[LabName]-CUDA: A Modern Benchmark Suite for CUDA-Capable GPUs

The ever-growing demand for massive computational power has led to the advent and rapid development of GPGPUs. This demand constantly drives research on improving performance of GPGPUs, which results in many solutions proposed each year; therefore, it is imperative to have a reliable approach to measure the impact of new ideas. Although measuring the performance impact on workloads of a benchmark suite is popular, it is only as reliable and representative of real-world performance as the underlying collection of workloads. With nearly all of the benchmark suites in use for evaluating the performance of CUDA-capable devices being outdated due to the advent of modern applications and recent hardware modifications, there is a serious need for a modern benchmark suite that covers a variety of modern-day applications and can take advantage of the components available in modern GPUs. Ease of use is another often overlooked factor of importance in a benchmark suite, which is lacking in many currently available suites. In this paper, we introduce [LabName]-CUDA[1] to address these issues and provide a benchmark suite that could advance research and innovation in the field in the upcoming years. The proposed suite, being capable of stressing individual GPU units, it is possible to measure the impacts of individual units and compare different approaches for implementing algorithms. Using experiments, we show that previous suites can potentially result in considerable difference to optimized implementations of the workloads used in real world, sometimes as much as 95 percent.

CCS Concepts: • **Computing methodologies → Graphics processors**; *Simulation evaluation.*

Additional Key Words and Phrases: Benchmarking, Benchmark Suite, GPU, GPGPU, CUDA

ACM Reference Format:
. 2025. [LabName]-CUDA: A Modern Benchmark Suite for CUDA-Capable GPUs. 1, 1 (January 2025), 22 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Graphics processing units (GPUs) have revolutionized high-performance computing, becoming indispensable in various fields such as scientific modeling, machine learning, and bioinformatics [34]. Their massive parallel processing capabilities have fueled orders of magnitude speedups on computationally intensive workloads, establishing GPUs as the driving force behind recent advances in these domains. As modern applications continue to demand more computational power, GPUs remain at the forefront of innovation.

[1][LabName] is a placeholder for the name of the laboratory in which this work was created. It is changed to comply with review rules and will change back to the laboratory's name upon acceptance of this work for publication.

Author's address:

ACM XXXX-XXXX/2025/1-ART
https://doi.org/XXXXXXX.XXXXXXX

To fully optimize and advance GPU designs, comprehensive benchmarking is imperative. Detailed and rigorous benchmarking provides vital insight into potential performance bottlenecks, inefficiencies, and opportunities for improvement across all aspects of GPU architectures. Well-designed benchmarks that stress test GPUs' emerging capabilities are crucial for holistically understanding modern GPUs and guiding their ongoing development.

However, most existing GPU benchmarks fail to adequately evaluate contemporary GPU design. These outdated benchmarks are based on deprecated libraries and target previous generation architectural features [25]. They neglect recent innovations such as configurable on-chip memory and cache compression. Consequently, existing benchmarks cannot fully characterize or tax the performance of modern GPUs and can mislead final conclusions based on the simulation results.

This work introduces an updated GPU benchmarking suite, which leverages new libraries and workloads tailored to exercise cutting-edge GPU capabilities. Our benchmark stresses critical subsystems including tensor cores, cache hierarchies, and on-chip data movement. By thoroughly targeting all major architectural components, this benchmark provides the comprehensive profiling necessary to analyze current GPUs and inform future advancement. Our approach establishes a robust framework for rigorous benchmarking moving forward.

This paper makes the following main contributions:

- Modernization of traditional GPU applications by implementing workloads using CUDA 12 libraries.
- Inclusion of scenarios that take advantage of hardware units that were later added to GPUs, such as tensor cores, special function, and cache compression units.
- Inclusion of long-running kernels that could exceed the run time of previously available benchmark kernels by orders of magnitude.
- Introducing new workloads in addition to traditional ones.
- Close to production implementation of workloads.

2 CHALLENGES OF DESIGNING BENCHMARK SUITE

In this section, we mention the important challenges for designing a timely benchmark suite for GPUs. they improve the quality of future evaluations.

2.1 Suiting to modern Workloads and Hardware

A significant part of designing a benchmark suite is selecting workloads and gathering datasets that truly represent real-world applications and possible future workloads [31]. Neglecting this step could quickly drive a benchmark suite to the point of obsolescence. It is crucial to keep in mind the features of the modern GPUs and how they could affect the performance of different workloads. It is also crucial to have a wide variety of workloads in the suite to give a full view of the performance of the evaluated systems. The advancements in the field of artificial intelligence and machine learning have led to the ever-growing usage of GPUs for these applications and the advent of many deep learning algorithms that are commonly run on GPUs. Any of these algorithms has its own unique characteristics and similarities to other deep learning algorithms. This means that although they are similar in some aspects, each of them has its unique resource utilization and demands and has a different behavior and performance from the others [51]. While it is impossible to have an all-encompassing suite for deep learning algorithms, great care must be put into choosing a subset that encompasses the most commonly used algorithms for this field and represents this family of algorithms well. While deep learning algorithms have received a lot of attention lately, there are still many other algorithms that benefit from the massive parallelism of GPUs every day. Therefore, it is important to include a meaningful set of such applications and workloads as well. While it

is impossible to include all of the said workloads, great care must be taken when including the most common ones in addition to a set of algorithms that could represent a wide range of GPU algorithms. This set of algorithms should be able to include edge cases and peculiar behaviors of most common workloads. There are fundamental algorithms and operations, different combinations of which form the bulk of GPU algorithms and applications. As a result, not including them in a benchmark suite could put its integrity into question. Therefore, a well-curated benchmark suite should include these algorithms too.

2.2 Wide-ranging datasets

Just as important as workloads, is the data that each workload is run on. As the behavior of many different algorithms could change based on the data they are processing [50], lack of a proper dataset would mean they could only showcase a part of its behavior and therefore not be representative of actual runs of that particular algorithm. More recent GPUs benefit from a data compression unit inside their L2 cache [33], thus, they have become sensitive to the data as well, even if the program they are running is not. This makes it possible for GPUs to have a performance difference for the exact same algorithms when the input and output data have different levels of sparsity. This necessitates the inclusion of a dataset with various amounts of sparsity. The size of datasets could also reveal details about the memory management system. While smaller data have negligible memory management burden, larger data could push the memory management system to its limits and reveal the benefits and downsides of various memory management methods and distinguish them based on their latency and other performance-related factors.

2.3 Ease of Use and Maintainability

While it might not seem as evident as the other challenges, ease of use is an often overlooked challenge when it comes to benchmark suites. With almost all currently recognized suites [1, 4, 13, 40, 20, 2, 3, 49, 32, 36] going out of date, even the compilation of them has become increasingly difficult in recent years. While for some researchers, the compilation of suites is enough, others find that they need to study the workloads to understand the bottlenecks. In addition to testing and providing software and compiler modifications, researchers often need to modify the source code of the suites. Overcomplicated implementations, poor code style, bad practices, and lack of documentation would make it unbearable to work with such suites and slow down the performance of researchers. Maintainability is another important aspect closely related to ease of use. While it is natural for older versions of libraries and tools to become deprecated, it could lead to the obsolescence of benchmark suites. That is unless they are updated to adapt to the new version of libraries and tools. This seems like a simple enough task, but, in reality, it is extremely difficult to do for many of the recognized suites. The reason is either complications in the implementation or poor design or documentation.

