ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

PAUL SCHERRER INSTITUT
PSI

Rank Independence and Mixed Execution Spaces in IPPL

Master Thesis

in Computational Physics

Department of Physics

ETH Zurich

written by

Alessandro Vinciguerra

supervised by

Dr. A. Adelmann (PSI)

scientific advisors

Dr. M. Frey (Uni. of St Andrews)
Dr. S. Muralikrishnan (FZ Jülich)

September 30, 2023

Abstract

Two new features are implemented in the Independent Parallel Particle Layer library, a C++ library designed for high performance particle-in-cell simulations. First, the implementations of mathematical operations, as well as other relevant algorithms, are rewritten to be independent of the number of dimensions. This increases flexibility and makes the library applicable to a greater range of problems. Second, the components of the library are extended to allow the end user to make use of multiple different backends for parallel computing in the same application, such as OpenMP and CUDA. This makes it possible to more efficiently use all the available resources in a heterogeneous machine. We look at plasma physics phenomena with known analytical properties and verify that the new implementations yield correct results in three dimensions or fewer. We also confirm that the overheads introduced to support rank independence do not affect performance by reproducing existing benchmarking data. Finally, we test the hybrid CPU/GPU execution capabilities and verify that a measurable performance gain is achievable.

Contents

Part I

Introduction

Chapter 1

High Performance Computing

1.1 Supercomputing Clusters

As the available computing power increases, it becomes possible to obtain approximate solutions to larger and more complex physical systems in the same amount of time by using numerical simulations. Modern high performance computing (HPC) uses distributed and shared memory parallelism, dividing the workload among hundreds or thousands of computing nodes.

The cluster architectures often differ greatly from one another; not only can the processors be different, but the graphics cards can come from different manufacturers as well. Thus, portability becomes essential for any software designed for HPC. Libraries should be written in a hardware agnostic manner such that they can be compiled and used on any cluster, regardless of available accelerators.

Flexibility is a useful trait not just for portability across architectures, but also across fields and applications. If a software library allows users to solve a larger variety of problems and provides greater control over how these are solved, it can be more easily reused across multiple projects, reducing development and testing time. A more widely used library also benefits from more attention and thus external testing and quality control. An overarching goal for HPC software development is thus to maximize portability and flexibility.

1.2 Relevant Frameworks

Within a single process on CPUs, parallelism is achieved through the use of threads. *Open Multi-Processing*, or OpenMP for short, is a cross-platform interface for setting up parallel computations. With a compatible compiler, a block of code can be marked for parallel execution using preprocessor macros. An environment variable can be used to set the number of threads that will be spawned for this parallel block.

The original motivation behind dedicated graphics cards was the need for higher performance in matrix operations for visual processing. By the nature of these operations, GPU computations are intrinsically parallel, but the setup for these computations can be specific to each GPU architecture. Thus, graphics cards require their own toolchains to be programmed. In this project we will focus on NVIDIA graphics cards, which are programmed using the *CUDA toolkit*.

Parallel computation makes efficient use of the available resources to reduce total runtime, but the process is still limited by what is available on the device. Instead of building machines and GPUs with massive amounts of internal memory, modern HPC overcomes this limitation

by distributing tasks and data across multiple devices. However, this comes at the cost of introducing communication between the devices.

The *Message Passing Interface* (MPI) is an open standard for message-passing in parallel computing. Multiple copies of the program run simultaneously and exchange data to maintain synchronization and internal consistency. There are two primary implementations of the standard: MPICH and OpenMPI (not to be confused with OpenMP); here we will focus on the latter, which is used in this project. With MPI, each instance of the program is given a unique numerical identifier and can communicate with the other instances, making it possible for each process to have information about the global simulation state despite only storing a part of it. With CUDA-aware MPI, data can be sent directly between GPUs as long as there is a physical connection between them. The available bandwidth for these messages depends on the nature of the communication and on the hardware configuration. In general, the latency incurred by sending data between nodes is detrimental to overall performance and scaling. This is discussed further in Chapter 5.

Chapter 2

Project Scope

2.1 The Independent Parallel Particle Layer

The Independent Parallel Particle Layer (IPPL) is a high performance C++ library [1] that provides data structures representing different components relevant for physics simulations, such as particles and fields, as well as a selection of numerical solvers. These components can then be used to simulate different systems. In this project, we focus mainly on the applications in ALPINE [2], a set of small plasma physics simulations designed to showcase the features of IPPL.

Given the nature of modern HPC, performance is not the only important feature for IPPL: portability is also essential to ensure that the library can be used on a variety of architectures and HPC systems. Kokkos [3] is a performance portability ecosystem that enables the writing of hardware agnostic, high performance C++ programs. Several backends for parallel execution are supported, but their specific properties are hidden behind an abstraction layer. IPPL delegates all parallel dispatch and device-memory[1] allocations to Kokkos and can therefore be used on any platform supported by Kokkos, including, in particular, OpenMP and CUDA, the two platforms used in this project.

IPPL 1.0 was first released in 2009 [4]. The library has been effectively rewritten to leverage newer language features and frameworks, most notably C++20 and Kokkos. IPPL is still undergoing active development and does not yet include all the useful features of the original release. In general, this means that certain features and implementation choices are restricted to systems fitting certain assumptions. These will be discussed where relevant.

The state of the library prior to the beginning of this project is tagged as IPPL 2.4.0. We note that there are thus two distinct reference points to which we will allude in this project. The 1.0 release acts as a target, showcasing the features that were once supported and may be supported in the future. We refer to it when considering limitations of IPPL 2.4.0 stemming from the lack of a particular feature from IPPL 1.0. In contrast, the 2.4.0 release is the status quo on which this project should improve. We refer to it when discussing changes made to the library design in order to achieve the goals of the project.

[1] In this document, we sometimes use "device" to generically refer to any hardware component capable of performing computations, whether this is a CPU core, a GPU, or an entire compute node. However, "device memory" is a common term for GPU memory. The CPU-accessible memory is referred to as host memory.

2.2 Project Goals

The scope of this project can be divided into two independent parts, which we will discuss separately.

2.2.1 Rank Independence

The first goal of this project is to increase the breadth of problems for which IPPL can be used. Up to the 2.4.0 release, most algorithms present in the library were only compatible with problems in three dimensions. This imposes a restriction on the kind of problems that IPPL can be used to solve. We note that it is possible to artificially create a one- or two-dimensional problem by creating a mesh with a single cell along the undesired axes, but this leads to a large quantity of redundant vector elements: if a vector $v \in \mathbb{R}^3$ is stored for a 1D problem, then v_y, v_z are both effectively unused entries. Since GPU memory is at a premium, it should be used as efficiently as possible.

This restriction on the problem dimensionality stems from the implementation of the algorithms, which currently assumes that, for example, fields are three-dimensional arrays and therefore have elements identified by exactly three indices. To achieve *dimensionality independence* (or *rank* independence), the algorithms have to be rewritten to avoid this assumption and instead accept a variable number of indices for identifying field elements.

We note that Kokkos currently only supports parallelism for arrays with up to six dimensions. Thus, a rank independent IPPL would support problems from 1D up to 6D.

2.2.2 Mixed Execution Spaces

The second goal is to add support for *mixed execution spaces*. In its development up to this point, IPPL has been compiled to support either CUDA or OpenMP as *backends* for parallel computation, but not both at the same time. Memory allocations for structures used in parallel computations are delegated to Kokkos, which chooses GPU memory by default if it is available. All parallel computations are then performed using a suitable accelerator based on where the data is stored. However, Kokkos can be compiled to support multiple accelerators and allows the user to choose whether a given data structure should be stored on the host or on a GPU. This makes it possible to allocate data on both the host and the GPU; these can then work in parallel on separate computations.

We note here that, for a GPU build, while the computations may be occurring exclusively on the device, the main thread of execution continues to be on the CPU. This means that the main thread launches tasks to be executed on the device (known as *kernels*) but otherwise does nothing during the actual computations. Instead, the main execution thread must wait until the results of the computation are available so that relevant data from the simulation can be saved to disk. This idle time represents wasted CPU time. By saving data on both the host and the device and enabling both OpenMP and CUDA in the same build, this idle time can be reduced: a program can be designed to split the workload between the host and device such that both are busy for approximately equal durations.

2.3 The Particle-in-Cell Scheme

The particle-in-cell (PIC) scheme is a popular approach used in various types of simulations. With this scheme, each time step of the simulation is broken down into four steps, which are

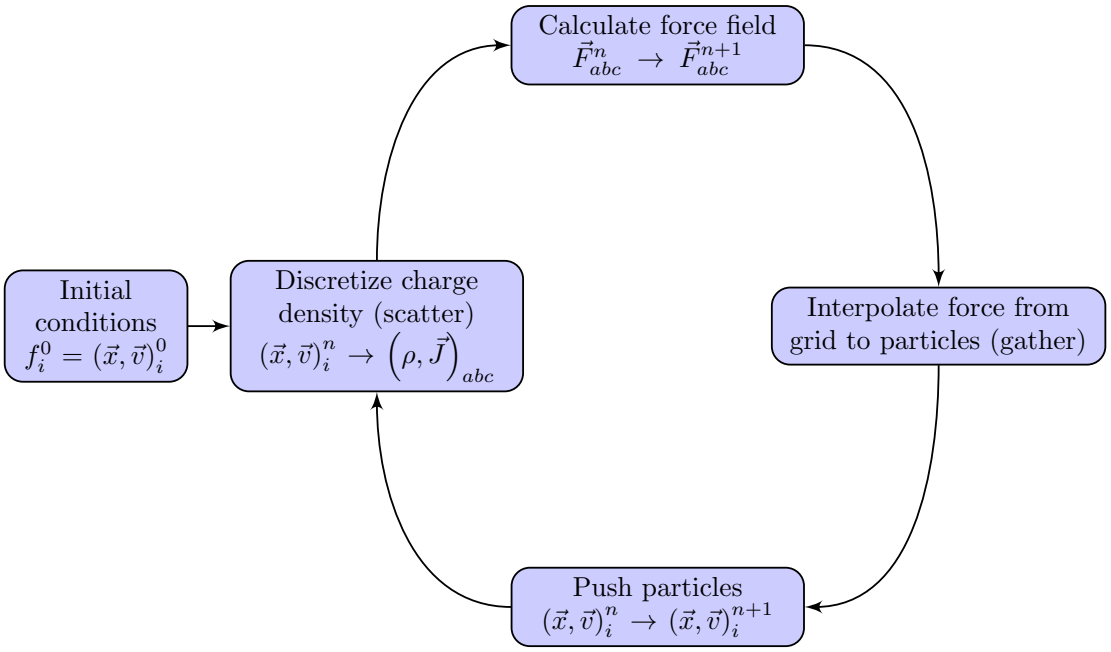

Figure 2.1: The particle-in-cell loop.

shown in Figure 2.1. Given a set of initial conditions $(\vec{x}_0, \vec{v}_0)$ at an initial time t_0, the PIC loop allows us to evolve the system and predict its state at some later time t_f.

In the context of this project, we will use the PIC scheme to implement simulations of plasma physics phenomena. However, the PIC scheme is not limited to plasma physics, nor is IPPL limited to simulations using the PIC scheme. We focus on plasma physics problems in this project and thus restrict the simulations to a physically meaningful number of dimensions, but note that a rank independent version of IPPL could be used for problems in higher dimensions as well.

The first step is an interpolation from the continuous coordinate space $\mathbb{R}^n$ onto a discrete grid; we can compute the charge and current densities $\rho, \vec{J}$. This is known as the "scatter" step. A field solver then computes the relevant parameters for the equations of motion; in the electrostatic case, we only need ρ. We compute the electric potential ϕ by solving the Poisson equation $-\nabla^2\phi = \rho$ and use this to determine the electric field $\vec{E} = -\nabla\phi$. The obtained parameters are again discretized fields, so we need to perform another interpolation to determine the value of these parameters at the particles' exact locations; this is known as the "gather" step. For the electrostatic case, the Lorentz force reduces to $\ddot{\vec{x}} = \frac{q}{m}\vec{E}(\vec{x})$ and we can employ any appropriate integrator to evolve the system.

Part II

Rank Independence

Chapter 3

Implementation

3.1 Essential Concepts

We begin with some notation and terminology to ensure that discussions about IPPL in this document are clear.

3.1.1 Domain Discretization

In numerical simulations, it is important to distinguish between the *physical domain* of the problem and the *discretized domain* in the simulation. Consider physical phenomena defined in n spatial dimensions. The physical domain is a continuous subset of Euclidean space $\mathbb{D} \subset \mathbb{R}^n$ in which the phenomena of interest occur. We generally want to consider the values of scalar and vector fields in this domain. To represent a field with unknown analytical form in a finite amount of memory, the domain must be discretized, i.e. reduced to a finite, discrete set of points. This set forms a *mesh* $\mathbb{M} \subset \mathbb{R}^n, |\mathbb{M}| < \infty$.

The current version of IPPL only supports uniform meshes. This means the points are uniformly spaced in each direction, although the spacing does not have to be consistent across all axes. The points of the mesh thus form a lattice with $N_d \in \mathbb{N}$ unique coordinate values along each axis d. We refer to the choice of these values $\vec{N} = (N_1, N_2, \ldots) \in \mathbb{N}^n$ as the *mesh refinement*. This restriction on mesh topology also means that the physical domain is necessarily a hyperrectangle (or at least parameterizable as such). If the origin of the domain is $\vec{r_0} = (O_1, O_2, \ldots, O_n)$, we can write the domain as a product of intervals $\mathbb{D} = \prod_{i=1}^{n} [O_i, O_i + L_i]$. The *mesh spacing* given by $\vec{h} = (L_1/N_1, L_2/N_2, \ldots) \in \mathbb{R}^n$ gives the distances between orthogonally adjacent points in the mesh.

For a problem in n dimensions, the values of a field F in a given domain can be stored in a rank n tensor[1], which we will write as $F \in \mathbb{R}^{\otimes n}$. In particular, there is an isomorphism $\mathbb{D} \to \mathbb{N}^n$ between the physical coordinate space and the *index space* of tuples uniquely identifying the tensor elements. We can use this transformation to easily compute field values based on the indices of the elements being computed. It should be noted that the exact transformation depends on the properties of the mesh; we relegate further details to Section 10.1, as they are not relevant to the present discussion.

[1] In this context we use "tensor" as the generic term for a higher dimensional array. A matrix would thus be a rank 2 tensor and a field in 4D can be represented using a rank 4 tensor. This is different from the meanings of the term common to mathematics (i.e. a multilinear map) or physics (e.g. the electromagnetic field tensor $F_{\mu\nu}$).

3.1.2 Domain Decomposition

Modern HPC is distributed, meaning that it involves sharing the workload among several devices instead of using just one device. In IPPL, fields can be decomposed such that each device utilized by the program only stores a part of the data. The full physical domain is divided among the devices, reducing the memory footprint on each individual device. This makes it possible to simulate much larger systems than would otherwise be possible using the memory available on a single device, at the cost of performance overheads due to data transfers between the devices, or *ranks*[2]. Figure 3.1 shows an example domain decomposition across 9 MPI ranks.

Each rank r stores a subdomain $\mathbb{D}_r \subset \mathbb{D}$ of the full domain in the simulation. We say that two ranks r, r' have *neighboring subdomains* if $\mathbb{I}_{rr'} := \mathbb{D}_r \cap \mathbb{D}_{r'} \neq \emptyset$. Two ranks whose domains share a single point are known as *vertex neighbors*. If $\dim \mathbb{I}_{rr'} = 2$, they are *edge neighbors*. This pattern continues for intersections with higher dimension. For example, in Figure 3.1, ranks 2 and 5 are edge neighbors whereas ranks 4 and 8 are vertex neighbors.

Inter-rank communication is necessary for stencil operations; on the boundary of each rank's local subdomain, the value of the stencil may depend on data in a neighboring subdomain, which is not stored on the same device. As an example, consider a Laplacian stencil evaluated at a point on the edge of a rank's local subdomain. This computation is dependent on the value stored on another rank. To ensure that the relevant data is easily accessible to each rank, the subdomains are padded with a layer of *halo cells* (or *ghost* cells). The values in the halo cells representing interior boundaries are updated via inter-rank communication to contain the same value as the closest interior cells of the neighboring subdomain. The values for the physical boundary are updated based on the boundary conditions in the simulation. Periodic boundary conditions require inter-rank communication to determine what values are present on the opposite side of the field. The halo cells are populated before applying any operations that depend on data on other ranks. We denote with n_g the thickness of this layer. In most applications, we choose $n_g = 1$; this is sufficient, as the Laplacian implementation in IPPL uses the 7-point stencil. However, an application that makes use of a larger stencil would require more halo cells.

3.2 Rank Independence via Wrappers

We can achieve rank independence in functors[3] by adding an additional layer of abstraction to the parallel dispatch. Kokkos expects functors to take multiple separate arguments, namely a number of indices equal to $\dim \mathbb{D}$. By creating a wrapper object that collates these arguments into a vector, we can write our functors to take a single argument instead, namely a vector $I \in \mathbb{N}^{\dim \mathbb{D}}$. Since the rank of the domain is known at compile time throughout the program, it is possible for the compiler to instantiate an appropriate function object for the domain. Experienced C++ programmers may already be thinking of a wrapper-free approach utilizing parameter packs, but this is subject to limitations that we will discuss in Section 3.5.

To simplify the syntax used in IPPL and its derived programs, we extend the IPPL API to encompass parallel dispatch, reducing the amount of boilerplate required for rank independent parallel dispatch. Functors are wrapped in a container object that collates variadic input arguments and passed on to Kokkos, which continues to handle platform-specific dispatch. This makes it easy for end users to write rank independent algorithms within the IPPL framework, as the vector structure naturally identifies each input argument with its dimension.

