CS610 : Programming For Performance

Assignment 4

Submitted By : Kushagra Srivastava, 220573

Problem 1

Consider the following 7-point 3D stencil code.

```
#define N 64
double in[N][N][N], out[N][N][N];
for (int i=1; i<N-1; i++) {
  for (int j=1; j<N-1; j++) {
    for (int k=1; k<N-1; k++) {
      out[i][j][k] = 0.8 * ( in[i-1][j][k] + in[i+1][j][k]
                           + in[i][j-1][k] + in[i][j+1][k]
                           + in[i][j][k-1] + in[i][j][k+1] );
    }
  }
}
```

In the stencil pattern, the value of a point is a function of the six neighboring points (two each in the i, j, and k planes).

Implement four versions of the above stencil pattern. Your goal is to obtain as much speedup as possible, especially with the second version. Initialize the elements of `in` with random values.

(i) Implement a naïve CUDA kernel (no optimizations required) that directly implements the above code.

(ii) Use shared-memory tiling to improve memory access performance. Investigate and describe how the size of the block/tile computed by each thread block influences performance. Experiment with blocks whose side lengths come from the set $\{1, 2, 4, 8\}$.

(iii) Apply additional loop/code transformations (e.g., loop unrolling and loop permutation) on top of version (ii). Explain your optimizations and highlight their impact.

(iv) Re-implement version (iii) using pinned (page-locked) host memory (e.g., `cudaHostAlloc(. . . )`, `cudaHostAllocDefault`).

Compare the performance of the different kernel versions and explain your observations. Use `cudaEvent` APIs to time CUDA kernels. Use Nsight Systems or Nsight Compute to profile and analyze results.

You may implement the aforementioned versions in different files.

Solution

CPU Kernel

Metrics	N = 64	N = 256	N = 512
Total Time	503 μs	34347 ms	331799 ms

Table 1: Averaged over 5 Runs

Naïve CUDA kernel

- **Thread Mapping**: Each CUDA thread computes its 3D coordinates (i, j, k) using `blockIdx` and `threadIdx` across x, y, and z dimensions
- **Boundary Checking**: Threads check if their coordinates are within valid interior bounds $[1, n-1)$ to avoid computing halo/boundary regions
- **Global Memory Access**: Each thread reads 6 neighboring values directly from global device memory with the pattern `in`$[(i \pm 1) \times n^2 + (j \pm 1) \times n + (k \pm 1)]$

- **Stencil Computation**: Applies the 6-point weighted average formula $0.8 \times \sum_{\text{6 neighbors}}$ and writes the result to global memory at $\texttt{out}[i \times n^2 + j \times n + k]$

Metrics	N = 64	N = 256	N = 512
H2D Data Transfer Time	0.64 ms	29.34 ms	241.52 ms
Kernel Execution Time	0.15 ms	1.89 ms	15.64 ms
D2H Transfer Time	0.68 ms	31.08 ms	254.85 ms
Total Time	1.53 ms	62.32 ms	512.81 ms
Kernel speedup vs CPU	3.3×	18.10×	21.65×
Overall speedup vs CPU	0.34×	0.55×	0.68×

Table 2: Averaged over 5 Runs

Shared Memory Tiling

- **Shared Memory Allocation**: Each thread block allocates a 3D shared memory tile of size $10 \times 10 \times 10$ to accommodate blocks up to $8 \times 8 \times 8$ plus halo regions (1 element padding on each side for the 6-point stencil)
- **Cooperative Data Loading**: Threads collaboratively load their corresponding input elements into shared memory at position $\texttt{tile}[lx+1][ly+1][lz+1]$, while boundary threads additionally load halo data from neighboring regions in global memory
- **Synchronization**: `__syncthreads()` ensures all threads complete loading the tile and halos before computation begins, preventing race conditions
- **Stencil Computation from Shared Memory**: Each thread reads the 6 neighbors from fast shared memory using local indices ($\texttt{tile}[lx \pm 1][ly+1][lz+1]$, etc.) instead of slow global memory, reducing memory latency and bandwidth usage

