2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)

Implementing and Evaluating OpenCL on an ARMv8 Multi-Core CPU

Jianbin Fang*, Peng Zhang*, Tao Tang*, Chun Huang*, Canqun Yang*
*Software Institute, College of Computer, National University of Defense Technology, Changsha, China
Email: {j.fang, zhangpeng13a, taotang84, chunhuang, canqun}@nudt.edu.cn

***Abstract*—The OpenCL standard allows targeting a large variety of CPU, GPU and accelerator architectures using a single unified programming interface and language. But guaranteeing portability relies heavily on platform-specific implementations. In this paper, we provide an OpenCL implementation on an ARMv8 multi-core CPU, which efficiently maps the generic OpenCL platform model to the ARMv8 multi-core architecture. With this implementation, we first characterize the maximum achieved arithmetic throughput and memory accessing bandwidth on the architecture, and measure the OpenCL-related overheads. Our results demonstrate that there exists an optimization room for improving OpenCL kernel performance. Then, we compare the performance of OpenCL against serial codes and OpenMP codes with 11 benchmarks. The experimental results show that (1) the OpenCL implementation can achieve an average speedup of 6× compared to its OpenMP counterpart, and (2) the GPU-specified OpenCL codes are often unsuitable for this ARMv8 multi-core CPU.**

***Keywords*-OpenCL; FT-1500A; performance; programming**

I. Introduction

Nowadays platforms often incorporate specialized processing capabilities (e.g., GPUs, MICs, FPGAs or DSPs) to handle particular tasks. Although using such specialized processing units (i.e., SPU) gains in performance and/or energy efficiency, programming these units is challenging. In particular, programmers have to use vendor-specific programming interfaces (e.g., CUDA for NVIDIA GPUs) to exploit the architectual diversity, and they have to learn a different programming interface when moving to a new platform. As an alternative, leveraging a portable programming interface such as OpenCL is regarded as a promising approach [10].

With the prevalence of heterogeneous systems comprising both CPU and accelerators, we argue that it is of great significance for the heterogeneous programming interface to support both CPU and SPU as accelerator targets. First, a large amount of OpenCL legacy code can run seamlessly on the CPU-only environment and avoid the need of painful code porting. Second, CPU features more and more hardware cores and memory bandwidth, minimizing the gap between CPU and its accelerator counterpart [9]. At the same time, the cost of moving data between OpenCL host and devices can be avoided. Third, it is critical for a heterogeneous programming interface to manage the CPU and the SPU cores under an umbrella. In this way, programmers or runtimes can make a decision of how to distribute workloads: CPU-only, SPU-only, or both. These motivate the need for the heterogeneous programming interfaces to support not only SPU, but the traditional CPU as well.

FT-1500A is a 16-core ARMv8-based processor by the Phytium corporation and works in a CPU-only form (Section II-B). In this paper, we present an efficient OpenCL implementation on the FT-1500A CPU, which maps the OpenCL device model to the ARMv8 multi-core architecture. This implementation consists of a kernel compiler and an OpenCL runtime: the kernel compiler (based on LLVM) translates an OpenCL kernel into the ARMv8 binary code, and the runtime schedules commands in queues to execute on the multiple processing cores of FT-1500A.

We characterize our implementation in terms of *compiling overheads*, *arithmetic throughput*, and *device memory bandwidth*. The results demonstrate that there exists an optimization space of further reducing the compiling overheads and generating more efficient kernel binaries (e.g., inter-thread vectorization). Then we use real-world applications from the `Parboil` suite to compare the performance of our OpenCL with the serial code and the OpenMP code [22]. Our experimental results show that our OpenCL implementation performs on average 6× faster than its OpenMP counterpart. The performance improvements largely come from the efficient implementation of *critical* regions and the improved utilization of data locality. The results also show that the OpenCL codes run faster than the serial codes for the majority of the benchmarks. Among the eleven benchmarks, we notice that three OpenCL codes perform worse than the serial codes. This is due to the GPU-specified optimizations embedded in the OpenCL codes (e.g., using local memory and/or coalesced memory accesses).

To summarize, we make the following contributions.

- We provide an efficient implementation of the OpenCL programming standard based on the LLVM compiling infrastructure for the FT-1500A CPU (Section III).
- We characterize the implementation (overheads and device capability) on FT-1500A compared to Intel's OpenCL implementation on E5-2620 (Section IV).
- We evaluate the performance of our OpenCL implementation against the serial code and the OpenMP counterpart with 11 benchmarks (Section V).

Although our OpenCL framework is implemented on FT-1500A, we argue that it is equally applicable to other ARMv8-based multi-core CPUs. Our framework is also extensible to ARMv8-based accelerators by free-riding LLVM.

