1. Architecture Overview

1.1. Project Summary

SM_2D is a high-performance 2D deterministic transport solver for proton therapy dose calculation using CUDA-accelerated GPU computing. The project implements a hierarchical S-matrix solver with block-sparse phase-space representation.

In Plain English:
Imagine you're trying to predict exactly where thousands of tiny invisible particles (protons) will go when they shoot through the human body. SM_2D is like a super-powered calculator that figures this out by breaking the problem into millions of tiny pieces, solving each piece simultaneously on a graphics card (GPU), and tracking where every bit of energy ends up. This helps doctors plan cancer treatments more precisely.

1.1.1. Key Statistics

Metric	Value
Total Files	30+ C++ source files
CUDA Kernels	6 major kernels (K1-K6)
Lines of Code	15,000 lines
Memory per Simulation	3 GB GPU memory
Grid Size	Up to 200 × 640 cells

Table 1: System Metrics

1.2. Understanding C++ Concepts

What is C++?
C++ is a high-performance programming language that gives programmers direct control over computer hardware. Think of it like driving a manual transmission car instead of an automatic - you have more control and can go faster, but you need to know what you're doing.

Why C++ for SM_2D?
- **Speed**: C++ compiles directly to machine code, running as fast as possible
- **Control**: Direct access to memory and hardware features
- **GPU Integration**: Seamless integration with NVIDIA's CUDA for parallel computing

1.2.1. Classes and Objects

In Plain English:
A **class** is like a blueprint for a house. It defines what the house will look like and what it can do, but it's not a house itself. An **object** is the actual house built from that blueprint.

In SM_2D:
- **Class**: "Particle" - defines what a particle is (position, energy, direction)
- **Object**: A specific proton at a specific location with specific energy

1.2.2. Pointers and Memory

Technical Note:
A **pointer** is a variable that stores the memory address of another variable. Think of it like a forwarding address - it doesn't contain the mail itself, but tells you where the mail is. In C++, pointers are crucial for GPU computing because we need to transfer data between CPU memory (RAM) and GPU memory (VRAM). Pointers let us efficiently pass large arrays without copying them.

1.2.3. Templates

In Plain English:
Templates are like cookie cutters - you can use the same cutter with different types of dough. Instead of writing separate functions for integers, floats, and doubles, you write one template that works with any type.

In SM_2D, templates let our physics code work with different precision levels without rewriting everything.

1.2.4. Header Files (.hpp) vs Source Files (.cpp)

Header vs Source
Header files (.hpp): Like the table of contents and index of a book. They declare what functions and classes exist, but don't contain the actual code.

Source files (.cpp): Like the actual chapters of a book. They contain the implementation - the actual code that makes things work.

Why separate them?
- Faster compilation (only recompile what changes)
- Cleaner organization (interface vs implementation)
- Shared declarations (multiple files can include the same header)

1.3. System Architecture

1.3.1. Architecture Layers

The system is organized into six distinct layers:

1.3.1.1. Input Layer

Purpose: Load and prepare all configuration and physics data

In Plain English:
Like gathering all your ingredients before cooking. This layer reads the recipe (sim.ini), gets the physics data from NIST (like a professional cooking database), and checks for any special instructions from the command line.

Components:
- `sim.ini`: Configuration file with simulation parameters
- NIST PSTAR physics data: Real-world measurements of how protons interact with matter
- Command-line parameters: User overrides and options

NIST PSTAR Database
The National Institute of Standards and Technology (NIST) maintains the PSTAR database, which

contains precise measurements of how protons lose energy when traveling through different materials. Think of it as a giant lookup table created from decades of experiments. SM_2D uses this data instead of trying to calculate everything from scratch.

1.3.1.2. Core Layer

Purpose: Fundamental data structures that organize particle information

In Plain English:
This is the "filing system" of the simulation. Instead of throwing all particle information into a giant pile, it organizes everything into neat grids and blocks so we can find and process particles efficiently.

Components:

1. Energy/Angle Grids

   In Plain English:
   Imagine a spreadsheet where rows are different particle energies and columns are different directions. This grid lets us quickly look up properties for particles at any energy moving in any direction.