3 MOTIVATION

Having struggled with GPU benchmark suites in our past research projects, we noticed how much real applications have diverged from benchmark suites. We often had to evaluate our work on a number of suites, and still, the results missed many of the more recent GPU applications, such as many deep learning workloads. Current benchmarks fail to utilize the available hardware resources in more recent GPUs as well, which kept us wondering whether the exact same workload would have the same performance in a real-world application. In addition, we sometimes even struggled with compiling them because of their use of deprecated tools and libraries. We often had to test multiple past versions of libraries until we could successfully compile them. Even in the case of a

successful compile, it was still pretty difficult to understand how they worked and even harder to perform even the smallest modifications on them to test out software and compiler-related ideas.

In the face of these challenges, we decided to create a benchmark suite to address the mentioned issues. Firstly, by curating a list of the modern and traditional GPU workloads and then by making sure that we are able to utilize the available on-chip resources in modern GPUs, including a wide variety of workloads, to eliminate the need for testing on a number of benchmark suites. Having struggled with other benchmark suites in the past, we were determined to create an easy-to-use and maintainable benchmark suite. Toward this end, we chose a Google code style for C++ language [19] to strictly follow to make our code readable, avoid unnecessary use of external dependencies, and keep our implementations as simple as possible. To make our work even easier to use and more flexible, we have created scripts for data generation instead of storing them on a server, which makes our work more accessible and easy to extend the initially provided datasets.

4 [LABNAME]-CUDA WORKLOADS

Aiming at covering a wide variety of modern GPU applications, as well as using as many hardware capabilities as possible, we implemented numerous algorithms that represent popular applications or provide building blocks for them. Naturally, these algorithms require appropriate datasets to become proper benchmark workloads. We took great care of gathering and constructing datasets for these algorithms, either using real-world datasets that are common to the corresponding algorithms or carefully crafting them in the case of more fundamental algorithms in order to showcase the behavior of modern hardware according to the dataset. We have grouped these workloads into four categories. Fundamental algorithms, which provide building blocks of most GPU applications, graph algorithms, deep learning algorithms, and the fourth category includes workloads that do not fit in any of the former but are not less important.

To include all possible implementations and target specific units, workloads can have multiple implementations. We use the following tags to specify various implementations of each workload that are available in our suite.

***CUDA*:** This tag specifies the basic implementation of each workload that is created using general CUDA commands and does not use any additional library or special hardware.

***CULib*:** This tag shows there is an implementation of the workload that was created using CUDA libraries provided by Nvidia. These libraries are often highly optimized and offer great performance, but are ambiguous.

***SFU*:** The SFU tag signifies the use of special function unit in an implementation. Special Function Unit (SFU) offers extremely fast estimates for some of the common yet time-consuming floating point operations on GPUs. These operations include calculation of Sine and Cosine functions. The SFU implementations take advantage of this unit to provide a faster implementation of the workload.

***TC*:** This tag marks the use of tensor cores in implementation. Tensor cores are basically hardware accelerators for GEMM operations. While being a more recent addition to GPUs, tensor cores proved themselves to be incredibly useful in modern workloads such as neural networks. This tag signifies implementations that take advantage of these units to speed up the operations.

***CoOp*:** This tag showcases the use of cooperative groups [6] in the implementation of the algorithm. Some algorithms are executed in different steps or stages. The traditional method for synchronization between these steps and therefore implementing these is an algorithm is through executing them in consecutive kernel calls, which introduces considerable overhead

in terms of execution time. Algorithms with this tag take advantage of cooperative groups to perform this synchronization and, therefore, eliminate the overhead of mentioned kernel calls. This implementation is in addition to the traditional implementation through consecutive kernel calls.

***Torch*:** This tag is used exclusively in deep learning workloads. This tag signals that the workload was implemented using torch libraries. Torch libraries offer optimized and easy to use functions for developing and deploying deep learning algorithms and underlying operations. Initially released as PyTorch for python[37], the torch has since expanded and currently has C++ and Java interfaces as well. Due to these reasons, torch has become one of the most sought after ways for researching, developing, and deploying deep learning algorithms.

Additionally, With the exception of deep learning workloads, every other workload can be compiled with or without cache compression enabled by setting a single boolean flag. L2 cache data compression unit is a feature of modern GPUs that is often not properly examined in other benchmark suites. With the aim of exposing the behavior of this particular unit, as well as other sparsity-related capabilities, we carefully crafted the data to have various degrees of sparsity in each of the represented sizes. With the exception of deep learning workloads, every other workload can be compiled with or without cache compression enabled by setting a single boolean flag.

4.1 Fundamental Algorithms

Providing the building blocks of most applications, these algorithms are among the most important workloads. The performance of these algorithms is crucial for the performance of almost all the GPU applications, thus, there are numerous implementations and optimizations for these algorithms, some of which use special hardware units in GPUs.

4.1.1 Reduction. *CUDA, CoOp*

The reduction algorithm is used to compute the summation of all elements in a large array or a high-dimensional vector on highly parallel processors, such as GPUs.

The algorithm executes in multiple steps and requires a total of $\log_2 N$ steps to compute the total sum of all elements in the array, where N is the number of elements. In the first step, $\frac{N}{2}$ threads are spawned, with each thread adding two elements together and storing the result in the position of the first element. In subsequent steps, the number of active threads is halved, and each thread processes two results from the previous step, storing the result in the first element's position. After $\log_2 N$ steps, the final result is stored in the first element. Notably, in each step, $\frac{N}{2^{\text{step}}}$ threads are active.

The performance of the reduction algorithm is primarily constrained by memory bandwidth and access latency. Additionally, factors such as context-switching latency between threads, warp scheduling overhead, and synchronization delays can significantly impact performance. While the use of cooperative groups [6] reduces the overhead associated with multiple kernel calls, it introduces more threads in the later steps of the algorithm. This provides an interesting case study for analyzing GPU scheduler behavior. Furthermore, GPU caches can help mitigate memory access latency, especially in later steps where the number of accessed elements decreases by half with each step.

4.1.2 Matrix Addition. *CUDA*

Matrix addition simply involves performing element-wise addition of matrices with the same dimensions. While extremely simple, this operation is part of many GPU applications. In this application, one thread is created per element of the result matrix, which performs the appropriate

addition for that element and stores the result in its place. The execution time of this simple workload is determined solely by memory bandwidth, access latency, and the overhead of context switching between GPU threads. Since each element of the matrices is accessed only once, the GPU cache does not provide any benefit either.

4.1.3 Vector Addition. *CUDA*

In addition to matrix addition, there is a vector addition workload in the suite that adds two vectors together instead of two matrices. Naturally, this workload is extremely similar to matrix addition.

4.1.4 Matrix Multiplication. *CUDA, CULib, TC*

Matrix multiplication is another extremely common operation of GPU applications. This workload simply multiplies two matrices with the proper dimension and calculates the result. In order to perform this operation, a thread is created per element of the resulting matrix that performs the operations required for calculating the value of that element.
While being very similar to matrix addition, multiplication has more complex operations, loops, and access each element of the multiplier and multiplicand more than once, therefore the execution time of this operation is bound to the performance of many more parts of GPU such as ALU, pipeline, and cache. While the tensor core's primary operation is MAC (multiplication and accumulation), they can be used to speed up matrix multiplication as well, therefore, we included an implementation of this operation that takes advantage of tensor cores as well.

4.1.5 Vector Multiplication. *CUDA*

A vector multiplication workload is also available in the suite. While this operation is conducted many times in matrix multiplication, this workload showcases how a single vector multiplication could be parallelized and the resource utilization of this parallelized implementation.

