GreenLA: Green Linear Algebra Software for GPU-Accelerated Heterogeneous Computing

Jieyang Chen*§, Li Tan*†§, Panruo Wu*, Dingwen Tao*, Hongbo Li*,
Xin Liang*, Sihuan Li*, Rong Ge‡, Laxmi Bhuyan*, and Zizhong Chen*◇
*University of California, Riverside, ‡Clemson University
†Ultrascale Systems Research Center, Los Alamos National Laboratory
{jchen098, pwu011, dtao001, hli035, xlian007, sli049, bhuyan, chen}@cs.ucr.edu,
darkwhite29@gmail.com, rge@clemson.edu

***Abstract*—While many linear algebra libraries have been developed to optimize their performance, no linear algebra library considers their energy efficiency at the library design time. In this paper, we present `GreenLA` - an energy efficient linear algebra software package that leverages linear algebra algorithmic characteristics to maximize energy savings with negligible overhead. `GreenLA` is (1) *energy efficient*: it saves up to several times more energy than the best existing energy saving approaches that do not modify library source codes; (2) *high performance*: its performance is comparable to the highly optimized linear algebra library MAGMA; and (3) *transparent to applications*: with the same programming interface, existing MAGMA users do not need to modify their source codes to benefit from `GreenLA`. Experimental results demonstrate that `GreenLA` is able to save up to three times more energy than the best existing energy saving approaches while delivering similar performance compared to the state-of-the-art linear algebra library MAGMA.**

***Index Terms*—energy, performance, critical path, algorithmic slack prediction, DVFS, CPU, GPU, dense matrix factorizations**

I. Introduction

Scientific applications running nonstop on large-scale High Performance Computing (HPC) systems must efficiently use energy for execution. Comprising millions of components, today's HPC systems already consume megawatts of power; to meet an insatiate demand for performance from mission-critical applications, future systems will consist of even more components and consume more power [3]. Efficient use of energy by scientific applications not only reduces energy costs but also allows greater performance under a given power budget and improves system reliability.

Scientific applications can greatly benefit from heterogeneous computing technologies for energy efficiency. Heterogeneous computing systems are accelerated with many-core processing units. Such accelerators, including GPUs and coprocessors, consist of hundreds to thousands of low-power cores, delivering highly power efficient computing. By offloading massively parallel and compute intensive kernels to accelerators, scientific applications can achieve better performance while consuming less energy. Today, five of the top ten [6] computing systems are accelerated with either Intel Xeon Phi coprocessors or NVIDIA GPUs. Moreover, libraries and packages commonly used by scientific applications begin to have heterogeneous computing versions, such as MAGMA [4] numerical linear algebra library.

Improving the energy efficiency of commonly used libraries is an effective approach to energy efficient scientific computing. Unfortunately, existing libraries are focused on performance, inconsiderate of energy savings opportunities that do not adversely impact performance. For example, MAGMA decomposes a program to tasks and schedules sequential and less parallelizable tasks on CPU and larger more parallelizable ones on GPU. Consequently, MAGMA achieves better performance than its counterpart libraries for homogeneous CPU computing. Yet, inherent in the DAG-based task scheduling in MAGMA, processing units scheduled with tasks on non-critical paths unavoidably experience idle time, i.e., slack. The slack can be further exploited for energy savings by leveraging hardware power-aware techniques including Dynamic Voltage and Frequency Scaling (DVFS). DVFS has been used to save energy on CPU by scaling down CPU speed during underused execution phases [16] [30] [29] [31], and now is also available on memory [13] [14] and GPU cards [25] [5].

Numerical linear algebra libraries are used in a wide spectrum of high performance scientific applications. These libraries solve systems of linear equations, linear least square problems, and eigenvalue/eigenvector problems. Among numerical linear algebra operations, dense matrix factorizations can sometimes take a large portion of execution time or even dominate the whole scientific application execution.

This paper presents `GreenLA` - an energy efficient linear algebra software package for heterogeneous scientific computing on GPU-accelerated systems. At the initial stage of the project, we analyzed highly optimized dense matrix factorization algorithms including Cholesky, LU and QR factorizations. Then we developed `GreenLA` to exploit algorithmic knowledge of linear algebra operations to predict slack on CPU and GPU, and use application-level DVFS strategies to reclaim the slack for energy savings. Compared to OS level solutions that

§Authors contributed equally; ◇ Corresponding author.

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00 ©2016 IEEE

rely on online learning and prediction for DVFS scheduling decisions, GreenLA accurately pinpoints and fully reclaims the slack, achieving more energy savings with less overhead. As a software package, GreenLA can work in place of existing numerical linear algebra library MAGMA. Moreover, GreenLA can be freely enabled or disabled, less intrusive than the OS level solutions.

The main contributions of this paper are:

- This paper develops GreenLA - an energy efficient linear algebra software package that effectively leverages the algorithmic characteristics of the linear algebra operations to maximize energy savings. GreenLA exploits linear algebra algorithmic knowledge combined with light-weight online profiling to accurately predict the length of tasks and slacks, and hence can maximize the reclamation of slacks via algorithm-based DVFS scheduling.
- GreenLA saves up to three times more energy than the best existing energy saving approaches that do not modify the library source codes;
- GreenLA achieves comparable performance to the highly optimized linear algebra library MAGMA but needs less energy than MAGMA.
- GreenLA is transparent to applications. With the same programming interface as the existing library MAGMA, existing MAGMA users do not need to modify their source codes to benefit from GreenLA.

The remainder of this paper is organized as follows. Section II introduces background knowledge. We present GreenLA design in Section III and its implementation in Section IV. Evaluation methodology and experimental results are provided in Sections V and VI respectively. Section VII discusses the related work and Section VIII concludes the paper.

II. Dense Matrix Factorizations on CPU-GPU Heterogeneous Systems

Dense matrix factorizations solve systems of linear equations, linear least square problems, and eigenvalue/eigenvector problems, etc. The commonly used algorithms include Cholesky, LU and QR factorizations. These algorithms decompose the coefficient matrix A in a linear system $Ax = b$ into simpler forms, such as LL^T and PLU. Consequently the solution x can be calculated using forward and backward substitutions. Widely accepted heterogeneous computing libraries including MAGMA [4] provide routines of these matrix factorizations as standard functionality.

Although each suitable for different problem classes, Cholesky, LU and QR factorizations share similar algorithmic characteristics. Figure 1 illustrates the main steps of highly-optimized dense matrix factorizations on CPU-GPU heterogeneous systems in a global view. Factorizing a panel matrix is a Level 2 BLAS [1] operation and involves diagonal matrices factorization and panel factorization. Due to the low computational complexity and high sequentiality, panel factorization is performed on the CPU, shown as the green/yellow boxes. Updating a trailing matrix is a Level 3 BLAS operation with high computation complexity and high degrees of parallelism. It is performed on the GPU for performance efficiency, shown as the white boxes.

Figure 1 also demonstrates how a dense matrix factorization proceeds on a CPU-GPU platform with data movement between CPU and GPU in a local view. As mentioned, factorizing the panel matrices is executed on the CPU; and updating the trailing matrix is massively parallelized on the GPU. The *panel matrices* calculated on the CPU are offloaded to the GPU and used by the GPU to update the trailing matrices. For the sake of performance, the next panel matrix that is updated on the GPU is immediately copied back to the CPU before the entire trailing matrix finishes. As such, panel factorization is simultaneously executed on the CPU as the rest of trailing matrix is updated on the GPU. These processes proceed by a submatrix block, starting from the left upper corner of the global matrix and finishing when the whole global matrix is fully factorized.

