MOCL: An Efficient OpenCL Implementation for the Matrix-2000 Architecture

ABSTRACT

This paper presents the design and implementation of an Open Computing Language (OpenCL) framework for the Matrix-2000 many-core architecture. This architecture is designed to replace the Intel XeonPhi accelerators of the TianHe-2 supercomputer. We share our experience and insights on how to design an effective OpenCL system for this new hardware accelerator. We propose a set of new analysis and optimizations to unlock the potential of the hardware. We extensively evaluate our approach using a wide range of OpenCL benchmarks on a single and multiple computing nodes. We present our design choices and provide guidance on how to optimize code on the new Matrix-2000 architecture.

KEYWORDS

Heterogeneous Computing; OpenCL; Runtime and Compiler Optimization

ACM Reference format:


1 INTRODUCTION

TianHe-2A (TH-2A) is an upgrade of the leading TOP500 high-performance-computing (HPC) system TianHe-2 (TH-2) [11]. The most significant enhancement of TH-2A over its predecessor is replacing the Intel Xeon Phi (Knights Corner) accelerators with a proprietary accelerator called Matrix-2000. The fully upgraded system has a total of 4,981,760 cores (with 92% of the cores are provided by Matrix-2000) and 3.4 PB primary memory, and reaches a theoretical peak performance of 94.97 PFlops/s, doubling the theoretical peak performance of the former TianHe-2 system 1.

While the Matrix-2000 accelerator provides the potential for higher performance, its potential can only be realized if the software can make effective use of it. To unlock the hardware potential, we need to provide an efficient programming model. We believe that OpenCL is a good candidate for this purpose. This is because it is emerging as a standard for heterogeneous computing, and allows the same code to be executed across a variety of processors. Since existing OpenCL frameworks mainly target CPUs and GPUs, they are not directly applicable to Matrix-2000 [1, 2, 7, 14, 15, 18, 20]. Providing an efficient OpenCL implementation for Matrix-2000 is unique in that the hardware architecture differs from a many-core GPU with a smaller number of cores and runs a lightweight operating system. To exploit the hardware, we have to develop a compiler to translate kernels into target-specific binaries and provide a runtime to manage task dispatching and data communication between the host and the accelerator.

This paper presents the design and implementation of MOCL, an OpenCL programming interface for Matrix-2000. MOCL consists of two main components: a kernel compiler and a runtime. The kernel compiler is built upon the LLVM compiler infrastructure [19]. It translates each OpenCL kernel to an executable binary to run on Matrix-2000. At the core of the compiler is a set of compiling passes to translate an OpenCL work-group 2 into a work-item loop. We map distinct work-item loops to different hardware threads where the work-items in the same group are executed by a single thread to run in a serial manner. We present a push-based task dispatching strategy to distribute work-item loops to hardware threads. Our mapping strategy is different from many OpenCL implementations which execute work-items within a group in parallel. Given that each supernode of an Matrix-2000 has 32 cores instead of hundreds of cores provided by modern GPUs, by mapping a work-group to run on a hardware core (so that we can have up to 32 work-group running in parallel), our strategy avoids the overhead of scheduling thousands of work-items. As we will show later in the paper, this strategy leads to better performance compared to the conventional OpenCL mapping strategies. On top of this, we propose a set of optimization techniques including lock-free atomics to exploit the hardware design.

We evaluate our approach by applying it to 70 OpenCL benchmarks from five well-established benchmark suites. We test the performance of our OpenCL implementation on both a single and multiple Xeon-Matrix2000 nodes of the upgraded TH-2A system. We compare various design choices and show that the chosen ones lead to good performance. We show that our approach, in combination of MPI for cross-node communication, provides good scalability across multiple computing nodes. We provide extensive discussions on how to design OpenCL programs on the Matrix-2000 architecture, offering useful insights on how to write efficient code for this unique accelerator.

The main contribution of this paper is on sharing the experience and insights of designing an effective OpenCL framework for the Matrix-2000 architecture. Our OpenCL implementation has now been deployed to the upgraded TH-2A supercomputer and is ready to be made available to the public.

---

1As of January 2018, TianHe-2 is ranked as the second faster HPC system with a peak performance of 54.9 petalops in the fiftieth TOP500 list: https://www.top500.org/

2An OpenCL work-group is a collection of work-items that can execute on a single compute unit. Here, a work-item is an invoke of the kernel on a given input.
2 BACKGROUND

In this section, we provide an overview of the OpenCL programming interface and the Matrix-2000 architecture.

