HeteroBench: Multi-kernel Benchmarks for Heterogeneous Systems

Hongzheng Tian
hongzhet@uci.edu
University of California, Irvine
Irvine, CA, USA
Hewlett Packard Enterprise
Milpitas, CA, USA

Alok Mishra
alok.mishra@hpe.com
Hewlett Packard Enterprise
Milpitas, CA, USA

Zhiheng Chen
zhihenc5@uci.edu
University of California, Irvine
Irvine, CA, USA

Rolando P. Hong Enriquez
rhong@hpe.com
Hewlett Packard Enterprise
Milpitas, CA, USA

Dejan Milojicic
dejan.milojicic@hpe.com
Hewlett Packard Enterprise
Milpitas, CA, USA

Eitan Frachtenberg
eitan.frachtenberg@hpe.com
Hewlett Packard Enterprise
Milpitas, CA, USA

Sitao Huang
sitaoh@uci.edu
University of California, Irvine
Irvine, CA, USA

Abstract

The end of Moore's Law and Dennard scaling has driven the proliferation of heterogeneous systems with accelerators, including CPUs, GPUs, and FPGAs, each with distinct architectures, compilers, and programming environments. GPUs excel at massively parallel processing for tasks like deep learning training and graphics rendering, while FPGAs offer hardware-level flexibility and energy efficiency for low-latency, high-throughput applications. In contrast, CPUs, while general-purpose, often fall short in high-parallelism or power-constrained applications. This architectural diversity makes it challenging to compare these accelerators effectively, leading to uncertainty in selecting optimal hardware and software tools for specific applications.

To address this challenge, we introduce HeteroBench, a versatile benchmark suite for heterogeneous systems. HeteroBench allows users to evaluate multi-compute kernel applications across various accelerators, including CPUs, GPUs (from NVIDIA, AMD, Intel), and FPGAs (AMD), supporting programming environments of Python, Numba-accelerated Python, serial C++, OpenMP (both CPUs and GPUs), OpenACC and CUDA for GPUs, and Vitis HLS for FPGAs. This setup enables users to assign kernels to suitable hardware platforms, ensuring comprehensive device comparisons.

What makes HeteroBench unique is its vendor-agnostic, cross-platform approach, spanning diverse domains such as image processing, machine learning, numerical computation, and physical simulation, ensuring deeper insights for HPC optimization. Extensive testing across multiple systems provides practical reference points for HPC practitioners, simplifying hardware selection and performance tuning for both developers and end-users alike. This suite may assist to make more informed decision on AI/ML deployment and HPC development, making it an invaluable resource for advancing academic research and industrial applications.

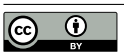
Keywords

Benchmark suite, heterogeneous computing, Python, OpenMP, CUDA, OpenACC, HLS, High Performance Computing, CPU, GPU, FPGA

ACM Reference Format:
Hongzheng Tian, Alok Mishra, Zhiheng Chen, Rolando P. Hong Enriquez, Dejan Milojicic, Eitan Frachtenberg, and Sitao Huang. 2025. HeteroBench: Multi-kernel Benchmarks for Heterogeneous Systems. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE '25), May 5–9, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3676151.3719366

1 Introduction

With the increasing complexity of computational tasks in Artificial Intelligence (AI), Machine Learning (ML) and High Performance Computing (HPC), there is a growing need for scalable heterogeneous computing systems that can efficiently manage diverse workloads. These systems typically comprise numerous computing units, including high-performance multi-core processors (CPUs), graphics processing units (GPUs), and reconfigurable hardware accelerators like FPGAs. Compared to systems with a single resource, heterogeneous systems offer several advantages, such as improved computational and power efficiency, better resource utilization, and enhanced flexibility in handling diverse workloads [29, 31]. One significant advantage of heterogeneous systems is their ability to optimize specific tasks by leveraging the strengths of different types of accelerators. For instance, GPUs are highly efficient for parallel processing tasks achieving 25× speedup over CPUs [26], while FPGAs offer customizable hardware that is particularly advantageous for applications requiring real-time processing, such as financial



ICPE '25, Toronto, ON, Canada

ACM ISBN 979-8-4007-1073-5/2025/05
https://doi.org/10.1145/3676151.3719366

algorithms [17], network processing [13] and have better power efficiency than GPUs [32]. Distributing workloads across different types of accelerators balances performance, improves power efficiency, and supports diverse applications from ML to scientific simulations, leading to more sustainable computing practices.

Given these benefits, heterogeneous computing systems are gaining increasing attention from both academics and the HPC industry. However, to fully utilize the potential of heterogeneous systems, it is crucial to have a robust and comprehensive benchmark suite that can evaluate their performance across various applications. Such a suite should not only assess system performance but also aids researchers design and optimize compilers and runtime systems for heterogeneous platforms.

Despite the numerous benchmark suites designed for heterogeneous platforms, many have become outdated with the evolution of computing systems. Most existing benchmark suites (discussed in Section 2) lack sufficient heterogeneity in supported accelerators and programming languages. Python, despite its popularity in AI/ML, is often excluded from HPC benchmarks due to increased performance overhead and less precise hardware control than compiled languages like C++. However, Python's inclusion is critical for AI/ML developers working on several platforms, where performance evaluation and system optimization are critical. This lack of support restricts developers' ability to evaluate performance accurately and optimize systems effectively.

Python is the preferred language for AI/ML development due to its simplicity and robust ecosystem, with libraries like TensorFlow and PyTorch. These libraries leverage C++ backends for computationally intensive tasks, allowing developers to write in Python while taking advantage of C++'s performance. However, many users – primarily data scientists and application researchers rather than advanced computer engineers – often miss out on full performance potential, as Python's slower execution can become a bottleneck. This slowdown is mainly due to Python's interpreted nature, which incurs significant execution overhead. Without the support of third-party libraries implemented in lower-level languages, pure Python execution becomes much less efficient for computationally intensive tasks.

A tool that translates performance-critical Python code to optimized C++ could bridge the gap, allowing users to keep Python's simplicity of use while achieving C++-level speed. To support the development of such a tool, a benchmark suite that analyzes Python and C++ performance on heterogeneous platforms, such as CPUs, GPUs, and FPGAs, is essential.

To address this need, we present **HeteroBench**[1], a versatile benchmark suite designed for heterogeneous HPC systems, incorporating several multi-compute kernel benchmark applications, in Python and C++ to execute across various hardware platforms efficiently. HeteroBench contains the following features:

- Multi-language support to meet diverse developer needs. It features several multi-kernel benchmark applications in various versions: standard Python, Numba-accelerated Python, standard serial C++, OpenMP-enhanced C++ for both CPUs and GPUs, OpenACC-enhanced C++ for GPUs, CUDA for NVIDIA GPUs, and Vitis HLS-enhanced C++ for FPGAs.
- Compatibility with a variety of accelerators for comprehensive performance evaluations, including CPUs, GPUs (from NVIDIA, AMD, and Intel), and FPGAs (from AMD Xilinx).
- User-friendly high-level parallel programming (Python, OpenMP, OpenACC directive) for enhanced accessibility and adaptability.
- Multi-kernel design guides users to customize the placement across different hardware platforms, facilitating performance evaluation and optimization for heterogeneous systems.
- Extensive testing across various platforms validated the effectiveness and versatility of HeteroBench, showcasing performance characteristics of various hardware configurations for researchers and practitioners in heterogeneous computing systems.

The rest of this paper is organized as follows. Section 2 reviews existing benchmark suites for heterogeneous systems. Section 3 gives the overview concepts and the selection of each benchmark application. We introduce HeteroBench implementation designs in Section 4 and show evaluation results in Section 5. In Section 6 we discuss our design choices and finally, we conclude in Section 7.

2 Related Work

As heterogeneous platforms have evolved, numerous benchmark suites have emerged; however, many have become outdated or unsuitable for today's integrated CPU-GPU-FPGA systems. We selected some representative benchmark suites and summarized them in TABLE 1.

2.1 Heterogeneous Benchmark Suites with Single-type Accelerators

One of the earliest and most referenced benchmark suites is UC Berkeley's 13 Dwarfs [4], introduced when GPU-based platforms were still emerging. While it provided valuable insights into various computational patterns, such as dense and sparse linear algebra or spectral methods, it falls short of meeting the demands of

[1] GitHub: https://github.com/HewlettPackard/HeteroBench/

Criteria	13 Dwarfs	Polybench	MachSuite	Rosetta	Chai	Rodinia	HosNa	HeteroBench
Multi-kernel Applications	✓	×	✓	✓	✓	✓	✓	✓
CPU Implementation	✓	✓	×	✓	✓	✓	✓	✓
GPU Implementation	✓	✓	×	×	✓	✓	✓	✓
FPGA Implementation	×	×	✓	✓	×	✓	✓	✓
Python Implementation	×	✓	×	×	×	×	×	✓
Multiple-brand GPUs*	×	×	×	×	×	×	×	✓

*(NVIDIA, AMD, Intel)

Table 1: Comparison of Selected Different Benchmark Suites.

modern HPC systems. Today's workloads require more specialized benchmarks that reflect the growing importance of AI/ML workloads, hybrid computing paradigms, and large-scale distributed systems. Additionally, newer platforms introduce heterogeneous architectures that include AI-focused GPUs and tightly integrated FPGA solutions, which were not considered in the original design of 13 Dwarfs. Consequently, this benchmark suite is insufficient for evaluating the performance and scalability of contemporary heterogeneous systems.

