DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics, Cognition, Intelligence

Evaluation of reduced-footprint memory layouts for the hyperbolic PDE solver ExaHyPE2

Xing Zhou

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics, Cognition, Intelligence

Evaluation of reduced-footprint memory layouts for the hyperbolic PDE solver ExaHyPE2

Bewertung speichereffizienter Speicherlayouts für den hyperbolischen PDE-Solver ExaHyPE2

Author:	Xing Zhou
Supervisor:	Prof. Dr. Michael Bader
Advisor:	M.Sc. Marc Marot-Lassauzaie
Submission Date:	Feburary 01, 2025

I confirm that this master's thesis in robotics, cognition, intelligence is my own work and I have documented all sources and material used.

Munich, Feburary 01, 2025 Xing Zhou

Acknowledgments

I would like to express my deepest gratitude to my advisor, Mr. Marc Marot-Lassauzaie. His invaluable guidance and support were crucial in the successful completion of this work. His insightful ideas and technical expertise greatly inspired me and helped me overcoming the challenges encountered throughout this research.

I would also like to extend my sincere thanks to Mr. Pawel Radtke for his enthusiastic support and invaluable assistance. His expertise with the usage of modified LLVM compiler and suggestions for the GPU optimization was essential to the progress of this work.

I am also grateful to my family and friends for their steadfast support, patience, and encouragement during this work. Their belief in me provided the emotional strength needed to persevere through the ups and downs of this work.

Abstract

Graphics Processing Units (GPUs) are indispensable in modern high-performance computing (HPC) systems due to their exceptional parallel processing capabilities. However, their performance is often constrained by the overhead of memory transfers between the CPU and GPU, particularly in memory-bound tasks. This study investigates two optimization techniques—bit-level data packing and buffer aggregation—to improve memory transfer efficiency in ExaHyPE2, an open-source framework for solving hyperbolic partial differential equations.

Experimental results demonstrate that buffer aggregation is the most effective strategy for optimizing GPU kernel execution in ExaHyPE2, significantly reducing synchronization overhead. Multi-threading was also explored to overlap memory transfers and kernel execution, but it yielded minimal benefits. While data packing alone did not show substantial performance improvements in ExaHyPE2 kernels, it proved highly effective for memory-bound tasks and exhibited enhanced performance when combined with buffer aggregation.

The proposed optimization strategies offer valuable insights into addressing memory transfer challenges and have broad applicability to HPC applications requiring efficient GPU offloading.

Contents

1. Introduction

GPUs are specifically designed for parallel processing, making them particularly well-suited for computationally intensive tasks such as numerical simulations, deep learning, and large-scale data processing. In the realm of exascale computing, GPUs play a pivotal role, serving as the primary source of computational power in modern high-performance computing (HPC) systems. To fully leverage the potential of GPU acceleration and achieve optimal performance, exascale simulation software must be carefully optimized for GPU offloading.

To fully leverage the computational power of GPUs, tasks with a high degree of parallelism should be executed on the GPU whenever possible. This process, referred to as "GPU offloading" however,is often hindered by the significant overhead associated with memory transfers between the CPU and GPU. For many memory-bound tasks, these data transfers constitute a substantial portion of the overall execution time. Consequently, optimizing the memory layout becomes a critical factor in improving the performance of GPU-accelerated programs.

We investigate ExaHyPE2 ("An Exascale Hyperbolic PDE Engine") [17], an open-source software framework designed for solving and simulating systems of first-order hyperbolic partial differential equations (PDEs). We explore several approaches to optimize memory layout for efficient data transfer.

The first approach involves packing data at the bit level using a modified LLVM/-Clang compiler [16]. Packing here just means compressing the data. This technique aims to minimize the data size, thereby reducing the overhead of memory transfer, while maintaining the required level of accuracy. Unlike traditional data compression, this method offers fine-grained control over the degree of compression by operating directly at the bit level. Unnecessary bits that do not contribute to the required accuracy are discarded, and the number of compressed bits can be adjusted dynamically to meet varying precision requirements. Additionally, this approach is significantly faster than conventional data compression algorithms due to its simplicity, relying primarily on truncation and bit-level operations. More details on the data packing method can be found in [15].

The second optimization method is designed to address the issue of excessively scattered data distribution required for a single kernel invocation in ExaHyPE2. We copy the all the patches (data that need to be transferred to GPU for one kernel execution) into a large buffer, and then copy this buffer to the GPU in a single operation using just one API call, rather than copying the smaller patches individually which ExaHyPE2 originally does. Although this approach incurs some additional time to copy all the data into the buffer, it significantly reduces the overhead associated with multiple API calls and synchronization, leading to overall performance improvements.

Our study suggests that the second approach (Buffer Aggregation) is highly effective for optimizing ExaHyPE2 GPU kernels. While data packing alone did not significantly impact the ExaHyPE2 GPU kernel benchmark, its effectiveness was notably enhanced when combined with buffer aggregation. We also explored the use of multi-threading to handle multiple large buffers, but this approach did not yield expected results, despite some overlap between memory transfer and kernel execution.

Although motivated by ExaHyPE2, we believe the approaches presented in this paper are applicable to a broad range of similar HPC applications that require efficient GPU offloading.

The remainder of this document is organized as follows: Chapter 2 provides the necessary background knowledge for understanding this paper. Chapter 3 reviews relevant related works. In Chapter 4, we present the concepts and implementation details of the proposed optimization methods. Chapter 5 reports the experimental results, analyzes the effectiveness of these methods, and discusses the factors contributing to their performance. Chapter 6 summarizes the key findings of this study and explores potential directions for future research.

2. Fundamentals

2.1. Hyperbolic PDEs and Finite Volume Method

Hyperbolic PDEs, typically derived from conservation laws in physics, are fundamental for describing wave phenomena. Examples include electromagnetic waves water waves, and seismic waves, etc. Hyperbolic PDEs have a tight relationship with our everyday life and enhancing the simulation speed of hyperbolic PDEs holds significant scientific and practical importance.

ExaHyPE2 accepts first-order hyperbolic partial differential equations

$$\frac{\partial Q}{\partial t} + \nabla \cdot F(Q) + \sum_{i=1}^{d} B_i(Q) \frac{\partial Q}{\partial x_i} = S(Q) \tag{2.1}$$

Here, Q is the state vector representing all physical quantities (such as density, velocity, energy, etc.); $F(Q)$ is the flux function describing the flow of these quantities; $B_i(Q)$ represents the dependence on the spatial variable x_i (e.g., diffusion terms); and $S(Q)$ is the source term representing external effects or other influences. The number of dimensions is denoted by d [10].

The Finite Volume Method (FVM) [11] is a numerical approach designed to solve PDEs. In FVM, the computational domain is divided into discrete grid cells, and the method approximates the cell averages of the solution variable by integrating over each grid cell. The evolution of these cell averages over time is governed by the fluxes through the edges of the grid cells. The core challenge in FVM lies in determining accurate numerical flux functions that approximate the true fluxes using the available cell averages.

Unlike classical finite difference methods, which approximate derivatives at specific grid points and often fail near discontinuities, FVM is based on the integral form of the governing equations focusing on accurately approximating solutions that may contain discontinuities. High-resolution FVMs, in particular, have proven to be highly effective in capturing discontinuous solutions with minimal numerical oscillations [6].

2.2. ExaHyPE2

ExaHyPE2 [17, 21] is a computational framework designed to solve hyperbolic partial differential equations (PDEs) with high efficiency and scalability. Built upon dynamically adaptive Cartesian meshes, it employs explicit time-stepping methods and adaptive mesh refinement (AMR) to handle complex simulations. The framework is powered by Peano [20], a project that leverages a spacetree data structure to construct and manage dynamically adaptive grids. Peano optimizes computational efficiency by linking grid storage directly with traversal operations and enforces a unique traversal order through space-filling curves, which improves both memory usage and parallel performance.

Spatial Discretization ExaHyPE2's spatial discretization is based on dynamically adaptive Cartesian meshes, implemented using an octree-based AMR approach. The computational domain is embedded within a cube and recursively subdivided into smaller cubes at each refinement level, which can be efficiently refined or coarsened based on the simulation's requirements. This grid structure adapts dynamically at each time step, enabling the framework to handle evolving solutions with precision and efficiency.

Solvers While ExaHyPE2 supports various numerical discretization methods, including Finite Volume (FV), Runge-Kutta Discontinuous Galerkin (RKDG), and ADER-DG methods [22], we focus on Finite Volume, with a Rusanov solver [14] for handling the flux calculations. The solver can operate with 2D and 3D regular Cartesian meshes, referred to as "patches," embedded within each cube. This approach, combining tree structure for mesh management and patch-based discretization, allows for a reasonable arithmetic load relative to the mesh management overhead. [7] This balance is crucial for efficient computation, especially when scaling to large numbers of processing units.

The update for each patch is computed based on the source term and the fluxes across the volume faces, as described in the equation (2.1). These updates are handled through a callback mechanism, where users can inject their domain knowledge into the simulation. This modular design hides all other program logic, including mesh traversal, data storage, and parallelization, from the user, aligning with the Hollywood principle – "don't call us, we'll call you" [21].

GPU Acceleration The inherently parallelizable nature of its numerical methods – such as FV, RKDG, and ADER-DG – makes the framework well suited for utilizing GPUs as accelerators, enabling significant speedup in simulations. ExaHyPE2 utilizes a sophisticated GPU offloading strategy to further enhance performance. Tasks are pooled into a separate queue, and once the queue reaches a threshold, the tasks are offloaded to the GPU in a batch. This approach ensures that the GPU can handle a large number of tasks simultaneously, maximizing its parallel computational potential. The pooling or batching of tasks allows for the execution of high-concurrency GPU compute kernels, making it possible to handle even small enclaves scattered across subdomains. The batch size is well chosen to optimize GPU utilization, striking a balance between GPU load and task overlap, ensuring efficient computation across all available processing units.

