An OpenCL Framework for Homogeneous Manycores with no Hardware Cache Coherence

Jun Lee, Jungwon Kim, Junghyun Kim, Sangmin Seo, and Jaejin Lee
Center for Manycore Programming
School of Computer Science and Engineering
Seoul National University, Seoul, 151-744, Korea
{jun, jungwon, junghyun, sangmin}@aces.snu.ac.kr, jlee@cse.snu.ac.kr
http://aces.snu.ac.kr

Abstract—Recently, Intel has introduced a research prototype manycore processor called the Single-chip Cloud Computer (SCC). The SCC is an experimental processor created by Intel Labs. It contains 48 cores in a single chip and each core has its own L1 and L2 caches without any hardware support for cache coherence. It allows maximum 64GB size of external memory that can be accessed by all cores and each core dynamically maps the external memory into their own address space. In this paper, we introduce the design and implementation of an OpenCL framework (i.e., runtime and compiler) for manycore architectures with no hardware cache coherence. We have found that the OpenCL coherence and consistency model fits well with the SCC architecture. The OpenCL’s weak memory consistency model requires relatively small amount of messages and coherence actions to guarantee coherence and consistency between the memory blocks in the SCC. The dynamic memory mapping mechanism enables our framework to preserve the semantics of the buffer object operations in OpenCL with a small overhead. We have implemented the proposed OpenCL runtime and compiler and evaluate their performance on the SCC with OpenCL applications.

Keywords—Single-chip Cloud Computer, OpenCL, Compilers, Runtime, Cache coherence, Memory consistency

I. INTRODUCTION

In these days, most of processor manufacturers put more cores in a single chip to improve performance and power efficiency. 2, 4, 6, 8, and 12-core homogeneous processors are already common in the market, and most of them support hardware cache coherence to provide a single, coherent shared memory space between cores. Much more cores will be integrated together in a single chip in near future. Although the hardware cache coherence mechanism has been successful so far, it is quite doubtful that this will still work well with future manycore processors.

As the number of cores increases, the on-chip interconnect becomes a major performance bottleneck and takes up a significant amount of power budget. A complicated cache coherence protocol worsens the situation by increasing the amount of messages that go through the interconnect. For this reason, there have been many studies on the relationship between on-chip interconnects and cache coherence protocols for manycores to find an alternative to the current hardware cache coherence mechanism [1], [2], [3], [4]. As one of these efforts, Intel recently introduced the Single-Chip Cloud Computer (SCC). The SCC experimental processor[5], [6] is a 48-core ‘concept vehicle’ created by Intel Labs as a platform for many-core software research. It contains 48 cores in a single chip and each core has its own L1 and L2 caches but there is no hardware cache coherence support. Instead, it has a small but fast local memory called message passing buffer (MPB) to support the message passing programming model. In addition, each core can dynamically map any external main memory blocks to its own address space at any time.

Recently, the Open Computing Language (OpenCL)[7] has been introduced. OpenCL is a framework for building parallel applications that are portable across heterogeneous parallel platforms. 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 defines its own memory hierarchy and memory consistency model to provide software developers with a complete abstraction layer of the target architecture.

In this paper, we introduce the design and implementation of an OpenCL framework (i.e., runtime and compiler) for manycore processors with no hardware cache coherence mechanism, such as the SCC. We have found that OpenCL’s coherence and relaxed memory consistency model fits well with the SCC architecture. The memory model requires relatively small amount of messages exchanged between cores to guarantee coherence and consistency in the SCC. This enables us to efficiently implement the OpenCL framework on top of the SCC’s message passing architecture. Moreover, the SCC’s dynamic memory mapping mechanism enables our framework to easily preserve the semantics of the buffer object operations in OpenCL with a small overhead. It only requires modifying the control registers of each core to transfer memory blocks between different cores without using any expensive memory copy operations, such as message passing and DMA operations.

Except the core that takes the role of the host processor in OpenCL, each core in the SCC is viewed as an individual
OpenCL compute unit in a single OpenCL compute device. In other words, our OpenCL framework can execute an OpenCL program without any modification in the source as long as the program is written for a single compute device, such as a GPU or CPU.

Our OpenCL-C-to-C compiler translates the OpenCL kernel code to the C code that runs on SCC cores. The OpenCL runtime distributes the workload in the kernel to each core based on a technique called symbolic array bound analysis. It also balances the workload and minimizes coherence actions between cores. The OpenCL-C-to-C compiler also translates the OpenCL kernel code to a code that performs a symbolic array bound analysis at run time. The runtime merges the data distributed across each core’s private memory to guarantee memory consistency with a negligible overhead by exploiting the SCC’s dynamic memory mapping mechanism.

We have implemented our OpenCL runtime and OpenCL-C-to-C translator. We evaluate our approach with 9 real OpenCL applications and show its effectiveness. We analyze the characteristics of our OpenCL framework on the SCC and show how it exploits the features of the SCC efficiently. As far as we know, our OpenCL proposal is the first work for building a complete and transparent software layer for the SCC architecture to improve ease of programming and to achieve high performance.

II. BACKGROUND

We briefly describe the SCC architecture and the OpenCL platform in this section.

A. SCC Architecture

The Single-Chip Cloud Computer (SCC)[5], [6] is an experimental homogeneous manycore processor created by Intel as a platform for manycore software research. Figure 1 shows the block diagram of the SCC architecture. It contains 48 IA-32 cores on a single die as a form of 6x8 2D-mesh network of tiles. Each tile contains two cores. Each core has its own private 16KB L1 data and instruction caches and 256KB unified L2 cache, but there is no hardware cache coherence support. Instead, each tile has a local memory buffer, also called as message passing buffer (MPB). Each MPB is 16KB size and any core can access these 24 on-die MPBs.

