Phase-based Frequency Scaling for Energy-efficient Heterogeneous Computing

Lorenzo Carpentieri*, Antonio De Caro*, Majid Salimi Beni†*, Kaijie Fan*, and Biagio Cosenza*

*Department of Computer Science, University of Salerno, Italy

†Faculty of Informatics, Vienna University of Technology, Austria

***Abstract*—Energy efficiency has been a major challenge for exascale computing. Frequency scaling is a powerful technique to achieve energy savings in modern heterogeneous systems, and can be applied either at a coarse granularity, by application, or at a fine granularity, by setting the frequency for each computational kernel. The chosen granularity significantly impacts the performance and energy consumption of applications due to frequency-change overhead.**

We propose a novel phase-based method that minimizes the frequency-change overhead and improves performance and energy efficiency on heterogeneous multi-GPU systems. Our approach detects different phases through application profiling and DAG analysis, and sets an optimal frequency for each phase. Our methodology also considers MPI programs, where the overhead can be hidden by overlapping frequency-change with communication. Experimental results show up to 37% energy saving and $1.87\times$ speedup for various benchmarks on a single GPU, and 68% energy saving and $3.63\times$ speedup on two multi-GPU applications.

***Index Terms*—DVFS, Frequency Scaling, Energy Efficiency, Heterogeneous Computing, MPI, SYCL**

I. Introduction

Energy efficient computing has become an important research topic spanning various fields, including cloud [1], [2] and edge computing [3]–[7], big data analytics [8]–[10], fog [11], and in-memory computing [12]. Energy efficiency is also a major challenge for exascale computing as large-scale computing systems consume significant amounts of energy, leading to increased costs for electricity and cooling [13]–[15], contributing to higher carbon emissions [7], [16], [17]. A significant amount of research has focused on various techniques to achieve better energy efficiency in large-scale HPC systems [18], [19]. In particular, Dynamic Voltage and Frequency Scaling (DVFS) and power capping are two core techniques that have been proven to significantly reduce energy consumption [20]–[23].

While frequency scaling has been successfully applied to GPUs, its application to large-scale heterogeneous systems presents several challenges. In many large-scale clusters, frequency scaling is typically not available to users due to potential technical issues. Energy management frameworks such as GEOPM [24] and EAR [25] address this issue by providing interfaces for energy monitoring and control. They also integrate with existing job schedulers [26]. However, relying solely on the job scheduler to implement energy optimization techniques such as frequency scaling is a limiting factor in achieving higher energy savings. In fact, in such *coarse-grained* approaches, frequency scaling is applied at the job level, meaning the entire application will run with the same frequency.

Research has shown that different kernels can have diverse energy characterization, resulting in different optimal frequencies on GPU architectures [27], [28]. In fact, the *fine-grained* approaches have shown higher energy savings compared to the coarse-grained methods [29].

Unfortunately, changing the frequency is not free of cost. We have experimentally calculated a time-to-change the frequency of $0.33\,\mathrm{ms}$, $0.30\,\mathrm{ms}$, and $0.60\,\mathrm{ms}$, on AMD MI100, Intel Max 1100, and NVIDIA V100S GPUs, respectively. This means that, especially for large applications with multiple tasks (computational kernels), changing the frequency for each kernel execution can incur in unnecessary performance and energy costs. While purely fine-grained approaches may incur unnecessary overhead due to frequent frequency changes, it is also possible to hide the latency of frequency changes by overlapping them with communication. This can be achieved in multi-GPU and multi-node applications using, for example, asynchronous MPI communications.

This paper proposes a novel energy optimization methodology that applies frequency scaling to distributed heterogeneous systems. The proposed methodology overcomes the limitations of existing coarse- and fine-grained approaches by proposing an adaptive phase-based approach. Given a *Direct Acyclic Graph* (DAG) representation of the computation, for example, extracted from a SYCL task graph [30], along with profiling information (i.e., time, energy, loops), our approach automatically identifies energy phases and applies the optimal frequency for each phase. Furthermore, by encoding MPI information in the DAG, our method can further hide the frequency-change latency through communication overlap in multi-GPU and multi-node applications. Our methodology is fundamentally abstracted from the underlying hardware, and we demonstrate its portability to GPU systems from various vendors (i.e., AMD, Intel, NVIDIA).

In summary, this paper makes the following contributions:

- A novel phase-based frequency scaling methodology for heterogeneous systems based on the task DAG and profiling of the application.

- An extension of phase-based frequency scaling for multi-GPU and multi-node programs, which extends the task DAG with information about the application's MPI communications, allowing for efficient overlap of the frequency change overhead with the MPI communications.
- An experimental evaluation of five single-GPU benchmarks on the AMD MI100, Intel Max 1100, and NVIDIA V100S GPUs, along with an energy scalability analysis of two real-world applications that scale across 4 Intel Max 1100 and up to 16 NVIDIA A30 GPUs.

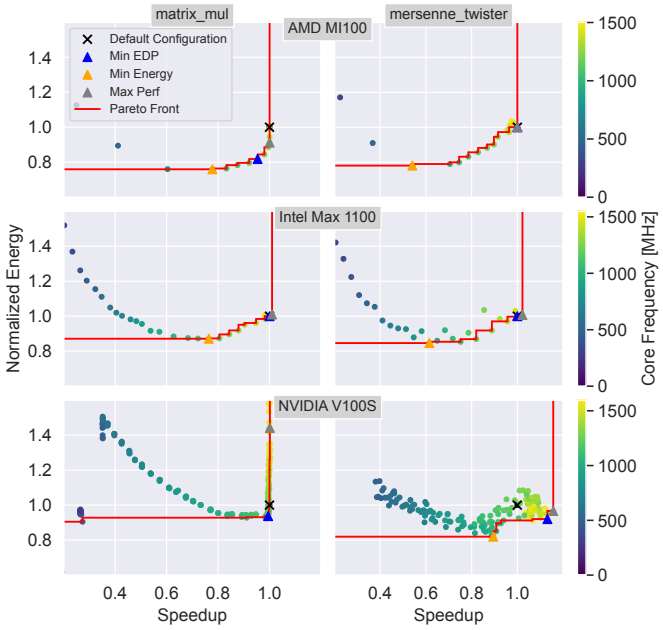


Fig. 1: Multi-objective characterization of `matrix_mul` (left) and `mersenne_twister` (right) benchmarks.

II. Background

A. Impact of Frequency Scaling

Nowadays, most GPU vendors provide an energy management interface to enable power and energy profiling and core frequency scaling. Examples include AMD's ROCm SMI [31], Intel's Level Zero [32], and NVIDIA's NVML [33]. Frequency scaling, as an optimization strategy, can significantly reduce the energy consumption of a task. While some hardware supports changing both core and memory frequency, most modern data center GPUs only allow changing the core frequency. Generally, energy efficiency comes at the cost of performance, creating a multi-objective problem where we can investigate different trade-offs.

Figure 1 shows the energy-speedup tradeoff for two kernels on three different GPUs. *Default configuration* shows the default GPU frequency set by the manufacturer. For NVIDIA GPUs, this corresponds to the default core frequency with AutoBoost disabled. For AMD, we determined this value by comparing the *performance level automatic* with the manually configured performance level. For Intel, the default frequency is derived by evaluating and comparing different min-max core frequency ranges. *Min EDP* is the frequency that minimizes the *Energy Delay Product* (EDP) [34], while *Min Energy* minimizes the energy consumption. *Max Perf* is the frequency that maximizes performance. The *Pareto front* represents a set of optimal solutions where no solution can be improved without compromising another objective. In this context, the Pareto front helps to identify trade-offs between performance and energy consumption. The energy characterization presents the multi-objective energy-performance distribution of different core-frequency settings, using the default frequency as a baseline. We utilized two kernels from the SYCL-Bench suite [35], `matrix_mul` (size: 5000×5000) and `mersenne_twister` (size: 524288), and executed both kernels across all supported frequencies on three different GPUs (AMD MI100, Intel Max 1100, and NVIDIA V100S). For all three GPUs, it is clear that scaling the frequency can lead to improvements in at least one of the objectives: energy and performance. For instance, in the case of the NVIDIA V100, with Min EDP as the energy target, `matrix_mul` achieves an $8\,\%$ energy savings with a performance loss of about $1\,\%$, while `mersenne_twister` achieves a $10\,\%$ energy saving and a $15\,\%$ performance gain. The relationship between time and energy depends on whether a kernel is memory or compute bound. Compute-bound kernels benefit more from higher core frequencies compared to memory-bound kernels.