   Energy Grid: 256 bins from 0.1 to 250 MeV (log-spaced) **Angle Grid**: 512 bins from $-90°$ to $+90°$

Grid Type	Configuration
Energy Grid	• Range: 0.1 MeV ⟶ 250 MeV • Bins: 256 (log-spaced) • More detail at low energy
Angle Grid	• Range: $-90°$ ⟶ $+90°$ • Bins: 512 (linear-spaced) • Evenly distributed

Table 2: Energy and angle grid configuration

2. Block Encoding (24-bit)

   Block Encoding: Memory Compression
   Instead of storing "energy bin 1234 and angle bin 5678" as two separate numbers (taking 8 bytes), we pack them into a single 24-bit number (3 bytes). This saves 62.5% of memory!

   It's like writing "12:34" instead of "12 hours and 34 minutes" - same information, less space.

Field	Bits	Range
Energy bin (b_E)	Bits 12-23	0-4095
Angle bin (b_theta)	Bits 0-11	0-4095
Total: 24 bits (3 bytes)		

Table 3: 24-bit block encoding scheme

3. Phase-Space Storage (Hierarchical)

In Plain English:
Phase space is just a fancy physics term for "all possible states of a particle." It includes position, direction, and energy. Instead of storing every particle individually, we group particles into "blocks" based on their energy and direction, then only store the blocks that actually have particles.

Technical Note:
The hierarchical structure means: (1) Global phase space is divided into cells, (2) Each cell is divided into blocks based on energy/angle, (3) Each block contains 512 local bins for sub-cell position, (4) Only active blocks are allocated memory (sparse storage). This can reduce memory usage by 70-90%.

4. Bucket Emission (Inter-cell)

Buckets: Particle Transfer System
When particles cross from one cell to another, we don't immediately process them. Instead, we collect them in "buckets" for each destination cell. Think of it like a mail sorting room - letters going to different zip codes are sorted into different buckets, then each bucket is delivered in one trip.

This is much more efficient than delivering each letter individually!

1.3.1.3. Physics Layer

Purpose: Model how protons interact with matter

In Plain English:
This layer contains the "rules of the game" - how protons lose energy, how they scatter, how they get absorbed, etc. Each module is like a separate rulebook for one physical process.

Components:

1. Highland Multiple Coulomb Scattering (MCS)

In Plain English:
When protons pass through matter, they don't go in straight lines. They get deflected by atomic nuclei, bouncing around like a pinball. This is called "Multiple Coulomb Scattering" (MCS).

Analogy: Imagine walking through a crowded room. You'll bump into people and get pushed sideways, even if you're trying to walk straight. That's what happens to protons in tissue.

Highland Formula
The Highland formula predicts how much protons will scatter based on their energy and the material they're passing through. Higher energy = less scattering. Denser materials = more scattering.

$$\sigma_{\text{theta}} = \frac{13.6 \quad \text{MeV}}{\beta c p} \sqrt{\frac{s}{X_0}} \left[1 + 0.038 \ln\left(\frac{s}{X_0}\right)\right]$$

Don't worry if this looks scary - it just means "calculate scattering angle based on energy and material."

2. Vavilov Energy Straggling

In Plain English:
Protons don't lose energy at a perfectly constant rate. Sometimes they lose a little, sometimes a lot. This variation is called "straggling." It's like how your car's gas mileage varies from trip to trip even on the same route.

Why it matters: If we ignore straggling, our predictions would be too smooth and wouldn't match real-world measurements.

Technical Note:
Vavilov distribution is a more accurate model than the simpler Gaussian (normal) distribution. It accounts for the fact that sometimes a proton loses a LOT of energy in a single collision (a "delta ray"), creating a long tail in the distribution.

3. Nuclear Attenuation

Nuclear Reactions: Particle Disappears
Sometimes a proton hits an atomic nucleus head-on and gets absorbed or scattered out of the beam entirely. This is "nuclear attenuation." It's like a billiard ball hitting another ball so hard that it bounces off the table.

In SM_2D, we track this as "lost weight" - the particle's contribution to the dose is gone, but energy is conserved because we account for it.

4. R-based Step Control

In Plain English:
How far should a proton travel before we recalculate its properties? That's the "step size." We use the particle's range (R) - the total distance it can travel before stopping - to determine step size.

Rule: Step size = min(2% of remaining range, distance to cell boundary)

This means we take smaller steps when the particle is almost stopped (near the Bragg peak) for better accuracy.

