DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Computational Science and Engineering

Multi-GPU Implementation of a Hyperbolic Finite Volume Solver in ExaHyPE

Siba Zgheib

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Computational Science and Engineering

Multi-GPU Implementation of a Hyperbolic Finite Volume Solver in ExaHyPE

Multi-GPU-Implementierung eines hyperbolischen Finite-Volumen-Lösers in ExaHyPE

Author:	Siba Zgheib
Supervisor:	Prof. Dr. Michael Georg Bader
Advisor:	M.Sc. Mario Wille
Submission Date:	November 15, 2023

I confirm that this Master's Thesis in Computational Science and Engineering is my own work and I have documented all sources and material used.

Munich, November 15, 2023 Siba Zgheib

Acknowledgments

I would like to acknowledge the guidance and support provided by my advisor, Mr. Mario Wille, and my supervisor, Professor Michael Bader, throughout my research. Their insights and assistance have been valuable in shaping my academic journey.

A special thanks to Ahmad Belbeisi for his continuous support and motivation. Ahmad's encouragement has been a driving force, pushing me to do my best.

I also want to express my gratitude to my family for their unwavering emotional support. Their understanding and encouragement have made my academic pursuits possible.

Abstract

This thesis explores the multi-GPU implementation of a patch-based hyperbolic finite volume solver in ExaHyPE 2, a versatile numerical simulation framework for solving hyperbolic partial differential equations. We investigate two approaches to offload patches to the GPU: one utilizing OpenMP and the other employing a hybrid approach involving MPI with OpenMP. These approaches are analyzed and compared to determine their respective benefits and limitations. We then apply one of these methods to benchmark the finite volume Rusanov solver and extend our analysis to simulate the 3D Euler equations using a patch-based enclave solver. All implementations are rigorously benchmarked and tested. Our goal is to provide a comprehensive understanding of the potential of multi-GPU acceleration within the ExaHyPE 2 framework, enabling scientists and engineers to perform simulations more efficiently on large scale machines and gain deeper insights into complex phenomena.

Contents

1 Introduction

In the ever-expanding realm of scientific and engineering simulations, the pursuit of computational accuracy and efficiency remains a driving force. From weather forecasting to astrophysical modeling, from fluid dynamics to seismic wave propagation, numerical simulations have become indispensable tools for understanding and predicting complex physical phenomena. As computational demands grow, HPC technologies must evolve to meet the challenge.

This thesis delves into the exciting domain of ExaHyPE 2, a cutting-edge, highly scalable and adaptable numerical simulation framework designed to address a wide range of problems in science and engineering. At the heart of ExaHyPE 2 lies its hyperbolic finite volume solver, a numerical method capable of solving hyperbolic partial differential equations with high precision. Hyperbolic equations, which govern the propagation of waves and shock waves in various fields, are of paramount importance in many scientific and engineering applications.

In this paper, we explore a critical enhancement to the ExaHyPE 2 framework – the implementation of a multi-GPU approach to accelerate simulations involving the hyperbolic finite volume solver. The integration of multiple GPUs into the computational pipeline represents a groundbreaking development, enabling researchers to harness the immense parallel processing power of modern GPU architectures for solving complex hyperbolic problems.

The motivation behind this research stems from the pressing need for faster and more efficient simulations. Scientists and engineers are increasingly tasked with simulating larger and more intricate systems, necessitating computational power that traditional CPU-based approaches alone can no longer provide. Multi-GPU implementations have emerged as a potent solution to this challenge, offering the prospect of reducing simulation times from weeks or months to mere hours or minutes, while also accommodating greater complexity.

In this paper, we embark on a comprehensive exploration of the multi-GPU implementation within ExaHyPE 2, addressing the challenges, benefits, and implications of this innovative approach. We will investigate two distinct approaches to efficiently benefit from multiple GPUs: one utilizing OpenMP and the other employing a hybrid approach involving MPI with OpenMP. These approaches will be compared and analyzed to determine their respective advantages and limitations.

Furthermore, we will apply one of these methods to benchmark the finite volume Rusanov solver, a critical component in the hyperbolic finite volume solver. We will

also extend our analysis to apply the selected method to the Euler equations using a patch-based enclave solver, offering a holistic assessment of its performance across different simulation scenarios. All implementations will be rigorously benchmarked and tested, ensuring a thorough evaluation of their effectiveness and efficiency.

Our goal is to provide researchers and practitioners with a comprehensive understanding of the potential of multi-GPU acceleration in the ExaHyPE 2 framework and to offer insights into the advantages and trade-offs of different multi-GPU approaches. By achieving this, we hope to catalyze advancements in computational simulations, enabling scientists and engineers to gain deeper insights into complex phenomena and solve real-world problems more efficiently. Through this research, we aim to demonstrate that the multi-GPU implementation of the hyperbolic finite volume solver in ExaHyPE 2 is a transformative step towards unlocking new frontiers in numerical simulation, offering unparalleled computational power and paving the way for breakthroughs in various scientific and engineering disciplines.

In Chapter 2, we delve into an in-depth exploration of related work, focusing on the architecture of ExaHyPE 2 and the GPU offloading mechanisms it employs. Chapter 3 presents our comprehensive methodology, which encompasses the implementation of the *STREAM* triad and Matrix-Matrix multiplication benchmarks using two distinct approaches: a pure OpenMP implementation and a hybrid approach combining MPI with OpenMP. We thoroughly analyze and compare their respective performance.

Chapter 5 delves into the intricate details of our multi-GPU offloading implementation within ExaHyPE 2, specifically applied to a Rusanov kernel benchmark suite. Chapter 6 extends this implementation to encompass the Euler equations, demonstrating its versatility and applicability. Finally, in Chapter 7, we draw our work to a close, offering a comprehensive conclusion and discussing potential avenues for future enhancements and improvements.

2 Related Work

2.1 ExaHyPE 2

ExaHyPE 2, referred to as the Exascale Hyperbolic PDE Engine, stands as a robust computational solution tailored for the intricate task of solving strictly hyperbolic partial differential equations (PDEs) within the context of large-scale simulations [1]. This versatile software harnesses the combined power of MPI and OpenMP for parallelization and is further enhanced with GPU offloading capabilities using the OpenMP target directive [2] , [3]. The orchestration of the computational mesh and dynamic adaptive mesh refinement (AMR) [4] is effectively managed through the Peano framework [5], a sophisticated tool designed to partition Cartesian grids into spacetrees based on the Peano curve. This close integration between Peano and ExaHyPE 2 ensures that it can efficiently scale to meet the demands of the forthcoming exascale computing era.

ExaHyPE 2 is primarily focused on the numerical solution of first-order hyperbolic PDEs, which are fundamental in capturing the dynamics of complex systems. The core equation governing these PDEs is expressed as

$$\frac{\partial}{\partial t}Q + \nabla \cdot F(Q, \nabla Q) + B(Q) \cdot \nabla Q = S(Q) + \sum_{i=1}^{n_{ps}} \delta_i. \tag{2.1}$$

In this equation, Q represents the state variable, $F(Q, \nabla Q)$ encapsulates the intricate fluxes, $B(Q) \cdot \nabla Q$ accommodates the non-conservative products, and $S(Q)$ characterizes the source terms. Additionally, the introduction of point sources δi adds further complexity to the system. To address diverse application scenarios, users are offered the flexibility to incorporate custom solvers. Properly specified boundary and initial conditions, along with maximum eigenvalues for adaptive time-stepping methods, complete the system's requirements.

For the accurate and efficient solution of these PDEs, ExaHyPE 2 relies on a finite volume (FV) scheme. This discretization technique partitions the system into discrete volumes, enabling updates at each discrete time step. The utilization of a first-order Godunov scheme for this purpose is detailed as

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+\frac{1}{2}}^n - F_{i-\frac{1}{2}}^n\right). \tag{2.2}$$

When dealing with the Riemann problem at volume interfaces, the fluxes $F_{j+\frac{1}{2}}^n$ are

determined through dedicated solvers. A typical example is the local Lax-Friedrichs (Rusanov) solver, outlined as

$$F^n_{j+\frac{1}{2}} = \frac{F\left(Q^n_j\right) + F\left(Q^n_{j+1}\right)}{2} + \frac{S\left(Q^n_j\right) + S\left(Q^n_{j+1}\right)}{2} - \max\left(\left|\lambda^{\max}_j\right|, \left|\lambda^{\max}_{j+1}\right|\right) \frac{Q^n_{j+1} - Q^n_j}{2}. \tag{2.3}$$

The judicious selection of solvers and schemes is a critical aspect of achieving the right balance between computational efficiency and solution accuracy. Advanced time integration schemes, such as Runge-Kutta [6], have the potential to enhance precision but require multiple evaluations of the system at various intermediate stages.

In summary, ExaHyPE 2 emerges as an indispensable computational ally for unraveling the dynamics of complex systems governed by hyperbolic PDEs. Its powerful capabilities, adaptable solvers, and effective use of adaptive mesh refinement make it a valuable asset across a wide spectrum of scientific domains, offering a refined and nuanced approach to modeling and simulation.

2.2 Code Architecture

The ExaHyPE 2 software framework operates with dynamically adaptive Cartesian grids, generated through a modified Peano octree method [5]. This process involves iterative subdivision of a cube within the computational domain. To strike a balance between computational workload and grid management, ExaHyPE 2 combines a tree structure with patches, creating an adaptive Cartesian grid. During each time step, the software conducts grid traversals, updating patches through a callback mechanism that abstracts the underlying program logic, data storage, and parallelization details.

ExaHyPE 2's approach to parallelism is tri-fold. It initiates by partitioning the spatial domain into non-overlapping segments, one for each MPI rank. These MPI segments are further divided into subdomains using a space-filling curve (SFC) [7] and then assigned to CPU threads, establishing a hierarchical MPI+X parallel structure. Mesh traversals adhere to a bulk-synchronous processing (BSP) model, with parallelization managed through MPI and OpenMP [8]. It is important to note that the efficiency of this approach can be challenged when dealing with rapidly changing adaptive mesh refinement.

To address load imbalances, each thread identifies patches within its designated subdomain, treating them as individual tasks. Critical patches and those requiring adaptive mesh refinement are integrated into the mesh traversal, while other patches are managed as separate tasks following the enclave tasking concept [9]. This strategy ensures equitable distribution of computational workload among threads during bulk-synchronous processing (BSP) phases, thus addressing load imbalances.

Concerning GPU offloading, enclave tasks are accumulated in a queue until a predetermined threshold of enclave tasks is reached. At this point, all the patches within

the queue are offloaded to the GPU simultaneously, enabling the execution of high-concurrency GPU compute kernels. These tasks within a batch are termed *ready tasks*, allowing for concurrent processing and enhancing internal concurrency. The flexibility to dynamically combine enclave segments ensures efficient GPU offloading, even in cases involving small geometric enclaves distributed across thread subregions.

The choice of the threshold value $\|P_{GPU}\|$ significantly impacts the optimization of GPU parallel potential. Opting for a smaller $\|P_{GPU}\|$ is preferable to effectively utilize the GPU, as it enables multiple threads to concurrently offload to the GPU, thereby leveraging multiple cores [10].

In summary, ExaHyPE 2's software architecture integrates adaptive Cartesian grids with a tree and patch combination. It employs multi-layer parallelism, including MPI+X, and utilizes GPU offloading strategies to maximize GPU performance while addressing load balancing challenges.

2.3 Implementation of GPU Offloading Using `target map`

In our research, we place significant emphasis on the role played by the compute kernels denoted as $\mathcal{K}_p^{\text{Euler,2D}}$ and $\mathcal{K}_p^{\text{Euler,3D}}$. These kernels are instrumental in our computational framework, as they are responsible for advancing the solution over a patch of $p \times p$ or $p \times p \times p$ finite volumes to the subsequent timestep. The entire patch update cycle involves intricate data transfer operations, as elegantly described by Wille et al. in their work on efficient GPU offloading with OpenMP for a hyperbolic finite volume solver on dynamically adaptive meshes [11].

The data transfer process can be concisely represented by the composition of various operators:

$$(\mathcal{R} \circ \mathcal{F} \circ \mathcal{K} \circ \mathcal{A} \circ \mathcal{P})Q(t)$$

In this framework, the operator $\mathcal{P}$ is responsible for transmitting the solution $Q(t)$ to the GPU, while $\mathcal{R}$ retrieves it from the GPU back to the user's memory. Additionally, the operator $\mathcal{A}$ manages all the temporary variables residing on the GPU that are required by $\mathcal{K}$, and $\mathcal{F}$ handles the deallocation of these temporary memory blocks. The ExaHyPE 2 framework orchestrates these operations across a set of patches as follows:

$$\{(\mathcal{R} \circ \mathcal{F} \circ \mathcal{K} \circ \mathcal{A} \circ \mathcal{P})Q_c(t)\}_{c \in [1, \|\mathbb{P}_{\text{GPU}}\|]}$$

The practical procedure for implementing GPU offloading using OpenMP, as elucidated by Wille et al. in their research, follows a systematic approach:

1. Initially, data originating from patches is organized into a comprehensive array on the host in an Array of Structures (AoS) format. Although this data is initially stored in the main memory, it is meticulously moved to the GPU. This transfer

predominantly encompasses a substantial, contiguous array consisting of double-precision values. Following this, a systematic process on the GPU involves the creation and population of a list of pointers with device pointers. While OpenMP's declare mapper constructs are capable of managing this operation, it is recommended to utilize `omp_get_mapped_ptr` for improved efficiency.

2. The computational kernel efficiently allocates all the temporary data required for patch handling into a single, large memory block. This allocation process is seamlessly performed using the `map(alloc:...)` construct.

3. A kernel's execution becomes more straightforward when you incorporate an `omp target` block alongside a `distribute` directive.

4. Once the kernel finishes its execution, all the temporary data is released collectively.

5. Using OpenMP's `map` clauses, the results obtained from the GPU are methodically moved back into the host's memory. This process is carried out iteratively, managing each patch and timestep using the `omp target exit data map(from:...)` construct.

This methodical process, as outlined by Wille et al., offers a comprehensive approach to GPU offloading with OpenMP, ensuring efficient data transfer and parallel execution of computational kernels.

2.4 An Examination of Multi-GPU Configurations

The deployment of multiple GPUs in numerical simulations within the realm of high-performance computing represents a transformative leap in the capabilities of modern research endeavors. This innovation revolutionizes the way we conduct numerical simulations and computations, addressing some of the most pressing challenges faced by scientists and engineers. Multi-GPU usage is particularly instrumental in accelerating research processes, as it enables parallel processing on a grand scale. This means that computations can be divided into smaller, manageable tasks and distributed across multiple GPUs, each working concurrently on its portion of the problem. The result is a remarkable reduction in simulation runtimes, transforming problems that would have taken weeks or even months into tasks that can be completed in a fraction of the time.

In addition to speed, the introduction of multi-GPU configurations broadens the scope and depth of scientific exploration. Complex problems, such as climate modeling, high-energy physics, molecular dynamics, and quantum chemistry, often require immense computational resources to simulate real-world scenarios accurately. The collective power of multiple GPUs makes it feasible to simulate more extensive and detailed models, leading to a better understanding of intricate phenomena. Researchers can now

delve deeper into simulations with finer spatial and temporal resolutions, providing a more faithful representation of the physical world. This enhanced fidelity equips scientists with the means to make more precise predictions, uncover hidden patterns, and gain valuable insights into various fields of study.

Furthermore, as the demand for high-performance computing continues to grow across academic, industrial, and governmental sectors, multi-GPU setups are becoming increasingly indispensable. They offer a cost-effective solution to meet the escalating computational requirements of modern research, allowing institutions to achieve their scientific goals without incurring prohibitive infrastructure costs. In essence, the use of multiple GPUs in numerical simulations empowers researchers to push the frontiers of scientific discovery and innovation, fostering breakthroughs that were once considered unattainable. It is a pivotal tool in the arsenal of high-performance computing, propelling us towards a new era of computational science.

In [12], a novel approach is introduced for harnessing the collective power of multiple GPUs in large-scale scientific computations. Central to this approach is the implementation of an advanced domain decomposition technique designed to efficiently distribute computational workloads across multiple GPUs. To address the challenges associated with communication overhead in multi-GPU configurations, the authors utilize a combination of MPI and OpenACC, ensuring effective inter-GPU communication. Benchmark tests conducted across various computational scenarios demonstrate the efficacy of this approach. Notably, for the three-dimensional wave equation, the multi-GPU implementation achieved a remarkable speedup of up to 3.5 times compared to a single GPU setup, and for the Lattice Boltzmann Method, a substantial speedup of approximately 4 times was observed. In real-world applications spanning fluid dynamics and quantum mechanics, the proposed approach consistently outperformed traditional single-GPU solutions, especially in computationally intensive scenarios. These results underscore the transformative potential of multi-GPU configurations in scientific computing and emphasize the critical importance of optimized communication and workload distribution strategies to unlock the full capabilities of multiple GPUs, paving the way for future advancements in high-performance computing.

In [13], the authors introduce a dynamic strategy known as the hybrid approach, which combines the collaborative utilization of both CPUs and multiple GPUs to tackle large-scale scientific computations. This approach recognizes the diverse computational requirements of various tasks, optimizing the use of GPUs for parallelizable operations, such as matrix multiplications and Fourier transforms, which multiple GPUs can efficiently handle. In situations where tasks demand sequential processing or intricate branching, the flexible architecture of CPUs takes the lead to ensure effective execution. The hybrid approach maximizes the strengths of each component while mitigating their respective weaknesses, ultimately enhancing computational efficiency and enabling high-performance computing for a wide range of scientific simulations.

This versatile approach not only involves the synergistic use of CPUs and GPUs but

also extends to the deployment of multiple GPUs, creating a multifaceted solution to address the complexities of large-scale scientific computations. Task delegation within the hybrid model optimally leverages the parallel processing capabilities of multiple GPUs for tasks requiring parallel execution, thereby reducing simulation runtimes and enabling the exploration of more extensive and intricate scientific questions. However, the architecture of GPUs, optimized for parallelism, may introduce bottlenecks for tasks demanding sequential processing or complex branching. In such scenarios, CPUs with their adaptable architecture seamlessly manage these tasks, ensuring efficient execution without the constraints of GPU parallelism. This holistic approach recognizes that different computational tasks possess unique requirements and allocates them optimally to the appropriate computing resources. By harnessing the raw computational power of multiple GPUs and the algorithmic prowess of CPUs, the hybrid approach emerges as a balanced, efficient, and scalable solution that addresses the diverse computational needs of scientific simulations. To facilitate efficient communication between CPUs and multiple GPUs within the hybrid system, the paper presents advanced data transfer protocols designed to streamline information flow among these components, ensuring seamless collaboration. These protocols prioritize data transfers, maintain data integrity, and dynamically allocate communication bandwidth, reducing potential bottlenecks. This robust CPU-GPU communication, combined with multi-GPU usage, paves the way for a new era in high-performance computing, providing an optimal solution to meet the increasing demands of large-scale scientific simulations.

