UNIVERSITÀ DEGLI STUDI FIRENZE

Mean Shift

Parallel Computing Final-Term

Angelo D’Amante
Fabian Greavu

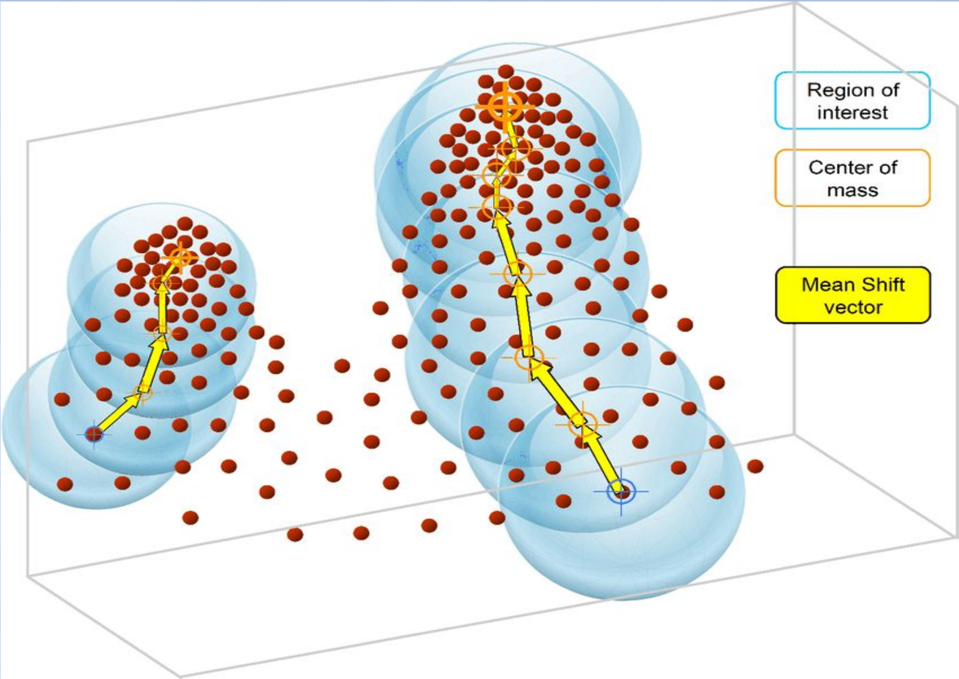
Introduction

Introduction: Mean shift idea

Mean Shift is a non-parametric feature-space analysis technique for locating the maximas of a density function

Applications:
- **clustering**
- **computer vision**
- **image processing**

Main idea: move each point to the center of mass of his surroundings and iterate on the dataset until convergence.



UNIVERSITÀ DEGLI STUDI FIRENZE

Introduction: Mean shift parallelization

Mean shift algorithm has 2 main parts:

1. for 1, 2, …, K:
   shift x ← m(x) for all x in dataset
2. reduce dataset to centroids

Idea:
Step 1 Sequential → Parallel

Introduction: Mean shift squential algorithm

Pseudo-Code Algorithm:

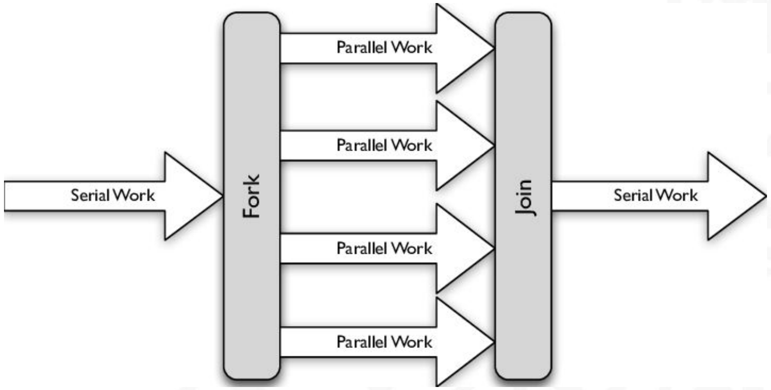
```
new_data = original_points
while iteration < I do
	for p in new_data do
		num, den = 0
		for op in original_points do
			d = Distance(p, op)
			gaussian = K(dist)
			num = num + op * gaussian
			den = den + gaussian
		end for
		p = num/den
	end for
end while
return reduce_centroids(data, eps)
```

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x) x_i}{\sum_{x_i \in N(x)} K(x_i - x)}$$

$$k(x) = e^{-\frac{x^2}{2\sigma^2}},$$

OpenMP Implementation

OpenMP: Introduction



- Advantages of the ***fork-join*** model.
- Only few directives needed for this task.
- Benefit from both ***shared*** and ***private*** memory.

- Exploit both workload sharing mechanism:
  - $\boldsymbol{Static \rightarrow set\ by\ User}$
  - $\boldsymbol{Dynamic \rightarrow set\ by\ OpenMP}$

OpenMP: Solutions

```
#pragma omp parallel for default(none) shared(⋯) schedule(static) num_threads(num_threac
```

```
procedure MEANSHIFT(original_points)
    new_data = original_points
    while iteration < I do
        for p in new_data do
            num = 0
            den = 0
            for op in original_points do
                d = Distance(p, op)
                gaussian = K(dist)
                num = num + op ∗ gaussian
                den = den + gaussian
            end for
            p = num/den
        end for
    end while
    data = reduce_centroids(data, ϵ)
    return data
```

Static Version

Dynamic Version

```
#pragma omp parallel for default(none) shared(⋯) schedule(dynamic)
```

OpenMP: Experiments

Two CPU's architectures examined.

1. **Sequential** and **Parallel** timings are taken by:
   CPU Intel(R) Core(TM) i7-8750H CPU @ 2.2GHz

2. **Sequential** and **Parallel** timings are taken by:
   CPU Intel(R) Core(TM) i7-9700K CPU @ 3.6GHz
   Each core boosts up to 4.6 Ghz OOTB

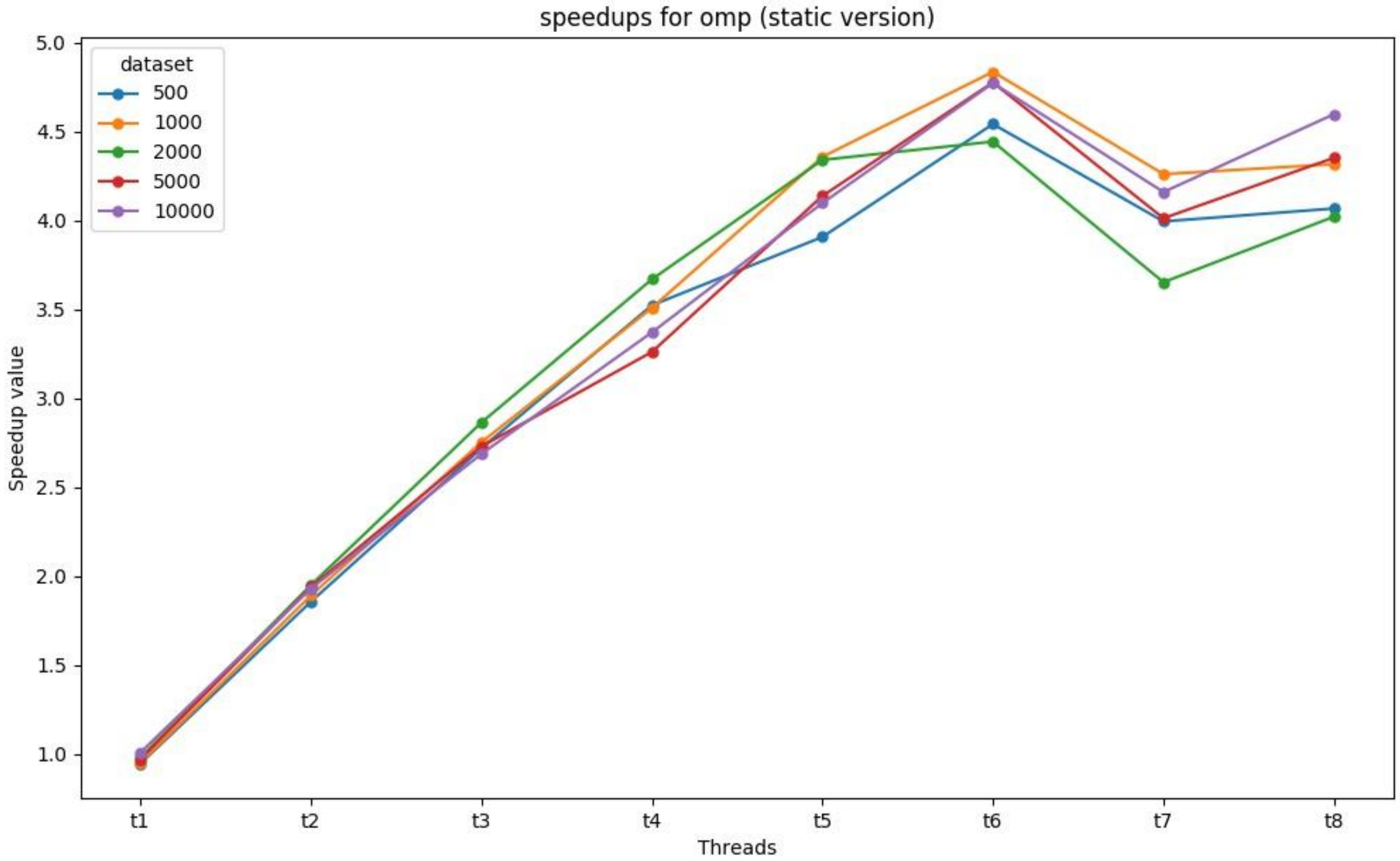
Dataset = {500, 1000, 2000, 5000, 10000}