0-7695-6329-5/17/31.00 ©2017 IEEE
DOI 10.1109/ISPA/IUCC.2017.00131

II. Background

In this section, we introduce the OpenCL programming interface and describe FT-1500A's architecture details.

A. Open Computing Language

Open Computing Language (OpenCL) is a relatively new standard for parallel programming of many-core architectures [23]. Addressing functional portability, 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 graphics processing units (GPUs), 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*.

To maximize hardware utilization, OpenCL provides *vector data type* and related built-in functions. The vector data type is defined with the type name, i.e., `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `float`, `long`, `ulong`, followed by a literal value n that defines the number of elements in the vector. Thus, a vector is a fixed-length collection of scalar data elements, e.g., `float4` is a vector that contains 4 elements typed `float`. Using vector types is regarded as explicit and high-level vectorization [11].

B. The FT-1500A CPU

FT-1500A (codenamed "Earth") is a family of a 64-bit ARMv8-based multi-core system on chips developed by Phytium and launched in 2016 [25]. It incorporates multiple Xiaomi FTC660 cores, a customized implementation of the ARMv8 specification and is manufactured with a 28nm process. The exact number of cores varies by models. The FT-1500A used in this context has 16 cores and runs at 1.5 GHz, as shown in Figure 1. And SIMD instructions are available to manipulate 128-bits data at a time. Therefore, the theoretical peak floating point performance is 192 Gflops/s in single precision and 96 Gflops/s in double precision.

On FT-1500A, each core has a private L1 data cache sized of 32 KB and a private L1 instruction cache sized of 48 KB. An L2 cache (2 MB) is shared by every four cores (Figure 1). Then all the cores share a 8 MB last-level cache [26]. Besides, the chip has four DDR3-1600 memory controllers, which can deliver a total of 51.2 GB/s memory access bandwidth. It has two $\times$16 or four $\times$8 PCIE Gen3 interfaces, and two GEthernet interfaces. Its maximum power consumption is of 35 Watts. This processor is used as OA servers, web servers and cloud computing servers, etc. to run web services, transaction processing, datacenter storage, database and network switching workloads.

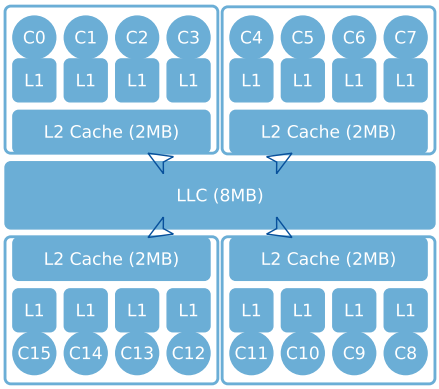


Figure 1. The conceptual view of the FT-1500A architecture. *Cx* represents the x^{th} hardware core and *LLC* represents the last-level cache.

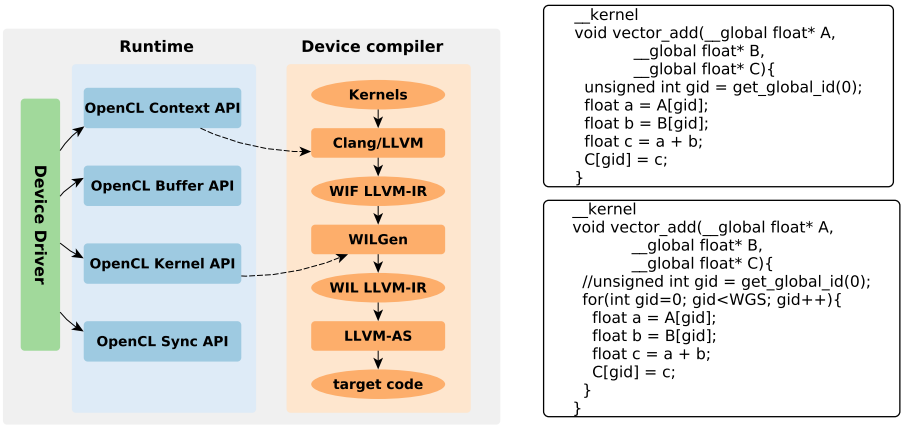
As for the software part, FT-1500A runs `Linux kylin v3.14.0` and supports the OpenMP programming model (`v3.1`). But it has no support of OpenCL yet, which prevents us from running the OpenCL-compatible software. Therefore, we take FT-1500A as a case to demonstrate how we customize OpenCL on an ARMv8 multi-core CPU.

III. Design and Implementation

In this section, we present the design and the implementation details of OpenCL on FT-1500A. The overall design is shown in Figure 2. Our compiling infrastructure on FT-1500A consists of two parts: the kernel compiler and the runtime system (Section III-B and III-C). The kernel compiler aims to compile the kernels into device-specific binaries, whereas the runtime implements the OpenCL APIs [9].

