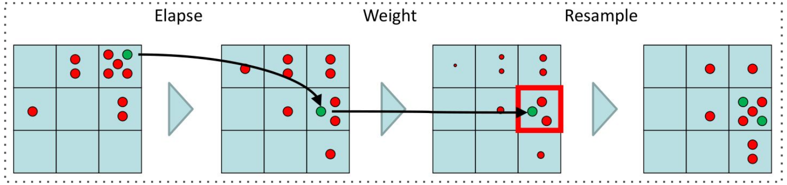
Particle Filter:
Likelihood Kernel

Mateen Ulhaq

Overview

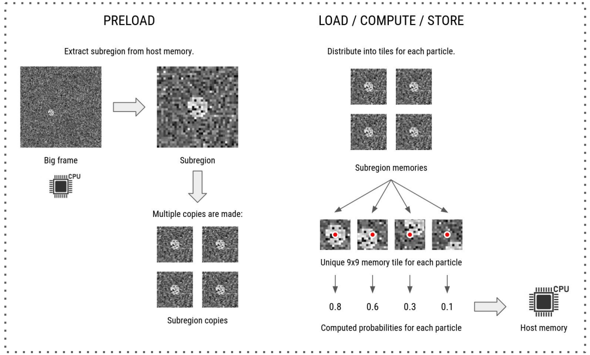
1. Review



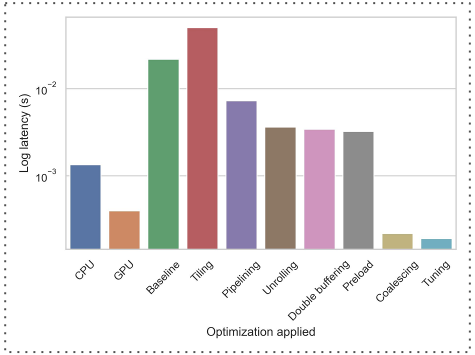
3. Optimizations

```
for (int job_idx = 0;
  switch (job_idx % 2)
    case 0:
      load(job_idx - 0
      compute(job_idx
      store(job_idx -
      break;
```

2. Architecture



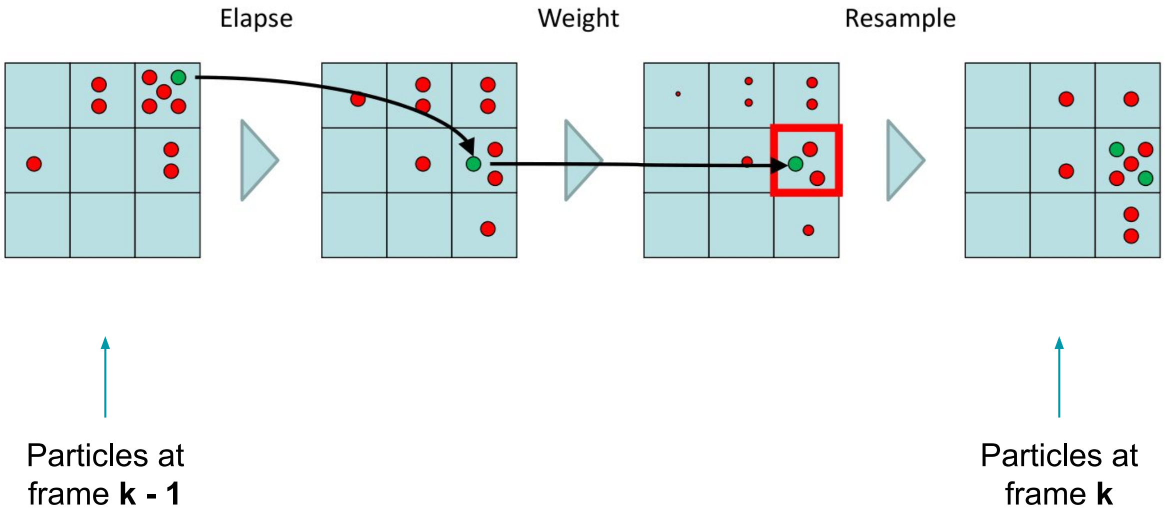
4. Results



Review of particle filter

Main loop:

- Elapse motion model.
- Estimate “likelihood” that particles are in correct positions by checking the video frame.
- Reweight particles.
- Resample particles.

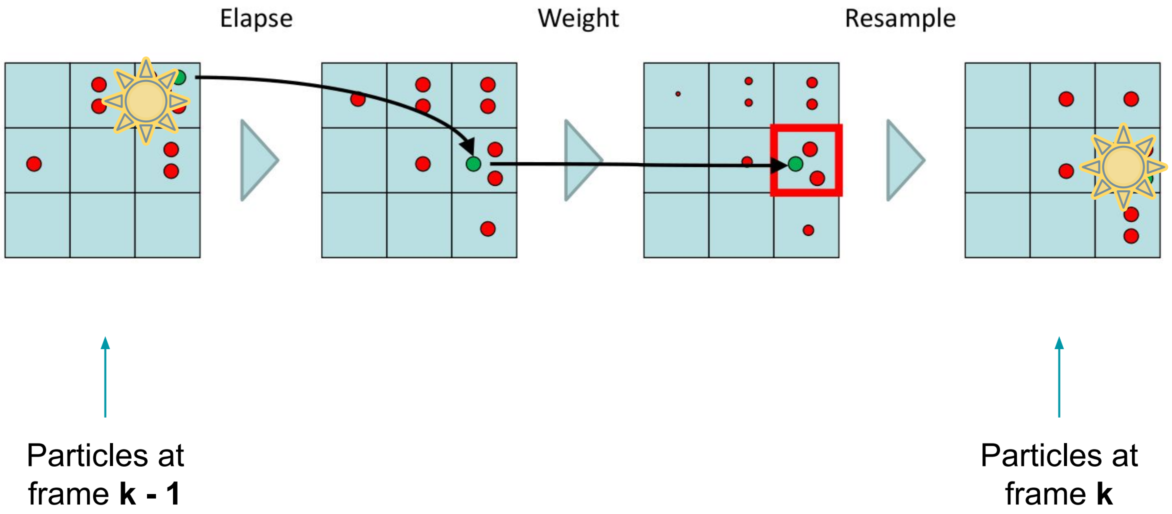


Review of particle filter

Main loop:

- Elapse motion model.
- Estimate “likelihood” that particles are in correct positions by checking the video frame.
- Reweight particles.
- Resample particles.

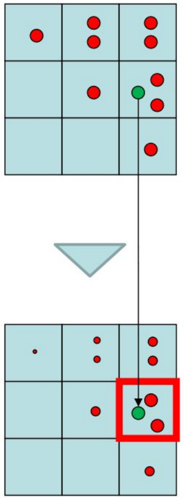
Estimated object position (xe, ye) is “weighted average” of particles.

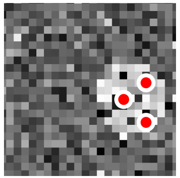


Likelihood

For each particle, check how many pixels are actually "white" in image.

This gives likelihood that the particle is in the center of the white object.





