International Journal of Research (IJR) Vol-2, Issue-2 February 2015 **ISSN 2348-6848**

Survey on Heterogeneous Computing Paradigms

Rohit R. Khamitkar
PG Student, Dept. of Computer Science and Engineering
R.V. College of Engineering
Bangalore, India
rohitrk.10@gmail.com

Anala M. R.
Professor, Dept. of Computer Science and Engineering
R.V. College of Engineering
Bangalore, India
analamr@rvce.edu.in

***Abstract*—**
Nowadays Graphics Processing Units (GPU) play a vital role in parallel computing. GPUs provide performance in terms of exascale and petascale levels and thus reduce the computing time of applications. These architectures of GPU help to increase the performance of applications compared to conventional CPUs. Compute and/or data intensive tasks can be moved and executed in GPU's parallel microprocessors and the rest of the sequential can be executed in the CPU. This paper discusses on survey of performance comparison of executing applications in conventional CPUs and executing the same applications in GPU in terms of execution time. The experiments used applications like matrix multiplication, differential evolution algorithm, simulated annealing algorithm and other implementations of algorithms for performance comparison. After surveying results of these experiments conducted, it is found that GPUs help increase the performance of applications by reducing their execution time.

***Keywords*—GPU; CPU-GPU; CUDA; OpenACC; Multi-core; Parallel programming.**

I. INTRODUCTION

Parallel programming is a process of decomposing the domain problem into well-defined co-ordinated units. All units are implemented using efficient algorithms and each unit utilizes the available cores in the machine and performs the task in parallel.

A program cannot be completely parallelized. Program contains codes which needs to be executed sequentially and some parts which can be parallelized for improving performance. The programmer has to identify the regions which need to be executed in sequential manner and the regions which can be parallelized. This requires programmer to have strong computational skills and good knowledge of hardware architecture and domain problem.

Four levels of parallelism in hardware are defined in Flynn's taxonomy [Flynn, 1972][1]. They are, Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD) and Multiple Instruction Multiple Data (MIMD). MIMD has two subdivisions and they are, Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD).

Nowadays, Graphics Processing Units (GPUs) have become popular co-processors for complex scientific computing and high performance computing platforms because of their high processing capabilities. Each GPU card has multiple cores (thousands of cores) and each core can parallelly perform a computation. This helps in running the parallelizable code on GPU cores and the serial code on Central Processing Units (CPUs). As these GPUs are used for general purpose computing along with the CPUs, they are now also called as General Purpose GPUs (GPGPUs). Figure 1 depicts the working of this heterogeneous architecture, where GPUs are being used as accelerators [2].

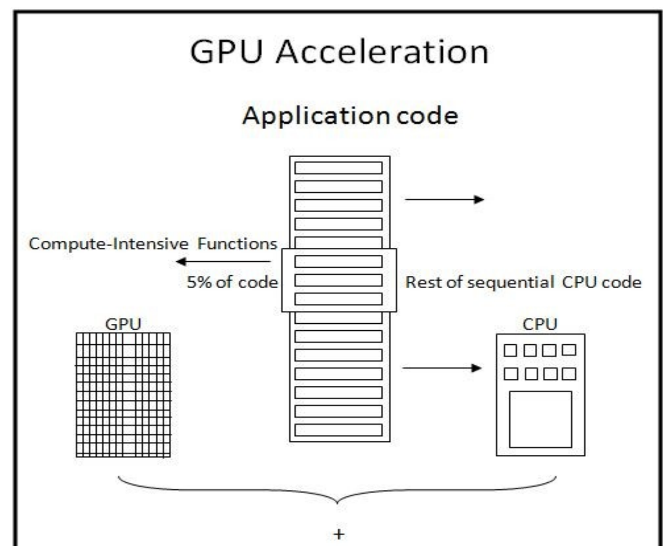

Figure 1: Working of GPU acceleration

Compiler directives, such as OpenMP, can be used for parallel programming using multi-core CPUs in scientific computing where parallelism often appears in regular repetition constructs such as for, while and do loops. Recent versions of OpenMP API also support SIMD programming, programming of accelerators and better optimization using thread affinity. It also provides a new mechanism to specify regions of code where compute and/or data intensive code is moved to another computing device [3].