2.1 Open Computing Language

Open Computing Language (OpenCL) is a standardized programming model for heterogeneous computing [17]. OpenCL uses a generic platform model comprising a host and one or several devices, which are seen as the computation engines. They might be central processing units (CPUs) or “accelerators” such as Matrix-2000, attached to a host processor (a CPU). Devices have multiple processing elements (PEs), further grouped into several compute units (CUs), a global memory and local memories.

An OpenCL program has two parts: kernels that execute on one or more devices, and a host program that executes on the host (typically a traditional CPU). The host program defines the contexts for the kernels and manages their execution, while the computational task is coded into kernel functions. When a kernel is submitted to a device for execution, an index space of work-items (instances of the kernel) is defined. Work-items, grouped in work-groups, are executed on the processing elements of the device in a lock-step fashion. Each work-item has its own private memory space, and can share data via the local memory with the other work-items in the same work-group. All work-items can access the global memory, and/or the constant memory.

2.2 Matrix-2000

Figure 1(a) gives a high-level view of Matrix-2000. A Matrix-2000 processor has 128 computing cores running at 1.2 GHz, offering a peak performance of 2.4576 TFlop/s. Computing cores are grouped into four supernodes (SNs), 32 cores per SN. A SN is further broken down into clusters with four cores per cluster, and cores within a cluster share a coherent data cache. The SNs are connected through a scalable on-chip communication network. Each computing core is an in-order RISC core with a 12-stage pipeline. The core is a 256-bit vector instruction set architecture with two 256-bit vector functional units.

The topology of the network on chip (NoC) within a SN is a 4×2 mesh. Figure 1(b) shows the basic network structure with a total of eight routers in the NoC. A cluster and a directory control unit (DCU) are pinned to a router. Two DDR4-2400 memory control units (MCUs) are integrated into each SN and SNs communicate through a Fast Interconnect Transport (FIT) port.

At the system’s level, each node of TH-2A has two Intel Ivy Bridge CPUs with 64GB of DDR3 RAM, and two Matrix-2000 accelerators with 128GB of DDR4 RAM. The CPU uses a 16× PCI Express 3.0 connection to communicate with the accelerator. Furthermore, a host operating system (OS) runs on the CPUs and a light-weight Linux runs on each of the Matrix-2000 processors. The host and the accelerator can communicate through the Unix socket via the PCIe.

3 MOCL DESIGN AND IMPLEMENTATION

In this section, we describe the overall design and implementation of OpenCL on Matrix-2000. We focus on the OpenCL kernel compiler and the runtime system.

3.1 Overall Design

Figure 2 shows an overview of the software stack and the OpenCL components. The left part shows the Xeon CPU and its software stack and the right part shows the Matrix-2000 and its software stack. The Xeon CPU and the Matrix-2000 are physically connected with PCIe. Both the host side and the device side have an OS running on the hardware.
runtime system (Section 3.2 and 3.3). The kernel compiler translates the OpenCL kernels into device-specific binaries (mtx) to run on the Matrix-2000, whereas the runtime implements the OpenCL host APIs and manages the runtime context [12]. Furthermore, we use a resident process as daemon on Matrix-2000 to detect requests (e.g., process creation/destroy) from the host CPU.

As we note in Section 2.1, an OpenCL program has two parts: kernels and a host program. Kernel compilation is triggered when the host program invoked the OpenCL clbuildprogram API. The host program is first compiled by a host compiler (e.g., gcc), which is then linked with the runtime library (i.e., libOpenCL.so). During runtime, the compiled kernels and the required data are first offloaded onto Matrix-2000. Then the offloaded tasks are scheduled to run on the idle hardware threads. The results are transferred back to the host side. Finally, the runtime contexts on the host and the device will be cleaned up after execution.

### 3.2 Kernel Compiler

#### 3.2.1 Compiler Description

Our compiler translates OpenCL C kernel codes into the mtx binaries. This cross-compilation process is taken place on the x64 host CPU to generate target codes for Matrix-2000. Our kernel compiler is built based on the LLVM infrastructure v5.0 and conforms to the OpenCL v1.2 specification.