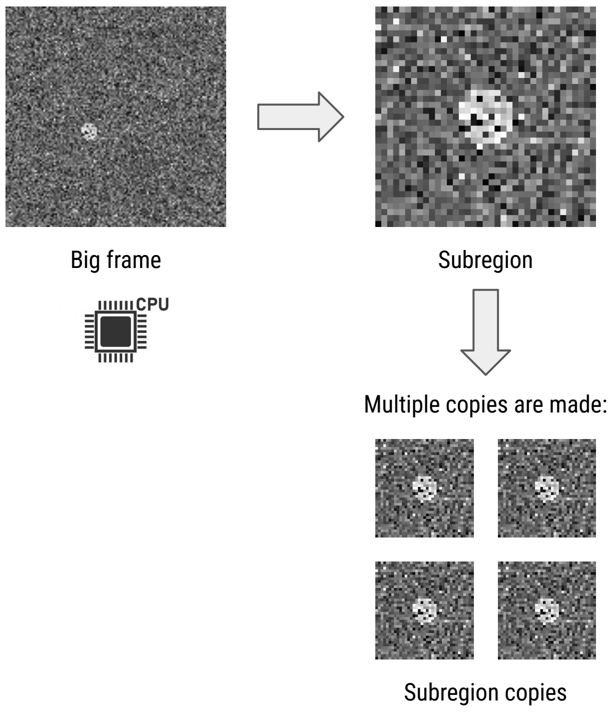
```
for (int j = 0; j < NUM_PARTICLES; j++) {
  float p = 0.0;

  for (int i = 0; i < NUM_OBJ_PIXELS; i++) {
    int x = particleX[j] + objXYoffsets[i * 2 + 1];
    int y = particleY[j] + objXYoffsets[i * 2];
    int xyIndex = fabs(y * FRAME_WIDTH + x);
    xyIndex = xyIndex >= FRAME_HEIGHT * FRAME_WIDTH ? 0 : xyIndex;
    int pixel_intensity = I[xyIndex];

    int b = pixel_intensity - INTENSITY_BLACK;
    int w = pixel_intensity - INTENSITY_WHITE;
    p += b * b - w * w;
  }

  likelihood[j] = p * (1.0 / (LIKELIHOOD_NORMALIZE_FACTOR * NUM_OBJ_PIXELS));
}
```

Architecture

(BEST DESIGN ACHIEVED)

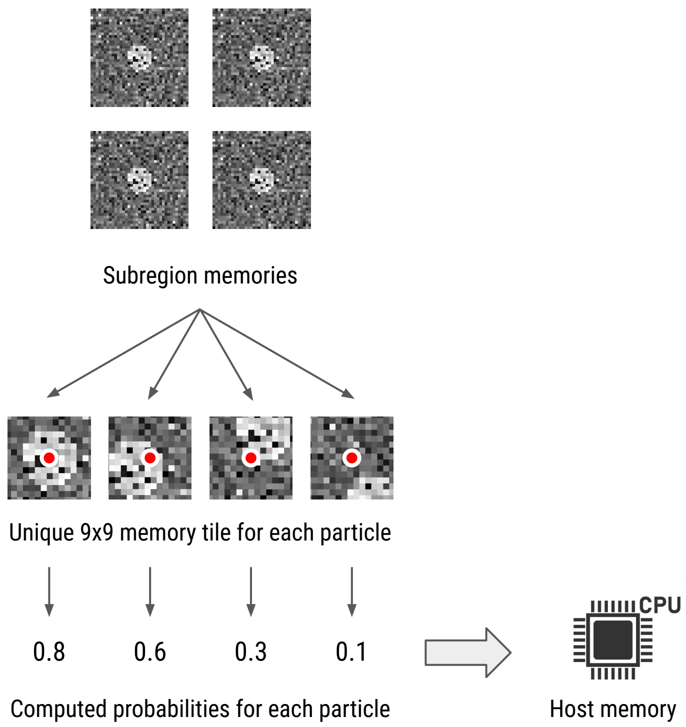
PRELOAD

Extract subregion from host memory.



LOAD / COMPUTE / STORE

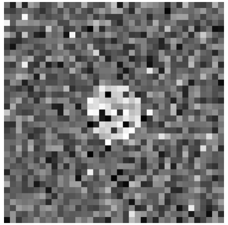
Distribute into tiles for each particle.



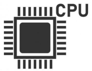
Preload

Extract subregion from host memory.

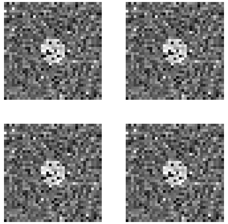
Big frame



Subregion



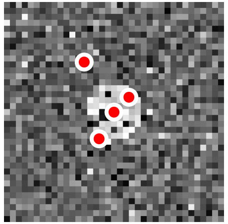
Multiple copies are made:



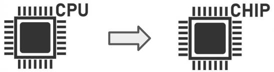
Subregion copies

```
void load_subregion(
  const if_uint8 I[FRAME_HEIGHT * FRAME_WIDTH_COAL],
  uchar I_subreg[SUBREGION_COPIES][SUBREGION_HEIGHT][SUBREGION_WIDTH_COAL],
  const int xsub0, const int ysub0
) {
  // Parallelized definition of the following:
  //
  // load_tile_coalesced(kkk, I_subreg[kkk], x0, y0);

  const int ii = ysub0 / IF_FACTOR_UINT8;
  const int jj = xsub0 / IF_FACTOR_UINT8;
  const int width_factor = IF_WIDTH_UINT8 / IF_WIDTH_UINT8_N;

  for (int i = 0; i < SUBREGION_HEIGHT; i++) {
    for (int js = 0; js < SUBREGION_WIDTH_COAL / width_factor; js++) {
#pragma HLS pipeline
      const if_uint8 z = I[(ii + i) * FRAME_WIDTH_COAL + (jj + js)];

      for (int jd = 0; jd < width_factor; jd++) {
        const uchar z_val =
          z.range((jd + 1) * IF_WIDTH_UINT8_N - 1, jd * IF_WIDTH_UINT8_N);

        for (int kkk = 0; kkk < SUBREGION_COPIES; kkk++) {
          I_subreg[kkk][i][js * width_factor + jd] = z_val;
        }
      }
    }
  }
}
```

Load: initialize



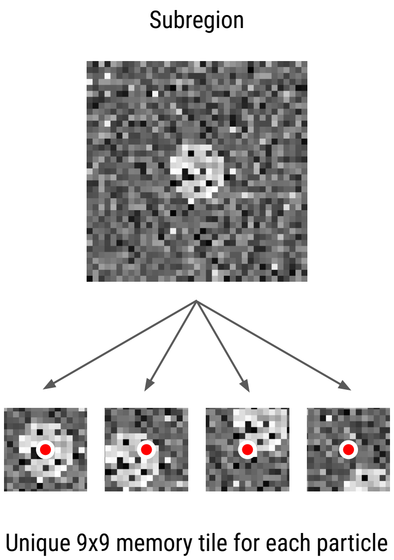
Particle positions copied from host to ensure parallel access.




```
void load(
  const int job_idx,
  const if_uint8_n I[SUBREGION_COPIES][SUBREGION_HEIGHT][SUBREGION_WIDTH_COAL],
  uchar tile[COMPUTE_UNITS][TILE_HEIGHT][TILE_WIDTH],
  float_type p[COMPUTE_UNITS],
  const if_int particleX[NUM_PARTICLES],
  const if_int particleY[NUM_PARTICLES]  //
) {
#pragma HLS inline off

  int_type particleX_buf[COMPUTE_UNITS];
  int_type particleY_buf[COMPUTE_UNITS];

#pragma HLS array_partition variable = particleX_buf complete
#pragma HLS array_partition variable = particleY_buf complete

  // Return if this unit has no work to do.
  if (job_idx < 0 || job_idx >= NUM_JOBS)
    return;

  const int k = job_idx * COMPUTE_UNITS;

LOAD_POSITIONS:
  for (int kk = 0; kk < COMPUTE_UNITS; kk++) {
    int_type x = particleX[k + kk];
    int_type y = particleY[k + kk];
    particleX_buf[kk] = x;
    particleY_buf[kk] = y;
    p[kk] = 0;
  }
```

Load: tiles

Distribute into tiles for each particle.

