1. CUDA Kernel Pipeline Documentation

1.1. GPU Programming Fundamentals

1.1.1. What is a GPU?

KEY CONCEPT

1.1.2.

A GPU (Graphics Processing Unit) is a specialized processor designed for parallel computation. Unlike a CPU which has a few powerful cores optimized for sequential tasks, a GPU has thousands of smaller cores optimized for doing many simple operations simultaneously.

In Plain English: Imagine you need to paint 10,000 fence posts.

- CPU approach: One master painter paints each post one at a time (fast per post, but serial)
- GPU approach: 10,000 apprentice painters each paint one post simultaneously (slower per painter, but massively parallel)

For particle transport simulations, we need to process millions of particles. A GPU can process thousands of particles at the same time.

1.1.3. What is a CUDA Kernel?

KEY CONCEPT

1.1.4.

A CUDA kernel is a function that runs on the GPU. When you launch a kernel, it executes on multiple GPU threads in parallel. The same code runs on different data elements - this is called **Single Instruction, Multiple Data** (SIMD).

In Plain English: Think of a kernel as a recipe that multiple chefs follow simultaneously. Each chef (thread) has the same recipe (kernel code) but works on different ingredients (data).

```
// CPU function - runs once
void cpu_function(int* data, int size) {
    for (int i = 0; i \< size; i++) {
        data[i] = data[i] * 2;
    }
}

// GPU kernel - runs on thousands of threads simultaneously
__global__ void gpu_kernel(int* data, int size) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx \< size) {
        data[idx] = data[idx] * 2;
    }
}
```

1.1.5. What are Threads and Blocks?

KEY CONCEPT

1.1.6.

CUDA organizes threads into a two-level hierarchy:

2. Threads

The smallest unit of execution. Each thread executes the kernel code independently.

3. Blocks

Groups of threads (typically 32-1024 threads) that can cooperate through shared memory and synchronization.

4. Grid

A collection of blocks that execute the same kernel.

Component	Description	Example
Grid	All blocks executing kernel	All cells in simulation
Block	Group of threads (32-1024)	Threads 0-255 process Cell 0
Thread	Single execution unit	Thread 0 processes Cell 0, Slot 0

Table 1: CUDA Thread Hierarchy

Each thread represents one worker that processes particles in one cell. All threads in a block work together and can share fast memory.

4.0.1. What is Shared Memory vs Global Memory?

KEY CONCEPT

4.0.2.

5. Global Memory

Large but slow (400-800 clock cycles latency). All threads can access any location. Like main RAM in a computer.

6. Shared Memory

Small but fast (1 clock cycle latency). Only threads within the same block can access it. Like CPU L1 cache.

In Plain English:

- Global Memory: A library where books are stored in the basement (takes time to fetch)
- Shared Memory: A small reading table where your discussion group keeps frequently-used books (instant access)

Memory Type	Characteristics	Latency
Shared Memory	Only threads in same block can access. Used for temp variables, partial sums.	1 cycle
L2 Cache	Shared by all blocks. Intermediate caching layer.	100 cycles
Global Memory	All threads can access any location. Stores all particle data, phase-space arrays (GB scale).	400+ cycles

Table 2: GPU Memory Hierarchy

6.1. Overview

SM_2D implements a 6-stage CUDA kernel pipeline for deterministic proton transport. The pipeline processes particles through hierarchical refinement, with coarse transport for high-energy particles and fine transport for the critical Bragg peak region.

6.2. Pipeline Architecture

6.2.1. Kernel Sequence

The simulation loop iterates through six kernels per step:

Kernel	Purpose
K1: ActiveMask	Detect cells needing fine transport
K2: CoarseTransport	High-energy transport (E > 10 MeV)
K3: FineTransport	Low-energy transport (E ≤ 10 MeV)
K4: BucketTransfer	Inter-cell particle transfer
K5: ConservationAudit	Validate conservation laws
K6: SwapBuffers	Exchange in/out pointers