A. Overall Mapping

The OpenCL standard allows targeting a large range of accelerating architectures using a single unified programming interface and language. This is achieved by efficiently mapping the OpenCL platform model (Section II-A) to the underlying hardware. Overall, we map each hardware core as a compute unit, each of which further has one processing element. Note that the processing element can deal with a 128-bit vector one time. In this way, work-groups in the NDRange are scheduled to different cores and run in a concurrent manner. All the work-items in the same work-group are executed on the same core sequentially [8]. We



(a) The framework (b) Work-item loop

Figure 2. The compiling infrastructure on FT-1500A: (a) shows the overall framework, and (b) showcases an example of performing the work-item loop transformation.

choose not to execute a work-item on a core and map the work-items of a work-group onto multiple cores, because a significant context switching overhead would be introduced.

OpenCL defines a hierarchical device memory model: private memory, local memory, constant memory, and global memory (Section II-A). All the memory regions are allocated in the main memory. Since only one work-group is active on a core, we allocate one local memory per core and reuse it for all the work-groups. This approach reduces off-chip memory bandwidth usage, efficiently utilizes caches, and saves expensive allocations [9]. Besides, constant buffers are marked read-only by our runtime system.

B. Kernel Compiler

Many researchers have proposed compilation techniques for OpenCL kernels to CPUs, where work-groups are scheduled to run on distinct hardware threads [9, 14, 15, 16, 17, 20]. Further, scheduling work-items within a work-group to execute on a hardware thread is done either by wrapping a region with a *work-item loop* [14, 16, 17, 20], or by using *user-level threads* [9]. In this context, we use the *work-item-loop* scheduling approach for reduced overheads. Figure 2(b) shows an example kernel function before and after performing the work-item loop transformation. This is achieved by using nested loops around the kernel body to replace the execution of work-items within a work-group.

Figure 2(a) shows how we transform application kernels into device-specific binaries on FT-1500A. The compiler uses a C front-end along with the LLVM framework with extensions to support OpenCL. Thus, our compiler takes an OpenCL C kernel as input and translates it into the LLVM IR format for a single work-item (`WIF LLVM-IR`). This IR kernel is then transformed into work-item loops (`WIL LLVM-IR`), as illustrated in Figure 2(b). The LLVM-IR is then optimized using standard optimization passes provided in LLVM [9]. Finally, the target binary is generated with `LLVM-AS` and the system linker.

When producing `WIL`, we have to respect the synchronization semantics of the work-group *barriers* inside the kernel source code [17]. In this work, we partition the code with barriers into separate *regions*. Each code region is transformed into a `WIL` and scheduled to run in parallel. At the same time, the work-item loops are annotated with LLVM metadata that retains the information of parallel iterations. In this way, we can exploit the metadata and rely on the mature compiler infrastructure to optimize `WIL`.

C. Runtime System

The runtime system needs to implement the OpenCL APIs: the context APIs, the buffer APIs, the kernel APIs, and the synchronization APIs (Figure 2(a)). The interaction between host and device is performed in terms of *commands*, which are broadly categorized into *kernel execution commands*, *data transferring commands* and *synchronization commands*. The host issues commands to devices, and the commands are put into *command queues*. Devices fetch commands from the queue and perform the corresponding actions to finish the offloading work. Specifically, *data transferring commands* are executed on the device using the `memcpy()` function. Thus, executing this command is that we move data elements from one buffer to another in the main memory of FT-1500A. When executing *kernel commands*, the host will launch the kernel function and dispatch the predefined work-groups onto the spawned threads (and hardware cores). Our API implementations of the OpenCL framework are generic in C/C++, and thus are portable [17]. The runtime system calls the kernel compiler when building kernel code and right before launching kernels.

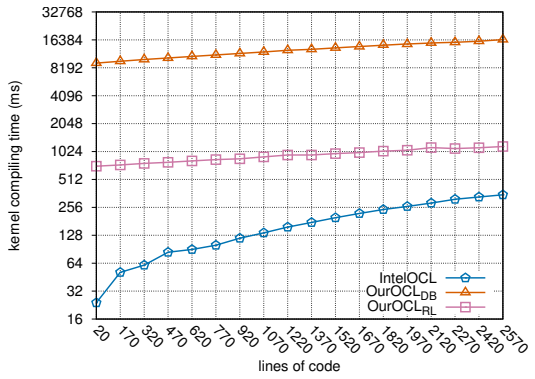


Figure 3. Comparing the compiling overheads on FT-1500A and E5-2620.

```
__kernel
void arith_speed(__global float* input,
                 __global float* output,
                 unsigned int iterations){
  unsigned int i;
  unsigned int gid = get_global_id(0);
  float a = (float)(input[0]);
  float b = (float)(input[1]);
  float tmp;
  for(i = 0; i < iterations; ++i) {
    tmp = a + b; a = b; b = tmp;
    tmp = a + b; a = b; b = tmp;
    ......
    tmp = a + b; a = b; b = tmp;
  }
  output[gid] = a;
}
```

Figure 4. A microbenchmark to measure the maximum achieved flops [19].