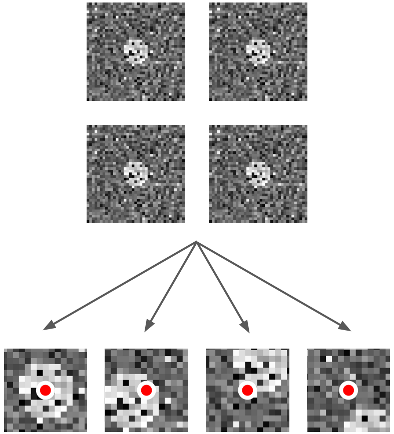



```
LOAD_TILES:
  for (int kk = 0; kk < COMPUTE_UNITS; kk += SUBREGION_COPIES) {
    // Parallelized definition of the following:
    //
    // load_tile<if_uint8_n, TILE_HEIGHT, TILE_WIDTH, SUBREGION_WIDTH_COAL>(
    //   &I[0][0][0],
    //   tile[kk],
    //   y - OBJ_RADIUS,
    //   x - OBJ_RADIUS);

    for (int i = 0; i < TILE_HEIGHT; i++) {
      for (int j = 0; j < TILE_WIDTH; j++) {
  #pragma HLS pipeline
        for (int kkk = 0; kkk < SUBREGION_COPIES; kkk++) {
  #pragma HLS unroll
          const int_type x = particleX_buf[kk + kkk];
          const int_type y = particleY_buf[kk + kkk];
          const int ii = y - OBJ_RADIUS - y0;
          const int jj = x - OBJ_RADIUS - x0;
          const int z = (i + ii >= 0 && i + ii < SUBREGION_HEIGHT)
              && (j + jj >= 0 && j + jj < SUBREGION_WIDTH_COAL)
            ? I[kkk][ii + i][jj + j]
            : INTENSITY_AVERAGE;
          tile[kk + kkk][i][j] = z;
        }
      }
    }
  }
```

Load: tiles

Distribute into tiles for each particle.

Subregion memories

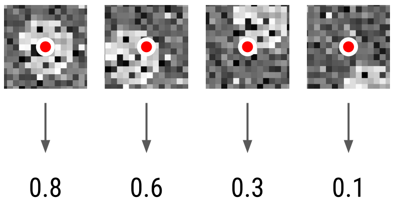


Unique 9x9 memory tile for each particle

```
LOAD_TILES:
  for (int kk = 0; kk < COMPUTE_UNITS; kk += SUBREGION_COPIES) {
    // Parallelized definition of the following:
    //
    // load_tile<if_uint8_n, TILE_HEIGHT, TILE_WIDTH, SUBREGION_WIDTH_COAL>(
    //   &I[0][0][0],
    //   tile[kk],
    //   y - OBJ_RADIUS,
    //   x - OBJ_RADIUS);

    for (int i = 0; i < TILE_HEIGHT; i++) {
      for (int j = 0; j < TILE_WIDTH; j++) {
  #pragma HLS pipeline
        for (int kkk = 0; kkk < SUBREGION_COPIES; kkk++) {
  #pragma HLS unroll
          const int_type x = particleX_buf[kk + kkk];
          const int_type y = particleY_buf[kk + kkk];
          const int ii = y - OBJ_RADIUS - y0;
          const int jj = x - OBJ_RADIUS - x0;
          const int z = (i + ii >= 0 && i + ii < SUBREGION_HEIGHT)
            && (j + jj >= 0 && j + jj < SUBREGION_WIDTH_COAL)
            ? I[kkk][ii + i][jj + j]
            : INTENSITY_AVERAGE;
          tile[kk + kkk][i][j] = z;
        }
      }
    }
  }
```

Compute

Unique 9x9 memory tile for each particle



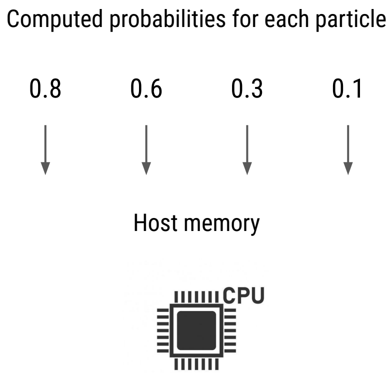
Computed probabilities for each particle

```
void compute(
  const int job_idx,
  float_type p[COMPUTE_UNITS],
  const uchar tile[COMPUTE_UNITS][TILE_HEIGHT][TILE_WIDTH],
  const if_int objXYoffsets[NUM_OBJ_PIXELS * 2]
) {
  // Return if this unit has no work to do.
  if (job_idx < 0 || job_idx >= NUM_JOBS)
    return;

  for (int i = 0; i < NUM_OBJ_PIXELS; i++) {
#pragma HLS pipeline II = 5
    int x = objXYoffsets[i * 2 + 1] + OBJ_RADIUS;
    int y = objXYoffsets[i * 2 + 0] + OBJ_RADIUS;

    for (int kk = 0; kk < COMPUTE_UNITS; kk++) {
#pragma HLS unroll
      uint8_type pixel_intensity = tile[kk][y][x];
      int b = pixel_intensity - INTENSITY_BLACK;
      int w = pixel_intensity - INTENSITY_WHITE;
      p[kk] += (b * b - w * w) * LIKELIHOOD_FACTOR;
    }
  }
}
```

Unrolled to process particles in parallel.

Memory partitioned to prevent particle interference.

