UNIVERSITY OF CALIFORNIA
RIVERSIDE

Improving Energy Efficiency of Basic Linear Algebra Routines on Heterogeneous Systems With Multiple GPUs

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Hadi Zamani Sabzi

March 2023

Dissertation Committee:

Prof. Laxmi Bhuyan, Chairperson
Prof. Nael Abu-Ghazaleh
Prof. Zizhong Chen
Prof. Daniel Wong

The Dissertation of Hadi Zamani Sabzi is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

Finishing this dissertation marks the end of a major chapter of my life and the beginning of a new innings. I thank you all from the bottom of my heart, for your unquestionable support, continuous encouragement and endless patience, that helped me navigate my PhD Journey.

First of all, I would like to thank my advisor, Dr. Laxmi Bhuyan for his unconditional support. His technical, editorial and general advice, and mentoring was pivotal in the completion of the dissertation. I would also like to thank my dissertation committee members, Zizhong Chen and Dr. Daniel Wong for their support and endless help in my research, and Dr. Nael Abu-Ghazaleh for his support and guidance as this dissertation shaped from a proposal to a complete study. Their constructive feedback has helped me immensely to improve the quality of this dissertation.

Next, I would like to thank my amazing collaborators, Dr. Zizhong Chen, Dr. Daniel Wong, Devashree Tripathy, and Ali Jahanshahi. I would like to thank all the professors, mentors, co-workers, teachers, students, friends in my life for being there for me and believing in me. I thank my labmates and fellow graduate researchers : Devashree Tripathy, Liang Zhou, Ali Jahanshahi, And Sourav Panda.

Finally, most importantly, I would like to express my gratitude towards my family for their unconditional support during my pursuit of this arduous journey. Especially, I thank my parents for providing me a strong education foundation to begin with and rowing the seeds of research aptitude since childhood, my brothers and sister, Shahram, Babak, Hamed, and Roya for always motivating me, for their love, support and encouragement.

ABSTRACT OF THE DISSERTATION

Improving Energy Efficiency of Basic Linear Algebra Routines on Heterogeneous Systems With Multiple GPUs

by

Hadi Zamani Sabzi

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2023
Prof. Laxmi Bhuyan, Chairperson

The current trend of ever-increasing performance in high performance computing (HPC) applications comes with tremendous growth in energy consumption. Because existing libraries are mainly concerned with performance, they do not make efficient use of heterogeneous computing systems, resulting in energy inefficiency. Hence, improving the energy efficiency of critical applications running on HPC systems is necessary to deliver better performance at a given power budget. The aim of this dissertation is to develop techniques and frameworks to improve the energy efficiency of the high performance applications on heterogeneous system with GPUs while maintaining the reliability and performance requirements.

In our first approach, we present GreenMM framework for matrix multiplication, which reduces energy consumption in GPUs through undervolting without sacrificing the performance. The idea is to undervolt the GPU beyond the minimum operating voltage (V_{min}) to save maximum energy while keeping the frequency constant. Since such undervolting may give rise to faults, we design an Algorithm Based Fault Tolerance (ABFT) algorithm

to detect and correct those errors. We target Matrix Multiplication (MM), as a key kernel used in many scientific applications. Empirically, we explore different errors and derive a fault model as a function of undervolting levels and matrix sizes. Then, using the model, we configure the proposed fault tolerant MM algorithm. We show that energy consumption is reduced up to 19.8%. GreenMM also improves the GFLOPS/Watt by 9% with negligible performance overhead.

In our second study, we present a framework for GPU applications, which reduces energy consumption in GPUs through Safe Overclocking and Undervolting (SAOU) without sacrificing performance. The idea is to increase the frequency beyond the safe frequency $f_{safeMax}$ and undervolt below $V_{safeMin}$ to get maximum energy saving. Since such overclocking and undervolting may give rise to faults, we employ an enhanced checkpoint-recovery technique to cover the possible errors. Empirically, we explore different errors and derive a fault model that is used to determine the appropriate undervolting and overclocking level for maximum energy saving. Similarly, we target MM kernel for error correction using the checkpoint and recovery (CR) technique as an example of scientific applications. In case of MM, SAOU achieves up to 22% energy reduction through undervolting and overclocking without sacrificing the performance.

In our third study, we introduce GreenMD, an energy-efficient framework for heterogeneous systems for LU factorization utilizing multi-GPUs. LU factorization is a crucial kernel from the MAGMA library, which is highly optimized. The aim is to apply DVFS by leveraging slacks intelligently on both CPU and multiple GPUs. To predict the slack times, accurate performance models are developed separately for CPUs, GPUs, and

PCIe bus based on the algorithmic knowledge and manufacturer's specifications. We also determine the appropriate level of undervolting for both CPUs and GPUs through offline profiling. Reducing voltage below threshold values may give rise to errors; hence we extract the minimum safe voltages ($V_{safeMin}$) for the CPUs and GPUs utilizing a low overhead profiling phase and apply them before execution. It is shown that GreenMD improves the CPU, GPUs, and total energy about 59%, 21%, and 31%, respectively, while delivering similar performance to the state-of-the-art linear algebra library MAGMA.

In our fourth study, we introduce a fault tolerant algorithm for LU factorization on heterogeneous systems with GPUs. We have developed a fault tolerant algorithm by constructing a local and global checksums. The local checksums are used to detect the errors and the global checksums are used to correct the errors. Using the local checksums, we can detect the errors in the middle of the computation which enables us to tolerate more number of faults during the whole execution. LU factorization has three main phases. These phases have different sensitivity to the error. For each phase, we introduce an appropriate level of fault tolerance to prevent the error propagation to other phases. Since, we check the correctness of the computation in each iteration, if there is any error in the system, only the small fraction of the computation is affected and can be covered easily in compared to the previous works such as ABFT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High Performance Computing (HPC) applications like molecular dynamics, weather prediction and drug discovery demand parallel processing environments. General Purpose Graphics Processing Units (GPGPUs) have evolved as high performance accelerators due to their SIMD (Single Instruction Multiple Data) processing architecture. Modern GPUs with hundreds of computing cores are capable of 7.8 $TFLOP/s$ of double precision floating-point (FP64) and 15.7 $TFLOP/s$ of single precision (FP32) [57]. Moreover, GPUs are equipped with huge memory bandwidth as high as 1 TB/s. These characteristics make them well-suited for use as accelerators in HPC applications, especially for numerical computations and vector processing. Given their high computational capabilities, the GPUs consume a significant portion of the total system energy. Because existing libraries are mainly concerned with performance, they do not make efficient use of heterogeneous computing systems, resulting in energy inefficiency. Hence, improving the energy efficiency of critical applications running on HPC systems is necessary to deliver better performance at a given

power budget. Matrix multiplication (MM), and Matrix decomposition algorithms are heavily used in many important numerical computations. The matrix-multiplication kernel, referred to as GEMM in the Basic Linear Algebra Subroutines (BLAS) [56], is frequently used as a basic numerical calculation library in CPUs. GEMM and LU decomposition routines are critical to the performance of High Performance LINPACK benchmark (HPL) and many software packages solving problem in linear algebra such as LAPACK, ScaLAPACK, MUMPS and SuperLU. Improving the energy efficiency usually comes with degrading the reliability. For instance reducing the operating voltage will result in experiencing more faults in the system. Soft errors can occur in heterogeneous computing systems that use GPUs. Recent studies have revealed that GPUs are susceptible to soft errors and any error in long running kernels, if not solved, could cause restart that considerably rises the GPU workload. According to the previous works, soft error rate increases as the voltage reduces more [46]. Hence, there is trade-off between performance, energy efficiency, and reliability. Previous works have not considered the impact of energy efficiency, reliability, performance together. In this thesis we aim to improve the energy efficiency of the heterogeneous systems while marinating the performance and reliability requirements.

1.1 GreenMM: Energy Efficient GPU Matrix Multiplication Through Undervolting

Over the past few years, there have been significant efforts to study different techniques improving energy efficiency of GPUs such as Dynamic Voltage and Frequency Scaling (DVFS) [34] [60], and load balancing in the CPU-GPU heterogeneous systems [59]

[79]. However, DVFS techniques result in performance degradation due to lowering of the frequency.

GPUs are designed to operate in worst case operating conditions in terms of process, temperature and voltage variation. A comprehensive study has been done on several commercial GPU cards showing that there exists about 20% voltage guardband on different GPU cards, which, when utilized, can result in up to 25% energy saving on GPU cards [46].

The impact of undervolting for energy saving has thoroughly been analyzed recently by reducing the voltage down to the safe minimum voltage [46] [49]. Leng et al. [46] explore energy benefits of reducing voltage of the GPU chip down to the safe limit. We aim to save even more energy through undervolting the GPU beyond the safe minimum operating voltage and tackling the possible GPU faults by employing a configurable low-overhead fault tolerant (FT) algorithm. According to [46] and our observations, different applications affect the V_{min} at which the program executes correctly but fails when the voltage is reduced any further. The errors can be classified into Silent Data Corruption (SDC), Run-time Faults, Segmentation Faults, and Operating System (OS) crash. Some types of errors lead to divergence in the application control flow, and as a result, increase the execution time and even in some rare cases end up in an infinite loop [46]. The most predominant error is SDC. In Fast Fourier Transform (FFT), Matrix Multiplication and Hotspot benchmarks, the SDC errors lead to 24%, 42% and 55% faulty executions, respectively [69].

The reliability loss due to undervolting is not acceptable for most scientific computing cases. There are software level fault tolerant techniques such as DMR (dual modular redundancy) [66] and TMR (triple modular redundancy) [58], which take advantage of

redundancies for handling erroneous cases, and checkpointing that tolerate errors in a checkpoint-restart manner [74]. These techniques are not very efficient for large scale scientific applications due to large energy and performance overheads [29]. In such cases, algorithm based fault tolerance (ABFT) [41], which tolerates errors at the application level, plays a crucial role in error detection and correction in the systems. ABFT techniques in GPUs were introduced for MM [27], Cholesky [16] and Fast Fourier Transform [73]. Tan et al. [93] proposed a technique for undervolting CPUs and correcting errors through ABFT techniques.

In our first study, we introduce an energy efficient and ABFT framework, ***GreenMM***, which tolerates system errors due to undervolting. In our proposed framework, we use a combination of undervolting and ABFT for GPUs to guarantee energy, power, reliability, and performance efficiency of the system. First, we experimentally determine $V_{safeMin}$, which is the undervolting level beyond which the Operating System crashes for different applications. The proposed GreenMM framework exploits the voltage slack between V_{min} and $V_{safeMin}$ using a lightweight offline profiling to accurately predict the needed fault coverage capability as a function of matrix size, undervolting level and architectural details. We modify the offline ABFT algorithm by incorporating a number of faults. Online ABFT algorithms have also been proposed to reduce the overhead for detection and correction of large number of faults [41]. The basic idea is to decompose the large matrix into several blocks, which are individually protected through checksums. Unlike the offline algorithm, the overhead is lower and faults are not propagated to the output. GreenMM framework is developed for both offline and online algorithms. GreenMM achieves comparable performance (with 1.5%

performance overhead) to highly optimized cuBLAS-MM in the cuBLAS library, but needs a lot less energy, which enhances the performance per watt of the GPU.

To summarize, GreenMM has two parts, GPU Undervolting model and **F**ault **T**olerant **MM**. In *GPU Undervolting model* we determine the fault rate, V_{min} and $V_{safeMin}$ for MM.The undervolting is started from nominal voltage till V_{min}, during which no fault is encountered. However, when we undervolt further from V_{min} till $V_{safeMin}$, FT-MM corrects the errors on the fly. GreenMM makes the following contributions:

- We experimentally determine the Vmin and $V_{safeMin}$ for different applications, including matrix multiplication.
- We develop a fault model for GPU undervolting and determine number of faults as a function of matrix size and degree of undervolting.
- We design a fault tolerant framework, "*GreenMM*", for matrix multiplication that provides peak performance on GPUs. We incorporate the number of faults and modify the original MM to implement offline and online FT-MM algorithms.
- *GreenMM* is transparent to applications which utilize the matrix multiplications, i.e. it uses the same programming interface as cuBLAS-MM and GreenMM users do not need to modify source code of the cuBLAS.
- *GreenMM* is portable, i.e. it can be used with any GPU architecture just by changing some architecture specific parameters in the model.
- We present various experimental results in terms of energy, power, performance and reliability. GreenMM achieves up to 19.5% energy reduction compared to the original MM. Beside that, it improves the $GFLOPS/Watt$ of the GPU up to 9%.

1.2 SAOU: Safe Adaptive Overclocking and Undervolting for Energy-Efficient GPU Computing

Ever-increasing performance demands have led to the GPU-based acceleration in a wide range of computing systems from mobile devices to super computers. While GPUs deliver high computational capability, they consume a significant portion of total system energy. As technology advances towards deep sub-micron level, the static power becomes a serious problem. Several techniques including Dynamic Voltage and Frequency Scaling (DVFS) and power gating techniques have improved energy efficiency of the GPUs [26,34,60]. However, DVFS and power gating techniques usually degrade performance due to lowering the frequency or putting the components into sleep mode.

GPUs are designed to operate in worst case operating conditions in terms of process, temperature and voltage variation. A comprehensive study has been done on several commercial GPU cards showing that there exists about 20% voltage guardband on different GPU cards, which, when utilized, can result in up to 25% energy saving on GPU cards [46]. Through GreenMM framework, we have explored undervolting to save energy for GPUs for a Matrix Multiplication (MM) algorithm [105]. GreenMM saves the energy through undervolting beyond the $V_{safeMin}$ and employing algorithm based fault tolerant (ABFT) technique to correct the errors [105]. $V_{safeMin}$ is identified as the minimum voltage for a GPU to operate without generating any fault. However, ABFT can be applied to only very regular algorithms, like matrix multiplication [42]. On the other hand, checkpoint and recovery (CR) is a general technique that can be applied to both regular and irregular applications. In SAOU, as an example, we have built our framework on top of GreenMM by

enabling the CR technique. We aim at using the voltage guardband to save energy while preserving performance. Since the GPU is undervolted at a fixed frequency, it does not incur any performance degradation.

Overclocking is another technique that reduces the execution time through boosting the frequency to a higher level. Even though the power consumption increases, the total energy can be saved due to the reduction in the execution time. However, overclocking beyond $f_{safeMax}$ may raise errors due to having less time for charging and discharging transistors. We experimentally determine the safe values for the maximum safe frequency for different applications, similar to the undervolting experiments done earlier [46, 105].

We develop an undervolting and overclocking model and validate with several applications. We specifically target cuBLAS Matrix Multiplication (cuBLAS-MM), a key kernel used in many scientific applications and implement checkpoint and recovery (CR) on top of cuBLAS-MM.

CR is a general resilience technique that is often used to handle hard errors but it can also recover soft errors. Several CR mechanisms are developed for CPUs [64, 98]. However, none of them is feasible on NVIDIA GPUs due to the absence of particular runtime APIs to extract computation state inside the kernel. Considering these limitations, a handful of GPU CR schemes are developed [35, 68, 76, 90]. CheCuda [90] is built on top of checkpoint and recovery library called BLCR [35] to store the system state. Since the BLCR does not support CUDA contexts, before checkpointing, it stores and destroys CUDA contexts, then runs BLCR to reallocate all destroyed GPU contexts. These extra phases, incur a huge performance overhead to the system. NVCR is another checkpoint and recovery library

which is transparent to the applications [68]. CheCL is another checkpoint and recovery technique that follows CheCuda but it is designed for OpenCL-based applications [88]. All these checkpoint and recovery techniques reload GPU state and re-launch kernels from the beginning which incurs huge performance overhead.

We propose application specific incremental CR to preserve the computation state. Our technique is similar to the in-kernel, in-memory, and incremental CR technique, employed in [76]. However, we specifically target the cuBLAS-MM library and extend it to handle faults arising from undervolting and overclocking. Through a detailed fault model based on our experiments, we determine the locations of checkpoints in the kernel. SAOU preserves computation states in the GPU device memory to eliminate transfer time between CPU and GPU. We also adopt an incremental CR that only saves the variables that have been modified since the last checkpoint. It can improve the performance but requires us to keep track of the modified variables. Once a failure happens, the preserved computation state will be loaded and the execution will be resumed from the last checkpoint. Clearly, the cost of a checkpoint will vary with the amount of states required to be saved and the bandwidth available to the storage mechanism being used to preserve the state.

This work presents performance and energy consumption using the combination of overclocking, undervolting and FT algorithm. Through experiments, we show that SAOU achieves up to 22% energy reduction for a $10K \times 10K$ matrix multiplication. SAOU makes the following contributions:

- First, we empirically find $f_{safeMax}$ and $V_{safeMin}$ for different applications.
- Next, based on our overclocking and undervolting results, we develop a fault model to determine the number of checkpoints and their locations in the application.
- We design an incremental in-kernel and in-memory CR technique to overcome any transient faults during the execution.
- We verify the operation by applying the proposed technique to cu-BLAS-MM library and executing it on a NVIDIA GTX 980 GPU.

SAOU framework makes the following contributions:

- First, we empirically find $f_{safeMax}$ and $V_{safeMin}$ for different applications.
- Next, based on our overclocking and undervolting results, we develop a fault model to determine the number of checkpoints and their locations in the application.
- We design an incremental in-kernel and in-memory CR technique to overcome any transient faults during the execution.
- We verify the operation by applying the proposed technique to highly optimized MM kernel and executing it on a NVIDIA GTX 980 GPU.

1.3 GreenMD: Energy-Efficient Matrix Decomposition on Heterogeneous Multi-GPU Systems

HPC systems are increasingly using heterogeneous systems, FPGAs and GPUs, with multicore processors to boost the performance of scientific applications. GPUs, in particular, have been widely employed for HPC due to their extraordinarily high compute capability and easy programmability. High performance GPUs, on the other hand, with large

power consumption [63]. Because existing libraries are mainly concerned with performance, they do not make efficient use of heterogeneous computing systems, resulting in energy inefficiency. Hence, improving the energy efficiency of critical applications running on HPC systems is necessary to deliver better performance at a given power budget.

LU factorization is an algorithm used for solving dense linear algebra problems, and is widely used in many scientific and engineering applications [55]. It is included in several popular linear algebra libraries, such as Linear Algebra Package (LAPACK) [6] and Linpack [72] benchmarks. Existing LU factorization implementations are concerned primarily with performance, ignoring the potential for energy savings that do not have a negative impact on performance. When LU factorization is running on heterogeneous system equipped with GPU, it divides the workload between CPUs and GPUs. CPU handles the panel factorization, which is sequential in nature. The GPUs update the row panel and trailing matrix because they involve large computation that is highly parallelized. During the execution, either the CPU or the GPUs could be on non-critical paths that can experience idle time, or slack. These slacks can be exploited for energy savings, by exploring the power-aware techniques such as the Dynamic Voltage and Frequency Scaling (DVFS).

Over the last few years, significant efforts have been made to apply various techniques, such as Dynamic Voltage and Frequency Scaling (DVFS) [32] [61] and undervolting [105]. DVFS approaches have been employed to save energy during underutilized execution phases of the execution, called slack, on CPU [53] [5] [81] [80], and GPU [33] [95] [105] [19]. Using algorithmic knowledge, the amount of slack on different components can be estimated to determine the appropriate level of DVFS so that all the components finish execution

at the same time. Hence, the main task is to accurately predict the slack during the execution so that the exact frequency can be computed and DVFS can be enabled. A prior research on LU factorization utilized the slack using a simple performance model in a single GPU environment [19]. It profiled the application to determine the execution time of the first iteration, and then estimated the slack for the next iterations based on the prior iteration. However, we develop a more accurate performance model based on the amount of computation extracted from the algorithmic knowledge for heterogeneous systems with multiple GPUs that does not need profiling phase and performance overhead. Also, we extend the DVFS technique to multiple GPUs and present both performance and energy saving results with two GPUs. We derive the execution times of a multicore CPU and multiple GPUs separately and verify the results through experiment. Then the amount of slack is determined at every iteration of the LU factorization, frequency is calculated, and DVFS is enabled at runtime. Our measurement shows that DVFS reduces the energy consumption by 15%.

DVFS techniques are mainly focused on the dynamic power. Static power can be reduced only through the voltage reduction, called undervolting. However undervolting below the threshold value will introduce errors. Manufacturers specify large safety margins in the nominal frequency-voltage operating points of CPUs, up to 30% [31] [22]. As a result, they are inherently conservative and not energy efficient [10]. Leng et al. investigate the voltage guardband of the GPUs and they observe that there is about 20% voltage guardband on different GPU architectures [47]. There are many efforts to operate hardware at sub-nominal voltage levels on the CPUs [31] [22] [70] [108] and GPUs [94] [50] [47] [105]. However, most

of this work focuses on applying undervolting in a conservative way, based on the worst voltage constraint across all workloads and running frequencies. This limits the potential gains since some applications can operate at a higher degree of undervolting. Hence, we empirically extract the minimum safe voltage ($V_{safeMin}$) of the underlying CPU/GPUs for various running frequencies while executing the LU factorization. Using the extracted $V_{safeMin}$ for both CPU and GPUs, we apply the undervolting during the execution of the LU factorization. It is shown that undervolting reduces the energy consumption by 16% beyond DVFS.

GreenMD propose a framework that improves the energy efficiency of heterogeneous multi-GPU systems while maintaining the reliability and performance requirement of LU factorization. GreenMD is built on top of the LU factorization from the highly optimized linear algebra library MAGMA. First, we develop accurate performance models for CPU, GPU, and PCIe bus based on the algorithmic knowledge and underlying hardware details and verify through rigorous experiments. Then we predict the slack, and employ the DVFS on both the CPU and GPUs during the slack periods to save energy. GreenMD also extracts and utilizes the maximum level of undervolting at a fixed frequency to improve the energy efficiency of the system without sacrificing performance. GreenMD is portable, which means it can be used with any GPU and CPU architecture by simply adjusting a few architecture specific parameters. In summary, GreenMD makes the following contributions:

- Using algorithmic knowledge and hardware configuration, we develop a more accurate performance model for both CPU, GPUs, and the PCIe bus to estimate the execution time of the LU factorization during the different iterations of the execution.

- We implement DVFS in CPU, single and multiple GPUs based on the accurate performance models.
- We implement undervolting in CPU, single and multiple GPUs based on the empirical observations.
- We evaluate the performance, energy saving, and reliability through real implementation and achieve 31% total energy saving for double precision LU factorization with a matrix of size 18K*18K.

1.4 Fault Tolerant Matrix Decomposition on Heterogeneous Multi-GPU System

With scaling down the technology and employing more number of computing cores in the hardware accelerators, they become more prone to different types of faults and errors. Hence, with increasing the number of cores, the probability of fault increases and as a result, it is vital to employ an fault tolerant algorithm for the long running applications. However, employing the appropriate method of fault tolerance with a reasonable overhead is critical to recover the application from the error. The general fault tolerant algorithms incur huge amount of performance overhead. So it is necessary to employ a proper FT algorithm to reduce the performance penalty.

LU factorization is one of the key kernels in high performance computing and soft errors or computation errors are the errors we aim to cover during the LU factorization. A single error in the LU factorization can spread to large sections of the matrix. Thus it is

essential to often verify the matrix for correctness to prevent the errors from spreading to the point where they are no longer recoverable. We have discovered that different matrix sections demonstrate varying degrees of sensitivity to errors, resulting in certain sections requiring less frequent verification and some sections more frequent verification. The LU factorization has been powered by the global matrix checksum to recover from fail-stop failures [25]. With this approach, just one checksum was used, and it was assumed that failure details would come from another source. If one of the processes fails to run, the system can quickly indicate the process' location. Therefore, the error detection step is not necessary when a fail-stop failure must be addressed. On the other hand, soft errors typically leave no sign that an error has happened. An inaccurate computation result may be the only indication of a soft error.

To tolerate the soft errors, algorithm-based fault tolerance (ABFT) is a known technique that applies a checksum to a matrix and ensures that the sum will remain accurate even after performing an operation on the matrix [39]. When the sum is incorrect at the end of the computation, there has been a mistake. The sum may be used to determine the correct values or the computation may be repeated, depending on the approach and the type of error. To detect and correct a single soft error using ABFT, we need to add both row and column checksum which incurs a performance overhead. For more number of faults, we need to apply more number of weighted checksum vectors. For instance to tolerate two errors, we should append two weighted checksum vectors in each dimension of the matrix.

In case of LU factorization, we have developed the idea of a checksum on a matrix by figuring out how to construct it so that it is correct throughout the entire LU factorization.

We are able to provide fault tolerance for the right looking LU factorization by utilizing a local and global checksum. For local checksum, we append the checksum to each block of input matrix. For global checksum, we construct the additional blocks that consist of sums of the local blocks in a row. Error detection is done using the local checksum and error correction is done using either the global checksum or repeating the computation. LU factorization has three main steps. Panel factorization, row panel update and trailing matrix update. If any error is detected in the panel factorization, the whole computation will be repeated. This phase is done on the CPU side due to its sequential nature. If the error occurs in the row panel update or the trailing matrix update phase, the global checksum will be used to recover from the error. With this approach, we check the correctness of the computation in each iteration and if there is any error in the system, the small fraction of the computation needs to recovered either using the global checksum or repeat the faulty fraction.

In summary, our proposed fault tolerant LU factorization makes the following contributions:

- Introducing the local and global checksum for the multi-GPU implementation of right looking LU factorization.
- Reducing the overhead of the fault tolerant algorithm by reducing the number of checksum vectors and verifying the correctness of the computation in the middle of the computation to reduce the performance overhead of the recovery.
- Implementing the local and global checksum in a way that the checksum relationship is maintained during the execution.

1.5 Dissertation Organization

The rest of the dissertation is organized as follows: In chapter 2, we describe GreenMM framework that improves the energy efficiency of GPUs while running highly optimized matrix multiplication (MM) kernel. Chapter 3 introduces the fault tolerant SAOU framework which employs undervolting along with overclocking to improve the energy efficiency of the GPUs. Chapter 4 discusses the GreenMD framework that uses the slack reclamation technique along with undervolting to improve the energy efficiency of heterogeneous systems equipped with multiple GPUs. Chapter 5 discusses fault tolerant LU factorization that uses both local and global checksum in a blocked LU factorization on heterogeneous systems with multiple GPUs. And finally, Chapter 6 discusses the related work and chapter 7 concludes this dissertation by giving a summary of our work.

