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

Peng Zhang
National University of Defense Technology
China
zhangpeng13a@nudt.edu.cn

Jianbin Fang
National University of Defense Technology
China
j.fang@nudt.edu.cn

Canqun Yang
National University of Defense Technology
China
canqun@nudt.edu.cn

Tao Tang
National University of Defense Technology
China
taotang84@nudt.edu.cn

Chun Huang
National University of Defense Technology
China
chunhuang@nudt.edu.cn

Zheng Wang
Lancaster University
United Kingdom
z.wang@lancaster.ac.uk

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 how to optimize code on the new Matrix-2000 architecture.

KEYWORDS

Heterogeneous Computing; OpenCL; Runtime and Compiler Optimization

ACM Reference Format:
Peng Zhang, Jianbin Fang, Canqun Yang, Tao Tang, Chun Huang, and Zheng Wang. 2018. MOCL: An Efficient OpenCL Implementation for the Matrix-2000 Architecture. In *CF '18: CF '18: Computing Frontiers Conference, May 8–10, 2018, Ischia, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3203217.3203244

1 INTRODUCTION

TianHe-2A (TH-2A) is an upgrade of the leading TOP500 high-performance-computing (HPC) system TianHe-2 (TH-2) [10]. 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 [10, 15]. 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, 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 fit 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 [12].

Since existing OpenCL frameworks mainly target CPUs and GPUs [1, 2, 6, 20, 23, 25], they are not directly applicable to Matrix-2000. 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 need 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 **O**pen**CL** programming interface for **M**atrix-2000. `MOCL` consists of two main components: a kernel compiler and a runtime. The kernel compiler is built upon the LLVM compiler infrastructure [24]. It translates each OpenCL kernel of a program 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, so that the work-items within the same group are executed by a single thread to run in a sequential 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. The idea is to partition the work to be run by 32 parallel threads, so that each parallel thread executes on one of the 32 cores of a Matrix-2000 super-node. As we will show later in the paper, this strategy leads to better performance compared to

[1]As of January 2018, TianHe-2 is ranked as the second fastest HPC system with a peak performance of 54.9 petaflops in the fiftieth TOP500 list: https://www.top500.org/
[2]An OpenCL work-group is a collection of work-items that can execute on a single compute unit. Here, a work-item is an invocation of the kernel on a given input.

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 suits. 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 with MPI for cross-node communication, provides good scalability across multiple computing nodes. We provide extensive discussions on how to optimize OpenCL programs on the Matrix-2000 architecture, offering useful insights on how to write efficient code for this 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.

2 BACKGROUND

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

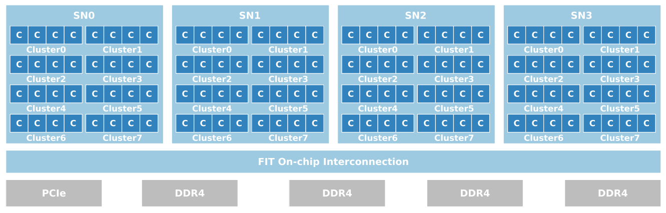
2.1 Open Computing Language

Open Computing Language (OpenCL) is a standardized programming model for heterogeneous computing [22]. OpenCL defines a generic platform model comprising a host and one or several devices known as computation engines. The computation engines can be central processing units (CPUs), or "accelerators" like GPUs and Matrix-2000 processors. A device can have multiple *processing elements* (PEs). The PEs can be grouped into several *compute units* (CUs) which have *global* and *local* memory.

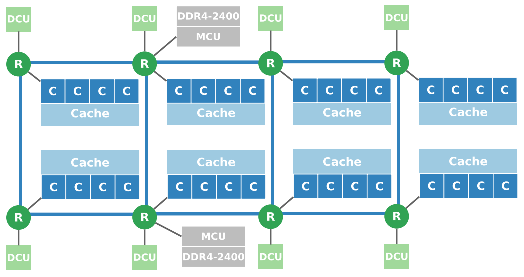
An OpenCL program consists of two parts: *kernels* to run on one or more devices, and a *host program* that executes on the host (which is typically a CPU). The host program defines and manages the kernel execution contexts. The computational task is typically coded into kernel functions. When a kernel 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 other work-items in the same work-group [13, 14, 16]. All work-items can access the *global memory*.

2.2 The Matrix-2000 Architecture

Figure 1(a) provides a high-level overview of Matrix-2000. A Matrix-2000 processor has 128 computing cores running at 1.2 GHz, offering a peak performance of 2.46 Tflops. Computing cores are grouped into four super-nodes (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. Each core is a 256-bit vector instruction set architecture with two 256-bit vector functional units.



