JCBI
Journal of Computing & Biomedical Informatics
ISSN: 2710 - 1606
Volume 07 Issue 02
2024
Research Article
https://doi.org/10.56979/702/2024

CPU vs. GPU: Performance comparison of OpenCL Applications on a Heterogeneous Architecture

Muhammad Nadeem Nadir[1], Muhammad Siraj Rathore[2*], Asad Hayat[3], and Junaid Abdullah Mansoor[4]

[1]The Univeristy of Lahore, Lahore, 54590, Pakistan.
[2]Capital Univeristy of Sciecne and Technology, Islamabad,46000,Pakistan.
[3]Taiyuan University of Science and Technology,Taiyuan, 030000, China.
[4]University of Management and Technology, Lahore,54770, Pakistan.
[*]Corresponding Author: Muhamamd Siraj Rathor. Email: muhammad.siraj@cust.edu.pk

Received: May 21, 2024 **Accepted:** August 12, 2024 **Published:** September 01, 2024

Abstract: The objective of researchers and developers has always been to attain superior performance for their computing applications. In this regard, the use of Graphic Processing Unit (GPU) is very common and initially it is used to accelerate the performance of graphic applications. The success of GPU has attracted researchers and they have shown keen interest to use GPU acceleration for regular applications. However, there have been many studies in recent past claiming, even though the application is well suited for parallelism it is not guaranteed to run faster on the GPU. In this this paper we compare performance of commonly used OpenCL applications both on CPU and GPU platforms. We measure the execution time of each application on both platforms and investigate why an application performed better on a particular platform. In this regard, we analyze the source code of each application and identify program features which contributes towards the better performance on a particular platform. The study has identified that loop unrolling and data dimensionality are crucial program features that can be leveraged to utilize the parallel processing capabilities of a GPU platform. We find that when maximum loop unrolling is used with two-dimensional input data, the 2D Convolution application executes around 20 times faster on GPU. Similarly, when the level of loop unrolling reduces, the performance gain also decreases on GPU. Ultimately, in the absence of loop unrolling along single-dimensional input data, CPU performs better. In this case, the ATAX application executes around 9x faster on CPU as compared to GPU.

Keywords: Heterogeneous Computing; CPU/GPU; Loop Unrolling; GP-GPU; OpenCL.

1. Introduction

Engineers have often sought to build efficient computer systems to solve complicated computing problems as quickly as possible. In parallel computing world, apart from multi-core computer systems, another promising technology to improve the application performance is GPU. A GPU contains many small computing units where processing tasks can be distributed in parallel to improve the application performance [1]. Initially, GPUs are often used to create computer graphics, to do jobs that were previously handled by CPUs [2]. After the success of GPU for graphics applications, the attention has been diverted to use GPU for general purpose computing i.e., General Purpose Graphical Processing Unit (GP-GPU). GP-GPUs have a significant impact on ground-breaking scientific research, and many high performance computing (HPC) servers make use of several GP-GPUs to achieve supercomputing levels [2][3].

The success of GPU has given rise to a novel computer architecture paradigm known as discrete heterogeneous architecture [4][5]. In this architecture, a combination of distinct processing units, such as CPUs, GPUs, FPGAs, and specialized accelerators, are integrated into a single system [6]–[8]. Each processing unit is optimized for specific types of tasks, leveraging their unique strengths. CPUs excel at sequential processing and managing complex tasks [9], GPUs are adept at parallel processing and accelerating data-intensive operations [10], FPGAs provide flexibility through customizable logic for specific workloads, and specialized accelerators target particular applications like AI and machine learning[11].

Although GP-GPUs excel in parallel computing due to their vast number of cores, which run at lower rates than CPUs, but are more appropriate for parallel work load distribution. However, on the other hand, it has been observed in the literature [12][13][14], that although an application may be well suited for parallel processing, using a GPU does not always guarantee faster processing speed. In this situation, for a given parallel application, one may wonder which platform should be used for the faster execution of that application. An application may execute faster on either a multi-core CPU or a GPU platform. In other words, which architecture (CPU or GPU) is better for a given general-purpose application? To address this question, it is necessary to conduct an application's code analysis and identify the factors that can impact the performance of the application on a particular platform.

In our work we have selected Open Computing Language (OpenCL) [15] based applications for performance study both on CPU and GPU architectures. The OpenCL is a platform independent programming framework which is widely used by the programming community [16] and the part of Polybench suits [17]. The study involves a performance comparison of 11 selected OpenCL applications on both CPU and GPU architectures. To make a fair comparison, we have selected applications from different domains such as image processing and data mining. Furthermore, some selected applications are processing intensive while others are memory intensive. Then we execute each application both on CPU and GPU platforms and compare execution times on both platforms. In this way we figure out which platform is better suited for that application (with lower execution time indeed).

Our performance comparison reveals that there is not a single winner (CPU or GPU) for these applications since some applications executes faster on CPU while others run faster on GPU. Our results indicate that it is not straight forward to decide the suitable parallel platform for a specific application. It is well known that the real benefit of either a multi-core CPU or a GPU platform can only be exploited when we enable parallel processing through programming [1]. This led us to a detailed examination of selected OpenCL applications code. We analyze how much a particular application exploits GPU acceleration and is there any possibility that the result is dependent on the developers' programming skills rather than the hardware. In this way, we learn not only the better platform for an application but we also learn how an application should be properly coded in order to exploit the parallelism of GPU platform. Our code analysis reveals the software features for these selected applications that affect the execution time on CPU and GPU platforms. In this regard, we find that loop unrolling and data dimensionality are significant factors that affect program's performance when running on either platform.

In summary, the aim of this study is addressing the following research questions:

- Which selected OpenCL applications execute faster on GPU and which applications execute faster on CPU?
- Which software characteristics have a significant impact on an application's execution time when running on CPU and GPU platforms?

To be more specific, the performance of the following six selected applications are found lower on GPU: ATAX, BICG, MVT, Gesummy, Correlation and Covariance. Our code analysis reveals that the lower performance is the result of poor GPU hardware exploitation. For instance, there is only single dimensional input data for these six applications along very limited (or even 0 in case of ATAX) loop unrolling capabilities which make it hard to distribute the work load in parallel among processing units of a GPU. We believe that the performance of these application can be improved on GPU by addressing these issues. According to our knowledge, there is no existing work that compare the performance of OpenCL applications on CPU and GPU platforms and thus we consider this as our main contribution. We justify why a specific platform is suitable for a specific application which is not evident from the current litrature.

The rest of this paper is structured as follows: Section 2 discusses the related work, while Section 3 provides information on our selected applications along criteria we have used to evaluate performance. In Section 4, we present the OpenCL applications performance results on both CPU and GPU platforms along a follow up discussion based on source code analysis. Lastly, in Section 5, we draw conclusions about the paper and share our perspectives on future research directions

2. Related Work

The performance comparison of CPU and GPU platforms can be frequently found in literature. S.Kim et al [18] evaluated the performance of the HiBench benchmark suite on an integrated Intel HD Graphics 4600 GPU. They aimed to determine if the GPU could accelerate MapReduce tasks in an Apache Hadoop data center cluster system. The researchers observed that integrated GPU outperformed CPU with a substantial speed-up.

F Li et al [19] conducted experiments on different architectures to evaluate various programs in the BLAS application suite, including GEMM and GEMV. The researchers used three types of CPUs, namely Xeon E5-2620v3, Intel i7-7700, and i5-7500, and two types of GPUs, GTX1070 and GTX1080 Ti. The findings indicated that the Core i7-7700 outperformed the Xeon E5-2620v3, yet both CPUs were outperformed by the GTX1080 Ti and GTX1070 GPUs. As the matrix dimension increased in GEMM and GEMV, the processing time increased on both architectures, but the GPU still outperformed the CPU.

Z.Huang et al [20] compared the CPU and GPU performance for matrix multiplication programs, which are known to be both popular and time-consuming computing processes. The researchers analyzed the efficiency of GPU computing for various data scales and development methods. They used NVIDIA Tesla P100 GPU and Intel Xeon E5-2640 CPU for the experiments. The results showed that the CPU outperformed the GPU for small input data, likely because the matrix size was small and the degree of parallelism in the GPU was not significant. Consequently, several computation units of GPU remained underutilized. However, as the matrix size increased, the GPU was fully utilized and outperformed the CPU.

V.Saahithyan et al [21] investigated various fundamental image processing algorithms performance on both CPU and GPU, using images of different dimensions for testing. The researchers observed that the GPU's effectiveness in image processing problems is heavily influenced by the size and nature of the problem. As the size of matrix increase, GPU outperformed the CPU in terms of performance, nevertheless, this advantage was limited to a specific matrix size. In contrast, the CPU consistently outperformed under varying matrix sizes. The research concludes that determining the feasibility of utilizing a GPU for image processing tasks is contingent on the problem size and the particular algorithms employed.