B. Frequency Tuning Granularity

Based on the granularity of frequency tuning, there are two common approaches used in related work. **Coarse-grained** approaches [28], [36], [37] optimize the entire application by setting a single frequency for the whole program. Although this method can be easily implemented by job schedulers and is effective for single-kernel applications, it becomes inefficient for more complex applications with multiple kernel executions. In fact, each kernel may have different energy characterizations, requiring more fine-grained energy tuning, such as setting a distinct frequency for each individual kernel. In **fine-grained** approaches [29], [38], [39], instead, each kernel in the program can be assigned an optimal frequency. This approach covers the shortcomings of the coarse-grained approach but typically requires the application to manage frequency scaling.

III. Motivation and Overview

Our work is motivated by the need for more precise frequency tuning and the desire to minimize the overhead associated with frequency changes.

A. Frequency Scaling Overhead

This Section shows that changing the frequency imposes an overhead, which can influence the effectiveness of energy-tuning granularity. To show this, we designed a benchmark called `fsbench1` consisting of two kernel types with different energy characteristics. The first kernel is a `matrix_mul` with 1024×1024 elements and an optimal frequency of

1110 MHz, and the second is a `median_filter` with a 2048×2048 input size and an optimal frequency of 645 MHz. We first run the `matrix_mul` in a loop and then run the `median_filter` in another loop right after the first loop. The optimal frequency is selected based on the *Min Energy* target in this case. We implement frequency scaling through two methods: coarse-grained and fine-grained, utilizing ROCm for AMD, LevelZero for Intel, and NVML for NVIDIA.

Figure 2 shows the time taken for each approach when we repeat the `fsbench1` benchmark with a loop count of 512 (512 invocations for the first kernel, followed by 512 for the second kernel). The coarse-grained approach sets a frequency once at the beginning of the program, whereas the fine-grained approach sets the frequency 1024 times, once before each kernel invocation.

If we do not consider the overhead of frequency changing (the hatched area), the fine-grained technique will outperform the coarse-grained method for all three GPUs, as it sets an optimal frequency for each kernel. However, considering the high overhead of changing frequency (the hatched area), the fine-grained approach becomes even slower for Intel and NVIDIA compared to the coarse-grained method. For NVIDIA, which experiences a higher overhead for frequency changes, each frequency change with NVML takes almost 0.6 ms, making the fine-grained approach around 47 % slower than coarse-grained, overall.

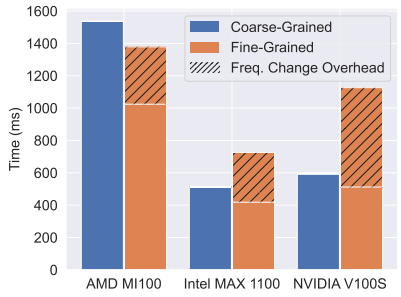


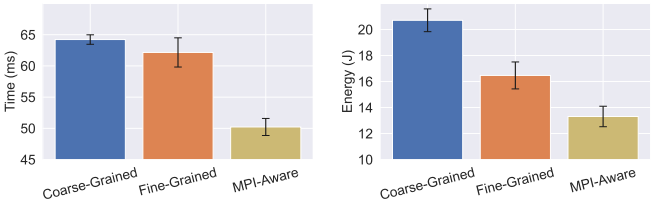
Fig. 2: Frequency scaling overhead for coarse- and fine-grained approaches on `fsbench1`.

As we showed, changing the frequency with vendor-provided tools incurs an overhead in the range of a fraction of a millisecond. This is a problem: first, if we have an application consisting of many lightweight kernels, each with a runtime of microseconds or a fraction of a millisecond. In this case, the fine-grained method would have a very high overhead, and this overhead may become higher than the benefit we get from frequency scaling. Second, when multiple consecutive kernels in the application require the same optimal frequency, changing the frequency for each individual kernel is inefficient.

B. Towards a Phase-based MPI-aware Approach

Considering the challenges of coarse- and fine-grained frequency tuning methods, there is a need for a third approach that addresses their shortcomings. This paper proposes a **phased-based** approach that aims to minimize the frequency changes in fine-grained approaches while maintaining their energy efficiency.

The first key insight is that we can *group tasks with similar energy characteristics* into a single phase. After identifying the application phases, we will set an optimized frequency for each phase. The second key insight is that it is possible to *hide the frequency change latency by overlapping it with communication*, similar to typical computation-communication overlap optimization. This strategy is effective in multi-GPU and multi-node applications, where a significant amount of time is spent communicating between GPUs and nodes. We have experimentally evaluated the optimization potential of overlapping frequency changes while the GPUs are communicating.



(a) Execution time (b) Energy consumption

Fig. 3: Execution time and energy consumption of the coarse-grained, fine-grained, and MPI-aware approach for `fsbench2` on 4 Intel Max 1100 GPUs.

To assess this, we designed a simple benchmark called `fsbench2` consisting of 2 kernels (with different energy characteristics) and 2 MPI collective operations. We call a kernel after each MPI collective operation in the following order: communication 1 (`MPI_Bcast`), kernel 1 (`geometric_mean`), communication 2 (`MPI_Reduce`), kernel 2 (`vector_addition`). Intuitively, we will have 2 phases, assigning one phase to each kernel. Using the techniques described later in Section V, we overlap the frequency change overhead with MPI communications. Figure 3 shows the time taken and energy consumption for the coarse-grained, fine-grained (which is equivalent to a phase-based technique with only 2 phases), and the MPI-overlapped (which is phase-based with frequency change overhead overlapped with MPI communications) across 4 GPUs. The results indicate that such overlapping can significantly enhance performance, reducing the time and energy consumption of the fine-grained approach by 20 % and 19 %, respectively, compared to the same method without overlapping.

C. Overview

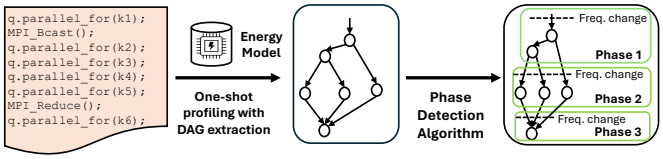


Fig. 4: The proposed Phase-based approach overview.

Figure 4 provides an overview of our phase-based approach. The application code comprises multiple computational kernels in the form of SYCL `parallel_for` and MPI communications. First, we do a one-shot profiling of the application to collect the required time and energy data at the default frequency. During this step, we also extract a DAG representing the dependencies between tasks (kernel executions). In order to save time and avoid profiling the application at all possible frequencies, we utilize an energy prediction model [29] that predicts the speedup and normalized energy across all frequencies. Once the DAG is built, we apply our phase detection algorithm to identify the application's energy phases. Then, we set a desired frequency for each phase to optimize energy and performance.

While the proposed methodology is based on SYCL and MPI, it remains very generic and can be easily applied to other programming models. Furthermore, the energy prediction model can be replaced with any energy model capable of predicting speedup and energy values. In Sections IV and V, we present the theoretical framework for automatic phase detection on single- and multi-device systems, separately.

IV. Phase Detection on Single Device

The benefits of employing a fine-grained approach may be constrained by the frequency change overhead (see Fig. 2). Our methodology for a single device, shown in Figure 5, introduces a phase detection algorithm that aims to minimize the number of frequency changes while preserving the energy efficiency of the fine-grained approach. This approach begins with the input of a SYCL code targeting, e.g., a single GPU. ❶ In the first step, after conducting a one-shot profiling and predicting the energy using the model, we generate a *Directed Acyclic Graph* (DAG) that represents task (kernel execution) dependencies enriched with their time and energy characteristics. ❷ In the second step, the phase detection algorithm is applied to the DAG to detect the phases. This is achieved by identifying groups of tasks in the application that require a similar frequency. A frequency is then set for all tasks within the identified phase.

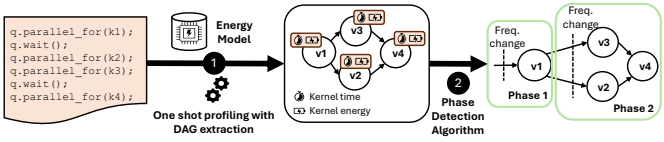


Fig. 5: Single-device energy-aware DAG modeling.

In the rest of this section, we present the theoretical framework used to identify the energy phases: the energy-aware DAG model, the algorithm based on dynamic programming, and the cost function.

A. Energy-aware DAG Model

We model the execution of a parallel program using a computational DAG, where tasks are kernels executed on the device, represented by graph vertices, and inter-task dependencies are captured by graph edges. The DAG is extended with energy and runtime information extracted from the one-shot profiling step. Formally, we define an *Energy-annotated DAG*, which is 4-tuple $C_e = (V, E, t, e)$ of finite sets, such that: V is the set of vertices representing the tasks of the profiled application; E is the set of edges defining data dependencies between tasks; t and e are the time and energy functions defined as $t : V \times F \rightarrow R$ and $e : V \times F \rightarrow R$, where F is the set of available frequencies.

