1. CUDA ⊠널 파이프라인 문서

1.1. ⊠⊠

SM_2D⊠ 결정론적 양성자 수송을 위한 6단계 CUDA 커널 파이프라인을 구현합니다. 파이프라인은 계층적 세분화를 통해 입자를 처리하며, 고에너지 입자용 거친 수송과 중요한 브래그 피크 영역용 정밀 수송을 사용합니다.

1.1.1. ⊠널 시퀀스

시뮬레이션 루프⊠ 각 단계마다 6개 커널을 반복합니다:

⊠널	목적
K1: ActiveMask	정밀 수송이 필요한 셀 감지
K2: CoarseTransport	고에너지 수송(E > 10 MeV)
K3: FineTransport	저에너지 수송(E <= 10 MeV)
K4: BucketTransfer	셀 간 입자 전송
K5: ConservationAudit	보존 법칙 검증
K6: SwapBuffers	입력/출력 포인터 교환

Table 1: CUDA 커널 파이프라인

1.1.2. 파이프라인 흐름

단계	⊠널	설명
입력	입력: 셀의 입자들 $(t = 0)$	
K1	ActiveMask 활성 마스크	작업: 정밀 수송이 필요한 셀 표시 • 각 셀의 입자 에너지 확인 • $E < 10$ MeV이면 “활성”으로 표시 • 출력: ActiveMask 배열
K2	CoarseTransport 거친 수송	대상: 비활성 셀 $(E > 10$ MeV) • 평균 에너지 손실만 사용 • 분산 추적, 샘플링 안 함 • 더 큰 단계 크기 • 빠름 (3-5배 속도 향상)
K3	FineTransport 정밀 수송	대상: 활성 셀 $(E <= 10$ MeV) • 에너지 분산 샘플링 • MCS 산란 샘플링 • 더 작은 단계 크기 • 정확함 (1 percent 미만 오차)
K4	BucketTransfer 버킷 전송	작업: 셀을 떠난 입자 이동 • 4개 이웃 확인 $(\pm x, \pm z)$ • 각 이웃의 유출 버킷에서 입자 읽기 • 올바른 셀의 위상 공간에 추가
K5	ConservationAudit 보존 감사	작업: 입자나 에너지가 손실되지 않았⊠지 확인 • 총 입력 가중치 계산 • 총 출력 가중치 계산 • 검증: 입력 = 출력 + 흡수 • 위반 보고
K6	SwapBuffers 버퍼 교환	작업: 다음 시간 단계 준비 • 입력/출력 포인터 교환 (CPU 측) • 이전 출력이 다음 입력이 됨 • 2.2 GB 데이터 복사 회피
출력	출력: $t = \Delta t$의 입자	다음 반복 준비 완료

Table 2: CUDA 파이프라인 상세 흐름

1.2. K1: ActiveMask ⊠⊠

1.2.1. 파일

`src/cuda/kernels/k1_activemask.cu`

1.2.2. 간단한 설명

K1은 시뮬레이션 그리드의 모든 셀을 스캔하여 저에너지 입자$(E <= 10$ MeV)가 있⊠ 셀을 식별합니다. 마스크(0과 1의 목록)를 생성하여 정밀 수송이 필요한 셀을 표시합니다.

1.2.3. 왜 필요한가

고에너지 입자($E > 10$ MeV)⊠ 브래그 피크에서 멀리 선량 분포에 큰 영향을 미치지 않습니다. 저에너지 입자($E <= 10$ MeV)⊠ 브래그 피크 안이나 근처에 있으며 작은 오차가 큰 선량 오차를 유발합니다. K1은 어떤 셀이 중요한지 식별하여 계산 리소스를 효율적으로 배분합니다.