Syberfedlt et al [22] use NVIDIA Geforce 980 GPU and intel i7-4790k CPU for experiments and their results indicated that the level of data and the number of parallel instances have a significant impact on

the efficiency of a platform. The CPU is more effective than GPU for little quantities of data, but only up to a certain number of simultaneous instances. Regardless of the quantity of data processed, the GPU will always win over the CPU when there are a high number of parallel instances.

Peitao Song et al [23] explore the potential of modern GPUs by developing a 2D MOC application that takes advantage of the high parallelization capabilities of the MOC method. The authors investigate three levels of parallelization: ray-level, group-energy-level, and polar-angle-level. They compare the performance of NVIDIA GPU with that of the Intel Xeon E5-2690 v4 CPU in both serial and parallel computations using 12 CPU cores. The results show that the GPU outperforms the CPU, achieving speedups of over 55 times and 5 times in comparison to serial and parallel CPUs, respectively.

Chenyang Zhang [23] conducts a performance comparison of the AMD Ryzen 5 2400G integrated GPU architecture with a discrete Core i3-8300 CPU and a discrete GTX 1080 GPU architecture using the iMLBench machine learning benchmark. The findings reveal that the integrated architecture has an average performance that is 7.71% worse than the discrete GPU in certain machine learning applications. However, the integrated architecture performs better in machine learning tasks that have high transmission occupancy, like KNN and BP (Back propagation), due to its zero-copy optimization. It is worth noting that the BP kernel has numerous branches, which can lead to a decrease in GPU performance.

In recent years, the machine learning models are also used to predict the faster platform (CPU or GPU) for a given application [9][24][25]. In these studies, software features are extracted from an application and then correlation analysis is performed to select important features. The machine learning model is trained on selected features to predict the application's performance (i.e., execution time) on both platforms. In this regard, different machine learning algorithms are used in different studies. For instance, the work presented in [9] compares the prediction accuracy of random forest, decision tree and naive bayes machine learning algorithms whereas in [24], KNN, random forest , Linear regression and gradient boosting are compared.

Paulinoet al [26] evaluate the performance of OpenCL code on FPGA as a result of using a variety of coding techniques, such as the use of single-task kernels combined with data vectorization, combined with the use of local memories, and burst accesses to local memory. This study examines these facets using the widely recognized k-means algorithm. Beginning with a sequential OpenCL implementation of the algorithm, make small adjustments, assess each version's performance and power use, and repeat as necessary. Study shows that the FPGA provides speedups up to 1.54 times for four scenarios and energy savings up to 80% in all cases while running the identical OpenCL code on a 4 GHz Intel i7-6700K CPU.

Johnston et al [27] introduces the Architecture Independent Workload Characterization (AIWC) tool as a means to characterize OpenCL kernels based on architecture-independent features. This tool's application is integrated into a methodology aimed at predicting the execution times of accelerators. The method entails leveraging AIWC features to create a model with the ability to predict execution times for a set of 37 computational kernels on 15 varied devices, including CPUs, GPUs, and MIC architectures. The predictive accuracy of this model is impressive, showing only a minor average deviation of 1.2% from experimentally measured run-times. Additionally, the predicted execution time discrepancies range from 9 microseconds to 1 second based on problem size.

Breyer et al [28] conducts a comparative analysis of OpenMP, CUDA, OpenCL, and SYCL, including hipSYCL and DPC++ implementations. They assess usability, performance, and portability across diverse hardware like GPUs from NVIDIA, AMD, and Intel, and CPUs from AMD and Intel. The main contribution of this work is Parallel Least Squares Support Vector Machine (PLSSVM) library that implements backends

for the four aforementioned programming frameworks. By using this library, authors identify the optimal framework for scientific computing and AI workloads based on the target hardware.

In summary, the previous studies have demonstrated that speedup on GPU can be achieved for certain programs, but it is not guaranteed that the GPU will always be faster. The performance of a program depends on the particular constraints of that program, which can determine whether the GPU or the CPU is better suited for running it. We have also seen that different machine learning predictors are used to find a faster platform. However, it specifically emphasizes extracting and analyzing only the features of the host programs, leaving out the extraction of kernel program features. The kernel programs are executed within a specific architecture and play a crucial role in overall system performance [29][30].

We believe that incorporating kernel program features alongside host program features could provide a more holistic understanding of the architecture's performance characteristics and enable a more comprehensive comparison among different programs. In this context we analyze the source code of OpenCL applications by looking at the features of both host and kernel programs. We believe that our thorough source code analysis is useful from two perspectives, Firstly, we understand why a particular platform is better suited for a specific application. Secondly, the limitation of an application is identified that hinders its performance on a particular platform. For instance, a program with too many dependent instructions cannot perform better on a GPU platform (due to limited exploitation of GPU parallelism). The identification of a performance bottleneck is useful for application developers to eliminate the bottleneck in the next version of that application (with improved expected performance of course).

3. OpenCL Application Selection and Methodology

In this section we present the process of our application selection and performance analysis. We select different type of applications from the most famous and widely used Polybench suite [17]. The appropriate application selection is crucial for the fair comparison. For instance, selecting only those applications which are apparently suitable for one particular platform does not seem a fair comparison. It is generally believed that computationally intensive applications are better suited to execute on CPU [31]–[34][35] whereas memory intensive applications benefit from the GPU [36]–[39]. Therefore, this study has selected a mix of 11 OpenCL applications from diverse domains to ensure a fair comparison between the CPU and GPU.

The Table 1 shows the selected applications which belong to the different domains of Image Processing, Linear Algebra, Data mining and Stencils. It can be noticed that 3 selected applications are computationally intensive whereas 6 applications are memory intensive. The remaining two applications are both computationally and memory intensive. Based on the assumption that computation intensive applications should perform better on CPU and memory intensive should perform better on GPU, 2MM, 3MM and Gemm should perform better on a CPU whereas 2D-Convoluton, Correlation, Covariance, ATAX, Bicg and MVT should perform better on GPU. However, we will see later that results are not in line with this assumption and we figure out other factors which actually impact the performance on a particular platform. In this regard the code analysis of an application is essential to explore how performance can be improved on a parallel hardware.

The OpenCL applications are executed as benchmark (without any modifications) on both platforms. Each OpenCL application consists of two programs: a host program and a kernel program. On a CPU, the main control core executes the host program while the remaining cores process the kernel program. However, on a GPU, the CPU executes the host program, and the GPU executes the kernel program. It is important to mention that OpenCL applications are parallel applications which means that application is developed to execute on a parallel platform. Both CPU and GPU are parallel platforms and it is not obvious

which platform is better suited for a particular application (and why). The objective of our study is to find the answer for this research question.

3.1. List of Contributions

After executing each application both on CPU and GPU platforms, we compare their respective execution times to find answers of the following questions.

- Firstly, finding which platform (CPU or GPU) is faster for an application. A platform with less execution time is considered faster indeed. We determine faster platform first of all.
- Secondly, finding why a particular platform is suitable for a particular application. We examine the source code of applications and identify the software features that make them well-suited for a particular platform. In section 4 (results and discussion), we explore the differences in execution time between CPUs and GPUs, and provide an explanation of the software program features that impact the performance on CPU-GPU architectures.
- Thirdly, our study provides a guideline to the application developers by identifying the performanace bottleneck which should be fixed to speed up the application either on CPU or GPU platform.

Table 1. Domain and Applications selected for performance comparison

Domain	Applications	Computation Intensive	Memory Intensive
Stencils	FDTD-2D	✓	✓
Image Processing	2D-Convolution	×	✓
Data mining	Correlation	×	✓
	Covariance	×	✓
Linear Algebra	2MM	✓	×
	3MM	✓	×
	ATAX	×	✓
	Bicg	×	✓
	MVT	×	✓
	Gemm	✓	×
	Gesummv	✓	✓

The summary of our proposed methodology is shown in Fig 1, firstly select an OpenCL application (from a set of applications belonging to different domains) and execute on both CPU and GPU platforms. The execution time is measured on both platforms. Then we compare results and determine the better platform with less execution time. Finally, we investigate why a particular platform is better suited for an application. In this regard, we analyze the source code of that application and find out programing parameters which affect the application's execution time.

4. Result and Discussion

In this section we present our experiment setup and report our findings. The experimental setup is explained in section 4.1. We measure the execution time of 11 selected OpenCL applications on both platforms and categorize our results into two cases. In the first case (section 4.2), there are applications which perform better on GPU. Then in the second case (section 4.3), there are applications which perform

better on CPU. In section 4.4, we investigate through application source code analysis that why an application performs better on a particular platform.

