Energy of Communication: Measurement and Modelling for Parallel Hybrid Programs on Heterogeneous Hybrid Servers

Hafiz Adnan Niaz

The thesis is submitted to University College Dublin
in fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Science

School of Computer Science and Informatics

Head of School: Prof. Neil Hurley

Research Supervisor: Assoc. Prof. Alexey Lastovetsky

Research Co-Supervisor: Asst. Prof. Ravi Reddy Manumachu

September 2025

Acknowledgements

First and foremost, I am profoundly grateful to Almighty God for granting me the opportunity, determination, and strength to undertake and complete this research. His endless mercy, guidance, and blessings have been a constant source of support throughout this journey. It is through His grace that I have been able to contribute in a meaningful way, with the hope of benefiting His creation.

I extend my deepest gratitude to my supervisor, Dr. Alexey Lastovetsky, for welcoming me into the Heterogeneous Computing Laboratory (HCL) and for his steadfast mentorship throughout my doctoral studies. His continuous support, profound expertise, and exceptional guidance in high-performance heterogeneous computing have significantly shaped my academic and research development. His rigorous standards, critical insights, and commitment to excellence consistently challenged me to think more deeply and strive for higher quality in my work. It has been both inspiring and a true privilege to conduct my research under his supervision.

I would also like to express my sincere appreciation to my co-supervisor, Dr. Ravi Reddy Manumachu, for his unwavering support, thoughtful guidance, and constant encouragement throughout my PhD journey. His patience, approachability, and genuine dedication to my progress made a meaningful difference during both the rewarding and challenging phases of my research. His constructive feedback, technical insights, and engaging discussions substantially strengthened the quality and clarity of my work. I am truly grateful for his mentorship, which fostered my confidence, independence, and growth as a researcher.

I am also thankful to my Doctoral Study Panel members, Dr. Mel Ó Cinnéide and Dr. Liam Murphy, for their insightful guidance and valuable suggestions throughout my research.

I would like to extend my heartfelt thanks to my dear friend, Dr. Maryam Gillani, for her unwavering support, and the countless moments of joy we shared over good food and endless tea breaks. Her kindness, encouragement, and friendship have been a source of comfort and strength throughout this journey.

I am grateful to all my colleagues at the Heterogeneous Computing Laboratory for their support and insightful discussions. My sincere thanks also go to the School of Computer Science at University College Dublin for providing continuous support throughout my doctoral studies.

I am deeply indebted to my family for their unwavering love and support. In particular,

I extend my heartfelt thanks to my beloved mother and father. Their constant belief in my abilities and their continuous encouragement have been a source of strength and motivation throughout my academic journey. Thank you for always being there for me, offering guidance, nurturing my dreams, and standing by me every step of the way. Your sacrifices, love, and dedication have made this achievement possible, and for that, I am forever grateful.

Finally, I would like to thank my brothers and my sister for their unconditional love and constant support. Their presence in my life has been a continual source of joy, comfort, and motivation. I am truly grateful for their encouragement throughout every step of this academic journey.

I would also like to acknowledge the Heterogeneous Computing Laboratory (HCL) at University College Dublin for providing the computational resources and collaborative environment that significantly contributed to the results presented in this thesis.

To my family.

Abstract

The energy consumption of Information and Communications Technology (ICT) accounted for 7% of global electricity usage and is forecast to be around the average of the best-case and expected scenarios (7% and 21%) by 2030. This trend makes the energy efficiency of digital platforms a new grand technological challenge.

Modern digital servers in clouds, data centres, and High-Performance Computing (HPC) environments have become highly heterogeneous, with multicore CPUs integrated with accelerators such as graphics processing units (GPUs) to address this challenge. While innovations in energy-efficient hardware primarily drive the response, developing energy-efficient software that leverages application-level energy optimization techniques on these servers is also essential.

The energy consumption of applications executing on such servers is primarily due to computations running in parallel on the computing devices and data transfers between them. Historically, the main focus of energy researchers has been to minimize and model the energy consumed by computations, due to the widespread belief that energy consumption from data transfers is insignificant.

Therefore, a significant gap remains in measurement and modelling the energy consumption of inter-device data transfers. This gap is unexplored because no research has tackled this topic. As a result, no method exists to accurately measure the energy of data transfer or a model to predict this energy.

This work aims to fill and address this crucial gap and demonstrates that the energy consumption of inter-device data transfers in data-intensive heterogeneous hybrid applications can be substantial and comparable to energy consumption by computations. Furthermore, there is a good opportunity for energy savings due to the non-linear nature of the energy of data transfers with data transfer size.

First, we comprehensively study the energy consumption of data transfer between a host CPU and an accelerator GPU on the heterogeneous hybrid servers using the three mainstream energy measurement methods: (a) System-level physical measurements based on external power meters (ground-truth), (b) Measurements using on-chip power sensors, and (c) Energy predictive models.

We develop a novel methodology to accurately measure the energy consumption of data transfer between a pair of devices using external physical power meters considered the *ground-truth*. The ground-truth method is accurate but prohibitively costly and time-consuming. On the other hand, the on-chip sensors in Intel multicore CPU processors are inaccurate, and the Nvidia GPU sensors do not capture data transfer activity. Due to practical limitations with the first two methods, this research focuses on the third approach.

Energy predictive models employing performance events have emerged as a promising alternative to other mainstream methods for developing software power meters used in runtime energy profiling, since they are cost-effective and represent the most accurate runtime method for the measurement of energy consumption of applications.

We propose a novel methodology that employs a fast selection procedure for selecting a small subset of performance events to develop a software power meter based on linear energy predictive model for predicting the energy consumption of a data transfer activity. We then design and develop reliable software power meters based on linear energy predictive models that leverage selected performance events to accurately predict the energy consumption of a single activity, whether it is computation or data transfer.

The software power meters are evaluated using three parallel scientific programs running on a heterogeneous hybrid server for each single activity. The results show that the software power meters achieve high accuracy with an average prediction error of 1% for computation activity, and 6% for data transfer activity across all three parallel programmes.

In the next phase, this work introduces the fundamental properties for a set of software power meters: *Concurrency and Orthogonality*, which are essential for achieving accurate runtime energy profiling of parallel hybrid programs on heterogeneous hybrid servers. We present a methodology for developing concurrent and orthogonal software power meters that provide accurate runtime estimation of energy consumption associ-

ated with independently powered parallel computation and communication activities.

We apply this methodology to develop concurrent and orthogonal software power meters for three heterogeneous hybrid servers that consist of Intel multicore CPUs and Nvidia GPUs from different generations. These concurrent and orthogonal software power meters implement system-level linear energy predictive models that employ disjoint sets of performance events. Furthermore, the accuracy and efficiency of the proposed concurrent and orthogonal software power meters are evaluated by estimating the dynamic energy consumption of independently powered parallel computation and communication activities in three parallel hybrid programmes using our three heterogeneous hybrid servers.

The results show an average prediction error of just 2.5% for dynamic energy consumption by these concurrent and orthogonal software power meters across our servers. This confirms their effectiveness and reliability, and validates the practicality of the proposed concurrent and orthogonal software power meters for accurate runtime energy profiling on heterogeneous hybrid servers, addressing a long-standing gap in the field.

List of Publications

- (Published) H. A. Niaz, R. R. Manumachu and A. Lastovetsky, **"Accurate and Reliable Energy Measurement and Modelling of Data Transfer Between CPU and GPU in Parallel Applications on Heterogeneous Hybrid Platforms"** in IEEE TRANSACTIONS ON COMPUTERS(TC), vol. 74, no. 3, pp. 1011-1024, IEEE, 2024. DOI: 10.1109/TC.2024.3504262. [1]

- (Published) H. A. Niaz, R. R. Manumachu and A. Lastovetsky, **"Concurrent and Orthogonal Software Power Meters for Accurate Runtime Energy Profiling of Parallel Hybrid Programs on Heterogeneous Hybrid Servers"** in IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS (TPDS), vol. 37, no. 02, pp. 322-339, Feb. 2026, DOI: 10.1109/TPDS.2025.3637511. [2]

Contents

List of Figures

List of Tables

Statement of Original Authorship

I hereby certify that the submitted work is my own work, was completed while registered as a candidate for the degree stated on the Title Page, and I have not obtained a degree elsewhere on the basis of the research presented in this submitted work.

Chapter 1

Introduction

1.1 Background and Motivation

The energy consumption of Information and Communications Technology (ICT) accounted for 7% of the global electricity usage in 2020 and is forecast to be around the average of the best-case and expected scenarios (7% and 21%) by 2030 [3], [4]. This trend makes the energy efficiency of digital platforms a major technological challenge. Modern digital platforms such as supercomputers, clusters, clouds and data centres have become highly heterogeneous, with multicore CPUs integrated with accelerators such as graphics processing units (GPUs) to address this challenge.

While innovations in energy-efficient hardware play an important role, developing energy-efficient software employing application-level energy optimization techniques is equally essential. Parallel computations on computing devices and data transfers between computing devices are the dominant contributors to the energy consumption of hybrid applications executing on modern platforms.

Historically, most research has focused on developing solutions for energy modelling and optimization of computations ([5], [6], [7], [8], [9]), based on the assumption that the energy consumption of data transfers is negligible. As a result, the energy consumption of device data transfers has remained largely unexplored. Consequently, a significant gap exists in the accurate measurement and modelling of energy consumption for data transfers between computing devices.

At present, there is no reliable method to accurately measure the energy of data transfer, nor a model that can predict this energy between computing devices. This research addresses this critical gap.

This work demonstrates that the energy consumption of data transfers in data intensive heterogeneous hybrid applications can be substantial and comparable to the energy consumption of computations. Furthermore, due to the non-linear relationship between data transfer size and energy consumption, there exists significant potential for optimiz-

ing and reducing the energy cost of data transfers.

In this thesis, the term *data transfer* refers to the transfer of data between a pair of computing devices, typically a CPU and an accelerator, in a heterogeneous hybrid platform. Other related terms used in the literature include data movement and communication. For clarity and consistency, this thesis uses the term *data transfer* throughout.

1.2 Dominant Energy Efficiency Approaches

There are two main approaches that respond to the energy efficiency challenges in computing:

1. **Hardware approach**
2. **Software approach**

The first approach deals with energy-efficient hardware devices at the transistor or gate level, aiming to produce electronic devices that consume as little power as possible.

The second approach deals with the development of energy-efficient software. Depending on its scope and solution, this category can be further divided into the following two approaches:

- System-level
- Application-level

1. **System-level**

   The system-level approach to energy optimization of computing tries to optimize the execution environment rather than applications running in the environment [10], [11], [12], [13]. The main technique is to switch off or reduce the power level of the components of the system that are not used in the execution of the applications (or just to reduce the power level of the underused component).

   This mainstream approach relies on energy-efficient hardware parameters as decision variables such as Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) to optimize the energy efficiency of the execution of the application. A computing component executing an application typically incurs two forms of power consumption:

   - Static Power
   - Dynamic Power.

Static power, also referred to as idle power, is consumed even when the component is not actively running an application. Dynamic Power Management (DPM) turns off the electronic components or moves them to a low-power state when idle, to reduce energy consumption. Dynamic power is consumed due to the switching activity in the processor's circuits. Dynamic Voltage and Frequency Scaling (DVFS) can increase or reduce the dynamic power a processor consumes by changing the operating frequency.

This system-level approach has been thoroughly studied and has resulted in production-level solutions and tools used in ICT, from large data and supercomputing centers to mobile devices.

2. **Application-level**

The other approach, which is to optimize applications rather than the executing environment, has not attracted similar attention and, as such, is understudied. The application-level solution methods include optimization of applications by using application-level decision variables and aim to optimize the application rather than the executing environment [5], [6], [8], [14], [15] . The main decision variables include the number of threads, number of processors and workload distribution.

The state-of-the-art solutions for optimizing parallel applications on heterogeneous hybrid platforms for energy can be classified into the following categories:

(a) Methods optimizing the applications for only energy using an energy model of computation [16].

(b) Methods optimizing the applications for performance and energy by employing performance and energy models of computation [5], [6], [8].

These methods employ application-specific energy models that are functions of workload size. The models for accelerator software components involving a host CPU and a GPU in the hybrid application are integrated models that model totally and not separately the energy of data transfer between the CPU and accelerator memory, the execution of the accelerator component performing the accelerator code, and the data transfer between the accelerator memory and CPU.

Therefore, the methods do not measure and employ separate energy predictive models for computation and data transfers. Having separate models allows a more fine-grained decomposition of the energy consumed by software components in a heterogeneous hybrid application. Furthermore, this will facilitate the design and development of holistic bi-objective optimization methods optimizing parallel hybrid applications for energy and performance.

As a result, accurate measurement and modelling of inter-device data transfer energy remains an open challenge, particularly for heterogeneous hybrid platforms. This thesis addresses this gap by developing accurate measurement methodologies and runtime software power meters. Very few works try to address the fundamental problem of the optimal design and configuration of software for execution on modern, highly heterogeneous and hybrid platforms [17], [18], [19].

This thesis exclusively addresses this largely overlooked aspect of high-performance computing: the accurate energy measurement of data transfer energy consumption between computing devices, particularly in data-intensive heterogeneous hybrid applications. Accurate measurement of energy consumption during an application execution is key to energy minimization at the application level. Accurately measuring and modelling communication energy is therefore critical for the design of energy-efficient applications.

1.3 Reseach Objective, Problem and Contribution

1.3.1 Research Objective

In computer programs, two fundamental operations dominate execution: *computation and communication*. Communication refers to the transfer of data between devices, while computation involves processing that data. During the execution of data-intensive tasks on modern platforms, large volumes of data must be moved across different devices, incurring significant energy costs.

Consequently, the energy consumed by communication or data transfers has emerged as a critical challenge. As data movement and communication increasingly contribute to both time and energy consumption, it becomes essential to understand and reduce their energy footprint to enable smarter and more sustainable computing. The objective of this research is to develop accurate and reliable methods and software tools to measure and model the energy consumption of data transfers between computing devices in heterogeneous hybrid servers.

1.3.2 Research Questions

This research addresses the following research questions:

- **RQ1:** How to accurately and reliably measure the energy of data transfers between CPU and GPU in a heterogeneous hybrid server?
- **RQ2:** How to accurately, reliably and efficiently measure the energy of data transfer between a CPU and GPU at runtime?

- **RQ3:** How to accurately and efficiently measure energy consumption of both computation and communication activities in a parallel hybrid application?
- **RQ4:** How to accurately and efficiently measure energy consumption of parallel data transfers between a CPU and multiple GPUs in a heterogeneous hybrid server?

These research questions guide the design, development and validation of the methodologies and software power meters presented in this thesis.

1.3.3 Research Problem

This research aims to develop a practical solution for accurately and reliably measuring the energy cost of communication or data transfer between different computing devices for point-to-point and parallel communication within a heterogeneous server. However, there is currently *no state-of-the-art measurement method* available to accurately measure the energy associated with data transfer either offline or at run-time between computing devices in a server. Additionally, there is *no software tool* that can automatically and efficiently measure or predict the energy consumption of data transfers.

For the comprehensive optimization of application energy consumption on heterogeneous server, it is essential to have accurate measurement methods for measuring data transfer energy at both offline and run-time. Without such measurements, it is impossible to comprehensively optimize energy usage at the application level.

This represents a clear and significant open research problem in the field, and through this research, we aim to propose effective and practical solutions to address it.

1.3.3.1 Problem Statement

Based on the discussion above, the core problem addressed in this research is the development of an accurate and efficient run-time software tool, that automatically and accurately predicts the energy consumption of data transfers between different software components of a hybrid parallel application executing across multiple computing devices on a heterogeneous hybrid server.

The eventual ultimate goal is to build an efficient run-time software tool, that can run at run time when the application is running, so the application at run time could get accurate information about the energy cost of different components of an application, including both the computation and communication. In essence, the tool should be capable of measuring the energy consumption of each software component as well as the overall energy footprint of the application.

1.3.4 Research Contribution

The key contributions of this research can be organized into the following main categories:

Measurement methodology contributions:

1. Developed a first detailed methodology for accurately measuring the energy of data transfers using system-level physical measurements from external power meters.
2. Developed the first in-house software library, *Libedm*, for accurate energy measurement of data transfers between computing devices (CPU and GPUs).
3. Demonstrated experimentally that data transfer energy is significant and asymmetric from host-to-device (CPU to GPU) and device-to-host direction (GPU to CPU).
4. Demonstrated experimentally that on-chip sensors are inaccurate for measuring data transfer energy.

Software power meter modelling contributions:

1. Developed comprehensive methodology for selecting performance events for accurate energy predictive models.
2. Developed accurate software power meters based on linear energy predictive models that leverage performance events as predictor variables for measuring data transfer energy.
3. Developed independent, runtime software power meters based on the linear energy predictive models for measuring the dynamic energy consumption of both computation and communication activities separately.

Concurrent and orthogonal software power meter contributions:

1. Introduced the properties of concurrency and orthogonality for software power meters essential for accurate runtime dynamic energy profiling of parallel hybrid programs running on heterogeneous hybrid servers.
2. Developed the comprehensive methodology for constructing concurrent and orthogonal software power meters that enable accurate runtime estimation of dynamic energy consumption associated with independently powered parallel computation and communication activities.
3. Developed concurrent and orthogonal software power meters for three heterogeneous hybrid servers from different generations.

Experimental validation contributions:

1. Validated software power meters using three parallel hybrid applications.
2. Demonstrated accurate runtime energy profiling of independently powered parallel computation and communication activities.
3. Demonstrated the high accuracy and efficiency of software power meters in profiling runtime energy consumption for three parallel hybrid applications.

1.4 Thesis Structure

This thesis is structured as follows:

Chapter 2 introduces the foundational terminology and concepts essential to understanding the work. It also presents an overview of the energy measurement methods and existing energy predictive models developed for estimating the energy consumption of computations. Chapter 3 presents a comprehensive methodology for accurately measuring the energy consumption of data transfers between CPU and GPU using an external physical power meter. It also demonstrates that vendor-provided on-chip power sensors are too inaccurate to be considered an alternative to external power meters. Chapter 4 introduces a fast and systematic approach for selecting a small subset of performance events to develop a software power meter based on a linear energy predictive model that leverages performance events for predicting the energy consumption of a single activity, whether it is computation or data transfer. It also demonstrates the validation of the developed software power meters by applying them to estimate the energy consumption of each single activity in three parallel scientific programmes on a heterogeneous hybrid server. Chapter 5 presents and defines two fundamental critical properties of software power meters *concurrency and orthogonality*, and outlines a detailed methodology for constructing a set of concurrent and orthogonal software power meters that enable accurate runtime estimation of dynamic energy consumption associated with independently powered parallel computation and communication activities. Chapter 5 also describes the validation of the methodology by demonstrating the high accuracy and efficiency of independently powered parallel computation and communication activities in profiling runtime energy consumption for three parallel hybrid programs across three heterogeneous hybrid servers from different generations. Finally, Chapter 6 summarizes the key contributions, discusses the limitations and future research directions, and concludes the thesis.

Chapter 2

Background and Related Work

In this chapter, we present a brief overview of the relevant literature related to our work. Since there is a lack of comprehensive research on energy modelling of data transfer, we focus mainly on energy measurement approaches for computations.

The chapter is organized as follows. Section 2.1introduces fundamental terminology that will be used throughout the thesis. Section 2.2 presents current state-of-the-art techniques for measuring computing energy. Section 2.3 provides an overview of developed energy predictive models for CPU and GPU computations, and also highlights the emerging efforts on energy consumption of data transfers in heterogeneous systems. Finally, Section 2.4 summarizes the key findings and concludes the chapter.

2.1 Energy Efficient Computing Terminology

In this thesis, we study data transfers between a CPU and accelerator GPU in heterogeneous hybrid servers. The term *"Data Transfer"* specifically refers to the movement of data between a pair of computing devices typically a CPU and an accelerator within a heterogeneous hybrid server. When we use the phrase “a pair of computing devices”, we mean a CPU and a GPU. Related terms commonly found in the literature include data movement and communication. Although these terms are often used interchangeably, we consistently use data transfer throughout this work for clarity and consistency.

We define a heterogeneous hybrid server as a system comprising a multicore CPU and one or more accelerators. The computing devices in such servers are the CPU and the accelerators. In this work, we mean a CPU and an accelerator when we refer to *a pair of communicating computing devices*. Some servers have direct links between the accelerators. These servers are out of the scope of this work.

A parallel hybrid program running on a heterogeneous hybrid server comprises several software components (kernels) executing in parallel. *An activity in a software component pertains to either computations within the component or the data transfer between*

a pair of communicating computing devices.

2.1.1 Execution Time

The execution time T_E of a data transfer between a pair of computing devices is measured on the CPU side using the CPU processor clock. The process involves the following steps: querying the processor clock to record the start time, executing the data transfer, querying the processor clock again to record the end time, and then calculating the difference between the start and end times. The execution time of a computation activity is measured in a similar fashion. The execution time is reported in seconds (Sec).

2.1.2 Static Energy

The static energy consumption during the data transfer activity between a CPU and an accelerator is the product of the servers's static or idle power and the execution time of the data transfer activity. It is measured on the CPU side using the HCLWattsUp energy measurement API. HCLWattsUp [20] is an energy measurement API, which provides interface functions that gather the readings from the power meters to determine the average power and energy consumption during the execution of an application.

2.1.3 Dynamic Energy

The dynamic energy consumption of an activity is also measured on the CPU side. The measurement steps differ for methods using a software power meter and ground-truth. We describe the measurement step sequence in the ground-truth method using the HCLWattsUp energy measurement API [20]. The sequence includes starting the energy meter using HCLWattsUp's *start()* function, executing the activity, stopping the energy meter using HCLWattsUp's *stop()* function, and then querying the energy meter for the total energy consumption of the activity using HCLWattsUp's *energy()* function. All the steps in the sequence are invoked from the CPU side.

Consider a program executing only one activity. To calculate the dynamic energy E_D of the activity, the following formula is used:

$$E_D = E_T - (P_S \times T_E) \tag{2.1}$$

In this formula, P_S represents the static power consumption of the server, E_T is the total energy consumption of the server during the activity, and T_E is the duration of the activity in seconds.

For a software power meter, the measurement step sequence includes starting the power meter, executing the activity, stopping the power meter, and then querying the

power meter for dynamic energy consumption. The units used in this work for power and energy consumption are Watts (W) and Joules (J).

2.1.4 Statistical Confidence

In all experiments utilizing the ground-truth method, we ensure the reliability of our measurements by repeating the experiment until the sample mean falls within a 95% confidence interval and achieves a precision (or standard deviation of the mean) of 2.5%. For this purpose, we use the Student's t-test, which assumes that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions using the Shapiro-Wilk Test.

2.1.5 Prediction Error

The prediction error of a software power meter is calculated as follows:

$$\text{Prediction Error (\%)} = \left(\frac{|\text{Actual} - \text{Estimated}|}{\text{Actual}} \right) \times 100 \tag{2.2}$$

where $Actual$ is the value obtained using the ground-truth approach and $Estimated$ is the value given by the software power meter.

2.2 State-of-the-Art Energy Measurement Methods

There are three mainstream energy measurement methods [21], [22], [23] that are widely used and can be employed for determining component-level energy consumption:

1. **System-level Physical Power Measurements Based on External Power Meters (Ground-truth Method)**
2. **Measurements Using On-Chip Power Sensors**
3. **Energy Predictive Models**

2.2.1 System-level Physical Power Measurements Based on External Power Meters (Ground-truth Method)

The first approach considered the *ground-truth* is the most accurate energy measurement method [21]. However, there are many challenges when employing it to measure the energy consumption of applications. First, this approach gives the node's total instantaneous power consumption, which is usually the power reading provided by the external power meter operating at the maximum sampling rate.

Therefore, obtaining the energy consumption solely by the application is not straightforward and challenging due to persistent background noise arising from periodical OS, disk, network book-keeping tasks and fans. In addition, the power meter has a manufacturer-reported accuracy of measurement. The WattsUp Pro power meter employed in this work has a reported accuracy of $\pm$ 1.5% of the actual value, whereas the ANSI standard WT310 Yokogawa boasts an accuracy of $\pm$ 0.1%.

To eliminate any potential interference (noise) from activities unrelated to an application execution, one must employ statistical averaging (for example, using the Student's t-test). However, Fahad et al. [21] explain the trade-offs of energy measurement using external power meters, considered the *ground-truth*. The authors find the ground-truth method very time-consuming, also reported in [8], [24].

Fahad et al. [21] demonstrate that this statistical procedure is highly time-consuming mainly due to slow convergence to a sample mean meeting a required precision (or standard deviation of the mean) of 2.5%, which represents consistency in repeated measurements. A higher precision requirement will further delay the convergence. Second, the approach is unsuitable for runtime energy profiling in platforms lacking power meter hardware and software infrastructure.

Although power meters provide accurate energy measurements, they have some major downsides. First, they are very expensive devices, and deploying them at scale, such as attaching one to every node in a large system is not feasible, making the setup significantly more costly. It is expensive not only in terms of installation costs but also in terms of large execution time to obtain one experimental data point with sufficient statistical confidence.

Based on our experience, acquiring a single experimental data point with adequate statistical confidence and reliability may require several hours. However, the results obtained through this approach are considered as the *Ground-truth* for energy measurement. It is the nature of the method that it is time-consuming because of the need to average the runs [25].

2.2.2 On-chip Power Sensors and Vendor-specific Libraries

Unlike the *ground-truth* method, the second energy measurement method on-chip power sensors enable online energy measurement through vendor-specific programmatic interfaces. There are vendor-specific libraries available for retrieving the power data from these sensors.

However, this approach too has several disadvantages. First, there is a lack of comprehensive professional documentation describing how power or energy consumption is measured. For example, it is unclear whether the on-chip power sensors are based on software energy predictive models or electronic hardware models (employing current

estimates and input voltage). Hence, on-chip sensors function as black boxes, making it difficult to relate the energy consumption of the computing elements containing the sensors with the overall energy consumption of an application.

Second, on-chip power sensors for Intel multicore CPUs have been shown to be inaccurate when profiling CPU computations in heterogeneous hybrid servers [21]. In contrast, the on-chip power sensors for Nvidia GPUs, which utilize the Nvidia Management Library (NVML), have improved significantly in accuracy for profiling GPU computations, particularly in the latest processor generations.

Hackenberg et al. [26] report systematic errors in RAPL energy counters and find that it is inclined towards certain types of workload and can give poor power predictions for others. Fahad et al. [21] present a comparative study of on-chip sensors against the ground truth. They demonstrate that using Intel RAPL in energy optimization may lead to significant energy losses. Experiments with a DGEMM matrix-multiplication hybrid application have shown that using RAPL for building energy profiles for energy optimization methods results in energy losses ranging from 37% to 84% (depending on matrix sizes) in comparison with the accurate ground-truth profiles.

Research works [21], [27] report poor accuracy and anomalies with sensors in older generations of Nvidia GPUs (Nvidia Tesla K20c and K40c). The latest Nvidia GPUs (for example, the A40 GPU processor generation and later) provide accurate on-chip power sensors whose readings can be obtained programmatically using the Nvidia Management Library (NVML) [28] interface. However, these power sensors do not capture any data transfer activity on the GPU side.

While cheap and efficient, this on-chip power sensors approach lacks experimental accuracy along with uncertainties relevant to the measurement information. On-chip energy sensors often produce dynamic energy profiles that diverge significantly from ground-truth measurements. Comprehensive and extensive experiments conducted on a range of widely used CPUs, accelerators, and scientific kernels have revealed that energy profiles constructed using advanced on-chip power sensors often lack qualitative accuracy [19], [21], [29], [30].

2.2.3 Energy Predictive Models

The third approach involves using software energy predictive models that employ Performance Monitoring Counters (PMCs) which are measurable runtime software and hardware activities as predictor variables. PMCs are special-purpose hardware registers in modern processor architectures to store the counts of software and hardware activities and to provide low-level performance analysis and tuning. Performance events and resource utilizations are the principal source of model variables primarily due to their high positive correlation with energy consumption. Performance events, however, are pre-

ferred due to their better prediction accuracy compared to utilization variables.

This method has emerged as only realistic and a promising alternative to using physical power meters for developing energ predictive model for runtime energy profiling compared to other mainstream methods. There are two main reasons for this. First, if implemented correctly, its runtime accuracy can surpass that of other approaches. In fact, a single runtime measurement obtained from other methods (such as power meters and on-chip sensors) can significantly differ from the accurate but expensive statistical ground-truth mean. For instance, the average prediction error of these other methods used on our dedicated HPC servers, after three repetitions, ranged from 30% to 40%.

A software power meter that uses a linear energy predictive model with reliable performance events, defined as those events that yield consistent counts across runs, regardless of environmental noise, will produce the same ground-truth mean value even when environmental noise is present. This indicates that the performance event-based method can achieve the same level of accuracy as the ground-truth method when there is no noise. Additionally, it is likely to be more accurate in noisy conditions because it provides measurements that would align with the ground-truth method under ideal, noise-free circumstances.

In addition, a software power meter is cost-effective since the reliable performance events employed in accurate linear energy predictive models exhibit the same counts from run to run and do not need statistical averaging. Therefore, the time to obtain an experimental data point using this approach is significantly less than the other approaches.

To summarize, energy predictive models employing reliable performance events is the most cost-effective and potentially the most accurate method for the measurement of energy consumption of applications.

2.3 Energy Predictive Models on Modern Computing Platforms

We now survey prominent research works in energy predictive modelling of computations and data transfers for CPUs and GPUs based on performance events. The energy measurement approach based on software energy predictive models employs various measurable runtime performance-related predictor variables. Performance events and resource utilizations are the principal source of model variables primarily due to their high positive correlation with energy consumption. Performance events, however, are preferred due to their better prediction accuracy compared to utilization variables.

2.3.1 Energy Predictive Models of Computations for Multicore CPUs

Software-based energy prediction models rely on various runtime performance-related variables for energy prediction. Among these, performance events and resource utilization serve as key model variables, primarily because of their strong positive high correlation with energy consumption.

The research works such as [31], [32], [33], [34], and [35] focus on resource utilization like CPU, memory, disk, and network usage. On the other hand, works like [19], [36], [37], [38], [39], [40], [41], [42], and [43] are centered around performance events. However, performance events are often preferred over utilization variables due to their good prediction accuracy.

Economou et al. [44] developed a linear power prediction model that uses CPU, disk, and network utilization metrics, along with a performance event containing the off-chip memory access counts. Rivoire et al. [45] evaluated five real-time power models for full systems. Four of these models relied on resource utilization, while the fifth was based on the approach proposed by [44].

Their findings showed that the performance event-based model was the most accurate because it accounted for the key factors influencing dynamic power consumption in the system. Khokhriakhov et al. [15] introduced a qualitative linear dynamic energy model that uses CPU utilization and performance events from the PMC performance monitoring counter group to analyze and explain energy nonproportionality observed in their multicore CPU platforms.

Although research on energy predictive models reports excellent accuracy, they typically report the prediction accuracy of total energy with a very high static power base. In addition, most reported results surveyed in [29] were not reproducible. One cause of inaccuracy of performance event based energy models discovered in 2017 is that many popular performance events are not additive on modern multicore processors [46].

The numbers of non-additive performance events increase with the increase of cores (very few non-additive PMCs are found in the case of single core). Another cause of inaccuracy is the violation of basic laws of energy conservation in these models, including non-zero intercept in dynamic energy models, negative coefficients in additive terms, as well as non-linearity of some advanced models (including machine learning models) [46], [47].

Thus, the accuracy of performance event based dynamic energy models can be improved by removing non-additive performance events from models and enforcing basic energy conservation laws. Applying this technique has significantly improved the accuracy of state-of-the-art models, bringing it to 25-30% [19].

2.3.2 Energy Predictive Models for GPU Computations

Energy predictive modelling for Nvidia GPUs exhibits limited attention compared to the plethora of research for CPU processors. One primary reason is that older generations of Nvidia GPUs (preceding Nvidia A40) are poorly instrumented for runtime energy modelling. Based on our experiments on these GPUs, many key events and metrics overflow for large problem sizes due to restrictive 32-bit integers being dedicated to storing the values. Hence, they were unsuitable for energy predictive modelling of parallel hybrid applications with large problem sizes. However, the latest generation GPUs, such as Nvidia A40, provide better energy instrumentation support to facilitate accurate runtime modelling of energy consumption.

Like the performance events for multicore CPUs, Nvidia GPU processors provide CUPTI events and metrics, which are similar to performance events for multicore CPUs, that are employed in energy predictive models for GPU computations [48], [49], [50]. Abe et al. [51] propose energy models employing performance events for different generations of Nvidia GPUs. Adhinarayanan et al. [52] propose application-specific online statistical regression power models employing performance events.

Guerreiro et al. [53] propose linear energy predictive models for core and memory domains that utilize utilization variables obtained from CUPTI events and metrics on GPU devices. They report an average prediction error ranging from 2.5% to 8% for Tesla K40c GPUs. However, it is important to note that key events overflowed for the problem sizes used in the HPC workloads analyzed in this work with this particular GPU. Wang et al. [54] use CUPTI events and metrics in their machine learning energy predictive model, which is employed in the dynamic optimization of machine learning training workloads for performance and energy efficiency.

They report an average prediction error of approximately 5% for their model. In contrast, our proposed software power meter for GPU computations employs NVML, as we determined it to be accurate within $\pm$ 5% of the actual values. Looking ahead, we plan to explore the development of more accurate performance event-based models for GPU computations in our future work.

2.3.3 Energy Predictive Models for Communications

Gamell et al. [55] propose constant bandwidth energy models for the network and main memory. The energy model estimates the energy consumption of m bytes between a pair of processes via a communication link with a peak theoretical bandwidth of B bytes as $(m/B) \times P_{tdp} \times t$, where P_{tdp} is the theoretical peak power (specified in the datasheet) of the communication link and t is the time of the data transfer.

Khaleghzadeh et al. [56] consider the energy cost of communications between two heterogeneous hybrid servers connected by a Gigabit Ethernet network switch, which

consumes a constant power irrespective of the amount of data transferred. Malik et al. [57] propose an application-specific constant bandwidth model predicting the energy of data transfer between a CPU and GPU for a heterogeneous hybrid matrix multiplication application.

Moreover, existing energy models of data transfer between computing devices are simplistic functions of bandwidth and peak power of the communication link between the computing devices, which fail to capture the complexity of real-world scenarios [55, 57].

2.4 Summary

Parallel computations on the computing devices and data transfers between the computing devices are the dominant contributors to the energy consumption of hybrid applications executing on modern digital servers. Historically, the primary focus of energy researchers has been on modelling and optimizing the energy consumption of computations, largely driven by the widespread belief that the energy consumption due to data transfers is insignificant.

Consequently, the energy consumption of device data transfers remains unexplored, with no prior research addressing this topic. As a result, no method currently exists to accurately measure the energy of data transfer or to predict this energy through a reliable model or software tool.

In this thesis, we comprehensively study the energy consumption of data transfer, and aim to develop a potential solution to measure the energy cost of data transfers between computing devices in the heterogeneous hybrid servers. We demonstrate that the energy consumption of device data transfers in data-intensive heterogeneous hybrid applications can be significant and, comparable to the energy consumed by computations.

Chapter 3

Accurate Energy Measurement of Data Transfers on Heterogeneous Hybrid Servers

3.1 Introduction

There are three mainstream energy measurement methods [21], [58]: (a) System-level Physical Measurements Using External Power Meters, (b) Measurements Using On-chip Power Sensors, and (c) Energy Predictive Models. The first approach considered ***ground truth*** is the most accurate. In this chapter, we comprehensively study the energy consumption of data transfers between devices on heterogeneous hybrid servers using the first two mainstream energy measurement methods: (a) System-level physical measurements using external power meters (*ground-truth*), (b) Measurements using on-chip power sensors.

This chapter is organized as follows. Section 3.2 outlines the experimental setup of the servers used in this work and explain the steps to prevent noise in energy measurements. Section 3.3 presents a first comprehensive methodology that employs external power meters to measure the energy consumption of data transfers on two heterogeneous hybrid servers, along with corresponding results and discussion. Section 3.4 explores energy measurement using on-chip sensors and provides an analysis of their accuracy and limitations. Finally, Section 3.5 summarizes the key findings and concludes the chapter.

3.2 Experimental Setup

We employ three Heterogeneous Hybrid Servers for our experiments and comparative study:

- HCLServer1 has an Intel multicore Haswell CPU consisting of 24 cores, an Nvidia K40c GPU with 64 GB main memory.
- HCLServer2 has a single-socket Intel Skylake multicore CPU consisting of 22 cores, and Nvidia P100 GPU with 96 GB main memory.
- HCLServer3 has a single-socket Icelake multicore CPU consisting of 32 cores, and two Nvidia A40 GPUs with 62 GB main memory.

Detailed specifications of the three heterogeneous hybrid servers, *Haswell k40c GPU server*, *Skylake P100 GPU server*, and *Icelake A40 GPU server* are given in the tables 3.1, 3.2 and 3.3. These nodes are representative of computers used in cloud infrastructures, supercomputers, and heterogeneous computing clusters. A power meter is installed between its input power sockets and the wall A/C outlets.

Table 3.1: Specifications of the heterogeneous hybrid server containing Intel multicore Haswell CPU and Nvidia K40c GPU.

Intel Haswell E5-2670V3	
No. of cores per socket	12
Socket(s)	2
CPU MHz	1200.402
L1d cache, L1i cache	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30720 KB
Total main memory	64 GB DDR4
Memory bandwidth	68 GB/sec
NVIDIA K40c GPU	
No. of processor cores	2880
Total board memory	12 GB GDDR5
L2 cache size	1536 KB
Memory bandwidth	288 GB/sec

Table 3.2: Specifications of the heterogeneous hybrid server containing a single-socket Intel Skylake multicore CPU and an Nvidia P100 GPU.

Intel Xeon Gold 6152	
Socket(s)	1
Cores per socket	22
L1d cache, L1i cache	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30976 KB
Main memory	96 GB
NVIDIA P100 PCIe	
No. of processor cores	3584
Total board memory	12 GB CoWoS HBM2
Memory bandwidth	549 GB/sec

Table 3.3: Specifications of the heterogeneous hybrid server containing a single-socket Icelake multicore CPU and two Nvidia A40 GPUs.

Intel Platinum 8362 Icelake	
No. of cores per socket	32
No. of threads per core	2
Socket(s)	1
L1d cache, L1i cache	1.5 MiB, 1 MiB
L2 cache, L3 cache	40 MiB, 48 MiB
Total main memory	62 GB DDR4-3200
TDP	265 W
NVIDIA A40 GPU	
No. of GPUs	2
No. of Ampere cores	10,752
Total board memory	48 GB GDDR6 (with ECC)
Memory bandwidth	696 GB/sec
TDP	300 W

3.2.1 Steps To Reduce Noise in Measurements

Several steps are taken in measuring the energy consumption to eliminate any potential interference (noise) from components not involved in data transfer between the host CPU and a GPU.

Energy consumption of the data transfer between the host CPU and a GPU may include contributions from SSDs, NIC, and fans. Therefore, we ensure that these contributions are minimal using the steps below:

- The program performing the data transfer between the host CPU and a GPU does not invoke any file I/O functions. We monitor the disk consumption during the program run and ensure that negligible I/O is performed by the program using tools such as *sar* and *iotop*.

- The sizes of the data transferred between the host CPU and a GPU do not exceed the main memory of both the CPU and GPU. Therefore, we ensure that no swapping (paging) occurs during the program run on the CPU side and that there are no failures on the GPU side.

- The program performing the data transfer between the host CPU and a GPU does not invoke any network I/O functions. We monitor the network activity using tools such as *sar* and *atop* and ensure that it is not significant.

Fans are significant contributors to energy consumption. Our hybrid server platform controls fans in zones: a) zone 0: CPU or System fans, b) zone 1: Peripheral zone fans. There are four levels to control the speed of fans:

- Standard: Baseboard management controller (BMC) controls both fan zones, with the CPU and Peripheral zones set at speed 50%;
- Optimal: BMC sets the CPU zone at speed 30% and the Peripheral zone at 30%;
- Heavy IO: BMC sets the CPU zone at speed 50% and the Peripheral zone at 75%;
- Full: All fans running at 100%.

To rule out fans' contribution to dynamic energy consumption, we set the fans at full speed before running the programs. Therefore, the energy consumption by fans is included in the static power consumption of the server.

Furthermore, during the application run, we monitor the server's temperatures and the fans' speeds with the help of Intelligent Platform Management Interface (IPMI) sensors. We make sure that there are no changes in the temperatures and the fans' speeds.

3.2.2 Measurement Accuracy and Uncertainty

The ground-truth method provides the node's total instantaneous power consumption, typically reported by the external power meter at its maximum sampling rate. Therefore, isolating the energy consumption attributable solely to the application is challenging due to persistent background noise arising from periodic OS, disk, and network bookkeeping tasks, as well as cooling-related activity.

In addition, the power meter has a manufacturer-reported measurement accuracy. The WattsUp Pro power meter employed in this work has a reported accuracy of $\pm$1.5% of the actual value, whereas the ANSI standard WT310 Yokogawa boasts an accuracy of $\pm$0.1%.

In addition to statistical stabilization, several experimental controls were employed to reduce run-to-run variability. These include fixing the system configuration across runs, minimizing background services, and maintaining stable cooling conditions (e.g., fixing fan speeds) to avoid temperature-induced power fluctuations. These precautions, described in Section 3.2.1, reduce environmental and system-level variability before applying statistical averaging.

To mitigate interference (noise) from activities unrelated to application execution, we employ statistical averaging (e.g., using the Student's t-test). In all experiments utilizing the ground-truth method, we ensure measurement reliability by repeating each experiment until the sample mean falls within a 95% confidence interval and achieves a precision (standard error of the mean) of 2.5%. In practice, meeting this stopping criterion typically required 5–10 repeated runs per data point (per transfer size and direction), depending on the platform.

For this purpose, we use the Student's t-test, which assumes that the individual observations are independent and that their population follows a normal distribution. We verify the validity of these assumptions using the Shapiro–Wilk test.

3.3 Methodology to Obtain Ground-truth Profiles for Data Transfer Between Compute Devices

3.3.1 Energy Measurement Using External Power Meters

We first present our methodology to accurately measure the energy consumption of data transfer between a host CPU and a GPU using the ***ground-truth*** method. We then apply the methodology to build the dynamic energy and execution time profiles of data transfer between a host CPU and a GPU on two heterogeneous hybrid servers, One containing an Intel Haswell multicore CPU and Nvidia K40c GPU and the other containing an Intel IceLake CPU and two Nvidia A40 GPUs as in Tables 3.1 & 3.3.

