1. Core Data Structures Module

1.1. Overview: Phase-Space Particle Storage

This module solves the fundamental challenge of tracking millions of particles in proton therapy simulation. Instead of storing each particle individually (massive memory), we use a ***"Phase-Space Grid"*** approach: particles in similar regions of phase-space (similar position, direction, and energy) are grouped together and their weights are summed.

Particle State:

- Position: (x, z)
- Direction: Angle θ
- Energy: E
- Weight: w

1.2. 1. Energy Grid: Binning Particle Energy

1.2.1. Structure

The **EnergyGrid** divides the energy range (0.1 to 250 MeV) into 256 logarithmic bins.

```
struct EnergyGrid {
    const int N_E;                    // Number of energy bins (256)
    const float E_min;                // Minimum energy (0.1 MeV)
    const float E_max;                // Maximum energy (250.0 MeV)
    std::vector<float> edges;         // N_E + 1 bin edges (log-spaced)
    std::vector<float> rep;           // N_E representative energies
};
```

1.2.2. Why Logarithmic Spacing?

Energy distributions are ***not uniform***:

- Many particles have low energy (near end of range)
- Fewer particles have high energy (near beam entrance)

Logarithmic spacing provides:

- ***Fine resolution*** where needed (low energies)
- ***Coarse resolution*** where sufficient (high energies)

1.2.3. Grid Layout

[Logarithmic Energy Bins (256 bins, 0.1-250 MeV):]

Bin	Range (MeV)	Width (MeV)	Resolution
0	0.10 - 0.11	0.01	Very fine
1	0.11 - 0.12	0.01	Very fine
50	1.05 - 1.15	0.10	Fine
200	140 - 155	15	Coarse
255	220 - 250	30	Very coarse

Bins are **smaller at low energies** (high resolution) and **larger at high energies**.

1.2.4. Finding Bins

Binary search finds the correct bin in $O(\log N)$ time (8 comparisons for 256 bins):

```
int bin = grid.FindBin(150.0f);          // Returns ~200
float E_rep = grid.GetRepEnergy(bin);    // Returns ~147 MeV
```

Representative energy uses the ***geometric mean***: $\text{rep}_i = \sqrt{\text{edges}_i \times \text{edges}_i + 1}$

1.2.5. Memory Layout

[EnergyGrid Memory (4 KB):]

`edges[257]`	`log-spaced bin edges (4 bytes each)`
`rep[256]`	`representative energies (4 bytes each)`

Total: $(257 + 256) \times 4$ bytes ≈ 4.1 KB

1.3. 2. Angular Grid: Binning Particle Direction

1.3.1. Structure

The **AngularGrid** divides the angle range (-90° to +90°) into 512 uniform bins.

```
struct AngularGrid {
    const int N_theta;                  // Number of theta bins (512)
    const float theta_min;              // Minimum angle (-90°)
    const float theta_max;              // Maximum angle (+90°)
    std::vector<float> edges;           // N_theta + 1 bin edges (uniform)
    std::vector<float> rep;             // N_theta representative angles
};
```

1.3.2. Why Uniform Spacing?

Angular distributions are:

- ***Centered around 0°*** (forward-directed)
- ***Relatively uniform*** near the peak
- ***Narrow*** (most particles within ±30°)

Linear/uniform spacing is appropriate here.

1.3.3. Grid Layout

[Uniform Angular Bins (512 bins, −90° to +90°):]

Bin	Range (degrees)	Width
0	−90.0, −89.65	0.35°
255	−0.175, +0.175	0.35° (nearly forward)
511	+89.65, +90.0	0.35°

Each bin width: $\Delta\theta = 180\frac{^\circ}{512} \approx 0.35^\circ$

1.3.4. Finding Bins

Direct calculation in $O(1)$ time:

```
int bin = grid.FindBin(5.7f);               // Direct formula
float theta_rep = grid.GetRepTheta(bin);    // Arithmetic mean
```

For uniform bins, representative value is the ***arithmetic mean***: $\text{rep}_i = 0.5 \times (\text{edges}_i + \text{edges}_i + 1)$

1.3.5. Memory Layout

[AngularGrid Memory (8 KB):]

`edges[513]`	`uniform bin edges (4 bytes each)`
`rep[512]`	`representative angles (4 bytes each)`

Total: $(513 + 512) \times 4$ bytes ≈ 8.1 KB

1.4. 3. Block Encoding: Compressing Phase-Space Coordinates

1.4.1. The Big Idea

With 256 energy bins and 512 angle bins, we have $256 \times 512 = 131,072$ possible combinations. We **compress** these into a single 24-bit integer called the ***"Block ID"***.