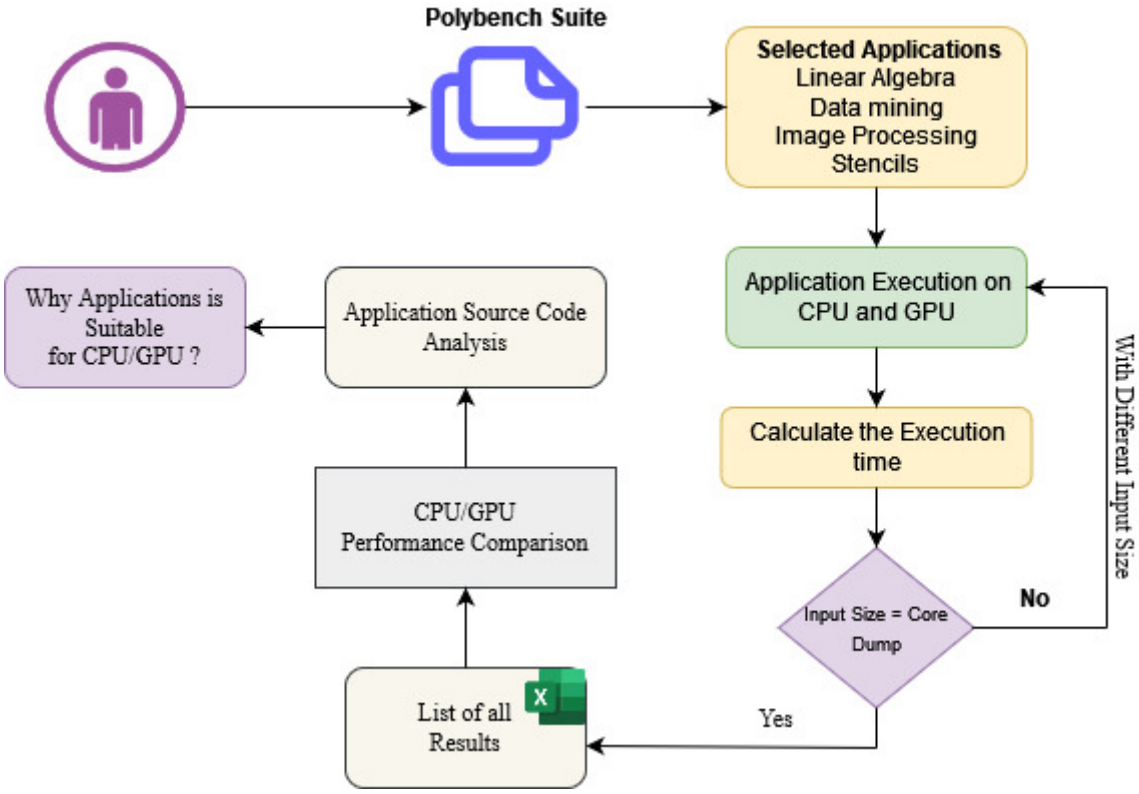

Figure 1. Flow of an application's execution time evaluation on CPU and GPU platforms

4.1. Experimental Setup

We use a Linux based computer system equipped with a sixth generation Intel Haswell i7-6700 processor. The hardware specifications are given in Table 2. The processor includes four cores, operating at a clock rate of 3.40 GHz. In addition, a discrete GPU, the NVIDIA GeForce GT 740, was also utilized, which includes 384 cores and has a memory bandwidth of 28.8 GB/s. The performance of these applications is evaluated on both platforms with varying input sizes. Input sizes are simple floating values, for instance, in the 2mm application, the input size is two, which corresponds to a 2 x 2 matrix. The 3MM application includes three matrix multiplications (A*B*, *C*D, and G = (A*B)*(C*D)). The ATAX kernel is one of the linear algebra kernels, which computes A^T time A_x. It takes A as a matrix of M x N and x as the vector of N length. The MVT application comprises a matrix-vector multiplication, but with a transposed matrix. Matrix A has dimensions of N x N, while vectors y^1 and y^2 both have a length of N. The input size is gradually increased, and the job execution time for each application is measured on both platforms. The application's run time is stored in a file. The run time denotes the amount of time that the CPU or GPU utilizes to execute the application with the provided input size.

4.2. Applications execute faster on GPU

It is observed that the GPU performs better for the 2D Convolution and FDTD-2D applications. The results are shown in figure 2 and 3 for 2D Convolution and FDTD-2D respectively. The input size is shown on x-axis whereas program execution time is shown on y-axis. We gradually increase the input size and measure the effect on the execution time of these applications. In general, we observe that both applications execute faster on GPU since the execution time is lower on GPU as shown in figure 2 and 3. However, for initial small input sizes the difference is small (few milli-seconds) and not clearly visible, however when input size increases the performance difference becomes more evident. For instance, when the input size

of 2D Convolution and FDTD-2D is 15500 (figure 2), GPU takes 0.168965 seconds whereas CPU takes 20.2373 seconds for the execution of the same application. Similar patterns can be found in figure 3.

Table 2. CPU-GPU Hardware Specifications.

Device	CPU	GPU
Model	Intel Core i7-6700	Nvidia GeForce GTX 740
Base Clock	3.4 GHz	0.980 GHz
Boost Clock	4 GHz	1.033 GHz
Total Cores	4	384
Memory Bandwidth	34.1 GB/s	28.8 GB/s

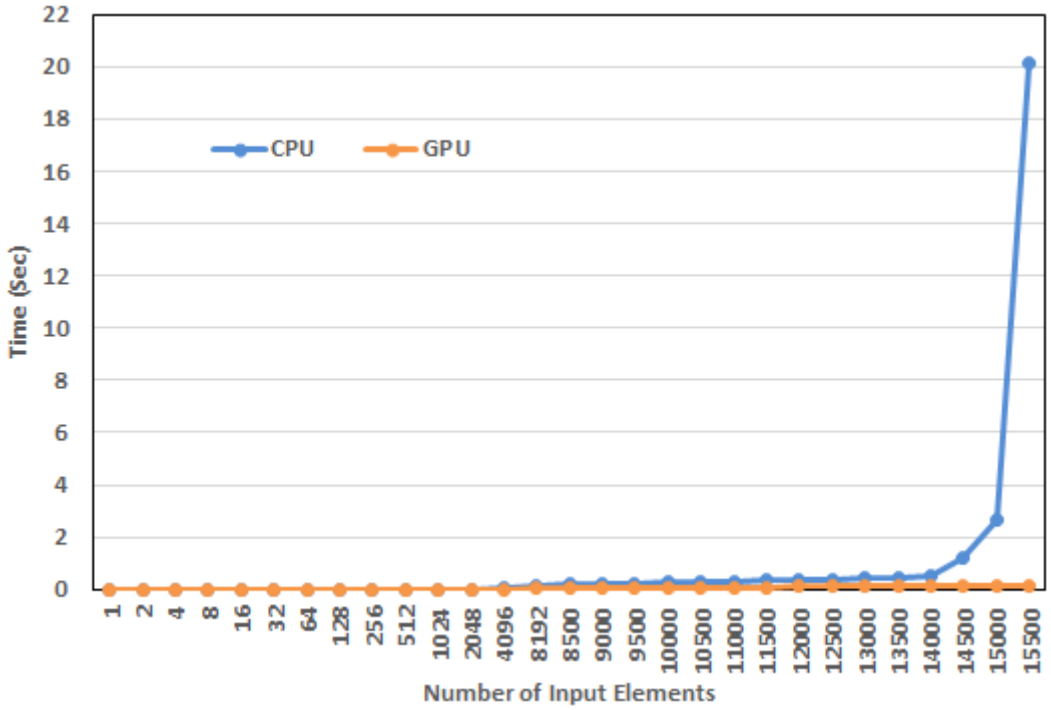

Figure 2. CPU-GPU execution time for 2DCONV Application.

The linear algebra applications, namely 2MM, 3MM, and GEMM, were tested with varying input sizes on both the CPU and GPU platforms. The results for these three applications are shown in figure 4, 5 and 6. Apparently, the pattern of these results looks similar to figures 2 and 3, where the GPU performs better when input size increases. However, when we closely analyze these results, we figure out that CPU exhibited marginally better for some input sizes in the range of 243 to 729 (but difference is very small and negligible).

4.3. Applications execute faster on CPU

MVT, BICG, GESMMV are Linear algebra applications and covariance and correlation are data mining applications that are executed on GPU and CPU with different input sizes. Figures 7-11 show the results for these applications. In all cases, we notice that CPU performs better when input size increases. However, for small input sizes the performance of GPU is slightly better (few milli-seconds) which is not negligible and not very evident in graphs.