B. Phase Detection Algorithms

In order to have a comparison and choose the most efficient algorithm for phase detection, we tested three different algorithms: *Dynamic programming* (Dp), *Greedy* (Gr), and *Clustering* (Cl). The first step in common between all the three algorithms is to apply a topological ordering on C_e, producing a sequence of tasks $T = (v_0, \ldots, v_{|T|-1})$, where $|T| = |V|$. In this work, we adopt the `in_order` queue property defined by SYCL; that is, the kernels execute in the same order in which they are submitted to the queue.

Dynamic programming (Dp). This approach is commonly used in optimization problems, where the aim is to find the best solution among many possible ones. Given T the topological order of C_e, the dynamic programming solution can be expressed using the following recurrence relation:

$$opt(i,j,k)=\begin{cases} Cost(i,j) & \text{if } i{=}j \vee k{=}0 \\ \min\begin{cases} \min\limits_{i\le l<j}\{opt(i,l,k-1)+opt(l+1,j,k-1)\} \\ Cost(i,j) \end{cases} & \text{otherwise} \end{cases} \tag{1}$$

where $opt(i, j, k)$ represents the cost of the optimal partitioning for tasks $(v_i, \ldots, v_j)$ with at most $2^k - 1$ splits, and $Cost(i, j)$ defines the cost of having $(v_i, \ldots, v_j)$ in the same phase. In iteration k, the algorithm evaluates whether to proceed with further subdivisions based on the optimal partitions found in iteration $k - 1$. The terms $opt(i, l, k - 1)$ and $opt(l+1, j, k-1)$ reflect the costs of splitting the interval $[i, j]$ at position l, where each subproblem has been solved in iteration $k - 1$.

The time complexity of Dp is $O(|T|^4)$, as it requires four nested loops on the number of tasks ($|T|$). However, as the number of tasks grows, a time complexity of $O(|T|^4)$ can limit the applicability of the phase-based approach in practice.

Greedy (Gr). The greedy approach (Gr) reduces the time complexity to $O(|T|^2)$ while providing solutions that are not necessarily optimal. The greedy algorithm examines each task in the list, one by one, and decides whether to add it to the current phase or the next phase. This decision is based on the cost function $Cost(i, j)$, as defined in Section IV-C. The algorithm can be implemented with a loop iterating over tasks in T, where in each iteration, the cost function is applied with time complexity of $O(|T|)$, resulting in an overall time complexity of $O(|T|^2)$. The greedy approach constructs phases by processing tasks in the order they appear in the list. Although this approach allows for on-the-fly phase detection, it can limit the potential to select optimal phases.

Clustering (Cl). The clustering approach offers greater flexibility by allowing tasks to be grouped in a more efficient way. The key idea is to prioritize the phases that provide the highest potential for energy savings and then determine, using the cost function $Cost(i, j)$, whether it is better to merge that phase with the next one. Initially, each task composes one phase (or cluster), creating a set of clusters that can be combined. At each iteration, the algorithm chooses the cluster that yields the maximum energy savings. Once a promising cluster is identified, the cost function helps decide whether combining it with the adjacent cluster is beneficial. If combining is advantageous, the clusters are merged; otherwise, the cluster is moved to the final list of phases. This process continues until no further clusters can be merged. The algorithm iterates over all tasks, applying the selection and cost function at every step, resulting in the same computational complexity as the Greedy algorithm. By prioritizing energy savings rather than following a strict task order, the clustering approach has the potential to detect more efficient phases.

Section VI provides an in-depth comparison of Gr and Cl approaches against the optimal approach (Dp) on a wide range of multi-kernel applications with different characteristics.

C. Cost Function

The phase detection algorithm leverages the cost function $Cost(i, j)$ to compute the cost of having a phase with tasks $(vi, \ldots, vj)$. The cost associated with a phase considers four factors: the number of times r_k that the task v_k is repeated (in one or more nested loops); the frequency change overhead ovh_e; the execution time of each task $t(v_k, f_{opt})$; and determining the optimal frequency f_{opt} for the entire phase by selecting the frequency that minimizes overall energy consumption across all included tasks. Our methodology provides a cost function that encapsulates all these aspects:

$$Cost(i, j) = ovh_e + \sum_{k=i}^{j} r_k \cdot t(v_k, f_{\text{opt}}) \tag{2}$$

Traditionally, loops in the source code are unrolled in the task DAG. This approach significantly impacts the complexity of algorithms operating on the DAG, leading to increased computational overhead and memory usage. Real-world applications may contain several tasks repeated multiple times inside loops. For instance, CloverLeaf, the real-world application used in this paper, encountered over 7000 kernel invocations. To address this challenge, instead of expanding loops in the task DAG, we opt to incorporate loop information as metadata attached to the nodes during the profiling step. For this, we count the number of kernel invocations for each kernel in the profiling phase. This strategy allows us to retain loop semantics without inflating the size of the DAG, providing a more efficient solution for managing loop-heavy scenarios. In order to handle loop information, for a task v_i, we define $L_i = \{\text{loops } \ell : v_i \text{ occurs in } \ell\}$. The repetition factor $reps_i(\ell)$ of a task v_i in a loop $\ell \in L_i$ is the number of times the task v_i is executed during the loop ℓ. The number of times that the task v_k is repeated is defined as $r_k = \prod_{\ell \in L_k} reps_k(\ell)$.

The overhead ovh_e considers two aspects: first, the overhead associated with a single frequency change (either time or energy, depending on the target metric), defined as ϵ, and second, the number of times the frequency change must be executed if it occurs within a loop ($reps_i$). Formally, the overhead of frequency changes for a phase composed of the tasks $(v_i, \ldots, v_j)$ is defined as follows:

$$ovh_e = \prod_{\ell \in L_i} reps_i(\ell) \cdot \epsilon \tag{3}$$

The optimal frequency f_{opt} that minimizes the energy consumption of the tasks $(v_i, \ldots, v_j)$ is computed as follows:

$$f_{opt} = \underset{f_t \in \{f_i, \ldots, f_j\}}{\arg\min} \sum_{k=i}^{j} (e(v_k, f_t) - e(v_k, f_k)) \cdot r_k \tag{4}$$

where $\{f_i, \ldots, f_j\}$ are the optimal frequencies that minimize the energy, respectively, for tasks $(v_i, \ldots, v_j)$. The optimal frequency selection for tasks $(v_i, \ldots, v_j)$ can be generalized to support other target metrics such as Max Perf or Min EDP.

V. Phase Detection on Multi-Node

Thus far, we have demonstrated how to detect phases for single-device applications. However, in multi-GPU and multi-node applications, communication also needs to be taken into account. MPI communications can introduce additional data dependencies to the DAG. On the other hand, we have the opportunity to hide the frequency change overhead within the communication time.

Figure 6 shows the phase detection algorithm adapted for MPI applications. ❶ Similar to single-device applications, the first step is to generate a task DAG called C_e, enriched with profiled information and energy predictions. ❷ The C_e is further augmented with MPI-related information, specifically MPI synchronization nodes with annotated runtimes to generate a C_{mpi} graph. ❸ The phase detection algorithm is then applied to the DAG to detect phases. The algorithm remains the same as that used for single-device applications but operates on a C_{mpi} DAG rather than a C_e DAG.

The rest of the section describes how the C_e DAG and the cost function, designed for the single-device context, are extended to handle multi-node/device scenarios.

A. MPI-aware DAG Modeling

By incorporating *MPI_Sync* nodes, the energy-aware DAG is enhanced as an energy- and MPI-aware DAG, considering the communication times in multi-device systems. With MPI blocking functions, the next scheduled task should wait for data from other devices to proceed. The integration of *MPI_Sync* nodes represents these waiting times, which can be used by the phase detection algorithm to perform MPI communication simultaneously with the frequency change. Formally, we extend our methodology with an MPI-aware DAG defined as a 4-tuple $C_{mpi} = (V', E', t', e')$, of finite sets such that: V' is the set of vertices representing the tasks of the profiled application with the *MPI_Sync* point; E' is the set

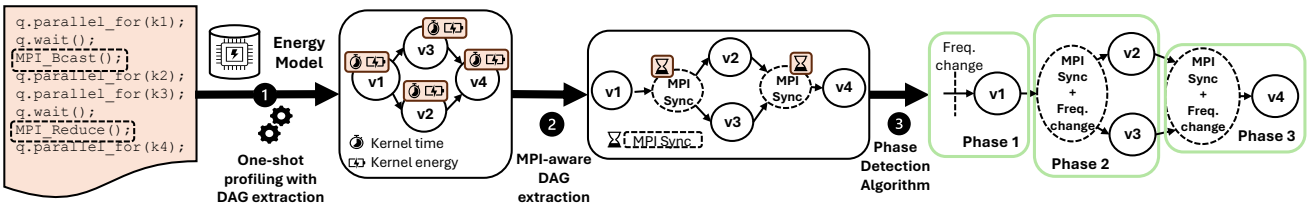


Fig. 6: MPI-aware phase detection algorithm.

of edges defining data dependencies between tasks, including the dependencies introduced by MPI communications; t' and e' are, respectively, the time and energy functions defined in the same way as C_e DAG. For t', the C_{mpi} DAG encapsulates the time spent in the new *MPI_Sync* node, while for e', the energy for the *MPI_Sync* node is always 0 since we are only interested in the communication time that can be exploited to hide the frequency change.