Metrics	N = 64	N = 256	N = 512
H2D Data Transfer Time	0.639 ms	32.65 ms	243.60 ms
Kernel Execution Time	0.441 ms	27.67 ms	187.06 ms
D2H Transfer Time	0.678 ms	16.23 ms	246.80 ms
Total Time	1.758 ms	76.55 ms	677.46 ms
Kernel speedup vs CPU	1.35×	1.29×	1.85×
Overall speedup vs CPU	0.34×	0.46×	0.51×

Table 3: Averaged over 5 Runs for Block Size = 1×1×1

Metrics	N = 64	N = 256	N = 512
H2D Data Transfer Time	0.630 ms	32.16 ms	243.03 ms
Kernel Execution Time	0.052 ms	6.08 ms	49.39 ms
D2H Transfer Time	0.628 ms	16.30 ms	306.85 ms
Total Time	1.311 ms	54.53 ms	599.27 ms
Kernel speedup vs CPU	11.39×	5.86×	7.01×
Overall speedup vs CPU	0.45×	0.65×	0.58×

Table 4: Averaged over 5 Runs for Block Size = 2×2×2

Metrics	N = 64	N = 256	N = 512
H2D Data Transfer Time	0.605 ms	38.14 ms	239.33 ms
Kernel Execution Time	0.023 ms	2.40 ms	20.99 ms
D2H Transfer Time	0.426 ms	16.40 ms	319.93 ms
Total Time	1.054 ms	56.93 ms	580.25 ms
Kernel speedup vs CPU	25.68×	14.83×	16.50×
Overall speedup vs CPU	0.56×	0.63×	0.60×

Table 5: Averaged over 5 Runs for Block Size = 4×4×4

Metrics	N = 64	N = 256	N = 512
H2D Data Transfer Time	0.612 ms	32.42 ms	242.75 ms
Kernel Execution Time	0.025 ms	1.45 ms	12.45 ms
D2H Transfer Time	0.419 ms	16.45 ms	323.49 ms
Total Time	1.056 ms	50.32 ms	578.68 ms
Kernel speedup vs CPU	23.75×	24.61×	27.83×
Overall speedup vs CPU	0.56×	0.71×	0.60×

Table 6: Averaged over 5 Runs for Block Size = 8×8×8

Loop Unrolling

- **Index Precomputation**: Precomputes the base index `idx_base` = $gx \times n^2 + gy \times n$ once per thread, reducing redundant arithmetic operations in memory accesses from $O(6)$ to $O(1)$ per stencil computation
- **Unrolled Halo Loading**: Uses `if-else` chains instead of independent `if` statements for halo loading, enabling better instruction-level parallelism and reducing branch divergence by eliminating redundant boundary checks
- **Register-Based Accumulation**: Accumulates the 6 neighbor values into a register variable `sum` before applying the scaling factor, reducing shared memory access operations and improving computational throughput
- **Optimized Write Pattern**: Writes output using the precomputed base index as `out`[idx_base + gz], improving memory coalescing by keeping the fastest-changing dimension (z) as the contiguous access pattern

Metrics	N = 64	N = 256	N = 512
H2D Data Transfer Time	0.59 ms	28.92 ms	243.59 ms
Kernel Execution Time	0.12 ms	1.54 ms	12.17 ms
D2H Transfer Time	0.62 ms	30.02 ms	245.58 ms
Total Time	1.44 ms	59.49 ms	499.38 ms
Kernel speedup vs CPU	3.5×	25.73×	21.65×
Overall speedup vs CPU	0.38×	0.63×	0.72×

Table 7: Averaged over 5 Runs for Block Size = 8×8×8

Pinned Memory

- **Pinned Memory Allocation**: Uses `cudaHostAlloc()` to allocate page-locked (pinned) host memory instead of regular pageable memory, preventing the OS from swapping these pages to disk and enabling Direct Memory Access (DMA) between host and device

- **Kernel Computation**: The kernel itself is identical to version (iii) with optimized shared memory tiling, index precomputation, and register-based accumulation—pinned memory only affects transfer performance, not computation
- **Faster Data Transfers**: Pinned memory enables higher bandwidth transfers via DMA, bypassing the intermediate staging buffer required for pageable memory, resulting in approximately 2× faster `cudaMemcpy()` operations
- **Performance Trade-offs**: While transfers are faster, pinned memory reduces available system RAM (cannot be swapped) and has allocation overhead, making it most beneficial for repeated transfers or large data sizes where transfer time dominates

