Self-accelerating Processing Workflows

(GPU vs CPU with insights to optimize)

- A framework that predicts on which a problem is the best to execute -

Research Project Proposal

29th March 2020

Department of Computer Science and Engineering

University of Moratuwa

Supervisors:

- **Appointed**

  Janaka Alawatugoda,

  Software Architect,

  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

Table of Figures

1. Introduction

The right resources should be utilized properly to improve the performance of a computer system. The GPU was originally invented for graphical processing in which a huge number of parallel and correlated computations are very common. Later, computer scientists recognized that the GPU can be utilized in some other parallel computation problems and use it for general-purpose programming, the GPGPU evolved. Not all problems are good to be scheduled on a GPU all the time. Because the GPU may be busy in graphical processing, still some problems are limited to the CPU and some problems are less efficient to be executed in a GPU compared to the CPU. The GPU is good at processing problems that have common instructions with input data parallelism. Though it has a bottleneck on its bandwidth to transfer data between a host and a device. This research proposes a solution, a library that determines on which platform (CPU or GPU) a specified problem is efficient to execute. This is similar and motivated by the branch prediction concept. So, programmers can characterise problems and leave the job to select the optimal platform dynamically to the system. Though there are researches to optimize the performance by scheduling properly, it does not have more related previous works.

2. Problem Statement

Many computations can be done either on CPU and GPU platforms. The computation time of the problem may vary in factor if a problem is executed on a wrong platform compared to the other one. A right choice may reduce the overall execution time a lot. Few related types of research have been conducted previously, but none of them provides an appropriate solution. Programmers have no option to do the selection dynamically and to execute two different codes in the CPU and GPU for a defined computational task. The right processor selection of problems would reduce the execution time by a certain factor.

3. Research Objectives

A library that essentially contains a set of functions that determine whether a problem of which context is given as arguments to the function should be executed in GPU or CPU to reduce execution time. The library must provide gain, reduction in overall execution time over the selection process computation.

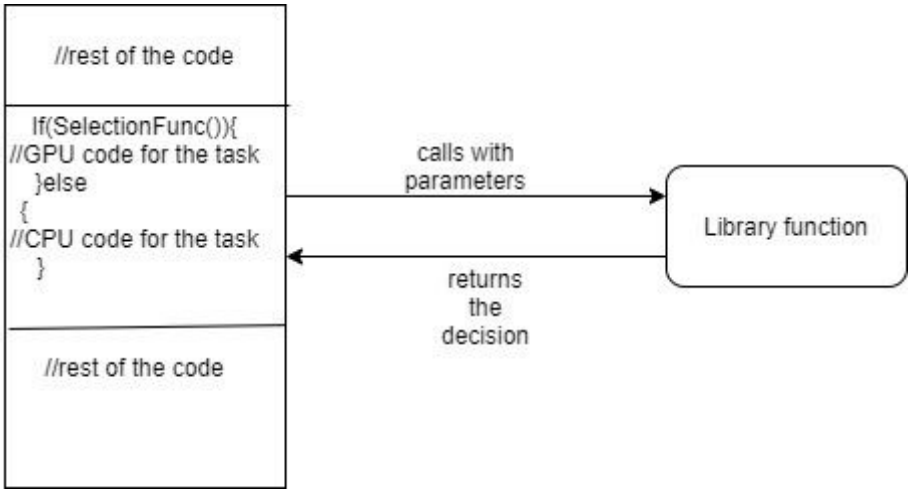


Figure 3.1 - Architectural diagram large in view

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

- A platform selection library for presenting the solution.
- It will contain a set of methods.
- Each method will be specialized for a particular group of aspects which will be considered for switching platforms.
- The programmer should be aware of or have any idea about using the library in a particular part of his code.
- He/She should also have knowledge about selecting the methods to be used for a particular evaluation.

4. Literature Review

4.1. Introduction

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 few hundreds to thousands of cores. Also, the GPU has access to a huge array of memory which enhances parallel computing. 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 CPU still does much better than 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 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. The device memory has 40-bit address space for both linear allocation, and to store references of objects or pointers to the references [8].