4.1.6 Matrix Multiplication and Accumulation (MAC). *CUDA, TC*

Multiplication and accumulation of matrices is the most common operation in artificial neural networks and deep learning algorithms, which constitute a significant portion of modern GPU applications. This operation involves the multiplication of two matrices followed by the addition of another matrix to the result. In mathematical terms: $Result = A + (B \times C)$
This operation utilizes GPU resources in a manner very similar to matrix multiplication. The primary operation of tensor cores is also MAC; therefore, an implementation of this operation using tensor cores is available in the suite.

4.1.7 2D/3D Correlation. *CUDA, CULib, TC*

This operation involves correlating a matrix with a mask (another matrix of smaller size). At its core, this operation consists of a series of element-wise multiplications and reductions, repeated for each element of the resulting matrix. It is one of the most fundamental operations in most image processing and computer vision algorithms. Both 2-dimensional and 3-dimensional variants of this operation are available in the suite. Similar to matrix multiplication and MAC, elements of the operands are accessed multiple times, and a large number of threads are created to perform this operation. Therefore, this operation stresses the same GPU resources as those operations but exhibits different behavior. It is also possible to take advantage of tensor cores to accelerate this operation, and a variant that utilizes tensor cores is included in the suite.

4.1.8 2D/3D Convolution. *CUDA, CULib, TC*

Convolutions are similar to correlations and are fundamental in image processing and computer vision. They perform the same arithmetic operations as correlations, with convolution achieved by flipping the mask in a correlation. Both 2D and 3D convolutions are included in the suite and, like correlations, place similar stress on GPU resources.

4.1.9 Fast Fourier Transform (FFT). *CUDA, CULib, SFU*

The Fourier transform is widely used in signal processing across science, engineering, and music. It converts a signal from its original domain, often time, to the frequency domain. For discrete signals (processed by computers), it is known as the discrete Fourier transform (DFT), with the formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi \frac{kn}{N}}$$

.

The Cooley-Tukey fast Fourier transform (FFT) algorithm efficiently computes the DFT and is suitable for GPU implementation due to its highly parallelizable structure. However, while faster than direct computation, it can still be time-intensive for large signals.

The FFT involves complex arithmetic operations that stress the ALU, memory, scheduler, and CUDA cores. As these operations are not directly supported by the ALU, special function units offer faster approximations through single instructions. This suite provides implementations using both simple CUDA arithmetic operations and special function units, enabling researchers to compare their performance.

4.1.10 Inverse Fast Fourier Transform (IFFT). *CUDA, CULib, SFU*

As the name suggests, this operation is the inverse of FFT and performs the inverse discrete Fourier transform (IDFT). It converts a signal from the frequency domain back to its original domain, often the time domain. The formula is:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{i2\pi \frac{kn}{N}}$$

Like FFT, the IFFT uses the parallel Cooley-Tukey algorithm. Due to the similarity in calculations, this workload stresses the same modules as the FFT workload. A special function unit variant is also available in this suite, as with the FFT.

4.1.11 Discrete Cosine Transform (DCT). *CUDA, SFU*

The discrete cosine transform (DCT) reconstructs a discrete signal as a sum of cosine functions with varying frequencies and amplitudes. It is widely used for image and signal compression in formats like JPEG, H.265, and MP3.

This workload is similar to FFT in terms of processes and operations, utilizing the same GPU modules in a comparable manner. However, differences between the two may lead to slight variations in bottlenecks and resource utilization. Like the FFT workload, this workload can be accelerated using special function units, and such an implementation is included in the suite.

4.1.12 Inverse Discrete Cosine Transform (IDCT). *CUDA, SFU*

Similar to the inverse fast Fourier transform (IFFT), the inverse discrete cosine transform (IDCT) reverses the discrete cosine transform (DCT), reconstructing the original signal. IDCT closely

resembles DCT in terms of processes, operations, and arithmetic. Given its similarity to both DCT and IFFT, it is anticipated to utilize GPU resources comparably, benefiting from similar optimization techniques and accelerators. Implementations of this workload are available using both special function units and standard ALU operations.

4.1.13 Sparse Matrix Addition. *CUDA*

For highly sparse matrices, alternative storage formats, such as compressed sparse row (CSR), also known as compressed row storage or Yale format, can reduce space and computation time. In CSR format, three values are stored for each non-zero element: the row index, the column index, and the element's value.

This workload performs the addition of two matrices stored in CSR format and outputs the result in the same format. While the arithmetic operations are similar to standard matrix addition, the workload involves far fewer operations and exhibits a significantly different memory access pattern. Consequently, it utilizes the same resources as matrix addition but in a different manner.

4.1.14 Sparse Matrix-Matrix Multiplication (SPMM). *CUDA, CULib*

This workload exemplifies sparse matrix operations, specifically sparse-dense matrix multiplication (SPMM), where a sparse matrix in CSR format is multiplied by a dense matrix, producing a dense matrix as the result.

Although the arithmetic operations are analogous to dense matrix multiplication, the workload involves fewer operations due to sparsity and exhibits a markedly different memory access pattern. Consequently, while resource utilization is similar to dense multiplication, the specifics of how resources are utilized differ.

4.1.15 Sparse Matrix-Vector Multiplication (SPMV). *CUDA, CULib*

SPMV, multiplies a sparse matrix by a dense vector. Just like other sparse matrix workloads, sparse matrices for this workload are stored in CSR format. Similar to other sparse matrix operations, this workload also utilizes the same resources as the dense version, but not in exactly the same manner.

4.1.16 Matrix Norm. *CUDA, CULib*

This workload calculates the norm of a square matrix given as input. The norm is a highly utilized feature of matrices and is used in many complex operations, such as matrix manipulations related to computer vision. Calculating the norm of a matrix efficiently is, therefore, as crucial as other matrix operations. Similar to many other matrix operations, the arithmetic operations involved in this process are not too complex, with most of the load being on the memory and the scheduler.

4.1.17 Matrix Inversion. *CUDA, CULib*

Another very common matrix operation used in many different algorithms is calculating the inverse of a square matrix. This workload computes the inverse matrix using the Gaussian elimination technique. Since Gaussian elimination simply involves a series of vector-by-scalar multiplications and vector additions, there are no complex arithmetic operations involved. Memory, on the other hand, is extensively accessed, with each part being accessed multiple times, thereby involving the cache in addition to memory bandwidth and access latency.

4.1.18 System of Equations. *CUDA, CULib*

This workload computes the solution for a system of equations presented in the form of a matrix

and a vector. While being a slightly different problem, similar to matrix inversion, this workload uses the Gaussian elimination technique to solve these equations. Using a method similar to matrix inversion, it is expected that this workload will exhibit similar behavior in terms of resource utilization as the matrix inversion workload.

4.1.19 Transpose. *CUDA*

This workload transposes the input matrix. Matrix transposition can be broken into many independent tasks with ease and, therefore, is highly suitable for a parallel processor such as a GPU. This workload lacks any arithmetic operations and is purely memory-bound. Since there are no arithmetic operations and each memory location is accessed only once for loading and once for storing, the performance of this workload is directly tied to memory bandwidth, access latency, context switching, and scheduling overhead—more so than any other workload. This fact, in turn, makes it a perfect showcase for evaluating the performance of fundamental GPU components such as memory, the scheduler, and context switching.