Most existing heterogeneous benchmark suites focus on systems with **single type of accelerators**, such as GPUs or FPGAs. For instance, benchmark suites like [11, 12, 18, 21, 28, 34] are designed primarily for GPU-based heterogeneous systems, typically utilizing CUDA or OpenCL, which require explicit memory allocation and data management. This constraint limits users' ability to customize benchmarks and restricts hardware platform compatibility, particularly between NVIDIA, AMD, and Intel GPUs. However, using standard C++ code with OpenMP [10]`pragmas` can enable parallel acceleration on CPUs and GPUs (from NVIDIA, AMD, or Intel) through NVIDIA CUDA [22], AMD ROCm [1] or Intel OneAPI [16] runtimes.

Similarly, FPGA-focused benchmark suites such as MachSuite [25] and Rosetta [39] primarily evaluate FPGA performance using high-level synthesis (HLS) C/C++ implementations. MachSuite includes 19 applications targeting only FPGA platforms, while Rosetta provides software versions that can also run on CPUs. However, the CPU version of Rosetta serve merely as references without any performance optimization, limiting their utility for thorough performance evaluation.

2.2 Heterogeneous Benchmark Suites with Multi-type Accelerators

Some benchmark suites extend beyond a single accelerator type, aiming to evaluate systems with **multiple types of accelerators** (e.g., CPUs, GPUs, and FPGAs). Rodinia [7] and Chai [12] primarily target CPU-GPU heterogeneous systems and have been adapted for FPGAs [9, 14]. However, these adaptations frequently require considerable code rewriting, making them challenging to replicate in other benchmark suites.

Other benchmark suites, such as HosNa [5], explicitly support CPUs, GPUs, and FPGAs but restrict hardware compatibility to Intel-exclusive platforms. This limits their applicability to broader heterogeneous computing research, which requires evaluating multiple hardware vendors.

A key limitation of many multi-accelerator benchmark suites is the lack of flexibility in **assigning compute kernels to different hardware accelerators**. Most frameworks assume that the entire workload will be executed on a single accelerator rather than allowing different kernels to be mapped to different accelerators.

2.3 Role of Python in Benchmark Development

Given Python's growing popularity among AI/ML practitioners, it is essential to include Python implementations in benchmark suites. Python's popularity has soared due to its ease of use and extensive libraries, making it a go-to language for many AI/ML applications, and as computational demands increase, there is a growing need to deploy Python applications on HPC systems or any other heterogeneous architecture. However, there is a significant lack of comprehensive compilation workflows that support this deployment, which hinders the effective utilization of Python in these contexts.

Despite many proposed compilation workflows, as previously mentioned, most are limited to targeting either GPUs [30] or FPGAs [8, 15, 38] individually. There is a critical need for a versatile compilation process that can handle Python applications and deploy them across a heterogeneous system that includes both GPUs and FPGAs. Benchmark suites with Python implementations will be crucial for compiler developers, providing necessary testing and validation to ensure that Python applications can be effectively compiled and optimized for execution on heterogeneous architectures that integrate multiple types of accelerators, including both GPUs and FPGAs.

Polybench [24] is another popular benchmark suite that has undergone several adaptations for GPUs and FPGAs. However, when considering deployment on heterogeneous systems that include both GPUs and FPGAs, several challenges issues arise. Polybench primarily consists of benchmarks with a single compute kernel, posing challenges when deploying multiple computing platforms for task-level parallelism or pipelining. It requires manual compute kernel partitioning, which is a substantial workload. Therefore, an ideal heterogeneous benchmark suite should encompass applications with multiple compute kernels, allowing for more efficient deployment and optimization.

3 The HeteroBench Suite Description

As discussed in Section 2, conventional benchmark suites often fail to address the intricate needs of heterogeneous HPC systems. Therefore, we purposely designed HeteroBench with the following key features:

- **Versatile Compatibility with Minimal Configuration**: Easily deployable on most systems, requiring minimal setup and supporting a wide range of GPU brands.
- **Flexible Accelerator Assignment**: Provides seamless configuration options to leverage GPUs, FPGAs, or parallelized CPUs for acceleration, adapting to available hardware resources.
- **Kernel-level Customization**: Each benchmark supports multiple computational kernels, enabling users to select and optimize kernels for various hardware backends based on their specific performance needs.
- **Standardized Kernel Algorithms for Fair Comparison**: Maintains consistent computational algorithms across all versions, ensuring fair, apples-to-apples comparisons and reliable performance benchmarking across different platforms.

Overall, the HeteroBench suite is designed to address the limitations of existing benchmarks by supporting diverse hardware configurations and offering flexible customization. It aims to facilitate the development and optimization of heterogeneous HPC systems, ensuring the efficient and unbiased evaluation of modern workloads across various platforms.

HeteroBench currently includes eleven benchmarks across four domains: image processing, artificial intelligence, numerical computation, and physical simulation, designed to expand over time

Benchmarks (abbreviation)	# of Compute Kernels	Application Domain
Canny Edge Detection (ced)	5	Image Processing
Sobel Filter (sbf)	3	
Optical Flow (opf)	8	
Convolutional neural Network (cnn)	5	Artificial Intelligence
Multi-layer Perceptron (mlp)	3	
Digit Recognition (dgr)	2	
One Head Attention (oha)	3	
Spam Filter (spf)	4	
3 Matrix Multiplications (3mm)	3	Numerical Computing
Alternating Direction Implicit (adi)	2	
Parallelize Particle (ppc)	2	Physical Simulation

Table 2: The benchmarks included in the HeteroBench suite. Abbreviations will be used in the rest of the paper. A brief description of each benchmark is listed in Appendices A.

as new benchmarks are developed and incorporated into the suite. The list of all benchmarks in the HeteroBench benchmark suite is presented in TABLE 2. Each benchmark comes with a baseline Python version. Additionally, a Python Numba [19] version is provided, that leverages just-in-time compilation (JIT) through Numba for improved performance. We also provide a standard sequential C++ version that complements the baseline Python code, as well as accelerated C++ versions that utilize OpenMP and HLS to accelerate computationally intensive parts (i.e., for loops) for execution on CPUs, GPUs, or FPGAs.

To showcase OpenMP performance on GPUs, we implemented both CUDA and OpenACC code, highlighting the differences between these parallelization methods on the same hardware. For a fair, apples-to-apples comparison, we took special care to ensure that the computational kernel algorithms remain consistent across all versions, minimizing major algorithmic differences regarding application initialization, file I/O, array definition, initialization, and similar aspects.

The HeteroBench architecture overview is presented in Figure 1. A top-level Python script administers the benchmark suite, allowing for smooth execution on a variety of computing platforms. Users are only required to customize two json configuration files. The first is an environment-related configuration file (`env_config.json`) that contains settings for various hardware platforms, such as the necessary compilers, compilation flags, and linked libraries. Default settings are provided for ease of use. The second is a benchmark-related configuration file (`env_config.json`) that specifies input/output data, tunable variables, and the target hardware for each computational kernel. These files allow users to customize execution without altering source code.

Benchmark execution is initiated via a command-line interface, where users specify the benchmark name, desired operation, target backend, and optional parameters. Figure 1 illustrates the execution flow using Canny edge detection as an example, deploying the CUDA version onto an NVIDIA GPU. Upon receiving a user command, the management script sequentially selects the requested benchmark, identifies the appropriate code version, compiles it using the designated hardware environment, and executes it on the