IV. Characterizing the Implementation

In this section, we first characterize the implementation overhead (the kernel start overhead and the kernel compiling overhead), and then the measure the device capability (the maximum achieved arithmetic throughput and memory accessing bandwidth) with our OpenCL implementation.

A. Characterizing Overheads

We compare the overheads of two OpenCL implementations: one is our implementation on FT-1500A, and the other is the one from Intel OpenCL `v16.1.1` on a dual-socket Intel Xeon E5-2620 v3 (short for E5-2620 thereafter). We consider the overheads from two aspects. First, we measure the kernel start overhead (the elapsed time between the `CL_PROFILING_COMMAND_QUEUED` and `CL_PROFILING_COMMAND_START` events) for these two platforms. For Intel OpenCL on E5-2620, this invocation overhead is around 8 microseconds. Meanwhile, it is 13.8 microseconds for the OpenCL implementation on FT-1500A. Launching an kernel is performed in a serial manner and thus the difference of the kernel start overhead results from the clocking difference, i.e., it is 1.5 GHz on FT-1500A versus 2.4 GHz on E5-2620.

Second, we compare the kernel compiling overheads (the elapsed time taken by `clBuildProgram` and `clCreateKernel`) as shown in Figure 3. We see that the compiling overheads increase over the lines of code on both platforms. On E6-2620, the compiling pass can be finished within 0.5 seconds. Meanwhile, the compiling overhead is up to 16 seconds on FT-1500A with LLVM compiled in the `debug` mode. When in the `release` mode, the compiling time is as long as one second. However, compiling the kernels with very few lines of code still takes much more time on the FT-1500A than on the E5-2620 CPU.

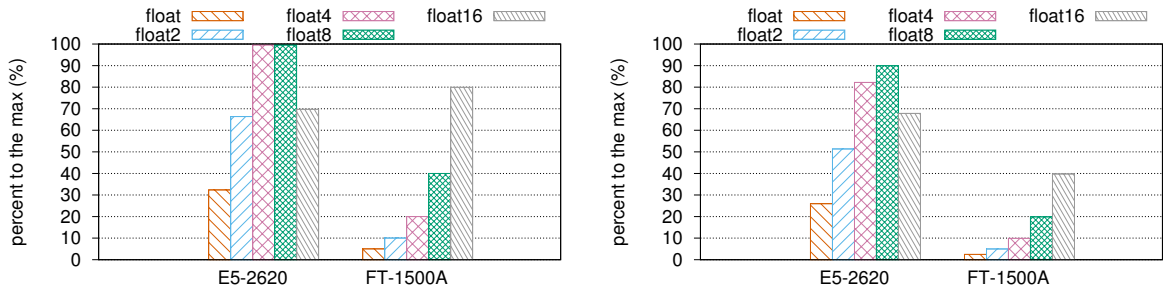
B. Characterizing the Device

1) Arithmetic Throughput: We measure the maximum achieved floating point operations per second (*flops*) on both FT-1500A and E5-2620. The microbenchmark is shown in Figure 4. Note that the statements between lines are dependent to avoid compiler optimizations. The results are obtained in two cases: (1) with a single work-group to occupy a single hardware core, and (2) with abundant work-groups to occupy all hardware cores.

Figure 5(a) shows the number of floating point additions per second measured on a single hardware core. Note that the results are normalized to that of the theoretical maximum flops which is calculated from the hardware specifications (i.e., a single core can run 6 GFLOPS on FT-1500A and 19.2 GFLOPS on E5-2620)[1]. With a single work-group in the global range, using `float4` or `float8` can obtain a maximum flops, peaking at 19.1 GFlops on E5-2620. This number is very close to its theoretical roof. Using `float` can achieve 32% of the upper-bound and using `float2` can reach 66% of the upper-bound on E5-2620. In such cases, the data vectors are too short to utilize the 256-bit SIMD lanes. This also indicates that the *auto-vectorizer* module of the Intel compiler does not work in this case [21]. When using `float16` on E5-2620, the data vectors are longer the SIMD lanes so that we have to split data vectors to fit the hardware lanes. We argue that this partitioning overhead leads to a decrease of the achieved flops.

