Evaluating Energy Efficiency of HPC Benchmarks

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Computer Science

By:

Karl Brzoska

Project Advisors:

Shubbhi Taneja
Joseph Manzano
Katarzyna Swirydowicz

Date: April 2024

Abstract

As we enter the exascale computing era, the focus on energy efficiency and performance for High-Performance Computing (HPC) systems, particularly when handling benchmarks such as the High Performance Conjugate Gradients (HPCG), is paramount. This project aims to assess the energy efficiency and performance of HPC systems by profiling on benchmarks like HPCG and scientific workflows. The goal is to investigate the impact of various system configurations and execution parameters on the power efficiency and performance, enabling the optimization of HPC systems for energy-efficient execution of computationally intensive tasks, thereby supporting sustainable and effective scientific computing practices.

Acknowledgements

I would like to thank Dr. Shubbhi Taneja for providing support and guidance throughout this project. I would also like to thank Drs. Joseph Manzano and Kasia Swirydowicz from the Pacific Northwest National Laboratory for their invaluable advice. I would also like to thank Adhiraj Budukh for his assitance throughout the project. In addition, this work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Specifically, it used the Bridges-2 system, which is supported by NSF award number ACI-1928147, at the Pittsburgh Supercomputing Center (PSC), as well as the Rockfish system, which is supported by NSF award number ACI-1920103, at Johns Hopkins University (JHU). This research was also performed using computational resources supported by the Academic and Research Computing group at Worcester Polytechnic Institute.

Contents

List of Tables

List of Figures

1 Introduction

In the dynamic landscape of High-Performance Computing (HPC), the march towards exascale computing has heralded significant technological advancements. The last decade has witnessed a monumental surge in the capabilities of the world's leading HPC systems, with a performance increase of 42-fold, escalating from 10,510.0 TFlops in November 2011 to an impressive 442,010.0 TFlops by November 2021 [3]. This exponential growth in computational power has, however, been accompanied by a corresponding increase in energy consumption, spotlighting the paramount challenge of boosting energy efficiency. While the Fugaku supercomputer achieved a significant milestone in reaching exascale computing in June 2020, it has since been surpassed by other systems in terms of computational power and efficiency rankings. The Frontier supercomputer, housed at the Oak Ridge National Laboratory in Tennessee, USA, leads as the world's most powerful supercomputer, with an HPL benchmark score of 1.194 exaflops. It is recognized as the only exascale system on the TOP500 list, showcasing advanced computational capabilities and efficiency. The Aurora system, still in completion, is expected to reach over two exaflops, promising further advancements in high-performance computing. Meanwhile, Fugaku now ranks fourth, with an HPL score of 443.01 petaflops, indicating a shift in the landscape of supercomputing power and highlighting the ongoing evolution and competition in the field [2].

Graphics Processing Units (GPUs) have been instrumental in the progress of HPC systems. Transitioning from auxiliary components to being at the core of computational powerhouses, GPUs now power seven of the top ten supercomputers globally as of 2021 [1]. Their critical role extends beyond sheer computational strength to being central in the quest for energy efficiency, with nine out of the ten most energy-efficient supercomputers relying on GPU technology. The introduction of Nvidia's ninth-generation Ampere GPU architecture in May 2020 marked a significant milestone, promising major improvements in performance and energy consumption over its Volta predecessor, thus setting new standards for energy-efficient computing [3],[4]. There has been research to even optimize.

This project aims to assess the energy efficiency and performance of High-Performance Computing (HPC) systems through the execution of HPCG benchmark [3]. This marks a pivot from our previous focus on Machine Learning (ML) applications, as documented in previous reports, towards a comprehensive analysis of scientific workflows. There is a huge use of these hpc systems in science. [2] By examining energy consumption and computational efficiency during the execution of HPCG benchmarks, the project seeks to uncover performance bottlenecks and devise strategies for enhancing both sustainability and computational power in HPC systems. The HPCG benchmark runs sparse algorithms. [6] There is work on improving these

systems. [13] This endeavor is crucial for advancing toward exascale computing with a sustained commitment to energy efficiency.

1.1 Project Objectives

This study aims to explore and enhance the energy efficiency of High-Performance Conjugate Gradients (HPCG) within high-performance computing (HPC). Under the guidance of Professor Shubbhi Taneja from the Computer Science Department at Worcester Polytechnic Institute (WPI), and in collaboration with experts from PNNL, this research seeks to identify, evaluate, and investigating workflows to identify performance characteristics and any bottlenecks that if correctly identified could reveal opportunities to save energy. The objective encompasses a detailed examination of the installation, configuration, and performance testing processess. Through this endeavor, the report will contribute to the broader discourse on sustainable computing, offering insights into the challenges and opportunities for advancing energy efficiency in the context of evolving computational demands.

1.2 Objective

We plan to execute the High-Performance Conjugate Gradient (HPCG) benchmark across high-performance computing systems to assess and compare their energy consumption and efficiency. This comprehensive evaluation will contribute to a deeper understanding of how HPCG operations utilize system resources and their impact on energy usage.

1.3 Background

This study embarks on a comprehensive exploration of high-performance computing (HPC) capabilities by conducting rigorous experiments using the High-Performance Conjugate Gradients (HPCG) benchmark, on two distinguished supercomputers: Rockfish at Johns Hopkins University (JHU) and Bridges at the Pittsburgh Supercomputing Center (PSC). These experiments are designed to delve into the computational efficiency and energy consumption patterns of these advanced HPC systems, providing invaluable insights into their performance characteristics under the load of sophisticated scientific computations. The following paragraphs detail the architectural nuances and configurations of the Rockfish and Bridges systems, setting the stage for understanding how their design influences the outcomes of the HPCG experiments. In the realm of high-performance computing (HPC), the architectural design and configuration of systems like Rockfish at Johns Hopkins University (JHU) and Bridges at the Pittsburgh Supercomputing Center

(PSC) stand as pivotal elements in advancing computational research. These systems are tailored to meet the diverse and demanding requirements of contemporary scientific investigations, encapsulating a blend of cutting-edge technology and strategic design principles to optimize performance, scalability, and energy efficiency.

1.4 Rockfish

The Rockfish HPC system, situated within the academic and research ecosystem of JHU, exemplifies a modern approach to supporting a wide array of computational and data-intensive research endeavors. While specific architectural details of Rockfish are proprietary or not widely disseminated in public domains, it is reasonable to infer from prevailing trends in HPC architectures that the system incorporates a robust network of compute nodes. These nodes are likely powered by multi-core processors that facilitate parallel computations, a cornerstone for accelerating scientific discovery. Accompanying these compute resources, high-capacity, fast-access storage systems are essential for managing the voluminous datasets characteristic of modern research. The interconnectivity among compute nodes is presumably ensured by a high-speed, low-latency network, such as InfiniBand, which is renowned for its high throughput and minimal latency, thereby enhancing the system's ability to perform large-scale, distributed computations efficiently. [4]

1.5 Bridges

Parallel to Rockfish, the Bridges system at PSC, including its more recent iteration, Bridges-2, has been explicitly designed to cater not only to traditional HPC applications but also to emergent computational paradigms such as artificial intelligence (AI), machine learning (ML), and big data analytics. Bridges distinguishes itself through a heterogeneous computing architecture that integrates a variety of compute nodes, including those with substantial memory capacities (reaching several terabytes per node) and GPU-accelerated nodes designed to expedite AI and ML tasks. The inclusion of advanced GPU technologies, such as NVIDIA's Tesla and A100 series, underscores the system's readiness to tackle compute-intensive applications, leveraging the parallel processing prowess of GPUs. The storage architecture within Bridges adopts a tiered approach, featuring both a high-performance parallel file system for active data access and secondary storage solutions to accommodate larger datasets and archival needs. The underlying network infrastructure, likely based on InfiniBand technology, supports the system's heterogeneous computational demands by facilitating swift data movement and inter-node communication. [11]

The software ecosystem across both Rockfish and Bridges encompasses a comprehensive suite of

tools, libraries, and frameworks, meticulously curated to support a broad spectrum of research activities. This includes operating systems optimized for HPC, job scheduling systems to efficiently manage computational workloads, and a plethora of scientific applications and libraries that empower researchers to push the boundaries of their respective fields.

In summary, the architectural blueprints of Rockfish and Bridges reflect a sophisticated understanding of the requirements for modern scientific computation. Through strategic hardware selection and integration, coupled with a robust software infrastructure, these systems provide the computational backbone necessary for advancing research across numerous scientific disciplines. As HPC continues to evolve, the adaptability and innovation embedded within the architectures of Rockfish and Bridges will remain critical in addressing the ever-expanding complexity of research challenges.

1.6 Motivation for using HPCG

In this study, the High Performance Conjugate Gradient (HPCG) benchmark serves a pivotal role in advancing our understanding of energy consumption and efficiency across high-performance computing (HPC) systems. As a metric designed to parallel real-world computational loads more closely than its predecessor, HPL, HPCG incorporates memory-bound operations that are inherently more reflective of the diverse and demanding nature of scientific computation today. By executing the HPCG benchmark on state-of-the-art HPC systems, we aim to gather empirical data on energy usage under load conditions that simulate actual application behavior, rather than peak theoretical performance. This provides a dual benefit: it allows for a more nuanced analysis of the energy profiles of supercomputers and helps in identifying opportunities for energy optimization. Crucially, HPCG's emphasis on data movement and problem-solving efficiency presents a more comprehensive framework for assessing how energy is consumed in the pursuit of solving complex problems. Through this benchmark, we can evaluate the trade-offs between computational performance and energy expenditure, enabling the development of more energy-efficient computing strategies and contributing to the broader goal of sustainable advancement in HPC technologies. [12]

[1]

Table 1: Comparison of NVIDIA A100 and V100 GPUs

Specification	A100	V100
Architecture	Ampere	Volta
CUDA Cores	6,912	5,120
Tensor Cores	432 (3rd Gen)	640 (2nd Gen)
Memory Capacity	40 GB or 80 GB HBM2e	16 GB or 32 GB HBM2
Memory Bandwidth	1.6 TB/s	900 GB/s
Peak FP32 Performance	19.5 TFLOPS	15.7 TFLOPS
Peak Tensor Performance (FP16)	Up to 312 TFLOPS	Up to 125 TFLOPS
Interconnect Bandwidth (NVLink)	600 GB/s	300 GB/s
Power Consumption	250W - 400W	250W - 300W

2 Related Work

Several previous studies have focused on similar topics to this project, including the performance and energy efficiency of certain benchmarks.

2.1 Performance Modeling of the HPC Benchmark

The evolution of benchmarks for assessing high-performance computing (HPC) systems has been pivotal in reflecting the computational demands of real-world applications. Traditionally, the High Performance LINPACK (HPL) benchmark has been the standard for ranking HPC systems. However, its relevance has diminished as it primarily measures compute-bound operations, which do not accurately represent the memory-bound nature of many contemporary scientific applications. This gap led to the development of the High Performance Conjugate Gradients (HPCG) benchmark, designed to offer a more comprehensive evaluation by focusing on operations that are more indicative of real-world applications' performance.

Heroux et al. (2013) introduced the HPCG benchmark, highlighting its design aimed at better mirroring the computational and data access patterns seen in current scientific computing tasks than what HPL offers [3]. The authors emphasized that HPCG seeks to provide a more realistic measure of a system's performance, focusing on memory-bound rather than compute-bound operations, aligning with the performance characteristics of many modern HPC applications [3].

Further extending the utility of HPCG, Marjanović et al. (2015) presented an in-depth analysis and performance modeling of the benchmark [9]. Their work underscored the predictive capability of HPCG

in evaluating system performance based on critical hardware characteristics: the effective bandwidth between the main memory and the CPU, and the highest occurring network latency between compute units. Their modeling demonstrates high accuracy in predicting HPCG performance, underscoring the benchmark's relevance in assessing future exascale systems [9].

The collective discourse surrounding HPCG accentuates its significance in steering the development towards more energy-efficient HPC systems. By emphasizing memory-bound operations and integrating factors such as network latency and memory bandwidth into its performance evaluation, HPCG advocates for hardware and software optimizations geared toward energy efficiency. This paradigm shift is crucial for achieving sustainable exascale computing, where energy consumption emerges as a formidable challenge to the scalability of HPC systems.

The performance model, predicated on straightforward hardware metrics, suggests that advancements in memory bandwidth optimization and reductions in network latency are key to enhancing HPCG scores [9]. Such improvements, by extension, could significantly impact the energy efficiency and performance of HPC systems in executing memory-intensive scientific applications. This insight forms a critical foundation for my MQP, aiming to contribute to the optimization efforts for better energy efficiency in HPC systems, particularly through enhancements targeting the HPCG benchmark.

The paper "Modeling CPU Energy Consumption of HPC Applications on the IBM POWER7" by Philipp Gschwandtner et al. addresses the challenge of optimizing energy consumption in High-Performance Computing (HPC) applications, particularly when hardware support for measuring power and energy is limited. The study focuses on developing in-band energy consumption models for the IBM POWER7 processor using hardware counters and linear regression. The models aim to predict energy consumption accurately without needing detailed, hardware-specific micro-benchmarks for training. Instead, they utilize high-level benchmarks, considering different instruction mixes influenced by compilers (GCC and IBM XL) and multithreading scenarios. [7]

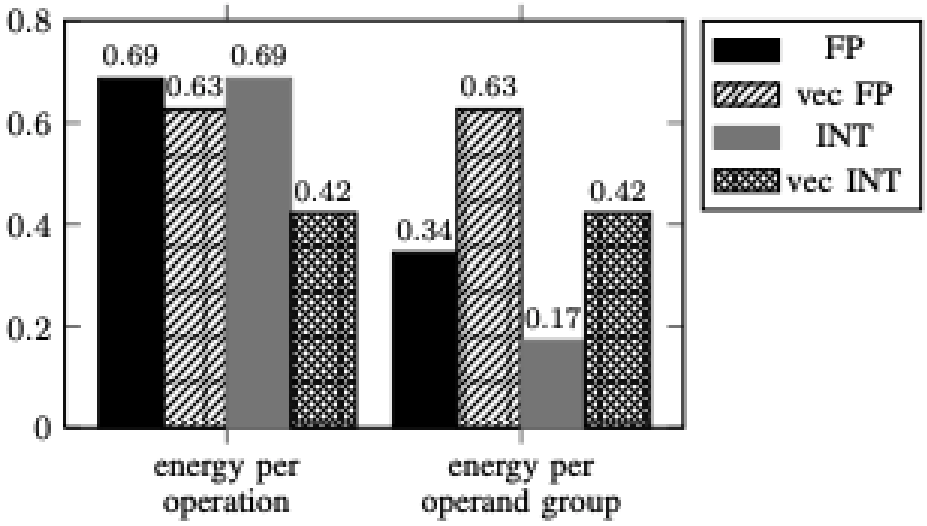

Figure 1: Energy Consumption on V100, 1 processes, 256x256x256

The research findings demonstrate that linear regression can model energy consumption with high accuracy (mean errors of approximately 1 percent and maximum errors of 5.3 percent for GCC compiled applications). The paper also explores the effects of different compilers and parallelism on energy modeling, revealing that the IBM XL compiler's complex usage of hardware makes energy consumption modeling more challenging compared to GCC. The study contributes to energy-efficient HPC by providing simple yet accurate models for CPU energy consumption, emphasizing the need for in-band measurements and the potential for auto-tuning and performance analysis in energy optimization efforts.

