Performance of OpenCL in MultiCore Processors

Salman Zaidi
University of Kaiserslautern, Embedded Systems Group
zaidi@cs.uni-kl.de

Abstract

OpenCL (Open Compute Language) is one of the popular cross platform parallel programming frameworks. However, most of the implementations focus mainly on parallelism using GPUs (Graphics Processing Units). With the advancement in modern multicore CPUs (Central Processing Units) and heterogeneous architecture, significant performance improvements can be seen in parallel computing using OpenCL. For that purpose, it is needed to figure out how current implementations affect performance in CPUs, and what changes are needed to fix these issues. Studies focusing on architectural constraints of the framework are looked upon to figure out possible solutions. In addition to that, CPU performance is compared between OpenMP (Open Multi-Processing) and OpenCL implementations. Another important domain in terms of parallel computing is its use in embedded systems, and the importance of OpenCL in heterogeneous platforms. For that purpose, OpenCL is compared with related frameworks for embedded systems. The results for all these studies verify that OpenCL can lead to better performance on multicore CPUs. The studies also provide new developers guidelines in obtaining best performance.

1 Introduction

In recent times, general purpose CPUs have failed to increase their performance at a larger scale as predicted by Gorden Moore [1]. Due to this reason the use of sequential code in processing is not seeing major improvements. In order to tackle such problem and reduce costs, programmers move toward parallel programming models.

With the introduction of AMD's (Advanced Micro Devices) Fusion and Intel's Sandy Bridge, there has been a rise in the popularity of heterogeneous architecture [2, 3]. Studies have pointed out many performance and energy efficiency advantages can be achieved by the use of these processors, however, building proper tools for that purpose is still a problem.

OpenCL has been designed to tackle this issue. It is a parallel processing framework which is cross platform and supports heterogeneous hardware. Programs written in OpenCL are cross platform and cross vendor, which means they can run on CPUs, GPUs, and other processors without modifying code [4]. Many vendors support OpenCL and have released frameworks for their platforms [5, 6]. Similarly, OpenMP [7] is another easy to use framework focusing multi-core CPUs, which is specifically designed for shared memory parallel machines. Parallel execution in OpenMP is achieved by annotating sequential C, C++, or Fortran code with compiler directives.

The major advantage of using OpenCL is portability, but it proves to be an issue in terms of performance. There have been unpredictable performance variations on different platforms, and optimization techniques usually differ depending on the architecture. As OpenCL is similar to CUDA (Compute Unified Device Architecture), a framework for Nvidia GPUs, there has been a lot of study on code translation
and optimization between the two frameworks [8, 9], which has resulted in abundance of GPU related code, and most applications are not optimized for multi-core CPUs, and perform poorly considering conventional programming methods.

OpenCL compute devices lead to the efficient use of heterogeneous platforms by integrating CPUs and GPUs for performance improvements. Modern CPUs have more vector units, which reduce the performance gap between CPUs and GPUs, also making CPUs perform better in some cases depending on input size and support for Instruction-Level Parallelism [10].

By optimizing CPU related code on OpenCL to reach OpenMP performance, extra effort can be avoided of developing solutions independent for CPU and GPU, mainly because OpenCL allows both platforms to share the same parallelism approach, and OpenMP only provides high level parallelism on a single platform [11]. OpenCL also provides more control in tuning the application due to its hardware abstraction.

Discussion above highlights possible performance gains achieved by using OpenCL in multi-core CPUs. Therefore, it is important to analyze and find out factors which impact performance of OpenCL applications on CPUs. In this paper, multiple studies are portrayed which evaluate the performance of OpenCL applications on modern multi-core CPUs. These focus the architectural perspective of how applications utilize resources on CPUs. OpenMP and OpenCL is also compared by fine tuning OpenMP implementations for OpenCL to isolate and identify significant issues.

These studies provide guidelines for new programmers to understand OpenCL architecture and figure out if their applications are utilizing CPU resources efficiently. In addition to that, CPU effectiveness on OpenCL is also looked upon. This would help developers understand OpenCL on CPUs to implement optimized applications from scratch, as well as port OpenMP applications to OpenCL.