OpenACC, a new specification for compiler directives, allows annotations of the compute intensive code and data regions similar to OpenMP for standard C and FORTRAN programs that are offloaded to GPUs [4].

The mainstream parallel programming model such as CUDA [Nickolls, Buck, Garland & Skadron, 2008][5] (Compute Unified Device Architecture, developed by NVIDIA and implemented only by the GPUs that they produce) and OpenCL[6], where explicit compute and data management can be done. Using CUDA program developers can directly access the virtual instruction set and memory of parallel computational elements in NVIDIA GPUs. Software developers can access CUDA platform through compiler directives, CUDA-accelerated libraries and extensions to the industry-standard programming languages, such as FORTRAN, C and C++.

CUDA is most widely used programming interface for scientific and high performance computing on GPUs, but has relatively low programming productivity because of its explicit and low- level programming abstractions. Porting an existing CPU-based program to CUDA requires lot of structural changes in the original code and also requires rewriting compute or data intensive CPU code into CUDA kernels.

CUDA has better memory access optimizations because of its low level programming abstractions. It has software-addressable on-chip memory which can be accessed as shared memory. Hence because of these memory access optimizations, CUDA exhibits relatively better performance. CUDA uses a hybrid parallelism of SIMD and SPMD architectures [Hoshino, Maruyama, Matsuoka & Takaki, 2013][7].

OpenACC on other hand is designed to be portable across devices from multiple vendors. However, this design decision restricts OpenACC from effectively utilizing vendor specific architectural features like on-chip memory and because of this some manual code transformation for memory access optimizations, like temporal blocking using the shared memory are prohibited leaving it completely to the compiler [Nguyen, Satish, Chhugani, Kim & Dubey, 2010][8].

OpenACC, in comparison to CUDA, requires fewer code changes for porting CPU-based programs. Its only requirement is annotating compute and/or data intensive code with OpenACC directives. Hence porting using OpenACC is comparatively easier. OpenACC, similar to CUDA, uses hybrid parallelism of SIMD and SPMD architectures [Hoshino et al., 2013][7].

This paper discusses more about survey on performance analysis of GPU-CPU heterogeneous computing using CUDA.

II. CUDA ARCHITECTURE

Figure 2 shows the memory model architecture and Figure 3 shows language Architecture for CUDA. CUDA programming model helps programmers expose fine grained parallelism required by multi-threaded GPUs.

GPU device has three types of memory: texture memory, constant memory and global memory. These are persistent memories and host can read and write to these memories.

The texture memory is cached read-only memory. They are designed for graphics applications that have certain memory access pattern which exhibit spatial locality. Constant memory is a read-only memory having 64KB of size and latency time close to register. Global memory is a read/write memory and can be shared between blocks and grids. They have size greater than 1GB and implement GDDR3/GDDR5 technology. They help data input to and output from kernels. Global memory may or may not be cached depending on compute capability of GPU device.

Multiprocessors have two types of on-chip memory, registers and shared memory, as shown in figure 2.Every processor in a multiprocessor of NVIDIA GPU has a set of 32-bit local read/write registers. Shared memory is shared among all the processors and can be accessed parallelly [Ujaldon, 2012][16].