Figure 5(a) also shows that, the longer the vectors are, the larger the achieved flops is, on FT-1500A. We can achieve 5% and 80% of the maximum flops on a single core when using `float` and `float16`, respectively. Due to the data dependency between statements (Figure 4), each statement consumes 5 cycles to produce results for its following statement. In such a case, the work from work-items is executed in a serial manner on a hardware core (20%) and one out of four lanes are used (25%). Therefore, we can obtain $5\% = 25\% \times 20\%$ with `float` on FT-1500A. This low hardware utilization can be mitigated by exploiting long

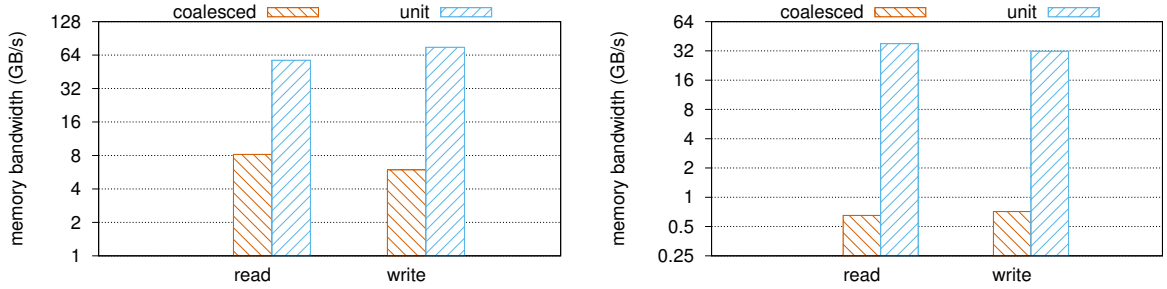
[1]On FT-1500A and E5-2620, we consider only `additions`, rather than the `multiply-add` operations. The overclocking feature is disabled on E5-2620. Thus, the pipeline will execute one floating operation per cycle.



(a) with a single workgroup

(b) with abundant workgroups

Figure 5. Comparing the arithmetic throughput on E5-2620 and FT-1500A. We use the maximum allowed number of work-items per work-group and repeat each run 100 times to obtain the mean value.



(a) E5-2620

(b) FT-1500A

Figure 6. The achieved memory bandwidth. We use a work-group sized of 512 on E5-2620 while using a workgroup sized of 64 on FT-1500A.

data vectors. This is why we observed a better performance when using vector data types on FT-1500A. In particular, although the length of `float8` or `float16` is larger than that of SIMD lanes (128 bits) on FT-1500A, we notice a further performance improvement.

When there are abundant work-groups, all the hardware cores are utilized (Figure 5(b)). On E5-2620, we notice that a close-to-bound flops (i.e., 90%) can be achieved when using `float8`. This is due to the fact that the SIMD lane is of 256 bits in width, which fits 8 floating data elements. Meanwhile, using `float16` incurs overheads of splitting the long vector. On FT-1500A, we see that the maximum achieved flops increases over vector length. The maximum achieved flops is only around 40% of its theoretical bound. The reasons are similar to that of a single work-group. For future work, we plan to explore vectorization and instruction pipeline overlapping between work-items.

2) Device Memory Bandwidth: We use two microbenchmarks to measure the device memory bandwidth: `coalesced` and `unit` [7]. The `coalesced` benchmark lets neighboring work-items in a work-group access the data elements close to each other while the `unit` benchmark lets a work-item read/write a continuous piece of data elements. The `unit` benchmark allows that neighboring work-items access data elements that are far from each other. Our experience shows that GPUs prefer accessing memory space in a *coalesced* manner [10]. In this section, we evaluate how the two benchmarks perform on FT-1500A and E5-2620.

Figure 6 compares the achieved memory bandwidth (`read` and `write`) on E5-2620 and FT-1500A. On E5-2620, reading data is 7× faster in the unit manner than in the coalesced manner, while writing data is 13× faster. On FT-1500A, we notice a speedup of 58× and 45× for reads and writes, respectively, in the unit fashion. Therefore, both FT-1500A and E5-2620 can achieve a much larger memory bandwidth in the *unit* manner, which differs from GPUs. This is due to the fact that threads prefer accessing contiguous data elements and can better exploit data locality on both CPUs. Further, our experimental results with the `stream` benchmarks peaks at 20 GB/s [24]. Using the OpenCL code on FT-1500A can obtain a memory bandwidth of 38 GB/s, which means the explicitly tiled code enables faster memory accesses. To conclude, the unit accessing fashion, rather than the coalesced accessing fashion, is preferred on the FT-1500A CPU.

V. Performance Comparision

In this section, we evaluate the performance of our OpenCL implementation with the `Parboil` benchmark suite. The benchmarks include `bfs`, `stencil`, `lbm`, `histo`, `sgemm`, `mri-q`, `cutcp`, `tpacf`, `mri-gridding`, `spmv`, and `sad`. We compare the

OpenCL performance with that of the serial code and the OpenMP code on the FT-1500A CPU. The evaluation includes two parts: (1) comparing the performance of the OpenCL code (with 16 cores) and the serial code (with one core), and (2) comparing the performance of the OpenCL code against its OpenMP counter part on 16 cores. We use `GCC v6.2 (-O3)` to compile the serial code and the OpenMP code and use our OpenCL implementation for the OpenCL code. When running OpenCL codes, we use the default work-group configurations. Besides, we run each experiment five times and calculate the medium value.