The methodology is automated in a software library (**libedm**) that we developed for this purpose [59]. The libedm's API is presented in the Appendix A.1.

A natural and intuitive approach to measure the dynamic energy consumption of a data transfer of m bytes between a pair of devices, d_A and d_B (in our case, host CPU and GPU or GPU and host CPU) comprises the following steps:

- Measure the dynamic energy consumption for sending m bytes from d_A to d_B followed by d_B to d_A.
- If the energy consumptions are $E_{A->B}$ and $E_{B->A}$, then one would calculate the dynamic energy consumption of data transfer of m bytes from d_A to d_B to be $\frac{E_{A->B}+E_{B->A}}{2}$.
- The same formula would be used for the dynamic energy consumption for sending m bytes from d_B to d_A.

However, there is an inherent assumption in this approach that the direction of data transfer is symmetric and does not matter. That is, the dynamic energy consumption for sending m bytes from d_A to d_B is the same as that for data transfer from d_B to d_A. One way to confirm the validity of this assumption is to conduct two round-trip experiments as follows:

- Start the energy measurement on the d_A side, send the data of m bytes from d_A to d_B and from d_B to d_A, stop the energy measurement on the d_A side and obtain the difference between the two measurements.

- Start the energy measurement on the d_B side, send the data of m bytes from d_B to d_A and from d_A to d_B, stop the energy measurement on the d_B side and obtain the difference between the two measurements.
- Now compare the energy consumption of the two round trips.

This assumption is difficult to validate when the pair of devices are the host CPU and GPU. Our methodology does not assume execution time or energy symmetry of data transfer between a host CPU and a GPU because all the measurements must be done on the CPU side. The GPU is a passive device and does not provide energy measurements that can be queried from the CPU side.

Our methodology to measure the dynamic energy consumption of data transfer of m bytes between the host CPU and a GPU device is to measure the dynamic energy consumption for a round trip comprising sending the data of m bytes from the host CPU to the GPU device and 1 byte from the GPU device to the host CPU.

Since the dynamic energy consumption of sending 1 byte is negligible compared to sending m bytes (where m is typically of the order of MB), this approach accurately measures the dynamic energy consumption of data transfer of m bytes between the host CPU and a GPU device. We adopt the same approach for measuring the dynamic energy consumption of data transfer of m bytes between the GPU device and a host CPU.

3.3.2 Main Steps for Energy Measurement of Data Transfer Between a host CPU and a GPU

The main steps to obtain the dynamic energy consumption for a data transfer of m bytes from a host CPU core to a GPU are outlined below. *All the steps are invoked from the CPU side in the main thread.*

- Query the CPU processor clock to obtain the start time.
- Start the energy meter on the CPU side using an energy measurement API's *start()* function [20].
- Launch a pthread from the program's main thread to perform the data transfer.
- Pin the pthread to a CPU core dedicated for this data transfer using the system call, sched_setaffinity().
- Send the data of size m bytes from the host CPU core to the GPU device.
- Send a data item of size 1 byte from the GPU device to the host CPU core.

- Stop the energy meter on the CPU side using the energy measurement API's *stop()* function [20].
- Query the CPU processor clock to get the end time and calculate the difference between the end and start times to get the execution time.
- The dynamic energy consumption of the data transfer is the difference between the total energy consumption and the static energy consumption, which is the static power consumption of the server multiplied by the execution time of the data transfer as described in section 2.1.

The execution time and dynamic energy of data transfer can be measured simultaneously since the overhead of energy measurement using the energy measurement API [20] is negligible.

Furthermore, to ensure reliability of the measurement, the steps above are run repeatedly until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) is achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions using the Shapiro-Wilk Test.

3.3.3 Main Steps for Energy Measurement of Data Transfer Between a GPU and the Host CPU

The main steps to obtain the dynamic energy consumption for a data transfer of m bytes from a GPU to the host CPU are outlined below. *All the steps are invoked from the CPU side in the main thread*. The only notable deviation from the steps above is that before commencing the measurement, we initiate the transfer of data of size m bytes from a host CPU core to the GPU device from the program's main thread. This data is then retained on the GPU device for the duration of the measurement.

- We send the data of size m bytes from a host CPU core to the GPU device from the program's main thread and keep this data on the GPU device for the remainder of the measurement.
- Query the CPU processor clock to obtain the start time.
- Start the energy measurement on the CPU side using an energy measurement API's *start()* function [20].
- Launch a pthread from the program's main thread to perform the data transfer.
- Pin the pthread to a CPU core dedicated for this data transfer using the system call, sched_setaffinity().

- Send the data of size m bytes from the GPU device to the host CPU core.
- Send a data item of size 1 byte from the host CPU core to the GPU device.
- Stop the energy meter on the CPU side using the energy measurement API's *stop()* function [20].
- Query the CPU processor clock to get the end time and calculate the difference between the end and start times to get the execution time.
- The dynamic energy consumption of the data transfer is the difference between the total energy consumption and the static energy consumption, which is the static power consumption of the server multiplied by the execution time of the data transfer as described in section 2.1.

3.3.4 Main Steps for Energy Measurement of p Data Transfers Between p Device Pairs

The main steps to obtain the dynamic energy consumption of parallel data transfers of data sizes, $\{m_1, \cdots, m_p\}$, between p pairs of devices where m_i is transferred between devices in the i-th pair are as follows:

- Query the CPU processor clock in the main thread to obtain the start time.
- Launch p pthreads from the program's main thread to perform the parallel data transfers.
- In each pthread, pin the thread to a dedicated CPU core using the system call, *sched_setaffinity()*.
- Start the energy measurement on the CPU side in the main thread using an energy measurement API's *start()* function [20].
- In pthread i ($i \in \{0, \cdots, p-1\}$), send the data of size m_i bytes from d_{i1} to d_{i2} in the i-th pair and a data item of size 1 byte from d_{i2} to d_{i1}.
- Invoke the *pthread_join()* call in the main thread to join with the data transfer pthreads.
- Stop the energy measurement on the CPU side in the main thread using the energy measurement API's *stop()* function [20].
- Query the CPU processor clock to get the end time and calculate the difference between the end and start times to get the execution time.

- The dynamic energy consumption of the p parallel data transfers is the difference between the total energy consumption and the static energy consumption, which is the static power consumption of the server multiplied by the total execution time of the parallel data transfers.

Note that the data transfers in the pthreads and the start of energy measurement in the main thread are synchronized by a barrier, which aims to start these operations at the same time.

3.3.5 Application of The Methodology

We apply our methodology to obtain the ground-truth dynamic energy profiles of data transfers between devices ((CPU, GPU) in heterogeneous hybrid server shown in Table 3.1) and between pair of devices (CPU, GPU_1) and (CPU, GPU_2) in the heterogeneous hybrid server shown in Table 3.3.

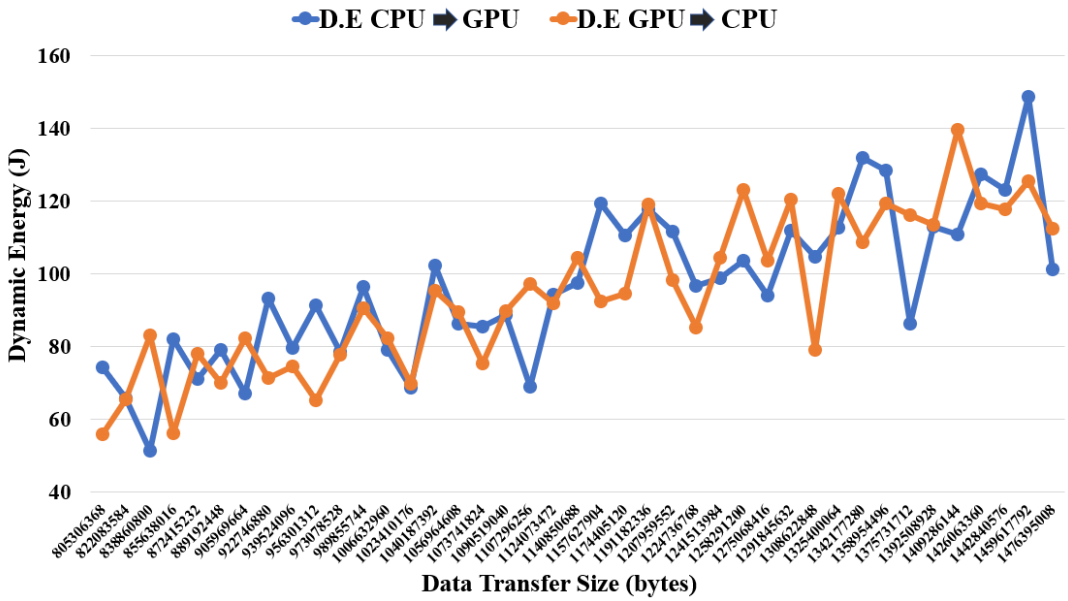
We analyse:

1. The dynamic energy and execution time of data transfers in forward and backward directions between a pair of devices (host CPU and a GPU).
2. The dynamic energy and execution time of data transfers between pairs of devices in serial.
3. The dynamic energy and execution time of parallel data transfers between pairs of devices.
4. The execution times involved in obtaining the profiles.

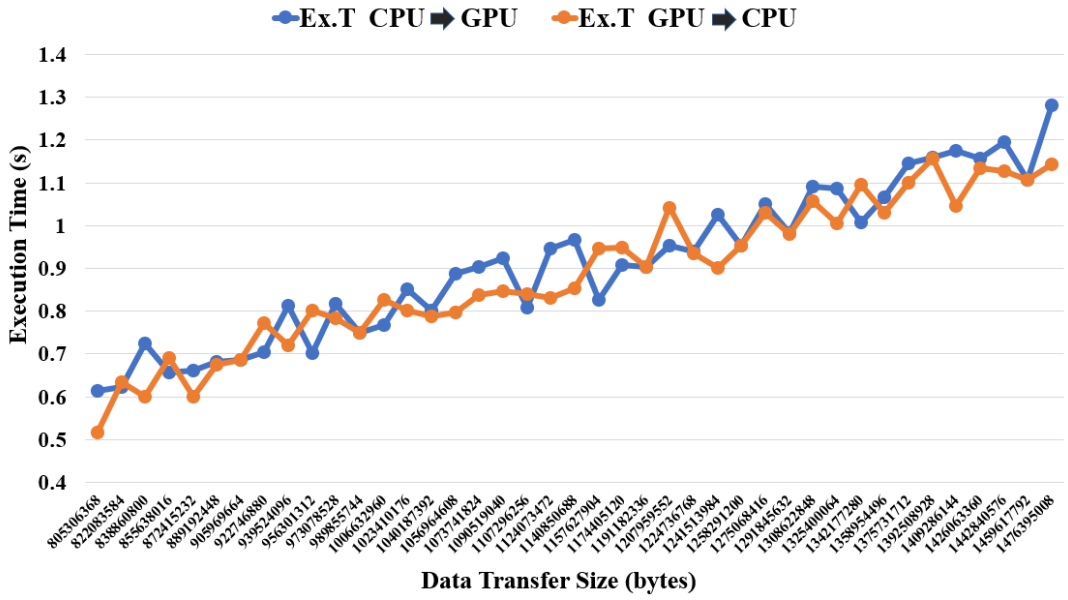
Each point in the profiles shown in Figures 3.1, 3.2, 3.3, 3.4, and 3.5 is obtained using repeated runs until the sample mean falls within a 95% confidence interval and achieves a precision (or standard deviation of the mean) of 2.5%. For clarity, the plots report only sample means; error bars are omitted because each point was repeated until the 95% confidence interval met the 2.5% precision criterion.

Figure 3.1 shows the dynamic energy and execution time profiles of data transfers between a pair of devices (CPU, K40c GPU) and (K40c GPU, CPU) in the Haswell K40c GPU server shown in Table 3.1). Figure 3.1(a) shows the dynamic energy consumption of a round-trip data transfer between the host CPU to K40c GPU direction, and K40c GPU to host CPU direction. Similarly, Figure 3.1(b) shows the execution time of data transfers between a pair of devices (CPU, K40c GPU)

Figure 3.2 shows the dynamic energy and execution time profiles of data transfers between the CPU and A40 GPU_1 in heterogeneous hybrid server shown in Table 3.3). Figures 3.2(a) and 3.2(b) show the dynamic energy consumption and execution time of a



(a) Dynamic Energy of data transfer from CPU $\Longrightarrow$ K40c GPU, and K40c GPU $\Longrightarrow$ CPU

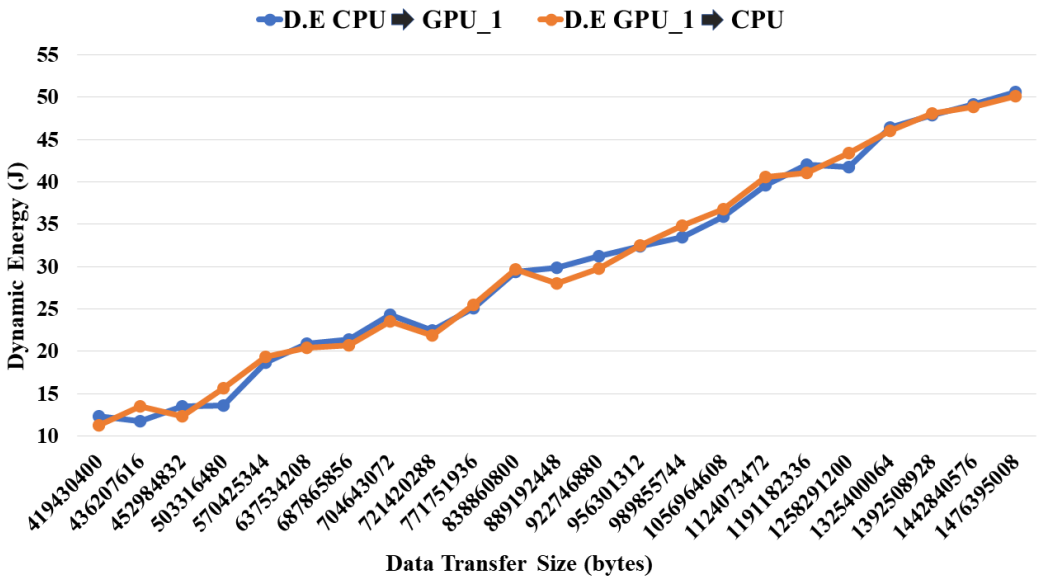


(b) Execution time of data transfer from CPU $\Longrightarrow$ K40c GPU, and K40c GPU $\Longrightarrow$ CPU

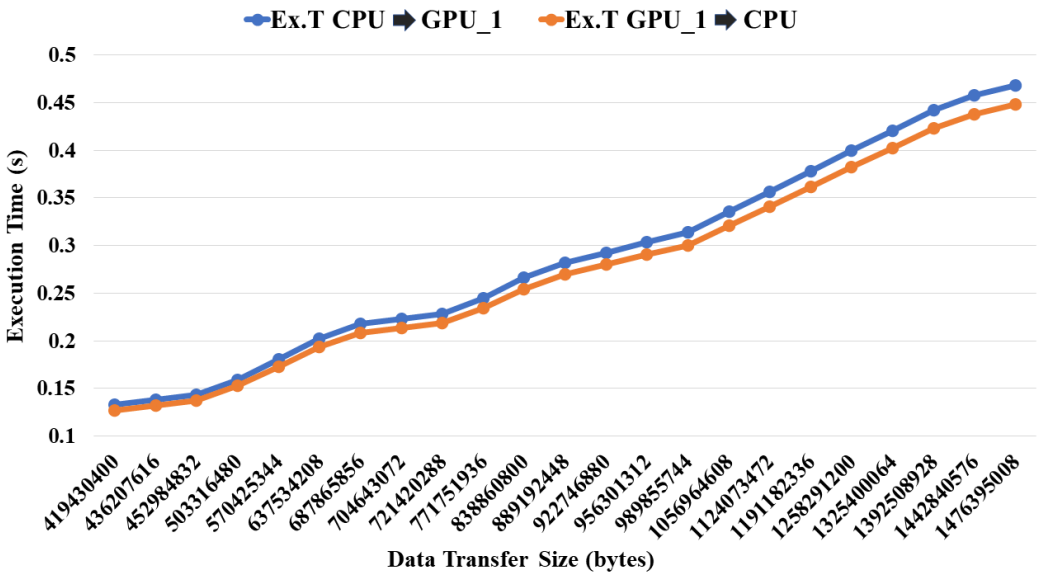
Figure 3.1: Dynamic energy and execution time profiles of data transfers between a pair of devices (CPU, K40c GPU) and (K40c GPU, CPU) in the heterogeneous server, comprising Nvidia K40c GPU as shown in Table 3.1.

round-trip data transfer between a pair of devices, (CPU, A40 GPU_1) and (A40 GPU_1, CPU), respectively.

Similarly, figure 3.3 shows the dynamic energy and execution time profiles of data transfers between the CPU and A40 GPU_2 in heterogeneous hybrid server shown in Table 3.3). Figures 3.3(a) and 3.3(b) depict the case for the pair of devices, (CPU, A40 GPU_2) and (A40 GPU_2, CPU).

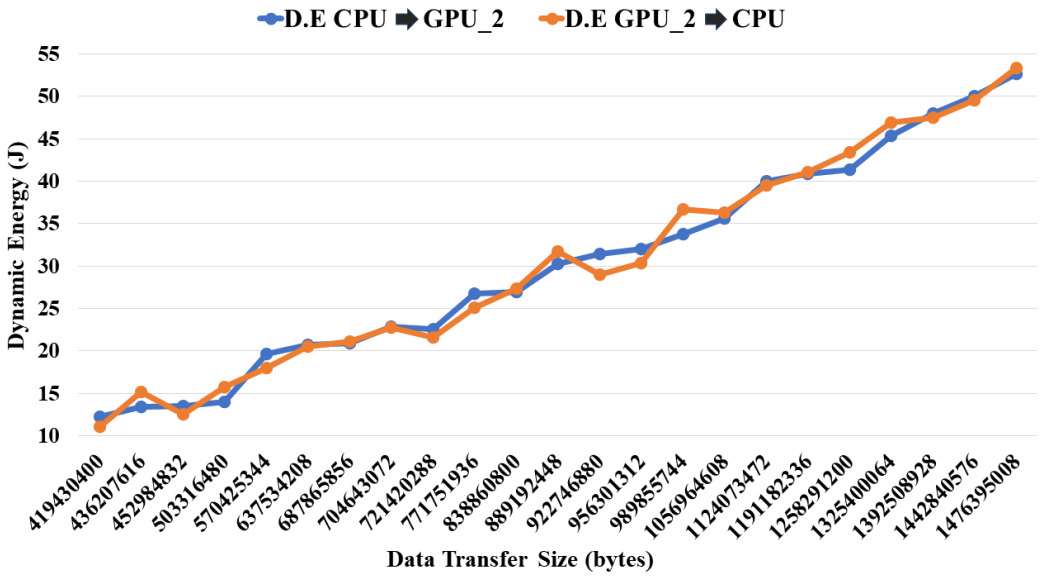


(a) Dynamic Energy: CPU $\Longrightarrow$ A40 GPU_1, A40 GPU_1 $\Longrightarrow$ CPU

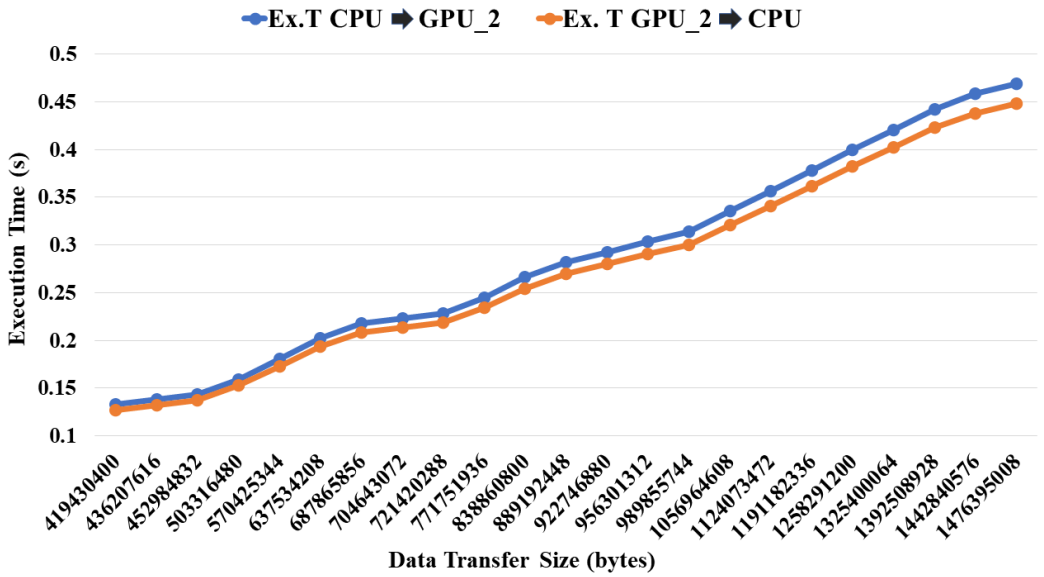


(b) Execution Time: CPU $\Longrightarrow$ A40 GPU_1, A40 GPU_1 $\Longrightarrow$ CPU

Figure 3.2: Dynamic energy and execution time profiles of data transfers between the CPU and A40 GPU_1 in the heterogeneous server comprising NVIDIA A40 GPU as shown in Table 3.3.

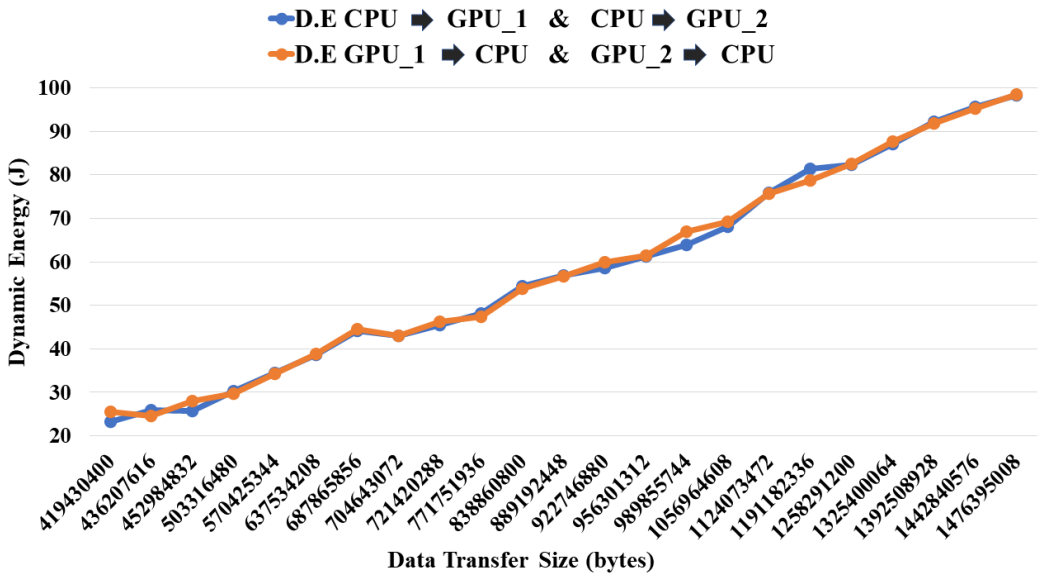


(a) Dynamic Energy: CPU $\Longrightarrow$ A40 GPU_2, A40 GPU_2 $\Longrightarrow$ CPU

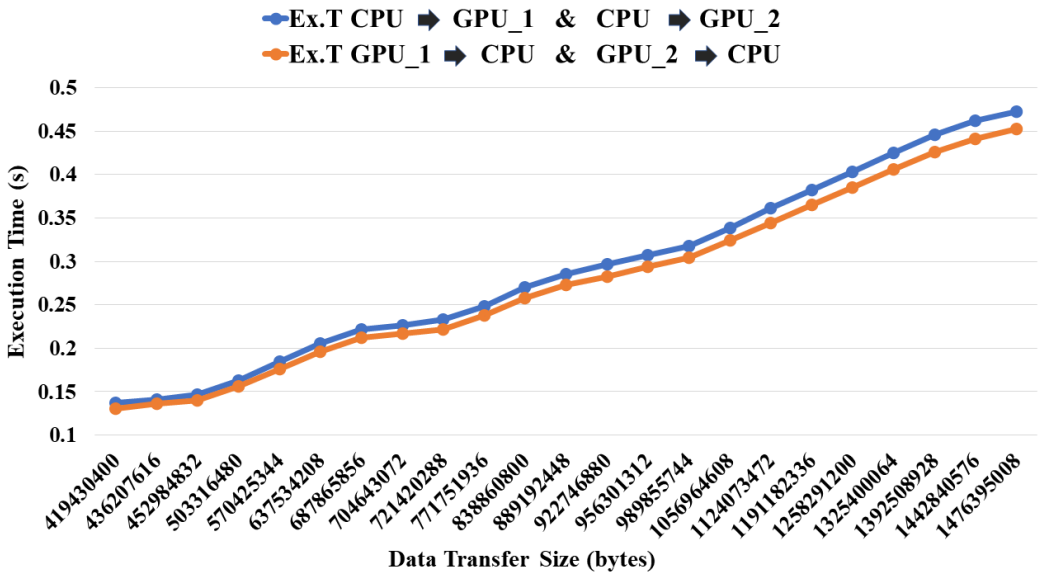


(b) Execution Time: CPU $\Longrightarrow$ A40 GPU_2, A40 GPU_2 $\Longrightarrow$ CPU

Figure 3.3: Dynamic energy and execution time profiles of data transfers between the CPU and A40 GPU_2 in the heterogeneous server comprising NVIDIA A40 GPU as shown in Table 3.3.

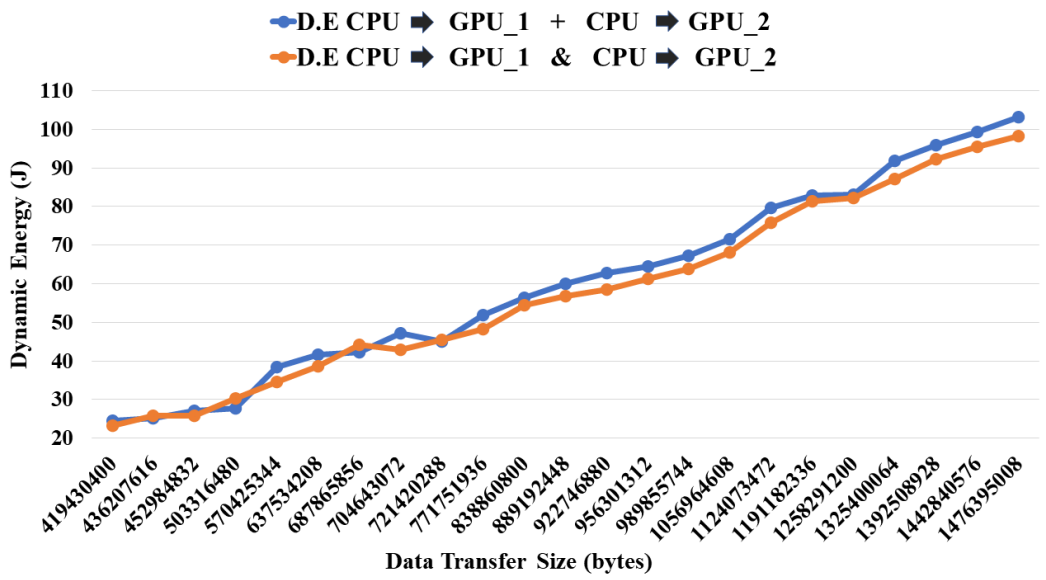


(a) Dynamic energy of data transfers between two pairs of devices CPU $\Longrightarrow$ A40 GPU_1 & CPU $\Longrightarrow$ A40 GPU_2 in parallel, A40 GPU_1 $\Longrightarrow$ CPU & A40 GPU_2 $\Longrightarrow$ CPU in parallel

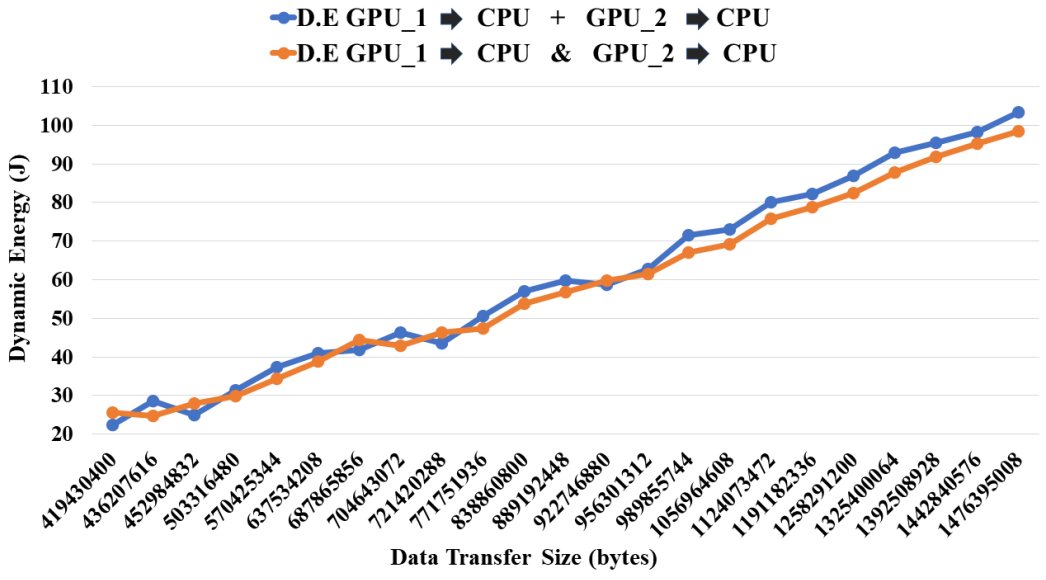


(b) Execution time of data transfers between two pairs of devices CPU $\Longrightarrow$ A40 GPU_1 & CPU $\Longrightarrow$ A40 GPU_2 in parallel, A40 GPU_1 $\Longrightarrow$ CPU & A40 GPU_2 $\Longrightarrow$ CPU in parallel

Figure 3.4: Dynamic energy and execution time profiles of parallel data transfers between the (CPU, A40 GPU_1), (CPU, A40_GPU 2) in the heterogeneous server comprising NVIDIA A40 GPU as shown in Table 3.3.



(a) Comparison between sum of serial dynamic energies from CPU $\Longrightarrow$ A40 GPU_1 + CPU $\Longrightarrow$ A40 GPU_2 , and dynamic energy of data transfers between two pairs of devices in parallel CPU $\Longrightarrow$ A40 GPU_1 & CPU $\Longrightarrow$ A40 GPU_2



(b) Comparison between sum of serial dynamic energies from A40 GPU_1 $\Longrightarrow$ CPU + A40 GPU_2 $\Longrightarrow$ CPU , and dynamic energy of data transfers between two pairs of devices in parallel A40 GPU_1 $\Longrightarrow$ CPU & A40 GPU_2 $\Longrightarrow$ CPU

Figure 3.5: Comparison of dynamic energy consumption between the sum of serial dynamic energies and dynamic energy of parallel data transfers between a pair of devices (CPU, A40 GPU_1), (CPU, A40_GPU 2) in the heterogeneous server comprising NVIDIA A40 GPU as shown in Table 3.3.

Figure 3.4 shows the dynamic energy and execution time profiles of data transfers happening in parallel between two pair of devices (CPU, A40 GPU_1) and (CPU, A40_GPU 2) in heterogeneous hybrid server shown in Table 3.3. Figures 3.4(a) and 3.4(b) show the dynamic energy consumption and execution time of data transfers happening in parallel between two pairs of devices, (CPU, A40 GPU_1) and (CPU, A40 GPU_2).

Figure 3.5 shows the sum of the dynamic energy consumption of data transfer between the pair of computing devices (CPU, A40 GPU_1) followed by data transfer between the pair (CPU, A40 GPU_2), and data transfers between the two pairs of devices in parallel in heterogeneous hybrid server shown in Table 3.3. Figure 3.5(a) shows the sum of the dynamic energy consumption of data transfer between a pair of devices (CPU, A40 GPU_1) followed by data transfer between the pair (CPU, A40 GPU_2), and data transfers between the two pairs of devices in parallel. Figure 3.5(b) shows the sum of the dynamic energy consumptions of data transfer between a pair of devices (A40 GPU_1, CPU) followed by data transfer between the pair, (A40 GPU_2, CPU), and data transfers between the two pairs of devices in parallel.

3.3.6 Results and Analysis

The summary of our findings for the data transfers between CPU and K40c GPU on heterogeneous server as in Table 3.1 is below:

- The dynamic energy and execution time of data transfer is highly non-linear with respect to data transfer size for both directions (CPU to K40c GPU and K40c GPU to CPU)
- Therefore, one cannot represent the functional relationship between dynamic energy and data transfer size and between execution time and data transfer size by linear models.
- The dynamic energy and execution time of data transfer is different for both directions (CPU to K40c GPU and K40c GPU to CPU), suggesting asymmetry.
- Hence, there must be different dynamic energy and execution time models for both directions to accurately represent this asymmetry.
- There is no correlation between dynamic energy and execution time, suggesting a potential opportunity for bi-objective optimization of data-intensive heterogeneous hybrid applications for energy and performance with data transfer size as a decision variable.

The summary of our findings for the data transfers between CPU and A40 GPUs on heterogeneous hybrid server as in Table 3.3 is below:

- The dynamic energy and execution time of data transfer is linear with respect to data transfer size for both directions (CPU to A40 GPU and A40 GPU to CPU) as in Figures 3.2, 3.3. Therefore, one can represent the functional relationship between dynamic energy and data transfer size and between execution time and data transfer size by linear models.
- The dynamic energy consumptions and execution times of data transfer between a pair of devices for both directions are almost equal suggesting symmetry as in Figures 3.2, 3.3. Therefore, one linear dynamic energy model and execution time model can be used to estimate the dynamic energy and execution time for both directions.
- The sum of dynamic energies of data transfers between a pair of devices, (CPU, A40 GPU_1) followed by a pair of devices (CPU, A40 GPU_2), is almost equal to the dynamic energy of parallel data transfers between these two pairs within the statistical confidence threshold of 5% set in our experiments as in Figure 3.5. Therefore, one can predict the dynamic energy and execution time of parallel data transfers between the two pairs of devices using just one linear dynamic energy model and execution time model.

The time to obtain each profile shown in the above figures is time-consuming and practically infeasible to be employed at runtime. For example, the time to obtain the dynamic energy profiles shown in the Figures 3.1, 3.2, 3.3, and 3.4 is around 8-10 hours and 12-15 hours for each direction respectively. Therefore, we investigate the energy measurement approaches involving on-chip sensors and software energy predictive models.

3.4 Energy Measurement Using On-chip Sensors

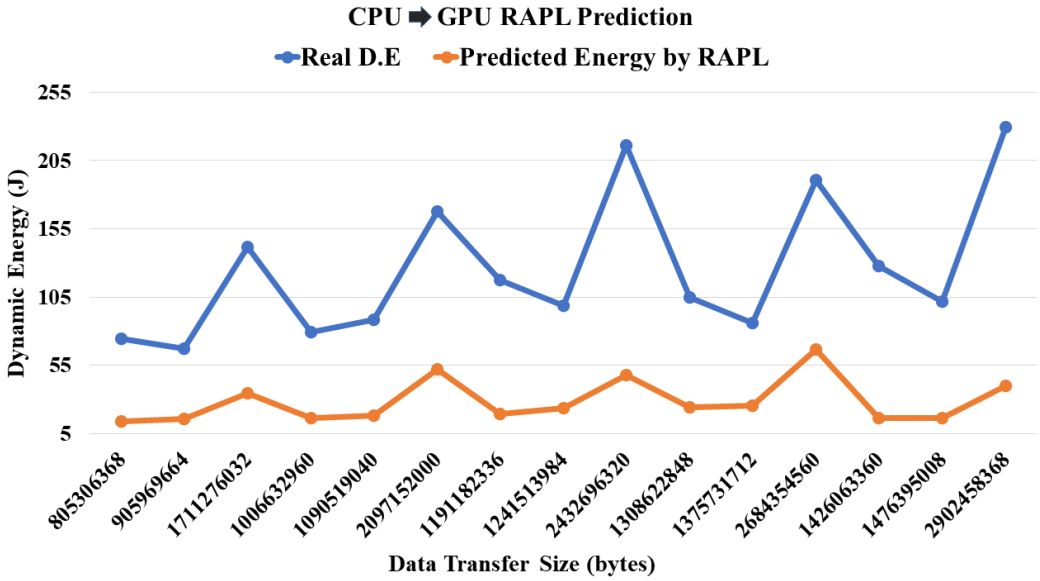
Mainstream CPU and GPU processors now provide on-chip power sensors that give power measurements at a high sampling frequency. The measurements can be collected using special programmatic interfaces. Intel multicore CPUs offer Running Average Power Limit (RAPL) [60] capability to monitor power and control frequency (and voltage). Nvidia GPUs have on-chip power sensors providing readings that can be obtained programmatically using Nvidia Management Library (NVML) [61] interface. The reported accuracy of the energy readings in the NVML manual is 5%.

However, we find that NVML *does not provide energy consumption for a data transfer between a host CPU and GPU*. Furthermore, on-chip power sensors, while cheap and

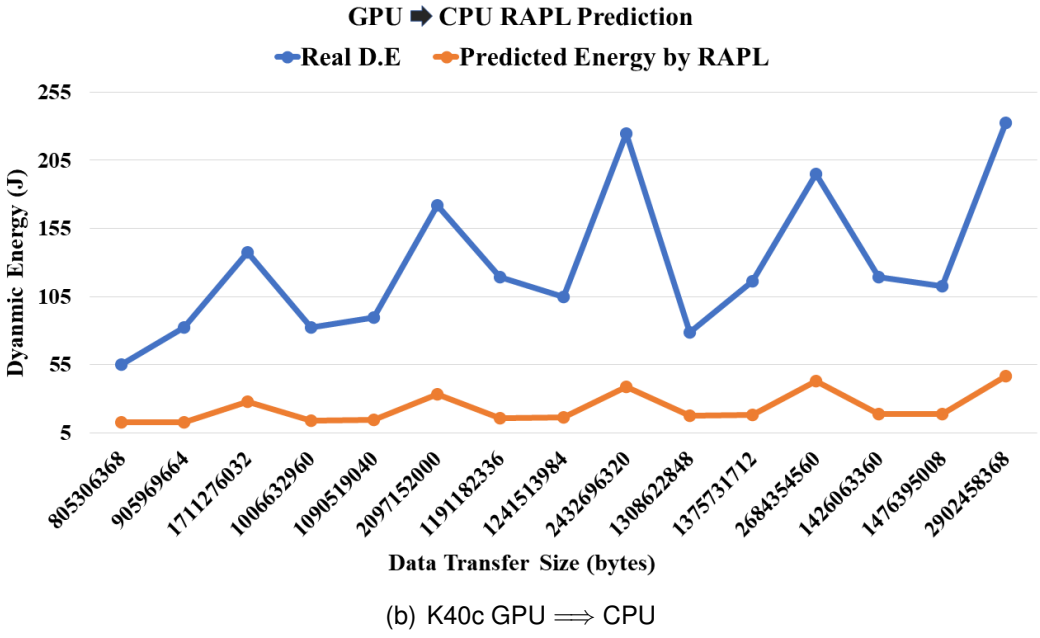
efficient, are inaccurate and poorly documented [21]. Extensive and sound experiments with several mainstream CPUs, accelerators, and highly optimized scientific kernels have shown that energy profiles constructed using state-of-the-art on-chip power sensors are qualitatively inaccurate [19], [29], [26], [27],[30].

For example, the shape of a dynamic energy profile built using on-chip sensors differs significantly from the shape obtained employing the ground-truth approach, showing the increase in energy consumption in situations where the ground-truth gives a decrease and vice versa.

Figure 3.6 compares the dynamic energy consumption of data transfer between a host CPU and a K40c GPU obtained using RAPL against the ground-truth method. One can see that RAPL is not accurate. Therefore, the energy measurements using on-chip sensors do not capture the holistic picture of the dynamic energy consumption during application execution.



(a) CPU $\Longrightarrow$ K40c GPU



(b) K40c GPU $\Longrightarrow$ CPU

Figure 3.6: Dynamic energy comparison of data transfers between pairs of computing devices (CPU, K40c GPU) and (K40c GPU, CPU) using RAPL and ground-truth measurement methods on the Haswell K40c GPU server.

3.5 Summary

The main focus of energy researchers has been to propose solutions to reduce energy consumption by computations due to the widespread belief that the energy consumption due to data transfers is insignificant. Therefore, the energy consumption of inter-device data transfers is unexplored because no research has tackled this area. As a result, no method or model exists to acurately measure and predict the energy of data transfer.

In this chapter, we comprehensively studied the energy consumption of data transfer between computing devices on heterogeneous hybrid servers using the two mainstream energy measurement methods: (a) System-level physical measurements using external power meters (*ground-truth*), (b) Measurements using on-chip power sensors.

First, we developed a methodology to accurately measure the energy consumption of data transfer between two devices using external physical power meters considered the *ground-truth*. We employed the methodology to build the energy and execution time profiles of data transfer on two heterogeneous server, one containing an Intel Haswell CPU and Nvidia K40c GPU and the other containing an Intel IceLake CPU and two Nvidia A40 GPUs.

While the energy and execution time profiles of data transfer for the A40 GPU server are symmetric and linear, the energy and execution time profiles of data transfer for the K40c GPU server are asymmetric and non-linear, suggesting a potential opportunity for bi-objective optimization for energy and performance.

While accurate, the *ground-truth* method has challenges. It is prohibitively time-consuming. In addition, it cannot be employed in dynamic environments (HPC platforms and data centres) containing nodes not equipped with power meters. Therefore, this finding necessitates exploring alternative energy measurement approaches (b) and (c).

We then investigated the second mainstream energy measurement method (b) on-chip sensor software tools to measure the energy of data transfer between a host CPU and a GPU. We showed that the on-chip sensor software tool for Intel multicore CPUs is inaccurate. In addition, the on-chip sensor software tool (NVML) for Nvidia GPU processors does not capture data transfer activity and, therefore, does not provide the energy consumption of data transfer between a host CPU and GPU.