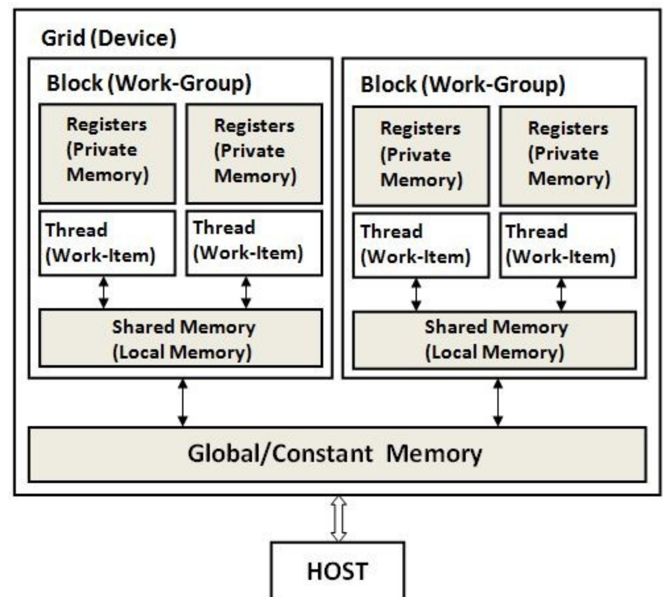

Figure 2: CUDA memory model architecture

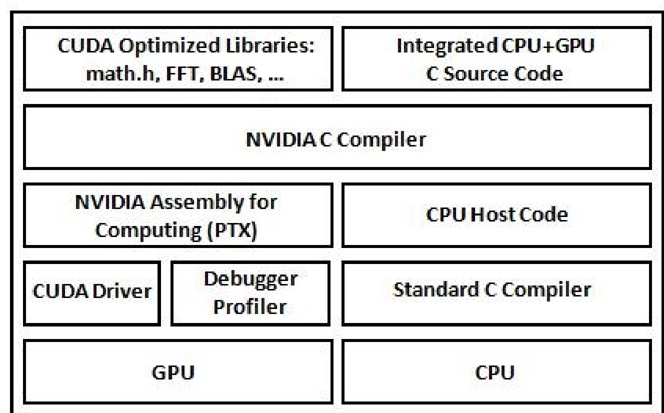

Figure 3: CUDA language architecture

CUDA programming model consists of threads that are categorized into thread blocks, grids and kernels. A thread block consists of a batch of tightly coupled threads identified by thread ID that communicate and synchronize with each other using shared memory and are executed on a single multiprocessor. A thread can also access its local registers and are independent for each thread. A grid consists of a set of loosely coupled blocks that are executed in a single multiprocessor. Blocks within a grid do not communicate or synchronize with each other. A kernel is a CUDA function written by the programmer and executed on the GPU device. Each kernel is executed N times in parallel by N different CUDA threads. The number of CUDA threads that should execute the kernel is mentioned in the kernel call. Warp is a collection of 32 parallel threads. The multiprocessor creates, manages, schedules and executes threads in warps and they execute warps in SIMD fashion. Each warp has consecutively increasing thread IDs and the first warp have thread ID 0. Since CUDA is implemented as an extension to C/C++, programmers can easily learn CUDA programming to parallelize applications in a short duration of time instead of learning a completely new programming language [Ujaldon, 2012][16] [Karunadasa & Ranasinghe, 2009][17] [18].

III. SURVEY ON ANALYSIS OF EXPERIMENTAL RESULTS

Applications that can be parallelized typically involve large problem size and high modeling complexity, i.e they process large amount of data and/or perform many iterations on data. Such problems can be solved using parallel computing. The problem is decomposed into sub problems and each sub problem is solved parallelly.

Following Experiments are surveyed to compare performance between CPU and GPU using different algorithms:

A. In [Zlotrg, Nosovic & Huseinovic, 2011][9], to test and compare GPU performance with the performance of CPU, a program on arithmetic computation is executed with different sizes of an array. Tests were performed under the following configurations:

1. Microsoft Windows 7 Ultimate operating system, Intel Core 2 Duo CPU @ 2.53 GHz with CUDA enabled NVIDIA GeForce 9600 GT GPU.
2. Microsoft Windows 7 Ultimate operating system, Intel Core i5 CPU @ 2.67 GHz with CUDA enabled NVIDIA GeForce GTS250 GPU.

Test for both configurations is repeated four times and the execution time for both GPU and CPU for each test is measured for evaluation.