The research paper [14] introduces a novel approach to leverage both CPUs and GPUs for large-scale scientific computations, with a specific focus on the Navier-Stokes equations in fluid dynamics. At its core, the study emphasizes the concept of multi-level parallelism, effectively harnessing fine-grained parallelism within GPU cores and coarse-grained parallelism across multiple GPU cores.

To implement this intricate architecture, the researchers employed a trio of advanced technologies: OpenMP, MPI, and CUDA. OpenMP was utilized to tap into the multi-core capabilities of CPUs, ensuring efficient load distribution across CPU cores. MPI played a vital role in managing inter-node communication in simulations spanning multiple compute nodes, optimizing data decomposition and result aggregation. Crucially, the GPU computations were driven by CUDA, offering a platform to design and execute parallel algorithms tailored to GPU architectures. The researchers intricately integrated CUDA kernels into their methodology, optimizing tasks like matrix operations and exploiting CUDA's memory hierarchy for peak performance.

The study's findings, derived from an array of benchmark tests and real-world scenarios, highlight the prowess of the hybrid system, underpinned by OpenMP, MPI, and CUDA. Results show a substantial reduction in computation times for fluid dynamics simulations, with the proposed method achieving up to an eightfold speedup when compared to conventional GPU-based approaches.

In essence, this research bridges advanced parallel computing techniques with the

specific requirements of the Navier-Stokes equations, capitalizing on the capabilities of OpenMP,PI, and CUDA. It marks a transformative path within the realm of high-performance computing for fluid dynamics.

The research paper [15], authored by Schive et al. delves into the development and optimization of hydrodynamic algorithms tailored for AMR simulations. Recognizing the computational challenges posed by hydrodynamic AMR simulations, the authors introduce a directionally unsplit hydrodynamic scheme, aiming to enhance both accuracy and efficiency.

Central to the research's implementation is a harmonious blend of parallel computing technologies: MPI, OpenMP, and CUDA. MPI is employed to manage inter-node communication, ensuring efficient data decomposition and distribution across multiple computer nodes in large-scale simulations. OpenMP, with its shared-memory parallel programming capabilities, optimizes task distribution across multi-core CPUs, enhancing the efficiency of operations best suited for CPU execution. NVIDIA's CUDA platform emerges as the linchpin for GPU-related computations. Using CUDA, the authors developed parallel algorithms optimized for tasks inherent to hydrodynamic computations, such as flux evaluations and data reconstruction.

The results from the study are illuminating. The introduced hydrodynamic scheme, fortified by the hybrid MPI-OpenMP-GPU parallelization approach, showcases a significant improvement in computational performance. Benchmark tests reveal that the proposed methodology not only reduces computational times but also enhances the accuracy of AMR simulations. Specifically, when compared to traditional methods, the hybrid approach achieves speedups of several magnitudes, underscoring its potential to revolutionize hydrodynamic AMR simulations.

In conclusion, [15] stands as a testament to the transformative power of blending advanced hydrodynamic schemes with cutting-edge parallel computing techniques. By integrating the strengths of MPI, OpenMP, and CUDA, the study paves the way for faster, more accurate, and efficient hydrodynamic AMR simulations, setting a new benchmark in the realm of computational fluid dynamics.

The previously referenced studies underscore the significance of adopting a multi-GPU approach for large-scale simulations. Given that many of these studies employ a hybrid methodology, we will conduct a detailed investigation and analysis of two distinct approaches. The first is a pure OpenMP offloading approach, which leverages OpenMP for data offloading, data distribution, and kernel computations. The second approach involves a hybrid MPI with OpenMP strategy, where MPI is employed for data distribution, and OpenMP is utilized for offloading and kernel computations. Both approaches will undergo a comprehensive comparative analysis to determine the superior strategy for implementation in ExaHyPE 2.

3 Methodology

In this section, we investigate two prospective methodologies for GPU offloading: a pure OpenMP approach and a hybrid approach that integrates OpenMP and MPI. We employ these methodologies to assess their performance on two benchmark types: a memory-bound benchmark, namely, *STREAM* triad, and a compute-bound benchmark, which involves matrix-matrix multiplication. The primary aim of this investigation is to determine the most suitable approach for implementation within ExaHyPE 2.

3.1 Compute-Bound and Memory-Bound Problems in High-Performance Computing

Compute-bound and memory-bound problems are two fundamental categories in high-performance computing, each with its unique characteristics.

Compute-bound problems are characterized by a situation where the primary bottleneck in performance lies in the processing power of the CPU or GPU. In such cases, the processor is predominantly engaged in intensive calculations and computations. Memory access is not the limiting factor, and optimization strategies for compute-bound problems typically focus on enhancing algorithm and code efficiency to maximize the utilization of available computational resources.

On the other hand, memory-bound problems are those in which the principal performance constraint arises from the time required to fetch data from memory. This leads to extended periods of processor idleness as it waits for data to be loaded. In memory-bound scenarios, optimization strategies are aimed at improving memory access patterns and reducing data transfer to enhance overall application efficiency.

Subsequent sections will deliver a more in-depth analysis of both problem types, encompassing the execution of the STREAM triad and matrix-matrix multiplicaiton benchmarks. These sections will provide intricate insights into their implementation, multi-GPU integration, and performance scalability.

3.2 The Roofline Model and Arithmetic Intensity

The Roofline model is a valuable framework used to assess and compare the performance capabilities of compute-bound and memory-bound problems. It features two key components:

1. *Arithmetic Intensity:* This is a fundamental metric within the Roofline model that quantifies the relationship between arithmetic operations (such as additions and multiplications) and memory accesses in a workload. High arithmetic intensity indicates a scenario where computation dominates memory access, while low arithmetic intensity signifies a disproportionate emphasis on data retrieval and the resulting underutilization of computational resources. Equation 3.1 represents a general formula used to compute arithmetic intensity.

$$\text{Arithmetic Intensity (AI)} = \frac{\text{Number of Arithmetic Operations}}{\text{Number of Memory Accesses}} \tag{3.1}$$

2. *Roofline Graph:* The Roofline model includes a *roofline graph* that represents the upper bound of achievable performance. This graph serves as a visual reference for the performance capabilities of a given hardware configuration. It helps identify the maximum performance attainable and any performance bottlenecks that might exist.

To compare compute-bound and memory-bound problems using the Roofline model and arithmetic intensity, one can analyze how these problems perform in relation to the roofline graph. Compute-bound problems will tend to have higher arithmetic intensity, indicating that computation dominates memory access. Memory-bound problems, on the other hand, will exhibit lower arithmetic intensity, highlighting the importance of memory access.

In the context of the problem at hand, we compute the arithmetic intensity of the triad operation as

$$\textit{Arithmetic Intensity (AI)} = \frac{1}{12}\,, \tag{3.2}$$

since there are 2 flops per iteration and 24 bytes of memory transfers for every iteration [16]. Similarly, we calculate the arithmetic intensity of matrix-matrix multiplicaiton as

$$\textit{Arithmetic Intensity (AI)} = \frac{M \cdot N \cdot K}{M \cdot K + N \cdot K + M \cdot N}\,, \tag{3.3}$$

since the product of B and C has M x N values, each of which is a dot-product of K-element vectors [17].

The utilized GPU is the RTX3080 Turbo, featuring a theoretical peak memory bandwidth of 760.3 GB/s and a theoretical peak double-precision performance of 465.1 GFLOPS. Leveraging this information, we construct the Roofline model depicted in Figure 3.1, employing matrices with identical dimensions (N=M=K=2^{14}). This visual representation reveals that the STREAM triad operation is memory-bound, while the matrix-matrix multiplication operation is compute-bound.

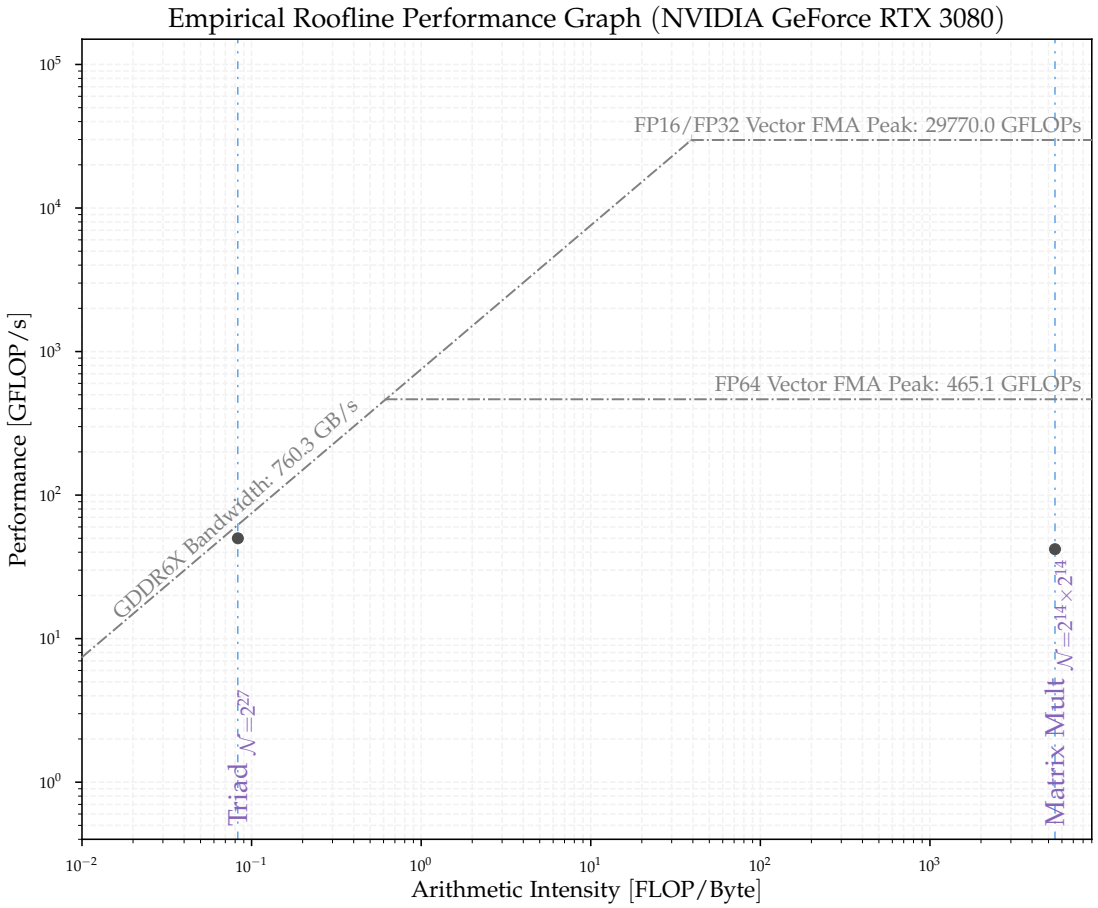


Figure 3.1: Empirical Roofline model for the *STREAM* triad and the matrix-matrix multiplication benchmark on a RTX3080.

3.3 STREAM Benchmark

The STREAM benchmark's [18] triad operation serves as the foundation for evaluating two distinct multi-GPU implementation methods. The triad operation encompasses three arrays: A, B, and C. It executes the element-wise operation for each array element as shown in Equation 3.4:

$$A[i] = B[i] + \text{scalar} \cdot C[i] \tag{3.4}$$

Primarily a memory-bound arithmetic operation, the triad benchmark gauges the system's bandwidth. We plotted the roofline model of the triad benchmark in Figure 3.1, and it can be seen that it is indeed a memory-bound problem. Within our code, the array size was altered from 2^6 to 2^{27}, subsequently determining the MFLOPS for each size through Equation 3.5:

$$MFLOPS = 2.0 \cdot STREAM\_ARRAY\_SIZE \cdot NTIMES \cdot \frac{1.0 \times 10^{-6}}{duration} \tag{3.5}$$

Here, *STREAM_ARRAY_SIZE* represents the array's size, *NTIMES* the computation repetition count, and *duration* the time interval between the triad computation's commencement and conclusion. The bandwidth is computed by multiplying the *MFLOPS* by 12, accounting for the memory usage of the three double arrays.

Arrays B and C were initialized using their respective indices, and the output array A was validated by comparing its sum against the sequential triad operation's results. The benchmark's first implementation approach solely utilized OpenMP for data distribution and kernel offloading. In contrast, the second method combined MPI for data distribution with OpenMP for kernel offloading. Both methods underwent testing for strong and weak scaling across four GPUs.

3.3.1 OpenMP *target* Offloading

In our first approach, we leverage OpenMP offloading for both kernel execution and data distribution. Our process begins with the allocation of the uninitialized arrays A, B, and C, each of size *CHUNK_SIZE*, on the GPUs. Subsequently, we proceed to initialize B and C on the GPUs using their respective indices. Algorithm 1 illustrates the allocation and initialization. In our specific case, *STREAM_TYPE* is of double precision, *OFFSET* is utilized for padding and is set to zero within the code, while *CHUNK_SIZE* is responsible for computing the appropriate chunk size for the arrays, taking into consideration the number of GPUs in use (*GPU_COUNT*). It also accounts for situations where the array size is not evenly divisible by the number of GPUs. To ensure accurate initialization across the various GPUs, we iterate through the devices, establish distinct sections, and offload the initialization to a dedicated team of threads.

Once the GPU arrays are initialized, Algorithm 2 outlines the procedure for performing the triad operation, all while monitoring the execution time based on wall time

Algorithm 1 OpenMP *target* triad allocation and initialization

Require: $STREAM\_ARRAY\_SIZE, NTIMES$

```
a ← new STREAM_TYPE[STREAM_ARRAY_SIZE + OFFSET]
b ← new STREAM_TYPE[CHUNK_SIZE + OFFSET]
c ← new STREAM_TYPE[CHUNK_SIZE + OFFSET]
scalar ← 2.0
CHUNK_SIZE ← (STREAM_ARRAY_SIZE + GPU_COUNT - 1) / GPU_COUNT
for i from 0 to GPU_COUNT do
    Target enter data map(alloc:a[0:CHUNK_SIZE],b[0:CHUNK_SIZE],c[0:CHUNK_SIZE])
device(i)
end for
Parallel NUM_THREADS(GPU_COUNT)
Sections
for j from 0 to GPU_COUNT do
    Section
    if GPU_COUNT == 1 then
        Target device(j)
    else
        Target device(j) nowait
    end if
    Teams
    Distribute parallel for
    for i from 0 to CHUNK_SIZE do
        b[i] ← c[i] ← i + j × CHUNK_SIZE
    end for
end for
```

measurements. Following the computation, the resultant array A is transferred back to the CPU, and the GPU memory resources are freed for arrays B and C. Subsequently, for each GPU, the elements of array A are aggregated through the **calculateResult** function to validate the results. Finally, the MFLOPS are computed using the **calculateMFLOPS** function.

Algorithm 2 OpenMP *target* triad kernel

Require: $STREAM\_ARRAY\_SIZE$, $NTIMES$
start ← **omp_get_wtime**()
Parallel NUM_THREADS(GPU_COUNT)
Sections
for *k* **from** 0 **to** GPU_COUNT **do**
    Section
    if GPU_COUNT == 1 **then**
        Target device(k)
    else
        Target device(k) **nowait**
    end if
    Teams
    for *j* **from** 0 **to** NTIMES **do**
        Distribute parallel **for**
        for *i* **from** 0 **to** CHUNK_SIZE **do**
            a[i] ← b[i] + scalar × c[i]
        end for
    end for
end for
stop ← **omp_get_wtime**()
duration ← stop - start
triad_result ← 0
for *i* **from** 0 **to** GPU_COUNT **do**
    Target exit data
    map(from:a[0:CHUNK_SIZE]) **map**(release:b[0:CHUNK_SIZE],c[0:CHUNK_SIZE]) device(i)
    triad_result ← triad_result + **calculateResult**(CHUNK_SIZE, a)
end for
triad_mflops ← **calculateMFLOPS**(STREAM_ARRAY_SIZE, duration, NTIMES)
Delete[] a, b, c

Observation 1 *Using OpenMP sections improves performance*: The sections construct is a non-repetitive work-sharing feature that encompasses a collection of organized blocks. These blocks are meant to be distributed among the threads in a team and

executed. Within this construct, each structured block is executed exactly once by one of the threads in the team, within the context of its implicit task. This construct can enhance performance through the facilitation of concurrent execution of code blocks, maintenance of equitable thread workload distribution, provision of execution order flexibility, scalability with the number of accessible processor cores, and streamlining the creation of parallel code.

Observation 2 *Using nowait with one GPU leads to incorrect and random results:* In OpenMP, using the *nowait* clause with a single GPU target construct can lead to incorrect and random results because it allows for asynchronous execution, enabling the CPU to continue processing without waiting for the GPU to finish its tasks. This can result in synchronization issues and data race conditions, as the CPU and GPU may operate on the same data simultaneously, leading to unpredictable and incorrect outcomes. To ensure correct synchronization when using a single GPU in OpenMP, you should avoid *nowait* and rely on the implicit synchronization provided by the target construct to guarantee that the GPU operations complete before continuing with CPU code that depends on the results.

3.3.2 Hybrid MPI with OpenMP *target* Offloading

The second approach involves utilizing MPI for data distribution and OpenMP for kernel offloading. As shown in Algorithm 3, the central concept is to designate distinct MPI ranks as GPU identifiers, with each rank corresponding to a specific GPU. For instance, rank 0 corresponds to GPU 0. In contrast to the pure OpenMP approach, the arrays B and C are initially instantiated on the CPU (specifically, on rank 0) with their associated indices. Subsequently, employing the *MPI_Scatter* directive, the arrays are distributed to local arrays within each rank, with each local array having a size of *CHUNK_SIZE*. The value of *CHUNK_SIZE* is calculated based on the overall array size and the number of GPUs employed. Following this distribution, OpenMP is employed to map the local arrays onto the GPUs, and the triad kernel is offloaded for computation.

