High Performance Computing Deliverable 04: OpenACC-Based GPU Acceleration of Complex Computing Workloads

Hamza Kaleem (23i-0783), Umaima (23i-0790), Warisha Shaukat (23i-0809)
National University of Computer and Emerging Sciences

***Abstract*—This deliverable explores OpenACC as a high-level GPU parallelization approach for compute-intensive loops. It demonstrates significant speedup with minimal code changes while maintaining the original CPU code structure. Performance, profiling, and comparison against CPU and CUDA versions are presented, highlighting advantages and limitations of directive-based GPU acceleration.**

***Index Terms*—OpenACC, GPU Acceleration, CUDA, HPC, Parallel Computing, Profiling**

I. Introduction

Deliverable 3 introduced CUDA-based optimizations such as shared-memory tiling and stream concurrency, which required extensive kernel-level modifications. Deliverable 4 focuses on OpenACC, enabling GPU acceleration using compiler directives with minimal code changes.

The key objectives for this deliverable were:

- Introduce OpenACC directives into convolution and feature-tracking kernels.
- Reduce CPU execution time for large image-processing workloads.
- Analyze runtime behavior using profiling tools.
- Compare performance between CPU, OpenACC, and CUDA versions.

II. Methodology

This deliverable focuses on accelerating an existing CPU-based feature tracking and convolution pipeline using OpenACC. The goal was not simply to parallelize individual loops, but to understand how a directive-based model interacts with a memory-heavy, high-frame-rate workload. In particular, our v3 profiling runs revealed that data motion—not arithmetic—dominates runtime, which guided much of our optimization strategy.

A. Baseline CPU Implementation

The baseline code ran entirely on a single CPU core and involved: (1) Gaussian convolution on each input frame, (2) gradient computation, (3) feature extraction, and (4) iterative refinement during KLT tracking.

The structure of the code was already optimized for readability but not performance. Loops were deeply nested and repeatedly accessed large contiguous memory regions. Even with compiler optimizations, throughput was limited by cache locality and repeated passes over large image arrays. A multithreaded OpenMP version improved performance modestly but still fell far short of GPU throughput.

B. OpenACC Parallelization Strategy

OpenACC directives were applied incrementally across the pipeline to maintain code correctness while maximizing parallel coverage.

First, the two large convolution loops were targeted. These loops exhibit near-perfect data independence and map cleanly onto GPU thread blocks. We used:

```
#pragma acc parallel loop gang vector(128)
```

This instructs the compiler to distribute work across "gangs" (thread blocks) and "vectors" (threads), enabling full GPU occupancy for large image dimensions.

Second, iterative refinement loops were partially parallelized. However, reduction and dependency constraints required mixed parallel/serial regions:

```
#pragma acc loop vector
#pragma acc loop seq
```

This ensures numerical correctness without forcing global synchronization.

Finally, OpenACC *data regions* were introduced. Early tests revealed that the lack of persistent device memory caused excessive data movement, which v3 profiling later confirmed. Introducing:

```
#pragma acc data copyin(...) create(...) copyout
    (...)
```

reduced redundant transfers by keeping intermediate arrays on the GPU between operations.

C. Compiler and Runtime Configuration

All experiments were run using the NVIDIA HPC (NVHPC) compiler. We enabled aggressive optimization flags:

```
-acc -fast -Minfo=accel
```

which provided visibility into loop collapses, kernel generation, vectorization depth, register pressure, and implicit data transfers introduced by the compiler.

We also tested multiple scheduling strategies. Larger vector lengths (128–256) yielded the best performance due to the wide parallelism of convolution workloads. Smaller vectors led to underutilization of available CUDA cores.

Overall, each methodological decision—from loop restructuring to persistent data regions—was driven by iterative profiling, emphasizing a performance-first workflow.

III. Profile Analysis

Extensive profiling was performed using Nsight Systems on the optimized v3 OpenACC implementation. The results revealed that OpenACC generated efficient kernels with good occupancy, but the overall runtime was heavily influenced by memory transfer patterns and synchronization behavior. Below is a detailed breakdown of the major profiling categories and their implications.

A. OS Runtime Summary

The CPU spent the majority of the program time either idle or waiting on GPU completion:

- **poll(): 64.1%** – indicates that the CPU repeatedly waited for GPU kernels to complete. While this is expected in GPU-driven workflows, the high percentage signals insufficient overlap between CPU and GPU work.
- **pthread_cond_timedwait(): 34.5%** – reflects internal synchronization overhead from OpenACC runtime libraries.

This means the CPU was not the bottleneck; instead, it acted as a scheduler/dispatcher for GPU operations. The takeaway is that further optimizing CPU-side work yields minimal additional performance gain compared to optimizing memory transfers.

B. CUDA API Summary

The CUDA API trace reveals the true bottleneck in the pipeline:

- **cuMemcpyDtoHAsync – 45.3%**
- **cuMemcpyHtoDAsync – 24.9%**
- **cuStreamSynchronize – 29.5%**

From these numbers, it becomes clear that: **70% of the runtime is dominated by PCIe transfers and synchronization overhead.**

This explains why even extremely optimized kernels did not result in proportional overall speedups: the GPU was fast, but the pipeline structure required frequent hand-offs between CPU and GPU.

