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

HADI ZAMANI and LAXMI BHUYAN, University of California, Riverside, USA
JIEYANG CHEN, Oak Ridge National Laboratory, USA
ZIZHONG CHEN, University of California, Riverside, USA

The current trend of performance growth in HPC systems is accompanied by a massive increase in energy consumption. In this article, 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. Our aim is to apply DVFS to this application by leveraging slacks intelligently on both CPUs and multiple GPUs. To predict the slack times, accurate performance models are developed separately for both CPUs and GPUs based on the algorithmic knowledge and manufacturer's specifications. Since DVFS does not reduce static energy consumption, we also develop undervolting techniques for both CPUs and GPUs. 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, GPU, and total energy about 59%, 21%, and 31%, respectively, while delivering similar performance to the state-of-the-art linear algebra MAGMA library.

CCS Concepts: • **Computing methodologies** → *Massively parallel algorithms;*

Additional Key Words and Phrases: Parallel computation, heterogeneous multi GPUs systems, energy efficiency, high performance applications

ACM Reference format:
Hadi Zamani, Laxmi Bhuyan, Jieyang Chen, and Zizhong Chen. 2023. GreenMD: Energy-efficient Matrix Decomposition on Heterogeneous Multi-GPU Systems. *ACM Trans. Parallel Comput.* 10, 2, Article 12 (June 2023), 23 pages.
https://doi.org/10.1145/3583590

1 INTRODUCTION

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 demands on the GPUs, on the other hand, have led their designs to come with large power consumption [23]. Because existing libraries are mainly concerned with performance, they do not make efficient use of heterogeneous computing systems, resulting

This work is partly supported by NSF Grant 1907401.
Authors' addresses: H. Zamani, L. Bhuyan, and Z. Chen, University of California, Riverside, USA; emails: hzama001@ucr.edu, bhuyan@cs.ucr.edu, chen@cs.ucr.edu; J. Chen, Oak Ridge National Laboratory, USA; email: chenj3@ornl.gov.

2329-4949/2023/06-ART12 $15.00
https://doi.org/10.1145/3583590

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 [21]. It is included in several popular linear algebra libraries, such as **Linear Algebra Package (LAPACK)** [4] and Linpack [26] 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 GPUs, it divides the workload between CPUs and GPUs. CPU handles the panel factorization, which is serial in nature. The GPUs update the trailing matrix because it involves large computation that can be highly parallelized. During the execution, either the CPU or GPUs could be on non-critical paths that can experience idle time, or slack. These slacks can be exploited for energy savings by exploring power-aware techniques such as **Dynamic Voltage and Frequency Scaling (DVFS)**.

Over the last few years, significant efforts have been made to apply various techniques, such as DVFS [12, 22] and undervolting [34]. DVFS approaches have been employed to save energy during underutilized execution phases of the execution, called slack, on CPUs [2, 20, 28, 29] and GPUs [7, 13, 33, 34]. The amount of slack on different components can be estimated using algorithmic knowledge 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. Prior research on LU factorization utilized the slack using a simple performance model in a single GPU environment [7]. 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 a 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 do not target the static power consumption, which is becoming predominant in today's technology. The 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% [9, 11]. As a result, they are inherently conservative and not energy efficient [5]. Leng et al. investigate the voltage guardband of the GPUs and observe that there is about a 20% voltage guardband on different GPU architectures [16]. There are many efforts to operate hardware at sub-nominal voltage levels on the CPUs [9, 11, 25, 37] and GPUs [16, 18, 32, 34]. 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 CPUs and GPUs, we apply the undervolting during the execution of the LU factorization. It is shown that undervolting reduces energy consumption by 16% beyond DVFS.

This article presents GreenMD, a framework that improves the energy efficiency of heterogeneous multi-GPU systems while maintaining the reliability and performance 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 them 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, this article makes the following contributions:

- Using algorithmic knowledge and hardware configuration, we develop a more accurate performance model for 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 and single and multiple GPUs based on the accurate performance models.
- We implement undervolting in CPU and single and multiple GPUs based on the empirical observations.
- We evaluate the performance, energy savings, and reliability through real implementation and achieve 31% total energy savings for double-precision LU factorization with a matrix of size 18K*18K.

The rest of article is organized as follows. The background and motivation with profiling results are discussed in Section 2. A slack predictor is developed in Section 3 based on the CPU/GPU performance models. Proposed methodology, including DVFS-based slack reclamation and undervolting, is presented in Sections 4 and 5, respectively. Experimental evaluations and results are provided in Section 6. Finally, related works and the conclusion are discussed in Sections 7 and 8.

2 BACKGROUND AND MOTIVATION

2.1 LU Factorization Overview

Figure 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 kernel "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 the 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.

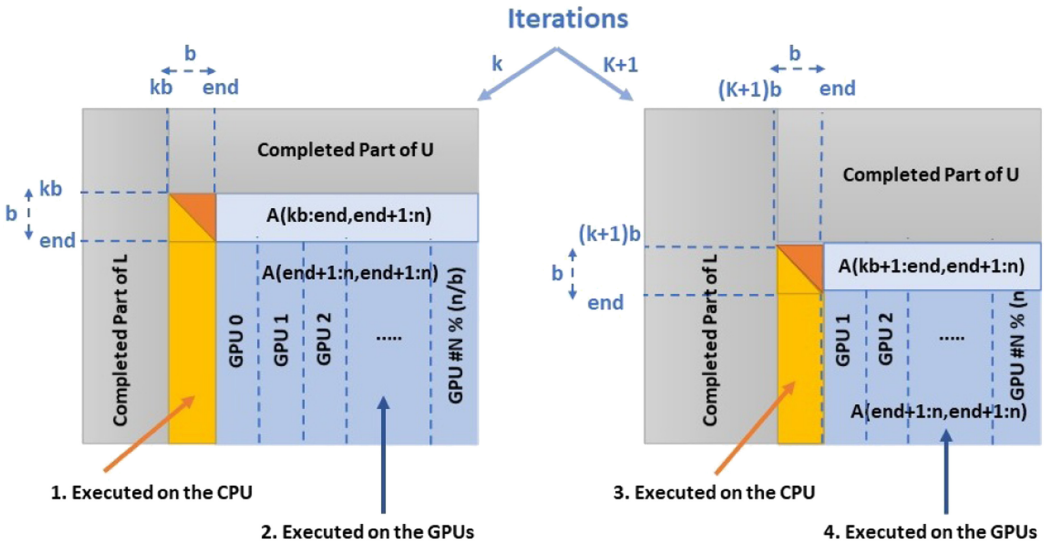

Fig. 1. Overview of the blocked LU factorization.