4.2 Graph Algorithms

Graph algorithms present unique challenges and opportunities when implemented on GPUs. While they can leverage the massive parallel processing capabilities of modern GPUs, their performance is often constrained by two critical factors: irregular memory access patterns and control flow divergence. This peculiar behavior, characterized by non-uniform memory access patterns, often leads to performance bottlenecks. As a result, these algorithms provide an ideal showcase for advancements aimed at mitigating these issues. In addition, graph applications are iterative and require multiple executions of the same code, as well as frequent kernel-wide synchronizations and barriers. Older implementations of these applications executed the same kernel multiple times. While this approach works, it incurs significant kernel call and data movement penalties. With the introduction of cooperative groups [6] in Kepler GPUs, it is since possible to implement and execute these algorithms in a single kernel call—an approach that, to the best of our knowledge, is absent in all other benchmark suites. Our suite includes both implementations: one using consecutive kernel calls and the other using cooperative groups.

4.2.1 Breadth First Search (BFS). *CUDA, CoOp*

This workload performs a highly parallelized breadth-first search (exploration) algorithm on an input graph in the form of a list of edges. While like depth-first search (DFS), BFS is an extremely common search algorithm in graphs, unlike DFS, it can be parallelized and therefore executed on GPUs. While irregular memory accesses and constant synchronizations limits the performance of this workload on GPUs, depending on the input graph, GPUs can perform this operation much quicker than CPUs.
The provided dataset for this workload is the dataset for 9th DIMACS Implementation Challenge - Shortest Paths [**dimacs**] which consist of graphs representing roads in different areas of united states.

4.2.2 Bellman-Ford Algorithm. *CUDA, CoOp*

The Bellman-Ford algorithm is a solution for the single-source shortest path (SSSP) problem in graphs, which is a very common problem in pathfinding. This algorithm can compute the shortest path from a single source to any vertex, even for graphs containing negatively weighted edges, or determine that there is a negatively weighted cycle present in the graph and, therefore, finding such a path would be impossible.

The time complexity of the Bellman-Ford algorithm is $O(V \cdot E)$, where V is the number of vertices and E is the number of edges, taking a maximum of V total steps and E comparisons in each step. When executing in parallel, E comparisons in each step can be done completely independently and in parallel, thus allowing a separate thread to perform each comparison. Taking advantage of this fact makes the execution of this algorithm on GPUs possible and considerably faster compared to CPUs, despite the fact that the performance is still limited as a result of memory access patterns and synchronizations.

Similar to the BFS workload, this workload takes the input graph as a list of edges, and the provided dataset for this workload is the same as the BFS workload.

4.2.3 *Floyd-Warshall Algorithm.* *CUDA, CoOp*

The Floyd-Warshall algorithm is a solution for the all-pairs shortest path problem in graphs, which computes the shortest path from each vertex to every other vertex in the graph. Just like the Bellman-Ford algorithm, this algorithm can operate on graphs containing negatively weighted edges or determine that a solution does not exist.

The time complexity of executing this algorithm sequentially is $O(V^3)$ (V being the number of vertices), taking a maximum of V steps and V^2 comparisons in each step. Similar to Bellman-Ford, these comparisons can be done in parallel, providing a window for GPU implementation of this algorithm. However, as with the Bellman-Ford algorithm, while GPU implementation can speed up the execution, the performance gain is still limited due to the same reasons.

Unlike the Bellman-Ford workload, the Floyd-Warshall workload takes input in the form of an adjacency matrix, and the provided dataset is carefully fabricated by our team to contain different scenarios.

4.2.4 *Kruskal Algorithm.* *CUDA*

Used to compute the minimum spanning tree in graphs, Kruskal's algorithm also has the potential for parallel execution. While this potential is less than that of some other algorithms, such as Bellman-Ford, it is nonetheless present.

With a time complexity of $O(V \cdot E)$ (V being the number of vertices and E being the number of edges) in sequential execution, the algorithm takes V steps, finding the lightest edge that does not create a cycle in each step. Like other graph algorithms, operations in each step can be executed in parallel, thus making the execution of this algorithm on GPUs reasonable for large graphs. While the operations of each step are more complex and require more memory accesses, the performance gain and limiting factors remain the same as in other graph algorithms.

4.2.5 *Disjoint Set Union Algorithm.* *CUDA*

Not only is the disjoint set union (DSU) algorithm performed as part of Kruskal's algorithm, but it also has potential for parallelization and GPU implementation. Therefore, we have included it in our suite as an independent workload as well.

In this implementation, the functions corresponding to finding union roots and merging two unions have been parallelized on the GPU. This algorithm does not require a weighted graph. A set of input data for this algorithm has been created by our team and is packaged alongside the suite.

4.2.6 *Kahn Algorithm.* *CUDA*

The Kahn algorithm provides a solution for the topological sort problem. Topological sort produces an ordered list of vertices in a directed acyclic graph (DAG) such that there are no edges from

a vertex to another vertex that precedes it in the ordered list. Being among the more complex problems in graph theory, topological sort poses an interesting challenge, especially for GPU implementation.

The Kahn algorithm has a time complexity of $O(V + E)$ for sequential execution (V being the number of vertices and E being the number of edges) or $O(V \cdot E)$ for a naive implementation. When performing operations in parallel, it would take V steps, with operations in each step that can be parallelized.

Since the topological sort problem is only solvable on directed acyclic graphs, many of the already available graph datasets cannot be used as input data for this workload. To address this issue, we have included a carefully crafted dataset of DAGs to be used as sample inputs for this workload.

4.2.7 Edmonds-Karp Algorithm. *CUDA*

A classical problem in graph theory and computer science, maximum flow is an optimization problem that provides powerful modeling to solve a wide variety of problems and perform many simulations. In addition, each solution to the maximum flow problem provides a solution for the min-cut problem, which is another classic problem in graph theory. It is almost impossible to overstate the importance of this problem and the corresponding algorithms, such as the Edmonds-Karp algorithm. Despite the importance of the algorithm, its wide use cases, and an efficient GPU implementation, to the best of our knowledge, our suite is the first GPU benchmark suite to include this workload.

The Edmonds-Karp algorithm has a worst-case time complexity of $O(V \cdot E^2)$ in sequential execution (V being the number of vertices and E being the number of edges). The algorithm performs a breadth-first search (BFS) in each of its steps and takes a maximum of $O(V \cdot E)$ stages to complete. As we have shown in another workload, BFS can be parallelized and executed on GPUs.

The algorithm takes the number of vertices, edges, a list of edges and their capacities, as well as source and sink vertices as input, and calculates the maximum flow. The input for this workload comes from a dataset, created by our team, which is packaged alongside the suite.

4.3 Deep Learning Algorithms

There's no argument that deep learning applications make up a large portion of GPU applications today. Most modern deep learning algorithms are aimed at computer vision and natural language processing. These applications are rare in currently available benchmark suites, and even when they are included, the workloads often consist of very simple neural networks and corresponding datasets. Furthermore, these applications are implemented from scratch and are not generalizable; differentiation and error back-propagation are hard-coded.

In contrast, most modern applications use libraries to implement their neural networks. These libraries provide efficient implementations of neural network modules and layers, as well as utilities for differentiation, error back-propagation, and training. Nevertheless, their implementation differs from that of the hard-coded neural networks present in current benchmarks.

Keeping this in mind, we implemented both simple, self-coded neural networks and more complex implementations using the Torch library. In this category, we have covered a variety of vision and language models that are commonly used in real-world applications.