Once the computation is completed, the local arrays B and C are released from GPU memory, and the local array A is remapped back to the CPU, where each chunk is gathered into the global array A. Similar to the OpenMP implementation, the verification and MFLOP calculation processes are executed within their respective functions.

3.4 Matrix-Matrix Multiplication

Consider the problem of matrix-matrix multiplicaiton, where we have two input matrices, B of size $(M \times K)$ and C of size $(K \times N)$. We aim to compute the resulting matrix A of size $(M \times N)$ as shown in Equation 3.6 using the block (column) distribution

Algorithm 3 Hybrid MPI with OpenMP *target* triad kernel

Require: $STREAM\_ARRAY\_SIZE$, $NTIMES$, $rank$, $NUM\_PROCESSES$

Initialize MPI
Parse STREAM_ARRAY_SIZE
Calculate CHUNK_SIZE based on STREAM_ARRAY_SIZE and NUM_PROCESSES
Initialize scalar
Allocate memory for a, b, and c (rank 0)
Initialize b and c (rank 0)
Scatter b and c to local_b and local_c (all ranks)
Target data
map(tofrom: local_a[0:CHUNK_SIZE]) **map** (to:local_b[0:CHUNK_SIZE],local_c[0:CHUNK_SIZE]) device(rank)
Target device(rank)
Teams
for $j = 0$ to $NTIMES$ **do**
    Distribute parallel for
    for $i = 0$ to $CHUNK\_SIZE$ **do**
        Compute $local\_a[i] = local\_b[i] + scalar \cdot local\_c[i]$
    end for
end for
Gather local_a to a (rank 0)
Target exit data
map(from:a[0:CHUNK_SIZE]) **map**(release:b[0:CHUNK_SIZE],c[0:CHUNK_SIZE]) device(rank)
if $rank = 0$ **then**
    triad_result ← **calculateResult**(CHUNK_SIZE, a)
    triad_mflops ← **calculateMFLOPS**(STREAM_ARRAY_SIZE, duration, NTIMES)
    Delete[] a, b, c
end if
Release memory for local_a, local_b, and local_c
Finalize MPI

method.

$$A[i,j] = \sum_{k} B[i,k] \cdot C[k,j] \tag{3.6}$$

The block (column) distribution method is designed for efficient parallel computation of the matrix product (Equation 3.6). This technique involves dividing the input matrices and distributing the computation across multiple processors, as described below:

1. **Input Matrices:** We have two input matrices, B and C, with dimensions as described above.
2. **Processor Assignment:** We have GPU_COUNT GPUs available, where GPU_COUNT is typically less than or equal to the number of columns in matrix C (i.e., N).
3. **Data Partitioning:**
   - Matrix B remains intact on all GPUs, and each GPU stores a full copy of matrix B.
   - Matrix C is divided into GPU_COUNT vertical columns. Each GPU is assigned a specific subset of columns from matrix C, according to Equation 3.7, where j is the column.

$$GPU\_ID = \frac{j \cdot GPU\_COUNT}{N} \tag{3.7}$$

4. **Computation:**
   - Each GPU is responsible for calculating a portion of the resulting matrix A.
   - Specifically, each GPU computes one or more columns of A, corresponding to the columns of C it owns.
   - The multiplication is performed for the full matrix B but only against the columns of B that the GPU has.
5. **Communication and Aggregation:**
   - Once all GPUs have completed their individual calculations, the partial results need to be aggregated to form the final matrix A.

3.4.1 OpenMP *target* Offloading

To implement OpenMP target offloading for utilizing multi-GPUs in the matrix-matrix multiplication benchmark, we employ a process analogous to that employed for the triad operation. Initially, we allocate and initialize the matrices `local_a`, `b`, and `c` on the GPUs, as demonstrated in Algorithm 4. The sizes of `local_a` and `c` are determined based on

the dimensions of the matrix N and the number of GPU devices GPU_COUNT. Matrix `b` is entirely distributed to each device.

The kernel for matrix-matrix multiplication is depicted in Algorithm 5, where we iterate over the devices and compute a segment of the resulting matrix in each iteration. The complete resultant matrix a is subsequently reconstructed by mapping the segmented portions of a back to the CPU and aggregating them into a.

Algorithm 4 OpenMP *target* matrix-matrix multiplication allocation and initialization

Require: N

```
1: a ← new double[N]
2: b ← new double[CHUNK_SIZE]
3: c ← new double[CHUNK_SIZE]
4: local_a ← new double[CHUNK_SIZE]
5: CHUNK_SIZE ← (N + GPU_COUNT - 1) / GPU_COUNT
6: for k from 0 to GPU_COUNT do
7:     Target enter data map(alloc:local_a[0:CHUNK_SIZE*N],b[0:N*N],c[0:CHUNK_SIZE*N])
   device(k)
8: end for
9: Parallel NUM_THREADS(GPU_COUNT)
10: Sections
11: for k from 0 to GPU_COUNT do
12:     Section
13:     if GPU_COUNT == 1 then
14:         Target device(k)
15:     else
16:         Target device(k) nowait
17:     end if
18:     Teams
19:     Distribute parallel for
20:     for i from 0 to N do
21:         for j from 0 to N do
22:             b[i * N + j] ← 2.0
23:         end for
24:         for j from 0 to N do
25:             c[j * N + i] ← 4.0
26:         end for
27:     end for
28: end for
```

Algorithm 5 OpenMP *target* matrix-matrix multiplication kernel

Require: N

start ← **omp_get_wtime**()
Parallel NUM_THREADS(GPU_COUNT)
Sections
for l **from** 0 **to** GPU_COUNT **do**
    Section
    if GPU_COUNT == 1 **then**
        Target device(l)
    else
        Target device(l) **nowait**
    end if
    Teams Distribute parallel **for**
    for $i = 0$ to N **do**
        for $j = 0$ to *chunk_size* **do**
            for $k = 0$ to N **do**
                $local\_a[j \cdot N + i] += b[i \cdot N + k] \cdot c[j \cdot N + k]$
            end for
        end for
    end for
end for
stop ← **omp_get_wtime**()
duration ← stop - start
triad_result ← 0
for i **from** 0 **to** GPU_COUNT **do**
    Target exit data
    map(from:local_a[0:CHUNK_SIZE]) **map**(release:b[0:CHUNK_SIZE],c[0:CHUNK_SIZE]) device(i)
    triad_result ← triad_result + **calculateResult**(CHUNK_SIZE, a)
end for
Parallel for
for i **from** 0 **to** $N - 1$ **do**
    for j **from** 0 **to** chunk_size $- 1$ **do**
        $a[(j + \text{start\_col}) \cdot N + i] \mathrel{+}= \text{local\_a}[j \cdot N + i]$
    end for
end for
matmul_mflops ← **calculateMFLOPS**(N, duration)
Delete[] a, b, c, local_a

3.4.2 Hybrid MPI with OpenMP *target* Offloading

In emulation of the triad operation, we have embraced methodologies for the multi-GPU offloading of matrix-matrix multiplication, employing a hybrid MPI with OpenMP target offloading. To realize this, we establish a uniform mapping of each rank to a designated target device. The pseudo-code of the hybrid MPI with OpenMP target offloading is shown in Algorithm 6. Notably, its structure closely mirrors the implementation of the triad operation, while adopting the algorithm details of the block (column) distribution method mentioned above.

Algorithm 6 Hybrid MPI with OpenMP *target* matrix-matrix multiplication kernel

Require: N, $rank$, $NUM\_PROCESSES$
1: Initialize MPI
2: Parse N
3: Calculate $CHUNK\_SIZE$ based on N and $NUM\_PROCESSES$
4: Define a, b, c, $local\_a$
5: **Target** enter data **map**(alloc: $local\_a[0 : CHUNK\_SIZE * N]$, $b[0 : N * N]$, $c[0 : CHUNK\_SIZE * N]$) device($rank$)
6: Initialize a, b, c on GPU
7: **Target** device(rank)
8: **Teams**
9: **Distribute** parallel for
10: **for** $i = 0$ to N **do**
11: **for** $j = 0$ to $chunk\_size$ **do**
12: **for** $k = 0$ to N **do**
13: $local\_a[j \cdot N + i] += b[i \cdot N + k] \cdot c[j \cdot N + k]$
14: **end for**
15: **end for**
16: **end for**
17: **Target** data **map**(from: $local\_a[0 : CHUNK\_SIZE * N]$) **map**(release: $b[: N * N]$, $c[0 : CHUNK\_SIZE * N]$) device($rank$)
18: Gather local_a to a (rank 0)
19: **if** $rank = 0$ **then**
20: matmul_mflops $\leftarrow$ **calculateMFLOPS**(N, duration)
21: **end if**
22: **Delete**[] a, b, c, local_a
23: Finalize MPI

4 Discussion

4.1 Evaluation of the Implementations

Taking the above benchmarks into consideration, when transitioning from single GPU utilization to a multi-GPU configuration, the process can become quite intricate, particularly when relying on pure OpenMP offloading. Several critical factors come into play:

- A for loop must be constructed to iterate through the GPU devices for the precise implementation of *device(GPU_ID)*.
- For a more optimized approach, it is recommended to use *sections*, which necessitates the addition of extra lines of code.
- To accommodate single GPU usage, conditions need to be added for the *nowait* construct, which can potentially impact overall performance.
- Diligent attention is required for data synchronization and distribution to and from the GPUs.

In contrast, when employing a hybrid MPI and OpenMP approach, there is no need for explicit device ID specification within for loops. Moreover, data distribution and synchronization are significantly streamlined through the utilization of built-in MPI features such as scatter and gather.

4.2 Benchmarking

The benchmarks were run on the Home One cluster. The compute node is powered by a single AMD EPYC 7402 CPU, offering 24 cores and 2-way hyper-threading, alongside 256 GB of RAM. It has four RTX 3080 Turbo GPUs, each providing 10 GB of video RAM, and a 512 GB NVMe PCIe Gen4 SSD. Both nodes operate on the Ubuntu 20.04.2 LTS operating system with kernel version 5.4.0-81. Each GPU has a theoretical peak bandwidth of 760.3 GB/s and a theoretical peak performance of 465.1 GFLOP/s for double precision. We tested weak scaling and strong scaling scenarios.

4.2.1 STREAM Triad

Performance Evaluation

To assess the performance of various computational approaches, we conducted a comprehensive analysis by manipulating the array size, ranging from 2^6 to 2^{27}, for different GPU configurations, including 1, 2, 3, and 4 GPUs. In Figures 4.1 and 4.2, we present the results, depicting GFLOP/s and bandwidth (GB/s) as functions of data size for each GPU configuration.

For smaller array sizes within the range of 10^2 to 10^5 on a logarithmic scale, the bandwidth exhibited a consistent trend across all GPU configurations, with minimal variations. However, as the array size increased, a conspicuous divergence emerged among the different GPU setups, with the four-GPU configuration achieving the highest bandwidth. This observation aligns with expectations, as larger data sizes mitigate communication overhead, allowing GPUs to leverage their computational power more effectively.

Comparing the performance of MPI and OpenMP, we note a discernible distinction. OpenMP outperformed MPI, reaching a peak bandwidth of 2400 GB/s, while MPI attained a maximum of 1950 GB/s for an array size of 2^{27}. This contrast underscores the importance of selecting the appropriate parallelization strategy, as it can significantly impact the computational efficiency of the system.

Weak Scaling

In the realm of high-performance computing, *weak scaling* serves as a pivotal metric for assessing the performance and efficiency of parallel computing systems. It evaluates how well a system can handle an increase in both the problem size and the number of processing units in a proportional manner. In essence, weak scaling is about maintaining a consistent workload per processing unit as computational demands and resources expand. The goal is to ensure that the system's performance remains stable and efficient throughout this scalability process.

This metric is often quantified as a ratio comparing the execution time required to complete a fixed amount of work, like a specific problem size, on a single processing unit to the execution time when both the problem size and the number of processing units are increased in a proportional fashion. In our particular case, this concept was put into action by progressively increasing the array size from 2^{24} to 2^{27}, all the while adding one GPU for each doubling of the array size. The outcome, as illustrated in 4.3 and 4.4, portrays a nearly ideal manifestation of weak scaling, characterized by a linear increase in bandwidth.

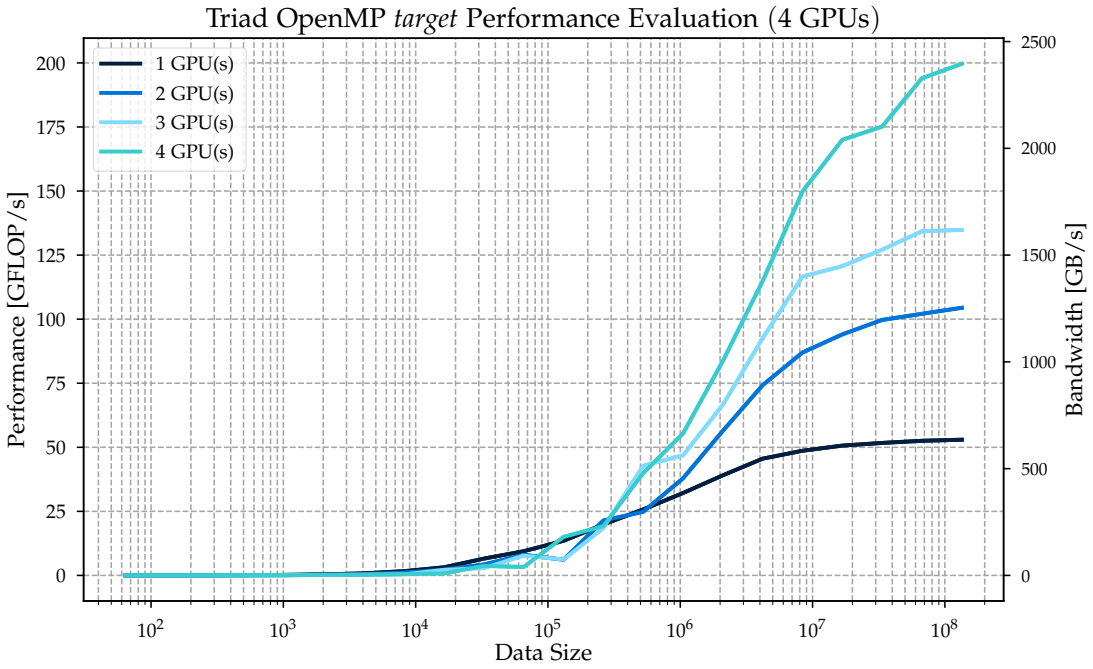


Figure 4.1: Variation in the performance (GFLOP/s) and bandwidth (GB/s) of the OpenMP *target* implementation of the triad benchmark (GB/s) with respect to the data size, tested across an increasing number of GPUs (1 to 4).

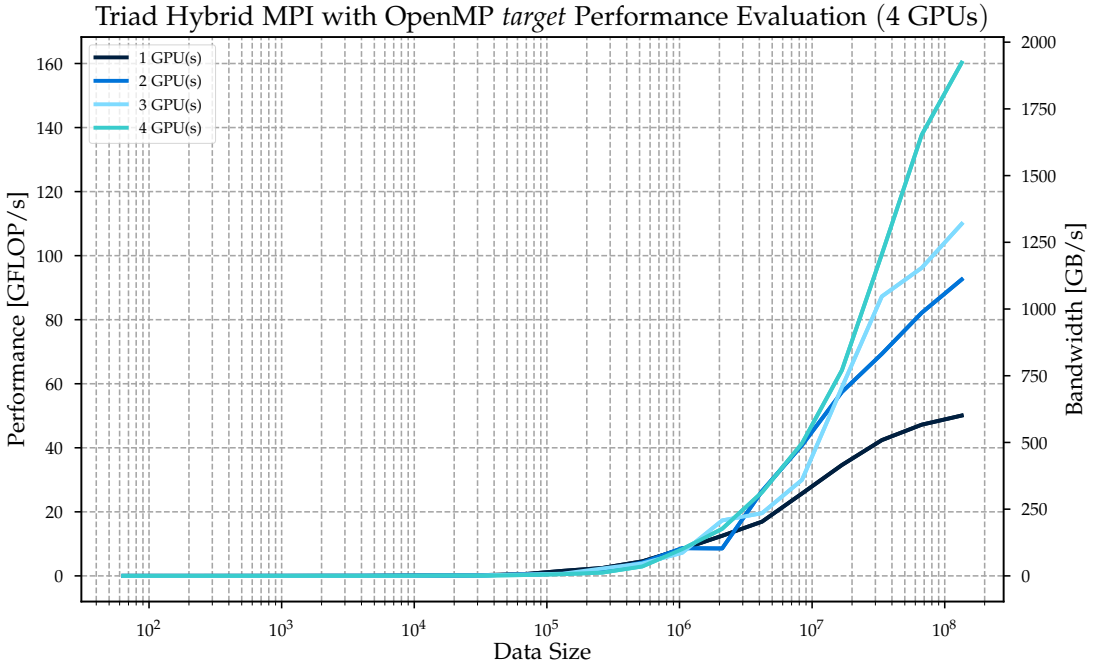


Figure 4.2: Variation in tghe performance (GFLOP/s) and bandwidth (GB/s) of the Hybrid MPI with OpenMP *target* implementation of the triad benchmark, as a function of the data size. The evaluation includes tests on an increasing number of GPUs (1 to 4).