Table 3: CUDA Kernel Pipeline

6.2.2. Visual Pipeline Flow

Stage	Description
Input	Particles in cells (t = 0)
↓	
K1: Active Mask	**Task**: Scan all cells and mark which ones need fine transport **For each cell**: 1. Check particle energies in cell 2. If any E < 10 MeV ⟶ mark as “active” 3. Create list of active cells for K3 **Output**: ActiveMask[0..N-1] (0 or 1 for each cell), ActiveList (compressed list)
↓	
K2: Coarse Transport	**Task**: Fast approximate transport for INACTIVE cells (E > 10 MeV) **For each INACTIVE cell**: 1. Use mean energy loss only (no straggling) 2. Track variance, don’t sample MCS 3. Larger step sizes 4. Fast (3-5x speedup) **Output**: Transported particles, energy deposition, outflow buckets
K3: Fine Transport	**Task**: Accurate Monte Carlo for ACTIVE cells (E ≤ 10 MeV) **For each ACTIVE cell**: 1. Sample energy straggling (Vavilov) 2. Sample MCS scattering (random angles) 3. Smaller step sizes 4. Accurate (1 percent error) **Output**: Transported particles, energy deposition, outflow buckets
↓	
K4: Bucket Transfer	**Task**: Move particles that left their cell **For each cell**: 1. Check 4 neighbors (±x, ±z) 2. For each neighbor’s outflow bucket: read particles heading to this cell, add them to this cell’s phase space **Output**: All particles now in correct cells
↓	
K5: Conservation Audit	**Task**: Verify no particles or energy were lost **For each cell**: 1. Count total weight IN (from start) 2. Count total weight OUT (left domain) 3. Count total weight ABSORBED (deposited) 4. Verify: IN = OUT + ABSORBED **Output**: Conservation errors (CPU-side)
↓ Swap Buffers	Previous output becomes next input

Table 4: CUDA Pipeline Flow (K1–K6)

6.3. K1: ActiveMask Kernel

6.3.1. File

`src/cuda/kernels/k1_activemask.cu`

6.3.2. What It Does

K1 scans every cell in the simulation grid and answers the question: "Does this cell contain low-energy particles that need accurate simulation?" It creates a mask (a list of 0s and 1s) where 1 means "process this cell carefully" and 0 means "fast processing is OK."

6.3.3. Why It Exists

Not all cells need expensive, accurate simulation. High-energy particles (E > 10 MeV) are far from the Bragg peak and don't affect the dose distribution much. Low-energy particles (E ≤ 10 MeV) are in or near the Bragg peak where small errors cause large dose errors. K1 identifies which cells matter.

6.3.4. How It Works

Stage	Process
Input	**All cells with particles**: • Cell 0: [E=150 MeV] [E=120 MeV] [E=180 MeV] ⟶ All high energy • Cell 1: [E=8 MeV] [E=12 MeV] [E=200 MeV] ⟶ Has low energy • Cell 2: [E=250 MeV] [E=300 MeV] ⟶ All high energy • Cell 3: [E=5 MeV] [E=7 MeV] ⟶ All low energy
Processing	**Each thread checks one cell**: **Thread 0 (Cell 0)**: • Minimum E = 120 MeV • 120 > 10, so NO low energy • Set ActiveMask[0] = 0 **Thread 1 (Cell 1)**: • Minimum E = 8 MeV • 8 ≤ 10, so YES low energy present • Set ActiveMask[1] = 1 **Thread 2 (Cell 2)**: • Minimum E = 250 MeV • 250 > 10, so NO low energy • Set ActiveMask[2] = 0 **Thread 3 (Cell 3)**: • Minimum E = 5 MeV • 5 ≤ 10, so YES low energy present • Set ActiveMask[3] = 1
Output	**ActiveMask array**: [0, 1, 0, 1, 0, 0, 1, ...] **ActiveList**: [1, 3, 6, ...] ⟵ Compressed list of active cells (Cells 1, 3, and 6 are active)

Table 5: K1 Active Mask Generation

6.3.5. Thread Configuration

Parameter	Value
Grid Size	`(Nx * Nz + 255) / 256`
Block Size	256 threads
Threads per Cell	1 thread processes 1 cell
Memory Access	Coalesced reads from block_ids_in, values_in

Table 6: K1 Thread Configuration

6.3.6. Memory Access Pattern

Concept	Description
Global Memory Layout	**Row-major organization**: Cell 0, Cell 1, Cell 2, Cell 3, Cell 4, Cell 5, Cell 6, Cell 7, ... Coalesced access: Adjacent threads read adjacent memory locations
Access Pattern	Thread 0 reads Cell 0 Thread 1 reads Cell 1 Thread 2 reads Cell 2 ... GPU can service all 256 threads in one memory transaction ⟶ 20-30x faster than scattered access

