Accelerating Discrete Wavelet Transform

Devashish Gupta, Parima Mehta, Rakesh Mugaludi

PID8: Final Project Presentation

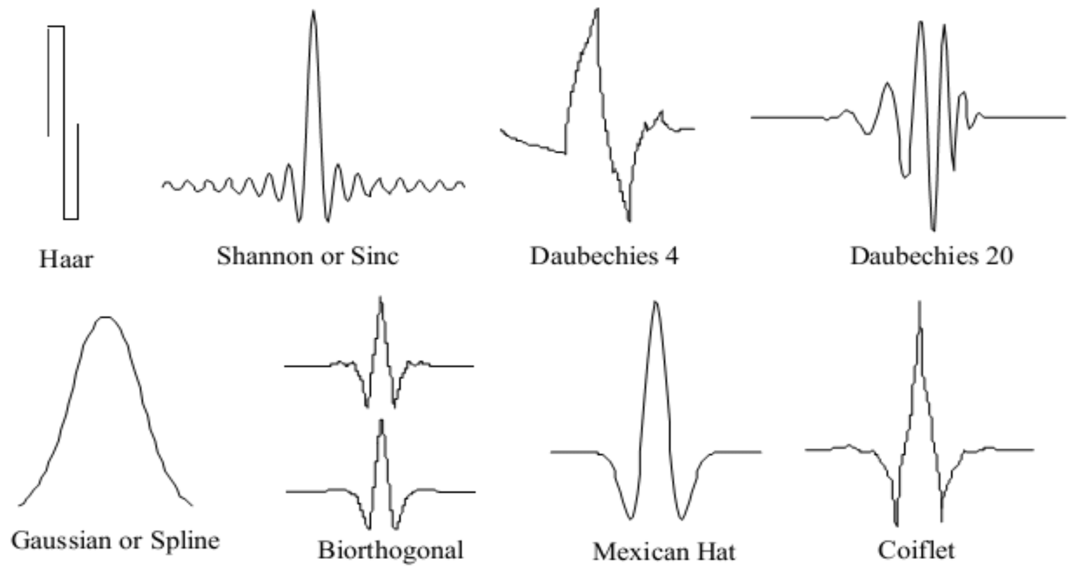
Project Category: Application

CSE 6230: High Performance Parallel Computing

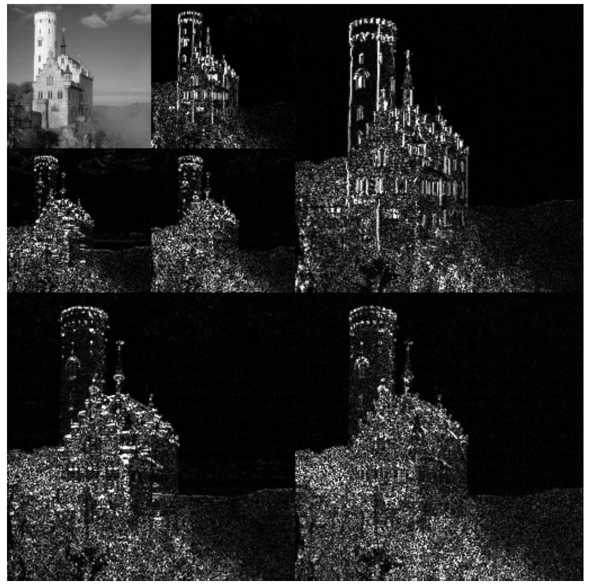
Georgia Tech

Introduction

- Discrete Wavelet Transform (DWT) is a powerful mathematical tool in signal processing and multiresolution analysis
- It is a generalization of the Fast Fourier Transform that allows capturing global and local features of the input data



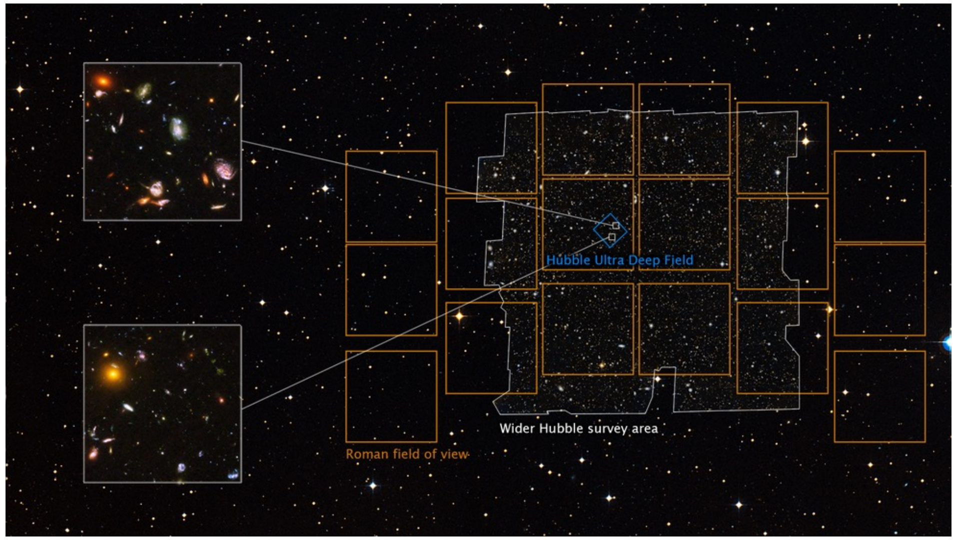
Widely used 1D mother wavelets



Multi-level 2-D DWT

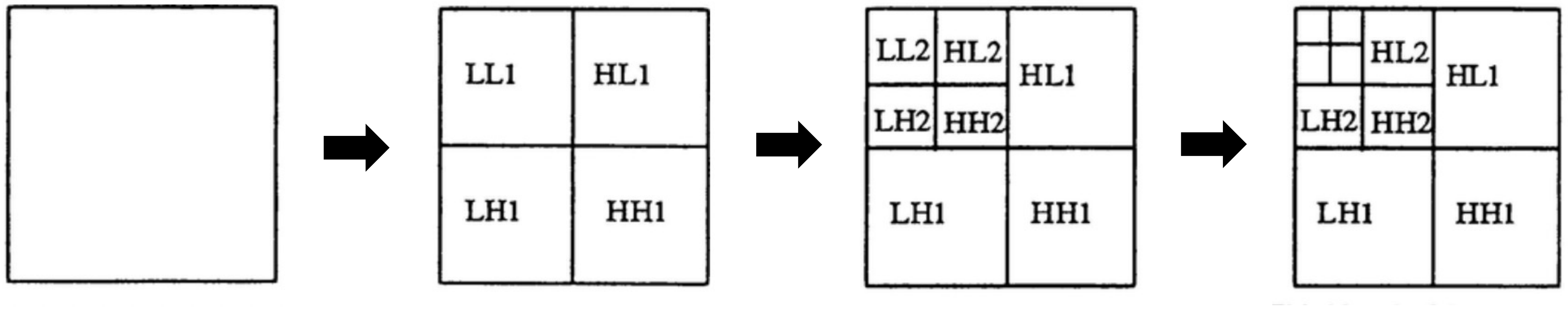
Motivation

- Diverse applications:
  - Image compression and denoising
  - Gravitational wave detection and analysis
  - EEG & ECG signal analysis
  - Feature extraction for ML
  - Multiresolution analysis of financial signals
  - Seismic data analysis, earthquake prediction



- Improvements in **time to solution** would benefit processing large images. Ex. James Webb Space Telescope generates 12.6GB of raw image data/hour.

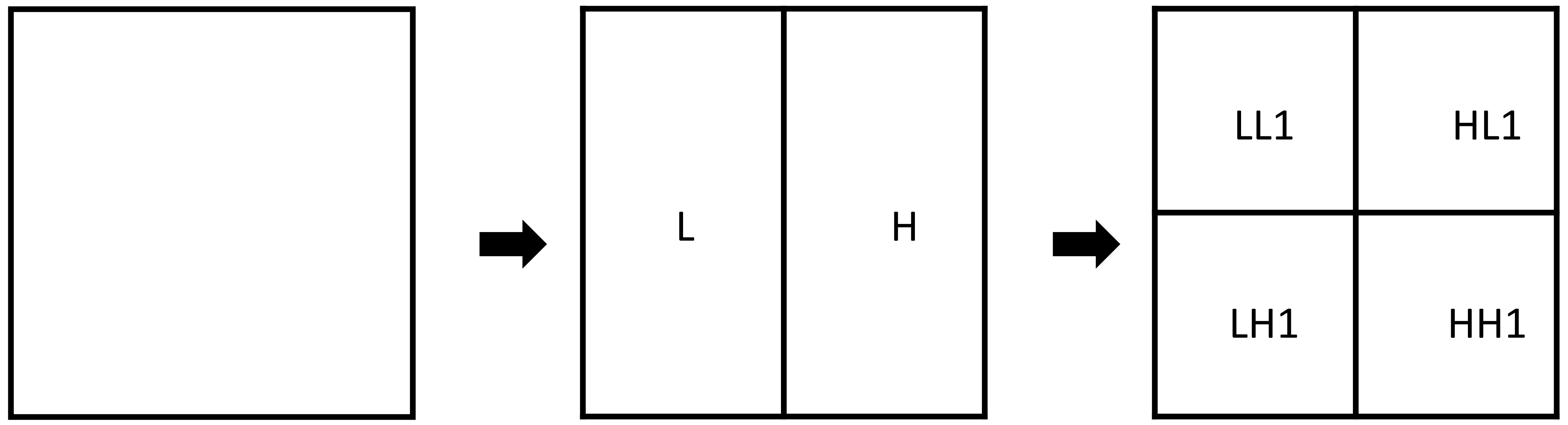
Problem: Multi-level 2-D DWT



Original Image　Decomposition Level 1　Decomposition Level 2　Decomposition Level 3

- Multi-level 2-D DWT is recursive
- At each level, image gets decomposed into low and high frequency bands
  - LL: Approximate
  - HL: Vertical
  - LH: Horizontal
  - HH: Diagonal

2-D Haar Row-Column DWT



Original Image (I)

n x n

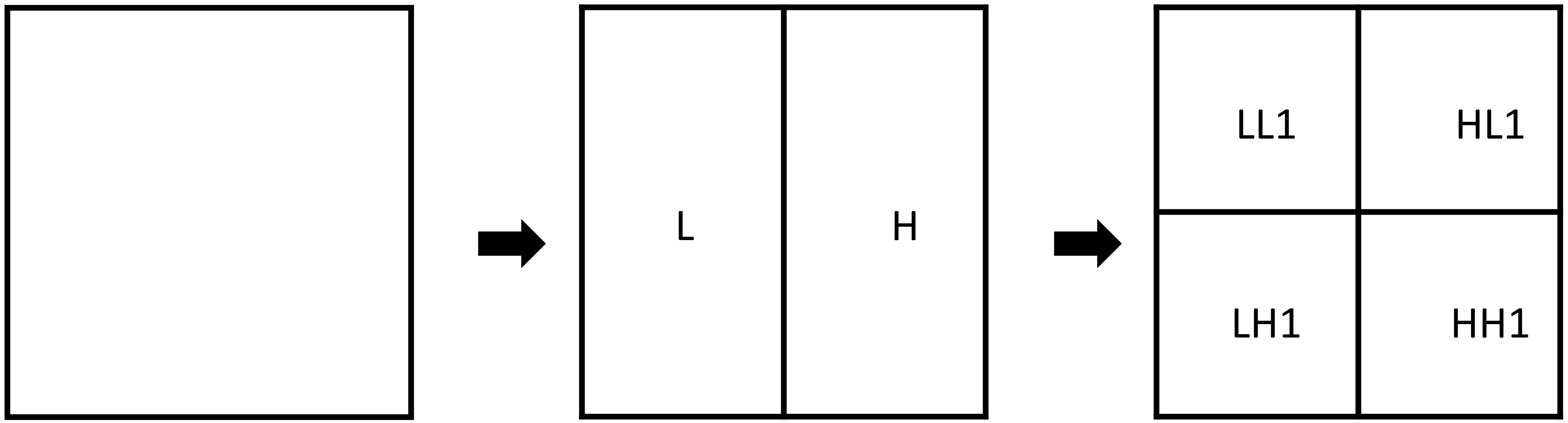
Row-wise 1-D DWT (R)

```
for i in [0, n):
    for j in [0, n/2):
        $L = \frac{I_{(i, 2j)} + I_{(i, 2j + 1)}}{\sqrt{2}}$
        $H = \frac{I_{(i, 2j)} - I_{(i, 2j + 1)}}{\sqrt{2}}$
```

Column-wise 1-D DWT (C)

```
for j in [0, n):
    for i in [0, n/2):
        $L\{L, H\} = \frac{R_{(2i, j)} + R_{(2i + 1, j)}}{\sqrt{2}}$
        $\{L, H\}H = \frac{R_{(2i, j)} - R_{(2i + 1, j)}}{\sqrt{2}}$
```

2-D Haar Row-Column DWT



Original Image (I)

n x n

Row-wise 1-D DWT (R)

Column-wise 1-D DWT (C)

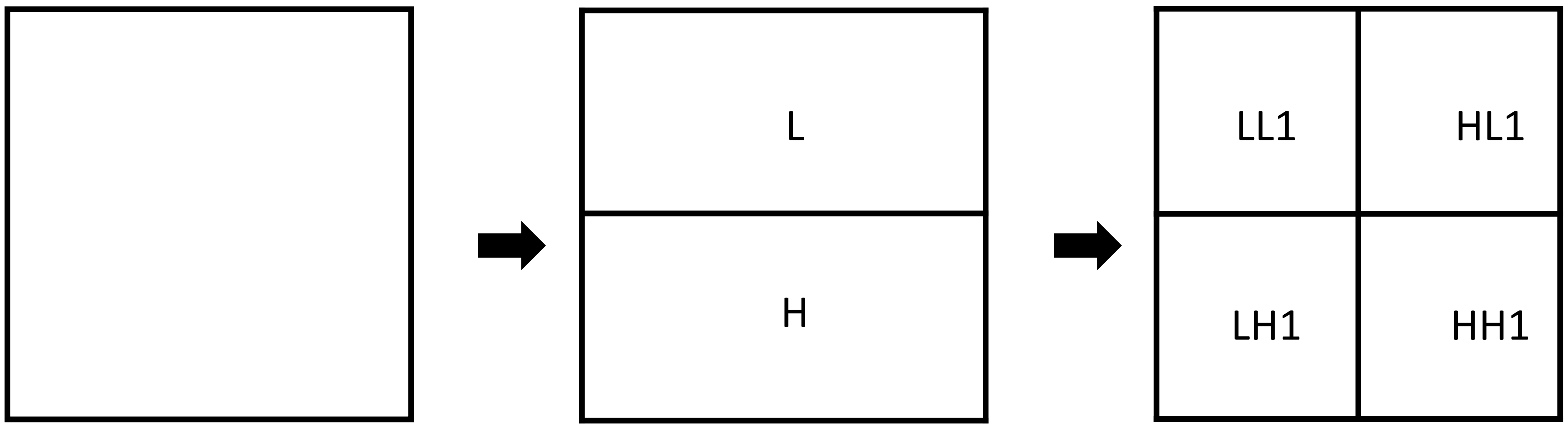
```
for i in [0, n):
    for j in [0, n/2):
        $L = \frac{I_{(i, 2j)} + I_{(i, 2j+1)}}{\sqrt{2}}$
        $H = \frac{I_{(i, 2j)} - I_{(i, 2j+1)}}{\sqrt{2}}$
```

```
for j in [0, n):
    for i in [0, n/2):
        $L\{L, H\} = \frac{R_{(2i, j)} + R_{(2i+1, j)}}{\sqrt{2}}$
        $\{L, H\}H = \frac{R_{(2i, j)} - R_{(2i+1, j)}}{\sqrt{2}}$
```

Issues?

Column-wise transform is expensive for row-major layout of image in memory

Transposition



Original Image (I)

n x n

Row-wise 1-D DWT (R)

```
for i in [0, n):
    for j in [0, n/2):
        $L = \frac{I_{(i,\ 2j)} + I_{(i,\ 2j+1)}}{\sqrt{2}}$
        $H = \frac{I_{(i,\ 2j)} - I_{(i,\ 2j+1)}}{\sqrt{2}}$
```

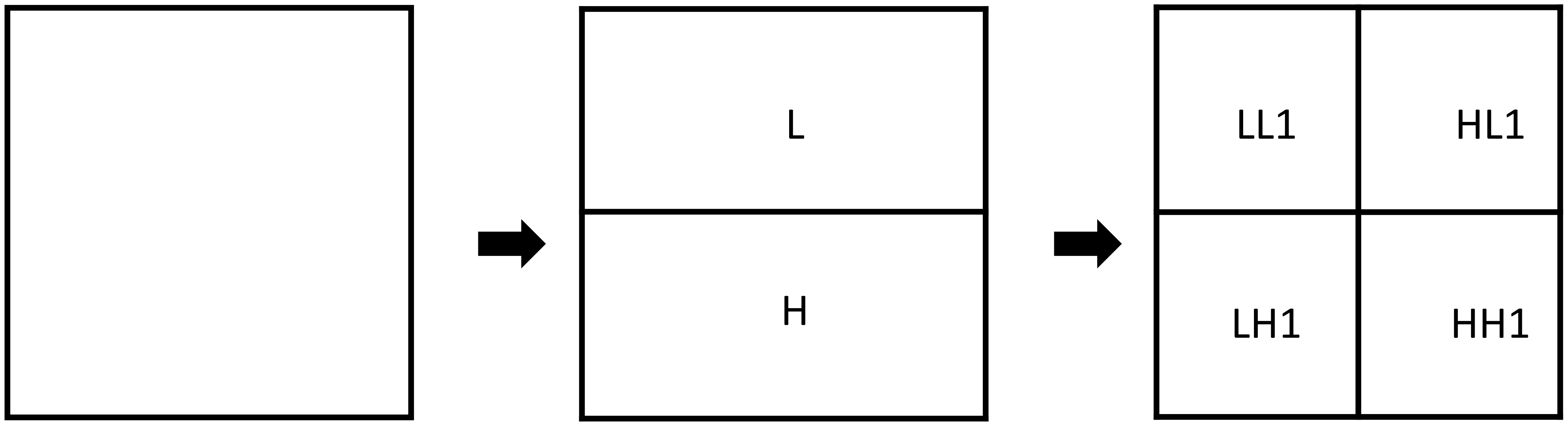
Row-wise 1-D DWT (C)

```
for i in [0, n):
    for j in [0, n/2):
        $L\{L, H\} = \frac{R_{(i,\ 2j)} + R_{(i,\ 2j+1)}}{\sqrt{2}}$
        $\{L, H\}H = \frac{R_{(i,\ 2j)} - R_{(i,\ 2j+1)}}{\sqrt{2}}$
```

Solution I:

Replace column-wise transform with row-wise transform via transposition

Transposition



Original Image (I)

n x n

Row-wise 1-D DWT (R)

```
for i in [0, n):
    for j in [0, n/2):
        $L = \frac{I_{(i,\,2j)} + I_{(i,\,2j+1)}}{\sqrt{2}}$
        $H = \frac{I_{(i,\,2j)} - I_{(i,\,2j+1)}}{\sqrt{2}}$
```

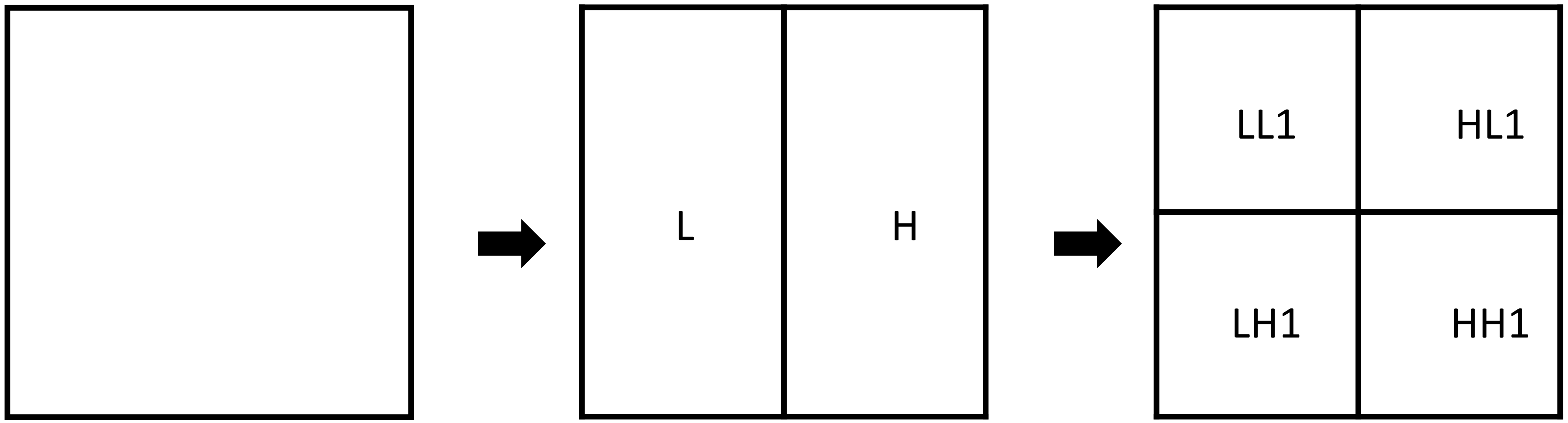
Row-wise 1-D DWT (C)

```
for i in [0, n):
    for j in [0, n/2):
        $L\{L, H\} = \frac{R_{(i,\,2j)} + R_{(i,\,2j+1)}}{\sqrt{2}}$
        $\{L, H\}H = \frac{R_{(i,\,2j)} - R_{(i,\,2j+1)}}{\sqrt{2}}$
```

Issues?

