Chair of Aerodynamics and Fluid Mechanics
TUM School of Computation, Information and Technology
Technical University of Munich

Migration of a CPU parallelized multi-resolution finite-volume code framework into a hardware-agnostic performance portable application using Kokkos

Scientific work for obtaining the academic degree

Master of Science (M.Sc.) Computational Science and Engineering

at the TUM School of Computation, Information and Technology of the Technical University of Munich

Supervisor PD Dr.-Ing. habil. Stefan Adami
Chair of Aerodynamics and Fluid Mechanics

Advisor Alexander Bussmann, M.Sc.
Chair of Aerodynamics and Fluid Mechanics

Fabian Fritz, M.Sc.
Chair of Aerodynamics and Fluid Mechanics

Submitted by Gaurav Gokhale

Submitted on June 25, 2024 in Garching

Acknowledgements

I would like to thank the Chair of Aerodynamics and Fluid Mechanics and PD Dr.-Ing. habil. Stefan Adami for the opportunity to pursue this thesis and for providing essential resources, including access to the clusters that allowed the implementation of the thesis.

I am also extremely grateful to my supervisors, Alexander Bussmann and Fabian Fritz, for their timely guidance and support throughout my thesis journey. Their expert insights and advice were crucial and invaluable, significantly contributing to the successful completion of my work.

Finally, I would like to thank my father, Santosh Gokhale and my mother, Shweta Gokhale, for their unwavering support and encouragement throughout my academic journey.

Abstract

This thesis aims to integrate the performance portability library Kokkos, into ALPACA, a finite-volume, multiresolution, multi-phase CFD code framework. The primary motivation behind this work is to make ALPACA hardware-agnostic and to enable its execution on different architectures. The thesis discusses the implementation aspects in detail, including the data structure and algorithmic changes and the issues faced during the porting process. The implementation is tested against the original ALPACA code to ensure algorithmic correctness. The performance of CUDA, OpenMP and the serial backend is analyzed in detail. A significant performance improvement is observed, especially in the case of the CUDA backend. The performance of the OpenMP backend is comparable to the original ALPACA code with MPI implementation. The serial backend is observed to be slower than the original ALPACA code. Also, various experimental optimizations are explored to identify possible avenues of parallelism. The thesis concludes with a discussion of future work and the potential improvements that can be made to the current implementation. Overall, the thesis lays the building block for a performance portable version of ALPACA using Kokkos.

Contents

List of Figures

List of Tables

Acronyms

ALPACA Adaptive Levelset Parallel Code Alpaca

AoSoA Array of Structure of Arrays

AMR Adaptive Mesh Refinement

CFD Computational Fluid Dynamics

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

CRTP Curiously Recurring Template Pattern

DNS Direct Numerical Simulation

EOS Equation of State

FLOPS Floating Point Operations Per Second

FPGA Field-Programmable Gate Array

GPU Graphics Processing Unit

HLLC Harten-Lax-van Leer-Contact

HIP Heterogeneous-Compute Interface for Portability

IC Internal Cells

LES Large Eddy Simulation

MHD Magnetohydrodynamics

MPI Message Passing Interface

MR multiresolution

NSE Navier-Stokes Equations

NUMA Non-Uniform Memory Access

PCI Peripheral Component Interconnect

PDE Partial Differential Equation

RANS Reynolds-Averaged Navier-Stokes

RTI Rayleigh-Taylor Instability

SIMD Single Instruction, Multiple Data

SM Streaming Multiprocessor

UVM Unified Virtual Memory

Chapter 1

Introduction

1.1 Motivation

Computational Fluid Dynamics (CFD) has become a pivotal tool in various domains as the demand for precise and efficient fluid simulations continues to rise across various engineering sectors. With the world moving towards exascale computing, the future of CFD simulations will inevitably be driven by the need for faster and more efficient algorithms. Developing algorithms that can leverage the parallelism offered by modern heterogeneous architectures is essential. The term *heterogenous* here refers to the different types of processing units available within a single system, such as CPU (Intel Xeon, AMD EPYC), GPUs (NVIDIA, AMD), FPGAs and other accelerators. Due to difference in architectures, the same code may not perform optimally across all of them. GPUs require different data structures to ensure coalesced memory access, increase the FLOPS per memory access, and utilize GPUs' high bandwidth. In most cases, this results in different code bases, one solely CPU-based and the other using one of the native GPU programming models, such as CUDA for NVIDIA GPUs or HIP for AMD GPUs. This approach is cumbersome, error-prone, and ineffective regarding development time and resources.

The traditional approach to parallelism has been to use Message Passing Interface (MPI) to distribute the workload across NUMA nodes and use OpenMP within a NUMA node. However, recently performance portable frameworks are gaining popularity that provide consistent performance levels with minimal architecture-dependent code. Performance portability is an active field of research in various domains,[35][29][4]. This thesis explores the integration of one such performance portable programming model, *Kokkos*[38][8], in the CFD code Adaptive Levelset Parallel Code Alpaca (ALPACA)[17]. ALPACA is a multiresolution, multiphase, compressible flow solver written in C++20. The idea is to keep MPI for inter-node parallelism and use *Kokkos* to achieve intra-node parallelism.

1.2 Related work

Several efforts have been made to integrate performance portable frameworks into applications in recent years. Some investigate the use of these models in small-scale applications, also called as mini-apps[16][9] or much larger code bases involving complex CFD solvers [19] or molecular dynamics simulation frameworks[4].

K-athena[12] is one example where the authors detail the porting process of integrating Kokkos into Athena++[33]. Athena++ is a Magnetohydrodynamics (MHD) code initially written for the CPU. Upon integrating Kokkos into Athena++, the authors achieved significant performance improvements for various architectures. It uses entities called meshblocks, which are distributed across MPI ranks. These mesh blocks are updated independently, given the boundary information. The authors have provided scaling studies for Intel Skylake CPUs, Intel Xeon Phis, and NVIDIA GPUs. The studies demonstrate a speedup of 30 using 24,576 GPUs compared with 172,032 CPUs on the Summit supercomputer at Oak

Ridge National Laboratory.

Cabana[31] stands as another example of a performance-portable particle-based simulation library. Leveraging Kokkos for on-node parallelism, the library employs cuda-aware-MPI for multi-node communication, taking advantage of high-speed interconnects such as Infiniband or Omnipath. Support extends to all Kokkos backends, including OpenMP, CUDA (NVIDIA GPUs), HIP (AMD GPUs), and SYCL (Intel GPUs). Moreover, Cabana enhances the `Kokkos::View` data structure to furnish custom data structures tailored for particles and particle algorithms. Exascale design patterns like the `Kokkos::ScatterView` are utilized, effectively managing race conditions when programming for GPU applications. A GPU-resident strategy, where the data is directly initialized on the GPU, is employed, in contrast to the GPU offload approach, aiming to minimize intermittent copies to the host as much as possible. Flexibility in deciding ideal data layouts and memory access patterns is achieved through AoSoA data structures. Templates on Memory Space and Execution space are leveraged, offering users the choice of any compatible combination. This flexibility allows for using different threading backends on a device, such as CUDA and OpenMP-target on NVIDIA-GPUs. It uses kernel fusions to mitigate the kernel launch overhead and avoid duplicate data computation. Additionally, it notes that cache performance and global data reuse can be improved by adopting this approach in some instances.

Sparc[19], developed at Sandia National Laboratories, represents a significant advancement in the realm performance portable transonic and hypersonic simulations. This versatile tool is designed to support comprehensive simulations encompassing aerodynamic, aerothermal, and aerostructure aspects. One of Sparc's notable features is its compatibility with both unstructured and structured meshes, enhancing its flexibility in handling various simulation scenarios. To address the challenge of race conditions, which are a common issue in on-node programming models, Sparc employs a 3D odd-even graph coloring technique. This approach not only mitigates race conditions but also contributes to the robustness of the simulations. The architecture of Sparc is characterized by a modular structure, which is organized around separate objects for different algorithmic aspects, such as Problem, TimeSolver, and Linear/Nonlinear Solver, among others. This modular design facilitates ease of use and adaptability to specific simulation requirements. For linear solvers, Sparc offers the capability to interface with Trilinos, further extending its functionality and applicability. Another key feature of Sparc is its implementation of kernel composition and static dispatch, which are instrumental in achieving efficient GPU implementation. This is particularly relevant when considering the increasing reliance on GPUs for high-performance computing tasks. Furthermore, Sparc leverages Kokkos for on-node parallelism and MPI for inter-node parallelism, thereby ensuring scalable performance across different computing architectures. This adaptability is evidenced by Sparc's reasonable performance on a variety of architectures, including Intel Xeon Phi and NVIDIA Tesla GPUs. Despite its current capabilities, Sparc is continuously being optimized to enhance its performance portability. The use of Kokkos is highlighted as a promising framework in this regard, although ongoing work aims to further refine Sparc's performance across different architectures.

Handling a large number of nodes or blocks in Adaptive Mesh Refinement (AMR) poses significant challenges when executed on accelerated nodes. One of the main difficulties is the management of the mesh hierarchy, which contributes to significant overhead. Additionally, the accumulated latency from launching kernels becomes notable with the increase in the number of blocks. Parthenon[11], which is built on top of Athena++ and K-athena, addresses these issues by offering high-level abstractions specifically designed for block-based AMR codes. By exploiting on-node parallelism through Kokkos, Parthenon ensures efficient execution. Its design follows a device-first/device resident strategy, which streamlines the packaging of various data structures directly in the device memory, thereby optimizing performance. For managing inter-node parallelism, Parthenon utilizes asynchronous MPI communication, which is configured with GPU-aware MPI libraries. This approach facilitates efficient data exchange between nodes. Moreover, Parthenon introduces an intermediate abstraction layer on top of Kokkos. This layer simplifies the process of setting parameters for different hardware, thereby reducing the complexity for the end-user. As a result, downstream applications can seamlessly integrate

Parthenon as a "plug-and-play" library for their AMR codes, leveraging its capabilities to enhance performance. One of the key features of Parthenon is its ability to mitigate the latency associated with kernel launches. This is achieved by aggregating multiple blocks into a single kernel launch. Through the use of `VariablePacks` and `MeshBlockPacks`, Parthenon allows for efficient grouping of blocks. Furthermore, users have the flexibility to determine the optimal number of mesh blocks to pack together at runtime, enhancing performance. Parthenon also incorporates effective load balancing strategies that facilitate the rebuilding of the tree hierarchy and redistribution of blocks among devices following mesh refinement. To minimize the overhead post-refinement, it employs a strategy that coarsens the mesh periodically, based on a runtime parameter. A study highlighted in the literature provides in-depth insights into the performance improvements in AMR codes, especially in terms of reducing communication and kernel launch overheads, which are commonly identified as bottlenecks. Notably, Parthenon achieves a weak scaling efficiency of approximately 92% till 73728 GPUs. Its compatibility with Kokkos enables Parthenon to support a wide range of architectures, including AMD and NVIDIA GPUs, Intel and AMD x86 CPUs, IBM Power9, and Fujitsu A64FX CPUs, thereby broadening its applicability across various computing platforms.

AMReX[39] is another block-structured AMR framework that offers a range of temporal and spatial discretization schemes suitable for various domains such as additive manufacturing, astrophysics, combustion, and wind plant modelling. It provides essential functionalities for AMR codes, including time stepping, interpolation, re-meshing, load balancing, data containers and iterators. This flexibility allows users to select the package that best meets their needs, offering a high level of abstraction and complete control over the algorithms. AMReX also features a lightweight abstraction layer designed to obscure architecture-specific details, enabling users to maximize hardware potential by simply specifying the operations performed on a data block while AMReX handles the mapping to the underlying hardware. It employs hash-based algorithms to communicate ghost cells or inter-level exchange and supports GPU-aware MPI on GPU machines. AMReX adopts a strategy of packing multidimensional data into single-dimensional arrays and utilizes a kernel-fusing mechanism for communication. For CPU-based applications, OpenMP is used for on-node parallelism, whereas CUDA, HIP, or DPC++ (Intel) are employed for GPU-based applications. The framework's approach to parallel operations is similar to Kokkos/RAJA[5] but is specifically tailored to meet the needs of AMR codes. The authors also detail the implementation of their abstraction layer and its application in this work.

The thesis starts with a brief background about ALPACA and the governing equations that it solves. It then explains the fundamental differences between CPU and GPU architectures and the available programming models. A brief comparison of these programming models is also provided to justify the choice of Kokkos. The thesis then moves to give a brief overview of the philosophy of Kokkos, and its relevant capabilities are discussed. Then, in Chapter 3, the implementation aspects of porting the code to GPU are discussed, specifically the integration of Kokkos in the build process of ALPACA, modifications to the data structures to incorporate Kokkos multidimensional arrays or Views, and the porting process. Chapter 4 provides a detailed comparison of the CUDA backend, OpenMP backend and serial backend with CPU-only ALPACA version. It also provides profiling results and identifies possible bottlenecks in the current implementation. Finally, Chapter 5 summarises the results and provides an outlook for future work.

Chapter 2

Background

This chapter delves into the theoretical underpinnings of ALPACA, offering a concise overview of the governing equations that underlie the framework. It explores certain facets of block-based Multiresolution, shedding light on its significance and application within the context of ALPACA. Additionally, the chapter touches upon the fundamental distinctions between CPU and GPU architectures, providing a brief comparison that highlights their unique characteristics and functionalities. Furthermore, various hardware-agnostic programming models are analysed, focusing on understanding the rationale behind the choice of Kokkos. This comparison serves to elucidate the advantages of Kokkos and position it within the broader landscape of programming models that cater to diverse hardware configurations.

2.1 ALPACA

ALPACA is a simulation framework designed for multiphase compressible flows. At its core, it employs a finite volume method, utilizing approximate Riemann solvers, ensuring accurate representations of flow dynamics. ALPACA incorporates an adaptive block-based multiresolution scheme, complemented by local time stepping functionalities, effectively managing computational loads while maintaining precision. ALPACA uses a Direct Numerical Simulation (DNS) approach contrary to the Large Eddy Simulation (LES) or Reynolds-Averaged Navier-Stokes (RANS), where additional models are used with governing equations to capture the physical effects. While more accurate, the DNS approach requires finer discretization, requiring more computing power. It uses MPI for parallelization for distributed as well as shared memory architectures. ALPACA's modular structure offers flexibility, enabling users to customize components at compile-time. Additionally, the application is built using C++20 and adheres to object-oriented programming principles, ensuring a robust and efficient design.

2.1.1 Governing Equations

ALPACA essentially solves the compressible Navier-Stokes Equations (NSE) which are given by:

$$\frac{\partial \rho}{\partial t} = -\mathrm{div}\rho v, \tag{2.1a}$$

$$\frac{\partial \rho v}{\partial t} = -\mathrm{div}(\rho v \otimes v + \Pi) + \rho g, \tag{2.1b}$$

$$\frac{\partial \rho E}{\partial t} = -\mathrm{div}(\rho E v + \Pi v - q) + \rho g \cdot v, \tag{2.1c}$$

where, $\rho(x,t)$ is density, $v(x,t)$ is the velocity vector, $g(x,t)$ is gravitational acceleration, $E = e + v^2/2$ is the total energy where e is specific internal energy, $q = -k\nabla T$ are the heat fluxes where k is thermal conductivity and T is temperature.

These equations (2.1) describe the flow of a fluid in d-dimensional space where $d = 1, 2, 3$ and time. Equation (2.1a) is the continuity equation, (2.1b) is the momentum conservation equation, and (2.1c) is the energy conservation equation. The above equations (2.1) are expressed in differential form. Solving these equations numerically in scenarios where shockwaves or discontinuities occur might render the numerical methods unstable. Hence, using the integral form of these governing equations is more desirable. They are given as:

$$\int_V \frac{\partial \rho}{\partial t} dV = -\oint_S (\rho v) \cdot n dS, \tag{2.2a}$$

$$\int_V \frac{\partial \rho v}{\partial t} dV = -\oint_S (\rho v \otimes v + p\mathbf{I} + \tau) \cdot n dS + \int_V \rho g dV, \tag{2.2b}$$

$$\int_V \frac{\partial \rho E}{\partial t} dV = -\oint_S (\rho E v + p\mathbf{I} v + \tau v + q) \cdot n dS + \int_V \rho g \cdot v dV, \tag{2.2c}$$

where, $p(x,t)$ is pressure, $\mathbf{I}$ is the Identity matrix, $\tau(x,t)$ is the viscous stress tensor, V is the material volume, and S is the area of the volume. Rearranging the terms in (2.2) gives the following flux based formulation:

$$\int_V \frac{\partial U}{\partial t} dV = -\oint_S (F^c + F^\mu + F^q) dS + \int_V f dV. \tag{2.3}$$

Here, U is the state vector, F^c is the convective term contribution, F^μ is the viscous term contribution, F^q are the heat flux densities, f is the volume-force vector including gravity and other forces. The thermodynamic closure of NSE is attained by an Equation of State (EOS) which is given by:

$$p = f(\rho, e). \tag{2.4}$$

Equation (2.4) gives the relation between pressure, density and energy. The above equations, (2.3) and (2.4), form the basis of the simulation framework for ALPACA for single-phase simulations, which are then solved numerically. While ALPACA can solve multiphase flows, only single phase part is within the scope of this thesis. Interested readers can refer to [17] for details about multiphase flows.

2.1.2 Riemann solver

ALPACA uses Finite Volume Method with explicit appoximate Riemann solver[36] between neighbor cells. Various Riemann solvers are available in ALPACA [18]. However, for this thesis, only two of the Riemann solvers are implemented with Kokkos, namely Roe[30] and HLLC[37]. These two solvers require left and right eigenvectors for every cell to transform the original system of equations to characteristic space. In addition to this, the Roe solver requires eigenvalues and the scalar advection velocity in each cell.

There exist many reconstruction stencils for use with the Riemann solver in ALPACA. All the available reconstruction stencils can be used within the current thesis implementation.