1.2.4. 작동 방식

단계	설명	결과
입력	모든 셀의 입자 데이터	
스캔	각 스레드가 하나의 셀 처리: • 셀의 모든 입자 에너지 확인 • 최소 에너지 찾기 • 10 MeV 임계값과 비교	병렬 실행
판정	• 최소 $E > 10$ MeV: ActiveMask = 0 • 최소 $E <= 10$ MeV: ActiveMask = 1 • 가중치도 확인 (1e-12 최소)	마스크 생성
출력	• ActiveMask 배열 (0 또⊠ 1) • ActiveList (압축된 활성 셀 목록)	K3용 입력

Table 3: K1 활성 마스크 생성 과정

1.2.5. 스레드 구성

매개변수	값
그리드 크기	`(Nx * Nz + 255) / 256`
블록 크기	256 스레드
셀당 스레드	1 스레드가 1 셀 처리
메모리 접근	block_ids_in, values_in에서 병렬 읽기

Table 4: K1 스레드 구성

1.2.6. 시그니처

```
__global__ void k1_activemask(
    // □력 위상 공간
    const uint32_t* __restrict__ block_ids_in,
    const float* __restrict__ values_in,

    // 그리드 매개변수
    const int Nx, const int Nz,

    // 임계값
    const float b_E_trigger,      // 에너지 임계값 (기본값: 10 MeV)
    const float weight_active_min, // 최소 가중치 (기본값: 1e-12)

    // 출력
    uint8_t* __restrict__ ActiveMask
);
```

1.2.7. 알고리즘

```
__global__ void k1_activemask(...) {
    // 단계 1: 이 스레드가 처리하는 셀 계산
```

```
    int cell = blockIdx.x * blockDim.x + threadIdx.x;
    if (cell >= Nx * Nz) return;  // 경계 확인

    // 단계 2: 누적 변수 초기화
    float total_weight = 0.0f;
    bool has_low_energy = false;

    // 단계 3: 이 셀의 모든 □자 슬롯 스캔
    for (int slot = 0; slot < Kb; ++slot) {
        uint32_t bid = block_ids_in[cell * Kb + slot];
        if (bid == EMPTY_BLOCK_ID) continue;

        uint32_t b_theta, b_E;
        decode_block(bid, b_theta, b_E);
        float E = get_rep_energy(b_E);

        if (E < b_E_trigger) {
            has_low_energy = true;
        }

        for (int i = 0; i < LOCAL_BINS; ++i) {
            total_weight += values_in[flat_index(cell, slot, i)];
        }
    }

    // 단계 4: 출력 쓰기
    ActiveMask[cell] = (has_low_energy && total_weight > weight_active_min) ? 1 : 0;
}
```

1.3. K2: ⊠⊠ ⊠⊠ ⊠⊠

1.3.1. 파일

`src/cuda/kernels/k2_coarsetransport.cu`

1.3.2. 간단한 설명

K2⊠ 근사치를 사용하여 고에너지 입자를 빠르게 수송합니다. 모든 물리학적 세부 사항을 시뮬레이션하⊠ 대신 평균값을 사용하여 3-5배 속도 향상을 달성합니다.

1.3.3. 왜 필요한가

고에너지 입자(브래그 피크에서 멈)⊠ 최종 선량 분포에 큰 영향을 미치지 않습니다. 150 MeV 입자의 위치에서 5% 오차⊠ 대부분의 에너지가 멀리 퇴적되므로 중요하지 않습니다. 근사치를 사용하여 허용 가능한 정확도로 3-5배 속도 향상을 얻을 수 있습니다.

1.3.4. K3와의 주요 차이점

기능	**K2** (거침)	**K3** (정밀)	차이
에너지 분산	아니오(평균만)	예(Vavilov)	3% 정확도 영향
MCS 샘플링	아니오(분산만)	예(무작위 샘플링)	5% 확산 영향
단계 크기	큼	작음	2-3배 속도 향상
정확도	5%	1 percent 미만	임상적 수용 가능

Table 5: K2 vs K3 비교

1.3.5. 간소화된 물리

물리 과정	**K2** 간소화 방법
에너지 손실	표준: Vavilov 분포에서 샘플링 거침: 평균값만 사용 • 결과: 3% 오차, 2배 빠름
다중 쿨롱 산란(MCS)	표준: 무작위 각도 θ N(0, sigma²) 샘플링 거침: sigma² 누적, 아직 샘플링 안 함 • 결과: 5% 확산 오차, 3배 빠름
단계 크기	표준: ds = min(물리한, 경계거리) 거침: ds = 2-3배 더 큼 • 결과: 더 적은 단계, 더 빠른 실행
핵 반응	정밀과 동일 (근사 불가) • w = **exp(-sigma_nuclear** ds)

Table 6: K2 물리적 간소화

1.3.6. 스레드 구성

매개변수	값
그리드 크기	`(Nx * Nz + 255) / 256`
블록 크기	256 스레드
처리	셀당 1 스레드 (활성 셀 건너뜀)
속도 향상	정밀 수송 대비 3-5배