5. Fermi-Eyges Lateral Spread

Technical Note:
The Fermi-Eyges theory calculates how proton beams spread out sideways as they penetrate tissue. It's based on the idea that multiple small scattering events add up to a Gaussian (bell curve) distribution. This is crucial for predicting the "penumbra" - the fuzzy edge of the beam where dose falls off gradually.

1.3.1.4. CUDA Pipeline

Purpose: Execute physics calculations on GPU in parallel

In Plain English:
If CPU computing is like one person doing math problems one at a time, GPU computing is like having 10,000 people all doing math problems simultaneously. The CUDA pipeline is the "assembly line" that organizes this parallel workforce.

Kernel	Description
K1: ActiveMask	Find which cells actually have particles Only process cells with particles (skip empty ones)
K2: Coarse Transport	Move high-energy particles quickly Large steps, simple physics (far from Bragg peak)
K3: Fine Transport	Move low-energy particles precisely (MAIN PHYSICS) Small steps, full physics (near Bragg peak) Energy loss, scattering, straggling, deposition
K4: Bucket Transfer	Move particles between cells Collect particles that crossed cell boundaries
K5: Conservation Audit	Check that nothing was lost Verify weight and energy conservation
K6: Swap Buffers	Prepare for next step Swap input/output buffers for next iteration

Table 4: CUDA kernel pipeline showing parallel execution flow

Key Features:
- K1: ActiveMask: Identifies active cells (skips empty regions)
- K2: Coarse Transport: Fast transport for high-energy particles
- K3: Fine Transport: Detailed physics for low-energy particles (main kernel)
- K4: Bucket Transfer: Moves particles between cells
- K5: Conservation Audit: Verifies physics conservation laws
- K6: Swap Buffers: Exchanges input/output for next step

Why Kernels?
A **kernel** is a function that runs on the GPU. Unlike regular functions that run once, kernels run thousands of times in parallel - once for each piece of data.

Analogy: A regular function is like one chef chopping vegetables. A GPU kernel is like 10,000 chefs each chopping one vegetable simultaneously.

1.3.1.5. Output Layer

Purpose: Collect results and generate reports

In Plain English:
After all the particle transport is done, we need to present the results in a useful form. This layer generates dose maps, depth-dose curves, and verifies that our simulation conserved energy properly.

Components:
- 2D Dose Distribution: Color map showing radiation dose at each point
- Depth-Dose Curve: Plot of dose vs depth (shows Bragg peak)
- Conservation Report: Verifies that energy in = energy out

1.4. Module Dependency Graph

1.4.1. Foundation Layer

LUT Module	Config Loader	Logger
…r_lut	…sim.ini	…Logging
…nist_loader	…CLI args	…Debug

Table 5: Foundation layer modules

Purpose: Provide basic infrastructure used throughout the codebase

1. LUT Module (`r_lut`, `nist_loader`)

   In Plain English:
   LUT stands for "Look-Up Table." Instead of calculating complex physics formulas every time, we pre-calculate values and store them in a table. It's like using a multiplication table instead of doing the math every time.

   What it does:
   - Loads NIST PSTAR stopping power data
   - Creates range-energy look-up tables
   - Provides fast interpolation for any energy

   Technical Note:
   Range-energy tables are built by integrating stopping power: $R(E) = \int_0^E \left(\frac{1}{\frac{\mathrm{dE}}{\mathrm{dx}}}\right)$ dE. The table uses log-spaced energy bins for better resolution at low energies where most of the interesting physics happens.

2. Config Loader

   Configuration Management
   The config loader reads `sim.ini` and creates a `Config` object that all other modules can access. This ensures everyone uses the same parameters.

   Example: If you set `grid.nx = 200` in sim.ini, every module will use 200 cells in the x-direction. No magic numbers, no confusion.

3. Logger

   In Plain English:
   The logger is like a flight recorder - it records everything that happens during the simulation. When something goes wrong, we can look at the logs to figure out what happened.

   Log levels:
   - DEBUG: Detailed info for developers
   - INFO: General progress updates
   - WARNING: Something unusual but not fatal
   - ERROR: Something went wrong

1.4.2. Data Structures Layer

Grids	Block Encoding	Local Bins	Psi Storage	Buckets
...Energy	...24-bit	...4D	...Hier	...Emit
...Angle	...Pack	...Sub	...Sparse	...Trans

Table 6: Data structure modules

1. Grids (Energy, Angle)

   In Plain English:
   Grids are the “coordinate system” for particle properties. Just like a map has latitude/longitude, we have energy/angle grids to categorize particles.