Given the limitations of the on-chip sensor software tools, we turn to the third approach (c) energy predictive models to accurately predict the energy of data transfers between pairs of devices, which is explained in the next chapter 4.

Chapter 4

Energy Predictive Modelling on Heterogeneous Hybrid Servers

4.1 Introduction

Since the ground-truth method for obtaining dynamic energy profiles of data transfers is prohibitively time-consuming, and on-chip sensors in Intel multicore CPUs are inaccurate, while the Nvidia GPU sensor tool (NVML) fails to capture data transfer activity and thus cannot provide the energy consumption of data transfer between a host CPU and GPU, we turn to the third approach using software energy predictive models that employ performance events as predictor variables.

This method has emerged as a promising alternative for developing software power meters for runtime energy profiling compared to other mainstream methods. There are two main reasons for this. First, if implemented correctly, its runtime accuracy can surpass that of other approaches.

In addition, a software power meter is cost-effective since the reliable performance events employed in accurate linear energy predictive models exhibit the same counts from run to run and do not need statistical averaging. Therefore, the time to obtain an experimental data point using this approach is significantly less than the other approaches.

A software power meter that uses a linear energy predictive model with reliable performance events, defined as those events that yield consistent counts across runs regardless of environmental noise, will produce the same ground-truth mean value even when environmental noise is present. This indicates that the performance event-based method can achieve the same level of accuracy as the ground-truth method when there is no noise. Additionally, it is likely to be more accurate in noisy conditions because it provides measurements that would align with the ground-truth method under ideal, noise-free circumstances.

The term *software power meter* is used to denote a linear energy predictive model based on performance events, designed to measure an activity (computation or data transfer). Therefore, it refers specifically to a performance event-based software power meter.

To summarize, the third approach using energy predictive models employing reliable performance events, is the most cost-effective and potentially the most accurate method for the measurement of energy consumption of applications. As the only realistic alternative to the ground-truth method, this approach is ideal for use at runtime in environments lacking power meters, emphasizing its practicality.

However, the number of performance events is significant, and collecting all of them is practically infeasible at runtime. Hence, selecting a small subset of performance events that can effectively capture all the energy consumption activities during the data transfer or an application execution is a challenging task.

In this chapter, we propose a novel methodology that employs a fast selection procedure for selecting a small subset of performance events to develop a software power meter based on linear energy predictive model for predicting the energy consumption of a single activity that is data transfer in one direction (host-to-device or device-to-host).

We then design and develop reliable software power meters based on linear energy predictive models that leverage selected performance events to accurately predict the energy consumption of a single activity, whether it is computation or data transfer. We experimentally determine the accuracy of our proposed software power meters using three parallel scientific programs on our heterogeneous server.

This chapter is organized as follows. Section 4.2 presents a fast and systematic procedure for selecting additive and energy-correlated performance events to develop a software power meter for predicting the energy consumption of a single activity. Section 4.3 outlines the design, training, and evaluation of accurate software power meters for predicting the energy consumption of a single activity whether it is computation or data transfer, using carefully selected performance events for the two heterogeneous hybrid servers. Section 4.4 details the experimental validation of the software power meters through separate energy estimation of each single activity (computation or data transfer) in three parallel applications, along with results and discussion. Finally, Section 4.5 summarizes the key findings and concludes the chapter.

4.2 Software Power Meters for Data Transfer Between Computing Devices

4.2.1 Overview

The approach, based on software energy predictive models and utilizing a range of measurable runtime performance-related predictor variables, is a practical and cost-effective alternative to the expensive ground-truth approach. Energy predictive models that employ performance event counts as predictor variables are at the forefront [19], [41], [42], [43], [48], [49], [50]. These models use hardware performance counters. Hardware performance counters are specialized hardware registers provided in modern processor architectures that store the counts of software and hardware activities called performance events. These counters are instrumental in low-level performance analysis and tuning.

Likwid [62] is a well-known mainstream tool for obtaining the performance event counts. It offers c number of core-level and socket-wide counters and e performance events where c and e vary from one processor architecture to another and e is much greater than c. Furthermore, the values of c and e increase with every processor generation. The counters are further divided into core-level and socket-wide counter groups of a specific hardware component of the architecture.

For example, there are 160 counters, {CBOX0C0,...,CBOX0C3,...,CBOX39C0,..., CBOX39C3}, that belong to the CBOX group that can capture the counts of activities of the different caching agents in the Intel IceLake multicore CPU processor. A caching agent is a per-core unit inside the cores that maintains the cache coherency between multiple processor cores. Therefore, each group of counters can store the counts of performance events of that group. Furthermore, all the counters can gather as many performance event counts as possible in one application run.

Likwid offers 143 counters across 16 groups, and 1665 performance events for the Haswell multicore CPU in the heterogeneous server in Table 3.1. It offers 328 counters across 18 groups, and 2001 performance events for the Skylake multicore CPU in the heterogeneous server in Table 3.2. It offers 347 counters across 18 groups, and 2773 performance events for the Intel IceLake multicore CPU in the Icelake A40 GPU server in 3.3.

PMC (Performance Monitoring Counters) is one such counter group that captures core-level performance events. There are eight counters in the PMC group for the Intel IceLake multicore CPU processor. Hence, to gather all the 258 core-level performance events in the PMC group, one must run the application 33 times. If one includes the cost of statistical averaging to obtain the mean and variance for each event, then collecting all the architecture events becomes prohibitive and practically infeasible at runtime.

The CUDA Profiling Tools Interface (CUPTI) [63] tool provides performance events

and metrics for Nvidia A40 GPU that are typically employed for performance profiling. Like the performance events for multicore CPUs, the CUPTI events and metrics have also been used in dynamic energy predictive models [48], [49], [50]. For the Nvidia A40 GPU, the Nsight Compute profiler is used to obtain the events and the metrics. However, they are also significant in number. Furthermore, the number of events increases with each new processor generation.

Therefore, choosing a small subset of performance events that effectively captures all the energy consumption activities during application execution is a challenging problem. While statistical methods like correlation and Principal Component Analysis (PCA) seem like natural solutions, relying solely on them proves ineffective in generating accurate energy predictive models, as the following reasons illustrate.

One effective technique is the *additivity test* proposed in [46]. The authors employ this test to check the additivity of core-level performance events belonging to the PMC counter group. They find that the cause of the inaccuracy of PMC-based energy models is that many popular core-level performance events (obtained using PMCs) are not additive on modern multicore processors.

The additivity test is based on the experimental observation that energy is additive. Indeed, the energy consumption of serial execution of two applications A and B will equal the sum of their consumptions,

$$E_{AB} = E_A + E_B \tag{4.1}$$

Therefore, any PMC event x in a linear energy predictive model should be additive,

$$x_{AB} = x_A + x_B \tag{4.2}$$

The authors [46] employ the additivity test to check the additivity of core-level performance events in the PMC counter group. They find that the cause of the inaccuracy of PMC-based energy models is that many popular core-level performance events (obtained using PMCs) are not additive on modern multicore processors. The accuracy of PMC-based energy models can be improved by removing non-additive PMC-based performance events.

Furthermore, the other leading cause of inaccuracy is the violation of basic laws of energy conservation in these models, including non-zero intercept in dynamic energy models, negative coefficients in additive terms, as well as non-linearity of some advanced models (including ML models) [46], [47]. The theory of energy predictive models of computing [47] formalizes properties of performance event count-based energy predictive models derived from the fundamental physical law of energy conservation.

In addition, it provides practical implications of the basic energy conservation laws for the accuracy of linear energy predictive models, including selection criteria for model

variables, model intercept, and model coefficients to construct accurate and reliable linear energy predictive models.

Thus, the accuracy of dynamic energy models employing PMC-based performance events can be improved by removing non-additive PMCs from models and enforcing basic energy conservation laws. Shahid et al. [19] propose the most accurate linear dynamic energy predictive models exhibiting 10-20% accuracy on popular scientific kernels by using additivity and following the selection criteria of the theory for better model prediction accuracy. The models employ processor utilization, memory utilization, and Likwid PMC-based performance events as model variables.

In this work, we propose a fast selection procedure that combines a natural grouping of performance events derived from the processor architecture, additivity, the theory of energy predictive models of computing, and high correlation to select a small subset of performance events that can be employed in software power meters based on linear energy models to accurately predict the energy of a single activity that is data transfer between a host CPU and a GPU.

We observe no events or metrics in CUPTI dedicated to capturing data transfer activity on the GPU side. This finding reflects the passive role of GPUs in data transfer between CPUs and GPUs, making the GPU's share of energy-consuming activities insignificant compared to the CPU.

Therefore, our procedure primarily focuses on selecting a subset of Likwid performance events provided for the multicore CPU. It analyzes all the available Likwid performance events for a server to select a subset unlike the research works [46], [47] that focus only on core-level events belonging to the PMC performance monitoring counter group.

The procedure has three main stages and is automated in a software library (**libedm**) that we developed for this purpose [59]. It takes the desired maximum average additivity percentage error and positive correlation as input and aims to determine performance events that can be obtained in one application run. The procedure ensures that enough performance monitoring counter registers are selected to store all the selected performance events in one application run. Finally, the procedure is fast compared to an approach that analyzes all pair-wise correlations, which is infeasible since the number of performance events is significant.

4.2.2 Fast Selection Procedure For Performance Events

The inputs to the procedure are n base applications and m compound applications, the maximum average additivity percentage error, $\mathbb{A}, 0 \leq \mathbb{A} \leq 100$, and the minimum positive correlation, $\rho, 0 \leq rho \leq 100$. Note that we use the same parameter ρ for checking correlation between two performance events and between a performance event and dy-

namic energy consumption. A *compound application* is defined as the serial execution of two base applications. If the base applications are A and B, we denote their compound application by AB. Each base application $j, j \in \{0, \cdots, n-1\}$, involves data transfers of size, $\{d_{1j}, \cdots, d_{pj}\}$, between p pairs of computing devices where d_{ij} is transferred between devices in the i-th pair.

Our proposed procedure does not use the Pearson's correlation coefficient to determine the correlation between two performance events or the correlation of a performance event with dynamic energy. It checks the functional relationship between the pair of performance events. To determine if a performance event is highly correlated with dynamic energy, it checks the trends of the functional relationships between the performance event and dynamic energy with data transfer size. If it finds significant non-linearity in a functional relationship, then the correlation is deemed not highly positive.

The procedure's output is a small set of highly additive and positively correlated Likwid performance events. Furthermore, the procedure aims to determine performance events that can be obtained in one application run. It returns a boolean flag indicating whether the performance events can be obtained in one application run.

The procedure has three stages. The first stage's main steps determining the highly additive and positively correlated performance events [46], [47] are detailed below. The performance events with additivity error ($\leq \mathbb{A}$) are selected. From the selected highly additive performance events, performance events with positive correlation ($\geq \rho$) are shortlisted for the second stage.

4.2.2.1 First Stage

The main steps of the first stage are as follows:

- Query the CPU processor clock to obtain the start time. Start the energy meter on the CPU side using an energy measurement API's *start()* function [20].
- Execute a base or compound application in the input dataset.
- Stop the energy meter on the CPU side using the energy measurement API's *stop()* function [20].
- Query the CPU processor clock to get the end time and calculate the difference between the end and start times to get the elapsed execution time. The dynamic energy consumption of the application execution is obtained.
- The event values during the data transfers in the application are obtained using the *Likwid-perfctr* tool.

- We verify that the execution time and dynamic energy consumption of a compound application is the sum of the execution times and dynamic energy consumptions of the constituent base applications.

- The additivity error of a performance event for a compound application is calculated as follows:

$$\text{Additivity error (\%)} = \left| \frac{(e_A + e_B) - e_{AB}}{(e_A + e_B + e_{AB})/2} \right| \times 100 \tag{4.3}$$

  where e_{AB}, e_A, e_B are the performance event counts for the compound (AB) and constituent base applications, A and B, respectively. The sample average of this error is obtained from multiple experimental runs. The additivity error of the performance event is the maximum of errors for all the compound applications in the dataset.

- For each application, create an event record containing the dynamic energy and performance event values. Since performance event counts may vary widely across different counters (e.g., some in thousands and others in millions), the event records are then normalized using the *min - max* scaling or normalization:

$$x' = \frac{(x - \min(x))}{\max(x) - \min(x)}$$

  where x is the original performance event count, $\min(x)$ and $\max(x)$ are the minimum and maximum values of the event counts, and x' is the normalized value. For each event count, we subtract the minimum value and divide by the difference between the maximum and minimum value.

- After normalization, we calculate the correlation between each performance event and dynamic energy consumption. Correlation measures the strength of the relationship between performance event counts and dynamic energy usage. A high positive correlation indicates that a performance event is a strong predictor of energy consumption and is therefore valuable for constructing accurate energy predictive models.

- The performance events with insignificant counts (less than or equal to 10) and not deterministic and reproducible are eliminated. A performance event is deemed deterministic and reproducible if it exhibits the same value (within a tolerance of 5.0%) for different executions of the same application with the same runtime configuration on the same server.

- The performance events with additivity error ($\leq \mathbb{A}$) are selected.

- Finally, from the selected highly additive performance events, performance events with positive correlation ($\geq \rho$) are shortlisted for the second stage.

4.2.2.2 Second Stage

In the second stage, we determine the final shortlist of performance events, starting with a natural grouping of performance events derived from the processor architecture. This stage comprises two main steps: the *intra-group* and *inter-group* steps. The intra-group step prunes the performance events in each group based on correlation with each other and dynamic energy consumption.

The inter-group step prunes the performance events across groups based on pairwise correlation with each other. We follow this approach since Likwid provides an architecture grouping of performance events for all the multicore CPU systems employed in this work. However, this approach is challenging for an arbitrary system with no such natural grouping.

The second stage aims to find a minimal set of performance events that are highly positively correlated with dynamic energy and not highly positively correlated with each other, thereby capturing separate independent contributions to dynamic energy that can be obtained in one application run. One can employ two approaches to finding a grouping of performance events that are highly correlated with each other and selecting one performance event as a representative from a group.

The first approach is a *bottom-up approach (cluster by synthesis)* where all pairwise correlations are analyzed to create meaningful clusters. However, this approach is practically infeasible since the number of performance events is significant. The second approach is a *top-down approach (cluster by analysis)*, which begins with groups of performance events that naturally exist together since they record the activities of the same hardware component in processor architecture.

For example, Tables 4.1, 4.2 show performance counter groups, {BBOX, ..., WBOX} and {CBOX, ..., WBOX}, provided by Likwid for the Intel Haswell and Icelake multicore CPU as in Tables 3.1, 3.3. For Intel Icelake multicore CPU, the CBOX performance monitoring counter group contains the performance events occurring in the last level cache (LLC) coherency engine in the uncore. The WBOX group has performance events recording the activities of the power control unit (PCU) in the uncore.

The main steps of the second stage are as follows:

1. The performance monitoring counter groups of the shortlisted performance events from stage one are determined and labelled $g_1, ..., g_G$, where G is the number of groups. Each group $g_i, i \in \{1, \cdots, G\}$ is then divided into sub-groups of performance events highly positively correlated with each other (correlation $\geq \rho$) as follows:

- Consider a group $g_i, i \in \{1, \cdots, G\}$. Let us assume it contains c_i shortlisted performance events from stage one and labelled $\{E_1, \cdots, E_{c_i}\}$. We place the event E_1 in a sub-group s_{i1}. We then check the correlation of E_2 with E_1. If both are highly positively correlated with each other, E_2 is placed in the sub-group s_{i1}. Then, if E_3 is highly positively correlated with each of the events in s_{i1}, it is placed in s_{i1} and so until the last event E_{c_i}. We then consider the remaining events not placed in s_{i1} (given by the set difference, $g_i - s_{i1}$) and repeat this process until all the performance events in g_i are allocated to sub-groups.
- The automated procedure does not use the statistical method of correlation to decide whether the performance events are highly positively correlated or not. It checks the functional relationship between the pair of performance events for significant non-linearity to decide whether the performance events are highly positively correlated or not.
- We iterate over the sub-groups in each group. For each group $g_i, i \in \{1, \cdots, G\}$, we pick it's first sub-group and select one performance event with the highest positive correlation ($\geq \rho$) with dynamic energy. If two or more performance events have the same highest positive correlation, we randomly pick one. We call the chosen performance event the *representative* of the sub-group. We then consider the second sub-group and select a representative that has the highest positive correlation ($\geq \rho$) with dynamic energy and not highly positively correlated with the *representative* of the first sub-group. We continue this process until we exhaust all the sub-groups in g_i.

2. For each group $g_i, i \in \{1, \cdots, G\}$, we iterate over its sub-groups. For each sub-group in g_i, we check the correlation of its representative with each of the representatives of the sub-groups in g_{i+1} followed by g_{i+2} and so on until g_G. Consider a sub-group in g_i and a sub-group in g_{i+1}. If the representatives of the two sub-groups are not highly positively correlated with each other (correlation $< \rho$), we select both representatives in the final shortlist. Otherwise, we select only the representative of the sub-group in g_i. Furthermore, we remove the sub-group in g_{i+1} from further consideration.

4.2.2.3 Third Stage

In the third stage, for each group selected in the second stage, we check if enough performance monitoring counters are available to store the shortlisted performance events of that group. If yes, the procedure terminates and returns the shortlisted performance events. Otherwise, we iterate over the groups with more shortlisted performance events

than the counters. We increase the correlation (ρ) in steps of 1%. For each group, we remove performance events that have a correlation with dynamic energy $< \rho$.

We repeat until the procedure finds the desired set of performance events until correlation (ρ) is set to the maximum of 99%. If the procedure does not find the desired set of performance events with correlation (ρ) equal to 99%, we try reducing the additivity error ($\mathbb{A}$) in steps of 1%. We remove performance events in each group with additivity error $> \mathbb{A}$. We repeat until the procedure finds the desired set of performance events or until the additivity error is set to the minimum of 1%. Finally, the procedure returns a boolean flag with the value 'Y' if it finds the desired set of performance events and 'N' otherwise.

Table 4.1: Performance monitoring counter groups in Intel Haswell multicore CPU of the Haswell K40c GPU server as shown in Table 3.1.

Performance Counter Groups	Description
BBOX	Measurements of the Home Agent (HA) in the uncore
CBOX	Last level cache (LLC) coherency engine in the uncore
MBOX	Integrated memory controllers (iMC) in the uncore
PBOX	Measurements of the Ring-to-PCIe (R2PCIe) interface in the uncore
PMC	Core-local general purpose counters
PWR	Measurements of the current energy consumption through the RAPL interface
QBOX	Measurements of the QPI Link layer (QPI) in the uncore
SBOX	Socket internal traffic through ring-based networks
UBOX	System configuration controller, interrupt traffic, and system lock master
WBOX	Power control unit (PCU) in the uncore

Table 4.2: Performance monitoring counter groups in Intel IceLake multicore CPU of the Icelake A40 GPU server as shown in Table 3.3.

Performance Counter Groups	Description
CBOX	Last level cache (LLC) coherency engine in the uncore
IBOX	Responsible for maintaining coherency for Integrated Input/Output controller (IIO) traffic
MBOX	Integrated memory controllers (iMC) in the uncore
M2M	Mesh2Mem (M2M) which connects the cores with the Uncore devices. The interface between the Mesh and the Memory Controllers.
PMC	Core-local general purpose counters
PWR	Measurements of the current energy consumption through the RAPL interface
QBOX	Measurements of the QPI Link layer (QPI) in the uncore
SBOX	Socket internal traffic through ring-based networks
TCBOX	Integrated Input/Output controller (IIO) counters
UBOX	System configuration controller, interrupt traffic, and system lock master
WBOX	Power control unit (PCU) in the uncore

4.2.3 Applying the Procedure for Heterogeneous Hybrid Servers

We use our methodology to determine the highly additive and positively correlated performance events for each direction (host CPU to GPU and GPU to host CPU) for the two heterogeneous hybrid servers in Tables 3.1 and 3.3.

4.2.3.1 Justification of Thresholds and Sensitivity Analysis

The additivity threshold and correlation threshold used in this work are based on the physical properties of energy consumption and the need to ensure reliable and accurate energy predictive models.

Energy consumption is inherently additive. Therefore, performance events employed in linear energy predictive models must also exhibit strong additivity. In this work, we set the maximum average additivity percentage error, $\mathbb{A}$, to 5%. This value ensures that the selected performance events closely follow the additive property of energy consumption while allowing minor deviations due to measurement noise, hardware variability, and runtime system activities. Experimental observations show that performance events satisfying this additivity threshold produce accurate and stable linear energy predictive models.

Similarly, we set the minimum positive correlation threshold, ρ, to 95%. This ensures that the selected performance events exhibit a strong and consistent functional relationship with dynamic energy consumption. Performance events with high positive correlation follow the same trend as dynamic energy with increasing data transfer size. This ensures that the selected events accurately capture the underlying hardware activities contributing to energy consumption.

In this work, correlation is determined by analysing the functional relationship between performance events and dynamic energy consumption rather than relying on statistical correlation measures such as the Pearson correlation coefficient. We developed this correlation assessment approach specifically for this research, since the relationship between hardware performance events and energy consumption is assumed to be physically meaningful and largely deterministic. Generic statistical correlation measures are designed for stochastic relationships and may not adequately reflect the structured and monotonic behaviour observed in event-to-energy mappings in our experiments.

Therefore, analysing trend consistency between performance events and dynamic energy provides a more appropriate selection criterion for this problem. Specifically, the functional relationship between performance event counts and dynamic energy is analysed across varying data transfer sizes, and performance events that exhibit a consistent monotonic increase or decrease matching the dynamic energy trend across multiple experimental runs are considered highly positively correlated.

We also analysed the sensitivity of the selection procedure to variations in the additivity and correlation thresholds. Relaxing the additivity threshold results in the inclusion of performance events that violate the additive property of energy, which leads to reduced model accuracy. Similarly, relaxing the correlation threshold results in the inclusion of performance events that do not consistently follow the dynamic energy trend, reducing predictive accuracy.

Conversely, tightening the additivity and correlation thresholds results in fewer performance events being selected. While this improves theoretical model robustness, it may eliminate useful performance events and reduce model completeness. Therefore, the selected thresholds of 5% additivity error and 95% correlation provide an appropriate balance between model accuracy, robustness, and practical feasibility.

These thresholds enable the selection of a small set of reliable performance events that accurately capture the energy consumption behaviour while allowing efficient deployment of software power meters at runtime. Furthermore, these thresholds ensure that the selected performance events can be obtained concurrently within the available performance monitoring counter registers, enabling efficient runtime deployment without requiring multiple application executions.

The number of highly additive performance events for the two directions in the Haswell k40c GPU server are 495 and 488. The number of highly additive and positively correlated output from the first stage are 367 and 363, respectively, for the two directions. Similarly, the number of highly additive performance events for the two directions in the Icelake A40 GPU server are 461 and 480. The number of highly additive and positively correlated output from the first stage are 398 and 411, respectively, for the two directions.

The number of performance monitoring counter groups (G) shortlisted in the first stage are 8 and 9 for the two directions in the Haswell k40c GPU server and 6 and 9 for the two directions in the Icelake A40 GPU server. Figures 4.1, 4.2 shows the resulting decomposition of the CBOX and M2M (Mesh2Mem) group into sub-groups in the intra-group step of the procedure's second stage. The plot on the top for both figures 4.1, 4.2 shows events belonging to the same sub-group having a high positive correlation with each other. The plot on the bottom for both figures 4.1, 4.2 shows events belonging to two sub-groups lacking a high positive correlation.

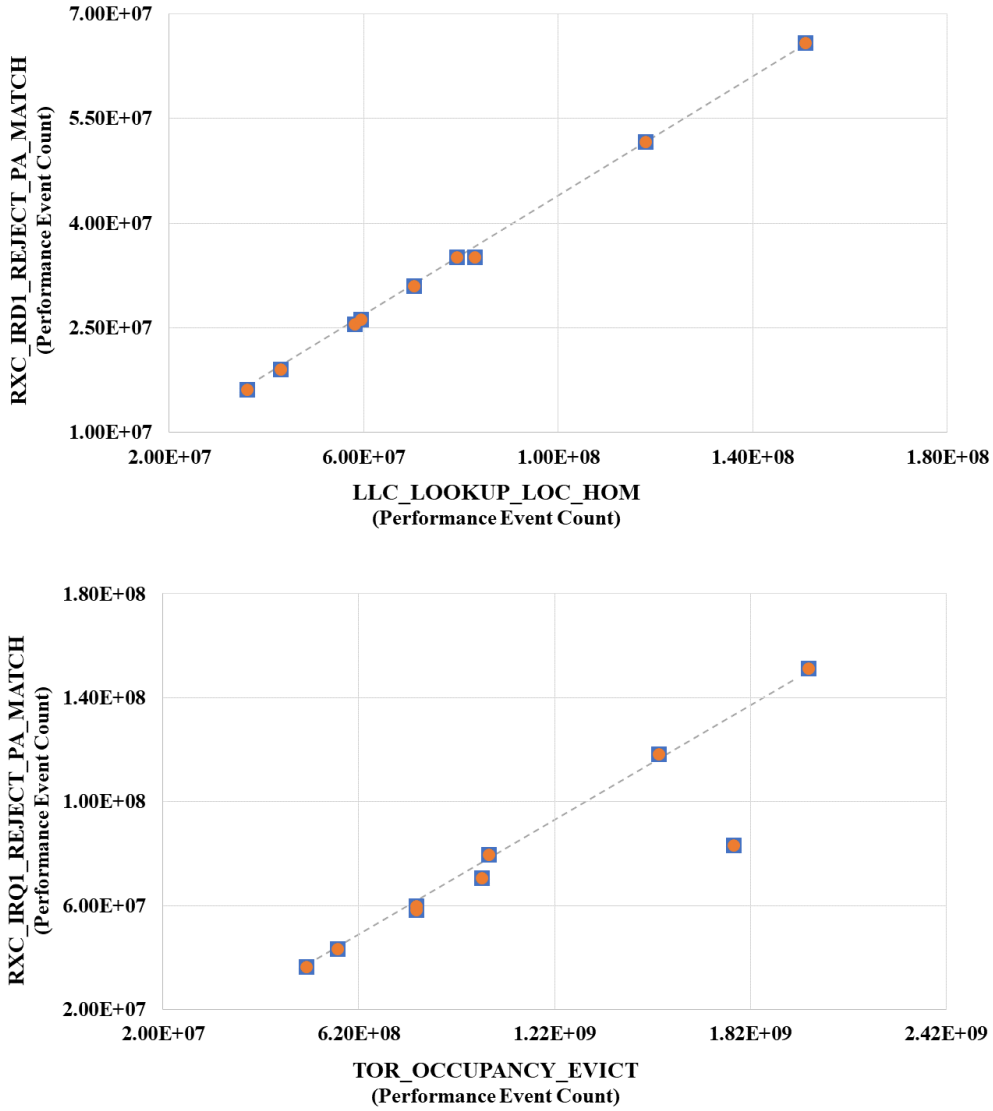


Figure 4.1: a). Correlation between performance events of one sub-group in the CBOX group. b). Correlation between performance events of two different sub-groups in the CBOX group

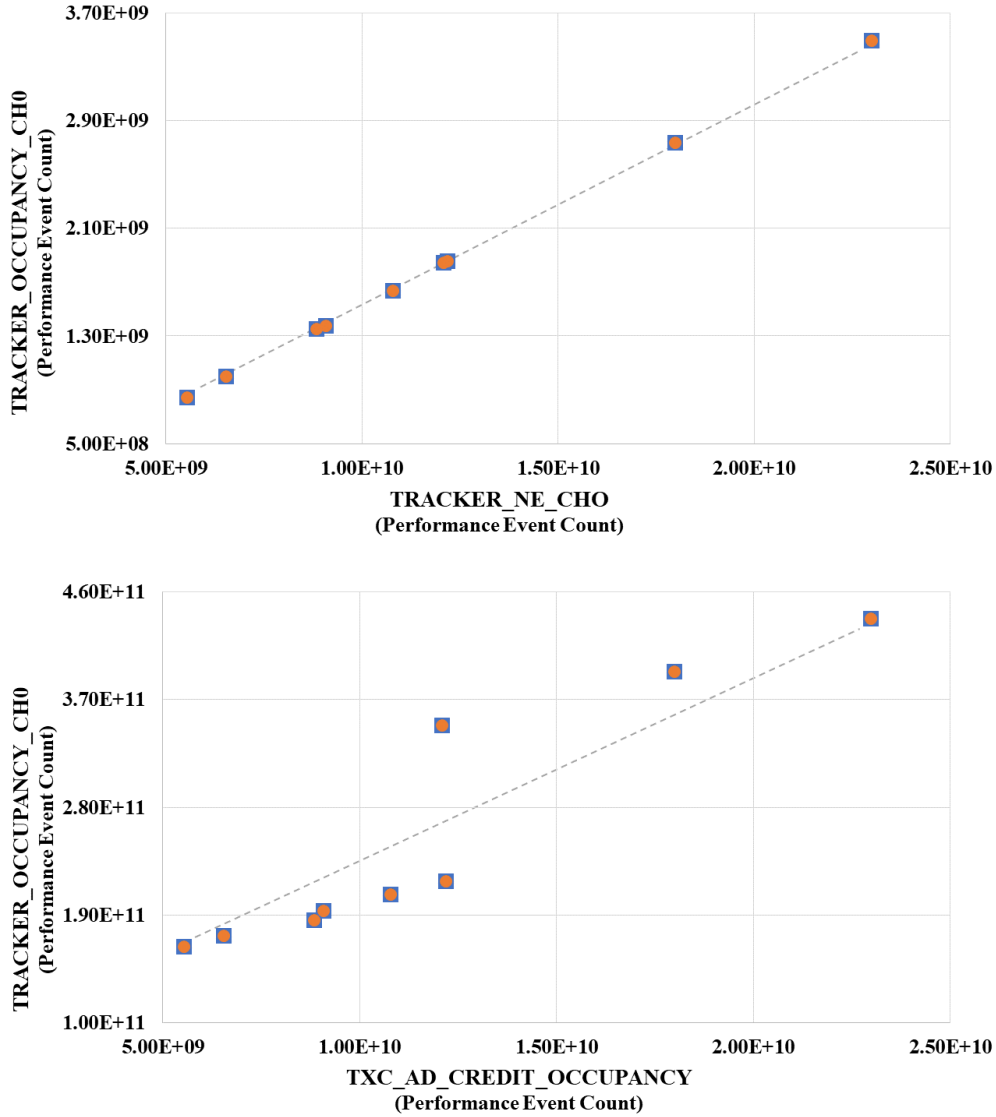


Figure 4.2: a). Correlation between performance events of one sub-group in the M2M (Mesh2Mem) group b). Correlation between performance events of two different sub-groups in the M2M group.

Furthermore, Figure 4.3 show the high positive correlation of the representative events with dynamic energy two sub-groups in CBOX and M2M. Figure 4.4 illustrates the result of the inter-group step in the second stage. It shows the lack of a high positive correlation between representatives of sub-groups in different performance monitoring counter groups.

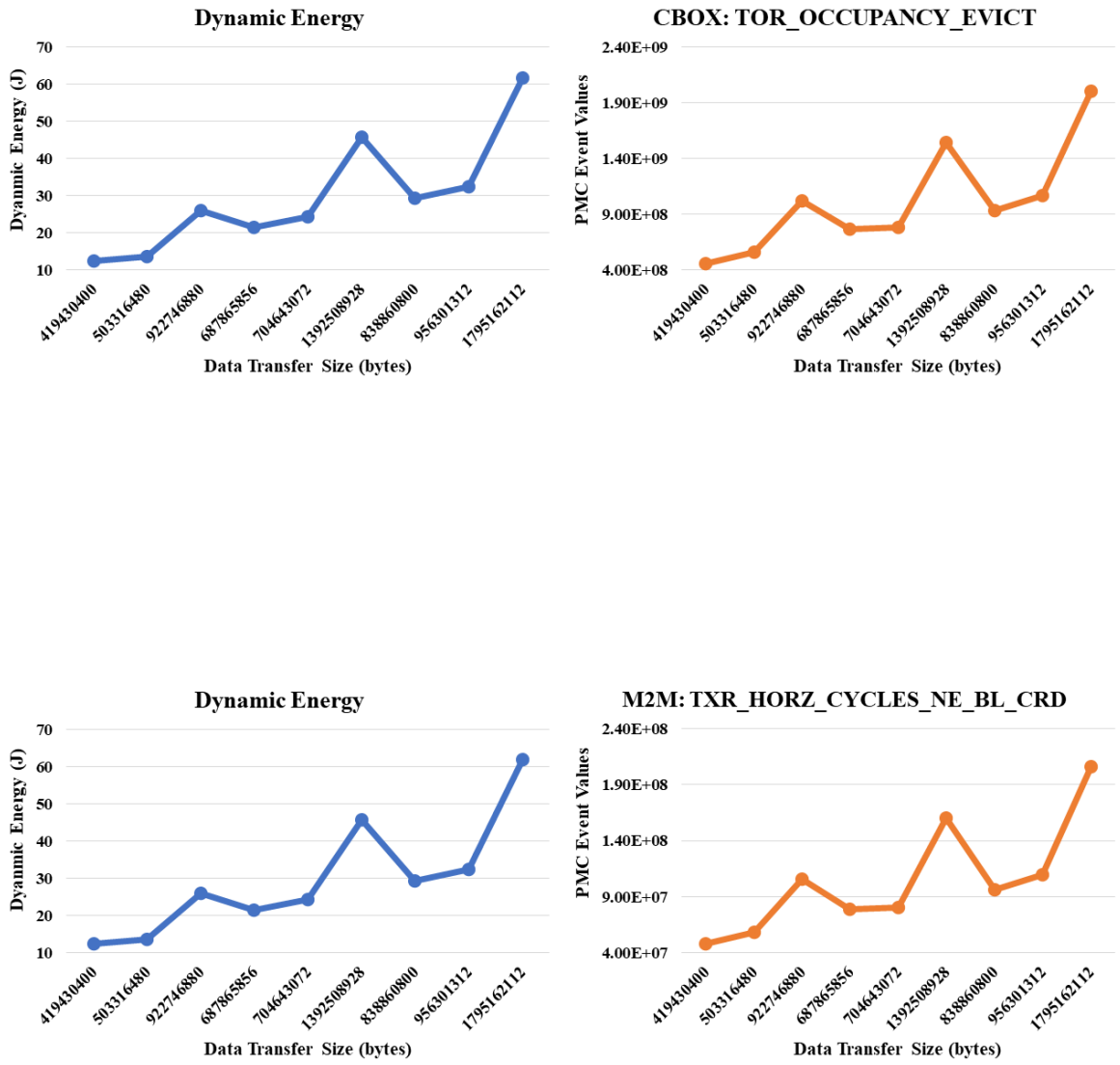


Figure 4.3: Correlation of the performance events with dynamic energy in the groups, CBOX and M2M. The performance events are highly positively correlated with dynamic energy and follow the same trend as the dynamic energy.

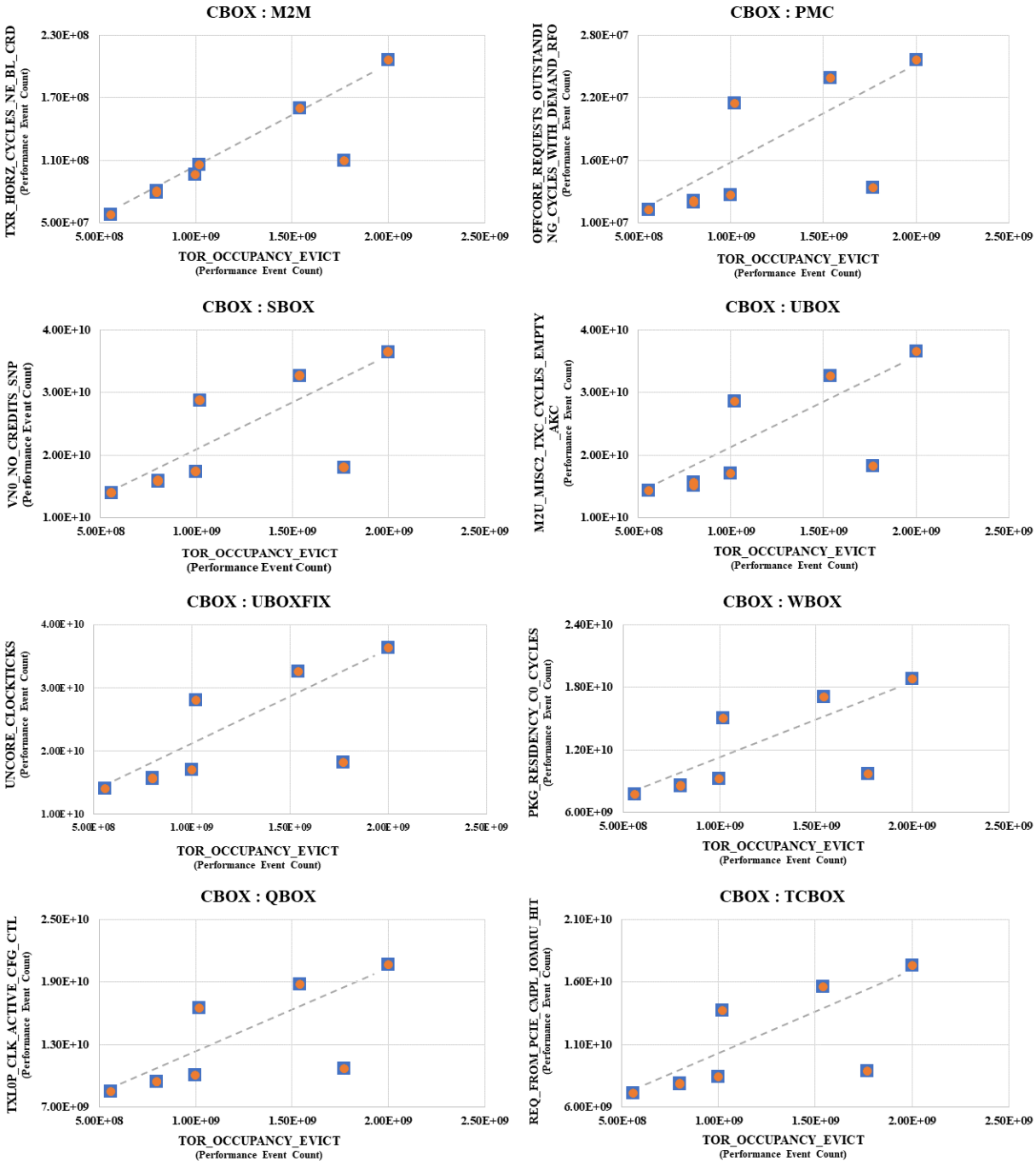


Figure 4.4: Correlation between representative performance events of sub-groups in different performance monitoring counter groups. For example, the first figure CBOX:M2M at the top refers to the correlation plot between representative performance events of sub-groups, TOR_OCCUPANCY_EVICT and TXR_HORZ_CYCLES_NE_BL_CRD, in CBOX and M2M, respectively.

Table 4.3 shows the final shortlist of highly additive and positively correlated performance event sets for data transfer activity in both directions for the heterogeneous server containing an Intel Haswell multicore CPU and Nvidia k40c GPU, as shown in Table 3.1. The number of performance events for the host CPU to K40c GPU and K40c GPU to host CPU directions are 9 and 10, respectively. The software power meters for the data transfer activity in K40c GPU server are given in the Table 4.4.

Table 4.5 shows the final shortlist of highly additive and positively correlated performance events for the data transfer activity in both directions for the heterogeneous server containing a Icelake multicore CPU and two Nvidia A40 GPUs as shown in Table 3.3. The number of performance events for the host CPU to A40 GPU_1 and A40 GPU_1 to host CPU directions are 6 and 9, respectively. Table 4.7 presents the software power meters for the Icelake A40 GPU server.

Furthermore, the shortlisted performance events are not further pruned in the third stage of the procedure by tuning $\mathbb{A}$ and ρ since they can be obtained in one application run.

The summary of our findings is presented below:

1. The performance events capture the off-core chip traffic. They represent the data traffic between the main memory and the caches, QPI traffic, and Power Control Unit (PCU) measurements.

2. The shortlisted performance events belong to different performance monitoring counter groups, and each group has enough dedicated performance monitoring counters to store the individual performance event counts. Therefore, the event counts can be obtained in one application run. This feature allows efficient deployment of our models employing these selected event sets at runtime.

Table 4.3: System-level performance event sets for data transfer in both directions for the heterogeneous server in Table 3.1 containing an Intel Haswell multicore CPU and Nvidia K40c GPU.

CPU $\Longrightarrow$ K40c GPU	
Group Name	**Performance Events**
MBOX	WR_CAS_RANK1_BANK14
PMC	L2_RQSTS_RFO_HIT
CBOX	TOR_INSERTS_NID_AL
QBOX	RXL0P_POWER_CYCLES
BBOX	RING_AK_USED_CW_EVEN
PWR	PWR_PKG_ENERGY
WBOX	WBOX_CLOCKTICKS
SBOX	TXR_INSERTS_AD_CRD
RBOX	RXR_INSERTS_DRS

K40c GPU $\Longrightarrow$ CPU	
Group Name	**Performance Events**
MBOX	RD_CAS_RANK0_BANK7
PMC	L2_TRANS_ALL_REQUESTS
CBOX	TOR_INSERTS_WB
QBOX	RXL_FLITS_G1_HOM
BBOX	RING_AK_USED_CW_ODD
PWR	PWR_DRAM_ENERGY
WBOX	WBOX_CLOCKTICKS
SBOX	TXR_INSERTS_AD_CRD
RBOX	RXR_INSERTS_DRS
UBOXFIX	UNCORE_CLOCK

Table 4.4: Data transfer software power meters implementing the linear dynamic energy predictive models for the heterogeneous server in Table 3.1 containing an Intel Haswell multicore CPU and Nvidia K40c GPU.