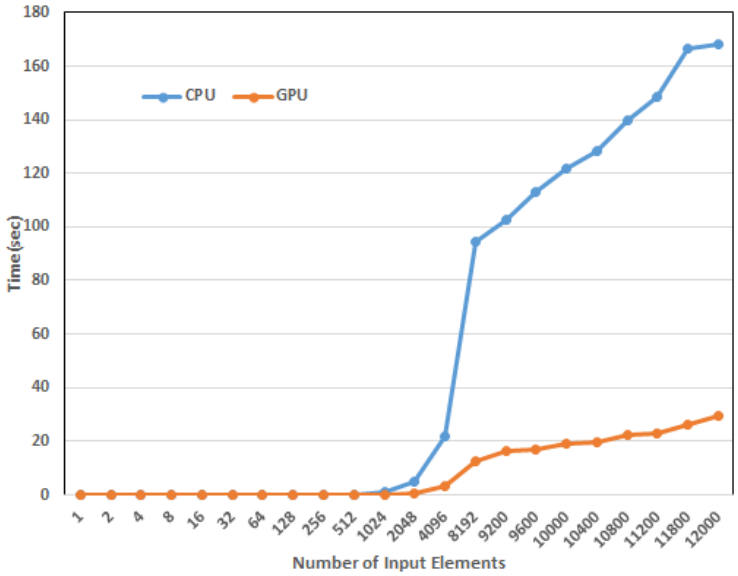

Figure 3. CPU-GPU execution time for FDTD-2D application.

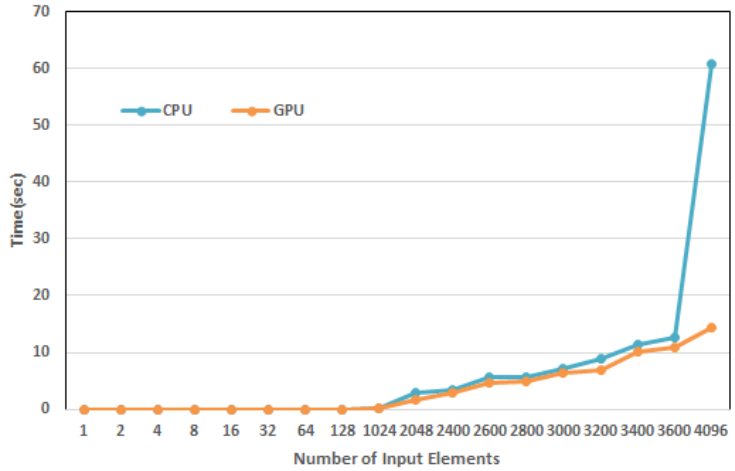

Figure 4. 2MM application execution time of CPU-GPU

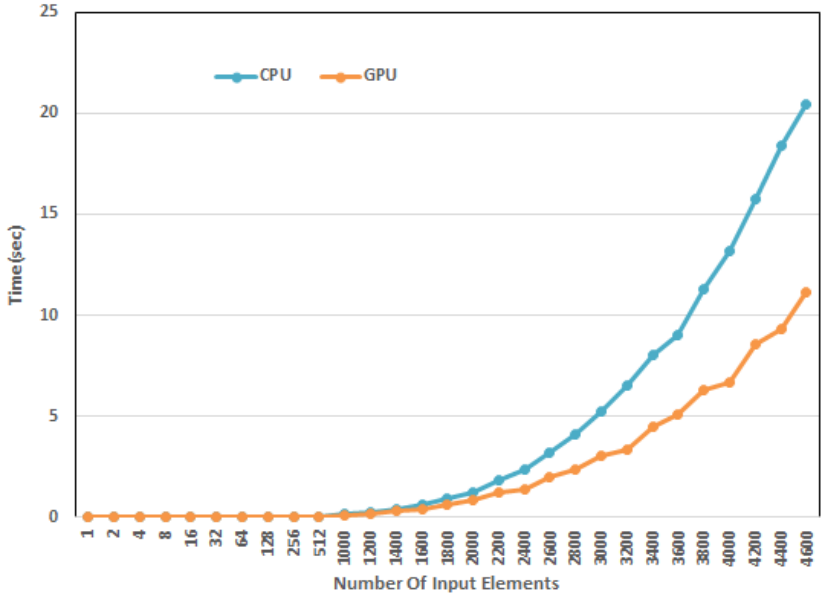

Figure 5. CPU-GPU execution time for 3MM application.

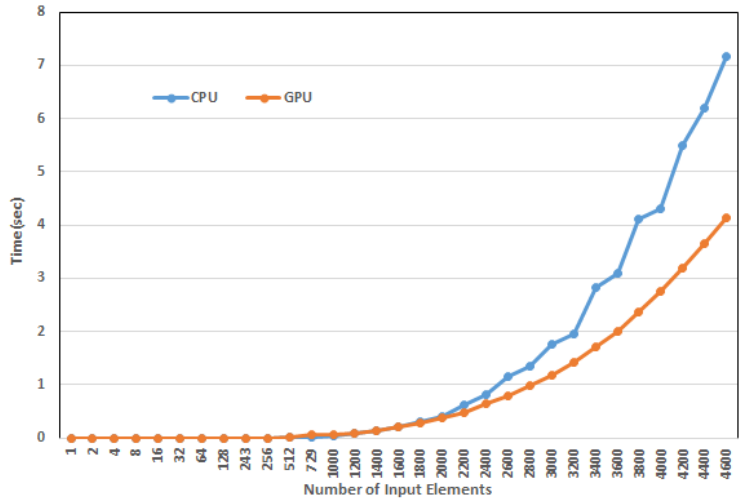

Figure 6. CPU-GPU execution time for GEMM application.

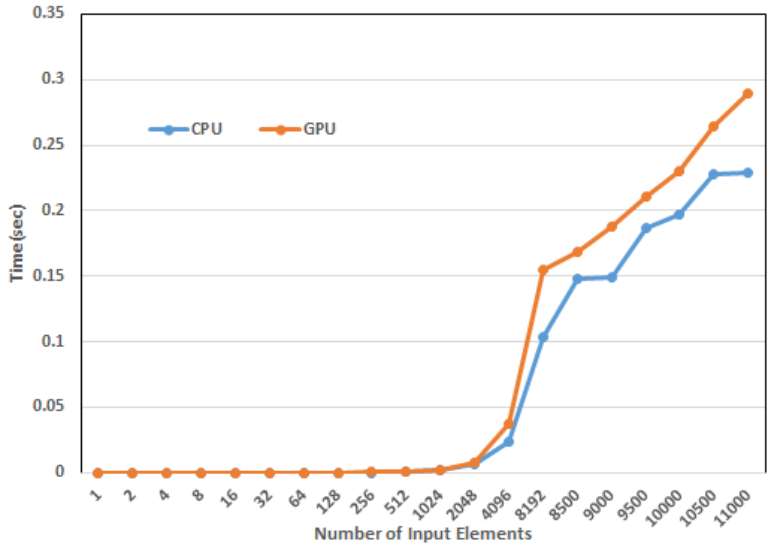

Figure 7. CPU-GPU execution time for MVT application.

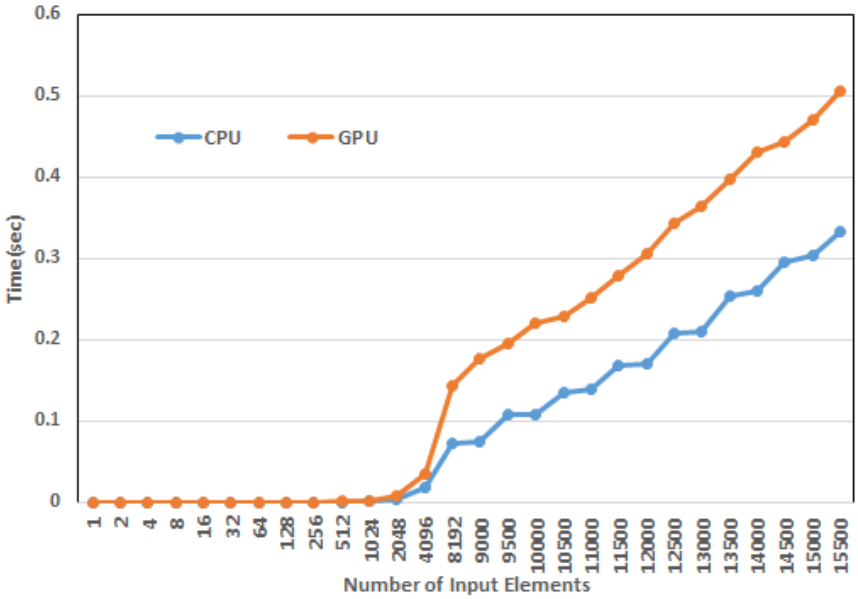

Figure 8. CPU-GPU execution time for BICG application.