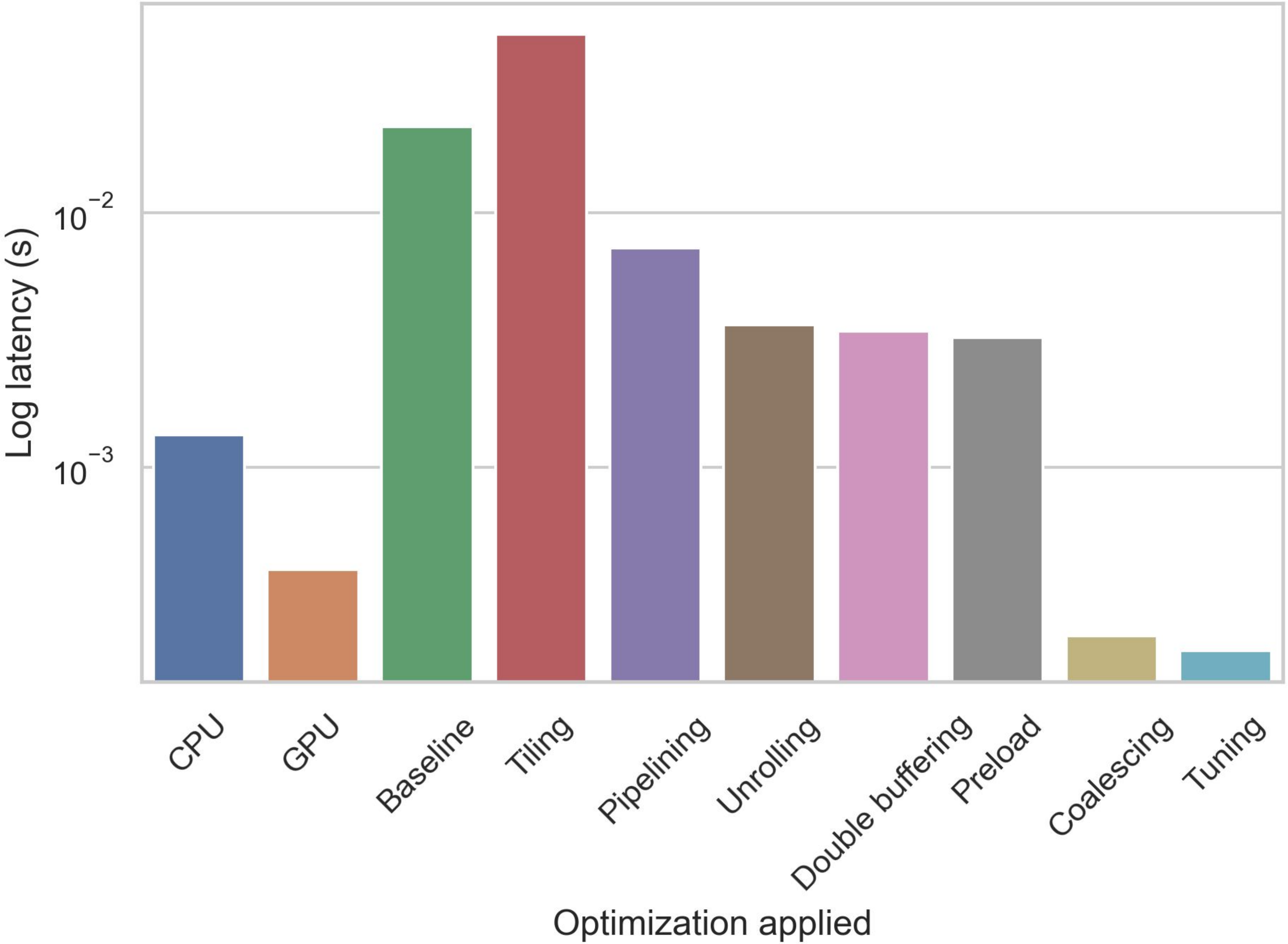
Store




```
void store(
  const int job_idx,
  if_float likelihood[NUM_PARTICLES],
  const float_type p[COMPUTE_UNITS]  //
) {
#pragma HLS inline off

  // Return if this unit has no work to do.
  if (job_idx < 0 || job_idx >= NUM_JOBS)
    return;

  const int k = job_idx * COMPUTE_UNITS;

STORE:
  for (int kk = 0; kk < COMPUTE_UNITS; kk++) {
    likelihood[k + kk] = p[kk];
  }
}
```

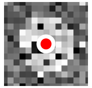
Optimizations

(STEP-BY-STEP)


Log latency (s)
10^{-2}
10^{-3}
CPU
GPU
Baseline
Tiling
Pipelining
Unrolling
Double buffering
Preload
Coalescing
Tuning
Optimization applied

Base

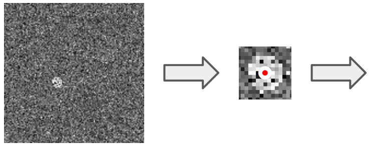
Loop over "object" region around each particle.



```
void calculate_likelihood(
    if_float likelihood[NUM_PARTICLES],
    const if_uint8 I[FRAME_HEIGHT * FRAME_WIDTH],
    const if_int particleX[NUM_PARTICLES],
    const if_int particleY[NUM_PARTICLES],
    const if_int objXYoffsets[NUM_OBJ_PIXELS * 2]
) {
    for (int j = 0; j < NUM_PARTICLES; j++) {
        float_type p = 0.0;

        for (int i = 0; i < NUM_OBJ_PIXELS; i++) {
            int x = particleX[j] + objXYoffsets[i * 2 + 1];
            int y = particleY[j] + objXYoffsets[i * 2];
            int xyIndex = fabs(y * FRAME_WIDTH + x);
            xyIndex = xyIndex >= FRAME_HEIGHT * FRAME_WIDTH ? 0 : xyIndex;
            uint8_type pixel_intensity = I[xyIndex];

            int b = pixel_intensity - INTENSITY_BLACK;
            int w = pixel_intensity - INTENSITY_WHITE;
            p += b * b - w * w;
        }

        likelihood[j] = p * (1.0 / (LIKELIHOOD_NORMALIZE_FACTOR * NUM_OBJ_PIXELS));
    }
}
```

Tiling

```
void calculate_likelihood(
  if_float likelihood[NUM_PARTICLES],
  const if_uint8 I[FRAME_HEIGHT * FRAME_WIDTH],
  const if_int particleX[NUM_PARTICLES],
  const if_int particleY[NUM_PARTICLES],
  const if_int objXYoffsets[NUM_OBJ_PIXELS * 2]  //
) {
  if_uint8 tile[TILE_HEIGHT][TILE_WIDTH];

  for (int k = 0; k < NUM_PARTICLES; k++) {
    int_type x = particleX[k];
    int_type y = particleY[k];

    load(I, tile, x, y);
    float_type p = compute(tile, objXYoffsets, x, y);
    likelihood[k] = p;
  }
}
```



0.8

Big frame

```
void load(
  const if_uint8 I[FRAME_HEIGHT * FRAME_WIDTH],
  if_uint8 tile[TILE_HEIGHT][TILE_WIDTH],
  const int x, const int y  //
) {
  int x0 = x - OBJ_RADIUS;
  int y0 = y - OBJ_RADIUS;
  load_tile<if_uint8, TILE_HEIGHT, TILE_WIDTH, FRAME_WIDTH>(I, tile, y0, x0);
}
```

Load tile around current particle from host.

```
float_type compute(
  const if_uint8 tile[TILE_HEIGHT][TILE_WIDTH],
  const if_int objXYoffsets[NUM_OBJ_PIXELS * 2],
  const int particleX,
  const int particleY  //
) {
  float_type p = 0.0;

  for (int i = 0; i < NUM_OBJ_PIXELS; i++) {
#pragma HLS pipeline off
    int x = objXYoffsets[i * 2 + 1] + OBJ_RADIUS;
    int y = objXYoffsets[i * 2 + 0] + OBJ_RADIUS;
    uint8_type pixel_intensity = tile[y][x];
    int b = pixel_intensity - INTENSITY_BLACK;
    int w = pixel_intensity - INTENSITY_WHITE;
    p += b * b - w * w;
  }

  return p * (1.0 / (LIKELIHOOD_NORMALIZE_FACTOR * NUM_OBJ_PIXELS));
}
```

Use tile for compute.

