# 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

Heterogeneous computing has an emerging trend nowadays. Different processing units in a heterogeneous system makes the application development more challenging [14]. 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 General-Purpose GPU Programming (GPGPU) evolved. Though GPUs show higher performance for parallel data streams, CPUs can complete some small or complex tasks in less time up to some certain limits, we call that limits as 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 and also depend on the availability of the GPU and CPU resources. Problems that can only be solved either by a GPU or a CPU are eliminated from this research. The main objective of this research is to find out a solution, a library that drives the execution flow of incoming tasks into the right processor on which the tasks has less latency and high throughput. It would make the effective speed of the servers faster.

**Commented [1]:** optimal problems???

This concept has been adopted from and motivated by the branch predictor concept where a CPU determines the right branch in a conditional code block within program that is more likely to be taken, in advance. The framework targets the 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 need 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) as well.  A set of attributes that influences the execution load of the computational tasks is gathered from developers using a method and used to predict the appropriate processor. For example, array size, dimension etc. There have been 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.

**Commented [2]:** we didnt utilize the attributes

# 2. Problem Statement and Motivation

## *2.1. Problem*

A task may either be executed in a CPU or a GPU, but the execution time may vary depending on many factors. For some tasks GPU would efficiently save time and for some CPU would. Profit from different processing units for the tasks are varying and depending on various factors such as properties of the accelerators, deployment environment, time of the day, complexity of the tasks and system's current state influence significantly, e.g. contention [14][19]. Hence, the tasks cannot be pre-classified whether they are efficient to run on a GPU or a CPU.

Moreover, A CPU can outperform a GPU up to some limits for some computations that can be executed in both CPU and GPU. Inappropriate scheduling of computations into wrong processors are inefficient and time consuming [17]. Many applications utilize GPUs to seek gain but leave CPUs sitting idle [18]. Therefore, a right choice reduces the overall execution times of the computations. However, the trade-off points (benchmarks) that are used to select the appropriate processor will vary with the present workloads in the system, deployment to deployment and 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 manually switch applications that are specially written for CPU and GPU for optimization purposes. Either way, the overall execution time is high since execution flow in practical applications consist of mixed kinds of computations.

**Commented [3]:** do we need that word???

**Problem statement:**

*"Develop a solution that predicts the optimal processor at runtime which has less latency and high throughput for computations at different instances in a heterogeneous environment."*

**Commented [4]:** modified

## *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 framework consists of models, representing computational tasks, evaluating related computations with some characteristics values of the tasks and determining the most 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 per 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.

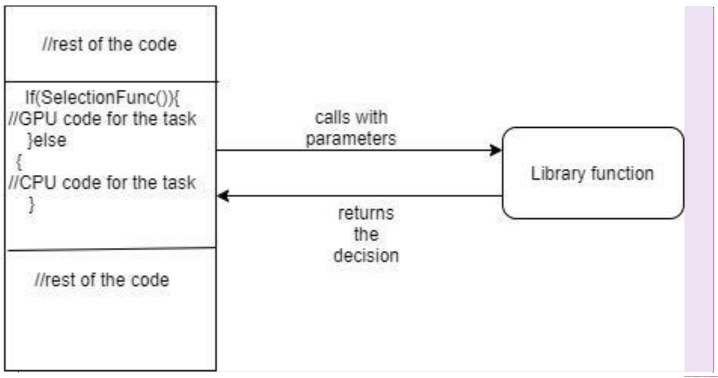
For real time, the high latency means growth of pending data to be processed and the incoming rate cannot be caught up which might result in total failure of a system depending how critical the real time processing is. For example, the financial Risk management system at LSEG Technology and there are many more similar in nature which are facing such troubles in a heterogeneous environment. Therefore, the same solution to yield the time benefits is attainable in all such systems.

