OpenCL Performance Evaluation on Modern Multi Core CPUs

Joo Hwan Lee, Kaushik Patel, Nimit Nigania, Hyojong Kim, Hyesoon Kim
School of Computer Science, College of Computing
Georgia Institute of Technology, Atlanta, GA, USA
Email: {joohwan.lee, kaushik.patel86, nnigania3, hkim606, hyesoon.kim}@gatech.edu

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

***Keywords*-OpenCL Performance on CPU; Scheduling Overhead; ILP; Data Transfer; Locality; Vectorization**

I. Introduction

The heterogeneous architecture has gained popularity as can be seen from Intel's Sandy Bridge, and AMD's Fusion[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 GPU architectures, 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 model.

The reasons we consider CPUs for OpenCL compute devices are (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) and (2) because a CPU has 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.

Here, we evaluate the performance of OpenCL applications on multicore CPUs from the architectural perspective, regarding how the application would utilize architecture resources on CPUs. We thoroughly evaluate OpenCL applications on various aspects that could change the performance of OpenCL applications. We revisit generic performance metrics that have been lightly evaluated in previous works especially for running OpenCL 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 the following:

1) We provide programmers with a guideline to understand the performance of OpenCL applications. Programmers can verify whether the OpenCL kernel fully utilizes the computing resources.
2) 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 to utilize OpenCL implementation for CPU execution with minimal tuning, which reduces the programming overhead to implement a separate CPU version. Some previous studies about OpenCL on CPUs handle some aspects presented in this paper, but they lack quantitative evaluations, making them hard to use when programmers want to estimate the performance impact of each aspect.

Section II describes the architectural aspects to understand OpenCL performance on CPUs. Then, we evaluate OpenCL applications regarding those aspects in section III. We review related work in section IV, and conclude the paper.

II. Backgrounds on Criteria

In this section, we describe the backgrounds of several parameters that affect OpenCL application performance. They include thread scheduling, instruction-level parallelism, data transfer, data locality, and compiler auto vectorization. These aspects have been emphasized in academia and industry to improve application performance on CPUs. 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.

A. Thread Scheduling

OpenCL programmer can explicitly set workgroup size, or let the OpenCL implementation decide it. If `NULL` is used for workgroup size when the host program calls `clEnqueueNDRangeKernel`, the OpenCL implementation partitions global workitems into appropriate number of workgroups.

Given a program, workgroup size determines the amount of workload in a workgroup, and the number of workgroups of a kernel. On GPUs, a workgroup or multiple groups is/are executed on a streaming multiprocessor (SM), which is equivalent to one physical core on the CPU. Similarly, a workgroup is handled by a logical core of the CPU, even though it depends on the implementation[6], [7]. Workload size per workgroup that is too small makes the workgroup scheduling overhead more significant in total execution time on CPUs since the thread context switching overhead becomes larger.

It is true that scheduling overhead is not a fundamental problem with the OpenCL model. Better OpenCL implementation can have less overhead than other suboptimal implementations. There have been many proposals to reduce the scheduling overhead [7], [8], [6]. For example, SnuCL [6] overcomes the overhead of a high number of workitems by serializing them to have less number of threads. However, it is also true that multiple OpenCL implementations on 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. The results from our experiments agree with the above inferences.

B. Instruction Level Parallelism (ILP)

One of the performance problems of OpenCL applications on CPUs is that usually the kernel is written to utilize the TLP not for ILP. Since the OpenCL programming model is an SIMT model, it is common for an OpenCL application to have a massive number of threads, and all instructions 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. On the contrary, on conventional programming model such as OpenMP, independent instructions exist between different loop iterations. For maximum performance on CPUs, OpenCL kernel should be written to have more independent instructions.

C. Memory Allocation, Data Transfer

OpenCL assumes a distributed memory system for its target, a system where communication between host and compute devices is 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 the 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 explicit 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 device[3], which becomes unnecessary when we only use the host for computation.

