15-618 Final Report

Parallel Video Reconstruction via Motion-Aware Predictive Upsampling

Andrew Kim (akim2) Martin Lee (hyungwol)

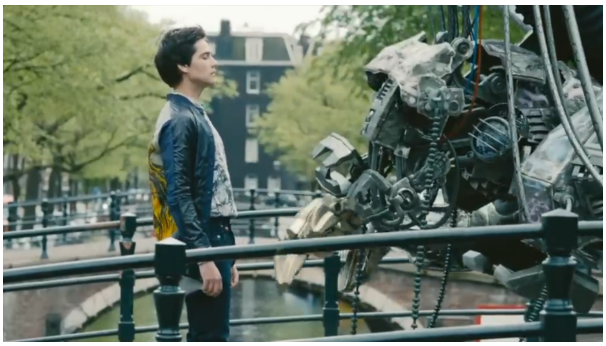
Link to Project Website
Link to Github Repository

Summary

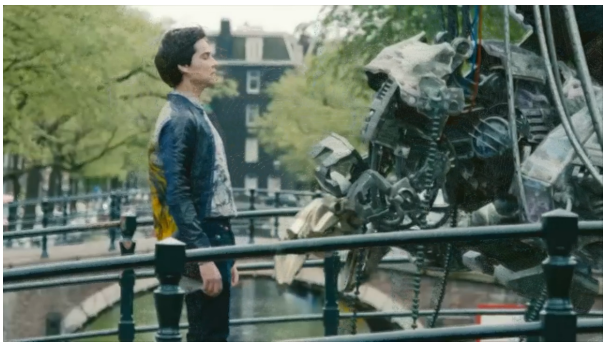
For our 15-618 final project, we parallelized a motion-aware video reconstruction pipeline that recovers dense frames from aggressively subsampled inputs. The system performs tile classification, predictive upsampling, and refinement, each of which exposes substantial data parallelism. We implemented the pipeline on both CPUs and GPUs using OpenMP and CUDA, allowing us to evaluate how effectively parallel hardware offsets the additional computation introduced by reconstruction while preserving PSNR/SSIM quality. Our deliverables include a complete CPU+GPU implementation, performance analysis comparing serial, OpenMP, and CUDA versions, and visual demonstrations of reconstruction quality. Experiments were run on GHC Linux machines for CPU/OpenMP and on PSC V100 GPU nodes for CUDA; the system relies on OpenMP, CUDA, and OpenCV for parallelism, acceleration, and frame processing.

Background

Our project addresses the problem of video reconstruction from *subsampled* frames, where only every `factor`-th pixel is retained and the remaining values must be inferred. The underlying algorithm treats each frame as a dense 2D RGB array and processes it through three conceptual stages: motion-aware tile classification, predictive upsampling, and iterative local refinement. These operations exploit the spatial structure of video and the observation that motion tends to be locally concentrated.



(a) Original frame



(b) Reconstructed frame

Figure 1: Comparison between an original frame and the corresponding frame reconstructed by our method. Despite reduced input sampling, the reconstruction maintains global structure and object boundaries, with only minor texture loss.

Each frame is partitioned into a uniform grid of tiles of size `tileSize` × `tileSize`. Tiles form the natural unit of computation: once a tile's motion status is determined, its reconstruction is fully independent of other tiles. A per-pixel `mask` records which pixels survived subsampling, and per-tile flags `tileActive` denote whether a tile exhibits motion.

- **Key Data Structures:** The algorithm maintains (1) dense RGB frames stored as 2D arrays, (2) a tile grid defining fixed-size regions of the frame, (3) a binary motion mask indicating active vs. static tiles, and (4) intermediate buffers for predicted and refined reconstructions.
- **Inputs and Outputs:** Each input frame is first subsampled using a spatial stride `factor`, leaving only a sparse set of known pixels. The output is a fully reconstructed video sequence where missing values are inferred through interpolation and smoothing. Intermediate products—motion maps, predicted frames, and refinement results—are used for analysis.
- **Algorithmic Components:** The core computations are:
  1. *Motion-aware tile classification*: average per-tile motion is estimated using the Sum of Absolute Differences (SAD) on subsampled RGB pixels.
  2. *Predictive upsampling*: missing pixels are estimated using bilinear interpolation conditioned on the subsampling mask.
  3. *Iterative refinement*: a Jacobi-style smoothing procedure iteratively updates each pixel using a 3×3 local stencil to enforce spatial consistency.
- **Workload Structure and Parallelism:**
  - **Dependencies:** The conceptual pipeline is sequential (classify → reconstruct → refine), but within each stage, tiles and pixels are independent, enabling abundant parallelism.
  - **Data Parallelism:** Each core step applies uniform per-tile or per-pixel computations, making the workload naturally data-parallel. Classification computes SAD per tile; reconstruction computes pixel values independently; refinement applies a uniform stencil.

