Improving CPU Performance For Heterogeneous Computing Using OpenCL

Shashank B. Thombre, Balagopal Bhallamudi, Abhinav Choudhury

International Institute of Information Technology Bhubaneswar, Bhubaneswar.

Abstract

With the rapid development of smart phones, tablets, and pads, there has been widespread adoption of Graphic Processing Units (GPUs). The hand-held market is now seeing an ever-increasing rate of development of computationally intensive applications, which require significant amounts of processing resources. To meet this challenge, GPUs can be used for general-purpose processing. We are moving towards a future where devices will be more connected and integrated. This will allow applications to run on handheld devices, while offloading computationally intensive tasks to other compute units available. There is a growing need for a general programming framework which can utilize heterogeneous processing units such as GPUs and DSPs. OpenCL, a widely used programming framework has been a step towards integrating these different processing units on desktop platforms.

A thorough study of the factors that impact the performance of OpenCL on CPUs. Specifically, starting from the two main architectural mismatches between many-core CPUs and the OpenCL platform—parallelism granularity and the memory model we identify such performance "traps" that lead to performance degradation in OpenCL for CPUs. Using multiple code examples, we quantify the impact of these traps, showing how avoiding them can give up to 3 times better performance.

1.Introduction

In recent years, modern processor architectures have shifted their focus on increasing the amount of parallelism rather than just increasing the processor clock speed. Currently every computer system we can call it as a heterogeneous system as it contains central processing unit (CPUs) with multiple cores, as well as graphical processing units (GPUs)[1]. Both CPUs and GPUs can be used for the parallel computing. Some time ago, using a GPU for general purpose computation was a major programming challenge. But thanks to execution models like CUDA (Compute Unified Device Architecture)[2], it has become simpler. Though games still represent a big market for graphical market manufacturers, GPUs can still be designed keeping general purpose computations in mind. GPUs are generally known for faster floating point operation executions and hence attracting research communities.

Applications with great amount of data parallelism perform considerably better on GPU as compared to the CPU. To take full advantage of heterogeneous parallel processing platforms Open Compute Language(OpenCL)[3] was introduced. By using the OpenCL an application can be partitioned such that part of code runs on the host processor i.e. CPU and part of the code runs on GPU. In OpenCL we are able to manage, OpenCL initialization, data transfers and kernel executions by set of language extensions and runtime Application Program Interfaces (APIs).

OpenCL has a lot of similarities with CUDA (Compute Unified Device Architecture) and unlike CUDA it is platform independent and hence it keeps gaining popularity for GPU computing. To take advantage of all parallel computing resources the applications need not be rewritten. Typically we write kernels using OpenCL. Kernels are nothing but the computationally intensive functions of the application. The applications generally perform several tasks while executing. The application needs to query platforms and devices available on system, create contexts, create command queues, and enqueue kernels to these command queues. Thus as compare to the normal execution the parallel execution shows almost 10x performance improvement for the simple applications used in our experiment.

According to its specifications [3], OpenCL shares the core parallelism approach with CUDA which is designed for NVIDIA GPUs. Thus there are visible mismatches between the OpenCL platform and the CPUs. Furthermore these mismatches may affect the performance of the CPUs. This slight performance degradation may be taken as parallelization overhead and further it can easily be reduced by slight code transformations[4][5].

2. OpenCL Programming Framework

The design and implementation of OpenCL framework is discussed in this section. The point of discussion revolves around an architecture consisting of multiple core CPUs, GPUs, accelerators and DSPs generally termed as heterogeneous accelerator embedded architectures. OpenCL provides general purpose GPU programming model provides way to address complex memory hierarchies and vector operations.

OpenCL is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. OpenCL gives software developers portable and efficient access to utilize the power of these heterogeneous processing platforms. OpenCL includes a language (based on C99) for writing *kernels* (functions that execute on OpenCL devices), plus application programming interfaces (APIs) that are used to define and then control the platforms. OpenCL is maintained by the non- profit technology consortium Khronos Group. It has been adopted by Intel, Advanced Micro Devices (AMD), NVIDIA, Altera, Samsung, Vivante, and ARM holdings.
The operational model of OpenCL mainly consists of 4 sub models platform model, execution model, programming model and memory model[1][3].

2.1 Platform Model

Figure 2.1 shows the OpenCL platform model in details. The platform model consists of one host and one or more OpenCL devices. OpenCL devices can be CPU cores, GPUs, accelerators or DSPs. OpenCL devices are divided into one or more compute units (CUs), which are further divided into processing elements(PEs). Computation on a device occurs with a processing element.