Section 2 explains related work. Section 3 describes the OpenCL architecture. Section 4 specifies the architectural aspects to understand the OpenCL performance on CPUs. Section 5 contains evaluation of OpenCL applications based on those aspects. Section 6 discusses the findings and concludes the paper.

2 Related Work

In the domain of parallel programming and embedded systems, SysML (Systems Modeling Language) and AADL (Architecture Analysis & Design Language) are some of the known tools used to develop software and hardware models of real time embedded systems. The main difference between OpenCL and these languages is that OpenCL is used to write multi platform code for existing supported systems [4]. Whereas, these languages can be utilized to create a totally new system or architecture to perform a specific task [12].

The main purpose of these languages is to define a problem as a model using various modeling languages. This model is then used to auto generate code. However, the code provided is not specific to any platform and only provides general implementations. Programmers then have to write specific code for performance improvements. This in turn makes the applications platform specific as compared to OpenCL which is platform independent.

3 OpenCL Platform

OpenCL is a framework developed by the KHRONOS group to write parallel programs that execute across heterogeneous systems. Programs written in OpenCL are cross platform and cross vendor, which means they can run on CPUs, GPUs, and other processors without modifying code. OpenCL provides
an abstraction for low level hardware routines as well as consistent memory and execution models to support massively parallel code execution. The OpenCL architecture describes how hardware is mapped in OpenCL terminology. Furthermore, to understand the problem in detail, OpenCL execution model and OpenCL memory model are discussed.

3.1 OpenCL Architecture

Figure 1 shows an overview of the OpenCL architecture. An OpenCL platform runs on a CPU host which is connected to one or more compute devices. A compute device can be a CPU, GPU, or any other processing device. The host is responsible for controlling all these compute devices. Each compute device can have multiple Compute Units (CU) (similar to cores). Every compute unit has multiple Processing Elements (PE) on which OpenCL Kernels are executed [13]. A kernel is the basic unit of executable code, similar to a C function. Due to architectural differences between vendors and platforms, it is difficult to understand the mapping of compute units and processing elements to actual hardware, but compute units can be considered cores (such as CUDA cores) on the hardware level, and processing elements are the threads (or SIMD lanes in GPUs). An OpenCL program is defined on an N-Dimensional range problem space, where N can be 1, 2, or 3. The NDRange is further discussed in the OpenCL execution model.

![Figure 1: The OpenCL platform architecture.](image)

3.2 OpenCL Execution Model

Figure 2 shows the OpenCL execution model describing how OpenCL divides its problem to a compute device such as CPU or GPU, and its compute units such as CUDA cores. An OpenCL program can be divided into two parts. The first part is run on the host and is responsible for the whole execution of the code, which starts from assigning workload and controlling compute devices to collecting results. The second part are the OpenCL Kernels which are considered the basic element of execution [11]. OpenCL kernels execute exactly once for each point defined in the index space which is known as NDRange [14]. In OpenCL these points are known as work items.
The concept of NDRange can be understood by an example of a simple addition of two arrays inside a for loop in C/C++.

```c
for (int i=0; i<10; i++)
{
    c[i] = a[i] + b[i]
}
```

The index space in this case would be all 10 statements executed inside this loop [15]. Unlike the loops in C which are executed sequentially and in-order, OpenCL is free to execute these work items in parallel and in any order. This allows OpenCL to take advantage of parallel computing resources. A single work item is executed by a PE, therefore, work items can be combined into a work group to be assigned to a CU. Work items inside a work group can share data using local memory. The difference between a work item and a processing element is that a work item is the software which a processing element executes, whereas the processing element is the actual mapping of a hardware performing that execution.

A programmer defines the NDRange for execution, which is the total number of work items, and is called the global size in the OpenCL API. Based on available resources, a programmer has the option set a work group size at runtime, which is known as the local size in the OpenCL API. There is also an option to let OpenCL automatically select a suitable local sized based on processing capabilities of the system.
3.3 OpenCL Memory Model

Figure 3 provides an overview of the OpenCL memory model. There are four distinct memory regions in OpenCL. Global memory can be accessed for reading and writing by all the work items. Constant memory is a global memory region for constants and is read only during kernel execution. Local memory is shared between all work items in a single work group. This type of memory can be mapped either on the compute device or as a part of the global memory depending on the hardware. Private memory is a region only accessible by a single work item [14].