## 3. Research Objectives and Outcomes

The primary outcome of this research is a hardware independent framework that could drive incoming tasks into the optimal processing by considering and evaluating some properties of the tasks given by the programmer. The decisions are also expected to be taken based on the workloads present in the processors. So, the library is independent of hardware details.

The framework should also not be computationally intensive as 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.

Solutions for any problem vary based on requirements and optimizations, the implementation part is necessarily left to the programmer. Therefore, the framework should be able to integrate new computational models developed by the programmer for custom tasks. By computational models we mean the different tasks that need to be processed. Programmers can add new solutions models as per requirement by extending an abstract model that comes with the framework.



**Commented [5]:** need to remove???

**Commented [JJ6]:** We have eliminated this model I believe

*Figure 3.1 - Architectural diagram large in view*

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

- Find 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.
- The solution should adapt to the nature of input computations and avoid them assigned to wrong units.
- The evaluation process (execution of library) should be asynchronous to avoid latency in the system by its processing
- A framework implementation in C++ for presenting the solution and the computational models contained in the framework set the optimal processor after a varying constant called, REVISE_COUNT.
- A programmer should be able to integrate new computational models as per his needs.

# 1. 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.

In “Seamlessly Portable Applications” paper[14], the authors try to decouple the accelerator specific codes from the main application flow So it can be deployed across various heterogeneous environments. They create a repository and add the accelerator specific implementations of various tasks for different accelerators in it. As an API to access the repository, they provide a function which is used to filter the implementations according to the need of the programmer. If more than one implementation is found to be compatible with the need of the programmer, the DLS (the system that chooses the appropriate processing unit) will look at the performance database to analyse the history runs of that particular implementation and compare the time taken to execute with the other implementation for that particular problem size. In case not all implementations’ problem sizes are evaluated, the selector tries to interpolate the values [Kicherer et al. 2011] using the runtimes from other problem sizes. If that is not possible, it chooses the unevaluated implementation and measures its runtime to update the database afterwards. In that way, the history-based selector predicts the fastest implementation for given problem size to be executed. This has some impact on the execution time, but as has been shown [Kicherer et al. 2011], benefit can be obtained when more accurate decisions can be made for further runs. But once the platform is selected, it wont switch to another platform during runtime. Further this system can be used for tasks and implementation for functions that are already in the library. Though we can add functions to the repository, there won't be any details related to that implementation in the performance database during initial runs to be used by the history based selector. In such a case the system should execute the tasks and record the time details in the performance database. Though this work is related to our project, the scope is different. Their major task is to decouple the accelerator code from the main flow. The selection of better implementation is secondary. We can use the execution history database idea to improve our prediction.

“Adaptive runtime selection for GPU” paper[15] proposes a profiling technique to select the best GPU version of a task and execute it simultaneously with the CPU version. The processor which finishes the tasks earlier is the winner. Whichever the version that finishes the tasks first will kill the other run. In this way they attain performance gain. In case the CPU wins, it can directly run another code and reuse the GPU as it becomes available again. If GPU is the winner, other threads can be launched on the available cores. In this case, the performance hit may promote the GPU. The aim is to utilize the resources fully at the same time attaining faster performance possible. But running on both CPU and GPU may waste resources or consume additional time than running it alone in CPU or GPU.

STARPU [16] is a framework which provides programmer to achieve efficiency from heterogeneous system during runtime. Since there exists no ultimate scheduling strategy that addresses all algorithms, programmers who need to hard-code task scheduling within their hand-tuned code may experiment important difficulties to select the most appropriate strategy. Many parameters may indeed influence which policy is best suited for a given input to select the appropriate processor. Empirically selecting at runtime the most efficient one makes it possible to benefit from scheduling without putting restrictions or making excessive assumptions. The programmer is free to select a STARPU offered low level scheduling mechanisms (e.g., work stealing) so that scheduler programmers can use them in a high level fashion, regardless of the underlying (possibly heterogeneous) target architecture. Since all scheduling strategies have to implement the same interface, they can be programmed independently from applications, and the user can select the most appropriate strategy at runtime.  Initially the tasks that are to be executed in CPU and other accelerators are divided into "codelets". Codelets are implementations of a particular task for a particular task. These codelets will be queued to the accelerators. The codelets will be selected to execute based on the scheduling strategy.  This method gives the independent to choose the selection strategy to the programmer. But the degree of right prediction also depends on the programmer side. He needs to know about all strategy to use them at right context. But we are trying to dive deep and find an optimal strategy to make prediction rather than depending on a group of strategies. Because this will bound the applicability of our prediction system upto the tasks to which the provided strategies a suited.