Different from GPUs, Matrix-2000 runs a lightweight OS and supports POSIX Threads. Thus, the design of the kernel compiler resembles the traditional multi-core compilers. Optimizing OpenCL on multi-core CPUs is a heavily studied area and there is much research to draw upon. Our implementation follows the approaches used in prior work. We found that these strategies give good results on Matrix-2000. Specially, we schedule OpenCL work-groups to run on distinct hardware threads [12, 14, 16, 18, 24, 25]. Furthermore, scheduling work-items within a work-group to execute on a hardware thread is done either by wrapping a code region with a work-item loop [14, 16, 18, 25], or by using user-level threads [12]. In this context, we use the work-item-loop scheduling approach for reduced overheads. Forming work-item loops is achieved by using nested loops around the kernel body to replace the execution of work-items within a work-group.

Figure 3(a) shows how an OpenCL kernel is translated into a device binary on Matrix-2000. Our compiler takes an OpenCL kernel and translates it into LLVM intermediate representation (IR) for a single work-item (WIF LLVM-IR). This IR kernel is transformed into work-item loops (WIL LLVM-IR), which is then optimized by using existing LLVM optimization passes [12]. Finally, the target binary is generated with LLVM-AS and linked with libraries.

When looking into the internal structure of the kernel compiler (Figure 3(b)), we see that it relies heavily on the LLVM compiling framework. At the front end, clang is used to translate OpenCL C kernel codes into the LLVM intermediate format. Then we need to link the built-in library into the kernel. At the core of our compiler are a set of compiling passes used to transform WIF into WIL. Once it is done, we assemble the intermediate code into mtx binaries, which is submitted to the target device for execution.

After generating work-item loops, we package them into a dynamic library (i.e., kernel.so in Figure 3(b)), which is then transferred to the device. Thereafter, the device-side runtime system will open the library and get the handle of this work-group function. During runtime, the work-group functions will work as the entry function of worker threads on the device.

#### 3.2.2 Dealing with Barriers

OpenCL C uses barriers to synchronize work-items within a work-group. When producing WIL, we need to respect the synchronization semantics of the work-group barriers inside the kernel source code [14]. In this work, we partition the code with barriers into separate code regions, each of which is transformed into a work-item loop. To deal with barriers, we classify them into three categories.

**Category 1:** Unconditional barrier is the barrier that occurs on all code execution paths and separates the whole function body into two or more code regions. Thus, we create a work-item loop for each code region, and the content of each loop body is the code located in the corresponding code region. Figure 4 shows that the kernel has an unconditional barrier, which partitions the DCT kernel into two parts, each enclosed by a work-item loop.

**Category 2:** Conditional barrier is the barrier that the work-items from some work-groups follow one branch, while the work-items from other work-groups follow another. According to the OpenCL specification, the work-items from the same work-group must reach the same barriers [17]. That is, if one branch of if-statement has a barrier, then all the work-items from the same work-group will execute the same branch. Therefore, evaluating the conditional variables must yield the same results among the work-items of a work-group. To handle such conditional barriers, we create a work-item loop for the code regions before and after the barrier within a branch, and place the branch statement outside of the work-item loops. We also generate work-item loops for the code regions before and after the if-statement. Figure 5 demonstrates a kernel with a conditional barrier, and its transformed format with a total of three work-item loops.

**Category 3:** Loop barrier is the barrier that is located in a loop-statement. Loop barrier is a special case of conditional barrier: each work-item must iterate the same number of times, and, in
each iteration, the loop index must be the same among all the work-items of a work-group. To handle such barriers, we create work-item loops for the code regions before and after the barrier and the work-items of a work-group. To handle such barriers, we create work-item loops for the code regions before and after the barrier. And the loop iterations are serially enumerated, i.e., the SEO (sequential execution order) constraint. Thus, the work-items within a work-group are executed one by one and in a sequential fashion. In terms of memory access, the work-items of this work-group will access local variables sequentially, and there is no chance of running these operations concurrently within a work-group. Therefore, the atomic operations on local memory can be replaced by equivalent functional operations without synchronization. Figure 6 shows the implementation of the atomic_add function with a regular addition statement. We see that the statements run sequentially to update the atomic variable.