Work groups communicate through shared memory and synchronization primitives, however they have independent memory access. Due to this, OpenCL uses a relaxed consistency shared memory model. In a single work group, consistency between work items is insured by a work group barrier. An in-order command queue or synchronization points are used to make memory updates visible to all work items [11].

4 Criteria

Majority of OpenCL applications focus mainly on parallelism in GPUs, and due to the difference in CPU and GPU architecture, the code does not scale well on CPUs. To understand this lack of performance
and make better use of the features provided by OpenCL, a deeper look is needed from the architectural perspective.

In order to build better performing applications, various aspects such as API overhead, scheduling overhead, instruction level parallelism, address space, data location, data locality, and vectorization have to be understood. The main objective is to compare how they differ from conventional parallel programming models for CPUs, and based on that, try to fine tune and optimize OpenCL applications for better CPU performance [10].

This in turn leads to develop a guideline for programmers interested in switching to heterogeneous architecture and cross platform support, making it easier for them to identify and avoid bad practices and implement efficient code.

This section further takes a look at above mentioned aspects in detail, although they are commonly emphasized in academia as well as industry, most OpenCL applications are not implemented following these aspects.

4.1 API Overhead

OpenCL carries a greater overhead for launching kernels compared to other conventional parallel programming models for CPUs, where it is almost negligible. OpenCL host program also coordinates the execution which carries an API overhead due to specific OpenCL API function calls.

The main reason for this overhead is the support of multiple platforms and combinations of hardware. Conventional models for parallel processing do not need a separate context for their working because they are only running on a single type of processing unit as compared to OpenCL which has to coordinate between multiple different types of processing units.

Another reason is the usage of just in time compilation during run-time [16]. In other programming models, compilation is statically done. This compile time adds an overhead in OpenCL execution time. In many cases, it is noted that these compilation times actually exceed the general execution time of an OpenCL application.

4.2 Thread Scheduling

As already mentioned, OpenCL follows a Single-instruction, multiple thread (SIMT) model [17], which is defined in OpenCL as having work groups and work items. In terms of performance, programmers have the choice to tune the number of work items, which results in increasing or decreasing work group size. The choice of work group size directly affects performance based on the type of hardware being used. This can be further explained in terms of work items and work groups.

4.2.1 Number of work items

Generally, GPUs are capable of performing a large number of very small tasks, whereas CPUs on the other hand, have the capability to perform bigger tasks in a smaller amount. This is due to the architectural differences between the two platforms, CPUs are limited by the number of cores and cannot achieve high thread level parallelism. In terms of OpenCL, if there are a large number of work items performing a short work, it will affect CPU performance compared to GPUs.
4.2.2 Number of work items and instruction level parallelism

Another reason for poor performance on CPUs is the lack of instruction level parallelism in OpenCL implementations. Compared to OpenMP, where programs are generally written to utilize features common in CPUs such as superscalar executions and branch prediction \[13\], OpenCL programs are written to cater every kind of platform, which results in only having thread level parallelism.

4.2.3 Workgroup size

Generally, a workgroup is executing on a streaming multiprocessor on a GPU, which is equivalent to a physical core on a multicore CPU. A smaller workload size per work group leads to creating a significant scheduling overhead on CPUs, because it increases the thread context switching overhead. OpenCL programmers have the choice to explicitly set a workgroup size or leave it on the OpenCL implementation to decide. This is done by returning a value when the OpenCL implementation calls clEnqueueNDRangeKernel. If NULL is passed, the implementation automatically partitions global work items into an appropriate number of work groups.

4.3 Memory Allocation and Data Transfer

OpenCL has the option to choose from unified or disjoint memory space \[19\]. Normally, CPU programs use the unified memory space both for sequential and parallel code. This is mainly because there is no need to transfer data to different compute devices in such case, which leads to writing programs the same way in OpenCL implementations because of familiarity with the method.

OpenCL programs have a lot of options in terms of memory allocation and data transfer, based on what type of architecture they are utilizing, they can set up different parameters to store memory at specific levels and utilize different data transfer APIs. As data has to be transferred between host and compute device for execution \[4\], in the case of using CPUs, this creates unnecessary work as the compute device is the same as host. However, these rewriting efforts can be a burden on programmers due to lack of specific knowledge in most cases.