OpenCL provides the programmer many types of memory object allocation flags when the programmer calls `clCreateBuffer`, that could change the performance of data transfer and kernel execution.

1) First, programmers can specify if the memory object is a read-only memory object(`CL_MEM_READ_ONLY`) or write-only(`CL_MEM_WRITE_ONLY`) when referenced inside a kernel. If the programmer does not specify it, the default option is to create a memory object which can be read and written by the kernel(`CL_MEM_READ_WRITE`).
2) The other option that programmers can specify is where to allocate a memory object. When the programmer does not specify allocation location, the memory object is allocated on the device memory. 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 on the device memory, there is no need to transfer the computational result back from the device memory to the host memory.

OpenCL also provides different APIs for data transfer between the host and compute devices. The host code can enqueue commands to read data from a memory object to the host memory(`clEnqueueReadBuffer`) or commands to write data to memory object from the host memory(`clEnqueueWriteBuffer`). The programmer can also map a memory object for reading or writing to have the pointer of the mapped object(`clEnqueueMapBuffer`).

D. Affinity

Most conventional parallel programming model support affinity, such as `CPU_AFFINITY` in OpenMP[9]. Unfortunately, this feature 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 support for affinity as one of the performance limitations of OpenCL on CPUs and suggest a potential solution to enhance OpenCL performance on CPUs. We found the benefit of better utilizing cache on many OpenCL applications by utilizing affinity.

E. Vectorization

Utilizing SIMD units has been one of the key performance optimization techniques for CPUs [10]. Since SIMD instructions can perform computation on more than one data item at the same time, SIMD utilization could make the program more efficient. Many vendors have released various SIMD instruction extensions on their instruction set architectures, such as MMX[11].

Various methods have been proposed to utilize the SIMD instruction: using optimized function libraries such as Intel IPP[12] and MKL[13], using C++ vector classes with an Intel compiler[14], or using DSL compilers such as the Intel SPMD Program Compiler[15]. Programmers can also program in assembly or use intrinsic functions. To help programmers write programs utilizing SIMD instruction easily, auto vectorization has been implemented in many modern compilers [10], [14].

It is quite natural for programmers to expect that a programming model difference has no effect on compiler auto vectorization on the same architecture. For example, if an application written in both OpenCL and OpenMP share the same code structure, programmers would expect that both codes are vectorized in a similar fashion thereby giving similar performance numbers. Even though it depends on the implementation, it is not usually true. Unfortunately, today's compilers are very fragile about vectorizable patterns, which depends on the programming model. Programs should satisfy certain conditions in order to fully take advantage of compiler auto vectorization [10]. Our evaluation verifies the possible effect of programming models on vectorization.

III. Evaluation

A. Methodology

The experimental environment for our evaluation is described in Table I. Our evaluation was done on a heterogeneous computing platform, consisting of a multicore CPU and a GPU; the OpenCL application was executed on an Intel OpenCL platform[4], and NVidia OpenCL platform[5].

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

Table I
Experimental environment

First, we use simple applications and Parboil benchmarks by Grewe et al. [16]. Table II and III describe each application and their default characteristics.

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 significant enough running time, 90 seconds in our evaluation. This is sufficiently long to have a multiple number of kernel executions. Using the average kernel execution time per kernel invocation calculated, we use normalized throughput to present the performance differences of different architectures.

