Self-accelerating Processing Workflows

GPU vs CPU with insights to optimize

- A framework that predicts the optimal platform for a computation -

Progress Report

10th August 2020

University of Moratuwa

Department of Computer Science and Engineering

Supervisors:

- **Appointed**

  Janaka Perera,

  Associate Architect - Accelerated Systems

  LSEG.

- **Internal**

  Dr Adeesha Wijayasiri,

  Senior Lecturer,

  Department of Computer Science and Engineering,

  University of Moratuwa.

Team Members:

- Balarajah Abinayan - 160007J.
- Jeyakeethan Jeyaganeshan - 160256U.
- Thiyakarasa Nirojan - 160442L.

Table of Contents

List of Figures

List of Tables

1. Introduction

Effective computational speed can be increased by assigning computations to their optimal processors. In this research we are considering CPU and GPU. The GPU was originally invented for graphical processing where immense parallel and correlated computations are very common. Later, computer scientists recognized that the GPU was good at other general-purpose programming involving parallel computations; the GPGPU evolved. Though GPUs show higher performance for parallel data streams, CPUs can complete some of the tasks in less time up to some certain limits, we call them benchmarks here. This is because of the overheads due to the high data transfer time between host and device and high context switching time of GPU. The benchmarks vary with hardware of the system being experimented and also depend on the availability of the GPU and CPU resources. CPU and GPU optimal problems are eliminated from this research. The main objective of this research is to find out a solution, a library that determines the optimal processor on which a given specified problem has less latency and high throughput. It would make the effective computing speed faster.

This concept has been adopted from and motivated by the branch prediction concept where a CPU determines in advance, the right branch within program lines that is more likely to be taken. The library is targeted for developers and expected to provide a way to do the processor selection automatically based on attributes related to the computations and present utilization of the processors. Developers have to write code for both CPU and the GPU but do not have to worry which is the execution-time optimal processor. It will be very useful in a heterogeneous environment and this concept might be extensible for other processors, FPGA (Field Programmable Gate Array) and TPU (Tensor Processing Unit). Developers need to pass attributes that are extracted from the computations to the functions that he used from the library. For example, array size, dimension etc. There were several research projects conducted regarding optimizing the performance of heterogeneous systems. None of them has considered present workloads in the system but some operating system scheduling tasks related research projects do.

2. Problem Statement and Motivation

2.1. Problem Statement

A particular task can be done by both CPU or GPU. But the execution time may vary depending on many factors. For some tasks GPU would efficiently save time and for some CPU would. Tasks cannot be pre-classified to determine whether they are efficient to run on GPU or CPU. Depending on the current factors of CPU and GPU it will vary in run time.

A CPU can outperform a GPU up to some limits for some computations that can be executed in both CPU and GPU. Therefore, a right choice reduces the overall execution times of the computations. However, the benchmarks vary with the present workloads in the system, deployment to deployment and based on the time of the day. Hence, the benchmarks are not pre-determinable. Hence the optimal platform cannot be identified during the programming period. It is needed to switch applications specially written for CPU and GPU for optimization manually. Either way, it increases the overall execution time since practical applications consist of mixed kinds of computations.

2.2. Motivation

Effective computational speed can be increased by assigning computations to their optimal processors. It might also prevent starvation in some instances. The branch predictor in CPU hardware was another successful implementation being a motivation for this research. A library of functions evaluating related computations with some characteristics values of the computation and determining a more suitable processor type based on present workloads would help to improve the overall performance of a system.

There are two types of scenarios we are interested in here, batch processing and real time processing. The batch processing has high throughput which executes millions of data points at once. On the other hand, the real time processing has low latency which deals with thousands of data points for a second. A single data point can result in from 25 – 100 GPU kernel launches. Inappropriate scheduling of computations into wrong processors are inefficient and time consuming. If we could prevent such events from happening, the latency can be reduced, the effective speed of the system is boosted and the time savings we could achieve is immense.

For real time, the high latency means growth of pending data to be processed and the incoming rate can not be catched up which might result in total failure of a system depending how critical the real time processing is. For example, the financial Risk management system at LSEG Technology and there are more which are getting such advantages of GPU if used and are similar in nature. Therefore, the same solution to yield the time benefits is usable in all such systems.

3. Research Objectives and Outcomes

Primary objective of this research is a library that evaluates different computational models whether the models should better be executed in GPU or CPU to reduce latency. The library is only provided with estimated numerical properties of the models. The functions make decisions using the dynamic benchmarks obtained periodically from static benchmarks that have been measured with some standards values in long term. The decisions are also expected to be based on the workloads present in the processors. So, the library is independent of hardware details. The framework can not be computational intensive that the library must provide certain gain, reduction in overall execution time over the cost of the decision making process and data transfer time in between host and the device.