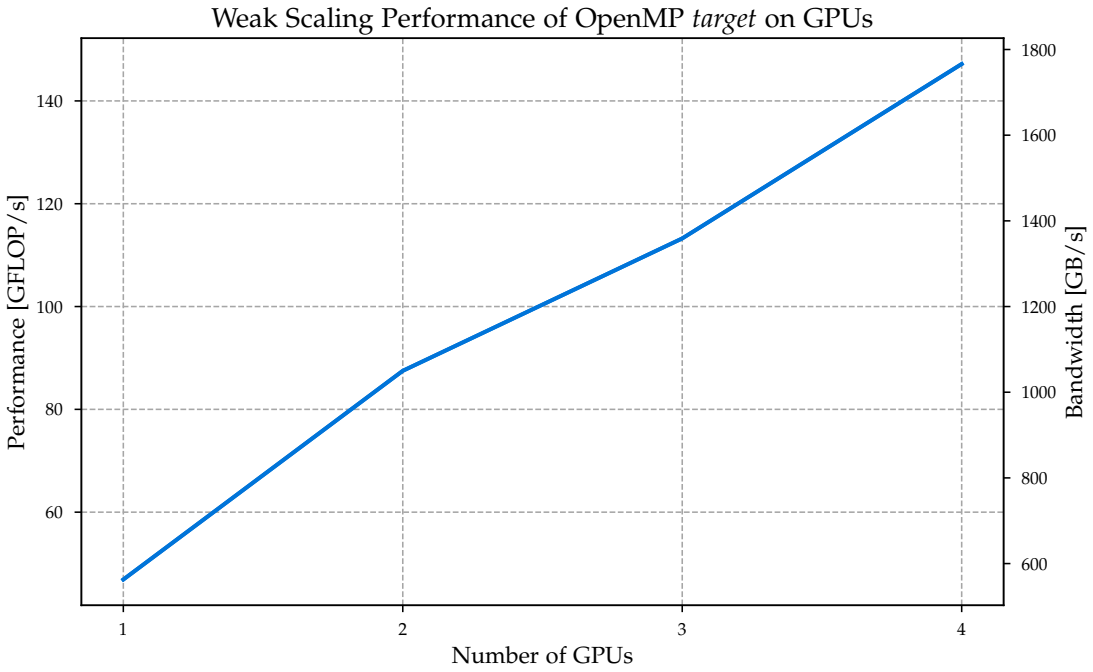


Figure 4.3: Near-linear weak scaling of the OpenMP *target* implementation of the triad benchmark, plotted as performance (GFLOP/s) and bandwidth (GB/s) in relation to the number of GPUs.

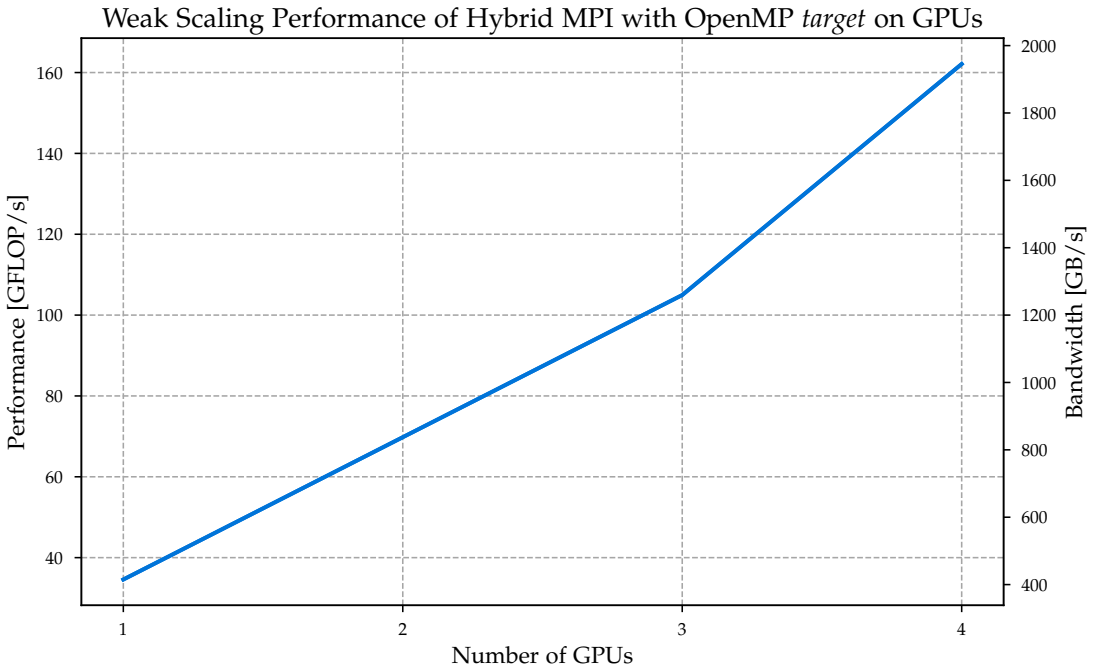


Figure 4.4: Near-linear weak scaling of the hybrid MPI with OpenMP *target* implementation of the triad benchmark, plotted as performance (GFLOP/s) and bandwidth (GB/s) in relation to the number of GPUs.

Strong Scaling

Strong scaling stands as a pivotal performance metric within parallel computing, offering a nuanced evaluation of how efficiently a parallel algorithm or application can manage a fixed-size problem as the number of processors or computational resources expands. It maintains a constant problem size while concurrently diminishing the time required to resolve it as additional resources are integrated. In essence, strong scaling quantifies the competence of a parallel system in the equitable distribution of workloads across multiple processors, with the ultimate goal of diminishing execution time as the number of processors multiplies, all the while preserving the problem size in stasis.

The gold standard is an ideal scenario where a linear or near-linear speedup is achieved, symbolizing the utmost efficiency in resource utilization. For instance, in our specific case, we selected an array size of 2^{27} and held this number in place, incrementally increasing the number of GPUs used from 1 to 4.

The visual representation of this phenomenon, as illustrated in Figures 4.5 and 4.6 for strong scaling employing the OpenMP and hybrid MPI approaches, serves as an exemplar of this captivating concept. Just as we observed in the case of weak scaling, these unveil an instance of near-ideal scaling, with a linear increase in bandwidth directly corresponding to the increasing number of GPUs while the data size remains constant.

4.2.2 Matrix-Matrix Multiplication

Performance Evaluation

To evaluate the performance of diverse computational methodologies, a comprehensive analysis was conducted, involving the manipulation of the matrix size (N) across a range spanning from 2^2 to 2^{14}. This analysis was performed across varying GPU configurations, encompassing 1, 2, 3, and 4 GPUs. In Figures 4.7 and 4.8, the results are presented, showcasing GFLOP/s and bandwidth (GB/s) as functions of matrix size for each GPU configuration. Both OpenMP and MPI methodologies yielded closely aligned results. As depicted in the graphs, it becomes evident from the bandwidth data that matrix-matrix multiplication is predominantly compute-bound. This is evidenced by the decline in bandwidth as the matrix size increases, concurrent with a rapid upsurge in performance for larger matrix sizes. However, as the matrix size grows to a substantial scale, the rate of performance improvement begins to decelerate, owing to the inherent simplicity of the matrix multiplication algorithm employed. This method, outlined in Algorithms 5 and 6 lack the exploitation of cache locality and fails to implement optimization techniques such as loop interchange or cache blocking [19].

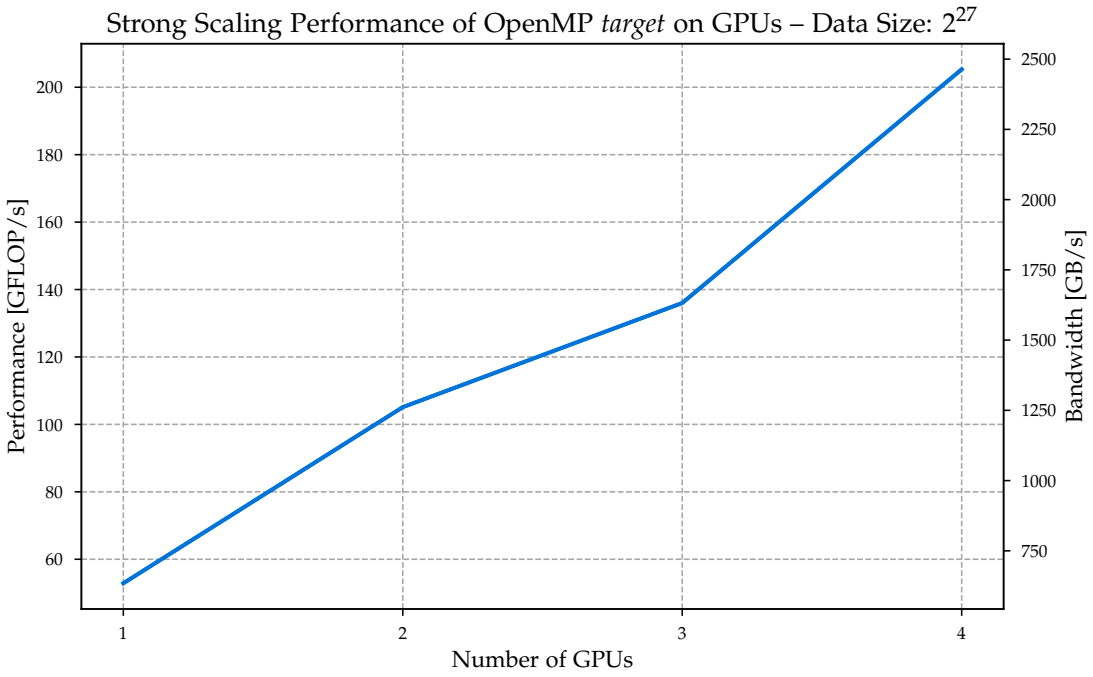


Figure 4.5: Near-linear strong scaling of the OpenMP *target* implementation of the triad benchmark, plotted as performance (GFLOP/s) and bandwidth (GB/s) in relation to the number of GPUs.

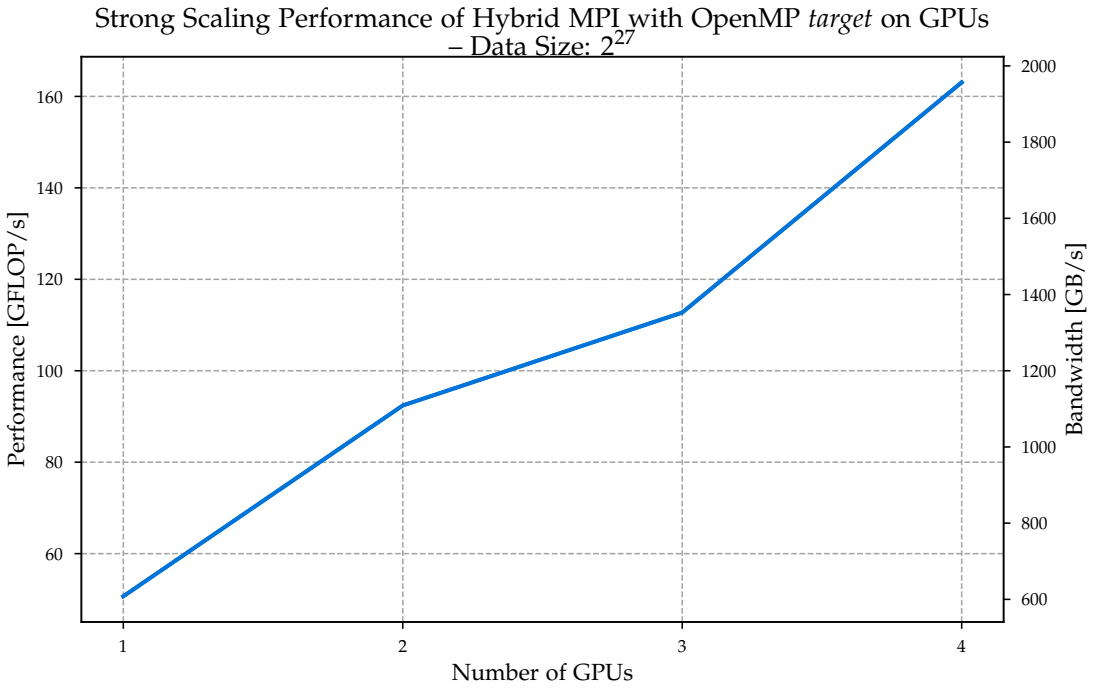


Figure 4.6: Near-linear strong scaling of the hybrid MPI with OpenMP *target* implementation of the triad benchmark, plotted as performance (GFLOP/s) and bandwidth (GB/s) in relation to the number of GPUs - Data Size: 2^{27}.

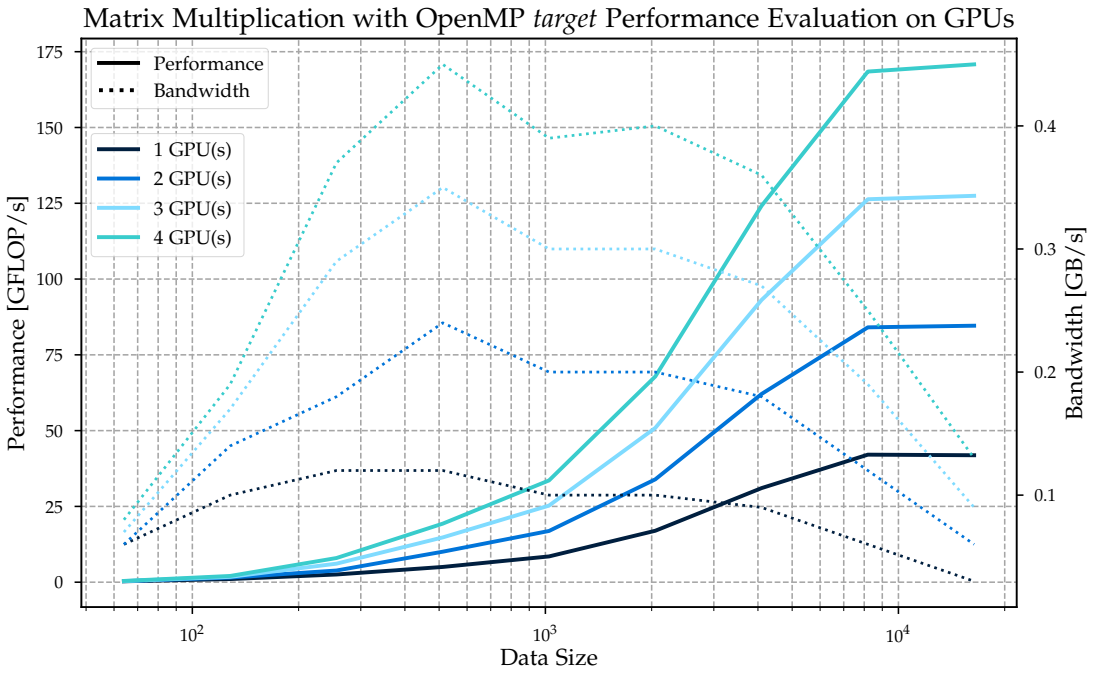


Figure 4.7: Variation in the performance (GFLOP/s) and bandwidth (GB/s) of the OpenMP *target* implementation of the matrix-matrix multiplication benchmark, as a function of data size. The evaluation includes tests on an increasing number of GPUs (1 to 4).

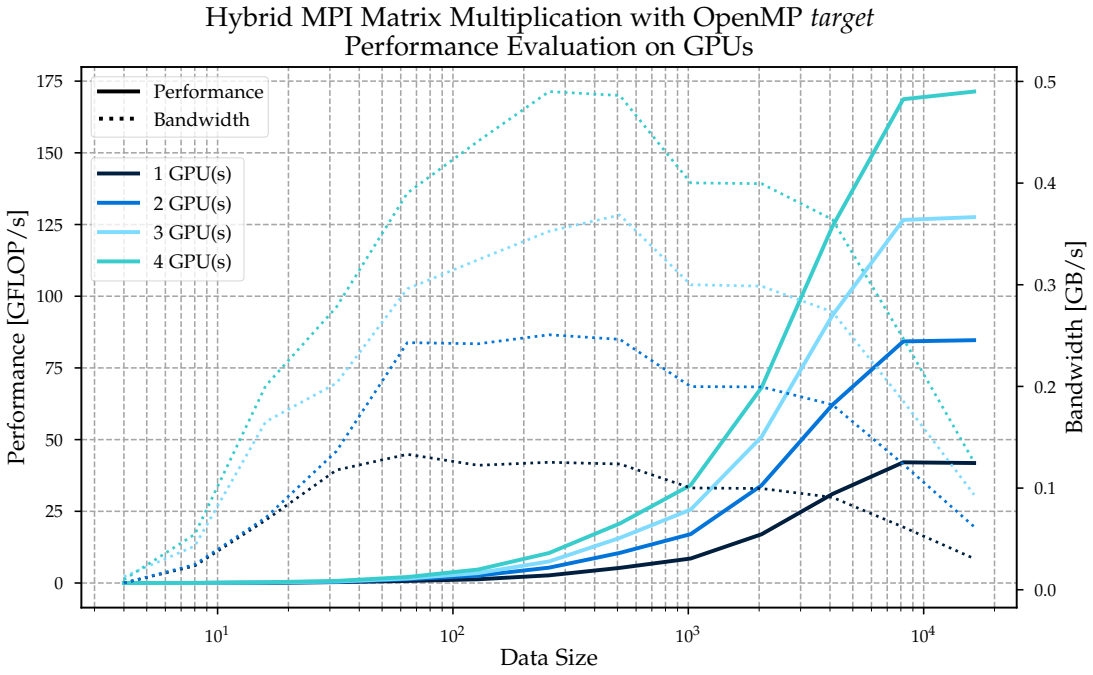


Figure 4.8: Variation in the performance (GFLOP/s) and bandwidth (GB/s) of the Hybrid MPI with OpenMP *target* implementation of the matrix-matrix multiplication benchmark, as a function of data size. The evaluation includes tests on an increasing number of GPUs (1 to 4).

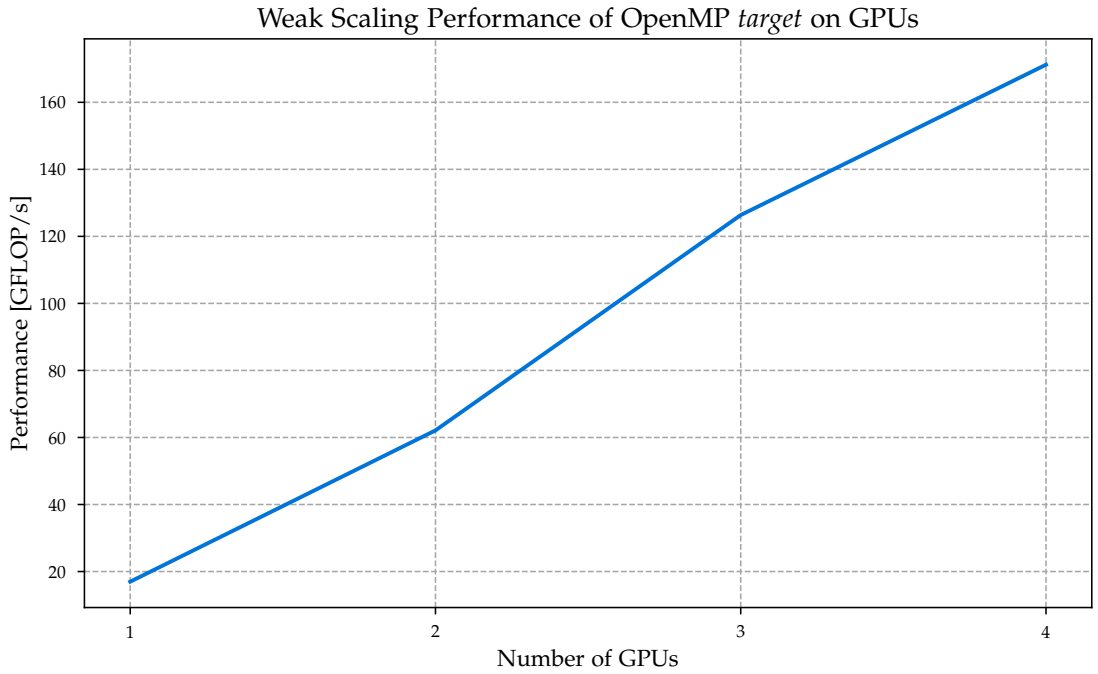


Figure 4.9: Near-linear weak scaling of the OpenMP *target* implementation of the matrix-matrix multiplication benchmark, plotted as performance (GFLOP/s) and bandwidth (GB/s) in relation to the number of GPUs.

