1. ⊠어 데이터 구조 모듈

1.1. 위상 공간 입자 저장소 이해하기

⊠성자 치료 시뮬레이션에서 우리는 조직을 통과하는 수백만 개의 입자를 추적해야 합니다. 각 입자는 위치 (x, z), 방향 θ, 에너지 E, 통계적 가중치 w를 가집니다.

모든 입자를 개별적으로 저장하면 엄청난 메모리와 연산이 필요합니다. 대신 우리는 "⊠상 공간 그리드" 접근 방식을 사용합니다: "위상 공간"의 유사한 영역(유사한 위치, 방향, 에너지)에 있는 입자들을 함께 그룹화하고 그 가중치를 합산합니다.

—

1.2. 1. 에너지 그리드: 입자 에너지 빈닝

1.2.1. 구조 정의

EnergyGrid는 전체 에너지 범위(0.1에서 250 MeV)를 256개의 "빈(bin)"으로 나눕니다.

```
struct EnergyGrid {
    const int N_E;                     // □너지 빈 수(256)
    const float E_min;                 // 최소 □너지(0.1 MeV)
    const float E_max;                 // 최대 □너지(250.0 MeV)
    std::vector<float> edges;          // N_E + 1 빈 경계(로그 간격)
    std::vector<float> rep;            // N_E 대표 □너지
};
```

1.2.2. 왜 로그 간격인가요?

⊠성자 치료에서 에너지 분포는 균일하지 않습니다:

- 저에너지 입자가 많음 (사거리 끝부분 근처)
- 고에너지 입자는 적음 (빔 입구 근처)

선형 그리드는 고에너지에서 빈을 낭비하고 저에너지에서 해상도가 낮습니다. 로그 간격은 필요한 곳에 세밀한 해상도를 제공합니다.

1.2.3. 에너지 그리드 레이아웃

[에너지 빈 간격 (로그 스케일):]

빈	범위 (MeV)	ΔE (MeV)	해상도
Bin 0	[0.10, 0.11)	0.01	매우 높음
Bin 1	[0.11, 0.12)	0.01	매우 높음
Bin 50	[1.05, 1.15)	0.10	높음
Bin 200	[140, 155)	15	낮음
Bin 255	[220, 250)	30	매우 낮음

1.2.4. 에너지 빈 찾기

에너지가 $E = 150.0$ MeV인 입자의 빈을 찾습니다:

```
int bin = grid.FindBin(150.0f);  // 이진 탐색: O(log N)
float E_rep = grid.GetRepEnergy(bin);  // 기하 평균
```

"대표 에너지"는 로그 빈의 기하 평균입니다:

$$\text{rep}_i = \sqrt{\text{edges}_i \times \text{edges}_i + 1}$$

1.2.5. 메모리 레이아웃

[**EnergyGrid** 메모리 조직:]

`N_E`	`= 256 (4 bytes)`
`E_min`	`= 0.1 (4 bytes)`
`E_max`	`= 250.0 (4 bytes)`

edges[] 배열 (257 요소):

`edges[0]`	`= 0.100000`
`edges[1]`	`= 0.110281`
`edges[2]`	`= 0.121619`
`...`	`...`
`edges[256]`	`= 250.000000`

rep[] 배열 (256 요소):

`rep[0]`	`= 0.105087`
`rep[1]`	`= 0.115893`
`rep[2]`	`= 0.127848`
`...`	`...`
`rep[255]`	`= 234.520788`

전체 크기: $(3 + 257 + 256) \times 4$ bytes ≈ 4.1 KB

—

1.3. 2. 각도 그리드: 입자 방향 비닝

1.3.1. 구조 정의

AngularGrid는 입자 방향을 빈화합니다. 2D 시뮬레이션(X-Z 평면)에서 입자들은 빔 축(Z 방향)에 대해 $-90°$에서 $+90°$까지의 각도로 이동할 수 있습니다.

```
struct AngularGrid {
    const int N_theta;              // 세타 빈 수(512)
    const float theta_min;          // 최소 각도(-90°)
    const float theta_max;          // 최소 각도(+90°)
    std::vector<float> edges;       // N_theta + 1 빈 경계(균일)
    std::vector<float> rep;         // N_theta 대표 각도
};
```

1.3.2. 왜 균일 간격인가요?

