Mean shift clustering algorithm

comparison between sequential, openMP and CUDA implementations

Parallel Computing (9 CFU)

Francesca Del Lungo and Matteo Petrone

Prof. Marco Bertini

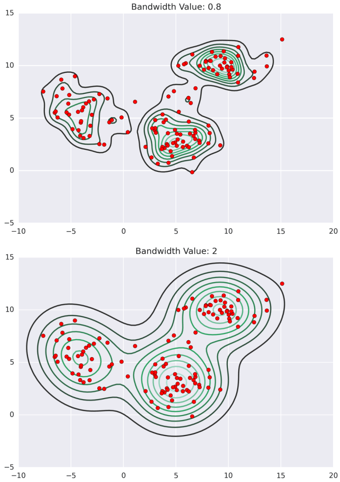
January 2021

Contents overview

Introduction

Mean Shift

- Non-parametric **clustering** procedure
- Computational Complexity: $\mathcal{O}(n^2)$.
- **Embarrassingly parallel** structure



Mean Shift

- It builds upon the concept of **kernel density estimation** (KDE).
- At each step of the procedure a kernel function is applied to each point that causes the points to shift in the direction of the nearest peak of the KDE surface:

$$K(x) = e^{-\frac{x^2}{2\sigma^2}} \tag{1}$$

where x: distance, σ : *bandwidth*.

Implementation

Sequential Version

Algorithm 1 Mean shift algorithm

procedure $\textsc{MeanShift}(\textit{original\_points})$

  $\textit{shifted\_points} \leftarrow \textit{original\_points}$

  while $\textit{iteration} < \textit{N\_ITERATIONS}$ **do**

    for $\textit{point}$ **in** $\textit{shifted\_points}$ **do**

      $\textit{point} \leftarrow \textsc{ShiftPoint}(\textit{point}, \textit{original\_points})$

Implemented in C++

Sequential Version

Algorithm 2 Mean shift subroutine

procedure $\textsc{ShiftPoint}(\mathit{point}, \mathit{original\_points})$

$\quad \mathit{num} \leftarrow 0$

$\quad \mathit{den} \leftarrow 0$

$\quad$ **for** op **in** $\mathit{original\_points}$ **do**

$\qquad \mathit{dist} \leftarrow \textsc{ComputeDistance}(\mathit{point}, \mathit{op})$

$\qquad w \leftarrow \textsc{Kernel}(\mathit{dist}, \mathit{BANDWIDTH})$

$\qquad \mathit{num} \leftarrow \mathit{num} + \mathit{op} * w$

$\qquad \mathit{den} \leftarrow \mathit{den} + w$

$\quad$ **return** $\mathit{num}/\mathit{den}$

Parallel Version: OpenMP

- OpenMP (Open Multiprocessing) is an API for shared-memory parallel programming.
- It follows the the **fork-join** programming model.
- It does not require program restructuring, but only the addition of **compiler directives**

Parallel Version: OpenMP

Algorithm 3 Mean shift openMP algorithm

procedure OMPMEANSHIFT(*original_points*)
  shifted_points ← *original_points*
  while *iteration* < *N_ITERATIONS* **do**
    #pragma omp parallel for schedule(static)
    for *point* **in** *shifted_points* **do**
      point ← SHIFTPOINT(*point*, *original_points*)

Parallel Version: CUDA naive

- Each Point to be shifted assigned to a thread.
- **Coalesced access** to global memory: points are stored as *Structure of Arrays* $[x_1, ..., x_n, y_1, ...y_n, z_1, ..., z_n]$.
- Access to the array is done: *BlockDim.x * BlockIdx.x* + *threadIdx.x*.