Benefits:

1. ***Single Lookup***: One integer instead of two
2. ***Cache Efficiency***: Contiguous in memory
3. ***GPU-Friendly***: Fast integer operations
4. ***Memory Savings***: 24 bits vs 32 bits for two indices

1.4.2. Bit Layout

[32-bit word (lower 24 bits used):]

b_E (12 bits)	b_theta (12 bits)
Bits 12-23	Bits 0-11

Example: $b_\theta = 283$, $b_E = 150$

`block_id`	`= 0x095BB = 614,683`
`Encoding`	`(150 << 12) | 283 = 614,400 + 283`

1.4.3. Encoding/Decoding

```
// Encode: (b_theta, b_E) -> block_id
__host__ __device__ inline uint32_t encode_block(uint32_t b_theta, uint32_t b_E) {
    return (b_theta & 0xFFF) | ((b_E & 0xFFF) << 12);
}

// Decode: block_id -> (b_theta, b_E)
__host__ __device__ inline void decode_block(uint32_t block_id,
                                             uint32_t& b_theta,
                                             uint32_t& b_E) {
    b_theta = block_id & 0xFFF;
    b_E = (block_id >> 12) & 0xFFF;
}
```

1.4.4. Design Trade-offs

Requirement	Minimum Bits	Chosen	Headroom
Energy bins	8 bits (256)	12 bits	16× capacity
Angular bins	9 bits (512)	12 bits	8× capacity

Why 12 bits each? Powers of 2 are efficient, with room for expansion up to 4096 bins each.

1.5. 4. Local Bins: Fine-Grained Position Tracking

1.5.1. Four-Dimensional Subdivision

Each block contains 512 local bins tracking **four independent dimensions**:

Dimension	Symbol	Bins	Purpose
Local angle	θ_{local}	8	Angular variation
Local energy	E_{local}	4	Energy variation
X position	x_{sub}	4	Transverse location
Z position	z_{sub}	4	Depth location

Total: $8 \times 4 \times 4 \times 4 = 512$ local bins per block

1.5.2. Structure

[Cell (2mm × 2mm) divided into 4×4 sub-cells:]

Z \ X	0.0	0.5	1.0	1.5
0.0	0	1	2	3
0.5	4	5	6	7
1.0	8	9	10	11
1.5	12	13	14	15

Each position subdivided by θ_{local} (8) and E_{local} (4) Total: $16 \times 8 \times 4 = 512$ local bins

1.5.3. Encoding/Decoding

```
// Encode 4D coordinates to 16-bit index
__host__ __device__ inline uint16_t encode_local_idx_4d(
    int theta_local, int E_local, int x_sub, int z_sub
) {
    int inner = E_local + N_E_local * (x_sub + N_x_sub * z_sub);
    return static_cast<uint16_t>(theta_local + N_theta_local * inner);
}

// Decode 16-bit index to 4D coordinates
__host__ __device__ inline void decode_local_idx_4d(
    uint16_t lidx, int& theta_local, int& E_local, int& x_sub, int& z_sub
) {
    theta_local = static_cast<int>(lidx) % N_theta_local;
    int remainder = static_cast<int>(lidx) / N_theta_local;
    E_local = remainder % N_E_local;
    remainder /= N_E_local;
    x_sub = remainder % N_x_sub;
    z_sub = remainder / N_x_sub;
}
```

1.5.4. Position Offsets

Each sub-bin has a center position relative to cell center:

x_sub	Offset Formula	Offset (mm)	Position
0	-0.375 × dx	-0.75	Left edge
1	-0.125 × dx	-0.25	Left-center
2	+0.125 × dx	+0.25	Right-center

3	+0.375 × dx	+0.75	Right edge

(for cell size $\mathrm{dx} = 2.0$ mm, same pattern for z_{sub})