(a) The conceptual structure of Matrix-2000.



(b) The SuperNode (SN) topology.

Figure 1: An overview of the Matrix-2000 accelerator.

Figure 1(b) shows the topology of the network on chip (NoC) within a SN in Matrix-2000. The NoC is a 4×2 mesh, with a total of eight routers. A cluster and a directory control unit (DCU) are pinned to a router. Furthermore, 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 level, each computing 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 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 DESIGN AND IMPLEMENTATION OF MOCL

In this section, we describe the overall design and implementation of MOCL. Here we focus on the OpenCL kernel compiler and the runtime system.

3.1 Overall Design

Figure 2 provides an overview of the software stack and the components of MOCL. At the hardware level, the host CPU and the Matrix-2000 accelerators are connected via PCIe; at the software level, both sides run an operating system. The driver works as the communication backbone between host and accelerators. In the user space, a user communication library is developed to enable the host-device communication.

MOCL is developed on top of the user communication library and the LLVM compiling infrastructure. It consists of the kernel compiler and the runtime system (Section 3.2 and 3.3). The kernel

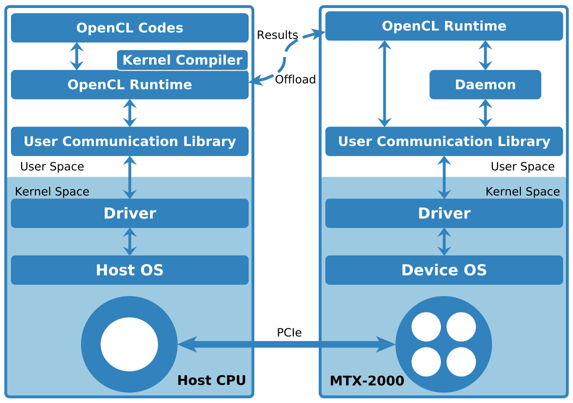


Figure 2: An overview of the software stack and MOCL. The left part shows the host CPU and its software stack and the right part shows the Matrix-2000 and its software stack. The host CPU and the Matrix-2000 are physically connected through PCIe. The host and the device run an individual operating system.

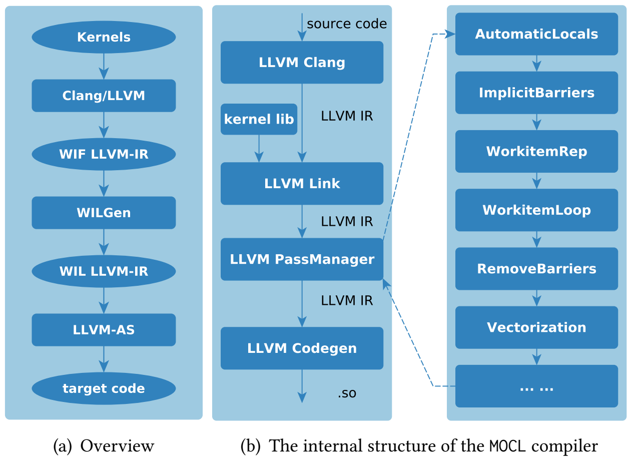


Figure 3: The compilation work flow on Matrix-2000: (a) shows the overall framework, and (b) shows the data analysis flow and compilation passes based on LLVM.

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 [19]. 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 invokes 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 a Matrix-2000 accelerator. Then, the offloaded tasks are scheduled to run on the idle hardware threads. At the end of kernel execution, the results will be transferred back to the host.

3.2 The MOCL Kernel Compiler

3.2.1 Compiler Description. The compiler of MOCL translates OpenCL C kernel codes into the Matrix-2000 instructions, `mtx` and stored as binaries. This is a cross-compilation process taken place on the host CPU . The kernel compiler implements the OpenCL `v1.2` specification.

Unlike GPUs, Matrix-2000 runs a lightweight operating system 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 general strategies used in prior work [11, 19–21, 23, 28, 29]. We found that these strategies give good results on Matrix-2000. Specifically, we schedule OpenCL work-groups to run on distinct hardware threads. There are two ways for scheduling work-items within a work-group to execute on a hardware thread: (1) wrapping a code region to a *work-item loop* [20, 21, 23, 29], or using *user-level threads* [19]. MOCL employs the first strategy by statically partitioning the work-items into a number of work-item loops (`WIL`) to reduce dynamic thread scheduling overhead. Forming work-item loops is achieved by using nested loops around the kernel(work-item function, WIF), where each loop performs a certain number of work-items of a work group.