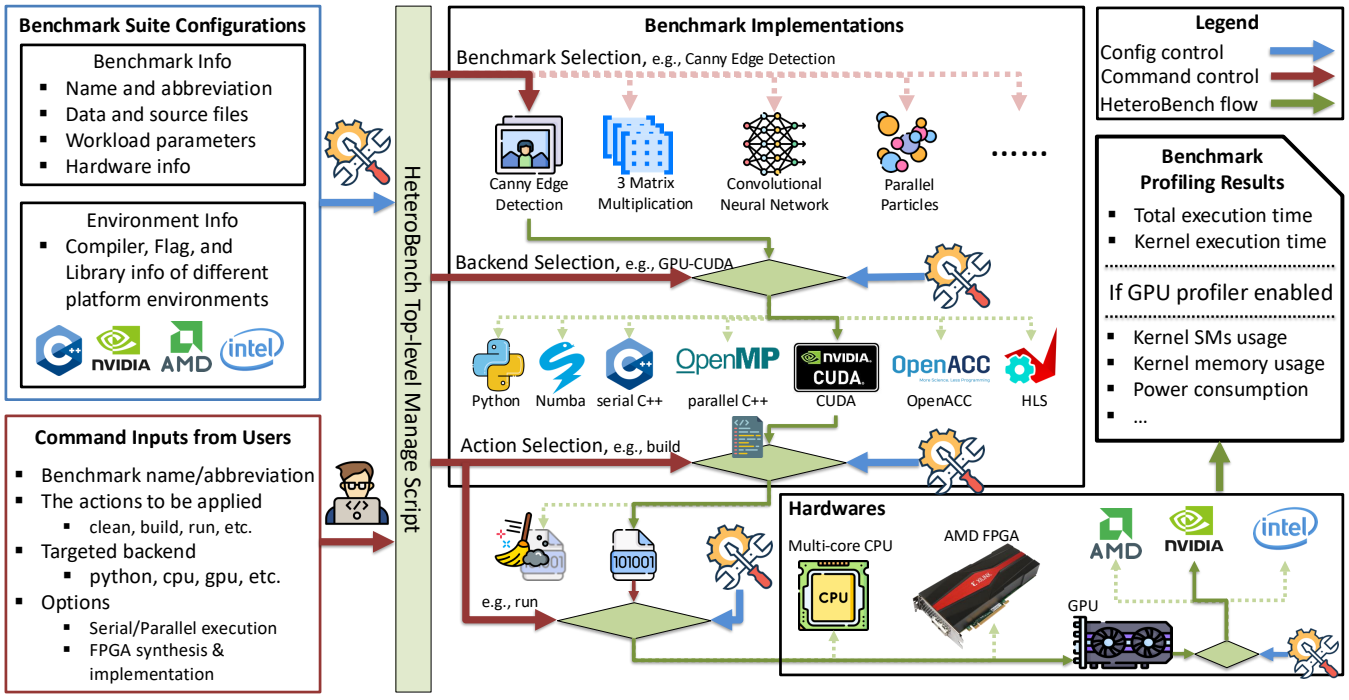


Figure 1: The overview of HeteroBench suite architecture.

target platform. Finally, the profiling results are collected and returned to the user. This automated workflow enables a streamlined benchmarking process with a single command.

4 Design and Implementation

Our benchmark suite distinguishes itself from most others by providing multiple versions of the same kernel in a multi-kernel application in different programming languages to support heterogeneous architectures. For GPUs, we chose OpenMP and OpenACC to ensure code consistency across code versions, enabling fair and consistent comparisons. Unlike CUDA and OpenCL, which require substantial code modifications to optimize for GPUs, both OpenMP and OpenACC use compiler directives, resulting in simpler and faster development, especially when working across multiple GPU vendors. Their high-level abstractions can help handle GPU parallelism without getting entangled in vendor-specific details, which can be a considerable advantage in the heterogeneous computing environment. They are best suited for scientific applications and environments that require portability and ease of usage.

However, while OpenMP and OpenACC provide valuable abstractions for GPU programming, they can restrict users' access to low-level optimizations, which may affect performance on some GPU workloads. To provide a more comprehensive assessment, we implemented CUDA versions of all benchmarks, serving as a critical performance reference. This approach enables a direct comparison of OpenMP with GPU-specific programming on identical hardware, offering insights into the potential performance trade-offs, and helping users make informed decisions about the best programming approach for their needs.

In addition, by providing Python versions, we make our benchmarks accessible to a broader audience, enabling AI/ML developers to evaluate performance within a familiar environment. Additionally, the Python versions serve as a baseline to highlight the performance improvements achieved through various optimizations and parallelization techniques. This also supports compiler designers in developing and testing tools that target Python for heterogeneous platforms, facilitating advancements in Python performance on modern hardware.

4.1 Python and Serial C++ Code Design

The design of the baseline Python and standard serial C++ code forms the foundation of HeteroBench, ensuring that each benchmark operates correctly before parallelization and hardware acceleration are applied. For benchmarks without standard golden results, we use output from the baseline Python or serial C++ code as a reference.

While Python offers extensive computational libraries, we manually implemented key computations to maintain consistency across Python and C++ versions. This approach ensures robustness, maintainability, and extensibility while enabling fair and reliable performance comparisons across programming environments. These Python applications serve as our baseline for all experiments.

The C++ implementation mirrors the structure and flow of the Python code, maintaining consistency across versions. Our C++ design emphasizes modular, configurable components that allow for easy integration and minimal modification across different hardware backends. While some benchmarks are adapted from existing works, as elaborated in Appendices A.3, A.6, A.8, and A.10, we split kernels into individual files rather than combining them into a single project file. This approach allows users to select and target specific kernel implementations for heterogeneous execution with ease.

4.2 Parallel C++ Code Design

The parallel C++ code design leverages multi-threading to enhance the performance on CPU and GPU platforms. We begin by identifying the computationally intensive sections of the serial C++ code, focusing on loops and function calls that can benefit the most from parallelization. Using OpenMP and OpenACC `pragmas`, we annotate these sections to enable parallel execution. We choose OpenMP and OpenACC for their simplicity and widespread support on different hardware, allowing us to achieve significant performance gains with minimal code modifications.

To illustrate the modification from python implementation to a serial C++ loop to a parallelized loop using OpenMP and OpenACC `pragmas`, consider the examples shown in Table 3. Other codes (like OpenMP target offloading, CUDA and HLS) are not represented here to save space.

```python
for i in range(N):
    for j in range(M):
        result[i][j] = process(data[i][j])
```

Listing 1: Python for loop

```cpp
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < M; ++j) {
    result[i][j] = process(data[i][j]);
}}
```

Listing 2: Serial C++ for loop

```cpp
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < M; ++j) {
    result[i][j] = process(data[i][j]);
}}
```

Listing 3: Parallel C++ for loop with OpenMP

```cpp
#pragma acc data copyin(data[0:N][0:M])
#pragma acc parallel loop collapse(2)
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < M; ++j) {
    result[i][j] = process(data[i][j]);
}}
#pragma acc data copyout(result[0:N][0:M])
```

Listing 4: GPU offloading with OpenACC

Table 3: Comparison of Python, C++ Serial, OpenMP, and OpenACC Implementations

In the serial implementation in Listings 1 and 2, the `process` function processes each element of the `data` array sequentially. In contrast, in the parallel version shown in Listing 3 and Listing 4, the

`omp parallel for` and `acc parallel loop` directives instruct the OpenMP and OpenACC runtimes to parallelize the execution of the nested loops. The `collapse(2)` clause merges the two loops into a single iteration space, allowing both OpenMP and OpenACC to distribute iterations across multiple threads more effectively.

For GPU code, we need to use the `omp declare target` or `acc routine` directives in OpenMP and OpenACC, respectively, to indicate that the `process` function should be compiled for execution on the device. Data management directives are also needed to transfer data between the host and the device, as shown in Listing 4.

However, achieving the ideal speedup depends on several factors, especially in a more complex situation (detailed discussion and analysis in Section 5), including:

- **Thread Management Overhead**: There is a cost associated with creating and managing threads.
- **Load Balancing**: The workload must be evenly distributed among all threads to prevent situations where some threads finish much earlier than others.
- **Memory Bandwidth**: Multiple threads accessing memory simultaneously can lead to bandwidth saturation, limiting speedup.
- **Cache Coherence and Contention**: Threads might contend for cache resources, leading to potential performance degradation.
- **Synchronization Overhead**: Any synchronization required between threads can introduce delays.

The parallel C++ code design aims to maximize computational throughput, reduce execution time, and maintain the accuracy and robustness of the benchmarks. While OpenMP and OpenACC are used for parallelization across both CPU and GPU, users must have the appropriate environments set up, such as NVIDIA's CUDA [22], AMD's ROCm [1], or Intel's oneAPI [16], depending on their specific hardware. This ensures that the benchmarks can fully leverage the capabilities of the underlying hardware, ensuring optimal performance. Without the correct environment, users may encounter compatibility issues or fail to take advantage of the parallel processing power available, resulting in failed or suboptimal execution or less efficient benchmarking results.

4.3 CUDA Code Design

In contrast to the OpenMP/OpenACC implementation, the CUDA code explicitly includes memory allocation and data transfer operations, which are necessary for moving data between the host (CPU) and the device (GPU). These steps involve allocating device memory using `cudaMalloc`, transferring data with `cudaMemcpy`, and ensuring proper synchronization before and after kernel execution. CUDA codes are generally longer due to the added complexities of detailed control over memory allocation, explicit data movements, thread management, synchronization, and the explicit configuration of threads and blocks.

Despite these additions, the computational kernels in the CUDA implementation remain consistent with those in the OpenMP C++ code. We have carefully structured the CUDA kernels to match the logic and flow of the OpenMP/OpenACC version, ensuring that the computation remains identical. This approach maintains fairness in the performance comparisons, as any observed differences in execution time can be attributed to the underlying parallelization model and hardware acceleration capabilities rather than code structure or logic variations.