Versions:

- Static with number of threads in [1, 8]
- Dynamic Version

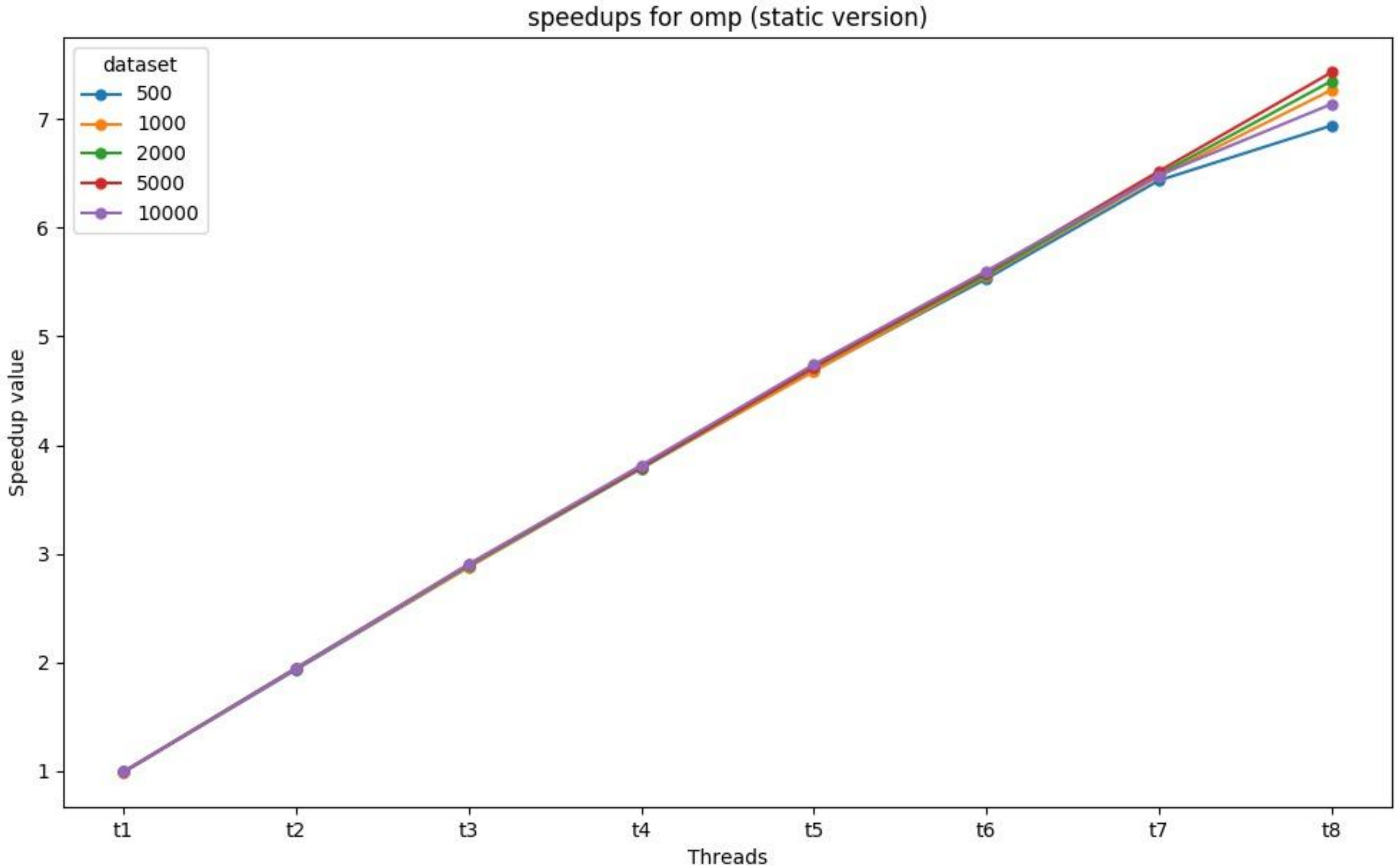
UNIVERSITÀ DEGLI STUDI FIRENZE

OpenMP: Results



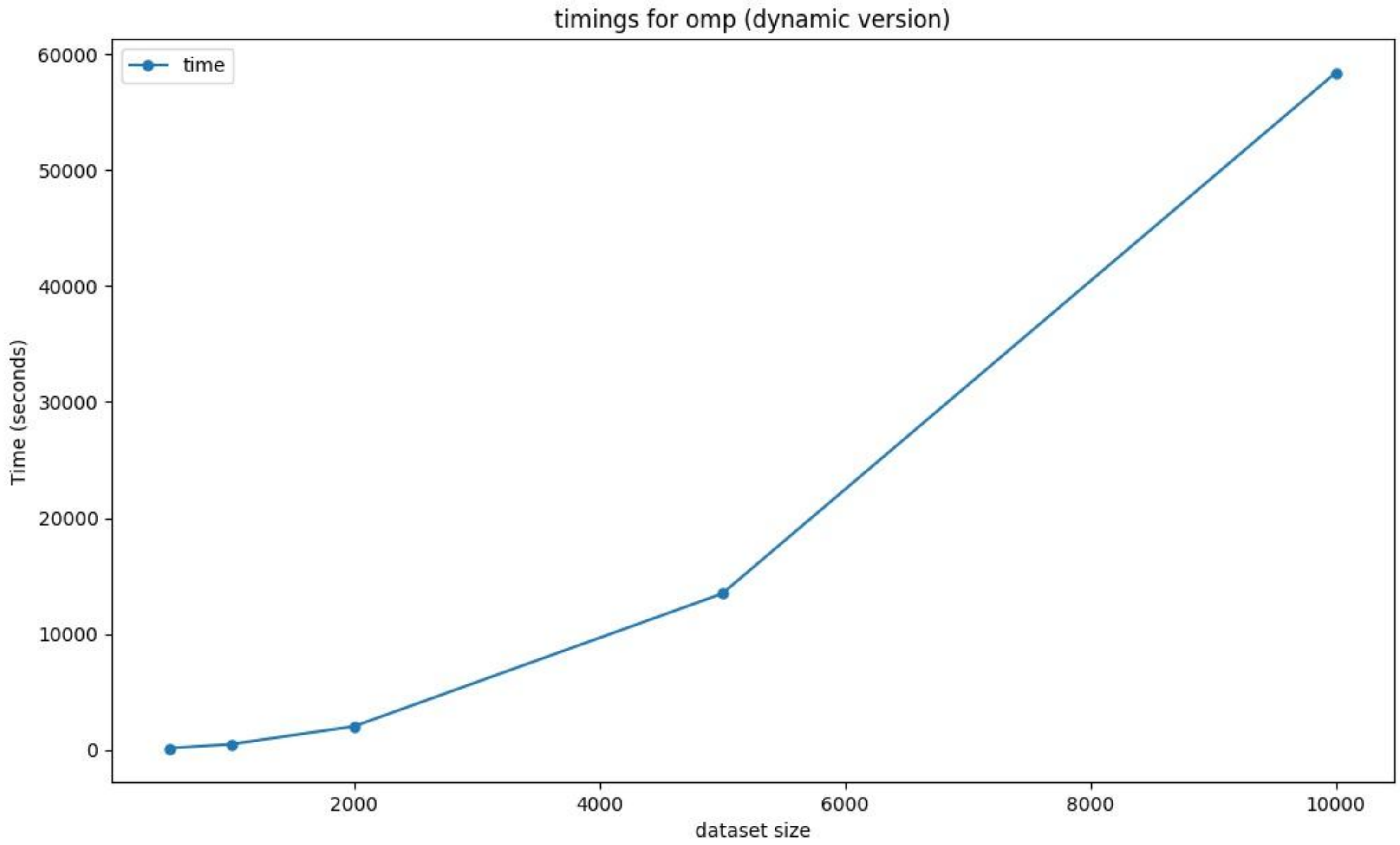
CPU Intel(R) Core(TM) i7-8750H CPU @ 2.2GHz