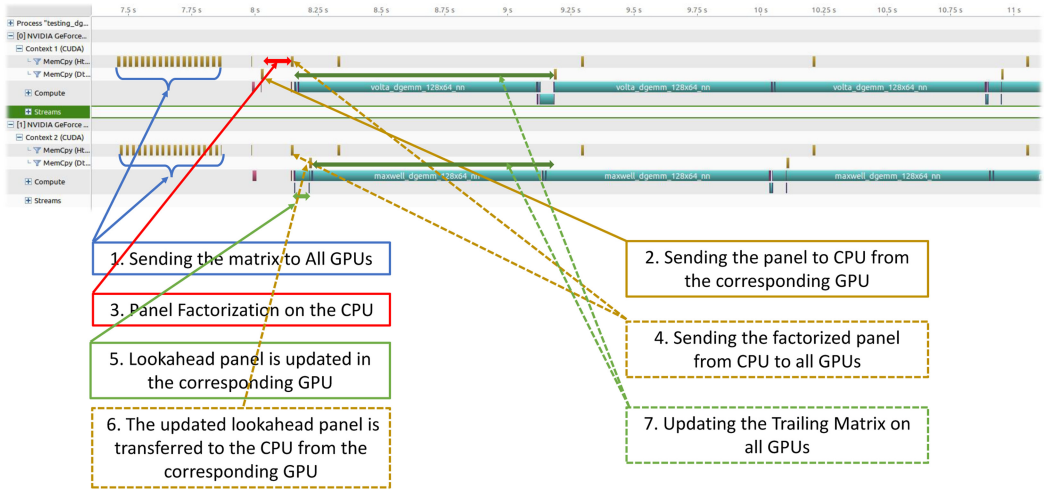

Fig. 2. Trace of the execution of multi-GPU version of double-precision LU factorization for a matrix of size 18K.

2.2 Profiling Observations

Figure 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 GPU and CPU timelines at the same time, we profile the LU factorization on the CPUs and GPUs separately. For GPU, we used the 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 factorization algorithms, distribute the workload between CPU and GPU at each iteration. Figure 2 shows the beginning trace of LU

factorization on a 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 the trailing matrix update, which is performed on the GPU. However, the amount of slack on the GPUs increases as LU factorization approaches the end of execution. This is because the size of the trailing matrix update on two GPUs is decreasing over time and takes less and less time to execute on the GPUs 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.

- (1) At the beginning the whole matrix is divided and copied 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 the trailing matrix. After the look-ahead panel update, the GPU continues to update the rest of trailing matrix along with the other GPUs.
- (3) Panel factorization is done on CPU.
- (4) The factorized panel is sent to all GPUs for the update phase for the next iteration.
- (5) In the next iteration, the 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 is distributed across multiple GPUs, is updated on the corresponding GPUs.

3 SLACK PREDICTOR

In each iteration, either the CPU or the GPU is in the critical path. The 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. Recall that the earlier paper on DVFS for LU factorization used profiling with a simple performance model [7]. However, our techniques do not involve any benchmarking and the performance can be directly computed.

3.1 Performance Model of LU Factorization on the CPU

By dividing the 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 (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. The MAGMA library is highly optimized and is getting close to the maximum peak

performance.

$$Compute_{capability} = p_{cpu} \times \frac{freq}{Max_{frequency}} \tag{1}$$

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 [10]

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

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 3:

$$M_k = \left(\frac{m}{b} - k\right) \times b. \tag{3}$$

Hence, the amount of operations, W_{panel}, will be estimated through Equation (4):

$$W_{panel_k} = \left((\frac{m}{b} - k) \times b) - \frac{b}{3}\right) \times b^2. \tag{4}$$

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 Equation (5) for a given iteration k:

$$T_{CPU_k} = \frac{(((\frac{m}{b}) - k) \times b) - \frac{b}{3}) \times b^2}{p_{cpu} \times \frac{freq}{Max_{frequency}}}. \tag{5}$$

But with a 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 [10]. As a result, we can estimate the panel factorization execution time of the multi-threaded CPU at the kth iteration as 6:

$$T'_{CPU_k} = \frac{(((\frac{m}{b}) - k) \times b) - \frac{b}{3}) \times b^2 + b^3 \times log\ (N_{cpu-threads})}{p_{cpu} \times \frac{freq}{Max_{frequency}} \times N_{cpu-threads}}. \tag{6}$$

Since we would run the CPU at different frequencies for DVFS, we compared the estimated values through Equation (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 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 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 demonstrate a 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 time to charge and discharge the transistors. So timing errors are one of the main reasons for faults/errors. However, in 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 [30]. Besides, different frequencies can change the temperature of the processor, which could lead to different amounts of error rate. In other words, the error rate is dependent on frequency, execution time, and temperature.

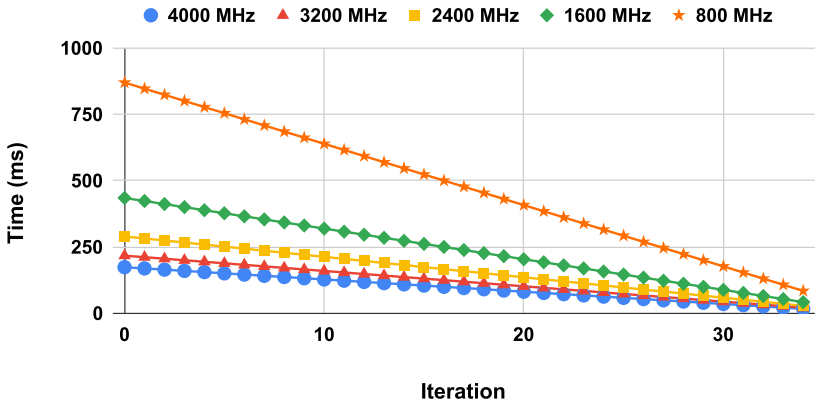

Fig. 3. The estimated time of CPU w.r.t various frequencies.