B. MPI-aware Overhead

The cost function defined for C_e is extended to C_{mpi} in order to consider, in the overhead, the time saved by hiding the frequency change overhead during MPI blocking communications. Formally, Eq. 3 is extended as follows:

$$ovh_{mpi} = \prod_{\ell \in L_i} reps_i(\ell) \cdot (\epsilon - \textit{MPI\_Sync}) \tag{5}$$

where the *MPI_Sync* value for *MPI_Sync* node represents the time or energy spent in MPI blocking communication, which is 0 for the nodes representing the tasks. When *MPI_Sync* $> \epsilon$, the overhead (ovh_{mpi}) is considered 0 since the cost of frequency changes can be fully overlapped with the communication process.

C. MPI Collective Communications

In MPI blocking communications, all the involved processes wait until the operation they are performing is completed before allowing the program to proceed. Synchronization occurs internally within the blocking calls. Therefore, it is impossible to overlap these communications with the calls to frequency change APIs. In this case, we need to change these blocking calls to non-blocking and then overlap them with the frequency change function calls. Listing 1 illustrates an example of such a case in which we replace the `MPI_Bcast` with `MPI_Ibcast` and a corresponding `MPI_Wait` in Listing 2.

```
MPI_Bcast();
Freq_change();

q.parallel_for(kernel);
```

Listing 1: Frequency change without overhead hiding.

```
MPI_Ibcast();
Freq_change();
MPI_Wait();
q.parallel_for(kernel);
```

Listing 2: Frequency change with overhead hiding.

D. Stencil Communications

Stencil communications are common patterns in many MPI applications, where each process communicates with its neighbor processes in different dimensions. Here, we demonstrate that our method for overlapping communication and frequency change also applies to stencils. Listings 3 and 4 provide an example of a one-dimensional stencil in which we overlap communication with the call to change the device frequency.

```
MPI_Sendrecv();

Freq_change();

q.parallel_for(kernel);
```

Listing 3: Frequency change example in stencils.

```
MPI_Irecv();
MPI_Isend();
Freq_change();
MPI_Waitall();
q.parallel_for(kernel);
```

Listing 4: Frequency change overhead hiding in stencils.

The proposed overlapping technique can be applied to all MPI communication operations, whether they are blocking or non-blocking. If the communications are already non-blocking, taking into account the data dependency between the communication and the following kernel, we can overlap the frequency modification time with the data transfer. The only case where the current method is not applicable is when the non-blocking MPI communications overlap with kernel execution. In this case, since the kernel is already executing during the MPI communication, we cannot modify the frequency of the GPU as it impacts the currently running kernel.

VI. Phase-detection Algorithms Evaluation

This section evaluates the Gr and Cl algorithms in comparison to the optimal solution Dp. The algorithms are evaluated by comparing the energy savings achieved by their solutions. We compare the energy saved by Gr and Cl to the energy savings of the optimal solution (Dp).

In order to have diverse scenarios and evaluate applications with diverse characteristics, we selected twelve single-kernel benchmarks from SYCL-Bench [35], each with different characteristics and energy requirements. By combining these kernels in various ways, we developed several multi-kernel test applications (with up to 1000 kernel calls), each exhibiting distinct characteristics, allowing us to test the algorithms in a comprehensive manner. Considering that phase selection algorithms must consider the kernels' runtime, repetitions of kernels in loops, and optimal frequency, we classified our test applications into three distinct classes ($R1$, $R2$, $R3$) to encompass a broad range of scenarios observed in real-world applications.

$R1$: Multi-kernel applications where kernels have similar runtimes but different energy requirements with optimal frequencies ranging from 135 MHz to 1597 MHz. In this context, we ensure that no single kernel dominates in execution time to

provide insight into how well the algorithms perform in such balanced scenarios.

$R2$: In contrast to the first class, some kernels have significantly longer runtimes than others. This setup tests how effectively the algorithms handle unbalanced workloads, where certain kernels disproportionately affect overall energy consumption.

$R3$: The third class consists of test applications containing different numbers of kernels repeated within loops. These loops introduce complexity by adding repeated kernel invocations, potentially impacting the application's energy characteristics and runtime. This class assesses the ability of algorithms to optimize energy savings in scenarios where the application features more complex control flows and loop patterns.

In addition to these three classes, we define $E1$ and $E2$ as the edge border cases. $E1$ consists of tasks where the optimal frequency of task v_i differs from v_{i+1}, covering a scenario with varying energy behaviors in consecutive tasks. Unlike for $E2$, there is a drastic frequency change every four or eight tasks.

Table I summarizes the accuracy results of Gr and Cl in percentage compared to the Dp approach, which is the optimal solution, when minimizing *MIN_ENERGY* or *MIN_EDP*. The reported numbers are the average accuracy across all tested applications in each class. For both algorithms, in the $R1$-$R3$ test cases, we save $90\,\%$ to $100\,\%$ of the energy saved by the optimal algorithm. The clustering method consistently achieves greater energy savings than the Greedy approach, suggesting that an online Greedy solution that identifies phases while scheduling tasks may overlook certain optimization opportunities. Examining the edge cases, we observe that both algorithms achieve $83\,\%$ of the energy saved by the optimal solution in the E_1 scenario. In contrast, with E_2 increasing the similarity between consecutive tasks, the energy savings come closer to the optimal solution.

In the rest of the paper, we employ the Clustering phase detection algorithm, which offers high accuracy near the optimal solution and has lower time complexity compared to the Dynamic Programming algorithm.

TABLE I: Phase detection algorithms accuracy in percentage.

	Greedy (G_r)		**Clustering** (C_l)	
App. Class	*MIN_ENERGY*	*MIN_EDP*	*MIN_ENERGY*	*MIN_EDP*
R1	95.1	100	96.5	100
R2	92.0	100	93.2	100
R3	97.3	100	98	100
E1	83	100	83	100
E2	94.1	100	99	100

VII. Implementation

The theoretical framework is translated into a concrete implementation where energy profiling is tailored to device-specific APIs, tasks are extracted from SYCL kernel executions, and MPI communications are specifically handled.

A. Profiling Time and Energy

The first step in identifying the application phases is to analyze and profile the application and extract the required features. The profiling information comprises timing data for kernels, the number of times each kernel is invoked, MPI communications, and energy characterization of the kernels. For MPI profiling, we recorded the time spent on each collective or point-to-point operation. The time and energy profiling of each kernel is performed using the SYnergy API and model [29] along with the SYCL events.

Our methodology involves one-shot profiling at the default device frequency, which gathers all the required information. By utilizing the model, which requires only the LLVM IR of code as input, we can automatically predict time and energy values for all supported frequencies for each device, thereby eliminating the need to execute the application multiple times. In our experiments, the profiling process involved running each application five times at the default frequency to gather reliable time and energy metrics. For all applications, including both single- and multi-GPU applications, this step took approximately $7\,\mathrm{minutes}$.

Although the phase-based approach has been implemented using SYCL, it is decoupled from the programming model, energy prediction model, and energy/time profiler used. In fact, energy profiling can be done with other available libraries, and the model can be replaced with others that predict time and energy values [27], [40]–[42] or can be entirely replaced by a profiling step that runs the application using all available frequencies on each device.

B. Task DAG Creation with SYCL

We implemented our DAG approach on top of SYCL. By default, a SYCL queue executes kernel functions based on dependency information. A SYCL program specifies the data needed to execute a particular kernel, including access modes and memory types. The SYCL runtime ensures that kernels are executed in an order that ensures correctness by building a DAG of tasks at runtime. Generally, a DAG does not specify the order of execution for tasks but only establishes partial ordering constraints represented as edges. However, SYCL queues may operate in an in-order manner, where the schedule follows the same order of submissions to the queue. This limitation simplifies the analysis and is also used by related work [29].

