Adaptive Energy Binning in OPALX

Master Thesis

ETH Zurich

written by

Alexander Paul Liemen | alexander.liemen@inf.ethz.ch

supervised by

Dr. Andreas Adelmann

scientific adviser

Dr. Mohsen Sadr

July 30, 2025

Abstract

This thesis presents an adaptive binning algorithm for efficiently computing self-consistent electromagnetic fields in relativistic high-performance particle accelerator simulations, where energy binning is essential for reducing the full MAXWELL equations to the more tractable VLASOV-POISSON equation. Building on the established OPAL code, the work ports, extends, and reengineers the energy binning functionality within the new, GPU-enabled, and performance-portable OPALX framework. In this approach, a fine histogram is first computed from the particle distribution and then optimally merged using a dynamic programming technique that minimizes a custom cost function. This cost function incorporates a term derived from SHANNON entropy, which promotes the least biased (maximum entropy) bin distribution by penalizing imbalances in particle counts. Additional bias terms and optional user-provided parameters fine-tune the merged histogram to balance modeling error against statistical noise. Benchmark results on a shifted GAUSSIAN velocity distribution demonstrate a 60% reduction in field solver computational cost without compromising accuracy. Overall, the adaptive binning strategy significantly enhances simulation efficiency and scalability on both single- and multi-rank CPU and GPU platforms, offering a robust solution for efficient field solvers in high-performance accelerator modeling.

Contents

1 Introduction **5**

2 Problem Setup **7**
2.1 Introduction of the VLASOV Equation . . . 7
2.2 The VLASOV-POISSON System . . . 7
2.3 Application of VLASOV-POISSON Equations in OPALX . . . 8
2.4 The Physics Behind Energy Binning . . . 8
2.5 Literature on Optimal Bin Number . . . 10
2.6 Particle Emission, Histogram Generation and Energy Binning in OPAL . . . 13
2.6.1 Implementation of the Binned Field Solver . . . 14
2.6.2 Implementation Challenges and Problems . . . 15

3 Main Contribution **17**
3.1 Top-Level Implementation Strategy . . . 17
3.2 GPU Enabled Histogram Generation . . . 20
3.3 Particle (Argument) Sorting . . . 22
3.4 Efficient `scatter`/`gather` Operations . . . 24
3.5 Adaptive Histogram Merging Algorithm . . . 25
3.6 Optimal Cost Function . . . 27
3.6.1 SHANNON Entropy . . . 28
3.6.2 Width Bias Term . . . 30
3.6.3 Deviation Penalty Term . . . 31
3.6.4 Sparse Penalty Term . . . 32
3.7 Final Algorithm Implementation . . . 33

4 Results **36**
4.1 Convergence Study and Improvements with Adaptive Binning . . . 36
4.2 Ablation Study on Cost Function Hyper Parameters . . . 37
4.3 Gaussian Test Case in OPALX . . . 41
4.4 Flattop Test Case in OPALX . . . 41
4.5 Scaling Study on the Binning Algorithm . . . 43

5 Conclusion **45**

A Appendix **48**
A.1 Reproducibility and General Compile Instructions . . . 48
A.1.1 Reproduce Plots and Diagrams . . . 48
A.1.2 Binning Code Integration . . . 49
A.1.3 Run `LandauDamping` with Binning . . . 51
A.2 Detailed Algorithm Descriptions . . . 52
A.2.1 `DualView` Histogram Class . . . 52
A.2.2 Binning Variable Selector Concept . . . 55

A.2.3 Pre-Compiled `Kokkos` Reducer Objects 57
A.2.4 Histogram Merging using Dynamic Programming and Parallelization Consideration 60

1 Introduction

In order to design or study particle accelerators, it is essential to simulate these machines to optimize their design and performance, mitigate risks, and improve cost efficiency [8]. Accurate accelerator modeling requires simulating a large number of particles and their interactions both among themselves and with external fields.

One such program is OPAL (Object Oriented Parallel Accelerator Library), developed by the Paul-Scherrer Institute (PSI) and available via their Git repository [1]. OPAL is a well-maintained, widely used, and thoroughly tested software package that meets these simulation requirements. However, it was developed before GPUs became widely available and is parallelized solely using MPI. To address this limitation, a new software package called OPALX is currently under development. OPALX is based on IPPL (Independent Parallel Particle Layer, [16]), a performance-portable library for particle-mesh methods. The goal is to reengineer OPAL so that it scales efficiently on modern supercomputers while taking advantage of GPU acceleration. In this thesis, we focus on porting, improving, and making the energy binning functionality (see section 16.4 in [3]) GPU enabled for integration into OPALX.

A quick overview of energy binning. Energy binning is a quasi-static technique used to compute the self-consistent electromagnetic fields of a relativistic particle bunch with a significant energy (or velocity) spread[1]. Rather than solving the full time-dependent Maxwell equations, the method partitions the bunch into multiple reference frames. Particles are grouped into bins according to their energies (or velocities). For each bin, the space-charge field is computed in a co-moving frame, effectively performing an electrostatic Poisson solve in that bin's rest frame. These electrostatic field solutions are then Lorentz-transformed back to the laboratory frame and superposed, thereby approximating the complete electromagnetic field of the beam. This approach is particularly important in experiments where particles are accelerated non-uniformly and not simultaneously emitted, leading to a large velocity spread within the bunch.

The application-related importance of energy binning. For example, OPAL has been applied at the Argonne Wakefield Accelerator (AWA) facility [17]. In simple terms, a laser pulse is directed onto a target to release electrons, which are then accelerated by an external field (see figure 1). A single simulation timestep (approximately 0.1 ps) is much smaller than the pulse duration (approximately 12 ps). As a result, electrons are emitted successively following a "flattop" temporal distribution [3, 17]. However, within just a few timesteps, relativistic effects become significant, and the standard electrostatic approximation breaks down. At this point, the velocity spread approaches $\sim c$, making energy binning essential.

[1] A reference used by OPAL to justify this technique is provided in [9]. However, this paper only applies binning without providing a rigorous mathematical justification. A more detailed explanation is given in section 2.4.

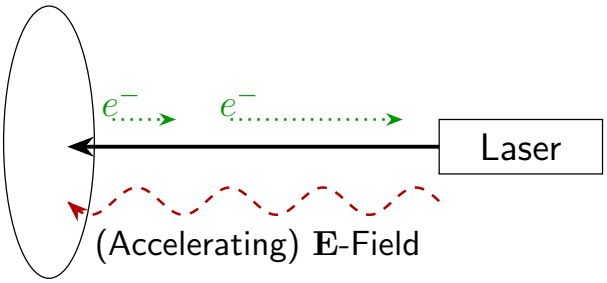


Figure 1: *Simplified emission process as modeled by OPAL for the simulation of the AWA gun [17, p. 2]. Particles are accelerated by the external electric field (red) and emitted through the laser hitting the cathode (disc on the left).*

Previous codes using energy binning. In addition to [9] and OPAL, [18] employs energy binning in combination with a Fast Multipole Method (FMM) to avoid solving the full MAXWELL equations. They concluded that even a small number of bins (4–8) can sufficiently reduce the modeling error. However, this work did not address GPU acceleration or parallelization techniques.

Goals of this thesis. The objective of this thesis is to develop and test a performance-portable, GPU-enabled algorithm for energy binning in conjunction with OPALX or IPPL. This work explores various implementation and optimization strategies to maximize the algorithm's performance while integrating it with OPALX 's existing field solver and Particle-In-Cell (PIC) methodologies. Ultimately, the results will be evaluated through scaling studies on multi rank CPU and GPU configurations.

2 Problem Setup

In plasma physics and accelerator modeling, especially with a focus on OPALX, the behavior of charged particle distributions is described by kinetic equations that account for both external and self-consistent electromagnetic fields. A fundamental framework for this purpose is the VLASOV-POISSON system, which provides a simplified approximate description of the VLASOV equation. One necessary approximation is the assumption of non-relativistic velocities, a requirement that will not necessary anymore when implementing energy binning.

2.1 Introduction of the Vlasov Equation

The dynamics of a collisionless plasma, where particle interactions are dominated by collective electromagnetic effects rather than binary collisions, is governed by the so-called VLASOV equation. For a species of particles with mass m and charge q, the VLASOV equation describes the evolution of the distribution function $f(\mathbf{r}, \mathbf{v}, t)$, representing the density of particles in phase space. The equation reads [7, p. 4]:

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{r}} f + \frac{q}{m}(\mathbf{E} + \mathbf{v} \times \mathbf{B}) \cdot \nabla_{\mathbf{v}} f = 0, \tag{1}$$

where the force acting on the particles was set to the LORENTZ force and where $\mathbf{E}(\mathbf{r}, t)$ and $\mathbf{B}(\mathbf{r}, t)$ are the electric and magnetic fields, respectively.

2.2 The Vlasov-Poisson System

In scenarios where magnetic fields are negligible, and the plasma is non-relativistic with *slowly varying* or *static* external electric fields and no external magnetic field, the VLASOV equation couples with POISSON's equation to form the VLASOV-POISSON system. POISSON's equation for the electric potential ϕ is given by

$$\nabla^2 \phi = -\frac{\rho}{\varepsilon_0}, \tag{2}$$

where ρ is the charge density related to the distribution function by

$$\rho(\mathbf{r}, t) = q \int \mathrm{d}^3\mathbf{v}\, f(\mathbf{r}, \mathbf{v}, t). \tag{3}$$

The electric field $\mathbf{E}$ is then obtained from the potential through $\mathbf{E} = -\nabla\phi$. Substituting $\nabla\phi$ into the VLASOV equation yields the following:

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{r}} f - \frac{q}{m} \nabla\phi \cdot \nabla_{\mathbf{v}} f = 0. \tag{4}$$

The kinetic equation (4) together with the POISSON equation (2) constitutes the VLASOV-POISSON system. It can for example be used to describe electrostatic phenomena in plasmas where relativistic effects – and therefore magnetic effects – are minimal.

2.3 Application of Vlasov-Poisson Equations in OPALX

OPALX is designed to incorporate both external[2] and self-consistent fields. The current program operates under the assumption that external fields are quasi-static, meaning they vary slowly enough that their influence on the self-consistent field[3] is negligible.

This assumption justifies the use of the VLASOV-POISSON system, albeit under the additional assumption that particles move non-relativistically. However, by employing the aforementioned *energy binning* technique, OPAL extends the applicability of the VLASOV-POISSON approximation to scenarios involving relativistic beams with significant energy spreads.

The primary objective of this study is to implement this energy binning functionality in OPALX as well. This approach allows for a more accurate capture of relativistic effects without resorting to the full VLASOV-MAXWELL system. In OPAL, POISSON's equation is solved numerically for each energy bin, and an integrator is used to sum the contributions from both external and self-consistent fields into a single force term within a common reference frame, thereby approximating the full VLASOV-MAXWELL equations[4].

2.4 The Physics Behind Energy Binning

Thus far, we have focused on the application aspects of energy binning and demonstrated that the technique produces reasonable results, as reported in several studies (see section 1). In this section, we provide a more rigorous argument to justify the method by first restating the algorithm and outlining its underlying approximations.

Binning Algorithm. The energy binning algorithm is summarized by algorithm[5] 1. In essence, particles are first grouped into bins according to their velocity (or energy) and then the space-charge field is solved for each bin in its co-moving frame.

Approximations. The algorithm makes several key approximations:

- The velocity spread within each bin is neglected; particles in a bin are assumed to have a common velocity $\mathbf{v}_b$, making them effectively at rest in the bin's co-moving frame.

[2]External fields refer, for example, to the magnetic field generated by a coil module located around the simulation domain. These elements, which can be modeled by OPAL, are an integral part of the particle dynamics.

[3]For example, a rapidly oscillating magnetic field in a coil would induce an electric field observable by the particles. In such cases, the magnetic field is included in the external forces, while its secondary electric field contribution is neglected.

[4]This approximation holds under the assumptions of slowly varying external fields and collisionless particle dynamics.

[5]Note that it uses the full LORENTZ transformation equations in three dimensions, even though the boost will only happen in z direction. This is the direction the particles are accelerated in.

- In the limit $N_b \to \infty$, the discretized (binned) fields converge to the exact fields because the velocity spread $\Delta v \to 0$. This justifies the approximation.
- In the rest frame, $\mathbf{B}'_b = \mathbf{0}$ by the nature of static charge distributions, as dictated by Maxwell's equations in electrostatics.

Charge and Current Density Decomposition. In a beam of particles, the lab-frame charge density and current density are defined as:

$$\rho(\mathbf{x}, t) = \sum_{i=1}^{N} q_i \, \delta\left(\mathbf{x} - \mathbf{x}_i(t)\right), \qquad \mathbf{J}(\mathbf{x}, t) = \sum_{i=1}^{N} q_i \, \mathbf{v}_i \, \delta\left(\mathbf{x} - \mathbf{x}_i(t)\right) \tag{5}$$

or, in a continuum formulation,

$$\rho(\mathbf{x}, t) = \int q \, f(\mathbf{x}, \mathbf{v}, t) \, \mathrm{d}\mathbf{v}, \qquad \mathbf{J}(\mathbf{x}, t) = \int q \, \mathbf{v} \, f(\mathbf{x}, \mathbf{v}, t) \, \mathrm{d}\mathbf{v}, \tag{6}$$

where $f(\mathbf{x}, \mathbf{v}, t)$ is the particle distribution function. For energy binning, we decompose the charge density over the beam velocity (assumed to be primarily along the z-axis). Let v denote the scalar velocity in the beam direction. We define the conditional charge density $\rho_v(\mathbf{x}, v, t)$ for particles in the velocity range $[v, v + \Delta v]$ as:

$$\rho_v(\mathbf{x}, v, t) = q \int_{\Omega_v} f(\mathbf{x}, \mathbf{w}, t) d\mathbf{w}, \quad \Omega_v = \{\mathbf{w} : v \leq \|\mathbf{w}\| \leq v + \delta v\}. \tag{7}$$

The total charge density then becomes:

$$\rho(\mathbf{x}, t) = \int \rho_v(\mathbf{x}, v, t) \, \mathrm{d}v. \tag{8}$$

The charge density then would be defined in the same way, just with another factor w in the integrand of equation (7). And similarly, since the Maxwell equations are linear, the electric field may be decomposed as

$$\mathbf{E}(\mathbf{x}, t) = \int \mathbf{E}_v(\mathbf{x}, t) \, \mathrm{d}v. \tag{9}$$

Discretizing the velocity space into N_b bins, we approximate the integrals by sums:

$$\rho(\mathbf{x}, t) \approx \sum_{b=1}^{N_b} \rho_b(\mathbf{x}, t), \qquad \mathbf{E}(\mathbf{x}, t) \approx \sum_{b=1}^{N_b} \mathbf{E}_b(\mathbf{x}, t), \tag{10}$$

where $\rho_b(\mathbf{x}, t)$ represents the contribution from particles with velocity in the range corresponding to bin b.

Lorentz Transformation of Fields. For each bin b, we solve the electrostatic Poisson equation in the rest frame:

$$\nabla'^2 \Phi'_b(\mathbf{x}') = -\frac{\rho'_b(\mathbf{x}')}{\varepsilon_0}, \qquad \mathbf{E}'_b(\mathbf{x}') = -\nabla' \Phi'_b(\mathbf{x}'), \tag{11}$$

with the assumption $\mathbf{B}'_b = \mathbf{0}$ since static charges do not produce a magnetic field. The full[6] LORENTZ transformation equations from the rest frame (where the rest frames is given by the primed quantities) to the lab frame (unprimed quantities) for a boost with velocity $\mathbf{v}_b$ are

$$\mathbf{E}_b = \gamma_b \left(\mathbf{E}'_b + \mathbf{v}_b \times \mathbf{B}'_b\right) - \frac{\gamma_b^2}{\gamma_b + 1} \mathbf{v}_b \frac{\mathbf{v}_b \cdot \mathbf{E}'_b}{c^2},$$
$$\mathbf{B}_b = \gamma_b \left(\mathbf{B}'_b - \frac{\mathbf{v}_b \times \mathbf{E}'_b}{c^2}\right) - \frac{\gamma_b^2}{\gamma_b + 1} \mathbf{v}_b \frac{\mathbf{v}_b \cdot \mathbf{B}'_b}{c^2}.$$

Since we assume $\mathbf{B}'_b = \mathbf{0}$ in the rest frame, these equations reduce to

$$\mathbf{E}_b = \gamma_b \mathbf{E}'_b - \frac{\gamma_b^2}{\gamma_b + 1} \mathbf{v}_b \frac{\mathbf{v}_b \cdot \mathbf{E}'_b}{c^2}, \tag{12}$$
$$\mathbf{B}_b = -\gamma_b \frac{\mathbf{v}_b \times \mathbf{E}'_b}{c^2}. \tag{13}$$

Therefore, a magnetic field in the lab frame arises solely from the Lorentz boost.

Convergence Argument. As $N_b \to \infty$, the width of each velocity bin $\Delta v \to 0$, and the binned approximations in equation (10) converge to the exact integrals in equations (8) and (9). By the linearity of Maxwell's equations, the discrete sum of the individual fields $\mathbf{E}_b$ converges to the total field $\mathbf{E}(\mathbf{x}, t)$, ensuring that the energy binning algorithm converges to the correct solution.

In summary, energy binning is justified by decomposing the total charge density (and consequently the fields) into contributions from particles in different velocity ranges. In each bin, the particles are assumed to be nearly at static, so that in the bin's rest frame the fields can be obtained from solving the electrostatic POISSON equation. The fields are then transformed back to the lab frame using the complete LORENTZ transformation equations (equations (12) and (13)). The static nature of the charge distribution in the rest frame (i.e., $\mathbf{B}'_b = 0$) is well justified by electrostatics, and as the number of bins increases, the binned solution converges to the exact solution. Thus, the accuracy of the energy binning algorithm depends primarily on the bin configuration and the accuracy of the underlying POISSON solver. Again, the final algorithm can be seen in 1.

2.5 Literature on Optimal Bin Number

The number of bins used in energy binning significantly influences both the computational cost and the accuracy of the solution. OPALX must run a separate routine to solve POISSON equation for each bin, which constitutes the primary computational expense[7]. Although POISSON equations corresponding to bins are independent from one another, solving them in parallel would require storing independent field instances that

[6] Full meaning, they are not simplified to one dimension.
[7] This will be evident in the timing plots presented in section 4.

Algorithm 1 Energy Binning Algorithm for Space-Charge Field Computation for N Particles with Position, Velocity, and Charge (in SI units). Note that the actual implementation might look different to account for memory optimizations.

Input: Particle distribution $\{x_i, v_i, q_i\}_{i=1}^{N}$.
Parameters: Histogram configuration with N_b bins.