Expensive for large images

Loop Reordering



Original Image (I)

n x n

Row-wise 1-D DWT (R)

Column-wise 1-D DWT (C)

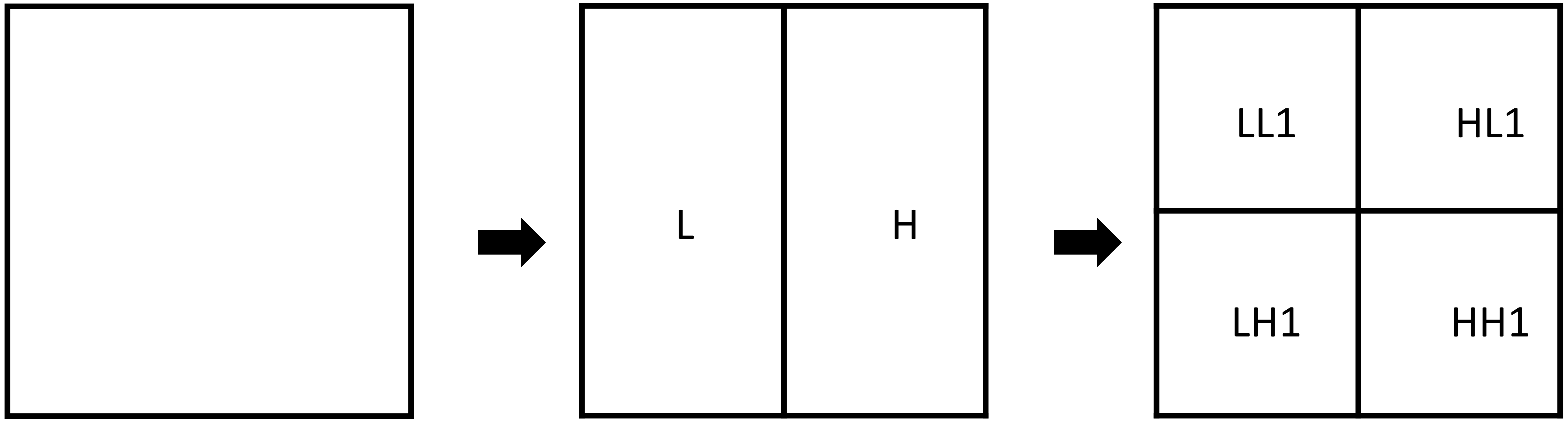
```
for i in [0, n):
    for j in [0, n/2):
        $L = \frac{I_{(i, 2j)} + I_{(i, 2j + 1)}}{\sqrt{2}}$
        $H = \frac{I_{(i, 2j)} - I_{(i, 2j + 1)}}{\sqrt{2}}$
```

```
for i in [0, n/2):
    for j in [0, n):
        $L\{L, H\} = \frac{R_{(2i, j)} + R_{(2i + 1, j)}}{\sqrt{2}}$
        $\{L, H\}H = \frac{R_{(2i, j)} - R_{(2i + 1, j)}}{\sqrt{2}}$
```

Solution II:

Loop-reordering exploits row-major layout; effectively lowering cache miss rate

Loop Reordering



Original Image (I)

n x n

Row-wise 1-D DWT (R)

Column-wise 1-D DWT (C)

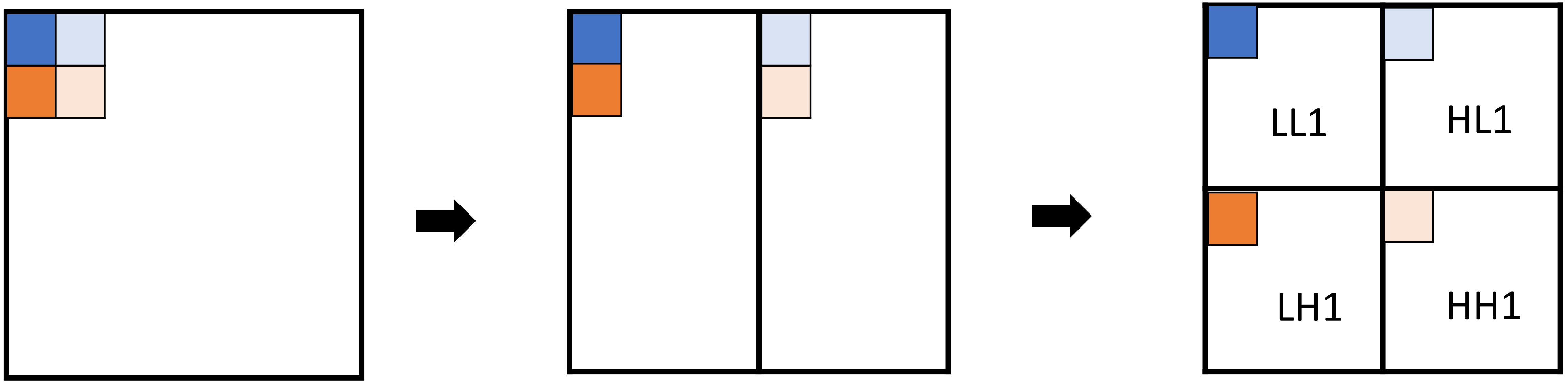
```
for i in [0, n):
    for j in [0, n/2):
        $L = \frac{I_{(i,\ 2j)} + I_{(i,\ 2j+1)}}{\sqrt{2}}$
        $H = \frac{I_{(i,\ 2j)} - I_{(i,\ 2j+1)}}{\sqrt{2}}$
```

```
for i in [0, n/2):
    for j in [0, n):
        $L\{L, H\} = \frac{R_{(2i,\ j)} + R_{(2i+1,\ j)}}{\sqrt{2}}$
        $\{L, H\}H = \frac{R_{(2i,\ j)} - R_{(2i+1,\ j)}}{\sqrt{2}}$
```

Issues?

Entire image undergoes row-wise transform followed by column-wise transform; inefficient data reuse

2-D Haar Block-Based DWT



Original Image (I)
n x n

Intermediate row-transform

2-D Block-based DWT

```
for i in [0, n/2):
    for j in [0, n/2):
```

$$LL1 = \frac{I_{(2i,\,2j)} + I_{(2i,\,2j+1)} + I_{(2i+1,\,2j)} + I_{(2i+1,\,2j+1)}}{2}$$

$$HL1 = \frac{I_{(2i,\,2j)} - I_{(2i,\,2j+1)} + I_{(2i+1,\,2j)} - I_{(2i+1,\,2j+1)}}{2}$$

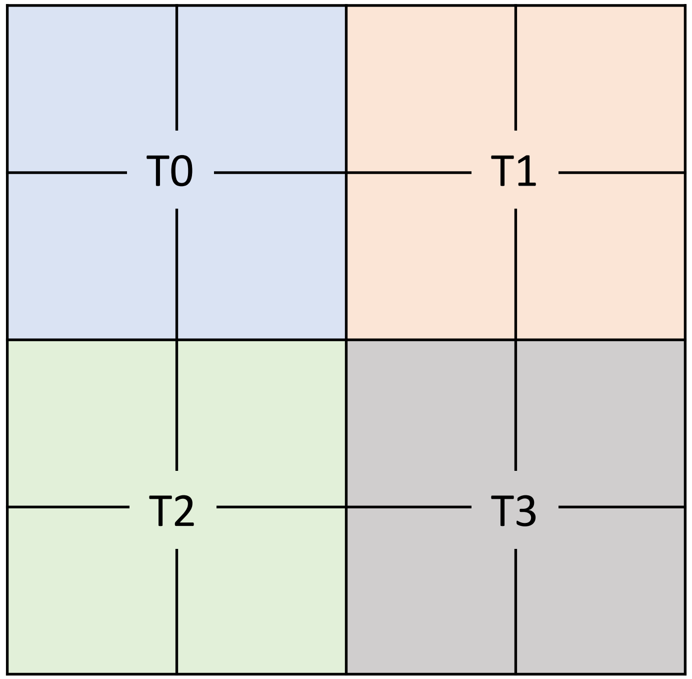
$$LH1 = \frac{(I_{(2i,\,2j)} + I_{(2i,\,2j+1)}) - (I_{(2i+1,\,2j)} + I_{(2i+1,\,2j+1)})}{2}$$

$$HH1 = \frac{(I_{(2i,\,2j)} - I_{(2i,\,2j+1)}) - (I_{(2i+1,\,2j)} - I_{(2i+1,\,2j+1)})}{2}$$

Solution:

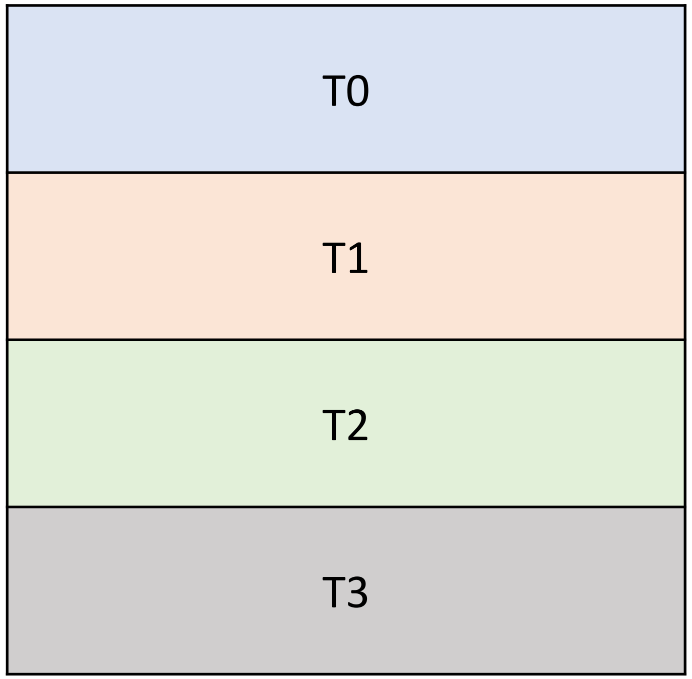
- Perform row AND column transform on a 2x2 block of input image
- Allows efficient data reuse as image is read/written only once

OpenMP Realization



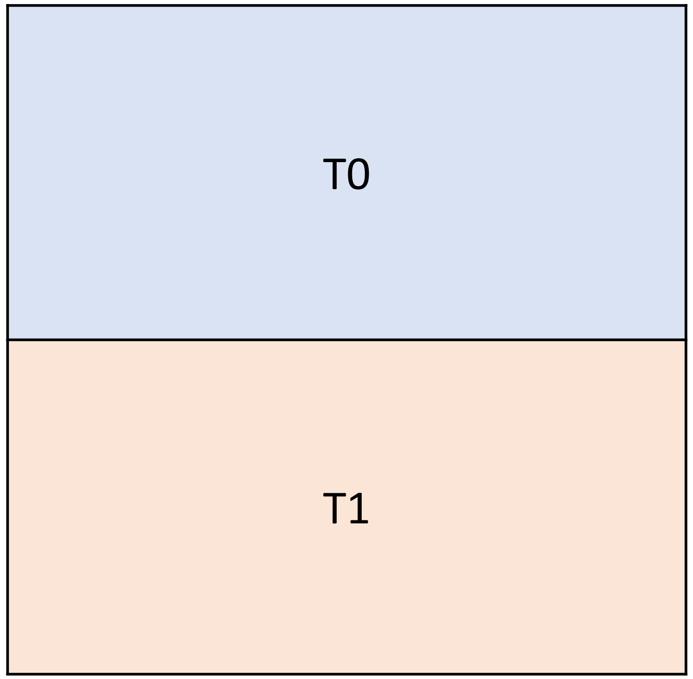
Tiled Transposed Variant

- 1 Tile/thread
- Tile size determined based on cacheline size to avoid false-sharing



Loop Re-ordered Variant

- 1 Row/thread
- Exploits row-major layout

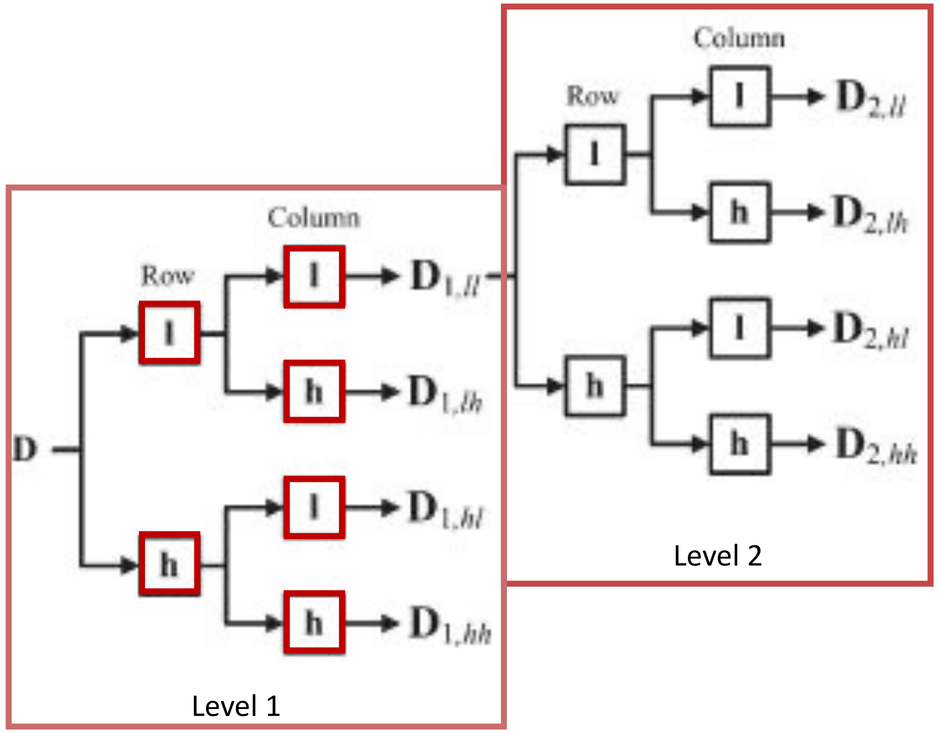


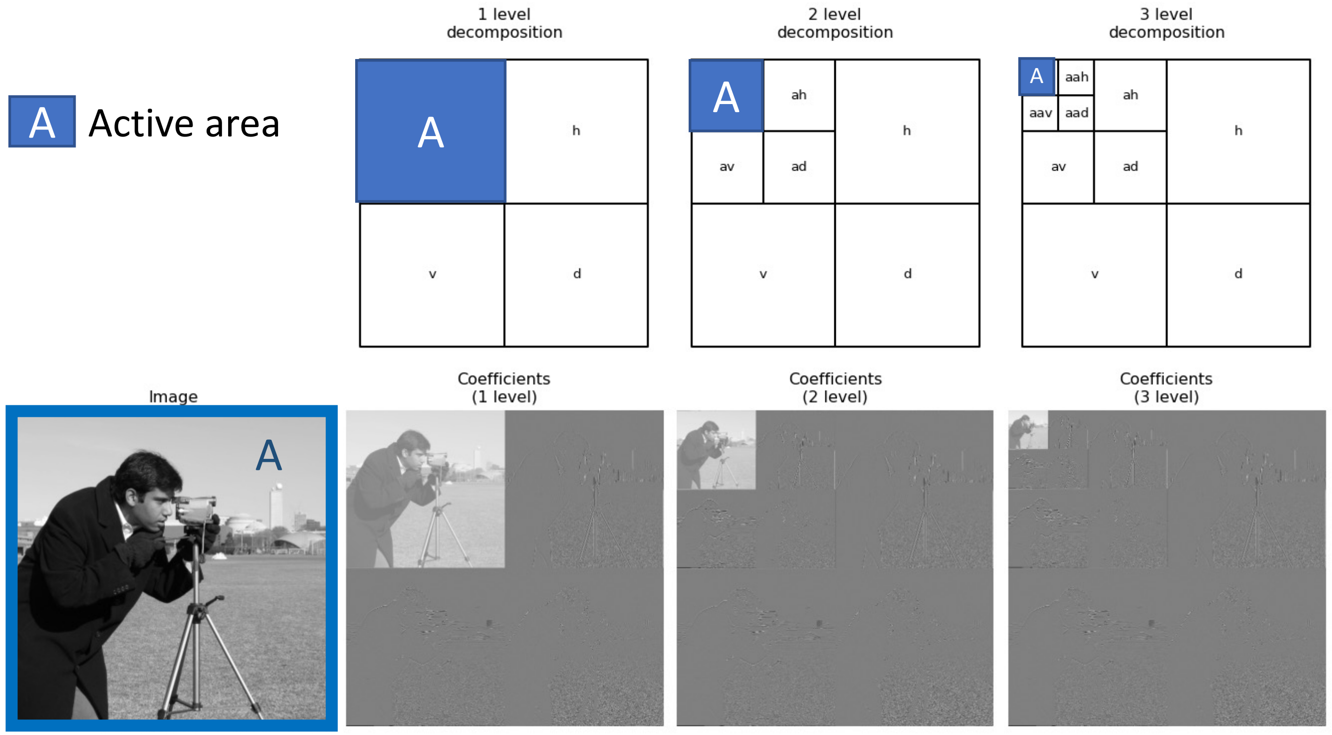
Block-based Variant

- 2 Rows/thread
- Exploits row-major layout

Parallelizing on GPU

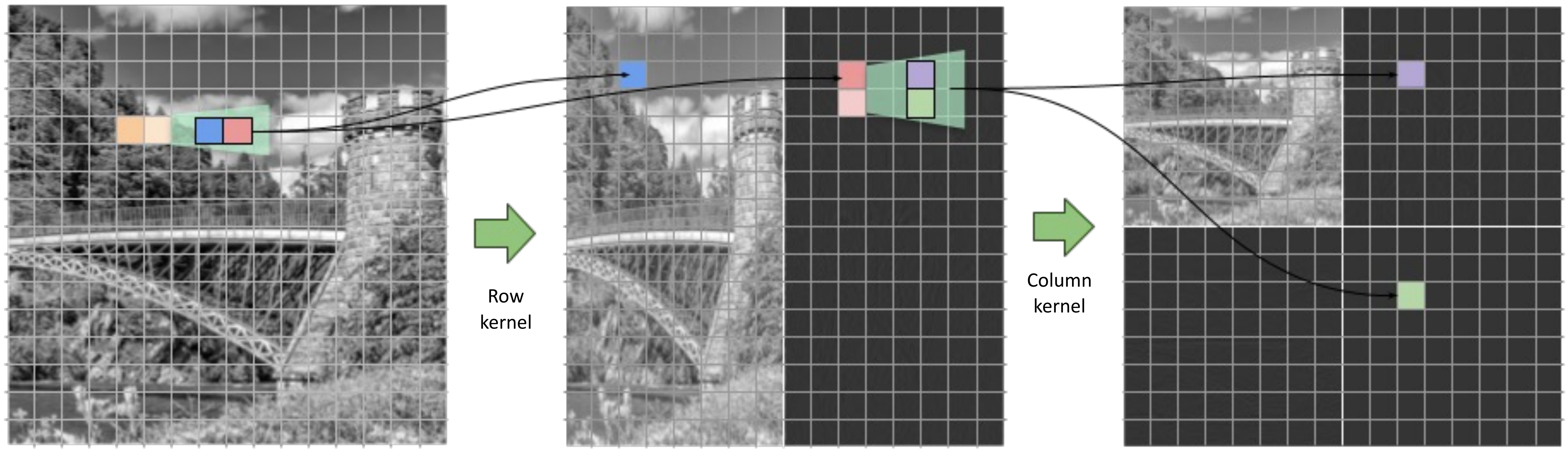
- Forward Haar-DWT:
  - Each level involves a row-wise operation and a column-wise operation
  - Every level is hierarchically dependent of the previous level
  - Active operating pixel area drops after every level





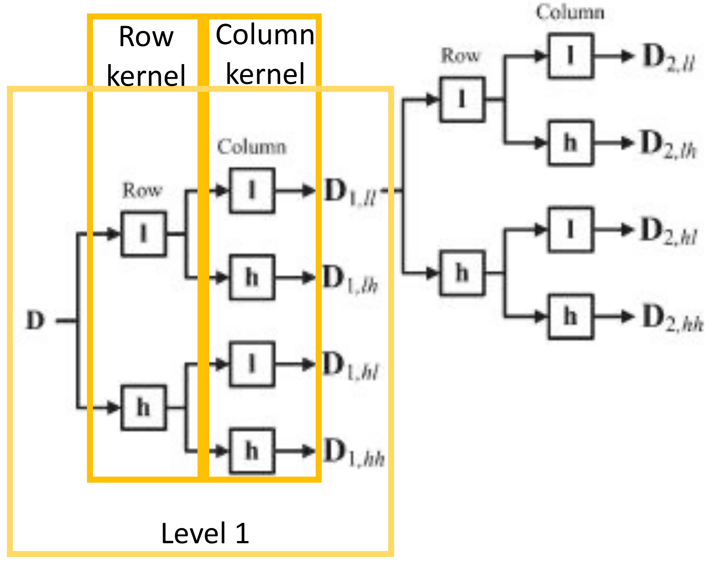
GPU Row-Column DWT

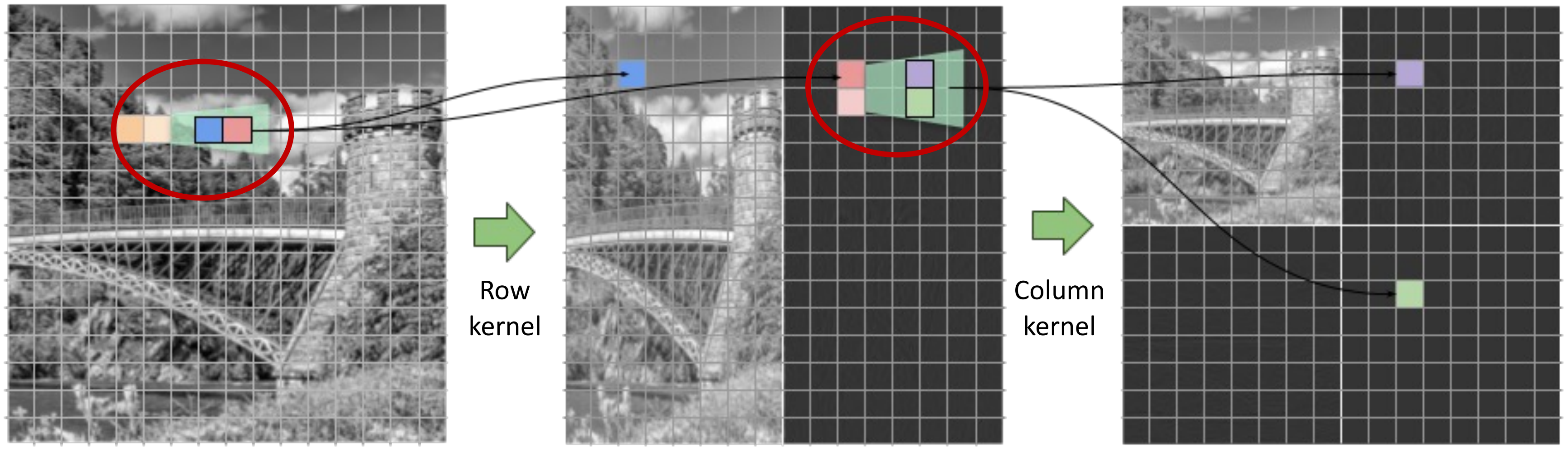
- GPU Row-Column Haar-DWT:
  - For loop indexing over image replaced with GPU threads
  - Each level involves a row pass kernel and a column pass kernel
  - Iterates kernel call with varying grid dimensions as level changes



