High Performance Computing

National University of Computer and Emerging Sciences

Deliverable # 03

Complex Computing Problem

By

Hamza Kaleem — 23i-0783
Umaima — 23i-0790
Warisha Shaukat — 23i-0809

Contents

1 Optimizations Implemented

1.1 Shared Memory

Shared memory is leveraged in the convolution kernel to address the high latency and limited bandwidth of global memory accesses. The optimized kernel uses **tiled shared memory** to cache portions of the input image and the convolution kernel for each thread block. This approach reduces global memory traffic by approximately 30.5%, increases cache reuse, and improves memory throughput.

1.2 Batch Processing

To remove PCIe bottlenecks from frequent host–device transfers, convolution and feature-tracking files functions were updated to use a **batch processing pipeline** using GPU-side storage pools (`d_batch_inputs`, `d_batch_outputs`, `d_batch_temps`). Instead of issuing a transfer for every frame, multiple datasets are grouped into a single transaction. This reduces the number of host-to-device transfers, maximizes PCIe bandwidth utilization, and reduces kernel launch overhead across batches.

1.3 CUDA Streams

The `trackFeatures.cu` file implements a multi-stream execution model using a fixed-size **StreamPool** (`NUM_STREAMS = 4`). Each CUDA stream handles independent operations such as image pyramid generation or feature refinement. This approach launches kernels asynchronously across multiple streams. This concurrency increases overall SM occupancy and reduces idle GPU cycles.

1.4 Persistent Memory Pools and Resource Reuse

Persistent GPU-side data pools (`ConvolutionPool`, `StreamPool`) eliminate overhead caused by repeated memory allocation. This greatly reduces the use of expensive operations like `cudaMalloc` and `cudaFree` that were being called in a loop in the KLT source code. Additionally, kernel coefficients are stored permanently on the GPU, avoiding redundant uploads and maintaining data locality.

1.5 Other Optimization Techniques

Additional fine-grained optimizations were incorporated:

- **Launch configuration tuning:** Thread block sizes were optimized for maximum occupancy without exceeding shared memory per block. A two-dimensional thread and block configuration was used to efficiently map image pixels to GPU threads.

- **Occupancy optimization:** Register and shared memory usage were balanced to ensure multiple warps could remain active per Streaming Multiprocessor (SM), effectively hiding instruction and memory latency.
- **Static kernel residency:** Gaussian and derivative filters are uploaded once per session, avoiding repeated data transfers.
- **Unified error handling:** The `cudaCheck()` macro standardizes error detection, ensuring consistent runtime stability and early fault isolation.

2 Performance Results

2.1 CUDA API Summary

Name	Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)
cudaMemcpyAsync	66.2	767,046,879	23,166	33,110.9	2,190.0	1,627	14,353,739	488,297.4
cudaStreamSynchronize	15.4	178,811,727	15,363	11,639.1	13,345.0	466	115,534	5,634.9
cudaStreamCreateWithFlags	8.4	97,404,459	8	12,175,557.4	1,477.5	1,209	97,389,416	34,431,598.5
cudaMemcpy	5.9	68,315,115	36	1,897,642.1	1,863,906.5	112,707	4,075,157	1,764,528.2
cudaLaunchKernel	3.8	44,099,169	15,354	2,872.2	2,769.0	2,536	132,633	1,479.6

Table 1: CUDA API Summary

2.2 CUDA GPU Kernel Summary

Name	Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)
convolveImageVertKernel_SM	29.2	21,749,640	63	345,232.4	247,975.0	13,857	1,305,925	427,758.6
convolveImageHorizKernel_SM	29.2	21,705,952	63	344,538.9	247,175.0	16,416	1,299,557	425,404.3
computeGradientSumKernel_SM	24.5	18,216,061	7,614	2,392.4	2,240.0	1,888	3,329	281.9
computeIntensityDifferenceKernel	17.2	12,773,080	7,614	1,677.6	1,632.0	1,344	2,336	153.2

Table 2: CUDA GPU Kernel Summary

2.3 CUDA GPU MemOps Summary

Operation	Time (%)	Total Time (ns)	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)
[CUDA memcpy Device-to-Host]	51.0	383,153,989	22,905	16,728.0	1,440.0	512	11,167,514	374,799.3
[CUDA memcpy Host-to-Device]	49.0	367,890,419	297	1,238,688.3	194,726.0	352	4,095,059	1,673,782.0

Table 3: CUDA GPU MemOps Summary

2.4 Discussion of Results

The aforementioned optimization techniques minimized redundant memory operations and allowed better overlap of computation and data transfer. The most impactful enhancement was the introduction of **batch processing** which significantly reduced the number of Host-to-Device transfers from **37,656** to just **297**. Batch processing also greatly reduced GPU idle time. As a result of this the total execution time decreased from **12.3 seconds** to **7.2 seconds**, achieving a measured **1.7x speedup**.

3 Conclusion

The profiling results confirm the effectiveness of the implemented optimizations. Shared memory tiling and stream concurrency yielded major speedups, while batch processing significantly reduced Host-to-Device. However, Device-to-Host transfers (22,905) still dominate GPU time, representing the primary bottleneck. Future optimizations could include kernel fusion or asynchronous reduction strategies to minimize D2H synchronization overheads and achieve further throughput gains.