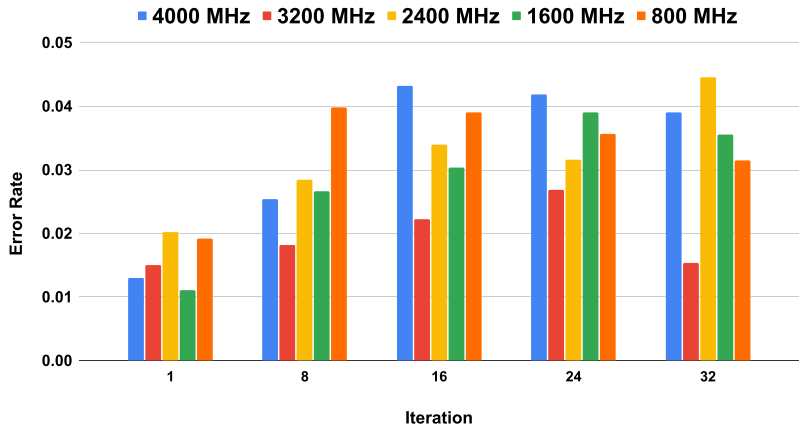

Fig. 4. Error rate in performance model on the underlying CPU.

3.2 Performance Model of the GPU Kernel

Similar to the CPU performance model provided in Equation (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 operations by the GPU computation rate.

In each iteration of the LU factorization, the size of the 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

$$Row_{width_k} = \left(\frac{m}{b} - k\right) \times b \tag{7}$$

$$Column_{width_k} = \left(\frac{n}{b} - k\right) \times b. \tag{8}$$

Therefore, the total number of operations (workload) required for a trailing matrix update at a given iteration k can be written as [10]

$$Worklaod_{GPU_k} = \left(\left(\frac{m}{b} - k\right) \times b\right) * \left(\left(\frac{n}{b} - k\right) \times b\right)^2. \tag{9}$$

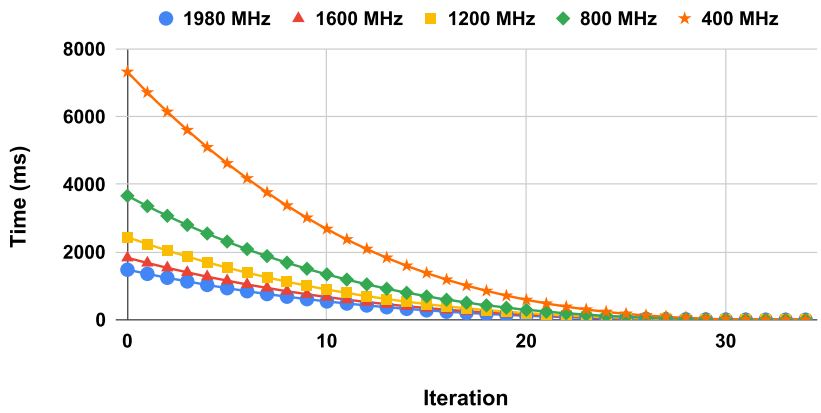

Fig. 5. The estimated time of single GPU w.r.t various frequencies using Equation (10).

Therefore, extending Equation (5) to the GPU, Equation (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 the ith GPU at a given iteration k.
- $Workload_{GPU_{i@k}}$ is the total workload of the ith GPU at a given iteration k.

At iteration k, Equation (9) is used to compute the total number of floating-point operations required. If multiple GPUs are employed, the computation load is distributed evenly among them, and any extra panel assigned to certain GPUs due to the uneven number of panels is assigned a workload calculated using Equation (9).

Which n is the matrix size,

$$T_{GPU_{i@k}} = \frac{Workload_{GPU_{i@k}}}{P_{gpu} \times \frac{freq}{Max_{frequency}} \times N_c}. \tag{10}$$

Figure 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 1,408 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 so forth into account. Figure 6 shows the error rate of the execution time on a single GPU in the presence of various frequencies. The results show that the average error rate is about 5%. However, the error rate is increased toward the last iterations. This is because, with the update matrix getting smaller and smaller, the GPU is not fully utilized. The execution time is getting smaller, which results in a larger error rate. Using Equation (10), we also estimate the execution time of two GPUs. Figure 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 1,980 MHz. It shows that the execution time of the GPU is inversely proportional to the number of GPUs at different iterations. In the case of two GPUs, the execution time of the 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%.

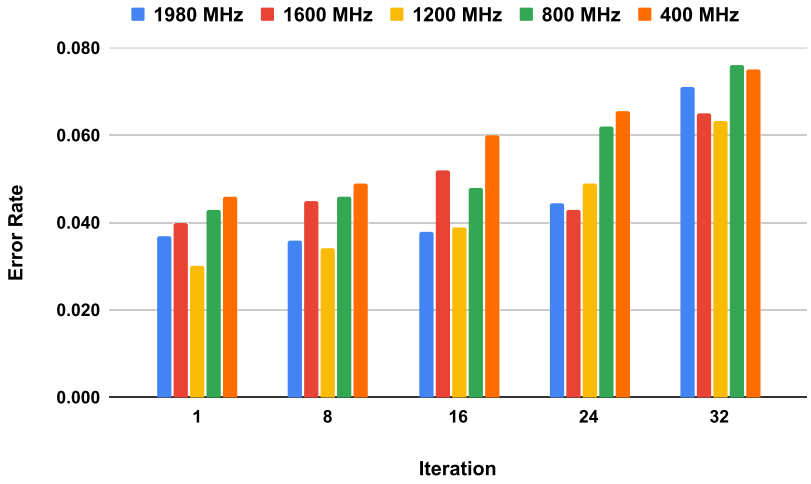

Fig. 6. Error rate of GPU performance model provided in Equation (10).

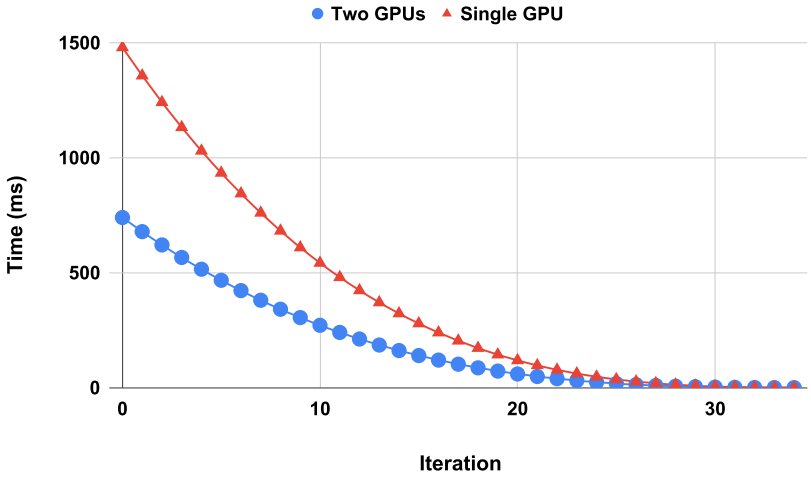

Fig. 7. Estimated GPU time with single and two GPUs at default frequency of 1,980 MHz.