- **Locality and SIMD/SIMT Compatibility:** Bilinear interpolation and Jacobi refinement operate on small spatial neighborhoods (2×2 and 3×3 respectively). These structured stencil accesses lead to strong spatial locality and map efficiently onto both SIMD hardware (CPU) and SIMT execution models (GPU).

Algorithms 1–3 summarize the computation performed in each stage.

Algorithm 1 Motion-aware tile classification (SAD-based)

procedure ClassifyTiles(curr, prev, τ, tileSize)
  if prev is empty **then**
    mark all tiles as Active
    return
  end if
  for all tiles T in the frame **in parallel do**
    sad $\leftarrow 0$
    for all pixels $(y, x) \in T$ **do**
      sad $\leftarrow$ sad $+ \|\text{curr}[y, x] - \text{prev}[y, x]\|_1$ ▷ RGB L1
    end for
    avgSAD $\leftarrow$ sad$/(|T| \cdot 3)$
    if avgSAD $> \tau$ **then**
      tileActive$[T] \leftarrow 1$ ▷ moving tile
    else
      tileActive$[T] \leftarrow 0$ ▷ static tile
    end if
  end for
end procedure

Algorithm 2 Predictive upsampling via bilinear interpolation

procedure ReconstructFrame(subsampled, mask, factor)
  for all pixels (y, x) in frame **in parallel do**
    if mask$[y, x] = 1$ **then**
      pred$[y, x] \leftarrow$ subsampled$[y, x]$ ▷ known sample
    else
      $(x_0, y_0), (x_1, y_1) \leftarrow$ neighboring grid points
      clamp (x_0, x_1, y_0, y_1) to valid image bounds
      gather corner colors only where mask$= 1$
      if no valid corners **then**
        pred$[y, x] \leftarrow$ nearest sampled grid point
      else
        compute bilinear weights $(w_{00}, w_{10}, w_{01}, w_{11})$
        pred$[y, x] \leftarrow$ weighted average of valid corners
      end if
    end if
  end for
  return pred
end procedure

Algorithm 3 Tile-aware iterative refinement (Jacobi smoothing)

procedure RefineTiles(img, mask, tileActive, K)
  cur $\leftarrow$ img; allocate next of same size
  for $t = 1$ to K **do**
    for all pixels (y, x) **in parallel do**
      $T \leftarrow$ tile containing (y, x)
      if tileActive$[T] = 0$ **then**
        next$[y, x] \leftarrow$ cur$[y, x]$ ▷ static tile, no smoothing
      else if mask$[y, x] = 1$ **then**
        next$[y, x] \leftarrow$ cur$[y, x]$ ▷ anchor known samples
      else
        next$[y, x] \leftarrow$ average of 3×3 neighborhood around (y, x) in cur
      end if
    end for
    swap(cur, next) ▷ Jacobi update
  end for
  return cur
end procedure

Approach

Our implementation instantiates the algorithmic pipeline described in the Background section on both CPU and GPU hardware. The system processes video frames sequentially but exploits

massive parallelism within each stage through OpenMP on CPUs and CUDA on GPUs. The design goal was to build a high-throughput reconstruction pipeline that minimizes redundant work and maximizes hardware utilization.

End-to-End System Architecture

The implementation consists of four major components:

1. **Frame I/O and subsampling** using OpenCV and a custom subsampling routine that produces both a sparse frame and a binary pixel-level `mask`.
2. **Motion-aware tile classification**, which compares the current and previous subsampled frames via SAD and produces a `tileActive` mask.
3. **Tile-restricted predictive upsampling**, which reconstructs only active tiles using bilinear interpolation.
4. **Tile-restricted iterative refinement**, which performs K Jacobi iterations only on active tiles, using a 3×3 neighborhood stencil.