4.3.1 Multi-Layer Perceptron. *CUDA*

Although the model itself and its coding style are not common today, the training and inference of this model on a curated dataset have been included with the goal of highlighting the underlying operations of more complex models.

4.3.2 *Layered Neural Network.* *CUDA*

While slightly more complex than a multi-layer perceptron, this workload uses the same coding style, is not directly in use today, and has been included (for training and inference) on a curated dataset for similar reasons.

4.3.3 *ResNet-18.* *Torch*

Winner of the ImageNet 2015 competition, ResNet broke barriers in convolutional neural networks in terms of depth with a simple architecture. By adding residual connections, residual networks demonstrated their effectiveness in models as deep as a thousand layers, as well as in shallower networks [23]. Due to their simplicity and effectiveness, residual networks have remained in wide use, either on their own or as a backbone for more complex networks.

ResNet-18 training and inference on the Tiny ImageNet dataset [46] are included in the suite. The choice of Tiny ImageNet instead of the main ImageNet dataset is driven by practicality. The massive size of the ImageNet dataset eliminates any practical benchmarking use cases.

4.3.4 *VGG Net.* *Torch*

The first attempt at deep convolutional neural networks, VGG net, secured second place in the ImageNet image classification competition in 2014. The revolutionary network pushed the depth of neural networks to nineteen layers [39]. Despite the rise of more advanced architectures, VGG remains in use for simpler object classification tasks.

Like ResNet, training and inference code for this algorithm on the Tiny ImageNet dataset are included in the suite.

4.3.5 *Inception Net.* *Torch*

Also known as Google LeNet, the Inception network was another attempt at deeper neural networks at the time. Inception Net won the ImageNet competition in 2014. With its unique architecture and techniques, Inception introduced an effective twenty-two-layer network [41]. Although many modern neural networks outperform the original Inception network, the network itself and its variants are still in use due to their relatively low resource requirements and reasonable accuracy.

Like ResNet, training and inference code for this algorithm on the Tiny ImageNet dataset are included in the suite.

4.3.6 *YOLOv1, YOLOv3, and YOLOv5.* *Torch*

YOLO, or You Only Look Once, is a groundbreaking approach to real-time object detection that frames the task as a regression problem, predicting bounding boxes and class probabilities directly from full images in a single evaluation [38]. Introduced in 2015, YOLO enabled real-time object detection using commercial hardware. Over the next few years, YOLO received incremental updates and newer versions.

The next major step for YOLO was YOLOv3, introduced in 2018. YOLOv3 improved the model's ability to detect objects at different scales by generating multiple outputs from several parts of a single neural network [18]. Improvements to the YOLO algorithm continued in the following years, one such improvement being YOLOv5. While not as revolutionary as v1 or v3, YOLOv5 enhances the accuracy of the YOLO model [27].

YOLOv1, YOLOv3, and YOLOv5 training and inference on the COCO and Pascal VOC datasets [30] [17] are included in the suite.

4.3.7 *Mask R-CNN.* *Torch*

Improving R-CNN and extending its use to semantic segmentation, Mask R-CNN is probably the only variant of R-CNN that is still in use today [22]. Training and inference for this algorithm have been included in the presented suite to represent more complex uses of neural networks and serve as a sample segmentation algorithm.

4.3.8 *LLaMA 3.2 (1B & 3B).* *Torch*

Introduced by Meta in 2023, LLaMA became the first open-source pre-trained large language model (LLM). The open-source model outperformed some contemporary proprietary models with more parameters in certain tasks while struggling in others [48]. Later in 2023, LLaMA 2 was introduced, improving the performance of the original LLaMA while remaining open-source with available pre-trained weights [47]. In 2024, LLaMA 3 [14], including versions 3.1 and 3.2, was published in a similar manner to the original. These newer models range from one billion to four hundred five billion parameters. As is common with larger neural networks, these models can operate using half-precision floating-point values. Keeping this in mind, the smallest variants of these models can easily be run on most high-end consumer and data-center GPUs. Since the most common use of large language models is inference rather than training, we have included inference only for the smallest variants of each version of these models in this suite. The inference process uses the original pre-trained weights of the models. Performing inference alone is a massive task, and training on a single GPU is nearly impossible. To fit these models on a single GPU, the workload loads the weights and executes operations using half-precision floating-point values. Both 1 billion and 3 billion parameters variants of this algorithm are included in this suite.

4.3.9 *Gemma and Gemma 2.* *Torch*

Gemma is another family of open language models introduced by Google in 2024. Gemma is considerably smaller than LLaMA and most modern language models. The goal of the Gemma models is to mimic the considerably larger Gemini model [42]. Gemma comes in two variants, with two and seven billion parameters [44].

Later in 2024, Gemma 2 was introduced to improve the performance of Gemma, offering three variants with 2, 9, and 27 billion parameters [43].

Like LLaMA, the inference workload for the smallest variants of these models with 2 billion parameters have been included in this benchmark suite.

4.4 Other Important Algorithms

This category contains some algorithms that, although widely used in applications, do not fit into any of the above categories.

4.4.1 *Bloom Filter.* *CUDA*

Created in 1970 by B. H. Bloom, the Bloom filter is a space-efficient probabilistic method based on hashing, which provides a technique for filtering common elements in a set. While the algorithm can lead to false positives, it offers an effective way to filter common elements in a large set. Therefore, it is commonly utilized in stream processing for big data applications. It is also possible to perform filtering operations in parallel, making it feasible to implement on GPUs.

This workload is a parallel implementation of the Bloom filter for checking whether an element is repeated more than a specified number of times in a large dataset.

4.4.2 Page-Rank. *CUDA*

Introduced in 1998 by Page et al.[35], the Page-Rank algorithm ranks web pages based on citations. Popularized by Google for search engines, its applications have since expanded to other domains. The algorithm involves matrix operations, which are often computationally intensive due to their large size. However, these operations can be accelerated using GPUs, enabling faster implementations.

This workload provides an efficient GPU-based implementation of the Page-Rank algorithm.

4.4.3 Histogram. *CUDA*

As the name suggests, this workload takes a series of values as input, divides them into bins, and calculates the histogram by counting the elements in each bin. The data range and number of bins are provided as command-line arguments, which also determine the size of each bin. This operation is naturally performed in parallel using massively parallel GPUs.

4.4.4 N-Body Simulation. *CUDA, CoOp, SFU*

The N-body problem lacks a closed-form solution, making simulation essential. It involves calculating gravitational forces, updating positions, and adjusting velocities. The simulation requires frequent synchronizations, implemented via consecutive kernel calls or cooperative groups. Operations can be accelerated using special function units (SFUs). This suite includes four versions, combining cooperative groups or kernel calls with ALU- or SFU-based implementations.

4.4.5 Join. *CUDA*

The join operation merges values from two tables based on a common column, selecting rows with matching values. It is widely used in relational databases and data processing. Traditionally executed on CPUs, the rise of larger datasets has made GPUs popular for this task. Another driver of GPU adoption is the need to preprocess data on GPUs for neural network training later in the pipeline.

This problem can be parallelized in three ways. Given a first table with N rows and a second with M rows:

1. Create N threads, each comparing one row of the first table against all M rows of the second.
2. Create M threads, each comparing all N rows of the first table against one row of the second.
3. Create $N \times M$ threads, each comparing a single row from both tables.

These approaches differ in their use of atomic operations, memory access patterns, and scheduler behavior. To account for these variations, this suite includes three workload variants, one for each partitioning method.