Chapter 2

GreenMM: Energy Efficient GPU Matrix Multiplication Through Undervolting

2.1 GPU Undervolting Model

Microprocessor manufacturers usually append an operating guard-band (a static voltage margin) as high as 20% of the nominal voltage, to ensure that the microprocessor functions reliably over varying load and environmental conditions [107]. The guard-bands also account for errors occurring from the load line, aging effects, noise and calibration error [78]. The guard-band grows with increase in variations in technology scaling. However, because we do not encounter these errors every time; significant energy saving can be achieved by reducing guard-band to a much lower supply voltage [45]. In our work, we

aim at using the voltage slack between the nominal voltage and the actual OS safe voltage to save energy while preserving the performance. We use a similar approach as in [46] to reach $V_{safeMin}$, we also build a fault model empirically as a function of the undervolting level and matrix size. In GreenMM, we go a step further by aggressively undervolting and correcting subsequent errors using the ABFT. Shrinking microprocessor feature size and diminishing the noise guard-band increase the transient fault rate. We undervolt till the safe minimum voltage V_{min} without experiencing any faults. Going beyond V_{min}, system may experience soft errors. Although, GreenMM works for all kinds of soft errors, main focus is specifically on transient and computation errors such as SDCs [46]. SDC occurs when the program finishes its execution normally without any error message but results in a wrong output. These errors can be covered at the application level. CUDA run-time errors such as driver faults or segmentation faults caused by memory management drivers can be detected by inspecting the standard error output. Operating System crash occurs after a specific undervolting level (application-dependent), and it is not possible to undervolt the GPU below the "OS crash point voltage" or $V_{safeMin}$.

2.1.1 Fault Distribution in GPU

In order to determine the number of faults to tolerate, we profile the application. We perform sensitivity analysis of different applications by reducing the voltage beyond V_{min} and by recording the faults at each voltage. The sensitivity analysis results help us to reach the minimum voltage at which we can tolerate errors for a given application. First, we execute an application at nominal voltage and record the output as "golden output". Then, starting from base voltage of 1.075V, the underlying GPU (GTX 980) is undervolted

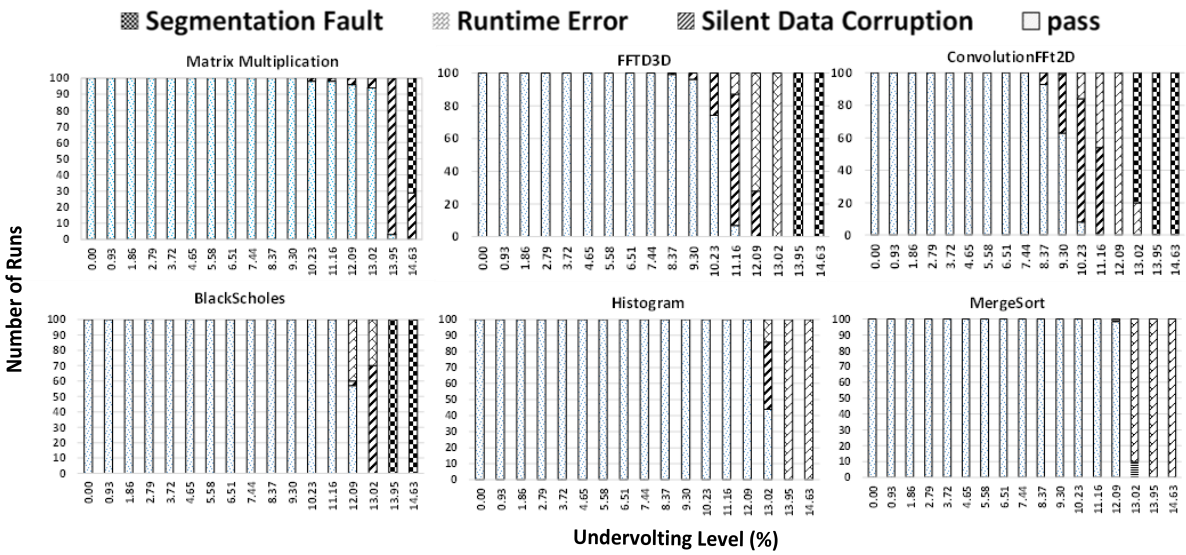

Figure 2.1: Error distribution below nominal voltage for different benchmarks using GTX 980

in step sizes of 10mV. The application is executed 100 times for each level of undervolting and the corresponding output is compared with the golden output to verify correctness. If the output does not match with the golden output, then the application has experienced a failure for that execution. To force the GPU to reduce its voltage at a fixed frequency, we reduce the target power limit of GPU. Fault distribution of different applications such as FFTD3D, FFTD2D, Histogram, MergeSort and BlackScholes on NVIDIA GTX 980 are shown in Figure 2.1. Applications that belong to Rodinia benchmark, are used extensively for performance evaluation of GPU architectures [11]. X-axis denotes the undervolting level starting at 1.075V, and Y-axis denotes the fault types along with their frequencies. Each application experiences different types of errors at different voltages. Some applications such as FFT2D, and FFTD3D show more number of SDC errors as compared to BlackScholes

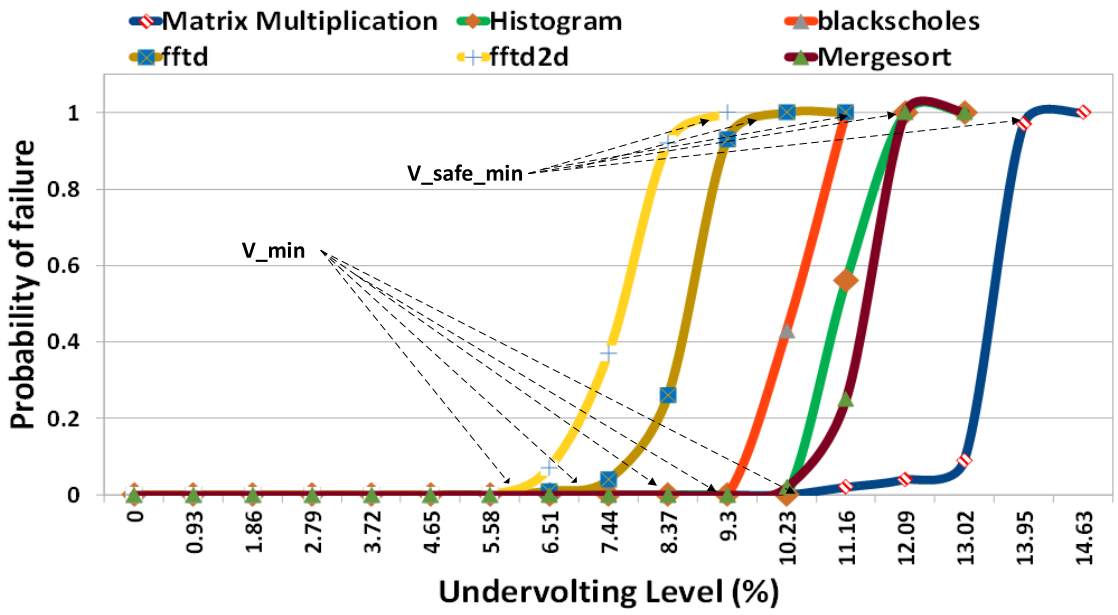

Figure 2.2: Probability of failure for different Rodinia benchmarks and cuBLAS-MM from cuBLAS library

and MergeSort benchmarks. Since SDC errors can be handled at the application level, we only focus on SDC errors.

2.1.2 GPU Fault Model

The probability of failure is given by,

$$P_f = \frac{Number\ of\ failures}{Number\ of\ application\ runs} \tag{2.1}$$

P_f is derived by counting number of failures in Figure 2.1. Figure 2.2 shows P_f for different applications as a function of undervolting. V_{min} is the minimum voltage at which the program executes correctly. ($V_{safeMin}$) refers to theoretical lowest safe supply voltage under which the system can operate without crashing. As shown in Figure 2.2, applications

have different undervolting levels for V_{min} and $V_{safeMin}$, which means different amounts of energy can be saved through undervolting while working with different applications. We observe a significant voltage guard-band whose margin varies from one application to another. As shown in Figure 2.2, we have more voltage guard-bands in Matrix Multiplication in when compared with the guard bands in other applications which means we can save more energy in case of MM.

Reliability of application R(t) at time t is the probability that there is no failure in the system until time t. We find R(t) where t is the execution time in equation 3.2.

$$R\left(t\right) = 1 - P_f\left(t\right) \tag{2.2}$$

The failure rate is obtained using Weibull lifetime reliability model, a well-accepted model for transient and permanent soft errors as in equation 3.3 [65]. Since we consider undervolting at a fixed frequency, the failure rate model is a function of supply voltage [93].

$$R\left(t\right) = e^{-\lambda t} \tag{2.3}$$

The failure rate calculated for different applications is shown in Figure 2.3, where X-axis represents the undervolting level and Y-axis represents the failure rate per minute. The failure rate of applications BlackScholes, FFTD2D, Histogram, FFTD3D, Mergesort and cuBLAS-MM are obtained experimentally. As shown in Figure 2.3, V_{min} for CUDA applications at a specific level of undervolting are different. In [46], it is observed that programs have different activity patterns which can lead to different voltage droops. The

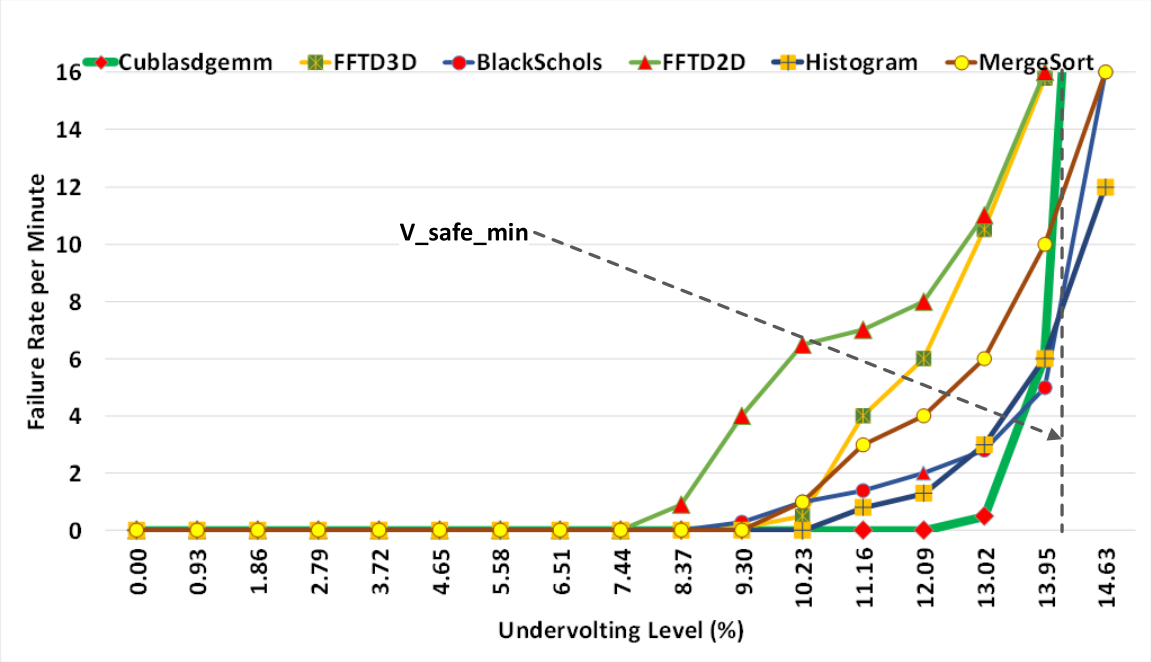

Figure 2.3: Failure rate of different Rodinia benchmarks and cuBLAS-MM from cuBLAS library

voltage droop is the main reason of GPU voltage noise. So, at a specific voltage, different intra and inter-kernel activities can lead to different failure rates. It is shown that the voltage noise, and specifically $\frac{di}{dt}$ droop, has the largest impact on $V_{safeMin}$ in [46]. Microarchitectural events, such as cache misses, cause pipeline stalls and large $\frac{di}{dt}$ droops lead to different guard-bands and $V_{safeMin}$. Because cuBLAS-MM is highly optimized, and all GPU components are active most of the time, there is no large $\frac{di}{dt}$ droop which could lead to lower voltage noise margin and larger guard-band.

2.2 GreenMM: Energy Saving Methodology

GreenMM introduces an adaptive FT-cuBLAS-MM algorithm; which aggressively saves energy and power on GPUs through undervolting with a negligible performance over-

head. GreenMM works with NVIDIA GPUs irrespective of the underlying GPU architecture. Figure 2.4 shows the overview of GreenMM. GreenMM finds the maximum level of undervolting for the underlying GPU and configures the adaptive FT-cuBLAS-MM to tolerate the potential faults with regards to the failure rate of the underlying GPU at the maximum level of undervolting. To find the failure rate of cuBLAS-MM, GreenMM reduces the voltage of GPU progressively up to $V_{safeMin}$ and according to the fault model which is described in Section 2.1 find the failure rate of the GPU at each undervolting level. Then, based on the failure rate and execution time of given matrix, estimates the number of faults. Since these phases should be done before MM computation, execution time of MM is not determined. So, GreenMM uses an estimation model to predict the execution time of any arbitrary size. With multiplying the estimate execution time and failure rate of the GPU, the number of faults is determined and now we can configure the FT-cuBLAS-MM. It uses NVML library commands to reduce voltage of the GPU by changing the GPU target power limit and voltage offset.

2.2.1 Offline Profiling

GreenMM finds the optimum working voltage of the GPU for cuBLAS-MM, going beyond the V_{min} and correcting the potential errors. Incorporating fault tolerance mechanism increases the execution time, which in turn increases the energy consumption. GreenMM, carefully calibrates the level of undervolting so that the energy saving is more than the energy overhead. Optimum working voltage is found through an offline profiling phase which is done only once for each GPU. Offline profiling creates the failure rate model and MM

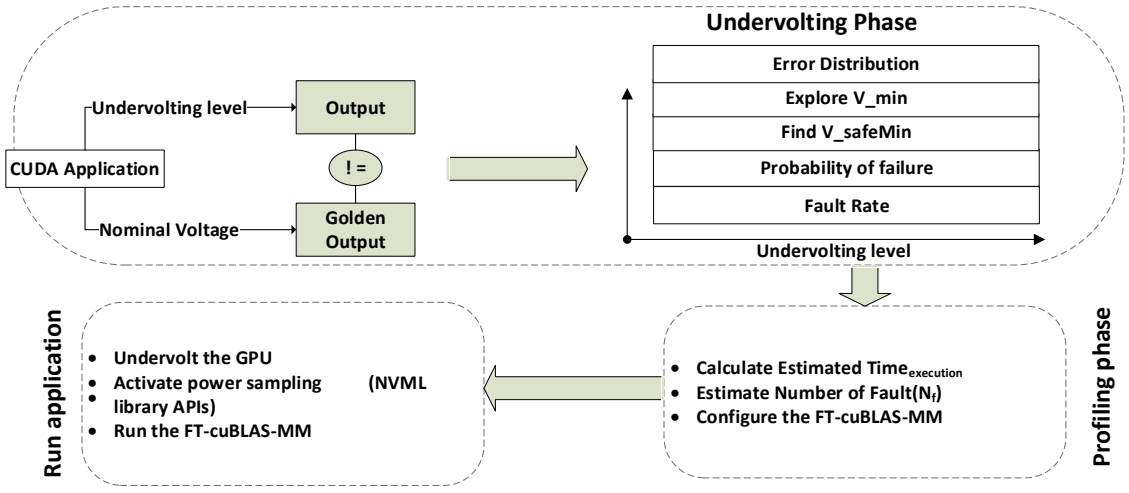

Figure 2.4: GreenMM overview

execution time estimation model to estimate the number of faults for any MM sizes with regards to the underlying GPU. The offline profiling phase which is shown in Figure 2.5 is split into two parts:

Phase 1: Determine The Maximum Undervolting Level ($V_{safeMin}$) And Fault Rate (λ)

We execute matrices of small sizes on the GPU to minimize the profiling time and obtain maximum undervolting level ($V_{safeMin}$) and fault rate (λ), as described in Section 2.1.

In GreenMM, the offline profiling phase takes into account the aging effect. Also, the effects due to process variation and temperature were explored on various applications on different GPU cards in [46]. They concluded that process variation and temperature have a relatively uniform impact on $V_{safeMin}$ across all applications in a given GPU card; and the

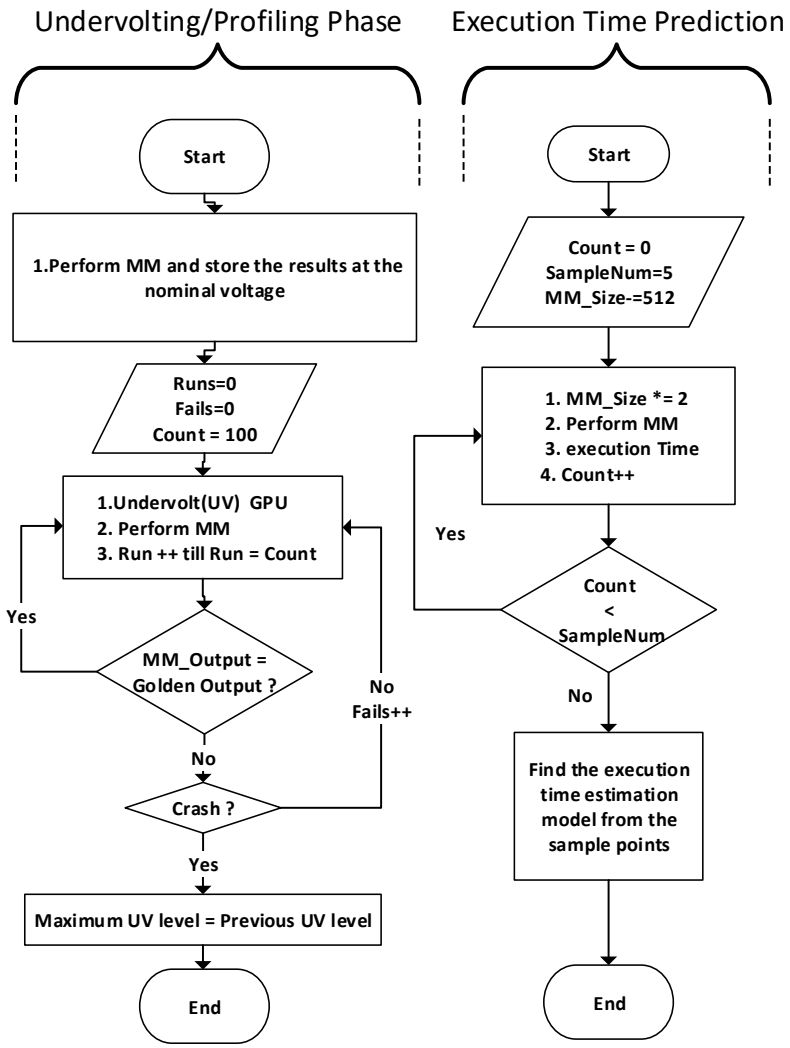

Figure 2.5: Overview of offline profiling

effect of aging is negligible (1-2 % in the long term). Moreover, The effect of temperature rise is already included in the number of faults, as plotted in Figures 2.1 and 2.9, because our fault model already considers the increase in temperature during long executions. However, we performed additional experiments to measure temperature while undervolting. Due to limited resources, we run MM in a loop to have enough time to observe the temperature changes. The variation in temperature of the GPU over time with and without undervolting

is shown in Figure 2.6. It is observed that, after running kernel continuously, the temperature of GPU remains the same after a period of time and there is only about 11 °C variation in the temperature in presence of maximum level of undervolting. Ref. [46] shows that $V_{safeMin}$ at 70 °C is about $20mV$ higher than the values at 40 °C for various GPU cards. Temperature variation in GreenMM is about 11 °C which is not big enough to make a sensible change on $V_{safeMin}$.

Phase 2: Estimate Number Of Faults Based On Matrix Size And Fault Rate (λ)

The number of faults in an application can be obtained by multiplying λ with the execution time, as shown in equation 3.5. Failure rate remains same irrespective of the input data size for a given application as in equation 3.3. Hence, we estimate the execution time of MM for a given matrix size on a specific GPU through a simple profiling.

$$T = ax^3 + b \tag{2.4}$$

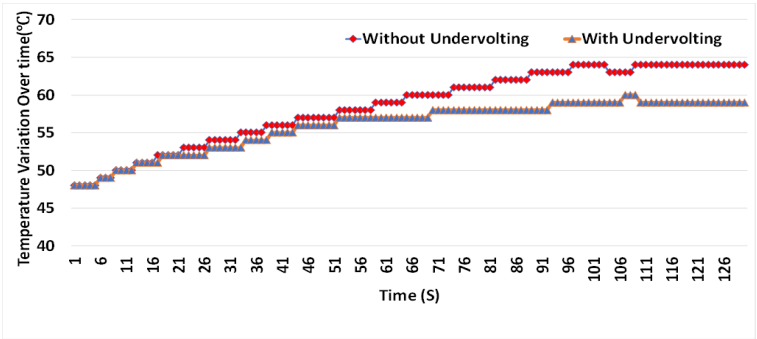

Figure 2.6: Temperature variation over time without and with maximum level of undervolting

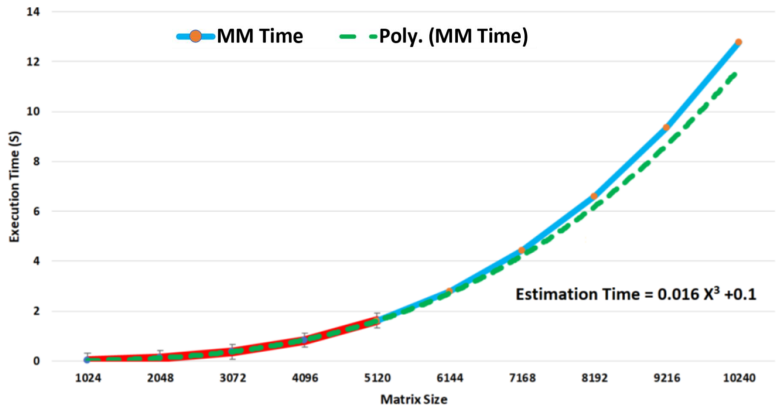

Figure 2.7: Execution time estimation model vs. the real execution time

$$F = \lambda * T \tag{2.5}$$

Due to different compute resources like SM, register file size, cache sizes and shared memory size, execution time of the MM for a given size could be vary in different GPUs. Due to memory constraints, the GPU cannot handle matrix multiplication of any arbitrary size. The time complexity of cuBLAS-MM as a function of matrix size is provided in equation 3.4, where a and b are architecture-specific constants [86] [77]. We run MM for different sizes to calculate the values of a and b for the underlying GPU. Figure 2.7 shows the experimental execution time (blue) and the execution time calculated theoretically from equation 3.4. Moreover, the red line shows sample points that were used to derive values of a and b which can be used for prediction of execution time for larger matrices, shown as green dashed line. Then, we compare the estimated execution time with the real experimental results for bigger matrices. As the results show, the estimation error is negligible.

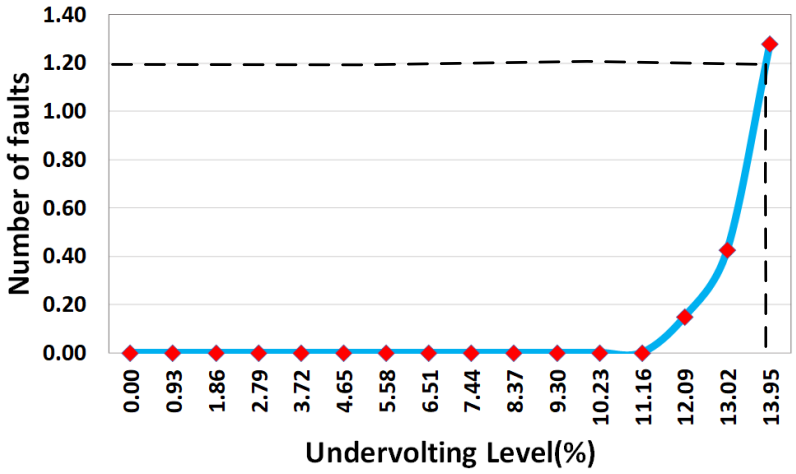

Figure 2.8: Number of faults according to the undervolting level for matrices with size of 10K on NVIDIA GTX 980

Due to memory constraints on NVIDIA GTX 980, we use matrix of size 10K for GreenMM. The $V_{safeMin}$ is 86.05% of nominal voltage (undervolting level is 13.95%); the number of faults is 1.2 as shown in Figure 2.8. Hence, FT-cuBLAS-MM should tolerate at least 2 faults with the input size of 10K * 10K. If GPU memory supports matrices bigger than 10K * 10K, they may experience more number of faults.

As the size of the matrix increases, the execution time as well as the number of faults also increase as shown in Figure 2.9. The matrix size varies between 10K and 100K as the undervolting level is changed from 0% to 13.95% in Y-axis; Z-axis shows the number of faults. For a given undervolting level, the number of faults for large matrix sizes is more than the number of faults in small matrices.. In the following, we propose an adaptive FT algorithm than can be configured to handle different number of faults.

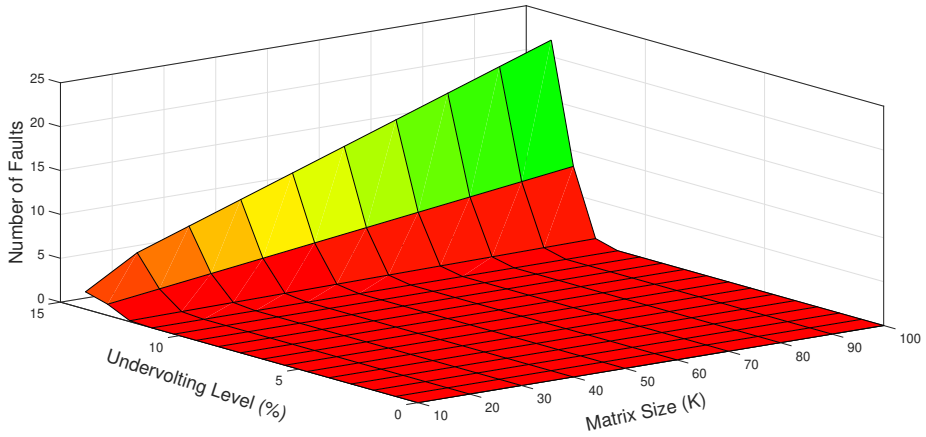

Figure 2.9: Estimated Number of faults for different matrix sizes given the undervolting levels.

2.2.2 Offline FT-cuBLAS-MM

The ABFT for Matrix Multiplication has a very low performance overhead when compared with other techniques [8]. The basic idea of ABFT is to encode input matrices with checksums to detect and correct the corrupted data. The traditional ABFT introduced by Huang et al. [41] is capable of correcting one fault by checking correctness at the very end of computation. In our work, we introduce an enhanced offline version, FT-cuBLAS-MM, which is capable of tolerating any arbitrary number of faults by increasing the number of weighted check-sum vectors. Algorithm 7 describes the pseudo-code for the offline FT-cuBLAS-MM.