A. OpenCL Code versus Serial Code

Figure 7(a) shows the OpenCL performance against its serial counter part. We see that the OpenCL code significantly outperforms the serial code for `stencil`, `tpacf`, `lbm`, `mri-q`, `sgemm`, `spmv`, `cutcp`, and `sad`. The speedup varies over benchmarks and datasets. For most benchmarks and datasets, we notice that the speedup is less than 16. This is expected since FT-1500A has a total of 16 hardware cores. We also see that the OpenCL version of `stencil` runs 112× as fast as the serial code with the `large` dataset. This application works on a 3D data grid iteratively: the serial version reads data in the order of $Z \rightarrow Y \rightarrow X$, whereas the data elements in the X dimension are physically located close to each other. This noncontiguous memory accesses are detrimental to data locality on FT-1500A. For the OpenCL code, this three dimensional loop is transformed into a 3D grid of work-items. Among them, the ones in the Z dimension are neighbors, but they will access the memory in a *unit* manner (Section IV-B2). Therefore, this large speedup is not only because the OpenCL code can use all the hardware cores, but the OpenCL `stencil` code can access the memory space in a cache-friendly manner on FT-1500A. Besides, the speedup is very limited for the `small` datasets (e.g., `spmv` and `sad`). This is due to the fact that spawning threads in OpenCL is an overhead compared to that of the serial codes.

On the other hand, our OpenCL's performance is poor on the FT-1500A CPU for `mri-gridding`, `bfs`, and `histo`. `histo` is a histogramming operation that accumulates the number of occurrences of each output value in the input data set [22]. The output histogram is a two-dimensional matrix of char-type bins that saturate at 255. This benchmark is intensively optimized and customized for GPUs. First, loading the data elements of an image from global memory is performed in a coalesced manner, which is preferred by GPUs, rather than the ARMv8 CPU. Second, local memory is used to stage the temporal histograming results. This type of memory space is mapped to on-chip scratch-pads in GPUs, which is unavailable on FT-1500A. Actually, this memory space is allocated in the global memory space in our OpenCL implementation. In this case, staging data in local memory often degrades the overall performance.

Another case that the OpenCL code runs remarkably worse than the serial code is `bfs`. This application takes every node in the current frontier and enqueues all unexplored neighbors to the next frontier. This process iterates until all the nodes in the graph have been visited [22]. Thus, this requires the traversing kernel to be launched iteratively and the kernel launch overheads become nonnegligible. Besides, the OpenCL code manages a hierarchical queue system [22]. Similar to `histo`, local memory is used in `bfs` to stage the local queue of frontiers. To summarize, OpenCL performs worse than the serial code for the GPU-customized code.

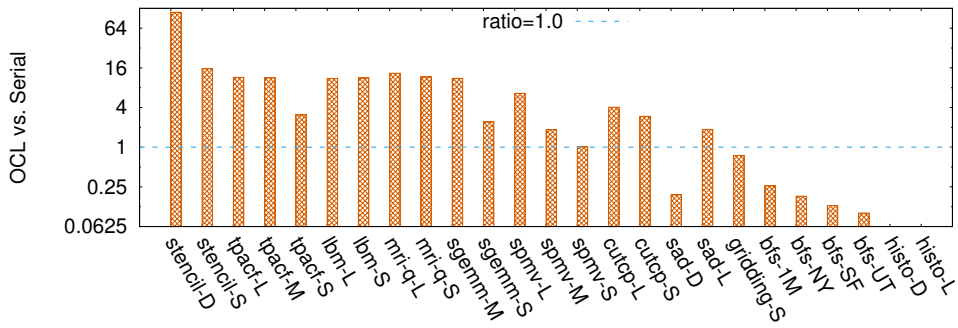
B. OpenCL Code versus OpenMP Code

Figure 7(b) compares the performance of OpenCL versus OpenMP on FT-1500A. We see that OpenCL outperforms OpenMP for the majority of the benchmarks and datasets. The OpenCL code performs, on average, 6× as fast as the OpenMP code. For `bfs`, OpenCL runs significantly faster than OpenMP. When looking into their code, we notice that the OpenMP version uses the `critical` clause such that all threads execute this code region in a serial manner. Therefore, the threaded code runs slower than the one with a single thread. Meanwhile, the OpenCL code avoids this issue by using `atomic` operations. It is the same for `tpacf`, `mri-gridding`, and `histo`, where the accumulation code is put into the `critical` region.

