PC-2022/23 Parallel Mean Shift Clustering

Panconi Christian
E-mail address
christian.panconi@stud.unifi.it
panconi.christian@gmail.com

Abstract

Mean Shift is a general nonparametric technique for finding the modes of an unknown probability density function using a discrete set of samples.
Applied to cluster analysis, Mean Shift results in a density estimation-based iterative method which is able to identify arbitrarily shaped clusters without using a priori knowledge on their number.
The method's complexity is $O(n^2)$ where n is the number of data points, making it not scalable and time consuming for large datasets in its basic form, but an efficient implementation on massively parallel architectures can be obtained since the underlying problem solved by the Mean Shift technique is embarassingly parallel.
This report examines a possible parallelization technique for the Mean Shift algorithm on GPUs using CUDA.

1. Mean Shift Procedure and Clustering

In order to outline the mathematics of the Mean Shift procedure, this section is a brief literature review using [2] as reference.

1.1. KDE and MS Procedure

Mean Shift is a procedure which aims to find the local maxima of an unknown probability density function using kernel density estimation (KDE).
Given n data points x_i , $i = 1, \dots, n$ in $\mathbb{R}^d$, the *multivariate kernel density estimator*, using a kernel function $K(x)$ and a symmetric positive definite $d \times d$ bandwidth matrix H, computed in a point x is:

$$\hat{f}(x) = \frac{1}{n} \sum_{i=1}^{n} K_H(x - x_i) \tag{1}$$

where:

$$K_H(x) = |H|^{(-1/2)} K\Big(H^{(-1/2)} x\Big)$$

To simplify the definition of the bandwidth matrix, H is chosen proportional to the identity matrix $H = h^2 I$, thus only the definition of the h parameter is required. By using this kind of bandwidth matrix the kernel density estimator becomes:

$$\hat{f}(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K\Big(\frac{x - x_i}{h}\Big) \tag{2}$$

The *d-variate* kernel $K(x)$ is chosen from a special class of radially symmetric kernels satisfying:

$$K\big(x\big) = c_{k,d} \cdot k(\|x\|)$$

where $c_{k,d}$ is a normalization constant, $k(x)$ is called the *profile* of the kernel and it suffices to define the kernel. The profile:

$$k_N(x) = \exp\Big(-\frac{1}{2}x\Big)$$

yields the *multivariate normal kernel*:

$$K_N(x) = (2\pi)^{-d/2} \exp\Big(-\frac{1}{2}\|x\|^2\Big)$$

With the profile notation the density estimator (2) can be expressed as:

$$\hat{f}(x) = \frac{c_{k,d}}{nh^d} \sum_{i=1}^{n} k\Big(\Big\|\frac{x - x_i}{h}\Big\|^2\Big) \tag{3}$$

In a feature space with underlying density function $f(x)$ the modes are located among its stationary points where $\nabla f(x) = 0$, the mean shift procedure aims to find these stationary points without estimating the density, by using a *density gradient estimator*.
The gradient estimator is obtained as the gradient of the density estimator $\hat{f}$:

$$\hat{\nabla} f_{h,K}(x) \equiv \nabla \hat{f}_{h,K}(x) = \frac{2c_{k,d}}{nh^{d+2}} \sum_{i=1}^{n} (x - x_i) k'\Big(\Big\|\frac{x - x_i}{h}\Big\|^2\Big) \tag{4}$$

Defining the function:

$$g(x) = -k'(x)$$

and introducing it into (4) results in the following form for the gradient estimator:

$$\begin{aligned}&\hat{\nabla} f_{h,K}(x)\\&= \frac{2c_{k,d}}{nh^{d+2}} \sum_{i=1}^{n} (x_i - x) g\Big(\Big\| \frac{x - x_i}{h} \Big\|^2 \Big)\\&= \frac{2c_{k,d}}{nh^{d+2}} \left[\sum_{i=1}^{n} g\Big(\Big\| \frac{x - x_i}{h} \Big\|^2 \Big) \right] \left[\frac{\sum_{i=1}^{n} g\Big(\Big\| \frac{x - x_i}{h} \Big\|^2 \Big) x_i}{\sum_{i=1}^{n} g\Big(\Big\| \frac{x - x_i}{h} \Big\|^2 \Big)} - x \right]\end{aligned}$$