3 Benchmarking and Experimental Methodology

3.1 Experiment Setup

Accessing High-Performance Computing Systems The preliminary phase of setting up the High Performance Conjugate Gradients (HPCG) benchmark involved securing access to two renowned high-performance computing (HPC) systems: Rockfish at Johns Hopkins University (JHU) and Bridges at the Pittsburgh Supercomputing Center (PSC). Access was granted through institutional affiliations and collaborations, providing an opportunity to deploy and assess the HPCG benchmark on these advanced computational platforms. Upon securing the necessary permissions, the HPCG benchmark software was acquired. The source code for HPCG was downloaded from the official repository, ensuring that the most recent and

stable version of the benchmark was obtained for an accurate evaluation. It was then compiled.

3.2 Initiating HPCG Runs

With the HPCG benchmark successfully compiled and the systems configured, the execution phase commenced. The runs were initiated using the Message Passing Interface (MPI) to orchestrate parallel computations across the multiple nodes of the Rockfish and Bridges systems. The benchmark was launched using the command mpirun, which is the standard runner for MPI-enabled applications. The specific command used was:

mpirun -np ¡number$_o f_p rocesses >$ $./bin/xhpcg - -nx =< grid_s ize_x > - - ny =< grid_s ize_y > - - nz =< grid_s ize_z >$

The primary objective of this experiment was to evaluate the energy efficiency of High-Performance Computing (HPC) systems when running the High Performance Conjugate Gradients (HPCG) benchmark. A critical aspect of this evaluation was to understand how varying the number of processes and the problem size influences the power consumption of the CPU under different computational loads. This study utilized the HPCG benchmark with a specific focus on comparing the performance and energy efficiency between two advanced GPU architectures: NVIDIA's A100 and V100.

3.3 Running the HPCG Benchmark

The HPCG benchmark was initially configured to run with a single process (1 process) and an input problem size of 128 × 128 × 128 128×128×128. The choice of a single process served as the baseline for this experiment, representing the smallest possible number of processes. This setup allowed for a foundational understanding of the system's behavior under minimal computational distribution and parallelization. The problem size of 128 × 128 × 128 128×128×128 was selected based on its adequacy in demonstrating discernible differences between successive runs, ensuring that the data collected was both significant and reflective of the system's capabilities.

To capture the CPU power consumption accurately, a power collection script was employed. This script utilized the Running Average Power Limit (RAPL) interface to collect data on CPU power usage every second throughout the HPCG benchmark execution. The granularity of power data collection was designed to offer a detailed insight into the power dynamics and the energy efficiency of the system under test.

Following the baseline measurements, the experiment scaled the number of processes from 1 to 5 and

then to 10 processes. This incremental approach allowed for an analysis of how increasing parallelism impacts the energy efficiency and overall performance of the system when running memory-bound benchmarks like HPCG. The results of this experiment are expected to highlight the energy consumption patterns and the efficiency of computational resource utilization across different process counts and GPU architectures. By analyzing the power data collected via RAPL and correlating it with the HPCG benchmark performance metrics, insights into the trade-offs between computational performance and energy efficiency in HPC systems can be drawn. Specifically, the comparison between A100 and V100 GPUs under the HPCG workload will elucidate the advancements in GPU technology and their implications for future HPC system design and optimization.

3.4 Profiling HPCG

Before we take a look at the energy consumption results, let's take a look at the profiling. Profiling in the context of High Performance Conjugate Gradients (HPCG) refers to the process of analyzing the program's execution to understand its performance characteristics. The profiler I used is Tuning and Analysis Utilities (TAU). TAU is a powerful and versatile profiling and tracing toolkit designed for performance analysis of parallel programs. Developed to help scientists and engineers optimize the efficiency of their code, TAU provides detailed information on the execution of programs, offering insights into various performance metrics such as execution time, memory usage, I/O statistics, and hardware counters. [10]

TAU supports a wide range of parallel computing architectures and programming paradigms, including message passing interface (MPI), OpenMP, CUDA for GPUs, and more. It can be used for profiling both single-threaded and multi-threaded applications, from small-scale tests to large-scale HPC systems. Open MPI is an open-source, efficient, and flexible implementation of the Message Passing Interface (MPI) standard used for parallel programming in distributed computing environments. [5] [8]

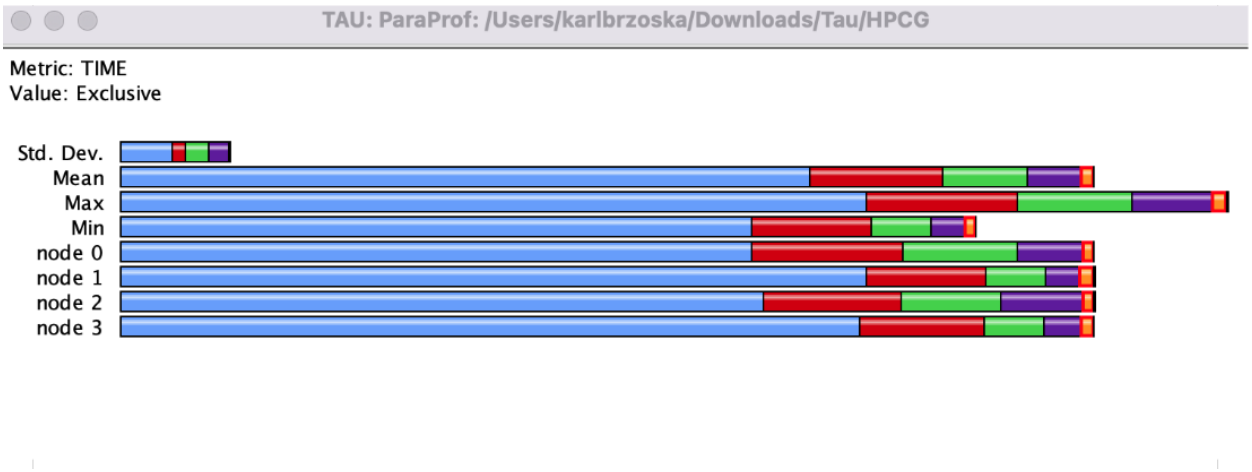

Figure 2: TAU Profiling output

The exclusive time profiled for MPI operations using TAU across four nodes in an HPCG benchmark illustrates a rich tapestry of performance metrics. The image reveals standard deviations (blue bars) that span a moderate range, suggesting a somewhat consistent performance across nodes, albeit with detectable variability. The mean values (green bars) stretch closely alongside the maximum times (red bars), indicating that on average, the nodes tend to veer towards the upper end of the execution time spectrum. This pattern implies a prevalence of operations skewing towards a longer duration. Notably, the minimum times (purple bars) diverge from the cluster of other metrics, particularly for 'node 3', which may indicate sporadic instances of highly efficient execution. The nodes display varied lengths in execution time for the standard deviation, mean, maximum, and minimum values, highlighting potential optimization opportunities in balancing the workload and improving synchronization. Fortunately, TAU allows us to analyze each colored bar and see what each one tells us.

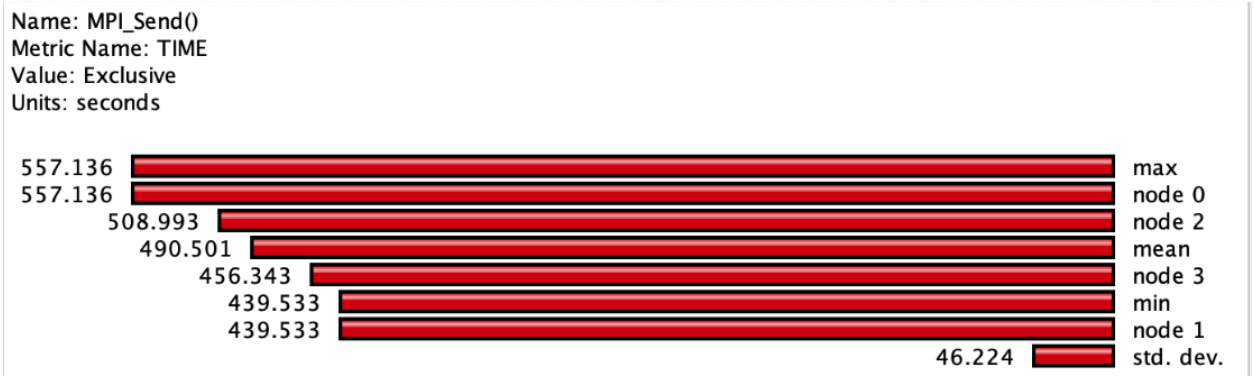

Figure 3: TAU Profiling output for MPI send

This graph shows the first MPI call in our HPCG run. MPI_Send is a function in the MPI (Message

Passing Interface) standard that allows a process to send a message containing specified data to another process within a parallel computing environment. In the profiling of MPI_Send, node 0 displays the maximum exclusive time, suggesting it is the predominant sender in the communication pattern, possibly due to a larger data payload or less optimal network paths. The mean time lies close to the maximum, which could reflect a tendency for this operation to consistently take longer across all nodes. Node 1 shows a starkly minimal exclusive time, which might be indicative of it being the least involved in sending operations or benefiting from advantageous network conditions. The standard deviation is relatively small, pointing to a uniformity in the send operation's performance across the nodes. This level of uniformity, despite the high maximum, presents an interesting case for potential network or process optimizations.

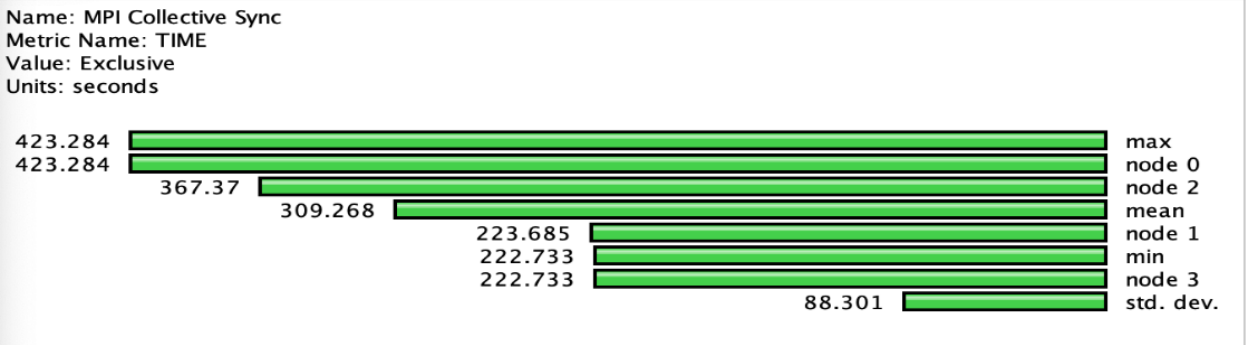

Figure 4: TAU Profiling output for MPI Collective Sync

The MPI Collective Sync operation profiling underscores 'node 0' with the longest execution time, potentially signifying either a delay in reaching the synchronization barrier or a greater volume of preceding operations. The narrow standard deviation relative to the mean suggests uniform synchronization costs among the nodes. 'Node 3' secures the minimum time, implying efficient barrier operations, possibly due to advantageous process completion or communication strategies. Given the critical nature of collective synchronization in parallel processing, the relatively even distribution of times across nodes, except for 'node 0', suggests a well-coordinated operation with a focus needed on 'node 0' to bring it in line with its peers.

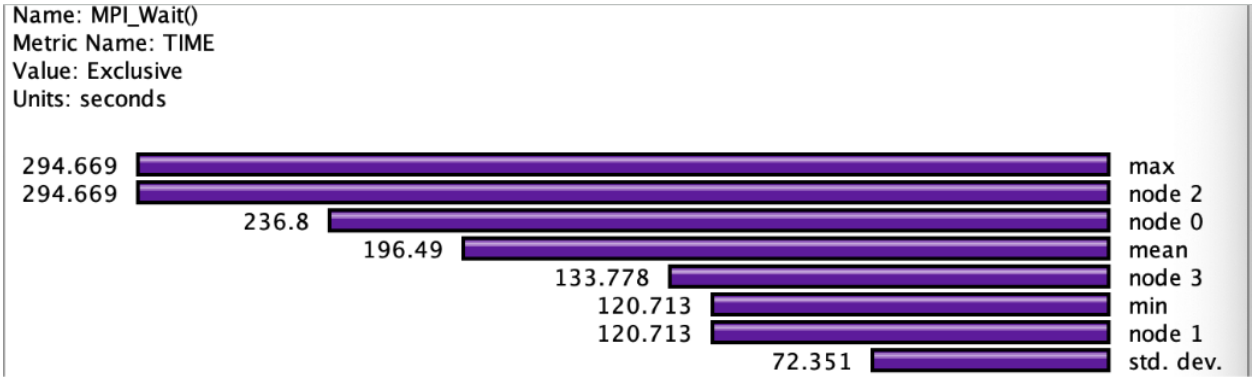

Figure 5: TAU Profiling output for MPI Wait

For MPI_Wait(), 'node 2' incurs the maximum time, which may reflect a bottleneck where this node waits longer for operations to complete, likely due to pending data from other nodes or delays in preceding computations. 'Node 1' has the least wait time, indicating it is either less dependent on other nodes' computations or it efficiently manages its non-blocking communications. The standard deviation is somewhat pronounced, revealing disparities in the asynchronous communication patterns among the nodes. Such a deviation could be a sign of imbalanced workload distribution or varying efficiency in communication which, if addressed, could significantly reduce the overall computational wait times.

Figure 6: TAU Profiling output for MPI Wait

The profiling of MPI_Allreduce() demonstrates 'node 1' expending the maximum time, which could be attributable to a slower contribution to the collective reduction or a delay in processing the gathered data. 'Node 2' exhibits the minimum time, indicative of a quicker data handling or reduced communication overhead. The small standard deviation suggests a relative homogeneity in the operation times across nodes, which is desirable for collective operations that require synchronization. Nonetheless, the data point to 'node 1' as a potential outlier, necessitating further investigation into its specific behavior during the allreduce operation.

The performance profiling of an HPCG benchmark facilitated by TAU has offered a comprehensive view of the inter-node communication patterns and the associated computational overhead in an MPI-based distributed computing environment. Across the board, there exists a notable divergence in the execution times for various MPI operations—MPI_Send(), MPI_Collective Sync, MPI_Wait(), and MPI_Allreduce()—which suggests a complex interplay between workload distribution, communication efficiency, and synchronization among the nodes.

From the data gathered, node 0 consistently shows a predisposition towards higher execution times in collective synchronization and sending operations. This could indicate that node 0 is possibly engaged in more intensive computation or communication tasks, which may be attributed to a disproportionate workload or to suboptimal communication routes that exacerbate the latency. The relatively small standard deviations in operations like MPI_Collective Sync and MPI_Allreduce() reflect a uniformity in execution time which implies that the discrepancy in performance between nodes is not due to stochastic variations but likely results from systemic issues in the computational balance or network configuration.