Metrics	`N = 64`	`N = 256`	`N = 512`
`H2D` Data Transfer Time (**Pinned**)	`0.26 ms`	`16.79 ms`	`133.81 ms`
`H2D` Data Transfer Time (**Normal**)	`0.59 ms`	`28.92 ms`	`213.59 ms`
`H2D` Data Transfer Speedup	`2.35×`	`1.81×`	`1.47×`
Kernel Execution Time	`0.12 ms`	`1.54 ms`	`12.17 ms`
`D2H` Transfer Time (**Pinned**)	`0.28 ms`	`15.67 ms`	`128.87 ms`
Total Time	`0.68 ms`	`34.08 ms`	`274.06 ms`
Kernel speedup vs CPU	`4.34×`	`23.84×`	`25.08×`
Overall speedup vs CPU	`0.81×`	`1.07×`	`1.19×`

Table 8: Averaged over 5 Runs for Block Size = `8×8×8`

Performance Results

These functions were tested on `csews` with IP `172.27.28.135`, with the following hardware configurations,

- **Processor**: `Intel(R) Core(TM) i9-14900` with `32` cores
- **Cache**:
  - L1d: `896 KiB (24 instances)`
  - L1i: `1.3 MiB (24 instances)`
  - L2: `32 MiB (12 instances)`
  - L3: `36 MiB (1 instance)`

Problem 2

Implement an inclusive prefix sum algorithm with CUDA for unsigned integer type for large input sizes that overflow the GPU memory. You are free to adapt any known parallel prefix sum algorithm for CUDA, but you are not allowed to use library implementations (e.g., Thrust).

You will implement two versions: (i) with the "copy–then–execute" model without using UVM features, and (ii) use UVM support with `cudaMallocManaged()`. The "copy–then–execute" model should perform better than UVM for input sizes that fit in the GPU. Increase the value of the input to oversubscribe the GPU memory for the UVM implementation.

Use memory advises (e.g., `cudaMemAdvise()`) and prefetch hints (e.g., `cudaMemPrefetchAsync()`) to improve the performance. Usually, you have to try out all possible variations of the hints and check which is the best empirically.

Solution

Implementation

1. **Hierarchical Parallel Prefix Sum:** Two-level scan using warp-level shuffles (`__shfl_up_sync`) and shared memory for block-level aggregation. Multi-block inputs use recursive block-sum scanning with offset propagation.
2. **CPU Baseline:** Sequential inclusive scan with `std::chrono` timing for performance comparison baseline.
3. **Copy-Then-Execute (CTE):** Explicit memory management via `cudaMalloc`/`cudaMemcpy`. Reports kernel-only time (computation) and total time (with H2D/D2H transfers).
4. **Unified Virtual Memory (UVM):** `cudaMallocManaged` with `cudaMemAdvise` hints and `cudaMemPrefetchAsync` for optimized automatic memory management. Reports kernel and total time separately.

Performance Results

These functions were tested on `gpu3`

Problem Size	CPU (ms)	copy-then-execute (ms)	UVM
2^{10}	0 ms	4.89 ms	3.78 ms
2^{20}	4 ms	13.21 ms	6.56 ms
2^{30} (4 GB)	5775 ms	242.34 ms	205.83 ms
2^{32} (16 GB)	-	296.22 ms	297.15 ms

Table 9: Performance comparison without data initialization or copying

Problem Size	CPU (ms)	copy-then-execute (ms)	UVM
2^{10}	0 ms	9.76 ms	8.66 ms
2^{20}	4 ms	17.76 ms	13.71 ms
2^{30} (4 GB)	5775 ms	4575.34 ms	1374.54 ms
2^{32} (16 GB)	-	23490.7 ms	31210.5 ms

Table 10: Performance comparison with data initialization or copying

Problem 3

Parallelize the attached C code (`problem3-v0.c`) using CUDA. You will implement four versions.

(i) The first version will be a vanilla port of the C code. Your goal in this version is to ensure correctness. The challenges are in mapping the large iteration space of the 10D loop nest and writing back the output data from the device to the host. Implementing optimizations is optional.

(ii) Improve the performance of version (i) using different possible optimizations (e.g., shared memory tiling, launch multiple kernels, data prefetching and `memadvise`).

(iii) Implement the C code with UVM support with CUDA. Use memory advises and prefetch hints to improve the performance.