OpenMP: Results



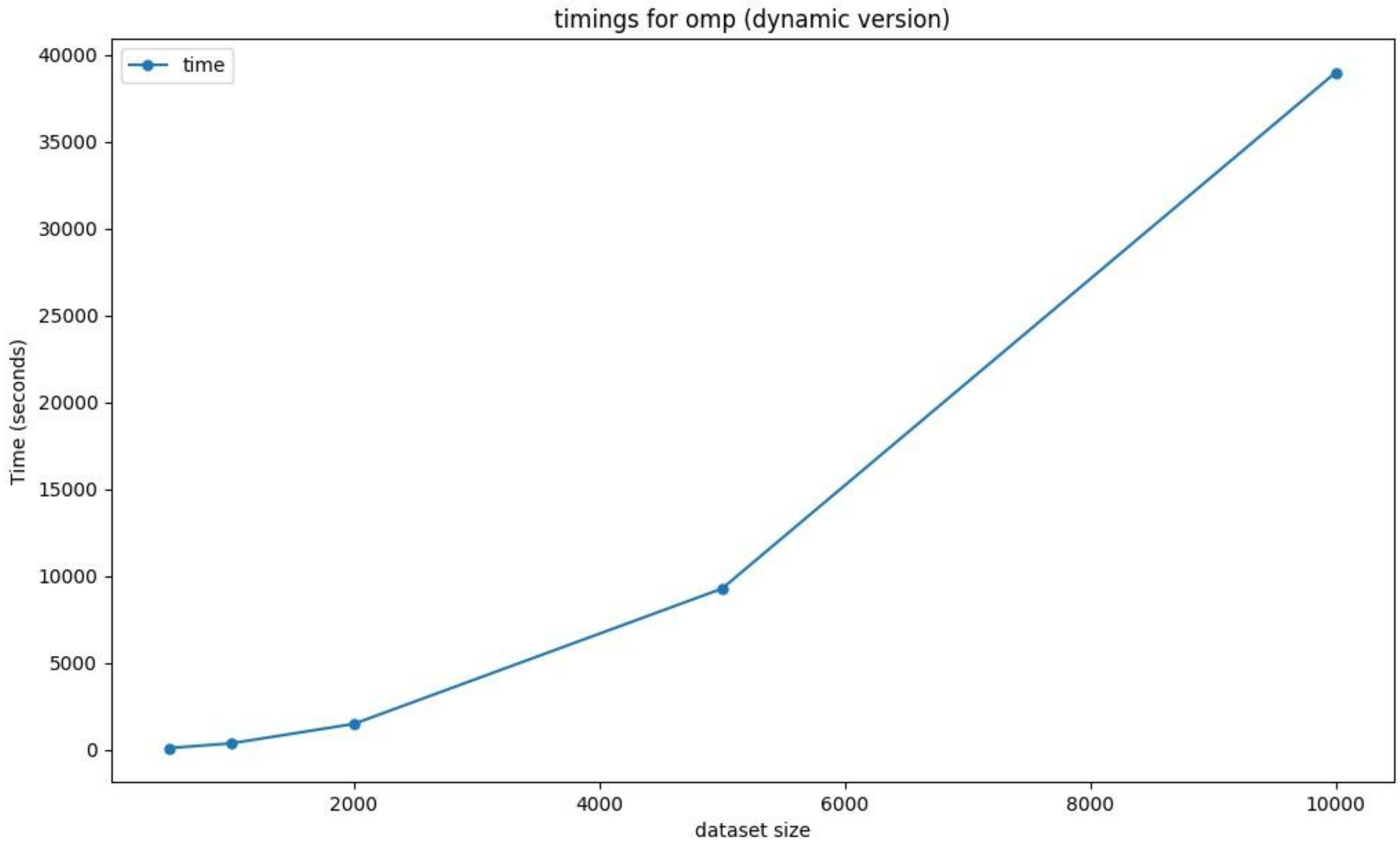
CPU Intel(R) Core(TM) i7-9700K CPU @ 3.6GHz

OpenMP: Results



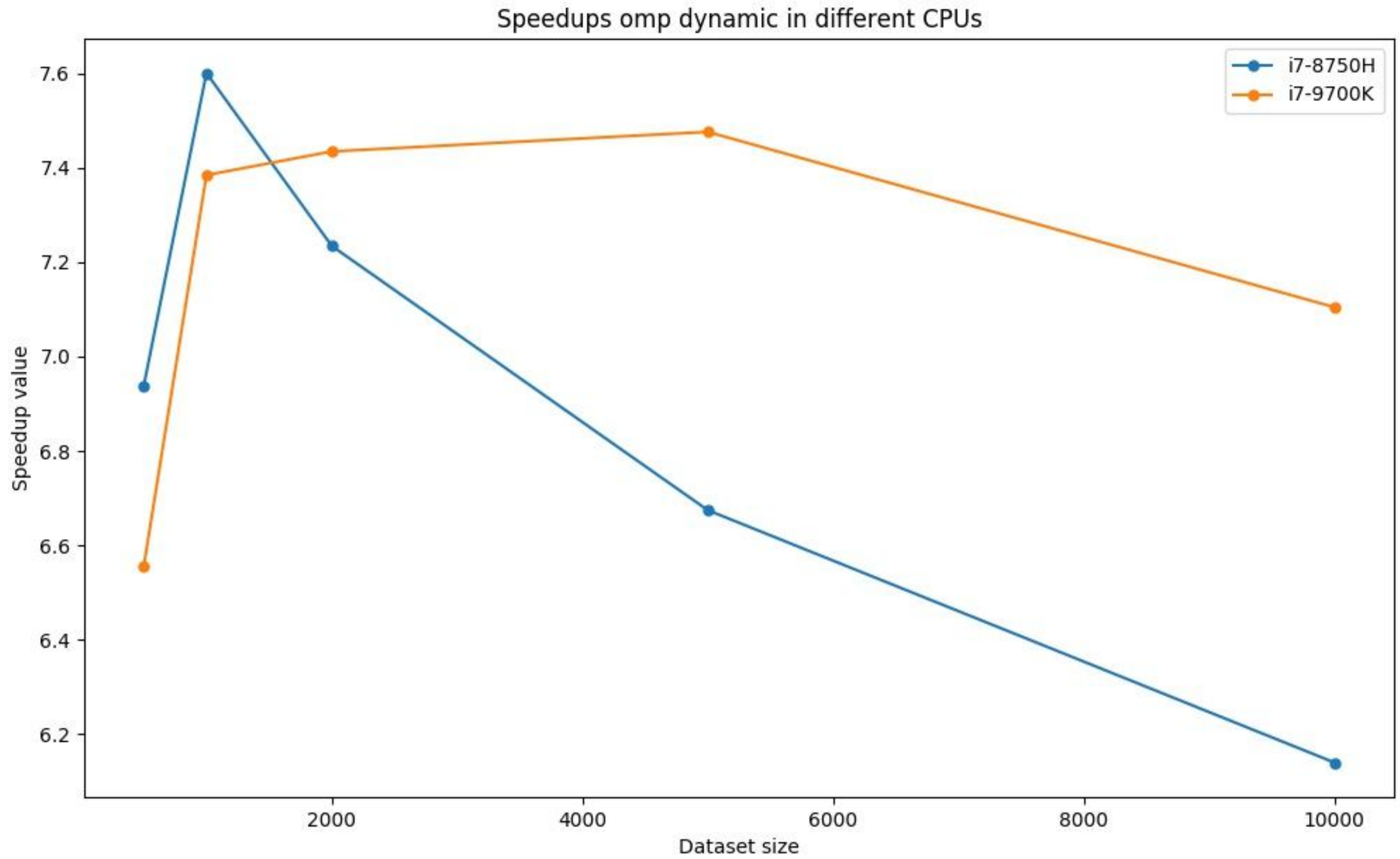
CPU Intel(R) Core(TM) i7-8750H CPU @ 2.2GHz