2.3. OpenMP GPU offloading

OpenMP provides a high-level, pragma-based approach to parallel programming. Starting from OpenMP 4.0, support for GPU offloading was introduced, allowing developers to annotate specific regions of code to be executed on GPUs. OpenMP enables a unified programming experience across CPUs and GPUs with minimal code modification.

OpenMP offloading adopts a host-centric execution model, where the host manages the scheduling, synchronization, and memory management for tasks executed on target devices. On the device side, execution begins with the main thread of a team . The `teams directive` in OpenMP enables the creation of a league of teams, where each team starts with a single main thread. The `distribute` directive assigns tasks across teams. Additional threads within a team can be spawned using the `parallel` directive. The `for` directive schedules loop iterations among threads within a team.

This hierarchical organization aligns closely with GPU execution models. GPUs consist of multiple streaming multiprocessors (SMs), each capable of running numerous threads grouped into execution units such as warps (on NVIDIA GPUs) or wavefronts (on AMD GPUs). In practice, OpenMP maps teams to SMs and threads within a team to hardware threads inside the SM. This mapping mirrors the hierarchical structure of GPU execution, optimizing the utilization of computational resources.

OpenMP also provides a series of control clauses to manage memory transfer between host and device. For example, we can use `target enter data` and `target exit data` clause to transfer data between host and device.

Code 2.1 shows an example of a simple OpenMP GPU offloading program containing the basic syntax of OpenMP GPU offloading.

```
// X, Y are two arrays with N doubles in it. a is a double scalar.
// We want to use GPU to parallelize Z = a * X + Y

#pragma omp target enter data map(to: X[0 : N], Y[0 : N])
#pragma omp target enter data map(alloc: Z[0 : N])

#pragma omp target distribute parallel for simd
for (int i = 0; i < N; ++i) {
    Z[i] = a * X[i] + Y[i];
}

#pragma omp target exit data map(from : Z[0 : N])
#pragma omp target exit data map(delete: X[0 : N], Y[0 : N])
```

Listing 2.1: A simple GPU offloading program

3. Related Work

There has been significant progress in making OpenMP GPU offloading increasingly efficient. Carlo B. et al. [2] Samuel F. et al. [1] integrated GPU support for OpenMP Offloading Directives into Clang, allowing GPU offloading through simple OpenMP directives without needing torewrite the whole program using CUDA or SYCL. What's more, it allows adding more functions based on the clang compiler with OpenMP GPU offloading function. Alok M. et al. [13] points out that in many cases the OpenMP USM model can cause bad performance. Several guidelines for programmers to achieve better performance are also given. Artem C. et al. [4] presents ideas that focus on improving the execution of high-level parallel code in GPUs, including a code transformation that sets up a pipeline of kernel execution and asynchronous data transfer. This transformation enables the overlap of communication and computation. With overlap between data transfer and kernel execution, the total kernel time may be reduced further in multi-threading case.

Lossy data compression techniques have been applied on high-performance computing (HPC) applications. Calhoun et al. [3] explored the feasibility of lossy compression in checkpoint-restart workflows for PDE simulations, demonstrating that compression errors can be masked by numerical truncation errors, enabling effective restarts without compromising simulation accuracy. Lindstrom and Isenburg [12] focused on fast and efficient compression methods for floating-point data, emphasizing online and lossless compression schemes to mitigate I/O bottlenecks in large-scale simulations, while maintaining high throughput and effective bandwidth. Building on these efforts, Tao et al. [18] introduced a multidimensional prediction-based lossy compression algorithm, significantly improving prediction accuracy and compression factors for irregular scientific datasets while ensuring strict error control. These works collectively underscore the potential of advanced compression techniques to address the challenges of data size and I/O in modern HPC environments.

Pawel L. et al. [15] implemented a language extension based on C++, compressing the data in bit-level, which to some extent overcome a shortage in C++ that fixed-size built-in type cannot control the bits they use to represent data with all kinds of value ranges or precision. Thomas G. et al. [8] introduces a wrapper of data in C++ named "memory accessor" which hides the data compression behind the memroy access. It breaks up the tight coupling between the precision format used for arithmetic operations and the storage format employed for memory operations. Mario W. et al. [21] proposed a method to effectively manage GPU memory for multi-threading cases, in order to overcome an OpenMP bottleneck in the administration of GPU memory. Although small data patches are batched together for kernel execution, the data still stays on different locations in memory and they are accessed with pointers. This causes unnecessary overhead when the data is transferred onto GPU as smalls chunks.

4. Methods and Implementation

To optimize the memory layout of ExaHyPE2 GPU offloading, we need to identify what is really necessary for the memory transfer and what can be optimized. The objective is to ensure that we only pay the computational and memory costs for what is essential. This leads to two key questions:

1. Can the size of the data being transferred be further reduced while ensuring that the computation remains unaffected and the results remain accurate?

2. How efficient is the memory transfer? Is there room for further optimization? Are there additional unnecessary overheads that can be eliminated or optimized?

These questions guide us toward the optimization methods introduced in this section.

ExaHyPE2 still faces several challenges that need to be addressed to improve its performance.

In order to increase the GPU utilization, ExaHyPE2 does not transfer patch data, which is generated on the host device, onto the GPU immediately. Instead it waits until a sufficient number of patches are generated and transfers all of them at once via one kernel execution. This strategy can potentially increase the GPU usage efficiency.

Currently, ExaHyPE2 uses OpenMP's `target enter data` constructs [19, 9] to transfer patches one by one for one kernel execution. However, this approach is inefficient because multiple small patches, scattered across different memory locations, need to be transferred. As a result, the number of memory transfers significantly exceeds the number of kernel executions, often by several orders of magnitude. This creates substantial overhead due to API calls and synchronization.

Another major challenge arises from the disparity between memory bandwidth and processor computation power. Over the years, the arithmetic performance of processors has grown much faster than memory bandwidth. Unless there is a revolutionary breakthrough in chip technology, this trend is expected to persist. Consequently, memory-bound programs often achieve lower performance in terms of floating-point operations per second (FLOPS) compared to compute-bound programs.

This phenomenon is well captured by the roofline model [5], as illustrated in Figure 4.1. The roofline model shows that for some algorithms, performance can be severely constrained by low memory bandwidth. Moreover, an algorithm that is compute-bound on one machine may become memory-bound on another with a different hardware configuration. This means the algorithm cannot achieve its peak potential speed and is instead limited by the memory bandwidth of the system.

The same issue affects ExaHyPE2, where the performance of simulations can be heavily constrained by the bandwidth of CPU-GPU data transfers. Even on systems with high CPU-GPU bandwidth, this limitation could become more prominent in the future as computational power continues to grow. To address this, we aim to improve the efficiency of memory transfers in ExaHyPE2, effectively raising the bandwidth limit as described by the roofline model (see Figure 4.1) and enabling programs to run closer to their peak performance.

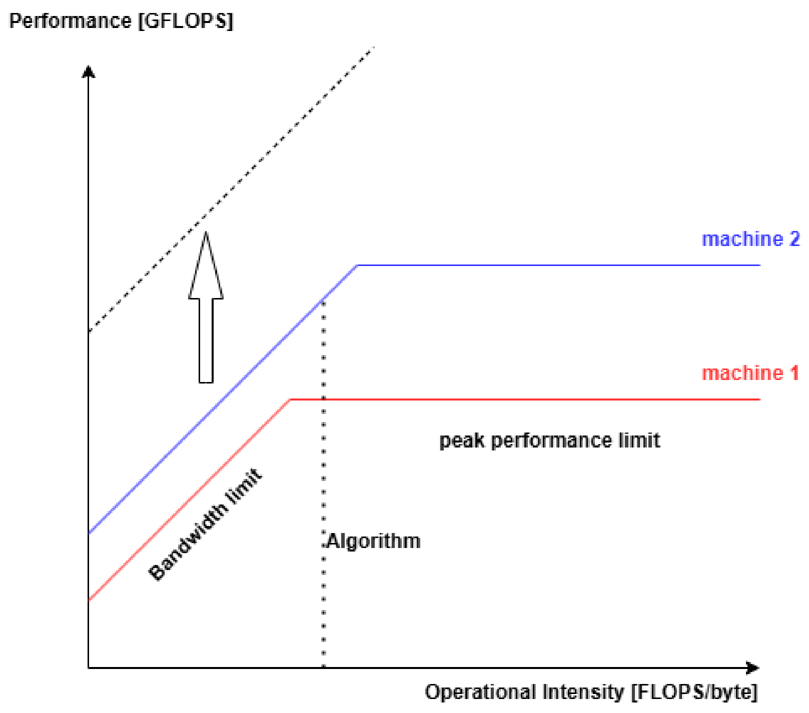

Figure 4.1.: Roofline model predicting the performance of an algorithm in two different machines. The algorithm is compute bound in machine 1 and memory bound in machine 2. Through optimized memory layouts the bandwidth limit will move upwards, allowing the algorithm to achieve higher performance in machine 2.

To address the challenges outlined above, we explore several optimization approaches, which can be combined to achieve a more significant overall performance improvement.

4.1. Data structure and Baseline

Before delving into the details of the optimization methods, it is important to first understand the data structure of patches in ExaHyPE2 and the mechanics of GPU offloading.

Each patch contains data, denoted as $\boldsymbol{Q}$, which resides in the heap memory and is represented by a pointer to a double (*double∗*). As previously mentioned, multiple patches can be grouped into batches to enable more efficient computations. Additionally, there is a data structure called "CellData," which is responsible for managing a batch of patches. The structure of "CellData" is as follows:

```
struct CellData {
   int numberOfCells;
   double** QIn;
   double** QOut;
   // some other data members ...
};
```

Listing 4.1: Shortened implementation of the CellData class

Here, ***numberOfCells*** specifies the batch size, ***QIn*** points to an array of pointers, where each pointer refers to an input patch data $\boldsymbol{Q}$ located in the heap memory. Similarly, ***QOut*** points to an array of pointers, where each pointer refers to a pre-allocated patch data buffer into which the updated $\boldsymbol{Q}$ will be copied.

