**Self-accelerating Processing Workflows**

**GPU vs CPU with insights to optimize.**

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

Final Report

**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

[1. Abstract 3](#_Toc64297717)

[2. Introduction 3](#_Toc64297718)

[3. Problem Statement and Motivation 4](#_Toc64297719)

[4. Research Objectives and Outcomes 6](#_Toc64297720)

[5. Literature Review 7](#_Toc64297721)

[6. Methodology 14](#_Toc64297722)

[7. Conclusion 39](#_Toc64297723)

[8. Fellow Research Possibilities 40](#_Toc64297724)

[9. References (IEEE) 41](#_Toc64297725)

List of Figures

Figure 3.1 - Architectural diagram large in view ………………………………………….... 5

Figure 4.1 - Host to device data transfer overhead with file sizes …………………………. 9

Figure 5.1 - Execution Mechanism ……………………………………………………….. 10

Figure 5.2 - Class Diagram ………………………………………………………………… 12

Figure 6.1 - Methodology Follow …………………………………………………...……… 14

Graph 7.1 - Vector addition execution time ………….…………………………………….. 16

Graph 7.2 - Vector dot-multiplication execution time ……………………………………... 17

List of Tables

Table 7.1 - Vector addition execution time ………………………………………………… 15

Table 7.2 - Vector dot-multiplication time …………………………………………………. 17

# 

# Abstract

In this research we have proposed a technique to dynamically utilize the available processors in a heterogenous computing environment by assigning suitable workloads to relevant processor during runtime in order to attain a gain in overall processing time. By assigning during runtime, we tend to consider the current system loads. We are much constricted to the fact that the time taken to decide the appropriate processor should not be greater than the execution time of the task in any one of the processors. Hence the performance gain is achieved. Even though researches have been done in relevant field, mostly they require prior knowledge of the input and context of execution. But in this research, we will be adapting to the right processor during runtime which is unique. We have formed an algorithm which will analyse both the previous runs of the particular task and current behaviour of the processors to the tasks. Based on the output of the algorithm we will be switching the processing unit. Further we are not restricted to particular tasks or class of tasks. We formed the solution independent of the computational tasks.

# 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 have less latency and high throughput. It would make the effective speed of the servers faster.

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 expects 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.

# Problem Statement and Motivation

**3.1. 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.

**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.”*

**3.2. 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.

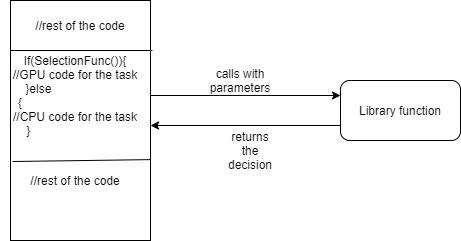
For real time, the high latency means growth of pending data to be processed and the incoming rate cannot be caught up which might result in total failure of a system depending how critical the real time processing is. For example, the financial Risk management system at LSEG Technology and there are 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.

# 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.

**

*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.

# Literature Review

**5.1. 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 will not 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 will not 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 up to 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 focused 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 are 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 allocations, and to store references of objects or pointers to the references [8].

**5.2. CPU vs GPU**

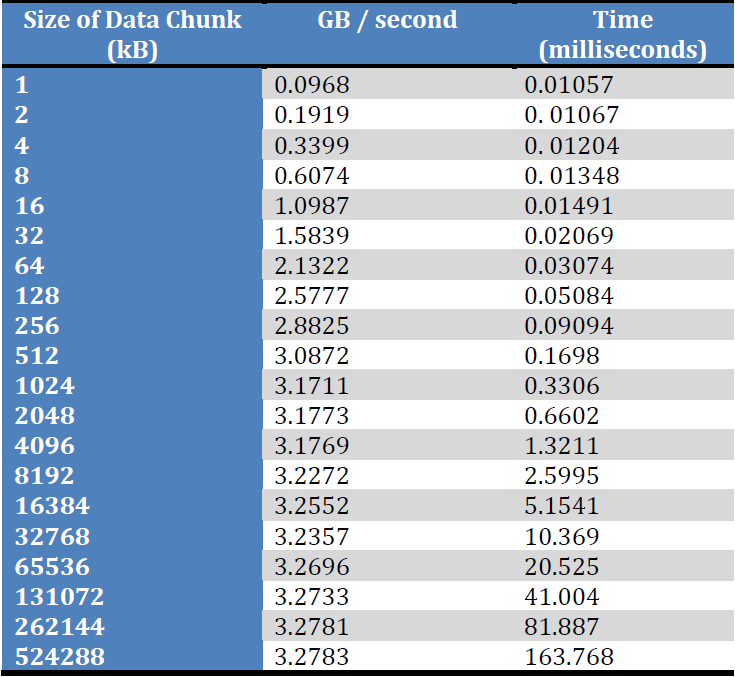
CPUs often have not more than a few tenths of cores and each core accepts an independent instruction set. A host can have one or more CPUs, but general hosts are built with single CPUs. The CPUs were single-precision (32-bit) previously, but all modern CPUs support double precision(64-bits) to do lengthy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

**

*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.

# Methodology

**6.1. Introduction**

We are focussed to develop a solution which is suitable for any type of computational problems that can be executed in a heterogenous environment. So, a generalized solution is expected that works for any kind of problem.

From the literature review, we were able to identify that the researches have been using three main solution patterns to solve the issue. They can be classified into 3 kinds and are,

1. Previous history-based model [14].
2. Running in all the available processors and the process that finished first will kill all other processes [17].
3. Load balancing models [19].

Based on our requirements which is that we need a generalized solution, we eliminated and refined some solution techniques as they are designed for particular computational models. Further we have designed an algorithm which consist of techniques from above literature survey and that we build considering various aspects of the solution.

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. This is where our novelty idea differs from other solutions.

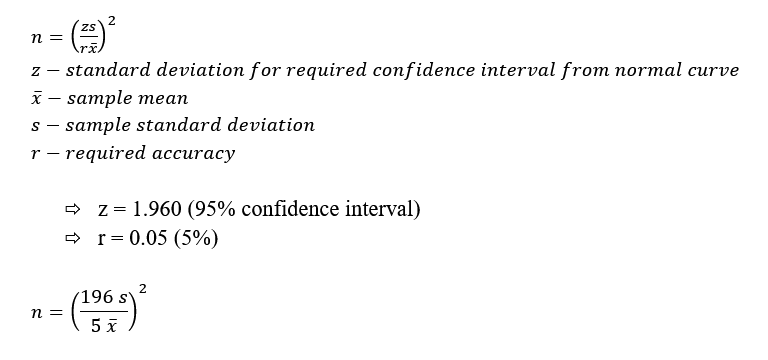
As explained in section 3, research objectives and outcomes, our proposed solution is a framework consisting of computational models evaluating tasks, they are defined for. The framework can be integrated with new computational models having custom implementations for the CPU and the GPU. This strategy can be scaled up for heterogeneous systems containing any number of accelerators in future. The optimal processor decision-making process is based on execution time and some attributes which impact the execution. e.g., the relative hardware capability, the GPU computation to data transfer gap. 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.