Table 7: K1 Memory Access Pattern (Coalesced)

6.3.7. Signature

```
__global__ void k1_activemask(
    // Input phase-space
    const uint32_t* __restrict__ block_ids_in,
    const float* __restrict__ values_in,

    // Grid parameters
    const int Nx, const int Nz,

    // Thresholds
    const float b_E_trigger,      // Energy threshold (default: 10 MeV)
    const float weight_active_min, // Minimum weight (default: 1e-12)

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

6.3.8. Algorithm

```
__global__ void k1_activemask(...) {
    // STEP 1: Calculate which cell this thread handles
    int cell = blockIdx.x * blockDim.x + threadIdx.x;
    if (cell >= Nx * Nz) return;  // Boundary check

    // STEP 2: Initialize accumulator variables
    float total_weight = 0.0f;
    bool has_low_energy = false;
```

```
    // STEP 3: Scan all particle slots in this cell
    for (int slot = 0; slot \< Kb; ++slot) {
        // Read block ID (energy/angle bin information)
        uint32_t bid = block_ids_in[cell * Kb + slot];
        if (bid == EMPTY_BLOCK_ID) continue;  // Skip empty slots

        // Decode block ID to get energy bin
        uint32_t b_theta, b_E;
        decode_block(bid, b_theta, b_E);

        // Get representative energy for this bin
        float E = get_rep_energy(b_E);

        // STEP 4: Check if low energy present
        if (E \< b_E_trigger) {
            has_low_energy = true;
        }

        // STEP 5: Accumulate particle weight
        for (int i = 0; i \< LOCAL_BINS; ++i) {
            total_weight += values_in[flat_index(cell, slot, i)];
        }
    }

    // STEP 6: Write output
    // Mark active if: (low energy present) AND (sufficient weight)
    ActiveMask[cell] = (has_low_energy && total_weight > weight_active_min) ? 1 : 0;
}
```

6.3.9. In Plain English

Think of K1 as a quality control inspector. Imagine you have a warehouse with thousands of boxes (cells), each containing items (particles) of different values (energies). The inspector needs to identify which boxes contain valuable items that need special handling.

K1 quickly checks each box:

- If any item has value > 1000 ($E > 10$ "MeV"), for standard handling
- Items at 1000 or below: careful handling

This way, expensive "careful handling" (K3 fine transport) is only used where it matters.

6.4. K2: Coarse Transport Kernel

6.4.1. File

`src/cuda/kernels/k2_coarsetransport.cu`

6.4.2. What It Does

K2 transports high-energy particles quickly using approximations. Instead of simulating every physics detail, it uses average values. It's like taking a highway instead of city streets - faster but less detailed.

6.4.3. Why It Exists

High-energy particles (far from the Bragg peak) don't affect the final dose distribution much. A 5% error in a 150 MeV particle's position doesn't matter because it will deposit most of its energy far away. By using approximations, we get 3-5x speedup with acceptable accuracy.

6.4.4. How It Works

Component	Description
Input	**Inactive cells** (ActiveMask = 0): • Cell 0: Particles at E=150 MeV, E=180 MeV • Cell 2: Particles at E=200 MeV, E=250 MeV
Transport Approximations	1. **Energy Loss**: Mean value only • Standard: E_new = E_old - dE - straggling • Coarse: E_new = E_old - dE_mean • Result: 3% error, 2x faster 2. **Multiple Coulomb Scattering**: Variance only • Standard: Sample random angle θ N(0, sigma^2) • Coarse: Accumulate sigma^2, don't sample yet • Result: 5% error in spread, 3x faster 3. **Step Size**: Larger steps • Standard: ds = min(physics_limit, boundary_distance) • Coarse: ds = 2-3x larger • Result: Fewer steps, faster execution 4. **Nuclear Reactions**: Same as fine (can't approximate) • w = **exp(-sigma_nuclear** ds)
Output	• Approximate particle positions and energies • Energy deposition: EdepC array (used for dose calculation) • Outflow buckets: Particles that left cells

Table 8: K2 Coarse Transport

6.4.5. Thread Configuration

Parameter	Value
Grid Size	`(Nx * Nz + 255) / 256`
Block Size	256 threads
Processing	1 thread per cell (skips active cells)
Speedup	3-5x vs fine transport

Table 9: K2 Thread Configuration

6.4.6. Key Differences from K3

Feature	K2 (Coarse)	K3 (Fine)	Difference
Energy straggling	No (mean only)	Yes (Vavilov)	3% accuracy impact
MCS sampling	No (variance only)	Yes (random sampling)	5% spread impact
Step size	Larger	Smaller	2-3x speedup
Accuracy	5%	1 percent	Clinical acceptable

Table 10: K2 vs K3 Comparison

6.4.7. Signature