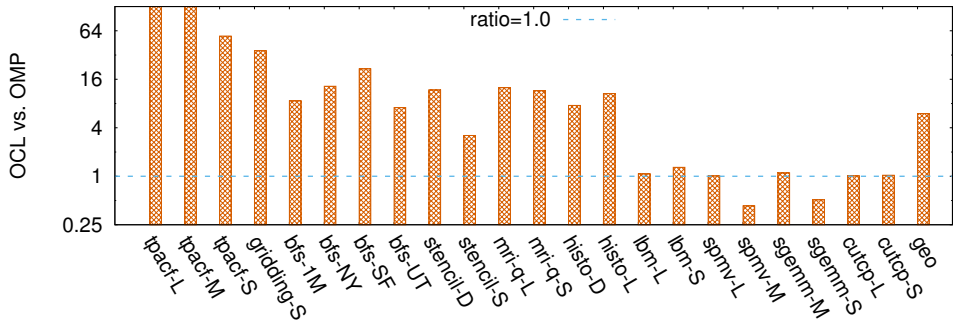
For the OpenCL version of `mri-q`, we also notice a remarkable performance improvement compared to the OpenMP version. This application takes a majority of the execution time to calculate the *sin* and *cos* operations. We argue that the performance boost comes from the customized mathematical *built-ins* [17]. To achieve good performance for the computationally bounded kernels, we have a vectorized implementation for these *built-in* functions. Besides, the OpenCL version exploits registers where each work-item works on four data elements. This technique will improve data locality and task granularity at the register level.

For `stencil`, the OpenCL version performs much better than its OpenMP counterpart. The reason is the same as that of the serial code, where the degraded performance is due to the noncontinuous memory accesses. Meanwhile, the OpenCL code can avoid this by transforming the three dimensional loop into a 3D grid of work-items with the *unit* manner of memory accesses. For the OpenMP code, interchanging the loop index can significantly minimize the performance gap with the OpenCL code.

To summarize, the applications in OpenCL perform well on the FT-1500A CPU, and can achieve a competitive performance to the OpenMP version. Our code runs particularly fast for `tpacf`, `mri-gridding`, `bfs`, `stencil`, `hist`, and `mri-q`. This is due to the uage of the *critical* pragma and the inefficient memory accesses in the OpenMP codes. To minimize the performance gaps, we have to apply the optimizations embedded in the OpenCL codes.



(a) OpenCL codes versus serial codes



(b) OpenCL codes versus OpenMP codes

Figure 7. A comprehensive performance comparison of the OpenCL code versus (a) the serial code (b) the OpenMP code on FT-1500A with the `Parboil` benchmark suite. The y-axis represents the performance ratio between the OpenCL code and its serial counter part. OpenCL performs better when $ratio > 1.0$. All the available datasets are used in our experiment where 'L', 'M', 'S', and 'D' denote `Large`, `Medium`, `Small`, and `Default`, respectively. Since there is no OpenMP version for `sad`, we omit this one when comparing OpenCL against OpenMP.

VI. Related Work

There exist a variety of OpenCL implementations. 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 atomics capability [5]. 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 [3]), and it builds on the work of the Mesa project to support multiple platforms [4]. BEIGNET is an implementation released by Intel in 2013 for its GPUs (Ivy Bridge and newer) [1]. POCL is a CPU-only OpenCL implementation built on Clang and LLVM. In addition to the traditional CPUs, POCL supports the TTA and HSA architecture [17, 18]. Similar to POCL, FreeOCL also supports a large range of multi-core CPUs with the help of the generic C++ compilers [2]. But this implementation is indeed CPU-oriented and cannot be extended to accelerators in a straightforward way. Considering POCL's portable capability and its potential of supporting an accelerating device, we choose to port and extend POCL onto FT-1500A.

To characterize and compare the OpenCL performance of existing and future devices, Thoman et al. propose a suite of microbenchmarks [19]. These microbenchmarks are used to measure quantities such as arithmetic throughput, the bandwidth and latency of various address spaces, and the dynamic branching penalties on many-core architectures. In this context, we choose to use the microbenchmarks on arithmetic throughput and compiling overheads. To compare and contrast architectural designs and programming systems in a fair and open forum, Danalis et al. have designed the Scalable HeterOgeneous Computing benchmark suite (SHOC) [7], which is a spectrum of programs that test the performance and stability of these scalable heterogeneous computing systems. At the low level, SHOC uses microbenchmarks to assess architectural features of the system: compute units and memory hierarchy. At the high

level, SHOC uses application kernels to determine system-wide performance including many system features such as intranode and internode communication among devices. In this context, we use the microbenchmarks to measuring the bandwidth when accessing different memory space.

In [12, 13], 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 FT-1500A CPU, we have similar observations that the GPU-customized OpenCL codes perform even worse than the serial code. This motivates us to generates efficient codes for FT-1500A from the OpenCL codes with GPU-specific optimizations in the future.