Generating the weights of the checksum vectors, encoding the column checksums, and the row checksum are done according to algorithms into the matrix is done according algorithms 2, 3 and 4 respectively. The result of $C^f = A^c * B^r$ is a full checksum matrix. At the end of computation, we check full checksum relationship again and if the relationship

Algorithm 1 The pseudo-code for Detection Phase

1: Generate checksum weights vectors v_1 and v_2
2: Encode $A \rightarrow A^c$
3: Encode $B \rightarrow B^r$
4: $C^f = A^c \times B^r$
5: Recompute the checksum for C^f
6: Verify full checksum relationship of C^f

does not hold, then our result is faulty; thereafter, faults are detected and corrected using equation 2.6.

$$C_{ij} = \Sigma_{j=1}^{n} c_{ij}^{f} - \Sigma_{k=1,k\neq j}^{n} c_{ik}^{f} \tag{2.6}$$

2.2.3 Online FT-cuBLAS-MM

Offline FT-cuBLAS-MM only checks correctness of results at the end of computation. We design an online version of FT-cuBLAS-MM to check correctness of MM during computation, so that we can prevent faults to be propagated. We introduce an Online FT-cuBLAS-MM that can handle different number of faults. Fault coverage capability of FT-cuBLAS-MM is determined before starting the MM computation. However, the key problem here is that we must use MM algorithm such that it maintains the checksum

Algorithm 2 Generating weighted checksum vectors for each block

1: **for** $i = 0, 1, ..., nb$ **do**
2: $\quad v_1[i] = 1$
3: **end for**
4: **for** $i = 0, 1, ..., nb$ **do**
5: $\quad v_2[i] = 1 + i$
6: **end for**

relationship even in the middle of the computation. In [20], it is proved that outer product Matrix Multiplication maintains checksum relationship in each iteration of computation. For a matrix with size of N, we have at most N opportunities to tolerate faults during the entire MM computation. The fault detection phase, which is always active, increases the performance overhead. So, to achieve high performance, we can invoke the FT-cuBLAS-MM routine once in every several iterations. There is a trade-off between the number of iterations and overhead of online FT-cuBLAS-MM. The number of iterations to invoke FT-cuBLAS-MM is closely related to the number of faults that may happen during the computation. If the failure rate of system increases, then we should check more frequently, otherwise, there is no need to employ an algorithm with higher fault coverage capability. The algorithm to perform MM has several steps. The detailed steps of the algorithm are shown in algorithm 5.

During each iteration, we update checksum of the result matrix to maintain full checksum relationship. Then, we compute sum of each row and column in the result matrix and compare it with the row and column checksum. If the check is passed we move to the next iteration, otherwise, if any checksum does not match, we locate the exact position of error through comparing the checksums. To correct the error (C_{ij}), we simply add the difference of j_{th} checksum column and the sum of j_{th} column to the result matrix element at location (i, j). GreenMM corrects two errors at the same time regardless of the error patterns. Also, it corrects any number of errors which may happen in the same row or column.

Algorithm 3 Column checksum update for A(m * k)

1: **for** $j = 0, 1, ..., k-1$ **do**
2: **for** $i = 0, 1, ..., m-1$ **do**
3: $ColChk_{v1}[j] = \Sigma_{i=0}^{m-1} V_1[i] * A[i][j]$
4: $ColChk_{v2}[j] = \Sigma_{i=0}^{m-1} V_2[i] * A[i][j]$
5: **end for**
6: **end for**

As shown in algorithm 5, online FT-cuBLAS-MM algorithm consists of following steps:

1. Move input matrices to the GPU using cudaMemCpy API.

2. Generate checksum weights vectors in the CPU and move them to the GPU. The weights are generated according to algorithm 2. Due to frequent accesses to weights vector in GPU, to get peak performance, pitched device memory is allocated using cudaMallocPitch API that allocates linear memory space for better efficiency in terms of performance and power.

3. Divide input matrices into blocks given the number of faults and do MM without checksums.

4. Invoke cuBLAS-MM to update column checksum for each block according to algorithm 3.

Algorithm 4 Row checksum update for B(k * n)

1: **for** $i = 0, 1, ..., k-1$ **do**
2: **for** $j = 0, 1, ..., n-1$ **do**
3: $RowChk_{v1}[i] = \sum_{j=0}^{n-1} V1[j] * B[i][j]$
4: $RowChk_{v2}[i] = \sum_{j=0}^{n-1} V2[j] * B[i][j]$
5: **end for**
6: **end for**

Algorithm 5 Pseudo-code for online FT-cuBLAS-MM

1: Initialization
   NB = N (Matrix Size)/ B (Block Size)
2: **for** $i = 1, ..., NB$ **do**
3: $A_B, B_B \rightarrow GPU$
4: Update $C_B \rightarrow$ cuBLAS-MM(A_B, B_B, C_B)
5: Update ${A_B}^c \rightarrow$ cuBLAS-MM$(A_B, ColChk_v, A_B^c)$
6: Update $C_B^c \rightarrow$ cuBLAS-MM(A_B^c, B_B, C_B^c)
7: Update $B_B^r \rightarrow$ cuBLAS-MM$(B_B, RowChk_v, B_B^r)$
8: Update $C_B^r \rightarrow$ cuBLAS-MM(A_B, B_B^r, C_B^r)
9: Recalculate $\rightarrow$ $C_B\_ColChk2$
10: **while** $C_B\_ColChk1 \neq C_B\_ColChk2$ **do**
11: Do Correction
12: **end while**
13: Recalculate $\rightarrow$ $C_B\_RowChk2$
14: **while** $C_B\_RowChk1 \neq C_B\_RowChk2$ **do**
15: Do Correction
16: **end while**
17: Update C
18: **end for**
19: $C \rightarrow CPU$

5. Invoke cuBLAS-MM to update C

6. Invoke cuBLAS-MM to update row checksum of B given the equations described in algorithm 4.

7. Update row checksum of C by invoking cuBLAS-MM

8. Recalculate column and row checksums of C by invoking a simple kernel which adds elements of the result matrix.

9. Compare recalculated checksums and old checksums to locate the potential error. Any potential errors can be located by comparing the column and the row checksums. Since computers do floating point calculations in finite precision, the checksum relationship can not hold exactly due to round-off errors. So, we need a threshold to distinguish

between round-off errors and computation errors. Too large thresholds may hide the computation errors, while, too small thresholds may interrupt correct computation. In comparison phase, according to [99], e^{-10} has been chosen as a conservative threshold to distinguish between round-off and computation errors.

10. Correct any potential errors according to equation 2.6

2.3 Evaluation

2.3.1 Experimental Setup

All experiments are performed on NVIDIA GTX 980, the architectural specifications can be found in Table 2.2. Given the limited memory size of the GPU, we were able to evaluate the results for up to a matrix size of 10K. We reduced nominal voltage of the GPU in step sizes of 10 - 12mV until the $V_{OSCrashpoint}$ using the MSI After Burner [46]. By decreasing the target power limit of the GPU, we can enforce specific operating voltage. We use NVIDIA System Management Interface (Nvidia-smi), a widely used command line utility on top of NVIDIA Management Library (NVML), to measure power consumption of the GPU at 10ms intervals. Some important commands on power management in NVIDIA GPUs are shown in Table 2.1. The execution times of cuBLAS-MM and FT-cuBLAS-MM are shown in Figure 2.10. The overhead of fault tolerance is large for small matrices, however, the overhead decreases with increase in the matrix size. For small matrices, there is 8% performance overhead, while in case of bigger matrices (10K), performance overhead of FT-cuBLAS-MM comes down to 1.5%.

There is no need for fault tolerance till V_{min} as the probability of error occurrence is zero. The detection phase is activated when undervolting beyond V_{min} to detect potential errors, however, the correction phase is activated only if an error is detected in the detection phase. The detection phase accounts for majority of the overhead in the FT-cuBLAS-MM. For instance, when the matrix size is 10K, the detection phase takes $139ms$ while the correction phase takes only $0.24ms$, which means the number of faults to be corrected has low impact on the performance. In case of 10K matrix size, the maximum number of faults we need to tolerate is 1.2 as shown in Figure 2.8; which can be handled by offline FT-cuBLAS-MM. Offline FT-cuBLAS-MM is a special case of online FT-cuBLAS-MM when the number of faults is less than or equal to 2. Here, the block size is the same as the matrix size.

2.3.2 Performance and Energy Saving Evaluation of FT-cuBLAS-MM

When the matrix size increases, the failure rate remains the same. However, the number of errors increases. To evaluate the overhead of FT-cuBLAS-MM, faults are injected directly into partial sum results at random locations and in random iterations according to fault model described in Section 2.1. Fault injection in a controlled manner emulates the impact of hardware transient faults on MM computation. We observed errors in the

Table 2.1: Power management commands using the NVML library

Command	Description
nvmlDeviceGetPowerUsage	Retrieves power usage for the GPU and its associated circuitry in milliwatts
nvmlDeviceSetPersistenceMode	Enables persistent mode to prevent driver from unloading
nvmlDeviceSetPowerManagementLimit	Sets new power limit for the device
nvmlDeviceSetApplicationsClocks	Sets clocks that applications will lock to
Accuracy	Power Measurement Accuracy & Reading is accurate to within +/- 5% of the current power draw

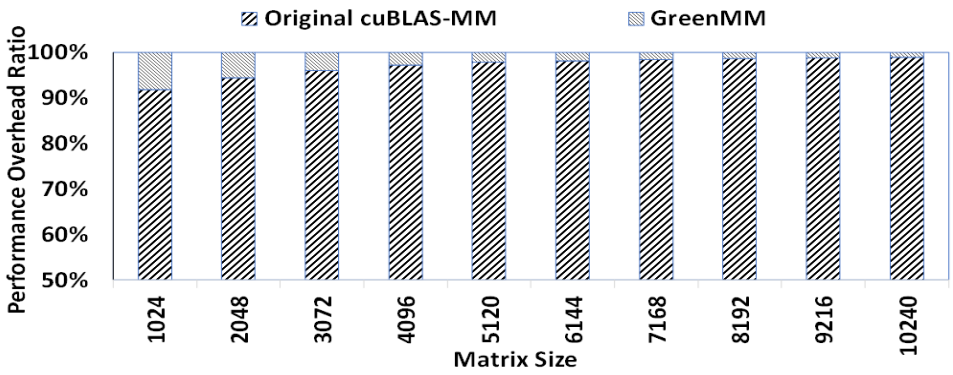

Figure 2.10: Performance overhead of matrix multiplication for different matrix sizes in presence of two errors

output, however, they were detected and corrected by the offline or online FT-cuBLAS-MM depending on the size of matrix and the number of faults.

FT-cuBLAS-MM as described in Section 2.2, improves the reliability of computation and tolerates any arbitrary number of faults. This is because, we check correctness of the partial results in each iteration. There is a trade-off between reliability of computation, energy consumption and performance overhead. We measure the performance ($GFLOPS$) of the cuBLAS-MM and FT-cuBLAS-MM on NVIDIA GTX 980 in the presence of different number of faults for a 10K matrix. Since the actual number of faults at $V_{safeMin}$ (i.e. 13.95%) undervolting level is 1.2, we evaluate the performance overhead by manually injecting faults

Table 2.2: NVIDIA GTX 980 specifications

Processor	2048 CUDA-core NVIDIA Maxwell GeForce GTX 980
Peak Perf.	4.6 TFLOPs
Memory	4 GB GDDR5
Base Clock	1126 MHz
Boost Clock	1216 MHz
Memory Clock	7 GHz
Default Voltage	1.075 V

into 10K matrix. Increase in the number of faults results in increased performance overhead, as shown in Figure 2.11. The performance is 165 $GFLOPS$ in presence of 2 errors and 162 $GFLOPS$ in presence of 16 faults. On average, the performance overhead for different number of faults is 1.5%.

The energy consumption of the GPU is calculated by multiplying power (at each undervolting level) with the execution time of MM. Figure 2.12 shows the energy saving in FT-cuBLAS-MM versus the original cuBLAS-MM in presence of different undervolting levels and number of faults. Since no fault occurs till V_{min}, fault detection phase is disabled.

The fault detection and correction phases are activated when we undervolt from V_{min} till $V_{safeMin}$. The X-axis denotes the undervolting level and the corresponding number of faults. Figure 2.12 (a) shows the energy saving at different undervolting levels for matrix size of 10K with and without fault tolerance. Undervolting level at V_{min} for the original cuBLAS-MM is 10.23% without any faults. Undervolting beyond V_{min} results in faults; and the maximum number of faults is 1.2 at undervolting level of 13.95% as shown in Figure 2.8. So, offline FT-cuBLAS-MM is used to correct the faults. For a matrix of size 10K,

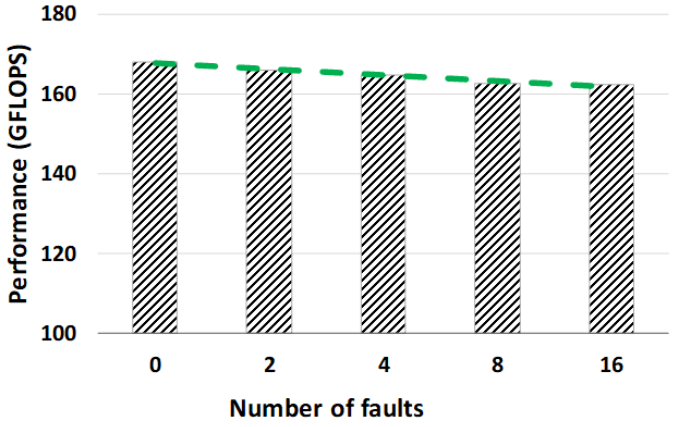

Figure 2.11: Performance evaluation of the FT-cuBLAS-MM

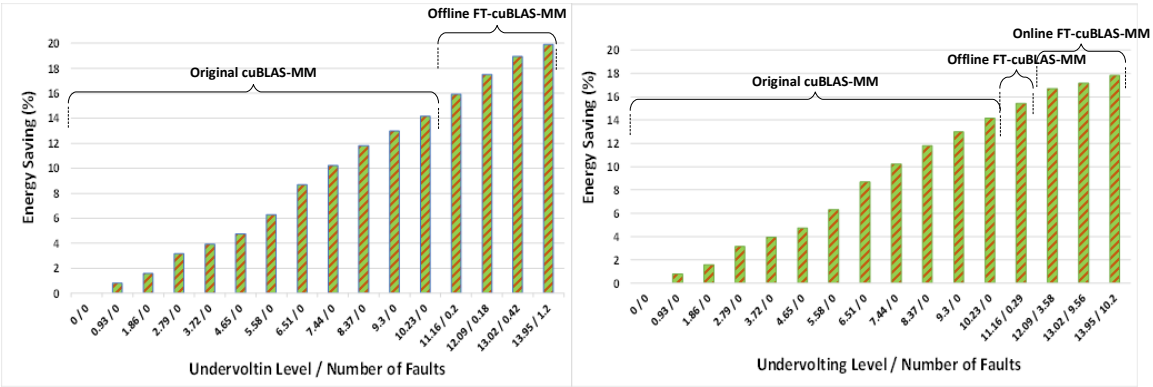

Figure 2.12: Energy saving in the FT-cuBLAS-MM versus the original cuBLAS-MM given different undervolting levels and number of faults

the cuBLAS-MM can save energy up to 14% just by undervolting and without any fault tolerance, but with GreenMM the energy saving is increased up to 19.8% due to undervolting beyond V_{min}.

Figure 2.12 (b) shows the energy saving at different undervolting levels for matrix size 40K with and without fault tolerance. Undervolting beyond V_{min}, results in faults; and the maximum number of faults is 10.2 at undervolting level 13.95%. Offline FT-cuBLAS-MM with two weighted-check sum vectors can not cover those number of faults. Hence, we activate the Online FT-cuBLAS-MM to tolerate the faults. Although, going beyond V_{min} results in more number of faults, we can still save 4% additional energy by activating the FT-cuBLAS-MM. For a matrix of size 40K, when there are no faults, the cuBLAS-MM saves energy up to 14% with undervolting, but when using undervolting in combination with FT-cuBLAS-MM the energy saving increases to 18%.

2.3.3 Performance/Watt and Total Energy Consumption Evaluation

cu-BLAS-MM is not open source; so, the number of operations cannot be calculated accurately; however, the number of floating point operations that take place when multiplying 2 matrices can be estimated according to equation 2.7.

$$N_{fp} = 2n^3 - n^2 \tag{2.7}$$

With assuming the same amount of operations in both cases (with and without ABFT), and measuring the extra execution time which is needed for ABFT part, we can have a fair comparison.

As shown in Figure 2.13, despite the performance overhead of ABFT, GreenMM has higher performance per watt ($GFLOPS/Watt$) in comparison to the original cuBLAS-MM. This is because, we can save significant power by just undervolting the GPU. Figure 2.13 shows $GFLOPS/Watt$ of the GPU. X-axis shows the number of faults and the Y-axis shows $GFLOPS/Watt$ improvement ratio when compared with the performance of the original

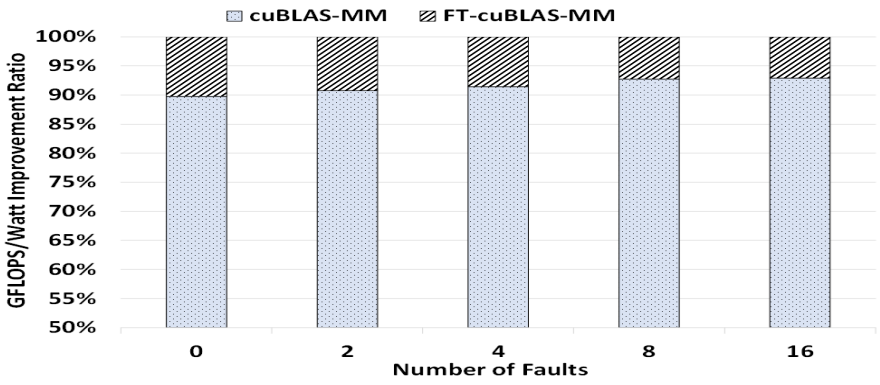

Figure 2.13: Comparing performance in GPU with default voltage versus undervolted GPU in presence of different number of faults.

cuBLAS-MM without undervolting. When there are two faults, at 13.95% undervolting level, GreenMM improves $GFLOPS/Watt$ of the GPU by 9%. When the number of faults increases to 16, there will be 7% improvement in $GFLOPS/Watt$ over the original cuBLAS-MM without undervolting. MM heavy applications such HPL and ScaLAPACK involves a time-consuming task to deal with MM computation. Trailing matrix updates consumes more than 90% of the computation cost in HPL [97]. GreenMM can be employed to compute this phase. Since GreenMM is transparent to the users, it can be integrated into HPL and other MM heavy applications supporting GPUs.

Chapter 3

SAOU: Safe Adaptive Overclocking and Undervolting for Energy-Efficient GPU Computing

3.1 Fault Model

As clock frequency of the system is pushed beyond the maximum clock frequency, system may experience errors due to timing faults. It is because with increasing the clock frequency, a circuit node may not have enough time to fully charge and discharge the load capacitance. Therefore, increasing frequency leads to higher probability of logic failure [62]. In this section, we develop a realistic model for error probability in different applications at a given frequency.

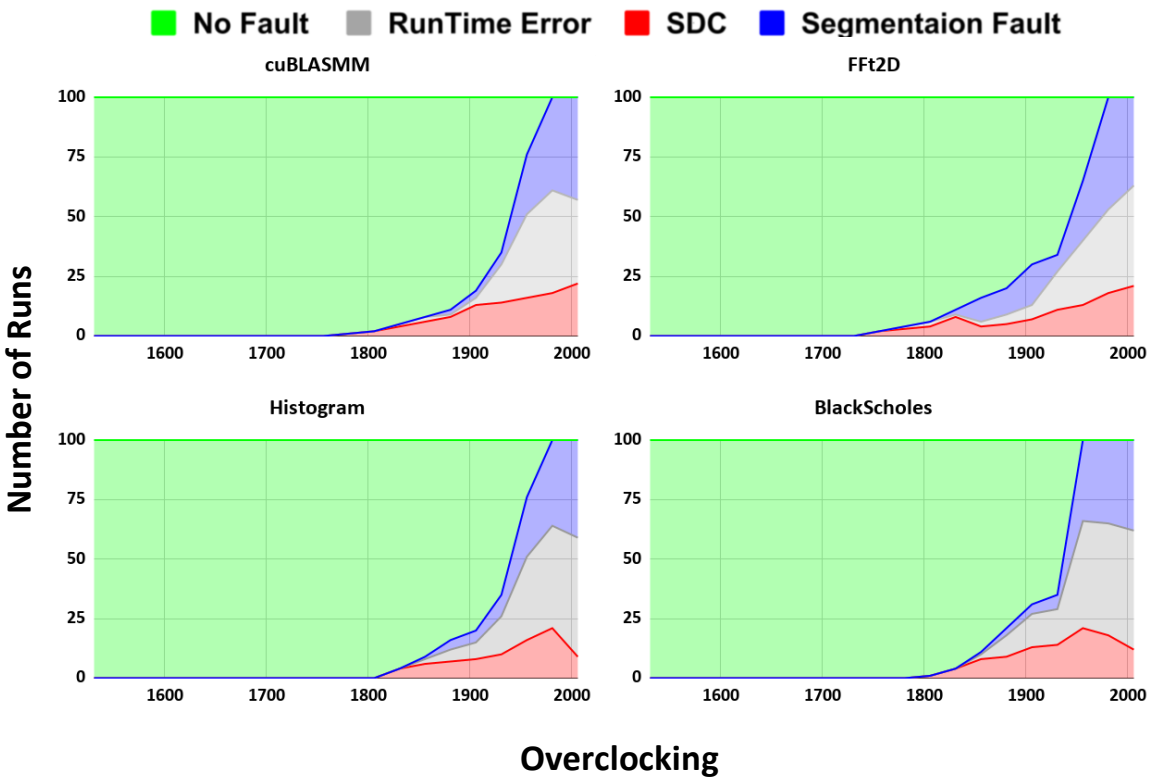

Figure 3.1: Fault distribution w.r.t. overclocking

The probability of failure is given by,

$$P_f = \frac{Number\ of\ failures}{Number\ of\ application\ runs} \tag{3.1}$$

We define f_{max}, as the maximum nominal frequency of the GPU at which the program executes correctly and $f_{safeMax}$ as the theoretical highest safe frequency under which the system can operate without crashing. Reliability of application R(t) is the probability that there is no failure in the system during the execution time t.

$$R\left(t\right) = 1 - P_f\left(t\right) \tag{3.2}$$

The failure rate is obtained using Weibull lifetime reliability model, a well-accepted model for transient and permanent soft errors as in equation 3.3 [65].

$$R(t) = e^{-\lambda t} \tag{3.3}$$

Hence, we calculate the λ according to execution time t and R(t), which are measured through the profiling phase. λ gives the number of errors per minute, which will be used to determine the number of sufficient checkpoints at a given frequency.

We executed applications from Rodinia benchmark [11] and cuBLAS library are executed on Nvidia GTX 980 GPU at different frequencies. We increase the frequency starting from 1404 MHz, which is the default maximum frequency of the Nvidia GTX 980 GPU. At each frequency, using the MSI After Burner, we keep the frequency constant during the execution time. The frequency is increased in steps of 25 MHz. To estimate the failure rate, we run each application 100 times at each level of overclocking, and the corresponding output is compared with the golden output which is extracted by running the application at default frequency. If the output does not match with the golden output, then the application has experienced a failure during the execution. The failure rate calculated for for applications is shown in Figure 3.2, where X-axis represents overclocking level and Y-axis represents failure rate per minute. Applications have different failure rates due to different activity patterns they experience during the execution time. different activity patterns which can lead to different voltage droops. The voltage droop is the main reason of GPU voltage noise. So, at a specific voltage, different intra and inter-kernel activities can lead to different failure rates. It is observed that the voltage noise, and specifically $\frac{di}{dt}$ droop, has the largest

impact on $V_{safeMin}$. Microarchitectural events, such as cache misses, cause pipeline stalls and large $\frac{di}{dt}$ droops lead to different guard-bands and $V_{safeMin}$. Because cuBLAS-MM is highly optimized, and all GPU components are active most of the time, there is no large $\frac{di}{dt}$ droop which could lead to lower voltage noise margin and larger guard-band [46].

Applications have different failure rates due to different activity patterns they experience during the execution time [46]. It is observed that increasing the frequency or lowering the voltage results in less timing margin and higher error probability [46]. At a given frequency, the applications have different failures due to micro-architectural events such as cache misses, cause pipeline stalls which can cause voltage noise or timing errors. Because cuBLAS-MM is highly optimized, and all GPU components are active most of the time, so there is no large di/dt droop which could lead to lower voltage noise and as a result timing error.

To find sensitivity of overclocking, we also record error distribution of different applications during the overclocking. Silent data corruption (SDC), CUDA run-time errors and OS crash are among notable types of errors that system may experience during overclocking. SDC refers to when program finishes its execution normally without any error messages, however, producing a wrong result. In profiling phase, errors can simply be detected by comparing results against the golden output. CUDA run-time errors including segmentation faults and driver faults are detected through standard error output. OS crash happens when undervolting level is beyond the $V_{safeMin}$. Since, SDC errors can be covered through fault-tolerant algorithm, we only focus on SDC errors and we do not go for the frequencies which can cause errors other than SDCs. Figure 3.1 shows behavior of different

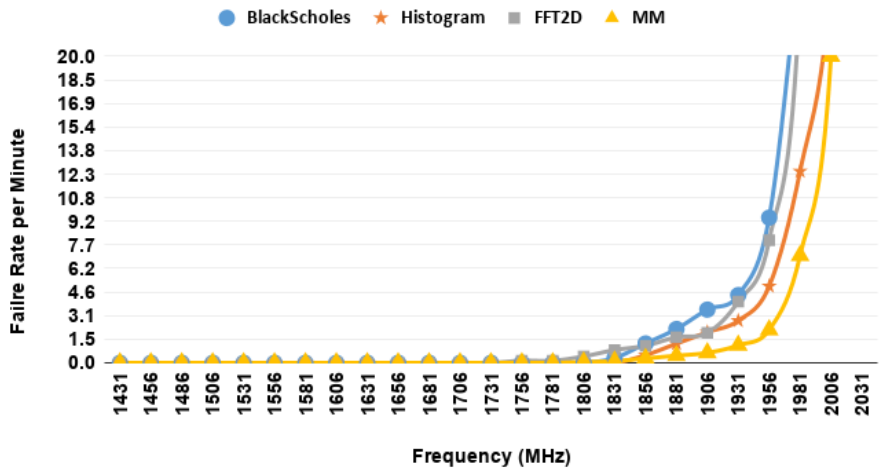

Figure 3.2: Failure rate of different Rodinia benchmarks and cuBLAS-MM from cuBLAS library w.r.t overclocking

applications including cuBLAS-MM, FFTD2D, BlackScholes, and Histogram in regard to overclocking. X-axis denotes frequency and Y-axis denotes fault types. As shown in Figure 3.1, due to different activity patterns which explained earlier, we observe different SDC rates. For instance, cuBLAS-MM, and FFT2D have larger SDC rate compared to Histogram and BlackScholes.