GPU Row-Column DWT

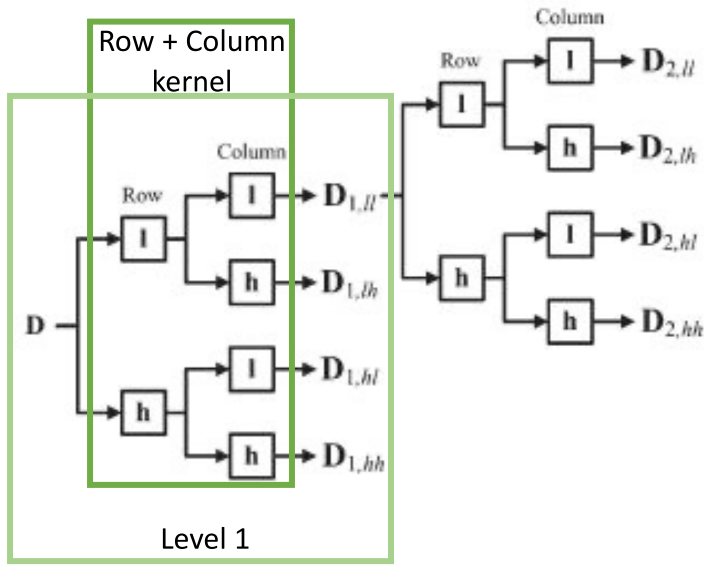
- GPU Row-Column Haar-DWT issues:
  - Excessive memory access: 4 global reads/writes per final pixel writes per thread
    - 2 full passes over image
  - Low arithmetic intensity
  - Global memory synchronization: 2 kernel passes can slow down/add overhead
  - Hierarchical dependency: single level computation still hierarchical

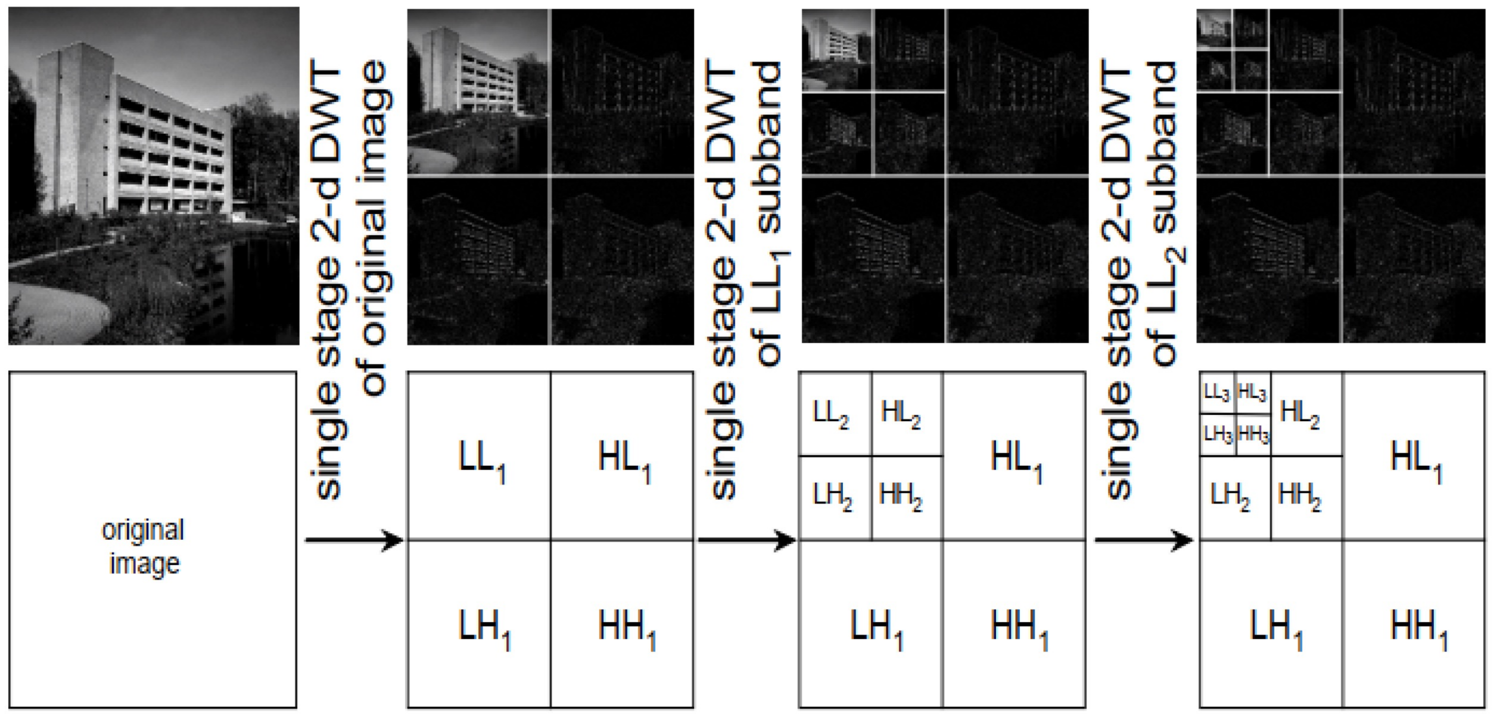




Improving GPU Row-Column DWT

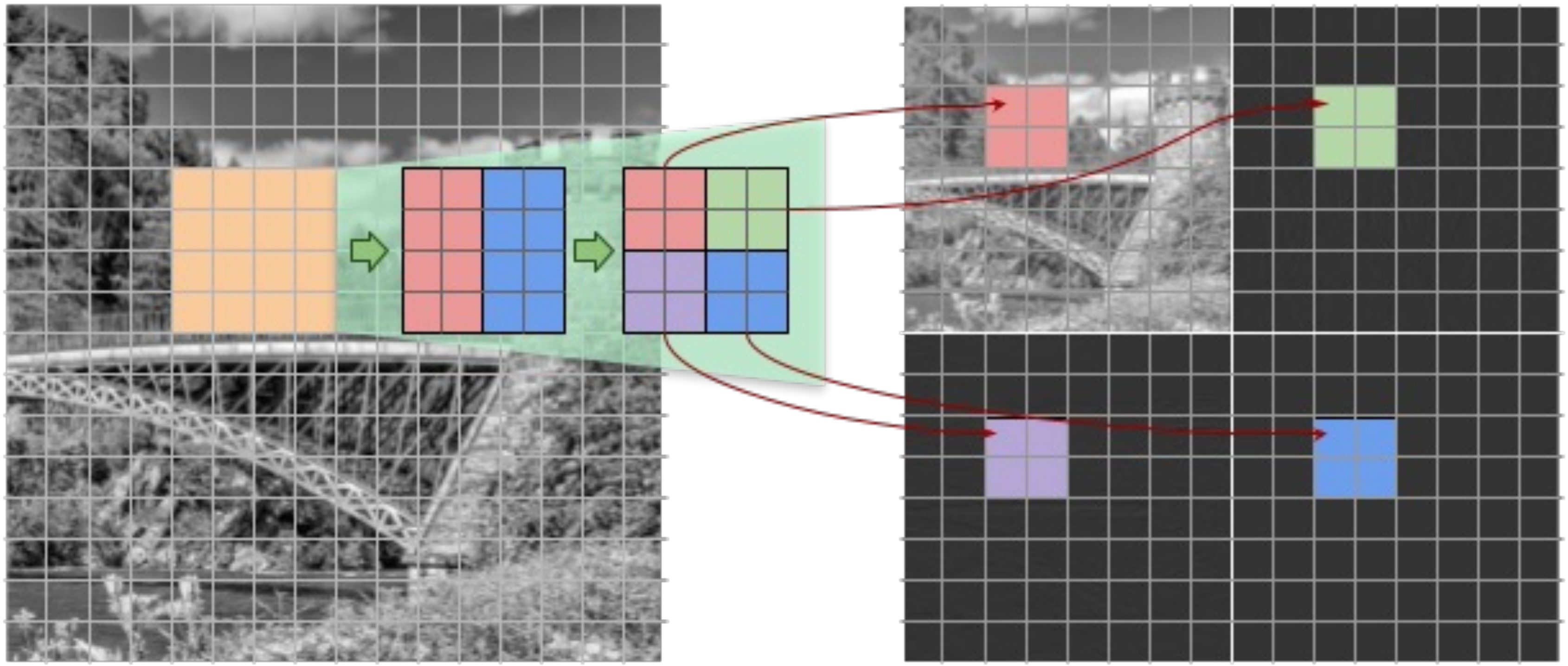
- Improving GPU Row-Column Haar-DWT further:
  - Having a single stage kernel per level to increase arithmetic intensity
  - Operating in tiles to combine row and column kernels
  - Reducing the number of global reads/writes via shared memory use




original image
single stage 2-d DWT of original image
single stage 2-d DWT of LL1 subband
single stage 2-d DWT of LL2 subband
LL1
HL1
LH1
HH1
LL2
HL2
LH2
HH2
LL3
HL3
LH3
HH3

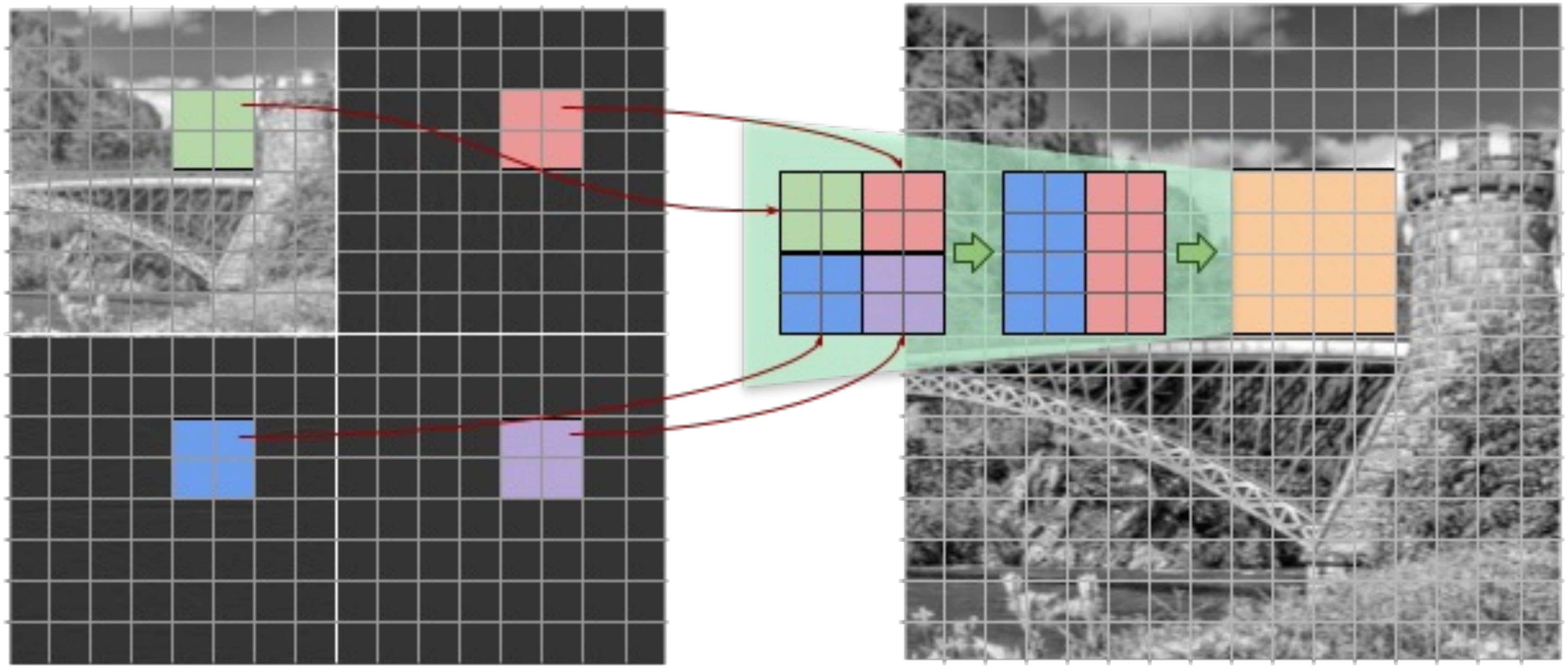
GPU Tiled smem: Forward DWT

- Tiled and shared memory forward Haar-DWT :
  - Each block (32x32) processes a tile (32x32 px) of the image independently
  - Tile loaded from the image into shared memory
  - Row and column component passes performed in shared memory
  - Wavelet decomposition written back appropriately



GPU Tiled smem: Inverse DWT

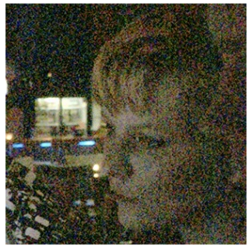
- Tiled and shared memory Inverse Haar-DWT :
  - Wavelet decomposition tile loaded into shared memory
  - A tile of the image is independently processed by a block
  - Inverse column and row component passes performed in shared memory
  - Tile written to the image from shared memory



Datasets

- We sourced images from noisy image datasets such as for low-light photography. We also sourced on deep field astronomical images to create the following benchmark images (bitmaps)

Benchmark images


256 x 256

512 x 512

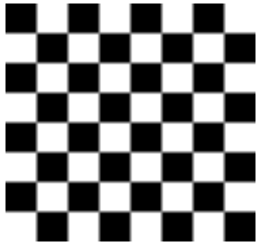
1024 x 1024

2048 x 2048

4096 x 4096

8192 x 8192

Verification/debug images


Checker

Asymmetric

High freq detail

High contrast

Testbed

- We tested our sequential and OpenMP implementations on the PACE cluster with the following resource configuration

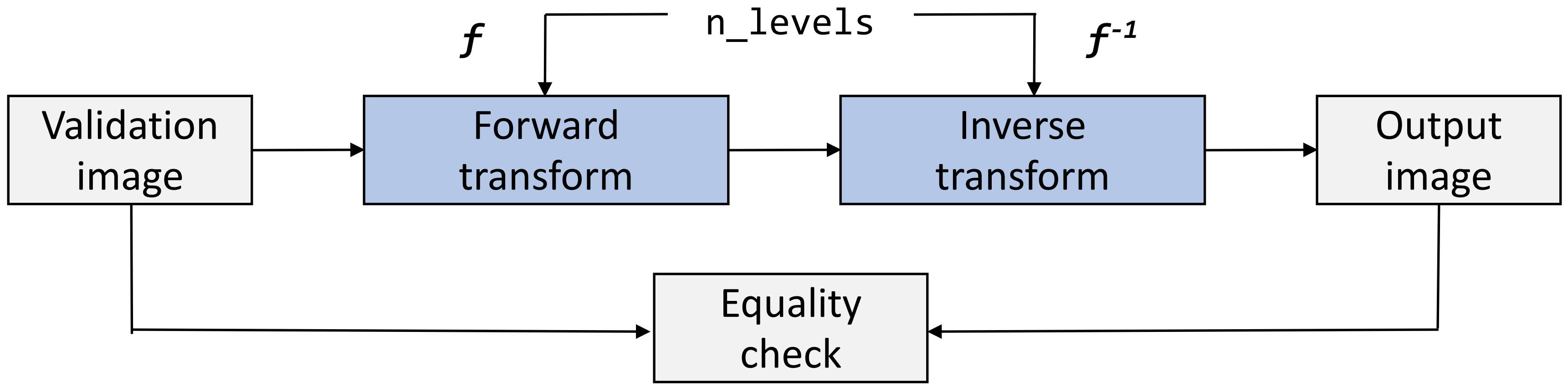
```
qsub -l walltime=02:00:00 -l nodes=1:ppn=24 -l
pmem=8gb -q coc-ice -I
```

- Accelerated implementations were tested on NVIDIA V100 GPUs with the following resource configuration

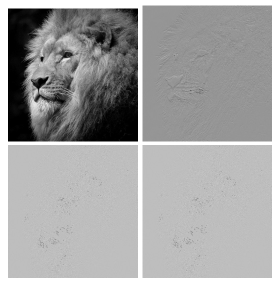
```
qsub -l walltime=02:00:00 –l nodes=1:gpus=1:teslav100
-l pmem=8gb -q coc-ice-gpu -I
```

Validation

- Validation of our algorithms was performed in two steps, by composing the forward and inverse transforms.



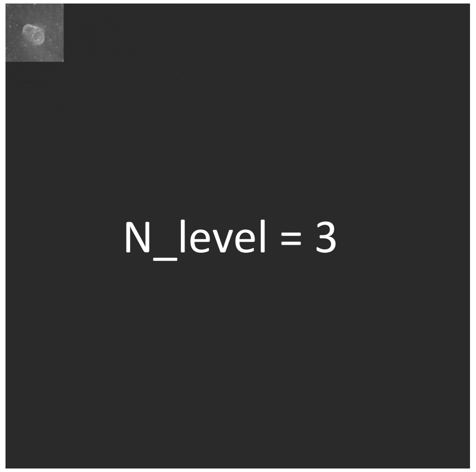
Input image

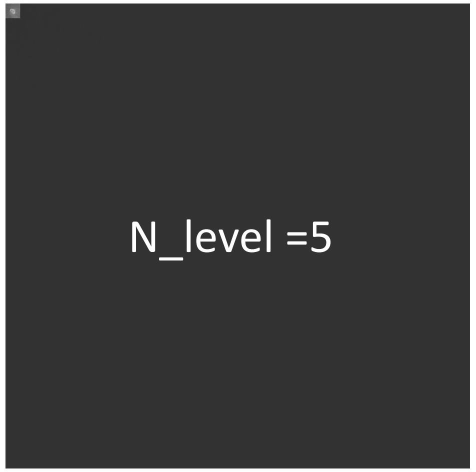

Detail coefficient image

Output image

CPU Validation

Input Image





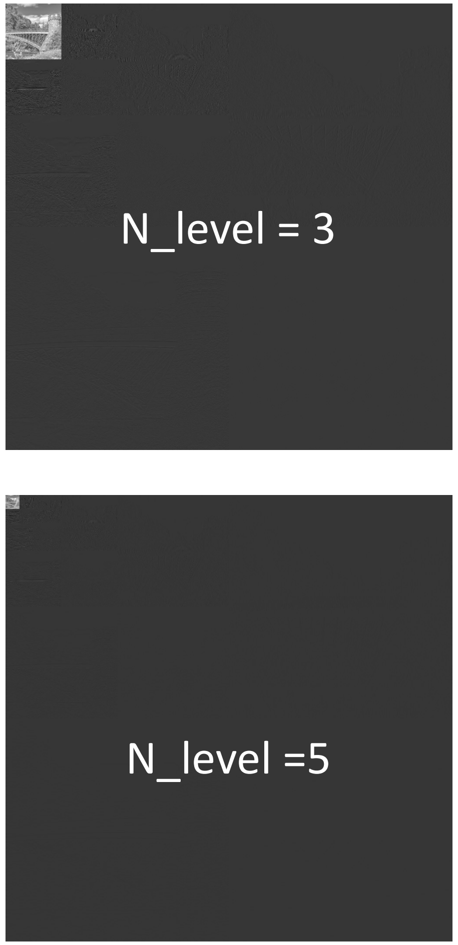
Forward Haar Transform

Reconstructed Image

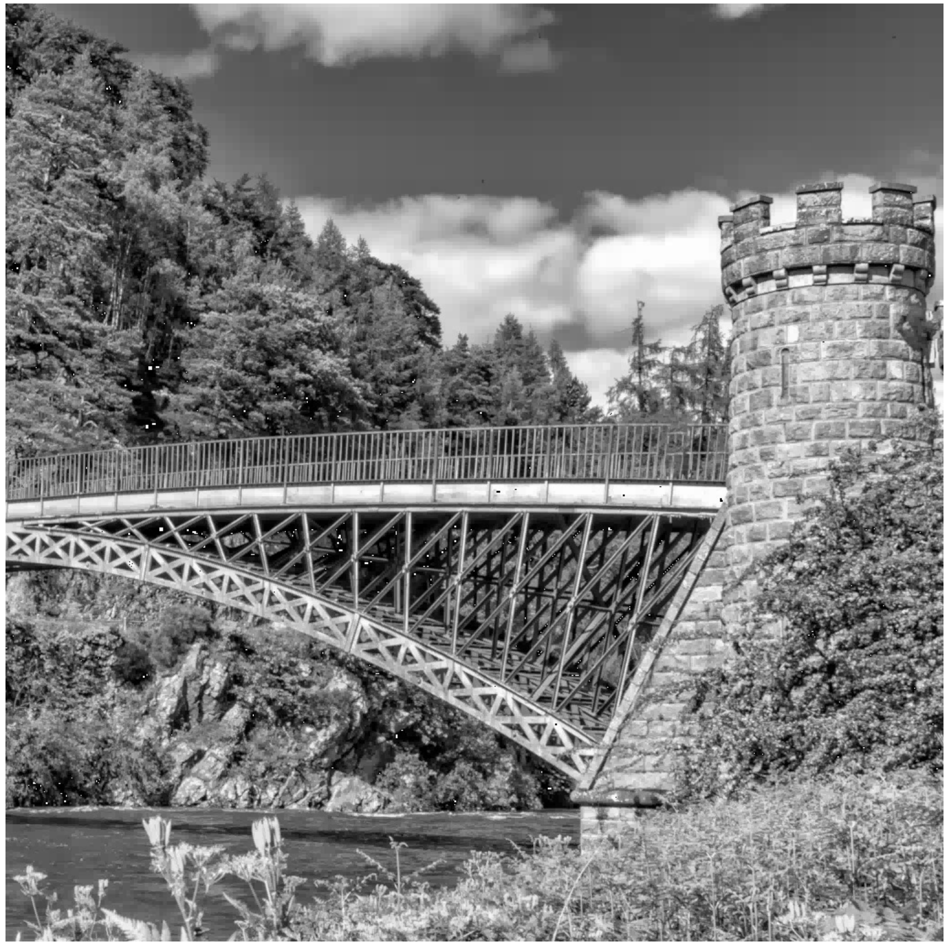
OpenMP implementation validation for image size: 4096 x 4096