Step 1: Bin Particles by Velocity.
Define bin edges $\{v_b\}_{b=1}^{N_b+1}$.
for each particle i **do**
    Assign particle to bin b such that $v_b \leq v_i < v_{b+1}$.
end for

Step 2: Solve the Electrostatic Problem in Each Bin's Rest Frame.
for each bin b **do**
    Compute Lorentz factor: $\gamma_b = 1/\sqrt{1 - \frac{v_b^2}{c^2}}$.
    Transform the lab-frame charge density $\rho_b(\mathbf{x}, t)$ to the rest frame obtaining $\rho_b'(\mathbf{x}')$.
    Solve Poisson's equation $\nabla'^2 \Phi_b' = -\frac{\rho_b'}{\varepsilon_0}$ to obtain $\mathbf{E}_b' = -\nabla' \Phi_b'$.
end for

Step 3: Transform Fields Back to the Laboratory Frame.
for each bin b **do**
    Use the Lorentz transformation:

$$\mathbf{E}_b = \gamma_b \left(\mathbf{E}_b' + \mathbf{v}_b \times \mathbf{B}_b'\right) - \frac{\gamma_b^2}{\gamma_b + 1} \mathbf{v}_b \frac{\mathbf{v}_b \cdot \mathbf{E}_b'}{c^2},$$

$$\mathbf{B}_b = \gamma_b \left(\mathbf{B}_b' - \frac{\mathbf{v}_b \times \mathbf{E}_b'}{c^2}\right) - \frac{\gamma_b^2}{\gamma_b + 1} \mathbf{v}_b \frac{\mathbf{v}_b \cdot \mathbf{B}_b'}{c^2}.$$

end for

Step 4: Sum the Field Contributions from all Bins.
Initialize $\mathbf{E}_{\text{total}} = \mathbf{0}$ and $\mathbf{B}_{\text{total}} = \mathbf{0}$.
for each bin b **do**
    $\mathbf{E}_{\text{total}} \leftarrow \mathbf{E}_{\text{total}} + \mathbf{E}_b$.
    $\mathbf{B}_{\text{total}} \leftarrow \mathbf{B}_{\text{total}} + \mathbf{B}_b$.
end for
Return: $\mathbf{E}_{\text{total}}, \mathbf{B}_{\text{total}}$.

exceed available memory in large systems[8]. Thus, there is a clear incentive to use more bins to enhance accuracy, while simultaneously minimizing the number of bins to reduce computational cost and noise.

Previous studies, such as [9] and [18, p. 4245], have shown that using relatively few bins can significantly improve accuracy compared to single-frame electrostatic approximations. In particular, [18] conducted convergence studies demonstrating that errors in their example decrease substantially with just a few bins, as illustrated in figure 2.

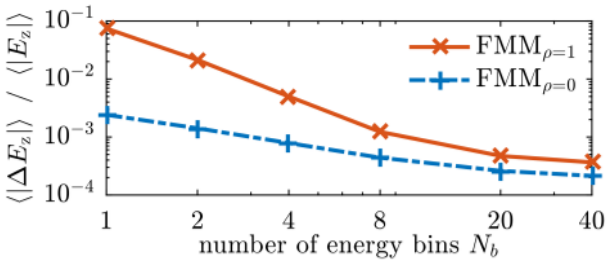


Figure 2: *Convergence study from [18, p. 4245]. Here, ρ is a parameter that controls the phase space correlation in the initial distribution, and E_z is the electric field component computed by their solver.*

This convergence behavior will be further validated in section 4.1, where additional binning technique considerations are discussed.

For the implementation of the binning algorithm, it is essential to define what constitutes an "optimal" binning configuration. The following guidelines are proposed to outline the criteria for the algorithm and the final histogram:

- The algorithm must be capable of partitioning the initial particle distribution into several distinct bins.
- It should group particles in a manner that produces a balanced histogram. An almost empty bin, which would introduce noise in the right hand side of POISSON equation and necessitate a full solver routine, despite contributing negligibly to the final field. Such bins should be merged with adjacent neighbors.
- The algorithm should be user-configurable, allowing adjustments to favor smaller or larger bins, thereby controlling the modeling error as needed.
- It must be robust with respect to variations in the initial distribution; its behavior should remain consistent whether the distribution is GAUSSIAN or clustered at one end of the value range.

[8]This limitation arises from the current implementation of the solvers in IPPL. While computation time is manageable for small simulation grids, larger grids can quickly become memory-bound. For instance, a 2048^3 grid of 64-bit floating point numbers occupies roughly 64 GB of memory. The potential time savings from utilizing 100% of the available concurrency do not justify the extra memory footprint, even on modern hardware.

- The algorithm should significantly reduce the number of bins when most particles have velocities close to c (the speed of light), as is typical in some particle accelerators.
- Since the number of particles may vary throughout the simulation, the algorithm must operate without requiring the total number of particles to be specified in advance.
- Ultimately, the goal is to minimize computational cost by reducing the number of bins[9] while still maintaining a reasonably accurate self-consistent field.

2.6 Particle Emission, Histogram Generation and Energy Binning in OPAL

This section will explain how OPAL handles binning and where the old approach needs refinement.

OPAL uses the `Distribution` class[10] to manage all particle emissions both before and during runtime. Figure 3 illustrates how OPAL handles the emission process and histogram generation.

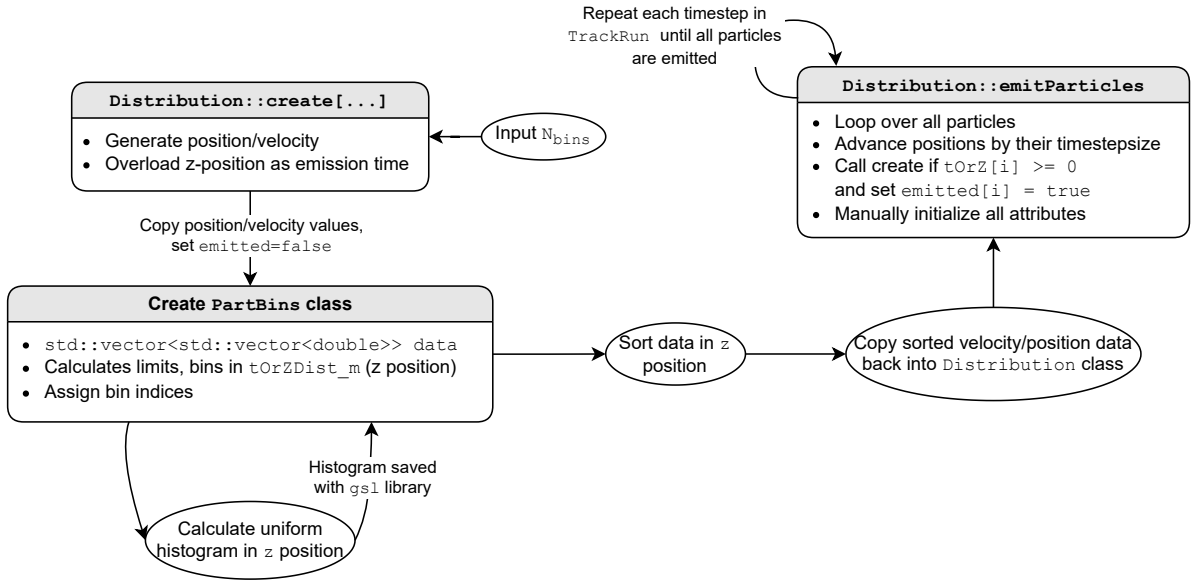


Figure 3: *Conceptual design of the binning routine in OPAL. The diagram shows the relationship between the* ***PartBins*** *class, which creates the histogram configuration, and the* ***Distribution*** *class, which is responsible for generating the position/velocity values and for injecting particles into the simulation during runtime.*

The user specifies a fixed number of bins, N_{bins}, which remains constant throughout the

[9] The POISSON solvers used in OPALX are grid-based and therefore do not necessarily scale with the number of particles once particles are scattered onto the grid. Consequently, nearly empty bins lead to an unnecessary performance penalty.

[10] Located under `src/Distribution/` in [1]. The importance of injecting particles into the simulation during runtime is explained in section 1.

simulation. OPAL first generates particle positions and velocities in the `Distribution` class and then initializes the `PartBins` class by explicitly copying this data into a two-dimensional `std::vector` within the `PartBins` instance. After this initialization, three key aspects are considered:

Binning Parameter. First, we must decide which variable is most appropriate for binning. Although one could compute each particle's energy explicitly, our setup allows us to simplify the process. Assuming that significant relativistic effects occur only in the z (beam) direction and that this component is unaffected by inter-particle interactions, we consider a single particle in a homogeneous electric field $\mathbf{E} = E_0\,\hat{\mathbf{e}}_\mathbf{z}$.

If the particle starts at rest (i.e., position and velocity are zero, $z = v = 0$), the work done by the electric field is

$$W = \text{force} \cdot \text{displacement} = qE_0 z,$$

where q is the particle's charge and m its mass. Since W equals the kinetic energy, and the total energy is given by $E = mc^2 + E_{\text{kin}}$, setting $E_{\text{kin}} = W$ yields

$$E = mc^2 + qE_0 z.$$

This proportionality between energy and z suggests that binning should be performed along the z direction, which is also the direction in which the beam is accelerated.

Histogram Generation and Binning. The histogram is constructed by dividing the range of z components into N_{bins} equally spaced intervals and assigning each particle a corresponding bin index. The `GSL` library is then used to generate the histogram from the list of bin indices, and finally, the particles are sorted based on z.

Particle Emission during Runtime. The third position component is interpreted as the emission time prior to particle injection. Specifically, this value is initially sampled as a negative time, representing the delay before the particle is emitted[11]. The emission time is updated at every timestep until it becomes positive, at which point the particle is injected into the particle bunch along with its other attributes (e.g., mass and charge). Additionally, since timesteps are discrete while the emission time may fall between timesteps, each particle is assigned a `timestepsize` attribute. This attribute determines the time increment applied to the particle in the first timestep after injection.

2.6.1 Implementation of the Binned Field Solver

Generating the bin indices and using it to inject particles into the simulation is only the first part. This step is finished after all particles are injected and only exists to accurately resolve the emission laser depicted in figure 1. However, another use of bins is in the field

[11] This is where the `Flattop` distribution is applied in the AWA example described in section 1.

solver, as explained in 2.4. Algorithm 2 depicts how the self-consistent electromagnetic fields of the particles are calculated.

Algorithm 2 Binned Self Consistent Electrical Field Computation in OPAL.

1: $\mathbf{E}_f \leftarrow 0$
2: **for** each bin **do**
3: **for** each particle **do**
4: Set particle's charge to 0 if outside the current bin.
5: **end for**
6: Scatter *all* charges onto a grid.
7: Scale charge density respecting LORENTZ transformation to bin's rest frame.
8: Solve POISSON equation on the grid to compute the field $\mathbf{E}'_{\text{bin}}$.
9: Transform $\mathbf{E}'_{\text{bin}}$ to the lab frame and update total field:

$$\mathbf{E}_f \leftarrow \mathbf{E}_f + \text{TransformToLabFrame}(\mathbf{E}'_{\text{bin}}, \mathbf{B}'_{\text{bin}} = 0)$$

10: Reset particle charge attribute to "charge per particle".
11: **end for**
12: Gather the total $\mathbf{E}_f$ onto the particles.

By comparing algorithm 1 and 2, we notice that OPAL technically solves the field for all particles N_{bin} times. However, if we set the charge values outside each bin to zero, the contribution to the field the particles are scattered on, is zero. Therefore, the solution is identical as if we had only scattered the particles in each bin.

2.6.2 Implementation Challenges and Problems

The implementation of OPAL's binning routine as explained by figure 3 and algorithm 2 lead to some problems impacting performance and memory usage that are summarized in the following.

- Data copying is slow and unnecessary when the data remains unchanged.
- Copying data into `PartBins` and then back to the `Distribution` class leads to inefficient memory usage since the data itself is not changed.
- Scattering every particle instead of only particles per bin results in a computational cost that scales linearly with the number of bins, i.e., it takes N_{bin} times longer than necessary. In optimal conditions, we would only have to iterate once through the particles, since one particle can only be part of one bin.
- Bin indices may be recalculated during simulation while the number of bins does not change. In later simulation stages, when most particles are nearly at c, fewer bins

are required, leading to up to $N_{\text{bin}} - 1$ unnecessary[12] POISSON solver calls.

- Most importantly, the old OPAL implementation is not GPU-enabled, which limits the performance significantly for processes that iterate for example through a large numbers of particles in the bunch.
- Bin in velocity instead of position (energy). Ignoring self fields, the energy spread in a particle bunch will not change after some time in the simulation, only the mean energy increases. Therefore, we have no indicator of when it is possible to decrease the number of bins when particles are all close to light speed. Therefore, we use velocity as a binning variable. However, the new algorithm provides further flexibility with respect to the binning strategy and binning variable as explaiend in A.2.2.

This thesis aims to optimize these points in OPALX by introducing GPU enabled computation, no particle attribute copy operations, no unnecessary `scatter` calls and the automatic optimization of N_{bin} in each timestep. A short description of the proposed method is provided in figure 13, which will be discussed in detail in section 3.

[12] "Unnecessary" meaning that additional bins do not lead to a significant accuracy gain. It does not refer to bins being empty.

3 Main Contribution

The following section presents the key considerations underlying the design of the final binning algorithm in OPALX. Each component of the algorithm is introduced sequentially, with relevant comparisons to alternative approaches where appropriate. The rationale for specific design decisions is discussed, particularly with respect to computational and memory efficiency within OPALX. A complete explanation of the algorithm is provided in section 3.7.

3.1 Top-Level Implementation Strategy

Before implementing the histogram generation and binned field solver, we must decide how to store bin indices in a manner that guarantees fast computation without wasting memory. The design of a new binning class must address the following points:

- Identify whether a particle belongs to a given bin.
- Work directly with the particle bunch or shallow copies (references to the original data arrays without duplicating the actual data) to avoid the overhead of copying data between classes.
- Efficiently update the bin index when necessary.
- Incorporate newly emitted particles into the binning routine seamlessly.
- Either set charges to zero for particles outside a specific bin while preserving their original charge[13] or provide an efficient way to iterate only over particles within a specific bin.

Based on these requirements, there are two primary implementation strategies:

Maintain Particle Index Array. In this approach, an index array is maintained along with a separate histogram array that stores particle indices rearranged by bin index. This index array maps ranges of particle indices to specific bins, enabling efficient scatter calls. The drawback is that the index array must be extended or rebuilt every time a particle is added, and sorting the indices can become computationally expensive.

Modify the Particle Container. Here, the bin index is stored directly as an additional attribute in the particle container. This eliminates the need to rearrange the particle bunch when bin indices change. However, iterating over particles within a single bin becomes challenging and may require employing the same strategy as OPAL (see algorithm 2).

Both approaches are schematically outlined in figure 4.

[13] This is necessary because particle charges may not be the same throughout the bunch.

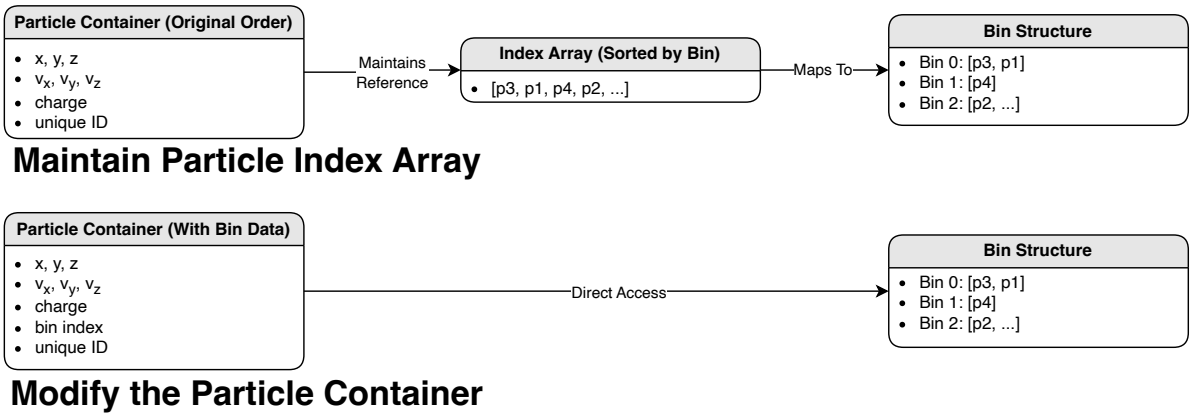


Figure 4: *Schematic comparison of two particle binning strategies: (top) maintaining a separate index array sorted by bin for efficient bin access, and (bottom) storing the bin index directly in the particle container for direct but less structured access to bin membership.*

The first approach is only viable if the ordering of particles remains unchanged throughout the simulation[14]. Since this cannot be assumed, the first approach is less favorable.

Furthermore, building an index array is non-trivial: one must either maintain a separate array to store bin indices or sort an existing index array based on the binning variable. Without a dedicated bin index array, sorting is limited to $\mathcal{O}(n \log n)$, where n is the number of particles. With an index array, however, the computational complexity is $\mathcal{O}(n)$ as described in section 3.3. Building such an array also facilitates the second approach.

Even when using the second approach, efficiently iterating through bins remains a challenge. To avoid unnecessary `scatter` calls, as discussed in section 2.6.2, it is advantageous to also maintain an index array containing the argument sort of the particle indices. This approach enables $\mathcal{O}(n)$ sorting and efficient iteration through bins, as detailed in section 3.4. The class diagram of the new `AdaptBins` class for OPALX is illustrated in figure 5.

[14]However, this assumption may not hold in the future. There is ongoing discussion in IPPL about sorting particles for access pattern optimization in other processes, which – although unrelated to the binning routine – could interfere with this approach.

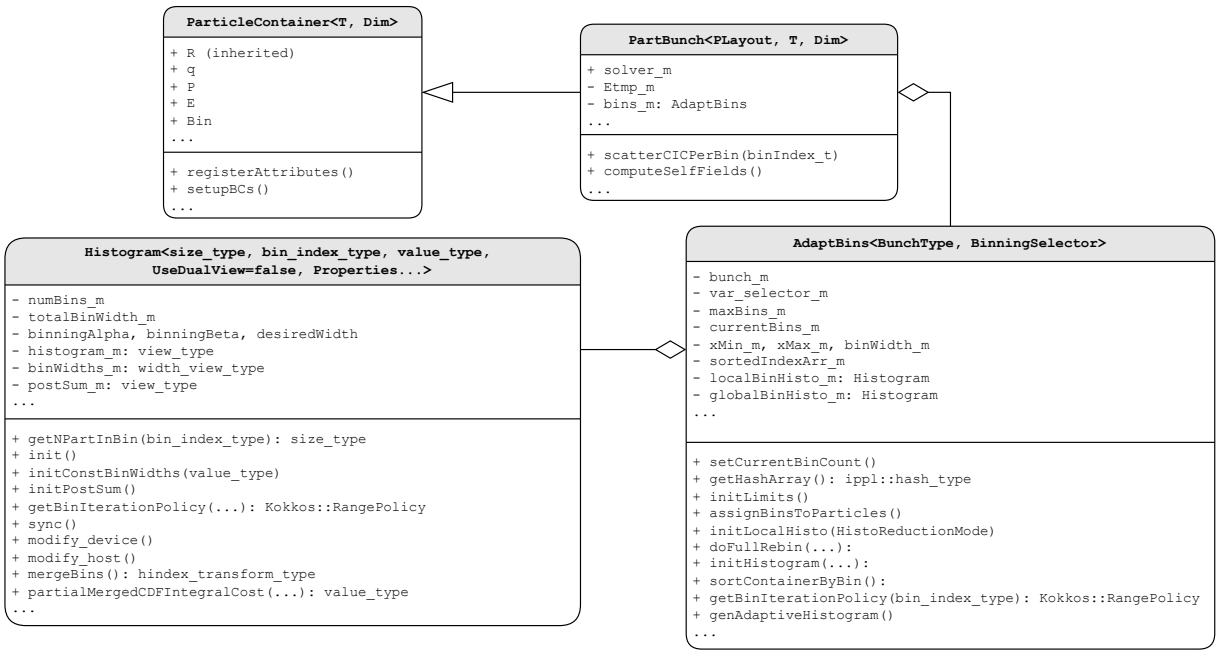


Figure 5: *Class diagram showing the relationships between* ***`ParticleContainer`****,* ***`ParticleBunch`****, and the newly created* ***`AdaptBins`*** *and* ***`Histogram`*** *classes. The diagram has been simplified to emphasize the new functionality. The* ***`ParallelTracker`*** *instance calls* ***`computeSelfFields`*** *on the* ***`PartBunch`*** *instance, which in turn invokes all functions necessary to run the binned field solver.*

The class diagram reflects the following design choices:

- `ParticleContainer` stores a particle attribute `Bin` representing each particle's bin index.
- `PartBunch` maintains an instance of `AdaptBins` and a temporary electric field ($\mathbf{E}$) for the binned field solver.
- `AdaptBins` contains an `ippl::hash_type` member named `sortedIndexArr_m`, which holds the sorted particle index array. This array is initialized using the `sortContainerByBin()` function.
- `AdaptBins` provides a function `getBinIterationPolicy` that enables iteration over particles in a specific bin.
- `AdaptBins` includes a function `genAdaptiveHistogram` that merges the global histogram as explained in section 3.5 and as required by the improvements discussed in section 2.6.2.
- The histogram data is stored in an instance of the `Histogram` class (detailed in section A.2.1), which retains information on counts and bin widths to support non-uniform histograms.