4.3.1 Memory Allocation Flags

Whenever the programmer calls clCreateBuffer, OpenCL provides multiple options for memory object allocation flag which could lead to the variation of performance of data transfer and kernel execution. The flag is responsible for specifying the access type of the object and the location where it is allocated \[10\].

4.3.2 Access type

Programmers can choose to specify if the memory object is read only (CL_MEM_READ_ONLY) or write only (CL_MEM_WRITE_ONLY) when reference inside a kernel. Generally, memory which is used as input can be marked as read only, and output can be marked as write only. If the access type is not specified, the default option is to use both read and write access (CL_MEM_READ_WRITE) which can also be explicitly defined \[10\].
4.3.3 Where to allocate

By default, when the programmer does not specify a location, the memory object is allocated on the OpenCL compute device. By using the flag CL_MEM_ALLOC_HOST_PTR, the memory is allocated on the host. When using this flag, there is no need to transfer any results back to host from compute device.

4.3.4 Different data transfer APIs

In OpenCL, programmer has also the option to transfer data between host and compute devices. This could be viewed as implicit and explicit data transfer, where in explicit transfer, the host enqueue commands to read data from an OpenCL memory object created by clCreateBuffer to its own memory object mostly created by malloc call (using clEnqueueReadBuffer API). Similarly, the host can also write data using clEnqueueWriteBuffer API. Whereas, in implicit transfer, there an option available to have a host accessible pointer mapping an OpenCL memory object using clEnqueueMapBuffer API [10].

4.4 Vectorization

One of the key techniques for optimization in CPUs is to utilize SIMD units, which allows computation on more than one data item at a time. Many vendors have extended their instruction set architectures by releasing various SIMD instructions, such as MMX [20].

Most of the methods proposed to utilize SIMD instructions leads towards rewriting the code, which is difficult. Due to this, many modern compilers have implemented autovectorization [21][22].

However, programmers expect that writing a similar program in OpenCL and OpenMP would result in same type of vectorization. But this is not the case as compilers are very fragile about vectorizable patterns, which differ based on the programming model. To take full advantage of autovectorization, applications have to specify certain conditions.

4.5 Thread affinity

Another important factor in performance is to figure out where to place threads on CPUs. There could be a latency difference between connections as threads adjacent to each other communicate faster. Proper placement could lead to eliminating communication overhead. For this reason, most conventional parallel processing models support affinity, such as CPU_AFFINITY in OpenMP [7].

In the case of OpenCL, unfortunately, no such support is available. This is due to the reason that OpenCL focuses more on portability than efficiency, and work items which are considered logical threads, are not tightly coupled with a physical thread.

5 Evaluation

The effects of architectural constraints are quantitatively studied and presented in the evaluation. A comparison is made on how discussed criteria affects the performance and differs from anticipated results. This section further describes the methodology of the evaluation, and focuses on the results based on the criteria described.
<table>
<thead>
<tr>
<th>CPUs</th>
<th>Intel Xeon E5645</th>
</tr>
</thead>
<tbody>
<tr>
<td># Cores</td>
<td>4</td>
</tr>
<tr>
<td>Vector width</td>
<td>SSE 4.2, 4 single precision FP</td>
</tr>
<tr>
<td>Caches</td>
<td>L1D/L2/L3: 64 KB/256 KB/12MB</td>
</tr>
<tr>
<td>FP peak performance</td>
<td>230.4GFlops</td>
</tr>
<tr>
<td>Core frequency</td>
<td>2.40GHz</td>
</tr>
<tr>
<td>DRAM</td>
<td>4GB</td>
</tr>
<tr>
<td>GPUs</td>
<td>NVidia GeForce GTX 580</td>
</tr>
<tr>
<td># SMs</td>
<td>16</td>
</tr>
<tr>
<td>Caches</td>
<td>L1/Global L2: 16 KB/768 KB</td>
</tr>
<tr>
<td>FP peak performance</td>
<td>1.56 TFlops</td>
</tr>
<tr>
<td>Shader Clock frequency</td>
<td>1544MHz</td>
</tr>
<tr>
<td>O/S</td>
<td>Ubuntu 12.04.1 LTS</td>
</tr>
<tr>
<td>Platform</td>
<td>Intel OpenCL Platform 1.5 for CPU, NVidia OpenCL Platform 4.2 for GPU</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel C/C++ compiler 12.1.3</td>
</tr>
</tbody>
</table>