GPU Validation

Input Image



Forward Haar Transform


Reconstructed Image

GPU implementation validation for image size: 1024 x 1024

Performance Metric & Baseline

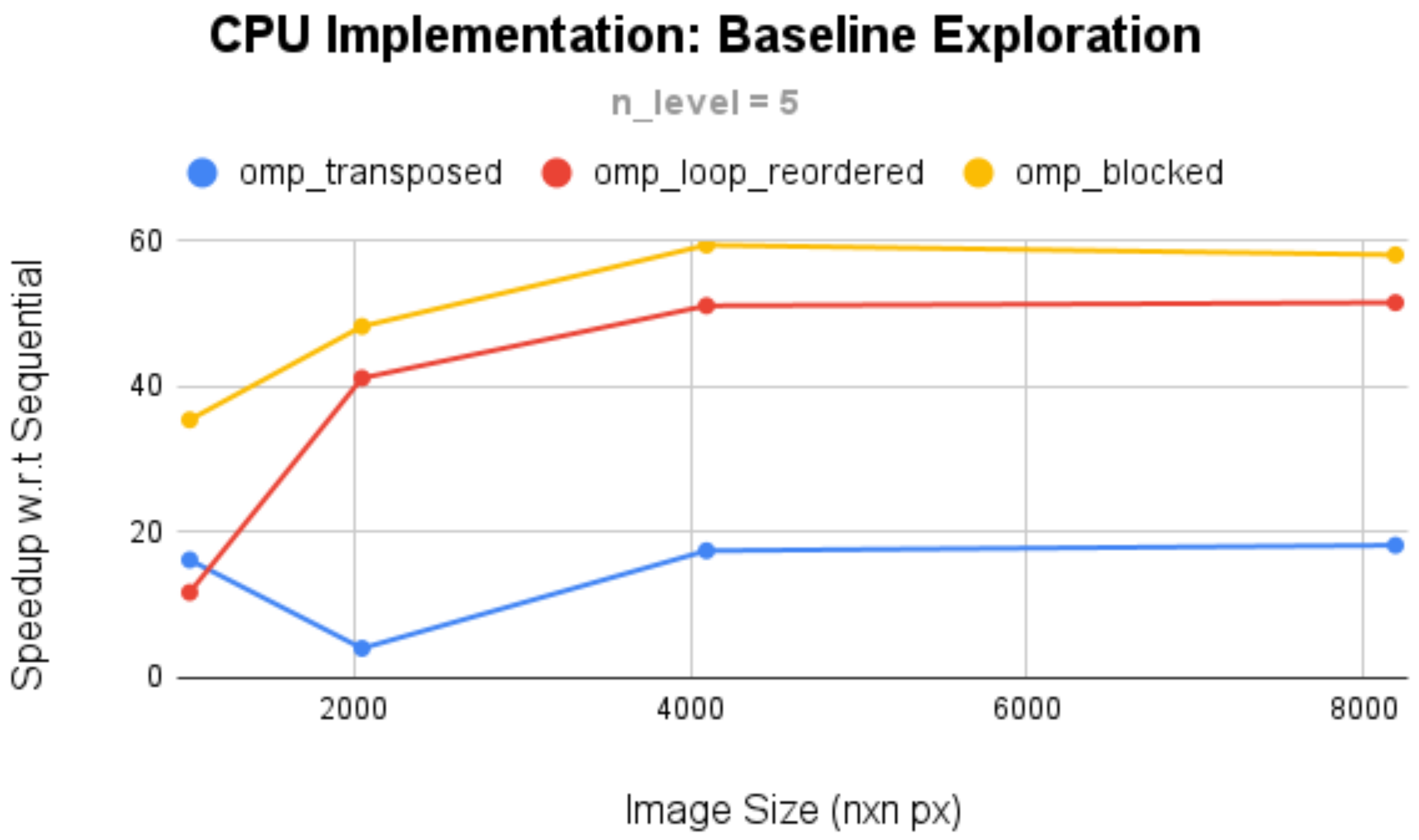
- To evaluate the performance of our algorithms, we adopted **time to solution (ms)** for 2-D forward Haar transform as our metric
- **Baseline**:
  - CPU: Sequential implementation
  - GPU: Shared memory OpenMP implementation (with 24 cores)
- Four versions of the CPU implementation were tested:

`sequential, openmp_loop_reordered, openmp_blocked, openmp_transposed`

- Two versions of the GPU implementation were tested:

`gpu_row_column, gpu_tiled_shared_mem`

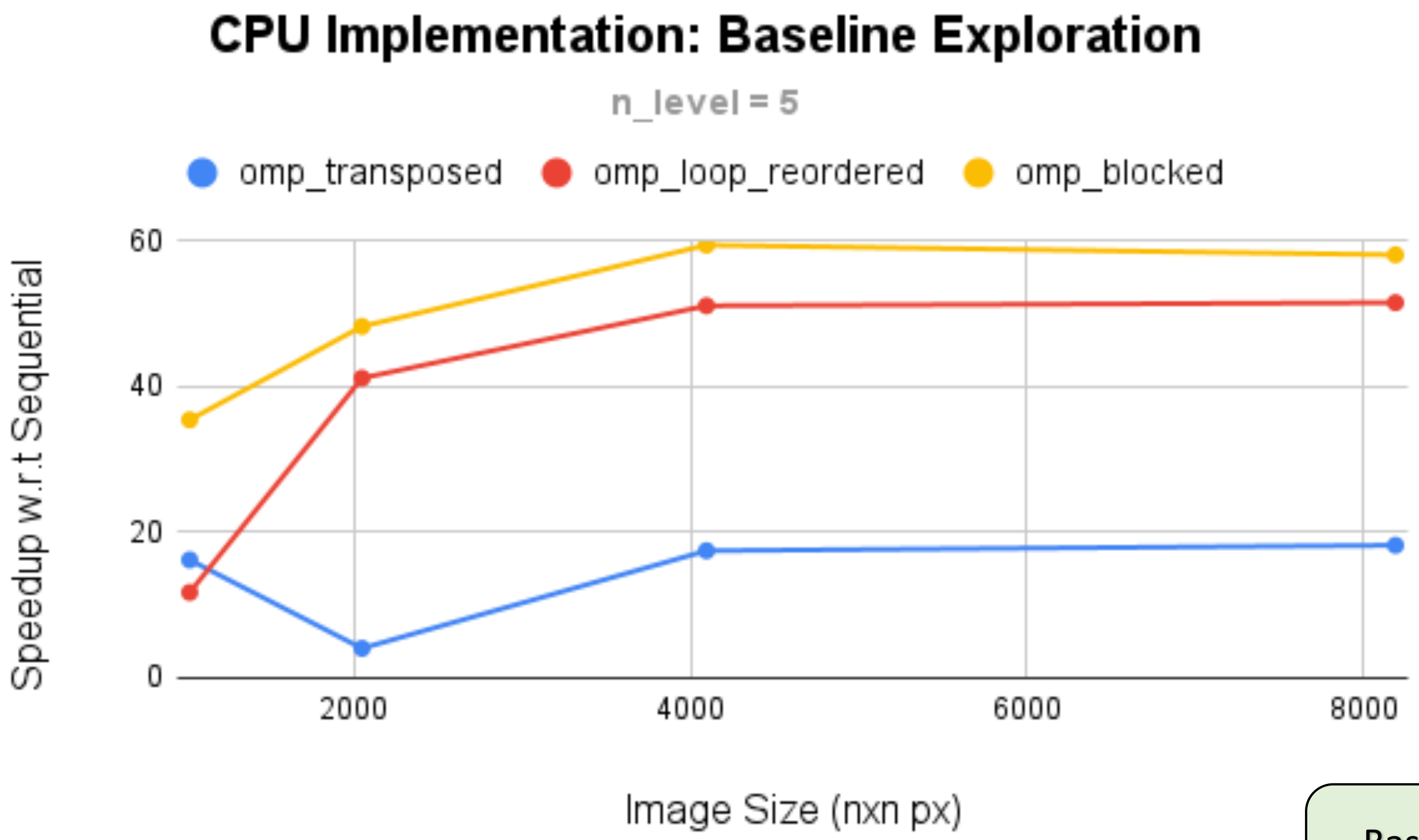
CPU Baseline Exploration



Insight

- Transposition is better than loop reordering for smaller images
- For larger images, loop reordering and blocking give better speedup
- **Block-based approach** is the most efficient, irrespective of image size

CPU Baseline Exploration

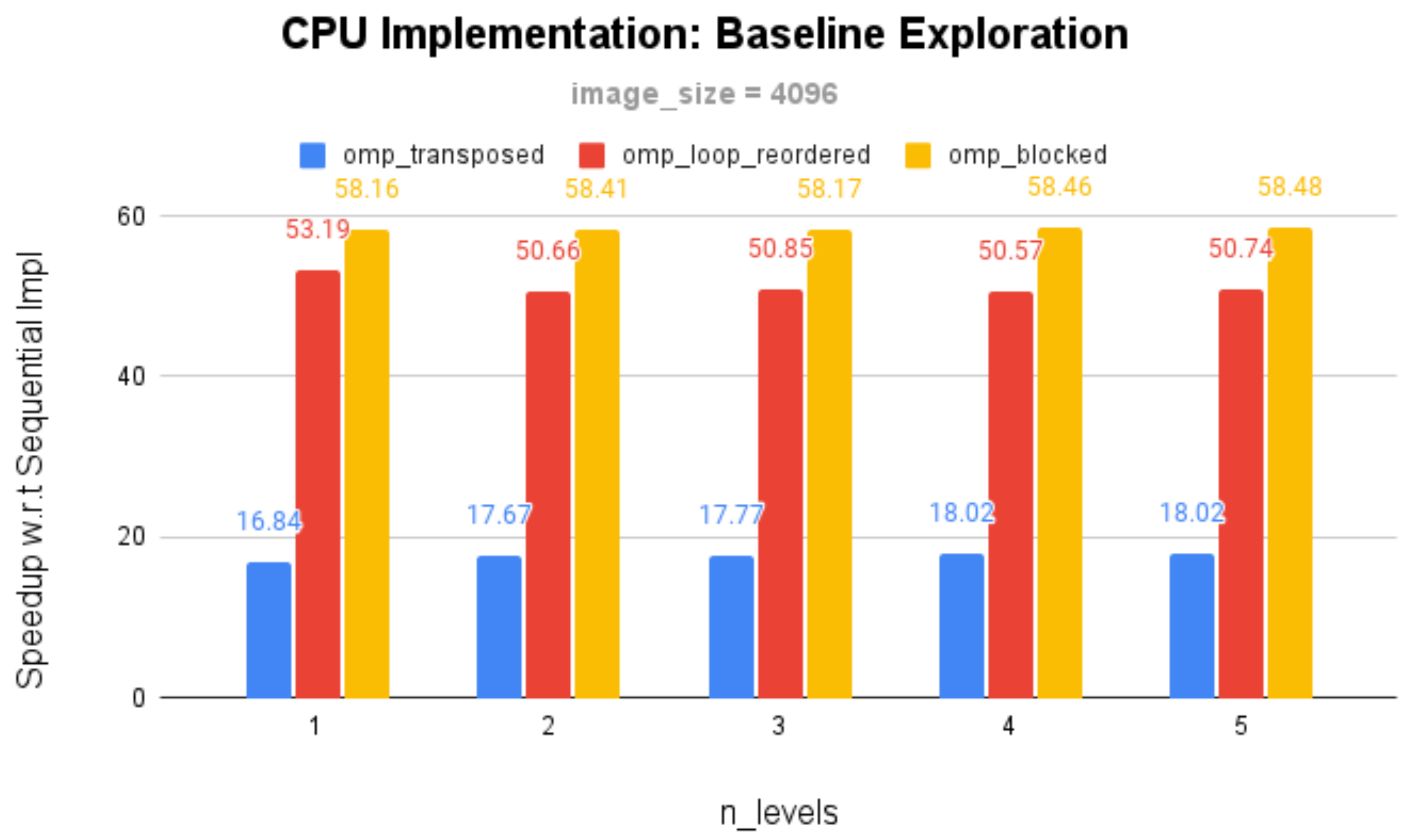


Insight

- Transposition is better than loop reordering for smaller images
- For larger images, loop reordering and blocking give better speedup
- **Block-based approach** is the most efficient, irrespective of image size

Baseline: Shared memory block-based DWT (24 cores)

CPU Baseline Exploration

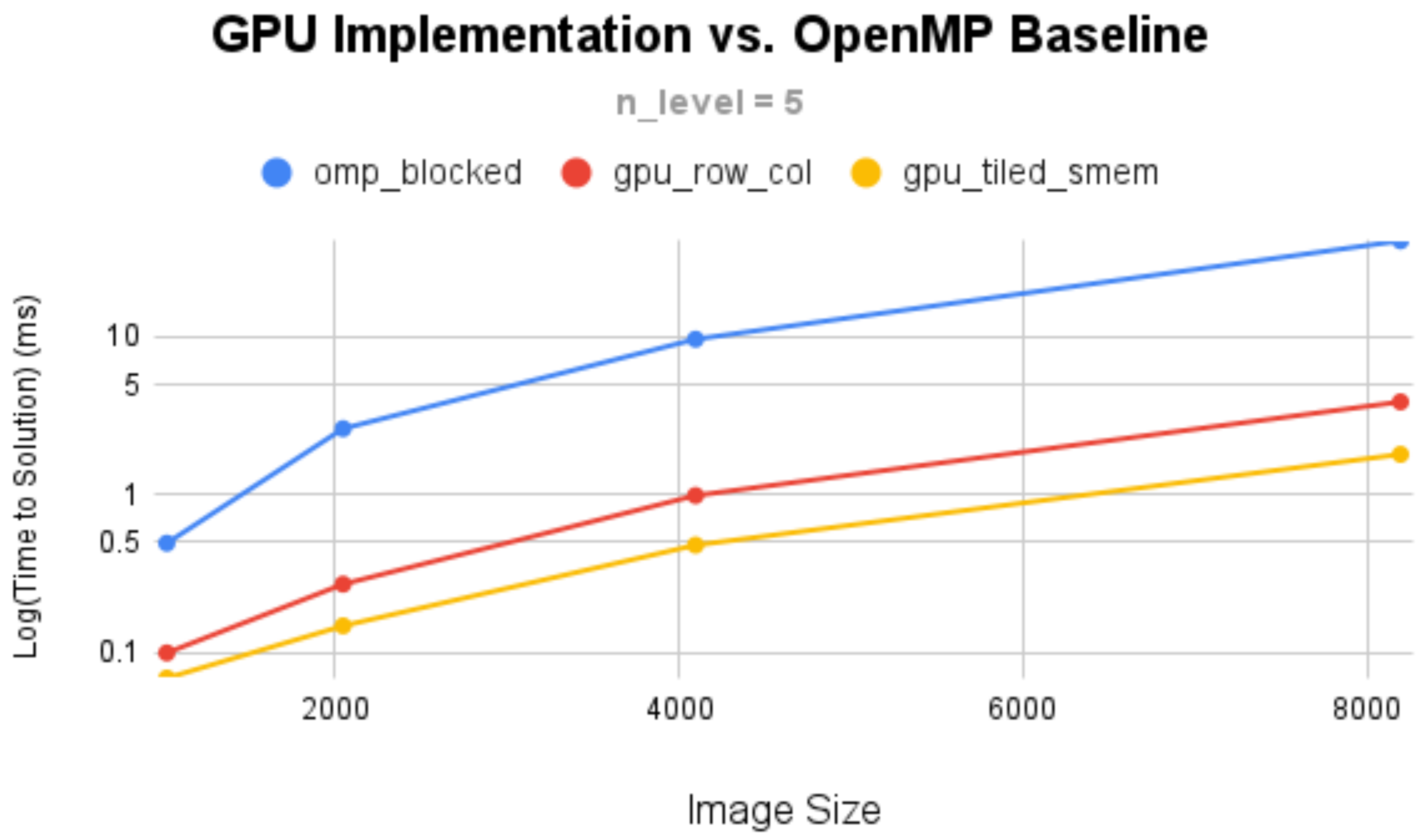


Insight

- Time to solution does not vary significantly across levels due to halving of workload
- Transposition speedup increases across levels as image size decreases (expensive for large images)

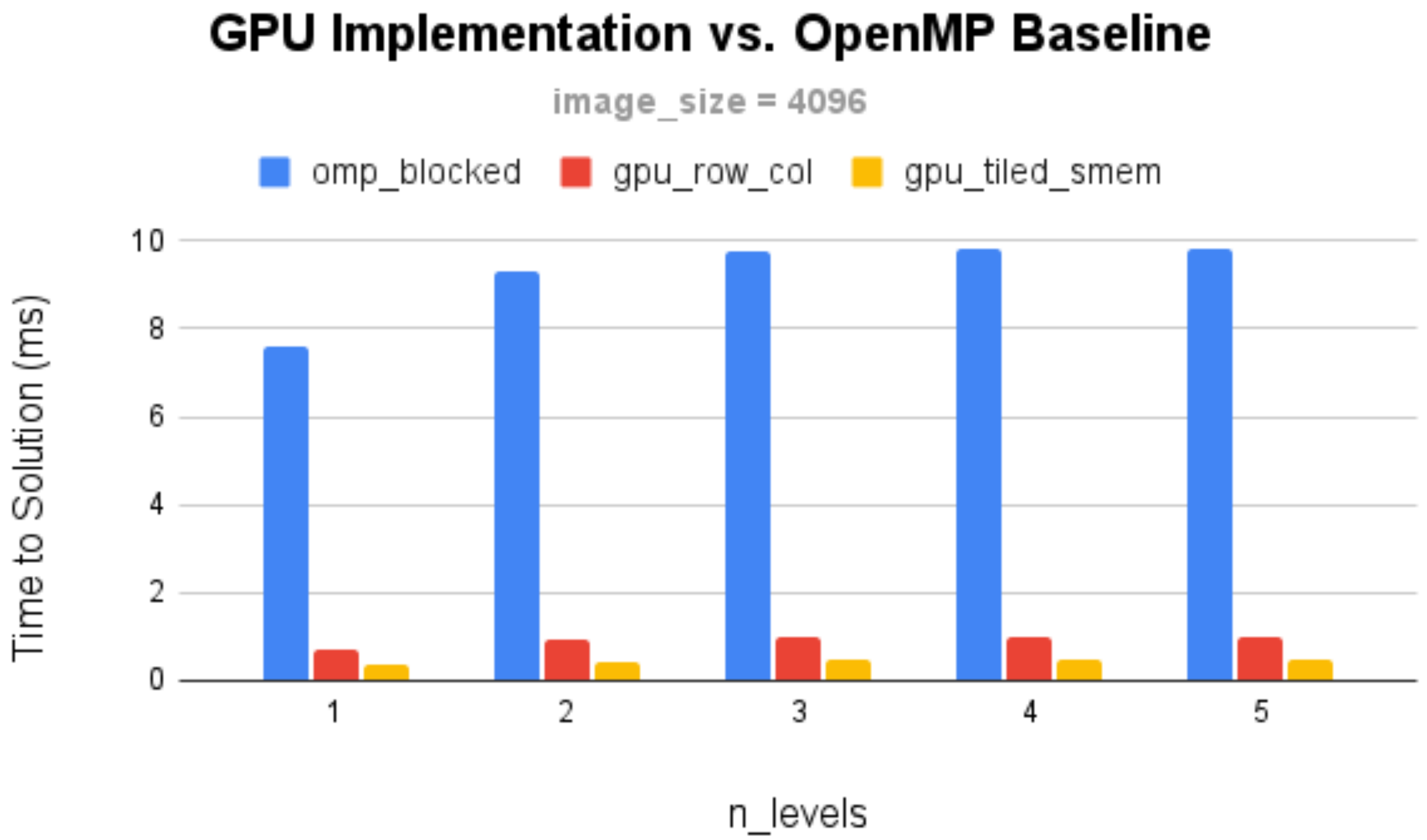
Performance Evaluation

- Baseline: Shared memory block-based DWT (24 cores)



