An OpenCL Framework for Heterogeneous Multicores with Local Memory*

Jaejin Lee†, Jungwon Kim†, Sangmin Seo†, Seungkyun Kim†, Jungho Park†, Honggyu Kim†, Thanh Tuan Dao†, Yongjin Cho†, Sung Jong Seo‡, Seung Hak Lee‡, Seung Mo Cho‡, Hyo Jung Song‡, Sang-Bum Suh‡, and Jong-Deok Choi‡

†School of Computer Science and Engineering, Seoul National University, Seoul 151-744, Korea
‡Samsung Electronics Co., Nongseo-dong, Giheung-gu, Yongin-si, Geonggi-do 446-712, Korea

{jlee, jungwon, sangmin, seungkyun, jungho, honggyu, thanhtuan, yongjin}@aces.snu.ac.kr
{sj1557.seo, s.hak.lee, seungm.cho, hjsong, sbuk.suh, jd11.choi}@samsung.com

http://aces.snu.ac.kr

ABSTRACT

In this paper, we present the design and implementation of an Open Computing Language (OpenCL) framework that targets heterogeneous accelerator multicore architectures with local memory. The architecture consists of a general-purpose processor core and multiple accelerator cores that typically do not have any cache. Each accelerator core, instead, has a small internal local memory. Our OpenCL runtime is based on software-managed caches and coherence protocols that guarantee OpenCL memory consistency to overcome the limited size of the local memory. To boost performance, the runtime relies on three source-code transformation techniques, *work-item coalescing*, *web-based variable expansion* and *preload-poststore buffering*, performed by our OpenCL C source-to-source translator. Work-item coalescing is a procedure to serialize multiple SPMD-like tasks that execute concurrently in the presence of barriers and to sequentially run them on a single accelerator core. It requires the web-based variable expansion technique to allocate local memory for private variables. Preload-poststore buffering is a buffering technique that eliminates the overhead of software cache accesses. Together with work-item coalescing, it has a synergistic effect on boosting performance. We show the effectiveness of our OpenCL framework, evaluating its performance with a system that consists of two Cell BE processors. The experimental result shows that our approach is promising.

*This work was supported in part by the Creative Research Initiatives (Center for Manycore Programming, 2009-0081569) of MEST/KOSEF and by Samsung Electronics. ICT at Seoul National University provided research facilities for this study.

Categories and Subject Descriptors

D.3.4 [**PROGRAMMING LANGUAGES**]: Processors—*Code generation, Compilers, Optimization, Run-time environments*

General Terms

Algorithms, Design, Experimentation, Languages, Measurement, Performance

Keywords

OpenCL, Compilers, Runtime, Software-managed caches, Memory consistency, Work-item coalescing, Preload-poststore buffering

1. INTRODUCTION

Multicore accelerator architectures speed up computationally intensive parts of applications. They are widening their user base to all computing domains. This results in a new trend in applications ranging from embedded and consumer software to high-performance computing applications. It has been known that exploiting special hardware accelerators can dramatically speedup applications, and such accelerator architectures are available in the form of off-the-shelf multicore processors, such as general-purpose GPUs[31] and Cell BE processors[17, 21]. Following this trend, Open Computing Language (OpenCL)[23] is being developed by the Khronos Group.

OpenCL is a framework for building parallel applications that are portable across heterogeneous parallel platforms that consist of multicore CPUs, GPGPUs, and other multicores, such as DSPs and Cell BE processors. It has a strong CUDA[31] heritage, but it provides a common hardware abstraction layer across different multicore architectures. It allows application developers to focus their efforts on the functionality of their application, rather than the lower-level details of the underlying architecture.

OpenCL consists of a programming language, called OpenCL C, and OpenCL runtime API functions. OpenCL C is a subset of C99 with some extensions that include support for vector types, images, and memory hierarchy qualifiers. It is used to write computational kernels that execute on many

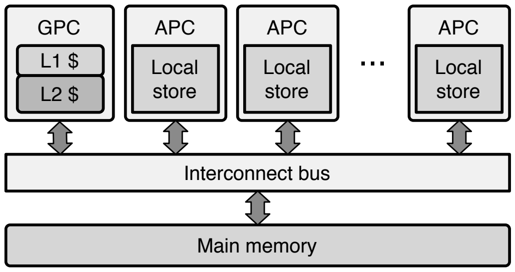


Figure 1: The target architecture.

(hundreds of) virtual processing elements (PEs). The API functions are used to configure and manage the OpenCL platform and to coordinate executions of the computational kernels.

In this paper, we describe the design and implementation of an OpenCL framework. It targets any multicore accelerator architecture that is similar to the architecture shown in Figure 1. The target architecture consists of a general-purpose processor core (GPC) and multiple accelerator processor cores (APCs). The GPC typically performs system management tasks and executes privileged OS kernel routines. APCs are dedicated to accelerating compute intensive workloads. GPCs and APCs are connected by a high speed interconnect bus.

The GPC is typically backed by a deep on-chip cache hierarchy. The hardware guarantees coherence between the caches and the main memory. On the other hand, APCs typically do not have any caches. Instead, each APC has a small internal local store that is not coherent with the main memory or with the local store in another APC. The programmer or software is responsible for explicitly moving data between the local store and the main memory. Communication mechanisms between cores are DMA transfers and mailbox messages. DMA transfers are for moving data and code between the main memory and an APC's local store. Mailboxes are for control communication between an APC and the GPC. A typical example of such an architecture is Cell BE[17, 21].

There are two major challenges in the design and implementation of the OpenCL framework on such an architecture. One is how to implement hundreds of virtual PEs with a single accelerator core and make them efficient. In order to guarantee the portability and high-performance of OpenCL applications across different parallel platforms, this problem must be solved. The other is how to overcome the limited size and incoherency of the local store. This issue is directly related to the correctness and performance of OpenCL applications.

Our OpenCL framework is based on software-managed caches[16, 20, 25, 35] and consistency protocols that guarantee OpenCL memory consistency to overcome the limited size and incoherency of the local store. In addition, it relies on a compiler technique, called *work-item coalescing* to efficiently provide hundreds of virtual PEs with a single accelerator core. Another compiler technique, called *preload-poststore buffering*, eliminates the need of software-managed caches for some data access patterns and overcomes the limited size of the local store, resulting in significant performance boosting. The major contributions of this paper are the following:

- We describe the design and implementation of our OpenCL framework for multicore accelerator architectures with small local stores.
- We propose the work-item coalescing technique. It efficiently serializes executions of hundreds of virtual PEs for an OpenCL kernel on a single accelerator core. As far as we know, there is no literature or a proprietary tool that describes such optimizations.
- As part of the work-item coalescing algorithm, we propose web-based variable expansion. It deals with variable expansion across different loop nests while the classical variable expansion technique is only applicable to a variable inside a single loop nest to remove loop carried dependences on the variable.
- We propose the preload-poststore buffering technique. It enables gathering DMA transfers for global array accesses together and minimizes the time spent waiting for them to complete by overlapping them. It has a synergistic effect on performance together with work-item coalescing. The previous buffering technique found in the literature aims at strided, regular array references in the innermost loop with sequential control flow. Ours is applicable to a multiply nested loop that contains conditionals and irregular array references.
- We show the effectiveness of our OpenCL framework by implementing the OpenCL runtime and a source-to-source translator. We evaluate its performance with an accelerator system that consists of two Cell BE processors using 12 OpenCL benchmark applications.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 briefly describes OpenCL and its features. Section 4 explains the design and implementation of our OpenCL runtime. Section 5, Section 6 and Section 7 describe the work-item coalescing, web-based variable expansion and preload-poststore buffering algorithms, respectively. Section 8 presents the results of evaluating the effectiveness of our OpenCL framework. Finally, Section 9 concludes the paper.