This research can be perceived in following manner to perform the experiments in an efficient way. We have classified the factors into 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.

It is obvious that other than the factors listed above, unidentified factors also impact on the execution time of the task. We have considered “raw execution time” analysis in order to include all the factors affecting the performance of a heterogenous environment. This analysis is not an optimal one as we have some subcases in above factors which we cannot predict using this analysis, but it will give us a platform to form our general solution. Further, in order to mitigate the effects of the factors that are not considered in raw execution time analysis, we have built other analysis techniques. For example, we have built a machine learning model to encounter the random flow of input data values.

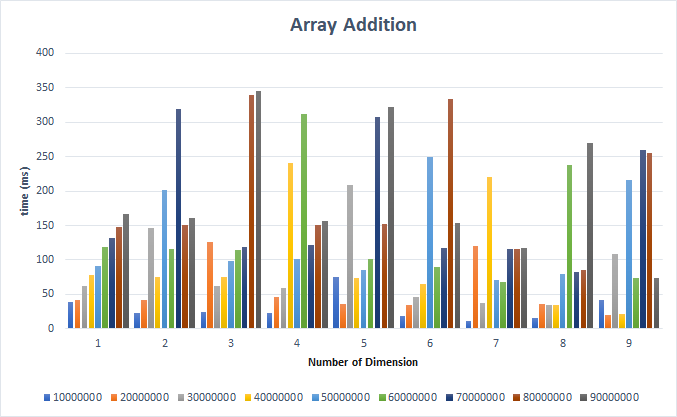
We have used following equation to conform that the accuracy and confidence level of the experiments met the required criteria shown below (95% confidence interval and 5% accuracy). The number of measurements needed to ensure the criteria was calculated and the experiments conducted for the number of times.



**6.2. Dive into context**

*6.2.1. Explore GPU*

We first did experiment only in GPU by changing factors influencing execution time to understand how an GPU behaves with the factors. The execution time decreased with number of dimensions since the GPU is good at parallel computing.



The above graph shows the performance of multi-dimensional array addition using GPU only. Let us consider A, B as two equal dimension and equal size arrays. The array addition operation will result in array C. The X - axis denotes the dimension of the input arrays A, B. Each set of colours in the X - axis denotes the size of the input arrays A, B. Y- axis denotes the time taken by the GPU to perform the addition operation. In dimension view, as dimension increases, the time taken by GPU to process the data decreases. In array size view, as size increases, the time taken by the GPU to process increases. But some exceptions were there.

*6.2.2. CPU and GPU execution curve analysis*

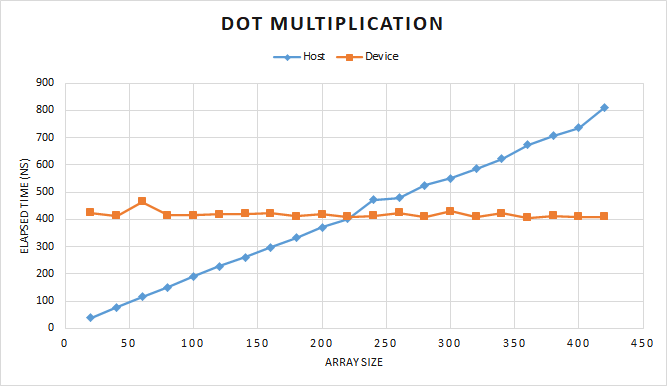
We experimented and calculated execution time for array addition and dot multiplication with varying array sizes on both CPU and GPU. Results from the experiments have been logged and shown below how execution time vary with size of input arrays on CPU and GPU in one graph for dot multiplication and elementwise array addition, respectively.

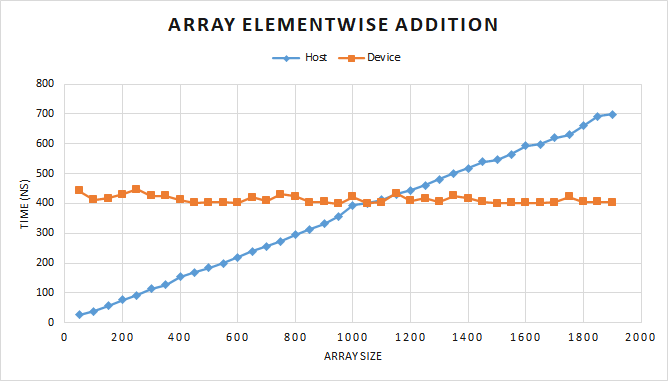
|  |  |  |  |
| --- | --- | --- | --- |
| **Array Size** | **Host Time** |  | **Device Time** |
| **100** | **0.00212042** |  | **0.00039633** |
| 200 | 0.00444981 |  | 0.00023719 |
| -- | -- |  | -- |
| 800 | 0.01605261 |  | 0.000225 |
| 900 | 0.0176719 |  | 0.00022228 |
| 1000 | 0.01917692 |  | 0.0002281 |
| 1100 | 0.02090968 |  | 0.00022673 |
| 1200 | 0.02668534 |  | 0.00022467 |
| 1300 | 0.02593127 |  | 0.00026113 |

|  |  |  |
| --- | --- | --- |
| **Array Size** | **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 1: Elementwise array addition Table 2: Dot multiplication*

Initial solution for this problem was static benchmark. We used standard problems, 100 in this case (highlighted in tables above) to overcome the different hardware capabilities issue. The standard problem will be executed on both CPU and the GPU initially. The relative benchmarks are updated by comparing time elapsed for it. 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 elapsed 100ns for the standard computation execution will be, 1100 as per the graph, Array Elementwise Addition below. It means that functions called with the argument value less than 1100 will be returned with true (CPU) and false (GPU) otherwise for the period.