Tiling

1D load/store:

```
template<typename T, int DIM>
void load_tile(const T* src, T dst[DIM], const int ii) {
#pragma HLS inline off
LOAD_TILE_1D:
  for (int i = 0; i < DIM; i++) {
    dst[i] = src[ii + i];
  }
}
```

ii is the offset for src[].

```
template<typename T, int DIM>
void store_tile(const T src[DIM], T* dst, const int ii) {
#pragma HLS inline off
STORE_TILE_1D:
  for (int i = 0; i < DIM; i++) {
    dst[ii + i] = src[i];
  }
}
```

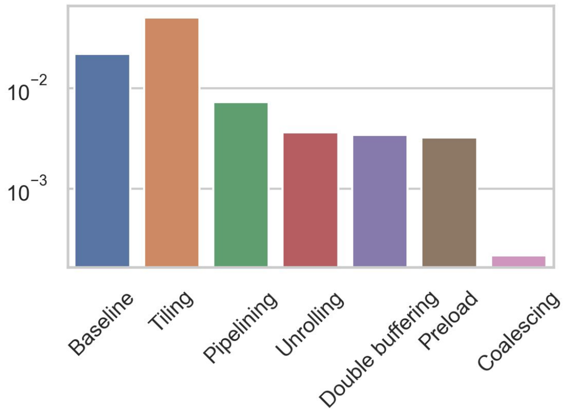
2D load/store:

```
template<typename T, int DIM_I, int DIM_J, int DIM_BIG_J>
void load_tile(const T* src, T dst[DIM_I][DIM_J], const int ii, const int jj) {
#pragma HLS inline off
LOAD_TILE_2D_LI:
  for (int i = 0; i < DIM_I; i++) {
  LJ:
    for (int j = 0; j < DIM_J; j++) {
      dst[i][j] = src[(ii + i) * DIM_BIG_J + (jj + j)];
    }
  }
}
```

```
template<typename T, int DIM_I, int DIM_J, int DIM_BIG_J>
void store_tile(const T src[DIM_I][DIM_J], T* dst, const int ii, const int jj) {
#pragma HLS inline off
STORE_TILE_2D_LI:
  for (int i = 0; i < DIM_I; i++) {
  LJ:
    for (int j = 0; j < DIM_J; j++) {
      dst[(ii + i) * DIM_BIG_J + (jj + j)] = src[i][j];
    }
  }
}
```

Pipelining

II=1 achieved.

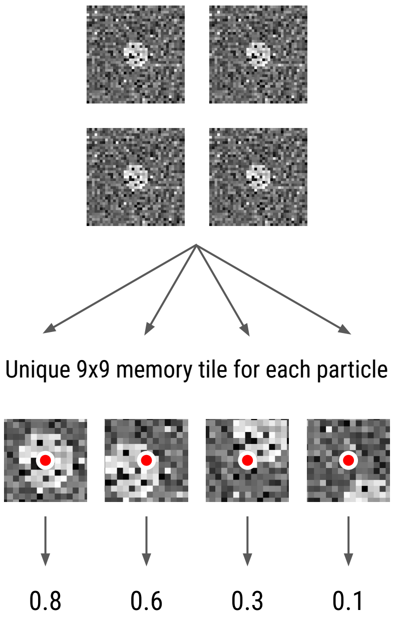



```
float_type compute(
  const if_uint8 tile[TILE_HEIGHT][TILE_WIDTH],
  const if_int objXYoffsets[NUM_OBJ_PIXELS * 2],
  const int particleX,
  const int particleY  //
) {
  float_type p = 0.0;

  for (int i = 0; i < NUM_OBJ_PIXELS; i++) {
#pragma HLS pipeline on
    int x = objXYoffsets[i * 2 + 1] + OBJ_RADIUS;
    int y = objXYoffsets[i * 2 + 0] + OBJ_RADIUS;
    uint8_type pixel_intensity = tile[y][x];
    int b = pixel_intensity - INTENSITY_BLACK;
    int w = pixel_intensity - INTENSITY_WHITE;
    p += b * b - w * w;
  }

  return p * (1.0 / (
    LIKELIHOOD_NORMALIZE_FACTOR * NUM_OBJ_PIXELS));
}
```

Unrolling

Distribute into tiles for each particle.

Subregion memories



Computed probabilities for each particle

Load:

Get position of pixel (i, j).

```
for (int kkk = 0; kkk < SUBREGION_COPIES; kkk++) {
#pragma HLS unroll
    const int_type x = particleX_buf[kk + kkk];
    const int_type y = particleY_buf[kk + kkk];
    const int ii = y - OBJ_RADIUS;
    const int jj = x - OBJ_RADIUS;
    const int z = (i + ii >= 0 && i + ii < SUBREGION_HEIGHT)
        && (j + jj >= 0 && j + jj < SUBREGION_WIDTH_COAL)
      ? I[kkk][ii + i][jj + j]
      : INTENSITY_AVERAGE;
    tile[kk + kkk][i][j] = z;
}
```

Load pixel (i, j) into particle's tile.

Compute:

```
for (int kk = 0; kk < COMPUTE_UNITS; kk++) {
#pragma HLS unroll
    uint8_type pixel_intensity = tile[kk][y][x];
    int b = pixel_intensity - INTENSITY_BLACK;
    int w = pixel_intensity - INTENSITY_WHITE;
    p[kk] += (b * b - w * w) * LIKELIHOOD_FACTOR;
}
```

Unrolled to process particles in parallel.

Memory partitioned to prevent particle interference.

Double buffering

- Currently load-bound.
- If compute-bound, may want to add 3rd buffer to separate compute+store bottleneck.

```
for (int job_idx = 0; job_idx < NUM_JOBS + 1; job_idx++) {
  switch (job_idx % 2) {
    case 0:
      load(job_idx - 0, I, tile[0], p[0], particleX, particleY);
      compute(job_idx - 1, p[1], tile[1], objXYoffsets_buf);
      store(job_idx - 1, likelihood, p[1]);
      break;

    case 1:
      load(job_idx - 0, I, tile[1], p[1], particleX, particleY);
      compute(job_idx - 1, p[0], tile[0], objXYoffsets_buf);
      store(job_idx - 1, likelihood, p[0]);
      break;
  }
}
```

Double buffering

Early exit.

```
void load(
  const int job_idx,
  const if_uint8 I[FRAME_HEIGHT * FRAME_WIDTH],
  if_uint8 tile[COMPUTE_UNITS][TILE_HEIGHT][TILE_WIDTH],
  float_type p[COMPUTE_UNITS],
  const if_int particleX[NUM_PARTICLES],
  const if_int particleY[NUM_PARTICLES]  //
) {
#pragma HLS inline off

  // Return if this unit has no work to do.
  if (job_idx < 0 || job_idx >= NUM_JOBS)
    return;

  const int k = job_idx * COMPUTE_UNITS;

LOAD:
  for (int kk = 0; kk < COMPUTE_UNITS; kk++) {
    int_type x = particleX[k + kk];
    int_type y = particleY[k + kk];
    load_tile<if_uint8, TILE_HEIGHT, TILE_WIDTH, FRAME_WIDTH>(
      I, tile[kk], y - OBJ_RADIUS, x - OBJ_RADIUS);
    p[kk] = 0;
  }
}
```

Coalescing

1D load:

```
template<
  typename Tsrc, typename Tdst,
  int Tsrc_width, int Tdst_width, int DIM>
void load_tile_coalesced(
  const Tsrc* src, Tdst dst[DIM], const int ii
) {
  const int width_factor = Tsrc_width / Tdst_width;
  for (int is = 0; is < DIM / width_factor; is++) {
    const Tsrc z = src[ii + is];
    for (int id = 0; id < width_factor; id++) {
      dst[is * width_factor + id] =
        z.range((id + 1) * Tdst_width - 1, id * Tdst_width);
    }
  }
}
```

2D load:

```
void load_tile_coalesced(
  const Tsrc* src,
  Tdst dst[DIM_I][DIM_J],
  const int ii,
  const int jj  //
) {
  const int width_factor = Tsrc_width / Tdst_width;
  for (int i = 0; i < DIM_I; i++) {
    for (int js = 0; js < DIM_J / width_factor; js++) {
#pragma HLS pipeline
      const Tsrc z = src[(ii + i) * DIM_BIG_J + (jj + js)];
      for (int jd = 0; jd < width_factor; jd++) {
        dst[i][js * width_factor + jd] =
          z.range((jd + 1) * Tdst_width - 1, jd * Tdst_width);
      }
    }
  }
}
```