Figure 4 shows results for configuration 1. From the graph shown in the figure it can be observed that performance of CPU is better for array sizes less than 5000. This is because there are no sufficient data for GPU to efficiently take advantage of data parallelism. There is also overhead involved in data transfer between GPU and CPU, creation of threads and synchronization in the GPU. For array sizes greater than 5000 it can be seen that performance of GPU is better than the CPU as it processes data parallelly. GPU takes only about 0.5 seconds to execute whereas CPU's execution time increases with increase in array size and for data size of 35,000 CPU takes about 6 seconds to execute. Hence for larger data sizes GPU performance is approximately 5 times better than the CPU performance.

In the graph in figure 4it is also observed that at some instances GPU execution time increases suddenly from 0.5 second to about 2 seconds. From further tests it is found that the sudden increase in execution time is because of the overhead in transfer of data from host (CPU) memory to device (GPU) memory. This overhead can be reduced with improved graphics processor and memory.

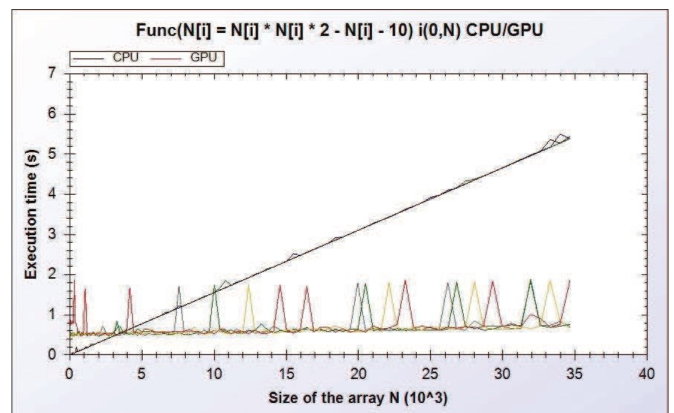

Figure 4: Performance results for configuration 1[Zlotrg et al., 2011][9]

Figure 5 shows results for configuration 2. From the graph it can be observed that performance of CPU is better for array sizes of less than 1000. For array sizes greater than 1000 performance of GPU is better than performance of CPU unlike configuration 1 where performance of GPU is better when array sizes are greater than 5000. It is also observed in the graph in figure 5 that execution times of both GPU and CPU are less compared to the corresponding execution times for GPU and CPU of configuration 1. This is because configuration 2 uses new and powerful hardware for both CPU and GPU. Here it is also noticeable that sudden increase in execution time because of data transfer from host to device memory in configuration 1 is almost completely eliminated in configuration 2.

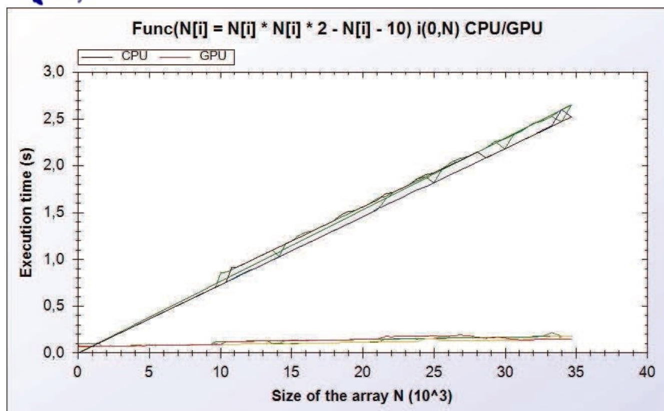

Figure 5: Performance results for configuration 2[Zlotrg et al., 2011][9]

B. Following experiment performed in [Bajrami, Ašić, Cogo, Trnka & Nosovic, 2012][10] to compare the performance of GPU and CPU, uses a hardware configuration consisting of Intel Core 2 Quad Q6600 CPU, NVIDIA GeForce G 103M GPU. This test uses Simulated Annealing algorithm, a metaheuristic algorithm based on material crystallization process. This process is done while the temperature is being increased to the melting point and then decreased to the minimum pre-defined value. The algorithm was implemented in C for executing in CPU and in CUDA C for executing in GPU. The experiment is repeated for different number of starting points.