### 3.2.3 Lock-free Atomic Functions

The OpenCL C programming language provides a rich set of built-in functions for scalar and vector operations [17]. Among them, atomic functions can provide sequential fashion. In terms of memory access, the work-items of the runtime system implement the OpenCL APIs and generate the LLVM library calls. For example, the atomic_add function can be implemented based on the external library call _sync_fetch_and_add. The specific mappings between the OpenCL atomic functions and the LLVM library calls are shown in Table 1. Note that this is the case when the atomic variables are located in _global memory.

When atomic variables are located in _local memory, the aforementioned mapping approach is equally applicable. But we note that, the synchronization overhead from concurrent work-items can be avoided in this case. In MOCL, a work-group is transformed into a work-item loop by our kernel compiler, which is scheduled to a hardware thread. And the loop iterations are serially enumerated, i.e., the SEO (sequential execution order) constraint. Thus, the work-items within a work-group are executed one by one and in a sequential fashion. In terms of memory access, the work-items of this work-group will access local variables sequentially, and there is no chance of running these operations concurrently within a

![Figure 4: WI-loop generation for DCT which have an unconditional barrier.](image)

![Figure 5: WI-loop generation for reduction which has a conditional barrier.](image)

Table 1: The mapping between OpenCL built-in atomics and the LLVM library calls.

<table>
<thead>
<tr>
<th>OpenCL built-in atomics</th>
<th>LLVM library calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_add</td>
<td>_sync_fetch_and_add</td>
</tr>
<tr>
<td>atomic_sub</td>
<td>_sync_fetch_and_sub</td>
</tr>
<tr>
<td>atomic_xor</td>
<td>_sync_fetch_and_xor</td>
</tr>
<tr>
<td>atomic_max</td>
<td>_sync_fetch_and_max</td>
</tr>
<tr>
<td>atomic_and</td>
<td>_sync_fetch_and_and</td>
</tr>
<tr>
<td>atomic_min</td>
<td>_sync_fetch_and_min</td>
</tr>
<tr>
<td>atomic_cmpxchg</td>
<td>_atomic_compare_exchange_n</td>
</tr>
<tr>
<td>atomic_xchg</td>
<td>_atomic_exchange_n</td>
</tr>
<tr>
<td>atomic_inc</td>
<td>_atomic_fetch_and_add</td>
</tr>
<tr>
<td>atomic_dec</td>
<td>_atomic_fetch_and_sub</td>
</tr>
</tbody>
</table>

The runtime system implements the OpenCL APIs and generates the OpenCL library (libOpenCL.so). As shown in Figure 2, the runtime system has two parts: the host-side runtime and the device-side runtime. To facilitate a first-time communication and manage the runtime context, we create a daemon process on the device side.
Figure 6: The implementation of atomic_add when atomic variables are located in _local memory. Q denotes the address space, T denotes the type of atomic variables, and the atomic function returns the old value in the end.

Figure 7: The interaction between host and device.

3.3.1 Host Runtime. The runtime system needs to implement the OpenCL APIs. The host-device interaction is performed in terms of commands, which are broadly categorized into buffer allocation and deallocation, kernel compilation and execution commands, data movement commands, and synchronization commands. Figure 7 shows the workflow of a command from host to device. The host issues commands to devices, and the commands are put into command queues. Then, devices fetch commands at the queue and perform the corresponding actions to finish the offloading work. The interaction between host and device relies on the user communication library.

As an example shown in Figure 7, the buffer allocation command (i.e., clCreateBuffer) is pushed into the command queue and executed on the device using the malloc function to allocate a buffer space. When executing kernel commands, the host launches the kernel function and dispatch the predefined workgroups onto the hardware threads. The runtime system calls the kernel compiler when building kernel code. Also, Figure 7 illustrates that the daemon process is used to detect requests from the host. To manage the device-side runtime and run the offloading tasks, we use a thunk process, which is created during the OpenCL initialization stage and released when finalizing the program.

3.3.2 Device Runtime. Figure 7 shows that the thunk process manages the device-side runtime and runs the offloading tasks. This process is created and started to run at the very beginning of an OpenCL program, and is destroyed when the program exits.

We create a thread pool with a total of 32 threads on a single node, our runtime would emit the message that a single node, our runtime would emit the message that...