   Implementation:

```
class EnergyGrid {
    int n_bins;           // 256 bins
    double* E_bins;       // Array of bin edges
    int get_bin(double E); // Find bin for energy E
}
```

2. Block Encoding (24-bit ID)

   Technical Note:
   Block encoding is implemented using bit shifting and masking: `block_id = (b_E << 12) | b_theta`. To decode: `b_E = block_id >> 12; b_theta = block_id & 0xFFF`. This is extremely fast - a single CPU instruction.

3. Local Bins (4D sub-cell)

   In Plain English:
   Each cell is divided into 4×4 = 16 sub-cells in position (x, z), and each of those is divided into 8×4 = 32 sub-bins in (theta, E). Total: 512 local bins per block.

Structure	Configuration
Full Cell	Size: dx × dz
Sub-cells	4×4 = 16 (x, z position)
Per sub-cell	8 theta × 4 E = 32 bins
Total per block	16 × 32 = 512 bins

Table 7: Cell subdivision structure

4. Psi Storage (Hierarchical)

   Psi (Phase-Space Density)
   Ψ (pronounced “sigh”) represents how many particles are in a particular state. It’s like a 4D histogram where each bin counts particles with specific energy, angle, and position.

Why "Psi"?: It's the Greek letter ψ (ψ), traditionally used in transport theory to represent particle flux.

In Plain English:
Hierarchical storage means we don't allocate memory for empty bins. If a cell has no particles at certain energies/angles, we just don't store those blocks. This is "sparse storage" and can save 70-90% of memory.

5. Buckets (Emission)

Technical Note:
Buckets are implemented as hash maps from cell ID to particle list. When particles cross cell boundaries, K4 kernel appends them to the appropriate bucket. After transport, we iterate through buckets and deposit particles into their destination cells. This is O(n) instead of O(n^2).

1.4.3. Physics Module Layer

Highland MCS	**Energy Strag.**	**Nuclear**	**Step Ctrl**	**Fermi-Eyges**
...Scat.	...Vavilov	...Abs.	...R-based	...Lateral
...Ang.	...Fluct.	...Loss	...Adapt.	...Spread

Table 8: Physics module dependencies

1. Highland MCS Module

   File: `src/physics/highland.hpp`

In Plain English:
This module calculates how much particles scatter when passing through matter. It implements the Highland formula, which is the standard method used in medical physics.

```
class HighlandMCS {
    double sigma_theta(double E, double material, double step);
    // Returns scattering angle standard deviation
}
```

2. Energy Straggling Module

   File: `src/physics/energy_straggling.hpp`

Vavilov Distribution
The Vavilov distribution describes energy loss fluctuations. It has two parameters:
- κ: Controls the shape (small = Gaussian-like, large = Landau-like)
- β^2: Controls the width

In SM_2D, we use the randlib library to sample from this distribution.

3. Nuclear Module

   File: `src/physics/nuclear.hpp`

In Plain English:
This module handles nuclear reactions - when a proton hits an atomic nucleus and gets absorbed or scattered away. We track this as “lost weight” to ensure energy conservation.

4. Step Control Module

   File: `src/physics/step_control.hpp`

Technical Note:
Adaptive step sizing is crucial for accuracy. We use $\mathrm{d}s = \min(0.02R, \mathrm{d}x, \mathrm{d}z)$. The 2% rule ensures we have at least 50 steps throughout the particle’s range, enough to accurately capture the Bragg peak. The cell boundary terms ensure we don’t skip over boundaries.

5. Fermi-Eyges Module

   File: `src/physics/fermi_eyges.hpp`

In Plain English:
This module calculates how proton beams spread out sideways. It’s important for predicting the “penumbra” - the fuzzy edge where the beam falls off.

1.4.4. CUDA Kernels Layer

K1	K2	K3	K4	K5	K6
ActiveMask	Coarse	Fine	Bucket	Audit	Swap
Mask	Trans.	Trans.	Trans.	Check	Buf

Table 9: CUDA kernel pipeline showing sequential execution

1. K1: ActiveMask

   File: `src/cuda/kernels/k1_activemask.cu`

In Plain English:
Before doing any work, we need to know which cells actually have particles. K1 creates a “mask” - a list of active cells. This way, K2 and K3 can skip empty cells and save time.