에너지와 달리 양성자 치료에서 각도 분포는:

- ***0°*** 주변에 중심화됨 (전방 지향 빔)
- 피크 근처에서 상대적으로 균일
- 좁음 (대부분의 입자가 ±30° 이내)

각도 퍼짐이 제한되고 더 균일하기 때문에 선형/균일 간격을 사용합니다.

1.3.3. 각도 그리드 레이아웃

[각도 빈 간격 (균일):]

빈 범위 (도)	**Δθ** (도)	물리적 의미

Bin 0	`[-90.0, -89.65)`	`≈ 0.35`	후방 (희귀)

Bin 255	`[-0.175, +0.175]`	`≈ 0.35`	거의 전방!
Bin 511	`[+89.65, +90.0]`	`≈ 0.35`	후방 (희귀)

각 빈 폭: $\Delta\theta = 180° / 512 \approx 0.35°$

1.3.4. 각도 빈 찾기

```
// θ = 5.7°로 이동하는 입자
int bin = grid.FindBin(5.7f);  // 직접 계산: O(1)
float theta_rep = grid.GetRepTheta(bin);  // 산술 평균
```

균일 빈의 경우 대표값은 산술 평균입니다:

$$\text{rep}_i = 0.5 \times (\text{edges}_i + \text{edges}_i + 1)$$

1.3.5. 메모리 사용량

전체 크기: (3 + 513 + 512) × 4 bytes ≈ 8.1 KB

—

1.4. 3. 블록 인코딩: 위상 공간 좌표 압축

1.4.1. 왜 함께 인코딩하는가?

우리는 에너지를 256개 빈으로, 각도를 512개 빈으로 나누었습니다. 이는 $256 \times 512 = 131,072$가지 가능한 조합의 ***2D*** 위상 공간을 만듭니다!

이 두 개를 별도의 숫자로 저장하는 대신, 단일 24비트 정수로 압축합니다. 이를 "블록 ***ID***"라고 부릅니다.

이점:

1. 단일 조회: 두 개 대신 하나의 정수 $\longrightarrow$ 더 빠른 GPU 메모리 접근
2. 캐시 효율성: 블록 ID는 메모리에서 연속적
3. ***GPU*** 친화적: 정수 연산은 GPU에서 빠름
4. 메모리 절약: 24비트 vs 두 개별 인덱스의 32비트

1.4.2. 비트 레이아웃

[블록 **ID** 비트 구조:]

필드	비트 범위	값 범위
`b_E`	`Bits 12-23`	`0-255`
`b_theta`	`Bits 0-11`	`0-511`

[예제 인코딩:]

- `b_theta = 283`, `b_E = 150`
- `block_id = 0x095BB = 614,683`
- 공식: `block_id = (b_E << 12) | b_theta`

1.4.3. 인코딩과 디코딩

```
// 인코딩: (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);
}

// 디코딩: 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;            // 하위 12비트 추출
    b_E = (block_id >> 12) & 0xFFF;        // 상위 12비트 추출
}
```

1.4.4. 빈 블록 마커

```
constexpr uint32_t EMPTY_BLOCK_ID = 0xFFFFFFFF;  // = 4,294,967,295
```

이 특수 값은 사용되지 않은 슬롯을 표시합니다:

- 모든 32비트가 1로 설정됨 (최대 uint32_t 값)
- 유효한 블록 ID가 될 수 없음 (유효한 ID는 24비트만 사용)

1.4.5. 왜 12비트씩인가?