(iv) Implement a version using different transformations provided by Thrust. A few Thrust transformations are well-suited for the given problem.

Compare the performance of the four versions. You are allowed to partition the work across multiple kernel launches given the large iteration space.

Solution

Optimised

- **Parallelization Strategy:** The implementation maps the first 5 dimensions of the 10D grid search space to the GPU's thread hierarchy (dimensions 1-3 to block indices and dimensions 4-5 to thread indices within blocks), while the remaining 5 dimensions (6-10) are handled through nested sequential loops within each thread. This approach achieves a grid configuration of $s_1 \times s_2 \times s_3$ blocks with thread blocks of size $\min(14, s_4) \times \min(14, s_5) \times 1$, where s_i represents the number of grid points along dimension i.
- **Constraint Evaluation and Result Storage:** Each thread iterates through all combinations of the last 5 dimensions, evaluating 10 linear constraint equations at each grid point by computing $|c_i \cdot x - d_i| \leq k \cdot e_i$ for $i = 1, \ldots, 10$. Valid solutions satisfying all constraints are stored using atomic operations (`atomicAdd`) to maintain a global counter and append results to device memory arrays, which are subsequently copied back to the host and written to a file in sorted order.

UVM

- **Unified Virtual Memory (UVM) with Managed Allocations:** Replaced explicit GPU memory allocations (`cudaMalloc`) and host-device transfers (`cudaMemcpy`) with unified managed memory (`cudaMallocManaged`).
- **Memory Advise Hints for Performance Optimization:** Applied CUDA memory advise APIs to optimize UVM behavior: `cudaMemAdviseSetReadMostly` marks grid and coefficient arrays as predominantly read-only to enable efficient multi-GPU replication.
- **Explicit Prefetching for Reduced Page Fault Overhead:** Utilized `cudaMemPrefetchAsync` to asynchronously migrate input data (grid parameters and constraint coefficients) to GPU memory before kernel launch, and prefetch results back to CPU after computation.

Thrust

- **Thrust Library:** Utilized the Thrust C++ template library to implement grid search using high-level parallel primitives instead of explicit CUDA kernels. The implementation employs `thrust::counting_iterator` to generate a linear index space spanning the entire 10-dimensional grid.
- **Automatic Memory Management via Device Vectors:** Leveraged `thrust::device_vector` and `thrust::host_vector` containers for automatic memory allocation, deallocation, and host-device transfers. This RAII-based approach eliminates manual `cudaMalloc`/`cudaFree` and `cudaMemcpy` calls, reducing memory management errors.

Metric	Vanilla	Optimised	UVM	Thrust
Data Read Time	696 μs	606 μs	89 μs	1207 μs
Memory HtoD Time	299 ms	228 ms	1 ms	186 ms
Kernel Compute Time	109623 ms	16962 ms	13621 ms	25934 ms
Total Time	109884 ms	17192 ms	13838 ms	26121 ms

Table 11: Execution Times

Problem 4

Convolution is an array operation where each output data element is a weighted sum of a collection of neighboring input elements. The weights are provided via an input array that we will call the convolution filter.

(i) Implement a CUDA kernel to compute 2D convolution for each element of an input 2D array by averaging the values of its neighboring elements.

(ii) Implement a CUDA kernel to compute 3D convolution for each element of an input 3D array by averaging the values of its neighboring elements in a cuboid of unit length keeping the element under consideration at the center.

- Use single-precision floating point type.
- Assume that the 2D and 3D arrays are squares and cubes. That is, sides have the same dimension and are a multiple of eight.
- The convolution filter will be a square with odd dimensions.
- Assume the missing boundary elements (i.e., ghost cells) are zero.

Implement two versions of the 2D and 3D kernels. The first one should be a basic version that uses global memory while the second one is your most optimized version that implements (a) tiling using shared memory, (b) constant memory for the filter, and (c) loop unrolling. Compare and report the performance improvements.

Solution

2D Convolution Implementation

The 2D convolution computes each output pixel as the average of its surrounding neighbors within a `FILTER SIZE` window. The CUDA implementation parallelizes over all pixels, assigning one thread per output pixel.