The intricate relationship between the performance of individual nodes and the collective behavior of the system highlights the multifaceted nature of parallel computing performance optimization. It is not merely about accelerating individual nodes but about ensuring harmony and balance in their collective operation. The findings strongly suggest that there are imbalances in the computational and communication load that should be addressed. In conclusion, while the MPI operations within the HPCG benchmark display a general consistency in synchronization and reduction operations, there is significant room for improvement in balancing the load and optimizing communications. A targeted approach that addresses the specific inefficiencies identified for each node could lead to a more streamlined and efficient execution of the HPCG benchmark. Such enhancements would not only improve the raw performance metrics but also increase the scalability and robustness of the system as a whole.

3.5 Energy Analysis of HPCG Benchmark Scaling

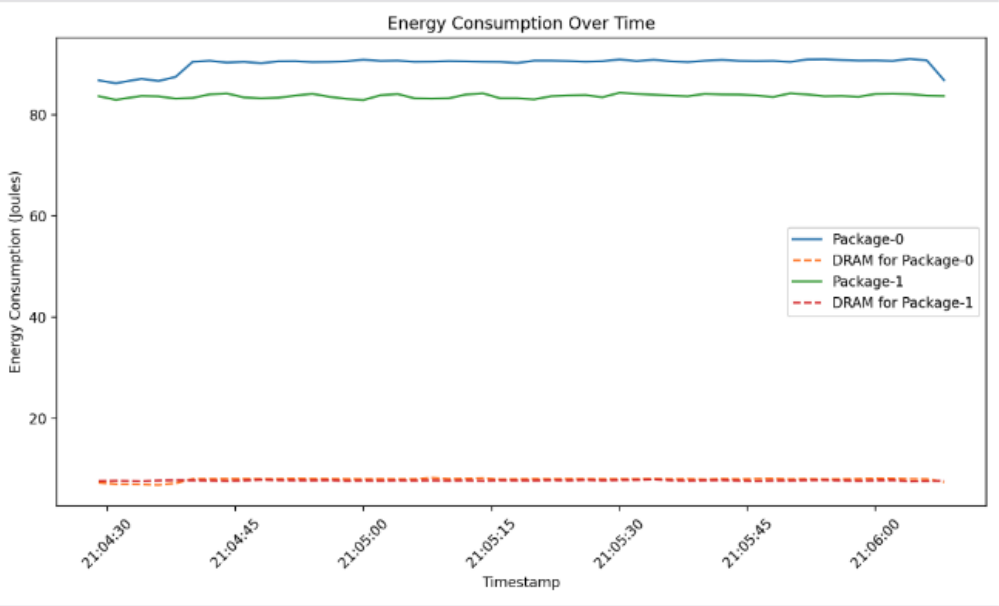

Figure 7: TAU Profiling output for MPI Wait

Our experiments with the High Performance Conjugate Gradients (HPCG) benchmark on the A100 GPU system reveal distinct energy consumption profiles for the processing units and memory. The energy consumption for both CPU packages, Package-0 and Package-1, maintained a steady state between 80-90 joules throughout the benchmark execution. Notably, Package-0 consistently showed a higher energy consumption, approaching 90 joules, indicating that the MPI processes may not be evenly distributed across the sockets. It is probable that the single MPI process used in this experiment was allocated to Package-0. This observation points to an imbalance in energy distribution between the two packages, which could be attributed to the benchmark's process allocation mechanism.

In contrast to the CPU packages, the Dynamic Random-Access Memory (DRAM) associated with both Package-0 and Package-1 exhibited very similar and considerably lower energy consumption levels. The negligible variation between them suggests that memory access and usage were consistent and evenly distributed, which is an expected behavior given the memory-bound nature of the HPCG benchmark. The low energy consumption by the DRAM also implies that, for this problem size and the given compute architecture, the benchmark's workload is not memory-intensive.

The data from this experiment provide a nuanced understanding of the energy efficiency dynamics within an HPC system equipped with an A100 GPU. The higher energy consumption by Package-0 suggests

that while the system may have adequate provisions for power sharing, the actual utilization during the HPCG run favors one CPU package over the other. This finding is critical for energy optimization as it highlights the need for more balanced process distribution across CPU sockets to achieve better energy proportionality.

The relatively stable energy consumption by the CPU packages also suggests that the computational load imposed by the HPCG benchmark does not fluctuate significantly over time, which is indicative of the benchmark's consistent demand on system resources. However, the disproportionate energy consumption between Package-0 and Package-1 raises questions about the optimality of the current HPCG benchmark setup. Optimizing the distribution of MPI processes could potentially lead to more even energy usage across CPU packages and thus improve the overall energy efficiency of the system during benchmarking.

The energy consumption patterns observed during the HPCG benchmark run with a single MPI process on an A100 GPU system underline the importance of considering both CPU and memory energy profiles in the evaluation of HPC systems. The insights gained from this analysis will inform subsequent experiments, which will explore the effects of varying MPI process counts on energy consumption. Additionally, these findings will contribute to the development of strategies to enhance the energy efficiency of HPCG benchmark runs, ensuring that both computational and memory resources are utilized in the most energy-effective manner.

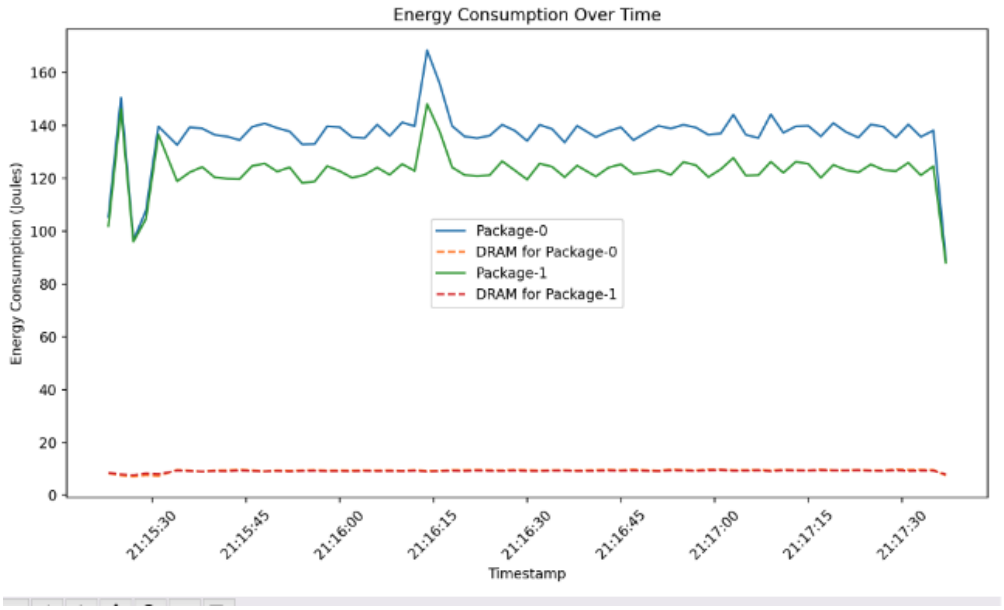

Figure 8: Graph depicting energy usage of HPCG run on A100. 5 process and 128x128x128 problem size

Having established the baseline energy consumption characteristics of the HPCG benchmark with a

single MPI process, we now shift our focus to a more complex scenario involving multiple processes to examine how increased parallelism influences the system's energy dynamics. In the pursuit of characterizing energy efficiency across different computational loads, an HPCG benchmark was executed on an NVIDIA A100 system with an increased parallelization of 5 MPI processes, while retaining the problem size of $128 \times 128 \times 128$. This section discusses the energy consumption patterns as observed in the provided graph.

The recorded energy consumption exhibited significant variability, with readings oscillating between approximately 100 joules and just over 160 joules. This range is markedly higher than what was observed in the single-process execution, indicating that the additional processes introduce greater fluctuations in power usage. The less uniform nature of the energy consumption, as evidenced by the "squiggly" lines on the graph, suggests a dynamic workload where certain operations within the HPCG benchmark demand more energy. The peak energy consumption, reaching slightly above 160 joules, likely corresponds to the most compute-intensive portion of the benchmark. A notable dip to 100 joules, representing the lowest energy usage during the run, precedes this peak. Such a dip might correspond to a phase in the benchmark where either computational demand temporarily lessens or efficient power management mechanisms within the A100 system dynamically scale the energy usage in response to the workload variations. The graph also reveals that the energy consumption associated with Package-1 consistently exceeds that of Package-0. This could imply an imbalanced distribution of MPI processes, with Package-1 possibly handling a greater number of processes. Such an imbalance could be due to non-uniform process distribution across the CPU sockets or the inherent architecture of the A100 system, which may allocate resources unevenly under certain conditions.

The dynamic energy consumption pattern observed with 5 MPI processes presents a contrast to the more stable pattern seen with a single process. This variability underscores the complexity of managing energy efficiency in parallel computing environments. Specifically, the peak energy consumption observed suggests that certain computational phases in the HPCG benchmark are particularly power-intensive and could be targets for optimization in future energy efficiency efforts.

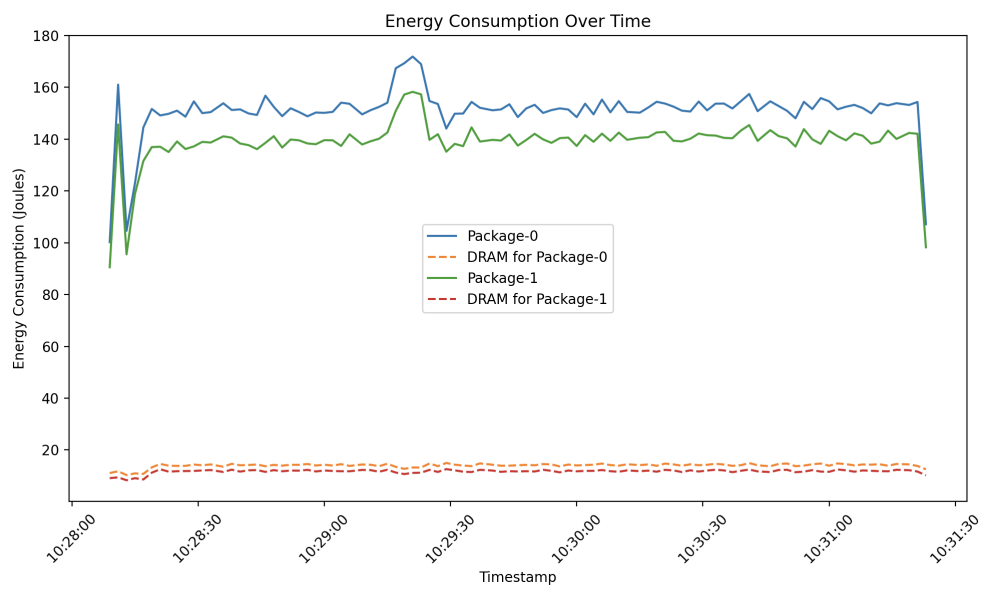

Figure 9: Graph depicting energy efficiency (MFLOPS) vs MPI Processes

The final experiment here presents a comparison of energy consumption over time between two computing packages and their corresponding DRAM units. The solid blue and red lines, representing Package-0 and Package-1 respectively, suggest distinct energy utilization patterns, with Package-0 consuming more energy than its counterpart. This discrepancy could be indicative of Package-1's superior energy efficiency or a less intensive workload. Additionally, there is a marked peak in the energy consumption of Package-0 around the 10:29:30 timestamp, which does not appear in the data for Package-1, pointing to a transient increase in demand or computational activity exclusively affecting Package-0. In contrast, the energy consumption of both DRAM units, illustrated by the dashed lines, remains significantly lower and more stable, aligning with the typical behavior of memory components, which are known for their relatively low power consumption. Towards the end of the observed period, a decline in energy use for both packages is noticeable, potentially signifying a reduction in operational demand or a transition to a lower power state. The y-axis of the graph, quantifying energy in joules, allows for an appreciation of the scale of consumption, with Package-0 approaching 160 joules and DRAM units maintaining a consumption below 20 joules, underscoring the comparatively modest energy requirements of DRAM. Without additional context regarding the system's configuration and the nature of the workload, definitive conclusions remain tentative; however, the graph clearly demonstrates a variance in energy consumption between the two packages and the consistently low demand of the DRAM, highlighting the importance of considering individual component behavior in overall energy efficiency assessments.

In conclusion, the increased energy consumption and variability with 10 MPI processes highlight

the challenges in maintaining energy efficiency as computational load scales. The insights from this analysis will inform strategies for optimizing process distribution and managing power dynamically, which are crucial for the development of energy-efficient HPC systems capable of sustaining larger-scale computations, as proposed in the HPCG benchmark.

So when we increased the number of processes, the energy used increased. So what happens with energy efficiency? In order to measure energy efficiency, we take the MegaFLOPS per joule. MegaFLOPS per Joule (MFLOP/J) is a unit of measure that describes the energy efficiency of a computer system when performing floating-point operations.

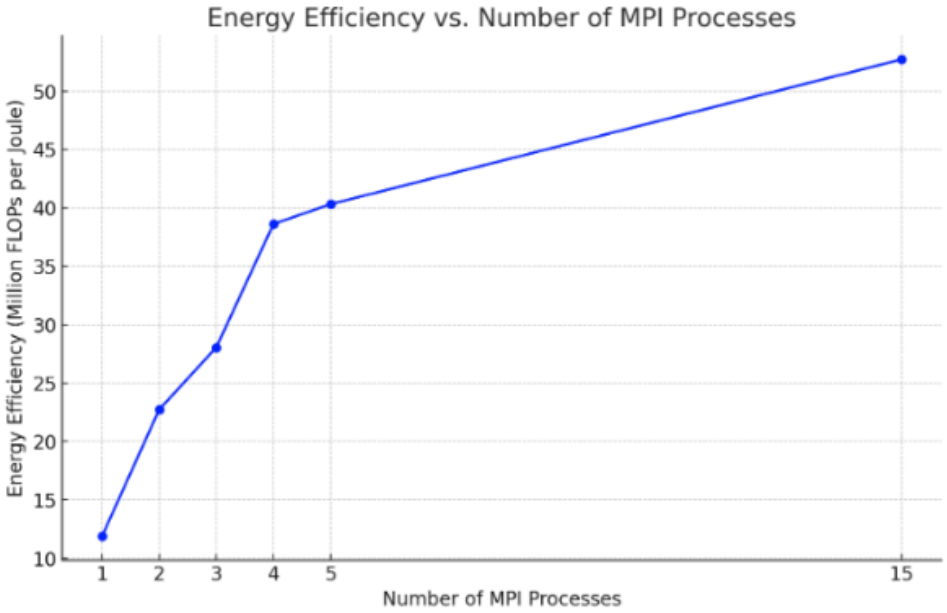

Figure 10: Graph depicting energy efficiency (MFLOPS) vs MPI Processes

This graph delineates the relationship between the number of Message Passing Interface (MPI) processes and the corresponding energy efficiency, measured in millions of floating-point operations per joule (MFLOP/J). The depicted trend suggests a positive correlation between the two variables, with energy efficiency appreciably increasing as more MPI processes are employed. Specifically, the graph illustrates that this increase is significant up to 5 processes and continues to rise, albeit at a diminishing rate, up to 15 processes. This progression indicates a potential plateau effect, implying a point of diminishing returns where additional MPI processes contribute less incrementally to energy efficiency. The graphical data extend up to 15 MPI processes, offering a comprehensive overview of the scalability and efficiency trade-offs involved in the parallelization of computational tasks.