4.4 HLS C++ Code Design

As mentioned in the previous sections, our goal is to keep the algorithm of C++ code consistent across CPUs, GPUs, and FPGAs while making minimal code changes. However, the Vitis HLS [36] implementation without specific adaption can be significantly slow (see Section 5) due to the following reasons associated with the direct `directives` insertion:

- **Inefficient Memory Access:** The HLS C++ framework treats I/O pointer arguments as master Advanced eXtensible Interface (AXI) [36] interfaces, which necessitates that every array access goes off-chip to High Bandwidth Memory (HBM) [2], leading to inefficient data transfer between kernels, compounded by the interference of the Xilinx Runtime Library APIs [35] and under-utilization of the on-chip Block Random Access Memory (BRAM).
- **Lack of Task-Level Parallelism:** Separate kernels do not leverage the FPGA's advantage in task-level parallelism. This results in a purely sequential execution of different kernels, for which low clock frequency-driven FPGAs are not competent.
- **Variable loop boundary:** Suggested by the HLS user guide [3], unlike the general C++ that takes loop boundaries as kernel augment, variable loop boundaries for kernel reusing. This can be solved by replacing it with static loop iterations.

The limitations above indicate an under-utilization of the FPGA's acceleration potential, which causes some benchmarks to not be synthesizable or executed correctly on the FPGA board. Consequently, we manually rewrote these benchmarks from different computation patterns to make them work properly and optimize their FPGA execution flow.

For example, the image processing benchmark `opf` produces and consumes data in a strict sequential order. On-chip buffers and FIFOs are necessary to achieve the sequential pattern. Thus, we utilize efficient unified buffers for the kernels to access both row and column elements sequentially.

The `dgr` benchmark was noted in previous research [39] for its high compute-to-communication ratio. Each test instance requires 10 to 1000 cycles for calculating the Hamming distance and performing KNN voting. Additionally, training samples and labels are stored on-chip and reused for all test instances, establishing it as a compute-bound application. Consequently, the HLS optimizations applied to this benchmark include array partition, loop pipelining, and unrolling.

For the time-consuming `3mm` benchmark, we consider the pragma `dataflow`, to run the code in task-level parallel of the first two matrix multiplications. The inputs are bound into different ports to enable the `dataflow`. Moreover, the variable loop boundaries in matrix multiplication are converted into static iterations.

4.5 HeteroBench Top Design

The HeteroBench suite is managed by a top-level Python script that automates benchmark compilation and execution. This script abstracts the complexity of heterogeneous computing environments, ensuring seamless integration across diverse hardware platforms.

As mentioned in Section 3, the script loads the environment and benchmark configuration files, which define hardware settings, compiler options, and execution parameters. Based on these configurations, it dynamically generates the necessary build and execution commands.

In addition, to maintain portability and flexibility, the script leverages the Jinja [23] template engine to dynamically generate Makefiles based on the provided configuration parameters. Instead of relying on static Makefiles for each benchmark, the template-driven approach allows the system to customize compilation flags, target architectures, and library dependencies at runtime. Users who prefer manual compilation and execution can directly use the generated Makefiles to build and run specific benchmarks, allowing for greater customization.

During execution, the script generates a benchmark-specific Makefile, injecting hardware-specific parameters. It then invokes the appropriate `make` targets using Python's `subprocess` module, ensuring consistent compilation across different platforms.

This templated Makefile approach also simplifies adding new benchmarks. Users only need to update the configuration file and provide a Jinja-based Makefile template, eliminating extensive manual setup and making the suite easier to extend.

5 Evaluation and Results

All environment variable settings are included in the environment-configuration file, which can be passed to the top-level Python script to generate the benchmark's Makefiles. This file contains essential environment variables, including various compiler information (e.g., `clang++`, `nvc++`, `icpx`), C++ flags, OpenMP/OpenACC libraries, AMD Xilinx toolkit-related paths, and other necessary configurations. We have pre-configured all variables to minimize the need for user modifications. This setup allows users to run the benchmarks with minimal effort, leveraging the predefined settings tailored for common environments. If a user's actual environment differs from our presets, only a small portion of the configuration file needs to be adjusted, ensuring a convenient and straightforward setup process.

For different types of GPUs, we assume that users have already installed the corresponding toolkits, such as CUDA, ROCm, and OneAPI. For FPGA, we assume that users have installed the AMD Xilinx development tools and have access to the necessary development boards.

We have conducted tests of our benchmark suite on a total of four servers, each equipped with different CPUs and GPUs. To ensure a fair comparison, we present the CPU benchmark results only from the AMD EPYC 7713 64-Core Processor. This includes results for Python, Python-Numba, serial C++, and parallel C++ enhanced by OpenMP. For GPUs, we tested on NVIDIA A100 80GB, AMD MI210, and Intel Data Center GPU Max 1100. For FPGA, we collected the data from AMD Xilinx U280.

By standardizing the environment setup and providing flexible configuration options, we facilitate a consistent and reproducible evaluation process for various hardware platforms. To account for initialization overhead, we perform a warm-up iteration before measuring execution time. Each benchmark is then executed 20 times, and the average runtime is reported. This approach helps mitigate variability caused by transient system states. For a detailed analysis of runtime fluctuations, refer to Section 5.3.

5.1 Lines of Code Comparison

The number of lines of code (LOC) in various programming languages often indicates their ease of use and the level of abstraction available. Languages, like Python, that require fewer lines of code to achieve the same functionality, typically provide higher-level abstractions, built-in functions, and simpler syntax. This simplifies the process of writing and maintaining code, making these languages more accessible even to non-software engineers.

	python	numba	serial_c++	cpu_omp	gpu_omp	gpu_acc	cuda	hls_c++
ced	233	239	351	356	382	381	534	704
opf	285	294	533	541	630	630	831	808
sbf	125	129	210	216	231	231	312	407
cnn	287	294	432	444	414	414	659	919
mlp	229	232	369	374	463	466	576	1077
oha	255	262	401	411	450	450	596	685
dgr	199	199	230	239	265	262	335	391
spf	230	235	391	394	414	414	497	680
3mm	119	123	297	298	309	309	374	439
adi	88	147	253	261	262	263	340	458
ppc	275	183	618	620	686	687	733	350

Table 4: Number of lines of code for all benchmarks.

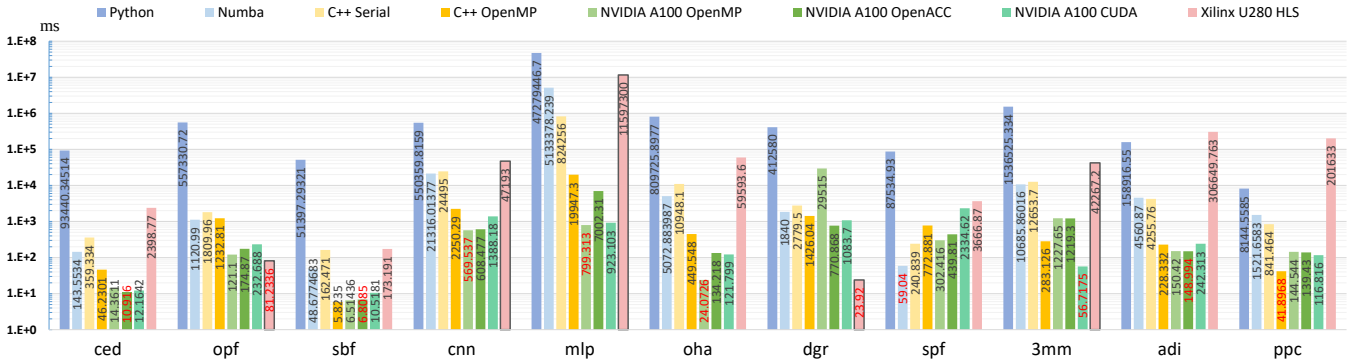
TABLE 4 displays the total number of lines of code written for each benchmark across all code versions. As mentioned before, to ensure a fair comparison, we avoided using any existing computational libraries, such as `NumPy`, in the Python code's computational kernels. Despite this, the Python code still has significantly fewer lines than the C++ code, while the FPGA code consistently requires the most lines for the same algorithm due to its lower-level nature and requirement for explicit hardware control.

5.2 Profiling Results

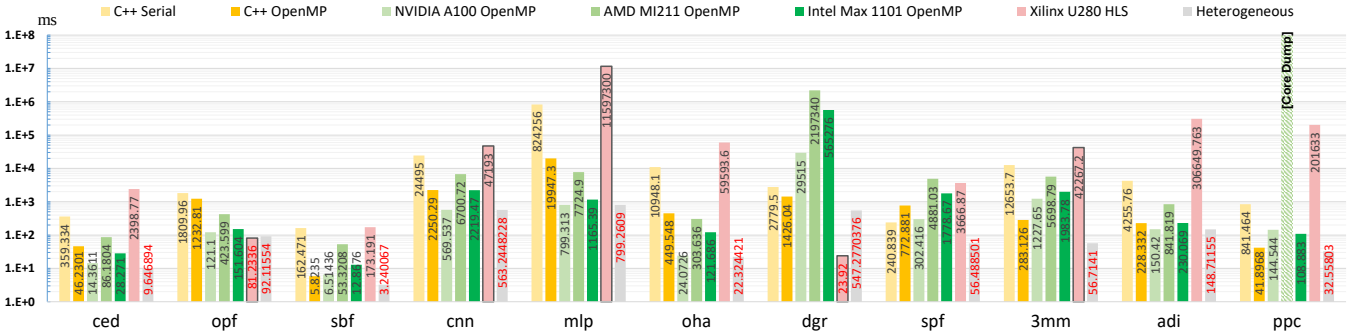
5.2.1 Compare Across Different Code Versions. In Figure. 2a, the results show the overall runtime of a single iteration for each benchmark in different code versions. All the tests presented here are homogeneous, meaning that all the kernels in a given benchmark run on the same hardware. For instance, the bar labeled "NVIDIA A100 OpenMP" in `opf` represents that all kernels in `opf` are accelerated by OpenMP and executed on the NVIDIA A100 GPU.