This design defines how the data required for the binned field solver and for bin assignment

is stored. The primary memory overhead in OPALX arises from the bin index attribute and the sorted particle index array, both of which scale linearly with the number of particles. The index array uses the type `ippl::hash_type`, which employs `int` values[15]. In contrast, the bin attribute in the `ParticleContainer` can be manually set; it is currently defined as a `short int` to reduce memory usage compared to a standard `int`.

3.2 GPU Enabled Histogram Generation

Generating a histogram from a distribution of particles where we already know the bin indices is conceptually a trivial task. One has to iterate through the bin index array and increment a counter each time a bin index is met. However, this is only the optimal way when working in serial. As soon as parallelism is introduced, we run into race conditions while updating the histogram counts. Here we look at three different possibilities of implementing a histogram reduction in parallel:

Strict atomic operations. Using `Kokkos::atomic_increment`, one can make sure that only one thread at a time writes into a variable. The problem is that in the best case, only N_{bin} threads can increment a number simultaneously. For the worst case this means that the maximum achievable performance is reached after N_{bin} threads. More threads will not decrease computation time.
It is worth mentioning that newer GPU generations try to improve atomic operations more and more, making it easier for programmers to implement algorithms like these [4]. But even on GPU, the atomic performance is not the best performance achievable as seen in figure 6b.

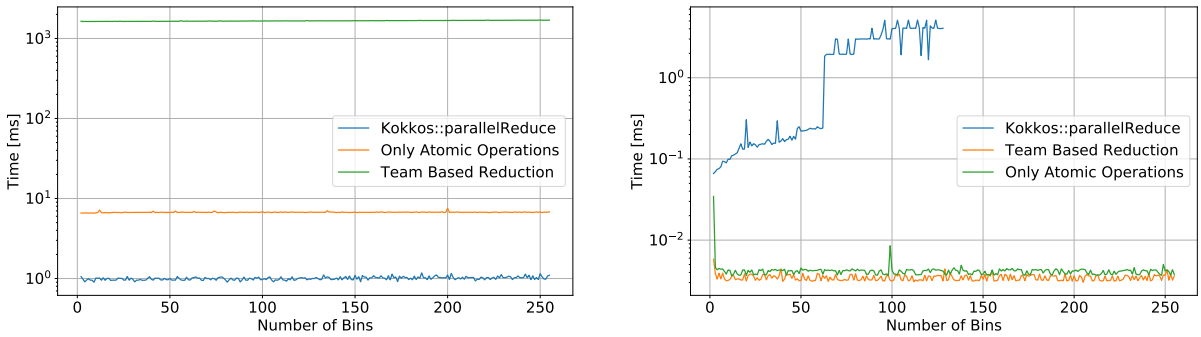
`Kokkos::parallel_reduce` on an array. This method generates multiple histogram copies and employs a hierarchical reduction approach to combine the results. This design allows individual threads to populate their respective local histograms independently, while a designated thread subsequently merges them without the need to handle atomic memory operations. When atomic operations constitute a performance bottleneck, particularly in CPU-based implementations, this approach represents the optimal solution.

Hirachical parallelism: `Kokkos::TeamPolicy`. This is a combination of the two previous reduction algorithms. `Kokkos` allows grouping threads together into teams. Each team gets its own range of particles to iterate over and builds its own local histogram using atomics in the GPUs scratch memory. Then the team local histograms are combined, again, using atomics. The atomics collision probability decreases overall, since there are only a few threads writing at the same time. The standard number of threads `Kokkos` tries to use per team on a Nvidia A100 is 128[16].

[15] This requirement stems from IPPL's communicator when `hash_type` is used to identify particles that need to be communicated between ranks.

[16] A team size of 128 threads is chosen because it matches four warps on Nvidia GPUs, which ensures high occupancy and efficient use of the hardware [13, p. 21]. This size also strikes a practical balance

Figure 6 shows how these three algorithms perform with different number of bins. Plot 6a



(a) Timings run on 16 CPU Cores on merlin6.

(b) Timings run on one Nvidia A100 GPU on gwendolen.

Figure 6: *Logarithmic timing comparison for different methods used to build a histogram. Each x axis shows the number of bins, the y axis the time the corresponding algorithm needed to generate the whole histogram. The algorithm was run using 10^6 particles.*

clearly shows that `Kokkos::parallel_reduce` is far superior for all histogram sizes that will be relevant in OPALX. The problem with the team based approach on CPU is the missing concurrency. Since there are not enough threads available, we will basically get `parallel_reduce` but without a hierarchical reduction step. The atomics approach also behaves as expected, since atomic operations are usually slow on CPU.

The GPU plot 6b shows that `parallel_reduce` becomes very inefficient at higher bin counts[17]. The reason is that it duplicates the data[18] in GPU memory which quickly becomes expensive. The big jump in the blue line happens at exactly 63 bins, which could be the point where the histogram does not fit into the register anymore. Only for very small bin numbers[19], it can be efficient enough to be usable.
The atomics approach is very consistent and performs quite well, which confirms that GPUs handle atomics generally well. The only exception is at very small bin numbers. For one bin, we have basically single thread performance. The team-for approach handles this better, since the data duplication per team leads to way less atomic collisions. Both methods should improve for larger bin counts, since that decreases atomic collision probability. However, the team-based approach outperforms pure atomics on average by 26% on the test setup from figure 6b.

between minimizing atomic contention and making effective use of shared memory. While `Kokkos::Auto` would select a similar value internally, explicitly specifying 128 allows for deterministic partitioning of particle ranges, which is a big advantage for this implementation.

[17]Note that the blue line in the right plot stops at 128 bins. The reason is simply that this is the reduction that uses pre-compiled reducer objects as explained in A.2.3. The more bins we want to be able to calculate, the more objects need to be created at compile time, which quickly becomes a not insignificant time.

[18]Since it needs to generate as many histogram instances as there are available threads or particles.

[19]E.g. at most ≥ 5 as indicated by figure 6b where the green blue line goes down while the green/orange lines go up.

To select the optimal histogram reduction method, we implement a function that considers both the execution environment and the number of bins. For CPU execution, `parallel_reduce` is always preferred. On GPU, `parallel_reduce` is favored for very small bin counts (as indicated by `maxArrSize` in the code and in figure 7), while for larger bin counts, the team-based approach is generally selected. This selection logic is illustrated in Diagram 7.

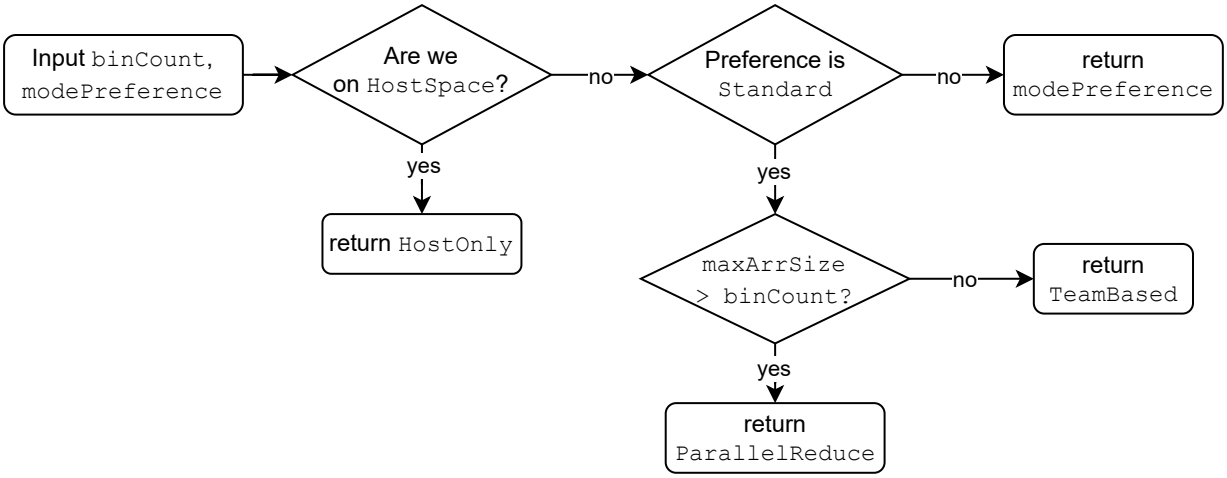


Figure 7: *Decision tree to decide that decides which of the histogram reduction methods described in 3.2 is the most suitable.*

Additionally to the preference towards the quickest method, it allows the user to favor a method by providing a `modePreference` value of the type `HistoReductionMode`. If the execution of `modePreference` is not possible, it will select the next best method.

Laslty, we can adress the difficulty of running `parallel_reduce` on GPU. `Kokkos` requires a static array (not dynamically allocated, since that is not possible inside a `Kokkos` kernel), but this is not possible if the number of bins is not known at compile time. One could simply initialize and use a larger array. However, this would mean that the whole advantage of small bin counts is lost. The solution is to use recursive variadic templates and a selector function to generate several variants of the reduction object, which can be accessed during runtime. A more detailed explanation of the code can be found in section A.2.3.

3.3 Particle (Argument) Sorting

As explained in section 3.1, we need to be able to efficiently iterate through particles per bin. The solution is to maintain an index array in the `AdaptBins` class that corresponds to the argument sort of the particle bunch, and the bin index is the key to be sorted after. Standard comparison-based sorting algorithms, which operate without prior knowledge of data distribution, achieve an optimal computational complexity of $\mathcal{O}(n \log n)$, where n is the number of particles. However, since we know that the set of the sorting keys contains discrete integers in a known range (meaning key $\in \{1, ..., N_{\text{bin}}\}$) and we already have the histogram computed, we can utilize bin sort. Bin sort only needs the postfix sum of the

histogram and one iteration through the particles to generate the index array. This has the complexity of $\mathcal{O}(N_{\text{bins}} \cdot n)$ operation. The relevant part of the algorithm can be seen in the following:

```
Kokkos::View<size_type*> bin_offsets = getPostSum(localBinHisto_m);

Kokkos::parallel_for("InPlaceSortIndices", localNumParticles,
   KOKKOS_LAMBDA(const size_type& i) {
       size_type target_bin = bins(i);
       size_type target_pos = Kokkos::atomic_fetch_add(&bin_offsets(target_bin), 1);
       indices(target_pos)  = i;
});
```

The only problem is that the algorithm needs one atomic operation to update the post sum every time a particle is found in the corresponding bin. We already saw in 3.2 that atomics on modern GPUs are very efficient.

Accessing the elements in a sorted manner will only be necessary inside the `scatter` routine after the histogram is merged. The impact from atomic collisions gets worse with less number of bins (collision probability increases). Meaning that it is not a good idea to sort the particles right before it is needed, but directly after the fine histogram is generated. More bins means less atomic collision probability, which will result in a faster execution time. Figure 8 shows execution time on GPU against the number of bins. We did not carry out this experiment on CPU, mainly because the number of available threads is not often above the number of bins (usually 128 for the fine histogram), meaning that the atomics collision effect does not come into effect.

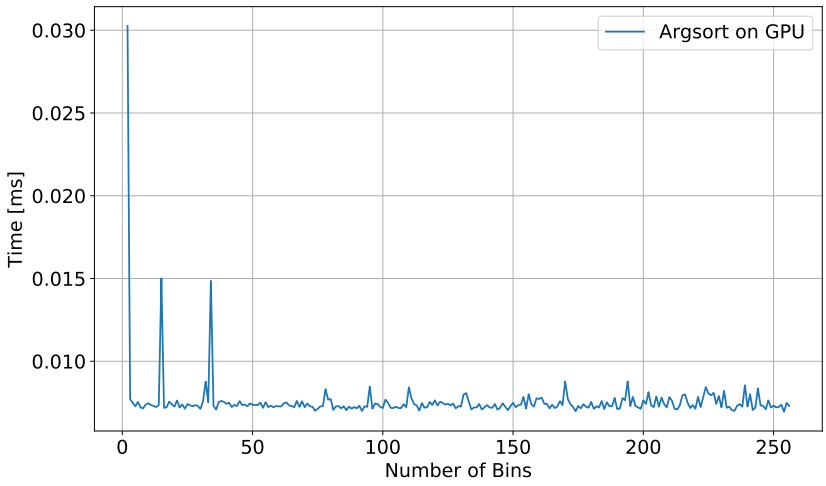


Figure 8: *Execution time of the argument sort against the number of bins. The simulation was run using* 10^6 *particles.*

Figure 8 confirms our expectation that the execution time decreases with the number of bins, even though it is only a small effect. However, we see that there is a big spike for very few bins. We already observed a similar effect in 6b, where one bin corresponds to roughly single threaded performance due to the necessary atomic operations on only one variable.

We also see that the execution time does not increase significantly after roughly 50 bins. That makes sorting with the fine histogram (at 128 bins) the better choice[20].

3.4 Efficient `scatter`/`gather` Operations

To efficiently implement the binned solver routine in OPALX, it is desired to limit the grid interpolation calls inside `scatter` to only those particles residing within a specific energy bin. Since particles in the bunch are not necessarily sorted by their bin index, two main modifications were introduced in IPPL [14]:

Custom Iteration and Hashing in Scatter. The scatter operation is modified to accept a custom iteration range policy as well as an indexing array defined by an `ippl::hash_type`. This allows the `scatter` function to iterate only over particles within a specified bin. The code uses a hashing array (when provided) to map the original index to the correct particle index for scattering. In this context, the map is given by `sortedIndexArr_m` referenced in 3.3.

Additive Gather. In order to obtain the complete field solution on a particle (by accumulating contributions from each individual bin), the `gather` function is modified to add the field value to the particle's attribute instead of overwriting it. This is controlled by a Boolean parameter[21].

The following code excerpt illustrates the modifications made to the interface of the `scatter` method of the `ParticleAttrib` class:

```
template <typename Attrib1, typename Field, typename Attrib2,
          typename policy_type = Kokkos::RangePolicy<typename Field::execution_space>>
inline void scatter(const Attrib1& attrib, Field& f, const Attrib2& pp,
                    policy_type iteration_policy, typename Attrib1::hash_type hash_array =
    {}) {
    attrib.scatter(f, pp, iteration_policy, hash_array);
}
```

It shows how this `scatter` implementation takes a `Kokkos::RangePolicy` as well as a `ippl::hash_type` to define over which indices inside the hash type the `scatter` should iterate. The iteration policy is obtained from the post sum of the histogram, provided by `getIterationPolicy` as outlined in figure 5.

The `gather` operation was also modified to allow either an additive or an overwriting behavior on the particle attribute, depending on a Boolean parameter. The following code excerpt shows this modification:

[20] It is sufficient to sort only the fine bunch. Consider a particle in bin k, where $i \leq k \leq j$. If bins i to j are merged and assigned to bin l, then the former index k will necessarily be mapped to l as well.

[21] This is one way to do it. Another way would be to save a temporary field instance and to call `gather` only once. The second way is the preferred, since modifying a field instance is much quicker than calling the interpolation inside `gather`.

```
template <typename Attrib1, typename Field, typename Attrib2>
inline void gather(Attrib1& attrib, Field& f, const Attrib2& pp,
                   const bool addToAttribute = false) {
    attrib.gather(f, pp, addToAttribute);
}
```

By allowing the gather function to add to existing field values rather than overwrite them, the complete field solution for each particle can theoretically be assembled from the contributions of each individual bin. The final implementation, however, does not use `addToAttribute`. Instead, it uses a temporary field instance and call `gather` only after the complete field is built.

3.5 Adaptive Histogram Merging Algorithm

The overarching goal is to partition the particles into bins and to solve the electrostatic field for each bin individually. A naive approach that divides the velocity range into, say, ten equally sized bins yields a reasonably accurate solution; however, there is potential for improvement, particularly in reducing overall runtime.

On one hand, finer bins generally produce a more accurate self-consistent field by reducing modeling error. On the other hand, wider bins can decrease noise in the solution to the Poisson equation. A second consideration is the wasted computation time in bins that contain significantly fewer particles than others. For instance, as illustrated in the right panel of figure 9, bins in the tails of the distribution may contain very few particles compared to bins near the peak. In such cases, the contribution of a bin with half as many particles is roughly half that of a bin with double the particles (assuming a uniform spatial distribution). It is therefore advantageous to accept a higher modeling error in sparsely populated (wider) bins, which in turn allows for a more accurate treatment of densely populated (narrower) bins. This concept is visually summarized by the difference between the left and right panels of figure 12.

The task can be stated as follows: *Compute a fine histogram from a particle distribution and merge neighboring bins until the optimal non-uniform histogram is obtained.* In order to formalize "optimality", we define a cost function $C[\mathcal{F}]$ that assigns a value to a histogram configuration $\mathcal{F}$.

Let

$$\mathcal{F} = \{(a_1, a_2), \ldots, (a_N, a_{N+1})\}$$

be a histogram configuration, where each element represents a bin with edges satisfying $a_1 \leq a_2$. For simplicity, we assume that all histograms have edges in the interval $[0, 1]$. Define the set of all possible histogram configurations merged from a fine uniform histogram with $N_{\max}$ bins as

$$\mathcal{A} := \left\{ \{(a_1, a_2), \ldots, (a_n, a_{n+1})\} \;\middle|\; n \leq N_{\max},\ a_i \in \left\{ 0, \frac{1}{N_{\max}}, \ldots, 1 \right\} \right\}.$$

Our goal is to find the configuration that minimizes the cost function:

$$\mathcal{F}_{\text{optimal}} = \underset{\mathcal{F}\in\mathcal{A}}{\operatorname{argmin}}\, C[\mathcal{F}].$$

For the merging algorithm to work efficiently, the cost function must be separable into contributions from individual bins. In other words, if we define

$$C[\mathcal{F}] \overset{!}{=} \sum_{i=1}^{N_\mathcal{F}} c[(v_i, v_{i+1})],$$

where $c[(v, w)]$ denotes the cost associated with a bin with edges $v \leq w$, then for a merged histogram

$$\mathcal{G} = \left\{(w_1, w_2), \ldots, \left(w_{N_\mathcal{G}}, w_{N_\mathcal{G}+1}\right)\right\} \quad (\text{with } w_{N_\mathcal{G}+1} = 1),$$

the cost is defined as

$$C[\mathcal{G}] = \sum_{i=1}^{N_\mathcal{G}} c[(w_i, w_{i+1})].$$

It is important to note that in general,

$$c[(w_i, w_{i+1})] \neq \sum_{v_j \in \left\{v_1,\ldots,v_{N_\mathcal{F}}\right\} \,|\, w_i \leq v_j < w_{i+1}} c[(v_j, v_{j+1})],$$

where the sum adds up the cost value of every individual small bin that lies in the range of $[w_i, w_{i+1}]$. This ensures that merging bins can change the overall cost in a nontrivial way.

