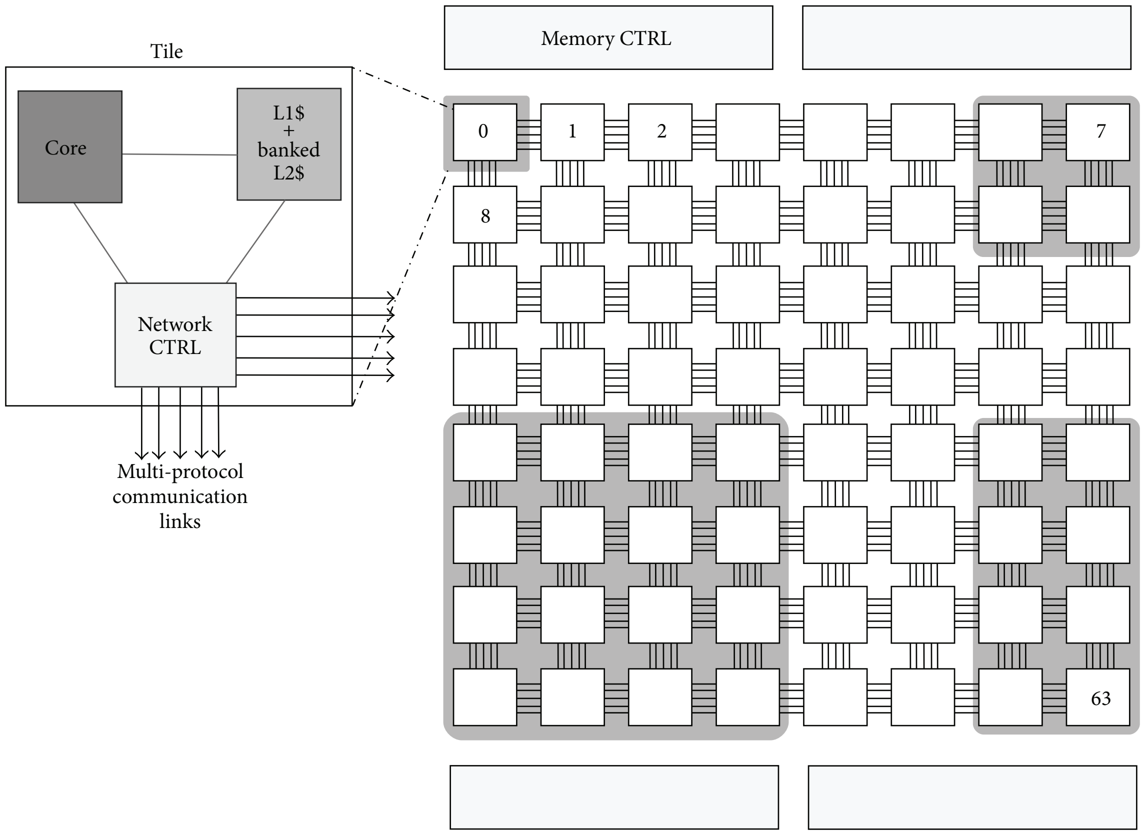


Figure 4: TILEPro64 topology. Tiles are connected by an 8 × 8 mesh on-chip network. Each tile contains a 32-bit VLIW integer core, a private 16 KB IL1 cache, a private 8 KB DL1 cache, a 64 KB bank of the shared 4 MB L2 cache and a network controller.

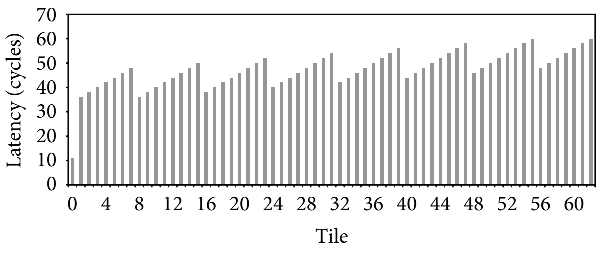


Figure 5: Latencies measured while accessing a cache line from different home cache access from tile 0 under isolation. Latencies increase in the presence of multiprogrammed workloads, OS, and hypervisor interference. Tile 63 runs dedicated system software and is excluded from measurement. Off-tile access takes 4–6 times longer.

always write-through to the home cache with a store update if the line is found in the L1 cache. Load latency of a cache line depends on home cache location and is nonuniform as shown in Figure 5. Remote home caches take four to six times longer to access than local home caches.

TILEPro64 system software also provides data distribution mechanisms. Cache lines in a main memory page can be distributed uniformly to all home caches or to a single home cache. The home cache of an allocated line can additionally be changed at a high cost through migration [14]. The performance impact of data distribution on the TILEPro64 is similar to NUMA systems. Memory allocations through malloc are uniformly distributed to all home caches.

We now explain our experiment to demonstrate locality-aware scheduling effectiveness. Consider *map* [15]—a common parallelization pattern as shown in Listing 1. Tasks operate on separate chunks of data in the map pattern. We execute the map program using two different strategies. Data is uniformly distributed to all home caches and work-stealing scheduling is used to assign tasks to idle cores in the first strategy. Data is distributed *per-allocation* to separate home caches and locality-aware scheduling is used to assign tasks to cores such that data is supplied by the local home cache in the second strategy.

The top and bottom graphs in the first column of Figure 10 show performance of the map program under the two strategies, respectively. The second strategy outperforms the first. Tasks execute faster under locality-aware scheduling since data is supplied by the local home cache. Selectively assigning home caches to cache lines rather than uniformly distributing them is beneficial for locality-aware scheduling. Task performance suffers from nonuniform home cache access latencies due to uniform data distribution and work-stealing scheduling.

We conclude that overheads from memory access latencies are significant in OpenMP programs. Proper choice of data distribution and scheduling is crucial for performance.

```
/* Allocate and initialize data */
for(int i=0; i<N; i++) {
     list[i] = malloc(sizeof(int) * SZ);
     initialize(i, list[i], SZ);
}
/* Work in parallel */
for(int i=0; i<N; i++) {
#pragma omp task input(list[i][0:SZ-1])
     map(list[i], SZ);
}
#pragma omp taskwait
```

Listing 1: Parallel map implemented using OpenMP tasks.

Our goal is to provide simple and portable abstractions that minimize memory access overheads by performing data distribution and enabling scheduling that can exploit locality arising from data distribution.

3. Runtime System Assisted Data Distribution

Runtime system assisted data distribution is one mechanism for increasing performance portability. Handling specific OS and hardware details can be delegated to an architecture-specific runtime system which has a global view of program execution.

We propose a memory allocation and distribution mechanism controlled by a simple data distribution policy that is chosen by the programmer. The distribution policy choice is deliberately kept simple with only a few choices in order to provide predictable behavior and be easy to understand for the programmer, just like process binding hints in OpenMP are defined. There are two different policies available to the programmer as shown in Table 1. *Unit* and *location* abstractions used in policy descriptions are explained in Table 2.

We demonstrate how the data distribution policies work and propose preliminary interfaces for policy selection using an example program in Listing 2. The program makes memory allocation requests A–E which span eight units of memory. Requests A and B use a proposed interface called `omp_malloc` whose signature is similar to `malloc`. The user selects the data distribution policy for requests A and B by setting a proposed environment variable called OMP_DATA_DISTRIBUTION to one of `standard`, `fine`, or `coarse` prior to the program invocation. The `standard` data distribution policy choice refers to the machine default—first-touch for NUMA systems and uniform distribution for TILEPro64. Memory requested using `omp_malloc` is distributed to different locations based on the global data distribution policy selected. Requests C–E use `omp_malloc_specific`—an extension of `omp_malloc`—to override the global policy and distribute specifically instead. Machine level results of policy actions are shown in Figure 6.

We provide heuristics in Table 3 to assist in the choice of data distribution policy. The heuristics are based on the number of data allocations through `malloc` in the original program and the number of tasks operating on those allocations. Programs with many tasks and a single malloc call will benefit from using the fine policy since cores can issue multiple outstanding requests to different nodes/home caches. Programs with a single task and many malloc calls can use the coarse policy to improve bandwidth since memory is likely to be fetched from different network links. Programs with many malloc calls and many tasks that operate on allocated data are likely to improve performance with the coarse policy due to reduced network contention assuming tasks work on allocations in isolation.

Table 3: Heuristics to select data distribution scheme.

		Number of tasks operating on data	
		One	Many
Number of `malloc` calls	One	Regular `malloc`	Fine
	Many	Coarse	Coarse

We have built the runtime system assisted distribution scheme using readily available support—libnuma on NUMA systems and special allocation interfaces on the TILEPro64. The overhead of the distribution scheme is low since our implementation wraps the system API with a few additional book-keeping instructions. The book-keeping instructions track the round-robin node selection counter for the coarse distribution policy and cache location affinity of data when requested by the locality-aware scheduling policy described in Section 4.

Programmers do not need to be concerned about NUMA node/home cache identifiers and topology in order to use our data distribution scheme. The distribution policy choice is kept simple with only those choices that are easy to predict and understand for the programmer. Programmers can also incrementally distribute data by targeting specific memory allocation sites. We also provide precise control for expert programmers in our implementation allowing them to override the global data distribution policy and request fine or coarse data distribution for a specific allocation. We have implemented two simple distribution policies to demonstrate the potential of our data distribution scheme. Runtime system developers can use the extensibility of our

```
int main(...) {
    ...
    /* Allocate data */
    size_t sz = 8 * UNIT_SIZE;
    /* NUMA system: UNIT_SIZE = PAGE_SIZE */
    /* TILEPro64: UNIT_SIZE = PAGE_SIZE/CACHE_LINE_SIZE */
    void* A = omp_malloc(sz);
    void* B = omp_malloc(sz);
    void* C = omp_malloc_specific(sz, OMP_MALLOC_COARSE);
    void* D = omp_malloc_specific(sz, OMP_MALLOC_COARSE);
    void* E = omp_malloc_specific(sz, OMP_MALLOC_FINE);
    /* Initialize data */
    init(A, B, C, D, E, sz, ...);
    /* Work in parallel */
    #pragma omp parallel
    {
    ...
}
...
}
$ export OMP_DATA_DISTRIBUTION=<standard | fine | coarse>
$ <invoke program>
```

Listing 2: Program using the proposed interface for selecting data distribution policies.

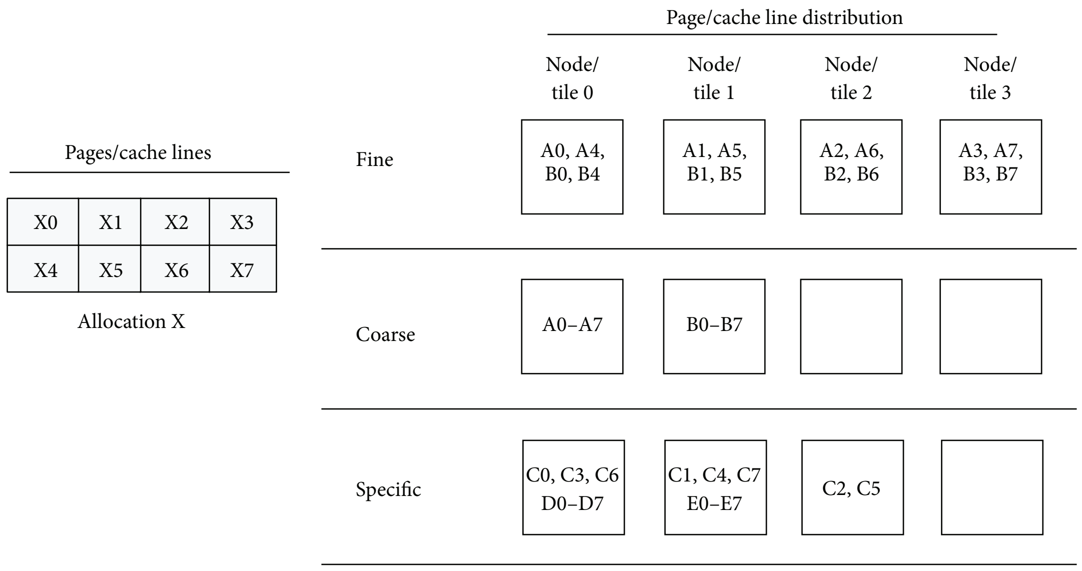


Figure 6: Data distribution results on an example four-node/four-tile ma-chine. We simplify illustration by using eight cache lines per page. In reality, over 64 cache lines typically make up a page.

scheme to provide more advanced distribution policies as plug-ins. Programmers can be educated about distribution policies in a manner similar to existing education about `for`-loop scheduling policies within the OpenMP specification.

4. Locality-Aware Task Scheduling

Our implementation of locality-aware scheduling aims to further leverage the performance benefits of data distribution. The main idea behind our locality-aware scheduler is to schedule tasks to minimize memory access latencies. The locality-aware scheduler uses an architecture-specific task queue organization and takes locality-aware decisions both during work-dealing and work-stealing. Work-dealing refers to actions taken at the point of task creation and work-stealing is actions taken when threads are idle.

Knowing the data footprint of tasks is crucial for the scheduler we expect data footprint information to come from the programmer through task definition clauses which do not yet exist in the OpenMP specification. We currently estimate the data footprint of each task through the information provided by the `depend` clause in the OpenMP 4.0 specification. The estimate is fragile when programmers specify an incomplete depend clause that is sufficient for scheduling decisions but underestimate the data footprint. The limitation can be overcome if programmers use low-effort expressive constructs such as array-sections to express a large fraction of the data footprint in the depend clause in return for improved performance.

The locality-aware scheduler binds task queues to architectural locations to which data can be distributed. There is a task queue per NUMA node on NUMA systems and per home cache on the TILEPro64. Tasks are added at the front and removed from the back of task queues. The scheduler binds one thread to each core.

4.1. NUMA Systems. We describe the work-dealing algorithm of the locality-aware scheduler in Algorithm 1. The scheduler deals a task at the point of task creation to the node queue having the least total memory access latency for pages not in the last-level cache (LLC). The individual access latencies are computed by weighting NUMA node distances with the node-wise distribution D of the data footprint of the task.

NUMA node distances are obtained from OS tables which are cached by the scheduler for performance reasons. The distribution D is calculated using page locality information cached by the data distribution mechanism. The complexity of the access cost computation is $O(N^2)$ where N is the number of NUMA nodes in the system, typically a small number.

Tasks are immediately added to the local queue when scheduling costs outweigh the performance benefits decided by two thresholds. The first threshold—$Sum(D) > sizeof(LLC)/C$—ensures that tasks have a working set size exceeding the LLC size per core. The second threshold—$Standard\_Deviation(D) > 0$—ensures that scheduling effort will not be wasted on tasks with a perfect data distribution.

Distributed task queues may lead to load-imbalance and in our experience the performance benefits from load-balancing often trumps those from locality. We have therefore implemented a work-stealing algorithm to balance the load. Stealing is still preferred over idle threads although cycles spent dealing tasks are wasted.

We show the stealing algorithm of the scheduler in Algorithm 2. Threads attempt to steal when there is no work in the local queue. Candidate queues for steals are ranked based on NUMA node distances. The algorithm includes a threshold which prevents tasks from being stolen from nearly empty task queues which would incur further steals for threads in the victim node. There is an exponential back-off for steal attempts when work cannot be found.

4.2. Manycore Processors. We describe the work-dealing algorithm of the locality-aware scheduler in Algorithm 3. The scheduler deals a task to the home cache queue having the least total memory access latency for cache lines not in the private L1 cache. The individual access latencies are computed by weighting home cache access latencies with the home-cache distribution D of the data footprint of the task. Home cache access latencies are calculated by benchmarking the interconnection during runtime system initialization. The scheduler avoids recalculation by saving latencies across runs. The distribution D is calculated using home cache locality information cached by the data distribution mechanism. The complexity of the access cost computation is $O(N^2)$ where N is the number of home caches in the system.

Tasks are immediately added to the local queue if scheduling costs outweigh the performance benefits. The algorithm ignores distribution policies which potentially distribute data finely to all home caches (condition $p == coarse$). Only tasks with a working set exceeding the L1 data cache are analyzed (condition $sum(D) > sizeof(L1)$).

Another condition—*exists a*—minimizes scheduling effort by using programmer information about the access intensity to data dependences in the list T.depend_list. The index a denotes the most intensely accessed data dependence in the list. The scheduler queues tasks with intensity information in the queue associated with the home cache containing the intensively accessed dependence. Note that we rely on a custom clause to indicate intensity since existing task definition clauses in OpenMP do not support the notion.

We have implemented a work-stealing algorithm to balance load on task queues. Queues are grouped into fixed size vicinities and idle threads are allowed to steal tasks from queues in the same vicinity. Cross-vicinity steals are forbidden. Threads additionally back off when work cannot be found. The size of the vicinity is selected by the programmer prior to execution. We allow vicinity sizes of 1, 4, 8, 16, and 63 tiles in our implementation as shown by tile groups in Figure 4. A vicinity size of 1 only includes the task queue of the member thread; vicinity size of 63 includes task queues of all threads.

5. Experimental Setup

We evaluated data distribution assistance and locality-aware scheduling using benchmarks described in Table 4. The benchmarks were executed using MIR, a task-based runtime system library which we have developed. MIR supports the OpenMP tied tasks model and provides hooks to add custom scheduling and data distribution policies which allows us to compare different policies within the same system. We programmed the evaluation benchmarks using the runtime system interface directly since MIR does not currently have a source-to-source translation front-end.

We ran each benchmark in isolation 20 times for all valid combinations of scheduling and data distribution policies. We recorded the execution time of the critical path of the parallel section and collected execution traces and performance counter readings on an additional set of runs for detailed analysis.

We used a work-stealing scheduler as the baseline for comparing the locality-aware scheduler. The work-stealing scheduler binds one thread to each core and uses one task queue per core. The task queue is the lock-free dequeue by Chase and Lev [16]. The implementation is an adaption of the queue from the Glasgow Haskell Compiler version 7.8.3

Table 4: Pattern-based [15] and real-world benchmarks.

Benchmark	Behavior	Data distribution heuristic guidance
Map (pattern-based)	1D vector scaling	Coarse
Reduction (pattern-based)	Iterative implementation of merge phase of BOTS Sort	Fine
Vecmul	Vector cross product	Coarse
Matmul	Blocked matrix multiplication with BLAS operations in task computation	Coarse
Jacobi	Blocked 2D heat equation solver	Fine
SparseLU	LU factorization of sparse matrix. Derived from BOTS SparseLU.	Coarse

runtime system. Each thread adds newly created tasks to its own task queue. Threads look for work in their own task queue first. Threads with empty task queues select victims for stealing in a round-robin fashion. Both queuing and stealing decisions of the work-stealing scheduler are fast but can result in high memory latencies during task execution since the scheduling is oblivious to data locality and NUMA node/remote cache access latencies.

5.1. NUMA System. We used the Opteron 6172 processor based eight-NUMA node system described in Section 2 for evaluation. Both runtime system and benchmarks were compiled using the Intel C compiler v13.1.1 with -O3 optimization. We used per-core cycle counters and dispatch stall cycle counters to, respectively, measure execution time and memory access latency of tasks.

5.2. Manycore Processor. Both runtime system and benchmarks were compiled using the Tilera GNU compiler with -O3 optimization. We used integer versions of evaluation benchmarks to rule out effects of slow software-emulated floating-point operations. Benchmark inputs were selected to minimize off-chip memory access. We also minimized the adverse effect of evicting local home cache entries to memory by disabling local L2 (inclusive) caching. We used per-core cycle counters and data cache stall cycle counters to, respectively, measure execution time and memory access latency of tasks.

The locality-aware scheduler avoids long home cache access latencies. The L1 cache also mitigates the impact of long home cache access latencies. We separated effects of locality-aware scheduling by disabling L1 caching in previous work [9] but enabled L1 caching in the current setup for a more realistic scenario.

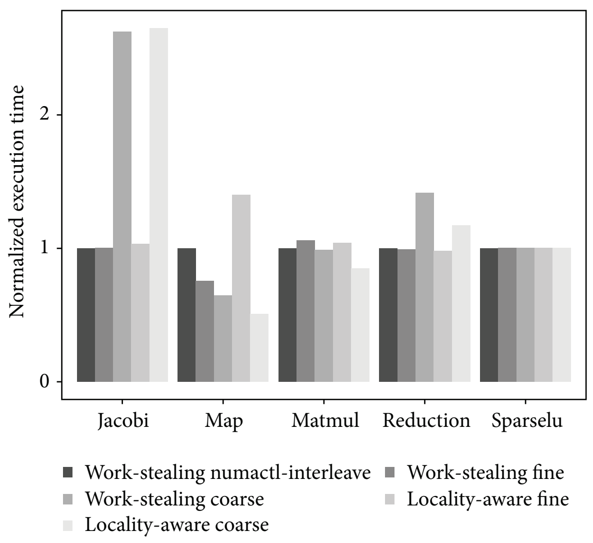


Figure 7: Performance of data distribution combined with work-stealing and locality-aware scheduling on eight-node Opteron system. Execution time is normalized to performance of work-stealing with memory page interleaving using `numactl` for each benchmark. Inputs to Map: 48 floating-point vectors, 1 MB each; Jacobi: 16384 × 16384 floating-point matrix and block size = 512; Matmul: 4096 × 4096 floating-point matrix and block size = 128; SparseLU: 8192 × 8192 floating-point matrix and block size = 256; Reduction: 256 MB floating-point array and depth = 10. Combination of numactl page-wise interleaving and locality-aware scheduling is excluded since the locality-aware scheduler does not currently support querying numactl for page locality information. Locality-aware scheduling, in combination with heuristic-guided data distribution, improves or maintains performance compared to work-stealing.

6. Results

We show performance of evaluation benchmarks for combinations of data distribution and scheduling policy for the eight-node Opteron system in Figure 7. The fine distribution is a feasible replacement for numactl since execution times with the work-stealing scheduler are comparable to page-wise interleaving using numactl. Performance degrades when distribution policies violate the guidelines in Table 3 for both work-stealing and locality-aware schedulers. For example, performance of Matmul degrades when the fine distribution policy is used. The locality-aware scheduler coupled with proper data distribution improves or maintains performance compared to the work-stealing scheduler for each benchmark.

We use thread timelines for Map and Matmul in Figure 8 to explain that reduced memory page access time is the main reason behind the difference in task execution times of the work-stealing and locality-aware scheduler.

The thread timeline indicates time spent by threads in different states and state transition events. Threads are shown on the y-axis, time is shown on the x-axis, and memory access latencies are shown on the z-axis. The z-axis is represented using a linear green to blue gradient which encodes memory

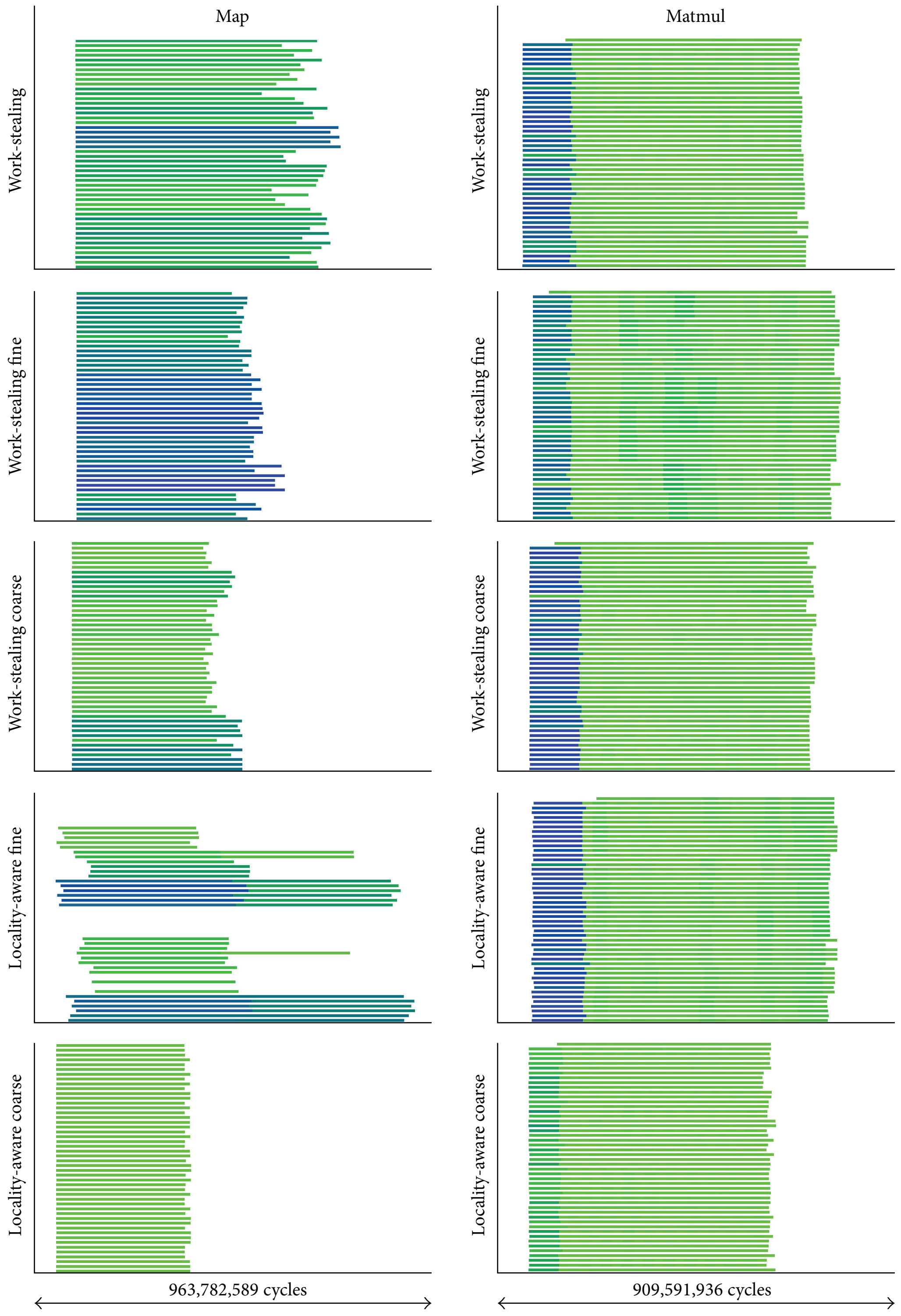


FIGURE 8: Thread timelines showing task execution on the eight-node Opteron system. Threads are shown on the y-axis and time is shown on the x-axis. Memory access latencies are encoded using a green-blue gradient. Tasks stall for fewer cycles under locality-aware scheduling combined with heuristic-guided data distribution.

access latencies measured at state transition boundaries. Green represents lower memory access latencies and blue represents higher ones. We filter out all thread states except task execution. Timelines of a benchmark are time aligned (same x-axis span) and gradient aligned (same z-axis span). Timelines are additionally zoomed-in to focus on task execution and omit runtime system initialization activity.

Understanding benchmark structure is also necessary to explain the performance difference. Each task in the Map benchmark scales a separate vector in a list. Coarse distribution places all memory pages of a given vector in a single node whereas fine distribution spreads the pages uniformly across all nodes.

The locality-aware scheduler combined with coarse distribution minimizes node access latency by ensuring that each task accesses its separate vector from the local node. The behavior can be confirmed by low memory access latencies seen in Figure 8 (light green). The work-stealing scheduler with coarse distribution loses performance due to increased remote memory access latencies as indicated by the relatively higher memory access latencies (dark green and blue).

We can also explain performance of cases that violate the guidelines by using timelines. The locality-aware scheduler with fine distribution detects that pages are uniformly distributed across nodes and places all tasks in the same local queue. The imbalance can not be completely recovered from since steals are restricted. The work-stealing scheduler with fine distribution balances loads more effectively in comparison.

Each task in the Matmul benchmark updates a block in the output matrix using a chain of blocks from two input matrices. Coarse distribution places all memory pages of a given block in a single node whereas fine distribution spreads the pages uniformly across all nodes. The memory pages touched by a task are located on different nodes for both coarse and fine distribution. The locality-aware scheduler with fine distribution detects that data is evenly distributed and falls back to work-stealing by queuing tasks in local queues. Task execute for a longer time with both schedulers as indicated by similar memory access latency (similar intensity of green and blue). However, the locality-aware scheduler with coarse distribution exploits locality arising from distributing blocks in round-robin as indicated by the relatively lower memory access latency (lighter intensity of green and blue) in comparison to the work-stealing scheduler.

We show the performance of evaluation benchmarks for combinations of data distribution and scheduling policy for the TILEPro64 processor in Figure 9. Results are similar to those on the eight-node Opteron system. Performance degrades when distribution policies are chosen against heuristic guidelines in Table 3 for both work-stealing and locality-aware schedulers. The locality-aware scheduler coupled with proper data distribution improves or maintains performance compared to the work-stealing scheduler for each benchmark. Locality-aware scheduler performance is also sensitive to vicinity sizes.

SparseLU is a counter-example whose performance degrades with heuristic-guided coarse distribution and work-stealing scheduling. Performance is also maintained with both coarse and fine distribution on NUMA systems. SparseLU tasks have complex data access patterns which require a data distribution scheme more advanced than fine and coarse.

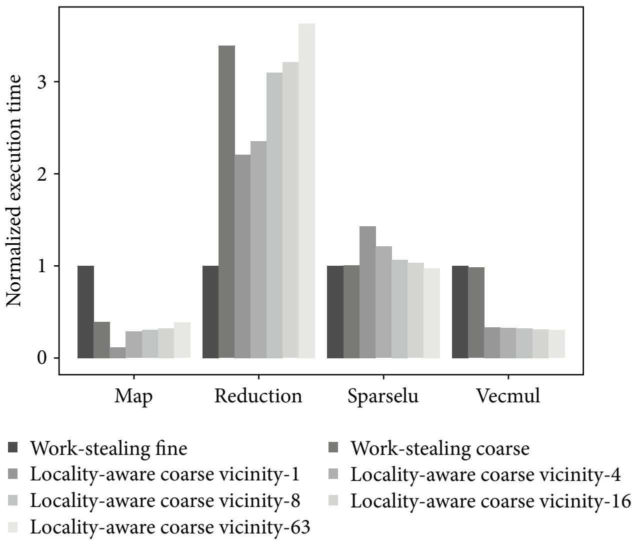


Figure 9: Performance of data distribution combined with work-stealing and locality-aware scheduling on TILEPro64. Execution time is normalized to performance of work-stealing scheduling with fine distribution for each benchmark. Inputs to Map: 63 integer vectors, 32 kB each; Reduction: 700 kB integer array and depth = 6; Vecmul: 128 integer vectors, 28 kB each; SparseLU: 1152 × 1152 integer matrix, block size = 36, and intensity heuristic enabled for tasks executing the *bmod* function. Locality-aware scheduling, in combination with heuristic-guided data distribution, improves or maintains performance compared to work-stealing.

Reduction allocates memory using a single `malloc` call. Coarse distribution is a bad choice since all cache lines are allocated in a single home cache. Locality-aware scheduling serializes execution by scheduling tasks on the core associated with the single home cache. Stealing from larger vicinities balances load to win back performance.

Thread timelines for Map and Vecmul in Figure 10 confirm that reduced cache line access time is the main reason behind the reduction in task execution times. The work-stealing scheduler loses performance by being oblivious to locality despite balancing the load evenly.

We can explain vicinity sensitivity using timelines for Map and Vecmul benchmarks in Figure 11. Increasing vicinity sizes for Map increases the risk of tasks being stolen by threads far from the home cache. Stolen tasks experience large and nonuniform cache line access latencies as shown by long blue bars. Threads fast enough to pick tasks from their own queue finish execution faster. Larger vicinity sizes promote better load-balancing and improve performance in Vecmul.

The locality-aware scheduler can safely be used as the default scheduler for all workloads without performance degradation. There is a performance benefit in using the locality-aware scheduler for workloads which provide strong locality with data distribution. The locality-aware scheduler

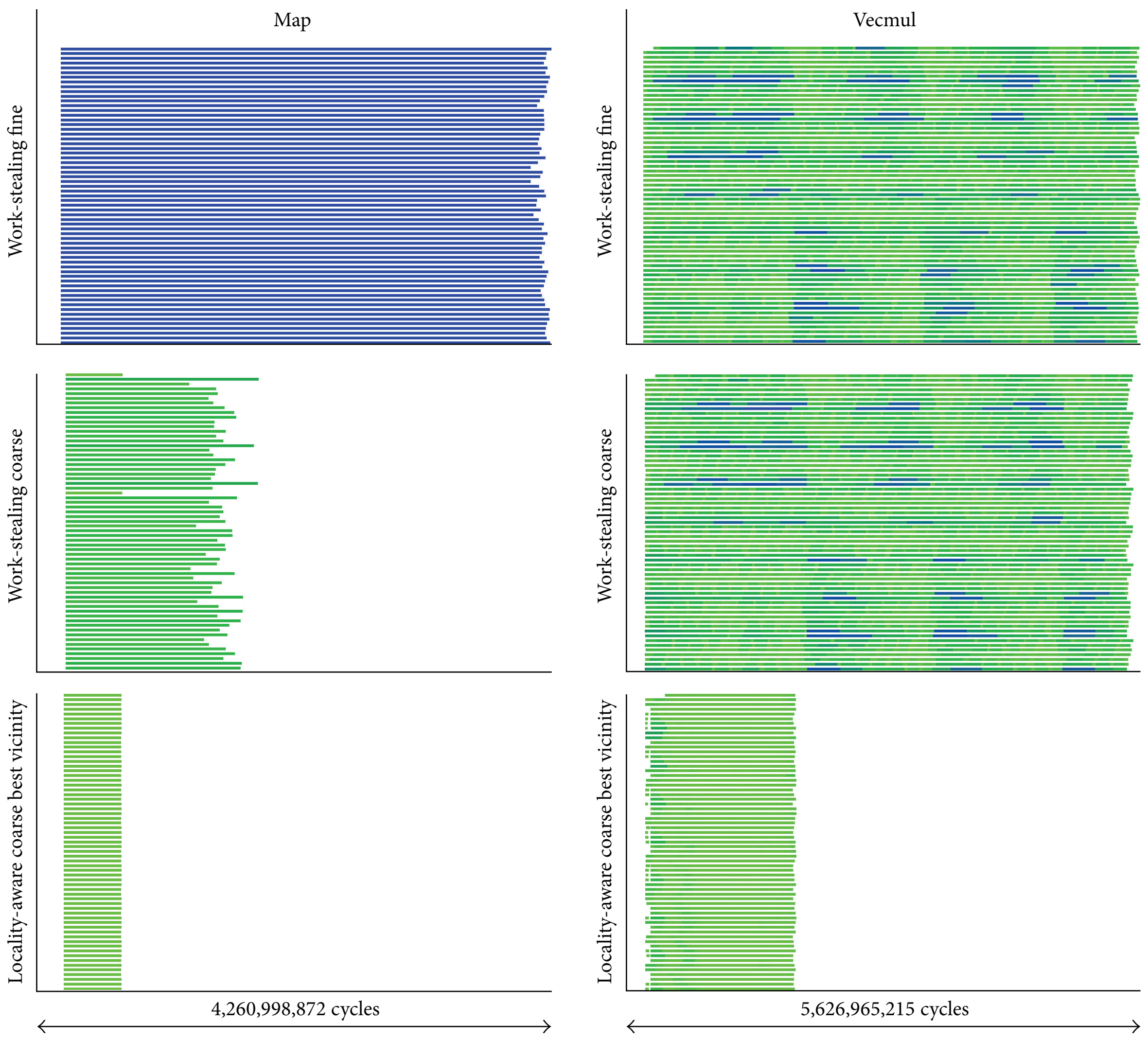


Figure 10: Thread timelines showing task execution on the TILEPro64. Threads are shown on the y-axis and time is shown on the x-axis. Memory access latencies are encoded using a green-blue gradient. Tasks access memory faster under locality-aware scheduling combined with heuristic-guided data distribution.

falls back to load-balancing similar to work-stealing scheduler for workloads which do not improve locality with data distribution.

7. Related Work

Numerous ways of how to distribute data programmatically on NUMA systems have been proposed in the literature. We discuss the proposals that are closest to our approach.

Huang et al. [17] propose extensions to OpenMP to distribute data over an abstract notion of locations. The primary distribution scheme is a block-wise distribution which is similar to our coarse distribution scheme. The scheme allows precise control of data distribution but relies on compiler support and additionally requires changes to the OpenMP specification. Locations provide fine-grained control over data distribution at the expense of programming effort.

The Minas framework [4] incorporates a sophisticated data distribution API which gives precise control on where memory pages end up. The API is intended to be used by an automatic code transformation in Minas that uses profiling information for finding the best distribution for a given program. The precise control is powerful but requires expert programmers who are capable of writing code that will decide on the distribution required.

Majo and Gross [18] use fine-grained data distribution API to distribute memory pages. Execution profiling is used to get data access patterns of loops and used for both guiding code transformation and data distribution. Data distribution is performed in between loop iterations which guarantee that each loop iteration accesses memory pages locally.

Runtime tracing techniques that provide automatic page migration based on hardware monitoring through performance counters have the same end goal as we do: to provide good performance with low programming effort. Nikolopoulos et al. [19] pioneered the idea of page migration with user-level framework. Page access is traced in the background and *hot* pages are migrated closer to the accessing node.

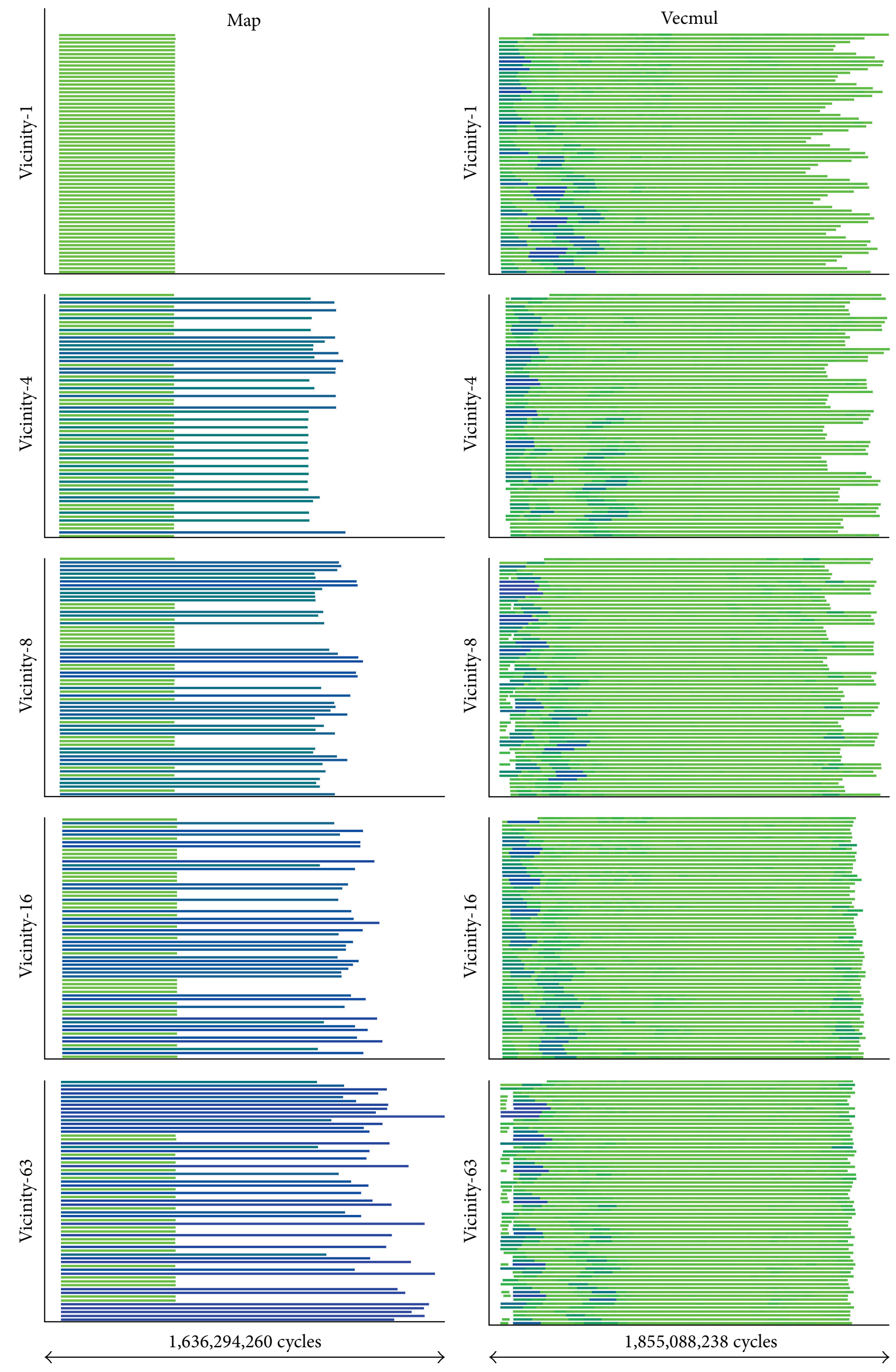


Figure 11: Vicinity size sensitivity on the TILEPro64. On each thread timeline, threads are shown on the y-axis and time is shown on the x-axis. Memory access latencies are encoded using a green-blue gradient. Increasing vicinity size improves load balance but adversely affects memory access time.