It is evident that, except for `adi` and `ppc`, the Python versions of most benchmarks have the longest execution times. This outcome is expected because, to ensure fair comparisons across different compilation environments and hardware platforms, we maintained consistency in the code and did not use any existing computational libraries, such as `NumPy`, in the Python implementations. As a result, the Python code takes significantly longer to execute, highlighting the limitations of Python as an interpreter-based language. Without optimizations provided by third-party libraries (often implemented in lower-level languages like C/C++), Python's performance lags substantially behind.

Using `Numba` to accelerate the Python code brings substantial improvements. `Numba` compiles Python code to machine code at runtime, leveraging performance gains typically associated with



(a) Total execution time comparison across different languages for each benchmark. Data for Python, Numba, C++ serial, and C++ OpenMP were collected on MD EPYC 7713 64-Core Processor.



(b) Total execution time comparison across different devices for each benchmark. Data for CPU serial and CPU OpenMP were collected on MD EPYC 7713 64-Core Processor. The heterogeneous version was created after testing the homogeneous versions by selecting the fastest platform for each kernel and combining them to form the final version.

Figure 2: The total execution time comparison in the *Log10* scale for each benchmark in a single iteration. The fastest one is highlighted in red. The non-solid bar in ppc is due to execution errors on AMD MI210. The bars with a solid border for Xilinx U280 indicate they are adapted/optimized to ensure proper FPGA execution.

compiled languages. This optimization reduces execution times significantly (e.g., in ced, sbf, and spf), making the Numba-accelerated versions competitive with other implementations.

In both serial and parallel CPU implementations, performance improves significantly across most benchmarks. The parallel CPU version employs OpenMP directives to efficiently distribute workloads across multiple cores, minimizing bottlenecks and idle time. Interestingly, in the ppc benchmark, the parallel CPU version outperforms all GPU versions, due to the workload's compatibility with CPU architectures, where thread management overhead is reduced, and memory access patterns are more efficiently handled.

However, using OpenMP does not always guarantee better performance, as seen in spf in Figure. 2a. The sigmoid kernel in this benchmark performs scalar operations and is not parallelized in this comparison. When parallel optimizations are applied to the CPU, we observe that performance can degrade compared to serial execution. This is likely due to task granularity: the amount of work done per iteration is too small, making the overhead of parallelization more significant. Creating and managing threads for each loop iteration adds overhead, and the lightweight operations inside the loop do not compensate for this overhead.

Regarding GPU results, intuitively, CUDA versions should be the fastest as they offer more low-level control, allowing finer-grained GPU code optimizations. However, based on our results, CUDA is not always the fastest. For instance, in mlp, the OpenMP version is the fastest, while in dgr, it performs significantly slower than the other two versions. In ced, OpenACC is the fastest, but it lags behind the other versions in mlp. In 3mm, however, the CUDA version vastly outperforms the other implementations. This may be due to vendor-specific optimizations for OpenMP and OpenACC, which vary across different hardware platforms.

GPUs are not always the best option for all benchmarks. Compared to fully optimized FPGA code (e.g., opf and dgr), even the NVIDIA A100 cannot achieve the same level of performance. For other computationally intensive benchmarks, HLS implementations typically show longer execution times, particularly in ced, cnn, mlp, 3mm, adi, and ppc. As discussed in Section 4.4, achieving satisfactory performance on FPGAs using pragma-based optimizations alone is challenging without extensive optimizations for each application. For instance, the opf benchmark could not be run correctly on FPGA with pragma-based optimization alone, requiring substantial individual optimization efforts. Therefore, our optimized opf

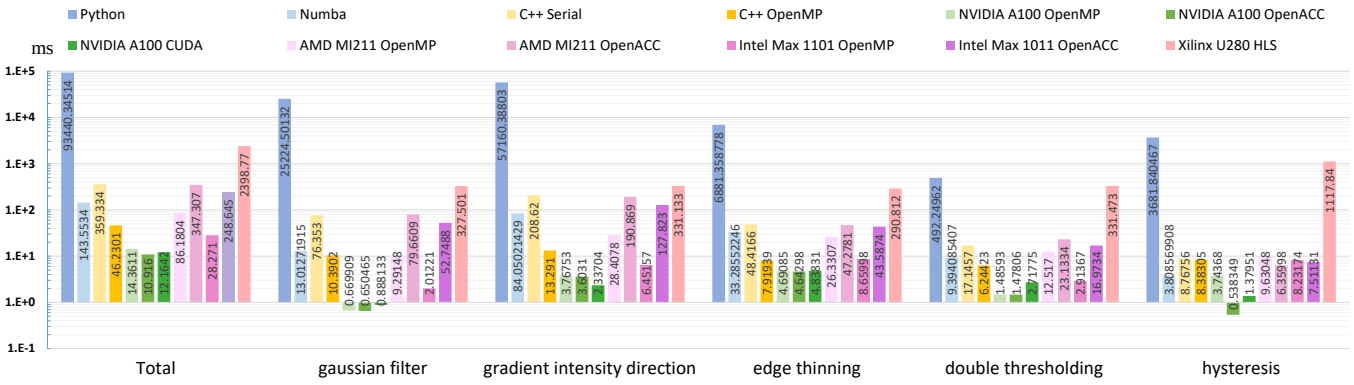


Figure 3: The execution time comparison of ced in the *Log10* scale for each kernel in a single iteration.

and dgr are the only two benchmarks where the FPGA versions outperform all others.

5.2.2 Compare Across Different Hardware Devices. In Figure. 2b, we focus on comparing the total execution time for each benchmark across different hardware devices using the same code (OpenMP-enhanced) for CPUs, various GPUs, and the heterogeneous version. The heterogeneous version was created after testing the homogeneous versions by selecting the fastest platform for each kernel and combining them into the final version.

Notably, the heterogeneous versions don't have any benchmark kernel to run on an FPGA. As discussed earlier, we aimed to maintain code consistency across all versions, and under these conditions, the performance of HLS implementations is suboptimal. Assigning a single kernel to run on an FPGA within a benchmark would degrade the overall performance rather than enhance it. Thus, we opted to exclude FPGAs from the heterogeneous configuration to ensure a fair and meaningful comparison of performance.

In some benchmarks, such as dgr, 3mm, and ppc, the parallel CPU implementation performs surprisingly well, even surpassing all GPU platforms. This could be due to the nature of the workload, which may favor memory access patterns and reduce the overhead of thread management on the CPU, allowing it to outperform the GPUs in these specific cases.

As for the GPU results, the NVIDIA A100 GPU generally performs the best among the three GPUs in most benchmarks, except for the ppc benchmark, where Intel Max 1101 is nearly 25% faster than the NVIDIA A100. Notably, the AMD MI210 consistently underperforms compared to the other two GPUs in almost every test, and it even encounters a core dump during the execution of ppc. Since we used the same code and applied the same optimization pragmas across all platforms, this suggests that AMD ROCm's OpenMP compilation optimizations lag behind those of NVIDIA and Intel.

Moreover, the most notable result is that the heterogeneous version is the fastest across all benchmarks (except for fully optimized FPGA design like opf and dgr). This is expected, as the heterogeneous version combines the best kernels from different devices, resulting in standout overall performance, further validating the effectiveness of our heterogeneous approach. More detailed results will be presented in the section 5.2.3.

Overall, the analysis in Figure. 2b highlights that no single hardware platform is universally the best choice. Performance varies significantly based on the computational characteristics of each benchmark and the level of optimizations applied to the hardware. While GPUs generally show strong performance, they are not always the best option, particularly when a task involves both serial and parallel kernels. The heterogeneous version showcases the advantages of selectively assigning kernels to the hardware best suited for a given task, achieving optimal overall performance.

In Section 5.2.1, we discussed that on the NVIDIA A100 GPU, the fastest code versions for the ced, mlp, and 3mm benchmarks are OpenACC, OpenMP, and CUDA, respectively. Now, we provide a detailed breakdown of the execution time for individual kernels within these benchmarks as examples.

5.2.3 Profiling Details. As shown in Figure 3, OpenACC indeed achieves the shortest total runtime on the NVIDIA A100, followed by CUDA and then OpenMP. However, on the other two GPU platforms (AMD and Intel), OpenMP outperforms OpenACC with a noticeable margin. When analyzing the first four kernels (which exhibit high degrees of parallelism), the performance gap between OpenMP and OpenACC on the NVIDIA A100 is minimal. This suggests that AMD and Intel GPUs are better optimized for OpenMP in highly parallel workloads, while NVIDIA offers comparable optimization for both OpenMP and OpenACC.