Without specifying the exact form of the cost function $c[(w_i, w_{i+1})]$, we can outline a recursive algorithm that finds the minimum achievable cost by merging neighboring bins. The naive recursive algorithm is given in snippet 1.

```
def mergeBins(k, prefixCount, prefixWidth, maxBinRatio):
    if k == 0: return 0  # Base case: no bins left, cost is 0
    minCost = LARGE_VALUE

    # Try all possible starting indices i for the last merged bin
    for i from 0 to k-1:
        segCost = computeCost(i, k, prefixCount, prefixWidth, maxBinRatio)
        cost    = segCost + mergeBins(i, prefixCount, prefixWidth, maxBinRatio)
        minCost = min(cost, minCost)
    return minCost
```

Snippet 1: *Naive recursive algorithm that calculates the minimum cost achievable by merging neighboring bins from an initial fine histogram.*

This function calculates the minimum cost achievable by merging bins in the interval $[0, k-1]$, by considering all possible partitions. However, due to its exponential runtime, this recursive method is impractical for large N_{max}.

By employing dynamic programming, we can store the cost for each sub-problem (each configuration) and backtrack to determine the optimal merging strategy. This transformation reduces the runtime complexity to $\mathcal{O}(N_{\text{max}}^2)$, assuming that evaluating $c[\mathcal{F}]$ is $\mathcal{O}(1)$, thereby making the algorithm constant with respect to the overall simulation size.

In summary, the adaptive histogram merging algorithm consists of the following key steps:

Define the cost function. The overall cost is the sum of costs associated with each bin, with the cost for a merged bin defined independently of the sum of the costs of its constituent fine bins.

Naive recursion. Formulate a recursive solution that explores every possible merging configuration.

Dynamic programming. Optimize the recursive algorithm by storing intermediate results to reduce redundant computations, thereby achieving $\mathcal{O}(N_{\text{max}}^2)$ complexity.

The dynamic programming formulation and its implementation details are further discussed in section A.2.4.

3.6 Optimal Cost Function

Section 3.5 explained how the algorithm works that reduces the fine histogram to a potentially non uniform histogram with less bins. In order to run the algorithm, a cost function was introduced that judges how good a merged bin is. Assume that we want to merge bins i to j of a fine configuration. Let

$$N_{i,j} \stackrel{!}{=} \frac{1}{N_{\text{sim}}} \sum_{k=i}^{j} N_k$$

be the number of total particles in the merged bin normalized by the total number of particles N_{sim} in the simulation, where N_k is the number of particles per bin in the fine configuration. Similarly, define the total width[22] of the merged bin as $\Delta v_{i,j} \stackrel{!}{=} v_j - v_i$. Then, we can propose the cost function as follows:

$$c[(i,j)] \stackrel{!}{=} \underbrace{N_{i,j} \log\left(N_{i,j}\right) \Delta v_{i,j}}_{\text{Shannon Entropy}} + \underbrace{\alpha\, \Delta v_{i,j}^2}_{\text{Width Bias}} + \underbrace{\beta \left(\Delta v_{i,j} - \Delta v_{\text{bias}}\right)^2}_{\text{Deviation Penalty}} + \underbrace{\frac{\Delta v_{\text{bias}}}{N_{i,j}} \Theta(\Delta v_{\text{bias}} - N_{i,j})}_{\text{Sparse Penalty}}, \tag{14}$$

where for now $\alpha, \beta \in \mathbb{R}$ are arbitrary parameters, Θ the heavy side function and Δv_{bias} the desired velocity spread per bin that will be explained together with the four terms in the following.

Equation (14) is based on SHANNON entropy, explained in 3.6.1, and has three more penalty terms used to refine the final histogram. $\alpha, \beta \in \mathbb{R}$ are ultimately user provided parameters able to tweak the final result should the user prefer for example lower modeling error in the final field. These additional terms of the sum are explained in 3.6.2, 3.6.3 and 3.6.4.

[22] This value is already normalized, since we assume a value range of 0 to 1 for $c = 1$.

3.6.1 Shannon Entropy

As established before, we need to run a POISSON solver for every bin. Running the solver without enough particles leads to an increased statistical error for the field in that bin. Even though the fields of a bin with few particles will have a overall smaller contribution to the total field, it makes sense to favor histograms where the bin values (number of particles per bin) are balanced in an effort to reduce statistical fluctuations.

It turned out that SHANNON entropy becomes a very useful tool in that regard. CLAUDE ELWOOD SHANNON defined 1948 in [20] his entropy $H(X)$ as

$$H(X) = -\sum_i p(x_i)\ \log p(x_i).$$

X is a discrete random variable with possible outcomes x_i and their respective probabilities $p(x_i)$. Usually, $H(x)$ can be used in information theory to quantify information available in a random variables possible outcomes. A few examples where it turns out to be useful are:

Data compression. Shannon entropy can be used in the field of image analysis. For example, [10] tries to find some order parameter based on images of e. g. crystalline lattices that otherwise would require complex image analysis tools.

Cryptography. Here it can be used to quantify how effective or how "random" a cipher phrase is. The goal would be to maximize $H(X)$, where X is the set of characters (e.g. bits) to choose from for the cipher. However, in real world scenarios, this is not always the optimal approach to maximize security as outlined in [11].

Machine learning. SHANNON entropy can be used in feature selection, regularization, anomaly detection or overfitting by incorporating entropy as a regularization term in the loss function [19].

It has been shown that SHANNON entropy is useful in many fields that have to do with judging how much information is necessary to accurately represent any given data. Especially data compression is conceptually similar to merging a fine histogram into a coarser one. It makes sense to interpret merging as compression of the fine histogram data. Therefore, SHANNON entropy can help answering the question "what is the minimum number of bins necessary to accurately represent the information of the given distribution".

For a continuous particle density distribution $f(v)$, where $v \in [0, 1] =: V$ is in the set of the normalized velocity, we can write

$$H(V) = -\int \mathrm{d}v\, f(v)\ \log\left(f(v)\right) \approx -\sum_{i=1}^{N_\mathrm{b}} \Delta v_i\, f(v_i)\ \log\left(f(v_i)\right),$$

where the sum discretizes the integral using the rectangle rule with N_b rectangles. In the language of bins, Δv_i is the width of the i-th bin and v_i the left edge of the bin.

For f given, $f(v_i)$ can be estimated assuming N_{sim} is the total number of particles and N_i the number of particles counted in bin i. Then,

$$f(v_i) = \frac{N_i}{N_{\text{sim}}}, \qquad H(V) = -\sum_{i=1}^{N_\text{b}} \Delta v_i \, \frac{N_i}{N_{\text{sim}}} \, \log\left(\frac{N_i}{N_{\text{sim}}}\right)$$

is the SHANNON entropy for a given histogram configuration. Notice that this is exactly the same as the first term in the cost function (14). In this context, a high $H(X)$ can be interpreted as having a more uniform distribution of particle numbers in the bins. On the other side, a lower entropy would indicate data concentration in specific bins. As stated in the beginning of this section, it makes sense to focus on balancing out particle numbers per bin. The goal should therefore be maximize H, or minimize $-H$, since the algorithm will minimize the cost function. This effectively penalizes configurations in which particle numbers per bin deviate significantly, thereby promoting a balanced histogram and reducing statistical errors in subsequent field solves.

Figure 9 shows an example of using SHANNON entropy in the cost function versus simply a uniformly binned histogram.

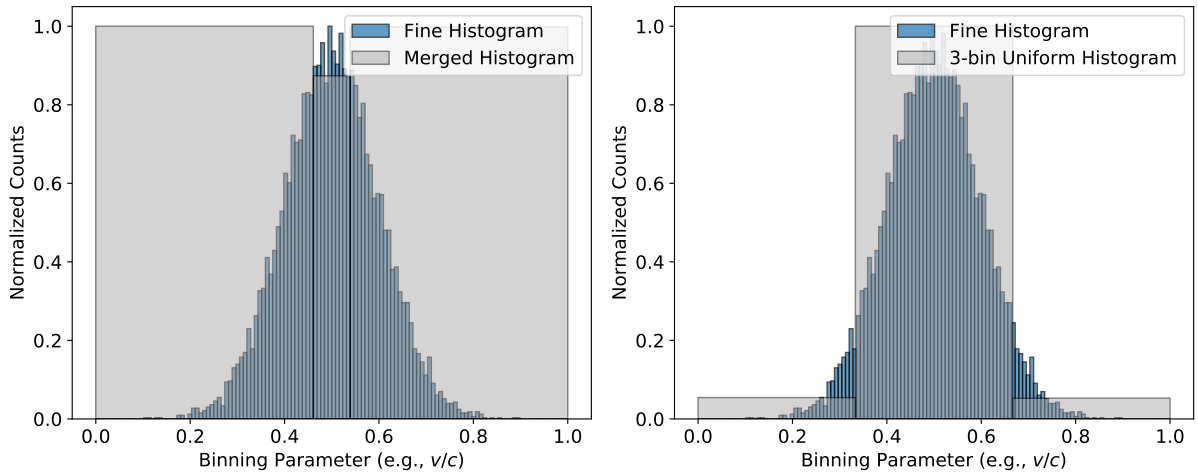


Figure 9: *Two initial* GAUSSIAN *distributions with* $\mu = 0.5$, $\sigma = 0.1$ *and* 10^4 *particles. The left plot shows the optimal merged histogram when using only* SHANNON *entropy in the cost function. The right plot uses uniform binning. The fine histogram in the background is generated using* 128 *uniform bins.*

The right plot in figure 9 shows what happens when the binning is done uniformly[23]. It groups a large number of particles in the middle and leaves only a small number of particles on the sides. The left plot shows a more balanced histogram. All bins have roughly the same number of particles. However, the left plot resolves the peak of the distribution better than the right plot. This behavior is exactly what we want for the binning algorithm: balance

[23] "Uniformly" denotes a binning approach in which the bin width remains constant at $\frac{1}{N_{\text{bins}}}$, and particle assignment is determined by their distribution across the resulting uniform grid.

out the histogram while maintaining as much information about the initial distribution as possible.

3.6.2 Width Bias Term

SHANNON entropy is able to balance the histogram, but cannot control the overall resolution or the actual number of bins in the final histogram. It necessary to add other terms to the cost function that can refine the outcome. The *width penalty term* will be defined as

$$c[(i,j)] \mid_{\text{Width Bias}} \stackrel{!}{=} \alpha \cdot \Delta v_{i,j}^2,$$

where $\alpha \in \mathrm{R}$ and $v_{i,j} \stackrel{!}{=} v_j - v_i$ are defined in 3.6.

Why does this term make sense? The goal of this term is to penalize bins that are either too narrow or too wide. The merging algorithm minimizes the cost function. To favor narrower bins, the costs should decrease for smaller $\Delta v_{i,j}$. Therefore, it is advisable to select a strictly monotonically increasing function. Since $\Delta v_{i,j} > 0$ always applies[24], any monomial is a valid choice. Depending on whether $\alpha > 0$ or $\alpha < 0$ is set, smaller bins or larger bins are preferred. However, simply $c \propto \Delta v_{i,j}$ has not shown any significant effect on the final histogram. On the other hand, $\propto \Delta v_{i,j}^2$ initially has a smaller slope and for large $\Delta v_{i,j}$ a larger slope than $\propto \Delta v_{i,j}$. As a result, the effect becomes more significant the larger the current bin is, which is exactly what makes this term useful.

The effect[25] that this term has can be seen in figure 10.

[24] When $c[(i,j)]$ is calculated, we define $j > i$ and therefore get $v_j > v_i \Rightarrow \Delta v_{i,j} > 0$. It is always strictly greater since the merging algorithm is only run if the total velocity spread is not 0.

[25] Note that it does not make sense to compare *only* the width bias term, since $\alpha < 0$ would simply lead to one bin, $\alpha > 0$ to the maximum number of bins, here 128. The effect only makes sense once it is used together with SHANNON entropy.

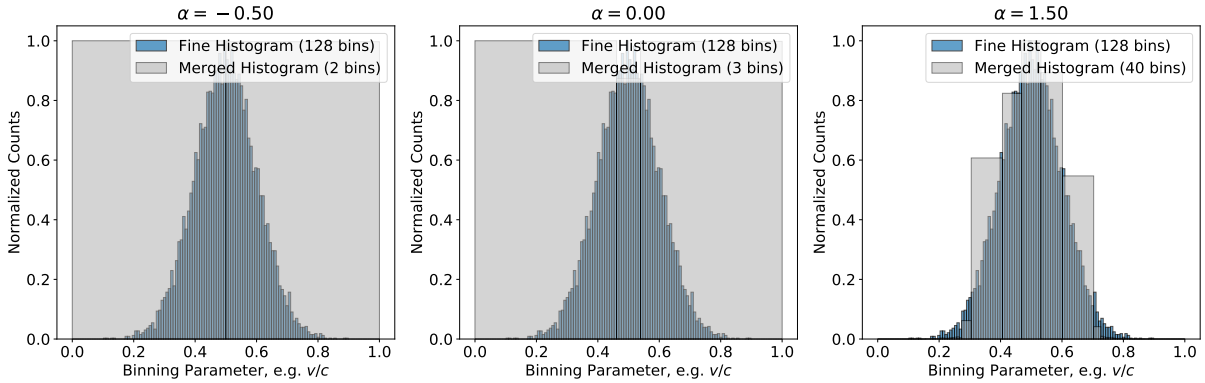


Figure 10: *Influence of the width bias term in (14) for different values of α. The initial distribution is a normal distribution with $\mu = 0.5$, $\sigma = 0.1$ and 10^4 values. The fine histogram is generated uniformly with 128 bins. The merged histogram is the result of using* Shannon *entropy together with the width bias term.*

$\alpha = 0$ recovers the left histogram in 9. A negative value results in a histogram with less number of bins since the cost decreases for bigger $\Delta v_{i,j}$. The opposite happens for $\alpha > 0$ as can be seen in the third plot of figure 10. All three plots show exactly the desired behavior and give the user the possibility to tweak the final modeling error by favoring larger or smaller bins leading to less or more bins in the final histogram.

Even though the desired behavior is shown, we see in the third plot of figure 10 that the tails of the distribution are barely merged. The effect of these unmerged bins are unnecessary field solver calls where the actual contribution to the final solution is minimal, which is exactly why we want to bin in the first place as described in 2.5. The effect of unmerged bins appears only for big enough values of α, which is $\alpha \gtrsim 0.8$ for the distribution used in figure 10. If α is too big, we quickly observe

$$c[(i,j)]\ |_{\text{Shannon Entropy}} \ll c[(i,j)]\ |_{\text{Width Bias}}\ .$$

In this case, the width bias term has by far the greater influence on the overall cost leading to bins being as small as possible in regions where Shannon entropy has overall small values, which is mostly seen at the tails of distributions where we would generally expect bins to be merged. This issue[26] will be solved in section 3.6.4.

3.6.3 Deviation Penalty Term

The terms obtained from 3.6.1 and 3.6.2 are not enough, since they have no way of steering the merged histogram towards a desired bin width. They are only able to favor smaller/bigger bins and balancing out the bin counts. However, as a user, it might make sense to tell

[26]Note that Shannon entropy is not enough as stated 3.6.2, since it undershoots the number of bins in all example distributions used for this analysis. So it makes sense that we want $\alpha > 0$, which usually ends up in exactly this effect of unmerged bins at the tails. If we want to be able to tweak the final number of desired bins, we therefore have to add another penalty term.

the algorithm to additionally favor a specific bin width. Therefore, the following term is a good choice:

$$c[(i,j)]\ |_{\text{Deviation Penalty}} \overset{!}{=} \beta \cdot (\Delta v_{i,j} - \Delta v_{\text{bias}})^2,$$

where the variables are defined in 3.6. Since we want the cost to be minimized, we need[27] $\beta \geq 0$. Then the quadratic function is minimized for $\Delta v_{i,j} = \Delta v_{\text{bias}}$. Therefore, this part will always favor bins whose width is closer to Δv_{bias}. The effect that $c[(i,j)]\ |_{\text{Deviation Penalty}}$ has together with SHANNON entropy can be seen in figure 11.

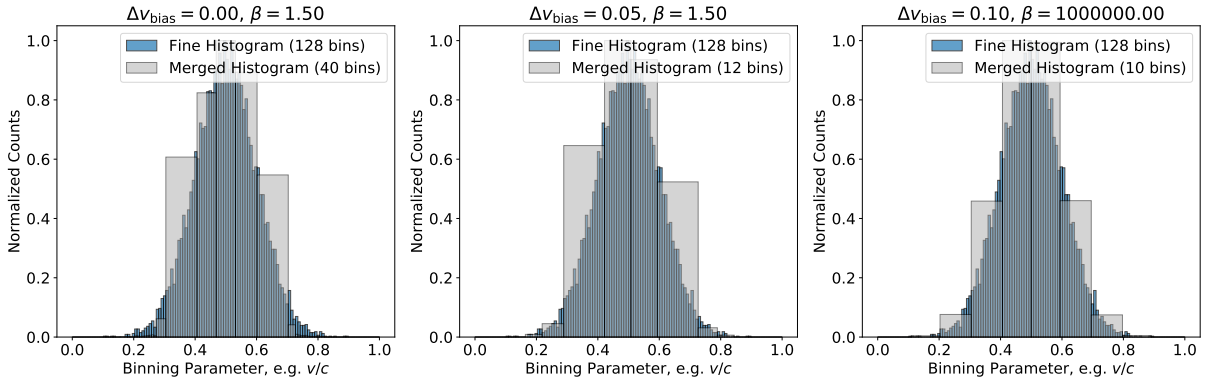


Figure 11: *Influence of the deviation penalty term in (14) for different values of Δv_{bias} and β. Initial distribution and fine histogram are the same as in 10. The merged histogram is the result of using* SHANNON *entropy together with only the deviation penalty term.*

The first two plots have the same $\beta = 1.5$ value and differ in Δv_{bias}. For $\Delta v_{\text{bias}} = 0$ we see that it recovers the right plot of 10, since in that case

$$c[(i,j)]\ |_{\text{Deviation Penalty}} \overset{!}{=} c[(i,j)]\ |_{\text{Width Bias}}$$

for $\alpha = \beta$. In the second plot of 11 we see that the most significant[28] bins in the middle got bigger to approximately match $\Delta v_{\text{bias}} = 0.05$.

The third plot of 11 shows the possibility of forcing uniform binning for an expected number of bins N by setting $\Delta v_{\text{bias}} \overset{!}{=} N^{-1}$ and $\beta \gg 1$. The actual value of β is not important, it just needs to be big enough to overshadow $c[(i,j)]\ |_{\text{Shannon Entropy}}$. Finally, we still see the same effect mentioned in 3.6.2 where the tail experience almost no merging.

3.6.4 Sparse Penalty Term

Finally, we need to address the problem of having unmerged bins at the tails of distributions as seen for example in figure 11. The reason why these appear is because of the quadratic width penalty terms and explained in more detail in 3.6.2. The solution is to add another term to the cost, where the penalty grows inverse proportional to the number of elements in

[27] Setting $\beta = 0$ would deactivate this part of the cost function.

[28] Most significant in the sense that bins with higher particle counts will end up having a bigger impact on the final electric field.

a bin. We can also restrict this penalty to bins smaller than Δv_{bias}, since they are already big enough per user input. A suitable penalty term is

$$c[(i,j)] \mid_{\text{Sparse Penalty}} \overset{!}{=} \frac{\Delta v_{\text{bias}}}{N_{i,j}} \cdot \Theta(\Delta v_{\text{bias}} - N_{i,j}), \qquad \Theta(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases},$$

where all parameters are defined in 3.6. $N_{i,j}^{-1}$ leads to the desired behavior where the cost increases drastically for bin counts that are very small. The scaling of Δv_{bias} ensures that the cost starts at a reasonably small penalty for bins already close to the desired width and is independent of the choice of Δv_{bias}. The effect of this new term can be seen in figure 12.

