Profile-driven Optimization: 2D Stencil Kernel

Yeonjin Park (2025-22575)
Graduate School of Data Science
Seoul National University

December 20, 2025

Abstract

This project explores optimization strategies for 2D Stencil operations on GPUs. While initially hypothesizing warp divergence as the bottleneck, profiling with NVIDIA Nsight Compute identified the kernel as memory-bound. To address this, I implemented Shared Memory Tiling and Thread Coarsening. Tiling degraded performance due to suboptimal occupancy and hardware misalignment. Conversely, Thread Coarsening significantly improved performance by leveraging instruction-level parallelism (ILP) and register reuse. The impact of the stencil radius on optimization effectiveness is also analyzed.

1 Introduction

Stencil computations are fundamental in scientific computing and image processing. This report presents a profile-driven optimization of a 2D stencil kernel on GPUs. Initially, I focused on minimizing warp divergence caused by boundary handling. However, profiling revealed the kernel was strictly memory-bound, necessitating a shift towards optimizing memory throughput. I evaluate Shared Memory Tiling and Thread Coarsening, demonstrating that while naive tiling can degrade performance due to occupancy issues, coarsening effectively hides memory latency. Finally, I investigate the sensitivity of these optimizations to the stencil radius.

2 Problem Analysis and Motivation

2.1 Initial Hypothesis: Warp Divergence

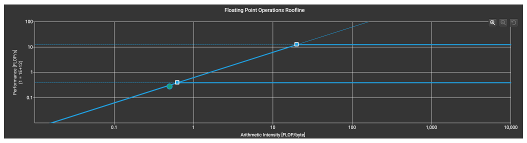
Initially, I hypothesized that boundary checks would cause significant warp divergence. However, profiling the baseline "Strawman" implementation with NVIDIA Nsight Compute (Figure 2) revealed that the kernel achieved 86.09% of the device's peak DRAM throughput. The primary bottleneck was memory dependency stalls, accounting for 77.5% of latency. This confirmed the kernel was memory-bound, shifting the optimization goal to reducing global memory traffic and hiding latency.

2.2 Profiling Reality: Memory Bound

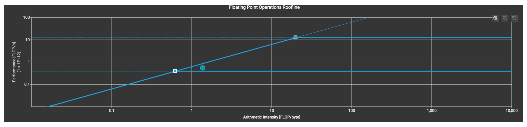
To validate this hypothesis, I profiled the baseline "Strawman" implementation using NVIDIA Nsight Compute. The results contradicted my initial assumption. As illustrated in the Roofline analysis (Figure 1a and 1b), the kernel's performance was not limited by compute capability or instruction issuance but was heavily throttled by memory bandwidth.

Specifically, the baseline kernel achieved a DRAM throughput of **86.09%** of the device's theoretical peak, with an execution time of **0.996 ms**. The profiler highlighted that the root cause of latency was memory dependency stalls. As shown in Figure 1c, each warp spent an average of 21.5 cycles stalled waiting for a scoreboard dependency on L1TEX (L1 Texture/Cache) operations. This accounts for approximately 77.5% of the total latency between instructions, confirming that execution is stalled primarily due to memory fetches.

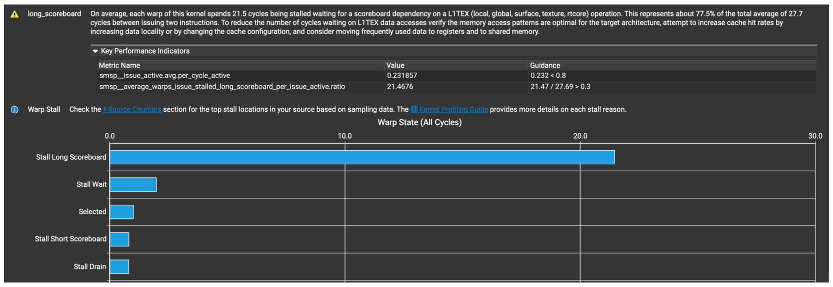
These metrics clearly indicated that the kernel was **memory-bound**. The high number of global memory accesses (5 loads and 1 store per thread for a 5-point stencil) saturated the memory hierarchy. This insight necessitated a shift in optimization strategy: rather than focusing on control flow (warp divergence), the primary goal must be to reduce global memory traffic or hide memory latency. This led to the exploration of **Shared Memory Tiling** to improve data reuse and **Thread Coarsening** to increase arithmetic intensity.