The original GPU offloading algorithm, referred to as the "baseline," is presented in Algorithm 1. In this approach, patches are transferred to the GPU one by one, computations are performed, and the resulting patch data is then copied back to the CPU one at a time.

Algorithm 1 Algorithm GPU kernel invocation baseline

```
1: Input: QIn, N (number of patches, batch size)
2: Output: QOut
3:
4: for $i \leftarrow 0$ to $N$ do
5:     copy $QIn[i]$ onto GPU in $QInGPU[i]$
6: end for
7:
8: Do computations on $QInGPU$
9: $QOutGPU \leftarrow$ Result
10:
11: for $i \leftarrow 0$ to $N$ do
12:     copy $QOutGPU[i]$ onto CPU in $QOut[i]$
13: end for
```

4.1.1. Implementation

A simplified implementation of the Baseline method is provided below, with non-essential code omitted for clarity and conciseness. Refer to Code 4.2.

Note that we use `mappedQIn` and `mappedQOut` to represent pointers to the data patches on the GPU. After transferring all the data to the GPU, it is necessary to transfer `mappedQIn` and `mappedQOut` separately as well.

Lines 5–14 transfer the patches to the GPU one by one. Lines 16 perform computations through a GPU kernel invocation. Finally, lines 19–26 copy the necessary results from the GPU to the CPU and release resources on the GPU.

4.2. Data Packing

The first optimization method is data packing. This approach involves compressing the data on the CPU before transferring it to the GPU and decompressing it on the GPU prior to performing the actual computations. Similarly, when transferring the results from the GPU back to the CPU, the data is compressed on the GPU and subsequently decompressed on the CPU before further use.

4.2.1. Bit level compression

Before delving into the details of the method, it is essential to understand how bit-level compression works. Since ExaHyPE2 uses `double` as the floating-point number type,

```
double** mappedQIn = new double*[numPatches];
double** mappedQOut = new double*[numPatches];

// copy patches (QIn) to GPU one by one
for (int o = 0; o < numPatches; ++i) {
    const double* currentQIn = patchData.QIn[i];
    double* currentQOut = patchData.QOut[i];
    #pragma omp target enter data map(to: currentQIn[0:InSize])
    #pragma omp target enter data map(alloc: currentQOut[0:OutSize])
    mappedQIn[i] = omp_get_mapped_ptr(currentQIn, targetDevice);
    mappedQOut[i] = omp_get_mapped_ptr(currentQOut, targetDevice);
}
#pragma omp target enter data map(to: mappedQIn[0:numPatches])
#pragma omp target enter data map(to: mappedQOut[0:numPatches])

// gpu kernel (omp target clause) ...

// copy the result patches (QOut) on CPU one by one
#pragma omp target exit data map(delete: mappedQOIn[0:numPatches])
#pragma omp target exit data map(delete: mappedQOut[0:numPatches])
for (int i = 0; i < numPatches; i++) {
    const double* currentQIn = patchData.QIn[i];
    double* currentQOut = patchData.QOut[i];
    #pragma omp target exit data map(delete: currentQIn[0:InSize])
    #pragma omp target exit data map(from: currentQOut[0:OutSize])
}

delete[] mappedQIn;
delete[] mappedQOut;
```

Listing 4.2: Shortened code of GPU offloading in ExaHyPE2

this discussion focuses exclusively on the compression of variables with the `double` type. For compression techniques applied to other data types, please refer to [15].

According to the IEEE standard, a double-precision floating-point number consists of 64 bits, where 1 bit represents the sign, 11 bits represent the exponent, and 52 bits represent the mantissa (fraction). The structure of a 64-bit double-precision floating-point number is illustrated in Figure 4.2.

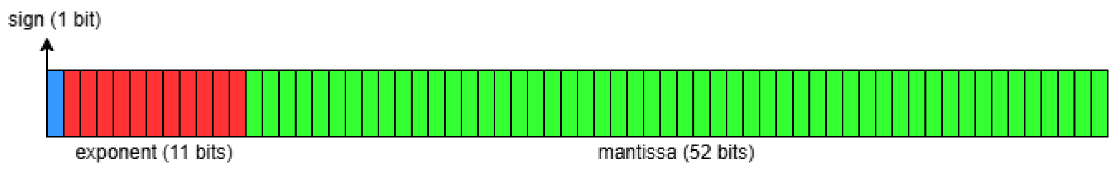

Figure 4.2.: Bit structure of a 64-bit double floating-point number

The real value represented by a 64 bit double is calculated in following formula

$$(-1)^{sign}(1.b_{51}b_{50}...b_0)_2 \times 2^{e-1023}$$

or

$$(-1)^{sign}(1 + \sum_{i=1}^{52} b_{52-i}2^{-i}) \times 2^{e-1023}$$

In many cases, the full precision of the mantissa is not required to represent a floating-point number. The modified LLVM compiler [15] provides a method to truncate certain bits of the mantissa, enabling lossy compression. By using the annotation `[[clang::truncate_mantissa(BITS)]]`, the mantissa can be stored with only *BITS* bits, while the exponent and the sign bit retain their original precision. For performance reasons, computations must adhere to the built-in data formats, and all operations are automatically handled by the compiler.

Since this is a lossy compression method, the final accuracy of the results must be evaluated to determine the appropriate number of mantissa bits for the GPU kernel in ExaHyPE2.

4.2.2. Implementation

The pseudo-code in Algorithm 2 illustrates the process of GPU offloading and kernel invocation with data packing.

Algorithm 2 Algithm GPU kernel inovocation with packing

```
1: Input: QIn, N (number of patches, batch size)
2: Output: QOut
3:
4: for $i \leftarrow 0$ to $N$ do
5:     pack $QIn[i]$ into $QInPacked[i]$
6: end for
7:
8: for $i \leftarrow 0$ to $N$ do
9:     copy $QInPacked[i]$ into $QInPackedGPU[i]$
10: end for
11:
12: // GPU kernel
13: for $i \leftarrow 0$ to $N$ do
14:     unpack $QInPackedGPU[i]$ into $QInGPU[i]$
15: end for
16:
17: // GPU kernel
18: Do computations based on $QInGPU$ on GPU and store the result on GPU in
    $QOutGPU$...
19:
20: // GPU kernel
21: for $i \leftarrow 0$ to $N$ do
22:     pack $QOutGPU[i]$ into $QOutPackedGPU[i]$
23: end for
24:
25: for $i \leftarrow 0$ to $N$ do
26:     copy $QOutPackedGPU[i]$ onto CPU into $QOutPacked[i]$
27: end for
28:
29: for $i \leftarrow 0$ to $N$ do
30:     unpack $QOutPacked[i]$ into $QOut[i]$
31: end for
```

We use a wrapper class `PackedDouble` to implement the compression of a `double` scalar. See code 4.3.

```cpp
struct PackedDouble {
    [[clang::truncate_mantissa(20)]]
    double _d;

    PackedDouble() : _d(0.0) {}
    PackedDouble(double other)
    {
        _d = other;
    }
    operator double() const
    {
        return _d;
    }
    PackedDouble& operator=(double other)
    {
        _d = other;
        return *this;
    }
};
```

Listing 4.3: Shortened implementation of the PackedDouble class for data packing

The only member of `PackedDouble` is a `double`, which contains the entire 64 bits of data. During memory transfer operations, some bits may be automatically truncated. We also provide user-defined conversion functions to ensure a smooth transition between `PackedDouble` and `double`.

Additionally, we need some simple code to pack and unpack the data on both the CPU and GPU.

```cpp
// unpack QIn on GPU
#pragma omp target teams distribute parallel for simd collapse(2)
for (int patch = 0; patch < numPatches; ++patch) {
    for (int i = 0; i < InSize; ++i) {
        mappedQIn[patch][i] =
            mappedQInPacked[patch][i]._d;
    }
}
```

Listing 4.4: GPU kernel for unpacking

```
// pack QOut on GPU
#pragma omp target teams distribute parallel for simd collapse(2)
for (int patch = 0; patch < numPatches; ++patch) {
    for (int i = 0; i < OutSize; ++i) {
        mappedQOutPacked[patch][i]._d =
            mappedQOut[patch][i];
    }
}
```

Listing 4.5: GPU kernel for packing

In fact, if we do not add additional unpacking code, the `PackedDouble` can be directly used as a `double` during computation (the runtime will handle the unpacking automatically), and the code will still execute correctly. However, for certain kernels, the same data may be accessed multiple times, leading to repeated decompression. This can result in unnecessary overhead and slow down execution.

Additionally, the strategy of automatically unpacking at runtime leads to interface inconsistencies, requiring extensive modifications to the computation functions in ExaHyPE2. A more compatible interface (possibly using templates or the `auto` keyword) may be designed in the future to increase the extensibility of the program.

4.3. Buffer Aggregation

As discussed previously, the original method of managing data is not optimal. Too many small data chunks introduce repeated overhead, slowing down data transfer and access.

The optimization strategy "Buffer Aggregation" involves first copying all the small data chunks into one large buffer before transferring the entire buffer to the GPU in a single operation. This strategy leaves the content of the data untouched but requires only one invocation of the memcpy API. By reducing the overhead from multiple API calls and synchronization, this approach can significantly improve performance. The process of buffer aggregation is shown in Algorithm 3.

