iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 10 **Issue:** VII **Month of publication:** July 2022

DOI: https://doi.org/10.22214/ijraset.2022.45835

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 10 Issue VII July 2022- Available at www.ijraset.com

Programming in OpenCL and its advantages in a GPU Framework

K S Harshavardhan
ISE, RVCE, Mysore Rd, RV Vidyaniketan, Post, Bengaluru, Karnataka, India 560059

Abstract: ***OpenCL is a framework used for building applications that mostly run on heterogenous platforms containing CPUs, GPUs, and DSPs. OpenCL provides an interface for parallel programming that can be used to take advantage of the GPUs high parallel computing power. Programmers who need complete control over the parallelization process and who are required to write portable heterogeneous code mostly use OpenCL. OpenCL views a processing unit as a collection of compute units which in turn are made up work items. According to OpenCL's workflow and memory hierarchy, each work item is a thread as far in terms of control and memory model. A collection of work items is called a work group which is mapped to a compute unit. The language used to write "compute kernels" is called kernel language. OpenCL uses C/C++ to carry over the kernel computations done on the device. The host code specifies the kernel specifications that is needed for the computation of the device which includes creating buffers, calling kernels, mapping the memory back to CPU from device, etc. OpenCL also has specific optimization techniques that helps improve parallelization while computing on a GPU which results in better performance numbers.***

Keywords: Work item, Kernel, Optimization, OpenCL, GPGPU, GPU, Parallel programming

I. INTRODUCTION

In general purpose engineering and scientific computing, GPU computing refers to using GPU(s) to perform calculations that can usually be divided into smaller individual calculations. Due to its parallel processing architecture, GPU is able to perform multiple calculations over a stream of data simultaneously. A CPU usually consists of 4 to 8 cores while a GPU consists of hundreds of smaller cores. This parallel architecture can be taken advantage of using OpenCL kernels as it allows a normal C/C++ program that would be executed sequentially to be processed parallelly. An OpenCL application can be divided into two parts: host code and device code. The host code is usually written in a general language such as C or C++ and is compiled using a conventional compiler and the device code is written in OpenCL C/C++ and it is compiled by the OpenCL driver. The host can be described as the outer control logic that helps configure the GPU framework. It is responsible for the initialization of kernels, creating buffers for the kernel, setting up kernel arguments, mapping the memory back to CPU and the destruction of the kernel objects. In many cases the host is the main CPU that is used for the configuration of the GPU and the host code is the code written for the same. The device is the GPU that executes the kernel, and the device code is the OpenCL code that helps reducing the execution time of the code. This is done by parallelizing for loops, while loops, byte loading, etc. Drastic reduction in execution times can be seen in programs where multiple bytes are being operated on in a similar fashion.

II. LITERATURE REVIEW

Multiple studies have been conducted in the past to explore the parallel computing nature of GPUs and the advantages of OpenCL kernels. This paper [1] talks about the advancement in recent GPUs and how their parallel computing power helps in solving computationally challenging problems like analyzing magnetic fields in a very small amount of time. This paper presents a multidimensional Fast Fourier Transform-based parallel implementation of a magnetostatic field computation on GPUs. A specialized 3D Fast Fourier Transform library has been developed for magnetostatic field calculation on GPUs which made it possible to take full advantage of the symmetry in field computations and other optimizations in the GPU architecture. The results showed up to 95x and 8.7x faster for single-precision floating-point precision and 66x and 4.6x faster for double-precision floating-point precision, respectively, for an equivalent serial implementation or OOMMF. when compared with the widely used CPU-based parallel OOMMF programs and equivalent serial implementations on the CPU.

This paper [2] compares the performance numbers between the OpenCL and OpenMP programs for multi core CPUs as these two are the most commonly used parallel programming frameworks despite being different in use cases as OpenCL is a more considered to be more suitable for cases with hundreds of short kernels.

©IJRASET: All Rights are Reserved

whereas OpenMP is used for complex longer running code that needs parallelization. It also explores the possibility of improving the performance numbers by setting the CPU affinity.

It is found that improper use of multi-core CPUs, incompatible OpenCL compilers and unique OpenCL parallelism are the main reasons for poor OpenCL performance. After configuring the versions of OpenCL to make it more suitable for the CPU, it indicated that OpenCL achieved similar performance in 80% of the times. Therefore, OpenCL is considered to be a good alternative to multi-core CPU programming.

