Self-accelerating Processing Workflows

B.Abinayan,
Undergraduate,
Department of Computer Scinece and Enginnering,
University of Moratuwa.

J.Jeyakeethan,
Undergraduate,
Department of Computer Scinece and Enginnering,
University of Moratuwa.

T.Nirojan,
Undergraduate,
Department of Computer Scinece and Enginnering,
University of Moratuwa.

***Abstract* — This paper proposes a solution to dynamically utilize relevant processors in a real-time heterogeneous system to improve its overall performance. The solution is a hybrid self-flow algorithm, combination of statistical time and input weights (ML) based algorithms. The algorithm would yield performance gain over its processing time by utilizing CPU appropriately. Most related researches require only the prior knowledge of the inputs and its context of execution. This research has considered current loads in processors when the inputs are given such as contention, peak and off-peak hours etc. The algorithm takes decisions based on both the previous runs of a task and weights of new inputs of the task. The weights are evaluated using machine learning. A caching strategy used to decrease prediction times. The gain achieved in the experiments on personal laptops are satisfiable and it will be even more in production systems.**

Keywords — GPU programming, Heterogenous systems, Load balancing.

I. INTRODUCTION

Compute intensive tasks are typically programmed for GPUs if they are parallelizable. Powerful CPU of servers containing 10s of cores may be sitting idle while the GPU is suffering from contention [18]. The CPU can process the tasks faster up to some size complexity limits of the tasks which would save time and improve overall performance. The limits for tasks are varying and depending on various factors such as complexity of the tasks, hardware accelerators, deployment environment, time of the day and system's current state, e.g., contention [14][19]. Therefore, optimal processing units for inputs of the tasks cannot not be predetermined during the programming period and it varies.

Moreover, inappropriate scheduling of the computations into wrong processors is inefficient and time consuming [17]. It may also freeze the CPU and normal flow of the program will be affected since the CPU is responsible for processing main flow of the program. But directing them into appropriate processing units reduces the overall execution times and improves overall system performance. Applications specially written for CPU and GPU are switched manually for optimization purposes. The above CPU and GPU modes both may have high overall execution time since request stream in the practical applications is a mixture of inputs that are suitable for both processors.

II. 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."

III. MOTIVATION

CPU has less latency for light inputs of compute intensive tasks that were run on GPUs. Extracting and executing such inputs on the CPU from GPUs workloads may increase throughput and also reduce GPU's contention. It also prevents starvation of computation requests in some instances. It is able to provide high quality services with low end hardwares as it utilizes the CPU also. The branch predictor in CPU architecture was another successful implementation being a motivation for this research.

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.

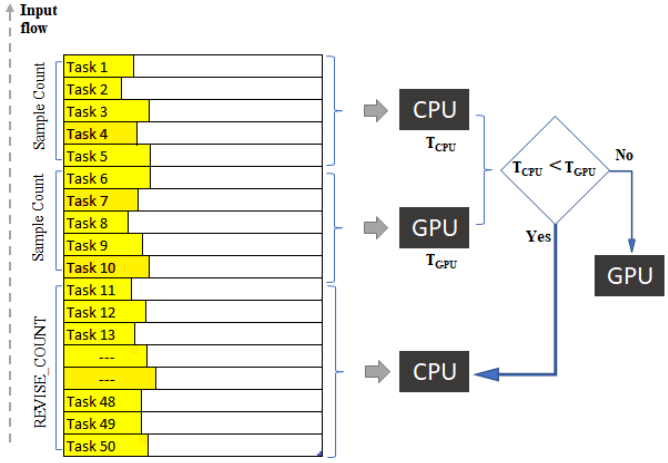
IV. METHODOLOGY

Our solution is based on statistical analysis and machine learning analysis. Hence our algorithm is a combination of a statistical model and a machine learning model.

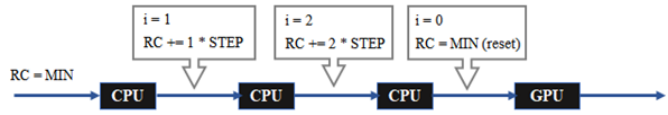
In this section for the evaluation purpose, we are considering the Matrix multiplication as the computational model to test our algorithm. Since we are considering the matrix multiplication, there will be two matrices. In order to perform matrix multiplication between two matrices, the row of A must be equal to the column of the B. Let them be A with dimension (x, y) and B with dimension (y, z). Hence the input stream will contain sets of (A, B) pairs.