2.1.3 Equation of State

As seen earlier in equation (2.4), the closure of NSE is obtained with EOS. ALPACA offers various types of EOS [18], but it is restricted to only Stiffened EOS for this thesis which is given in equation (2.5)

$$p(\rho, e) = (\gamma - 1)\rho e - \gamma B, \tag{2.5}$$

where, γ is the ratio of specific heat capacities, B is the background pressure constant which is set to zero

2.1.4 Block Based Multiresolution

Various problems arise when solving for compressible flows, which involve capturing complex phenomena like discontinuities and sharp gradients. Such problems require high resolution to accurately capture these effects, which can be prohibitively expensive if applied uniformly over the entire domain. This is where multiresolution (MR) comes into the picture, which helps achieve finer resolution only in the areas of interest whilst keeping coarser resolution for the rest of the domain as can be seen in figure 2.1. ALPACA uses block-based multiresolution (MR) to refine entire blocks instead of single cells. Each block has an exact number of internal cells, which can be specified in ALPACA within user specifications as a compile-time constant. Each *parent* block is superimposed by refined *child* blocks. A *leaf* is the block with the finest cells at that location, and the PDE is solved only for the leaves in the domain. The decision for refinement is made on a per-block basis and not for individual cells. If a parent block is marked for refinement, it spawns two child blocks in 1D, four child blocks in 2D, and eight child blocks in 3D. This leads to a binary/quad/octree-based implementation of the algorithm.

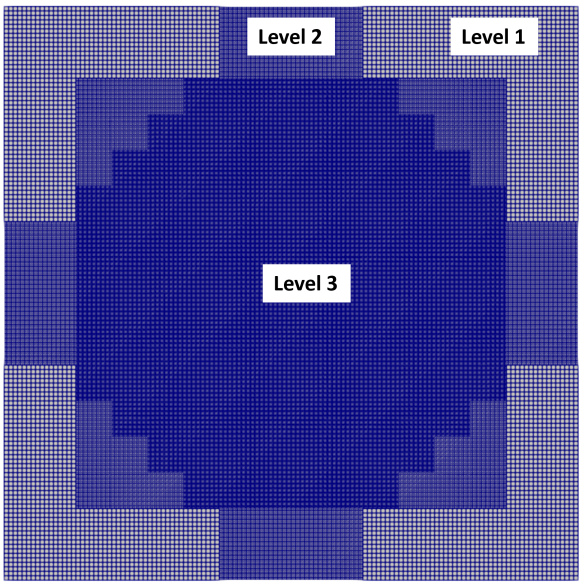


Figure 2.1: Toro Implosion case mesh with 3 levels of refinement. Level 1 has the coarsest and level 3 is the finest resolution.

Each block is covered by a layer of four halo/ghost cells in each direction, which reduces the need for data exchange. The blocks can be updated more independently using the halo cells. These halo cells must be filled before time integration, either within or from another rank, depending on where the neighbouring block lies. If there are any fine/coarse or coarse/fine interfaces between neighbouring blocks, an averaging or prediction operation is performed to determine the values within halo cells. ALPACA uses linear interpolation for averaging from child (fine) to parent (coarse) blocks and fifth order interpolation based on [14] for prediction from parent (coarse) to child (fine) blocks. The advantage of using block-based multiresolution is that it gives uniform access patterns that are cache friendly and can be highly optimized for Single Instruction, Multiple Data (SIMD) execution.

2.2 CPUs Vs GPUs

2.2.1 CPU vs GPU comparison

The architectures fundamentally differ when comparing a Central Processing Unit (CPU) and a Graphics Processing Unit (GPU). CPU serves as the primary computational unit, responsible for executing various computing tasks needed for operating systems and applications to function. A typical CPU is designed to prioritize speedy task completion with minimal latency while maintaining the capability to swiftly transition between different operations. On the other hand, a GPU is a specialized hardware component optimized for efficiently processing complex mathematical operations in parallel, surpassing the capabilities of a general CPU in these aspects. Although initially designed for graphics rendering in gaming and animation, GPUs now find widespread applications beyond their initial scope.

Modern superscalar CPU architectures come with a much larger instruction set with complex instructions. This enables them to handle a wide variety of tasks but may decrease the peak throughput. GPUs, on the other hand, have simpler architectures and a smaller instruction set. They leverage the massive number of cores available to execute these instructions on multiple data (SIMD). For better understanding of the two architectures, consider the comparisons of GPUs and CPUs available on two different clusters at the chair of aerodynamics and fluid mechanics at TUM.

(a) Hardware comparison of *kanon* cluster

Feature	**NVIDIA GeForce RTX 2080 Ti[28]**	**Intel(R) Xeon(R) CPU E5-2609 v4mk[21][20]**
SM	68	-
Cores	4352 (CUDA)	8
Max clock speed	1635MHz	1700MHz
RAM	11GB	128GB
Memory bandwidth	616GB/s	59.7GB/s
FP64	-	217.6GFLOPS
FP32	14.2TFLOPS	-
FP16	38.5TFLOPS	-

(b) Hardware comparison of *kushana* cluster

Feature	**NVIDIA RTX A6000[27]**	**AMD EPYC 7702P[3][2]**
SM	84	-
Cores	10752 (CUDA)	64
Max clock speed	1800MHz	3350MHz
RAM	48GB	220GB
Memory bandwidth	768GB/s	204.8GB/s
FP64	-	2.048TFLOPS
FP32	38.7TFLOPS	-
FP16	38.7TFLOPS	-

Table 2.1: Architecture Comparison CPU vs GPU of *kushana* and *kanon* clusters in the chair of fluid mechanics and aerodynamics at TUM. *Kanon* cluster has 7 NVIDIA RTX 2080Ti GPUs with Intel Xeon E52609 CPU and *Kushana* has 4 NVIDIA RTX A6000 GPUs with AMD EPYC 7702P CPU.

It is important to understand the GPU terminology here. A GPU consists of many cores or *Streaming Multiprocessor (SM)* as NVIDIA calls them, or *computing units* as called by AMD. Each core has certain number of threads which are called as a block. Each thread block gets assigned a warp

which is then mapped to the streaming multiprocessor. These SMs then execute an instruction on multiple data in a lockstep.

As can be seen from the table 2.1, the CPU has a higher clock speed than the GPU for both the clusters. But the sheer number of cores that the GPU has combined with the higher memory bandwidth means that the GPU has higher peak floating point performance as compared to CPU. It is important to note that GPUs on both the clusters do not have a native double precision processing units and emulates it using software. Even adjusting for it, the performance of the CPU would be approximately 450GFLOPS for FP32 on kanon and 4.096TFLOPS for FP32 on Kushana, which is still lower than the GPU counterparts as seen in table 2.1a and table 2.1b. However, the GPU has much less memory of 11GB on kanon and 48GB on kushana than the CPU which has 128GB and 220GB respectively. This is the limiting factor when running memory-intensive applications on GPU.

2.2.2 Application workflow on heterogeneous architectures

Modern compute nodes typically consist of two or more *sockets*, with each socket having multiple CPU cores and one or more GPUs per socket. Though a GPU has different memory and execution space, it is not a self-standing device. The program's control flow still lies with the CPU, which operates with the GPU using the PCI-Express bus. This often entails that the data has to be initialized directly on the GPU or copied from CPU to GPU. The GPUs recently also have started coming with Unified Virtual Memory (UVM), which can be accessed from both the CPU and the GPU. This aids the developers to some extent in that the data migration is taken care of by the UVM depending on the context from where it is accessed. However, using UVM comes with its own sets of issues, like page faults, and it can still be faster to have data allocated in the global memory space of the GPU. A typical GPU-accelerated program follows five fundamental steps:

1. Allocation of memory on GPU
2. Copy of data from CPU memory space to GPU memory space
3. Invoke *kernels* or parallel regions for performing compute intensive tasks on GPU
4. Copy data from GPU memory space to CPU memory space for I/O of the results
5. Free the GPU memory

It is also important to note that the kernel launches are usually done asynchronously unless a synchronization construct is called on the CPU. The CPU can manage other workloads and queue compute-intensive tasks on the GPU. The kernels are completed in the order they are dispatched, ensuring no race condition exists between kernels. If the data processed inside a kernel is supposed to be used by the CPU, then both the *host* and the *device* must be synchronized before any data can be used. However, keeping these synchronization points to a minimum is crucial as they restrict the parallelization and often lead to poor performance.

GPU computing is not an alternative to the CPU. The goal is to identify which tasks will suit which architecture and use both CPUs and GPUs in tandem. If the problem size is small and has a complex control flow with many dependencies, then CPU is a good choice for such applications. However, GPU is more suitable for applications that show massive data parallelism. This is why it is becoming important to have hardware agnostic programming models which can enable the developers to write code that can be executed on both CPU and GPU without much change in the codebase.

2.2.3 GPU framework comparison

While MPI is the de-facto standard for distributed inter-node parallelism, numerous programming models exist for extracting on-node parallelism. Some are pragma or directives based, like OpenMP and

OpenACC[15], which rely on the compiler to generate the code for the target architecture. Then, there are vendor-specific programming models like CUDA for NVIDIA GPUs or HIP for AMD GPUs. Finally, libraries like Thrust[6], Kokkos[8], SYCL[1], RAJA [5], and OpenCL[34] provide a set of abstractions to write code that can be executed on different architectures. There have been numerous attempts to compare these frameworks to provide an understanding of the differences between them and the performance that can be expected from them. In [24], the authors have compared OpenCL, OpenMP, OpenAcc and CUDA for performance, energy consumption and programming productivity. For this study, they considered various applications from the Rodinia [7] and SPEC accel [32] benchmark suites. The authors mention that while the general trend is for CUDA to have some performance benefits over OpenACC and OpenMP, it is primarily application-dependent.

Since ALPACA involves a lot of 3D stencil operations, a benchmark study was performed to get a clearer picture of the performance and programming productivity of these frameworks. The benchmark consisted of two input and output vectors of the same sizes. The vectors were considered to store 3D data, and the stencil operation performed was a simple sum of its front-back, left-right and top-bottom neighbours, including the current index, dividing it by 8.0 and writing it to the output vector. The host version of the stencil operation is shown in the code snippet 2.1 below:

```
void stencil3d(const std::vector<double>& input, std::vector<double>&
                output, int size)
{
    for(int k=1; k<size-1; k++)
    {
        for(int j=1; j<size-1; j++)
        {
            for(int i=1; i<size-1; i++)
            {
                int index = k * size * size + j * size + i;

                output[index] = (input[index] + input[index + 1]
                                + input[index -1] + input[index + size]
                        + input[index - size] + input[index + size*size]
                                + input[index - size*size] ) / 8;
            }
        }
    }
}
```

Listing 2.1: Implementation of stencil operation on 3D data (Host version), depicted with three tightly nested for loops iterating over each index in the input array and writing results to the output array.

The same operation was performed using CUDA, OpenACC, Thrust, SYCL and Kokkos. The entire stencil was repeated 100 times to hide the latency of kernel launch and also the data-copy operations. The sample code for CUDA and Kokkos is shown below for comparison in code snippet 2.2 and 2.3:

```
const int BLOCK_SIZE = 8;

__global__ void stencil3d_kernel(double* input,double* output,int size)
{
 int tid_x = blockIdx.x * blockDim.x + threadIdx.x;
 int tid_y = blockIdx.y * blockDim.y + threadIdx.y;
 int tid_z = blockIdx.z * blockDim.z + threadIdx.z;
 int tid = tid_z*size*size + tid_y*size + tid_x;

 if(tid_x < size-1 && tid_y < size-1 && tid_z < size - 1
    && tid_x > 0 && tid_y > 0 && tid_z > 0)
 {
    output[tid] = (input[tid] + input[tid + 1] + input[tid - 1] +
                   input[tid + size] + input[tid - size] +
                   input[tid + size * size] + input[tid - size * size])
```

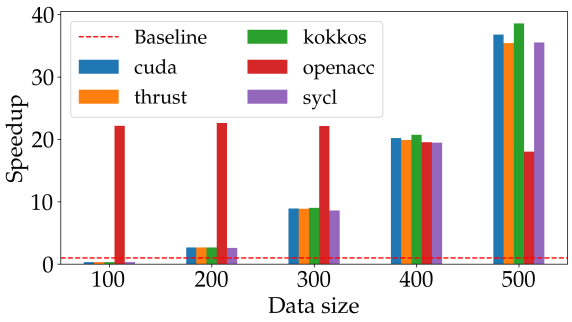
```
                    / 8.0;
}
}
```

Listing 2.2: CUDA version of a stencil operation on 3D data, showcasing a kernel code snippet where each thread, identified by blockIdx.x/y/z, executes the stencil operation.

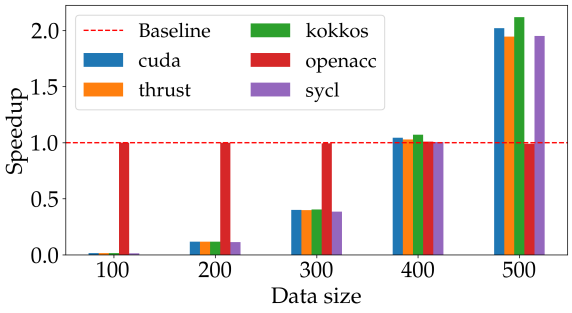
```
typedef Kokkos::View<double***> Mat3D;
typedef Kokkos::MDRangePolicy< Kokkos::Rank<3> > mdrange_policy;

void stencil3d_kokkos(const Mat3D& input, Mat3D& output, int size)
{
 Kokkos::parallel_for( "stencil3d", mdrange_policy({0,0,0},
                                    {size,size,size}),
 KOKKOS_LAMBDA ( const int i , const int j, const int k )
 {
    if(i > 0 && j > 0 && k > 0 &&
       i < size-1 && j < size-1 && k < size-1)
        output(i,j,k) = (input(i,j,k) + input(i+1,j,k) + input(i-1,j,k)
                        + input(i,j+1,k) + input(i,j-1,k)
                        + input(i,j,k-1) + input(i,j,k+1))/8.0;
 });
    Kokkos::fence(); // Ensure all parallel tasks are complete
};
```

Listing 2.3: Kokkos version of a stencil operation on 3D data, featuring the use of parallel_for for parallel dispatch, where indices i, j, k are accessible within a functor or anonymous function defined with KOKKOS_LAMBDA.

The comparison is done on the kushana cluster at the chair of aerodynamics and fluid mechanics at TUM. The details about the hardware are discussed in table 2.1b. Comparison has been made for two optimization levels, O0 and O3 for CPU as can be seen in figure 2.2a and figure 2.2b.



(a) Comparison of code performance at O0 optimization level, which excludes compiler optimizations such as software pipelining, loop unrolling, and predictive commoning.



(b) Comparison of code performance at O3 optimization level, characterized by aggressive optimizations including vectorization, loop unrolling, and unrestricted compiler memory utilization.

Figure 2.2: Comparison of GPU frameworks for stencil operation. The baseline represents normalized CPU performance. Data size indicates the size per dimension of 3D data. Speedup is calculated as the ratio of CPU time to GPU time for each framework and data size, averaged over five runs.

Observation 1

The performance of Kokkos, SYCL and Thrust is comparable with CUDA for all the cases. The maximum speedup obtained for both O0 and O3 optimization levels is seen for the largest data size, indicating that GPU is more efficient for larger data sizes.

Observation 2

Even though appropriate compiler flags were used for OpenACC, it was found to be slower than the rest of the frameworks. This is due to the fact that OpenACC is dependent on the compiler to generate the code for the target architecture. The compiler might not be able to optimize the code as well as the other frameworks. In fact the maximum speedup obtained using OpenACC was just as high the O3 CPU optimization, indicating that the GPU was not getting utilized.

Observation 3

The programming producitivity of OpenACC was found to be the highest considering just the code changes required to port the code to GPU. Since it is directive based, minimal changes were required to the code. But the compiler flags were difficult to figure out and the default nvcc compiler was not able to port the code to GPU

Observation 4

Thrust provided good interface for dynamic data structures like host/device vectors and also resembled STL library to a large extent. However, it mainly supports CUDA and OpenMP backends and lacks support for Intel and AMD GPUs, thus not giving performance portability.

Obervation 5

SYCL was found to be a bit more involved, with its syntax of queue and queue handler, but it follows the same principles as Thrust and Kokkos. It gives more control over the access traits of the data (read/write). Inspite of having performance portability going in its favor, it did have some issues when coming to the integration aspect in existing codebase. SYCL or AdaptiveCPP as it is called now, creates a new compiler *acpp* on installing sycl, and the user must use this compiler for compiling their code. While this was clear for such a small example, the documentation is lacking for how to do it for larger codebases. And in the event of inavailability of this compiler on supercomputing clusters, it creates unnecessary friction to compile the code with ease. Moreover, it wasn't as performant as Kokkos as can be seen from the graph in figure 2.2.

Observation 6

Kokkos was the most performant version for this benchmark. It also follows similar philosophy as SYCL, but it was found that it is more widely used and has better documentation. It was found that it is easier to integrate this into existing codebase since it is based on *CMake*.

It is important to note that the above observations are based on a simple stencil operation, and the performance might vary for more complex operations. An expert in the above programming models can optimize the code for better performance. However, given a timeframe and a simple operation, this exercise aimed to assess which framework was the most effective in terms of performance and programming productivity. Considering the above observations, Kokkos was used for this thesis.

2.3 Kokkos overview

This section briefly overviews some features of Kokkos, namely the *View* data structure, parallel dispatch, profiling, Kokkos with MPI and Kokkos build procedure.

2.3.1 Kokkos::View

`Kokkos::View` is a data structure in the Kokkos parallel programming library designed for multi-dimensional array representation and manipulation. It can have eight runtime and compile time dimensions [38]. It provides a flexible interface for managing and accessing data efficiently across different execution spaces, such as CPUs, GPUs, and accelerators. The user can specify different data Layouts, such as LayoutRight (row-major) or LayoutLeft (column-major), whichever suits the existing code base. This is important to maintain interoperability when incrementally porting a code to GPU.

```
Kokkos::View<double*[17][17][17],Kokkos::LayoutRight> demoView("demo",4);
```

Listing 2.4: `Kokkos::View` with double data type featuring three compile-time dimensions (17) and one runtime dimension (4), utilizing LayoutRight (row-major) storage.