Table 1: Hardware Configurations of Experiment 1

<table>
<thead>
<tr>
<th>Name</th>
<th>Processor</th>
<th>Cores</th>
<th>HW threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>N8</td>
<td>2.40 GHz Intel Xeon E5620 (hyper-threaded)</td>
<td>2x Quad Core</td>
<td>16</td>
</tr>
<tr>
<td>D6</td>
<td>2.67 GHz Intel Xeon X5650 (hyper-threaded)</td>
<td>2x Six Core</td>
<td>24</td>
</tr>
<tr>
<td>MC</td>
<td>2.10 GHz AMD Opteron 6172 (Magnycours)</td>
<td>4x Twelve Core</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 2: Hardware Configurations of Experiment 2

5.1 Methodology

Two different studies focusing on OpenCL performance on CPUs were looked at. The experimental enviroment for both studies is mentioned in Table 1 [10] and Table 2 [11] respectively.

In the first experiment OpenCL kernel was either executed on Intel OpenCL platform [5], or Nvidia OpenCL platform [6]. An execution framework was implemented which was used to control and vary multiple OpenCL aspects on the applications without any code changes. OpenCL Nvidia Benchmarks [6] were used for evaluation. Some simple applications were also used to further evaluate some constraints. In addition to that, Parboil Benchmarks [23] were also used. Wall clock execution time was used for measuring, kernels were iterated until a long enough execution time was reached for better results.

In the second experiment, benchmarks were selected from the Rodinia benchmark suite [24]. These benchmarks are designed to cover different parallel patterns using the Berkeley Dwarfs as guidelines. Each benchmark has an equivalent implementation in OpenMP, CUDA, and OpenCL. Five of the benchmarks were chosen for performance comparison (K-means, PathFinder, HotSpot, CFD, and BFS) [11]. Iterative tuning was performed on the OpenCL implementations to reach a performance reference set by the OpenMP implementations. If the performance difference is less than 10%, they were considered to have similar performance.
As previously discussed in section 4.1, OpenCL has to create work items and work groups and map them to different compute devices and processing elements before the actual execution can start. This requires additional processing time which can affect performance.

The first experiment calculated the time cost of each API function in the execution of every OpenCL application in NVIDIA OpenCL Benchmarks. The workload size used was the default provided in the benchmarks. Figure 4 shows the ratio of execution time of kernels and auxiliary API functions. It is evident from the figure that almost 90% of the execution time in all of the different benchmarks is taken by auxiliary functions compared to the actual execution of the kernels.

The study further evaluates the main causes of this slowdown and breaks it down into three responsible OpenCL tasks, which are defining kernel execution, cross platform support, and JIT compilation. One of the possible fixes to reduce these overheads is the concept of caching. The main idea available in OpenCL is to extract the compiled binary using clGetProgramInfo API function and store it using FILE I/O functions. This lets the programmer to store the cache on disk and use the binary for execution instead of OpenCL performing JIT compilation everytime an application is run.

The second experiment does not directly look at the performance effects from API overhead. But in evaluating and fine tuning multiple algorithms, the study specifically mentions the HotSpot algorithm, where two kernels were merged to reduce the number of kernel invocations, which resulted in a 14% performance increase[11].

Another important factor to reduce overhead is to define a workload size. As for these benchmarks, the default workload size was used which is relatively small, leading to less execution and more overhead. By choosing a larger workload size, kernel execution would increasing, relatively decreasing the overhead. However, it is up to the programmer to choose a workload size based on the problem and the resources available.

![Figure 4: Execution time distribution of kernel execution and auxiliary API functions. Taken from Lee et al. 2014](image.png)

**5.2 API Overhead**

As previously discussed in section 4.1, OpenCL has to create work items and work groups and map them to different compute devices and processing elements before the actual execution can start. This requires additional processing time which can affect performance.