Figure 6 shows the results for performance comparison between CPU and GPU with increasing number of starting points and threads. It is observed that for small number threads with small data sets take more time to execute as it involves overhead in data transfer between host and device memory. By increasing the number of threads the performance of GPU becomes better compared to both CPU and previously selected number of threads.

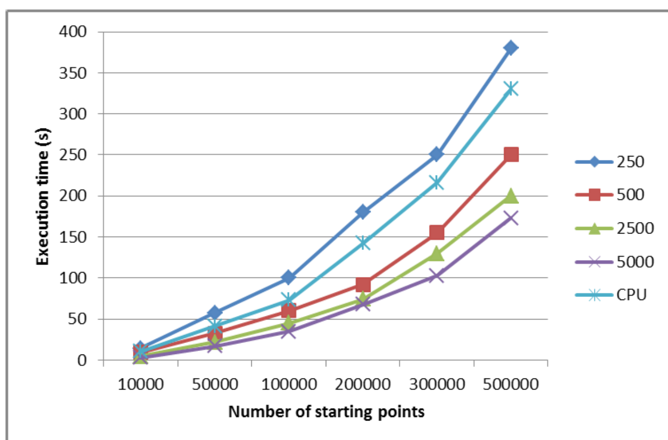

Figure 6: Comparison of performance between CPU and GPU with different number of starting points and threads (color)[Bajrami et al., 2012][10]

C. The Sparse Matrix-Vector multiplication using iterative method is implemented in CUDA [Hassani, Fazely, Choudhury & Luksch, 2013][11] to check the performance of GPU. The hardware configuration consists of AMD x64 @ 2.1GHz having 64 cores, NVIDIA Tesla c2070 GPU and 64-bit Linux operating system.

Figure 7 shows the result for the experiment conducted. From the figure it can be concluded that performance of GPU is better than the performance of the CPU.

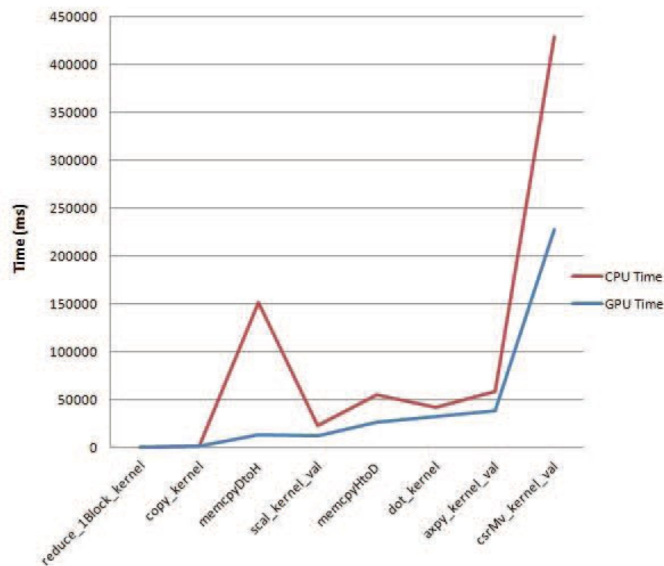

Figure 7: Performance results of CPU Vs GPU for Sparse Matrix-Vector Multiplication [Hassani et al., 2013][11]

D. In the following experiment conducted in [Veronese & Krohling, 2010][12], Differential Evolution (DE) algorithm implemented in C-CUDA is used for execution of algorithm in GPU and an implementation of the DE algorithm in C for execution in CPU.

The experimental setup consists of AMD Athlon x2 5200+ @ 2.7GHz Dual Core CPU with 512KB of cache per core, 3GB DDR2 RAM @ 800MHz, NVIDIA GTX 285 GPU with 1GB GDDR3 memory.