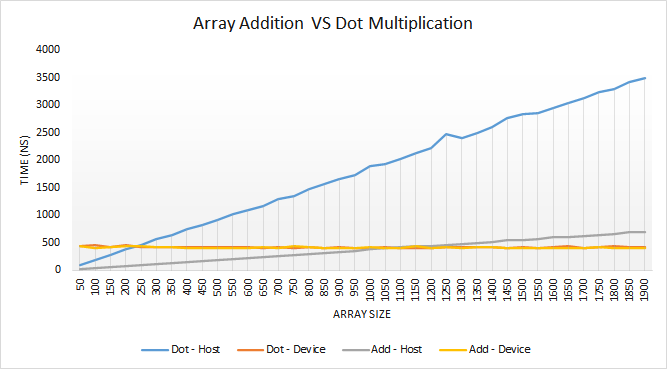




Graph 5.1: Vector addition execution time

The above graphs show the time taken by the host-CPU and device-GPU against varying array time for the array addition and dot multiplication. Both the graphs look similar. We can see that for small array size CPU outperforms better than GPU. As the size increases, the GPU performs better than CPU. This is because, initially for small sizes the load to CPU is low so it can execute the task quickly so that it is even smaller than the time required for the data transfer from CPU memory to GPU memory. But as size increases, the load to CPU increases. Hence it takes high time for execution which is even higher than the data transfer time plus the execution time of the GPU. But as with GPU with many cores, it is able to execute the operations easily even for high sizes.

The dataset from both computational models have been plotted in one axis below to compare how complexity of a problem impact execution time on host and device.



CPU to cross the boundary happens soon when the computation become more computationally intensive. The GPU time roughly not vary with the size. The CPU time increasing with the array size, but cross over the GPU time at a point, which meant to be the boundary point here.

After experimented in multiple machines we come to an idea that, we cannot define static benchmark for mapping a task to processing units. We went through several hardware related research papers and we drop the idea of doing hardware-based experiments since it is irrelevant to our problem and proposed solution.

**6.3. Formation and Development of the Main Algorithm**

A hardware independent framework implemented in C++ contains 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.

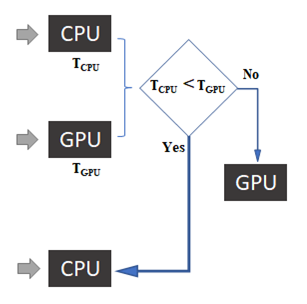
Programmers will create an object of any of the available model or create a model for his own needs. Each time he wants to use the model, a programmer will set the data and invoke execute() method on the object. The execute() method will invoke the best from two functions implemented for CPU and GPU implicitly. He can manually set to a fixed processing unit for a problem by giving the unit id to the execute method. e.g., execute(1); for CPU. Manual mode will not be evaluated, and it is there for the code reusability policy.

We have followed various approaches.

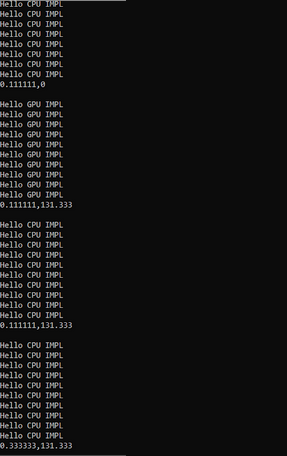
* Last N Average Time Comparison
* Sampling Execution Time Algorithm
* Sampling Algorithm with Varying Revise Count
* Machine Learning (ML)5.3.1
* Hybrid of Sampling Algorithm and ML

*6.3.1. Last N Average Time Comparison*

The algorithm works in the way such that initially two disjoint consecutive batch of data will be given as the input for both CPU and GPU from the input stream. The timer will measure the time taken when executing CPU and GPU to process the particular size of input. It will be logged in a text file in the way "CPU" along with its execution time and "GPU" along with its execution time. The execution time averages are kept in memory and compared. The algorithm will let the upcoming input data to the processor which would take less time. Similarly, for every consecutive batches, the last CPU average execution time and the last GPU average execution time will be compared, and the next input set will be directed to the processor that had less average time. For every comparison, the time taken in the previous runs on each processor will be considered.



This algorithm has many falls, identified later. If at a particular case due to the influence of external factors such as system loads, the average execution time may become unexpectedly high, then we will not be able to execute any other batch of data in CPU again. Because since the average execution time is extremely high for CPU, this algorithm will always prefer GPU. It means the averages values will not be reduced at all once it has increased since execution time has raised to high value due to some unexpected issues. Hence, there may be instances one processor sitting idle. Therefore, the other accelerator continues the execution forever which makes this approach inefficient. We came up with the following “Sample execution time algorithm” to overcome this issue.



*6.3.2. Sampling Execution Time Algorithm*

The algorithm works in a way such that initially two disjoint consecutive sets of problems will be given as input for both CPU and GPU from the input stream. The size of the consecutive input data given is known as "SAMPLE\_COUNT". The timer will measure the average time taken to execute on CPU and the GPU to process the particular input data. Then it will compare both execution times and a third set of input data of a number called “REVISE\_COUNT” will be directed to the processor which took less time. This process repeats after execution of revise count number of problems.

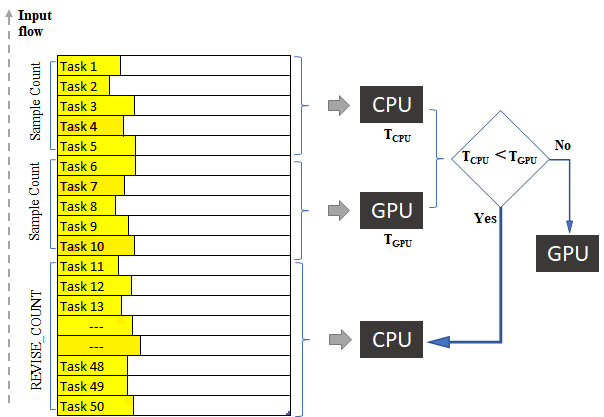
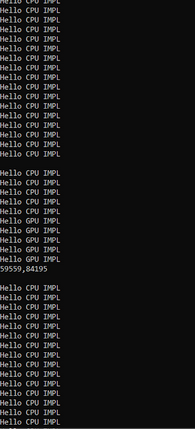


Figure 8.1 - Execution Mechanism



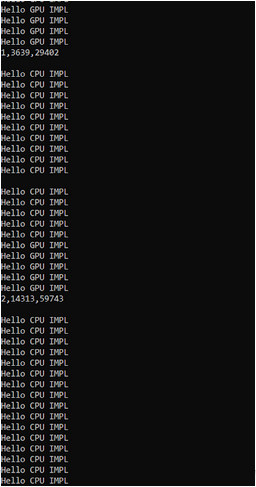
***Lacks in this approach:***

1. It is hard to select the most appropriate REVISE\_COUNT value that matches all kind of stream.
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.