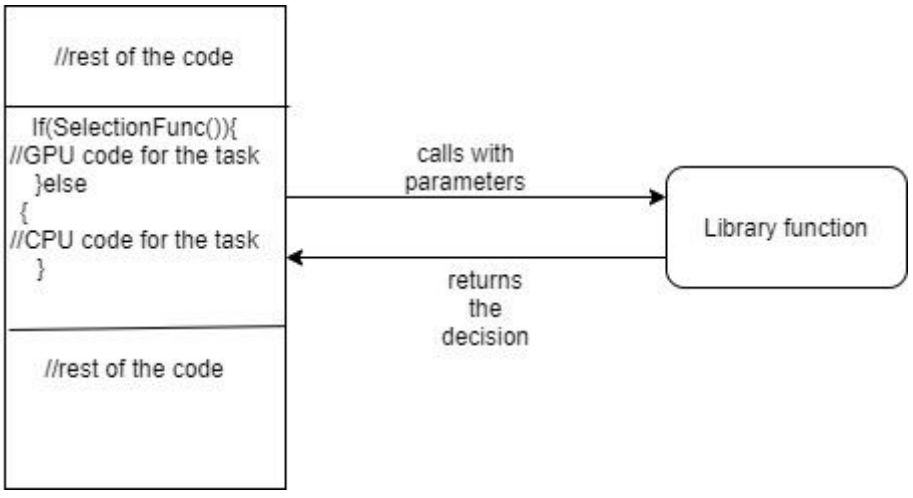


Figure 3.1 - Architectural diagram large in view

The above figure illustrates the solution model of the project. Our main objectives are,

- Find out a strategy to determine the boundary points of computational models and modularize the algorithm.
- Experiment the boundary points of computational models along with performance levels of the system.
- A library implementation in C++ for presenting the solution and the functions contained in the library would return a boolean value representing the processor selection decision.
- A programmer should be able to include computational models as per his needs.

4. Literature Review

4.1. Introduction

A literature review allows one to gain and demonstrate skills in both information seeking and critical appraisal. It also helps to generate a hypothetical analysis of the outcome of a research.

The purpose of this section is to fill the knowledge gap on parallel programming techniques, GPU programming and review the previous art related to the research. The paper [13] argues for the use of a hybrid analytical performance modelling approach is a practical way to build fast and efficient methods to select an appropriate target for a given computatio. This research focuses on the issue of building a selector to decide a parallel loop nest should be executed in a CPU or in a GPU. This paper makes progress toward addressing the following research problem: How to construct runtime target device selection heuristics, what are the biggest challenges involved, and how to make such heuristics suitable for production environments. The ability to automatically choose the processing unit which will execute a given section of code can result in a critical performance advantage. Existing analytical models strive to capture the complexity of the architectures that they are modelling, and the interplay between the levels of abstraction used to represent said architectures.

The main designing goal of any processor is to achieve higher performance in computing; especially to increase the throughput and reduce the latency. The execution time may be more important than the resources being used for some critical and complex problems, and some business applications. A CPU consists of a few cores while a GPU

consists of a few hundred to thousands of cores. Also, the GPU can access an array of memory addresses in parallel as a stream. The parallel computing had a trend and will be the future of computing since the speed of a CPU cannot be increased any more as the number of transistors per square inch is bounded as per Moore's law. The GPUs are used for general purposes (GPGPU) computing such as grid computing, machine learning, data mining, cryptography (neural networks), bioanalysis molecular dynamics. As authors of the paper "CPU - GPU Processing" [1] states, GPGPU vector processing is not the solution to everything and CPUs still do much better than the GPU for certain problems.

The CUDA by NVidia, DirectCompute by Microsoft, OpenCL by Apple/Khronos and OpenGL or DirectX are some popular architectures which enable GPGPU pipelines without the need for data conversions. Floating-point computation was impossible on a GPU but adopted over time for GPGPU and high precision graphics processing [1]. It is possible to dynamically execute a problem on a GPU since kernels are loaded out of the device memory. So, the programming Interface allows to allocate, deallocate and copy data from host to device and vice versa in runtime. The allocation of the device memory can be either merely linear or structured objects such as CUDA arrays. A device memory would have at least 40-bit address space for both linear allocation, and to store references of objects or pointers to the references [8].