4.2. CPU Limitations

CPUs often have not more than 8 cores and each core accepts an independent instruction set. A host can have one or more CPUs though general hosts are built with a CPU. 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. A CPU has a higher context switching time to provide concurrency with the less number of cores.

Although the CPU has a higher 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. This is just not enough to handle huge growing datasets and their applications. The throughput required for modern computations cannot be achieved only with a CPU any more. Also, it is not efficient to use CPU platforms, even though some hosts such as supercomputers, clustered computing systems are fast enough to handle such huge datasets since many of the computations are similar but executed serially.

Author of the paper [12] suggests optimizations for the CPU to improve performance. They are SIMD purpose reorganization of memory access, multithreading and cache blocking.

4.3. GPU Limitations

The GPU was originally invented for graphics processing purposes and hence they were good at processing similar operations in bulk over a huge amount of data. The ALU of past GPUs were limited to 24-bit precision as 24-bits are enough to represent the colour of a pixel. It had been several attempts made to utilize the GPU for some general-purpose programming. However, multi-instruction emulation of sequences had been required for integer arithmetic that are more than 24-bits [5]. The limitation of serial programming and emergence of the computation requirements over a large database such as Data Mining, Medical Imaging and Artificial Intelligence leads to a new revolutionized GPU for general-purpose programming which supports 32-bit and 64-bit precision [5].

Modern GPUs have the ability to do almost all the operations that a CPU can do. Fermi GPU is a good example to illustrate here. It has 16 to 32 Streaming Multiprocessor(SM) which can be thought of as small independent processors, but sharing a global memory space in the GPU [11], [5]. Since a Fermi GPU can do 512 operations in parallel, the GPU is said to have 512 cores. However, at this time, a Fermi GPU can only execute a maximum of 32 kernels at once; two kernels per warp means 32 independent instructions per cycle. 32 kernels schedule and execute 512 threats from different thread blocks in the 512 cores in 16 SMs. But concurrency wise, a GPU can handle up to 65,535 at once in grids up to 1024x1024x64 [6]. A SM is scheduled with one or more threat blocks but a threat block can only be scheduled to a SM [5]. A thread block may consist of up to 1024 threats, but only any set of 32 threats (a warp defined by the programmer) can only be executed at once in a SM [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 function that executes on a SM of a GPU by a collection of threads in parallel is called a kernel. The programmer would configure the kernels on which set of threads the kernels should be invoked. However, the thread block scheduling to a SM at a given time is handled by a hardware scheduler.[7]

The computability of a GPU is limited to its bandwidth, meaning that the memory transfer time between the host and the device is high. A GPU is connected to the host via PCI-Express [5]. It is required to copy data into the device and from the device after computation. If more data transfers need to be done, then it is not efficient to execute the computation in the CPU. Therefore, the arithmetic intensity is used to measure the suitability of a problem to a GPU. The GPU has different levels of memory hierarchy as follows,

- Each threat owns a Private Memory
- Each block owns a Shared Memory, which is shared among its constrained threads,
- Global Memory that accessible to all threads
- ReadOnlyMemory stores the Constants and Textures defined during compilation
- Also, in Cuda architecture, the GPU has direct access to the system DRAM also

The device can directly access the main memory of the host which avoids the need of copying data to the device and enables oversubscription of the device, but is still very slow [8]. The load/store units are responsible for setting the source and destination address (creating a path) before transferring data between the host and the device. Each SM in a Fermi GPU has 16 such units and hence such paths can be created for sixteen threads per clock [5]. The bandwidth bottleneck may be removed by increasing load/store units.