3.3 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 the LogGP model [8] 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 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. 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

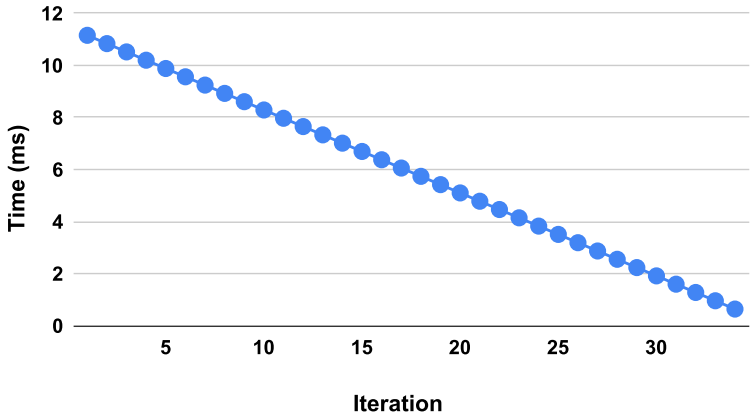

Fig. 8. The estimated copy time of double-precision LU factorization for a matrix of size 18K.

on a different stream. As a result, the time required to transfer k bytes utilizing n streams will be determined using Equation (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 Equation (12). Here m is the row width of matrix; b is the block size or the panel width, which remains the same over the iterations.

$$T_{copy_k} = L + O + K/B + (n-1) \times g \tag{11}$$

$$T_{copy_k} = L + O + ((m - k \times b) \times b \times 8)/B + (n-1) \times g \tag{12}$$

Even though the newer GPUs are equipped with dual copy engines and they might share resources, which 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 8 shows the estimated time between the 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 a single GPU (NVIDIA GeForceGTX 1660 SUPER) and CPU (Intel(R) Core(TM) i7-6700k). Figure 9 shows that the error rate is very negligible in the iterations when the PCIe bus is fully utilized. However, the error rate 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 the 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 iterations, 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 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 the 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 10. Slack occurs only when the CPU waits

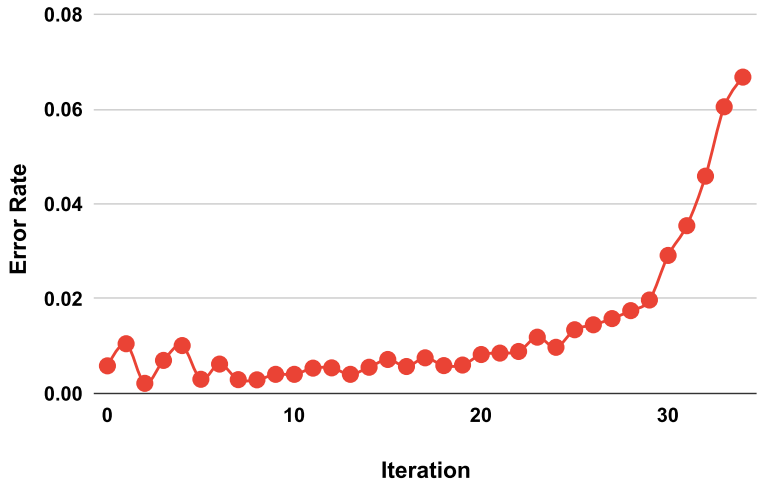

Fig. 9. The error rate of the PCIe model for a matrix of size 18K.

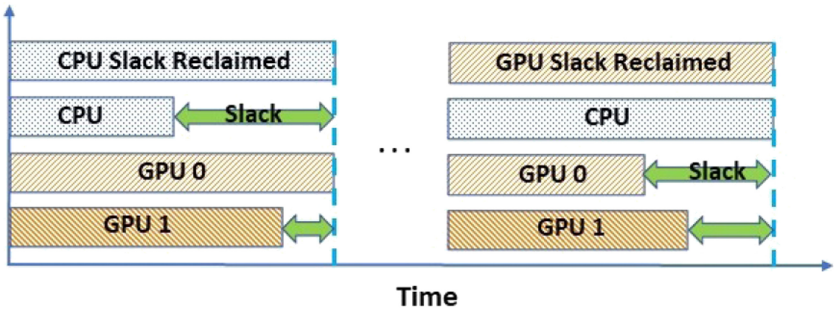

Fig. 10. An overview of slack reclamation.

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 1 provides the overview of the DVFS controller that estimates the panel factorization time on the CPU, matrix update time on GPUs, and data copy time between the CPU and GPU using the equations provided in Section 3. 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 the CPU and GPU are given by Equations (13) and (14). These values are extracted according to Equations (6), (10), and (12), respectively. In each iteration the copy time is the summation of both H2D and D2H transfers that happens between CPU and GPUs.

$$slack_{CPU_k} = Max(T_{GPU_{i@k}}) - T_k^{CPU} - T_k^{Copy}; 0 \leq i < \frac{N}{nb} \tag{13}$$

$$slack_{GPU_{i@k}} = T_k^{CPU} + T_k^{Copy} - T_{GPU_{i@k}} \quad ; \quad 0 \leq i < \frac{N}{nb} \tag{14}$$

ALGORITHM 1: 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
```

If Equation (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 (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 the **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 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 3 and 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 (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:

$$f_{opt} = f_{default} \times \frac{\max(T_{CPU}, T_{GPU}) - slack}{\max(T_{CPU}, T_{GPU})}. \tag{15}$$

The frequency of the CPU/GPUs is adjusted using the pseudo code provided in Algorithm 2. 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.

4.1 Scalability of the Proposed Method

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. In

ALGORITHM 2: 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
```

this article, 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 a heterogeneous system with GPUs. According to the empirical results provided in the article, at earlier iterations, since the panel widths are larger, the CPU takes more time than the GPU to factorize the panel and we still experience slack on the GPU unless more CPUs and potentially GPUs are employed to help the CPU to factorize the panel. And if the number of GPUs increases, in later iterations, 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.