where the quantity $\sum_{i=1}^{n} g\Big(\Big\| \frac{x - x_i}{h} \Big\|^2 \Big)$ is assumed positive and the first term is proportional to the density estimator in x computed using the kernel $G(x) = c_{g,d} g(\|x\|^2)$:

$$\hat{f}_{h,G}(x) = \frac{c_g, d}{nh^d} \sum_{i=1}^{n} g\Big(\Big\| \frac{x - x_i}{h} \Big\| \Big)$$

The other term is the *mean shift vector*:

$$m_{h,G}(x) = \frac{\sum_{i=1}^{n} g\Big(\Big\| \frac{x - x_i}{h} \Big\|^2 \Big) x_i}{\sum_{i=1}^{n} g\Big(\Big\| \frac{x - x_i}{h} \Big\|^2 \Big)} - x \qquad (5)$$

which is the difference between the weighted mean, using the kernel G for weights, and x.
Since the mean shift vector is proportional to the density gradient estimator $\hat{\nabla} f_{h,K}$, it always points in the direction of maximum increase in the estimated density and can be used to define a gradient ascent method to find the stationary points of $\hat{f}$, i. e., the density modes.
The *mean shift procedure* is obtained by successive:

- computation of the mean shift vector $m_{h,G}(x)$.
- translation of the kernel $G(x)$ by the mean shift vector:

$$x \longleftarrow x + m_{h,G}(x)$$

Such iterative procedure, under suitable hypotheses on the kernel K, is guaranteed to generate a sequence $\{y_j\}_{j=1,2,\ldots}$ which converge to a point $y^\star$ where $\hat{\nabla} f_{h,K}(y^\star) = 0$.
Depending on the kernel, the procedure can either converge in a finite number of steps or be infinitely convergent, as in the case of a normal kernel. In the practice, the stopping criterion is a lower bound on the magnitude of the mean shift vector.

1.2. MS Clustering

Mean Shift can be employed in clusters detection by running the procedure to compute the convergence points $y_i^\star$ for every point x_i, then the clusters can be formed by grouping together all the points which converges to the same mode.

If Z is the set of the stationary points of $\hat{f}$, $Z = \{z \in \mathbb{R}^d | \hat{\nabla} f_{h,K}(z) = 0\}$, each convergence point $y_i^\star \in Z$, and the clusters:

$$C_l = \Big\{ x_i | y_i^\star = z_l \Big\} \quad l = 1, \ldots, K$$

where K is the number of different modes obtained by the mean shift procedure.
While this is mathematically correct, in a practical application, the finite precision and the stopping criterion will cause the algorithm to generate approximate stationary points $\hat{y}_i^\star$ where $|m_{h,G}(\hat{y}_i^\star)| < \epsilon$. These approximate stationary points will be all distinct, but using adequate value for ϵ, the convergence points obtained from the x_i belonging to same cluster will be near to each other.
A possible solution to produce the clusters from the set of convergence points [1] is to build an undirected graph where the set of nodes is $V = \{x_i, i = 1, \ldots, n\}$ and there is an edge between two nodes if the distance between their respective convergence points is lower than a given threshold:

$$E = \{(x_i, x_j) \ | \ d(\hat{y}_i^\star, \hat{y}_j^\star) < \delta, i \neq j\}$$

The connected components of $\mathcal{G} = \langle V, E \rangle$ provide the clusters, eventually after applying some postprocessing operations to remove small clusters and outliers (saddle points).
In [1] the authors also provide an algorithm to find such connected components without explicitly building the graph under reasonable conditions on the clusters structure.

2. Algorithms

The complete Mean Shift clustering algorithm is composed by two phases:

1. computation of the convergence points $\hat{y}_i^\star$ for every dataset point x_i using the Mean Shift procedure.
2. clusters formation by finding the connected components of the graph built using the convergence points.