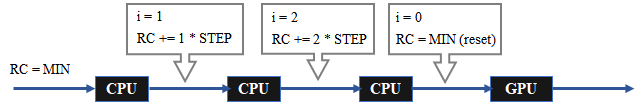
The main issue is that we need to keep comparing the execution time continuously at a particular time interval, even though the input is favourable to either CPU or GPU for a long time. It has been solved with a varying revise count int the following algorithm.

*6.3.3. Sampling Algorithm with Varying Revise Count*

This algorithm is similar to the above algorithm. But the size of the third set of input data after the sampling will be a constant "REVISE\_COUNT\_MIN" initially. If a processor wins continuously, revise count value will be incremented by a value, "REVISE\_COUNT\_STEP" up to a certain limit, "REVISE\_COUNT\_MAX". The revise count value will be between "REVISE\_COUNT\_MIN" and "REVISE\_COUNT\_MAX". If the consecutive execution chain is broken, the revise count will be reset to a minimum value. This process will repeat forever.



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



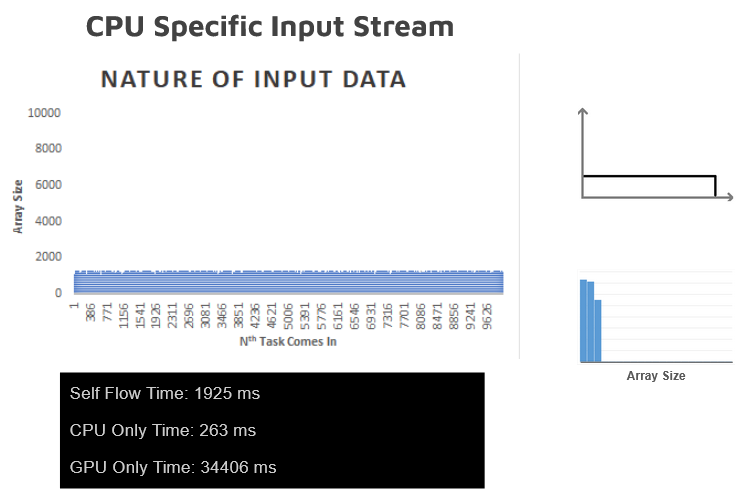
*Figure 5.2: REVISE COUNT increment method*

**6.4. Experiments with the Algorithm**

The intermediate system with sampling algo with varying revise count was experimented with 5 kind of input having different natures. The streams were generated automatically before they were given to the algorithm and evaluated in three cased, only on CPU, only on GPU and self-flow (the sampling algorithm) modes. The results were logged, and graph plotted in excel using the logged data. The experiment was conducted with our experiment model at that time, element wise vector addition of two equal size arrays.

Primary graphs illustrating the nature of the input stream. Y (vertical) axis of the graph shows the length of arrays coming in the stream. X (horizontal) axis shows the order of input stream flow. The subgraphs on right shows square wave of the same graph and the overall distribution of the array sizes given for the experiment, respectively.

*6.4.1. CPU Specific Input Stream*

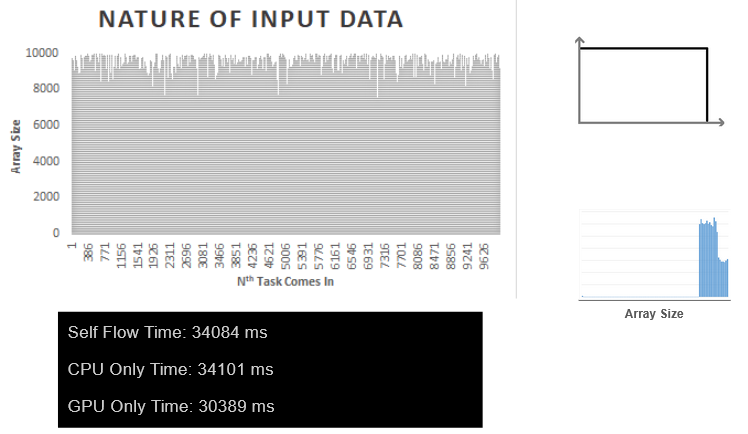


All inputs in this stream are less complex and small, and hence as its name suggests, are good to be executed on CPU. GPU may take more time for execution of the inputs than the CPU because of the initialization setup costs of the computation.

**Observation:**

Since the stream is CPU specific, CPU only time was very low, and GPU took long time for execution. In self-flow mode, it took a time that was very close the CPU only execution time. It must be noted that, in typical programs, the computations in the streams that have been experimented here are executed on the GPU by default. Therefore, it could be considered as a huge gain over the default execution setup.

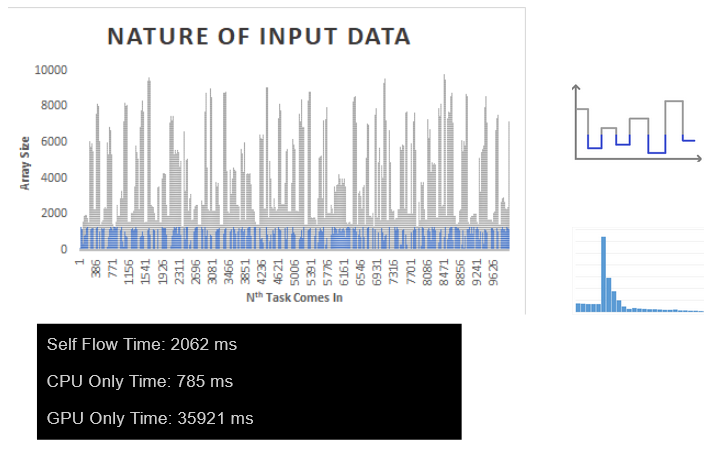
*6.4.2. GPU Specific Input Stream*



In this stream, problems are large and optimal to be executed in GPU and run them on CPU will be a counterpart and cause severe latency. The distribution graph on right below describes the nature.

**Observation:** Self-flow time has exceeded the GPU only time, which was not a gain at this time. because that some samples have been scheduled in CPU for evaluations and got delayed. It can be optimized by preventing the Highly GPU specific tasks executed in the CPU at all. We improved the algorithm and considering some attributes of the computations impact the execution to solve this issue.