In this paper [4], authors have strategically compared CUDA and OpenCL parallel programming languages. The paper uses real time performance for different frameworks. There is a great potential offered by GPUs for the performance and efficiency of critical large computational science applications but taking advantage of this can be difficult due to the need to adapt to the special and rapidly evolving computing environments. This article introduces a simple technique, GPU Runtime Code Generation (RTCG), and two open-source toolkits that support this technique, PyCUDA and PyOpenCL. With the help of these toolkits, the article proposes the combination of a high-level scripting language with the parallel computing performance of a GPU as a compelling two-tiered computing platform, potentially offering better performance and productivity over conventional single-tier, static systems.

In this paper [7], a performance comparison between GPU and CPU is done on a discrete heterogenous architecture. This study briefly describes an evolutionary journey of GPUs and the comparison with CPU is done considering latency and throughput as parameters. For a given task, based on the execution time of a GPU and CPU, written with CUDA language, the two parameters are measured with increasing size of workload. It is found that GPU is 51% faster than the multithreaded CPU when GPU achieves 100% occupancy when the task size is increased. Throughput of GPU is found to be 2.1 times higher than that of CPU for large task size.

This paper [6] focuses on situations where GPU is more suitable, and it compares parallel computing with normal CPU performance. A GPU platform is commonly assumed to implement parallelization, and this platform has been shown to be better than the traditional CPU platform in numerous previous studies, but it is still unclear if that stays true even in the case of parallelizing multiple independent runs, since most of the previous studies have focused on parallelization approaches in which the parallel runs are dependent on each other. This paper aims to explore the performance of the GPU in comparison with the CPU with respect to multiple independent runs in order to determine the most efficient platform.

III. METHODOLOGY

A. Architecture

The class diagram and flow chart shown below gives us an abstract overview of the entire OpenCL Infrastructure. Various components involved and their interaction to allocate memory, map memory, create objects, pass parameters, create kernels, and delete kernel objects are shown in the given figure below. The flowchart briefly describes the flow of an OpenCL application from the host to the device. The host code is responsible for the configuration of the GPU and along with the function calls specific to the kernel. On a high level, the device runs the kernel by launching multiple work items and parallelizing repetitive parts of a code like a for loop.

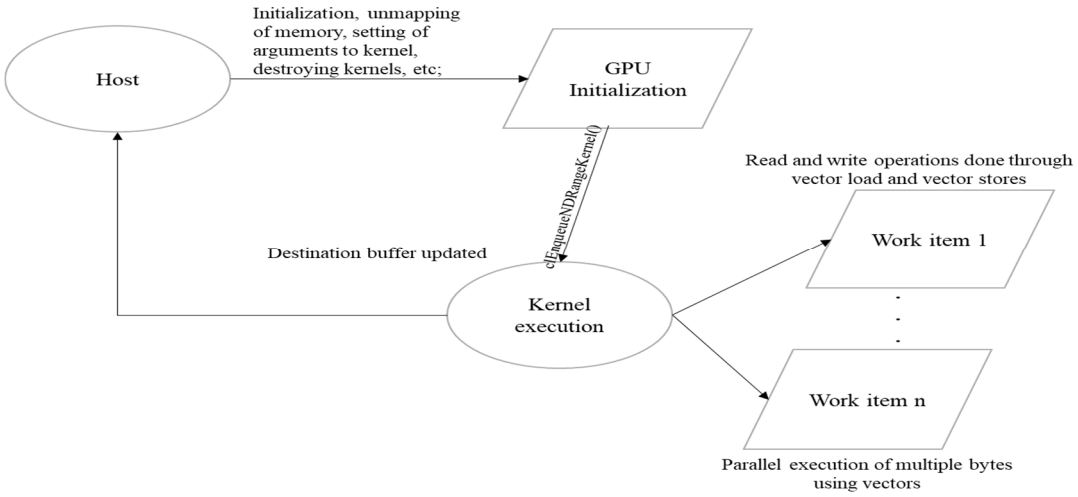


Fig. 1. Architecture diagram of the OpenCL workflow

B. Methodology

The below summary describes the steps involved in the working and implementation of an OpenCL application.