4.2. CPU vs GPU

CPUs often have not more than a few tenths of cores and each core accepts an independent instruction set. A host can have one or more CPUs but general hosts are built with single CPUs. The CPUs were single-precision (32-bit) previously, but all modern CPUs support double precision(64-bits) to do lengthy computations. CPUs' cores have few more functionalities from GPUs' cores. It means that a CPU is capable of doing some operations that a GPU cannot do. Although the CPU has a less context switching time, the throughput of a CPU is limited to the number of cores in CPU as the frequency of a core is limited as per Moore's law. Therefore, CPUs are not efficient for huge growing datasets and data science applications since computations involved in such applications are similar but take more time if executed serially. Author of the paper [12] suggests a few optimizations for the CPU to improve performance. They are SIMD purpose reorganization of memory access, multithreading and cache blocking.

The GPUs were originally invented for graphics processing purposes. Therefore traditional GPUs were limited to 24-bit precision because 24-bits were enough to represent the colour of each pixel. They were good at processing similar operations in bulk over a huge amount of data. In the traditional GPUs, multi-instruction emulation of sequences was required for integer arithmetic that are longer than 24-bits [5]. The limitation of serial computing and emergence of parallel computation needs over a large dataset leads to the GPGPU and a new revolutionized GPU which supports 32-bit and 64-bit precisions [5].

Modern GPUs are capable of doing almost all the operations that a CPU can do. A GPU is essentially an array of processors called streaming multiprocessors (SMX). SMXs share a global memory space in the GPU which we generally called the capacity of the GPUs [11], [5]. Each SMX consists of several computation units called cores and can execute at least one kernel over the threads to be executed in the units. It reflects the single instruction multiple data (SIMD) concepts. A SM is scheduled with one or more threat blocks by a hardware scheduler but a threat block can only be scheduled to an SMX [5], [7]. A thread block may consist of thousands of threats, but only any set of threats counts fits the SMX cores (warps are defined as per code) can only be executed at once in an SMX [8].

The threads are organized into thread blocks and grids of a thread block either by the programmer manually or the compiler using predefined rules [5]. Also, each thread block has 16 kilobytes of register memory. The CUDA compiler will automatically utilize the local memory if the limit is exceeded [9]. The computability of a GPU is limited to its bandwidth since it is connected to the host via PCI-Express, meaning that the memory transfer time between the host and the device is significant. Therefore this research needs to consider the memory transfer time to ensure that there is time benefits from the decisions of the functions. Therefore, the arithmetic intensity is used to measure the suitability of a problem to a GPU. The device can also directly access the main memory of the host which reduces the data transfer overhead, but very slow and a rare case scenario [8]. GPUs provide more bandwidth for larger files in general [11]. Also, a computation may not perfectly fit within the hardware implementation of a GPU. Therefore the relationship between the execution time and the size of the computation is not linear because of the varying data rate, segmentation and swapping and pagination overhead. The following table shows how the transfer time changes with the size of the data in a GeForce GTX460 GPU.

Though, this research is independent of hardware specifications and based on the outcomes of the experiments conducted on a computer system that has a GPU device without concerning hardware implementations of the system. However, underlying hardware may affect the relationship between the static and dynamic benchmarks, and it becomes non linear. A GPU is utilized with 100 percent efficiency only when all the threats in a warp are following a kernel throughout. The Fermi GPU is built with Four SFU (Special Function Units) per SM. If computation needs SFU, it would consume more clock cycles. A warp (32 threats) could complete SFU computations over eight clocks [5]. These issues are eliminated by implementing separate specific functions for different computations kinds.

Size of Data Chunk (kB)	GB / second	Time (milliseconds)
1	0.0968	0.01057
2	0.1919	0. 01067
4	0.3399	0. 01204
8	0.6074	0. 01348
16	1.0987	0.01491
32	1.5839	0.02069
64	2.1322	0.03074
128	2.5777	0.05084
256	2.8825	0.09094
512	3.0872	0.1698
1024	3.1711	0.3306
2048	3.1773	0.6602
4096	3.1769	1.3211

Figure 4.1 - Host to device data transfer overhead with file sizes [11]

The context switching time of a CPU is around 10 times smaller than a GPU. So, a program that consists of more serial code is a counterpart to a GPU as it often requires context switching [7]. In practice, some applications can only be executed on a multi-core CPU and some can be on a GPU [7]. But computations that are specific to either CPU or GPU are out of the scope to this research.

In order to get benefits out of accelerators such as GPUs, one has to find computationally expensive (calculation dominated) parts of the program which can run independently and separate them into so-called kernels. These kernels are then executed by the GPU. This

process is not always possible. Some programs have very little computation and a lot of copying memory around. These applications are bandwidth dominated (data dominated) and will not perform well on external accelerators compared to the CPU [3]. This research does not focus on, which computations best fit either for CPU or the GPU. Rather focuses on computations that have boundary points where GPU becomes more suitable when their size or properties exceed some boundary values..