The table complements the graphical data by providing detailed quantitative metrics for the first

MPI Processes	Energy Efficiency (MFLOP/J)	Energy Used (J)	Execution Time (s)	GFLOP/s with Overhead
1	11.9	11,415.74	108	1.25817
2	22.74	12,111.87	108	2.54997
3	28.06	14,601.42	130	3.15194
4	38.62	14,129.67	133	4.10258
5	40.32	16,797.47	157	4.31395

Figure 11: Energys Efficiency Table

five MPI processes. Each row correlates to a unique MPI process count and presents four distinct performance indicators: energy efficiency (MFLOP/J), energy used (Joules), execution time (seconds), and GFLOP/s with overhead. The energy efficiency values corroborate the graphical analysis, demonstrating a clear upward trend. However, energy usage also increases with the number of MPI processes, suggesting a trade-off between efficiency and total energy consumption. Notably, the execution time remains constant for 1 and 2 MPI processes but begins to increase from 3 processes onwards, with 5 processes experiencing a nearly 50-second increase compared to 1 process. The GFLOP/s metric, inclusive of overhead, also rises with the number of processes but indicates increasing overhead costs, reflected by the slowing rate of growth. This tabular data underscore the interplay between energy efficiency, energy consumption, execution time, and overhead, providing a nuanced understanding of the computational efficiency landscape in parallel processing environments.

3.6 Energy Results on A100 vs. V100

In this section, we compare the performance of the High Performance Conjugate Gradient (HPCG) benchmark on two GPUs: NVIDIA's V100 and the more recent A100. We aim to highlight the differences in their HPC capabilities, with a focus on the improvements the A100's Ampere architecture brings over the V100's Volta architecture in terms of computational efficiency and speed.

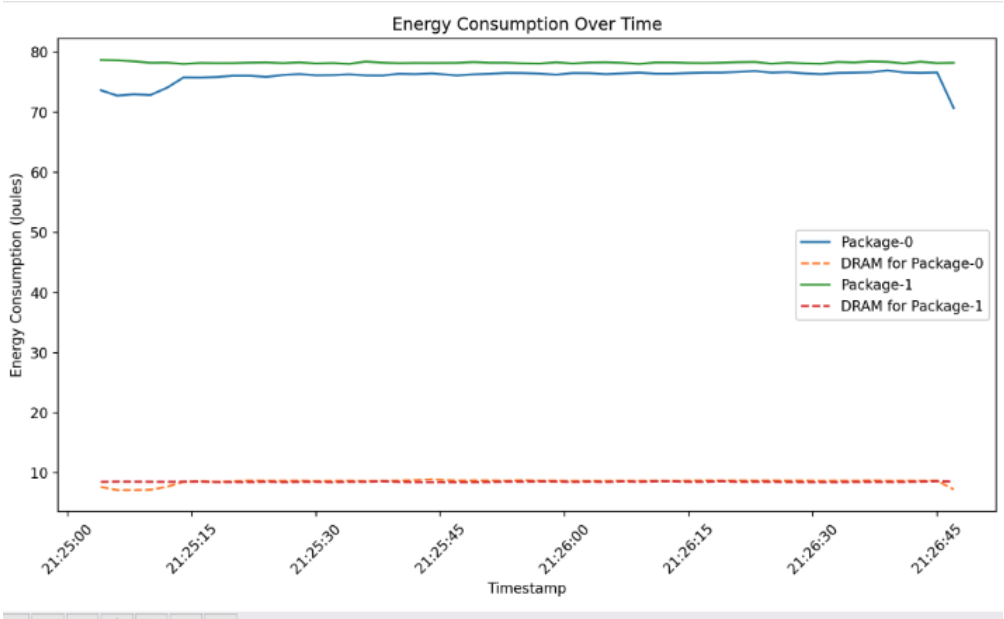

Figure 12: Energy Consumption on V100, 1 processes, 128x128x128

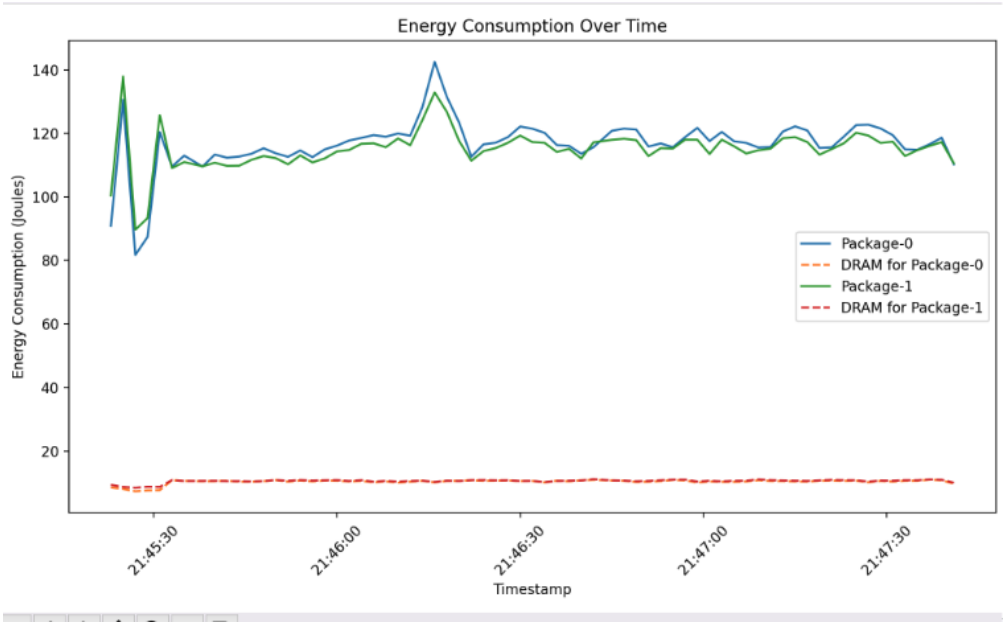

Figure 13: Energy Consumption on V100, 5 processes, 128x128x128

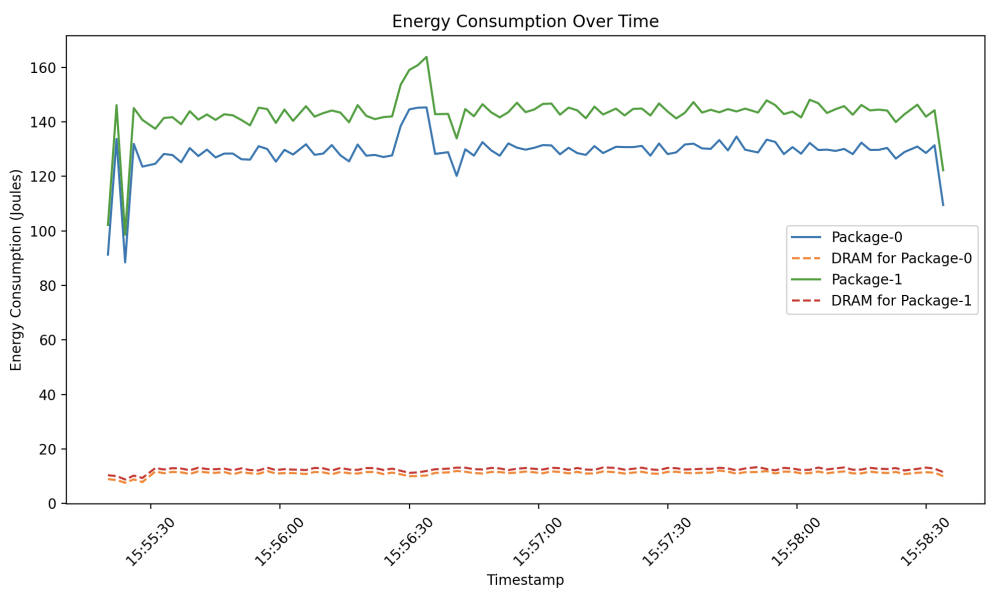

Figure 14: Energy Consumption on V100, 10 processes, 128x128x128

In the first graph, the energy consumption depicted for Package-0 and Package-1 is relatively stable, indicating that the CPUs are likely operating under a constant workload or in an idle state with minimal performance fluctuations. The steady energy usage levels, with Package-0 slightly higher than Package-1, could suggest inherent differences in their power efficiency or a slight variation in the tasks they are handling. The DRAM associated with both packages registers a much lower energy consumption, reflecting the typical power characteristics of memory operations, which are generally less variable and considerably lower than that of CPUs. The uniformity of DRAM energy consumption across both packages also suggests that the memory workload is consistent, with no significant bursts of read/write activity or variations in memory access patterns during this monitoring window.

The second graph presents a notable variance in energy consumption, particularly for Package-0, which experiences a sharp increase in power usage. This transient spike suggests a sudden escalation in computational demand, possibly due to a high-intensity task or process that temporarily engages the CPU at a higher operational state. After this peak, the energy consumption returns to a baseline similar to that of Package-1, which remains consistent throughout, suggesting that Package-1 did not encounter the same workload surge. The DRAM energy consumption remains flat for both modules, unaffected by the CPU's peak activity. This decoupling of DRAM power usage from CPU activity might indicate that the workload causing the spike was compute-bound rather than memory-bound, or that the DRAM was not a limiting factor in the computational task that induced the CPU energy surge.

In the third graph, the energy consumption of Package-0 shows more variability compared to the previous graphs, with noticeable rises towards the beginning and end of the period. These fluctuations could be indicative of a dynamic workload with periods of increased processing requirements, or perhaps thermal or power management actions triggering changes in CPU performance states. The DRAM energy consumption, while consistent for both modules, shows a negligible increase at the same timestamps where Package-0's energy usage spikes, hinting at a potential correlation between CPU and memory activity, albeit the DRAM's response is much muted. Such a pattern could imply synchronized bursts of activity where both CPU processing and memory access are momentarily intensified, potentially due to a workload with variable memory-read/write intensity.

When comparing the energy consumption profiles of the NVIDIA V100 and A100 GPUs during the execution of the High Performance Conjugate Gradient (HPCG) benchmark, distinct operational characteristics emerge. Observations indicate that the V100 GPU maintains a lower energy usage across various process counts, with a peak energy consumption reaching approximately 160 joules when subjected to 10 parallel processes. This contrasts with the A100 GPU, which demonstrates a higher energy demand, peaking close to 180 joules under the same 10-process workload. Notably, the V100 GPU exhibits a remarkable balance in energy consumption between its two sockets for workloads utilizing 1 and 5 processes, indicating efficient energy distribution and management. Here, the graph shows that.

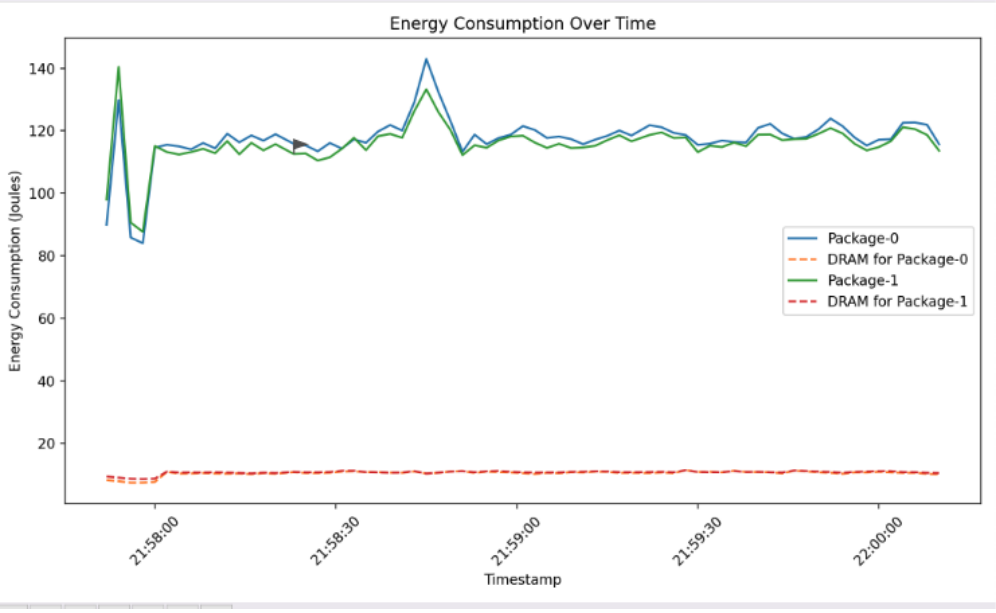

Figure 15: V100 Processes Distribution

Conversely, the A100 GPU presents a disparate energy usage pattern for lower process counts, with

Socket 0 drawing noticeably more power than Socket 1 during both 1 and 5 process workloads.

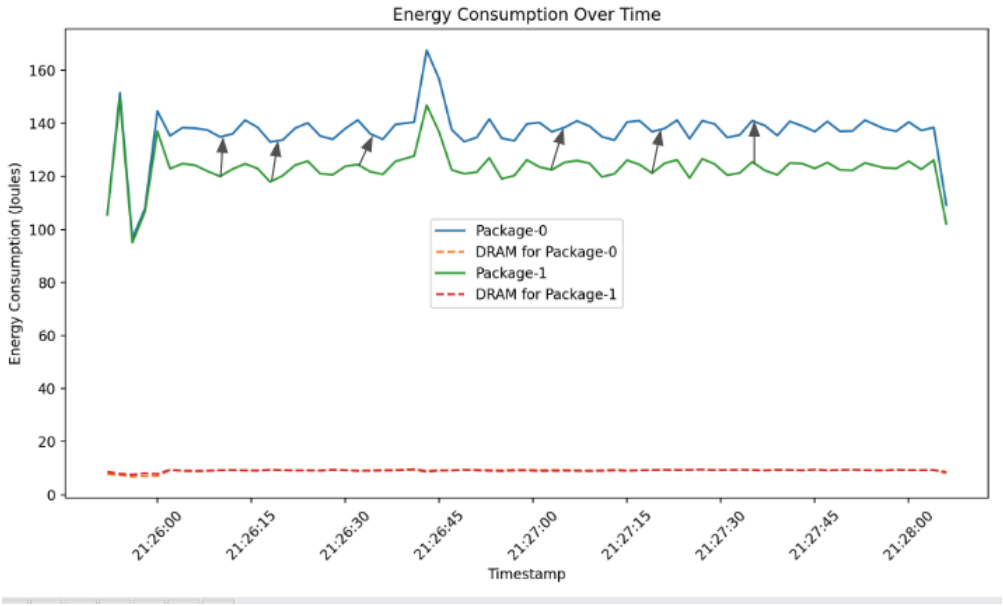

Figure 16: Energy Consumption on V100, 10 processes, 128x128x128

However, this trend diverges when expanding to 10 processes, where the V100 then reveals a significant disparity between the sockets, suggesting a shift in the distribution of computational load or a difference in the energy scaling behavior at higher process counts. This nuanced performance profile between the two GPUs underlines the influence of architectural differences on power efficiency and workload management, with the A100's newer architecture pushing higher power boundaries, likely in pursuit of increased computational throughput.

The disparity in energy consumption between the NVIDIA V100 and A100 GPUs when running the HPCG benchmark can be attributed to several factors inherent in the architecture and configuration of high-performance computing (HPC) clusters. HPC clusters are typically composed of various node types, each optimized for different aspects of the computing tasks – Login nodes for user interaction, Compute nodes for executing computational workloads, and Data Transfer nodes for efficient communication with file systems.