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 X 1600, 1600 X 3200, 4000 X 8000	16 X 16
Reduction	reduce	640000, 2560000, 10240000	256
Histogram	histogram256	409600	128
Prefixsum	prefixSum	1024	1024
Blackscholes	blackScholes	1280 X 1280, 2560 X 2560	16 X 16
Binomialoption	binomialoption	255000, 2550000	255
MatrixmulNaive	matrixMul	800 X 1600 1600 X 3200, 4000 X 8000	16 X 16

Table II
Characteristics of the Simple Applications

Benchmark	Kernel	global work size	local work size
CP	cenergy	64 X 512	16 X 8
MRI-Q	computePhiMag computeQ	3072 32768	512 256
MRI-FHD	RhoPhi FH	3072 32768	512 256

Table III
Characteristics of the Parboil Benchmarks

B. Thread Scheduling

1) Number of Workitems: To evaluate the effect of 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 maintain the total amount of computation the same, we reduce the the number of workitems to execute the kernel. The number of workitems coalesced increase from `1` to `1000` workitems by multiplying `10` for each step. Figure 1 shows the performance of `Square`, `Vectoraddition` applications with different amount of computation per workitem. Table IV shows the number of workitems used in this evaluation.

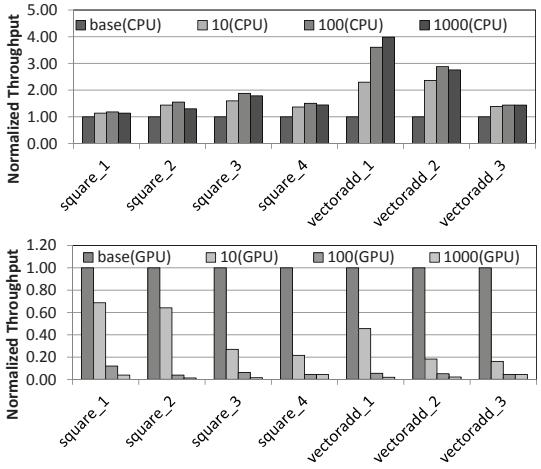

Figure 1. Performance of Square, Vectoraddition applications with different workload per workitem

Benchmark	base	10x	100x	1000x
Square_1	10000	1000	100	100
Square_2	100000	10000	1000	100
Square_3	1000000	100000	10000	1000
Square_4	10000000	1000000	100000	10000
VectorAdd_2	110000	11000	1100	110
VectorAdd_2	1100000	110000	11000	1100
VectorAdd_3	5500000	550000	55000	5500

Table IV
NUMBER OF WORKITEMS FOR EACH APPLICATION

From the figure, we find a performance gain for allocating more work per workitem on CPUs. A noticeable example is a case of vector addition, 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.

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

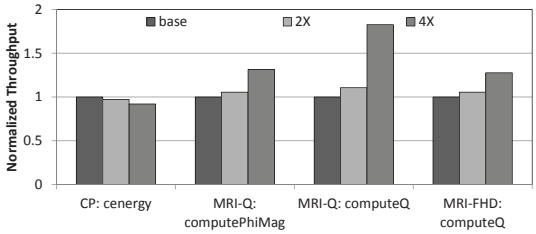

Figure 2. Performance of Parboil benchmarks with different workload per workitem

Figure 2 shows the performance of Parboil benchmarks with a similar experiment. The number of workitems coalesced increase from one to four workitems by multiplying two for each step. We find the same performance gain of allocating more work per workitem. The performance the `MRI-FHD:RhoPhi` kernel remains same with different workload per workitem.

2) Workgroup Size: We also evaluate the effect of the number of threads in workgroups, both on CPUs and GPUs. We vary the number of threads in a workgroup by changing the workgroup size (`local_work_size`) on kernel invocation. We maintain the total number of workitems of the kernel as the same. Table V shows the different workgroup size for each benchmark, and Figure 3 shows the performance of applications with different workgroup sizes. The `NULL` argument means that the workgroup size is dependent on OpenCL implementation.