Terboven et al. [20] presented a next-touch dynamic page migration implementation on Linux. An alternative approach to page migration, which is expensive, is to move threads instead, an idea exploited by Broquedis et al. [21] in a framework where decisions to migrate threads and data are based on information about thread idleness, available node memory, and hardware performance counters. Carrefour is a modification of the Linux kernel that targets traffic congestion for NUMA systems through traffic management by page replication and page migration [22]. One advantage of the approach is that performance will improve without having to modify applications.

Dynamic page migration requires no effort from the programmer, which is a double edged sword. The benefit of getting good performance without any effort is obvious, but when the programmer experiences bad performance it is difficult to analyze the root cause of the problem. Performance can also be affected by input changes. Attempts at reducing the cost of page migration by providing native kernel support give promising results for matrix multiplication on large matrices [23].

Locality-aware scheduling for OpenMP has been studied extensively. We focus on other task-based approaches since our approach is based on tasks.

Locality domains where programmers manually place tasks in abstract bins have been proposed [1, 24]. Tasks are scheduled within their locality domain to reduce remote memory access. MTS [25] is a scheduling policy structured on the socket hierarchy of the machine. MTS uses one task queue per socket which is similar to our task queue per NUMA node. Only one idle core per socket is allowed to steal bulk work from other sockets. Charm++ uses NUMA topology information and task communication information to reduce communication costs between tasks [26]. Chen et al. [27] reduce performance degradation from cache pollution and stealing tasks across sockets in multisocket systems by memory access aware task-graph partitioning.

Memphis uses hardware monitoring techniques and provides methods to fix NUMA problems on general class of OpenMP computations [7]. Monitoring crossbar (QPI) related and LLC cache miss related performance counters is used to measure network activity. Memphis provides diagnostics to the programmer for when to pin threads, distribute memory, and keep computation in a consistent shape throughout the execution. Their recommendations have inspired the design of our locality-aware scheduler and our evaluation methodology.

Liu and Mellor-Crummey [28] add detailed NUMA performance measurement and data distribution guidance capability to HPCToolkit. They report several case studies where coarse (block-wise) distribution improves performance over default policies. Their multiarchitecture tool is a good starting-point for implementing advanced data distribution policies.

Schmidl et al. propose the keywords scatter and compact for guiding thread placement using SLIT-like distance matrices [29]. Our names for data distribution, fine and coarse, are directly inspired by their work.

Task and data affinity mechanisms discussed in our work are greatly inspired by the large body of research on NUMA optimizations for OpenMP runtime systems. The implicit memory allocation and architectural locality based scheduling mechanisms we implemented in the runtime system are inspired by a similar work on NUMA systems by Broquedis et al. [30].

Few works have tackled data distribution and locality-aware scheduling on manycore processors.

Yoo et al. [31] provide an in-depth quantitative analysis of locality-aware scheduling for data-parallel programs on manycore processors. They conclude that work-stealing scheduling cannot capture locality present in data-parallel programs which we also demonstrate through scheduling results for the map program. They propose a sophisticated locality-aware scheduling and stealing technique that maximizes the probability of finding the combined memory footprint of a task group in the lowest level cache that can accommodate the footprint. The technique however requires task grouping and ordering information obtained by profiling read-write sets of tasks and off-line graph analysis.

Vikranth et al. [32] propose to restrict stealing to groups of cores based on processor topology similar to our vicinity-based stealing approach.

Tousimojarad and Vanderbauwhede [33] cleverly reduce access latencies to uniformly distributed data by using copies whose home cache is local to the access thread on the TILEPro64 processor. Zhou and Demsky [2] build a NUMA-aware adaptive garbage collector that migrate objects to improve locality on manycore processors. We target standard OpenMP programs written in C which makes it difficult to migrate objects.

Techniques to minimize cache access latency by capturing access patterns and laying out data both at compile-time and runtime have been proposed for manycore processors. Lu et al. [34] rearrange affine for-loops during compilation to minimize access latency to data distributed uniformly on banked shared caches of manycore processors. Marongiu and Benini [35] extend OpenMP with interfaces to partition arrays which are then distributed by their compiler backend based on profiled access patterns. The motivation for their work is enabling data distribution on MPSoCs without hardware support for memory management. Li et al. [36, 37] use compilation-time information to guide the runtime system in data placement. R-NUCA automatically migrates shared memory pages to shared cache memory using OS support reducing hardware costs for cache coherence [38].

8. Conclusions

We have presented a data distribution and memory page/cache line locality-aware scheduling technique that gives good performance in our tests on NUMA systems and manycore processors. The major benefit is usage simplicity which allows ordinary programmers to reduce their suffering from NUMA effects which hurt performance. Our technique is easy to adopt since it is built using standard components provided by the OS. The locality-aware scheduler can be used as the default scheduler since it will fall back to behaving like a work-stealing scheduler when locality is missing, something also indicated from our measurements.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The work was partially funded by European FP7 project ENCORE Project Grant Agreement no. 248647 and Artemis PaPP Project no. 295440.

References

[1] S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins, "Characterizing and mitigating work time inflation in task parallel programs," in *Proceedings of the 24th International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*, pp. 1–12, Salt Lake City, Utah, USA, November 2012.

[2] J. Zhou and B. Demsky, "Memory management for many-core processors with software configurable locality policies," *ACM SIGPLAN Notices*, vol. 47, no. 11, pp. 3–14, 2012.

[3] F. Broquedis, J. Clet-Ortega, S. Moreaud et al., "Hwloc: a generic framework for managing hardware affinities in HPC applications," in *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '10)*, pp. 180–186, February 2010.

[4] C. P. Ribeiro, M. Castro, J.-F. Méhaut, and A. Carissimi, "Improving memory affinity of geophysics applications on NUMA platforms using minas," in *High Performance Computing for Computational Science—VECPAR 2010*, vol. 6449 of *Lecture Notes in Computer Science*, pp. 279–292, Springer, Berlin, Germany, 2011.

[5] A. Kleen, *A NUMA API for Linux*, Novel, Kirkland, Wash, USA, 2005.

[6] C. Terboven, D. Schmidl, T. Cramer, and D. An Mey, "Assessing OpenMP tasking implementations on NUMA architectures," in *OpenMP in a Heterogeneous World*, vol. 7312 of *Lecture Notes in Computer Science*, pp. 182–195, Springer, Berlin, Germany, 2012.

[7] C. McCurdy and J. S. Vetter, "Memphis: finding and fixing NUMA-related performance problems on multi-core platforms," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '10)*, pp. 87–96, March 2010.

[8] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, "Locality-aware task scheduling and data distribution on NUMA systems," in *OpenMP in the Era of Low Power Devices and Accelerators*, vol. 8122 of *Lecture Notes in Computer Science*, pp. 156–170, Springer, Berlin, Germany, 2013.

[9] A. Muddukrishna, A. Podobas, M. Brorsson, and V. Vlassov, "Task scheduling on manycore processors with home caches," in *Euro-Par 2012: Parallel Processing Workshops*, Lecture Notes in Computer Science, pp. 357–367, Springer, Berlin, Germany, 2013.

[10] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *Proceedings of the International Conference on Parallel Processing (ICPP '09)*, pp. 124–131, Vienna, Austria, September 2009.

[11] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the AMD opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.

[12] D. Molka, R. Schöne, D. Hackenberg, and M. Müller, "Memory performance and SPEC OpenMP scalability on quad-socket x86_64 systems," in *Algorithms and Architectures for Parallel Processing*, vol. 7016 of *Lecture Notes in Computer Science*, pp. 170–181, Springer, Berlin, Germany, 2011.

[13] AMD, BIOS and kernel developer's guide for AMD family 10h processors, 2010.

[14] Tilera, *Tile Processor User Architecture Manual*, 2012, http://www.tilera.com/scm/docs/UG101-User-Architecture-Reference.pdf.

[15] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, Elsevier, 2012.

[16] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '05)*, pp. 21–28, ACM, Las Vegas, Nev, USA, July 2005.

[17] L. Huang, H. Jin, L. Yi, and B. Chapman, "Enabling locality-aware computations in OpenMP," *Scientific Programming*, vol. 18, no. 3-4, pp. 169–181, 2010.

[18] Z. Majo and T. R. Gross, "Matching memory access patterns and data placement for NUMA systems," in *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO '12)*, pp. 230–241, April 2012.

[19] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguade, "Is data distribution necessary in OpenMP?" in *Proceedings of the ACM/IEEE Conference on Supercomputing (CDROM '07)*, p. 47, November 2000.

[20] C. Terboven, D. Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and thread affinity in OpenMP programs," in *Proceedings of the Workshop on Memory Access on Future Processors: A Solved Problem? (MAW '08)*, pp. 377–384, May 2008.

[21] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier, "Dynamic task and data placement over NUMA architectures: an OpenMP runtime perspective," in *Evolving OpenMP in an Age of Extreme Parallelism*, vol. 5568 of *Lecture Notes in Computer Science*, pp. 79–92, Springer, Berlin, Germany, 2009.

[22] M. Dashti, A. Fedorova, J. Funston et al., "Traffic management: a holistic approach to memory placement on NUMA systems," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pp. 381–394, ACM, March 2013.

[23] B. Goglin and N. Furmento, "Enabling high-performance memory migration for multithreaded applications on linux," in *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS '09)*, pp. 1–9, May 2009.

[24] M. Wittmann and G. Hager, "Optimizing ccNUMA locality for task-parallel execution under OpenMP and TBB on multicore-based systems," Computing Research Repository, http://arxiv.org/abs/1101.0093.

[25] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *International Journal of High Performance Computing Applications*, vol. 26, no. 2, pp. 110–124, 2012.

[26] L. L. Pilla, C. P. Ribeiro, D. Cordeiro, and J.-F. Méhaut, "Charm++ on NUMA platforms: the impact of SMP optimizations and a NUMA-aware load balancer," in *Proceedings of the 4th Workshop of the INRIA-Illinois Joint Laboratory on Petascale Computing*, Urbana, Ill, USA, 2010.

[27] Q. Chen, M. Guo, and Z. Huang, "Adaptive cache aware bitier work-stealing in multisocket multicore architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2334–2343, 2013.

[28] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on NUMA architectures," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*, pp. 259–271, ACM, Orlando, Fla, USA, February 2014.

[29] D. Schmidl, C. Terboven, and D. an Mey, "Towards NUMA support with distance information," in *OpenMP in the Petascale Era*, vol. 6665 of *Lecture Notes in Computer Science*, pp. 69–79, Springer, Berlin, Germany, 2011.

[30] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P. Wacrenier, "Dynamic task and data placement over numa architectures: an openmp runtime perspective," in *Evolving OpenMP in an Age of Extreme Parallelism*, vol. 5568 of *Lecture Notes in Computer Science*, pp. 79–92, Springer, Berlin, Germany, 2009.

[31] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, "Locality-aware task management for unstructured parallelism: a quantitative limit study," in *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '13)*, pp. 315–325, ACM, Portland, Ore, USA, July 2013.

[32] B. Vikranth, R. Wankar, and C. R. Rao, "Topology aware task stealing for on-chip NUMA multi-core processors," *Procedia Computer Science*, vol. 18, pp. 379–388, 2013.

[33] A. Tousimojarad and W. Vanderbauwhede, "A parallel task-based approach to linear algebra," in *Proceedings of the IEEE 13th International Symposium on Parallel and Distributed Computing (ISPDC '14)*, pp. 59–66, IEEE, 2014.

[34] Q. Lu, C. Alias, U. Bondhugula et al., "Data layout transformation for enhancing data locality on NUCA chip multiprocessors," in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*, pp. 348–357, IEEE, September 2009.

[35] A. Marongiu and L. Benini, "An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs," *IEEE Transactions on Computers*, vol. 61, no. 2, pp. 222–236, 2012.

[36] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones, "Compiler-assisted data distribution for chip multiprocessors," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*, pp. 501–512, ACM, September 2010.

[37] Y. Li, R. Melhem, and A. K. Jones, "Practically private: enabling high performance CMPs through compiler-assisted data classification," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*, pp. 231–240, ACM, September 2012.

[38] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 184–195, 2009.

Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 621730, 15 pages
http://dx.doi.org/10.1155/2015/621730

Research Article

Multi-GPU Support on Single Node Using Directive-Based Programming Model

Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran, and Barbara Chapman

Department of Computer Science, University of Houston, Houston, TX 77004, USA

Correspondence should be addressed to Rengan Xu; rxu6@uh.edu

Received 15 May 2014; Accepted 29 September 2014

Academic Editor: Xinmin Tian

Existing studies show that using single GPU can lead to obtaining significant performance gains. We should be able to achieve further performance speedup if we use more than one GPU. Heterogeneous processors consisting of multiple CPUs and GPUs offer immense potential and are often considered as a leading candidate for porting complex scientific applications. Unfortunately programming heterogeneous systems requires more effort than what is required for traditional multicore systems. Directive-based programming approaches are being widely adopted since they make it easy to use/port/maintain application code. OpenMP and OpenACC are two popular models used to port applications to accelerators. However, neither of the models provides support for multiple GPUs. A plausible solution is to use combination of OpenMP and OpenACC that forms a hybrid model; however, building this model has its own limitations due to lack of necessary compilers' support. Moreover, the model also lacks support for direct device-to-device communication. To overcome these limitations, an alternate strategy is to extend OpenACC by proposing and developing extensions that follow a task-based implementation for supporting multiple GPUs. We critically analyze the applicability of the hybrid model approach and evaluate the proposed strategy using several case studies and demonstrate their effectiveness.

1. Introduction

Heterogeneous architecture has gained great popularity over the past several years. These heterogeneous architectures are usually comprised of accelerators that are attached to the host CPUs, and such accelerators could include GPUs, DSPs, and FPGA. Although heterogeneous architectures help in increasing the computational power significantly, they also pose potential challenges to programmers before the capabilities of these new architectures could be well exploited. CUDA [1] and OpenCL [2] offer two different interfaces to program GPUs. But in order to perform effective programming using these interfaces, the programmers need to thoroughly understand the underlying architecture and the language/model. This affects productivity. To overcome these difficulties, a number of high-level directive-based programming models have been proposed that include HMPP [3], PGI [4], and OpenACC [5]. These models simply allow the programmers to insert directives and runtime calls into an application code, making partial or full Fortran and C/C++ code portable on accelerators. OpenACC is an emerging interface for parallel programmers to easily create and write simple code that executes on accelerators. In August 2013, the OpenACC standard committee released a second version of the API, OpenACC 2.0. Vendor companies, Cray and PGI, provide compiler support for OpenACC 2.0. CAPS, before they ran out of business, was also providing support for OpenACC 2.0. The model is aiming to port scientific applications to more than one GPU. Several large applications in the fields of geophysics, weather forecast require massive parallel computations and such applications could easily benefit from multiple GPUs. How can we create suitable software that could take advantage of multiple GPUs without losing performance portability? This remains a challenge. In a large cluster, multiple GPUs could reside within a single node or across multiple nodes. If multiple GPUs are used across nodes and each node has one GPU, then the CPU memory associated with the corresponding GPU is independent of the CPU memory associated with another GPU. This makes the communication between the GPUs easier since the CPU memories associated with those GPUs are distributed. However, multiple GPUs could also be in a single node. Since each node in a cluster is usually a shared memory system which has multiple processors, the same shared memory is associated with multiple GPUs. This makes the communication between these GPUs more difficult since they share the same CPU memory and thus the possible memory synchronization issue arises. This paper only focuses on multi-GPU within a single node.

In this paper, we develop strategies to exploit usage of multiple GPUs.

(i) Explore the feasibility of programming multi-GPU using the directive-based programming approaches.

(ii) Evaluate performance obtained by using the OpenMP and OpenACC hybrid model on a multi-GPU platform.

(iii) Propose extensions to OpenACC to support programming multiple accelerators within a single node.

We categorize our experimental analysis into three types: (a) port completely independent routines or kernels to multi-GPU, (b) divide one large workload into several independent subworkloads and then distribute each subworkload to one GPU. In these two cases, there is no communication between the GPUs, and (c) use workload in manner that requires different GPUs to communicate with each other.

The organization of this paper is as follows. Section 2 highlights related work in this area; Section 3 provides an overview of OpenMP and OpenACC directive-based programming models. In Section 4, we will discuss our strategies to develop the hybrid model using OpenMP and OpenACC-based hybrid model and port three scientific applications to multi-GPU within single node with NVIDIA's GPU cards attached. In Section 5 we propose newer extensions to the OpenACC programming model addressing limitations of the hybrid model. Section 6 provides the conclusion of our work.

2. Related Work

Existing research on directive-based programming approach for accelerators focuses on mapping applications to only single GPU. Liao et al. [6] provided accelerator support using OpenMP 4.0 in ROSE compiler. Tian et al. [7] and Reyes et al. [8] developed open source OpenACC compilers, OpenUH, and accULL, respectively. Both their approaches use source-to-source technique to translate OpenACC program to either CUDA program or OpenCL program. Besides compiler development, there is also extensive research [9–11] on the porting applications using OpenACC. These works, however, all utilize only a single GPU, primarily because both OpenACC and OpenMP do not yet provide support for multiple accelerators.

A single programming model may be insufficient to provide support for multiple accelerator devices; however, the user can apply hybrid model to achieve this goal. Hart et al. [12] used Coarray Fortran (CAF) and OpenACC and Levesque et al. [13] used MPI and OpenACC to program multiple GPUs in a GPU cluster. The inter-GPU communication in these works is managed by the Distributed Shared Memory (DSM) model CAF or distributed memory model MPI. Unlike these works, our work uses OpenMP and OpenACC hybrid model. We also develop an approach that extends the OpenACC model to support multiple devices in a single cluster node.

Other works that target multiple devices include OmpSs [14] that allows the user to use their own unique directives in an application so that the program can run on multiple GPUs on either the shared memory system or distributed memory system. StarPU [15] is a runtime library that schedules tasks to multiple accelerators. However, the drawbacks of these approaches are that both OmpSs and StarPU require the user to manually write the kernel that is to be offloaded to the accelerators. Moreover, their approach is not part of any standard thus limiting the usability.

To the best of our knowledge, the only other programming model that supports multiple accelerators without the need to manually write accelerator kernel files is HMPP [3], a directive-based approach. However, HMPP directives are quite challenging in terms of their usability and porting applications to accelerators and even complicated when the applications are large and complex enough.

In this paper, we have adopted a task-based concept by proposing extensions to the OpenACC model to support multiple accelerators. This work is based upon our previous work in [16]. The new work compared to our previous work is the model extension part. Related work that uses tasking concept for GPUs includes Chatterjee et al. [17] who designed a runtime system that can schedule tasks into different Stream Multiprocessors (SMs) in one device. In their system, at a specific time, the device can only execute the same number of thread blocks as the number of SMs (13 in Kepler 20c), thus limiting the performance. This is because their system is designed for tackling load balancing issues among all SMs primarily for irregular applications. Extensions to the OpenACC model were proposed by Komoda et al. [18] to support multiple GPUs. They proposed directives for the user to specify memory access pattern for each data in a computing region and the compiler identifies the workload partition. Typically, it is quite complicated if it is the user that identifies the memory access pattern for all data, especially when a data is multidimensional or accesses multiple index variables. In our extensions to the OpenACC model we allow the user to partition the workload, thus further simplifying the application porting, and make the multi-GPU support general enough to cover most application cases.

3. Overview of OpenACC and OpenMP

OpenMP is a high-level directive-based programming model for shared memory multicore platforms. The model consists of a set of directives, runtime library routines, and environment variables. The user just needs to simply insert the directives into the existing sequential code, with minor changes or no changes to the code. OpenMP adopts the fork-join model. The model begins with an initial main thread, then a team of threads will be forked when the program encounters a parallel construct, and all other threads will join the main thread at the end of the parallel construct. In the parallel region, each thread has its own private variable and does the work on its own piece of data. The communication between different threads is performed by shared variables. In the case of a data race condition, different threads will update the shared variable atomically. Starting from 3.0, OpenMP introduced *task* concept [19] that can effectively express and solve the irregular parallelism problems such as unbounded loops and recursive algorithms. To make the task implementation efficient, the runtime needs to consider the task creation, task scheduling, task switching, task synchronization, and so forth. OpenMP 4.0 released in 2013 includes support for accelerators [20].

OpenACC [5], similar to OpenMP, is a high-level programming model that is being extensively used to port applications to accelerators. OpenACC also provides a set of directives, runtime routines, and environment variables. OpenACC supports three-level parallelism: coarse grain parallelism "gang," fine grain parallelism "worker," and vector parallelism "vector." While mapping the OpenACC three levels of parallelism to the low level CUDA programming model, all of PGI, CAPS [21], and OpenUH [22] map each gang to a thread block, workers to the Y-dimension of a thread block and vector to the X-dimension of a thread block. This guarantees fair performance comparison when using these compilers since they use the same parallelism mapping strategy. The execution model assumes that the main program runs on the host, while the compute-intensive regions of the main program are offloaded to the attached accelerator. In the memory model, usually the accelerator and the host CPU consist of separate memory address spaces; as a result, data being transferred back and forth is an important challenge to address. To satisfy different data optimization purposes, OpenACC provides different types of data transfer clauses in 1.0 specification and possible runtime routines in the 2.0 document. To fully utilize the CPU resource and remove potential data transfer bottleneck, OpenACC also allows asynchronous data transfer and asynchronous computation with the CPU code. Also the model offers an *update* directive that can be used within a data region to synchronize data between the host and the device memory.

4. Multi-GPU Support with OpenMP and OpenACC Hybrid Model

In this section, we will discuss strategies to explore programming multi-GPU using OpenMP and OpenACC hybrid model within a single node. We will evaluate our strategies using three scientific applications. We study the impact of our approach by comparing and analyzing the performances achieved by the hybrid model (multi-GPU implementation) against that of a single GPU implementation.

The experimental platform is a server machine that is a multicore system consisting of two NVIDIA Kepler 20Xm GPUs. The system itself has Intel Xeon x86_64 CPU with 24 cores (12 × 2 sockets), 2.5 GHz frequency, and 62 GB main memory. Each GPU has 5 GB global memory. We use CAPS OpenACC for S3D and PGI OpenACC for matrix multiplication and 2D heat equation. PGI compiler is not used for S3D since it cannot compile the code successfully. The 2D heat equation program compiled by CAPS compiler is extremely long so we do not show the result here. CAPS compiler does compile the matrix multiplication program but we leave the performance comparison with other compilers later and here we only verify the feasibility of hybrid programming model. We use GCC 4.4.7 as CAPS host compiler for all C programs. For a Fortran program, we use PGI and CAPS (pgfortran as the host compiler of CAPS) to compile the OpenACC code. We use the latest versions of CAPS and PGI compilers, 3.4.1 and 14.2, respectively. CUDA 5.5 is used for our experiments. The CAPS compiler performs source-to-source translation of directives inserted code into CUDA code and then calls *nvcc* to compile the generated CUDA code. The flags passed to CAPS compiler are "–nvcc-options -Xptxas=-v,-arch,sm_35,-fmad=false," and the flags passed to PGI compiler are "-O3 -mp -ta=nvidia,cc35,nofma." We consider wall-clock time as the evaluation measurement. We ran all experiments for five times and then averaged the performance results. In the forthcoming subsections, we will discuss both single and multi-GPU implementations for the S3D thermodynamics application kernel, matrix multiplication, and 2D heat equation.

OpenMP is fairly easy to use, since all that the programmer needs to do is to insert OpenMP directives in the appropriate places and, if necessary, make minor modifications to the code. The general idea of an OpenMP and OpenACC hybrid model, as shown in Figure 1, is that we need to manually divide the problem among OpenMP threads and then associate each thread with a particular GPU. The easy case is when the work in each GPU is independent of each other and no communication among different GPUs is involved. But there may be cases where the GPUs will have to communicate with each other and this will involve the CPUs too. Different GPUs transfer their data to their corresponding host threads, these threads then communicate or exchange their data via shared variable, and finally the threads transfer the new data back to their associated GPUs. With the GPU direct technique [23], it is also possible to transfer data between different GPUs directly without going through the host. This has not been plausible in OpenMP and OpenACC hybrid model so far, but in Section 5 we will propose some extensions to the OpenACC programming model to accommodate this feature.

4.1. S3D Thermodynamics Kernel. S3D [24] is a flow solver that performs direct numerical simulation of turbulent combustion. S3D solves fully compressible Navier-Stokes, total energy, species, and mass conservation equations coupled with detailed chemistry. Apart from the governing equations, there are additional constitutive relations, such as the ideal gas equation of state, models for chemical reaction rates, molecular transport, and thermodynamic properties. These relations and detailed chemical properties have been implemented as kernels or libraries suitable for GPU computing. Some research on S3D has been done in [13, 25], but the code they used is not accessible for us. For the experimental purpose of our work, we only chose two separate and independent kernels of the large S3D application, discussed in detail in [26].

```
!$acc data copyout(c(1:np), h(1:np)) copyin(T(1:np),...)
do m = 1, MR
  call calc_mixcp(np, nslvs, T, midtemp, ... , c)
  call calc_mixenth(np, nslvs, T, midtemp, ... , h)
end do
!$acc end data
```

Algorithm 1: S3D thermodynamics kernel in single GPU.

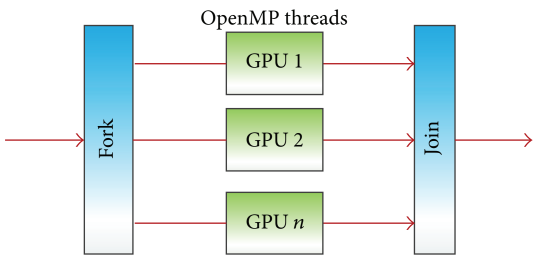


Figure 1: A multi-GPU solution using the hybrid OpenMP and OpenACC model. Each OpenMP thread is associated with one GPU.

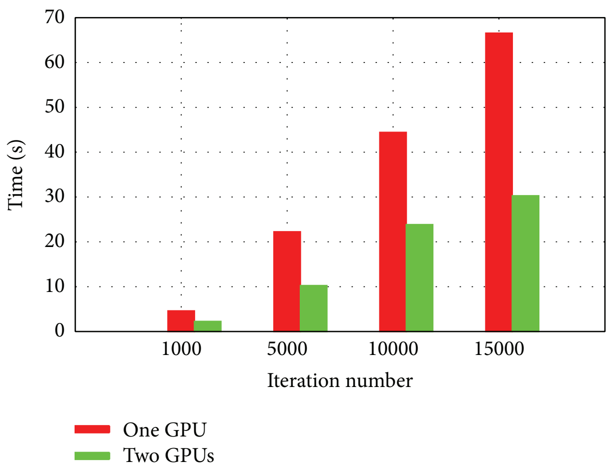


Figure 2: Performance comparison of S3D.

We observed that the two kernels of S3D have similar code structures and their input data are common. Algorithm 1 shows a small code snippet of a single GPU implementation. Both the kernels, *calc_mixcp and calc_mixenth*, are surrounded by a main loop with *MR* iterations. Each kernel produces its own output result, but their results are the same as that of the previous iteration. The two kernels can be executed in the same accelerator sequentially while sharing the common input data, which will stay on the GPU during the whole execution time. Alternatively, they can be also executed on different GPUs simultaneously since they are totally independent kernels.

In order to use multi-GPU, we distribute the kernels to two OpenMP threads and associate each thread with one GPU. Since we have only two kernels, it is not necessary to use *omp for*; instead we use *omp sections* so that each kernel is located in one section. Each thread needs to set the device number using the runtime *acc_set_device_num (int devicenum, acc_device_t devicetype)*. Note that the device number starts from 1 in OpenACC, or the runtime behavior would be implementation-defined if the *devicenum* were to start from 0. To avoid setting the device number in each iteration and make the two kernels work independently, we apply loop fission and split the original loop into two loops. Finally we replicate the common data on both the GPUs. The code snippet in Algorithm 2 shows the implementation for multi-GPU. Although it is a multi-GPU implementation, the implementation in each kernel is still the same as that of a single GPU implementation. Figure 2 shows the performance results of using single GPU and two GPUs. It is observed that two GPUs almost always take approximately half the time taken for a single GPU. This illustrates the performance advantage of using multiple GPUs over single GPU.

4.2. Matrix Multiplication. With S3D, we had distributed different kernels of one application to multiple GPUs. An alternate type of a case study would be where the workload of only one kernel is distributed to multiple GPUs, especially if the workload is very large. We will use square matrix multiplication as an illustration to explore this case study. We chose this application since this kernel is extensively used in numerous scientific applications. This kernel does not comprise complicated data movements and can be parallelized by simply distributing work to different thread. We also noticed a large computation to data movement ratio.

Typically matrix multiplication takes matrix A and matrix B as input and produces matrix C as the output. When multiple GPUs are used, we will use the same amount of threads as the number of GPUs on the host. For instance, if the system has 2 GPUs, then we will launch 2 host threads. Then we partition matrix A in block row-wise which means that each thread will obtain partial rows of matrix A. Every thread needs to read the whole matrix B and produce the corresponding partial result of matrix C. After partitioning the matrix, we use OpenACC to execute the computation of each partitioned segment on one GPU.

Algorithm 3 shows a code snippet for the multi-GPU implementation for matrix multiplication. Here we assume that the number of threads could be evenly divided by the square matrix row size. Since the outer two loops are totally independent, we distribute the *i* loop into all gangs and

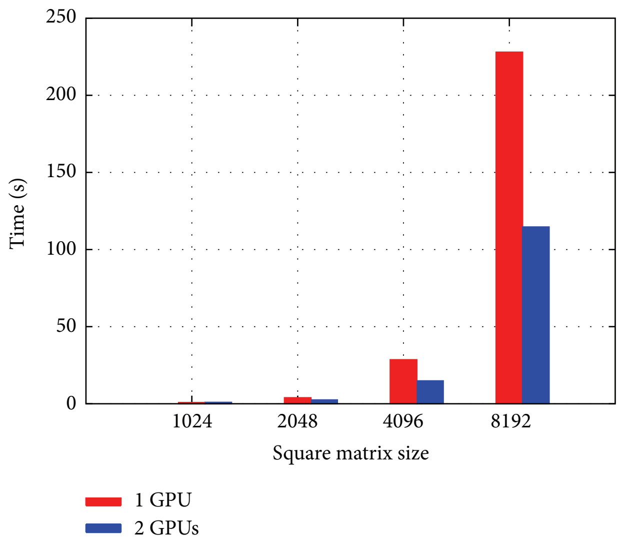
```
call omp_set_num_threads(2)
!$omp parallel private(m)
!$omp sections
!$omp section
call acc_set_device_num(1, acc_device_not_host)
!$acc data copyout(c(1:np)) copyin(T(1:np),...)
do m = 1, MR
  call calc_mixcp(np, nslvs, T, ... , c)
end do
!$acc end data
!$omp section
call acc_set_device_num(2, acc_device_not_host)
!$acc data copyout(h(1:np)) copyin(T(1:np),...)
do m = 1, MR
  call calc_mixenth(np, nslvs, T, ... , h)
end do
!$acc end data
!$omp end sections
!$omp end parallel
```

Algorithm 2: S3D thermodynamics kernel in multi-GPU using hybrid model.

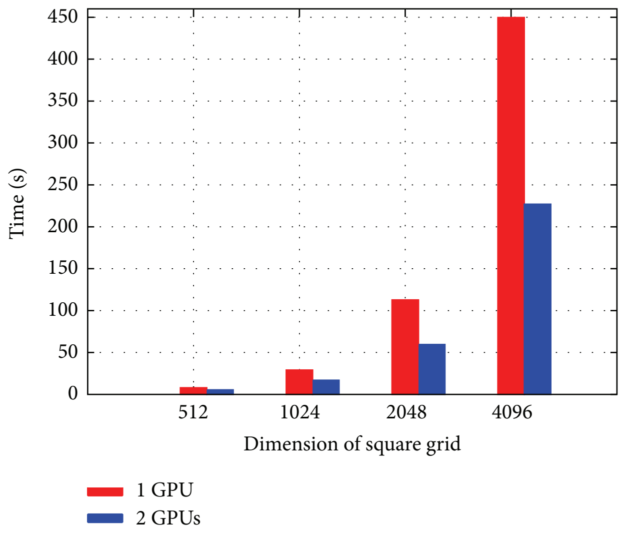
```
omp_set_num_threads(threads);
#pragma omp parallel
{
    int i, j, k;
    int id, blocks, start, end;
    id = omp_get_thread_num();
    blocks = n/threads;
    start = id*blocks;
    end = (id+1)*blocks;
    acc_set_device_num(id+1, acc_device_not_host);
    #pragma acc data copyin(A[start*n:blocks*n])\
                     copyin(B[0:n*n])\
                     copyout(C[start*n:blocks*n])
    {
      #pragma acc parallel num_gangs(32) vector_length(32)
       {
       #pragma acc loop gang
       for(i=start; i<end; i++){
           #pragma acc loop vector
           for(j=0; j<n; j++){
             float c = 0.0f;
             for(k=0; k<n; k++)
                 c += A[i*n+k] * B[k*n+j];
             C[i*n+j] = c;
           }
       }
    }
}
```

Algorithm 3: A multi-GPU implementation of MM using hybrid model.

the j loop into all vector threads of one gang. We have used only 2 GPUs for this experiment; however, more than 2 GPUs can be easily used as long as they are available in the platform. In this implementation, we assume that the number of GPUs can be evenly divided by the number of threads. We use different workload size for our experiments. The matrix dimension ranges from 1024 to 8192. Figure 3(a) shows the performance comparison while using one and two GPUs. For all data size except 1024, the execution time with 2 GPUs is almost half of that with only 1 GPU. For



(a) Matrix multiplication



(b) 2D heat equation

FIGURE 3: Performance comparison using hybrid model.

1024 ∗ 1024 as the matrix size, we barely see any benefit using multiple GPUs. This is possibly due to the overhead incurred due to the creation of host threads and GPU context setup. Moreover, the computation is not large enough for two GPUs. When the problem size is more than 1024, the multi-GPU implementation shows a significant increase in performance. In these cases, the computation is so intensive that the aforementioned overheads are being ignored.

4.3. 2D Heat Equation. We notice that, in the previous two cases, the kernel on one GPU is completely independent of the kernel on the other GPU. Now we will explore a case where there is communication between different GPUs. One such interesting application is 2D heat equation. The formula to represent 2D heat equation is explained in [27] and is given as follows:

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right), \tag{1}$$

where T is temperature, t is time, α is the thermal diffusivity, and x and y are points in a grid. To solve this problem, one possible finite difference approximation is

$$\frac{\Delta T}{\Delta t} = \alpha \left[\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2} \right], \tag{2}$$

where ΔT is the temperature change over time Δt and i, j are indexes in a grid. In this application, there is a grid that has boundary points and inner points. Boundary points have an initial temperature and the temperature of the inner points is also updated. Each inner point updates its temperature by using the previous temperature of its neighboring points and itself. The operation that updates temperature for all inner points in a grid needs to last long enough. This implies that many iterations are needed before arriving at the final stable temperatures. In our program, the number of iterations is 20,000, and we increase the grid size gradually from 512 ∗ 512 to 4096 ∗ 4096. Our prior experience working on single GPU implementation of 2D heat equation is discussed in [11]. Algorithm 4 shows the code snippet for the single GPU implementation. Inside the kernel that updates the temperature, we distribute the outer loop into all gangs and the inner loop into all vector threads inside each gang. Since the final output will be stored in *temp1* after pointer swapping, we just need to transfer this data out to host.

Let us discuss the case where the application uses two GPUs. Algorithm 5 shows the program in detail. In this implementation, *ni* and *nj* are X and Y dimensions of the grid (it does not include boundary), respectively. As shown in Figure 4, we partitioned the grid into two parts along Y dimension and run each part on one GPU. Before the computation, the initial temperature is stored in *temp1_h*, and after updating the temperature, the new temperature is stored in *temp2_h*. Then we swap the pointer so that in the next iteration the input of the kernel points to the current new temperature. Since updating each data point needs its neighboring points from the previous iteration, two GPUs need to exchange the halo data in every iteration. The halo data refers to the data that needs to be exchanged by different GPUs. So far by simply using high-level directives or runtime libraries, data cannot be exchanged directly between different GPUs and the only workaround is to first transfer the data from one device to the host and then from the host to another device. In 2D heat equation, different devices need to exchange the halo data; therefore, the halo data updating would go through the CPU. Because different GPUs use different parts of the data in the grid, we do not have to

```
void step_kernel{...}
{
  #pragma acc parallel present(temp_in[0:ni*nj], temp_out[0:ni*nj]) \
                       num_gangs(32) vector_length(32)
  {
     // loop over all points in domain (except boundary)
     #pragma acc loop gang
     for (j=1; j < nj-1; j++) {
         #pragma acc loop vector
         for (i=1; i < ni-1; i++) {
             // find indices into linear memory
             // for central point and neighbours
             i00 = I2D(ni, i, j);
             im10 = I2D(ni, i-1, j);
             ip10 = I2D(ni, i+1, j);
             i0m1 = I2D(ni, i, j-1);
             i0p1 = I2D(ni, i, j+1);
     // evaluate derivatives
    d2tdx2 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
    d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];
     // update temperatures
    temp_out[i00] = temp_in[i00]+tfac*(d2tdx2 + d2tdy2);
         }
     }
  }
}
#pragma acc data copy(temp1[0:ni*nj]) \
                 copyin(temp2[0:ni*nj])
{
     for (istep=0; istep < nstep; istep++) {
         step_kernel(ni, nj, tfac, temp1, temp2);
         // swap the temp pointers
         temp = temp1;
         temp1 = temp2;
         temp2 = temp;
     }
}
```

Algorithm 4: Single GPU implementation of 2D heat equation.

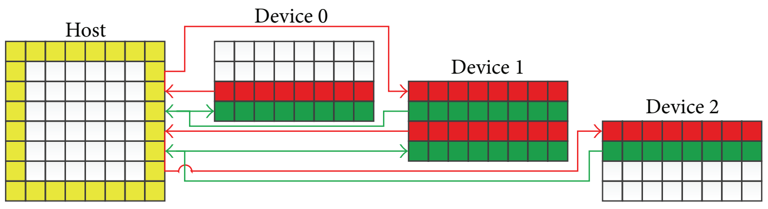


Figure 4: Multi-GPU implementation strategy for 2D heat equation using the hybrid model. Consider that there are 3 GPUs (Devices 0, 1, and 2). The grid in the left has 6 rows (excluding boundaries, i.e., the top and the bottom rows). By splitting the 6 rows into 3 parts, each GPU is expected to compute only 2 rows. However, the computation for a data point requires the value of the neighboring points (top, bottom, left, and right data points); hence, simply considering 2 rows of the grid for 1 GPU is not enough. For GPU Device 0, the last row added already has the left, top, and right data points but lacks data points from the bottom; hence, the bottom row needs to be added, leading to 3 rows in total. For GPU Device 1, the first and the second rows do not have data points from the top and the bottom, respectively, hence requiring an addition of the top and bottom rows. This leads to 4 rows in total. For GPU Device 2, the first row does not have data points from the top and requires the addition of the top row. This leads to 3 rows in total. Another point to note is that values in the rows added need to be updated from other GPUs as indicated by the arrows.

```
omp_set_num_threads(NUM_THREADS);
rows = nj/NUM_THREADS;
LDA = ni + 2;
// main iteration loop
#pragma omp parallel private(istep)
{
  float *temp1, *temp2, *temp_tmp;
  int tid = omp_get_thread_num();
  acc_set_device_num(tid+1, acc_device_not_host);
  temp1 = temp1_h + tid*rows*LDA;
  temp2 = temp2_h + tid*rows*LDA;
  #pragma acc data copyin(temp1[0:(rows+2)*LDA]) \
                   copyin(temp2[0:(rows+2)*LDA])
  {
    for(istep=0; istep < nstep; istep++){
      step_kernel(ni+2, rows+2, tfac, temp1, temp2);
      /* all devices (except the last one) update the lower halo to the host */
      if(tid != NUM_THREADS-1){
        #pragma acc update host(temp2[rows*LDA:LDA])
      }
      /* all devices (except the first one) update the upper halo to the host */
      if(tid != 0){
        #pragma acc update host(temp2[LDA:LDA])
      }
      /*  all host threads wait here to make sure halo data from all devices
         have been updated to the host */
      #pragma omp barrier
      /* update the upper halo to all devices (except the first one) */
      if(tid != 0){
        #pragma acc update device(temp2[0:LDA])
      }
      /* update the lower halo to all devices (except the last one) */
      if(tid != NUM_THREADS-1){
        #pragma acc update device(temp2[(rows+1)*LDA:LDA])
      }
      temp_tmp = temp1;
      temp1 = temp2;
      temp2 = temp_tmp;
    }
    /* update the final result to host */
    #pragma acc update host(temp1[LDA:row*LDA])
  }
}
```

Algorithm 5: Multi-GPU implementation with hybrid model of 2D heat equation.

allocate separate memory for these partial data; instead we just need to use private pointer to point to the different positions of the shared variables *temp1_h* and *temp2_h*. Let *tid* represent the id of a thread; then that thread points to the position $tid * rows * (ni + 2)$ of the grid (because it needs to include the halo region), and it needs to transfer $(rows + 2) * (ni + 2)$ data to the device where rows are equal to $nj/NUM\_THREADS$. The kernel that updates the temperature in the multi-GPU implementation is exactly the same as the one in single GPU implementation.