(a) Roofline Model Analysis



(b) Roofline Model Analysis (Radius=2)


long_scoreboard On average, each warp of this kernel spends 21.5 cycles being stalled waiting for a scoreboard dependency on a L1TEX (local, global, surface, texture, rtcore) operation. This represents about 77.5% of the total average of 27.7 cycles between issuing two instructions. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality or by changing the cache configuration, and consider moving frequently used data to registers and to shared memory.

Key Performance Indicators

Metric Name	Value	Guidance
smsp__issue_active.avg.per_cycle_active	0.231857	0.232 < 0.8
smsp__average_warps_issue_stalled_long_scoreboard_per_issue_active.ratio	21.4676	21.47 / 27.69 > 0.3

Warp Stall Check the Source Counters section for the top stall locations in your source based on sampling data. The Kernel Profiling Guide provides more details on each stall reason.

(c) Warp State Statistics

Figure 1: Nsight Compute profiling results for the baseline kernel.

3 Proposed Optimization Strategies

3.1 Baseline Implementation (Strawman)

The baseline kernel, referred to as the "Strawman" implementation, is a naive parallelization of the stencil operation.

- **Concept:** This strategy maps one GPU thread to one output pixel in the 2D grid. The kernel operates directly on data residing in global memory without utilizing shared memory or other on-chip caches explicitly.
- **Implementation:** As shown in Listing 1, the kernel heavily relies on conditional statements (`if`) to handle boundary conditions. For every neighbor access (up, down, left, right), the code checks if the coordinate is within the valid image dimensions.
- **Limitation:** Each thread performs 5 global memory loads and 1 global memory store for a 5-point stencil, resulting in a low arithmetic intensity and heavy memory traffic.

```
__global__ void stencil_baseline(const float* __restrict__ in,
                                 float* __restrict__ out, int width, int height) {
    // Calculate global row and column indices
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // Check if within image bounds
    if (col < width && row < height) {
        float result = in[row * width + col];

        // Access neighbors with boundary checks (At most 5 loads)
        if (row + 1 < height)   result += in[(row + 1) * width + col];
        if (row - 1 >= 0)       result += in[(row - 1) * width + col];
        if (col + 1 < width)    result += in[row * width + (col + 1)];
        if (col - 1 >= 0)       result += in[row * width + (col - 1)];

        // Write result to global memory (1 store)
        out[row * width + col] = result;
    }
}
```

Listing 1: Naive 2D Stencil Kernel (Strawman)

3.2 Approach 1: Shared Memory Tiling

To reduce global memory traffic, I implemented tiling using shared memory.

- **Concept:** The core idea is to load a block of data, including the necessary "halo" regions (data required by boundary threads of the block but residing outside the block's logical domain), into fast on-chip shared memory.

- **Implementation:** As illustrated in Listing 2, threads cooperatively load the required data into shared memory. A `__syncthreads()` barrier ensures all data is loaded before computation proceeds.