As seen in 2.4, the defined View has four dimensions, three of which are compile-time, and one dimension is runtime. The runtime dimension is specified as four in the initialization of the View. Each View also consists of a name or hook used to identify the View in profiling output or error messages. The View is initialized in the default execution space on the device (GPU) in case Kokkos is built with any GPU backends or on the host (CPU) with OpenMP or serial backends.

For the usual workflow of an application of reading the inputs, initialization on the host, and transfer of data to the device for computation, Kokkos provides a *mirror View*. It has the same traits as the original View but is allocated on the host if the default execution space is a device. This provides an easy way to transfer data between host and device by using `deep_copy` function as shown in 2.5.

```
    // Create a mirror view of demoView
    auto demoMirrorView = Kokkos::create_mirror_view(demoView);

    // Initialize the demoMirrorView on host......

    //copy the data from device to host
    Kokkos::deep_copy(demoMirrorView, demoView);
```

Listing 2.5: Utilizing `Kokkos::MirrorView` for seamless copying between device and host memory spaces.

Another helpful feature is the `Kokkos::subview`, which helps to acquire slices of the original View as shown in 2.6. This is particularly useful when the user wants to perform operations on a subset of data contained in a `Kokkos::View`. The subview has the semantics of a `std::shared_ptr`, and its changes are automatically reflected in the original View.

```
    // Create a mirror view of demoView
    auto demosubview = Kokkos::subview(demoView,
                                       std::make_pair(0,2),
                                       Kokkos::ALL,
                                       Kokkos::ALL,
                                       Kokkos::ALL);

    // Do some operations on demosubview......
```

Listing 2.6: Employing `Kokkos::subview` to extract data slices from a `Kokkos::View`.

2.3.2 Parallel dispatch

Kokkos provides three main parallel dispatch strategies: `parallel_for`, `parallel_reduce` and `parallel_scan`. For this thesis, only `parallel_for` and `parallel_reduce` are relevant and are discussed in this section.

`Parallel_for` is similar to the OpenMP construct `#pragma omp parallel_for`. Parallel_for distributes the index range across available hardware resources and executes the loop body concurrently. The iteration range can be a single dimension (`RangePolicy`) or multi-dimensional (`MDRangePolicy`). The `MDRangePolicy` is particularly useful when we have tightly nested for loops.

The `parallel_for` takes in a functor or lambda as its argument. This functor/lambda should provide the `operator()` method, which takes the thread indices as the input argument, and this operator then provides the work to be done by each thread.

`Parallel_reduce` is used to perform reduction operations like taking sum, maximum or minimum across threads, for instance, when calculating timestep. It follows a similar workflow as `parallel_for` except that the `operator()` now takes an additional argument, which is the reduction variable by reference as can be seen in code snippter 2.7. Each thread then uses this to perform the subsequent reduction operation.

```
1
2 //Create an MDRangePolicy for 3 dimensions
3 Kokkos::MDRangePolicy<Kokkos::Rank<3>> mdrange_policy(
4     {0, 0, 0},
5     {sizeX, sizeY, sizeZ}
6 );
7
8 // Call parallel_reduce over mdrange_policy to reduce local_result into
9 // result
10
11 Kokkos::parallel_reduce("mdrange_reduce_kernel", mdrange_policy,
12         KOKKOS_LAMBDA(int i, int j, int k, int& local_result) {
13             local_result += i + j + k;
14         }, result);
```

Listing 2.7: Using `Kokkos::parallel_reduce` for scalar results, the functor or lambda provides threads with an additional argument to update the local result value.

Kokkos also provides a `TeamPolicy`, which is helpful for hierarchical parallelism. It is beneficial when the loops are not tightly nested, and certain groups of threads access the same memory repeatedly. The user can create a so-called *scratch_space*, a shared memory space for each team of threads. The shared memory has a much higher bandwidth than the global memory and can considerably speed up the kernels if used properly. Another use of `TeamPolicy` is the ability to launch `parallel_for` from within a `parallel_for` region.

2.3.3 Profiling

Kokkos also comes with profiling tools like kernel timings, memory consumption, and space time stack in the form of *KokkosTools*[22]. The user can enable profiling by simply setting the environment variable `KOKKOS_PROFILE_LIBRARY` to the path of the tool, for instance *path_to_simple_kernel_timer.so*. The user can then simply run the application and Kokkos will dynamically load the profiling tool library. It generates the profiling report as a .dat file which can be read using *kp_reader* which is another kokkos tool. It provides simple output using the names/hooks which we have provided already in the code when initializing a View or a parallel dispatch operation. We can also explicitly set the profiling regions using `Kokkos::Profiling::pushRegion` and `Kokkos::Profiling::popRegion`.

2.3.4 Kokkos and MPI

For using MPI with Kokkos, especially when using Kokkos GPU backends, it is essential to have the MPI libraries correctly linked and configured for GPU-aware communication. This configuration allows for the direct communication of data buffers between GPUs, eliminating the need for intermediate copying to host memory. Such an approach leverages the high bandwidth offered by NVLink or Infiniband interconnects between GPUs, significantly enhancing data transfer efficiency. Moreover, `Kokkos::View` is a contiguous data structure and can be seamlessly integrated with MPI calls with `data()` method, facilitating straightforward data management and communication as can be seen in listing 2.8.

```
1
2 //Kokkos View of 3 dimensions, this is allocated on
3 //by default for device backend
4 Kokkos::View<double[X][Y][Z]> send_data("data");
5 Kokkos::View<double[X][Y][Z]> recv_data("data");
6
7 //Initialize the data/some operations on the data
8
9 if(send_rank)
10     //Send the GPU buffers directly
11     MPI_Send(send_data.data(), X*Y*Z, MPI_DOUBLE, destination_rank,
12              tag, MPI_COMM_WORLD);
13
14 if(recv_rank)
15     //Recv the GPU buffers directly
16     MPI_Recv(recv_data.data(), X*Y*Z, MPI_DOUBLE, source_rank, tag,
17              MPI_COMM_WORLD, &status);
```

Listing 2.8: Integrating Kokkos and MPI allows you to directly utilize GPU buffers in GPU-aware MPI calls.

2.3.5 Kokkos build procedure

Kokkos uses *CMake* build system and can be easily integrated into an existing codebase using *CMake*. The user can enable a single device parallel backend (CUDA, HIP or SYCL) along with serial or OpenMP backend. The choice of the backend can be changed simply by compiling Kokkos with the desired backend and linking to that built version of Kokkos. Note that in case of device parallel backends, an additional keyword specifying the compute capability must be added for optimal performance. A detailed list of available configurations and *CMake keywords* can be found in the Kokkos documentation [38].

Chapter 3

Implementation

Transitioning any CPU-only codebase for compatibility with GPU architectures necessitates significantly transforming the underlying codebase's design principles. Kokkos aims to mitigate these challenges, facilitating a smoother, incremental transition of the existing codebase. It achieves this by ensuring compatibility with specific native C++ data structures, such as `std::array`, and seamlessly integrating into the build process. In the context of this thesis, Kokkos has been employed exclusively within the single-phase and multiresolution segments of ALPACA. This section explores the nuances of incorporating Kokkos into ALPACA, highlighting the implementation strategies and challenges encountered.

3.1 Kokkos Integration in ALPACA build

As discussed in the Kokkos overview section, it is straightforward to integrate Kokkos into an existing codebase using *CMake*. Since ALPACA already uses *CMake* as its build system, we can link to the Kokkos library compiled with the desired backend by adding the following in *CMakeLists.txt*:

```
1 find_package(Kokkos QUIET REQUIRED)
2 target_link_libraries(alpaca Kokkos::kokkos)
```

Listing 3.1: Kokkos Integration in ALPACA CMakeLists.txt. find_package requires that the environment variable Kokkos_DIR is set to Kokkos installation path compiled with desired backend

It is important to note that Kokkos uses *nvcc_wrapper* to compile the code when using CUDA backend, which is a wrapper around *nvcc* compiler to handle the compilation of C++ code. One of the caveats of this approach is that the Kokkos backend to be used has to be specified at compile time; for this, users must maintain various Kokkos versions compiled for different backends. Another option is to add the Kokkos as an in-tree build in ALPACA's cmake. However, this would entail that Kokkos will be compiled every time ALPACA is built.

3.2 Incorporating Kokkos::Views

The primary data structure in ALPACA, which stores the data for various fields like conservatives, prime states, boundary jump fluxes and boundary conservatives, in the struct `FieldBuffer`. It contains a `std::array` Fields consisting of 'N' elements, each of which is a d-dimensional array of doubles where d is the active dimensions, determined by the compile-time constant. For instance, if we are solving the Euler equations in three dimensions, the conservatives field buffer would have a `std::array` of five 3-dimensional arrays, one for each active equation being solved namely mass, momentum x/y/z and energy. Since the existing codebase is quite large, to incrementally port the code to Kokkos, it

was decided to add `Fields_K`* to the `FieldBuffer`, which is a `Kokkos::View` in tandem with the `std::array` having the same dimensions. This View would stay on the device and is used for the computations in the `Advance` function.

Since the initialization of the `FieldBuffer` still takes place on the host in the `std::array`, a copy of View is created on the host using the `Kokkos::create_mirror_view` function. To eliminate the need for explicit data transfer between the `std::array` and the host mirror View, the mirror View is set up to directly reference the same underlying data as the `std::array`. This would ensure that a `deep_copy` from the `Fields_K` View to its host mirror View would also update the `std::array` with the new data. It is explained in the figure 3.1.

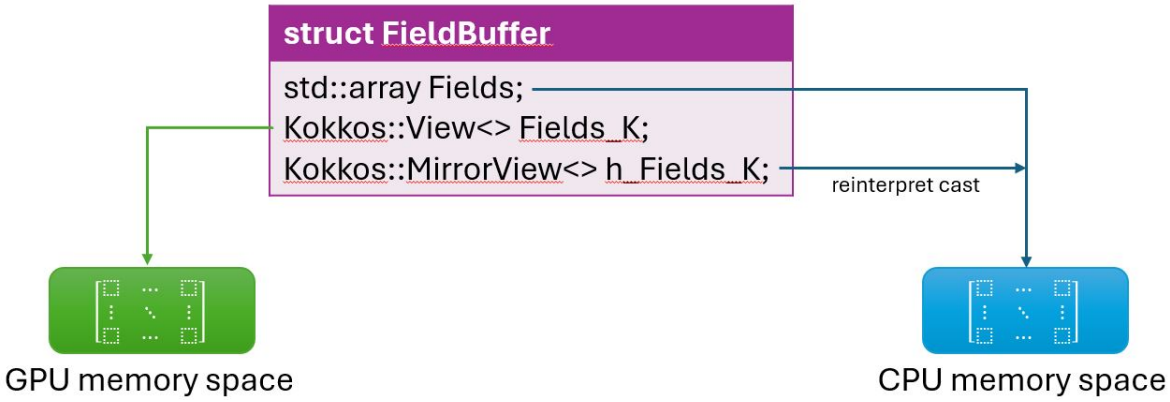


Figure 3.1: Incorporating `Kokkos::View` in `FieldBuffer`, the `Fields_K` view exists on the device, while `h_Fields_K` and `std::array` fields exist on the host, both pointing to the same memory location.

With the addition of `Kokkos::Views` to the `FieldBuffer`, acquiring the buffer existing on GPUs through overloading the access methods inside the block is possible. Since it is impossible to overload a function by return type in C++, tag dispatching is used to select the correct version of the access function. For instance, `auto GetAverageBuffer(Equation const equation)` would return the `std::array` buffer, while `auto GetAverageBuffer(KokkosTag,Equation const equation)` would return the `Fields_K` View. Adding the empty struct `KokkosTag` does not result in additional overhead since the compiler optimizes it as an unused variable.

3.3 Data Management

It is important to note that the `Fields_K` View is created with LayoutRight, which is not the default layout for the execution space `Kokkos::CUDA`. This is done to ensure the mirror View 'sees' the data in the same format as stored in `std::array` when we `reinterpret_cast` it to double.

The usual workflow followed when porting was to first copy the data from the host to the device before a function that is to be ported, modify the function to use the `Fields_K` View instead of the Fields, and then copy the data back to the host after the function has been executed as explained in figure 3.2. This ensures the code is still functional and can be tested for correctness as the porting process is carried out.

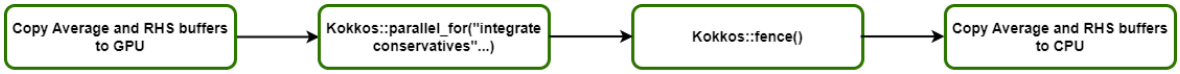


Figure 3.2: Incremental porting of the function `IntegrateConservatives`.

The above approach can quickly become cumbersome once the number of functions ported to

*The subscript K indicates a Kokkos datatype

GPU increases. An enum class was added to denote the `ActiveBufferState` to ease the development process. This enum was added as a member to the `FieldBuffer` and was used to keep track of the currently active buffer. It had two states, CPU and GPU. When a function accesses the buffer from a block using one of its access methods, a check is performed to ensure that the buffer being accessed is the most recent one. If not, the buffer is copied to the required memory, either CPU/GPU and the buffer is returned. This addition meant the only thing required to port a function to GPU was correctly calling the access function with the `KokkosTag`. It would do the necessary copies and then return the `Fields_K` buffer. Note that this was only added to help in the development process and was removed once the porting process was complete.

3.4 Porting

In the discussion found in 2.3.2, two methods of porting are explored, namely `mdrangepolicy` and `teampolicy`. These approaches are thoroughly examined within this section, with practical examples provided from ALPACA to illustrate their application.

3.4.1 Porting using MDRangePolicy

This section shows the porting of tightly nested for loops to `Kokkos::parallel_for` regions using the MDRangePolicy. Here, `IntegrateConservatives` is used as an example function to demonstrate its equivalent version in Kokkos. The function is called inside `IntegrateNode` for each node on each level. It is important to note here that the function is parallelized over a block inside a node, not over the nodes. The loop over nodes remains sequential. The function in the original codebase is as follows:

```
for( Equation const eq : MF::ASOE() ) {
    double const( &u_old )[CC::TCX()][CC::TCY()][CC::TCZ()]
    = block.GetAverageBuffer( eq );

    double( &u_new )[CC::TCX()][CC::TCY()][CC::TCZ()]
    = block.GetRightHandSideBuffer( eq );
    for( unsigned int i = CC::FICX(); i <= CC::LICX(); ++i ) {
        for( unsigned int j = CC::FICY(); j <= CC::LICY(); ++j ) {
            for( unsigned int k = CC::FICZ(); k <= CC::LICZ(); ++k ) {
                u_new[i][j][k] = u_old[i][j][k] + timestep
                                                * u_new[i][j][k];
            }
        }
    }
}
```

Listing 3.2: CPU version of function `IntegrateConservatives`.

In 3.2, the function is called for each equation in the active set of equations. The corresponding buffer is acquired from the block for each active equation. Each block has a fixed number of Internal Cells (IC) in each active dimension. This function can be parallelized over the internal cells or a subset of internal cells and the active equations.

```
Kokkos::parallel_for(
    "IntegrateConservatives",

    mdrange4_policy({0,CC::FICX(),CC::FICY(),CC::FICZ()},
    {MF::ANOE(),CC::LICX()+1,CC::LICY()+1,CC::LICZ()+1}),

    CF::IntegrateConservatives(
    block.GetAverageBuffer(KokkosTag{}).Fields_K,
    block.GetRightHandSideBuffer(KokkosTag{}).Fields_K,
```

```
10      timestep));
```

Listing 3.3: Kokkos version of `IntegrateConservatives`, parallelized over equations and internal cells.

The equivalent Kokkos version of the `IntegrateConservatives` can be seen in 3.3. Here `mdrange4_policy` is a typedef for Rank 4 `Kokkos::MDRangePolicy` as can be seen in 3.4. The CF is short for namespace `CellFunctors`, which consists of all the functors needed by `Kokkos::parallel_for` regions.

```
1     using mdrange4_policy = Kokkos::MDRangePolicy<Kokkos::Rank<4>>;
```

Listing 3.4: `MDRangePolicy` Rank 4 used to iterate over 4 indices.

```
1 struct IntegrateConservatives {
2     KokkosFieldBufferViewConst const u_old_;
3     KokkosFieldBufferView u_new_;
4     double const timestep_;
5
6     IntegrateConservatives( KokkosFieldBufferViewConst const u_old,
7     KokkosFieldBufferView u_new, double const dt )
8         : u_old_( u_old ),
9           u_new_( u_new ),
10          timestep_( dt ) {}
11
12    KOKKOS_INLINE_FUNCTION void operator()(int const eq,int const i,
13                                           int const j,int const k)
14    const {
15        u_new_( eq, i, j, k ) = u_old_( eq, i, j, k )
16                              + timestep_ * u_new_( eq, i, j, k );
17        }
18    };
```

Listing 3.5: `IntegrateConservatives` functor featuring `operator()` and constructor.

Listing 3.5 shows the contents of `CF::IntegrateConservatives` functor which carries out the actual computation of integration of conservatives. The functor takes the `Conservatives` Field Buffers, `AverageBuffer` and `RightHandSideBuffer` from the material block and the timestep as the input arguments needed for the integration step. As explained previously, the `Fields_K` is a `Kokkos::View`, the GPU counterpart of the Fields array inside `FieldBuffer`. Inside the functor, the `operator()` is overloaded so that the `parallel_for` can access it with thread indices. The `operator()` is marked with a `KOKKOS_INLINE_FUNCTION` which ensures that the operator can be called on device. It is important to note here that the functor is constructed every time this function is called, so it must not be expensive to construct this functor. Since the `Kokkos::Views` are like shared pointers pointing to the memory on GPU, copying them is not expensive, so they are passed by value to the functor. Also, the `operator()` is required to be const by Kokkos for thread-safe access to the functor. So only the data pointed to by the pointers can be changed inside these functors and all the other attributes have to remain constant.