1.6. 5. PsiC: Hierarchical Phase-Space Storage

1.6.1. Structure

PsiC (Phase-space Cell) is the master data structure storing **all particle weights**:

```
struct PsiC {
    const int Nx;                    // Grid X dimension (e.g., 200)
    const int Nz;                    // Grid Z dimension (e.g., 640)
    const int Kb;                    // Max blocks per cell (32)

    // Storage layout: [cell][slot][local_bin]
    std::vector<std::array<uint32_t, 32>> block_id;  // Block IDs
    std::vector<std::array<std::array<float, LOCAL_BINS>, 32>> value;  // Weights

private:
    int N_cells;  // Nx × Nz (e.g., 128,000 cells)
};
```

1.6.2. Storage Hierarchy

[Four-Level Organization:]

Level 1: Spatial Grid ($N_x \times N_z$ cells)
Level 2: Individual Cell (i, j)
Level 3: Block Slots (up to 32 per cell)
Level 4: Local Bins (512 per slot)

[Example Cell (grid position 50, 100):]

Slot	Block ID	Meaning
0	0x000123	$\theta \approx -87.4°$, $E \approx 0.12$ MeV
1	0x000456	$\theta \approx -82.2°$, $E \approx 0.11$ MeV
2	EMPTY	Unused

1.6.3. Slot Allocation

```
int PsiC::find_or_allocate_slot(int cell, uint32_t bid) {
    // Pass 1: Check if block already exists
    for (int slot = 0; slot < Kb; ++slot) {
        if (block_id[cell][slot] == bid) {
            return slot;  // Reuse existing slot
        }
    }
    // Pass 2: Find empty slot
    for (int slot = 0; slot < Kb; ++slot) {
        if (block_id[cell][slot] == EMPTY_BLOCK_ID) {
            block_id[cell][slot] = bid;  // Allocate new slot
            return slot;
        }
    }
    return -1;  // ERROR: No space available!
}
```

1.6.4. Accessing Weights

```
// Get/set weight from specific local bin
float get_weight(int cell, int slot, uint16_t lidx) const;
void set_weight(int cell, int slot, uint16_t lidx, float w);
```

1.6.5. Memory Analysis

Component	Size	Notes
`block_id`	`16.4 MB`	`128,000 × 32 × 4 bytes`
`value`	`8.6 GB`	`128,000 × 32 × 512 × 4 bytes (max)`
Typical usage	`~1.1 GB`	`~13% of max (sparse)`

Why sparse? Most cells have < 10 active blocks (not 32), and many local bins are empty.

1.7. 6. OutflowBucket: Inter-Cell Particle Transfer

1.7.1. Purpose

When particles exit their current cell, the **OutflowBucket** temporarily holds them before transfer to neighboring cells.

1.7.2. Structure

```
struct OutflowBucket {
    static constexpr int Kb_out = 64;  // Max emission buckets (2× slots!)

    std::array<uint32_t, Kb_out> block_id;        // Block IDs
    std::array<uint16_t, Kb_out> local_count;     // Particle counts
    std::array<std::array<float, LOCAL_BINS>, Kb_out> value;  // Weights
};
```

$K_{\text{out}} = 64$ ensures capacity for temporary particle accumulation.

1.7.3. Four-Face System

[Each cell has 4 faces, each with its own bucket:]

Face	Direction	Index
+z	Forward	0
-z	Backward	1
+x	Right	2
-x	Left	3

Particles crossing boundaries are immediately sorted into the appropriate bucket by direction.

1.7.4. Transfer Flow

[Particle Movement Between Cells:]

Condition	Action
$x + \Delta x > +\frac{\text{dx}}{2}$	Emit to +x bucket (Cell $i+1, j$)
$x + \Delta x < -\frac{\text{dx}}{2}$	Emit to -x bucket (Cell $i-1, j$)
$z + \Delta z > +\frac{\text{dz}}{2}$	Emit to +z bucket (Cell $i, j+1$)
$z + \Delta z < -\frac{\text{dz}}{2}$	Emit to -z bucket (Cell $i, j-1$)

Source cell removes particle weight; destination cell receives it.