In "**Workload Partitioning for Accelerating Applications on Heterogeneous Platforms**" [17] they propose a systematic approach to determine the optimal workload partitioning and the right hardware configuration for accelerating data parallel applications on heterogeneous platforms, a novel analytical partitioning model, based on combining only two metrics (the relative hardware capability and the GPU computation to data transfer gap), to predict the optimal workload partitioning and  a profiling-based method (with multiple profiling options that allow users to tune the overhead-to-accuracy balance) to estimate the two metrics. They try to find the ratio to divide the workload between the CPU and GPU. They also consider the latency in the data transfer to GPU. The ratio is calculated using the performance and data transfer rate which they calculate by using execution time on respective accelerators. Different types of profiling techniques were used to calculate the execution time to find workload partition ratio. In online profiling, where we take the given problem size as input, profile the application one time, and calculate the ratio directly. In offline profiling various sample data sets are used as training data to train linear regression models. The model is not machine learning one but statistical. Hence the partition ratio is calculated and workload is balanced between CPU and GPU. They didn't mention about dynamically balancing the workload during runtime. The workload is balanced only during the initial configuration step.

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 computation. 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.

In “Runtime Coordinated Heterogeneous Tasks in Charm++”, they describe a runtime managed system for coordinating heterogeneous execution with balanced load. This system manages data transfers to and from GPU devices and schedules work across the computational resources of the system. The programmer needs only tag methods and parameters to enable heterogeneous execution. CHARM++  is a task based, asynchronous parallel programming framework with an adaptive runtime system (RTS). Furthermore  they augment the CHARM++ runtime with Accel framework, adding the capability to schedule heterogeneous work across the host and device based on a provided heuristic. The Accel Framework, or ACCEL, dynamically decides where entry methods should be executed. ACCEL has a variety of strategies to determine where to execute particular entry methods. The strategy is passed to as a runtime argument. Example strategies include +accelHostOnly, +accelDeviceOnly,

+accelPercentDevice, which specify a static division of work between the computing resources. This paper discusses about static load  balance and execution in heterogeneous systems during runtime. But we handle workloads dynamically which changes according to the current changes in context of interest.

The paper  "Cost-Aware Function Migration in Heterogeneous Systems"  too considers the performance gain in heterogeneous systems. The idea is based on online learning of the implementations which assist in guided execution of the best implementation. Initially all the available accelerators specific implementations of a specific task is allowed to engage in online learning, i.e every implementation will be executed five times in alternating manner for five different cost values. Cost value is defined as any combination of factors related to the tasks.e.g dimension of the matrix. The cost value against time is logged. Next, the guided execution phase will begin. In this phase, the system will try to search for the current cost value of the data using the results obtained in the online learning phase. If its cost value is already present, then appropriate implementation will be executed. If no implementation can be found for the particular cost value, a regular check is scheduled, the system falls back to the online learning phase for one iteration, and the classification will be updated afterwards. This technique is similar to the history based selection.