3.4 Implementation Details

3.4.1 Handling Kernel Variables. For _global, _constant and _local variables, they are not private to a single work-item. When generating work-item loops, we leave such variables intact. For private variables, things become complicated. When a private variable is used in only one work-item loop, we make no change of it. When the private variable is used in more than one work-item loops, but its value is the same for different work-items, we also make no change. At the same time, we need move the statements which update the content of the variables outside of the work-item loop. For other type of private variables, we have to allocate an array for them, which is located by the local index of work-items.

3.4.2 Runtime Profiling. To facilitate the performance analysis of the runtime system, we provide a profiling module to the device runtime. When we submit a kernel function, the device runtime will assemble kernel arguments, fetch idle threads from the thread pool, dispatch tasks to the fetched threads, execute tasks on each thread, and return threads back. We record the time consumed by each stage during runtime. With this module, we can analyze and measure the performance metrics such as load balancing and scheduling cost, so as to help improve MOCL’s efficiency.

3.4.3 Optimization for Scalability. When our OpenCL framework works collaboratively with MPI on multiple devices or compute nodes, we meet with two issues. The first one is that, when creating multiple processes on a node, our framework would fail to run. Each node of TH-2A has eight compute devices out of two Matrix-2000s, and it is natural to use one process to control a device. But our OpenCL framework caches kernel binaries in the local file system, and they would be stored to the same location when using multiple processes. This results in the write conflicts from different processes. To address this issue, we provide an environment variable MOCL_ENV_CACHE to indicate where to save mtx binaries.

The second issue is that, when creating multiple processes on a single node, our runtime would emit the message that devices
cannot be found. This is because when our framework initializes devices, it will create a thunk process on all devices. As a consequence, each process would contend to interact with all the devices simultaneously. To this end, we delay the thunk creation from device initialization to context creation. Then MOCL will only create the thunk process on the device that it really used.

4 EXPERIMENTAL SETUP

Hardware Platform. We evaluate our approach on both a single and multiple compute nodes of the TH-2A supercomputer. Each compute node of TH-2A is equipped with two Intel Ivy Bridge CPUs and two proprietary Matrix-2000 accelerators. Each node has 192 GB memory. The host CPUs and the Matrix-2000 are connected through PCIe. More details of the Matrix-2000 architecture can be found in Section 2.2.

Systems Software. Both the CPU and the accelerator runs an operating system (OS). The host CPU runs Redhat Linux v7.0 (with kernel 3.10). The Matrix-2000 runs a lightweight OS with Linux kernel 3.14. We use an in-house driver to enable data communications between the host and Matrix-2000.

Benchmarks. To validate and evaluate our OpenCL implementation, we use 70 benchmarks from the NVIDIA SDK, AMD SDK, SNU NPB, Shoc and Parboil11 suites. To evaluate our approach across computing nodes, we use Clover, a mini-app that solves the compressible Euler equations on a Cartesian grid, using an explicit, second-order accurate method. It uses OpenCL for kernel execution and MPI for cross-node execution. We analyze these benchmarks and collect information from them to guide the implementation of MOCL.

Performance Report. For each test case, we report the geometric mean performance across all benchmarks. We run each test case multiple times, until the difference between the upper and lower confidence bounds under a 95% confidence interval setting is smaller than 5%. We compare our approach to POCL [3], an open-sourced OpenCL implementation based on LLVM, in terms of kernel compilation policies and scheduling cost.

5 EXPERIMENTAL RESULTS

In this section, we first evaluate the efficiency of MOCL, then we give some optimization tips for users who want to write efficient OpenCL programs on Matrix-2000. At last, we evaluate the scalability of MOCL on the TianHe-2A system.

5.1 Kernel Compilation Policies

In MOCL, an OpenCL kernel is compiled when the OpenCL API cEnqueueNDRange is invoked by the host program. At this point, the work-group size is known to the compiler. Thus, we can pass this work-group size as either a constant parameter or a variable parameter to the compiler. We refer the first strategy as constant parameter compilation (CPC) and the latter as variable parameter compilation (VPC). These two policies differ in that CPC has to recompile the kernel when using a different work-group size. This occurs when an OpenCL kernel is called in an iterative manner and the work-group size changes between iterations. In contrast, using a variable parameter (VPC) can avoid the overhead of recompilation.