- **Expected Benefit:** The most significant advantage of this strategy is the reduction of redundant global memory accesses. In the naive implementation, neighboring threads redundantly load the same data from global memory. Tiling ensures that each data element is loaded from global memory only once per tile and stored in shared memory. Subsequent accesses during the stencil computation are served from the much faster shared memory. While this design also eliminates divergence during the computation phase (as boundary checks are handled during loading), the bandwidth saving from data reuse is the primary performance driver.

```
__global__ void stencil_tiled(const float* __restrict__ in,
                              float* __restrict__ out, int width, int height) {
  // Allocate shared memory with halo space
  __shared__ float tile[TILE_WIDTH + 2 * RADIUS][TILE_WIDTH + 2 * RADIUS];

  // Calculate global row and column indices for loading
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int col_o = blockIdx.x * TILE_WIDTH + tx;
  int row_o = blockIdx.y * TILE_WIDTH + ty;
  int col_i = col_o - RADIUS;
  int row_i = row_o - RADIUS;

  // Load data into shared memory with halo regions
  if (row_i >= 0 && row_i < height && col_i >= 0 && col_i < width) {
    tile[ty][tx] = in[row_i * width + col_i];
  } else {
    tile[ty][tx] = 0.0f; // Handle out-of-bounds with padding
  }
  __syncthreads(); // Barrier: Wait for all loads to complete

  // Perform stencil operation using shared memory
  if (tx < TILE_WIDTH && ty < TILE_WIDTH &&
      row_o >= 0 && col_o >= 0 && row_o < height && col_o < width) {

    // Unrolled for RADIUS 1 case
    float result = 0.0f;
    result += tile[ty - 1 + RADIUS][tx + RADIUS]; // Top
    result += tile[ty + 1 + RADIUS][tx + RADIUS]; // Bottom
    result += tile[ty + RADIUS][tx - 1 + RADIUS]; // Left
    result += tile[ty + RADIUS][tx + 1 + RADIUS]; // Right
    result += tile[ty + RADIUS][tx + RADIUS];     // Center

    out[row_o * width + col_o] = result;
  }
}
```

Listing 2: Shared Memory Tiling Kernel

3.3 Approach 2: Thread Coarsening

Thread coarsening merges the work of multiple threads into one.

- **Concept:** Instead of one thread computing one pixel, each thread is assigned to compute multiple consecutive pixels. In this project, we process 4 pixels per thread along the column direction.

- **Implementation:** In Listing 3, each thread computes 4 output pixels consecutively along the row direction. The loop for pixel processing is manually unrolled.

- **Expected Benefit:**
  - **Instruction-Level Parallelism (ILP):** The compiler can interleave independent instructions. While waiting for a memory load for one pixel (latency), the GPU can issue arithmetic instructions for another pixel (hiding latency).
  - **Redundant Work Reduction:** Common index calculations are performed once and reused. Registers act as a sliding window to reduce redundant loads.

  - **Reduced Synchronization:** No shared memory barriers are needed.

```
__global__ void stencil_coarsen4(const float* __restrict__ in,
                                  float* __restrict__ out,
                                  const int width, const int height) {
  // Global row and column indices
  const int col = blockIdx.x * blockDim.x + threadIdx.x;
  // Coarsening by factor of 4 in row direction
  const int row_base = (blockIdx.y * blockDim.y + threadIdx.y) * 4;
  const bool has_left = (col - 1) >= 0;
  const bool has_right = (col + 1) < width;

  // Process 4 pixels in the row direction
  int row = row_base;
  float result = 0.0f;
  float bottom = 0.0f, middle = 0.0f, top = 0.0f;

  if (col < width) {
    // --- Row 0 ---
    if (row - 1 >= 0 && row - 1 < height) {
      top = in[(row - 1) * width + col];
    }
    if (row >= 0 && row < height) middle = in[row * width + col];
    if (row + 1 >= 0 && row + 1 < height) bottom = in[(row + 1) * width + col];

    if (row < height) {
      result = top + middle + bottom;
      if (has_left) result += in[row * width + (col - 1)];
      if (has_right) result += in[row * width + (col + 1)];
      out[row * width + col] = result;
    }

    // ... (Repeat for Row 1, 2, 3 with sliding window logic) ...
  }
}
```

Listing 3: Thread Coarsened Kernel (Factor=4, No Shared Memory)

3.4 Approach 3: Hybrid

Finally, I combined the two strategies.