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*

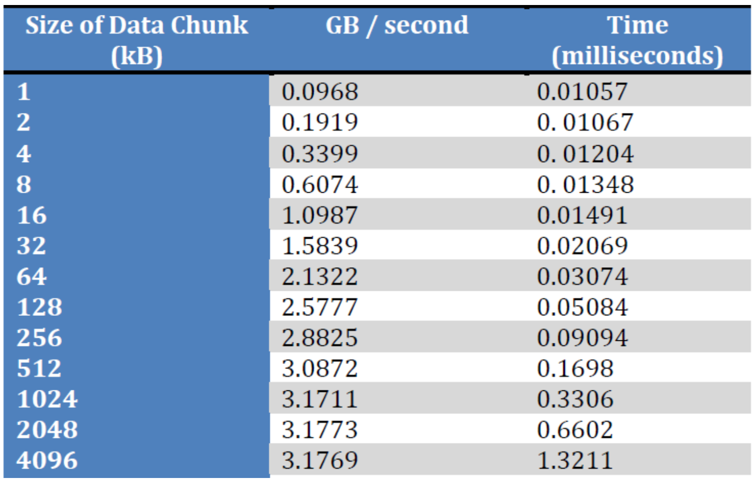
CPUs often have not more than a few tenths of cores and each core accepts an independent instruction set. A host can have one or more CPUs but general hosts are built with single CPUs. The CPUs were single-precision (32-bit) previously, but all modern CPUs support double precision(64-bits) to do lengthy 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 threads) 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.

## 2. Proposed Solution

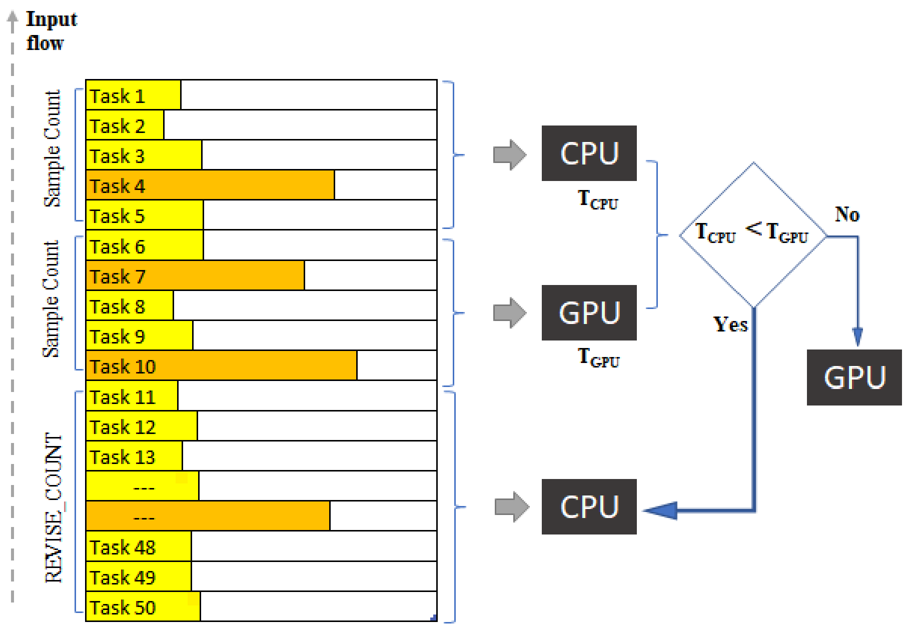


Figure 5.1: Execution Mechanism

### *5.1. Introduction*

From the literature review we were able to identify that the researches have been using three main solution patterns to solve the issue. They have used

1.previous history based model.

2.running in all the available processors and the process that finished first will kill all other processes.

3.load balancing models.

But in which phase of execution, they have used the above techniques is a point to be noted. Most of the researches have used these models to make predictions during the initiation phase of the task. They have not dealt it during runtime dynamically. That is where our novelty idea differ from other solutions.