Using this approach, all the tightly nested for loops can be transitioned to `Kokkos::parallel_for` region, which can be executed on GPU using CUDA backend or on CPU threads if using OpenMP backend. It is important to note that in the case of Serial backend, the loops are executed as they would be in a regular CPU version.

3.4.2 Porting using TeamPolicy

There are cases when the loops are not tightly nested, and for such hierarchical parallelism, as explained in the section 2.3.2, team policy can be used. Here, the computation of viscous dissipation fluxes in each direction is used to demonstrate the use of teams policy and hierarchical parallelism.

```
for(unsigned int i = 0; i < CC::ICX(); ++i) {
    for(unsigned int j = 0; j < CC::ICY(); ++j) {
        for(unsigned int k = 0; k < CC::ICZ(); ++k) {

            for(unsigned int r = 0; r < DTI(DS::DIM()); ++r) {
                // subtract tau_fluxes from dissipative fluxes ...
            }

            if constexpr(IsEquationActive(Equation::Energy)) {
                // add energy flux in each direction, which involves
                // inner product
            }
        }
    }
}
```

Listing 3.6: CPU version for complete viscous dissipative fluxes.

The first notable area where parallelization can be effectively implemented is within the outermost layer, specifically across the three 'for' loops. Expanding further, another promising avenue for parallel execution lies within the loops that iterate over various dimensions. Moreover, a deeper analysis reveals that the inner product calculation, which inherently involves iterating over dimensions, presents itself as an additional opportunity for parallelization.

```
Kokkos::parallel_for("ViscousFluxesCompleteDissipativeFluxes",
                Kokkos::TeamPolicy(CC::ICX()*CC::ICY(), Kokkos::AUTO),
                CF::ComputeViscousFluxes(dissipative_flux_x,
                                         dissipative_flux_y,
                                         dissipative_flux_z,
                                         tau_flux,
                                         velocity_at_cell_faces)
                    )
```

Listing 3.7: GPU version for complete viscous dissipative fluxes showcasing use of team policy.

The Kokkos equivalent version of 3.6 is shown in 3.7. Here, the two arguments for `Kokkos::TeamPolicy` are the number of teams to launch and the number of threads per team. This is equivalent to CUDA's number of blocks and threads per block. The user can select the number of teams, while the other argument is dependent on hardware, and using `Kokkos::AUTO` lets Kokkos choose appropriate threads per team. In this case, the number of teams is chosen to be the size of the outer two loops in 3.6. So essentially, we launch as many teams as the outer for loops in the X and Y directions. As before, `CF::ComputeViscousFluxes` is a functor that does the actual computation.

```
struct ComputeViscousFluxes {
    // Attributes and constructor for the functor...
    KOKKOS_INLINE_FUNCTION
    void operator()(const member_type& teamMember) const{
        //acquire the i and j indices from the league_rank
        int const i = teamMember.league_rank() / CC::ICY();
        int const j = teamMember.leagure_rank() % CC::ICY();

        //first parallelize over Z-direction
        Kokkos::parallel_for(
        Kokkos::TeamThreadRange( teamMember, CC::ICZ() ),
                                 [=, this]( int k ) {
            //Parallelize similarly over dimensions
            //and subtract tau_fluxes from dissipative fluxes
            Kokkos::parallel_for("parallel over dimensions");

            //If energy equation is active, calculate energy_flux
            //using parallel_reduce to replace inner product
            if constexpr(IsEquationActive(Equation::Energy)) {
```

```
20                  double energy_flux_x, energy_flux_y, energy_flux_z;
21                  Kokkos::parallel_reduce(....); //reduce into energy_flux
22                  //after reduction subtract the energy flux from
23                  //dissipative fluxes...
24              }
25          }); //Z-direction
26      } //operator()
27  };
```

Listing 3.8: Functor demonstrating complete viscous dissipative fluxes using a team policy.

The code listing 3.8 shows the contents of the functor. The `operator()` is overloaded in the functor, similar to the way shown in the previous section. However, it takes an argument of type `Kokkos::TeamPolicy<>::member_type`, which gives us built-in functions if we want to perform some operation as a team. This differs from the `MDRangePolicy`, which takes the indices i,j, and k as the argument. Since we launched as many teams as the outer loops, we must first acquire the indices i and j from the team's rank. Then, we launch a parallel region from within each team for the Z-direction. Again, each thread launched from the Z-direction launches a parallel region to parallelize over the dimensions for the `tau_fluxes` subtraction. Once this is done, the energy fluxes are calculated as a reduction to replace the inner product. Note that `std::inner_product` is a CPU function and cannot be used directly on GPU. It has to be replaced by an equivalent for loop. Once the `energy_flux` is calculated, it can be subtracted from the dissipative fluxes to obtain their final value. This shows that a fine-grained parallelism can be obtained using team policy.

3.5 Regression Testing

There were many changes made to the base ALPACA version and it is necessary to check if the implementation does not introduce any new bugs. To ensure that the ported code is working correctly, regression testing was carried out. The regression testing was done by comparing the final output hdf5 files written for three cases namely, Rayleigh Taylor Instability, Toro Implosion, and Full Implosion from the ALPACA testsuite. The differences observed for CUDA backend are outlined below:

Table 3.1: Regression Testing of the CUDA backend.

Case	Density	Pressure	VelocityX	VelocityY	VelocityZ
Rayleigh Taylor	2.903e-11	1.0778e-11	6.868e-12	6.5884e-12	0.0000
Toro Implosion	5.0529e-13	4.6886e-13	2.9266e-13	2.9031e-13	0.0000
Full Implosion	8.2079e-03	8.2589e-03	2.6368e-03	2.6368e-03	3.6155e-03

As can be seen in the the table 3.1, there are some discrepancies between the CUDA backend and the ALPACA develop branch. These discrepancies are due to the differences in GPU and CPU architectures as outlined in 2.2.1, different instruction sets, and different precision support. It is also observed that the discrepancies are more pronounced in the Full-Implosion case, which had a non-zero shear viscosity of $1e-5$. On setting the shear viscosity to zero, the discrepancies in Full-Implosion case were reduced to the same order of magnitude as the other two cases.

Since OpenMP backend runs on CPU, it was observed that these discrepancies vanished for all the three cases and the results matched exactly with the ALPACA develop branch. This shows that the current ported version works correctly and any floating point discrepancies are due to the differences in GPU and CPU architectures.

3.6 Optimizations

This section discusses the possible optimizations for the initial Kokkos version. Some of these optimizations are not included in the final version of the code as they require some additional changes to the data structures of ALPACA. However, their impact on the code performance is discussed if it is decided to include them in future code versions.

3.6.1 Allocation of temporary buffers

Some functions in the codebase involve creating temporary buffers for calculation. For instance, consider the function `UpdateFluxes` from the space solver class. It has various buffers at the start of the function, such as the `ValidFaceFluxIndicator`, `FaceFluxes`, and `VolumeForces`. This function then calls the `UpdateConvectiveFluxes`, which has buffers for `advection contribution`, `roe eigenvectors left/right` and `fluxfunctionwavepspeeds` (eigenvalues). Moreover, the `ComputeFluxes` function in `UpdateConvectiveFluxes` also has `flux_simple_` buffer, which it uses to calculate fluxes if the positivity check is active. There are similar buffers inside the source term solver. When porting these functions to Kokkos, these buffers must be converted into `Kokkos::Views` instead of `std::arrays`. This ensures they are allocated on GPU when the default execution space is CUDA.

This brings us to the critical issue arising from many GPU allocations/deallocations. The entire loop of `UpdateFluxes` is called for all the nodes, meaning every time the loop is called, the temporary buffers get allocated, are used inside the functions, and then are destructed. While this is fine for a small number of nodes, it creates a significant bottleneck for a large number of nodes, severely impacting the performance. This effect can be observed by running the memory events profiling tool available in Kokkos.

The allocated buffers were moved to the respective classes as attributes to resolve this issue. This means that they would be allocated only once during the initialization phase and would get reused. This modification reduced the device memory allocations from 877768 to 30583. It also improved the performance of the GPU version by about 8% when tested for RTI case in ALPACA testsuite.

3.6.2 Removing Kokkos::fences and setting execution space argument in deep_copy

As discussed earlier, the CPU and GPU have different execution and memory spaces. This means that the CPU can queue the kernel launches on the GPU and only wait for the GPU when results of specific GPU kernels are required on the CPU, like reduction in case of timestep computation. The `Kokkos::fence()` synchronizes all the execution spaces in the code and blocks the code until the GPU finishes computation. This is detrimental to the performance of the code. In addition, there are certain implicit blocking operations, like deallocating a `Kokkos::View` and `Kokkos::deep_copy` without any execution space argument. The deallocations issue was taken care of by the previous optimization. For the deep copying behaviour, Kokkos provides another overload of `deep_copy`, which takes in the execution space as the argument. If the `DefaultExecutionSpace` is given as an argument to the `deep_copy`, which is `Kokkos::Cuda`, when compiled for CUDA backend, it gets queued like the other kernels on GPU and is not blocking. This ensures that the only places we block CPU and GPU are where we reduce something, like the calculation of error norm in the `Remesh` function and the timestep calculation. In other places, there is no need for the CPU to wait and synchronize with GPU operations. This optimization led to an improvement of 1.3x in GPU performance for the RTI case in ALPACA testsuite.

3.6.3 Loop fusions

The launch of a kernel on a GPU is a fundamental operation in GPU programming. It involves transferring control from the CPU to the GPU, which incurs some latency. This latency is not negligible, so it's crucial to minimize the number of kernel launches wherever possible. One effective strategy is to combine loops that have similar iteration ranges. Let's consider the example of `ComputeVectorGradientAtCellFaces`. In this function, the reconstruction is carried out for each dimension. The loops differ within the dimension only by 1 depending on the dimension which is under consideration. For instance, the loop ranges go as:

- DIRECTION-X FROM `[FICX-1,FICY,FICZ,0,0]`
  TO `[LICX+1,LICY+1,LICZ+1,DIM,DIM]`
- DIRECTION-Y FROM `[FICX,FICY-1,FICZ,0,0]`
  TO `[LICX+1,LICY+1,LICZ+1,DIM,DIM]`
- DIRECTION-Z FROM `[FICX,FICY,FICZ-1,0,0]`
  TO `[LICX+1,LICY+1,LICZ+1,DIM,DIM]`

Here, all the indices are the same except for the First internal cell of the respective dimension. To combine these loops into one loop, we can launch `parallel_for` with the range:

- FROM `[FICX,FICY,FICZ,0,0]`,
  TO `[LICX+1,LICY+1,LICZ+1,DIM,DIM]`

Then depending on the dimension, we can subtract one from the First Internal cell index inside the `operator()` of the functor. This reduces the kernel launches by a third for 3D cases. In this example, the loop indices were close enough to consider combining them. In cases where the loop indices are far off, though the loops can be combined, conditional statements inside the kernel must be added to ensure that the accesses are within bounds. This not only creates bug-prone and difficult-to-read code, but it also harms GPUs when conditionals are added inside a kernel.

3.6.4 Concurrent GPU kernels (streams)

This experimental optimization was tried to alleviate the sequential traversal over nodes in the computation of the right-hand side. As mentioned, the operations over field buffers inside a block are parallelized in the current implementation. However, in practice, multiple nodes are traversed sequentially within each function. For instance, consider the function `ComputeMaterialRightHandSide` shown in 3.9.

```
for( auto const& level : levels ) {
    for( Node& node : tree_.LeavesOnLevel( level ) ) {
        // compute fluxes for levelset and materials and store in
        // conservatives rhs
        space_solver_.UpdateFluxes( KokkosTag{}, node );
    }
}
```

Listing 3.9: `UpdateFluxes` in the CPU version of `ComputeMaterialRightHandSide`.

Here, the fluxes are computed for each node in the tree. Since the nodes are independent on each other, the computation of fluxes for each node can be done concurrently. This can be achieved by launching concurrent kernels on GPU using CUDA streams. The streams are independent of each other and can run concurrently on GPU. The equivalent Kokkos version of the CUDA streams is the `Kokkos::partition_space`. This function takes in an execution space and a vector of weights. It partitions the specified execution space into multiple spaces which can run concurrently. However, one important thing to note is that the execution queues of created partitioned spaces are no longer

synchronised and require explicit fences before proceeding with the computation. This is demonstrated in listing 3.10.

```
//Make sure the previous work on GPU is finished
Kokkos::DefaultExecutionSpace().fence();

//required to dispatch alternate nodes to different streams
int nodeCounter = -1;
for( auto const& level : levels ) {
    for( Node& node : tree_.LeavesOnLevel( level ) ) {
        // compute fluxes for levelset and materials and store
        // in conservatives rhs
        nodeCounter++;

        // pass the required stream and the partition space to the
        // function
        space_solver_.UpdateFluxes( KokkosTag{}, node,
                                    nodeCounter % 2,
                                    cuda_instances[nodeCounter % 2] );
    }// node
}// level

//Make sure all partition spaces are finished with their work and
//synchronize with the default space
cuda_instances[0].fence();
cuda_instances[1].fence();
Kokkos::DefaultExecutionSpace().fence();
```

Listing 3.10: `ComputeMaterialRightHandSide` streams version with two streams, alternating node assignments.

This is because the partitioned spaces are unaware of each other and can run concurrently. If these fences are not added, a race condition might occur, which is difficult to debug. Considering these things, a vector of two `Kokkos::Cuda` execution spaces was stored as an attribute in the modular algorithm assembler for this experiment.

```
std::vector<Kokkos::Cuda> cuda_instances
         = partition_space( Kokkos::DefaultExecutionSpace(), 1, 1 );
```

Listing 3.11: Creating a vector of `Kokkos::Cuda` execution spaces with two equally weighted partition spaces.

This creates two streams with equal weights in addition to the default stream. Since there are now two streams, all the temporary buffers which were discussed in section 3.6.1 now have to have two of those buffers. This results in more memory footprint than the version without streams. Once these are added, we can simply launch the subsequent functions inside `UpdateFluxes` with the particular `cuda_instance`. For instance consider the `ComputeAdvection` function as shown in listing 3.12.

```
//launch the parallel_for on that partition space(cuda_instance)
Kokkos::parallel_for("ComputeAdvection",
                     mdrange3_policy(cuda_instance,
                                     {0,0,0},
                                     {CC::TCX(),CC::TCY(),CC::TCZ()}),
CF::ComputeAdvection<DIR>( ... ) );
```

Listing 3.12: `ComputeAdvection` streams version demonstrating the use of partition space in MDRange policy.

With these additions, it was observed that the performance was improved by some margin in the case of 32 internal cells, while the cases of internal cells 16 and 128 produced no significant improvement. These are discussed more in detail in section 4.1.3. In general, it was observed that the streams only benefit when the kernels are small enough to be executed concurrently, but at the same time are sufficiently big to run long enough to create overlap.

One possible solution to maximize overlap is to launch the `UpdateFluxes` itself with multiple threads using `std::threads` or using `Kokkos::pthreads` backend. This would launch the kernels

concurrently on the GPU. However, the `UpdateFluxes` function has to be thread-safe for this to work. The function currently takes in the node by reference in the current loop iteration. In the case of doing this in a multi-threaded environment, this would cause an issue that when the loop goes to the next iteration, the reference is updated, and this would cause undefined behaviour. So, it is required that we have to wait before the threads complete queuing their work on GPU by joining the spawned threads inside each iteration. Overall, using multi-GPU to reduce the number of nodes per GPU could be beneficial instead of doing streams on a single GPU. However, to explore this in detail, more work has to be done to modify data structures in ALPACA to make it thread-safe.

3.6.5 AoS vs SoA (Combined GPU Buffers)

The current data structures in ALPACA involve a node consisting of a block per each material it contains. For single-phase cases, each node consists of one block. Each block, in turn, has various Field Buffers, as discussed previously. The tree stores all the nodes at a particular multiresolution level. Hence, it forms a data structure similar to an Array of Structures. This leads to an iteration loop over all nodes whenever we want to compute on specific field buffers. It is important to note that the update over nodes is independent in certain code sections. For instance, the computation of the right-hand side and integration step can be done parallelly over nodes. With the current data structure of trees, nodes and blocks, it is not possible to access all the buffers in parallel on the GPU. A possible solution is changing the data structure to Structure of Arrays. This structure would contain an array of all Field buffers from all the nodes. Since the `Kokkos::Views` are essentially shared pointers, a separate data structure can be maintained, which can be initialized after the entire tree structure is constructed on the CPU.

```
struct GPU_BUFFERS {
 std::array<KokkosFieldBufferView, CC::NN()> averagesAll;
 std::array<KokkosFieldBufferView, CC::NN()> rightHandSideAll;
 std::array<KokkosFieldBufferView, CC::NN()> initialsAll;
 std::array<KokkosFieldBufferView, CC::NN()> primeStatesAll;
 std::array<KokkosFieldBufferView, CC::NN()> parametersAll;

 //similar buffers for jump boundaries, jump fluxes, etc...
}
```

Listing 3.13: Structure of Arrays `GPU_BUFFERS` consists of multiple arrays, where each array comprises `CC::NN()` number (set to 512 at compile-time) of Kokkos Field Buffers.

In 3.13, the `CC::NN()` is the number of nodes added as a compile-time constant in user specifications in ALPACA. For this experiment, `CC::NN()` was selected as 512. These arrays can be populated with the respective `KokkosFieldBufferViews` after the simulation is initialized. Since the `Kokkos::Views` are shared pointers, these arrays are assigned by using a shallow copy, and there is no need to update the other data inside the block. This means that the `GPU_BUFFERS` can be passed to the functions that require all the buffers in parallel. This would also mean that the temporary buffers mentioned in section 3.6.1 must be allocated for 512 nodes, as all these node calculations will be done in parallel. Once all these changes are done, we can modify the `MDRangePolicy` of existing loops to include another dimension consisting of the number of nodes as shown in 3.14.

```
Kokkos::parallel_for( "ComputeAdvection",
                      mdrange4_policy( {0,0,0,0},
               {nodeCount,CC::TCX(),CC::TCY(),CC::TCZ()}),

CF::ComputeAdvection<DIR>(advection,
                          averagesAll,
                          primeStatesAll));
```

Listing 3.14: `ComputeAdvection` for the `GPU_BUFFERS` version with modified `MDRangepolicy` to include the number of nodes.