Benchmark	base	case_1	case_2	case_3	case_4
Square	NULL	1	10	100	1000
VectorAddition	NULL	1	10	100	1000
Matrixmul	16X16	1X1	2X2	4X4	8X8
Blackscholes	16X16	1X1	1X2	2X2	2X4
MatrixmulNaive	16X16	1X1	2X2	4X4	8X8

Table V
WORKGROUP SIZE FOR EACH APPLICATION

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

`Square`, `Vectoraddition`, naive implementation of `Matrixmul`, shows performance increase with increased workgroup sizes on CPU. On `Square`, `Vectoraddition` applications, performance achieved with NULL workgroup size is less than the peak performance we achieve. This implies that programmers should explicitly set the workgroup size for the maximum performance. The performance

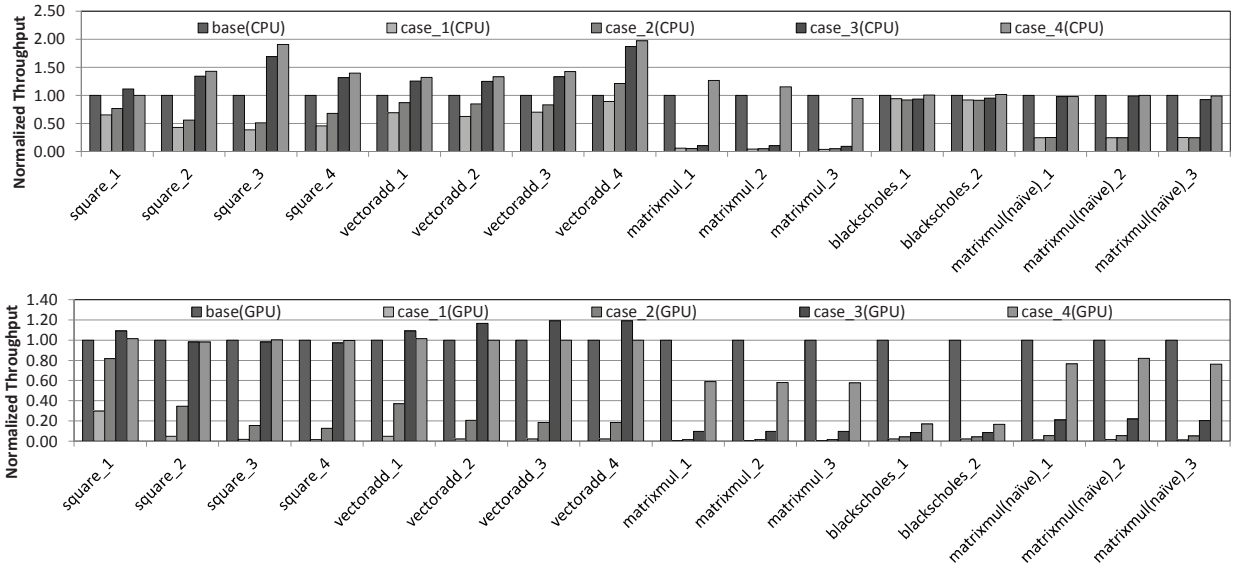

Figure 3. Performance of applications with different workgroup size on CPUs and GPUs

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[1], performance increases with a large workgroup size. It is because the overhead of managing a large number of workgroups, many number of threads in many implementations, is reduced. We can also find that performance is saturated at a certain workgroup size.

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 workload size on CPUs is `8 X 8`, but the optimal size on GPUs is `16 X 16`. This is because `Matrixmul` utilizes the local memory in OpenCL by blocking and 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 workload can be different.

Unlike other applications, `Blackscholes` shows different performance behavior between on CPUs and on GPUs. As we can see on Figure 4, 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, overhead of managing a large number of workgroups becomes negligible. On the contrary, number of warps in a SM is limited by the workgroup size on GPUs, which makes the performance on GPUs low on small workgroup sizes.

[1]The evaluated CPUs have SMT processors, so multiple logical cores share one physical core.