- **Concept:** A hybrid approach that uses **Shared Memory Tiling** for data reuse and **Thread Coarsening** for ILP.
- **Implementation:** Listing 4 shows the implementation. The shared memory size is increased to accommodate the coarsened tile.
- **Benefits and Trade-offs:**
  - **Synergy (Latency Hiding + Data Reuse):** By combining both techniques, we can fetch data from Shared Memory (avoiding Global Memory latency) while simultaneously processing multiple elements to hide the remaining arithmetic or Shared Memory latency. This theoretically provides the highest throughput potential.
  - **Occupancy Mitigation:** As analyzed in the evaluation, pure tiling suffered from severe occupancy issues due to irregular block sizes. Coarsening increases the work per thread, which can sometimes allow for better block dimensions that align more closely with hardware constraints, partially mitigating the occupancy loss seen in pure tiling. However, the complexity of managing both shared memory and registers can introduce new overheads.

```
__global__ void stencil_tiled_coarsen4(const float* __restrict__ in,
                                        float* __restrict__ out,
                                        int width, int height) {
  // Allocated shared memory
  __shared__ float tile[TILE_WIDTH + 2 * RADIUS][TILE_WIDTH + 2 * RADIUS];

  // Calculate shared memory indexes
  const int tile_origin_col = blockIdx.x * TILE_WIDTH;
  const int tile_origin_row = blockIdx.y * TILE_WIDTH;
  const int tx = threadIdx.x;
  const int ty = threadIdx.y;

  // Load input (Unrolled for Coarsening Factor)
  {
    // Calculate input coordinates with halo offset
    const int input_col = tile_origin_col + tx - RADIUS;
    const int shared_row_base = ty * COARSENING_FACTOR;
    const int input_row_base = tile_origin_row + shared_row_base - RADIUS;

    // Unrolled load for 4 rows
    // --- Row 0 ---
    if (shared_row_base < TILE_WIDTH + 2 * RADIUS) {
      float value = 0.0f;
      int input_row = input_row_base;
      if (input_col >= 0 && input_col < width && input_row >= 0 && input_row < height)
     {
        value = in[input_row * width + input_col];
      }
      tile[shared_row_base][tx] = value;
    }
    // ... (Repeat for Row 1, 2, 3) ...
    // (omitted for brevity, same pattern as Row 0)
  }

  __syncthreads();

  // Compute output (Unrolled)
  {
    const int output_col = tile_origin_col + tx;
    const int shared_col = tx + RADIUS;

    // Check validity: calculation threads only
    if (tx >= TILE_WIDTH || ty >= TILE_WIDTH / COARSENING_FACTOR || output_col >=
    width) {
      return;
    }

    const int output_row_start = tile_origin_row + ty * COARSENING_FACTOR;
    const int shared_row_start = ty * COARSENING_FACTOR + RADIUS;

    // Register sliding window
    float bottom = tile[shared_row_start + 1][shared_col];
    float middle = tile[shared_row_start][shared_col];
    float top = tile[shared_row_start - 1][shared_col];

    int out_index = output_row_start * width + output_col;

    // --- Row 0 (Divergence-free Compute) ---
    {
      const int output_row = output_row_start + 0;
      const int shared_row = shared_row_start + 0;

      if (output_row < height) {
        float result = bottom + middle + top;
        result += tile[shared_row][shared_col + 1]; // Right
        result += tile[shared_row][shared_col - 1]; // Left
        out[out_index] = result;
      }
      out_index += width;
    }
    // ... (Repeat for Row 1, 2, 3 with sliding window logic) ...
  }
}
```

Listing 4: Hybrid Kernel: Tiling + Coarsening (Unrolled)

4 Evaluation

4.1 Experimental Setup

Experiments were conducted on an NVIDIA TITAN RTX GPU. The input dataset is a large 2D single-precision float array of size 8192×8192. We measured the kernel execution time (averaged over multiple runs) and derived the effective memory throughput and computational throughput.

To ensure a fair comparison, we aimed to keep the thread block size as close to 256 threads as possible for all strategies.

- **Strawman (Baseline):** We used a standard 16×16 thread block (256 threads), which is a multiple of the warp size (32).
- **Tiling:** To produce a 16×16 output tile while accounting for a radius $R = 1$, we expanded the thread block to 18×18 (324 threads). This expansion is necessary to load the halo region into shared memory.
- **Coarsening:** We used a 32×8 thread block (256 threads). Each thread processes 4 pixels vertically, effectively covering a 32×32 data tile.
- **Hybrid:** To combine tiling with coarsening (factor 4), we needed to load a larger tile. For a 32×32 output tile, the required input tile size (including halo) is 34×34. We used a thread block size of 34×9 (306 threads). Each thread loads roughly 4 elements and computes 4 elements.