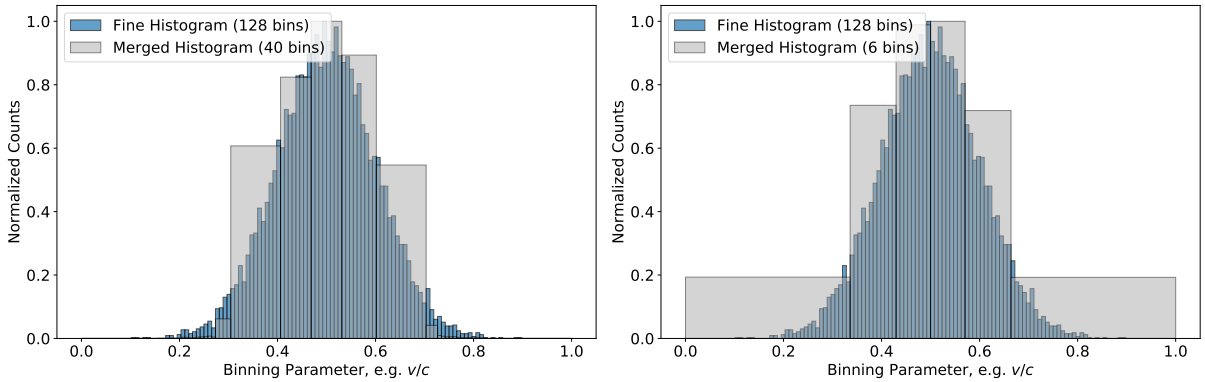


Figure 12: *Influence of the sparse penalty term in (14) for $\alpha = 1.5$, $\beta = 0$ and $\Delta v_{\text{bias}} = 0.05$. Initial distribution and fine histogram are the same as in figure 10. The merged histogram is the result of using the full cost function (14) with the specified parameters. Only the merged histogram in the right subplot uses the sparse penalty term.*

The left plot shows the behavior of unmerged bins. However, the right plot shows how the small bins are penalized, finally replicating the desired optimal behavior as defined in 2.5. Not only does the final cost function replicate what is intuitively optimal, but also allows the user to tweak the actual influence of each part to their liking and allowing more or less bins/modeling error as needed.

3.7 Final Algorithm Implementation

This section aims to summarize how all parts of the algorithm explained in the subsections of 3 work together to become the complete adaptively binned field solver. Figure 13 shows a diagram summarizing every part. The diagram will be explained in more detail in the following.

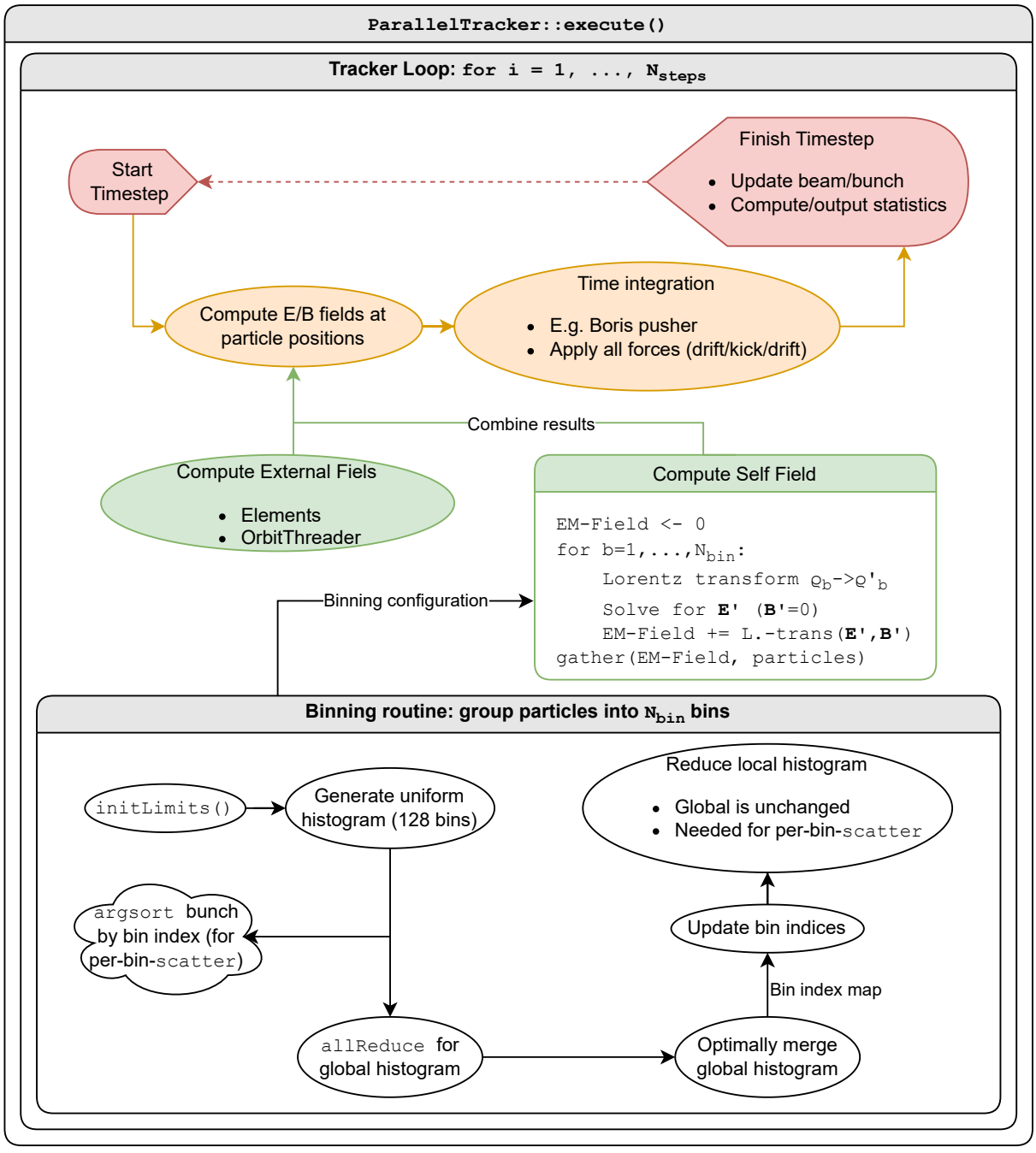


Figure 13: *Top level diagram of OPALX's tracker routine. The graph shows how the* `ParallelTracker` *in OPALX calculates the self-consistent and external fields, integrates the particle bunch and repeats for every timestep until the simulation ends. The major part implemented in this thesis is the bottom container labeled "Binning routine".*

The diagram shows mainly four layers in which each timestep can be grouped.

Red Layer: start/stop. The first layer is not part of the particle simulation itself. OPAL

computes beam statistics at the end of each timestep and writes them to a file/outputs them to the consoles if desired by the user.

Orange Layer: Update attributes. After calculating the fields, they can be combined and gathered for every particle. If the electric and magnetic field vectors at the particle position are known, the drift and kick can be performed in the time integrator (for example Leapfrog or BORIS algorithm). In between, there are some steps left out with respect to LORENTZ- and other transformations, since the fields need to be combined in the same reference frame as outlined in algorithm 1.

Green Layer: Compute fields. Computing the electric and magnetic field has two parts to it, external and self-consistent fields:

- External fields are computed from so called "elements". This can for example be a coil wrapped around the simulation domain exerting an electric field onto the particles. All these contributions need to be combined and finally gathered onto the particles.
- The important factor here is the self-consistent field. Conceptually, we calculate the electric field contribution from every particle in its reference frame, transform it into a common lab frame and combine the field contributions.
  In reality, we solve the POISSON equation for every bin, back transform the fields into the lab frame and combine them to get the total field values. Once the total fields are obtained, we can gather them onto the particles to combine them with the external fields.

Binning Layer: Adaptive binning. The final layer is the adaptive binning routine developed in this thesis. It has a view steps mainly consisting of generating a very fine histogram, distributing it among ranks, optimally merging the global histogram as explained in section 3.5, updating the particles bin indices according to the new merged configuration and finally re-generating the local histogram with the new bin indices. A fine bin count of 128 balances computational efficiency and distribution resolution: it permits using smaller data types while exceeding literature recommendations (section 2.5) to resolve particle distributions accurately.
The last step is necessary, since the local histogram and its prefix sum are used to determine range policies necessary to iterate over particles in each bin as outlined in 3.4.

The explanation of these four levels concludes describing the algorithm, how it is designed and how it is implemented in OPALX.

4 Results

4.1 Convergence Study and Improvements with Adaptive Binning

In section 2.4, we already established that a uniformly binned field solution converges to the full solution as it approximates the integral in equation (8). Therefore, we can calculate a fine (reference-) solution to a problem my choosing a high number of bins, e.g. 30, and comparing the difference of the solution for different bin numbers. This can be seen in figure 14. The difference ("relative error") is calculated from the following equation:

$$\text{Rel. Error} = \frac{\|\phi_{\text{text}} - \phi_{\text{reference}}\|_2}{\|\phi_{\text{reference}}\|_2}. \tag{15}$$

This section uses equation (15) to quantify the convergence experiments.

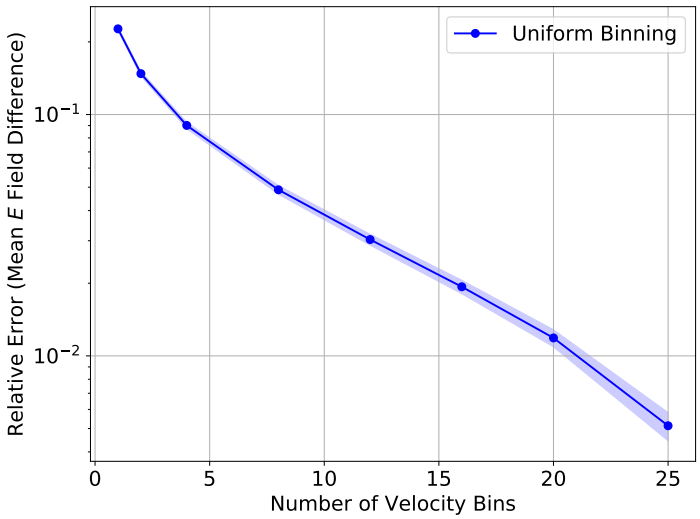
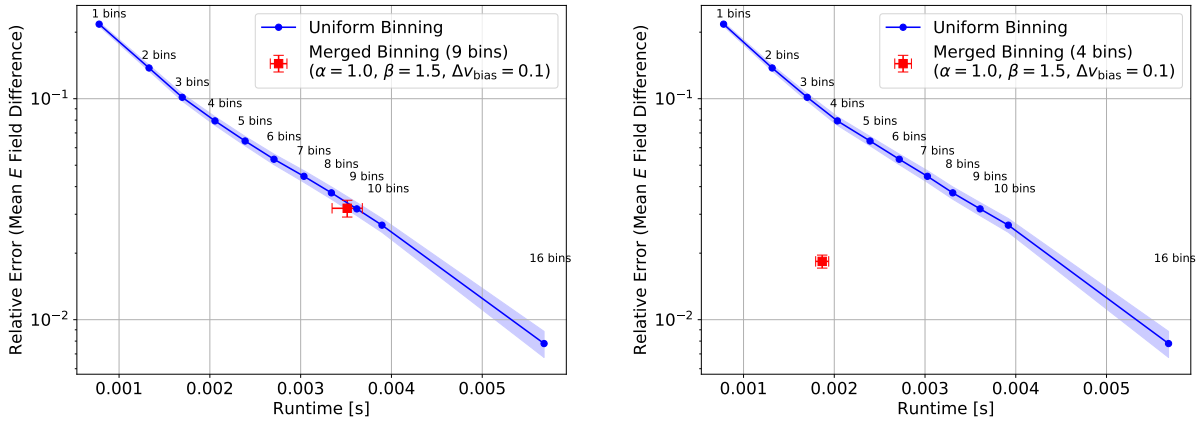


Figure 14: *Simple convergence experiment comparing the binned field solver to a fine solution. The plot shows the relative accuracy from equation (15) against the number of bins used to solve for the field.* 1024×20 *particles are uniformly sampled in position and velocity, each data point was calculated over 200 runs. The solver was configured as in figure 15.*

We can see how the error decreases with the number of bins used as expected. Note that the problem units and parameters are arbitrary. This plot is meant to visualize convergence and later to quantify runtime speedup through the adaptive binning routine.

Next, we can investigate if merging actually has an impact on computation time. Figure 15 shows two plots where the relative field solver accuracy is plotted against computation time. The experiment uses normalized units and only plots the relative difference compared to a fine solution. However, the fine solution is generated again for each run such that it actually compares solves for the same distribution. Both plots show the expected behavior of lower modeling error for higher bin counts. Furthermore, a uniform distribution will



(a) *Uniform velocity distribution in* $[0, c]$.

(b) *Normal distributed velocity with* $\mu = 0.9c$, $\sigma = 0.1c$

Figure 15: *Solver time versus relative accuracy for different initial velocity distributions with* $1024 \times 10 = 10240$ *particles and a simple FFT solver over* 64 *one-dimensional grid points. The blue line indicates a uniform binning, the red dot the field solver with the merged histogram. The data was generated in Python over 200 runs for each data point. The mean result was plotted and the errorbars are given by the standard deviation between runs. The* E *field difference was computed according to equation* (15) *with a reference solution that was calculated using* 20 *bins.*

also lead to uniform binning in the merged algorithm. This is the reason why we see the red dot in 15a at the same height as the line from the uniform binning. However, once we use a shifted normal distribution, which has a long tail without many particles, we see that the merging algorithm is able to improve runtime by approximately 60% for the same accuracy[29].

4.2 Ablation Study on Cost Function Hyper Parameters

The merging algorithm is supposed to be robust with respect to the input distribution, it should never lead to drastic binning behavior impacting accuracy in significant way. Meaning, it should not be possible to tweak the input distribution such that it generates something like the third plot in 11 or the top right plot in figure 16. That said, there are some parameter suggestions that need to be adhered to. A summary of what parameters can be set in OPALX's input file and their influence is given in table 1.

[29]The calculation was done by assuming that the mean of the merged field solver has the same accuracy as the 16 bin configuration in 15a, but only a computation time of four bins.

OPALX Option	Description	Effect when Varied
`MAXBINS`	Number of fine histogram bins used.	Increases the resolution of the initial histogram, but usually not necessary to change.
`BINNINGALPHA`	Binning width bias.	Negative values (e.g., -1.5) favor merging into wider bins, leading to coarser segmentation. 0 deactivates the bias, while positive values (e.g., 1.5) encourage narrower bins.
`BINNINGBETA`	Deviation penalty parameter.	Higher values impose a stronger penalty for deviations from the desired bin width Δv_{bias}, enforcing more uniformity in the merged bins. Lower values make the cost function less sensitive to bin width variations.
`DESIREDWIDTH`	Target width Δv_{bias} for the merged bins.	A smaller desired width targets a finer resolution in the merged histogram, potentially leading to more bins. A larger Δv_{bias} promotes coarser bins.

Table 1: *Overview of user tunable parameters in the histogram merging cost function described in 3.6. The table explains the role of each parameter and describes the expected behavior when they are modified.*

After section 3.6, we can set $\Delta v_{\text{bias}} = 0.1$, $\alpha = 1.0$, $\beta = 1.5$ as a starting point and try different parmeter combinations for the same initial distribution. Figure 16 initialized a GAUSSIAN distribution and tries a few different parameter to merge them.

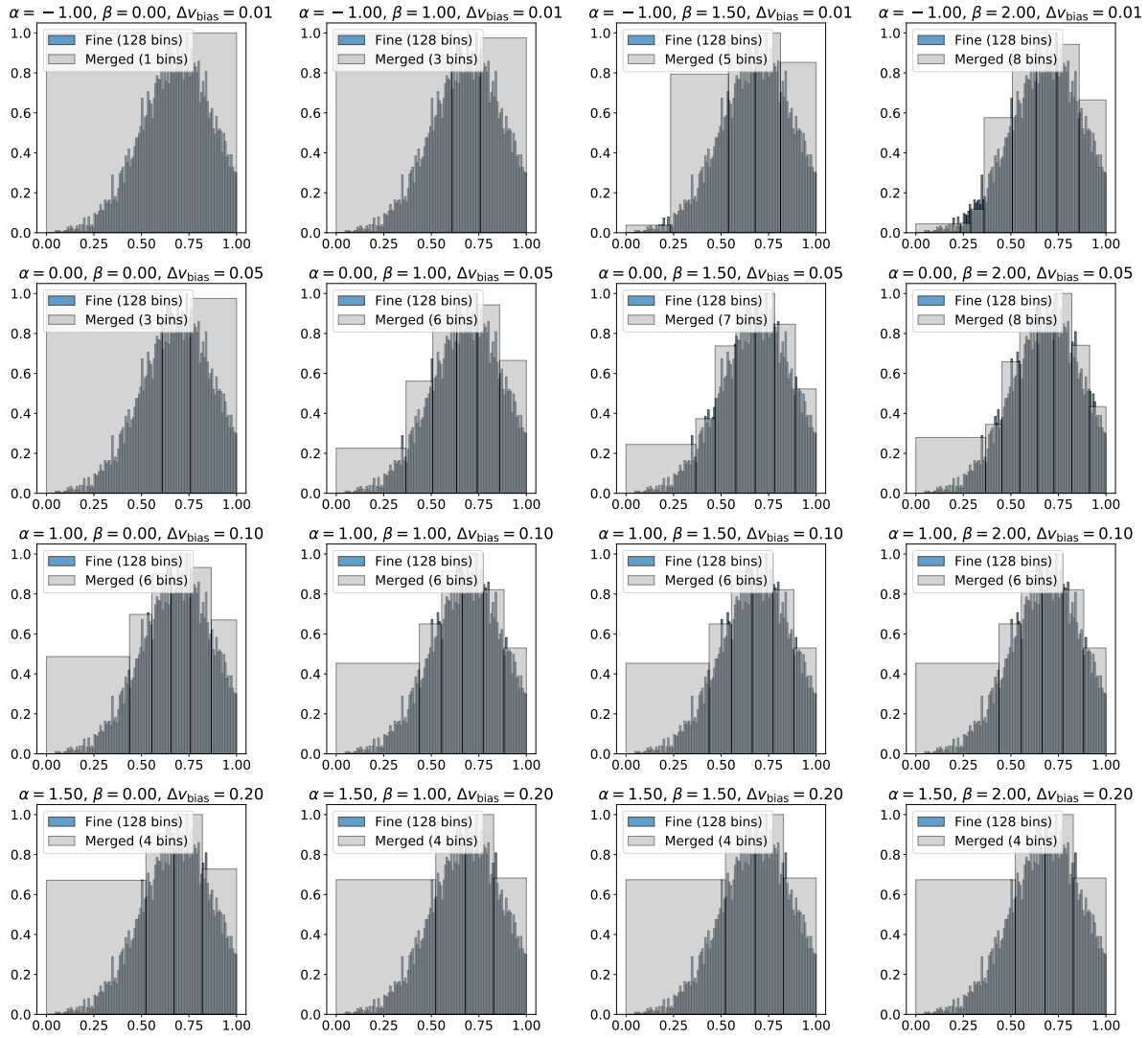


Figure 16: *Binning behavior for different user provided cost function parameter combinations of α, β and Δv_{bias}. Each plot is generated from a* GAUSSIAN *distribution with $\mu = 0.7$, $\sigma = 0.2$, 10^4 particles and 128 fine bins. The parameters used are given in each plot title. Each plot has the binning parameter (for example v/c) on the x axis and the normalized counts on the x axis.*

We can observe that negative α parameters in combination with $\beta = 0$ tend to generate bins that are too big, leading to increased modeling error. We see the same for $\Delta v_{\text{bias}} = 0.2$. With $\beta = 0$ we see that the influence of α is quite big, which could lead to unpredictable behavior in different distributions. Considering the goal of five to ten bins as suggested in section 2.5, it seems like a sweet spot is found with the following ranges:

$$\begin{aligned} 0.75 &\leq \alpha, \beta \leq 1.5 \\ 0.05 &\leq \Delta v_{\text{bias}} \leq 0.15 \\ \text{MAXBINS} &\sim 128 \end{aligned}$$

These parameter ranges serve as guidelines. All tested combinations within these ranges produced adequately merged histograms, characterized by well-resolved peaks and a bin count between 1 and 10. This bin range adapts to the distribution tail extent while preserving key features.