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, a set of compiling passes is 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 runtime system loads the library to obtain the handle of each work-group function. During runtime, the work-group functions serve as the entry function of worker threads that run on the device.

3.2.2 Support Synchronization Barriers. OpenCL C uses *barriers* to synchronize work-items within a work-group. When producing work-item loops, we need to respect the synchronization semantics of the work-group *barriers* inside the kernel source code [20]. In this work, we partition the code with barriers into separate *code regions*, each of which is transformed into a work-item loop. We support three types of barriers, described as follows.

Category 1: An *unconditional barrier* 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: A *conditional barrier* diverges the execution path of the kernel, so that work-items from some work-groups follow

```
__kernel void DCT(__global float * output,
  __global float * input, __global float * dct8x8,
  __local  float * inter, const uint width,
  const uint blockWidth,  const uint inverse){

  /* ... */
  for(uint k=0; k < blockWidth; k++){
    uint index1 = (inverse)? i*blockWidth + k : k * blockWidth + i;
    uint index2 = getIdx(groupIdx, groupIdy, j, k, blockWidth, width);
    acc += dct8x8[index1] * input[index2];
  }
  inter[j*blockWidth + i] = acc;

  barrier(CLK_LOCAL_MEM_FENCE);

  acc = 0.0f;
  for(uint k=0; k < blockWidth; k++){
    uint index1 = i* blockWidth + k;
    uint index2 = (inverse)? j*blockWidth + k : k* blockWidth + j;
    acc += inter[index1] * dct8x8[index2];
  }
  output[idx] = acc;
}
```

(a) Unconditional barriers

```
__kernel void DCT(__global float * output,
  __global float * input, __global float * dct8x8,
  __local  float * inter, const uint width,
  const uint blockWidth,  const uint inverse){
/* WI_loop_1 { */
  /* ... */
  for(uint k=0; k < blockWidth; k++){
    uint index1 = (inverse)? i*blockWidth + k : k * blockWidth + i;
    uint index2 = getIdx(groupIdx, groupIdy, j, k, blockWidth, width);
    acc += dct8x8[index1] * input[index2];
  }
  inter[j*blockWidth + i] = acc;
/* } */
/* WI_loop_2 { */
  acc = 0.0f;
  for(uint k=0; k < blockWidth; k++){
    uint index1 = i* blockWidth + k;
    uint index2 = (inverse)? j*blockWidth + k : k* blockWidth + j;
    acc += inter[index1] * dct8x8[index2];
  }
  output[idx] = acc;
/* } */
}
```

(b) Resulting code after work-item loop generation

Figure 4: Work-item loop code generation for the `DCT` OpenCL kernel which has an unconditional `barrier`.

```
__kernel void reduce6(__global T *g_idata,
  __global T *g_odata, unsigned int n,
  __local volatile T* sdata){

  /* pre-process */

  // do reduction in shared mem
  if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; }

      barrier(CLK_LOCAL_MEM_FENCE);

  }

  /* post-precess */

}
```

(a) Barriers in if-statement

```
__kernel void reduce6(__global T *g_idata,
  __global T *g_odata, unsigned int n,
  __local volatile T* sdata){
/* WI_loop_1 { */
  /* pre-process */
/* } */
  // do reduction in shared mem
  if (blockSize >= 512) {
/* WI_loop_2 { */
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; }
/* } */
  }

/* WI_loop_3 { */
  /* post-process */
/* } */
}
```

(b) Resulting code after work-item loop generation

Figure 5: Work-item loop code generation for the `reduction` OpenCL kernel which has a conditional `barrier`.

one branch, while work-items from other work-groups follow another branch. According to the OpenCL specification, the work-items from the same work-group must reach the same barriers [22]. That is, if one branch of an *if-statement* has a barrier, then all the work-items from the same work-group will execute that 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 loop. We also generate work-item loops for the code regions before and after the *if-statement*. As an example, Figure 5 shows a kernel with a conditional barrier, and the transformed code that has three work-item loops.

Category 3: A *loop barrier* locates inside a *loop-statement*. A loop barrier is a special case of the conditional barrier because 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 within the loop body, and place the loop statement outside them.