The benchmarked functions, numbered from f_1 up to f_6, correspond to functions, numbered from f_4 up to f_9, respectively in [Yao, Liu & Lin, 2004][13].The experiment is run as two case studies.

Case Study 1

The number of dimensions is set to 100 and is kept constant throughout all of experiments conducted. Population size, i.e. number of individuals, is set to 100 for first experiment. Each experiment is iterated 10,000 times. Figure 8 shows the execution time, which is averaged over 20 runs, for all 6 optimization problems implemented in C and C-CUDA. It is observed from the results that the performance of GPU is again better compared to the performance of CPU. The best computational performance is achieved for CUDA-C $f_6(x)$

which has a speed up of about 19 times over respective C implementation.

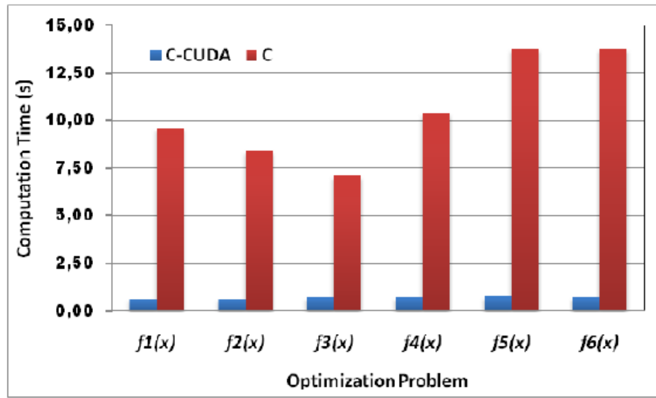

Figure 8: Execution time for *f1* up to *f6* using C and C-CUDA for case study 1 [Veronese et al., 2010][12]

Case Study 2

In case study 2, the experiments are conducted with same number of dimensions as in case study 1 but number of individuals is increased to 1,000 from 100 and maximum number of iterations is increased to 100,000 from 10,000.The computation time for all six optimization problems in C and CUDA-C are shown in figure 9. The best computational performance is achieved for CUDA-C $f_1(x)$ which has a speedup of about 34 times over its respective C implementation.

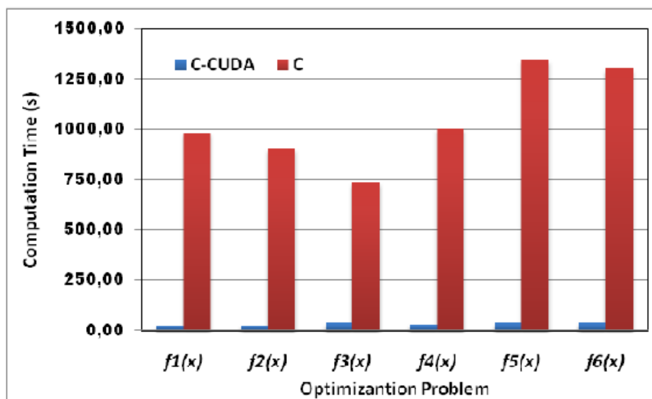

Figure 9: Execution time for *f1* up to *f6* using C and C-CUDA for case study 2 [Veronese et al., 2010][12]

E. In [Thomas & Daruwala, 2014][14], the experiment is performed on host with configuration having Intel Core i5-3210 @ 2.5GHz CPU with 2 cores supporting 2 threads each and has 4GB of RAM. GPU used is NVIDIA GT630M @ 0.95GHz and has 2GB memory. Application is written in two versions, one for CPU and other for GPU. The application performs a task where each element of an array of certain length is incremented. The application is executed 100 times. The computation time for each execution is recorded and the averaged value is considered for results.

Figure 10 shows the results of the experiment. Here N is the number of data elements considered for computation. From the figure it can be deduced that when the value of N is small, the performance of CPU is better than GPU, but as the value of N increases the performance of GPU becomes better.