요구 사항	최소 비트	선택된	여유
에너지 빈	8비트 (256값)	12비트	16배 더 많은 용량
각도 빈	9비트 (512값)	12비트	8배 더 많은 용량

—

1.5. 4. 로컬 빈: 세밀한 위치 추적

1.5.1. 4D 세분화

블록이 (에너지, 각도)별로 입자를 그룹화하는 반면, 우리는 여전히 셀 내부의 위치를 추적해야 합니다. 로컬 빈은 정밀한 위치 추적을 위해 각 셀을 *512*개 하위 영역으로 나눕니다.

차원	기호	빈	추적하는 것
로컬 각도	θ_{local}	8	세밀한 각도 변화
로컬 에너지	E_{local}	4	세밀한 에너지 변화
X 위치	x_{sub}	4	횡방향 위치
Z 위치	z_{sub}	4	깊이 위치

$8 \times 4 \times 4 \times 4 = 512$ 블록당 총 로컬 빈

1.5.2. 인코딩 공식

```
// 4D 좌표를 16비트 인덱스로 인코딩
__host__ __device__ inline uint16_t encode_local_idx_4d(
    int theta_local, int E_local, int x_sub, int z_sub
) {
    // 내부 인덱스: E + 4*(x + 4*z)
    int inner = E_local + N_E_local * (x_sub + N_x_sub * z_sub);
    // 외부 인덱스: θ + 8*inner
    return static_cast<uint16_t>(theta_local + N_theta_local * inner);
}
```

1.5.3. 위치 오프셋

각 하위 빈은 셀 중심에 대한 중심 위치를 가집니다:

```
// 하위 빈 중심에서 X 오프셋 (범위: -0.375*dx에서 +0.375*dx)
__host__ __device__ inline float get_x_offset_from_bin(int x_sub, float dx) {
    return dx * (-0.375f + 0.25f * x_sub);
}
```

[하위 빈 중심 오프셋 **(dx = 2.0 mm):**]

x_sub	오프셋 (mm)	셀 내 위치
0	-0.75	좌측 가장자리
1	-0.25	좌측-중앙
2	+0.25	우측-중앙
3	+0.75	우측 가장자리

1.5.4. 메모리 사용량

블록당: 512 × 4 bytes = 2 KB

—

1.6. 5. PsiC: 계층적 위상 공간 저장소

1.6.1. 구조 정의

PsiC는 “Phase-space Cell”의 약자입니다 - 시뮬레이션에서 모든 입자 가중치를 저장하는 마스터 데이터 구조입니다.

```
struct PsiC {
    const int Nx;                        // 그리드 X 차원 (예: 200)
    const int Nz;                        // 그리드 Z 차원 (예: 640)
    const int Kb;                        // 셀당 최대 블록 (32)

    // 저장소 레이아웃: [cell][slot][local_bin]
    std::vector<std::array<uint32_t, 32>> block_id;  // 블록 ID
    std::vector<std::array<std::array<float, LOCAL_BINS>, 32>> value;  // 가중치

private:
    int N_cells;  // Nx × Nz (예: 128,000 셀)
};
```

1.6.2. 저장소 계층

```
레벨 1: 공간 그리드 (Nx × Nz 셀)
├─ 레벨 2: 개별 셀 (i, j)
│  ├─ 레벨 3: 블록 슬롯 (셀당 최대 32개)
│  │  ├─ 슬롯 0: 블록 ID = 0x095BB → 512 로컬 빈
│  │  ├─ 슬롯 1: 블록 ID = 0x08A42 → 512 로컬 빈
│  │  └─ 슬롯 2: 블록 ID = EMPTY     → (사용되지 않음)
│  └─ 각 슬롯은 하나의 (θ, E) 조합□ 대한 입자 가중치 저장
└─ 대부분의 셀은 < 32개 활성 블록 (희소!)
```