Weak Scaling

Weak scaling was implemented by progressively increasing the matrix size from 2^{11} to 2^{14}, all the while adding one GPU for each doubling of the matrix size. The outcomes, as illustrated in Figures 4.9 and 4.10, portray a nearly ideal weak scaling, characterized by a linear increase in performance.

Strong Scaling

Strong scaling was implemented by selecting an array size of 2^{14} and incrementally increasing the number of GPUs used from 1 to 4. The resulting graphs are shown in Figures 4.11 and 4.12, where both implementations show ideal strong scaling.

4.3 Chosen Approach for ExaHyPE 2

Upon a thorough examination of various implementation strategies and a rigorous performance comparison, we have decisively opted for the hybrid approach, specifically combining MPI with OpenMP offloading. This choice stems from the inherent

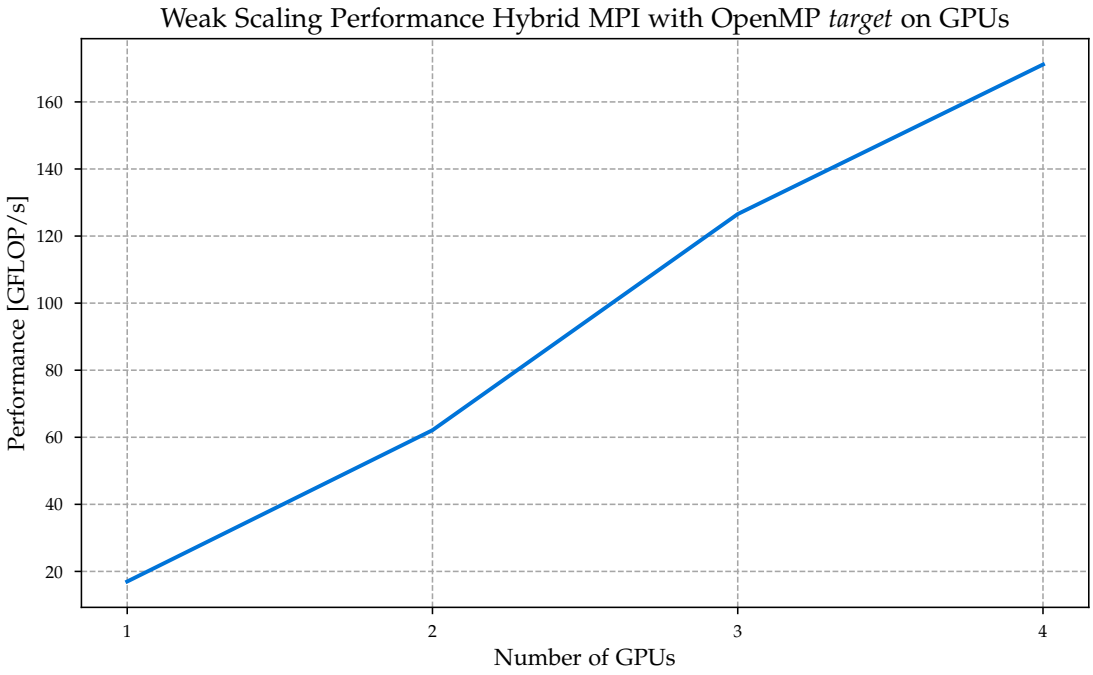


Figure 4.10: Near-linear weak scaling of the hybrid MPI with OpenMP *target* implementation of the matrix-matrix multiplication benchmark, plotted as performance (GFLOP/s) and bandwidth (GB/s) in relation to the number of GPUs.

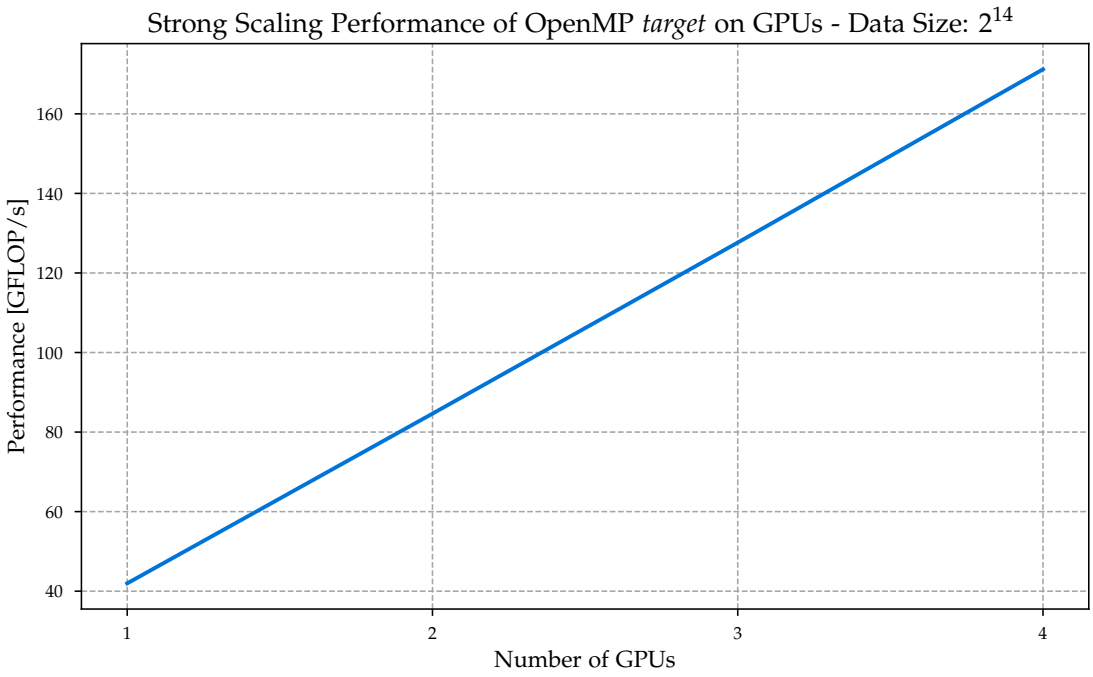


Figure 4.11: Linear strong scaling of the OpenMP *target* implementation of the matrix-matrix multiplication benchmark, plotted as performance (GFLOP/s) and bandwidth (GB/s) in relation to the number of GPUs - Data Size: 2^{14}.

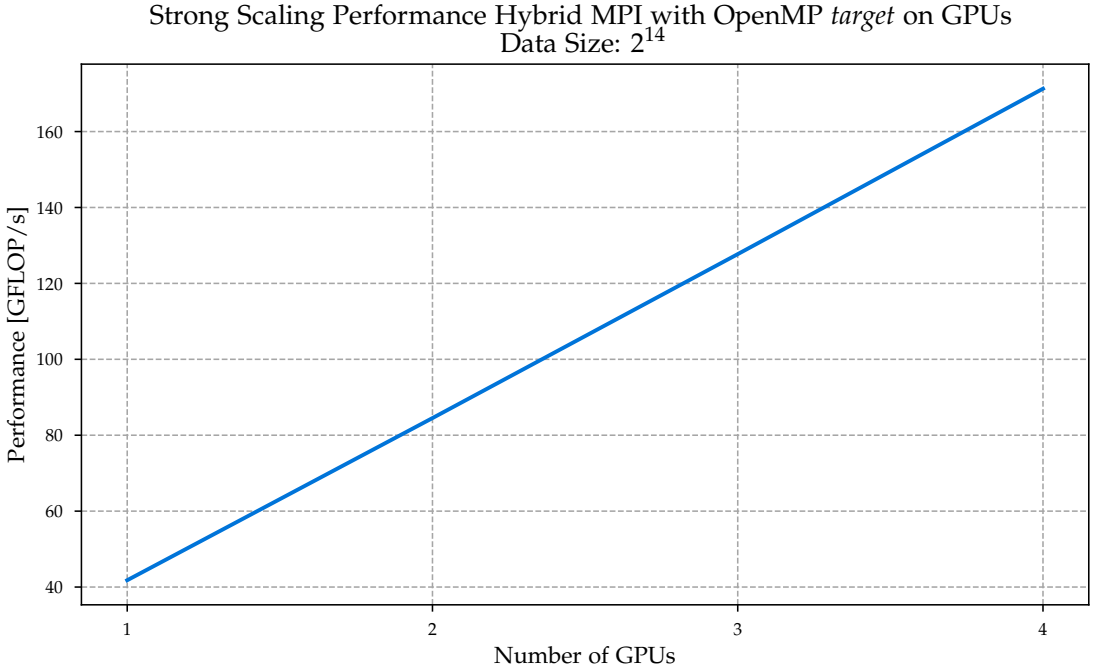


Figure 4.12: Linear strong scaling of the hybrid MPI with OpenMP *target* implementation of the matrix-matrix multiplication benchmark, plotted as performance (GFLOP/s) and bandwidth (GB/s) in relation to the number of GPUs - Data Size: 2^{14}.

advantages of the hybrid approach, notably its seamless integration of built-in MPI functions that are explicitly designed for data distribution and synchronization, all while maintaining an impressive level of performance efficiency. Given the complexity of the ExaHyPE 2 codebase we are addressing, the hybrid approach emerges as the pragmatic choice. It not only simplifies the implementation process but also assures the attainment of more precise results through a less labor-intensive methodology. This hybrid approach of MPI and OpenMP not only streamlines the development process but also promises to yield reliable and robust results in a highly efficient manner.

5 Implementation in ExaHyPE 2

5.1 Rusanov Kernel Benchmarks in ExaHyPE 2

The implementation of the Euler equations within ExaHyPE 2 is accompanied by a mini-app which evaluates the performance of the respective Rusanov kernels. This benchmark is designed to evaluate the computational speed of the core compute kernels. It exclusively measures throughput on a single node or GPU, providing valuable insights into the inherent efficiency of these essential computational components within the system. Notably, this benchmark does not consider extraneous factors such as multithreading coordination overhead, MPI data exchange, grid management, time-stepping behavior, or halo data exchange. While its relevance may be limited for extensive, large-scale simulations, wherein scalability, input/output overheads, and other variables take precedence, it provides an essential theoretical understanding of the capabilities of these core compute routines.

The experiments produce sets of results for various combinations of the number of threads responsible for launching compute tasks and the number of patches processed in each launch. In the case of a single thread, thread 0 within a multithreaded environment is assigned to execute various kernel implementations for a specific number of patches. Some of these kernels incorporate internal parallelization, while others do not. Subsequently, following the completion of these single-thread tests, two threads are introduced to simultaneously execute the same type of kernel. This approach is designed to stress-test the system.

For each realization, six distinct measurements are obtained:

1. The first measurement records the total time taken for a single kernel execution.
2. The second measurement quantifies the update per degree of freedom within a single kernel.
3. The third measurement focuses on the overall runtime of parallel launching, encompassing the setup and release of temporary data structures. It also accounts for the duration until all threads have completed their tasks. Consequently, this value is expected to be greater than the first measurement.

4. The fourth measurement provides the time required for a degree of freedom update when launching multiple kernels in parallel.

5. The fifth and sixth measurements furnish the raw data, enabling the assessment of the standard deviation of the measurements. It is noteworthy that each value is typically sampled multiple times to reduce potential noise.

For our evaluation, we will only utilize the first measurement, which is the total time taken for a single kernel execution.

The expected outcomes of the experiments depend on the system's ability to fully utilize available processing cores. When a single kernel invocation for multiple patches by one thread already optimally utilizes the available cores, launching multiple kernels should yield either identical throughput or potentially a degradation in performance due to inherent overhead. Conversely, if a single kernel launch for multiple patches cannot effectively engage all available threads, launching multiple kernels in parallel is expected to enhance overall throughput.

5.2 Hybrid MPI OpenMP

In this section, we describe how we transformed the Rusanov benchmark to accommodate the multiple GPU. This was achieved by enabling the utilization of MPI for data distribution and synchronization, and utilizing the rank as the device ID (*GPU_ID*) for offloading of kernels to the GPU through OpenMP's *target* clause. The adaptation process comprises three main stages: domain decomposition, reduction, and barrier synchronization. Once the MPI implementation is integrated, the GPU kernels are invoked with their respective (*GPU_ID*). It is worth noting that this MPI implementation also extends parallelization to the host benchmark. The pseudo code for the MPI implementation is shown in Algorithm 7.

5.2.1 Domain Decomposition

Each instance of the kernel procedure is invoked with $\|P_{GPU}\|$, representing the quantity of patches to be utilized. In the single GPU implementation, this kernel is subsequently offloaded to a specific GPU in accordance with (*GPU_ID*) through OpenMP offloading. In the context of a multi-GPU implementation, the initial step involves partitioning the number of total patches (*PATCHES*) based on the desired number of GPUs, necessitating adjustments to loop boundaries. The quantity of GPUs employed equals the number of MPI ranks, meaning that if four GPUs are intended for use, the benchmark must execute with four MPI ranks, with each rank's id (*rank*) corresponding to a device ID (*GPU_ID*).

To facilitate this, we calculate the number of patches allocated to each MPI rank (*LOCAL_PATCHES*). Its determination takes into account the rank's id (*rank*), the total

number of ranks (*NUM_PROCESSES*), and whether the total number of patches is evenly divisible by the total number of ranks. The calculation for *LOCAL_PATCHES* for the last MPI rank, accounting for any remainder (*REMAINDER*) resulting from an uneven division of patches among ranks, is presented in Equation 5.1. For all other ranks, except the final one, the calculation for *LOCAL_PATCHES* is simply the number of patches divided by the number of MPI ranks, as depicted in Equation 5.2.

$$LOCAL\_PATCHES = \frac{PATCHES}{NUM\_PROCESSES} + PATCHES \,\%\, NUM\_PROCESSES \tag{5.1}$$

$$LOCAL\_PATCHES = \frac{PATCHES}{NUM\_PROCESSES} \tag{5.2}$$

5.2.2 Reduction

The kernel benchmark is invoked once per MPI rank, resulting in the offloading of the kernel on a per-rank basis. Subsequently, each kernel execution yields a corresponding output, from which we compute the error and maximum deviation relative to a reference case. As each rank maintains its local error and local maximum difference, we incorporate an MPI reduction operation to sum these values, thereby computing the total error and maximum deviation specific to the given number of patches.

5.2.3 Barrier Synchronization

Within the main function, the kernel benchmark is invoked with varying numbers of patches, starting with a minimum value equal to the number of MPI ranks and progressively increasing up to a maximum number of patches by doubling the patch count in each iteration. The choice of the minimum value is strategically made to maintain the correctness of the domain decomposition.

Given that one MPI rank might complete its computations faster than others, this rank can return to the main function and start a new call to kernel. This situation can introduce a race condition regarding the values of local error and local maximum difference. To solve this issue, a barrier is introduced following the function call to ensure that all ranks collectively wait until each of them completes the execution of the kernel.

5.3 Validation

To verify the execution of this benchmark on the designated number of GPUs, we continuously monitored the number of GPUs in use, the fluctuating GPU utilization for each individual GPU, and the GPU memory consumption in Mebibytes (MiB). The testing encompassed configurations involving 1 to 4 GPUs, representing the maximum

Algorithm 7 ExaHyPE 2 Rusanov kernel benchmark multi-GPU implementation

```
Require: PATCHES, LAUNCHING_THREADS
Create a GridTraversalEvent event
Create a CellMarker marker with event
procedure assessKernel(kernel, markerName, LAUNCHING_THREADS, GPU_ID,
PATCHES)
    NUM_PROCESSES = number of MPI ranks
    patchesPerProcess = PATCHES / NUM_PROCESSES
    REMAINDER = PATCHES % NUM_PROCESSES
    startPatch = rank · patchesPerProcess
    endPatch = startPatch + patchesPerProcess + (rank == NUM_PROCESSES - 1 ?
REMAINDER : 0)
    LOCAL_PATCHES = endPatch - startPatch
    for reduceMaxTimeStep in [0, 1] do
        for j in [0, NumberOfSamples - 1] do
            Create a CellData patchData with size LOCAL_PATCHES
            for i in [0, LOCAL_PATCHES - 1] do
                Initialize patchData attributes
                Call initInputData on patchData.QIn[i]
            end for
            Call one kernel for each MPI rank
            if Accuracy > 0.0 then
                Store outcome
                Compute local errors and local max difference on each rank
                Reduce localErrors and localMaxDifference with MPI
                if errors > 0 then
                    Log error and abort if necessary
                end if
            end if
            for i in [0, LOCAL_PATCHES - 1] do
                Free memory for patchData.QIn[i] and patchData.QOut[i]
            end for
        end for
    end for
end procedure
if GPU offloading with OpenMP is enabled then
    Assess kernels with OpenMP using GPU_ID as current MPI rank
end if
```

available on a single cluster node, and varying numbers of patches. The examination yielded the following insights:

- The observed count of GPUs in use precisely matched the intended number of GPUs.
- A uniform distribution of volatile GPU utilization was evident across all GPUs, indicating effective load balancing.
- The memory allocation across GPUs demonstrated uniformity, affirming the expected outcome of the domain decomposition.
- A synchronized barrier operation following the function call exhibited the desired behavior. At the start of a new patch configuration, GPU utilization across all GPUs uniformly reduced to zero, and memory was offloaded simultaneously. This was verified by monitoring the GPU memory usage.
- We also conducted a comparative analysis of the multi-GPU benchmark results against single-GPU, parallel, and sequential host (CPU) kernel calls. The outcomes were found to be consistent, thereby validating the computational correctness.