A system composed of computing units, with different characteristics and strategies for data processing is usually called a hybrid system (H-system) due to the presence of heterogeneous computing units [4]. Often, the decisions whether a problem to be executed on which computing unit are hard-coded by programmers based on their applications. This is resulting in inappropriate or inefficient scheduling of jobs and processes and to an unoptimized use of hardware resources.

5. Proposed Solution

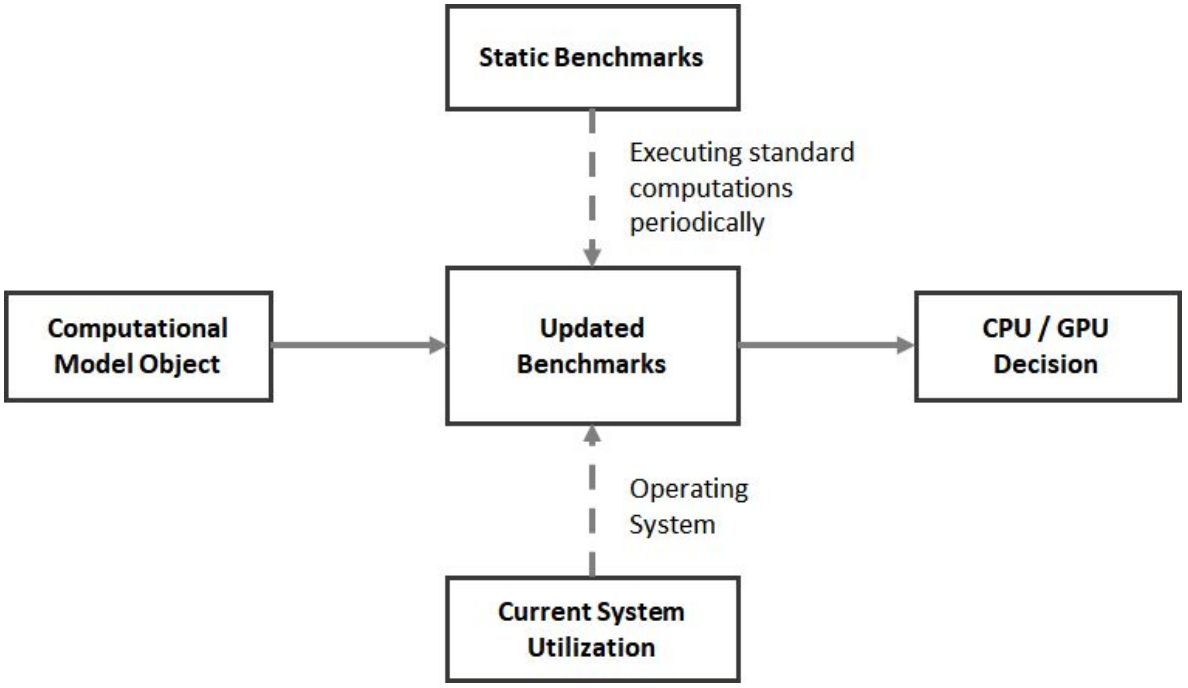


Figure 5.1: High-level Architecture

5.1. Introduction

As explained in section 3, research objectives and outcomes, our proposed solution is a library of functions evaluating computational models. A few models that would give us gain in CPUs execution time over GPUs are experimented in this research. The computations are packed into functions in the library. The functions accept attributes that influence the execution time as arguments. The optimal processor decision-making process is based on benchmarks that are revised from some performantly stored values. This makes the decision process more efficient. It also accounts the utilization of the system every time when decisions are made.

5.2. Operational Mechanisms

Benchmarks of each function where the CPU execution times exceeds the GPU execution time are stored in permanent storage along the execution time of a small computation as a standard. The benchmarks are static and will not be used for the decision making directly. A benchmark updater thread is running to periodically revise the static benchmarks based on the new execution time of the small standard computation of each function. The revised benchmarks are called dynamic benchmarks and either they will not be used for the decision making process. Each function in the library determines the optimal processor by using the related dynamic benchmarks and the present workload in the system gathered using c++ system libraries. This makes the library independent of hardware and present workload balance in the system. It will ensure a gain, reduction in overall execution time over the data transfer time in between host and the device.

The relationship between the static and dynamic benchmarks are not known and have not been analysed well when preparing this progress report. It is necessary to do more experiments in various hardware systems to come to a conclusion regarding the relationships. We call it an alpha relationship. Also, the immediate accounting of the present performance of the system with the data retrieved from the operating system is named as beta-relationship. Effective benchmarks are functions of the alpha, beta and the static benchmarks stored permanently.

5.3. Implementation and Usage

The implementation of the library and benchmarks updater program is implemented in the C++ programming language. Developers can include the library into his project(s) and call appropriate functions, giving related properties as arguments. The functions would return a boolean, true if the processor decision was to the GPU or false if it was the CPU. The functions are probably enclosed within an if-else block where the blocks contain the codes utilizing the CPU and the GPU respectively. The following figure shows the block diagram design of the solution.