Model Description	**Linear Regression Model**
CPU $\rightarrow$ K40c GPU	4.69E-09 $\times$ WR_CAS_RANK1_BANK14 + 7.45E-08 $\times$ L2_RQSTS_RFO_HIT + 1.78E-07 $\times$ TOR_INSERTS_NID_AL + 00E0 $\times$ RXL0P_POWER_CYCLES + 3.84E-07 $\times$ RING_AK_USED_CW_EVEN + 1.81E-10 $\times$ PWR_PKG_ENERGY + 00E0 $\times$ WBOX_CLOCKTICKS + 00E0 $\times$ TXR_INSERTS_AD_CRD + 2.26E-07 $\times$ RXR_INSERTS_DRS
K40c GPU $\rightarrow$ CPU	4.69E-09 $\times$ RD_CAS_RANK0_BANK7 + 7.45E-08 $\times$ L2_TRANS_ALL_REQUESTS + 1.78E-07 $\times$ TOR_INSERTS_WB + 00E0 $\times$ RXL_FLITS_G1_HOM + 3.84E-07 $\times$ RING_AK_USED_CW_ODD + 1.81E-10 $\times$ PWR_DRAM_ENERGY + 00E0 $\times$ WBOX_CLOCKTICKS + 00E0 $\times$ TXR_INSERTS_AD_CRD + 2.26E-07 $\times$ RXR_INSERTS_DRS + 1.44E-09 $\times$ UNCORE_CLOCK

Table 4.5: System-level performance event sets for data transfer in both directions for the heterogeneous server in Table 3.3 containing a single-socket Icelake multicore CPU and Nvidia A40 GPUs.

CPU $\Longrightarrow$ A40 GPU_1	
Group Name	**Performance Events**
M2M	TRACKER_OCCUPANCY_CH0
PMC	OFFCORE_REQUESTS_OUTSTANDING _DEMAND_RFO
CBOX	LLC_LOOKUP_LOC_HOM
QBOX	UPI_CLOCKTICKS
SBOX	VN1_NO_CREDITS_NCB
TCBOX	TXN_REQ_OF_CPU_MEM _WRITE_IOMMU1
A40 GPU_1 $\Longrightarrow$ CPU	
M2M	TXR_HORZ_CYCLES_NE_BL_CRD
PMC	OFFCORE_REQUESTS_OUTSTANDING_CYCLES _WITH_DEMAND_RFO
CBOX	TOR_OCCUPANCY_EVICT
QBOX	TXL0P_CLK_ACTIVE_CFG_CTL
SBOX	VN0_NO_CREDITS_SNP
TCBOX	REQ_FROM_PCIE_CMPL_IOMMU_HIT
UBOXFIX	UNCORE_CLOCKTICKS
UBOX	M2U_MISC2_TXC_CYCLES_EMPTY_AKC
WBOX	PKG_RESIDENCY_C0_CYCLES

4.3 Training and Testing of Software Power Meters for Data Transfers

The software power meters based on linear dynamic energy predictive models that leverage normalized performance events as predictor variables, are built using specialized non-negative linear regression (Huber Regressor [64]). Based on the practical applications of the theory of energy predictive models in computing [47], our models maintain a zero intercept and non-negative regression coefficients, since dynamic energy must be zero when no hardware activity occurs, in accordance with the physical law of energy conservation.

We utilize the Huber Regressor from the Python's Scikit-learn package to construct these models. The Huber Regressor is a robust regression algorithm that is less sensitive to outliers compared to ordinary least squares (OLS) linear regression. The mathematical form of a model is shown below:

$$E_D = \beta_1 \times e_1 + \cdots + \beta_n \times e_n \tag{4.4}$$

where E_D is the dynamic energy consumption, which is the dependent variable, $\{e_1, \cdots, e_n\}$ are the performance events, and $\{\beta_1, ..., \beta_n\}$ are positive regression coefficients or the model parameters. The ground-truth dynamic energy measurements

obtained using external power meters are used as the reference values during the training process to determine the regression coefficients.

To train and test these models, we split the datasets into 70% for training and 30% for testing. These records are obtained by executing the base and compound applications employed in shortlisting the performance events, respectively.

Table 4.4 presents the data transfer software power meters implementing the linear energy predictive models for the two directions in the heterogeneous server described in Table 3.1, containing an Intel Haswell multicore CPU and an Nvidia K40c GPU. The average prediction errors for data transfers for the directions, CPU to K40c GPU and K40c GPU to CPU, in the Haswell k40c GPU server are 4% and 2%.

Table 4.7 presents the software power meters implementing the linear energy predictive models for the Icelake A40 GPU server shown in Table 3.3, containing an Intel Icelake multicore CPU and two Nvidia A40 GPUs. The average prediction errors of these software power meters are as follows: 1% for CPU computations, CPU to A40 GPU_1 and A40 GPU_1 to CPU, are 6% and 3%, respectively. The software power meters for GPU computations utilize NVML and their average prediction error is 5%.

Figures 4.5 and 4.6 compare the dynamic energy of serial data transfer predicted by the software power meters based on linear energy predictive model employing performance events as predictor variables against the *ground truth* employing power measurements using physical external power meters for the two heterogeneous hybrid servers in Tables 3.1 and 3.3.

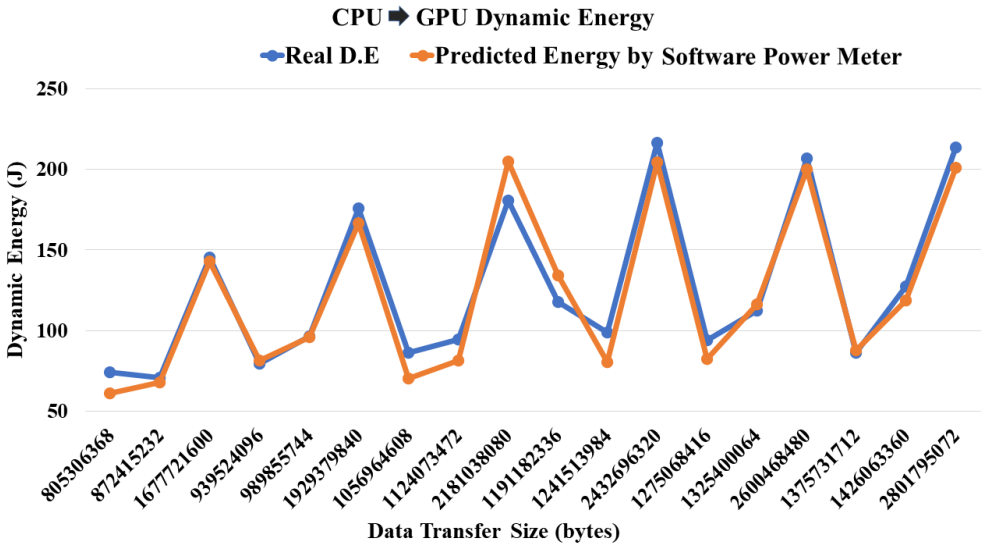
The prediction acccuracy is calculated as follows:

$$\text{Accuracy (\%)} = \frac{|\text{GroundTruth} - \text{Software Power Meter (Model)}|}{\text{GroundTruth}} \times 100 \tag{4.5}$$

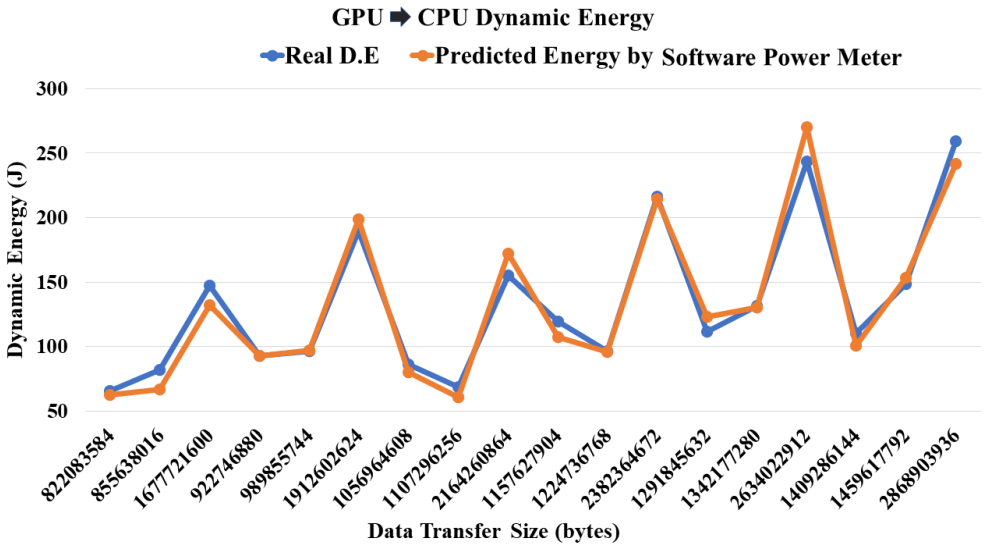
The summary of our findings are presented below:

1. The software power meters based on linear energy predictive models employ different sets of performance events for different data transfer directions in the Icelake A40 GPU server. However, these different software power meters predict the same dynamic energy consumption for data transfer of the same size, which is in full agreement with the groundtruth dynamic energy profiles that are almost the same for both directions for this server.

2. This finding signifies that different performance events (belonging to the same performance monitoring counter groups) are able to comprehensively capture the energy consumption activities of data transfers in the two directions.

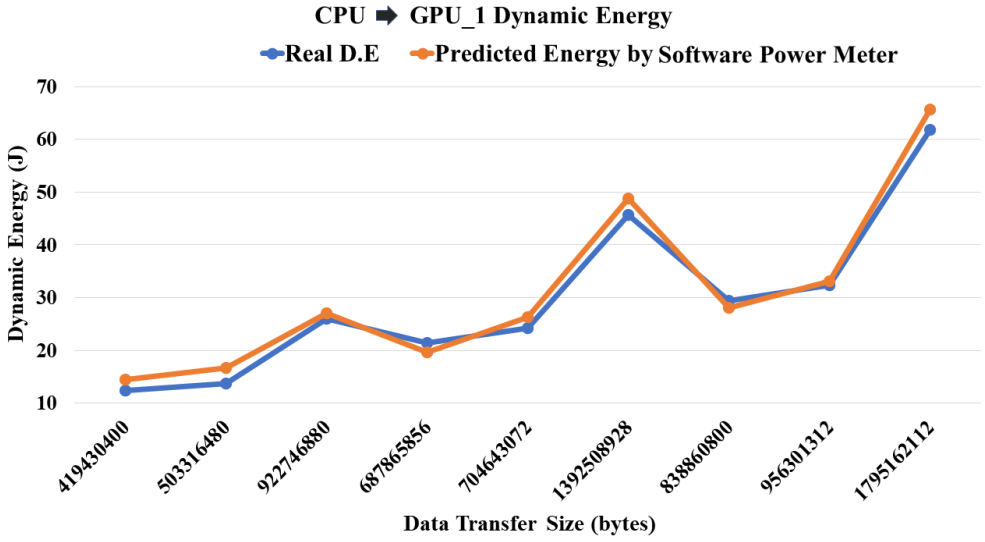


(a) Dynamic energy comparison of data transfers between pair of devices (CPU $\Longrightarrow$ K40c GPU) measured by ground-truth measurement method with the software power meter.

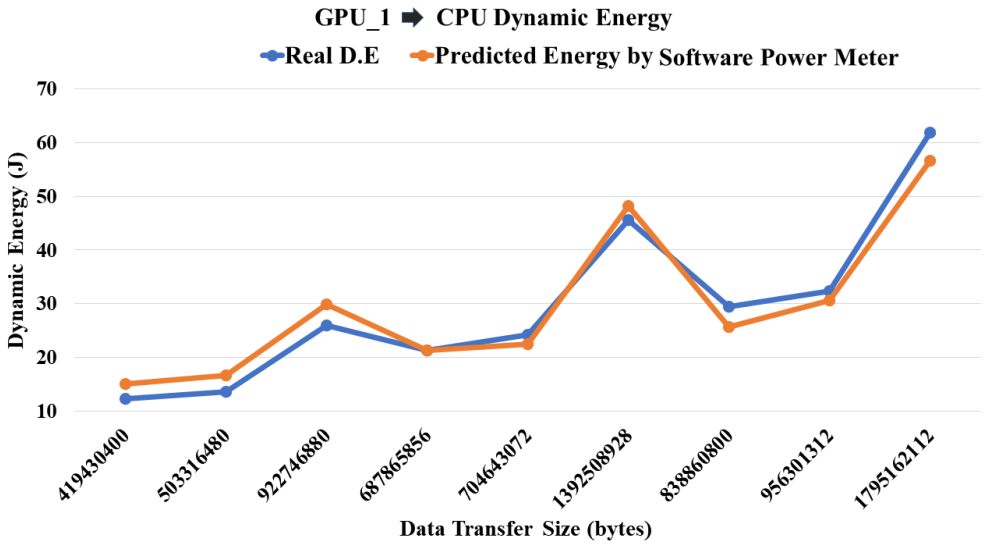


(b) Dynamic energy comparison of data transfers between pair of devices (K40c GPU $\Longrightarrow$ CPU) measured by ground-truth measurement method with the software power meter.

Figure 4.5: Dynamic energy comparison of data transfers between pair of devices (CPU,K40c GPU) and (K40c GPU,CPU) measured by ground-truth measurement method with the energy predicted by software power meters based on linear energy predictive model employing performance events as predictor variables for the heterogeneous hybrid server containing Intel Haswell multicore CPU as shown in Table 3.1.



(a) Dynamic energy comparison of data transfers between pair of devices CPU $\Longrightarrow$ A40 GPU_1 measured by ground-truth measurement method with the software power meter.



(b) Dynamic energy comparison of data transfers between pair of devices A40 GPU_1 $\Longrightarrow$ CPU measured by ground-truth measurement method with the software power meter.

Figure 4.6: Dynamic energy comparison of data transfers between pair of devices (CPU,A40 GPU_1) and (A40 GPU_1,CPU) measured by ground-truth measurement method with the energy predicted by software power meters based on linear energy predictive model employing performance events as predictor variables for the heterogeneous hybrid server containing Icelake multicore CPU shown in Table 3.3.

4.3.1 Software Power Meters for CPU Computation and Data Transfer activities

We employ our methodology comprising the fast selection procedure described earlier in section 4.2.2 to shortlist the performance events for predicting the dynamic energy of computation activity on the Intel Icelake multicore CPU as in Table 3.3. The shortlisted performance events for the Intel Icelake multicore CPU are presented in Table 4.6.

Table 4.6: System-level performance event sets for CPU computation for the heterogeneous server in Table 3.3 containing a single-socket Icelake multicore CPU and two Nvidia A40 GPUs.

Groups	Performance Events
TCBOX	IOMMU1_PWT_CACHE_LOOKUPS
PMC	CORE_POWER_LVL0_TURBO_LICENSE
M2M	TXC_AD_CREDIT_OCCUPANCY
SBOX	VN0_NO_CREDITS_SNP
QBOX	TXL0P_CLK_ACTIVE_DFX
CBOX	CBOX_CLOCKTICKS
UBOXFIX	UNCORE_CLOCKTICKS

Table 4.7: Software power meters implementing the linear dynamic energy predictive models for the heterogeneous server in Table 3.3 containing a single-socket Icelake multicore CPU and two Nvidia A40 GPUs.

Model Description	Linear Regression Model
CPU computations	2.24E-08 × IOMMU1_PWT_CACHE_LOOKUPS + 00E0 × CORE_POWER_LVL0_TURBO_LICENSE + 00E0 × TXC_AD_CREDIT_OCCUPANCY + 00E0 × VN0_NO_CREDITS_SNP + 4.69E-09 × TXL0P_CLK_ACTIVE_DFX + 00E0 × CBOX_CLOCKTICKS + 3.11E-09 × UNCORE_CLOCKTICKS
CPU → A40 GPU_1	00E0 × TRACKER_OCCUPANCY_CH0 + 3.27E-09 × OFFCORE_REQUESTS_OUTSTANDING_DEMAND_RFO + 00E0 × LLC_LOOKUP_LOC_HOM + 00E0 × UPI_CLOCKTICKS + 4.59E-10 × VN1_NO_CREDITS_NCB + 4.53E-09 × TXN_REQ_OF_CPU_MEM_WRITE_IOMMU1
A40 GPU_1 → CPU	9.10E10 × TXR_HORZ_CYCLES_NE_BL_CRD + 1.91E-08 × OFFCORE_REQUESTS_OUTSTANDING_CYCLES_WITH_DEMAND_RFO + 00E0 × TOR_OCCUPANCY_EVICT + 00E0 × TXL0P_CLK_ACTIVE_CFG_CTL + 1.84E-09 × VN0_NO_CREDITS_SNP + 00E0 × REQ_FROM_PCIE_CMPL_IOMMU_HIT + 00E0 × UNCORE_CLOCKTICKS + 2.06E-11 × M2U_MISC2_TXC_CYCLES_EMPTY_AKC + 2.3E-08 × PKG_RESIDENCY_C0_CYCLES

The software power meters for the server containing Icelake multicore CPU and Nvidia A40 GPUs are outlined in Table 4.7. The average prediction errors of these software power meters are as follows: 1% for CPU computation, Data Transfer of CPU to

A40 GPU_1 and A40 GPU_1 to CPU, are 6% and 3%, respectively. The software power meters for GPU computations utilize NVML, and their average prediction error is 5%. Table C.4 in appendix shows the list of applications employed for training and testing the energy predictive model of computation for the Intel Icelake multicore CPU.

4.4 Experimental Validation of Software Power Meters

We experimentally determine the accuracy of our software power meters based on linear energy predictive models that levarage performance events as predictor variable, using three highly optimized scientific parallel applications on our Icelake A40 GPU server, as shown in Table 3.3.

The three heterogeneous parallel applications are Matrix Addition (HDGEADD), Matrix Multiplication (HDGEMM), and 2D fast Fourier Transform (2D-HFFT). Briefly, a hybrid parallel application comprises several software components (kernels) executing in parallel. A one-to-one mapping exists between the software components and computing devices of the hybrid server on which the application is executed.

Executing an accelerator component involves a dedicated CPU core, running the hosting thread, and the accelerator itself, performing the accelerator code. The execution of the accelerator component includes data transfer between the CPU and accelerator memory, computations by the accelerator code, and data transfer between the accelerator memory and CPU. Executing a CPU component involves only the CPU cores performing the multithreaded CPU code.

Each hybrid application has three software components: CPU, A40 GPU_1, and A40 GPU_2. CPU_1 comprises 22 (out of the total of 24) CPU cores. A40 GPU_1 component represents the first Nvidia A40 GPU, and a host CPU core is connected to this GPU via a dedicated PCI-E link. Similarly, A40 GPU_2 denotes the second Nvidia A40 GPU and a host CPU core connected to this GPU via a dedicated PCI-E link.

An activity in a software component pertains to either computations within the component or the data transfer between a pair of communicating computing devices. For each hybrid parallel application, the following activities are carried out:

1. Data transfer from CPU to A40 GPU_1.
2. Data transfer from CPU to A40 GPU_2.
3. Computation on CPU.
4. Computation on A40 GPU_1.
5. Computation on A40 GPU_2.

6. Data transfer from A40 GPU_1 to the CPU.

7. Data transfer from A40 GPU_2 to the CPU.

All activities during the execution of the parallel applications are executed serially, except for the computation activities (3 to 5), which are executed in parallel across the CPU, A40 GPU_1, and A40 GPU_2.

Seven software energy power meters are deployed in each parallel application: three for computations activities, in the CPU, A40 GPU_1, and A40 GPU_2 software components, and four for data transfers activities from the host CPU to both A40GPUs and both A40GPUs to host CPU. The energy of data transfers is predicted by our software power meters. The energy of computations in the software component involving the multicore CPU is also obtained using the software power meter presented in Table 3.3.

Our software power meters for computations on the A40 GPUs employ the power readings provided by the on-chip power sensors in the GPUs obtained programmatically using the Nvidia Management Library (NVML) [28] interface. The experimental error of the NVML sensors is 5%. Furthermore, each hybrid application is instrumented with our software power meters for computations and data transfers activities using the software power meter API. Section B.2 in the appendix illustrates the instrumentation process for the matrix multiplication application. The procedure is similar for the other two applications.

We employ the same matrix sizes for all the applications, starting from 30720 x 30720 to 31200 x 31200 with a stepsize of 16.

For each matrix size, we determine the following:

- Estimated and actual dynamic energy of data transfer activities between pairs of devices by employing software power meters and the *ground-truth* method.
- Estimated and actual dynamic energy of computation activities on the multicore CPU by employing the software power meter and the *ground-truth* method.
- Estimated and actual dynamic energy of computations on the two GPUs using the software power meter and the *ground-truth* method.
- The total dynamic energy consumption during the application execution using the *ground-truth* method.
- The sum of the estimated dynamic energy consumption of computation activities and data transfer activities during the application execution using our software power meters.

4.4.1 Matrix Addition (HDGEADD)

Figure 4.7 illustrates the hybrid parallel matrix addition application (HDGEADD) computes the matrix addition ($C = A + B$) of two dense square matrices, A and B, of size $N \times N$. Each software component *i* is assigned a number of rows of A, B and C that is provided as an input parameter to the application. The matrices A, B and C are horizontally partitioned such that each software component is assigned several contiguous rows of A, B and C that is provided as an input parameter to the application. The application comprises 3 software components executed in parallel (one CPU component, one GPU_1 component, and one GPU_2 component). Each software component *i* computes the matrix addition, $C_i = A_i + B_i$.

The application comprises five main stages. The first stage consists of data transfers of A_2, B_2, and C_2 from the host CPU to A40 GPU_1. The second stage consists of data transfers of A_3, B_3, and C_3 from the host CPU to A40 GPU_2. The third stage involves local parallel computations in the three software components. The computations in the software component involving the multicore CPU are performed using the Intel MKL (DGEADD) library routine. The computations in the software components involving the A40 GPUs are performed using the CUBLAS library routine. The fourth and fifth stages involves data transfer of the result matrices C_2 and C_3 from A40 GPU_2 and A40 GPU_1 to the host CPU.

Table 4.8 shows the average prediction error of the seven software energy power meters deployed in the matrix addition application for several matrix sizes. Figure 4.16 shows the total dynamic energy estimated by our software power meters versus the *ground-truth* energy for the matrix addition (HDGEADD) application.

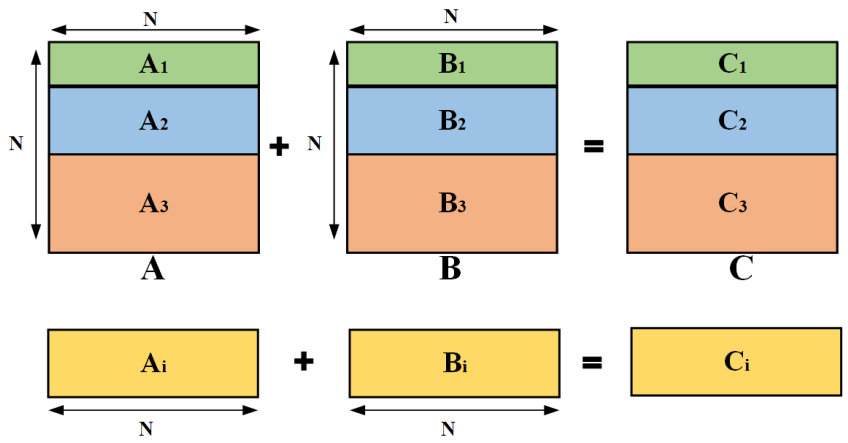


Figure 4.7: Parallel matrix addition application (HDGEADD) computing the matrix addition ($C = A + B$) of two dense square matrices A and B of size $N \times N$.

Table 4.8: Comparison between the estimated energy consumption provided by software power meters and the ground-truth method based on power measurements using external power meters for HDGEADD application.

Description	Avg. Prediction Error
CPU computations	1%
A40 GPU_1 computations	5%
A40 GPU_2 computations	5%
CPU $\rightarrow$ A40 GPU_1 and A40 GPU_2 data transfer	6%
A40 GPU_1 and A40 GPU_2 $\rightarrow$ CPU data transfer	3%

4.4.2 Matrix Multiplication (HDGEMM)

Figure 4.8 illustrates the hybrid parallel matrix multiplication application (HDGEMM) computing the matrix product ($C+ = A \times B$) of two dense square matrices A and B of size $N \times N$. Each software component *i* is assigned a number of rows of A, B and C that is provided as an input parameter to the application. The application comprises 3 software components executed in parallel (one CPU component, one GPU_1 component, and one GPU_2 component). All the components share the matrix B.

Each software component is assigned several contiguous rows of A and C provided as an input parameter to the application. The CPU software component is assigned N_1 rows of A and C. The A40 GPU_1 software component is assigned N_2 rows of A and C. Finally, the A40 GPU_1 software component is assigned the remaining $(N - N_1 - N_2)$ number of rows of A and C. Each software component *i* computes the matrix product, $C_i = A_i \times B$.

The application comprises five main stages. The first stage consists of data transfers of A_2, B, and C_2 from the host CPU to A40 GPU_1. The second stage consists of A_3, B, and C_3 from the host CPU to A40 GPU_2. The third stage involves local parallel computations in the software components. The computations in the CPU software component

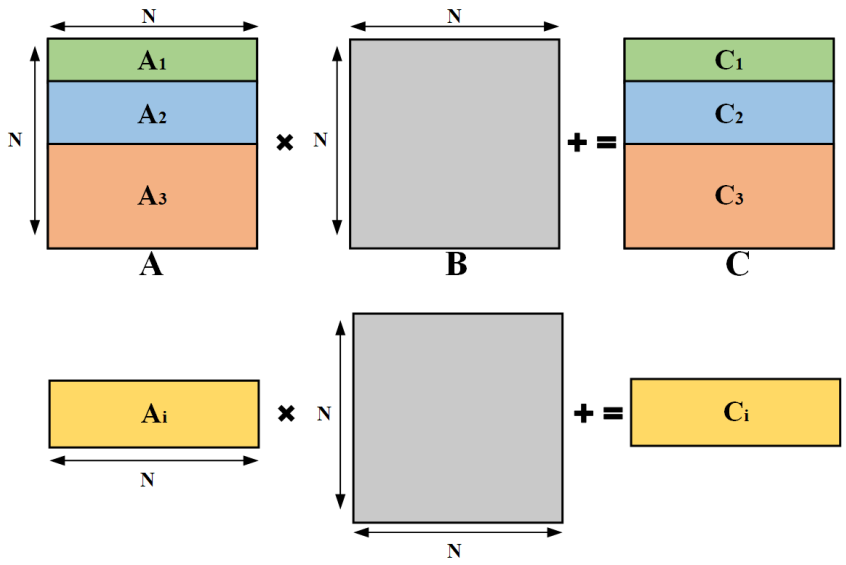


Figure 4.8: Matrix partitioning between the software components in hybrid parallel matrix multiplication application (HDGEMM) computing the matrix product ($C+ = A \times B$) of two dense square matrices A and B of size $N \times N$.

Table 4.9: Comparision between the estimated energy consumption provided by software power meters and the ground-truth method based on power measurements using external power meters for HDGEMM application.

Description	**Avg. Prediction Error**
CPU computations	1%
A40 GPU_1 computations	5%
A40 GPU_2 computations	5%
CPU $\rightarrow$ A40 GPU_1 and A40 GPU_2 data transfer	6%
A40 GPU_1 and A40 GPU_2 $\rightarrow$ CPU data transfer	3%

are performed using the Intel MKL DGEMM library routine. The computations in the software components involving the A40 GPUs are performed using the CUBLAS DGEMM library routine. The forth and fifth stages involves data transfer of the result matrices C_2 and C_3 from A40 GPU_2 and A40 GPU_1 to the host CPU respectively.

Table 4.9 shows the average prediction error of the seven software energy power meters deployed in the matrix multiplication application for several matrix sizes. Figure 4.17 shows the total dynamic energy estimated by our software power meters versus the *ground-truth* energy for the matrix multiplication (HDGEMM) application.

4.4.3 2D Fast Fourier Transform (HFFT)

Figure 4.9 illustrates the hybrid parallel 2D fast Fourier Transform application (HFFT) computing the 2D-FFT of a signal matrix S of size $N \times N$. The application comprises three software components executed in parallel (one CPU component, one GPU_1 com-

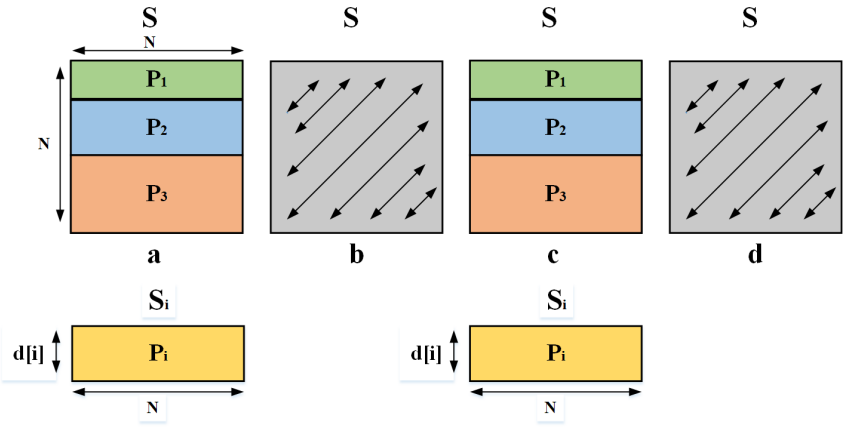


Figure 4.9: Hybrid parallel 2D fast Fourier Transform application (HFFT) computing the 2D-FFT of a signal matrix S of size $N \times N$.

Table 4.10: Comparision between the estimated energy consumption provided by software power meters and the ground-truth method based on power measurements using external power meters for HFFT application.

Description	**Avg. Prediction Error**
CPU computations	1%
A40 GPU_1 computations	5%
A40 GPU_2 computations	5%
CPU $\rightarrow$ A40 GPU_1 and A40 GPU_2 data transfer	6%
A40 GPU_1 and A40 GPU_2 $\rightarrow$ CPU data transfer	6%

ponent, and one GPU_2 component). The matrix S is partitioned such that each software component is assigned several contiguous rows of S provided as a parameter to the application. The 2D FFT is accomplished in four steps: (1) Computing 1D FFTs on *N* rows of the signal matrix N. (2) Transpose of the signal matrix. Steps (3) and (4) are the repetition of steps (1) and (2), respectively.

The application comprises twelve stages. The first stage consists of data transfers of P_2 from the host CPU to A40 GPU_1. The second stage consists of data transfers of P_3 from the host CPU to A40 GPU_2. The third stage involves local parallel computations in the software components. The computations in the software component involving the multicore CPU are performed using the Intel MKL FFT library routine. The computations in the software components involving the A40 GPUs are performed using the CUFFT library routine. The fourth and fifth stage involves data transfer of the result matrices P_2 and P_3 from A40 GPU_2 and A40 GPU_1 to the host CPU respectively. The sixth stage involves the transpose of the matrix S performed on the CPU side. The six stages are repeated. We do not consider the energy consumption of matrix transposition on the CPU since it is insignificant.

Table 4.10 shows the average prediction error of the seven software energy power

meters deployed in the 2D fast Fourier Transform application for several matrix sizes. Figure 4.18 shows the total dynamic energy estimated by our software power meters versus the *ground-truth* energy for the 2D fast Fourier Transform application.

4.4.4 Comparison Between The Energy of Computations and Data Transfers Activities

Figures 4.10, 4.11 and 4.12 compare the dynamic energy of CPU computation activities estimated by software power meters against the ground-truth dynamic energy profiles for the three parallel applications matrix addition, matrix multiplication and 2D fast Fourier transform, executed on a heterogeneous hybrid server shown in Table 3.3. The average prediction error is 1%.

Figures 4.13, 4.14 and 4.15 shows the dynamic energy consumption of computation and data transfer activities for various matrix sizes employed in three parallel applications, matrix addition, matrix multiplication and 2D fast Fourier transform, executed on a heterogeneous hybrid server shown in Table 3.3. The dynamic energy consumption is measured using the ***ground-truth*** method.

The dynamic energy of data transfers activity is significant compared to the computations activity on the multicore CPU processor for all three applications, contravening the widespread notion that data transfers consume negligible dynamic energy.

Figures 4.16, 4.17, and 4.18 present the total dynamic energy measured using the *ground-truth* method compared with the sum of the estimated dynamic energy consumption of computation and data transfer activities predicted by our software power meters for the matrix addition (HDGEADD), matrix multiplication (HDGEMM), and 2D fast Fourier Transform (HFFT) applications across various matrix sizes.

The results demonstrate that the software power meters achieve high accuracy, with an average prediction error of 1% for computation activity and 6% for data transfer activity across all three parallel applications.

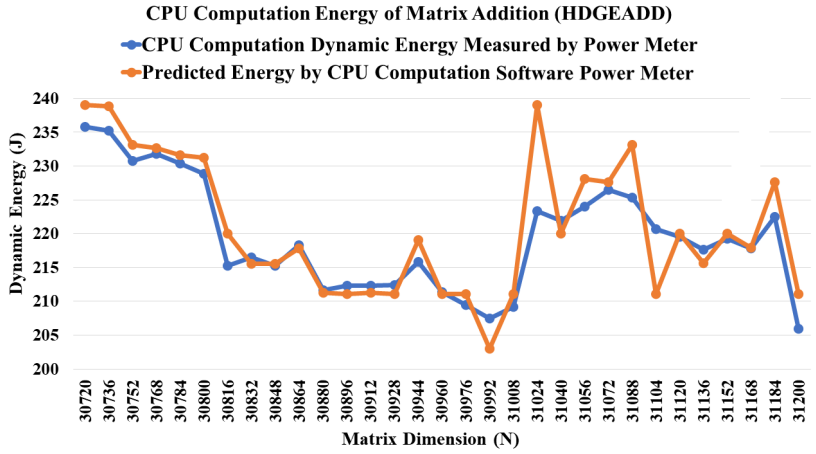


Figure 4.10: Ground-truth dynamic energy profiles versus the dynamic energy of computation activity estimated by the software power meter of CPU computation for the hybrid Matrix Addition (HDGEADD) application.

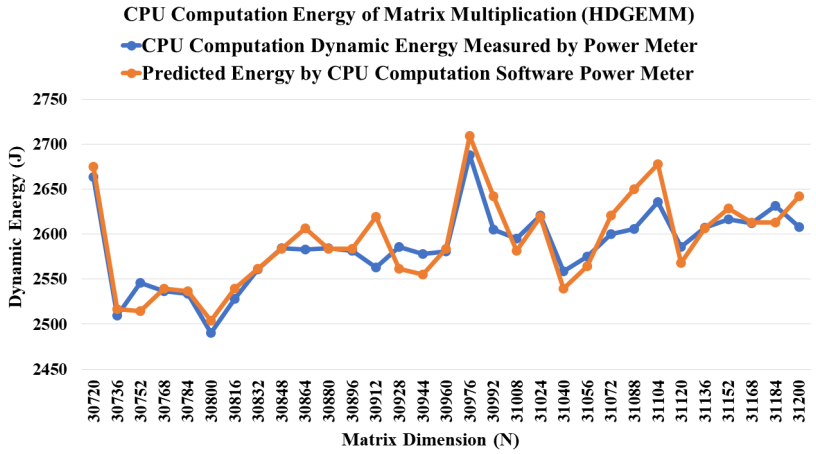


Figure 4.11: Ground-truth dynamic energy profiles versus the dynamic energy of computation activity estimated by the software power meter of CPU computations for the hybrid Matrix Multiplication (HDGEMM) application.

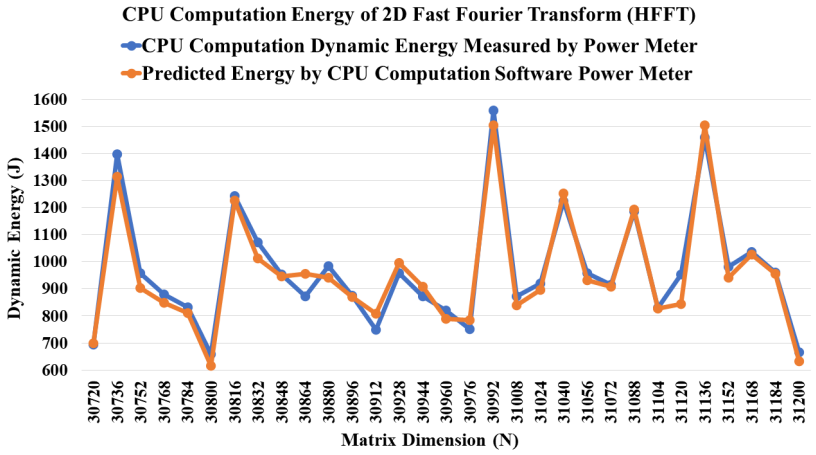


Figure 4.12: Ground-truth dynamic energy profiles versus the dynamic energy of computation activity estimated by the software power meters of CPU computation for the hybrid Fast Fourier Transform (HFFT) application.

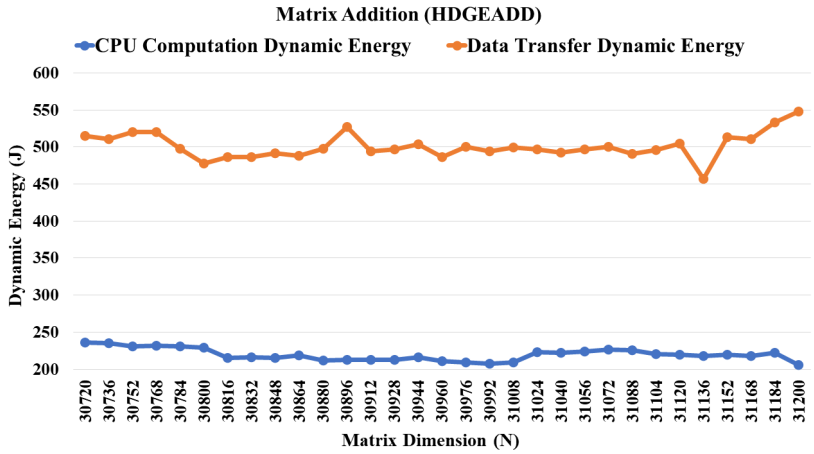


Figure 4.13: Comparison between dynamic energies of computations and data transfers between CPU and A40 GPUs for the hybrid Matrix Addition (HDGEADD) application.

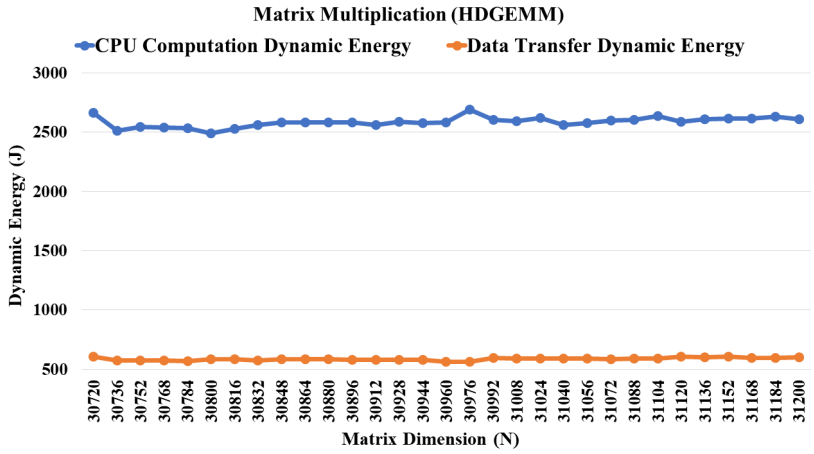


Figure 4.14: Comparison between dynamic energies of computations and data transfers between CPU and A40 GPUs for the hybrid Matrix Multiplication (HDGEMM) application.

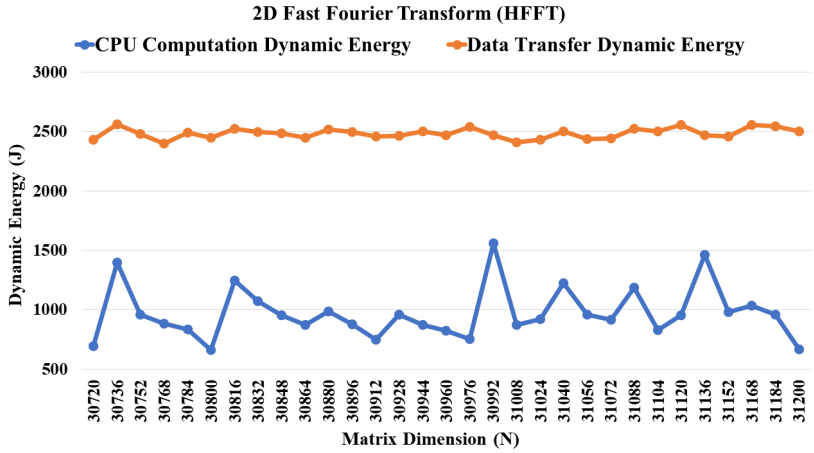


Figure 4.15: Comparison between dynamic energies of computations and data transfers between CPU and A40 GPUs for the hybrid 2D Fast Fourier Transform (HFFT) application.

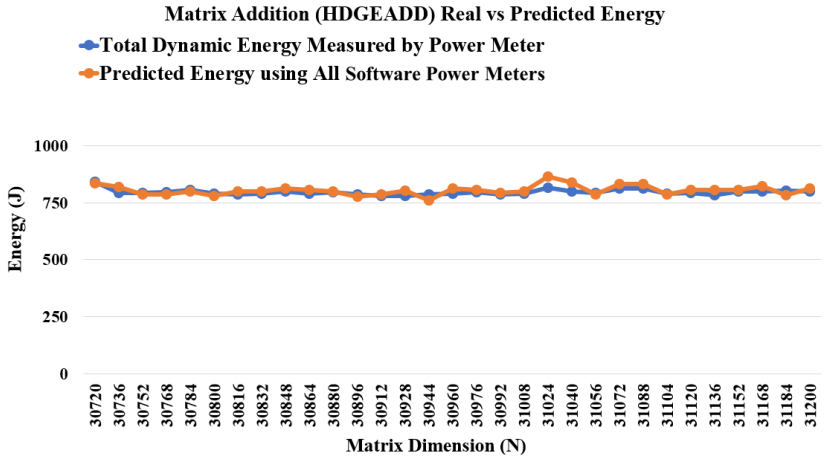


Figure 4.16: Ground-truth dynamic energy profiles versus the total dynamic energy estimated by software power meters for data transfers activity and computations activity for the matrix addition (HDGEADD) application.