A GeForce GTX460 GPU has a bandwidth maximum of 115.2 GB/sec but the maximum theoretical value of a PCI slot of the latest version is 32 GB/sec. This differentiation is because the GPU could provide more bandwidth for larger files. In practical terms, GPUs are not capable of utilizing the full bandwidth and the data transfer speed often not exceed 3.278GB/sec. Also, the host's bandwidth is limited to 4GB/sec [11]. The following table would illustrate how the maximum bandwidth changes with the size of the data,

Size of Data Chunk (kB)	GB / second	Time (milliseconds)
1	0.0968	0.01057
2	0.1919	0. 01067
4	0.3399	0. 01204
8	0.6074	0. 01348
16	1.0987	0.01491
32	1.5839	0.02069
64	2.1322	0.03074
128	2.5777	0.05084
256	2.8825	0.09094
512	3.0872	0.1698
1024	3.1711	0.3306
2048	3.1773	0.6602
4096	3.1769	1.3211
8192	3.2272	2.5995
16384	3.2552	5.1541
32768	3.2357	10.369
65536	3.2696	20.525
131072	3.2733	41.004
262144	3.2781	81.887
524288	3.2783	163.768

Figure 4.1 - Host - GPU data rates vs file sizes [11]

But, the data transfer rates are high within the device, but vary with the level of the hierarchy. It may cost more execution time if the data to be processed does not fit within a block because of the segmentation and swapping and pagination overhead.

The DMA engine provides GPU access to CPU memory. But the access rate is slow. The memory that is pinned by the CPU can only be accessed by the DMA. Though, the DMA allows GPU to directly access the pinned memory but slow.[7] Therefore, an atomic computation cannot be executed both in a CPU and a GPU if they are sharing variables but a problem can be executed partially, the serial part in the CPU and the parallel part GPU.

Read / Write	Memory Block Size	Speed (GB/sec)
Local	32	190 / 181
Local	64	288 / 328
Local	128	299 / 388
Local	256	289 / 385
Local	512	277 / 368
Global	32	7.4 / 3.7
Global	64	5.8 / 3.5
Global	128	4.8 / 3.4
Global	256	4.3 / 3.4
Global	512	4.1 / 3.3

Figure 4.2 - Bandwidth of local and global memories of a GPU [11]

A warp is a group of threats (often 32) that are executed in an SM (single instruction) [5]. A global scheduler is employed to assign threat blocks to the SMs [5]. SM scheduler schedules a set of threats in a block as a warp to its core itself [5]. A GPU is utilized with 100 percent efficiency only when all the threats are in a warp following a kernel throughout. All threads in a warp are expected to take the same path and threat divergence occurs otherwise. For example: on a conditional branch. It leads to serious performance degradations. However, the inter-warp divergence of threats does not impact performance [7]. Single Instruction Multiple Thread does only bring a GPU more powerful and not useful if it has a less number of threats per instruction. The Fermi GPU is built with Four SFU(Special Function Units) per SM. If computation needs SFU, it would consume more clock cycles. A warp (32 threats) could complete SFU computations over eight clocks [5].

Also, the context switching time is around 10 times smaller than the GPU. So, a program that consists of more serial code is a counterpart to a GPU as it often requires context switching.

All these losses need to be mitigated by the gain achieved with the parallelism in the huge number of cores of the GPU. GPU computations still require at least one CPU thread. The thread is to issue commands to the GPU kernels typically do not consume many CPU cycles. Even Though, increased parallelism achieved by the GPU will only improve the performance of parallel execution code. The serial part of a program is still a bottleneck [7]. In practice, Some applications can only be executed on a multi-core CPU and some can be on a GPU [7]. Author of the paper [12] suggests two key optimizations to improve GPU performance and are minimizing global synchronization and using local-shared buffers.

The results of the experiments, mentioned in Anna Syberfeldt and Tom Ekblom s' paper [2], showed that the amount of data and the number of available parallel instances have a decisive influence on which platform is more efficient. With larger amounts of data, the CPU is more efficient when a limited number of parallel instances are available, as using the GPU is associated with a significant overhead that negates the possible improvements gained by parallelization. With small amounts of data, the CPU is still more efficient than the GPU, but only up to a relatively low number of parallel instances.

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].

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]. Usually, the decision whether a problem to be executed in the GPU or the CPU is hard-coded by the programmer based on the problem to be solved.

This is resulting in inappropriate or inefficient scheduling of jobs and processes and to an unoptimized use of hardware resources.