Static tiles are passed through without modification, reducing unnecessary computation and significantly improving performance, especially on low-motion sequences. Furthermore, we removed the temporal refinement stage, which was originally part of the proposed reconstruction pipeline. Empirical evaluation showed that it did not yield meaningful improvements in reconstruction quality and, in some cases, introduced motion-smearing artifacts that decreased visual fidelity. As a result, we excluded temporal refinement from the final system.

Technologies Used

The system is implemented in `C++17` and parallelized using:

- **OpenMP**: for CPU tile-parallel and pixel-parallel computation.
- **CUDA**: for GPU acceleration of reconstruction and refinement.
- **OpenCV**: for video input/output and frame conversion.

CPU experiments ran on CMU GHC machines, and GPU experiments on PSC nodes with NVIDIA V100 GPUs.

Mapping to Parallel Hardware

CPU (OpenMP)

Loop nests operating on tiles or pixels are parallelized using:

- `#pragma omp parallel for` for pixel-level work,
- `#pragma omp parallel for collapse(2)` for tile-level work.

Because tile processing is independent and refinement uses a Jacobi update, no intra-iteration synchronization is required beyond OpenMP barriers between Jacobi steps.

GPU (CUDA)

GPU kernels use:

- 2D thread blocks (typically 16×16 threads),
- SIMT execution with each thread computing one pixel,
- coalesced memory access enabled by row-major storage.

We use persistent device buffers for frames, masks, and intermediate results to avoid repeated allocation and to reduce PCIe transfer overhead.

Three dedicated kernels mirror the algorithmic stages:

1. **Reconstruction kernel**: copies known pixels or performs bilinear interpolation.
2. **Full-frame refinement kernel**: applies the Jacobi stencil.
3. **Tile-aware refinement kernel**: restricts computation to active tiles using structured predicates.

Modifications for Better Parallel Mapping

The baseline serial algorithm performs full-frame reconstruction and refinement on every frame. To better exploit parallel hardware, we:

- introduced a **tile-activity mask** to avoid redundant work,
- eliminated refinement on static tiles,
- rewrote GPU kernels to avoid warp-divergent branching,
- employed persistent device buffers for all CUDA data,
- coalesced memory accesses by aligning data layouts with thread-block geometry.

These revisions significantly reduced computation and overhead, particularly on the GPU.

CPU vs. GPU tile classification.

We also investigated whether the tile-classification stage (`classifyTilesSADSubsample`), which computes a per-tile SAD between consecutive subsampled frames, should be offloaded to the GPU. Although the operation is parallel across tiles, each tile contains only four pixels in our configuration, resulting in very low arithmetic intensity. To evaluate the trade-off, we implemented a CUDA version and compared it against our OpenMP CPU implementation, with results summarized in Table 1.

Metric	CPU Classify (OpenMP)	CUDA Classify
Tile classify time (ms/frame)	1.03	4.03
Tile classify share of compute	15.5%	41.6%
Compute-only time (s)	3.24	4.75
Compute-only time (ms/frame)	6.62	9.70
Quality (MSE/SSIM/PSNR)	identical	identical

Table 1: Performance comparison of CPU vs. CUDA tile classification (720p_20s).

The experiment shows that GPU classification increases total compute time by 47% and quadruples per-frame classification cost, despite producing the same reconstruction quality. Because kernel-launch overhead dominates and the GPU remains underutilized for such fine-grained work, tile classification is best kept on the CPU with OpenMP. In our final design, only the computationally heavier reconstruction and refinement stages are offloaded to CUDA.

Optimization Process

Initial Implementation The initial GPU pipeline suffered from excessive kernel-launch overhead, full-frame refinement regardless of motion, and repeated `cudaMalloc`/`cudaFree` calls.

Diagnosing Bottlenecks Profiling with CUDA events and runtime timers revealed:

- iterative refinement dominated frame time,
- transfer overhead was substantial when repeated every frame,
- many frames contained few active tiles, making full-frame processing wasteful.

Iterative Optimizations Through successive profiling and refactoring, we:

- introduced tile-aware refinement to eliminate work on static tiles,
- reduced memory traffic via persistent buffers,
- ensured coalesced memory access and minimized warp divergence,
- tuned block sizes and loop ordering for occupancy.