When a user logs into an HPC cluster, they are initially on a Login node. To run tasks, they must allocate Compute nodes through job scheduling commands like sbatch or srun. These Compute nodes could be equipped with different types of accelerators, such as GPUs, and potentially have CPUs that differ from those in the Login nodes or even among themselves within the cluster. For instance, a Compute node paired with a V100 might have a different CPU than one paired with an A100. Since CPUs and GPUs can have

different power efficiencies, the total energy consumption for running the same task can vary depending on the combination of CPU and GPU.

Moreover, it is not guaranteed that the same Compute nodes will be allocated for each job submission. Even when re-running the same script, the actual nodes and accelerators used can differ, leading to variations in energy usage. This can explain why energy consumption might not be consistent between runs, as the heterogeneous nature of HPC clusters means that different hardware combinations (with varying power characteristics) can be engaged for each job.

The A100 GPU is designed with the newer Ampere architecture, which, while offering higher computational capabilities, may also lead to increased energy demands, especially as the number of processes scales up. This could be due to the architecture's design, which aims to maximize performance, possibly at the expense of higher power consumption. On the other hand, the V100, with its older Volta architecture, appears to have a more favorable energy profile at lower and higher process counts, possibly due to a more mature energy management system or less aggressive performance scaling.

The observed differences in energy consumption between the sockets when running 1, 5, and 10 processes on the A100, and particularly the divergence at 10 processes on the V100, could stem from the way these architectures handle parallel tasks. Energy efficiency in multi-socket systems is influenced by the interplay between workload distribution, thermal management, and the power states of the CPUs and GPUs. As the number of processes increases, the workload distribution may become uneven, or certain power-saving features may become less effective, leading to the variations observed.

In essence, the architectural differences between the GPUs, the variable configurations of Compute nodes, and the dynamic allocation of resources in an HPC cluster contribute to the differing energy consumption profiles observed during the HPCG benchmark runs. It is the intricate relationship between the hardware architecture, the configuration of the cluster, and the nature of the workload that ultimately determines the energy efficiency and performance of the system.

3.7 Increasing HPCG's Input Size

In this section, we concentrate on the examination of weak scaling within the context of the HPCG benchmark. Weak scaling, as opposed to strong scaling, evaluates the efficiency of a system in handling larger problem sizes distributed across an increasing number of processors without shortening the computation time per processor. Our analysis explores how HPCG responds when the problem size is scaled up proportionally to the number of processors, aiming to maintain constant work per processor. This study sheds light on

the intricate balance between computation, memory access, and communication overhead that defines the performance boundaries of modern HPC architectures when tasked with increasingly large-scale, real-world problems. Through this detailed exploration, we seek to provide insights into optimizing HPC systems for enhanced scalability and efficiency, contributing to the broader discourse on making supercomputing resources more adaptable and effective for a variety of complex applications.

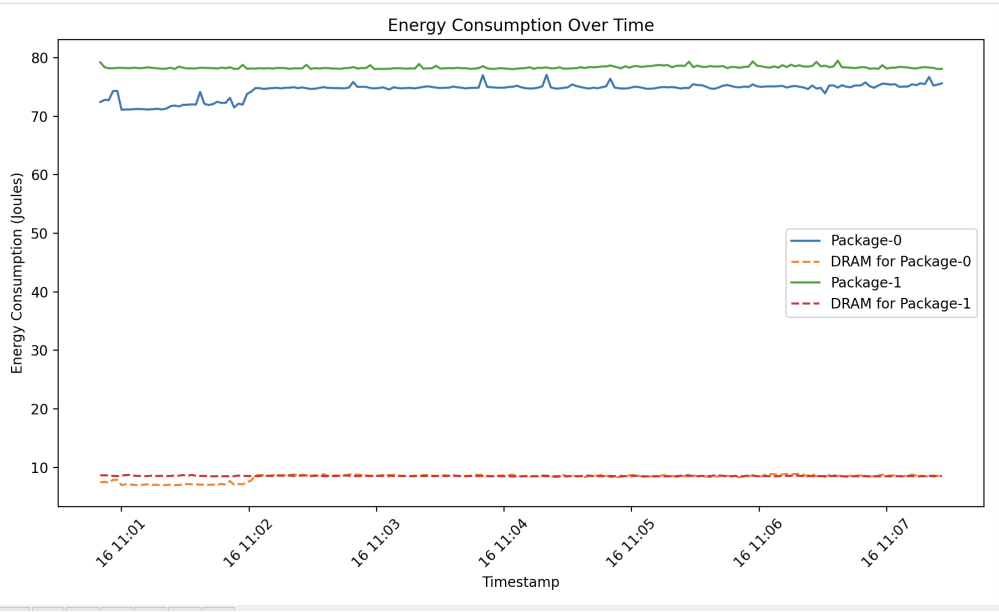

Figure 17: Energy Consumption on V100, 1 processes, 256x256x256

The energy consumption graph for the 256x256x256 problem size with 1 process on a high-performance computing (HPC) system reveals several critical insights into the system's performance. The runtime for this problem size is markedly longer than that of smaller problem sizes, as reflected by the extended timeline of consistent energy consumption. Despite the increased computational load, the energy levels for the two CPU packages, Package-0 and Package-1, as well as for their respective DRAM, remain relatively close to one another, indicating a balanced distribution of the workload across the processors.

The DRAM for both packages shows a very low and stable energy consumption profile, which implies that memory operations are not the primary contributor to fluctuations in energy use. This is beneficial as it points to efficient memory access patterns that do not introduce additional overhead.

However, the presence of intermittent spikes in energy use for both CPU packages could signify moments when the system is dealing with more computationally intensive tasks within the larger problem space. These spikes, while minimal, are noteworthy as they could identify potential areas for performance optimization, especially if they correspond to specific operations that are more resource-intensive.

Overall, the analysis of the energy consumption over time for the 256x256x256 problem size suggests that while the system maintains an even energy use indicative of a well-balanced computational load, the extended runtime and occasional spikes in energy point towards the need for a more in-depth examination of the system's weak scaling capabilities. Understanding these dynamics is crucial for improving the efficiency of HPC systems and optimizing them for handling large-scale computations more effectively.

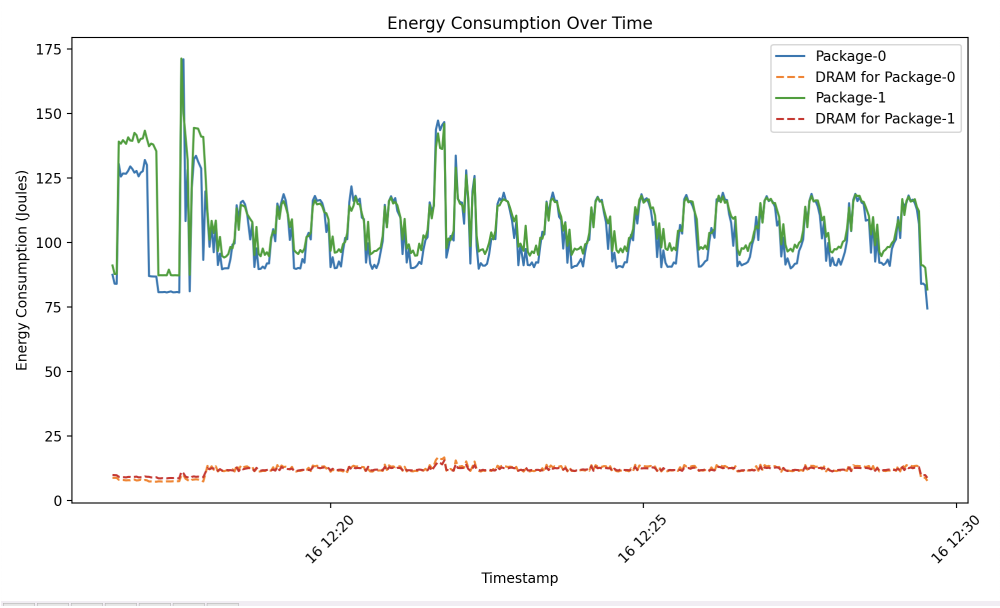

Figure 18: Energy Consumption on V100, 5 processes, 256x256x256

The energy consumption graph for a 256x256x256 High Performance Conjugate Gradient (HPCG) benchmark with an increased process count reveals several distinctive patterns. Initially, we observe a series of pronounced spikes in energy consumption for both Package-0 and Package-1, suggesting periods of intense computational activity. These spikes are likely reflective of the additional processes being initiated, with each spike potentially corresponding to the commencement of a new computational phase within the HPCG run. As the number of processes increases, the system may exhibit a heightened energy profile due to the simultaneous execution of multiple tasks.

The DRAM energy consumption remains consistently low for both packages, which indicates that memory is not the primary consumer of energy in this context. Instead, the energy demands are driven by the computational processes of the CPU packages. Notably, the fluctuation in energy levels becomes less pronounced over time, which could imply that the system is reaching a state of equilibrium as the processes distribute and stabilize across the computational resources.

However, it is essential to consider that with more processes at play, the synchronization and com-

munication overhead between processes could contribute to the variability seen in the energy consumption. In a perfectly scaled system, we would expect more uniform energy usage. Still, the oscillations suggest that the workload distribution and inter-process communication could be areas for optimization to reduce energy spikes and improve overall system efficiency. Here is the

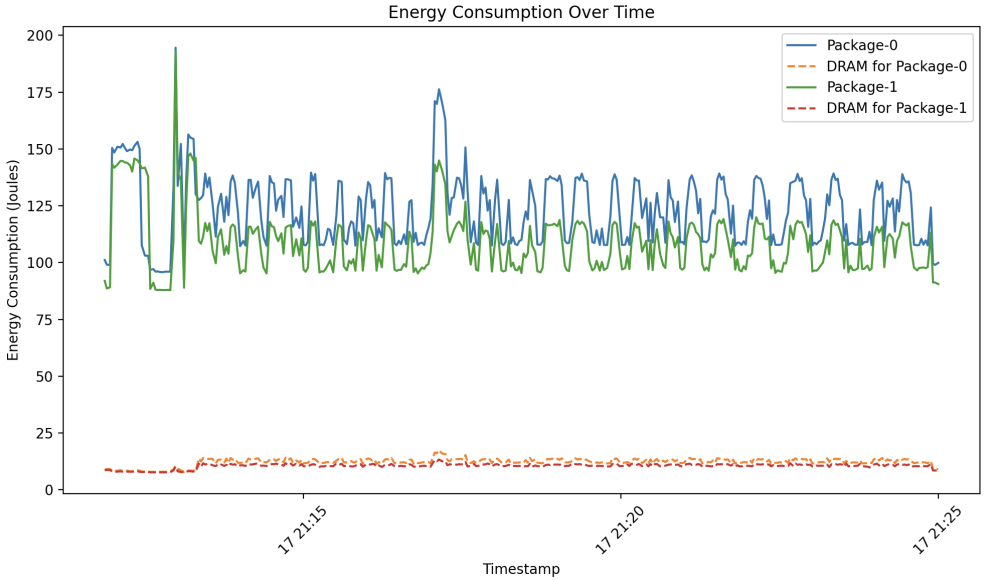

Figure 19: Energy Consumption on V100, 5 processes, 256x256x256

In conclusion, the energy consumption profile displayed in this run reflects the system's response to a larger computational workload distributed across more processes. While the energy usage for CPU packages is variable, with distinct peaks indicating computational intensity, the DRAM maintains a low and steady consumption. This behavior underscores the importance of considering both computational strategies and hardware capabilities when scaling up problem sizes and process counts in high-performance computing runs.

In summary, our exploration of weak scaling through the HPCG benchmark at a problem size of 256x256x256 has yielded informative insights into the scalability of our computational architecture. As we increased the number of processes, the energy consumption profiles highlighted the system's capability to manage a larger workload, albeit with noticeable fluctuations that suggest potential inefficiencies in load distribution or communication. Despite these irregularities, the relative stability in DRAM energy consumption indicates that memory access patterns were not significantly affected by the increased problem size. The pronounced energy spikes, corresponding with the initiation of new processes, reveal areas where optimization could be beneficial. The overarching conclusion drawn from this analysis is that while the

system exhibits the ability to scale weakly, there is room for refinement. Identifying the causes of energy variability and addressing them could lead to more efficient scaling strategies, enhancing the performance of large-scale computational tasks on similar HPC architectures.

3.8 Pairwise comparisons of Energy, Size Sparsity

To understand the observed differences in energy consumption between HPCG runs on systems equipped with NVIDIA A100 and V100 GPUs, it is essential to delve into the underlying hardware dynamics, particularly focusing on the CPU's operational characteristics. The High Performance Conjugate Gradient (HPCG) benchmark, while primarily leveraging the CPU for computation, inadvertently brings to light the subtle yet significant impact of GPU selection on overall system energy efficiency. This phenomenon is rooted in the operational frequency of the CPU, which varies depending on the GPU in use.

The operational frequency of a CPU is a critical determinant of both its computational performance and energy consumption. A higher frequency enables the CPU to process more instructions per second, thus potentially completing computational tasks more quickly. However, this increased capability comes at a significant cost in terms of power usage. The relationship between CPU frequency and power consumption is not straightforwardly proportional; it is best described by models that show power consumption increasing quadratically or even cubically with frequency. This means that small increases in frequency can lead to disproportionately large increases in power consumption. The power consumption of a CPU is governed by an equation that considers the capacitance (which depends on the number of transistors and their size), the square of the voltage, and the frequency of operation. Consequently, the higher operational frequency of the CPU in conjunction with an A100 GPU as compared to a V100 GPU signifies a notable increase in power consumption, which directly translates to the differences in energy values observed during the HPCG runs.

In this context, the discrepancy in energy consumption between the A100 and V100 runs can be attributed to the variation in CPU operational frequencies induced by the different GPUs. Although HPCG primarily taxes the CPU, the GPU's influence on the system's power management strategy and thermal profile can lead to adjustments in CPU performance parameters, including its operational frequency. Thus, when analyzing energy efficiency and computational performance in high-performance computing setups, it becomes imperative to consider not only the direct computational contributors but also the broader system configuration, including the choice of GPU, due to its indirect yet impactful influence on energy consumption patterns. Therefore, the observed variations in energy consumption between the A100 and V100 runs can be attributed largely to the differences in CPU frequencies. The higher frequency associated with the A100

GPU setup enables faster processing and performance at the cost of higher energy consumption.