III. GreenLA Energy Saving Methodology

At the coarse level, the matrix factorization algorithms repeatedly assign two dependent types of tasks to CPU and GPU respectively. In each iteration slack presents on both CPU and GPU. GreenLA reclaims the slack for energy savings with three main components. First, it identifies the critical path and slack on the non-critical paths on the CPU and GPU by analyzing the heterogeneous algorithms. Second, it uses algorithmic knowledge to predict and quantify the slack. Third, it exploits DVFS on the CPU and GPU to fully reclaim the slack on the non-critical paths for energy savings. The following subsections present detailed design of each components.

A. Critical Path and Slack Analysis

For task-parallel applications, a *slack* refers to a time period when one computer component idly waits for another. Typical causes of slack include load imbalance, inter-task or interprocess communication, and memory access stalls. Due to the pervasive presence in applications and systems, slack has been recognized as important energy saving opportunities in HPC. In a task-parallel application, a Critical Path (CP) is a particular sequence of tasks spanning from the beginning to the end of the execution where the total slack amounts to zero. While slack on the non-critical paths is usually exploited for energy savings, it is non-trivial to fully reclaim them without impacting application performance [32].

As shown in Figure 1, slack is present on the CPU and GPU in the heterogeneous matrix factorization algorithms. Specifically, the CPU must wait for the next updated panel from the GPU, and the GPU must wait for the factorized panel from the CPU. In addition, either the CPU waits for the GPU to finish updating the trailing matrix or the GPU waits for the CPU to finish factorizing the panel matrix. Moreover, the slack varies over iterations as the task sizes change. Being able to accurately quantify and predict the slack is necessary before reclaiming them for optimal energy savings.

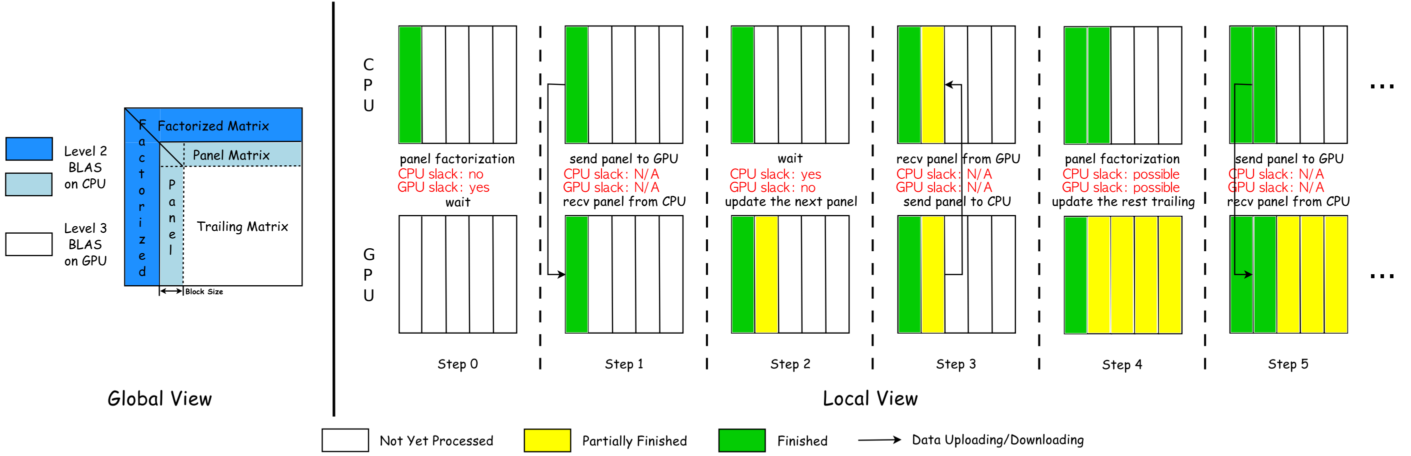

Fig. 1. Global View and Local View of Dense Matrix Factorizations on CPU-GPU Heterogeneous Systems.

B. Online Algorithmic Slack Prediction

For energy saving purposes, we must first know where and when the slack occurs and for how long they last. Given an application, one method to obtain such knowledge is to instrument the source code with timing functions and run the instrumented program with various problem sizes to collect the profiles. Alternatively, OS-level profiling can be performed with hardware performance counters on processors. Neither method is portable, and both methods require extensive profiling. In `GreenLA`, we investigate an *online algorithmic slack prediction* approach that accurately predicts the varying slack at runtime with minimum profiling.

Given the heterogeneous matrix factorization algorithms, the slack on the CPU and GPU is mainly impacted by the software and hardware parameters.

- *Problem size*: the sizes of the panel matrix and trailing matrix scheduled on the CPU and GPU respectively based on the algorithmic characteristics;
- *CPU compute capacity*: the number of floating point operations the CPU are able to perform in one second for the assigned tasks;
- *GPU compute capacity*: the number of floating point operations the GPU are able to performance in one second for the assigned tasks.
- *Data transfer speed*: the number of bytes the GPU are able to transfer between CPU memory and GPU memory.

Figure 2 plots the difference between CPU and GPU task execution time for the first 100 iterations of LU factorization with various problem sizes and compute rates. For instance, 10240 and 20480 are two global matrix sizes, and high_low means that the CPU runs at the highest speed and the GPU runs at the lowest speed. A non-zero difference between the execution time indicates that either CPU or GPU waits for the other in the current iteration. Appropriately adjusting the compute rate of CPU or GPU can narrow and even eliminate the time difference.

In `GreenLA`, we leverage our prior knowledge about the factorization algorithms to quantitatively predict the slack between CPU and GPU in each iteration. Two major factors that will affect the behavior of slack are the CPU/GPU execution time and the data copy time between CPU and GPU.

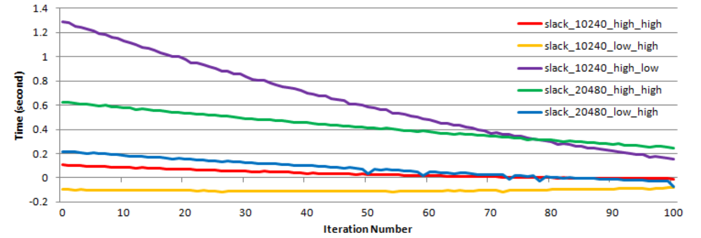

Fig. 2. The difference between CPU and GPU execution time for the first 100 iterations of LU Factorization. The difference varies over iterations, and is a function of problem size and the CPU and GPU compute rates.

