POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Furlan Alberto

GPU101 - Convolution

Academic Year: 2024-2025

Introduction

The goal of the project is to develop and optimize a 2D convolution algorithm taking advantage of CUDA starting from a plain .cpp version.
Convolution is an operation involved in image and signal processing, deep learning and plays a crucial role in applications such as edge detection, feature extraction, and neural network computations.
Due to the way the convolution is performed, GPUs provide an efficient platform for accelerating computing operations due to their massive parallelism.
In order to achieve this CUDA framework, enables fine-grained control over GPU resources to optimize computational efficiency. However, efficiently mapping a 2D convolution onto the GPU requires careful handling of memory access, synchronization, and workload distribution to achieve peak performance.
Therefore, in order to obtain a decent implementation, I went through several steps, such as:

1. Algorithm design: in order to do so I took into account the architecture and the input's dimension and size in order to perform a proper grid and block setup before launching the kernel itself.
   This is a key aspect in order not to launch a higher number of threads than what you really need.
2. Optimization and performance analysis: this was essential to understander strengths and weaknesses of the algorithm itself and therefore to understand the exact sections of the code that impact the most on the overall performance.
3. Testing: essential to check the bare correctness of the whole program and to benchmark the performances.

Design

Algorithm use case

The algorithm is designed to perform a bi-dimensional convolution using a bi-dimensional size-fixed kernel.
More-over it is meant to operate on already-linearized bi-dimensional input data, as kernels are meant to operate on linear data.

Optimization steps

I developed a total of five version of the algorithm.

Plain version convolution

This version doesn't feature any kind of optimization, as this was meant to be used to measure the performance boost compared to the original cpp version.

Shared memory convolution

Shared memory is a very fast on-device memory shared among all threads in the specific block, that allows the developer to load data into it as to minimize data transfer overheads as well as cpu-to-gpu interactions.
Moreover shared memory allows to reduce the overall global memory accesses due to coalesced access.

Tiling convolution

Tiling is a technique that involves loading blocks of shared memory and reusing them to further more reduce the memory accesses.
This is especially useful for problems like matrix multiplication or convolution, where neighboring data points are reused.
This requires threads to be synched before starting any operation to ensure proper value loading and in order to properly perform tiling the tile size should be computed accordingly to the input itself.

Streams convolution

Streams enable overlapping kernel execution with memory transfers, improving throughput, this allows for kernels and memory copies to run in parallel.
Parallelization and scheduling is handled by the CUDA toolchain scheduler.

Streams and tiling combined

I came up with this idea in order to overcome device limitations on grid and block sizes that I found out through running integration tests.
Each CUDA-enabled device features specific characteristics, those defining active constraints that prevent launching kernels with a grid size that overcomes the max grid size available.
For instance let's refer to an Nvidia Tesla T4, that features a maximum block size of 1024 x 1024 x 64, maximum grid size 2.147.483.647, 65535, 65535 and a maximum shared memory size of 49152.
Let's try to feed the kernel with a 100000x100000 matrix (that will break the hardware bounds) and let's compute the amount of threads that this input would try to ask for:

- The grid size would be set to 6250 x 6250 x 1 as we're performing a bi-dimensional convolution.
- The block size would be set to 16 x 16 x 1, as we're performing a bi-dimensional convolution.
- The actual amount of threads along any of the two axis would be: 6250 x 16 = 100000 (as we could already imagine by looking at the input).

- This would exceed the limit of the y-axis maximum grid size and this would end up in a memory violation.

For this reason we could pair tiling with streams utilization and exploit the scheduler in order to split the workload.

Tests

I designed two suites in order to test output correctness and measure the performances.
Both of them take advantage of a generator that enforces multithreading input and kernel generation through openMP library.