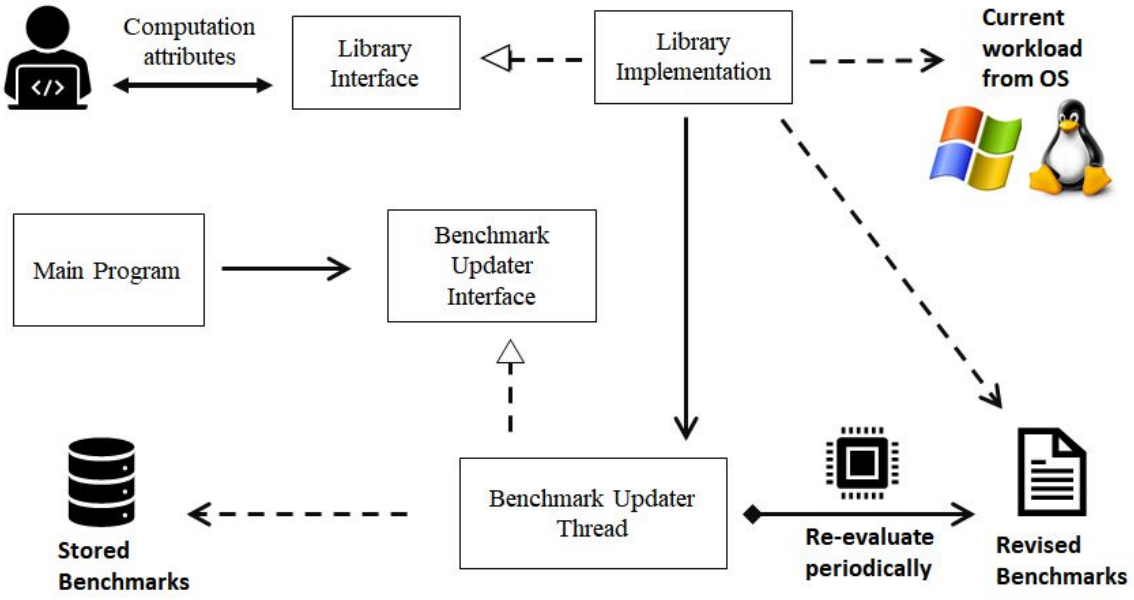


Figure 5.2: Class Diagram

6. Research Methodology

This will essentially be an experimental methodology since the research is based on the results, the benchmarks that are measured in some computers. Execution time data will be measured for each computational model by changing the attributes of the model for both CPU and the GPU. The benchmarks where GPU outperforms CPU could be determined with the data. CPU and GPU execution times of a small standard of the computation are stored along with the benchmarks. Hence, the benchmarks are relative to the standard values not to the hardware architecture.

Methodology is designed as a 4 step process. They are,

1. Analyzing few computational models
2. Implementing the automation codes for testing
3. Establishing the relationship
4. Implementing the framework

6.1. Analyzing few computational models

In this section we will pick some simple computation models and analyze how the attributes relate with the boundary points. It involves the process of identifying the attributes that affect the performance of CPU and GPU. We can extract the features through experiment, and tracking research papers and other documents. For example, the array size and dimension of an array [10]. Mostly we will try to find software-related aspects.

6.2. Implementing the automation code for testing

This section is the implementation section based on results from above. By the experiments, we meant to implement GPU and CPU implementation of the computational models and executing the relevant code on the relevant platforms. This process of experiment is automated. As the output, we will be receiving the time taken to execute the code on the relevant platform.

6.3. Establishing the relationship

We will try to make all aspects fixed except the one we are testing on. Then execute the code on both CPU and GPU separately by changing the independent variable. The graph will be drawn with an independent feature on x-axis and time taken to execute on the y-axis. The results will be used to establish benchmarks in functions. For graphical analysis, we have planned to use Matlab. The above mentioned benchmark is not static. It will be subjected to change based on the machines it is being executed.

6.4. Implementing the framework

The final and important step in the process. In this step, we decide how to make a function toggle between the selection of the appropriate platform for the execution of the code. That is the way of formation of logic behind the functions in our library. We have decided to use statistical results to some extent. The features and results obtained from the above steps will be used. The appropriate platform will be selected on run time based on the relationship that is obtained by executing the inbuilt function on the particular machine. The library is planned to be written in c++.