Additionally, a utility to generate input data with varying characteristics (1-dominated, 0-dominated, or completely random) is provided.

4.4.6 Nearest Neighbor. *CUDA, SFU, CoOp*

The nearest neighbor workload takes an unordered set of 3-dimensional coordinates as input and finds the nearest neighbor (i.e., the point with the shortest distance to each point). A kernel is launched for each point, with distance calculations and comparisons performed in parallel. Alternatively, cooperative groups can be used instead of multiple kernel calls to reduce overhead. Execution can also be accelerated by using special function units (SFUs) instead of standard

Table 1. Workloads categorized by the specific GPU feature they target.

Target	Workloads
Cooperative Groups	Reduction, BFS, Bellman-Ford, Floyd-Warshall, N-Body Simulation, Nearest Neighbor
Tensor Cores	Matrix Multiplication, Matrix Multiplication and Accumulation (MAC), 2D Correlation, 3D Correlation, 2D Convolution, 3D Convolution,
Special Function Unit	FFT, IFFT, DCT, IDCT, N-Body Simulation, Nearest Neighbor
Cache Data Compression	Fundamental Algorithms (All), Graph Algorithms (All), Miscellaneous Algorithms (All)

arithmetic operations. To account for these possibilities, our suite includes four variations of this workload.

4.4.7 AES. *CUDA*

The Advanced Encryption Standard (AES) is a symmetric encryption block cipher algorithm, widely used both independently and as part of stronger cipher algorithms. Its specifications were published by the U.S. National Institute of Standards and Technology (NIST) in 2001 [16] and have remained in mainstream use since. As a block cipher, AES can be executed in parallel by applying the same algorithm to fixed-size data blocks (128 bits in the original specification), making it well-suited for GPU acceleration. Our suite includes this commonly used algorithm and an accompanying corpus as a workload to explore GPU-based cryptographic operations.

4.4.8 SHA-3. *CUDA*

Released by the U.S. National Institute of Standards and Technology (NIST) in 2015 [15], Secure Hash Algorithm 3 (SHA-3) is a strong family of permutation-based hashing algorithms, with SHA-256 being the most commonly used variant. SHA-3 is widely employed in digital signatures, message authentication codes, key derivation functions, pseudo-random functions, password storage, and other security applications [15]. Additionally, SHA-3 algorithms are relatively resistant to quantum analysis [26], making them increasingly important as quantum computing advances.

Given the significance of this algorithm family, we have included SHA-512, the strongest variant, as a workload in our suite to demonstrate GPU usage

4.5 Targeted Units and Features

A subset workloads have multiple implementations, each utilizing and stressing separate units of GPU, with the goal of drawing comparison, showcasing benefits and exposing bottlenecks of targeted units. Every unit has a specific purpose and thus, they are only effective for certain workloads. Table 1 displays the workload that target a specific GPU unit and categorizes them by the unit they utilize. All these workloads also have baseline implementations to demonstrate the effect of the respective units.

5 UNIQUE FEATURES OF OUR WORK

In this section, we discuss the unique features of our work, how they benefit researchers, and how they improve the quality of future evaluations.

5.1 Multiple implementations for many workloads

For many of the workloads covered in this suite, more than one implementation is available. There are various reasons for this. These reasons include different workload distributions between threads and varying practices for implementing algorithms [24].

One of the most fundamental aspects of GPU programming is distributing work among a large number of threads so that the program can leverage the massive parallelism available in GPUs. Although some algorithms have only one efficient way to achieve this, others may allow multiple approaches. These distributions impose different overheads on the underlying system and utilize resources in different ways. While most benchmark suites include just one implementation, we included as many as possible. This approach enables a more comprehensive study of the systems under test and helps identify bottlenecks and areas for improvement.

There are also different approaches developers use when writing programs, whether it involves utilizing different hardware resources [**Li2016SFU**] or opting for libraries instead of writing custom code [5, 7, 9, 10]. These approaches lead to different outcomes in terms of performance and resource utilization. Since multiple variants of each workload are used in practice, they are all valid for performance evaluation. Unlike other suites, we chose to include all the variants we could find. The variations in our benchmark result from using special function units (SFUs) instead of standard GPU arithmetic, employing cooperative groups [6] instead of sequential kernel calls, and utilizing efficient, specialized libraries instead of generic implementations.

5.2 Utilization of Standard Libraries

For many GPU workloads, the use of specific libraries [5, 7, 9, 10] is more common than writing code from scratch. The most prominent example of this is deep learning applications [8]. NVIDIA also provides libraries that include highly optimized implementations of common algorithms [12, 28]. In practice, most developers use these libraries, whereas benchmarks often do not. To better reflect the real-world performance of GPU applications, we used these libraries in our work, either in addition to from-scratch implementations for some workloads or as standalone solutions for certain deep learning workloads.

5.3 Addressing Sparsity in Datasets

As mentioned earlier, modern hardware has become sensitive to the sparsity of application data, regardless of the algorithm. This sensitivity is due to the addition of data compression units to the L2 cache [45]. Although varying sparsity levels in application data are often lacking in other benchmark suites, they represent an important performance factor in modern hardware. Therefore, we carefully crafted our datasets to include both highly sparse and less sparse data to address this issue.

5.4 Adherence to Well-Established Coding Standards

Dedicated to maintaining the quality of our work and the code we write, we strictly follow the Google C++ style guide [19], which is renowned for promoting clean and maintainable code. By doing so, we enable researchers to effortlessly understand the inner workings of each workload. This, in turn, helps researchers better identify bottlenecks, areas for improvement, and potential issues within their work.

5.5 Implementation with Maintainability in Mind

Having struggled with using available benchmarks just a few years after their release, we understand how crucial it is to maintain a benchmark suite beyond its initial release. That is why we started

this project with maintainability in mind. One reason we chose to follow the Google code style [19] for this project is to promote clarity and consistency. Additionally, we minimized the use of external libraries to prevent potential issues caused by their deprecation. Another step we took toward this goal was avoiding unnecessary complexities in design and implementation. We hope these efforts will keep this benchmark suite maintainable, relevant, and easy to use for years to come.

5.6 Automated Dataset Generation and Collection

We have written scripts to generate datasets on the end user's machine wherever possible and to gather datasets from their original sources when necessary. By doing so, we eliminated the need for a server to store all datasets, making maintenance easier. In addition, this approach allows end users to generate similar data and extend the provided datasets with relative ease. It also eliminates the need for large downloads, reducing the burden on users with limited Internet access.

6 EXPERIMENTAL RESULTS

With the [LabName]-CUDA able to stress individual units, that were previously left out of examination by current benchmarks, it becomes possible to measure the impact of each unit on actual workloads. We demonstrate this possibility by providing measurements for the impact of two of the targeted features and analyzing the results. We focus on special function unit (SFU) [29] and cooperative groups [6] for this purpose. SFU and cooperative groups have been left out of many evaluations while tensor cores have been the subject of many research projects. Both of these units have the potential to considerably change the execution time of the applications that use them but the extent of this change is rarely examined. Cache compression was left out as well, as it is an extremely complex topic. The impact of this unit not only depends on the algorithm, but also the input data. Size of the working set, sparsity and entropy of the data of the workload all impact the results achieved when this unit is enabled. Studying its behavior, therefore, is a complex subject and out of scope for this paper. Not all algorithms and workloads have operations that could be done by SFU or need barriers so they can use cooperative groups for them and therefore, a smaller subset of workloads that had the potential to use them are used for these evaluations.