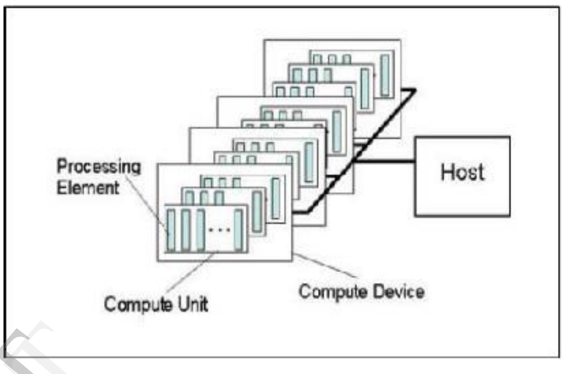


Figure 2.1: OpenCL Platform model: Host with Compute devices, Compute Units and Processing elements [1].

The OpenCL application submits commands from the host to execute computations on the processing elements within a device. The processing elements within a compute unit execute a single stream of instructions as SIMD units.

2.2 Execution Model

OpenCL program executes on GPU in the form of kernel(s) that execute on one or
more OpenCL devices and a host program that executes on the CPU. The host program defines the context for the kernels and manages their execution. Kernel defines an index space depending on the application. An instance of the kernel executes for each point in this index space [9]. This kernel instance is called a work-item. The work-item is identified as a point in the index space, by a global ID for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data can vary per work-item. Work-items are organized into work-groups. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work items. A unique local ID is assigned to work-items, within a work-group so that a single

work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit. The index space is called an NDRange which is an N-dimensional index space, where N is one, two or three.

2.3 Programming Model

The OpenCL execution model supports data parallel and task parallel programming model[9].

In simpler words data parallel model can be explained as the sequence of operations applied on multiple elements of memory object.
Independent data elements can easily be transferred to the respective processing elements and execution can be done in parallel.

In task parallel programming model a single instance of a kernel executed independent of any index space.

2.4 Memory Model

Work-item(s) executing a kernel have access to four distinct memory regions. This is distinctly shown in Figure 2.2:

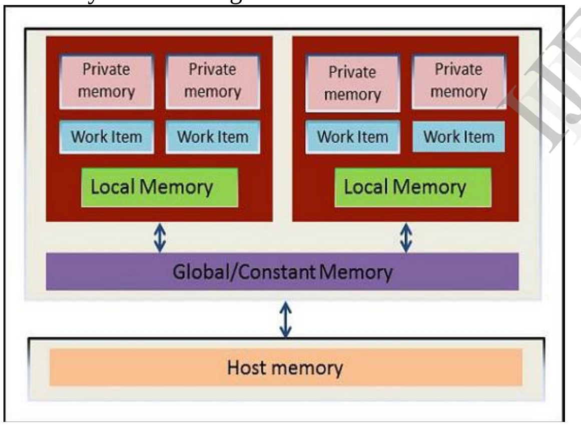


Figure 2.2: OpenCL memory model[2]

Global Memory : This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object.
Constant Memory: A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.
Local Memory: A memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group.
Private Memory: A region of memory private to a work-item. Variables defined in one work-items private memory are not visible to another work-item.

3. Experimental setup

All the experiments are carried out on real hardware. A machine with an Intel core i5-2450m processor running at 2.50 GHz and 2 GB RAM. The Microsoft VC++ IDE 2010 along with windows7 operating system are used for all the programming.

We have run each program for 5 times and execution time mentioned below is the average execution time. In each execution the time of execution varies slightly and cannot be constant for even same data. This happens due to local system environment. Vector addition, matrix multiplication, vector dot product programs are used for the performance analysis.

Though parallel execution of programs shows us significant performance improvement, the memory model mismatch between OpenCL(same as GPUs) and CPU further affects the performance of CPUs. On GPUs the host and devices are physically independent of each other i.e. having separate memory spaces. Thus to complete a parallel task, explicit data transfers between host and device are required which may require additional time.

While mapping OpenCL on CPUs the host and device are the same CPU sharing the same memory space. Thus the explicit data transfer from host to device(H2D) and device to host(D2H) becomes unnecessary. We are eliminating this data transfer overhead by using zero copy technique. In this technique instead of initializing data on host and copying on device buffers we are creating buffers on device and initializing data directly onto buffers. The host accesses the same data (buffers) by using pointer to these buffers. Thus, instead of making multiple copies of the data, host accesses data by using a pointer. (zero copy)

3.1 Vector addition

While running vector addition program in non parallel environment it is found that on an average it takes 1 milisecond time for execution. Sufficiently large size data is chosen for clear time discrimination. Two vectors used are of size 65536.

Figure 3.1 shows the performance of vector addition running in parallel and the time spent in data transfer. It has been observed that the execution time depends upon the local work item size that is specified in the program. The performance for vector addition is 3x faster(including overhead).