Coalescing

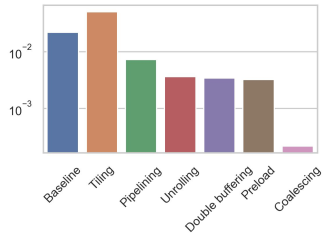
FRAME_WIDTH_COAL = WIDTH / WIDTH_FACTOR

```
void load_subregion(
  const if_uint8 I[FRAME_HEIGHT * FRAME_WIDTH_COAL],
  uchar I_subreg[SUBREGION_COPIES][SUBREGION_HEIGHT][SUBREGION_WIDTH_COAL],
  int x0, int y0
) {
  const int ii = 0;
  const int jj = 0;
  const int width_factor = IF_WIDTH_UINT8 / IF_WIDTH_UINT8_N;
  for (int i = 0; i < SUBREGION_HEIGHT; i++) {
    for (int js = 0; js < SUBREGION_WIDTH_COAL / width_factor; js++) {
#pragma HLS pipeline
      const if_uint8 z = I[(ii + i) * FRAME_WIDTH_COAL + (jj + js)];
      for (int jd = 0; jd < width_factor; jd++) {
        const uchar z_val =
          z.range((jd + 1) * IF_WIDTH_UINT8_N - 1, jd * IF_WIDTH_UINT8_N);
        for (int kkk = 0; kkk < SUBREGION_COPIES; kkk++) {
          I_subreg[kkk][i][js * width_factor + jd] = z_val;
        }
      }
    }
  }
}
```

Make copies of memory for faster tile load stage.

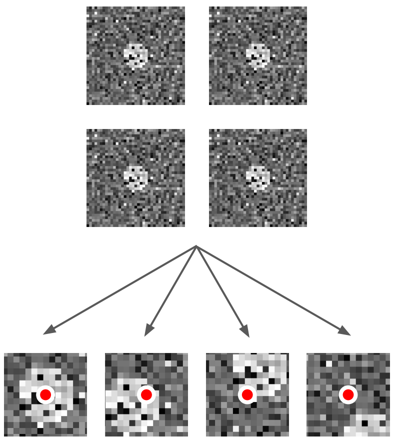
Multiple memories mean we can load multiple particle tiles in parallel.

Costs only memory.



Distribute into tiles for each particle.

Subregion memories



Unique 9x9 memory tile for each particle

Use copies of memories for faster tile loading.

Multiple memories mean we can load multiple particle tiles in parallel.

```
LOAD_TILES:
  for (int kk = 0; kk < COMPUTE_UNITS; kk += SUBREGION_COPIES) {
    // Parallelized definition of the following:
    //
    // load_tile<if_uint8_n, TILE_HEIGHT, TILE_WIDTH, SUBREGION_WIDTH_COAL>(
    //   &I[0][0][0],
    //   tile[kk],
    //   y - OBJ_RADIUS,
    //   x - OBJ_RADIUS);

    for (int i = 0; i < TILE_HEIGHT; i++) {
      for (int j = 0; j < TILE_WIDTH; j++) {
  #pragma HLS pipeline
        for (int kkk = 0; kkk < SUBREGION_COPIES; kkk++) {
  #pragma HLS unroll
          const int_type x = particleX_buf[kk + kkk];
          const int_type y = particleY_buf[kk + kkk];
          const int ii = y - OBJ_RADIUS - y0;
          const int jj = x - OBJ_RADIUS - x0;
          const int z = (i + ii >= 0 && i + ii < SUBREGION_HEIGHT)
              && (j + jj >= 0 && j + jj < SUBREGION_WIDTH_COAL)
            ? I[kkk][ii + i][jj + j]
            : INTENSITY_AVERAGE;
          tile[kk + kkk][i][j] = z;
        }
      }
    }
  }
```

Loads one tile at a time.
Better idea: load multiple tiles in parallel!

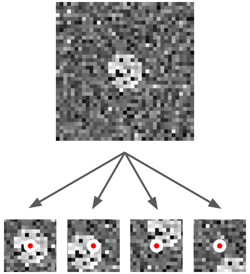
M particles/loop over (P/M) loops.

$O(P/M) < O(P)$

Raster scan loader

load() stage [ALTERNATIVE]

- ONE subregion memory.
- Loops over each pixel in subregion ONCE.
- Good for certain problem sizes (e.g. small subregion or highly scattered particles).



```
// Fast loading for small subregions by looping over each pixel in subregion.
LOAD_TILES:
  for (int i = 0; i < SUBREGION_HEIGHT; i++) {
    for (int ja = 0; ja < SUBREGION_WIDTH_COAL; ja++) {
  #pragma HLS pipeline
      const if_uint8_n z = I[0][i][ja];
      for (int jb = 0; jb < IF_FACTOR_UINT8_N; jb++) {
        const int j = ja * IF_FACTOR_UINT8_N + jb;
        for (int kk = 0; kk < COMPUTE_UNITS; kk++) {
          int_type x = particleX_buf[kk];
          int_type y = particleY_buf[kk];
          int i_t = i - (y - OBJ_RADIUS);
          int j_t = j - (x - OBJ_RADIUS);
          if (i_t >= 0 && i_t < TILE_HEIGHT && j_t >= 0 && j_t < TILE_WIDTH)
            tile[kk][i_t][j_t] =
              z.range((jb + 1) * WIDTH_UINT8_N - 1, jb * WIDTH_UINT8_N);
        }
      }
    }
  }
```

P particles over H·W loops.