The section 4.1.3 provides the results of this experiment in detail. This experiment aimed to show that there is a scope to extract more parallelism by parallelizing over nodes. This can further be extended to parallelize the internal halo updates as well. That would require maintaining two lists of nodes, one containing all the parents and the other containing their corresponding neighbours. Then, the halo updates can be performed concurrently on all the nodes.

3.7 Issues

As mentioned earlier in the section, porting a code to run on another hardware such as GPU requires major changes to the philosophy of the codebase. This section describes some of the issues faced during the porting process.

3.7.1 Runtime polymorphism

There is a clear distinction between the memory and execution spaces of CPU and GPU. As such, the functions to be executed on GPU have to be marked with special directives such as `__device__` or `__global__` in CUDA. These directives are helpful when we want to re-use the same function on CPU and GPU. These directives enable the compiler to create device-side code to execute these functions on GPU. The Kokkos way of specifying the `__global__` directive is to use the `KOKKOS_INLINE_FUNCTION`. This directive is used to specify that the function is to be executed on GPU.

In the case of runtime polymorphic classes, the Base class pointer is used to point to a Derived Class object. The correct call to the function is resolved by using Vtables and VPointers. A call to a virtual function dereferences the VPointer and then finds the correct function in its VTable. In the case of GPU, there are two VTables, one which resides on the CPU and one on the GPU. When we copy a virtual object to the GPU, it also copies the VPointers it contains, essentially pointing to addresses on the CPU. This creates an issue at runtime because calling a CPU function on a GPU is not legal. This behaviour was observed in the case of the Equation Of States object in ALPACA. It has a base class EquationOfState, which consists of derived EOS like Stiffened Gas[13], Noble-Abel[23], Stiffened Gas Complete[25] and Tait[10].

Consider, for instance, the function `ConvertConservativesToPrimeStates` as shown in listing 3.15. The EOS object is used to calculate the pressure and the speed of sound. This object is passed by value to the functor. This object is used inside the functor to call the `ComputePressureAndTemperature` function. However, since the EOS object is initialized on the CPU, its VTable points to the CPU's function, leading to an error.

```
Kokkos::parallel_for( "ConvertConservativesToPrimeStates",
                      mdrange3_policy({ 0, 0, 0 },
                     { CC::TCX(), CC::TCY(), CC::TCZ() }),
                     CF::ConvertConservativesToPrimeStates(
                         conservatives.Fields_K,
                         prime_states.Fields_K,
                         material_manager_.GetMaterial( material ).
                         GetEquationOfState() ) );
```

Listing 3.15: `ConvertConservativesToPrimeStates parallel_for` region, where the functor takes an EquationOfState object by value for use in its `operator()`.

There are two possible solutions to this problem:

- Initialize the EOS object directly on GPU by using placement new.
- Use CRTP and std::variant to replace runtime polymorphism with compile-time polymorphism.

Both these approaches come with their own set of challenges. In the first approach, once an EOS object is initialized directly on the GPU, it would not be possible to access it on the CPU should there be any requirement. Also, the container inside Material, which contains the EOS, has to be adapted to store the pointers to the EOS object on the GPU. In the second approach, flexibility is lost when selecting the EOS at runtime. Also, the EOS cannot be contained inside a container in material class for multiple materials.

The second approach was tried in the codebase. First, a `std::variant` was created as can be seen in 3.16, which contained the various equations of States.

```
typedef std::variant<StiffenedGasKokkos,
                     StiffenedGasKokkosSafe>
                     EquationOfStateVariant;
```

Listing 3.16: `std::variant-based` generic EquationOfState supporting two EOS types that can be copied on the device.

Within this variant, only the required state equation can be initialised depending on whichever EOS was requested in the instantiation procedure. To access the correct EOS, we would then need to place the entire call to functor inside an `std::visit` as can be seen in listing 3.17, which would then copy the correct EOS object to GPU.

```
auto eos_variant = material_manager_.GetMaterial( material ).
GetEquationOfStateVariant();
      std::visit( [&]( auto&& eos ) {
         CF::ConvertConservativesToPrimeStates trial(
            conservatives.Fields_K,
            prime_states.Fields_K,
            eos );
        Kokkos::parallel_for( "test",
                              mdrange3_policy(
                                 { 0, 0, 0 },
                           { CC::TCX(), CC::TCY(), CC::TCZ() } ),
                           CF::ConvertConservativesToPrimeStates(
                               conservatives.Fields_K,
                               prime_states.Fields_K,
                               eos
                           ) );
      },
                    eos_variant );
```

Listing 3.17: Usage of `EquationOfStateVariant` where `std::visit` selects the correct EOS and copies it to the functor.

However, for this thesis, it was decided to implement only one equation of state of type Stiffened Gas. This could be made generic later with one of the abovementioned approaches.

3.7.2 Race Conditions

When transforming sequential for-loops to multi-threaded versions to be used on GPU, one aspect to remember is the race conditions that might occur due to multiple threads accessing and modifying the same memory location. For instance, consider the function `FluxSplitting::ComputeFluxes` as shown in listing 3.18

```
// initialize loop counters
std::array<...> positive_characteristic_flux;
std::array<...> negative_characteristic_flux;
std::array<...> characteristic_flux;
for( unsigned int i = x_start; i <= x_end; ++i ) {
    for( unsigned int j = y_start; j <= y_end; ++j ) {
        for( unsigned int k = z_start; k <= z_end; ++k ) {
            // Calculate the fluxes...
```

```
            }
        }
}
```

Listing 3.18: Original version of `FluxSplitting::ComputeFluxes` accesses arrays sequentially, allowing for reuse.

For porting the function naively to Kokkos, the for loops can be replaced with `Kokkos::parallel_for` and pass the necessary arrays `positive_characteristic_flux`, `negative_characteristic_flux` and `characteristic_flux` to the functor. However, this would mean that all the threads would access the exact same copy of the arrays inside the functor and modify them. This would ultimately lead to race conditions and incorrect results. A possible fix for this is to move the initialization of arrays inside the `operator()` of the functor. This would mean that each thread now has its copy of the array and can modify it without any race condition. However, this also means more memory footprint as each thread allocates and deallocates the arrays every time this function is called. A general rule of thumb to avoid race conditions is to ensure that the memory being accessed inside the `operator()` should depend on the thread indices. If that is not the case, then there is a high chance there will be a race condition on that data.

3.7.3 Kokkos::subview type

The `Kokkos::subview` is used to create a slice of the original view, as mentioned in the section 2.3. This is useful when we want to access only a part of the view. The type of this `Kokkos::subview` is an implementation detail that is hidden by Kokkos by default. It is recommended to use the auto keyword to declare the subview type. However, in some cases, the type of subview must be known. For instance, consider the `KokkosFieldBufferView`, a typedef for a `Kokkos::View` containing 'N' Fields. However, in some cases, only a single Field is required inside a function. To pass this subview as a function argument, we must explicitly define the subview type. Kokkos provides an alias template `Kokkos::Subview` for this, which can be used to define the type of Kokkos subview. This can be seen in the example below in listing 3.19.

```
struct //subViewHolder
{
    Kokkos::Subview<KokkosFieldBufferView,
                    int,
                    decltype( Kokkos::ALL ),
                    decltype( Kokkos::ALL ),
                    decltype( Kokkos::ALL )>
              s;
    } subViewHolder;
```

Listing 3.19: `Kokkos::Subview` alias template to deduct the type of the subview of `KokkosFieldBufferView`

First, we need to define a struct containing the member `Kokkos::Subview`, defining the actual subview we want to take. Then, we can use `decltype` on this member 's' to get the subview type as shown in listing 3.20.

```
using KokkosFieldView = decltype(subViewHolder.s);
```

Listing 3.20: `KokkosFieldView` is now introduced as a new subview type of the `KokkosFieldBufferView`.

This newly created typedef can be used wherever there is a need to define the type of subview of `KokkosFieldBufferView`. Similarly, whenever we must explicitly define a subview type of a `Kokkos::View`, we must use this `Kokkos::Subview` idiom to create a new typedef, which can then be used.

Chapter 4

Results

A variety of factors affect the GPU performance of ALPACA like the number of levels, number of blocks and the total number of cells. This section compares the performance of the CUDA, OpenMP and serial backend of Kokkos with the base ALPACA version. The single-phase Rayleigh Taylor Instability test case is used for these tests. It consists of a heavy fluid layer accelerated into a lighter fluid by gravity. A small disturbance is added in the initial condition which leads to formation of mushroom shaped instability. All the tests are performed on kushana cluster, specifications of which are mentioned in table 2.1b. The NVIDIA A6000 GPU is based on AMPERE architecture. Note that it can only support native-single precision operations. All the double-precision operations are emulated using software and are slower in comparison to native-double precision GPUs like NVIDIA V100.

The tests are divided into three major parts. The first part tests the scaling of CUDA backend for varying total cells as well as varying internal cells. The second part consists of function wise profiling of serial and CUDA backends compared with single core and 16 core runs respectively of base ALPACA. Finally the last section deals with the performance of OpenMP compared with base ALPACA. All the tests performed are averaged over 3 runs. For the GPU runs, the standard deviation on average did not exceed 6%, with an outlier having 20% deviation. For the CPU runs on the another hand, more standard deviations were observed, with some values exceeding 30%.

4.1 CUDA backend

This section deals with the performance of the CUDA backend of Kokkos. The tests are performed for both increasing number of total cells and constant total cells. Also, the effect of remeshing is studied to assess the efficiency of CUDA backend to handle the node refinement. The speedup is calculated as the time taken by 8/16 cores of CPU divided by the time taken by the GPU.

4.1.1 Increasing total cells

This section deals with scaling study with increasing number of total cells. Cell is the smallest element in the mesh on which the governing equations are solved. Multiple cells make up a block/node. These blocks are refined when they satisfy certain criteria. Multiple such blocks make up the final domain. As such, there are two main ways to increase the total number of cells whilst keeping the remeshing disabled (MR level 0). One is to increase the number of nodes or blocks while keeping the internal cells constant. Another way is to increase the number of internal cells while keeping the number of nodes constant

Internal Cells Constant-Increase Number of Nodes

This section deals with the first way to increase total cells. The internal cells are kept constant and the number of nodes are increased. The cell size is kept constant at 0.015625 for all the cases. To ensure that the CPU and GPU versions are compared fairly, the number of iterations is fixed to 10000.

Table 4.1: Speedup (CPU Time / GPU Time) for increasing total cells and the number of nodes for dimension=2, where the node ratio is the number of nodes in the x-direction **X** number of nodes in the y-direction. The IC denotes the number of internal cells in a block for block based multiresolution.

	NodeRatio 4x4		NodeRatio 8x8		NodeRatio 16x16	
Internal Cells	**Total Cells**	**Speedup**	**Total Cells**	**Speedup**	**Total Cells**	**Speedup**
IC16	4096	0.0703	16384	0.0521	65536	0.0468
IC32	16384	0.1617	65536	0.1498	262144	0.1486
IC64	65536	0.3259	262144	0.3472	1048576	0.3368

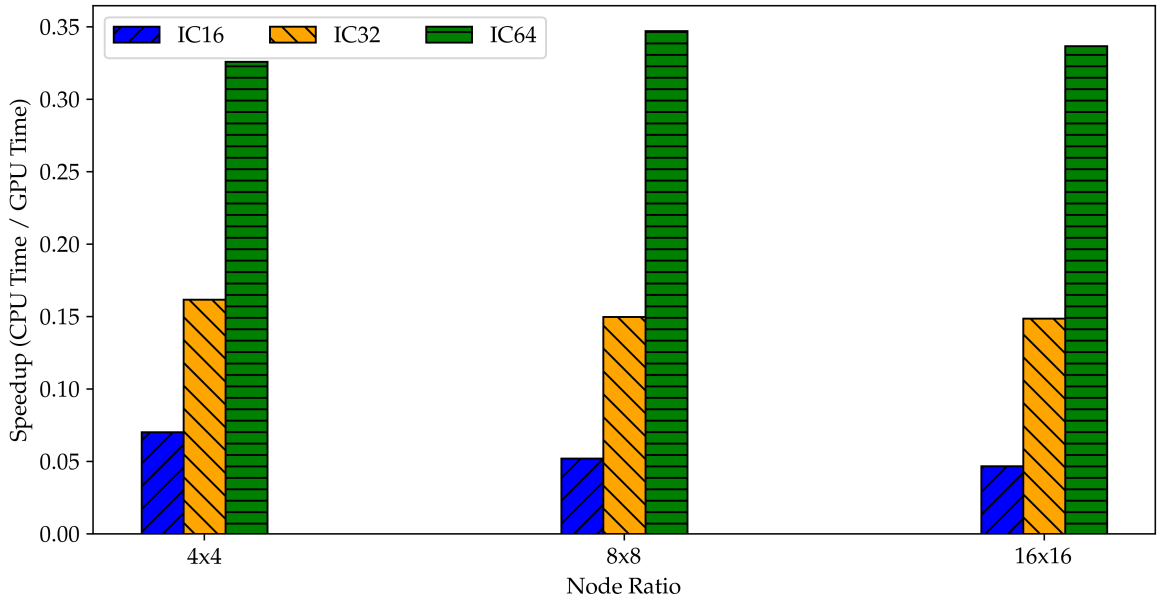


Figure 4.1: Speedup (CPU time / GPU time) for increasing total cells and increasing node ratio for dimension=2.

Figure 4.1 shows the speedups for the cases in table 4.1. As can be seen no speedup is observed when comparing 16 cores of CPU to a single GPU for two dimensional case. The speedup remains almost constant when going from 16 nodes to 256 nodes for each constant internal cells. It was observed that the number of internal cells of 16 for 2D is particularly suitable for CPU which might have better cache behvior. Also, distributing the cases across 16 ranks largely reduces the load on a single core of the CPU where each core can have anywhere between 1 node to 16 nodes as compared to load on GPU which can range between 16-256 nodes. It was also observed from the output of `nvidia-smi` that the GPU utilization was varied between 75-90% for the 2D cases, with the maximum utilization achieved for case with more internal cells. Though higher occupancy or GPU utilization does not always mean more efficiency, less occupancy almost always interferes with the ability to hide memory latencies[26]. Another way to look at the timings is the scaling behavior of the GPU when nodes are increased vs when internal cells are increased.

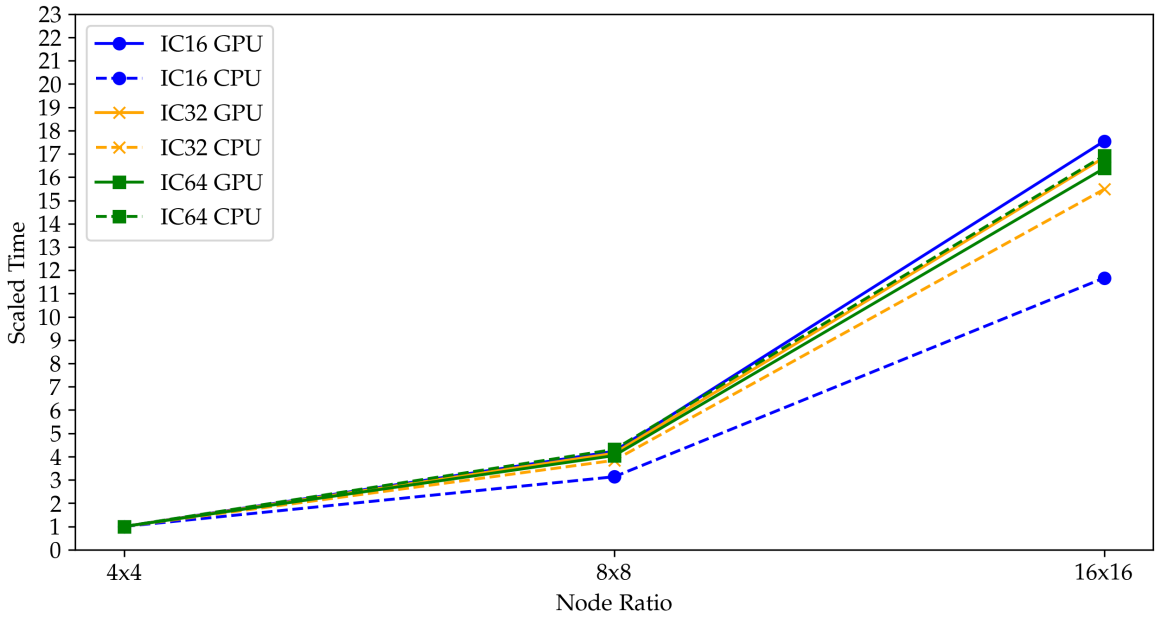


Figure 4.2: Scaling with node ratio, where compute loop times are scaled relative to the time taken for a 4x4 node configuration, for dimension=2. The node ratio is defined as the number of nodes in the x-direction **X** number of nodes in the y-direction.

In figure 4.2, the scaling behavior of GPU and CPU is observed when the number of nodes are increased. All the times are scaled to the smallest case of 4x4 nodes. It can be seen that going from 4x4 nodes to 8x8 and then to 16x16, the times are quadrupled every time for both the CPU and GPU cases. It is expected behavior since the nodes and subsequently total cells are also quadrupled for each case.