A. *Statistical Model*

The algorithm works in a way such that initially two disjoint consecutive sets of data 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 size of a number called “REVISE_COUNT” will be directed to the processor which took less time to execute the previous two data sets. This process repeats after execution of “revise count” number of problems.



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 during consecutive sampling processes, revise count value is 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 revised count will be reset to a minimum value which is “REVISE_COUNT_MIN”. This process will repeat forever.



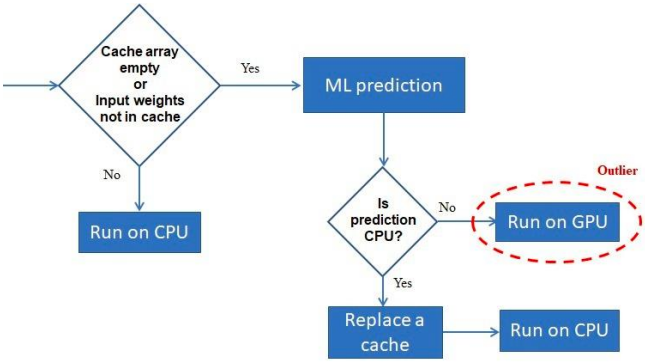
B. *Machine Learning Model*

If a large problem is sent to the CPU, it may freeze the program. It created a need to account weights of the inputs. There were two approaches; statistical weight estimation and using an ML model to estimate the decision. ML model was selected from those two though some papers oppose using ML for this purpose. Because it has prediction overheads, and such solutions are often limited in production systems.

Every input data stream is 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.

A pre-existing algorithm, XgBoost was used since it has less prediction time compared to other algorithms. XgBoost internally uses decision trees to create models for training dataset of time complexity lg(n) yet prediction time could not be negligible. Hence, we introduced caching mechanism. The characteristics of input data can be accessed from the model object using an abstract method getAttributes() that must be implemented by the developer.

Labeled dataset is inevitable when it comes to ML training. The datasets are loaded from csv files created by the logger. The dataset had to be generated manually with a normally distributed input stream executed on both processors and comparing their execution time. Anyhow, the ML decisions are just approximations to avoid undesirable situations.



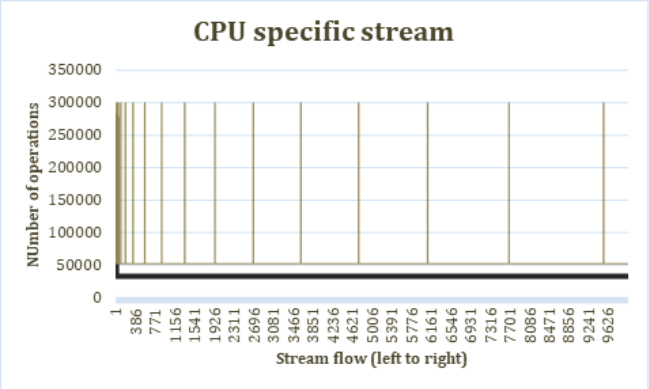
As in the flow chart above the system will check for attributes of given task in cache. At the start of execution cache will be empty. So, at the beginning ML model will be used to predict the optimal processor for the task. If the prediction result is GPU, the task will be executed in GPU but if the 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. Therefore, it will compare the attributes of current task with attributes in the cache. If all attributes of the input being evaluated are less than or equal to corresponding attributes of any of record in the cache, the 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.