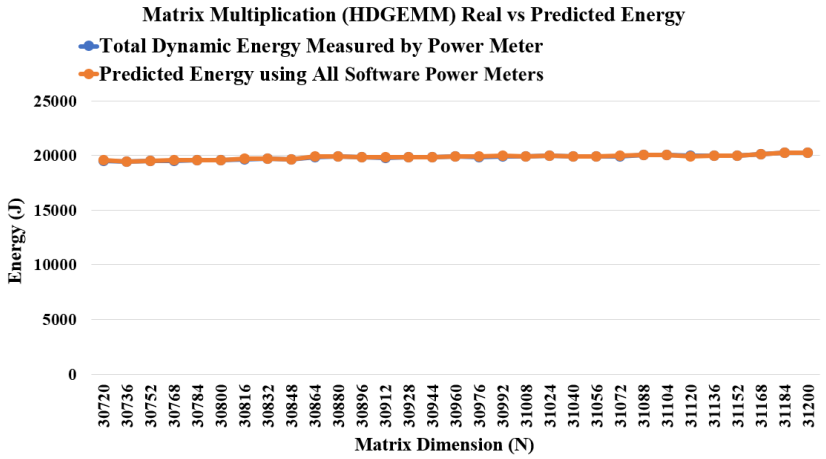


Figure 4.17: Ground-truth dynamic energy profiles versus the total dynamic energy estimated by software power meters for data transfers activity and computations activity for the matrix multiplication (HDGEMM) application.

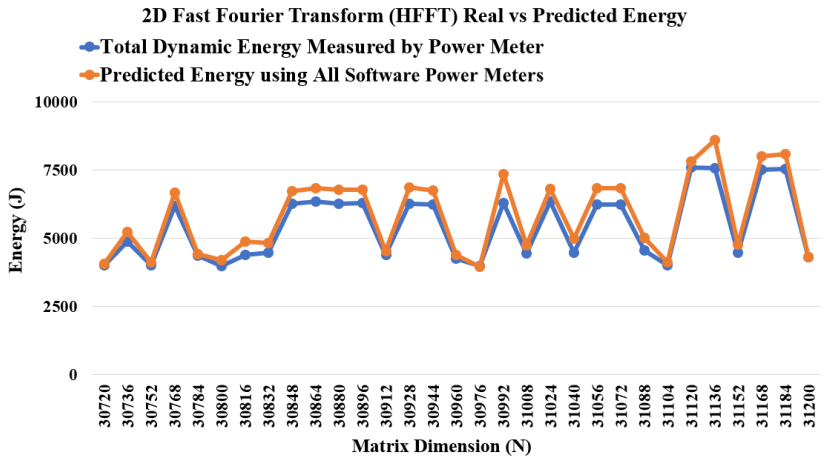


Figure 4.18: Ground-truth dynamic energy profiles versus the total dynamic energy estimated by software power meters for data transfers activity and computations activity for the 2D fast Fourier Transform (HFFT) application.

4.5 Summary

Given the limitations of the first two mainstream energy measurement methods, the third approach, energy predictive models that employ performance events as predictor variables, has emerged as a promising alternative for developing software power meters for runtime energy profiling compared to other mainstream methods.

However, the number of performance events is significant, and selecting a small subset of performance events that can effectively capture all energy consumption activities during a data transfer, while being collected in a single application run, is a challenging problem. To address this, we proposed a novel methodology that employs a fast selection procedure based on the natural grouping of performance events derived from the processor architecture, the additivity test, and high positive correlation to develop a linear energy predictive model for predicting the energy consumption of a single activity.

Building upon this methodology, we designed and developed reliable system-level software power meters based on linear energy predictive models that leverage performance events as predictor variables to estimate the energy of data transfer activity in both host-to-device and device-to-host directions on heterogeneous hybrid servers.

We developed independent, runtime software power meters based on the proposed linear energy predictive models that leverage performance events as predictor variables to estimate the energy consumption of a single activity whether it is data transfer or computation. These software power meters utilize selected performance events to separately estimate the dynamic energy consumption of computations activity and the dynamic energy consumption of data transfer activities during the execution of heterogeneous hybrid applications, enabling independent and accurate profiling of each component.

We validated our methodology and evaluated the accuracy of our software power meters by using three parallel scientific applications on our heterogeneous server, which contain an Intel Icelake multicore CPU and two Nvidia A40 GPUs, comparing ground-truth energy profiles with the energy predicted by software power meters. The results show that the software power meters achieve high accuracy with an average prediction error of 1% for computation activity, and 6% for data transfer activity across all three parallel applications.

Chapter 5

Concurrent and Orthogonal Software Power Meters on Heterogeneous Hybrid Servers

5.1 Introduction

State-of-the-art software power meters have previously been developed using linear energy predictive models that employ shortlisted performance events. These software power meters estimate the dynamic energy consumption of a single activity, either computation or data transfer, during the execution of a parallel hybrid program, and the sum of the estimations of all computation and data transfer activities can be used as an estimation of the total dynamic energy consumption by the program.

However, these models and software power meters were developed and validated under the assumption that no data transfer activity is performed in parallel with other data transfer or computation activities, which is a serious limitation on the class of programs, where the developed meters can be employed.

Therefore, the proposed energy predictive models are not suitable for employment in software power meters for arbitrary parallel hybrid programs as they lack two fundamental properties: *concurrency* and *orthogonality*.

Concurrency of software power meters refers to their capability to simultaneously collect their respective sets of performance events capturing accurately and reliably the concurrent execution of the sensed computation and communication activities. This means that software power meters can gather all performance events at once during a single application run. Furthermore, the performance event sets of these software power meters do not have to be disjoint; in theory, two concurrent software power meters can share an event and read it independently at different points during an application run.

On the other hand, *orthogonality* pertains to a pair of software power meters measuring two independently powered activities and means that the software power meters can accurately and reliably measure the energy of these activities whether they happen serially or in parallel. For two software power meters to be considered orthogonal, they must only sense and measure their own activities and remain unaffected by the activities monitored by their counterpart. An essential condition for achieving orthogonality is that the sets of performance events of the software power meters must be disjoint.

We will now further elaborate on the reasons why the state-of-the-art models are unsuitable.

First, the design and implementation of the previously developed methodology identifies a small subset of performance events for a single activity only: CPU computations, data transfers from the CPU to GPU, or data transfers from GPU to the CPU. These sets of performance events are then used to create accurate linear energy predictive models for each activity. The methodology does not account for the simultaneous execution of computation and communication activities.

Second, the methodology will likely find non-orthogonal sets, where the selected performance events are not disjoint. Even if the sets are orthogonal, they may remain inaccurate for runtime energy profiling of parallel computation and communication activities, since the methodology does not consider their concurrent execution. As a result, the performance event sets do not accurately reflect runtime energy profiling for parallel or concurrent execution of computation and communication activities in parallel hybrid programs.

To summarize, state-of-the-art research can be divided into two main categories:

a) Studies that propose energy predictive models for CPUs and GPUs. These models can function as practical software power meters for estimating the energy consumption of concurrent execution in CPU and GPU computational activities, but not for communication activities; and

b) Studies that utilize a methodology to construct accurate energy predictive models for serial computations and data transfers, while overlooking the critical property of orthogonality in their design and implementation.

In this chapter, we define the fundamental properties such as concurrency and orthogonality of software power meters that are essential for accurately profiling runtime energy consumption of parallel hybrid programs on heterogeneous hybrid servers. We then present a methodology for developing concurrent and orthogonal software power meters that provide accurate runtime estimation of energy consumption associated with independently powered parallel computation and communication activities in parallel hybrid programs.

We employ the methodology to develop concurrent and orthogonal software power meters for three heterogeneous hybrid servers that feature Intel multicore CPUs and Nvidia GPUs from different generations. These software power meters implement system-level linear energy predictive models that employ disjoint sets of performance events, ensuring accurate and scalable runtime energy profiling across diverse architectures.

For each server, this methodology finds software power meters for computations taking place in the multicore CPU, computations in a GPU, and data transfers between the CPU and GPU (one power meter for each direction). A crucial aspect of this methodology's success in finding concurrent and orthogonal software power meters for both computations and data transfers is that modern processor architectures provide numerous redundant performance events that capture both on-core and off-core activities. While the data transfer power meters include performance events associated with off-core chip traffic, the power meters for computations primarily utilize core-local performance events belonging to the Performance Monitoring Counter (PMC) group.

We demonstrate the accuracy and efficiency of our proposed concurrent and orthogonal software power meters through runtime energy profiling of three parallel hybrid programs on the three hybrid servers. The average prediction error for total dynamic energy consumption on our servers is only 2.5%. Our software power meters can accurately monitor all three hybrid programs because they are designed for system-level use rather than being tailored to individual applications. As a result, no modifications to the power meters are necessary to capture energy consumption activities specific to any application.

This chapter is organized as follows. Section 5.2 defines the two fundamental properties, concurrency and orthogonality of software power meters. Section 5.3 details the methodology to construct the concurrent and orthogonal software power meters based on linear energy predictive models that leverage disjoint sets of performance events as predictor variables. In section 5.4, the development of concurrent and orthogonal software power meters for three heterogeneous hybrid servers from different generations is explained. The section 5.5 discusses experiments demonstrating the high accuracy and efficiency of these concurrent and orthogonal software power meters in profiling runtime energy consumption for three parallel hybrid programs across the three heterogeneous hybrid servers. Finally, Section 5.6 provides the summary and concludes the chapter.

5.2 Concurrency and Orthogonality of Software Power Meters

We will begin by defining the concept of independently powered activities within a parallel hybrid program. This definition is crucial for understanding the intuition behind concurrent and orthogonal software power meters. A parallel hybrid program running on a heterogeneous hybrid server at any moment of time comprises one or several simultaneously running activities. The types of activities include:

1. Computations on the CPU cores, excluding the cores that are dedicated to host GPUs and connected to them via PCIe links.
2. Computations on a GPU.
3. Data transfer (or communication) between the CPU and a GPU in either forward or reverse direction

We define a pair of activities as being independently powered if the total dynamic energy consumption of the pair equals the sum of the dynamic energy consumption of each individual activity, regardless of whether or not the activities are running simultaneously. Furthermore, a set of activities is considered independently powered if each pair of activities within that set is independently powered.

To clarify the concept of independently powered activities, we examine pairs of such activities in the execution of a parallel hybrid program on the heterogeneous hybrid server used in our experiments, Skylake GPU server, which consists of a single-socket Intel multicore CPU and an Nvidia P100 GPU. In this setup, CPU and GPU computation activities are found to be independently powered.

Additionally, a communication activity between a host CPU core and the GPU and CPU computation activity are also independently powered. It is important to note that the host CPU core is not shared between these two activities, allowing us to treat them as independently powered. However, parallel communication activities between the CPU and GPU whether in forward or reverse directions, or both, are not independently powered, as they share the same power source.

Consider the execution of a parallel hybrid program on a heterogeneous hybrid server equipped with a dual-socket multicore CPU and two GPUs: the first GPU has affinity to the first CPU socket, while the second GPU has affinity to the second CPU socket. In this scenario, a pair of communication activities involving different GPUs will be independently powered, regardless of the direction of data transfer. Now consider a class of parallel hybrid programs where all activities in any program in this class are independently powered on a given server.

Our goal is to develop a set of software power meters, with one dedicated to each activity, that would correctly and accurately measure the dynamic energy consumed by these activities in any program within this class. The power meters will function correctly if and only if the following properties are satisfied:

1. All power meters must be able to run in parallel.
2. If a specific activity does not occur during the measured time interval, the power meter designed to measure this activity will return zero (or negligible) dynamic energy consumption.
3. If a specific activity is running during a given time interval, the power meter designed to measure this activity must accurately report the dynamic energy consumed by this activity during that time interval.

If these properties are satisfied, the software power meters that are implemented as linear dynamic energy predictive models using normalized performance events as predictor variables, will provide accurate measurements of dynamic energy consumption for each activity during any time interval of the program's execution. Furthermore, the sum of the measured values will equal the total dynamic energy consumed by the program within that time interval.

Now, we formulate certain properties that these software power meters should have in relation to the performance events. Software power meters that meet these properties will also satisfy properties (1) to (3), ensuring their correctness.

The properties are outlined as follows:

(a) If only one activity is running during the execution of a program, the software power meter designed to measure this activity will return the accurate dynamic energy consumed by this activity. Moreover, each performance event of the software power meter will have the same (or almost the same) count for different runs of the program.

(b) All performance events used by the software power meters can be collected in parallel.

(c) Software power meters do not share performance events, meaning each power meter has a unique set of performance events used as predictors.

(d) If a specific activity does not occur within the measured time interval, all performance events of the software power meter designed to measure this activity will have zero (or negligible) counts.

(e) Consider two programs, each running a different activity, A and B. In this case, each performance event of the software power meters designed to measure these activities will return the same (or almost the same) count, regardless of the order in which the programs are executed.

These properties are formulated to be directly testable through experiments.

A set of software power meters satisfying property (b) is called *concurrent*.

A set of software power meters fulfilling properties (c), (d) and (e) is termed *orthogonal*.

5.2.1 Intuitive Explanation and Running Example

This section provides an intuitive explanation of concurrency and orthogonality using a simple running example. Consider a parallel hybrid program that executes the following stages: a host to device data transfer from the CPU to GPU, then computations on the CPU and GPU in parallel, and finally a device to host data transfer from the GPU to the CPU. During the first stage, only the data transfer activity is active. During the second stage, CPU computations and GPU computation activities are active simultaneously. During the third stage, only the reverse data transfer activity is active.

A set of software power meters is *concurrent* if all performance event sets required by the software power meters can be collected within the same application run during these stages. In practice, concurrency means that the performance events required by all software power meters can be collected together within the same application run, enabling the CPU computation and communication software power meters to operate simultaneously without requiring additional runs.

A set of software power meters is *orthogonal* if each software power meter reports negligible energy when its activity is absent and reports accurate energy when its activity is present, even if other independently powered activities execute in parallel.

For the running example above, during the CPU to GPU data transfer stage, the CPU computation power meter should report negligible energy, while the CPU to GPU data transfer power meter should report the data transfer energy. During the computation stage, the communication power meters should report negligible energy while the CPU computation power meter reports the CPU computation energy. During the GPU to CPU data transfer stage, the CPU computation power meter again reports negligible energy, while the GPU to CPU data transfer power meter reports the reverse data transfer energy. If these conditions hold, the sum of the reported energies provides an accurate decomposition of the program energy into independently powered computation and communication activities.

5.3 Methodology to Construct Concurrent and Orthogonal Software Power Meters

5.3.1 Overview

In this section, we outline our methodology for constructing concurrent and orthogonal software power meters designed for accurate runtime energy profiling of parallel hybrid programs on a heterogeneous server, which comprises one multicore CPU and two Nvidia GPUs.

We implement software power meters to measure the dynamic energy consumption of CPU computation and communication activities between CPU and GPU using linear energy predictive models. These models utilize Likwid performance events [62] as model variables. Likwid organizes performance events into groups that are customized for specific processor models. Detailed descriptions of the performance events and their groupings can be found in the appendix section C.1.

We observe that no CUPTI events or metrics capture data transfer activity on the GPU side. This finding highlights the passive role of GPUs in data transfer between CPUs and GPUs, indicating that the GPU's share of energy-consuming activities is insignificant compared to the CPU. Furthermore, our power meters for GPU computational activities rely on the NVML API for the on-chip sensors of the GPU. We have determined that these on-chip sensors provide accurate and reliable data for GPU generations starting from A40 GPU. Additionally, through comprehensive experiments on our Intel Icelake A40 server, we have confirmed that our two GPU software power meters for the two A40 GPUs are both concurrent and orthogonal.

We begin by discussing the challenging task of shortlisting Likwid performance events. Likwid is a well-known mainstream tool to obtain performance event counts of multicore CPU processors. It provides a certain number of core-local and socket-wide counters, denoted as c, and offers e performance events. The values of c and e vary between different processor architectures, with e typically being much larger than c. Moreover, both c and e tend to increase with each new processor generation. The counters are further divided into two groups: core-local and socket-wide, with each group capable of storing counts for its corresponding performance events.

However, the large number of available performance events makes it practically infeasible to collect all of them during runtime. Consequently, identifying a small subset of performance events that effectively captures all energy consumption activities during application execution is a significant challenge. While statistical methods, such as correlation and Principal Component Analysis (PCA), might seem like suitable solutions, relying solely on these techniques proves ineffective for generating accurate energy predictive models.

An effective technique for selecting performance events is the additivity test proposed by [46]. Additivity is based on the experimental observation that the total energy consumption when executing two applications in sequence is equal to the sum of the energy consumptions for each application run individually. Consequently, the additivity criterion follows a straightforward and intuitive rule: the performance event count for the serial execution of two applications should equal the sum of the counts observed during the individual executions of each application.

The authors [46] point out that the inaccuracy of energy models based on performance events arises because many commonly used core-local performance events are not additive on modern multicore processors. Therefore, the accuracy of performance monitoring counter (PMC)-based energy models can be enhanced by excluding non-additive performance events. Furthermore, a significant source of inaccuracy in these models is the violation of fundamental energy conservation laws.

This includes issues such as having a non-zero intercept in dynamic energy models, negative coefficients in additive terms, and the non-linearity present in some advanced models, including machine learning (ML) models [46], [47]. The theory of energy predictive models in computing [47] formalizes properties of energy predictive models based on performance event counts, which are derived from the fundamental law of energy conservation.

The practical implications of this theory for improving the prediction accuracy of linear energy predictive models are consolidated in a consistency test. This test includes criteria for selecting model variables, intercepts, and coefficients to develop accurate and reliable linear energy predictive models. Thus, to enhance the accuracy of dynamic energy models that utilize performance events, it is essential to eliminate non-additive performance events from the models and ensure adherence to basic energy conservation laws.

5.3.2 Fast Selection Procedure for Performance Event Sets

The methodology employs a fast selection procedure that integrates a natural grouping of performance events based on processor architecture, additivity, the theory of energy predictive models, and high positive correlation. This process is much quicker than an approach that evaluates all possible pair-wise correlations, which is impractical given the large number of performance events. This procedure aims to select concurrent and orthogonal sets of performance events.

The inputs to the procedure are sets of applications, the base additivity error, $\mathbb{A}, 0 \leq \mathbb{A} \leq 100$, the base positive correlation, $\rho, 0 \leq \rho \leq 100$, the statistical precision (standard deviation of the mean), $\epsilon, 0 \leq \epsilon \leq 100$, and the threshold, $\tau, 0 \leq \tau \leq 1$.

The application sets include:

- B_{C_1} comprising k base applications involving CPU computations.
- B_{H2D_1} and B_{H2D_2} comprising l_1 and l_2 base applications, respectively, involving data transfers of different sizes from the host CPU to GPU_1 and the host CPU to GPU_2, respectively.
- B_{D2H_1} and B_{D2H_2} comprising m_1 and m_2 base applications, respectively, executing data transfers from GPU_1 to CPU and from GPU_2 to CPU, respectively.
- n compound applications.
- h carefully designed synthetic parallel hybrid benchmarks comprising three CPU threads that lead the execution of CPU and GPU computations and are responsible for data transfers between the host CPU and GPUs.

A *compound application* is defined as the sequential execution of two base applications. If the base applications are denoted as A and B, we represent their compound application as AB. The total number of compound applications, n, is derived from a random selection of base application pairs. Compound applications encompass both CPU computations and data transfers. However, each individual base application consists of either a CPU computation or a data transfer activity, but not both.

The parallel hybrid benchmarks used in the procedure are representative of the class of parallel hybrid programs, where all activities within the program are independently powered. In each parallel hybrid benchmark, all simultaneously executed activities are always independently powered. Specifically, data transfers between the host CPU and GPUs occur simultaneously in one direction. However, if multiple data transfers use the same CPU-GPU pair, those transfers happen sequentially. For instance, a data transfer from the host CPU to a GPU is followed by a data transfer from that same GPU back to the host CPU.

The procedure does not utilize Pearson correlation coefficient to assess the correlation between two performance events or to evaluate the correlation of a performance event with dynamic energy consumption. Instead, it examines the functional relationship between a performance event and dynamic energy (or a pair of events) to determine if a strong positive correlation exists. A correlation is considered not significantly positive if there is notable non-linearity in the functional relationship. We use the same parameter ρ when checking the correlation between two performance events as well as between a performance event and dynamic energy consumption.

The procedure outputs concurrent and orthogonal performance event sets for CPU computations and data transfers. According to the definition of concurrency, a single application run is enough to gather the concurrent sets of performance events. However, if the sets of performance events are not concurrent, multiple application runs will be nec-

essary. As a result, the procedure provides a positive integer that indicates the number of application runs needed to collect all the performance event sets.

The procedure has four stages.

5.3.2.1 First Stage

In the first stage, it selects all highly additive (with an additivity error $\leq \mathbb{A}$) and highly positively correlated performance events with dynamic energy ($\geq \rho$). This stage relies solely on the input compound applications.

The main steps of this stage are as follows:

- Query the CPU processor clock to retrieve the start time. Start the energy meter on the CPU side using an energy measurement API's *start()* function [20].
- Execute a base or compound application from the input dataset.
- Stop the energy meter on the CPU side by using the energy measurement API's *stop()* function.
- Query the CPU processor clock again to obtain the end time, then calculate the difference between the end and start times to determine the elapsed execution time. The dynamic energy consumption of the application execution is obtained using the energy measurement API's *energy()* function.
- Gather event values during the application execution using the *Likwid-perfctr* tool.
- Verify that the execution time and dynamic energy consumption of a compound application equals the sum of the execution times and dynamic energy consumptions of its constituent base applications. If this verification fails, stop processing the current compound application and move on to the next one.
- For each application, create an event record that includes the dynamic energy and performance event values. Since performance event counts may vary widely across different counters (e.g., some in thousands and others in millions), the event records are then normalized using min-max scaling or normalization.

  $$x' = \frac{(x - \min(x))}{\max(x) - \min(x)}$$

  We find the global minimum and maximum event counts. Then for each event count, we subtract the minimum value and divide by the difference between the maximum and minimum value.

- Eliminate performance events that have insignificant counts (10 or fewer) or are not reliable. A performance event is considered deterministic and reproducible if it shows the same value (within a tolerance of 2.5%) across different executions of the same application, under the same runtime configuration and on the same platform.

- Calculate the additivity error for a performance event for a compound application using the following formula:

$$\text{Additivity error (\%)} = \left| \frac{(e_A + e_B) - e_{AB}}{(e_A + e_B + e_{AB})/2} \right| \times 100 \tag{5.1}$$

  Here, e_{AB}, e_A, e_B are the performance event counts for the compound (AB) and constituent base applications, A and B, respectively. Compute the sample average of this error from multiple experimental runs. The overall additivity error for the performance event is defined as the maximum error observed among all compound applications in the dataset.

- Select performance events with additivity error $\leq \mathbb{A}$.

- From the highly additive performance events that have been selected, shortlist those with a positive correlation ($\geq \rho$) for the next stage. Correlation measures the strength of the relationship between performance event counts and dynamic energy usage. A high positive correlation indicates that a performance event is a strong predictor of energy consumption and is therefore valuable for constructing accurate energy predictive models.

5.3.2.2 Second Stage

The second stage partitions the output from the first stage into five sets: $\{P_{C_1}, P_{H2D_1}, P_{H2D_2}, P_{D2H_1}, P_{D2H_2}\}$. The set P_{C_1} contains events attributed to CPU computation activity. The sets P_{H2D_1} and P_{H2D_2} include events specific to data transfer activities from the host CPU to GPU_1 and GPU_2, respectively. The sets P_{D2H_1} and P_{D2H_2} contain events related to data transfer from GPU_1 to the CPU and from GPU_2 to the CPU, respectively.

This stage uses input application sets that comprise the base applications: $\{B_{C_1}, B_{H2D_1}, B_{H2D_2}, B_{D2H_1}, B_{D2H_2}\}$.

The main steps of this stage are outlined below:

- The procedure begins by categorizing the shortlisted performance events from stage one into performance monitoring counter groups provided by Likwid. These groups are labeled as $g_1, g_2, \ldots, g_G$. We will describe the specific Likwid groups

for the hybrid servers used in this study in the next section. Let group $g_i, i \in \{1, \cdots, G\}$ contain c_i performance events, $\{e_{i1}, e_{i2} \cdots, e_{ic_i}\}$. The analysis for each group $g_i, i \in \{1, \cdots, G\}$ proceeds as follows:

- For each event $e_{ij}, i \in \{1, \cdots, G\}, j \in \{1, \cdots, c_i\}$ and each base application b from the sets, $\{B_{C_1}, B_{H2D_1}, B_{H2D_2}, B_{D2H_1}, B_{D2H_2}\}$, the rate of the event count, $r(e_{ij}, b)$, is calculated as follows: $r(e_{ij}, b) = \frac{count(e_{ij},b)}{t(b)}$. In this formula, *count*(x,y) returns the normalized count of event x during the execution of the application y, and $t(y)$ is the execution time of application y.
- Let τ be the specified threshold. Then for each event $e_{ij}, i \in \{1, \cdots, G\}, j \in \{1, \cdots, c_i\}$ and for each activity $p \in \{C_1, H2D_1, H2D_2, D2H_1, D2H_2\}$, we define the following criteria for inclusion of e_{ij} in P_p:
  1. The minimum rate of the event count of e_{ij} across all base applications involving activity p must be greater than τ:

$$\min_{b \in B_p} r(e_{ij}, b) > \tau$$

  2. The highest rate (the maximum of the maximum rates) of the event count of e_{ij} across all base applications involving activities other than p must be less than or equal to τ:

$$\max_{q \neq p} \{\max_{b \in B_q} r(e_{ij}, b)\} \leq \tau$$

  If both conditions (1) and (2) are satisfied, then e_{ij} is included in P_p.
- If e_{ij} is included in P_p or if all activities have been considered, we proceed to the next event. If not, we move on to the next activity.

5.3.2.3 Third Stage

The inputs to the third stage consist of h synthetic parallel hybrid benchmarks and the five performance event sets, $\{P_{C_1}, P_{H2D_1}, P_{H2D_2}, P_{D2H_1}, P_{D2H_2}\}$, obtained from the second stage. The objective of this stage is to analyze the impact of interference caused by independently powered activities running in parallel on these performance events while also eliminating events that exhibit cross-contamination.

The stage follows these main steps:

- For each hybrid program $h_k, k \in \{1, \cdots, h\}$ and for each event set, $P_p, p \in \{C_1, H2D_1, H2D_2, D2H_1, D2H_2\}$:
  - The procedure analyzes the counts of all events in the event set, P_p. For each performance event, it calculates the relative percentage difference between

the normalized event counts obtained for the hybrid program execution and those from the stand-alone execution of the base program used in the hybrid to implement the p-th activity. If this relative percentage difference exceeds the specified tolerance level, ϵ, the performance event is eliminated from consideration. This test removes from P_p all performance events, whose counts are affected by parallel execution of other activities. If such events were to be used as predictors in linear dynamic energy models employed by software power meters, they would violate the orthogonality property (e) formulated in Section 5.2.

The five performance event sets produced in the third stage, $\{P_{C_1}, P_{H2D_1}, P_{H2D_2}, P_{D2H_1}, P_{D2H_2}\}$, are orthogonal for the h synthetic parallel hybrid benchmarks utilized in this stage. We consider these benchmarks to be representative of the class of parallel hybrid programs where all simultaneously executed activities are independently powered.

5.3.2.4 Fourth Stage

In the final stage, these five orthogonal performance event sets are used as input. The objective of this stage is to make the sets concurrent by eliminating redundant events in each set that duplicate the contributions of other events in the linear dynamic energy model employing the set.

The process consists of the following steps:

- The procedure first checks if there are enough performance monitoring counters to store the shortlisted performance events from the five sets. If there are sufficient counters, the procedure terminates and returns the performance event sets, along with the indication that one application run is sufficient to obtain all the performance events.
- For each performance event set, $P_p, p \in \{C_1, H2D_1, H2D_2, D2H_1, D2H_2\}$, the procedure examines the Likwid performance monitoring counter groups associated with the performance events in the set. It then performs intra-group and inter-group sub-steps, which are detailed below. Specifically, these two sub-steps involve analysis of the Likwid performance monitoring counter groups within a performance event set. The analysis utilizes all event records, including the performance event counts and the dynamic energy consumption data generated in the previous stages.
  - In the intra-group step, the procedure prunes the performance events within each group based on their correlation with one another and their dynamic

energy consumption.

  - In the inter-group step, the focus shifts to pruning performance events across different groups by examining pairwise correlations. This step aims to identify a minimal set of disjoint performance events that are highly positively correlated with dynamic energy, while not being highly correlated with each other. This approach captures separate independent contributions to dynamic energy that can be obtained from a single application run.

- If the performance event sets are not concurrent, the procedure increases the correlation (ρ) in increments of 1% (up to the maximum allowed by the user-specified statistical precision, ϵ). For each increment, it removes all highly additive performance events (with an additivity error $\leq \mathbb{A}$) that have a positive correlation (at least $\geq \rho$). If the resulting disjoint performance event sets are concurrent, the procedure exits.

- If the performance event sets remain non-concurrent, the procedure reduces the additivity error ($\mathbb{A}$) in increments of 1% (again, up to the maximum allowed by the user-specified statistical precision, ϵ). For each decrement, it removes all performance events with an additivity error greater than $\mathbb{A}$.

If the procedure fails to identify the desired disjoint performance event sets after these steps, it will determine the number of application runs needed to obtain all the shortlisted performance events. It will then return this number along with the performance event sets.

5.3.3 Concrete Examples of Event Sets that Satisfy or Violate the Properties

Tables 5.1, 5.2, and 5.3 provide concrete examples of performance event sets that satisfy both concurrency and orthogonality on three heterogeneous hybrid servers. For example, in Table 5.3, the CPU computation power meter uses the event set listed under CPU Computation, while each data transfer direction uses its own separate event set. These sets are disjoint, which supports orthogonality, and they are simultaneously measurable on the target server, which supports concurrency.

To further clarify the distinction, we relate this discussion to the performance event sets developed in Chapter 4. In Chapter 4, the event selection procedure identifies a small subset of performance events for modeling the energy of a single activity in isolation. For example, the CPU computation software power meter in Chapter 4 uses events such as `CBOX_CLOCKTICKS` (group CBOX) and `TXC_AD_CREDIT_OCCUPANCY`

(group M2M). These events were selected because they correlate well with CPU computation energy when computation is executed alone.

However, when computation and communication activities execute in parallel, these events may not uniquely represent CPU computation. If `CBOX_CLOCKTICKS` or `TXC_AD_CREDIT_OCCUPANCY` produce non negligible counts during data transfer phases, the CPU computation software power meter could report non negligible energy even when no CPU computation is taking place. In such a case, the event set violates the orthogonality property because it does not clearly distinguish between computation and communication activities.

In contrast, the concurrent and orthogonal software power meters constructed in this Chapter 5 use disjoint event sets that are validated under parallel execution conditions. For example, the CPU computation power meter uses events such as `READ_NO_CREDITS_ANY` (group CBOX) and `DISTRESS_ASSERTED_VERT` (group M2M), while the CPU to GPU data transfer power meter uses a separate set including `DIRECT_GO_OPC_EXTCMP` (group CBOX) and `TXR_HORZ_ADS_USED_AD_UNCRD` (group M2M). These event sets are selected so that each set primarily reflects its corresponding activity and produces negligible counts when that activity is absent. This ensures orthogonality.

A violation of concurrency would occur if the total number of required events within a single performance monitoring counter group exceeded the available registers for that group. In this case, the required events cannot be collected together within a single application run, and multiple runs would be needed to obtain all event counts. This prevents simultaneous runtime energy profiling of computation and communication activities. The event selection and pruning stages of the proposed methodology ensure that the final event sets satisfy the register constraints and can be collected within a single run, thereby satisfying the concurrency property.

5.3.4 Scalability of the Methodology

The methodology's scalability pertains to its capacity to identify concurrent and orthogonal performance event sets as the number of physical CPU sockets and GPUs increases. Three factors influence scalability: the number of registers available for core-local and socket-wide performance monitoring counters, as well as the physical topology of the server, which is determined by two key components: the number of physical CPU sockets and GPUs.

We will first examine how the three factors influence scalability by using our Intel Icelake A40 server as an example. This server features a single-socket multicore CPU and two A40 GPUs. Likwid categorizes performance events on this server into eleven performance monitoring counter groups, specifically {CBOX, ..., WBOX}. There are four

core-local and seven socket-wide performance monitoring counter groups. Each group is assigned a specific number of registers that allow for the simultaneous collection of performance event counts coresponding to that group during a single application run.

For this server, Likwid provides four registers to store performance event counts related to the PCIe socket-wide counter group, known as PBOX. In contrast, it offers 120 registers for the Last Level Cache (LLC) socket-wide counter group, referred to as CBOX. Additionally, in the core-local performance monitoring counter groups, performance event counts are recorded for each core, whereas in the socket-wide performance monitoring counter groups, the counts are aggregated for the entire socket.

Based on our methodology applied to three hybrid servers, we find that a CPU computation software power meter uses performance events sourced from core-local Performance Monitoring Counter (PMC) groups. In contrast, a data transfer software power meter relies on performance events from each socket-wide performance monitoring counter group. This difference is due to the fact that data transfer power meters incorporate performance events from socket-wide performance monitoring counter groups, which capture off-core chip traffic.

So, the four data transfer software power meters (which monitor communication between the host CPU and the two A40 GPUs in both directions) designed for the Intel Icelake A40 server will exhaust all the registers allocated to the PBOX performance monitoring counter group because this server is a single-socket system. This issue means that our methodology does not scale efficiently if a single-socket CPU server has more than two GPUs.

Specifically, if a single-socket host CPU connects to an additional GPU, it will require two more events (one for each data transfer direction) from the PBOX group. In total, this would necessitate six counters from the PBOX group; however, this server only has four available. As a result, the power meters will be unable to maintain the desired concurrency property.

Consider the scenario where a server has a ratio of the number of physical CPU sockets to the number of GPUs that is greater than or equal to 0.5. In this case, our methodology will scale effectively because the number of registers available for a socket-wide Performance Monitoring Counter (PMC) group will be the product of the number of sockets and the number of registers in that group. Each register provides a separate socket-wide count.

For example, take a dual-socket multicore CPU server that hosts four GPUs. In this setup, the PBOX performance monitoring counter group will provide eight socket-wide counts (four registers, with each register offering two socket-wide counts). These eight counts can be utilized by the eight power meters dedicated to the data transfers between the CPU and GPUs, where there are two data transfers for each CPU-GPU pair. Therefore, the scalability of our methodology is not a concern for hybrid servers

with a favorable ratio of CPU sockets to GPUs.

5.3.5 Applying the Methodology to Select Performance Event Sets for Three Heterogeneous Hybrid Servers

Tables 5.1, 5.2, and 5.3 shows the concurrent and orthogonal performance event sets for the three heterogeneous hybrid servers, Haswell k40c GPU server, Skylake P100 GPU server, and Icelake A40 GPU servers as shown in Tables 3.1, 3.2, and 3.3. The parameters input to the methodology are: $\mathbb{A}$ = 5%, ρ = 95%, ϵ = 2.5%, and τ = 0.025.

Table 5.1: Concurrent and orthogonal performance event sets for CPU computations and data transfer in both directions for the heterogeneous server in Table 3.1 containing an Intel Haswell multicore CPU and Nvidia K40c GPU.

CPU Computation	
Groups	**Performance Events**
PMC	L2_RQSTS_ALL_DEMAND_DATA_RD_MISS
MBOX	RD_CAS_RANK0_ALLBANKS
QBOX	DIRECT2CORE_FAILURE_RBT_MISS
CBOX	RXR_ISMQ_RETRY_QPI_CREDITS
BBOX	RING_BL_USED_CCW
PBOX	RXR_CYCLES_NE_NCB
Data Transfer CPU $\Longrightarrow$ K40c GPU	
Groups	**Performance Events**
PMC	LOAD_HIT_PRE_SW_PF
MBOX	MAJOR_MODES_ISOCH
QBOX	TXR_BL_NCS_CREDIT_ACQUIRED_VN0
CBOX	RXR_INSERTS_PRQ_REJ
BBOX	ADDR_OPC_MATCH_OPC
PBOX	TXR_NACK_CW_DN_AK
Data Transfer K40c GPU $\Longrightarrow$ CPU	
Groups	**Performance Events**
PMC	MISALIGN_MEM_REF_LOADS
MBOX	POWER_SELF_REFRESH
QBOX	RXL_FLITS_G2_NDR_AK
CBOX	LLC_LOOKUP_REMOTE_SNOOP
BBOX	IMC_WRITES_PARTIAL
PBOX	TXR_NACK_CW_UP_AD

Table 5.2: Concurrent and orthogonal performance event sets for CPU computations and data transfer in both directions for the heterogeneous server in Table 3.2 containing an Intel Skylake multicore CPU and Nvidia P100 PCIe GPU.

CPU Computation	
Groups	**Performance Events**
PMC	FP_ARITH_INST_RETIRED_DOUBLE
CBOX	RXC_IRQ1_REJECT_LLC_VICTIM
MBOX	CAS_COUNT_WR_WMM
IRP	CACHE_TOTAL_OCCUPANCY_ANY
M2M	RPQ_CYCLES_REG_NO_CREDITS_CHN2
Data Transfer CPU $\Longrightarrow$ GPU	
Groups	**Performance Events**
PMC	FP_ARITH_INST_RETIRED_SCALAR_SINGLE
CBOX	TOR_OCCUPANCY_IO_MISS
MBOX	ACT_COUNT_RD
IRP	IRP_ALL_OUTBOUND_INSERTS
M2M	BYPASS_M2M_INGRESS_TAKEN
Data Transfer GPU $\Longrightarrow$ CPU	
Groups	**Performance Events**
PMC	ILD_STALL_LCP
CBOX	HORZ_RING_IV_IN_USE_RIGHT
MBOX	CAS_COUNT_RD_UNDERFILL
IRP	IRP_ALL_INBOUND_INSERTS
M2M	DDRT_RPQ_CYCLES_REG_CREDITS_CHN1

Table 5.3: Concurrent and orthogonal performance event sets for CPU computations and data transfer directions in the heterogeneous server in Table 3.3 containing a single-socket Icelake multicore CPU and two Nvidia A40 GPUs.

CPU Computation	
Groups	**Performance Events**
CBOX	READ_NO_CREDITS_ANY
PMC	FP_ARITH_INST_RETIRED_SCALAR_DOUBLE
M2M	DISTRESS_ASSERTED_VERT
TCBOX	REQ_FROM_PCIE_CMPL_REQ_OWN
UBOX	M2U_MISC1_RXC_CYCLES_NE_CBO_NCB
WBOX	FREQ_TRANS_CYCLES
Data Transfer CPU $\Longrightarrow$ A40 GPU1	
Groups	**Performance Events**
CBOX	DIRECT_GO_OPC_EXTCMP
PMC	FP_ARITH_INST_RETIRED_SCALAR_SINGLE
M2M	TXR_HORZ_ADS_USED_AD_UNCRD
PBOX	IIO_CREDITS_ACQUIRED_DRS_ANY
TCBOX	INBOUND_ARB_REQ_WR
Data Transfer A40 GPU1 $\Longrightarrow$ CPU	
Groups	**Performance Events**
CBOX	RXC_ISMQ0_REJECT_BL_RSP_VN0
PMC	MEM_LOAD_RETIRED_L3_HIT
M2M	VERT_RING_AK_IN_USE_DN_EVEN
PBOX	TXC_INSERTS_AD_ANY
TCBOX	NUM_REQ_OF_CPU_BY_TGT_MEM
Data Transfer CPU $\Longrightarrow$ A40 GPU2	
Groups	**Performance Events**
CBOX	REQUESTS_INVITOE
PMC	MEM_LOAD_L3_HIT_RETIRED_XSNP_NONE
M2M	PREFCAM_DROP_REASONS_CH0_STOP_B2B
PBOX	RXC_INSERTS_ALL
TCBOX	INBOUND_ARB_REQ_FINAL_RD_WR
Data Transfer A40 GPU2 $\Longrightarrow$ CPU	
Groups	**Performance Events**
CBOX	RXC_ISMQ1_REJECT_ANY0
PMC	OFFCORE_REQUESTS_DEMAND_DATA_RD
M2M	RXR_OCCUPANCY_BL_UNCRD
PBOX	IIO_CREDITS_USED_DRS_1
TCBOX	INBOUND_ARB_REQ_REQ_OWN

5.4 Software Power Meters for Computation and Communication

We design and develop system-level software power meters based on linear energy predictive models that leverage concurrent and orthogonal sets of performance events as predictor variables. Based on the practical applications of the theory of energy predictive models in computing [47], our models maintain a zero intercept and non-negative regression coefficients. We utilize the Huber Regressor from the Python's Scikit-learn package to construct these models. The Huber Regressor is a robust regression algorithm that is less sensitive to outliers compared to ordinary least squares (OLS) linear regression.

To train and test these models, we split the datasets into 70% for training and 30% for testing. These records are obtained by executing the applications that are used to select the performance event sets. Section C.2 in the appendix shows the set of applications used for training and testing the energy predictive models.

The Tables in appendix C.4 and C.5 shows the list of applications employed for training and testing the energy predictive models for computations on the multicore CPU, and the energy predictive models for data transfers between CPU and GPU respectively. Table C.6 shows the list of parallel hybrid applications employed for selection of performance event sets.

We hypothesize that no performance events would be shared in the selected event sets employed in the computation and data transfer energy predictive models. We base our hypothesis on the rationale that the performance events that capture the energy activity of computations would primarily be core-level performance monitoring counter groups. In contrast, a data transfer software power meter relies on performance events from each socket-wide performance monitoring counter group.

This difference is due to the fact that data transfer software power meters incorporate performance events from socketwide performance monitoring counter groups, which capture off-core chip traffic, data traffic between the main memory and the caches, QPI traffic, and Power Control Unit (PCU) measurements. The event sets have no shared performance events, thereby validating our hypothesis.

In addition, the shortlisted performance events are all CPU performance events, and there are no events capturing data transfer activity on the GPU side. This finding reflects the passive role of GPUs in data transfer between CPUs and GPUs, making the GPU's share of energy-consuming activities insignificant compared to the CPU. Furthermore, the shortlisted performance events belong to different performance monitoring counter groups, and each group has enough dedicated performance monitoring counters to store the individual performance event counts. Therefore, the event counts can be obtained in one application run. This feature allows our software power meters employing these selected disjoint event sets to be efficiently deployed at runtime.

For the hybrid parallel applications executed on our servers, the performance event sets of the energy predictive software power meters for both data transfer and computation are concurrent, orthogonal. There is no overlap in the performance events in the data transfer and computation energy predictive software power meters, confirming our hypothesis.