Algorithm 1 shows the Mean Shift procedure where the mean shift vector $m_{h,G}(y_i)$ at lines 3 and 4 is computed using eq. (5).
Employing a normal kernel profile $k_N(x) = exp(-\frac{1}{2}x)$ the function g is $g(x) = \frac{1}{2} exp(-\frac{1}{2}x)$ and the expression for the mean shift vector becomes:

$$m_{h,G}(x) = \frac{\sum_{j=0}^{n} exp(-\frac{1}{2} \Big\| \frac{x - x_j}{h} \Big\|^2) x_j}{\sum_{j=0}^{n} exp(-\frac{1}{2} \Big\| \frac{x - x_j}{h} \Big\|^2)} - x \qquad (6)$$

Since the computation of the mean shift vector at a given point x_i requires iterating through all the other points and the inner while loop is executed at least once for every x_i,

Algorithm 1: Mean Shift Procedure

Data: x_i , n , h , ϵ
Result: the convergence points $\hat{y}_i^\star$

```
1 for i = 1, ... , n do
2     y_i ⟵ x_i
3     while |m_{h,G}(y_i)| > ε do
4         y_i ⟵ y_i + m_{h,G}(y_i)
5     end
6     ŷ_i* ⟵ y_i
7 end
```

Algorithm 2: Clustering Connected Components

Data: $\hat{y}_i^\star$, n , δ
Result: the clusters $C_1, \ \dots \ , C_K$

```
1  K ⟵ 1
2  C_1 ⟵ {x_1}
3  for j = 1, ... , n do
4      for k = 1, ... , K do
5          if d(ŷ_j*, C_k) < δ then
6              C_k ⟵ C_k ∪ {x_j}
7          else
8              K ⟵ K + 1
9              C_K ⟵ {x_j}
10         end
11     end
12 end
```

the time complexity of the algorithm is $O(n^2)$.
The number of iterations for a single point to converge depends on the initial distance of the point from the nearest mode.
Connected components can be found by building the graph and using a DFS-based algorithm, but this can be quite inefficient on large datasets due to the $O(n^2)$ complexity. The fast connected component algorithm from [1] is a more efficient way of finding the clusters. Such method requires a condition on the clusters referred by the authors as "tight clusters assumption": there exists $\delta > 0$ that is larger than the diameter of every component, but smaller than the distance between any two points belonging to different components.
If this condition is not met the clustering results will be poor, but since the points on which the algorithm operates are the convergence points obtained by the mean shift procedure it becomes reasonably easy to satisfy because the approximate stationary points $\hat{y}_i^\star$ are close to the different modes (if using an appropriate value ϵ in the stopping criterion).
Algorithm 2 shows the connected components detection procedure, where the distance of a point $\hat{y}_i^\star$ from a cluster C_k (line 5) is:

$$d(\hat{y}_i^\star, C_k) = \min_{j:x_j \in C_k} d(\hat{y}_i^\star, \hat{y}_j^\star)$$

To verify the condition $d(\hat{y}_i^\star, C_k) < \delta$, the minimum computation can be interrupted as soon as a point which verifies the condition is found in the cluster C_k.
With this algorithm, under the tight clusters assumption, the clusters can be found on average in $O(nK)$.

The parameters required by the complete algorithm are:

- h: the bandwidth parameter of the kernel used by the Mean Shift procedure.
- ϵ: the lower bound for the magnitude of the mean shift vector. It determines the convergence speed for every point of the mean shift procedure, low values for ϵ will produce $\hat{y}_i^\star$ closer to the modes, at the cost of increasing the average number of iterations per point.
- δ: the threshold on the converge points distance required by the connected components algorithm. For a good clustering result, this parameter should be chosen to satisfy the "tight clusters assumption". Values which are too low for δ can cause clusters to be split while values which are too high can incorrectly merge different clusters.

3. Parallelization and implementation details

A characteristic of the Mean Shift Procedure is that the computations needed for a given point point x_i are independent from the computations required for all the other points. This property makes the problem of finding the Mean Shift convergence points (phase 1) an *embarrassingly parallel problem*, where the work can be easily split among an high number of threads, allowing for efficient implementations on massively parallel architectures.
The reported parallelization technique uses a GPU to run the Mean Shift procedure, with implementation in C++ (GCC 10.3.0) and CUDA (version 11.7).