The failure rate of different applications due to undervolting are measured similar to the approach used in [105]. As shown in Figure 3.3, failure rate of application increases exponentially. This means, even, if all types of errors are in SDC form, which can be covered through a FT algorithm, we should checkpoint more frequently which incurs a huge performance overhead. So there is no need to reduce voltage if number of checkpoints is going to be high. In MM application, according to fault distribution and failure rate model, we only checkpoint at the end of each thread and we are able to correct the potential errors due to undervolting. Our resilience technique which is used to cover the potential errors, is

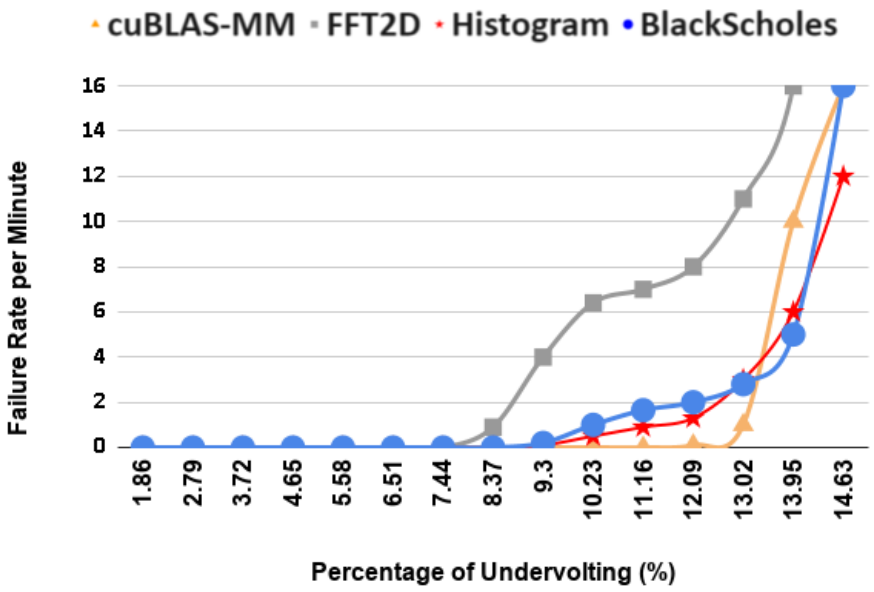

Figure 3.3: Failure rate w.r.t undervolting

discussed in section 3.2.

Our aim is to carefully calibrate the level of undervolting and overclocking so that the energy saving is more than the energy overhead. Voltage and frequency are estimated through an offline profiling phase which is done only once for each GPU. The offline profiling phase is split into two phases:

Phase 1: Extracting The Overclocking And Undervolting Level And Error Rate (λ)

We execute matrices of small sizes on the GPU to minimize the profiling time and obtain maximum tolerable overclocking and undervolting level, and fault rate (λ) as described in Section 3.1.

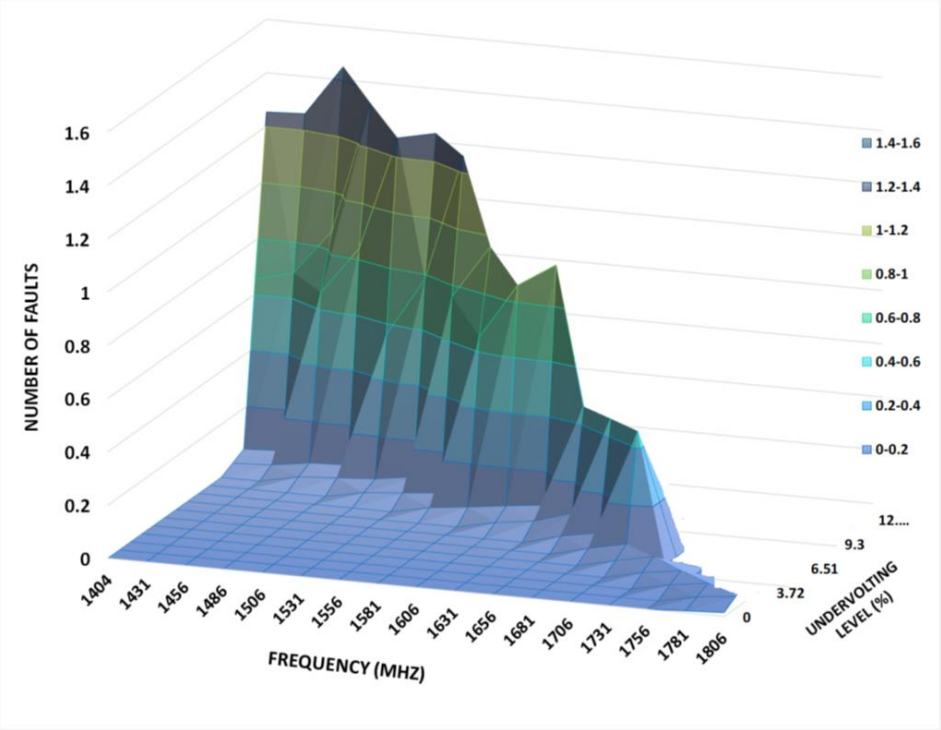

Figure 3.4: Number of faults in regard to overclocking and undervolting for matrix with size of 10K

Phase 2: Estimate The Number Of Faults

The number of faults in an application can be obtained by multiplying λ with the execution time, as shown in equation 3.5. Failure rate remains same irrespective of the input data size for a given application as in equation 3.3. Hence, we estimate the execution time of MM for a given matrix size on a specific GPU through a simple profiling.

$$T = \alpha * (ax^3 + b) \tag{3.4}$$

$$F = \lambda * T \tag{3.5}$$

Due to different compute resources like SM, register file size, cache sizes and shared memory size, execution time of the MM for a given size could vary in different GPUs. Due to memory constraints, the GPU cannot handle matrix multiplication of any arbitrary size. The time complexity of cuBLAS-MM as a function of matrix size is provided in equation 3.4, where a and b are architecture-specific constants [86] [77]. After finding these constant values we can estimate the execution time of the given matrix. Multiplying the failure rate and estimated execution time provides the number of faults. We find the number of faults for different pairs of frequency and voltage as shown in Figure 3.4 for matrix multiplication with input size of 10K. X-axis, Y-axis, and Z-axis shows undervolting level (%), frequency (MHz) and number of faults respectively. For matrix of size 10K, on average, we observe 1.45 faults during the execution.

3.2 Checkpoint and Recovery

A checkpoint is a snapshot of a system state, including stack, heap, global and register values. It also keeps the copy of the contents of application process address space. In CPU domain, several checkpoint and recovery (CR) techniques have been developed at different levels including kernel, library, and application level. It is not feasible to extract the mentioned information in kernel and library level. This is because, GPU is handled by driver rather than the operating system and there is no available API to access the required information during the execution. As a result, in a faulty case, it is not possible to reload threads computing state inside the kernel and resume the execution. Therefore, checkpoint and recovery techniques usually relaunch kernel from the beginning [35] [90] [68]. We can

overcome this by modifying the application code to keep track of the necessary information. Hence, we introduce a technique that is able to recover only the corrupted information.

We adopt an incremental checkpointing [76] which preserves only the data that has changed since the last checkpoint. In matrix multiplication, we only checkpoint partial results which are updated during execution. Each thread block (TB) is responsible for a chunk of matrix. Depending on the size of input matrix and TB size, input matrices are divided into several TBs, all of which, might not be accommodated on the GPU at a time. In GTX 980 GPU, the maximum number of threads is 1024 per TB and 16 SMs which can execute 16 TBs. If we decrease number of threads within the thread block, we can accommodate more number of TBs in GPU at a time. Due to constraints on GPU resources (16 SMs, and 128 CUDA cores/MP), we can have limited number of TBs at a time. After executing each TB, GPU replaces the TB with other TBs in the queue. However, as shown in Algorithm 6, before replacing the executed TB, we check the correctness of the partial results within the TB. If no error is detected, we checkpoint the partial results of the current TB. To reduce the performance penalty, we start with storing them in private memory space which is local to each thread. If there is not enough space in private memory, data will be pushed back into global memory. In case of matrix multiplication, according to failure rate model, it is sufficient to only checkpoint at the end of each TB. But for bigger matrices, due to higher probability of faults, we might need to checkpoint more frequently. Proposed checkpoint and recovery technique incurs very small overhead on the energy and performance in comparison with the baseline. When TB size is 1024 threads, energy and performance overheads are 0.5% and 0.4% respectively.

Algorithm 6 High level pseudo-code for execution process of CR-enabled matrix multiplication

```
1: Initialize()
2: cudaMalloc(&d_A, size A);
3: cudaMalloc(&d_B, size B);
4: cudaMalloc(&d_C, size C);
5: cudaMemcpy(d_A, h_A, size A, hostToDevice);
6: cudaMemcpy(d_B, h_B, size B, hostToDevice);
7: Invoke MM-Kernel();
8: _syncthreads();
9: if (results!=correct) then
10:    restore();
11: else
12:    checkpoint();
13: end if
14: Go to 7
```

3.2.1 Implementation Details

The state of GPU application can be represented by variables declared in the program. However due to the complex memory hierarchy of the GPU, those variables are spread in different memory locations: register, local memory, shared memory and global memory.

As shown in algorithm 6, to ensure an appropriate in-kernel checkpoint and recovery, all threads synchronize before checkpoint and during recovery. Due to memory inconsistency, it is not guaranteed that all threads finish at the same time. In other words, it is mandatory for all threads to participate in checkpointing. As an example, when thread X is checkpointing, thread Y can update a shared variable between them. CUDA supports synchronization between threads within a TB with a built-in instruction called "syncthreads()".

The same problem exists if we extend this to threads of different TBs which share a global variable. Due to lack of any built-in synchronization mechanism between different

TBs, similar problem might occur [1]. To solve synchronization problem between TBs, we propose an algorithm which synchronizes TBs before checkpointing. As shown in Figure 3.5, we define array of global variables which can be accessed by all TBs. The array length depends on the number of TBs that can run at a time on GPU. Before checkpoint module, we use instruction "synchthreads()" to synchronize the threads within the TB and make sure all of them are finished. Once, all threads within the TB are completely executed, we set the global variable corresponding to the TB to "1". Now, to solve the memory inconsistency between different TBs, we ensure all TB flags are set to "1". Therefore, threads within different TBs will not modify the shared global value anymore.

Checkpoint and recovery is embedded into the code through a pre-compiling phase at compile time. Since the GPU drivers are closed source, the modifications are done at the application level. During pre-compiling phase, we transform application source code into a format where run-time support calls are inserted for constructing computation state. In pre-compiling phase, we follow the below tasks:

1) Buffer allocation: Create buffers to hold GPU state in device memory.

2) Insert checkpoint location

3) Synchronize GPU threads before checkpointing

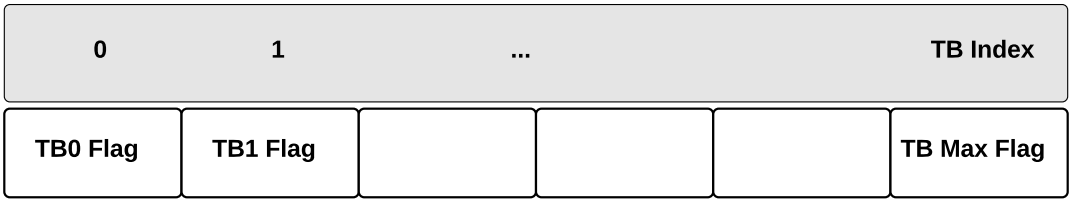

Figure 3.5: Thread block level synchronization using array of global variables

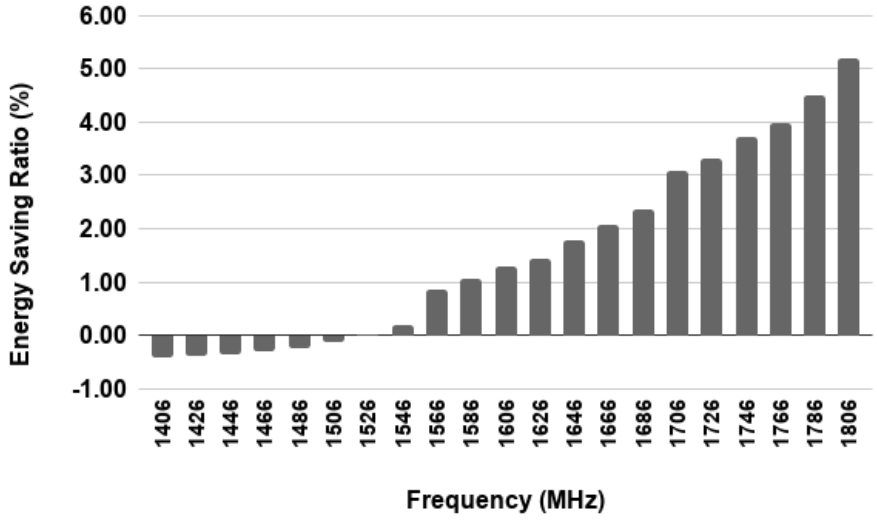

Figure 3.6: Energy saving ratio in regard to overclocking in presence of CR algorithm

4) Copy variables from GPU local, shared and global memory to allocated buffers which can be register, private memory or device global memory.

At run-time, as shown in algorithm 6, when the execution stream reaches a checkpoint, it synchronizes threads in all TBs and checks for faults. If at least one thread detects an error, the entire block goes to restoration phase. Otherwise, it goes to checkpointing phase. In restoration phase, all threads within corrupted block jump to previous checkpoint label. Then, each thread, copies threads private information from its backup to its original location. The shared memory is also reloaded from its backup located in device memory. In case of matrix multiplication, due to different performance penalties, registers, private memory, and device global memory are in priority for storing the partial results. If there is not enough registers and private memory, we save the computation states into device global memory. During the checkpoint and recovery, we need to check for the correctness of the computation. Similar to approach used in GreenMM [105], we use checksum data

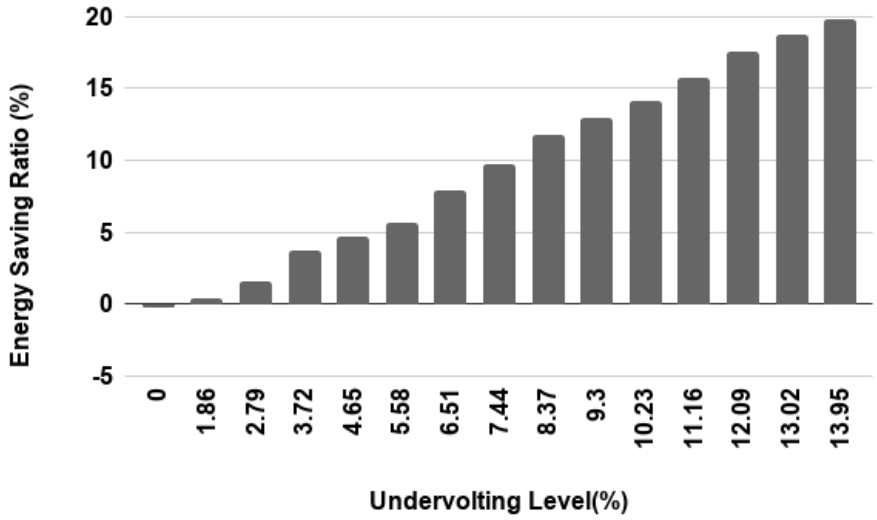

Figure 3.7: Energy saving ratio in regard to undervolting in presence of CR algorithm

redundancy for detection phase [41]. To compute C=A*B. Algorithm 7 describes pseudo-code for detection phase. A^c and B^r are encoded input matrices and C^f is a full checksum matrix [20]. At the end of each iteration right before checkpoint location, we recompute checksum again. If checksum relationship does not hold up, the computation is faulty and recovery phase will be invoked.

Algorithm 7 The pseudo-code for error detection phase

```
1: Initialize(A^c and B^r)
2: C^f = A^c× B^r
3: Recompute C^f from C
4: if (C^f doesn't maintain checksum relationship) then
5:     error_detected();
6: else
7:     no_error_detected();
8: end if
```

3.3 Evaluation

3.3.1 Experimental Setup

All experiments were performed on NVIDIA GeForce GTX 980 GPU. For matrix multiplication application, matrix with size of 10K was considered due to memory constraints (4 GB RAM). The GPU overclocking and undervoling was done using MSI After-Burner [2]. For undervolting, the power budget was reduced to enforce the GPU to operate at a specific voltage. For overclocking, the memory frequency was set to its default value and only the core clock frequency was modified. NVIDIA System Management Interface (Nvidia-smi) was used to monitor GPU utilization and report power consumption of every 10ms. We evaluate the energy savings of MM due to overclocking only, undervolting only and combination of overclocking and undervolting considering the power consumption and execution time.

We used matrix multiplication application (cuBLAS-MM) as it is a key sub-routine for many scientific applications like HPL and ScaLAPACK [97] [7]. For instance, MM constitutes of more than 90% of the computation cost in HPL [97]. Our proposed method can easily be integrated into these applications to save considerable amount of energy. Similar approach can be used while implementing checkpoint and recovery (CR) for different applications in order to save energy due to fusion of overclocking and undervolting.

3.3.2 Results

Figure 3.6 shows percentage of energy reduction in a matrix size of 10K, compared to the original execution of without overclocking or CR. As shown in Figure 3.6, at default frequency, there is negligible energy overhead of about 0.5%. However, as we continue

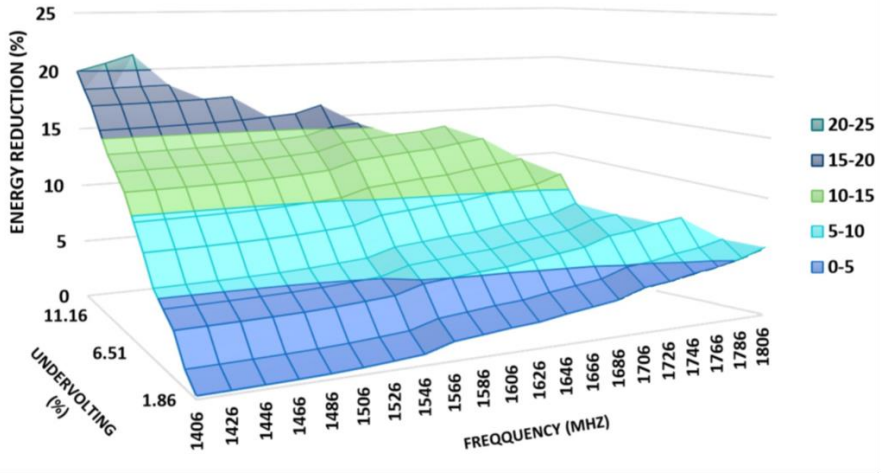

Figure 3.8: Energy reduction for given frequencies and voltages

overclocking, 5.3% energy reduction can be achieved in comparison with the baseline. As shown in the Figure 3.7, for the same matrix, SAOU saves energy up to 12.74% just by undervolting till V_{min} . There is no error in the system till V_{min} as per Fig. 3.3. However, SAOU can save up to 20% with going beyond V_{min} and correcting possible errors with checkpoint and recovery technique. Performance Overhead is mostly incurred by detection phase. According to experimental results, for a matrix of size 10K, the error detection time is 1.15% of the total execution time.

We also evaluated the energy consumption of combined undervolting and overclocking. Figure 3.8 shows the energy consumption of combined undervolting and overclocking. X-axis and Y-axis denote frequency (MHz) and undervolting level respectively. With combined overclocking and undervolting, we are able to save about 22% in comparison the original system.

Chapter 4

GreenMD: Energy-Efficient Matrix Decomposition on Heterogeneous Multi-GPU Systems

4.1 Background And Motivation

4.1.1 LU Factorization Overview

Figure 4.1 demonstrates an overview of the LU factorization algorithm. The left side, shows the LU factorization at iteration k, while the right side shows the LU factorization at iteration k+1. In a matrix of size n*n, during iteration k, a set of $k * n/b$ columns (the panel shown in yellow) is factored on the CPU, where b is the block size and n is the row width. The remaining part of the matrix is then subjected to the elementary transformations that result from the panel factorization. This phase is called trailing matrix update. The

updating of the trailing matrix requires kernels 'row swap' (DLASWP), 'triangular solve' (DTRSM), and 'matrix multiplication' (DGEMM). In other words, updating the trailing sub involves swapping up $k * (n/b)$ rows of the trailing sub-matrix (DLASWP), applying a triangular solver to the top $k * n/b$ rows of the trailing sub-matrix (DTRSM), and finally invoking the matrix multiplication for the part shown in blue (DGEMM). On the right side, at iteration k+1, one column of tiles is transferred from the GPU to the CPU to get factorized and be ready for the next iteration of LU factorization. This column of blocks, called Look-ahead panel which is used in the next iteration, is transferred before the GPU finishes the trailing matrix update at iteration k. Look-ahead is a communication and computation overlapping technique that reduces the GPU idle time while waiting for the CPU cores to deliver the panel factorization results. This procedure is repeated several times and in each iteration, the CPUs factor a panel shown in yellow color, while the GPUs update their portions of the trailing matrix shown in the blue color.

4.1.2 Profiling Observations

Figure 4.2 shows the partial timing profile for a double precision LU factorization for a matrix of size 18K on the heterogeneous system with Two GPUs. Since the profiling tools are not capable of observing the GPUs and CPU timelines at the same time, we profile the LU factorization on the CPUs and GPUs separately. For GPU, we used Nvidia Visusal Profiler to profile LU factorization. The NVIDIA Visual Profiler is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. In a heterogeneous multi-GPU system, matrix decomposition algorithms including matrix LU

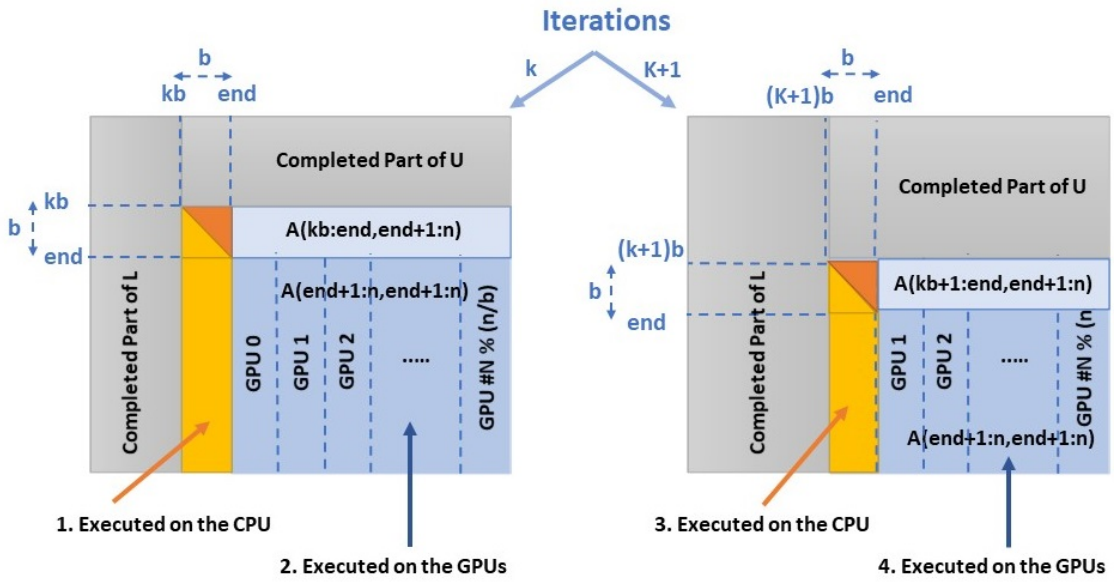

Figure 4.1: Overview of the blocked LU factorization

factorization algorithms, distribute the workload between CPU and GPU at each iteration. Figure 4.2 shows the beginning trace of LU factorization on heterogeneous system with Two GPUs for a matrix of size 18K. We enlarged the timing profile to observe the details, but this resulted in the loss of information from the entire LU factorization. Observation shows that there is no slack on the GPUs at the beginning of the LU factorization. This is because, at earlier iterations, panel factorization, which is performed on the CPU, takes less time than trailing matrix update, which is performed on the GPU. However, the amount of slack on the GPUs increases as LU factorization approaches to the end of execution. This is because the size of the trailing matrix update on two GPUs is decreasing over the time and takes less and less time to execute on the GPUs in compared to the panel factorization. In the following, we explain different components (1 to 7) in the figure so that we can develop a performance model accurately.

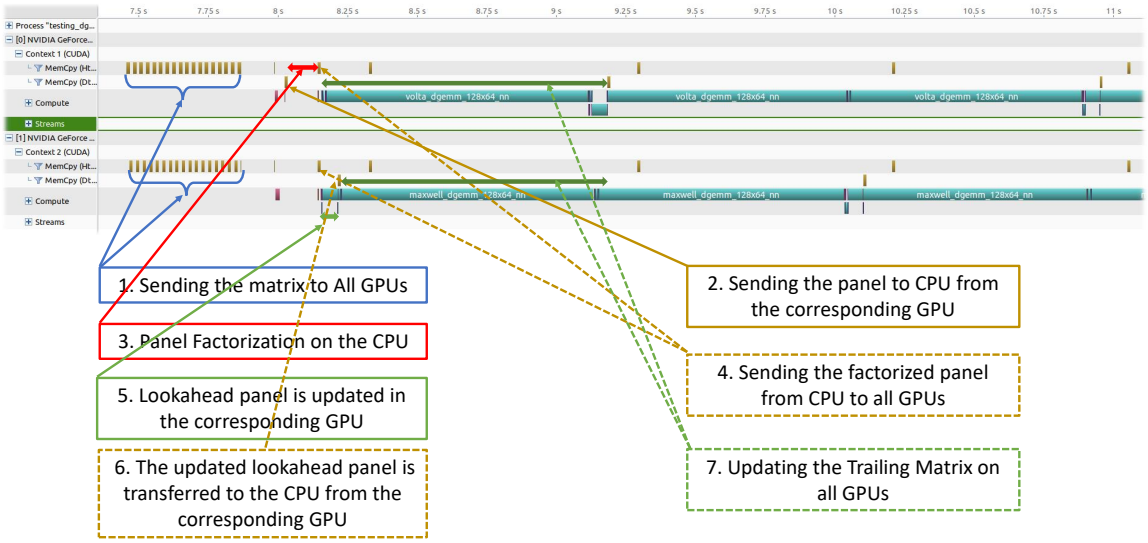

Figure 4.2: Trace of the execution of multi-GPU version of double precision LU factorization for a matrix of size 18K

- 1) At the beginning the whole matrix is divided and copied to into the GPUs. The block columns are distributed across multiple GPUs using a 1-D block-column cyclic distribution. The even and odd block-columns are transferred to GPU 0 and 1, respectively. In the case of two GPUs. The input matrix size is 18K*18K, and the block size is 512*512. As a result, each GPU holds 18 block columns with each block column containing 36 blocks of 512*512 size. Since the GPUs share the PCIe bus with the CPU, these column-blocks are transferred in a round-robin fashion.

