Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 859491, 20 pages
http://dx.doi.org/10.1155/2015/859491

Research Article

OpenCL Performance Evaluation on Modern Multicore CPUs

Joo Hwan Lee, Nimit Nigania, Hyesoon Kim, Kaushik Patel, and Hyojong Kim

School of Computer Science, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA

Correspondence should be addressed to Joo Hwan Lee; joohwan.lee@gatech.edu

Received 15 May 2014; Accepted 29 September 2014

Academic Editor: Xinmin Tian

Copyright © 2015 Joo Hwan Lee et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Utilizing heterogeneous platforms for computation has become a general trend, making the portability issue important. OpenCL (Open Computing Language) serves this purpose by enabling portable execution on heterogeneous architectures. However, unpredictable performance variation on different platforms has become a burden for programmers who write OpenCL applications. This is especially true for conventional multicore CPUs, since the performance of general OpenCL applications on CPUs lags behind the performance of their counterparts written in the conventional parallel programming model for CPUs. In this paper, we evaluate the performance of OpenCL applications on out-of-order multicore CPUs from the architectural perspective. We evaluate OpenCL applications on various aspects, including API overhead, scheduling overhead, instruction-level parallelism, address space, data location, data locality, and vectorization, comparing OpenCL to conventional parallel programming models for CPUs. Our evaluation indicates unique performance characteristics of OpenCL applications and also provides insight into the optimization metrics for better performance on CPUs.

1. Introduction

The heterogeneous architecture has gained popularity, as can be seen from AMD's Fusion and Intel's Sandy Bridge [1, 2]. Much research shows the promise of the heterogeneous architecture for high performance and energy efficiency. However, how to utilize the heterogeneous architecture considering performance and energy efficiency is still a challenging problem. OpenCL is an open standard for parallel programming on heterogeneous architectures, which makes it possible to express parallelism in a portable way so that applications written in OpenCL can run on different architectures without code modification [3]. Currently, many vendors have released their own OpenCL framework [4, 5].

Even though OpenCL provides portability on multiple architectures, portability issues still remain in terms of performance. Unpredictable performance variations on different platforms have become a burden for programmers who write OpenCL applications. The effective optimization technique is different depending on the architecture where the kernel is executed. In particular, since OpenCL shares many similarities with CUDA, which was developed for NVIDIA GPUs, many OpenCL applications are not well optimized for modern multicore CPUs. The performance of general OpenCL applications on CPUs lags behind the performance expected by programmers considering conventional parallel programming models. The expectation comes from programmers' experience with conventional programming models. OpenCL applications show very poor performance on CPUs when compared to applications written in conventional programming models.

The reasons we consider CPUs for OpenCL compute devices are as follows.

(1) CPUs can also be utilized to increase the performance of OpenCL applications by using both CPUs and GPUs (especially when a CPU is idle).

(2) Because modern CPUs have more vector units, the performance gap between CPUs and GPUs has been decreased. For example, even for the massively parallel kernels, sometimes CPUs can be better than GPUs, depending on input sizes. On some workloads with high branch divergence or with high instruction-level parallelism (ILP), the CPU can also be better than the GPU.

A major benefit of using OpenCL is that the same kernel can be easily executed on different platforms. With OpenCL,

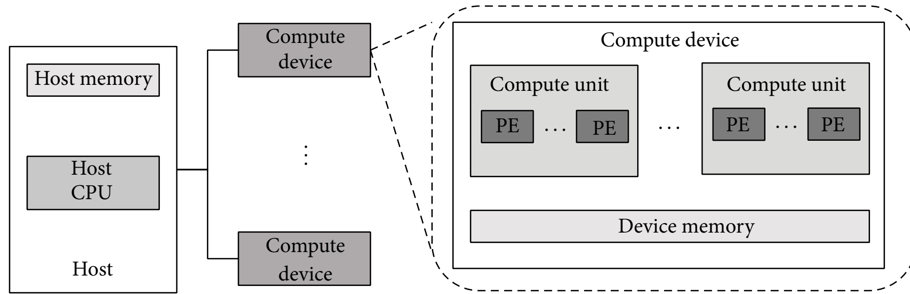


Figure 1: OpenCL platform model.

it is easy to dynamically decide which device to use at run-time. OpenCL applications that select a compute device between CPUs and GPUs at run-time can be easily implemented. However, if the application is written in OpenMP, for example, it is not trivial to split an application to use both CPUs and GPUs.

Here, we evaluate the performance of OpenCL applications on modern out-of-order multicore CPUs from the architectural perspective, regarding how the application utilizes hardware resources on CPUs. We thoroughly evaluate OpenCL applications on various aspects that could change their performance. We revisit generic performance metrics that have been lightly evaluated in previous works, especially for running OpenCL kernels on CPUs. Using these metrics, we also verify the current limitation of OpenCL and the possible improvement in terms of performance. In summary, the contributions of this paper are as follows.

(i) We provide programmers with a guideline to understand the performance of OpenCL applications on CPUs. Programmers can verify whether the OpenCL kernel fully utilizes the computing resources of the CPU.

(ii) We discuss the effectiveness of OpenCL applications on multicore CPUs and possible improvement.

The main objective of this paper is to provide a way to understand OpenCL performance on CPUs. Even though OpenCL can be executed on CPUs and GPUs, most previous work has focused on only GPU performance issues. We believe that our work increases the understandability of OpenCL on CPUs and helps programmers by reducing the programming overhead to implement a separate CPU-optimized version from scratch. Some previous studies about OpenCL on CPUs discuss some aspects presented in this paper, but they lack both quantitative and qualitative evaluations, making them hard to use when programmers want to estimate the performance impact of each aspect.

Section 2 describes the background and architectural aspects to understand the OpenCL performance on CPUs. Then, we evaluate OpenCL applications regarding those aspects in Section 3. We review related work in Section 4 and conclude the paper.

2. Background and Criteria

In this section, we describe the background of several aspects that affect OpenCL application performance on CPUs: API overhead, thread scheduling overhead, instruction-level parallelism, data transfer, data locality, and compiler autovectorization. These aspects have been emphasized in academia and industry to improve application performance on CPUs on multiple programming models. Even though most of the architectural aspects described in this section are well-understood fundamental concepts, most OpenCL applications are not written considering these aspects.

2.1. API Overhead. OpenCL has high overhead for launching kernels, which is negligible on other conventional parallel programming models for CPUs. In addition to the kernel execution on the compute device, OpenCL needs OpenCL API function calls in the host code to coordinate the executions of kernels that are overheads. The general steps of an OpenCL application are as follows [3]:

(1) Open an OpenCL context.

(2) Create a command queue to accept the execution and memory requests.

(3) Allocate OpenCL memory objects to hold the inputs and outputs for the kernel.

(4) Compile and build the kernel code online.

(5) Set the arguments of the kernel.

(6) Set workitem dimensions.

(7) Kick off kernel execution (enqueue the kernel execution command).

(8) Collect the results.

The complex steps of OpenCL applications are due to the OpenCL design philosophy emphasizing portability over multiple architectures. Since the goal of OpenCL is to make a single application run on multiple architectures, they make the OpenCL programming model as flexible as possible. Figure 1 shows the OpenCL platform model and how OpenCL provides portability. The OpenCL platform consists of a host and a list of compute devices. A host is connected to one or more compute devices and is responsible for managing resources on compute devices. The compute device is an abstraction of the processor, which can be any

type of processor, such as a conventional CPU, GPU, and DSP. A compute device has a separate device memory and a list of compute units. A compute unit can have multiple processing elements (PEs). By this abstraction, OpenCL enables portable execution.

On the contrary, flexibility for various platform supports does not exist on conventional parallel programming models for multicore CPUs. Many of the APIs in OpenCL, which take a significant execution time on OpenCL application do not exist on conventional parallel programming models. The compute device and the context in OpenCL are implicit on conventional programming models. Users do not have to query the platform or compute devices and explicitly create the context.

Another example of the unique characteristics of OpenCL compared to conventional programming models is the "just-in-time compilation" [6] during run-time. In many OpenCL applications, kernel compilation time by the JIT compiler incurs the execution time overhead. On the contrary, compilation is statically done and is not performed during application execution for the application written in other programming models.

Therefore, to determine the actual performance of applications, the time cost to execute the OpenCL API functions also should be considered. From evaluation, we find that the API overhead is larger than the actual computation in many cases.

2.2. Thread Scheduling. Unlike other parallel programming languages such as TBB [7] and OpenMP [8], the OpenCL programming model is a single-instruction and multiple-thread (SIMT) model just like CUDA [9]. An OpenCL kernel describes the behavior of a single thread, and the host application explicitly declares the number of threads to express the parallelism of the application. In OpenCL terminology, a single thread is called a `workitem` (a `thread` in CUDA). The OpenCL programmer can form a set of workitems as a `workgroup` (a `threadblock` in CUDA), where the programmer can synchronize among workitems by `barrier` and `mem_fence`. A single workgroup is composed of a multi-dimensional array of workitems. Figure 2 shows the OpenCL execution model and how an OpenCL kernel is mapped on the OpenCL compute device. In OpenCL, a kernel is allocated on a compute device, and a workgroup is executed on a compute unit. A single workitem is processed by a processing element (PE). For better performance, programmers can tune the number of workitems and change the workgroup size.

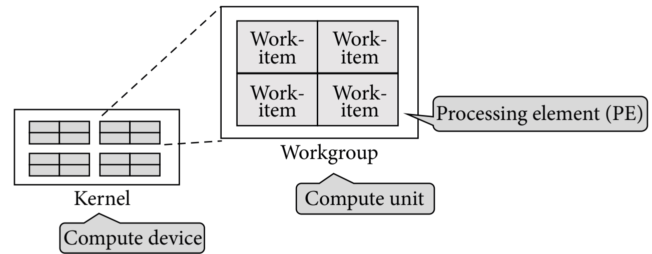


Figure 2: OpenCL execution model.

It is common for OpenCL applications to launch a massive number of threads for kernels expecting speedup by parallel execution. However, portability of OpenCL applications in terms of performance is not maintained on different architectures. In other words, an optimal decision of how to parallelize (partition) a kernel on GPUs does not usually guarantee good performance on CPUs. The partitioning decision of a kernel is done by changing *the number of workitems* and *workgroup size*.

2.2.1. Number of Workitems. First, the number of workitems and the amount of work done by a workitem affect performance differently on CPUs and GPUs. A massive number of short workitems hurts performance on CPUs but helps performance on GPUs. The performance difference comes from the different architectural characteristics between CPUs and GPUs. On GPUs, a single workitem is processed by a scalar processor (SP) or one single SIMD lane. As is widely known, GPUs are specialized for supporting a large number of concurrently running threads, and high thread-level parallelism (TLP) is critical to achieve high performance [10–13]. On the contrary, on CPUs, the TLP is limited by the number of cores, so using more threads to do the same amount of work does not help performance on CPUs but hurts it due to the overhead of emulating a large number of concurrently executing workitems on a small number of cores.

2.2.2. Number of Workitems and Instruction-Level Parallelism (ILP). The number of workitems affects the instruction-level parallelism (ILP) of the OpenCL kernel on CPUs. Increasing ILP in GPU applications has not been a popular performance optimization technique. The reasons are as follows. First, the hardware can explore much TLP so ILP will not affect the performance significantly. Second, the hardware does not explore too much ILP. The GPU processor is an in-order scheduler processor and does not also support branch prediction to increase ILP. However, on CPUs, the hardware has been designed to increase ILP with multiple features such as superscalar execution and branch predictors.

A modern superscalar processor executes more than one instruction concurrently by dispatching multiple independent instructions during a clock cycle to utilize the multiple functional units in CPUs. Superscalar CPUs use hardware that checks data dependencies between instructions at run-time and schedule instructions to run in parallel [14].

