# CS4110-High Performance Computing with GPUs


## Project V4

Ibraheem Farooq 23i-0816
Rayyan Waqar 23i-0531
**CS-5A**

# Version 1 (V1): Baseline Sequential Analysis

The **V1 deliverable consisted of profiling and hotspot identification**. Using Linux time, gprof, and custom instrumentation, we discovered that execution was dominated by:

- convolveImageHoriz
- convolveImageVert
- _KLTComputeGradients
- _interpolate

Algorithmic bottlenecks align closely with the project instructions that emphasize identifying CPU andGPU communication challenges and compute density.

The V1 results established the guiding principles:

1. Convolutions must be ported to GPU early.
2. Pyramid operations involve repetitive data movement—must minimize transfers.
3. Interpolation kernels are simple but numerous—require parallelization.
4. Feature tracking loops cannot be fully parallelized but benefit from GPU-side data locality.

V1 served as the theoretical foundation to motivate the GPU-focused phases that followed.

# Version 2 (V2): Naive CUDA GPU Port

In V2, four functions were ported directly to CUDA:

- _interpolate
- _calculateMinEigenvalues
- convolveHorizontal
  convolveVertical

### 4.1 Naive GPU Performance (D2 Results)

| Version | Runtime | Speedup |
|---|---|---|
| V1 (CPU) | 1.903 s | 1× |
| V2 (GPU naive) | **30.298 s** | **0.063× (≈16× slower)** |

Here the CPU runtime is 1.903s because we ran it on the original 9 image dataset.

### 4.2 Why V2 Failed

Profiling showed catastrophic inefficiencies:

- **280,287** calls each of:
  - cudaMalloc
  - cudaFree
  - Host–device memory transfers (cudaMemcpy)
- 97.6% of GPU time spent on memory operations, not computation
- Kernel execution time <1% of total runtime

In short, *the naive implementation ported computation but ignored GPU memory hierarchy and communication overheads*, violating the main CCP learning objectives.

### 4.3 Key Lessons from V2

1. The GPU must **retain** data, not reallocate it repeatedly.
2. Transfers must be **batched and minimized**.
3. Launching many tiny kernels destroys performance.
4. GPU benefits only arise when the data stays on the device for long periods.

These observations directly shaped the V3 approach.

# Version 3 (V3): Optimized CUDA Implementation

V3 focused entirely on eliminating the V2 bottlenecks and implementing real CUDA optimizations: persistent buffers, shared memory, reduced transfers, and memory-access locality.

One important improvement done initially was increasing the dataset size from 9 images to 149 images.

### 5.1 Optimizations Implemented

#### a) Persistent GPU Buffers

All device memory required for the pipeline (pyramids, gradients, temp images) was allocated **once**, re-used across all frames.

- Result: **50 s to 15 s** (3.3× improvement)

**b) Shared Memory Tiling for Convolution**

Both horizontal and vertical convolution kernels were re-written using shared memory tiles.

- Result: **15 s to 9 s** (1.67× improvement)

**c) GPU-resident Pyramid Construction**

A custom down-sampling kernel allowed multi-level pyramid generation **without CPU involvement**, enabling the final caching optimization.

**d) Pyramid Caching (The Breakthrough Optimization)**

We discovered that the reference implementation rebuilt and transferred the image pyramid **for every single interpolation**, causing ~133,221 transfers.

By restructuring pyramid management:

- Transfers: **133,221 to ~100**
- Transfer time: **21.8 s to <0.3 s**
- Kernel throughput increased due to vastly reduced launch latency
- Total runtime: **9 s to <3 s**

## 5.2 Final V3 Performance

| Version | Runtime | Notes |
|---|---|---|
| Naive CUDA | 50 s | V2 baseline on 150 images |
| V3 Final | ≤**3 s** | 7 - 8x faster than CPU |
| CPU Baseline | 20 s | For updated dataset |

Thus, V3 achieved a **7–8× speedup over CPU**, meeting the expected theoretical bounds.

## 5.3 Profiling Tools Used

- Nsight Systems (transfer and launch analysis)
- Nsight Compute (warp-level metrics)
- nvprof (API call count and bandwidth patterns)

V3 validated nearly every HPC principle covered in the course.

# Version 4 (V4): OpenACC Implementation

The final version aimed to replicate V3 optimizations using **OpenACC**—a directive-based programming model for portable GPU acceleration.

## 6.1 OpenACC Directives Introduced in V4

#### A. Pixel-parallel convolution loops

- Fully parallel over both dimensions
- Memory already present on device
- Vector length tuned to match warp behavior

#### B. Unified data region for separable convolution

- Ensures all data stays on GPU during convolution pipeline
- Mirrors CUDA persistent-buffer behavior from V3

#### C. Combined data region for gradient computation

- Eliminates redundant transfers across two gradient convolutions
- Equivalent to CUDA caching solution in V3 but accomplished in one directive block

## 6.2 V4 Performance

GPU runtime: 3.347 s

This matches V3 performance (~3.3 seconds), confirming that the OpenACC port preserved all major optimizations.

## 6.3 Comparing CUDA V3 and OpenACC V4

| Aspect | V3 (CUDA) | V4 (OpenACC) |
|---|---|---|
| Performance | ~3 s | ~3.3 s |
| Implementation control | Very high | High (directive-based) |
| Code readability | Lower | Higher |
| Data management | Manual | Automated via present/present_or_copyin |

| | | |
|---|---|---|
| Transfer minimization | Manual caching | Natural via data regions |

OpenACC proved capable of achieving nearly the same speed as hand-optimized CUDA while dramatically reducing complexity.

Speedup Achieved by OpenACC was 6-7x

# 8. Conclusion

This project took the KLT tracker through four increasingly optimized versions:

- **V1** established bottlenecks through profiling.
- **V2** demonstrated a naive GPU port and why it fails without proper memory management.
- **V3** delivered a deeply optimized CUDA implementation with persistent buffers, shared-memory tiling, and pyramid caching, achieving **~16× speedup over naive GPU** and **~7–8× over CPU**.
  **V4** successfully re-expressed these optimizations using OpenACC, achieving comparable performance with much less implementation overhead.

By the end, all CCP CLOs were achieved: identifying hotspots (CLO2), developing data-parallel solutions (CLO3), and evaluating performance using real profiling tools (CLO4). The KLT tracker became a fully GPU-accelerated, highly optimized application demonstrating real-world HPC engineering.

**Github Repo:** https://github.com/ibbi1020/HPC_i230816_i230531.git