The evaluation results presented are achieved by timing workload executions on a RTX 4090 GPU, a top-of-the-line GPU from Ada Lovelace micro-architecture [11]. This GPU includes 128 Streaming Multiprocessors (SMs), each containing 128 CUDA cores, 4 tensor cores and 16 SFUs, totalling to 16384 CUDA cores, 512 tensor cores and 2048 SFUs. The chip contains 128 KB L1 data cache per SM and 72 MB L2 data cache. Finally, the board has 24 GB of GDDR6X memory, connected with 384 bit bus to GPU with 1.01 TB/s bandwidth.

6.1 Cooperative Groups

As mentioned before cooperative groups could implement device-wide barriers [6], thus eliminate the need for using kernel completion as one. This would in turn eliminate a major bottleneck caused by CPU-GPU communication overhead. However, using cooperative groups has its own overhead. This fact poses the question of how effective using cooperative groups would be.

Figure 1 shows the runtime of workloads that use cooperative groups normalized by the runtime of their equivalent that would use consecutive kernel calls, averaged over their datasets. As evident on this chart, using cooperative groups has the potential to significantly reduce the total execution time of workloads, reducing execution time between 21 to 95 percent (42 percent, in average). The improvement heavily depends on the workload and its compute time compared to communication overhead and input data; the more times execution reaches the barrier, the more effective this approach becomes.

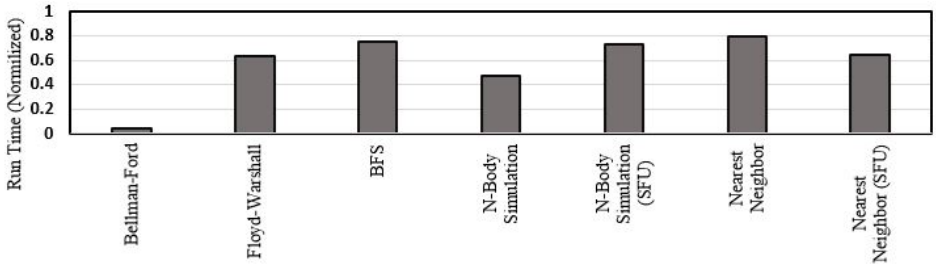

Fig. 1. Normalized run time of workloads using cooperative groups to create barriers (by consecutive kernel call implementation).

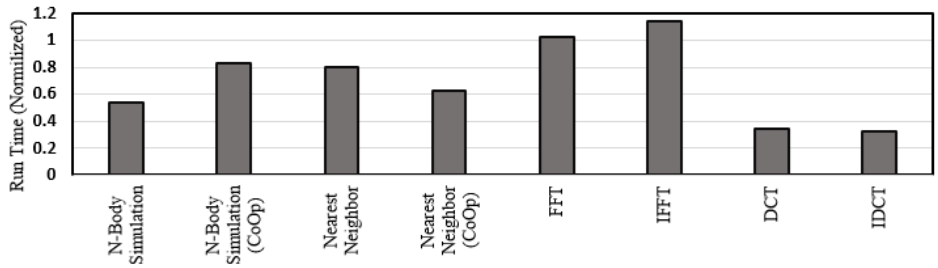

Fig. 2. Normalized run time of workloads using SFU (by CUDA arithmetic implementation).

Since most other benchmark suites neglected the use of cooperative groups [1, 4, 13, 40, 21, 20, 2, 3, 49, 32, 36], the results they produce, closely resemble the baseline for the normalization in the presented chart. As all developers strive to implement their applications in the most efficient manner, the results of previous suites could differ from real-world applications by as much as 95 percent.

6.2 Special Function Unit (SFU)

While SFU provides approximations instead of exact values, for most use cases these approximations have sufficient accuracy and the use of it has been widely reported [29]. While the speedup of a single operation realized by SFU is known, its behavior under load and in actual applications remains unknown, due to a lack of benchmarking workloads in prior research.

[LabName]-CUDA includes SFU implementations of workloads in addition to baseline implementations which use normal CUDA arithmetic, providing the opportunity to observe the impact of using the SFU in applications. Figure 2 shows one such comparison, showcasing the runtime of workloads using SFU normalized by their respective CUDA arithmetic counterpart. As evident in this chart, using SFUs can bring meaningful improvements for some workloads and can be combined with other measures to improve the execution time of programs, while making little to no difference in other workloads.

An interesting discovery of this work was the effect of using SFU with FFT and IFFT workloads. While these workloads are a perfect case for SFU and can effectively utilize this unit, we see a slight increase in their execution time. The presented result is averaged over the datasets, that come with this suite. The result is highly dependent on the input data and more on the input wave characteristics than its size. While for some input waves, we see considerable improvements, others present a longer runtime than the baseline, and some show negligible differences. Although these

algorithms can effectively use SFUs, on the flip side, they heavily stress these units and are prone to resulting stalls. Due to the limited number of SFUs in a GPU, during heavy utilization, threads will likely have to wait for them to become available to perform required operation and encounter stalls that can potentially even increase the total runtime, as observed in exhibited results. It is widely known that a lower number of SFUs compared to CUDA cores, reduces thread-level parallelism. However, the impact of this reduction of parallelism and increased number of stalls has been left out of studies. While FFT has the potential to highly benefit from faster operations offered by SFU, we observe performance degradation in them, which in turn puts the effectiveness of the current configuration of SFUs in contemporary GPUs into question.

Inclusion of these workloads not only reflects the actual benefit and potential bottlenecks of SFUs, but also streamlines future research of this unit and whether the cost of including it in GPUs is worth the improvements.

7 RELATED WORK

Several benchmark suites have been developed to evaluate GPU performance across diverse workloads and architectures. One of the most recently published suites, Altis [25], introduced in 2020, modernized GPGPU benchmarking in its time, by incorporating diverse workloads, emerging domains like DNNs, and support for advanced CUDA features to better evaluate modern GPU architectures and runtime systems. Despite this, the advent of new GPU units and the introduction of new workloads, have significantly reduced its effectiveness. Rodinia [4], introduced in 2009, evaluates heterogeneous platforms with applications inspired by the Berkeley dwarf taxonomy, exploring GPU-specific bottlenecks such as memory inefficiencies and synchronization overhead. Parboil (2012) [40] focuses on throughput computing with scalable algorithms in scientific computing and image processing domains, offering both baseline and optimized implementations. SHOC (2010) [13] supports CUDA and OpenCL, providing low- and high-level tests for GPU performance and scalability, while NUPAR (2015) [49] assesses advanced GPU features like nested parallelism and concurrent kernel execution through workloads spanning scientific and commercial applications. Similarly, Pannotia (2013) [3] targets irregular graph applications, highlighting challenges in SIMD architectures, while ISPASS 2009 [1] offers insights into GPU performance across cryptography, molecular dynamics, and finance. These suites collectively address bottlenecks such as branch divergence, memory bandwidth limitations, and thread-level parallelism.

To address benchmarking challenges for emerging workloads, proxy benchmark methodologies like PerfProx [36] have been proposed. PerfProx generates miniature proxy benchmarks that accurately mimic the performance behavior of real-world applications using hardware performance counters.