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 [22]. Among them, atomic functions can provide atomic operations on 32-bit signed, unsigned integers and single precision floating-point data types. The atomic functions are typically implemented using a set of *atomic* library calls that are generated by LLVM. 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. The list of functions support atomic variables stored in the global memory.

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

OpenCL built-in atomics	LLVM library calls
atomic_add	__sync_fetch_and_add
atomic_sub	__sync_fetch_and_sub
atomic_xchg	__atomic_exchange_n
atomic_inc	__sync_fetch_and_add
atomic_dec	__sync_fetch_and_sub
atomic_cmpxchg	__atomic_compare_exchange_n
atomic_min	__sync_fetch_and_min
atomic_max	__sync_fetch_and_max
atomic_and	__sync_fetch_and_and
atomic_or	__sync_fetch_and_or
atomic_xor	__sync_fetch_and_xor

When atomic variables are stored in local memory, the aforementioned mapping approach are still 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. The loop iterations will then be serially enumerated by following the SEO (sequential execution order) constraint. That is, 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,

```
// read, add, store
__attribute__((overloadable))
T atomic_add(volatile Q T *p, T val)
{
  T retval = *p;
  *p = retval + val;
  return retval;
}
```

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.

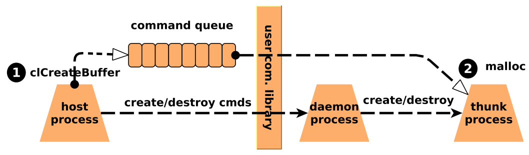


Figure 7: The interaction between host and device.

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 *p*. This lock-free atomics can significantly improve kernel performance (see Section 5.4) by avoiding the synchronization overheads.

3.3 The `MOCL` Runtime

The `MOCL` runtime implements the OpenCL APIs and generates the OpenCL library (`libOpenCL.so`). As shown in Figure 2, the runtime system has two components: a host-side runtime and a device-side runtime. To facilitate a first-time communication and manage the runtime context, we create a `daemon` process on the device side.

3.3.1 Host Runtime. The host runtime system needs to implement the OpenCL host APIs. The host-device interaction is performed using *OpenCL 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 issuing a command from the host to execute on the device. Here, the host commands are put into the command queues from which the devices fetch commands to execute. The host-device interaction is realized through the user communication library.

As an example shown in Figure 7, the buffer allocation command (i.e., *clCreateBuffer*) is pushed into the command queue (❶) and executed on the device using the `malloc()` function to allocate a buffer space (❷). When executing *kernel commands*, the host launches the kernel function and dispatches the predefined work-groups onto the hardware threads. The runtime system calls the kernel compiler when building kernel code, and the `daemon` process monitors 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.

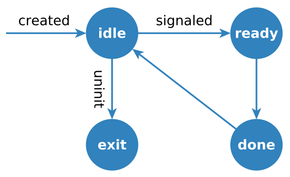


Figure 8: Thread states and their transitions.

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 beginning of an OpenCL program, and is destroyed before the program exits.

We create a thread pool with a total of 32 threads on a `SN` of Matrix-2000 when initializing the `thunk` process. Figure 8 shows that each thread has four states: `idle`, `ready`, `done`, and `exit`. The initial state of threads is idle. A *condition variable* is used to signal the idle threads to be ready. By doing so, we aim to avoid continually polling and save cycles. Note that, when the number of tasks (i.e., #work-groups) is less than the number of idle threads, we will use only a portion of them. Once the threads done with tasks, they will be switched back to be idle and put back into the thread pool again. `MOCL` will clean up the thread context and the associated data when the thunk process exits.

Distributing pending tasks to idle threads is at the core of the device runtime. In this context, a work-group is regarded as the basic unit of task distribution. When starting an OpenCL kernel, our runtime queries the number of idle threads and evenly dispatches all of the tasks to idle threads at the same time. If the number of idle threads is greater than that of tasks, only a portion of the threads are signaled. Different from the task dispatching strategy used in `POCL` [20], we call our approach a *push*-based strategy. Our runtime pushes tasks to available threads while in `POCL`, all the available threads are pulling tasks once they have completed the assigned tasks. Our task dispatching strategy has the benefit of mitigating the overhead of thread polling and synchronization.

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 the other types 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 for 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 Supporting Multiple Devices. MOCL can be used with MPI to support multiple computing devices. Recall that MOCL caches kernel binaries on the local file system. This means that each concurrent compilation process would generate a single binary. To avoid write conflicts, we provide an environment variable MOCL_ENV_CACHE to indicate where to save mtx binaries, so that each compilation process can store the binaries into a dedicate directory. Furthermore, MOCL only creates the thunk process on the device that is being 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 the 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 v.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 Parboil 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 [3]. 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-source OpenCL implementation based on LLVM, in terms of kernel compilation policies and scheduling cost.