Another important test is to actually see how robust the merging algorithm is with respect to the input distribution, is to sample different distributions that show different simulation stages in the actual AWA application as introduced in section 1. The comparison can be seen in figure 17.

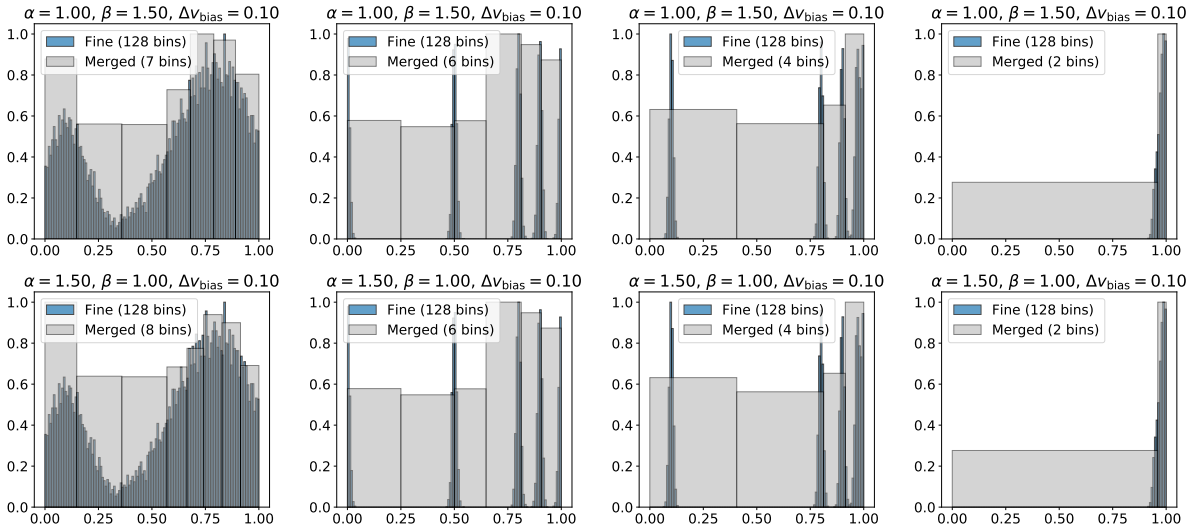


Figure 17: *Binning behavior for four different distributions. Each column has the same fine distribution. Each row has the same binning parameter. Every plot is a concatenation of* Gaussian *distributions with different σ and μ values with $\approx 10^4$ particles and 128 fine bins. The parameters used are given in each plot title. The axis are the same as in figure 16.*

The AWA experiment is represented in these plots in the sense that each simulation timestep might emit a small bunch of particles that will accelerate together as a bunch – similar to an emission after a flattop distribution as outlined in 4.4. The plots more to the right would represent the later stages of the simulation. The right most plots then shows how almost all particles are close to light speed. When all particles are at a similar velocity, we want to see that the merging algorithm reduces the number of bins drastically. For somewhere in between, when there are still bunches that are only slightly accelerated, we expect to separate them into their own bins. All of that can be perfectly seen in figure 17.

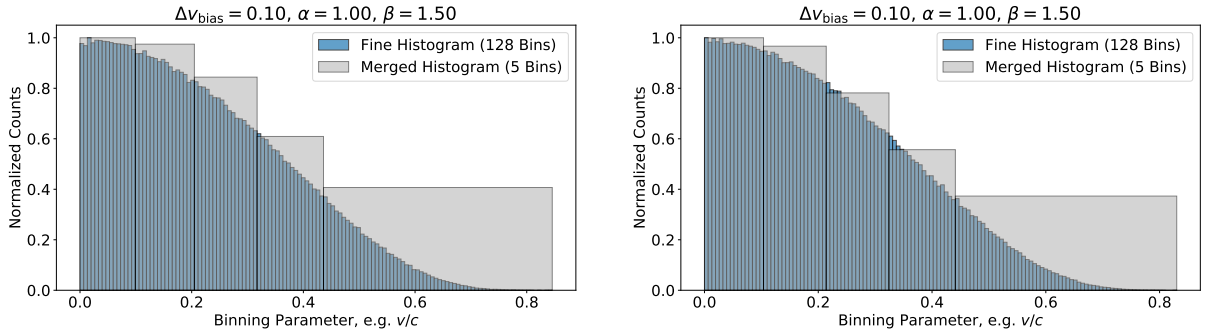
In addition, the first column can be compared with the fourth column in figure 17. The fourth column shows a very large bin, which would definitely lead to significant modeling errors in a uniform distribution. However, looking at the fine distribution, one can see that the main contribution to this bin comes from a smaller interval at the right end of the large bin. This in turn leads to the fact that there is still no large modeling error, since the variance of the particles in this bin remains rather small. However, if this is not the case,

as can be seen very clearly in the first column with the second and third bin, the algorithm does not merge these bins. Both points, again, show exactly the expected behavior.

Finally, it can be said that the cost function proposed in equation (14) and explained by section 3.6 together with the parameters suggested in section 4.2 are justified. Furthermore, it is shown that it makes sense to set default cost function parameters in OPALX to $\Delta v_{\text{bias}} = 0.1$, $\alpha = 1.0$, $\beta = 1.5$ and MAXBINS $= 128$.

4.3 Gaussian Test Case in OPALX

Previous histogram plots were mostly calculated using a Python to show how the algorithm works. However, since it is also implemented in OPALX, we can show that it is able to generate the same results on GPU as well as on CPU applications. Note that the histograms are generated locally on each rank and then communicated to build the global histogram. In the following, we look at the resulting histogram distribution on an example multi rank run on CPU as well as on GPU. The plots can be seen in figure 18. As we see, we get



(a) *Generated on 8 ranks with 8 CPUs each.*

(b) *Generated on 8 Nvidia A100 GPUs.*

Figure 18: *The histograms are generated in OPALX using a* `TYPE=GAUSS` *distribution with* $\vec{\sigma}_{\text{R}} = \{7.5 \cdot 10^{-4}, 7.5 \cdot 10^{-4}, 5.0 \cdot 10^{-4}\}$ *and* $\vec{\sigma}_{\text{P}} = \{0, 0, 0.3\}$ *and* 10^6 *particles. The data was obtained after the first timestep, which did not have a big effect on velocity, since there currently are no external accelerating fields ready in OPALX yet.*

the same results that we would expect after for example figure 16. Since both runs were performed on 8 ranks and the global histograms were plotted after calling `allReduce`, it is also clear that they had to have been initialized correctly on each rank.

4.4 Flattop Test Case in OPALX

The flattop test case is supposed to be close to a real world scenario and is explained in detail in [2]. The idea is to release particles into the simulation each timestep according to a distribution. This simulation is supposed to resemble the AWA gun as explained in 1.

This distribution already exists in OPALX and it is possible to use it to emit particles every timestep. However, they will not be accelerated since OPALX does not implement

the necessary elements at this moment. However, to simulate the same behavior as a constantly accelerated particle bunch, we will add 0.1 to the normalized particle momentum (its units are $[\beta\gamma]$) each time step. That way we can observe how the next particle bunch released into the simulation is accelerated and thereby changing the distribution. Figure 19 shows the histogram at different timesteps.

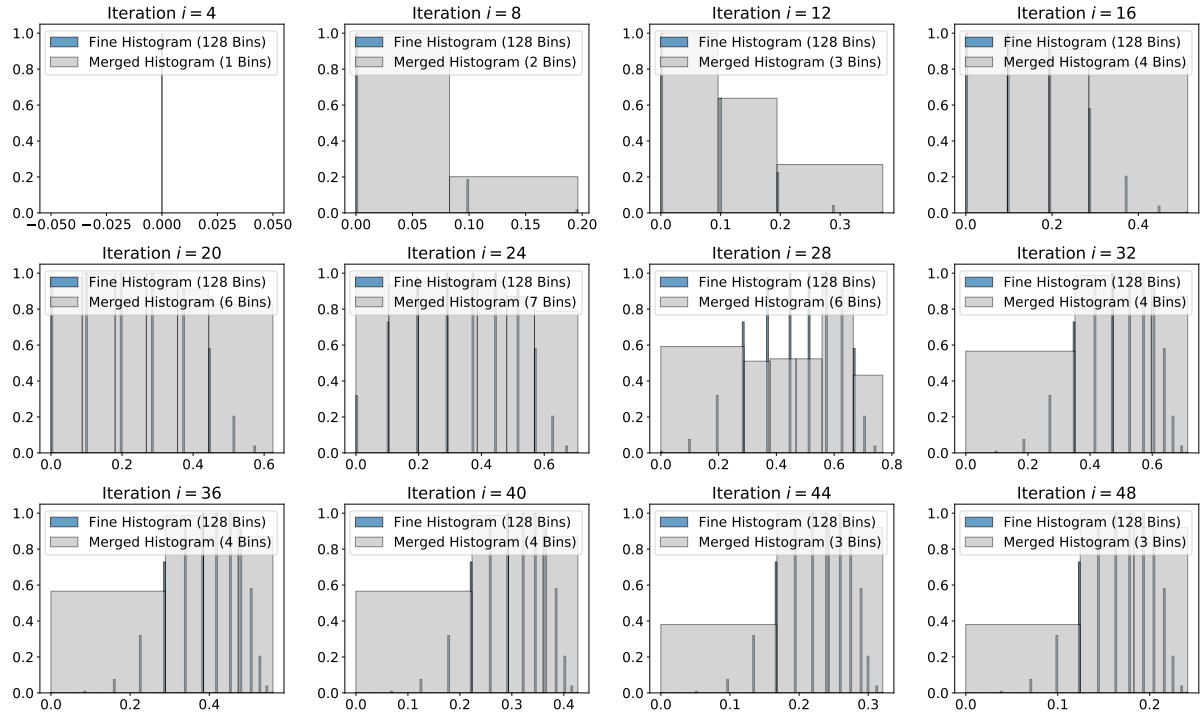


Figure 19: *Time evolution and binning for an example simulation where particles are injected according to a flattop distribution in OPALX. The x axis labels are normalized to begin at 0, even though the particles might already be evolved further. The flattop specific parameters in OPALX were chosen to be* $\sigma_x = \sigma_y = 7.5 \cdot 10^{-4}$, TRISE = TFALL = $6 \cdot 10^{-12}$, TPULSEFWHM = $2 \cdot 10^{-11}$, CUTOFFLONG = 4.0. *The axis are the same as in figure 16.*

One can see that the algorithm is able to capture even these discrete bunches which originate from the injection process. As they accelerate and get closer to light speed, one can see that multiple peaks are merged together, e.g. in the 8th plot. At later stages when the velocity spread decreases, for example the last plot only has an velocity spread of 0.25, the number of bins also decreases. In this example, they decrease from up to 7 at the peak of the emission phase, down to 3 where almost all particles are emitted and have similar velocities.

Even though this clearly shows how the algorithm can handle the flattop distribution[30], it is still just a manually constructed example. Later, once the necessary functionality in OPALX exists to have an accelerating field, it will be interesting to re-run all of these

[30]It was tested on CPU as well as on GPU for single and multi rank runs. The result was always the same.

experiments to check how the distribution actually looks like.

4.5 Scaling Study on the Binning Algorithm

After confirming that the algorithm itself works well by itself in Python and together with OPALX on multiple ranks on GPU as well as on CPU, we can look at the performance and scalability of each part of the algorithm. We will study strong scaling for each CPU and GPU. The performance plots can be seen in figure 20 and 21.

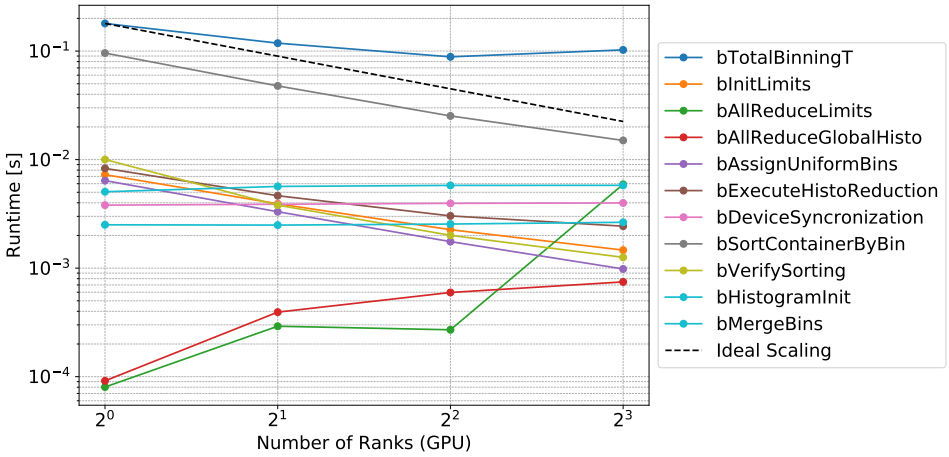


Figure 20: *Strong Scaling Plot on GPU. The simulation used standard cost function parameters, a 128 grid, 8 particles per cell, 20 timesteps and 8 threads per rank and 1 Nvidia A100 GPU per rank.*

The GPU plot confirms that the merge operation is independent of the number of ranks or the problem size. It is basically a constant runtime, only depending on the number of bins in the fine distribution. Bin assignment and coordinate limit initialization are scaling nearly perfectly. This is to be expected, since both consistent of very basic `Kokkos` kernel calls. We also see that the sorting and the histogram reduction scale really well. The histogram reduction deviates a bit from optimal scaling at the 8 ranks mark. This might be due to the fact that it already performs as fast as it gets. The team based approach would not necessarily benefit from a smaller particle number, since it is possible that a not insignificant part of the time comes from team scheduling and scratch memory allocation overhead.
The only parts that do not scale well are associated to communication over ranks. This also expected, since MPI communication time increases with more ranks.

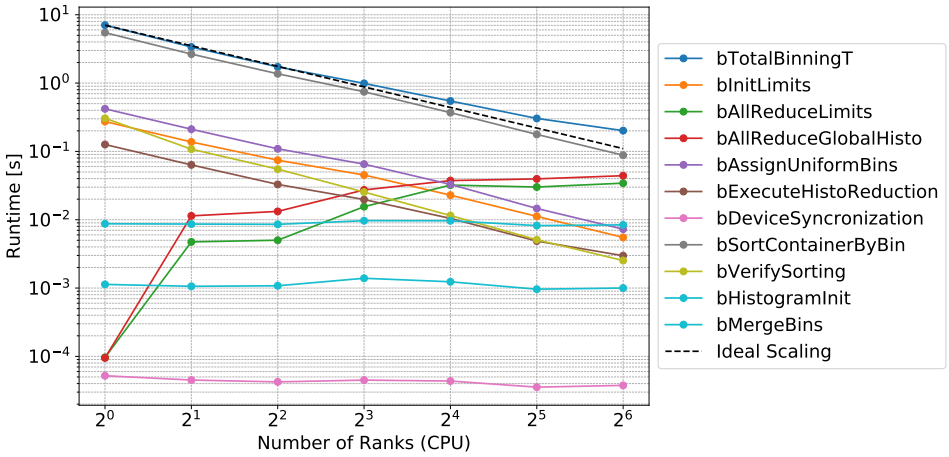


Figure 21: *Strong Scaling Plot on CPU. The simulation prerequisites are the same as in figure 20. 8 threads per rank and 8 ranks per node were used.*

Figure 21 shows a similar behavior as the plot on GPU for the parts that are independent of the problem size. Mainly the merge operation is constant. We also see that sorting, bin assignment, limit initialization and the histogram reduction scale almost perfectly. This is very similar to what we saw before. The only difference is the amount of time that MPI calls take up. These are the only parts that expectedly do not scale with the number of ranks. However, it seems like the overall impact is small compared to the total runtime of the binning routine. Therefore, the total binning time still scales really well. The reason why communication has such a big impact on GPU is simply because the absolute times on GPU are significantly smaller than on CPU.

Finally, a point that should not be overlooked is that the absolute time of the binning routine is one to several orders of magnitude smaller than the `scatter` calls and the field solver themselves. For larger simulations, where binning is all the more important, this effect is also all the greater. That is because the parts of the algorithm that do not scale well depend only on the number of bins (such as merging), i.e. have a constant runtime with respect to the problem size. The other parts, such as assigning the bins, are independent of, for example, the mesh and only depend on simple `Kokkos` kernels. This promises that the algorithm itself will not have a major impact on the actual runtime, but can save a significant amount of time through adaptive binning – such as for example the 60% found in section 4.1.

5 Conclusion

In this thesis, we have developed, implemented, and rigorously tested an adaptive binning algorithm that significantly improves the computational efficiency of field solvers in particle simulations while maintaining high accuracy. By leveraging a cost function with tunable parameters (α, β, and Δv_{bias}), the method dynamically merges neighboring bins in a fine histogram configuration based to the underlying particle distribution. This adaptive approach has been shown to reduce modeling error in a controlled manner, with convergence behavior that scales as N_{bins}^{-1}, as confirmed in [18].

A series of numerical experiments, including convergence studies and runtime-accuracy trade-off analyses, confirmed that the adaptive algorithm not only matches the accuracy of a fine, uniformly binned solution but can also achieve runtime improvements of 60% in cases where the velocity distribution is highly non-uniform[31]. Parameter studies (section 4.2) provided practical guidelines, suggesting default settings of $\Delta v_{\text{bias}} = 0.1$, $\alpha = 1.0$, $\beta = 1.5$, and MAXBINS = 128 to ensure robust performance across a variety of different types of particle velocity distributions.

The algorithm's robustness was further validated through test cases that mimic realistic simulation scenarios. In the Gaussian test case (section 4.3), both CPU and GPU implementations in OPALX yielded consistent and accurate global histograms. Moreover, the flattop test case (section 4.4) demonstrated that the method successfully captures the dynamic evolution of particle bunches, merging bins appropriately as particles accelerate and their velocity spreads decrease.

A comprehensive scaling study (section 4.5) revealed that the computational overhead of the adaptive binning routine is minimal compared to the scatter and field solver operations, particularly in large-scale, multi-rank simulations. Since even one extra bin can impact performance significantly by needing another field solver call, it is basically free in terms of computational cost to run this binning routine at every timestep. The core merging operation exhibits constant runtime behavior, independent of the problem size, while the overall algorithm scales efficiently on both CPU and GPU platforms.

Overall, the adaptive binning algorithm presented in this work not only enhances the performance of particle simulations in OPALX but also lays the groundwork for further advancements. Future work may involve integrating fully functional accelerating fields, where the binning actually becomes the most significant, and refining inter-rank communication strategies to further optimize the algorithm's performance. Here one could for example think about reusing old histograms when only few particles were exchanged to get an approximate histogram. The promising results indicate that adaptive binning is a viable and effective strategy for managing self field accuracy against computational complexity in OPALX.

[31] Non uniformity can for example be a GAUSSIAN distribution. Suitable for this example is a distribution with small variance and long tails.

References

[1] Andreas Adelmann et al. *OPAL (Object Oriented Parallel Accelerator Library)*. https://gitlab.psi.ch/OPAL/src. Accessed: 2025-03-09.