2. RELATED WORK

There are some new parallel programming models[3, 31, 33, 38] proposed for heterogeneous multicore architectures. Even though OpenCL inherited many features from CUDA[31], their approaches are significantly different. CUDA[31] and CTM[3] are programming models that are specific to a system that consists of CPUs and GPUs. Saha *et al.*[33] propose a programming model for a heterogeneous x86 platform. The target architecture consists of a general-purpose x86 CPU and Larrabee cores[34]. The model provides a uniform programming model for different platform configurations and a partly shared address space between the CPU and Larrabee cores. It also provides a direct user-level communication mechanism between the CPU and Larrabee cores. Wang *et al.*[38] propose an architecture, called *EXO*, to represent heterogeneous accelerators as MIMD resources and a programming environment, called *CHI*, which supports tightly-coupled integration of heterogeneous cores. In EXO, accelerator cores are exposed to the programmer as

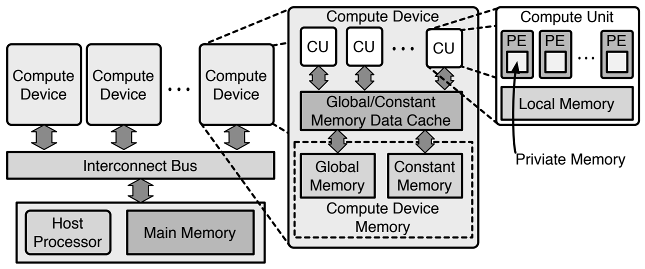


Figure 2: The OpenCL platform model.

```
void vec_add(int n,
             float *a,
             float *b,
             float *c)
{
  int i;

  for (i=0; i<n; i++)
    c[i] = a[i]+b[i];
}
```

(a) C function

```
__kernel
void vec_add(__global float *a,
             __global float *b,
             __global float *c)
{
  int id;

  id = get_global_id(0);
  c[id] = a[id]+b[id];
}
```

(b) OpenCL kernel

Figure 3: A C function and its OpenCL kernel.

application-level MIMD functional units. CHI supports shared memory multithreaded programs for a heterogeneous multicore processor. There are publicly available OpenCL implementations released from some industry-leading companies: Apple, AMD, IBM, and NVIDIA[4, 5, 19, 30].

There are different proposals of software-managed caches[6, 13, 16, 25, 35] for heterogeneous multicores with local memory, especially for Cell BE processors. Our OpenCL runtime uses the software-managed cache that is similar to the extended-set-index cache (ESC) proposed by Seo *et al.*[35]. ESC supports page-level caching. Lee *et al.*[24, 25] propose a software shared virtual memory (SVM) that is based on the ESC[35] and centralized release consistency for Cell BE processors. Our multiple writer protocol in the OpenCL runtime is similar to their multiple writer protocol included in the implementation of centralized release consistency.

The idea of overlapping remote communications for data accesses can be found in the studies of Unified Parallel C compilers and runtimes[11, 18] for distributed-memory multiprocessors. Chen *et al.*[10] propose a data prefetching technique for software-managed caches. Their technique targets Cell BE processors and exploits overlapping DMA transfers. Liu *et al.*[26] propose Direct Blocking Data Buffer (DBDB) for OpenMP compilers that targets Cell BE processors. Their technique aims at strided, regular array references in the innermost loop with sequential control flow. Our preload-poststore buffering algorithm is more general than theirs. It is applicable to a multiply nested loop that contains conditionals. Moreover, it handles irregular array references, such as subscript-of-subscript references, in addition to regular array references.

3. BACKGROUND

In this section, we briefly introduce OpenCL[23].

3.1 OpenCL Platform

The OpenCL platform (Figure 2) consists of a host processor connected to one or more *compute devices*, each of which contains one or more *compute units* (CUs). Each CU contains one or more *processing elements* (PEs). A PE is a *virtual* scalar processor.

An OpenCL application consists of a *host program* and *kernels*. The host program executes on the host processor and submits *commands* to perform computations on the PEs within a compute device or to manipulate memory objects. There are three types of commands: kernel execution, memory, and synchronization. A kernel is a function and written in OpenCL C. It executes on a single compute device. It is submitted to a *command-queue* in the form of a kernel execution command by the host program. A compute device may have one or more command queues. Commands in a command-queue are executed in-order or out-of-order depending on the queue type. Kernel programs can be compiled either at runtime or in advance.

3.2 Abstract Index Space

When a kernel is submitted, the host program defines an N-dimensional abstract index space, where $1 \leq N \leq 3$. Each point in the index space is specified by an N-tuple of integers with each dimension starting at 0. Each point is associated with an execution instance of the kernel, which is called *work-item*. The N-tuple is the global ID of the corresponding work-item. Each work-item is able to query its ID, and can perform a different task and access different data based on its ID (i.e., Single Program, Multiple Data (SPMD)[12]). An integer array of length N (i.e., the dimension of the index space) specifies the number of work-items in each dimension of the index space and is prepared by the host program for the kernel when the kernel command is enqueued.

A *work-group* contains one or more work-items. Each work-group has a unique ID that is also an N-tuple. An integer array of length N (i.e., the dimension of the index space) specifies the number of work-groups in each dimension of the index space. A work-item in a work-group is assigned a unique local ID within the work-group, treating the entire work-group as the local index space. The global ID of a work-item can be computed with its local ID, work-group ID, and work-group size. Work-items in a work-group execute *concurrently* on the PEs of a single CU.

3.3 Memory Regions

OpenCL defines four distinct memory regions in a compute device: global, constant, local and private (Figure 2). *Compute device memory* consists of the global and constant memory regions. These regions are accessible to work-items, and OpenCL C has four address space qualifiers to distinguish these memory regions: `__global`, `__local`, `__constant` and `__private`. They are used in variable declarations in the kernel code. Accesses to the global memory or the constant memory may be cached in the global/constant memory data cache (Figure 2) if there is such a cache in the device.

Figure 3 shows a sample kernel. The function `get_global_id(0)` returns the first element (i.e., 0 for the first dimension, 1 for the second, and 2 for the last) of the global ID of the work-item that executes the kernel. The host program creates memory objects for arrays `a`, `b`, and `c` in the device memory and initializes them appropriately. Then, it passes pointers to the memory objects to the kernel when it enqueues the kernel command.

3.4 Synchronization

A work-group barrier used in the kernel synchronizes work-items in a single work-group. Every work-item in the work-group must execute the barrier and cannot proceed beyond the barrier until all other work-items in the work-group reach the barrier. The work-group barrier enforces a global or local memory fence. Between work-groups, there is no synchronization mechanism available in OpenCL.

Synchronization between commands in a single command-queue can be specified by a command-queue barrier command. To synchronize commands in different command-queues, *events* are used. Each OpenCL API function that enqueues a command returns an event object that encapsulates the command status. In addition, most of OpenCL API functions that enqueue a command take an *event list* as an argument. This command cannot be issued for execution until all the commands associated with the event list complete.

3.5 OpenCL Memory Consistency

OpenCL defines a relaxed memory consistency model[1, 15] for consistent memory. An update to a memory location by a work-item may not be visible to all the other work-items at the same time. Instead, the local view of memory from each work-item is guaranteed to be consistent at synchronization points. Synchronization points include work-group barriers, command-queue barriers, and events.

4. OPENCL RUNTIME

In this section, we describe the design and implementation of our OpenCL runtime.

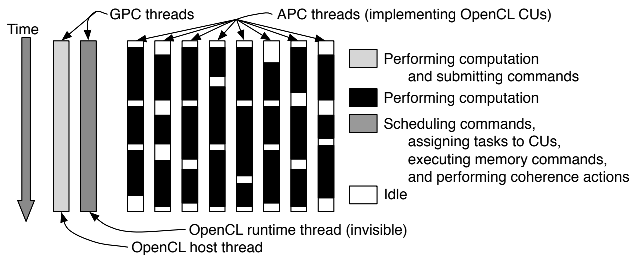
OpenCL Platform	Target Architecture
Host processor	GPC
CU	APC
Private memory	APC local store
Data cache	APC local store
Constant memory	Main memory
Compute device	A set of APCs
PE	Virtual PE
Local memory	APC local store
Global memory	Main memory
Main memory	Main memory

Table 1: Mapping platform components to the target architecture.