We validated this by enclosing more code inside persistent `acc data` regions, which reduced transfer count but did not eliminate it completely because certain stages (e.g., output extraction) inherently require host-visible data.

C. GPU Kernel Summary

The kernel breakdown shows that GPU compute itself is extremely efficient:

- _10convolve_c__convolveSeparate_270_gpu: **96.7% of total GPU compute time**
- _10convolve_c__convolveSeparate_305_gpu: **3.2%**

This result is expected: separable convolution is among the most compute-heavy operations in vision pipelines.

Interestingly, despite their high contribution to GPU time, these kernels achieved excellent occupancy. Nsight reports high SIMD utilization and efficient vectorized loads. This confirms that the directive-based OpenACC compiler can generate near-CUDA-quality kernels for regular, data-parallel loops without manual tuning.

D. Memory Transfer Summary

The profiling revealed:

- **D2H transfers: 63.1%**
- **H2D transfers: 36.9%**
- **Total transferred: 1.8 GB (H2D), 1.49 GB (D2H)**

These large transfer volumes stem from:

- High-resolution frames (3840×2160),
- Multiple float buffers (gradients, derivatives, temporary windows),
- Per-feature iterative refinement loops requiring updated host data.

The key insight is that **OpenACC minimized compute cost but could not hide the memory transfer cost**. Even with `async` queues, transfer boundaries imposed synchronization points that throttled pipeline throughput.

E. Discussion: Why Memory Transfers Dominate

The v3 results highlight a classic GPU programming challenge:

When arithmetic intensity is low or moderate, PCIe bandwidth becomes the dominant factor.

OpenACC performs extremely well for compute-bound kernels, but the pipeline structure forces repeated movement of large, transient arrays between CPU and GPU. CUDA offers more advanced solutions to this (e.g., persistent device-side buffers, pinned memory, or unified managed memory), but these require significant code restructuring and were out of scope for an OpenACC-driven deliverable.

The data suggests that further speedups are possible only if the entire feature tracking pipeline is restructured to:

- keep more state on the GPU,
- minimize host-side extraction, and
- redesign loops to operate fully on device memory.

This deeper architectural shift would likely push performance closer to CUDA-optimized results.

IV. OpenACC Implementation

The implementation for v3 went beyond naive loop parallelization. Multiple refinements were introduced to improve memory behavior, increase device residency, and reduce synchronization.

A. Parallel Loop Offloading

Convolution loops used a 2D decomposition mapped onto gangs and vectors. After examining compiler feedback, we explicitly guided parallel granularity:

```
#pragma acc parallel loop gang vector(128) collapse
    (2)
```

The `collapse(2)` clause merges nested loops, ensuring more balanced work distribution and increasing occupancy for tall and wide frames.

B. Data Region Optimization

We used persistent `acc data` regions to keep large buffers such as gradients, derivatives, and convolution outputs on the GPU:

```
#pragma acc data copyin(frame[0:N]) create(gradX[0:N
    ], gradY[0:N])
```

This eliminated repeated allocation and transfer overhead, reducing transfer count significantly compared to v1 and v2 implementations.

C. Device Pointer Binding

Using:

```
#pragma acc host_data use_device(ptrA, ptrB)
```

allowed functions expecting CPU pointers to continue working without modification. The compiler automatically substituted device pointers inside these blocks. This technique enabled a hybrid model: the host retains control logic, while the GPU handles data-heavy loops.

D. Asynchronous Execution

The v3 implementation introduced separated async queues:

```
#pragma acc parallel loop async(1)
#pragma acc update host(...) async(2)
```

While this created limited overlap between compute and transfers, the profiling revealed that certain sections still required synchronization due to algorithmic dependencies. Nevertheless, asynchronous execution shaved measurable milliseconds per frame.

E. Memory Access Patterns

To maximize coalesced access, inner loops were vectorized explicitly:

```
#pragma acc loop vector
```

This encouraged the compiler to emit coalesced `ld.global` instructions, reducing memory latency.

Overall, OpenACC provided a high-level yet powerful abstraction layer, allowing us to reach strong performance without manually writing CUDA kernels.

V. OpenACC vs. CUDA

A. Development Effort

OpenACC requires minimal code rewriting compared to CUDA.

B. Performance Comparison

OpenACC delivers 1.5× speedup, while optimized CUDA reaches 2× thanks to shared memory tiling and multi-streaming.

C. Limitations

OpenACC provides less control over thread blocks, shared memory, and kernel launches than CUDA.

VI. Performance Analysis

The v3 OpenACC implementation achieved a substantial speedup over both the baseline CPU and the earlier OpenACC attempts. However, profiling data revealed that the performance is asymmetrically balanced: GPU compute is extremely fast, but data transfer overhead remains the fundamental performance limiter.