Algorithm 3 Algithm GPU kernel inovocation with buffer aggregation

```
1: **Input:** QIn, N (number of patches, batch size)
2: **Output:** QOut
3:
4: allocate a huge buffer $QInHuge$
5:
6: for $i \leftarrow 0$ to $N$ do
7:     copy $QIn[i]$ into the place $QInHuge[i * patchSize]$ (in a sequence)
8: end for
9:
10: copy $QInHuge$ onto GPU as $QInHugeGPU$
11:
12: // GPU kernel
13: for $i \leftarrow 0$ to $N$ do
14:     point $QInGPU[i]$ to $QInHugeGPU[i * patchSize]$
15:     point $QOutGPU[i]$ to $QOutHugeGPU[i * patchSize]$
16: end for
17:
18: // GPU kernel
19: Do computations based on $QInGPU$ on GPU and store the result on GPU in $QOutGPU$...
20:
21: copy $QOutHugeGPU$ onto CPU into $QOutHuge$
22:
23: for $i \leftarrow 0$ to $N$ do
24:     point $QOut[i]$ to $QOutHuge[i * patchSize]$
25: end for
```

4.3.1. Implementation

We designed a wrapper class `GPUCellData` to wrap the original data structure `CellData` in order to apply the buffer aggregation operation on the data. It automatically handles buffer aggregation and GPU offloading, hiding the implementation details in the constructor and member functions, while leaving the caller-side code clear and minimally modified.

```cpp
struct GPUCellData {
    int numberOfCells;
    // Points to a huge buffer that stores all the data
    double* QInCopyInOneHugeBuffer;
    double* QOutCopyInOneHugeBuffer;
    // QIn and QOut contain pointers that point to the positions
    // of patches in the huge buffer
    double** QIn;
    double** QOut;

    // allocate memory on gpu and
    // copy the data from cpu (hostCellData object) onto gpu
    GPUCellData(const CellData& hostCellData);

    // Copying GPU Data is forbidden.
    GPUCellData(const GPUCellData&) = delete;
    GPUCellData& operator=(const GPUCellData&) = delete;

    // release the resources
    ~GPUCellData();

    // copy the data from gpu back to cpu into the hostCellData object
    void copyToHost(CellData& hostCellData);
};
```

Listing 4.6: Shortened implementation of the wrapper class GPUCellData

When a `GPUCellData` is constructed from a `CellData`, it automatically allocates the relevant memory on the GPU and performs the following operations: 1) copies all the patches into a large buffer on the CPU, and 2) transfers the large buffer to the GPU. Furthermore, it retains the interface of *QIn* and *QOut* with type `double**`, so the calculation code remains unmodified. After the computation, we only need to call `copyToHost()` once to copy the data back from the GPU. When the lifetime of the `GPUCellData` object ends, the memory on the GPU is automatically released through the destructor. Thanks to the wrapper, the code for GPU offloading is clearer and more elegant compared to before (see 4.2 and 4.7).

As the implementations of the member functions is too long to put here, more details can be referred in appendix A.

```
GPUCellData gpuPatchData(patchData);

// GPU kernel ...

gpuPatchData.copyToHost(patchData);
```

Listing 4.7: Shortened code of GPU offloading in ExaHyPE2 with Buffer Aggregation

4.3.2. Several huge buffers and Multi-threads

Based on Buffer Aggregation, we could use several large buffers instead of just one, where each buffer contains multiple data patches. We can then use multi-threading on the CPU to handle data transfer and kernel execution simultaneously for each buffer. Since each patch can be transferred and computed independently, we may overlap the data transfer of one large buffer with the kernel execution of another, thereby increasing GPU utilization efficiency.

We slightly change the data structure from `GPUCellData` to `GPUCellDataAsync` (See code 4.6 and 4.8).

```
struct GPUCellDataAsync {
    int numberOfCells;
    // Points to a huge buffer that stores all the data
    double* QInCopyInOneHugeBuffer;
    double* QOutCopyInOneHugeBuffer;
    // QIn and QOut contain pointers
    // that point to different positions in the huge buffer
    double** QIn;
    double** QOut;

    GPUCellDataAsync(
        int _targetDevice,
        int _numberOfCells,
        double** _QIn,
        double** _QOut,
        ... // other parameters
    );
    // ... other member variables and functions
};
```

Listing 4.8: Shortened structure of GPUCellDataAsync

We accept `double** QIn` and `double** QOut` as parameters in the constructor, instead of a `CellData` object. This allows us to construct multiple `GPUCellDataAsync` objects, rather than just one, from a `CellData` object. We then assign OpenMP tasks, with each task responsible for the GPU offloading of one `GPUCellDataAsync` object (constructed from a subset of patches). These tasks are automatically executed simultaneously by the OpenMP task mechanism (see code 4.9).

```
int nThreads = 8;
int numberOfCellsPerThread = patchData.numberOfCells / nThreads;

#pragma omp parallel
#pragma omp single
for (int threadId = 0; threadId < nThreads; ++threadId) {
    #pragma omp task
    {
    auto gpuPatchData = GPUCellDataAsync(
        targetDevice,
        numberOfCellsPerThread,
        patchData.QIn + threadId * numberOfCellsPerThread,
        patchData.QOut + threadId * numberOfCellsPerThread,
        ... // other parameters
    )

    // GPU kernel ...

    gpuPatchData.copyToHost();
    }
}
```

Listing 4.9: Shortened code of GPU offloading with several huge buffers and multi-threads

4.4. Combination

4.4.1. Implementation

We used a structure `GPUCellDataPacked` to implement the combination of the two methods. Most of the implementation is similar to that of `GPUCellData` (see 4.6), except that we use `PackedDouble*` instead of `double*` for the buffer on the GPU.

```
struct GPUCellDataPacked {
   int numberOfCells;
   // Points to a huge buffer that stores all the data
   PackedDouble* QInCopyInOneHugeBufferPacked;
   PackedDouble* QOutCopyInOneHugeBufferPacked;
   double* QInCopyInOneHugeBuffer;
   double* QOutCopyInOneHugeBuffer;
   // QIn and QOut contain pointers that point to the positions
   // of patches in the huge buffer
   double** QIn;
   double** QOut;
};
```

Listing 4.10: Shortened code of GPU offloading in ExaHyPE2 with Buffer Aggregation

4.5. Controlling number of memory transfer and computation

Another effective way to reduce the overhead caused by memory transfer is by decreasing the number of memory transfers relative to the number of computations (kernel calls).

The original algorithm transfers data back from the GPU to the CPU immediately after the calculation on the GPU is done. This means the number of memory transfers (back and forth) is equal to the number of computations. While transferring the data back to the CPU is necessary for certain operations, such as saving the results, it is unnecessary if we only need the final result at the end of the time period and do not care about the intermediate values. In some cases, we may only want the result data at specific time steps, rather than at every time step. In such cases, frequent data transfer between the CPU and GPU is unnecessary.

We can introduce a new parameter to control the ratio between computation steps and memory transfer steps. For example, if we set the ratio to 5, this means the data stays on the GPU for five kernel execution steps before being transferred to the CPU for other purposes, such as plotting. This can ideally reduce the time consumed by data transfer to 1/5 of the original time.

4.5.1. Implementation

We add a template parameter `IterationPerTransfer` to the function signature for GPU offloading and loop the GPU kernel call for `IterationsPerTransfer` times. As all allocation, data transfer and release are done before or after kernel call, this way the result will iterate `IterationsPerTransfer` steps before finally transferred back to CPU.

```
void timeStepWithRusanovPatchwiseHeapStateless<
    ..., // other tempate parameters
    IterationsPerTransfer>
    (int targetDevice, CellData& patchData)
{
    // ... other codes
    for (int i = 0; i < IterationsPerTransfer; ++i) {
        internal::timeStepWithRusanovPatchwiseStateless(
            ... // parameters
        );
    }
    // ... other codes
}
```

Listing 4.11: Shortened code of GPU offloading in ExaHyPE2

5. Results

Experimental Setup In order to validate the effectiveness of these methods, we isolated the GPU kernel from the rest of the project as much as possible to minimize unrelated interference and focus primarily on the analysis of GPU offloading. We varied the number of patches (batch size) $N_{patches}$, the number of threads $N_{threads}$ used to offload parts of the data to the GPU simultaneously, and the patch size S, which determines the size of each patch. We measured the total runtime of the kernel invocations and repeated the experiments multiple times to reduce random errors. These experiments were conducted on a set of equations, including the Euler equation, acoustic-wave equation, and elastic-wave equation, with similar results obtained across all cases. We used NVIDIA Nsight Systems as the performance optimization tool.

All tests were run on a local machine equipped with an Intel i7-14650HX CPU (16 cores) and an NVIDIA RTX 4060 Laptop GPU. The C++ compiler used was a modified LLVM/Clang compiler (designed to support bit-level data compression at the language level), based on the official Clang 18 [16]. The source code of the modified LLVM/Clang compiler is available at `github.com/pradt2/llvm-project/tree/hpc-ext`. Our experiment utilized Compute Unified Device Architecture (CUDA) version 12.4 as the underlying software stack. The meaningful test results are presented and explained in this section.

5.1. Data Packing

We studied data packing not only in the ExaHyPE2 GPU kernel but also in two other simpler cases: SAXPY and Matrix Multiplication (MM). These represent two typical types of HPC programs — memory-bound and compute-bound, respectively. In contrast, ExaHyPE2 is a more complex case, where the effects of optimization may depend on the specific equation and a variety of other parameters. The effects of data packing were studied in all three cases.

5.1.1. SAXPY

The algorithm of SAXPY is shown in Algorithm 4.

Algorithm 4 SAXPY Algorithm

```
Require: Vectors $X$, $Y$ of size $n$, scalar $a$
Ensure: Updated vector $Y$
1: for $i = 1$ to $n$ do
2:     $Y[i] \leftarrow a \cdot X[i] + Y[i]$
3: end for
4: return $Y$
```

SAXPY is a memory-bound program, meaning that most of the time is spent on data transfer between the CPU and GPU. This is confirmed by the GPU time summary from Nsight Systems, shown in Table 5.1. The program spent 98.3% of the GPU hardware time on data transfer, leaving only 1.7% for kernel execution.