4.2 Overall Performance Analysis (Radius $R = 1$)

Table 1 summarizes the performance metrics for each implementation.

Method	Block Size	Thread Count	Time (ms)	Memory	Speedup
Baseline	16×16	256	0.996	86.09%	1.00x
Tiling	18×18	324	1.770	48.10%	0.56x
Coarsening (x4)	32×8	256	0.958	90.87%	1.04x
Hybrid	34×9	306	0.948	90.34%	1.05x

Table 1: Performance comparison of stencil kernels at Radius=1.

4.2.1 Why Did Tiling Degrade Performance?

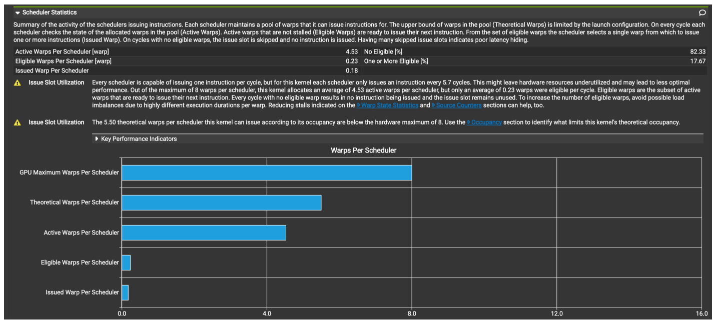
Contrary to textbook examples, Shared Memory Tiling performed significantly worse (1.79ms vs 0.996ms). The main reasons are twofold:

- **Occupancy and Hardware Alignment:** As discussed in the setup, the 18×18 thread block configuration leads to significant warp quantization loss and low theoretical occupancy.
  - **Warp Quantization Loss:** The thread block requires $\lceil 324/32 \rceil = 11$ warps. The 11th warp contains only 4 active threads ($324 - 10 \times 32$), leaving 28 threads idle. This wastes hardware execution slots within the warp.
  - **Reduced Occupancy:** The SM has a hard limit on the maximum number of warps it can schedule. The irregular block size of 324 threads (11 warps) causes fragmentation. For instance, if an SM can hold 32 warps, it can only accommodate $\lfloor 32/11 \rfloor = 2$ blocks. This utilizes only $2 \times 11 = 22$ warps out of the available 32 capacity, dropping the theoretical occupancy to $22/32 = 68.75\%$.
- **Uncoalesced Global Memory Access:** The memory access pattern for loading the tile (including halo) into shared memory disrupted coalescing. The adjusted block dimensions meant that threads were no longer accessing contiguous 128-byte segments in alignment with the DRAM bus, leading to wasted memory bandwidth.

4.2.2 Why Did Coarsening Improve Performance?

Thread Coarsening (Factor 4) achieved the best performance (0.958ms) with a memory throughput of **90.87%**.

- **Hiding Latency via ILP:** By assigning 4 pixels to one thread, we created a stream of independent memory instructions. While the GPU waits for the first pixel's data (DRAM access latency), it can issue instructions for the second, third, and fourth pixels. This Instruction-Level Parallelism effectively hides the memory latency.
- **Saturation:** The 90.87% throughput indicates that this kernel successfully saturated the device's available memory bandwidth, which is the theoretical limit for a memory-bound kernel.


Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	4.53	No Eligible [%]	82.33
Eligible Warps Per Scheduler [warp]	0.23	One or More Eligible [%]	17.67
Issued Warp Per Scheduler	0.18		

Issue Slot Utilization: Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 5.7 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 8 warps per scheduler, this kernel allocates an average of 4.53 active warps per scheduler, but only an average of 0.23 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, avoid possible load imbalances due to highly different execution durations per warp. Reducing stalls indicated on the Warp State Statistics and Source Counters sections can help, too.