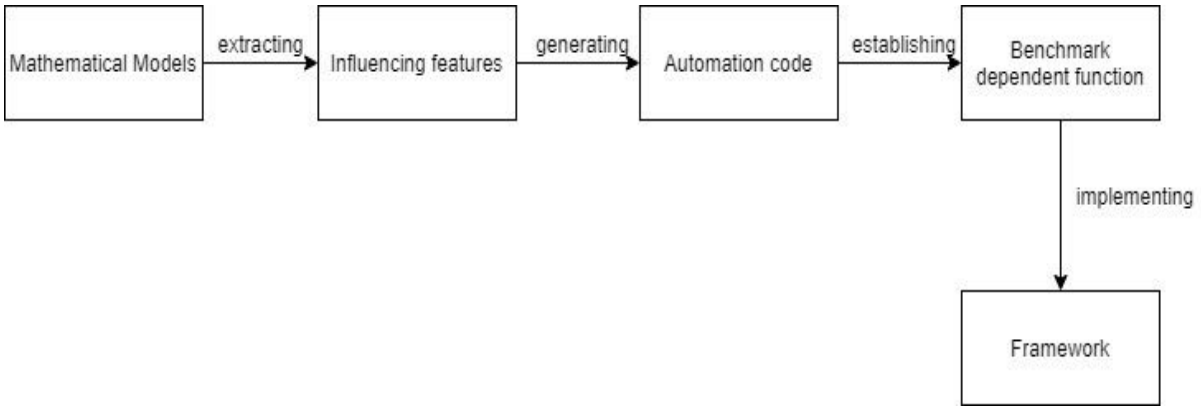


Figure 6.1: Methodology

7. Present Outcomes

Our experiment goal was to find out benchmarks for the functions discovered. There were three functions implemented for three specific computational models that may reduce execution time when executing on CPU up to some limits. Only single parameter computational models were chosen for the experiment to narrow the scope and the experiment time. For each function, enactment time was measured on both CPU and the GPU for different parameter values that were identified as influencing the time. With the data, the benchmark values could be determined. The benchmarks were stored along the execution time of the standard computation, possibly the least value of the parameter experimented.

Though C++ codes were there, it was hard to initialize random arrays of variable length and dimension as per our need quickly in C++. Therefore we have used Pycuda which made the job easy and also the time measurements for the CPU and GPU could be measured with better precision and in the same method so they were comparable. Pycuda accepts Cuda kernel codes written by programmers. Thus, the kernels had to be defined externally ourselves. Though the Pycuda facilitates the compiling and the data transfer between the host and the device itself. Therefore the elapsed time measured had included the data transfer time between the host and the device. Though, we will measure the final benchmarks to be embedded with the library in the C++ itself. Related code implementations have already been created.

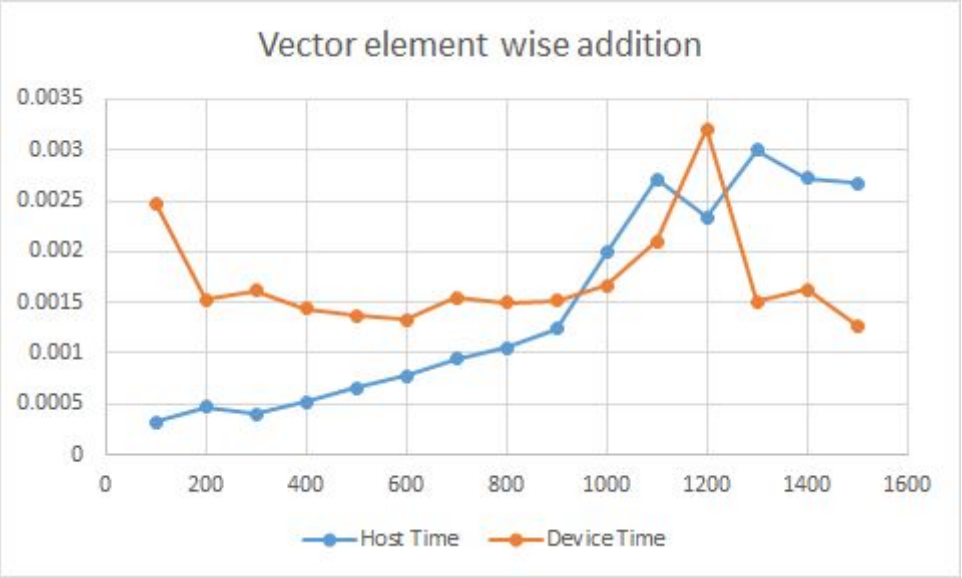
Measurements were taken a certain number of times for each time measuring experiments and averaged to converge the error in the measurements. The attributes values were changed and each function experimented likewise above and similarly for the execution times in CPU. All these processes were automated and results were copied into excel sheets.