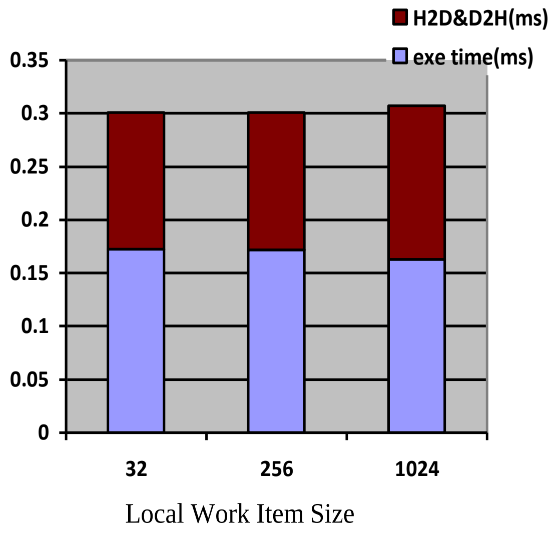


Figure3.1: Kernel Execution time and data transfer time for vector addition with varying the local work item size.

The data transfer time between host and device is considerably large hence it cannot be simply ignored. After applying the zero copy technique we reduced the data transfer overhead and hence gained 5x performance.

Figure 3.2 shows results after applying zero copy technique.

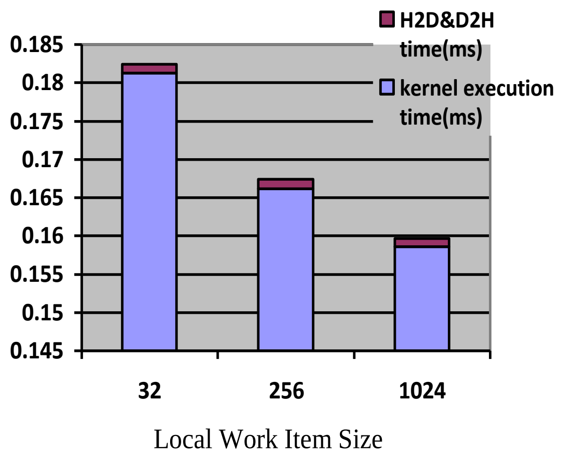


Figure3.2: Kernel Execution time and data transfer time for vector addition with varying the local work item size using zero copy technique.

3.2 Matrix Multiplication

We have chosen two matrices of size 128 x 128 for calculating results in parallel. While running in non parallel environment the average execution time for this calculation is 29 ms but while in parallel it shows significant speedup.

Figure 3.3 shows the actual execution results for the matrix multiplication in parallel.

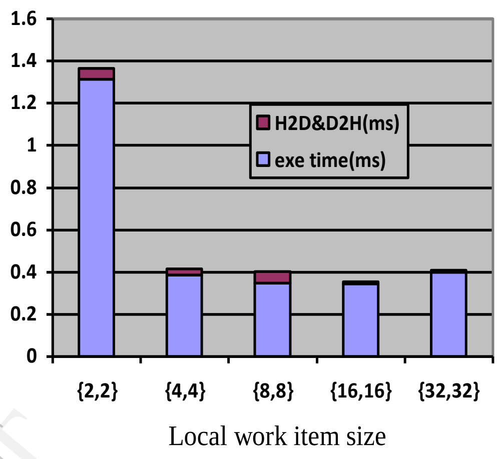


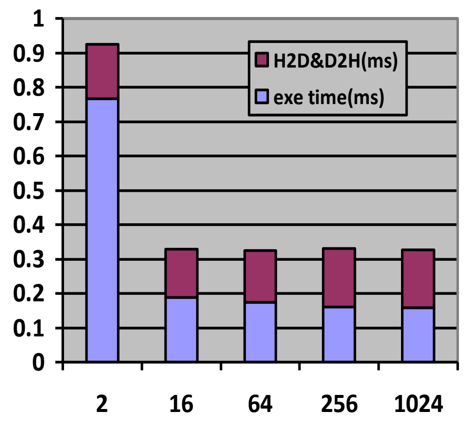
Figure 3.3 :Execution time for matrix multiplication with varying the local work item size and dat transfer overhead

From execution time we can conclude that the matrix multiplication runs at least 27 times faster in parallel using the OpenCL. The data transfer time is small compare to the actual kernel execution time. Hence does not affects overall execution time.

3.3 Dot Product

In dot product we are using two vectors of size 1024. In non parallel environment the average execution time is 1 ms but when executed in parallel using OpenCL it gives significant speedup.