To efficiently generate the task DAG of the applications, we used the oneAPI graph SYCL extension `sycl_ext_oneapi_graph` [30], which decouples command submission from command execution, allowing us to extract task order and dependencies during one-shot profiling. The profiling and modeling data also include the optimal frequency for each kernel, timings, loop information, and MPI communication times. The outcome of this step is a task DAG that represents the data dependencies between kernels and includes additional metadata to be used for phase detection.

C. Phase Detection

After creating the task DAG, we execute the phase detection algorithm described in Section IV. This algorithm takes the DAG as input and detects the phases, specifying where we need to set the frequency. To further clarify the phase detection process, Table II shows the simplified structure of some of the single-GPU applications used in our experiments. As shown, each application consists of multiple kernels, some of which are iterated inside the loops. On the right-hand side of the table, the *Min Energy* indicates the frequency selected for each kernel according to the *Min Energy* target, while *Runtime* represents the ratio of that kernel's runtime to the total runtime of all kernels, considering that each kernel may be called multiple times in the loops. The blue horizontal lines separate the phases detected by our algorithm, and the *Phase* value is the frequency chosen for all the kernels in that phase.

TABLE II: Single-GPU applications profile and detected phases on NVIDIA V100S.

	Code structure	Phase (MHz)	Min Energy Freq. (MHz)	Runtime (%)
ace	`for n in num_iters:`			
	`calculateForce()`	P1 1080	982	21.05 %
	`allenCahn()`		1080	55.52 %
	`boundCondPhi()`		772	1.11 %
	`thermalEquation()`	P2 202	202	16.06 %
	`boundCondU()`		630	1.11 %
	`swapGrid()`		532	5.15 %
aop	`generatePaths()`	P1 960	172	0.34 %
	`prepareSvd()`		960	96.09 %
	`for n in num_iters:`			
	`partialBeta()`	P2 202	202	2.38 %
	`finalBeta()`		150	0.05 %
	`updateCashflow()`		217	1.09 %
	`partialSums()`	P3 157	157	0.04 %
	`finalSum()`		682	0.01 %
mnist	`for n in num_iters:` `for i in train:`			
	`fwPass()`	P1 240	240	32.77 %
	`err()`		225	2.30 %
	`bwPass()`	P2 520	520	50.37 %
	`labeling()`		240	6.37 %
	`for i in test:`	P3 225		
	`fwPass()`		202	8.19 %

When specifying the phases, our phase-detection algorithm follows three primary objectives determined within its cost function (Equation 2). First, according to the cost function, the algorithm groups the kernels that have the *Min Energy* within a specific close range together and assigns the frequency of the kernel with the longest runtime to all kernels in that group: In `ace`: `calculateForce()`, `allenCahn()`, and `boundCondPhi()` are grouped as Phase1 (P1), and they got the frequency of 1080 MHz which is related to `allenCahn()`, having the highest runtime percentage among the others. Second, based on the cost function, it aims to group the kernels invoked within a loop into the same phase to prevent multiple frequency changes during the loop execution. In `aop`: `partialBeta()`, `finalBeta()`, and `updateCashflow()` are in a loop consecutively, and despite having different energy requirements, they are grouped into one phase. Third, it tries to unify phases with shorter runtimes with either the preceding or subsequent phase, which has a longer execution time. In `mnist`: kernels `bwPass()` and `labeling()` are grouped in one phase despite having different *Min Energy* frequencies. Given that the first kernel takes 50.37 % of the time and the second only 6.37 %, after assessing the cost of frequency change in the cost function, the algorithm maintains the frequency for the second kernel without alteration.

Our phase detection algorithm follows the same logic when considering MPI communications. By assessing the cost of frequency change and considering the MPI communication time, it enables the overlap of communication with the overhead of changing the frequency as in Section V.

D. Frequency Scaling with the SYnergy API

After detecting the phases, we need to set the desired frequency for each phase. For this purpose, we used the SYnergy API [29], which is designed to facilitate portable frequency scaling on heterogeneous systems, encapsulating vendor-specific libraries such as NVML, ROCm, and Level Zero. Programmers can apply coarse-grained frequency scaling by specifying the frequency for the entire application or use a fine-grained approach to adjust the frequency of a specific kernel. Our phase-based approach uses the SYnergy API to set the frequency for each phase.

VIII. Experimental Evaluation

In this section, we present the results of the experiments on single and multi-GPU/node. In all experiments, the **fine-grained** and **coarse-grained** approaches refer to two of the state-of-the-art methods, [29] and [43], respectively. The energy target for all experiments is *Min Energy*.

A. Experimental Setup

The experimental setup is summarized in Table III. We used 4 different machines: two single-node single-GPU systems with NVIDIA and AMD GPUs, a single-node system featuring 4 Intel GPUs from the Intel® Tiber™ AI Cloud platform, and a 4-node cluster equipped with 4 NVIDIA GPUs per node.

For the single-GPU experiments, we used five benchmarks from the HeCBench SYCL benchmark suite [44]: `ace`, `aop`, `srad`, `metropolis`, and `mnist`, and evaluated them with the largest available input sizes. Multi-GPU experiments include two real-world applications: CloverLeaf [45], a compressible Euler equations solver on a Cartesian grid, and miniWeather [46], which is from the domain of weather-like flows simulation. The applications are fed with the maximum possible input sizes for each configuration (weak scaling). We utilized the PowerCap interface [47] for host energy measurements.

TABLE III: Overview of the machines used in the experiments.

Machine	Nodes	GPUs per Node	CPU	SYCL Version	MPI Version	Energy Interface
A	1	1× NVIDIA V100S	2× Intel Xeon Gold 5218	Intel DPC++ 5 Feb 2024	NA	NVML
B	1	1× AMD MI100	2× AMD EPYC 7313	Intel DPC++ 20 June 2023	NA	ROCm
C	1	4× Intel Max 1100	2× Intel Xeon Platinum 8480	oneAPI DPC++ v2024.0.2	IntelMPI v2021.11.0	Level Zero
D	4	4× NVIDIA A30	2× Intel Xeon Gold 6338	oneAPI DPC++ v2024.0.2	IntelMPI v2021.11.0	NVML

TABLE IV: Aggregated absolute energy consumption values (in Joules) of host and device in single-node benchmarks running on NVIDIA V100S (Mach. A), AMD MI100 (Mach. B), and Intel Max 1100 (Mach. C) GPUs.

	srad			ace			mt.polis			aop			mnist		
Approach	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C
Coarse-Grained	236.57	1155.40	313.84	8358.34	374.26	1260.34	144.04	-	245.94	218.93	436.68	232.94	41.89	-	142.29
Fine-Grained	738.50	1159.58	312.98	6883.43	375.35	924.57	124.69	-	249.51	266.56	430.32	250.30	1680.30	-	228.12
Phase-Based	182.10	955.81	278.63	5881.66	292.08	870.49	104.26	-	217.94	195.14	406.58	210.88	41.22	-	137.39

B. Single-GPU Performance

Figure 7 illustrates the performance of the fine-grained and phase-based approaches compared to the coarse-grained approach for single-GPU benchmarks. For each benchmark, we fed the maximum possible input that each GPU could handle.

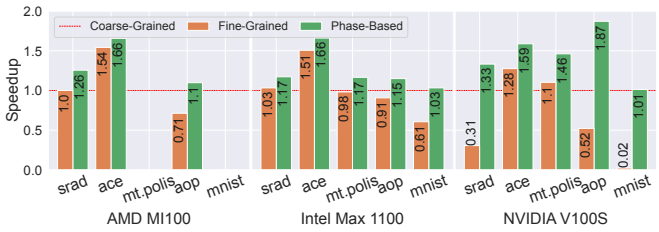


Fig. 7: Single-GPU benchmarks performance (higher is better).

In this figure, the phase-based method consistently outperforms the other two methods across all three GPUs, proving that the proposed phase-based approach effectively reduces the overhead associated with frequency changes in the fine-grained approach. The largest gap between fine-grained and phase-based performance across various GPUs is observed in the NVIDIA GPU, where the phase-based method enhances the performance of fine-grained for `srad`, `aop`, and `mnist` by $4.3\times$, $3.6\times$, and $50.5\times$, while for Intel GPU, these are $1.13\times$, $1.26\times$, and $1.68\times$, respectively. This is mainly related to the inefficiency of the fine-grained approach for applications comprising several lightweight kernels invoked multiple times within loops, such as `mnist`. In this case, the gain of changing the frequency for each kernel is not enough to overcome the overhead. This results in a significant slowdown of the fine-grained method, particularly on NVIDIA systems, mainly due to the high overhead of NVML compared to ROCm and Level Zero when changing the frequency (as illustrated in Figure 2). Also, the runtimes of each kernel differ across various GPUs. We have fed different input sizes to each GPU based on its available memory, resulting in varying speedup ratios when comparing the performance of different GPUs. Notably, we were not able to run `metropolis` and `mnist` on AMD due to the high memory requirements of these benchmarks.