```
__global__ void k2_coarsetransport(
    // Input phase-space (coarse cells only)
    const uint32_t* __restrict__ block_ids_in,
    const float* __restrict__ values_in,
    const uint8_t* __restrict__ ActiveMask,

    // Grid & physics
    const int Nx, const int Nz, const float dx, const float dz,
    const RLUT __restrict__ lut,

    // Output
    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
);
```

6.4.8. Simplified Physics

```
__device__ void coarse_transport_step(
    float& E, float& theta, float& x, float& z, float& w,
    float ds, const RLUT& lut
) {
    // ENERGY LOSS (mean only, no straggling)
    // Standard: Would sample from Vavilov distribution
    // Coarse:   Just use mean value
    float E_new = lut.lookup_E_inverse(lut.lookup_R(E) - ds);

    // MULTIPLE COULOMB SCATTERING (accumulate variance, don't sample)
    // Standard: theta += sample_gaussian(0, sigma_theta²)
    // Coarse:   theta_variance += sigma_theta²  (save for later)
    float sigma_theta = highland_sigma(E, ds, X0_water);
    theta_variance += sigma_theta * sigma_theta;

    // NUCLEAR ATTENUATION (same as fine - can't approximate)
    float sigma_nuc = Sigma_total(E);
    w *= exp(-sigma_nuc * ds);

    // POSITION UPDATE
    x += ds * sin(theta);
    z += ds * cos(theta);
```

```
    E = E_new;
}
```

6.4.9. In Plain English

Think of K2 as calculating travel time using average speed vs. actual traffic conditions.

Fine transport (K3): "I'll drive through the city and stop at every red light. This will take 23 minutes and 42 seconds."

Coarse transport (K2): "Average speed in the city is 25 mph, so it will take about 24 minutes."

The coarse calculation is:
- Faster (no need to track every detail)
- Close enough (error is small for high-energy particles)
- Scales better (can process thousands of particles quickly)

6.5. K3: Fine Transport Kernel (MAIN PHYSICS)

6.5.1. File

`src/cuda/kernels/k3_finetransport.cu`

6.5.2. What It Does

K3 is the heart of the simulation. It performs accurate Monte Carlo transport for low-energy particles in or near the Bragg peak. Every physics effect is simulated with random sampling to ensure accurate dose distribution.

6.5.3. Why It Exists

The Bragg peak is where most energy is deposited. Small errors here cause large dose errors. For example, a 1 mm position error at 150 MeV doesn't matter much, but at 10 MeV it can change the dose by 20%. K3 ensures 1 percent error where it matters.

6.5.4. How It Works

Stage	Process
Input	**Active cells** (from K1): ActiveList = [1, 3, 6, ...] • Cell 1: Particles at E=8 MeV, E=12 MeV ⟶ Bragg peak region • Cell 3: Particles at E=5 MeV, E=7 MeV ⟶ End of range • Cell 6: Particles at E=9 MeV, E=15 MeV ⟶ Bragg peak region
Thread Assignment	Thread 0 ⟶ processes Cell 1 (ActiveList[0]) Thread 1 ⟶ processes Cell 3 (ActiveList[1]) Thread 2 ⟶ processes Cell 6 (ActiveList[2])
Per-Particle Transport	**For each particle in active cell**: 1. **Decode Phase Space**: • Get position (x, z) within cell • Get energy (E) and angle (theta) • Get weight (w) - probability of this path 2. **Main Physics Loop**: a) Calculate step size ds • Physics limit: dE/dE constraint • Boundary: distance to cell edge • ds = min(physics_limit, boundary) b) Energy loss with straggling • Mean loss: mean_dE = stopping_power times ds • Sample straggling: dE Vavilov(kappa, beta) • E_new = E - mean_dE - straggle_dE ▸ Deposit: Edep += w times (E - E_new) c) Multiple Coulomb Scattering • Calculate sigma: sigma = Highland_formula(E, ds) • Sample angle: dtheta N(0, sigma^2) • Update: theta += dtheta d) Nuclear reactions • Probability: P = 1 - exp(-sigma_nuclear * ds) • Sample: random number in [0, 1] • If absorbed: remove particle, record energy • Else: update weight e) Move particle • x += ds * sin(theta) • z += ds * cos(theta) f) Check boundaries • If left cell: add to outflow bucket, break • If E < cutoff: record absorption, break
Output	• Edep[active_cells] ← **Output**: energy deposited (for dose) • OutflowBuckets[active_cells] ← Particles leaving cells • Absorbed ← Particles that stopped • Record outflow (for K4 transfer)

Table 1.1: K3 Fine Transport Process

6.5.5. Thread Configuration

Parameter	Value
Grid Size	`(n_active + 255) / 256`
Block Size	256 threads
Processing	1 thread per active cell
RNG States	1 per thread (independent random numbers)
Shared Memory	4 KB for local bin accumulation

Table 12: K3 Thread Configuration

6.5.6. Signature