Figure 9 shows speedup of kernel compilation and execution time for CPC over VPC. A speedup of over 1 means that CPC yields better performance. Overall, we see that using a constant work-group can often achieve a better performance than using a variable work-group. This is because the work-group size is fixed in such a case and we only have to compile kernels once. In addition, using a constant upper bound for a work-item loop brings more optimization opportunities such as vectorization. This is why using the CPC policy performs better even though the kernel compilation takes more time. Nonetheless, we note a different performance behavior for BT, LU, NvBlsh, ScReduce, ScScan, and PbbF. A common observation of these benchmarks can noted that the kernels are invoked iteratively and the work-group size changes between iterations. As a result, the overhead of recompiling kernels becomes nonneglible, and using VPC runs faster. In MOCL, we provide programmers with both CPC and VPC. In default, the CPC policy is enabled. But programmers can switch to the VPC policy by setting the environment variable MOCL_ENV_KCP=2.

5.2 Device Runtime Scheduling

Figure 10 shows the scheduling cost between MOCL and POCL. For this, we synthesize a microbenchmark with an aim to measuring the overhead of the scheduling strategies used in MOCL and POCL. This microbenchmark is an OpenCL program that has a host part for managing contexts and a kernel part with an empty body. Therefore, the scheduling overhead can be roughly estimated by measuring the running time from the start of task dispatching to the moment when all the work-groups have been done.

In POCL, each worker thread pulls a fixed amount of work-groups from the task pool in each round. Every time a thread is pulling, it will have to synchronize with the other threads. This synchronization overhead becomes nonneglible in particular when we have a large number of work-groups. This is due to the fact that the worker threads need to make pulling again and again. In MOCL, however, we use a master thread to push tasks (i.e., the work-groups) to worker threads in one time. This often holds because the number of work-groups is determined at the time of starting an NDRange. Therefore, the worker threads do not have to synchronize with each other thereafter. This elucidates the observation in Figure 10 that, when the number of work-groups grows, the scheduling cost of MOCL becomes significantly smaller than that of POCL.

When we only have a very few work-groups, the scheduling overhead of MOCL is also much smaller than that of POCL. This is because that MOCL only activates a necessary number of worker threads which are not more than the number of tasks and leaves others idle. By contrast, POCL will activate all the idle threads and dispatch tasks to each of them. Therefore, our scheduling strategy can avoid the unnecessary scheduling cost. This is particularly true when an OpenCL program starts its kernel(s) iteratively.

A natural extension to our scheduling approach is to leverage the work-stealing technique. That is, we first push tasks to the worker threads at one time and then apply the work-stealing technique among threads during runtime. By systematically analyzing the 70 benchmarks, however, we observe that most OpenCL programs...
We implement two versions of the atomic functions: one is built on
MOCL and the other on POCL. The x-axis represents the number of work-
groups, each having only one work-item and the y-axis denotes the scheduling time from dispatching tasks to their completion.

Figure 11: The workload per thread of EP and PbTPACF

have a rather even workload distribution among work-groups. As
shown in Figure 11(a), the one-time workload distribution can
guarantee a good load balancing. Thus, applying work-stealing on these programs is not a must. Meanwhile, programs such as
PbTPACF have branches and thus distinct work-items may run different amounts of workloads (Figure 11(b)). For such programs, it is
still necessary to implement the work-stealing technique. Provided that there are very few such programs among the 70 benchmarks,
we will leave this extension for the future work. To summarize, we
calculate that the average ratio (between the fastest worker thread
and the slowest one) of load balancing is around 90.5%, which is
considered to be well load balanced.

5.3 Evaluating Atomic Operations

We evaluate the performance of our atomic functions over several
programs, which are from the Parboil and AMD benchmark suites.
We implement two versions of the atomic functions: one is built on
the LLVM sync library calls, and the other uses our lock-free im-
plementation. We measure their kernel execution time to compare
the performance of our optimized atomic functions. From Table 2,
we see that our optimized atomic functions can achieve a speedup
of up to 4.32x, when compared to the baseline implementation.