These observations validate the functionality of the hybrid MPI with OpenMP offloading implementation.

5.4 Benchmarking

5.4.1 Performance Evaluation

We conducted a comprehensive performance analysis using 1, 2, 3, and 4 GPUs to evaluate a specific 3D kernel benchmark with patch sizes of 8. The number of patches (*PATCHES*) spanned from 2^2 to 2^{15}, with each step doubling the patch count. The graphical representation of our findings is illustrated in 5.1 Our analysis unveiled an interesting pattern: up to 2^{14} patches the temporal differences among the various GPU configurations were negligible. In some instances, a single GPU even exhibited superior performance over multiple GPUs. This intriguing phenomenon can be attributed to communication overhead.

Communication overhead represents the delays and inefficiencies introduced when data needs to be transferred between different processing units. In scenarios with relatively small problem sizes, communication overhead plays a significant role. The additional time spent on data transfers and synchronization processes can offset the potential benefits of parallelism.

However, as the problem size increased, we observed a remarkable improvement in performance when using multiple GPUs. This underscores the pivotal role of the

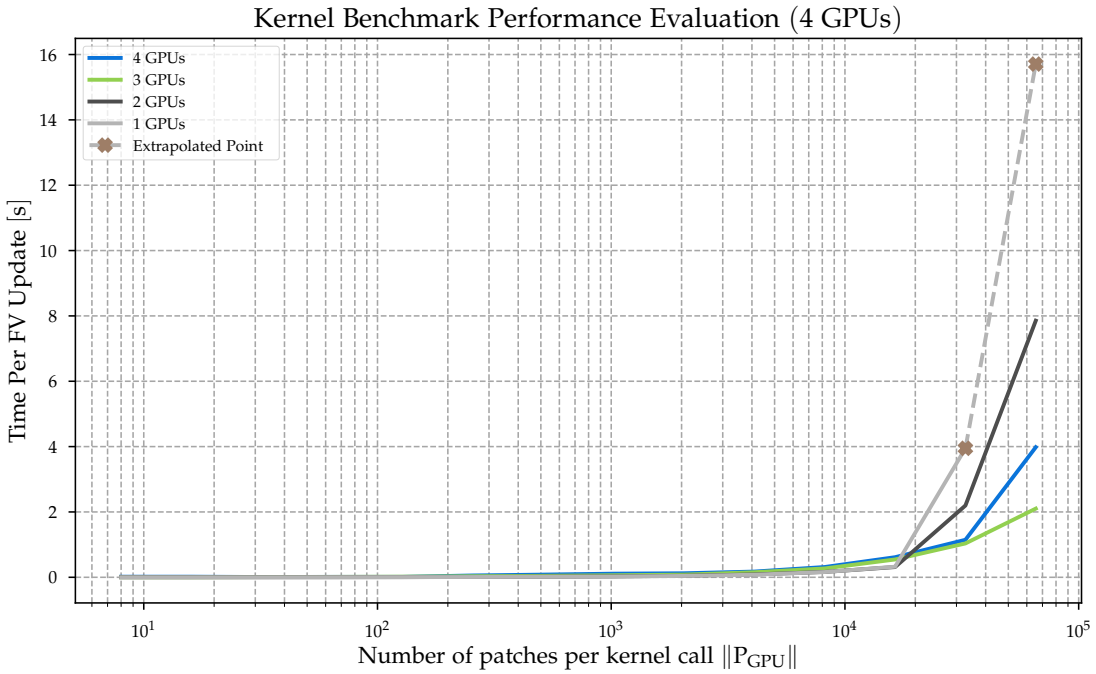


Figure 5.1: Performance of the ExaHyPE 2 Rusanov kernel benchmark multi-GPU implementation, measured in time per Finite Volume (FV) update (s), as a function of the number of patches per kernel call. Testing conducted on an increasing number of GPUs (1 to 4).

multi-GPU approach, particularly for applications like ExaHyPE 2, tailored to manage millions of patches. In such cases, the utilization of multiple GPUs offers the potential for substantial overall performance gains. It is important to note that the upper limit of the number of patches was set to 2^{15} due to the limited memory available on one GPU.

5.4.2 Weak Scaling

Weak scaling was performed by varying the number of patches (*PATCHES*) for the same problem as above from 2^{13} to 2^{16} while adding one GPU as the number of patches doubled. The resulting graph is show in Figure 5.2.

5.4.3 Strong Scaling

The third aspect of performance analysis undertaken is strong scaling. In this evaluation, the same problem as above was used, with the number of patches (*PATCHES*) set to 2^{16}. The selection of the number of patches took into consideration the available GPU memory capacity, limited to 10 GB of RAM, as well as the desire to minimize communication overhead among multiple GPUs.

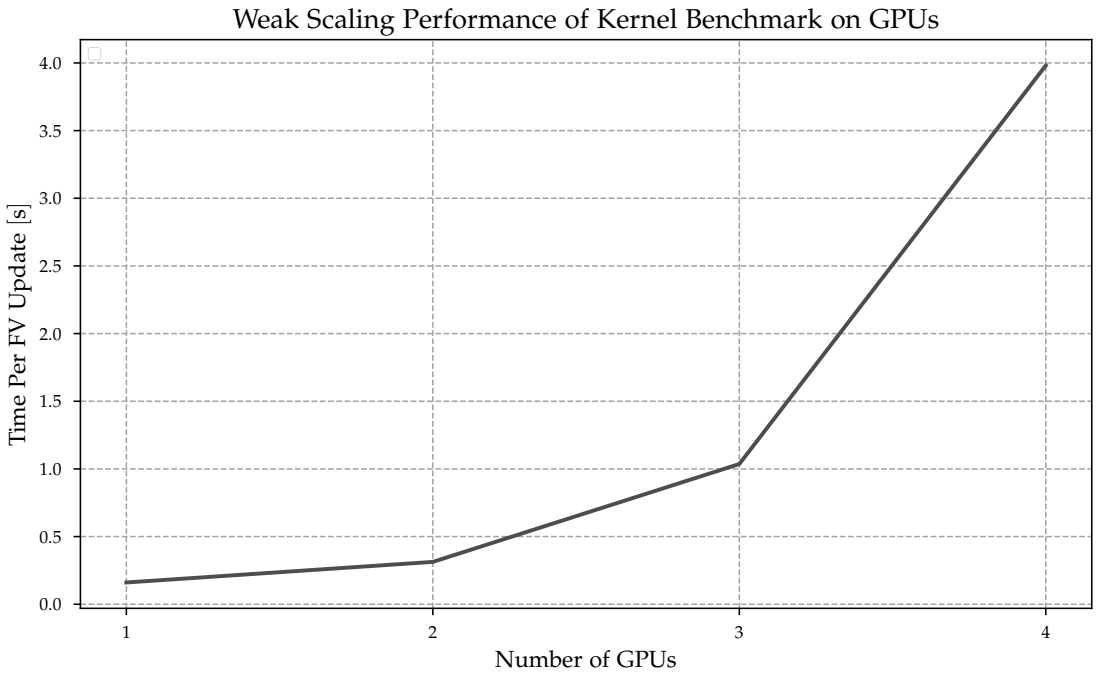


Figure 5.2: Weak scaling performance of the ExaHyPE 2 Rusanov kernel benchmark multi-GPU implementation, represented as time per Finite Volume (FV) update (s), with respect to the number of patches per kernel call.

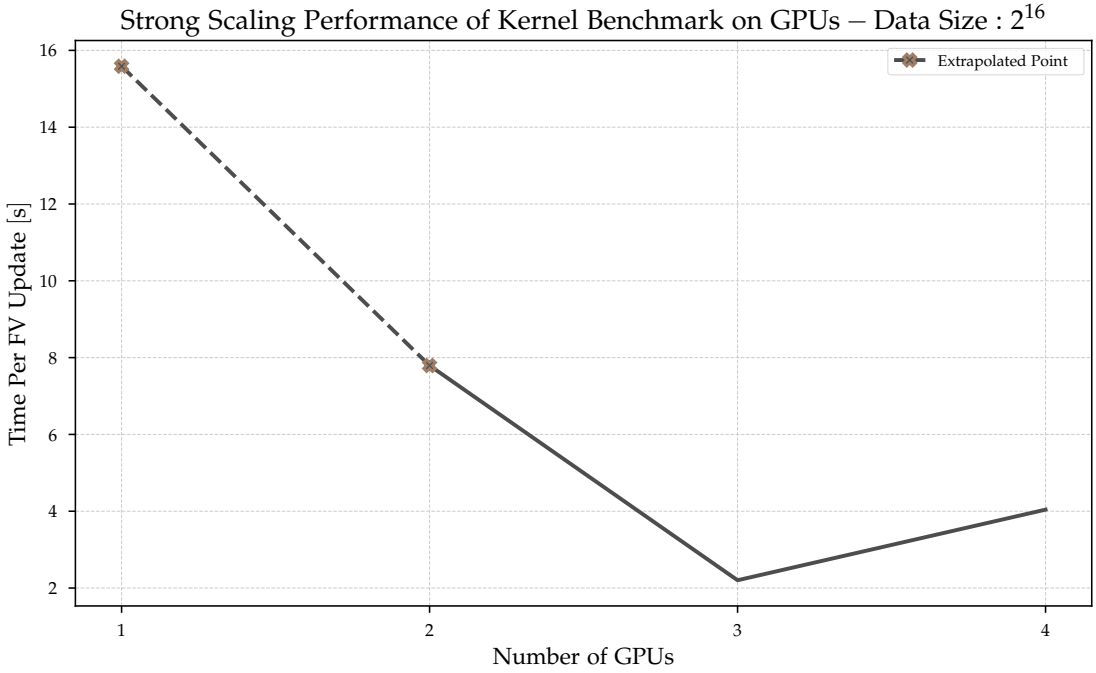


Figure 5.3: Strong scaling performance of the ExaHyPE 2 Rusanov kernel benchmark multi-GPU implementation, illustrated as time per Finite Volume (FV) update (s), with respect to the number of patches per kernel call. Data size: 2^{16}.

As depicted in Figure 5.3, the results of the strong scaling analysis reveal several key insights. It is evident from the graph that, for this specific number of patches, the optimal configuration involves employing three GPUs. This choice is attributed to the problem size being sufficiently substantial to fully exploit the capabilities of all three GPUs, rendering communication overhead negligible. When employing four GPUs, the kernel execution time is marginally longer compared to three GPUs, with an additional 2 seconds attributable to the problem size being insufficient to mitigate the impact of communication overhead.

Conversely, utilizing only two GPUs results in a fourfold decrease in performance compared to the three-GPU setup, taking 8 seconds. It is important to note that this problem size is not amenable to execution on a single GPU. To provide a basis for comparison, we have interpolated the expected execution time on a single GPU, which is represented by the dotted line.

In light of the foregoing findings, it is evident that as the number of patches increases, the utilization of multiple GPUs will undoubtedly lead to substantial performance improvements.

6 Case Study

In this section, we simulate the 3D Euler equations using a patch-based enclave solver in ExaHyPE 2, incorporating the multi-GPU approach using hybrid MPI with OpenMP *target* offloading. The implementation is subjected to validation and benchmarking.

6.1 The Euler Equations

The Euler equations are fundamental in fluid dynamics, describing the motion of an ideal fluid in the absence of energy losses due to friction or heat conduction. In their first order conservative form for two dimensions, assuming no external forces, these equations are given by

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho u_1\\ \rho u_2\\ E_t\end{pmatrix} + \nabla\begin{pmatrix}\rho u_1 & \rho u_2\\ \rho u_1^2 + p & \rho u_1 u_2\\ \rho u_2 u_1 & \rho u_2^2 + p\\ (E_t + p)u_1 & (E_t + p)u_2\end{pmatrix} = \vec{0}. \tag{6.1}$$

These equations govern the conservation of mass, momentum, and energy in a two-dimensional fluid flow. They can also be expressed more concisely as

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ j\\ E\end{pmatrix} + \nabla\cdot\begin{pmatrix}j\\ \frac{1}{\rho}j\otimes j + pI\\ \frac{1}{\rho}j\,(E+p)\end{pmatrix} = \vec{0}. \tag{6.2}$$

Here, ρ represents density, u_1 and u_2 are velocity components, E_t is total energy, and p is pressure. To implement these equations effectively, specifying appropriate initial conditions, boundary conditions, eigenvalues, and a flux function is essential. The eigenvalues for the 2D Euler equations are key to understanding wave propagation in the fluid, depending on fluid velocity and the wave speed, influenced by the adiabatic index γ and the energy. In practice, initial conditions should define the fluid's starting state, boundary conditions are imposed at domain boundaries, and eigenvalues play a crucial role in numerical solutions. The eigenvalues of the two-dimensional Euler equations are given by

$$\begin{pmatrix}\lambda_1\\ \lambda_2\\ \lambda_3\end{pmatrix} = \begin{pmatrix}u - c\\ u\\ u + c\end{pmatrix}. \tag{6.3}$$

where c is the wave propagation speed. For dry air, which can be approximated as an ideal gas, c depends on the pressure as

$$\begin{pmatrix} \gamma \\ p \\ c \end{pmatrix} = \begin{pmatrix} 1.4 \\ (\gamma - 1)(E_t - \frac{1}{2\rho}(\rho u)^2) \\ \sqrt{\frac{\gamma p}{\rho}} \end{pmatrix}. \tag{6.4}$$

For the three-dimensional Euler equations, an extension of the 2D equations to three dimensions, the equations follow a similar structure, accounting for fluid motion in three spatial dimensions. The eigenvalues and pressure-wave speed relationship are analogous to the 2D case.

The three-dimensional Euler equations are very similar to the two-dimensional equations and can be represented as

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho u_3 \\ E_t \end{pmatrix} + \nabla \begin{pmatrix} \rho u_1 & \rho u_2 & \rho u_3 \\ \rho u_1^2 + p & \rho u_1 u_2 & \rho u_1 u_3 \\ \rho u_2 u_1 & \rho u_2^2 + p & \rho u_2 u_3 \\ \rho u_3 u_1 & \rho u_3 u_2 + p & \rho u_3^2 + p \\ (E_t + p)u_1 & (E_t + p)u_2 & (E_t + p)u_3 \end{pmatrix} = \vec{0}. \tag{6.5}$$

The PDE is:

$$\frac{\partial \rho(x)}{\partial t} + \nabla \cdot \begin{bmatrix} j_0 \\ j_1 \end{bmatrix} = S_0, \tag{6.6}$$

$$\frac{\partial j_0(x)}{\partial t} + \nabla \cdot \begin{bmatrix} 0.4E + \frac{j_0^2}{\rho} - 0.4\left(\frac{0.5j_0^2 + 0.5j_1^2}{\rho}\right) \\ \frac{j_0 j_1}{\rho} \end{bmatrix} = S_1, \tag{6.7}$$

$$\frac{\partial j_1(x)}{\partial t} + \nabla \cdot \begin{bmatrix} \frac{j_0 j_1}{\rho} \\ 0.4E + \frac{j_1^2}{\rho} - 0.4\left(\frac{0.5j_0^2 + 0.5j_1^2}{\rho}\right) \end{bmatrix} = S_2, \tag{6.8}$$

$$\frac{\partial E(x)}{\partial t} + \nabla \cdot \begin{bmatrix} \frac{j_0(1.4E - 0.4\left(\frac{0.5j_0^2 + 0.5j_1^2}{\rho}\right))}{\rho} \\ \frac{j_1(1.4E - 0.4\left(\frac{0.5j_0^2 + 0.5j_1^2}{\rho}\right))}{\rho} \end{bmatrix} = S_3. \tag{6.9}$$

The eigenvalues are:

$$\lambda_{\max,0}(x) = \begin{bmatrix} \frac{j_0}{\rho} - \gamma\sqrt{\frac{E - \left(\frac{0.5j_0^2 + 0.5j_1^2}{\rho}\right)}{\rho}} \\ \frac{j_1}{\rho} - \gamma\sqrt{\frac{E - \left(\frac{0.5j_0^2 + 0.5j_1^2}{\rho}\right)}{\rho}} \end{bmatrix}, \tag{6.10}$$

$$\lambda_{\max,1}(x) = \begin{bmatrix} \frac{j_0}{\rho} \\ \frac{j_1}{\rho} \end{bmatrix}, \tag{6.11}$$

$$\lambda_{\max,2}(x) = \begin{bmatrix} \frac{j_0}{\rho} \\ \frac{j_1}{\rho} \end{bmatrix}, \tag{6.12}$$

$$\lambda_{\max,3}(x) = \begin{bmatrix} \frac{j_0}{\rho} + \gamma\sqrt{\frac{E-\left(\frac{0.5j_0^2+0.5j_1^2}{\rho}\right)}{\rho}} \\ \frac{j_1}{\rho} + \gamma\sqrt{\frac{E-\left(\frac{0.5j_0^2+0.5j_1^2}{\rho}\right)}{\rho}} \end{bmatrix}, \tag{6.13}$$

where γ is 0.748331477354788.

6.2 Enclave Tasking

As described in [8], enclave tasking introduces a novel approach distinct from traditional domain decomposition strategies. Each time-step involves a dual traversal of the mesh, classifying cells into two categories: skeleton cells, situated near partition boundaries or AMR resolution transitions, and enclave cells, encompassing the remainder. The primary traversal, upon encountering a skeleton cell, initiates updates, determining the new local solution, permissible time-step size, and the required values for adjacent cells in the subsequent time-step. Handling data from skeleton cells involves interpolation, restriction, MPI transmission, or local copying to another logical subpartition.

Conversely, when the primary traversal engages with an enclave cell, the local update is mapped onto a task within the thread-local traversal. Post the primary grid sweep, there is an exchange of partition boundary data. The secondary grid traversal patiently awaits task completion, integrating the task outcome into the solution representation, and reducing the permissible time-step size per rank. The final serial phase orchestrates global time-step size reduction and concludes all MPI data exchanges.

The realization of enclave tasking unfolds as a sequence of two taskloop constructs per time-step [8]. In contrast to conventional implementations, the initial taskloop serves as a task producer without synchronization with spawned enclave tasks. Despite the removal of implicit barriers, it is still termed as a task group. The local domain decomposition, though consistent, generates a multitude of small enclave tasks per primary sweep.