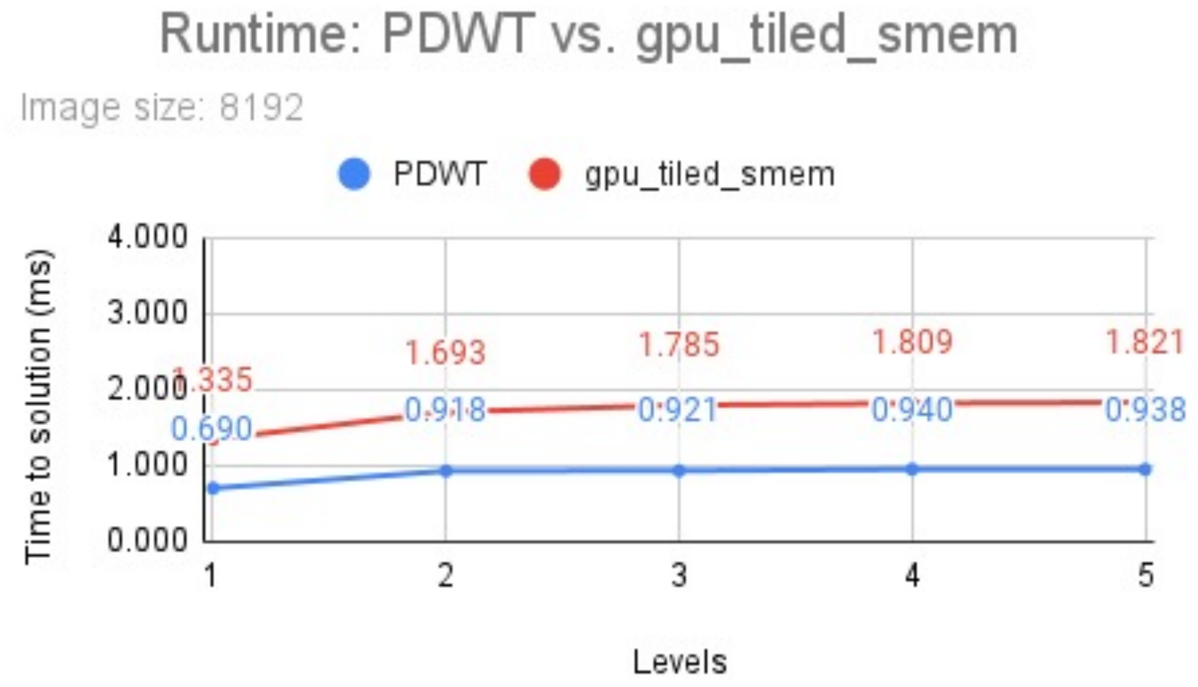
Performance Evaluation

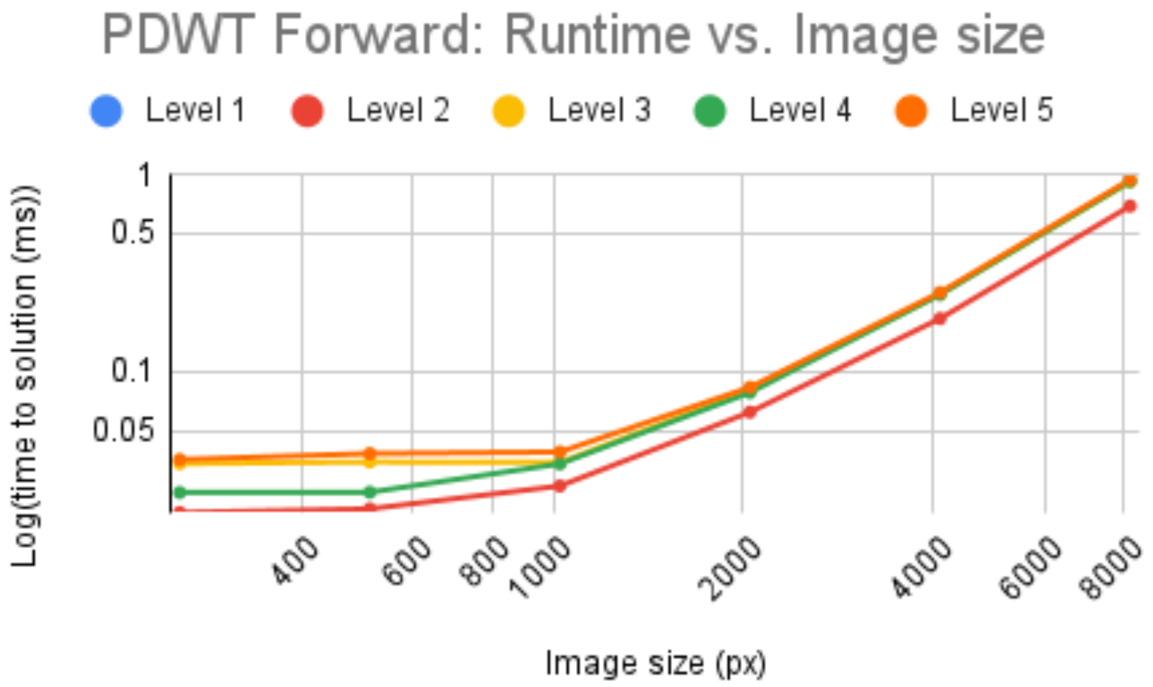
- Baseline: Shared memory block-based DWT (24 cores)



Performance Comparison

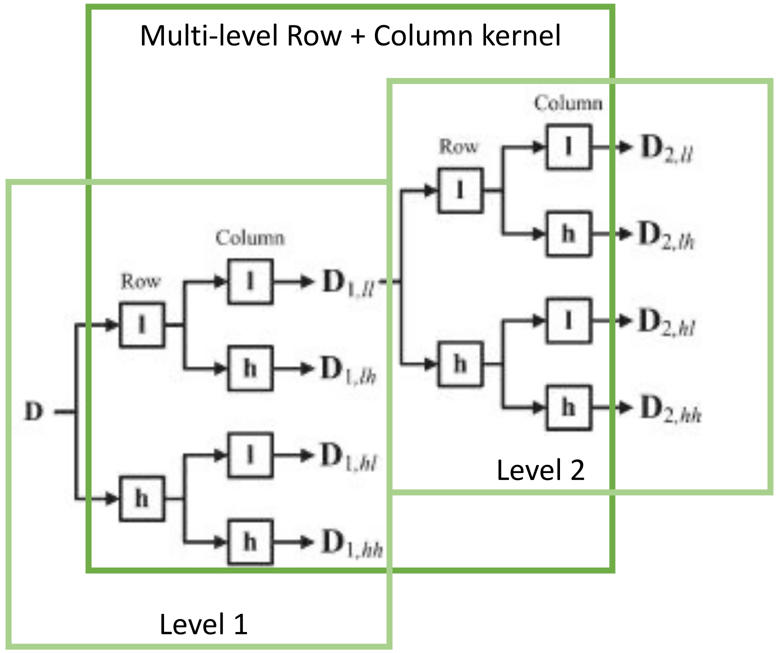
Comparing with PDWT kernel (doesn't do coefficient image stitching, unfair advantage over in-place transform)

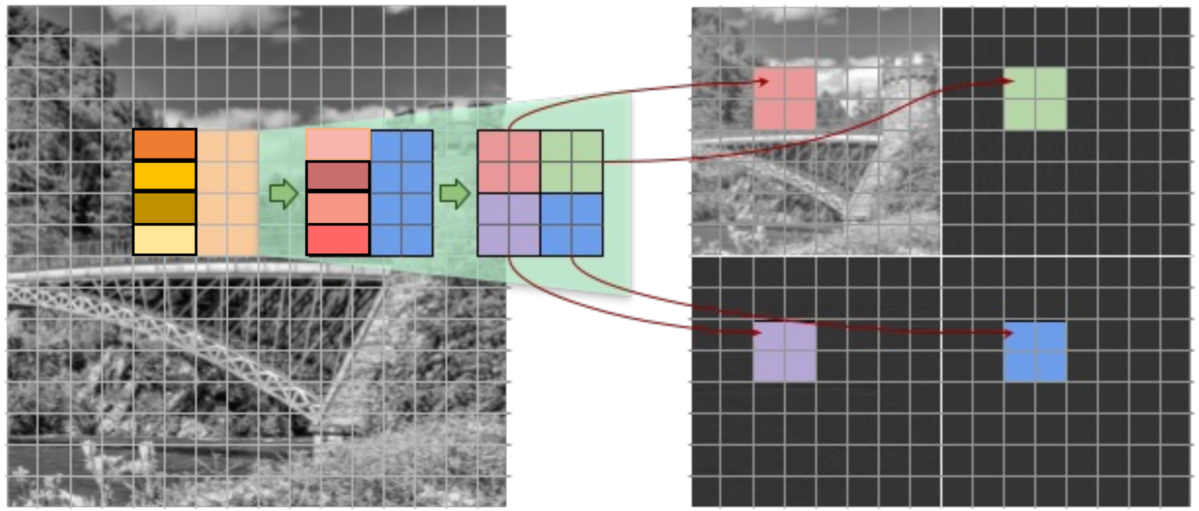




Future directions

- Vectorized computation and memory access
- Computing multiple levels in one kernel:
  - Levels form a pyramidal structure in terms of computation and memory
  - Increases arithmetic intensity
  - Reduces global reads/writes



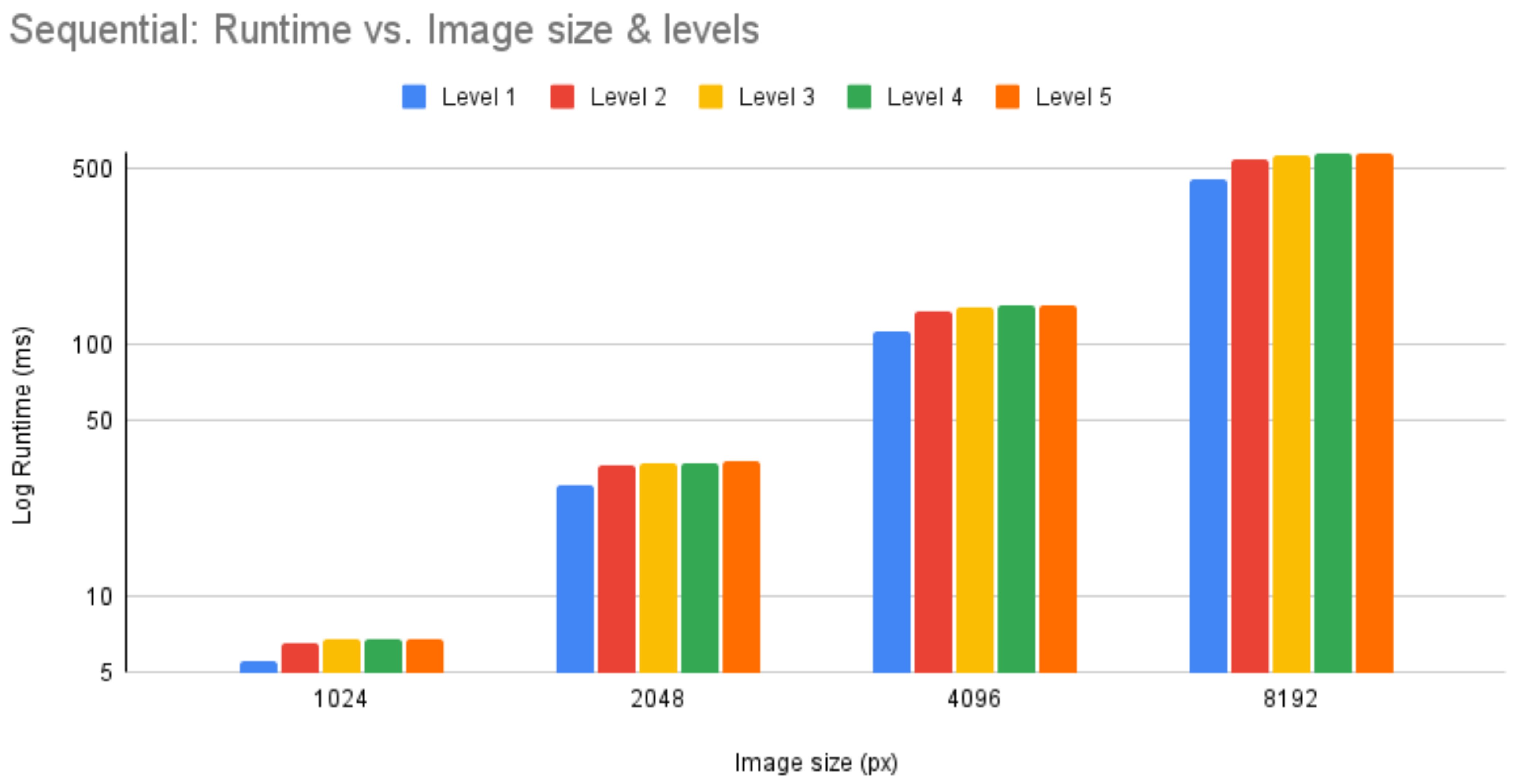


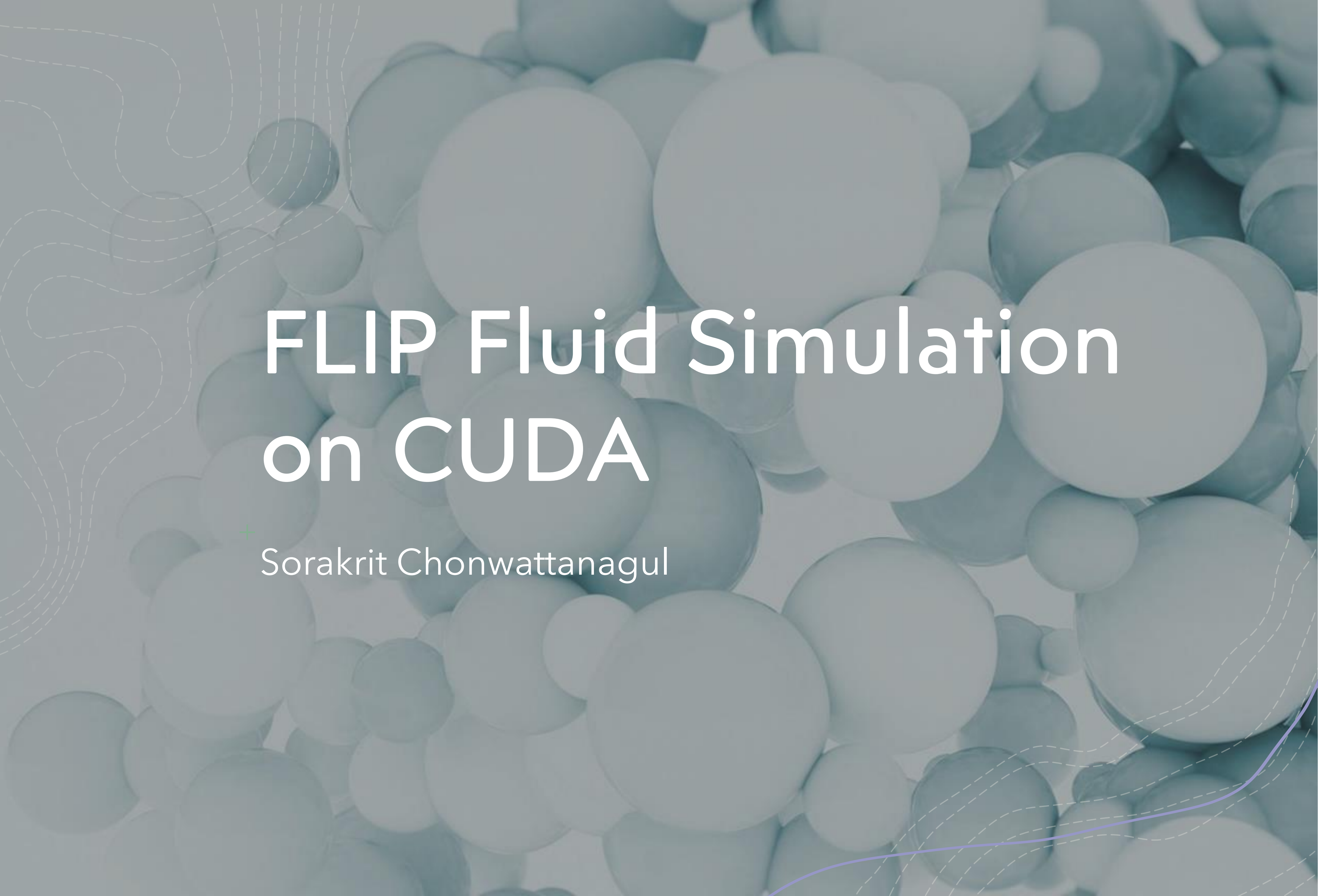
References

- Wavelets and Multiresolution Analysis
- https://github.com/ahandre94/haar
- https://github.com/pierrepaleo/PDWT
- https://ieeexplore.ieee.org/document/4912922
- https://people.math.sc.edu/Burkardt/c_src/haar/haar.html
- https://link.springer.com/article/10.1007/s11227-012-0750-5
- https://www.fit.vut.cz/research/publication-file/11054/postprint.pdf
- https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=b24b1da45aebfb0a46515c5d6303283c65bb1e31
- https://www.fastcompression.com/benchmarks/benchmarks-dwt.htm

Thank you

Appendix


Sequential: Runtime vs. Image size & levels
Level 1
Level 2
Level 3
Level 4
Level 5
Log Runtime (ms)
500
100
50
10
5
1024
2048
4096
8192
Image size (px)



FLIP Fluid Simulation on CUDA

Sorakrit Chonwattanagul

Overview

- Aims to simulate specifically liquids inside a fixed domain
- Reproduce algorithm for GPUs
- Hybrid method, using grids and particles
    - Physics calculations are done on the grid
    - Particles determine shape of fluid

- Source code at: https://github.com/keptsecret/FLIPonCUDA

Overview

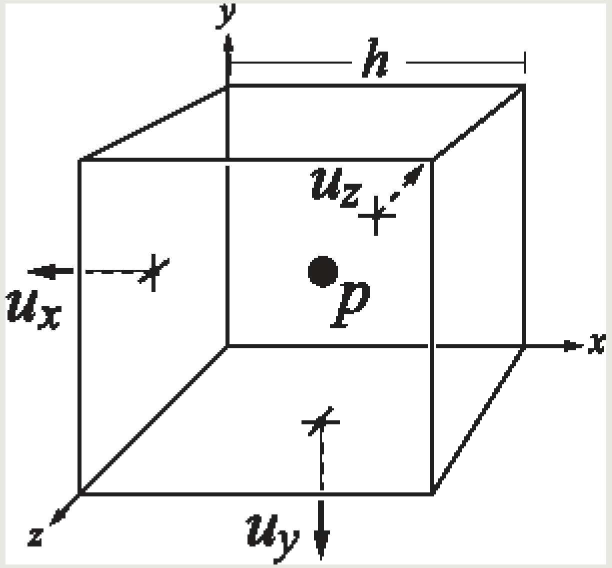
FLIP procedure

1. Simulate particle movement
   1. Create mesh from particles
2. Advect particle velocity to grid
3. Apply forces, e.g. gravity
4. Solve incompressibility (pressure)
5. Advect velocity field to particles

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u},$$
$$\nabla \cdot \vec{u} = 0.$$

Overview

- Uses marker-and-cell method with staggered grid
- 8 marker particles placed in each cell
  - Jittered initialization
- Boundary cells of domain set as solid
- No correction methods implemented
  - Particle separation to preserve volume



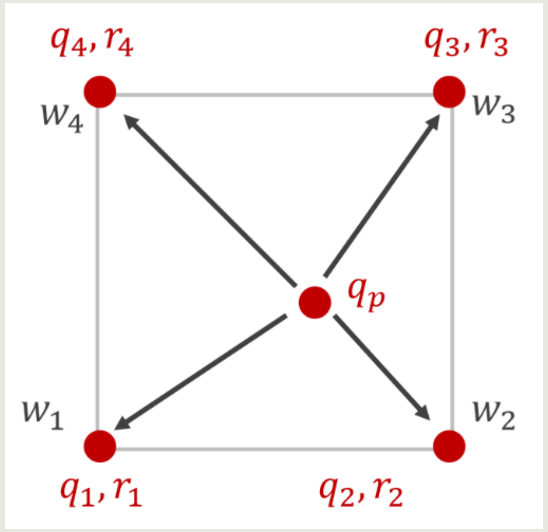
Overview

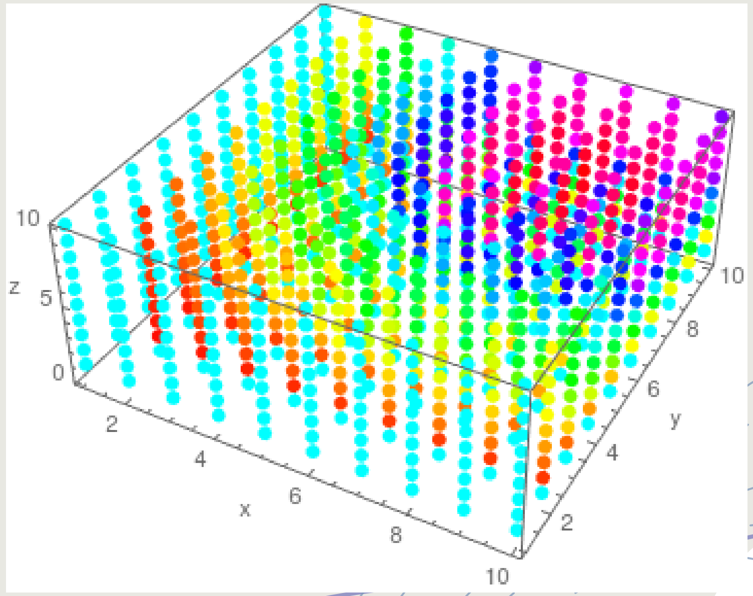
FLIP procedure

1. Simulate particle movement	Simple loop
1. Create mesh from particles	CUDA accelerated (partially)
2. Advect particle velocity to grid	CUDA accelerated
3. Apply forces, e.g. gravity	Simple loop
4. Solve incompressibility (pressure)	Simple loop (not really but…)
5. Advect velocity field to particles	CUDA accelerated