Our OpenCL runtime maps each OpenCL platform component to a component in the target architecture. This mapping is summarized in Table 1. The GPC becomes the OpenCL host. A set of $M(\geq 1)$ APCs becomes a compute device, and each APC becomes an OpenCL CU. The OpenCL runtime provides $N(\geq 1)$ virtual PEs in each APC. The APC emulates the PEs.

To accelerate data accesses from the device memory, the runtime implements a software-managed cache[16, 20, 25, 35] in each APC's local store. It caches the contents of the global and constant memory. The software cache in combination with our coherence protocol provides the OpenCL global/constant memory data cache that satisfies the OpenCL memory consistency model.

Figure 4 illustrates the threads implemented in our OpenCL runtime. A thread running on an APC implements a CU. The GPC runs two threads: host thread and *OpenCL runtime thread* (ORT). The host thread executes the OpenCL host program. The host thread and the ORT share command-queues. The ORT continuously invokes a *command scheduler*, a *command executor*, and a *coherence manager* for *each* compute device in turn.



Time
GPC threads
APC threads (implementing OpenCL CUs)
Performing computation and submitting commands
Performing computation
Scheduling commands, assigning tasks to CUs, executing memory commands, and performing coherence actions
Idle
OpenCL runtime thread (invisible)
OpenCL host thread

Figure 4: Threads in our OpenCL runtime.

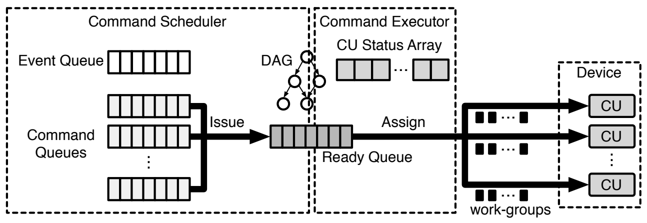


Figure 5: The command scheduler and the command executor.

4.1 Command Scheduler

The host thread and the command scheduler share an event-queue (Figure 5). The ORT has a *ready-queue* where issued commands from command-queues are enqueued and wait for execution. Whenever the host thread enqueues a command in a command-queue, an event object that encapsulates the command status is created. The host thread enqueues the event object in the event-queue.

The ORT checks if the event-queue is empty. When the event-queue is not empty, say it contains N elements, the ORT invokes the command scheduler. The command scheduler determines execution ordering between queued commands and issues them to the ready-queue. It maintains a directed acyclic graph (DAG) that is shared with the command executor. Each node in the DAG is an event object. Each edge $m \rightarrow n$ represents the execution ordering of two commands that are associated with event objects m and n.

The command scheduler dequeues N event objects one by one from the event-queue. It adds the dequeued event object to the DAG by determining the execution ordering between the dequeued object and the objects in the DAG. The execution ordering is obtained from the command-queue type (in-order or out-of-order) and the ordering enforced by command-queue barriers and event synchronizations. If an edge is added to the DAG due to synchronization, we call it a *synchronization edge*. When the dequeued object waits for an event object that is not in the DAG (the associated command has already completed), the dequeued object becomes a *synchronization node*. Synchronization edges and nodes become synchronization points where the command executor flushes the software-managed caches. When there are event objects with no incoming edges in the resulting DAG, the command scheduler dequeues the associated commands from their command-queues and issues them to the ready-queue.

4.2 Command Executor

When the ready-queue is not empty, the ORT invokes the command executor. The command executor executes the commands in the ready-queue by scheduling work-groups on the APC threads and performing memory transfer operations between the host memory and device memory. It maintains an array, called *CU status array*, which records the status of all the CUs (i.e., APCs) in the compute device. Each element of the array records whether the corresponding CU is busy (i.e., performing computation) or idle (i.e., ready to take another task).

The command executor dequeues a command from the ready-queue. If the command is a kernel command, it assigns work-groups for the kernel to the available CUs by checking the CU status array. When there are still remaining idle CUs after assigning all the work-groups in the kernel, it dequeues another kernel command and assigns work-groups for the kernel to the idle CUs. When there are no remaining work-groups to schedule and all the CUs that execute the kernel have completed, the kernel command completes execution. If the dequeued command is a memory command, then the command executor executes it directly. Otherwise, the command is a synchronization command.

When the host thread enqueues a blocking command, it goes to sleep after enqueueing the command. The command executor wakes up the host thread after the command completes. This sleep and wake-up mechanism is implemented with a semaphore between the host thread and the command executor.

4.3 Coherence Manager

Since there is no hardware coherence mechanism supported by the target architecture between the APC's local store (i.e., software-managed cache) and the main memory (i.e., the global or constant memory), the runtime must provide a coherence mechanism. The coherence manager performs coherence actions for software-managed caches. To exploit the software-managed caches, each global or constant memory access in the kernel code is replaced with a cache access API function by our source-to-source translator. Whenever a CU (i.e., APC thread) experiences a miss in its cache or writes back a cache block to the global or constant memory, the CU sends a block request to the ORT. The coherence manager handles the request. The coherence manager implements a multiple-writer protocol[22, 25] to solve coherence problems that are related to false sharing. It maintains a directory entry for each global and constant memory block.

In our runtime, all work-items in a single work-group execute on the same APC (i.e., CU) and the local memory for the CU is allocated in the APC's local store. Thus, the local memory is consistent across work-items in the same work-group at all times without regard to a work-group barrier. The same is true for the global memory because each work-item in the same work-group accesses the same software-managed cache implemented in the APC's local memory. Consequently, the global or local memory fence at a work-group barrier is automatically enforced.

To guarantee OpenCL memory consistency for shared memory objects between commands, the command executor flushes software-managed caches whenever it dequeues a command from the ready-queue or it removes an event object from the DAG after the associated command has completed. More specifically, if the dequeued command is associated with a synchronization node in the DAG, the command executor flushes all cached dirty blocks in the APCs (i.e., CUs). If an outgoing edge from the removed event object is a *synchronization edge*, the command executor also flushes all cached dirty blocks in the APCs (i.e., CUs) where the work-groups in the associated kernel command have executed.

4.4 Emulating PEs

When a work-group is assigned to a CU, each work-item in the work-group executes as if it were executed on one or more virtual PEs concurrently. Our runtime assumes that a work-item executes on a single virtual PE. When a work-group is assigned to a CU, the corresponding APC thread executes each work-item in the work-group one by one performing context switching.

When a work-item reaches a work-group barrier, the APC thread saves the contents of the work-item's private memory and processor state in a reserved space in the main memory. Then, it switches to the next work-item. After executing the kernel code for all work-items until they reach the barrier, the runtime executes them one by one again from the program point just beyond the barrier after restoring the saved context and the contents of the private memory for each work-item. Work-item context switching is implemented with standard C `setjmp` and `longjmp` functions[14] and DMA operations to save and restore the processor state and private memory (if possible, by exploiting the double buffering mechanism).

```
#define get_global_id(N) \
  (__global_id[N] + (N == 0 ? __i : (N == 1 ? __j : __k)))

int __i, __j, __k;

__kernel void vec_add
  (__global float *a, __global float *b, __global float *c)
{
  int id;

  for ( __k = 0; __k < __local_size[2]; __k++ ) {
    for ( __j = 0; __j < __local_size[1]; __j++ ) {
      for ( __i = 0; __i < __local_size[0]; __i++ ) {
        id = get_global_id(0);
        c[id] = a[id] + b[id];
      }
    }
  }
}
```

Figure 6: Transforming the kernel in Figure 3 (b).

5. WORK-ITEM COALESCING

Executing work-items on a CU by switching one work-item to another incurs a significant overhead. To avoid this overhead, we rely on the work-item coalescing technique that is provided by our OpenCL C source-to-source translator.

Under OpenCL execution semantics, work-items in a single work-group execute concurrently. When a kernel and its callee functions do not contain any barrier, any execution ordering defined between the two statements from different work-items in the same work-group satisfies the OpenCL semantics. Based on this observation, the translator simply encloses the kernel code with a *triply nested loop* when the kernel and its callee functions do not contain any barrier. This loop iterates on the index space of a single work-group. We name the loop as a *work-item coalescing loop* (WCL).