Figure 3(b) shows the performance comparison of the different implementations, that is, single and multi-GPU implementations. While comparing the performances of multi-GPU with single GPU, we notice that there is a trivial performance difference when the problem size is small. However, there is a significant increase in performance using multi-GPU for larger grid sizes. With the grid size as 4096 * 4096, the speedup of using two GPUs is around 2x times faster than the single GPU implementation. This is because as the grid size increases, the computation also increases significantly, while the halo data exchange is still small enough. Thus, the ratio of the computation/communication becomes larger. Using multi-GPU can be quite advantageous to decompose the computation.

5. Multi-GPU Support with OpenACC Extension

We see that programming using multi-GPU using OpenMP and OpenACC hybrid model shows significant performance benefits in Section 4. However, there are some disadvantages too in this approach. First, the users need to learn two different programming languages which may impact the productivity. Second, in this approach the device-to-device communication happens via the host bringing more unnecessary data movement. Third, providing support for such hybrid model is a challenge for compilers. Compiler A provides support for OpenMP and Compiler B provides support for OpenACC; as a result, it is not straightforward for different compilers to interact with each other. For instance, Cray compiler does not allow OpenACC directives to appear inside OpenMP directives [28]; therefore, the examples in Algorithms 3 and 5 are not compilable by Cray compiler. Although CAPS compiler provides support for OpenACC, it still uses an OpenMP implementation from another host compiler, also a challenge to follow and maintain. Ideally, one programming model should provide support for multi-GPU. Unfortunately the existing OpenACC standard does not yet provide support for multiple accelerator programming. To solve these problems, we propose some extensions to the OpenACC standard in order to support multiple accelerator devices.

5.1. Proposed Directive Extensions. The goal is to help the compiler or runtime realize which device the host will communicate with, so that the host can issue the data movement and kernel launch request to the specific device. The new extensions are described as follows:

(1) `#pragma acc parallel/kernels deviceid` (*scalar-integer-expression*): this is to place the corresponding computing region into a specific device;

(2) `#pragma acc data deviceid` (*scalar-integer-expression*): this is the data directive extension for the structured data region;

(3) `#pragma acc enter/exit data deviceid` (*scalar-integer-expression*); this is the extension for unstructured data region;

(4) `#pragma acc wait device` (*scalar-integer-expression*): this is used to synchronize all activities in a particular device since by default the execution in each device is asynchronous when multiple devices are used;

(5) `#pragma acc update peer to` (*list*) `to_device` (*scalar-integer-expression*) `from` (*list*) `from_device` (*scalar-integer-expression*);

(6) void `acc_update_peer(void* dst, int to_device, const void* src, int from_device, size_t size)`.

The purpose of (5) and (6) is to enable device-to-device communication. This is particularly important when using multiple devices, since in some accelerators device can communicate directly with another device without going through the host. If the devices cannot communicate directly, the runtime library can choose to first transfer the data to a temporary buffer in the host and then transfer it from the host to another device. For example, in CUDA implementation, two devices can communicate directly only when they are connected to the same root I/O Hub. If this requirement is not satisfied, then the data transfer will go through the host. (Note that we believe these extensions will address direct device-to-device communication challenge; however, such direct communication also requires necessary support from the hardware. Our evaluation platform did not fulfill the hardware needs; hence, we have not evaluated the benefit of these extensions quantitatively yet and we will do so as part of the future work.)

5.2. Implementation Strategy. We implement the extensions discussed in Section 5.1 in our OpenUH compiler. Our implementation is based on the hybrid model of pthreads + CUDA. CUDA 4.0 and later versions simplify multi-GPU programming by using only one thread to manipulate multiple GPUs. However, in our OpenACC multi-GPU extension implementation, we use multiple pthreads to operate multiple GPUs and each thread is associated with one GPU. This is because the memory allocation and free operations are blocking operations. If a programmer uses data `copy/copyin/copyout` inside a loop, the compiler will generate the corresponding data memory allocation and transfer runtime APIs. Since the memory allocation is blocking operation, the execution in multiple GPUs cannot be parallel. CUDA code avoids this by allocating memory for all data first and then performs data transfer. In OpenACC, however, this is unavoidable because all runtime routines are generated by the compiler and the position of these routines cannot be randomly placed. Our solution is to create a set of threads and each thread manages the context of one GPU which is shown in Figure 5. This is a task-based implementation. Assume we have n GPUs attached to a CPU host, initially the host creates n threads, and each thread is associated with one GPU. Each thread creates an empty first-in first-out (FIFO) task queue which waits to be populated by the host main thread. Depending on the directive type and deviceid clause in the original OpenACC directive annotated program, the compiler generates the task enqueue request for the main thread. The task here means any command issued by the host and executed either on the host or on the device. For example, memory allocation, memory free, data transfer, kernel launch, and device-to-device communication, all of these, are different task types.

Algorithm 8 includes the definitions of the task structure and the thread controlling a GPU (refer it to GPU thread). The task is executed only by GPU thread. A task could be synchronous or asynchronous to the main thread. In the current implementation, most tasks are asynchronous except device memory allocation because the device address is passed from a temporary argument structure, so the GPU thread must wait for this to finish. Each GPU thread manages a GPU context, a task queue, and some data structures in order to communicate with the master thread. Essentially this is still the master/worker model and the GPU threads are

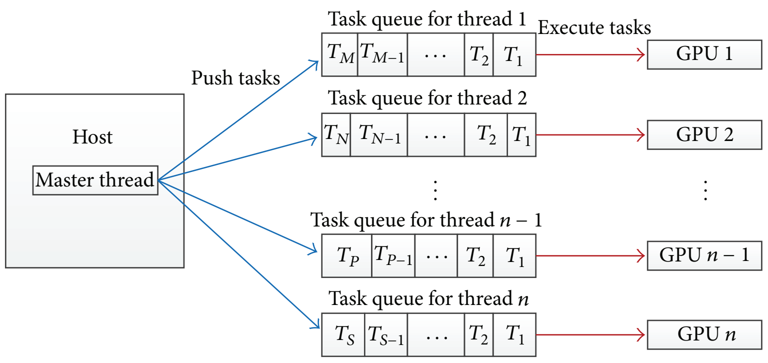

Host
Master thread
Push tasks
Task queue for thread 1
T_M T_{M-1} ... T_2 T_1
Execute tasks
GPU 1
Task queue for thread 2
T_N T_{N-1} ... T_2 T_1
GPU 2
Task queue for thread $n-1$
T_P T_{P-1} ... T_2 T_1
GPU $n-1$
Task queue for thread n
T_S T_{S-1} ... T_2 T_1
GPU n

Figure 5: Task-based multi-GPU implementation in OpenACC. T_i $(i = 1, 2, \ldots)$ means a specific task.

```
(1)  function WORKER_ROUTINE
(2)      Create the context for the associated GPU
(3)      pthread_mutex_lock(···)
(4)      context_created++;
(5)      while context_created! = num_devices do
(6)          pthread_cond_wait(···)   ▷wait until all threads created their contexts
(7)      end while
(8)      pthread_mutex_unlock(···)
(9)      if context_created == num_devices then
(10)         pthread_cond_broadcast(···)
(11)     end if
(12)     Enable peer access among all devices
(13)     while (1) do
(14)         pthread_mutex_lock(&cur_thread → queue_lock)
(15)         while cur_thread → queue_size == 0 do
(16)             pthread_cond_wait(&cur_thread → queue_ready, &cur_thread → queue_lock)
(17)             if cur_thread → destroyed then
(18)                 pthread_mutex_unlock(&cur_thread → queue_lock)
(19)                 Synchronize the GPU context   ▷the context is blocked until the device has
     completed all preceding requested tasks
(20)                 pthread_exit(NULL)
(21)             end if
(22)         end while
(23)         cur_task = cur_thread → queue_head; ▷fetch the task from the queue head
(24)         cur_thread → queue_size−−;
(25)         if cur_thread → queue_size == 0 then
(26)             cur_thread → queue_head = NULL;
(27)             cur_thread → queue_tail = NULL;
(28)         else
(29)             cur_thread → queue_head = cur_task → next;
(30)         end if
(31)         pthread_mutex_unlock(&cur_thread → queue_lock);
(32)         cur_task → routine((void∗)cur_task → args);        ▷execute the task
(33)     end while
(34) end function
```

Algorithm 6: The worker algorithm for multi-GPU programming in OpenACC.

workers. Algorithms 6 and 7 show a detailed implementation for the worker thread and master thread, respectively.

To enable device-to-device communication, we must enable such peer-to-peer access explicitly and this requires that all worker threads have created the GPU contexts. So each worker first creates the context for the associated GPU, and then it waits until all workers have created the GPU contexts. The worker that is the last one to create the context will broadcast to all worker threads so that they can start to enable the peer-to-peer access. The worker then enters

```
(1)  function ENQUEUE_TASK_XXXX
(2)      Allocate memory and populate the task argument
(3)      Allocate memory and populate the task
(4)      pthread_mutex_lock(&cur_thread → queue_lock)
(5)      if cur_thread → queue_size == 0 then      ▹push the task into the FIFO queue
(6)          cur_thread → queue_head = cur_task;
(7)          cur_thread → queue_tail = cur_task;
(8)          pthread_cond_signal(&cur_thread → queue_ready); ▹signal the worker that the queue is not empty and the task is ready
(9)      else
(10)         cur_thread → queue_tail → next = cur_task;
(11)         cur_thread → queue_tail = cur_task;
(12)     end if
(13)     cur_thread → queue_size++;
(14)     pthread_mutex_unlock(&cur_thread → queue_lock);
(15)     if cur_task → async == 0 then              ▹if the task is synchronous
(16)         pthread_mutex_lock(&cur_thread → queue_lock);
(17)         while cur_task → work_done == 0 do  ▹wait until this task is done
(18)             pthread_cond_wait(&cur_thread → work_done, &cur_thread → queue_lock);
(19)         end while
(20)         pthread_mutex_unlock(&cur_thread → queue_lock);
(21)     end if
(22) end function
```

Algorithm 7: The master algorithm for multi-GPU programming in OpenACC.

```
typedef struct _task_s
{
    int type; //task type (e.g. memory allocation and kernel launch, etc.)
    void* (*routine)(void*); // the task routine
    _work_args *args; // point to the task argument
    int work_done; // indicate whether the task is done
    int async; // whether the task is asynchronous
    struct _task_s *next; // next task in the task queue
} _task;
typedef struct
{
    int destroyed; // whether this thread is destroyed
    int queue_size; // the task queue size
    pthread_t thread; // the thread identity
    context_t *context; // the GPU context associated with this thread
    int context_id; // the GPU context id
    _task *queue_head; // head of the FIFO task queue
    _task *queue_tail; // tail of the FIFO task queue
    pthread_mutex_t queue_lock;
    pthread_cond_t queue_ready; // the task queue is not empty and ready
    pthread_cond_t work_done;
    pthread_cond_t queue_empty;
} _gpu_thread;
```

Algorithm 8

an infinite loop that waits for the incoming tasks. When the task is ready in the FIFO task queue, it will fetch the task from the queue head and execute that task. When there is no task available, the worker just goes to sleep to save CPU resource. Whenever a master pushes a task into the FIFO queue of a worker, it will signal to that worker that the queue is not empty and the task is ready. If the master notifies that the worker be destroyed, the worker will complete all pending tasks and then exit (see Algorithm 6).

5.3. Benchmark Example. In this section, we will discuss how to port some of the benchmarks discussed in Section 4 using the OpenACC extensions instead of using the hybrid model. The programs using the proposed directives are compiled

```
for(d=0; d<num_devices; d++)
{
    blocks = n/num_devices;
    start = id*blocks;
    end = (id+1)*blocks;
    #pragma acc data copyin(A[start*n:blocks*n])\
                     copyin(B[0:n*n])\
                     copyout(C[start*n:blocks*n])\
                     deviceid(d)
    {
      #pragma acc parallel deviceid(d)\
            num_gangs(32) vector_length(32)
       {
       #pragma acc loop gang
       for(i=start; i<end; i++){
          #pragma acc loop vector
          for(j=0; j<n; j++){
            float c = 0.0f;
            for(k=0; k<n; k++)
                c += A[i*n+k] * B[k*n+j];
            C[i*n+j] = c;
          }
       }
    }
}
for(d=0; d<num_devices; d++){
    #pragma acc wait device(d)
}
```

Algorithm 9: A multi-GPU implementation of MM using OpenACC extension.

by OpenUH compiler with "-fopenacc -nvcc,-arch=sm_35,-fmad=false" flag. We also compare the performance with that of the CUDA version. All CUDA codes are compiled using "-O3 -arch=sm_35 -fmad=false" flag.

Algorithm 9 shows a code snippet of multi-GPU implementation of matrix multiplication using the proposed OpenACC extensions. Using the proposed approach, the user still needs to partition the problem explicitly into different devices. This is because if there is any dependence between devices, it is difficult for the compiler to do such analysis and manage the data communication. For the totally independent loop, we may further automate the problem partition in compiler as part of the future work. Figure 6 shows the performance comparison using different models. We can see that the performance of manual CUDA version and OpenACC extension version is much better than that of the hybrid model. CAPS compiler seems to have not performed well at all using the hybrid model implementation. The performance of the proposed OpenACC extension version is the best and it is very close to the optimized CUDA code. There are several reasons for this. First, the loop translation mechanisms from OpenACC to CUDA in different compilers are different. Loop translation means the translation from OpenACC nested loop to CUDA parallel kernel. In the translation step, the OpenACC implementation in OpenUH compiler uses redundant execution model which has no synchronization between different OpenACC parallelism like gang and vector. However, PGI compiler uses another execution model which loads some scalar variables into shared memory in gang parallelism and then the vector threads fetch them from shared memory. The detailed comparison between these two loop translation mechanisms is explained in [29]. OpenUH compiler does not need to do those unnecessary shared memory store and load operations and therefore reduces those overhead. Second, we found that CAPS compiler uses similar loop translation mechanism as OpenUH. However, its performance is still worse than OpenUH compiler. The possible reason is that it has nonefficient runtime library implementation. Since CAPS itself does not provide OpenMP support, it needs complex runtime management to interact with the OpenMP runtime in other CPU compilers. This result demonstrates the effectiveness of our approach that not only simplifies the multi-GPU implementation but also maintains high performance.

We also port the 2D heat equation to the GPUs using the proposed OpenACC directive extension. In the code level, the user does not need to make the device-to-device communication go through the host anymore; instead the `update peer` directive can be used to reduce the code complexity and therefore improve the implementation productivity. Algorithm 10 shows the detailed multi-GPU implementation code using the OpenACC extension and Figure 7 explains this implementation graphically. Compared to Figure 4 that uses the hybrid model, it is obvious to see that the data

```
for(d=0; d<num_devices; d++){
    #pragma acc enter data copyin(temp1_h[d*rows*LDA:(rows+2)*LDA]) device(d)
    #pragma acc enter data copyin(temp2_h[d*rows*LDA:(rows+2)*LDA]) device(d)
}
for(istep=0; istep<nstep; istep++){
    for(d=0; d<num_devices; d++)
        step_kernel_(ni+2, rows+2, tfac, temp1_h+d*rows*LDA, temp2_h+d*rows*LDA)
    }
    /* wait to finish the kernel computation */
    for(d=0; d<num_devices; d++){
        #pragma acc wait device(d)
    }
    /* exchange halo data */
    for(d=0; d<num_devices; d++){
        if(d > 0){
          #pragma acc update peer to(temp2_h[d*rows*LDA:LDA]) to_device(d)
                                  from(temp2_h[d*rows*LDA:LDA]) from_device(d-1)
        }
        if(d < num_devices - 1){
        #pragma acc update peer to(temp2_h[(d+1)*rows*LDA+LDA:LDA]) to_device(d)
                           from(temp2_h[(d+1)*rows*LDA+LDA:LDA]) from_device(d+1)
        }
    }
    /* swap pointer of in and out data */
    temp_tmp = temp1_h;
    temp1_h = temp2_h;
    temp2_h = temp_tmp;
}
for(d=0; d<num_devices; d++){
    #pragma acc exit data copyout(temp1_h[(d*rows+1)*LDA:rows*LDA]) deviceid(d)
}
for(d=0; d<num_devices; d++){
    #pragma acc wait device(d)
}
```

ALGORITHM 10: Multi-GPU implementation with OpenACC extension of 2D heat equation.

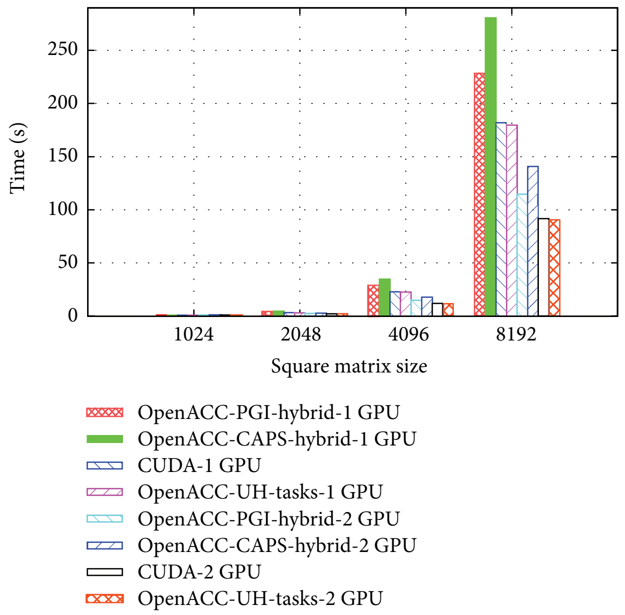


FIGURE 6: Performance comparison for MM using multiple models. PGI and CAPS compilers compile the hybrid model implementation.

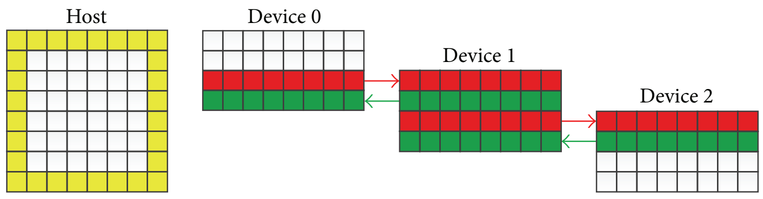


Figure 7: Multi-GPU implementation of 2D heat equation using OpenACC extension.

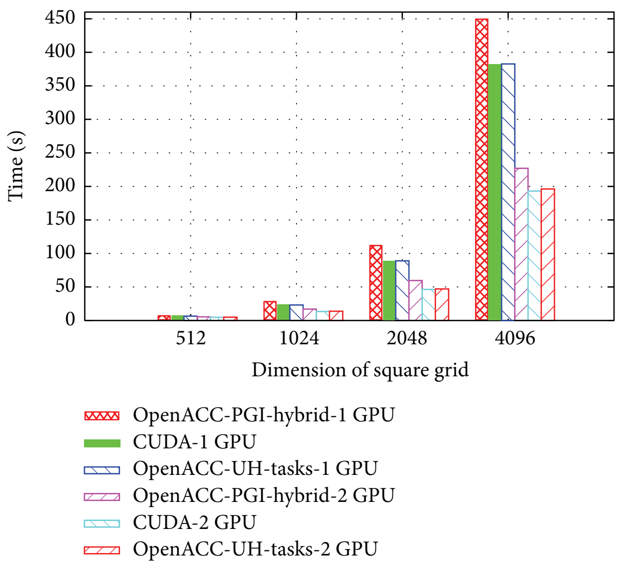


Figure 8: Performance comparison for 2D heat equation using multiple models. PGI compiler compiles the hybrid model implementation.

transfer between devices is much simpler now. Figure 8 shows the performance comparison using different models. The performance of the hybrid model version using CAPS compiler is not shown here because it is too slow, that is, around 5x slower than PGI's performance. When the grid size is 4096 $*$ 4096, the execution time of OpenACC version is around 60 seconds faster than the hybrid model and it is close to that of the optimized CUDA code. We notice that there is almost no performance loss with our proposed directive extension.

6. Conclusion and Future Work

This paper explores the programming strategies of multi-GPU within a single node using the hybrid model, OpenMP and OpenACC. We demonstrate the effectiveness of our approach by exploring three applications of different characteristics. In the first application where there are different kernels, each kernel is dispatched to one GPU. The second application has a large workload that is decomposed into multiple small subworkloads, after which each subworkload is scheduled on one GPU. Unlike the previous two applications that consist of totally independent workloads on different GPUs, the third application has some communication between different GPUs. We evaluated the hybrid model with these three applications on multi-GPU and noticed reasonable performance improvement. Based on the experience gathered in this process, we have proposed some extensions to OpenACC in order to support multi-GPU. By using the proposed directive extension, we can simplify the multi-GPU programming but still obtain much better performance compared to the hybrid model. Most importantly, the performance is close to that of the optimized manual CUDA code.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors would like to thank CAPS for helping them understand its software infrastructure. The authors would also like to thank Oscar Hernandez for his S3D code and Graham Pullan for his lecture notes on heat conduction that helped them understand the application better.

References

[1] CUDA, http://www.nvidia.com/object/cuda_home_new.html.

[2] OpenCL Standard, http://www.khronos.org/opencl.

[3] *HMPP Directives Reference Manual*, (HMPP Workbench 3.1), 2015, https://www.olcf.ornl.gov/wp-content/uploads/2012/02/HMPPWorkbench-3.0_HMPP_Directives_ReferenceManual.pdf.

[4] The Portland Group, *PGI Accelerator Programming Model for Fortran and C*, Version 1.3, The Portland Group, 2010.

[5] OpenACC Standard Home, http://www.openacc-standard.org/.

[6] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman, "Early experiences with the openMP accelerator model," in *OpenMP in the Era of Low Power Devices and Accelerators: Proceedings of the 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16–18, 2013*, vol. 8122 of *Lecture Notes in Computer Science*, pp. 84–98, Springer, Berlin, Germany, 2013.

[7] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman, "Compiling a high-level directive-based programming model for GPG-PUs," in *Languages and Compilers for Parallel Computing: 26th International Workshop, LCPC 2013, San Jose, CA, USA, September 25–27, 2013, Revised Selected Papers*, pp. 105–120, Springer International Publishing, 2014.

[8] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande, "accULL: an OpenACC implementation with CUDA and OpenCL support," in *Euro-Par 2012 Parallel Processing*, vol. 7484

of *Lecture Notes in Computer Science*, pp. 871–882, Springer, Berlin, Germany, 2012.

[9] S. Lee and J. S. Vetter, "Early evaluation of directive-based GPU programming models for productive exascale computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*, pp. 1–11, IEEE Computer Society Press, Salt Lake City, Utah, USA, November 2012.

[10] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC—first experiences with real-world applications," in *Euro-Par 2012 Parallel Processing*, vol. 7484 of *Lecture Notes in Computer Science*, pp. 859–870, Springer, Berlin, Germany, 2012.

[11] R. Xu, S. Chandrasekaran, B. Chapman, and C. F. Eick, "Directive-based programming models for scientific applications—a comparison," in *Proceedings of the SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC '12)*, pp. 1–9, IEEE, Salt Lake City, Utah, USA, November 2012.

[12] A. Hart, R. Ansaloni, and A. Gray, "Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 5–16, 2012.

[13] J. M. Levesque, R. Sankaran, and R. Grout, "Hybridizing S3D into an exascale application using OpenACC: an approach for moving to multi-petaflops and beyond," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*, pp. 1–11, IEEE Computer Society Press, Salt Lake City, Utah, USA, November 2012.

[14] J. Bueno, J. Planas, A. Duran et al., "Productive programming of GPU clusters with OmpSs," in *Proceedings of the IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS '12)*, pp. 557–568, IEEE, May 2012.

[15] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[16] R. Xu, S. Chandrasekaran, and B. Chapman, "Exploring programming multi-GPUs using OpenMP and OpenACC-based hybrid model," in *Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW '13)*, pp. 1169–1176, IEEE, Cambridge, Mass, USA, May 2013.

[17] S. Chatterjee, M. Grossman, A. Sbîrlea, and V. Sarkar, "Dynamic task parallelism with a GPU work-stealing runtime system," in *Languages and Compilers for Parallel Computing*, vol. 7146 of *Lecture Notes in Computer Science*, pp. 203–217, Springer, Berlin, Germany, 2013.

[18] T. Komoda, S. Miwa, H. Nakamura, and N. Maruyama, "Integrating multi-GPU execution in an OpenACC compiler," in *Proceedings of the 42nd Annual International Conference on Parallel Processing (ICPP '13)*, pp. 260–269, IEEE, Lyon, France, October 2013.

[19] E. Ayguadé, N. Copty, A. Duran et al., "The design of OpenMP tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

[20] Technical report on directives for attached accelerators, 2012, http://openmp.org/wp/openmp-specifications/.

[21] *CAPS OpenACC Parallism Mapping*, 2015, http://exxactcorp.com/index.php/software/prod_list/5.

[22] R. Xu, X. Tian, Y. Yan, S. Chandrasekaran, and B. Chapman, "Reduction operations in parallel loops for GPGPUs," in *Proceedings of the Programming Models and Applications on Multicores and Manycores (PMAM '14)*, pp. 10–20, ACM, New York, NY, USA, 2007.

[23] *NVIDIA Kepler GK110 Architecture Whitepaper*, 2014, http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[24] J. H. Chen, A. Choudhary, B. de Supinski et al., "Terascale direct numerical simulations of turbulent combustion using S3D," *Computational Science & Discovery*, vol. 2, no. 1, Article ID 015001, 2009.

[25] K. Spafford, J. Meredith, J. Vetter, J. Chen, R. Grout, and R. Sankaran, "Accelerating S3D: a GPGPU case study," in *Euro-Par 2009—Parallel Processing Workshops*, vol. 6043 of *Lecture Notes in Computer Science*, pp. 122–131, Springer, Berlin, Germany, 2010.

[26] O. Hernandez, W. Ding, B. Chapman, C. Kartsaklis, R. Sankaran, and R. Graham, "Experiences with high-level programming directives for porting applications to GPUs," in *Facing the Multicore—Challenge II*, vol. 7174 of *Lecture Notes in Computer Science*, pp. 96–107, Springer, Berlin, Germany, 2012.

[27] G. Pullan, "Cambridge cuda course 25–27 May 2009," http://www.many-core.group.cam.ac.uk/archive/CUDAcourse09/.

[28] *Cray C and C++ Reference Manual*, 2014, http://docs.cray.com/books/S-2179-81/S-2179-81.pdf.

[29] R. Xu, M. Hugues, H. Calandra, S. Chandrasekaran, and B. Chapman, "Accelerating Kirchhoff migration on GPU using directives," in *Proceedings of the 1st Workshop on Accelerator Programming Using Directives (WACCPD '14)*, pp. 37–46, IEEE, 2014.

Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 859491, 20 pages
http://dx.doi.org/10.1155/2015/859491

Research Article

OpenCL Performance Evaluation on Modern Multicore CPUs

Joo Hwan Lee, Nimit Nigania, Hyesoon Kim, Kaushik Patel, and Hyojong Kim

School of Computer Science, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA

Correspondence should be addressed to Joo Hwan Lee; joohwan.lee@gatech.edu

Received 15 May 2014; Accepted 29 September 2014

Academic Editor: Xinmin Tian

Utilizing heterogeneous platforms for computation has become a general trend, making the portability issue important. OpenCL (Open Computing Language) serves this purpose by enabling portable execution on heterogeneous architectures. However, unpredictable performance variation on different platforms has become a burden for programmers who write OpenCL applications. This is especially true for conventional multicore CPUs, since the performance of general OpenCL applications on CPUs lags behind the performance of their counterparts written in the conventional parallel programming model for CPUs. In this paper, we evaluate the performance of OpenCL applications on out-of-order multicore CPUs from the architectural perspective. We evaluate OpenCL applications on various aspects, including API overhead, scheduling overhead, instruction-level parallelism, address space, data location, data locality, and vectorization, comparing OpenCL to conventional parallel programming models for CPUs. Our evaluation indicates unique performance characteristics of OpenCL applications and also provides insight into the optimization metrics for better performance on CPUs.

1. Introduction

The heterogeneous architecture has gained popularity, as can be seen from AMD's Fusion and Intel's Sandy Bridge [1, 2]. Much research shows the promise of the heterogeneous architecture for high performance and energy efficiency. However, how to utilize the heterogeneous architecture considering performance and energy efficiency is still a challenging problem. OpenCL is an open standard for parallel programming on heterogeneous architectures, which makes it possible to express parallelism in a portable way so that applications written in OpenCL can run on different architectures without code modification [3]. Currently, many vendors have released their own OpenCL framework [4, 5].

Even though OpenCL provides portability on multiple architectures, portability issues still remain in terms of performance. Unpredictable performance variations on different platforms have become a burden for programmers who write OpenCL applications. The effective optimization technique is different depending on the architecture where the kernel is executed. In particular, since OpenCL shares many similarities with CUDA, which was developed for NVIDIA GPUs, many OpenCL applications are not well optimized for modern multicore CPUs. The performance of general OpenCL applications on CPUs lags behind the performance expected by programmers considering conventional parallel programming models. The expectation comes from programmers' experience with conventional programming models. OpenCL applications show very poor performance on CPUs when compared to applications written in conventional programming models.

The reasons we consider CPUs for OpenCL compute devices are as follows.

(1) CPUs can also be utilized to increase the performance of OpenCL applications by using both CPUs and GPUs (especially when a CPU is idle).

(2) Because modern CPUs have more vector units, the performance gap between CPUs and GPUs has been decreased. For example, even for the massively parallel kernels, sometimes CPUs can be better than GPUs, depending on input sizes. On some workloads with high branch divergence or with high instruction-level parallelism (ILP), the CPU can also be better than the GPU.

A major benefit of using OpenCL is that the same kernel can be easily executed on different platforms. With OpenCL,

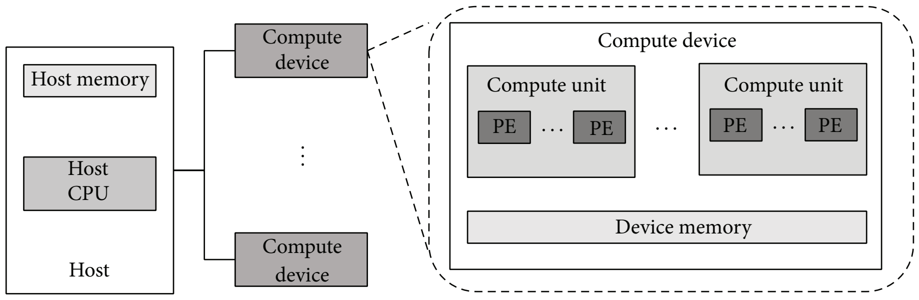


Figure 1: OpenCL platform model.

it is easy to dynamically decide which device to use at run-time. OpenCL applications that select a compute device between CPUs and GPUs at run-time can be easily implemented. However, if the application is written in OpenMP, for example, it is not trivial to split an application to use both CPUs and GPUs.

Here, we evaluate the performance of OpenCL applications on modern out-of-order multicore CPUs from the architectural perspective, regarding how the application utilizes hardware resources on CPUs. We thoroughly evaluate OpenCL applications on various aspects that could change their performance. We revisit generic performance metrics that have been lightly evaluated in previous works, especially for running OpenCL kernels on CPUs. Using these metrics, we also verify the current limitation of OpenCL and the possible improvement in terms of performance. In summary, the contributions of this paper are as follows.

(i) We provide programmers with a guideline to understand the performance of OpenCL applications on CPUs. Programmers can verify whether the OpenCL kernel fully utilizes the computing resources of the CPU.

(ii) We discuss the effectiveness of OpenCL applications on multicore CPUs and possible improvement.

The main objective of this paper is to provide a way to understand OpenCL performance on CPUs. Even though OpenCL can be executed on CPUs and GPUs, most previous work has focused on only GPU performance issues. We believe that our work increases the understandability of OpenCL on CPUs and helps programmers by reducing the programming overhead to implement a separate CPU-optimized version from scratch. Some previous studies about OpenCL on CPUs discuss some aspects presented in this paper, but they lack both quantitative and qualitative evaluations, making them hard to use when programmers want to estimate the performance impact of each aspect.

Section 2 describes the background and architectural aspects to understand the OpenCL performance on CPUs. Then, we evaluate OpenCL applications regarding those aspects in Section 3. We review related work in Section 4 and conclude the paper.

2. Background and Criteria

In this section, we describe the background of several aspects that affect OpenCL application performance on CPUs: API overhead, thread scheduling overhead, instruction-level parallelism, data transfer, data locality, and compiler autovectorization. These aspects have been emphasized in academia and industry to improve application performance on CPUs on multiple programming models. Even though most of the architectural aspects described in this section are well-understood fundamental concepts, most OpenCL applications are not written considering these aspects.

2.1. API Overhead. OpenCL has high overhead for launching kernels, which is negligible on other conventional parallel programming models for CPUs. In addition to the kernel execution on the compute device, OpenCL needs OpenCL API function calls in the host code to coordinate the executions of kernels that are overheads. The general steps of an OpenCL application are as follows [3]:

(1) Open an OpenCL context.

(2) Create a command queue to accept the execution and memory requests.

(3) Allocate OpenCL memory objects to hold the inputs and outputs for the kernel.

(4) Compile and build the kernel code online.

(5) Set the arguments of the kernel.

(6) Set workitem dimensions.

(7) Kick off kernel execution (enqueue the kernel execution command).

(8) Collect the results.

The complex steps of OpenCL applications are due to the OpenCL design philosophy emphasizing portability over multiple architectures. Since the goal of OpenCL is to make a single application run on multiple architectures, they make the OpenCL programming model as flexible as possible. Figure 1 shows the OpenCL platform model and how OpenCL provides portability. The OpenCL platform consists of a host and a list of compute devices. A host is connected to one or more compute devices and is responsible for managing resources on compute devices. The compute device is an abstraction of the processor, which can be any type of processor, such as a conventional CPU, GPU, and DSP. A compute device has a separate device memory and a list of compute units. A compute unit can have multiple processing elements (`PEs`). By this abstraction, OpenCL enables portable execution.

On the contrary, flexibility for various platform supports does not exist on conventional parallel programming models for multicore CPUs. Many of the APIs in OpenCL, which take a significant execution time on OpenCL application do not exist on conventional parallel programming models. The compute device and the context in OpenCL are implicit on conventional programming models. Users do not have to query the platform or compute devices and explicitly create the context.

Another example of the unique characteristics of OpenCL compared to conventional programming models is the "just-in-time compilation" [6] during run-time. In many OpenCL applications, kernel compilation time by the JIT compiler incurs the execution time overhead. On the contrary, compilation is statically done and is not performed during application execution for the application written in other programming models.

Therefore, to determine the actual performance of applications, the time cost to execute the OpenCL API functions also should be considered. From evaluation, we find that the API overhead is larger than the actual computation in many cases.

2.2. Thread Scheduling. Unlike other parallel programming languages such as TBB [7] and OpenMP [8], the OpenCL programming model is a single-instruction and multiple-thread (SIMT) model just like CUDA [9]. An OpenCL kernel describes the behavior of a single thread, and the host application explicitly declares the number of threads to express the parallelism of the application. In OpenCL terminology, a single thread is called a `workitem` (a `thread` in CUDA). The OpenCL programmer can form a set of workitems as a `workgroup` (a `threadblock` in CUDA), where the programmer can synchronize among workitems by `barrier` and `mem_fence`. A single workgroup is composed of a multi-dimensional array of workitems. Figure 2 shows the OpenCL execution model and how an OpenCL kernel is mapped on the OpenCL compute device. In OpenCL, a kernel is allocated on a compute device, and a workgroup is executed on a compute unit. A single workitem is processed by a processing element (`PE`). For better performance, programmers can tune the number of workitems and change the workgroup size.

It is common for OpenCL applications to launch a massive number of threads for kernels expecting speedup by parallel execution. However, portability of OpenCL applications in terms of performance is not maintained on different architectures. In other words, an optimal decision of how to parallelize (partition) a kernel on GPUs does not usually guarantee good performance on CPUs. The partitioning decision of a kernel is done by changing *the number of workitems* and *workgroup size*.

2.2.1. Number of Workitems. First, the number of workitems and the amount of work done by a workitem affect

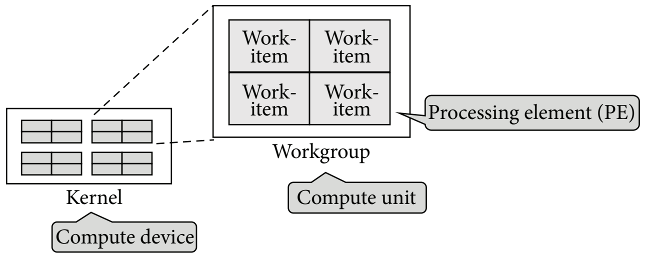


Figure 2: OpenCL execution model.

performance differently on CPUs and GPUs. A massive number of short workitems hurts performance on CPUs but helps performance on GPUs. The performance difference comes from the different architectural characteristics between CPUs and GPUs. On GPUs, a single workitem is processed by a scalar processor (`SP`) or one single SIMD lane. As is widely known, GPUs are specialized for supporting a large number of concurrently running threads, and high thread-level parallelism (`TLP`) is critical to achieve high performance [10–13]. On the contrary, on CPUs, the TLP is limited by the number of cores, so using more threads to do the same amount of work does not help performance on CPUs but hurts it due to the overhead of emulating a large number of concurrently executing workitems on a small number of cores.

2.2.2. Number of Workitems and Instruction-Level Parallelism (ILP). The number of workitems affects the instruction-level parallelism (`ILP`) of the OpenCL kernel on CPUs. Increasing ILP in GPU applications has not been a popular performance optimization technique. The reasons are as follows. First, the hardware can explore much TLP so ILP will not affect the performance significantly. Second, the hardware does not explore too much ILP. The GPU processor is an in-order scheduler processor and does not also support branch prediction to increase ILP. However, on CPUs, the hardware has been designed to increase ILP with multiple features such as superscalar execution and branch predictors.

A modern superscalar processor executes more than one instruction concurrently by dispatching multiple independent instructions during a clock cycle to utilize the multiple functional units in CPUs. Superscalar CPUs use hardware that checks data dependencies between instructions at run-time and schedule instructions to run in parallel [14].

One of the performance problems of OpenCL applications on CPUs is that usually the kernel is written mostly to utilize the TLP, not for ILP. The OpenCL programming model is an SIMT model, and it is common for an OpenCL application to have a massive number of threads. Since independent instructions computing different elements are separated into different threads, most instructions in a single workitem in the kernel are usually dependent on previous instructions, so that typically most OpenCL kernels have `ILP one`; only one instruction can be dispatched to execute in a workitem. On the contrary, on conventional programming models such as OpenMP, independent instructions exist between different loop iterations. For better performance on CPUs, the

OpenCL kernel should be written to have more independent instructions.