[2] Andreas Adelmann et al. *OPAL Manual: FlatTop*. Accessed: 2025-03-23. Paul Scherrer Institute. 2024. URL: https://amas.web.psi.ch/opal/Documentation/2024.1/#sec.distribution.gaussdisttypephotoinjector/.

[3] Andreas Adelmann et al. *The OPAL Framework: Version 2024.1*. Accessed: 2025-03-09. Paul Scherrer Institute. 2024. URL: https://amas.web.psi.ch/opal/Documentation/2024.1/.

[4] NVIDIA Developer Blog. *Inside Pascal*. https://developer.nvidia.com/blog/inside-pascal/. Accessed: March 23, 2025.

[5] OPAL Collaboration. *OPAL-X*. Accessed: March 24, 2025. ongoing. URL: https://gitlab.psi.ch/OPAL/opal-x/src.

[6] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. 2nd ed. Hoboken, NJ, USA: Wiley-Interscience, 2006. ISBN: 978-0-471-24195-9. URL: https://onlinelibrary.wiley.com/doi/book/10.1002/047174882X.

[7] Lukas Einkemmer. "Splitting methods for Vlasov–Poisson and Vlasov–Maxwell equations". PhD thesis. Innsbruck, Austria: University of Innsbruck, 2014. URL: https://www.uibk.ac.at/mathematik/personal/einkemmer/phd-thesis.pdf.

[8] Fermi National Accelerator Laboratory. *Leading Accelerator Technology*. https://www.fnal.gov/pub/science/particle-accelerators/accelerator-technology.html. Accessed: 2025-03-09.

[9] G. Fubiani et al. "Space charge modeling of dense electron beams with large energy spreads". In: *Phys. Rev. ST Accel. Beams* 9 (6 June 2006), p. 064402. DOI: 10.1103/PhysRevSTAB.9.064402. URL: https://link.aps.org/doi/10.1103/PhysRevSTAB.9.064402.

[10] Marta Grasa Lainez. "Shannon entropy, order and image compression". In: *Universitat de Barcelona* (2023). URL: https://diposit.ub.edu/dspace/bitstream/2445/189771/1/GRASA%20LAINEZ%20MARTA_5181461_assignsubmission_file_TFG-Grasa-Lainez-Marta.pdf.

[11] Martin E. Hellman. "An Extension of the Shannon Theory Approach to Cryptography". In: *IEEE Transactions on Information Theory* IT-23.3 (May 1977), pp. 289–294. URL: https://www-ee.stanford.edu/~hellman/publications/25.pdf.

[12] JGraph Ltd. *draw.io*. Version 22.1.16. Accessed: March 24, 2025. 2025. URL: https://app.diagrams.net.

[13] Sandia National Laboratories. *The Kokkos Lectures: Module 4 – Hierarchical Parallelism*. https://indico.math.cnrs.fr/event/12037/attachments/5040/8137/KokkosTutorial_04_HierarchicalParallelism.pdf. Accessed: 2025-06-16. 2024.

[14] Alexander Liemen. *Add Custom and Partial Iteration for Particle Scatter and Gather Operations*. https://github.com/IPPL-framework/ippl/pull/327. Accessed: 2025-03-23. 2025.

[15] Alexander Liemen. *HS24 Master Thesis*. GitHub repository. The is the main repository to my masters thesis in computational physics. 2024–2025. URL: https://github.com/aliemen/HS24-masters-thesis.

[16] Sriramkrishnan Muralikrishnan et al. "Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures". In: *Proceedings of the 2024 SIAM Conference on Parallel Processing for Scientific Computing (PP)*. SIAM. 2024, pp. 26–38. URL: https://github.com/IPPL-framework/ippl.

[17] N.R. Neveu et al. "Photoinjector Optimization Studies at the AWA". In: *Proc. 9th International Particle Accelerator Conference (IPAC'18)*. THPMF049. 2018, pp. 4169–4171. DOI: 10.18429/JACoW-IPAC2018-THPMF049. URL: http://jacow.org/ipac2018/papers/thpmf049.pdf.

[18] S.A. Schmid, H. De Gersem, and E. Gjonaj. "Energy-Binning Fast Multipole Method for Electron Injector Simulations". In: *Proc. IPAC'21* (Campinas, SP, Brazil). International Particle Accelerator Conference 12. https://doi.org/10.18429/JACoW-IPAC2021-THPAB229. JACoW Publishing, Geneva, Switzerland, Aug. 2021, TH-PAB229, pp. 4244–4246. ISBN: 978-3-95450-214-1. DOI: 10.18429/JACoW-IPAC2021-THPAB229. URL: https://jacow.org/ipac2021/papers/thpab229.pdf.

[19] Salomé A. Sepúlveda Fontaine and José M. Amigó. "Applications of Entropy in Data Analysis and Machine Learning: A Review". In: *The Moonlight* (Mar. 2025). URL: https://arxiv.org/pdf/2503.02921.

[20] Claude E. Shannon. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27.3 (July 1948), pp. 379–423, 623–656. DOI: 10.1002/j.1538-7305.1948.tb01338.x. URL: https://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf.

A Appendix

A.1 Reproducibility and General Compile Instructions

All files, code and resources necessary to reproduce the results are accessible via the main GitHub repository [15]. It contains the presentations for this thesis (beginning, midterm and final defense), the Python code and data used for the plots, the draw.io files used to generate the diagrams and the code repository for the `AdaptBins` and `Histogram` class with some IPPL test cases.

Note that [15] splits the C++ code from the rest of the thesis files in its own repository and that the main repository only links to it as a submodule.

The following subsections contain two instruction sets. For both examples, one has to install the necessary packages as recommended by the official IPPL [16] and OPALX [5] repositories. On the merlin and gwendolin clusters[32] the following module imports were used:

```
load_opal_gpu() {
    module clear
    module use unstable;

    module load gcc/12.3.0
    module load  gtest/1.13.0-1
    module load openmpi/4.1.5_slurm

    module use Libraries
    # module load ucx/2.14.1_slurm
    module load fftw/3.3.10_merlin6
    module load boost
    module load gsl
    module load hdf5
    module load H5hut
    module load cuda/12.1.1
    module load cmake/3.25.2
}
```

Compilation was also done `cmake` and also the official `cmake` parameter suggestions by IPPL and OPALX.

A.1.1 Reproduce Plots and Diagrams

This part is quite simple. All plots are generated using Python. [15] has a folder `notebooks-and-analysis/thesis` that has a bunch of Jupyter notebooks that contain the code used to generate all plots and graphs shown in this thesis. The code can be run without external data files. Timing data or simulation data is either obtained by python itself or linked with relative paths in the folder structure where the notebooks are located. No non standard Python libraries were used (only common ones like Numpy, Matplotlib or Pandas).

[32] Both are development clusters at PSI which were available to me during the development process.

The subdirectory `thesis-report/drawio-graphs` contains all diagram files that were generated using `draw.io` [12], a free and open source program that allows for example flow chart and diagram creation. All such diagrams used in this thesis are saved in the subdirectory.

The newest version of this thesis is saved under `thesis-report/latex`.

A.1.2 Binning Code Integration

The main code repository used for development of the adaptive binning routine is saved as a submodule under `binning-code/` in [15]. It contains all newly added files, such as for the `AdaptBins` and `Histogram` class, and also a test case under `test/alpine/`. This section will show what needs to be added in order to run the binning in an IPPL based program, with special focus on OPALX.

Add binning files. First, one has to add the binning files to the project. In OPALX this is done by putting all code files from `binning-code/` into `src/PartBunch/Binning/`. Then add `add_subdirectory (Binning)` to the `CMakeLists.txt` inside the `PartBunch` folder. Inside the `Binning` folder put the following into the `CMakeLists.txt` in order to correctly link the binning files:

```
set (_SRCS
    AdaptBins.tpp
    BinHisto.tpp
)
include_directories (
    ${CMAKE_CURRENT_SOURCE_DIR}
)
add_opal_sources (${_SRCS})
set (HDRS
    AdaptBins.h
    BinHisto.h
    BinningTools.h
    ParallelReduceTools.h
)

install (FILES ${HDRS} DESTINATION "${CMAKE_INSTALL_PREFIX}/include/PartBunch/
    Binning")
message (STATUS "Added Binning files.")
```

After that it should be linked correctly to be used in the actual program.

Save `AdaptBins` instance in `PartBunch`. The instance of the `AdaptBins` class is only initialized once and then saved in the particle bunch, which is `src/PartBunch/PartBunch.hpp` for OPALX. There one can define the following type declarations and `AdaptBins` instance:

```
using BinningSelector_t = typename ParticleBinning::CoordinateSelector<
    ParticleContainer_t>;
using AdaptBins_t = typename ParticleBinning::AdaptBins<ParticleContainer_t,
    BinningSelector_t>;
using binIndex_t = typename ParticleContainer_t::bin_index_type;

std::shared_ptr<AdaptBins_t> bins_m;
std::shared_ptr<VField_t<T, Dim>> Etmp_m;
```

The temporary field instance is technically no necessary for the binning, but will be required once the program actually loops over the bins to solve the field for each bin.

Initialize `AdaptBins`. After that, one has to initialize the bin class instance, which is likely done in the `PartBunch` constructor:

```
using bin_index_type = typename ParticleContainer_t::bin_index_type;
bin_index_type maxBins = Options::maxBins;
this->setBins(std::make_shared<AdaptBins_t>(
    this->getParticleContainer(),
    BinningSelector_t(2), // z axis of particle velocity
    static_cast<bin_index_type>(maxBins),
    // Cost function parameters
    Options::binningAlpha, Options::binningBeta, Options::desiredWidth
));
this->getBins()->debug(); // optional
```

Note that the `Options` are from the OPALX input file and can be set to whatever (e.g. double) are necessary.

`ParticleContainer` definitions. The particle container needs to have the following fields (inherits from `ippl::ParticleBase`):

```
using bin_index_type = short int;  // type in which bin attribute is saved

/// the energy bin the particle is in
ippl::ParticleAttrib<bin_index_type> Bin;
void registerAttributes() {
    ...
    this->addAttribute(Bin); // register bin attribute
    ...
}
```

Here one needs to make sure that the `getBinView()` function inside the `AdaptBins` class accesses the correct attribute of the particle bunch. The standard attribute name should be "`bunch_m->Bin`".

This description shows all changes made to OPALX[33] necessary to use the `AdaptBins` class.

Initialize Bins. Next, we can look at an example on how one would use the binning class. This will be done in the context of OPALX, but can be applied analogously on IPPL. Inside `PartBunch::computeSelfFields()` one first has to initiate the binning:

```
std::shared_ptr<AdaptBins_t> bins = this->getBins(); // get AdaptBins instance

bins->doFullRebin(bins->getMaxBinCount()); // generate fine histogram
bins->print();                             // optional, outputs histogram
bins->sortContainerByBin();                // sort with fine histogram
bins->genAdaptiveHistogram();              // call mergeBins on fine histogram
bins->print();                             // optional, output merged histogram
```

That completes initializing all the bins based on the current particle data.

[33] These are equivalent to changes that would be made in IPPL if one decides to use binning inside for example `alpine`. This, however is not necessary to explain, since [15] under `binning-code/test/alpine` already provides a working version of `LandauDamping` together with binning, see A.1.3.

Solve field per bin. What follows is the actual binned field solver. The routine will look something like this:

```
VField_t<double, 3>& Etmp = *(this->getTempEField());
Etmp = 0; // reset temporary field

// iterate over number of bins in new merged configuration
for (binIndex_t i = 0; i < bins->getCurrentBinCount(); ++i) {
    this->scatterCICPerBin(i); // scatter particles in bin i onto charge density field
                               // saved in the solver instance
    this->fsolver_m->runSolver(); // run solver on particle in bin i
    Etmp += bins->LTrans(this->fcontainer_m->getE(), i); // conceptual: transform solved E
                                                         // field into lab frame and add
                                                         // contribution of field of bin i
}
gather(this->pcontainer_m->E, Etmp, this->pcontainer_m->R); // gather field onto particles
```

The `scatterCICPerBin` function takes the current bin index and does something like the following:

```
scatter(*q, *rho, *R,
        this->bins_m->getBinIterationPolicy(binIndex),
        this->bins_m->getHashArray()
);
```

`getHashArray()` will return the sorted index array obtained by calling `sortContainerBy Bin()` and `getBinIterationPolicy(...)` will return a `Kokkos::RangePolicy` that iterates over the subsection of the hash array corresponding to the indices in the particle bunch corresponding to the particles in bin i.

After that follows the normal stepper routine concluding the binned field solver. What changes is only how the fields are calculated. Notice that except for what was talked about in 3.4, no changed were made to core IPPL functionality and the code very much remains separated from the rest of OPALX or IPPL.

A.1.3 Run `LandauDamping` with Binning

As discussed in A.1.2, there are no changes made in IPPL's core functionality. It is enough to paste the complete code folder in `binning-code/` [15] as inside for example `test/binning/` in IPPL. Then one can simply add `add_subdirectory (Binning)` in the `CMakeLists.txt` inside the `test` folder and `BinningLandauDamping` test case will get compiled, one just needs to make sure that the binning parameters are set manually in the code. It can be run using the normal attributes of `LandauDamping`, which are for example explained in the beginning of `BinningLandauDamping.cpp`. If the binning repository is not put into `test/binning/`, make sure to change the `CMakeLists.txt` under `binning/test/alpine/` to reflect that.

This application was developed exclusively for testing purposes and is not included in the final OPALX integration. It serves as a minimal prototype to evaluate binning in IPPL alongside a moving particle bunch as well as a preliminary binned field solver. The test demonstrated that the binning functionality operates robustly with various initial

distributions and confirmed the practicability and promising performance of a binned field solver before putting it into OPALX.

A.2 Detailed Algorithm Descriptions

This section is meant to explain important algorithm referenced in the report in more detail.

A.2.1 `DualView` Histogram Class

This section summarizes the design, functionality, and usage of the `Histogram` class. This class, defined within the `ParticleBinning` namespace, is templated to support arbitrary data types and `Kokkos` execution environments. Underlying data is stored either in `Kokkos::View` or `Kokkos::DualView` instances. In particular, it provides:

Data Management. Storage for particle counts, bin widths, and a cumulative (post) sum. All three are needed for efficient particle sorting 3.3 and the merging algorithm 3.5.

Initialization Routines. Methods to set up uniform bin widths, compute prefix sums, and synchronize host/device views. The user can set a template attribute to choose between saving data just on host or saving it on host and another memory space. This allows flexibility when working with data that needs to be available on device and on host, like for example the local histogram[34].

Adaptive Rebinning. The class contains the implementation of the binning algorithm explained in 2.4. When called, it generates and returns an index map that transforms the class local histogram instance to the merged configuration and updates it. This can only be done on CPU.

The `Histogram` class is declared as follows:

```
template <typename size_type, typename bin_index_type, typename value_type,
          bool UseDualView = false, class... Properties>
class Histogram {
public:
    // Type definitions for device and host views
    using view_type  = std::conditional_t<UseDualView,
                                          Kokkos::DualView<size_type*, Properties...>,
                                          Kokkos::View<size_type*, Properties...>>;
    using dview_type = typename DeviceViewTraits<UseDualView, view_type>::d_type;
    using hview_type = typename DeviceViewTraits<UseDualView, view_type>::h_type;

    // Similar type definitions for bin widths and index transformations
    // ...

    // Constructors
    Histogram() = default;
    Histogram(std::string debug_name, bin_index_type numBins, value_type totalBinWidth,
```

[34] It needs to be on device to efficiently generate the histogram and on host to call `allReduce` for the global histogram.

```
                value_type binningAlpha, value_type binningBeta, value_type desiredWidth);

    // Copy constructor and assignment operator
    Histogram(const Histogram& other);
    Histogram& operator=(const Histogram& other);

    // Initialization routines and synchronization methods
    void init();
    void sync();
    void initConstBinWidths(const value_type constBinWidth);
    void initPostSum();

    // Access functions for per-bin-scatter
    size_type getNPartInBin(bin_index_type binIndex);
    Kokkos::RangePolicy<> getBinIterationPolicy(const bin_index_type& binIndex1,
                                                const bin_index_type numBins = 1);

    // Adaptive bin merging functions
    value_type partialMergedCDFIntegralCost(
            const size_type& sumCount,
            const value_type& sumWidth,
            const size_type& totalNumParticles
    );
    hindex_transform_type mergeBins();

private:
    // Member variables for debugging and data storage
    std::string debug_name_m;
    bin_index_type numBins_m;
    value_type totalBinWidth_m;

    // Cost function Parameters
    value_type binningAlpha_m;
    value_type binningBeta_m;
    value_type desiredWidth_m;

    // Sorted histogram (either Kokkos::View or ::DualView)
    view_type       histogram_m;
    width_view_type binWidths_m;
    view_type       postSum_m;

    // Helper function to perform shallow copy of members
    void copyFields(const Histogram& other);
};
```

The `Histogram` class is constructed with a debugging name, the number of bins, the total bin width, and additional cost function parameters. For example, the constructor performs the following actions:

```
Histogram(std::string debug_name, bin_index_type numBins, value_type totalBinWidth,
          value_type binningAlpha, value_type binningBeta, value_type desiredWidth)
    : debug_name_m(debug_name)
    , numBins_m(numBins)
    , totalBinWidth_m(totalBinWidth)
    , binningAlpha_m(binningAlpha)
    , binningBeta_m(binningBeta)
    , desiredWidth_m(desiredWidth) {

    instantiateHistograms(); // Allocates Kokkos views for histogram, bin widths, and post
    -sum.
    initTimers();            // Initializes performance timers.
}
```

It saves all parameters associated with the binning routine locally, even the cost function parameters. That makes it independent of the OPALX `Options` values and therefore usable in other contexts, like IPPL[35].

The `init()` method synchronizes views and initializes the bin widths and cumulative sum:

```
void init() {
    sync(); // Synchronizes host/device views if using DualView.
    initConstBinWidths(totalBinWidth_m); // Sets a uniform bin width.
    initPostSum(); // Computes the prefix sum (post-sum) for bin counts.
}
```

The `sync` function is necessary if the class is used in `DualView` mode:

```
if constexpr (UseDualView) {
    if (histogram_m.need_sync_host() && histogram_m.need_sync_device()) {
        std::cerr << "Warning: Histogram was modified on host AND device -- overwriting
    changes on host." << std::endl;
    }
    if (histogram_m.need_sync_host()) {
        histogram_m.sync_host();
    } else if (histogram_m.need_sync_device()) {
        histogram_m.sync_device();
    } // else do nothing
}
```

The function will do nothing if called in normal `View` mode and make sure that data is synced between device and host if used in `DualView` mode. This only takes care of `histogram_m`. The widths and post sum attributes are only modified from inside the class and can therefore be synced automatically[36] when changed.

The number of particles in a given bin is accessed via `getNPartInBin()`:

```
size_type getNPartInBin(bin_index_type binIndex) {
    if constexpr (UseDualView) {
        return histogram_m.h_view(binIndex);
    } else if (std::is_same<typename hview_type::memory_space, Kokkos::HostSpace>::value)
    {
        return histogram_m(binIndex);
    } else {
        std::cerr << "Warning: Accessing without DualView might be inefficient!" << std::
    endl;
        Kokkos::View<size_type, Kokkos::HostSpace> host_scalar("host_scalar");
        Kokkos::deep_copy(host_scalar, Kokkos::subview(histogram_m, binIndex));
        return host_scalar();
    }
}
```

Listing 1: *Accessing Particle Count in a Bin*

This function is necessary to always allot the most efficient access possible if the data is needed on host. Since the internal view can be saved in whatever memory space, it is necessary to take into account possible device/host synchronization.