1. **Vector addition**

Number of Elements	Host Time	Device Time
100	0.00032395	0.00247511
200	0.00047535	0.00152941
--	--	--
800	0.00105543	0.00149808
900	0.00123664	0.00152057
1000	0.00199642	0.00166710
1100	0.00270865	0.00209979
1200	0.00233752	0.00320343
1300	0.00300046	0.00151011

Table 7.1: Vector addition execution time

The value that has been highlighted in the table is meant to be the standard value for the computational model, vector addition. The same size (100) of a vector addition computation will be executed periodically (once an hour) on both CPU and the GPU. The time elapsed for the computation will be compared with the value that exists in the table to calculate the relative, dynamic benchmarks for the computational model. It will help the library to make decisions independent of hardware and occupancy level of the system. For example, the benchmark for the vector addition in a system that gets the above values for the standard computation execution will be, 950 as per the graph (*Diagram 7.1*) shown below. For the period, functions called with the argument value less than 950 will return true (CPU) and false (GPU) otherwise.



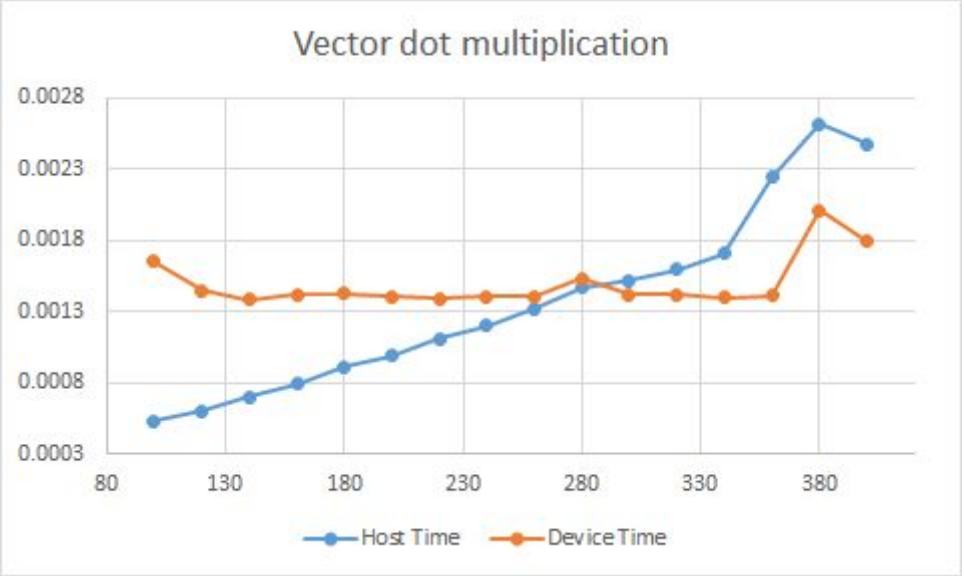
Graph 7.1: Vector addition execution time

The graph shows the benchmark for array size as 950 for vector addition. The CPU time always shows a trend of increasing while GPU time is not changing or decreasing. Therefore there will not be another point where the CPU execution time is less than the GPU time. However, the benchmark varies with some factors as discussed in section 5 and solved with the alpha and beta coefficients.

2. Vector dot-multiplication

Vector Size	Host Time	Device Time
100	**0.00052991**	**0.0016508**
120	0.00060279	0.00144538
240	0.00120232	0.00140586
260	0.00132297	0.00140383
280	0.00146732	0.00153183
300	0.00151745	0.00142245
320	0.00159642	0.00142043
340	0.00170831	0.00139689
360	0.00224578	0.00141597

Table 7.2: Vector dot-multiplication time



Graph 7.2: Vector dot-multiplication execution time

8. Progress in Timeline

Month	Task breakdown	Status
January	● Read and understand the context of the research topic	Completed
February	● Go through previous arts and create Bibliography ● Go deep and Analyze selected previous art	Completed
March	● Narrow down and define the Problem and Motivation ● Define and narrow the scope and the contribution ● Choose a methodology for the experiment ● Define appropriate measurements for the experiment ● Proposal and Proposal Presentation Preparation	Completed
April	● Extracting features ● Prioritizing features ● Set the features into classes	Completed In progress In progress
May	● Select and specify the functions ● Implement related algorithm	Completed
June	● Measuring the impacts of the features ● Create models and design of the Library	In progress
July	● Implement and code the Library	In progress
August	● Prepare for mid evaluation	Completed

9. Things to do

- Experiment the functions that have been found out while different workloads in a system and determine the relationship between the dynamic and static benchmarks (alpha).
- Find a solution to retrieve occupancy levels of CPU and GPU in C++ from different operating systems and relate them to the benchmarks, how they vary with the levels (beta).
- Complete implementation of the library. Design part has been completed almost but implementation is still far away from its completion.
- Library Evaluation: experiment performance of the functions on some machines under various conditions and test performance.
- Explore more functions, analyze their benchmarks and relationship between the benchmarks and performance level of the system if time permits.