```
__global__ void k3_finetransport(
    // Input: Active cell list
    const int* __restrict__ ActiveList,
    const int n_active,

    // Input phase-space
    const uint32_t* __restrict__ block_ids_in,
    const float* __restrict__ values_in,

    // Grid & physics
    const int Nx, const int Nz, const float dx, const float dz,
    const RLUT __restrict__ lut,
    const curandStateMRG32k3a* __restrict__ rng_states,

    // Output
    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
);
```

6.5.7. Intra-Bin Sampling

For variance preservation, particles are sampled uniformly within bins:

```
__device__ void sample_intra_bin(
    float& theta, float& E,
    int theta_local, int E_local,
    curandStateMRG32k3a* rng
) {
    // Sample uniform offset within bin
    float u_theta = curand_uniform(rng) - 0.5f;  // [-0.5, 0.5]
    float u_E = curand_uniform(rng) - 0.5f;

    // Add to representative values
    theta += u_theta * dtheta_bin;
    E *= pow(10.0f, u_E * dlogE_bin);  // Log spacing for E
}
```

Approach	Effect
WITHOUT intra-bin sampling	• All particles in same bin get SAME position • → Artificial clustering (bad statistics) • → Underestimated variance
WITH intra-bin sampling	• Each particle gets random offset within bin • → Preserves continuous distribution • → Correct variance

Table 13: Intra-Bin Sampling Impact

6.5.8. In Plain English

Think of K3 as a detailed weather simulation vs. K2's simple forecast.

K2 (coarse): "Temperature will be around 20°C today." (Uses averages)

K3 (fine): "Temperature will be 18.7°C at 9:23 AM, 19.2°C at 9:24 AM, ..." (Simulates every fluctuation)

For the Bragg peak, we need K3's detail because:
- Small energy changes → large position changes
- Small position errors → large dose errors
- Clinical accuracy requires 1 percent uncertainty

6.6. K4: Bucket Transfer Kernel

6.6.1. File

`src/cuda/kernels/k4_transfer.cu`

6.6.2. What It Does

K4 moves particles that left their cell during transport (K2/K3) to their new cells. Each cell has 4 "buckets" (one for each direction: ±x, ±z) that catch outgoing particles. K4 reads these buckets and deposits particles in the correct neighboring cells.

6.6.3. Why It Exists

Particles move! After K2/K3 transport, particles may have crossed cell boundaries. We need to collect all these "refugees" and put them in their new homes. This maintains spatial locality for the next iteration.

6.6.4. How It Works

Stage	Process
Input	**Outflow buckets from all cells** (after K2/K3): Each cell has 4 buckets: • Bucket 0: Particles leaving +z direction • Bucket 1: Particles leaving -z direction • Bucket 2: Particles leaving +x direction • Bucket 3: Particles leaving -x direction
Neighbor Discovery	**For Cell 5 at position (ix=5, iz=3)**: **Neighbors**: • +z: Cell 5 + Nx = 5 + 200*1 = 205 (if iz+1 < Nz) • -z: Cell 5 - Nx = 5 - 200 = −195 (out of bounds) • +x: Cell 5 + 1 = 6 (if ix+1 < Nx) • -x: Cell 5 - 1 = 4 (if ix-1 >= 0)
Transfer Process	**For each receiving cell**: 1. Check 4 neighbors' buckets 2. For neighbor Cell 6's bucket 3 (particles going -x): • Read bucket contents • Each entry: (block_id, weight array) • Check if block_id already exists in this cell ▸ If YES: add weights to existing block ▸ If NO: allocate new slot, copy weights 3. Atomic slot allocation (thread-safe) • Multiple threads may try to allocate simultaneously • atomicCAS() ensures only one succeeds
Output	All particles now in: • Correct spatial cell (based on position) • Correct phase-space bin (based on E, theta, x, z) Ready for next iteration!

Table 14: K4 Bucket Transfer Process

6.6.5. Thread Configuration

Parameter	Value
Grid Size	`(Nx * Nz + 255) / 256`
Block Size	256 threads
Processing	1 thread per receiving cell
Atomic Operations	Yes (slot allocation, weight addition)
Shared Memory	1 KB for transfer buffer

Table 15: K4 Thread Configuration

6.6.6. Signature