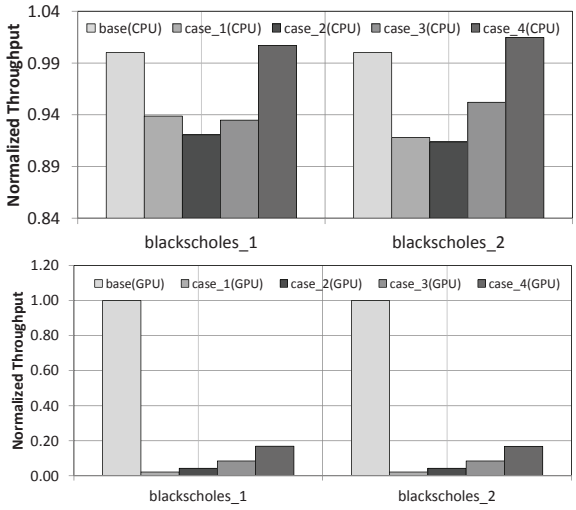

Figure 4. Performance of blackscholes with different workgroup size on CPUs and GPUs

Figure 5 shows the performance of Parboil benchmarks with different workgroup sizes. We increase the workgroup size from one to `16` times by multiplying `2` for each step. Since the workgroup size for `CP:cenergy` kernel is two-dimensional, we increase workgroup size of the kernel in two directions. `CP:cenergy(x)` represents the performance with workgroup sizes `1X8`, `2X8`, `4X8`, `8X8`, `16X8`. `CP:cenergy(y)` represents the performance with workgroup sizes `16X1`, `16X2`, `16X4`, `16X8`, and `16X16`. In general, we find the performance gain with a large workgroup size. The performance saturates when there is enough computation inside the workgroup.

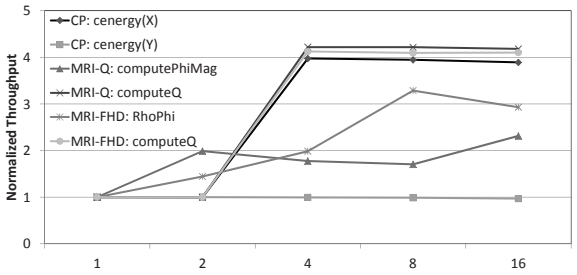

Figure 5. Performance of Parboil benchmarks with different workgroup size on CPUs

C. Instruction-Level Parallelism (ILP)

To evaluate the ILP effect on both CPUS and GPUs, we implemented a set of micro-benchmarks that share common characteristics. Each benchmark has an identical number of memory accesses, computations, and loop iterations inside the kernel. The only difference between each benchmark is the ILP by varying number of independent instructions. For example, in the case of `ILP 1`, the next instruction depends on the output of the previous instruction; but in the case of `ILP 2`, there is an independent instruction between two dependent instructions.

Figure 6 shows the performance with increasing ILP for the same kernel. We provide enough number of workitems to fully utilize TLP. The number of workitems remains same for all micro-benchmarks. The left y-axis represents throughput of the CPUs, and right one represents the one of 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 instruction-level parallelism.

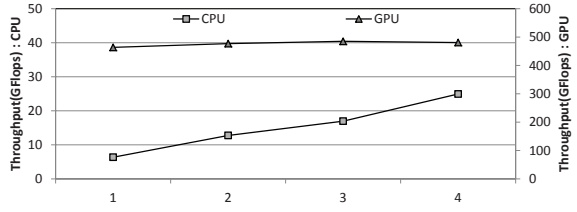

Figure 6. Performance of ILP micro-benchmark on: (CPU) Intel Xeon E5645, (GPU) GTX580

D. Memory Allocation, Data Transfer

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 these options. The combination we use is three dimensional. The options for our evaluation are described as follows. To measure exact execution performance, we use a blocking call for all kernel execution commands, and memory object commands.