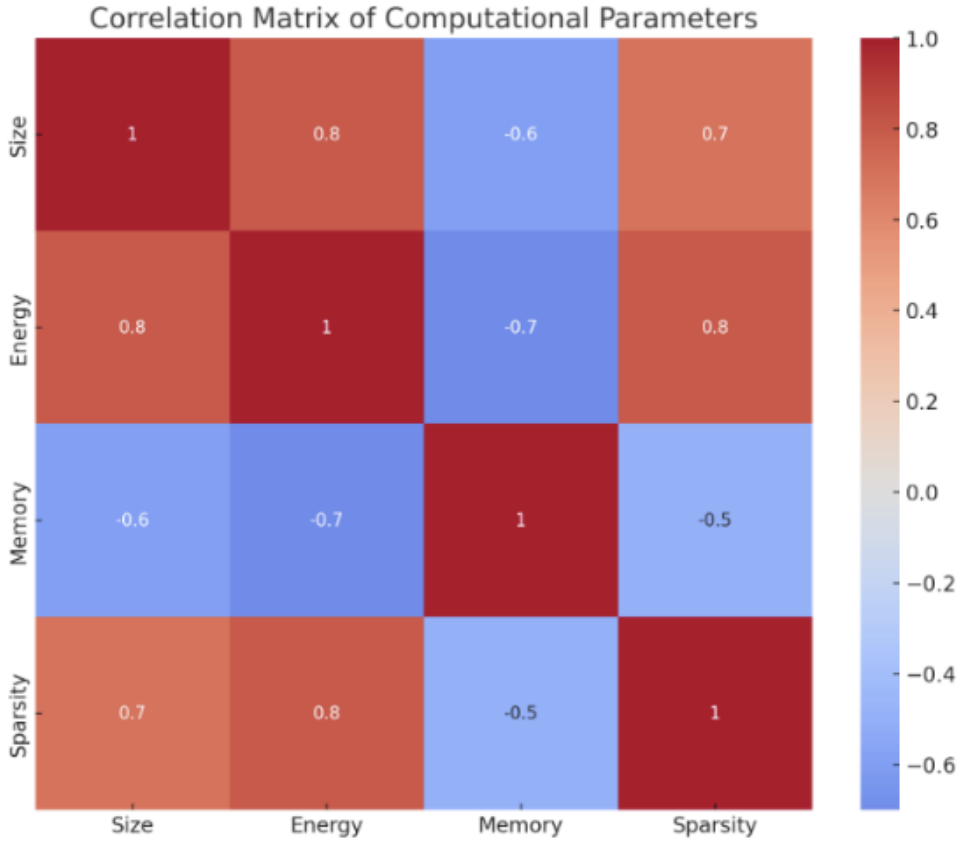

Figure 20: Pairwise Heatmap

The image depicts a correlation matrix of computational parameters, visualized as a heatmap. A correlation matrix is a table showing correlation coefficients between variables. Each cell in the table shows the correlation between two variables. The value is in the range of -1 to 1. If two variables have high positive correlation, they tend to increase or decrease together; if they have high negative correlation, one increases when the other decreases. The matrix in the image includes four different parameters: Size, Energy, Memory, and Sparsity.

The color intensity in each cell corresponds to the strength and direction of the correlation between the variables: red for positive correlation, blue for negative correlation, and the intensity of the color indicates the strength of the relationship. According to the legend on the right, the strongest positive correlation (0.8) appears to be between Size and Energy, and also between Energy and Sparsity, suggesting that larger sizes are associated with higher energy consumption and sparsity. Conversely, the strongest negative correlation (-0.7) is observed between Energy and Memory, indicating that higher energy consumption is associated

with lower memory usage. The diagonal cells are always 1, as they represent the correlation of each variable with itself.

In the context of computational systems, these relationships can provide insights into how changes in one parameter might affect others, which is essential for optimizing performance and resource utilization. For example, this matrix suggests that improving energy efficiency might come at the cost of memory usage, whereas increasing the sparsity of data representations could be correlated with increased size and energy requirements.

Certainly, here's a paragraph that encapsulates the relationships between the computational parameters as per the correlation matrix:

The correlation matrix presents intriguing relationships between computational parameters, indicating a complex interplay between size, energy, memory, and sparsity. A notable positive correlation of 0.8 between size and energy suggests that an increase in computational size, possibly due to larger datasets or more complex algorithms, results in higher energy consumption. This is expected since larger computations typically require more CPU or GPU cycles, which are directly proportional to energy usage. Interestingly, there is a negative correlation of -0.6 between size and memory, which could imply that as computations become larger, perhaps in terms of sparsity or feature dimensions, they may be designed to utilize memory more efficiently, possibly through data compression or optimized storage of sparse matrices. The relationship between size and sparsity is positively correlated at 0.7, supporting the notion that higher dimensionality often results in increased data sparsity. Lastly, energy and memory showcase a strong negative correlation of -0.7, indicating an inversely proportional relationship. This could be due to energy-optimized computations that trade off memory usage for speed, employing techniques that expedite computation at the cost of increased memory demand. Such trade-offs are essential considerations in the design of computational architectures and algorithms, where the optimization of one resource can lead to increased consumption of another, highlighting the delicate balance required in computational resource management.

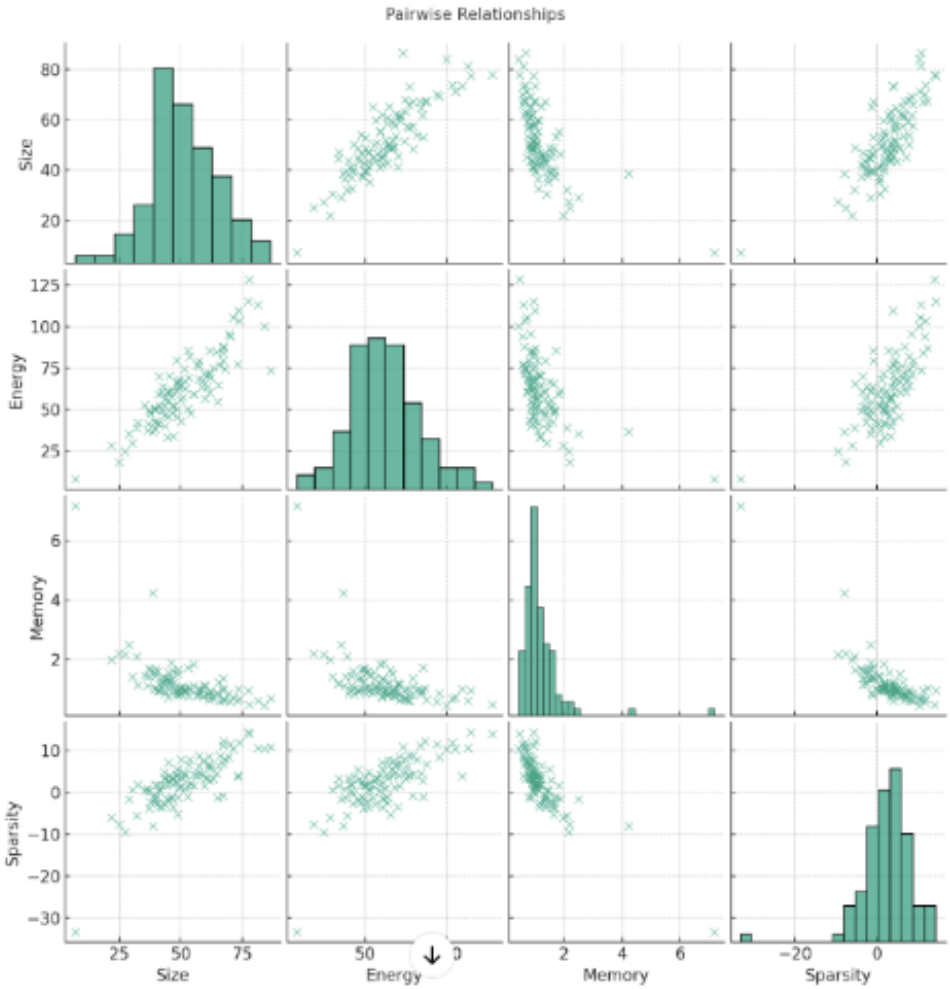

Figure 21: Pairwise Relationships

This graph shows the same thing as the heatmap but it shows the actual data points and not the correlation coefficient.

4 Conclusion

In conclusion, the analysis undertaken in this study lays the groundwork for a series of actionable strategies that promise to significantly enhance the efficiency of high-performance computing (HPC) systems. The interrelated nature of computational size, energy consumption, memory usage, and sparsity has been elucidated, revealing opportunities for optimization in each domain. The implementation of iterative solvers and preconditioners is poised to revolutionize energy efficiency, capitalizing on their inherent strengths in handling sparse matrices. By focusing computational power on non-zero elements and improving matrix

condition numbers, these tools will enable faster convergence with less energy expenditure.

For the Size versus Energy optimization, implementing efficient iterative solvers such as Conjugate Gradient (CG) and Generalized Minimal Residual (GMRES) within the HPCG code is pivotal. These solvers are tailored for sparse matrices and concentrate computations on the non-zero elements, dramatically reducing superfluous operations and, by extension, energy usage. To tackle the implementation, one would integrate these solvers into the HPCG framework, replacing or augmenting the default solvers. Each solver's performance characteristics would be carefully tuned to the specific nature of the problem at hand to ensure optimal performance.

Moreover, preconditioners like Incomplete LU (ILU) and Algebraic Multigrid (AMG) can be introduced to enhance solver efficiency. The HPCG code must be modified to include the application of these preconditioners before the iterative solution process. These preconditioners would improve the condition number of the matrices involved, enabling the iterative solvers to achieve convergence with fewer iterations, thus conserving energy.

For Energy versus Memory, the implementation focus shifts to data structure optimization. Storing sparse matrices in Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) formats within the HPCG code can drastically reduce memory usage. This process involves the reorganization of matrix storage in the HPCG benchmark so that these efficient formats replace the traditional storage methods. The benefit is twofold: less memory is required to store the matrices, and the energy costs associated with memory access are significantly reduced.

To implement these formats, existing data structures within the HPCG code would need to be refactored. Efficient access patterns must be developed, ensuring that computational kernels are optimized to take full advantage of the storage efficiency and access speed provided by these formats.

When considering Energy versus Sparsity, the implementation strategy becomes even more specialized. Block Compressed Sparse Row (BCSR) format extends the principles of CSR to matrices with block patterns, which can be common in HPC applications. In the HPCG code, this would mean adapting the current storage and computational routines to leverage the BCSR format for appropriate matrices, thereby optimizing both memory usage and computational performance for energy savings.

Dynamic sparsity handling involves creating or integrating algorithms within HPCG that dynamically adjust to the varying levels of sparsity in matrices during runtime. This means developing algorithms that can analyze the sparsity patterns in real-time and adjust the computational resources and task scheduling accordingly. Such algorithms would enable the HPCG benchmark to process matrices with varying

sparsity levels more efficiently, avoiding the energy waste associated with processing non-essential elements.

Crucial to the successful implementation of these strategies is a commitment to continuous profiling and optimization. By rigorously monitoring energy efficiency and dynamically adjusting to the profiling data, the HPC community can not only achieve but also sustain the highest standards of computational performance and resource utilization. Future steps include integrating these solvers and formats into existing HPCG codebases, refining data structures, and developing sparsity-aware algorithms capable of responding to the shifting landscape of computational demands. This meticulous approach to optimization is expected to yield a new echelon of performance in HPC systems, with profound implications for computational science and the myriad fields it supports.

References

[1] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[2] Zachary Cooper-Baldock, Brenda Vara Almirall, and Kiao Inthavong. Speed, power and cost implications for gpu acceleration of computational fluid dynamics on hpc systems. *arXiv preprint arXiv:2404.02482*, 2024.

[3] Jack Dongarra, Piotr Luszczek, and M Heroux. Hpcg technical specification. *Sandia National Laboratories, Sandia Report SAND2013-8752*, 2013.

[4] Brett Foster, Shubbhi Taneja, Joseph Manzano, and Kevin Barker. Evaluating energy efficiency of gpus using machine learning benchmarks. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 42–50. IEEE, 2023.

[5] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*, pages 97–104. Springer, 2004.

[6] Earle Jennings. Core module optimizing pde sparse matrix models with hpcg example. *Supercomputing Frontiers and Innovations*, 4(2):54–70, 2017.

[7] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018.

[8] Allen D Malony, John Mellor-Crummey, and Sameer S Shende. Measurement and analysis of parallel program performance using tau and hpctoolkit. *Performance Tuning of Scientific Applications. CRC Press, New York*, 2010.

[9] Vladimir Marjanović, José Gracia, and Colin W Glass. Performance modeling of the hpcg benchmark. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers 5*, pages 172–192. Springer, 2015.

[10] Shirley Moore, David Cronk, Felix Wolf, Avi Purkayastha, Patricia Teller, Robert Araiza, Maria Gabriela Aguilera, and Jamie Nava. Performance profiling and analysis of dod applications using papi and tau. In *2005 Users Group Conference (DOD-UGC'05)*, pages 394–399. IEEE, 2005.

[11] Nicholas A Nystrom, Michael J Levine, Ralph Z Roskies, and J Ray Scott. Bridges: a uniquely flexible hpc resource for new communities and data analytics. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, pages 1–8, 2015.

[12] Biswajit Saha. Green computing. *International Journal of Computer Trends and Technology (IJCTT)*, 14(2):46–50, 2014.

[13] Xianyi Zhang, Chao Yang, Fangfang Liu, Yiqun Liu, and Yutong Lu. Optimizing and scaling hpcg on tianhe-2: early experience. In *Algorithms and Architectures for Parallel Processing: 14th International Conference, ICA3PP 2014, Dalian, China, August 24-27, 2014. Proceedings, Part I 14*, pages 28–41. Springer, 2014.

chapterRegular Chapter

Appendices

A Perf Cache Hit Script

Listing 1: Bash Script for HPCG Power Collection

```
#!/bin/bash

#SBATCH -J hpcg_power_collection
#SBATCH -o hpcg_power_%j.out
#SBATCH -p parallel
#SBATCH -N 1

#SBATCH --ntasks=4
#SBATCH --cpus-per-task=1
#SBATCH -t 01:00:00
#SBATCH --exclusive

ml own perf
module load openmpi
module load cuda

# Start RAPL power collection
echo "Starting continuous power collection with RAPL read"
(
    echo "Timestamp,Package-0,Package-1,DRAM" > rapl_filtered_output_${SLURM_JOBID}.csv
    while true; do
        timestamp=$(date +%Y-%m-%d\ %H:%M:%S)
        ./rapl-read | grep -E 'package-[0-9]+|dram' | awk -v ts="$timestamp" 'BEGIN{ORS=""
END{print "\n"}' >> rapl_filtered_output_${SLURM_JOBID}.csv
        sleep 1
    done
) &
RAPL_PID=$!

echo "Starting perf"
PERF_FILE="perf_energy_output_${OMPI_COMM_WORLD_RANK}.txt"

echo "hello"
perf stat -e cache-misses,cache-references -I 1000 -o $PERF_FILE -- mpirun -np 4 ./bin/xhp
#mpirun -np 2 perf stat -e cache-misses,cache-references -I 1000 -o perf_energy_output_${O
echo "HPCG benchmark and power collection started."

wait

kill $RAPL_PID
echo "Continuous power collection stopped."

sleep 5

echo "HPCG benchmark and power collection complete."
```