Each thread computes the converge point starting from a different dataset point, but due to the CUDA execution model these threads are scheduled in blocks. The computing resources assigned to a block are released only when all the threads of the block terminates. Each thread converges in its own number of iterations, and this can lead to situations where resources are occupied by blocks with a low number of active (not yet converged) threads.
To reduce this effect the kernels can be launched using small values for the block size, in this way a block will stall

Kernel	blockDim.x	tileSize	Regs/block	blocks/SM	Available shared mem/block	d_{max}	Occupancy
simple	32	/	1024	32	3072 bytes	24	0.5
	64	/	2048	32	3072 bytes	12	1.0
tiled	32	32	1728	32	3072 bytes	7	0.5
	64	64	3072	21	4681 bytes	6	0.66

Table 1: resources usage for some kernel launch configurations on a device with CC 6.1 . The kernel type, blockDim.x and the tile size (if tiled) determine the occupancy and the maximum data dimensionality which does not limit the occupancy d_{max}.

less resources, but the occupancy will be limited. Another solution is to organize the input such that threads within the same block have a similar processing time. This requires an heuristics which depends on the nature of the dataset and the task.
Regarding the block size, the smallest useful value is the warp size (32), but using too small blocks limits the kernel occupancy. For example, referring to compute capability 6.1, the maximum number of resident blocks on a streaming multiprocessor (SM) is 32, with 32 threads per block, the number of threads per multiprocessor is $32 \cdot 32 = 1024$ and since the hardware limit for the number of threads per SM is 2048, the kernel occupancy will be $1024/2048 = 0.5$, i.e. only half of the GPU resources are utilized. An optimal value for the block size would be 64 which maximizes the theoretical occupancy.

Two different implementations for the Mean Shift kernel have been examined, the first one is referred as *"simple"* implementation, the second one uses the tiling technique and hence it is referred as *"tiled"* implementation. Both use one-dimensional thread blocks.

In the *simple* implementation each thread starts from a different dataset point and repeatedly computes the mean shift vector and the shifted point, until convergence. The shifted points y_i computed at each iteration are kept in a shared memory buffer, thus the kernel requires $blockDim.x \cdot d \cdot sizeof(float)$ bytes of shared memory per block. This shared memory requirement sets a limit on the maximum data dimensionality (d) that the kernel can handle without lowering the occupancy and it depends the available shared memory of the device.
With CC 6.1 the maximum shared memory per SM is 96 Kb and when scheduling 32 blocks per SM every block has 3 Kb of available shared memory, assuming 4 bytes for a float and 64 threads per block ($blockDim.x = 64$), the maximum data dimensionality that the *simple* kernel can handle to avoid occupancy limitations from the shared memory is $d_{max} = 3 \text{ Kb}/(64 \cdot 4 \text{ bytes}) = 12$.

In the *tiled* version each iteration of the Mean Shift procedure is split into sub-iterations where each block collaboratively loads a so called "tile" of data into shared memory and then each thread computes the partial accumulation for the quantities $\sum_{i=1}^{n} g\Big(\big\| \frac{x-x_i}{h} \big\|^2 \Big) x_i$ and $\sum_{i=1}^{n} g\Big(\big\| \frac{x-x_i}{h} \big\|^2 \Big)$. When the tiling loop ends, each thread can compute the shifted point y_i and the mean shift vector $m_{h,G}(y_i)$ using these quantities.
Some threads will converge faster than others, but they still need to continue to load the tiles in the next iterations. This implies a mechanism to stop the whole block when all the threads have converged, in particular this has been implemented using a shared integer variable on which every thread performs an atomic increment once they have converged and the whole block exit when this counter reaches $blockDim.x$.
This kernel requires additional shared memory to hold the data tiles and the shared counter, for a total of $d \cdot (2blockDim.x + tileSize) \cdot sizeof(float) + sizeof(int)$ bytes. The tile size has been fixed to $blockDim.x$.