Time Percentage	Total Time	Operation
68.7%	284.477 ms	memcpy Host to Device
29.5%	122.229 ms	memcpy Device to Host
1.7%	7.174 ms	kernel (compute)

Table 5.1.: saxpy GPU time summary without packing. The size of the vectors is 2^{22}. The offloading and calculation is repeated for 20 times.

Programs where the bottleneck is caused by the large size of data to be transferred can benefit the most from data compression. We conducted experiments with packing *mantissa* = 20, which means that the packed `double` now has only 32 bits of data. As expected, the transfer size was reduced by half. This is confirmed by the profiling results shown in Tables 5.2 and 5.3.

Total Size	Operation
1.25 GB	memcpy Host to Device
640.00 MB	memcpy Device to Host

Table 5.2.: saxpy total size of data transferred without Packing. The size of the vectors is 2^{22}. The offloading and calculation are for 20 times.

Total Size	Operation
640.00 MB	memcpy Host to Device
320.00 MB	memcpy Device to Host

Table 5.3.: saxpy total size of data transferred with Packing, 20 bits of mantissa. The size of the vectors is 2^{22}. The offloading and calculation are for 20 times.

As a result, the total time consumed was significantly reduced, as shown in Figure 5.4. Although there is some additional kernel time introduced by the packing and unpacking process, it is negligible compared to the reduction in data transfer time — from 284.47ms / 122.23ms to 79.59ms / 40.17ms.

Time Percentage	Total Time	Operation
56.2%	79.593 ms	memcpy Host to Device
28.3%	40.174 ms	memcpy Device to Host
6.5%	9.175 ms	kernel (compute)
3.1%	4.460 ms	kernel (packing or unpacking)
3.1%	4.412 ms	kernel (packing or unpacking)
2.8%	3.922 ms	kernel (packing or unpacking)

Table 5.4.: saxpy GPU time summary with packing, 20 bits of mantissa. The size of the vectors is 2^{22}. The offloading and calculation are for 20 times.

We use Mean Absolute Error (MAE, also known as L1 loss) to estimate the precision of the result. It is calculated with the formula

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - x_i|$$

The MAE of the results with different numbers of mantissa bits is shown in Table 5.5. A mantissa of 20 indicates that the packed double is reduced to half of its original size, while a mantissa of 52 means no compression is applied.

mantissa	MAE
20	2.54e-4
28	1.01e-6
36	3.87e-9
44	1.56e-11
52	1.36e-14

Table 5.5.: The precision of result with different bits of mantissa by Dense Matrix Multiplication.

In conclusion, simple memory-bound programs, where data transfer consumes most of the time, can benefit greatly from data compression.

5.1.2. Matrix Multiplication

Dense Matrix Multiplication (MM) is a typical example for compute bound programs. We use the naive algorithm for the calculation to ensure that it is compute-bound, allowing us to compare the results with those of SAXPY.

Algorithm 5 Naive Matrix Multiplication

```
Require: Matrices $A$ of size $M \times K$, $B$ of size $K \times N$
Ensure: Matrix $C$ of size $M \times N$, where $C = AB$
1: Initialize $C[i][j] \leftarrow 0$ for $i = 1, \dots, N$ and $j = 1, \dots, P$
2: for $i = 1$ to $M$ do
3:     for $j = 1$ to $N$ do
4:         for $k = 1$ to $K$ do
5:             $C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j]$
6:         end for
7:     end for
8: end for
```

Figure 5.6 shows the GPU time summary for a Matrix Multiplication (MM) program. The kernel time accounts for 75.2% of the total time.

Time Percentage	Total Time	Operation
75.2%	954.103 ms	kernel (compute)
16.4%	205.666 ms	memcpy Host to Device
8.4%	106.167 ms	memcpy Device to Host

Table 5.6.: Matrix multiplication GPU time summary without packing. The size of matrices are $2^{10} \times 2^{10}$. The offloading and calculation are for 100 times.

Although Dense MM is compute-bound, some acceleration can still be gained from data packing. From Tables 5.6 and 5.7, we can see that the percentage of time spent on data transfer decreases from 24.8% to 13.9%, while the kernel time remains unchanged (the additional kernel time for data packing and unpacking is negligible). The benefits of optimization are less significant than in memory-bound programs like SAXPY, but they are not insignificant. On one hand, it does reduce the total runtime of the program. On the other hand, compute-bound programs are likely to become memory-bound in the future due to the rapid growth in processor computing power. As a result, this optimization will become increasingly beneficial moving forward.

Time Percentage	Total Time	Operation
85.7%	951.820 ms	kernel (compute)
0.2%	1.944 ms	kernel (packing or unpacking)
0.1%	1.779 ms	kernel (packing or unpacking)
0.1%	1.470 ms	kernel (packing or unpacking)
8.9%	98.491 ms	memcpy Host to Device
5.0%	55.292 ms	memcpy Device to Host

Table 5.7.: Matrix multiplication GPU time summary with packing, 20 bits of mantissa. The size of matrices are $2^{10} \times 2^{10}$. The offloading and calculation are for 100 times.

The precision of the results with different numbers of mantissa bits is shown in Table 5.5. A mantissa of 20 indicates that the packed double is reduced to half of its original size, while a mantissa of 52 means no compression is applied.

mantissa	MAE
20	1.03e-6
28	4.07e-9
36	1.57e-11
44	6.12e-14
52	4.55e-17

Table 5.8.: The precision of result with different bits of mantissa by SAXPY.

When compared to the precision of Dense MM (see Table 5.5), SAXPY achieves more accurate results with the same number of mantissa bits. This suggests that for programs involving simpler computations (often memory-bound), the data packing method is more suitable and effective.

5.1.3. ExaHyPE2 GPU kernel

The GPU kernel in ExaHyPE2 is much more complex than those in SAXPY and MM.

Table 5.9 shows the GPU time summary for the baseline (without data packing). We can observe that the time spent on data transfer and kernel execution are nearly the same. The parameters used were: patch size = 16, number of patches = 2048, and the experiment was repeated 10 times.

Time Percentage	Total Time	Operation
45.0%	47.282 ms	kernel (compute)
27.7%	29.152 ms	memcpy Host to Device
27.3%	28.723 ms	memcpy Device to Host

Table 5.9.: GPU time summary of calling ExaHype2 kernel with FV Rusanov solver without packing. The patch size is 16 and the number of patches is 2048. The offloading and calculation are for 10 times.

After adding data packing, the results are shown in Table 5.10.

Time Percentage	Total Time	Operation
46.7%	47.347 ms	kernel (compute)
21.7%	21.943 ms	memcpy Device to Host
18.4%	18.598 ms	memcpy Host to Device
6.7%	6.780 ms	kernel (packing or unpacking)
6.6%	6.643 ms	kernel (packing or unpacking)

Table 5.10.: GPU time summary of calling ExaHype2 kernel with FV Rusanov solver with packing. The bits of mantissa is 20. The patch size is 16 and the number of patches is 2048. The offloading and calculation are for 10 times.

We use a mantissa of 20 bits so that the total size of the data is compressed by half. While the data transfer time is reduced, the additional kernel time for data packing and unpacking is not negligible. As a result, although the pure data transfer time decreases, the total runtime of the program does not decrease significantly. In fact, with different patch sizes, the total runtime may even increase. This can be observed in Figures 5.1 and 5.2, where the runtimes for different patch sizes are compared.

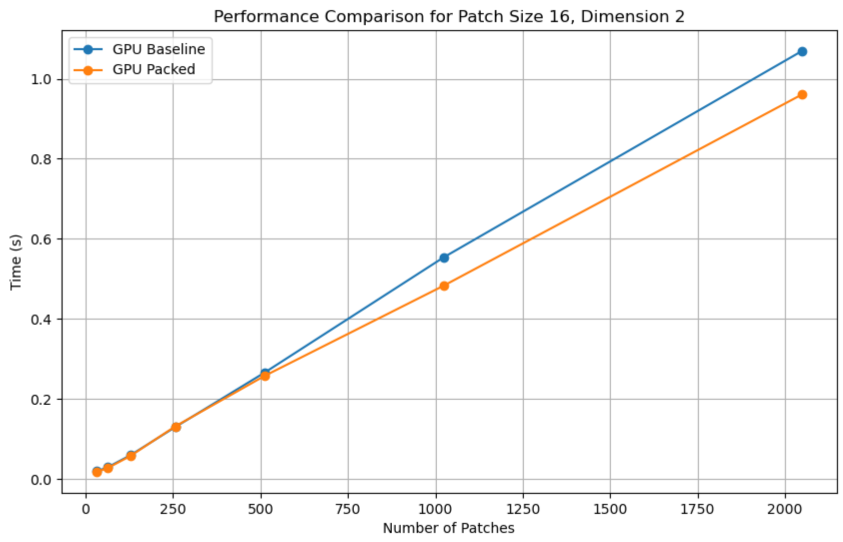

Figure 5.1.: Total run time of calling ExaHype2 kernel with FV Rusanov solver with different number of patches. The patch size is 16.

