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

Hadi Zamani
University of California, Riverside
hzama001@ucr.edu

Devashree Tripathy
University of California, Riverside
devashree.tripathy@email.ucr.edu

Laxmi Bhuyan
University of California, Riverside
bhuyan@cs.ucr.edu

Zizhong Chen
University of California, Riverside
chen@cs.ucr.edu

ABSTRACT

The current trend of ever-increasing performance in scientific applications comes with tremendous growth in energy consumption. In this paper, 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 can set the undervolting and overclocking level for maximum energy saving. We target cuBLAS Matrix Multiplication (cuBLAS-MM) kernel for error correction using the checkpoint and recovery (CR) technique as an example of scientific applications. In case of cuBLAS, SAOU achieves up to 22% energy reduction through undervolting and overclocking without sacrificing the performance.

KEYWORDS

Overclocking, Undervolting, Checkpoint-Recovery, Energy Efficiency

1 INTRODUCTION

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 [1–3]. However, DVFS and power gating techniques usually degrade performance due to lowering the frequency or putting the components into sleep mode.

ISLPED '20, August 10–12, 2020, Boston, MA, USA

ACM ISBN 978-1-4503-7053-0/20/08…$15.00
https://doi.org/10.1145/3370748.3406553

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 [4]. Through GreenMM framework, we have explored undervolting to save energy for GPUs for a Matrix Multiplication (MM) algorithm [5]. GreenMM saves the energy through undervolting beyond the $V_{safeMin}$ and employing algorithm based fault tolerant (ABFT) technique to correct the errors [5]. $V_{safeMin}$ 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 [6]. On the other hand, checkpoint and recovery (CR) is a general technique that can be applied to both regular and irregular applications. In this paper, 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, similar to undervolti- III describes checkpoint and recovery technique. Our evaluationng, overclocking beyond $f_{safeMax}$ may raise errors due to having less time for charging and discharging transistors. In this paper, we experimentally determine the safe values for the maximum safe frequency for different applications, similar to the undervolting experiments done earlier [4, 5].

We develop an undervolting and overclocking model in this paper 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 [7, 8]. 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 [9–12]. CheCuda [10] is built on top of checkpoint and recovery library called BLCR [11] to store the system state. Since the BLCR does not support CUDA contexts, before checkpointing, it stores and destroys

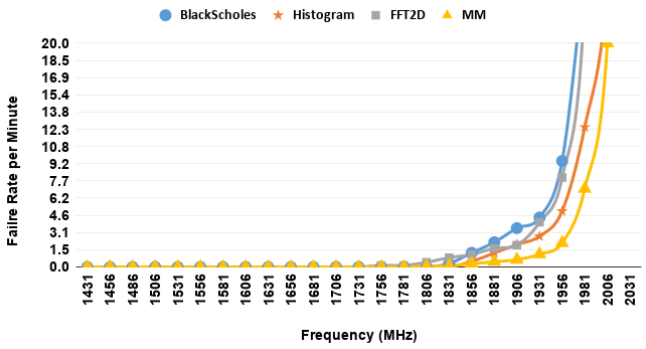

Figure 1: Failure rate of different Rodinia benchmarks and cuBLAS-MM from cuBLAS library w.r.t overclocking