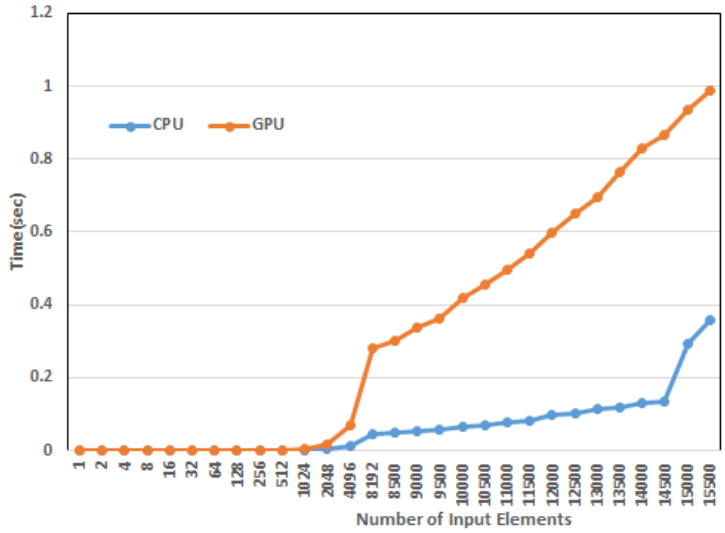

Figure 9. CPU-GPU execution time for GESUMMV application

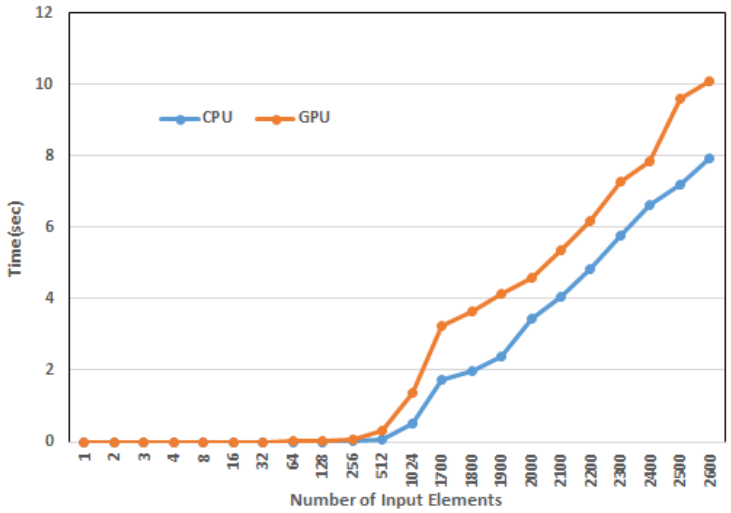

Figure 10. CPU-GPU execution time for covariance application

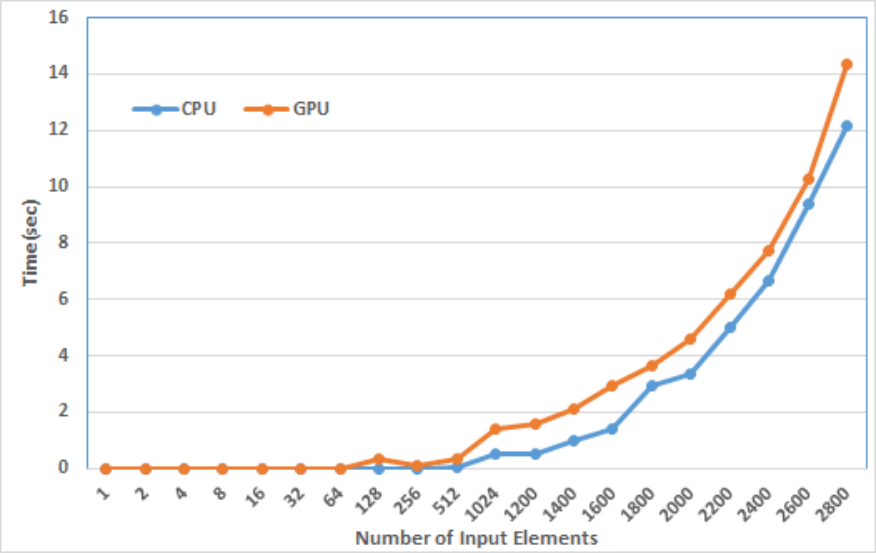

Figure 11. CPU-GPU execution time for Correlation application.

The ATAX application is also tested on CPU and GPU using different input sizes. In this case the CPU consistently outperformed the GPU in terms of execution time for all input sizes. Figure 12 presents the results, which show that the performance difference between CPU and GPU is insignificant for small input sizes. However, as the input size increases, the difference becomes more pronounced. For instance, when the input size is 4096, the CPU completes the execution of the ATAX application in 0.018337 seconds, whereas the GPU takes 8.756636 seconds to complete the same task.

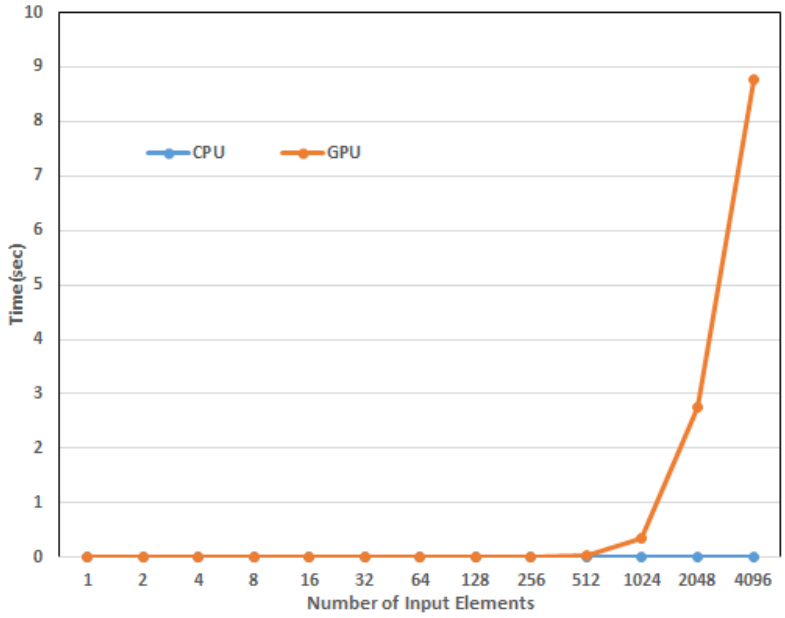

Figure 12. CPU-GPU execution time for ATAX application

4.4. Applications execute faster on CPU

We have conducted experiments to assess the performance of 11 different OpenCL applications on both CPU and GPU architectures. The summary of our results is shown in Table 3. It can be noticed that there is no clear winner (CPU or GPU) and 5 applications (2D Convolution, FDTD-2D, 2MM, 3MM and Gemm) performed better on GPU whereas the remaining 6 applications performed better on CPU. It can be observed that GPU is not a winner at all although it has more potential to perform parallel work (with 384 functional units as compared to 4 CPU cores). Furthermore, in Section 3, we have discussed that generally it is believed that computationally intensive applications are better suited for CPU whereas memory intensive applications are better suited for GPU. However, our results are different since some computationally intensive applications performed better on GPU whereas some memory intensive applications performed better on CPU (Table 3). It indicates that there are certainly other important programming features which may impact the performance of an application on a parallel platform. This leads us to the code analysis of OpenCL applications in order to justify the suitability of an application for a particular platform.

We analyze the source code of both the host and kernel programs of these OpenCL applications. We identify two important program features which affect the performance of an application on CPU-GPU platforms. The first important feature is the level of loop unrolling and the second feature is data dimensionality. Before moving forward, we briefly describe these two features:

Loop unrolling: is a technique used to decrease the overhead of loop control by increasing the size of the loop body. This approach involves rewriting the loop body as a series of individual statements that are executed repeatedly. Loop unrolling is commonly used in many OpenCL programs [40][41].

Dimensionality of input data: OpenCL enables concurrent execution of work items on processing units within a computing unit, which are grouped into workgroups. The supported index space in OpenCL is known as NDRange, which can have one, two, or three dimensions. An integer array of length N specifies the dimensions of an NDRange, with N denoting the number of dimensions. Each work item is assigned a global ID and a corresponding local ID in N dimensions within its workgroup [40].

It is important to mention that a high level of loop unrolling and multi-dimensional input data are desirable features to exploit the potential of a parallel hardware [42][43]. Since it is possible to split a program's instructions and data among multiple computational units in parallel. However, our code analysis reveals how well these features are utilized in these 11 applications and the impact in terms of performance on both CPU and GPU platforms. The detailed program analysis is given below.