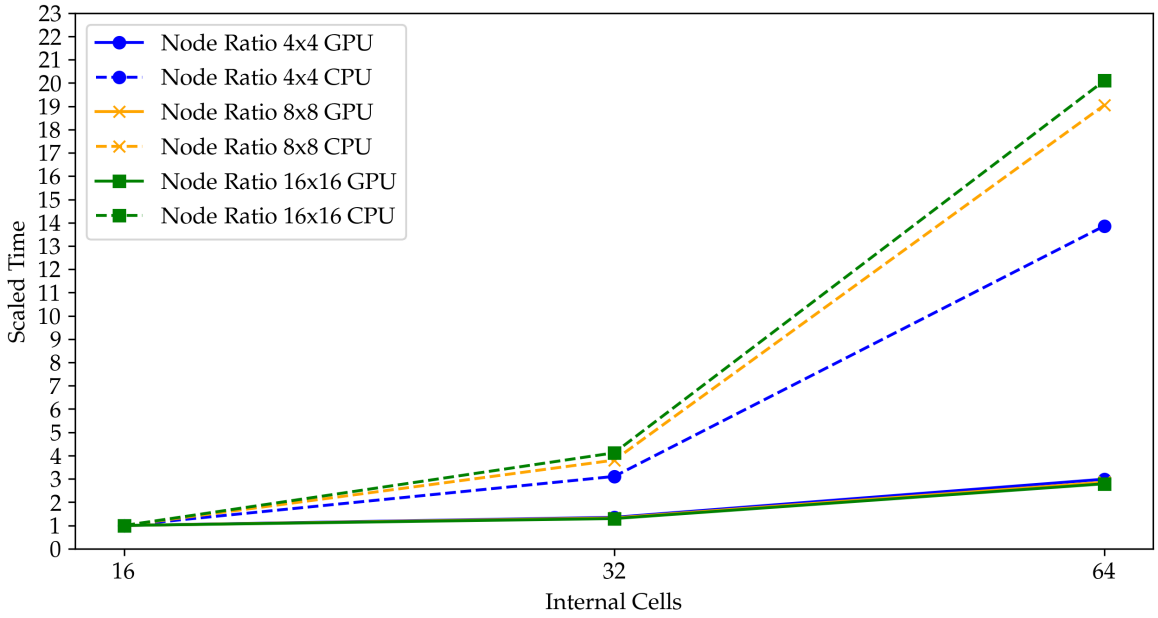


Figure 4.3: Scaling with internal cells, where compute loop times are adjusted relative to the time taken for the Internal Cells 16 case, for dimension=2.

Interestingly, as seen in figure 4.3, the GPU scales much better than CPU when going from internal cells 16 to 64. The CPU still continues to exhibit the same behavior of quadrupling time for each

increase in internal cells. But GPU times increase only by 1.5-1.7x for each increase in internal cells even though the total cells are getting quadrupled. This indicates that the GPU is better utilized when there are more internal cells than there are more nodes. This behavior is studied in more detail in the next section.

Table 4.2: Speedup (CPU Time / GPU Time) for increasing total cells and the number of nodes for dimension=3, where the node ratio is the number of nodes in the x-direction **X** number of nodes in the y-direction **X** number of nodes in z-direction. The IC denotes the number of internal cells in a block for block based multiresolution.

	NodeRatio 2x2x2		NodeRatio 4x4x4		NodeRatio 8x8x8	
Internal Cells	**Total Cells**	**Speedup**	**Total Cells**	**Speedup**	**Total Cells**	**Speedup**
IC16	32768	2.6141	262144	1.4286	2097152	1.3573
IC32	262144	6.4371	2097152	3.0883	16777216	2.1433
IC64	2097152	7.1406	16777216	3.3809	-	-

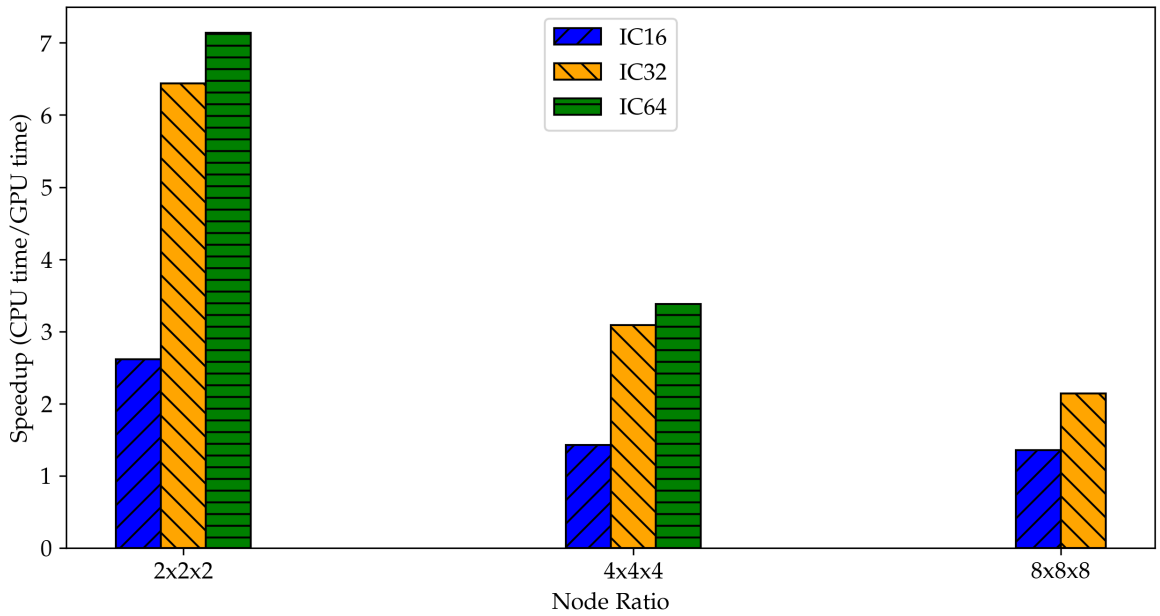


Figure 4.4: Speedup (CPU time / GPU time) for increasing total cells and increasing node ratio for dimension=3. The 2x2x2 case is run with 8 MPI ranks, rest of the cases are run with 16 MPI ranks.

As seen in figure 4.4, a speedup of 7.14x was achieved for configurations with eight nodes and sixty-four internal cells, though this speedup decreased when expanding from eight to sixty-four nodes due to the adjustment from eight to sixteen CPU ranks for larger cases. The data indicates a notable preference for 3D over 2D cases in terms of speedup, suggesting that GPUs are more effectively utilized in 3D scenarios. This effectiveness is linked to the degree of data parallelism within a case. GPU utilization was high, around 90-95% for 3D cases, and even reached 97% for the sixty-four internal cell scenario, demonstrating that GPUs were nearly fully utilized. As mentioned earlier, this does not mean that the GPU is more performant at higher utilization [26], it simply indicates that the 3D cases are more effectively hiding the memory latencies as compared to the 2D cases.

4.1.2 Increasing Internal Cells, NodeRatio 1 for GPU and suitable ratio as per MPI ranks for CPU

The section discusses the second method for increasing the total number of cells, where internal cells are increased while maintaining a constant number of nodes. This is further detailed in the cases listed in tables 4.3 and 4.4. It is noted that, with the exception of the first case in both the 2D and 3D CPU versions, where 4 and 8 ranks are used respectively, the number of MPI ranks is consistently held at 16 for CPU and 1 for GPU across the other cases. The reason for the number of ranks being 16 is because it gave the least compute loop time for the CPU case for the chosen number of nodes of 16. Furthermore, the number of iterations is kept constant at 10000 for all cases.

Table 4.3: Speedup (CPU time / GPU time) for increasing total cells and the internal cells for dimension=2. The node ratio is the number of nodes in the x-direction **X** number of nodes in the y-direction. The IC denotes the number of internal cells in a block for block based multiresolution.

	CPU		GPU		
Total Cells	**IC**	**NodeRatio**	**IC**	**NodeRatio**	**Speedup**
1024	16	2x2	32	1x1	0.3578
4096	16	4x4	64	1x1	0.3097
16384	32	4x4	128	1x1	0.3963
65536	64	4x4	256	1x1	0.3844
262144	128	4x4	512	1x1	0.4441
1048576	256	4x4	1024	1x1	0.9194

As can be seen in figure 4.5, there is a clear increase in speedup when comparing cases with a higher number of internal cells, showing that speedup increases as the number of internal cells increases. This indicates that there weren't sufficient floating point operations per memory access for lower internal cells, and the CPU with its vectorized code was surpassing the GPU. However, the 2D cases in GPU are still not as fast as the CPU when compared for 16 MPI ranks.

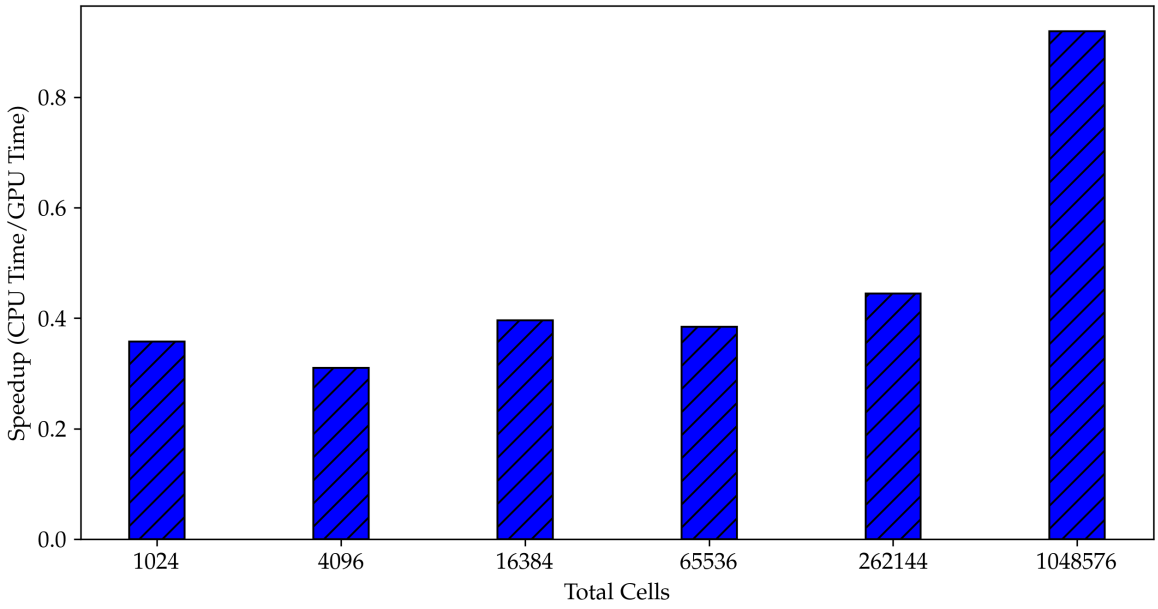


Figure 4.5: Speedup (CPU time / GPU time) for increasing total cells and increasing internal cells for dimension=2. The 2x2 case (1024 total cells) is run with 4 MPI ranks, rest of the cases are run with 16 MPI ranks.

The highest speedup of approximately 0.91 can be seen in the case of 1048576 total cells with internal cells of 1024 for GPU and 256 for CPU.

Table 4.4: Speedup (CPU time / GPU time) for increasing total cells and the internal cells for dimension=3. The node Ratio is given as number of nodes in x direction **X** number of nodes in y direction **X** number of nodes in z direction. The IC denotes the number of internal cells in a block for block based multiresolution.

	CPU		GPU		
Total Cells	**IC**	**NodeRatio**	**IC**	**NodeRatio**	**Speedup**
32768	16	2x2x2	32	1x1x1	2.9784
262144	16	4x4x4	64	1x1x1	1.8332
2097152	32	4x4x4	128	1x1x1	2.6996
16777216	64	4x4x4	256	1x1x1	3.4940

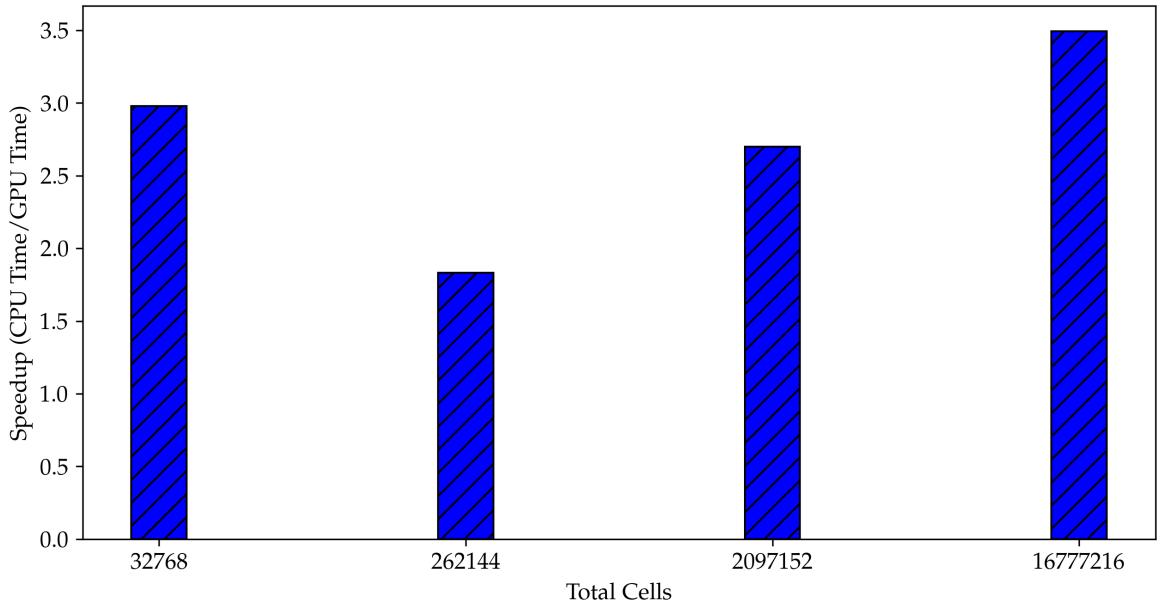


Figure 4.6: Speedup (CPU time / GPU time) for increasing total cells and increasing internal cells for dimension=3. The 2x2x2 case is run with 8 MPI ranks, rest of the cases are run with 16 MPI ranks.

When comparing 3D cases, it's observed that the GPU version is significantly faster, with an average speedup of 3x as can be seen in figure 4.6. Also, there's a slight dip in speedup from 2.9x to 1.8x when the total number of cells increases from 32768 to 262144. This reduction in performance efficiency is attributed to the CPU ranks growing from 8 to 16. Despite this, the speedup trend reverses and starts increasing as the total cells continue to rise, underscoring the fact that the GPU scales more effectively for 3D cases. Additionally, it was observed that GPU utilization is more efficient for larger problem sizes indicating that having a bigger problem effectively hides the memory latencies. The highest speedup of 3.49x can be seen for the case with the most number of total cells (16777216) with internal cells 256 for GPU and 64 for the CPU.

4.1.3 Total Cells constant

In this section, we examine the scenario where the total number of cells is kept constant while exploring the impact of increasing the number of nodes. The base case for this study involves the RTI 3D with level 0, which comprises a total of 2097152 cells. As the number of nodes increases from 1 to 512,

there is a reduction in the cells from 128 to 16 as can be seen in table 4.5. Throughout this investigation, the number of iterations remains constant at 10000 for all cases.

Table 4.5: Constant number of total cells while varying node ratio and internal cells for dimension=3. The Node Ratio is given as number of nodes in x direction **X** number of nodes in y direction **X** number of nodes in z direction.

Total Cells	IC	NodeRatio
2097152	128	1x1x1
2097152	64	2x2x2
2097152	32	4x4x4
2097152	16	8x8x8

As can be seen in 4.7, the time taken for the internal cells 16 case is almost twice as much as the time taken for the base case containing 128 internal cells and 1 node. This demonstrates the fact that in the current implementation, the loop over internal cells is parallelised and the loop over nodes is sequential. Also, higher number of internal cells entail an increased data locality, which is beneficial for the GPU. Due to the high number of nodes in the IC16 case (512) as compared to the base case containing 128 internal cells and 1 node, the GPU is not able to utilize data parallelism effectively, leading to a longer processing time.

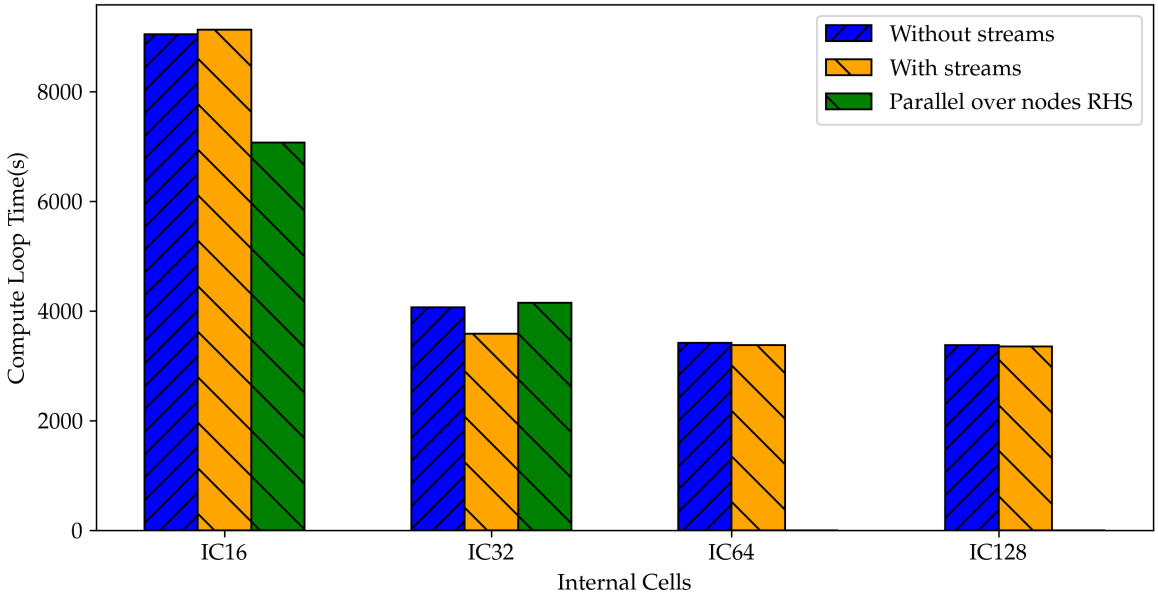


Figure 4.7: Comparison of compute loop times with constant total cells, varying node ratio, and varying internal cells for dimension=3. The total cells are kept constant at 2097152.