- 2) In each iteration, one GPU is responsible for updating the look-ahead panel and sending it to the CPU while the other GPUs are updating the rest of trailing matrix. After the look-ahead panel update, the GPU continue to update the rest of trailing matrix along with the other GPUs.

- 3) Panel factorization is done on CPU.
- 4) Factorized panel is sent to all GPUs for the update phase for the next iteration.
- 5) in the next iteration, look-ahead panel update is done on a single GPU holding the next panel.
- 6) the updated look-ahead panel is transferred to the CPU. This phase is done to overlap the communication and computation.
- 7) Then, the rest of the trailing matrix, which are distributed across multiple GPUs are updated on the corresponding GPUs.

4.2 Slack Predictor

In each iteration, either the CPU or the GPU is in the critical path. Critical path consists of a group of tasks that takes the maximum time among different paths. Identifying the critical path allows us to extract slack duration in a non-critical path. We precisely estimate the slack length at a given iteration, and then reclaim the slack by utilizing the DVFS. Using the algorithmic knowledge, we estimate the execution time of each iteration on the CPU/GPUs based on the amount of operations and their compute capacities. It may be reminded that the earlier paper on DVFS for LU factorization used profiling with a simple performance model [19]. However, our techniques do not involve any bench-marking and the performance can be directly computed.

4.2.1 Performance Model Of LU Factorization On The CPU

By dividing number of operations (workload) by the compute capability of the underlying architecture, the execution time of the CPU can be estimated. The compute capability is defined as equation 4.1, where p_{cpu} is the maximum peak performance. Peak performance is defined as the maximum number of floating point operations per second for the underlying architecture, $freq$ and $Max_{frequency}$ are the current running frequency and maximum frequency of the CPU, respectively. MAGMA library is highly optimized and is getting close to the maximum peak performance. .

$$Compute_{capability} = p_{cpu} \times \frac{freq}{Max_{frequency}} \tag{4.1}$$

The number of operations or workload at kth iteration, W_{panel}, required to perform panel factorization on a single block column of matrix with size of $M_k \times b$ can be calculated as [28]

$$W_{panel} = (M_k - \frac{b}{3}) \times b^2 \tag{4.2}$$

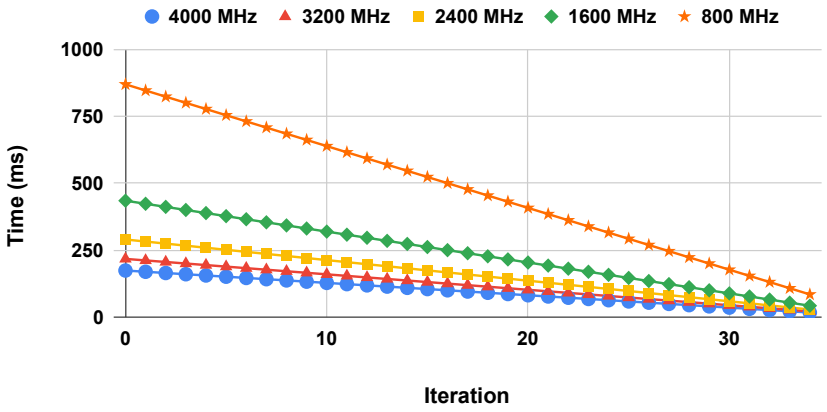

Figure 4.3: The estimated time of CPU w.r.t various frequencies

In LU factorization with matrix of size $m \times n$, and block size b, M_k varies during each iteration k, which can be calculated as 4.3.

$$M_k = (\frac{m}{b} - k) \times b \tag{4.3}$$

Hence, the amount of operations, W_{panel}, will be estimated through equation 4.4.

$$W_{panel_k} = ((\frac{m}{b} - k) \times b) - \frac{b}{3}) \times b^2 \tag{4.4}$$

Panel factorization is naturally a sequential program. So the estimation time for panel factorization, T_{CPU_k} for a CPU thread can be determined by the equation 4.5 for a given iteration k.

$$T_{CPU_k} = \frac{(((\frac{m}{b}) - k) \times b) - \frac{b}{3}) \times b^2}{p_{cpu} \times \frac{freq}{Max_{frequency}}} \tag{4.5}$$

But with huge number of computing cores provided by the multicore architectures, MAGMA is written to use multi-threading even for panel factorization. Since using multiple threads brings in extra overhead, the number of flops required for the panel factorization increases by about $b^3 \times log\ (N_{cpu-threads})$, where $N_{cpu-threads}$ is the number of threads participating in the panel factorization [28]. As a result, we can estimate the panel factorization execution time of the multi-threaded CPU at kth iteration as 4.6.

$$T'_{CPU_k} = \frac{(((\frac{m}{b}) - k) \times b) - \frac{b}{3}) \times b^2 + b^3 \times log\ (N_{cpu-threads})}{p_{cpu} \times \frac{freq}{Max_{frequency}} \times N_{cpu-threads}} \tag{4.6}$$

Since we would run the CPU at different frequencies for DVFS, we compared the estimated values through equation 4.5 against the measured values for various frequencies. The empirical results are extracted for double precision LU factorization with matrix of size 18K. The panel factorization is running on the "CPU Intel(R) Core(TM)i7-6700k" while employing only one thread. Figure 4.3 presents the analytical results for execution time of the panel factorization during different iterations. The execution time is inversely proportional to the frequency. We also measured the error rate of the execution time for different frequencies. Figure 4.4 shows the error rates for different frequencies w.r.t different iterations. According to the results, the average error rate for different iterations is less than 4% for different frequencies. The Empirical results shown in Figure 4.4 demonstrates different amount of error rate for different frequencies. This is because, there is a different amount of time needed to charge and discharge the transistors with different frequencies. In higher frequencies, there is less amount of time to charge and discharge the transistors. So timing errors are one of the main reasons for faults/errors. However, lower frequencies, the execution time increases and as a result, the probability of observing the errors increases as well. So the probability of fault is dependent on both the frequency and the execution time [82]. Besides, different frequencies can change the temperature of the processor which could lead to different amount of error rate. In other words, the error rate is dependent on frequency, execution time and temperature.

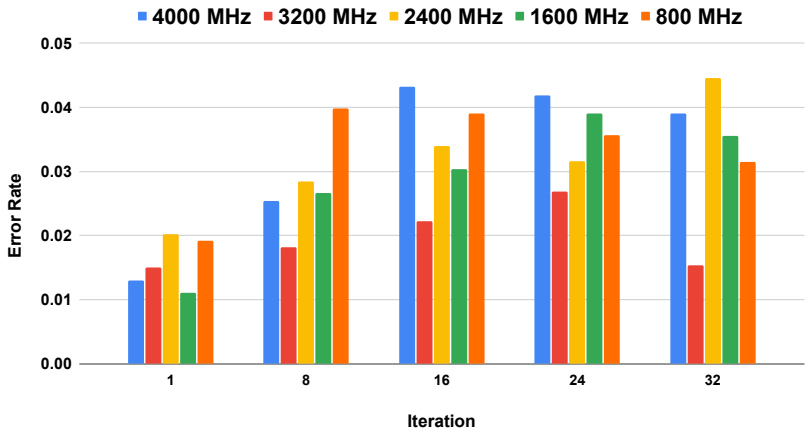

Figure 4.4: Error rate in performance model on the underlying CPU

4.2.2 Performance Model of the GPU Kernel

Similar to the CPU performance model provided in equation 4.5, we develop the GPU performance model for one hardware thread and then extend it to consider all the GPU computing cores (threads). However, unlike CPU, the GPU is an SIMT architecture and there is no resource conflict. The GPU throughput is obtained simply by multiplying per thread throughput with the number of cores in the GPU. The GPU execution time is estimated by dividing the number of floating point operation by the GPU computation rate.

In each iteration of the LU factorization, the size of trailing sub-matrix is reduced by the block size from both row and column width. Given the matrix size as mxn and block size b, the sub-matrix row and column width at iteration k can be calculated as:

$$Row_{width_k} = (\frac{m}{b} - k) \times b \tag{4.7}$$

$$Column_{width_k} = (\frac{n}{b} - k) \times b \tag{4.8}$$

Therefore, the total number of operations (workload) required for trailing matrix update at a given iteration k can be written as [28]:

$$Worklaod_{GPU_k} = ((\frac{m}{b} - k) \times b) * ((\frac{n}{b} - k) \times b)^2 \tag{4.9}$$

Therefore, extending the equation 4.5 to the GPU, the equation 4.10 can be derived to estimate the execution time of each GPU at a given iteration k.

Where

- N_c is the number of cores per GPU.
- $freq$ is the running frequency of the GPU.
- $T_{GPU_{i@k}}$ is the time required for the kernel (trailing matrix update) running on ith GPU at a given iteration k.
- $Workload_{GPU_{i@k}}$ is the total workload of the ith GPU at a given iteration k.

Equation 4.9 calculates the total number of floating point operations at iteration k. In the case of multiple GPUs, the amount of computation on each GPU will be divided by the number of GPUs. In case the number of panels in the trailing matrix is not divisible by the number of GPUs, some GPUs will hold an extra panel to update. The workload for the

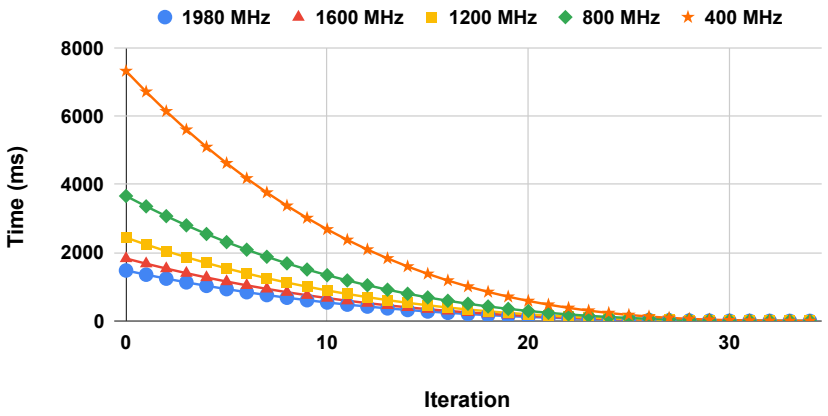

Figure 4.5: The estimated time of single GPU w.r.t various frequencies using Equation 4.10

extra panel can be calculated using the equation 4.9.

$$T_{GPU_{i@k}} = \frac{Workload_{GPU_{i@k}}}{P_{gpu} \times \frac{freq}{Max_{frequency}} \times N_c} \tag{4.10}$$

Figure 4.5 shows the estimated execution time of double precision LU factorization with an input matrix of size 18K on a single GPU card for different running frequencies. The "GPU GTX 1660 super" card, which has a total of 1408 processing cores, is used to update the trailing matrix. To validate our performance model, we also extracted the empirical results and calculated the execution time error rate for the same configurations. The kernel execution time is measured from the host which takes the kernel launch time and etc. into account. Figure 4.6 shows the error rate of the execution time on a single GPU in presence of various frequencies. The results show that the average error rate is about 5%. However, the error rate is increases towards the last iterations. This is because, with the update matrix getting smaller and smaller, GPU is not fully utilized. The execution time is getting smaller,

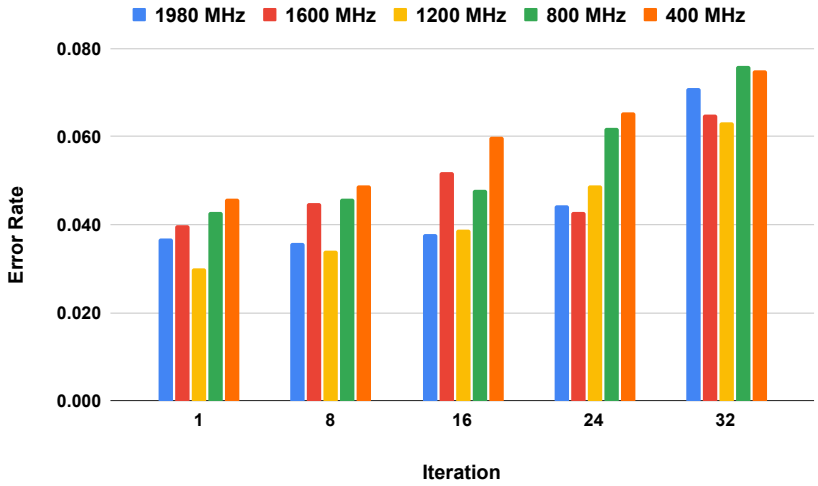

Figure 4.6: Error rate of GPU performance model provided in equation 4.10

which results in larger error rate. Using equation 4.10, we also estimate the execution time of Two GPUs. Figure 4.7 compares the estimated values of single and double GPUs while running the trailing matrix update. The running frequency is fixed at the default value of 1980 MHz. It shows that the execution time of GPU is inversely proportional to the number of GPUs at different iterations. In the case of Two GPUs, the execution time of trailing matrix is almost divided by two for different iterations. We also extracted the experimental result for two GPUs and the average error rate for the estimated execution time was about 4.6%.

4.2.3 Performance Model Of Data Transfer

We must first model the PCIe bus latency before we can model the transfer time between CPU and GPU. We take LogGP model [21] as a starting point and expand it to multiple streams. According to the model, transferring a single chunk of size k bytes takes "L+O+(k)B". The time it takes for a single byte to transmit from the source to the

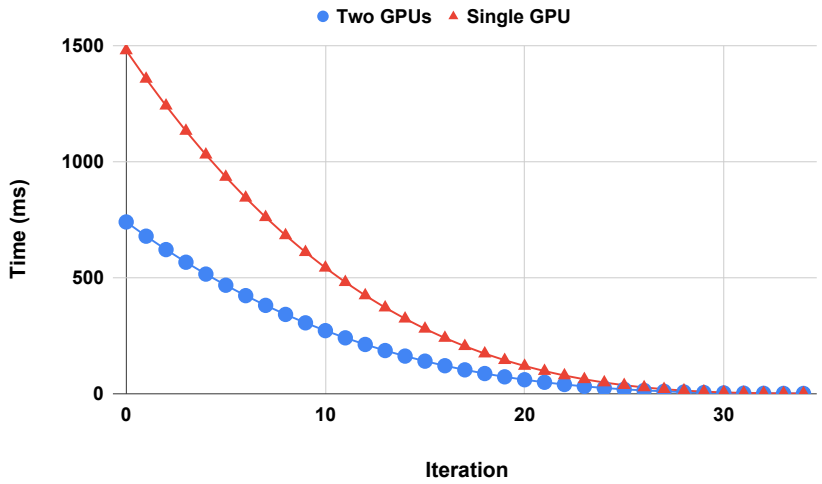

Figure 4.7: Estimated GPU time with single and Two GPUs at default frequency of 1980 MHz

destination endpoint is denoted by the 'L'. 'O' specifies the amount of time the processor is working on the transmission. In other words, this is the amount of time the CPU spends registering the DMA request with the controller, while 'B' represents the PCIe bus's bandwidth. We don't have any overhead on the device side because we're using DMA transfer, which writes data directly into memory. Even with DMA transfers, the data is split into data chunks and transmitted across PCIe in a continuous stream. We define 'g' as the gap between these chunks. This is the amount of time it takes to start a new transfer API. As a result, transferring multiple chunks requires "$L + O + k_1G + (n-1)g + k_nG$". When many streams are present, each stream is transferred one at a time, and 'g' will be the overhead to initiate a new transfer API on a different stream. As a result, the time required to transfer K bytes utilizing n streams will be determined using the equation 4.11. Hence, in case of double precision LU factorization with matrix of size $m \times n$, the copy time for each iteration can be extracted using the equation 4.12. Here m is the row width of matrix,

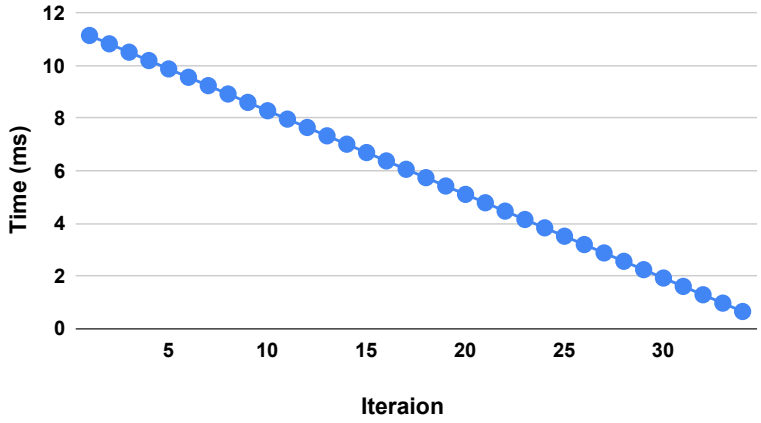

Figure 4.8: The estimated copy time of double precision LU factorization for a matrix of size 18K

b is the block size or the panel width which remains same over the iterations.

$$T_{copy_k} = L + O + K/B + (n-1) \times g \tag{4.11}$$

$$T_{copy_k} = L + O + ((m - k \times b) \times b \times 8)/B + (n-1) \times g \tag{4.12}$$

Even though, the newer GPUs are equipped with dual copy engines and they might share the resources that affects the transfer time, we are not considering the dual copy engine because in case of LU factorization, the copy transfers are not occurring at the same time. Figure 4.8 shows the estimated time between CPU and GPU for different iterations of double precision LU factorization with matrix size of 18K. The estimated results illustrate that the copy time is proportional and inversely proportional to the size of the data and bandwidth, respectively. 'L', 'O', 'g', and 'B' are extracted using a simple profiling phase. We also compared the estimated values with the experimental results of the copy time for a matrix of size 18k in case of single GPU (NVIDIA GeForceGTX 1660 SUPER) and CPU

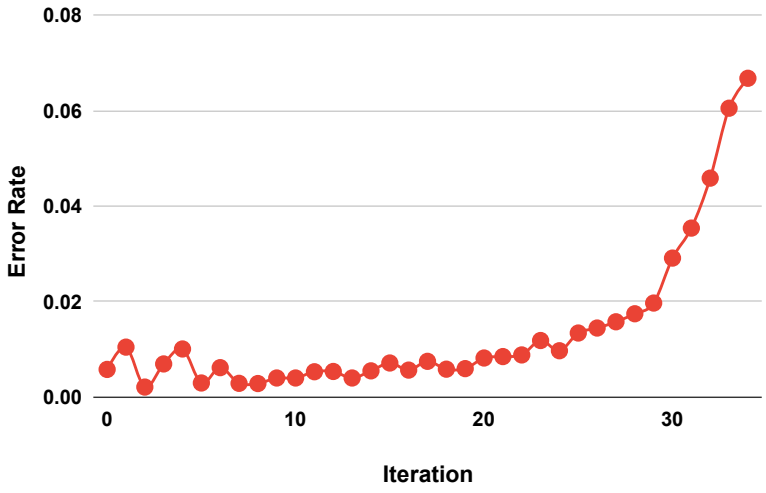

Figure 4.9: The error rate of the PCIe model for a matrix of size 18K

(Intel(R) Core(TM) i7-6700k). Figure 4.9 shows that the error rate is very negligible in the iterations when the PCIe bus is fully utilized. However, the error rate is increases due to under utilization of the PCIe bus during the last iterations of LU factorization. For example, the amount of data that is transferred between CPU and GPU in the first and last iterations of the LU factorization are 73 MB, and 2.42 MB, respectively. On average, the error rate of the estimation model is about 1.3% for different iteration which is very small.

However, it may be observed that, compared to hundreds of msecs in CPU and thousands of msecs in GPU execution times, the data transfer time is negligible.

4.3 DVFS-Based Slack Reclamation

A slack is a period of time when one computer component waits for another. Load imbalance, inter-task or inter-process communication, and memory access stalls are all common causes of slack. A Critical Path is a certain sequence of tasks that spans from the

beginning to the end of the execution and has zero slack. Slack is only observed in non-critical path of the application. While slack on non-critical paths is commonly utilized for energy savings, fully reclaiming them without affecting application performance is difficult. During the LU factorization, slack occurs on the CPU or GPU at each iteration, as shown in figure 4.10. Slack occurs only when the CPU waits for the GPU to update the next panel or when the GPU waits for the CPU to factorize the panel. If the CPU finishes earlier in an iteration, we can slow it down to finish at the same time as the GPU and vice versa. As the panel and matrix update sizes change across iterations, the amount of slack changes. Then we use DVFS on the CPU and GPU to take full advantage of the slack on non-critical paths. We can reduce power consumption without affecting performance for the next iteration by properly reducing the frequency of the processing units to only eliminate the slack.

Algorithm 8 provides the overview of DVFS controller that estimates the panel factorization time on CPU, matrix update time on GPUs, and data copy time between CPU and GPU using the equations provided in section 4.2. Let T_k^{CPU}, $T_{GPU_{i@k}}$, and T_k^{Copy} represent the CPU time, ith GPU execution time at iteration k, and the transfer time, respectively. The slack lengths of LU factorization at each iteration for CPU and GPU are

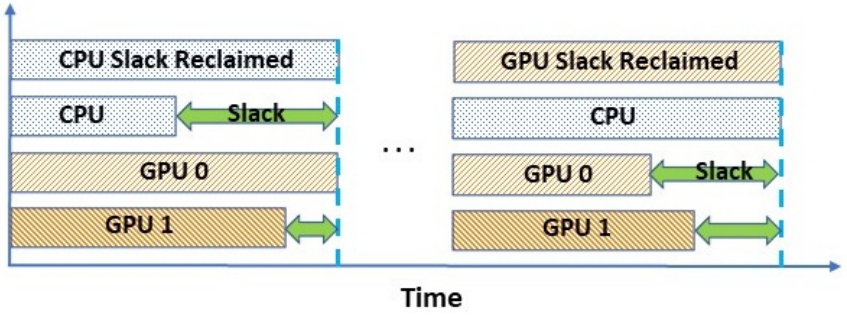

Figure 4.10: An overview of slack reclamation.

given by equations 4.13 and 4.14. These values are extracted according to the equations 4.6, 4.10, and 4.12 respectively. In each iteration the copy time is the summation of both H2D and D2H transfers that happens between CPU and GPUs.

$$slack_{CPU_k} = Max(T_{GPU_{i@k}}) - T_k^{CPU} - T_k^{Copy}; 0 \leq i < \frac{N}{nb} \tag{4.13}$$

$$slack_{GPU_{i@k}} = T_k^{CPU} + T_k^{Copy} - T_{GPU_{i@k}} \ ; \quad 0 \leq i < \frac{N}{nb} \tag{4.14}$$

If the equation 4.13 has a positive value, the CPU has slack time, and the CPU should wait for the GPU to finish its execution. To find the slack on different GPUs, we use equation 4.14 for each individual GPU. Similarly, if it has a positive value, the GPU will have the slack. There is no slack in the CPU or GPU when the value is 0.

We can reduce power consumption without affecting performance for the next iteration by properly reducing the frequency of the processing units to only eliminate the slack. As we have shown in the earlier section, the execution time of the LU factorization is proportional to the frequency of computing units.

We use Advanced Configuration and Power Interface (ACPI) to change the CPU core's frequency at runtime to reduce the voltage/frequency transition time. Our frequency enforcement is performed by manipulating each core's "scaling_setspeed" file. When this

Algorithm 8 Overview of DVFS methodology

```
Initialize ()
Profiling Phase ()
for i = 0, 1, ..., iteration − 1 do
    cpu_time ←cpuPredict()
    gpu_time ←gpuPredict()
    copy_time ←copyPredict()
    cpuSlack_length ←slackPrediction(cpu_time , gpu_time, copy_time)
    if slack_length ≠ 0 then
        if cpu_time < gpu_time then
            Invoke DVFS_CPU()
        else
            Invoke DVFS_GPU()
        end if
    end if
end for
```

device file is modified, Linux triggers a group of system calls that adjust the CPU core's frequency in about 40 microseconds. Compared to the hundreds of msecs of CPU and GPU execution times per iteration (Figures 4.3 and 4.7), this overhead is negligible.

We derive the equation to find the optimum frequency for the units on the non-critical path based on the current and target execution time. Equation 4.15 below is derived to calculate the amount of optimum frequency, where f_{opt} and $f_{default}$ are the optimum and default frequencies of the component with slack at a given iteration. The frequency of the CPU/GPUs is adjusted using the pseudo code provided in algorithm 9. We choose the minimum frequency if the adjusted frequency is less than the minimum defined frequency. When an adjusted frequency is not supported by the hardware, two consecutive available frequencies are used to eliminate the slack. This is because, only available discrete frequencies offered by CPU and GPU DVFS are taken into account in GreenMD.

$$f_{opt} = f_{default} \times \frac{\max(T_{CPU}, T_{GPU}) - slack}{\max(T_{CPU}, T_{GPU})} \tag{4.15}$$

Algorithm 9 Frequency adjustment during the slack

```
if f_optimum ≤ f_min then
   f_optimum = f_min
end if
if f_min ≤ f_optimum ≤ f_max then
   if f_optimum ∉ available_frequencies then
      f_lower ← adjacent_frequency(f_optimum, available_frequencies)
      f_upper ← adjacent_frequency(f_optimum, available_frequencies)
      Adjust_frequency(slack,f_lower,f_upper)
   else if Adjust_frequency(slack,f_optimum) then
   end if
end if
```

With increasing the number of GPUs, since, using the algorithmic knowledge, the amount of operations are known on each GPU, we can still estimate the execution time of each GPU during each iteration of the LU factorization and as a result, we can find the slack. The amount of slack on the CPU and GPUs will be varied based on the size of the input matrix and the number of GPUs. The input size of the input matrix was limited by memory size of the GPUs. For the same input size of the matrix, if we increase the number of GPUs, the amount of slack will be even more on the GPUs and we can save even more energy. This is because the main source of power consumption is consumed by the GPUs in an heterogeneous system with GPUs. According to the empirical results provided through the offline profiling, at earlier iterations, since the panels widths are larger, the CPU takes more time than the GPU to factorize the panel ans we still experience slack on the GPU unless more number of CPUs and potentially GPUs are employed to help the CPU to factorize the panel. And if the number of GPUs increases, in later iteration, when the trailing matrix size is getting smaller and smaller, there could still be slack in the GPUs that can be utilized for energy saving.

4.4 Undervolting

Since we don't compromise the LU cauterization's performance, DVFS can only be utilized to reclaim slack on non-critical path components. This is because using the DVFS on critical paths degrades the application's overall performance for compute-intensive workloads [9]. DVFS is mainly concerned with dynamic power consumption. However, the static power is becoming predominant in today's technology. Hence, we also employ undervolting (at a fixed frequency) to reduce both static and dynamic power while maintaining performance.