2.2.3. Workgroup Size. The second important component is the workgroup size. Workgroup size determines the amount of work in a workgroup and the number of workgroups of a kernel. On GPUs, a workgroup or multiple groups are executed on a streaming multiprocessor (`SM`), which is equivalent to a physical core on the multicore CPU. Similarly, a workgroup is processed by a logical core of the CPU [15, 16]. (Even though it depends on the implementation, many implementations have this characteristic in common.). A workload size that is too small per workgroup makes the scheduling overhead more significant in total execution time on CPUs since the thread context switching overhead becomes bigger.

An OpenCL programmer can explicitly set workgroup size or let the OpenCL implementation decide. If `NULL` value is passed for workgroup size when the host application calls `clEnqueueNDRangeKernel`, the OpenCL implementation automatically partitions global workitems into the appropriate number of workgroups.

2.2.4. Proposed Solutions and Limitations. Many proposals to reduce the scheduling overhead by serialization have been presented [15–17]. Scheduling overhead is not a fundamental problem with the OpenCL programming model. Better OpenCL implementation can have less overhead than other suboptimal implementations. Serialization is a technique that serializes multiple workitems into a single workitem. For example, SnuCL [15] overcomes the overhead of a large number of workitems by serializing them to have fewer workitems in the run-time. However, even with serialization, multiple OpenCL implementations for CPUs still have high scheduling overhead due to the complexity of compiler analysis. Therefore, instead of using many workitems, as is usually the case for OpenCL applications on GPUs, we are better off assigning more work to each workitem with fewer workitems on CPUs. The results from our experiments agree with the above inferences.

2.3. Memory Allocation and Data Transfer. In general, a parallel programming model can have two types of address space options: unified memory space and disjoint memory space [18]. Conventional programming models for CPUs provide the unified memory address space both for the sequential code and for parallel code. The benefit of unified memory space is easy programming, with no explicit data transfer for kernel execution.

On the contrary, even though it is hard for programmers to program, OpenCL provides disjoint memory space to programmers. This is because most heterogeneous computing platforms have disjoint memory systems due to the different memory requirements of different architectures. OpenCL assumes for its target a system where communication between the host and compute devices are performed explicitly by a system network, such as PCI-Express. But, the assumption of discrete memory systems is not true when we use CPUs as compute devices for kernel execution. The host and compute devices share the same memory system resources such as last-level cache, on-chip interconnection, memory controllers, and DRAMs.

The drawback of disjoint memory address space is that it requires the programmer to explicitly manage data transfer between the host and compute devices for kernel execution. In common OpenCL applications, the data should be transferred back and forth in order to be processed by the host or the compute device [3], which becomes unnecessary when we use only the host for computation. To minimize the data transfer overhead on a specific architecture, OpenCL programmers usually have to rewrite the host code [3]. Often, they need to change the memory allocation flags or use different data transfer APIs for performance. For example, the programmer should allocate memory objects on host memory or device memory depending on target platform. These rewriting efforts have been a burden for programmers and have even been a waste of time due to the lack of architectural or run-time knowledge of a specific system in most cases.

2.3.1. Memory Allocation Flags. One of rewriting efforts is changing the memory allocation flag. OpenCL provides the programmer multiple options for memory object allocation flags when the programmer calls `clCreateBuffer` that could affect the performance of data transfer and kernel execution. The memory allocation flag is used to specify how the object is accessed by a kernel and where it is allocated.

Access Type. First, programmers can specify if the memory object is a read-only memory object (`CL_MEM_READ_ONLY`) or write-only one (`CL_MEM_WRITE_ONLY`) when referenced inside a kernel. The programmer can set memory objects used as input to the kernel as read-only and memory objects used as output from the kernel as write-only. If the programmer does not specify access type, the default option is to create a memory object that can be read and written by the kernel (`CL_MEM_READ_WRITE`). `CL_MEM_READ_WRITE` can also be explicitly specified by programmers.

Where to Allocate. The other option that programmers can specify is where to allocate a memory object. When the programmer does not specify allocated location, the memory object is allocated on the device memory in the OpenCL compute device. OpenCL also supports the pinned memory. When the host code creates memory objects using the `CL_MEM_ALLOC_HOST_PTR` flag, the memory object is allocated on the host-accessible memory that resides on the host. Different from allocating the memory object in the device memory, there is no need to transfer the result of kernel execution back to the host memory when the result is required by the host.

2.3.2. Different Data Transfer APIs. OpenCL also provides different APIs for data transfer between the host and compute devices. The host can enqueue commands to read data from an OpenCL memory object that is created by `clCreateBuffer` call to the memory object that is mostly created by `malloc` call in the host memory (by `clEnqueueReadBuffer` API). The host can also enqueue commands to write data to the OpenCL memory object from the memory object in the host memory (by `clEnqueueWriteBuffer` API). The programmer can also map an OpenCL memory object to have the host-accessible pointer of the mapped object (by `clEnqueueMapBuffer` API).

2.4. Vectorization and Thread Affinity

2.4.1. Vectorization. Utilizing SIMD units has been one of the key performance optimization techniques for CPUs [19]. Since SIMD instructions can perform computation on more than one data item at the same time, SIMD utilization could make the application more efficient. Many vendors have released various SIMD instruction extensions on their instruction set architectures, such as MMX [20].

Various methods have been proposed to utilize the SIMD instruction: using optimized function libraries such as Intel IPP [21] and Intel MKL [22], using C++ vector classes with Intel ICC [23], or using DSL compilers such as the Intel SPMD Program Compiler [24]. Programmers can also program in assembly or use intrinsic functions. However, all of these methods assume rewriting the code. Due to this limitation, and to help programmers easily write applications utilizing SIMD instruction, autovectorization has been implemented in many modern compilers [19, 23].

It is quite natural for programmers to expect that a programming model difference has no effect on compiler autovectorization on the same architecture. For example, if a kernel is written in both OpenCL and OpenMP and both implementations are written in a similar manner, programmers would expect that both codes are vectorized in a similar fashion, thereby giving similar performance numbers. Even though it depends on the implementation, this is not usually true. Unfortunately, today's compilers are very fragile about vectorizable patterns, which depend on the programming model. Applications should satisfy certain conditions in order to fully take advantage of compiler autovectorization [19]. Our evaluation in Section 3.5.1 shows an example of this fragility and verifies the possible effect of programming models on vectorization.

2.4.2. Thread Affinity. Where to place threads can affect the performance on modern multicore CPUs. Threads can be placed on each core in different ways, which can create a performance difference. The performance impact of the placement would increase with more processors on the system.

The performance difference can occur for multiple reasons. For example, because of the different latency on the interconnection network, threads that are far away will take longer to communicate with each other, whereas threads close to the adjacent core can communicate more quickly. Also, an application that requires data sharing among adjacent threads can benefit if we assign these adjacent threads to nearby cores. Proper placement can also eliminate the communication overhead by utilizing shared cache. For the performance reason, most conventional parallel programming models support affinity, such as `CPU_AFFINITY` in OpenMP [8].

Table 1: Experimental environment.

CPUs	Intel Xeon E5645
# Cores	4
Vector width	SSE 4.2, 4 single precision FP
Caches	L1D/L2/L3: 64 KB/256 KB/12 MB
FP peak performance	230.4 GFlops
Core frequency	2.40 GHz
DRAM	4 GB
GPUs	NVidia GeForce GTX 580
# SMs	16
Caches	L1/Global L2: 16 KB/768 KB
FP peak performance	1.56 TFlops
Shader Clock frequency	1544 MHz
O/S	Ubuntu 12.04.1 LTS
Platform	Intel OpenCL Platform 1.5 for CPU NVidia OpenCL Platform 4.2 for GPU
Compiler	Intel C/C++ compiler 12.1.3

Unfortunately, thread affinity is not supported in OpenCL. An OpenCL workitem is a logical thread, which is not tightly coupled with a physical thread even though most parallel programming languages provide this feature. The reason for the lack of this functionality is that the OpenCL design philosophy emphasizes portability over efficiency.

We present the lack of affinity support as one of the performance limitations of OpenCL on CPUs compared to other programming languages for CPUs. We would like to present a potential solution to enhance OpenCL performance on CPUs. We found the benefit of better utilizing cache on OpenCL applications by thread affinity. An example is presented in Section 3.5.2.

3. Evaluation

Given the preceding background on the anticipated effects of architectural aspects to understand the OpenCL performance on CPUs, the goal of our study is to quantitatively explore these effects.

3.1. Methodology. The experimental environment for our evaluation is described in Table 1. Our evaluation was performed on a heterogeneous computing platform consisting of a multicore CPU and a GPU; the OpenCL kernel was executed either on the Intel OpenCL platform [4] or the NVidia OpenCL platform [5]. We implemented an execution framework so that we can vary and control many aspects on the applications without code changes. The execution framework is built as an OpenCL delegator library that invokes OpenCL libraries from vendors: the one from Intel for kernel execution on CPUs and the other from NVidia for kernel execution on GPUs.

Table 2: List of NVidia OpenCL benchmarks for API overhead evaluation.

Benchmark
oclBandwidthTest, oclBlackScholes, oclConvolutionSeparable, oclCopyComputeOverlap,
oclDCT8x8, oclDXTCompression, oclDeviceQuery, oclDotProduct, oclHiddenMarkovModel,
oclHistogram, oclMatrixMul, oclMersenneTwister, oclMultiThreads, oclQuasirandomGenerator,
oclRadixSort, oclReduction, oclSimpleMultiGPU, oclSortingNetworks, oclTranspose,
oclTridiagonal, oclVectorAdd

We use different applications for each evaluation. To verify the API overhead, We use NVIDIA OpenCL Benchmarks [5]. For other aspects, including scheduling overhead, memory allocation, and data transfer, we first use simple applications for evaluation. We also vary the data size of each application. The applications are ported to the execution framework we implemented. After evaluation with simple applications, we also use the Parboil benchmarks [25, 26]. Tables 2, 3, and 4 describe evaluated applications and their default parameters.

We use the wall-clock execution time. To measure stable execution time without fluctuation, we iterate the kernel execution until the total execution time of an application reaches a long enough running time, 90 seconds in our evaluation. This is sufficiently long to have a multiple number of kernel executions for all applications in our evaluation. Using the average kernel execution time per kernel invocation calculated, we use normalized throughput to clearly present the performance difference on multiple sections.

3.2. API Overhead. As we discussed in Section 2.1, the OpenCL application has API overhead. To verify the API overhead, we measured the time cost of each API function in executing the OpenCL application in NVIDIA OpenCL Benchmarks [5]. The workload size for each benchmark is the size the application provides as a default. Figure 3 shows the ratio of the execution time of kernel execution and auxiliary API functions to the total execution time of each OpenCL benchmark. (Auxiliary API functions are OpenCL API functions called in the host code to coordinate kernel execution.) The last column `total` means the arithmetic mean of the data from each benchmark. From the figure, we can see that a large portion of execution time is spent on auxiliary API functions instead of kernel execution.

For detailed analysis, we categorized OpenCL APIs into 16 categories. We group multiple categories for visibility in the following. Figure 4 provides a detailed example of API overheads by showing the execution time distribution of each API function category for `oclReduction`. `Enqueued Commands` category includes kernel execution time and data transfer time between the host and compute device and accounts for 12.1% of execution time. We find that the API overhead is larger than the actual computation.

3.2.1. Overhead due to Various Platform Supports. Figure 5 shows the ratio of the execution time of each category to the total execution time of each OpenCL benchmark. The figure shows the performance degradation due to the flexibility of various platforms. We see that the API functions in `Platform`, `Device`, and `Context` categories consume over 80 percent of the total execution time of each OpenCL benchmark on average. The need to call API functions in these categories comes from the fact that each OpenCL application needs to set up an execution environment for which the detailed mechanism would change, depending on the platform. From our evaluation, we also see that each call to the API functions in these categories requires a long execution time. In particular, context management APIs incur a large execution time overhead. Figure 6 shows the execution time distribution of `clCreateContext` and `clReleaseContext` to total execution time in each benchmark. These functions are called at most once on each OpenCL benchmark. But in conventional parallel programming models, context and device are implicit, so there is no need to call such management functions.

3.2.2. Overhead due to JIT Compilation. The list of OpenCL kernels in the application is represented by the `cl_program` object. `cl_program` object is created using either `clCreateProgramWithSource` or `clCreateProgramWithBinary`. JIT compilation is performed by either calling the `clBuildProgram` function or a sequence of `clCompileProgram` and `clLinkProgram` functions for the `cl_program` object to build the program executable for one or more devices associated with the program [3].

JIT compilation overhead is another source of the API overhead. Figure 7 shows the execution time distribution of `Program` category to the sum of execution time of all categories except `Platform`, `Device`, and `Context` categories. We exclude these 3 categories that we have evaluated in previous section. The figure clearly shows the performance degradation due to the JIT compilation. We see that the API functions in `Program` category consume around 33% of the total execution time for 13 categories of the API functions including kernel execution. Execution time overhead of `clBuildProgram` is not negligible in most benchmarks.

Caching. Caching JIT compiled code can help reduce the overhead. Some of the caching ideas are available in OpenCL. Programmers can extract compiled binary by using the `clGetProgramInfo` API function and store it using FILE I/O functions. When the kernel code is not modified since caching, programmers can load the cached binary on disk and use the binary instead of performing JIT compilation on every execution of the application.

3.2.3. Summary. In this section, we can see the high overhead of explicit context management (Section 3.2.1) and JIT compilation (Section 3.2.2) in OpenCL applications. These are unique characteristics of OpenCL compared to conventional programming models for portable execution over multiple architectures.

Table 3: Configurations of simple applications.

Benchmark	Kernel	Global work size	Local work size
Square	Square	10000, 100000, 1000000, 10000000	NULL
Vectoraddition	vectoadd	110000, 1100000, 5500000, 11445000	NULL
Matrixmul	matrixMul	800 × 1600, 1600 × 3200, 4000 × 8000	16 × 16
Reduction	reduce	640000, 2560000, 10240000	256
Histogram	histogram256	409600	128
Prefixsum	prefixSum	1024	1024
Blackscholes	blackScholes	1280 × 1280, 2560 × 2560	16 × 16
Binomialoption	binomialoption	255000, 2550000	255
Matrixmul(naive)	matrixMul	800 × 1600, 1600 × 3200, 4000 × 8000	16 × 16

Table 4: Configurations of the Parboil benchmarks.

Benchmark	Kernel	Global work size	Local work size
CP	Cenergy	64 × 512	16 × 8
MRI-Q	computePhiMag	3072	512
	computeQ	32768	256
MRI-FHD	RhoPhi	3072	512
	computeFHD	32768	256

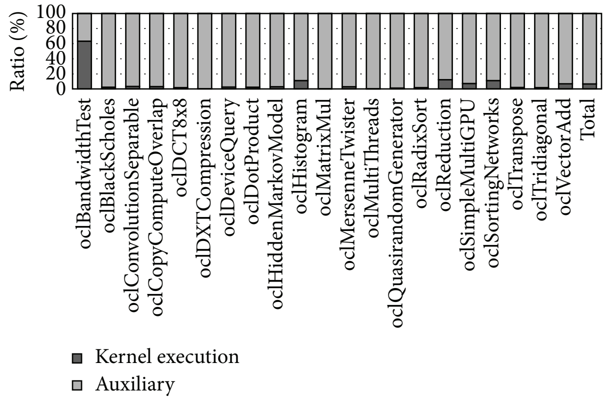


Figure 3: Execution time distribution of kernel execution and auxiliary API functions.

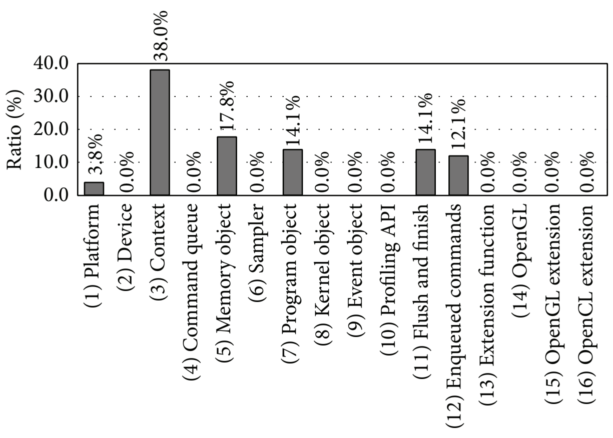


Figure 4: Execution time distribution of each category of API function for `oclReduction`.

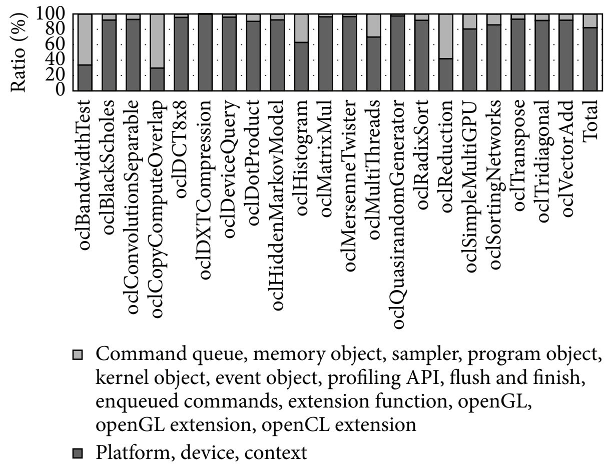


Figure 5: Execution time distribution of each category of API functions.

It should be noted that the workload size for the evaluation in Section 3.2 is the size that the application provides as the default workload size, which is relatively small. Therefore, these overheads can be reduced with a large workload size and thus a long kernel execution time. But it is also true that these overheads are not negligible with small workload size, so the programmer should consider the workload size when they decide whether to use OpenCL or not.

3.3. Thread Scheduling

3.3.1. Number of Workitems. Associated with the discussion in Section 2.2.1, to evaluate the effect of the number of workitems and the workload size per workitem, we perform an experiment on OpenCL applications by allocating more computation per workitem. We coalesce multiple workitems into a single workitem by forming a loop inside the kernel. To keep the total amount of computation the same, we reduce the number of workitems to execute the kernel. The number of workitems coalesced increases from `1` to `1000` workitems by multiplying by `10` for each step. Figure 8 shows the performance of `Square` and `Vectoraddition` applications with a different amount of computation per

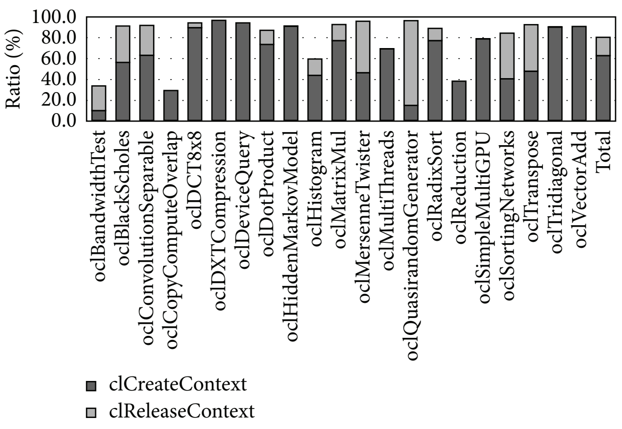


FIGURE 6: Execution time distribution of `clCreateContext` and `clReleaseContext`.

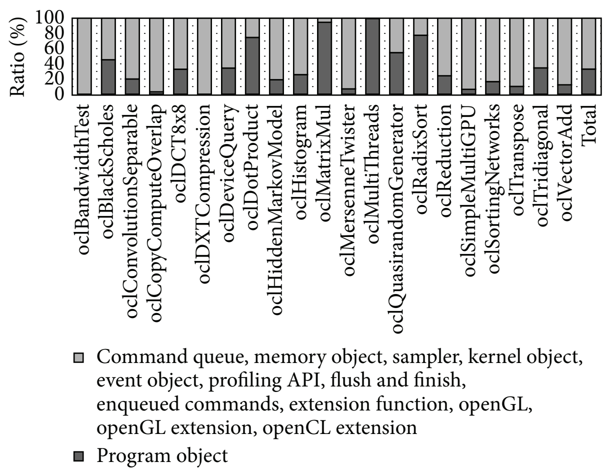


FIGURE 7: Execution time distribution of `Program` category API functions except `Platform`, `Device`, and `Context` categories.

workitem. Table 5 shows the number of workitems used in this evaluation.

From Figure 8, we find a performance gain for allocating more work per workitem on CPUs. A noticeable example is a case of `Vectoraddition`, where we add an array of numbers. If we create as many workitems as the size of arrays, we end up creating significant overhead on CPUs. When we reduce the number of workitems, we see a major performance improvement for CPUs. We could also find that the performance is saturated sometimes when the workload assigned per each workitem goes over a certain threshold. This shows that when each workitem has a sufficient workload, scheduling overhead is reduced.

Compared to CPUs with high overhead of handling many workitems, GPUs have low overhead for maintaining a large number of workitems, as our evaluation shows. Furthermore, reducing the number of workitems degraded performance on GPUs significantly. The large performance degradation on GPUs is because we could no longer take advantage of many processing units on GPUs.

One of the reasons for performance improvement by allocating more workload per workitem is the reduced number of instructions. Figure 9 shows the number of dynamic instructions of `Square` and `Vectoraddition` applications with a different amount of computation per workitem. The left figure of Figure 9 shows the dynamic instruction count including instructions from OpenCL APIs on top of instructions from the OpenCL kernel. And the right figure of Figure 9 represents the instructions only from the kernel.

For this evaluation, we implement a tool based on Pin [27] that counts the number of instructions. The tool also identifies the function to which the instruction belongs. From Figure 9, we can see that the number of instructions is reduced with more workload per workitem even though the amount of computation is the same regardless of the number of workitems. The number of instructions from OpenCL APIs as well as that from the kernels increases, so that the scheduling overhead exists on both OpenCL APIs and the JIT compiled OpenCL kernel binary. Figure 10 shows reduced overhead on OpenCL APIs with increased workload per workitem. The instructions from OpenCL APIs are for scheduling, not for computation intended by programmers represented as an OpenCL kernel. So a reduced number of instructions from OpenCL APIs means reduced overhead.

Figure 11 shows the performance of Parboil benchmarks with a similar experiment [25, 26]. The number of workitems coalesced is different depending on the benchmark since we could not increase the workload per workitem in the same manner for all kernels. We find a similar performance gain of allocating more work per workitem. Figure 12 represents the reduced number of dynamic instructions with increased workload per workitem.

3.3.2. Number of Workitems and Instruction-Level Parallelism (ILP). As we discussed in Section 2.2.2, the number of workitems, and therefore how to parallelize the computation, also affects the instruction-level parallelism (ILP) of the OpenCL kernel on CPUs. Coalescing multiple workitems can not only reduce the scheduling overhead but also improve the performance by utilizing ILP.

To evaluate the ILP effect on both the CPU and the GPU, we implemented a set of compute-intensive microbenchmarks that share common characteristics. Every benchmark has an identical number of dynamic instructions and memory accesses. Each benchmark also has the same instruction mixture, such as a ratio of the number of branch instructions over the total number of instructions. The only difference between each benchmark is ILP by varying the number of independent instructions. From the baseline implementation, we increase the number of operand variables, so that the number of independent instructions can increase. For example, in the case of `ILP 1`, the next instruction depends on the output of the previous instruction so that the number of independent instructions is one; but in the case of `ILP 2`, an independent instruction exists between two dependent instructions.

Figure 13 shows the performance with increasing ILP. We provide enough workitems to fully utilize TLP. The number of workitems remains the same for all microbenchmarks. The

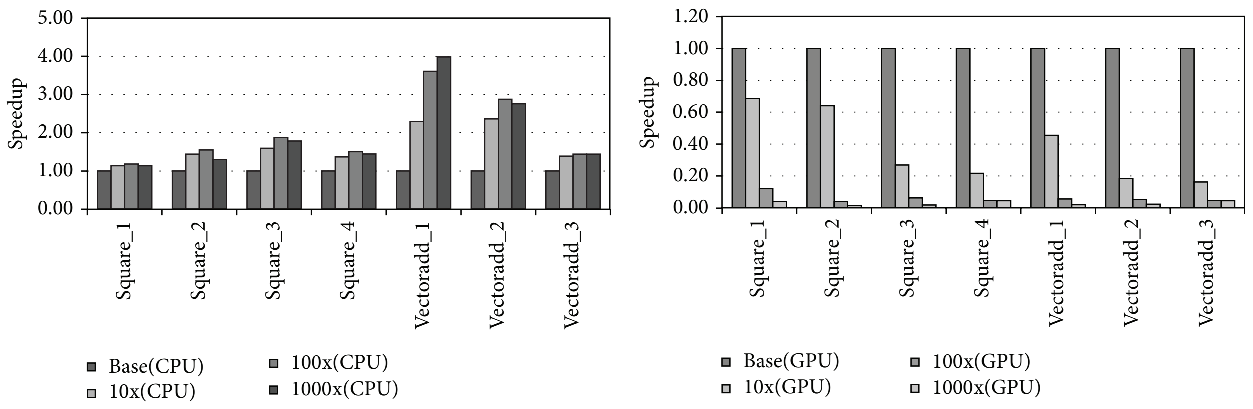


Figure 8: Performance of Square and Vectoraddition applications with different workload per workitem.

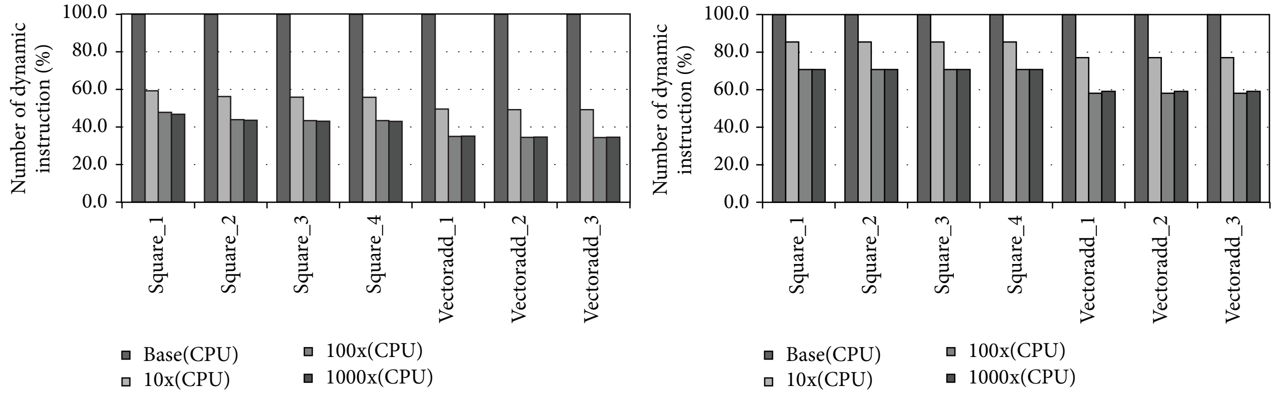


Figure 9: The number of dynamic instructions of Square and Vectoraddition applications with different workload per workitem including (L) instructions from OpenCL APIs and (R) kernel only.

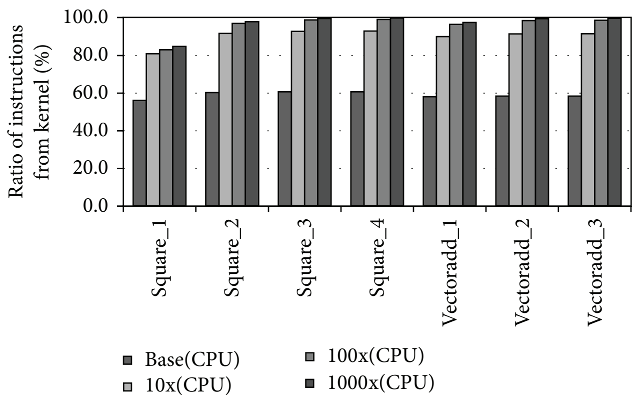


Figure 10: The ratio of instructions from kernel over the instructions around clEnqueueNDRangeKernel for Square and Vectoraddition applications with different workload per workitem.

left y-axis represents the throughput of the CPUs, and the right one represents the throughput of the GPUs. From the figure, we find that performance improves depending on the ILP value of the OpenCL kernel on CPUs. On the contrary, there is no performance variation on GPUs with different degrees of instruction-level parallelism.

3.3.3. Workgroup Size. Associated with the discussion in Section 2.2.3, the number of workitems in workgroups can affect the performance of the OpenCL application. We evaluate the effect of workgroup size, both on CPUs and GPUs. We vary the number of workitems in a workgroup by passing a different argument for workgroup size (`local_work_size`) on kernel invocation. We maintain the total number of workitems of the kernel as the same. Table 6 shows the different workgroup size for each benchmark, and Figures 14, 16, and 18 show the performance of applications with different workgroup sizes. When the `NULL` argument is passed on kernel invocation, the workgroup size is implicitly defined by the OpenCL implementation.

The benchmarks can be categorized into three categories, depending on the behavior. The first group consists of `Square`, `Vectoraddition`, and naive implementation of `Matrixmul`; `Matrixmul` belongs to the second group; and `Blackscholes` belongs to the last.

`Square`, `Vectoraddition`, and naive implementation of `Matrixmul` show a performance increase with increased workgroup sizes on the CPU, as can be seen in Figure 14. On the `Square` and `Vectoraddition` applications, performance achieved with the `NULL` workgroup size is less than the peak performance we achieve. This implies that

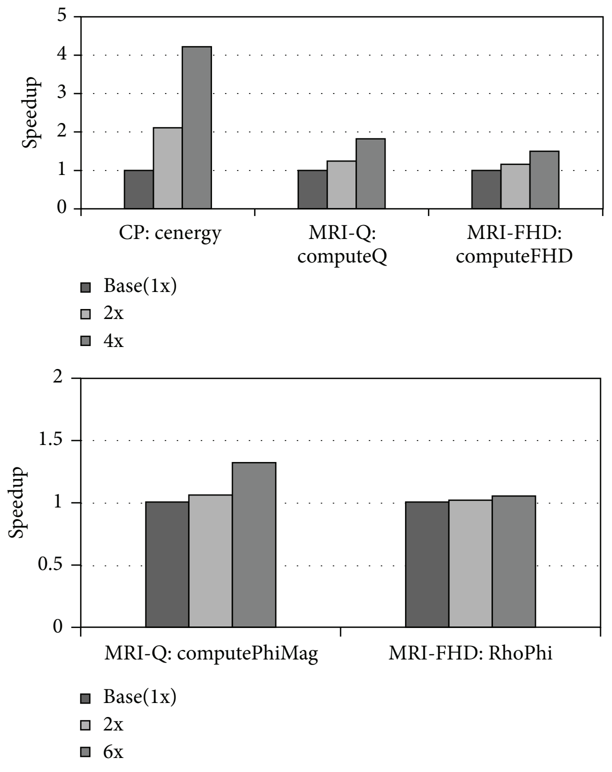


Figure 11: Performance of Parboil benchmarks with different workloads per workitem.

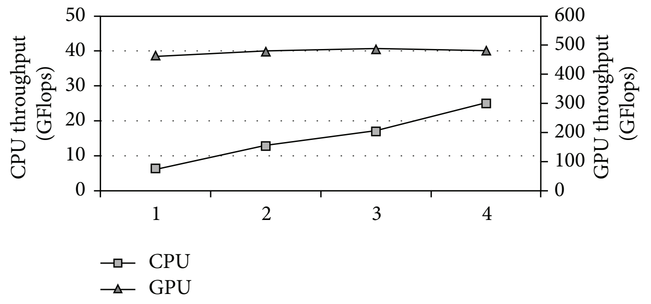


Figure 13: Performance of ILP microbenchmark on the CPU and the GPU.

Table 5: Number of workitems for each application.

Benchmark	Base	10x	100x	1000x
Square_1	10000	1000	100	10
Square_2	100000	10000	1000	100
Square_3	1000000	100000	10000	1000
Square_4	10000000	1000000	100000	10000
Vectoradd_1	110000	11000	1100	110
Vectoradd_2	1100000	110000	11000	1100
Vectoradd_3	5500000	550000	55000	5500

Table 6: Workgroup size for each application.

Benchmark	Base	Case_1	Case_2	Case_3	Case_4
Square	NULL	1	10	100	1000
Vectoraddition	NULL	1	10	100	1000
Matrixmul	16×16	1×1	2×2	4×4	8×8
Blackscholes	16×16	1×1	1×2	2×2	2×4
Matrixmul(naive)	16×16	1×1	2×2	4×4	8×8

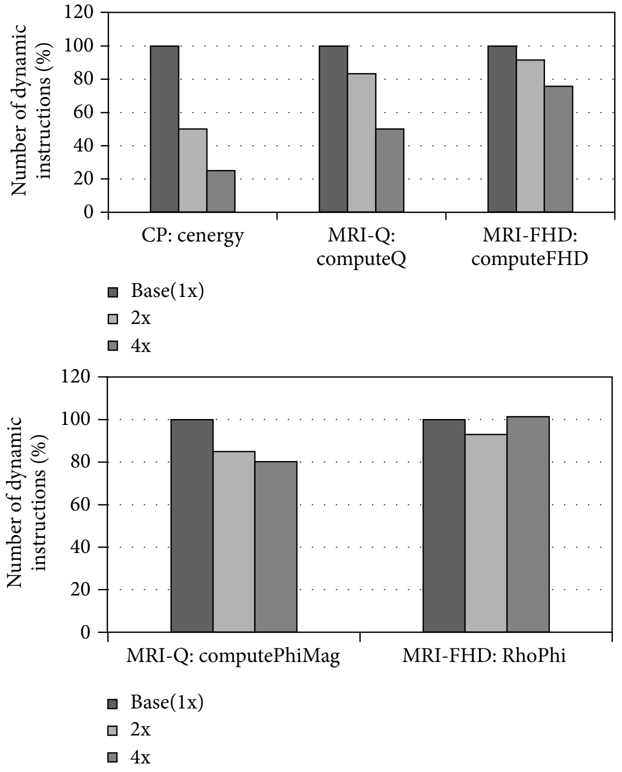


Figure 12: The number of dynamic instructions of Parboil benchmarks with different workload per workitem.

programmers should explicitly set the workgroup size for the maximum performance. The performance with a small workgroup size is also bad on GPUs since the workgroup is allocated per SM, so that the small workgroup size makes GPUs unable to utilize many warps in an SM. Even though no hardware TLP is available inside a logical core on CPUs (the evaluated CPU is an SMT processor, so multiple logical cores share one physical core), performance increases with a large workgroup size. This is because the overhead of managing a large number of workgroups, many threads in many implementations, is reduced. We also find that performance is saturated at a certain workgroup size.

The left figure of Figure 15 shows the number of dynamic instructions of `Square`, `Vectoraddition`, and naive implementation of `Matrixmul` with different workgroup size on CPUs. The right figure of Figure 15 shows the ratio of instructions from kernel over the instructions around `clEnqueueNDRangeKernel` for those applications with a different workgroup size. From the left figure of Figure 15, we can see that the number of instructions is reduced with larger workgroup size. This is because the number of instructions from OpenCL APIs is reduced, as can be seen from the right figure of Figure 15. The number

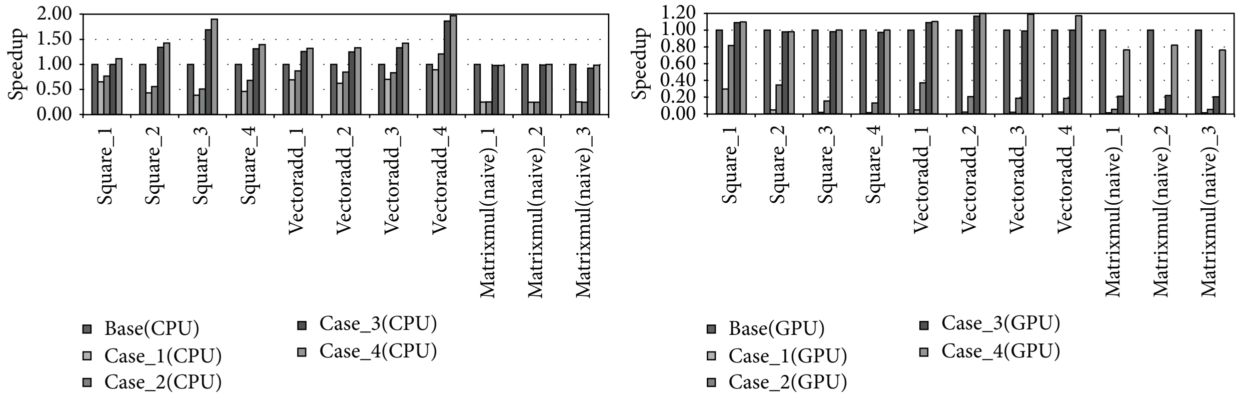


Figure 14: Performance of applications with different workgroup size on CPUs and GPUs.

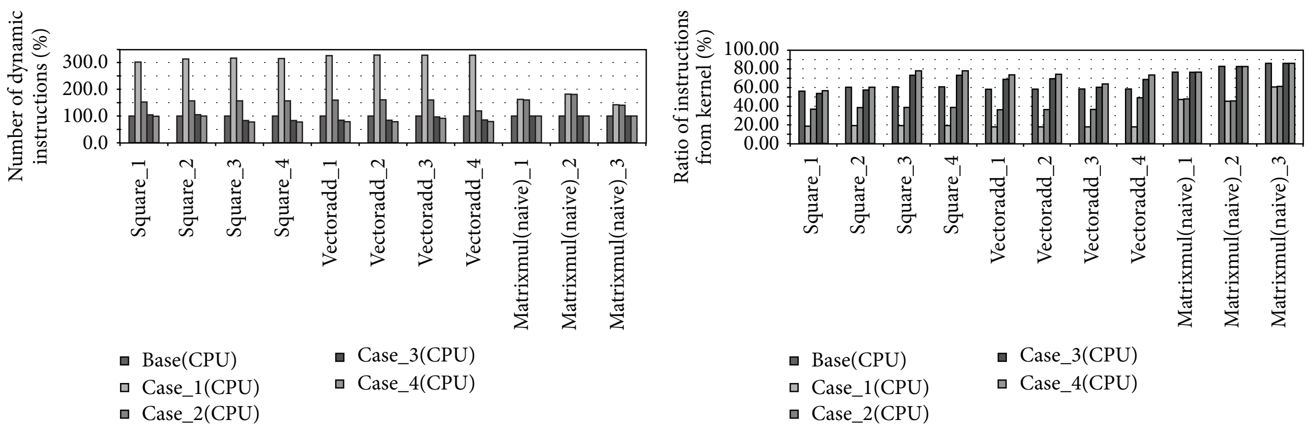


Figure 15: (U) The number of dynamic instructions of Square, Vectoraddition, and naive implementation of Matrixmul with different workgroup size on CPUs. (L) The ratio of instructions from kernel over the instructions around clEnqueuNDRangeKernel for Square, Vectoraddition, and naive implementation of Matrixmul with different workgroup size.

of instructions from the OpenCL kernel remains the same regardless of workgroup size.

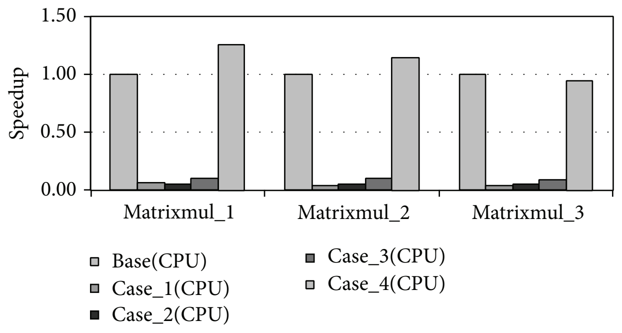
As we can see from Figure 16, we also see a significant performance increase on the `Matrixmul` application with an increased workgroup size. The optimal workgroup size of this application is different, depending on platforms. For inputs 1 and 2, the optimal workgroup size on CPUs is 8×8, but the optimal size on GPUs is 16×16. Here, the performance depends not only on the scheduling overhead, but also on the cache usage. `Matrixmul` utilizes the local memory in OpenCL by blocking. Workgroup size can change the local memory usage of the kernel. Since the size of the cache in CPUs and the scratchpad memory in GPUs are different, the optimal workgroup size can be different. Figure 17 shows the reduced number of dynamic instructions of Matrixmul with increasing workgroup size.

Unlike other applications, `Blackscholes` shows different performance behavior on CPUs and on GPUs. As we can see in Figure 18, the workgroup size does not change the performance on CPUs, but it affects the performance significantly on GPUs. Since the workload allocated on a single workitem is relatively long compared to other applications, the overhead of managing a large number of workgroups becomes negligible. On the contrary, the number of warps in the SM is defined by the workgroup size on GPUs, which makes the performance on GPUs low on small workgroup sizes. Figure 19 shows that the number of instructions does not change much for Blackscholes, regardless of workgroup size.