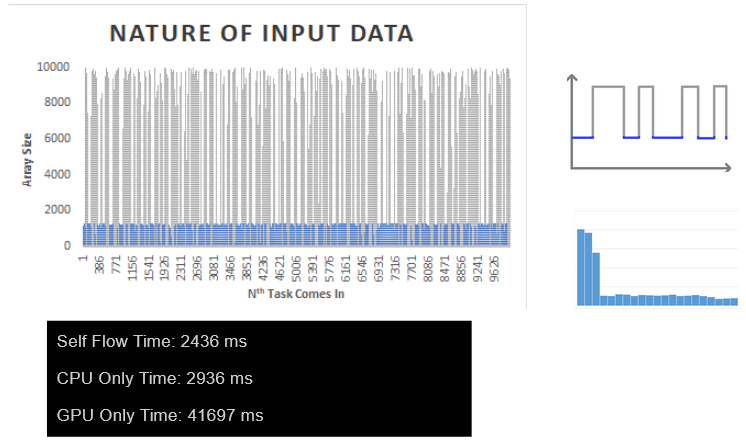
*6.4.3. Square Aligned Wave Input Stream*



The blue lines in the graphs indicate input clusters favourable for the CPU and the ash lines indicate input clusters favourable for the GPU. Inputs are coming in clusters, alternatingly favourable for CPU and GPU. But the widths of the cluster are random.

**Observation:** This square wave was generated randomly but favorable for CPU as it shows less time in CPU only mode. Anyway, time from self-flow mode is also close to the CPU only time. Therefore, we can assume this algorithm retains our goal.

*6.4.4. Binary Aligned Input Stream*

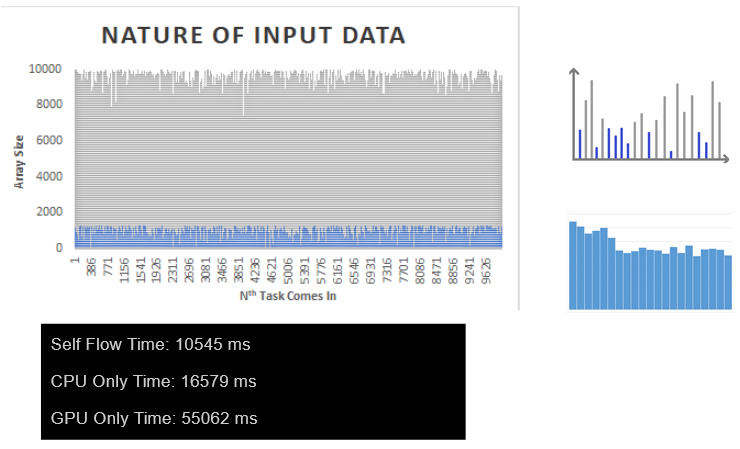


In this case, the clusters alternating between two maximum values waves as you can see in the square wave on the right.

**Observation:** Here we have got the Self Flow Time less than the CPU and the GPU which was a highly success point. This might because both CPU and the GPU have been utilized.

The observation shows that the algorithm sensitive to the binary edges and executing them in the appropriate processing units.

*6.4.5. Odd Input Stream*



There will be no cluster in this input stream. The sizes are alternating zig zags between two fixed values set.

**Observation:** In this case also the self-flow time achieved a great gain, less than that of CPU and GPU which is an unexpected scenario. It might be because of the throughput from executing the input stream on both processing units in random paths taken by the algo. The REVISE COUNT value would not be incremented in this case and would be stayed as the minimum always.

Most of the time in our experiments the self-flow times are closer to the smaller value but some time less than both CPU only and GPU only times, which can be considered as high successes of this research.

**Lack in this approach:**

* This algorithm cannot tackle with all kind of the nature of the input data. It cannot handle pure odd input stream, means the noisy input stream.
* It is unable to detect such anomaly cases since it does not consider the weights of the computation.

It will cause a severe latency if a very big computation comes when the CPU is in operation. We wanted to consider some attributes defining the input problems to overcome this issue. We have tried Machine Learning approach for that.

**6.5. Outlier Issues**

One of the major challenges we faced during this research was to hope with different patterns of inputs. They play a major role in the performance of our prediction system. For example, if there is a sudden surge in input size among small inputs, it will be sent it to the wrong processor. This would degrade performance of 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 effects related to this outlier issue.

ML model alone can predict the best processor for execution. But the issue is that this model makes prediction based on only the features. Current system context will not be taken into consideration. Further frequently engaging ML model for predicting would incur additional time in prediction process which would affect our performance.

**6.6. Massive changes in Research Context**

This point onwards, the research model shifted from dot multiplication model to matrix multiplication model for further experiments. The experiments conducted with the dot multiplication model showed the good performance for most of the input stream types. But the presupposition was that the dot matrix multiplication model is too simple, and it would not yield the same as complex models.

This research will not be rationale if it we compare results from a single core of CPU and the GPU. Also, parallel execution on CPU was very important since this primary concern of this research is server systems and typical servers have tens of cores and is significant. We have used Open MP threads since we have used windows system for experiments. The constructor of the “ComputationalModel” class has been overloaded with an argument, number of cores to create that number of threads in CPU to on the Open MP (OMP) to utilize the all-CPU cores.

**6.7. Logger Class Integration.**

A logger module has been integrated to log the performance matrix to analyse performance of the system as per the project requirement. The data would also be useful to fine tune the configurations of the system and to tune the machine learning hyper parameters more appropriate. It is logged in the CSV format. The performance data for each model is logged massively at beginning including the time for execution. But limited to some required data after some time to avoid overhead of logging. The log files generated for the machine learning contain the attributes estimate a computation followed by the decision offered by the algorithm in each line.

**6.8. Machine Learning Approach**

We had the outlier issue and to solve that we were having two approaches to solve the issue. They were the statistical estimation approach and the input weights-based ML approach. we decided to use machine learning approach for following reasons,

* Tasks and number of attributes impact execution of the tasks are not known in advance.
* Dataset can be summarized using ML model with less human interventions.
* The relationship between the attributes and execution time does not converge to a general equation for all problem types.

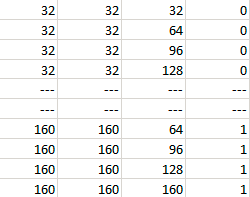
XgBoost machine learning algorithm, a pre-existing algorithm was used here since it has less prediction time compared to other algorithms. The datasets are loaded from csv files created by the logger. The model will have attributes of the particular input stream. The characteristics of input data can be accessed using an abstract method getAttributes() that must be implemented by the developer.