5 EXPERIMENTAL RESULTS

In this section, we first present the overall performance MOCL, and evaluate its efficiency. Then we share our experiences on how to write efficient OpenCL programs on Matrix-2000. Finally, we evaluate the scalability of MOCL on the TH-2A system.

5.1 Overall Performance

We present the performance of OpenCL programs on Matrix-2000, by comparing the parallel versions with the serial versions. For this, we build two versions of MOCL: one uses 32 threads per device, and the other one uses a single thread. Figure 9 presents the overall performance of the 70 programs. Compared to the serial programs, we can obtain an average speedup of 7.1x with MOCL.

For the programs such as btncs, fwtf, the speedup is smaller than 1. This is because their kernels are invoked for over 1400 times, but the kernel execution takes only 100us per iteration. The overhead of thread management is larger than the kernel execution span, which leads to a slow parallel version on Matrix-2000. And for other programs such as binsch, dwh, and LU, the parallel versions only run slightly faster than their serial counterparts. This is due to the fact that these programs have only 1 or 2 work-groups, and there are insufficient work-groups to fully utilize the resources.

On the other hand, we can achieve a significant speedup when (1) the program has enough work-groups, (2) the kernel execution time is much larger than the overhead of thread management, and (3) the workloads across threads are balanced. For example, EP has 4096 work-groups and its kernel executes 11s at a time. Moreover, the hardware threads on Matrix-2000 can achieve a balanced load, shown in Figure 12(a). As a result, EP obtains a speedup of 31.7x.

5.2 Kernel Compilation Policies

In MOCL, an OpenCL kernel is compiled when the OpenCL host API clEnqueueNDRange is invoked. 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 10 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, ScReduc, ScScan, and PbBFS. A common observation of these benchmarks can be noted that the kernels are invoked iteratively and the work-group size changes between iterations. As a result, the overhead of recompiling kernels becomes nonneglectable, 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.

5.3 Device Runtime Scheduling

Figure 11 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,

[3]CloverLeaf: http://uk-mac.github.io/CloverLeaf/.

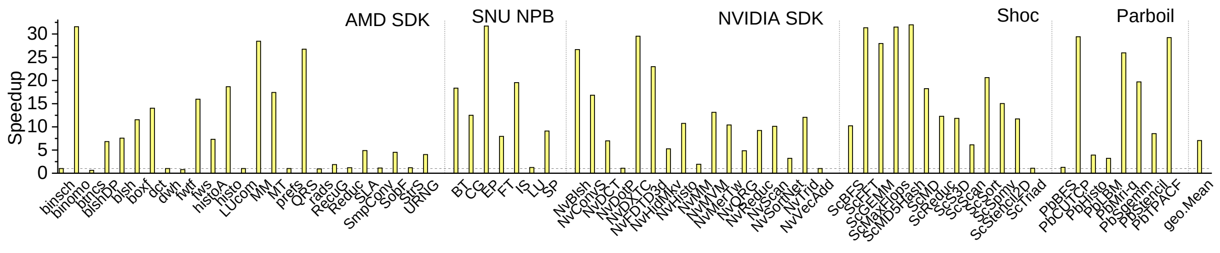


Figure 9: Speedup of OpenCL programs over the serial versions on Matrix-2000. The value of the reference line is equal to 1.0.

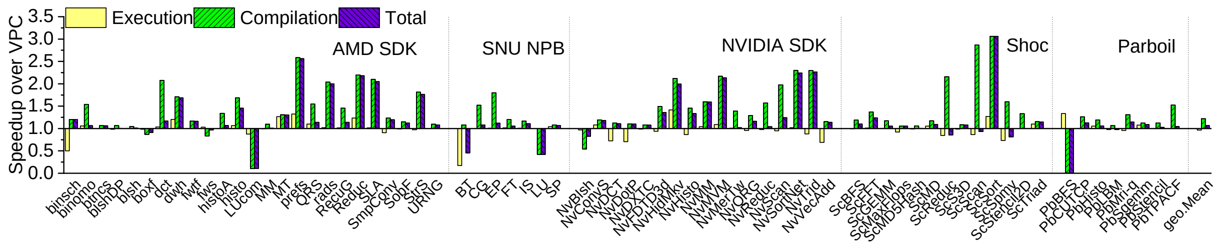


Figure 10: Speedup of kernel compilation strategy CPC over VPC.