The MPB is a small but fast memory area that is shared by all cores to enable fast message passing between cores. The operations on the MPB also go through each core’s cache, but coherence between the caches and MPBs is not guaranteed either. To solve this problem, the SCC architecture provides a new instruction called CL1INVMB and a new memory type called MPBT. The MPBT data is cached by the L1 cache only. When a core wants to update a data item in the MPB, it can invalidate the cached copy using the CL1INVMB instruction. The CL1INVMB invalidates all the MPBT cache lines. Since explicit management of MPBs for message passing is quite burdensome, Intel provides an MPI-like message passing interface, called RCCE[8]. RCCE provides basic message passing operations, such as send, receive, broadcast, put, and get. It also provides synchronization operations such as acquire, release, and barrier. The appearance of the interface looks just like the MPI interface, but the internal implementation is based on the shared MPB management and CL1INVMB instruction. Our OpenCL runtime exploits the RCCE library to handle communication between the cores.

There are four memory controllers in the SCC. Each controller supports up to 16GB DDR3 memory. While this implies that a maximum capacity of 64GB memory is provided by the SCC, each core is able to access only 4GB of memory because it is based on the IA-32 architecture. In order to break this limitation, the SCC allows each core to alter its own memory map. Each core has a lookup table, called LUT, which is a set of configuration registers that map the core’s 32-bit physical addresses to the 64GB system memory. Each LUT has 256 entries, and each entry handles a 16MB segment of the core’s 4GB physical address space. An LUT entry can point to a 16MB system memory segment, an MPB, or a memory-mapped configuration register. Each LUT entry can be modified at any time. On an L2 cache miss, the requested 32-bit physical address is translated into a 46-bit system address. The system memory address space consists of 4 different 16GB regions of the external memory, 24 16KB regions of MPBs, and regions for memory mapped configuration registers of each core.

Each core’s LUT is initialized during the bootstrap process, but the memory map can be altered by any core at any time. Consequently, the software can freely map any memory region as a shared or private region. However, sharing the same memory region would cause a serious problem because there is no hardware support for cache coherence. Our OpenCL framework provides a convenient and safe way to exploit this type of architecture efficiently while guaranteeing coherence and consistency.

B. OpenCL Platform Model

The OpenCL platform model (Figure 2) consists of a host processor connected to one or more OpenCL compute devices. A compute device is divided into one or more
compute units (CUs), each of which contains one or more processing elements (PEs).

An OpenCL program consists of two parts: kernels and a host program. The kernels are executed on one or more compute devices. The host program runs on the host processor and enqueues a command to a command-queue that is attached to a compute device. A kernel command executes a kernel on the PEs within the compute device. A memory command controls a buffer object, and a synchronization command enforces an ordering between commands. The OpenCL runtime schedules the enqueued kernel command on the associated compute device and executes the enqueued memory or synchronization command directly.

When a kernel command is enqueued, an abstract index space has been defined. The index space called NDRange is an N-dimensional space, where N is equal to 1, 2, or 3. An NDRange is defined by an N-tuple of integers and specifies the extent of the index space (the dimension and the size). An instance of the kernel executes for each point in this index space. This kernel instance is called a work-item, and is uniquely identified by its global ID (N-tuples) defined by its point in the index space. Each work-item executes the same code but the specific pathway and accessed data can vary.

Figure 2. The OpenCL platform model.

One or more work-items compose a work-group, which provides more coarse-grained decomposition of the index space. Each work-group has a unique work-group ID in the work-group index space and assigns a unique local ID to each work-item within itself. Thus a work-item is identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the PEs in a single CU.

As shown in Figure 2, each work-item running on a PE accesses four distinct memory regions: global, constant, local, and private. The global memory and constant memory are shared by all CUs in a compute device (i.e., all work-items in all work-groups). The constant memory is a region that remains constant during the execution of a kernel. Local memory is shared by all PEs in a CU (i.e., all work-items in a single work-group). The PE private memory is private to each PE (i.e., private to each work-item).

C. Memory Consistency Model

OpenCL defines a relaxed memory consistency model 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. A work-group barrier synchronizes work-items in a single work-group, and a command-queue barrier command synchronizes commands in a single command-queue. To synchronize commands in different command-queues, events are used. Between work-groups during kernel execution, there is no synchronization mechanism available in OpenCL.

III. THE OPENCL FRAMEWORK

In this section, we describe the design and implementation of our OpenCL framework for the SCC.

A. OpenCL Platform Model and SCC

Before elaborating on our OpenCL framework, we first explain how the OpenCL platform model is mapped to the SCC. Table I summarizes the mapping between the OpenCL platform and the SCC. A single core in the SCC becomes the host processor. We will call the host processor as a host core in this paper. In our approach, the OpenCL runtime thread runs on the host core where the OpenCL host program also runs. Our OpenCL runtime maps the entire SCC but the host core as an OpenCL compute device. Each core becomes a CU and there are a total of 47 CUs (SCC cores).

Table I

<table>
<thead>
<tr>
<th>Mapping Between the OpenCL Platform Model and The SCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCL Platform</td>
</tr>
<tr>
<td>SCC</td>
</tr>
<tr>
<td>Host processor</td>
</tr>
<tr>
<td>Main memory</td>
</tr>
<tr>
<td>Compute device</td>
</tr>
<tr>
<td>Global/constant memory</td>
</tr>
<tr>
<td>CUs</td>
</tr>
<tr>
<td>Local memory</td>
</tr>
<tr>
<td>Private memory</td>
</tr>
<tr>
<td>Global/constant memory cache</td>
</tr>
</tbody>
</table>

We assume each core in the SCC has a distinct private system memory region (PSMR) that consists of multiple system memory sections. Sections in the host core’s PSMR are configured as the sections of the OpenCL main memory, global memory, and constant memory to reduce the data transfer overhead between the main memory and global/constant memory.