B Rapl Power Collection Script

Listing 2: Bash Script for HPCG Benchmark and Power Collection

```
#!/bin/bash
#SBATCH -J hpcg_power_collection
#SBATCH -o hpcg_power_%j.out
#SBATCH -p a100
#SBATCH -N 1
#SBATCH --gres=gpu:a100:2
#SBATCH --ntasks=10
#SBATCH --cpus-per-task=1
#SBATCH -t 01:00:00
#SBATCH --exclusive # Request exclusive access to the node

# Load required modules
module load openmpi
module load cuda
# Load TAU module or setup TAU environment
module load tau

# Configure TAU to capture cache performance metrics
export TAU_METRICS="PAPI_L2_TCM,PAPI_L2_TCH"

echo "Starting continuous power collection with RAPL read"
(
    echo "Timestamp,Package-0,Package-1,DRAM" > rapl_filtered_output_${SLURM_JOBID}_${SLURM
    while true; do
        timestamp=$(date +%Y-%m-%d\ %H:%M:%S)
        ./rapl-read | grep -E 'package-[0-9]+|dram' | awk -v ts="$timestamp" 'BEGIN{ORS=""
END{print "\n"}' >> rapl_filtered_output_${SLURM_JOBID}_${SLURM_NODEID}.csv
        sleep 1
    done
) &
MONITOR_PID=$!

echo "Running HPCG benchmark with TAU instrumentation"
# Use TAU's mpirun wrapper or specify TAU options directly if necessary
# Assuming shared-mpi-openmp is the correct configuration to use
mpirun -np 1 tau_exec -T shared-mpi-openmp ./bin/xhpcg --nx=128 --ny=128 --nz=128

kill $MONITOR_PID
echo "Continuous power collection stopped."

sleep 5

echo "HPCG benchmark and power collection complete."
```