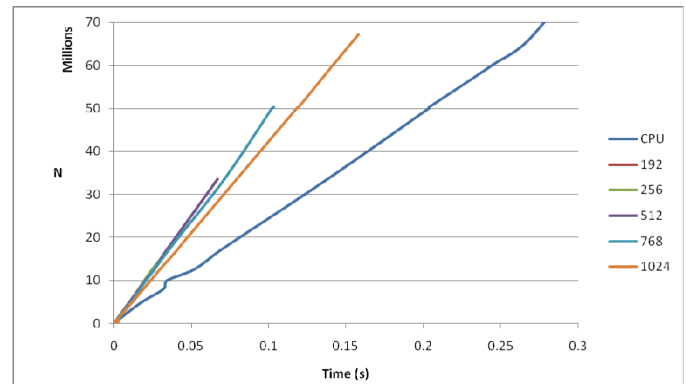

Figure 10: Computation time for CPU and GPU for block sizes of 192, 256, 512, 768 and 1024. N is the number of data elements [Thomas et al., 2014][14]

F. The experiment performed in [Gupta & Babu, 2011][15] uses Natural Language Processing (NLP) application. The main time consuming process in NLP applications is string matching because of large size of lexicon. As data dependency is minimal in string matching process, NLP applications are ideal for parallelization. Here the performance of single-core CPU, multi-core CPU and GPU are compared. Lexical Analysis and Shallow Parsing are implemented as NLP using C++ for single-core, OpenMP for multi-core and CUDA for GPU execution.

The hardware configuration uses, Intel Core 2 Duo P8600 @ 2.40GHz CPU and NVIDIA GeForce G210M @ 800MHz having 16 cores with 1GB GDDR3 memory.

Figure 11 shows the results of the experiment conducted. From the results in figure it can be observed that the performance of the multi-core CPU is better than the performance of single-core CPU and the performance of GPU is better than the performance of multi-core CPU for all experiments conducted.

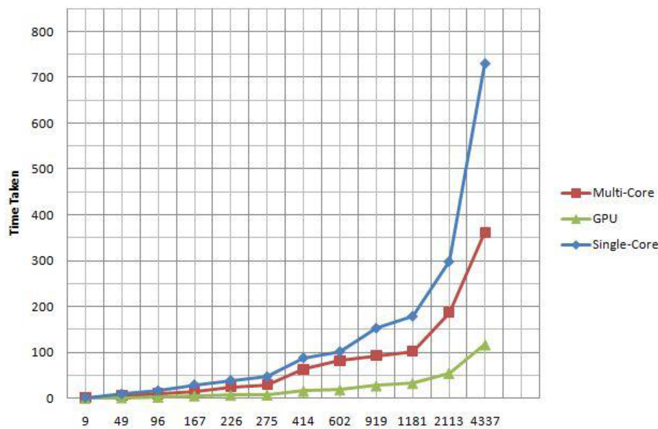

Figure 11: Time taken v/s number of words, performance comparison graph for single-core, multi-core and GPU [Gupta et al., 2011][15]

IV. CONCLUSION

The GPUs have evolved to support general purpose computing to improve the performance of an application by executing them in parallel and thus reducing their execution time. From the survey of results of the experiments conducted for different algorithms, it is concluded that GPGPUs can be efficiently utilized to solve many complex scientific computations.

The challenging part in parallelizing any application is to identify whether the algorithm implemented supports parallelism. Some algorithms perform better when executed sequentially. If these applications are parallelized, then the performance would decrease rather than improving. So first understanding the execution of application is very important before parallelizing. If the current implementation of the algorithm does not support parallelism, then the same algorithm should be modified or a different algorithm should be chosen to perform the same task which supports parallelism. Understanding the architecture of the device is also very important to efficiently parallelize the application. Once all these bottlenecks have been identified and removed, application can be optimized to execute parallelly on the GPU using any of the heterogeneous programming models.