As explained in section 3, research objectives and outcomes, our proposed solution is a framework consisting of computational models evaluating tasks that are related to them. The framework can be integrated with new computational models having custom implementations for the CPU and the GPU. It can be scaled up for heterogeneous systems containing any number of accelerators. The functions also accept attributes that influence the execution time as arguments to tackle with the nature of the input data. The optimal processor decision-making process is based on sets of sample execution time average and the arguments as well. Initially, two adjacent disjoint samples sets are executed and evaluated in the processing units and the optimal one set for the rest up to a number, called REVISE_COUNT. The optimal processor decision-making process is based on an algorithm which may be involved in many factors. eg.the relative hardware capability, the GPU computation to data transfer gap. The factors are classified under three main categories.

1. Nature of input data - e.g- Aligned or unaligned flow of data.
2. Task based - number of attributes, data types
3. Hardware - software interface - data transfer, allocated block and grid sizes

Further we also consider "raw execution time" analysis in order to include minor influential factors such as contention of the processing units and availability of resources. Thus, it also accounts the utilization of the system every time when decisions are made.

## *5.2. Operational Mechanisms*

The runtime system is developed based on the previous similar works and experiments undertaken. Programmers will create an object of any of the models with some system specific parameters such as number of cores in CPU, etc. Each time he wants to use the model, a programmer will set the data and call execute() method on the object.The execute() method will invoke the best from two functions implemented for CPU and GPU implicitly. He can manually set a fixed processing unit for a problem by giving the unit id to the execute method. E.g. execute(1); for CPU. Manual mode function calls are not evaluated. Initially, two adjacent disjoint samples are executed and evaluated in the processing units and the optimal one set for the rest up to a number called, REVISE_COUNT. Since the samples are evaluated in the machine itself at the time of execution, it 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.

After the REVISE_COUNT exceeded, again samples from each evaluated and switch the processing unit if needed. REVISE_COUNT is incremented if the same unit is set for the next period but reset to a minimum value if the unit has been switched.

The following figure shows how the REVISE COUNT value is incremented..

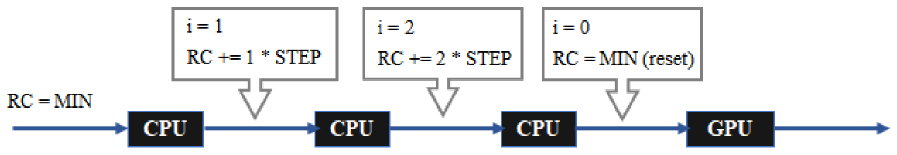


*Figure 5.2: REVISE COUNT increment method*

**Outlier Issues**

One of the major challenges we have in this research is the different patterns of inputs that is being expected. They play a major role in the performance of our prediction system. For example, if there is a sudden surge in the values among small values of the input parameters of a task, we may send it to the wrong processor. This would be a performance degradation for our system. So there is a need to fine tune our framework to tackle this issue. Hence we introduced machine learning model to decrease the effect related to this outlier issue.

The model is trained with the sample data and it will predict data during particular interval. The tasks' input parameters are the features of the model. The label would be the processing time. Here we will be getting both CPU and GPU time labels as an output from the model. Hence from this model alone we can predict the best processor for executing. But the issue is that this model makes prediction based on only the features. Current system context wont be taken into consideration. Further frequently engaging ML model for predicting would incurr additional time in prediction process which would affect our performance.

### *5.3. Implementation and Usage*

The implementation of the framework is in the C++ programming language. The frameworks would consist of a few standard computational models. A new computational model can be added to the framework with custom implementations. Programmer has to extend a class, “ComputationalModel” and must implement two abstract methods, CPUImplementation and GPUImplementation. Therefore the implementation part is customizable to the programmers and gives them full flexibility to add new computational models.

The execute method executes the problem in the appropriate processing unit that has been set with the sampling strategy after the revise count. Also, the method itself is responsible for updating the revise count and setting the optimal processor for the next revise count period.It is overloaded with an argument, processor id which can be used to manually execute a problem in a processing unit without evaluations and is there for the policy of code reusability.