One of the performance problems of OpenCL applications on CPUs is that usually the kernel is written mostly to utilize the TLP, not for ILP. The OpenCL programming model is an SIMT model, and it is common for an OpenCL application to have a massive number of threads. Since independent instructions computing different elements are separated into different threads, most instructions in a single workitem in the kernel are usually dependent on previous instructions, so that typically most OpenCL kernels have `ILP one`; only one instruction can be dispatched to execute in a workitem. On the contrary, on conventional programming models such as OpenMP, independent instructions exist between different loop iterations. For better performance on CPUs, the

OpenCL kernel should be written to have more independent instructions.

2.2.3. Workgroup Size. The second important component is the workgroup size. Workgroup size determines the amount of work in a workgroup and the number of workgroups of a kernel. On GPUs, a workgroup or multiple groups are executed on a streaming multiprocessor (`SM`), which is equivalent to a physical core on the multicore CPU. Similarly, a workgroup is processed by a logical core of the CPU [15, 16]. (Even though it depends on the implementation, many implementations have this characteristic in common.). A workload size that is too small per workgroup makes the scheduling overhead more significant in total execution time on CPUs since the thread context switching overhead becomes bigger.

An OpenCL programmer can explicitly set workgroup size or let the OpenCL implementation decide. If `NULL` value is passed for workgroup size when the host application calls `clEnqueueNDRangeKernel`, the OpenCL implementation automatically partitions global workitems into the appropriate number of workgroups.

2.2.4. Proposed Solutions and Limitations. Many proposals to reduce the scheduling overhead by serialization have been presented [15–17]. Scheduling overhead is not a fundamental problem with the OpenCL programming model. Better OpenCL implementation can have less overhead than other suboptimal implementations. Serialization is a technique that serializes multiple workitems into a single workitem. For example, SnuCL [15] overcomes the overhead of a large number of workitems by serializing them to have fewer workitems in the run-time. However, even with serialization, multiple OpenCL implementations for CPUs still have high scheduling overhead due to the complexity of compiler analysis. Therefore, instead of using many workitems, as is usually the case for OpenCL applications on GPUs, we are better off assigning more work to each workitem with fewer workitems on CPUs. The results from our experiments agree with the above inferences.

2.3. Memory Allocation and Data Transfer. In general, a parallel programming model can have two types of address space options: unified memory space and disjoint memory space [18]. Conventional programming models for CPUs provide the unified memory address space both for the sequential code and for parallel code. The benefit of unified memory space is easy programming, with no explicit data transfer for kernel execution.

On the contrary, even though it is hard for programmers to program, OpenCL provides disjoint memory space to programmers. This is because most heterogeneous computing platforms have disjoint memory systems due to the different memory requirements of different architectures. OpenCL assumes for its target a system where communication between the host and compute devices are performed explicitly by a system network, such as PCI-Express. But, the assumption of discrete memory systems is not true when we use CPUs as compute devices for kernel execution. The host and compute devices share the same memory system resources such as last-level cache, on-chip interconnection, memory controllers, and DRAMs.

The drawback of disjoint memory address space is that it requires the programmer to explicitly manage data transfer between the host and compute devices for kernel execution. In common OpenCL applications, the data should be transferred back and forth in order to be processed by the host or the compute device [3], which becomes unnecessary when we use only the host for computation. To minimize the data transfer overhead on a specific architecture, OpenCL programmers usually have to rewrite the host code [3]. Often, they need to change the memory allocation flags or use different data transfer APIs for performance. For example, the programmer should allocate memory objects on host memory or device memory depending on target platform. These rewriting efforts have been a burden for programmers and have even been a waste of time due to the lack of architectural or run-time knowledge of a specific system in most cases.

2.3.1. Memory Allocation Flags. One of rewriting efforts is changing the memory allocation flag. OpenCL provides the programmer multiple options for memory object allocation flags when the programmer calls `clCreateBuffer` that could affect the performance of data transfer and kernel execution. The memory allocation flag is used to specify how the object is accessed by a kernel and where it is allocated.

Access Type. First, programmers can specify if the memory object is a read-only memory object (`CL_MEM_READ_ONLY`) or write-only one (`CL_MEM_WRITE_ONLY`) when referenced inside a kernel. The programmer can set memory objects used as input to the kernel as read-only and memory objects used as output from the kernel as write-only. If the programmer does not specify access type, the default option is to create a memory object that can be read and written by the kernel (`CL_MEM_READ_WRITE`). `CL_MEM_READ_WRITE` can also be explicitly specified by programmers.

Where to Allocate. The other option that programmers can specify is where to allocate a memory object. When the programmer does not specify allocated location, the memory object is allocated on the device memory in the OpenCL compute device. OpenCL also supports the pinned memory. When the host code creates memory objects using the `CL_MEM_ALLOC_HOST_PTR` flag, the memory object is allocated on the host-accessible memory that resides on the host. Different from allocating the memory object in the device memory, there is no need to transfer the result of kernel execution back to the host memory when the result is required by the host.

2.3.2. Different Data Transfer APIs. OpenCL also provides different APIs for data transfer between the host and compute devices. The host can enqueue commands to read data from an OpenCL memory object that is created by `clCreateBuffer` call to the memory object that is mostly created by `malloc` call in the

host memory (by `clEnqueueReadBuffer` API). The host can also enqueue commands to write data to the OpenCL memory object from the memory object in the host memory (by `clEnqueueWriteBuffer` API). The programmer can also map an OpenCL memory object to have the host-accessible pointer of the mapped object (by `clEnqueueMapBuffer` API).

2.4. Vectorization and Thread Affinity

2.4.1. Vectorization. Utilizing SIMD units has been one of the key performance optimization techniques for CPUs [19]. Since SIMD instructions can perform computation on more than one data item at the same time, SIMD utilization could make the application more efficient. Many vendors have released various SIMD instruction extensions on their instruction set architectures, such as MMX [20].

Various methods have been proposed to utilize the SIMD instruction: using optimized function libraries such as Intel IPP [21] and Intel MKL [22], using C++ vector classes with Intel ICC [23], or using DSL compilers such as the Intel SPMD Program Compiler [24]. Programmers can also program in assembly or use intrinsic functions. However, all of these methods assume rewriting the code. Due to this limitation, and to help programmers easily write applications utilizing SIMD instruction, autovectorization has been implemented in many modern compilers [19, 23].

It is quite natural for programmers to expect that a programming model difference has no effect on compiler autovectorization on the same architecture. For example, if a kernel is written in both OpenCL and OpenMP and both implementations are written in a similar manner, programmers would expect that both codes are vectorized in a similar fashion, thereby giving similar performance numbers. Even though it depends on the implementation, this is not usually true. Unfortunately, today's compilers are very fragile about vectorizable patterns, which depend on the programming model. Applications should satisfy certain conditions in order to fully take advantage of compiler autovectorization [19]. Our evaluation in Section 3.5.1 shows an example of this fragility and verifies the possible effect of programming models on vectorization.

2.4.2. Thread Affinity. Where to place threads can affect the performance on modern multicore CPUs. Threads can be placed on each core in different ways, which can create a performance difference. The performance impact of the placement would increase with more processors on the system.

The performance difference can occur for multiple reasons. For example, because of the different latency on the interconnection network, threads that are far away will take longer to communicate with each other, whereas threads close to the adjacent core can communicate more quickly. Also, an application that requires data sharing among adjacent threads can benefit if we assign these adjacent threads to nearby cores. Proper placement can also eliminate the communication overhead by utilizing shared cache. For the performance reason, most conventional parallel programming models support affinity, such as `CPU_AFFINITY` in OpenMP [8].

Unfortunately, thread affinity is not supported in OpenCL. An OpenCL workitem is a logical thread, which is not tightly coupled with a physical thread even though most parallel programming languages provide this feature. The reason for the lack of this functionality is that the OpenCL design philosophy emphasizes portability over efficiency.

We present the lack of affinity support as one of the performance limitations of OpenCL on CPUs compared to other programming languages for CPUs. We would like to present a potential solution to enhance OpenCL performance on CPUs. We found the benefit of better utilizing cache on OpenCL applications by thread affinity. An example is presented in Section 3.5.2.

Table 1: Experimental environment.

CPUs	Intel Xeon E5645
# Cores	4
Vector width	SSE 4.2, 4 single precision FP
Caches	L1D/L2/L3: 64 KB/256 KB/12 MB
FP peak performance	230.4 GFlops
Core frequency	2.40 GHz
DRAM	4 GB
GPUs	NVidia GeForce GTX 580
# SMs	16
Caches	L1/Global L2: 16 KB/768 KB
FP peak performance	1.56 TFlops
Shader Clock frequency	1544 MHz
O/S	Ubuntu 12.04.1 LTS
Platform	Intel OpenCL Platform 1.5 for CPU NVidia OpenCL Platform 4.2 for GPU
Compiler	Intel C/C++ compiler 12.1.3

3. Evaluation

Given the preceding background on the anticipated effects of architectural aspects to understand the OpenCL performance on CPUs, the goal of our study is to quantitatively explore these effects.

3.1. Methodology. The experimental environment for our evaluation is described in Table 1. Our evaluation was performed on a heterogeneous computing platform consisting of a multicore CPU and a GPU; the OpenCL kernel was executed either on the Intel OpenCL platform [4] or the NVidia OpenCL platform [5]. We implemented an execution framework so that we can vary and control many aspects on the applications without code changes. The execution framework is built as an OpenCL delegator library that invokes OpenCL libraries from vendors: the one from Intel for kernel execution on CPUs and the other from NVidia for kernel execution on GPUs.

Table 2: List of NVidia OpenCL benchmarks for API overhead evaluation.

Benchmark
oclBandwidthTest, oclBlackScholes, oclConvolutionSeparable, oclCopyComputeOverlap,
oclDCT8x8, oclDXTCompression, oclDeviceQuery, oclDotProduct, oclHiddenMarkovModel,
oclHistogram, oclMatrixMul, oclMersenneTwister, oclMultiThreads, oclQuasirandomGenerator,
oclRadixSort, oclReduction, oclSimpleMultiGPU, oclSortingNetworks, oclTranspose,
oclTridiagonal, oclVectorAdd

We use different applications for each evaluation. To verify the API overhead, We use NVIDIA OpenCL Benchmarks [5]. For other aspects, including scheduling overhead, memory allocation, and data transfer, we first use simple applications for evaluation. We also vary the data size of each application. The applications are ported to the execution framework we implemented. After evaluation with simple applications, we also use the Parboil benchmarks [25, 26]. Tables 2, 3, and 4 describe evaluated applications and their default parameters.

We use the wall-clock execution time. To measure stable execution time without fluctuation, we iterate the kernel execution until the total execution time of an application reaches a long enough running time, 90 seconds in our evaluation. This is sufficiently long to have a multiple number of kernel executions for all applications in our evaluation. Using the average kernel execution time per kernel invocation calculated, we use normalized throughput to clearly present the performance difference on multiple sections.

3.2. API Overhead. As we discussed in Section 2.1, the OpenCL application has API overhead. To verify the API overhead, we measured the time cost of each API function in executing the OpenCL application in NVIDIA OpenCL Benchmarks [5]. The workload size for each benchmark is the size the application provides as a default. Figure 3 shows the ratio of the execution time of kernel execution and auxiliary API functions to the total execution time of each OpenCL benchmark. (Auxiliary API functions are OpenCL API functions called in the host code to coordinate kernel execution.) The last column `total` means the arithmetic mean of the data from each benchmark. From the figure, we can see that a large portion of execution time is spent on auxiliary API functions instead of kernel execution.

For detailed analysis, we categorized OpenCL APIs into 16 categories. We group multiple categories for visibility in the following. Figure 4 provides a detailed example of API overheads by showing the execution time distribution of each API function category for `oclReduction`. `Enqueued Commands` category includes kernel execution time and data transfer time between the host and compute device and accounts for 12.1% of execution time. We find that the API overhead is larger than the actual computation.