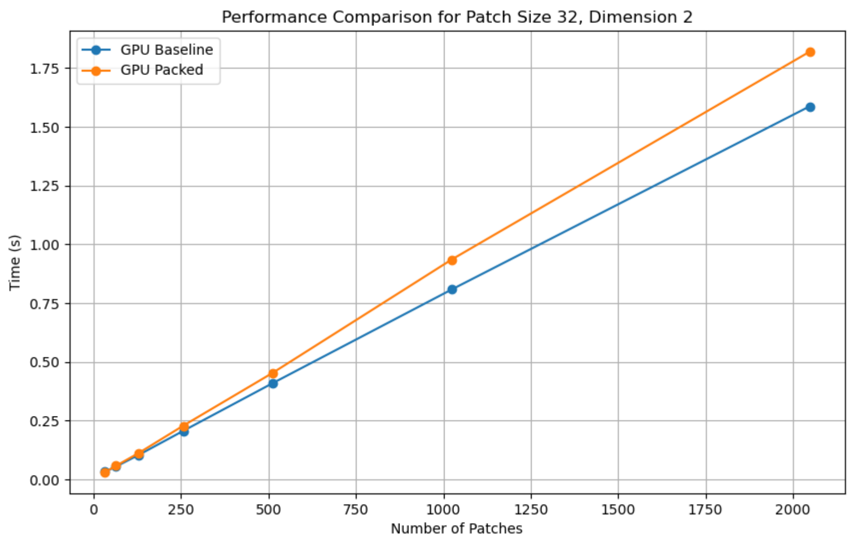

Figure 5.2.: Total run time of calling ExaHype2 kernel with FV Rusanov solver with different number of patches. The patch size is 32.

One possible reason for the poor performance is that there are too many small data chunks to transfer. For each small data chunk, a CUDA API call needs to be invoked, which introduces overhead in addition to the actual GPU time for copying data. Furthermore, the synchronization of these data transfers also consumes significant time. Figure 5.3 shows the timeline graph from Nsight System. The two small red blocks represent the GPU hardware time for two data copies. The red and yellow blocks below represent API times. We can observe that most of the time is wasted on API overhead and synchronization. This is the issue we address with Buffer Aggregation.

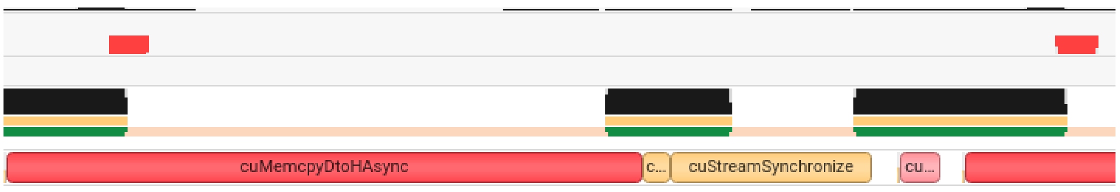

Figure 5.3.: Data transfer graph between two small data chunks. The two small red blocks represent the GPU time used for two data copies. The block below are API times.

5.2. Buffer Aggregation

Experiments prove that using buffer aggregation yields excellent results. Figures 5.4 and 5.5 show the comparison between the baseline and buffer aggregation. With a patch size of 4, we achieved a speedup of about 15x. Even with a patch size of 32, we achieved a speedup of about 2.5x. It is also easy to understand why buffer aggregation achieves more significant results when the patch size is small.

Experiments proves that that using buffer aggregation yields excellent results. Figure 5.4 and 5.5 show the comparison between the baseline and buffer aggregation. By patch size = 4, we got a speedup about 15x. Even by patch size = 32, we got a speedup about 2.5x.

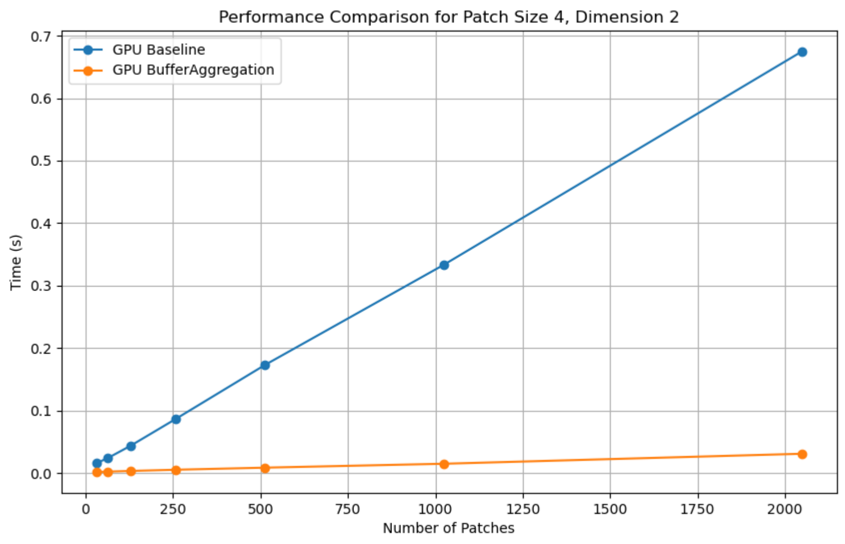

Figure 5.4.: Total run time of calling ExaHype2 kernel with FV Rusanov solver with different number of patches. The patch size is 4.

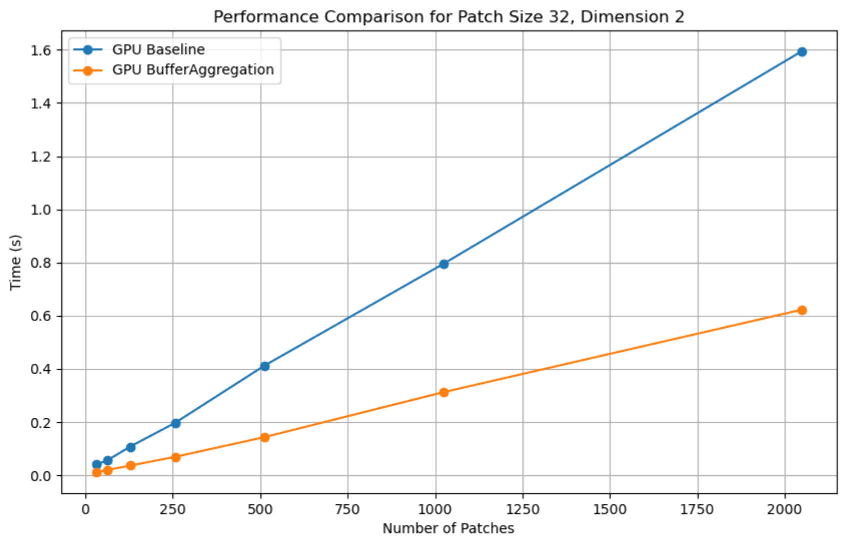

Figure 5.5.: Total run time of calling ExaHype2 kernel with FV Rusanov solver with different number of patches. The patch size is 32.

Several huge buffers

However, things do not always go smoothly. We implemented this idea using OpenMP tasks, and experiments show that breaking one huge buffer into several does not reduce the total runtime at all; in fact, it even slows down the execution speed. Figures 5.6 and 5.7 show the execution graphs from Nsight System. These graphs indicate that overlap did occur, but the efficiency of the overlap is not high enough. On the other hand, more buffers to transfer and more kernels to execute result in additional overhead, lowering efficiency. As a result, the total runtime increases. This is counterintuitive.

Due to the complexity of the program, the time saved may not compensate for the extra time consumed by synchronization among threads. However, multi-threading may still be valuable in multi-GPU cases.

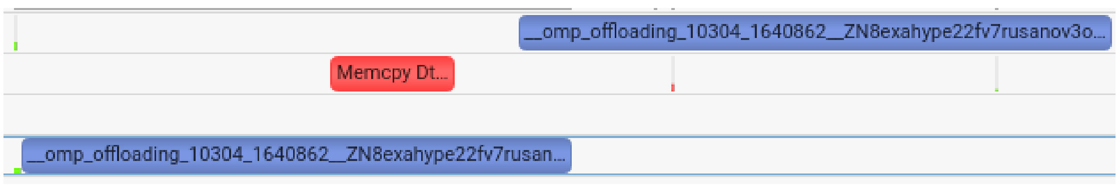
__omp_offloading_10304_1640862__ZN8exahype22fv7rusanov3o...
Memcpy Dt...
__omp_offloading_10304_1640862__ZN8exahype22fv7rusan...

Figure 5.6.: The execution graph from Nsight System with several huge buffers

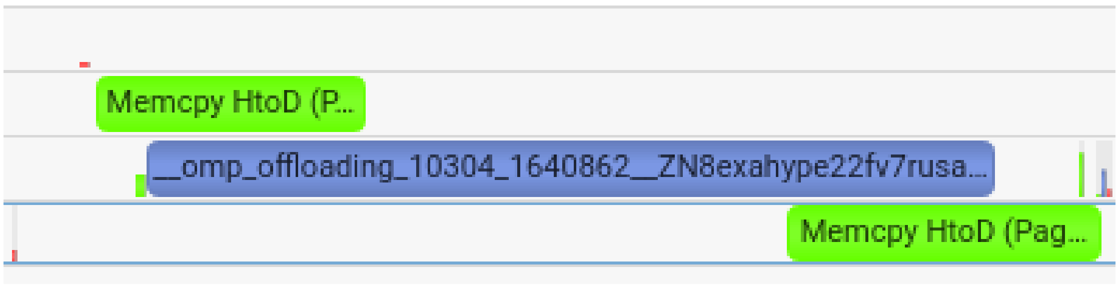

Figure 5.7.: Total run time of calling ExaHype2 kernel with FV Rusanov solver with different number of patches. The patch size is 32.

However, using several huge buffers may still work with multiple GPUs. Since data transfer and kernel execution can be completely independent for each GPU, parallelizing with multiple GPUs is theoretically efficient. However, relevant experiments have not been conducted yet due to time constraints and the lack of GPUs. Further research can be done in this area.

5.3. Combination

In this section, we combine the two optimization strategies discussed above. These two methods are not conflicting; instead, they can enhance each other's effectiveness. Without buffer aggregation, the impact of data packing is not very significant because the actual hardware memory transfer time is overshadowed by the additional overhead of the API and synchronization (see section 4.2). Now that buffer aggregation can, to some extent, eliminate the impact of this overhead, the effectiveness of data packing becomes more apparent.