1) *Analyzing the code:* Before the entire process of writing a program in OpenCL, the code segment that is being converted into kernels needs to be analyzed. Areas with for loops and while loops need to be identified where there is scope for parallel processing of a byte stream. Once this is done, the parameters need to be determined after which the kernel can be designed.
2) *Designing the kernel:* As the code segment is now divided into kernels, these segments need to be converted into OpenCL kernels. Global dimensions need to be determined based on the operation after which the code can be written for a single instance that would be applicable for all the bytes. The number of parallel instances of the kernel is determined based on the value of the global dimensions. Any operations that are essential for the next iteration/instance is avoided in the kernel and only operations which are independent of the prior iterations are taken into consideration while writing a kernel.
3) *Designing the host code:* Based on the function, a code segment may be broken into a single or multiple kernels. Kernels and the memory buffers must be defined initially after which the buffers are unmapped from the CPU. Global dimensions are set for each kernel and the parameters are set after which the kernel is called. Once the kernel is done with the execution, the memory is mapped back to the CPU and the kernel object is deleted.
4) *Compilation:* The kernel is initially compiled using OpenCL drivers after which a buffer or a header file is returned automatically. This header file is used to compile host OpenCL applications using the C API. The header file however is not necessary to just compile the kernel and it is just the result after compilation.
5) *Optimization:* The kernels can be kept similar to the original code segment, but the performance can be further enhanced by using OpenCL specific functions like vloadn() and vstoren() that are used to load and store multiple bytes in a single operation. Vectors can also be used in mathematical operations like multiplication and shift operations in cases where the same operation needs to be run on a stream of data.

IV. OPTIMIZATION

A. Kernel performance optimization

Some applications may be complex and may contain several steps making it difficult to decide to come up with the number of kernels required to design the OpenCL application. This question may not have an easy or a definitive answer, but these are some of the practices that need to be considered:

1) Good balance between compute and memory
2) Avoiding register spilling
3) Enough number of waves to hide latency

The below mentioned points can be followed to achieve these:

a) A big kernel can be split into multiple small kernels provided this results in better data parallelization.
b) Kernel fusion can be followed i.e. multiple kernels can be fused into one kernel provided memory traffic can be decreased and parallelization can be maintained.

B. Memory performance optimization

Many applications are memory bound rather than compute bound making it crucial to master memory optimization. OpenCL defines 4 types of memory namely, global, local, constant, and private memory. Each of these types are different hence it becomes important to use the correct type in each case. The following table describes the differences among the types:

Memory	Definition	Relative latency	Location
Local	Shared by all work items in a work group	Medium	On-chip, inside SP
Constant	Constant for all work items in a work group	Low for on-chip allocation, and high otherwise	On-chip if it can fit in. Otherwise in system RAM
Private	Private to a work item	Based on where the memory is allocated by the compiler	In SP as register or local memory or in system RAM (compiler determined)
Global	Accessible by all work items in all work groups	High	System RAM

Table 1. Differences between the different types of memory in OpenCL

Apart from using the correct type of memory, it is also essential to use coalesced load and store wherever applicable. Coalesced store/load refers to the combining of store/load requests from numerous neighboring work items. Coalesced load and store work in a similar manner to read and write, respectively. Vectorized store /load refers to multiple data store/load in a vectorized way for single work items. Vectors are used for loading multiple bytes from a buffer and storing multiple bytes into the output buffer in a single operation.

Another important factor in memory optimization in choosing the correct datatype. Using a larger datatype like long while using vector load and store can sometimes be counterproductive as it may increase the execution time, hence it is crucial to choose the appropriate datatype along with the vectorization techniques. In addition to that, using floating half data type instead of floating data type can improve performance as GPUs have dedicated hardware to accelerate half data type calculation.