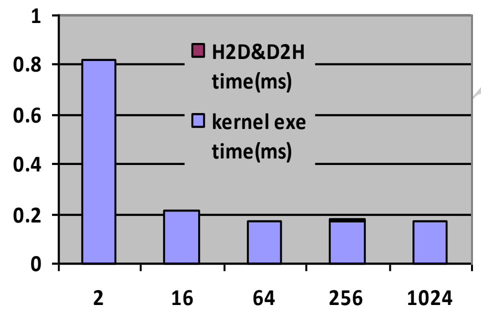
Figure 3.4 shows the actual results obtained while executing the dot product program in parallel with variations in execution time according to local work item size and data transfer time.



Local work item size

Figure3.4: Execution time and data transfer time for dot product with varying the local work item size.

The data transfer time in case of dot product is considerably large and it adds to total execution time. Figure3.5 shows the results after applying zero copy technique on dot product.



Local work item size

Figure3.4: Execution time and data transfer time for dot product with varying the local work item size using zero copy technique.

From execution time we can conclude that the dot product runs at least 5 times faster in parallel using the OpenCL.

4. Discussion

In this experimentation, it is found that better computational performance and better device utilization can be achieved when the parallelism is implemented on a particular systems. The host sends the kernels to execute on the available devices. In OpenCL every host program consists of 13 sub tasks. These are executed by the host to ensure the parallelism. The sub tasks are:

1. Discover and initialize platform
2. Discover and initialize devices
3. Create a context
4. Create a command queue
5. Create device buffer
6. Write host data to device buffer
7. Create and compile the program
8. Create the kernel
9. Set the kernel arguments
10. Configure the work item structure
11. Enqueue the kernel for execution
12. Read the output buffer back to the host
13. Release the OpenCL resources

The main purpose of running applications in parallel is to fully utilize available hardware and reduce the execution time as well.

On 15 November 2011, the Khronos Group announced the OpenCL 1.2 specification, which added significant functionality over the previous versions in terms of performance and features for parallel programming. Most notable features include device partitioning, separate compilation and linking objects, enhanced image support, built-in kernels, DirectX functionality.

5. Conclusion and future work

The current developments in semiconductor technology enable us to use GPUs at very low prices. To explore these GPUs for computational purpose along with the other hardware OpenCL provides the best framework.

In this experimentation, we have studied the performance of OpenCL on a heterogeneous platform. OpenCL gives significant amount of performance gain over non parallel implementation.

It has been observed that while running two or more OpenCL applications on a heterogeneous platform, each application creates its own command queue and enqueues the OpenCL kernels to these command queues independent of other applications. The OpenCL runtime selects the kernels from these commands in round robin manner for execution on device. Thus execution time shown also includes the waiting time in the command queue. By using proper scheduling algorithms further performance improvement can be achieved[7].

Also according to the specifications [3], OpenCL shares the core parallelism approach with CUDA, which is designed for the NVIDIA GPUs. Thus when computations are done on CPUs there are visible mismatches between the OpenCL platform and CPU architecture. This affects performance significantly for the CPUs. The OpenCL code is thus further needs to be optimized and tuned properly. To optimize the OpenCL code is fully programmers responsibility according to the performance of system and application requirements[5].

6. References

[1] K. Group. *"OpenCL- The OpenCL standardfor parallel programming of Heterogenous Systems"*, 2012. http://www.khronos.org/opencl

[2] Group Khronos OpenCL parallel computing for heterogeneous devices, khronos group , page1-50,2009.

[3] NVIDIA, "*CUDA c programming guide*." http://developer.download.nvidia.com/compute/DevZone/docs/html/c/doc/CUDA_C_programming_guide.pdf,2012

[4] K. Group. " The OpenCL Specification V1.2" 2011. http://www.khoronos.org/registry/cl/specs/opencl-1.2.pdf.

[5] R. Karrenberg and S. Hack, *"Improving Performance of OpenCL on CPUs"*, in compiler construction 2012,pp 1-20,Springer,2012

[6] Z.Wang and M. O'Boyle, *"Mapping parallelism to multi-cores: A machine learning based approach,"* in *Proc. 2009 ACM PPoPP*, Raleigh,NC, 2009, pp. 75–84.

[7]. Jie Shen, Jian Fang, Henk Sips, Ana Lucia, "*Performance Traps in OpenCL for CPUs"*, 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing,pp 38-45,2013

[8] Arshdeep Bahga and Vijay K. Madisetti, "A Dynamic Resource Management and Scheduling Environment for Embedded Multimedia and Communications Platforms." in IEEE Embedded Systems Letters, VOL. 3, NO. 1, pp 24-27,MARCH 2011
[9] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, Dana Schaa,*"Heterogeneous computing using OpenCL"* Elsevier Inc, 225 Wyman Street, Waltham, MA 02451, USA, November 27, 2012