These changes yielded large performance improvements and made the GPU pipeline competitive with—and often faster than—our CPU implementation.

External Code

We used OpenCV solely for video decoding and encoding. All reconstruction, classification, and refinement logic is implemented from scratch.

Results

Evaluation Metrics

We evaluate our system using both *performance* and *reconstruction quality* metrics. On the performance side, we report only two quantities: **compute time** and **speedup**. On the quality side, we use the standard full-reference metrics **PSNR** and **SSIM**.

Performance Metrics

- **Compute Time (ms/frame).** The end-to-end runtime required to reconstruct a single frame, excluding disk I/O. This reflects the absolute latency of the pipeline and is directly comparable across serial, OpenMP, and CUDA configurations.
- **Speedup** Defined relative to the single-threaded CPU baseline:

$$\text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}}.$$

  Speedup indicates how effectively each parallel configuration (OpenMP or CUDA) exploits available hardware.

Quality Metrics

- **Peak Signal-to-Noise Ratio (PSNR).** PSNR is derived from the mean squared error (MSE) between the reconstructed frame $\hat{I}$ and the ground truth I:

$$\text{MSE} = \frac{1}{N}\sum_{p=1}^{N} \|\hat{I}_p - I_p\|_2^2, \qquad \text{PSNR} = 10\log_{10}\left(\frac{255^2}{\text{MSE}}\right).$$

  Higher PSNR corresponds to lower pixelwise distortion.
- **Structural Similarity Index (SSIM).** SSIM measures similarity in local luminance, contrast, and structural patterns. Ranging from –1 to 1, values near 1 indicate strong perceptual similarity. We compute frame-level SSIM using the standard windowed formulation of Wang et al. and report the mean across all windows.

Compute time and speedup together characterize the system's parallel performance, while PSNR and SSIM quantify the fidelity of the reconstructed frames.

Experimental Setup

All experiments were run on CMU GHC Linux machines (CPU/OpenMP) and PSC nodes with NVIDIA V100 GPUs (CUDA).

We evaluated our reconstruction pipeline on input videos at two resolutions: **720p** (1280×720) and **1080p** (1920×1080). Our experiments use two clips extracted from the short film *Tears of Steel* — © Blender Foundation (`https://mango.blender.org/`), licensed under CC BY 3.0. The 20-second clip contains 490 frames, and the 60-second clip contains 1440 frames, reflecting the frame rate of the source material. These sequences include a mixture of low-motion dialogue shots and

high-motion, effects-heavy scenes, providing a realistic and challenging testbed for our motion-aware predictive upsampling pipeline.

Each configuration was run three times, and we report the mean. GPU experiments include data transfer overheads unless explicitly stated. CPU experiments use OpenMP with dynamic scheduling to reduce load imbalance in tile-based processing. GPU kernels were compiled with `-O3` and launched using block sizes tuned per stage (128 for prediction, 256 for refinement) to ensure high occupancy.

To study scalability and behavior under different workloads, we swept over multiple knob settings:

- **Tile sizes**: $\{8 \times 8,\ 16 \times 16,\ 32 \times 32\}$
- **Subsample factors**: $\{2,\ 4,\ 8\}$
- **Problem sizes**: 20-second (490-frame) and 60-second (1440-frame) clips at 720p and 1080p

This setup enables us to analyze both quality–performance tradeoffs and hardware utilization characteristics across CPU and GPU implementations.

Compute Time Results

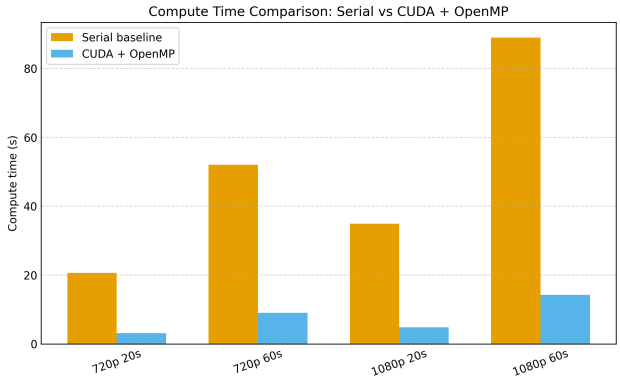


Figure 2: Compute time for the serial baseline vs. CUDA + OpenMP hybrid implementation.