Figure 20 shows the performance of Parboil benchmarks with different workgroup sizes. We increase the workgroup size from one to 16 times by multiplying by 2 for each step. Since the workgroup size for `CP:cenergy` kernel is two-dimensional, we increase the workgroup size of the kernel in two directions. `CP:cenergy(x)` represents the performance with workgroup sizes 1×8, 2×8, 4×8, 8×8, and 16×8. `CP:cenergy(y)` represents the performance with workgroup sizes 16×1, 16×2, 16×4, 16×8, and 16×16. In general, we find the performance gain with a large workgroup size. The performance saturates when there is enough computation inside the workgroup. Figure 21 shows that the performance gain is due to reduced scheduling overhead, which is represented by a reduced number of dynamic instructions.

3.3.4. *Summary.*

Here, we summarize the findings on thread scheduling for OpenCL applications.



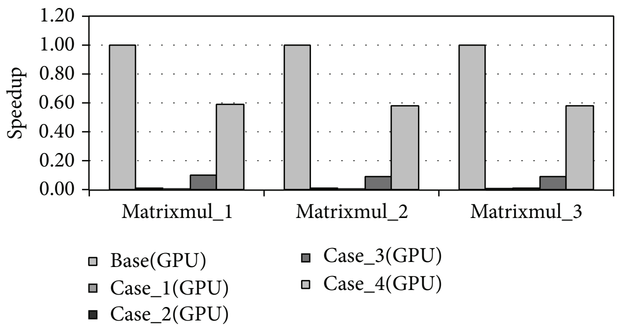


Figure 16: Performance of Matrixmul with different workgroup size on CPUs and GPUs.

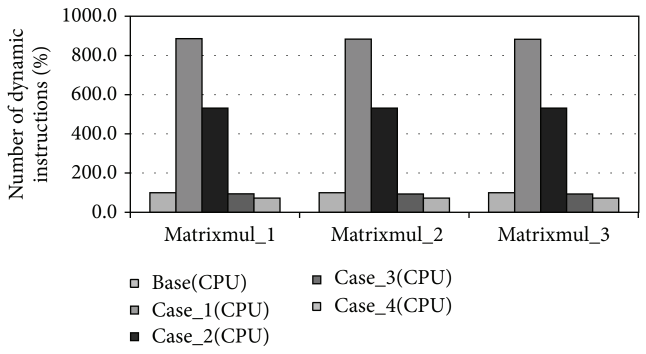


Figure 17: The number of dynamic instructions of Matrixmul with different workgroup size on CPUs.

(1) Allocating more work per workitem by manually coalescing multiple workitems reduces scheduling overhead on CPUs (Section 3.3.1).

(2) High ILP increase performance on CPUs but not on GPUs (Section 3.3.2).

(3) Workgroup size affect performance both on CPUs and GPUs. In general, large workgroup size increases performance by reducing scheduling overhead on CPUs and enables utilizing high TLP on GPUs. Workgroup size can also affect the cache usage (Section 3.3.3).

3.4. Memory Allocation and Data Transfer. Associated with the discussion in Section 2.3, to evaluate the performance effect of different memory object allocation flags and different APIs for data transfer, we perform an experiment on OpenCL applications with different combinations of the following options. To measure exact execution performance, we use a blocking call for all kernel execution commands and memory object commands so that no command overlaps with other commands. The combination we use is three-dimensional as follows.

3.4.1. Evaluated Options for Memory Allocation and Data Transfer

(1) Different APIs for data transfer:

   (i) explicit transfer: `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` for explicit read and write;

   (ii) mapping: `clEnqueueMapBuffer` with `CL_MAP_READ`, `CL_MAP_WRITE` for implicit read and write.

(2) Kernel access type when referenced inside a kernel:

   (i) the kernel accesses the memory object as read-only/write-only:

      (a) `CL_MEM_READ_ONLY` for the input to the kernel;

      (b) `CL_MEM_WRITE_ONLY` for computation results from the kernel;

   (ii) the kernel accesses the memory object as read/write: `CL_MEM_READ_WRITE` for all memory objects.

(3) Where to allocate a memory object:

   (i) allocation on the device memory;

   (ii) allocation on the host-accessible memory on the host (pinned memory).

3.4.2. Metric: Application Throughput. The throughput we present here is the performance, including data transfer time, between the host and compute devices, not just the kernel execution throughput on the compute device. For example, the throughput of an application becomes half of the throughput when we consider only the kernel execution time if the data transfer time between the host and the compute device equals the kernel execution time. The way we calculate the throughput of an application is illustrated in

$$\text{Throughput\_app} = \frac{\text{Throughput\_kernel}}{\text{kernel\_time} + \text{transfer\_time}}. \quad (1)$$

3.4.3. Different Data Transfer APIs. We compare the performance of different data-transfer APIs on all possible allocation flags. (The combinations are as follows: (1) read-only/write-only memory object + allocation on the device; (2) read-only/write-only memory object + allocation on the

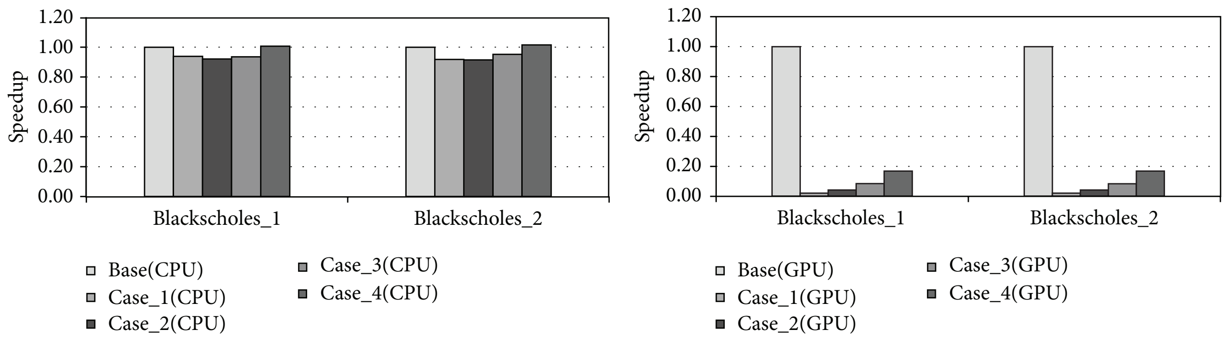


Figure 18: Performance of Blackscholes with different workgroup size on CPUs and GPUs.

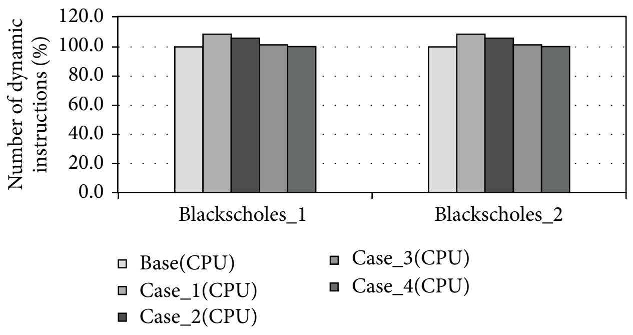


Figure 19: The number of dynamic instructions of Blackscholes with different workgroup size on CPUs.

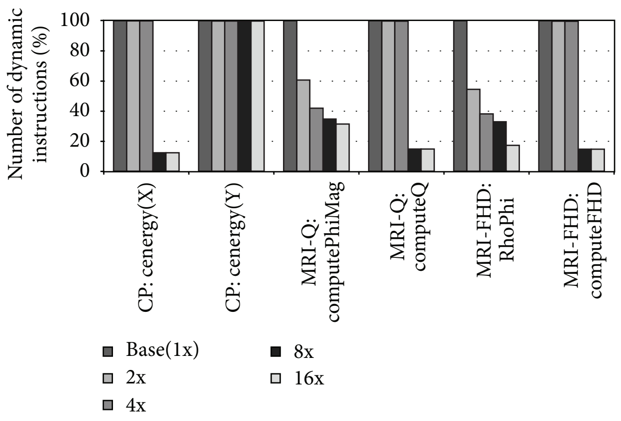


Figure 21: The number of dynamic instructions of Parboil benchmarks with different workgroup size on CPUs.

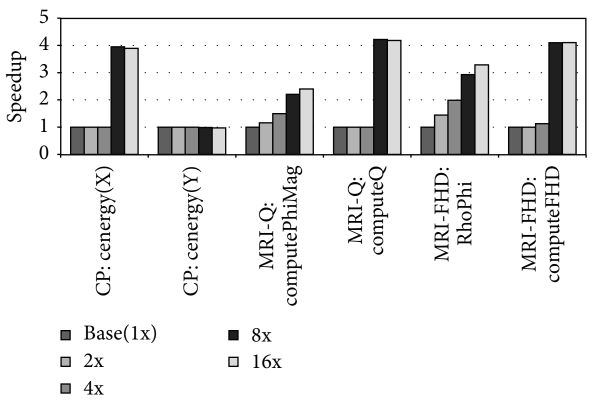


Figure 20: Performance of Parboil benchmarks with different workgroup size on CPUs.

host; (3) read-write memory object + allocation on the device; (4) read-write memory object + allocation on the host.) Figure 22 shows the performance of the benchmarks with different APIs for data transfer. The y-axis represents the normalized application throughput (`Throughput_app`) when we use mapping for data transfer over the baseline when we use explicit data transfer APIs. We find that mapping APIs have superior performance compared to explicit data transfer APIs, regardless of the decision on other dimensions. First, the performance of mapping APIs is superior wherever the memory object is allocated: on device memory or on pinned memory on host. Second, mapping APIs also perform better regardless of the decision for allocating the memory object as read-only/write-only or as read/write object.

Different APIs change data transfer time. Figure 23 shows the normalized data transfer throughput from the host to a compute device between different data transfer APIs. Figure 24 shows the one from compute device to host. The data transfer time is shorter with mapping APIs. The difference of data transfer throughput increases with increases in workload sizes and therefore increases in data transfer sizes.

We also report the performance of Parboil benchmarks with different APIs for data transfer [25, 26]. Since the data transfer time is much shorter than the kernel execution time on Parboil benchmarks, instead of using application throughput as shown in (1), we report the data transfer time from the host to device, and data transfer time from the device to host with different APIs. Figure 25 shows the different data transfer time of the Parboil benchmarks with different APIs for data transfer. The y-axis represents the data transfer time in milliseconds. The left figure in Figure 25 shows the data transfer time from the host to the compute device with different data transfer APIs. The right figure shows the one from the compute device to the host. As with simple applications, we find that the data transfer time is shorter with mapping APIs on these benchmarks.

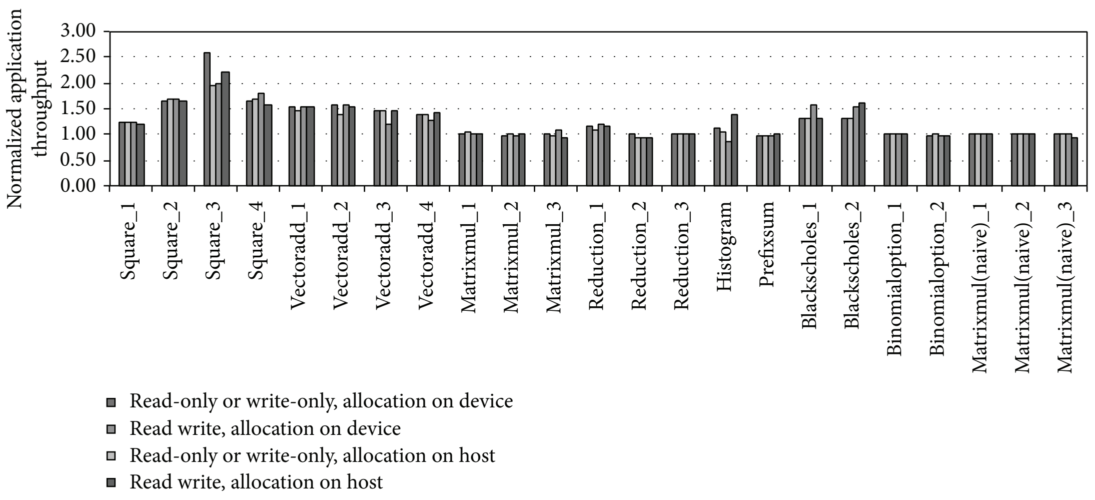


Figure 22: Normalized application throughput of mapping over explicit data transfer for all combinations on other dimensions. The performance of mapping APIs is superior to explicit data transfer on all possible combinations.

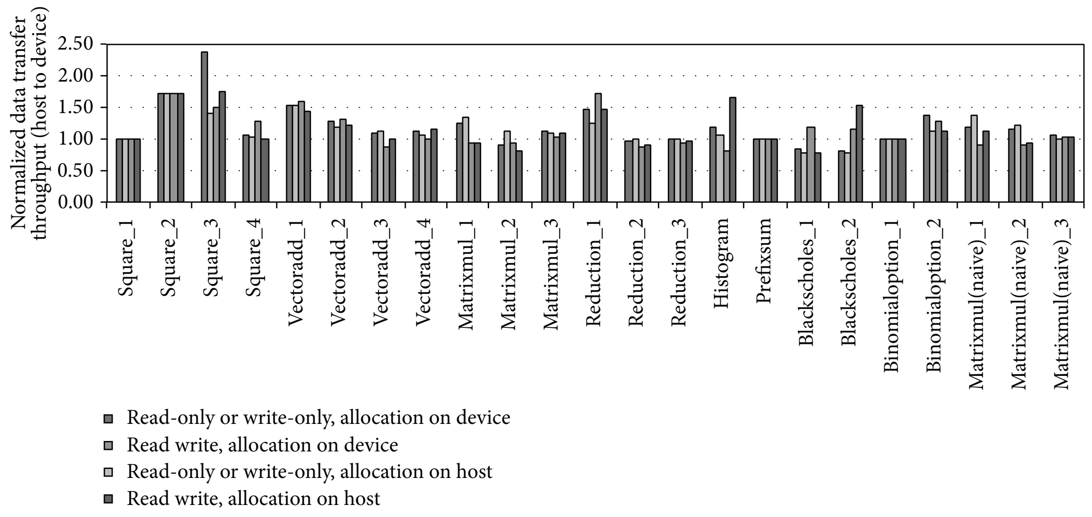


Figure 23: Normalized data transfer (host to device) throughput of mapping over explicit data transfer for all combinations on other dimensions.

The difference of data transfer time is due to the different behaviors of different APIs. When the host code explicitly transfers data between the host and the compute device, the OpenCL run-time library should allocate a separate memory object for the device and copy the data between the memory object allocated by the `malloc` call and the memory object allocated for the device that is allocated by the `clEnqueueReadBuffer` call. However, a separate memory object is not needed when the host code uses mapping; only returning a pointer of the memory object is needed. So, copying between memory objects becomes unnecessary.

3.4.4. Kernel Access Type When Referenced inside a Kernel. We also verify the performance effect of specifying a memory object as read-only/write-only or as read/write. Figure 26 shows the performance implication of this flag. The y-axis represents the normalized throughput when we allocate the memory object as read-only/write-only from the baseline when we allocate the object as read/write. OpenCL implementations can utilize the detailed information of how the memory object is accessed in the OpenCL kernel for optimization instead of naively assuming all objects are read and modified in the OpenCL kernel. However, we do not see a noticeable performance difference with our evaluated workloads. Kernel execution time and data transfer time between the host and compute device do not differ regardless of this memory allocation flag.

3.4.5. Where to Allocate a Memory Object. Finally, we also verify the performance effect of the allocation location of memory objects. Programmers can allocate the memory object on the host memory or the device memory. Figure 27 shows the performance of benchmarks with

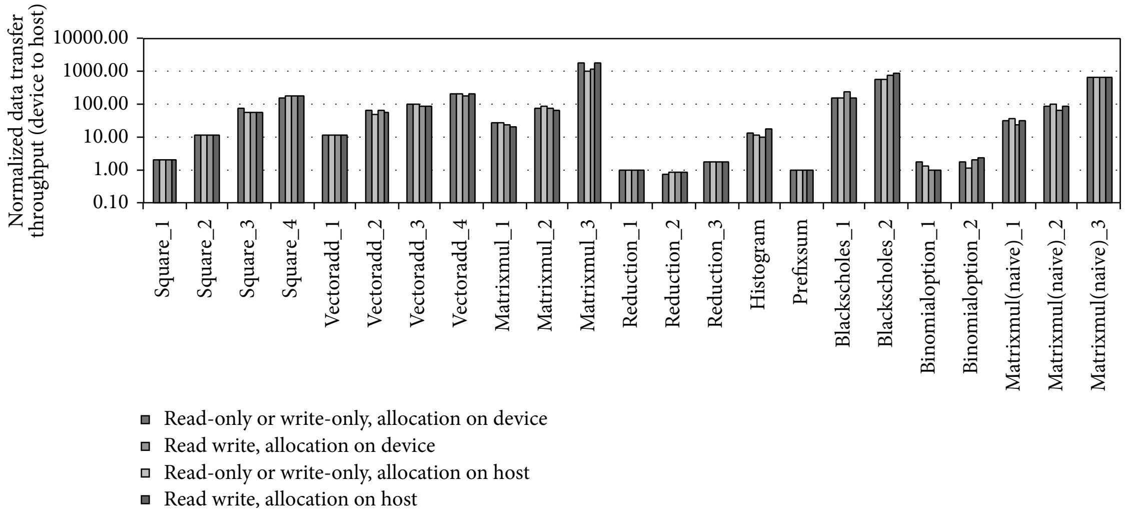


Figure 24: Normalized data transfer (device to host) throughput of mapping over explicit data transfer for all combinations on other dimensions.

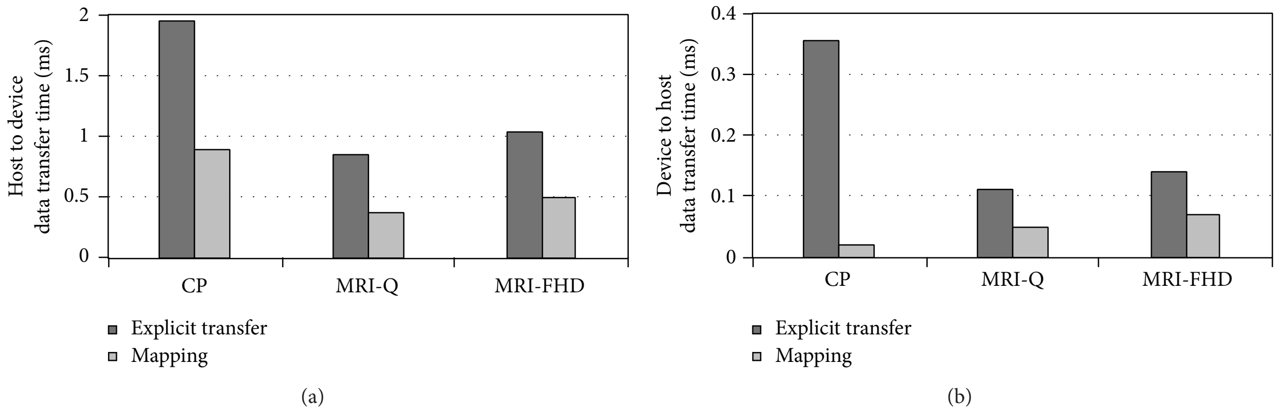


Figure 25: Data transfer time with different APIs for data transfer. (Left) host to device and (Right) device to host.

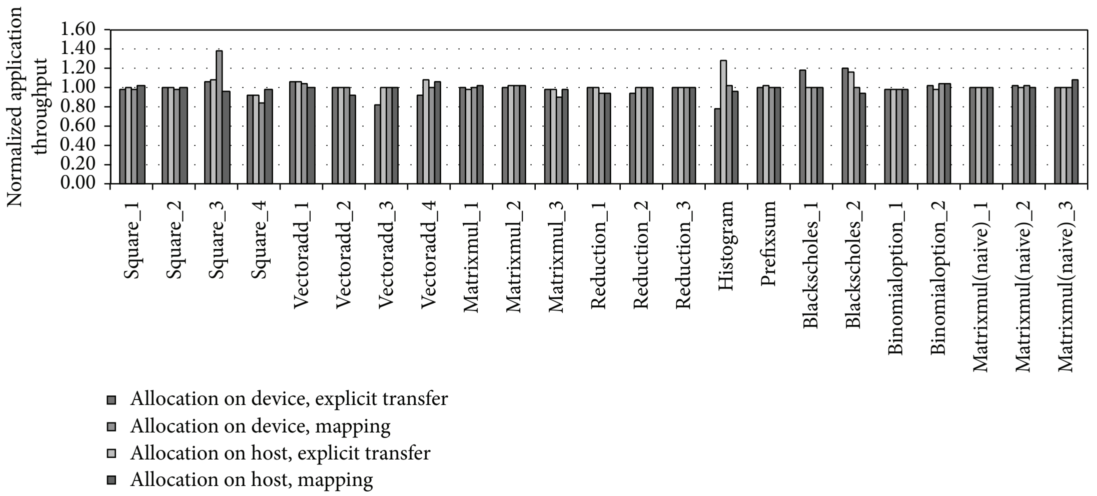


Figure 26: Normalized application throughput of read-only/write-only memory objects over read/write memory objects for all combinations on other dimensions. There is no noticeable performance difference.

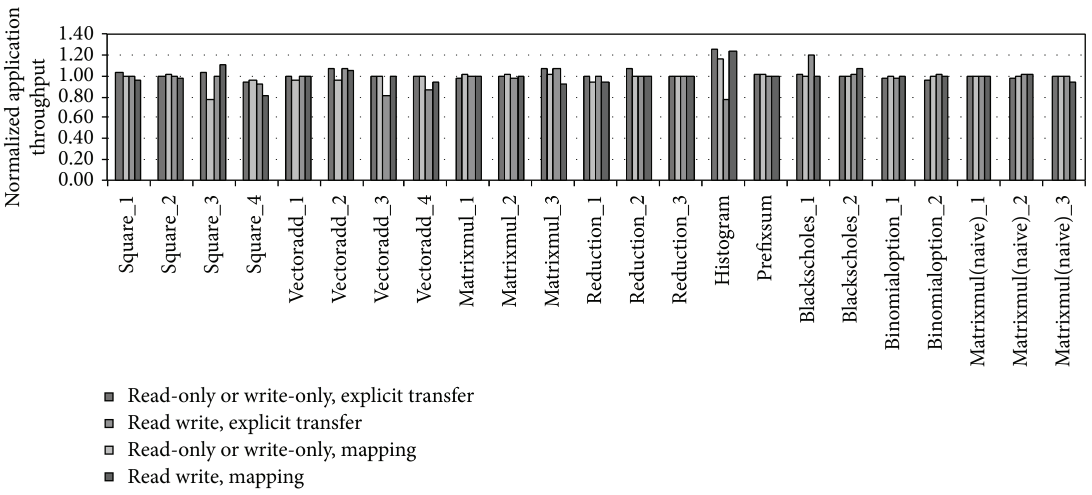


FIGURE 27: Normalized application throughput of the pinned memory over the device memory for all combinations on other dimensions. Where to allocate a memory object does not change the performance much on CPUs.

different allocation locations. The y-axis represents the normalized throughput when we allocate the memory object on the host memory from the baseline when we use the device memory. We find that an allocation location does not have a huge impact on performance both for kernel execution time and data transfer time. This is because the device memory and the host memory reference the same memory, the main memory of the system when the compute device is the CPU. Therefore, a different memory allocation location does not imply performance differences. On the contrary, when the compute device is not the CPU, memory allocation location can affect the performance significantly.

3.4.6. Summary. In this section we find that mapping APIs perform superior compared to explicit data transfer APIs with reduced data transfer time by eliminating the copying overhead on CPUs. Allocated location and kernel access type do not affect the performance on CPUs.

3.5. Vectorization and Thread Affinity

3.5.1. Vectorization. We evaluate the possible effect of programming models on vectorization, even though vectorization is more about compiler implementation. For evaluation, we port the OpenCL kernels to identical computations being performed by their OpenMP counterparts. We map multiple workitems on OpenCL to a loop to port OpenCL kernels to their OpenMP counterparts. We utilize the Intel C/C++ 12.1.3 compiler and the Intel OpenCL platform 1.5 for our evaluation. The programmer's expectation is that when we run the same computation in the OpenCL and OpenMP applications, both runs should give comparable performance numbers. However, the results show that this assumption does not hold. For the evaluated benchmarks, the OpenCL kernels outperform their OpenMP counterparts. Figure 28 shows the different performance of OpenMP and OpenCL implementations. The reason for this mismatch is the different way OpenMP and OpenCL compilers vectorize code.

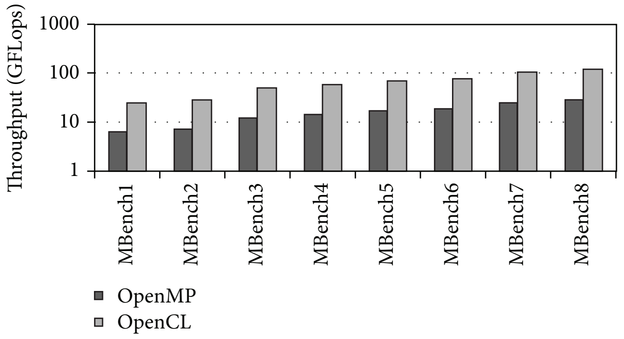


FIGURE 28: Performance impact of vectorization.

OpenCL Vectorization. The vectorization by the OpenCL kernel compiler is coalescing workitems. OpenCL vectorization enables the execution of several workitems together by a single vector instruction. Vectorization enables multiple work items to be processed concurrently on a single thread. For example, if the target instruction set is SSE 4.2, and the computation is based on a single precision floating point, then four workitems could make progress concurrently, so they are coalesced into a single workitem. By doing this, vectorized OpenCL code would have fewer dynamic instruction counts compared to nonvectorized code.

OpenMP Vectorization. On the other hand, the OpenMP compiler vectorizes loops by unrolling a loop combined with the generation of packed SIMD instructions. To be vectorized, a loop should be countable, have single entry and single exit, and have a straight control flow graph inside the loop [28]. Many factors could prevent the vectorization of a loop in OpenMP. Two key factors are (1) `non-contiguous memory access` and (2) `data dependence`.

(1) Noncontiguous memory access:

  (i) four consecutive floats may be loaded directly from the memory in a single SSE instruction;

```
/*OpenMP computation that doesn't vectorize due to dependencies.*/
int main(){
   ...
   for (int j = 0; j < 4; j++){
      FMUL(_a[j], _b[j])
      FMUL(_a[j], _b[j])
      FMUL(_a[j], _b[j])
      FMUL(_a[j], _b[j])
      FMUL(_a[j], _b[j])
      FMUL(_a[j], _b[j])
   }
   ...
}
/*Similar OpenCL kernel computation which vectorizes.*/
void VectorAdd (..., __global float *dm_src, __global float *dm_dst){
   ...
   for (int j = 0; j < 4; j++){
      FMUL(_a[j], _b[j])
      FMUL(_a[j], _b[j])
      FMUL(_a[j], _b[j])
      FMUL(_a[j], _b[j])
      FMUL(_a[j], _b[j])
      FMUL(_a[j], _b[j])
   }
   ...
}
```

Algorithm 1: Vectorization on OpenCL versus OpenMP. The equivalent code in OpenCL is vectorizable while OpenMP code is not vectorizable.

but if the four floats to be loaded are not consecutive, we will have a load using multiple instructions; loops with a nonunit stride are an example of the above scenario.

(2) Data dependence:

(i) vectorization requires changes in the order of operations within a loop since each SIMD instruction operates on several data elements at once; but such a change of order might not be possible due to data dependencies.

Example. Algorithm 1 shows an example of how different vectorization mechanisms from OpenMP and OpenCL compilers affect whether identical codes are to be vectorized or not. When there is a true data dependence inside an OpenCL kernel or inside a loop iteration in OpenMP `parallel for` section, the OpenCL kernel is vectorized, while the OpenMP code is not. Therefore, they show different performance even when vectorization of OpenMP loops seems possible. The vectorization of an OpenCL kernel is relatively straightforward because no dependency checks are required as in the case of traditional compilers. Even though we only show the example of when the OpenCL compiler shows the benefit, the opposite case is also possible: when the OpenMP compiler vectorizes code, but the OpenCL compiler fails.

New OpenMP Compiler. We have also evaluated OpenMP vectorization with OpenMP 4.0 SIMD extension and the newer compiler (Intel C/C++ compiler 15.0.1). The evaluation revealed comparable performance of OpenMP and OpenCL implementations. Compiler vectorization is dependent on the compiler implementation.

3.5.2. Thread Affinity. We evaluate the performance benefit using the CPU affinity in OpenMP. We use `OMP_PROC_BIND` and `GOMP_CPU_AFFINITY` to control the scheduling of threads on the processors [8]. When the `OMP_PROC_BIND` is set to be true, the threads will not be moved between processors. `GOMP_CPU_AFFINITY` enables us to control the allocation of a thread on a particular core.

We use a simple application for evaluation. The aim of the application is to verify the effects of binding threads to cores in terms of cache utilization. Performance can improve when the OpenCL run-time library maps logical threads of a kernel on physical cores so that it can utilize the cached data of the previous kernel execution. The application we use consists of two kernels: `Vector Addition` and `Vector Multiplication`. Computation of each kernel is distributed among eight cores: and the computation of second kernel is dependent on the first one, using the data produced by the first one.

Table 7 shows the method we use. The upper table in Table 7 represents the (a) `Aligned` case, and the lower table represents the (b) `Misaligned` case. The numbers in the table represent the logical thread IDs. Threads with identical IDs of both the kernels access the same data. On the (a) `Aligned` case, we bind threads of the second kernel to the cores on which the threads of the first kernel

Table 7: Performance impact of CPU affinity.

(a) Aligned

	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
Computation 1	0	1	2	3	4	5	6	7
Computation 2	0	1	2	3	4	5	6	7

(b) Misaligned

	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
Computation 1	0	1	2	3	4	5	6	7
Computation 2	6	3	4	0	2	1	7	5

```
/∗First Kernel: Vector Addition.∗/
#pragma omp parallel for shared(a, b, c) private (i)
for (int i = 0; i < MAX_INDEX; i++){
   c[i] = a[i] + b[i];
}
/∗(a) Aligned Second Kernel: Vector Multiplication.∗/
#pragma omp parallel for shared(b, c, d) private (i)
for (int i = 0; i < MAX_INDEX; i++){
   d[i] = b[i] + c[i];
}
/∗(b) Misaligned Second Kernel: Vector Multiplication.∗/
#pragma omp parallel for shared(b, c, d) private (i)
for (int i = 0; i < MAX_INDEX; i++){
      int j = MAX_INDEX − 1 − i;
   d[j] = b[j] + c[j];
}
```

Algorithm 2: Code snippet of simple application.

are bound. On the (b) `Misaligned` case, we shuffle this binding. Algorithm 2 shows the code snippet of this simple application.

As we expect, the (a) `Aligned` case shows higher performance than does the (b) Misaligned case. The (b) `Misaligned` one runs longer by 15%. This is because during the execution of the second kernel, the cores on the CPU encounter cache misses on their private caches. On the contrary, the (a) `Aligned` case would have more cache hits than the (b) `Misaligned` case because the data accessed by the second kernel would already be on the cache after the execution of the first kernel on the (a) `Aligned` case.

As the results show, even though OpenCL emphasizes portability, adding the affinity support to OpenCL may provide a significant performance improvement in some cases. Hence, we argue that coupling logical threads with physical threads (cores on the CPU) is needed on OpenCL, especially for CPUs. The granularity for the assignment could be a workgroup; in other words, the programmer can specify the core where a specific workgroup would be executed. This functionality would help to improve the performance of OpenCL applications. For example, data from different kernels can be shared without a memory request if the programmer allocates cores for specific workgroups in consideration of the data sharing of different kernels. The data can be shared through the private caches of cores.

4. Related Work

Multiple research studies have been done on how to optimize OpenCL performance on GPUs. The GPGPU community provides TLP [29] as a general guideline for optimizing GPGPU applications since GPGPUs are usually equipped with a massive number of processing elements. Since OpenCL has the same background as CUDA [9], most OpenCL applications are written to better utilize TLP. The widely used occupancy metric indicates the degree of TLP. However, this scheme cannot be applied on CPUs since even when the TLP of the application is large, the physical TLP available on CPUs is limited by the number of CPU cores, so that the context switching overhead is much higher on CPUs than on GPUs for which this overhead is negligible.

Several publications refer to the performance of OpenCL kernels on CPUs. Some focus on algorithms and some refer to the performance difference by comparing it with GPU implementation and OpenMP implementation on CPUs [16, 30, 31]. However, to the best of our knowledge, our work is the first to provide a broad summary, combining application with the architecture knowledge to provide a general guideline to understand OpenCL performance on multicore CPUs.

Ali et al. compare OpenCL with OpenMP and Intel's TBB on different platforms [30]. They mostly discuss the scaling effects and compiler optimizations. But it misses out on why the optimizations listed in the paper give the performance benefit mentioned and lacks quantitative evaluation. We, too, evaluate the performance of OpenCL and OpenMP for a given application. However, our work considers various aspects that can change application performance and provide quantitative evaluations to help programmers estimate the performance impact of each aspect.

Seo et al. discuss OpenCL performance implications for the NAS parallel benchmarks and give a nice overview of how they optimize the benchmarks by first getting an idea of the data transfer and scheduling overhead and then coming up with ways to avoid them [31]. They also show how to rewrite a good OpenCL code, given an OpenMP code. Stratton et al. describe a way to implement a compiler for fine-grained SPMD-thread programs on multicore execution platforms [16]. For the fine-grained programming model, they start with CUDA, saying that it will apply to OpenCL as well. They focus on the performance improvement over the baseline. Our work is more generalized and broad compared to these previous studies and also includes some of the important points that are not addressed in these papers.

One of the references that is very helpful to understand the performance behavior of OpenCL is a document from Intel [32]. It broadly lays out some general guidelines to follow to get better performance out of OpenCL applications on Intel processors. However, it does not discuss the performance improvement and also does not state how much benefit can be achieved.

5. Conclusion

We evaluate the performance of OpenCL applications on modern multicore CPU architectures. Understanding the performance in terms of architectural resource utilization is helpful for programmers. In this paper, we evaluate various aspects, including API overhead, thread scheduling, ILP, data transfer, data locality, and compiler-supported vectorization. We verify the unique characteristics of OpenCL applications by comparing them with conventional parallel programming models such as OpenMP. Key findings of our evaluation are as follows.

(1) OpenCL API overhead is not negligible on CPUs (Section 3.2).

(2) Allocating more work per workitem therefore reducing the number of workitems helps performance on CPUs (Section 3.3.1).

(3) Large ILP helps performance on CPUs (Section 3.3.2).

(4) Large workgroup size is helpful for better performance on CPUs (Section 3.3.3).

(5) On CPUs, Mapping APIs perform superior compared to explicit data transfer APIs. Memory allocation flags do not change performance (Section 3.4).

(6) Programming model can have possible effect on compiler-supported vectorization. Conditions for the code to be vectorized can be complex (Section 3.5.1).

(7) Adding affinity support to OpenCL may help performance in some cases (Section 3.5.2).

Our evaluation shows that considering the characteristics of CPU architectures, the OpenCL application can be optimized further for CPUs, and the programmer needs to consider these insights for portable performance.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors would like to thank Jin Wang and Sudhakar Yalamanchili, Inchoon Yeo, the Georgia Tech HPArch members, and the anonymous reviewers for their suggestions and feedback. We gratefully acknowledge the support of the NSF CAREER award 1139083 and Samsung.

References

[1] AMD, AMD Accelerated Processing Units (APUs), http://www.amd.com/en-us/innovations/software-technologies/apu.

[2] Intel, "Products (Formerly Sandy Bridge)," http://ark.intel.com/products/codename/29900/Sandy-Bridge.

[3] Khronos Group, "OpenCL: the open standard for parallel programming of heterogeneous systems," http://www.khronos.org/opencl.

[4] Intel, "Intel OpenCL SDK," http://software.intel.com/en-us/articles/intel-opencl-sdk/.

[5] NVIDIA, "NVIDIA OpenCL SDK," http://developer.nvidia.com/cuda/opencl/.

[6] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys*, vol. 35, no. 2, pp. 97–113, 2003.

[7] Intel, Intel Threading Building Blocks, http://threadingbuildingblocks.org/.

[8] The OpenMP Architecture Review Board, OpenMP, http://openmp.org/wp/.

[9] NVIDIA, CUDA Programming Guide, V4.0, 2011.

[10] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-M. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pp. 73–82, February 2008.

[11] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, and W.-M. W. Hwu, "Program optimization study on a 128-core GPU," in *Proceedings of the 1st Workshop on General Purpose Processing on Graphics Processing Units (GPGPU '07)*, October 2007.

[12] S. Ryoo, C. I. Rodrigues, S. S. Stone et al., "Program optimization space pruning for a multithreaded GPU," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08)*, pp. 195–204, 2008.

[13] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '08)*, pp. 31:1–31:11, November 2008.

[14] R. Balasubraamonian, S. Dwarkadas, and D. H. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pp. 237–248, December 2001.

[15] J. Kim, S. Seo, J. Lee, J. Nah, and G. Jo, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*, pp. 341–351, June 2012.

[16] J. A. Stratton, V. Grover, J. Marathe et al., "Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs," in *Proceedings of the 8th International Symposium on Code Generation and Optimization (CGO '10)*, pp. 111–119, ACM, April 2010.

[17] G. Diamos, "The design and implementation Ocelot's dynamic binary translator from PTX to Multi-Core x86," Tech. Rep. GIT-CERCS-09-18, Georgia Institute of Technology, 2009.

[18] B. Saha, X. Zhou, H. Chen et al., "Programming model for a heterogeneous x86 platform," in *Proceedings of the ACM*

SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09), pp. 431–440, June 2009.

[19] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*, pp. 372–382, Galveston, Tex, USA, October 2011.

[20] L. Gwennap, "Intel's MMX speeds multimedia," Microprocessor Report, 1996.

[21] Intel, "Intel Integrated Performance Primitives," https://software.intel.com/en-us/intel-ipp.

[22] Intel, Intel Math Kernel Library, http://software.intel.com/en-us/intel-mkl.

[23] Intel, Intel C and C++ Compilers, https://software.intel.com/en-us/c-compilers.

[24] M. Pharr and W. R. Mark, "ispc: a SPMD compiler for high-performance CPU programming," in *Proceedings of the Innovative Parallel Computing (InPar '12)*, pp. 1–13, IEEE, San Jose, Calif, USA, May 2012.

[25] D. Grewe and M. F. P. O'Boyle, "A static task partitioning approach for heterogeneous systems using OpenCL," in *Proceedings of the 20th International Conference on Compiler Construction (CC '11)*, pp. 286–305, Saarbrücken, Germany, March 2011.

[26] The IMPACT Research Group and UIUC, "Parboil benchmark suite," http://impact.crhc.illinois.edu/Parboil/parboil.aspx.

[27] C.-K. Luk, R. Cohn, R. Muth et al., "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pp. 190–200, June 2005.

[28] Intel, A Guide to Auto-Vectorization with Intel C++ Compilers, http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers.

[29] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, pp. 152–163, June 2009.

[30] A. Ali, U. Dastgeer, and C. Kessler, "OpenCL for programming shared memory multicore CPUs," in *Proceedings of the MULTI-PROG Workshop at HiPEAC*, 2012.

[31] S. Seo, G. Jo, and J. Lee, "Performance characterization of the NAS Parallel Benchmarks in OpenCL," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '11)*, pp. 137–148, Austin, Tex, USA, November 2011.

[32] Intel, "Writing Optimal OpenCL Code with Intel OpenCL SDK," http://software.intel.com/file/37171.

Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 269764, 14 pages
http://dx.doi.org/10.1155/2015/269764

Research Article

Effective SIMD Vectorization for Intel Xeon Phi Coprocessors

Xinmin Tian,[1] Hideki Saito,[1] Serguei V. Preis,[2] Eric N. Garcia,[1] Sergey S. Kozhukhov,[2] Matt Masten,[1] Aleksei G. Cherkasov,[2] and Nikolay Panchenko[2]

[1] *Mobile Computing and Compilers Software and Service Group, Intel Corporation, Santa Clara, CA 95054, USA*
[2] *Mobile Computing and Compilers Software and Service Group, Intel Corporation, 6/1 Prospect Akademika, Novosibirsk 125009, Russia*

Correspondence should be addressed to Xinmin Tian; xinmin.tian@intel.com

Received 15 May 2014; Accepted 29 September 2014

Academic Editor: Sunita Chandrasekaran

Efficiently exploiting SIMD vector units is one of the most important aspects in achieving high performance of the application code running on Intel Xeon Phi coprocessors. In this paper, we present several effective SIMD vectorization techniques such as less-than-full-vector loop vectorization, Intel MIC specific alignment optimization, and small matrix transpose/multiplication 2D vectorization implemented in the Intel C/C++ and Fortran production compilers for Intel Xeon Phi coprocessors. A set of workloads from several application domains is employed to conduct the performance study of our SIMD vectorization techniques. The performance results show that we achieved up to 12.5x performance gain on the Intel Xeon Phi coprocessor. We also demonstrate a 2000x performance speedup from the seamless integration of SIMD vectorization and parallelization.

1. Introduction

The Intel Xeon Phi coprocessor is based on the Intel Many Integrated Core (Intel MIC) architecture, which consists of many small, power efficient, in-order cores, each of which has a powerful 512-bit vector processing unit (SIMD unit) [1]. It is designed to serve the needs of applications that are highly parallel, make extensive use of SIMD vector operations, or are memory bandwidth bound. Hence, it is targeted for highly parallel, high performance computing (HPC) workloads [2] in a variety of fields such as computational physics, chemistry, biology, and financial services [3]. The Intel Xeon Phi Coprocessor 5110P has the following key specifications:

(i) 60 cores, 240 threads (4 threads/core),
(ii) 1.053 GHz,
(iii) 1 TeraFLOP double precision theoretical peak performance,
(iv) 8 GB memory with 320 GB/s bandwidth,
(v) 512 bit wide SIMD vector engine,
(vi) 32 KB L1, 512 KB L2 cache per core,
(vii) fused multiply-add (FMA) support.

One Teraflop theoretical peak performance is computed as follows: 1.053 GHz $\times$ 60 cores $\times$ 8 double precision elements in SIMD vector $\times$ 2 flops per FMA. As such, any compute bound applications trying to achieve high performance on Intel Xeon Phi coprocessors need to exploit a high degree of parallelism and wide SIMD vectors. Using a 512-bit vector unit, 16 single precision (or 8 double precision) floating point (FP) operations can be performed as a single vector operation. With the help of the fused multiply-add (FMA) instruction, up to 32 FP operations can be performed at each core at each cycle. In comparison to the current 128-bit SSE and 256-bit AVX vector extensions, this new coprocessor can pack up to 8x and 4x the number of operations into a single instruction, respectively.

Wider SIMD vector units cannot be effectively utilized by simply extending the vectorizer for Intel SSE and Intel AVX architecture. Consider the following simple example. There exists a scalar loop that executes N-iterations. Using the vector length of VL, a vector loop would execute floor (N/VL) full vector iterations followed by N mod VL scalar remainder iterations. Unless N is sufficiently larger than VL, executing N mod VL scalar iterations can still be a significant portion of the vector execution of such a loop. In what follows, we

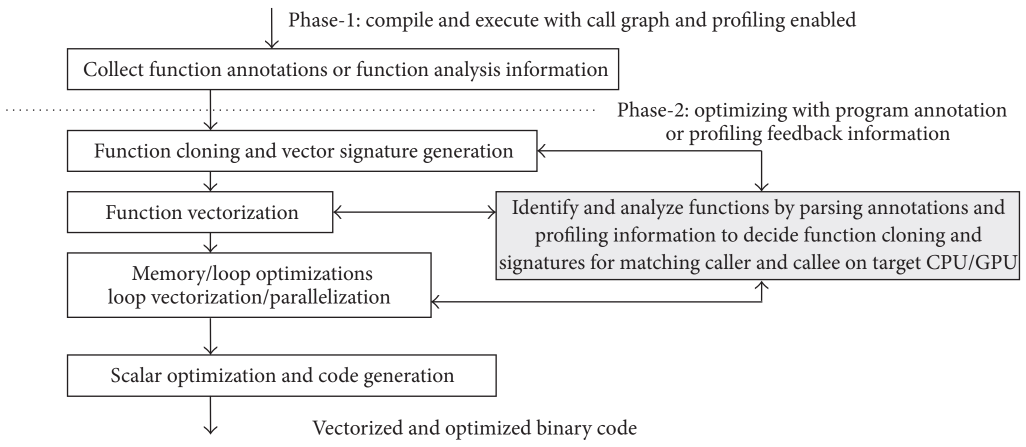

Phase-1: compile and execute with call graph and profiling enabled
Collect function annotations or function analysis information
Phase-2: optimizing with program annotation or profiling feedback information
Function cloning and vector signature generation
Function vectorization
Identify and analyze functions by parsing annotations and profiling information to decide function cloning and signatures for matching caller and callee on target CPU/GPU
Memory/loop optimizations loop vectorization/parallelization
Scalar optimization and code generation
Vectorized and optimized binary code

Figure 1: SIMD vector compilation infrastructure for function and loop vectorization.

will discuss two approaches in handling such "less-than-full-vector" situations: the first technique is masked vectorization and the second technique is small matrix optimization and 2-dimensional (2D) vectorization.

Furthermore, architectural or microarchitectural differences between Intel Xeon Phi coprocessors and Intel Xeon processors necessitate that new compiler techniques be developed. This paper focuses on three SIMD vectorization techniques and makes the following contributions.

(i) We propose an extended compiler scheme to vectorize short trip-count loops and peeling and remainder loops that are classified as "less-than-full-vector" cases, with a masking capability supported by the Intel MIC architecture.

(ii) We describe our specific data alignment strategies for achieving optimal performance through vectorization, as the Intel MIC architecture is much more demanding on memory alignment than the Intel AVX architecture [4].

(iii) We describe our 2-dimensional vectorization method which is beyond the conventional loop vectorization for small matrix transpose and multiplication operations by fully utilizing long SIMD vector units, swizzle, shuffle, and masking support on the Intel MIC architecture.

The rest of this paper is organized as follows: Section 2 provides a high-level overview of Intel C/C++ and Fortran compilers. In Section 3, the compiler details of "less-than-full-vector" loop vectorization are described and discussed. Specific data alignment strategies for the Intel Xeon Phi coprocessor and the schemes of performing data alignment optimization are discussed in Section 4. Section 5 presents the 2D vectorization methods for small matrix transpose and multiplication. Section 6 discusses related work. Section 7 provides the performance results with a set of workloads and microbenchmarks. Section 8 concludes the paper.

2. Compiler Architecture for Vectorization

This section describes the Intel C/C++ and Fortran compiler support for the Intel Xeon Phi coprocessor at a high level with respect to loop vectorization and the translation and optimization of SIMD *vector* extensions [5–7]. The compiler translates serial C/C++ and Fortran code via automatic loop analysis or based on annotations using the SIMD pragma and *vector* attributes into SIMD instruction sequences. The compilation process is amenable to many optimizations such as loop parallelization, memory locality optimizations, classic loop transformations and optimizations, redundancy elimination, and dead code elimination before and after the loop/function vectorization. Figure 1 depicts the SIMD compilation infrastructure of the Intel C/C++ and Fortran compilers for automatic loop vectorization and compiling SIMD pragma, *vector* function annotations, and associated clauses. The framework consists of four major parts.

(i) Perform automatic loop analysis and identify and analyze programmer annotated functions and loops by parsing and collecting function and loop vector properties. In addition, our compiler framework can apply interprocedural analysis and optimization with profiling and call-graph creation for automatic function vectorization.

(ii) Generate vectorized function variants with properly constructed signatures via function cloning and vector signature generation.

(iii) Vectorize SIMD for loops that are identified by the compiler or annotated using SIMD extensions (#pragma SIMD can be used to vectorize outer loops) and cloned vector function bodies and all arguments by leveraging and extending our automatic loop vectorizer.

(iv) Enable classical scalar, memory, and loop optimizations and parallelization effectively, before or after loop and function vectorization, for achieving good performance.

```
float x, y[31];
for (k=0; k<31; k++) {
    x = x + fsqrt(y[k]);
}
```

Algorithm 1

3. Less-than-Full-Vector Loop Vectorization with Masking

Intel Xeon Phi coprocessor provides long (512-bit) SIMD vector hardware support for exploiting more vector-level parallelism. The long SIMD vector unit imposes the requirement of packing more scalar loop iterations into a single vector loop iteration, which also results in more iterations in the peeling loop, and/or in the remainder loop remaining nonvectorized, due to the fact that they do not constitute the full SIMD vector (or less-than-full-vector) unit of Intel MIC architecture. For example, consider the short trip-count loop as shown in Algorithm 1.

When the loop is vectorized for Intel SSE2 with vector length = 4 (128-bit), the remainder loop will have 3 iterations. When the loop is vectorized for the Intel MIC architecture with vector length = 16 (512-bit), the remainder loop will have 15 iterations. In another situation, if the loop is unrolled by 16, then the remainder loop will have 15 iterations, leaving the remaining 15 iterations in a scalar execution form. Thus, vectorizing the peeling and remainder loops (i.e., short trip-count loop in general) is very important for the Intel MIC architecture. This section describes how to apply vectorization, with masking support, to peeling and remainder loops (i.e., short trip-count loop) with special guarding masks to prevent the SIMD code from exceeding original loop and memory access boundaries. At a high level, the following steps describe our vectorization scheme without vectorization of peeling and remainder loops.

(i) s0: select alignment, vector length, and unroll factor.

(ii) s1: generate alignment setup code.

(iii) s2: compute the trip count of the peeling loop.

(iv) s3: emit the scalar peeling loop.

(v) s4: generate the vector loop initialization code.

(vi) s5: emit the main vector loop.

(vii) s6: compute the trip count of the remainder loop.

(viii) s7: emit the scalar remainder loop.

Given the simple example as shown in Algorithm 2, the loop trip-count "n" and the pointer "y" ($\&y[0]$) have a memory alignment that is unknown at compile time.

On the Intel MIC architecture the vector length is 512 bits, which requires 64-byte alignment for efficient memory accesses. To achieve 64-byte aligned memory loads/stores, we need to pack 16 float (32-bit) elements for each single vector iteration and generate a peeling loop. Pseudocode 1 shows the vectorized loop based on the vectorization steps

```
float foo(float *y, int n)
{   int  k; float x = 10.0f;
    for (k = 0; k < n; k++) {
      x = x + fsqrt(y[k])
    }
    return x;
}
```

Algorithm 2

$[s0, s1, \ldots, s7]$ described above. The "less-than-full-vector" loops, that is, the peeling and remainder loops, are not vectorized.

Note that we performed loop unrolling for the main vectorized loop, which allows the hardware to issue more instructions per cycle by hiding memory access latency and reducing branching. To enable the "less-than-full-vector" (i.e., peeling loop, remainder loop, or short trip-count loop) vectorization, the loop vectorization scheme is extended as below.

(i) s0: select alignment, vector length and unroll factor.

(ii) s1: generate alignment setup code.

(iii) s2: compute the trip count of peeling loop.

  (a) Create a vector of 16 elements with value $\langle 0, \ldots, 15\rangle$.

  (b) Create a vector of 16 elements with value $\langle \text{peeledTripCount}, \ldots, \text{peeledTripCount}\rangle$.

(iv) s3: emit the vectorized peeling loop with masking operations.

(v) s4: generate the main vector loop initialization code.

(vi) s5: emit the main vector loop.

(vii) s6: compute the trip count of the remainder loop.

  (a) Create a vector of 16 elements with the value $\langle \text{mainTripCount}, \ldots, \text{mainTripCount+15}\rangle$.

  (b) Create a vector of 16 elements with the value $\langle \text{origTripCount}, \ldots, \text{origTripCount}\rangle$.

(viii) s7: emit the vectorized remainder loop with masking operations.

Pseudocode 2 shows the vectorized loops based on the extended vectorization schemes $[s0, s1, \ldots, s7]$ described as above.

In the cases of short trip-count loop vectorization of peeling and remainder loops with runtime trip-count and alignment checking, loops are vectorized as efficiently as possible. These loops are vectorized with optimal vector lengths and an optimal amount of profitable unrolling regardless of a known loop trip count. This provides better utilization of SIMD vector hardware without sacrificing the performance of short loops. This scheme allows us to completely eliminate

```
misalign = &y[0] & 63
peeledTripCount = (63 - misalign)/sizeof(float)
x = 10.0f;
do k0 = 0, peeledTripCount-1  // peeling loop
   x = x + fsqrt(y[k0])
enddo
x1_v512 = (m512)0
x2_v512 = (m512)0
mainTripCount = n - ((n - peeledTripCount) & 31)
do k1 = peeledTripCount, mainTripCount-1, 32
  x1_v512 = _mm512_add_ps(_mm512_fsqrt(y[k1:16]),x1_v512)
  x2_v512 = _mm512_add_ps(_mm512_fsqrt(y[k1+16:16]), x2_v512)
enddo
// perform vector add on two vector x1_v512 and x2_v512
x1_v512 = _mm512_add_ps(x1_v512, x2_512);
// perform horizontal add on all elements of x1_v512, and
// the add x for using its value in the remainder loop
x = x + _mm512_hadd_ps(x1_512)
do k2 = mainTripCount, n // Remainder loop
   x = x + fsqrt(y[k2])
enddo
```

Pseudocode 1: Pseudocode without vectorizing "less-than-full-vector" loops.

```
misalign = &y[0] & 63
peeledTripCount = (63 - misalign) / sizeof(float)
x = 10.0f;
// create a vector: <0,1,2,...15>
k0_v512 = _mm512_series_pi(0, 1, 16)
// create vector: all 16 elements are peeledTripCount
peeledTripCount_v512 = _mm512_broadcast_pi32(peeledTripCount)
x1_v512 = (m512)0
x2_v512 = (m512)0
do k0 = 0, peeledTripCount-1, 16
   // generate mask for vectorizing peeling loop
   mask = _mm512_compare_pi32_mask_lt(k0_v512, peeledTriPCount_v512)
   x1_v512 = _mm512_add_ps_mask(_mm512_fsqrt(y[k0:16]), x1_v512, mask)
enddo
mainTripcount = n - ((n - peeledTripCount) & 31)
do k1 = peeledTripCount, mainTripCount-1, 32
  x1_v512 = _mm512_add_ps(_mm512_fsqrt(y[k1:16]), x1_v512)
  x2_v512 = _mm512_add_ps(_mm512_fsqrt(y[k1+16:16]), x2_v512)
enddo
// create a vector: <mainTripCount,mainTripCount+1 ... mainTripCount+15>
k2_v512 = _mm512_series_pi(mainTripCount, 1, 16)
// create a vector: all 16 elements has the same value n
n_v512 = _mm512_broadcast_pi32(n)
step_v512 = _mm512_broadcast_pi32(16)
do k2 = mainTripCount, n, 16 // vectorized remainder loop
   mask = _mm512_compare_pi32_mask_lt(k2_v512, n_v512)
   x1_v512 = _mm512_add_ps_mask(_mm512_fsqrt(y[k2:16]), x1_v512, mask)
   k2_v512 = _mm512_add_ps(k2_v512, step_v512)
enddo
x1_v512 = _mm512_add_ps(x1_v512, x2_512);
// perform horizontal add on 8 elements and final
// reduction sum to write the result back to x.
x = x + _mm512_hadd_ps(x1_512)
```

Pseudocode 2: Pseudocode with vectorizing "less-than-full-vector" loops using mask.

scalar execution of the loop in favor of masked SIMD vector code generation. Special properties of the mask are used to match unmasked code generation in most cases. For example, masked scalar memory loads that could be unsafe under an empty mask are considered safe under a remainder mask since it is never empty.

Without adding the capability of short trip-count loop vectorization, the loops in the ConvolutionFFT2D benchmark with 7 iterations and double precision data type would end up as a fully scalar execution. Applying vectorization with masking to these short trip-count loops results in a ~2x to ~5x speedup for the 7-iteration short trip-count (or less-than-full-vector) loops in the ConvolutionFFT2D benchmarks on the Intel MIC Architecture.

4. Alignment Strategy and Optimization

The Intel Xeon Phi coprocessor is much more sensitive to data alignment than the Intel Xeon E5 processor, so developing an Intel MIC oriented alignment strategy and optimization schemes is one of the key aspects for achieving optimal performance.

(i) Similar to Intel SSE4.2, the SIMD load+op instructions require vector size alignment, which is 64-byte alignment for the Intel MIC architecture. However, simple load/store instructions require the alignment information to be known at compile time on the Intel Xeon Phi coprocessor.

(ii) Different from prior Intel SIMD extensions, *all* SIMD load/store instructions including gather/scatter require at least element size alignment. Misaligned elements will cause a fault. This necessitates the Intel MIC architecture ABI [8] to require that all memory accesses be elementwise aligned.

(iii) There are no special unaligned load/store instructions in the Intel Initial Many Core Instruction (Intel IMCI) set. This is overcome by using unpacking loads and packing stores that are capable of dealing with unaligned (element-aligned) memory locations. Due to their unpacking and packing nature, these instructions cannot be directly used for masked loads/stores, except under special circumstances.

(iv) The faulting nature of masked memory access instructions in Intel IMCI adds extra complexity to those instructions addressing data outside paged memory and may fail even if actual data access is masked out. The exceptions are gather/scatter instructions.

Therefore, the compiler aggressively performs data alignment optimizations using traditional techniques such as alignment peeling and alignment multiversioning.

Alignment peeling implies the creation of a preloop that executes several iterations on unaligned data in order to reach an aligned memory address. As a result, most of these iterations are executed using aligned SIMD operations. The preloop can be vectorized with masking as described in Section 2. Unfortunately, this scheme works only for one set of coaligned memory addresses, and the others are assumed to be unaligned. In addition, our multiversioning optimization can be applied to the second set of coaligned locations by examining them dynamically. Aligned or unaligned operations are used based on the results of the examination.

For unmasked unaligned (element-aligned) vector loads and stores, the compiler uses unpacking/packing load and store instructions. They are safe in this scenario and perform much better than gather/scatter instructions. If the compiler cannot prove the safety of the entire address range of a particular memory access, it inserts a zero-mask check in order to avoid a memory fault. All instructions with the same mask are emitted under a single check to avoid execution under the empty mask and to eliminate multiple checks of the same condition.

Unpacking and packing instructions may cause fault when they are used with a mask, as they may address masked-out invalid memory. On-the-fly data conversion may cause fault even without masking. Thus, for unaligned masked and/or converting loads/stores, the compiler uses gather/scatter instructions instead of safety, even though this degrades performance. Memory faults would never happen if each memory access had at least one vector (64 bytes) of memory paged after its initial address. This can be achieved by padding each data section in the program and each dynamically allocated object with 64 bytes. For developers who are willing to do the padding to achieve optimal performance from masked code, the compiler knob-opt-assume-safe-padding was introduced. Under this knob, unaligned masked and/or converting load/store operations are emitted as unpacking loads/packing stores.

(i) In unmasked converting cases, as well as cases with peel/remainder masks, the compiler emits loads/stores directly. The mask in this case will work since it is dense.

(ii) For an arbitrary masking scenario, an unmasked load unpack instruction is used, which is safe due to the padding assumption, followed by a masked move (blend). The "nonempty-mask" check guarantees that the 64-byte padding is always enough for safety; that is, at least one item within the vector is to be loaded. Thus, the tail end of the memory access is within 64 bytes from meaningful data.

The safe-padding optimization has provided notable improvements on a number of benchmarks, for example, 10% gain on BlackScholes and selected Molecular Dynamics kernels.

5. Small Matrix Operations 2D Vectorization

Frequently seen in HPC workloads, operations on small matrices are a growing, profitable set of calculations for vectorization on Intel Xeon Phi coprocessors. With the wider SIMD unit support, the Intel C/C++ and Fortran compilers are enhanced to vectorize common operations on small matrices along 2 dimensions. Small matrices are matrices whose data can reside entirely in one or two 512-bit SIMD

```
real, dimension(4,4):: A, B, C
real sum
integer j, l, i
do j = 1, 4
  do l = 1, 4
    sum = 0.0
    do i = 1, 4
      sum = sum + A(i,l) * B(i,j)
    enddo
    C(l,j) = sum
  enddo
enddo
```

Algorithm 3: Small matrix multiplication summation.

Table 1: Contents of vector register A_v512 after load.

A_v512	A[1][1]	A[1][2]	A[1][3]	A[1][4]
	A[2][1]	A[2][2]	A[2][3]	A[2][4]
	A[3][1]	A[3][2]	A[3][3]	A[3][4]
	A[4][1]	A[4][2]	A[4][3]	A[4][4]

Table 2: Contents of vector register B_v512 after load.

B_v512	B[1][1]	B[1][2]	B[1][3]	B[1][4]
	B[2][1]	B[2][2]	B[2][3]	B[2][4]
	B[3][1]	B[3][2]	B[3][3]	B[3][4]
	B[4][1]	B[4][2]	B[4][3]	B[4][4]

Table 3: A′_v512 after zero initialization.

A′_v512	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

registers. Consider the example Fortran loop nest with 32-bit float (or real) type as shown in Algorithm 3.

With nonunit stride references present in the inner loop of Algorithm 3, the conventional inner loop vectorization will not provide the most efficient vectorization of the loop nest. The outer loop vectorization faces similar issues. The Intel C/C++ and Fortran compiler employs the wider SIMD vector unit of the Intel MIC architecture and vectorizes this example loop across all three loop nesting levels, named as 2-dimensional (2D) vectorization on small matrices.

The vectorization approach is detailed below with vector intrinsic pseudocode. For visualization, Tables 1–13 depict a snapshot of the various vector unit contents after each corresponding instruction. Tables 1–13 represent a vector unit, whose name is in the leftmost column and its contents in the rightmost four columns. Of the rightmost four columns, the lowest addressed element is in the top left corner and each consecutive element follows a row-major addressing order.

Table 4: Vector register contents after first shuffle.

A′_v512	A[1][1]	0	0	0
	0	A[2][2]	0	0
	0	0	A[3][3]	0
	0	0	0	A[4][4]

Table 5: Vector register contents after second shuffle.

A′_v512	A[1][1]	0	0	A[4][1]
	A[1][2]	A[2][2]	0	0
	0	A[2][3]	A[3][3]	0
	0	0	A[3][4]	A[4][4]

Table 6: Vector register contents after third shuffle.

A′_v512	A[1][1]	0	A[3][1]	A[4][1]
	A[1][2]	A[2][2]	0	A[4][2]
	A[1][3]	A[2][3]	A[3][3]	0
	0	A[2][4]	A[3][4]	A[4][4]

Table 7: Vector register contents after the final shuffle.

A′_v512	A[1][1]	A[2][1]	A[3][1]	A[4][1]
	A[1][2]	A[2][2]	A[3][2]	A[4][2]
	A[1][3]	A[2][3]	A[3][3]	A[4][3]
	A[1][4]	A[2][4]	A[3][4]	A[4][4]

Table 8: Vector register contents after load with broadcast.

t1_v512	A[1][1]	A[2][1]	A[3][1]	A[4][1]
	A[1][1]	A[2][1]	A[3][1]	A[4][1]
	A[1][1]	A[2][1]	A[3][1]	A[4][1]
	A[1][1]	A[2][1]	A[3][1]	A[4][1]

Table 9: Vector register contents illustrating swizzle.

t2_v512	B[1][1]	B[1][2]	B[1][3]	B[1][4]
	B[1][1]	B[1][2]	B[1][3]	B[1][4]
	B[1][1]	B[1][2]	B[1][3]	B[1][4]
	B[1][1]	B[1][2]	B[1][3]	B[1][4]

First, array data is loaded into a vector unit. With a wider SIMD vector unit, the compiler is able to load the entire A and B matrix each into a single vector unit.

(a) Matrices A and B are loaded into two SIMD registers:

```
//Load A matrix from memory into vector
register,
A_v512 = ⟨A[1][1], A[1][2], ... ...,
  A[4][3], A[4][4]⟩.
```

For more details see Table 1.

```
//Load B matrix from memory into vector
register,
B_v512 = ⟨B[1][1], B[1][2], ... ...,
B[4][3], B[4][4]⟩.
```

For more details see Table 2.

Table 10: C_v512 vector unit contains elementwise product of t1_v512 and t2_v512.

C_v512	t1_v512 ∗ t2_v512

Table 11: t1_v512 vector register contents illustrating final load with broadcast.

t1_v512	A[1][4]	A[2][4]	A[3][4]	A[4][4]
	A[1][4]	A[2][4]	A[3][4]	A[4][4]
	A[1][4]	A[2][4]	A[3][4]	A[4][4]
	A[1][4]	A[2][4]	A[3][4]	A[4][4]

Table 12: t2_v512 vector register contents illustrating final swizzle.

t2_v512	B[4][1]	B[4][2]	B[4][3]	B[4][4]
	B[4][1]	B[4][2]	B[4][3]	B[4][4]
	B[4][1]	B[4][2]	B[4][3]	B[4][4]
	B[4][1]	B[4][2]	B[4][3]	B[4][4]

Table 13: Final C_v512 vector unit contains sum of existing values of C_v512 and elementwise products t2_v512 and t_v512.

C_v512	t2_v512 ∗ t1_v512 + C_v512

Next, the compiler optimizes the multiplication operation between matrix A and matrix B, through a series of data layout transformations and vector multiplication and addition operations. The compiler identifies a matrix multiplication in this loop and permutes the elements in matrix A and matrix B setting up simple vector multiplications and additions.

(b) We can simplify the multiplication needed through a transposition of the elements of A, followed by a multiply and add of each row B and with each row of transposed A. We start by transposing the elements of A.

```
//First, create a vector unit of zeros.
A′_v512 = _mm512_setzero()
```

For more details see Table 3.

For the transpose operation, we use a set of new Intel MIC _mm512_mask_shuf128 × 32( ) intrinsic calls. Similarly in classic architecture, this shuffle intrinsic is bound by four 128-bit "lanes" in each vector register. Thus, this intrinsic contains arguments for permutation patterns for each of the four 128-bit lanes, as well as a permutation pattern for each of the four 32 bit boundaries within each of those lanes. The arguments are as follows:

_m512 res = _mm512_mask_shuf128 × 32(_m512 v1, (I16) vmask, _m512 v2, (SI32)perm128, (SI32)perm32),

(i) res: result vector unit,

(ii) v1: blend-to-vector unit; the values in this vector unit will be blended with the shuffled elements of the v2, according to the write mask,

(iii) vmask: write mask; the write mask is a bit vector specifying which elements to overwrite in v1 with the shuffle elements of v2,

(iv) v2: incoming data vector unit; this vector unit holds the elements which are to be shuffled,

(v) perm128: 128-bit lane permutation; this value specifies the permutation order of the vector unit's 128-bit lanes,

(vi) perm32: elementwise permutation; this value specifies the permutation order of the each of the four 32 bit boundaries within each 128-bit lane,

```
//Begin transpose operation by
shufflingelements into
//desired order. Shuffle used to insert
matrix diagonal
//into transpose result vector unit,
A′_v512 = _mm512_mask_shuf128 ×32(A′_
v512, 0 × 8421,A_v512,
_MM_PERM_DCBA, _MM_PERM_DCBA).
```

For more details see Table 4.

```
//Shuffle the next four elements and
blend-in with the
//elements written from previous
shuffle,
A′_v512 = _mm512_mask_shuf128 × 32(A′_
v512, 0 × 4218,A_v512, _MM_PERM_CBAD,
_MM_PERM_ADCB).
```

For more details see Table 5.

```
//Shuffle the next four elements and
blend-in with the
//elements written from previous
shuffle,
A′_v512 = _mm512_mask_shuf128 × 32(A′_
v512, 0 × 2184, A_v512, _MM_PERM_BADC,
_MM_PERM_BADC)
```

For more details see Table 6.

```
//Shuffle the final four elements and
blend-in with the
//elements written from previous shuffle
to obtain the
//complete transpose,
A′_v512 = _mm512_mask_shuf128 × 32(A′_
v512, 0 × 1842, A_v512, _MM_PERM_ADCB,
_MM_PERM_CBAD).
```

For more details see Table 7.

After the elements of matrix A have been permuted through transposition, each element of A and B is now in the correct position within each vector unit for a vector product, resulting in the same behavior as the dot product of rows and columns.

(c) Next, we perform the multiplication of each row of the transposed A with each row of B, maintaining a sum of the products from row to row:

```
//Load the first row of A′_v512 and
broadcast that row to
//each of the remaining three rows
t1_v512 = _mm512_extload_ps(A′_v512[0 : 4],
_MM_FULLUPC_NONE, _MM_BROADCAST_4 × 16,
0).
```

For more details see Table 8.

Another useful intrinsic used in this optimization is the Intel MIC _mm512_swizzle_ps( ) intrinsic. This intrinsic is similar to that of the shuffle above except it only permutes each 128-bit lane and not each of the 32 boundaries within those lanes. The arguments are as follows:

_m512 res = _mm512_swizzle_ps(_mm512 v1, SI32 perm)

(i) res: result vector unit,

(ii) v1: incoming data vector unit to be permuted,

(iii) perm: permutation pattern for each 128-bit lane,

```
//Load the first row of B_v512 and
broadcast that row to
//each of the remaining three rows
t2_v512 = _mm512_swizzle_ps (B_v512,
_MM_SWIZ_REG_AAAA).
```

For more details see Table 9.

```
//Multiply each element of t1_v512
with each element of
//t2_v512 and store result in C_v512
C_v512 = _mm512_mul_ps (t1_v512, t2_v512).
```

For more details see Table 10.

```
//Load the second row of A′_v512 and
broadcast that row
//to each of the remaining three rows
t1_v512 = _mm512_extload_ps(A′_
v512[4 : 8], _MM_FULLUPC_NONE, _MM_
BROADCAST_4 × 16, 0)
//Load the second row of B_v512 and
broadcast that row to
//each of the remaining three rows
t2_v512 = _mm512_swizzle_ps (B_v512,
_MM_SWIZ_REG_BBBB).
```

Each subsequent multiplication must be accumulated for each row. These multiplications and additions are the corresponding dot product of rows and columns found in matrix multiplication, but because of the earlier transpose, no further permuting is required:

```
//Add the existing values of C_v512
with the product of
//t1_v512 and t2_v512 and store
result in C_v512
C_v512 = _mm512_madd213_ps (t2_v512,
t1_v512, C_v512)

//Load the third row of A′_v512
and broadcast that row to
//each of the remaining three rows
t1_v512 = _mm512_extload_ps(A′_
v512[8 : 12], _MM_FULLUPC_NONE,
_MM_BROADCAST_ 4 × 16, 0)

//Load the third row of B_v512 and
broadcast that row to
//each of the remaining three rows
t2_v512 = _m512_swizzle_ps (B_v512,
_MM_SWIZ_REG_CCCC)

//Add the existing values of C_v512
with the product of
//t1_v512 and t2_v512 and store result
in C_v512
C_v512 = _mm512_madd213_ps (t2_v512,
t1_v512,C_v512)

//Load the fourth row of A′_v512
and broadcast that row
//to each of the remaining three rows
t1_v512 = _mm512_extload_ps(A′_
v512[12:16], _MM_FULLUPC_NONE,
_MM_BROADCAST_4 × 16, 0).
```

For more details see Table 11.

```
//Load the fourth row of B_v512
and broadcast that row to
//each of the remaining three rows
t2_v512 = _mm512_swizzle_ps (B_v512,
_MM_SWIZ_REG_DDDD).
```

For more details see Table 12.

```
//Add the existing values of C_v512
with the product of
//t1_v512 and t2_v512 and store result
in C_v512
C_v512 = _mm512_madd213_ps (t2_v512,
t1_v512, C_v512).
```

For more details see Table 13.

After the simplified matrix multiplication, the loop further requires that results be stored in the C matrix. With all elements correctly computed and residing in vector unit only one store operation is generated.

(d) Finally, the result vector unit of values is stored to the C array:

```
//The elements of vector register
C_v512 are then stored
```

```
    //to memory at &C[1][1]
      ⟨C[1][1],  &C[1][2],  ...  C[4][3],
      &C[4][4]⟩ = C_v512.
```

The 512-bit long SIMD vector unit of the Intel MIC architecture supports consumption of both matrix dimensions for 2D vectorization, fitting an entire small matrix (4×4 float type) into one 512-bit SIMD vector register. This enables more efficient flexible vectorization and optimizations for small matrix operations. For example, the scalar version of single precision 4×4 matrix multiply computation naively executes 128 memory loads, 64 multiplies, 64 additions, and 16 memory stores. The small matrix 2D vectorization reduces instructions to 2 vector loads from memory, 4 multiplications, 4 shuffles, 4 swizzles, 3 additions, and 1 vector store to memory for a reduction of approximately 15x in number of instructions.

6. Performance Evaluation

This section presents the performance results measured on an Intel Xeon Phi coprocessor system using a set of workloads and microbenchmarks.

6.1. Workloads. We have selected a set of workloads to demonstrate the performance benefits and importance of SIMD vectorization on the Intel MIC architecture. These workloads exhibit a wide range of application behavior that can be found in areas such as high performance computing, financial services, databases, image processing, searching, and other domains. These workloads include the following.

6.1.1. NBody. NBody computations are used in many scientific applications such as astrophysics [9] and statistical learning algorithms [10]. The main computation involves two loops that iterate over the bodies and computes a pairwise interaction between them.

6.1.2. 2D 5 × 5 Convolution. Convolution is a common image filtering computation used to apply effects such as blur and sharpen. For a given 2D image and a 5×5 spatial filter containing weights, this convolution computes the weighted sum for the neighborhood of the 5×5 set of pixels.

6.1.3. Back Projection. Back projection is commonly used for performing cone-beam image reconstruction of CT projection values [11]. The input consists of a set of 2D images that are "back-projected" onto a 3D volume in order to construct a 3D grid of density values.

6.1.4. Radar (1D Convolution). The 1D convolution is widely used in applications such as radar tracking, graphics, and image processing.

6.1.5. Tree Search. In memory tree structured index search is a commonly used operation in database applications. This benchmark consists of multiple parallel searches over a tree with different queries, where the path through the tree is determined based on the comparison of results of the query and node value at each tree level.

Table 14: Target system configuration.

System parameters	Intel Xeon Phi processor
Chips	1
Cores/threads	61 and 244
Frequency	1 GHz
Data caches	32 KB L1, 512 KB L2 per core
Power budget	300 W
Memory capacity	7936 MB
Memory technology	GDDR5
Memory speed	2.75 (GHz) (5.5 GT/s)
Memory channels	16
Memory data width	32 bits
Peak memory Bandwidth	352 GB/s
SIMD vector length	512 bits

6.2. System Configuration. The detailed information on the configuration of the Intel Xeon Phi Coprocessor used for the performance study and for evaluating the effectiveness of SIMD vectorization techniques is provided in Table 14.

6.3. Performance Results. All benchmarks were compiled as native executable using the Intel 13.0 product compilers and run on the Intel Xeon Phi coprocessor system specified in Table 14. To demonstrate the performance gains obtained through the SIMD vectorization, two versions of the binaries were generated for each workload. The baseline version was compiled with OpenMP parallelization only (-mmic -openmp -novec); the vectorized version is compiled with vectorization (default ON) and OpenMP parallelization (-mmic -openmp).

The performance scaling is derived from the OpenMP-only execution and OpenMP with 512-bit SIMD vector execution on the Intel Xeon Phi coprocessor system that we described at beginning of this section. That is, when the workload contains 32-bit single precision computations, 16-way vectorization may be achieved. When the workload contains 64-bit double-precision computations, 8-way vectorization is achieved.

Figure 2 shows the normalized SIMD performance speedup of five workloads. The generated SIMD code of these workloads achieved SIMD speedup ranging from 2.25x to 12.45x. Besides those classical HPC applications with regular array accesses and computations, the workload with a large amount of branching codes, such as tree search used in database applications, achieves 2.25x speedup as well with SIMD vectorization based on the masking support in the Intel MIC architecture.

6.3.1. Impact of Less-than-Full-Vector Loop Vectorization. To examine the impact of the less-than-full-vector loop vectorization, a simple microbenchmark was written with three

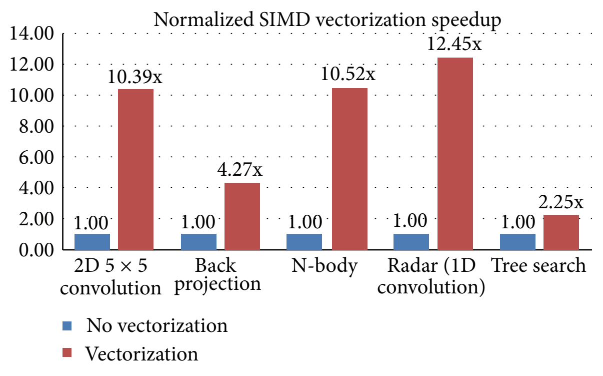


Figure 2: Performance results of workloads.

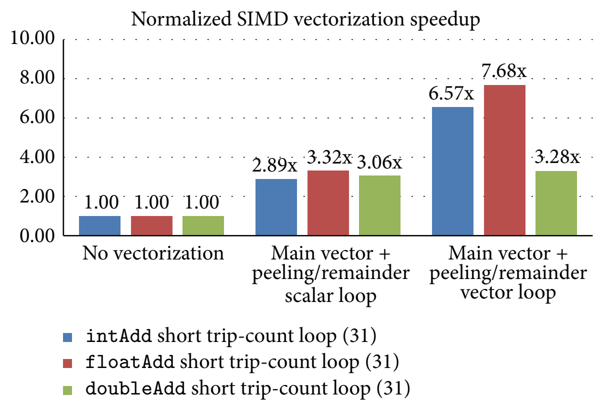


Figure 3: Performance gain with "less-than-full-vector" loop vectorization.

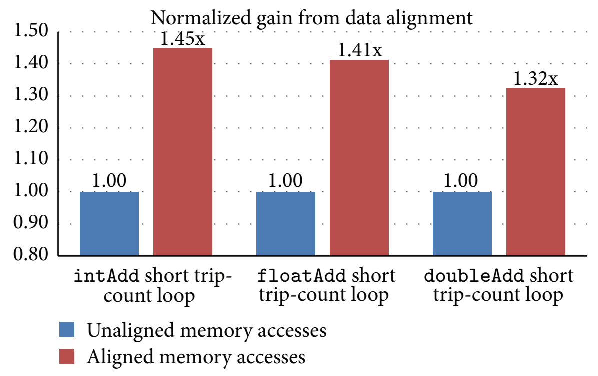


Figure 4: Performance gain with data alignment.

small kernel functions: `intAdd`, `floatAdd`, and `doubleAdd`. Each of them has a short trip-count loop that takes 3 arrays, a, b, c of size 31, and does an elementwise addition with respect to `int`, `float`, and `double` data types. The vector length is 16 iterations for loops in the `intAdd` and `floatAdd` kernels and 8 iterations for the loop in the `doubleAdd` kernel function. This experimental setup ensures the `intAdd` and `floatAdd` loops contain a 15-iteration remainder loops, and the `doubleAdd` loop contains a 7-iteration remainder loop which can be vectorized with the "*less-than-full-vector*" loop SIMD vectorization technique using masking support described in the Section 2.

Figure 3 shows performance gains from vectorization without "*less-than-full-vector*" loop vectorization and with "*less-than-full-vector*" loop vectorization for three short trip-count loops in the `intAdd`, `floatAdd`, and `doubleAdd` kernel functions. The generated SIMD code of these loops achieves a speedup ranging from 2.89x to 3.32x without "*less-than-full-vector*" loop vectorization. With "*less-than-full-vector*" loop vectorization, the performance speedup is improved significantly and ranges from 3.28x to 7.68x. Note that, in this measurement, all data are 64-byte aligned, there are no peeling loops generated, and the aligned memory load/store instructions such as `vmovaps` and `vmovapd` [1] are generated to achieve optimal performance. The next subsection shows the data alignment impact on the Intel MIC architecture.

6.3.2. Impact of Data Alignment. These kernel loops used in Section 6.3.1 are reused for this measurement. In this study, the difference is that we do not provide alignment information of the arraysa, b, and c. Without alignment information, given these loops are short trip-count loops with constant trip count, the compiler generates SIMD instructions:

(i) `vloadunpackld` and `vloadunpackhd` to load data from unaligned memory locations and `vpackstoreld` and `vpackstorehd` [1] to store data to unaligned memory locations for the vectorized main loop,

(ii) `vgatherdps` and `vscatterdps` instructions [1] to load and store for the vectorized remainder loop with write mask.