This significant finding means one can develop separate software power meters for computation and separate software power meters for data transfer activities, thereby providing a more fine-grained decomposition of the energy consumed by software components in a heterogeneous hybrid application.

The system-level software power meters (linear energy predictive models) for the Haswell K40c GPU server are outlined in Table 5.4. The average prediction errors of these software power meters are as follows: 2% for CPU computations and 3% for the two data transfer software power meters between the host CPU and the K40c GPU server. The software power meters for GPU computations utilize NVML and their average prediction error is 5%.

The system-level software power meters (linear energy predictive models) for the Skylake P100 GPU server are outlined in Table 5.5. The average prediction errors of these software power meters are as follows: 1% for CPU computations and 2% for the two data transfer software power meters between the host CPU and the P100 GPU server. The software power meters for GPU computations utilize NVML and their average prediction error is 5%.

The system-level software power meters (linear energy predictive models) for the Icelake A40 GPU server are outlined in Table 5.6. The average prediction errors of these software power meters are as follows: 1% for CPU computations and 2% for the four data transfer software power meters between the host CPU and the two A40 GPU servers. The software power meters for GPU computations utilize NVML and their average prediction error is 5%.

Table 5.4: Software power meters implementing the linear dynamic energy predictive models for Haswell K40c GPU server as in Table 3.1.

Model Description	Linear Regression Model
CPU computations	7.45E-08 × L2_RQSTS_ALL_DEMAND_DATA_RD_MISS + 3.53E-05 × RD_CAS_RANK0_ALLBANKS + 00E0 × DIRECT2CORE_FAILURE_RBT_MISS + 9.63E-06 × RXR_ISMQ_RETRY_QPI_CREDITS + 1.29E-07 × RING_BL_USED_CCW + 1.78E-07 × RXR_CYCLES_NE_NCB
CPU → K40c GPU	4.69E-09 × LOAD_HIT_PRE_SW_PF + 2.12E-08 × MAJOR_MODES_ISOCH + 00E0 × TXR_BL_NCS_CREDIT_ACQUIRED_VN0 + 5.06E-06 × RXR_INSERTS_PRQ_REJ + 1.07E-06 × ADDR_OPC_MATCH_OPC + 7.45E-08 × TXR_NACK_CW_DN_AK
K40c GPU → CPU	6.11E-09 × MISALIGN_MEM_REF_LOADS + 7.17E-07 × POWER_SELF_REFRESH + 00E0 × RXL_FLITS_G2_NDR_AK + 8.92E-06 × LLC_LOOKUP_REMOTE_SNOOP + 3.93E-07 × IMC_WRITES_PARTIAL + 7.34E-08 × TXR_NACK_CW_UP_AD

Table 5.5: Software power meters implementing the linear dynamic energy predictive models for Skylake P100 GPU server as in Table 3.2.

Model Description	Linear Regression Model
CPU computations	1.29E-09 × FP_ARITH_INST_RETIRED_DOUBLE + 9.63E-07 × RXC_IRQ1_REJECT_LLC_VICTIM + 3.38E-08 × CAS_COUNT_WR_WMM + 1.11E-06 × CACHE_TOTAL_OCCUPANCY_ANY + 6.64E-07 × RPQ_CYCLES_REG_NO_CREDITS_CHN2
CPU → P100 GPU	5.81E-08 × FP_ARITH_INST_RETIRED_SCALAR_SINGLE + 7.15E-07 × TOR_OCCUPANCY_IO_MISS + 2.88E-07 × ACT_COUNT_RD + 7.35E-06 × IRP_ALL_OUTBOUND_INSERTS + 1.57E-06 × BY-PASS_M2M_INGRESS_TAKEN
P100 GPU → CPU	2.08E-08 × ILD_STALL_LCP + 8.68E-07 × HORZ_RING_IV_IN_USE_RIGHT + 2.81E-07 × CAS_COUNT_RD_UNDERFILL + 7.33E-06 × IRP_ALL_INBOUND_INSERTS + 1.68E-06 × DDRT_RPQ_CYCLES_REG_CREDITS_CHN1

Table 5.6: Software power meters implementing the linear dynamic energy predictive models for Icelake A40 GPU server as in Table 3.3.

Model Description	Linear Regression Model
CPU computations	6.73E-05 × FP_ARITH_INST_RETIRED_SCALAR_DOUBLE + 6.32E-07 × READ_NO_CREDITS_ANY + 2.23E-07 × DISTRESS_ASSERTED_VERT + 4.863E-05 × REQ_FROM_PCIE_CMPL_REQ_OWN + 0.00E00 × M2U_MISC1_RXC_CYCLES_NE_CBO_NCB + 3.24E-07 × FREQ_TRANS_CYCLES
CPU → A40 GPU1	4.60E-07 × DIRECT_GO_OPC_EXTCMP + 8.14E-08 × FP_ARITH_INST_RETIRED_SCALAR_SINGLE + 1.28E-08 × TXR_HORZ_ADS_USED_AD_UNCRD + 6.40E-11 × IIO_CREDITS_ACQUIRED_DRS_ANY + 7.13E-07 × INBOUND_ARB_REQ_WR
A40 GPU1 → CPU	2.24E-08 × RXC_ISMQ0_REJECT_BL_RSP_VN0 + 3.93E-07 × MEM_LOAD_RETIRED_L3_HIT + 8.16E-08 × VERT_RING_AK_IN_USE_DN_EVEN + 3.79E-12 × TXC_INSERTS_AD_ANY + 4.66E-07 × NUM_REQ_OF_CPU_BY_TGT_MEM
CPU → A40 GPU2	3.10E-07 × REQUESTS_INVITOE + 1.69E-08 × MEM_LOAD_L3_HIT_RETIRED_XSNP_NONE + 4.76E-08 × PREF-CAM_DROP_REASONS_CH0_STOP_B2B + 1.69E-11 × RXC_INSERTS_ALL + 3.93E-07 × INBOUND_ARB_REQ_FINAL_RD_WR
A40 GPU2 → CPU	3.24E-08 × RXC_ISMQ1_REJECT_ANY0 + 1.11E-08 × OF-FCORE_REQUESTS_DEMAND_DATA_RD + 4.66E-08 × RXR_OCCUPANCY_BL_UNCRD + 1.66E-11 × IIO_CREDITS_USED_DRS_1 + 5.41E-07 × INBOUND_ARB_REQ_REQ_OWN

5.5 Experimental Validation of concurrent and orthogonal Software Power Meters

We demonstrate the accuracy of our proposed concurrent and orthogonal software power meters based on linear energy predictive models that employ disjoint sets of performance events as predictor variable on three heterogeneous hybrid servers using three highly optimized parallel hybrid scientific applications which are matrix addition (HDGEADD), matrix multiplication (HDGEMM), and 2D fast Fourier transform (HFFT2D).

An activity in a software component pertains to either computations within the component or the data transfer between a pair of communicating computing devices. Therefore, at any given moment during its execution, the parallel hybrid program can engage in a single activity, such as computation on the CPU, computation on one of the accelerators, or data transfer between a pair of computing devices, or it may perform multiple such activities in parallel.

In each hybrid application, the data transfers between host CPU cores and GPUs in the same direction occur simultaneously. These activities are independently powered. However, data transfers that occur between the same pair of a host CPU core and GPU in opposite directions are not independently powered and do not run in parallel. Specifically, a data transfer from the GPU to the CPU always logically follows a data transfer from the CPU to the GPU. Therefore, in all of our applications, whenever activities occur in parallel, they are all independently powered.

Additionally, every hybrid application is equipped with our proposed software power meters to monitor computations and data transfers using the software power meter API, which is explained in the appendix section C.3. For servers in Tables 3.1 and 3.2, four energy software power meters are integrated into each application: two software power meters monitor computations in the CPU, and GPU, while two software power meters track data transfers between the host CPU and the GPU components.

For server in Table 3.3, seven energy software power meters are integrated into each application: three software power meters monitor computations in the CPU, A40 GPU_1, and A40 GPU_2 components, while four software power meters track data transfers between the host CPU and the A40 GPU components.

The experiments are repeated for a data point obtained using the ground-truth method until the sample mean falls within the 95% confidence interval, achieving a precision (or standard deviation of the mean) of 2.5%. To accomplish this, we use Student's t-test, assuming that the individual observations are independent and that the population follows a normal distribution. We validate these assumptions using Shapiro-Wilk Test.

For data points obtained from software power meters, we perform three repetitions and calculate the average prediction error, which is the average of the prediction errors from the three runs. Finally, we report the overall average prediction error of a software

power meter, calculated by averaging the average prediction errors obtained across all data points.

5.5.1 Profiling Overhead and Accuracy Trade-Off

An important requirement of runtime energy profiling methods is that they should impose negligible overhead on application execution while maintaining high accuracy. The concurrent and orthogonal software power meters proposed in this work are designed to operate at runtime using hardware performance counters and on-chip sensor readings (NVML), which are accessed programmatically during application execution.

It is important to distinguish between the construction phase and the deployment phase of the software power meters. The process of constructing concurrent and orthogonal software power meters involves offline experimental analysis, performance event selection, and model validation. This phase requires careful experimentation and may incur computational cost. However, this cost is incurred only once during model development.

At runtime, the deployed software power meters do not introduce additional execution overhead. The shortlisted performance events are collected in a single application run using hardware-supported performance monitoring counters. Since these counters are implemented in hardware and are designed for performance analysis, collecting their values does not require repeated executions or statistical averaging and does not interfere with application execution. Similarly, GPU computation energy is obtained through the NVML interface, which provides access to on-chip sensor readings without interrupting the running program.

In contrast, the ground-truth method based on external power meters requires repeated application executions to obtain statistically reliable energy measurements. This repeated process significantly increases profiling time and makes the method unsuitable for runtime deployment.

The experimental results presented in this chapter demonstrate that the proposed concurrent and orthogonal software power meters achieve high accuracy, with an average prediction error of 2.5% across multiple applications, while introducing no observable runtime overhead.

Therefore, the proposed methodology achieves an effective trade-off between accuracy and overhead. The construction of the software power meters requires offline effort, but once deployed, they provide accurate runtime energy estimation without adding execution delay or noticeable contention.

5.5.2 Parallel Hybrid Applications: Description

5.5.2.1 Matrix Addition (HDGEADD)

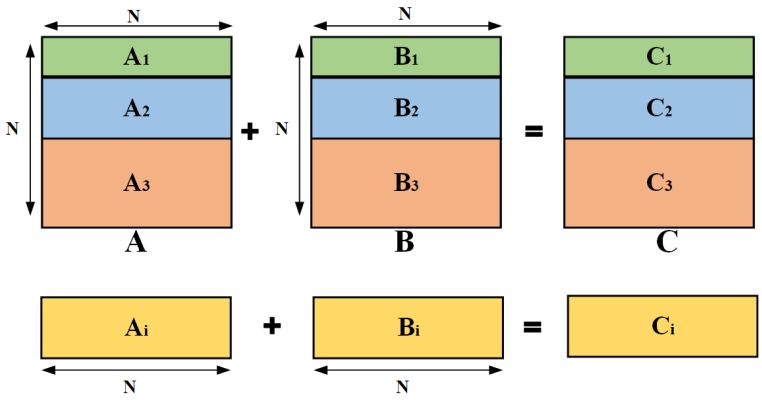


Figure 5.1: Parallel matrix addition application (HDGEADD) computing the matrix addition (C += $A+B$) of two dense square matrices A and B of size $N \times N$. It is executed on the Icelake A40 GPU server. The application comprises 3 software components executed in parallel (one CPU component, one GPU_1 component, and one GPU_2 component). Each software component i computes the matrix addition, C_i += $A_i + B_i$.

The parallel hybrid matrix addition application, referred to as HDGEADD, computes the matrix addition (C += $A + B$) for two dense square matrices, A and B, both of size $N \times N$. Figure 5.1 illustrates the application. The matrices A, B and C are horizontally partitioned so that each software component is assigned several contiguous rows of A, B and C provided as input parameters to the application. Each software component *i* computes the matrix addition, C_i += $A_i + B_i$.

The application comprises three main stages:

1. *Data Transfer Stage:* In the first stage, the matrices A_2, B_2, and C_2 are transferred from the host CPU to A40 GPU_1, while A_3, B_3, and C_3 are transferred to A40 GPU_2.

2. *Computation Stage:* The second stage involves local computations in the software components. For the component that utilizes the multicore CPU, the computations are carried out using the Intel Math Kernel Library (MKL) routine (DGEADD). In the components that use the A40 GPU, the computations are performed using the CUBLAS library routine.

3. *Result Transfer Stage:* The third stage includes transferring the result matrices C_2 and C_3 from A40 GPU_2 and A40 GPU_1 back to the host CPU.

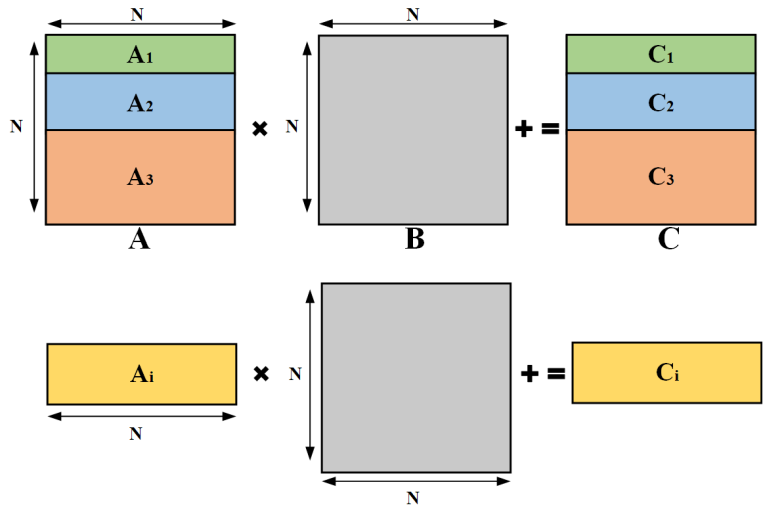


Figure 5.2: Matrix partitioning between the software components in parallel hybrid matrix multiplication application (HDGEMM) computing the matrix product ($C+ = A \times B$) of two dense square matrices A and B of size $N \times N$. The matrix B is shared by all the components. It is executed on the Icelake A40 GPU server. The application comprises 3 software components executed in parallel (one CPU component, one GPU_1 component, and one GPU_2 component). Each software component i computes the matrix product, C_i += $A_i \times B$.

5.5.2.2 Matrix Multiplication (HDGEMM)

The parallel hybrid matrix multiplication application (HDGEMM) computes the matrix product (C += $A \times B$) for two dense square matrices A and B, each of size $N \times N$. Figure 5.2 illustrates the application. The application consists of three software components: the CPU, A40 GPU_1, and A40 GPU_2. All components share the matrix B. The matrices A and C are horizontally partitioned, with each software component assigned several contiguous rows of A and C, as specified by input parameters to the application. The CPU is responsible for N_1 rows of A and C. The A40 GPU_1 component is assigned N_2 rows of A and C, while the A40 GPU_2 component takes on the remaining rows, specifically $N - N_1 - N_2$ of A and C. Each software component i (where $i \in \{1, 2, 3\}$) computes the matrix product C_i += $A_i \times B$.

The application consists of three main stages:

1. *Data Transfer Stage:* The first stage involves transferring data: A_2, B, and C_2 from the host CPU to A40 GPU_1, and A_3, B, and C_3 from the host CPU to A40 GPU_2.

2. *Computation Stage:* The second stage is dedicated to performing local computations within the software components. The computations in the CPU software component are performed using the Intel MKL DGEMM library routine, while the computations in the software components that involve the A40 GPUs are performed using the CUBLAS DGEMM library routine.

3. *Result Transfer Stage:* The third stage entails transferring the resulting matrices,

C_2 and C_3, from A40 GPU_2 and A40 GPU_1 back to the host CPU.

5.5.2.3 2D Fast Fourier Transform (HFFT)

The parallel hybrid 2D fast Fourier Transform application (HFFT2D) computes the 2D-FFT of a signal matrix S of size $N \times N$. Figure 5.3 illustrates the application. The application comprises three software components executed in parallel (one CPU component, one GPU_1 component, and one GPU_2 component). The matrix S is partitioned so that each component receives several contiguous rows of S provided as input.

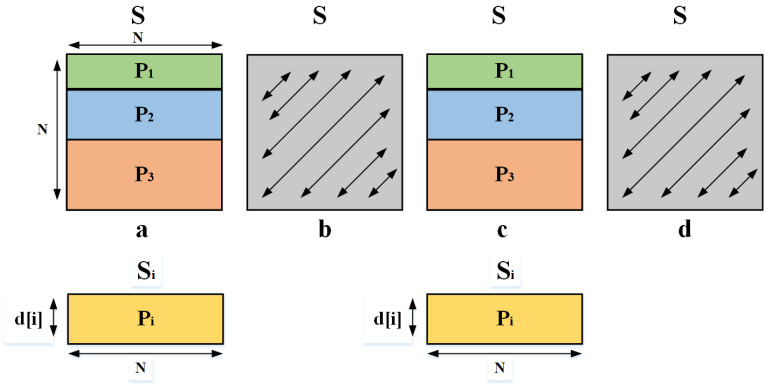


Figure 5.3: Parallel hybrid 2D fast Fourier Transform application (HFFT2D) computing the 2D-FFT of a signal matrix S of size $N \times N$. It is executed on the Icelake A40 GPU server. The application comprises three software components executed in parallel (one CPU component, one GPU_1 component, and one GPU_2 component). The matrix S is partitioned such that each software component is assigned several contiguous rows of S provided as a parameter to the application. The 2D FFT accomplished in four steps: (1) Computing 1D FFTs on N rows of the signal matrix N. (2) Transpose of the signal matrix. Steps (3) and (4) repeat steps (1) and (2), respectively.

The 2D FFT is accomplished in four steps: (1) Computing 1D FFTs on *N* rows of the signal matrix N; (2) Transposing the signal matrix; (3) Repeating step (1); and (4) Repeating step (2).

The application consists of eight stages. Data Transfer Stage: The first stage involves transfering data P_2 from the host CPU to A40 GPU_1 and P_3 from the host CPU to A40 GPU_2. Computation Stage: The second stage involves local computations in the software components. The computations in the software component involving the multicore CPU are performed using the Intel MKL FFT library routine. The computations in the software components involving the A40 GPUs are performed using the CUFFT library routine.

Result Transfer Stage: The third stage entails transfering the result matrices P_2 and P_3 from A40 GPU_2 and A40 GPU_1 to the host CPU. The fourth stage involves the transposing the matrix S on the CPU. These four stages are repeated. It is important to note that we do not consider the energy consumption of the matrix transposition on the

CPU, as it is negligible.

5.5.3 Experimental Results and Discussion

We present experimental results showing energy predictions made by our concurrent and orthogonal software power meters for the three hybrid servers: the Haswell K40c GPU server (refer to Table 3.1), the Skylake P100 GPU server (refer to Table 3.2), and Icelake A40 GPU server (refer to Table 3.3) for three parallel applications.

5.5.3.1 CPU Computation

Figures 5.4 and 5.5 illustrate the energy predictions by the software power meter for CPU computations on the heterogeneous hybrid servers described in Tables 3.1 and 3.2, across three different applications. The x-axis represents the matrix size N ranging from 10240 × 10240 to 20480 x 20480, while the y-axis indicates the dynamic energy consumption in joules.

Figure 5.6 illustrates the energy predictions by the software power meter for CPU computations on the heterogeneous hybrid server described in Table 3.3, across three different applications. The x-axis represents the matrix size N, ranging from 25600 × 25600 to 35840 × 35840, while the y-axis indicates the dynamic energy consumption in joules.

Each plot has two curves: the blue curve shows the actual dynamic energy measured using the ground-truth data from external power meters, the orange curve displays the predicted energy consumption by the CPU computation software power meter. The average prediction error is 2.5%.

Additionally, the CPU computation software power meter reports negligible energy consumption during phases when the CPU computation activities are absent.

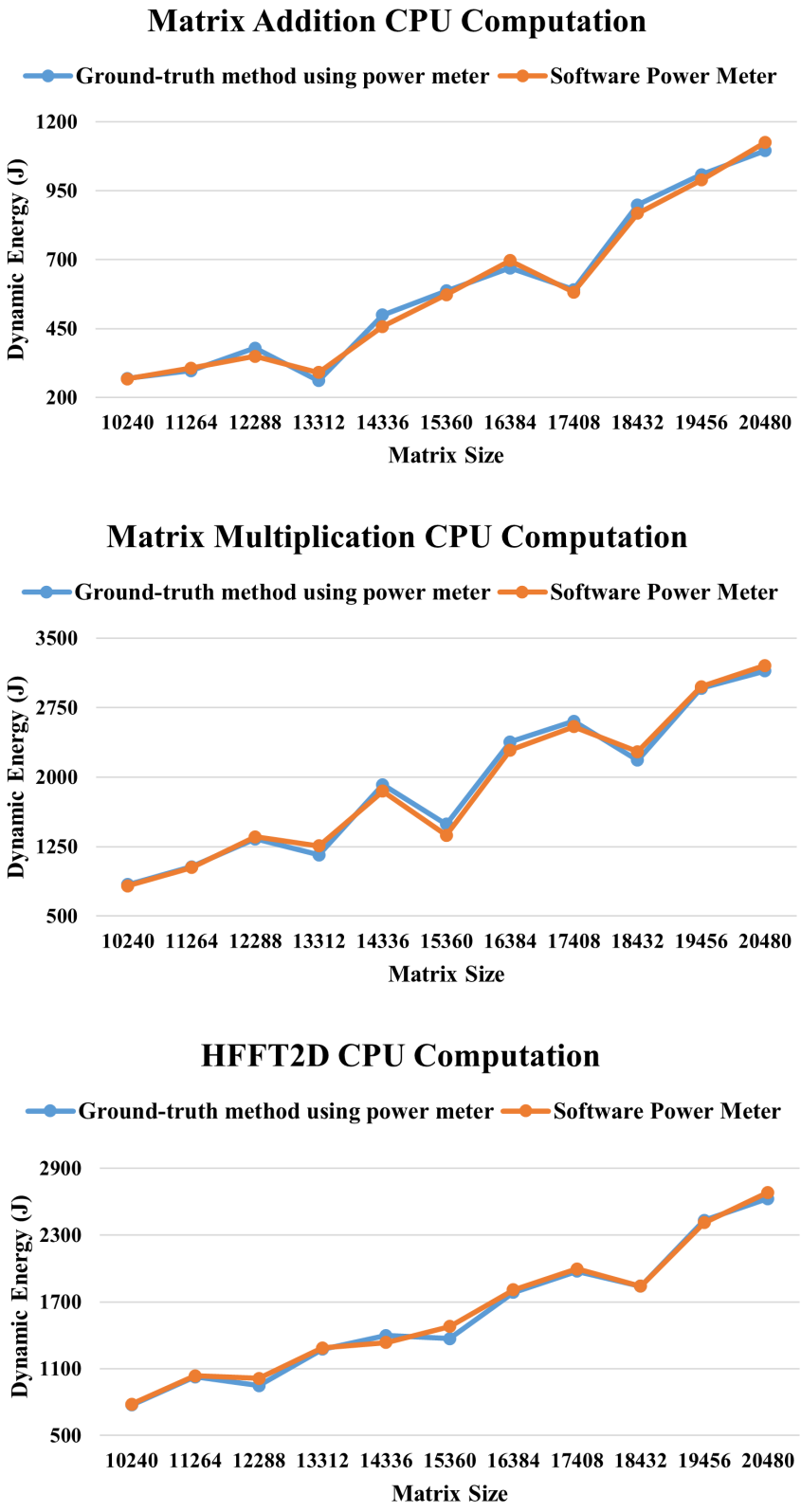


Figure 5.4: The plots compare the dynamic energy consumptions of CPU computation component estimated by the CPU software power meter (indicated in orange) and the ground truth measurement method (shown in blue) for the three parallel hybrid programs executed on the heterogeneous hybrid server (refer to Table 3.1).

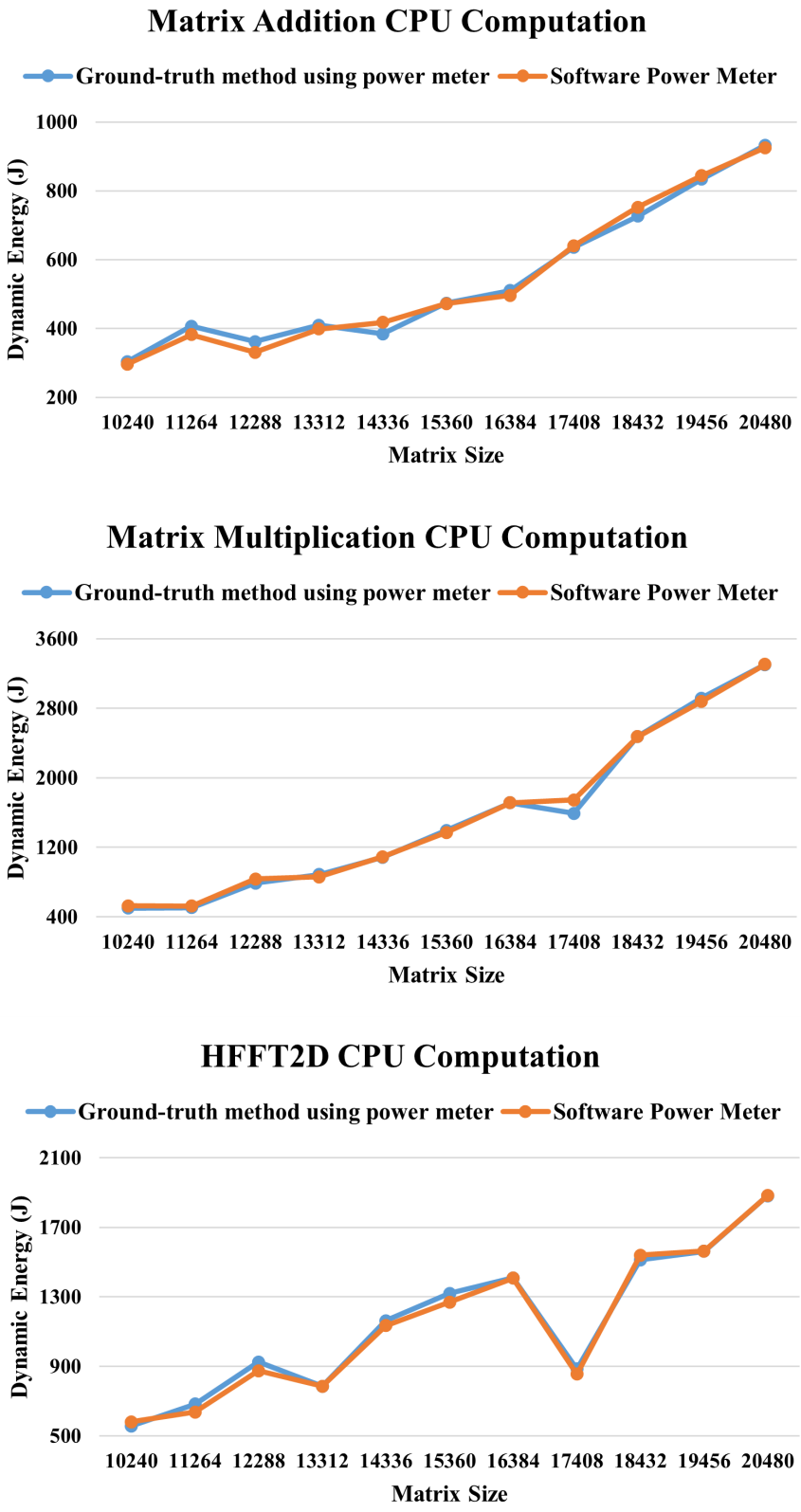


Figure 5.5: The plots compare the dynamic energy consumptions of CPU computation component estimated by the CPU software power meter (indicated in orange) and the ground truth measurement method (shown in blue) for the three parallel hybrid programs executed on the heterogeneous hybrid server (refer to Table 3.2).

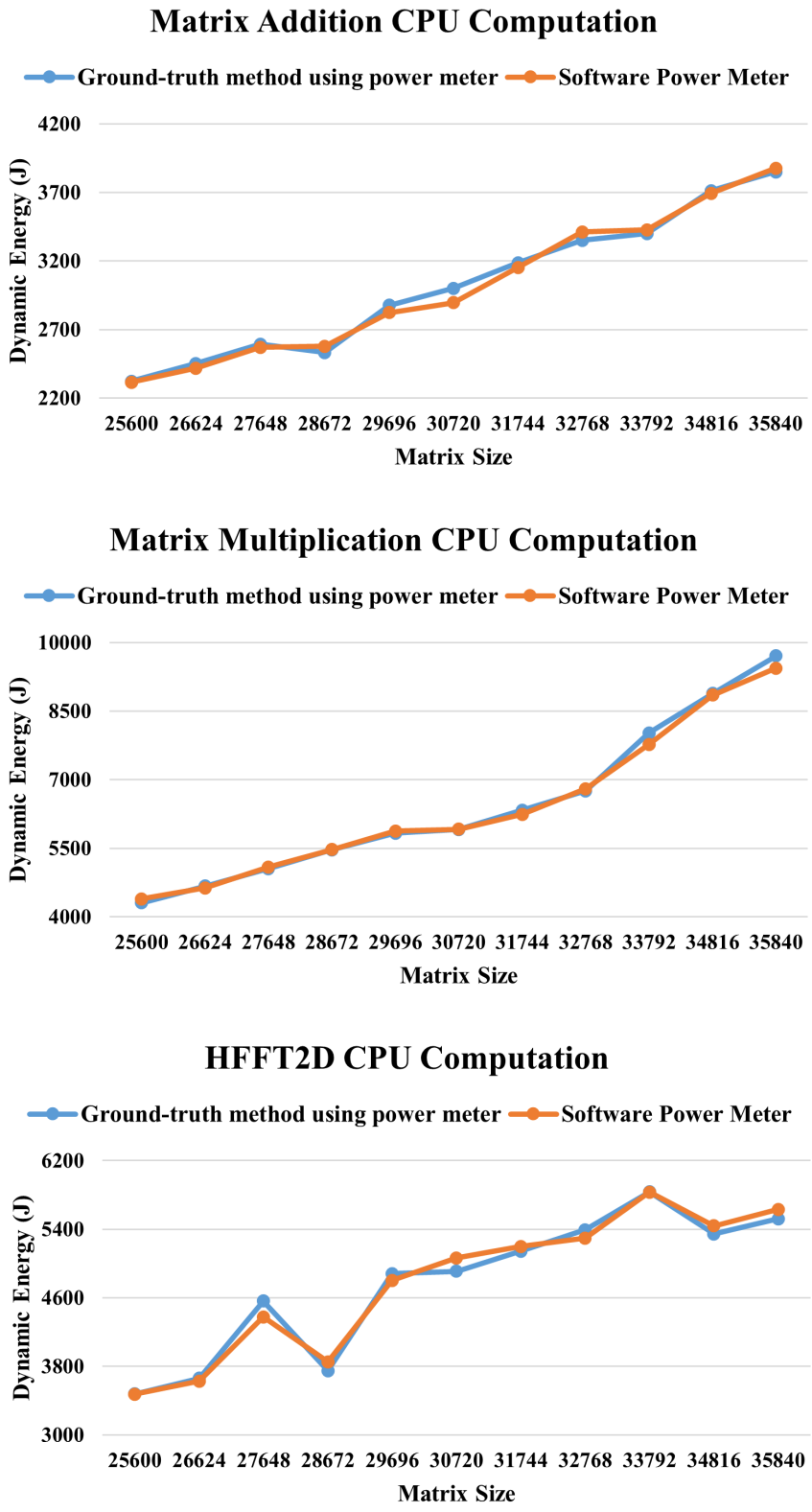


Figure 5.6: The plots compare the dynamic energy consumptions of CPU computation component estimated by the CPU software power meter (indicated in orange) and the ground truth measurement method (shown in blue) for the three parallel hybrid programs executed on the heterogeneous hybrid server (refer to Table 3.3).

5.5.3.2 Data Transfers

Figures 5.7 and 5.8 illustrate the energy predictions by the software power meter for bidirectional data transfers between the host CPU and GPUs of each heterogeneous hybrid servers described in Tables 3.1 and 3.2, across three different applications. The x-axis represents the matrix size N ranging from 10240 × 10240 to 20480 x 20480, while the y-axis indicates the dynamic energy consumption in joules.

Figure 5.9, 5.10 and 5.11 illustrate the energy predictions by the software power meters for the data transfers between the host CPU and A40 GPUs of heterogeneous hybrid server described in Table 3.3, across three different applications. The x-axis represents the matrix size N, ranging from 25600 × 25600 to 35840 × 35840, while the y-axis indicates the dynamic energy consumption in joules.

Each plot again features two curves: the blue curve shows the actual dynamic energy measured using ground-truth data from external power meters, while the orange curve displays the predicted energy consumption by the data transfer software power meter. The average prediction error remains at 2.5%.

Moreover, the software power meter for a specific data transfer activity records negligible dynamic energy consumption for phases when this activity is absent.

Figure 5.7: The plots compare the dynamic energy consumption of data transfers between the CPU and GPU estimated by the data transfer software power meters (indicated in orange) and the ground truth measurement method (shown in blue) for each parallel hybrid application executed on the heterogeneous hybrid server (Table 3.1).

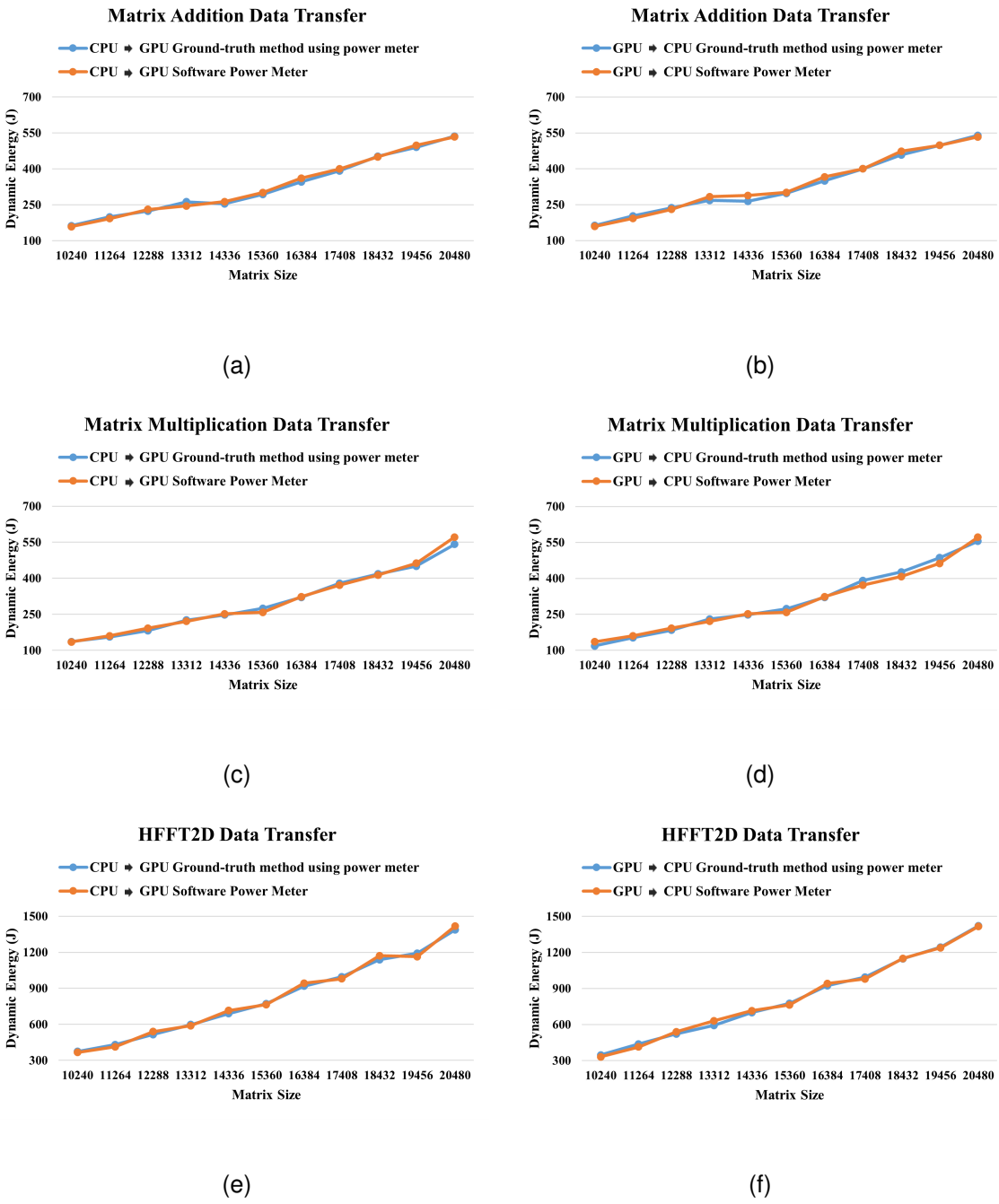
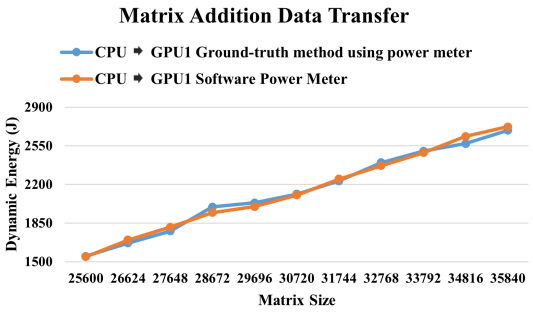
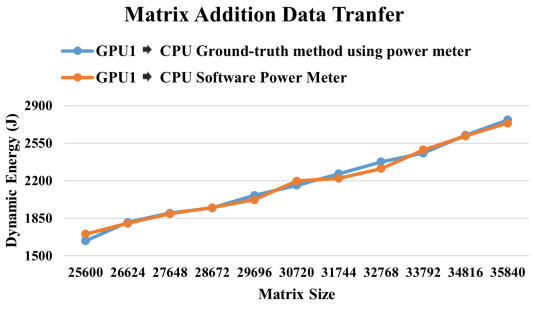


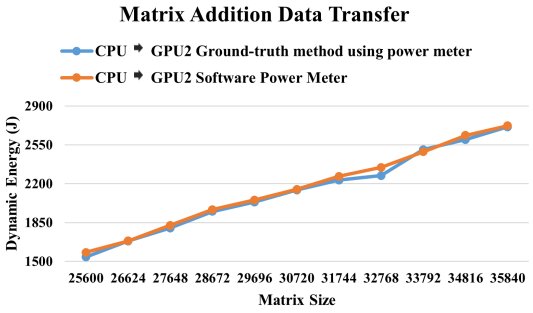
Figure 5.8: The plots compare the dynamic energy consumption of data transfers between the CPU and GPU estimated by the data transfer software power meters (indicated in orange) and the ground truth measurement method (shown in blue) for each parallel hybrid application executed on the heterogeneous hybrid server (Table 3.2).



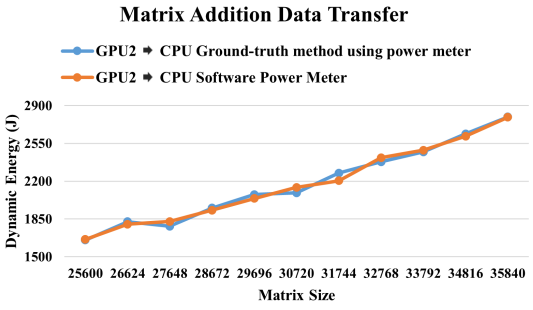
(a)



(b)

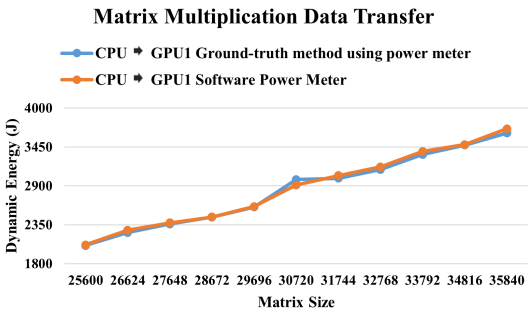


(c)

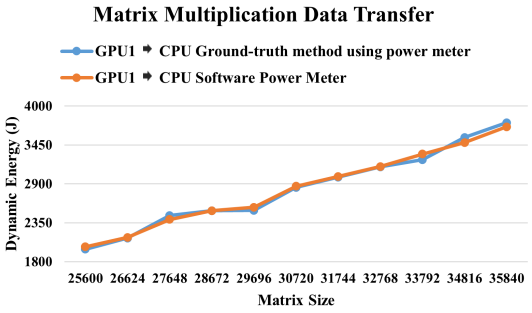


(d)

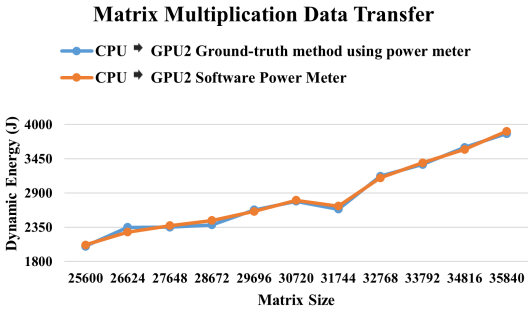
Figure 5.9: The plots compare the dynamic energy consumption of data transfers between the CPU to A40 GPUs and A40 GPUs to CPU estimated by the data transfer software power meters (indicated in orange) and the ground truth measurement method (shown in blue) for the parallel hybrid matrix addition application executed on the heterogeneous hybrid server (refer to Table 3.3).



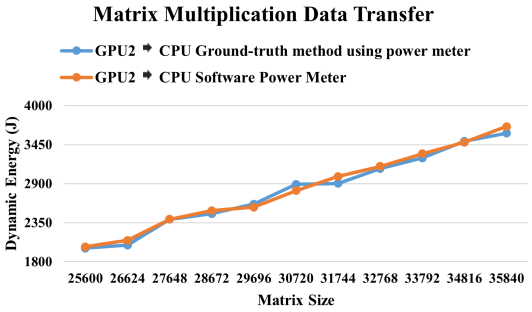
(a)



(b)



(c)



(d)

Figure 5.10: The plots compare the dynamic energy consumption of data transfers between the CPU to A40 GPUs and A40 GPUs to CPU estimated by the data transfer software power meters (indicated in orange) and the ground truth measurement method (shown in blue) for the parallel hybrid matrix multiplication application executed on the heterogeneous hybrid server (refer to Table 3.3).