C Rapl Power Collection Script from University of Maine

Listing 3: Complete RAPL Read Source Code

```
/* Read the RAPL registers on recent (>sandybridge) Intel processors */
/* There are currently three ways to do this: */
/* 1. Read the MSRs directly with /dev/cpu/??/msr */
/* 2. Use the perf_event_open() interface */
/* 3. Read the values from the sysfs powercap interface */

/* MSR Code originally based on a (never made it upstream) linux-kernel */
/* RAPL driver by Zhang Rui <rui.zhang@intel.com> */
/* https://lkml.org/lkml/2011/5/26/93 */
/* Additional contributions by: */
/* Romain Dolbeau -- romain @ dolbeau.org */

/* For raw MSR access the /dev/cpu/??/msr driver must be enabled and */
/* permissions set to allow read access. */
/* You might need to "modprobe msr" before it will work. */

/* perf_event_open() support requires at least Linux 3.14 and to have */
/* /proc/sys/kernel/perf_event_paranoid < 1 */

/* the sysfs powercap interface got into the kernel in */
/* 2d281d8196e38dd (3.13) */

/* Compile with: gcc -O2 -Wall -o rapl-read rapl-read.c -lm */

/* Vince Weaver -- vincent.weaver @ maine.edu -- 11 September 2015 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <inttypes.h>
#include <unistd.h>
#include <math.h>
#include <string.h>

#include <sys/syscall.h>
#include <linux/perf_event.h>

#define MSR_RAPL_POWER_UNIT		0x606

/*
 * Platform specific RAPL Domains.
 * Note that PP1 RAPL Domain is supported on 062A only
 * And DRAM RAPL Domain is supported on 062D only
 */
/* Package RAPL Domain */
#define MSR_PKG_RAPL_POWER_LIMIT	0x610
#define MSR_PKG_ENERGY_STATUS		0x611
#define MSR_PKG_PERF_STATUS		0x613
#define MSR_PKG_POWER_INFO		0x614

/* PP0 RAPL Domain */
```

```
#define MSR_PP0_POWER_LIMIT             0x638
#define MSR_PP0_ENERGY_STATUS           0x639
#define MSR_PP0_POLICY                  0x63A
#define MSR_PP0_PERF_STATUS             0x63B

/* PP1 RAPL Domain, may reflect to uncore devices */
#define MSR_PP1_POWER_LIMIT             0x640
#define MSR_PP1_ENERGY_STATUS           0x641
#define MSR_PP1_POLICY                  0x642

/* DRAM RAPL Domain */
#define MSR_DRAM_POWER_LIMIT            0x618
#define MSR_DRAM_ENERGY_STATUS          0x619
#define MSR_DRAM_PERF_STATUS            0x61B
#define MSR_DRAM_POWER_INFO             0x61C

/* PSYS RAPL Domain */
#define MSR_PLATFORM_ENERGY_STATUS      0x64d

/* RAPL UNIT BITMASK */
#define POWER_UNIT_OFFSET       0
#define POWER_UNIT_MASK         0x0F

#define ENERGY_UNIT_OFFSET      0x08
#define ENERGY_UNIT_MASK        0x1F00

#define TIME_UNIT_OFFSET        0x10
#define TIME_UNIT_MASK          0xF000

static int open_msr(int core) {

        char msr_filename[BUFSIZ];
        int fd;

        sprintf(msr_filename, "/dev/cpu/%d/msr", core);
        fd = open(msr_filename, O_RDONLY);
        if ( fd < 0 ) {
                if ( errno == ENXIO ) {
                        fprintf(stderr, "rdmsr: No CPU %d\n", core);
                        exit(2);
                } else if ( errno == EIO ) {
                        fprintf(stderr, "rdmsr: CPU %d doesn't support MSRs\n",
                                        core);
                        exit(3);
                } else {
                        perror("rdmsr:open");
                        fprintf(stderr,"Trying to open %s\n",msr_filename);
                        exit(127);
                }
        }

        return fd;
}
```

```
static long long read_msr(int fd, int which) {

        uint64_t data;

        if ( pread(fd, &data, sizeof data, which) != sizeof data ) {
                perror("rdmsr:pread");
                exit(127);
        }

        return (long long)data;
}

#define CPU_SANDYBRIDGE         42
#define CPU_SANDYBRIDGE_EP      45
#define CPU_IVYBRIDGE           58
#define CPU_IVYBRIDGE_EP        62
#define CPU_HASWELL             60
#define CPU_HASWELL_ULT         69
#define CPU_HASWELL_GT3E        70
#define CPU_HASWELL_EP          63
#define CPU_BROADWELL           61
#define CPU_BROADWELL_GT3E      71
#define CPU_BROADWELL_EP        79
#define CPU_BROADWELL_DE        86
#define CPU_SKYLAKE             78
#define CPU_SKYLAKE_HS          94
#define CPU_SKYLAKE_X           85
#define CPU_KNIGHTS_LANDING     87
#define CPU_KNIGHTS_MILL        133
#define CPU_KABYLAKE_MOBILE     142
#define CPU_KABYLAKE            158
#define CPU_ATOM_SILVERMONT     55
#define CPU_ATOM_AIRMONT        76
#define CPU_ATOM_MERRIFIELD     74
#define CPU_ATOM_MOOREFIELD     90
#define CPU_ATOM_GOLDMONT       92
#define CPU_ATOM_GEMINI_LAKE    122
#define CPU_ATOM_DENVERTON      95

/* TODO: on Skylake, also may support  PSys "platform" domain,  */
/* the whole SoC not just the package.                          */
/* see dcee75b3b7f025cc6765e6c92ba0a4e59a4d25f4                 */

static int detect_cpu(void) {

        FILE *fff;

        int family,model=-1;
        char buffer[BUFSIZ],*result;
        char vendor[BUFSIZ];

        fff=fopen("/proc/cpuinfo","r");
        if (fff==NULL) return -1;
```

```
while(1) {
        result=fgets(buffer,BUFSIZ,fff);
        if (result==NULL) break;

        if (!strncmp(result,"vendor_id",8)) {
                sscanf(result,"%*s%*s%s",vendor);

                if (strncmp(vendor,"GenuineIntel",12)) {
                        printf("%s not an Intel chip\n",vendor);
                        return -1;
                }
        }

        if (!strncmp(result,"cpu family",10)) {
                sscanf(result,"%*s%*s%*s%d",&family);
                if (family!=6) {
                        printf("Wrong CPU family %d\n",family);
                        return -1;
                }
        }

        if (!strncmp(result,"model",5)) {
                sscanf(result,"%*s%*s%d",&model);
        }

}

fclose(fff);

printf("Found ");

switch(model) {
        case CPU_SANDYBRIDGE:
                printf("Sandybridge");
                break;
        case CPU_SANDYBRIDGE_EP:
                printf("Sandybridge-EP");
                break;
        case CPU_IVYBRIDGE:
                printf("Ivybridge");
                break;
        case CPU_IVYBRIDGE_EP:
                printf("Ivybridge-EP");
                break;
        case CPU_HASWELL:
        case CPU_HASWELL_ULT:
        case CPU_HASWELL_GT3E:
                printf("Haswell");
                break;
        case CPU_HASWELL_EP:
                printf("Haswell-EP");
                break;
        case CPU_BROADWELL:
        case CPU_BROADWELL_GT3E:
```

```
                        printf("Broadwell");
                        break;
                case CPU_BROADWELL_EP:
                        printf("Broadwell-EP");
                        break;
                case CPU_SKYLAKE:
                case CPU_SKYLAKE_HS:
                        printf("Skylake");
                        break;
                case CPU_SKYLAKE_X:
                        printf("Skylake-X");
                        break;
                case CPU_KABYLAKE:
                case CPU_KABYLAKE_MOBILE:
                        printf("Kaby Lake");
                        break;
                case CPU_KNIGHTS_LANDING:
                        printf("Knight's Landing");
                        break;
                case CPU_KNIGHTS_MILL:
                        printf("Knight's Mill");
                        break;
                case CPU_ATOM_GOLDMONT:
                case CPU_ATOM_GEMINI_LAKE:
                case CPU_ATOM_DENVERTON:
                        printf("Atom");
                        break;
                default:
                        printf("Unsupported model %d\n",model);
                        model=-1;
                        break;
        }

        printf(" Processor type\n");

        return model;
}

#define MAX_CPUS        1024
#define MAX_PACKAGES    16

static int total_cores=0,total_packages=0;
static int package_map[MAX_PACKAGES];

static int detect_packages(void) {

        char filename[BUFSIZ];
        FILE *fff;
        int package;
        int i;

        for(i=0;i<MAX_PACKAGES;i++) package_map[i]=-1;

        printf("\t");
```

```
        for(i=0;i<MAX_CPUS;i++) {
                sprintf(filename,"/sys/devices/system/cpu/cpu%d/topology/physical_package_i
                fff=fopen(filename,"r");
                if (fff==NULL) break;
                fscanf(fff,"%d",&package);
                printf("%d-(%d)",i,package);
                if (i%8==7) printf("\n\t"); else printf(",");
                fclose(fff);

                if (package_map[package]==-1) {
                        total_packages++;
                        package_map[package]=i;
                }

        }

        printf("\n");

        total_cores=i;

        printf("\tDetected %d cores in %d packages\n\n",
                total_cores,total_packages);

        return 0;
}


/*******************************/
/* MSR code                    */
/*******************************/
static int rapl_msr(int core, int cpu_model) {

        int fd;
        long long result;
        double power_units,time_units;
        double cpu_energy_units[MAX_PACKAGES],dram_energy_units[MAX_PACKAGES];
        double package_before[MAX_PACKAGES],package_after[MAX_PACKAGES];
        double pp0_before[MAX_PACKAGES],pp0_after[MAX_PACKAGES];
        double pp1_before[MAX_PACKAGES],pp1_after[MAX_PACKAGES];
        double dram_before[MAX_PACKAGES],dram_after[MAX_PACKAGES];
        double psys_before[MAX_PACKAGES],psys_after[MAX_PACKAGES];
        double thermal_spec_power,minimum_power,maximum_power,time_window;
        int j;

        int dram_avail=0,pp0_avail=0,pp1_avail=0,psys_avail=0;
        int different_units=0;

        printf("\nTrying /dev/msr interface to gather results\n\n");

        if (cpu_model<0) {
                printf("\tUnsupported CPU model %d\n",cpu_model);
                return -1;
        }
```

```
switch(cpu_model) {

        case CPU_SANDYBRIDGE_EP:
        case CPU_IVYBRIDGE_EP:
                pp0_avail=1;
                pp1_avail=0;
                dram_avail=1;
                different_units=0;
                psys_avail=0;
                break;

        case CPU_HASWELL_EP:
        case CPU_BROADWELL_EP:
        case CPU_SKYLAKE_X:
                pp0_avail=1;
                pp1_avail=0;
                dram_avail=1;
                different_units=1;
                psys_avail=0;
                break;

        case CPU_KNIGHTS_LANDING:
        case CPU_KNIGHTS_MILL:
                pp0_avail=0;
                pp1_avail=0;
                dram_avail=1;
                different_units=1;
                psys_avail=0;
                break;

        case CPU_SANDYBRIDGE:
        case CPU_IVYBRIDGE:
                pp0_avail=1;
                pp1_avail=1;
                dram_avail=0;
                different_units=0;
                psys_avail=0;
                break;

        case CPU_HASWELL:
        case CPU_HASWELL_ULT:
        case CPU_HASWELL_GT3E:
        case CPU_BROADWELL:
        case CPU_BROADWELL_GT3E:
        case CPU_ATOM_GOLDMONT:
        case CPU_ATOM_GEMINI_LAKE:
        case CPU_ATOM_DENVERTON:
                pp0_avail=1;
                pp1_avail=1;
                dram_avail=1;
                different_units=0;
                psys_avail=0;
                break;
```

```
                case CPU_SKYLAKE:
                case CPU_SKYLAKE_HS:
                case CPU_KABYLAKE:
                case CPU_KABYLAKE_MOBILE:
                        pp0_avail=1;
                        pp1_avail=1;
                        dram_avail=1;
                        different_units=0;
                        psys_avail=1;
                        break;

        }

        for(j=0;j<total_packages;j++) {
                printf("\tListing paramaters for package #%d\n",j);

                fd=open_msr(package_map[j]);

                /* Calculate the units used */
                result=read_msr(fd,MSR_RAPL_POWER_UNIT);

                power_units=pow(0.5,(double)(result&0xf));
                cpu_energy_units[j]=pow(0.5,(double)((result>>8)&0x1f));
                time_units=pow(0.5,(double)((result>>16)&0xf));

                /* On Haswell EP and Knights Landing */
                /* The DRAM units differ from the CPU ones */
                if (different_units) {
                        dram_energy_units[j]=pow(0.5,(double)16);
                        printf("DRAM: Using %lf instead of %lf\n",
                                dram_energy_units[j],cpu_energy_units[j]);
                }
                else {
                        dram_energy_units[j]=cpu_energy_units[j];
                }

                printf("\t\tPower units = %.3fW\n",power_units);
                printf("\t\tCPU Energy units = %.8fJ\n",cpu_energy_units[j]);
                printf("\t\tDRAM Energy units = %.8fJ\n",dram_energy_units[j]);
                printf("\t\tTime units = %.8fs\n",time_units);
                printf("\n");

                /* Show package power info */
                result=read_msr(fd,MSR_PKG_POWER_INFO);
                thermal_spec_power=power_units*(double)(result&0x7fff);
                printf("\t\tPackage thermal spec: %.3fW\n",thermal_spec_power);
                minimum_power=power_units*(double)((result>>16)&0x7fff);
                printf("\t\tPackage minimum power: %.3fW\n",minimum_power);
                maximum_power=power_units*(double)((result>>32)&0x7fff);
                printf("\t\tPackage maximum power: %.3fW\n",maximum_power);
                time_window=time_units*(double)((result>>48)&0x7fff);
                printf("\t\tPackage maximum time window: %.6fs\n",time_window);

                /* Show package power limit */
```

```
        result=read_msr(fd,MSR_PKG_RAPL_POWER_LIMIT);
        printf("\t\tPackage power limits are %s\n", (result >> 63) ? "locked" : "u
        double pkg_power_limit_1 = power_units*(double)((result>>0)&0x7FFF);
        double pkg_time_window_1 = time_units*(double)((result>>17)&0x007F);
        printf("\t\tPackage power limit #1: %.3fW for %.6fs (%s, %s)\n",
                pkg_power_limit_1, pkg_time_window_1,
                (result & (1LL<<15)) ? "enabled" : "disabled",
                (result & (1LL<<16)) ? "clamped" : "not_clamped");
        double pkg_power_limit_2 = power_units*(double)((result>>32)&0x7FFF);
        double pkg_time_window_2 = time_units*(double)((result>>49)&0x007F);
        printf("\t\tPackage power limit #2: %.3fW for %.6fs (%s, %s)\n",
                pkg_power_limit_2, pkg_time_window_2,
                (result & (1LL<<47)) ? "enabled" : "disabled",
                (result & (1LL<<48)) ? "clamped" : "not_clamped");

        /* only available on *Bridge-EP */
        if ((cpu_model==CPU_SANDYBRIDGE_EP) || (cpu_model==CPU_IVYBRIDGE_EP)) {
                result=read_msr(fd,MSR_PKG_PERF_STATUS);
                double acc_pkg_throttled_time=(double)result*time_units;
                printf("\tAccumulated Package Throttled Time : %.6fs\n",
                        acc_pkg_throttled_time);
        }

        /* only available on *Bridge-EP */
        if ((cpu_model==CPU_SANDYBRIDGE_EP) || (cpu_model==CPU_IVYBRIDGE_EP)) {
                result=read_msr(fd,MSR_PP0_PERF_STATUS);
                double acc_pp0_throttled_time=(double)result*time_units;
                printf("\tPowerPlane0 (core) Accumulated Throttled Time "
                        ": %.6fs\n",acc_pp0_throttled_time);

                result=read_msr(fd,MSR_PP0_POLICY);
                int pp0_policy=(int)result&0x001f;
                printf("\tPowerPlane0 (core) for core %d policy: %d\n",core,pp0_po

        }


        if (pp1_avail) {
                result=read_msr(fd,MSR_PP1_POLICY);
                int pp1_policy=(int)result&0x001f;
                printf("\tPowerPlane1 (on-core GPU if avail) %d policy: %d\n",
                        core,pp1_policy);
        }
        close(fd);

}
printf("\n");

for(j=0;j<total_packages;j++) {

        fd=open_msr(package_map[j]);

        /* Package Energy */
        result=read_msr(fd,MSR_PKG_ENERGY_STATUS);
```

```
        package_before[j]=(double)result*cpu_energy_units[j];

        /* PP0 energy */
        /* Not available on Knights* */
        /* Always returns zero on Haswell-EP? */
        if (pp0_avail) {
                result=read_msr(fd,MSR_PP0_ENERGY_STATUS);
                pp0_before[j]=(double)result*cpu_energy_units[j];
        }

        /* PP1 energy */
        /* not available on *Bridge-EP */
        if (pp1_avail) {
                result=read_msr(fd,MSR_PP1_ENERGY_STATUS);
                pp1_before[j]=(double)result*cpu_energy_units[j];
        }


        /* Updated documentation (but not the Vol3B) says Haswell and   */
        /* Broadwell have DRAM support too                              */
        if (dram_avail) {
                result=read_msr(fd,MSR_DRAM_ENERGY_STATUS);
                dram_before[j]=(double)result*dram_energy_units[j];
        }


        /* Skylake and newer for Psys                                   */
        if ((cpu_model==CPU_SKYLAKE) ||
                (cpu_model==CPU_SKYLAKE_HS) ||
                (cpu_model==CPU_KABYLAKE) ||
                (cpu_model==CPU_KABYLAKE_MOBILE)) {

                result=read_msr(fd,MSR_PLATFORM_ENERGY_STATUS);
                psys_before[j]=(double)result*cpu_energy_units[j];
        }

        close(fd);
}

printf("\n\tSleeping 1 second\n\n");
sleep(1);

for(j=0;j<total_packages;j++) {

        fd=open_msr(package_map[j]);

        printf("\tPackage %d:\n",j);

        result=read_msr(fd,MSR_PKG_ENERGY_STATUS);
        package_after[j]=(double)result*cpu_energy_units[j];
        printf("\t\tPackage energy: %.6fJ\n",
                package_after[j]-package_before[j]);

        result=read_msr(fd,MSR_PP0_ENERGY_STATUS);
```

```
                pp0_after[j]=(double)result*cpu_energy_units[j];
                printf("\t\tPowerPlane0 (cores): %.6fJ\n",
                        pp0_after[j]-pp0_before[j]);

                /* not available on SandyBridge-EP */
                if (pp1_avail) {
                        result=read_msr(fd,MSR_PP1_ENERGY_STATUS);
                        pp1_after[j]=(double)result*cpu_energy_units[j];
                        printf("\t\tPowerPlane1 (on-core GPU if avail): %.6f J\n",
                                pp1_after[j]-pp1_before[j]);
                }

                if (dram_avail) {
                        result=read_msr(fd,MSR_DRAM_ENERGY_STATUS);
                        dram_after[j]=(double)result*dram_energy_units[j];
                        printf("\t\tDRAM: %.6fJ\n",
                                dram_after[j]-dram_before[j]);
                }

                if (psys_avail) {
                        result=read_msr(fd,MSR_PLATFORM_ENERGY_STATUS);
                        psys_after[j]=(double)result*cpu_energy_units[j];
                        printf("\t\tPSYS: %.6fJ\n",
                                psys_after[j]-psys_before[j]);
                }

                close(fd);
        }
        printf("\n");
        printf("Note: the energy measurements can overflow in 60s or so\n");
        printf("      so try to sample the counters more often than that.\n\n");

        return 0;
}

static int perf_event_open(struct perf_event_attr *hw_event_uptr,
                    pid_t pid, int cpu, int group_fd, unsigned long flags) {

        return syscall(__NR_perf_event_open,hw_event_uptr, pid, cpu,
                        group_fd, flags);
}

#define NUM_RAPL_DOMAINS        5

char rapl_domain_names[NUM_RAPL_DOMAINS][30]= {
        "energy-cores",
        "energy-gpu",
        "energy-pkg",
        "energy-ram",
        "energy-psys",
};


static int check_paranoid(void) {
```

```
        int paranoid_value;
        FILE *fff;

        fff=fopen("/proc/sys/kernel/perf_event_paranoid","r");
        if (fff==NULL) {
                fprintf(stderr,"Error! could not open /proc/sys/kernel/perf_event_paranoid
                        strerror(errno));

                /* We can't return a negative value as that implies no paranoia */
                return 500;
        }

        fscanf(fff,"%d",&paranoid_value);
        fclose(fff);

        return paranoid_value;

}

static int rapl_perf(int core) {

        FILE *fff;
        int type;
        int config[NUM_RAPL_DOMAINS];
        char units[NUM_RAPL_DOMAINS][BUFSIZ];
        char filename[BUFSIZ];
        int fd[NUM_RAPL_DOMAINS][MAX_PACKAGES];
        double scale[NUM_RAPL_DOMAINS];
        struct perf_event_attr attr;
        long long value;
        int i,j;
        int paranoid_value;

        printf("\nTrying perf_event interface to gather results\n\n");

        fff=fopen("/sys/bus/event_source/devices/power/type","r");
        if (fff==NULL) {
                printf("\tNo perf_event rapl support found (requires Linux 3.14)\n");
                printf("\tFalling back to raw msr support\n\n");
                return -1;
        }
        fscanf(fff,"%d",&type);
        fclose(fff);

        for(i=0;i<NUM_RAPL_DOMAINS;i++) {

                sprintf(filename,"/sys/bus/event_source/devices/power/events/%s",
                        rapl_domain_names[i]);

                fff=fopen(filename,"r");

                if (fff!=NULL) {
                        fscanf(fff,"event=%x",&config[i]);
```

```
                printf("\tEvent=%s Config=%d ",rapl_domain_names[i],config[i]);
                fclose(fff);
        } else {
                continue;
        }

        sprintf(filename,"/sys/bus/event_source/devices/power/events/%s.scale",
                rapl_domain_names[i]);
        fff=fopen(filename,"r");

        if (fff!=NULL) {
                fscanf(fff,"%lf",&scale[i]);
                printf("scale=%g ",scale[i]);
                fclose(fff);
        }

        sprintf(filename,"/sys/bus/event_source/devices/power/events/%s.unit",
                rapl_domain_names[i]);
        fff=fopen(filename,"r");

        if (fff!=NULL) {
                fscanf(fff,"%s",units[i]);
                printf("units=%s ",units[i]);
                fclose(fff);
        }

        printf("\n");
}

for(j=0;j<total_packages;j++) {

        for(i=0;i<NUM_RAPL_DOMAINS;i++) {

                fd[i][j]=-1;

                memset(&attr,0x0,sizeof(attr));
                attr.type=type;
                attr.config=config[i];
                if (config[i]==0) continue;

                fd[i][j]=perf_event_open(&attr,-1, package_map[j],-1,0);
                if (fd[i][j]<0) {
                        if (errno==EACCES) {
                                paranoid_value=check_paranoid();
                                if (paranoid_value>0) {
                                        printf("\t/proc/sys/kernel/perf_event_paran
                                        printf("\tThe value must be 0 or lower to
                                }

                                printf("\tPermission denied; run as root or adjust
                                return -1;
                        }
                        else {
                                printf("\terror opening core %d config %d: %s\n\n"
```

```
                                                package_map[j], config[i], strerror(errno)
                                        return -1;
                                }
                        }
                }
        }

        printf("\n\tSleeping 1 second\n\n");
        sleep(1);

        for(j=0;j<total_packages;j++) {
                printf("\tPackage %d:\n",j);

                for(i=0;i<NUM_RAPL_DOMAINS;i++) {

                        if (fd[i][j]!=-1) {
                                read(fd[i][j],&value,8);
                                close(fd[i][j]);

                                printf("\t\t%s Energy Consumed: %lf %s\n",
                                        rapl_domain_names[i],
                                        (double)value*scale[i],
                                        units[i]);

                        }

                }

        }
        printf("\n");

        return 0;
}

static int rapl_sysfs(int core) {

        char event_names[MAX_PACKAGES][NUM_RAPL_DOMAINS][256];
        char filenames[MAX_PACKAGES][NUM_RAPL_DOMAINS][256];
        char basename[MAX_PACKAGES][256];
        char tempfile[256];
        long long before[MAX_PACKAGES][NUM_RAPL_DOMAINS];
        long long after[MAX_PACKAGES][NUM_RAPL_DOMAINS];
        int valid[MAX_PACKAGES][NUM_RAPL_DOMAINS];
        int i,j;
        FILE *fff;

        printf("\nTrying sysfs powercap interface to gather results\n\n");

        /* /sys/class/powercap/intel-rapl/intel-rapl:0/ */
        /* name has name */
        /* energy_uj has energy */
        /* subdirectories intel-rapl:0:0 intel-rapl:0:1 intel-rapl:0:2 */

        for(j=0;j<total_packages;j++) {
```

```
        i=0;
        sprintf(basename[j],"/sys/class/powercap/intel-rapl/intel-rapl:%d",
                j);
        sprintf(tempfile,"%s/name",basename[j]);
        fff=fopen(tempfile,"r");
        if (fff==NULL) {
                fprintf(stderr,"\tCould not open %s\n",tempfile);
                return -1;
        }
        fscanf(fff,"%s",event_names[j][i]);
        valid[j][i]=1;
        fclose(fff);
        sprintf(filenames[j][i],"%s/energy_uj",basename[j]);

        /* Handle subdomains */
        for(i=1;i<NUM_RAPL_DOMAINS;i++) {
                sprintf(tempfile,"%s/intel-rapl:%d:%d/name",
                        basename[j],j,i-1);
                fff=fopen(tempfile,"r");
                if (fff==NULL) {
                        //fprintf(stderr,"\tCould not open %s\n",tempfile);
                        valid[j][i]=0;
                        continue;
                }
                valid[j][i]=1;
                fscanf(fff,"%s",event_names[j][i]);
                fclose(fff);
                sprintf(filenames[j][i],"%s/intel-rapl:%d:%d/energy_uj",
                        basename[j],j,i-1);

        }
}

/* Gather before values */
for(j=0;j<total_packages;j++) {
        for(i=0;i<NUM_RAPL_DOMAINS;i++) {
                if (valid[j][i]) {
                        fff=fopen(filenames[j][i],"r");
                        if (fff==NULL) {
                                fprintf(stderr,"\tError opening %s!\n",filenames[j
                        }
                        else {
                                fscanf(fff,"%lld",&before[j][i]);
                                fclose(fff);
                        }
                }
        }
}

printf("\tSleeping 1 second\n\n");
sleep(1);

/* Gather after values */
for(j=0;j<total_packages;j++) {
```

```
                for(i=0;i<NUM_RAPL_DOMAINS;i++) {
                        if (valid[j][i]) {
                                fff=fopen(filenames[j][i],"r");
                                if (fff==NULL) {
                                        fprintf(stderr,"\tError opening %s!\n",filenames[j
                                }
                                else {
                                        fscanf(fff,"%lld",&after[j][i]);
                                        fclose(fff);
                                }
                        }
                }
        }

        for(j=0;j<total_packages;j++) {
                printf("\tPackage %d\n",j);
                for(i=0;i<NUM_RAPL_DOMAINS;i++) {
                        if (valid[j][i]) {
                                printf("\t\t%s\t: %lfJ\n",event_names[j][i],
                                ((double)after[j][i]-(double)before[j][i])/1000000
                        }
                }
        }
        printf("\n");

        return 0;

}

int main(int argc, char **argv) {

        int c;
        int force_msr=0,force_perf_event=0,force_sysfs=0;
        int core=0;
        int result=-1;
        int cpu_model;

        printf("\n");
        printf("RAPL read -- use -s for sysfs, -p for perf_event, -m for msr\n\n");

        opterr=0;

        while ((c = getopt (argc, argv, "c:hmps")) != -1) {
                switch (c) {
                case 'c':
                        core = atoi(optarg);
                        break;
                case 'h':
                        printf("Usage: %s [-c core] [-h] [-m]\n\n",argv[0]);
                        printf("\t-c core : specifies which core to measure\n");
                        printf("\t-h ----- : displays this help\n");
                        printf("\t-m ----- : forces use of MSR mode\n");
                        printf("\t-p ----- : forces use of perf_event mode\n");
                        printf("\t-s ----- : forces use of sysfs mode\n");
```

```
                exit(0);
            case 'm':
                force_msr = 1;
                break;
            case 'p':
                force_perf_event = 1;
                break;
            case 's':
                force_sysfs = 1;
                break;
            default:
                fprintf(stderr,"Unknown option %c\n",c);
                exit(-1);
            }
        }

        (void)force_sysfs;

        cpu_model=detect_cpu();
        detect_packages();

        if ((!force_msr) && (!force_perf_event)) {
            result=rapl_sysfs(core);
        }

        if (result<0) {
            if ((force_perf_event) && (!force_msr)) {
                result=rapl_perf(core);
            }
        }

        if (result<0) {
            result=rapl_msr(core,cpu_model);
        }

        if (result<0) {

            printf("Unable to read RAPL counters.\n");
            printf("* Verify you have an Intel Sandybridge or newer processor\n");
            printf("* You may need to run as root or have /proc/sys/kernel/perf_event_p
            printf("* If using raw msr access, make sure msr module is installed\n");
            printf("\n");

            return -1;

        }

        return 0;
}
```