We first focus on the execution time. We use LU factorization as an example here. It would be similar for Cholesky and QR factorization. Assuming the execution time of the first iteration of a $N \times N$ with block size nb factorization are known, we denote the CPU time for the $N \times nb$ *panel factorization* as T_0^{CPU}, and the GPU time for the $N' \times N'$ *trailing updating* as T_0^{GPU} where $N' = N - nb$. We can use the algorithmic information to predict the execution time of the remaining iterations. For now we consider fixed compute capacity for the CPU and GPU. As the time complexities of *panel factorization* and *trailing updating* are $O(\mathbb{N}^3)$ for a matrix size $\mathbb{N}$, the execution time for iteration k and $k+1$ have the following relation.

$$\frac{T_{k+1}^{CPU}}{T_k^{CPU}} = \frac{O(\mathbb{N}_{k+1}^3)}{O(\mathbb{N}_k^3)} = \frac{(N-(k+1)\times nb)\times nb^2}{(N-k\times nb)\times nb^2} = 1 - \frac{nb}{N-k\times nb} \quad (1)$$

$$\frac{T_{k+1}^{GPU}}{T_k^{GPU}} = \frac{O(\mathbb{N}_{k+1}^3)}{O(\mathbb{N}_k^3)} = \frac{(N-(k+1)\times nb)^2\times nb}{(N-k\times nb)^2\times nb} = \left(1 - \frac{nb}{N-k\times nb}\right)^2 \quad (2)$$

Similarly, we can also predict the data copy time between CPU and GPU. Assuming the data copy time of the first iteration is T_0^{COPY} for transferring a panel to GPU from CPU and then transfer it back. Assuming the data transfer speed is constant, we can use T_0^{COPY} and the size change of panel over iterations to predict future data copy time. In LU factorization, the space complexity of panel needed to be transferred is $O(\mathbb{N}^2)$, the data copy time for iteration k and $k+1$ have the following relation:

$$\frac{T_{k+1}^{COPY}}{T_k^{COPY}} = \frac{O(\mathbb{N}_{k+1}^2)}{O(\mathbb{N}_k^2)} = \frac{(N-(k+1)) \times nb^2}{(N-k) \times nb^2} = 1 - \frac{1}{N-k} \tag{3}$$

Since, CPU must wait for data copy is done before it can starts it computation, the slack during iteration k can be quantified as follows:

$$slack_k = \left| \, T_k^{CPU} + T_k^{COPY} - T_k^{GPU} \right|; \; 0 \leq k < \frac{N}{nb} \tag{4}$$

A positive value indicates that the GPU have slack time to wait for the CPU, while a negative value indicates that the CPU has slack to wait for the GPU. Value zero means that the CPU and GPU finish the current iterations at the same time. The initial slack is the time difference for the first iteration, i.e., $k = 0$.

$$slack_0 = \left| \, T_0^{CPU} + T_0^{COPY} - T_0^{GPU} \right| \tag{5}$$

From Equations 1-4, we can quantify the slack for the rest CPU-GPU tasks, provided with the execution time of the first iteration. This algorithmic slack prediction is accurate and lightweight compared to OS level slack prediction. By using the prior algorithmic knowledge, only minimal profiling is necessary.

C. CP-Aware Slack Reclamation

With predicted slack over the iterations, we explore *CP-aware slack reclamation* to save energy [22] [28] [32]. To adjust the time it takes for CPU or GPU, we can either adjust the computation time or data copy time. However, since it is usually hard to accurately adjust data transfer speed and it also brings much less energy saving benefit than computation, we focus on adjusting the execution time. Specifically, we employ properly reduced compute capability on the processing units on the non-critical path such that the total slack amount to zero. We exploit DVFS, an effective power-saving hardware technology available on the CPU and GPU for this purpose. DVFS-capable processing units have multiple performance/power states. Prior studies [17] [9] [7] show that running these processing units at lower states during slack significantly reduces power and energy without impacting overall application performance. As shown in Figure 2, with different inputs and power states, even within one run, slack can reside at either CPU side or GPU side. We apply accordingly either CPU DVFS or GPU DVFS depending on the source of slack.

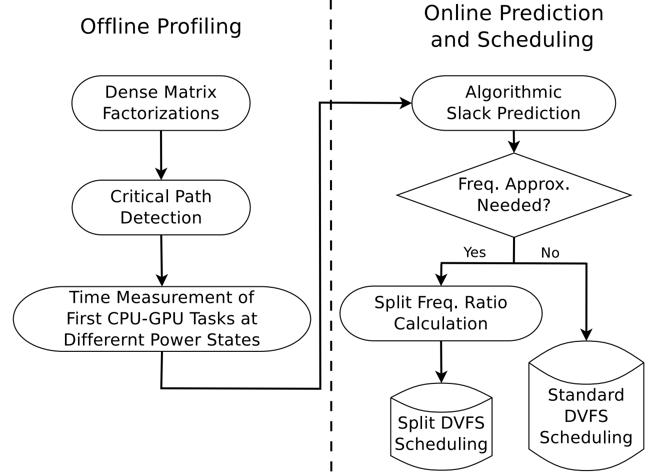

Fig. 3. Offline and Online Framework of GreenLA.

In order to quantitatively evaluate GreenLA, we theoretically analyze the upper bound of possible energy savings for heterogeneous matrix decomposition algorithms. Theoretically, the maximum energy savings is as follows:

$$\Delta E_{sys} = \frac{E_{new}}{E_{old}} = \frac{(P_s^{new} + P_d^{new} + P_{other}) \times T_{new}}{(P_s^{old} + P_d^{old} + P_{other}) \times T_{old}} \tag{6}$$

$$T_{new} = T'_{new} + T_{DVFS} \tag{7}$$

Here, P_s and P_d are the static power and dynamic power respectively consumed by the CPU and GPU, where P_d is a function of processor frequency for DVFS-capable components. P_{other} is the power consumption of other computer components. T_{new} is the execution time of GreenLA including the overhead brings by the DVFS, while T_{old} is the execution time of the original heterogeneous factorization. We denote $P_d = nP_{total}$, $P_s + P_{other} = (1-n)P_{total}$, where n is a ratio of dynamic CPU/GPU power P_d within the total system power costs. By assuming $T_{new} = T_{old}$ and adopting $P_d = \propto f^{2.4}$ from [15], we can simplify Equation 7 as:

$$\Delta E_{sys} = \frac{nP_{total}\frac{f_{new}}{f_{old}}^{2.4} + (1-n)P_{total}}{nP_{total} + (1-n)P_{total}} = 1 - n\left(1 - \frac{f_{new}}{f_{old}}^{2.4}\right)$$
$$= 1 - n\left(1 - \frac{nb}{N}\sum_{1}^{\frac{N}{nb}}\left(\frac{\min(T_k^{CPU}, T_k^{GPU})}{\max(T_k^{CPU}, T_k^{GPU})}\right)^{2.4}\right) \tag{8}$$

IV. GreenLA Design and Implementation

We present the design and implementation details of GreenLA, including strict and relaxed slack reclamation and coupled GPU DVFS.

Figure 3 illustrates the architecture and main components of GreenLA. It minimally profiles the application offline to obtain the execution time of the first CPU-GPU tasks at different power states and first data copy time. Such data is used

to derive the slack time during the first iteration. GreenLA uses online algorithmic slack prediction to accurately obtain the slack for the rest of the CPU-GPU tasks. In cases where an available frequency is unable to eliminate a slack, we split the slack and use two consecutive available frequencies.

TABLE I
NOTATION IN ALGORITHMS AND FORMULATION.

$task$	One task of CPU-GPU dense matrix factorizations
f_l	The lowest CPU/GPU core frequency set by DVFS
f_h	The highest CPU/GPU core frequency set by DVFS
f_l'	The lowest GPU core frequency paired with the highest GPU memory frequency set by DVFS
f_{ideal}	The optimal ideal frequency to eliminate slack
T	Execution time of a task running at f_h
T_x	Execution time of a task running at f_x
$slack$	Amount of time that a task can be delayed by w/o increasing the total runtime of the application
f_{lower}	The neighboring frequency smaller than f_{ideal}
f_{upper}	The neighboring frequency greater than f_{ideal}
$fSet$	The frequency set consisting of all used frequencies
CP	One task trace consisting of tasks to finish the application with the total slack of zero
$LastFreq$	Frequency used after the last frequency scaling
r	Ratio between two durations at split frequencies

A. Strict Slack Reclamation

With strict slack reclamation, we apply DVFS on the non-critical path in each iteration of the factorization algorithm. By properly lowering the frequency of the processing units to just eliminate the slack, we can reduce power consumption without impacting performance for the current iteration.