```
__global__ void k4_transfer(
    // Input: Outflow buckets from all cells
    const OutflowBucket* __restrict__ OutflowBuckets,

    // Grid
    const int Nx, const int Nz,

    // Output phase-space
    uint32_t* __restrict__ block_ids_out,
    float* __restrict__ values_out
);
```

6.6.7. Atomic Slot Allocation

```
__device__ int find_or_allocate_slot(
    uint32_t* block_ids,
    int cell,
    uint32_t bid
) {
    // First pass: check if exists
    for (int slot = 0; slot \< Kb; ++slot) {
        if (block_ids[cell * Kb + slot] == bid) {
            return slot;
        }
    }

    // Second pass: allocate empty slot
    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;  // Successfully allocated
        }
    }

    return -1;  // No space available
}
```

Approach	Behavior
WITHOUT atomic operations (RACE CONDITION)	• Thread A reads: slot[5] = EMPTY • Thread B reads: slot[5] = EMPTY ⟵ Both think it's free! • Thread A writes: slot[5] = BLOCK_42 • Thread B writes: slot[5] = BLOCK_99 ⟵ Overwrites A! • ⟶ Thread A's data lost
WITH atomicCAS (THREAD-SAFE)	• Thread A: atomicCAS(slot[5], EMPTY, BLOCK_42) ⟶ SUCCESS • Thread B: atomicCAS(slot[5], EMPTY, BLOCK_99) ⟶ FAIL (because slot[5] now contains BLOCK_42) • Thread B: tries slot[6], ... • ⟶ All data preserved

Table 16: Atomic Operations for Thread Safety

6.6.8. In Plain English

Think of K4 as a postal service sorting mail. After transport (K2/K3), all the "mail" (particles) is in the wrong sorting bins. K4 is like postal workers who:

1. Check their assigned bin (cell)
2. Look at mail from 4 neighboring sorting centers
3. Find mail addressed to their bin
4. Sort it into the correct slots

The atomic operations are like two workers trying to use the same sorting slot at the same time. Only one can succeed - the other must find a different slot.

6.7. K5: Conservation Audit Kernel

6.7.1. File

`src/cuda/kernels/k5_audit.cu`

6.7.2. Purpose

Verify weight and energy conservation per cell.

6.7.3. What It Does

K5 is the accountant of the simulation. It checks that no particles or energy were lost during transport. For each cell, it verifies: "What came in = What went out + What was absorbed."

6.7.4. Why It Exists

Conservation laws are fundamental. If weight is not conserved, the simulation has a bug. K5 catches:

- Particles disappearing (bug in transport logic)
- Energy not deposited (leak in accounting)
- Numerical errors accumulating over time

6.7.5. How It Works

Stage	Process
Input	**Phase-space before and after transport**: BEFORE (in arrays): • Cell 0: weight_in = 1.0 • Cell 1: weight_in = 0.8 • Cell 2: weight_in = 0.5 AFTER (out arrays): • Cell 0: weight_out = 0.7 • Cell 1: weight_out = 0.6 • Cell 2: weight_out = 0.4 ABSORBED: • Cell 0: cutoff = 0.2, nuclear = 0.1 • Cell 1: cutoff = 0.15, nuclear = 0.05 • Cell 2: cutoff = 0.08, nuclear = 0.02
Conservation Check	**For Cell 0**: • W_in = 1.0 • W_out = 0.7 • W_cut = 0.2 • W_nuc = 0.1 Expected = W_out + W_cut + W_nuc = 0.7 + 0.2 + 0.1 = 1.0 Actual = W_in = 1.0 Difference = |Expected - Actual| = 0.0 Relative = Difference / W_in = 0.0 / 1.0 = 0.0 Pass? YES (0.0 < 1e-6) **For Cell 1**: Expected = 0.6 + 0.15 + 0.05 = 0.8 Actual = 0.8 Difference = 0.0 Pass? YES
Output	reports[0] = {W_in: 1.0, W_out: 0.7, error: 0.0, pass: true} reports[1] = {W_in: 0.8, W_out: 0.6, error: 0.0, pass: true} reports[2] = {W_in: 0.5, W_out: 0.4, error: 0.0, pass: true} … **Summary**: 100% cells passed conservation check

Table 17: K5 Conservation Audit Process

6.7.6. Thread Configuration

Parameter	Value
Grid Size	`(Nx * Nz + 255) / 256`
Block Size	256 threads
Processing	1 thread per cell
Memory Access	Read-only (no atomics needed)

Table 18: K5 Thread Configuration

6.7.7. Signature