Figure 6 shows the result of transforming the kernel code in Figure 3 (b). This code executes all work-items in a single work-group sequentially one by one. When a work-group is assigned to a CU, the CU (i.e., APC) actually executes this code for the kernel to emulate its virtual PEs. The size of the work-group index space is determined by the array `__local_size` provided by the OpenCL runtime. Depending on the values of `__local_size[0]`, `__local_size[1]`, and `__local_size[2]`, the WCL effectively becomes a triply, doubly, or singly nested loop at run time. The runtime provides the array `__global_id` that contains the global ID of the first work-item (when the local index space is three dimensional, its local ID is (0,0,0)) in the work-group.

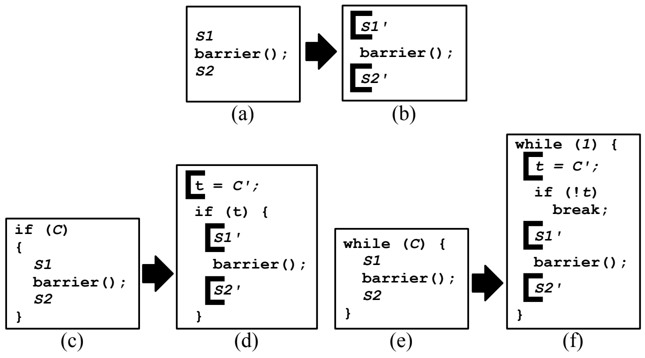


Figure 7: Coalescing work-items.

5.1 Observations

Consider the body of a kernel in Figure 7 (a). Due to the work-group barrier, the execution of work-items in a single work-group cannot be serialized. Our translator transforms the body to the code shown in Figure 7 (b). A thick left angle bracket in Figure 7 (b) stands for the WCL. `S1'` and `S2'` are obtained after applying variable expansion to `S1` and `S2`. Each of `S1` and `S2` is a compound statement or a list of statements. The barrier function in Figure 7 (b) is just a place holder and it inhibits some compiler optimizations by the target native compiler, such as loop fusion or code motion across the position of the barrier.

The variable expansion technique is applied to work-item private variables. It is similar to scalar expansion[2], but it also expands work-item private arrays. We elaborate on the web-based variable expansion technique later in Section 6.

There is no specific execution ordering defined by OpenCL between two statements from different work-items if both of them appear either before the barrier or after the barrier, but any two statements separated by the barrier must honor the execution ordering enforced by the barrier. The transformed code enforces an execution ordering of statements that is consistent with the ordering enforced by the original kernel code.

According to the OpenCL specification[23], a work-group barrier must satisfy the following conditions:

- *If a barrier is inside a conditional statement, then all work-items must enter the conditional if any work-item enters the conditional statement and executes the barrier.*

- *If a barrier is inside a loop, all work-items must execute the barrier for each iteration of the loop before any are allowed to continue execution beyond the barrier.*

These conditions imply that once entering a conditional or loop, each statement inside the construct can be executed for all work-items in the same work-group by iterating the work-group local index space, and the number of iterations of the loop is the same for all the work-items. In addition, making a decision on whether entering/exiting the construct or not for all work-items in the same work-group can be done by checking the result of evaluating the condition for a single work-item.

Figure 7 (c) shows the body of a kernel that contains a barrier in a conditional statement. Based on the observation, we can transform this code to that shown in Figure 7 (d) after applying variable expansion. The condition `C'` is evaluated and assigned to a temporary scalar variable `t`, and `t` does not need to be expanded. The resulting code honors the execution ordering enforced by the barrier in the original kernel. Again, the barrier function in Figure 7 (d) is just a place holder. Similarly, the code in Figure 7 (e) is converted to the code in Figure 7 (f).

5.2 Identifying WCRs

A kernel code region that needs to be enclosed with a WCL (i.e., enclosed with a thick left angle bracket in Figure 7) is called a *work-item coalescing region* (WCR). Our translator performs work-item coalescing by identifying WCRs, and then applying web-based variable expansion. To identify WCRs in a kernel, it performs the depth-first traversal on the abstract syntax tree (AST) of the kernel. A WCR is a maximal subtree or statement list that does not contain any work-group barrier. The details of the algorithm are omitted due to the page limit.

When the kernel invokes a function that contains a barrier, the function can be inlined with all its ancestors in the static call graph. Then, the inlined kernel can be processed with our work-item coalescing algorithm. However, our algorithm does not work well with kernels that contain `goto`s. If the kernel code contains a `goto`, we rely on the context switching mechanism to execute the kernel.

6. WEB-BASED VARIABLE EXPANSION

A work-item private variable that is defined in one WCR and used in another needs a separate location for different work-items. This is also true for work-item private arrays. Scalar variable references are replaced with references to a new temporary three-dimensional array that has a separate location for each work-item. Scalar expansion[2] is typically applied to a single loop nest to remove dependences at the expense of additional memory. However, our variable expansion technique is applied to multiple loop nests to transfer data defined in one loop nest to another. Thus, a *web* for a variable is the unit of expansion in our algorithm. A *web* for a variable is all du-chains of the variable that contain a common use of the variable[28]. A *du-chain* for a variable connects a definition of the variable to all uses reached by the definition[28]. The advantage of using a web as the unit of expansion is that we do not need to expand all the references of the variable.

After identifying webs for a work-item private variable, the algorithm collects (in T) webs whose definitions and uses belong to more than one WCRs. Then, it partitions T into equivalence classes. Two different webs are in the same equivalence class if and only if they have a common WCR to which at least one of their definitions or uses belongs. Vari-

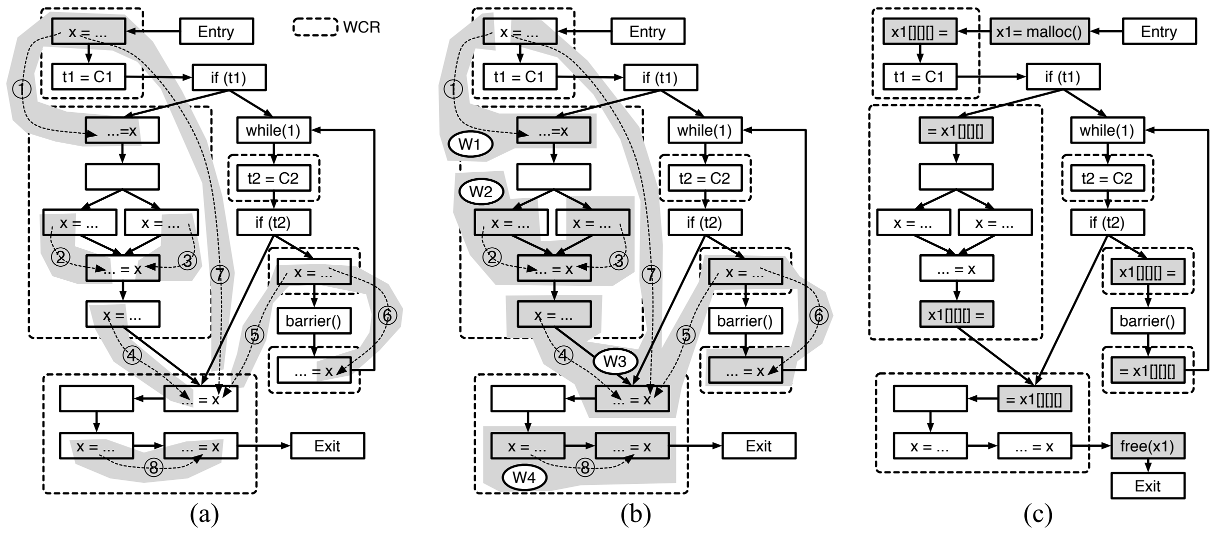


Figure 8: (a) Identifying du-chains. (b) Identifying webs. (C) After variable expansion.

able references in each equivalence class will be replaced with references of a new temporary three-dimensional array.