Prior studies have shown that the execution time of compute-intensive workloads is proportional to the frequency of processing units. Based on this observation, we derive the ideal target frequency for the processing units on the non-critical path for given current and targeting execution time. In GreenLA, we take into account the available discrete frequencies provided by CPU/GPU DVFS. In cases that the ideal target frequency is not equivalent to an available frequency, we use a weighted sum of two available neighboring frequencies and run the processing units at each frequency for a ratio of duration. Table I lists the notation used in the algorithms and formulation henceforth.

Algorithm 1 details the selection of the CPU or GPU frequency if slack occurs and Algorithm 2 presents the frequency approximation [35] [29] with *CP-aware* energy efficient DVFS scheduling. For simplicity and readability, we use five helping functions in these two algorithms: SetFreq(), GetApproxRatio(), GetSlack(), GetCurFreq(), and GetOptFreq(). Of these, SetFreq() is a wrapper of CPU/GPU DVFS APIs that set specific CPU/GPU frequencies and GetCurFreq() is used to inquire the current frequency in use. The other three functions are more complex and will be detailed next.

These three functions use prior knowledge of the mapping between frequency and execution time for the CPU and GPU tasks. Specifically, GreenLA records runtime of the first CPU-GPU tasks at different frequencies by instrumenting

Algorithm 1 *CPU/GPU DVFS Scheduling*

CPU_GPU_DVFS(CP, $fSet$, $task$, k)
1: **if** ($task \in CP$) **then**
2: SetFreq(f_h)
3: **else**
4: $slack \leftarrow$ GetSlack($task$, k)
5: **if** ($slack > 0$) **then**
6: $f_{ideal} \leftarrow$ GetOptFreq($task$, $slack$)
7: $LastFreq \leftarrow$ GetCurFreq()
8: $fSet \leftarrow fSet \cup f_{ideal} \cup LastFreq$
9: **if** ($T_{CPU} < T_{GPU}$) **then** /* CPU DVFS */
10: Call CP_SSR($slack$, $fSet$) or CP_RSR($slack$, $fSet$)
11: **else if** ($T_{CPU} > T_{GPU}$) **then** /* GPU DVFS */
12: Call GPU_DVFS($slack$, $fSet$)
13: **end if**

Algorithm 2 *Strict CP-aware Slack Reclamation*

CP_SSR($slack$, $fSet$)
1: **if** ($f_l \leq f_{ideal} \leq f_h$) **then**
2: **if** ($f_{ideal} \notin fSet$) **then**
3: $r \leftarrow$ GetApproxRatio(T_{lower}, T_{upper}, $slack$)
4: SetFreq(f_{lower}, f_{upper}, r)
5: **else** SetFreq(f_{ideal})
6: **else if** ($f_{ideal} < f_l$) **then**
7: SetFreq(f_l)
8: **end if**

timestamps in the source codes, and use them to estimate the runtime and slack for the rest of the CPU-GPU tasks using Equations 1-4. Given T, T_{lower}, T_{upper}, and $slack$ of each pair of CPU-GPU tasks, we split frequency with ratio r and the ideal frequency f_{ideal} can be solved as follows:

$$\begin{aligned} & T + slack = T_{lower} \times r + T_{upper} \times (1 - r) \\ & f_{ideal} \times (T + slack) = f_h \times T \\ & \text{where } T \mapsto f_h, T_{lower} \mapsto f_{lower}, T_{upper} \mapsto f_{upper} \\ & \text{subject to } f_l' \leq f_{lower} < f_{ideal} < f_{upper} \leq f_h \end{aligned} \tag{9}$$

The resulting target frequency f_{ideal} is compared against the available frequencies (i.e., line 2 in Algorithm 2). If it matches an available frequency, the matched available frequency can be used directly. Otherwise, neighboring frequencies f_{lower} and f_{upper} are assigned in accordance with the ratios r and $1 - r$ individually. In case that f_{ideal} is lower than the lowest available frequency, the lowest available frequency is adopted, as sketched in Algorithm 2.

B. Relaxed Slack Reclamation

In contrast to strict slack reclamation that applies DVFS at each iteration of the algorithms, relaxed slack reclamation forms multiple iterations into groups and apply DVFS at the group level. Relaxed slack reclamation offers two advantages. First, it reduces the time and energy overhead incurred by

Algorithm 3 *Relaxed CP-aware Slack Reclamation*

```
CP_RSR(slack, fSet)
 1: if (f_l ≤ f_ideal ≤ f_h) then
 2:    if (f_ideal ∉ fSet) then
 3:       r ← GetApproxRatio(T_lower, T_upper, slack)
 4:       if (r < RlxFctr) then
 5:          if (LastFreq ≠ f_upper) then
 6:             SetFreq(f_upper)
 7:             LastFreq = f_upper
 8:          else Do Nothing
 9:       else SetFreq(f_lower, f_upper, r)
10:    else SetFreq(f_ideal)
11: else if (f_ideal < f_l) then
12:    SetFreq(f_l)
13: end if
```

frequent DVFS scheduling [31]. Second, it reclaims extra slack on the CPU and GPU caused by data dependencies, in addition to the slack caused by workload imbalance that strict reclamation targets. In each iteration, the GPU waits for the factorized panel from the CPU, and the CPU waits for the updated panel from the GPU. The slack is reclaimed by relaxed slack reclamation but not by strict slack reclamation.

We use a *relaxation factor* ($RlxFctr$ in Algorithm 3) to determine the number of iterations in a group for scheduling decisions. As shown in Algorithm 3, if the calculated split frequency ratio r is less than $RlxFctr$ (e.g., 0.05), the duration at f_{lower} is negligible according to Equation 9. In this case, we run the processing units at f_{upper}. The selection of $RlxFctr$ is based on algorithmic characteristics. Slack varies with iterations and the variation rate provides us the criteria of choosing an appropriate $RlxFctr$ for the optimized energy efficiency. Note that even with a constant $RlxFctr$, the number of iterations in a group may vary over time during a run.

C. Coupled GPU Core and Memory DVFS

DVFS can be applied to power-scalable hardware components, including CPU, GPU, and memory. It is noteworthy that on today's architectures such as GPU, core and memory frequencies are coupled and have to be switched simultaneously as a combination [5], which differs from CPU DVFS where only core frequency is scaled. Table II lists memory-core frequency pairs for two NVIDIA GPU.

TABLE II
GPU MEM.-CORE FREQ. PAIRS (UNIT: MHZ).