To ensure that the microprocessor functions reliably under varying load and environmental conditions, microprocessor manufacturers usually append an operational guard-band (a static voltage margin) of up to 20% of the nominal voltage [107]. The guard-bands additionally take into consideration errors caused by the load line, aging effects, noise, and calibration error [78]. Because these errors do not occur frequently, significant energy savings can be achieved by lowering guard-band voltage to a much lower supply voltage [45]. In our work, we aim to save energy by utilizing the voltage guard-band between the nominal voltage and the actual OS safe voltage while maintaining performance. To extract $V_{safeMin}$ during LU factorization, we take a similar approach as described in [47]. We undervolt to the safe minimum voltage $V_{safeMin}$ without introducing any fault into the system. System may experience soft errors if $V_{safeMin}$ is exceeded.

To determine the $V_{safeMin}$ for LU factorization, we profile the LU factorization for small matrices. According to [105], the sensitivity analysis is performed by lowering the voltage below the nominal voltage (1.075 V). First, we run the program at nominal voltage and record the result as "golden output". We start with the corresponding voltage

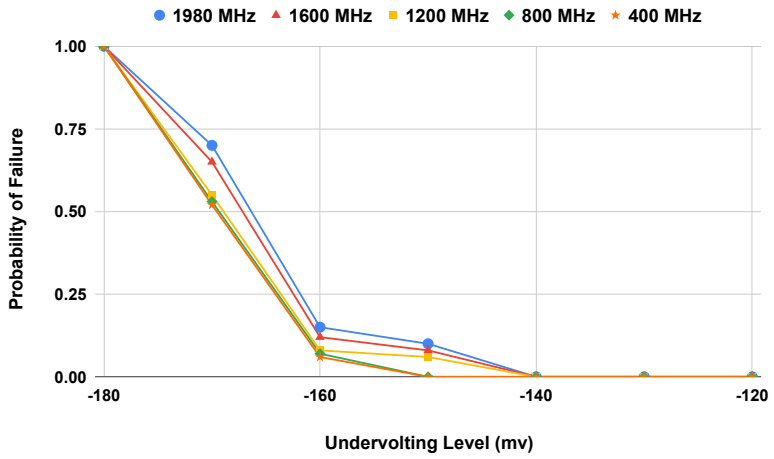

Figure 4.11: GPU's probability of failure w.r.t undervolting

in the CPU and reduce the voltage in 10 mv steps at a given fixed frequency. We run the application 100 times and record the number of faulty runs. If the output does not match the golden output, the application's run has failed. The frequency is then changed, and the undervolting is repeated at a new fixed frequency. We extract the $V_{safeMin}$ for both CPU and GPU with different running frequencies. This is because the maximum level of undervolting could be different for different frequencies as observed in Figure 4.12 and 4.11. The probability of failure for the CPU is shown in Figure 4.12. There is no error until the undervolting level reaches 150 mv. At lower running frequencies, we can decrease the voltage even further (10 mv) without observing the error.

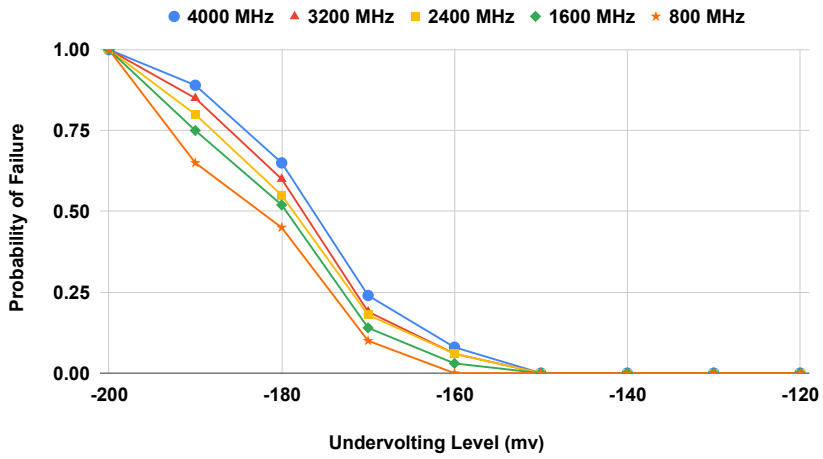

Figure 4.12: CPU's probability of failure w.r.t undervolting

4.5 Evaluation

4.5.1 Implementation Details

All experiments are preformed on heterogeneous system with 8-cores Intel CPU and Two homogeneous GEFORCE GTX 1660 SUPER, whose architectural specifications are listed in Table 5.1. We were able to evaluate the results for up to a 18K matrix size due to GPU's limited device memory. The proposed power management algorithm is embedded inside the application code and called right before the next iteration. Considering the performance overhead of DVFS, the $DVFS_{CPU}()$ and $DVFS_{GPU}()$ functions are called if

Table 4.1: Experimental setup configuration

Component	CPU	GPU
Architecture	Intel(R) Core(TM) i7-6700k	NVIDIA GeForce GTX 1660 SUPER
Minimum Frequency	800 MHz	300 MHz
Maximum Frequency	4200	1980
Memory	16 GB	6 GB
Cache	L1(128 KB) L2 (1 MB) L3 (8 MB)	L1 (64 KB per SM) L2 (1536 KB)
OS	Ubuntu 20.04	

there is enough slack available in the next iteration. We use "linux-intel-undervolt" and "cpupower frequency-set" APIs to undervolt and scale the CPU frequency. These APIs can be used for Intel CPUs with an integrated voltage controller (FIVR). In CPU, we change only the CPU frequency and do not touch the memory frequency. However, in case of GPU, since the GPU core and memory frequencies are coupled, changing the core frequency might change the memory's frequency as well.

For GPU profiling phase and extracting the safe minimum voltage of the GPU, MSI After Burner [2] was used. However, MSI After Burner is not supported on Linux operating system. So, to reduce the voltage of the GPU, we used the similar approach employed in [105]. Since there is no direct API to reduce the voltage, we reduce the voltage of GPU by lowering the GPU's target power limit at a fixed frequency. To undervolt the GPU, several APIs from NVIDIA Management Library (NVML) are used as listed in Table 4.2.

It is not possible to truly disable GPU Boost in modern NVIDIA architectures without resorting to very risky procedures involving flashing custom firmware. However, it is still possible to lock the graphics frequency in recent GPUs. We use the NVML library's 'nvmlDeviceSetGpuLockedClocks' API to fix the frequency and, as a result, the voltage. This API effectively locks the graphics frequency, ensuring that it remains constant at the desired frequency with tiny variations, which could be due to the auto boosting option.

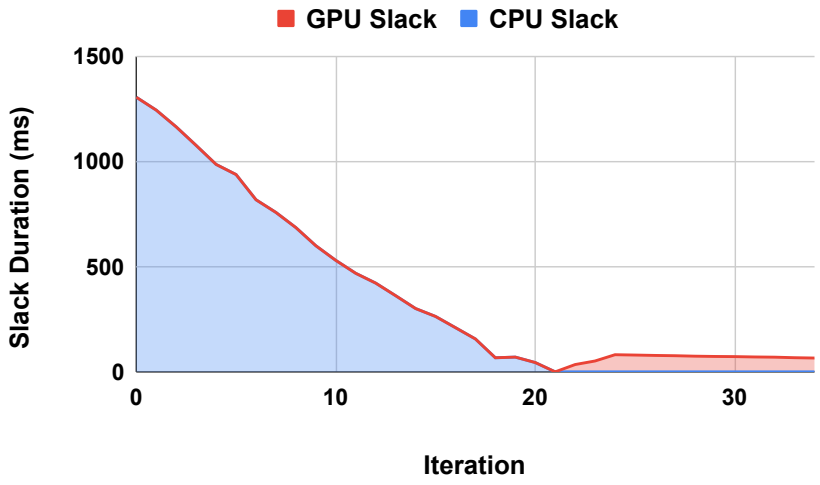

Figure 4.13: The amount of CPU and GPU slack for double precision LU factorization with one GPU and matrix of size 18K

4.5.2 Results

We executed the LU factorization in presence with one and two GPUs. In case of single GPU, the amount of slack on the CPU and GPU is shown in Figure 4.13 for a matrix size of 18K x 18K. The slacks for CPU and single GPU are observed in iterations 0-21 and 22-34, respectively. This is because, even though the GPU is equipped with huge number of computing core, it has larger ratio of "$workload/(compute - capability)$" compared with the CPU till iteration 21. Since there is no slack on the GPU for iterations before iteration 21, we do not change the GPU frequency until then. We only adjust the CPU frequency to

Table 4.2: Power management and undervolting APIs

API	Desciption
nvmlDeviceSetPersistenceMode	Enables persistent mode to prevent driver from unloading
nvmlDeviceSetPowerManagementLimit	Set new power limit of the GPUdevice.
nvmlDeviceGetClock	Retrieves the clock speed for the clock specified by the clock type and clock ID.
nvmlDeviceSetGpuLockedClocks	Set clocks that device will lock to.
nvmlDeviceGetTotalEnergyConsumption	Read the GPU's total energy consumption
nvmlDeviceResetApplicationsClock	Resets the application clock to the default value
linux-intel-undervolt	Undervolt the Intel CPUs
cpupower frequency-set	Sets the CPU's frequency

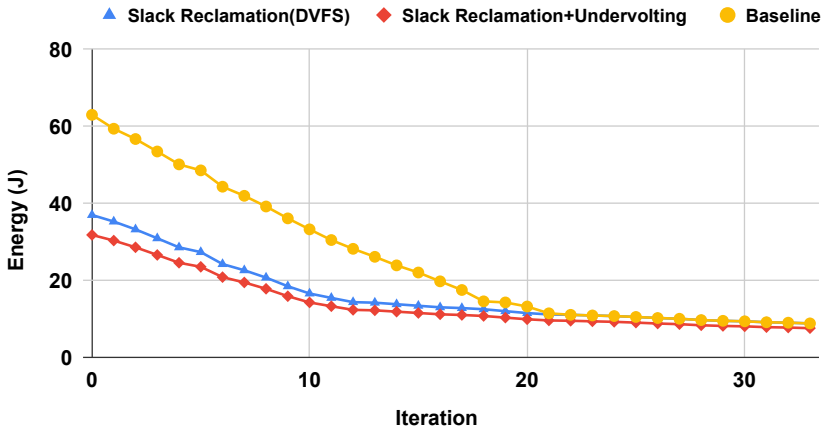

Figure 4.14: The CPU energy improvement of double precision LU factorization with single GPU for a matrix of size 18K

reclaim the slack allowing both CPU and GPU to complete their tasks at the same time at a given iteration. If the amount of adjusted frequency is less that the minimum frequency of the underlying architecture, we set the frequency to the minimum frequency. Similarly, if the adjusted frequency is greater than the maximum frequency, set the the frequency to the maximum frequency.

Along with slack reclamation, we also extracted the maximum level of undervolting when no fault is introduced in the system. At a given frequency corresponding to each iteration, we apply maximum level of safe undervolting. The amount of energy consumed at default scenario (baseline) and proposed approach w.r.t iteration is shown in Figure 4.14. The X-axis represents the iteration number, while the Y-axis represents the amount of energy consumed during each iteration. Using DVFS along with undervolting, for a matrix of size 18K, we were able to save the CPU energy consumption up to 51%. We also measured the energy consumption of the single GPU, Figure 4.15 shows the GPU's energy consumption

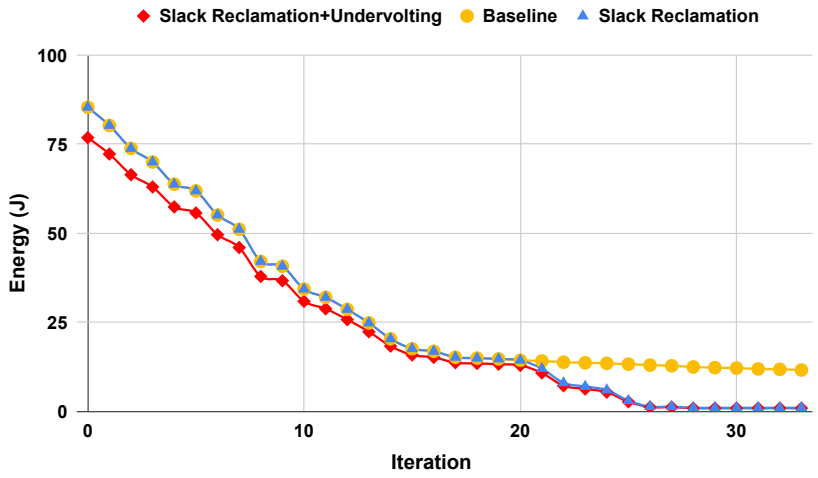

Figure 4.15: GPU Energy improvement of double precision LU factorization in presence of single GPUs for a matrix of size 18k

for the default configuration as well as the proposed method. Because there is no slack till iteration 21, the energy improvement comes only from undervolting. However, after that both DVFS and undervolting leads to more energy reduction. Figure 4.15 shows that, on average, we save energy about 18% on the single GPU.

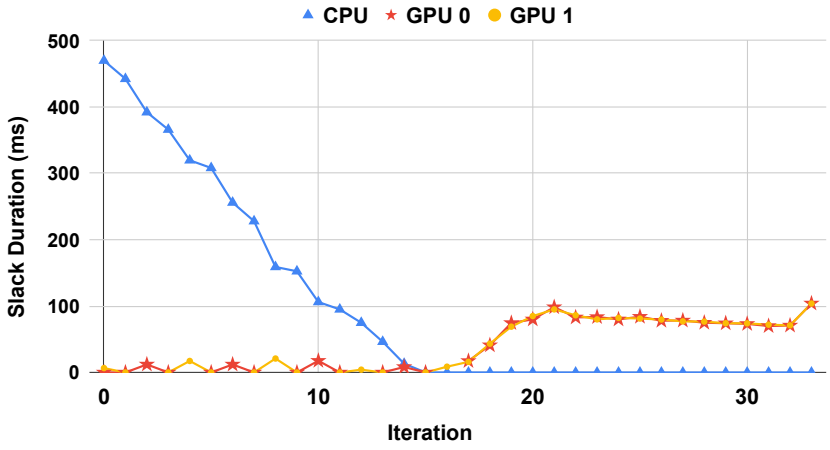

Figure 4.16: The amount of slack for double precision LU factorization in presence of Two GPUs for matrix of size 18K

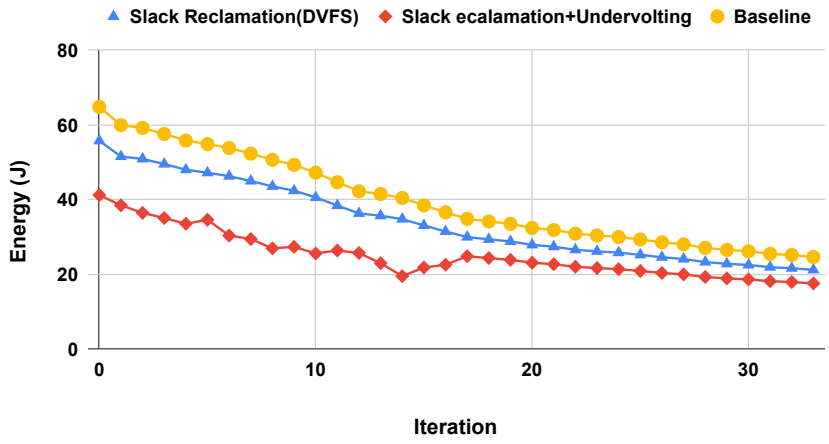

Figure 4.17: The CPU energy improvement of double precision LU factorization with two GPUs for a matrix of size 18K

We have also extracted the results for heterogeneous system with two GPUs. In this case, we observed less slack in the CPU at earlier iterations and more slack in the GPUs at later iterations, compared to a single GPU. This is because the trailing matrix update is done in parallel in both GPUs reducing the update time and the CPU slack. Figure 4.16 shows the amount of slack for different iterations for both CPU and GPUs. Compared with

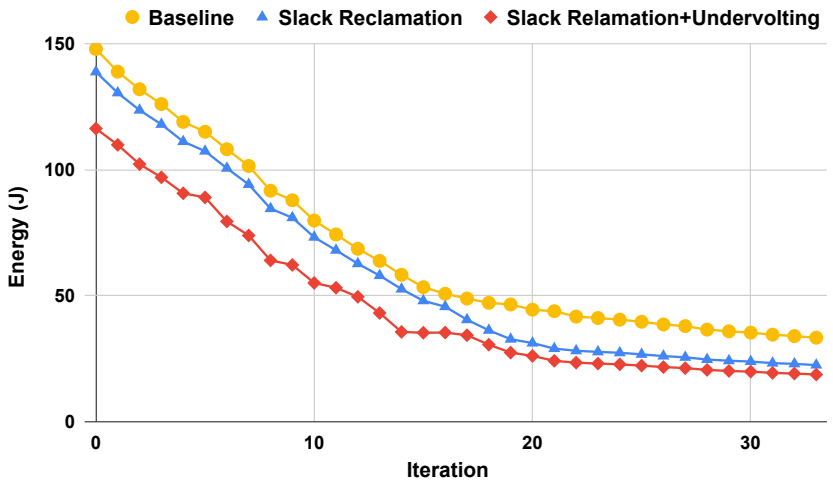

Figure 4.18: Total CPU and GPU energy improvement of double precision LU factorization with two GPUs for a matrix of size 18k

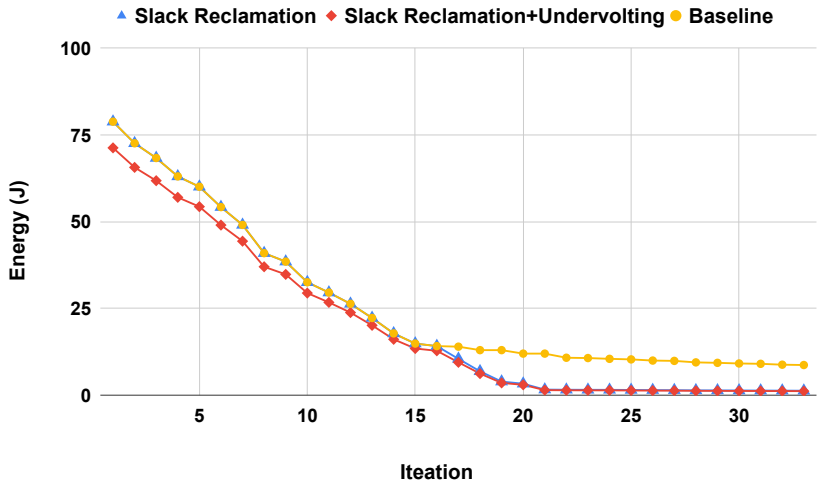

Figure 4.19: GPU energy improvement of double precision LU factorization in presence of Two GPUs for a matrix of size 18k

a single GPU case in Figure 4.13, the slack at the CPU is reduced by almost half. We fully reclaim the CPU slack and adjust the CPU's frequency to a desired frequency, allowing both CPU and GPU iterations to be completed at the same time. In the second half of iterations, we also change the frequency of the GPUs to reclaim the slack in both GPUs. The frequencies are automatically and independently enabled by the API during the execution. Also, Figure 4.17 shows the amount of CPU energy consumption in presence of DVFS and undervolting. Compared to single GPU results, illustrated in Figure 4.14, we observe less energy improvement in the CPU in the earlier iterations and more energy improvement during the late iterations. This is because, in case of two GPUs, the CPU experiences less amount of slack during the earlier iterations and more amount of slack during the late iterations, which leads to less and more energy improvement during these periods.

Similar to the CPU, we also extracted the energy improvement for GPU using the proposed method. The energy consumption of GPU for the default configuration and

the proposed method is shown in Figure 4.19. Since there is no slack in the first half of the iterations, the energy improvement comes only from undervolting the GPU. However, the energy improvement in the second half of the iterations comes from both DVFS and undervolting. According to Figure 4.19, on average, we save the total energy consumption of the GPUs about 21%. In comparison to the results illustrated in Figure 4.15 for a single GPU, the small improvement in the energy saving comes mainly from the undervolting part. This is because, even though the trailing matrix execution time is reduced into half, the total power consumption doubles due to the use of two GPUs keeping the energy consumption almost the same. Figures 4.20 and 4.18 show the total energy consumption of LU factorization for a matrix of size 18k with single and two GPUs, respectively. As shown in figure 4.18, in the first half of iterations, when only the CPU experiences slack, we save 26.2% while in the second half of the iteration, when the GPUs only have slacks, we save 41.8%. However, the energy consumption is much less than the first half because the

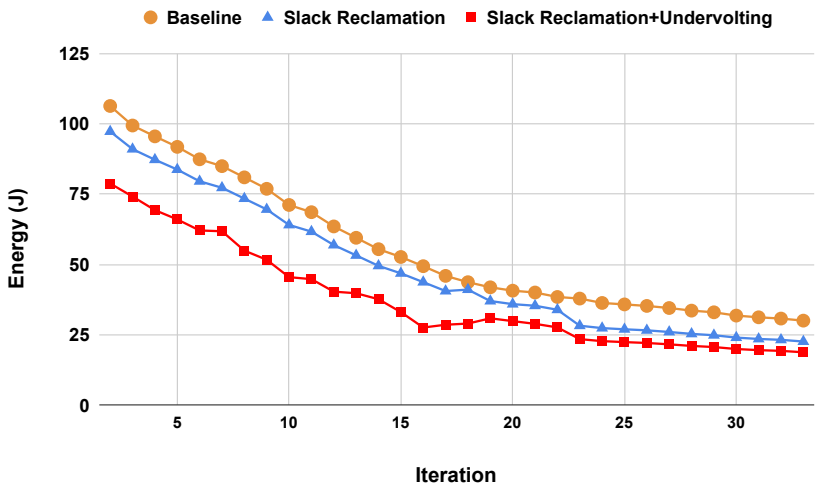

Figure 4.20: Total CPU and GPU energy improvement of double precision LU factorization with single GPUs for a matrix of size 18k

Table 4.3: Energy improvement of CPU, GPU/GPUs in heterogeneous systems with one GPU and two GPUs

	Energy Improvement (%)		
	CPU	**GPU/GPUs**	**Total**
Heterogeneous system with one GPU	51%	18%	32.4
Heterogeneous system with two GPUs	59%	21%	31%

trailing matrix gets smaller and the execution time reduces. Overall, there is a 31% energy improvement in total energy for the entire LU factorization with Two GPUs.

Chapter 5

Fault Tolerant Matrix Decomposition On Heterogeneous Multi-GPU System

5.1 Background And Motivation

As explained in details, in chapter 4, LU factorization of a matrix A generates a lower triangular matrix L and an upper triangular matrix U. In the right-looking block LU algorithm, the matrix is divided into blocks that are NB in size (called a blocking factor). Each step involves processing one block row and one block column while also updating the square trailing sub-matrix. In our approach, the sequential phase (panel factorization) is executed on the CPU the parallel phases (row panel update and trailing matrix update) are executed on the GPU/GPUs. These phases have different sensitivity to the faults. In the

following we will discuss the fault model on each of these phases and how we should address the faults/errors in these phases.

5.1.1 Fault Model

There are two types of errors that might happen during the computation. Hard errors and soft errors. Hard error occurs due to fatal damages such as circuit destruction and it is not transient. Hard errors are repeatable with the correct sequence of actions within the circuit. Unlike a hard error, there is no damage in the semiconductor itself in the case of a soft error and it is a transient error. Soft errors are not easily detectable and the program finishes its execution without any warning. If the soft errors are not detected during the computation, they might propagate to different sections of a long running computation kernels which makes it very difficult to dealt with. A single soft errors can affect nearly the entire matrix in case of LU factorization, if not detected and covered properly. So, we aim to address the soft errors which are more important and predominant types of errors in today's technologies.

5.1.2 Challenges Of Fault Tolerant LU Factorization

In the right looking LU factorization, since the checksum relationship is maintained during the computation, we have built the fault tolerant algorithm on top of the LU factorization. In the right looking LU factorization, Gaussian elimination is used to find the elements of L by dividing the elements of original matrix by the element on the diagonal. However, with floating point numbers, a value that is extremely close to zero could be on the diagonal and not be immediately apparent as a problem. If this element is zero, division

is obviously not possible. Algorithms for LU factorization involve pivoting, where the row with the greatest element in the current column is switched with the current row, to make sure that this doesn't happen. In right looking LU factorization, in each iteration, panel factorization, row panel update and trailing matrix updates are performed sequentially until the entire matrix is factored. The LU factorization is performed by iterating over the panels with the width of one block size. In each iteration, only one column with the width of block size is factored and the rest of matrix is also updated in each iteration. The height of the panel decreases by one block size in each iteration and there won't be any change in the panels of the matrix which are already factorized. The panel factorization, row panel update, and trailing matrix update are the steps that change a particular portion of the matrix. Matrix elements are broadcast to be used in the updates following the first two phases. These phases have different sensitivity to the error. Errors can spread outside of each section through the broadcast, at which point they may become irrecoverable. We are only concerned about error propagation along rows because we only utilize row checksums which will be explained later. The errors can propagate across the row after the broadcast following panel factorization.

5.1.3 Error Propagation In LU Factorization

Different areas of the matrix have different levels of vulnerability to the integrity of the calculation. We assume that errors can only occur during calculation, so that stored values will not become wrong. When an error propagate, it may turns into a number of incorrect values, making it impossible for the fault tolerance method to correct them. The good news is that in the case of LU factorization, there are specific locations where errors

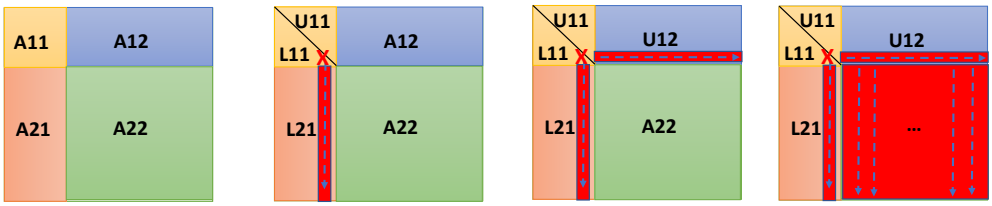

Figure 5.1: Error propagation scheme of error in different sections of the matrix

can be identified before they spread to the rest of the matrix. Figure 5.1 shows the error propagation of the error that might happen in different sections of the LU factorization. The column panel is the most important section that needs to be verified during the execution. Any errors that happen here will spread to all the rows. The remaining matrix is updated using this panel. The components in the same row to the right of the erroneous element will also be impacted by the error which makes the recovery impossible. This panel is factored individually before being transmitted to the rest of the matrix. To avoid errors propagating, the verification must be performed before the broadcast. The stored sums for this panel will be zeros as long as no errors occurred due to the way the local checksum is constructed. Because only one block of sums needs to be recalculated, this check is the shortest.