Figure 2 shows that GPU acceleration reduces end-to-end compute time by a significant margin across all tested inputs. Even on shorter 20-second clips, the hybrid pipeline achieves substantial reductions in latency, and the gap widens for longer sequences.

Speedup Results

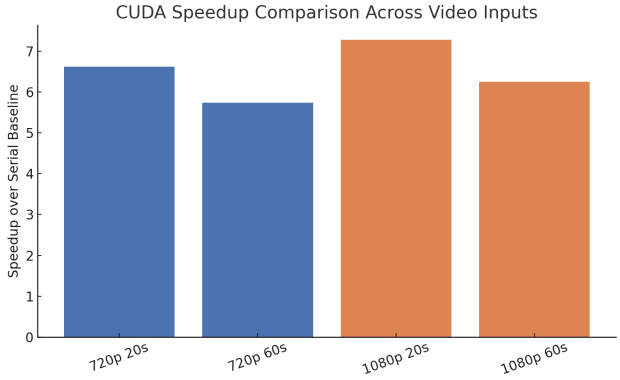


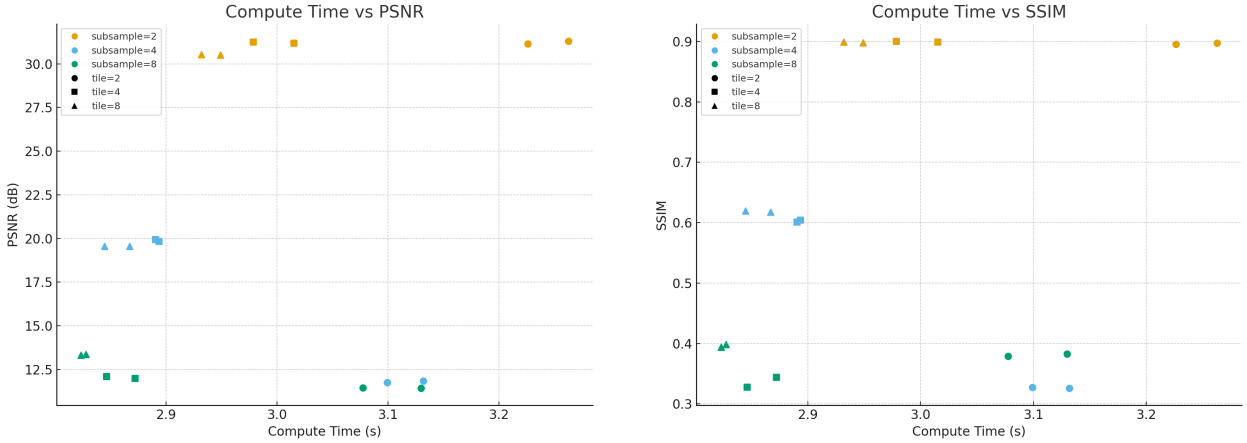
Figure 3: Speedup of CUDA + OpenMP over the serial baseline across video sizes.

Across all four video inputs, we observe speedups of:

- **6.6**× (720p, 20s)
- **5.7**× (720p, 60s)
- **7.3**× (1080p, 20s)
- **6.2**× (1080p, 60s)

The highest speedup occurs on the 1080p 20-second clip, where larger tiles and higher pixel density provide greater arithmetic intensity for GPU execution.

Quality Fidelity Results



(a) Compute Time vs PSNR

(b) Compute Time vs SSIM

Figure 4: Quality–performance tradeoff across knob configurations.
Colors indicate subsample factor and marker shapes indicate tile size.

Although increasing the subsample factor should, in theory, reduce input bandwidth proportionally, our results show that the resulting reconstructions degrade too severely to be usable. At higher subsample factors, too few pixels survive the initial sampling stage, leaving the interpolation and refinement stages without enough information to recover meaningful structure. This causes PSNR and SSIM to collapse sharply (e.g., SSIM dropping from 0.90 at subsample=2 to 0.33–0.40 at subsample=8).

Because the primary goal of this project is to reconstruct frames with high visual fidelity, rather than to maximize compression at the expense of quality, configurations with large subsample factors are not viable. Even though they would reduce bandwidth, the reconstructed frames no longer resemble the original content in a meaningful way. Therefore, we prioritize lower subsample factors—specifically subsample = 2—which still provide significant compression while preserving enough information for accurate reconstruction.