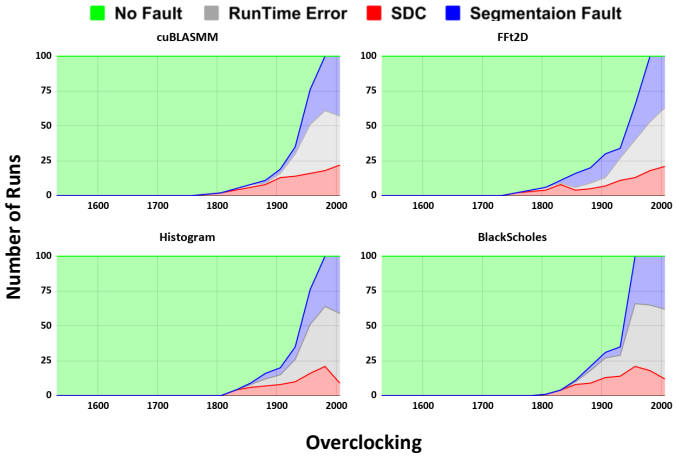

Figure 2: Fault distribution w.r.t. overclocking

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[12]. CheCL is another checkpoint and recovery technique that follows CheCuda but it is designed for OpenCL-based applications[13]. 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 [9]. 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 .

This paper 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.

The rest of the paper is organized as follows. Section II describes overclocking, undervolting and corresponding fault model. Section III describes checkpoint and recovery technique. Our evaluation results are discussed in section IV. Finally, conclusion and final remarks are provided in Section V.

2 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 [14]. In this section, we develop a realistic model for error probability in different applications at a given frequency.

The probability of failure is given by,

$$P_f = \frac{Number\ of\ failures}{Number\ of\ application\ runs} \tag{1}$$

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.

$$R(t) = 1 - P_f(t) \tag{2}$$

The failure rate is obtained using Weibull lifetime reliability model, a well-accepted model for transient and permanent soft errors as in equation 3 [15].

$$R(t) = e^{-\lambda t} \tag{3}$$

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 [16] 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 1, 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 [4].

Applications have different failure rates due to different activity patterns they experience during the execution time [4]. It is observed that increasing the frequency or lowering the voltage results in less timing margin and higher error probability [4]. 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 2 shows behavior of different 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 2, 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 [5]. As shown in Figure 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

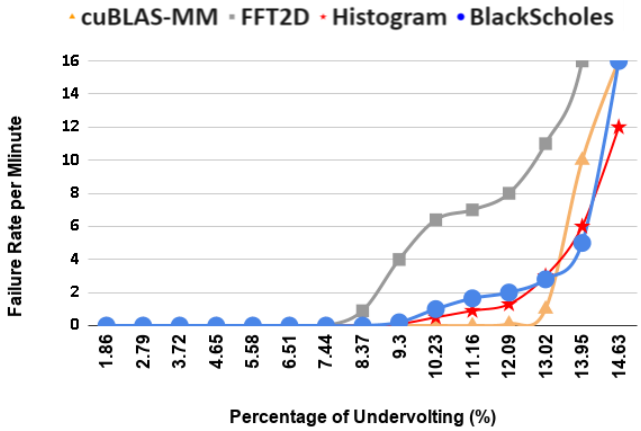

Figure 3: Failure rate w.r.t undervolting

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 discussed in section 3.

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:

2.0.1 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 2.

2.0.2 Phase 2: Estimate number of faults. The number of faults in an application can be obtained by multiplying λ with the execution time, as shown in equation 5. Failure rate remains same irrespective of the input data size for a given application as in equation 3. Hence, we estimate the execution time of MM for a given matrix size on a specific GPU through a simple profiling.

$$T = \alpha * ax^3\ b \tag{4}$$

$$F = \lambda * T \tag{5}$$

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 4, where a and b are architecture-specific constants [17][18]. 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 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.