[2] In MPI terminology, each instance of the running process is known as a rank; this is not to be confused with the mathematical meaning of rank.

[3] In this context, a functor is an object that can be called like a function. This can be a lambda expression or an instance of an explicitly defined struct with a defined function call operator. See also 20.14 § 1 in [6].

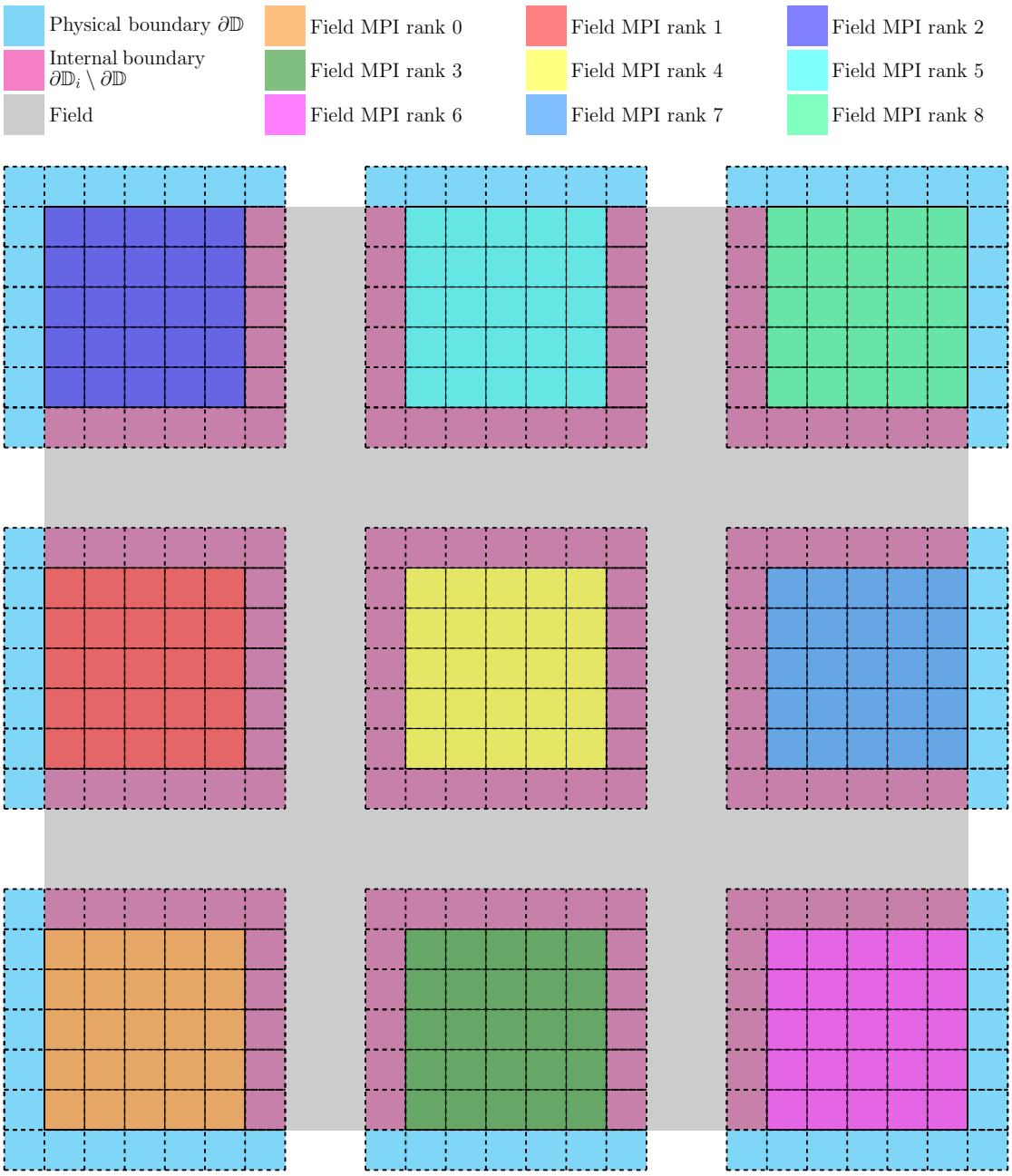

Figure 3.1: A visualization of how a 2D field would be stored across 9 ranks. The full domain is divided into subdomains $\mathbb{D}_i$. Each rank stores the field values in the interior of its subdomain and has $n_g = 1$ layer of halo cells to represent internal boundaries or the physical boundary of the domain, depending on which subdomain is stored on the rank. The gray region represents the full extent of the field. The diagram shows a discretization with $N = 15$. We note that in practice, the number of ranks and N are generally both chosen to be powers of two for optimal load distribution. Diagram originally made for [5], adapted from a pre-existing diagram courtesy of Dr. M. Frey.

```
using mdrange_type = Kokkos::MDRangePolicy<Kokkos::Rank<3>>;
Kokkos::parallel_for(
        "Assign initial rho based on PDF",
        mdrange_type({nghost, nghost, nghost},
                {rhoview.extent(0) - nghost, rhoview.extent(1) - nghost,
                rhoview.extent(2) - nghost}),
        KOKKOS_LAMBDA(const int i, const int j, const int k) {
                // local to global index conversion
                const size_t ig = i + lDom[0].first() - nghost;
                const size_t jg = j + lDom[1].first() - nghost;
                const size_t kg = k + lDom[2].first() - nghost;
                double x = (ig + 0.5) * hr[0] + origin[0];
                double y = (jg + 0.5) * hr[1] + origin[1];
                double z = (kg + 0.5) * hr[2] + origin[2];

                Vector_t xvec = {x, y, z};

                rhoview(i, j, k) = PDF(xvec, alpha, kw, Dim);
        });
```

Listing 3.1: Setting field values using a PDF in IPPL 2.4.0. This implementation is specific to a field in three dimensions; the arguments to the functor are the indices i, j, k that identify a field element.

Vectors and fields in IPPL can be used in mathematical expressions. The evaluation of such expressions is accomplished efficiently using expression templates [7]. While perhaps less elegant than a wrapper-free approach, we benefit from obtaining the element indices in vector form because they become utilizable in these expressions. In particular, it becomes possible to perform a coordinate transformation $(i, j, k, \ldots) \mapsto (x, y, z, \ldots)$ from index space to physical space in a single line in the kernel.

Element access in Kokkos views is achieved via the function call operator (`operator()`). This also requires separate index arguments, but we can adapt the `apply` function from the standard library to work around this issue[4]. Given a rank n tensor of elements in $\mathbb{K}^m$ and a vector containing n indices, the behavior of this function is summarized as:

$$\begin{aligned} \text{apply} : (\mathbb{K}^m)^{\otimes n} \times \mathbb{R}^n &\to \mathbb{K}^m \\ (F, \vec{v}) &\mapsto F(v_1, v_2, \ldots, v_n) \equiv F_{v_1, v_2, \ldots, v_n}. \end{aligned}$$

This resolves the issue of accessing view elements in a rank independent manner. Together with the additional layer of abstraction in the parallel dispatch, we are equipped to rewrite IPPL kernels for arbitrary rank fields. Listings 3.1 and 3.2 show the same kernel implemented in IPPL 2.4.0 and using the new interface, respectively.

[4]The function from the standard library can only be used with standard library containers. We use IPPL vectors both for the expression template capabilities and because standard library vectors are not supported by CUDA.

```
using index_array_type = typename ippl::RangePolicy<Dim>::index_array_type;
ippl::parallel_for(
        "Assign initial rho based on PDF", P->rho_m.getFieldRangePolicy(),
        KOKKOS_LAMBDA(const index_array_type& args) {
                // local to global index conversion
                Vector_t<Dim> xvec = (args + lDom.first() - nghost + 0.5) * hr
                                                                + origin;

                ippl::apply(rhoview, args) = PDF(xvec, alpha, kw, Dim);
        });
```

Listing 3.2: Setting field values using a PDF in a rank independent kernel. The functor takes just one argument, namely an array containing the indices $i, j, \ldots$ that identify a field element. The length of the array is equal to the number of dimensions.

3.3 Interpolation Strategies

We recall that the PIC loop involves two interpolation stages. Particle attributes are "scattered" onto a grid to construct a field. The particle positions are continuous; in order to compute, for example, the charge density ρ, we must interpolate the particle position onto the discrete mesh of the field. The same operation must then be done in reverse when we "gather" the field values to be able to evaluate the equations of motion, such as with the electric field.

In the original IPPL release, several interpolation algorithms were available. The current version of IPPL only supports cloud-in-cell interpolation; additional algorithms will be implemented in future work. With this interpolation method, particle attributes contribute to the field values at all of the mesh points nearest to the particle position in the scatter phase. The converse is true for the gather phase, where a weighted sum of the field values is used to compute the value of the particle attribute. In two dimensions, the field values at the corners of the mesh cell containing a charge density ρ_c at the point (x, y) are computed in the scatter phase as follows:

$$\begin{aligned}
\rho_{ij} &= \rho_c \frac{(\Delta x - x)(\Delta y - y)}{\Delta x \Delta y} \\
\rho_{i+1,j} &= \rho_c \frac{x(\Delta y - y)}{\Delta x \Delta y} \\
\rho_{i+1,j+1} &= \rho_c \frac{xy}{\Delta x \Delta y} \\
\rho_{i,j+1} &= \rho_c \frac{(\Delta x - x)y}{\Delta x \Delta y},
\end{aligned}$$

where $\Delta x, \Delta y$ denote the dimensions of the mesh cell containing the grid points with the given indices [8]. Here, the interpolation uses the particles' physical coordinates. In IPPL, these coordinates are transformed into index space before interpolation; the side lengths of the mesh cell are thus always normalized to one. This rescaled setup is depicted in Figure 3.2.

The cloud-in-cell interpolation strategy can be extrapolated to more or fewer dimensions. Listing 3.3 shows the full expression for the "gather" operation for a single particle in 3D, as implemented in IPPL 2.4.0. The geometrical interpretation of the interpolation weights is analogous to that in Figure 3.2.

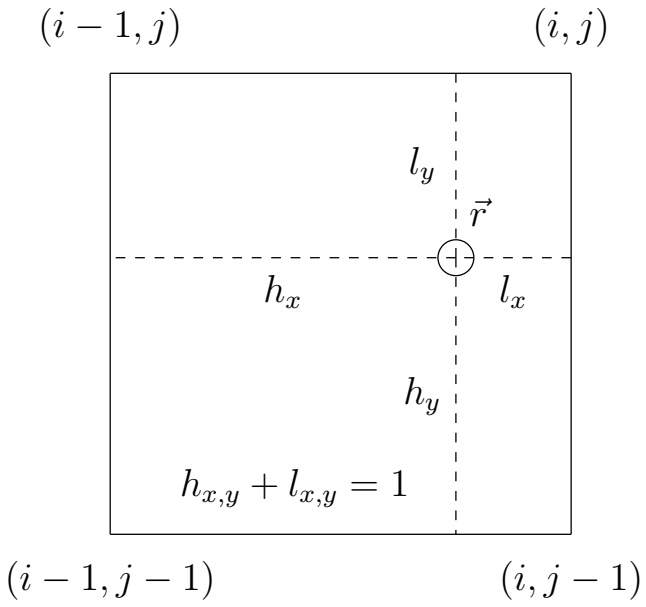

Figure 3.2: Cloud-in-cell interpolation in two dimensions. The particle is located at $\vec{r} = (x, y)$. The indices of the corners of the mesh cell are given as shown. The interpolation weights $h_{x,y}, l_{x,y}$ play the same role as the scale factors $\frac{x}{\Delta x}, \frac{\Delta x - x}{\Delta x}, \ldots$; the ratios of the weights correspond to the ratios of the particle's distances from the corresponding edges of the mesh cell: the field value at the mesh point closest to the particle's physical location contributes the most to the attribute value when gathering. Conversely, in the scatter phase, this mesh point receives the greatest contribution from the attribute value.

```
val =     wlo[0] * wlo[1] * wlo[2] * view(i - 1, j - 1, k - 1)
        + wlo[0] * wlo[1] * whi[2] * view(i - 1, j - 1, k)
        + wlo[0] * whi[1] * wlo[2] * view(i - 1, j, k - 1)
        + wlo[0] * whi[1] * whi[2] * view(i - 1, j, k)
        + whi[0] * wlo[1] * wlo[2] * view(i, j - 1, k - 1)
        + whi[0] * wlo[1] * whi[2] * view(i, j - 1, k)
        + whi[0] * whi[1] * wlo[2] * view(i, j, k - 1)
        + whi[0] * whi[1] * whi[2] * view(i, j, k);
```

Listing 3.3: Gathering a value using cloud-in-cell interpolation in three dimensions. Here, `val` is the gathered value, `wlo` and `whi` are the low and high interpolation weights, and `view` is a Kokkos view containing the field values.

In order to be able to implement the scatter and gather operations in an arbitrary number of dimensions, we must find an efficient generalization of the interpolation operations. The key is in the pattern of the index manipulations. The number of points to and from which we must interpolate is given by $2^{\dim \mathbb{D}}$. The interpolation then involves some product of the high and low weights and the field value at some offset indices. In particular, the index offsets and choice of weights coincides exactly with the binary representations of the natural numbers in the interval $[0, 2^{\dim \mathbb{D}})$.

As an example, we can examine the second term in the sum shown in Listing 3.3. The starting indices are i, j, k; in this term, the first two indices are decreased by one and the third is unchanged. Correspondingly, we use the first two components of the low weights and the third component of the high weights. Thus, we use the low weights for lowered indices and high weights for unchanged indices. If we use zero-indexing to identify the terms of the sum, this is term $1_{10} \equiv 001_2$. We note that the 1 bits exactly identify the axes for which we choose the high weights and leave the indices intact and the 0 bits identify the axes with the low weights and reduced indices. Careful inspection will reveal that this is the case for all the terms in the sum[5].

This pattern immediately yields an efficient compile-time expansion of cloud-in-cell interpolation. By using C++ parameter packs and the index sequence utilities in the standard metaprogramming library, we can create a function that makes $2^{\dim \mathbb{D}}$ calls to a scatter or gather operation. Each call is identified by its index in the interval $[0, 2^{\dim \mathbb{D}})$ and utilizes the weights and index offsets described by the binary expansion of that index, whose bits can be easily queried using bitwise operations.

The number of bits to consider is exactly $\dim \mathbb{D}$. This pattern can thus be generalized to domains of arbitrary rank; the number of interpolation points simply grows exponentially. We leverage pack expansion and fold expressions in C++ to generalize the scatter and gather operations to any number of dimensions. This new implementation is more complex than the original, since the interpolations are generated sequentially instead of appearing directly in the code. However, the terms are all generated at compile time and can be inlined by compiler optimizations. The compiler should thus be able to generate code that is effectively equivalent to the original implementation. As we will see in Chapter 5, where we discuss performance in greater detail, the added complexity in the implementation has negligible impact on the overall performance.

3.4 Identifying Domain Neighbors

In IPPL 2.4.0, field layouts stored separate lists for vertex, edge, and face neighbors, as these were the only kind that could appear. This assumes three-dimensional domains. In order to achieve rank independence, the field layouts need to be able to identify neighboring domains beyond just 3D domains, but also properly search for neighbors in fewer dimensions. One option is to simply extend the interface and add more neighbor lists. However, there is a more elegant method.

The concept of a cube[6] can be generalized to more than three dimensions. A cube in four dimensions is commonly known as a tesseract. More generally, a cube in n-dimensions can be

[5] Of course, the terms in the sum are written in a specific order such that this statement is true. Since addition is commutative, the terms could be reordered and this statement would no longer hold. However, the existence of such an ordering is equivalent to the existence of a one-to-one mapping between $[0, 2^{\dim \mathbb{D}}) \cap \mathbb{N}$ and the terms of the sum. This latter property enables the procedural generation of the entire sum.

[6] The domain does not necessarily have the same length in each direction. We are thus interested in the properties of not just generalized cubes, but generalized rectangles. These are known as orthotopes, hyperrectangles, or boxes. We use the terms “cube” and “hypercube” for simplicity as the properties discussed here are also true for hyperrectangles.

referred to as an n-cube. The generalization is also applicable to fewer dimensions. Vertices, edges, and faces are 0-, 1-, and 2-cubes, respectively. Collectively, these generalized cubes are commonly referred to as *hypercubes*. For $m < n$, the number of m-cubes contained in an n-cube is $2^{n-m}\binom{n}{m}$ [9]. It can be easily proven by induction (see Appendix A) that

$$\sum_{m=0}^{n} 2^{n-m}\binom{n}{m} = 3^n. \tag{3.1}$$

In other words, an n-cube contains a total of 3^n hypercubes of equal or smaller dimension, where the only hypercube of equal dimension is, of course, the n-cube itself. Hence, there are $3^n - 1$ *internal hypercubes*. For example: a normal cube (i.e. a 3-cube) contains $3^3 = 27$ total hypercubes, namely the cube itself, 6 faces, 12 edges, and 8 vertices. This property means that, given an n-cube, we can uniquely identify all internal hypercubes with a natural number $0 \leq i < 3^n - 1$.

By imposing a set of rules on the assignment of these numbers, we can implement an encoding to consistently map each internal hypercube Ξ to a unique natural number $\mu(\Xi)$. For each rank r, we can then create a single nested list[7] of neighbors such that all ranks r' with intersection $\mathbb{I}_{rr'} \subset \Xi$ are stored in the inner list at index $\mu(\Xi)$.