[35]This is the case for all of the binning class. In order to use it, one only has to initialize it with an IPPL particle bunch and call the necessary member functions.

[36]Automatically meaning that the `Histogram` class calls the corresponding `sync_...` function automatically if applicable.

A.2.2 Binning Variable Selector Concept

In order to group particles into bins, we first need to make a choice which attribute or variable should be used to bin in. OPAL uses z position as explained in 2.6, while this thesis focuses on the velocity component as explained in 2.6.2. However, the algorithm introduces a variable selector concept that is completely adaptable and can be modified to bin in values of a function dependent on arbitrary particle attributes.

In this section, the design and functionality of the `CoordinateSelector` struct is explained. The provided example struct provides a method to extract a specific coordinate value (for example, position or velocity) for each particle, which can then be used for binning. It is templated on the particle bunch type and requires certain types and an `operator()` with a fixed signature to be defined. The `CoordinateSelector` is defined as follows:

```
template <typename bunch_type>
struct CoordinateSelector {
    /// Type representing the value of the binning variable (e.g., position or velocity).
    using value_type = typename bunch_type::Layout_t::value_type;

    /// Type representing the size of the particle bunch.
    using size_type = typename bunch_type::size_type;

    /// Type representing the view of particle positions.
    using position_view_type = typename bunch_type::particle_position_type::view_type;

    position_view_type data_arr; ///< Kokkos view of the particle data array.
    const int axis;              ///< Index of the coordinate axis to use for binning.

    /**
     * @brief Constructs a CoordinateSelector for a specific axis.
     *
     * @param axis_ Index of the axis to use for binning (e.g., 0 for x, 1 for y, 2 for z)
    .
     */
    CoordinateSelector(int axis_) : axis(axis_) {}

    /**
     * @brief Updates the data array view with the latest particle data.
     *
     * This function updates data_arr to reflect the latest particle data in the
     * container by retrieving the view from bunch->P. This ensures data_arr is
     * synchronized with any recent changes to the particle data.
     */
    void updateDataArr(std::shared_ptr<bunch_type> bunch) {
        data_arr = bunch->P.getView();
    }

    /**
     * @brief Returns the value of the binning variable for a given particle index.
     *
     * This function is called by the binning routines to obtain the value used
     * for binning.
     *
     * @param i Index of the particle in the data array.
     * @return value_type Normalized value of the specified coordinate axis.
     */
    KOKKOS_INLINE_FUNCTION
    value_type operator()(const size_type& i) const {
        const value_type value = fabs(data_arr(i)[axis]);
        return value / sqrt(1 + value * value); // Normalize to v/c, so v in [0, 1]
```

```
46     }
47 };
```

It is designed with the following key features:

Template Parameter. The struct is templated on `bunch_type`, which represents the particle bunch and provides required type definitions such as `value_type` and `size_type`.

Member `data_arr`. This `Kokkos::View` stores the particle data array and is updated via `updateDataArr()` method to ensure that any changes (for instance, due to reallocation in `bunch->create()`) are seen.

Member `axis`. This integer determines the coordinate axis (e.g., x, y, or z) that is used for binning.

`updateDataArr()` Method. Updates the internal data view by extracting the current particle data from the particle bunch.

`operator()` Overload. Returns the binning value for a given particle index. Needs to be available on device execution space. In this implementation, the value is obtained by taking the absolute value of the velocity at the specified axis, then calculating normalized velocity from it:

$$\text{normalized value} = \frac{|\text{value}|}{\sqrt{1 + \text{value}^2}},$$

which ensures the result lies in the interval $[0, 1]$ (i.e., normalized to $\frac{v}{c}$) and represents velocity in OPALX. Note that `bunch->P` has units $[\beta\gamma]$ in OPALX.

Below is an example that shows how to instantiate and use the `CoordinateSelector` in the context of particle binning:

```
CoordinateSelector<bunch_type> selector(2); // Selects the z-axis for binning
selector.updateDataArr(bunch); // Updates the internal view with the latest particle data

// Obtain the binning value for particle at index i:
auto binValue = selector(i);
```

The selector is designed such that it needs to be initialized only once in the beginning together with the `AaptBins` instance. Its design allows for easy extension or modification – for example by defining custom selectors for other variables such as energy. A user can provide their own selector struct when the following members are given:

- `operator()` takes an index of type `size_type`, returns a `value_type` and is implemented as a `KOKKOS_INLINE_FUNCTION` if used on GPU.
- Needs to implement `updateDataArr()`. This function is called every time before the selector is used.
- It needs to be assignable. If implicit member copies are not possible, it needs to implement a proper copy constructor.

A.2.3 Pre-Compiled `Kokkos` Reducer Objects

In normal `Kokkos` GPU kernels dynamic memory allocation is not allowed[37]. To enable flexible reduction operations with arrays of varying sizes, a family of reducer objects is pre compiled to be used together with `Kokkos::parallel_reduce`. The following code defines a templated structure, `ArrayReduction`, and uses template metaprogramming (object generation at compile time with `std::variant` and `std::integer_sequence`) to generate a variant type that can hold any of the precompiled reducer types and sizes.

The `ArrayReduction` structure is a templated reducer object that holds a statically allocated array (size is known at compile time). This object is intended to be used as a reducer object inside `Kokkos`'s parallel reduction.

```
template<typename SizeType, typename IndexType, IndexType N>
struct ArrayReduction {
    SizeType the_array[N];

    // Default constructor: initialize all elements to zero.
    KOKKOS_INLINE_FUNCTION
    ArrayReduction() {
        for (IndexType i = 0; i < N; i++ ) {
            the_array[i] = 0;
        }
    }

    // Copy constructor: perform element-wise copy.
    KOKKOS_INLINE_FUNCTION
    ArrayReduction(const ArrayReduction& rhs) {
        for (IndexType i = 0; i < N; i++ ) {
            the_array[i] = rhs.the_array[i];
        }
    }

    // Assignment operator: element-wise assignment.
    KOKKOS_INLINE_FUNCTION
    ArrayReduction& operator=(const ArrayReduction& rhs) {
        if (this != &rhs) {
            for (IndexType i = 0; i < N; ++i) {
                the_array[i] = rhs.the_array[i];
            }
        }
        return *this;
    }

    // Addition operator: accumulates values from another reducer.
    KOKKOS_INLINE_FUNCTION
    ArrayReduction& operator+=(const ArrayReduction& src) {
        for (IndexType i = 0; i < N; i++ ) {
            the_array[i] += src.the_array[i];
        }
        return *this;
    }
};
```

Explanation of the struct:

[37] It is kind of possible when using for example team scratch memory as it was done in 3.2 for the team based reduction. For this context, `parallel_reduce`, it is not applicable.

- The template parameters are `SizeType` (for the array's data type), `IndexType` (for indexing the array), and `N`, the size of the array.
- The default constructor initializes the array elements to zero, which is needed for reduction identity.
- The copy constructor and assignment operator ensure a proper element-wise copy.
- The `operator+=` is defined so that two `ArrayReduction` objects can be accumulated – this is required by `Kokkos`.

To pre-compile reducer objects for different array sizes, we define a compile-time constant `maxArrSize`. This constant limits the range of possible array sizes for the reducer.

```
template<typename IndexType>
constexpr IndexType maxArrSize = 5; // Adjust as needed (higher = more compile times)
```

Since the array size is only known at runtime[38] (as `binCount`), we use template metaprogramming to pre-compile a variant type that contains reducer types for all sizes in the range $[1, \texttt{maxArrSize}]$.

```
template<typename SizeType, typename IndexType, typename Sequence>
struct ReductionVariantHelper;

template<typename SizeType, typename IndexType, IndexType... Sizes>
struct ReductionVariantHelper<SizeType, IndexType,
                              std::integer_sequence<IndexType, Sizes...>> {
    // The variant will contain ArrayReduction objects for sizes 1 through maxArrSize.
    using type = std::variant<ArrayReduction<SizeType, IndexType, Sizes + 1>...>;
};

// Type alias for easy usage.
template<typename SizeType, typename IndexType>
using ReductionVariant = typename ReductionVariantHelper<SizeType, IndexType,
    std::make_integer_sequence<IndexType, maxArrSize<IndexType>>>::type;
```

Explanation of this step:

- The templated struct `ReductionVariantHelper` is declared but not defined.
- A specialization is provided that accepts a `std::integer_sequence` of indices.
- Using parameter pack expansion, we generate a `std::variant` containing `ArrayReduction` types for each size from 1 to `maxArrSize`.
- The type alias `ReductionVariant` makes it convenient to refer to this variant type.

Next, the reducer object is created[39] based on on `binCount`. To create a reducer object with the correct size at runtime, we define a recursive helper function.

```
template<typename SizeType, typename IndexType, IndexType N>
ReductionVariant<SizeType, IndexType> createReductionObjectHelper(IndexType binCount) {
```

[38] The adaptive histogram merging algorithm knows the number of bins only after the merged configuration is calculated.

[39] Technically, it is not created and only accessed. It was created at compile time.

```
    if constexpr (N > maxArrSize<IndexType>) {
        throw std::out_of_range("binCount is out of the allowed range");
    } else if (binCount == N) {
        return ArrayReduction<SizeType, IndexType, N>();
    } else {
        return createReductionObjectHelper<SizeType, IndexType, N + 1>(binCount);
    }
}
```

It is templated on the desired bin size `N` and checks whether `N` is too big for the pre-compiled reducer objects. If the passed bin count matches the templated `N`, it can return the reducer objects. Otherwise, it goes one layer deeper into the recursion in `N` and checks again. If `N` is positive and in the range of available reducers, it will return the correct reducer object eventually. Note that the template parameter of the returned reducer needs to come from a templated parameter of the function, so it needs to be known at compile time. This workaround is necessary, since `binCount` is a runtime value.

Finally, we define a wrapper function to initiate the recursive search starting from size 1.

```
template<typename SizeType, typename IndexType>
ReductionVariant<SizeType, IndexType> createReductionObject(IndexType binCount) {
    return createReductionObjectHelper<SizeType, IndexType, 1>(binCount);
}
```

The function takes the desired `binCount`, which is a runtime value, and calls the recursive helper starting with `N = 1`. Note that the return value is not the reducer object itself. It is a `std::variant` containing the one reducer object where the templated `N` matches the passed `binCount`. Later, this variant can be used together with `std::visit` to access the reducer. In practice this might look as follows.

```
// ensure (binCount <= maxArrSize) beforehand; otherwise runtime error
auto to_reduce = createReductionObject<size_type, bin_index_type>(binCount);
std::visit([&](auto& reducer_arr) {
    executeInitLocalHistoReduction(reducer_arr);
}, to_reduce);
```

Note that it uses `auto` to determine the reducer type. This is the easiest solution if we consider that the return type can be different. Alternatively, one could use a template on `std::visit` to get the reducer type from the template instantiation after `reducer_arr` is passed to the lambda. However, using `auto` is the easiest solution. Note that the `execute...` function is explicitly templated on the reducer type, which is necessary to use it inside a normal `Kokkos` kernel:

```
Kokkos::parallel_reduce("initLocalHist", bunch_m->getLocalNum(),
    KOKKOS_LAMBDA(const size_type& i, ReducerType& update) {
        bin_index_type ndx = binIndex(i);  // Determine the bin index for this particle
        update.the_array[ndx]++;           // Increment the corresponding bin count in the
     reduction array
    }, Kokkos::Sum<ReducerType>(to_reduce)
);
```

Here, `to_reduce` is the reduction object instantiated inside `std::visit`.

In summary, this design enables the usage of `Kokkos::parallel_reduce` over arrays whose size is defined only at runtime by ensuring that all necessary array sizes are pre-compiled while selecting the correct type at runtime via a variant.

A.2.4 Histogram Merging using Dynamic Programming and Parallelization Consideration

Section 3.5 explains how the adaptive histogram is generated, meaning it explains the adaptive binning algorithm. Before, the algorithm is only discussed on a high level with its naive recursive implementation outlined in snippet 1. As discussed, this leads to exponential runtime, which is not an applicable solution. However, runtime can be improved drastically with the help of memoization or ultimately dynamic programming. The following aims to explain in detail how the final algorithm looks like and how one can derive it from the recursive implementation.

The goal is to merge neighboring bins such that the resulting merged bins minimize a cost function based on bin counts and widths. This DP (dynamic programming) solution is derived from a naive recursive formulation described by 1. There, `prefixCount` and `prefixWidth` are arrays that store cumulative sums of counts and widths of the fine histogram[40]. `computeCost()` computes the cost of merging bins from index `i` to `k` as explained in 3.6. The recursion explores all possible partitions and decides on the best one, which results in an exponential number of calls.

Using Dynamic Programming. To avoid redundant computations, the recursive approach is transformed into a DP solution by storing intermediate results. We define two DP arrays:

- `dp(k)`: The minimal cost possible when calculating the optimal merged histogram configuration covering the interval $[0, k-1]$.
- `prevIdx(k)`: The index at which the last optimal merge occurred for the interval $[0, k-1]$.

The cost array will be used to asses if a tested configuration leads to a lower cost or not. *prevIdx* is used to reconstruct the histogram configuration leading to the lowest possible cost.

Computing Prefix Sums. Before filling the DP arrays, we compute the prefix sums of bin counts and widths. This allows efficient computation of the sum over any subinterval:

```
1 hview_type prefixCount("prefixCount", n+1);
2 hwidth_view_type prefixWidth("prefixWidth", n+1);
3 prefixCount(0) = 0;
4 prefixWidth(0) = 0;
5 for (bin_index_type i = 0; i < n; ++i) {
6     prefixCount(i+1) = prefixCount(i) + oldHistHost(i);
7     prefixWidth(i+1) = prefixWidth(i) + oldBinWHost(i);
```

[40]Here we use 128 uniformly distributed bins for the most part. However, the algorithm can be used with arbitrary bin configurations – also non uniform.

```
}
```

Here, `oldHistHost` and `oldBinWHost` are the host views of the fine histogram counts and bin widths. This makes it also clear that the algorithm (for now) only works on CPU.

Initializing the DP Arrays. We then allocate and initialize the DP arrays. A large constant value (here, `largeVal`) is used to initialize it with a very high initial cost:

```
Kokkos::View<value_type*, Kokkos::HostSpace> dp("dp", n+1);
Kokkos::View<int*,        Kokkos::HostSpace> prevIdx("prevIdx", n+1);
value_type largeVal = std::numeric_limits<value_type>::max() / value_type(2);
for (bin_index_type k = 0; k <= n; ++k) {
    dp(k) = largeVal;
    prevIdx(k) = -1;
}
dp(0) = value_type(0);  // Base case: zero cost for an empty interval.
```

Filling the DP Arrays in $\mathcal{O}(n^2)$. We iterate over all possible ending indices `k` for testing the cost if we were to merge indices i to k and. Consider every possible partition index `i`:

```
for (bin_index_type k = 1; k <= n; ++k) {
    for (bin_index_type i = 0; i < k; ++i) {
        size_type  sumCount = prefixCount(k) - prefixCount(i);
        value_type sumWidth = prefixWidth(k) - prefixWidth(i);
        value_type segCost  = largeVal;
        if (sumCount > 0) {
            segCost = cost(sumCount, sumWidth, totalNumParticles);
        }
        value_type candidate = dp(i) + segCost;
        if (candidate < dp(k)) {
            dp(k) = candidate;
            prevIdx(k) = i;
        }
    }
}
```

Snippet 2: *Main dynamic programming loop calculating the optimal merged histogram configuration based on the provided cost function* `cost()`.

`sumCount` and `sumWidth` represent the total particle count and bin width for the interval $[i, k]$, where i and k represent bin indices from the fine histogram. `cost()` computes the cost for merging bins in that interval. The DP array `dp` is updated every time a lower cost is found, and `prevIdx` records the corresponding index marking the starting index to merge from leading to that cost.

Reconstructing the optimal merge partition. After filling the DP arrays, the optimal partition (i.e., the merge boundaries) is reconstructed by backtracking through `prevIdx`:

```
std::vector<int> boundaries;
boundaries.reserve(20);
int cur = n;
while (cur > 0) {
    int start = prevIdx(cur);
    if (start < 0) {
        std::cerr << "Error: prevIdx(" << cur << ") < 0. Merging aborted." << std::endl;
        break;
    }
    boundaries.push_back(start);
    cur = start;
```

```
}
std::reverse(boundaries.begin(), boundaries.end());
boundaries.push_back(n);  // Final boundary.
```

This step yields the new bin boundaries by backtracking from the last bin to the first. The final number of bins is usually below 20 in real life scenarios of OPALX, which is why 20 places were reserved in the `std::vector`.

Building new histogram arrays and mapping old to new bins indices. Using the computed boundaries, we form new arrays for the merged bin counts and widths:

```
Kokkos::View<size_type*,  Kokkos::HostSpace> newCounts("newCounts", mergedBinsCount);
Kokkos::View<value_type*, Kokkos::HostSpace> newWidths("newWidths", mergedBinsCount);
for (size_type j = 0; j < mergedBinsCount; ++j) {
    bin_index_type start = boundaries[j];
    bin_index_type end   = boundaries[j+1] - 1;  // Inclusive.
    size_type  sumCount  = prefixCount(end+1) - prefixCount(start);
    value_type sumWidth  = prefixWidth(end+1) - prefixWidth(start);
    newCounts(j) = sumCount;
    newWidths(j) = sumWidth;
}
```

Furthermore, a lookup table is generated to map each bin index in the fine histogram to the new bin index in the merged histogram:

```
hindex_transform_type oldToNewBinsView("oldToNewBinsView", n);
for (size_type j = 0; j < mergedBinsCount; ++j) {
    bin_index_type startIdx = boundaries[j];
    bin_index_type endIdx   = boundaries[j+1]; // Exclusive.
    for (bin_index_type i = startIdx; i < endIdx; ++i) {
        oldToNewBinsView(i) = j;
    }
}
```

This lookup table is returned by the `mergBins()` function inside the `Histogram` class and can be applied after calling the merging routine:

```
bin_view_type binIndex                  = getBinView();
hindex_transform_type adaptLookup       = globalBinHisto_m.mergeBins();
dindex_transform_type adaptLookupDevice = dindex_transform_type("adaptLookupDevice",
                                                                currentBins_m);
Kokkos::deep_copy(adaptLookupDevice, adaptLookup);
setCurrentBinCount(globalBinHisto_m.getCurrentBinCount());

// 2. Map old indices to the new histogram ("Rebin")
Kokkos::parallel_for("RebinParticles", bunch_m->getLocalNum(), KOKKOS_LAMBDA(const
    size_type& i) {
    bin_index_type oldBin = binIndex(i);
    binIndex(i) = adaptLookupDevice(oldBin);
});
```

The merging algorithm is called on the global histogram. Then the bin indices are updated in the particle bunch using the provided index map and the local histogram can be built again as outlined in the diagram 13.

After constructing the new merged bin arrays, the old histogram data is overwritten and the post-sum is recalculated. This dynamic programming approach efficiently computes

the optimal merge configuration, enabling adaptive rebinning in simulations in OPALX avoiding the pitfalls of naive recursion.

Parallelization considerations. Apart from filling the prefix sums, there is only one part that can be parallelized effectively, which is part of the DP loop seen in snippet 2. The outer loop cannot be parallelized since the results of each iteration (the values inside `dp` and `prevIdx`) depend on the previous iteration. However, the inner loop can be implemented using `Kokkos::parallel_reduce` with a arg-minimum value reduction object to find the index i that leads to the minimum cost possible. However, the histogram is usually at most 128 bins big. Considering that one would need to copy some values to the device and then deal with overhead from launching the kernel, there was no observable computation time advantage. Even worse, the problem size is too small to even outperform the serial implementation on the tested systems. Therefore, it is not possible to speed up this part using GPUs.