One of the possible ways to mitigate this, as discussed in 3.6, is to use streams to overlap the computation over nodes. This strategy has shown some benefits in the case of 32 internal cells, where the time taken is reduced by 15%. However, this approach does not offer any advantage in the case of 16 internal cells. To investigate this issue further, the profiling outputs can be referred to, as illustrated in figure 4.8.

(a) Profiling Output for Internal Cells 16.

(b) Profiling Output for Internal Cells 32.

(c) Profiling Output for Internal Cells 64.

(d) Profiling Output for Internal Cells 128.

Figure 4.8: Profiling output for constant total cells, varying node ratio, and varying internal cells for streams version with two cuda streams. The two streams can be seen as Stream 14 and Stream 15. The red box highlights the regions of overlap of two streams

As seen in figure 4.8a, the analysis of the case with 16 internal cells reveals a very slight overlap, indicating that there isn't enough computation in the kernel for it to last longer for overlap to occur. Conversely, in the scenario depicted in figure 4.8b, which comprises 32 internal cells, there is a noticeable improvement in overlap. The kernel's extended duration leads to a more significant overlap, resulting in a slight increase in performance, as observed in 4.7. However, the situations described in figures 4.8c and 4.8d might appear to involve overlap, but the kernels are too extensive for concurrent execution. Consequently, one stream remains idle while another is executing a kernel, explaining the lack of observed benefits for 64 and 128 internal cells. It's crucial to acknowledge that, although the streams were launched sequentially, achieving actual benefit would require launching the streams using threads concurrently. This adjustment necessitates making the `UpdateFluxes` thread-safe, a modification not pursued in the current work due to its complexities.

Another way to enhance performance was to parallelize over nodes, as described in earlier optimizations. This approach resulted in a significant reduction of about 23% in compute loop time compared to the base GPU version, as illustrated in Figure 4.7. However, this method is not scalable with an increase in internal cells because it necessitates the pre-allocation of all buffers. This high memory footprint meant that cases with 64 and 128 internal cells were unable to run on the current GPU setup.

4.1.4 Effect of Remeshing with constant effective resolution

In this section, the effect of remeshing in the CUDA backend is assessed. During the remeshing process, entire blocks are refined or coarsened based on certain criteria. This might lead to expensive allocations and deallocations of memory, since the block allocation is still currently carried out on the CPU.

Table 4.6: Node size and node ratio for four different refinement levels to keep same effective resolution for dimension=2. The node ratio is given as number of nodes in x direction **X** number of nodes in y direction.

Level	NodeRatio	NodeSize
0	16x64	0.015625
1	8x32	0.03125
2	4x16	0.0625
3	2x8	0.125
4	1x4	0.25

In this study, the cell size and the effective resolution at maximum level is kept constant at

9.765625000e-04 and 256 x 1024 IC respectively. The node size and subsequently nodeRatio is adjusted accordingly to get the desired resolution as shown in table 4.6. The end time for these cases is set at 1.0 sec. The sample outputs of the RTI case are shown for different refinement levels in figure 4.10.

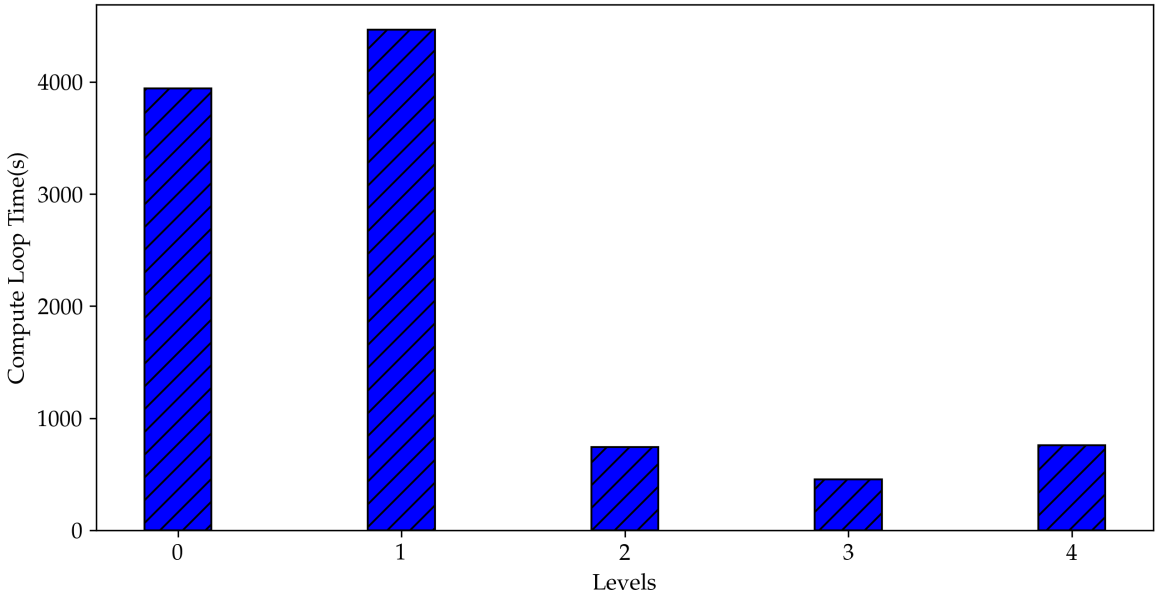
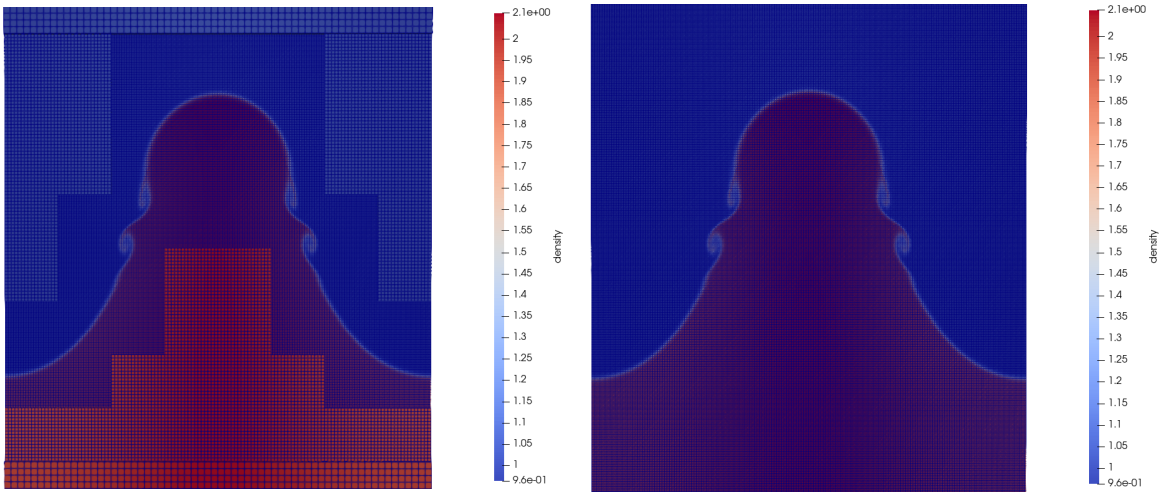


Figure 4.9: Comparison of compute loop times to see the effect of remeshing for dimension=2 for four levels of refinement.

As can be seen in 4.9, the time taken for 3 levels of refinement is actually the lowest. This demonstrates that the time taken for allocation and deallocation is not as expensive as having a greater number of blocks themselves. The bottleneck in the process is still the looping over nodes, which happens in a sequential manner.



(a) Density contour for RTI case with multiresolution level 4

(b) Density contour for RTI case with multiresolution level 0

Figure 4.10: Density contour for RTI case for refinement levels 4 and 0. It can be seen that the having 4 levels of refinement allows for coarser resolution blocks in some regions and finer resolution blocks in some.

4.2 Function wise profiling

This study aimed to assess the impact of integrating Kokkos into different parts of the computational loop, comparing it with the baseline ALPACA version to determine any performance enhancements or drawbacks. Specifically, the RTI case for 3-dimensions that employed two refinement levels was used to test this. For this analysis, a configuration of 16 internal cells was adopted.

4.2.1 CUDA backend profiling

In Figure 4.11, we delve into a function-wise profiling comparison, contrasting the performance metrics of a CPU equipped with 16 cores against that of a GPU. The insights from this graphical representation shed light on the varying degrees of efficiency gains achieved by incorporating the Kokkos CUDA backend. One of the standout observations from this comparison is the distinct performance enhancement seen in the `AverageMaterial` function when executed on the GPU. This function, in particular, benefits remarkably from the CUDA backend. On the other hand, the IntegrateMaterial function does not exhibit a similar level of improvement, indicating a minimal advantage when transitioned to the GPU. This discrepancy highlights that not all functions benefit uniformly from GPU acceleration, and some may remain relatively unaffected. This profiling analysis serves as a crucial tool for identifying potential bottlenecks within the computational framework.

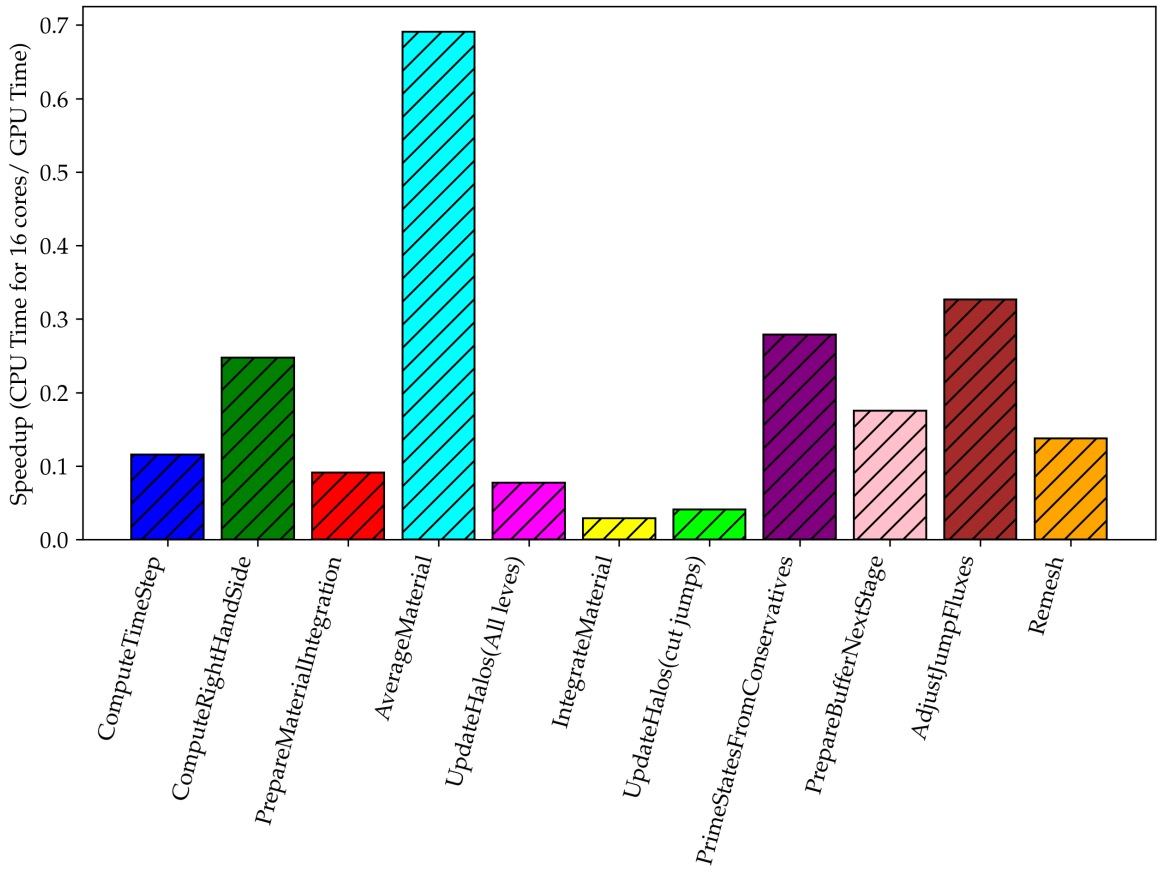


Figure 4.11: Function wise profiling for CPU 16 cores and CUDA backend.

4.2.2 Serial Backend Profiling

The primary goal of transitioning to hardware-agnostic frameworks such as Kokkos lies in their support for a variety of backends. Within this context, a comparative analysis between the serial backend

and the original ALPACA codebase reveals insightful observations. The serial backend, which essentially forgoes any form of on-node parallelism, is theoretically expected to perform on par with the original ALPACA codebase. A visual representation detailed in the figure 4.12 showcases function-wise profiling for both CPU (1 core) and the Serial Backend, indicating a similarity in trends between CPU vs. GPU profiling noted in previous sections.

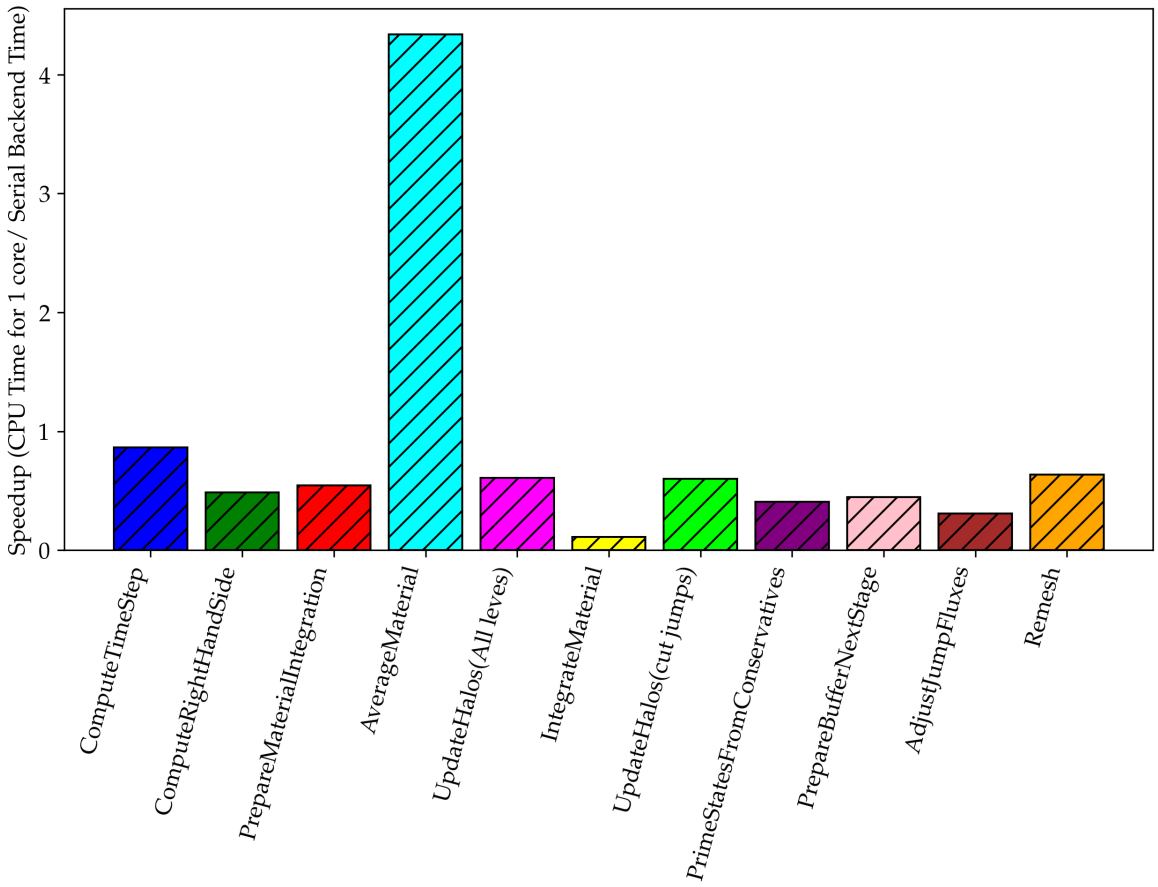


Figure 4.12: Function wise profiling for CPU 1 core and Kokkos Serial Backend

Remarkably, the `AverageMaterial` function witnesses a nearly fourfold speed increase in the serial backend compared to its CPU counterpart. However, this boost in speed appears to be an isolated case when considering the broader scope of the Kokkos version, which, in contrast, exhibits a notable decrease in performance across other functions. Specifically, these functions experience a slowdown of approximately 2x, and in certain instances like `IntegrateMaterial`, the performance degradation is even more pronounced. The disparity in performance between the `AverageMaterial` function and other functions within the Kokkos version suggests that there may be underlying issues affecting the overall efficiency of the serial backend of the codebase. One possible explanation for this inconsistency is the absence of auto-vectorization in the Kokkos serial backend version. Auto-vectorization is a powerful feature that allows compilers to automatically optimize code by taking advantage of the vector processing capabilities of modern CPUs, thereby significantly speeding up computations. The lack of auto-vectorization in the Kokkos version could be a critical factor contributing to the observed slowdown in performance. It is evident from this experiment that there is a clear need for further optimization efforts across the Kokkos version focusing on the serial backend.

4.3 OpenMP

In assessing the performance of the OpenMP backend of Kokkos, we utilized the base RTI case featuring 2 levels of refinement, 16 internal cells, and 3 dimensions. It's crucial to highlight that no modifications were made to the code to ensure OpenMP compatibility; instead, the same version linked to Kokkos' OpenMP backend was directly employed. This approach underscores the advantage of leveraging Kokkos as a hardware abstraction layer, allowing for the seamless use of the same code across different backends.