A space in each CU core’s PSMR becomes the CU’s local memory. Since our OpenCL runtime emulates PEs with the associated CU core, the private memory of each PE is mapped to a space in the CU core’s PSMR. The remaining
space of the CU core’s PSMR is used for buffering data to maintain coherence and consistency.

The OpenCL global/constant cache is implemented by each CU core’s caches and the coherence/consistency management protocol provided by the runtime.

```
__kernel void foo(__global float *A, __global float *B, __global float *C, int P, int Q)
{
    int id = get_global_id(0);
    C[id] = A[id] + B[P+Q*id];
}
```

```
void foo(__global float *A, __global float *B, int id);
{
    id = get_global_id(0);
    (A-memOffA)[id] = (B-memOffB)[P+Q*id];
}
```

Figure 3. (a) An OpenCL kernel code. (b) The C code generated by our OpenCL-C-to-C translator.

B. Emulating PEs

In OpenCL, a CU executes a set of work-groups, and work-items in a work-group are executed by one or more PEs in the same CU concurrently. Our OpenCL runtime emulates the PEs using a kernel transformation technique, called work-item coalescing[9], [10], provided by our OpenCL-C-to-C translator. The work-item coalescing technique makes the CU core (i.e., an SCC core) executes each work-item in the work-group one by one sequentially using a loop that iterates over the local work-item index space. Figure 3 (a) shows an OpenCL kernel code and Figure 3 (b) shows the C code generated by our OpenCL-C-to-C translator. The triply nested loop in Figure 3 (b) is such a loop after the work-item coalescing technique has been applied. Since the index space of the kernel program in Figure 3 (a) is one dimensional, __local_size[2] and __local_size[1] in Figure 3 (b) will be one.

When there are work-group barriers contained in the kernel (i.e., each work-item in the work-group needs to wait at the barrier until others arrive at the barrier), the work-item coalescing technique divide the code into work-item coalescing regions (WCRs)[9]. A WCR is a maximal code region that does not contain any barrier. Since a work-item private variable whose value is defined in one WCR and used in another needs a separate location for each work-item to transfer the variable’s value between different WCRs, the variable expansion technique[9] is applied to WCRs. Then, the work-item coalescing technique executes each WCR using a loop that iterates over the local work-item index space. The execution order of WCRs in work-item coalescing preserves the barrier semantics.

C. Workload Distribution

Our OpenCL runtime divides and distributes the kernel workload to each CU core. Since there is no dependence between work-groups in OpenCL, the unit of workload distribution is a work-group. The runtime distributes work-groups in a kernel across CU cores. A set of work-groups that is executed by a CU core is called a work-group assignment. The runtime also distributes data in the global memory that are accessed by the work-group assignments.

To distribute the workload, the runtime partitions the work-group index space into disjoint work-group assignments in a well-balanced and well-formed manner. The resulting partition is well balanced in the number of work-groups in each assignment to make all CU cores busy. A partition is well formed if the work-group IDs in each assignment are contiguous in each dimension of the work-group ID. Figure 4 shows some examples of well-balanced and well-formed work-group partitions.

The runtime has a built-in partition table that is indexed with the dimension (1, 2, or 3) of the work-group index space. Note that the dimension of the work-group index space will be known at run time. Each table entry records all well-balanced and well-formed partitions a priori. When the dimension of the work-group index space is known at run time, the runtime selects a partition in the table that minimizes the memory management overhead. A buffer object is a memory object that stores a linear collection of bytes. An array is typically allocated as a buffer object in OpenCL and accessed by a kernel using a pointer. The host program can create the buffer object in the OpenCL global memory and pass the pointer as an argument to the kernel.

If the IDs of all work-groups in the work-group assignment of a CU are contiguous in each dimension, then the host launches the kernel only once for the entire work-groups in the assignment. However, if they are non-contiguous (i.e., the partition is not well formed), the host needs to launch the same kernel for each set of work-groups with continuous IDs in the assignment. That is, the kernel is launched multiple times. In addition, a non-well-formed
partition typically introduces multiple non-contiguous buffer access ranges for a work-group assignment in the partition. These non-contiguous buffer access ranges incur much more memory management and transfer overhead.

To select an optimal partition that has a minimal memory management overhead, our OpenCL framework relies on a symbolic array bound analysis [11], [12]. This gives the maximal buffer access ranges for each work-group assignment in each candidate partition in the selected table entry.

### D. Symbolic Array Bound Analysis

Our OpenCL-C-to-C translator performs the symbolic array bound analysis and generates code for the runtime to determine maximal buffer access ranges. It conservatively identifies access ranges of buffers (i.e., arrays) that are accessed by each work-group assignment. Note that each work-item of an OpenCL kernel typically accesses a buffer using its global ID, work-group ID, and local ID in an SPMD[13] manner. The result provides symbolic lower and upper bounds for each array references in the kernel. Our OpenCL runtime exploits this information by resolving the symbols at run time when the size of index space is determined.

<table>
<thead>
<tr>
<th>Array</th>
<th>Read Range</th>
<th>Write Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>([G_L, G_U])</td>
<td>([G_L, G_U])</td>
</tr>
<tr>
<td>B</td>
<td>([P+Q^3G_L, P+Q^3G_U]) or ([P+Q^3G_U, P+Q^3G_L])</td>
<td>([G_L, G_U])</td>
</tr>
<tr>
<td>C</td>
<td>([G_L, G_U])</td>
<td>([G_L, G_U])</td>
</tr>
</tbody>
</table>

For example, Table II shows the result of symbolic array bound analysis of the kernel in Figure 3 (a). \(G_L\) and \(G_U\) are symbolic lower and upper bounds of \(g_{\text{global}}(0)\) in Figure 3 (a), respectively. Read Range shows the maximal array regions that may be read by any execution of the kernel. Similarly, Write Range shows the maximal array regions that may be modified by any execution of the kernel. These regions are defined by unresolved symbols, such as \(G_L, G_U, P\) and \(Q\). The values of these symbols are defined when the host program enqueues the kernel command into a command queue at run time. Thus, just before the kernel is issued to a compute device, our runtime resolves the values of these symbols by running the code generated by the OpenCL-C-to-C translator and obtains buffer access ranges for the work-group assignment for each CU core. With this buffer access range information, the runtime selects a partition that has a minimal memory management cost.