C. Host and Device Energy Analysis

Figure 8 shows the corresponding energy consumption of the three approaches on both the device and host while running benchmarks on various GPUs. The corresponding total absolute energy consumption values (host + device) are provided in Table IV. The host energy refers to the energy consumed by the CPU for the entire application.

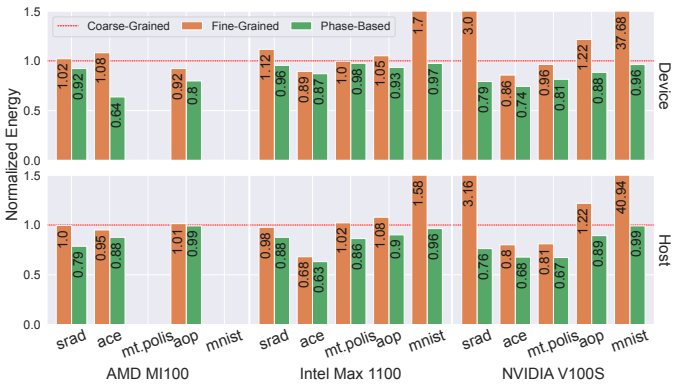


Fig. 8: Normalized energy consumption on the device (above) and host (below) for single-GPU benchmarks (lower is better).

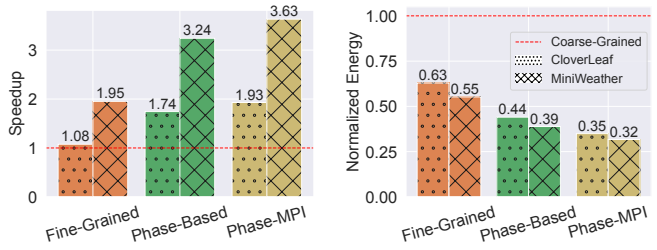
Similar to the performance in Figure 7, here in Figure 8, a similar consistent trend is observed: on the device and host, for all three GPUs, the phase-based approach exhibits lower energy consumption compared to both coarse-grained and fine-grained methods across all benchmarks. In particular, for the device energy of `aop` and `mnist`, the fine-grained approach consumes $1.12\times$ and $1.75\times$ more energy on Intel and $1.39\times$ and $39.25\times$ more energy on NVIDIA compared to the phase-based method. This increase is mainly attributed to the high number of GPU frequency changes occurring in these two benchmarks using the fine-grained approach. Regarding host energy, the decline in energy consumption of these benchmarks using the phase-based technique is linked to the reduction in their runtime. Our approach shortens the

overall execution time of the benchmarks, leading to a decrease in both device and host (CPU) energy consumption.

D. Real-world MPI Applications

In this section, we compare our phase-based MPI-aware approach with both coarse- and fine-grained methods on multiple GPUs using two real-world MPI+SYCL applications: CloverLeaf and miniWeather.

Figure 9 shows the speedup and normalized energy (relative to the coarse-grained approach) for these two applications on a single node equipped with four Intel GPUs (*Mach. C*). The corresponding absolute energy consumption values are in Table V. The *Phase-Based* represents the phase-based approach without considering the MPI communications, while the *Phase-MPI* represents the phase-based approach with MPI communications overlapping with the frequency scaling overhead. In Figure 9a, the phase-based approach consistently outperforms both the coarse- and fine-grained methods for the two applications. For CloverLeaf and miniWeather, it shows better performance $1.61\times$ and $1.66\times$ than the fine-grained approach, respectively. In addition, phase-MPI further improves the performance of the phase-based by $1.10\times$ and $1.12\times$ for CloverLeaf and miniWeather, respectively. Overall, the phase-MPI method improves the performance by $1.79\times$ and $1.86\times$ for CloverLeaf and miniWeather compared to the state-of-the-art fine-grained method.



(a) Performance (higher is better) (b) Energy (lower is better)

Fig. 9: The normalized performance and energy consumption of CloverLeaf and miniWeather on 4 Intel Max 1100 GPUs.

TABLE V: Absolute energy consumption values (in kJ) for CloverLeaf and miniWeather using different frequency scaling methods on 4 Intel Max 1100 GPUs.

	Coarse-Grained	**Fine-Grained**	**Phase-Based**	**Phase-MPI**
CloverLeaf	125.32	78.86	55.18	43.81
miniWeather	38.42	21.22	14.93	12.18

In Figure 9b, the phase-based approach is more energy efficient than coarse- and fine-grained methods for the two applications. It achieves an energy savings of 30 % relative to the fine-grained method for both CloverLeaf and miniWeather. In phase-MPI, there is an additional improvement: compared to the phase-based method, phase-MPI reduces energy consumption by 21 % for CloverLeaf and 19 % for miniWeather. Overall, phase-MPI reduces the energy consumption of the state-of-the-art fine-grained approach by 45 % and 43 % for CloverLeaf and miniWeather, respectively. As shown, our proposed phase-based method improves both energy efficiency and performance in real-world applications, each consisting of multiple kernels with varying energy requirements and execution times. Notably, miniWeather has 12 different kernel types, with 1940 kernel calls in total, and CloverLeaf consists of 37 different kernels with 7854 kernel invocations in total.

E. Energy Scalability

Figure 10 shows the energy scaling of miniWeather up to 16 NVIDIA A30 GPUs using a weak scaling approach. Both the performance and energy scaling results align with our previous findings in this study. At any number of GPUs, the coarse-grained is always slower than the fine-grained, and they are both slower than our MPI-aware phase-based approach. This holds also true for energy efficiency, with our phase-based MPI-aware method proving to be more energy efficient than the other two approaches. For example, with 16 GPUs, our method saves energy by 35% and achieves a $1.45\times$ higher performance compared to the state-of-the-art fine-grained method.

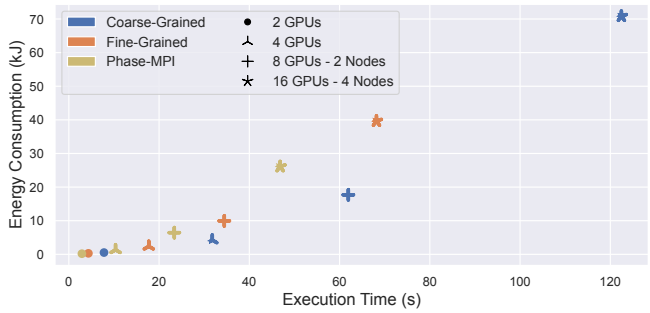


Fig. 10: Energy scalability of miniWeather on 2-, 4-, 8- and 16 GPUs using different frequency scaling methods.

IX. Related Work

As CPU-GPU heterogeneous architectures are now broadly used in exascale computing, they bring high computing capability but also consume significant power and energy. In fact, power and energy consumption are considered primary issues in large-scale HPC [48], [49]. Many researchers have proposed approaches for tackling this issue [50]–[52].

DVFS-based technique DVFS is one of the widely used techniques to enhance energy efficiency in HPC. Numerous studies have investigated different DVFS-based mechanisms from different perspectives. For example, some researchers focused on analyzing the impact of DVFS on multi-objective optimization using machine learning [53], [54]. Some researchers focused on combining frequency scaling with other energy-efficient techniques, such as power capping [38]. Furthermore, DVFS techniques can be implemented on different hardware and heterogeneous processors [28], [29], [54]–[57]. SYnergy [29] is another frequency scaling-based approach

that targets heterogeneous hardware to achieve fine-grained energy saving. Countdown [57] is a runtime library on CPUs, which can automatically reduce power consumption by adding DVFS capabilities into standard MPI libraries. Countdown supports both fine and coarse granularity MPI to inject power management calls. *However, existing DVFS-based methods do not consider MPI applications on CPU-GPU heterogeneous architecture.*

Phase-aware DVFS technique From a granularity perspective, many researchers started to investigate frequency scaling based on the phases rather than the fine- or coarse-grained approaches [58]–[63]. Among them, Qiu et al. [62] presented a three-phase DVFS algorithm that achieves higher energy saving by clustering task slacks via task graph unrolling. Booth et al. [63] proposed a phase-based voltage and frequency scaling that chooses the phases according to the Segments of code with unique performance and power attributes using Hidden Markov Models. *However, existing phase-aware methods are not implemented on modern heterogeneous architecture.*

DVFS in MPI applications Energy efficiency with the DVFS-based techniques has also been applied to distributed memory context [57], [64]–[66]. Rountree et al. [67] used linear programming to resolve the NP-complete problem of when to change the frequency in an MPI program. Zamani et al. [68] aimed to apply DVFS to a specific application on multiple GPUs. They used the algorithmic knowledge from the application to predict slack times and do frequency scaling on both CPU and GPUs. Endrei et al. [41] performed frequency scaling and proposed a statistical model to find the best tradeoff between performance and energy efficiency in MPI applications. *None of the related work, however, tried to hide the overhead of frequency scaling, and none of them provided a generic approach applicable to any application type.*

TABLE VI: Comparison against the state of the art.