3.2.1. Overhead due to Various Platform Supports. Figure 5 shows the ratio of the execution time of each category to the total execution time of each OpenCL benchmark. The figure shows the performance degradation due to the flexibility of various platforms. We see that the API functions in `Platform`, `Device`, and `Context` categories consume over 80 percent of the total execution time of each OpenCL benchmark on average. The need to call API functions in these categories comes from the fact that each OpenCL application needs to set up an execution environment for which the detailed mechanism would change, depending on the platform. From our evaluation, we also see that each call to the API functions in these categories requires a long execution time. In particular, context management APIs incur a large execution time overhead. Figure 6 shows the execution time distribution of `clCreateContext` and `clReleaseContext` to total execution time in each benchmark. These functions are called at most once on each OpenCL benchmark. But in conventional parallel programming models, context and device are implicit, so there is no need to call such management functions.

3.2.2. Overhead due to JIT Compilation. The list of OpenCL kernels in the application is represented by the `cl_program` object. `cl_program` object is created using either `clCreateProgramWithSource` or `clCreateProgramWithBinary`. JIT compilation is performed by either calling the `clBuildProgram` function or a sequence of `clCompileProgram` and `clLinkProgram` functions for the `cl_program` object to build the program executable for one or more devices associated with the program [3].

JIT compilation overhead is another source of the API overhead. Figure 7 shows the execution time distribution of `Program` category to the sum of execution time of all categories except `Platform`, `Device`, and `Context` categories. We exclude these 3 categories that we have evaluated in previous section. The figure clearly shows the performance degradation due to the JIT compilation. We see that the API functions in `Program` category consume around 33% of the total execution time for 13 categories of the API functions including kernel execution. Execution time overhead of `clBuildProgram` is not negligible in most benchmarks.

Caching. Caching JIT compiled code can help reduce the overhead. Some of the caching ideas are available in OpenCL. Programmers can extract compiled binary by using the `clGetProgramInfo` API function and store it using FILE I/O functions. When the kernel code is not modified since caching, programmers can load the cached binary on disk and use the binary instead of performing JIT compilation on every execution of the application.

3.2.3. Summary. In this section, we can see the high overhead of explicit context management (Section 3.2.1) and JIT compilation (Section 3.2.2) in OpenCL applications. These are unique characteristics of OpenCL compared to conventional programming models for portable execution over multiple architectures.

Table 3: Configurations of simple applications.

Benchmark	Kernel	Global work size	Local work size
Square	Square	10000, 100000, 1000000, 10000000	NULL
Vectoraddition	vectoadd	110000, 1100000, 5500000, 11445000	NULL
Matrixmul	matrixMul	800 × 1600, 1600 × 3200, 4000 × 8000	16 × 16
Reduction	reduce	640000, 2560000, 10240000	256
Histogram	histogram256	409600	128
Prefixsum	prefixSum	1024	1024
Blackscholes	blackScholes	1280 × 1280, 2560 × 2560	16 × 16
Binomialoption	binomialoption	255000, 2550000	255
Matrixmul(naive)	matrixMul	800 × 1600, 1600 × 3200, 4000 × 8000	16 × 16

Table 4: Configurations of the Parboil benchmarks.

Benchmark	Kernel	Global work size	Local work size
CP	Cenergy	64 × 512	16 × 8
MRI-Q	computePhiMag	3072	512
	computeQ	32768	256
MRI-FHD	RhoPhi	3072	512
	computeFHD	32768	256

Figure 3: Execution time distribution of kernel execution and auxiliary API functions.

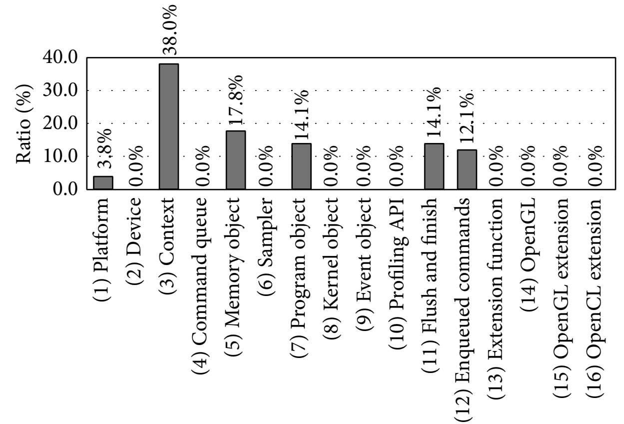


Figure 4: Execution time distribution of each category of API function for `oclReduction`.

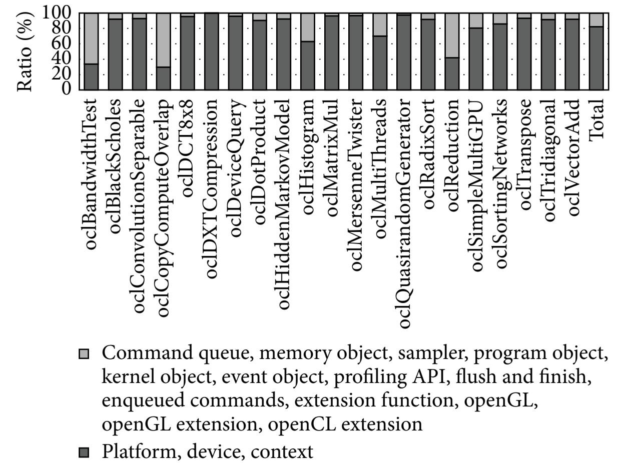


Figure 5: Execution time distribution of each category of API functions.

It should be noted that the workload size for the evaluation in Section 3.2 is the size that the application provides as the default workload size, which is relatively small. Therefore, these overheads can be reduced with a large workload size and thus a long kernel execution time. But it is also true that these overheads are not negligible with small workload size, so the programmer should consider the workload size when they decide whether to use OpenCL or not.

3.3. Thread Scheduling

3.3.1. Number of Workitems. Associated with the discussion in Section 2.2.1, to evaluate the effect of the number of workitems and the workload size per workitem, we perform an experiment on OpenCL applications by allocating more computation per workitem. We coalesce multiple workitems into a single workitem by forming a loop inside the kernel. To keep the total amount of computation the same, we reduce the number of workitems to execute the kernel. The number of workitems coalesced increases from `1` to `1000` workitems by multiplying by `10` for each step. Figure 8 shows the performance of `Square` and `Vectoraddition` applications with a different amount of computation per

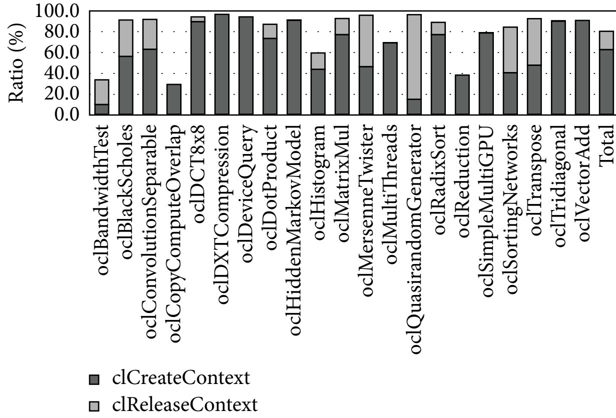


Figure 6: Execution time distribution of `clCreateContext` and `clReleaseContext`.

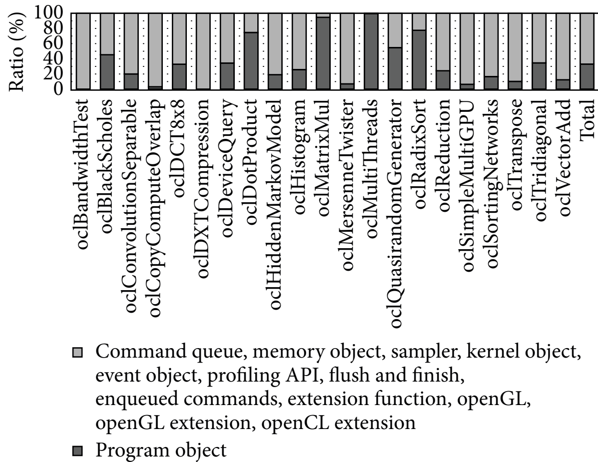


Figure 7: Execution time distribution of `Program` category API functions except `Platform`, `Device`, and `Context` categories.

workitem. Table 5 shows the number of workitems used in this evaluation.

From Figure 8, we find a performance gain for allocating more work per workitem on CPUs. A noticeable example is a case of `Vectoraddition`, where we add an array of numbers. If we create as many workitems as the size of arrays, we end up creating significant overhead on CPUs. When we reduce the number of workitems, we see a major performance improvement for CPUs. We could also find that the performance is saturated sometimes when the workload assigned per each workitem goes over a certain threshold. This shows that when each workitem has a sufficient workload, scheduling overhead is reduced.

Compared to CPUs with high overhead of handling many workitems, GPUs have low overhead for maintaining a large number of workitems, as our evaluation shows. Furthermore, reducing the number of workitems degraded performance on GPUs significantly. The large performance degradation on GPUs is because we could no longer take advantage of many processing units on GPUs.

One of the reasons for performance improvement by allocating more workload per workitem is the reduced number of instructions. Figure 9 shows the number of dynamic instructions of `Square` and `Vectoraddition` applications with a different amount of computation per workitem. The left figure of Figure 9 shows the dynamic instruction count including instructions from OpenCL APIs on top of instructions from the OpenCL kernel. And the right figure of Figure 9 represents the instructions only from the kernel.

For this evaluation, we implement a tool based on Pin [27] that counts the number of instructions. The tool also identifies the function to which the instruction belongs. From Figure 9, we can see that the number of instructions is reduced with more workload per workitem even though the amount of computation is the same regardless of the number of workitems. The number of instructions from OpenCL APIs as well as that from the kernels increases, so that the scheduling overhead exists on both OpenCL APIs and the JIT compiled OpenCL kernel binary. Figure 10 shows reduced overhead on OpenCL APIs with increased workload per workitem. The instructions from OpenCL APIs are for scheduling, not for computation intended by programmers represented as an OpenCL kernel. So a reduced number of instructions from OpenCL APIs means reduced overhead.

Figure 11 shows the performance of Parboil benchmarks with a similar experiment [25, 26]. The number of workitems coalesced is different depending on the benchmark since we could not increase the workload per workitem in the same manner for all kernels. We find a similar performance gain of allocating more work per workitem. Figure 12 represents the reduced number of dynamic instructions with increased workload per workitem.

3.3.2. Number of Workitems and Instruction-Level Parallelism (ILP). As we discussed in Section 2.2.2, the number of workitems, and therefore how to parallelize the computation, also affects the instruction-level parallelism (ILP) of the OpenCL kernel on CPUs. Coalescing multiple workitems can not only reduce the scheduling overhead but also improve the performance by utilizing ILP.

To evaluate the ILP effect on both the CPU and the GPU, we implemented a set of compute-intensive microbenchmarks that share common characteristics. Every benchmark has an identical number of dynamic instructions and memory accesses. Each benchmark also has the same instruction mixture, such as a ratio of the number of branch instructions over the total number of instructions. The only difference between each benchmark is ILP by varying the number of independent instructions. From the baseline implementation, we increase the number of operand variables, so that the number of independent instructions can increase. For example, in the case of `ILP 1`, the next instruction depends on the output of the previous instruction so that the number of independent instructions is one; but in the case of `ILP 2`, an independent instruction exists between two dependent instructions.