In the secondary traversals, busy waits are embedded to incorporate simulation outcomes into the computational mesh. This integration is pivotal for updating the patch halo and synchronizing patches with their neighbors [8]. In the baseline OpenMP implementation, busy waiting unfolds as the code repetitively polls the hashmap. While the required task outcome is not yet in the hashmap, the polling code releases

the semaphore, issues a taskyield, and then polls again. This approach reflects a straightforward implementation of the consumer in a producer-consumer pattern.

6.3 Implementation

To facilitate the execution of the Euler equations on multiple GPUs, a minimal adaptation of the task orchestration is required to establish the mapping between MPI ranks and GPU device IDs. This adaptation entails modifying the tasking strategy instantiation by including parameters for the MPI communicator's size and rank. Subsequently, the rank is utilized as the target device identifier (GPU_ID) during the creation of the tasking pattern, specifically employing a *fuse all* tasking orchestration. This tasking pattern enhances task execution by opportunistically fusing tasks, thereby optimizing processing efficiency. The degree of task fusion depends on specific conditions; tasks are either fused immediately upon creation or temporarily stored in a local queue for subsequent fusion when processing threads become available. This deferred fusion mechanism does not impede task production threads, ultimately facilitating efficient task processing. Additionally, the strategy designates the target execution device, which, in this context, corresponds to the MPI rank.

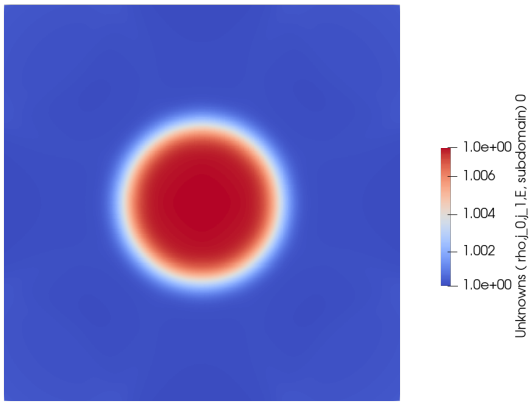
6.4 Validation

We conducted a two-dimensional Euler point explosion simulation utilizing two MPI ranks. We visualized the outcomes in ParaView. Figure 6.1a exhibits the density variation, and Figure 6.1b presents the velocity variation. The visualization exhibits physical behavior. Figure 6.1c demonstrates the energy. We can conclude from those results that for the two-dimensional case, utilizing multiple GPUs yields expected results.

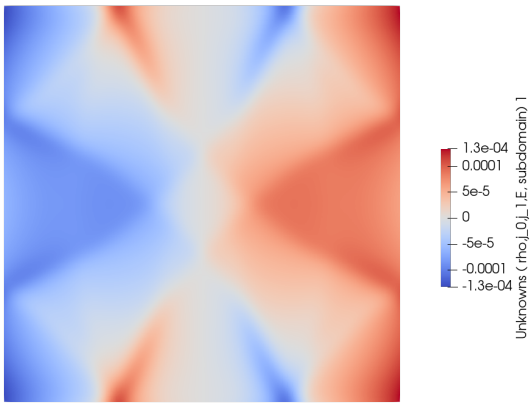
Subsequently, we conducted a three-dimensional Euler point explosion simulation utilizing two MPI ranks, using patch size 8, 19683 total patches, and the *fuse all* tasking orchestration. We visualized the outcomes in ParaView. Figure 6.2a exhibits the density variation, and Figure 6.2b presents the velocity variation. The visualization also reveals characteristic physical behavior. Figure 6.2c demonstrates the energy. Therefore, our multi-GPU implementation's results are valid.

6.5 Benchmarking

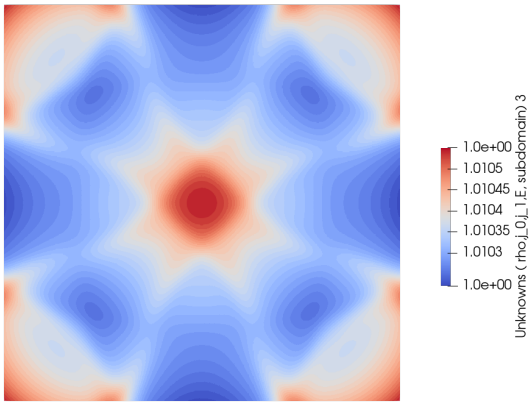
The three-dimensional Euler point explosion simulation was rigorously benchmarked, employing a patch size of 8, 19683 total patches, and the *fuse all* tasking orchestration. The experiment encompassed a variation of MPI ranks, specifically 1, 2, 3, and 4. Consistent with these parameters, we conducted multiple runs by adjusting the number



(a) Density

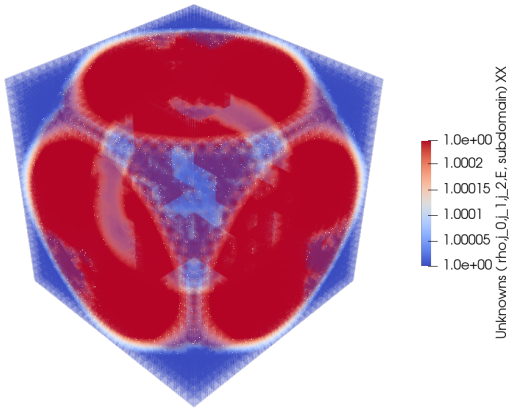


(b) J_0 Velocity

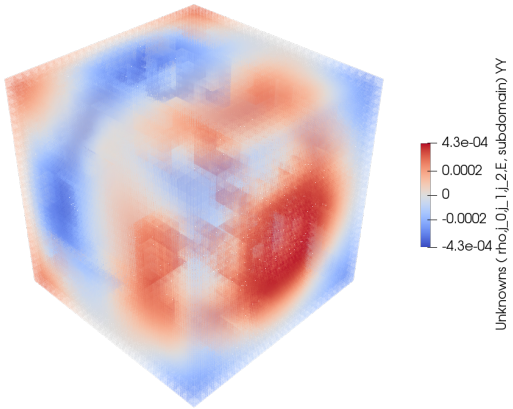


(c) Energy

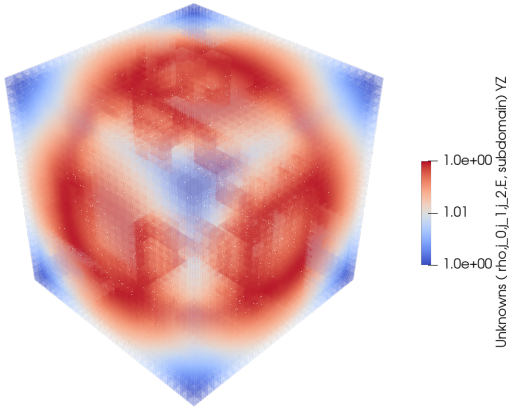
Figure 6.1: Two-dimensional Euler point explosion density, velocity and energy respectively using two MPI ranks (two GPUs).



(a) Density



(b) YY Velocity Component



(c) Energy

Figure 6.2: Three-dimensional Euler point explosion density, velocity and energy respectively using two MPI ranks (two GPUs).

of tasks to fuse—specifically, 1, 1024, 1024*2, and 1024*4. The comprehensive results are detailed in Figure 6.3.

Key observations focused on GPU memory usage, percentage of GPU utilization, and the time taken to complete the simulation. Figure 6.3 illustrates the superior performance achieved by utilizing either 1 GPU or 2 GPUs, attributed to the relatively small domain size causing communication overhead.

In analyzing simulation time in relation to the number of tasks to fuse, a noteworthy finding was the optimal choice of 1024 tasks to fuse, resulting in a minimum time of 11 seconds with 1 GPU. Importantly, a significant difference in simulation times emerged when using a small number of tasks to fuse (1 in our case) compared to larger numbers. Increasing the number of tasks led to a remarkable reduction in simulation time, by a factor of 6 for 1 GPU and 3 for 2 GPUs. This behavior is explained by the frequent movement of tasks between the CPU and GPU, leading to substantial overhead when using a small number of tasks to fuse. Moreover, utilizing a very large number of tasks on a small domain may result in underutilization of GPUs, as there may not be enough tasks to offload. The superiority of 1 GPU or 2 GPUs depends on the number of tasks to fuse, with 2 GPUs outperforming 1 GPU by a factor of two for 1 task to fuse. However, this relationship fluctuates with varying numbers of tasks to fuse.

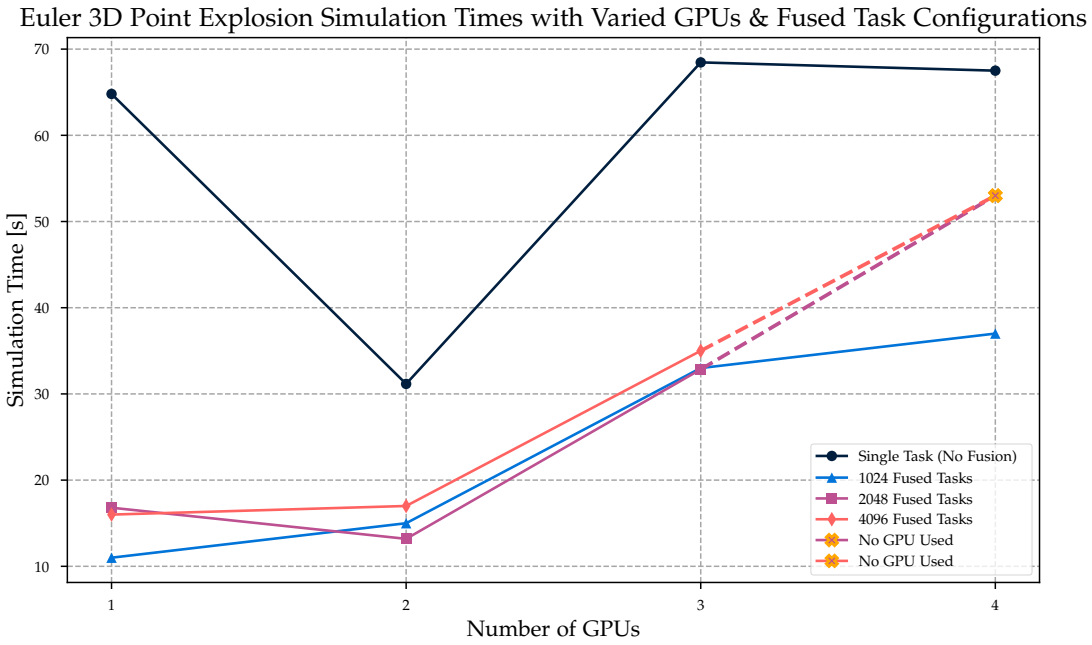


Figure 6.3: Euler 3D point explosion simulation time as a function of the number of GPUs and the number of tasks to fuse.

Considering the percentage of GPU utilization, we observed that the lower the

number of tasks to fuse, the more the GPUs are utilized. Regarding GPU memory usage, an inverse relationship was noted, with higher numbers of tasks leading to increased memory consumption. Figure 6.3 visually represents the unpredictable behavior resulting from manipulating these parameters.

It is paramount to note that these results are contingent on a relatively small domain utilizing only one node. The outcomes may significantly differ when using a smaller cell size (finer domain) with more than one node. Additionally, the bandwidth limitation of the MPI backend communication plays a pivotal role in performance [8] and warrants in-depth investigation, as discussed further in Chapters 7 and 8. It is not correct to assume that to get the best performance, the number of tasks to fuse must be increased as much as possible, since the bandwidth limitations can severely affect the performance, especially when using multiple ranks. This is discussed further in Chapters 7 and 8.

Given the complexities of the codebase and enclave tasking, determining the optimal choice of MPI ranks is inherently non-deterministic. This underscores the imperative need for an autotuner, a subject to be explored in greater detail in Chapters 7 and 8.

7 Conclusion

In conclusion, our exploration into the realm of multi-GPU programming within the ExaHyPE 2 framework has yielded valuable insights and promising results. Through a meticulous investigation of two distinct approaches, namely OpenMP *target* and hybrid MPI with OpenMP *target* offloading, we have discovered that the latter stands out as the optimal way to harness the parallel processing power of multiple GPUs effectively.

Our benchmarking efforts, particularly focusing on the Rusanov kernel benchmark suite in ExaHyPE 2, have demonstrated the potential of our multi-GPU implementation, showcasing promising results that indicate a substantial reduction in simulation times. It is noteworthy that the scalability of this approach is contingent upon the problem size, emphasizing the importance of ensuring a sufficiently large computational domain to minimize communication overhead when scaling the number of GPUs.

However, despite the promising strides in our multi-GPU implementation, we encountered limitations when attempting to scale up the problem size in the context of the Euler equations using a patch-based enclave solver in ExaHyPE 2 with multiple MPI ranks. The constraint of utilizing only one node led to a notable challenge: a flurry of MPI activity [8]. This term encapsulates the intensified communication between processes, resulting in a saturation of network bandwidth. In our case study, we observe this when using a cell size smaller than 0.0025, which leads to MPI communication hanging when using multiple MPI ranks on one node.

In the intricate landscape of parallel computing, particularly when dealing with complex or large-scale computations, the necessity for frequent data exchange between processes is inherent. In our scenario, the elevated volume and frequency of MPI communication proved to be a bottleneck, hindering the seamless scalability of our implementation. This limitation underscores the importance of not only optimizing algorithms and parallelization strategies but also considering the interplay of network communication in the overall performance.

Additionally, the non-deterministic challenges in determining the optimal number of MPI ranks for the Euler equations using enclave tasking and task orchestration for fusion highlight the need for an autotuner. An autotuner would dynamically adjust the configuration (number of enclave tasks to be fused and number of total patches/tasks) to optimize performance, addressing the unpredictability introduced by the code base and tasking complexity.

As we reflect on the outcomes of our research, this challenge serves as a reminder of the intricacies involved in achieving optimal scalability in multi-GPU implementations.

Addressing such limitations and delving into the intricacies of MPI activity will be crucial for further advancements, highlighting the ongoing need for innovative solutions and optimizations to fully unleash the potential of multi-GPU programming in the realm of how ExaHyPE 2 wants to fully exploit heterogeneous hardware architectures.

Our goal, articulated in the introduction, was to provide researchers and practitioners with a comprehensive understanding of the potential of multi-GPU acceleration in the ExaHyPE 2 framework. By achieving this, we hope to catalyze advancements in computational simulations, enabling scientists and engineers to gain deeper insights into complex phenomena and solve real-world problems more efficiently.

8 Discussion and Future Work

Looking forward, our current research paves the way for crucial advancements within the ExaHyPE 2 framework, suggesting several avenues for future work. Foremost among these is the imperative need for the development of an autotuner. Such an autotuner would not only optimize configurations but also dynamically adapt to the intricacies of different scenarios. Of particular significance is the exploration of methods to determine the optimal number of MPI ranks for the Euler equations using enclave tasking and task orchestration for fusion in a non-deterministic environment, ensuring adaptability to varying computational workloads. This autotuner should also take into consideration the cell size, patch size, and the available hardware at hand to solve the Euler equations using enclave tasking and task orchestration for fusion more efficiently.

The development of a dedicated profiler for enclave tasking represents another key future initiative. A task-specific profiler would provide granular insights into enclave tasking performance, enabling a more nuanced understanding that, in turn, can inform dynamic adjustments by the autotuner.

The dynamic adaptability of the autotuner itself is a crucial frontier. Future efforts should focus on enabling the autotuner to adjust configurations (number of enclave tasks to be fused and number of total patches/tasks) dynamically, responding to evolving runtime conditions. This adaptability is fundamental for optimizing performance across varying workloads and system states.

Addressing potential floating-point errors within the multi-GPU implementation is paramount to ensure the accuracy and stability of simulation results. A comprehensive examination and mitigation strategy for these errors would contribute significantly to the robustness of the framework.

Documentation improvement stands out as an essential aspect of future work, ensuring that researchers and practitioners have clear, comprehensive, and user-friendly guidance for leveraging the multi-GPU capabilities within ExaHyPE 2.

Further optimization opportunities lie in exploring the balance between asynchronous and synchronous MPI communication methods, resolving any syntax inconsistencies for enhanced code maintainability, implementing computation and communication overlap, and ensuring a robust reliance on MPI itself, independent of backend intricacies.

Finally, investigating and resolving anomalies or inefficiencies in GPU offloading when employing multiple MPI ranks represents a critical area for refinement. Understanding and addressing these challenges will contribute to the reliability and performance of the multi-GPU approach.

In essence, these future directions collectively aim to propel ExaHyPE 2 into new frontiers of efficiency and adaptability, establishing it as a robust framework for addressing complex simulations across diverse scientific and engineering domains.

Bibliography

[1] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Köppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl. "ExaHyPE: An Engine for Parallel Dynamically Adaptive Simulations of Wave Problems." In: (2019).

[2] S. Tian, J. Chesterfield, J. Doerfert, and B. Chapman. "Experience report: writing a portable GPU runtime with OpenMP 5.1." In: *IWOMP 2021*. Ed. by S. McIntosh-Smith, B. de Supinski, and J. Klinkenberg. Vol. 12870. LNCS. Springer, Cham, 2021, pp. 159–169. DOI: 10.1007/978-3-030-85262-7_11.

[3] J. Huber and et al. "Efficient Execution of OpenMP on GPUs." In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 41–52. DOI: 10.1109/CGO55325.2022.9419786.

[4] A. Dubey, M. Berzins, C. Burstedde, M. Norman, D. Unat, and M. Wahib. "Structured Adaptive Mesh Refinement Adaptations to Retain Performance Portability with Increasing Heterogeneity." In: *Computing in Science and Engineering* 23.5 (2021), pp. 62–66. DOI: 10.1109/MCSE.2021.3079378.

[5] T. Weinzierl. "The Peano Software—Parallel, Automaton-based, Dynamically Adaptive Grid Traversals." In: *ACM Transactions on Mathematical Software* 45 (2015). DOI: 10.1145/3319797.

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2007. ISBN: 978-0521880688.

[7] M. Bader. *Space-filling Curves: An Introduction With Applications in Scientific Computing*. 2013. DOI: 10.1007/978-3-642-31046-1.

[8] H. Schulz, G. Gadeschi, O. Rudyy, and T. Weinzierl. "Task Inefficiency Patterns for a Wave Equation Solver." In: *OpenMP: Enabling Massive Node-Level Parallelism*. Ed. by S. McIntosh-Smith, B. de Supinski, and J. Klinkenberg. Vol. 12870. Lecture Notes in Computer Science. Springer, Cham, 2021. DOI: 10.1007/978-3-030-85262-7_8.