To allocate the APC local store space for the new temporary array of an equivalence class, we construct the dominator and postdominator trees of the kernel CFG. Then, we identify a `malloc()` node and a `free()` node that satisfy the following conditions:

- Among common ancestors of all the definitions of the equivalence class in the dominator tree, the `malloc()` node is the lowest node in the tree that dominates the `free()` node and is postdominated by the `free()` node.
- Among common ancestors of all the uses of the equivalence class in the postdominator tree, the `free()` node is the lowest node in the tree that postdominates the `malloc()` node and is dominated by the `malloc()` node.

If the identified `malloc()` node belongs to a WCR, we create a CFG node that contains `malloc()` for the array and make it as the immediate predecessor of the WCR. Otherwise, we make the new CFG node as the immediate predecessor of the `malloc()` node in the CFG. Inserting a CFG node that contains `free()` for the array is similar to the case of `malloc()`.

For example, in Figure 8 (a), there are six du-chains: {①, ⑦}, {②}, {③}, {④}, {⑤, ⑥}, and {⑧}. These du-chains produce four different webs: W1 = {①}, W2 = {②, ③}, W3 = {④, ⑤, ⑥, ⑦}, and W4 = {⑧}. Figure 8 (b) contains the CFG after identifying webs for variable `x`. The definitions and uses of `x` in W1 do not belong to the same WCR. The same is true for W3. Thus, we partition {W1, W3} into equivalence classes. After partitioning, we have a single equivalence class {W1, W3} as a result because W1 and W3 has a WCR in common due to ⑦ (or ① and ④). The references of `x` in W1 and W3 are to be replaced with references of the same three-dimensional array. Figure 8 (c) shows the result of expansion. Note that we do not need to expand the references of `x` in W2 and W4 because they are defined and used in the same WCR.

In our variable expansion, references to a private array are to be replaced with references to a new temporary array that has three more dimensions than the original array. To do so, each reference to an original array element is treated as a reference to the entire original array. That is, the entire array will be treated as a scalar variable, and we can apply the same variable expansion algorithm. A pointer is also treated as a scalar variable and expanded in the same way. However, we do not apply our work-item coalescing algorithm to a kernel that contains a private variable whose address is taken by `&` operator because we cannot apply our variable expansion algorithm to such a case.

The details of the algorithm are omitted due to the page limit. Applying loop invariant code motion to WCLs is very helpful to minimize the local store space required for variable expansion.

7. PRELOAD-POSTSTORE BUFFERING

We apply preload-poststore buffering to the kernel code after coalescing work-items. Before we execute a loop, we preload buffer arrays with the array elements accessed in the loop. The array references in the loop are replaced with direct references to the buffer arrays. After executing the loop, the buffer arrays are written back (poststored) to the global memory if they have been modified.

With software caches, each APC has to wait for the DMA transfer of each missing cache block to complete. Preload-poststore buffering enables gathering DMA transfers together for array accesses and minimizes the time spent waiting for them to complete by overlapping them. While the conventional double buffering technique exploits overlapping computation and communication (e.g., DMA transfers), our preload-poststore buffering technique exploits overlapping different communications together in the stage of preload or poststore. With a strip-mining technique, the double buffering technique facilitates our preload-poststore buffering when the available space for the buffer is small.

7.1 Observations

Consider the code in Figure 9 (a). Without loss of generality, assume that the triply nested loop is the WCL and it encloses a WCR. The innermost loop contains five different

```
for ( k = 0; k < ls[2]; k++ ) {
  for ( j = 0; j < ls[1]; j++ ) {
    for ( i = 0; i < ls[0]; i++ ) {

      if (i < 100) {
        a[j][i] = c[j][b[i]];
      }
      c[j][b[i]] = a[j][3*i+1] + a[j][i+1024];

    }
  }
}
```

(a)

```
ALLOCATE BUFFERS;
for ( k = 0; k < ls[2]; k++ ) {
  for ( j = 0; j < ls[1]; j++ ) {
    PRELOAD(buf_b,&b[0], ls[0]);
    PRELOAD(buf_a1,&a[j][0], ls[0]+1024);
    for ( i = 0; i < ls[0]; i++ )
      PRELOAD(buf_a2[i], &a[j][3*i+1]);

    WAITFOR(buf_b);
    for ( i = 0; i < ls[0]; i++ )
      PRELOAD(buf_c[i], &c[j][buf_b[i]]);

    for ( i = 0; i < ls[0]; i++ ) {
      if (i < 100) {
        buf_a1[i] = buf_c[i];
      }
      buf_c[i] = buf_a2[i] + buf_a1[i+1024];
    }

    POSTSTORE(buf_a1,&a[j][0], ls[0]+1024);
    for ( i = 0; i < ls[0]; i++ )
      POSTSTORE(buf_c[i], &c[j][buf_b[i]]);
  }
}
DEALLOCATE BUFFERS;
```

(b)

Figure 9: (a) The kernel code that contains an irregular access pattern. (b) After applying preload-poststore buffering.

array element references: $a[j][i]$, $a[j][i+1024]$, $a[j][3*i+1]$, $b[i]$, and $c[j][b[i]]$.

We use the Bounded Regular Section Descriptor (BRSD)[7] to represent the section of array accessed by a regular array reference or the range of a loop index. For example, the range of the innermost loop index is $[0 : ls[0]-1 : 1]$, where 0 is the lower bound, $ls[0]-1$ is the upper bound, and 1 is the stride. In the innermost loop, the reference $a[j][i]$ accesses the array section $a[j][0 : ls[0]-1 : 1]$. Note that j is loop invariant for the innermost loop. Similarly, $a[j][i+1024]$, $a[j][3*i+1]$, and $b[i]$ respectively access array sections $a[j][1024 : ls[0]+1023 : 1]$, $a[j][1 : 3*ls[0]-2 : 3]$, and $b[0 : ls[0]-1 : 1]$.

Since each of $a[j][0 : ls[0]-1 : 1]$ and $a[j][1024 : ls[0]+1023 : 1]$ is contiguous (stride one) and they are two different sections of the same array, we can assign them a single buffer array with the size of $ls[0]+1024$.

Suppose that we assigned a single buffer array to both of $a[j][0 : ls[0]-1 : 1]$ and $a[j][1 : 3*ls[0]-2 : 3]$. But, this is wasting the buffer space because $a[j][1 : 3*ls[0]-2 : 3]$ is not contiguous. In this case, it is better to assign two different buffer arrays with size $ls[0]$ to $a[j][0 : ls[0]-1 : 1]$ and $a[j][1 : 3*ls[0]-2 : 3]$. We load them with $ls[0]$ elements of a each before the innermost loop. This is totally legal because the equation $i = 3*i+1$ has no solution in the range of i, $[0 : ls[0]-1 : 1]$. $a[j][i]$ and $a[j][3*i+1]$ do not access the same array element in the same iteration of the innermost loop. Suppose that we have $a[j][3*i-6]$ in the code instead of $a[j][3*i+1]$. The equation $i = 3*i-6$ has a solution $i = 3$. This implies that when $i = 3$, the work-item that assigns a value to $a[j][i]$ has to read the value again at $a[j][3*i-6]$. Thus, we cannot assign two different buffers to the references $a[j][i]$ and $a[j][3*i-6]$. We do not apply preload-poststore buffering to them in this case.

We can assign buffer arrays to subscript-of-subscript references in the same way. Note that $c[j][b[i]]$ in Figure 9 (a) may have a loop-carried flow (read after write, true) dependence[2, 39] for the innermost loop due to the subscript of subscript. A conservative data dependence analysis[2, 39] can detect the dependence. However, loop-carried data dependences do not matter in this case because OpenCL does not define any specific execution ordering between two global accesses from different work-items. Moreover, an update to an array in a WCR by a work-item may not be visible to another work-item due to OpenCL memory consistency model. Note that a WCR does not contain a barrier. Figure 9 (b) shows the code after preload-poststore buffering is applied to the code in Figure 9 (a).