V. EXPERIMENTAL RESULTS

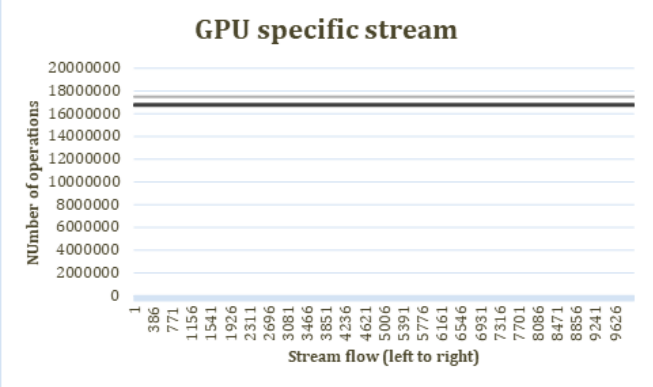
The solution must be tested with some different continuously arriving input stream to confirm that it works properly. It must yield less overall execution time than bare execution in GPU. The hybrid model has been tested with five kinds of input streams, covering edge cases of the research problem. They are CPU favor input stream (less complex inputs), GPU favor input stream (large inputs), binary input stream, square wave input stream and the random input stream with which CPU would have less chances for utilization.

Hybrid model decisions were logged as 0’s for CPU and 1’s for GPU. The logged data was plotted as graphs to make analysis easy. The graphs showed that the decisions taken by the model were almost correct. The hybrid model has been five times experimented with such randomly generated five input streams and results were averaged. That verified the

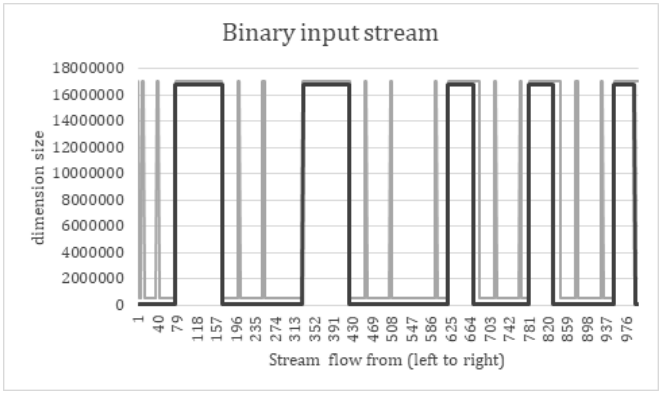
algorithm flow and showed achievable gain using the algorithm in optimal and yield drains during worse cases. For example, a GPU favor input streams cannot utilize the CPU at all and therefore, there would be little overhead but not the gain. In the graphs, X axis shows the inputs flow, and the y axis shows complexity in terms of the number of operations. The thin line shows hybrid model decisions, low and high. Low denotes the CPU execution high denotes the GPU execution. The thick line indicates the flow of the input stream.



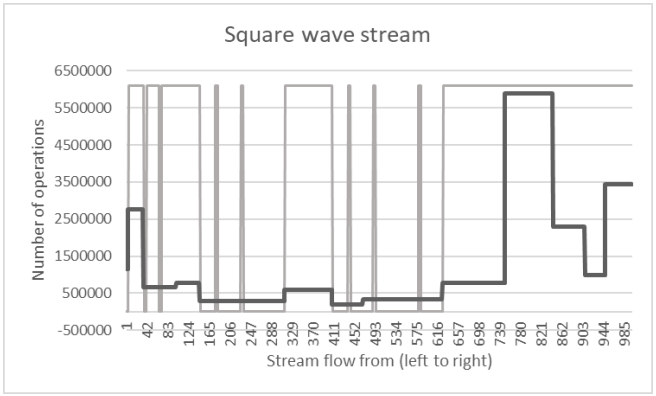
The surge peaks along the thin line mean to the sampling phase, the 5 samples executed on the GPU. The algorithm perfectly predicted and executed on the optimal processor other than the sampling executions on the GPU. For this input stream, the gain was noticeable. Though the gain was less in time which 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.



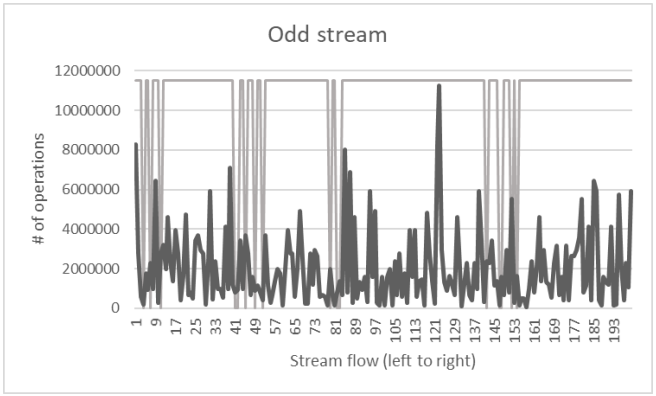
This stream contains only larger inputs, it is obvious we cannot achieve a gain in this case. nevertheless, we have utilized both processors parallelly. There were no peaks in this graph and no sampling taken place because the outliers are caught and sent to GPU. Therefore, no sampling phase and CPU executions were there for this stream at all. But, if continuously large problems were coming, it would have internally 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.