For CUDA, performance on the NVIDIA GPU varies depending on the kernel parallelism. For instance, CUDA outperforms the other two versions in the highly parallel `gradient intensity direction` kernel but falls behind in the less parallel `double thresholding` kernel. This variability reflects the strengths of CUDA for fine-tuned, low-level parallel optimizations, which depend heavily on the parallel structure of the code.

A particularly interesting case is the `hysteresis` kernel. Since this kernel has little to no parallelism, it theoretically should run serially. However, for the sake of code consistency, we still applied parallel pragmas to it. The results are surprising. On the CPU, C++ OpenMP shows almost no speedup over the serial version, and on AMD and Intel GPUs, OpenMP offers no improvement or even slows down the execution. This outcome aligns with expectations, as compilers are generally designed to ignore parallel directives for

non-parallel workloads. Similarly, OpenACC demonstrates minimal speedup on AMD and Intel GPUs.

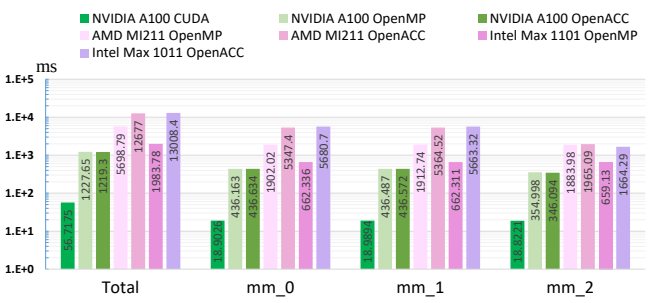


Figure 4: The execution time comparison of `3mm` in the *Log10* scale for each kernel in a single iteration.

Unexpectedly, the NVIDIA GPU delivers excellent performance for all three code versions, even for this low-parallelism kernel. In particular, the OpenACC version achieves up to a 15x speedup. This is also why the OpenACC version is the fastest in terms of total time comparison. One possible reason could be that the `nvc++` compiler is better optimized for managing such low-parallelism code segments, efficiently offloading them to GPU cores. Another explanation could involve NVIDIA's architecture handling lightweight tasks more gracefully, minimizing the overhead associated with launching parallel threads. This unique behavior highlights NVIDIA's superior support for various types of workloads, even those with limited parallelism, compared to other vendors.

Another example, shown in Figure 4, is the `3mm` benchmark, which further supports our earlier observations. Each kernel in `3mm` exhibits a high degree of parallelism. Similar to the `ced` example, the performance difference between OpenACC and OpenMP on the NVIDIA A100 GPU is minimal. However, on AMD and Intel GPUs, OpenMP outperforms OpenACC. The high level of parallelism also enables CUDA to perform exceptionally well, achieving double the speed of the other two implementations on the same platform.

Figure 5 presents the results of testing the heterogeneous implementation on another server featuring an Intel Core 13900K and an NVIDIA RTX 3090, chosen specifically to evaluate the portability of our benchmark suite across different hardware setups. The benchmark results follow a similar pattern to those in Figure 2, though the NVIDIA A100 in Figure 2 consistently performed better than the heterogeneous implementation on the RTX 3090. By selecting a platform with a smaller performance gap between the CPU and GPU, we aimed to highlight the advantages of heterogeneous execution better.

Our analysis of the homogeneous versions revealed that the `gradient_xy_calc` kernel performs optimally on the GPU, while other kernels execute more efficiently on the CPU. Following this insight, we deployed these eight kernels between the CPU and GPU accordingly, opting to use the OpenMP version on the GPU for simplicity, as it demonstrated minimal performance difference compared to OpenACC or CUDA. As shown in the third group of Figure 5, this approach achieved a final runtime of 175.325 ms, which is faster than the two homogeneous versions, demonstrating the advantage of effective heterogeneous execution.

However, heterogeneous designs do not consistently outperform homogeneous configurations. If kernels are assigned to hardware accelerators without careful consideration, or if hardware changes occur within the system, the overall runtime may increase significantly, as illustrated in the fourth and fifth groups in Figure 5. In the worst case, the final result can even be up to 40% slower than the C++ OpenMP implementation running solely on the CPU. This result highlights the significance of selective kernel placement and the requirement for system-specific tailoring to achieve optimal performance. Effective heterogeneous execution relies on tailored configurations to fully leverage hardware capabilities.

5.3 Variability Analysis

To demonstrate some of the deeper insights that can be gleaned from HeteroBench, we analyzed the variability in performance measured across several combinations of kernels and platforms. Figure 6 shows the distributions of kernel run times as standard box plots. We can make several observations from these data:

- **Long-tailed distributions:** Most run times are long-tailed, meaning the bulk of the measurements are concentrated around the median while several outliers pull the mean performance to the right, significantly affecting the overall mean. This indicates that typical runs are "normal" or "fast", centered in a narrow band around the median, but any introduction of delay can cause a noticeable slowdown. This behavior represents both a challenge

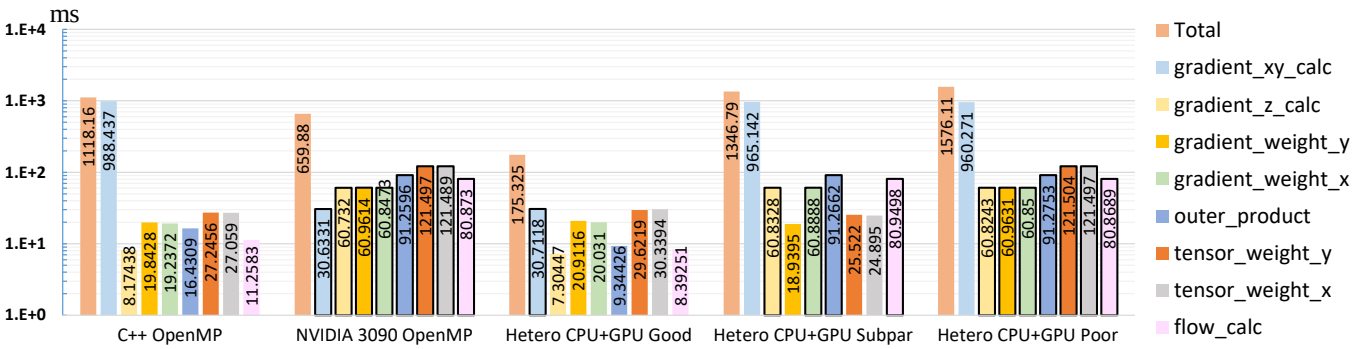


Figure 5: The execution time comparison between homogeneous and heterogeneous implementations of `opf` in the *Log10* scale for each kernel in a single iteration on NVIDIA RTX 3090. The bars with solid borders denote the kernels run on GPU.

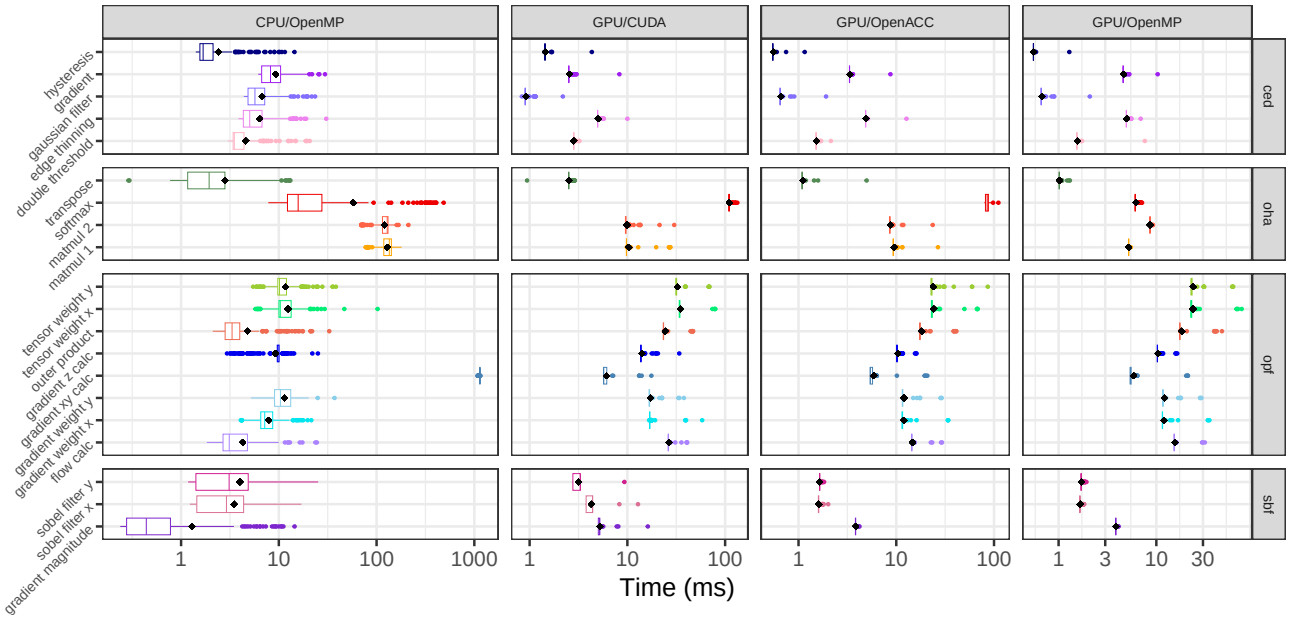


Figure 6: Run time distributions as box plots across four benchmarks with multiple kernels and four platforms (n=200, log-scale). Vertical bars represent 25^{th}, 50^{th}, and 75^{th} percentiles while black diamonds represent mean performance.

and an opportunity. One slow kernel can adversely affect the entire system's performance, while debugging the most egregious outliers can lead to the diagnosing of system or software improvements that could reduce both run time and its unpredictability.