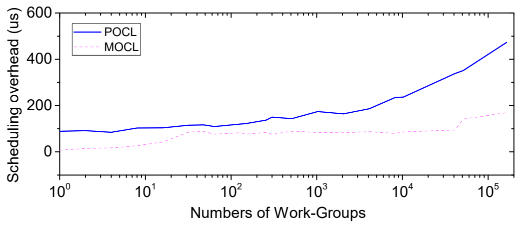
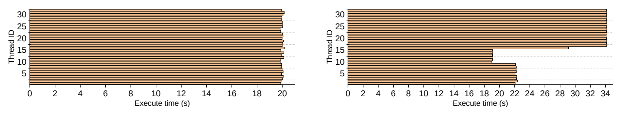


Figure 11: Comparing the scheduling overhead between POCL and MOCL. The x-axis represents #work-groups, each having only one work-item and the y-axis denotes the scheduling time from dispatching tasks to their completion.



(a) Load distribution of EP (b) Load distribution of PbTPACF

Figure 12: The workload per thread of EP and PbTPACF

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 nonnegligible 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 theads 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 explains the observation in Figure 11 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 no 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 have a rather even workload distribution among work-groups. As shown in Figure 12(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 12(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.

5.4 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 implementation. We measure their kernel execution time to compare

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.

Programs	Atomics	Locations	Speed up
StrS	`atomic_{inc, dec}`	`__local`	1.11
rads	`atomic_inc`	`__local`	1.08
histoA	`atomic_inc`	`__local`	4.32
PbBFS	`atomic_{add, min, xchg}`	`__{local, global}`	1.00
PbHisto	`atomic_{add, min, max}`	`__{local, global}`	1.00
PbTPACF	`atomic_inc`	`__local`	1.28

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. Most atomic functions in `PbBFS` and `PbHisto` work on updating variables in `__global` memory. As we perform no optimizations on such atomic functions, these benchmarks can gain a slight performance improvement. Other programs such as `histoA` 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.

5.5 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.

5.5.1 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 reimplemented with SEO in a straightforward manner. In Figure 13, 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 from 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. Specifically, `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.

5.5.2 On Local Memory Usage. The local memory in OpenCL can be used to stage the data from global memory, or used to cache the result produced by other work-items. Using local memory can enable faster memory accesses, because local memory is located on-chip and has a much smaller access latency on GPUs. But it is not the same for Matrix-2000, where the architecture has no such physical on-chip buffers. In such a case, local memory is implemented as

```
__kernel void reduce(__global uint4* input,
  __global uint4* output, __local uint4* sdata)
{
  unsigned int tid = get_local_id(0);
  unsigned int bid = get_group_id(0);
  unsigned int gid = get_global_id(0);
  unsigned int localSize = get_local_size(0);
  unsigned int stride = gid * 2;
  sdata[tid] = input[stride] + input[stride + 1];
  barrier(CLK_LOCAL_MEM_FENCE);
  for(unsigned int s = localSize >> 1; s > 0; s >>= 1)
  {
    if(tid < s)
    {
        sdata[tid] += sdata[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if(tid == 0) output[bid] = sdata[0];
}
```

(a) Reduction with explicit synchronization

```
__kernel void reduce(__global uint4* input,
  __global uint4* output, __local uint4* sdata)
{
  unsigned int tid = get_local_id(0);
  unsigned int bid = get_group_id(0);
  unsigned int gid = get_global_id(0);
  unsigned int localSize = get_local_size(0);
  unsigned int stride = gid * 2;
  if (tid == 0)
    output[bid] = 0;
  output[bid] += input[stride] + input[stride + 1];
}
```

(b) Reduction with implicit synchronization

Figure 13: Reduction operation optimization for `Reduction`.

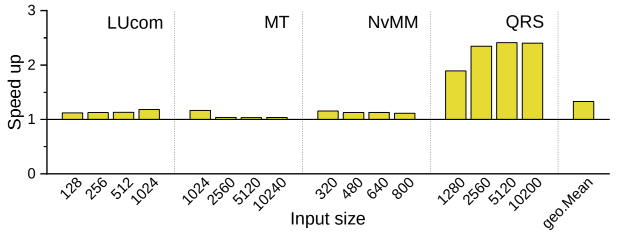
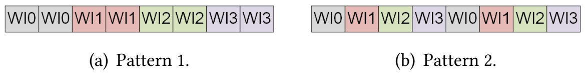


Figure 14: The performance without local memory compared to with local memory. The kernels without using local memory run up to 2.4x faster.

a buffer in the global memory space. Therefore, accessing local memory is same as accessing global memory on Matrix-2000.