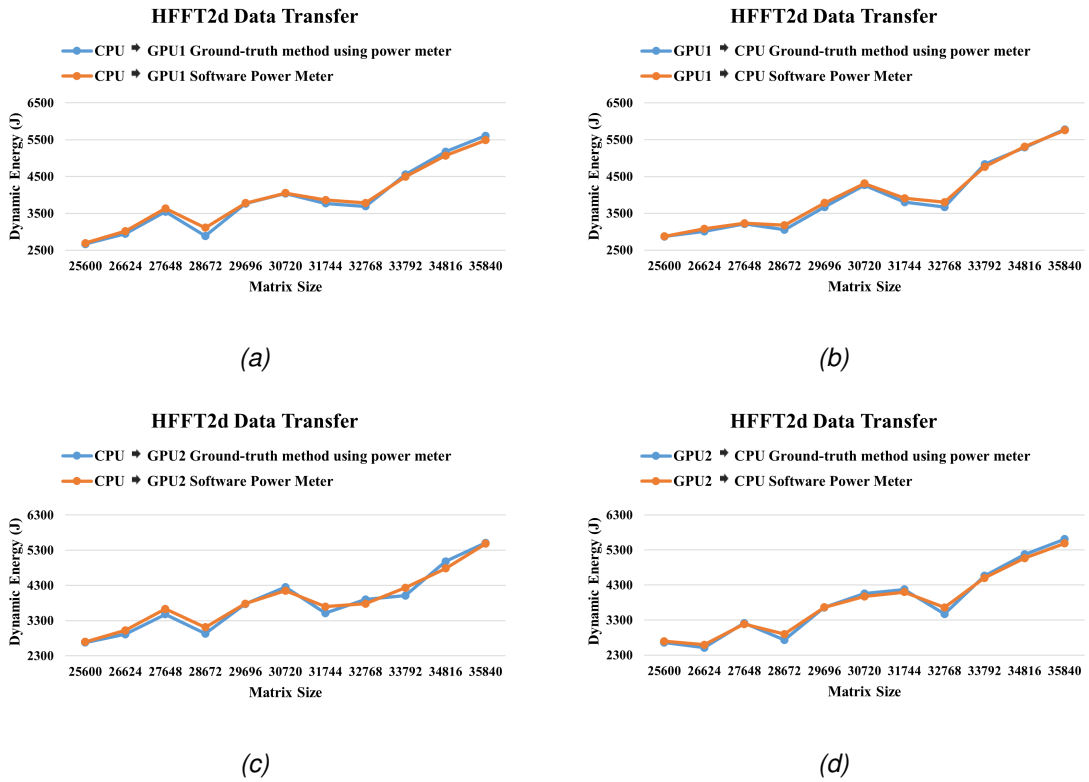


Figure 5.11: The plots compare the dynamic energy consumption of data transfers between the CPU to A40 GPUs and A40GPUs to CPU estimated by the data transfer software power meters (indicated in orange) and the ground truth measurement method (shown in blue) for the parallel hybrid hfft2d application executed on the heterogeneous hybrid server (refer to Table 3.3).

5.5.3.3 Total Dynamic Energy

Figures 5.12 and 5.13 shows the total dynamic energy consumption estimated by all our software power meters (indicated in orange) and the ground truth measurement method (shown in blue) for the three parallel hybrid programs on the heterogeneous hybrid servers described in Tables 3.1 and 3.2. The x-axis represents the matrix size N ranging from 10240 × 10240 to 20480 x 20480, while the y-axis indicates the dynamic energy consumption in joules.

The total dynamic energy consumption estimated using software power meters is calculated as the sum of the energy estimates of all the software power meters. For the computations on the GPU for servers as in Table 3.1 and 3.2, we utilize the ground-truth measurements since the on-chip sensors in the K40c and P100 GPUs have poor prediction accuracy.

The average prediction errors experimentally observed for Haswell K40c GPU server (in Table 3.1) are 3%, 4%, and 4% for the three parallel hybrid applications, matrix addition, matrix multiplication and 2D fast Fourier transform. For Skylake P100 GPU server (in Table 3.2), the average prediction errors experimentally observed are 2%, 3%, 4%.

Figure 5.14 illustrates the total dynamic energy consumption estimated by our software power meters, represented in orange. This is compared to the measurements obtained through the ground truth method, shown in blue, and an approach that uses the average of three repetitions with hardware power meters across the three parallel hybrid programs.

The x-axis represents the matrix size N, ranging from 25600 × 25600 to 35840 × 35840, while the y-axis indicates the dynamic energy consumption in joules. The total dynamic energy consumption estimated by the software power meters is calculated by summing the dynamic energy estimates from all the power meters.

The average prediction error when using hardware power meters with three repetitions is 25% compared to the ground truth measurements. In contrast, the average prediction errors of the software power meters against the ground truth for the three applications matrix addition, matrix multiplication and 2D fast Fourier transform are 3%, 3%, and 4%, respectively. However, these percentages are affected by the higher average prediction error of 5% from software power meters used for GPU computations. These meters rely on power readings from the GPUs' on-chip power sensors.

The experimental results confirm that our proposed software power meters are concurrent and orthogonal across three different hybrid CPU-GPU servers. Additionally, the software power meters demonstrate high prediction accuracy for each server. The average prediction error for dynamic energy consumption by these software power meters is just 2.5% across our servers.

Importantly, for each server, the same software power meters can effectively profile the dynamic energy consumption of all three hybrid programs, as they are designed to be system-level rather than application-specific. Thus, no modifications to the power meters are necessary to capture any missing application-specific energy-consuming activities.

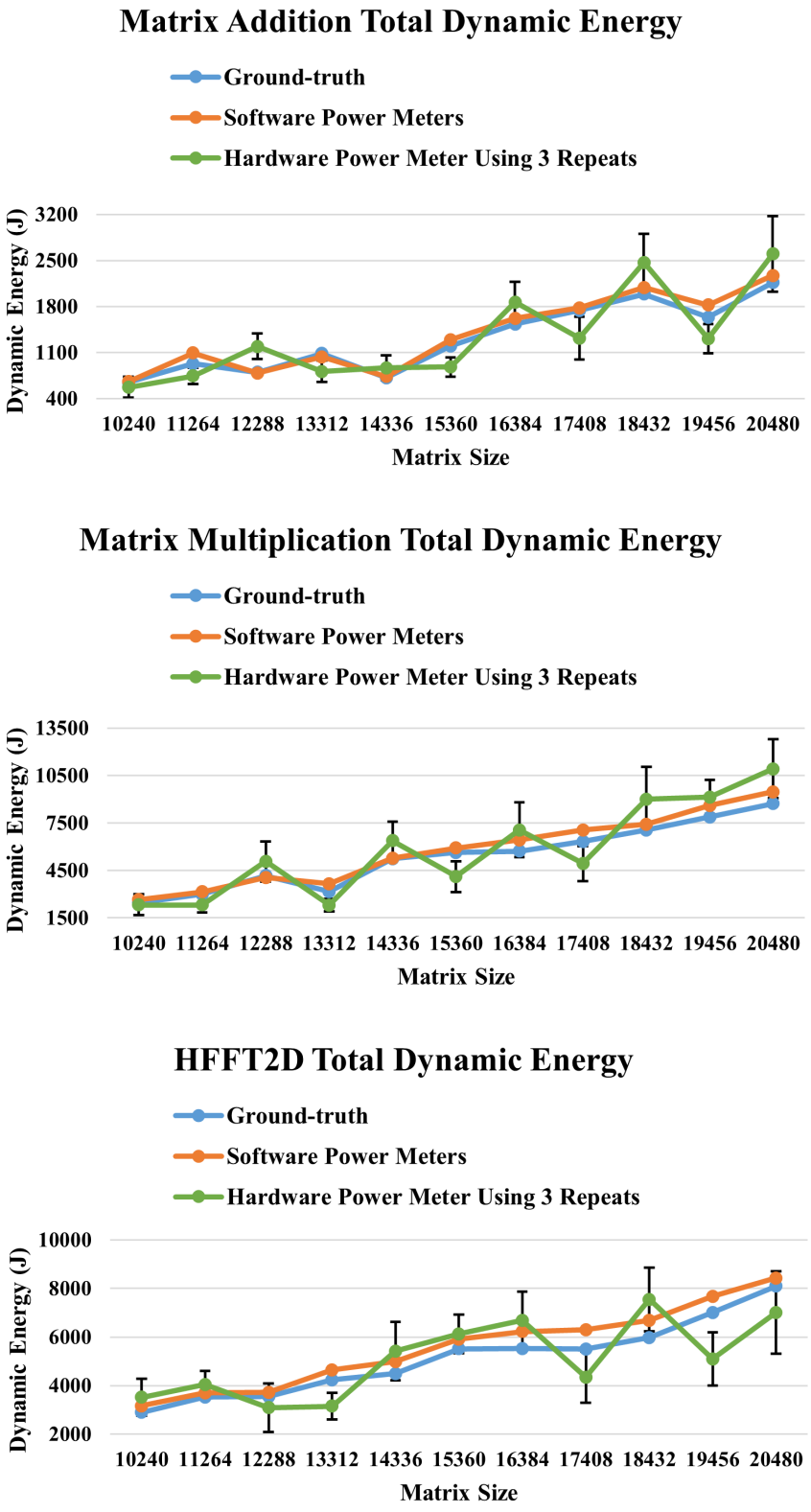


Figure 5.12: The plots demonstrate the total dynamic energy consumption recorded using three different approaches for the three parallel hybrid programs executed on the heterogeneous hybrid server (refer to Table 3.1). The approaches include software power meters (indicated in orange), the ground truth measurement method (shown in blue), and a method utilizing hardware power meters that employs the average of three repetitions for each data point (represented in green). Each green point is shown by a vertical error bar with the average value in the middle. The standard error of the green points ranges from ±18% to ±26% of the average.

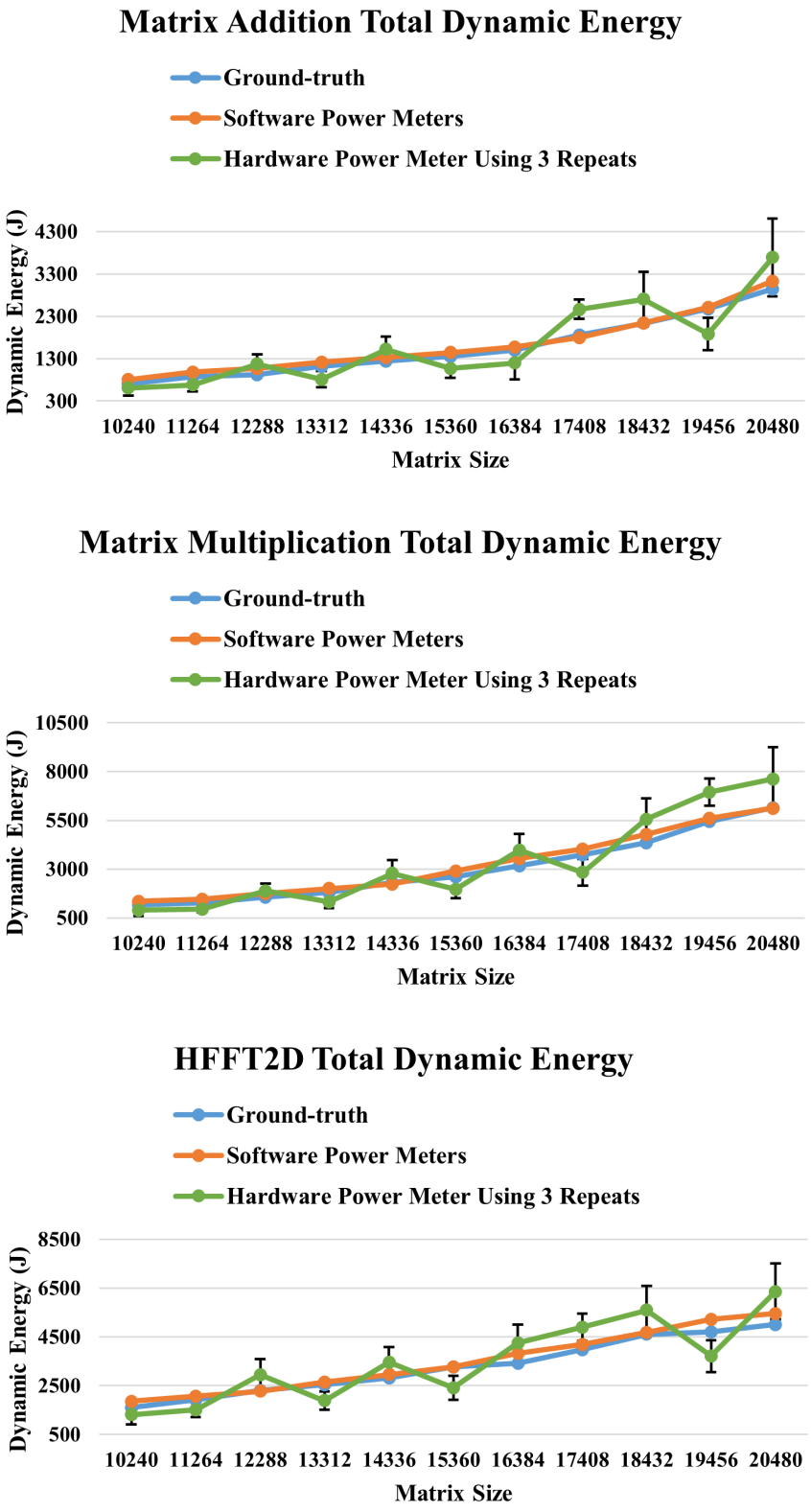


Figure 5.13: The plots demonstrate the total dynamic energy consumption recorded using three different approaches for the three parallel hybrid programs executed on the heterogeneous hybrid server (refer to Table 3.2). The approaches include software power meters (indicated in orange), the ground truth measurement method (shown in blue), and a method utilizing hardware power meters that employs the average of three repetitions for each data point (represented in green). Each green point is shown by a vertical error bar with the average value in the middle. The standard error of the green points ranges from ±16% to ±26% of the average.

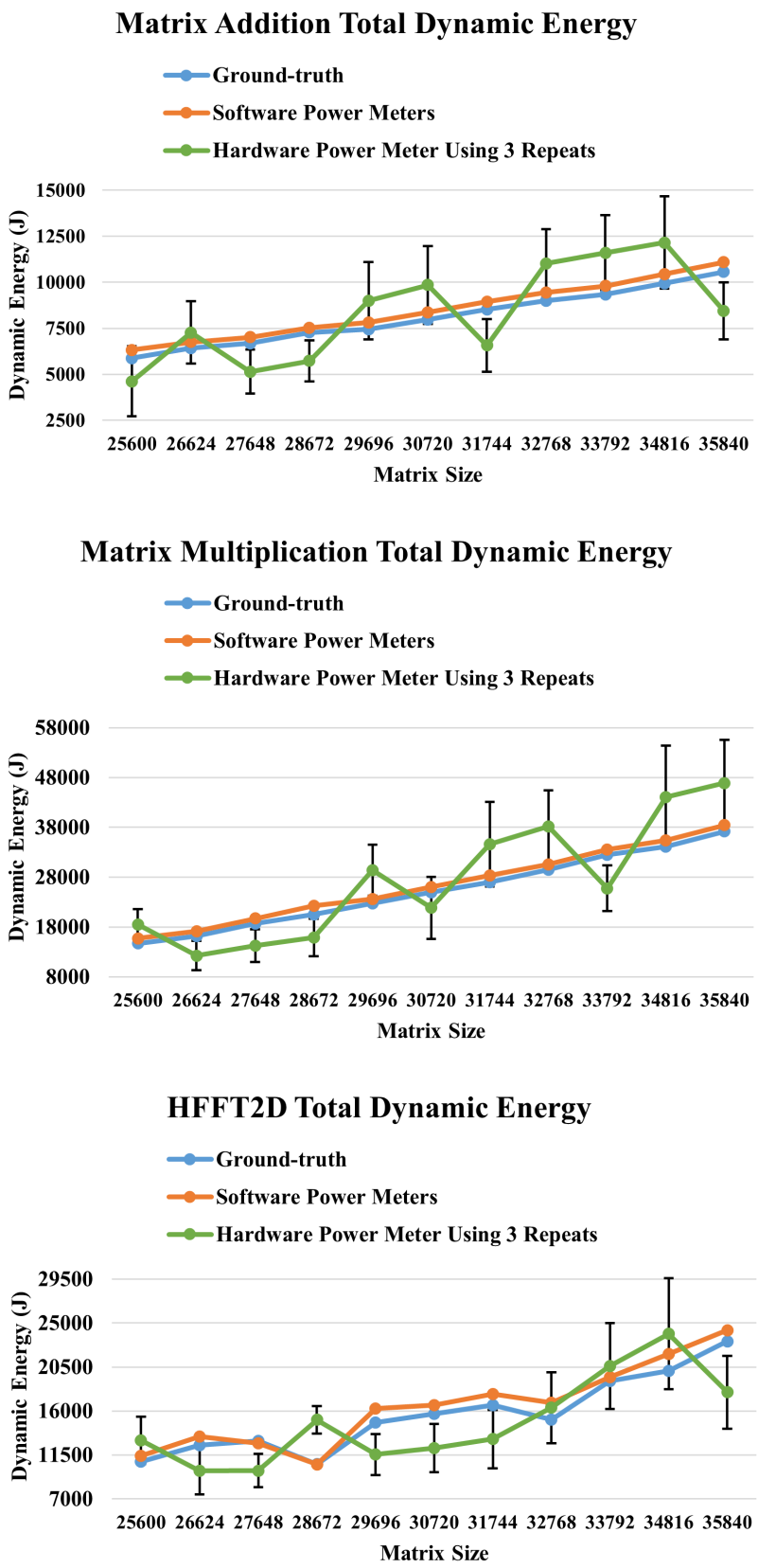


Figure 5.14: The plots demonstrate the total dynamic energy consumption recorded using three different approaches for the three parallel hybrid applications executed on the heterogeneous hybrid server (refer to Table 3.3). The approaches include software power meters (indicated in orange), the ground truth measurement method (shown in blue), and a method utilizing hardware power meters that employs the average of three repetitions for each data point (represented in green). Each green point is shown by a vertical error bar with the average value in the middle. The standard error of the green points ranges from ±16% to ±25% of the average.

5.6 Summary

Conducting runtime energy profiling for parallel hybrid programs on high-end digital servers presents a challenge, as it requires accurate and efficient measurement of the dynamic energy consumed by different parallel computation and communication activities of the program.

State-of-the-art software power meters have been developed using linear energy predictive models that employ shortlisted performance events to predict the energy consumption of a single activity, either computation or data transfer, during the execution of parallel hybrid programs, and the sum of the estimations of all computation and data transfer activities can be used as an estimation of the total dynamic energy consumption by the program.

However, these linear energy predictive models are not suitable for employment as software power meters for accurate runtime energy profiling of parallel hybrid programs running on heterogeneous hybrid servers, as they lack two fundamental properties: concurrency and orthogonality.

This is because the previously developed methodology identifies a small subset of performance events for a single activity only and does not account for the parallel execution of computation and communication activities. As a result, the performance event sets may not accurately reflect runtime energy profiling for parallel or concurrent execution of computation and communication activities in parallel hybrid programs. These limitations highlight the need for a novel methodology that ensures both concurrency and orthogonality in the design of software power meters.

In this chapter, we have defined these two essential properties and presents the first comprehensive methodology for constructing a set of concurrent and orthogonal software power meters that enable accurate runtime estimation of dynamic energy consumption associated with independently powered parallel computation and communication activities in parallel hybrid programs.

We applied this methodology to design and develop concurrent and orthogonal software power meters for three heterogeneous hybrid servers that combine Intel multicore CPUs and Nvidia GPUs from different generations. These software power meters implement system-level linear energy predictive models that employ disjoint sets of performance events, ensuring accurate and scalable runtime energy profiling across diverse

architectures. For each server, this methodology finds software power meters for computations taking place in the multicore CPU, computations in a GPU, and data transfers between the CPU and GPU (one power meter for each direction).

Finally, we demonstrated the high accuracy and efficiency of our concurrent and orthogonal software power meters by using them to estimate the dynamic energy consumption of independently powered parallel computation and communication activities in three parallel hybrid programs running on the three heterogeneous hybrid servers.

Our software power meters can accurately monitor all three hybrid programs because they are designed for system-level use rather than being tailored to individual applications. As a result, no modifications to the power meters are necessary to capture energy consumption activities specific to any application.

Chapter 6

Conclusion

This thesis addressed the research questions defined in Chapter 1.3.2. Specifically, we developed accurate ground-truth measurement methodologies (RQ1), efficient runtime software power meters based on performance-event predictive models (RQ2), and concurrent and orthogonal software power meters capable of independently measuring computation and communication energy (RQ3 and RQ4). The experimental results demonstrate that the proposed methodologies and software power meters achieve high accuracy and efficiency across multiple heterogeneous hybrid server platforms.

Heterogeneous hybrid servers with accelerators are getting increasingly popular in high-end digital platforms, such as HPC clusters, data centers, and clouds, to address the grand energy efficiency technological challenge. Developing energy-efficient software that leverages application-level energy optimization techniques is also essential to tackling the pressing technological challenge of energy efficiency on modern heterogeneous computing platforms.

Energy optimization of parallel programs on such servers is a challenge, as it requires an accurate and efficient measurement of the energy consumed by different parallel computation and communication activities of the program. A parallel hybrid program running on a heterogeneous hybrid server involves the concurrent execution of both computation and communication activities.

Historically, the main focus of energy researchers has been on developing solutions for energy modeling and optimization of computations, due to the widespread belief that the energy consumption from data transfers is insignificant and negligible. As a result, the energy consumption of device data transfers (communications) has remained unexplored, with no prior research addressing this issue.

Consequently, a significant gap exists in the measurement and modeling of energy consumption for data transfers between computing devices on heterogeneous hybrid servers. At present, there is neither a method to accurately measure the energy of data transfers nor a model to predict this energy between computing devices, and our study aims to address and fill this critical gap.

In this work, we comprehensively studied the energy consumption of data transfer between a host CPU and a GPU accelerator on heterogeneous hybrid servers using the three mainstream energy measurement methods: (a) System-level physical measurements based on external power meters *(ground-truth)*, (b) Measurements using on-chip power sensors, and (c) Energy predictive models.

First, we proposed a methodology to accurately measure the energy consumption of data transfer between a host CPU and a GPU using the ground-truth method. While accurate, the *ground-truth* method is prohibitively time-consuming. We then investigated the accuracy of on-chip sensor software tools to model and predict the energy of data transfer between a host CPU and a GPU.

We showed that the on-chip sensor software tool for Intel multicore CPUs is inaccurate. In addition, the on-chip sensor software tool (NVML) for Nvidia GPU processors does not capture data transfer activity and, therefore, does not provide the energy consumption of data transfer between a host CPU and GPU.

Therefore, we focused on the third approach, which employs energy predictive models using performance events. Among the three main approaches for energy profiling, it is the third approach that is a promising alternative due to its cost-effectiveness, lower time consumption, and better runtime accuracy compared to the other two approaches.

However, selecting a small subset of performance events that can be obtained in one application run and effectively capture all the energy consumption activities during a data transfer is a challenging problem. We proposed a methodology that employs a fast selection procedure to solve the problem based on the natural grouping of performance events derived from the processor architecture, additivity test, and correlation to estimate the energy consumption of a single activity.

We then designed accurate and reliable software power meters based on linear energy predictive models employing the selected performance events for the data transfer activity. Finally, we developed independent and accurate runtime software power meters

based on our proposed energy predictive models that employ performance events to estimate the dynamic energy consumption of a single activity whether it is computation or data transfers.

We tested the accuracy of our software power meters using three parallel scientific programs (matrix addition, matrix multiplication, and fast Fourier transform) on a heterogeneous server comprising one Intel Icelake multicore CPU and two Nvidia A40 GPUs. The results show that the software power meters achieve high accuracy with an average prediction error of 1% for computation activity, and 6% for data transfer activity across all three parallel programmes against the *ground-truth*.

The state-of-the-art software power meters based on linear energy predictive models seem promising for predicting the energy consumption of a single activity, either computation or data transfers, during the execution of parallel hybrid programs, and the sum of the estimations of all computation and data transfer activities can be used as an estimation of the total dynamic energy consumption by the program.

However, these linear energy predictive models are not suitable for employment as software power meters for accurate runtime energy profiling of parallel hybrid programs running on heterogeneous hybrid servers, as they lack two fundamental properties: concurrency and orthogonality.

This is because the previously developed methodology identifies a small subset of performance events for a single activity only and does not account for the parallel execution of computation and communication activities. As a result, the performance event sets may not accurately reflect runtime energy profiling for parallel or concurrent execution of computation and communication activities in parallel hybrid programs. These limitations highlight the need for a novel methodology that ensures both concurrency and orthogonality in the design of software power meters.

We formulate and define these key fundamental properties for software power meters that are essential for accurately profiling runtime energy consumption of parallel hybrid programs on heterogeneous hybrid servers. We then present a methodology for developing concurrent and orthogonal software power meters that provide accurate runtime estimation of energy consumption associated with independently powered parallel computation and communication activities in parallel hybrid programs.

We applied this methodology to design and develop concurrent and orthogonal software power meters for three heterogeneous hybrid servers that combine Intel multicore CPUs and Nvidia GPUs from different generations. These software power meters implement system-level linear energy predictive models that employ disjoint sets of performance events, ensuring accurate and scalable runtime energy profiling across diverse architectures. For each server, this methodology finds software power meters for computations taking place in the multicore CPU, computations in a GPU, and data transfers between the CPU and GPU (one power meter for each direction).

Finally, we demonstrated the high accuracy and efficiency of our concurrent and orthogonal software power meters by using them to estimate the dynamic energy consumption of independently powered parallel computation and communication activities in three parallel hybrid programs running on the three heterogeneous hybrid servers. The average prediction error for dynamic energy consumption by these software power meters is only 2.5% across our servers.

Our software power meters can accurately monitor all three hybrid programs because they are designed for system-level use rather than being tailored to individual applications. As a result, no modifications to the software power meters are necessary to capture energy consumption activities specific to any application.

6.1 Limitations of This Work

While this thesis presents accurate and practical methodologies and software power meters for measuring and predicting the energy consumption of computation and data transfer activities on heterogeneous hybrid servers, several limitations remain that provide opportunities for further research and development.

First, this work focuses on energy measurement and modeling of data transfers between host CPUs and GPUs within a single heterogeneous hybrid server. Energy consumption associated with communication between GPUs through direct interconnects such as NVLink or emerging interconnects such as CXL, as well as communication across multiple nodes in distributed systems using communication frameworks such as MPI, has not been explicitly investigated.

Second, the developed methodologies and software power meters were validated on

three representative heterogeneous hybrid servers comprising Intel multicore CPUs and Nvidia GPUs. Although these platforms span multiple processor generations, further validation across a wider range of architectures, including other accelerator technologies such as AMD GPUs and FPGAs, would strengthen the general applicability of the proposed methodology.

Third, this work focuses on system-level energy consumption associated with computation and inter-device data transfers. Finer-grained energy accounting associated with on-chip data movement, including memory hierarchy traffic such as cache, shared memory, and memory controller activity, was not explicitly modeled.

Finally, although the developed software power meters provide accurate runtime energy profiling, their integration into production runtime systems, schedulers, and large-scale HPC or cloud environments was beyond the scope of this thesis. Such integration would enable real-time energy-aware scheduling, resource management, and application optimization, and represents an important direction for future work.

Despite these limitations, this thesis establishes a practical and robust foundation for accurate measurement and runtime profiling of energy consumption in heterogeneous hybrid computing systems.

6.2 Future Work

This research opens several promising directions for future investigation and development.

First, the proposed methodologies and software power meters can be extended to measure and model the energy consumption of communication across multiple GPUs connected through high-speed interconnects such as NVLink, PCIe peer-to-peer, and emerging interconnect technologies such as CXL. This will enable accurate energy profiling of modern multi-accelerator systems.

Second, future work can extend the methodology to distributed heterogeneous computing environments involving multiple compute nodes connected through network interconnects such as InfiniBand and Ethernet. In particular, extending the methodology to model and measure MPI-based communication energy will enable comprehensive energy profiling of large-scale HPC and cloud-based parallel applications.

Third, further research can explore finer-grained energy modeling of on-chip data movement, including energy consumption associated with memory hierarchy components such as cache levels, memory controllers, and accelerator memory subsystems. This will enable more detailed decomposition of application energy consumption.

Fourth, the developed software power meters can be integrated into runtime systems, schedulers, and auto-tuning frameworks to enable energy-aware scheduling, optimization, and dynamic decision-making in heterogeneous computing environments.

Fifth, future work can extend the methodology to support other accelerator architectures, including AMD GPUs, FPGAs, and emerging heterogeneous computing platforms, improving portability and applicability across diverse computing environments.

Finally, combining the developed energy profiling methodologies with performance, cost, and carbon footprint models will enable multi-objective optimization of applications, supporting energy-efficient and environmentally sustainable computing.

Bibliography

[1] H. A. Niaz, R. R. Manumachu, and A. Lastovetsky, "Accurate and reliable energy measurement and modelling of data transfer between CPU and GPU in parallel applications on heterogeneous hybrid platforms," *IEEE Transactions on Computers*, vol. 74, no. 3, pp. 1011–1024, 2025.

[2] H. A. Niaz, R. R. Manumachu, and A. Lastovetsky, "Concurrent and orthogonal software power meters for accurate runtime energy profiling of parallel hybrid programs on heterogeneous hybrid servers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 37, no. 2, pp. 322–339, 2025.

[3] A. SG Andrae, "New perspectives on internet electricity use in 2030," *Engineering and Applied Science Letter*, vol. 3, no. 2, pp. 19–31, 2020.

[4] C.-J. Wu, "Special issue on environmentally sustainable computing," *IEEE Micro*, vol. 43, no. 01, pp. 7–8, 2023.

[5] A. Lastovetsky and R. R. Manumachu, "New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1119–1133, 2016.

[6] R. R. Manumachu and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on homogeneous multicore clusters for performance and energy," *IEEE Transactions on Computers*, vol. 67, no. 2, pp. 160–177, 2017.

[7] A. Chakrabarti, S. Parthasarathy, and C. Stewart, "A pareto framework for data analytics on heterogeneous systems: Implications for green energy usage and performance," in *Parallel Processing (ICPP), 2017 46th International Conference on*, pp. 533–542, IEEE, 2017.

[8] 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, 2021.

[9] A. Lastovetsky and R. R. Manumachu, "Energy-efficient parallel computing: Challenges to scaling," *Information*, vol. 14, no. 4, 2023.

[10] J. Kołodziej, S. U. Khan, L. Wang, and A. Y. Zomaya, "Energy efficient genetic-based schedulers in computational grids," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 4, pp. 809–829, 2015.

[11] Y. Kessaci, N. Melab, and E.-G. Talbi, "A pareto-based metaheuristic for scheduling hpc applications on a geographically distributed cloud federation," *Cluster Computing*, vol. 16, pp. 451–468, 2013.

[12] 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*, pp. 179–191, 2016.

[13] L. Yu, Z. Zhou, S. Wallace, M. E. Papka, and Z. Lan, "Quantitative modeling of power performance tradeoffs on extreme scale systems," *Journal of Parallel and Distributed Computing*, vol. 84, pp. 1–14, 2015.

[14] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of eulag kernel on intel xeon phi through load imbalancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 787–797, 2016.

[15] S. Khokhriakov, R. R. Manumachu, and A. Lastovetsky, "Multicore processor computing is not energy proportional: An opportunity for bi-objective optimization for energy and performance," *Applied energy*, vol. 268, p. 114957, 2020.

[16] H. Khaleghzadeh, R. Reddy Manumachu, and A. Lastovetsky, "Efficient exact algorithms for continuous bi-objective performance-energy optimization of applications with linear energy and monotonically increasing performance profiles on heterogeneous high performance computing platforms," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 20, p. e7285, 2023.

[17] R. R. Manumachu and A. Lastovetsky, "On energy nonproportionality of cpus and gpus," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 34–44, IEEE, 2022.

[18] A. Shahid, M. Fahad, R. R. Manumachu, and A. Lastovetsky, "Energy predictive models of computing: theory, practical implications and experimental analysis on multicore processors," *IEEE Access*, vol. 9, pp. 63149–63172, 2021.

[19] A. Shahid, M. Fahad, R. R. Manumachu, and A. Lastovetsky, "Improving the accuracy of energy predictive models for multicore cpus by combining utilization and performance events model variables," *Journal of Parallel and Distributed Computing*, vol. 151, pp. 38–51, 2021.

[20] M. Fahad and R. R. Manumachu, "HCLWattsUp: energy API using system-level physical power measurements provided by power meters," May 2022.

[21] M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "A comparative study of methods for measurement of energy of computing," *Energies*, vol. 12, no. 11, p. 2204, 2019.

[22] A. Noureddine, R. Rouvoy, and L. Seinturier, "A review of energy measurement approaches," *ACM SIGOPS Operating Systems Review*, vol. 47, no. 3, pp. 42–49, 2013.

[23] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with papi," in *2012 41st international conference on parallel processing workshops*, pp. 262–268, IEEE, 2012.

[24] H. Khaleghzadeh, M. Fahad, R. Reddy Manumachu, and A. Lastovetsky, "A novel data partitioning algorithm for dynamic energy optimization on heterogeneous high-performance computing platforms," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 21, p. e5928, 2020.

[25] A. Shahid, M. Fahad, R. R. Manumachu, and A. Lastovetsky, "A comparative study of techniques for energy predictive modeling using performance monitoring counters on modern multicore cpus," *IEEE Access*, vol. 8, pp. 143306–143332, 2020.

[26] D. Hackenberg, T. Ilsche, R. Schöne, D. Molka, M. Schmidt, and W. E. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," in *IEEE international symposium on Performance analysis of systems and software (ISPASS)*, pp. 194–204, IEEE, April 2013.

[27] M. Burtscher, I. Zecena, and Z. Zong, "Measuring GPU power with the K20 built-in sensor," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pp. 28:28–28:36, ACM, 2014.

[28] Nvidia, "Nvidia management library: Nvml api reference guide.," 2024.

[29] K. O'brien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou, "A survey of power and energy predictive models in hpc systems and applications," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, pp. 1–38, 2017.

[30] E. Calore, A. Gabbana, S. F. Schifano, and R. Tripiccione, "Thunderx2 performance and energy-efficiency for hpc workloads," *Computation*, vol. 8, no. 1, p. 20, 2020.

[31] M. B. Srivastava, A. P. Chandrakasan, and R. W. Brodersen, "Predictive system shutdown and other architectural techniques for energy efficient programmable computation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 1, pp. 42–55, 1996.

[32] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of simplepower: a cycle-accurate energy estimation tool," in *Proceedings of the 37th Annual Design Automation Conference*, pp. 340–345, ACM, 2000.

[33] T. Heath, B. Diniz, B. Horizonte, E. V. Carrera, and R. Bianchini, "Energy conservation in heterogeneous server clusters," in *10th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pp. 186–195, ACM, 2005.

[34] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *34th Annual International Symposium on Computer architecture*, pp. 13–23, ACM, 2007.

[35] A. Kansal and F. Zhao, "Fine-grained energy profiling for power-aware application design," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, p. 26, Aug. 2008.

[36] F. Bellosa, “The benefits of event: driven energy accounting in power-sensitive systems,” in *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, ACM, 2000.

[37] C. Isci and M. Martonosi, “Runtime power monitoring in high-end processors: Methodology and empirical data,” in *36th annual IEEE/ACM International Symposium on Microarchitecture*, p. 93, IEEE Computer Society, 2003.

[38] T. Li and L. K. John, “Run-time modeling and estimation of operating system power consumption,” *SIGMETRICS Performance Evaluation Review*, vol. 31, pp. 160–171, June 2003.

[39] B. C. Lee and D. M. Brooks, “Accurate and efficient regression modeling for microarchitectural performance and power prediction,” *SIGARCH Computer Architecture News*, vol. 34, pp. 185–194, Oct. 2006.

[40] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade, “Decomposable and responsive power models for multicore processors using performance counters,” in *Proceedings of the 24th ACM International Conference on Supercomputing*, pp. 147–158, ACM, 2010.

[41] W. L. Bircher and L. K. John, “Complete system power estimation using processor performance events,” *IEEE Transactions on Computers*, vol. 61, pp. 563–577, Apr. 2012.

[42] R. Basmadjian and H. de Meer, “Evaluating and modeling power consumption of multi-core processors,” in *Future Energy Systems: Where Energy, Computing and Communication Meet (e-Energy), 2012 Third International Conference on*, pp. 1–10, May 2012.

[43] J. Haj-Yihia, A. Yasin, Y. B. Asher, and A. Mendelson, “Fine-grain power breakdown of modern out-of-order cores and its implications on skylake-based systems,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, p. 56, 2016.

[44] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan, “Full-system power analysis and modeling for server environments,” in *In Proceedings of Workshop on Modeling, Benchmarking, and Simulation*, pp. 70–77, 2006.

[45] S. Rivoire, P. Ranganathan, and C. Kozyrakis, “A comparison of high-level full-system power models,” in *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower’08, USENIX Association, 2008.

[46] A. Shahid, M. Fahad, R. Reddy, and A. Lastovetsky, “Additivity: A selection criterion for performance events for reliable energy predictive modeling,” *Supercomputing Frontier Innovations*, vol. 4, pp. 50–65, Dec. 2017.

[47] A. Shahid, M. Fahad, R. R. Manumachu, and A. L. Lastovetsky, “Energy predictive models of computing: Theory, practical implications and experimental analysis on multicore processors,” *IEEE Access*, vol. 9, pp. 63149–63172, 2021.

[48] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, “Statistical power modeling of GPU kernels using performance counters,” in *International Green Computing Conference and Workshops (IGCC)*, IEEE, 2010.

[49] Y. Zhang, Y. Hu, B. Li, and L. Peng, “Performance and power analysis of ATI GPU: A statistical approach,” in *Proceedings of the IEEE Sixth International Conference on Networking, Architecture, and Storage*, NAS ’11, pp. 149–158, IEEE Computer Society, 2011.

[50] S. Song, C. Su, B. Rountree, and K. W. Cameron, “A simplified and accurate model of power-performance efficiency on emergent GPU architectures,” in *27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 673–686, IEEE Computer Society, 2013.

[51] Y. Abe, H. Sasaki, S. Kato, K. Inoue, M. Edahiro, and M. Peres, “Power and performance characterization and modeling of GPU-accelerated systems,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 113–122, 2014.

[52] V. Adhinarayanan, B. Subramaniam, and W.-c. Feng, “Online power estimation of graphics processing units,” in *Proceedings of the 16th IEEE/ACM International Sym-*

posium on Cluster, Cloud, and Grid Computing, CCGRID '16, p. 245–254, IEEE Press, 2016.

[53] J. a. Guerreiro, A. Ilic, N. Roma, and P. Tomás, "Modeling and decoupling the GPU power consumption for cross-domain DVFS," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, p. 2494–2506, Nov. 2019.

[54] F. Wang, W. Zhang, S. Lai, M. Hao, and Z. Wang, "Dynamic GPU energy optimization for machine learning training workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2943–2954, 2022.

[55] M. Gamell, I. Rodero, M. Parashar, *et al.*, "Exploring power behaviors and tradeoffs of in-situ data analytics," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[56] 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, 2021.

[57] T. Malik and A. Lastovetsky, "Towards optimal matrix partitioning for data parallel computing on a hybrid heterogeneous server," *IEEE Access*, vol. 9, pp. 17229–17244, 2021.

[58] M. Fahad, A. Shahid, R. R. Manumachu, and A. L. Lastovetsky, "Accurate energy modelling of hybrid parallel applications on modern heterogeneous computing platforms using system-level measurements," *IEEE Access*, vol. 8, pp. 93793–93829, 2020.

[59] H. C. L. (HCL), "Accurate and reliable energy measurement and modeling of data transfer between software components in parallel hybrid applications on heterogeneous platforms." `https://csgitlab.ucd.ie/manumachu/libedm.git`, February 2024.

[60] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *Ieee micro*, vol. 32, no. 2, pp. 20–27, 2012.

[61] Nvidia, "Nvidia management library: Nvml api reference guide.." `https://docs.nvidia.com/deploy/nvml-api/index.html` "[Online accessed: 17-Febraury-2024]", 2024.

[62] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th international conference on parallel processing workshops*, pp. 207–216, IEEE, 2010.

[63] NVIDIA Corporation, "CUDA profiling tools interface (CUPTI) - 1.0," 2021.

[64] P. J. Huber, *Robust Estimation of a Location Parameter*, pp. 492–518. New York, NY: Springer New York, 1992.

[65] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009.

[66] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," in *IMPACT Technical Report*, 2012. University of Illinois at Urbana-Champaign.

[67] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, (New York, NY, USA), p. 63–74, Association for Computing Machinery, 2010.

[68] L.-N. Pouchet, T. Yuki, and S. Rajopadhye, "Polybench/gpu: A benchmark suite for gpu computing," in *Workshop on General Purpose Processing Using GPUs (GPGPU)*, ACM, 2016.

[69] NVIDIA Corporation, "CUDA samples," 2023.

[70] M. Murnane and R. R. Manumachu, “Parallel hybrid programs on heterogeneous hybrid servers.” `https://csgitlab.ucd.ie/manumachu/openh/-/tree/master/Applications?ref_type=heads`, 2025.

Appendix A

Energy Measurements of Data Transfer on Heterogeneous Hybrid Servers

A.1 libedm: Library Functions To Obtain Dynamic Energy Using Ground-truth Method

Libedm [59] library provides three API functions, *hcl_measure_tenergy()*, *hcl_measure_time()*, and *hcl_measure_denergy()*, to obtain the total energy consumption, execution time, and dynamic energy consumption of a set of parallel data transfers between disjoint pairs of computing devices in a heterogeneous hybrid server.

Libedm reuses the *HCLWattsUp* API [20], which provides interface functions that gather the readings from the power meters to determine the average power and energy consumption during the execution of an application for the whole node.

Each function accepts an input argument, pIn, that contains a confidence interval and a precision. Therefore, the function executes the set of parallel data transfers repeatedly until the sample average (of the response variable, such as total energy) lies within the given confidence interval, and the given precision is achieved.