Table 2: The performance evaluation of the optimized atomic functions when atomic variables are located in __local memory. The speedup is the performance of our implementation versus the one based on external library calls.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Atomics</th>
<th>Locations</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>StrS</td>
<td>atomic_{inc, dec}</td>
<td>__local</td>
<td>1.11</td>
</tr>
<tr>
<td>rad0</td>
<td>atomic_{inc}</td>
<td>__local</td>
<td>1.08</td>
</tr>
<tr>
<td>btest4</td>
<td>atomic_{inc}</td>
<td>__local</td>
<td>3.32</td>
</tr>
<tr>
<td>PbBFS</td>
<td>atomic_{add, min, max}</td>
<td>__local, global</td>
<td>1.00</td>
</tr>
<tr>
<td>PbBhisto</td>
<td>atomic_{add, min, max}</td>
<td>__local, global</td>
<td>1.00</td>
</tr>
<tr>
<td>PbTPACF</td>
<td>atomic_{inc}</td>
<td>__local</td>
<td>1.28</td>
</tr>
</tbody>
</table>

variables in __global memory. As we perform no optimizations on
such atomic functions, these benchmarks can gain a slight perfor-
man ce improvement. Other programs such as histo and PbTPACF,
work on updating atomic variables in __local memory. We note
that such benchmarks can yield significant speedups, which comes
from the usage of our lock-free atomics.

5.4 Programming and Code Optimization on
Matrix-2000: A Programmer’s Perspective

With the help of our OpenCL framework, programmers can take
Xeon-Matrix2000 as a conventional heterogeneous platform. But
we argue that the following optimization guidelines are required to
achieve high performance on this platform.

5.4.1 On Reduction Operations. In MOCL, the work-items in a
work-group are executed in a sequential way, i.e., the sequential
execution order (SEO). By leveraging this implicit constraint, we
can remove the usage of local memory and barrier. We analyze
the benchmarks and find that the reduction operation can be reim-
plemented with SEO in a straightforward manner. In Figure 12,
we show how we use SEO to implement reduction operations for
Reduc in the AMD-SDK benchmark suite. In the original kernel,
it uses a loop and barrier to implement reduction. When taking
the SEO constraint into account, we can simply implement the
reduction operation in an equivalent way that we sum the private
result for the first work-item to the last one.

We rewrite the benchmarks with reductions and implement the
reduction operation without local memory or barrier. The speedup
can be up to 1.5x, when compared to the original kernel. Specif-
ically, Reduc, IS, PbTPACF, PbBFS can achieve a speedup of 1.42x,
1.03x, 1.51x, and 1.07x, respectively. Therefore, we recommend
that programmers remove the usage of local memory and load data
elements directly from the global space for reductions.
As doing so, we will introduce extra memory accesses and synchronizations.

Figure 12: Reduction operation optimization for Reduction.

---

5.4.3 On Memory Access Patterns. There are two typical memory access patterns: (1) a work-item accesses memory contiguously (Figure 14(a)), and (2) the access distance between two neighbouring work-items is far from each other (Figure 14(b)). In MOCL, the work-items in one work-group are scheduled for execution in a sequential fashion. Thus, pattern 1 can bring much larger memory bandwidth than pattern 2. As a result, pattern 1 is preferred by programmers to implement OpenCL kernels. We compare the performance of these patterns with DCT and QRS, and see that using pattern 1 can yield a speedup of 1.03x and 1.01x, respectively.

---

5.5 Evaluating the Scalability of MOCL

We evaluate the scalability of MOCL on either multiple Matrix-2000s or multiple compute nodes, each with multiple devices.

**On Multiple Matrix-2000s.** In order to evaluate MOCL’s performance on multiple devices, we select the benchmarks which use multiple devices from Nvidia and AMD benchmark suites. We run these programs in three cases: using 1 device, 4 devices and 8 devices. From Figure 15, we see how the performance increases when using increasingly more devices.

**On Multiple Nodes with Multiple Matrix-2000s.** We run the mini-app Clover on 256 nodes of TH-2A to evaluate the scalability of MOCL. Clover is implemented with MPI and OpenCL, where each MPI process is used to control a device. To fully utilize the TH-2A system, we run 8 processes per compute node, which corresponds to 8 devices. We first run Clover with a fixed input size, and change the number of nodes and devices. The execution time of Clover is shown in Figure 16(a). We see that the execution time decrease when increasing the number of MPI processes. But when the number of MPI processes is large than 8, the change of execution time is very slight. Then we keep the task size of each MPI process constant, and vary the number of nodes and input size. The execution time is shown in Figure 16(b). When the input dataset increases, we see that the execution time increase slowly. It means that MOCL achieves a good strong and weak scalability on the TH-2A system.
6 RELATED WORK