V. RESULTS

```
/* kernel.cl
 * Matrix multiplication: C = A * B.
 * Device code.
 */

// OpenCL Kernel
__kernel void
matrixMul(__global float* C,
          __global float* A,
          __global float* B,
          int wA, int wB)
{

   int tx = get_global_id(0);
   int ty = get_global_id(1);

   // value stores the element that is
   // computed by the thread
   float value = 0;
   for (int k = 0; k < wA; ++k)
   {
      float elementA = A[ty * wA + k];
      float elementB = B[k * wB + tx];
      value += elementA * elementB;
   }

   // Write the matrix to device memory each
   // thread writes one element
   C[ty * wA + tx] = value;
}
```

Fig. 2. OpenCL GPU code for matrix multiplication

The above image represents the implementation of a simple program: matrix multiplication. The kernel uses the global memory type for the buffers that has been unmapped from CPU and the global dimension of this kernel is 2 as it involves 2 for loops and the values of these dimensions depends on the height and width of the matrices.

Benchmark Parameters	Optimized OpenCL time, sec	Original OpenMP time, sec	Optimized OpenMP time, sec
Grid size: 2048 X 2048 # of iterations: 4800	22.2	65.9	8.11
predefined well-conditioned matrix, size: 8000 X 8000	3.32	368	3.24
sequence length: 16384 penalty value: 10	5.52	11.3	3.63
pre-generated file with 193K elements	9.36	28.3	9.03
pre-generated file with 16M nodes and 100M edges	1.59	1.03	0.71

Table 2. OpenCL performance numbers compared with OpenMP numbers

The above table represents the results of the same algorithm run on OpenCL and OpenMP. Firstly, the OpenMP code was extended with target pragmas to allow the code to utilize the Intel XP coprocessor. The OpenCL code ran on the Intel Xeon Phi coprocessor out of the box. The OpenMP code was then modified and used in the Intel VTun Amplifier XE which helped in identifying the regions that needed to be optimized.

VI. FUTURE DISCUSSIONS

OpenCL being a relatively new framework that has not been tested to a great extent in the real world, the exact optimal optimization techniques for each API/situation is not known and hence new optimization techniques can be explored via trial and error by checking different permutations and combinations which can lead to better performance numbers. The device being used too plays a role in this and hence it is difficult to completely generalize a particular optimization technique. Also, OpenCL has the added benefit that it can run on CPUs and is already supported by Intel and AMD making it compatible with CPU cores and hence there is no need to change the algorithmic framework. Lastly, OpenCL has several vendors whose products are leading in the chip industry making it even more promising.

VII. CONCLUSION

With the rise of more powerful GPUs, the latency factor has drastically reduced over the years. GPU was being used predominantly for image rendering specially in the gaming industry, but it is also being used for general computing now and OpenCL is playing a huge role in this. OpenCL can be used to build a GPU framework for a CPU based architecture to improve performance specially in image and matrix related functions where similar operations are performed on a stream of data. Due to the reasons mentioned above, OpenCL is among the top choices for many vendors for General Purpose GPU computing (GPGPU).

VIII. ACKNOWLEDGMENT

Prof. Sharadadevi Kanganurmath (Assistant Professor) was instrumental in the effective completion of this study, and the author would like to express their gratitude to them.

REFERENCES

[1] Khan, Fiaz Gul, et al. “An Optimized Magnetostatic Field Solver on GPU Using Open Computing Language.” Concurrency and Computation: Practice and Experience, vol. 29, no. 5, 2016, https://doi.org/10.1002/cpe.3981. 2.