Since the number of hypercubes is 3^n, the natural choice is to use a ternary encoding (i.e. base-3) for the indices. We write the index as $\mu \equiv (\ldots, \mu_z, \mu_y, \mu_x)_3$, where each digit encodes a piece of information about the hypercube's relation to the corresponding coordinate axis. In the implementation chosen for this project, we choose $\mu_A = 2$ if the hypercube is parallel to the A axis, i.e. it spans a continuous range with nonzero measure along that axis. If instead it is confined to the upper boundary of the domain along that axis, we choose $\mu_A = 1$. Otherwise, it must be on the lower boundary and we choose $\mu_A = 0$.

As an example, consider the unit cube $\mathbb{D} = [0, 1]^3$. The upper face along the X axis is the subdomain $\Sigma = \{1\} \times [0, 1] \times [0, 1]$. This face is parallel to the YZ plane and therefore also spans non-trivial ranges along these axes. All points in Σ have X-coordinate 1, which is the upper boundary of the unit cube along the X axis. Thus, the encoded index for this subdomain is $\mu(\Sigma) = 221_3 \equiv 25_{10}$. For a three-dimensional field layout, the face neighbor ranks whose subdomains intersect the upper X face of the local subdomain are therefore stored in the inner list at index 25.

To better illustrate the encoding, Figure 3.3 shows the encoded values for hypercubes in 2D and 3D. Each hypercube is labeled with its encoded index, excluding the occluded faces of the 3-cube, whose labels are omitted to reduce clutter in the diagram. The labels are color coded according to the dimensionality of their corresponding hypercube.

3.5 Rank Independence via Templates

We note that there is an alternative approach to achieve rank independent kernels. Lambda functions in C++ can be generic, which means they can be written to take an arbitrary number of arguments of arbitrary type by using parameter packs. For simple parallel dispatch kernels that only take the indices as arguments, the parameter pack can be deduced by the compiler. Given an n-dimensional range policy, Kokkos will call the functor lambda with n index arguments; the parameters of the instantiated lambda operator are thus deduced to be n instances of the index type.

In most cases, the indices can then be easily manipulated using pack expansion. This approach is particularly elegant for trivial kernels such as field assignment, where the indices do not require

[7] A nested list is a list of lists. More precisely, the neighbor list is actually stored as an array of vectors.

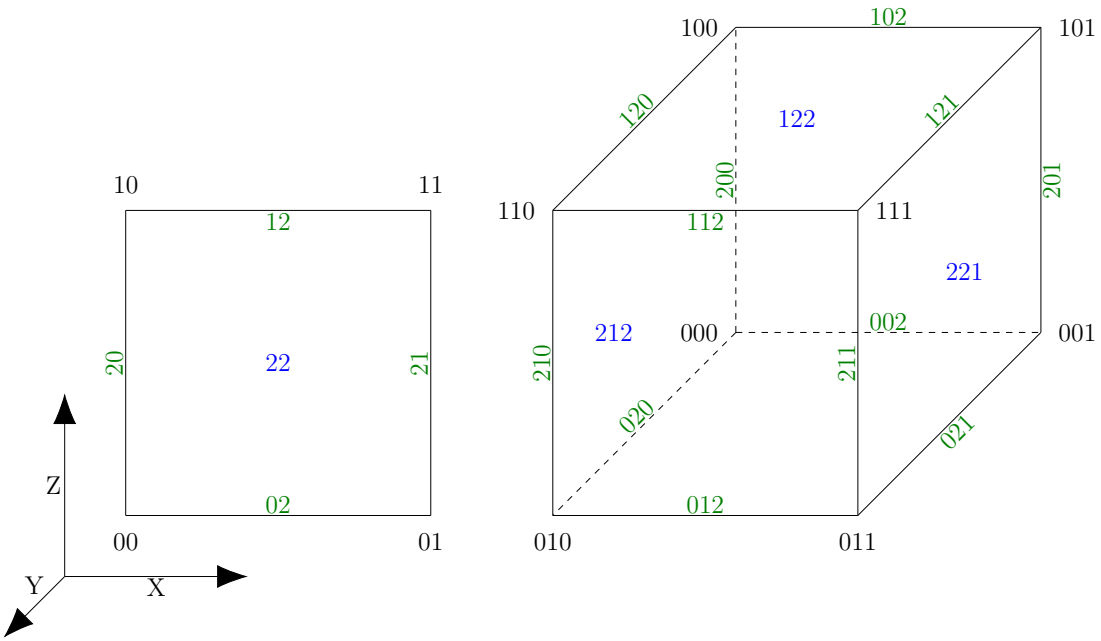

Figure 3.3: Encoded values for hypercubes in 2D and 3D. For the 2D values, we consider a square in the XZ plane. The dashed lines in the 3-cube represent occluded edges. Vertices are labeled in black; edges are labeled in green; faces are labeled in blue. To reduce clutter, only the visible faces of the 3-cube are labeled. For each visible face, the opposite face is identified by the same value with a 0 in place of the 1. The 3-cube itself is identified by $\mu = 222_3$.

any manipulation; the arguments can simply be forwarded to the views instead of using the `apply` function.

However, the C++ standard explicitly states that parameter packs are not deduced if they are not the last arguments in a function (13.10.2.5 § 5.7 in [6]). Reduction and scan kernels in Kokkos require that the functors take the indices first and additional arguments last, which means that the parameter packs cannot be deduced for these operations. For these kernels, a wrapper object is still necessary to concretely specify the number of index arguments.

This limitation in template argument deduction could be easily bypassed if Kokkos reduction and scan functors were modified to take their non-index arguments first instead of last. The argument order is a design choice that can be changed; discussions with the Kokkos team [10] confirmed that there is no technical limitation that requires the additional arguments to come last. Should there be enough demand for such a change, it is plausible that a future Kokkos release could incorporate this change. This would eliminate the need for any wrapper objects and templating could be used to trivially achieve rank independence in any Kokkos kernels.

Unfortunately, there is another hurdle preventing this from being a viable solution. Templated lambdas were not used for Kokkos kernels in this project due to a limitation in the NVIDIA CUDA Compiler Driver (NVCC). In December of 2022, NVIDIA released version 12 of their CUDA toolkit [11]. The documentation for the new compiler describes the limitations on lambda support, which explicitly forbid generic extended host/device lambdas (14.7.2 § 8 in [12]). These are exactly the kind of lambdas used for Kokkos functors.

The templated approach was tested in CPU builds using GCC and OpenMP, yielding correct results. If a future update to the CUDA toolkit were to remove the aforementioned restriction on

extended lambdas, then IPPL could be modified to use templated lambdas instead of the functor wrapper. In the kernels that benefit from receiving the indices as a vector (such as for coordinate conversions), this vector can still be constructed from the arguments using pack expansion as is currently done in the functor wrapper. For other kernels where the modifications to the indices are minimal, the use of pack expansion over the `apply` function would yield much more elegant code.

Chapter 4

Verification of Results

We verify the correctness of the implementation using two mini-apps from ALPINE, namely Landau damping and the two-stream instability. These problems have been studied extensively in the existing literature (see [2] for detailed literature references) and thus have known analytical properties that can be easily verified.

The physical domain $\mathbb{D}$ is the hypercuboid with side lengths $L_d = 2\pi k_d^{-1}$, where $\vec{k} \in \mathbb{R}^{\dim \mathbb{D}}$ is the wave vector. For these simulations, we choose $k_d \equiv 0.5$, so the domain is a simple hypercube $[0, 4\pi]^{\dim \mathbb{D}}$.

4.1 Landau Damping

We consider a plasma with an initial phase distribution given by

$$f_0(\vec{x}, \vec{v}) = (2\pi)^{-\frac{3}{2}} \exp\left(-\frac{v^2}{2}\right) \prod_{d=1}^{\dim \mathbb{D}} (1 + \alpha \cos k_d x_d). \tag{4.1}$$

The strong and weak limits of the Landau damping problems are characterized by damping coefficients $\alpha = 0.5$ and $\alpha = 0.05$, respectively.

To reproduce the results from [2], we choose the same simulation parameters. We set the mesh refinement to $N_d \equiv 32$ and use a particle density of $N_c = 2\,560$ particles per cell. In the 3D case, this corresponds to a total of $N_p = 83\,886\,080$ particles. In 2D, this becomes $N_p = 2\,621\,440$ due to the reduction in mesh size. The result in 2D is not comparable to any prior ALPINE results, but the analytical damping rate is known to be the same. Thus, we expect the results to be visually similar.

We can evaluate the rank independent implementation by verifying the damping rate in fewer spatial dimensions. Figure 4.1 shows the plasma energy evolution over time in three dimensions in the weak limit. Figures 4.2 and 4.3 show the evolution in only two dimensions in the weak and strong limits, respectively. The simulations agree with the analytical predictions in all cases.

The plots shown in this section were generated using MATLAB [13] scripts courtesy of Dr. S. Muralikrishnan.

4.2 Two-Stream Instability

We consider an initial electron distribution given by

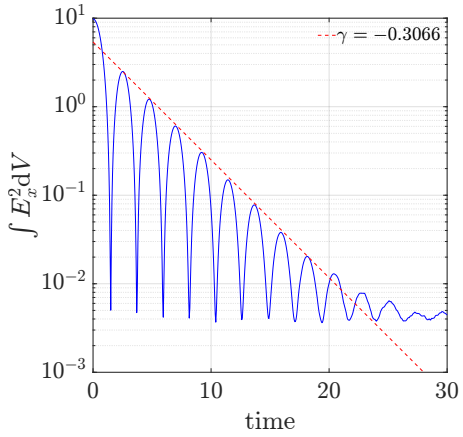

Figure 4.1: Energy evolution for the weak Landau damping limit in three dimensions.

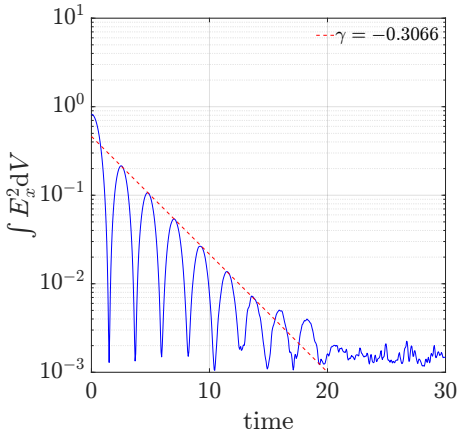

Figure 4.2: Energy evolution for the weak Landau damping limit in two dimensions.

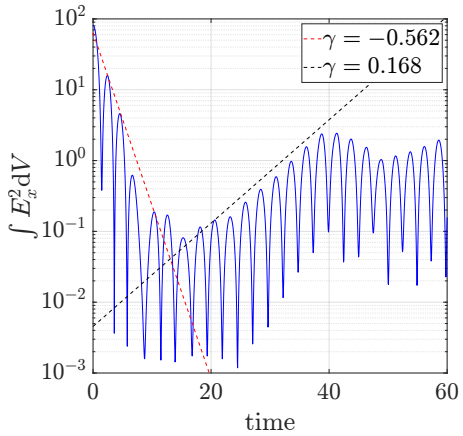

Figure 4.3: Energy evolution for the strong Landau damping limit in two dimensions.

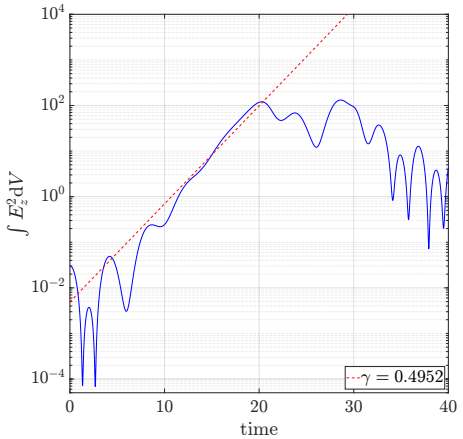

Figure 4.4: Energy growth for the two-stream instability in 2D.

$$f_0(\vec{x},\vec{v}) = \frac{1}{\sigma^3(2\pi)^{\frac{3}{2}}}\left((1-\varepsilon)\exp\left(-\frac{|\vec{v}-\vec{v}_{b1}|^2}{2\sigma^2}\right)+\varepsilon\exp\left(-\frac{|\vec{v}-\vec{v}_{b2}|^2}{2\sigma^2}\right)\right)(1+\alpha\cos kx_{\dim\mathbb{D}}), \tag{4.2}$$

where $\varepsilon = k = 0.5$, $\alpha = 0.01$, $\sigma = 0.1$, and $\vec{v}_{b1}, \vec{v}_{b2} = \mp\frac{\pi}{2}\vec{e}_{\dim\mathbb{D}}$. In particular, we mainly consider the last spatial axis (e.g. in 3D, $x_{\dim\mathbb{D}}$ is the z component of the position and $\vec{e}_{\dim\mathbb{D}}$ is $\hat{z}$, the unit vector in z direction). This makes the two-stream instability a particularly fitting test case for rank independence, as the effects of the phenomenon are predominantly restricted to just one spatial dimension. Simulating this problem as a true 1D-1V problem instead of 3D-3V would reduce the program's memory footprint.

There is a known value for the growth rate of the electric field energy in this scenario. This is presented in Figure 5 of [2] and Figure 3 of [14]. We can verify that the same growth rate is obtained with the new implementation. Since the two-stream instability only affects one spatial dimension, the same growth rate should be observed in fewer spatial dimensions.

The two-stream instability also produces distinctive patterns in phase space. In this implementation, the relevant physics are restricted to the final dimension; we can see the effects of the instability in the z-p_z plane in 3D and in the y-p_y plane in 2D. These effects are also shown in Figure 4 of [14]. For ease of comparison with these results, we plot the phase space at similar points in time using a similar color scheme.

Figure 4.4 shows the evolution of the field energy over time in this simulation. Figures 4.5 and 4.6 show the evolution of the phase space distribution over time in a 2D simulation of the phenomenon.

We note here that we run this simulation in 2D despite prior statements regarding the appeal of running it in 1D due to limitations of heFFTe[1] and IPPL. The default field solver in ALPINE is an FFT solver. This is a performant solver but it can only be used in 2D and 3D because these are the only dimensionalities supported by heFFTe. When running ALPINE in one dimension or more than three dimensions, the only available solver is the conjugate gradient solver. The CG algorithm is often used in conjunction with a preconditioner, but this was not implemented when the CG solver was introduced in IPPL due to time constraints on that project. This drastically limits the performance of this solver. We therefore choose to run the simulation with a higher memory footprint but with the FFT solver.

The simulation was run on NVIDIA's DGX A100 machine, using all eight A100 GPUs. The mesh refinement chosen for the simulation was $N \equiv 2\,048$ and the particle density was $N_c = 30$ particles per cell ($N_p = 125\,829\,120$ total particles).

The phase space plots were generated using C++ bindings to Python's `matplotlib` module [15, 16]. The energy growth rate plot was generated using MATLAB [13] scripts courtesy of Dr. S. Muralikrishnan.

[1] Highly Efficient FFT for Exascale, a C++ library for Fourier transforms. IPPL delegates FFTs to this library.

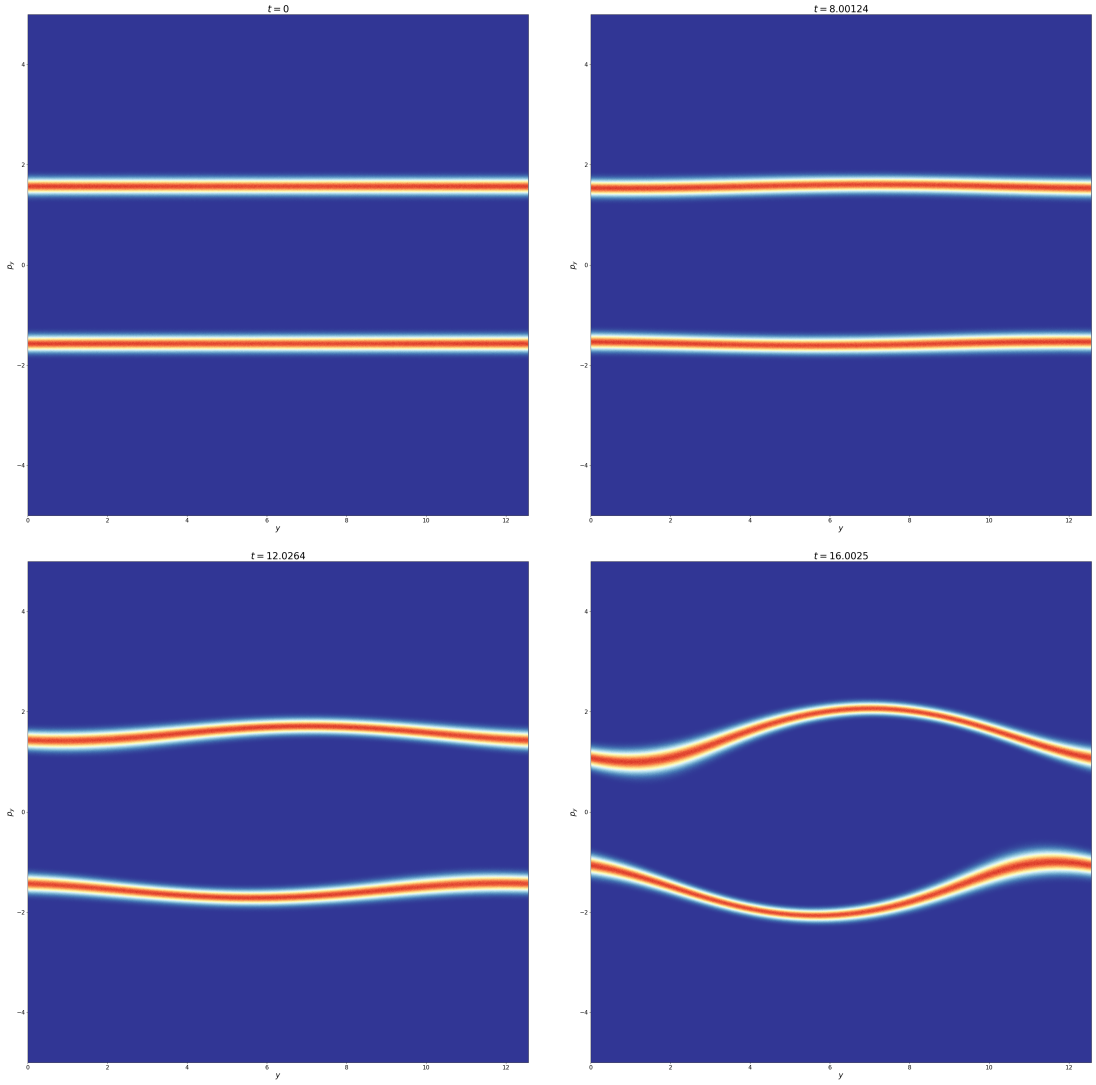

Figure 4.5: Phase space for the two-stream instability at early stages.