Performance Analysis

Our results show that the GPU-accelerated approach substantially outperforms the serial baseline due to:

- **Massive pixel-level parallelism:** Reconstruction and refinement operate on millions of independent pixel updates.
- **Uniform stencil structure:** The Jacobi refinement kernel maps cleanly to SIMT execution.
- **Reduced redundant work through tile masking:** Only active (moving) tiles incur the full computational cost.

Execution Time Breakdown

To better understand the performance characteristics of our heterogeneous CPU–GPU pipeline, we measured a detailed execution-time breakdown on the `frames_720` dataset under the optimal configuration (subsample = 2, tile = 4, iterations = 2). Across 490 processed frames, the CUDA-accelerated pipeline achieves an average compute-only time of **6.35 ms/frame** (3.11 s total). Table 2 summarizes the distribution of this compute time across the major pipeline components.

Stage	**Time (ms/frame)**	**% of Compute**
Subsampling	1.38	21.7%
Tile classification (CPU)	0.70	11.1%
Reconstruction (full frame)	0.076	1.2%
Reconstruction (active tiles)	2.56	40.3%
Refinement (full frame)	0.012	0.19%
Refinement (active tiles)	1.60	25.2%
Total	**6.35**	**100%**

Table 2: Per-frame compute-time breakdown for the CUDA pipeline (720p_20s, subsample 2, tile 4, 2 iterations).

These measurements reveal several important characteristics of the workload. First, nearly all computational effort is concentrated within active tiles: although only 35% of tiles are active on average, they account for approximately 65% of total compute time. Full-frame stages contribute less than 2%, confirming that tile-aware masking successfully avoids redundant work. Tile-aware reconstruction is the single largest component (40.3%), followed by tile-aware refinement (25.2%).

GPU kernel and transfer profile. Using CUDA event profiling, we further decomposed GPU-side overheads:

- Total kernel execution time: **58.95 ms** (0.12 ms/frame)
- Host → device transfers: **1319.35 ms** (2.69 ms/frame)
- Device → host transfers: **621.07 ms** (1.27 ms/frame)
- `cudaMalloc`/`cudaFree`: 0.80 ms total (negligible)

Although the GPU kernels execute extremely efficiently (50–70 μs per launch), PCIe transfers are roughly **30× more costly than kernel execution**, dominating the GPU execution profile. Combined H2D and D2H transfers account for **over 60% of compute time**, making data movement the primary bottleneck in the heterogeneous design rather than raw computation.

Bottleneck Analysis

This detailed breakdown highlights several factors that fundamentally limit the achievable speedup:

- **PCIe transfer overheads (primary bottleneck):** Even with persistent device buffers, each frame requires updating masks and frame data. These transfers scale with resolution and dominate total compute time.
- **Memory-bound stencil operations:** Bilinear interpolation and 3×3 Jacobi smoothing both have very low arithmetic intensity. Their performance saturates global memory bandwidth long before compute throughput, limiting GPU utilization.
- **Tile-classification cost on the CPU:** While inexpensive in absolute terms, CPU-side SAD classification still accounts for 11% of compute time. GPU offloading was tested but increased runtime by 47% due to kernel-launch overheads and additional transfers.
- **Limited overlap between CPU and GPU work:** Classification must finish before reconstruction begins, and D2H transfers complete before tile classification of the next frame. These serialization points reduce pipeline throughput.

Overall, these results show that the performance of our system is not computation-bound but **data-movement-bound**. Despite extremely efficient GPU kernels, PCIe bandwidth and memory-intensive stencil operations constrain speedup to approximately 7–8$\times$ over the serial baseline, consistent with the measured bottleneck distribution.

Appropriateness of Hardware

The GPU proved highly effective for this workload. The 2D per-pixel computations and structured memory accesses map extremely well to SIMT execution, while the CPU performed adequately for coarse-grained tasks such as tile classification. Overall, the heterogeneous design aligns naturally with the algorithmic structure, and the achieved speedups demonstrate that the chosen hardware was well matched to the problem.