Table 7: K2 스레드 구성

1.3.7. 시그니처

```
__global__ void k2_coarsetransport(
    // 력 위상 공간 (거친 셀만)
    const uint32_t* __restrict__ block_ids_in,
    const float* __restrict__ values_in,
    const uint8_t* __restrict__ ActiveMask,

    // 그리드 및 물리
    const int Nx, const int Nz, const float dx, const float dz,
    const RLUT __restrict__ lut,

    // 출력
    uint32_t* __restrict__ block_ids_out,
    float* __restrict__ values_out,
    double* __restrict__ EdepC,
    float* __restrict__ AbsorbedWeight_cutoff,
    float* __restrict__ AbsorbedWeight_nuclear,
    double* __restrict__ AbsorbedEnergy_nuclear,
    OutflowBucket* __restrict__ OutflowBuckets
);
```

1.3.8. 간소화된 물리 코드

```
__device__ void coarse_transport_step(
    float& E, float& theta, float& x, float& z, float& w,
    float ds, const RLUT& lut
```

```
) {
    // 에너지 손실 （평균만， 분산 없음）
    float E_new = lut.lookup_E_inverse(lut.lookup_R(E) - ds);

    // 다중 쿨롱 산란 （분산 누적， 샘플링 안 함）
    float sigma_theta = highland_sigma(E, ds, X0_water);
    theta_variance += sigma_theta * sigma_theta;

    // 핵 감쇠 （정밀과 동일）
    float sigma_nuc = Sigma_total(E);
    w *= exp(-sigma_nuc * ds);

    // 위치 업데이트
    x += ds * sin(theta);
    z += ds * cos(theta);

    E = E_new;
}
```

1.4. K3: 정밀 수송 ⊠널 (주요 물리)

1.4.1. 파일

`src/cuda/kernels/k3_finetransport.cu`

1.4.2. 간단한 설명

K3⊠ 시뮬레이션의 핵심입니다. 브래그 피크 안이나 근처의 저에너지 입자에 대한 정확한 몬테카를로 ⊠송을 수행합니다. 정확한 선량 분포를 보장하기 위해 ⊠든 물리학적 효과가 무작위 샘플링으로 시뮬레이션됩니다.

1.4.3. 왜 필요한가

브래그 피크⊠ 대부분의 에너지가 퇴적되⊠ 곳입니다. 여기서 작은 오차가 큰 선량 오차를 유발합니다. 10 MeV에서 1mm 위치 오차⊠ 선량을 20% 변경할 수 있습니다. K3⊠ 중요한 곳에서 1 percent 미만 오차를 보장합니다.

1.4.4. 작동 방식

단계	설명
활성 셀 처리	• ActiveList의 셀만 처리 • 각 스레드가 하나의 활성 셀 할당
RNG 초기화	• 각 스레드가 고유한 무작위 상태 가져옴 • 재현 가능한 독립적인 난수 보장
슬롯 처리	• 각 셀에⊠ 여러 입자 "슬롯"이 있음 • 빈 슬롯 건너뜀
로컬 빈 처리	• 각 슬롯에⊠ 512개의 로컬 빈이 있음 • 무시할 수 있⊠ 가중치 건너뜀
위상 공간 디코딩	• 빈 인덱스를 물리적 값으로 변환 • 각도(θ), 에너지(E), 위치(x, z)
빈 내 샘플링	• 빔 내 균일 위치 샘플링 • 분산 보존을 위해 필수
주요 수송 루프	1. 단계 크기 제어 (물리 + 경계) 2. 분산이 포함된 에너지 손실 3. MCS 샘플링 4. 핵 감쇠 5. 위치 업데이트 6. 종료 조건 확인
원자적 누적	• 스레드 안전 출력 • EdepC, AbsorbedWeight, AbsorbedEnergy

Table 8: K3 입자당 수송 과정

1.4.5. 빈 내 샘플링의 중요성

빈 내 샘플링 없이	빈 내 샘플링으로
동일한 빈의 모든 입자가 동일한 위치를 얻음 • 인공적 클러스터링 (나쁜 통계) • 과소평가된 분산	각 입자가 빔 내 무작위 오프셋을 얻음 • 연속 분포 보존 • 올바른 분산