NVIDIA Kepler Tesla K20c		NVIDIA Kepler Tesla K40c	
Memory Freq.	Core Freq.	Memory Freq.	Core Freq.
2600	758 705 666 640 614	3004	875 810 745 666
324	324	324	324

Algorithm 4 *GPU Core/Memory DVFS Scheduling*

```
GPU_DVFS(slack, fSet)
 1:  if (f'_l ≤ f_ideal ≤ f_h) then
 2:     Call CP_SSR(slack, fSet) or CP_RSR(slack, fSet)
 3:  else if (f_l ≤ f_ideal < f'_l) then
 4:     r ← GetApproxRatio(T_l, T'_l, slack)
 5:     SetFreq(f_l, f'_l, r)
 6:  else if (f_ideal < f_l) then
 7:     SetFreq(f_l)
 8: end if
```

As discussed earlier, in our scenario, slack may occur either on the CPU side or on the GPU side. Algorithm 1 strategically makes CPU/GPU DVFS decisions depending on the source of slack. In particular, for the case of eliminating slack from GPU side, due to the coupled core and memory frequencies of GPU, a combined GPU core/memory DVFS scheduling strategy is necessary. Line 3-7 in Algorithm 4 details the combined strategy, where split frequency ratio r is calculated similarly as Equation 9. Equation 10 shows the calculation, and the difference is that scaling down to f_l pairs with memory frequency reduction to the lowest.

$$\begin{aligned} &T + slack = T_l \times r + T_l' \times (1-r) \\ &f_{ideal} \times (T + slack) = f_h \times T \\ &\text{where } T \mapsto f_h, T_l \mapsto f_l, T_l' \mapsto f_l' \\ &\text{subject to } f_l \leq f_{ideal} < f_l' \end{aligned} \tag{10}$$

In our experiments, GPU tasks that update the trailing matrices involve considerable computation and memory accesses. Therefore simultaneously decreasing GPU core/memory frequencies has dual performance impact on computation and memory accesses. Our approach takes care of these scenarios since the dual slowdown has been recorded in T', which is the runtime of a task at the lowest core and memory frequencies.

V. EVALUATION

In this section we detail the evaluation of `GreenLA` on a GPU-accelerated heterogeneous system: a linear algebra library of dense matrix factorizations (Cholesky, LU and QR) with an energy efficient CPU/GPU DVFS co-scheduling approach via online algorithmic slack prediction.

A. Evaluation Methodology

For comparison purposes, we present a state-of-the-art OS level method for slack prediction, and another type of classic DVFS scheduling strategy for saving energy. We stress the difference in other approaches against ours, and argue that our solution can outperform in both the accuracy of slack prediction and the amount of energy savings in our scenario. *OS Level Slack Prediction.* As another important slack prediction method, *OS level slack prediction* either work for a specific type of applications sharing similar features, e.g., with stable/slowly-varying execution characteristics, or require considerable training to obtain accurate prediction results.

Online prediction mechanism presented in [24] [30] [29] is based on a simple assumption that task behavior is identical every time a task is executed. It is however defective for applications with variable workloads, such as matrix factorizations, where the remaining unfinished matrices become smaller as the factorizations proceed. Execution time shrinks and slack varies as the workloads become lighter, which invalidates the above prediction mechanism.

Regardless of the simplest prediction above, several enhanced history-based workload prediction algorithms have been proposed to handle the variation in HPC runs and produce more accurate prediction results [34] [12] [21] [18]. The RELAX algorithm employed in CPU MISER [18] exploits both prior predicted profiles and current runtime measured profiles: $W'_{i+1} = (1-\lambda)W'_i + \lambda W_i$, where λ is a relaxation factor for adjusting the percentage of dependent information on the current measurement. This enhanced prediction can also be error-prone for dense matrix factorizations, since using a fixed λ cannot handle length variation of iterations of the core loop due to the shrinking remaining unfinished matrices. The use of 2-D block cyclic data distribution further brings complexity to the prediction. Moreover, statistical predictive models have been adopted for accurate workload prediction, e.g., Hidden Markov Models (HMM) used in [36] and Predictive Bayesian Network (PBN) used in [23]. Using offline training and learning based on historical records, results with high accuracy were achieved (average prediction error 3.3% via HMM and 0.43% via PBN). Although effective offline, online slack prediction for HPC applications using statistical models can be costly: Considerable amount of execution traces are required to train the statistical predictive models for accurate slack prediction. For instance, the training dataset in [36] was obtained by running applications on one server and evaluated on one different server, which can be impractical for HPC runs as discussed earlier.

Race-to-halt Energy Saving. As the name suggests, *race-to-halt* (or *race-to-idle*) is an energy saving strategy that enforces power-scalable processors (e.g., CPU and GPU) to *race* when workloads are ready for processing, and to *halt* when no tasks are present and the processors are idle/waiting. In other words, *race* refers to executing the workloads at the highest frequency and voltage of the processors for the peak performance until the finish of the workloads, and *halt* implies that processor frequency and voltage are switched to the lowest level from the end of the last executed workload to the start of the next workload. This straightforward solution can effectively save energy without incurring performance loss due to the following inferences: (a) The peak performance of processors is guaranteed during computation as in original runs; (b) the peak performance of processors is not needed when no tasks are being executed and processors are waiting for data. As discuss earlier, in our scenario, slack can arise at either CPU side or GPU side, depending on various factors. In either case, the peak performance of the idle/waiting processors is not necessary. Per *race-to-halt*, we apply to them the lowest power state during the slack and switch back to the highest one until the next workload is available. *race-to-halt* is *CP-free* such that no CP detection is required before any energy saving decisions are made. Thus it is generally lightweight and easy to implement.

We implemented OS and library level approaches using *race-to-halt* and online HMM-enabled statistical slack prediction individually. Further, we implemented relaxed slack reclamation to compare with the default strict slack reclamation. Evaluated metrics include slack prediction accuracy, and energy and performance efficiency. For readability, we henceforth denote different test cases as follows:

- `MAGMA`: The original MAGMA runs of different-scale CPU-GPU Cholesky, LU and QR factorizations without any energy saving approaches;
- `OS_r2h`: The OS level implementation [2] based on a CPU race-to-halt workload prediction algorithm similar to the RELAX algorithm;
- `lib_r2h`: The library level implementation based on algorithmic race-to-halt on both CPU and GPU;
- `OS_cpsr_str`: The OS level implementation based on online HMM-enabled statistical slack prediction, with strict slack reclamation for each iteration;
- `OS_cpsr_rlx`: The OS level implementation based on online HMM-enabled statistical slack prediction, with relaxed slack reclamation ($RlxFctr = 0.05$) for blocked iterations;
- `lib_cpsr_str`: The library level implementation based on online algorithmic slack prediction, with strict slack reclamation for each iteration;
- `lib_cpsr_rlx`: The library level implementation based on online algorithmic slack prediction, with relaxed slack reclamation ($RlxFctr = 0.05$) for blocked iterations.

Among all energy saving solutions, `lib_cpsr_rlx` empirically achieves the optimal energy efficiency with negligible performance loss, and thus we adopt it as our `GreenLA` in the comparison against MAGMA later.

B. Experimental Setup

We applied all above test scenarios to CPU-GPU Cholesky, LU and QR factorizations (MAGMA version 1.6.1) with multiple global matrix sizes each (ranging from 5120 to 20480). However, due to limit space, we only show the result for input size of 20480×20480. For other input matrix sizes, the results are similar. All experiments were performed on a power-aware many-core CPU-GPU server. Table III lists hardware configuration of the experimental platform. The total system dynamic and static/leakage energy consumption of the above runs was measured using nvidia-smi tool [5] provided by NVIDIA, and following [26], we used PowerPack [19], an integrated software/hardware framework for profiling and analysis of power/energy costs of HPC systems and applications. A separate meter node with PowerPack deployed was used to collect power/energy costs of all hardware components of the system, and the data was recorded in a log file and accessed after the above runs.

TABLE III
HARDWARE CONFIGURATION FOR EXPERIMENTS.