Figure 13 shows the performance with increasing ILP. We provide enough workitems to fully utilize TLP. The number of workitems remains the same for all microbenchmarks. The

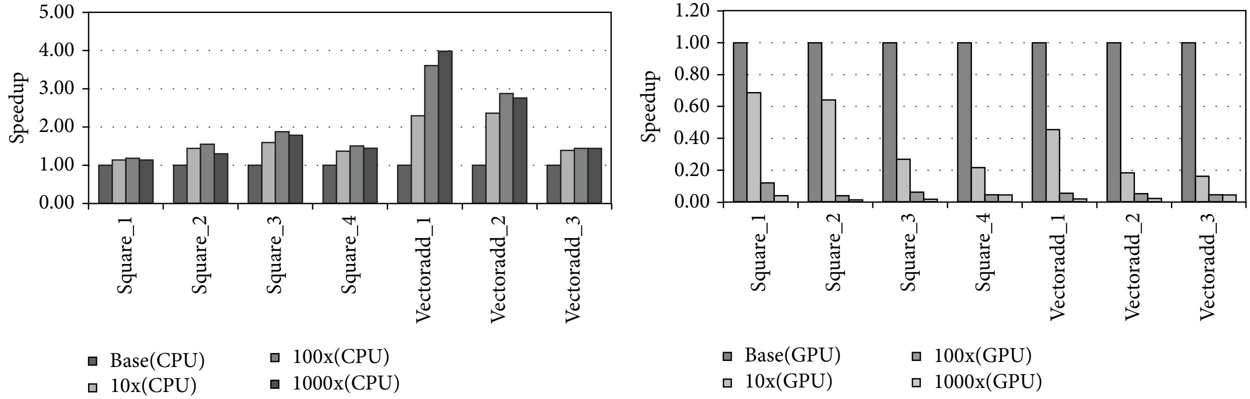


Figure 8: Performance of Square and Vectoraddition applications with different workload per workitem.

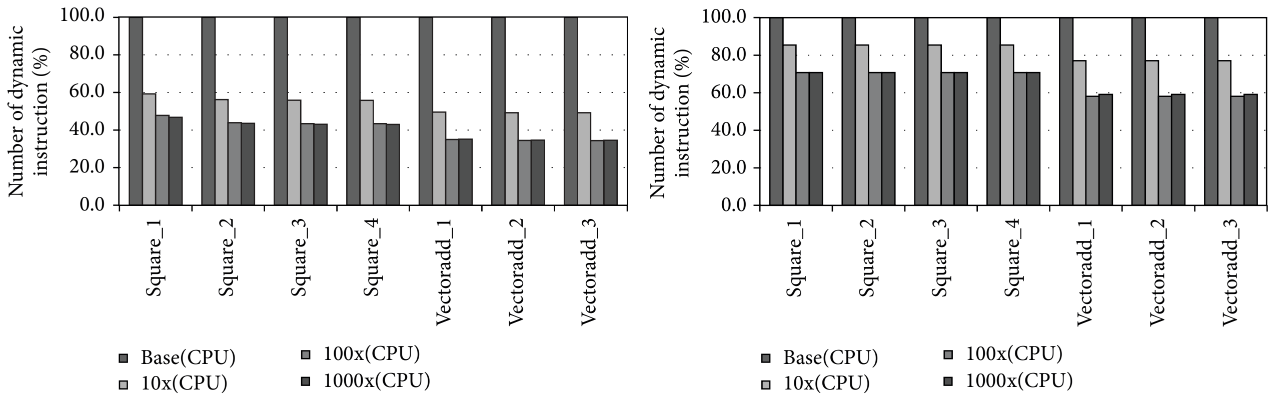


Figure 9: The number of dynamic instructions of Square and Vectoraddition applications with different workload per workitem including (L) instructions from OpenCL APIs and (R) kernel only.

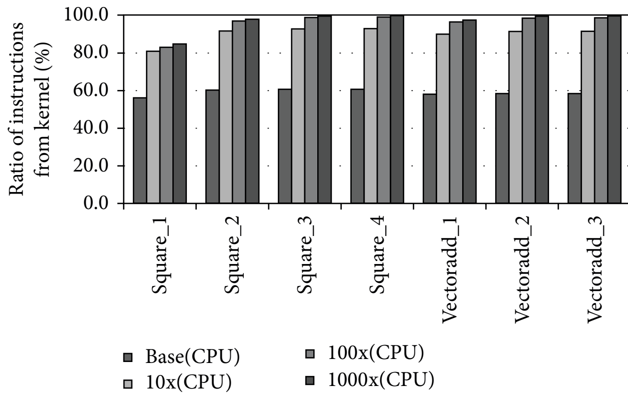


Figure 10: The ratio of instructions from kernel over the instructions around clEnqueueNDRangeKernel for Square and Vectoraddition applications with different workload per workitem.

left y-axis represents the throughput of the CPUs, and the right one represents the throughput of the GPUs. From the figure, we find that performance improves depending on the ILP value of the OpenCL kernel on CPUs. On the contrary, there is no performance variation on GPUs with different degrees of instruction-level parallelism.

3.3.3. Workgroup Size. Associated with the discussion in Section 2.2.3, the number of workitems in workgroups can affect the performance of the OpenCL application. We evaluate the effect of workgroup size, both on CPUs and GPUs. We vary the number of workitems in a workgroup by passing a different argument for workgroup size (`local_work_size`) on kernel invocation. We maintain the total number of workitems of the kernel as the same. Table 6 shows the different workgroup size for each benchmark, and Figures 14, 16, and 18 show the performance of applications with different workgroup sizes. When the `NULL` argument is passed on kernel invocation, the workgroup size is implicitly defined by the OpenCL implementation.

The benchmarks can be categorized into three categories, depending on the behavior. The first group consists of `Square`, `Vectoraddition`, and naive implementation of `Matrixmul`; `Matrixmul` belongs to the second group; and `Blackscholes` belongs to the last.

`Square`, `Vectoraddition`, and naive implementation of `Matrixmul` show a performance increase with increased workgroup sizes on the CPU, as can be seen in Figure 14. On the `Square` and `Vectoraddition` applications, performance achieved with the `NULL` workgroup size is less than the peak performance we achieve. This implies that

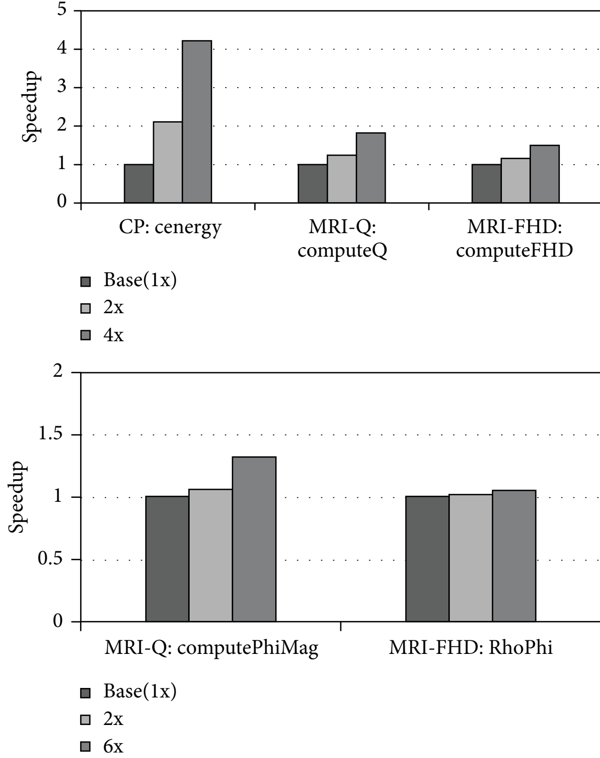


Figure 11: Performance of Parboil benchmarks with different workloads per workitem.

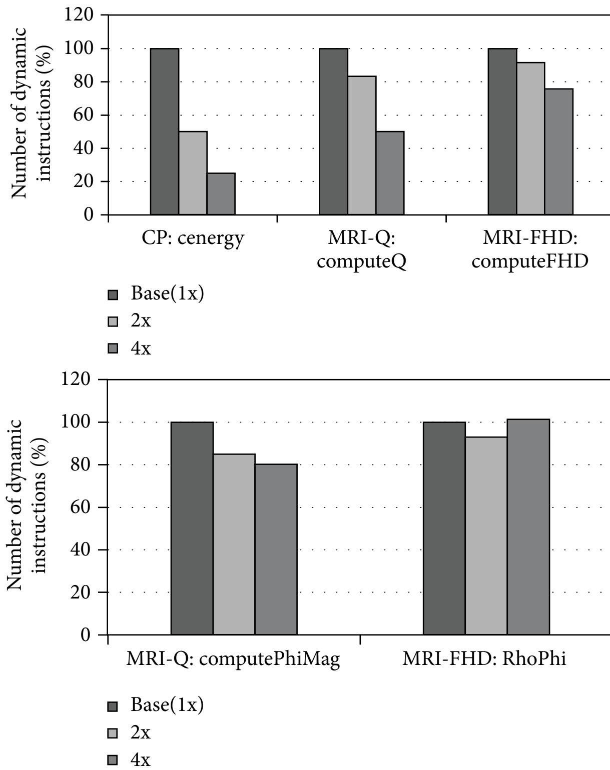


Figure 12: The number of dynamic instructions of Parboil benchmarks with different workload per workitem.

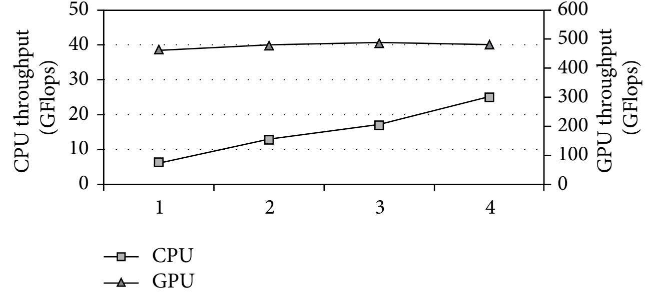


Figure 13: Performance of ILP microbenchmark on the CPU and the GPU.

Table 5: Number of workitems for each application.

Benchmark	Base	10x	100x	1000x
Square_1	10000	1000	100	10
Square_2	100000	10000	1000	100
Square_3	1000000	100000	10000	1000
Square_4	10000000	1000000	100000	10000
Vectoradd_1	110000	11000	1100	110
Vectoradd_2	1100000	110000	11000	1100
Vectoradd_3	5500000	550000	55000	5500

Table 6: Workgroup size for each application.

Benchmark	Base	Case_1	Case_2	Case_3	Case_4
Square	NULL	1	10	100	1000
Vectoraddition	NULL	1	10	100	1000
Matrixmul	16 × 16	1 × 1	2 × 2	4 × 4	8 × 8
Blackscholes	16 × 16	1 × 1	1 × 2	2 × 2	2 × 4
Matrixmul(naive)	16 × 16	1 × 1	2 × 2	4 × 4	8 × 8

programmers should explicitly set the workgroup size for the maximum performance. The performance with a small workgroup size is also bad on GPUs since the workgroup is allocated per SM, so that the small workgroup size makes GPUs unable to utilize many warps in an SM. Even though no hardware TLP is available inside a logical core on CPUs (the evaluated CPU is an SMT processor, so multiple logical cores share one physical core), performance increases with a large workgroup size. This is because the overhead of managing a large number of workgroups, many threads in many implementations, is reduced. We also find that performance is saturated at a certain workgroup size.