There exist various OpenCL implementations, which are shown in Table 3. On the one hand, we notice that most vendor implementations are close-source, except the one from AMD. This open source Linux Compute project is Radeon Open Compute ROCm for Radeon Graphics GCN 3 and 4 (Hawaii, Fiji, Polaris) and Intel Xeon E5v3 and Corev3 CPU (Haswell and newer) or new AMD Ryzen with PCIe Gen3 atomic capability [4]. Meanwhile, the OpenCL implementation from TI is customized to TI SoCs (an ARM CPU + a TI DSP) [6]. On the other hand, the open-source implementations are typically developed and maintained by academia. The Gallium Compute Project maintains an implementation of OpenCL mainly for AMD Radeon GCN (formerly known as CLOVER), and it builds on the work of the Mesa project to support multiple platforms [2], BEIGNET is an implementation released by Intel in 2013 for its integrated GPUs (Ivy Bridge and newer) [1]. POCI is a CPU-oriented OpenCL implementation built on Clang and LLVM. In addition, POCI supports the TTA and HSA architecture [14, 15]. As of April 2017, POCI has an experimental support for NVIDIA GPU devices via a new backend which makes use of the LLVM NVPTX backend and the CUDA driver API. Similar to POCI, FreeOCL also supports a large range of multi-core CPUs with the help of the generic C++ compilers [7]. But this framework is purely CPU-oriented and cannot be extended to accelerators in a straightforward way. Although these frameworks provide us with valuable building blocks, none of them can be directly applicable to the Xeon-Matrix2000 platform.

In [12], Gummaraju et al. present Twin Peaks, a software platform for heterogeneous computing. This allows codes originally targeted for GPUs execute efficiently on CPUs as well. In particular, they propose several techniques in the runtime system to efficiently utilize the caches and functional units present in CPUs. The experimental results show that the techniques enable GPGPU-style code to run efficiently on multicore CPUs with minimal runtime overheads. In [20], Lee et al. present the design and implementation of an OpenCL framework for architectures which consist of a general-purpose processor core and multiple accelerator cores without caches but a small internal local memory. Their OpenCL C kernel translator contains three source-code transformation techniques to boost performance. In [25], Stratton et al. describe a framework (MCUDA), which allows CUDA programs to be executed efficiently on shared memory, multi-core CPUs. The framework consists of a set of source-level compiler transformations and a runtime system for parallel execution. This approach can achieve a better performance than the OpenMP version on multicore CPUs. These OpenCL implementations are targeted for multicore CPUs. Our kernel compiler resembles theirs by using the work-item loop strategy. Different from these work, our kernel compiler further optimizes kernel performance by fully leveraging the SEO constraint.

To compare and contrast architectural designs and programming systems, Danalis et al. have designed the Scalable Heterogeneous Computing benchmark suite (SHOC) [10], which is a spectrum of programs that test the performance and stability of these scalable heterogeneous computing systems. In this context, we use these benchmarks to evaluate the performance of our design. In [22, 23], Shen et al. compare the performance of OpenCL and OpenMP on three x86_64 multicores. They identify the factors that significantly affect the overall performance of the OpenCL code. By taking a reasonable OpenMP implementation as a performance reference, they optimize the OpenCL code to reach or exceed this threshold. The authors find that the performance of OpenCL codes is affected by hard-coded GPU optimizations which are unsuitable for multi-core CPUs, the fine-grained parallelism of the model, and the immature OpenCL compilers. On the Matrix-2000 architecture, we have similar observations that the GPU-customized OpenCL codes perform even worse than the serial code. This motivates us to generate efficient codes for Matrix-2000 from the OpenCL codes with GPU-specific optimizations in the future.

7 CONCLUSION

This paper has presented the design and implementation of an OpenCL kernel compiler and runtime for the Matrix-2000 accelerator. The core of our compiler are a set of compiling passed built on the LLVM infrastructure. We provide extensive discussions on our design choices, and optimization strategies for implementing the OpenCL compiler and runtime. We evaluate our approach by applying to 70 OpenCL benchmarks on a single accelerator, and across accelerators and computing nodes. Experimental results show that our optimization strategies give better performance when compared
with a state-of-the-art open-sourced OpenCL implementation. We share our experience on code optimization on Matrix-2000, providing useful insights on how to effectively program this unique architecture.

REFERENCES