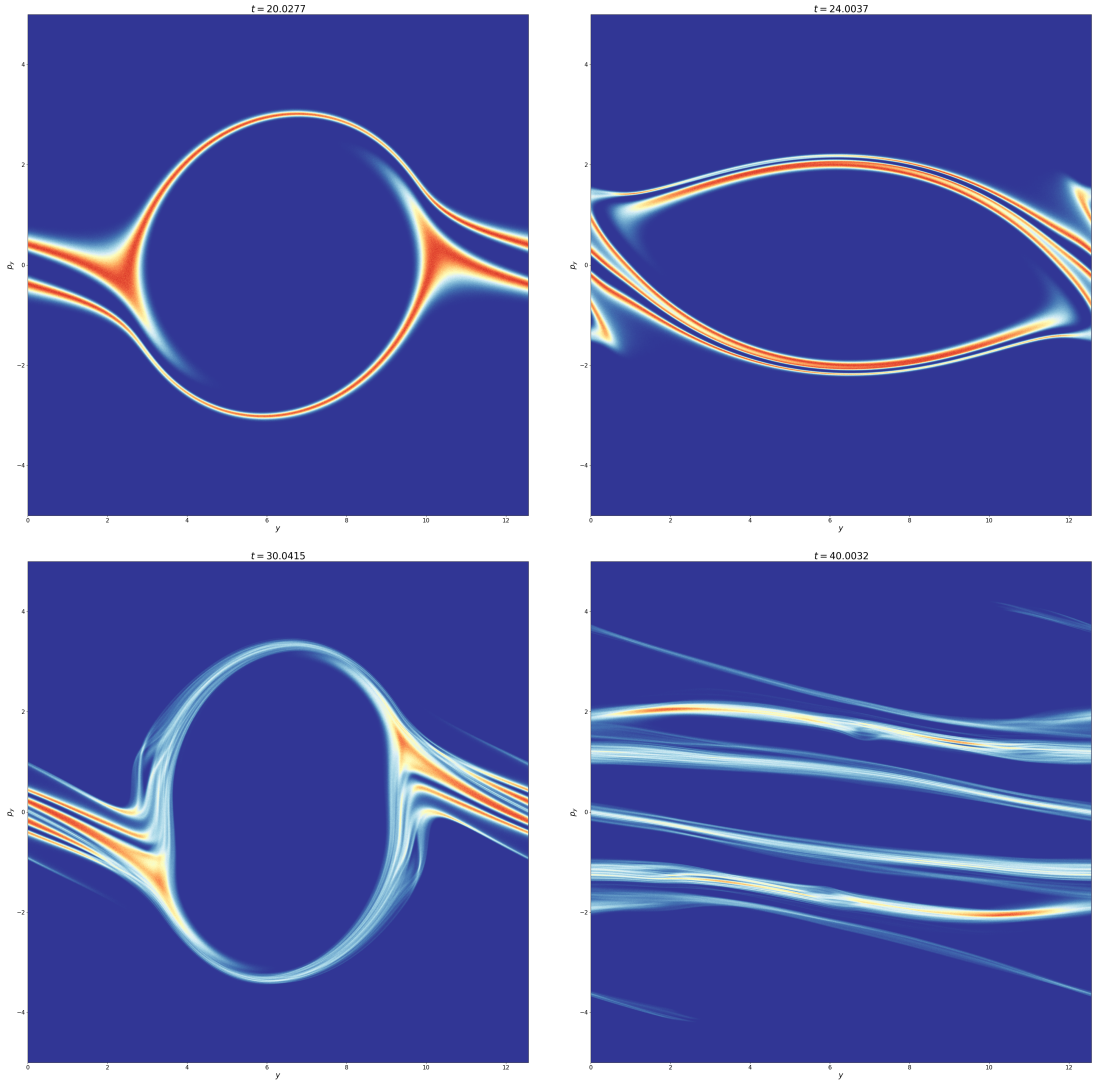

Figure 4.6: Phase space for the two-stream instability at late stages.

Chapter 5

Performance Studies

5.1 Scaling in 3D

To verify that the overheads introduced to achieve rank independence did not significantly impact performance, we reproduce a subset of previously published strong scaling results for the Landau damping mini-app in ALPINE [2]. The parameters used for the reproduction study correspond to cases B and D from the aforementioned paper:

- Case B: 1024^3 grid, 8 particles per cell
- Case D: 512^3 grid, 64 particles per cell

In both cases, we consider the limit of weak Landau damping, with $\alpha = 0.05$.

The scaling studies are performed on the Piz Daint cluster at the Swiss National Supercomputing Center (CSCS). Although this cluster will soon be decommissioned, by using the same hardware, we ensure that our results are directly comparable to those in [2]; we refer the reader to this paper for the reference results. We use the GPU partition, which contains Cray XC50 nodes, each with one 16 GB NVIDIA P100 GPU, 2 Intel Xeon E5-2690 v3 cores at 2.6 GHz, and 16 GB of RAM [17]. We run the simulation on 128, 256, 512, and 1024 nodes. For Case D, we also run the simulation on 2048 nodes. In each case, each MPI rank is allocated one GPU. The runtimes are shown in Figure 5.1.

5.2 Scaling in 2D

We perform a smaller scaling study on the NVIDIA DGX A100 at PSI to confirm that the rank independent algorithms exhibit the same behavior in 2D. The machine is equipped with eight A100 GPUs, each with 40 GB of memory [18]. We again consider the limit of weak Landau damping.

With one fewer dimension, the amount of memory occupied by the fields shrinks considerably. We therefore use a much finer mesh to create a sufficiently large workload. For this study, we choose a mesh refinement of $N_d \equiv 4\,096$ and a particle density of 32 particles per cell. Figure 5.2 shows the runtimes of the different components of the program. We note that the runtime of the solve step actually increases with the GPU count at this scale; this is expected behavior that has already been verified in prior internal experiments. The solver workload is insufficient to outweigh the communication costs incurred by decomposing the problem. We have seen in Figure 5.1 that the solver does scale properly for larger problems.

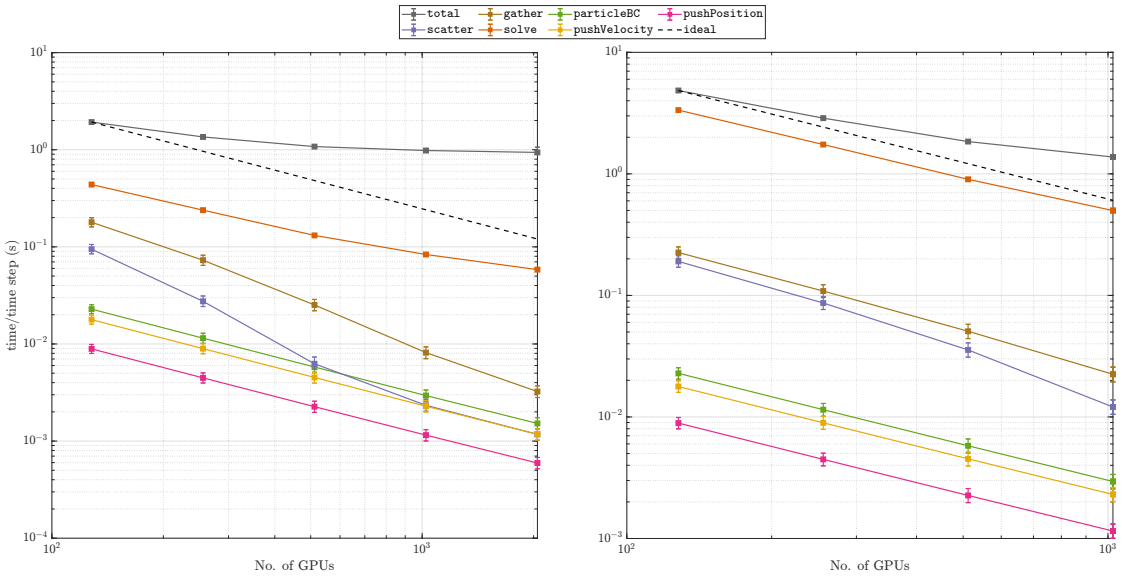

Figure 5.1: Scaling results in 3D on Piz Daint GPUs in Case B (left) and Case D (right). Plot generated using MATLAB [13] scripts courtesy of Dr. S. Muralikrishnan.

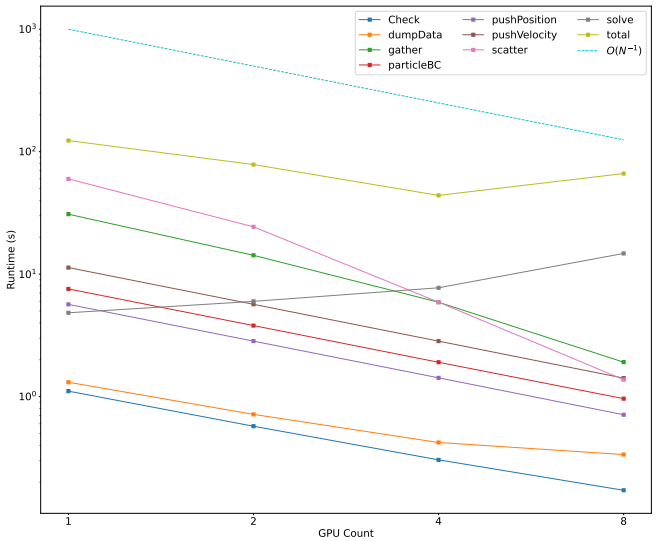

Figure 5.2: Scaling results in 2D on the DGX A100. Plot generated using Python [16, 19].

Part III

Mixed Execution Spaces

Chapter 6

Background

We begin by recalling the premise of the problem and defining the key concepts.

6.1 Terminology

6.1.1 Parallelism in Kokkos

Parallel execution in Kokkos is controlled by many factors, descriptions of which can be found in the documentation [20]. In this part of the project, we focus on what Kokkos calls *execution spaces* and *memory spaces*. These describe where code is executed and where memory is allocated.

As mentioned previously, Kokkos can be compiled to include support for several *backends*. If both CUDA and OpenMP are enabled during compilation, the user gains access to both as execution spaces and can use Kokkos to dispatch kernels to either of them. For simplicity, Kokkos also defines a default execution space. This means that the user is not required to explicitly choose an execution space for each kernel. Kokkos prioritizes GPU computation over CPU computation, so in a CUDA+OpenMP build, CUDA will take precedence and be chosen as the default execution space.

It is important to note that, in general, code running on a GPU does not have access to host memory, and the converse is true for code running on the host. This means that a kernel executed with OpenMP cannot access GPU memory and a CUDA kernel cannot access host memory. This illustrates the need to also distinguish the different memory allocation regions.

The memory spaces are defined not only in terms of where the data is physically stored, but also from which execution spaces they can be accessed. A Kokkos build with both CUDA and OpenMP defines not only CUDA and host memory, but also the CUDA UVM and "host pinned" memory spaces, which are generally accessible from both host and device execution spaces. These may be the subject of future study; in this project, we focus primarily on host- and device-exclusive memory spaces.

We recall that a Kokkos view is a data type that represents a multidimensional array. This type accepts up to four template parameters, which Kokkos uses to determine how the view should behave. In particular, the user can choose an execution space and/or a memory space for the view via these parameters. We also note that, because these properties are specified via template parameters, views with different properties are also fundamentally different types. This creates a few challenges for the storage of data in multiple memory spaces. These will be discussed in more detail in Section 7.1.

The execution and memory spaces complement each other when defining the behavior of a Kokkos view. Should only one of the two be explicitly specified, it can be used to deduce the other. For example, if the user specifies that a view should use host memory, that view will default to OpenMP as its execution space, if available[1]. Analogously, if the user specifies OpenMP as the execution space, its memory space will default to host memory.

6.1.2 Synchronicity and General Parallelism

Modern processors have multiple *cores*, each of which can perform tasks independently of the others. A program with multiple *threads* can thus perform multiple tasks at the same time, as long as the number of threads of execution does not exceed the number of cores[2]. However, each thread can only perform one task at a time and must complete an instruction before it can continue to the next. This brings us to the concept of *blocking calls*. When a blocking function is called, control does not return to the caller until the function terminates. In other words, the thread is blocked from doing other work until the function is complete. In contrast, a *non-blocking* function dispatches work in a different thread and terminates before that work is completed. Launching a CUDA kernel can be non-blocking, since the computations are being performed on the GPU; once the work has been dispatched to the GPU, control can return to the caller and the thread is free to continue with other tasks while the kernel is running. We say that the kernel dispatch is non-blocking and that the GPU computation executes *asynchronously* with respect to the calling thread.

After dispatching work to the GPU, it may be necessary to obtain the result of the computation on the host. This requires ensuring that the computation has been completed, since the obtained result would otherwise be incomplete. This can be achieved by waiting for all[3] outstanding operations on the GPU to complete. This form of synchronization is known as a *fence*[4].

For completeness, we recall that, with distributed computing, there is also the level of node parallelism, which we control with MPI. Each MPI rank executes its own copy of the program concurrently with the other ranks. If each rank executes exactly the same code, they will remain in sync up to random latency. However, if different ranks execute different code, it may be necessary to wait for all ranks to reach a common state before proceeding. This synchronization method is known as a *barrier*. We note that MPI collective functions, which involve multiple ranks, can be implicit barriers if they are blocking calls. For example, `MPI_Allreduce` is a blocking collective that requires data from all ranks. Thus, it blocks the calling thread from continuing execution and prevents the rank from proceeding until all other ranks reach the same call.

In order to design an application that maximally utilizes the available resources of a node by leveraging these forms of parallelism, it is necessary to be aware of all possible synchronizations in the application. Maximum resource utilization is achieved when there is zero waiting time; this is difficult to achieve in practice, but we can approach this limit by dispatching as much asynchronous work as possible between blocking calls. In the context of concurrent CPU and

[1] A host execution space must always be available. If there is no backend for parallel execution on the host, serial execution is used instead.

[2] The number of threads running on a modern personal computer at any given time far exceeds the number of cores. The illusion of concurrent execution is achieved by rapidly switching between tasks such that the delay is not noticeable by humans. In the context of this document, we refer to computations as simultaneous only if they truly occur concurrently.

[3] A new Kokkos feature that would allow the polling of individual kernels is under active development.

[4] In some sources this may also be referred to as a *memory barrier*. In this document, we will use the Kokkos terminology and refer to this as a "fence" to avoid ambiguity with MPI barriers.

GPU computation, one would have to match the runtime of the GPU computations with that of any concurrent CPU computations.

6.2 Motivation

Parallel dispatch and memory management for Kokkos views is fully delegated to Kokkos in IPPL. In version 2.4.0, default parameters are used for all views, which means that device memory is always used if available. This reduces flexibility, increases the memory footprint on the GPU, and reduces the overall efficiency of resource utilization. After the main execution thread dispatches the requisite kernels to the GPU, the CPU cores remain idle until the GPU is done with the computation. This can be easily seen with the help of a profiler.

We only have access to NVIDIA GPUs for internal IPPL development. We therefore use an NVIDIA profiler to analyze the library. Using the Nsight Systems profiler, we can examine the CUDA events in ALPINE. This is a powerful tool for analyzing CUDA kernels and performance bottlenecks that are connected to GPU usage, but can also be used to examine non-NVIDIA components, including OpenMP and system calls. These, however, are significantly more sparsely annotated than the CUDA data points. In particular, it is much harder to identify the origin of any non-NVIDIA data point in the profiler. This lack of information means that the profiler data must, to some extent, be interpreted. We thus make a clear note here that **all claims about non-NVIDIA components made using information from Nsight reports contain an element of guesswork**. We make educated guesses about what the profiler data is truly saying based on our intuition and intimate knowledge of the programs being profiled. Nevertheless, we must acknowledge that the data can be misinterpreted. It may be useful to test other profilers in the future to determine if they can provide clearer information and reduce the amount of guesswork involved.

The profiler can be configured to profile system calls during the program, generating an *operating system runtime trace*, or OSRT trace. Figure 6.1 shows the CPU and GPU activity around one solve step in the Landau Damping simulation. The OSRT trace shows that the CPU is entirely occupied by `poll` during the GPU computation. We interpret this, coupled with the CPU utilization graphs, as evidence that the main thread of execution is simply waiting for the GPU to complete its current task before continuing.