The left figure of Figure 15 shows the number of dynamic instructions of `Square`, `Vectoraddition`, and naive implementation of `Matrixmul` with different workgroup size on CPUs. The right figure of Figure 15 shows the ratio of instructions from kernel over the instructions around `clEnqueueNDRangeKernel` for those applications with a different workgroup size. From the left figure of Figure 15, we can see that the number of instructions is reduced with larger workgroup size. This is because the number of instructions from OpenCL APIs is reduced, as can be seen from the right figure of Figure 15. The number

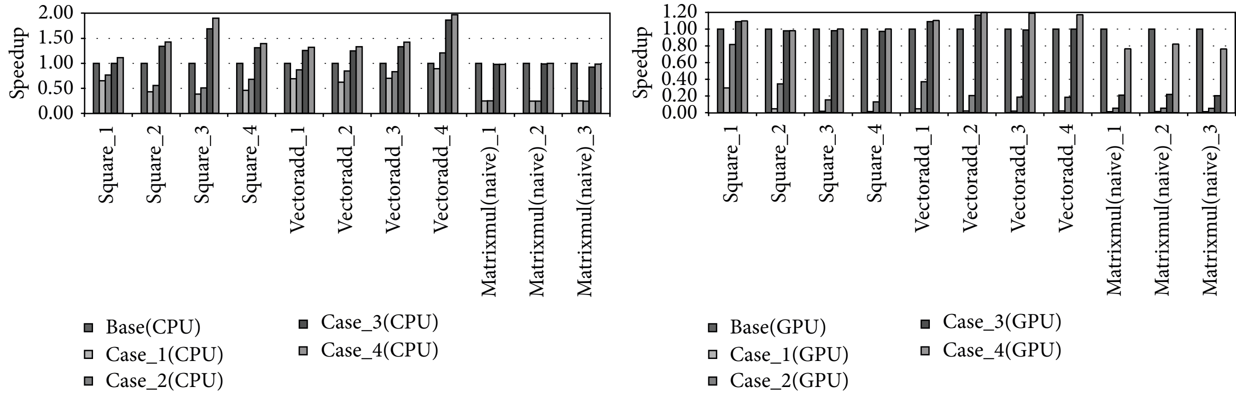


Figure 14: Performance of applications with different workgroup size on CPUs and GPUs.

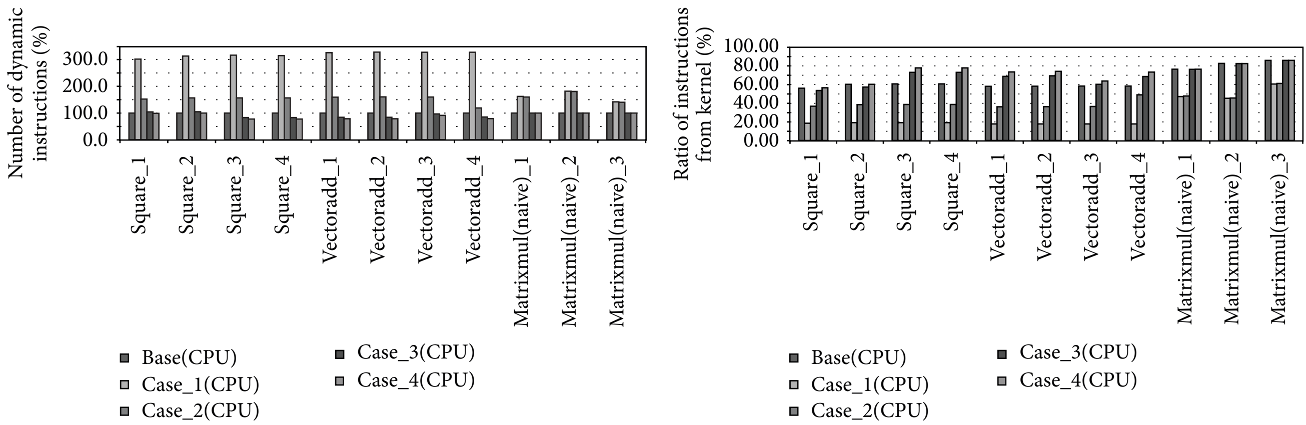


Figure 15: (U) The number of dynamic instructions of Square, Vectoraddition, and naive implementation of Matrixmul with different workgroup size on CPUs. (L) The ratio of instructions from kernel over the instructions around clEnqueuNDRangeKernel for Square, Vectoraddition, and naive implementation of Matrixmul with different workgroup size.

of instructions from the OpenCL kernel remains the same regardless of workgroup size.

As we can see from Figure 16, we also see a significant performance increase on the `Matrixmul` application with an increased workgroup size. The optimal workgroup size of this application is different, depending on platforms. For inputs 1 and 2, the optimal workgroup size on CPUs is 8×8, but the optimal size on GPUs is 16×16. Here, the performance depends not only on the scheduling overhead, but also on the cache usage. `Matrixmul` utilizes the local memory in OpenCL by blocking. Workgroup size can change the local memory usage of the kernel. Since the size of the cache in CPUs and the scratchpad memory in GPUs are different, the optimal workgroup size can be different. Figure 17 shows the reduced number of dynamic instructions of Matrixmul with increasing workgroup size.

Unlike other applications, `Blackscholes` shows different performance behavior on CPUs and on GPUs. As we can see in Figure 18, the workgroup size does not change the performance on CPUs, but it affects the performance significantly on GPUs. Since the workload allocated on a single workitem is relatively long compared to other applications, the overhead of managing a large number of workgroups becomes negligible. On the contrary, the number of warps in the SM is defined by the workgroup size on GPUs, which makes the performance on GPUs low on small workgroup sizes. Figure 19 shows that the number of instructions does not change much for Blackscholes, regardless of workgroup size.

Figure 20 shows the performance of Parboil benchmarks with different workgroup sizes. We increase the workgroup size from one to 16 times by multiplying by 2 for each step. Since the workgroup size for `CP:cenergy` kernel is two-dimensional, we increase the workgroup size of the kernel in two directions. `CP:cenergy(x)` represents the performance with workgroup sizes 1×8, 2×8, 4×8, 8×8, and 16×8. `CP:cenergy(y)` represents the performance with workgroup sizes 16×1, 16×2, 16×4, 16×8, and 16×16. In general, we find the performance gain with a large workgroup size. The performance saturates when there is enough computation inside the workgroup. Figure 21 shows that the performance gain is due to reduced scheduling overhead, which is represented by a reduced number of dynamic instructions.

3.3.4. Summary. Here, we summarize the findings on thread scheduling for OpenCL applications.

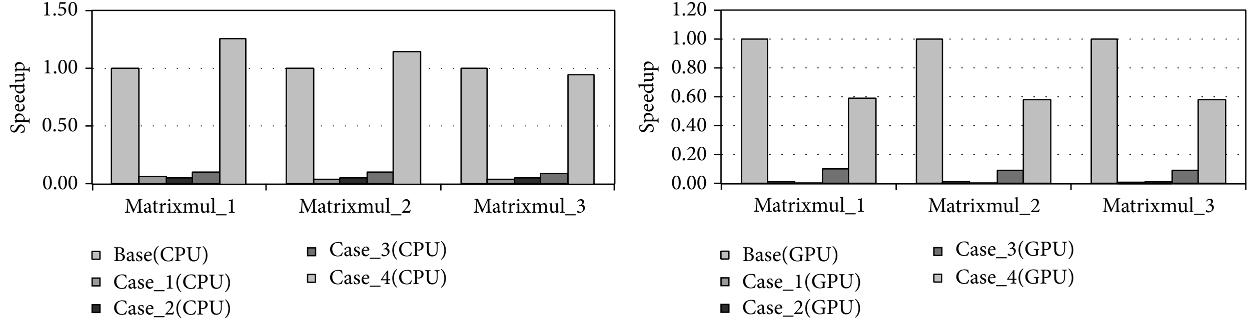


Figure 16: Performance of Matrixmul with different workgroup size on CPUs and GPUs.

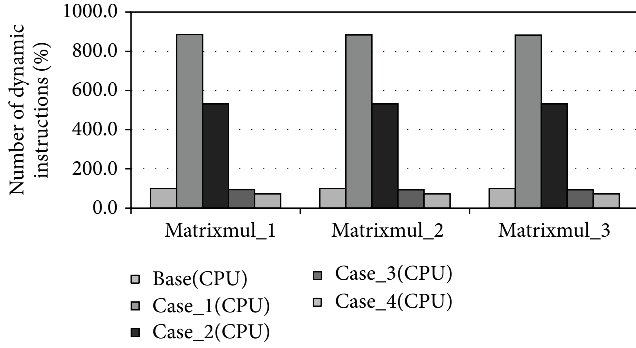


Figure 17: The number of dynamic instructions of Matrixmul with different workgroup size on CPUs.

(1) Allocating more work per workitem by manually coalescing multiple workitems reduces scheduling overhead on CPUs (Section 3.3.1).

(2) High ILP increase performance on CPUs but not on GPUs (Section 3.3.2).

(3) Workgroup size affect performance both on CPUs and GPUs. In general, large workgroup size increases performance by reducing scheduling overhead on CPUs and enables utilizing high TLP on GPUs. Workgroup size can also affect the cache usage (Section 3.3.3).

3.4. Memory Allocation and Data Transfer. Associated with the discussion in Section 2.3, to evaluate the performance effect of different memory object allocation flags and different APIs for data transfer, we perform an experiment on OpenCL applications with different combinations of the following options. To measure exact execution performance, we use a blocking call for all kernel execution commands and memory object commands so that no command overlaps with other commands. The combination we use is three-dimensional as follows.

3.4.1. Evaluated Options for Memory Allocation and Data Transfer

(1) Different APIs for data transfer:

- (i) explicit transfer: `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` for explicit read and write;
- (ii) mapping: `clEnqueueMapBuffer` with `CL_MAP_READ`, `CL_MAP_WRITE` for implicit read and write.

(2) Kernel access type when referenced inside a kernel:

- (i) the kernel accesses the memory object as read-only/write-only:
  - (a) `CL_MEM_READ_ONLY` for the input to the kernel;
  - (b) `CL_MEM_WRITE_ONLY` for computation results from the kernel;
- (ii) the kernel accesses the memory object as read/write: `CL_MEM_READ_WRITE` for all memory objects.

(3) Where to allocate a memory object:

- (i) allocation on the device memory;
- (ii) allocation on the host-accessible memory on the host (pinned memory).

3.4.2. Metric: Application Throughput. The throughput we present here is the performance, including data transfer time, between the host and compute devices, not just the kernel execution throughput on the compute device. For example, the throughput of an application becomes half of the throughput when we consider only the kernel execution time if the data transfer time between the host and the compute device equals the kernel execution time. The way we calculate the throughput of an application is illustrated in

$$\text{Throughput_app} = \frac{\text{Throughput_kernel}}{\text{kernel_time} + \text{transfer_time}}. \quad (1)$$

3.4.3. Different Data Transfer APIs. We compare the performance of different data-transfer APIs on all possible allocation flags. (The combinations are as follows: (1) read-only/write-only memory object + allocation on the device; (2) read-only/write-only memory object + allocation on the

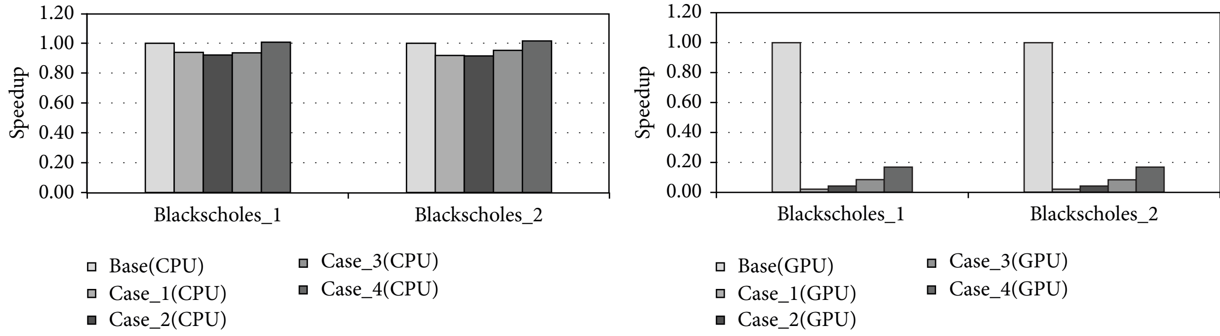


Figure 18: Performance of Blackscholes with different workgroup size on CPUs and GPUs.