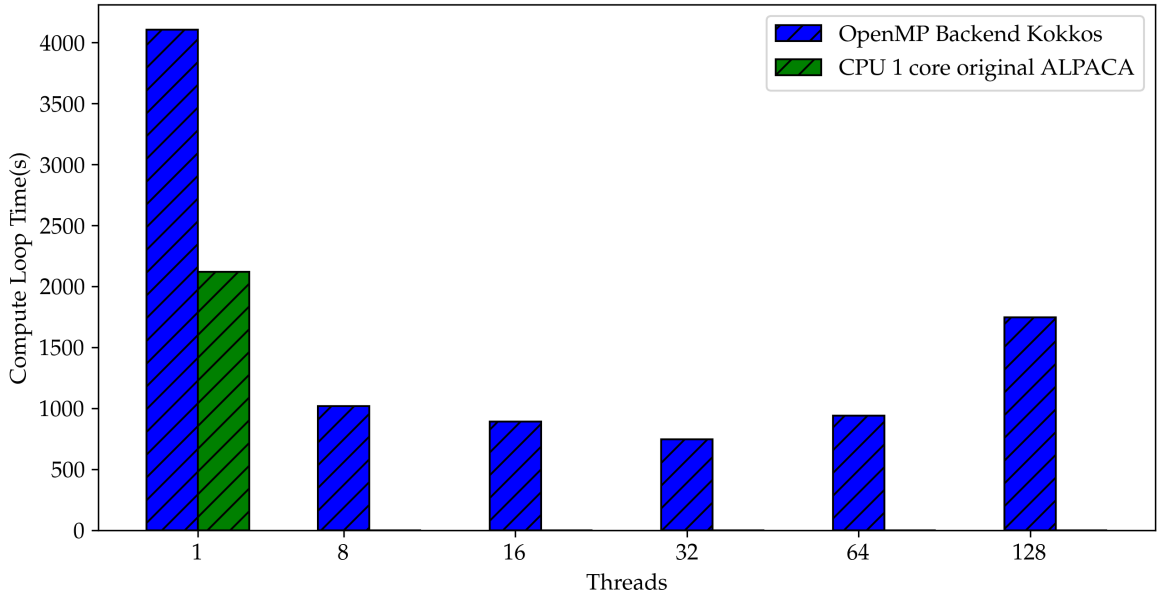


Figure 4.13: Compute loop times for OpenMP backend for increasing number of threads

As depicted in the figure 4.13, the performance of the OpenMP backend with a single thread mirrors that of the serial backend and falls short of the base ALPACA version's single-thread performance. Nevertheless, a notable improvement is observed as the number of threads increases, with the optimal performance being achieved at 32 threads. Beyond this point, performance diminishes, likely due to the costs associated with spawning additional threads and the overhead of managing them.

Chapter 5

Summary & Future work

The thesis aimed to explore and assess the integration of performance portability within ALPACA. This work sets the foundation for future advancements in the field by incorporating Kokkos to facilitate this integration. The focus was mainly on porting ALPACA's single-phase and multiresolution aspects to Kokkos. An essential prerequisite for this successful integration involved replacing ALPACA's `FieldBuffer` with `Kokkos::Views`. Although this task was partially accomplished, challenges remain, particularly in adapting segments of the code that depend on `std::array` for fields output and multi-phase flows, which also require conversion to `Kokkos::Views`.

Subsequently, modifications were made to the code's structure by replacing traditional for-loops over the internal cells within a block with Kokkos parallel dispatch. Modifications were also necessary to ensure compatibility with device execution spaces. This included enabling the copy functionality for certain objects, substituting run-time polymorphism in the EOS with a singular EOS choice, and applying `KOKKOS_INLINE_FUNCTION` to functions needing device execution. These changes collectively contributed to the observed performance improvements, marking a promising first step towards fully porting a substantial codebase like ALPACA to Kokkos. Notably, the GPU performance, particularly with the CUDA backend on NVIDIA GPUs, showed significant gains in 3D simulations, although it underperformed in 2D scenarios.

The pursuit of performance portability involved testing the code on different backends. These included CUDA, OpenMP, and Serial, where the code worked correctly without needing any changes. The OpenMP backend notably outperformed the single-threaded CPU version, indicating an improvement. However, the serial backend's performance was slower compared to the original CPU version, highlighting areas for further optimization and development. This journey toward integrating performance portability into ALPACA with Kokkos reveals the complexities and the potential of leveraging such tools to enhance the ALPACA's performance and efficiency.

5.1 Future Work

The first aspect regarding future enhancements would be to extend the coverage of Kokkos implementation into multi-phase part of the codebase. This would enable then to fully leverage the Kokkos performance portability with its `DualView` functionality. Also, the serial backend's performance needs to be improved to match the original CPU version. Many sections of the code still depend on CPU implementation and necessitate using `std::array` in field buffers. A significant improvement can be achieved by replacing these with `Kokkos::Views` wherever possible. In instances where CPU access remains essential, `Kokkos::DualView` serves as a viable solution to access data across both CPU and GPU spaces, offering an opportunity to streamline processes and enhance efficiency.

Several strategies have been identified to optimize the codebase further. Optimizing kernels to reduce memory allocations inside the kernel, for instance, in the `FluxSplitting::ComputeFluxes` function, and utilizing memory more efficiently inside the kernel can be a crucial area of focus. This might

involve overloading the stencils `constexpr` to work compatibly with `Kokkos::Views`. Additionally, employing shared memory, such as in `ComputeRoeEigenDecomposition`, can significantly reduce global memory access and thus improve performance. Moreover, tiled traversal of the grid can improve cache utilization and overall performance. Streams could be used to overlap calculation over dimensions to extract more parallelism. A more involved optimization would be parallelizing over nodes.

Further enhancements include extending to multi-GPU implementation using GPU-aware MPI, allowing direct communication of GPU buffers using `Kokkos::Views` with the existing MPI infrastructure. This approach necessitates some adjustments in areas like `HaloManager` and `JumpFluxAdjustment`, where data needs to be packed in contiguous memory on the GPU before being sent. Making the EOS generic through CRTP and `std::variant`, identifying and optimizing the bottleneck for the Serial backend, and optimizing OpenMP are other avenues for improving the overall performance. Additionally, efforts to make functors inexpensive to construct and to reduce memory copies can significantly contribute to the overall performance and efficiency of a performance portable ALPACA.

Appendix A

Code Excerpts

```
void stencil3d_sycl(sycl::queue& queue, sycl::buffer<double, 3>& input,
                    sycl::buffer<double, 3>& output, int size)
{
    queue.submit([&] (cl::sycl::handler& cgh)
    {
        sycl::accessor input_acc(input, cgh, sycl::read_only);
        sycl::accessor output_acc(output, cgh, sycl::write_only);

        cgh.parallel_for<Stencil3D>(
        cl::sycl::range<3>(size-2,size-2,size-2),
        [=](cl::sycl::id<3> idx){
        int i = idx[0] + 1;
        int j = idx[1] + 1;
        int k = idx[2] + 1;

        output_acc[i][j][k] = (input_acc[i][j][k]+input_acc[i-1][j][k]
                             + input_acc[i+1][j][k]+input_acc[i][j-1][k]
                             + input_acc[i][j+1][k]+input_acc[i][j][k-1]
                             + input_acc[i][j][k+1])/8.0;
        });
    });
    queue.wait_and_throw(); //wait for the work to finish
};
```

Listing A.1: Stencil operation on 3D data(SYCL version). The queue is similar to parallel_for of Kokkos which maps the threads to the loop indices.

```
void stencil3d_thrust(const thrust::device_vector<double>& d_input,
                      thrust::device_vector<double>& d_output, int size)
{
    StencilFunctor stencil_functor(
                          thrust::raw_pointer_cast(d_input.data()),
                          size);

    thrust::transform(thrust::counting_iterator<int>(0),
                      thrust::counting_iterator<int>(size*size*size),
                      d_output.begin(), stencil_functor);
}
```

Listing A.2: Thrust transform function, with iterator over the data size. The transform function calls the functor over the iterator range

```
struct StencilFunctor
{
    const double* input;
```

```
    const int size;

    StencilFunctor(const double* _input, const int _size):input(_input),
                                                          size(_size) {}

    __host__ __device__
    double operator()(const int idx) const
    {
        int tid_x = idx % size;
        int tid_y = (idx / size) % size;
        int tid_z = idx / (size * size);

        if (tid_x < size - 1 && tid_y < size - 1 && tid_z < size - 1 &&
            tid_x > 0 && tid_y > 0 && tid_z > 0)
        {
            int tid = tid_z * size * size + tid_y * size + tid_x;
            return (input[tid] + input[tid + 1] + input[tid - 1] +
                    input[tid + size] + input[tid - size] +
                    input[tid + size * size] + input[tid - size * size])
                    / 8.0;
        }
        else
            return 0;
    }
};
```

Listing A.3: Contents of the thrust functor. This is similar to the functor of Kokkos which overloads operator()

Bibliography

[1] Alpay, A., Soproni, B., Wünsche, H., and Heuveline, V. “Exploring the possibility of a hipSYCL-based implementation of oneAPI”. In: *Proceedings of the 10th International Workshop on OpenCL*. IWOCL ’22. , Bristol, United Kingdom, United Kingdom, Association for Computing Machinery, 2022. ISBN: 9781450396585. DOI: 10.1145/3529538.3530005. URL: https://doi.org/10.1145/3529538.3530005.

[2] *AMD EPYC 7002 Series Datasheet*. AMD Product Literature. https://www.amd.com/content/dam/amd/en-us/Documents/products/epyc/amd-epyc-7002-series-datasheet.pdf.

[3] *AMD Processor Tech Docs*. AMD Documentation. https://www.amd.com/content/dam/amd/en-us/Documents/processor_tech_docs/24592.pdf.

[4] Artigues, V., Kormann, K., Rampp, M., and Reuter, K. “Evaluation of performance portability frameworks for the implementation of a particle-in-cell code”. In: *arXiv e-prints* (Nov. 2019), arXiv:1911.08394.

[5] Beckingsale, D. A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., Pearce, O., Robinson, P., Ryujin, B. S., and Scogland, T. R. “RAJA: Portable Performance for Large-Scale Scientific Applications”. In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2019, pp. 71–81. DOI: 10.1109/P3HPC49587.2019.00012. URL: https://github.com/LLNL/RAJA.

[6] Bell, N. and Hoberock, J. “Chapter 26 - Thrust: A Productivity-Oriented Library for CUDA”. In: *GPU Computing Gems Jade Edition*. Ed. by Hwu, W.-m. W. Applications of GPU Computing Series. Boston: Morgan Kaufmann, 2012, pp. 359–371. ISBN: 978-0-12-385963-1. DOI: https://doi.org/10.1016/B978-0-12-385963-1.00026-5. URL: https://www.sciencedirect.com/science/article/pii/B9780123859631000265.

[7] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. “Rodinia: A benchmark suite for heterogeneous computing”. In: *2009 IEEE International Symposium on Workload Characterization (IISWC)* (2009), pp. 44–54. URL: https://api.semanticscholar.org/CorpusID:206915521.

[8] Edwards, H. C., Trott, C. R., and Sunderland, D. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001257.

[9] *ExaMiniMD*. GitHub repository. https://github.com/ECP-copa/ExaMiniMD.

[10] Fedkiw, R. P., Aslam, T., Merriman, B., and Osher, S. “A Non-oscillatory Eulerian Approach to Interfaces in Multimaterial Flows (the Ghost Fluid Method)”. In: *Journal of Computational Physics* 152.2 (1999), pp. 457–492. ISSN: 0021-9991. DOI: https://doi.org/10.1006/jcph.1999.6236. URL: https://www.sciencedirect.com/science/article/pii/S0021999199962368.

[11] Grete, P., Dolence, J. C., Miller, J. M., Brown, J., Ryan, B., Gaspar, A., Glines, F., Swaminarayan, S., Lippuner, J., Solomon, C. J., Shipman, G., Junghans, C., Holladay, D., Stone, J. M., and Roberts, L. F. "Parthenon–a performance portable block-structured adaptive mesh refinement framework". In: *Journal Title* XX.X (2022), pp. 1–19.

[12] Grete, P., Glines, F. W., and O'Shea, B. W. "K-Athena: a performance portable structured grid finite volume magnetohydrodynamics code". In: *IEEE Transactions on Parallel and Distributed Systems* (2022). Accepted for publication.

[13] Harlow, F. and Amsden, A. *Fluid Dynamics. A LASL Monograph.* Tech. rep. Los Alamos, NM (United States): Los Alamos National Lab. (LANL), 1971.

[14] Harten, A. "Multiresolution algorithms for the numerical solution of hyperbolic conservation laws". In: *Communications on Pure and Applied Mathematics* 48.12 (1995), pp. 1305–1342. DOI: https://doi.org/10.1002/cpa.3160481201. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpa.3160481201. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpa.3160481201.

[15] Herdman, J. A., Gaudin, W. P., Perks, O., Beckingsale, D. A., Mallinson, A. C., and Jarvis, S. A. "Achieving Portability and Performance through OpenACC". In: *2014 First Workshop on Accelerator Programming using Directives*. 2014, pp. 19–26. DOI: 10.1109/WACCPD.2014.10.

[16] Heroux, M. A., Doerfler, D. W., Crozier, P. S., Thornquist, H. K., Numrich, R. W., Williams, A. B., Edwards, H. C., Keiter, E. R., Rajan, M., and Willenbring, J. M. "Improving performance via mini-applications". In: (Sept. 2009).

[17] Hoppe, N., Adami, S., and Adams, N. A. "A modular massively parallel computing environment for three-dimensional multiresolution simulations of compressible flows". In: (2020). arXiv: 2012.04385 [physics.comp-ph].

[18] Hoppe, N., Winter, J. M., Adami, S., and Adams, N. A. "ALPACA - a level-set based sharp-interface multiresolution solver for conservation laws". In: *Computer Physics Communications* 272 (2022), p. 108246. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2021.108246. URL: https://www.sciencedirect.com/science/article/pii/S0010465521003581.

[19] Howard, M., Fisher, T., Hoemmen, M., Dinzl, D., Overfelt, J., Bradley, A., Kim, K., and Rajamanickam, S. "Employing Multiple Levels of Parallelism for CFD at Large Scales on Next Generation High-Performance Computing Platforms". In: *Tenth International Conference on Computational Fluid Dynamics (ICCFD10)*. ICCFD10-079. Sandia National Laboratories. Barcelona, Spain, July 2018.

[20] *Intel Architecture Performance Primitives*. Intel Whitepaper.

[21] *Intel Xeon Processor E5-2609 v4*. Intel Product Specifications. https://www.intel.com/content/www/us/en/products/sku/92990/intel-xeon-processor-e5-2609-v4-20m-cache-1-70-ghz/specifications.html.

[22] *KokkosTools*. GitHub repository. https://github.com/kokkos/kokkos-tools/tree/develop.

[23] Le Métayer, O. and Saurel, R. "The Noble-Abel Stiffened-Gas equation of state". In: *Physics of Fluids* 28 (2016), p. 046102. DOI: 10.1063/1.4945981. URL: https://hal.archives-ouvertes.fr/hal-01305974.

[24] Memeti, S., Li, L., Pllana, S., Kołodziej, J., and Kessler, C. "Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption". In: New York, NY, USA: Association for Computing Machinery, 2017. ISBN: 9781450351164. DOI: 10.1145/3110355.3110356. URL: https://doi.org/10.1145/3110355.3110356.

[25] Menikoff, R. and Plohr, B. J. "The Riemann problem for fluid flow of real materials". In: *Rev. Mod. Phys.* 61 (1 Jan. 1989), pp. 75–130. DOI: 10.1103/RevModPhys.61.75. URL: https://link.aps.org/doi/10.1103/RevModPhys.61.75.

[26] NVIDIA. “CUDA C++ Best Practices Guide”. In: (). URL: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#occupancy.

[27] *NVIDIA RTX A6000 Datasheet*. NVIDIA Product Literature. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[28] *NVIDIA Turing GPU Architecture*. Whitepaper. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[29] Pennycook, S., Sewall, J. D., Jacobsen, D. W., Deakin, T., and McIntosh-Smith, S. “Navigating Performance, Portability, and Productivity”. In: *Computing in Science and Engineering* 23.05 (2021), pp. 28–38. ISSN: 1558-366X. DOI: 10.1109/MCSE.2021.3097276.

[30] Roe, P. “Approximate Riemann solvers, parameter vectors, and difference schemes”. In: *Journal of Computational Physics* 43.2 (1981), pp. 357–372. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(81)90128-5. URL: https://www.sciencedirect.com/science/article/pii/0021999181901285.

[31] Slattery, S. R. et al. “Cabana: A Performance Portable Library for Particle-Based Simulations”. In: *Journal of Open Source Software* 7.72 (2022), p. 4115. DOI: 10.21105/joss.04115.

[32] *SPEC - Standard Performance Evaluation Corporation*. Website. https://www.spec.org/accel/.

[33] Stojne, J. M., Tomida, K., White, C. J., and Felker, K. G. “The athena++ adaptive mesh refinement framework: Design and magnetohydrodynamic solvers”. In: (2020).

[34] Stone, J. E., Gohara, D., and Shi, G. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: 12.3 (2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[35] Straatsma, T., Antypas, K., and Williams, T. *Exascale Scientific Applications: Scalability and Performance Portability*. 1st. Chapman and Hall/CRC, 2017. DOI: 10.1201/b21930.

[36] Toro, E. F. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*. Berlin Heidelberg: Springer Science & Business Media, 2013. ISBN: 978-3-540-49834-6.

[37] Toro, E. F. “The HLL and HLLC Riemann Solvers”. In: *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 293–311. ISBN: 978-3-662-03490-3. DOI: 10.1007/978-3-662-03490-3_10. URL: https://doi.org/10.1007/978-3-662-03490-3_10.

[38] Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., and Wilke, J. “Kokkos 3: Programming Model Extensions for the Exascale Era”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[39] Zhang, W., Myers, A., Gott, K., Almgren, A., and Bell, J. “AMReX: Block-Structured Adaptive Mesh Refinement for Multiphysics Applications”. In: *Journal Title* XX.X (2020), pp. 1–16.