1.7.5. Bucket Operations

```
// Find existing block slot or allocate new one
int find_or_allocate_slot(uint32_t bid);

// Clear all bucket data (reset to EMPTY)
void clear() {
    for (int slot = 0; slot < Kb_out; ++slot) {
        block_id[slot] = EMPTY_BLOCK_ID;
        local_count[slot] = 0;
        for (int lidx = 0; lidx < LOCAL_BINS; ++lidx) {
            value[slot][lidx] = 0.0f;
        }
    }
}
```

1.7.6. Memory Layout

Component	Size
Per face	~131.5 KB (64 × 512 × 4 bytes)
All 4 faces per cell	~526 KB
Active working set	~526 KB (one cell at a time)

Important: Buckets are temporary buffers, only needed for the currently processing cell.

1.8. 7. Design Rationale

1.8.1. Why Block-Sparse Storage?

Most of the (θ, E) phase-space is empty in proton therapy (particles cluster in specific regions). Block-sparse storage allocates only for occupied regions, achieving 70% memory savings vs dense storage.

1.8.2. Why 24-bit Block Encoding?

Approach	Characteristics
Without encoding (hash table)	Slow on GPU (20-50 cycles)
With encoding (single integer)	Fast direct array access (1-2 cycles)

1.8.3. Why 512 Local Bins?

Option	Trade-off
Too few (e.g., 64)	Poor variance preservation, high error
Too many (e.g., 4096)	More memory, more particles needed
512 (chosen)	Good balance, clinical accuracy less than 1 percent

Breakdown: $8(\theta) \times 4(E) \times 4(x) \times 4(z)$

1.8.4. Why Fixed Slots per Cell?

Approach	Characteristics
Dynamic allocation	Fragmentation, unpredictable access
Fixed 32 slots	Contiguous memory, GPU coalescing, O(32) search

Fixed slots provide predictable memory layout crucial for GPU performance.

1.8.5. Why Logarithmic Energy Bins?

Proton energy loss follows $\mathrm{dE/dx} \approx \frac{1}{E}$ (Bethe-Bloch):

- Low energy → rapid loss → need fine bins
- High energy → slow loss → coarse bins OK

Logarithmic spacing gives $\Delta\frac{E}{E} \approx \mathrm{constant}$ (percentage resolution), matching physics.

1.8.6. Why 4 Face Buckets?

In 2D (X-Z plane), each cell has exactly 4 neighbors. Each face needs its own bucket because particles exit in different directions and cannot mix +x and -x outflows (different destinations).

1.9. Summary

1.9.1. Complete System

Component	Structure	Purpose
EnergyGrid	256 log bins (0.1-250 MeV)	Energy → bin index
AngularGrid	512 uniform bins (-90° to +90°)	Angle → bin index
Block Encoding	24-bit: b_θ (12) | b_E (12)	$(\theta_{\text{bin}}, E_{\text{bin}})$ → ID
Local Bins	512 bins: $8(\theta) \times 4(E) \times 4(x) \times 4(z)$	Fine position tracking
PsiC	128K cells, 32 slots, 512 bins	Main weight storage
OutflowBucket	4 faces, 64 slots each	Inter-cell transfer

1.9.2. Key Concepts

1. ***Hierarchical Organization***: Spatial grid → Cells → Blocks → Local Bins
2. ***Block-Sparse Storage***: Only allocate for occupied (θ, E) regions
3. ***Efficient Encoding***: Compress coordinates into integers
4. ***GPU Optimization***: Fixed-size arrays, predictable layouts
5. ***Physics-Driven Design***: Log energy bins, 4D local subdivision

1.9.3. Performance

Operation	Complexity	Notes
Find energy bin	$O(\log N_{\text{E}})$	Binary search: 8 steps
Find angle bin	$O(1)$	Direct calculation
Encode/decode block	$O(1)$	Bit operations
Find cell slot	$O(K_b)$	Linear search: 32 steps
Access weight	$O(1)$	Direct array access

1.9.4. Memory Efficiency

- Dense storage: 8.6 GB
- Block-sparse actual: 1.1 GB
- Savings: 87% (due to phase-space sparsity)

—

SM_2D Core Data Structures Documentation

Version 2.0 - Streamlined with Enhanced Tables

Source files:

src/include/core/grids.hpp src/include/core/block_encoding.hpp src/include/core/local_bins.hpp src/include/core/psi_storage.hpp src/include/core/buckets.hpp