First, we report findings of our code analysis for 2D Convolution and FDTD-2D (where GPU performed better for all input sizes). We notice that there are two similarities in the programs of both applications. The first is that data is partitioned into two dimensions. The second is that all loops are unrolled in kernel program. We believe loop unrolling and multidimensional input data contribute towards better performance of GPU since these features support to exploit the parallelism of GPU platform. For instance, one iteration of an unrolled loop or portion of data may be processed on one GPU unit whereas at the same time another loop iteration or portion of data is processed on the other GPU unit.

The source code excerpt for the both host and kernel programs of 2D Convolution application is shown in Figure 13 to demonstrate complete loop unrolling. It can be seen in the host program that there are two nested loops that iterate over the rows (i) and columns (j) of the output matrix B. These nested loops execute the 2D convolution operation, applying a 3x3 filter to the input matrix A. The outer loop iterates over rows, ranging from the second row to the second-to-last row (indices 1 to NI - 2), and the inner loop iterates over columns, ranging similarly from the second column to the second-to-last column (indices 1 to NJ - 2). Within each iteration of the nested loops, the convolution operation calculates the weighted sum of neighboring elements from matrix A using the provided filter coefficients and stores the result in the corresponding position of matrix B.

The code for kernel program Convolution2D_kernel is also shown in the Figure 13. The convolution operation is performed using a 3x3 filter on an input matrix A and the results are written to an output matrix B. The kernel identifies its global indices (i, j) using get_global_id, applies the filter coefficients c11 to c33 to neighboring elements of A, and accumulates the weighted sum to compute the corresponding element of B. The conditional statement ensures the computation is performed only for interior elements, preventing out-of-bounds access. In this code loop unrolling is achieved since each work item computes a single output element independently and thus exploits the parallelism of GPU platform.

According to the study, the performance of 2MM, 3MM, and GEMM applications is generally better on GPU except few input sizes as describe in Section 4.2. The code analysis of these 3 applications identifies some similarities. We find that two-dimensional data usage is a common feature in all three applications. Then we also notice that there is some degree of loop unrolling for all applications. However, the degree of loop unrolling is less than 2D Convolution and FDTD-2D where all loops are unrolled. For these 3 applications, we find only limited loop unrolling. To be more precise, we find two loops in 2MM with two nested loops in each loop, in 3MM there are three loops with two nested loops in each loop, and in GEMM there is one loop with two nested loops. It was observed that only the nested loops were unrolled, while the outermost loop was not unrolled. It shows partial loop unrolling which is different than the complete loop unrolling of 2D Convolution and FDTD-2D. Since the loop unrolling and multidimensionality of data favors to exploit parallelism of GPU, we achieve better results on GPU by enlarge, however, for some input

sizes, the performance of CPU is marginally better. Since the performance difference is very small and negligible, we believe the application is still suitable to execute on a GPU platform.

```
void conv2D(DATA_TYPE* A, DATA_TYPE* B)
{
 int i, j;
 DATA_TYPE c11, c12, c13, c21, c22,
 c23, c31, c32, c33;
 c11 = +0.2;  c21 = +0.5;  c31 = -0.8;
 c12 = -0.3;  c22 = +0.6;  c32 = -0.9;
 c13 = +0.4;  c23 = +0.7;  c33 = +0.10;
 for (i = 1; i < NI - 1; ++i) // 0
  {
   for (j = 1; j < NJ - 1; ++j) // 1
   {
    B[i*NJ + j] = c11 * A[(i - 1)*NJ + (j - 1)]
    +  c12 * A[(i + 0)*NJ + (j - 1)]  +  c13 *
    A[(i + 1)*NJ + (j - 1)]+ c21 * A[(i - 1)*NJ
    + (j + 0)]  +  c22 * A[(i + 0)*NJ + (j + 0)]
    +  c23 * A[(i + 1)*NJ + (j + 0)] + c31 * A[
    (i - 1)*NJ + (j + 1)]  +  c32 * A[(i + 0)*NJ
    + (j + 1)]  +  c33 * A[(i + 1)*NJ + (j + 1)];
              }
       }
}
```

a) Host program

```
__kernel void Convolution2D_kernel
(__global DATA_TYPE *A, __global
DATA_TYPE *B, int ni, int nj)
{
 int j = get_global_id(0);
 int i = get_global_id(1);
 DATA_TYPE c11, c12, c13, c21, c22,
  c23, c31, c32, c33;
  c11 = +0.2;  c21 = +0.5;  c31 = -0.8;
  c12 = -0.3;  c22 = +0.6;  c32 = -0.9;
  c13 = +0.4;  c23 = +0.7;  c33 = +0.10;
 if ((i < (ni-1)) && (j < (nj - 1))
      && (i > 0) && (j > 0))
  {
   B[i*nj + j] =  c11 * A[(i - 1) * nj
   + (j - 1)]  + c21 * A[(i - 1) * nj +
   (j + 0)] + c31 * A[(i - 1) * nj + (j + 1)]
   + c12 * A[(i + 0) * nj + (j - 1)]  + c22 *
   A[(i + 0) * nj + (j + 0)] + c32 * A[(i + 0)
   * nj + (j + 1)]+ c13 * A[(i + 1) * nj +
   (j - 1)]  + c23 * A[(i + 1) * nj + (j + 0)]
   + c33 * A[(i + 1) * nj + (j + 1)];
      }
}
```

b) kernel Program

Figure 13. 2D Convolution Host and Kernel Programs

As an example of partial loop unrolling, we share source code excerpt of 2MM application in Figure 14. it can be seen in the host program that there are two loops each with two nested loops as discussed above. In additional the aforementioned partial loop unrolling can also be observed from the kernel program.

We also analyze the source code of MVT, BICG, GESUMV, Covariance, and Correlation applications which perform better on a CPU platform. We find it common for all applications that they use one dimension of data with partial loop unrolling. In case of MVT there are two loops where each loop contains one nested loop, for BICG there is one loop with a nested loop, for Covariance there are three loops each with a nested loop, and for Correlation application there are four loops each with a nested loop. For all five applications, only nested loop is unrolled from each loop and the outer loop didn't unroll. Since the data is one dimensional and loop unrolling is limited (only one loop is unrolled), something which makes it difficult to exploit the parallelism of a GPU platform, the performance of GPU is not better in general.

As an example of partial loop unrolling in this set of applications, we share source code excerpt of BICG application in Figure 15. it can be seen in the host program that there is only one loop with a nested loop as discussed above. In additional the partial loop unrolling can also be noticed from the associated kernel program.

Through our experiments, we found that ATAX is an application that is better suited for CPU as its execution is consistently faster on the CPU across all input sizes. Our code analysis indicates that there is no loop unrolling at all with single dimensional input data. In the absence of loop unrolling and multi-

dimensionality of data, the performance is not better on GPU platform. The source code excerpt for ATAX is given in the Figure 16 where no loop unrolling can be observed in the kernel program.

```
void mm2_cpu(DATA_TYPE* A, DATA_TYPE* B,
DATA_TYPE* C, DATA_TYPE* D, DATA_TYPE* E)
{
   int i, j, k;
    for (i = 0; i < NI; i++)
      {
        for (j = 0; j < NJ; j++)
         {
           for (k = 0; k < NK; ++k)
                {
                  C[i*NJ + j] += A[i*NK + k] * B[k*NJ + j];
                       }
                }
        }
    for (i = 0; i < NI; i++)
      {
        for (j = 0; j < NL; j++)
          {
            for (k = 0; k < NJ; ++k)
              {
                E[i*NL + j] += C[i*NJ + k] * D[k*NL + j];
                       }
                }
        }
}
```

a) user program

```
__kernel void mm2_kernel1(__global DATA_TYPE
*A, __global DATA_TYPE *B, __global DATA_TYPE
*C, int ni, int nj, int nk)
{
    int j = get_global_id(0);
    int i = get_global_id(1);
      if ((i < ni) && (j < nj))
        {
         int k;
          for (k = 0; k < nk; k++)
              {
  C[i * nj + j] += A[i * nk + k] * B[k * nj + j];
              }
        }
}
__kernel void mm2_kernel2(__global DATA_TYPE
*C, __global DATA_TYPE *D, __global DATA_TYPE
*E, int ni, int nl, int nj)
{    int j = get_global_id(0);
     int i = get_global_id(1);
       if ((i < ni) && (j < nl))
        {
          int k;
           for (k = 0; k < nj; k++)
             {
   E[i * nl + j] += C[i * nj + k] * D[k * nl + j];
              }
        }
}
```