We note here that this interpretation of the profiler data comes with a caveat. In parallel computing, the idea of measuring the runtime of specific components loses meaning because multiple operations could be occurring at the same time. This means that their runtimes overlap and that they could be sharing the same hardware resources in the overlapping period, which would adversely affect the total runtime. Up to IPPL 2.4.0, all parallel kernels would use all available resources and there was no opportunity for multiple kernels executing in parallel. This removes the possibility of overlap; kernel timing was thus a meaningful source of data. However, parallel dispatch in Kokkos can be asynchronous in some cases. It is therefore necessary to use a fence and wait for a kernel to complete in order to measure its total runtime. The timers throughout IPPL and ALPINE therefore artificially introduce fences and force the CPU to always wait for GPU computations to complete. The polling shown by the profiler is, thus, by design. Nevertheless, the CPU utilization graphs support the above conclusion; removing the timers would not remove the idle time. Kokkos guarantees that kernels will be executed in the same order as their dispatch and the kernels cannot overlap if they are all running on the GPU. Eventually, data will have to be collected and copied over to the host process, such as for logging or data processing purposes. It is at these points that all outstanding kernels will have to complete before execution can continue, and a fence will still be necessary. Without timers, the

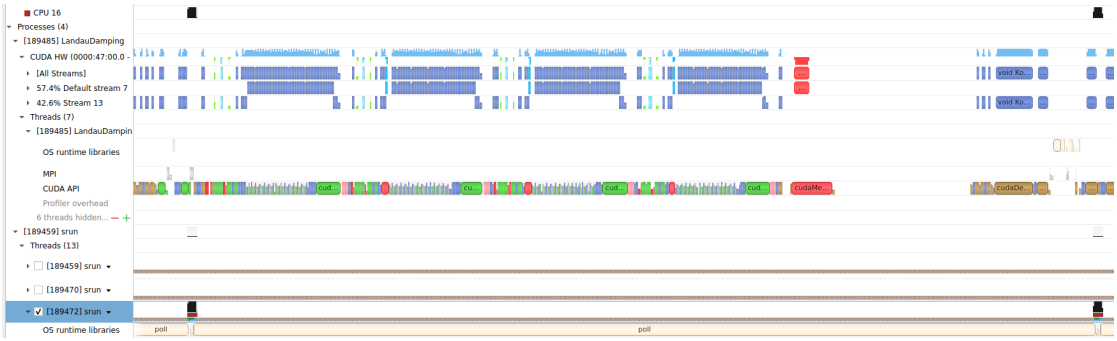

Figure 6.1: Timeline view of the resource usage around one solve call in the PIC loop in Landau damping. The OSRT trace and CPU utilization plots suggest that the CPU only polls the GPU while the computation is running and does no useful work. Only a small part of the program trace is presented here; the full trace shows that there is a call to `poll` *for each time step in the simulation, which we consider to be evidence in favor of our interpretation.*

CPU would poll for a much longer period of time after all kernels are dispatched, rather than for shorter periods between each kernel. The total idle time would be similar in both cases.

This idle time represents wasted resource availability, as the CPU consumes power even when idle. For cluster users who pay for node hours, this idle time is also a waste of financial resources; the cost of using a node depends on the total wall time across all nodes and all resources in the nodes, so an idle CPU has non-zero cost but produces zero data.

The second objective in this project is to make it possible for IPPL users to design their applications in such a way that this idle time is minimized. By introducing the ability to explicitly choose the memory spaces used by different components in a simulation, it becomes possible to perform computations simultaneously on the host and the device.

Chapter 7

Implementation

7.1 Designing for Mixed Execution

Data structures in IPPL are effectively wrappers around Kokkos views that expose specific properties and operations. Since Kokkos supports having multiple accelerators and views in different memory spaces, it is comparatively straightforward to bring this feature to IPPL as well. The implementation of mixed execution spaces in IPPL is primarily inspired by Kokkos' design. Kokkos views are templated not just on their data type, but also on other properties, such as memory layout and memory space.

We effectively borrow the existing static analysis tools in Kokkos to provide the same functionality in IPPL types. By extending the template parameters of IPPL types with a parameter pack and forwarding this to the internal Kokkos views, the end user is able to choose how the data is internally stored. Kokkos views expose their properties (in particular, their execution and memory spaces), which makes it possible to rewrite the algorithms in IPPL to take user-specified memory spaces into consideration.

The use of template parameters for determining the properties of Kokkos views and parallel dispatch policies makes it possible to statically verify key properties at compile time, such as whether it will be at all possible to copy between two given views. This is a very useful feature for development, but also presents a notable drawback for applications wishing to make use of multiple execution spaces. Instantiations of the same template only refer to the same type if all template parameters are equivalent. Thus, a Kokkos view in CUDA memory is not the same data type as a Kokkos view in host memory. This increases the complexity of any data structure wishing to work with Kokkos structures in multiple memory spaces.

One of the goals of this part of the project was to make it possible for a particle bunch to store particle attributes in multiple memory spaces. Supporting mixed execution thus introduces the need for a new data type that is capable of containing objects of types corresponding to each available memory space.

While C++ does provide some data structures for storing arbitrary objects, these are not type safe and offer far greater flexibility than what is required. Instead, the type safe union `std::variant<Ts...>` offers a more controlled container for objects of different types. The available memory spaces are known at compile time, so we can construct variants that admit types of structures in all possible memory spaces. Using C++ `constexpr` functions, we can associate each memory space with a unique numerical index. We then couple this with an array of variants exposed through a type-safe interface to create an array indexed by memory space instead of by unsigned integers. Listing 7.1 shows a small snippet of this structure's interface.

```
template <template <typename> class Type, typename... Spaces>
class MultispaceContainer {
        // a simplification of the actual implementation
        // for illustrative purposes
        using Types = std::variant<Type<Spaces>...>;

        std::array<Types, sizeof...(Spaces)> elements_m;
public:
        template <typename Space>
        Type<Space>& get();
};
```

Listing 7.1: A small snippet of the public interface of the space-indexed container in IPPL. The container is templated on the desired type (e.g. particle attribute) and the full set of desired spaces. The user can then access an element from a given memory space via the class interface. The exact definition of the `Types` alias is more nuanced than presented here, but we omit further details to present a simpler overview.

Listing 7.2 shows how this container is used by IPPL's particle bunch type to track attributes across memory spaces.

7.2 Accelerator Agnostic Operations

7.2.1 Operations in Multiple Memory Spaces

In addition to storing objects of different types, we need to be able to perform operations on them. The individual operations on each particle attribute are not dependent on the memory space itself; we therefore also need a means to perform the same operations on particle attributes independent of their exact data type.

It is not possible to iterate through a space-indexed array via conventional means. Any operation applied to the array elements must be compiled for the specific memory space; this means that any iteration must occur at compile time and not at runtime. To ensure type safety, the numerical indices of the elements are not exposed; the elements are thus identified by data type and not by index. This all leads to the conclusion that iterating through this array requires a templated loop body. Accelerator agnostic operations are thus implemented using templated functions, which can then be instantiated for each memory space.

Templated declarations can only appear at namespace or class scope (13.1 § 4 in [6]). Creating top-level structures for every operation we wish to apply to particle attributes would quickly pollute the namespace and significantly increase the difficulty of maintaining the codebase. The elegant solution is thus to use generic lambdas, with which we can define closure objects with templated function call operators even inside functions.

Generic lambdas with explicit template parameters are a C++20 feature. Using explicit parameters makes it possible to write readable and elegant code that acts on data structures in any memory space. An example is shown in Listing 7.3. We need a function here because, while the operation to be performed in each memory space is the same, the data types are different, thus requiring the instantiation of several different utility functions. By using a lambda function, we can confine these utility functions to the scope in which they are relevant and keep the codebase easier to maintain.

```
template <typename MemorySpace>
using container_type = std::vector<particle_attribute<MemorySpace>*>;

template <class PLayout, typename... IP>
template <typename MemorySpace>
void ParticleBase<PLayout, IP...>::addAttribute(
                detail::ParticleAttribBase<MemorySpace>& pa) {
        // attributes_m has type
        // MultispaceContainer<container_type,
        //                Kokkos::CudaSpace, Kokkos::HostSpace, ...>
        attributes_m.template get<MemorySpace>().push_back(&pa);
        pa.setParticleCount(localNum_m);
}
```

Listing 7.2: A separate array is used to store the attributes in each memory space. When an attribute is added to the bunch, the correct array is obtained based on the memory space in which the attribute is stored. Intermediate utility structs and type aliases are omitted in this snippet for brevity; the member `attributes_m` *is a space-indexed container of vectors of pointers to particle attributes. Here, only two memory spaces are listed for illustrative purposes; in practice, all available memory spaces are present.*

```
unsigned total = 0;
detail::runForAllSpaces([&]<typename MemorySpace>() {
        total += attributes_m.template get<MemorySpace>().size();
});
return total;
```

Listing 7.3: Counting the total number of attributes across all memory spaces used to describe a particle bunch. Here, `attributes_m` *is a data member of the particle bunch. It is an instance of the space-indexed array type.*

However, one must be wary when using new technologies. The C++20 standard was published in December of 2020 [21], but its adoption is not yet universal. The use of C++20 in CUDA-compatible code requires CUDA 12, which was only released in December of 2022 [11]. Especially when shared computing resources are involved, the adoption is then further delayed by rigorous testing. While C++20 provides considerable advantages in the development of this new feature, it does present an obstacle to the adoption of IPPL in external projects. With C++20 as a hard requirement for the library, we restrict the range of machines on which it can be used. We believe that the results from this project are sufficient evidence that the benefits are more than enough to compensate for this restriction on compatibility.

7.2.2 Internal Overheads

We also note that for particle attributes in particular, storing data in multiple memory spaces incurs additional overhead when particles leave a rank's local domain or are otherwise deleted. In IPPL, the properties of particles in bunches are stored in arrays. In ALPINE, for example, a bunch of charged particles is described by the positions and momenta $\vec{x}, \vec{p} \in \mathbb{R}^{\dim \mathbb{D}}$ as well as the charges $q \in \mathbb{R}$ of each individual particle. In addition, the electric field $\vec{E} \in \mathbb{R}^{\dim \mathbb{D}}$ at the location of each particle (as computed in the gather phase) is stored. We allocate an array for each property and store data for each particle in the bunch. Elements with the same index across these arrays describe the same particle.

Memory allocations and deallocations incur significant runtime penalties on GPUs. Therefore, when particles are deleted, IPPL does not release the memory allocated to an attribute array. The number of particles in each rank fluctuates as particles move between the ranks' subdomains, so the memory can be reused later in the simulation when the rank may have to store more particles[1]. Because the arrays are not resized, IPPL instead stores the particle count separately from the array size and partitions the array into a valid region and an invalid region. Valid data is stored from the beginning while the space at the end of the array constitutes the invalid region. The size of the valid region matches the particle count.

When particles are deleted, the valid region shrinks. If the deleted particles were not the final elements in the valid region, then some elements in the valid region will now be invalid while valid particles will be present in the invalid region. We store the indices of the particles in the wrong region and the valid particles are moved into the valid region by overwriting the invalidated particle data in said region. This occurs in parallel using Kokkos; the indices must thus be stored in the same memory space as the particle attribute itself. If attributes are stored in multiple memory spaces, these indices have to be copied to each of these memory spaces.

The effects of this overhead were not explored in this project. Mathematical operations involving fields and particle attributes are performed in parallel in IPPL, so the terms must be stored in the same memory space. The same is true for field and particle data used in the scatter and gather phases. The particle data is updated in each time step according to

$$\vec{x}_1 = \vec{x}_0 + \vec{p}_0 \Delta t,$$
$$\vec{p}_1 = \vec{p}_0 + \frac{q}{m} \vec{E} \Delta t.$$

The particle attributes in ALPINE must therefore all be stored in the same memory space. A future study should investigate the magnitude of this overhead and how it affects other simulations that might store particle attributes in different memory spaces.

[1] The goal of *load balancing* is to ensure that each rank is consistently allocated an equal share of the particles, even as they move around in the domain. IPPL uses orthogonal recursive bisection (ORB) to decompose the domain. A detailed explanation of load balancing and ORB is beyond the scope of this document.

Chapter 8

Applications

8.1 Preliminary Testing

Unlike rank independence, properly leveraging mixed execution spaces involves more than just the implementation of a new feature. The goal is to maximize resource utilization and minimize idle processor time. However, in order to achieve this, the duration of the CPU and GPU workloads must match. This cannot be generalized ahead of time, as the exact runtimes will depend strongly on the simulation parameters and the nature of the simulation itself. Thus, the minimization of idle time is verified empirically and achieved on a per-application basis.

By using the NVIDIA Nsight profiler to analyze both OpenMP and CUDA events, we can verify that host and device code can run concurrently. Figure 8.1 shows the traces for a CUDA kernel that simply sleeps for a large number of GPU cycles alongside an OpenMP loop. This simple check verifies the asynchronicity of Kokkos parallel dispatch and also makes it clear what information is imparted by the profiler timeline. Figure 8.2 enlarges the section where the kernels are launched.

This preliminary testing highlights the fact that there will only be parallel execution on the host and device if the workload on the GPUs is sufficiently large and there are no blocking calls in the main execution thread.

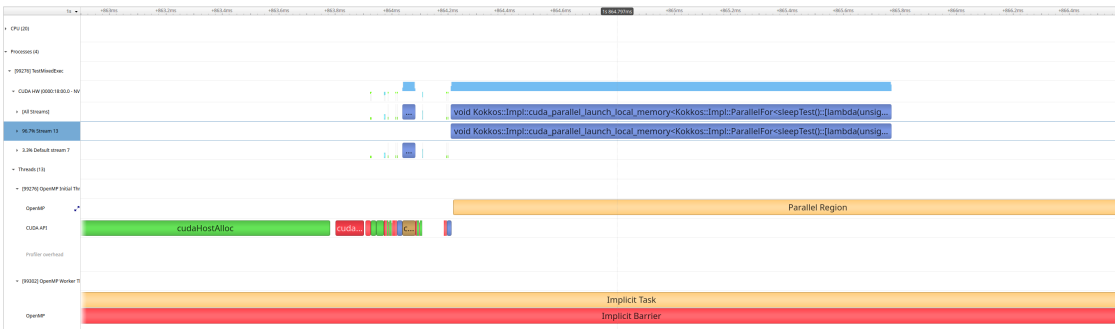

Figure 8.1: Profiler traces for 100 thousand iterations of a CUDA kernel that sleeps for one million GPU cycles and an OpenMP loop that prints some numbers. The profiler shows that the CUDA streams can work in the background without blocking the controlling process.

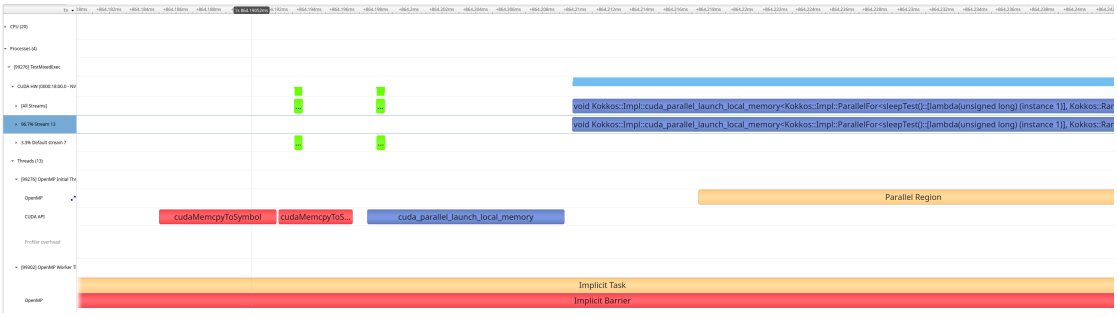

Figure 8.2: A significantly enlarged view of the same trace as Figure 8.1 showing only the section of the trace where the kernels are dispatched.

8.2 Offloaded Postprocessing at Fixed Intervals

The simplest demonstrative use case for mixed execution spaces in IPPL is the data logging in ALPINE. Instead of logging particle and field statistics after every time step, we can decrease the time step size to improve accuracy and only compute the statistics on the CPU after several time steps have been completed. By balancing the step size and the logging frequency, we can tune the runtimes of the host and device code to be approximately equal, thus minimizing any idle time in the main execution thread.

We note here that minimizing the idle time does not necessarily correspond to any overall performance gains. Execution on CPUs is significantly slower than on GPUs. In this example, we artificially increase the GPU workload by simulating a greater number of time steps in order to increase the GPU runtime to match the CPU runtime. We can thus expect the overall runtime to increase. Nevertheless, if we can demonstrate that the cost of offloading to the CPU can be offset by the performance gains of increased parallelism, this will show that mixed execution could feasibly decrease runtime for a larger simulation.

We again use the Landau damping simulation in 3D. The physical domain is given by $\mathbb{D} = [0, 4\pi]^3$ for the weak Landau damping limit. PIC simulations often choose a time step size fulfilling $\omega_0 \Delta t \leq 0.2$, where ω_0 is the characteristic frequency of the system being modeled [22]. ALPINE uses the non-dimensional form of the Vlasov-Poisson equation [2], yielding the time step bound $\Delta t \leq 0.2$, and chooses the step size based on the mesh refinement. For a mesh refinement of $\vec{N} = (N_x, N_y, N_z)$, ALPINE chooses a conservative step size of

$$\Delta t = \min_{i \in \{x,y,z\}} \left\{ 0.05, \frac{2\pi}{N_i} \right\}. \tag{8.1}$$

When launching the simulation, the user is also required to provide the number of desired time steps N_t. For this experiment, we interpret this to refer to the number of data points saved to disk during the simulation. We consider different logging intervals T_L and only compute the field statistics on the CPU every T_L time steps. To reproduce the same data in each simulation, the data must be computed at the same physical times in the simulation. Thus, for $T_L > 1$ we use an adjusted time step size of $\Delta t' = \Delta t / T_L$. Then, the total number of time steps becomes $N_t T_L$ but the simulation output remains the same across all tests.

As a prototype implementation, we can use standard library threads to enable parallel CPU/GPU execution. This ensures that CPU execution can continue even during blocking calls made in the PIC loop. After every T_L time steps, we copy the field data from the device to the host and spawn a new thread in which field properties are computed and saved to disk.