Tests configurations

The integration tests run on two configurations only, as their aim is to ensure correctness.
The performance tests run on six different configurations in order to test the kernels behaviors under various workloads:

- 1000 x 1000;
- 10000 x 10000;
- 25000 x 25000;
- 50000 x 50000;
- 60000 x 60000;
- 75000 x 75000;

Integration tests

The main goal of these tests is just to ensure the output is correct and consistent.
These tests perform three types of checks in order to assure the output is correct:

- Output comparison with a slightly enhanced version of the original convolution algorithm (again throughout openMP)
- Output consistency check: wrong kernel launch-time parameters setup may lead to an output made up of zeroes only.
- Any type of compile-time and runtime exception.

These tests are realized using the google test cpp library.

Performance tests

These tests perform some really basic performance checks measuring:

- Kernel execution time;
- Kernel memory occupation;

These are done with a slight modification of the google test cpp library and were performed just on the basic version of the convolution and the last one (i.e.: the one featuring both CUDA streams and tiling).
These tests aim at highlighting the differences in the various kernels on different workloads, as for instance, performing a basic convolution on a 1000 x 1000 matrix is more efficient than performing a convolution that uses both tiling and streams.

Performance Report

The kernel was analyzed by means of Nvidia Nsyght Compute software to perform live profiling and Nvidia compute-sanitizer to check eventual memory violations.

Shared memory version

GPU utilization and throughput analysis

METRIC	VALUE
Compute (SM) Throughput	97.63%
Memory Throughput	97.63%
L1/TEX Cache Throughput	98.83%
L2 Cache Throughput	8.40%
DRAM Throughput	18.44%
Elapsed Time (Single Run)	10.71 ms

Both compute and memory throughput are very high: this means the kernel is able to almost fully utilize the GPU resources.
Caches performances are good for the L1 but pretty low for the L2 as well as DRAM's throughput, this means the L2 is causing more global memory accesses.
This could lead to new optimization techniques by reducing redundant memory loads and taking more advantage of shared memory locality.

Compute workload and Instruction throughput analysis

METRIC	VALUE
Executed IPC (Instructions per Cycle)	1.93
Issue Slots Busy	48.33%
ALU Utilization	28.5%
SM Busy	48.33

This table shows the SM utilization is capped at 48.33%, this means the almost half of the SMs are in IDLE during computation.
This could be caused by memory access stalls that may cause these SMs being idle.

Memory workload analysis

METRIC	VALUE
Memory Throughput	86.19 GB/s
L1/TEX Hit Rate	85.25%
L2 Hit Rate	67.85%
Memory Pipes Busy	97.63%
Shared Memory Conflicts (Loads)	49.85% of wavefronts
Shared Memory Conflicts (Stores)	30.00% of wavefronts

This data highlights an high volume of memory transactions, due to an uncoalesced global memory access.
Shared memory conflicts happen quite often, particularly higher during loads, which contribute to memory latency.

Scheduler and warp efficiency

METRIC	VALUE
Active Warps per Scheduler	11.55
Eligible Warps per Scheduler	2.02
Issued Warps per Scheduler	0.48
Warp Cycles per Instruction	23.90
Stall Reason (Memory I/O)	40.7%

Again these statistics show how memory stalls impact on the performance, with a 40.7% overall score, meaning warps spend time waiting for memory access instead of executing instructions.

Nsight Compute gains estimations

METRIC	VALUE
Uncoalesced Global Loads	2.08%
Shared Memory Load Conflicts	49.27%
Shared Memory Store Conflicts	29.65%
Warp Scheduling Efficiency	2.37%

This estimations lead to some possible optimization steps ideas, such as:

1. Improve Global Memory Coalescing:
   - Ensure that threads access contiguous memory locations.
   - Use structure-of-arrays (SoA) instead of array-of-structures (AoS).
2. Reduce Shared Memory Bank Conflicts:
   - Use padding to avoid multiple threads accessing the same memory bank.
   - Reorder memory accesses for better alignment.
3. Increase Warp Scheduling Efficiency:
   - Optimize instruction scheduling to reduce memory stalls.
   - Overlap computation and memory access where possible.
4. Reduce Memory Access Stalls:
   - Implement tiling to reuse shared memory efficiently.
   - Use registers instead of global memory when possible.