As shown in Figure 4, with data alignment information, the performance of SIMD execution is 1.45x, 1.41x, and 1.32x better than unaligned cases with respect to `int`, `float`, and `double` types of three kernel functions. The alignment optimization described in Section 3 is critical to achieving optimal performance on Intel MIC architecture.

6.3.3. Impact of Small Matrix 2D Vectorization. Small matrix operations such as addition and multiplication have served as important parts of many HPC applications. A number of classic compiler optimizations such as loop complete unrolling, partial redundancy elimination (PRE), scalar replacement, and partial summation have been developed to achieve optimal vector execution performance. The conventional inner or outer loop vectorization for 3-level loop nests of 4 × 4 matrix operations is not performing well on Intel Xeon Phi coprocessor due to

(i) less effective use of 512-bit long SIMD unit, for example, for 32-bit float data type, when either inner loop or outer loop is vectorized. In this case 4-way vectorization is used instead of 16-way vectorization,

(ii) side-effects on classic optimizations, for example, the partial redundancy elimination, partial summation, and operator strength reduction, when the loop is vectorized.

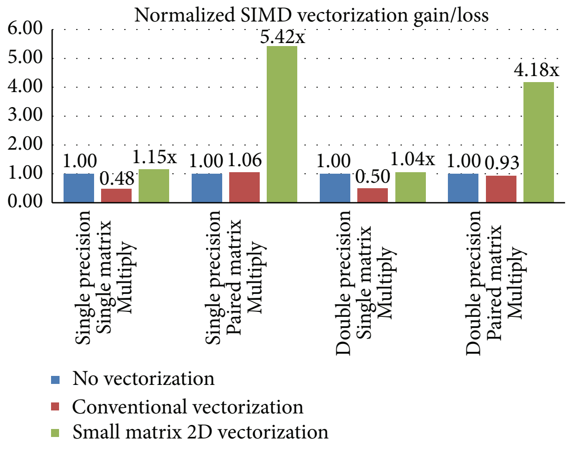


FIGURE 5: Performance gain/loss with SIMD vectorization.

As shown in Figure 5, the convention loop vectorization on small matrix (4 × 4) operations does cause performance degradation. For both cases of single precision and double precision matrix (4 × 4) multiplications, the performance degradation is ~50% when comparing against cases without vectorization, which are used as the baseline performance. In the case of the paired matrix multiplication, there are two matrix (4 × 4) multiplications done in a single loop nest, and matrix B is transposed for computing `sumy` (for more details see Algorithm 4).

The classical loop optimizations are not as effective as for the single matrix multiplication case due to the transpose operation of `matrix B` and paired matrix multiplications in the loop. Thus, the performance achieved with classical loop optimization is on-par with applying conventional loop vectorization, and no notable performance difference is observed as shown in Figure 5. Promisingly, applying the small matrix 2D vectorization we proposed in Section 4, we achieved a performance speedup 1.15x/1.04x for single matrix (4 × 4 float/double type) multiplication and a speedup 5.42x/4.18x for paired matrix (4 × 4 float/double type) transpose and multiplication, which demonstrates the effectiveness of small matrix 2D vectorization using long SIMD vector unit supported by Intel Xeon Phi coprocessor.

7. Seamless Integration with Threading

Effectively exploiting the power of a coprocessor like Xeon Phi requires that both thread- and vector-level parallelism are exploited. While the parallelization topic is beyond the scope of this paper, we would still like to highlight that the SIMD vector extensions can be seamlessly integrated with threading models such as OpenMP* 4.0 supported by the Intel compilers. Given the Mandelbrot example Mandelbrot computes a graphical image representing a subset of the Mandelbrot set (a well-known 2D fractal shape) out of a range of complex numbers. It outputs the number of points inside and outside the set.

In the *mandelbrot* workload, the function "*mandel*" in the *mandelbrot* program is a hot function and a candidate for SIMD vectorization, so we can annotate it with `#pragma omp declare SIMD`. At the caller site, the hot loop is a double nested *for* loop, the outer *for* loop is asserted with "omp parallel for" for threading, and the inner loop is asserted with "omp SIMD" for vectorization as shown in Algorithm 5. Note that the "guided" scheduling type is used for achieving a good load balance, as each call to "mandel" function does varying amount of work in terms of execution time due to "break" exit of the loop.

Figure 6 shows that the SIMD vectorization alone delivers a ~16x speedup, built with option –mmic –openmp –std=c99–O3 over the serial execution. The OpenMP parallelization delivers a 62.09x speedup with 61 threads using 61 cores with Hyperthreading OFF, a speedup 131.54x with 244 threads (61 cores with Hyperthreading ON, 4 HT threads per core) over the serial execution. The OMP PARALLEL FOR and SIMD combined execution delivers an OMP PAR + SIMD speedup 2067.9x with 244 threads, running on an Intel Xeon Phi system, which has 61-core on the chip with Hyperthreading ON. The performance scaling from 1 thread to 61 threads is close to linear. In addition, the Hyperthreading support delivers a ~2x performance gain by comparing the 244-thread speedup with the 61-thread speedup, which is better than the well-known 20%–30% expectation on the performance gain from Hyperthreading technology due to the nature of less computing resource contention in the workload, and 4 busy HT threads did hide latency well. For the system information details see Section 6.2.

8. Related Work

The compiler vectorization technology [12] had been one of the key loop transformations for traditional vector machine decades ago. However, the recent proliferation of modern SIMD architecture [1, 4] poses new constraints such as data alignment, masking for control flow, nonunit stride access to memory, and the fixed-length nature of SIMD vectors that shall demand more advanced vectorization technologies and vectorization friendly programming language extensions [7].

In the past three plus decades, the rich body of SIMD vectorization capabilities has been incorporated in a number of industry and research compilers [5, 6, 12–16]. These include works based on ICC (the Intel compiler) [5, 6], XLC (the IBM compiler) [13, 16], VAST [17], GCC [18, 19], and the SUIF compiler [20]. However, there are many unknown program factors such as loop trip count, memory access stride and patterns, alignment, and control flow complexity at compile-time that pose challenges to the modern optimizing compiler's ability to apply advanced and practical vectorization techniques and fulfill the semantic gap between application programs and the modern processors such as Intel Xeon Phi coprocessor for harnessing its computational power.

Compared to the conventional loop vectorization [5, 12, 20], the "less-than-full-vector" vectorization technique brings extra performance benefits for those vectorizable short

```
do j = 1, 4
  do k = 1, 4
    sumx = 0.0
    sumy = 0.0
    do i = 1, 4
      sumx = sumx + matrixA(i,k) * matrixB(i,j)
      sumy = sumy + matrixA(i,k) * matrixB(j,i)
    enddo
    matrixC(k,j) = sumx
    matirxD(j,k) = sumy
  enddo
enddo
```

Algorithm 4

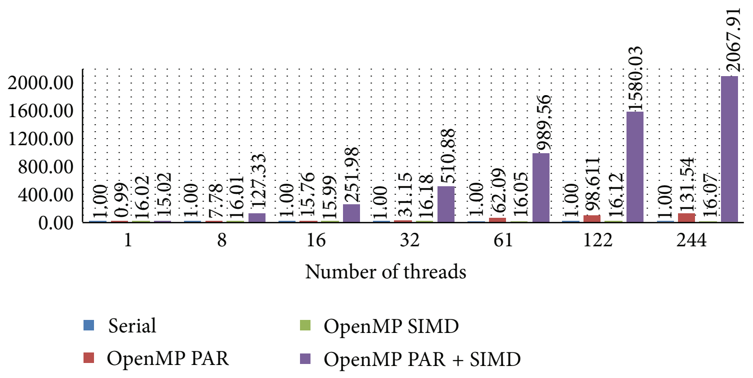


Figure 6: OpenMP* parallel for and SIMD speedup of mandelbrot workload.

trip-count loops, especially when the processor provides the long SIMD unit masking capability like the Intel Xeon Phi coprocessor. Our alignment optimizations are built on top of existing dynamic alignment optimizations as presented in [5, 6]. However, the alignment strategy described in this paper is designed to satisfy the requirement of Intel MIC architecture with optimal SIMD instruction selection and mask utilization for safe and optimal performance. Beyond traditional single-level loop vectorization [5, 12, 16, 18, 19, 21], the small matrix operation 2D vectorization increases vector-parallelism and improves the utilization efficiency of the long SIMD vector unit, swizzle, shuffle, broadcast, and mask support in Intel MIC architecture significantly.

In addition, programming language extensions such as OpenMP* SIMD extensions [22, 23] and Cilk Plus [3, 7] function vectorization and loop vectorization through the compiler has been paving the way to enable more effective vector-level parallelism [7, 22] in both C/C++ and Fortran programming languages. To support these SIMD vector programming models on the Intel Xeon Phi coprocessor effectively, the practical and effective vectorization techniques described in this paper are essential for achieving optimal performance and ensuring SIMD code execution safety on an Intel Xeon Phi coprocessor system.

9. Conclusions

Driven by the increasing prevalence of SIMD architecture in the Intel Xeon Phi coprocessor, we proposed and implemented new vectorization techniques to explore the effective use of its long SIMD units. This paper presented several practical SIMD vectorization techniques such as less-than-full-vector loop vectorization, Intel MIC specific data alignment optimizations, and small matrix operations 2D vectorization for the Intel Xeon Phi coprocessor. A set of workloads from several domains was employed to evaluate the benefits of our SIMD vectorization techniques. The results show that we achieved up to 12.5x performance gain on Intel Xeon Phi coprocessor. Mandelbrot workload demonstrated the seamless integration of SIMD vector extensions with threading and showed a 2067.91x performance speedup with the combined use of OpenMP "parallel for" and "SIMD" constructs using Intel C/C++ compilers on an Intel Xeon Phi coprocessor system.

Intel C/C++ and Fortran compilers are highly enhanced for programmers to harness the computational power of Intel Xeon Phi coprocessors for accelerating highly parallel applications found in chemistry, visual computing, computational physics, biology, financial services, pixel, multimedia,

```
#pragma omp declare SIMD uniform(max_iter) SIMDlen(32)
uint32_t mandel(fcomplex c, uint32_t max_iter)
{
    // Computes number of iterations(count variable)
    // that it takes for parameter c to be known to
    // be outside mandelbrot set
    uint32_t count = 1; fcomplex z = c;
    for (int32_t i = 0; i < max_iter; i += 1) {
      z = z * z + c;
      int t = (cabsf(z) < 2.0f);
      count += t;
      if (t == 0) { break;}
    }
    return count;
}
Caller site code:
int main() {
    ........
    #pragma omp parallel for schedule(guided)
    for (int32_t y = 0; y < ImageHeight; ++y) {
      float c_im = max_imag - y * imag_factor;
      #pragma omp SIMD safelen(32)
      for (int32_t x = 0; x < ImageWidth; ++x) {
        fcomplex in_val;
        in_val = (min_real + x*real_factor) + (c_im*1.0iF);
       count[y][x] = mandel(in_val, max_iter);
      }
    }
    ........
}
```

ALGORITHM 5: An example of OpenMP* parallel for and SIMD combined usage.

graphics, and HPC applications by effectively exploiting the use of the Intel MIC architecture SIMD vector unit beyond traditional loop SIMD vectorization.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] Intel Corporation, "Intel Xeon Phi Coprocessor System Software Developers Guide," 2012, http://software.intel.com/en-us/mic-developer.

[2] N. Satish, C. Kim, J. Chhugani et al., "Can traditional programming bridge the Ninja performance gap for parallel computing applications?" in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*, pp. 440–451, June 2012.

[3] J. Reinders, "An Overview of Programming for Intel Xeon processor and Intel Xeon Phi Coprocessor," 2012.

[4] Intel Corporation, *Intel Advanced Vector Extensions Programming Reference*, Document Number 319433-011, Intel Corporation, 2011.

[5] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian, "Automatic intra-register vectorization for the intel architecture," *International Journal of Parallel Programming*, vol. 30, no. 2, pp. 65–98, 2002.

[6] A. J. C. Bik, D. L. Kreitzer, and X. Tian, "A case study on compiler optimizations for the Intel Core^{TM} 2 duo processor," *International Journal of Parallel Programming*, vol. 36, no. 6, pp. 571–591, 2008.

[7] X. Tian, H. Saito, M. Girkar et al., "Compiling C/C++ SIMD extensions for function and loop vectorizaion on multicore-SIMD processors," in *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops (IPDPSW '12)*, pp. 2349–2358, May 2012.

[8] H. J. Lu, M. Garkar, M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, "System V Application Binary Interface K1OM Architecture Processor Supplement," Version 1.0, 2012, http://software.intel.com/en-us/forums/topic/278102.

[9] S. J. Aarseth, *Gravitational N-Body Simulations: Tools and Algorithm*, Cambridge Monographs on Mathematical Physics, Cambridge University Press, Cambridge, UK, 2003.

[10] A. G. Gray and A. W. Moore, "'N-body' problems in statistical learning," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 521–527, 2000.

[11] M. Kachelrieb, M. Knaup, and O. Bockenbach, "Hyperfast perspective cone-beam backprojection," in *Proceedings of the IEEE Nuclear Science Symposium Conference Record*, pp. 1679–1683, November 2006.

[12] R. Allen and K. Kennedy, "Automatic translation of FORTRAN programs to vector form," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 4, pp. 491–542, 1987.

[13] A. E. Eichenberger, K. O'Brien, P. Wu et al., "Optimizing compiler for the CELL processor," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*, pp. 161–172, IEEE, St. Louis, Mo, USA, September 2005.

[14] R. Karrenberg and S. Hack, "Whole-function vectorization," in *Proceedings of the 9th International Annual IEEE/ACM Symposium on Code Generation and Optimization*, pp. 141–150, Charmonix, France, April 2011.

[15] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, pp. 145–156, June 2000.

[16] P. Wu, A. E. Eichenberger, and A. Wang, "Efficient SIMD code generation for runtime alignment and length conversion," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*, pp. 153–164, March 2005.

[17] Crescent Bay Software, *VAST-F/AltiVec: Automatic Fortran Vectorizer for PowerPC Vector Unit*, 2004.

[18] D. Nuzman and A. Zaks, "Outer-loop vectorization—revisited for short SIMD architectures," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pp. 2–11, Toronto, ON, Canada, October 2008.

[19] D. Nuzman and R. Henderson, "Multi-platform auto-vectorization," in *Proceedings of the 4th International Symposium on Code Generation and Optimization (CGO '06)*, pp. 281–294, New York, NY, USA, March 2006.

[20] G. Cheong and M. S. Lam, "An optimizer for multimedia instruction sets," in *Proceedings of the 2nd SUIF Compiler Workshop*, August 1997.

[21] J. Shin, M. Hall, and J. Chame, "Superword-level parallelism in the presence of control flow," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*, pp. 165–175, IEEE Computer Society, March 2005.

[22] M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell, "Extending OpenMP* with vector constructs for modern multicore SIMD architectures," in *OpenMP in a Heterogeneous World: 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11–13, 2012. Proceedings*, Lecture Notes in Computer Science, pp. 59–72, Springer, Berlin, Germany, 2012.

[23] OpenMP Architecture Review Board, "OpenMP Application Program Interface," Version 4.0 (Release Candidate RC1), 2012.

Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 576498, 16 pages
http://dx.doi.org/10.1155/2015/576498

Research Article

Optimized Data Transfers Based on the OpenCL Event Management Mechanism

Hiroyuki Takizawa,[1] Shoichi Hirasawa,[1] Makoto Sugawara,[2] Isaac Gelado,[3] Hiroaki Kobayashi,[2] and Wen-mei W. Hwu[4]

[1]*Tohoku University/JST CREST, Sendai, Miyagi 980-8579, Japan*
[2]*Tohoku University, Sendai, Miyagi 980-8578, Japan*
[3]*NVIDIA Research, Santa Clara, CA 95050, USA*
[4]*The University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA*

Correspondence should be addressed to Hiroyuki Takizawa; takizawa@cc.tohoku.ac.jp

Received 15 May 2014; Accepted 29 September 2014

Academic Editor: Sunita Chandrasekaran

In standard OpenCL programming, hosts are supposed to control their compute devices. Since compute devices are dedicated to kernel computation, only hosts can execute several kinds of data transfers such as internode communication and file access. These data transfers require one host to simultaneously play two or more roles due to the need for collaboration between the host and devices. The codes for such data transfers are likely to be system-specific, resulting in low portability. This paper proposes an OpenCL extension that incorporates such data transfers into the OpenCL event management mechanism. Unlike the current OpenCL standard, the main thread running on the host is not blocked to serialize dependent operations. Hence, an application can easily use the opportunities to overlap parallel activities of hosts and compute devices. In addition, the implementation details of data transfers are hidden behind the extension, and application programmers can use the optimized data transfers without any tricky programming techniques. The evaluation results show that the proposed extension can use the optimized data transfer implementation and thereby increase the sustained data transfer performance by about 18% for a real application accessing a big data file.

1. Introduction

Today, many high-performance computing (HPC) systems are equipped with graphics processing units (GPUs) serving as data-parallel accelerators in addition to conventional general-purpose processors (CPUs). For such a heterogeneous HPC system, application programmers need to manage the system heterogeneity while exploiting the parallelism involved in their applications. For the rest of the paper, we will follow the OpenCL terminology and refer to the CPUs as *hosts* and data-parallel accelerators as *compute devices*.

One difficulty in programming such a heterogeneous system is that a programmer has to take the responsibility for appointing the right processors to the right tasks. In the current OpenCL standard, only the host can perform some of tasks because the compute device is dedicated to kernel computation. For example, only the host can access files and communicate with other nodes. To write the computation results of a kernel into a file, the results have to be first transferred from the device memory to the host memory after the kernel execution, and then the host writes the results to the file.

From the viewpoint of programmers, accelerator programming models such as CUDA [1] and OpenCL [2] are used for data transfers between the device memory and the host memory, MPI [3] is used for internode data communication, and file functions of each programming language, such as `fprintf` and `fscanf` in the C programming, are used for the file I/O. Hence, these three categories of data transfers are described with different programming models. Some data transfers done by different programming models could be dependent; a certain data transfer can be done only after its preceding data transfer. *In order to enforce such dependence*, one popular way is to block the host thread until the preceding data transfer has finished. This kind of blocking often inhibits overlapping parallel activities of the host and the device and exposes the data transfer latencies to the total execution time. One may create a dedicated host thread for synchronizing the dependent data transfers. However, such multithreading will further increase the programming complexity. Consequently, the application performance strongly depends on the programming skills and craftsmanship of the developers.

Another difficulty is that there is no standard way to coding those data transfers even for common data transfer patterns. Since application programmers are supposed to appropriately combine those data transfers for fully exploiting the potential of a heterogeneous HPC system, the code is often specialized for a particular system. For example, one compute device may be capable of directly accessing a file, and another may not. In this case, the file access code for the former device would be totally different from that for the latter one. Therefore, the code for data transfers is likely to be system-specific and some abstractions are required to achieve functional portability as well as performance portability. Although OpenCL has been designed for programming various compute devices, it provides interfaces only for data transfers between the host memory and the device memory, but not for the other kinds of data transfers.

To overcome the above difficulties, we need a "bridging" programming model that provides a standard way for coding data transfers among various memory spaces and storages of a heterogeneous parallel system in a unified fashion. In this paper, we focus on OpenCL as the accelerator programming model for high code portability and propose an OpenCL extension for abstraction of data transfers, though the idea could be trivially extrapolated to other GPU programming models such as CUDA. The proposed OpenCL extension named *clDataTransfer* provides an illusion that the compute devices are transferring data directly to files or other nodes. This paper focuses especially on internode communication and file access as typical data transfers that need collaboration of hosts and devices. The extension offers some OpenCL commands and functions for the data transfers. The internode communication and file access commands are executed in the same manner as the other OpenCL commands, and hence the OpenCL programming model is naturally extended so as to seamlessly access file data and also to improve the MPI interoperability.

The clDataTransfer extension provides a portable, standardized way to programming of internode communications and file accesses from/to the device memory. Although MPI and file functions are used internally to perform those data transfers with help of the hosts, those internal behaviors are invisible to application programmers; it can thereby hide the system-aware optimized implementations behind function calls. Hence, we can also expect that the clDataTransfer extension improves the performance portability of OpenCL applications across different system types, scales, and generations.

The rest of this paper is organized as follows. Section 2 briefly reviews the related work. Section 3 discusses the difficulties in joint programming of OpenCL, MPI, and the standard I/O package of the C library, so-called *Stdio*. Then, Section 4 proposes clDataTransfer, which is an OpenCL extension for the collaboration with MPI and Stdio. Section 5 discusses the performance impact of clDataTransfer through some evaluation results. Finally, Section 6 gives concluding remarks and our future work.

2. Related Work

In the OpenCL programming model, a CPU works as a *host* that manages one or more *compute devices* such as GPUs. To manage the interaction between the host and devices, OpenCL provides various resources that are instantiated as OpenCL objects such as contexts, command queues, memory objects, and event objects. A unique *handle* is given to every object and is used to access the resource. A context is a container of various resources and is analogous to a CPU process. A command queue is used to interact with its corresponding compute device; a host enqueues a command to have its compute device execute a task. A memory object represents a memory chunk accessible from hosts and devices. An event object is bound with a command in the command queue to represent the status of the command and is used to block the execution of other commands. Hence, it is used to describe the dependency among commands. Moreover, multiple events can be combined to an event list to express several previous commands.

For example, `clEnqueueReadBuffer` is a typical OpenCL function for enqueuing a command, which transfers data from the device memory to the host memory. The function signature is as in Algorithm 1.

OpenCL command enqueuing functions take three arguments for event management: the number of events in the waiting list (`numevts`), the initial address of the waiting list (`wlist`), and the address to which the event object of the enqueued command is passed (`evtret`). The enqueued command is able to be executed when all the preceding commands associated with the event objects in the waiting list have been completed.

In joint programming of MPI and OpenCL, a programmer needs to consider not only host-device communication using OpenCL but also internode communication using MPI. So far, some researchers have presented several MPI extensions to GPUs to ease the joint programming of MPI and CUDA/OpenCL. We will refer to these approaches as *GPU-aware MPI implementations*. Lawlor has proposed cudaMPI [4] that provides an MPI-like interface for communication between remote GPUs. MPI-ACC [5] uses the `MPI_Datatype` argument to indicate that the memory buffer passed to an MPI function is located in the device memory. MVAPICH2-GPU [6] assumes Unified Virtual Addressing (UVA), which provides a single memory space for host and device memories, and checks if the memory buffer passed to an MPI function is in the device memory. Then, MVAPICH2-GPU internally uses different implementations depending on whether the memory buffer is in the device memory or the host memory. Stuart et al. have discussed

```
cl_int
clEnqueueReadBuffer( cl_command_queue cmd, /* command queue */
                     cl_mem buf,           /* memory buffer */
                     cl_bool blocking,     / blocking */
                     size_t offset,        /* offset */
                     size_t size,          /* buffer size */
                     void* hbuf,           /* buffer pointer */
                     cl_uint numevts,      /* the number of events in the list */
                     cl_event* wlist,      /* event list */
                     cl_evett* evtret )    /* event object of event object */
```

Algorithm 1

various design options of MPI extension to support accelerators [7]. Gelado et al. proposed GMAC that provides a single memory space shared by a CPU and a GPU and hence allows MPI functions to access device memory data [8]. Those extensions allow an application to use a GPU memory buffer as the end point of MPI communication; the extended MPI implementations enable using MPI functions for internode communication from/to GPU memory buffers by internally using data transfer functions of CUDA/OpenCL.

By using GPU-aware MPI extensions, application developers do not need to explicitly describe the host-device data transfers such as `clEnqueueWriteBuffer` and `clEnqueueReadBuffer`. As with clDataTransfer, these extensions do not require tricky programming techniques to achieve efficient data transfers, because they hide the optimized implementations behind the MPI function calls.

In GPU-aware MPI extensions, all internode communications are still managed by the host thread visible to application developers. For example, if the data obtained by executing a kernel are needed by other nodes, the host thread needs to wait for the kernel execution completion in order to serialize the kernel execution and the MPI communication; the host thread is blocked until the kernel execution is completed.

Furthermore, MPI extension to OpenCL is not straightforward, as Aji et al. discussed in [5]. To keep OpenCL data transfers transparent to MPI application programs, the MPI implementation must acquire valid command queues in some way. Aji et al. assume that an MPI process mostly uses only one command queue and its handle is thus cached by the MPI implementation to be used in subsequent communications, even though this assumption could be incorrect. Even if the cached command queue is available for subsequent communications, there may exist a more appropriate command queue for the communications. clDataTransfer allows application programmers to specify the best command queue for communication. It should be emphasized that GPU-aware MPI extensions and clDataTransfer are mutually beneficial rather than conflicting. For example, although this work has implemented pipelined data transfers using standard MPI functions, it is possible for clDataTransfer to use MPI extensions for its implementation.

Stuart and Owens have proposed DCGN [9]. As with clDataTransfer, DCGN provides an illusion that GPUs communicate without any help of their hosts. Unlike clDataTransfer, DCGN provides internode communication API functions that are called from GPU kernels. When the API is called by a kernel running on a GPU, the kernel sets regions of device memory that are monitored by a CPU thread. Then, the CPU thread reads necessary data from the device memory and thus handles the communication requests from the GPU. Accordingly, DCGN allows a kernel to initiate internode communication. However, the requirement for host to monitor the device memory incurs a nonnegligible runtime overhead. On the other hand, in clDataTransfer, internode communication requests are represented as OpenCL commands. Hence, the host initiates the commands and the clDataTransfer implementation can rely on the OpenCL event management mechanism to synchronize with the commands.

An OpenCL memory object in the same context is shared by multiple devices. The OpenCL memory consistency model implicitly ensures that the contents of a memory object visible to the devices are the same only at their synchronization points. Once a device updates a memory object shared by multiple devices, the new memory content is implicitly copied to the memory of every device in the same context. Some OpenCL implementations [10] support creating a context shared by multiple devices across different nodes and thereby attain data sharing among remote devices while conforming the OpenCL specifications. However, in this approach, multiple devices sharing one context can have only a single memory space; they cannot have different memory contents even if some of the contents are not needed by all nodes. As a result, the contents could unnecessarily be duplicated to the device memory of every node, increasing the aggregated memory usage and also internode communications for the duplication.

GPU computing is employed not only for conventional HPC applications but also for data-intensive applications, for example, [11, 12], in which the data sizes are large and hence are stored in files. As only hosts can access the data stored in files, GPU computing requires additional data transfers between hosts and GPUs. Nonetheless, GPUs are effective to accelerate the kernel execution and reduce the total execution time in practical data-intensive applications. Overlapping the kernel execution with various data transfers such as file accesses and host-device data transfers is a key technique to reduce the data transfer latencies and obviously has common code patterns. However, as far as we know, there is no standard way to develop this pattern in a manner that is reusable in other applications. As recent and future

```
cl_command_queue cmd;
cl_kernel kern;
cl_event evt;

for(int i(0);i<N;++i){
   // (1) computation on a device
   clEnqueueNDRangeKernel(cmd,kern,...,0,NULL,&evt);

   // (2) read the result from device to host
   clEnqueueReadBuffer(cmd,...,1,&evt,NULL);
   clFinish(cmd); // the host thread is blocked

   // (3) exchange data with other nodes
   MPI_Sendrecv(...); // blocking function call

   // (4) write the received data to device memory
   clEnqueueWriteBuffer(cmd,...);
}
```

Listing 1: A simple pseudocode combining OpenCL and MPI.

HPC systems have hierarchical storage subsystems, high-speed local storages using nonvolatile memories will be available. In those cases, the overlapping would become more significant because host-device data transfer overheads increase relatively to the file access overhead.

3. Difficulties in Joint Programming

This section discusses some difficulties in joint programming of OpenCL and other libraries, such as MPI, which are called by host threads. Listing 1 shows a simple code of the joint programming of MPI and OpenCL. In this code, a command to execute a kernel is first enqueued by invoking `clEnqueueNDRangeKernel`. Another command to read the kernel execution result is then enqueued by `clEnqueueReadBuffer`. Using the event object of the first command, `evt`, the execution of the second command is blocked until the first command is completed. The second command enqueued by `clEnqueueReadBuffer` can be either blocking or nonblocking. The function call is nonblocking if the third argument is `CL_FALSE`; otherwise it is blocking. If it is nonblocking, we have to use a synchronization function such as `clFinish` to make sure that the data have already been transferred from device memory to host memory in advance of calling `MPI_Sendrecv`. In this naive implementation, the data exchange with other nodes must be performed after the data transfer from device memory to host memory; those data transfers must be serialized. Similarly, `MPI_Sendrecv` and `clEnqueueWriteBuffer` must be serialized. Therefore, kernel execution and all data transfers are serialized, which results in a long communication time exposed to the total execution time. In addition, the host thread is blocked whenever MPI and OpenCL operations are serialized. Although Listing 1 shows an example of joint programming of MPI and OpenCL, the same difficulties arise when combining OpenCL and Stdio (or any other file access programming interfaces).

To make matters worse, there is no standard way for the joint programming. Even for simple point-to-point communication between two remote devices, we can consider at least the following three implementations. One is the naive implementation as shown in Listing 1. In the implementation, host memory buffers should be page-locked (pinned) for efficient data transfers (although the OpenCL standard does not provide any specific means to allocate pinned host memory buffers, most vendors rely on the usage of `clEnqueueMapBuffer` to provide programmers with pinned host memory buffers). This can be also a point to make different vendors require different implementations to exploit pinned memory. Another implementation is to map device memory objects to host memory addresses by using `clEnqueueMapBuffer` and then to invoke MPI functions to transfer data from/to the addresses. After the MPI communication, `clEnqueueUnmapMemObject` is invoked to unmap the device memory objects. The other implementation is to overlap host-device data transfers with internode data transfers. In this implementation, data of a device memory object are divided into data blocks of a fixed size, called a *pipeline block size*, and host-device data transfers of each block are overlapped with internode data transfers of other blocks in a pipelining fashion [6]. In this paper, the three aforementioned implementations are referred to as *pinned*, *mapped*, and *pipelined* data transfers. Among those implementations, the best one changes depending on several factors such as the message size, device types, device vendors, and device generations. Also in the cases of overlapping host-device data transfers with file accesses there are many implementation options and parameters due to the variety of file access speeds in a hierarchical storage subsystem. Accordingly, an application developer might need to implement multiple versions to optimize data transfers

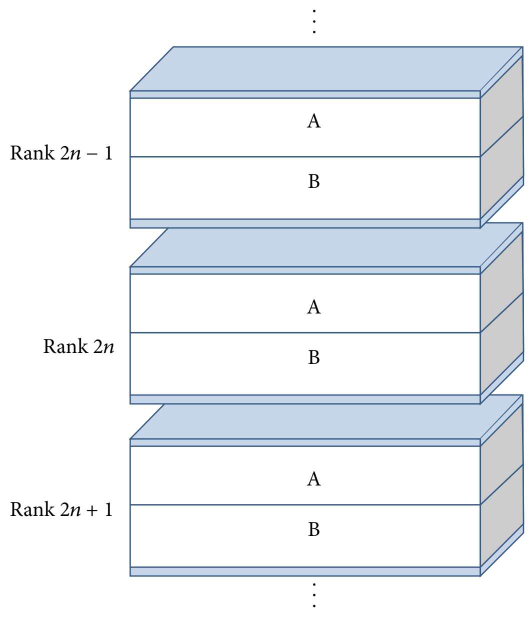


FIGURE 1: One-dimensional domain decomposition.

for performance portability of an application program across various systems.

Another common approach to hide the communication overhead is to overlap the data transfers and computation through double buffering [11, 13]. To this end, the computation is usually divided into two stages. While executing the first stage computation, the first stage data transfer is performed to prepare for the second stage computation. If the computation and data transfer are inside a loop, the second stage data transfer for the first stage computation of the next iteration is performed during the second stage computation of the current iteration.

In OpenCL programming, this overlapping optimization can be achieved using two in-order execution command queues. Listing 2 shows a simplified version of the Himeno benchmark code described in [13], which is originally written in CUDA and MPI. In the code, `jacobi_kernel_*` functions in Lines (9), (18), (28), and (35) invoke kernels using the command queue `cmd1` to update the memory object specified by the second argument. The code assumes one-dimensional domain decomposition, in which each decomposed domain is further halved into upper and lower portions, A and B. Figure 1 illustrates the domain decomposition assumed by the code. The top plane of A and the bottom plane of B are halo regions that have to be updated every iteration by exchanging data with neighboring nodes. Hence, if the MPI rank of a process is an even number, during calculating A, the process updates the halo region included in B. Then, it calculates B during exchanging data for updating the halo of A. On the other hand, if the MPI rank of a process is an odd number, the process first calculates B during updating the halo of A. Then, it calculates A during exchanging data for updating the halo of B. As a result, the communication time is not exposed to the total execution time as shown in Figure 2(a) unless the communication time exceeds the computation time.

As the number of MPI processes increases, the computation time becomes shorter because the domain processed by each GPU becomes smaller. However, the second stage communication cannot start even if the first stage computation is completed earlier and hence the data are ready for the second stage communication as shown in Figure 2(b). This is because the host thread is often blocked and tied up in the first stage communication in order to serialize the MPI and OpenCL operations.

Since the code in Listing 2 is simple, there are some workaround techniques to solve this problem. However, in the case where more advanced optimization techniques such as pipelining are applied to the data transfers, the host thread is stalled more frequently to timely synchronize MPI and OpenCL operations in multiple parallel activities of an application. In general, there are at least three parallel activities in an application: host computation, device computation, and nonblocking MPI communication. If there are dependent operations of MPI and OpenCL, the host thread is usually blocked to serialize the operations, which inhibits overlapping of the parallel activities. Also, host thread blocking is often used even in a serial application if the host thread needs to load data from a file, send them to the device memory, and retrieve the computation results from the device memory. Multithread programming or complex asynchronous I/O APIs would be required to properly manage those parallel activities. In this way, an application code becomes more complicated and system-specific, resulting in low code readability, maintainability, and portability. This motivates us to design a bridging programming model that can explicitly describe the dependencies among MPI, OpenCL, and file access operations in order to initiate data transfers without any help of the host thread.

4. An OpenCL Extension for Collaboration with MPI and Stdio

This paper proposes *clDataTransfer*, an OpenCL extension to facilitate and standardize the joint programming of MPI, Stdio, and OpenCL. The key idea of this extension is to use OpenCL commands for internode data transfers, file accesses, and data transfers between hosts and local devices.

The major advantages of clDataTransfer are summarized as follows.

(1) Performance portability: the implementation details of internode data transfers and file accesses are hidden behind extended commands and can be used via a simple programming interface similar to the standard OpenCL interface.

(2) Event management: a host thread is not responsible for serializing internode communications, file operations, and host-device communications. Instead, an event object is used to block the subsequent commands until the preceding command is completed.

```
cl_command_queue cmd1, cmd2;
cl_mem p_new, p_old, p_tmp;

for(int i(0);i<N;++i){
  //swap pointers
  p_tmp = p_new; p_new = p_old; p_old = p_tmp;
  if( rank%2 == 0)  {
    // the upper portion is calculated
    jacobi_kernel_even_A(cmd1,p_new,...);
    // the bottom plane is updated
    MPI_Irecv(...);
    clEnqueueReadBuffer(cmd2,p_old,CL_FALSE,...);
    clFinish(cmd2);  //  blocking
    MPI_Send(...);     //  blocking
    MPI_Wait(...);     //  blocking
    clEnqueueWriteBuffer(cmd2,p_old,CL_FALSE,...);
    // the lower portion is calculated
    jacobi_kernel_even_B(cmd2,p_new,...);
    // the top plane is updated
    MPI_Irecv(...);
    clEnqueueReadBuffer(cmd1,p_new,CL_FALSE,...);
    clFinish(cmd1);  //  blocking
    MPI_Send(...);     //  blocking
    MPI_Wait(...);     //  blocking
    clEnqueueWriteBuffer(cmd1,p_new,CL_FALSE,...);
  }
  else {
    jacobi_kernel_odd_B(cmd1,p_new,...);
    MPI_Irecv(...);
    clEnqueueReadBuffer(cmd2,p_old,CL_FALSE,...);
    clFinish(cmd2);  //  blocking
    MPI_Send(...);     //  blocking
    MPI_Wait(...);     //  blocking
    clEnqueueWriteBuffer(cmd2,p_old,CL_FALSE,...);
    jacobi_kernel_odd_A(cmd2,p_new,...);
    MPI_Irecv(...);
    clEnqueueReadBuffer(cmd1,p_new,CL_FALSE,...);
    clFinish(cmd1);  //  blocking
    MPI_Send(...);     //  blocking MPI_Wait (...);  //  blocking
    clEnqueueWriteBuffer(cmd1,p_new,CL_FALSE,...);
  } clFinish(cmd1);clFinish(cmd2);   /* error calculation */
}
```

Listing 2: A Himeno benchmark code with overlapping communication and computation.

(3) Collaboration for latency hiding: clDataTransfer can collaborate with MPI and Stdio in order to hide data transfer latencies in a pipelining fashion.

By encapsulating file accesses into OpenCL commands, the clDataTransfer extension offers two file access commands: `clEnqueueReadBufferToStdioFile` and `clEnqueueWriteBufferFromStdioFile`. `clEnqueueReadBufferToStdioFile` reads data from a device memory buffer and writes the data to a file, and `clEnqueueWriteBufferFromStdioFile` reads data from a file and writes the data to a device memory buffer. The function signatures are as in Algorithm 2.

Similarly, the clDataTransfer extension offers `clEnqueueSendBuffer` and `clEnqueueRecvBuffer`, which enqueue commands of transferring data from and to a device memory buffer, respectively. These clDataTransfer functions are direct counterparts of `MPI_Send` and `MPI_Recv` [3] and hence take the same arguments of rank, tag, and communicator as those two MPI functions. For example, the function signature of `clEnqueueRecvBuffer` is as in Algorithm 3.

When one MPI process invokes those functions for sending a command to a device, the device becomes a *communicator device* for one MPI communication and works as if it communicates instead of the host thread. The data sent to the MPI rank are received by the communicator device,

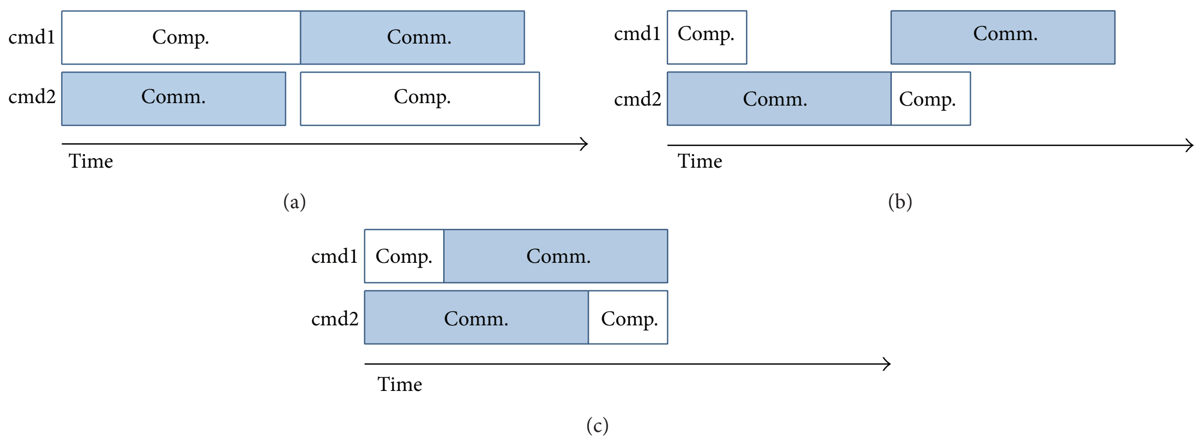


Figure 2: Overlapping communications and computations. (a) The communication time is overlapped with the computation time. (b) The computation time is too short to hide the communication time. Since joint programming of OpenCL and MPI cannot express the dependency between the first communication and the second computation, the host thread is blocked to execute them in a correct order. (c) The second communication can potentially start earlier because the host thread is not blocked.

```
cl_int clEnqueueReadBufferToStdioFile(
                                 cl_command_queue cmd, /* command queue */
                                 cl_mem mem,   /* memory buffer to be read */
                                 cl_bool blk,  /* blocking function call      */
                                 size_t off,   /* offset */
                                 size_t bsz,   /* buffer size */
                                 FILE* fp,     /* file pointer */
                                 cl_uint nev,  /* the number of events in the list */
                                 const cl_event* evl, /* event list */
                                 cl_event* evt) /* event object of the function call */
cl_int clEnqueueWriteBufferFromStdioFile(
                                 cl_command_queue cmd, /* command queue */
                                 cl_mem mem,   /* memory buffer to be written */
                                 cl_bool blk,  /* blocking function call      */
                                 size_t off,   /* offset */
                                 size_t bsz,   /* buffer size */
                                 FILE* fp,     /* file pointer */
                                 cl_uint nev,  /* the number of events in the list */
                                 const cl_event* evl, /* event list */
                                 cl_event* evt) /* event object of the function call */
```

Algorithm 2

```
cl_int
clEnqueueRecvBuffer(cl_command_queue cmd, /* command queue */
                      cl_mem buf, /* memory buffer to receive data */
                      cl_bool blocking, /* blocking function call */
                      size_t offset,  /* offset */
                      size_t size,    /* buffer size */
                      int src,        /* sender’s rank */
                      int tag,        /* tag */
                      MPI_Comm comm,  /* communicator */
                      cl_uint numevts, /* the number of events in the list */
                      const cl_event* wlist, /* event list */
                      cl_event* evtret ) /* event object of the function call */
```

Algorithm 3

```
if( rank == 0 ){
 clEnqueueSendBuffer(cmd, buf, CL_TRUE, off, sz, 1,...);
}
else if(rank == 1){
 clEnqueueRecvBuffer(cmd, buf, CL_TRUE, off, sz, 0,...);
}
```

Listing 3: A code with the OpenCL extension for device-to-device communication.

and the received data are stored in the memory space of the communicator device, that is, `buf`. The MPI rank of the sender is given to the function, and the sender could be either the host thread or the communicator device associated with the MPI rank.