5 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 DVFS on critical paths degrades the application's overall performance for compute-intensive workloads [3]. 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 [36]. The guard-bands additionally take into consideration errors caused by the load line, aging effects, noise, and calibration error [27]. Because these errors do not occur frequently, significant energy savings can be achieved by lowering guard-band voltage to a much lower supply voltage [19]. 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 [16]. We undervolt to the safe minimum voltage $V_{safeMin}$ without introducing any fault into the system. The 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 [34], 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 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

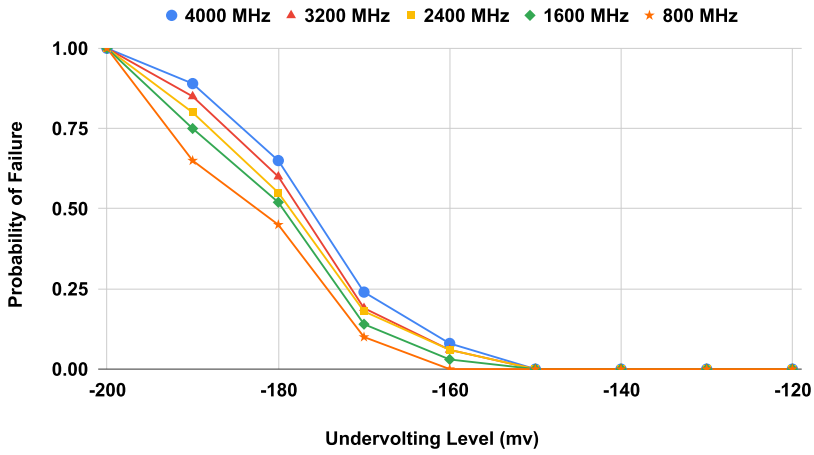

Fig. 11. CPU's probability of failure w.r.t undervolting.

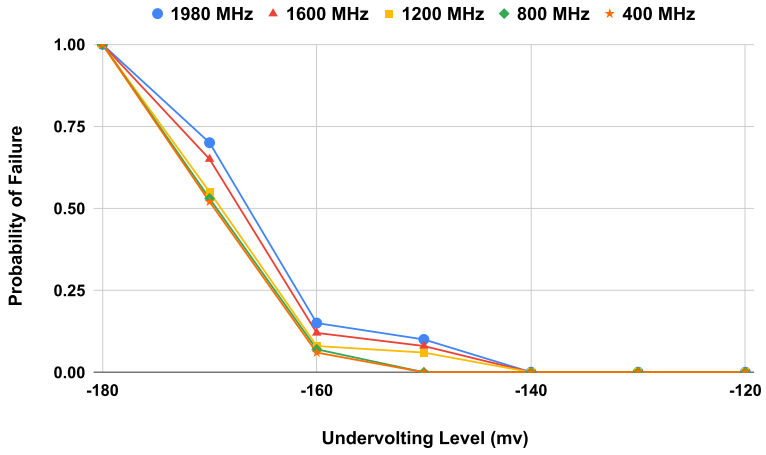

Fig. 12. GPU's probability of failure w.r.t undervolting.

of undervolting could be different for different frequencies as observed in Figures 11 and 12. The probability of failure for the CPU is shown in Figure 11. 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.

Similar to CPU, we extract the $VsafeMin$ for the underlying GPU as well. For a maximum frequency of 1,980 MHz, the GPU default voltage is 1.045 V. Similarly, the underlying GPU (GTX 1660 super) is undervolted in 10 mv steps. For each level of undervolting, the application is executed 100 times and the corresponding output is compared to the golden output to verify correctness. Similar to the CPU, the probability of GPU failure as well as $V_{safeMin}$ is extracted for different frequencies, as depicted in Figure 12. According to the empirical results, we are able to undervolt the GPU about 140 to 150 mv for different frequencies. Hence, in our method, we undervolt the CPU and GPUs 150 mv and 140 mv, respectively. Prior to the application's execution, the GPUs are undervolted. This is due to the lack of a runtime API for undervolting the GPU during execution. Undervolting the GPU is also done in a similar way as in [34], and as explained below in Section 6.

Table 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	4,200 MHz	1,980 MHz
Memory	16 GB	6 GB
Cache	L1 (128 KB) L2 (1 MB) L3 (8 MB)	L1 (64 KB per SM) L2 (1,536 KB)
OS	Ubuntu 20.04	

Table 2. Power Management and Undervolting APIs

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

6 EVALUATION

6.1 Implementation Details

All experiments are preformed on a heterogeneous system with 8-core Intel CPU and Two homogeneous GEFORCE GTX 1660 SUPER, whose architectural specifications are listed in Table 1. We were able to evaluate the results for up to a 18K matrix size due to the 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 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 the 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 the GPU profiling phase and extracting the safe minimum voltage of the GPU, MSI After Burner [1] was used. However, MSI After Burner is not supported on the Linux operating system. So, to reduce the voltage of the GPU, we used a similar approach to that employed in [34]. Since there is no direct API to reduce the voltage, we reduce the voltage of the GPU by lowering the GPU's target power limit at a fixed frequency. To undervolt the GPU, several APIs from the **NVIDIA Management Library (NVML)** are used as listed in Table 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.

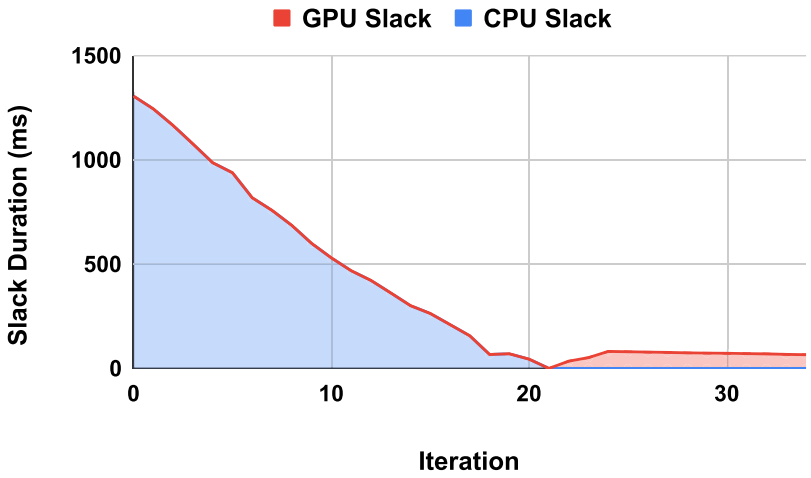

Fig. 13. The amount of CPU and GPU slack for double-precision LU factorization with one GPU and matrix of size 18K.