Figure 8.3 shows part of the profiler trace for this setup. We see that the CUDA operations are performed concurrently with the host side processing.

Due to the time step adjustment, the conventional metric of runtime per time step is not meaningful for this experiment. Instead, the fixed quantity is the total number of logged data points. We therefore instead measure the runtime per logged data point. This duration includes the computation time for T_L time steps on the GPUs, so clearly it must be positively correlated with the logging period. However, if we divide this by T_L, we can determine at which point the cost of CPU offloading is offset by the reduced idle time.

We denote with T_{T_L} the total runtime of the simulation with logging period T_L. The reference runtime T_{ref} is the runtime with no mixed execution and exactly N_t time steps. With CPU offloading, the runtime on the GPUs per logged data point should be directly proportional to T_L. However, some time is lost when the simulation waits for the logging to complete, increasing the total runtime. We can thus define an efficiency-like quantity by comparing the true runtime to the theoretical value:

$$\eta(T_L) = \frac{T_{\text{ref}} T_L}{T_{T_L}}. \tag{8.2}$$

This quantity is plotted in Figure 8.4 and describes whether the time saved by offloading work to the CPU offsets the time lost due to slower execution. For the presented parameters, $\eta(128) > 1$, meaning that we have 128 time steps on the GPU for each logged data point instead of just one, but the total runtime increases by a factor less than 128. This shows that we can indeed offset the cost of offloading to a slower execution space if the workload on the GPUs is great enough.

8.3 Adaptive CPU Offloading

In the previous section, we examined the efficiency of the CPU offloading with different logging periods. This demonstrated the existence of a threshold after which the offloading becomes beneficial, but it is not practical for real applications. The exact workload might not always be constant and it is inconvenient for users if they have to run a similar parameter search for their own applications. This is especially true when we consider that the optimal workload balance depends not only on the application itself, but also the application parameters. Thus, in order to determine how to optimally balance CPU and GPU workloads, the parameter search must be performed using a real simulation. For studies involving many compute nodes, this is an impractically expensive prospect. Instead of offloading work to the CPU at fixed intervals, it would instead be more efficient if the offloading frequency could automatically adapt to the workload.

We split the available MPI ranks into two groups. Half of the ranks run the simulation on the GPUs (CUDA) while the other half perform postprocessing using CPUs (OpenMP). In both cases, the workload consists of repeating the same base task until the simulation is complete: the GPU ranks repeatedly compute the next time step of the simulation and the CPU ranks repeatedly process the simulation state and dump relevant data to disk. Let T_G, T_C denote the time taken for the GPU and CPU ranks to each complete their base task once. We note that these are, a priori, unknown values, but we can assume $T_G < T_C$. The time taken to complete the postprocessing must then match the runtime of some number of GPU time steps plus some leftover time, i.e.

$$T_C = \alpha T_G + \delta T, \tag{8.3}$$

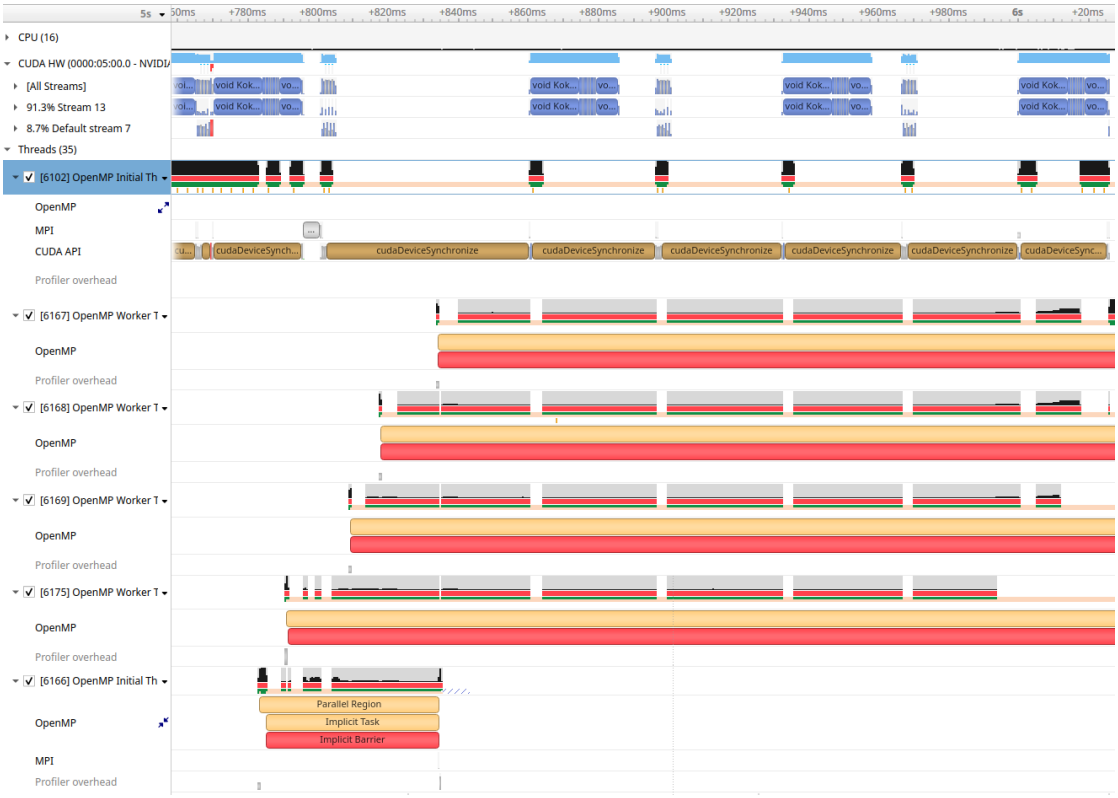

Figure 8.3: OpenMP and CUDA traces for the Landau damping simulation with statistics computations and logging launched on new threads. Note that this screenshot is edited to remove superfluous threads. CUDA stream events are shown at the top; a subset of the OpenMP threads are shown below them. The yellow and red blocks correspond to tasks spawned by the parallel region at the very bottom. Some labels are outside the screenshot bounds.

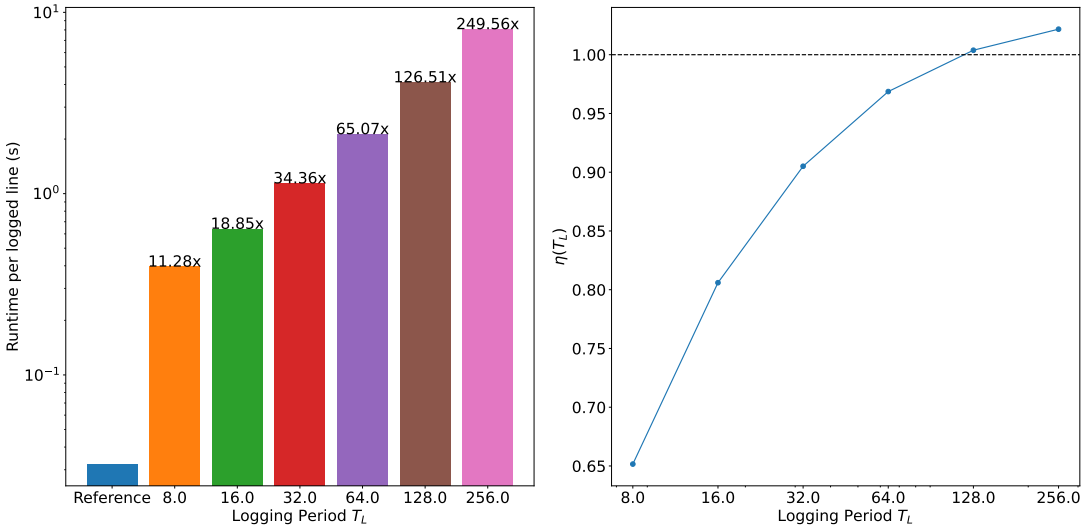

Figure 8.4: On the left, the runtimes with different logging periods compared to the reference runtime. On the right, efficiency of CPU offloading for different T_L. The cost of mixed execution in this example is only offset for $T_L \gtrsim 128$. The behavior of η is strongly dependent on the exact hardware and on the application. This dataset was generated by running the simulation on a GTX 1080.

where $\alpha \in \mathbb{N}$ and $\delta T < T_G$. The optimal setup corresponds to $\delta T = 0$; then the CPU offloading occurs exactly when needed and there is no idle time. To estimate α, we can regularly query the CPU ranks to determine whether they have completed their task, i.e. whether T_C has elapsed. If the task is complete, we can offload the next task. If we perform this query after every time step, i.e. with frequency $f_Q := T_G^{-1}$, then the time during which the CPU ranks are idle is at most T_G and we will have found the optimal workload balance for the chosen f_Q. We note that the optimal balance for a given query frequency f_Q is also the optimal value for any $f'_Q = \beta^{-1} f_Q$ as long as $\beta | \alpha$, as is clear from Equation 8.3.

The queries themselves will also add overhead to the program, seeing as they involve inter-rank communication. To minimize this, we can use `MPI_Iprobe`, which checks if a message is pending without actually receiving the message. We can implement a polling system as follows:

On the CPU ranks:

1. Send a notification message when ready to process more data
2. Receive data
   - If simulation has ended, abort
3. Process the data
4. Repeat

On the GPU ranks:

1. Compute the next time step in the simulation
2. Check if there is a pending message from the CPU ranks
   - If so, send new data
3. Repeat until simulation complete

This polling system allows the program to always offload work to the CPUs almost as soon as they become available, minimizing the idle time and thus making full use of available resources regardless of the simulation parameters.

8.4 Adaptive Noise Reduction using Sparse Grids

While the field statistics mentioned in Section 8.2 provide a simple visualization of a system's evolution over time, a direct visualization of the charge distribution can also be helpful. For this, we use ParaView [23]. After the scatter step is complete, the charge density field can be dumped to disk; this can then be rendered however we choose. If this is done at regular intervals throughout the simulation, we can directly see the evolution of the particles' spatial distribution.

We compute the system evolution using the PIC scheme, which allows us to work in three dimensions (3D-3V) and find approximate solutions to six dimensional phase space equations [24]. However we must contend with a trade-off: using a lower particle density introduces a greater level of noise in the simulation [25] but higher fidelity simulations are more costly. With a noise reduction algorithm, we can run simulations with fewer particles without compromising as much on the quality of the processed output. Such an algorithm has already been studied in [26], albeit using IPPL 1.0. To investigate the applications of mixed execution spaces in the context of postprocessing, we adapt the existing implementation to work with the new IPPL library. We provide a brief summary of the algorithm here and refer the reader to [26] for further details.

To reduce the noise introduced by numerical errors, the charge density is scattered onto a secondary grid that is sparser[1] than the original and then interpolated back to the original grid; the sparse grids act as a low-pass filter and eliminate the high frequency noise introduced by the particle dynamics. By using FFTs to examine the frequency components of the charge density, we can also adapt the choice of sparse grids and change the level of noise reduction depending on the level of noise in the data. For a starting mesh refinement of 2^n, the level of noise reduction is parameterized by a value $\tau \in [1, n] \cap \mathbb{N}$, where $\tau = 1$ corresponds to the sparsest grid and thus the most aggressive noise reduction while $\tau = n$ corresponds to no noise reduction at all, as the sparse grid is identical to the original grid. With the implementation used in this project, the sparse grids generated by $\tau = n - 1$ in three dimensions are invalid; thus the range of useful values for this parameter is limited to $[1, n - 2] \cap \mathbb{N}$ for the purposes of this study.

We verify the implementation of the noise reduction algorithm using the Landau damping and Penning trap simulations. In particular, we look at the algorithm's effects on the simulation state and the algorithm's internal state, which is always saved to disk. The noise reduction algorithm is employed to smoothen the charge density field after the scatter step in the PIC loop. This eliminates particle noise in the solve phase, which slightly modifies the dynamics of the system. However, the overall dynamics of the systems remain the same.

Figure 8.5 shows the field energy over time in the Landau damping simulation with a 128^3 grid and 10 particles per cell over 600 time steps. Each curve shows the energy evolution with different levels of noise reduction applied. We see that the decay rate in the simulation matches the analytical rate for a longer period of time when the noise is more strongly reduced. Even with only minimal noise reduction with $\tau = 5$, there is a clear improvement on the noisy simulation.

In Figures 8.6 and 8.7 we see snapshots of the charge distribution in the Penning trap simulation with and without noise reduction applied. The simulation parameters are as follows: 128^3 grid, 1 particle per cell, 300 time steps. For ease of comparison, the contours in both images are generated using identical color maps corresponding to 20 equidistant points[2] in the interval $[-6.9, 0]$. In this simulation, adaptive noise reduction was enabled. Figures 8.8 and 8.9 show the optimal value of τ and the errors associated with the different levels of noise reduction, respectively, over the course of the entire simulation.

[1] A sparser grid has a lower mesh refinement but approximates the same physical domain.

[2] The depicted contours actually only use the first 19 of these points. The value 0 is excluded to remove a non-physical shell of charges that would otherwise appear in ParaView and occlude the actual particle bunch.

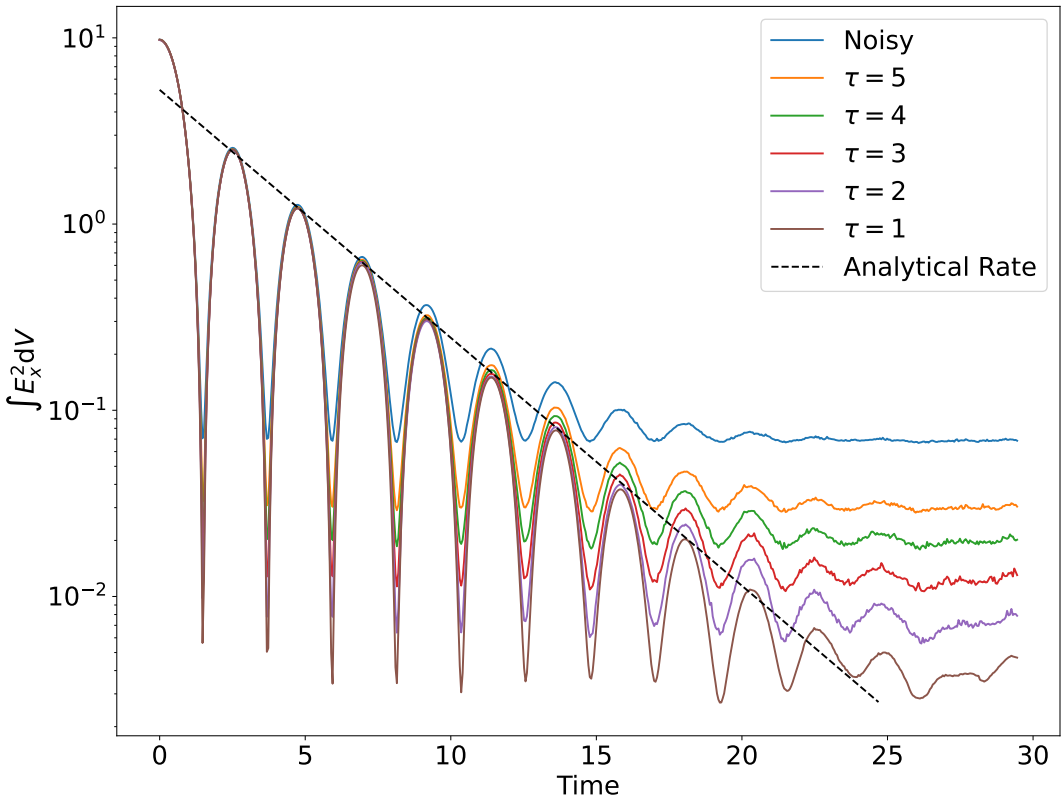

Figure 8.5: The simulation follows the analytical damping rate for longer when more aggressive noise reduction is applied. We recall that higher values of τ correspond to less aggressive noise reduction, whereas $\tau = 1$ denotes the most aggressive noise reduction.

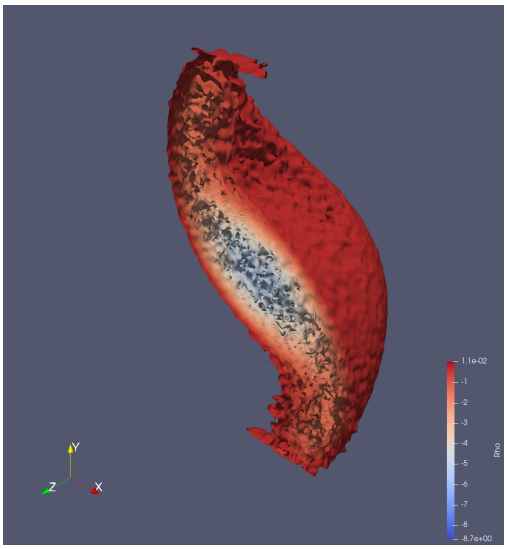

Figure 8.6: The system state in the Penning trap simulation at $t = 0.7$ with no noise reduction. Since IPPL does not currently support VTK dumps with decomposed fields, the simulation was run on one rank. Charge density distributions rendered using ParaView [23].

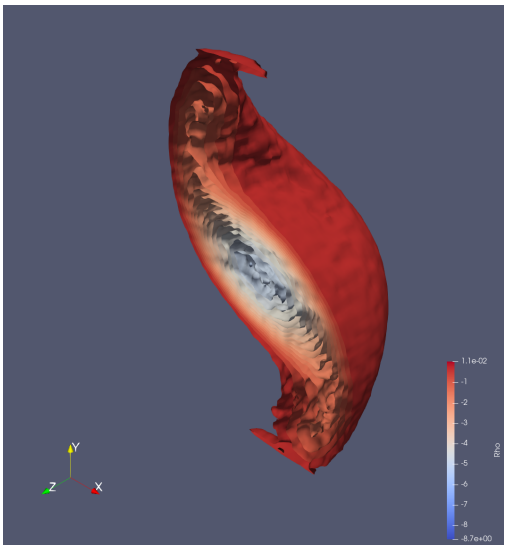

Figure 8.7: The system state in the Penning trap simulation at the same time step with the same parameters, but with adaptive noise reduction applied in each iteration.