Optimized Implementation (Shared Memory, Constant Memory, Loop Unrolling) The optimized version improves upon the naïve implementation through multiple layers of CUDA-level optimization:

a) **Tiling with Shared Memory:** Each thread block loads a tile of the input image into shared memory, including the halo elements required for neighboring computations. Shared memory provides low-latency access and drastically reduces redundant global reads.

- The tile size (`TILE_WIDTH`) determines occupancy and data reuse. Tiles of size $4, 8$, and 16 are tested.

b) **Constant Memory for Filter:** The convolution filter, being small and read-only, is stored in `__constant__` memory. This allows all threads in a warp to access the same filter value.

c) **Loop Unrolling:** Since the filter dimensions are fixed and small, inner loops are unrolled using `#pragma unroll`, reducing loop-control overhead and increasing instruction-level parallelism.

Performance Results

All runs were averaged over 10 iterations.

Implementation	Time (s)
CPU (Sequential)	`0.12 ms`
GPU Naïve (Global)	`0.05 ms`
GPU Optimized (`TILE = 4`)	`0.025 ms`
GPU Optimized (`TILE = 8`)	`0.029 ms`
GPU Optimized (`TILE = 16`)	`0.032 ms`

Table 12: Execution times for 2D convolution ($N = 2^6$).

Implementation	Time (s)
CPU (Sequential)	1.32 ms
GPU Naïve (Global)	0.012 ms
GPU Optimized (TILE = 4)	0.011 ms
GPU Optimized (TILE = 8)	0.007 ms
GPU Optimized (TILE = 16)	0.006 ms

Table 13: Execution times for 2D convolution ($N = 2^8$).

Implementation	Time (s)
CPU (Sequential)	27.035 ms
GPU Naïve (Global)	0.0808 ms
GPU Optimized (TILE = 4)	0.0020 ms
GPU Optimized (TILE = 8)	0.0031 ms
GPU Optimized (TILE = 16)	0.0256 ms

Table 14: Execution times for 2D convolution ($N = 2^{10}$).

3D Convolution Implementation

The 3D convolution extends the 2D approach to a cubic input of size $N \times N \times N$, with each element computed as the mean of its neighbors within a $(\texttt{FILTER SIZE})^3$ volume.

Optimized Implementation (Shared Memory, Constant Memory, Loop Unrolling) The 3D version adopts similar optimization principles as the 2D case but requires more careful resource management due to the volumetric data.

a) **3D Shared-Memory Tiling:** A 3D tile of the input cube is loaded into shared memory. For each block of threads, the shared tile includes a halo region in all three dimensions. Threads synchronize after loading before performing convolution.

b) **Constant Memory for Filter:** The 3D filter (e.g., $3 \times 3 \times 3$) is stored in `__constant__` memory. Threads reading the same weight index exploit constant cache broadcast.

c) **Loop Unrolling:** For small kernels, the innermost three loops are unrolled to eliminate control overhead and expose instruction-level parallelism, letting the compiler optimize the accumulation pipeline.

Performance Behavior: Shared-memory tiling offers significant improvements for large volumes, as each global element fetched is reused multiple times by neighboring threads. Constant memory further improves efficiency, especially when the same filter weights are repeatedly accessed.

Performance Results (3D Convolution)

Implementation	Time (s)
CPU (Sequential)	13.49 ms
GPU Naïve (Global)	0.07 ms
GPU Optimized (TILE = 4)	0.057 ms
GPU Optimized (TILE = 8)	0.054 ms
GPU Optimized (TILE = 16)	0.033 ms
GPU Optimized (TILE = 16)	0.033 ms

Table 15: Execution times for 3D convolution ($N = 2^6$).

Implementation	**Time (s)**
CPU (Sequential)	`943.67 ms`
GPU Naïve (Global)	`1.62 ms`
GPU Optimized (`TILE = 4`)	`0.92 ms`
GPU Optimized (`TILE = 8`)	`0.72 ms`
GPU Optimized (`TILE = 16`)	`0.34 ms`

Table 16: Execution times for 3D convolution ($N = 2^8$).

Implementation	**Time (s)**
CPU (Sequential)	`55440.2 ms`
GPU Naïve (Global)	`FILL ms`
GPU Optimized (`TILE = 4`)	`63.84 ms`
GPU Optimized (`TILE = 8`)	`46.99 ms`

Table 17: Execution times for 3D convolution ($N = 2^{10}$).