1) APIs for data transfer :
   a) Copy : `clEnqueueReadBuffer`, `clEnqueueWriteBuffer` for explicit read and write
   b) Mapping : `clEnqueueMapBuffer` with `CL_MAP_READ`, `CL_MAP_WRITE` for read and write
2) Where to allocate a memory object :
   a) Allocation on a compute device memory
   b) Allocation on the host accessible memory on the host(pinned memory)
3) Kernel access - when referenced inside a kernel:
   a) Memory object is read-only/write-only : `CL_MEM_READ_ONLY` for an input to a kernel, `CL_MEM_WRITE_ONLY` for computation results
   b) Memory object is read/write : `CL_MEM_READ_WRITE` for all memory objects

The throughput we present here is the computational performance including data transfer time between the host and compute devices as shown in Equation (1).

$$Throughput\_app = \frac{Throughput\_kernel}{kernel\_time + transfer\_time} \quad (1)$$

We compare the performance of different data-transfer APIs on all possible allocation flags. Figure 7 shows the performance of the benchmarks with different APIs for data transfer. The y-axis represents the normalized throughput when we use mapping for data transfer from the baseline when we explicitly read and write. From the results, we find mapping APIs have superior performance compared to explicit data transfer, regardless of the decision on other dimensions. Mapping APIs perform superior wherever the memory object is allocated. Mapping APIs perform better regardless of the decision of allocating the memory object as read-only/write-only, or as read/write object.

The different performance of different APIs comes from the different data transfer time. Different APIs do not affect the kernel execution time. The data transfer time is shorter with mapping APIs. The performance gap increases with increases in workload sizes and therefore, increases in data transfer sizes.

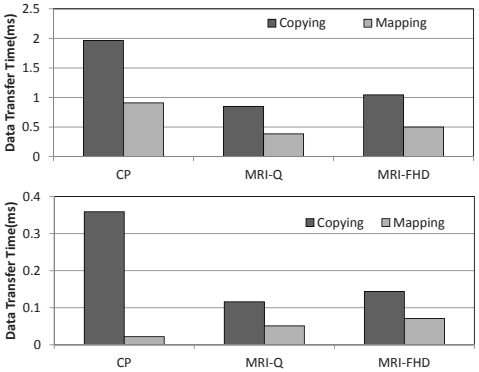

Figure 8. Data transfer time with different APIs for data transfer. (Upper) host to device, (Lower) device to host

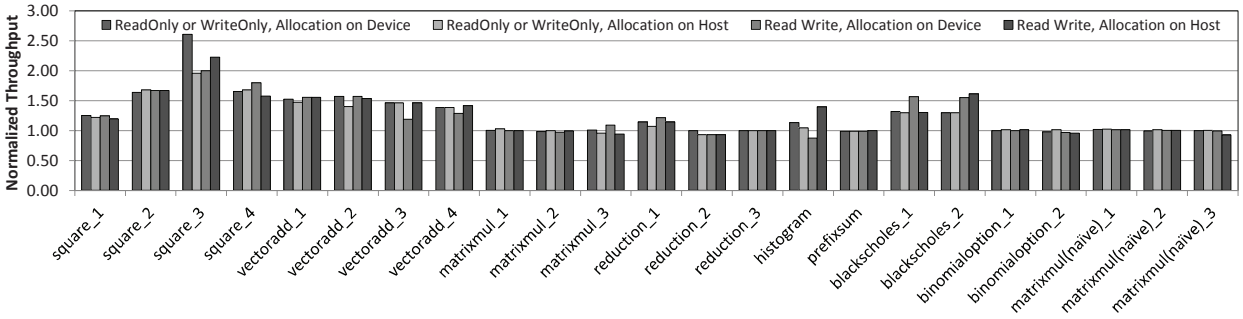

Figure 7. Normalized application throughput of mapping over copying for all combinations on other dimension. Mapping APIs perform superior to explicit data transfer on all possible combinations.