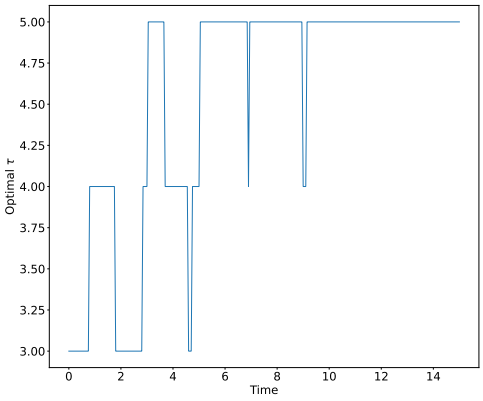

Figure 8.8: Evolution of the optimal value for τ during the Penning trap simulation with adaptive noise reduction.

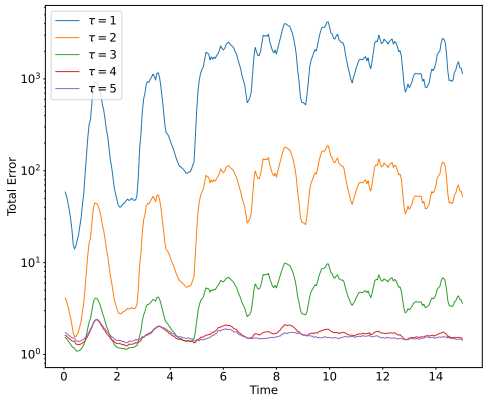

Figure 8.9: Total error for the different levels of noise reduction during the same simulation.

8.5 Performance Benefits

In Sections 8.2 and 8.3, we discussed the efficiency of offloading work to the CPUs at various intervals and how to balance the workloads to minimize idle time for both CPUs and GPUs. However, minimizing idle time does not automatically improve performance. We must contend with the overhead incurred by copying between the host and the device. This is a penalty that depends strongly on the hardware configuration of the machine. Given sufficient bandwidth for host/device copy operations, the overhead may be small enough to be negligible compared to the reduction in runtime. However, one of the core goals of IPPL is portability across hardware architectures. Thus, it is important to verify that these overheads are justified even without such strict requirements on the system specifications.

To evaluate the performance gains from offloading work to the CPU, we construct a simple program to emulate the same CPU/GPU interactions of a more complex mixed accelerator program and employ the adaptive offloading methodology described above. We envision a hypothetical program that makes use of IPPL as a PIC component of a larger simulation. One could run the PIC component on the GPUs while the CPUs continue with another component. In the context of a fluid dynamics simulation, the number of PIC time steps computed on the device could be hundreds of times greater than the number of time steps on the host. This could be an ideal setup for the mixed accelerator capabilities implemented in this project.

We again use the Penning trap mini-app from ALPINE and examine the impact of offloading the noise reduction to the CPUs under different conditions. Since the primary component of these simulations is the time stepping loop, the total program runtime should be directly proportional to the number of time steps N_t. Let T_G be the runtime of the noise reduction algorithm on GPUs and let T_H be the runtime overhead incurred by communication between the device and the host. By offloading the noise reduction to the CPUs and allowing the simulation to continue in parallel, the program runtime should decrease by $T_G - T_H$ for each instance of offloading. If the field sizes don't change, this will be a constant value. The runtime savings should thus also be directly proportional to N_t.

For a first set of results, we run the simulations on one GTX 1080. When offloading is disabled, there is only one MPI rank and the noise reduction is performed on the GPUs. We refer to this as *on-device* noise reduction. When offloading is enabled, the program is run with one additional rank for postprocessing with 10 CPU cores allocated and one OpenMP thread per core. During internal development, the number of CPU cores allocated per rank for benchmarking studies is much greater. We thus acknowledge that the results presented in this section are limited by the available hardware. The machines used in the study are designed primarily for GPU workloads and thus do not have a large CPU core count. To minimize communication costs associated with the offloading, we choose to ensure that all MPI communication is exclusively intra-node and thus restrict the simulation to one node.

Figure 8.10 shows the profiler traces for an example run of the Penning trap simulation with postprocessing offloaded to the CPUs as described above. The profiler data conclusively shows that the postprocessing rank can perform a complex algorithm fully in parallel to the work being performed on the GPUs in another rank.

To ensure consistency and comparability between the runs with and without offloading, we adjust the frequency with which the offloaded tasks are performed on the GPUs. With offloading enabled, postprocessing is only performed for a small portion of all the time steps in the simulation. We thus ensure that the offloaded tasks are performed approximately the same number of times on the GPUs. We note that the number of times the task is performed on the CPUs is a hardware- and application-dependent, unpredictable, and inconsistent parameter. For simplicity, we simply look at the number of instances of noise reduction and emulate the offloading

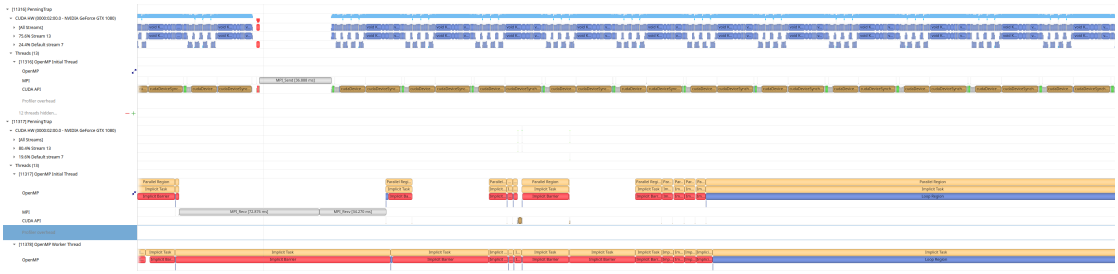

Figure 8.10: Profiler traces for two MPI ranks running the Penning trap simulation. The simulation itself runs on one rank (top half) using one GPU while the second MPI rank (bottom half) only uses the CPU cores and performs noise reduction. Here, we see an instance of data being sent to the second rank for postprocessing. Note that most of the threads are hidden in this screenshot in order to focus solely on the primary threads of each rank. In this run, the noise reduction level is fixed at $\tau = 5$.

frequency; if the CPU ranks receive work for 10% of time steps, then the GPU-only runs perform the offloaded tasks for only 10% of the requested time steps. The regularity results in N_r being slightly higher for runs with on-device noise reduction.

In addition, to maintain consistent dynamics, the results of the noise reduction are discarded in both cases. This means that, when performing the noise reduction on GPUs, there is some overhead incurred by copying the charge density field. This further decreases the margin between runtimes with and without offloading, but we consider this overhead to be negligible for the purposes of this study.

Figure 8.11 shows the runtime of the Penning trap simulation for different values of N_t with and without offloaded noise reduction. The level of noise reduction is fixed for the entire simulation at $\tau = 5$. The data shows that offloading fixed noise reduction does not reduce the GPU workload by a significant amount; at this scale, the offloading would save about 10 seconds for about every 6600 time steps of simulation. Nevertheless, the data does confirm our reasoning and that offloading work can reduce overall runtime under the right conditions.

Figure 8.12 shows the runtime of the same simulation with constant N_t but varying particle density. In this study, the postprocessing rank only runs the noise reduction algorithm. This requires the charge density field after the scatter step and no particle data. The workload on the simulation rank thus increases linearly with the particle count[3] while the workload on the postprocessing rank remains constant. The runtime savings are already on the order of a few seconds per particle per cell.

With these experiments, we conclusively show that it is possible to run complex algorithms in parallel on both CPUs and GPUs. The data confirms that there exists a threshold after which offloading to CPUs provides any desired reduction in total runtime and that the runtime savings can be amplified by increasing the GPU workload in proportion to the CPU workload. While this does not lead to large performance gains in ALPINE, there is significant potential for such gains in a large scale simulation. This project thus lays the foundation for larger studies in the future that could benefit from this feature.

We reiterate here that these results are strongly dependent on the hardware, not only due to the overhead incurred by copying data between the GPU and the host, but also due to the performance of the components themselves. The machine used for these experiments is equipped with GTX 1080s, which NVIDIA classifies as having a compute capability of 6.1 [27]. The throughput

[3] Aside from the solve phase, which depends only on the mesh refinement and not the particle count, all phases in the PIC loop have linear complexity in N_p.

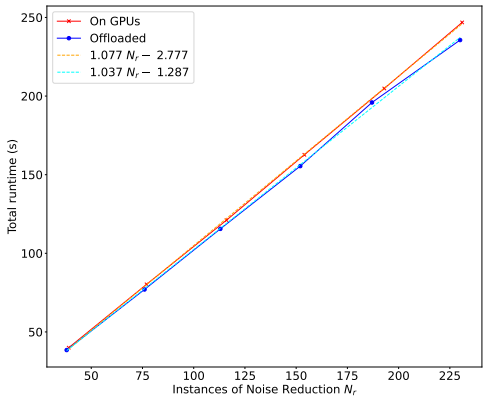

Figure 8.11: Total program runtime with on-device noise reduction compared to offloaded noise reduction with 10 OpenMP threads for postprocessing. The simulation runtime is directly proportional to N_r. This is itself linear in N_t and thus a useful proxy variable. The data confirms that offloading more work to the CPUs saves more time overall, although the benefit is very small under this configuration. The amount of time saved grows linearly with the simulation duration. Here, the noise reduction level is fixed at $\tau = 5$.

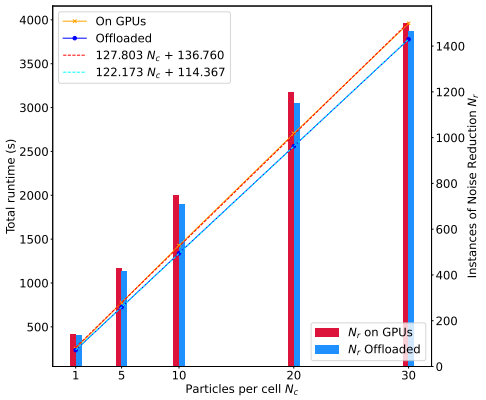

Figure 8.12: The lines show the runtime with different N_c. The bars show the number of instances of noise reduction in the program. The more time steps the GPUs complete before noise reduction is complete on the host, the fewer instances of noise reduction there will be before the simulation is complete. With adaptive noise reduction, a larger workload is being offloaded to the CPUs. The data confirms that increasing the particle density increases work for the GPUs but not the CPUs, resulting in greater runtime savings.

of arithmetic operations on a GTX 1080 with single precision is 32 times higher compared to double precision [12]. If we were to repeat these experiments with various components of the simulation in single precision rather than double, the runtime savings could potentially be up to 32 times smaller than those presented in this report.

To investigate the benefits of mixed execution on faster GPUs, we repeat the same experiment using the newer A100 cards[4]. The host side capabitilies are also enhanced, as the DGX A100 is equipped with 128 CPU cores [18], enabling the use of more threads on the host.

In general, we avoid oversubscribing the CPU cores to ensure that benchmarking data is representative of a workload free of external influences. However, because the A100s have such greater performance than the 1080s, we need significantly many more cores to ensure that the host side computations can keep up with the GPU computations. We make an exception for this run and allocate all 128 cores in the machine to each of the two ranks in order to maximize performance in the postprocessing rank. With multithreading enabled, each core can run two threads. Since the simulation rank will primarily use the GPUs, we believe the performance impact of oversubscribing the cores is negligible.

In Figure 8.13 we see the total program runtime using the same configuration as in Figure 8.12, but using an A100 with 128 OpenMP threads instead of a GTX 1080 and 10 threads. The difference in GPU performance is evident from the data, as the total runtimes are now an order of magnitude smaller. Due to the increased GPU performance, there are only about a third as

[4]The DGX A100 at PSI was only updated to support CUDA 12 towards the end of the project. Most benchmarking tests were performed primarily on the older nodes, despite their inferior performance, before this update became available. The experiments were repeated to obtain additional data about the possible performance gains by offloading work on different machines.

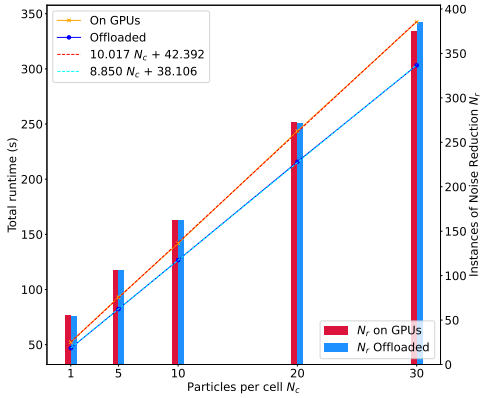

Figure 8.13: The same experiment as in Figure 8.12 is repeated on the DGX A100 with 128 OpenMP threads and a more modern GPU. Offloading work to the CPUs provides greater benefit in proportion, but the absolute savings are much smaller. Due to the speed of the GPUs, the number of instances of noise reduction is also much lower.

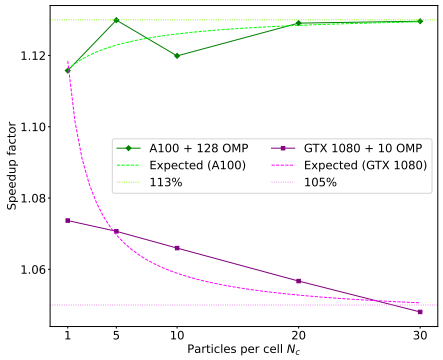

Figure 8.14: Speedup achieved by offloading for the two machines used in this study. Theoretical speedup curves are based on the fit parameters in Figures 8.12 and 8.13. The configuration with the GTX 1080 bears diminishing returns at higher N_c while the speedup with the A100 configuration yields greater speedup with higher N_c.

many instances of offloaded noise reduction compared to the results on the GTX 1080.

We can quantify the speedup using the ratio between the runtimes with on-device and offloaded noise reduction. By using the linear fits shown in Figures 8.12 and 8.13, we can also obtain an analytical predicted speedup curve and compare this to the actual results. The speedup is shown in Figure 8.14. While offloading work to the CPUs does decrease the overall runtime in both cases, the benefit on the GTX 1080 decreases with higher particle density. Based on the fit parameters, we would expect the gains to continue to fall for higher particle densities and asymptotically approach a speedup of 105%. In contrast, the performance gains on the A100 increase with higher particle density and approach 113%.

The nature of the curves in Figure 8.14 can be inferred from the linear fit parameters. Since the speedup is given by the ratio of two linear functions, we have

$$\frac{\mathrm{d}}{\mathrm{d}x}\frac{ax+b}{cx+d} = \frac{ad-bc}{(cx+d)^2} > 0 \Leftrightarrow ad > bc. \tag{8.4}$$

Aside from N_c, all parameters were fixed. The offset parameters b, d then effectively represent the overheads and particle-independent workloads of the simulation while the slope parameters a, c convey the workload which grows with the particle density. It is not immediately clear where these overheads lie; the only changes in the environment between the two tests is the hardware configuration and the thread count on postprocessing rank. We note that the latency of host-device copies are likely different on the two machines. Further investigation may reveal how each of these factors contribute to the diminishing returns in the runs with the GTX 1080.

Part IV

Conclusions

Chapter 9

Summary

9.1 Rank Independence

We successfully achieve rank independence in IPPL by introducing an additional abstraction layer above Kokkos' parallel dispatch and rewriting IPPL algorithms and field operations to accept an arbitrary number of indices. These indices are stored in vectors to bypass limitations of the CUDA compiler, but this design choice has the welcome side effect of simplifying coordinate conversions in the functors. This will significantly simplify future work on alternative meshes and centering methods, which we discuss in further detail in Section 10.1.

Using existing results, we verify that the new implementations of IPPL algorithms and operations are correct in three dimensions. We study the Landau damping and two-stream instability problems in 2D to verify correctness in fewer dimensions.

A scaling study of the new implementations on the Piz Daint cluster reproduces the same benchmarking data as [2]. We thus verify that the additional abstraction layer and added complexity of the kernels do not significantly affect runtime and deduce that compiler optimizations are advanced enough to eliminate the added complexity such that no overheads are introduced in the object code.

The implementation in IPPL is not tied to any range of dimensionalities. In practice, the library can be applied to problems in 1D up to 6D, as these are the dimensionalities supported by Kokkos. Should a future update to Kokkos add support for higher or even arbitrary rank kernels, IPPL will immediately benefit. The code readability can also be further improved if CUDA's restrictions on generic device lambdas are lifted. The flexibility provided by this feature will enable the use of IPPL for a much wider range of problems.

9.2 Mixed Execution Spaces

We supplement the IPPL data types with optional template parameters for specifying additional properties for the underlying Kokkos views. This enables the user to choose different accelerators for each object and make better use of the available resources.

Benchmarking tests highlight the speed difference between CPU and GPU execution and thus the need for larger workloads on GPUs to ensure that offloading work to the CPUs has a significant benefit. The ALPINE logging tests provide an empirical demonstration of the GPU and CPU workloads that can be completed in the same amount of time. With the polling system shown in the noise reduction test, it becomes possible for a program to automatically offload

work to the CPUs at the optimal rate. The data confirms that larger performance gains can be obtained by increasing the GPU workload, indicating that a larger scale simulation can benefit from mixed execution to a greater extent than shown in this project. We reiterate that the results are strongly application- and hardware-dependent.