Figures 5.8 and 5.9 show the results of combining data packing and buffer aggregation. We can observe that when the patch size is small, such as 4, data packing does not provide further optimization beyond buffer aggregation. However, as the patch size increases, the effect of data packing becomes more significant. For example, when the patch size grows to 36, data packing can provide an additional 15% performance gain.

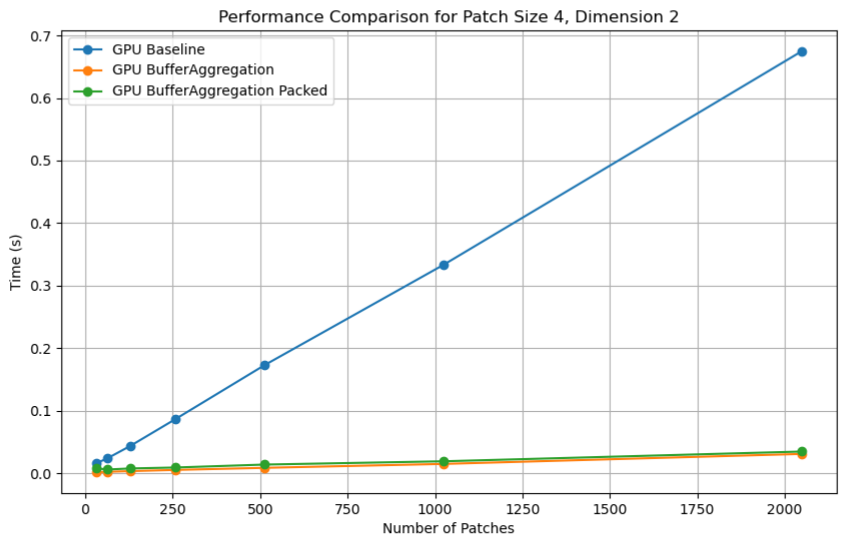

Figure 5.8.: Total run time of calling ExaHype2 kernel with FV Rusanov solver with different number of patches. The patch size is 4.

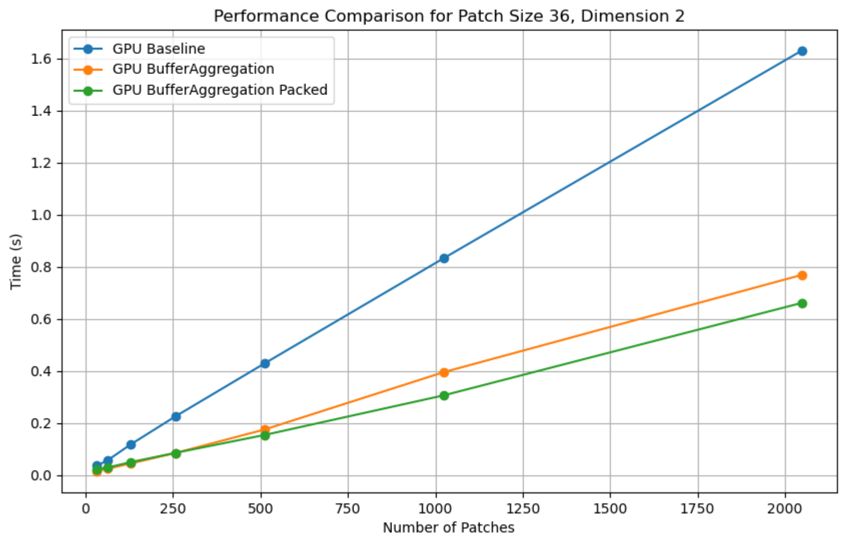

Figure 5.9.: Total run time of calling ExaHype2 kernel with FV Rusanov solver with different number of patches. The patch size is 36.

6. Conclusion and Outlook

6.1. Conclusions

Based on our experimental results, we can draw several conclusions that summarize the key findings of our work.

Conclusion 1 Buffer aggregation significantly reduces the overhead of API calls and memory transfer synchronization, leading to substantial performance improvements.

Our results suggest that the synchronization time required for numerous small data transfers consumes the majority of the memory transfer time. The idea behind buffer aggregation is to reduce synchronization overhead as much as possible, at the cost of some additional data copying. This strategy effectively eliminates a significant portion of the overhead.

Additional optimizations in the handling of the bits can also be considered.

Conclusion 2 Data packing alone does not yield significant improvements but provides additional optimization when combined with buffer aggregation.

In the baseline approach, synchronization is the most time-consuming part. The effect of optimizations in other areas becomes noticeable only when the synchronization overhead is minimized (i.e., well-optimized). This suggests a multi-layered optimization strategy: optimization should start with addressing the bottleneck. Just because a strategy or method does not yield immediate results does not mean it is ineffective; its benefits may be masked by other factors.

Conclusion 3 The impact of using multi-threading to overlap multiple data structures is limited.

We aimed to use multi-threading to overlap memory transfers and kernel executions, but we did not observe any significant optimization. Similar results were reported by [21]. This is likely due to dynamic memory allocation within the GPU kernel. To ensure thread-safe memory allocation and kernel execution, synchronization must be introduced. As a result, complete overlap of memory transfer and kernel execution is not feasible, explaining why the impact of multi-threading is negligible in this context.

6.2. Outlook

CUDA and SYCL are two other popular languages for GPU offloading. CUDA has been developed over many years and is specifically designed for NVIDIA GPUs. Although it is not multi-platform, it is highly optimized for the hardware, providing fine control over the program's behavior (such as shared memory and thread behavior within a warp), which cannot be easily achieved with OpenMP. SYCL, on the other hand, is a relatively new language designed for heterogeneous computing. In recent years, the performance of SYCL programs has improved significantly, and in some cases, it has even surpassed CUDA in certain applications (see Figure 6.1).

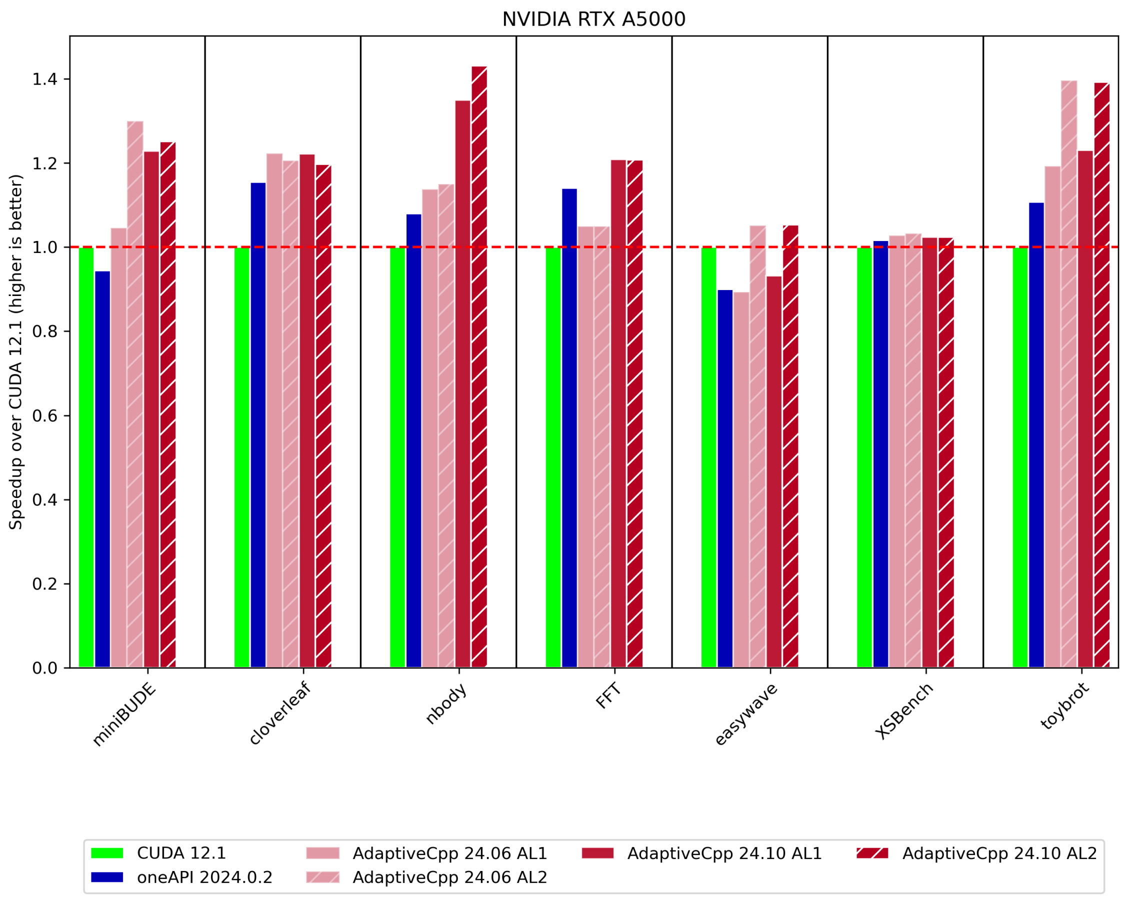

Figure 6.1.: Performance comparison between CUDA and oneAPI on NVIDIA GPUs. AdaptiveCpp and oneAPI are two implementations of SYCL.

To date, our work has been based on OpenMP GPU offloading. Future work could involve implementing the optimization methods using CUDA and SYCL, followed by a comparison of their performance. Moreover, CUDA and SYCL offer more functionalities and finer control over program behavior, which may enable further optimizations.