OpenMP: Results



CPU Intel(R) Core(TM) i7-9700K CPU @ 3.6GHz

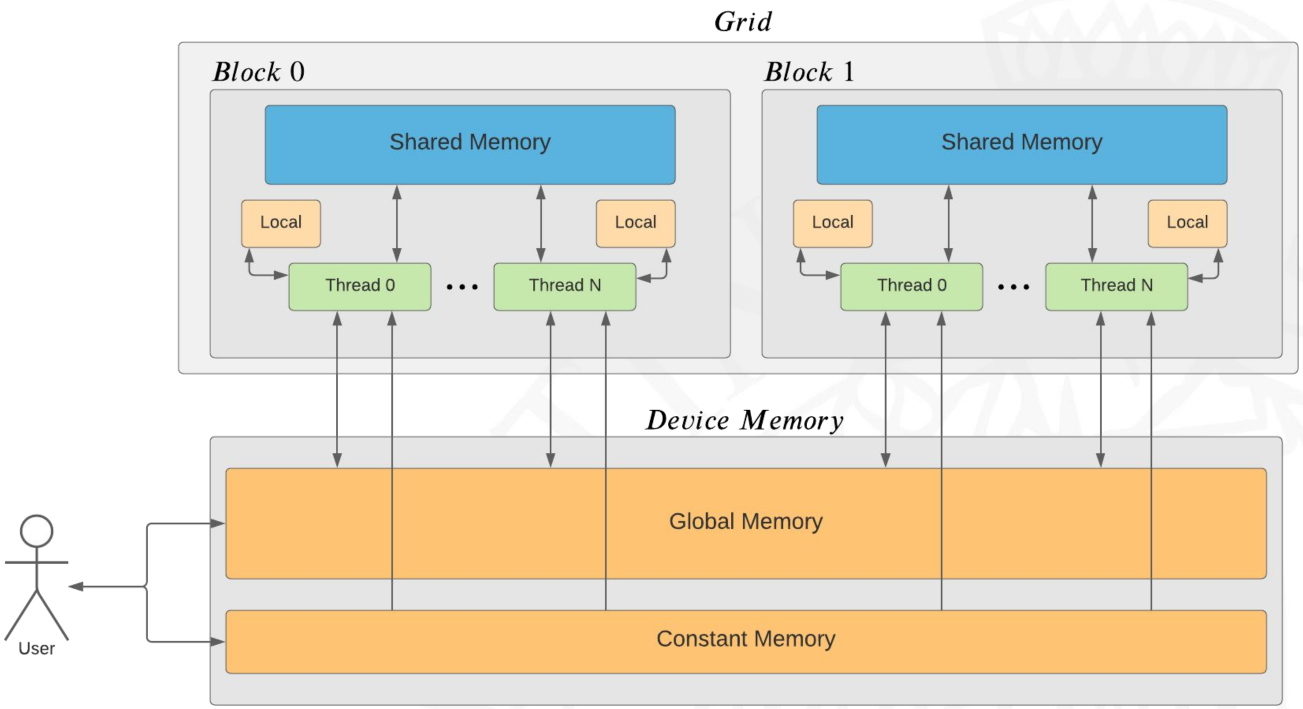
OpenMP: Results



CPU Intel(R) Core(TM) i7-8750H CPU @ 2.2GHz vs CPU Intel(R) Core(TM) i7-9700K CPU @ 3.6GHz

CUDA Implementation

CUDA: Introduction



- **SIMT** (Single Instruction Multiple Thread) model.
- User decides number of threads $T \rightarrow \left\lceil \frac{N}{T} \right\rceil$ blocks.
- Exploit both **global** and **shared** memory access.

CUDA: Solutions

Naive:

```
csv = load_csv( … );

float *dev_data, *dev_data_tmp;
size_t data_bytes = POINTS_NUMBER * D * sizeof(float);
cudaMalloc(&dev_data, data_bytes);
cudaMalloc(&dev_data_tmp, data_bytes);

cudaMemcpy(dev_data, data.data(), data_bytes,
cudaMemcpyHostToDevice);
cudaMemcpy(dev_data_tmp, data_next.data(), data_bytes,
cudaMemcpyHostToDevice);

for (size_t i = 0; i < NUM_ITER; ++i) {
        compute_weights_naive_kernel<<<BLOCKS,
TW>>>(dev_data, dev_data_tmp, POINTS_NUMBER);
        cudaDeviceSynchronize();
        temp_data = dev_data;
        dev_data = dev_data_tmp;
        dev_data_tmp = temp_data;
}
cudaMemcpy(data.data(), dev_data, data_bytes,
cudaMemcpyDeviceToHost);
centers = resuce_to_centroids(data, MIN_DISTANCE)
```

Shared Memory:

```
csv = load_csv( … );

float *dev_data, *dev_data_tmp;
size_t data_bytes = POINTS_NUMBER * D * sizeof(float);
cudaMalloc(&dev_data, data_bytes);
cudaMalloc(&dev_data_tmp, data_bytes);

cudaMemcpy(dev_data, data.data(), data_bytes,
cudaMemcpyHostToDevice);
cudaMemcpy(dev_data_tmp, data_next.data(), data_bytes,
cudaMemcpyHostToDevice);

for (size_t i = 0; i < NUM_ITER; ++i) {
        compute_weights_shared_mem_kernel<<<BLOCKS,
TW>>>(dev_data, dev_data_tmp, POINTS_NUMBER,
BLOCKS);
        cudaDeviceSynchronize();
        temp_data = dev_data;
        dev_data = dev_data_tmp;
        dev_data_tmp = temp_data;
}
cudaMemcpy(data.data(), dev_data, data_bytes,
cudaMemcpyDeviceToHost);
centers = resuce_to_centroids(data, MIN_DISTANCE)
```

```
const int TW = 64;
const int BLOCKS = (POINTS_NUMBER + TW - 1) / TW;
```