6.2 Results

We executed the LU factorization in presence with one and two GPUs. In the case of a single GPU, the amount of slack on the CPU and GPU is shown in Figure 13 for a matrix size of 18K x 18K. The slacks for CPU and single GPU are observed in iterations 0 to 21 and 22 to 34, respectively. This is because, even though the GPU is equipped with a huge number of computing cores, it has a 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 reclaim the slack allowing both the CPU and GPU to complete their tasks at the same time at a given iteration. If the amount of adjusted frequency is less than 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, we set 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 the maximum level of safe undervolting. The amount of energy consumed at the default scenario (baseline) and proposed approach w.r.t iteration is shown in Figure 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 15 shows the GPU's energy consumption 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 lead to more energy reduction. Figure 15 shows that, on average, we save energy about 18% on the single GPU.

We have also extracted the results for a 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 16 shows the amount of slack for different iterations for both the CPU and GPUs. Compared with a single GPU case in Figure 13, the

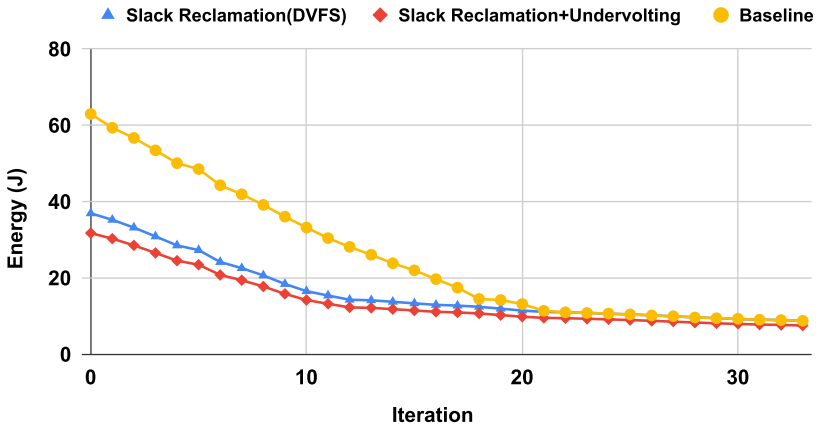

Fig. 14. CPU energy improvement of double-precision LU factorization with single GPU for a matrix of size 18K.

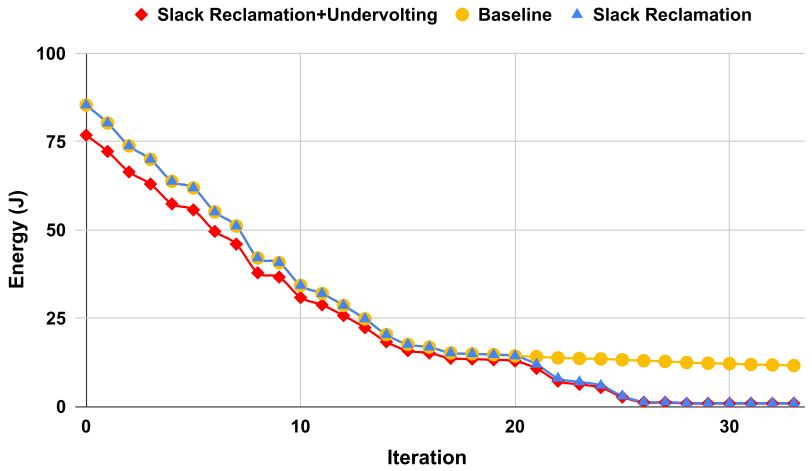

Fig. 15. GPU energy improvement of double-precision LU factorization in the presence of single GPUs for a matrix of size 18K.

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 17 shows the amount of CPU energy consumption in the presence of DVFS and undervolting. Compared to single GPU results, illustrated in Figure 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, respectively.

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

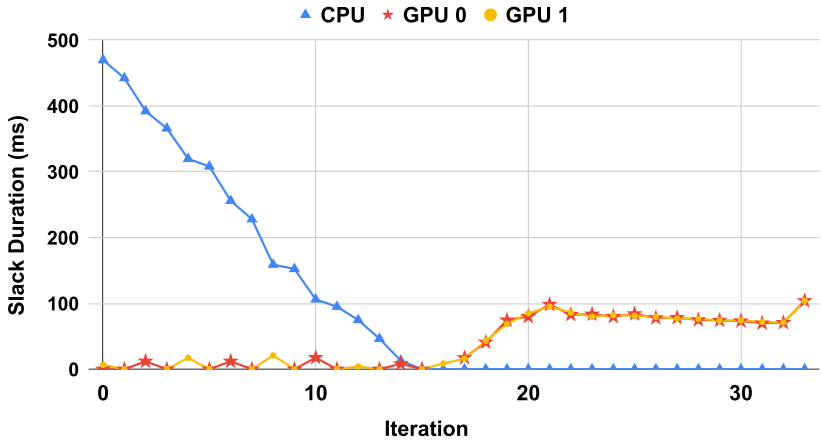

Fig. 16. The amount of slack for double-precision LU factorization in the presence of two GPUs for a matrix of size 18K.

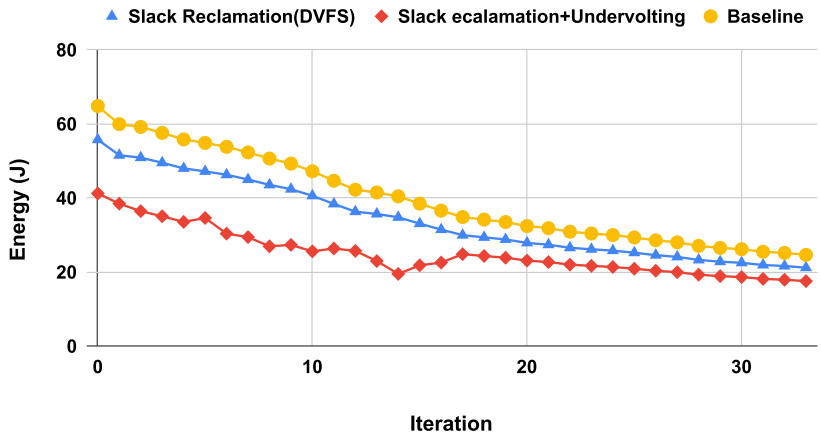

Fig. 17. The CPU energy improvement of double-precision LU factorization with two GPUs for a matrix of size 18K.

method is shown in Figure 18. 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 18, on average, we save the total energy consumption of the GPUs about 21%. In comparison to the results illustrated in Figure 15 for a single GPU, the small improvement in the energy savings comes mainly from the undervolting part. This is because, even though the trailing matrix execution time is reduced by half, the total power consumption doubles due to the use of two GPUs keeping the energy consumption almost the same. Figures 19 and 20 show the total energy consumption of LU factorization for a matrix of size 18K with single and two GPUs, respectively. As shown in Figure 20, 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 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.