7.2 Conditions for Buffering

We apply preload-poststore buffering algorithm to a subclass of loops that are *well behaved*[28]. A well-behaved loop is a `for` loop in the form `for(`$e1$; $e2$; $e3$`)` *stmt*, in which $e1$ assigns a value to an integer variable i, $e2$ compares i to a loop invariant expression, $e3$ increments or decrements i by a loop invariant expression, and *stmt* contains no assignment to i. In addition, we assume that there is no aliasing between arrays in preload-poststore buffering.

Let i be the index variable of a *well-behaved* loop $\mathcal{L}$. An array subscript expression is *affine to the loop index variable* if it is in the form of $c*i+d$, where c and d are loop invariant to $\mathcal{L}$. It is *contiguous* if c is either 1 or -1. An array subscript expression is *affine to an array reference* if it is in the form of $c*x+d$, where x is an array reference and c and d are loop invariant to $\mathcal{L}$. Let $a[e_0, e_1, ..., e_{n-1}]$ be an n-dimensional array reference contained in the loop. The array reference is *affine to the loop index variable* if each e_j is affine to the loop index variable or loop invariant to $\mathcal{L}$. It is contiguous if e_{n-1} is contiguous and each e_j other than e_{n-1} is loop invariant to $\mathcal{L}$.

In preload-poststore buffering, an array reference is treated as a buffering candidate if it satisfies the following condition:

Condition 7.1 (Buffering candidate). *An array reference $a[e_0, e_1, ..., e_{n-1}]$ is a buffering candidate if each e_j is affine to the loop index variable or to a buffering candidate x_j, and x_j is not defined in $\mathcal{L}$, where x_j is a reference to an array other than a .*

We try to assign a different buffer to a different array reference until there is no room available in the buffer space. But, we try to assign a single buffer array to different references that are contiguous and partly overlap with each other. Suppose that we have performed a reaching definitions data-flow analysis for array references in $\mathcal{L}$ and assume that $\mathcal{L}$ is the innermost loop of a WCL. To assign buffers, array references must satisfy the following two conditions:

Condition 7.2 (Single buffer I). *If a definition of an array reference x reaches a use of another reference y of* *the same array and they have a loop-independent flow (read-after-write) dependence, then a single buffer array must be assigned to x and y.*

In other words, there is a loop-independent flow (read after write) dependence between them in a flow-sensitive manner. The equations $e_0^x = e_0^y$, $e_1^x = e_1^y$, ..., $e_{n-1}^x = e_{n-1}^y$ of the subscript expressions of the two references $x(= a[e_0^x, e_1^x, ..., e_{n-1}^x])$ and $y(= a[e_0^y, e_1^y, ..., e_{n-1}^y])$ have a solution in the range of the loop index variable if they access the same location in the same iteration. Note that loop-carried dependences do not matter because $\mathcal{L}$ is the innermost loop of the WCL.

Condition 7.3 (Single buffer II). *If a definition d_x of an array reference x reaches a definition d_y of another reference y of the same array and a definition g_y (possibly $g_y = d_y$) of y reaches a definition g_x (possibly $g_x = d_x$) of x, and if they have a loop-independent output (write-after-write) dependence, then a single buffer array must be assigned to the two references.*

If two different buffer arrays are assigned to x and y, we cannot determine the ordering of their poststores and one may overwrite the other in the global memory.

The above two conditions become stricter for the loop that are enclosed with the WCL because this loop is executed by each work-item. In this case, we have to consider loop-carried dependences. To obtain proper conditions for those loops, we replace "a loop-independent flow (read-after-write) dependence" with "a flow dependence" (loop independent or loop carried) and "a loop-independent output (write-after-write) dependence" with "an output dependence" (loop independent or loop carried) in Condition 7.2 and Condition 7.3, respectively.

Since we have a limited buffer space due to the limited size of the local store, it is not beneficial to assign a single buffer to a contiguous reference and a non-contiguous reference, or two non-contiguous references. In addition, if the access stride small enough compared to the page size of the software-managed cache, it is not beneficial to apply preload-poststore buffering to such a case.

Using the above three conditions, we can obtain a set of array references that are *bufferable* for $\mathcal{L}$. After assigning buffers to bufferable array references with a constraint on the size of available buffer space, we can obtain equivalence classes induced by the buffers assigned to them. Two different array references are in the same equivalence class if and only if the same buffer is assigned to them. We can define a partial ordering, called *poststore ordering*, between the assigned buffers using the result of the reaching definitions analysis. If a definition of an array reference x in an equivalence class buffer$_x$ reaches a definition of another reference y of the same array in different equivalence class buffer$_y$, then buffer$_x$ precedes buffer$_y$ in the poststore ordering. This ordering is used to properly order poststores of the buffers.

We apply preload-poststore algorithm to the innermost loop $\mathcal{L}$ (i.e., `__i` loop) of the WCL. With a given size of the available buffer space, after applying the algorithm to $\mathcal{L}$, we apply it to the inner loops of $\mathcal{L}$ one by one, from outside to inside if there is a well-behaved loop nested in $\mathcal{L}$ until there is no room in the available buffer space. As mentioned before, depending on the type of a loop (i.e., whether it is the innermost loop of the WCL or not), the type of dependences in Condition 7.2 and Condition 7.3 will be different. We omit the details of our preload-poststore algorithm due to the page limit.

7.3 Interaction with Software-managed Caches

Buffers used in preload-poststore buffering may share data with the software-managed caches in the OpenCL runtime, resulting in incoherent blocks in the software-managed caches. To solve this problem, the OpenCL runtime aligns the start address of a memory object to the cache block (page) boundary when there is a request from the host for allocating the memory object in the global memory. The preload-poststore buffering algorithm is modified to support the following: it either assigns buffers to all the references of the same array or no buffer is assigned to the references. If no buffer is assigned to the references of an array, the references access the software-managed cache. This is the simplest way of guaranteeing coherence between the software-managed caches and buffers used in preload-poststore buffering.

8. EVALUATION

In this section, we present our evaluation methodology and results for the current implementation of our OpenCL runtime and source-to-source translator.

8.1 Methodology

Target machine. We evaluate the performance of our OpenCL implementation using an IBM QS22 Cell blade server with two 3.2GHz PowerXCell 8i processors. It runs Fedora Linux 9. The Cell BE processor consists of a single Power Processor Element (PPE) and eight Synergistic Processor Elements (SPEs). Each SPE has 256KB of local store without any cache. The local store is not coherent to the main memory. An SPE can transfer data from/to the external main memory using asynchronous DMA transfers.

Benchmark applications. We run 12 OpenCL applications that are from various sources: PARSEC[8], IBM[19], NAS[29], Parboil[36], NVIDIA[30, 37], and Rodinia[9]. Their details are described in Table 2. The applications marked with "Vector" in the precision field are those that exploit explicit vector types.

To generate an OpenCL application from a non-OpenCL source, we convert the application manually to OpenCL programs. After conversion, all the applications that perform single precision floating point computations have been tested for their portability across different OpenCL platforms: NVIDIA OpenCL[30] on a GPU, Apple OpenCL[5] on a GPU, and AMD OpenCL[4] on an AMD multicore and an Intel multicore.