CUDA, OpenACC and many other programming models are available for programming applications for heterogeneous architecture. OpenACC provides high level programming directives using which parallelizing applications becomes easier. OpenACC also supports portability. If the application is designed to be executed on multiple devices from different vendor, then the application should be parallelized using OpenACC. If the application is run on NVIDIA GPUs, then CUDA should be the choice as CUDA provides programmer to efficiently utilize the device specific features like controlling the data movement between host and device memory which helps improve the performance of applications. Since CUDA provides low level programming abstractions, programmer is required to learn the device architecture and rewrite the CPU functions in to CUDA kernels which might affect the existing code structure and performance.

CUDA is still preferred programming model for scientific computing applications as they provide better memory access optimizations which help improve the performance of applications.

GPGPUs are not designed to replace CPU programming. CPUs execute sequential code better than the GPGPUs. GPGPUs should to be used to help CPU applications improve its performance by utilizing GPGPU's parallel architecture for its data/compute intensive tasks.

REFERENCES

[1] M. Flynn, 1972, Some computer organizations and their effectiveness, Trans. Comput. C-21(9): 948–960.

[2] NVIDIA Tesla, "What is GPU computing? GPGPU, CUDA and Kepler explained", http://www.nvidia.com/object/what-is-gpu-computing.html

[3] The OpenMP® API specification for parallel programming, "OpenMP 4.0 Specifications Released", http://openmp.org/wp/2013/07/openmp-40/.

[4] "The OpenACC Application Programming Interface, Version 1.0", November 2011, http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf

[5] J. Nickolls, I. Buck, M. Garland, and K. Skadron, Mar. 2008. "Scalable Parallel Programming with CUDA," ACM Queue, vol. 6, no. 2, pp. 40–53.

[6] Khronos OpenCL Working Group, "The OpenCL Specification, Version 1.2", https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[7] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and, Ryoji Takaki, 2013. "CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application", 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing.

[8] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, 2010. "3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs," in ACM/IEEE International Conference for High Performance Computing, Networking, Storageand Analysis (SC'10), pp. 1–13.

[9] Davor Zlotrg, Novica Nosovic and Alvin Huseinovic, November 2011. "Utilizing CUDA Architecture for Improving Application Performance", 19th Telecommunications TELFOR 2011, IEEE.

[10] Emran Bajrami, Maida Ašić, Emir Cogo, DinoTrnka, Novica Nosovic, 2012. "Performance Comparison of Simulated Annealing Algorithm Execution on GPU and CPU", MIPRO.

[11] Rashid Hassani, Amirreza Fazely, Riaz-Ul-Ahsan Choudhury, Peter Luksch, 2013. "Analysis of Sparse Matrix-Vector Multiplication Using Iterative Method in CUDA", IEEE Eighth International Conference on Networking, Architecture and Storage.

[12] Lucas de P. Veronese and Renato A. Krohling, 2010. "Differential Evolution Algorithm on the GPU with C-CUDA", IEEE.

[13] X. Yao, Y. Liu, G. Lin , 2004. "Evolutionary programming using mutationsbased on Levi probability distribution," IEEE Trans. on EvolutionaryComputation, vol. 8, no.1, pp. 1-24.

[14] Winnie Thomas and Rohin D. Daruwala, 2014. "Performance comparison of CPU and GPU on a discrete heterogeneous architecture", 2014 International Conference on Circuits, Systems, Communication and Information Technology Applications (CSCITA), IEEE.

[15] Shubham Gupta, Prof. M.Rajasekhara Babu, 2011. "Performance Analysis of GPU compared to Single-core and Multi-core CPU for Natural Language Applications", (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 2, No. 5.

[16] Manuel Ujaldon, 2012. "High Performance Computing and Simulations on the GPU using CUDA", IEEE.

[17] N. P. Karunadasa & D. N. Ranasinghe, 2009. "Accelerating High Performance Applications with CUDA and MPI", Fourth International Conference on Industrial and Information Systems, ICIIS 2009, IEEE.

[18] NVIDIA CUDA Toolkit Documentation, "CUDA C Programming Guide", http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.