CUDA: Solutions

Naive:

```
__global__ void naive_kernel(data, data_tmp,
POINTS_NUMBER){
  int tid = (blockIdx.x * blockDim.x) + threadIdx.x;
  int x = tid*2, y = tid*2+1;
  if (tid < POINTS_NUMBER) {
    float new_position = {0., 0.}; // and other variables
    for (int i = 0; i < POINTS_NUMBER; ++i) {
      int xloop = i*2, yloop = i*2+1;
      eucl_dict =// calculate distance data[x], data[xloop]
      if (eucl_dist <= RADIUS) {
        weight = expf(-eucl_dist / BANDWIDTH);
        new_position += ... //update position
      }
    }
    data_tmp[x, y] = new_position // move p
  }
}
```

Shared Memory:

```
const int BLOCKS = (POINTS_NUMBER + TW - 1) / TW;
__global__ void sm_kernel(data, data_tmp, POINTS_NUMBER){
  __shared__ float local_data[TW * 2];
  __shared__ float flag_data[TW];
  int tid = (blockIdx.x * blockDim.x) + threadIdx.x;
  int x = tid*2, y = tid*2+1; // and other variables
  for (int t = 0; t < BLOCKS; ++t) {
    tid_in_tile = t * TILE_WIDTH + threadIdx.x;
    if (tid_in_tile < POINTS_NUMBER) {
      local_data[threadIdx.x * 2] = data[tid_in_tile * 2]
      local_data[threadIdx.x * 2 +1] = data[tid_in_tile * 2 +1]
      flag_data[threadIdx.x] = 1;
    }else{
      flag_data[threadIdx.x] = 0;
    }
    __syncthreads();
    for (int i = 0; i < TW; ++i) {
      int local_x_tile = i*2, local_y_tile = i*2+1;
      eucl_dict = … // calculate distance dtat[x], data[x_in_tile]
      if (eucl_dist <= RADIUS) {
        weight = expf(-eucl_dist / BANDWIDTH);
        new_position += ... //update position data[x], data[local_x_tile]
      }
    }
    __syncthreads();
  }
  if (tid < POINTS_NUMBER) {new_position = ... // move p }
}
```

CUDA: Experiments

- **Sequential Timings** are taken by:

  CPU Intel(R) Core(TM) i7-9700K CPU @ 3.6GHz

  (Boosting at 4.6Ghz ootb)

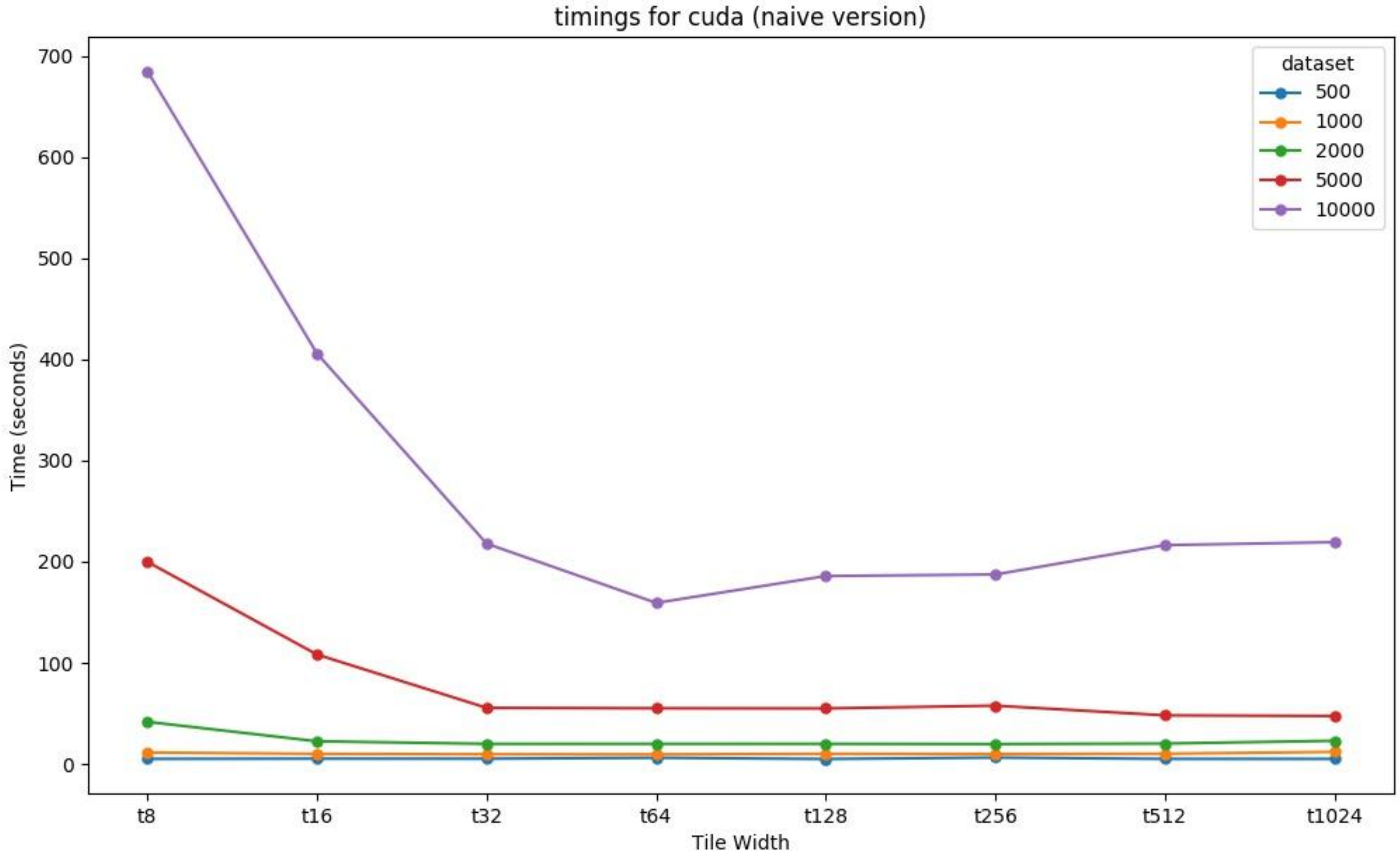
- **Parallel Timings** are taken by:

  GPU NVIDIA GeForce GTX 1050 Ti

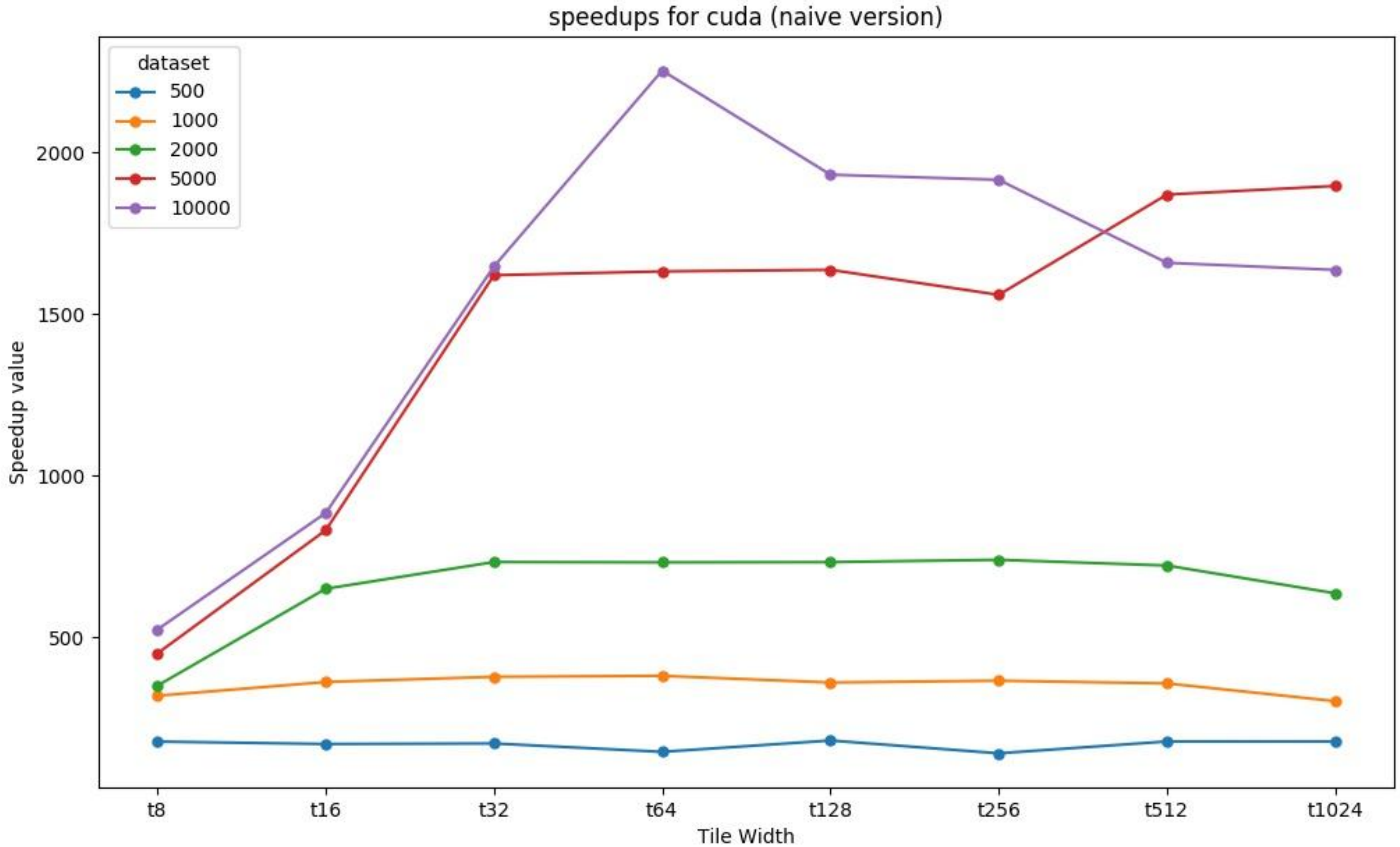
  With 4096 MB dedicated and 768 CUDA Cores