References

- Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. "Image Quality Assessment: From Error Visibility to Structural Similarity (SSIM)." Available at: `https://ece.uwaterloo.ca/~z70wang/research/ssim/`.
- "Sum of Absolute Differences (SAD)." Overview and applications in computer vision and block matching. ScienceDirect Topics. Available at: `https://www.sciencedirect.com/topics/computer-science/sum-of-absolute-difference`.
- OpenCV Documentation. "Open Source Computer Vision Library." Available at: `https://docs.opencv.org/`.

Division of Work

This project was completed collaboratively by Andrew Kim and Martin Lee, with both partners contributing to system design, debugging, evaluation, and the writing of the final report. The

intended credit distribution is **50% − 50%**. A detailed breakdown of individual responsibilities is provided below.

Andrew Kim (akim2)

- Led the implementation of the sequential reconstruction pipeline, including subsampling, spatial prediction, temporal refinement, and iterative smoothing.
- Designed and tuned the OpenMP-based CPU parallelization strategy.
- Implemented CPU-side motion/change detection and active-tile classification.
- Developed PSNR/SSIM evaluation metrics and created benchmarking and visualization scripts.
- Contributed to the design and integration of the heterogeneous CPU–GPU pipeline.

Martin Lee (hyungwol)

- Led the CUDA implementation of spatial prediction, temporal refinement, and stencil-based kernels.
- Implemented GPU-side active-tile reconstruction and the associated tile-mask workflows.
- Performed GPU optimization, including shared memory usage, memory coalescing, and tile/block geometry tuning.
- Built support for overlapping CPU motion detection with GPU reconstruction.
- Conducted large-scale performance experiments on PSC GPU nodes and integrated results into the final evaluation.

A Additional Experiment Tables

dataset	mode	subsample	tile	iterations	compute_sec	compute_ms_per_frame
frames_1080	baseline	2	4	2	34.856	72.617
frames_1080	cuda	2	4	2	4.789	9.977
frames_1080_60s	baseline	2	4	2	88.917	61.748
frames_1080_60s	cuda	2	4	2	14.232	9.883
frames_720	baseline	2	4	2	20.590	42.020
frames_720	cuda	2	4	2	3.112	6.351
frames_720_60s	baseline	2	4	2	52.011	36.144
frames_720_60s	cuda	2	4	2	9.056	6.294

Table 3: End-to-end compute time for baseline and CUDA implementations across datasets.

config_tag	subsample	tile_size	iterations	compute_time_s	avg_psnr_dB	avg_ssim	avg_mse
s2_t2_sad1.0_it2	2	2	2	3.263	31.296	0.8976	48.251
s2_t2_sad1.0_it3	2	2	3	3.226	31.151	0.8955	49.891
s2_t4_sad1.0_it2	2	4	2	2.978	31.260	0.9006	48.649
s2_t4_sad1.0_it3	2	4	3	3.015	31.195	0.8995	49.380
s2_t8_sad1.0_it2	2	8	2	2.932	30.535	0.8992	57.484
s2_t8_sad1.0_it3	2	8	3	2.949	30.526	0.8984	57.615
s4_t2_sad1.0_it2	4	2	2	3.132	11.830	0.3262	4266.190
s4_t2_sad1.0_it3	4	2	3	3.099	11.761	0.3275	4334.830
s4_t4_sad1.0_it2	4	4	2	2.890	19.941	0.6015	659.147
s4_t4_sad1.0_it3	4	4	3	2.893	19.839	0.6046	674.773
s4_t8_sad1.0_it2	4	8	2	2.867	19.562	0.6175	719.281
s4_t8_sad1.0_it3	4	8	3	2.845	19.548	0.6194	721.550
s8_t2_sad1.0_it2	8	2	2	3.077	11.440	0.3790	4667.810
s8_t2_sad1.0_it3	8	2	3	3.130	11.433	0.3828	4674.850
s8_t4_sad1.0_it2	8	4	2	2.846	12.101	0.3280	4008.700
s8_t4_sad1.0_it3	8	4	3	2.872	11.998	0.3444	4104.590
s8_t8_sad1.0_it2	8	8	2	2.823	13.326	0.3942	3022.910
s8_t8_sad1.0_it3	8	8	3	2.828	13.355	0.3986	3002.970

Table 4: Knob sweep over subsample factor, tile size, and iterations, with corresponding compute time and reconstruction quality metrics.