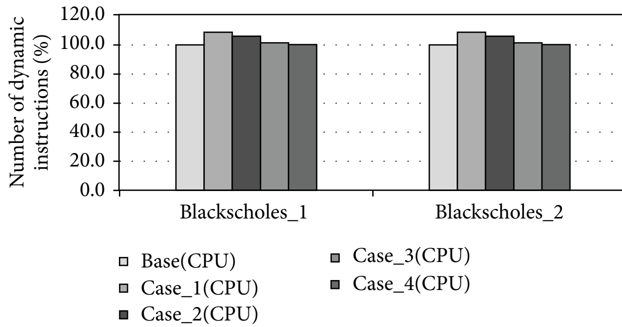


Figure 19: The number of dynamic instructions of Blackscholes with different workgroup size on CPUs.

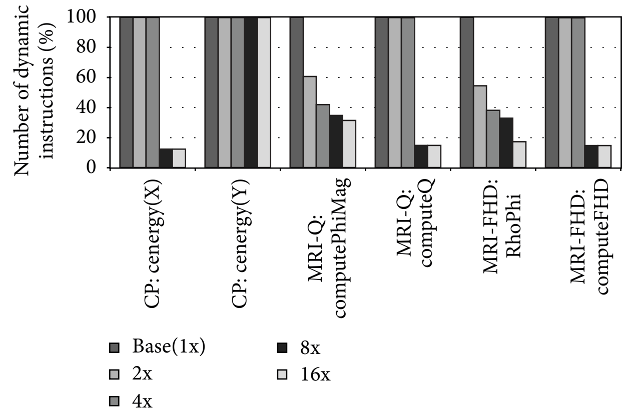


Figure 21: The number of dynamic instructions of Parboil benchmarks with different workgroup size on CPUs.

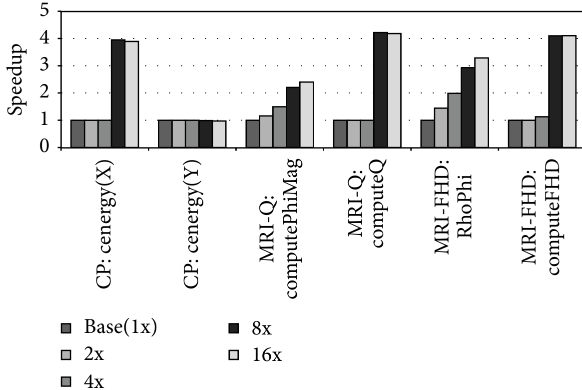


Figure 20: Performance of Parboil benchmarks with different workgroup size on CPUs.

host; (3) read-write memory object + allocation on the device; (4) read-write memory object + allocation on the host.) Figure 22 shows the performance of the benchmarks with different APIs for data transfer. The y-axis represents the normalized application throughput (`Throughput_app`) when we use mapping for data transfer over the baseline when we use explicit data transfer APIs. We find that mapping APIs have superior performance compared to explicit data transfer APIs, regardless of the decision on other dimensions. First, the performance of mapping APIs is superior wherever the memory object is allocated: on device memory or on pinned memory on host. Second, mapping APIs also perform better regardless of the decision for allocating the memory object as read-only/write-only or as read/write object.

Different APIs change data transfer time. Figure 23 shows the normalized data transfer throughput from the host to a compute device between different data transfer APIs. Figure 24 shows the one from compute device to host. The data transfer time is shorter with mapping APIs. The difference of data transfer throughput increases with increases in workload sizes and therefore increases in data transfer sizes.

We also report the performance of Parboil benchmarks with different APIs for data transfer [25, 26]. Since the data transfer time is much shorter than the kernel execution time on Parboil benchmarks, instead of using application throughput as shown in (1), we report the data transfer time from the host to device, and data transfer time from the device to host with different APIs. Figure 25 shows the different data transfer time of the Parboil benchmarks with different APIs for data transfer. The y-axis represents the data transfer time in milliseconds. The left figure in Figure 25 shows the data transfer time from the host to the compute device with different data transfer APIs. The right figure shows the one from the compute device to the host. As with simple applications, we find that the data transfer time is shorter with mapping APIs on these benchmarks.

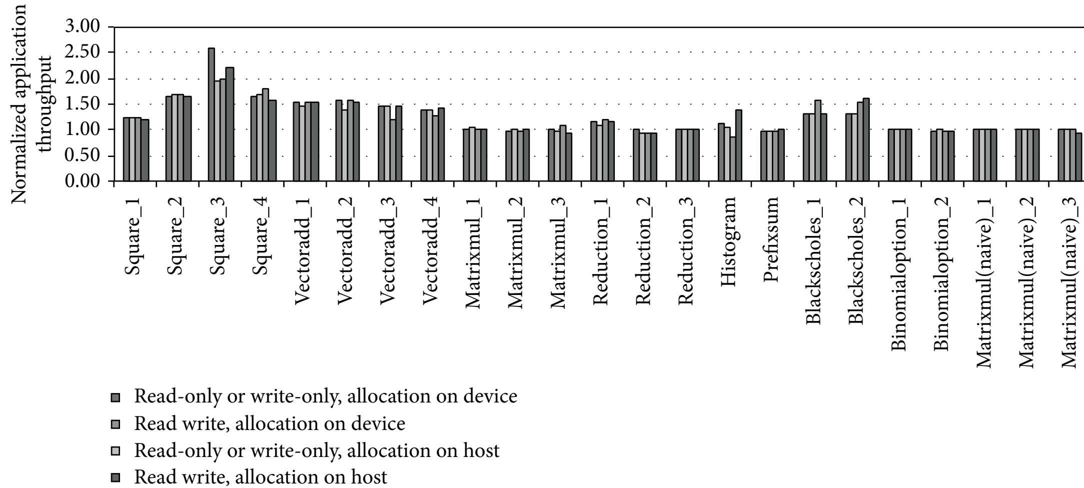


Figure 22: Normalized application throughput of mapping over explicit data transfer for all combinations on other dimensions. The performance of mapping APIs is superior to explicit data transfer on all possible combinations.

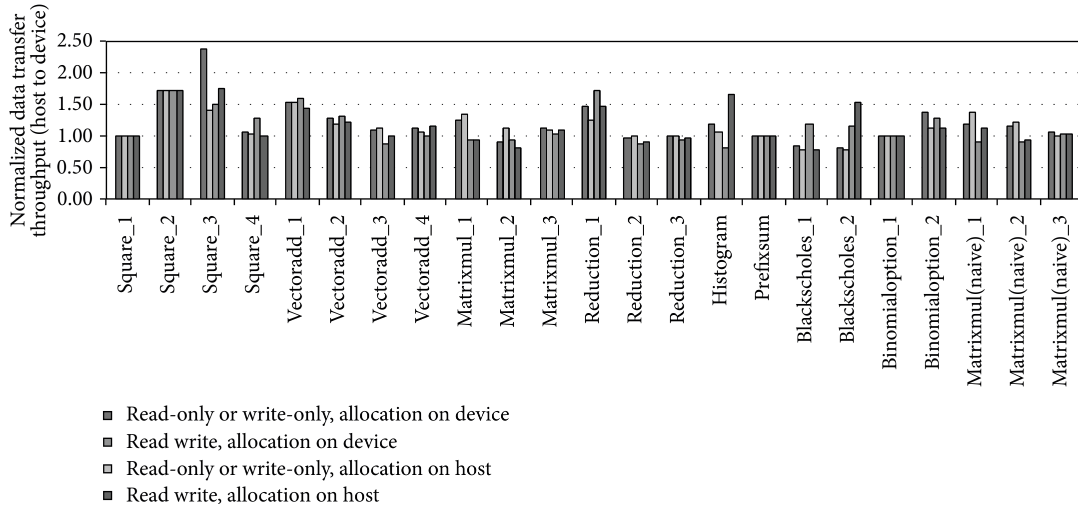


Figure 23: Normalized data transfer (host to device) throughput of mapping over explicit data transfer for all combinations on other dimensions.

The difference of data transfer time is due to the different behaviors of different APIs. When the host code explicitly transfers data between the host and the compute device, the OpenCL run-time library should allocate a separate memory object for the device and copy the data between the memory object allocated by the `malloc` call and the memory object allocated for the device that is allocated by the `clEnqueueReadBuffer` call. However, a separate memory object is not needed when the host code uses mapping; only returning a pointer of the memory object is needed. So, copying between memory objects becomes unnecessary.

3.4.4. Kernel Access Type When Referenced inside a Kernel. We also verify the performance effect of specifying a memory object as read-only/write-only or as read/write. Figure 26 shows the performance implication of this flag. The y-axis represents the normalized throughput when we allocate the memory object as read-only/write-only from the baseline when we allocate the object as read/write. OpenCL implementations can utilize the detailed information of how the memory object is accessed in the OpenCL kernel for optimization instead of naively assuming all objects are read and modified in the OpenCL kernel. However, we do not see a noticeable performance difference with our evaluated workloads. Kernel execution time and data transfer time between the host and compute device do not differ regardless of this memory allocation flag.

3.4.5. Where to Allocate a Memory Object. Finally, we also verify the performance effect of the allocation location of memory objects. Programmers can allocate the memory object on the host memory or the device memory. Figure 27 shows the performance of benchmarks with

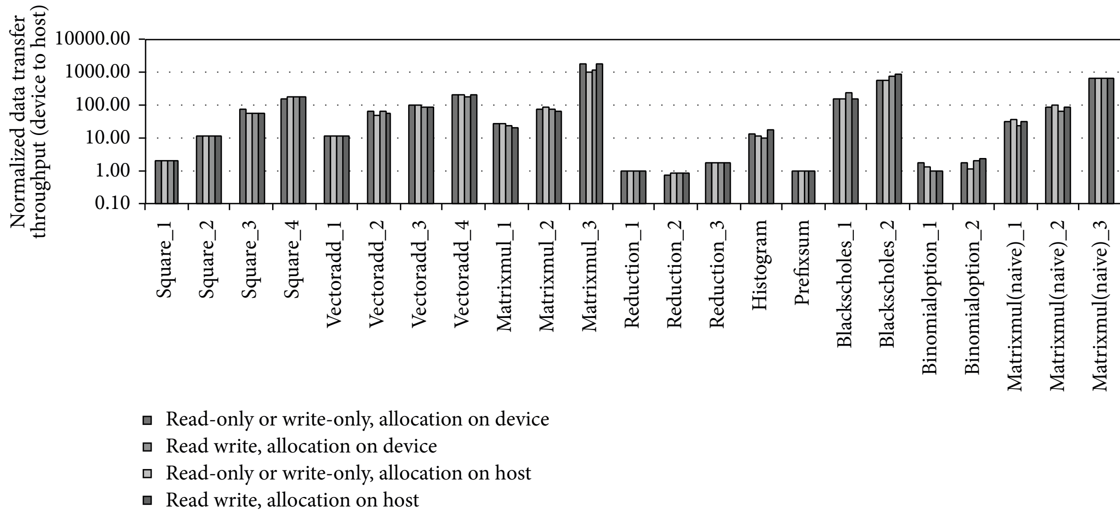


FIGURE 24: Normalized data transfer (device to host) throughput of mapping over explicit data transfer for all combinations on other dimensions.