In this stream, inputs clusters preferred for CPU and inputs clusters preferred for GPU come alternatively. The algorithm (thin line) has followed the edges of the input nature (thick line). The gain for this stream was also certain as expected. The performance was improved by 13.2% for this stream.



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 gain was 0.17% and negligible for this stream. Because it had very less likelihood of CPU favor problems coming in the stream. But it will be high in productions systems where their CPU contains tens of cores.



This stream consists of no cluster. Input sizes are all random. This stream would even not yield any gain. 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. It can be ignored and gain from other streams would compensate this.

Stream\Algorithm	CPU	GPU	Hybrid	
Binary	64144.8	9432.4	8332.4	+ 13.2%
Square	32303.4	7694.2	7681.4	+ 0.17%
GPU favor	184447.6	22167	22199.8	- 0.15
CPU favor	577.8	2586.8	623.6	+ 314.81%
Random	34750.2	9122.6	9217.6	- 1.03%

The table above shows the execution elapsed time taken to execute all five types of input data streams in only CPU, only GPU and the hybrid execution. The percentiles shown in the right-hand side shows the gain or loss from the algorithm over the GPU. Binary, square and CPU streams yield performance gain while other two had overhead due to the algorithm processing. It is impossible to earn gain for the streams by utilizing CPU that all inputs are suitable for GPUs.

However, there will be an overhead due to the algorithm processing, but the overheads were negligible compared to the gain achieved through other streams. The loss in GPU favor stream was due to the delay incurred by the algorithm. The loss incurred for the random input stream depends on the nature of random data.

VI. CONCLUSION.

The explored feasibility of the research was true. Executing low range problems takes less time on CPUs compared to GPUs while CPU can be utilized when GPUs are under contention. This research has analyzed the advantage of less execution time on CPU but not utilization of CPU since the above experiments were single threaded scheduling the input streams. The evaluation of the inputs was barely based on execution times of the samples, assuming that the same kind of inputs comes in clusters at an instance. It would consider present loads on the processing units which prevents overwhelming either of them. But outliers caused severe delays on CPUs. Therefore, it was needed to evaluate the inputs streams with regards to some of their properties provided by programmers. XgBoost ML model has been used for this purpose but works well even though some papers have not suggested an ML model solution for this kind of scheduling problems. The final solution was general where programmers can add new compute intensive GPU executable problems as separate models with custom CPU and GPU implementations to the library. The “ComputationalModel” base class would evaluate their inputs and send them to the right unit. This solution can be scaled up and adopted for many other complex computational models when it comes to production.

VII. ACKNOWLEDGMENT

We would like to express our deep and sincere gratitude to our supervisors, Janaka Perera, Associate Architect at LSEG and Adeesha Wijayasiri, Senior Lecturer, Department of Computer Science and Engineering, University of Moratuwa for giving us the opportunity to do research and providing invaluable guidance throughout the research. Their dynamism, motivation and sincerity have deeply inspired us to complete this research successfully.

VIII. REFERENCES

[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.

[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.

[3] 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.

[4] Cullinan, C., Wyant, C. and Frattesi, T., 2020. Computing Performance Benchmarks among CPU, GPU, and FPGA.

[5] 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.

[6] 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.

[7] Cullinan, C., Wyant, C. and Frattesi, T., 2020. Computing Performance Benchmarks among CPU, GPU, and FPGA. p.15.

[8] 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”.

[9] 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.

[10] 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.

[11] A. Chikin, J. Amaral, K. Ali and E. Tiotto, Toward an Analytical Performance Model to Select between GPU and CPU Execution, 2019.

[12] 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.

[13] 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.

[14] 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.

[15] 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.