Figure 5 shows the code generated by our OpenCL-C-to-C translator for the kernel in Figure 3 (a) after the symbolic analysis. The runtime runs this code for each work-group assignment to obtain its buffer access ranges. Each buffer (i.e., array) argument in the kernel is replaced with a pointer to a RANGE data item. A RANGE data item contains four bound variables to represent buffer access range: lower and upper bounds for read and write. Set_read_range and set_write_range set the read and write bound variables for the runtime. The \(\text{GLOBAL}_{-}\text{ID}_{-}L(N)\) and \(\text{GLOBAL}_{-}\text{ID}_{-}U(N)\) are macros for calculating the values of lower bound and upper bound of \(\text{get}_{-}\text{global}_{-}id(N)\) at run time using the index space determined by each work-group assignment. The macros use the run-time information, such as the ID of the first work-group in the work-group assignment (\(FWG\)), the number of work-groups in each work-group assignment (\(NWG\)), the number of work-items in a work-group (\(WGS\)), and the number of work-groups in the entire kernel index space (\(ENWG\)). Since the dimension of the index space can be one, two, or three, each of these values is three dimensional array.

To perform the symbolic analysis, our translator uses program slicing techniques[14], [15]. First, it inlines user-defined function calls in the kernel. Then it performs constant propagation, loop invariant and induction variable recognition and reaching definitions analysis for the kernel. After performing these analyses, the translator extracts the index computation slice for each array reference. Then the translator checks if the slice satisfies all the following conditions:

- Each index of the array reference is an affine function in global ID, work-group ID, and local ID.
- When the array reference is not contained in a loop, any variable used in its indices does not have more than one reaching definition.
- If the array reference is contained in a well-behaved affine loop, each of its index is an affine function in induction variables of the loops, global ID, work-group ID, and local ID.

A well-behaved loop[16] 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 \( \text{stmt} \) contains no assignment to \( i \). We say a well-behaved loop is \textit{affine in the global ID, work-group ID, and local ID} if both of the lower bound and upper bound of the loop index variable are affine functions in the global ID, work-group ID, and local ID. If the index subscript of array reference satisfies this condition, the translator iteratively applies forward substitution to the slice until there is no more forward-substitutable variable. If a kernel contains an array reference that does not satisfy the above conditions, our runtime conservatively assumes that the access range of each CU core is the entire buffer.

### E. Identifying an Optimal Partition

To select an optimal partition, we estimate the memory management overhead for each partition using a cost model that is based on the buffer access range information for each kernel.

When a CU core writes to a buffer and if the buffer access range of work-groups assigned to the CU core overlaps with that of work-groups assigned to another core, there is a memory consistency problem. Each CU core has to copy the global memory region of the buffer access range to its own private space and updates the local copy to avoid the problem. When the host reads the buffer from the device after executing the kernel, it \textit{merges} the updates done in the local copies and updates the original copy of the buffer.

Otherwise, the writer CU core just alters its LUT and calls \texttt{mmap()} function to map the global memory region of the buffer access range to its own address space. This incurs only the LUT modification and \texttt{mmap()} call overhead. There is no memory copy or message passing overhead involved. Moreover, at the end of a kernel execution, the CU core has to do nothing but calling \texttt{munmap()} function to unmap the region from its own address space. The LUT modification and \texttt{mmap()}/\texttt{munmap()} call overhead is small enough to be negligible.

Our memory management overhead \((\text{Cost}_{\text{manage}}(b)) \) for a buffer \( b \) basically consists of the memory copy overhead \( \text{Cost}_{\text{copy}}(b) \) and the merge overhead \( \text{Cost}_{\text{merge}}(b) \) due to the overlapping buffer access ranges.

Consider the case in Figure 6. Each CU writes to a buffer \( b \). A gray bar for each CU core is the buffer write range of the work-groups assigned to the CU core. We divide the buffer into disjoint segment intervals \((\text{SI}_i)\). A segment \((16\text{MB} \text{ in the SCC})\) is the unit of mapping between the 64GB system memory address space and the CU core’s 32-bit physical address space in the SCC. A part of the segment cannot be mapped separately. Using the lower and upper bounds of the segment intervals and the buffer access range of each CU core, we also divide the buffer into disjoint overlapping intervals \((\text{OI}_i)\). For each segment interval, we record the number of CUs that have the segment interval in their buffer access range. The same thing is true for each overlapping interval. When an overlapping interval’s CU count is more than one and at least one of the CUs writes to the interval, we say that the overlapping interval is \textit{write-shared}. When a segment interval overlaps with a write-shared overlapping interval, the segment interval is also said to be write-shared.

The memory copy overhead \( \text{Cost}_{\text{copy}}(b) \) for a buffer \( b \) is given by,

\[
\text{Cost}_{\text{copy}}(b) = \sum_{\text{OI} \in S} \text{Count}(\text{OI}) \cdot \text{Length}(\text{OI}) \cdot \alpha
\]

where \( S \) is the set write-shared overlapping intervals for \( b \), \( \text{Count}(\text{OI}) \) is the number of CUs that have \( \text{OI} \) in their access range, \( \text{Length}(\text{OI}) \) is the size of \( \text{OI} \) in bytes, and \( \alpha \) is the cost of copying a byte and obtained by measuring the actual cost on the target machine.