4.4. Provided previous solutions to overcome the limitations

CPU(s) and GPU were separate microchips but steps were taken to fuse them into a die sharing a single memory to reduce the memory latency due to the overhead in communication between them. It reduces data transfer time between the device and the host, and memory management is not required. It was not successful in the manner of using GPU for general purposes initially. The GPU was controlled by the CPU earlier, but modern GPUs can be operated independently [1].

Heterogeneous Computing is an evolving architecture, improves latency between CPU and GPU and is expected to be the future of CPU and GPU. It is designed as one chip consisting of two multicore CPUs for two specific tasks, and a GPU. However, the layout may still be necessary for the right selection of the section inside the chip [1].

Rootbeer is a project which manages serialization and generates and launches the kernel code required for a code written in Java automatically itself. It supports the following features of Java [10],

1. Single and multi-dimensional arrays of primitive and reference types
2. Instance and static fields
3. Composite objects
4. Dynamic memory allocation
5. Synchronized methods
6. Inner classes
7. Strings
8. Exceptions related to GPU

5. Methodology

This will essentially be an experimental methodology since the research is based on the results and outcomes of the predictions given by the function that is going to be implemented. It will analyze the method of execution, the execution flow of processing in CPU and GPU and what kind of problems will be more efficient to be executed in the CPU and the GPUs respectively. Research needs to be conducted about the problems processed in a system which will give us knowledge about the dimensions and the attributes resolving the weights of the problems. The parameters accepted by the function which defines the complexity and dimension information of the problems will be extracted with the outcomes and conclusions of the results and further research will be conducted with regards to the parameters.

Also, the framework can be implemented in various ways and has not been selected yet. It could be determined by keeping track of history such as with few past success counts of predictions, logging the successes of the predictions and using the machine learning or merely just predicting it with the weight and the type of problems. However, there should be resource gain from the prediction of the framework and it will be useless otherwise.

We have planned to achieve the solution in a 5-step process. They are

1. Extracting features
2. Prioritizing features
3. Establishing relationships
4. Formation of methods
5. Implement the techniques

5.1. Extracting features

In extracting features, we involve in the process of identifying the attributes that affect the performance of CPU and GPU. We can extract the features through tracking research papers and other documents. For example, the array size and dimension of an array [10]. Mostly we will try to find software-related aspects.

5.2. Prioritizing features

This process filters the features extracted based on the impact level of the attributes in the performance of the GPU or CPU. Giving equal priority for all attributes will create a very specific implementation and lock situations during clashes. It will be useful in some trade-off situations. Though we are implementing the library to a more generalized use, the consideration of hardware aspects of the CPU and GPU will be neglected.

5.3. Establishing relationships

The attributes will show different behaviours while considering individuals and groups. In this process, we are evaluating the behaviours of features when combined between them. The behaviours will be studied in a controlled experiment manner. We will try to make all aspects fixed except the one we are testing on. Then execute the code on both CPU and GPU separately by changing the independent variable. The graph will be drawn with an independent feature on x-axis and time taken to execute on the y-axis. The results will be used to establish benchmarks in functions. For graphical analysis, we have planned to use Matlab.