CUDA accelerated velocity field advection

- Still has lots of setting values in loops
- Particle velocity first transferred to a scalar field
- Scalar field values then determine velocity field values
  - This is mostly just copying values over





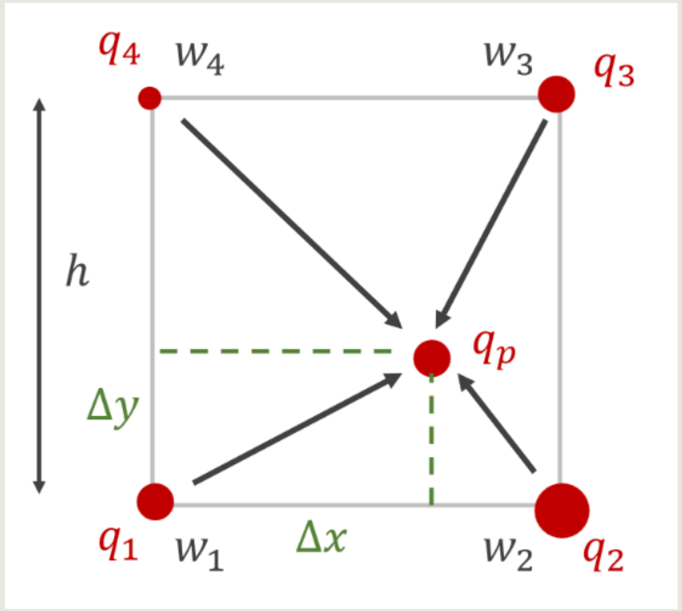
CUDA accelerated velocity field advection

- Scalar field represented by 3D array of floats
- Each particle contributes a velocity at its position in scalar field

- Split scalar field array into smaller 8x8x8 sub-arrays
- Determine which particles affect which sub-arrays
- CUDA kernel does calculation for set of sub-arrays
  - Each block handles one sub-array and its particles
  - Each thread handles one cell of sub-array

CUDA accelerated particle advection

- Transferring velocities from grid onto particles
- Lots of interpolation involved, mainly tricubic
- Interpolate one direction at a time: u_x, u_y, u_z



CUDA accelerated particle advection

- Split velocity field array into smaller sub-arrays
- Determine which particles are inside which sub-arrays
- CUDA kernel does calculation for set of sub-arrays
  - Each block handles one sub-array and its particles
  - Each thread handles one particle

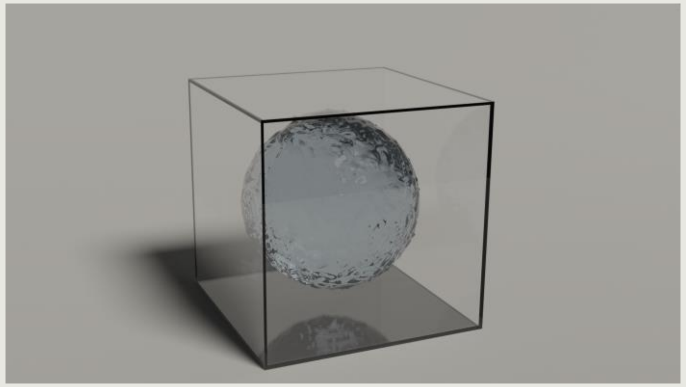
- Lots of padding of particle velocities and field values
  - When trying to interpolate values outside of domain

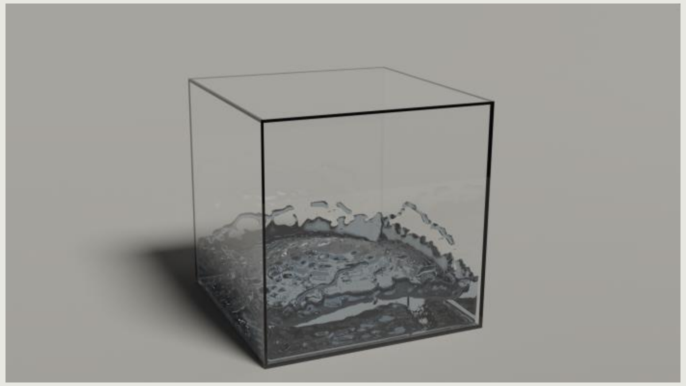
Testing

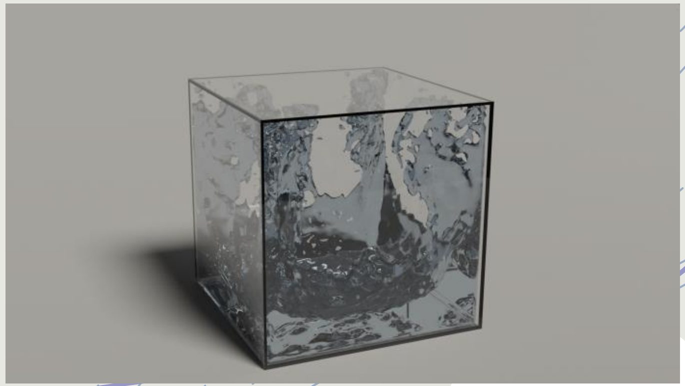
- Validation and baseline:
  - Compare to single-threaded simulation
    - Fixed initialization seed
  - Would like to use external FLIP solvers (e.g. Mantaflow) as comparison but:
    - Parameters are different (also more than this has)
    - Random initialization

Testing: Scenario 1

- Fluid sphere drop
- Simulated on a 64x64x64 grid with 0.125 cell size for 60 frames
- Particles created: 462,848



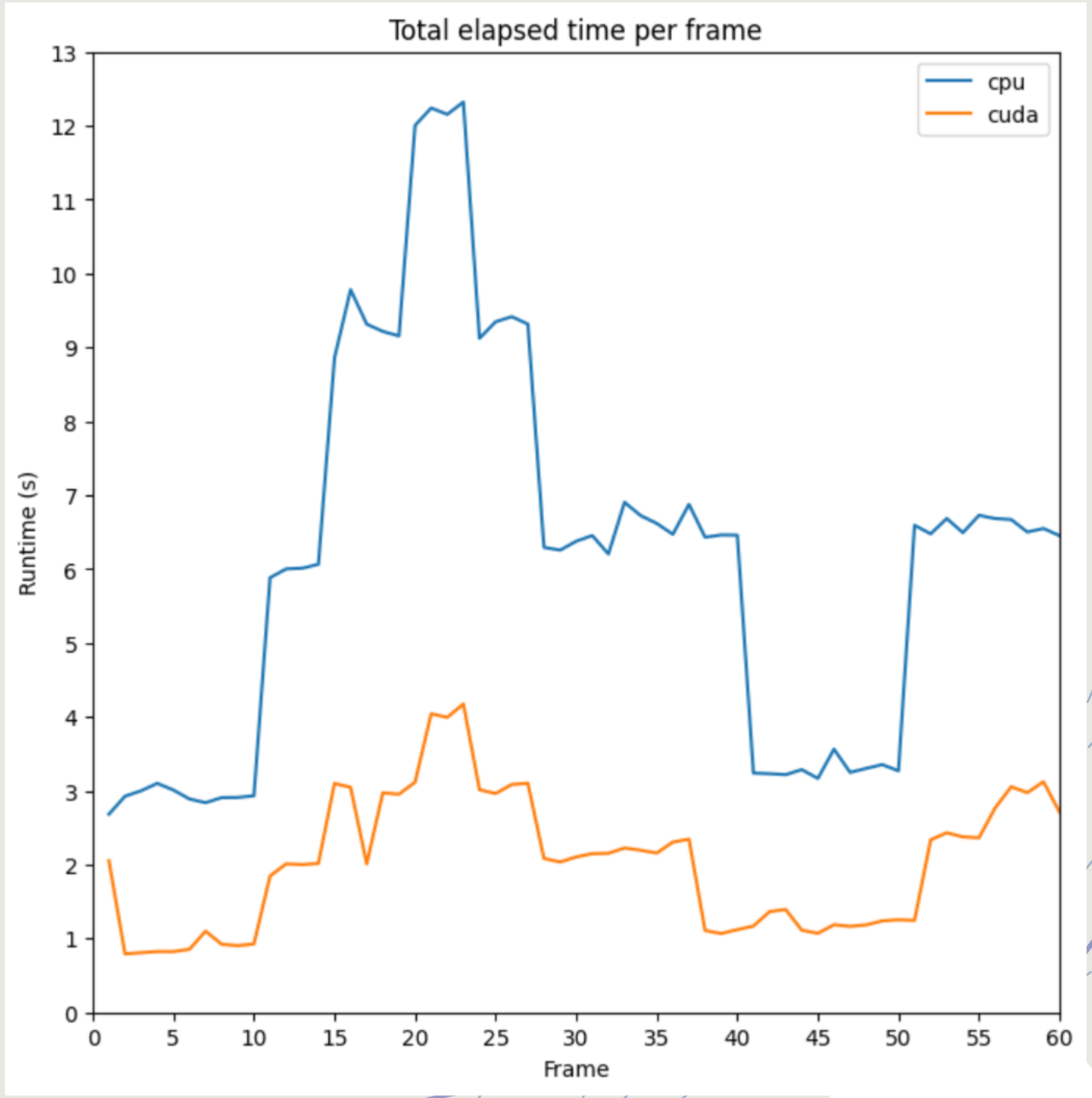




- All scenes rendered in Blender Cycles
- Tested on Intel i9-11900H @ 4.2GHz and RTX 3060 Mobile @ 1435 MHz

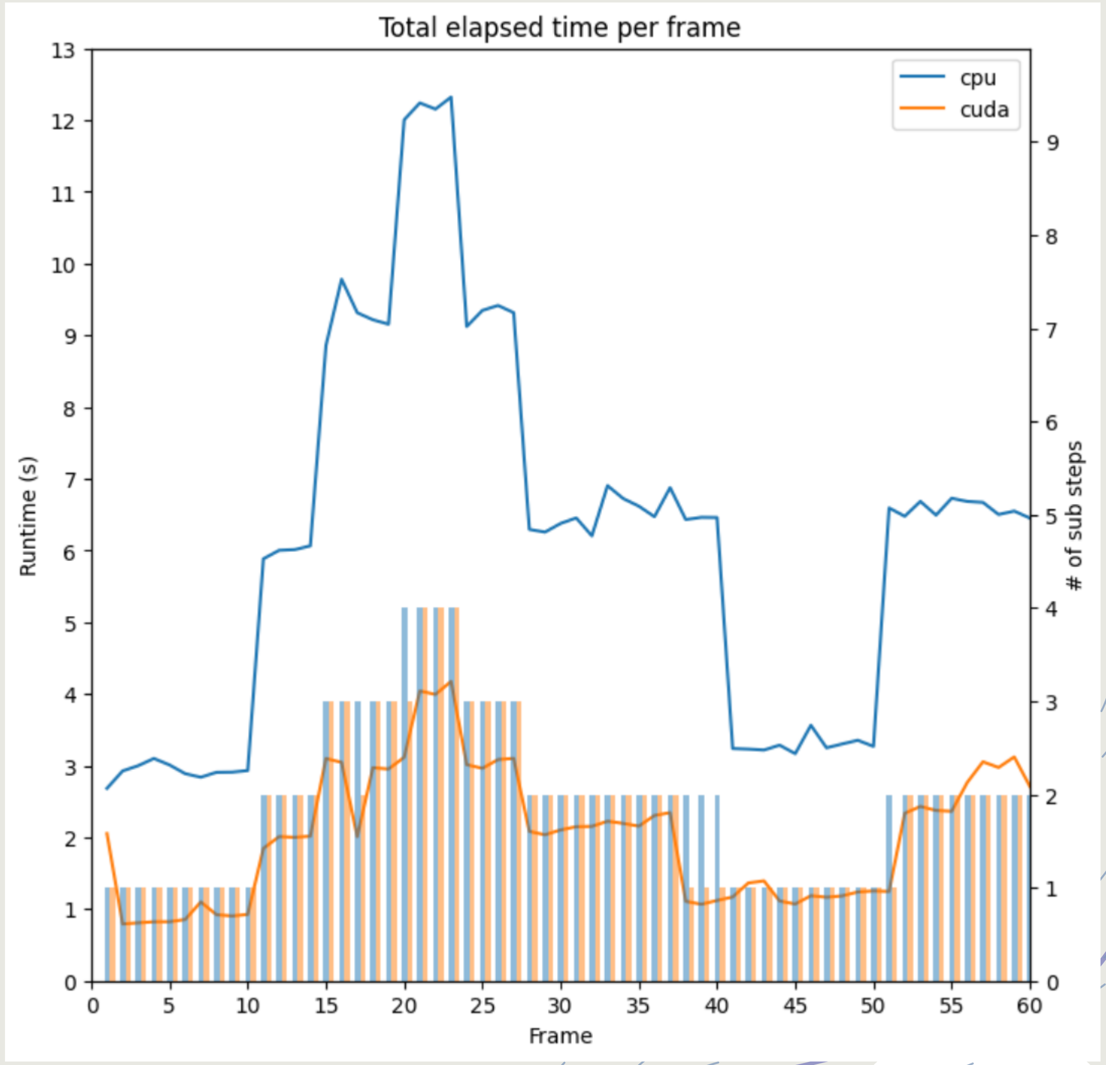
Testing: Scenario 1

+ Total running times
+ Between 2x-3x faster than single-threaded CPU



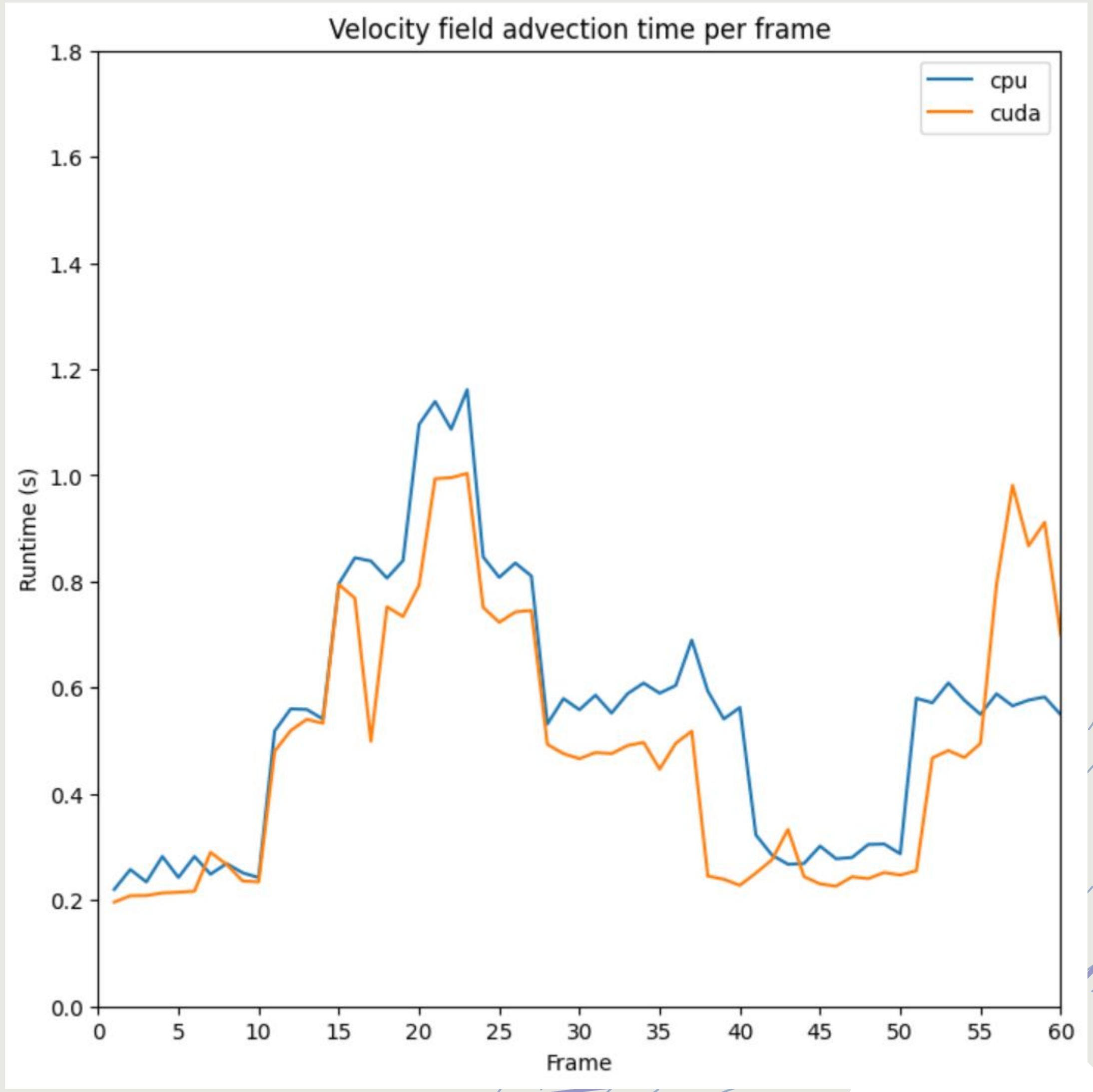
Testing: Scenario 1

+ Total running times
+ Between 2x-3x faster than single-threaded CPU



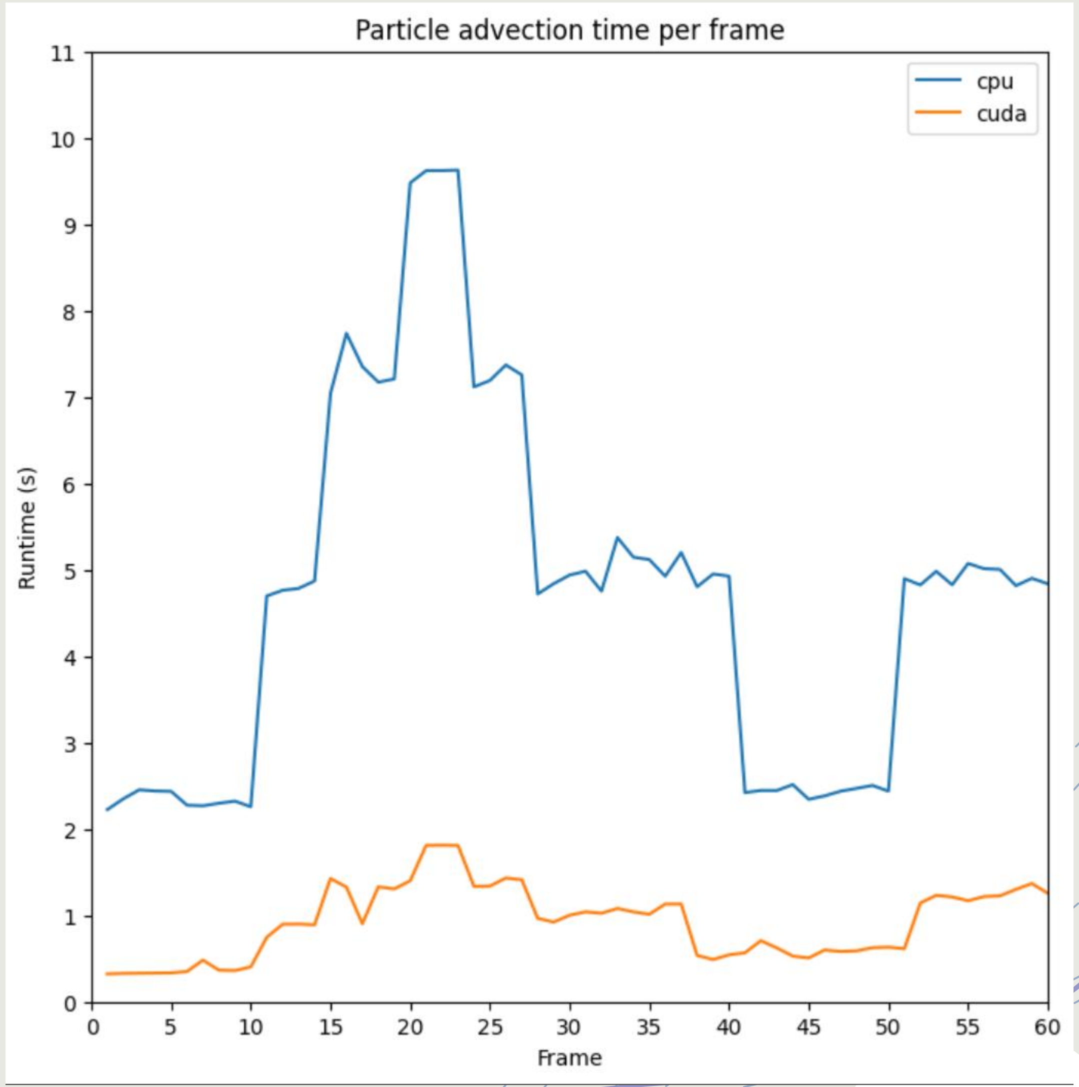
Testing: Scenario 1

- Velocity field advection
- Not much difference
- Should see more significant difference on a larger grid with smaller cells