In Figure 6, \( \text{OI}_2 \), \( \text{OI}_3 \) and \( \text{OI}_5 \) are write-shared. They need to be copied to each associated CU core’s private address space. Thus, \( \text{Cost}_{\text{copy}}(b) = (2 \cdot \text{Length}(\text{OI}_2) + 2 \cdot \text{Length}(\text{OI}_3) + 2 \cdot \text{Length}(\text{OI}_5)) \cdot \alpha = 36\text{MB} \cdot \alpha \).

When the runtime update the original buffer by merging local copies distributed to CU cores, it compares the local copies word by word with the original buffer. \( \text{Cost}_{\text{merge}}(b) \) is given by,

\[
\text{Cost}_{\text{merge}}(b) = \sum_{\text{OI} \in S} \text{Count}(\text{OI}) \cdot \text{Length}(\text{OI}) \cdot \beta + \text{Length}(\text{OI}) \cdot \alpha
\]

where \( S \) is the set of write-shared overlapping intervals for \( b \), and \( \beta \) is the cost of comparing two words (the size depends on the buffer type) and obtained by measuring the actual cost on the target machine. The first term \( \text{Count}(\text{OI}) \cdot \text{Length}(\text{OI}) \cdot \beta \) is the worst-case word-by-word comparison cost. The second term \( \text{Length}(\text{OI}) \cdot \alpha \) is the update cost for an write-shared overlapping interval. It is conservatively assumed that all words in the overlapping interval are updated. Since the OpenCL memory consistency
model does not guarantee a consistent memory view between different work-groups during the kernel execution, and there is no synchronization between work-groups, we may choose any update as the last update when there are multiple updates to the same location in the global memory. This is the reason why we do not multiply \( \text{Count}(OI) \) to the second term.

In Figure 6, \( \text{Cost}_{\text{merge}}(b) = (2 \cdot \text{Length}(OI) + 2 \cdot \text{Length}(OI) + 2 \cdot \text{Length}(OI)) \cdot \beta + ((\text{Length}(OI) + \text{Length}(OI) + \text{Length}(OI)) \cdot \alpha = 38MB \cdot \beta + 18MB \cdot \alpha. \)

Now, the total memory management cost \( \text{Cost}_{\text{manage}} \) of buffers for a kernel is given by,

\[
\text{Cost}_{\text{manage}} = \sum_{b \in B} (\text{Cost}_{\text{copy}}(b) + \text{Cost}_{\text{merge}}(b))
\]

where \( B \) is the set of buffers that is accessed in a kernel. The runtime computes \( \text{Cost}_{\text{manage}} \) for each partition and selects the partition that has the minimum \( \text{Cost}_{\text{manage}} \). Using this partition, the runtime distributes the work-groups and the buffer chunks to each CU core.

### F. Memory Management

To describe how the runtime manages the memory hierarchy and consistency in OpenCL, we explain what is done by our OpenCL runtime step by step.

**Initialization.** In the initialization phase of the OpenCL runtime, the runtime defines LUT entries to distribute the available system memory to each core. The allocated system memory region to each core is private to each core. For example, when we have a total of 32GB system memory, the runtime allocates a contiguous 2GB of the system memory region to the host core and a contiguous 624MB of the system memory region to each CU core. The OS kernel running on each core uses about 320MB of the allocated system memory and the runtime manages the remaining. The LUT in Figure 7 shows this memory allocation for a CU core. The CU core’s LUT has a total of 256 entries. The LUT is indexed by the 16MB segment number in the 32-bit physical address space. The first field of an LUT entry is the physical address of the 16MB segment in the 32-bit address space. The second field is the 46-bit system memory address of the 16MB segment. An MPB or configuration register can be also mapped by a single LUT entry.

**Buffer object creation.** Whenever the host program sends an OpenCL buffer object creation request to the runtime by invoking the \texttt{clCreateBuffer()} API function, the runtime allocates the buffer in the free space of the host core’s PSMR (note that the OpenCL global memory is mapped to an area in the host core’s PSMR). Since the runtime manages each core’s PSMR with 32-bit physical addresses through the core’s LUT, the runtime uses \texttt{mmap()} and \texttt{munmap()} to obtain the virtual address of the buffer. Then the runtime returns it to the host program. In addition, the runtime obtains the system memory address of the buffer object from the LUT and records it in the data structure of the buffer object for later use.

The host program invokes \texttt{clEnqueueWriteBuffer()} when it wants to write to a buffer object that has been created by \texttt{clCreateBuffer()}. The OpenCL runtime copies the data from the OpenCL main memory (allocated in the host core’s PSMR) to the designated buffer object in the OpenCL global memory (also allocated in the host core’s PSMR).

**Collecting runtime parameters.** By invoking \texttt{clSetKernelArg()}, the host program passes arguments to the kernel. The runtime records this information in a reserved space and passes them to the kernel when the kernel is launched. By invoking \texttt{clEnqueueNDRangeKernel()}, the host program delivers the information (i.e., size and dimension) about the work-item and work-group index spaces of the kernel to the runtime.

**Distributing buffer objects.** When the runtime dequeues a kernel command from the command queue and issues the kernel to CU cores for execution, the runtime knows all information about the kernel that is required to resolve the symbols in the symbolic array bound analysis. It calls the analysis function for the kernel at this point. Based on the buffer access range information obtained from the analysis and the cost model described in the previous section, the runtime selects an optimal work-group partition. The runtime assigns the work-group assignments in the selected partition to CU cores.

Then, the runtime distributes buffer objects that are accessed by CU cores. The host core sends a message to a CU
core $C$ after flushing its own L1 and L2 caches. The message contains the system memory addresses of segment intervals in the buffer access ranges. After receiving the message, $C$ flushes its L1 and L2 caches and alters its own LUT to map segment intervals to its own 32-bit physical address space. Then $C$ sends an acknowledgement message to the host core.