Table 9: 빈 내 샘플링 효과

1.4.6. [illegible]

[illegible]	[illegible]
[illegible]	(n_active + 255) / 256
[illegible]	256 [illegible]
[illegible]	[illegible] 1 [illegible]
RNG [illegible]	[illegible] 1[illegible] ([illegible])
[illegible] [illegible]	[illegible] [illegible] 4 KB

Table 10: K3 [illegible]

1.4.7. [illegible]

```
__global__ void k3_finetransport(
    // [illegible]: [illegible] [illegible]
    const int* __restrict__ ActiveList,
    const int n_active,

    // [illegible] [illegible]
    const uint32_t* __restrict__ block_ids_in,
    const float* __restrict__ values_in,

    // [illegible] [illegible]
    const int Nx, const int Nz, const float dx, const float dz,
    const RLUT __restrict__ lut,
    const curandStateMRG32k3a* __restrict__ rng_states,

    // [illegible]
    uint32_t* __restrict__ block_ids_out,
    float* __restrict__ values_out,
    double* __restrict__ EdepC,
    float* __restrict__ AbsorbedWeight_cutoff,
    float* __restrict__ AbsorbedWeight_nuclear,
    double* __restrict__ AbsorbedEnergy_nuclear,
    OutflowBucket* __restrict__ OutflowBuckets
);
```

1.4.8. [illegible] ([illegible])

```
__global__ void k3_finetransport(...) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= n_active) return;

    int cell = ActiveList[idx];
    curandStateMRG32k3a local_rng = rng_states[idx];

    float cell_Edep = 0.0;
    float cell_E_nuc = 0.0;
    float w_cutoff = 0.0;

    for (int slot = 0; slot < Kb; ++slot) {
        uint32_t bid = block_ids_in[cell * Kb + slot];
        if (bid == EMPTY_BLOCK_ID) continue;

        uint32_t b_theta, b_E;
        decode_block(bid, b_theta, b_E);
```

```
    for (int lidx = 0; lidx < LOCAL_BINS; ++lidx) {
        float w = values_in[flat_index(cell, slot, lidx)];
        if (w < weight_epsilon) continue;

        // 위상 공간 디코딩
        int theta_local, E_local, x_sub, z_sub;
        decode_local_idx_4d(lidx, theta_local, E_local, x_sub, z_sub);

        float theta = get_theta_from_bins(b_theta, theta_local);
        float E = get_energy_from_bins(b_E, E_local);
        float x = cell_x + get_x_offset_from_bin(x_sub, dx);
        float z = cell_z + get_z_offset_from_bin(z_sub, dz);

        // 주요 물리 루프
        while (true) {
            float ds = compute_max_step_physics(E, lut);
            ds = fminf(ds, compute_boundary_step(x, z, dx, dz, theta));

            // 분산이 포함된 에너지 손실
            float dE_straggle = sample_energy_loss_with_straggling(E, ds, &local_rng);
            float E_new = lut.lookup_E_inverse(lut.lookup_R(E) - ds) + dE_straggle;
            E_new = fmaxf(E_new, E_cutoff);

            float dE = E - E_new;
            cell_Edep += w * dE;

            // MCS 샘플링
            float sigma_theta = highland_sigma(E, ds, X0_water);
            float dtheta = sample_mcs_angle(sigma_theta, &local_rng);
            theta += dtheta;

            // 핵 감쇠
            float sigma_nuc = Sigma_total(E);
            float w_nuc = w * (1.0f - exp(-sigma_nuc * ds));
            w -= w_nuc;
            cell_E_nuc += w_nuc * E;

            // 위치 업데이트
            x += ds * sin(theta);
            z += ds * cos(theta);
            E = E_new;

            // 경계 확인
            if (left_cell) {
                emit_to_bucket(OutflowBuckets, cell, face, theta, E, x, z, w);
                break;
            }

            if (E < E_cutoff) {
                w_cutoff += w;
                break;
            }
        }
    }
}

// 원자적 누적
atomicAdd(&EdepC[cell], cell_Edep);
```

```
    atomicAdd(&AbsorbedWeight_cutoff[cell], w_cutoff);
    atomicAdd(&AbsorbedWeight_nuclear[cell], w - w_cutoff);
    atomicAdd(&AbsorbedEnergy_nuclear[cell], cell_E_nuc);

    rng_states[idx] = local_rng;
}
```