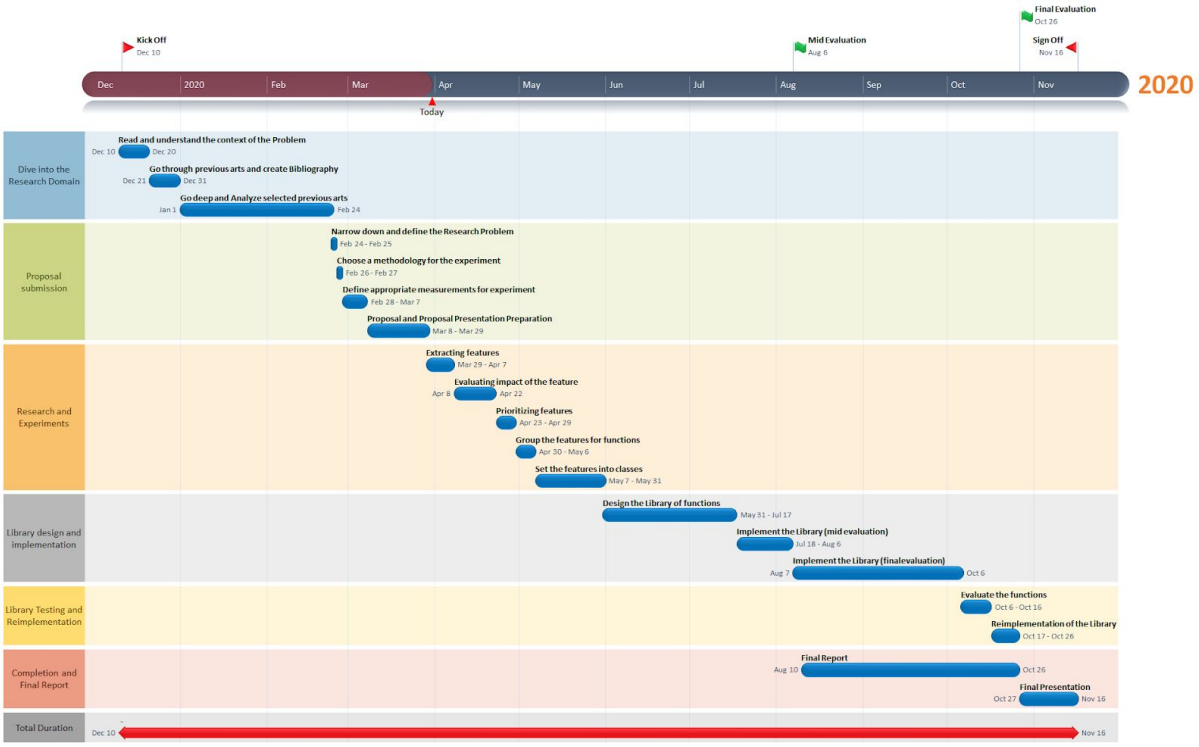
5.4. Formation of methods

In this process, we are trying to group the attributes based on the results obtained from the above processes (2nd and 3rd processes) which will help to reduce conflicts during decision making by the programmer to choose the function.

5.5. Implementation techniques

The final and important step in the process. In this step, we decide how to make a function toggle between the selection of the appropriate platform for the execution of the code. That is the way of formation of logic behind the functions in our library. We have decided to use statistical results to some extent. The features and results obtained from the above steps will be used. In case, we cannot find an appropriate pattern to set benchmarks, as an alternative we are discussing to use neural networks to attain a solution.

6. Timeline



7. Conclusion/Summary

The research is to evaluate a processing problem based on their characteristics provided by programmers to predict whether the problem should be executed in the CPU or GPU in order to reduce the computational time. The outcome of the research will be a framework that contains few functions to be used by programmers. It has also been planned to consider the previous prediction history to make succeeding predictions further accurate. The way how the previous history is stored and analyzed will be decided later. Since this research project is a company based research project, we are specifically analysing in a particular domain where the company is in. But we expect that it won't block the generalization view of the research, that is to provide a boost in the GPGPU area of computation to a great extent.

8. References (IEEE)

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

[2] A. Syberfeldt and T. Ekblom, "A comparative evaluation of the GPU vs. The CPU for parallelization of evolutionary algorithms through multiple independent runs", International Journal of Computer Science & Information Technology (IJCSIT) Vol 9, No 3, June 2017. [Accessed: 31- Dec- 2019] [Online]. http://aircconline.com/ijcsit/V9N3/9317ijcsit01.pdf

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

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

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

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

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

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

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

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

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

[12] Lee, V., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R. and Dubey P. https://dl.acm.org/doi/pdf/10.1145/1815961.1816021?casa_token=WGkb9giVyI8AAAAA:n3DRHmgyO46x6w3e-F12nW-pGJ9P9Pjzvntbs6DdXp4Eg2fYzQxo43Akqde585XkHFjEaDDi0oOd