Dataset to train the model was created by executing tasks on both processors with varying attributes and logged the execution times against each processor for every task. Then optimal processor for each task for certain attributes was logged against the task. The model will be trained automatically after certain time of period. Therefore, any changes in machine hardware and software will not affect the correctness of its predictions. Certain constants like the retrain period have to be configured for an application using this library when it is deployed to a system.

XgBoost algorithm combines multiple decision trees to establish strong learner. ML algorithms parameters must be configured appropriately to create good model from the dataset given. The depth of the trees is a parameter that decides accurateness of the predictions from the algorithm. The depth value is estimated based on the number of parameters and size of dataset given for training. We have used 6 for this research since predictions can be tentative enough to catch outliers but it must be configurable for each custom model.

Algorithm was tested with only Machine Learning (ML) to train better and check whether using only this approach would help to achieve our goals without going into the sampling algorithm. But despite the sampling algorithm, every input data in input stream needs to be evaluated with this ML model. After several experiments went on with this approach, it was found that a trained XgBoost model takes around 1 millisecond for predictions. The prediction time can be reduced if we predict for multiple input data at once or scan and evaluating the input stream that received in a buffer using a separate thread. This was not experimented since our present architectural structure and implementation would not support for this. This part is left for fellow researchers.

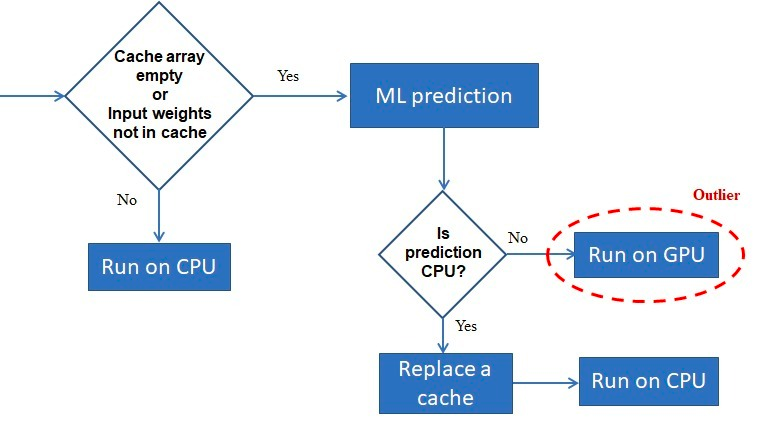
*6.8.1. ML Dataset*



A dataset that contains labels for some normally distributed input weights was needed to train an ml model. The dataset could not be created automatically since the decision involved both processors execution times. Therefore, a set of problems must be executed on both processors to obtain such a dataset. An input stream containing all possible combinations of the weights within a range has been generated. The step used between each attribute value was 32. The generated stream consisted of inputs data from 32x32x32 to 160x160x160. All values in the range were covered in the input stream. The labels were given for the dataset by executing the input on both processors and comparing corresponding execution times.

*6.8.2. Caching Mechanism*

Since the prediction time is high compared to the gain received from the optimality selection of processors, the results from ML only experiments were worse consuming more time compared to the times from executions purely in single processors. So, to minimize the overhead in the ML prediction time we implemented a caching mechanism which minimize the number of predictions by underlying ML model.



As in the flow chart above the system will check for attributes of given task in cache. At the start of execution cache will be empty. So, at the beginning ML model will be used to predict the optimal processor for the task. If the prediction result is GPU, the task will be executed in GPU but if the prediction is CPU the appropriate attribute set will be cached in cache and task will be executed in CPU. For upcoming tasks there will be some set of attributes in the cache. So, it will compare the attributes of current task with attributes in the cache. If same attributes found in cache or if every single unit of current attribute less than or equal to every single unit of attribute in the cache task will be executed in the CPU. Otherwise, it will use ML model to predict. If the prediction is CPU then the cache will be replaced with current attribute.

**6.9. Machine Learning Model Integration**

*6.9.1. Hybrid Model*

Sole machine learning model will not consider the current workload of the system. So, we integrated the ML model with the sampling algorithm. The basic sampling algorithm with varying revised count used along with the ML prediction approach to catch any outlier being executed in CPU. Though, every prediction needed to be evaluated before it was scheduled to a CPU. It recouped the gain that earned from the sampling algorithm since the prediction time close to the computation time of the matrix multiplication experimental model. Therefore, we came up with a caching mechanism to mitigate the problem, integrated to the system which caches some predictions that bounded above to reduce prediction time. It would store last few predictions of the high valued weights. The number of predictions to be cached is configurable as per our needs. But it was five throughout the experiment conducted, which is considered as sufficient for our experimental model.

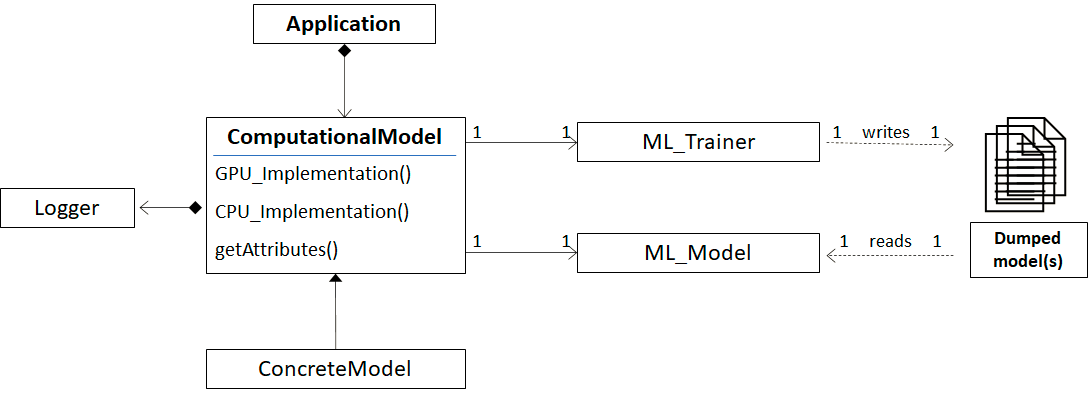


Figure . - Architecture of the final hybrid algorithm

Such worse scenarios would cause severe latency and may cause the program to shut down if CPU is the case. To solve this, the computations were evaluated using the trained ML model before sent through the sampling algorithm. If it detects a computation as an outlier, better to be executed in the other processor, it will switch processor immediately just for the problem and the algorithm continues as normal. It prevents the algorithm from the outlier trap.