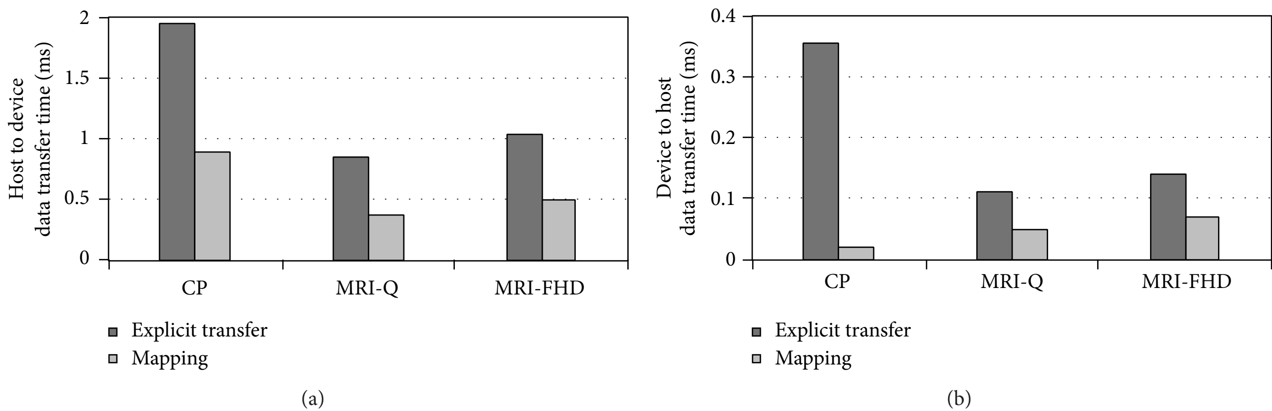


FIGURE 25: Data transfer time with different APIs for data transfer. (Left) host to device and (Right) device to host.

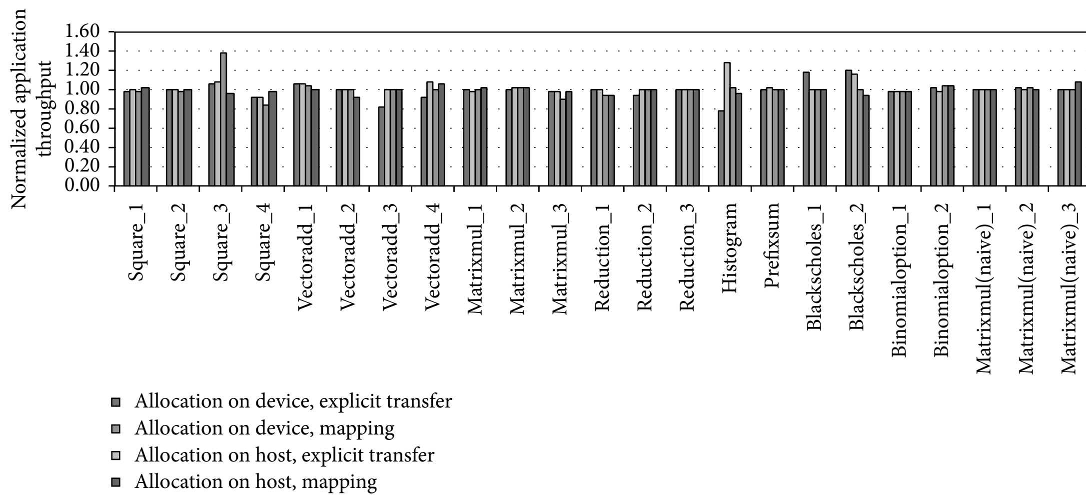


FIGURE 26: Normalized application throughput of read-only/write-only memory objects over read/write memory objects for all combinations on other dimensions. There is no noticeable performance difference.

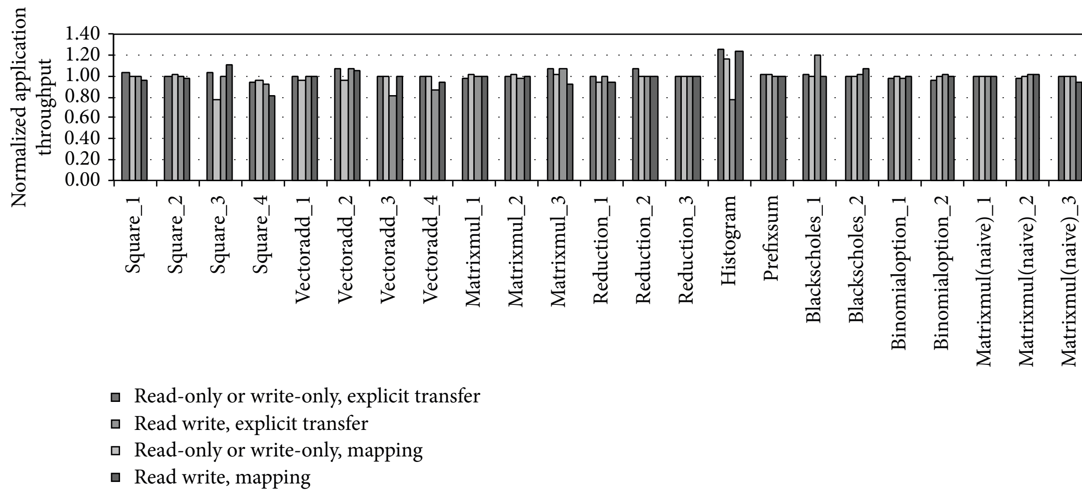


Figure 27: Normalized application throughput of the pinned memory over the device memory for all combinations on other dimensions. Where to allocate a memory object does not change the performance much on CPUs.

different allocation locations. The y-axis represents the normalized throughput when we allocate the memory object on the host memory from the baseline when we use the device memory. We find that an allocation location does not have a huge impact on performance both for kernel execution time and data transfer time. This is because the device memory and the host memory reference the same memory, the main memory of the system when the compute device is the CPU. Therefore, a different memory allocation location does not imply performance differences. On the contrary, when the compute device is not the CPU, memory allocation location can affect the performance significantly.

3.4.6. Summary. In this section we find that mapping APIs perform superior compared to explicit data transfer APIs with reduced data transfer time by eliminating the copying overhead on CPUs. Allocated location and kernel access type do not affect the performance on CPUs.

3.5. Vectorization and Thread Affinity

3.5.1. Vectorization. We evaluate the possible effect of programming models on vectorization, even though vectorization is more about compiler implementation. For evaluation, we port the OpenCL kernels to identical computations being performed by their OpenMP counterparts. We map multiple workitems on OpenCL to a loop to port OpenCL kernels to their OpenMP counterparts. We utilize the Intel C/C++ 12.1.3 compiler and the Intel OpenCL platform 1.5 for our evaluation. The programmer's expectation is that when we run the same computation in the OpenCL and OpenMP applications, both runs should give comparable performance numbers. However, the results show that this assumption does not hold. For the evaluated benchmarks, the OpenCL kernels outperform their OpenMP counterparts. Figure 28 shows the different performance of OpenMP and OpenCL implementations. The reason for this mismatch is the different way OpenMP and OpenCL compilers vectorize code.

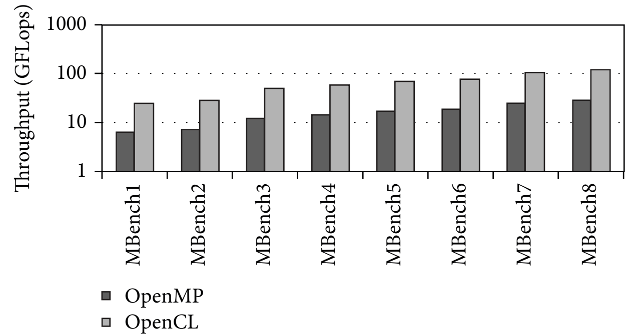


Figure 28: Performance impact of vectorization.

OpenCL Vectorization. The vectorization by the OpenCL kernel compiler is coalescing workitems. OpenCL vectorization enables the execution of several workitems together by a single vector instruction. Vectorization enables multiple work items to be processed concurrently on a single thread. For example, if the target instruction set is SSE 4.2, and the computation is based on a single precision floating point, then four workitems could make progress concurrently, so they are coalesced into a single workitem. By doing this, vectorized OpenCL code would have fewer dynamic instruction counts compared to nonvectorized code.

OpenMP Vectorization. On the other hand, the OpenMP compiler vectorizes loops by unrolling a loop combined with the generation of packed SIMD instructions. To be vectorized, a loop should be countable, have single entry and single exit, and have a straight control flow graph inside the loop [28]. Many factors could prevent the vectorization of a loop in OpenMP. Two key factors are (1) `non-contiguous memory access` and (2) `data dependence`.

(1) Noncontiguous memory access:

  (i) four consecutive floats may be loaded directly from the memory in a single SSE instruction;

```
/*OpenMP computation that doesn't vectorize due to dependencies.*/
int main(){
  ...
  for (int j = 0; j < 4; j++){
    FMUL(_a[j], _b[j])
    FMUL(_a[j], _b[j])
    FMUL(_a[j], _b[j])
    FMUL(_a[j], _b[j])
    FMUL(_a[j], _b[j])
    FMUL(_a[j], _b[j])
  }
  ...
}
/*Similar OpenCL kernel computation which vectorizes.*/
void VectorAdd (..., __global float *dm_src, __global float *dm_dst){
  ...
  for (int j = 0; j < 4; j++){
    FMUL(_a[j], _b[j])
    FMUL(_a[j], _b[j])
    FMUL(_a[j], _b[j])
    FMUL(_a[j], _b[j])
    FMUL(_a[j], _b[j])
    FMUL(_a[j], _b[j])
  }
  ...
}
```

Algorithm 1: Vectorization on OpenCL versus OpenMP. The equivalent code in OpenCL is vectorizable while OpenMP code is not vectorizable.

but if the four floats to be loaded are not consecutive, we will have a load using multiple instructions; loops with a nonunit stride are an example of the above scenario.

(2) Data dependence:

(i) vectorization requires changes in the order of operations within a loop since each SIMD instruction operates on several data elements at once; but such a change of order might not be possible due to data dependencies.

Example. Algorithm 1 shows an example of how different vectorization mechanisms from OpenMP and OpenCL compilers affect whether identical codes are to be vectorized or not. When there is a true data dependence inside an OpenCL kernel or inside a loop iteration in OpenMP `parallel for` section, the OpenCL kernel is vectorized, while the OpenMP code is not. Therefore, they show different performance even when vectorization of OpenMP loops seems possible. The vectorization of an OpenCL kernel is relatively straightforward because no dependency checks are required as in the case of traditional compilers. Even though we only show the example of when the OpenCL compiler shows the benefit, the opposite case is also possible: when the OpenMP compiler vectorizes code, but the OpenCL compiler fails.

New OpenMP Compiler. We have also evaluated OpenMP vectorization with OpenMP 4.0 SIMD extension and the newer compiler (Intel C/C++ compiler 15.0.1). The evaluation revealed comparable performance of OpenMP and OpenCL implementations. Compiler vectorization is dependent on the compiler implementation.

3.5.2. Thread Affinity. We evaluate the performance benefit using the CPU affinity in OpenMP. We use `OMP_PROC_BIND` and `GOMP_CPU_AFFINITY` to control the scheduling of threads on the processors [8]. When the `OMP_PROC_BIND` is set to be true, the threads will not be moved between processors. `GOMP_CPU_AFFINITY` enables us to control the allocation of a thread on a particular core.

We use a simple application for evaluation. The aim of the application is to verify the effects of binding threads to cores in terms of cache utilization. Performance can improve when the OpenCL run-time library maps logical threads of a kernel on physical cores so that it can utilize the cached data of the previous kernel execution. The application we use consists of two kernels: `Vector Addition` and `Vector Multiplication`. Computation of each kernel is distributed among eight cores: and the computation of second kernel is dependent on the first one, using the data produced by the first one.

Table 7 shows the method we use. The upper table in Table 7 represents the (a) `Aligned` case, and the lower table represents the (b) `Misaligned` case. The numbers in the table represent the logical thread IDs. Threads with identical IDs of both the kernels access the same data. On the (a) `Aligned` case, we bind threads of the second kernel to the cores on which the threads of the first kernel

Table 7: Performance impact of CPU affinity.