Streams and tiling version

GPU Utilization & Throughput

METRIC	VALUE
Compute (SM) Throughput	98.73%
Memory Throughput	98.73%
L1/TEX Cache Throughput	98.80%
L2 Cache Throughput	8.51%
DRAM Throughput	16.22% - 18.52%
Elapsed Time (Single Run)	~2.44 ms

This data show a very high compute and memory throughput, as well as the L1 cache. Nonetheless the DRAM output and the L2 cache hit rate are pretty low, indicating potential memory bottlenecks.

Compute workload and instruction throughput

METRIC	VALUE
Executed IPC (Instructions per Cycle)	2.07
Issue Slots Busy	51.67%
ALU Utilization	29.3%
SM Busy	51.67%

This informations show SM utilization is moderate (51.67%), meaning that while execution is balanced, it might benefit from more efficient memory access or workload distribution.

Memory Workload Analysis

METRIC	VALUE
Memory Throughput	81.31 - 90.33 GB/s
L1/TEX Hit Rate	83.89%
L2 Hit Rate	71.79%
Memory Pipes Busy	98.73%
Shared Memory Conflicts (Loads)	49.83% of wavefronts
Shared Memory Conflicts (Stores)	33.33% of wavefronts

This table shows the memory throughput is near peak (98.73%), meaning the memory system is being heavily used.
L2 hit rate is pretty high too, but could be enhanced.

Scheduler & Warp Efficiency

METRIC	VALUE
Active Warps per Scheduler	11.55
Eligible Warps per Scheduler	1.89
Issued Warps per Scheduler	0.52
Warp Cycles per Instruction	22.36
Stall Reason (Memory I/O)	34.7%

This data highlights 34.7% of warp stalls are due to memory I/O bottlenecks, which suggests that shared memory or DRAM access patterns are suboptimal.
As it can be seen, the warp availability is capped to 1.89.

Occupancy & Resource Utilization

METRIC	VALUE
Theoretical Occupancy	100%
Achieved Occupancy	96.39%
Achieved Active Warps per SM	46.27
Register Usage per Thread	38
Shared Memory Usage per Block	65.54 KB

This table highlights a good shared memory usage (65.54 KB/block), which might contribute to memory conflicts, as well as a good register usage per thread (38), which is reasonable but could potentially be optimized further.
High occupancy is set to 96.39% indicates efficient SM resource utilization.

Nsight Compute estimations

METRIC	VALUE
Theoretical Occupancy	100%
Achieved Occupancy	96.39%
Achieved Active Warps per SM	46.27
Register Usage per Thread	38
Shared Memory Usage per Block	65.54 KB

Possible improvement ideas:
1. Improve Global Memory Coalescing:
   - Ensure threads in a warp access consecutive memory addresses.
   - Avoid strided access patterns that cause excessive DRAM requests.
2. Reduce Shared Memory Bank Conflicts:
   - Reorder shared memory accesses.
   - Use padding to avoid multiple threads accessing the same memory bank.
3. Increase Instruction Parallelism:
   - Reduce stalling due to memory operations.
   - Balance compute vs. memory workload.
4. Optimize Warp Scheduling:
   - Increase eligible warps per cycle to avoid idle cycles.

Performance comparison

Metric	Basic Kernel	Tiled + Streams Kernel	Improvement
Execution Time (ms)	10.71	2.44	4.4x faster
SM Utilization (%)	48.33	98.73	2x better
Memory Throughput (GB/s)	86.19	90.33	Improved
L2 Cache Hit Rate (%)	67.85	71.79	Better caching

Memory considerations

I also performed a memory-only analysis with compute-sanitizer, a tool designed for memory profiling when running kernels.
These analysis highlighted some constraints on the input the various kernels can handle as I mentioned before.
This problems might be caused by out-of-bound memory accesses in the kernels themselves causing these violations and therefore the errors.
This aspected could drive some further optimization steps towards a more scalable algorithm.