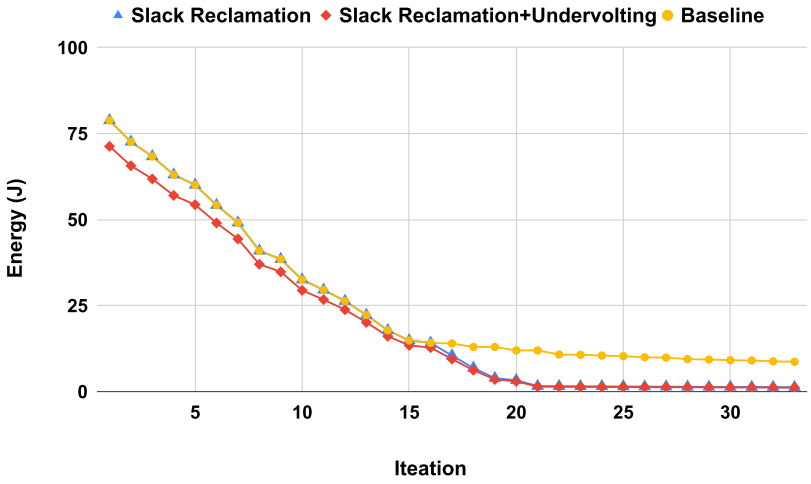

Fig. 18. GPU energy improvement of double-precision LU factorization in the presence of two GPUs for a matrix of size 18K.

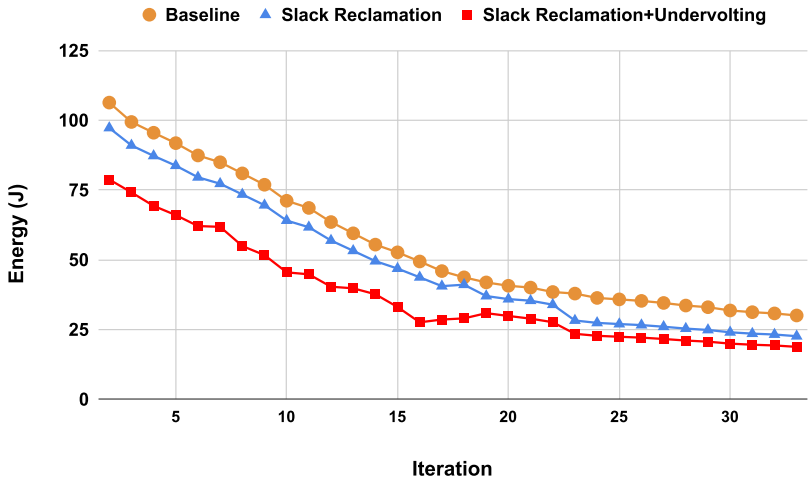

Fig. 19. Total CPU and GPU energy improvement of double-precision LU factorization with single GPUs for a matrix of size 18K.

7 RELATED WORKS

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. 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 and (2) works that introduce the runtime models to predict the GPU application performance and/or power consumption.

Jiao et al. [15] 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

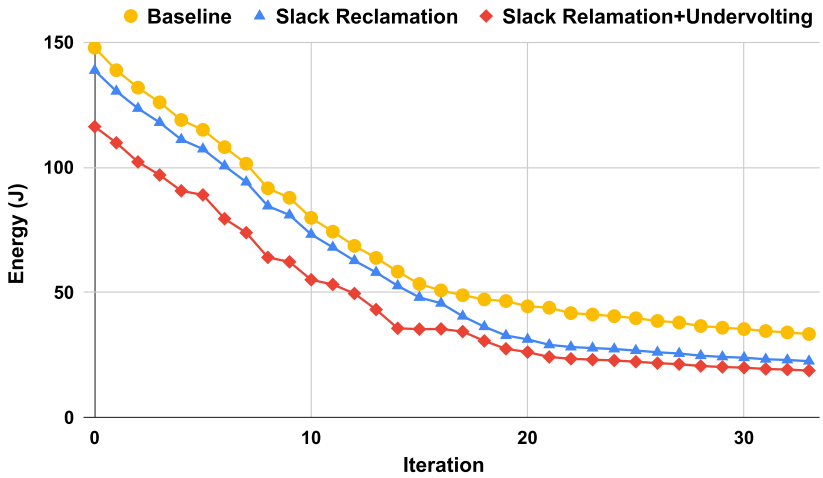

Fig. 20. Total CPU and GPU energy improvement of double-precision LU factorization with two GPUs for a matrix of size 18K.

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 [14, 24, 31], seeking to properly reflect the execution characteristics of GPGPU applications. In some research, the performance models are based on profiling as well as the algorithmic knowledge of the application. For instance, in [7], 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 [6], 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 [17]. Also, in [35], 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% to 15% error rate. However, these methods are frequently product specific and difficult to port to other architectures.

In order to leverage the **Dynamic Voltage and Frequency Scaling (DVFS)** technique, we present a performance model that incurs a maximum performance overhead of 5%. Our performance model is derived from the general GPU performance model and can be adapted to suit diverse compute-intensive applications with minor modifications to the introduced performance models.

8 CONCLUSION

In this article, we present GreenMD, an energy-efficient framework to improve the energy consumption of LU factorization on a heterogeneous mutli-GPU system. We profiled the execution

Table 3. Energy Improvement of CPU and 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%

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 the 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 the CPU and GPUs independently based on the minimum threshold voltages to avoid error.