- **CPU vs. GPU runtimes:** CPU run times for these kernels are not only significantly slower than GPU, as we had noted before, but their variability is also much higher. Some CPU-based kernels such as `gradient_magnitude` exhibit outliers that are orders of magnitude slower than the typical (median) performance, which is uncommon for their GPU counterparts. All GPU distributions appear more clustered and symmetric than most CPU distributions, with the mean and the median nearly identical. The reason for this difference is likely the complexity of the CPU environment, which is shared by other processes and managed by a mercurial operating system, leading to many more potential slowdown events, as noted before.
- **OpenACC performance:** Among the GPU frameworks, OpenACC appears to offer the lowest variability and smallest outliers (keeping in mind the logarithmic scale of the x-axis). In environments where performance predictability is as important as performance magnitude, this could be an important consideration when picking a framework.
- **Kernel variability:** Some kernels exhibit much higher variability than others, especially on CPUs (e.g., `softmax` and `matmul` kernels within the same `oha` application). This heterogeneity may again be interrelated with the different compute resources used by each kernel and how sensitive they are to interference and noise, offering another clue towards optimization.

This analysis highlights the variability in kernel runtimes and the need for understanding performance characteristics, allowing developers to optimize applications for better performance and efficiency across heterogeneous computing environments.

6 Discussion

This section addresses some aspects that may require further clarification based on our design choices and the motivation for developing HeteroBench.

6.1 Motivation for Developing HeteroBench

The primary motivation for designing HeteroBench is to address the lack of comprehensive benchmark suites for heterogeneous HPC systems. Most existing benchmarks are designed for homogeneous platforms—either focusing on GPUs, FPGAs, or CPUs in combination with one of the accelerators. Very few existing suites offer a unified approach that evaluates all three types of devices, making it difficult for researchers and developers to assess and optimize workloads for truly heterogeneous systems.

Even in cases where heterogeneous systems are supported, the entire application is typically executed on a single accelerator without the flexibility to distribute tasks across multiple devices. In scenarios where users want to deploy an application across multiple accelerators, they must manually partition the workload and assign individual tasks to specific devices, a process that is often labor-intensive and complex, and significantly increase development effort and limit the practical usability of heterogeneous platforms. To enable comprehensive and efficient benchmarking, it is essential to provide a solution that supports flexible task distribution across multiple hardware components.

To address these challenges, HeteroBench deliberately excludes single-kernel benchmarks, such as SPMV or K-means, commonly

found in traditional benchmark suites. These benchmarks pose significant challenges for task partitioning due to their single-kernel structure. Instead, it features multi-kernel benchmarks, enabling users to explore efficient task partitioning, workload balancing, and heterogeneous execution strategies. By offering a comprehensive and adaptable benchmark suite, HeteroBench empowers researchers to develop, evaluate, and optimize scalable computing solutions for heterogeneous architectures, fostering innovation and efficiency in high-performance computing.

6.2 Ensuring Code Consistency and Fair Comparisons

Another key aspect of our design is the inclusion of multiple versions for each benchmark, with consistent code structure across versions, apart from necessary pragma-based optimizations. This consistency minimizes performance differences caused by code variations and ensures that performance comparisons reflect the actual impact of different compilers, programming models, and hardware devices. By standardizing the codebase across different versions, we create a fair and reliable foundation for evaluating heterogeneous computing systems.

A common challenge faced by many existing benchmark suites is their tendency to optimize code specifically for a single accelerator to maximize performance. While such optimizations can yield impressive results on individual devices, they introduce bias when comparing performance across multiple platforms. They raise a fundamental question: does the observed performance improvement result from superior hand-tuned code design, compiler optimizations, or better hardware itself? Without a uniform structure, distinguishing between these factors becomes difficult, leading to potentially misleading conclusions.

For instance, to evaluate the efficiency of multiple processors, a reliable approach is to run identical instruction sets rather than tailoring workloads to each processor's strengths. In line with this principle, HeteroBench ensures that all benchmark versions maintain a consistent core structure, allowing for fair, meaningful, and unbiased performance comparisons across a variety of devices and programming models. This approach enhances the reliability of benchmarking results and supports objective assessments in heterogeneous computing environments.

6.3 Balancing Portability and Optimization

To ensure portability across a wide range of computing platforms, HeteroBench primarily relies on OpenMP, a widely adopted API that enables parallelization on both CPUs and GPUs with minimal code changes. OpenMP provides a high-level abstraction, making it easier for developers to write parallel code that can seamlessly run on different hardware architectures without requiring extensive rewrites.

In addition to OpenMP, we also provide CUDA versions for all benchmarks to explore the full performance potential achievable with NVIDIA's low-level programming model. To further diversify our benchmark suite and assess alternative GPU programming models, we have also integrated OpenACC versions to further diversify the suite and examine other GPU programming paradigms. OpenACC provides a directive-based approach, similar to OpenMP, but is specifically designed for offloading computations to GPUs with minimal programming complexity.

Looking ahead, we plan to extend HeteroBench by incorporating additional APIs such as HIP (for AMD GPUs) and OpenCL (for broader device support across AMD, Intel, and other vendors). This expansion will further enhance the portability, flexibility and versatility of HeteroBench, making it a valuable tool for evaluating modern heterogeneous computing systems across a wide spectrum of hardware architectures.

7 Conclusion and Future Work

In this paper, we introduced HeteroBench, a comprehensive heterogeneous benchmark suite comprising several multi-kernel benchmarks, designed to evaluate performance across multiple hardware backends, including CPUs, GPUs, and FPGAs. Unlike traditional benchmark suites that focus on homogeneous architectures or a single type of accelerator, HeteroBench is built to support applications utilizing diverse compute kernels, enabling more realistic performance assessments for diverse workloads. By providing implementations in both Python and C++, HeteroBench accommodates a wide range of users, from researchers exploring high-level algorithmic optimizations to developers fine-tuning low-level hardware performance.

Extensive tests on multiple servers with varying hardware configurations yield precise profiling results that demonstrate HeteroBench's effectiveness in assessing performance in heterogeneous computing systems. HeteroBench offers valuable insights for HPC professionals, enabling the testing, tuning, and optimization of HPC and ML frameworks for enhanced efficiency and performance across heterogeneous hardware.

Moving forward, we plan to expand HeteroBench by introducing new benchmarks, advanced parallelization techniques, and optimizations for emerging heterogeneous platforms. Additionally, we aim to incorporate energy consumption measurements, offering a more comprehensive evaluation of energy-performance trade-offs across different accelerators. By integrating both performance and sustainability considerations, HeteroBench will evolve into a holistic benchmarking framework, addressing the growing demands of modern heterogeneous computing.

Acknowledgments

This work is partially supported by the National Science Foundation Award number 2443992.

A Appendices

We briefly introduce the benchmarks used in this study. For each case we enumerate the distinct computational stages that justify the inclusion of these benchmarks in our HeteroBench suite.

A.1 Canny Edge Detection (ced)

Canny Edge Detection [6] is a fundamental image processing algorithm used to detect edges. The algorithm has five interdependent and computationally intensive stages: gaussian filtering, gradient intensity and direction calculation, edge thinning, double thresholding, and edge tracking by hysteresis. Each stage has its specific computational focus. For instance, the Gaussian filter stage involves significant nested for-loops to smooth the image and remove noise, while the edge thinning stage features numerous switch-case statements to refine the edge detection.

A.2 Sobel Filter (sbf)

Sobel Filter [27] applies a convolutional kernel to an image to detect edges and includes three main stages: detecting horizontal and vertical edges in the first two stages by applying 3×3 convolution kernels, and computing the gradient magnitude in the third stage.

A.3 Optical Flow (opf)

Optical Flow [39] is used to estimate the motion of objects between consecutive image frames. This benchmark involves multiple stages, including the computation of gradients in the x, y, and z directions, applying gradient and tensor weightings, computing the outer product of gradient vectors, and calculating the optical flow vectors.

A.4 Convolutional Neural Network (cnn)

Convolutional Neural Network (CNN) represents a deep learning model widely used in image recognition and classification. It includes several key operations: convolution, ReLU activation, max pooling, input padding, dot product and addition in fully connected layers, and the softmax function in the output layer.

A.5 Multi-layer Perceptron (mlp)

The Multi-layer Perceptron (MLP) is a classic neural network with multiple layers, where each node in one layer connects to every node in the next, facilitating dense and parallelizable computations. This benchmark evaluates accelerator performance when performing dense matrix operations and neural network computations. Our `mlp` benchmark includes four layers: the first three perform dot product and addition operations followed by a sigmoid activation, and the fourth layer performs dot product and addition operations followed by a softmax function.