b) Kernel Program

Figure 14. 2MM Host and Kernel Programs

```
void bicg_cpu(DATA_TYPE* A, DATA_TYPE* r,
 DATA_TYPE* s, DATA_TYPE* p, DATA_TYPE* q)
{
 int i,j;
       for (i = 0; i < NY; i++) {
              s[i] = 0.0;
       }
       for (i = 0; i < NX; i++){
         q[i] = 0.0;
          for (j = 0; j < NY; j++)
          {
           s[j] = s[j] + r[i] * A[i*NY + j];
           q[i] = q[i] + A[i*NY + j] * p[j];
              }
       }
}
```

a) Host Prpgram

```
__kernel void bicgKernel1(__global DATA_TYPE *A,
 __global DATA_TYPE *p, __global DATA_TYPE *q,
int nx, int ny)
{
  int i = get_global_id(0);
   if (i < nx){
     q[i] = 0.0;
     int j;
      for(j=0; j < ny; j++)
        {
         q[i] += A[i * ny + j] * p[j];
              }
        }
}
__kernel void bicgKernel2(__global DATA_TYPE *A,
 __global DATA_TYPE *r, __global DATA_TYPE *s,
int nx, int ny)
{
  int j = get_global_id(0);
   if (j < ny){
     s[j] = 0.0;
        int i;
        for(i = 0; i < nx; i++)
        {
         s[j] += A[i * ny + j] * r[i];
              }
        }
}
```

b) Kernel Program

Figure 15. BICG Host and Kernel Programs

```
__kernel void atax_kernel1(__global DATA_TYPE
*A, __global DATA_TYPE *x, __global DATA_TYPE
*tmp, int nx, int ny) {
    int i = get_global_id(0);
     if (i < nx)
      {
        for (j = 0; j < NY; j++)
        {
        tmp[i] = tmp[i] + A[i*NY + j] * x[j];
        }
         for (j = 0; j < NY; j++)
          {
           y[j] = y[j] + A[i*NY + j] * tmp[i];
                }
        }
}
__kernel void atax_kernel2(__global DATA_TYPE
*A, __global DATA_TYPE *y, __global DATA_TYPE
*tmp, int nx, int ny) {
    int j = get_global_id(0);
       if (j < ny)
        {
         int i;
          for(i=0; i < nx; i++)
           {
                y[j] += A[i * ny + j] * tmp[i];
                }
        }
}
```

a) Kernel Program

```
void atax_cpu(DATA_TYPE* A, DATA_TYPE* x,
 DATA_TYPE* y, DATA_TYPE* tmp)
{
  int i,j;

   for (i= 0; i < NY; i++)
        {
                y[i] = 0;
        }

   for (i = 0; i < NX; i++)
    {
        tmp[i] = 0;

   for (j = 0; j < NY; j++)
      {
    tmp[i] = tmp[i] + A[i*NY + j] * x[j];
    }
     for (j = 0; j < NY; j++)
        {
        y[j] = y[j] + A[i*NY + j] * tmp[i];
        }
    }
}
```

b) Host Program

Figure 16. Atax Host and Kernel Programs

Generally speaking, our analysis indicates that applications with the capability of maximum loop unrolling along the use of multi-dimensional input data are better suited for GPU. On the other hand, application without (or limited) loop unrolling and single dimensional input data are better suited for CPU (since it is difficult to exploit the parallelism of GPU platform). The Summary of our finds is presented in Table 4 below.

Table 3. Better performance platform for selected applications

Domain	Applications	Computation Intensive	Memory Intensive	Suitable Architecture
Stencils	FDTD-2D	✓	✓	GPU
Image Processing	2D-Convolution	×	✓	GPU
Data mining	Correlation	×	✓	CPU
	Covariance	×	✓	CPU
Linear Algebra	2MM	✓	×	GPU
	3MM	✓	×	GPU
	ATAX	×	✓	CPU
	Bicg	×	✓	CPU
	MVT	×	✓	CPU
	Gemm	✓	×	GPU
	Gesummv	✓	✓	CPU

Table 4. Summary of OpenCL applications code analysis

S. No	Applications	Multi-Dimensional Data	Loop Unrolling	Suitable Platform
1	2D Convolution FDTD-2D	Yes	Yes	GPU
2	2MM 3MM Gemm	Yes	Partial	GPU
3	Correlation Covariance Bicg MVT Gesummv	No	Partial	CPU
4	ATAX	No	No	CPU

5. Conclusions and Future Work

This study assesses the performance of OpenCL applications by running them on both CPU and GPU systems. The research aims to determine which applications perform better on each architecture and why a particular architecture is more suitable for a specific application. We find that 2DCONV, FDTD-2D, 2MM, 3MM, and GEMM perform faster on GPU, while Correlation, Covariance, ATAX, BICG, GESUMV and MVT execute faster on CPU. We analyze the source code of each application to figure out why a particular architecture is more suitable for a given application. Our study has identified two significant software features that have an impact on the performance of OpenCL applications: the number of data dimensions and the ability to unroll loops. Through the analysis of the 2DCONV and FDTD-2D applications, it was observed that the host program contained two dimensions of data and all loops are unrolled in the kernel program. These features are favorable for leveraging the parallelism of GPU architectures, which is why these applications demonstrated better performance on the GPU. As the degree of multidimensionality or loop unrolling decreases, the performance eventually becomes better on CPU since an application is not been able to effectively exploit the parallelism of a GPU architecture. The ATAX application exhibits better performance on CPU due to the absence of loop unrolling in the kernel program and the use of only one dimension in the host program. In addition, there are some cases when we observe partial loop unrolling. It is the case for 2MM, 3MM, and GEMM applications, the host program has two dimensions, and two of its loops have been unrolled (i.e., nested loops) from each outer loop in kernel program. We find performance is better on GPU in general. In contrast, for the BICG, GESUMMV, MVT, Correlation, and Covariance applications, it is discovered that the host program has only one-dimensional input data and only one nested loop was unrolled from each outer loop in the kernel program. As a result of limited loop unrolling and a single data dimension, these applications exhibited better performance on the CPU. We believe that the performance of CPU suitable applications can be improved considerably on GPU by increasing the data dimensionality and level of loop unrolling. In this context, as a future work, we are planning to modify the source code of these OpenCL applications for improved GPU performance and make a comparison with original applications (i.e., unmodified).

References

1. J. Manferdelli, … N. G.-P. of the, and U. 2008, "Challenges and opportunities in many-core computing," *ieeexplore.ieee.org*, vol. 5, pp. 808–815, 2008, Accessed: Sep. 04, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4484943/.
2. V. Bui, T. Pham, H. Nguyen, H. N. Tran Gia, and T. K. Mohd, "Heterogeneous Computing and the Real-World Applications," *2021 IEEE 12th Annu. Ubiquitous Comput. Electron. Mob. Commun. Conf. UEMCON 2021*, pp. 747–751, 2021, doi: 10.1109/UEMCON53757.2021.9666740.
3. T. Jia, Y. Wei, R. Joseph, and J. Gu, "An adaptive clock scheme exploiting instruction-based dynamic timing slack for a gpgpu architecture," *IEEE J. Solid-State Circuits*, vol. 55, no. 8, pp. 2259–2269, 2020, doi: 10.1109/JSSC.2020.2979451.
4. R. Li and J. Liu, "A Heterogeneous KBA Parallel Algorithm for the Cartesian Discrete Ordinates for Multizone Heterogeneous System," *ieeexplore.ieee.org*, pp. 565–571, 2023, doi: 10.1109/icccs57501.2023.10150477.
5. A. Bloch, S. Casale-Brunet, and M. Mattavelli, "Design Space Exploration for Partitioning Dataflow Program on CPU-GPU Heterogeneous System," *J. Signal Process. Syst.*, 2023, doi: 10.1007/s11265-023-01884-6.
6. Y. Sano, R. Kobayashi, N. Fujita, and T. Boku, "Performance Evaluation on GPU-FPGA Accelerated Computing Considering Interconnections between Accelerators," *ACM Int. Conf. Proceeding Ser.*, pp. 10–16, Jun. 2022, doi: 10.1145/3535044.3535046.
7. A. Rodríguez *et al.*, "Parallel multiprocessing and scheduling on the heterogeneous Xeon+FPGA platform," *J. Supercomput.*, vol. 76, no. 6, pp. 4645–4665, Jun. 2020, doi: 10.1007/s11227-019-02935-1.
8. M. Gu and G. P. Jiang, "Observability of Discrete-Time Two-Time-Scale Multi-Agent Systems with Heterogeneous Features under Leader-Based Architecture," *Mathematics*, vol. 11, no. 8, 2023, doi: 10.3390/math11081907.
9. U. Ahmed, J. C. W. Lin, G. Srivastava, and M. Aleem, "A load balance multi-scheduling model for OpenCL kernel tasks in an integrated cluster," *Soft Comput.*, vol. 25, no. 1, pp. 407–420, Jan. 2021, doi: 10.1007/s00500-020-05152-8.
10. Z. Liu, Z. Xie, W. Dong, M. Yuan, H. You, and D. Li, "A heterogeneous processing-in-memory approach to accelerate quantum chemistry simulation," *Parallel Comput.*, vol. 116, 2023, doi: 10.1016/j.parco.2023.103017.
11. H. Park and S. Kim, "Hardware accelerator systems for artificial intelligence and machine learning," *Adv. Comput.*, vol. 122, pp. 51–95, 2021, doi: 10.1016/bs.adcom.2020.11.005.
12. F. Mayer, M. Knaust, and M. Philippsen, "Openmp on fpgas—a survey," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2019, vol. 11718 LNCS, pp. 94–108, doi: 10.1007/978-3-030-28596-8_7.
13. S. Azmat, L. Wills, and S. Wills, "Accelerating adaptive background modeling on low-power integrated GPUs," *Proc. Int. Conf. Parallel Process. Work.*, pp. 568–573, 2012, doi: 10.1109/ICPPW.2012.77.
14. V. W. Lee *et al.*, "Debunking the 100X GPU vs. CPU Myth: An evaluation of throughput computing on CPU and GPU," in *Proceedings - International Symposium on Computer Architecture*, 2010, pp. 451–460, doi: 10.1145/1815961.1816021.
15. C. Bertoni *et al.*, "Performance portability evaluation of OpenCL benchmarks across Intel and NVIDIA Platforms," *Proc. - 2020 IEEE 34th Int. Parallel Distrib. Process. Symp. Work. IPDPSW 2020*, pp. 330–339, 2020, doi: 10.1109/IPDPSW50202.2020.00067.
16. D. Santos-Martins, L. Solis-Vasquez, A. F. Tillack, M. F. Sanner, A. Koch, and S. Forli, "Accelerating A uto D ock 4 with GPUs and Gradient-Based Local Search," *J. Chem. Theory Comput.*, vol. 17, no. 2, pp. 1060–1073, Feb. 2021, doi: 10.1021/acs.jctc.0c01006.
17. S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a High-Level Language Targeted to GPU Codes." Accessed: Jun. 03, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6339595/.