- **Experiments** with:
  - Dataset = {500, 1000, 2000, 5000, 10000}
  - Tile Width = {8, 16, 32, 64, 128, 256, 512, 1024}

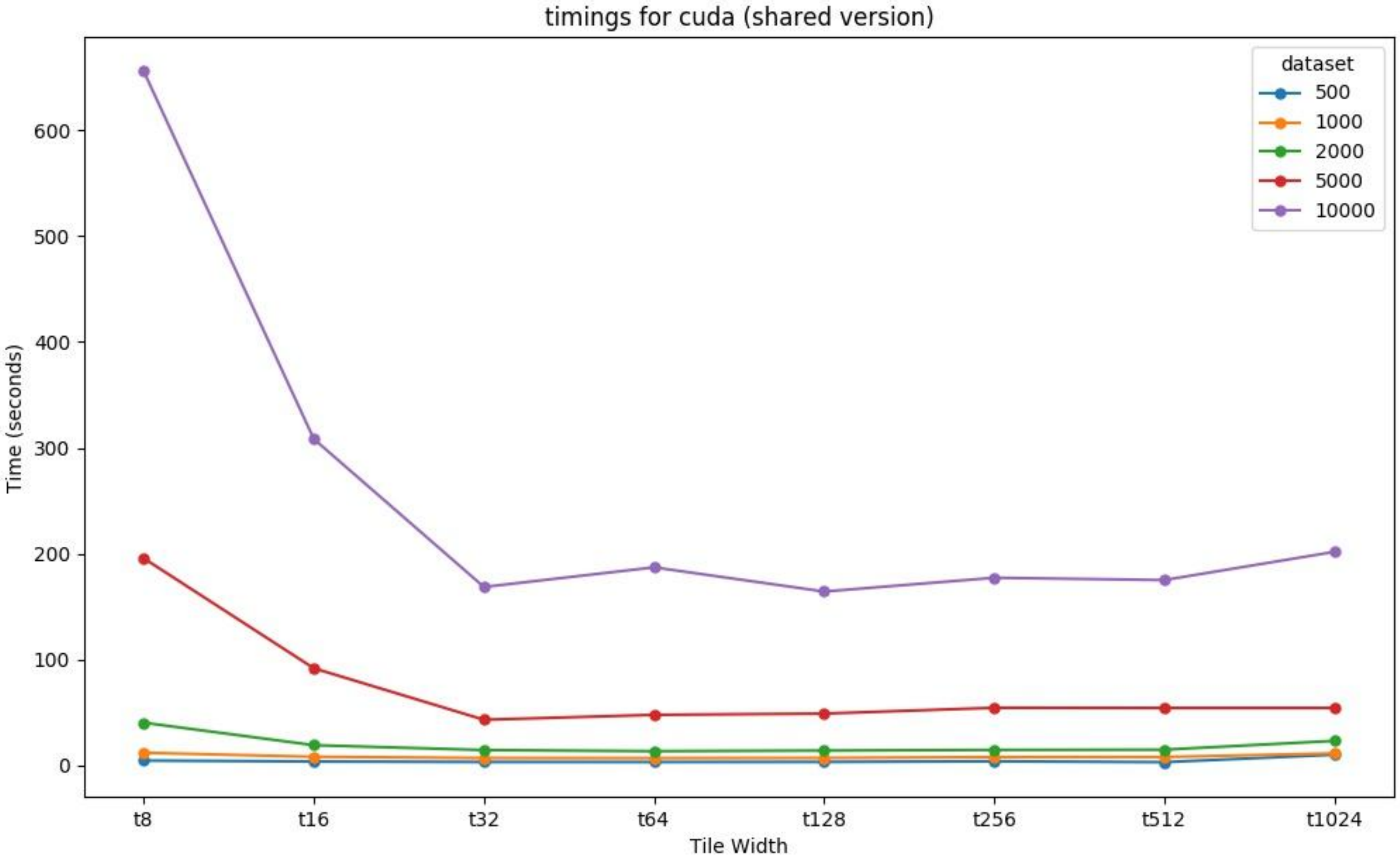
CUDA: naive results (timings)



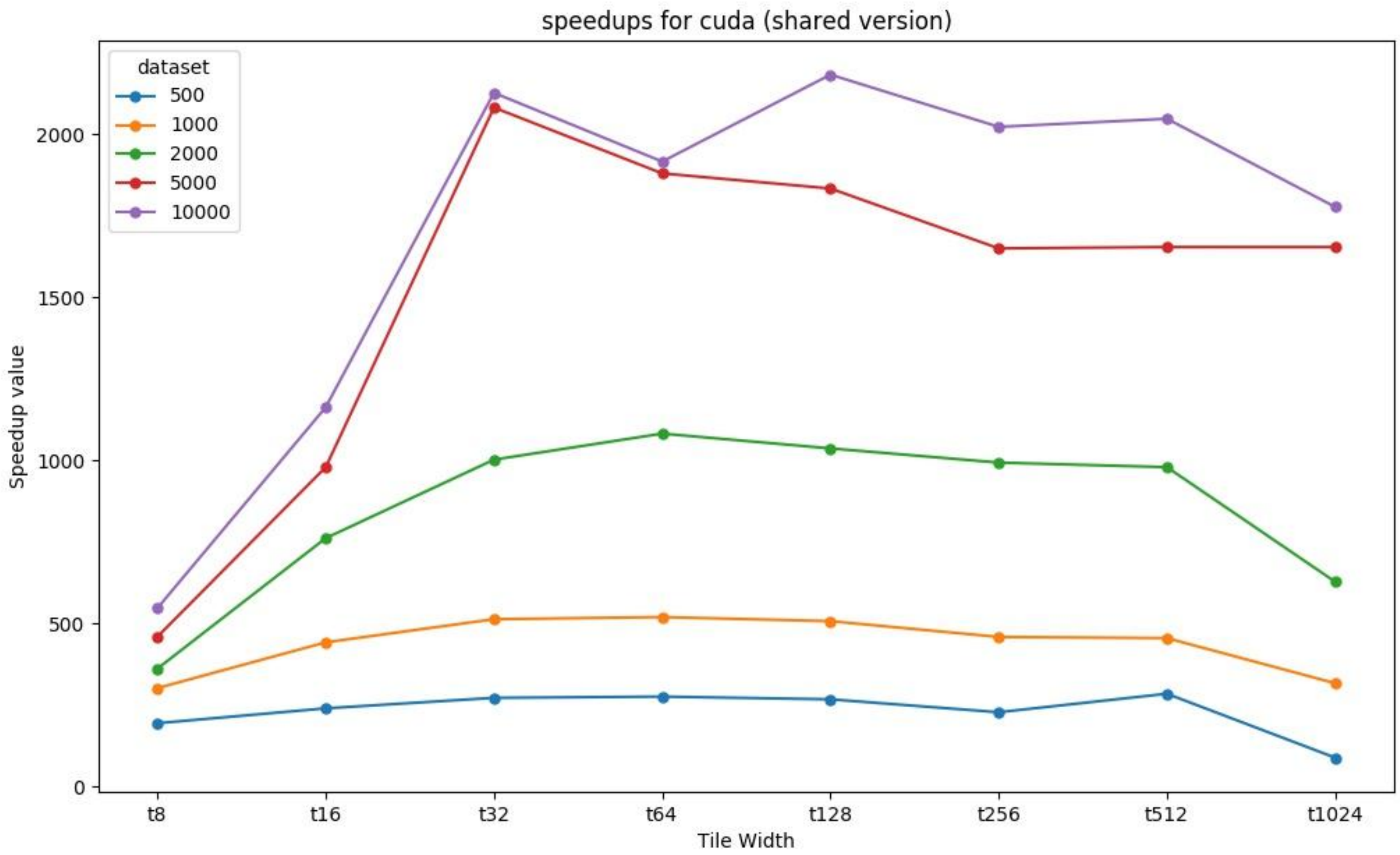
CUDA: naive results (speedups)