We also report the performance of Parboil benchmarks with different APIs for data transfer. Since the data transfer time is much shorter than the kernel execution time on Parboil benchmarks, instead of using application throughput as shown in Equation (1), we report the data transfer time from the host to device, and data transfer time from the device to host with different APIs. Different APIs do not affect the kernel execution time. Figure 8 show the performance of the Parboil benchmarks with different APIs for data transfer. Upper figure in Figure 8 shows the data transfer time from the host to a compute device with different data transfer APIs. Lower figure shows one from compute device to host. As with simple applications, we can find that the data transfer time is shorter with mapping APIs on these benchmarks.

The difference of data transfer time comes from the different behavior of different APIs. When the host code explicitly transfers data between the host and device, the OpenCL runtime should allocate a separate memory object and copy the data between the memory object. However, copying is not needed when the host code uses mapping; only returning a pointer is needed.

We also verify the performance effect of the allocation location of memory objects, Programmers can allocate the memory object on host memory or device memory. We find that allocation location does not have a major impact on performance. This is because device memory and host memory reference the same main memory of the system.

Finally, we verify the performance effect of specifying a memory object as read only or write only. We do not see a noticeable performance difference, which we omit in this paper for brevity.

E. 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[9]. 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 to a particular CPU. The aim is to verify the effects of mapping of a computation in terms of cache utilization. We use two kernels: `Vector Addition` and `Vector Multiplication`. The work is distributed among eight cores; and the second computation is dependent on the first one, in the sense of using the data produced by the first one. Figure 9 shows the method we use. The upper figure in Figure 9 represents the `aligned` case, and the lower figure represents the `misaligned` case. On the `aligned` case, we allocate computations of the second kernel on that threads that access the same data that is already accessed during the execution of the first kernel. On the `misaligned` case, we change this mapping. As we expect, the `aligned` case shows higher performance than does the `misaligned` case . The `misaligned` one runs longer by `15` percent. This is because during the second computation, the processors encounter cache misses on their private caches.

	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

(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	6	3	4	0	2	1	7	5

(b) misaligned

Figure 9. Performance Impact of CPU affinity

As the results shows, 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 physicals threads is needed on OpenCL, especially for CPUs. The granularity for the assignment could be workgroup; in other words, the programmer can specify the core where specific workgroup would be executed, so that data on different kernels can be shared without a memory request if the programmer allocates cores on specific workgroups in

consideration of data sharing of different kernels.

F. 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++ compiler and the Intel OpenCL platform for our evaluation. The expectation is that when we run the same computation in the OpenCL and OpenMP programs, 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 10 shows the performance of OpenMP and OpenCL implementations.

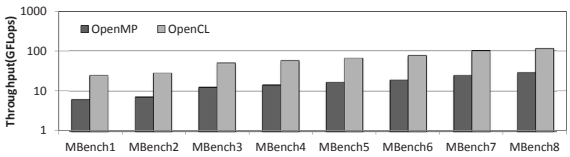

Figure 10. Performance impact of vectorization

The reason for this mismatch is the different way OpenMP and OpenCL compilers vectorize code. The OpenCL kernel compiler tries to vectorize in a way that allows it to execute several workitems together by a single vector instruction. 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 less thread creation compared to non-vectorized code.

On the other hand, the compiler tries to vectorize OpenMP programs, 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 a straight control flow graph inside the loop[17].

There are many factors that could prevent the vectorization of a loop in OpenMP. Two key factors are non-contiguous memory access and data dependence.

1) Noncontiguous memory access: Four consecutive floats may be loaded directly from the memory in a single SSE instruction. But if the four floats to be loaded are not consecutive, we will have a load using multiple instructions. Loops with a nonunit stride is an example of the above scenario.
2) Data dependence: 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.

```
...
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])
}
...
```

Figure 11. Vectorization on OpenCL vs. OpenMP. (The equivalent code in OpenCL can be vectorizable but this OpenMP code cannot be vectorizable.)

Figure 11 shows an example of different vectorization mechanisms from OpenMP and OpenCL compilers. 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 isn't. 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.