The first experiment calculated the time cost of each API function in the execution of every OpenCL application in NVIDIA OpenCL Benchmarks. The workload size used was the default provided in the benchmarks. Figure 4 shows the ratio of execution time of kernels and auxiliary API functions. It is evident from the figure that almost 90% of the execution time in all of the different benchmarks is taken by auxiliary functions compared to the actual execution of the kernels.

The study further evaluates the main causes of this slowdown and breaks it down into three responsible OpenCL tasks, which are defining kernel execution, cross platform support, and JIT compilation. One of the possible fixes to reduce these overheads is the concept of caching. The main idea available in OpenCL is to extract the compiled binary using clGetProgramInfo API function and store it using FILE I/O functions. This lets the programmer to store the cache on disk and use the binary for execution instead of OpenCL performing JIT compilation everytime an application is run.

The second experiment does not directly look at the performance effects from API overhead. But in evaluating and fine tuning multiple algorithms, the study specifically mentions the HotSpot algorithm, where two kernels were merged to reduce the number of kernel invocations, which resulted in a 14% performance increase[11].

Another important factor to reduce overhead is to define a workload size. As for these benchmarks, the default workload size was used which is relatively small, leading to less execution and more overhead. By choosing a larger workload size, kernel execution would increasing, relatively decreasing the overhead. However, it is up to the programmer to choose a workload size based on the problem and the resources available.
5.3 Thread Scheduling

As explained in section 4.2, OpenCL performance is also affected by varying number of work items and work groups, this happens due to architectural differences between platforms and how properly these logical things can be mapped to actual hardware.

5.3.1 Number of work items

As previously discussed Section 4.2.1, changing the number of work items can affect performance in CPUs and GPUs due to difference in number of processors between architectures.

To evaluate the effect of work item quantity and size of workload, both studies perform an experiment by allocating more computation to a work item in OpenCL applications. This is done by joining multiple work items by creating a loop inside the kernel. To make sure the amount of total execution remains the same, the number of work items is reduced.

![Figure 5: Performance of Square and Vector addition applications with different workload per workitem. Taken from Lee et al. 2014](image)

Figure 5 shows the evaluation performed by the first study. The figure shows how increasing the workload over a work item affects performance. It focuses on vector addition and square benchmarks on CPUs and GPUs. The results show that there is a notable performance gain for allocating more work per work item on CPUs. However, it is also noted that this reduces performance on GPUs as now all processing units available are not being utilized.

The second study evaluates the pathfinder algorithm in the same way, multiple data elements are merged together into the workload of a single work item, which is called MergeN optimization [11]. Figure 6 shows the performance of MergeN (N=4, 16), the results show a significant performance increase.

One of the reasons for this improvement is due to the reduced number of instruction. As now there are multiple work items inside a loop, and instructions are significantly reduced, leading in less API overhead from execution and JIT compilation.

5.3.2 Number of work items and Instruction Level Parallelism

Section 4.2.2 pointed out performance effects by the lack of instruction level parallelism in OpenCL implementations.
This is explained in the first study, where it is shown how ILP can affect performance, the study implements a set of compute intensive micro benchmarks sharing common characteristics. The only difference between each benchmark is the variation of ILP by varying how many independent instructions are available. This is done by increasing the number of operand variables, so in the baseline implementation, ILP 1, the next instruction depends on previous, but for the proceeding implementation, ILP2, an independent instruction exist between two dependant instructions. Figure 7 shows the increase of performance on CPUs due to the increase of TLP. It is also evident that performance on GPUs almost remains the same.

5.3.3 Workgroup size

Section 4.2.3 describes the differences between work groups on CPUs and GPUs lead to performance effects. The first study evaluated the effect of workgroup size both on CPUs and GPUs. Workgroup size of an OpenCL application can be changed by passing a different argument for local_work_size on kernel invocation. When the argument is NULL, the workgroup size is implicitly defined by the implementation.
Figure 8 shows the performance based on different workgroup sizes. The results show varying effects on different benchmarks. This shows that a workgroup size has to be manually adjusted by the programmer based on the architecture.

Figure 8: Performance of applications on different workgroup size on CPUs and GPUs. Taken from Lee et al. 2014

The second study evaluates workgroup size differences when trying to fine tune the K-Means and CFD algorithm for performance. Figure 9 shows the effect of workgroup size on performance on K-Means algorithm. These changes proved to have no significant effect.