Also, the row panel, is similarly vulnerable to errors. This section's error is not immediately harmful. It would be hard to recover from a second error unless it happened to occur in the same process column, because the entire column is affected. As a result, it is crucial to verify the accuracy of this part as well; however depending on the error rate, the time between checks may be adjusted. This panel's verification takes longer than the column panel's because all of the panel's sums must be recalculated.

Another section that is least susceptible to errors is trailing matrix. An error in this area is not propagated until a panel is used in a subsequent iteration, at which point the panel verification can address it. Even while it's still possible, the probability of a second error making recovery impossible is smaller than it was for the row panel. Again, depending on the error rate, it may be essential to examine this part of the matrix as well, though less frequently than the row panel. Because each sum in the trailing matrix needs to be redone, this verification again takes longer than the row panel verification.

5.2 Proposed Methodology

Figure 5.2 shows the overall overview of the proposed fault tolerant right looking LU factorization algorithm. In LU factorization there are three main phases that need to be done sequentially. 1) Panel factorization, row panel update and trailing matrix update. During the each phase, we should make sure that the possible errors that might happen in each phase won't be propagated to the next phase or iteration. At the beginning, once the matrix elements are located in the GPU's device memory, GPUs will generate row checksum vectors and the checksums will be appended to each local block or Thread-block(TB) and the block size is increased by one. The first phase of the LU factorization is panel factorization. Any error in the panel factorization could potentially, if not covered and propagated to the rest of the matrix, could potentially corrupt the entire matrix. Hence, we check the correctness of the panel factorization in each iteration frequently. If no error is detected, we continue with row panel update; otherwise, we go back to the beginning of the panel factorization in the current iteration and repeat the whole panel factorization again. This

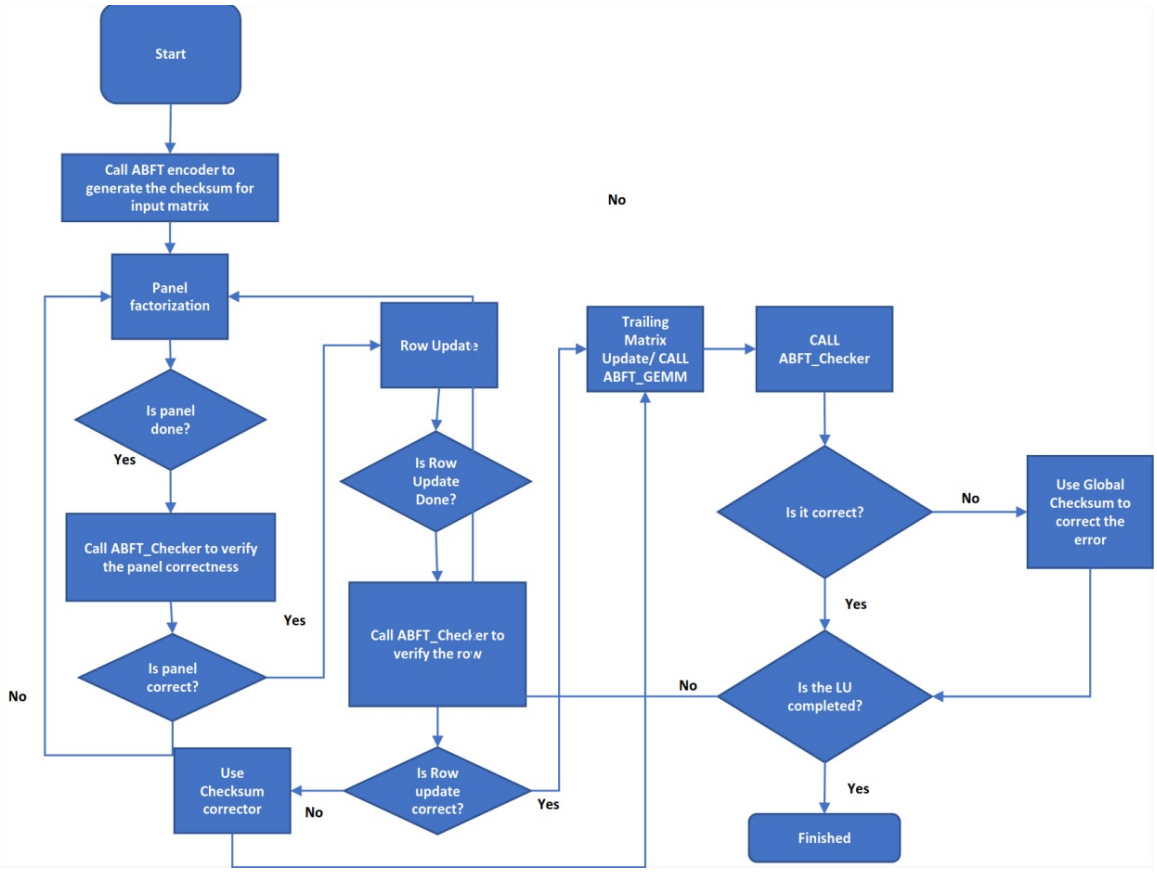

Figure 5.2: Overview of the Fault Tolerant LU factorization

sequential phase is done on the CPU side and if any error is detected, we should get the correct values for each block which are the values from the previous iteration. It means, we can get those correct values by going back to the beginning of the iteration.

Similarly, we verify the correctness the row panel update phase with the local checksums appended to the local matrices or blocks. In this case, if the local checksum shows an error, if the number of error is only one, we can cover that with employing the global checksums which are the summation of the local checksums in each row of the local matrix. If no error is detected in this phase, the LU factorization continues with executing the trailing matrix update which is the last phase LU factorization in each iteration. Similar

1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18
19	20	21	22	23	24	25	26	27
28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54
55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72

Figure 5.3: Original input matrix with block size of 4×3

to row panel update, the local checksums are used to detect the possible errors and the global checksums will be used to cover the potential errors. The number of errors can not exceed the one in each row of the trailing matrix. However, more number of errors can be covered if they are located in different rows of the trailing matrix update.

5.2.1 Checksum Setup

We have two types of checksums for the input matrix. Local checksum and global checksum. Local checksums are appended to each block of matrix and global checksums are the sums of all blocks which are added as a different block. In case of GPUs, these blocks will be executed as thread-block. Figure 5.3 shows the original matrix with a block size of 4×3. In a blocked LU factorization, the local row checksums are appended to each block of input matrix and the block width will be increased by one. The matrix with each local matrix with row checksums is shown in Figure 5.4. A local matrix's elements in a single

1	2	3	6	4	5	6	15	7	8	9	24
10	11	12	33	13	14	15	42	16	17	18	51
19	20	21	60	22	23	24	69	25	26	27	78
28	29	30	87	31	32	33	96	34	35	36	105
37	38	39	114	40	41	42	123	43	44	45	132
46	47	48	141	49	50	51	150	52	53	54	159
55	56	57	168	58	59	60	177	61	62	63	186
64	65	66	195	67	68	69	204	70	71	72	213

Figure 5.4: Original input matrix with local checksums added to each block. Block width is increased by 1

block of a row are summed, and the result is stored immediately following the block it is a part of. Checksum vectors for all local blocks are generated in the GPUs. To ensure that sums and their elements are always included in operations, the block size has been increased by one. Operating on a sum and the elements simultaneously is necessary for maintaining checksums at a fine-grained level. Also, Figure 5.5 shows the matrix with the appended global checksum as an additional blocks. These sums are sum of corresponding elements in the local matrices across the rows. Along with the local checksums that are shown in the last column of each block, global checksums are added to the input matrix. The global checksum is also generated by the GPU. Based on the block size, and the input matrix size, the number of blocks for global checksums will be varied. For instance, in case of global row checksum, if the input matrix is height is 10k and the block size is 512×512, there will be a total of 20 blocks of global checksum.

Periodically, the local checksum is verified; if the sum is incorrect, the global sum is used for recovery. At the end of each iteration, the checksum relationship between elements

1	2	3	6	4	5	6	15	7	8	9	24	12	15	18	45
10	11	12	33	13	14	15	42	16	17	18	51	39	42	45	126
19	20	21	60	22	23	24	69	25	26	27	78	66	69	72	207
28	29	30	87	31	32	33	96	34	35	36	105	93	96	99	288
37	38	39	114	40	41	42	123	43	44	45	132	120	123	126	369
46	47	48	141	49	50	51	150	52	53	54	159	147	150	153	450
55	56	57	168	58	59	60	177	61	62	63	186	174	177	180	531
64	65	66	195	67	68	69	204	70	71	72	213	201	204	207	612

Figure 5.5: Original input matrix with local and global checksums added

in the same row is maintained in the right looking LU factorization. However, to make sure that local checksums will be maintained at the end of each phase of the iteration, the right looking LU factorization performs the operations on the level of each block with the appended local checksums. For example, after the panel factorization, since only the elements within the panel are involved in the computation, the checksum relationship will be maintained. After the above mentioned steps, we check the correctness of computation by verifying the checksum relationship. To reduce the overhead of the verification, only local checksums are are used in this phase (detection phase). In case of panel factorization, when an error is detected, due to an error propagation, the panel must be entirely replaced with a recovered version. The recovered panel will therefore be the one from the prior iteration because it is built using components that are still in their pre-iteration state. In this approach, there is no other method for extracting the correct state. Any panel factorization error will require a recovery to right before the panel factorization. In other words, if any error is detected in this phase, we go back to the beginning of the loop and repeat the panel factorization.

Similar to panel factorization, the local checksums are used for the verification of row panel update and trailing matrix update phases as well. However; in these phases, if the error is detected, we use global checksum to cover the errors. If there are more than one error in the same row, global checksum can not recover the error. However, we can tolerate more number of faults if they are located in different rows. Also, if there are more number of errors within only one TB, the whole TB can be recovered using the corresponding global checksum. This is true, when there no errors on the other TBs across the same row. The global sums which will be used in recovery of trailing matrix update and row panel update, are executed concurrently with the local checksums. If any error is detected by the local checksum in one of the TBs, in order to be able to recover the error using the global checksum, we need to wait for the all TBs that are in the same row as the faulty TB to be finished.

5.2.2 Local Checksum After Panel Factorization

In general, recalculating the sums and comparing to the stored values are required for checksum verification. The overhead could be substantial due to the memory accesses. All checksums are either of U or the original matrix. The checksum indicates the fact that the equivalent point in U has a zero once the elements that made up the checksum have been replaced with components of L. When the panel is factored, the first nb $\times$ nb block contains a portion of U and the rest is L. The checksums will all be zeros in this area. Consequently, for the majority of the panel verification, no computation is needed, and memory access is minimized.

After a panel is factored, the sums for the portion of the panel that becomes L in a checksum matrix, where local row sums are computed for each block, will be zero. This is due to the algorithms' transformation of the sums from sums on the original matrix to sums on U alone. The U matrix contains zeros in the location where the L matrix is stored. Thus, the sums of zeros will zero. If there are no errors, the L sums are 0 in this case [24].

5.2.3 Proof Of Checksum Correctness During The Right Looking LU Factorization

In each iteration of right looking LU factorization, the trailing matrix from the previous step is used for the next iteration. The L and U parts are not changed again after each step. In one factorization step, a matrix with row checksums is transformed into panels of L and U and a trailing matrix with correct row checksums. Since the checksum is maintained in one step in the right looking LU factorization, it is preserved throughout the entire calculation [24]. In [24], it is proved that if the matrix has correct global and local checksums at the beginning of the main loop iteration in the right looking LU factorization, then the resulting matrix sections of U and the trailing matrix would also have correct global and local checksums after each iteration. In other words, the trailing matrix from the previous iteration is used in each subsequent iteration. The trailing matrix starts the iteration with all checksums within the trailing matrix consisting entirely of elements from the trailing matrix. As a result, it can be dealt as a whole matrix, separate from the parts that have previously been fully factored.

5.3 Evaluation

All experiments are performed on heterogeneous system with Two GPUs, the architectural specifications can be found in Table 5.1.

5.3.1 Checksum Setup And Detection Phase

For detection phase, along with the checksum setup, the detection phase should be done in each iteration. If there is no error, the overhead is the time it takes to check the local checksums after each phase. Each iteration involves three checksum verification: one for the panel, one for row panel update and another for the trailing matrix. The correctness of the elements must be verified by adding the elements inside each row of the block/TB, after which the result must be compared to the value that was previously stored. However, as explained earlier, for panel factorization, except one block, these number should be all zeros if no error has occurred. Therefore, it is not needed to add all the row elements of the panel. Hence, the overhead of the verification comes from only comparing the elements

Table 5.1: Experimental setup configuration

Component	**CPU**	**GPU**
Architecture	Intel(R) Core(TM) i7-6700k	NVIDIA GeForce GTX 1660 SUPER
Minimum Frequency	800 MHz	300 MHz
Maximum Frequency	4200	1980
Peak Performance	Intel	
Memory	16 GB	6 GB
Cache	L1(128 KB) L2 (1 MB) L3 (8 MB)	L1 (64 KB per SM) L2 (1536 KB)
OS	Ubuntu 20.04	

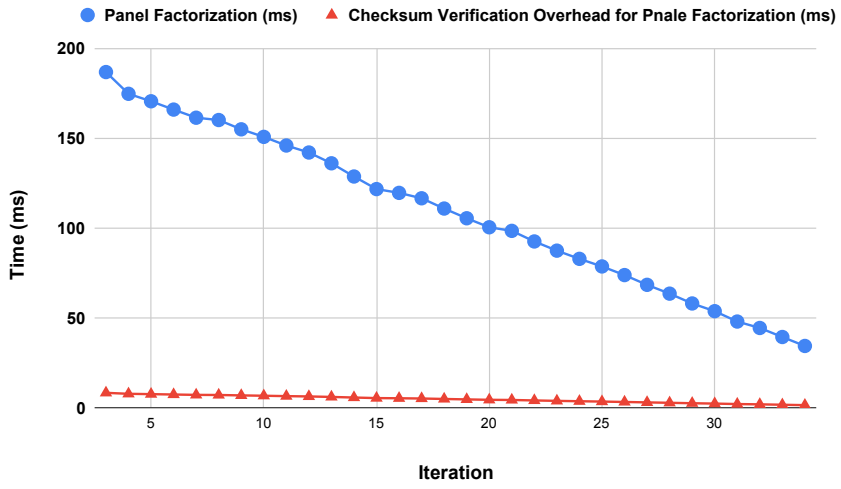

Figure 5.6: The checksum verification overhead of panel factorization

with zero. For row panel update and trailing matrix update, each element inside a block in the local matrix needs to be added together and validated. Using this method, we can guarantee that all the errors within different rows can be covered but it is not possible to cover multiple errors if they happen to be in different TBs across the same row. Multiple errors in the same block/TB can be corrected if no other TB/block is faulty. These blocks can be completely recovered from the global matrix.

We evaluated the checksum verification/detection of panel factorization, row panel update, and trailing matrix update. Figure 5.6 shows the the performance overhead of checksum verification for the panel factorization phase. The X-axis shows the iteration number and the Y-axis shows the execution time. The blue points shows the panel factorization time whereas the red line shows the time taken for the verification. The results are extracted while the CPU is running at the peak frequency of 4 GHz.

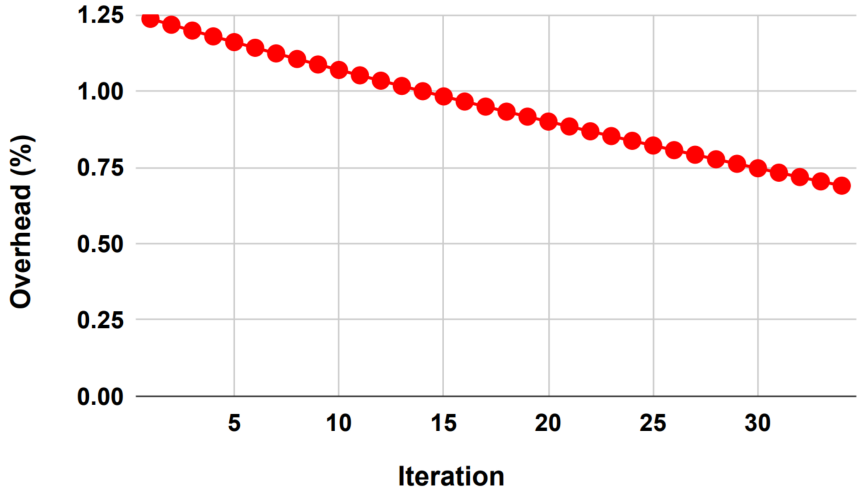

Figure 5.7: The checksum verification overhead of row panel update

We also extracted the results for checksum setup and verification in presence of different iterations. The checksum verification verifies the correctness of the panel with recalculating the sums of one block as well comparing the rest of the panel with zeros. Figure 5.8 shows the overhead of the checksum setup and panel factorization verification in compare to the total execution time the panel factorization. The maximum overhead is observed at the earlier iteration due to larger number of comparisons needed for verification. As the LU factorization proceeds, the height of the panel decreases and the number of comparisons decrease as well.

We also extracted the performance overhead of the checksum verification for row panel update and trailing matrix update phases. Figures 5.7 and 5.9 show the performance overhead of the checksum detection of the row panel update and trailing matrix update phases, respectively. X-axis shows the iteration number and Y-axis shows the performance overhead of the only detection phase. The detection phase's overhead is less than 1.25%

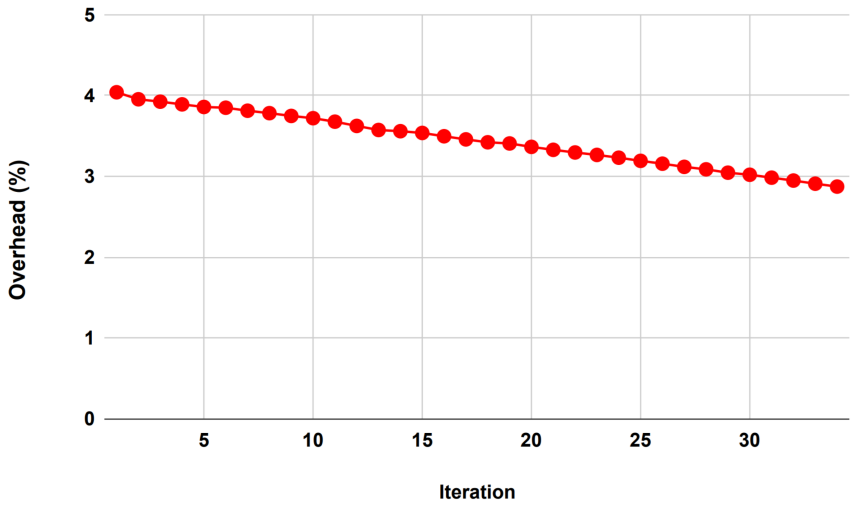

Figure 5.8: The checksum setup and verification overhead of panel factorization

and 1.5% for row panel update and trailing matrix update phases. This overhead does not include the checksum setup overhead and recovery overhead. The checksum setup overhead is similar to the checksum setup overhead of the panel factorization, since checksums for all phases including the panel factorization is generated on the GPUs due to its parallel nature.

We also evaluated the overhead of the global checksum generation. For checksum global generation, all the input matrix is stored on the GPU and the global checksum is

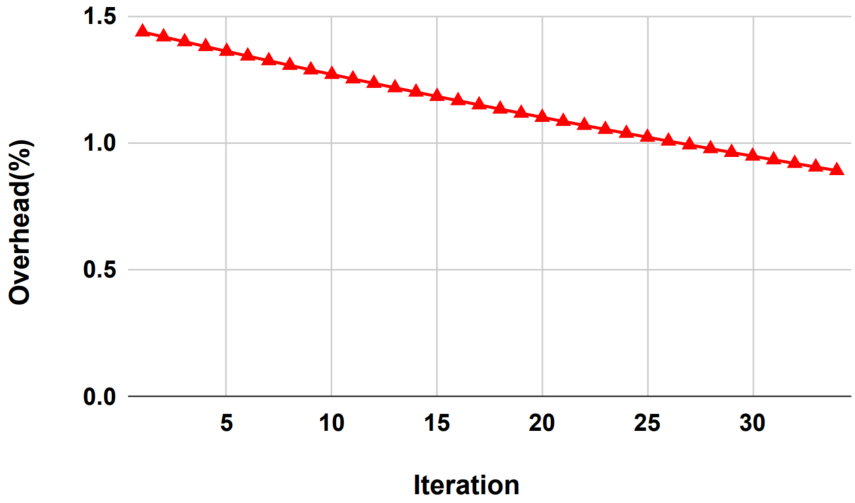

Figure 5.9: The checksum verification overhead of trailing matrix update

built by adding the same elements of different blocks. In compare to the baseline, the execution time of the LU factorization was increased by 5.6%. To find the actual overhead, this will be added to the overhead of detection and recovery as well. We also, calculated the overhead of the extra computation comes from the extra TBs belong to the global checksum (additional TBs). In case of matrix of size 18K with the block size of 512×512, we build the global checksum blocks/TBs across the rows. So, the number of extra TBs needed for global checksum will be 36. The total number of TBs for matrix of size 18K without the global checksum TBs is 1296. So the computation overhead coming from the extra TBs will be 2.7%.

Chapter 6

Related Work

This chapter covers the literature related to the studies proposed in this thesis. First, we provide the related work for DVFS, it's drawbacks, and GPU undervolting. Next, we discuss the overclocking, undervolting and fault tolerant technique works, and then, discuss the previous energy efficient approaches on LU factorization.

6.1 GreenMM

The ever-increasing popularity of GPUs has motivated development of energy efficient GPU architectures, most of which target for energy saving in general over many applications. However, very few of the architecture designs are targeted at reducing energy consumption of linear algebra basic routines such as cuBLAS-MM that are used in scientific application.

Dynamic Voltage and Frequency Scaling (DVFS), is one common approach to reduce power and energy consumption of a system [54]. Applying DVFS, based on system

utilization, the processors can operate in different power states whenever high performance is not necessary. DVFS in GPU domain can behave in a very different manner compared to DVFS in CPUs in regard to energy efficiency [30]. Besides the DVFS technique, GPU undervolting is another approach for improving GPU energy efficiency. Leng et al. [49], reduce chip voltage of the GPU to V_{min} without introducing any errors; which was achieved by leveraging guard-band voltage of the GPU. However they did not go beyond V_{min} because errors would occur with any further undervolting. In our work, we show that even beyond V_{min} there is opportunity to save more energy and correct potential faults by combining undervolting and Algorithm Based Fault Tolerance (ABFT) together. GPUs/CPUs use huge number of communications links which have made them seriously prone to coupling and inductance effects [85] [84]. By using undervolting, we also could relax the coupling and inductance effect and increase the reliability. In [104] [37], power gating is applied onto GPU to save energy on branch divergence and idle components respectively. [3] applies dynamic resource allocation to improve GPU energy efficiency. [89] attempts to reduce energy consumption by selecting between the CPU or GPU to run the application. In CPU domain, there are several studies which rely on hardware sensors to look for possibilities to reduce the operating voltage by monitoring critical path [44]. In [93], Tan et al. investigated the interplay between energy efficiency and reliability on the CPUs. In their approach, they combined undervolting with a fault tolerant technique to tolerate faults caused by undervolting on the CPUs. Their fault rate model is based on digital circuit failure, and not based on the CPU hardware. It is because they could not drive CPU undervolting to below the threshold value to generate faults. So, they emulated the errors and corrected them. They

used an analytic fault model and only considered a single soft error to correct. In GreenMM, we introduce a fault model taking into account the real system faults during undervolting through NVML APIs. We drove the GPU voltage under the threshold ($V_{safeMin}$), so that the number of faults is practically measured. Thus, our proposed fault model is more accurate and realistic.

Fault tolerant mechanisms such as redundancy-based techniques [71] can recover from hard failures, however, at a huge performance cost. These techniques are not useful in GPU applications due to high penalty in terms of energy consumption and performance. Checkpointing has been applied to tolerate failures on the GPU through restarting application from some previously saved correct state [74]. Checkpointing suffers from significant performance and memory overheads. Compared to aforementioned techniques, ABFT provides the advantage of negligible overhead along with the capability of detecting and correcting errors with low overhead. ABFT has widely been studied for improving linear algebra library on both CPUs [99] [51] and GPUs [13].

6.2 SAOU

Checkpoint/Restart (CR) is the most widely used fault-tolerance method for long-running applications (CR). Checkpoint/Restart Checkpointing involves writing the current state of a running process to a checkpoint file, which may then be used to restore the current state of the process. In other words, periodically, CR takes a system snapshot and store it in secondary storage (checkpointing phase). The current state is replaced with the previously saved state in the event of a failure or migration, and execution continues from the most

recent checkpoint (restoration phase). For CPUs, a number of CR techniques have been developed at various levels including kernel, library, and application [83] [36].

Due to the lack of a specific operating system and appropriate runtime APIs to extract compute information inside the kernel, none of the aforementioned approaches are currently practical on NVIDIA GPUs. However, researchers have created a few GPU CR algorithms in recent years despite these limitations.

The first GPU CR implementation was CheCuda [91]. CUDA applications using CUDA driver API features are the current target of CheCUDA. CheCUDA releases CUDA-related resources and saves all data of those resources prior to checkpointing. In order to store the resource data, it overrides the CUDA resource data types with its own C++ class types. Applications send those class objects as parameters to API functions. To intercept API calls and call the relevant API functions with CUDA data, CheCUDA additionally substitutes wrapper functions for the API functions. Therefore, even though recompilation of the code is necessary, CR of a CUDA application does not need modifying the source code. CheCuda is built on top of Berkeley Lab Checkpoint/Restart (BLCR) library [35]. BLCR is one of the CR solutions for Linux systems. BLCR, which is based on unique kernel modules, permits checkpointing a wide variety of applications, including distributed and multi-threaded ones. Like many other implementations, BLCR includes callback functions to give processes that need to use resources that aren't being checkpointed a method to achieve CR. Furthermore, BLCR enables task migration between two distinct nodes if the same shared libraries are accessible on both of them; a CUDA application may resume based on a checkpoint file produced on a different node. Due to the fact that this library does not

support CUDA contexts, CUDA contexts are backed up and deleted before checkpointing, followed by BLCR checkpointing and then all destroyed GPU contexts are reallocated.

NVCR is another CR algorithm that was introduced in [68]. With NVCR, applications do not need to be recompiled since NVCR does not change the addresses of the CUDA context after reallocation. Another program that uses the same strategy as CheCuda is CheCL [88], but it was developed especially for OpenCL applications.

All of the CR solutions mentioned previously restart kernels and reload GPU state in the event of failure and they are unable to reload a thread's kernel-level computing state. With these CR algorithms, we are not able to continue the application's execution and we need to come up with the solutions that do not need to restart the kernel.

6.3 GreenMD

There has been a lot of work done in recent years to investigate and improve the energy efficiency of the kernels that are frequently utilized in scientific applications. Kernels like matrix multiplication, LU factorization, Cholesky, and QR decomposition are examples of such kernels. There are general methods that are employed to increase the energy efficiency of the applications. Numerous methods have been suggested, which can be categorized into the following categories: 1) Studies on the effects of DVFS on the execution of applications; 2) And works that introduce the runtime models to predict the GPU application performance and/or power consumption.