1.4.9. 빈 내 샘플링 코드

```
__device__ void sample_intra_bin(
    float& theta, float& E,
    int theta_local, int E_local,
    curandStateMRG32k3a* rng
) {
    // 빈 내 균일 오프셋 샘플링
    float u_theta = curand_uniform(rng) - 0.5f;  // [-0.5, 0.5]
    float u_E = curand_uniform(rng) - 0.5f;

    // 대표 값에 추가
    theta += u_theta * dtheta_bin;
    E *= pow(10.0f, u_E * dlogE_bin);  // 로그 간격
}
```

1.5. K4: ⊠⊠ ⊠⊠ ⊠⊠

1.5.1. 파일

`src/cuda/kernels/k4_transfer.cu`

1.5.2. 간단한 설명

K4⊠ 수송(K2/K3) 중에 셀을 떠난 입자를 새 셀로 이동합니다. 각 셀에⊠ 4개의 “버킷”(각 방향당 하나: ±x, ±z)이 있어 나가⊠ 입자를 포착합니다. K4⊠ 이 버킷을 읽고 올바른 인접 셀에 입자를 넣습니다.

1.5.3. 왜 필요한가

입자⊠ 이동합니다! K2/K3 수송 후 입자⊠ 셀 경계를 넘었을 수 있습니다. 이 모든 입자를 수집하여 새 집에 넣어야 합니다. 이⊠ 다음 반복을 위한 공간적 지역성을 유지합니다.

1.5.4. 작동 방식

단계	설명
입력	각 셀에⊠ 4개의 버킷이 있음: • 버킷 0: +z 방향으로 떠나⊠ 입자 • 버킷 1: -z 방향으로 떠나⊠ 입자 • 버킷 2: +x 방향으로 떠⊠⊠ 입자 • 버킷 3: -x 방향으로 떠나⊠ 입자
이웃 계산	위치 (ix, iz)의 셀의 경우: • +z: cell + Nx (iz+1 < Nz이면) • -z: cell - Nx (iz-1 >= 0이면) • +x: cell + 1 (ix+1 < Nx이면) • -x: cell - 1 (ix-1 >= 0이면)
전송 과정	각 수신 셀에 대해: 1. 4개 이웃의 버킷 확인 2. 각 버킷의 내용 읽기 3. block_id가 이미 있⊠지 확인 • 예: 가중치를 기존 블록에 추가 • 아니오: 새 슬롯 할당 4. 원자적 슬롯 할당 (스레드 안전)
출력	• 모든 입자가 올바른 공간 셀에 있음 • 올바른 위⊠ 공간 빈 (E, θ, x, z) • 다음 반복 준비 완료

Table 11: K4 버킷 전송 과정

1.5.5. 원자적 연산의 중요성

원자적 연산 없이	**atomicCAS**로
경쟁 조건: • 스레드 A 읽기: slot[5] = EMPTY • 스레드 B 읽기: slot[5] = EMPTY • 스레드 A 쓰기: slot[5] = BLOCK_42 • 스레드 B 쓰기: slot[5] = BLOCK_99 • ⟶ 스레드 A의 데이터 손실	스레드 안전: • 스레드 A: atomicCAS ⟶ 성공 • 스레드 B: atomicCAS ⟶ 실패 • 스레드 B: slot[6] 시도 • ⟶ 모든 데이터 보존

Table 12: 원자적 슬롯 할당의 필요성

1.5.6. 스레드구성

매개변수	값
그리드크기	(Nx * Nz + 255) / 256
블록크기	256 스레드
처리	수신셀당 1 스레드
원자적연산	예 (슬롯할당, 가중치추가)
공유메모리	전송버퍼용 1 KB

Table 13: K4 스레드구성

1.5.7. 시그니처

```
__global__ void k4_transfer(
    const OutflowBucket* __restrict__ OutflowBuckets,
    const int Nx, const int Nz,
    uint32_t* __restrict__ block_ids_out,
    float* __restrict__ values_out
);
```