VII. Conclusion

In this paper, we present an implementation for the OpenCL standard on the FT-1500A CPU. This not only facilitates the OpenCL codes to run on the ARMv8 CPU, but provides users with an open programming interface on this new architecture. Then, we characterize this OpenCL implementation in terms of overheads, arithmetic throughput, and memory bandwidth. We also evaluate the OpenCL implementation over that of the serial code and the OpenMP codes on the FT-1500A CPU. Our experimental results show that our OpenCL implementation overtakes the serial codes for the majority of the benchmarks and performs on average $6\times$ as fast as its OpenMP counterpart. This performance advantage demonstrates the efficiency of our OpenCL implementation. In the future, we will investigate how to generate efficient code from the GPU-specific inputs. Also, we would like to analyze how to futher optimize the kernel compiler (e.g., `inter-work-item vectorization`).

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments. This work was funded by the National Natural Science Foundation of China under Grant No.61402488, No.61502514 and No.61602501, the National Key R&D Program of China under Grant No. 2017YFB0202003.

References

[1] Beignet opencl. https://www.freedesktop.org/wiki/Software/Beignet/, February, 2017.

[2] Freeocl opencl. http://www.zuzuf.net/FreeOCL/, February, 2017.

[3] Galliumcompute. https://dri.freedesktop.org/wiki/GalliumCompute/, February, 2017.

[4] Galliumcompute opencl. https://dri.freedesktop.org/wiki/GalliumCompute/, February, 2017.

[5] Rocm opencl. https://radeonopencompute.github.io/index.html, February, 2017.

[6] Ti opencl. http://git.ti.com/opencl, February, 2017.

[7] Anthony Danalis et al. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU'2010, Pittsburgh, Pennsylvania, USA, March 14, 2010*, pages 63–74, 2010.

[8] Gangwon Jo et al. Opencl framework for ARM processors with NEON support. In *WPMVP'2014, Orlando, Florida, USA, February 16, 2014*, pages 33–40, 2014.

[9] Jayanth Gummaraju et al. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *PACT'2010, Vienna, Austria, September 11-15, 2010*, pages 205–216, 2010.

[10] Jianbin Fang et al. A comprehensive performance comparison of cuda and opencl. In *ICPP'2011, Taipei, Taiwan, September 13-16, 2011*, pages 216–225, 2011.

[11] Jianbin Fang et al. Evaluating vector data type usage in opencl kernels. *Concurrency and Computation: Practice and Experience*, 27(17):4586–4602, 2015.

[12] Jie Shen et al. Performance gaps between openmp and opencl for multi-core cpus. In *ICPPW'2012, Pittsburgh, PA, USA, September 10-13, 2012*, pages 116–125, 2012.

[13] Jie Shen et al. An application-centric evaluation of opencl on multi-core cpus. *Parallel Computing*, 39(12):834–850, 2013.

[14] John A. Stratton et al. MCUDA: an efficient implementation of CUDA kernels for multi-core cpus. In *LCPC'2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, pages 16–30, 2008.

[15] John A. Stratton et al. Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In *CGO'2010, Toronto, Canada, April 24-28, 2010*, pages 111–119, 2010.

[16] Jungwon Kim et al. Snucl: an opencl framework for heterogeneous CPU/GPU clusters. In *ICS'12, Venice, Italy, June 25-29, 2012*, pages 341–352, 2012.

[17] Pekka Jääskeläinen et al. pocl: A performance-portable opencl implementation. *International Journal of Parallel Programming*, 43(5):752–785, 2015.

[18] Pekka Jääskeläinen et al. pocl: A performance-portable opencl implementation. *CoRR*, abs/1611.07083, 2016.

[19] Peter Thoman et al. Automatic opencl device characterization: Guiding optimized kernel design. In *Euro-Par'2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*, pages 438–452, 2011.

[20] Ralf Karrenberg et al. Improving performance of opencl on cpus. In *CC'2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 1–20, 2012.

[21] Intel Inc. Intel OpenCL Implicit Vectorization Module. http://llvm.org/devmtg/2011-11/Rotem_IntelOpenCLSDKVectorizer.pdf, 2011.

[22] John A. Stratton et al. . Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, IMPACT Technical Report, IMPACT-12-01, University of Illinois at Urbana-Champaign, 2012.

[23] Khronos OpenCL Working Group. *The OpenCL Specification V1.2*, November 2012.

[24] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/, February, 2017.

[25] Phytium Technology Co. Ltd. *FT-1500A/16*, February 2017. http://www.phytium.com.cn/Product/detail?language=1&product_id=9.

[26] Phytium Technology Co. Ltd. *FT-1500A/16*, February 2017. https://en.wikichip.org/wiki/phytium/ft-1500a/ft-1500a-16.