A.6 Digit Recognition (dgr)

Digit Recognition [39], a classic machine learning task involving classifying handwritten digits using a neural network model trained on the MNIST [37] dataset. This benchmark measures accelerator performance in neural network inference, which includes multiple layers of matrix multiplications and activations. The benchmark includes two primary kernels: one updates the k-nearest neighbors list for each input digit, and the other performs the voting process to determine the final classification using these neighbors.

A.7 One Head Attention (oha)

The One Head Attention benchmark is based on the attention mechanism [33] used in the general transformer model within large language models. This benchmark implements a single attention head, whose attention score is given by:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

where Q, K, and V are the query, key, and value matrices, respectively, and d_k is the dimension of the key vectors. In our implementation, there are three main compute kernels: softmax, matrix multiplication, and transpose. This benchmark provides valuable insights into the scalability of single-head attention computation, which is essential for optimizing transformer models on different computing architectures.

A.8 Spam Filter (spf)

The Spam Filter [39] application classifies email messages as spam or not using a machine learning model. The benchmark includes several computing stages: calculating the dot product, applying the sigmoid activation function, computing gradients, and updating parameters, which are essential for training and inference in text classification models.

A.9 3 Matrix Multiplications (3mm)

The 3 Matrix Multiplications application involves three consecutive matrix multiplication operations: $G = (A \times B) \times (C \times D)$. This benchmark tests system performance in handling multiple dense linear algebra operations, which is crucial in many scientific and engineering applications.

A.10 Alternating Direction Implicit (adi)

Alternating Direction Implicit [20] solves partial differential equations by breaking down a multidimensional problem into a series of more manageable one-dimensional problems. In our implementation, although the core computation is primarily handled by a single kernel, the presence of the initialization kernel justifies treating this benchmark as a multi-kernel application.

A.11 Parallelize Particle (ppc)

Parallelize Particle is a benchmark extensively used in simulations. It includes two main kernels: one to calculate the interaction forces between the particles and another to update their positions and velocities. Both kernels contain several sub-computational tasks that need to be parallelized effectively to achieve optimal performance. This benchmark is ideal for evaluating the scalability and efficiency of heterogeneous hardware accelerators in large-scale simulations due to the inherently parallel nature of particle simulations.

References

[1] AMD. 2016. AMD ROCm™ Software. https://www.amd.com/en/products/software/rocm.html
[2] AMD. 2023. Alveo U280 Data Center Accelerator Card User Guide. https://docs.amd.com/r/en-US/ug1314-alveo-u280-reconfig-accel

[3] AMD. 2024. Working with Variable Loop Bounds in Vitis HLS. https://docs.amd.com/r/en-US/ug1399-vitis-hls/Working-with-Variable-Loop-Bounds

[4] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley.* Technical Report UCB/EECS-2006-183. UC, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[5] Najmeh Nazari Bavarsad, Hosein Mohammadi Makrani, Hossein Sayadi, Lawrence Landis, Setareh Rafatirad, and Houman Homayoun. 2021. HosNa: A DPC++ Benchmark Suite for Heterogeneous Architectures. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE Computer Society, Los Alamitos, CA, USA, 509–516. doi:10.1109/ICCD53106.2021.00084

[6] John Canny. 1986. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8, 6 (1986), 679–698. doi:10.1109/TPAMI.1986.4767851

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, USA, 44–54. doi:10.1109/IISWC.2009.5306797

[8] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proceedings of the ACM on Programming Languages* 8 (2024), 593–620.

[9] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. 2018. Understanding Performance Differences of FPGAs and GPUs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, Los Alamitos, CA, USA, 93–96. doi:10.1109/FCCM.2018.00023

[10] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[11] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (Pittsburgh, Pennsylvania, USA) *(GPGPU-3)*. Association for Computing Machinery, New York, NY, USA, 63–74. doi:10.1145/1735688.1735702

[12] Juan Gomez-Luna, Izzat El Hajj, Li-Wen Chang, Victor Garcia-Flores, Simon Garcia de Gonzalo, Thomas B. Jablin, Antonio J. Pena, and Wen-mei Hwu. 2017. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, Los Alamitos, CA, USA, 43–54. doi:10.1109/ISPASS.2017.7975269

[13] Vinh Quoc Hoang and Yuhua Chen. 2023. Cost-effective network reordering using FPGA. *Sensors* 23, 2 (2023), 819.

[14] Sitao Huang, Li-Wen Chang, Izzat El Hajj, Simon Garcia de Gonzalo, Juan Gómez-Luna, Sai Rahul Chalamalasetti, Mohamed El-Hadedy, Dejan Milojicic, Onur Mutlu, Deming Chen, and Wen-mei Hwu. 2019. Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (Mumbai, India) *(ICPE '19)*. Association for Computing Machinery, New York, NY, USA, 79–90. doi:10.1145/3297663.3310305

[15] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen-Mei Hwu. 2021. PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow. *IEEE Trans. Comput.* 70, 12 (Dec. 2021), 2015–2028. doi:10.1109/TC.2021.3123465

[16] Intel. 2020. oneAPI: A New Era of Heterogeneous Computing. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html#gs.c6xt5d

[17] Mark Klaisoongnoen, Nick Brown, and Oliver Thomson Brown. 2022. Low-power option Greeks: Efficiency-driven market risk analysis using FPGAs. In *Proceedings of the 12th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies* (Tsukuba, Japan) *(HEART '22)*. Association for Computing Machinery, New York, NY, USA, 95–101. doi:10.1145/3535044.3535059

[18] Sohan Lal, Aksel Alpay, Philip Salzmann, Biagio Cosenza, Alexander Hirsch, Nicolai Stawinoga, Peter Thoman, Thomas Fahringer, and Vincent Heuveline. 2020. SYCL-Bench: A Versatile Cross-Platform Benchmark Suite for Heterogeneous Computing. In *Euro-Par 2020: Parallel Processing*, Maciej Malawski and Krzysztof Rzadca (Eds.). Springer International Publishing, Cham, 629–644.

[19] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. doi:10.1145/2833157.2833162

[20] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 11, 14 pages. doi:10.1145/3126908.3126958

[21] Perhaad Mistry, Yash Ukidave, Dana Schaa, and David Kaeli. 2013. Valar: a benchmark suite to study the dynamic behavior of heterogeneous systems. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (Houston, Texas, USA) *(GPGPU-6)*. Association for Computing Machinery, New York, NY, USA, 54–65. doi:10.1145/2458523.2458529

[22] Nvidia. 2007. Nvidia CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit

[23] Pallets. 2024. jinja2 template engien. https://jinja.palletsprojects.com/en/stable/

[24] Louis-Noel Pouchet. 2012. Polybench/c. https://web.cs.ucla.edu/~pouchet/software/polybench/

[25] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, Los Alamitos, CA, USA, 110–119. doi:10.1109/IISWC.2014.6983050

[26] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. doi:10.1145/3318464.3380595

[27] Irwin Sobel and Gary Feldman. 1968. An Isotropic 3x3 Image Gradient Operator. Presented at the Stanford Artificial Intelligence Project (SAIL). https://en.wikipedia.org/wiki/Sobel_operator Later popularized as the Sobel operator for edge detection in image processing.

[28] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter Mccardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–10. doi:10.1109/IISWC.2016.7581262

[29] Xiaoyong Tang and Zhuojun Fu. 2020. CPU–GPU utilization aware energy-efficient scheduling algorithm on heterogeneous computing systems. *IEEE Access* 8 (2020), 58948–58958.

[30] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[31] Bora Ucar, Cevdet Aykanat, Kamer Kaya, and Murat Ikinci. 2006. Task assignment in heterogeneous computing systems. *Journal of parallel and Distributed Computing* 66, 1 (2006), 32–46.

[32] Muthukumaran Vaithianathan, Mahesh Patil, Shunyee Frank Ng, and Shiv Udkar. 2023. Comparative Study of FPGA and GPU for High-Performance Computing and AI. *ESP International Journal of Advancements in Computational Technology (ESPIJACT)* 1, 1 (2023), 37–46.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[34] Qiang Wang, Pengfei Xu, Yatao Zhang, and Xiaowen Chu. 2017. EPPMiner: An Extended Benchmark Suite for Energy, Power and Performance Characterization of Heterogeneous Architecture. In *Proceedings of the Eighth International Conference on Future Energy Systems* (Shatin, Hong Kong) *(e-Energy '17)*. Association for Computing Machinery, New York, NY, USA, 23–33. doi:10.1145/3077839.3077858

[35] Xilinx. 2020. Xilinx Runtime Native APIs. https://xilinx.github.io/XRT/master/html/xrt_native_apis.html

[36] Xilinx. 2024. Vitis High-Level Synthesis. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html

[37] Christopher J.C. Burges Yann LeCun, Corinna Cortes. 1994. THE MNIST DATABASE of handwritten digits.

[38] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 741–755. doi:10.1109/HPCA53966.2022.00060

[39] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 269–278. doi:10.1145/3174243.3174255