1.5.8. 알고리즘

```
__global__ void k4_transfer(...) {
    int cell = blockIdx.x * blockDim.x + threadIdx.x;
    if (cell >= Nx * Nz) return;

    int ix = cell % Nx;
    int iz = cell / Nx;

    // 4개 이웃에서 수신
    int neighbors[4] = {
        iz + 1 < Nz ? cell + Nx : -1,  // +z
        iz - 1 >= 0 ? cell - Nx : -1,  // -z
        ix + 1 < Nx ? cell + 1 : -1,   // +x
        ix - 1 >= 0 ? cell - 1 : -1    // -x
    };

    for (int face = 0; face < 4; ++face) {
        int src_cell = neighbors[face];
        if (src_cell < 0) continue;

        const OutflowBucket& bucket = OutflowBuckets[src_cell * 4 + face];

        for (int k = 0; k < Kb_out; ++k) {
            uint32_t bid = bucket.block_id[k];
            if (bid == EMPTY_BLOCK_ID) continue;

            int slot = find_or_allocate_slot(block_ids_out, cell, bid);
            if (slot < 0) continue;

            for (int lidx = 0; lidx < LOCAL_BINS; ++lidx) {
                float w = bucket.value[k][lidx];
                if (w > 0) {
                    atomicAdd(&values_out[flat_index(cell, slot, lidx)], w);
                }
            }
```

```
        }
    }
}
```

1.5.9. 원자적 슬롯 할당

```
__device__ int find_or_allocate_slot(
    uint32_t* block_ids,
    int cell,
    uint32_t bid
) {
    // 첫 번째 패스: 존재 확인
    for (int slot = 0; slot < Kb; ++slot) {
        if (block_ids[cell * Kb + slot] == bid) {
            return slot;
        }
    }

    // 두 번째 패스: 빈 슬롯 할당
    for (int slot = 0; slot < Kb; ++slot) {
        uint32_t expected = EMPTY_BLOCK_ID;
        uint32_t* ptr = &block_ids[cell * Kb + slot];
        if (atomicCAS(ptr, expected, bid) == expected) {
            return slot;
        }
    }

    return -1;  // 사용 가능한 공간 없음
}
```

1.6. K5: ⊠⊠ ⊠⊠ ⊠⊠

1.6.1. 파일

`src/cuda/kernels/k5_audit.cu`

1.6.2. 간단한 설명

K5⊠ 시뮬레이션의 회계사입니다. 수송 중 입자나 에너지가 손실되지 않았⊠지 확인합니다. 각 셀에 대해 다음을 검증합니다: “들어온 것 = 나간 것 + 흡수된 것.”

1.6.3. 왜 필요한가

보존 법칙은 기본입니다. 가중치가 보존되지 않으면 시뮬레이션에 버그가 있⊠ 것입니다. K5⊠ 다음을 포착합니다:

- 사라지⊠ 입자 (수송 논리 버그)
- 퇴적되지 않은 에너지 (회계 누출)
- 시간이 지남에 따라 누적되⊠ 수치 오류

1.6.4. 작동 방식

셀	입력 (W_in)	출력+흡수 (W_out+W_cut+W_nuc)	상태
Cell 0	1.0	0.7 + 0.2 + 0.1 = 1.0	통과 (0.0% 오차)
Cell 1	0.8	0.6 + 0.15 + 0.05 = 0.8	통과 (0.0% 오차)
Cell 2	0.5	0.4 + 0.08 + 0.02 = 0.5	통과 (0.0% 오차)

Table 14: K5 보존 검증 예제

1.6.5. [illegible]

[illegible]	[illegible]
[illegible] [illegible]	(Nx * Nz + 255) / 256
[illegible] [illegible]	256 [illegible]
[illegible]	[illegible] 1 [illegible]
[illegible] [illegible]	[illegible] [illegible] ([illegible] [illegible] [illegible])

Table 15: K5 [illegible] [illegible]

1.6.6. [illegible]

```
__global__ void k5_audit(
    const uint32_t* __restrict__ block_ids_in,
    const float* __restrict__ values_in,
    const uint32_t* __restrict__ block_ids_out,
    const float* __restrict__ values_out,
    const float* __restrict__ AbsorbedWeight_cutoff,
    const float* __restrict__ AbsorbedWeight_nuclear,
    const double* __restrict__ AbsorbedEnergy_nuclear,
    const int Nx, const int Nz,
    AuditReport* __restrict__ reports
);
```