Algorithm:

```
For each cell:
  If cell has particles → mark as active
  Else → skip
Create list of active cells
```

2. K2: Coarse Transport

   File: `src/cuda/kernels/k2_coarsetransport.cu`

Coarse vs Fine Transport
Coarse: Fast, less accurate, for high-energy particles far from the Bragg peak **Fine**: Slower, more accurate, for low-energy particles near the Bragg peak

Using both gives us speed where we can afford it and accuracy where we need it.

3. K3: Fine Transport (MAIN PHYSICS)

   File: `src/cuda/kernels/k3_finetransport.cu`

   In Plain English:
   This is where most of the physics happens. K3 applies all five physics processes (energy loss, scattering, straggling, nuclear attenuation, energy deposition) to every particle in every active cell.

   Technical Note:
   K3 is the most computationally expensive kernel. It processes 512 local bins per block, applying physics calculations to each. This is where the GPU parallelism really pays off - thousands of threads all doing physics simultaneously.

4. K4: Bucket Transfer

   File: `src/cuda/kernels/k4_transfer.cu`

   In Plain English:
   After transport, some particles have crossed cell boundaries. K4 collects these particles and moves them to their destination cells using the bucket system.

5. K5: Conservation Audit

   File: `src/cuda/kernels/k5_audit.cu`

   Conservation Laws
   In physics, certain quantities are conserved (never created or destroyed):
   - Energy: Energy in = energy out + energy deposited
   - Weight: Particle weight in = weight out + weight absorbed

   K5 checks these laws at every step to catch numerical errors.

6. K6: Swap Buffers

   File: `src/cuda/kernels/k6_swap.cu`

   In Plain English:
   After each step, we need to swap input and output buffers. What was "output" becomes "input" for the next step. It's like using two whiteboards - write on one while reading from the other, then swap.

1.4.5. Sources Layer

Pencil Source	Gaussian Source
…Ideal beam	…Realistic
…Delta func	…Sigma width
…Testing	…Clinical

Table 10: Source module options

1. Pencil Source

File: `src/source/pencil_source.cpp`

In Plain English:
A "pencil beam" is an idealized beam with zero width - all particles start at exactly the same position with the same direction. It's useful for testing and validation because it's perfectly predictable.

2. Gaussian Source

File: `src/source/gaussian_source.cpp`

Technical Note:
Real proton beams aren't perfect pencils - they have a finite width described by a Gaussian distribution: $f(x) = \left(\frac{1}{\sigma\sqrt{2\pi}}\right) \exp\left(-x\frac{2}{2\sigma^2}\right)$. This source models that realistic beam profile for clinical accuracy.

1.4.6. Boundaries Layer

Boundaries	Loss Tracking
Reflective…	Where…
Absorbing…	How much…
Periodic…	Why…

Table 11: Boundary handling modules

1. Boundary Conditions

File: `src/boundary/boundaries.cpp`

In Plain English:
What happens when particles hit the edge of the simulation? That depends on the boundary condition:
- **Absorbing**: Particle disappears (like leaving the room)
- **Reflective**: Particle bounces back (like hitting a mirror)
- **Periodic**: Particle wraps around (like Pac-Man)

2. Loss Tracking

File: `src/boundary/loss_tracking.cpp`

Why Track Losses?
Particles can leave the simulation for different reasons:
- **Boundary loss**: Left the simulation volume
- **Cutoff loss**: Energy below threshold
- **Nuclear loss**: Absorbed by nucleus

Tracking these separately helps us validate our physics and understand where our approximations break down.

1.4.7. Audit Layer

Conservation	Global Budget	Reporting
Energy...	Sum...	CSV...
Weight...	Track...	Console...
Momentum...	Time...	File...

Table 12: Audit module components

1. Conservation Check

   File: `src/audit/conservation.cpp`

   In Plain English:
   This module verifies that our simulation obeys the laws of physics. After every step, it checks:
   - Total energy before = total energy after + energy deposited
   - Total particle weight before = total weight after + weight absorbed

2. Global Budget

   File: `src/audit/global_budget.cpp`

   Technical Note:
   The global budget tracks cumulative quantities over the entire simulation. It maintains running sums of energy deposited, weight absorbed, and particles lost. This is used for the final conservation report and for detecting gradual numerical drift.