Currently, the data packing methods require an additional kernel to perform data packing and unpacking on the GPU, introducing extra overhead. A promising approach for future work would be to allow the runtime to perform data packing and unpacking dynamically when the data is actually used, rather than performing it centrally beforehand. However, this raises concerns about data reuse, as the same data might be packed or unpacked multiple times. This is an interesting avenue for further research.

Since we have only tested all the optimization methods on a single-GPU platform, it would be valuable to investigate whether these results hold on a multi-GPU platform. In real-world usage, multi-GPU platforms are often preferred. Furthermore, since there is no need for synchronization between multiple GPUs, the approach using multi-threading with multiple large buffers may yield better results in such a setup.

A. Implementation of GPUCellData

```
struct GPUCellData {
    int numberOfCells;

    // Points to a huge buffer that stores all the data
    double* QInCopyInOneHugeBuffer;
    double* QOutCopyInOneHugeBuffer;

    // QIn and QOut contain pointers
    // that point to somewhere in the huge buffer
    double** QIn;
    double** QOut;

    GPUCellData() = delete;

    // allocate memory on gpu and
    // copy the data from cpu (hostCellData object) onto gpu
    GPUCellData(
        const CellData& hostCellData,
        const enumerator::AoSLexicographicEnumerator& inEnumerator,
        const enumerator::AoSLexicographicEnumerator& outEnumerator,
        int _targetDevice
    );

    // Copying GPU Data is forbidden.
    GPUCellData(const GPUCellData&) = delete;
    GPUCellData& operator=(const GPUCellData&) = delete;

    // For convenience I deleted move constructor and move assignment.
    // It can be implemented in future if needed.
    GPUCellData(GPUCellData&&) = delete;
    GPUCellData& operator=(GPUCellData&&) = delete;

    ~GPUCellData();


    // copy the data from gpu back to cpu into the hostCellData object
    // and release memory on gpu
    void copyToHost(
        CellData& hostCellData,
        const enumerator::AoSLexicographicEnumerator& outEnumerator,
    ) const;
};

exahype2::fv::rusanov::omp::GPUCellData::~GPUCellData()
{
    omp_target_free(QIn, targetDevice);
    omp_target_free(QOut, targetDevice);
    omp_target_free(QInCopyInOneHugeBuffer, targetDevice);
    omp_target_free(QOutCopyInOneHugeBuffer, targetDevice);
}

exahype2::fv::rusanov::omp::GPUCellData::GPUCellData(
    const CellData& hostCellData,
    const enumerator::AoSLexicographicEnumerator& inEnumerator,
    const enumerator::AoSLexicographicEnumerator& outEnumerator,
    int _targetDevice
) : targetDevice(_targetDevice)
{
    numberOfCells = hostCellData.numberOfCells;

    int hostDevice = omp_get_initial_device();

    QIn = (double**)omp_target_alloc(numberOfCells * sizeof(double*),
        targetDevice);
    QOut = (double**)omp_target_alloc(numberOfCells * sizeof(double*),
        targetDevice);
    QInCopyInOneHugeBuffer = (double*)omp_target_alloc(
        numberOfCells * inEnumerator.size() * sizeof(double),
        targetDevice);
    QOutCopyInOneHugeBuffer = (double*)omp_target_alloc(
        numberOfCells * outEnumerator.size() * sizeof(double),
        targetDevice);

    // copy the host QIn into one huge buffer
    double* tmpQIn = new double[numberOfCells * inEnumerator.size()];
    for (int i = 0; i < numberOfCells; ++i) {
        std::memcpy(&tmpQIn[i * inEnumerator.size()],
            hostCellData.QIn[i], inEnumerator.size() * sizeof(double));
    }

    #pragma omp target teams distribute parallel for simd
        firstprivate(inEnumerator, outEnumerator, QIn, QOut,
        QInCopyInOneHugeBuffer, QOutCopyInOneHugeBuffer)
        device(targetDevice)
    for (int i = 0; i < numberOfCells; ++i) {
        QIn[i] = &QInCopyInOneHugeBuffer[i * inEnumerator.size()];
        QOut[i] = &QOutCopyInOneHugeBuffer[i * outEnumerator.size()];
    }

    omp_target_memcpy(QInCopyInOneHugeBuffer, tmpQIn,
        numberOfCells * inEnumerator.size() * sizeof(double), 0, 0,
        targetDevice, hostDevice);
    delete[] tmpQIn;
}


void exahype2::fv::rusanov::omp::GPUCellData::copyToHost(
    CellData& hostCellData,
    const enumerator::AoSLexicographicEnumerator& outEnumerator,
    bool copyMaxEigenvalue
) const
{
    double* tmpQOut = new double[numberOfCells * outEnumerator.size()];

    int hostDevice = omp_get_initial_device();

    omp_target_memcpy(tmpQOut, QOutCopyInOneHugeBuffer,
        numberOfCells * outEnumerator.size() * sizeof(double), 0, 0,
        hostDevice, targetDevice);

    for (int i = 0; i < numberOfCells; ++i) {
        std::memcpy(hostCellData.QOut[i],
            &tmpQOut[i * outEnumerator.size()],
            outEnumerator.size() * sizeof(double));
    }
    delete[] tmpQOut;
}
```

Listing A.1: Full implementation of the wrapper class GPUCellData

List of Figures

List of Tables

Bibliography

[1] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung, C. Bertolli, and K. O'Brien. "Offloading Support for OpenMP in Clang and LLVM." In: *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 2016, pp. 1–11. DOI: 10.1109/LLVM-HPC.2016.006.

[2] C. Bertolli, S. F. Antao, G.-T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, et al. "Integrating GPU support for OpenMP offloading directives into Clang." In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–11.

[3] J. Calhoun, F. Cappello, L. N. Olson, M. Snir, and W. D. Gropp. "Exploring the feasibility of lossy compression for pde simulations." In: *The International Journal of High Performance Computing Applications* 33.2 (2019), pp. 397–410.

[4] A. Chikin, T. Gobran, and J. N. Amaral. "OpenMP code offloading: splitting GPU kernels, pipelining communication and computation, and selecting better grid geometries." In: *Accelerator Programming Using Directives: 5th International Workshop, WACCPD 2018, Dallas, TX, USA, November 11-17, 2018, Proceedings 5*. Springer. 2019, pp. 51–74.

[5] N. Ding and S. Williams. *An instruction roofline model for gpus*. IEEE, 2019.

[6] E. Dormy and A. Tarantola. "Numerical simulation of elastic wave propagation using a finite volume method." In: *Journal of Geophysical Research: Solid Earth* 100.B2 (1995), pp. 2123–2133.

[7] A. Dubey, M. Berzins, C. Burstedde, M. L. Norman, D. Unat, and M. Wahib. "Structured adaptive mesh refinement adaptations to retain performance portability with increasing heterogeneity." In: *Computing in Science & Engineering* 23.5 (2021), pp. 62–66.

[8] T. Grützmacher, H. Anzt, and E. S. Quintana-Ortı. "Using Ginkgo's memory accessor for improving the accuracy of memory-bound low precision BLAS." In: *Software: Practice and Experience* 53.1 (2023), pp. 81–98.

[9] J. Huber, M. Cornelius, G. Georgakoudis, S. Tian, J. M. M. Diaz, K. Dinel, B. Chapman, and J. Doerfert. "Efficient execution of OpenMP on GPUs." In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2022, pp. 41–52.

[10] P. D. Lax. *Hyperbolic partial differential equations*. Vol. 14. American Mathematical Soc., 2006.

[11] R. J. LeVeque. *Finite volume methods for hyperbolic problems*. Vol. 31. Cambridge university press, 2002.

[12] P. Lindstrom and M. Isenburg. "Fast and efficient compression of floating-point data." In: *IEEE transactions on visualization and computer graphics* 12.5 (2006), pp. 1245–1250.

[13] A. Mishra, L. Li, M. Kong, H. Finkel, and B. Chapman. "Benchmarking and evaluating unified memory for OpenMP GPU offloading." In: *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. 2017, pp. 1–10.

[14] S. Mohammadian, A. M. Moghaddam, and A. Sahaf. "On the performance of HLL, HLLC, and Rusanov solvers for hyperbolic traffic models." In: *Computers & Fluids* 231 (2021), p. 105161.

[15] P. K. Radtke, C. G. Barrera-Hinojosa, M. Ivkovic, and T. Weinzierl. "An extension of C++ with memory-centric specifications for HPC to reduce memory footprints and streamline MPI development." In: *arXiv preprint arXiv:2406.06095* (2024).

[16] P. K. Radtke and T. Weinzierl. "Compiler support for semi-manual AoS-to-SoA conversions with data views." In: *arXiv preprint arXiv:2405.12507* (2024).

[17] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Köppel, et al. "ExaHyPE: An engine for parallel dynamically adaptive simulations of wave problems." In: *Computer Physics Communications* 254 (2020), p. 107251.

[18] D. Tao, S. Di, Z. Chen, and F. Cappello. "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization." In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 1129–1139.

[19] S. Tian, J. Chesterfield, J. Doerfert, and B. Chapman. "Experience report: writing a portable GPU runtime with OpenMP 5.1." In: *OpenMP: Enabling Massive Node-Level Parallelism: 17th International Workshop on OpenMP, IWOMP 2021, Bristol, UK, September 14–16, 2021, Proceedings 17*. Springer. 2021, pp. 159–169.

[20] T. Weinzierl. “The Peano software—parallel, automaton-based, dynamically adaptive grid traversals.” In: *ACM Transactions on Mathematical Software (TOMS)* 45.2 (2019), pp. 1–41.

[21] 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: *International Conference on High Performance Computing*. Springer. 2023, pp. 65–85.

[22] O. Zanotti, F. Fambri, M. Dumbser, and A. Hidalgo. “Space–time adaptive ADER discontinuous Galerkin finite element schemes with a posteriori sub-cell finite volume limiting.” In: *Computers & Fluids* 118 (2015), pp. 204–224.