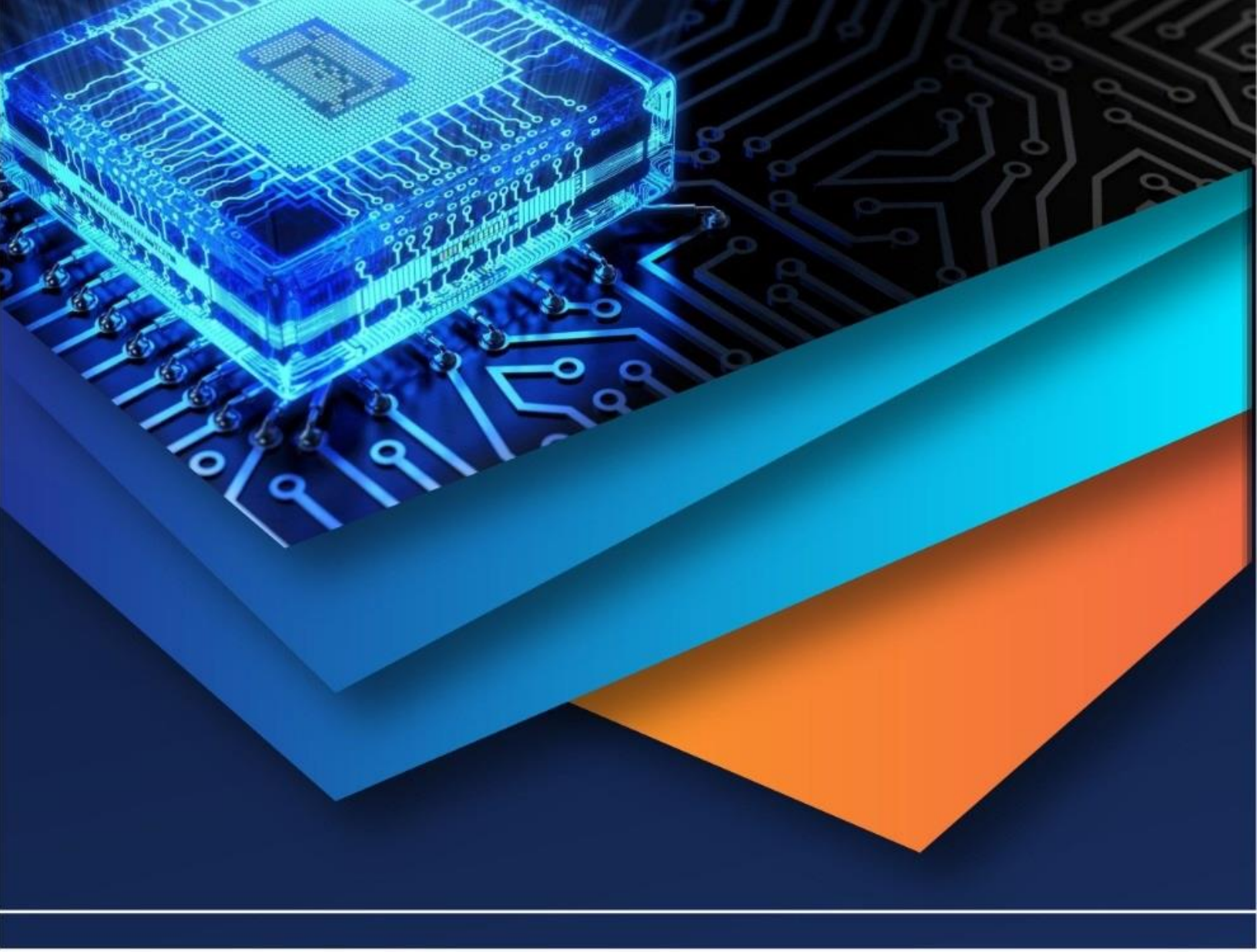
[2] Shen, Jie, et al. “Performance Gaps between Openmp and Opencl for Multi-Core Cpus.” 2012 41st International Conference on Parallel Processing Workshops, 2012, https://doi.org/10.1109/icppw.2012.18.

[3] Klöckner, Andreas, et al. “Pycuda and Pyopencl: A Scripting-Based Approach to GPU Run-Time Code Generation.” Parallel Computing, vol. 38, no. 3, 2012, pp. 157–174., https://doi.org/10.1016/j.parco.2011.09.001.Aggarwal, V., Jain, A., Khatter, H., & Gupta, K. (2019). Evolution of chatbots for smart assistance. VOLUME-8 ISSUE-10, AUGUST 2019, REGULAR ISSUE, 8(10), 77–83. https://doi.org/10.35940/ijitee.i8655.0881019

[4] Thomas, Winnie, and Rohin D. Daruwala. “Performance Comparison of CPU and GPU on a Discrete Heterogeneous Architecture.” 2014 International Conference on Circuits, Systems, Communication and Information Technology Applications (CSCITA), 2014, https://doi.org/10.1109/cscita.2014.6839271.

[5] 6. Syberfeldt, Anna, and Tom Ekblom. “A Comparative Evaluation of the GPU vs the CPU for Parallelization of Evolutionary Algorithms through Multiple Independent Runs.” International Journal of Computer Science and Information Technology, vol. 9, no. 3, 2017, pp. 01–14., https://doi.org/10.5121/ijcsit.2017.9301.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)