Testing: Scenario 1

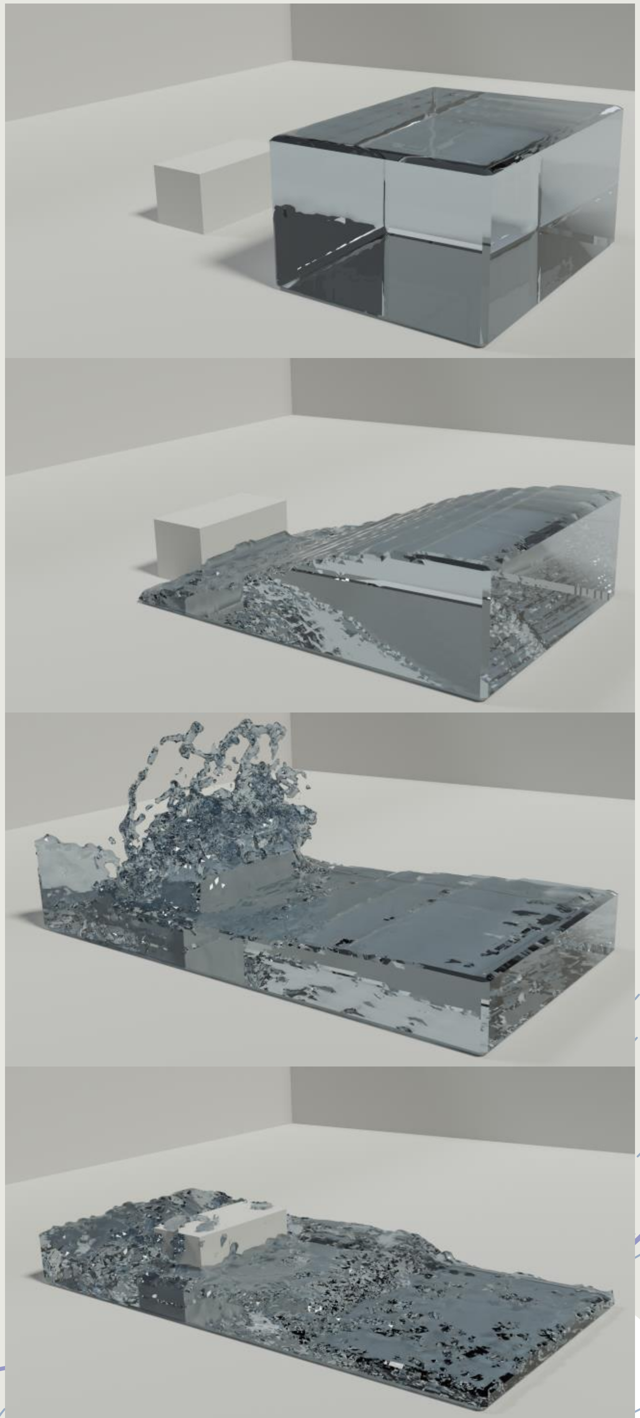
+ Particle advection
+ Most difference seen: about 5x-7x faster



Testing: Scenario 2

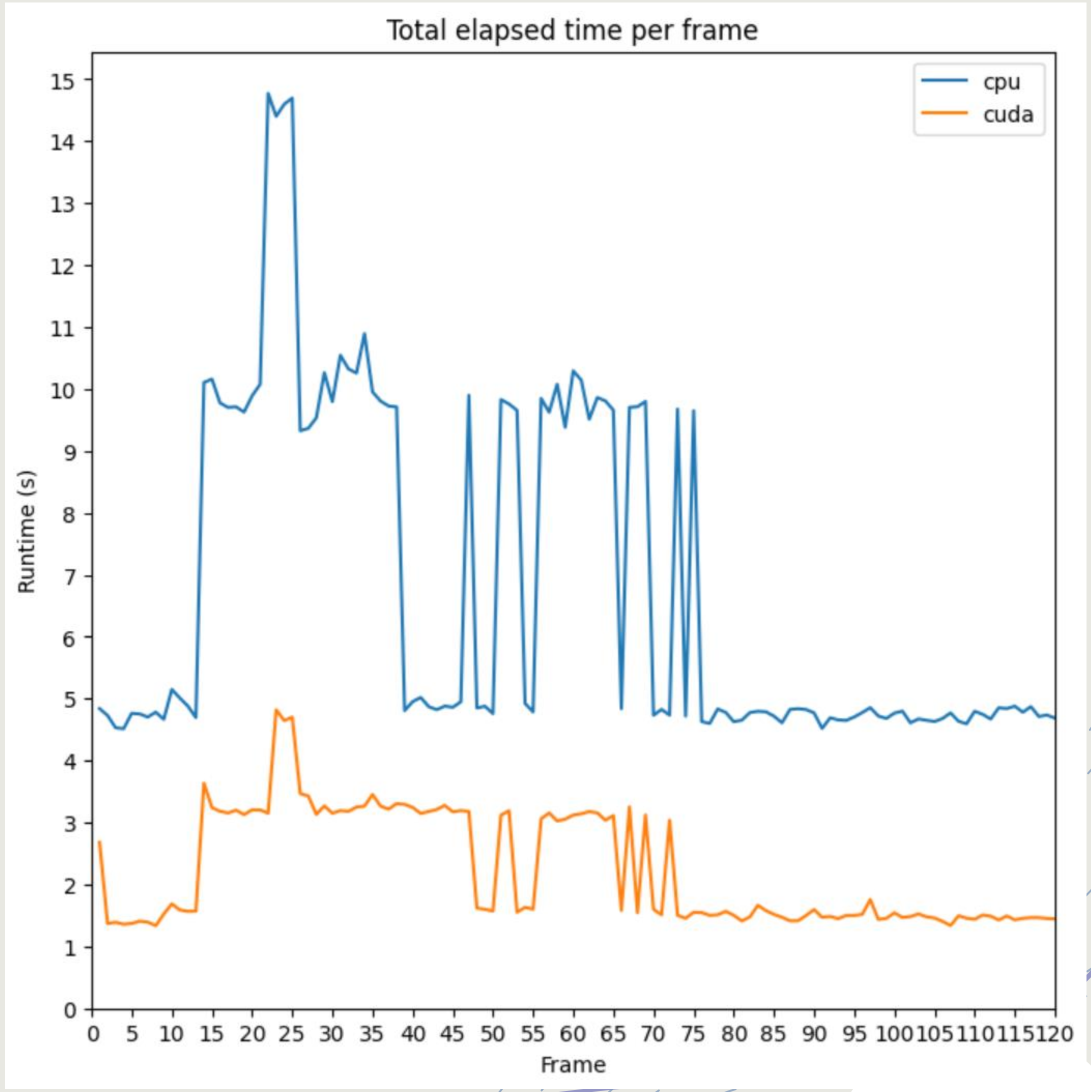
- Fluid cuboid wave with solid obstacle
- Simulated on a 128x64x64 grid with 0.125 cell size for 120 frames
- Particles created: 722,672

- All scenes rendered in Blender Cycles
- Tested on Intel i9-11900H @ 4.2GHz and RTX 3060 Mobile @ 1435 MHz



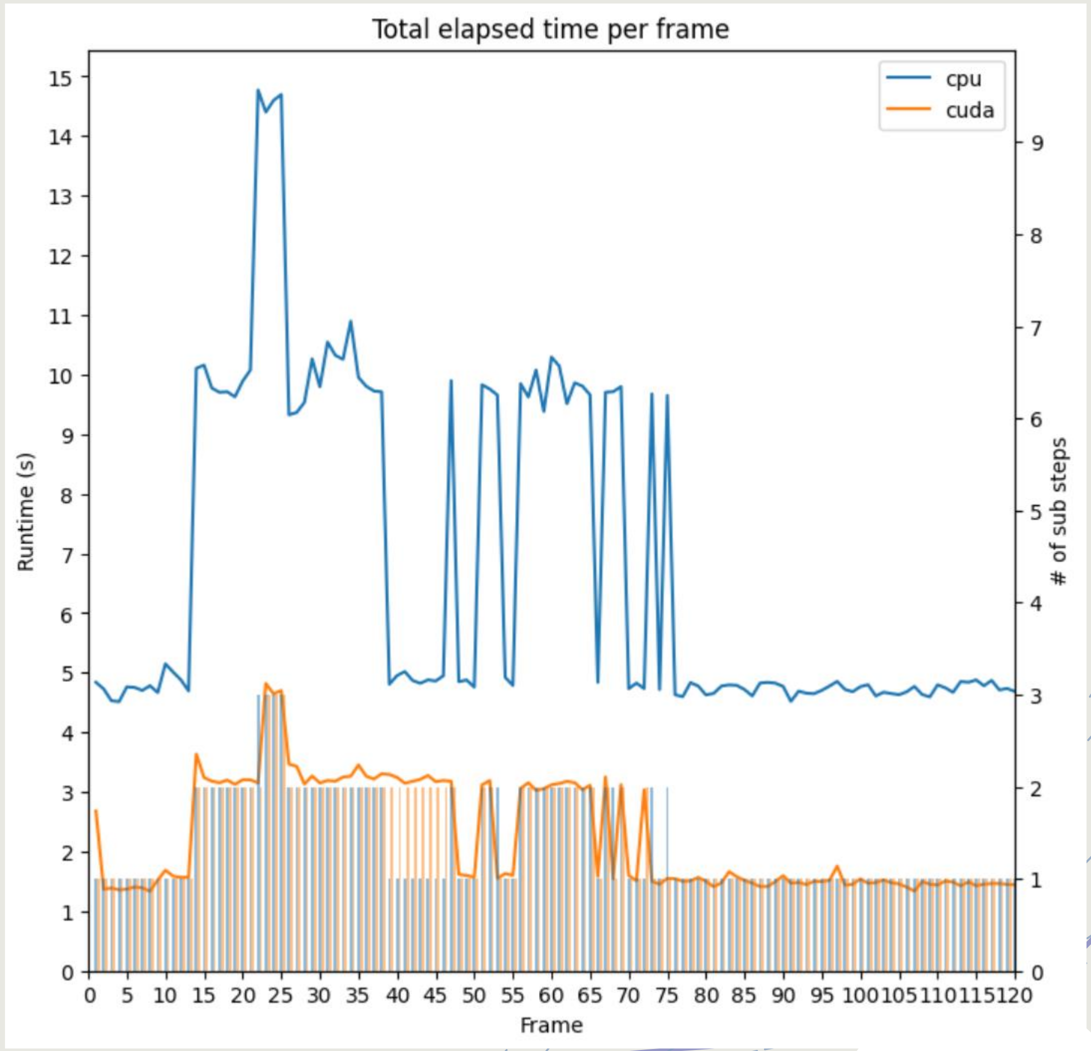
Testing: Scenario 2

- Total running times
- Again, ~3x faster than single-threaded CPU



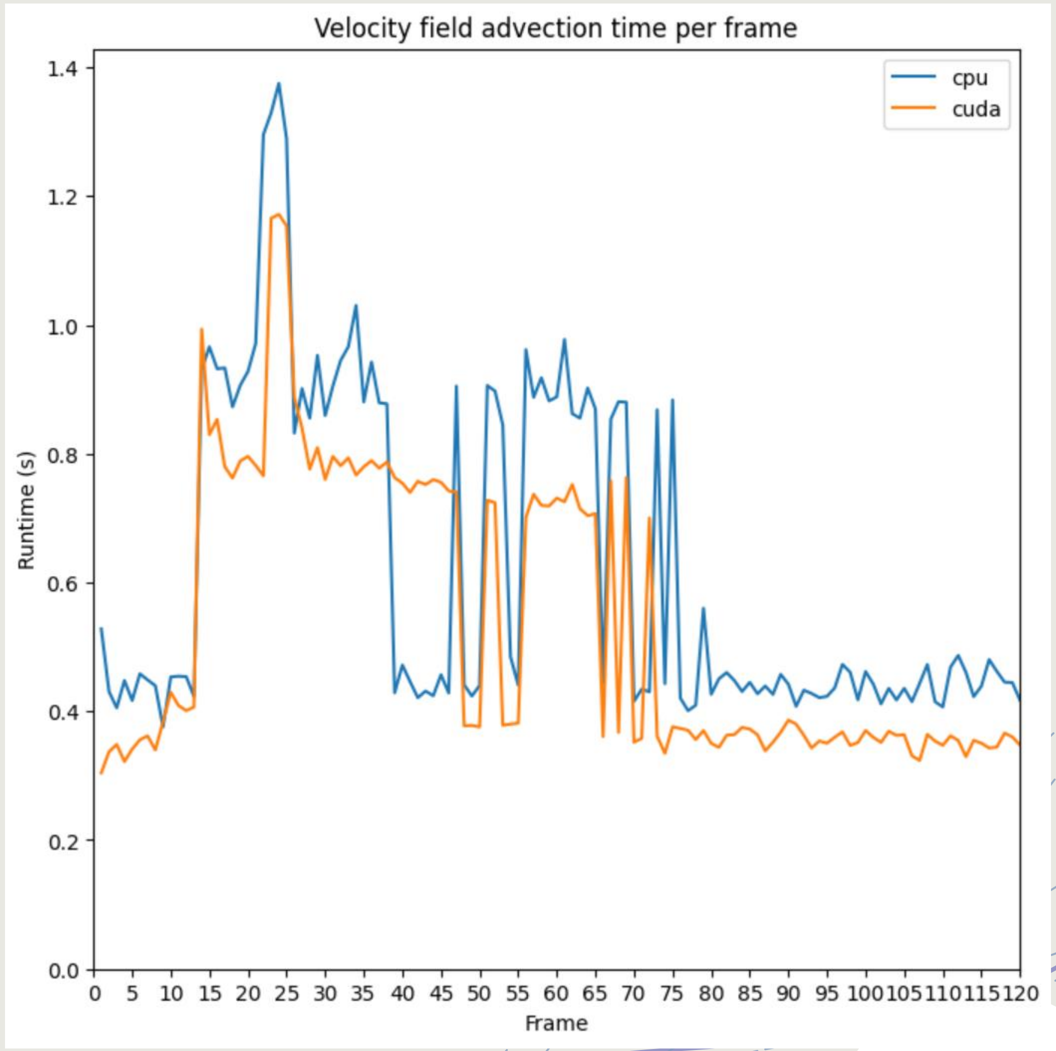
Testing: Scenario 2

+ Total running times
+ Again, ~3x faster than single-threaded CPU



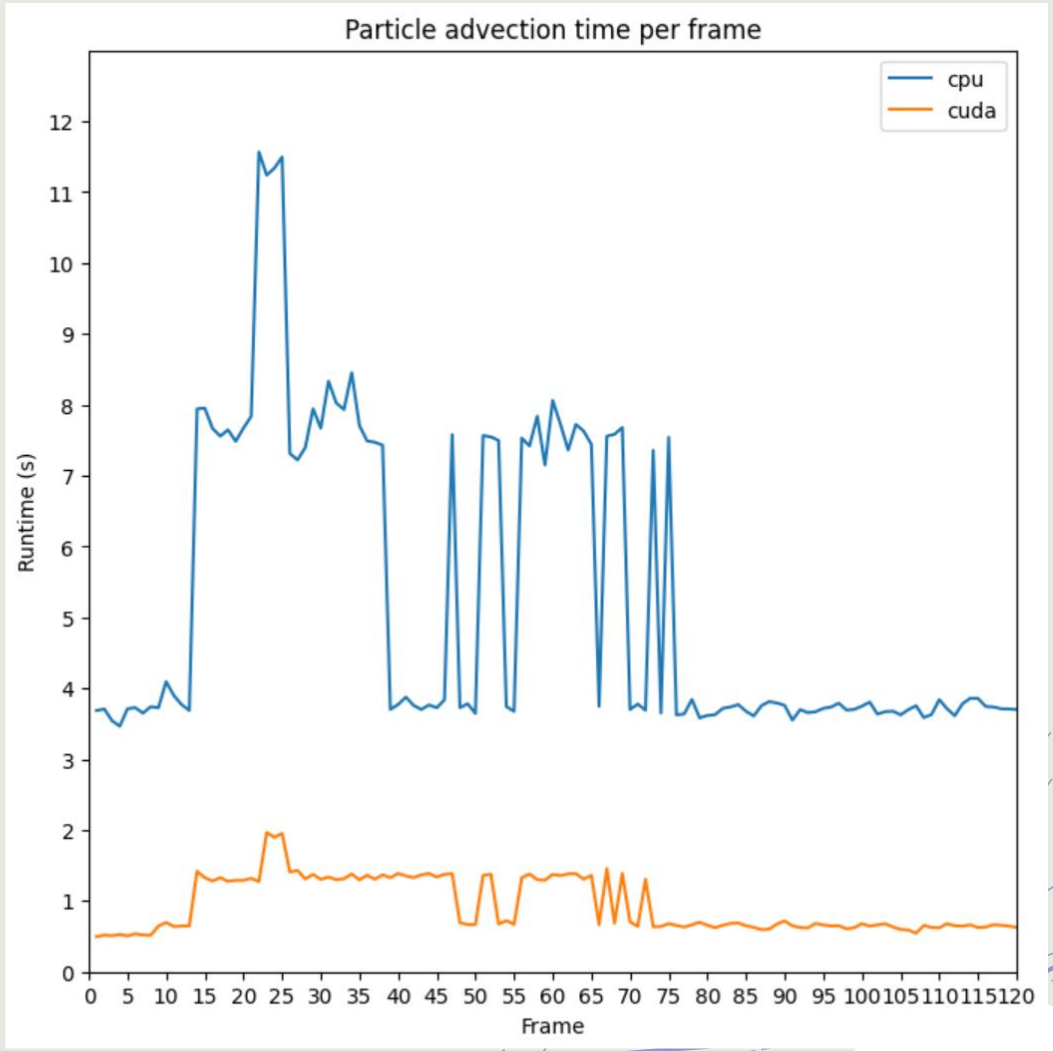
Testing: Scenario 2

- Velocity field advection
- Can see the general speedup here of about 1.2x-1.7x



Testing: Scenario 2

- Particle advection
- Again, the biggest speedup seen of 6x-8x



Times per sub-step

Scenario 1

Arch	Total time (s)	# steps	Time per step (s)	Velocity advection time per step (s)	Particle advection time per step (s)
CPU	368.68554	117	3.15115	0.28086	2.43153
CUDA	122.04680	111	1.09952	0.26287	0.49833
Speedup	3.02x	-	2.87x	1.07x	4.88x

Scenario 2

Arch	Total time (s)	# steps	Time per step (s)	Velocity advection time per step (s)	Particle advection time per step (s)
CPU	814.79249	168	4.84995	0.44612	3.77468
CUDA	268.95110	172	1.56366	0.37457	0.65610
Speedup	3.02x	-	3.10x	1.19x	5.75x

Implementation issues/speed bumps

- OpenVDB was harder to integrate in a CMake project than expected
  - Planned to use it for fast particle-to-mesh conversion
  - Had to write one/adapt from existing code (Marching cubes), unsure of how much overhead could have been saved
- Less parallelizable than expected
  - At least on GPU
- Lots of overhead copying data between host and device

Potential improvements

- Simulation Features:
  - Diffuse particles, e.g. bubbles, spray
  - Importing meshes as solid obstacles
- Parallelization Features:
  - Multithreading (though I don't expect to see much more speed up)
  - CUDA unified memory (`cudaMallocManaged`)
    - Makes memory accessible from CPU and GPU, reducing need to copy data
    - Scaling issue: potential to run out of memory on massive scenes

Accelerating Proximal Policy Optimization (PPO)

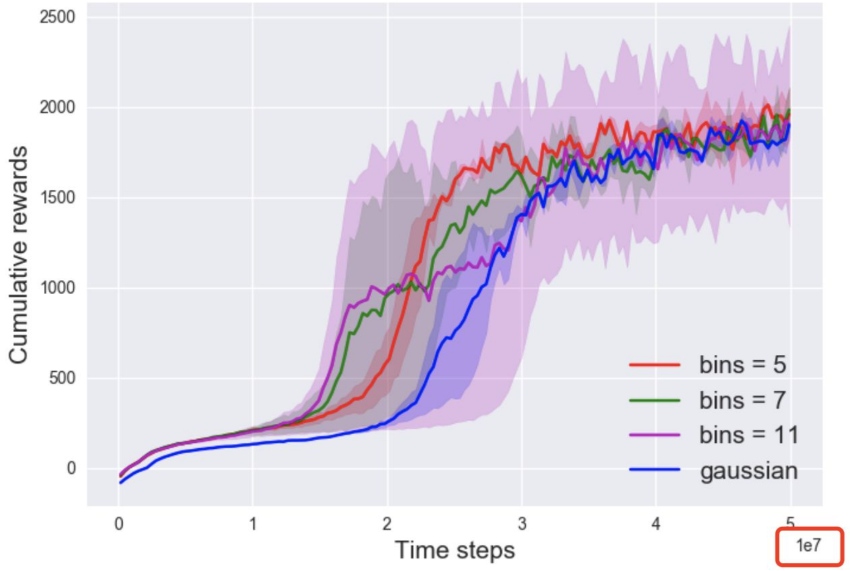
Akhil Goel, Matthew Woodward, Qingyu Xiao

Outline

- Problem Definition, Category, Performance Metrics
- Baselines & Dataset
- Proposed Solution
- Challenges
- Methods
- Results
- Discussion

Problem Definition

- Reinforcement Learning
- High sample volume
- Embarrassingly parallel
- Bottlenecks
  - Environment Simulation
  - Scaling across multiple GPUS (synchronization cost & errors)



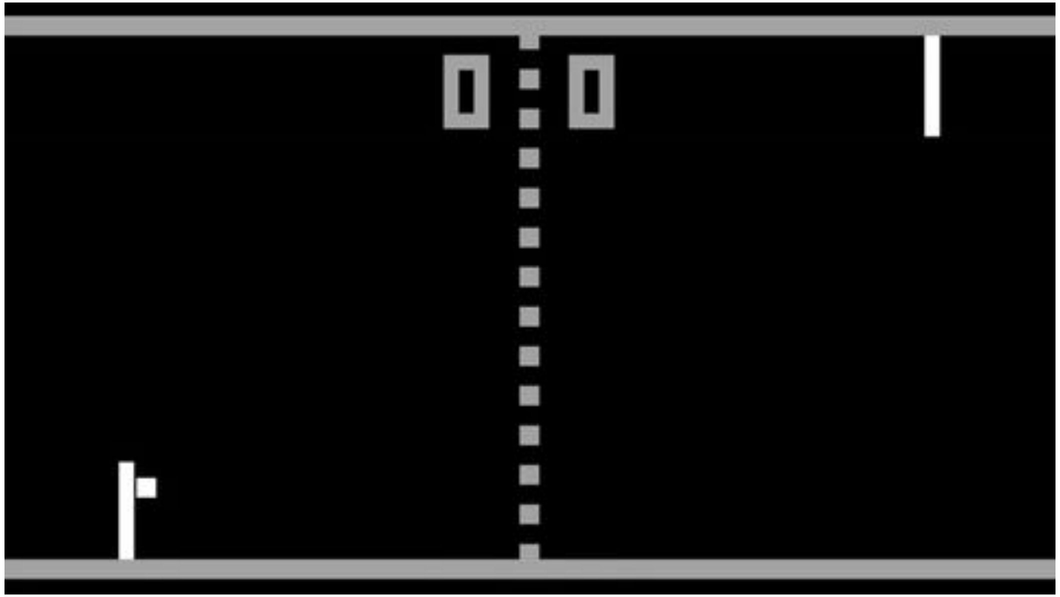
Training time steps is very large, roughly 5e7 time steps

Project Category & Performance Metric

- Reproducibility: DD-PPO (https://arxiv.org/abs/1911.00357)
  - Decentralized Distributed Proximal Policy Optimization
  - **Key Ideas:**
    1. Environment Acceleration
    2. Multi-GPU Scaling
- Scaling Metrics
  - Frames Per Second (FPS) (steps of experience / second)
  - Total Training Time (for fixed total samples)
- Validation Metric: Policy Reward

Baselines & Dataset

- Classical PPO (https://arxiv.org/abs/1707.06347)
  - Single Worker (CPU + GPU)
- Environment:
  - OpenAI Gym - Pong



Proposed Solution

- DD-PPO (https://arxiv.org/abs/1911.00357)
  - GPU Acceleration of Environments
  - Decentralized synchronous update with worker pre-emption

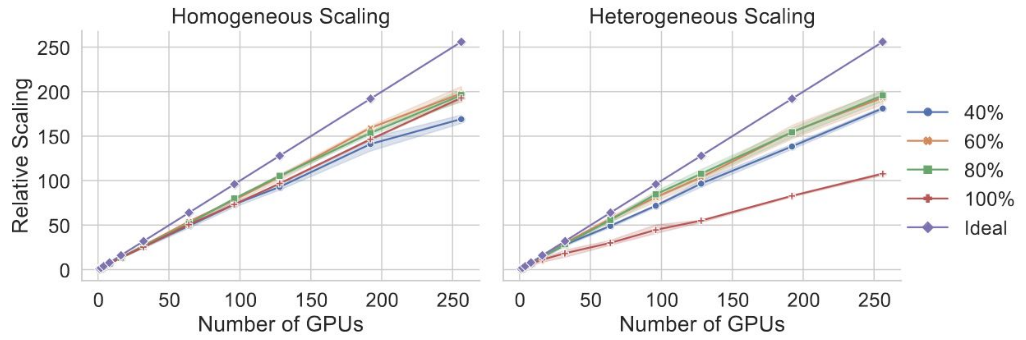


Figure 4: Scaling performance (in steps of experience per second relative to 1 GPU) of DD-PPO for various preemption threshold, $p\%$, values. Shading represents a 95% confidence interval.