Since GreenMD employs the same programming interface as MAGMA's LU factorization, it is transparent to programs that use the LU factorization and does not require users to change the application's source code. GreenMD is portable and can be utilized with any GPU architecture. Knowing how the workload is distributed across each underlying architecture and an understanding of algorithms are both necessary for this. This means that GreenMD can be used with any GPU architecture by just changing a few model parameters that are particular to each GPU architecture. Also, the approach we propose can be used to improve a broad range of kernels. The performance model will be adjusted and the appropriate level of undervolting will be determined using the offline profiling phase and algorithm information.

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 as shown in Table 3 showed that CPU and GPU energy consumption improved by around 51% 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 59% and 21%, respectively. The total energy saved during the entire execution is 31%.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their invaluable comments and suggestions.

REFERENCES

[1] MSI Afterburner. [n. d.]. http://goo.gl/fs2pti.

[2] R. Begum, M. Hempstead, G. P. Srinivasa, and G. Challen. 2016. Algorithms for CPU and DRAM DVFS under inefficiency constraints. *IEEE 34th International Conference on Computer Design (ICCD'16)*, 161–168.

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

[4] Anthony M. Castaldo and R. Clint Whaley. 2010. Scaling LAPACK panel operations using parallel cache assignment. *ACM Sigplan Notices* 45, 5 (2010), 223–232.

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

[6] Jieyang Chen, Hongbo Li, Sihuan Li, Xin Liang, Panruo Wu, Dingwen Tao, Kaiming Ouyang, Yuanlai Liu, Kai Zhao, Qiang Guan, and Zizhong Chen. 2018. Fault tolerant one-sided matrix decompositions on heterogeneous systems with GPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'18)*. 854–865. https://doi.org/10.1109/SC.2018.00071

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

[8] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. 1993. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.

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

[10] S. Donfack, S. Tomov, and J. Dongarra. 2014. Dynamically balanced synchronization-avoiding LU factorization with multicore and GPUs. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*. 958–965.

[11] Dimitris Gizopoulos, George Papadimitriou, Athanasios Chatzidimitriou, Vijay Janapa Reddi, Behzad Salami, Osman S. Unsal, Adrian Cristal Kestelman, and Jingwen Leng. 2019. Modern hardware margins: CPUs, GPUs, FPGAs Recent System-Level Studies. *IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS'19)*. (2019), 129–134. DOI: 10.1109/IOLTS.2019.8854386

[12] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás. 2019. DVFS-aware application classification to improve GPGPUs energy efficiency. *Parallel Comput.* 83 (2019), 93–117.

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

[14] Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. 280–289.

[15] Y. Jiao, H. Lin, P. Balaji, and W. Feng. 2010. 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*. 221–228. https://doi.org/10.1109/GreenCom-CPSCom.2010.143

[16] Jingwen Leng, Alper Buyuktosunoglu, Ramon Bertran, Pradip Bose, and Vijay Janapa Reddi. 2015. Safe limits on voltage reduction efficiency in GPUs: A direct measurement approach. *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 294–307. https://doi.org/10.1145/2830772.2830811

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

[18] Jingwen Leng, Yazhou Zu, and Vijay Janapa Reddi. 2014. Energy efficiency benefits of reducing the voltage guardband on the Kepler GPU architecture. In *Workshop on Silicon Errors in Logic-System Effects (SELSE'14)*.

[19] J. Leng, Y. Zu, M. Rhu, M. S. Gupta, and V. J. Reddi. 2014. GPUVolt: Modeling and characterizing voltage noise in GPU architectures. In *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED'14)*. 141–146. https://doi.org/10.1145/2627369.2627605

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

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

[22] Xinxin Mei, Ling Sing Yung, Kaiyong Zhao, and Xiaowen Chu. 2013. A measurement study of GPU DVFS on energy conservation. In *Proceedings of the Workshop on Power-aware Computing and Systems*. 1–5.

[23] R. M. Miller. 2013. Exascale computing. https://www.datacenterknowledge.com/archives/2010/12/10/exascale-computing-gigawatts-of-power.

[24] Rajib Nath and Dean Tullsen. 2015. The CRISP performance model for dynamic voltage and frequency scaling in a GPGPU. In *Proceedings of the 48th International Symposium on Microarchitecture*. 281–293.

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

[26] Antoine Petitet. 2004. HPL-a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/.

[27] N. Rohbani, M. Ebrahimi, S. Miremadi, and M. B. Tahoori. 2017. Bias temperature instability mitigation via adaptive cache size management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 3 (March 2017), 1012–1022. https://doi.org/10.1109/TVLSI.2016.2606579

[28] Barry Rountree, David K. Lowenthal, Bronis R. De Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. 2009. Adagio: Making DVS practical for complex HPC applications. In *Proceedings of the 23rd International Conference on Supercomputing*. 460–469.

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

[30] Smruti R. Sarangi, Brian Greskamp, Radu Teodorescu, Jun Nakano, Abhishek Tiwari, and Josep Torrellas. 2008. VARIUS: A model of process variation and resulting timing errors for microarchitects. *IEEE Transactions on Semiconductor Manufacturing* 21, 1 (2008), 3–13. https://doi.org/10.1109/TSM.2007.913186

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

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

[33] Zhenheng Tang, Yuxin Wang, Qiang Wang, and Xiaowen Chu. 2019. The impact of GPU DVFS on the energy and performance of deep learning: An empirical study. In *Proceedings of the 10th ACM International Conference on Future Energy Systems (e-Energy'19)*. Association for Computing Machinery, New York, NY, 315–325.

[34] Hadi Zamani, Yuanlai Liu, Devashree Tripathy, Laxmi Bhuyan, and Zizhong Chen. 2019. GreenMM: Energy efficient GPU matrix multiplication through undervolting. In *Proceedings of the ACM International Conference on Supercomputing*. 308–318.

[35] Yao Zhang and John D. Owens. 2011. A quantitative performance analysis model for GPU architectures. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 382–393. https://doi.org/10.1109/HPCA.2011.5749745

[36] Yazhou Zu, Charles R. Lefurgy, Jingwen Leng, Matthew Halpern, Michael S. Floyd, and Vijay Janapa Reddi. 2015. Adaptive guardband scheduling to improve system-level efficiency of the POWER7+. *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 308–321. DOI: 10.1145/2830772.2830824

[37] Yazhou Zu, Charles R. Lefurgy, Jingwen Leng, Matthew Halpern, Michael S. Floyd, and Vijay Janapa Reddi. 2015. Adaptive guardband scheduling to improve system-level efficiency of the POWER7+. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. IEEE, 308–321.

Received 7 September 2022; revised 20 November 2022; accepted 24 December 2022