3. Reporting

   File: `src/audit/reporting.cpp`

   In Plain English:
   After the simulation finishes, we need to present the results. The reporting module generates:
   - Console summary: Quick overview
   - CSV files: Detailed data for analysis
   - Plots: Visual representations

1.4.8. Validation Layer

In Plain English:
Before trusting our simulation for clinical use, we need to validate it - prove that it produces correct results. The validation layer compares our results against known analytic solutions and experimental data.

Validation Tests:
- Bragg Peak Validation: Compare depth-dose curve against analytic prediction
- Lateral Spread Validation: Compare penumbra against Fermi-Eyges theory
- Determinism Test: Run twice, verify identical results

1.5. Memory Layout

1.5.1. GPU Memory Breakdown

Buffer	Size	Type
PsiC_in/out	1.1 GB each	float32
EdepC	0.5 GB	float64
AbsorbedWeight_cutoff	0.25 GB	float32
AbsorbedWeight_nuclear	0.25 GB	float32
AbsorbedEnergy_nuclear	0.25 GB	float64
BoundaryLoss	0.1 GB	float32
ActiveMask/List	0.5 GB	uint8/uint32

Table 13: GPU Memory Layout

Total: 4.3 GB GPU memory

1.5.2. Memory Organization Visualization

In Plain English:
Think of GPU memory like a giant warehouse. The "PsiC_in" and "PsiC_out" are the main storage areas where we keep all the particle information. We need two of them so we can read from one while writing to the other (like having two whiteboards).

The other arrays are like specialized storage rooms - one for tracking energy deposition, one for tracking losses, etc.

1.5.3. Why Different Data Types?

Data Type Selection
float32: 32-bit floating point (6-7 decimal digits)
- Used for phase space (PsiC)
- Smaller memory, faster computation
- Sufficient precision for transport

float64: 64-bit floating point (15-16 decimal digits)
- Used for energy deposition (EdepC)
- Higher precision for accumulation
- Prevents round-off error in sums

uint8/uint32: Unsigned integers
- Used for masks and indices
- No negative values needed
- Compact storage

1.6. Phase-Space Representation

1.6.1. 4D Phase Space

In Plain English:
"Phase space" is physics jargon for "all the information needed to describe a particle's state." For us, that's 4 dimensions:
1. Position in x (transverse)
2. Position in z (depth)
3. Direction theta (angle)
4. Energy E

1.6.2. Block Encoding (24-bit)

Field	Bits	Range
Energy bin (b_E)	Bits 12-23	0-4095
Angle bin (b_theta)	Bits 0-11	0-4095
Total: 24 bits (3 bytes)		Savings: 62.5%

Table 14: Block encoding details

Parameter	Value
Energy bin	1500 (of 256 bins)
Compressed index	1500
12-bit binary	0101 1101 1100
Angle bin	800 (of 512 bins)
Compressed index	800
12-bit binary	0011 0010 0000
Combined block ID	0101 1101 1100 0011 0010 0000
Decimal value	6,080,832

Table 15: Block encoding example

1.6.3. Local Index (16-bit)

In Plain English:
Within each block, we have 512 local bins. To identify which bin a particle is in, we use a 16-bit local index calculated from 4 coordinates.

Component	Bits/Range
theta_local	8 values (0-7) ⟶ 3 bits
E_local	4 values (0-3) ⟶ 2 bits
x_sub	4 values (0-3) ⟶ 2 bits
z_sub	4 values (0-3) ⟶ 2 bits
Total: 9 bits (512 values fit in 16 bits)	

Table 16: Local index encoding

Step	Calculation
Formula	idx = theta_local + 8 × (E_local + 4 × (x_sub + 4 × z_sub))
Input values	z_sub = 1, x_sub = 2, E_local = 3, theta_local = 5
Inner bracket	2 + 4 × 1 = 6
Middle bracket	3 + 4 × 6 = 27
Final calculation	5 + 8 × 27 = 221
Result	Local index = 221

Table 17: Local index calculation example

Why Sub-Cells?
Sub-cells provide spatial resolution within each cell. Instead of assuming particles are uniformly distributed throughout the cell, we track their position at 4×4 sub-cell resolution. This improves accuracy, especially near the Bragg peak where dose gradients are steep.

It's like having a high-resolution camera instead of a low-resolution one - you can see finer details.