If there is no write-shared overlapping interval in the buffer access ranges of $C$, there is no memory copy. If $C$ modifies a buffer chunk, the updates are directly reflected to the buffer object located in the host core’s PSMR.

If there are write-shared overlapping intervals, the host core sends another message to $C$ after receiving the acknowledgement. The message makes $C$ copy the write-shared overlapping intervals to $C$’s PSMR. After receiving the message, it copies the overlapping intervals from host core’s PSMR (i.e., the global memory) to its own PSMR. After copying, $C$ sends an acknowledgement message to the host core.

When a buffer object is divided and distributed to each core, the addresses of array references in the original kernel code need to be modified. For example, consider the example of Figure 8. The black area in the host core’s buffer object is the access range of the work-group assignment that is assigned to a CU core. By invoking the `mmap()` system call with the 32-bit physical address of the buffer chunk obtained from the LUT, the CU core obtains the virtual address $A$ of the buffer object. However, if the runtime passes this virtual address to the kernel code, then an array reference $A[20]$ in the host core’s buffer object becomes a reference $A[0]$ in the CU core. To make a reference $A[20]$ in the CU core’s kernel access the location of $A[0]$ in the CU core’s private memory, we need to shift the starting address of the array in the CU’s kernel code by `memOffset` (=20 in this example). Thus, a reference $A[1d]$ in the original OpenCL kernel becomes $(A-memOffset)[1d]$ in the CU’s kernel. The `memOffset` is passed to the CU core’s kernel as an argument. A sample kernel code for a CU core is shown in Figure 3(b).

After receiving the acknowledgement message from all the CU cores, the host core sends a message to each CU core to launch the kernel for execution. Each CU core notifies the host core when it finishes executing the assigned work-groups.

**Merging local updates.** After executing a kernel, the host program typically invokes `clEnqueueReadBuffer()` to enqueue a memory command that copies the result from the OpenCL global memory to the OpenCL main memory. The buffer object in the host core’s PSMR (i.e., the global memory) is either consistent or inconsistent to the result of the kernel execution. When there is a write-shared interval, the associated distributed buffer chunks in the PSMR of CU cores are inconsistent to the original buffer chunk in the global memory. Otherwise, the buffer chunk in the global memory is consistent to the result of kernel execution.

To obtain correct result, the runtime checks for write-shared overlapping intervals when `clEnqueueReadBuffer()` is invoked. If there is no write-shared interval for a CU, the host core just makes the CU core flush its L1 and L2 caches. Note that no write-shared interval implies that the CU core directly updates the original buffer located in the host core’s PSMR exploiting the dynamic memory mapping mechanism in the SCC.

Otherwise, a set (say $P$) of CU cores have write-shared overlapping intervals. First, the host core flushes its own caches and makes the CU cores in $P$ flush their caches. The host core modifies its LUT to map the write-shared segment intervals. Then, the host core scans the entire buffer that contains write-shared overlapping intervals. When it meets a write-shared overlapping interval, it compares word by word (i.e., the size depends on the buffer type) the contents of the interval from different CU cores to the original buffer. If a different word is found, the host core updates this word to the original buffer in its PSMR. As mentioned before, since the OpenCL memory consistency model does not guarantee a consistent memory view between different work-groups during the kernel execution, and there is no ordering enforced by synchronization between work-groups, we may choose any update as the last update when there are multiple updates to the same location. Thus, the comparison process for a word is stopped when the first updated value for the word is detected. To perform this scan, the host core needs at least one LUT entry available to read the segments in a CU core. Figure 9 shows an example of this process.

**IV. Evaluation**

This section describes the evaluation results of the SCC with our OpenCL framework. To elicit the scalability of our approach, we compare the scalability of the SCC to that of a multicore system with four 12-core chip-multiprocessors.

**A. Target Machine**

The SCC core is a 32-bit P54C Pentium design that has been altered to increase the L1 data and instruction cache size to 16KB each. The ISA has been extended with a new CL1INVMB instruction, as described in Section II. There are 48 cores in one SCC chip. Although the SCC has multiple
frequency domains and the frequency of each domain can be dynamically varied, we fixed the frequency of all cores at 533MHz in this experiment. Each core has its own private 256KB L2 cache, and each tile has 16KB MPB. The SCC system has 32GB of system memory, 8GB for each memory controller for this experiment. This system memory can be shared by all cores using LUTs.

Each core runs a Linux OS that supports virtual memory but does not support the paging mechanism to hard disc. We use the RCCE library[8] provided by Intel to implement message passing mechanism between cores in our OpenCL runtime. Since the P54C Pentium core does not provide an L2 cache flush instruction, we flush caches by reading a contiguous space of 256KB memory. This takes about 600ms.

The counterpart of the SCC for the comparison is a multicore system with four 2.2 GHz 12-core AMD Opteron processors. Each core has its own L1 data and instruction caches (64KB each) and 512KB L2 cache. The 12MB L3 cache is shared between all cores, and coherence between L1 and L2 caches is supported by the hardware.

B. Benchmark Applications

We evaluate our approach with nine OpenCL applications from various sources: AMD[17], NAS[18], NVIDIA[19], Parboil[20], and PARSEC[21]. The characteristics of the applications are summarized in Table V. Only EP and TPACF contain some array references that do not satisfy the symbolic array bound analysis conditions. In this case, our runtime assumes that each work-group assignment in the partition accesses the entire buffer for the array references.

Our OpenCL runtime uses the RCCE library[8] for basic communication, such as passing kernel arguments. Table III shows the cost of the RCCE_send and RCCE_recv pair for different message sizes. Our runtime never uses messages whose size is over 128 bytes.