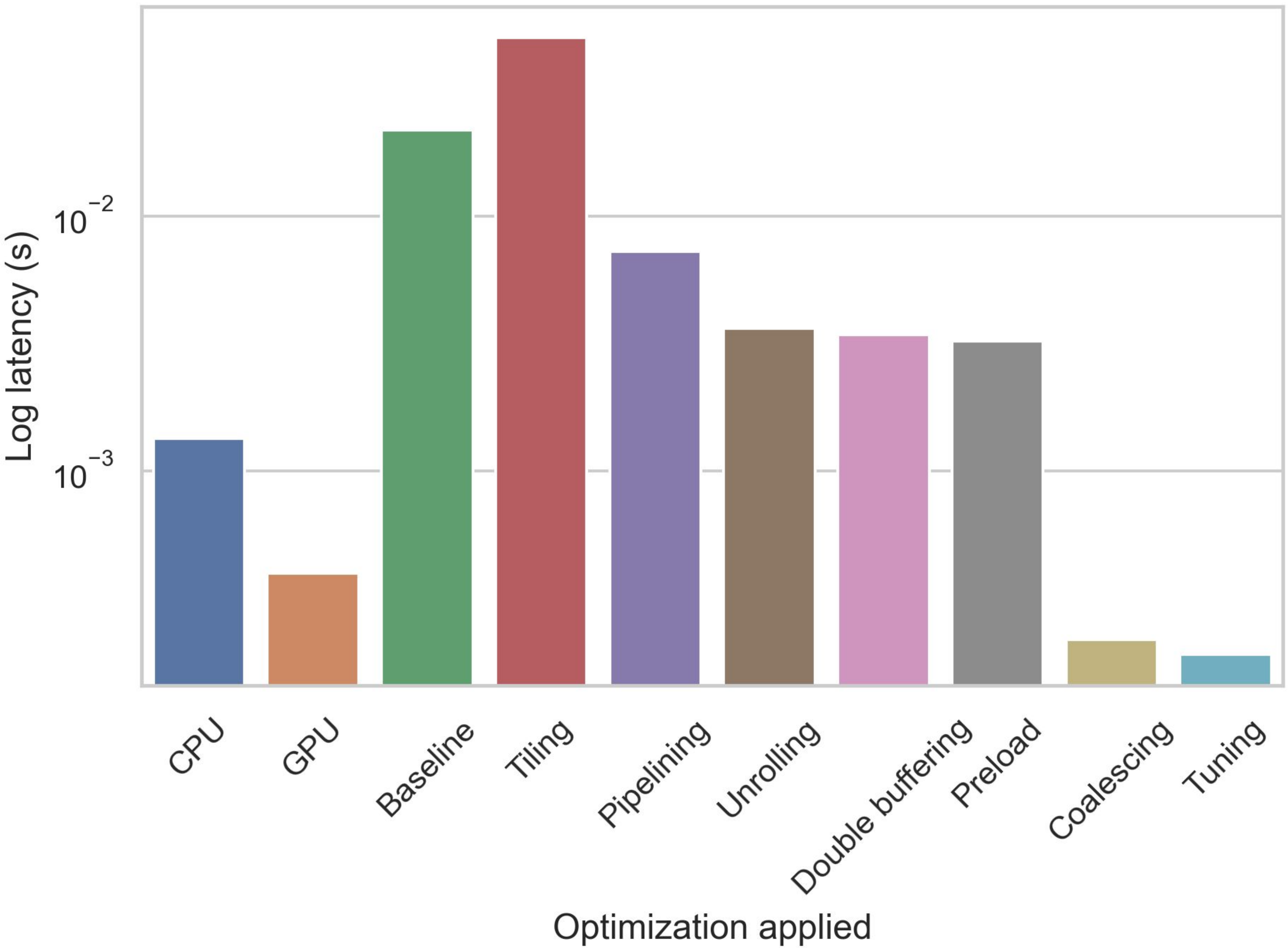
$O(HW) < O(P/M)$

...Sometimes.

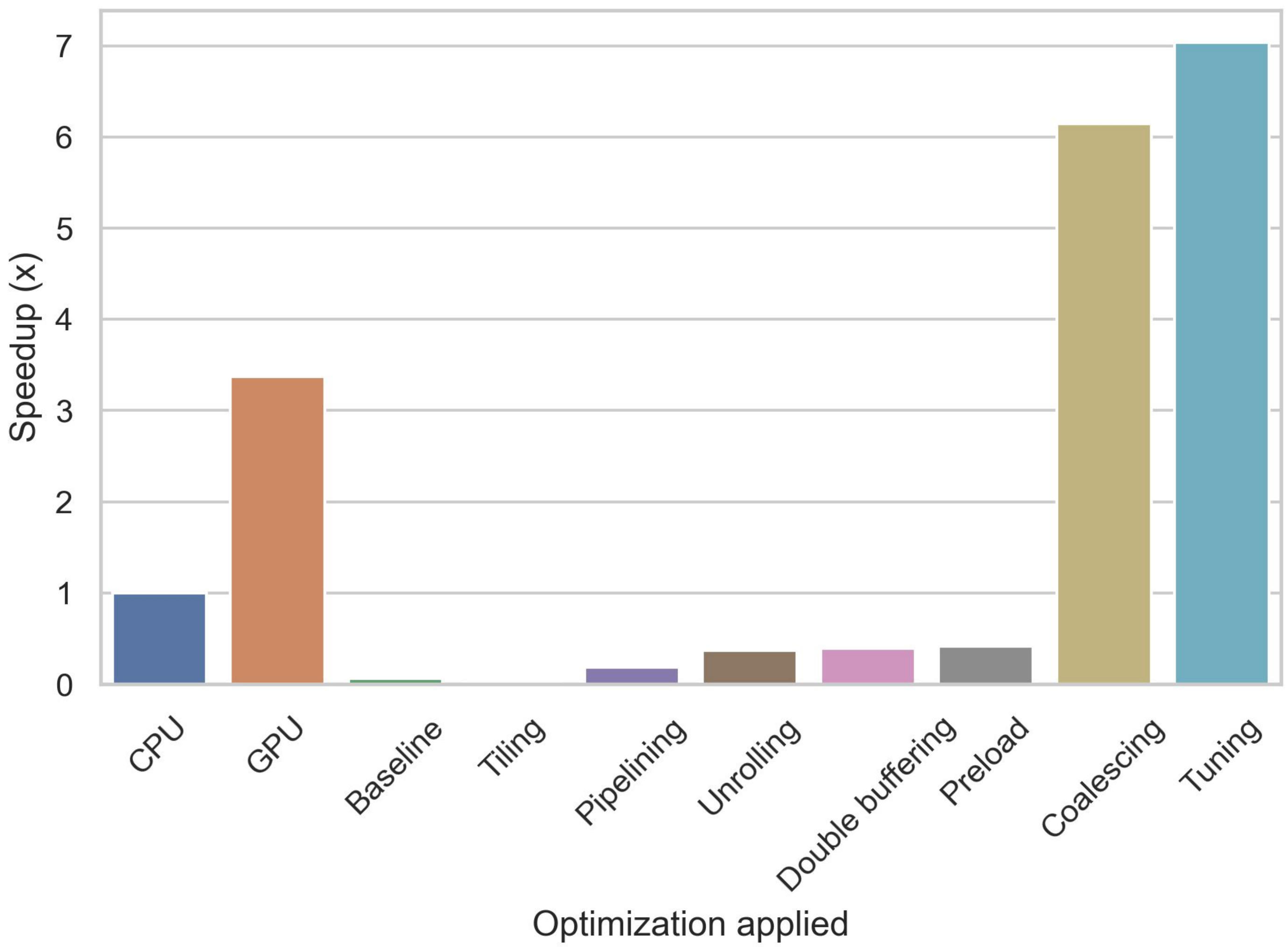
Array partitions

```
uchar tile[2][COMPUTE_UNITS][TILE_HEIGHT][TILE_WIDTH];
float_type p[2][COMPUTE_UNITS];
if_uint8_n I_subreg[SUBREGION_COPIES][SUBREGION_HEIGHT][SUBREGION_WIDTH_COAL];
int_type particleX_buf[COMPUTE_UNITS];
int_type particleY_buf[COMPUTE_UNITS];

#pragma HLS array_partition variable = tile complete dim = 1
#pragma HLS array_partition variable = tile complete dim = 2
#pragma HLS array_partition variable = p complete
#pragma HLS array_partition variable = I_subreg complete dim = 1
#pragma HLS array_partition variable = particleX_buf complete
#pragma HLS array_partition variable = particleY_buf complete
```

Results



Log latency (s)
10^{-2}
10^{-3}
CPU
GPU
Baseline
Tiling
Pipelining
Unrolling
Double buffering
Preload
Coalescing
Tuning
Optimization applied



Speedup (x)
0
1
2
3
4
5
6
7
CPU
GPU
Baseline
Tiling
Pipelining
Unrolling
Double buffering
Preload
Coalescing
Tuning
Optimization applied

Reports

==

== Utilization Estimates

==

* Summary:

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	43	-
FIFO	-	-	-	-	-
Instance	18	520	114448	120693	-
Memory	804	-	3208	6833	-
Multiplexer	-	-	-	29027	-
Register	-	-	356	-	-
Total	822	520	118012	156596	0
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	57	22	14	39	0
Available	4320	6840	2364480	1182240	960

+ Timing:

* Summary:

Clock	Target	Estimated	Uncertainty
ap_clk	3.33 ns	3.946 ns	0.90 ns

+ Latency:
  * Summary:

Latency (cycles) min	Latency (cycles) max	Latency (absolute) min	Latency (absolute) max	Interval min	Interval max	Pipeline Type
8563	48190	33.790 us	0.190 ms	8564	48191	none

+ Detail:
  * Instance:

Instance	Module	Latency (cycles) min	Latency (cycles) max	Latency (absolute) min	Latency (absolute) max	Interval min	Interval max	Pipeline Type
grp_compute_fu_2550	compute	1	570	3.946 ns	2.249 us	1	570	none
grp_load_subregion_fu_2757	load_subregion	8202	8202	32.365 us	32.365 us	8202	8202	none
grp_load_fu_2964	load	1	294	3.946 ns	1.160 us	1	294	none
grp_store_fu_3377	store	1	209	3.946 ns	0.825 us	1	209	none
grp_load_tile_short_138_s_fu_3386	load_tile_short_138_s	147	147	0.580 us	0.580 us	147	147	none

Cosim (untuned, unroll=100)

```
Report time        : Mon Aug  2 22:39:36 PDT 2021.
Solution           : solution1.
Simulation tool    : xsim.
```

RTL	Status	Latency (Clock Cycles) min	Latency (Clock Cycles) avg	Latency (Clock Cycles) max	Interval (Clock Cycles) min	Interval (Clock Cycles) avg	Interval (Clock Cycles) max	Total Execution Time (Clock Cycles)
VHDL	NA	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	67590	67590	67595	67590	67590	67595	608315

Run on 9 video frames.
⇒ average 67591 cycles/frame

GPU: kernel

```
__device__ double
calcLikelihoodSum(unsigned char* I, int* ind, int numOnes, int index) {
    double likelihoodSum = 0.0;
    int x;
    for (x = 0; x < numOnes; x++)
        likelihoodSum += (pow((double)(I[ind[index * numOnes + x]] - 100), 2)
                        - pow((double)(I[ind[index * numOnes + x]] - 228), 2))
            / 50.0;
    return likelihoodSum;
}
```

Likelihood sum.

```
__global__ void likelihood_kernel(
    double* arrayX, double* arrayY, double* xj, double* yj, double* CDF,
    int* ind, int* objxy, double* likelihood, unsigned char* I,
    double* u, double* weights,
    int Nparticles, int countOnes, int max_size, int k, int IszY, int Nfr,
    int* seed, double* partial_sums
) {
    int block_id = blockIdx.x;
    int i = blockDim.x * block_id + threadIdx.x;
    int y;
    int indX, indY;
    __shared__ double buffer[512];

    if (i < Nparticles) {
        arrayX[i] = xj[i];
        arrayY[i] = yj[i];
        weights[i] = 1 / ((double)(Nparticles));
        arrayX[i] = arrayX[i] + 1.0 + 5.0 * d_randn(seed, i);
        arrayY[i] = arrayY[i] - 2.0 + 2.0 * d_randn(seed, i);
    }

    __syncthreads();
```

“Elapse motion”. **[REMOVED]**

Check if region around particle is white.

```
    if (i < Nparticles) {
        for (y = 0; y < countOnes; y++) {
            indX = dev_round_double(arrayX[i]) + objxy[y * 2 + 1];
            indY = dev_round_double(arrayY[i]) + objxy[y * 2];
            ind[i * countOnes + y] = abs(indX * IszY * Nfr + indY * Nfr + k);
            if (ind[i * countOnes + y] >= max_size)
                ind[i * countOnes + y] = 0;
        }
        likelihood[i] = calcLikelihoodSum(I, ind, countOnes, i);
        likelihood[i] = likelihood[i] / countOnes;
        weights[i] = weights[i] * exp(likelihood[i]);
    }

    buffer[threadIdx.x] = 0.0;

    __syncthreads();

    if (i < Nparticles)
        buffer[threadIdx.x] = weights[i];

    __syncthreads();

    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (threadIdx.x < s)
            buffer[threadIdx.x] += buffer[threadIdx.x + s];
        __syncthreads();
    }

    if (threadIdx.x == 0)
        partial_sums[blockIdx.x] = buffer[0];

    __syncthreads();
}
```

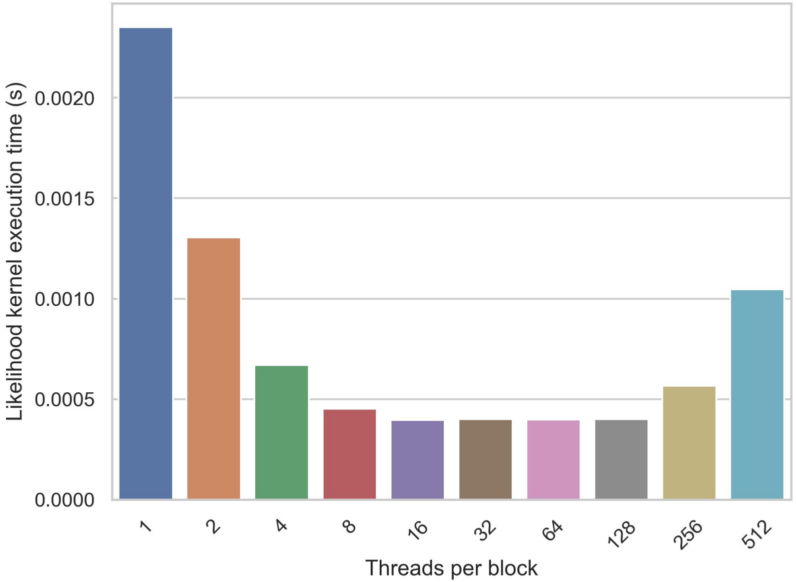
GPU: kernel

Nvidia GTX 1080 Ti GPU (Pascal)	
Number of CUDA cores	3584
Number of SMs	28
Number of cores per SM	128
Max # of blocks per SM	32
Max # of warps per SM	64
# of threads per warp	32
Max # of threads per SM	2048
Shared memory per SM	64KB
Global memory size (GB)	11
Global bandwidth (GB/s)	484

```
int num_blocks = ceil((double)Nparticles / (double)threads_per_block);

for (k = 1; k < Nfr; k++) {
    likelihood_kernel<<<num_blocks, threads_per_block>>>(
        arrayX_GPU, arrayY_GPU,
        xj_GPU, yj_GPU,
        CDF_GPU, ind_GPU, objxy_GPU, likelihood_GPU,
        I_GPU, u_GPU, weights_GPU,
        Nparticles, countOnes,
        max_size, k, IszY, Nfr, seed_GPU, partial_sums);

    sum_kernel<<<num_blocks, threads_per_block>>>(partial_sums, Nparticles);

    normalize_weights_kernel<<<num_blocks, threads_per_block>>>(
        weights_GPU, Nparticles, partial_sums, CDF_GPU, u_GPU, seed_GPU);

    find_index_kernel<<<num_blocks, threads_per_block>>>(
        arrayX_GPU, arrayY_GPU, CDF_GPU, u_GPU,
        xj_GPU, yj_GPU, weights_GPU, Nparticles);
}

cudaThreadSynchronize();
```

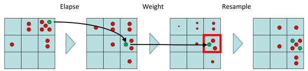
Likelihood kernel.

Other kernels.
[REMOVED]



References

[1] Rodinia Benchmark Suite

[2] Berkeley AI CS188: Lecture Slides (IMAGE) 

[3] https://www.codeproject.com/Articles/865934/Object-Tracking-Particle-Filter-with-Ease (IMAGE)