Parallel Versione: CUDA naive

Algorithm 4 CUDA naive kernel

procedure $\textsc{NaiveKernel}(shifted\_pts, orig\_pts)$

$idx \leftarrow threadIdx.x + blockIdx.x * blockDim.x$

if $idx < |orig\_pts|$ **then**

$sp \leftarrow shifted\_pts[idx]$

$tot\_weight \leftarrow 0$

$new\_spt \leftarrow 0$

for op **in** $orig\_pts$ **do**

$dist \leftarrow \textsc{ComputeDistance}(op, sp)$

$weight \leftarrow \textsc{Kernel}(dist, BW)$

$new\_spt \leftarrow new\_spt + op * weight$

$tot\_weight \leftarrow tot\_weight + weight$

$shifted\_pts[idx] \leftarrow new\_spt / tot\_weight$

Parallel Version: CUDA Tiling

- Limit global memory access making use of shared memory

Algorithm 5 CUDA kernel tiling - data loading

$tile \leftarrow shared\_mem\_array[TILE\_WIDTH]$

$n\_tiles \leftarrow (|orig\_pts| - 1)/(TILE\_WIDTH + 1)$

for $tile\_i$ **in** n_tiles **do**

   $tile\_idx \leftarrow tile\_i * TILE\_WIDTH + threadIdx.x$

   if $tile\_idx < |orig\_pts|$ **then**

      $tile[tx] \leftarrow orig\_pts[tile\_idx]$

   else

      $tile[tx] \leftarrow null\_pt$

   $\_\_syncthreads()$

Experiments

Dataset

- We generated sets of 3d points using *sklearn.dataset.make_blobs()*.
- It generates **gaussian distribution** with 3 centers and 1.5 standard deviation.
- Datasets dimensions: 100, 1.000, 10.000, 100.000, 500.000, 1.000.000.

Experiments setting

All the tests have been performed on a machine equipped with:

- CPU: Intel Core i7-860 @ 2.80GHz, with 4 cores/ 8 threads
- GPU: NVidia GeForce GTX 980, 4 GB (with CUDA 10.1)
- Average over 15 runs

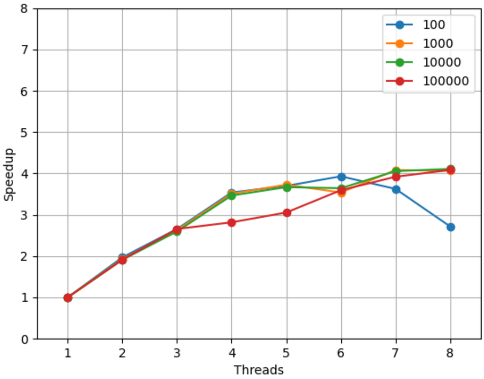
Speedup

To compare the performance of a sequential respect to a parallel implementation: **speedup** metric.

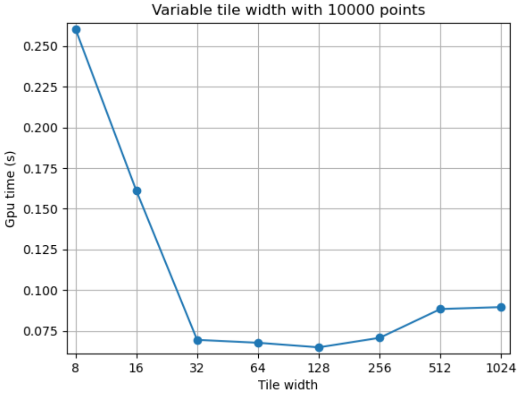
$$S = \frac{t_S}{t_P}$$

where t_s: sequential time execution, t_p: parallel time execution.

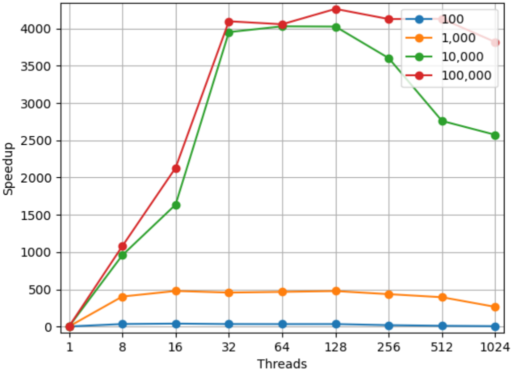
Speedup: Sequential over OpenMP



GPU Time varying tile width



Speedup: Sequential over CUDA



Conclusions

Conclusions

- Mean shift $\mathcal{O}(n^2)$, but **embarrassingly parallel** structure

OpenMP

- Implementation difficulty: very low
- Speedup: sub-linear

CUDA

- Implementation difficulty: higher
- Speedup: up to 4400x time faster.

Thanks for the attention