The simulations with offloaded noise reduction also serve to demonstrate the breadth of the possibilities for this feature. We show that running a highly non-trivial task on CPUs and a PIC simulation on GPUs at the same time is not only possible, but also beneficial to performance under the correct circumstances.

The concept of utilizing multiple accelerators in the same program is comparatively unexplored; this project only constitutes a brief excursion into the possibilities it provides. We find that postprocessing is a promising use case for maximizing resource utilization, since its completion is not required to continue the simulation. There is significant additional work that can be done to explore the extents to which this feature can be used at the application level.

Chapter 10

Future Work

10.1 Coordinate Transformations

The discretization of the physical domain means that field values are only computed at certain points in the domain; these values are then stored in a tensor $F \in \mathbb{R}^{\otimes \dim \mathbb{D}}$. There are several possible methods to choose how the points in physical space relate to the indices, including those shown in Figure 10.1. With *cell centering*, the indices are offset from the physical coordinates by 0.5 along each axis. Currently, only cell centering is available; in this case, the relation between the physical coordinates $\vec{r}$ and the indices $i, j, \ldots$ is given by

$$\vec{r} \equiv \begin{pmatrix} x \\ y \\ \vdots \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ \vdots \end{pmatrix} + \left(\begin{pmatrix} i & 0 & \cdots \\ 0 & j & \ddots \\ \vdots & \ddots & \ddots \end{pmatrix} - n_g I_{\dim \mathbb{D}} + \frac{1}{2} I_{\dim \mathbb{D}} \right) \vec{h}, \tag{10.1}$$

where $\vec{r}_0 \equiv (x_0, y_0, \ldots)$ is the origin of the physical domain, n_g is the number of ghost cells, and I_n is the $n \times n$ identity matrix.

The implementation of rank independent kernels using index vectors makes it convenient to convert between indices and physical coordinates by using vector expressions. In particular, by using expression templates [7], the relation in Equation 10.1 can be expressed as a single assignment in the code. IPPL should eventually support all centering schemes in the original release; thus, it will be necessary to generalize the coordinate transformation expressions. With the changes from this project, the coordinate transformations are now simpler to modify. We expect this to make it easier to implement the other centering schemes in the future.

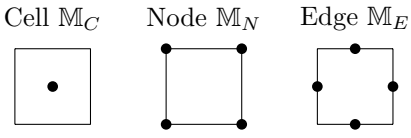

Figure 10.1: Different layouts for field evaluation. Here, the mesh refinement is equal along both axes. The squares represent the mesh cells obtained once the domain is discretized. The side length of the squares corresponds to h. For each layout, the field is evaluated at the locations given by the dots. The next release of IPPL is currently in development and supports only cell centering. Node and edge centering were also supported in the 1.0 release.

10.2 Differential Forms

Most operators in IPPL admit rank independent forms, but the cross product, and thus the curl operator, are only defined in three dimensions. However, it is known that the vector operators are specializations of the exterior derivative for differential forms [28]: if we identify the 1-forms $\mathrm{d}x^i$ with the unit vectors, we have for scalar fields f and vector fields $\vec{F} = (\xi, \eta, \zeta) \in \mathbb{R}^3$ the relations

$$\begin{aligned} \mathrm{d}f = \nabla f &\Leftrightarrow f = f(x, y, z) \\ \star\mathrm{d}\omega = \nabla \times \vec{F} &\Leftrightarrow \omega = \xi \mathrm{d}x + \eta \mathrm{d}y + \zeta \mathrm{d}z \\ \star\mathrm{d}\phi = \nabla \cdot \vec{F} &\Leftrightarrow \phi = \xi(\mathrm{d}y \wedge \mathrm{d}z) + \eta(\mathrm{d}z \wedge \mathrm{d}x) + \zeta(\mathrm{d}x \wedge \mathrm{d}y). \end{aligned}$$

To achieve true rank independence, one might consider implementing differential forms, exterior derivatives, and other related operators. This would make it possible to generalize the cross product and curl. The current implementation simply fails to compile if these operators are applied with $\dim \mathbb{D} \neq 3$.

10.3 C++20 Concepts

The implementation of mixed execution spaces can be refined further. The use of generic lambdas with explicit template parameters for accelerator agnostic operations has already made C++20 a requirement for IPPL. We can thus make use of other new features in the language, specifically *concepts*. The current implementation sometimes imposes strict requirements on data types. This is because a data type using default template parameters is different from one where those parameters are explicitly specified. At the application level, users might omit parameters regarding the execution space and use the default option, but within IPPL, the data types are required to provide the correct parameters. Some interfaces thus enforce explicit parameters, which makes them harder to use at the application level. In the future, IPPL interfaces should be templated directly on their input parameters and use constraints to statically verify compatibility.

10.4 Maximizing Resource Utilization

10.4.1 Performance Across Architectures

With the addition of support for mixed execution spaces, IPPL users are now able to make better use of the resources at their disposal. Most other components of IPPL are designed to be performant and portable, but resource utilization is hardware-specific; it thus becomes the responsibility of the user to empirically determine how to distribute the workload among available accelerators and design their applications accordingly.

The implementation of this feature introduces a dependency on C++20, whose compilation for NVIDIA GPUs requires CUDA 12. Due to its relatively recent release date, not all of the machines used during development immediately supported the new CUDA toolkit. Mixed execution testing was performed mainly on an older, heterogeneous GPU cluster. The nodes are equipped with varying quantities of NVIDIA GeForce GTX 1080s, 1080 Ti's, or RTX 2080 Ti's. These are GPUs designed for games and have much lower theoretical performance for double precision computations compared to single precision [12, 29, 30, 31]. On other machines, the performance gains or losses from the host-device copies could be very different.

At the time of writing, the Swiss National Supercomputing Center (CSCS) is preparing its new "Alps" cluster, whose nodes will be based on NVIDIA's Grace Hopper architecture [32]. The Grace Hopper Superchip provides a direct NVLink connection between the CPU and GPU, providing seven times the bandwidth of a PCIe connection [33]. This is particularly relevant to mixed execution benchmarking; the direct connection would drastically reduce the amount of time needed to copy data from the device to the host, reducing the overhead of CPU offloading. When the Alps cluster becomes available, the next step for mixed execution in IPPL will be to test the limits of this architecture and determine how to fully utilize both host and device computation resources.

By comparing the results from different machines, we will better understand how to adapt IPPL such that end users can fully utilize the resources at their disposal.

10.4.2 Host Side FFTs

Fourier transforms in IPPL are computed using heFFTe, which supports several backends. When computing the transforms on CPUs, the library can use FFTW [34] or MKL. Internal IPPL development uses FFTW in almost all cases. The FFTW library can be compiled to make use of multiple threads [35], but prior testing has shown that the FFTW library used in internal development was not compiled with this option enabled. FFTs are thus performed on a single thread, which drastically limits the potential host side performance and adversely affects the benefits of offloading adaptive noise reduction to the host.

An important next step is to look into the FFTW documentation and ensure that internal IPPL development uses the library with multithreading enabled. The experiments presented in Section 8.5 should then be repeated to confirm that the improved performance of host side FFTs also increases the benefit of offloading the noise reduction algorithm to the CPUs.

Chapter 11

Acknowledgements

I would like to thank my advisors for their continuous support throughout this project and for their valuable insight. I would also like to thank everyone who helped proofread and revise this document.

Credit is also due to Marc Caubet and the rest of the cluster maintenance team at PSI for their hard work. The new node for initial CUDA 12 testing was installed very quickly, making it possible to discover the previously overlooked limitations of the NVCC compiler that hindered the viability of the templated approach to rank independence. The rest of the machines in the GPU cluster were also updated very quickly, making it possible to employ C++20 for mixed accelerator builds of IPPL. Without these updates, this project and the quality of the code would have suffered greatly.

Discussions with the Kokkos team also provided useful information for the design of various implementations from this project. I am grateful for their availability and contributions.

Appendix A

Hypercube Count

Proposition: The total number of m-cubes in an n-cube, for $m \leq n$, is 3^n. That is,

$$\forall n \in \mathbb{N} : \sum_{m=0}^{n} 2^{n-m} \binom{n}{m} = 3^n. \tag{A.1}$$

Proof: We prove the statement by induction. The base case is for $n = 0$; we can see that this is true via geometric reasoning, as a 0-cube is simply a point, which clearly does not contain any other hypercubes. Thus, there is just one hypercube, namely the point itself. This statement is also trivial to prove analytically from the expression, as there is only one summand.

Assume the statement holds for some n. Then, for $n + 1$, we must show

$$\sum_{m=0}^{n+1} 2^{n+1-m} \binom{n+1}{m} \stackrel{?}{=} 3^{n+1}. \tag{A.2}$$

The binomial coefficients satisfy the recurrence relation

$$\binom{n+1}{m} = \binom{n}{m-1} + \binom{n}{m}. \tag{A.3}$$

We can substitute this in Equation (A.2) to obtain

$$\begin{aligned} \sum_{m=0}^{n+1} 2^{n+1-m} \binom{n+1}{m} &= \sum_{m=0}^{n+1} 2^{n+1-m} \left(\binom{n}{m-1} + \binom{n}{m} \right) \\ &= \sum_{m=0}^{n+1} 2^{n+1-m} \binom{n}{m-1} + \sum_{m=0}^{n+1} 2^{n+1-m} \binom{n}{m}. \end{aligned}$$

It should be noted that the $m = 0$ term in the first sum and the $m = n+1$ term in the second sum both disappear, as the terms $\binom{n}{-1}, \binom{n}{n+1}$ do not exist; at the edges of Pascal's Triangle, we have to either interpret Equation (A.3) as defining the non-existant elements as 0, or drop the invalid term in the expression. By replacing the index variable in the first sum with $m' = m - 1$ and dropping the invalid summands in both sums, we find

$$\begin{aligned}\sum_{m=0}^{n+1} 2^{n+1-m}\binom{n+1}{m} &= \underbrace{\sum_{m'=0}^{n} 2^{n-m'}\binom{n}{m'}}_{=3^n \text{ by assumption}} + \sum_{m=0}^{n} 2^{n+1-m}\binom{n}{m} \\ &= 3^n + 2\underbrace{\sum_{m=0}^{n} 2^{n-m}\binom{n}{m}}_{=3^n \text{ by assumption}} \\ &= 3^n + 2 \cdot 3^n \\ &= 3^{n+1}.\end{aligned}$$

Since the proposition holds for $n = 0$, it holds for all $n \in \mathbb{N}$. □

Appendix B

Computing Clusters

For convenience, we summarize the different machines used for the results presented in this project and their configurations. In addition, we discuss their capabilities and how this affected the choice of machine for different tests.

Piz Daint is a cluster of machines at CSCS. There are no plans to update the cluster nodes to CUDA 12, as the cluster is soon to be decommissioned and replaced with the new Alps cluster. It was used for the scaling study for 3D Landau damping in order to reproduce existing benchmarking results. In particular, the study was performed before the dependency on C++20 (and thus CUDA 12) was introduced.

Merlin 6 is a cluster of machines at PSI. For this project, only the GPU partition is relevant. New machines were added to the cluster over time, but the cluster was not subdivided. It is thus a heterogeneous cluster, with GPUs of several different architectures. The cluster consists primarily of 14 older machines. The GPUs on these machines are GTX 1080s, GTX 1080 Ti's, or GTX 2080 Ti's. These machines were updated to support CUDA 12 before this project moved to its second target and were the primary machines used during development and testing of applications running in multiple execution spaces.

In addition to the older machines, the cluster also includes one NVIDIA DGX A100 machine, equipped with 8 A100 cards. It was used extensively during development and testing of the wrapper approach to rank independence. This machine was used for the scaling study in 2D. The 2D study was smaller in scale, as it only served to verify minimal scaling behavior in 2D. The DGX A100 was also used to generate the phase space plots for the two-stream instability simulation. Due to software compatibility issues, this machine was only updated to support CUDA 12 towards the end of the project. Hence, it was not used extensively after the dependency on C++20 was introduced.

We acknowledge that the Merlin 6 cluster also includes one node equipped with eight NVIDIA A5000 cards. This was the first machine in the cluster that supported CUDA 12 and was used extensively during early development and preliminary testing of the templated approach to rank independence, as well as various later stages of this project. However, it was not used for any results presented in this document due to unresolved issues with inter-GPU communication on the node. A proper investigation into the cause of this issue is still pending.

List of Figures

List of Listings

Bibliography

[1] "IPPL Wiki." https://ippl-framework.github.io/ippl/. Accessed 2023-08-28.

[2] S. Muralikrishnan, M. Frey, A. Vinciguerra, M. Ligotino, A. J. Cerfon, M. Stoyanov, R. Gayatri, and A. Adelmann, "Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures," 2022.

[3] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[4] A. Adelmann, C. Kraus, Y. Ineichen, S. Russell, and J. Yang, "The OPAL Framework User's Reference Manual (Draft Copy)," 2009.

[5] A. Vinciguerra, "A performance portable conjugate gradient solver," ETH Unpublished Bachelor's Thesis, 2021.

[6] ISO/IEC, "Working Draft, Standard for Programming Language C++." https://open-std.org/JTC1/SC22/WG21/docs/papers/2020/n4849.pdf, 2020.

[7] J. Progsch, Y. Ineichen, and A. Adelmann, "A New Vectorization Technique for Expression Templates in C++," *CoRR*, vol. abs/1109.1264, 01 2011.

[8] C. K. Birdsall and D. Fuss, "Clouds-in-clouds, clouds-in-cells physics for many-body plasma simulation," *Journal of Computational Physics*, vol. 3, no. 4, pp. 494–511, 1969.

[9] S. Ziavras and N. Haravu, "Processor allocation strategies for modified hypercubes," *IEE Proceedings-Computers and Digital Techniques*, vol. 141, no. 3, pp. 196–204, 1994.

[10] C. Trott. Personal communication.

[11] "CUDA Toolkit 12.0 Released for General Availability." https://developer.nvidia.com/blog/cuda-toolkit-12-0-released-for-general-availability/. Accessed 2023-06-05.

[12] "CUDA C++ Programming Guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed 2023-06-05.

[13] T. M. Inc., "Matlab version: 9.14.0 (r2023a)," 2023.

[14] A. Ho, I. A. M. Datta, and U. Shumlak, "Physics-based-adaptive plasma model for high-fidelity numerical simulations," *Frontiers in Physics*, vol. 6, 2018.

[15] B. Evers, "matplotlib-cpp." https://github.com/lava/matplotlib-cpp, 2021. Accessed 2023-08-16.

[16] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, pp. 90–95, 2007.

[17] CSCS, "Piz Daint." https://www.cscs.ch/computers/piz-daint/. Accessed 2023-09-04.

[18] "NVIDIA DGX A100 Datasheet." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf. Accessed 2021-05-26.

[19] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, p. 357–362, 2020.

[20] Kokkos, "Kokkos documentation." https://kokkos.github.io/kokkos-core-wiki/. Accessed 2023-09-04.

[21] ISO/IEC, "ISO/IEC 14882:2020." https://www.iso.org/standard/79358.html, 2020.

[22] H. H. Fehske, R. R. Schneider, and A. A. Weisse, "Computational many-particle physics," Lecture Notes in Physics, 739, (Berlin, Germany), Springer, 2008.

[23] J. Ahrens, B. Geveci, and C. Law, "ParaView: An end-user tool for large data visualization," in *Visualization Handbook*, Elesvier, 2005. ISBN 978-0123875822.

[24] F. Filbet and E. Sonnendrücker, "Numerical methods for the Vlasov equation," in *Numerical Mathematics and Advanced Applications*, no. 1, pp. 459–468, Springer Milano, December 2012.

[25] C. Birdsall and A. Langdon, *Plasma Physics via Computer Simulation*. CRC Press, 1991.

[26] S. Muralikrishnan, A. J. Cerfon, M. Frey, L. F. Ricketson, and A. Adelmann, "Sparse grid-based adaptive noise reduction strategy for particle-in-cell schemes," *Journal of Computational Physics: X*, vol. 11, p. 100094, 2021.

[27] "CUDA GPUs - Compute Capability." https://developer.nvidia.com/cuda-gpus. Accessed 2023-08-22.

[28] R. W. R. Darling, *Exterior Calculus on Euclidean Space*, p. 24–52. Cambridge University Press, 1994.

[29] "NVIDIA GeForce GTX 1080 Specs." https://www.techpowerup.com/gpu-specs/geforce-gtx-1080.c2839. Accessed 2023-07-05.

[30] "NVIDIA GeForce GTX 1080 Ti Specs." https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877. Accessed 2023-07-05.

[31] "NVIDIA GeForce GTX 2080 Ti Specs." https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305. Accessed 2023-07-05.

[32] CSCS, "Preparing for the Migration from Daint-GPU to Grace-Hopper on Alps." https://www.cscs.ch/publications/tutorials/2023/video-of-the-webinar-upcoming-calls-for-proposals/video-of-the-course-in-situ-analysis-and-visualization-with-paraview-catalyst-and-ascent/video-of-the-webinar-preparing-for-the-migration-from-daint-gpu-to-grace-hopper-on-alps, 2023. Accessed 2023-07-05.

[33] NVIDIA, "NVIDIA Grace Hopper Superchip Architecture In-Depth." https://developer.nvidia.com/blog/nvidia-grace-hopper-superchip-architecture-in-depth/, 2022. Accessed 2023-07-05.

[34] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[35] M. Frigo and S. G. Johnson, "Multi-threaded FFTW." http://fftw.org/fftw3_doc/Multi_002dthreaded-FFTW.html#Multi_002dthreaded-FFTW. Accessed 2023-08-23.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Rank Independence and Mixed Execution Spaces in IPPL

Verfasst von (in Druckschrift):

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Vinciguerra	Alessandro

Ich bestätige mit meiner Unterschrift:

- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Peking, China 2023-09-04	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.