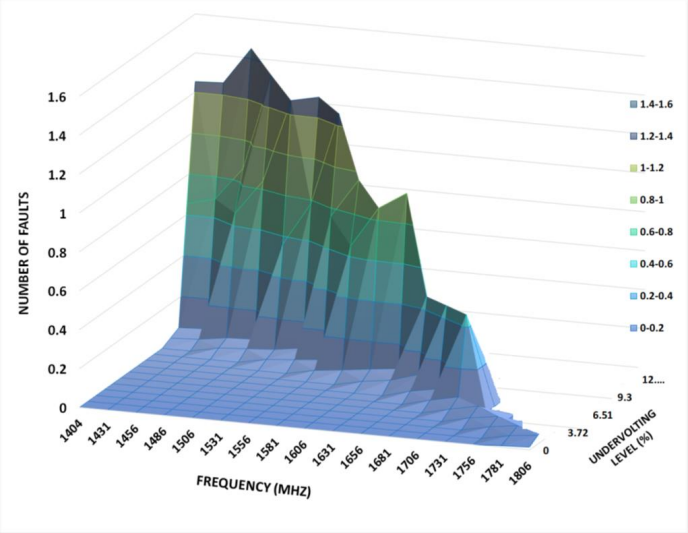

Figure 4: Number of faults in regard to overclocking and undervolting for matrix with size of 10K

3 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[11][10][12]. 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 [9] 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 1, 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.

Algorithm 1 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
```

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.

3.1 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 1, 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 [19]. To solve synchronization problem between TBs,

Figure 5: Thread block level synchronization using array of global variables

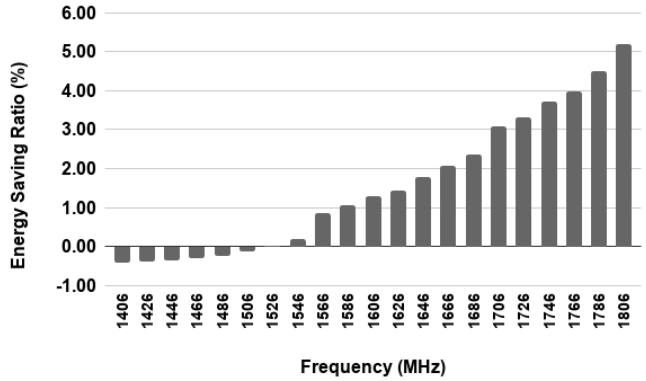

Figure 6: Energy saving ratio in regard to overclocking in presence of CR algorithm

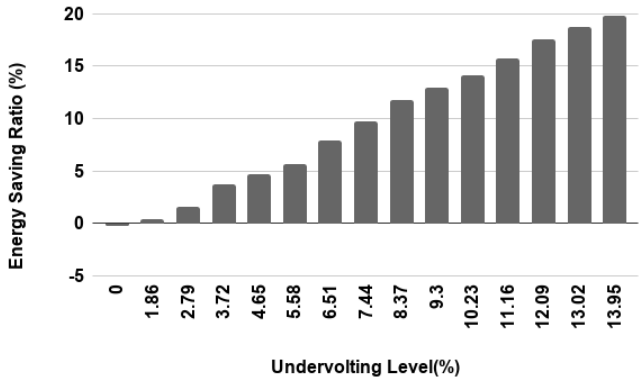

Figure 7: Energy saving ratio in regard to undervolting in presence of CR algorithm

we propose an algorithm which synchronizes TBs before checkpointing. As shown in Figure 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
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 1, 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 [5], we use checksum data redundancy for detection phase [20]. To compute C=A*B. Algorithm 2 describes pseudo-code for detection phase. A^c and B^r are encoded input matrices and C^f is a full checksum matrix [21]. 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.

4 EVALUATION

4.1 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 [22]. 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 [23][24]. For instance, MM constitutes of more than 90% of the computation cost in HPL [23]. Our proposed method can easily be integrated into these applications to save considerable amount of energy. Similar approach can be used while implementing

Algorithm 2 The pseudo-code for error detection phase