1.6.3. 슬롯 할당 알고리즘

```
int PsiC::find_or_allocate_slot(int cell, uint32_t bid) {
    // 1번 패스: 블록이 이미 존재하는지 확인
    for (int slot = 0; slot < Kb; ++slot) {
        if (block_id[cell][slot] == bid) {
            return slot;  // 기존 슬롯 발견 - 재사용!
        }
    }
    // 2번 패스: 빈 슬롯 찾기
    for (int slot = 0; slot < Kb; ++slot) {
        if (block_id[cell][slot] == EMPTY_BLOCK_ID) {
            block_id[cell][slot] = bid;  // 새 슬롯 할당
            return slot;
        }
    }
```

```
    return -1;   // 오류: 사용 가능한 공간 없음!
}
```

1.6.4. 가중치 접근

```
// 특정 로컬 빈□서 가중치 가져오기
float PsiC::get_weight(int cell, int slot, uint16_t lidx) const {
    return value[cell][slot][lidx];
}

// 특정 로컬 빈□ 가중치 설정
void PsiC::set_weight(int cell, int slot, uint16_t lidx, float w) {
    value[cell][slot][lidx] = w;
}
```

1.6.5. 메모리 분석

일반적인 시뮬레이션 그리드의 경우 (200 × 640 셀, 셀당 32개 슬롯):

구성 요소	메모리
`block_id`	`128,000 × 32 × 4 bytes = 16.4 MB`
`value` (완전히 조밀)	`128,000 × 32 × 512 × 4 bytes ≈ 8.6 GB`
`value` (희소, 13%)	`≈ 1.1 GB`

블록-희소 저장소의 장점:

1. 메모리 효율성: 점유된 (θ, E) 영역만 저장
2. *GPU* 친화적: 고정 크기 배열이 병합 메모리 접근 가능
3. 빠른 조회: $O(32)$ 슬롯 검색은 사소함
4. 동적 할당 없음: 모든 메모리가 사전에 할당

—

1.7. 6. OutflowBucket: □ □ □□ □□

1.7.1. 구조 정의

시뮬레이션 시간 단계 동안 입자가 이동하면 현재 셀을 **Exit**하여 인접 셀로 들어갈 수 있습니다. **OutflowBucket**은 인접 셀로 전송되기 전에 이 "유출" 입자들을 일시적으로 보관합니다.

```
struct OutflowBucket {
    static constexpr int Kb_out = 64;   // 최소 슬롯의 2배!

    std::array<uint32_t, Kb_out> block_id;        // 블록 ID
    std::array<uint16_t, Kb_out> local_count;     // 입자 수
    std::array<std::array<float, LOCAL_BINS>, Kb_out> value;  // 가중치
};
```

1.7.2. 4면 경계 시스템

각 셀은 4개 면을 가지며, 각 면은 자체 버킷을 가집니다:

면	방향	인덱스
+z	전방	0
-z	후방	1
+x	우측	2
-x	좌측	3

1.7.3. 전송 흐름

```
시간 단계 Δt: 입자 이동
├─ 새 위치: (x + Δx, z + Δz)
└─ 경계 확인:
    ├─ x + Δx > +dx/2  → +x 버킷으로 방출 (셀 i+1, j가 수신)
    ├─ x + Δx < -dx/2  → -x 버킷으로 방출 (셀 i-1, j가 수신)
    ├─ z + Δz > +dz/2  → +z 버킷으로 방출 (셀 i, j+1가 수신)
    └─ z + Δz < -dz/2  → -z 버킷으로 방출 (셀 i, j-1가 수신)

전송 완료:
├─ 소스 셀: 입자 가중치 제거
└─ 목적지 셀: 입자 가중치 수신
```

1.7.4. 버킷 연산

```
// 기존 블록 슬롯 찾기 또는 새 슬롯 할당
int OutflowBucket::find_or_allocate_slot(uint32_t bid) {
    // PsiC::find_or_allocate_slot과 동일한 알고리즘
}

// 모든 버킷 데이터 정리 (EMPTY로 재설정)
void OutflowBucket::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;
        }
    }
}
```