### *5.4. Lacks in this approach:*

1. It is hard to select the most appropriate REVISE_COUNT value.
2. It is waste to evaluate samples unwantedly if one accelerator specific problems are arriving continuously.
3. Evaluating the samples every fixed count would add overhead to the processing and it recoup the gain over the latency from the algorithm.

## 3. Research Methodology

This will essentially be an experimental methodology since the research and solution are based on the evaluation results and conclusions of the experiments. Since the research depends on different groups of aspects, dealing with them all, without classifying them based on their nature will not result in efficient results  The research is subdivided in order to perform the experiments in an efficient way.  Though this system can be used for any tasks, for experimental purposes we choose matrix addition as an example.

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

1. Experiment breakdown
2. Formation and development of the main algorithm
3. Testing the runtime system
4. Formation of library incorporating the runtime selection system

### *6.1. Experiment breakdown*

The following breakdown of the research is done based on the literature survey made. We were able to figure out many factors affecting the performance of heterogeneous systems. So we tried to classify those factors and conduct the experiment in order to gain a maximum output from our research.

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.

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.

- Raw execution time based analysis
- Hardware - software interface based analysis
- Task based analysis
- Raw Execution Time Based Analysis
- Nature of input data based analysis

### *6.2. Formation and development of the main algorithm*

This step is the foundation for the development of the system. Formation of the algorithm is done parallel to the experiments done in step 1. The algorithm is refined in each analysis done in step 1 by incorporating the necessary alterations and insertions in the algorithm.

### *6.3. Testing and implementing the automation code for testing*

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. The runtime system will be tested frequently after every change made in the selection algorithm. Hence the efficiency of the algorithm is confirmed. If not, alternate changes will be made in the algorithm for that particular analysis.

Afterwards, we will begin implementation 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.

***6.4. 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.This is the final step for the formation of the complete runtime system.

It is planned in a way such that initially the programmer will inherit a class from the library and implement the accelerator specific implementation in that class. Then in the main flow of the application the programmer will call a method from the library which is actually an invocation to the appropriate "implementation" selection system. 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 framework is planned to be written in c++.

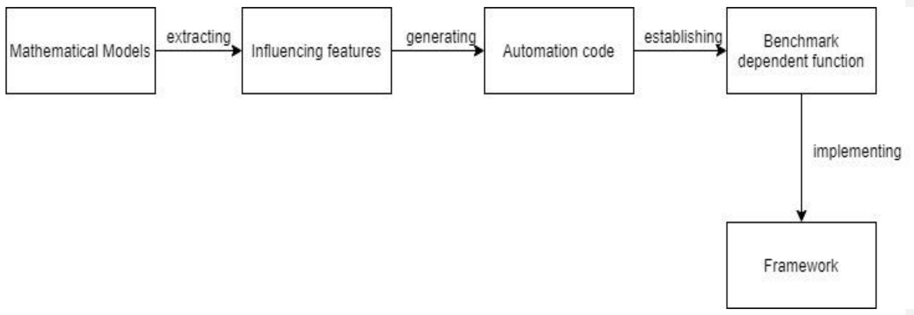


*Figure 6.1: Methodology*

## 4. 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.

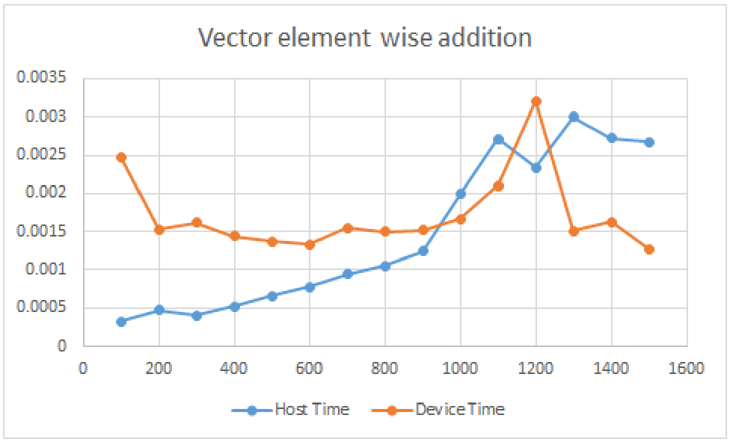
Measurements were taken a certain number of times for each time measuring experiments and averaged to converge the error in the measurements. The attributes values were changed and each function experimented likewise above and similarly for the execution times in CPU. All these processes were automated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