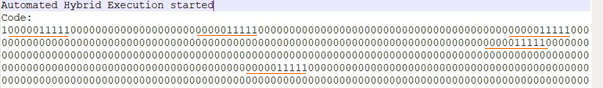
It would be beneficial to switch to GPU for the next batch if larger problems are detected consequently arriving. A counter in the Computational Model abstract class is used to swich to GPU if a certain number, “MAX\_OUTLIERS\_LIMIT” of larger problems coming consequently. And the counter is reset if any abnormality in the middle. This mechanism would help to detect next GPU line on the stream externally from the sampling algorithm and set the processor accordingly.

First, weight of every computation is compared with the predictions stored in the array iteratively. If all the attribute values in the weight is less than any prediction in the cached array, it means the problem is bounded by the exceeding weight and the computation better takes the CPU execution, that is no need to switch. If none of the prediction weight stored in the cached array bound the weight of the new problem, it is sent for the prediction to the ML model and the action upon the output. It would take little more time compared to the cache comparison. The computation will be executed in the GPU if it is predicted as an outlier but the platform that has been set by the sampling algorithm for the execution will not be changed. The cache will only be updated if new prediction is CPU. The new cache will be added into the cache array or compared and replaced with an existing one if it is full.

In the self-flow mode, where execution flow is based on the sampling algorithm, there are two places where a computation is assigned to the CPU. One is when sampling mode of the processor and the other is when execution mode of the processor. Therefore, to handle outliers in both places, the caching and outlier prediction mechanism are extracted into a separate method and a Boolean returned in place.

**6.10. Logging and Evaluation of Hybrid Model**

We have evaluated the hybrid model with 5 types of input stream. Hybrid model decisions have been logged as 0’s and 1’s. The 0 corresponds to CPU and the 1 corresponds to GPU. The below image shows an example snapshot of an execution flow taken by the hybrid algorithm for CPU input stream. The corresponding graph is also given below the image.



The highlighted areas mean to the sampling phase of the algorithm. Successive consequent zeros show the next that number of executions were on CPU as per the hybrid model decisions after the sampling phases. Since it is a CPU specific stream the decisions were correct, and the algorithm worked as expected. The revise count value was increased incrementally as CPU wins continuously. It would have reduced the sampling overhead.

*6.10.1. CPU Specific Input Stream*

|  |  |
| --- | --- |
| CPU only: | 577.8 ms |
| GPU only: | 2586.8 ms |
| Self-flow: | 613 ms |
| ML only: | 600.8 ms |
| Hybrid model: | 623.6 ms |

On the graph, x axis shows input stream flow, and the y axis shows number of operations in the inputs. The thin line shows hybrid model decisions, low and high. High denotes the GPU execution and low denotes the CPU execution. The surge peaks along the thin line mean to the sampling phase, the 5 samples executed on the GPU. The thick line indicates the flow of the input stream. The low bottom is sizes are preferable to execute in CPU while the high flat regions are preferrable for GPU. We can see that the arrays with large sizes are being executed in the GPU. The algorithm perfectly predicted and executed on the optimal processor other than the sampling executions on the GPU. For this input stream, the gain was noticeable. Though the gain was less in time which is around 2 seconds but very high if we consider it in percentage. The performance has been increased by 314.81% which is very significant.

*6.10.2. GPU Specific Stream*

|  |  |
| --- | --- |
| CPU only: | 184447.6 ms |
| GPU only: | 22167 ms |
| Self-flow: | 24035 ms |
| ML only: | 22161.8 ms |
| Hybrid model: | 22199.8 ms |

This stream entirely contains larger inputs, it is obvious we cannot achieve a gain in this case. nevertheless, we have utilized both processors parallelly. There were no peaks in this graph and no sampling taken place because the outliers are caught and sent to GPU. Hence no sampling phase and CPU executions were there for this stream at all. But, if continuously large problems were coming, it would have switched to GPU for next batches after it has caught a few outliers. There was a little overhead due to the algorithm and it was -0.15% and it is negligible.

*6.10.3. GPU Specific Stream*

|  |  |
| --- | --- |
| CPU only: | 64144.8 ms |
| GPU only: | 9432.4 ms |
| Self-flow: | 45824 ms |
| ML only: | 8061.2 ms |
| Hybrid model: | 8332.4 ms |

Here the input flow has two sizes of array. One size being suitable for GPU and the other being suitable for CPU. Further the flow’s frequency of fluctuation between these two array sizes is medium level.

*6.10.4. Square Wave Stream*

|  |  |
| --- | --- |
| CPU only: | 32303.4 ms |
| GPU only: | 7694.2 ms |
| Self-flow: | 8508 ms |
| ML only: | 8023.4 ms |
| Hybrid model: | 7681.4 ms |

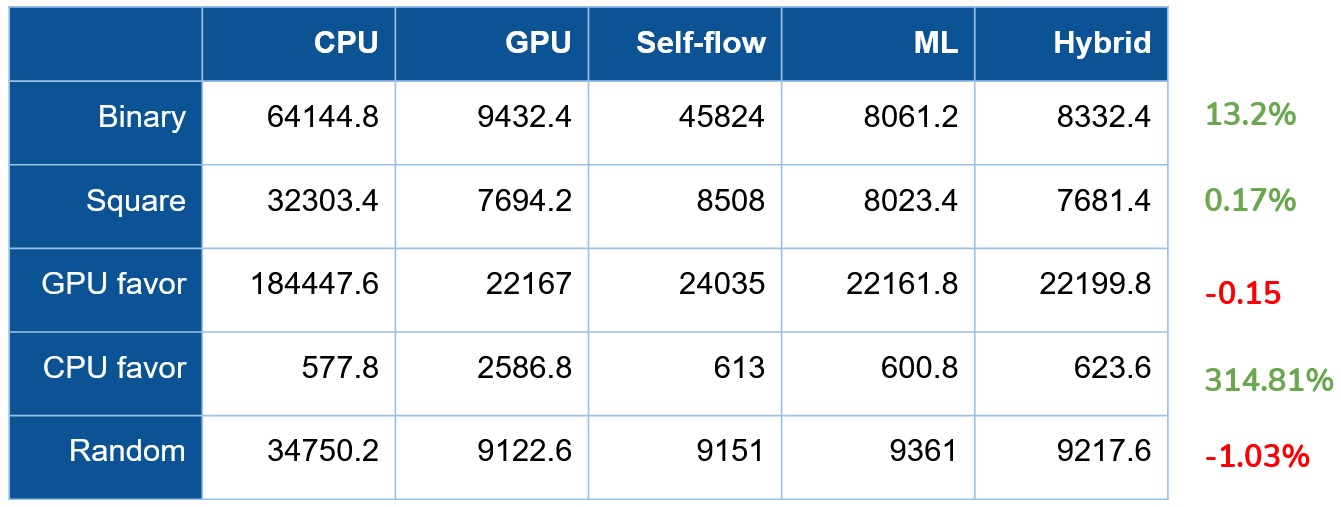
In this stream, the inputs come in clusters of random width and height. Clusters lying very below are only suits for a CPU and all others are good to be executed on a GPU. When the cluster values are very low in y axis, the decision by thin line was also down and it was high otherwise. It shows that algorithm worked as expected. The sudden peaks correspond to the sampling phase of GPU. The gain is very negligible for this stream. Because it had very less likelihood of CPU favor problems coming in the stream. The gain was 0.17% for above stream. It would be high in productions systems where their CPU contains tens and hundreds of cores.