IV. Related Work

Multiple research studies have been done on how to optimize OpenCL performance on GPUs. The GPGPU community provides TLP[18] 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[19], 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.

There have been several publications referring to the performance of OpenCL kernels on CPUs. Some of them focus on algorithms and some of them refer to the performance difference by comparing it with GPU implementation and OpenMP implementation on CPUs[20], [21], [22], [7]. 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 multi core CPUs.

Ali et al. compare OpenCL with OpenMP and Intel's TBB on different platforms[21]. They mostly discuss the scaling effects and compiler optimizations. But it misses out on why the optimizations listed in the paper give 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[22]. 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[7]. 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 documentation from Intel[23]. 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.

V. 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 thread scheduling, ILP, data transfer, 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) Large workgroup size is helpful for better performance on CPUs.
2) Large ILP helps performance on CPUs.
3) On CPUs, Mapping APIs perform superior compared to explicit data transfer APIs. Memory allocation flags do not change performance.
4) Adding affinity support to OpenCL may help performance in some cases.
5) Programming model can have possible effect on compiler-supported vectorization. Conditions for the code to be vectorized can be complex.

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.

Acknowledgments

We 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] Intel, "Sandy Bridge," http://software.intel.com/en-us/articles/sandy-bridge/.
[2] AMD, "Fusion," http://fusion.amd.com/.
[3] OpenCL, "The open standard for parallel programming of heterogeneous systems," http://www.khronos.org/opencl.
[4] Intel Corporation, "Intel OpenCL SDK," http://software.intel.com/en-us/articles/intel-opencl-sdk/.
[5] NVIDIA Corporation, "NVIDIA OpenCL SDK," http://developer.nvidia.com/cuda/opencl/.
[6] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," in *ICS-26*, 2012, pp. 341–352.
[7] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu, "Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs," in *CGO-8*, 2010, pp. 111–119.
[8] G. Diamos, "The Design and Implementation Ocelot's Dynamic Binary Translator from PTX to Multi-Core x86," Georgia Institute of Technology, Tech. Rep. GIT-CERCS-09-18, 2009.
[9] "OpenMP," http://openmp.org/wp/, The OpenMP Architecture Review Board.
[10] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An Evaluation of Vectorizing Compilers," in *PACT '11*, 2011, pp. 372–382.
[11] L. Gwennap, "Intel's MMX speeds multimedia," *Microprocessor Report*, Mar. 1996.
[12] "Intel Integrated Performance Primitives," http://software.intel.com/en-us/intel-ipp-archive, INTEL.
[13] "Intel Math Kernel Library," http://software.intel.com/en-us/intel-mkl, INTEL.
[14] ICC, "Intel c++ compiler," http://www.intel.com/cd/software/products/asmo-na/eng/compilers/clin/277618.htm.
[15] M. Pharr and W. R. Mark, "ispc: A SPMD Compiler for High-Performance CPU Programming," in *InPar 2012*, May 2012.
[16] D. Grewe and M. F. P. O'Boyle, "A static task partitioning approach for heterogeneous systems using OpenCL," in *CC'11*, 2011, pp. 286–305.
[17] Intel Corporation, "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/.
[18] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," in *ISCA*, 2009, pp. 152–163.
[19] *CUDA Programming Guide, V4.0*, NVIDIA Corporation.
[20] Intel Corporation, Intel Corporation, http://software.intel.com/.
[21] U. D. Akhtar Ali and C. Kessler, "OpenCL for programming shared memory multicore CPUs," in *MULTIPROG-2012 Workshop at HiPEAC-2012*, 2012.
[22] S. Seo, G. Jo, and J. Lee, "Performance characterization of the NAS Parallel Benchmarks in OpenCL," in *IISWC'11*, 2011, pp. 137–148.
[23] Intel Corporation, "Writing Optimal OpenCL Code with Intel OpenCL SDK," http://software.intel.com/file/37171.