Paper	Granularity	Frequency Scaling	Heterogeneity	MPI-aware
Qiu et al. [62]	Phase-based	✓	×	×
Booth et al. [63]	Phase-based	✓	×	×
Countdown [57]	Fine-&Coarse-grained	✓	×	✓
Wang et al. [43]	Coarse-grained	✓	×	×
SYnergy [29]	Fine-grained	✓	✓	×
Our work	Phase-based	✓	✓	✓

Table VI compares our work with the state of the art. While the state-of-the-art coarse-grained approach [43] is not well-tuned for multi-kernel and real-world applications, SYnergy [29], which is the state-of-the-art heterogeneous fine-grained approach, suffers from unnecessary frequency changes for all the kernels in the application. To the best of our knowledge, our work is the first to apply energy optimization to MPI programs by overlapping communication with frequency change.

X. Conclusion

In this paper, we highlighted the overhead of frequency scaling across different GPUs and proposed a phase-based frequency scaling technique for heterogeneous systems that minimizes this overhead while preserving the energy efficiency and performance of fine-grained frequency scaling approaches. Our method identifies energy phases using the task DAG enriched with profiling data. We developed a phase-detection algorithm that identifies these phases and then sets an optimal frequency for each phase. We proposed a novel approach for MPI programs to further hide frequency change overhead by overlapping it with MPI communications. Our approach reduces overhead, increasing energy efficiency and performance compared to state-of-the-art methods. It improves real-world application performance by $1.45\times$ and saves $35\,\%$ of energy on 16 GPUs compared to the state-of-the-art fine-grained method.

XI. Acknowledgment

This research has been funded by the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 956137 (LIGATE project).

We thank Intel for providing access to the Intel® Tiber™ AI Cloud platform.

We thank Ahmad Afsahi and Amirhossein Sojoodi from Queen's University, Canada, for providing access to their HPC resources and assisting with the multi-node experiments.

References

[1] S. Ibrahim, T.-D. Phan, A. Carpen-Amarie, H.-E. Chihoub, D. Moise, and G. Antoniu, "Governing energy consumption in Hadoop through CPU frequency scaling: An analysis," *Future Generation Computer Systems*, vol. 54, pp. 219–232, 2016.

[2] T.-D. Phan, S. Ibrahim, A. C. Zhou, G. Aupy, and G. Antoniu, "Energy-driven straggler mitigation in MapReduce," in *Euro-Par 2017: Parallel Processing: 23rd International Conference on Parallel and Distributed Computing*. Springer, 2017, pp. 385–398.

[3] A. Tundo, M. Mobilio, S. Ilager, I. Brandić, E. Bartocci, and L. Mariani, "An energy-aware approach to design self-adaptive AI-based applications on the edge," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 281–293.

[4] M. D. de Assunção, L. Lefevre, and F. Rossigneux, "On the impact of advance reservations for energy-aware provisioning of bare-metal cloud resources," in *2016 12th International Conference on Network and Service Management (CNSM)*. IEEE, 2016, pp. 238–242.

[5] M. D. d. Assunção and L. Lefèvre, "Bare-metal reservation for cloud: an analysis of the trade off between reactivity and energy efficiency," *Cluster Computing*, vol. 21, pp. 1289–1300, 2018.

[6] Q. Liang, W. A. Hanafy, N. Bashir, D. Irwin, and P. Shenoy, "Energy time fairness: Balancing fair allocation of energy and time for GPU workloads," in *2023 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2023, pp. 53–66.

[7] W. A. Hanafy, R. Bostandoost, N. Bashir, D. Irwin, M. Hajiesmaili, and P. Shenoy, "The war of the efficiencies: Understanding the tension between carbon and energy optimization," in *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, 2023, pp. 1–7.

[8] A. C. Zhou, T.-D. Phan, S. Ibrahim, and B. He, "Energy-efficient speculative execution using advanced reservation for heterogeneous clusters," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–10.

[9] T. De Matteis and G. Mencagli, "Proactive elasticity and energy awareness in data stream processing," *Journal of Systems and Software*, vol. 127, pp. 302–319, 2017.

[10] S. Ilager, A. N. Toosi, M. R. Jha, I. Brandic, and R. Buyya, "A data-driven analysis of a cloud data center: statistical characterization of workload, energy and temperature," in *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*, 2023, pp. 1–10.

[11] A. Gougeon, B. Camus, and A.-C. Orgerie, "Optimizing green energy consumption of fog computing architectures," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 75–82.

[12] Y. Taleb, S. Ibrahim, G. Antoniu, and T. Cortes, "Characterizing performance and energy-efficiency of the ramcloud storage system," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 1488–1498.

[13] A.-C. Orgerie, M. D. d. Assuncao, and L. Lefevre, "A survey on techniques for improving the energy efficiency of large-scale distributed systems," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, pp. 1–31, 2014.

[14] C. Wang, M. Zink, and D. Irwin, "Energy-agile design for parallel HPC applications," *Sustainable Computing: Informatics and Systems*, vol. 19, pp. 123–134, 2018.

[15] I. Dagli, A. Cieslewicz, J. McClurg, and M. E. Belviranli, "Axonn: energy-aware execution of neural network inference on multi-accelerator heterogeneous SoCs," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1069–1074.

[16] A. Souza, N. Bashir, J. Murillo, W. Hanafy, Q. Liang, D. Irwin, and P. Shenoy, "Ecovisor: A virtual energy system for carbon-efficient applications," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 252–265.

[17] A. Souza, S. Jasoria, B. Chakrabarty, A. Bridgwater, A. Lundberg, F. Skogh, A. Ali-Eldin, D. Irwin, and P. Shenoy, "Casper: Carbon-aware scheduling and provisioning for distributed web services," in *Proceedings of the 14th International Green and Sustainable Computing Conference*, 2023, pp. 67–73.

[18] F. Almeida, M. D. Assunção, J. Barbosa, V. Blanco, I. Brandic, G. Da Costa, M. F. Dolz, A. C. Elster, M. Jarus, H. D. Karatza *et al.*, "Energy monitoring as an essential building block towards sustainable ultrascale systems," *Sustainable Computing: Informatics and Systems*, vol. 17, pp. 27–42, 2018.

[19] R. Watanabe, M. Kondo, M. Imai, H. Nakamura, and T. Nanya, "Task scheduling under performance constraints for reducing the energy consumption of the GALS multi-processor SoC," in *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2007, pp. 1–6.

[20] M. Kondo, H. Sasaki, and H. Nakamura, "Improving fairness, throughput and energy-efficiency on a chip multiprocessor through DVFS," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 31–38, 2007.

[21] E. Arima, M. Kang, I. Saba, J. Weidendorfer, C. Trinitis, and M. Schulz, "Optimizing hardware resource partitioning and job allocations on modern GPUs under power caps," in *Workshop Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–10.

[22] B. Rountree, D. H. Ahn, B. R. De Supinski, D. K. Lowenthal, and M. Schulz, "Beyond dvfs: A first look at performance under a hardware-enforced power bound," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2012, pp. 947–953.

[23] J. Krzywda, A. Ali-Eldin, T. E. Carlson, P.-O. Östberg, and E. Elmroth, "Power-performance tradeoffs in data center servers: Dvfs, cpu pinning, horizontal, and vertical scaling," *Future Generation Computer Systems*, vol. 81, pp. 114–128, 2018.

[24] J. Eastep, S. Sylvester, C. Cantalupo, B. Geltz, F. Ardanaz, A. Al-Rawi, K. Livingston, F. Keceli, M. Maiterth, and S. Jana, "Global extensible open power manager: A vehicle for HPC community collaboration on co-designed energy management solutions," in *High Performance Computing: 32nd International Conference, ISC High Performance*. Springer, 2017, pp. 394–412.

[25] J. Corbalán, L. Alonso, J. Aneas, and L. Brochard, "Energy optimization and analysis with EAR," in *IEEE International Conference on Cluster Computing, CLUSTER 2020, Kobe, Japan, September 14-17, 2020*. IEEE, 2020, pp. 464–472. [Online]. Available: https://doi.org/10.1109/CLUSTER49012.2020.00067

[26] D. C. Wilson, F. Acun, S. Jana, F. Ardanaz, J. M. Eastep, I. C. Paschalidis, and A. K. Coskun, "An end-to-end HPC framework for dynamic power objectives," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, Denver, CO, USA, November 12-17, 2023*. ACM, 2023, pp. 1801–1811.