Runtime and source-to-source translator. We have implemented our OpenCL runtime for Cell BE multicore processors. We use IBM Cell SDK 3.1[20] to implement the runtime and to implement OpenCL built-in functions. We have implemented our OpenCL-C-to-C translator by modifying `clang` that is a C front-end for the LLVM compiler infrastructure[27]. In addition, we have implemented OpenCL host library and OpenCL C library for our OpenCL runtime. For array data dependence test, we use the Omega library[32] in our source-to-source translator. The OpenCL device in our runtime consists of 16 SPEs. A single PPE in the QS22 Cell blade server runs both the host thread and the ORT. The page size of software-managed caches in

Application	Source	Description	Precision	Input	A	B	C	D	E
BS	PARSEC	Black-Scholes PDE	Single	5242880, 100 iters.	300MB	1	0		X
BS-vec	IBM, OpenCL, TaskDB	Black-Scholes PDE	Vector, Single	5242880, 100 iters.	140MB	1	0		
Correlator	CUDA Zone	Radio astronomy signal correlator	Vector, Single	128 channels, 2 polarizations	224.3MB	1	0		X
CP	Parboil	Coulombic potential	Vector, Single	Default	4.1MB	1	0		X
EP	NAS	Embarrassingly parallel	Double	Class A	24MB	1	0		X
LC	Rodinia	Leukocyte tracking	Single	5 frames	1.6MB	3	7	X	X
MatrixMul	NVIDIA, OpenCL	Matrix multiplication	Single	3200x3200	156.3MB	1	2	X	X
MRI-Q	Parboil	Magnetic resonance imaging Q	Single	Large	5MB	2	0		X
NW	Rodinia	Needleman-Wunsch algorithm	Single	8192 dims.	768.2MB	2	10	X	X
PerlinNoise	IBM, OpenCL	Perlin noise generator	Vector, Single	1000 iters.	16KB	1	0		X
RPES	Parboil	Rys polynomial equation solver	Vector, Single	Default	79.4MB	2	2		X
TPACF	Parboil	Two-point angular correlation function	Single	Default	9.5MB	1	3	X	

A: global data size, B: # of kernels, C: # of work-group barriers, D: variable expansion, E: preload-poststore buffering.

Table 2: Applications used.

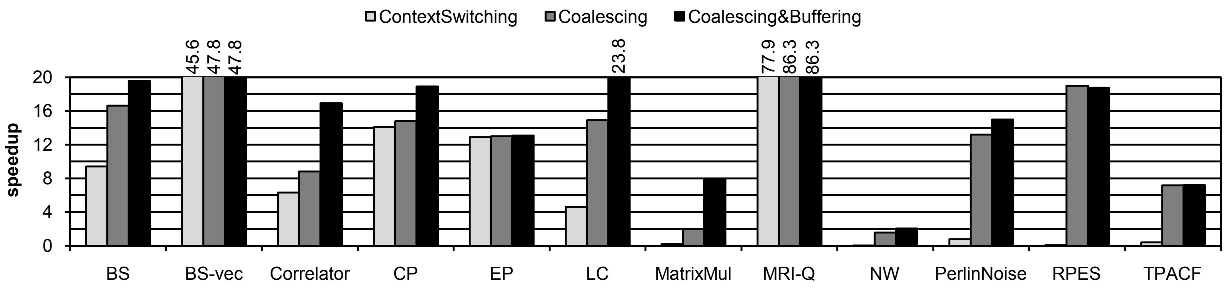
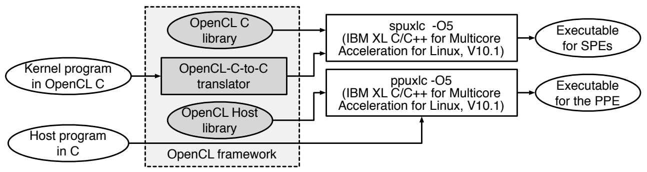


Figure 10: Speedup.

our runtime is configured to 8KB. The maximum number of work-items in a single work-group is 1024.



OpenCL C library
Kernel program in OpenCL C
OpenCL-C-to-C translator
OpenCL Host library
Host program in C
OpenCL framework
spuxlc -O5 (IBM XL C/C++ for Multicore Acceleration for Linux, V10.1)
ppuxlc -O5 (IBM XL C/C++ for Multicore Acceleration for Linux, V10.1)
Executable for SPEs
Executable for the PPE

Figure 11: Compilation process.

Figure 11 shows the compilation process for our OpenCL runtime. After translating the OpenCL C kernel program for each application with the source-to-source translator, our OpenCL runtime compiles the resulting C program with the spuxlc compiler in IBM XL C/C++ for Multicore Acceleration for Linux (V10.1)[21] to build the kernel program online and to generate an SPE executable. The ppuxlc compiler in the IBM XL C/C++ compiler suite compiles the host program to generate a PPE executable.

8.2 Results

Figure 10 shows the result of comparing different techniques proposed in this paper. It shows the speedup of each technique over a single PPE for each application. The bars labeled ContextSwitching shows the speedup of executing work-items by switching context using `setjmp` and `longjmp`. Coalescing labels the speedup obtained with work-item coalescing. Coalescing&Buffering denotes the speedup obtained using both work-item coalescing and preload-poststore buffering. The number of work-items in a single work-group varies from application to application and kernel to kernel. It is from 64 to 1024. Since we could not obtain a sequential version of BS-vec, we use the sequential version of BS to obtain the speedup of BS-vec.

Table 2 shows the number of kernels, the number of work-group barriers, whether web-based variable expansion in the work-item coalescing process is required or not, and the applicability of preload-poststore buffering for each application.

For all applications but RPES, Coalescing&Buffering performs best or is comparable to Coalescing. As you see in Table 2, preload-poststore buffering is not applied to BS-vec and TPACF. BS-vec does not have any array accesses. The target loop in TPACF is not a well-behaved loop. Preload-poststore buffering is applied to a loop in RPES, but the data accessed in the loop is very small and the loop overhead from buffering slightly degrades its performance. Similarly, preload-poststore buffering does not affect EP's performance because the number of global data reads and writes is very small.

LC, NW and RPES contain irregular array references. Our preload-poststore buffering was successful in LC and NW, but it was unsuccessful in RPES because the array indices are modified in another function through pointers. The global data access pattern of MRI-Q is very regular and has a lot of spatial locality. Moreover, it does not contain any work-group barrier in its kernels. This accounts for the huge speedup of MRI-Q in all of the execution schemes.

Due to the significant context switching overhead, Coalescing performs better than ContextSwitching for all applications. Especially, this fact manifests in the applications that contain barriers in their kernels, such as LC, MatrixMul, NW, RPES and TPACF. Of course, the performance of ContextSwitching also depends on the context size to be saved and restored from the main memory. The context size of RPES affects its performance a lot. PerlinNoise has only one kernel and this kernel is enqueued and executed many times. The data structure initialization overhead per kernel launch for ContextSwitching makes PerlinNoise very slow. The speedup of our OpenCL framework is especially bad for NW due to cache misses.

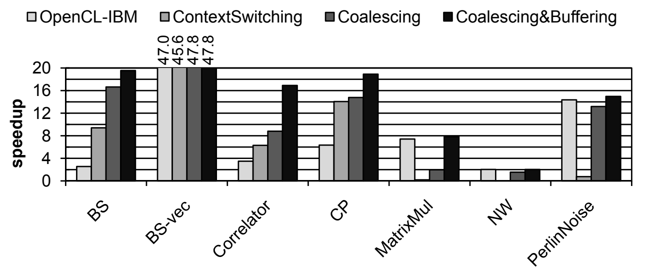


Figure 12: Comparison with the IBM OpenCL framework for Cell BE.

Figure 12 shows the result of comparing our OpenCL framework with the OpenCL framework in IBM OpenCL Development Kit for Linux on Power[19]. It is a publicly available technology preview version (version 0.2) of the OpenCL framework from IBM (as of June 2010) for Cell BE processors. All the applications perform single-precision floating-point computations because the IBM OpenCL framework does not support double precision. Thus, we do not show the speedup of EP. Since LC, MRI-Q, RPES, and TPACF fail for an out-of-resource error, abort at run time, or generate an invalid result even though they compile with the IBM XL OpenCL compiler well, we do not show their speedup either. The IBM OpenCL framework also exploits a software-managed cache[20] that supports a smaller cache line size (128B) than that of ours (512B – 16KB). Our cache supports page-level caching and coherence mechanism. Its organization is significantly different from IBM's.

For all applications, the speedup of Coalescing&Buffering is better than or comparable to that of OpenCL-IBM. Since the internal details of the IBM OpenCL runtime and compiler are not publicly available, we do not know the exact reason for the performance difference. Note that we cannot control the optimization performed by IBM OpenCL C compiler that is invoked by the IBM OpenCL framework.