In the case where both the sender and the receiver submit internode communication commands to their devices, those devices communicate with each other. Listing 3 shows a simple example of communication between remote devices. In this code, the communicator device of rank 0 sends the data of a memory buffer object to the communicator device of rank 1 without explicitly calling any MPI functions. Accordingly, devices appear to communicate with remote devices without help of their host threads. The implementation details of internode communication by combining MPI and OpenCL are hidden behind the OpenCL command execution. Hence, the application can use optimized implementations of efficient data transfers without using tricky programming techniques. If one MPI process needs to use multiple communicator devices, a unique tag is given to each MPI communication to specify which communicator device handles it.

4.1. Event Management. The clDataTransfer extension allows a programmer to use event objects in order to express the dependency among internode communication commands, storage file access commands, and other OpenCL commands. If a data transfer command provided by clDataTransfer needs the result of its preceding command, the programmer can get the event object of the preceding command and use it to block the execution of the data transfer command. This ensures that the data transfer is performed after the preceding command is completed. In this way, data transfer commands of clDataTransfer are incorporated into the OpenCL execution model in a natural manner. Accordingly, function calls of MPI and Stdio are encapsulated in OpenCL commands whose dependencies with other OpenCL commands are accurately enforced by the command queues. Unlike the conventional joint programming of MPI, Stdio, and OpenCL, the host thread does not need to wait for the preceding command completion. After enqueuing the commands by nonblocking function calls, the host thread immediately becomes available for other computations and data transfers; an application programmer can consider as if a device is able to work independently from the host thread. In due time, the OpenCL runtime will release the clDataTransfer command for timely execution of the MPI functions as shown in Figure 2(c), even though the two communications may or may not be performed concurrently.

Using the clDataTransfer extension, the code in Listing 2 can be simply rewritten as the code in Listing 4. This is an example that demonstrates simplification of common patterns in joint programming of OpenCL and other programming models. In this particular case, the clDataTransfer extension can halve the number of code lines for describing the same computation as the joint programming of OpenCL and MPI. Since there are dependencies among the enqueued commands, they are expressed by using event objects bound with the commands. In Listing 2, the second stage computations, `jacobi_even_A` and `jacobi_odd_B`, are blocked using event objects of the first communication, `e[1]`. The second stage communications are blocked using the event object of the first stage computation, `e[0]`. On the other hand, in Listing 4, the dependencies among the function calls are managed by the OpenCL event management mechanism, and the host thread is thus freed from controlling the computation and communication. In the code, `clEnqueueSendrecvBuffer` enqueues an OpenCL command for exchanging data between two MPI processes by internally invoking `MPI_Sendrecv` under control of the OpenCL event management. Therefore, the host thread is just waiting at the end of the iteration by calling `clFinish`.

4.2. Interoperability with Existing MPI Functions. In clDataTransfer, an MPI process uses clDataTransfer commands for transferring data from/to a device memory buffer. If an MPI process needs to transfer data from/to a host memory buffer, clDataTransfer allows the MPI process to use standard MPI functions such as `MPI_Isend` and `MPI_Irecv` to communicate with remote devices as well as remote hosts. Listing 5 shows that the MPI process of rank 0 receives data from a remote device managed by the MPI process of rank 1. A special `MPI_Datatype` value, `MPI_CL_MEM`, is given to the third argument of `MPI_Irecv` in order to express that the sender is supposed to be a communicator device and the data are in the device memory. If `MPI_CL_MEM` is given, the sender and receiver collaborate for efficient data transfers between host and device memories. A similar approach of using `MPI_Datatype` can be seen in [5], even though they extend only MPI but not OpenCL.

As shown in Listing 5, nonblocking MPI functions can be used for internode communication from/to a host memory buffer. Hence, the data need to be received

```
cl_command_queue cmd1, cmd2;
cl_mem p_new, p_old, p_tmp;
cl_event e[2];

for(int i(0);i<N;++i){
  p_tmp = p_new; p_new = p_old; p_old = p_tmp;
  if( rank%2 == 0) {
    jacobi_kernel_even_A(cmd1,p_new...0,NULL,&e[0]);
    clEnqueueSendrecvBuffer(cmd2,p_old,...0,NULL,&e[1]);
    jacobi_kernel_even_B(cmd2,p_new...1,&e[1],NULL);
    clEnqueueSendrecvBuffer(cmd1,p_new,...1,&e[0],NULL);
  }
  else {
    jacobi_kernel_odd_B(cmd2,p_new...0,NULL,&e[0]);
    clEnqueueSendrecvBuffer(cmd1,p_old,...0,NULL,&e[1]);
    jacobi_kernel_odd_A(cmd1,p_new...1,&e[1],NULL);
    clEnqueueSendrecvBuffer(cmd2,p_new,...1,&e[0],NULL);
  }
  clFinish(cmd1);clFinish(cmd2);
  /* error calculation */
}
```

Listing 4: A Himeno benchmark code with the proposed OpenCL extension.

```
(1) cl_context ctx;
(2) MPI_Request req;
(3) cl_event evt[2];
(4)
(5) if( rank == 0 ){
(6)   /* receiving data from a remote device */
(7)   MPI_Irecv(recvbuf, bufsz, MPI_CL_MEM, 1, 0, MPI_COMM_WORLD,&req);
(8)   /* creating an event object of MPI_Irecv */
(9)   evt[0] = clCreateEventFromMPIRequest(ctx,&req,NULL);
(10)  /* executing a kernel during the data transfer */
(11)  clEnqueueNDRangeKernel(..., &evt[1]);
(12)
(13)  /* executing this after the computation and communication */
(14)  clEnqueueWriteBuffer(cmd, buf, ..., 2, evt, NULL);
(15) }
(16) else if(rank == 1){
(17)  /* send data to a remote host */
(18)  clEnqueueSendBuffer(cmd, buf, CL_TRUE, 0, bufsz, 0,...);
(19) }
```

Listing 5: A code with the OpenCL extension for host-to-device communication.

before `clEnqueueWriteBuffer` in lines (14) is executed to write the data to the device memory of rank 0. In addition, a kernel in line (11) is executed during the internode communication. To express the dependency among nonblocking MPI function calls and OpenCL commands, the clDataTransfer extension offers a function to create an OpenCL event object that corresponds to `MPI_Request` of a nonblocking MPI function call. Using the event object, another OpenCL command can be executed after the nonblocking MPI function is completed; the dependence between an MPI operation and an OpenCL operation is properly enforced without host intervention. In Listing 5, the event object is used to ensure that `MPI_Irecv` is completed before writing data to a device memory buffer.

The MPI interoperability is very important because many applications have already been developed in such a way that CPUs manage all internode communications via MPI function calls. Considering the importance, the clDataTransfer extension is not designed as a standalone communication library but an OpenCL extension for interoperation with

Table 1: System specifications.

System	Masamune	Cichlid	RICC
CPU	Intel Xeon E5-2670	Intel Core i7 930	Intel Xeon 5570
GPU	GeForce GTX TITAN	Tesla C2070	Tesla C1060
NIC	GbE 1000BASE-T	GbE 1000BASE-T	InfiniBand DDR
OS	CentOS 6.4	CentOS 6.0	RHEL 5.3
Compiler	GCC-4.4.7	GCC-4.4.4	Intel Compiler 11.1
GPU Driver	319.37	290.10	295.41
OpenCL	OpenCL1.1 (CUDA5.5)	OpenCL1.1 (CUDA4.1.1)	OpenCL1.1 (CUDA 4.2.9)
MPI	Open MPI 1.5.4	Open MPI 1.6.0	Open MPI 1.6.1
Storage	SSD (Intel 910 400 GB)	NFS	NFS

MPI. With the interoperability, legacy applications can be ported incrementally to heterogeneous computing systems by gradually replacing the MPI function calls with the clDataTransfer extension. This does not mean that all internode communications should be replaced with the clDataTransfer extension. We argue that both MPI and OpenCL need to be extended for their efficient interoperation.

Although the clDataTransfer extension offers internode peer-to-peer communications among remote hosts and devices, it does not currently offer any collective communications. This is because the function calls of MPI collective communications are blocking and no OpenCL extension is required to describe the dependability among the collective communications and OpenCL commands. If optimized collective communications for device memory objects are required, we can hide the implementation details in MPI collective communication functions, rather than developing a set of special collective communication functions for device memory objects. As the MPI-3.0 standard will support nonblocking collective communications, some synchronization mechanisms between the nonblocking collective communications and OpenCL commands might be required in the future. In this case, it will be effective to further extend OpenCL to use its event management mechanism for the synchronization.

5. Evaluation and Discussions

In this section, the performance impact of the proposed extension is discussed by showing the effects of hiding the host-device data transfer latency and the performance improvement. In this work, a GPU program of the Smith Waterman algorithm [11] is first used to evaluate the performance gain by overlapping host-device data transfers with file accesses. Then, the Himeno benchmark [13] and the nanopowder growth simulation [14] are adopted for the evaluation of MPI interoperability, which is improved by the proposed extension.

Three systems called Masamune, Cichlid, and RICC are used for the following evaluation. Masamune is a single node PC with Intel Xeon E5-2670 CPU running at 2.60 GHz and one NVIDIA GeForce GTX TITAN GPU. Cichlid is a small PC cluster system of four nodes, each of which contains one Intel Core i7 930 CPU running at 2.8 GHz and one NVIDIA Tesla C2070 GPU. The nodes are connected via the Gigabit Ethernet network. On the other hand, in the multipurpose PC cluster of RIKEN Integrated Cluster of Clusters (RICC), 100 compute nodes are connected via an InfiniBand DDR network. Each of the compute nodes has two Intel Xeon 5570 CPUs and one NVIDIA Tesla C1060 GPU. The system specifications are summarized in Table 1.

5.1. Implementation. In this work, we have implemented the clDataTransfer extension on top of NVIDIA's OpenCL and Open MPI [15] as shown in Table 1. As most of currently available OpenCL implementations are proprietary, the clDataTransfer extension is designed so that it can be implemented on top of a proprietary OpenCL implementation. In the implementation, we have to consider at least three points. One point is how to implement clDataTransfer commands that mimic standard OpenCL commands. Another is how to implement nonblocking function calls. The other is how to implement pipelined data transfers.

To implement clDataTransfer commands whose execution is managed by the OpenCL event management system, user event objects are internally used to create event objects of those additional commands provided by the clDataTransfer extension. Since there are several different behaviors between standard event objects and user event objects, the runtime of the clDataTransfer extension has been developed so that user event objects of additional commands can mimic event objects of standard OpenCL commands. A simplified pseudocode of a clDataTransfer function is shown in Listing 6. When the function is executed, from the viewpoint of application programmers, the clDataTransfer runtime appears to work as follows. A user event object whose execution status is `CL_SUBMITTED` is first created when a clDataTransfer command is enqueued. Then, the clDataTransfer runtime automatically changes the execution status to `CL_COMPLETE` when the command is completed. This allows other commands to wait for the completion of a clDataTransfer command by using its user event object. Therefore, application programmers can use the event object of a clDataTransfer command in the same way as that of a standard OpenCL command.

The clDataTransfer function in Listing 6 can be invoked in either blocking or nonblocking mode. To invoke a clDataTransfer function without blocking the host thread, the clDataTransfer runtime internally spawns another thread dedicated to data transfers. Since most existing OpenCL

```
cl_int clDataTransferFunc( ...,
                               cl_uint numevts,        /* the number of events in the list */
                               cl_event* wlist,        /* event list */
                               cl_evett* evtret )      /* event object of event object */
{
   /* create a new user event object whose status is CL_SUBMITTED */
   *evtret = clCreateUserEvent(...);

   if( non_blocking = CL_TRUE)
     pthread_create(...,cldtThreadFunc,...);
   else
     cldtThreadFunc(...);

   return CL_SUCCESS;
}

/* numevt, wlist, and evtret are passed from the caller */
void* cldtThreadFunc(void* p)
{
   clWaitForEvent(numevt, wlist);

   /* pipelined data transfer */

   clSetUserEventStatus(*evtret, CL_COMPLETE);
   return NULL;
}
```

Listing 6: A simple pseudocode of a clDataTransfer function.

implementations are already spawning a CPU thread to support callbacks, the same thread can technically be used to handle the clDataTransfer function calls. Thus, no additional thread would be needed if clDataTransfer is implemented by OpenCL vendors.

As the clDataTransfer implementation needs to call MPI and file access functions from the host thread and the dedicated thread, their underlying implementations are assumed to be thread-safe. File access functions are generally thread-safe. On the other hand, in MPI, `MPI_Init_thread` should work with `MPI_THREAD_MULTIPLE`. To make Open MPI work correctly for InfiniBand in a multithreaded environment, IP over InfiniBand (IPoIB) is used for performance evaluation on RICC.

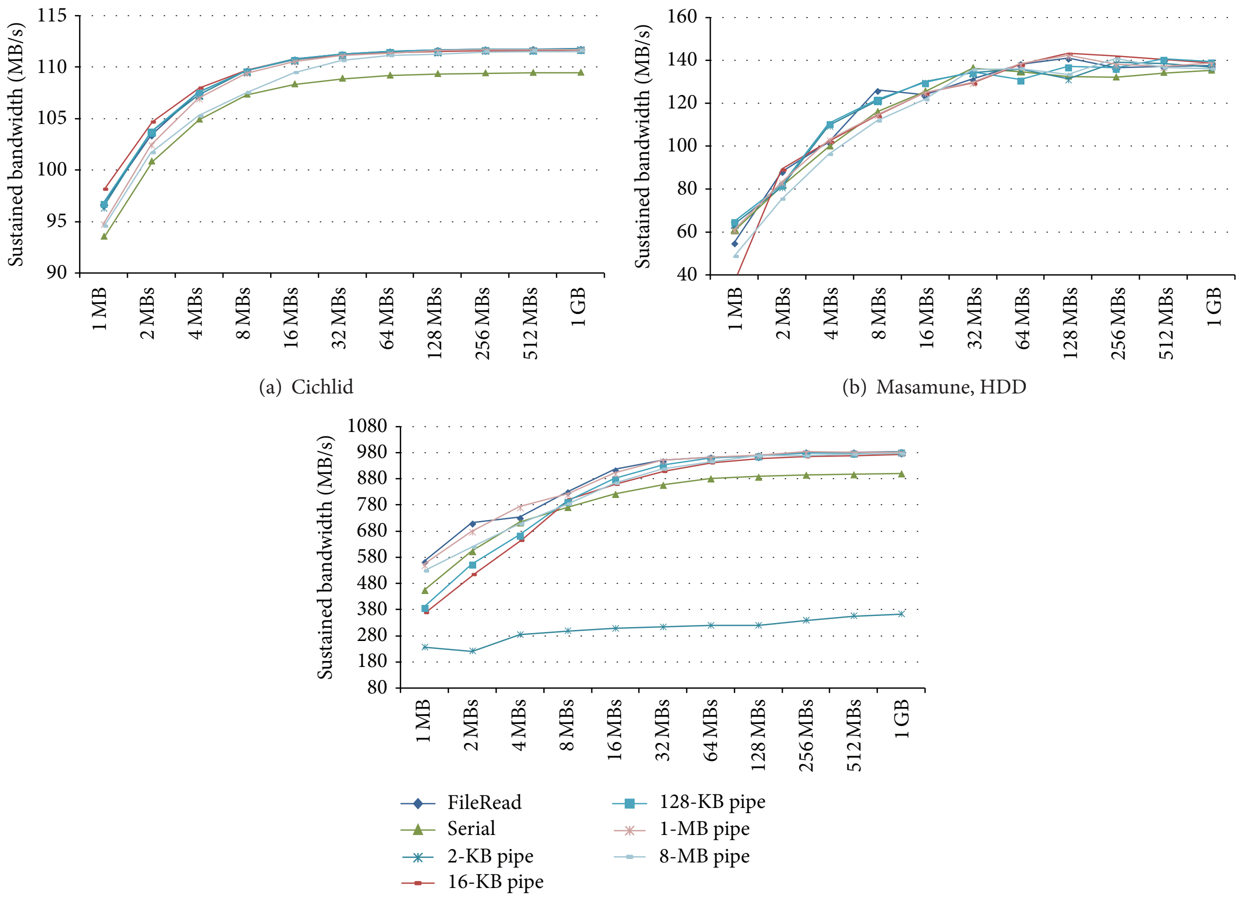
In our current implementation, pipelined data transfers are implemented by ourselves by reference to some papers on GPU-aware MPI implementations [5, 6] and encapsulated in clDataTransfer commands as shown in Listing 6. So far, wrapper functions of file I/O functions and some major MPI functions such as `MPI_Send` and `MPI_Recv` have been developed so that those functions can perform pipelined data transfers of overlapping host-device communication with internode communication when `MPI_CL_MEM` is given as the `MPI_Datatype` parameter.

5.2. Evaluation of File Access Performance

5.2.1. Evaluation of Sustained Data Transfer Bandwidths. The sustained bandwidths of data transfers from files to device memory buffers are evaluated to show that `clEnqueueWriteBufferFromStdioFile` can reduce the data transfer time compared to conventional serialized data transfers. To evaluate the sustained bandwidths with different storage's bandwidths, the solid state drive (SSD) and the hard disk drive (HDD) of Masamune are used as the local storages, and a shared file system of NFS is used as the global storage and accessed from Cichlid.

First, we evaluate how much the clDataTransfer extension can improve the sustained bandwidth. In the case of using `clEnqueueWriteBufferFromStdioFile`, data are read from a file and then sent to a device memory buffer. The bandwidth of a storage is lower than that of the data transfer between the host and the device via the PCI-express bus. Hence, the sustained bandwidth of the data transfer is limited by the storage bandwidth. Since `clEnqueueWrite-BufferFromSdtioFile` enables the host-device data transfer to be overlapped with the file read, it can reduce the data transfer time and hence achieve a higher sustained bandwidth than the sequential execution of those two data transfers.

Figure 3 shows the sustained bandwidths obtained with changing the data size and the pipeline buffer size. The vertical axis shows the sustained bandwidth, and the horizontal axis is the data size. In the figure, *Serial* means the data transfer time in the case of not hiding the host-device data transfer latency and N-pipe means the data transfer time of the pipelined implementation with an N-byte pipeline buffer. By hiding the latency more, the data transfer time approaches to the file read time, which is *FileRead* in the figure. These



(a) Cichlid

(b) Masamune, HDD

(c) Masamune, SSD

FIGURE 3: Sustained bandwidth (MB/s) of `clEnqueueReadBufferToStdioFile`.

results indicate that the clDataTransfer extension can hide the host-device data transfer latency and hence the sustained performance of the data transfer from a file to a device memory buffer is almost comparable to the sustained bandwidth of just reading a file, that is, FileRead. A programmer can use the optimized data transfer implementation by just enqueuing a clDataTransfer command.

In the case of reading from the HDD of Masamune, the file read time varies widely as shown in Figure 3. This is likely due to the bandwidth of the disk and the behaviors of the read-ahead thread in the OS kernel. As a result, the performance gain is unseen. The FileRead performance is sometimes even lower than that of `clEnqueueWriteBufferFromStdioFile` because of the intrinsic measurement accuracy.

5.2.2. Evaluation with the Smith Waterman Algorithm. In this work, a CUDA program of the Smith Waterman algorithm [11] is ported to OpenCL. Then, the performance of the OpenCL version is evaluated to show that clDataTransfer can hide the host-device data transfer latency of a real application by overlapping it with the file access latency. In the Smith Waterman program, the data transfer time can be overlapped with the computation time. However, the data transfer time is still partially exposed to the total execution time if the computation time is shorter than the data transfer time. The exposed data transfer time depends on the problem size. Therefore, in this evaluation, the overlap of computation and data transfer is disabled, and the fully exposed data transfer time is evaluated to clearly show the effect of overlapping the host-device data transfer latency with the file access latency.

The OpenCL program repeatedly reads the data in files to host memory buffers and sends them to device memory buffers. Suppose that `d_db` and `h_db` are handles of a device memory buffer and a host memory buffer, respectively. Their buffer size is `readsz`, and the file pointer is `fp`. Then, the original code has the following code pattern:

```
fread(h_db, readsz, 1, fp);
clEnqueueWriteBuffer(cmd,
d_db, CL_TRUE, 0, readsz, h_db, 0, NULL, NULL);
```

The above pattern is replaced with an additional OpenCL command enqueued by

```
clEnqueueWriteBufferFromStdioFile
(cmd, d_db, CL_TRUE, 0, readsz, fp, 0,
NULL, NULL);
```

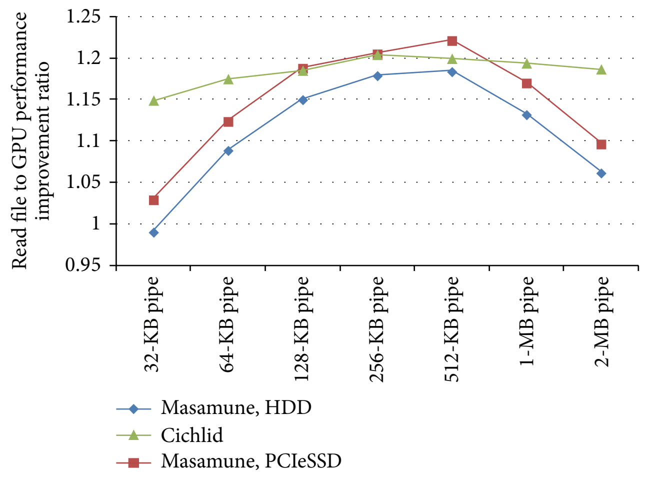


Figure 4: The improvement ratio of data transfer performance for the Smith Waterman algorithm.

The results of evaluating the data transfer time with changing the pipeline buffer size are shown in Figure 4. Here, the data transfer time is the total time of data transfers from a database file to a device memory buffer. These results indicate that the clDataTransfer extension can reduce the data transfer time if the pipeline buffer size is appropriately configured. The performance improvement of the clDataTransfer extension decreases if the pipeline buffer size is too small due to the runtime overhead of the pipeline implementation. It also decreases if the pipeline buffer size is too large compared to the data size, because pipelining with a too large buffer does not benefit from overlapping of data transfers. Accordingly, the optimal pipeline buffer size depends not only on the storage performance but also on the data size to be transferred from a file to a device memory buffer. The pipeline buffer size has to be dynamically adjusted because the data size is usually determined at runtime. Figure 4 discusses the effect of changing the pipeline buffer size on performance. Since the clDataTransfer extension hides the implementation details of data transfers, it is technically possible to employ empirical parameter tuning or autotuning for automatically finding the optimal pipeline buffer size, as in MVAPICH2-GPU's CUDA support.

In the Smith Waterman program, the data size to be read from a file ranges from 511 bytes to 4 Mbytes and hence is relatively small. The sustained bandwidths of both the file read and the host-device data transfer become lower for the transfer of a small data chunk. If the program is used for large input data, we believe that the performance improvement by clDataTransfer would become more remarkable as indicated in Figure 3.

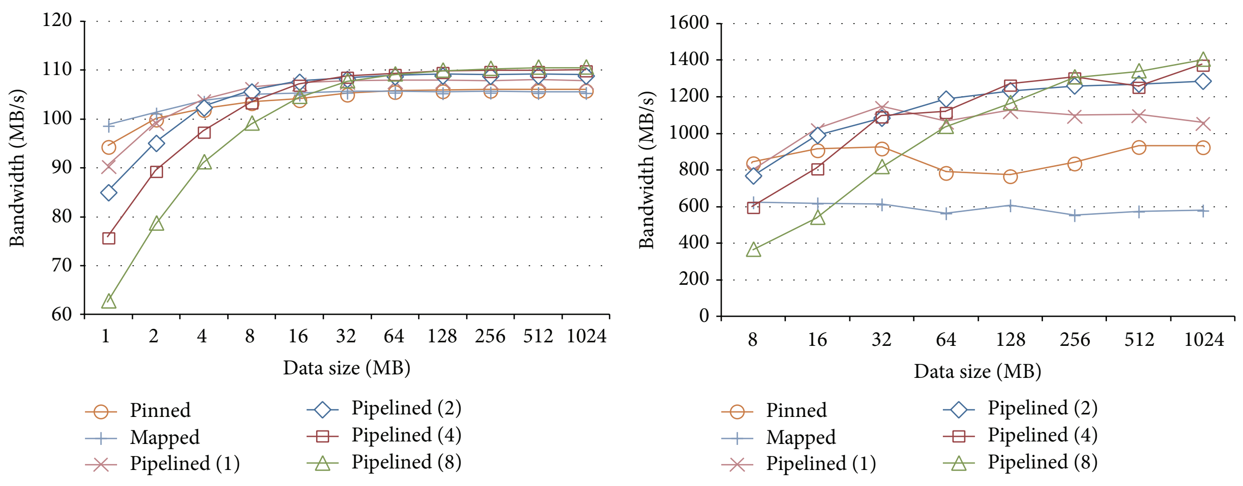
5.3. Evaluation of Internode Communication Performance

5.3.1. Point-to-Point Communication Performance. One advantage of the clDataTransfer extension over conventional joint programming of MPI and OpenCL is that the clDataTransfer extension can hide the implementation details of system-aware optimization for efficient data transfers.

Figure 5 shows the difference in sustained bandwidth among pinned, mapped, and pipelined implementations described in Section 3. In the figure, "pipelined(N)" indicates the results of pipelined data transfers with the pipeline buffer size of N Mbytes. The evaluation results in Figure 5(a) show that the performance difference among the three implementations is small in the Cichlid system. This is because their sustained bandwidths are limited by the bandwidth of the GbE interconnect network. The time for host-device communication is much shorter than that of internode communication, and hence the pipelined implementation hardly improves the sustained bandwidth. On the other hand, in Figure 5(b), there is a big difference in sustained bandwidth among the three implementations. Moreover, the sustained bandwidth of the pipelined implementation changes with the pipeline buffer size. Pipelining with a relatively small pipeline buffer is the most efficient when the message size is small because the pipeline buffer size needs to be smaller than the message size. On the other hand, a large pipeline buffer leads to a higher sustained bandwidth for large messages because the sustained bandwidth of sending each pipeline buffer usually increases with the pipeline buffer size. Accordingly, the optimal pipeline buffer size changes depending at least on the message size.

From the above results, it is obvious that system-aware optimizations are often required by multinode GPU applications to achieve a high performance, and hence some abstractions of internode data transfers are necessary for high performance-portability. For example, on RICC, the pinned data transfer is always faster than the mapped one, while the mapped data transfer is faster for small messages on Cichlid due to the short latency of the implementation. The clDataTransfer extension provides interfaces that abstract internode data transfers and thereby allows an application programmer to use optimized data transfers without tricky programming techniques. An automatic selection mechanism of the data transfer implementations can be adopted behind the interfaces. The current implementation of the clDataTransfer runtime can use either the pinned or the mapped data transfer for small messages, and the pipelined data transfer can be performed for large messages. The pipelined data transfer can also be implemented using either the pinned or the mapped data transfer. In the following evaluation, the mapped and pinned data transfers are used for Cichlid and RICC, respectively. Of course, other optimized data transfers can be incorporated into the runtime and available to application programs without changing their codes, which results in high performance-portability across system types, scales, and probably generations.

5.3.2. Evaluation with the Himeno Benchmark. The performance impact of using the clDataTransfer extension is first evaluated by comparing the sustained performances of three implementations for the Himeno benchmark. One implementation is called the hand-optimized implementation presented in [13]. The hand-optimized implementation uses pinned data transfers for exchanging halo data of about 750 Kbytes. Another is called the serial implementation that is almost the same as the hand-optimized implementation



(a) Cichlid (b) RICC

Figure 5: Sustained bandwidth of peer-to-peer communication.

but all the computations and communications are serialized. The performance of the serial implementation is supposed to be the lowest. The other is the implementation using the clDataTransfer extension, called the clDataTransfer implementation.

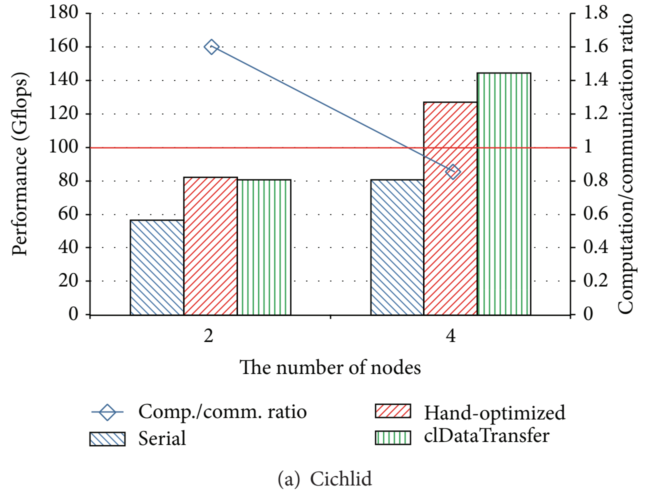
Figure 6 shows the sustained performances of the three implementations for the Himeno benchmark with M-size data. Since the hand-optimized implementation is well designed for overlapping the computations and communications, it can always achieve a higher performance than the serial implementation; the average speedup ratios are 51.2% and 15.2% for Cichlid and RICC, respectively. The performance of the clDataTransfer implementation is almost always comparable to that of the hand-optimized implementation because the communication times of both the hand-optimized and the clDataTransfer implementations are not exposed to their total execution times. Accordingly, the clDataTransfer extension allows an application programmer to easily overlap the communication and computation by simply sending internode communication commands to devices and utilizing OpenCL event objects to enforce the dependencies among OpenCL commands.

The results in Figure 6(a) are obtained using Cichlid whose network performance is low compared to the computation performance. The ratio of the computation time to the communication time in the serial implementation is also shown in the figure. Only in the case of Cichlid with four nodes, the ratio of the computation to the communication is less than one, and hence the communication time cannot completely be overlapped with the computation time when pinned data transfers are used for communication. In this case, the performance of the hand-optimized implementation is clearly lower than the clDataTransfer implementation. The main reason of the performance difference is that the mapped data transfer behind the clDataTransfer implementation is faster than the pinned data transfers. These results clearly show the importance of system-dependent optimizations for highly efficient data transfers. As the programming model of the clDataTransfer extension encapsulates the data transfers, an application programmer does not need to know the implementation details and can automatically use the optimized implementation from a simply written code such as shown in Listing 4.

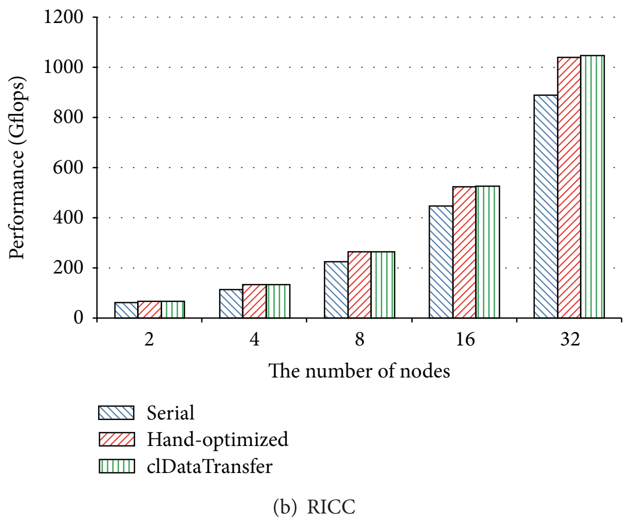
5.3.3. Evaluation with a Practical Application. The performance impact of the clDataTransfer extension is further discussed by taking the nanopowder growth simulation [14] as an example of real applications. The simulation code has been developed for numerical analysis of the entire growth process of binary alloy nanopowders in thermal plasma synthesis. Although various phenomena are considered to simulate the nanopowder growth process, about 90% of the total execution time of the original code is spent for simulating the process of coagulation among nanoparticles.

In the following evaluation, the clDataTransfer extension is applied to a parallel version of the simulation code, in which only the coagulation routine is parallelized using MPI, and its kernel loop is further accelerated using OpenCL. The other phenomena such as nucleation and condensation are computed by one host thread, and the coefficient data of about 42 Mbytes required by the coagulation routine are distributed from the host thread to each node at every simulation step. For the simulation code, two versions have been implemented to clarify the effect of using the optimized data transfers provided by the clDataTransfer extension. One is the baseline implementation that just uses `MPI_Isend` and `MPI_Recv` for coefficient data distribution. The other is the clDataTransfer implementation, which uses `MPI_Isend` with `MPI_CL_MEM` to send the coefficients in host memory buffers and `clEnqueueRecvBuffer` to receive them.

Figure 7 shows the results to compare the performances of the two implementations on RICC. Unlike the Himeno benchmark, the communication overheads are obviously



(a) Cichlid



(b) RICC

FIGURE 6: The performance for the Himeno benchmark.

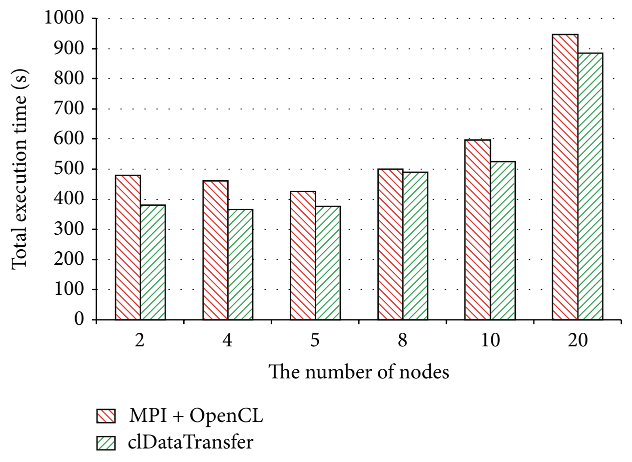


FIGURE 7: The execution time of the nanopowder growth simulation (700 simulation steps).

exposed to the total execution time of this simulation program. Due to the decomposition method for MPI parallelization, the number of nodes must be a divisor of 40. Because of the poor parallelism, the performance degrades when the number of nodes increases beyond 8.

As shown in Figure 7, the clDataTransfer outperforms the baseline implementation because it can exploit an optimized implementation that overlaps the host-device communication with the internode communication in a pipelined fashion for sufficiently large messages. Accordingly, these results indicate that a higher performance can be achieved by appropriately interoperating MPI and OpenCL, and the clDataTransfer enables us to express the interoperation in a simple and effective way.

In the above evaluation, by just replacing the combination of `MPI_Recv` and `clEnqueueWriteBuffer` with `clEnqueueRecvBuffer`, the pipeline data transfer is used for the communication and leads to a higher sustained bandwidth. Hence, the results also suggest that application programmers can incrementally improve their MPI programs so as to use the clDataTransfer extension. This is very important because most of existing applications have been developed using MPI.

6. Conclusions

This paper has proposed an OpenCL extension, clDataTransfer, to allow OpenCL to perform data transfers that need collaboration between hosts and compute devices. In the clDataTransfer extension, additional OpenCL commands are defined for encapsulating common programming patterns in data transfers from/to the device memory, such as internode communications and file accesses. The additional commands are executed in the same way as the other OpenCL commands. Using OpenCL event objects, we can express the dependency among both conventional and additional commands. Therefore, data transfers indicated by the additional commands are incorporated into the OpenCL execution model in a natural manner.

As data transfers are abstracted as OpenCL commands, the implementation details of the data transfers are hidden from application codes. Hence, clDataTransfer will be able to exploit new features of the latest devices without any user code change. As a result, clDataTransfer would allow today's applications to benefit from hardware improvements without making any code change or even without recompiling the application. That is, clDataTransfer can improve not only the performance but also the performance portabilities across system types, scales, and generations.

The performance evaluation results clearly show that clDataTransfer can achieve efficient data transfers while hiding the complicated implementation details, resulting in higher performance and scalability. Moreover, using the clDataTransfer extension, the host thread of an application is not blocked to serialize dependent operations of data transfers. As a result, the clDataTransfer extension allows an application programmer to easily use the opportunities to overlap communications and storage accesses with computations.

Although this work focuses on OpenCL, we believe that the idea itself could be applicable to other programming models such as CUDA. In the future, we will further improve the extension so that it can support other kinds of tasks that need help of host threads, such as system calls.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors would like to thank Professor Mayasa Shigeta and Professor Fumihiko Ino of Osaka University for allowing them to use their simulation codes in the performance evaluation. The authors would also like to thank the RIKEN Integrated Cluster of Clusters (RICC) at RIKEN for the user supports and the computer resources used for the performance evaluation. This research is partially supported by JST CREST "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems" and Grants-in-Aid for Scientific Research (B) nos. 25280041 and 25280012. The work is also partly supported by DoE Vancouver Project (DE-SC0005515).

References

[1] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann Publishers, 2007.

[2] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, Morgan Kaufmann, Boston, Mass, USA, 2011.

[3] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, The MIT Press, 1999.

[4] O. S. Lawlor, "Message passing for GPGPU clusters: cudaMPI," in *Proceedings of the IEEE International Conference on Cluster Comptuing and Workshops (CLUSTER '09)*, pp. 1–8, 2009.

[5] A. M. Aji, J. Dinan, D. Buntinas et al., "MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems," in *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications (HPCC '12)*, pp. 647–654, Liverpool, UK, June 2012.

[6] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters," *Computer Science: Research and Development*, vol. 26, no. 3-4, pp. 257–266, 2011.

[7] J. A. Stuart, P. Balaji, and J. D. Owens, "Extending MPI to accelerators," in *Proceedings of the 1st Workshop on Architectures and Systems for Big Data (ASBD '11)*, pp. 19–23, 2011.

[8] I. Gelado, J. Cabezas, N. Navarro, J. E. Stone, S. Patel, and W.-M. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*, pp. 347–358, March 2010.

[9] J. A. Stuart and J. D. Owens, "Message passing on data-parallel architectures," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*, pp. 1–12, May 2009.

[10] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices," in *Proceedings of the IEEE International Conference on Cluster Computing Workshops and Posters*, pp. 1–7, September 2010.

[11] Y. Munekawa, F. Ino, and K. Hagihara, "Design and implementation of the Smith-Waterman algorithm on the CUDA-compatible GPU," in *Proceedings of the 8th IEEE International Conference on BioInformatics and BioEngineering (BIBE '08)*, pp. 1–6, October 2008.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, vol. 25, pp. 1097–1105, 2012.

[13] E. H. Phillips and M. Fatica, "Implementing the Himeno benchmark with CUDA on GPU clusters," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS '10)*, pp. 1–10, April 2010.

[14] M. Shigeta and T. Watanabe, "Growth model of binary alloy nanopowders for thermal plasma synthesis," *Journal of Applied Physics*, vol. 108, no. 4, Article ID 043306, 2010.

[15] The Open MPI Project, "Open MPI: open source high performance computing," http://www.open-mpi.org/.