Component	CPU	GPU
Processor	2×10-core Intel Xeon Ivy Bridge E5-2670	2496 CUDA-core NVIDIA Kepler GK110 Tesla K20c
Peak Perf.	0.4 TFLOPS	1.17 TFLOPS
Core&Mem. Freq. Gear	Core:1.2-2.5(↑by0.1)GHz Mem.:Not DVFS-capable	See left column of Table II
Memory	64 GB RAM	5 GB RAM
Cache	64 KB L1, 256 KB L2, 25.6 MB L3	13 SMX units, 64 KB and 48 KB read-only d-cache
OS	Fedora 21, 64-bit Linux kernel 3.17.4	
Pwr. Meter	PowerPack	nvidia-smi with -ac option

VI. RESULTS

Next we present experimental results of our evaluation via fine-grained comparison. We first demonstrate the performance and energy efficiency of our approach by comparing to the widely used numerical linear algebra MAGMA library.

TABLE IV
AVERAGE ERROR RATES OF SLACK PREDICTION FOR FOUR RUNS EACH OF CHOLESKY/LU/QR FACTORIZATION.

Benchmarks & Test Scenarios	OS Level Statistical Slack Prediction		Library Level Algorithmic Slack Prediction
	Base Iter. (First 10%)	Base Iter. (First 20%)	
Cholesky (5120 - 20480)	10.51%	6.62%	0.96%
LU (5120 - 20480)	9.95%	5.45%	0.16%
QR (5120 - 20480)	11.29%	5.77%	0.52%

A. Average Error Rate of Slack Prediction

We first showcase the accuracy of slack prediction of the OS level statistical approach with two training datasets and our library level algorithmic approach. Table IV summarizes the average error rate of slack prediction of the two approaches. As stated in section 4.3, HMM-based statistical slack prediction requires a group of *base* iterations (usually the first few iterations of a HPC run) to serve as an online training dataset. The prediction accuracy is highly associated with the size of the training dataset according to Table IV: The more *base* iterations are used, the more accuracy is achieved. However, the highest accuracy of the OS level approach is 5.45% for LU, while our library level approach can be as accurate as having a 0.16% error rate for LU. This low error rate of slack prediction can greatly facilitate forthcoming energy saving. For further comparison, we select the OS level statistical approach with higher slack prediction accuracy for more experiments.

B. Total Energy Saving Comparison

As shown in Figure 4, our library level CP-aware slack reclamation approaches could save more energy than current state-of-the-art approaches. Current approaches could only either save less energy or even costs extra energy. For example, OS level race-to-halt only slows down CPU when CPU utilization is below a threshold, while library level race-to-halt reduces both CPU and GPU speed when no corresponding workloads are running according to algorithmic characteristics. Due to high online probing overhead, the OS level race-to-halt approach incurs even more energy consumption, while the more lightweight library level race-to-halt approach can save minor energy savings (up to 3.6%) as shown in Figure 4. Other two approaches we compared are OS level approaches statistical slack prediction `OS_cpsr_str` and `OS_cpsr_rlx`. Since the two solutions produced inaccurate slack prediction, the inaccuracy results in inappropriate timing and duration of DVFS, which cannot eliminate possible slack – saving less energy than the optimal, or incurs performance loss due to overdue or overdone DVFS – consuming even more energy than the original run. As shown in Figure 4, those two approaches consumed more energy(1%–2%). Even if the OS level slack prediction can achieve the same accuracy as our library level approaches, the OS level solutions can waste much more energy saving opportunities than our library level approaches due to the considerable amount of iterations used for training, compared to only execution information of the first iteration needed by our library approach. On the other hand, our library level CP-aware slack reclamation approaches could save several times more energy. specifically, the energy saved from our approach is 2.5x of the energy saved using current best approach in Cholesky factorization, 1.5x in LU factorization and 3x in QR factorization. Moreover, different than our strict slack reclamation, which tries to reclaim all slacks using DVFS, our relaxed slack reclamation only tries to apply DVFS when split frequency ratio is larger than the $RlxFctr$, which eliminated some unnecessary power state adjustments. The reduced number of power adjustment brings less DVFS performance overhead, which further saves more energy for the overall application. Note that, the Cholesky, LU and QR factorizations are very compute intensive. Based on the energy efficiency model in [11], their heterogeneous implementations in MAGMA have really high computation efficiency, which make them hard to save more energy. Although there are many hardware components involve the execution process, we only focus on reducing the energy consumption of CPU and GPU.

C. CPU Energy Saving Comparison

Now, we focus on the energy saving on the CPU side. Note that since we only focus on reducing the energy cost of CPU, the energy measurement here does not include RAM energy consumption. We can see from Figure 5 the single core CPU energy comparison of Cholesky, LU and QR factorization using different energy saving solutions. As mentioned before, OS level race-to-halt only adjust the performance/power state of the CPU, which also introduce more probing overhead, it cost more energy on the CPU side (7%–8%). As for the OS level online HMM-enabled statistical slack prediction approaches, they need first 10%–20% iterations to do online training on the CPU, which not only brings more overhead, but also wastes valuable slack reclamation(energy saving) opportunities. However, thanks to the high accurate algorithmic slack prediction, our library level CP-aware slack reclamation

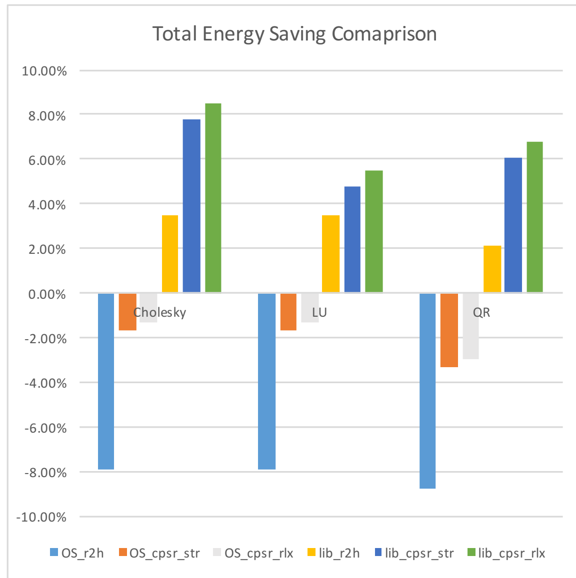

Fig. 4. The total amount of energy saved in percentage using several different energy saving solutions. Right two bars in each group show the result of our approach, which can save up to 3x more energy than the current best solution.

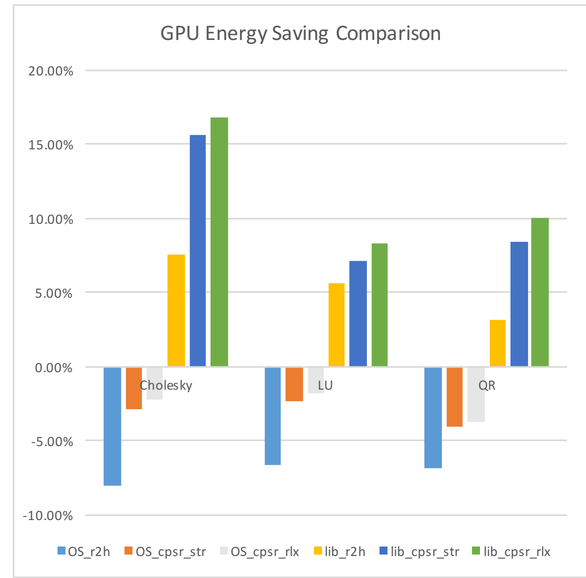

Fig. 6. The amount of energy saved in percentage on GPU using several different energy saving solutions. Positive values indicate energy saving. Negative values indicate extra energy cost. Our approaches(right two bars in each group) shows more energy saving on GPU than existing state-of-the-art solutions.