```
__global__ void k5_audit(
    // Input phase-space (both in and out)
    const uint32_t* __restrict__ block_ids_in,
    const float* __restrict__ values_in,
    const uint32_t* __restrict__ block_ids_out,
    const float* __restrict__ values_out,

    // Absorption arrays
    const float* __restrict__ AbsorbedWeight_cutoff,
    const float* __restrict__ AbsorbedWeight_nuclear,
    const double* __restrict__ AbsorbedEnergy_nuclear,

    // Grid
    const int Nx, const int Nz,

    // Output report
    AuditReport* __restrict__ reports
);
```

6.7.8. In Plain English

Think of K5 as balancing your checkbook. You want to verify:

Starting balance + Deposits = Ending balance + Withdrawals

If the numbers don't match, something went wrong:

- Maybe you forgot to record a withdrawal (bug in absorption)
- Maybe a deposit was lost (bug in transport)
- Maybe someone stole money (numerical error)

K5 checks this balance for every cell, every iteration. If any cell fails, you know there's a bug to fix.

6.8. K6: Swap Buffers

6.8.1. File

`src/cuda/kernels/k6_swap.cu`

6.8.2. Purpose

Exchange input/output buffers for next iteration (CPU-side pointer swap).

6.8.3. What It Does

K6 prepares the simulation for the next time step by swapping the input and output arrays. The output from this iteration becomes the input for the next iteration.

6.8.4. Why It Exists

After K2-K4, particles are in the "out" arrays. For the next iteration, these should be the "in" arrays. Instead of copying 2.2 GB of data, we just swap the pointers (addresses) of the arrays.

6.8.5. How It Works

Stage	Process
Before Swap	**Memory Layout**: block_ids_in ⟶ [Array A] (old input, now obsolete) block_ids_out ⟶ [Array B] (new output, just computed) values_in ⟶ [Array C] (old input, now obsolete) values_out ⟶ [Array D] (new output, just computed) **For next iteration**: • Array B should become input • Array D should become input • Array A should become output (will be overwritten) • Array C should become output (will be overwritten)
Pointer Swap	**CPU executes**: temp = block_ids_in block_ids_in = block_ids_out ⟵ Now points to Array B block_ids_out = temp ⟵ Now points to Array A temp = values_in values_in = values_out ⟵ Now points to Array D values_out = temp ⟵ Now points to Array C **No data copied!** Just pointer addresses exchanged
After Swap	**Memory Layout** (ready for next iteration): block_ids_in ⟶ [Array B] ⟵ Previous output, now input block_ids_out ⟶ [Array A] ⟵ Previous input, now output values_in ⟶ [Array D] ⟵ Previous output, now input values_out ⟶ [Array C] ⟵ Previous input, now output Next K1-K5 kernels will read from "in" arrays and write to "out" arrays, effectively overwriting the old input data

Table 19: K6 Buffer Swap Process

6.8.6. Implementation