*6.10.5. Odd input Stream*

|  |  |
| --- | --- |
| CPU only: | 34750.2 ms |
| GPU only: | 9122.6 ms |
| Self-flow: | 9151 ms |
| ML only: | 9361 ms |
| Hybrid model: | 9217.6 ms |

This stream consists of no cluster. Input sizes are all random and alternating zig zag. We cannot look for a gain in this case also. Because the stream has less likely hood to have small problems comes in cluster that fits CPU. This time we had overhead instead of gain. The drain due to algorithm processing was -1.03% in percentage and it can be ignored and gain from other streams would compensate this loss.

**6.11. Elapsed Time Averages**

****

**6.12. Final Proposed Solution for Research Problem**

As a result of the long refrainment of initial algorithm we were able to address the problem of dynamically utilizing the available processors in a heterogenous computing environment by assigning suitable workloads to relevant processor during runtime in order to attain a gain in overall processing time. Our final solution model is supported by the statistical and machine learning techniques. We tried to build the solution based on only statistical based and only machine learning based. Only statically based solution did not address the problem completely. Because it could not solve the outlier issue. And the only machine learning solution also did not perform well because the time taken by the model to make prediction is high. So, we were able to form a hybrid model which took advantage of both techniques and addresses those issues arose when the techniques were used alone.

# Conclusion

The research is to analyse 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 library that contains a set of functions to be used by programmers which will assist automatically in choosing between CPU and GPU with only getting minimum configuration details from the programmer. 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. We too have used machine learning in order to solve the issues that we couldn’t solve using only statistical model. The library is created for general computational model class which is not specific towards any particular computational model. 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 computing into another dimension and provide a boost in the GPGPU to a great extent.

Further as stated previously in the Methodology section, we have divided the factors affecting the decision making of selecting the appropriate processing unit under three main headings.

1. Computational task

2. Nature of the input stream

3. Hardware

All the above factors will have impact in the execution time. Since our evaluation is based on the low execution time, we chose execution time-based analysis which statistically analysis the execution time which could cover all the aspects. But this will not be best as it did not particularly dive deep into the analysis of any of these factors. But the machine learning part that we used in order to solve the issues that we couldn’t solve using statistical analysis actually analyses the attributes of the particular computational problem. In the case of matrix multiplication, it will consider the dimension of the matrixes. So, in this particular research we were able to address.

# Fellow Research Possibilities

Currently **we** are using a single thread to make prediction using machine learning model. But we hope we could extend prediction into multithreaded CPU task in order to make predictions faster so we could save time delay due to prediction which could enhance the performance.

Further, we could improve the performance by increasing the capability of the CPU. Because we could improve the performance by saving the time which take to transfer the data between the CPU memory and GPU memory. In order to improve the CPU performance, we could overclock the CPU. Hence the performance will be increased. Or we could disable hyperthreading which sacrifices the single core performance to get multi-tasking. By disabling hyper threading, we could improve the clock speed of a single core in the CPU.

# 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](https://uomcse-my.sharepoint.com/personal/jayakeethan_16_uomcse_lk/Documents/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](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](https://uomcse-my.sharepoint.com/personal/jayakeethan_16_uomcse_lk/Documents/https)[://link.springer.com/chapter/10.1007/978-3-642-31128-4\_34](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](http://computing)

[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] M. Kicherer and W. Karl, “Automatic task mapping and heterogeneity-aware fault tolerance: The benefits for runtime optimization and application development”, Journal of Systems Architecture - Embedded Systems Design, 2015. [online] [Accessed 29 March 2020].

<https://dl.acm.org/doi/10.1016/j.sysarc.2015.10.001>

[16] A. Chikin, J. Amaral, K. Ali and E. Tiotto, Toward an Analytical Performance Model to Select between GPU and CPU Execution, 2019. Available: https://ieeexplore.ieee.org/document/8778216. [Accessed 30 August 2020].

<https://doi.org/10.1016/j.sysarc.2015.10.001>

[17] J. Dollinger and V. Loechner, "Adaptive Runtime Selection for GPU," 2013 42nd International Conference on Parallel Processing, Lyon, 2013, pp. 70-79, doi: 10.1109/ICPP.2013.16. [online] [Accessed 29 March 2020].

<https://ieeexplore.ieee.org/document/6687340>

[18] C. Augonnet, S. Thibault, R. Namyst, PA., Wacrenier, “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”, 2009. Lecture Notes in Computer Science, vol 5704. Springer, Berlin, Heidelberg. [online] [Accessed 29 March 2020].

<https://doi.org/10.1007/978-3-642-03869-3_80>

[19] J. Shen, A. L. Varbanescu, Y. Lu, P. Zou and H. Sips, "Workload Partitioning for Accelerating Applications on Heterogeneous Platforms," in IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 9, pp. 2766-2780, 1 Sept. 2016, doi: 10.1109/TPDS.2015.2509972. [online] [Accessed 29 March 2020].

<https://ieeexplore.ieee.org/document/7360199>

[20] M. P. Robson, R. Buch and L. V. Kale, "Runtime Coordinated Heterogeneous Tasks in Charm++," 2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2), Salt Lake City, UT, 2016, pp. 40-43, doi: 10.1109/ESPM2.2016.011. [online] [Accessed 29 March 2020].

<https://ieeexplore.ieee.org/document/7831559>