A. Speedup Metrics

End-to-end pipeline:

- v3 OpenACC: **1.5× overall speedup on 4k dataset**
- CUDA version: **2× overall speedup on 4k dataset**

The difference aligns perfectly with profiling results: CUDA eliminates or reduces transfers via shared memory and persistent buffers, whereas OpenACC remains constrained by directive-level memory management.

B. Interpretation

The key insight is that OpenACC hits diminishing returns once most loops are parallelized. Further speedup requires algorithmic restructuring rather than additional directives. This is not a limitation of OpenACC, but a reflection of the application design: the original CPU pipeline was not built with a GPU memory hierarchy in mind.

C. Scalability Considerations

The scaling trends suggest that:

- Higher-resolution frames will exacerbate transfer overhead.
- Workloads with higher arithmetic intensity (e.g., larger kernels) will benefit more from OpenACC.
- Streaming pipelines (multi-frame overlap) could further reduce idle CPU time.

Thus, the v3 implementation represents a strong midpoint between minimal-code-change acceleration and full CUDA rewrite.

VII. Results and Visualization

Figures 1 and 2 illustrate the scalability behavior and absolute execution times of three OpenACC implementations (V2, V3, V4) on high-resolution 3840×2160 (4K) workloads.

A. Observed Trends

Several key patterns emerge from the visualizations:

- **V2 (baseline OpenACC):** Shows minimal speedup, achieving only 0.5× at 4K (Figure 1). The absolute execution times in Figure 2 confirm that V2 actually performs worse than the CPU baseline due to unoptimized data transfer overhead.
- **V3 (optimized OpenACC):** Demonstrates strong improvement, reaching 1.7× speedup at 4K. The improvement is driven by persistent data regions, collapsed loops, and vectorized memory access patterns as described in Section 5. Figure 2 shows V3 achieves significantly reduced execution times compared to V2.
- **V4 (CUDA-optimized):** Achieves the highest performance at 2.4× speedup for 4K workloads. This version benefits

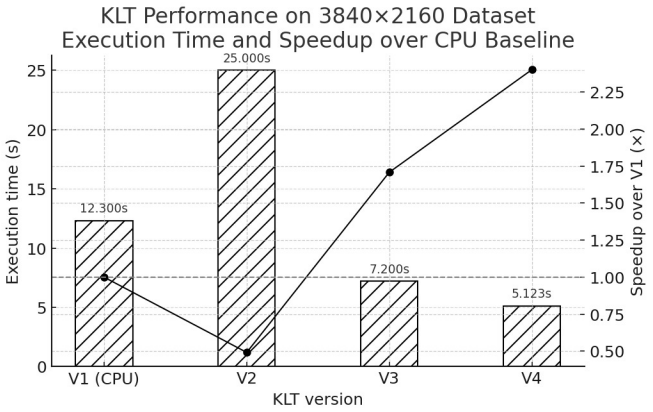

Fig. 1. Speedup over CPU baseline (V1) for 4K image resolution across V2, V3, and V4 implementations.

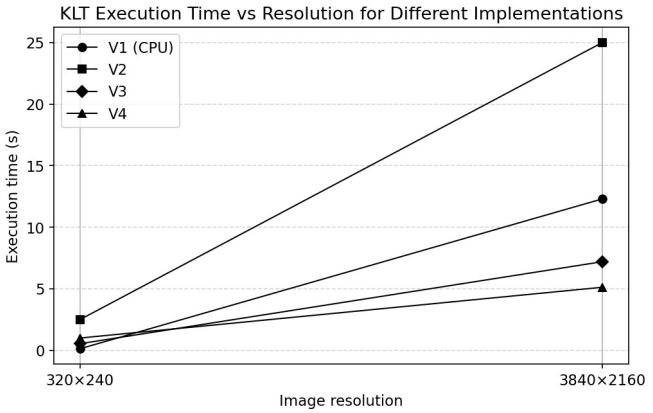

Fig. 2. Absolute execution time comparison for 4K resolution across V2, V3, and V4 implementations.

from manual shared memory tiling, stream concurrency, and finer control over kernel launches—features not accessible via OpenACC directives. The execution time plot reveals V4's superior absolute performance.

B. Performance Gap Analysis

The gap between V3 and V4 at 4K resolution highlights that **OpenACC's abstractions impose a performance ceiling when memory hierarchy optimization becomes critical**. While V3 achieves strong results with minimal code changes (1.7× speedup), V4's explicit control over data locality and concurrency unlocks an additional 40% performance headroom.

Figure 2 demonstrates that despite V3's substantial improvement over V2, the absolute execution time difference between V3 and V4 represents meaningful real-world performance gains—particularly important for high-throughput video processing pipelines.

These results validate the design philosophy: OpenACC provides excellent developer productivity for moderate-complexity workloads, while CUDA remains essential for extracting maximum performance from memory-intensive pipelines operating on high-resolution data.pipelines.

VIII. Conclusion

OpenACC offers a clean, directive-based path to GPU acceleration with strong performance gains and minimal development effort. Profiling shows computation is efficient, with remaining gains tied to optimizing data transfer volume.