When the runtime distributes buffer chunks, it exploits `mmap()` instead of the RCCE library. As you see in Table IV, using the `mmap()` function is three orders of magnitude more efficient than using the RCCE library for the buffer chunk distribution.

Table III

<table>
<thead>
<tr>
<th>Size</th>
<th>8 Bytes</th>
<th>16 Bytes</th>
<th>32 Bytes</th>
<th>64 Bytes</th>
<th>128 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>9 μs</td>
<td>9 μs</td>
<td>9 μs</td>
<td>10 μs</td>
<td>14 μs</td>
</tr>
</tbody>
</table>

Table IV

<table>
<thead>
<tr>
<th>Size</th>
<th>16 MB</th>
<th>32 MB</th>
<th>64 MB</th>
<th>128 MB</th>
<th>256 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCCE</td>
<td>1.92 s</td>
<td>3.84 s</td>
<td>7.68 s</td>
<td>15.36 s</td>
<td>30.72 s</td>
</tr>
<tr>
<td><code>mmap</code></td>
<td>2.23 ms</td>
<td>3.37 ms</td>
<td>5.36 ms</td>
<td>9.58 ms</td>
<td>18.24 ms</td>
</tr>
</tbody>
</table>

C. Communication Costs

As shown in the Table V, repetitive kernel executions make the poor scalability of the RPES. The ideal speedup of RPES with 48 cores is 24 that is computed using the Amdahl’s law. Since at every kernel launch, the runtime thread incurs a non-negligible overhead on distributing workload, such as sending messages that contain kernel arguments and system memory addresses of the buffer objects. A total of 71 kernel launches repeatedly generates this overhead, and the number of cores increases, this fact manifests itself in the speedup. Table VI also explains why the RPES does not scale well. It shows the ratio of communication overhead to the total execution time. Since the communication overhead dominates the execution time of RPES as the number of cores increases, it does not scale well.

The runtime overhead, such as the symbolic analysis, local-copy merge process for consistency, and communication overhead, are sensitive to the number of CU cores. However, as shown in column F and G in Table V, the ratio

D. Effect of Partitioning

As we discussed in the Section III, the existence of write-shared overlapping intervals is critical to the performance of our OpenCL framework. If the runtime selects a suboptimal partition that has a bigger memory management cost than an optimal partition, the memory copy and merge overhead will degrade the performance. For example, MatrixMul and CP have large write-shared overlapping intervals with a worst-case partition. An optimal partition for each of them does not have any write-shared overlapping interval. We see the effect of the optimal partition in Figure 10. It shows the speedup with 2, 4, 8, 16, 24, and 48 cores over a single core for both cases: optimal partition (Best) and worst-case partition (Worst). There is a significant difference between them when the number of cores increases.

E. Scalability

Figure 11 compares the speedup of each application over a single CU core for the SCC with our framework (SCC) and over a single AMD Opteron core for the 48-core AMD Opteron machine with the OpenCL framework available from AMD (Opteron)[17]. We vary the number of cores from 2 to 48. All applications but TPACF scale well on the SCC with our approach when compared to the Opteron machine.

As shown in the Table V, repetitive kernel executions make the poor scalability of the RPES. The ideal speedup of RPES with 48 cores is 24 that is computed using the Amdahl’s law. Since at every kernel launch, the runtime thread incurs a non-negligible overhead on distributing workload, such as sending messages that contain kernel arguments and system memory addresses of the buffer objects. A total of 71 kernel launches repeatedly generates this overhead, and the number of cores increases, this fact manifests itself in the speedup. Table VI also explains why the RPES does not scale well. It shows the ratio of communication overhead to the total execution time. Since the communication overhead dominates the execution time of RPES as the number of cores increases, it does not scale well.

The runtime overhead, such as the symbolic analysis, local-copy merge process for consistency, and communication overhead, are sensitive to the number of CU cores. However, as shown in column F and G in Table V, the ratio

Figure 10. Performance of different partitions.
Table V
CHARACTERISTICS OF THE OPENCL APPLICATIONS

<table>
<thead>
<tr>
<th>Application</th>
<th>Source</th>
<th>Description</th>
<th>Input</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinomialOption</td>
<td>AMD</td>
<td>Binomial option pricing</td>
<td>45120 samples</td>
<td>1.4MB</td>
<td>1</td>
<td>1</td>
<td>o</td>
<td>99.97%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Blackscholes</td>
<td>PARSEC</td>
<td>Black-Scholes PDE</td>
<td>7219200 options, 1000 iters</td>
<td>170MB</td>
<td>1</td>
<td>1</td>
<td>o</td>
<td>99.83%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>CP</td>
<td>Parboil</td>
<td>Coulombic potential</td>
<td>9024x9024, 1000 atoms</td>
<td>311MB</td>
<td>1</td>
<td>1</td>
<td>o</td>
<td>99.63%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>EP</td>
<td>PARSEC</td>
<td>Black-Scholes PDE</td>
<td>Class B</td>
<td>1.5MB</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>99.98%</td>
<td>0.01%</td>
<td>0.64%</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>NVIDIA</td>
<td>Matrix multiplication</td>
<td>4512x4512</td>
<td>233MB</td>
<td>1</td>
<td>1</td>
<td>o</td>
<td>99.76%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>MRI-Q</td>
<td>Parboil</td>
<td>Magnetic resonance imaging Q</td>
<td>Large</td>
<td>5MB</td>
<td>2</td>
<td>1, 2</td>
<td>o</td>
<td>99.96%</td>
<td>0.01%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Nbody</td>
<td>NVIDIA</td>
<td>N-Body simulation</td>
<td>48128 bodies</td>
<td>3MB</td>
<td>1</td>
<td>1</td>
<td>o</td>
<td>99.90%</td>
<td>0.01%</td>
<td>0.00%</td>
</tr>
<tr>
<td>RPES</td>
<td>Parboil</td>
<td>Kys polynomial equation solver</td>
<td>Default</td>
<td>79.4MB</td>
<td>2</td>
<td>1, 17</td>
<td>o</td>
<td>97.80%</td>
<td>0.19%</td>
<td>0.00%</td>
</tr>
<tr>
<td>TPACF</td>
<td>Parboil</td>
<td>Two-point angular correlation function</td>
<td>Default</td>
<td>9.5MB</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>99.95%</td>
<td>0.01%</td>
<td>0.32%</td>
</tr>
</tbody>
</table>