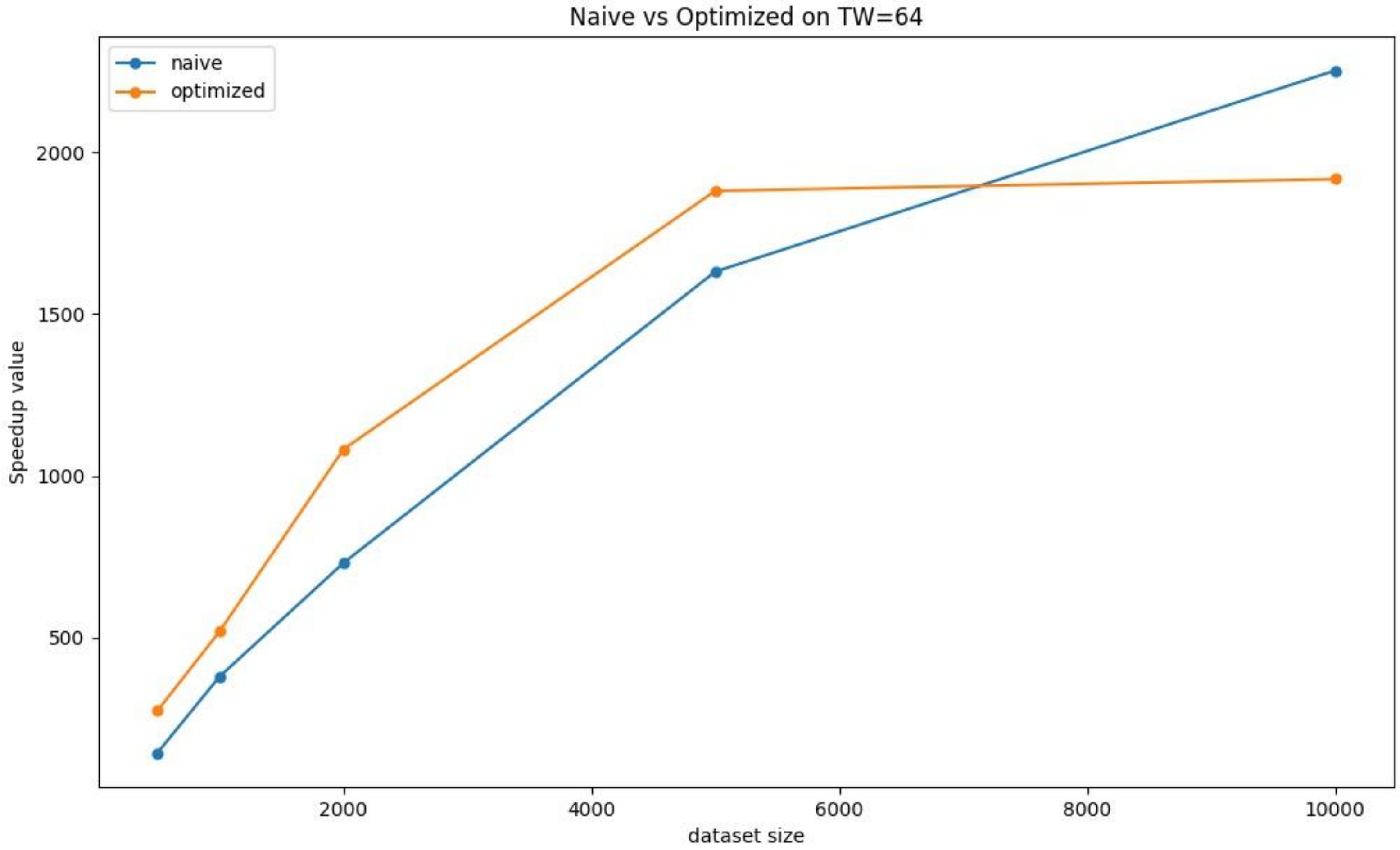
CUDA: shared memory results (timings)



CUDA: shared memory results (speedups)



CUDA: shared vs naive (speedups)



CUDA: comparison results (speedups)

Extra results: Comparison between different GPUS and architectures

- **Sequential Timings** are taken by:

  CPU Intel(R) Core(TM) i7-9700K CPU @ 3.6GHz

- **Parallel Timings** on:

  GPU NVIDIA GeForce **GTX 1050 Ti** m

  4096 MB dedicated and 768 CUDA Cores — **Pascal**

  GPU NVIDIA GeForce **GTX 1660 Ti**

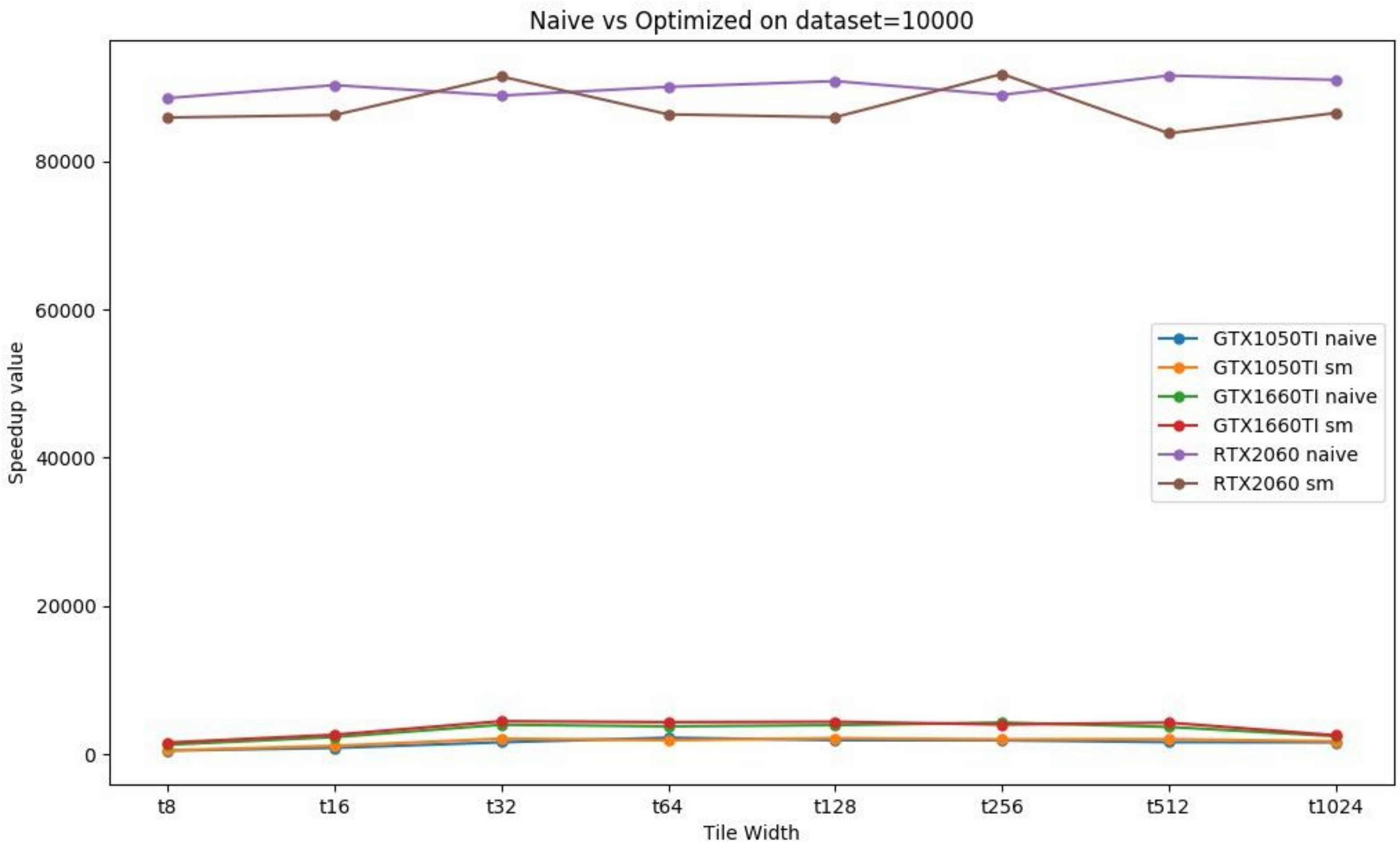
  6144 MB dedicated and 1536 CUDA Cores — **Turing**

  GPU NVIDIA GeForce **RTX 2060**

  6144 MB dedicated and 1920 CUDA Cores — **Turing**

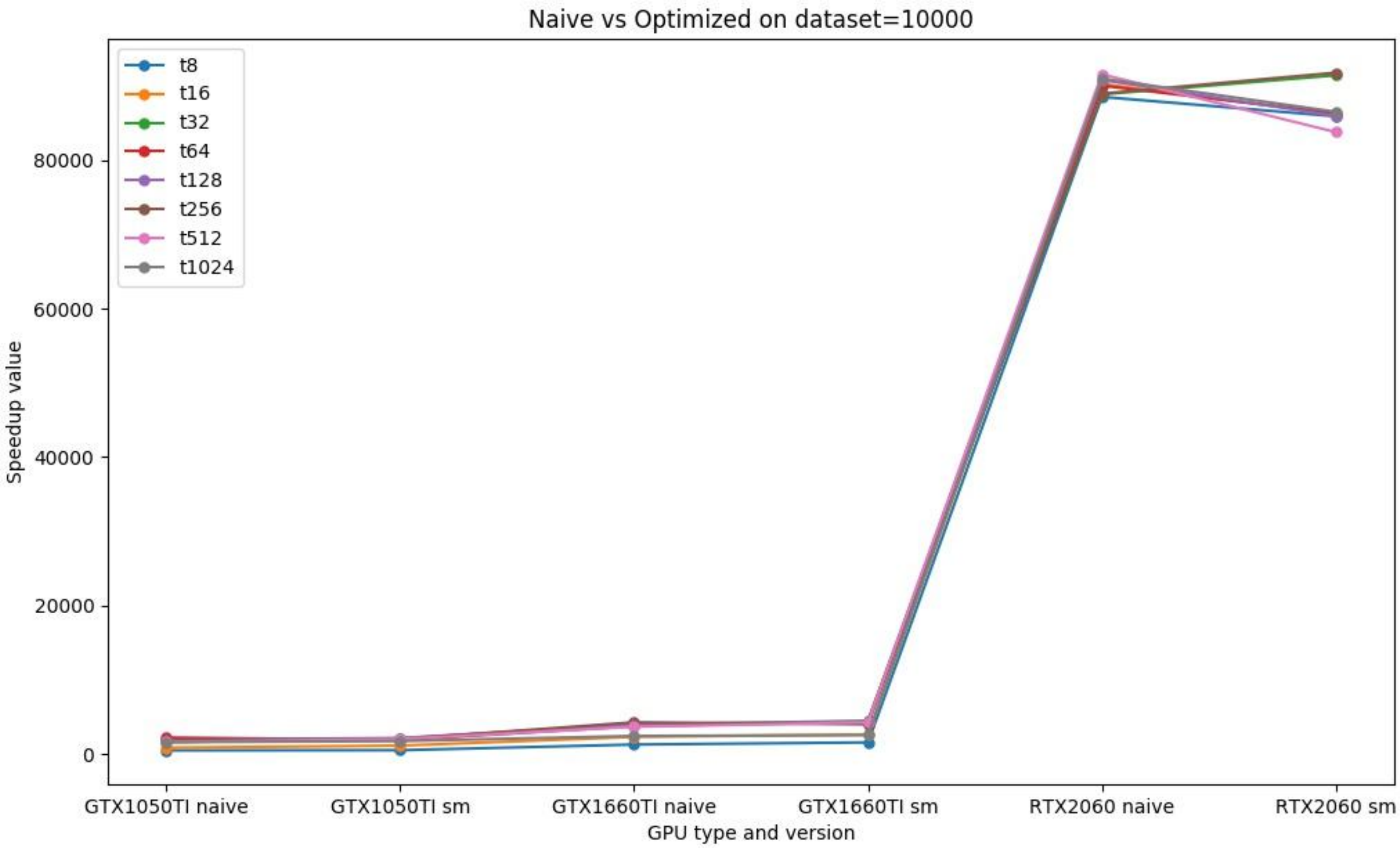
- **Experiments** with:
  - Dataset = {10000}
  - Tile Width = {8, 16, 32, 64, 128, 256, 512, 1024}

CUDA: comparison results (speedups)



RTX 2060 max speedup 91764 at TW=256

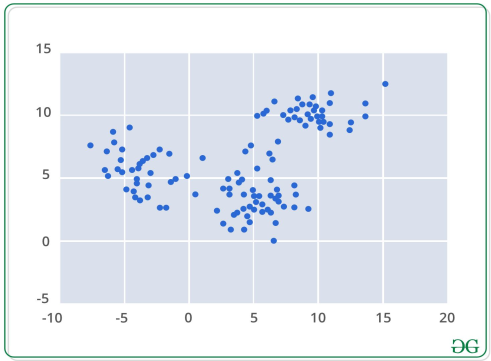
CUDA: comparison results (speedups)

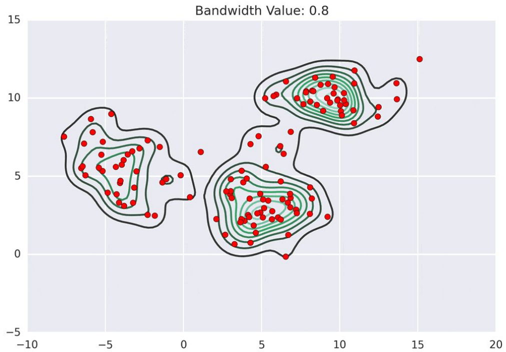


RTX 2060 max speedup 91764 at TW=256

Conclusions:

Three implementations of MM: one sequential and two parallel.





https://github.com/fabian57fabian/mean-shift-in-parallel

Thanks for your attention

Angelo D’Amante
Fabian Greavu