NVlabs/**cule**

CuLE: A CUDA port of the Atari Learning Environment (ALE)

3 Contributors · 15 Issues · 213 Stars · 31 Forks

Challenges

Anticipated

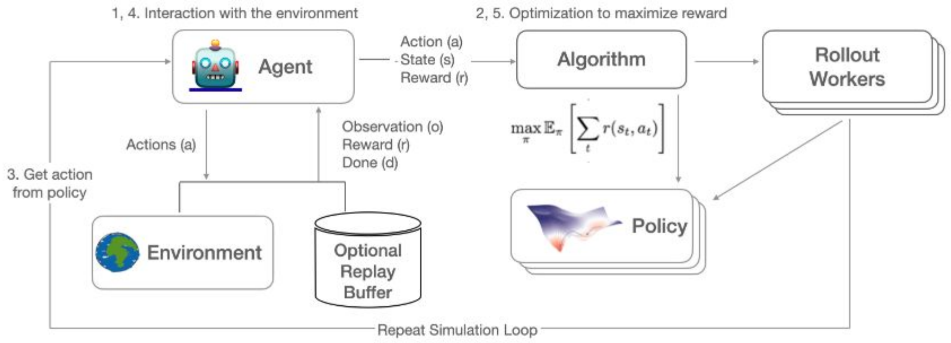
- Compute resources limit scaling tests
  - Computationally simpler environment
  - Measuring synchronization costs at small scale

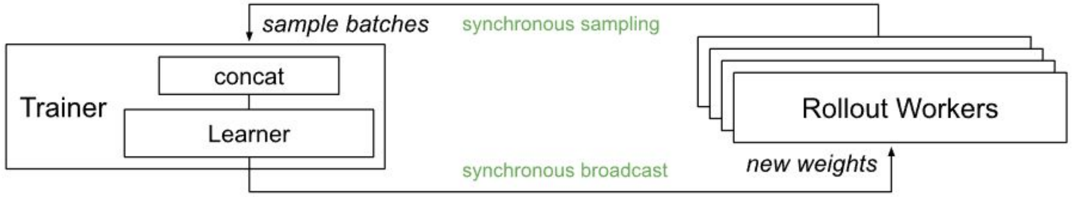
Not Anticipated

- Cuda Driver, GPU model, PyTorch Version, 3rd Party Library Compatibility
- Lack of Docker and Root Access
- “Exclusive-Process” GPU Mode

Methods

- **CULE**
  - GPU Accelerated Atari Learning Environments
  - Based on Pytorch and CUDA
- **Ray**
  - Distributed Computing
  - Actor-based programming model
  - Easy to use API
  - Work with Pytorch





Synchronous Sampling (e.g., A2C, PG, PPO)

Methods

Constant Hyperparameters

- ale_start_steps=400
- ***batch_size=256***
- episodic_life=True
- lr=0.00025
- max_episode_length=18000
- normalize=False
- ppo_epoch=3
- num_stack=4
- ***Num_steps=20***
- t_max=5,000,000
- optimizer=SGD

Methods

Independent Variables

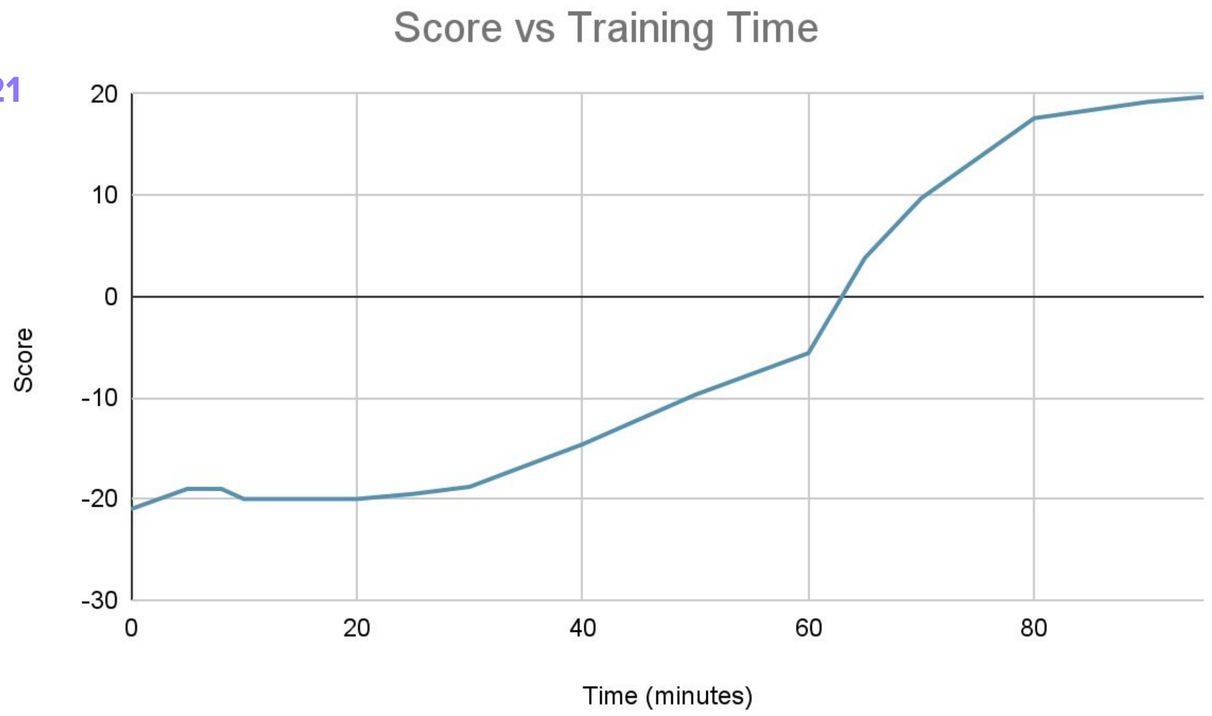
- GPU Acceleration of Environments
- Number of Parallel Atari Learning Environments
- Number of Batches
- Number of Steps

Dependent Variables

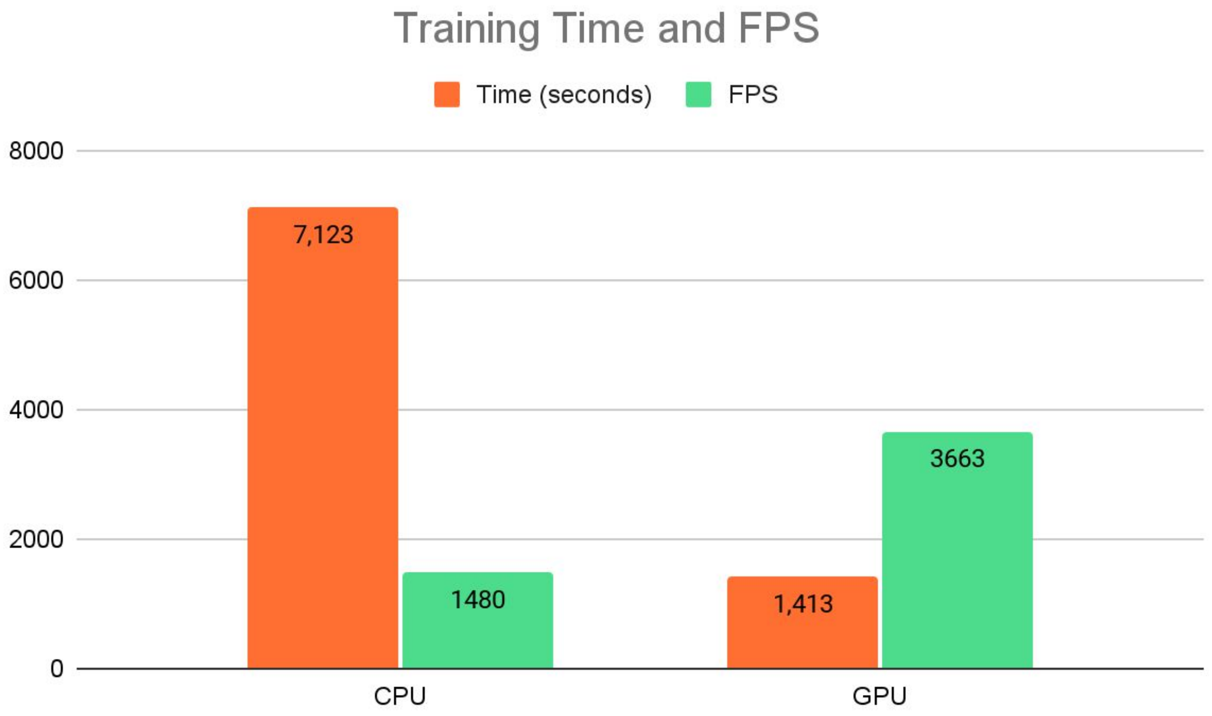
- Frames per Second (FPS) (steps of experience / second)
- GPU Memory Usage
- Total Time (per fixed training samples)

Validation: Pong using GPU CULE PPO

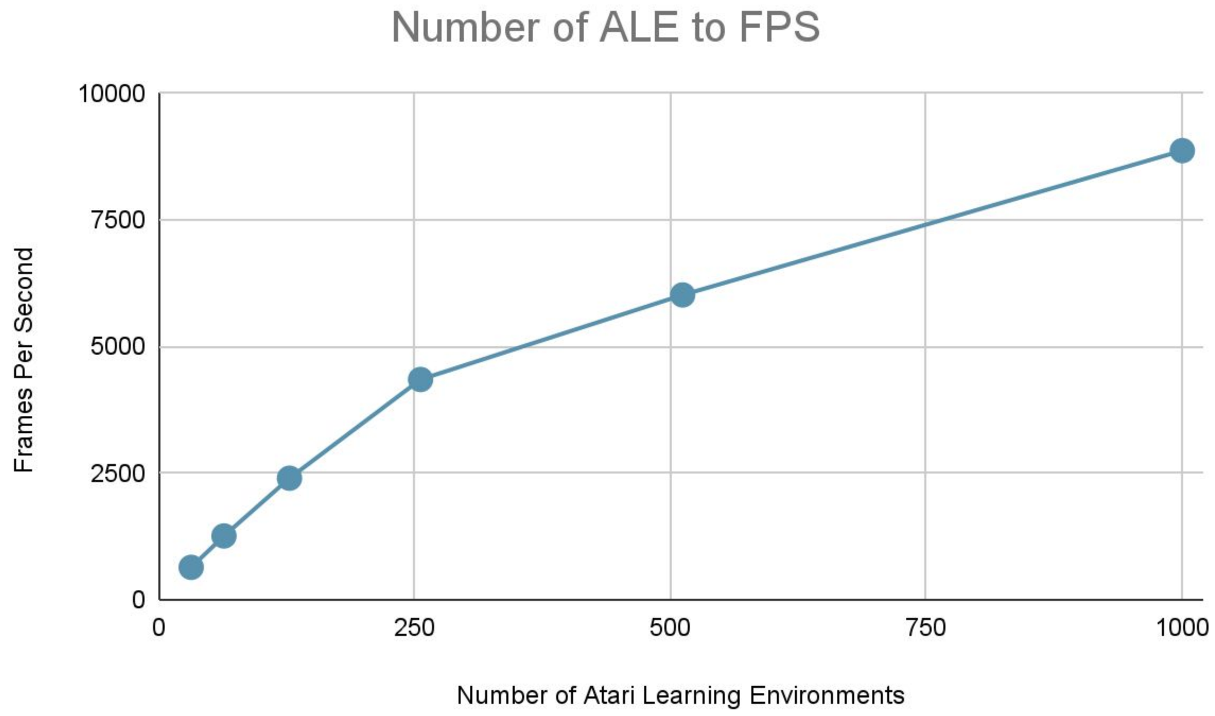
Game out of 21



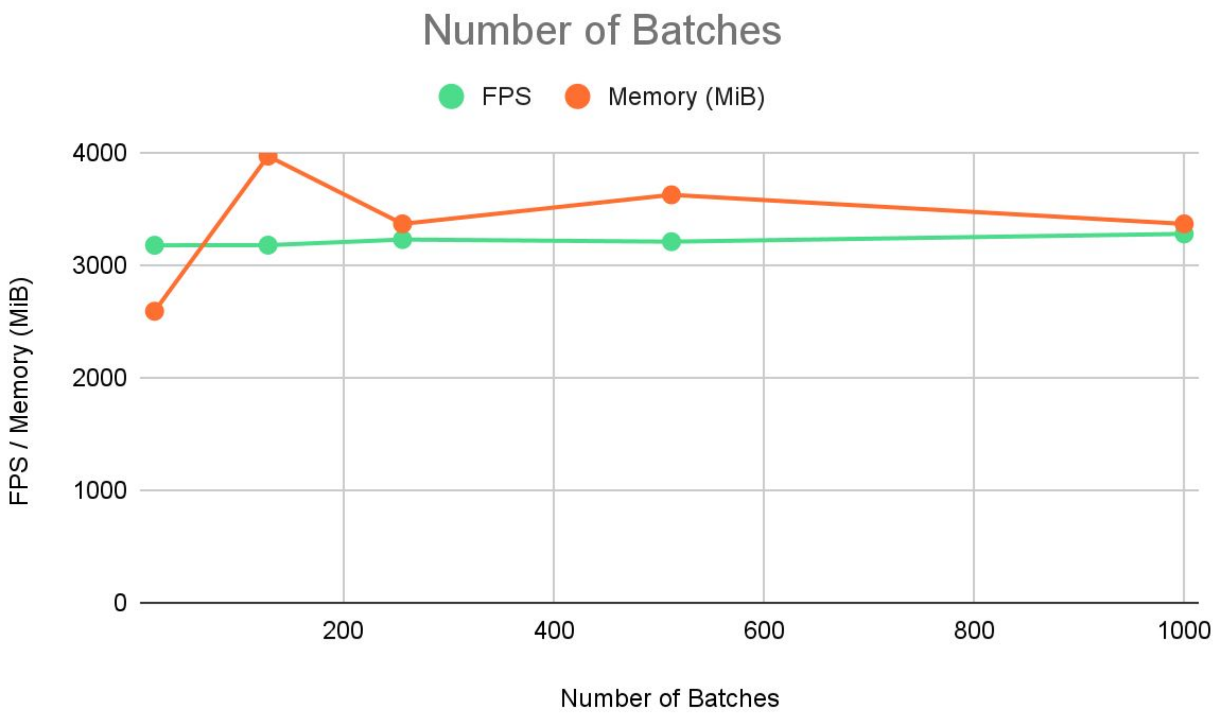
CPU vs GPU


Training Time and FPS
Time (seconds)
FPS
8000
6000
4000
2000
0
7,123
1480
1,413
3663
CPU
GPU

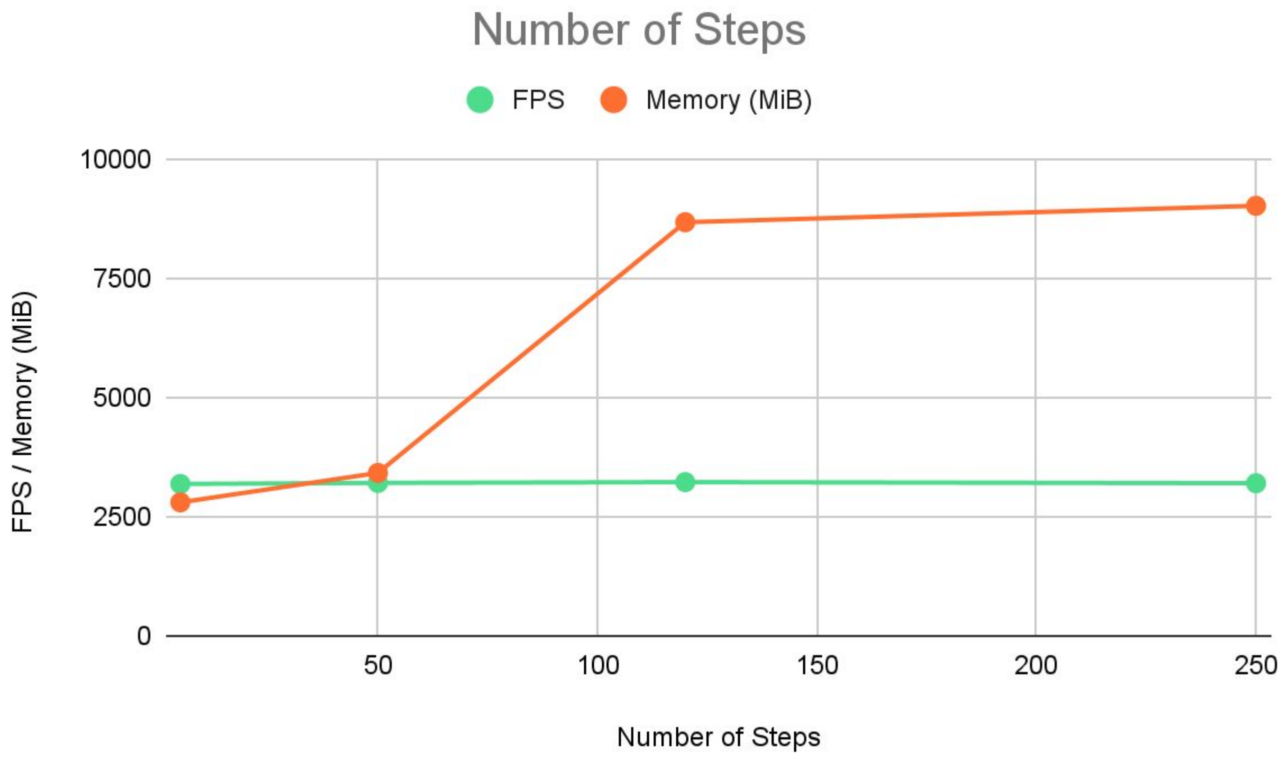
Number of Atari Learning Environments



Number of Batches



Number of Steps



Discussion

- Near-Linear Scaling for GPU Acceleration of Environment
- Limited by Memory Bandwidth
- Training quality can be hindered if only optimizing for performance
  - N-Steps improves model quality but higher memory footprint
- Batch_Size has no significant influence
  - Environment Simulation is bottleneck

References

- Wijmans, E., Kadian, A., Morcos, A., Lee, S., Essa, I., Parikh, D., ... & Batra, D. (2019). DD-PPO: Learning near-perfect pointgoal navigators from 2.5 billion frames. arXiv preprint arXiv:1911.00357.
- Tang, Yunhao & Agrawal, Shipra. (2020). Discretizing Continuous Action Space for On-Policy Optimization. Proceedings of the AAAI Conference on Artificial Intelligence. 34. 5981-5988. 10.1609/aaai.v34i04.6059.
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.