Jiao et al. [43] investigated the impacts of core and memory frequency on applications with various features with regard to the effects of DVFS on various applications. The

authors noted that as some applications were more sensitive than others to the scaling of each frequency domain, the effect of frequency scaling on performance and power consumption depended on the characteristics of the applications. An alternate strategy for DVFS requires the development of accurate performance and/or power models that enable GPU behavior prediction under various voltage and frequency conditions. In an effort to accurately represent the execution characteristics of GPGPU applications, GPU performance models are generally developed based on GPU pipeline analysis [38, 67, 87], seeking to properly reflect the execution characteristics of GPGPU applications. In some researches, the performance models are based on profiling as well as the algorithmic knowledge of the application. For instance, in [19], they introduce a performance model based on profiling the first iteration and then using the algorithmic knowledge to predict the next iterations. Since the error rate is only dependent on the profiling results of the first iteration, the error caused by profiling will be accumulated in the later iterations and will become around 11.4%. In other works [12], they use a similar approach, however, using an online calibration they avoid error from accumulating and reduce the error rate but this approach adds the overhead of the online calibration to the overhead of the performance model and needs application changes ahead of time.

Some of the GPU DVFS runtime power modeling techniques are based on empirical techniques, which call for the division of GPU micro-architectures and the analysis of the kernel binary code [48]. Also, in [106], the authors use a micro-benchmark-based methodology to create a throughput model for the instruction pipeline, shared memory access, and global memory access, the three main components of GPU execution time and they are able to

predict the performance of the GPU with a 5–15% error rate. However, these methods are frequently product-specific and difficult to port to other architectures.

To utilize the DVFS, we introduce a simple model with up to 5% performance overhead. Our performance model is based on the general GPU performance model which could be used for different compute-intensive applications with a slight modification. The introduced performance models

6.4 Fault Tolerant LU Decomposition

Almost every component of a computing system is susceptible to both hard and soft errors. These errors can either result in inaccurate results or an anomalous termination of the computing process; the former is known as a soft error, and the latter is known as a hard error.

In contrast to hard error failures, soft errors are more harder to identify at runtime, making them significantly more challenging to deal with [52] [17] [14] [102]. Recent studies have revealed that soft errors happen in the GPUs as well and the rate of soft errors substantially increases as the GPU workload increases [92].

As a general strategy, Triple Modular Redundancy (TMR) [58] may tolerate soft errors. TMR runs three identical computation and report the output based on the majority voting. Despite being a universal strategy that may be used for any application, it has an extremely high overhead. To solve this problem. Later research proposed online ABFT, which makes error correction considerably simpler and may detect faults during computation when they are not propagated to the rest of the computation which makes it easier to deal

with. Algorithm-based fault tolerance (ABFT) has frequently been used to identify errors in matrix operations. Using this method, the matrices being operated on are assigned with a row or column checksum. Many matrix operations can be verified to be valid at the end of the calculation using a checksum, and these checksums can also be used to identify possible errors after the computation is complete. ABFT can also be used for LU factorization. If one checksum vector is used, it can only detect the error while with having both row and column checksum vectors, it can detect and correct the error as well [4] [40] [40] [96].

Wu et al. proposed an online ABFT for matrix multiplication in [101]. Davies introduced an online ABFT for LU decomposition in [23]. On distributed memory computing systems, Wu et al. extended online ABFT to three fundamental one-sided matrix decompositions: Cholesky, LU, and QR [100]. When performing Cholesky decomposition on GPUs, Chen et al. developed an online ABFT technique to protect against both memory and computation error [18] [15]. a comprehensive checksum-based LU decomposition was introduced by Wu et al. to manage memory errors in CPU cache and registers [103].

For many matrix operations, the checksum method has been used to identify errors and correct them after the calculation. In some implementation of the LU factorization, we can also validate the relationship of the checksum in the middle of the computation. This will result in preventing the errors from being propagated to the rest of the matrix which makes it difficult to correct the errors. The incorrect elements in the matrix can be identified and corrected using a checksum of a matrix. In some instances, one checksum is needed to identify the error and a second is required to correct it. Diskless checkpointing is another approach that is used for long running kernels. The data must be saved before performing a checkpoint

so that it can be restored in the event of a failure. Saving the date to the disk is one of approaches can be used. However, the low speed of writing and reading to the disk could be a bottleneck if the disk needs to be shared between several processes. In case of failure, all processes are restarted from the point of the previous checkpoint, and their data is restored using the data stored on the disk. Diskless checkpointing [75] was developed to address this problem. To avoid the slow write to disk, each processor stores its own checkpoint state in memory. Diskless checkpointing and checksum-based checkpointing share common similarities. Each checksum processor functions as the redundancy processor in a diskless checkpoint when a checksum row is added to a processor grid. The difference is that the technique naturally maintains the checksum data's redundancy during the computation and there is no need to checkpoint the data anymore. As a result, there are two key advantages: the working processors do not need to use additional memory to store their checkpoint data, and there is less communication overhead when a checkpoint is made.

Chapter 7

Conclusion And Future Work

In this thesis, we introduced GreenMM, SAOU, GreenMD, and fault tolerant LU factorization. These studies investigate the energy efficiency and reliability of key kernels from high performance applications.

GreenMM presented a technique to save energy in GPUs through undervolting. First, we profiled error distribution of different applications from Rodinia benchmark to create an empirical fault model based on behaviour of the applications, while reducing the GPU voltage beyond V_{min}. After this point, the most predominant error was SDC error, which can be corrected at the application level. Then, we designed a ABFT based fault tolerant matrix multiplication algorithm to correct the errors. We evaluated energy consumption and performance on NVIDIA GTX 980. Our experiments showed that energy consumption can be reduced up to 19.8% using GreenMM, with performance overhead of 1.5%. Moreover, The $GFLOPS/WATT$ improvement of the GreenMM in comparison to the original MM for a matrix of size 10K is 9%.

In SAOU, we proposed an energy efficient framework, that reduces energy consumption of GPUs through undervolting and overclocking. Since going beyond the safe frequency or voltage give rise to faults, we designed a checkpoint and recovery (CR) technique to handle these faults. We created an empirical fault model to determine the number of checkpoints at each level of undervolting and overclocking. For a MM with matrix of size 10K, SAOU is able to save the energy consumption up to 22% through combined undervolting and overclocking without sacrificing the performance.

With GreenMD, we presented an energy efficient framework to improve the energy consumption of LU factorization on a heterogeneous mutil-GPU system. We profiled the execution trace of a system with two GPUs and explained various kernel executions and data transfers. This gave rise to detailed analytical models to predict the CPU/GPU execution time, and PCIe bus transfer time. Then we applied DVFS to CPU and GPUs based on the amount of slack predicted through our analysis. We designed appropriate APIs and inserted them into the kernel to independently control the DVFS at each iteration. We further improved the energy by reducing the voltage of CPU and GPUs independently based on the minimum threshold voltages to avoid error. GreenMD's energy consumption was evaluated for two heterogeneous systems, one with a single GPU and the other with two GPUs for a matrix size of 18K*18K. Our results showed that CPU and GPU energy consumption improved by around 59% and 18%, respectively in a heterogeneous system with a single GPU. Also, in a heterogeneous system with two GPUs, GreenMD reduces the energy consumption of the CPU and GPU by 51% and 21%, respectively. The total energy saved during the entire execution is 31%.

Also, we suggested a fault-tolerant technique for the LU factorization by incorporating local and global row checksums into the initial matrix. We targeted the soft errors that are becoming predominant types of errors nowadays. The local checksums were applied to each local block of the matrix. The global checksums were the sums of the local matrices and were placed at the end of the original matrix. In order to find errors in the different phases of the LU factorization, local checksum were used. Recovery during the panel factorization phase was achieved by rolling back to the previous state, or in this case, simply returning to the beginning of the iteration. And the recovery for the row panel update and tail matrix update phases was performed using the global checksum blocks. To enhance the fault coverage capability, local checksums were added to each block and fault tolerance was done in a block level. Also, to reduce the performance overhead of the FT algorithm, both local and global checksums were generated in the GPUs. To even reduce the performance overhead more, the frequency of the verification in less sensitive sections (row panel update and trailing matrix) can be adjusted based on the error rate.

Future work

There are many potential directions for future work in improving the energy efficiency of high-performance applications on heterogeneous systems with multiple GPUs. One potential area of focus is the development of more advanced power management techniques, such as those that can dynamically adjust voltage and frequency in response to changes in workload and other factors. Another possibility is to investigate new approaches to algorithm optimization, such as using machine learning techniques to identify and eliminate inefficiencies in code. Additionally, researchers could explore the use of hybrid systems that combine multiple types of hardware, such as CPUs and GPUs, in order to achieve more efficient systems. Another area of potential research is the use of energy-aware scheduling strategies to optimize energy consumption on heterogeneous systems, taking into account factors such as workload, and hardware utilization.

Bibliography

[1] AmirAli Abdolrashidi, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi Narayan Bhuyan, and Daniel Wong. Wireframe: Supporting data-dependent parallelism through dependency graph execution in gpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 600–611, 2017.

[2] MSI Afterburner. http://goo.gl/fs2pti.

[3] Pedro Alonso, Manuel F Dolz, Francisco D Igual, Rafael Mayo, and Enrique S Quintana-Orti. Reducing energy consumption of dense linear algebra operations on hybrid cpu-gpu platforms. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 56–62. IEEE, 2012.

[4] C.J. Anfinson and F.T. Luk. A linear algebraic model of algorithm-based fault tolerance. *IEEE Transactions on Computers*, 37(12):1599–1604, 1988.

[5] Rizwana Begum, Mark Hempstead, Guru Prasad Srinivasa, and Geoffrey Challen. Algorithms for CPU and DRAM DVFS under inefficiency constraints, 2016.

[6] Susan Blackford. LAPACK users' guide, 1997.

[7] Susan Blackford. ScaLAPACK users' guide, 1997.

[8] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[9] Enrico Calore, Alessandro Gabbana, Sebastiano Fabio Schifano, and Raffaele Tripiccione. Evaluation of dvfs techniques on modern hpc processors and accelerators for energy-aware applications. *Concurrency and Computation: Practice and Experience*, 29(12):e4143, 2017.

[10] Saumya Chandra, Kanishka Lahiri, Anand Raghunathan, and Sujit Dey. Variation-tolerant dynamic power management at the system-level. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(9):1220–1232, 2009.

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[12] Jieyang Chen, Hongbo Li, Sihuan Li, Xin Liang, Panruo Wu, Dingwen Tao, Kaiming Ouyang, Yuanlai Liu, Kai Zhao, Qiang Guan, and Zizhong Chen. Fault tolerant one-sided matrix decompositions on heterogeneous systems with gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 854–865, 2018.

[13] Jieyang Chen, Sihuan Li, and Zizhong Chen. Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus. In *Networking, Architecture and Storage (NAS), 2016 IEEE International Conference on*, pages 1–2. IEEE, 2016.

[14] Jieyang Chen, Sihuan Li, and Zizhong Chen. Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus. In *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–2, 2016.

[15] Jieyang Chen, Sihuan Li, and Zizhong Chen. Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus. In *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–2, 2016.

[16] Jieyang Chen, Xin Liang, and Zizhong Chen. Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 993–1002. IEEE, 2016.

[17] Jieyang Chen, Xin Liang, and Zizhong Chen. Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 993–1002, 2016.

[18] Jieyang Chen, Xin Liang, and Zizhong Chen. Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 993–1002, 2016.

[19] Jieyang Chen, Li Tan, Panruo Wu, Dingwen Tao, Hongbo Li, Xin Liang, Sihuan Li, Rong Ge, Laxmi Bhuyan, and Zizhong Chen. Greenla: green linear algebra software for gpu-accelerated heterogeneous computing. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 667–677. IEEE, 2016.

[20] Zizhong Chen. Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[21] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, 1993.

[22] Shidhartha Das, David Roberts, Seokwoo Lee, Sanjay Pant, David Blaauw, Todd Austin, Krisztián Flautner, and Trevor Mudge. A self-tuning dvs processor using delay-error detection and correction. *IEEE Journal of Solid-State Circuits*, 41(4):792–804, 2006.

[23] Teresa Davies and Zizhong Chen. Correcting soft errors online in lu factorization. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '13, page 167–178, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Teresa Davies and Zizhong Chen. Correcting soft errors online in lu factorization. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '13, page 167–178, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High performance linpack benchmark: A fault tolerant implementation without checkpointing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, page 162–171, New York, NY, USA, 2011. Association for Computing Machinery.

[26] K Dev, S Reda, I Paul, W Huang, and W Burleson. Workload-Aware power gating design and Run-Time management for massively parallel GPGPUs. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 242–247, July 2016.

[27] Chong Ding, Christer Karlsson, Hui Liu, Teresa Davies, and Zizhong Chen. Matrix multiplication on gpus with on-line fault tolerance. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pages 311–317. IEEE, 2011.

[28] S Donfack, S Tomov, and J Dongarra. Dynamically balanced Synchronization-Avoiding LU factorization with multicore and GPUs. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 958–965, May 2014.

[29] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[30] Rong Ge, Ryan Vogt, Jahangir Majumder, Arif Alam, Martin Burtscher, and Ziliang Zong. Effects of dynamic voltage and frequency scaling on a k20 gpu. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 826–833, Washington, DC, USA, 2013. IEEE Computer Society.

[31] Dimitris Gizopoulos, George Papadimitriou, Athanasios Chatzidimitriou, Vijay Janapa Reddi, Behzad Salami, Osman S Unsal, Adrian Cristal Kestelman, and Jingwen Leng. Modern hardware margins: Cpus, gpus, fpgas.

[32] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás. Dvfs-aware application classification to improve gpgpus energy efficiency. *Parallel Computing*, 83:93–117, 2019.

[33] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás. Modeling and decoupling the GPU power consumption for Cross-Domain DVFS. *IEEE Trans. Parallel Distrib. Syst.*, 30(11):2494–2506, November 2019.

[34] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás. Dvfs-aware application classification to improve gpgpus energy efficiency. *Parallel Computing*, 2018.

[35] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physcs: Conference Series*.

[36] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, 46(1):494, sep 2006.

[37] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 280–289, New York, NY, USA, 2010. ACM.

[38] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 280–289, 2010.

[39] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984.

[40] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984.

[41] Kuang-Hua Huang et al. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6):518–528, 1984.

[42] Kuang-Hua Huang et al. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6):518–528, 1984.

[43] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and performance characterization of computational kernels on the gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications Int'l Conference on Cyber, Physical and Social Computing*, pages 221–228, 2010.

[44] Charles R. Lefurgy, Alan J. Drake, Michael S. Floyd, Malcolm S. Allen-Ware, Bishop Brock, Jose A. Tierno, and John B. Carter. Active management of timing guardband to save energy in power7. In *Proceedings of the 44th Annual IEEE/ACM International*

Symposium on Microarchitecture, MICRO-44, pages 1–11, New York, NY, USA, 2011. ACM.

[45] J. Leng, Y. Zu, M. Rhu, M. S. Gupta, and V. J. Reddi. Gpuvolt: Modeling and characterizing voltage noise in gpu architectures. In *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 141–146, Aug 2014.

[46] Jingwen Leng, Alper Buyuktosunoglu, Ramon Bertran, Pradip Bose, and Vijay Janapa Reddi. Safe limits on voltage reduction efficiency in gpus: A direct measurement approach. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 294–307, New York, NY, USA. ACM.

[47] Jingwen Leng, Alper Buyuktosunoglu, Ramon Bertran, Pradip Bose, and Vijay Janapa Reddi. Safe limits on voltage reduction efficiency in gpus: A direct measurement approach. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 294–307, New York, NY, USA. ACM.

[48] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. *ACM SIGARCH Computer Architecture News*, 41(3):487–498, 2013.

[49] Jingwen Leng, Yazhou Zu, and Vijay Janapa Reddi. Energy efficiency benefits of reducing the voltage guardband on the kepler gpu architecture. *Proc. of SELSE*, 2014.

[50] Jingwen Leng, Yazhou Zu, and Vijay Janapa Reddi. Energy efficiency benefits of reducing the voltage guardband on the kepler gpu architecture. In *Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2014.

[51] Xin Liang, Jieyang Chen, Dingwen Tao, Sihuan Li, Panruo Wu, Hongbo Li, Kaiming Ouyang, Yuanlai Liu, Fengguang Song, and Zizhong Chen. Correcting soft errors online in fast fourier transform. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 30. ACM, 2017.

[52] Xin Liang, Jieyang Chen, Dingwen Tao, Sihuan Li, Panruo Wu, Hongbo Li, Kaiming Ouyang, Yuanlai Liu, Fengguang Song, and Zizhong Chen. Correcting soft errors online in fast fourier transform. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[53] Ching-Chi Lin, You-Cheng Syu, Chao-Jui Chang, Jan-Jan Wu, Pangfeng Liu, Po-Wen Cheng, and Wei-Te Hsu. Energy-efficient task scheduling for multi-core platforms with per-core DVFS. *J. Parallel Distrib. Comput.*, 86:71–81, December 2015.

[54] Wenjie Liu, Zhihui Du, Yu Xiao, David A Bader, and Chen Xu. A waterfall model to achieve energy efficient tasks mapping for large scale gpu clusters. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 82–92. IEEE, 2011.

[55] Xavier Luciani and Laurent Albera. Joint eigenvalue decomposition of non-defective matrices based on the lu factorization with application to ica. *IEEE Transactions on Signal Processing*, 63(17):4594–4608, 2015.

[56] Mark Harris Luke Durant, Olivier Giroux and Nick Stam. Basic linear algebra subprograms technical(blast) forum standard, 2001.

[57] Mark Harris Luke Durant, Olivier Giroux and Nick Stam. Volta whitepaper, 2017.

[58] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[59] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Xiaorui Wang. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 48–57. IEEE, 2012.

[60] Xinxin Mei, Ling Sing Yung, Kaiyong Zhao, and Xiaowen Chu. A measurement study of gpu dvfs on energy conservation. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, pages 10:1–10:5, New York, NY, USA, 2013. ACM.

[61] Xinxin Mei, Ling Sing Yung, Kaiyong Zhao, and Xiaowen Chu. A measurement study of gpu dvfs on energy conservation. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, pages 1–5, 2013.

[62] G Memik, M H Chowdhury, A Mallik, and Y I Ismail. Engineering Over-Clocking: Reliability-Performance Trade-Offs for High-Performance register files. *2005 International Conference on Dependable Systems and Networks (DSN'05)*.

[63] RM Miller. Exascale Computing, 2013.

[64] A Moody, G Bronevetsky, K Mohror, and B R d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, November 2010.

[65] DN Prabhakar Murthy, Min Xie, and Renyan Jiang. *Weibull models*, volume 505. John Wiley & Sons, 2004.

[66] Sayori Nakagawa, Satoshi Fukumoto, and Naohiro Ishii. Optimal checkpointing intervals of three error detection schemes by a double modular redundancy. *Mathematical and Computer Modelling*, 38(11):1357 – 1363, 2003. Stochastic models in engineering, technology, and management.

[67] Rajib Nath and Dean Tullsen. The crisp performance model for dynamic voltage and frequency scaling in a gpgpu. In *Proceedings of the 48th international symposium on microarchitecture*, pages 281–293, 2015.

[68] N A Nvcr. A transparent checkpoint-restart library for nvidia cuda. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW*, 2011.

[69] D. A. G. Oliveira, P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro. Gpgpus ecc efficiency and efficacy. In *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 209–215, Oct 2014.

[70] George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Dimitris Gizopoulos, Peter Lawthers, and Shidhartha Das. Harnessing voltage margins for energy efficiency in multicore cpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 503–516, 2017.

[71] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques,. Technical report, Berkeley, CA, USA, 2002.

[72] Antoine Petitet. Hpl-a portable implementation of the high-performance linpack benchmark for distributed-memory computers. *http://www. netlib. org/benchmark/hpl/*, 2004.

[73] Laercio L Pilla, P Rech, F Silvestri, Christopher Frost, Philippe Olivier Alexandre Navaux, M Sonza Reorda, and Luigi Carro. Software-based hardening strategies for neutron sensitive fft algorithms on gpus. *IEEE Transactions on Nuclear Science*, 61(4):1874–1880, 2014.

[74] James S Plank, Micah Beck, Gerry Kingsley, and Kai Li. *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.

[75] J.S. Plank, Kai Li, and M.A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[76] Behnam Pourghassemi and Aparna Chandramowlishwaran. cudaCR: An In-Kernel Application-Level Checkpoint/Restart scheme for CUDA-Enabled GPUs. *CLUSTER 2017*.

[77] Ronald L Rivest and Charles E Leiserson. *Introduction to algorithms*. McGraw-Hill, Inc., 1990.

[78] N. Rohbani, M. Ebrahimi, S. Miremadi, and M. B. Tahoori. Bias temperature instability mitigation via adaptive cache size management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(3):1012–1022, March 2017.

[79] N. Rohbani, Z. Shirmohammadi, M. Zare, and S. Miremadi. Laxy: A location-based aging-resilient xy-yx routing algorithm for network on chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(10):1725–1738, Oct 2017.

[80] Barry Rountree, David K Lowenthal, Bronis R De Supinski, Martin Schulz, Vincent W Freeh, and Tyler Bletsch. Adagio: making dvs practical for complex hpc applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 460–469, 2009.

[81] Barry Rountree, David K Lowenthal, Shelby Funk, Vincent W Freeh, Bronis R De Supinski, and Martin Schulz. Bounding energy consumption in large-scale mpi programs. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–9. IEEE, 2007.

[82] Smruti R. Sarangi, Brian Greskamp, Radu Teodorescu, Jun Nakano, Abhishek Tiwari, and Josep Torrellas. Varius: A model of process variation and resulting timing errors for microarchitects. *IEEE Transactions on Semiconductor Manufacturing*, 21(1):3–13, 2008.

[83] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul V. Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. *Proceedings of the ACM/IEEE SC2004 Conference*, pages 38–38, 2004.

[84] Z. Shirmohammadi and H. Z. Sabzi. Dr: Overhead efficient rlc crosstalk avoidance code. In *2018 8th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 63–68, Oct 2018.

[85] Z. Shirmohammadi, H. Z. Sabzi, and S. G. Miremadi. 3d-dycac: Dynamic numerical-based mechanism for reducing crosstalk faults in 3d ics. In *2017 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 87–90, Oct 2017.

[86] Steven S Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.

[87] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W Cameron. A simplified and accurate model of power-performance efficiency on emergent gpu architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 673–686. IEEE, 2013.

[88] H Takizawa, K Koyama, K Sato, K Komatsu, and H Kobayashi. CheCL: Transparent checkpointing and process migration of OpenCL applications. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 864–876, May 2011.

[89] H. Takizawa, K. Sato, and H. Kobayashi. Sprat: Runtime processor selection for energy-aware computing. In *2008 IEEE International Conference on Cluster Computing*, pages 386–393, 2008.

[90] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. CheCUDA: A Checkpoint/Restart tool for CUDA applications. *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2009.

[91] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. Checuda: A checkpoint/restart tool for cuda applications. In *Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '09, page 408–413, USA, 2009. IEEE Computer Society.

[92] Jingweijia Tan, Nilanjan Goswami, Tao Li, and Xin Fu. Analyzing soft-error vulnerability on gpgpu microarchitecture. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 226–235, 2011.

[93] L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge, and D. J. Kerbyson. Investigating the interplay between energy efficiency and resilience in high performance computing. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 786–796, May 2015.

[94] Li Tan, Shuaiwen Leon Song, Panruo Wu, Zizhong Chen, Rong Ge, and Darren J Kerbyson. Investigating the interplay between energy efficiency and resilience in high performance computing. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 786–796. IEEE, 2015.

[95] Zhenheng Tang, Yuxin Wang, Qiang Wang, and Xiaowen Chu. The impact of GPU DVFS on the energy and performance of deep learning: an empirical study. In *Proceedings of the Tenth ACM International Conference on Future Energy Systems*, e-Energy '19, pages 315–325, New York, NY, USA, June 2019. Association for Computing Machinery.

[96] M. Vijay and R. Mittal. Algorithm-based fault tolerance: a review. *Microprocessors and Microsystems*, 21(3):151–161, 1997. Fault Tolerant Computing.

[97] Q. Wang, J. Ohmura, S. Axida, T. Miyoshi, H. Irie, and T. Yoshinaga. Parallel matrix-matrix multiplication based on hpl with a gpu-accelerated pc cluster. In *2010 First International Conference on Networking and Computing*, pages 243–248, Nov 2010.

[98] Panruo Wu and Zizhong Chen. FT-ScaLAPACK: correcting soft errors on-line for ScaLAPACK cholesky, QR, and LU factorization routines. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, HPDC '14, pages 49–60. Association for Computing Machinery, June 2014.

[99] Panruo Wu and Zizhong Chen. Ft-scalapack: Correcting soft errors on-line for scalapack cholesky, qr, and lu factorization routines. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 49–60. ACM, 2014.

[100] Panruo Wu and Zizhong Chen. Ft-scalapack: Correcting soft errors on-line for scalapack cholesky, qr, and lu factorization routines. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 49–60, New York, NY, USA, 2014. Association for Computing Machinery.

[101] Panruo Wu, Chong Ding, Longxiang Chen, Feng Gao, Teresa Davies, Christer Karlsson, and Zizhong Chen. Fault tolerant matrix-matrix multiplication: Correcting soft errors on-line. In *Proceedings of the Second Workshop on Scalable Algorithms for Large-Scale Systems*, ScalA '11, page 25–28, New York, NY, USA, 2011. Association for Computing Machinery.

[102] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. Towards practical algorithm based fault tolerance in dense linear algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, page 31–42, New York, NY, USA, 2016. Association for Computing Machinery.

[103] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. Towards practical algorithm based fault tolerance in dense linear algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, page 31–42, New York, NY, USA, 2016. Association for Computing Machinery.

[104] Qiumin Xu and Murali Annavaram. Pats: pattern aware scheduling and power gating for gpgpus. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, pages 225–236. IEEE, 2014.

[105] H Zamani, Y Liu, D Tripathy, L Bhuyan, and others. GreenMM: energy efficient GPU matrix multiplication through undervolting. *Proceedings of the ACM*, 2019.

[106] Yao Zhang and John D. Owens. A quantitative performance analysis model for gpu architectures. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 382–393, 2011.

[107] Yazhou Zu, Charles R Lefurgy, Jingwen Leng, Matthew Halpern, Michael S Floyd, and Vijay Janapa Reddi. Adaptive guardband scheduling to improve system-level efficiency of the power7+. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 308–321. ACM.

[108] Yazhou Zu, Charles R Lefurgy, Jingwen Leng, Matthew Halpern, Michael S Floyd, and Vijay Janapa Reddi. Adaptive guardband scheduling to improve system-level efficiency of the power7+. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 308–321. IEEE, 2015.