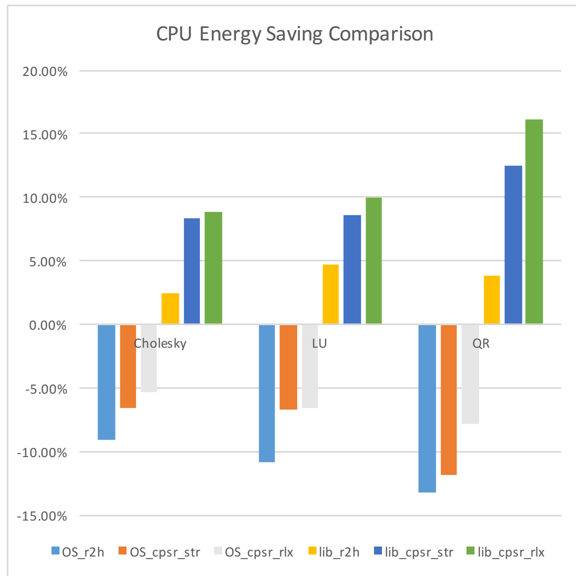

Fig. 5. The amount of energy saved in percentage on single core CPU using several different energy saving solutions. Positive values indicate energy saving. Negative values indicate extra energy cost. Our approaches(right two bars in each group) shows more energy saving on CPU than existing state-of-the-art solutions.

Fig. 7. Execution time of Cholesky, LU and QR factorization using several different energy saving solutions. Right two bars in each group show our results, which have similar performance than existing state-of-the-art solutions.

approaches could save more energy on the CPU side when the slack resides on CPU.

D. GPU Energy Saving Comparison

Next, we focus on the energy saving on the GPU side. As we can see from Figure 6, even GPU is assigned more computation tasks, it usually finish its tasks faster than the CPU, so slacks are more likely to occur on the GPU side, and thus it can save more energy. For library level race-to-halt approach, since it rely on algorithmic execution time prediction, it can save energy to some degree. But it still waste some energy in "halt" state, so it saves less energy than our approach. As for OS level race-to-halt, it does not adjust the performance/power of the GPU at all and poor CPU side adjustment brings more performance overhead, so the overall execution time is prolonged, which results higher GPU energy consumption. Similar as on the CPU, OS level online prediction approach suffers from inaccurate slack prediction, which leads to higher GPU energy consumption. Our approaches, on the other hand, could save up to 16% energy on the GPU.

E. Time Overhead

All kinds of performance loss is observed from the experiments as shown in Figure 7. Typical performance degrading factors for OS level solutions consist of dynamic monitoring overhead (`OS_r2h`), online training overhead (`OS_cpsr_str` and `OS_cpsr_rlx`), DVFS overhead (all solutions), and performance loss from overdue or overdone DVFS due to inaccurate slack prediction (`OS_cpsr_str` and `OS_cpsr_rlx`). On the other hands, observed performance loss for library level solutions is from frequency approximation errors and DVFS overhead. Among all approaches, `OS_cpsr_str` has the highest performance loss (up to 14.4%, due to overdone DVFS from inaccurate slack prediction), while our `lib_cpsr_str`/`lib_cpsr_rlx` incurs minor performance loss (as low as 1.2%).

VII. Related Work

The growing prevalence of heterogeneous architectures has motivated a large body of energy efficient approaches[27], but few of them were designed specifically for numerical linear algebra operations, such as dense matrix factorizations extensively used in HPC. Some efforts presented next can also be applied to heterogeneous systems with similar techniques.

OS-level Scheduling. Liu *et al.* [25] proposed several power-aware techniques for a CPU-GPU heterogeneous system including two static/dynamic mapping algorithms and one aggressive voltage reduction scheme. Decent power and energy savings were achieved (more than 20%) towards several matrix workloads, but our work targets different matrix algorithms with less slack and thus less energy saving opportunities. Their work focused on time-sensitive applications that require significant computational capacity, such as real-time scoring of bank transactions, live video processing, etc. Our work can also meet fine-grained timing requirements of applications, which is guaranteed by respecting tasks on the critical path. Hong *et al.* [20] proposed an integrated power and performance model to statically determine the optimal number of processors for a given application running on GPU, based on the intuition that using more cores is not necessary for applications reaching the peak memory bandwidth. By using fewer GPU cores, average 11% energy savings can be achieved for memory bandwidth limited applications. The proposed system can be used by a thread scheduler for online energy saving decision-making. This approach was also evaluated on different applications than us – the more energy savings do not demonstrate their strength.

Library-level Scheduling. Alonso *et al.* [8] incorporated two energy saving techniques at library level to schedule the computation of dense linear algebra operations on a hybrid platform of a multicore CPU and multiple GPU. Specifically, idle threads were blocked when no tasks to process, and busy-waiting threads were also blocked by synchronization primitives when waiting for a device to finish its work. Due to lack of consideration of algorithmic characteristics of dense linear algebra operations, the reported average energy cost reduction was around 4% for Cholesky and 7% for LU. Anzt *et al.* [10] applied energy efficient techniques on GPU-accelerated iterative linear solvers for memory-intensive sparse linear systems, and demonstrated that considerable energy savings (17.8% on average) can be fulfilled without harming performance noticeably, by setting CPU to a low power state during the time when GPU is running while CPU is busy-waiting. However, the proposed solution cannot work for our scenario where CPU and GPU frequently interact with data movement. Note that there exists more slack for sparse linear algebra operations and more energy savings are expected compared to dense ones.

Online and Offline Workload Prediction. There exist numerous solutions that predict workload and slack, facilitating energy saving decision-making, spanning from online to offline. Zhu *et al.* [36] proposed a power-aware consolidation scheme of scientific workflow tasks for energy and resource cost optimization. The pSciMapper framework consists of online consolidation and offline analysis for resource usage prediction (e.g., CPU utilization) using Hidden Markov Model (HMM), with reported average prediction error of moderate 3.3%. However, the drawback of this approach is considerable slowdown around 15%, which is unacceptable in HPC nowadays. For comparison purposes, we also adopt HMM-based slack prediction in an online fashion instead, and experimental results indicate a higher prediction error (up to 11.29%) can be incurred. Li *et al.* [23] applied a Predictive Bayesian Network to identify daily workload patterns and adjust resource provisioning accordingly for cloud datacenters. The prediction algorithm was evaluated to be considerably effective (only 0.43% average prediction error was observed). Our work differs from this offline workload prediction – `GreenLA` is able to achieve energy savings online for HPC runs using negligible amount of training dataset from the earlier stage of the runs. Tse *et al.* [33] proposed a novel Monte Carlo simulation framework that supports multiple types of hardware accelerators (FPGA and GPU) and provided scheduling interfaces to adaptively perform load balancing at runtime for performance and energy efficiency. The energy savings achieved is however from performance gain obtained from the collaborative simulation framework, not from an energy efficient strategy.