1.7. Physics Pipeline per Step

1.7.1. Step Sequence

In Plain English:
For each transport step, we apply a series of physics operations to every particle. This is the "physics pipeline" - a sequence of calculations that update particle properties.

Step	Operation
1. STEP CONTROL	How far should we go? ds = min(2% × R, dx, dz) Adaptive step size
2. ENERGY LOSS	Particle loses energy as it travels E_new = E_old - (dE/ds) × ds Continuous slowing down approximation
3. STRAGGLING	Add random energy fluctuation ΔE Vavilov(κ) Statistical fluctuation (not smooth!)
4. MCS	Particle direction changes θ_new = θ_old + σ_θ × N(0,1) Random angular deflection
5. NUCLEAR ATTENUATION	Some particles get absorbed W_new = W_old × exp(-σ × ds) Probabilistic absorption
6. ENERGY DEPOSITION	Deposit lost energy as dose E_dep = E_in - E_out This becomes the dose distribution!
7. BOUNDARY CHECK	Did particle leave the cell? If yes ⟶ emit to bucket If no ⟶ keep in local bin

Table 18: Physics pipeline showing sequential operations

1.7.2. Detailed Operation Descriptions

1. Step Control

Technical Note:
Step size is critical: too large ⟶ inaccurate, too small ⟶ slow. The 2% rule is a good compromise. We also check cell boundaries to ensure we don't skip over boundaries (which would mess up the bucket system).

2. Energy Loss

In Plain English:
As protons travel through matter, they lose energy by ionizing atoms. The rate of energy loss (dE/ds) is called the "stopping power" and comes from the NIST PSTAR database.

3. Straggling

Why Straggling Matters
If we ignored straggling, our depth-dose curve would be too smooth. The real world has fluctuations - some protons lose lots of energy, some lose little. Straggling creates the realistic "fuzziness" in the dose distribution.

It’s like how some cars get better gas mileage than others, even on the same route.

4. MCS

In Plain English:
Every time a proton passes near an atomic nucleus, it gets deflected. Over millions of interactions, this creates a net angular spread. The Highland formula predicts the standard deviation of this distribution.

5. Nuclear Attenuation

Technical Note:
Nuclear reactions are rare but important. When a proton hits a nucleus, it can be absorbed or scattered out of the beam. We model this as exponential decay: $W = W_0 e^{-\sigma \text{ ds}}$. The “lost” weight is tracked in the nuclear absorption array.

6. Energy Deposition

In Plain English:
The energy lost by protons doesn’t disappear - it gets deposited in the tissue as dose. This is what we’re trying to calculate! The energy deposition step adds up all the lost energy and stores it in the Edep array.

7. Boundary Check

Cell Crossing
Particles that cross cell boundaries need special handling. We can’t just write them to the neighbor’s array (that would cause race conditions in parallel). Instead, we collect them in buckets and process them after the main transport loop.

Think of it like checking out of a hotel before checking into another.

1.8. Directory Structure

Path	Description	
`run_simulation.cpp`	Main entry point - orchestrates everything	`sim.ini`
Configuration file - all simulation parameters	`visualize.py`	Python visualization - plot results
Directory	Contents	
`src/core/`	`grids.cpp` - Energy/angle grids (coordinate system for particles) `block_encoding.hpp` - 24-bit encoding (compress energy+angle) `local_bins.hpp` - 4D sub-cell partitioning (512 local bins/block) `psi_storage.cpp` - Hierarchical phase-space (sparse memory) `buckets.cpp` - Bucket emission (transfer particles between cells)	`src/physics/`

[illegible]

[illegible]

`nuclear.hpp` - Nuclear attenuation (particles absorbed by nuclei)

`step_control.hpp` - R-based step control (adaptive step sizing)

1.9. Key Design Principles

1.9.1. 1. Block-Sparse Storage

In Plain English:
Instead of storing every possible combination of energy, angle, and position (most of which are empty), we only store blocks that actually have particles. This is like only saving the pages you've written in a notebook, not all the blank pages too.

Storage Type	Blocks Stored	Memory Usage
Dense	All 16,384 blocks	16,384 × 512 × 4 bytes = 33.6 MB
Sparse	Only 2,304 blocks (14%)	2,304 × 512 × 4 bytes = 4.7 MB
Savings	86% memory reduction!	

Table 20: Dense vs sparse storage comparison

Memory Efficiency
In proton therapy, the phase space is mostly empty. At any given energy and angle, particles exist only in a small region of space. Block-sparse storage exploits this sparsity, reducing memory usage by 70-90%. This lets us simulate larger problems with the same GPU.