[27] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas, "GPGPU power modeling for multi-domain voltage-frequency scaling," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 789–800.

[28] K. Kraljic, D. Kerger, and M. Schulz, "Energy efficient frequency scaling on GPUs in heterogeneous HPC systems," in *International Conference on Architecture of Computing Systems*. Springer, 2022, pp. 3–16.

[29] K. Fan, M. D'Antonio, L. Carpentieri, B. Cosenza, F. Ficarelli, and D. Cesarini, "SYnergy: Fine-grained energy-efficient heterogeneous computing for scalable energy saving," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–13.

[30] E. Crawford, P. Reble, B. Tracy, and J. Miller, "Towards deferred execution of a sycl command graph," in *Proceedings of the 2023 International Workshop on OpenCL*, 2023, pp. 1–2.

[31] "AMD ROCm™ documentation," https://rocm.docs.amd.com/en/latest/, (Accessed on 03/21/2024).

[32] "Level Zero — Intel® software for general purpose GPU capabilities documentation," https://dgpu-docs.intel.com/technologies/level-zero.html, (Accessed on 03/21/2024).

[33] "NVML API reference guide :: GPU deployment and management documentation," https://docs.nvidia.com/deploy/nvml-api/index.html, (Accessed on 03/21/2024).

[34] J. H. Laros III, K. Pedretti, S. M. Kelly, W. Shu, K. Ferreira, J. Van Dyke, C. Vaughan, J. H. Laros III, K. Pedretti, S. M. Kelly *et al.*, "Energy delay product," *Energy-Efficient High Performance Computing: Measurement and Tuning*, pp. 51–55, 2013.

[35] L. Crisci, L. Carpentieri, P. Thoman, A. Alpay, V. Heuveline, and B. Cosenza, "SYCL-Bench 2020: Benchmarking SYCL 2020 on AMD, Intel, and NVIDIA GPUs," 2024.

[36] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux utility for resource management," in *Workshop on job scheduling strategies for parallel processing*. Springer, 2003, pp. 44–60.

[37] H. Zhang and H. Hoffmann, "PoDD: power-capping dependent distributed applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–23.

[38] M. Hao, W. Zhang, Y. Wang, G. Lu, F. Wang, and A. V. Vasilakos, "Fine-grained powercap allocation for power-constrained systems based on multi-objective machine learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1789–1801, 2020.

[39] M. Sourouri, E. B. Raknes, N. Reissmann, J. Langguth, D. Hackenberg, R. Schöne, and P. G. Kjeldsberg, "Towards fine-grained dynamic tuning of HPC applications on modern multi-core architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[40] M. Walker, S. Bischoff, S. Diestelhorst, G. Merrett, and B. Al-Hashimi, "Hardware-validated cpu performance and energy modelling," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2018, pp. 44–53.

[41] M. Endrei, C. Jin, M. N. Dinh, D. Abramson, H. Poxon, L. DeRose, and B. R. de Supinski, "Energy efficiency modeling of parallel applications," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 212–224.

[42] F. Antici, A. Bartolini, Z. Kiziltan, O. Babaoglu, and Y. Kodama, "Mcbound: An online framework to characterize and classify memory/compute-bound hpc jobs."

[43] Q. Wang, X. Mei, H. Liu, Y.-W. Leung, Z. Li, and X. Chu, "Energy-aware non-preemptive task scheduling with deadline constraint in DVFS-enabled heterogeneous clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4083–4099, 2022.

[44] Z. Jin and J. S. Vetter, "A benchmark suite for improving performance portability of the SYCL programming model," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 325–327.

[45] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Accelerating hydrocodes with OpenACC, openCL and CUDA," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 465–471.

[46] M. R. Norman, "miniweather," [Computer Software] https://doi.org/10.11578/dc.20201001.88, 2020.

[47] "Power capping framework — The Linux kernel documentation," https://www.kernel.org/doc/html/next/power/powercap/powercap.html, (Accessed on 03/28/2024).

[48] I. Saba, E. Arima, D. Liu, and M. Schulz, "Orchestrated co-scheduling, resource partitioning, and power capping on CPU-GPU heterogeneous systems via machine learning," in *Architecture of Computing Systems - 35th International Conference, ARCS*, ser. Lecture Notes in Computer Science, vol. 13642. Springer, 2022, pp. 51–67.

[49] O. Sarood, A. Langer, A. Gupta, and L. V. Kalé, "Maximizing throughput of overprovisioned HPC data centers under a strict power budget," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, 2014*, pp. 807–818.

[50] M. S. Z. Nine, T. Kosar, M. F. Bulut, and J. Hwang, "GreenNFV: Energy-efficient network function virtualization with service level agreement constraints," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–12.

[51] H. Ribic and Y. D. Liu, "Aequitas: Coordinated energy management across parallel applications," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12.

[52] N. Gholkar, F. Mueller, and B. Rountree, "Power tuning HPC jobs on power-constrained systems," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, 2016, pp. 179–191.

[53] K. Fan, B. Cosenza, and B. Juurlink, "Predictable GPUs frequency scaling for energy and performance," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[54] H. Khaleghzadeh, M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on heterogeneous HPC platforms for performance and energy through workload distribution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 543–560, 2020.

[55] M. A. H. Monil, M. E. Belviranli, S. Lee, J. S. Vetter, and A. D. Malony, "Mephesto: Modeling energy-performance in heterogeneous SoCs and their trade-offs," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 413–425.

[56] J. Corbalan, O. Vidal, L. Alonso, and J. Aneas, "Explicit uncore frequency scaling for energy optimisation policies with EAR in Intel architectures," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 572–581.

[57] D. Cesarini, A. Bartolini, P. Bonfà, C. Cavazzoni, and L. Benini, "Countdown: a run-time library for performance-neutral energy saving in MPI applications," *IEEE Transactions on Computers*, vol. 70, no. 5, pp. 682–695, 2020.

[58] E. S. A. Lozano and A. Gerstlauer, "Learning-based phase-aware multi-core CPU workload forecasting," *ACM transactions on design automation of electronic systems*, vol. 28, no. 2, pp. 1–27, 2022.

[59] J. Scheipl, A. Raoofy, M. Ott, and J. Weidendorfer, "Phase-aware system-side sampling for hpc," in *Proceedings of the 20th ACM International Conference on Computing Frontiers*, 2023, pp. 220–221.

[60] K. Malkowski, P. Raghavan, M. Kandemir, and M. J. Irwin, "Phase-aware adaptive hardware selection for power-efficient scientific computations," in *Proceedings of the 2007 international symposium on Low power electronics and design*, 2007, pp. 403–406.

[61] J. Kim, S. Yoo, and C.-M. Kyung, "Program phase-aware dynamic voltage scaling under variable computational workload and memory stall environment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 1, pp. 110–123, 2010.

[62] M. Qiu, Z. Ming, J. Li, S. Liu, B. Wang, and Z. Lu, "Three-phase time-aware energy minimization with DVFS and unrolling for chip multiprocessors," *Journal of Systems Architecture*, vol. 58, no. 10, pp. 439–445, 2012.

[63] J. D. Booth, J. Kotra, H. Zhao, M. Kandemir, and P. Raghavan, "Phase detection with hidden Markov models for DVFS on many-core processors," in *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 2015, pp. 185–195.

[64] V. W. Freeh, F. Pan, N. Kappiah, D. K. Lowenthal, and R. Springer, "Exploring the energy-time tradeoff in MPI programs on a power-scalable cluster," in *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2005, pp. 10–pp.

[65] C. Lively, X. Wu, V. Taylor, S. Moore, H.-C. Chang, C.-Y. Su, and K. Cameron, "Power-aware predictive models of hybrid (mpi/openmp) scientific applications on multicore systems," *Computer Science-Research and Development*, vol. 27, pp. 245–253, 2012.

[66] D. Li, D. S. Nikolopoulos, K. Cameron, B. R. de Supinski, and M. Schulz, "Power-aware mpi task aggregation prediction for high-end computing systems," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–12.

[67] 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 *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, pp. 1–9.

[68] H. Zamani, L. Bhuyan, J. Chen, and Z. Chen, "GreenMD: Energy-efficient matrix decomposition on heterogeneous multi-GPU systems," *ACM Transactions on Parallel Computing*, vol. 10, no. 2, pp. 1–23, 2023.