VIII. Conclusions and Future Work

Energy efficiency is becoming a critical factor of concern when achieving parallelism in high performance scientific computing in this era. The growing prevalence of heterogeneous architectures nowadays brings more concerns on saving energy for the emerging systems. Essentially fulfilling energy efficiency requires accurate slack prediction with minor performance degradation. Existing energy efficient approaches span from OS level to application level, which can either be

inaccurate or cost-inefficient due to variable execution patterns of the target applications and lengthy training of the employed prediction model. In this paper, we propose a lightweight energy efficient approach for widely used numerical linear algebra software that utilizes algorithmic characteristics to obtain accurate slack prediction and thus gain the optimal energy savings. Experimental results on a many-core CPU-GPU platform demonstrate that our library level solution can achieve up to 8.5% energy saving than original implementation with negligible performance loss (as low as 1.2%), which 3x more energy savings compared to classic race-to-halt and workload prediction approaches.

Although the currently achieved energy savings are moderate, provided a limited amount of slack for the target applications, more energy can be saved by reducing the minor performance loss incurred by our approach. It is practical and worthwhile since careful and fine-grained DVFS analysis is able to further decrease the number of DVFS switches and errors of frequency approximation. Possible energy savings can also be obtained from improved application characteristics that facilitate power reduction, such as CPU workload centralization and idle/unused core isolation, etc. We are also interested in investigating the energy impact of matrix factorization block sizes. It is possible that the optimal block size for performance differs from the optimal block size for energy costs. There may exist a trade-off between them. We further plan to extend the work to more scientific applications on other emerging hardware and architectures in the near future.

ACKNOWLEDGMENT

The authors would like to thank NVIDIA for providing GPU devices used for experiments. This work is partially supported by the NSF grants CCF-1305622, ACI-1305624, CCF-1513201, CCF-1551511, the SZSTI basic research program JCYJ20150630114942313, and the Special Program for Applied Research on Super Computation of the NSFC Guangdong Joint Fund (the second phase).

REFERENCES

[1] *BLAS (Basic Linear Algebra Subprograms)*. http://www.netlib.org/blas/.

[2] *CPUSpeed*. http://carlthompson.net/Software/CPUSpeed/.

[3] *Green500 Supercomputer Lists*. http://www.green500.org/.

[4] *MAGMA (Matrix Algebra on GPU and Multicore Architectures)*. http://icl.cs.utk.edu/magma/.

[5] *NVIDIA System Management Interface (nvidia-smi)*. https://developer.nvidia.com/nvidia-system-management-interface/.

[6] *TOP500 Supercomputer Lists*. http://www.top500.org/.

[7] P. Alonso, M. F. Dolz, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. DVFS-control techniques for dense linear algebra operations on multi-core processors. *CSRD*, 27(4):289–298, Nov. 2012.

[8] P. Alonso, M. F. Dolz, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Reducing energy consumption of dense linear algebra operations on hybrid CPU-GPU platforms. In *Proc. ISPA*, pages 56–62, 2012.

[9] P. Alonso, M. F. Dolz, R. Mayo, and E. S. Quintana-Ortí. Improving power efficiency of dense linear algebra algorithms on multi-core processors via slack control. In *Proc. HPCS*, pages 463–470, 2011.

[10] H. Anzt, V. Heuveline, J. Aliaga, M. Castillo, J. C. Fernández, R. Mayo, and E. S. Quintana-Ortí. Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms. In *Proc. IGCC*, pages 1–6, 2011.

[11] J. Choi, D. Bedard, R. J. Fowler, and R. W. Vuduc. A roofline model of energy. In *IPDPS*, pages 661–672. IEEE Computer Society, 2013.

[12] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *Proc. DATE*, pages 18–28, 2004.

[13] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power mamagement via dynamic voltage/frequency scaling. In *Proc. ICAC*, pages 31–40, 2011.

[14] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. Memscale: Active low-power modes for main memory. In *Proc. ASPLOS*, pages 225–238, 2011.

[15] R. Efraim, R. Ginosar, C. Weiser, and A. Mendelson. Energy aware race to halt: A down to EARtH approach for platform energy management. *IEEE Computer Architecture Letters*, 13(1):25–28, Jan. 2014.

[16] V. W. Freeh and D. K. Lowenthal. Using multiple energy gears in MPI programs on a power-scalable cluster. In *Proc. PPoPP*, pages 164–173, 2005.

[17] R. Ge, X. Feng, and K. W. Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *Proc. SC*, page 34, 2005.

[18] R. Ge, X. Feng, W.-C. Feng, and K. W. Cameron. CPU MISER: A performance-directed, run-time system for power-aware clusters. In *Proc. ICPP*, page 18, 2007.

[19] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. W. Cameron. PowerPack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658–671, May 2010.

[20] S. Hong and H. Kim. An integrated GPU power and performance model. In *Proc. ISCA*, pages 280–289, 2010.

[21] C.-H. Hsu and W.-C. Feng. A power-aware run-time system for high-performance computing. In *Proc. SC*, page 1, 2005.

[22] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. ISLPED*, pages 197–202, 1998.

[23] J. Li, K. Shuang, S. Su, Q. Huang, P. Xu, X. Cheng, and J. Wang. Reducing operational costs through consolidation with resource prediction in the cloud. In *Proc. CCGrid*, pages 793–798, 2012.

[24] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. In *Proc. SC*, page 107, 2006.

[25] C. Liu, J. Li, W. Huang, J. Rubio, E. Speight, and X. Lin. Power-efficient time-sensitive mapping in heterogeneous systems. In *Proc. PACT*, pages 23–32, 2012.

[26] H. Ltaief, P. Luszczek, and J. Dongarra. Profiling high performance dense linear algebra algorithms on multicore architectures for power and energy efficiency. *Computer Science - R&D*, 27(4):277–287, 2012.

[27] S. Mittal and J. S. Vetter. A survey of methods for analyzing and improving GPU energy efficiency. *CoRR*, abs/1404.4629, 2014.

[28] N. B. Rizvandi, J. Taheri, and A. Y. Zomaya. Some observations on optimal frequency selection in DVFS-based energy consumption minimization. *Journal of Parallel Distributed Computing*, 71(8):1154–1164, Aug. 2011.

[29] B. Rountree, D. K. Lowenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: Making DVS practical for complex HPC applications. In *Proc. ICS*, pages 460–469, 2009.

[30] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz. Bounding energy consumption in large-scale MPI programs. In *Proc. SC*, pages 1–9, 2007.

[31] L. Tan, L. Chen, Z. Chen, Z. Zong, R. Ge, and D. Li. HP-DAEMON: High performance distributed adaptive energy-efficient matrix-multiplication. In *Proc. ICCS*, pages 599–613, 2014.

[32] L. Tan and Z. Chen. Slow down or halt: Saving the optimal energy for scalable HPC systems. In *Proc. ICPE*, 2015.

[33] A. H. T. Tse, D. B. Thomas, K. H. Tsoi, and W. Luk. Dynamic scheduling Monte-Carlo framework for multi-accelerator heterogeneous clusters. In *Proc. FPT*, pages 233–240, 2010.

[34] A. Varma, B. Ganesh, M. Sen, S. R. Choudhury, L. Srinivasan, and J. Bruce. A control-theoretic approach to dynamic voltage scheduling. In *Proc. CASES*, pages 255–266, 2003.

[35] A. S. Wu, H. Yu, S. Jin, K.-C. Lin, and G. Schiavone. An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):824–834, Sept. 2004.

[36] Q. Zhu, J. Zhu, and G. Agrawal. Power-aware consolidation of scientific workflows in virtualized environments. In *Proc. SC*, pages 1–12, 2010.