Among all the benchmarks, we note several of them are using local memory to stage data: `LUcom`, `MT`, `NvMM`, and `QRS`. We rewrite these programs by removing the usage of local memory. Figure 14 shows that we obtain an up to 2.4x speedup, compared to the original kernel. We demonstrate the speedup of the optimized version compared to the version with local memory. Overall, the kernels without local memory achieve an average speedup of 1.3x over the original kernels. To summarize, it is not recommended that local memory be used in OpenCL kernels on Matrix-2000. As doing so, we will introduce extra memory accesses and synchronizations.

5.5.3 On Memory Access Patterns. There are two typical memory access patterns: (1) a work-item accesses memory contiguously (Figure 15(a)), and (2) the access distance between two neighbouring work-items is 1, but the data elements accessed by one work-item may be far from each other (Figure 15(b)). In `MOCL`, the work-items in one work-group are scheduled for execution in a sequential fashion. As a result, pattern 1 can bring a much larger memory


WI0 WI0 WI1 WI1 WI2 WI2 WI3 WI3
WI0 WI1 WI2 WI3 WI0 WI1 WI2 WI3

(a) Pattern 1. (b) Pattern 2.

Figure 15: Memory access patterns. This is the memory footprint of one work group. WI0 means the memory that work item 0 will access.

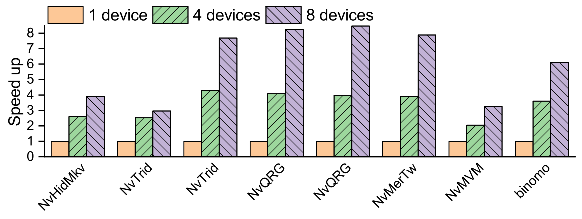


Figure 16: Evaluate the performance of multiple devices

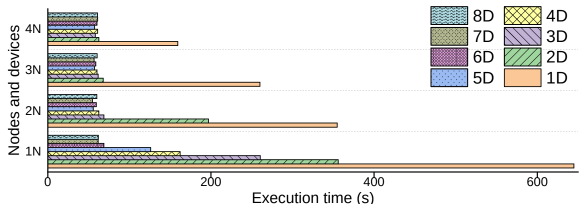
bandwidth than pattern 2, and thus is preferred by programmers to implement OpenCL kernels. We compare the performance of these patterns with PbSgemm, and see that using pattern 1 can yield a speedup of 17.74x.

5.6 Evaluating the Scalability of MOCL

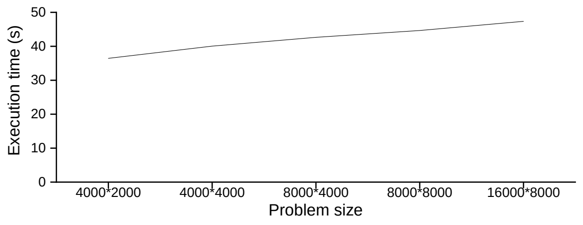
We evaluate the scalability of MOCL on either multiple Matrix-2000s or multiple compute nodes, each with multiple devices.

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 16, 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 17(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 17(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.



(a) Strong scalability



(b) Weak scalability

Figure 17: Evaluate the scalability of Clover across computing nodes.

6 RELATED WORK

There exist various OpenCL implementations. On the one hand, we notice that most vendor implementations are closed-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 atomics capability [4]. Meanwhile, the OpenCL implementation from TI is customized to TI SoCs (an ARM CPU + a TI DSP) [5]. 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]. POCL is a CPU-oriented OpenCL implementation built on Clang and LLVM. In addition, POCL supports the TTA and HSA architecture [20]. As of April 2017, POCL 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 POCL, FreeOCL also supports a large range of multi-core CPUs with the help of the generic C++ compilers [6]. 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 [19], 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 [25], 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 [29], 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) [9], 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 [26, 27], Shen et al. compare the performance of OpenCL and OpenMP on three x86_64 multicores. They identify the factors that significantly impact 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.

There is also work on OpenCL code generation [8, 18, 31, 32] and mapping [7, 17, 30, 33]. These approaches are complementary to our code generation framework.