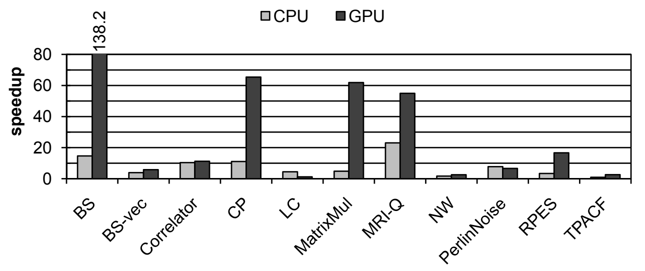


Figure 13: The speedup of the OpenCL applications with multicore CPUs and a GPU.

To understand the performance of the OpenCL applications on different platforms other than Cell BE, Figure 13 shows the speedups of the OpenCL applications on two Intel Xeon X5660 hexa-core processors (CPU) and an NVIDIA Tesla C1060 GPU (GPU). The speedup is obtained over a single x86 core in the Xeon processor. For CPU, the OpenCL framework contained in AMD ATI Stream SDK v2.1[4] is used. For GPU, the OpenCL framework from NVIDIA GPU Computing SDK 3.0[30] is used. Since EP requires a large private memory space for each work-item in the kernel, both of the OpenCL frameworks give a build error for EP. Thus, we do not show its performance in Figure 13.

Overall, our OpenCL framework is quite efficient. In addition, work-item coalescing and preload-poststore buffering are effective optimization techniques for the OpenCL framework targeting accelerator multicores with small local memory. The source code of our OpenCL framework is publicly available at the URL `http://aces.snu.ac.kr`.

9. CONCLUSIONS

In this paper, we present the design and implementation of an OpenCL runtime and OpenCL C source-to-source translator that target heterogeneous accelerator multicore architectures with local memory. The accelerator multicore architecture consists of a general-purpose processor core backed by a deep on-chip cache hierarchy and multiple accelerator cores that have a non-coherent, small internal local store.

Our runtime is based on software-managed caches and consistency protocols. It executes work-items concurrently by switching work-item contexts. To boost performance, the runtime relies on three source-code transformation techniques, work-item coalescing, web-based variable expansion and preload-poststore buffering. These algorithms are implemented in our source-to-source translator. Work-item coalescing is a procedure to serialize the concurrent executions of the work-items in a single work-group in the presence of barriers, and to sequentially execute them on a single accelerator core. Work-item coalescing requires web-based variable expansion to allocate memory for private variables. Preload-poststore buffering is a buffering technique that eliminates software cache access overhead and individual DMA operation overhead by overlapping multiple DMA transfers together.

We have shown the effectiveness of our OpenCL framework, evaluating its performance with a system that consists of two Cell BE processors using 12 OpenCL benchmark applications. Evaluation results show that our approach is effective and promising for the heterogeneous accelerator multicore architectures with a limited size of local memory.

10. REFERENCES

[1] S. Adve and M. Hill. Weak Ordering. In *ISCA'90: Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.

[3] AMD. ATI CTM Guide. `http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf`.

[4] AMD. OpenCL: The Open Standard for Parallel Programming of GPUs and Multi-core CPUs. `http://ati.amd.com/technology/streamcomputing/opencl.html`.

[5] Apple. OpenCL: Taking the graphics processor beyond graphics. `http://images.apple.com/macosx/technology/docs/OpenCL_TB_brief_20090903.pdf`.

[6] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, K. O'brien, and K. O'Brien. A novel asynchronous software cache implementation for the cell/be processor. In *LCPC '07: Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing*, October 2007.

[7] V. Balasundaram and K. Kennedy. A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 41–53, New York, NY, USA, 1989. ACM.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT'08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, October 2008.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous. In *IISWC '09: Proceedings of the IEEE International Symposium on Workload Characterization*, pages 44–54, Oct 2009.

[10] T. Chen, H. Lin, T. Zhang, K. M. O'Brien, and J. K. O'Brien. Orchestrating Data Transfer for the cell/B.E. Processor. In *ICS'08: Proceedings of the 22nd annual International Conference on Supercomputing*, pages 289–298, New York, NY, USA, 2008. ACM.

[11] W.-Y. Chen, C. Iancu, and K. Yelick. Communication Optimizations for Fine-Grained UPC Applications. In *PACT'05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 267–278, Washington, DC, USA, 2005. IEEE Computer Society.

[12] F. Darema. The SPMD Model: Past, Present and Future. *Lecture Notes in Computer Science*, 2131(1):1–1, January 2001.

[13] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine^{TM} architecture. *IBM Systems Journal*, 45(1):59–84, January 2006.

[14] R. S. Engelschall. Portable Multithreading: The Signal Stack Trick For User-Space Thread Creation. In *Proceedings of 2000 USENIX Annual Technical Conference*, pages 155–164, June 2000.

[15] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[16] M. González, N. Vujic, X. Martorell, E. Ayguadé, A. E. Eichenberger, T. Chen, Z. Sura, T. Zhang, K. O'Brien, and K. M. O'Brien. Hybrid Access-specific Software Cache Techniques for the Cell BE Architecture. In *PACT'08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 292–302, October 2008.

[17] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, March/April 2006.

[18] C. Iancu, P. Husbands, and P. Hargrove. HUNTing the Overlap. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 279–290, Washington, DC, USA, 2005. IEEE Computer Society.

[19] IBM. OpenCL Development Kit for Linux on Power. `http://www.alphaworks.ibm.com/tech/opencl`.

[20] IBM. *Software Development Kit for Multicore Acceleration version 3.1, Programmer's Guide*. IBM, 2008. `http://www.ibm.com/developerworks/power/cell/`.

[21] IBM, Sony, and Toshiba. *Cell Broadband Engine Architecture*. IBM, 2009. `http://www.ibm.com/developerworks/power/cell/`.

[22] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference*, pages 115–131, January 1994.

[23] Khronos OpenCL Working Group. *The OpenCL Specification Version 1.0*. Khronos Group, 2009. `http://www.khronos.org/opencl`.

[24] J. Lee, J. Lee, S. Seo, J. Kim, S. Kim, and Z. Sura. COMIC++: A Software SVM System for Heterogeneous Multicore Accelerator Clusters. In *HPCA'10: Proceedings of the 15th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, January 2010.

[25] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han. COMIC: A Coherent Shared Memory Interface for Cell BE. In *PACT'08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 303–314, October 2008.

[26] T. Liu, H. Lin, T. Chen, J. K. O'Brien, and L. Shao. DBDB: Optimizing DMA Transfer for the Cell BE Architecture. In *ICS'09: Proceedings of the 23rd International Conference on Supercomputing*, pages 36–45, New York, NY, USA, 2009. ACM.

[27] LLVM Team. The LLVM Compiler Infrastructure. `http://llvm.org`.

[28] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[29] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[30] NVIDIA. OpenCL for NVIDIA. `http://www.nvidia.com/object/cuda_opencl.html`.

[31] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture*. NVIDIA, June 2008. `http://developer.download.nvidia.com`.

[32] W. Pugh and Omega Project Team. The Omega Project: Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs. `http://www.cs.umd.edu/projects/omega`, 2009.

[33] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson. Programming Model for a Heterogeneous x86 Platform. In *PLDI'09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 431–440, New York, NY, USA, 2009. ACM.

[34] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, , and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics*, 27(3):Article 18, August 2008.

[35] S. Seo, J. Lee, and Z. Sura. Design and Implementation of Software-managed Caches for Multicores with Local Memory. In *HPCA'09:Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pages 55–66, February 2009.

[36] The IMPACT Research Group. Parboil Benchmark Suite. `http://impact.crhc.illinois.edu/parboil.php`, 2009.

[37] R. V. van Nieuwpoort and J. W. Romein. Using many-core hardware to correlate radio astronomy signals. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 440–449, New York, NY, USA, 2009. ACM.

[38] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–166, New York, NY, USA, 2007. ACM.

[39] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1991.