```
// Host-side function (no kernel launch)
void k6_swap_buffers(
    uint32_t*& block_ids_in,
    uint32_t*& block_ids_out,
    float*& values_in,
    float*& values_out
) {
    // Three-way XOR swap (no temporary needed)
    std::swap(block_ids_in, block_ids_out);
    std::swap(values_in, values_out);
}
```

6.8.7. Why No Kernel?

Pointer swap is a CPU operation - GPU memory doesn't need to be modified. This avoids 2.2 GB of memory copy per iteration.

Approach	Performance
WITHOUT POINTER SWAP (SLOW)	Option 1: Copy data cudaMemcpy(new_in, old_out, 2.2 GB, cudaMemcpyDeviceToDevice) → Takes 100 ms per iteration → Wastes memory bandwidth
WITH POINTER SWAP (FAST)	Option 2: Swap pointers swap(in_ptr, out_ptr) → Takes 0.001 microseconds per iteration → Zero memory bandwidth used → 100,000x faster!

Table 20: Pointer Swap Performance

6.8.8. In Plain English

Think of K6 as relabeling boxes instead of moving contents.

Slow way (copying): Take all documents out of Box A, put them in Box B. Takes hours.

Fast way (swapping): Just swap the labels on Box A and Box B. Takes 1 second.

K6 does the fast way. The data stays in memory - we just change what we call “input” and “output.”

6.9. Memory Access Patterns

6.9.1. Coalesced Access Strategy

Concept	Description
Global Memory Layout	**Memory is organized contiguously**: Cell 0: Slot 0, Bins 0-511 ⟶ Thread 0 reads Cell 0: Slot 1, Bins 0-511 … Cell 0: Slot Kb-1, Bins 0-511 Cell 1: Slot 0, Bins 0-511 ⟶ Thread 1 reads Cell 1: Slot 1, Bins 0-511 …
Coalesced Access	Thread 0 reads: block_ids_in[0], values_in[0:31] Thread 1 reads: block_ids_in[1], values_in[32:63] Thread 2 reads: block_ids_in[2], values_in[64:95] … Thread 255 reads: block_ids_in[255], values_in[8160:8191] GPU combines these into ONE memory transaction: “Read block_ids_in[0:255] and values_in[0:8191]” ⟶ 20-30x faster than scattered access
Scattered Access (BAD)	Thread 0 reads: block_ids_in[0] Thread 1 reads: block_ids_in[1000] ⟵ Jump! Thread 2 reads: block_ids_in[2000] ⟵ Jump! GPU must make 3 separate memory transactions ⟶ 20-30x slower

Table 21: Memory Access Patterns

6.9.2. Shared Memory Usage

Kernel	Shared Memory	Purpose
K1	256 B	Partial reduction for weight sum
K3	4 KB	Local bin accumulation
K4	1 KB	Bucket transfer buffer

Table 22: Shared Memory Usage

6.10. Performance Optimization Summary

Technique	Kernel(s)	Benefit
Active cell processing	K2, K3	Skip empty cells (60-90% savings)
Coarse/fine split	K2, K3	3-5x speedup for high-energy
Atomic operations	K4	Thread-safe slot allocation
Intra-bin sampling	K3	Variance preservation
Pointer swap	K6	Avoid 2.2 GB memory copy
Coalesced access	All	Max memory bandwidth

Table 23: Performance Optimizations

6.11. Launch Configuration Example

```
// Grid dimensions
dim3 grid( (Nx * Nz + 255) / 256 );
dim3 block(256);

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

// K3: Fine transport (smaller grid for active cells)
dim3 grid_fine( (n_active + 255) / 256 );
k3_finetransport<<<grid_fine, block>>>(...);

// Synchronization
cudaDeviceSynchronize();
```

6.12. Summary

The SM_2D CUDA pipeline uses a 6-stage kernel sequence to efficiently transport protons through matter:

- **K1 (ActiveMask)**: Identifies cells needing accurate simulation
- **K2 (Coarse)**: Fast transport for high-energy particles
- **K3 (Fine)**: Accurate Monte Carlo for Bragg peak region
- **K4 (Transfer)**: Moves particles between cells
- **K5 (Audit)**: Verifies conservation laws
- **K6 (Swap)**: Prepares for next iteration

Each kernel is optimized for GPU parallel execution with coalesced memory access, minimal thread divergence, and efficient use of shared memory.

—

SM_2D CUDA Pipeline Documentation

Version 2.0 - Enhanced with Tutorials