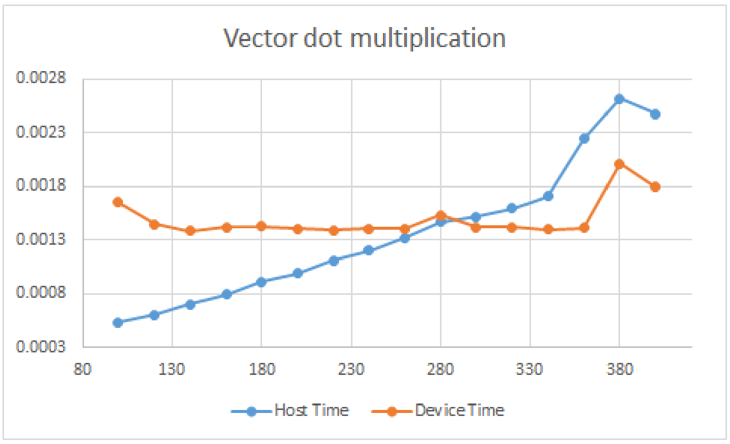
*Graph 7.1: Vector addition execution time*

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

| Number of elements | 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*

## 5. 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<br>● Go deep and Analyze selected previous art | Completed |
| March | ● Narrow down and define the Problem and Motivation<br>● Define and narrow the scope and the contribution<br>● Choose a methodology for the experiment<br>● Define appropriate measurements for the experiment<br>● Proposal and Proposal Presentation Preparation | Completed |
| April | ● Extracting features<br>● Prioritizing features<br>● Set the features into classes | Completed<br>In progress<br>In progress |
| May | ● Select and specify the functions<br>● Implement related algorithm | Completed |
| June | ● Measuring the impacts of the features<br>● Create models and design of the Library | In progress |
| July | ● Implement and code the Library | In progress |
| August | ● Prepare for mid evaluation | Completed |

## 6. Things to do

- Optimize the algorithm with varying REVISE_COUNT adapt to the problem stream nature.
- Combine the results from the Task Based Experiment with selection algorithm to determine optimal processor.
- Make the process asynchronous to reduce the overhead due to the framework calculations.
- Complete implementation and evaluate the library.
- If time permits, explore more functions, analyze their suitability and REVISE_COUNT relations.

## 7. 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 computational models to be used by programmers. The framework makes decisions based executions times of two preceding disjoint sets of samples and the properties provided. Therefore, it will consider the present workload in the system which prevents overwhelming either processors. This research is considering a few simple computations to narrow the scope. Programmers can add new computational models for their needs with their custom implementations. Also, this solution can be scaled up and adopted for many other complex 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.

## 8. 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/

[14] M. Kicherer, F. Nowak, R. Buchty and W. Karl, "Seamlessly portable applications: ACM Transactions on Architecture and Code Optimization", vol. 8, no. 4, pp. 1-20, 2012. Available: 10.1145/2086696.2086721 [Accessed 27 August 2020]. https://dl.acm.org/doi/pdf/10.1145/2086696.2086721

[15] Adaptive runtime selection for GPU

[16] STARPU

[17] Workload Partitioning for Accelerating Applications on Heterogeneous Platforms

[18] Runtime Coordinated Heterogeneous Tasks in Charm++

[19] Automatic task mapping and heterogeneity-aware fault tolerance: The benefits for runtime optimization and application development

•