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 passes 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-source OpenCL implementation. We share our experience on code optimization on Matrix-2000, providing useful insights on how to effectively program this unique architecture.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their constructive feedback. We also thank the National Supercomputing Center in Guangzhou for providing access to the machines with Matrix-2000. We thank the POCL developers (Pekka Jääskeläinen and James Price) for their help of understanding and running POCL. This work was partially funded by the National Key R&D Program of China under Grant No. 2017YFB0202003, the National Natural Science Foundation of China under Grant No.61602501; the UK Engineering and Physical Sciences Research Council under grants EP/M01567X/1 (SANDeRs) and EP/M015793/1 (DIVIDEND); and the Royal Society International Collaboration Grant (IE161012). For any correspondence, please contact Jianbin Fang (Email: j.fang@nudt.edu.cn)

REFERENCES

[1] 2017. Beignet OpenCL. https://www.freedesktop.org/wiki/Software/Beignet/. (2017).
[2] 2017. GalliumCompute OpenCL. https://dri.freedesktop.org/wiki/GalliumCompute/. (2017).
[3] 2017. Pocl OpenCL. http://pocl.sourceforge.net/. (2017).
[4] 2017. ROCm OpenCL. https://radeonopencompute.github.io/index.html. (2017).
[5] 2017. TI OpenCL. http://git.ti.com/opencl. (2017).
[6] February, 2017. FreeOCL OpenCL. http://www.zuzuf.net/FreeOCL/. (February, 2017).
[7] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end Deep Learning of Optimization Heuristics. In *PACT '17*.
[8] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *CGO '17*.
[9] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *GPGPU '10*.
[10] Jack Dongarra. 2017. *Report on the TianHe-2A System.* Technical Report ICL-UT-17-04.
[11] Jianbin Fang. Implementing and Evaluating OpenCL on an ARMv8 Multi-Core CPU. In *ISPA '17*.
[12] Jianbin Fang and others. A Comprehensive Performance Comparison of CUDA and OpenCL. In *ICPP '11*.
[13] Jianbin Fang, Henk Sips, Pekka Jaaskelainen, and Ana Lucia Varbanescu. Grover: Looking for Performance Improvement by Disabling Local Memory Usage in OpenCL Kernels. In *ICPP '14*.
[14] Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. 2014. Aristotle: A performance impact indicator for the OpenCL kernels using local memory. *Scientific Programming* 22, 3 (2014), 239–257.
[15] Jianbin Fang, Henk Sips, Lilun Zhang, Chuanfu Xu, Yonggang Che, and Ana Lucia Varbanescu. Test-driving Intel Xeon Phi. In *ICPE '14*.
[16] Jianbin Fang, Ana Lucia Varbanescu, Jie Shen, and Henk Sips. ELMO: A User-Friendly API to Enable Local Memory in OpenCL Kernels. In *PDP '14*.
[17] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. OpenCL task partitioning in the presence of GPU contention. In *LCPC '13*.
[18] Dominik Grewe, Zheng Wang, and Michael F.P. O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *CGO '13*.
[19] Jayanth Gummaraju and others. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *PACT '10*.
[20] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming* (2015).
[21] Ralf Karrenberg and Sebastian Hack. Improving Performance of OpenCL on CPUs. In *CC '12*.
[22] Khronos OpenCL Working Group 2012. *The OpenCL Specification V1.2.* Khronos OpenCL Working Group.
[23] Jungwon Kim and others. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *ICS'12*.
[24] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04*.
[25] Jaejin Lee and others. An OpenCL framework for heterogeneous multicores with local memory. In *PACT '10*.
[26] Jie Shen and others. Performance Gaps between OpenMP and OpenCL for Multi-core CPUs. In *ICPPW '12*.
[27] Jie Shen and others. 2013. An application-centric evaluation of OpenCL on multi-core CPUs. *Parallel Comput.* (2013).
[28] John A. Stratton and others. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *CGO '10*.
[29] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In *LCPC '08*.
[30] Ben Taylor, Vicent Sanz Marco, and Zheng Wang. Adaptive Optimization for OpenCL Programs on Embedded Heterogeneous Systems. In *LCTES '17*.
[31] Zheng Wang, Dominik Grewe, and Michael F. P. O'boyle. 2014. Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems. *ACM TACO* (2014).
[32] Zheng Wang, Daniel Powell, Björn Franke, and Michael O'Boyle. Exploitation of GPUs for the parallelisation of probably parallel legacy code. In *CC '14*.
[33] Yuan Wen, Zheng Wang, and Michael FP O'Boyle. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *HiPC '14*.