Issue Slot Utilization: The 5.50 theoretical warps per scheduler this kernel can issue according to its occupancy are below the hardware maximum of 8. Use the Occupancy section to identify what limits this kernel's theoretical occupancy.

Key Performance Indicators
Warps Per Scheduler
GPU Maximum Warps Per Scheduler
Theoretical Warps Per Scheduler
Active Warps Per Scheduler
Eligible Warps Per Scheduler
Issued Warp Per Scheduler
0.0
4.0
8.0
12.0
16.0

(a) Limited theoretical occupancy of tiling approach

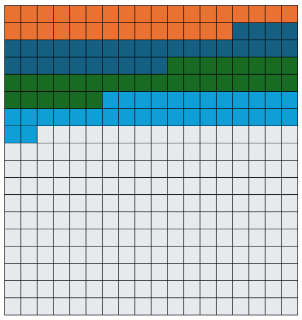

(b) Warps in 18 × 18 block

Figure 2: Nsight Compute profiling results for the baseline kernel.

4.2.3 Hybrid Approach (Tiling + Coarsening) and Occupancy

The hybrid approach yielded a similar performance (0.950ms) to pure coarsening.

- **Why not significantly faster?** While we combined the benefits of tiling (data reuse) and coarsening (ILP), the hybrid kernel still suffers from the same structural issue as the pure tiling approach: **Occupancy**.
- **Occupancy Analysis:** To support a 32×32 output tile with halo ($R = 1$) and coarsening (factor 4), we used a thread block size of $34 \times 9 = 306$ threads.
  - **Warp Quantization:** $\lceil 306/32 \rceil = 10$ warps. The 10th warp has only $306 - 9 \times 32 = 18$ active threads (14 idle).
  - **Block Fragmentation:** An SM with a 32-warp capacity can only fit $\lfloor 32/10 \rfloor = 3$ blocks. This results in $3 \times 10 = 30$ active warps out of 32, which gives an occupancy of $30/32 = 93.75\%$.
  - While this is better than the pure tiling case (68.75%), it is still suboptimal compared to the baseline or pure coarsening (100%). The overhead of shared memory synchronization and the complexity of the kernel likely offset the gains from improved occupancy compared to the 18×18 tiling case.

4.3 Effect of Stencil Radius

I extended the experiment by increasing the stencil radius to $R = 2$. Table 2 summarizes the performance at $R = 2$.

As shown in Table 2, the performance gap between the baseline and the optimized kernels (Coarsening and Hybrid) widens significantly at $R = 2$. The baseline implementation achieves only 61.68% memory throughput, indicating that it struggles to saturate the memory bandwidth efficiently as the data access complexity increases. In contrast, both Coarsening and Hybrid approaches maintain high memory throughput (over 86%), demonstrating their superior ability to hide latency and utilize available bandwidth. This suggests that as the problem complexity (radius) increases, optimization techniques that maximize ILP and data reuse become increasingly critical for overcoming the memory wall.

Method	Block Size	Thread Count	Time (ms)	Memory	Speedup
Strawman	16 × 16	256	1.39	61.68%	1.00x
Tiling	20 × 20	400	2.06	41.19%	0.67x
Coarsening (x4)	32 × 8	256	0.954	90.01%	1.45x
Hybrid	36 × 9	324	0.994	86.21%	1.40x

Table 2: Performance comparison of stencil kernels at Radius=2.

References

[MKYW20] Jackson Marusarz, Max Katz, Charlene Yang, and Samuel Williams. Accelerating HPC applications with NVIDIA Nsight Compute roofline analysis,

2020. Accessed: 2025-12-20. URL: https://developer.nvidia.com/blog/accelerating-hpc-applications-with-nsight-compute-roofline-analysis/.

[NVI12] NVIDIA Corporation. GPU performance analysis. GPU Technology Conference (GTC) 2012 Presentation, Session S0514, 2012. Accessed: 2025-12-19. URL: https://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0514-GTC2012-GPU-Performance-Analysis.pdf.

[NVI25] NVIDIA Corporation. *Nsight Compute Profiling Guide*. NVIDIA, 2025. Accessed: 2025-12-20. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html.