The API function *hcl_measure_tenergy()* determines the total energy consumption of $lp2p$ parallel data transfers between $lp2p$ disjoint pairs of computing devices.

```
int
hcl_measure_tenergy(
   const int verbosity,
   const hcl::Precision* pIn,
   const int lp2p, const int nTransfers,
   const size_t* nBytes,
   const int* lcores, const int** coreArray,
```

```
    EDMInit* edmInitFunc, EDMMain* edmMainFunc,
    EDMFinalize* edmFinalizeFunc,
    double *energymean, double *energysdev,
    double *bandwidths, hcl::Precision* pOut
);
```

The input argument pIn contains the statistical confidence parameters. The array $nBytes$ is of size $lp2p$, specifying the amount of data in bytes transferred between pairs of devices. The array element, $nBytes[i]$, contains the amount of data in bytes transferred between a CPU and an accelerator represented by i. $nTranfers$ specifies the number of data transfers between a CPU and an accelerator.

The input arguments, $lcores$ and $coreArray$, specify the cores to bind the data transfers. The array $lcores$ is of size $lp2p$ where the array element $lcores[i]$ gives the number of cores used to bind the data transfers between a CPU and an accelerator represented by i. The array element $coreArray[i]$ of size $lcores[i]$ contains the values of the cores used to bind the data transfers between a CPU and an accelerator represented by i.

The API function *hcl_measure_tenergy()* accepts three function pointers, *edmInitFunc*, *edmMainFunc*, and *edmFinalizeFunc*.

```
typedef int (*EDMInit) (
     const int, const size_t,
     const size_t, EDMThreadData*);
typedef void* (*EDMMain) (void*);
typedef int (*EDMFinalize) (EDMThreadData*);
```

The function *edmInitFunc* allocates and initializes the data and invokes library initialization functions specific to certain computing devices (such as GPU accelerators) needed for the parallel data transfers.

The function *edmMainFunc* executes the $lp2p$ parallel data transfers between $lp2p$ disjoint pairs of computing devices. It launches $lp2p$ pthreads where pthread i performs the data transfers between a CPU and an accelerator represented by i.

The function *edmFinalizeFunc* releases the resources and calls library finalization functions specific to certain computing devices (such as GPU accelerators).

hcl_measure_tenergy() returns four arguments, $energymean$, $energysdev$, $bandwidths$, and $pOut$ containing the total energy consumption, the standard de-

viation of the measurements, the average bandwidths of the $lp2p$ links between the $lp2p$ disjoint pairs of computing devices, and the precision achieved.

Furthermore, *hcl_measure_tenergy()* returns 0 if the precision is achieved and 1 otherwise. If statistical confidence criteria are not met, the output argument $pOut$ contains the precision achieved.

The API function *hcl_measure_time()* below determines the execution time of $lp2p$ parallel data transfers between $lp2p$ disjoint pairs of computing devices.

```
int
hcl_measure_time(
   const int verbosity,
   const hcl::Precision* pIn,
   const int lp2p, const int nTransfers,
   const size_t* nBytes, const int* lcores,
   const int** coreArray,
   EDMInit* edmInitFunc, EDMMain* edmMainFunc,
   EDMFinalize* edmFinalizeFunc,
   double *timemean, double *timesdev,
   double *bandwidths, hcl::Precision* pOut
);
```

The API function returns four arguments, $timemean$, $timesdev$, $bandwidths$, and $pOut$ containing the total execution time, the standard deviation of the measurements, the average bandwidths of the $lp2p$ links between the $lp2p$ disjoint pairs of computing devices, and the precision achieved.

Furthermore, *hcl_measure_time()* returns 0 if the precision is achieved and 1 otherwise. If statistical confidence criteria are not met, the output argument $pOut$ contains the precision achieved.

Finally, the API function *hcl_measure_denergy()* determines the dynamic energy consumption of $lp2p$ parallel data transfers between $lp2p$ disjoint pairs of computing devices. The input argument $basePower$ specifies the idle power consumption of the server.

```
int
hcl_measure_denergy(
   const int verbosity,
   const hcl::Precision* pIn,
   const double basePower,
   const int lp2p, const int nTransfers,
```

```
    const size_t* nBytes, const int* lcores,
    const int** coreArray,
    EDMInit* edmInitFunc, EDMMain* edmMainFunc,
    EDMFinalize* edmFinalizeFunc,
    double* timemean, double* timesdev,
    double *energymean, double *energydev,
    double *bandwidths, hcl::Precision* pOut
);
```

The API function returns six arguments, $timemean$, $timesdev$, $energymean$, $energydev$, $bandwidths$, and $pOut$ containing the total execution time, the standard deviation of the measurements of execution time, dynamic energy consumption, the standard deviation of the measurements of dynamic energy, the average bandwidths of the $lp2p$ links between the $lp2p$ disjoint pairs of computing devices, and the precision achieved.

Furthermore, *hcl_measure_denergy()* returns 0 if the precision is achieved and 1 otherwise. If statistical confidence criteria are not met, the output argument $pOut$ contains the precision achieved.

Appendix B

libedm: Software Power Meter API for Computations and Data Transfers

B.1 Software Power Meter API

We propose and illustrate the software power meter API included in our software library (**libedm**) [59]. The API functions allow the creation of software power meters based on our proposed linear energy predictive models to estimate the energy consumption of computations and data transfers activities. Furthermore, the power meters API supports only heterogeneous servers that contain a multicore CPU and one or more identical GPUs.

The API function *hcl_create_power_cmeter()* creates a software power meter for computations on a device represented by a tuple, {*dtype*, *devno*}, where *dtype* represents the device type (CPU or GPU) and *devno* signifies the device number within a class of devices given by the device type. The function returns a handle to the power meter in *pmeter* on successful power meter creation.

The argument *peventnames* is an array of size *npe* containing the names of the performance events. The argument, *modelf*, is a user-defined model function, which takes as input an array of performance event values and returns the dynamic energy consumption of the computations. For example, we pass a model function in the power meter creation call that estimates the dynamic energy consumption using our linear energy predictive models based on the performance event values provided as input.

```
typedef double ( *hcl_energy_model_func )(
   int npe, const double* peventarray);
int hcl_create_power_cmeter(
   hcl_device_type dtype, int devno,
   int npe, const char** peventnames,
   const hcl_energy_model_func* modelf,
   hcl_power_meter* pmeter
);
```

The API function *hcl_create_power_dmeter()* creates a software power meter for data transfers between the host CPU and one or more GPUs in the direction given by the *direction* parameter. The direction parameter accepts two values, *FORWARD* and *BACKWARD*, representing the data transfers from the host CPU to the GPUs and GPUs to the host CPU, respectively.

The argument, *modelf*, is a user-defined model function that returns the dynamic energy consumption of the data transfers given the performance event values. On successfully creation of power meter, the output argument, *pmeter*, provides a handle for accessing the power meter. It is important to note that only one power meter can be used for each direction to estimate the dynamic energy consumption of data transfers occurring in that direction within a specific code region.

```
int hcl_create_power_dmeter(
   int direction, int npe,
   const char** peventnames,
   const hcl_energy_model_func* modelf,
   hcl_power_meter* pmeter
);
```

The API function *hcl_start_power_meter()* and *hcl_stop_power_meter()* start and stop the power meter represented by identifier *meter* for energy estimation.

```
int
hcl_start_power_meter(
   const hcl_power_meter* meter);
int
hcl_stop_power_meter(
   const hcl_power_meter* meter,
   double *estenergy);
```

To estimate the energy consumption of a code region containing computations, the API function *hcl_start_power_meter()* should be called before the start of the region, and the API function *hcl_stop_power_meter()* should be called after the end of the region. However, some constraints exist on employing the software power meters for data transfer in a heterogeneous hybrid application. The software power meters for data transfer must be started in the main thread or an OpenMP master thread in a parallel region before initiating the data transfers and stopped in the main thread or an OpenMP master thread after the transfer of results from the GPUs to the host CPU. In the following section, we will illustrate the usage of data transfer power meters in a matrix multiplication application.

The API function *hcl_destroy_power_meter()* below releases the resources associated with the software power meter *meter*.

```
int hcl_destroy_power_meter(
   hcl_power_meter* meter);
```

B.2 Illustration of the Software Power Meter API in a Parallel Matrix Multiplication Application

We illustrate using the software power meter API functions through a parallel matrix multiplication application executing on our Icelake A40 GPU server. Figure B.1 illustrates the hybrid parallel matrix multiplication application (HDGEMM) computing the matrix product ($C+ = A \times B$) of two dense square matrices A and B of size $N \times N$.

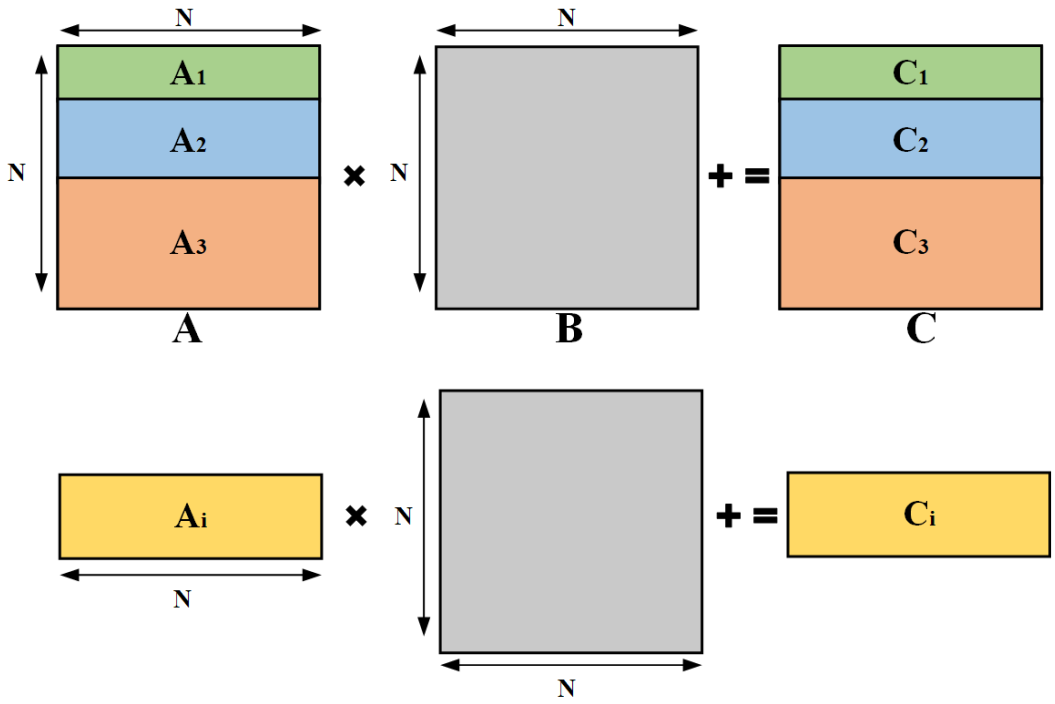


Figure B.1: Matrix partitioning between the software components in hybrid parallel matrix multiplication application (HDGEMM) computing the matrix product ($C+ = A \times B$) of two dense square matrices A and B of size $N \times N$.

The application has three software components: CPU, A40 GPU_1, and A40 GPU_2. All the components share the matrix B. The matrices A and C are horizontally partitioned such that each software component is assigned several contiguous rows of A and C provided as an input parameter to the application. The CPU software component is assigned N_1 rows of A and C. The A40 GPU_1 software component is assigned N_2 rows of A and C. Finally, the A40 GPU_1 software component is assigned the remaining $(N - N_1 - N_2)$ rows of A and C. Each software component *i* ($i \in \{1, 2, 3\}$) computes the matrix product, $C_i = A_i \times B$.

The application comprises three main stages. The first stage consists of data transfers of A_2, B, and C_2 from the host CPU to A40 GPU_1 and A_3, B, and C_3 from the

host CPU to A40 GPU_2. The second stage involves local computations in the software components. The computations in the CPU software component are performed using the Intel MKL DGEMM library routine. The computations in the software components involving the A40 GPUs are performed using the CUBLAS DGEMM library routine. The third stage involves data transfer of the result matrices C_2 and C_3 from A40 GPU_2 and A40 GPU_1 to the host CPU.

Figures B.2 and B.3 illustrate the use of our proposed power meters API functions to estimate the energy consumption of computations and data transfers.

The application-specific routines, *cpudgemm()*, *gpudgemm1()*, and *gpudgemm2()*, contain the device-specific computations. The routines, *gpudt1()* and *gpudt2()*, comprise the data transfers between the host CPU and A40 GPU_1 and the host CPU and A40 GPU_2.

```
#include <edm.h>
int main(int argc, char *argv[]) {
    double *A, *B, *C;
    int ncpuevents = 6; ncpu2gpuevents = 8, ngpu2cpuevents = 10;
    char* cpucevents[] = {
        "CORE_POWER_LVL0_TURBO_LICENSE",
        "TXC_AD_CREDIT_OCCUPANCY,"
        "IOMMU1_PWT_CACHE_LOOKUPS",
        "VN0_NO_CREDITS_SNP",
        "TXL0P_CLK_ACTIVE_DFX",
        "CBOX_CLOCKTICKS" };
    char* cpu2gpuevents[] = {
    "TRACKER_OCCUPANCY_CH0",
    "OFFCORE_REQUESTS_OUTSTANDING_DEMAND_RFO",
    "LLC_LOOKUP_LOC_HOM",
    "UPI_CLOCKTICKS",
    "VN1_NO_CREDITS_NCB",
    "TXN_REQ_OF_CPU_MEM_WRITE_IOMMU1" };
    char* gpu2cpuevents[] = {
    "TXR_HORZ_CYCLES_NE_BL_CRD",
    "OFFCORE_REQUESTS_OUTSTANDING_CYCLES_WITH_DEMAND_RFO",
    "TOR_OCCUPANCY_EVICT",
    "TXL0P_CLK_ACTIVE_CFG_CTL",
    "VN0_NO_CREDITS_SNP",
    "REQ_FROM_PCIE_CMPL_IOMMU_HIT",
    "UNCORE_CLOCKTICKS",
    "M2U_MISC2_TXC_CYCLES_EMPTY_AKC",
    "PKG_RESIDENCY_C0_CYCLES"};
    hcl_power_meter cmeter[3], dmeter[2];
    hcl_create_power_cmeter(HCL_CPU, 0,
       ncpuevents, cpuevents, cpumodelf, &cmeter[0]);
    hcl_create_power_cmeter(HCL_GPU, 0,
       0, NULL, hcl_nvml_modelf, &cmeter[1]);
    hcl_create_power_cmeter(HCL_GPU, 1,
       0, NULL, hcl_nvml_modelf, &cmeter[2]);
    hcl_create_power_dmeter(FORWARD, ncpu2gpuevents, cpu2gpuevents,
       cpu2gpumodelf, &dmeter[0]);
    hcl_create_power_dmeter(BACKWARD, ngpu2cpuevents, gpu2cpuevents,
       gpu2cpumodelf, &dmeter[1]);
}
```

Figure B.2: Illustration of the software power meter API in a parallel hybrid matrix multiplication application executing on a server comprising a Intel multicore CPU and two Nvidia A40 GPUs. Five energy power meters are deployed in this application. Three power meters for computations on the CPU, A40 GPU_1, and A40 GPU_2, and two for data transfers between host CPU to GPUs and GPUs to host CPU.

We first describe the code shown in Figure B.2. Lines 4-28 contain the arrays of shortlisted performance events for computations on the multicore CPU and data transfers from CPU to A40 GPU_1 and A40 GPU_1 to CPU.

Line 29 contain the declarations of the software power meters. Lines 30-35 contain the API invocations creating the software power meters for computations on the CPU and two GPUs using the function call, *hcl_create_power_cmeter*. The power meter, $cmeter[0]$, estimates the energy consumption of computations on the CPU. The power meters, $cmeter[1]$ and $cmeter[2]$, estimate the energy consumption of computations on the A40 GPU_1 and A40 GPU_2 respectively.

Lines 36-39 contain the API function invocations to create the data transfer power meters using the function call, *hcl_create_power_dmeter*. The power meters, $cpu2gpumeter$ and $gpu2cpumeter$, estimate the energy consumption of data transfers from the host CPU to the A40 GPUs and the A40 GPUs to the host CPU, respectively. Note that the power meters for data transfers are different in the two directions: host CPU to GPU and GPU to host CPU.

The model function $cpumodelf$ estimates the energy consumption of computations given the values of the performance events in the array, *cpucevents*. The pre-defined model function hcl_nvml_modelf is based on NVML and estimates the energy of computations on the A40 GPUs. The model functions, $cpu2gpumodelf$ and $gpu2cpumodelf$, estimate the energy consumption of data transfers from the host CPU to the GPUs and GPUs to the host CPU given the values of the performance events in the array, *cpu2gpuevents* and *gpu2cpuevents*, respectively.

Figure B.3 illustrates the rest of the code.

```
#include <edm.h>
int main(int argc, char *argv[]) {
#pragma omp parallel sections num_threads(3)
{
#pragma omp parallel num_threads(3)
{
#pragma omp master {
    hcl_start_power_meter(&dmeter[0]);
    hcl_start_power_meter(&dmeter[1]);
}
#pragma omp barrier
    if (omp_get_thread_num() == 0) {
       hcl_start_power_meter(&cmeter[0]);
       cpudgemm(N1, N, N, A, B, C);
       hcl_stop_power_meter(&cmeter[0], &cpuenergy);
    } else {
       if (omp_get_thread_num() == 1) {
          gpudt1(N2, N, N, &A[N1*N], B, &C[N1*N]);
          hcl_start_power_meter(&cmeter[1]);
          gpudgemm1(N2, N, N);
          hcl_stop_power_meter(&cmeter[1], &gpu1energy);
       } else {
          gpudt2(N1+N2, N, N, &A[(N1+N2)*N], B, &C[(N1+N2)*N]);
          hcl_start_power_meter(&cmeter[2]);
          gpudgemm2(N1+N2, N, N);
          hcl_stop_power_meter(&cmeter[2], &gpu2energy);
       }
    }
#pragma omp barrier
#pragma omp master {
    hcl_stop_power_meter(&dmeter[0], &cpu2gpuenergy);
    hcl_stop_power_meter(&dmeter[1], &gpu2cpuenergy);
}
}
    hcl_destroy_power_meter(&cmeter[0]);
    hcl_destroy_power_meter(&cmeter[1]);
    hcl_destroy_power_meter(&cmeter[2]);
    hcl_destroy_power_meter(&dmeter[0]);
    hcl_destroy_power_meter(&dmeter[1]);
    exit(EXIT_SUCCESS);
}
```

Figure B.3: The matrix multiplication application contains three OpenMP threads invoking the power meter API functions and software components (kernels) in parallel, one CPU component, and two GPU components. Five energy power meters are deployed in this application: three for computations in the CPU, A40 GPU_1, and A40 GPU_2 software components ($cmeter[0]$, $cmeter[1]$, and $cmeter[2]$), two for data transfers between the host CPU and the GPUs and the GPUs and the host CPU ($cpu2gpumeter$, $gpu2cpumeter$).

In Figure B.3, Line 3 comprises the parallel OpenMP region that launches three OpenMP threads to execute the three software components in parallel. In Lines 8-9, we invoke the API function, $hcl\_start\_power\_meter()$, to start data transfer power meters, $dmeter[0]$ and $dmeter[1]$. Lines 13-15 comprise the CPU software component execution. In Lines 13 and 15, we call the API functions, $hcl\_start\_power\_meter()$ and $hcl\_stop\_power\_meter()$, to start and stop the power meter $cmeter[0]$ for estimating the dynamic energy consumption of computations on the CPU. The estimated energy consumption is returned in the variable $cpuenergy$.

Lines 18-21 contain the code for the A40 GPU_1 software component execution. In Lines 19 and 21, we invoke the API functions, $hcl\_start\_power\_meter()$ and $hcl\_stop\_power\_meter()$, to start and stop the power meter $cmeter[1]$ for estimating the dynamic energy consumption of computations on the A40 GPU_1. Similarly, Lines 23-26 contain the A40 GPU_2 software component execution and API functions employing a power meter to estimate the dynamic energy consumption of computations on the A40 GPU_2. The estimated energy consumptions of computations on the GPUs are returned in the variables $gpu1energy$ and $gpu2energy$.

In Lines 31-32, we invoke the API function, $hcl\_stop\_power\_meter()$, to stop the data transfer power meters, $dmeter[0]$ and $dmeter[1]$. The estimated dynamic energy consumption of the data transfers from the CPU to A40 GPUs and the A40 GPUs to the CPU are returned in the variables $cpu2gpuenergy$ and $gpu2cpuenergy$, respectively. Note that the data transfer of results from the A40 GPU_1 to the host CPU happens in *gpudgemm1()* that invokes the CUBLAS DGEMM routine, and from the A40 GPU_2 to the host CPU in *gpudgemm2()*. The barrier in Line 11 ensures that the data transfer power meters are started successfully before initiating the data transfers from the host CPU to the GPUs. Finally, the barrier in Line 29 ensures that the data transfers from the GPUs to the host CPU are complete before stopping the power meters to get the dynamic energy estimate.

Lines 35-40 contain the API function invocations to destroy the power meters involved in estimating the energy of computations and data transfers in the application.

Appendix C

Concurrent and Orthogonal Software Power Meters on Heterogeneous Hybrid Servers

C.1 Likwid Performance Monitoring Counter Groups for the Multicore CPUs

Table C.1: Performance monitoring counter groups in Intel Haswell multicore CPU of the Haswell K40c GPU server as shown in Table 3.1.

Performance Counter Groups	Description
BBOX	Measurements of the Home Agent (HA) in the uncore
CBOX	Last level cache (LLC) coherency engine in the uncore
MBOX	Integrated memory controllers (iMC) in the uncore
PBOX	Measurements of the Ring-to-PCIe (R2PCIe) interface in the uncore
PMC	Core-local general purpose counters
PWR	Measurements of the current energy consumption through the RAPL interface
QBOX	Measurements of the QPI Link layer (QPI) in the uncore
SBOX	Socket internal traffic through ring-based networks
UBOX	System configuration controller, interrupt traffic, and system lock master
WBOX	Power control unit (PCU) in the uncore

Table C.2: Performance monitoring counter groups in Intel Skylake multicore CPU of the Skylake P100 GPU server as shown in Table 3.2.

Performance Counter Groups	Description
PMC	Core-local general purpose counters
PWR	Measurements of the current energy consumption through the RAPL interface
CBOX	Last level cache (LLC) coherency engine in the uncore
UBOX	System configuration controller, interrupt traffic, and system lock master
WBOX	Power control unit (PCU) in the uncore
MBOX	Integrated memory controllers (iMC) in the uncore
SBOX	Socket internal traffic through ring-based networks
SBOX	Socket internal traffic through ring-based networks
RBOX	M3UPI interface between the mesh and the Intel® UPI Link Layer
IBOX	IIO stacks for managing traffic between the PCIe domain and the Mesh domain

Table C.3: Performance monitoring counter groups in Intel IceLake multicore CPU of the Icelake A40 GPU server as shown in Table 3.3.

Performance Counter Groups	Description
CBOX	Last level cache (LLC) coherency engine in the uncore
IBOX	Responsible for maintaining coherency for Integrated Input/Output controller (IIO) traffic
MBOX	Integrated memory controllers (iMC) in the uncore
M2M	Mesh2Mem (M2M) which connects the cores with the Uncore devices. The interface between the Mesh and the Memory Controllers.
PMC	Core-local general purpose counters
PWR	Measurements of the current energy consumption through the RAPL interface
QBOX	Measurements of the QPI Link layer (QPI) in the uncore
SBOX	Socket internal traffic through ring-based networks
TCBOX	Integrated Input/Output controller (IIO) counters
UBOX	System configuration controller, interrupt traffic, and system lock master
WBOX	Power control unit (PCU) in the uncore

C.2 List of Benchmarks Employed for Selection of Performance Event Sets

Table C.4 shows the list of applications employed for training and testing the energy predictive models for computations on the multicore CPU. Table C.5 shows the list of applications employed for training and testing the energy predictive models for data transfers between CPU and GPU. Table C.6 shows the list of parallel hybrid applications employed for selection of performance event sets.

Table C.4: List of benchmarks employed for selection of performance event sets, training and testing the energy predictive models for computations on multicore CPUs.

Benchmark	**Description**
MKL FFT	Intel optimized 2-dimensional fast Fourier transform
MKL DGEMM	Intel optimized dense matrix multiplication of two square matrices
HPCG	Intel optimized High Performance Conjugate Gradient. 3-dimensional regular 27-point discretization of an elliptic partial differential equation
NPB IS	Integer Sort, Kernel for random memory access that sort small integers using the bucket sort technique
NPB LU	Lower-Upper Gauss-Seidel solver
NPB EP	Embarrassingly Parallel random number generator
NPB BT	Solve synthetic system of nonlinear partial differential equations using Block Tri-diagonal solver
NPB MG	Approximate 3-dimensional discrete Poisson equation using the V-cycle Multi Grid on a sequence of meshes
NPB FT	A 3D fast Fourier Transform partial differential equation benchmark
NPB DC	Arithmetic Data Cube, a data intensive grid benchmark representing data mining operations
NPB UA	Unstructured Adaptive mesh solving heat equation with convection and diffusion from moving ball
NPB CG	Solving an unstructured sparse linear system using Conjugate Gradient method
NPB SP	Solve synthetic system of nonlinear partial differential equations using Scalar Penta-diagonal solver
NPB DT	A graph benchmark evaluating communication throughput (Data Traffic)
LULESH	Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics
HPGMG	High Performance Geometric Multigrid
miniFE	Unstructured implicit finite element codes
miniMD	Parallel molecular dynamics (MD) code
Naive MM	Naive Matrix-matrix multiplication
Naive MV	Naive Matrix-vector multiplication

Table C.5: List of benchmarks employed for selection of performance event sets, training and testing the energy predictive models for data transfers between CPU and GPUs and computations on the GPU.

Benchmark	Open-Source Package	**Description**
Needleman-Wunsch	Rodinia [65]	Nonlinear global optimization method for DNA sequence alignments
SRAD	Rodinia	Speckle Reducing Anisotropic Diffusion
MRI-Q	Parboil [66]	Computes a matrix Q used in a 3D magnetic resonance image reconstruction algorithms in non-Cartesian space
SPMV	Parboil	Computes the product of a sparse matrix with a dense vector
STENCIL	Parboil	An iterative Jacobi stencil operation on a regular 3-D grid
TPACF	Parboil	Statistical analysis of the spatial distribution of observed astronomical bodies
LBM	Parboil	A fluid dynamics simulation of an enclosed, lid-driven cavity, using the Lattice-Boltzmann Method
MD	SHOC [67]	Molecular Dynamics particle simulation
Stencil2D	SHOC	2D stencil computation
S3D	SHOC	Combustion chemistry kernel
bicg	PolyBench/GPU [68]	Bi-conjugate gradient
cholesky	PolyBench/GPU	Cholesky Decomposition
gramschmidt	PolyBench/GPU	Gram-Schmidt decomposition
jacobi-2D	PolyBench/GPU	2D Jacobi stencil computation
syr2k	PolyBench/GPU	Symmetric rank-2k operations
BlackScholes	CUDA Samples [69]	Black-Scholes Option Pricing
nbody	CUDA Samples	CUDA N-Body Simulation
conjugateGradientMultiBlockCG	CUDA Samples	conjugateGradient using MultiBlock Cooperative Groups
cuSolverDn_LinearSolver	CUDA Samples	cuSolverDN's LU, QR and Cholesky factorization
matrixMulCUBLAS	CUDA Samples	Matrix Multiplication (CUBLAS)

Table C.6: List of parallel hybrid applications employed for selection of performance event sets.

Benchmark	Open-Source Package	**Description**
hdgeadd	OpenH [70]	Hybrid matrix addition
hdgemm	OpenH	Hybrid matrix multiplication
hfft2d	OpenH	Hybrid 2D fast Fourier transform
hfft3d	OpenH	Hybrid 3D fast Fourier transform

C.3 Software Power Meter API

We propose a software power meter API that enables the runtime integration of software power meters. These power meters implement our proposed linear energy predictive models to estimate the energy consumption of both computation and communication activities.

The API function *hcl_create_power_cmeter()* creates a software power meter for computations on a device represented by a 2-tuple, {*dtype*, *devno*}, where *dtype* represents the device type (CPU or GPU) and *devno* signifies the device number within a class of devices given by the device type. Upon successful creation of the power meter, the function returns a handle to the power meter in the variable, *pmeter*. The argument *peventnames* is an array of size *npe*, which contains the names of the performance events. The argument, *modelf*, is a user-defined model function that takes as input an array of performance event values and returns the dynamic energy consumption of the computations.

```
typedef double ( *hcl_energy_model_func )(
   int npe, const double* peventarray);
int hcl_create_power_cmeter(
   hcl_device_type dtype, int devno,
   int npe, const char** peventnames,
   const hcl_energy_model_func* modelf,
   hcl_power_meter* pmeter
);
```

The API function *hcl_create_power_dmeter()* is used to create a software power meter for monitoring data transfer between CPU and GPU. The direction of the data transfer is specified by the *direction* parameter, which accepts two values: *FORWARD* and *BACKWARD*. *FORWARD* indicates data transfers from the host CPU to the GPUs, while *BACKWARD* indicates transfers from the GPUs back to the host CPU. Additionally, the *modelf* argument is a user-defined function that calculates the dynamic energy consumption of the data transfers based on performance event values. Upon successful creation of the power meter, the output argument *pmeter* provides a handle for accessing the power meter. It is important to note that only one power meter can be used for each direction to estimate the dynamic energy consumption of data transfers occurring in that direction within a specific code region.

```
int hcl_create_power_dmeter(
   int direction, int npe,
   const char** peventnames,
   const hcl_energy_model_func* modelf,
   hcl_power_meter* pmeter
);
```

The API function *hcl_start_power_meters()* starts a set of power meters of size *nmeters* represented by an array *meterids* for energy estimation.

```
int hcl_start_power_meters(int nmeters,
   hcl_power_meter* meterids);
```

Conversely, the API function *hcl_stop_power_meters()* halts the operation of the meters represented by *meterids*. Additionally, the output array *energyestimates* contains the dynamic energy estimates where the array element $energyestimates[i]$ contains the estimate for meter i. Two more arrays, $times$ and $avgpowerestimates$, contain the times and average dynamic energy estimates, respectively.

```
int hcl_stop_power_meters(int nmeters,
   hcl_power_meter* meterids,
   double *times, double *avgpowerestimates,
   double *energyestimates);
```

The API functions, *hcl_start_power_meters* and *hcl_stop_power_meters*, provide the flexibility to selectively start and stop one or more power meters based on the given set of power meter IDs. To estimate the energy consumption of a specific code region, the API function *hcl_start_power_meters()* should be called at the beginning of the region, and the API function *hcl_stop_power_meters()* should be called at the end. These API functions require an array of meter IDs that represent the power meters associated with the activities involved in executing the code region.

The API function *hcl_free_power_meters()* below releases the resources associated with the set of meters given by *meterids*.

```
int hcl_free_power_meters(int nmeters,
   hcl_power_meter* meterids);
```

All the API functions mentioned above are not thread-safe, which means they must be called by a single CPU thread. However, the functions for creating power meters (*hcl_create_power_cmeter* and *hcl_create_power_dmeter*) and for destroying them (*hcl_free_power_meters*) can be called by different CPU threads. Additionally, the start and stop API functions, *hcl_start_power_meters* and *hcl_stop_power_meters*, may also be called by different CPU threads.

C.3.1 Instrumentation of a Parallel Hybrid Matrix Multiplication Application Using Software Power Meters

We demonstrate the functionality of our software power meter API through a parallel hybrid matrix multiplication application running on our Icelake server (see Table 3.3).

There are several methods to instrument parallel hybrid applications with our software power meters. In the most general case, we can measure specific activities during designated phases of the application. In this illustration, we present an approach that selectively starts and stops each power meter for individual activities, such as computation or data transfer. This method allows us to identify the sections of code or activities that consume the most energy. These sections then become targets for energy optimization, although exploring that aspect is beyond the scope of this work.

Figure B.1 illustrates the parallel hybrid matrix multiplication application (HDGEMM), which computes the matrix product C += $A \times B$ of two dense square matrices, A and B, each of size $N \times N$.

The application consists of three software components: the CPU, A40 GPU_1, and A40 GPU_2. All components share the matrix B. The matrices A and C are horizontally partitioned, with each software component assigned several contiguous rows of A and C, as specified by input parameters to the application. The CPU is responsible for N_1 rows of A and C. The A40 GPU_1 component is assigned N_2 rows of A and C, while the A40 GPU_2 component takes on the remaining rows, specifically $N - N_1 - N_2$ of A and C. Each software component i (where $i \in \{1, 2, 3\}$) computes the matrix product C_i += $A_i \times B$.

The application consists of three main stages. The first stage involves transferring data: A_2, B, and C_2 from the host CPU to A40 GPU_1, and A_3, B, and C_3 from the host CPU to A40 GPU_2. The second stage is dedicated to performing local computations

```
#include <edm.h>
int main(int argc, char *argv[]) {
    double *A, *B, *C;
    enum metertype { CPU = 0, GPU1, GPU2, H2D1, D2H1, H2D2, D2H2 };
    int NS = 7, nevents= 6;
    char* cpuevents[] = {
       "CBOX:READ_NO_CREDITS_ANY",
       "PMC:FP_ARITH_INST_RETIRED_SCALAR_DOUBLE",
       "M2M:DISTRESS_ASSERTED_VERT",
       "TCBOX:REQ_FROM_PCIE_CMPL_REQ_OWN",
       "UBOX:M2U_MISC1_RXC_CYCLES_NE_CBO_NCB",
       "WBOX:FREQ_TRANS_CYCLES" };
    char* h2d1events[] = {
       "CBOX:DIRECT_GO_OPC_EXTCMP",
       "PMC:FP_ARITH_INST_RETIRED_SCALAR_SINGLE",
       "M2M:TXR_HORZ_ADS_USED_AD_UNCRD",
       "PBOX:IIO_CREDITS_ACQUIRED_DRS_ANY",
       "TCBOX:INBOUND_ARB_REQ_WR",
       "IBOX:CACHE_TOTAL_OCCUPANCY_ANY"
    };
    char* d2h1events[] = {
       "CBOX:RXC_ISMQ0_REJECT_BL_RSP_VN0",
       "PMC:MEM_LOAD_RETIRED_L3_HIT",
       "M2M:VERT_RING_AK_IN_USE_DN_EVEN",
       "PBOX:TXC_INSERTS_AD_ANY",
       "TCBOX:NUM_REQ_OF_CPU_BY_TGT_MEM",
       "IBOX:MISC0_FAST_REQ"
    };
    // Similar event arrays for the
    // data transfers between CPU and GPU2
    hcl_power_meter meterids[NS];
    hcl_create_power_cmeter(HCL_CPU, 0,
       nevents, cpuevents, cpumodelf, &meterids[CPU]);
    hcl_create_power_cmeter(HCL_GPU, 0,
       0, NULL, hcl_nvml_modelf, &meterids[GPU1]);
    hcl_create_power_cmeter(HCL_GPU, 1,
       0, NULL, hcl_nvml_modelf, &meterids[GPU2]);
    hcl_create_power_dmeter(FORWARD, nevents, h2d1events,
       h2d1modelf, &meterids[H2D1]);
    hcl_create_power_dmeter(BACKWARD, nevents, d2h1events,
       d2h1modelf, &meterids[D2H1]);
    hcl_create_power_dmeter(FORWARD, nevents, d2h2events,
       h2d2modelf, &meterids[H2D2]);
    hcl_create_power_dmeter(BACKWARD, nevents, d2h2events,
       d2h2modelf, &meterids[D2H2]);
}
```

Figure C.1: Illustration of the software power meter API in a parallel hybrid matrix multiplication application executing on a server comprising a Intel multicore CPU and two Nvidia A40 GPUs. Seven energy power meters are deployed in this application. Three power meters for computations on the CPU, A40 GPU_1, and A40 GPU_2 ($meterids[CPU]$, $meterids[GPU1]$, and $meterids[GPU2]$). Two for data transfers between the host CPU and the A40 GPU_1 and the A40 GPU_1 and the host CPU. Finally, two for data transfers between the host CPU and the A40 GPU_2 and the A40 GPU_2 and the host CPU.

within the software components. Finally, the third stage entails transferring the resulting matrices, C_2 and C_3, from A40 GPU_2 and A40 GPU_1 back to the host CPU.

Figures C.1 and C.2 illustrate the use of our proposed software power meter API functions to estimate the energy consumption associated with computations and data transfers.

The application-specific routines - *cpudgemm()*, *gpudgemm1()*, and *gpudgemm2()* - are responsible for device-specific computations. Additionally, the routines *gpudt1()* and *gpudt2()* handle data transfers between the host CPU and A40 GPU_1, as well as between the host CPU and A40 GPU_2.

We first describe the code shown in Figure C.1. Lines 4-28 contain arrays that list the relevant performance events for computations on the multicore CPU and for data transfers between the host CPU and the A40 GPU devices.

In line 31, the array declaration containing the power meter identifiers is defined. Lines 32-37 include API function calls that create software power meters for computations on the CPU and two GPUs using the function call, *hcl_create_power_cmeter*. The power meter identified as $meterids[CPU]$ measures the energy consumption of computations on the CPU. Meanwhile, the power meters $meterids[GPU1]$ and $meterids[GPU2]$ estimate the energy consumption for computations on A40 GPU_1 and A40 GPU_2, respectively.

Lines 38-45 detail the API function calls that create data transfer power meters using the function call, *hcl_create_power_dmeter*. The power meters $meterids[H2D1]$ and $meterids[D2H1]$ estimate the energy consumption for data transfers between the host CPU and A40 GPU_1. Similarly, $meterids[H2D2]$ and $meterids[D2H2]$ estimate the energy consumption for data transfers between the host CPU and A40 GPU_2. It's important to note that the power meters for data transfers are distinct for the two directions: from the host CPU to the GPU and from the GPU back to the host CPU.

The model function $cpumodelf$ estimates the energy consumption of computations based on the performance events recorded in the array, *cpuevents*. The pre-defined function hcl_nvml_modelf utilizes Nvidia's NVML interface to estimate the energy consumed during computations on the A40 GPU. Additionally, the functions $h2d1modelf$ and $d2h1modelf$ estimate the energy consumption associated with data transfers between the host CPU and A40 GPU_1, using performance events captured in the arrays

```
#include <edm.h>
int main(int argc, char *argv[]) {
   double times[NS], avgpowers[NS], energyestimates[NS];
#pragma omp parallel sections num_threads(3)
{
#pragma omp section
   {
      hcl_start_power_meters(1, &meterids[CPU]);
      cpudgemm(N1, N, N, A, B, C);
      hcl_stop_power_meters(1, &meterids[CPU], &energyestimates[CPU]);
   }
#pragma omp section
   {
      hcl_start_power_meters(1, &meterids[H2D1]);
      gpudt1(N2, N, N, &A[N1*N], B, &C[N1*N]);
      hcl_stop_power_meters(1, &meterids[H2D1], &times[H2D1], &avgpowers[H2D1], &
      energyestimates[H2D1]);
      hcl_start_power_meters(1, &meterids[GPU1]);
      hcl_start_power_meters(1, &meterids[D2H1]);
      gpudgemm1(N2, N, N);
      hcl_stop_power_meters(1, &meterids[D2H1], &times[D2H1], &avgpowers[D2H1], &
      energyestimates[D2H1]);
      hcl_stop_power_meters(1, &meterids[GPU1], &times[GPU1], &avgpowers[GPU1], &
      energyestimates[GPU1]);
   }
#pragma omp section
   {
      hcl_start_power_meters(1, &meterids[H2D2]);
      gpudt2(N1+N2, N, N, &A[(N1+N2)*N], B, &C[(N1+N2)*N]);
      hcl_stop_power_meters(1, &meterids[H2D2], &times[H2D2], &avgpowers[H2D2], &
      energyestimates[H2D2]);
      hcl_start_power_meters(1, &meterids[GPU2]);
      hcl_start_power_meters(1, &meterids[D2H2]);
      gpudgemm2(N1+N2, N, N);
      hcl_stop_power_meters(1, &meterids[D2H2], &times[D2H2], &avgpowers[D2H2], &
      energyestimates[D2H2]);
      hcl_stop_power_meters(1, &meterids[GPU2], &times[GPU2], &avgpowers[GPU2], &
      energyestimates[GPU2]);
   }
}
   hcl_free_power_meters(NS, meterids);
   exit(EXIT_SUCCESS);
}
```

Figure C.2: The matrix multiplication application contains three OpenMP threads that execute the software components (kernels) in parallel. The software power meters are selectively started and stopped for individual computation and communication activities.

h2d1events and *d2h1events*, respectively. There are also two corresponding functions for data transfers between the host CPU and A40 GPU_2.

Figure C.2 illustrates the remainder of the code. Lines 3-4 show the declaration of the arrays that will store all the times, average power and dynamic energy estimates. Line 4 initiates a parallel OpenMP region, which consists of three threads executing the software components concurrently. In this line, we start the energy power meter for CPU computations using the API function call, *hcl_start_power_meters*. Line 9 includes the execution of the CPU software component. Finally, Line 10 stops the power meter and retrieves the dynamic energy estimate of the CPU computations, storing it in the array element *energyestimates[CPU]*.

In Line 14, we start the power meter for the data transfer from the host CPU to A40 GPU_1. Line 15 triggers the data transfer operations. In Line 16, we stop the power meter and obtain the dynamic energy estimate of the CPU computations for the array element, *energyestimates[H2D1]*. Line 19 includes computations performed on A40 GPU_1 and transfers the results back to the host CPU. It is important to note that the routine *gpudgemm1()* calls a CUDA library function that combines both the computations and the data transfer in a single invocation. Consequently, we start two power meters in Lines 17-18: one for the computations and the other for the data transfer from A40 GPU_1 to the host CPU. In Lines 20-21, we stop the power meters and retrieve the dynamic energy estimates in *energyestimates[GPU1]* and *energyestimates[D2H1]*.

Lines 23-33 present the code for the execution of the A40 GPU_2 software component. In Line 35, we release all the resources associated with the power meters using the API function *hcl_free_power_meters*. The total dynamic energy consumption of the program is calculated as the sum of the dynamic energy estimates stored in the array *energyestimates*.


UCD
DUBLIN