A: total data size in the device global memory, B: # of kernels, C: # of kernel executions, D: satisfying symbolic analysis conditions, E: % seq. exec. time of kernels, F: % exec. time of symbolic analysis run for 48 cores, G: % exec. time of merge process for 48 cores.

Figure 11. The speedup comparison between the SCC and the Opteron system. The speedup is obtained over a single core.

V. RELATED WORK

There have been many studies done on the scalable hardware cache coherence mechanisms for chip-multicore processors[1], [2], [3], [4]. These studies focus on the problem of how to improve the scalability and power consumption of the hardware cache coherence mechanism itself. There are very few studies performed on non-cache-coherent...
homogeneous multicore architectures. To our knowledge, our OpenCL proposal is the first work for building a complete and transparent software layer for the SCC architecture to improve ease of programming and to achieve high performance.

Saha et al.[22] propose a programming model for a heterogeneous x86 platform. The target architecture consists of a general-purpose x86 CPU and a Larrabee processor with 24 cores[23]. The two processors are connected to a PCI express bus. They provide a partly shared memory model between all cores in the system. Since their method relies on the hardware cache coherent mechanism, it is not applicable to our non-cache-coherent target architecture.

Gummaraju et al.[24] present their OpenCL platform that is capable of exploiting multicore CPUs. Their approach is similar to our work because they combine multiple parallel threads into one loop to emulate PEs with a single CPU core. They heavily rely on setjmp() and longjmp() to solve the work-group barrier problem, and choose a different way to reduce the context switching overhead between work-items. They implement their own light-weight setjmp() and longjmp() to reduce the context switching overhead, while our runtime relies on the work-item coalescing technique. Moreover, since their runtime also depends on the hardware cache coherence support, it is not applicable to our non-cache-coherent target architecture.

The Cell BE processor[25] has some similarity with the SCC, but it is a heterogeneous multicore processor. The SPEs have their own private local stores that are not coherent to the main memory. Instead, the DMA operations on the local stores and some communication mechanisms are provided. Consistency of the local stores should be fully managed by the programmer or the software. Lee et al.[9] propose an OpenCL framework for the Cell BE processor. They propose work-item coalescing technique and variable expansion technique to emulate the PEs on an SPE. Instead of directly managing coherence and consistency of the local stores, they exploit software caches to guarantee coherence and consistency. They also propose a buffering technique that exploits preloading.

Stratton et al.[10] propose a compiler technique to emulate parallel work units in a single processor. Their compiler technique translates a CUDA kernel program to the code that runs on homogeneous multicore processors. They propose a similar technique to the work-item coalescing technique to emulate parallel work units. Their compiler divides a parallel region by barriers and packs a chunk of parallel work units in a loop. They also expand the local variables to distinguish them using the thread id.

Kim et al.[26] presents an OpenCL framework that exploits multiple GPU devices and automatically builds a single GPU image on top of them. Their runtime manages virtual buffers in the host main memory and copies them to each device memory when required. They analyze the buffer access ranges using a sampling technique at run time. Using the buffer access ranges, they tries to select an optimal work-group partition that minimizes data transfer cost through the PCI-E bus between the host and GPUs. Our approach relies on a symbolic array bound analysis to identify buffer access ranges and tries to minimize the consistency management overhead in addition to the memory copy overhead to select an optimal partition.

VI. CONCLUSIONS

In this paper, we describe the design and implementation of the OpenCL framework for non-cache-coherent homogeneous manycore processors, such as Intel SCC. We find that the OpenCL parallel programming model can be an ideal software layer for such homogeneous manycores because of its relaxed memory consistency and data parallel programming model. We do not have any difficulties to provide the standard OpenCL API functions and the standard OpenCL applications do not need to be modified to exploit the underlying message passing architecture.

Our OpenCL runtime deploys each SCC core as a CU of the OpenCL platform model. To emulate PEs in a single core efficiently, our OpenCL-C-to-C compiler applies the work-item coalescing and variable expansion technique to the kernels. The runtime exploits the SCC’s dynamic memory mapping mechanism together with the symbolic array bound analysis technique to optimally distribute the kernel workload between CU cores. When the distributed buffer chunks are overlapped each other and at least one of them is for write, the runtime compares the values of the distributed memory chunks to eliminate any inconsistencies after kernel execution.

With nine OpenCL applications, we compare the scalability of our framework on the SCC with a 48-core, cache-coherent commodity multicore system. Our experimental results show that the SCC with our OpenCL framework achieves better scalability than or comparable scalability to that of the commodity manycore system. This partly justifies that the effectiveness of the non-cache-coherent homogeneous manycore processors and appropriateness of the OpenCL framework to such an architecture to achieve high performance.

ACKNOWLEDGMENT

This work was supported in part by grant 2009-0081569 (Creative Research Initiatives: Center for Manycore Programming) from the National Research Foundation of Korea. This work was also supported in part by the Ministry of Education, Science and Technology of Korea under the BK21 Project. ICT at Seoul National University provided research facilities for this study.
REFERENCES


[18] NASA Advanced Supercomputing Division, “NAS Parallel Benchmarks.”