1.9.2. 2. Hierarchical Refinement

In Plain English:
We use different levels of detail in different regions. Far from the Bragg peak (high energy), we use coarse transport (fast, less accurate). Near the Bragg peak (low energy), we use fine transport (slow, more accurate). This gives us speed where we can afford it and accuracy where we need it.

Coarse Transport (K2)	Fine Transport (K3)
High energy (> 50 MeV)	Low energy (< 50 MeV)
Large steps (2% of range)	Small steps (2% of range)
Simple physics	Full physics (all processes)
10× faster	Maximum accuracy
Far from Bragg peak	Near Bragg peak

Table 21: Hierarchical transport strategy

1.9.3. 3. GPU-First Design

Why GPU?
A modern GPU has thousands of cores vs a CPU's 8-16 cores. For embarrassingly parallel problems like particle transport, GPUs provide 10-100× speedup.

The catch: GPU programming is harder. You need to:

- Minimize data transfer between CPU and GPU
- Use parallel algorithms (no sequential dependencies)
- Manage memory carefully (GPU memory is limited)

In Plain English:
Our design puts everything on the GPU. We transfer data to GPU once, run the entire simulation there, then transfer results back. This minimizes slow CPU-GPU transfers.

Component	Operation	Speed
CPU	Config, NIST LUT	N/A
Transfer	CPU ⟶ GPU (once)	10 GB/s (slow)
GPU	PsiC_in, Physics, Compute	900 GB/s (fast!)
Kernels	K1 ⟶ K2 ⟶ K3 ⟶ K4 ⟶ K5 ⟶ K6	10 TFLOP
Transfer	GPU ⟶ CPU (once)	10 GB/s (slow)
CPU	Results (PsiC_out, EdepC)	N/A

Table 22: CPU-GPU data flow showing why we minimize transfers

1.9.4. 4. Conservation by Design

In Plain English:
Physics has conservation laws - energy and matter cannot be created or destroyed, only changed. Our simulation enforces these laws by checking conservation at every step. If something doesn't add up, we know there's a bug.

Quantity	Before Step	After Step
Weight	W_in = 1.000	W_out = 0.950 + W_abs = 0.050 + W_boundary = 0.000 = 1.000 ✓
Energy	E_in = 150.0 MeV	E_out = 140.0 + E_dep = 10.0 + E_loss = 0.0 = 150.0 ✓

Table 23: Conservation checking example

Why Conservation Matters
If our simulation violates conservation, it means we have a bug. Energy or particles are appearing or disappearing, which is impossible. By checking conservation at every step, we catch bugs early instead of getting wrong results at the end.

It's like balancing your checkbook - if the numbers don't add up, you know there's an error.

1.9.5. 5. Modular Physics

In Plain English:
Each physics process is in its own file. This makes the code easier to understand, test, and validate. If we want to improve the scattering model, we only need to modify `highland.hpp`, not the entire codebase.

Module	Class	Method
`highland.hpp`	Highland	scatter()
`energy_straggling.hpp`	Vavilov	straggle()
`nuclear.hpp`	Nuclear	absorb()
`step_control.hpp`	StepCtrl	step()
`fermi_eyges.hpp`	FermiEyges	spread()
All used by K3 Kernel		

Table 24: Modular physics architecture

Benefits
Easy to test (unit tests per module)
Easy to validate (compare to theory)
Easy to improve (modify one module)
Easy to understand (clear separation)

Table 25: Advantages of modular design

1.10. References

1.10.1. Data Sources

- NIST PSTAR Database: https://physics.nist.gov/PhysRefData/Star/Text/PSTAR.html
  - Stopping powers and ranges for protons in various materials
- PDG 2024: https://pdg.lbl.gov/
  - Particle Data Group review of particle physics
  - Highland formula for multiple Coulomb scattering
- ICRU Report 73: Stopping Powers for Electrons and Positrons
  - Fundamental reference for energy loss calculations

1.10.2. Further Reading

- "The Physics of Proton Therapy" - Harald Paganetti
- "Monte Carlo Methods in Particle Transport" - Bielajew
- "CUDA C Programming Guide" - NVIDIA

—

SM_2D Architecture Documentation

Version 2.0.0 - Enhanced Edition