```
Initialize(A^c and B^r)
C^f = A^c × B^r
Recompute C^f from C
if (C^f doesn't maintain checksum relationship) then
    error_detected();
else
    no_error_detected();
end if
```

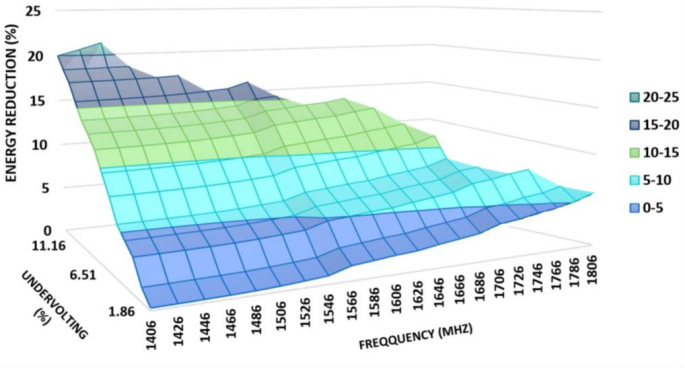

Figure 8: Energy reduction for given frequencies and voltages

checkpoint and recovery (CR) for different applications in order to save energy due to fusion of overclocking and undervolting.

4.2 Results

Figure 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 6, at default frequency, there is negligible energy overhead of about 0.5%. However, as we continue overclocking, 5.3% energy reduction can be achieved in comparison with the baseline.

As shown in the Figure 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. 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 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.

5 CONCLUSION

In this paper, we proposed an energy efficient framework, SAOU, 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 matrix of size 10K in cuBLAS-MM, SAOU is able to save the energy consumption up to 22% through combined undervolting and overclocking without sacrificing the performance.

ACKNOWLEDGMENT

This work is supported by NSF grant 1513201. The authors would like to thank the anonymous reviewers for their invaluable comments and suggestions.

REFERENCES

[1] "Dvfs-aware application classification to improve gpgpus energy efficiency," *Parallel Computing*, 2018.
[2] X. Mei, L. S. Yung, K. Zhao, and X. Chu, "A measurement study of gpu dvfs on energy conservation," in *Proceedings of the Workshop on Power-Aware Computing and Systems*, ser. HotPower '13, 2013, pp. 10:1–10:5.
[3] 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)*, Jul. 2016, pp. 242–247.
[4] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi, "Safe limits on voltage reduction efficiency in gpus: A direct measurement approach," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 294–307. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830811
[5] H. Zamani, Y. Liu, D. Tripathy, L. Bhuyan, and others, "GreenMM: energy efficient GPU matrix multiplication through undervolting," *Proceedings of the ACM*, 2019.
[6] K.-H. Huang *et al.*, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.
[7] 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*, Nov. 2010, pp. 1–11.
[8] P. Wu and Z. 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*, ser. HPDC '14. Association for Computing Machinery, Jun. 2014, pp. 49–60.
[9] B. Pourghassemi and A. Chandramowlishwaran, "cudaCR: An In-Kernel Application-Level Checkpoint/Restart scheme for CUDA-Enabled GPUs," *CLUSTER 2017*.
[10] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "CheCUDA: A Checkpoint/Restart tool for CUDA applications," *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2009.
[11] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux clusters," *J. Phys. Conf. Ser.*, vol. 46, no. 1, p. 067, Sep. 2006.
[12] N. A. Nvcr, "A transparent checkpoint-restart library for nvidia cuda," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW*, 2011.
[13] 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*, May 2011, pp. 864–876.
[14] 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)*.
[15] D. P. Murthy, M. Xie, and R. Jiang, *Weibull models*. John Wiley & Sons, 2004, vol. 505.
[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.
[17] S. S. Skiena, *The algorithm design manual: Text*. Springer Science & Business Media, 1998, vol. 1.
[18] R. L. Rivest and C. E. Leiserson, *Introduction to algorithms*. McGraw-Hill, Inc., 1990.
[19] A. Abdolrashidi, D. Tripathy, M. E. Belviranli, L. N. Bhuyan, and D. Wong, "Wireframe: Supporting data-dependent parallelism through dependency graph execution in gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 600–611.
[20] K.-H. Huang *et al.*, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.
[21] Z. Chen, "Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments," in *Parallel and Distributed Processing, 2008.*
[22] M. Afterburner, "http://goo.gl/fs2pti."
[23] 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*, Nov 2010, pp. 243–248.
[24] S. Blackford. (1997) ScaLAPACK users' guide.