1.6.7. [illegible]

```
__global__ void k5_audit(...) {
    int cell = blockIdx.x * blockDim.x + threadIdx.x;
    if (cell >= Nx * Nz) return;

    // [illegible] [illegible] [illegible]
    float W_in = 0.0f;
    for (int slot = 0; slot < Kb; ++slot) {
        if (block_ids_in[cell * Kb + slot] != EMPTY_BLOCK_ID) {
            for (int i = 0; i < LOCAL_BINS; ++i) {
                W_in += values_in[flat_index(cell, slot, i)];
            }
        }
    }

    // [illegible] [illegible] [illegible]
    float W_out = 0.0f;
    for (int slot = 0; slot < Kb; ++slot) {
        if (block_ids_out[cell * Kb + slot] != EMPTY_BLOCK_ID) {
            for (int i = 0; i < LOCAL_BINS; ++i) {
                W_out += values_out[flat_index(cell, slot, i)];
            }
        }
    }

    // [illegible] [illegible]
    float W_cut = AbsorbedWeight_cutoff[cell];
    float W_nuc = AbsorbedWeight_nuclear[cell];

    // [illegible] [illegible]
    float W_expected = W_out + W_cut + W_nuc;
    float W_diff = fabsf(W_in - W_expected);
```

```
    float W_rel = W_diff / fmaxf(W_in, 1e-20f);

    // 보고서 저장
    reports[cell].W_in = W_in;
    reports[cell].W_out = W_out;
    reports[cell].W_error = W_rel;
    reports[cell].pass = (W_rel < 1e-6f);
}
```

1.7. K6: ⊠⊠ ⊠⊠

1.7.1. 파일

`src/cuda/kernels/k6_swap.cu`

1.7.2. 간단한 설명

K6⊠ 입력과 출력 배열을 교환하여 다음 시간 단계를 준비합니다. 이 반복의 출력이 다음 반복의 입력이 됩니다.

1.7.3. 왜 필요한가

K2-K4 후 입자⊠ “out” 배열에 있습니다. 다음 반복을 위해 이것들이 “in” 배열이어야 합니다. 2.2 GB 데이터를 복사하⊠ 대신 배열의 포인터(주소)를 교환합니다.

1.7.4. 작동 방식

교환 전	교환 후
메모리 레이아웃: • block_ids_in ⟶ 배열 A (이전 입력) • block_ids_out ⟶ 배열 B (새 출력) • values_in ⟶ 배열 C • values_out ⟶ 배열 D	메모리 레이아웃: • block_ids_in ⟶ 배열 B (이제 입력) • block_ids_out ⟶ 배열 A (이제 출력) • values_in ⟶ 배열 D • values_out ⟶ 배열 C
데이터 복사 없음!	다음 반복 준비 완료

Table 16: K6 포인터 교환 효과

1.7.5. 포인터 교환의 이점

옵션 1: 데이터 복사	옵션 2: 포인터 교환
• cudaMemcpy(new_in, old_out, 2.2 GB) • 반복당 100 ms 소요 • 메모리 대역폭 낭비	• swap(in_ptr, out_ptr) • 반복당 0.001 microsec 소요 • 메모리 대역폭 0 사용 • 100,000배 더 빠름!

Table 17: K6 성능 비교

1.7.6. 구현

```
// 호스트 측 함수 (커널 실행 없음)
void k6_swap_buffers(
    uint32_t*& block_ids_in,
    uint32_t*& block_ids_out,
    float*& values_in,
    float*& values_out
) {
    std::swap(block_ids_in, block_ids_out);
    std::swap(values_in, values_out);
}
```

1.7.7. 커널이 없는 이유?

포인터 교환은 CPU 연산입니다 - GPU 메모리를 수정할 필요가 없습니다. 이⊠ 반복당 2.2 GB 메모리 복사를 피합니다.

1.8. 메모리 접근 패턴

1.8.1. 병렬 접근 전략

패턴	설명
병렬 접근 (좋음)	메모리가 연속적으로 구성됨: • 스레드 0 읽기: block_ids_in[0], values_in[0:31] • 스레드 1 읽기: block_ids_in[1], values_in[32:63] • 스레드 2 읽기: block_ids_in[2], values_in[64:95] GPU가 하나의 메모리 트랜잭션으로 결합: “block_ids_in[0:255] 및 values_in[0:8191] 읽기” ⟶ 분산 접근보다 20-30배 빠름
분산 접근 (나쁨)	• 스레드 0 읽기: block_ids_in[0] • 스레드 1 읽기: block_ids_in[1000] • 스레드 2 읽기: block_ids_in[2000] GPU가 3개의 별도 메모리 트랜잭션 수행 ⟶ 20-30배 느림