Other notable GPU benchmark suites include DeepBench [32], which evaluates operations critical to deep learning, such as matrix multiplication and convolutions, independent of frameworks. PolyBench/GPU extends PolyBench/C for GPU kernel performance testing across OpenCL, CUDA, and HMPP, with tunable instrumentation and parametric loop bounds [20, 21]. Lonestar [2] focuses on irregular workloads, analyzing performance for graph traversal, n-body simulations, and other irregular applications. These benchmarks, spanning domains from machine learning and graph analytic to scientific computation, provide valuable insights into optimization strategies for modern GPU architectures.

8 CONCLUSION

In this paper, we introduced the [LabName]-CUDA, a modern benchmark suite designed to evaluate GPU performance across different workloads, addressing limitations in existing benchmarks. The

suite consists of classical graph algorithms, deep learning models, cryptographic methods, and other classic algorithms, introducing a complete evaluation framework for CUDA capable GPUs.

Our work highlights the importance of including workloads that reflect real-world use cases, such as modern neural network architectures like ResNet, YOLO, and LLMs. By supporting multiple implementations, including both custom-coded algorithms and optimized library-based approaches, the suite enables researchers to assess performance variations due to coding styles, hardware configurations, and optimization strategies.

Moreover, [LabName]-CUDA suite supports automated dataset generation with varying sparsity levels and follows coding standards that ensure long-term usability and reproducibility. It provides a useful tool for researchers to evaluate GPU performance, find bottlenecks, and refine optimization techniques.

The meticulous targeting of separate GPU units in this suite, has revealed how they could affect the total execution time of applications, shedding light on how much real-world applications have digressed from previous benchmark implementations, exposing potential bottlenecks and areas of improvement regarding each unit and paving the way for effective evaluations of future research on GPUs.

REFERENCES

[1] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 163–174.

[2] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on gpus. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 141–151.

[3] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. 2013. Pannotia: understanding irregular gpgpu graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 185–195.

[4] 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 *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[5] NVIDIA Corporation. 2024. *cuBLAS*. Last updated on Nov, 2024. https://docs.nvidia.com/cuda/cublas/index.html.

[6] NVIDIA Corporation. 2024. *CUDA C++ Programming Guide / Coperative Groups*. Last updated on Nov, 2024. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#cooperative-groups.

[7] NVIDIA Corporation. 2024. *CUDA Toolkit Documentation 12.6 Update 3*. Last updated on Nov, 2024. https://docs.nvidia.com/cuda/.

[8] NVIDIA Corporation. 2024. *cuDNN API reference*. Last updated on Dec, 2024. https://docs.nvidia.com/deeplearning/cudnn/latest/.

[9] NVIDIA Corporation. 2024. *cuFFT API Reference*. Last updated on Nov, 2024. https://docs.nvidia.com/cuda/cufft/index.html.

[10] NVIDIA Corporation. 2024. *cuSPARSE API Reference*. Last updated on Nov, 2024. https://docs.nvidia.com/cuda/cusparse/index.html.

[11] NVIDIA Corporation. 2023. Nvidia ada gpu architecture. (2023). https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf.

[12] Neal C Crago, Sana Damani, Karthikeyan Sankaralingam, and Stephen W Keckler. 2024. Wasp: exploiting gpu pipeline parallelism with hardware-accelerated automatic warp specialization. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1–16.

[13] 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*, 63–74.

[14] Abhimanyu Dubey et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

[15] Morris J Dworkin. 2015. Sha-3 standard: permutation-based hash and extendable-output functions.

[16] Morris J Dworkin, Elaine Barker, James R Nechvatal, James Foti, Lawrence E Bassham, E Roback, James F Dray Jr, et al. 2001. Advanced encryption standard (aes).

[17] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. 2010. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88, 2, (June 2010), 303–338.

[18] Ali Farhadi and Joseph Redmon. 2018. Yolov3: an incremental improvement. In *Computer vision and pattern recognition*. Vol. 1804. Springer Berlin/Heidelberg, Germany, 1–6.

[19] Google. 2012. *Google C++ Style Guide*. Last updated on Aug, 2012. https://google.github.io/styleguide/cppguide.html.

[20] Scott Grauer-Gray and Louis-Noël Pouchet. 2012. Polybench/gpu: implementation of poly-bench codes for gpu processing. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*.

[21] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to gpu codes. In *2012 innovative parallel computing (InPar)*. Ieee, 1–10.

[22] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, 2961–2969.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

[24] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben Van Werkhoven, and Henri E Bal. 2023. Optimization techniques for gpu programming. *ACM Computing Surveys*, 55, 11, 1–81.

[25] Bodun Hu and Christopher J Rossbach. 2020. Altis: modernizing gpgpu benchmarks. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 1–11.

[26] Kyungbae Jang, Sejin Lim, Yujin Oh, Hyunjun Kim, Anubhab Baksi, Sumanta Chakraborty, and Hwajeong Seo. 2024. Quantum implementation and analysis of sha-2 and sha-3. *Cryptology ePrint Archive*.

[27] Glenn Jocher et al. 2022. Ultralytics/yolov5: v6. 2-yolov5 classification models, apple m1, reproducibility, clearml and deci. ai integrations. *Zenodo*.

[28] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. [n. d.] Cutlass: fast linear algebra in cuda c++. (). https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/.

[29] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, and Henk Corporaal. 2016. Sfu-driven transparent approximation acceleration on gpus. In *Proceedings of the 2016 International Conference on Supercomputing*, 1–14.

[30] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, 740–755.

[31] Mahmood Naderan-Tahan and Lieven Eeckhout. 2021. Cactus: top-down gpu-compute benchmarking using real-life applications. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 176–188.

[32] Sharan Narang. 2016. Deepbench: open source benchmarking tool for deep learning. Accessed: 2025-01-01. (2016). https://svail.github.io/DeepBench/.

[33] NVIDIA Corporation. 2020. *Kernel Profiling Guide*. Version v2020.2.1. (Oct. 2020). https://docs.nvidia.com/nsight-compute/2020.2/pdf/ProfilingGuide.pdf.

[34] Kyle A O'Connell et al. 2023. Accelerating genomic workflows using nvidia parabricks. *BMC bioinformatics*, 24, 1, 221.

[35] Lawrence Page. 1999. The PageRank citation ranking: Bringing order to the web. Tech. rep. Technical Report.

[36] Reena Panda and Lizy Kurian John. 2017. Proxy benchmarks for emerging big-data workloads. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 105–116.

[37] 2019. *Pytorch: an imperative style, high-performance deep learning library. Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, 12 pages.

[38] J Redmon. 2016. You only look once: unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*.

[39] Karen Simonyan. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[40] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: a revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 7.2.

[41] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 1–9.

[42] Gemini Team et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

[43] Gemma Team et al. 2024. Gemma 2: improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*.

[44] Gemma Team et al. 2024. Gemma: open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*.

[45] Guillaume Thomas-Collignon and Vishal Mehta. [n. d.] Optimizing cuda applications for nvidia a100 gpu. In https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21819-optimizing-applications-for-nvidia-ampere-gpu-architecture.pdf.

[46] 2024. Tiny imagenet challenge. https://www.kaggle.com/c/tiny-imagenet. Accessed: 2024-12-27. (2024).

[47] Hugo Touvron et al. 2023. Llama 2: open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

[48] Hugo Touvron et al. 2023. Llama: open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

[49] Yash Ukidave et al. 2015. Nupar: a benchmark suite for modern gpu architectures. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 253–264.

[50] Peng Wang and Zhibin Yu. 2023. Raybench: an advanced nvidia-centric gpu rendering benchmark suite for optimal performance analysis. *Electronics*, 12, 19, 4124.

[51] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. 2019. Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*.