Figure 9: Effect of workgroup size on K-Means algorithm. Taken from Shen et al. 2013

5.4 Memory Allocation and Data Transfer

Section 4.3 discussed effects of memory allocation on performance. The reason for this was the difference in mapping work items to CPUs and GPUs. Work is divided over threads on CPUs which do not have a separate memory as compared to CUDA cores on GPUs.
Most of these effects were only measured in the first study which evaluated different combinations of available memory options. These options included varying level of access for memory, storing data at host or at device, and implicit and explicit data transfer APIs. For accurate results, a blocking call for all memory object commands and kernel execution commands was used to ensure they do not overlap.

![Graph showing data transfer time with different APIs for data transfer.](image)

Figure 10: Data transfer time with different APIs for data transfer. (Left) host to device and (Right) device to host. Taken from Lee et al. 2014

Memory access changes and storing data at different places did not have any significant effects. The main reason for this was the device memory was the same as host memory in case of CPUs. However, using different data transfer APIs had the biggest impact in performance. Figure 10 shows the time taken by explicit data transfer and mapping APIs from host to device and device to host on the Parboil benchmarks.

The second study only noted negative performance impacts on CPUs on the implementations of PathFinder and HotSpot algorithm. The study stated that as both of the algorithms were optimized for GPUs and used local memory, the same architecture was not available in CPUs which resulted in the differences.

### 5.5 Vectorization

As previously stated in section 4.4, vectorization can cause performance impacts due to implementation on different frameworks leading to different vectorization techniques.

The first study evaluated these effects by porting OpenCL kernels to identical computations in OpenMP. In programmer’s expectation, these should perform comparably, however, results show that OpenMP performs poorly. This shows that compiler auto-vectorization does not work the same way on both implementations [10]. Figure 11 shows that OpenMP port of the OpenCL implementations are considerably slower.

In the second study, some of the benchmarks had the option to use auto vectorization available in the Intel compiler. In K-Means algorithm intel vectorization made it perform better. The Intel version was also able to outperform OpenMP due to this reason. However, results were not the same for other benchmarks. In Pathfinder algorithm, autovectorization reduced performance by 26-71% overhead, and similarly in BFS, performance was reduced by 5-12% [11]. This shows that vectorization is dependant on implementations.
5.6 Thread Affinity

As mentioned in section 4.5, OpenCL does not support thread affinity. But to highlight the importance of thread affinity, the first study measure performance benefits by utilizing CPU affinity in OpenMP. A simple application with two different kernels, Vector Addition and Vector Multiplication was used. Threads were binded in two different cases, an aligned case where threads of second kernel are executed on the same cores as the threads of first kernel. And a misaligned case, where threads of second kernel are executed on different kernels. The results showed a 15% difference in performance. It stated that kernels can also be aligned in a similar fashion in OpenCL implementations for performance gains.

These results show that the support of thread affinity can benefit OpenCL in some special cases where CPUs are used.

6 Conclusion

In this paper, two different studies were discussed that evaluated performance of multicore CPUs in OpenCL using multiple benchmarks and compared them with performance in GPUs. In addition to that, OpenMP and OpenCL implementations were compared. Understanding the architectural constraints and their effects can help programmers to avoid performance issues and utilize the framework for better heterogeneous support. By looking at these studies which evaluate multiple architectural aspects such as API overhead, thread scheduling, memory allocation, data Transfer, compiler-supported vectorization, and thread affinity, we can conclude the following points.

1. API overhead in OpenCL can cause significant performance issues on CPUs.
2. Increasing the work performed by a work item in a GPU can increase performance in a CPU.
3. Increasing the workgroup size can help performance in some cases.
4. Local data allocation in GPUs may have poor results in CPUs due to the differences in architecture.
5. Compiler Vectorization in OpenCL is not very mature, and variations in programming model can lead to different results in performance.
6. Adding Thread Affinity can lead to better performance on CPUs in some cases.

With these results in mind, the studies have showed that OpenCL can be a useful tool for parallel applications on CPUs as well as heterogeneous systems. It was also seen that by making some simpler tweaks, OpenCL can perform as good as OpenMP does on CPUs. This proves to be favorable for OpenCL because it supports multiple platforms and can make much more efficient use of resources.

References