1.7.5. 메모리 사용량

- 면당: $64 \times (4 + 2 + 512 \times 4)$ bytes ≈ 131.5 KB
- 셀당 (4면): ≈ 526 KB
- 실제 사용: 526 KB (활성 작업 세트, 임시)

—

1.8. 7. 메모리 요약 및 설계 근거

1.8.1. 전체 메모리 분해

데이터 구조	크기	비고
EnergyGrid	~4 KB	인스턴스 1개
AngularGrid	~8 KB	인스턴스 1개
PsiC	~1.1 GB	주 저장소, 희소
OutflowBuckets	~526 KB	활성 셀당
전체 (활성 세트)	~1.12 GB	

1.8.2. 설계 근거

1.9. 왜 블록-희소 저장소인가?

⊠성자 치료에서 위상 공간 점유도는 매우 불균형합니다. 대부분의 (θ, E) 공간은 비어있고 블록-희소 저장소는 점유된 영역에만 할당하여 조밀 저장소 대비 70% 메모리 절약을 달성합니다.

1.10. 왜 24비트 블록 인코딩인가?

단일 정수로 인코딩하면 직접 배열 접근이 가능하여 GPU 성능이 향상됩니다 (정수 연산: 1-2 클럭 vs 해시 계산: 20-50 클럭).

1.11. 왜 512개 로컬 빈인가?

너무 적은 빈은 분산 보존이 나쁘고 너무 많은 빈은 메모리 낭비입니다. 512 빈이 좋은 균형을 제공합니다: 8(θ) × 4(E) × 4(x) × 4(z).

1.12. 왜 셀당 고정 슬롯인가?

고정 크기 배열은 연속 메모리 $\longrightarrow$ GPU 병합, 조각화 없음, 예측 가능한 접근 패턴의 이점을 제공합니다.

1.13. 왜 로그 에너지 빈인가?

양성자 에너지 손실은 $\mathrm{dE/dx} \propto \frac{1}{E}$ (고에너지에서 베테-블로흐 공식)이므로 저에너지에서 세밀한 빈이 필요하고 로그 그리드는 물리와 일치합니다.

1.14. 왜 4면 버킷인가?

2D (X-Z 평면)에서 각 셀은 정확히 4개 이웃을 가지며 각 면은 자체 버킷이 필요합니다. (참고: 3D에서는 6면 필요)

—

1.15. 요약

1.15.1. 데이터 구조 계층

구성 요소	크기	용도
에너지 그리드	256 로그 빈	에너지 $\longrightarrow$ 빈 인덱스
각도 그리드	512 균일 빈	각도 $\longrightarrow$ 빈 인덱스
블록 인코딩	24비트	$(\theta, E) \longrightarrow$ 블록 ID
로컬 빈	512 빈	4D 세분화
PsiC	200×640 셀	주 저장소
유출 버킷	4면 × 64 슬롯	셀 간 전송

1.15.2. 핵심 개념

1. 계층적 조직: 공간 그리드 $\longrightarrow$ 셀 $\longrightarrow$ 블록 $\longrightarrow$ 로컬 빈
2. 블록-희소 저장소: 점유된 (θ, E) 영역에만 할당
3. 효율적 인코딩: 좌표를 정수로 압축
4. ***GPU*** 최적화: 고정 크기 배열, 예측 가능한 레이아웃
5. 물리 기반 설계: 로그 에너지 빈, 4D 로컬 세분화

1.15.3. 성능 특성

연산	복잡도	비고
에너지 빈 찾기	$O(\log N_{\mathrm{E}})$	이진 탐색: 8단계
각도 빈 찾기	$O(1)$	직접 계산
블록 인코딩/디코딩	$O(1)$	비트 연산
셀 슬롯 찾기	$O(K_b)$	선형 탐색: 최대 32단계
가중치 접근	$O(1)$	직접 배열 접근