Table 18: 메모리 접근 패턴 비교

1.8.2. 공유 메모리 사용

커널	공유 메모리	목적
K1	256 B	가중치 합산을 위한 부분 감소
K3	4 KB	로컬 빈 누적
K4	1 KB	버킷 전송 버퍼

Table 19: 공유 메모리 사용

1.9. 성능 최적화 요약

기법	커널	이점
활성 셀 처리	K2, K3	빈 셀 건너뜀 (60-90% 절약)
거침/정밀 분할	K2, K3	고에너지용 3-5배 속도 향상
원자적 연산	K4	스레드 안전 슬롯 할당
빈 내 샘플링	K3	분산 보존
포인터 교환	K6	2.2 GB 메모리 복사 회피
병렬 접근	모두	최대 메모리 대역폭

Table 20: 성능 최적화

1.10. 부록: 실행 구성 예제

1.10.1. 그리드 및 블록 설정

```
// 그리드 차원
dim3 grid( (Nx * Nz + 255) / 256 );
dim3 block(256);

// K1: ActiveMask
k1_activemask<<<grid, block>>>(...);

// K3: 정밀 수송 (활성 셀용 더 작은 그리드)
dim3 grid_fine( (n_active + 255) / 256 );
k3_finetransport<<<grid_fine, block>>>(...);

// 동기화
cudaDeviceSynchronize();
```

1.10.2. 쉬운 이해를 위한 비유

개념	비유
GPU vs CPU	**CPU** 방식: 한 명의 숙련된 화가가 펜스를 하나씩 칠함 **GPU** 방식: 10,000명의 견습 화가가 각자 펜스를 동시에 칠함
CUDA 커널	여러 요리사가 동시에 따르는 레시피 • 각 요리사(스레드)는 동일한 레시피(커널 코드) • 다른 재료(데이터)로 작업
스레드 계층	그리드: 모든 셀의 모든 스레드 블록: 협력할 수 있는 스레드 그룹 (32-1024개) 스레드: 실행의 가장 작은 단위
메모리 계층	전역 메모리: 책이 지하실에 보관된 도서관 (느림) 공유 메모리: 토론 그룹이 자주 사용하는 책을 두는 작은 테이블 (빠름)
K1: ActiveMask	품질 관리 검사원이 특별한 취급이 필요한 상자 식별 • 품목 가치 > 1000: 표준 취급 • 품목 가치 <= 1000: 신중한 취급
K2 vs K3	**K2** (거침): 도시의 평균 속도로 여행 시간 계산 **K3** (정밀): 모든 신호등과 정체 상황 시뮬레이션 브래그 피크 근처에서는 K3의 세부 사항이 필요
K4: 전송	우편 서비스 편지 분류 • 할당된 분류함(셀) 확인 • 4개 인접 분류 센터의 편지 확인 • 올바른 슬롯에 정리
K5: 감사	통장 정리 시작 잔액 + 입금 = 종료 잔액 + 출금 숫자가 맞지 않으면 버그가 있음
K6: 교환	상자 라벨 재지정 느린 방식: 상자 A의 모든 문서를 상자 B로 이동 (몇 시간) 빠른 방식: 상자 A와 B의 라벨 교환 (1초)

Table 21: CUDA 개념에 대한 쉬운 비유

1.11. ⊠⊠

SM_2D CUDA 파이프라인은 물질을 통한 양성자 수송을 효율적으로 수행하기 위해 6단계 커널 시퀀스를 사용합니다:

- **K1 (ActiveMask)**: 정확한 시뮬레이션이 필요한 셀 식별
- **K2** (거침): 고에너지 입자용 빠른 수송
- **K3** (정밀): 브래그 피크 영역용 정확한 몬테카를로
- **K4** (전송): 셀 간 입자 이동
- **K5** (감사): 보존 법칙 검증
- **K6** (교환): 다음 반복 준비

각 커널은 병렬 메모리 접근, 최소 스레드 분기, 효율적인 공유 메모리 사용으로 GPU 병렬 실행에 최적화되어 있습니다.

—

SM_2D CUDA □□□□□ □□

버전 2.0 - 체계적으로 정리됨