(a) Aligned

	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
Computation 1	0	1	2	3	4	5	6	7
Computation 2	0	1	2	3	4	5	6	7

(b) Misaligned

	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
Computation 1	0	1	2	3	4	5	6	7
Computation 2	6	3	4	0	2	1	7	5

```
/*First Kernel: Vector Addition.*/
#pragma omp parallel for shared(a, b, c) private (i)
for (int i = 0; i < MAX_INDEX; i++){
  c[i] = a[i] + b[i];
}
/*(a) Aligned Second Kernel: Vector Multiplication.*/
#pragma omp parallel for shared(b, c, d) private (i)
for (int i = 0; i < MAX_INDEX; i++){
  d[i] = b[i] + c[i];
}
/*(b) Misaligned Second Kernel: Vector Multiplication.*/
#pragma omp parallel for shared(b, c, d) private (i)
for (int i = 0; i < MAX_INDEX; i++){
    int j = MAX_INDEX − 1 − i;
  d[j] = b[j] + c[j];
}
```

Algorithm 2: Code snippet of simple application.

are bound. On the (b) `Misaligned` case, we shuffle this binding. Algorithm 2 shows the code snippet of this simple application.

As we expect, the (a) `Aligned` case shows higher performance than does the (b) Misaligned case. The (b) `Misaligned` one runs longer by 15%. This is because during the execution of the second kernel, the cores on the CPU encounter cache misses on their private caches. On the contrary, the (a) `Aligned` case would have more cache hits than the (b) `Misaligned` case because the data accessed by the second kernel would already be on the cache after the execution of the first kernel on the (a) `Aligned` case.

As the results show, even though OpenCL emphasizes portability, adding the affinity support to OpenCL may provide a significant performance improvement in some cases. Hence, we argue that coupling logical threads with physical threads (cores on the CPU) is needed on OpenCL, especially for CPUs. The granularity for the assignment could be a workgroup; in other words, the programmer can specify the core where a specific workgroup would be executed. This functionality would help to improve the performance of OpenCL applications. For example, data from different kernels can be shared without a memory request if the programmer allocates cores for specific workgroups in consideration of the data sharing of different kernels. The data can be shared through the private caches of cores.

4. Related Work

Multiple research studies have been done on how to optimize OpenCL performance on GPUs. The GPGPU community provides TLP [29] as a general guideline for optimizing GPGPU applications since GPGPUs are usually equipped with a massive number of processing elements. Since OpenCL has the same background as CUDA [9], most OpenCL applications are written to better utilize TLP. The widely used occupancy metric indicates the degree of TLP. However, this scheme cannot be applied on CPUs since even when the TLP of the application is large, the physical TLP available on CPUs is limited by the number of CPU cores, so that the context switching overhead is much higher on CPUs than on GPUs for which this overhead is negligible.

Several publications refer to the performance of OpenCL kernels on CPUs. Some focus on algorithms and some refer to the performance difference by comparing it with GPU implementation and OpenMP implementation on CPUs [16, 30, 31]. However, to the best of our knowledge, our work is the first to provide a broad summary, combining application with the architecture knowledge to provide a general guideline to understand OpenCL performance on multicore CPUs.

Ali et al. compare OpenCL with OpenMP and Intel's TBB on different platforms [30]. They mostly discuss the scaling effects and compiler optimizations. But it misses out on why the optimizations listed in the paper give the performance benefit mentioned and lacks quantitative evaluation. We, too, evaluate the performance of OpenCL and OpenMP for a given application. However, our work considers various aspects that can change application performance and provide quantitative evaluations to help programmers estimate the performance impact of each aspect.

Seo et al. discuss OpenCL performance implications for the NAS parallel benchmarks and give a nice overview of how they optimize the benchmarks by first getting an idea of the data transfer and scheduling overhead and then coming up with ways to avoid them [31]. They also show how to rewrite a good OpenCL code, given an OpenMP code. Stratton et al. describe a way to implement a compiler for fine-grained SPMD-thread programs on multicore execution platforms [16]. For the fine-grained programming model, they start

with CUDA, saying that it will apply to OpenCL as well. They focus on the performance improvement over the baseline. Our work is more generalized and broad compared to these previous studies and also includes some of the important points that are not addressed in these papers.

One of the references that is very helpful to understand the performance behavior of OpenCL is a document from Intel [32]. It broadly lays out some general guidelines to follow to get better performance out of OpenCL applications on Intel processors. However, it does not discuss the performance improvement and also does not state how much benefit can be achieved.

5. Conclusion

We evaluate the performance of OpenCL applications on modern multicore CPU architectures. Understanding the performance in terms of architectural resource utilization is helpful for programmers. In this paper, we evaluate various aspects, including API overhead, thread scheduling, ILP, data transfer, data locality, and compiler-supported vectorization. We verify the unique characteristics of OpenCL applications by comparing them with conventional parallel programming models such as OpenMP. Key findings of our evaluation are as follows.

(1) OpenCL API overhead is not negligible on CPUs (Section 3.2).

(2) Allocating more work per workitem therefore reducing the number of workitems helps performance on CPUs (Section 3.3.1).

(3) Large ILP helps performance on CPUs (Section 3.3.2).

(4) Large workgroup size is helpful for better performance on CPUs (Section 3.3.3).

(5) On CPUs, Mapping APIs perform superior compared to explicit data transfer APIs. Memory allocation flags do not change performance (Section 3.4).

(6) Programming model can have possible effect on compiler-supported vectorization. Conditions for the code to be vectorized can be complex (Section 3.5.1).

(7) Adding affinity support to OpenCL may help performance in some cases (Section 3.5.2).

Our evaluation shows that considering the characteristics of CPU architectures, the OpenCL application can be optimized further for CPUs, and the programmer needs to consider these insights for portable performance.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors would like to thank Jin Wang and Sudhakar Yalamanchili, Inchoon Yeo, the Georgia Tech HPArch members, and the anonymous reviewers for their suggestions and feedback. We gratefully acknowledge the support of the NSF CAREER award 1139083 and Samsung.

References

[1] AMD, AMD Accelerated Processing Units (APUs), http://www.amd.com/en-us/innovations/software-technologies/apu.

[2] Intel, “Products (Formerly Sandy Bridge),” http://ark.intel.com/products/codename/29900/Sandy-Bridge.

[3] Khronos Group, “OpenCL: the open standard for parallel programming of heterogeneous systems,” http://www.khronos.org/opencl.

[4] Intel, “Intel OpenCL SDK,” http://software.intel.com/en-us/articles/intel-opencl-sdk/.

[5] NVIDIA, “NVIDIA OpenCL SDK,” http://developer.nvidia.com/cuda/opencl/.

[6] J. Aycock, “A brief history of just-in-time,” *ACM Computing Surveys*, vol. 35, no. 2, pp. 97–113, 2003.

[7] Intel, Intel Threading Building Blocks, http://threadingbuildingblocks.org/.

[8] The OpenMP Architecture Review Board, OpenMP, http://openmp.org/wp/.

[9] NVIDIA, CUDA Programming Guide, V4.0, 2011.

[10] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-M. W. Hwu, “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’08)*, pp. 73–82, February 2008.

[11] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, and W.-M. W. Hwu, “Program optimization study on a 128-core GPU,” in *Proceedings of the 1st Workshop on General Purpose Processing on Graphics Processing Units (GPGPU '07)*, October 2007.

[12] S. Ryoo, C. I. Rodrigues, S. S. Stone et al., “Program optimization space pruning for a multithreaded GPU,” in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO ’08)*, pp. 195–204, 2008.

[13] V. Volkov and J. W. Demmel, “Benchmarking GPUs to tune dense linear algebra,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’08)*, pp. 31:1–31:11, November 2008.

[14] R. Balasubraamonian, S. Dwarkadas, and D. H. Albonesi, “Reducing the complexity of the register file in dynamic superscalar processors,” in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pp. 237–248, December 2001.

[15] J. Kim, S. Seo, J. Lee, J. Nah, and G. Jo, “SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters,” in *Proceedings of the 26th ACM International Conference on Supercomputing (ICS ’12)*, pp. 341–351, June 2012.

[16] J. A. Stratton, V. Grover, J. Marathe et al., “Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs,” in *Proceedings of the 8th International Symposium on Code Generation and Optimization (CGO ’10)*, pp. 111–119, ACM, April 2010.

[17] G. Diamos, “The design and implementation Ocelot’s dynamic binary translator from PTX to Multi-Core x86,” Tech. Rep. GIT-CERCS-09-18, Georgia Institute of Technology, 2009.

[18] B. Saha, X. Zhou, H. Chen et al., “Programming model for a heterogeneous x86 platform,” in *Proceedings of the ACM*

SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09), pp. 431–440, June 2009.

[19] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*, pp. 372–382, Galveston, Tex, USA, October 2011.

[20] L. Gwennap, "Intel's MMX speeds multimedia," Microprocessor Report, 1996.

[21] Intel, "Intel Integrated Performance Primitives," https://software.intel.com/en-us/intel-ipp.

[22] Intel, Intel Math Kernel Library, http://software.intel.com/en-us/intel-mkl.

[23] Intel, Intel C and C++ Compilers, https://software.intel.com/en-us/c-compilers.

[24] M. Pharr and W. R. Mark, "ispc: a SPMD compiler for high-performance CPU programming," in *Proceedings of the Innovative Parallel Computing (InPar '12)*, pp. 1–13, IEEE, San Jose, Calif, USA, May 2012.

[25] D. Grewe and M. F. P. O'Boyle, "A static task partitioning approach for heterogeneous systems using OpenCL," in *Proceedings of the 20th International Conference on Compiler Construction (CC '11)*, pp. 286–305, Saarbrücken, Germany, March 2011.

[26] The IMPACT Research Group and UIUC, "Parboil benchmark suite," http://impact.crhc.illinois.edu/Parboil/parboil.aspx.

[27] C.-K. Luk, R. Cohn, R. Muth et al., "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pp. 190–200, June 2005.

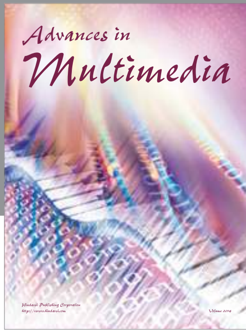
[28] Intel, A Guide to Auto-Vectorization with Intel C++ Compilers, http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers.

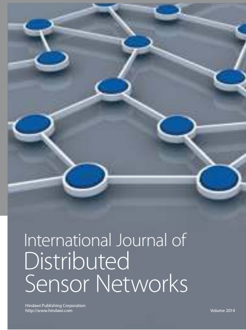
[29] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, pp. 152–163, June 2009.

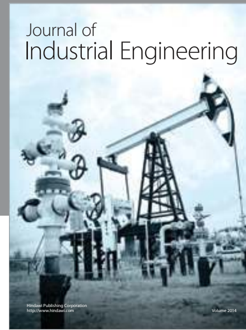
[30] A. Ali, U. Dastgeer, and C. Kessler, "OpenCL for programming shared memory multicore CPUs," in *Proceedings of the MULTIPROG Workshop at HiPEAC*, 2012.

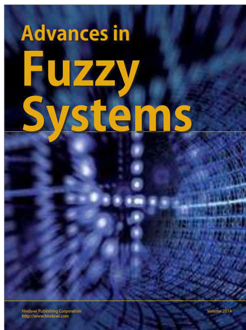
[31] S. Seo, G. Jo, and J. Lee, "Performance characterization of the NAS Parallel Benchmarks in OpenCL," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '11)*, pp. 137–148, Austin, Tex, USA, November 2011.

[32] Intel, "Writing Optimal OpenCL Code with Intel OpenCL SDK," http://software.intel.com/file/37171.









Hindawi

Submit your manuscripts at
http://www.hindawi.com