18. S. Kim, J. Bottleson, J. Jin, P. Bindu, S. C. Sakhare, and J. S. Spisak, "Power efficient MapReduce workload acceleration using integrated-GPU," *Proc. - 2015 IEEE 1st Int. Conf. Big Data Comput. Serv. Appl. BigDataService 2015*, pp. 162–169, 2015, doi: 10.1109/BigDataService.2015.12.
19. F. Li, Y. Ye, Z. Tian, and X. Zhang, "CPU versus GPU: which can perform matrix computation faster—performance comparison for basic linear algebra subprograms," *Neural Comput. Appl.*, vol. 31, no. 8, pp. 4353–4365, Aug. 2019, doi: 10.1007/s00521-018-3354-z.
20. Z. Huang, N. Ma, S. Wang, and Y. Peng, "GPU computing performance analysis on matrix multiplication; GPU computing performance analysis on matrix multiplication," *scholar.archive.org*, vol. 2019, no. 23, pp. 9043–9048, Dec. 2019, doi: 10.1049/joe.2018.9178.
21. V. Saahithyan and S. Suthakar, "Performance analysis of basic image processing algorithms on GPU," *Proc. Int. Conf. Inven. Syst. Control. ICISC 2017*, pp. 1–6, 2017, doi: 10.1109/ICISC.2017.8068687.
22. A. Syberfeldt and T. 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. pp. 01–14, Oct. 06, 2017, doi: 10.5121/ijcsit.2017.9301.
23. P. Song, Z. Zhang, L. Liang, Q. Zhang, and Q. Zhao, "Implementation and performance analysis of the massively parallel method of characteristics based on GPU," *Ann. Nucl. Energy*, vol. 131, pp. 257–272, 2019, doi: 10.1016/j.anucene.2019.02.026.
24. A. Hayat, Y. N. Khalid, M. S. Rathore, and M. N. Nadir, "A machine learning-based resource-efficient task scheduler for heterogeneous computer systems," *J. Supercomput.*, 2023, doi: 10.1007/s11227-023-05266-4.
25. Y. N. Khalid, M. Aleem, U. Ahmed, M. A. Islam, and M. A. Iqbal, "Troodon: A machine-learning based load-balancing application scheduler for CPU–GPU system," *J. Parallel Distrib. Comput.*, vol. 132, pp. 79–94, 2019, doi: 10.1016/j.jpdc.2019.05.015.
26. N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Optimizing OpenCL Code for Performance on FPGA: K-Means Case Study with Integer Data Sets," *IEEE Access*, vol. 8, pp. 152286–152304, 2020, doi: 10.1109/ACCESS.2020.3017552.
27. B. Johnston, G. Falzon, and J. Milthorpe, "OpenCL performance prediction using architecture-independent features," 2018. doi: 10.1109/HPCS.2018.00095.
28. M. Breyer, A. Van Craen, and D. Pflüger, "A Comparison of SYCL, OpenCL, CUDA, and OpenMP for Massively Parallel Support Vector Machine Classification on Multi-Vendor Hardware," May 2022, doi: 10.1145/3529538.3529980.
29. L. Riesebos *et al.*, "Modular software for real-time quantum control systems," 2022. doi: 10.1109/QCE53715.2022.00077.
30. Y. Chen *et al.*, "Smart scheduler: an adaptive NVM-aware thread scheduling approach on NUMA systems," *CCF Trans. High Perform. Comput.*, vol. 4, no. 4, pp. 394–406, Dec. 2022, doi: 10.1007/s42514-022-00110-2.
31. P. M. Athanas and H. F. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis," 1993. doi: 10.1109/2.204677.
32. M. Xin and H. Li, "An implementation of GPU accelerated MapReduce: Using Hadoop with OpenCL for data- and compute-intensive jobs," *Proc. - 2012 Int. Jt. Conf. Serv. Sci. Serv. Innov. Emerg. Econ. Cross-Disciplinary Cross-Cultural Perspect. IJCSS 2012*, pp. 6–11, 2012, doi: 10.1109/IJCSS.2012.22.
33. E. Danovaro, A. Clematis, A. Galizia, G. Ripepi, A. Quarati, and D. D'Agostino, "Heterogeneous architectures for computational intensive applications: A cost-effectiveness analysis," *J. Comput. Appl. Math.*, vol. 270, pp. 63–77, 2014, doi: 10.1016/j.cam.2014.02.022.
34. M. Chadha, A. Jindal, and M. Gerndt, "Architecture-Specific Performance Optimization of Compute-Intensive FaaS Functions," 2021. doi: 10.1109/CLOUD53861.2021.00062.
35. P. Prieto, P. Abad, J. A. Gregorio, and V. Puente, "Fast, Accurate Processor Evaluation through Heterogeneous,

Sample-based Benchmarking," 2021. doi: 10.1109/TPDS.2021.3080702.

36. S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," in *Computer Communication Review*, 2010, vol. 40, no. 4, pp. 195–206, doi: 10.1145/1851275.1851207.
37. S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with GPUs and FPGAs," 2008. doi: 10.1109/SASP.2008.4570793.
38. D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscription," *Proc. - 2020 IEEE 34th Int. Parallel Distrib. Process. Symp. IPDPS 2020*, pp. 451–461, 2020, doi: 10.1109/IPDPS47924.2020.00054.
39. H. Bitalebi and F. Safaei, "Criticality-aware priority to accelerate GPU memory access," *J. Supercomput.*, vol. 79, no. 1, pp. 188–213, Jan. 2023, doi: 10.1007/s11227-022-04657-3.
40. G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for GPGPU programs," *Proc. 2010 IEEE Int. Symp. Parallel Distrib. Process. IPDPS 2010*, pp. 1–11, 2010, doi: 10.1109/IPDPS.2010.5470423.
41. A. Salah, K. Li, K. M. Hosny, M. M. Darwish, and Q. Tian, "Accelerated CPU–GPUs implementations for quaternion polar harmonic transform of color images," *Futur. Gener. Comput. Syst.*, vol. 107, pp. 368–382, 2020, doi: 10.1016/j.future.2020.01.051.
42. F. Kastner, B. Janben, F. Kautz, M. Hubner, and G. Corradi, "Hardware/software codesign for convolutional neural networks exploiting dynamic partial reconfiguration on PYNQ," *Proc. - 2018 IEEE 32nd Int. Parallel Distrib. Process. Symp. Work. IPDPSW 2018*, pp. 154–161, 2018, doi: 10.1109/IPDPSW.2018.00031.
43. J. De Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of High-Level Synthesis Codes for High-Performance Computing," 2021. doi: 10.1109/TPDS.2020.3039409.