10. Conclusion

The research is to analyze processing problems in relation with their properties provided by programmers to predict whether the problem should be executed in the CPU or GPU in order to reduce the computational time. The outcome of the research will be a framework that contains a set of functions to be used by programmers. The functions in the framework make decisions based on the benchmarks stored in the disk not directly but with the relative dynamic benchmarks calculated periodically from the static benchmarks being attached with the library. It is expected to consider the present workload in the system from the OS at the moment when the decision is made to prevent overwhelming either processor. This research is considering a few simple computations to narrow the scope. However, this solution can be scaled up and adopted for many other computational models when it comes to production. But we believe that this research area has a wide scope and several things to be explored and they will be carried out by fellow researchers. This research will help to push the heterogeneous computings into another dimension and provide a boost in the GPGPU to a great extent.

11. References (IEEE)

[1] Z. Memon, F. Samad, Z. Awan, A. Aziz and S. Siddiqi, "CPU-GPU Processing", IJCSNS International Journal of Computer Science and Network Security, Vol.17, No.9, September 2017. [Accessed on: 30- Dec- 2019] [Online]. http://paper.ijcsns.org/07_book/201709/20170924.pdf

[2] 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 & Information Technology (IJCSIT) Vol 9, No 3, June 2017. [Accessed: 31- Dec- 2019] [Online]. http://aircconline.com/ijcsit/V9N3/9317ijcsit01.pdf

[3] S. Brinkmann (2020). "ResearchGate" [Accessed:21 Jan. 2020] [Online]. https://www.researchgate.net/post/Anyone_have_experience_in_programming_CPU_GPU_What_is_the_real_benefit_in_moving_everything_possible_from_CPU_to_GPU_programming

[4] Vella, F., Neri, I., Gervasi, O. and Tasso, S. (2012). A Simulation Framework for Scheduling Performance Evaluation on CPU-GPU Heterogeneous System. Computational Science and Its Applications – ICCSA 2012, pp.457-469. [Accessed:21 Jan. 2020] [Online]. https://link.springer.com/chapter/10.1007/978-3-642-31128-4_34

[5] NVIDIA Corporation, 2020. NVIDIA's Next Generation CUDA Compute Architecture - White Paper. [Accessed 27 March 2020] [online]. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[6] Cullinan, C., Wyant, C. and Frattesi, T., 2020. Computing Performance Benchmarks among CPU, GPU, and FPGA. [Accessed 27 March 2020] [online]. http://Computing Performance Benchmarks among CPU, GPU, and FPGA

[7] S. Goyat, A. Sahoo, “Scheduling Algorithm for CPU-GPU Based Heterogeneous Clustered Environment Using Map-Reduce Data Processing”, ARPN Journal of Engineering and Applied Sciences, Vol. 14, No. 1, January 2019. [Accessed on: 31-Dec- 2019] [Online]. http://www.arpnjournals.org/jeas/research_papers/rp_2019/jeas_0119_7546.pdf

[8] NVIDIA Organization, 2018. Docs.nvidia.com. [Accessed 27 March 2020] [online]. https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf

[9] Bakkum, P. and Skadron, K., 2010. Accelerating SQL Database Operations on a GPU with CUDA. [Accessed 27 March 2020] [online]. https://www.cs.virginia.edu/~skadron/Papers/bakkum_sqlite_gpgpu10.pdf

[10] P. C. Pratt-Szeliga, J. W. Fawcett and R. D. Welch, "Rootbeer: Seamlessly. Using GPUs from Java," 2012 IEEE 14th International Conference on High-Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, Liverpool, 2012, pp. 375-380. https://ieeexplore.ieee.org/document/6332196

[11] Cullinan, C., Wyant, C. and Frattesi, T., 2020. Computing Performance Benchmarks among CPU, GPU, and FPGA. p.15. [online] [Accessed 29 March 2020] https://web.wpi.edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking_Final.pdf

[12] Lee, V., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R. and Dubey P., “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”. [online] [Accessed 29 March 2020] https://dl.acm.org/doi/pdf/10.1145/1815961.1816021?casa_token=WGkb9giVyI8AAAAA:n3DRHmgyO46x6w3e-F12nW-pGJ9P9Pjzvntbs6DdXp4Eg2fYzQxo43Akqde585XkHFjEaDDi0oOd

[13] Chikin, A., Amaral, J., Ali, K., Tiotto, E., “Toward an Analytical Performance Model to Select between GPU and CPU Execution”. [online] [Accessed 8 August 2020]
https://ieeexplore.ieee.org/document/8778216/