[9] D. E. Charrier, B. Hazelwood, and T. Weinzierl. "Enclave Tasking for DG Methods on Dynamically Adaptive Meshes." In: *SIAM Journal on Scientific Computing* 42 (2020), pp. C69–C96. DOI: 10.1137/19M1276194.

[10] X. Qin, R. LeVeque, and M. Motley. "Accelerating an Adaptive Mesh Refinement Code for Depth-Averaged Flows Using GPUs." In: *J. Adv. Model. Earth Syst.* 11.8 (2019), pp. 2606–2628.

[11] M. Wille, T. Weinzierl, G. Brito Gadeschi, and M. Bader. "Efficient GPU Offloading with OpenMP for a Hyperbolic Finite Volume Solver on Dynamically Adaptive Meshes." In: *ISC High Performance 2023* (2023). DOI: 10.1007/978-3-031-32041-5_4.

[12] A. Eghtesad, K. Germaschewski, R. Lebensohn, and M. Knezevic. "A multi-GPU implementation of a full-field crystal plasticity solver for efficient modeling of high-resolution microstructures." In: *Computer Physics Communications* 254 (2020), p. 107231. DOI: 10.1016/j.cpc.2020.107231.

[13] C.-T. Yang, C.-L. Huang, and C.-F. Lin. "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters." In: *Computer Physics Communications* 182 (2011), pp. 266–269. DOI: 10.1016/j.cpc.2010.06.035.

[14] D. Jacobsen and I. Senocak. "Multi-level parallelism for incompressible flow computations on GPU clusters." In: *Parallel Comput.* 39 (2013), pp. 1–20.

[15] H.-Y. Schive, U.-H. Zhang, and T. Chiueh. "Directionally unsplit hydrodynamic schemes with hybrid MPI/OpenMP/GPU parallelization in AMR." In: *International Journal of High Performance Computing Applications* 26 (2011). DOI: 10.1177/1094342011428146.

[16] B. LAB. *Introduction to Performance Modelling*. URL: https://www.nersc.gov/assets/Uploads/Talk-PAM2018-Roofline.pdf.

[17] NVIDIA. *Deep Learning Performance Guide: Matrix Multiplication*. URL: https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html.

[18] J. D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. https://www.cs.virginia.edu/stream/.

[19] J. Gunnels, G. Henry, and R. van de Geijn. "High-Performance Matrix Multiplication Algorithms for Architectures with Hierarchical Memories." In: (July 2001).

List of Figures

List of Equations

List of Algorithms

Appendix

1 STREAM Benchmark

This section provides the C++ codes used for the STREAM triad benchmark using OpenMP *target* in Listing 8.1 and Hybrid MPI with OpenMP *target* in Listing 8.2.

1.1 OpenMP *target* Offloading

```
#include <cstdio>
#include <cstdlib>
#include <cerrno>
#include <chrono>
#include <algorithm>
#include <omp.h>

#ifndef STREAM_TYPE
#define STREAM_TYPE double
#endif

#ifndef OFFSET
#define OFFSET  0
#endif

#ifndef gpu_count
#define gpu_count 4
#endif

#ifndef POINTS
#define POINTS 1000
#endif

#ifndef GPU_CHECK_ID
#define GPU_CHECK_ID 0
#endif

double calculateMFLOPS(long STREAM_ARRAY_SIZE, double duration, long NTIMES) {
    return 2.0 * (double) STREAM_ARRAY_SIZE * (double) NTIMES * 1.0e-6 / duration;
}

double calculateResult(long STREAM_ARRAY_SIZE, double* vector) {
    double sum = 0;
    for (int i = 0; i < STREAM_ARRAY_SIZE; i++) {
```

```
35          sum += vector[i];
36      }
37      return sum;
38  }
39
40  void checkOffload() {
41      #pragma omp target device(GPU_CHECK_ID)
42      {
43          if (omp_is_initial_device()) {
44              printf("OFFLOAD UNSUCCESSFUL\n");
45          }
46      }
47  }
48
49  void runBenchmarks(long STREAM_ARRAY_SIZE, long NTIMES) {
50      STREAM_TYPE *a = new STREAM_TYPE[STREAM_ARRAY_SIZE+OFFSET];
51      STREAM_TYPE *b = new STREAM_TYPE[STREAM_ARRAY_SIZE+OFFSET];
52      STREAM_TYPE *c = new STREAM_TYPE[STREAM_ARRAY_SIZE+OFFSET];
53      STREAM_TYPE scalar = 2.0;
54      long CHUNK_SIZE = (STREAM_ARRAY_SIZE + gpu_count - 1) / gpu_count;
55      for(int i=0; i<gpu_count; i++) {
56          #pragma omp target enter data map(alloc:a[0:CHUNK_SIZE],b[0:CHUNK_SIZE],c[0:
        CHUNK_SIZE]) device(i)
57      }
58
59      #pragma omp parallel num_threads(gpu_count)
60      #pragma omp sections
61      {
62          for(int j=0; j<gpu_count; j++) {
63              #pragma omp section
64              {
65                  #pragma omp target device(j) nowait
66                  #pragma omp teams
67                  #pragma omp distribute parallel for
68                  for (long i = 0; i < CHUNK_SIZE; ++i) {
69                      b[i] = c[i] = i + j*CHUNK_SIZE;
70                  }
71              }
72          }
73      }
74      double start = omp_get_wtime();
75
76      #pragma omp parallel num_threads(gpu_count)
77      {
78          #pragma omp sections
79          {
80              for(int k=0;k<gpu_count;k++) {
81                  #pragma omp section
82                  {
83                      #if gpu_count == 1
84                          #pragma omp target device(k)
```

```
85                      #else
86                          #pragma omp target device(k) nowait
87                      #endif
88                      #pragma omp teams
89                      for (unsigned long j = 0; j < NTIMES; ++j)
90                      {
91                          #pragma omp distribute parallel for
92                          for (long i = 0; i <CHUNK_SIZE; ++i) {
93                              a[i] = b[i]+scalar*c[i];
94                          }
95                      }
96                  }
97              }
98          }
99      }
100     double stop = omp_get_wtime();
101     double duration= stop - start;
102     double triad_result = 0;
103
104     for(int i=0;i<gpu_count;i++) {
105         #pragma omp target exit data map(from: a[0:CHUNK_SIZE]) map(release:b[0:
        CHUNK_SIZE],c[0:CHUNK_SIZE]) device(i)
106         triad_result += calculateResult(CHUNK_SIZE, a);
107     }
108
109     double triad_mflops = calculateMFLOPS(STREAM_ARRAY_SIZE, duration, NTIMES);
110
111     delete[] a;
112     delete[] b;
113     delete[] c;
114
115     printf("| %10ld | %8.2f | %8ld | %.4e |\n", STREAM_ARRAY_SIZE, triad_mflops, NTIMES,
         triad_result);
116 }
117
118 int main(int argc, char *argv[]) {
119
120     if (argc < 1) {
121         printf("The two parameter STREAM_ARRAY_SIZE needs to be provided.\n");
122         exit(1);
123     }
124
125     char *pEnd;
126     long STREAM_ARRAY_SIZE = strtol(argv[1], &pEnd, 10);
127     long NTIMES = std::clamp(POINTS / STREAM_ARRAY_SIZE, 8l, 65536l);
128
129     checkOffload();
130
131     runBenchmarks(STREAM_ARRAY_SIZE, NTIMES);
132
133     return 0;
```

```
}
```

Listing 8.1: Triad benchmark with OpenMP *target*

1.2 Hybrid MPI with OpenMP *target* Offloading

```
#include <cstdio>
#include <cstdlib>
#include <cerrno>
#include <chrono>
#include <algorithm>
#include <omp.h>
#include <mpi.h>

#ifndef STREAM_TYPE
#define STREAM_TYPE double
#endif

#ifndef OFFSET
#define OFFSET 0
#endif

#ifndef POINTS
#define POINTS 1000
#endif

double calculateMFLOPS(long STREAM_ARRAY_SIZE, double duration, long NTIMES) {
    return 2.0 * (double) STREAM_ARRAY_SIZE * (double) NTIMES * 1.0e-6 / duration;
}

double calculateResult(long STREAM_ARRAY_SIZE, double* vector) {
    double sum = 0;
    for (int i = 0; i < STREAM_ARRAY_SIZE; i++) {
        sum += vector[i];
    }
    return sum;
}

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

    MPI_Init(&argc, &argv);
    int numProcesses, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (argc < 2) {
        if (rank == 0) {
            printf("The STREAM_ARRAY_SIZE parameter needs to be provided.\n");
        }
        MPI_Finalize();
```

```
45          exit(1);
46      }
47
48      char* pEnd;
49      long STREAM_ARRAY_SIZE = strtol(argv[1], &pEnd, 10);
50      long CHUNK = STREAM_ARRAY_SIZE / numProcesses;
51      long NTIMES = 1;
52      STREAM_TYPE scalar = 2.0;
53
54      #pragma omp target device(rank)
55      {
56          if (omp_is_initial_device()) {
57              printf("OFFLOAD UNSUCCESSFUL on GPU %d in process %d\n", rank, rank);
58          }
59      }
60
61      MPI_Barrier(MPI_COMM_WORLD);
62      STREAM_TYPE* a = nullptr;
63      STREAM_TYPE* b = nullptr;
64      STREAM_TYPE* c = nullptr;
65
66      if (rank == 0) {
67          a = new STREAM_TYPE[STREAM_ARRAY_SIZE + OFFSET];
68          b = new STREAM_TYPE[STREAM_ARRAY_SIZE + OFFSET];
69          c = new STREAM_TYPE[STREAM_ARRAY_SIZE + OFFSET];
70
71          #pragma omp parallel for
72          for (long i = 0; i < STREAM_ARRAY_SIZE; ++i) {
73              b[i] = c[i] = i;
74          }
75      }
76
77      STREAM_TYPE* local_a = new STREAM_TYPE[CHUNK + OFFSET];
78      STREAM_TYPE* local_b = new STREAM_TYPE[CHUNK + OFFSET];
79      STREAM_TYPE* local_c = new STREAM_TYPE[CHUNK + OFFSET];
80      MPI_Scatter(b, CHUNK, MPI_DOUBLE, local_b, CHUNK, MPI_DOUBLE, 0, MPI_COMM_WORLD);
81      MPI_Scatter(c, CHUNK, MPI_DOUBLE, local_c, CHUNK, MPI_DOUBLE, 0, MPI_COMM_WORLD);
82
83      std::chrono::system_clock::time_point start, stop;
84
85      #pragma omp target data map(tofrom: local_a[0:CHUNK]) map(to:local_b[0:CHUNK],
        local_c[0:CHUNK]) device(rank)
86      {
87          start = std::chrono::high_resolution_clock::now();
88          #pragma omp target device(rank)
89          #pragma omp teams
90          for (unsigned long j = 0; j < NTIMES; ++j) {
91              #pragma omp distribute parallel for
92              for (long i = 0; i < CHUNK; ++i) {
93                  local_a[i] = local_b[i] + scalar * local_c[i];
94              }
```

```
95          }
96          stop = std::chrono::high_resolution_clock::now();
97      }
98      auto duration = std::chrono::duration<double>(stop - start).count();
99
100     MPI_Gather(local_a, CHUNK, MPI_DOUBLE, a, CHUNK, MPI_DOUBLE, 0, MPI_COMM_WORLD);
101
102     #pragma omp target exit data map(release: local_a[0:CHUNK]) map(release: local_b[0:
        CHUNK], local_c[0:CHUNK]) device(rank)
103
104     if (rank == 0) {
105         double triad_result = 0;
106         triad_result = calculateResult(STREAM_ARRAY_SIZE, a);
107         double triad_mflops = calculateMFLOPS(STREAM_ARRAY_SIZE, duration, NTIMES);
108         delete[] a;
109         delete[] b;
110         delete[] c;
111
112         printf("| %10ld | %8.2f | %8ld | %.4e |\n", STREAM_ARRAY_SIZE, triad_mflops,
        NTIMES, triad_result);
113     }
114
115     delete[] local_a;
116     delete[] local_b;
117     delete[] local_c;
118
119     MPI_Finalize();
120
121     return 0;
122 }
```

Listing 8.2: Triad benchmark with hybrid MPI and OpenMP *target*

2 Matrix-Matrix Multiplication Benchmark

This section provides the C++ codes used for the Matrix-Matrix multiplication benchmark using OpenMP *target* in Listing 8.4 and Hybrid MPI with OpenMP *target* in Listing 8.2.

2.1 OpenMP *target* Offloading

```
1 #include <iostream>
2 #include <omp.h>
3 #include <cstdlib>
4 #include <chrono>
5
6 #ifndef num_devices
7 #define num_devices 4
```

```
8  #endif
9
10 int main(int argc, char** argv) {
11     if (argc < 2) {
12         printf("The N parameter needs to be provided.\n");
13         exit(1);
14     }
15     char* pEnd;
16     int N = strtol(argv[1], &pEnd, 10);
17
18     double* A = new double[N * N];
19     double* B = new double[N * N];
20     double* C = new double[N * N]();
21     double* temp_C = new double[N * N]();
22
23     int chunk_size = (N / num_devices);
24
25     for (int dev = 0; dev < num_devices; dev++) {
26         #pragma omp target enter data map(alloc: temp_C[0: chunk_size * N], A[:N * N], B
       [0: chunk_size * N]) device(dev)
27     }
28
29     #pragma omp parallel num_threads(num_devices)
30     #pragma omp sections
31     {
32         for (int dev = 0; dev < num_devices; dev++) {
33             #pragma omp section
34             {
35                 #pragma omp target device(dev)
36                 #pragma omp teams distribute parallel for
37                 for (int i = 0; i < N; i++) {
38                     for (int j = 0; j < N; j++) {
39                         A[i * N + j] = 4.0;
40                     }
41                     for (int j = 0; j < chunk_size; j++) {
42                         B[j * N + i] = 2.0;
43                     }
44                 }
45             }
46         }
47     }
48
49     std::chrono::system_clock::time_point start, stop;
50
51     start = std::chrono::high_resolution_clock::now();
52
53     #pragma omp parallel num_threads(num_devices)
54     #pragma omp sections
55     {
56         for (int dev = 0; dev < num_devices; dev++) {
57             #pragma omp section
```

```
58              {
59                  #if gpu_count == 1
60                      #pragma omp target device(dev)
61                  #else
62                      #pragma omp target device(dev) nowait
63                  #endif
64                  #pragma omp teams distribute parallel for
65                  for (int i = 0; i < N; i++) {
66                      for (int j = 0; j < chunk_size; j++) {
67                          for (int k = 0; k < N; k++) {
68                              temp_C[j * N + i] += A[i * N + k] * B[j * N + k];
69                          }
70                      }
71                  }
72              }
73          }
74      }
75
76      stop = std::chrono::high_resolution_clock::now();
77
78      auto duration = std::chrono::duration<double>(stop - start).count();
79
80      for (int dev = 0; dev < num_devices; dev++) {
81          int start_col = dev * chunk_size;
82
83          #pragma omp target exit data map(from: temp_C[0: chunk_size * N]) map(release: A
    [:N * N], B[0: chunk_size * N]) device(dev)
84
85          #pragma omp parallel for
86          for (int i = 0; i < N; i++) {
87              for (int j = 0; j < chunk_size; j++) {
88                  C[(j + start_col) * N + i] += temp_C[j * N + i];
89              }
90          }
91      }
92
93      double mflops = (2.0 * (double)N * (double)N * (double)N * 1.0e-9) / duration;
94      double bw = (mflops) / N;
95
96      printf("| %10ld | %8.2f | %8.2f  |\n", N, mflops, bw);
97
98      delete[] A;
99      delete[] B;
100     delete[] C;
101     delete[] temp_C;
102
103     return 0;
104 }
```

Listing 8.3: Matrix-matrix multiplication with OpenMP *target*

2.2 Hybrid MPI with OpenMP *target* Offloading

```
#include <iostream>
#include <omp.h>
#include <cstdlib>
#include <mpi.h>
#include <chrono>

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    int numProcesses, rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);

  if (argc < 2) {
    if (rank == 0) {
      printf("The N parameter needs to be provided.\n");
    }
    MPI_Finalize();
    exit(1);
    }

    char* pEnd;
    int N = strtol(argv[1], &pEnd, 10);

    int chunk_size = (N / numProcesses);
    double* A = new double[N * N];
    double* B = new double[N * N];
    double* C = new double[N * N]();

    double* temp_C = new double[chunk_size * N]();

    #pragma omp target enter data map(alloc: temp_C[0: chunk_size * N], A[:N * N], B[0:
    chunk_size * N]) device(rank)

    #pragma omp target device(rank)
    #pragma omp teams distribute parallel for
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i * N + j] = 4.0;
        }
        for (int j = 0; j < chunk_size; j++) {
            B[j * N + i] = 2.0;
        }
    }

    std::chrono::system_clock::time_point start, stop;

    start = std::chrono::high_resolution_clock::now();
  #pragma omp target device(rank)
  #pragma omp teams distribute parallel for
```

```
49   for (int i = 0; i < N; i++) {
50     for (int j = 0; j < chunk_size; j++) {
51       for (int k = 0; k < N; k++) {
52         temp_C[j * N + i] += A[i * N + k] * B[j * N + k];
53       }
54     }
55   }
56     stop = std::chrono::high_resolution_clock::now();
57
58     auto duration = std::chrono::duration<double>(stop - start).count();
59
60     #pragma omp target exit data map(from: temp_C[0: chunk_size * N]) map(release: A[:N
       * N], B[0: chunk_size * N]) device(rank)
61
62     MPI_Gather(temp_C, chunk_size * N, MPI_DOUBLE, C, chunk_size * N, MPI_DOUBLE, 0,
       MPI_COMM_WORLD);
63
64     double mflops = (2.0 * (double)N * (double)N * (double)N * 1.0e-6) / duration;
65     double bw = mflops / N;
66
67     delete[] A;
68     delete[] B;
69     delete[] C;
70     delete[] temp_C;
71
72     if(rank == 0) {
73         printf("| %10d | %8.2f | %8.2f |\n", N, mflops, bw);
74     }
75
76     MPI_Finalize();
77
78     return 0;
79 }
```

Listing 8.4: Matrix-matrix multiplication with hybrid MPI and OpenMP *target*