1.15.4. 메모리 효율성

- 조밀 저장소 필요: 8.6 GB
- 블록-희소 실제 사용: 1.1 GB
- 메모리 절약: 87% (위상 공간 희소성으로 인해)

—

2. ⊠⊠: ⊠⊠ ⊠⊠

2.1. A. EnergyGrid 사용예제

```
// 1단계: 그리드 생성
EnergyGrid grid(0.1f,    // E_min: 최소 양성자 에너지 (MeV)
               250.0f,  // E_max: 최대 양성자 에너지 (MeV)
               256);    // N_E: 에너지 빈 수

// 2단계: 150 MeV 양성자를 포함하는 빈 찾기
int bin = grid.FindBin(150.0f);  // ~200 반환

// 3단계: 이 빈의 대표 에너지 얻기
float E_rep = grid.GetRepEnergy(bin);  // ~147 MeV 반환
```

2.2. B. AngularGrid 사용예제

```
// 1단계: 그리드 생성
AngularGrid grid(-90.0f,  // theta_min: 최소 각도 (도)
                 90.0f,  // theta_max: 최대 각도 (도)
                 512);   // N_theta: 각도 빈 수

// 2단계: θ = 12.3° 입자를 포함하는 빈 찾기
int bin = grid.FindBin(12.3f);  // 빈 283 반환

// 3단계: 대표 각도 얻기
float theta_rep = grid.GetRepTheta(bin);  // ~12.12° 반환
```

2.3. C. 입자저장완전예제

```
// 입자 속성:
// - 위치: (x=100.5mm, z=250.3mm)
// - 각도: θ = 12.3°
// - 에너지: E = 150.7 MeV
// - 가중치: w = 0.00123

// 1단계: 그리드 셀 찾기
int i = static_cast<int>(100.5 / dx);  // 예: i = 50
int j = static_cast<int>(250.3 / dz);  // 예: j = 125
int cell = i + Nx * j;                 // cell = 50 + 200*125 = 25,050

// 2단계: 에너지 및 각도 빈 찾기
int b_E = energy_grid.FindBin(150.7f);    // b_E = 200
int b_theta = angle_grid.FindBin(12.3f);  // b_theta = 283

// 3단계: 블록 ID 인코딩
uint32_t block_id = encode_block(b_theta, b_E);  // 0x095BB

// 4단계: 슬롯 찾기 또는 할당
int slot = psi.find_or_allocate_slot(cell, block_id);  // slot = 5

// 5단계: 로컬 빈 인덱스 계산
uint16_t lidx = encode_local_idx_4d(5, 2, 1, 2);  // lidx = 337

// 6단계: 가중치 저장!
psi.set_weight(cell, slot, lidx, 0.00123f);
```

2.4. D. 버킷전송예제

```
// 셀 (i, j)의 입자가 +x 방향으로 이동

// 1단계: 버킷 슬롯 찾기
```

```
int slot = bucket[2].find_or_allocate_slot(0x095BB);

// 2단계: 버킷 가중치 추가
bucket[2].value[slot][lidx] += 0.00123;

// 3단계: 인접 셀서 수신
for (int slot = 0; slot < Kb_out; ++slot) {
    uint32_t bid = bucket[3].block_id[slot];
    if (bid == EMPTY_BLOCK_ID) continue;

    int local_slot = find_or_allocate_slot(cell, bid);

    for (int lidx = 0; lidx < LOCAL_BINS; ++lidx) {
        float w = bucket[3].value[slot][lidx];
        if (w > 0.0f) {
            value[cell][local_slot][lidx] += w;
        }
    }
}

// 4단계: 전송 후 버킷 정리
bucket[3].clear();
```

—

SM_2D 어데이터구조문서

버전 2.0 - 상세설명강화

질문이나문제는다음소스 드를참조하세요:

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