Together with the block size, the other factor that limits the achievable theoretical occupancy of the kernels is the number of registers used by each thread. In the *simple* kernel each thread uses 32 registers, when $blockDim.x = 64$ each block requires $32 * 64 = 2048$ registers and since the maximum number of registers per SM is 65536, each SM can run $65536/2048 = 32$ blocks in parallel (which matches the hardware limit in CC 6.1), allowing a 100% theoretical occupancy.
The *tiled* kernel is more expensive in terms of registers because each thread uses 48 registers and each block requires $48 * 64 = 3072$ registers, this means a maximum of $\lfloor 65536/3072 \rfloor = 21$ blocks in parallel on each SM, limiting the theoretical occupancy to $(64 * 21)/2048 = 0.66$, only 66%. In this case, the lower number of blocks that can be executed in parallel on each SM has the effect of raising the available shared memory for each block to $96 \text{ Kb}/21 = 4681$ bytes which can be used to handle data with higher dimensionality without further limiting the SMs occupancy.

Another trade-off between register usage and speed lies in how the square for the norms in the mean shift

Input order	*Kernel launch parameters*				*Average speedup wrt image size*				
	Type	blockDim.x	tileSize	Occupancy	128 x 128	142 x 142	176 x 176	256 x 256	512 x 512
raster	simple	32	/	0.5	41.5	39.0	46.6	54.2	50.9
		64	/	1.0	**51.8**	**46.1**	**50.5**	**55.7**	**58.0**
	tiled	32	32	0.5	45.2	39.5	48.6	54.2	53
		64	64	0.66	44.9	38.8	48.4	52	52.8
casual	simple	32	/	0.5	35.4	35.3	35.1	37.4	39.1
		64	/	1.0	**45.4**	**41.3**	32.9	**40.9**	**39.1**
	tiled	32	32	0.5	37.6	36.0	**36.0**	37.8	35.2
		64	64	0.66	35.3	31.1	31.0	37.5	35.1
blocks	simple	32	/	0.5	45.2	42.9	52.3	60.8	57.1
		64	/	1.0	**54.9**	**54.4**	54.8	**66.6**	**65.3**
	tiled	32	32	0.5	51.9	48.3	**55.4**	61.3	59.3
		64	64	0.66	51.9	48.1	55.0	61.4	59.3

Table 2: Average speedup obtained on the image segmentation task for $h = 0.07$ and pixels supplied to the algorithm in three different orders: raster, casual and "blocks sorted", using blocks of 8×8 pixels.

vector expression (eq. 5) is implemented: by using a simple multiplication or the builtin CUDA `powf` function. Simple multiplication uses 8 more registers than the `powf` function, but it' s faster. The gain in occupancy due to the lower number of registers per thread brought by the `powf` function has not showed to be worth in this case and thus the tested kernels compute the squares with a simple multiplication. The `powf` function has been used to compute the square for the mean shift vector norm in the convergence check, because it is executed way less times than the square for the mean shift vector computations. This has allowed to obtain an implementation for the *simple* kernel which uses 32 registers per threads and can achieve a theoretical 100% occupancy at the cost of some tolerable speed loss when checking the magnitude of the mean shift vector.

4. Evaluations

In order to evaluate the speedup gained by the parallel implementation, the algorithm has been applied to an image segmentation task. Each pixel of the original image is a data point in the five dimensions (x, y, R, G, B) re-scaled to $[0, 1]$ in each dimension.
The implementations have been evaluated on images of five different size: 128×128, 142×142, 176×176, 256×256 and 512×512 pixels.
The bandwidth parameter h has been varied using three different values: 0.07, 0.02, 0.1, while the other parameters have been kept fixed: $\epsilon = 0.001$, $\delta = 0.02$ and the maximum number of iterations per point limited to 100. The following hardware has been used for the evaluations:

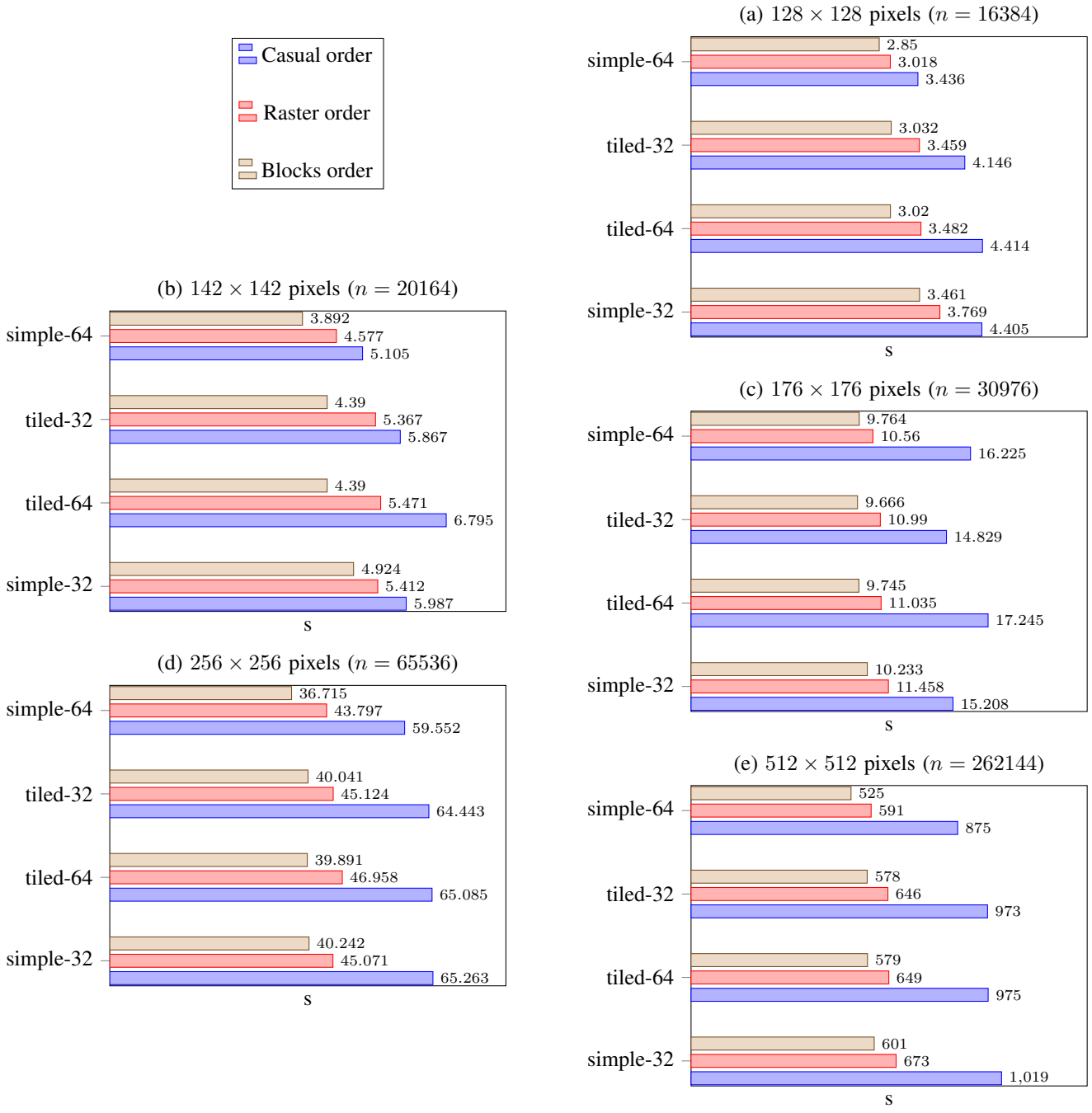
- AMD® Ryzen 7 1700 (8 core/16 threads, Q1 2017)
- NVidia® GeForce GTX 1060 (compute capability 6.1, Pascal arch., Q3 2016)

Table 2 reports the average speedup obtained by the CUDA implementation over a sequential C++ implementation varying image size and the order in which the image pixels are supplied to the CUDA kernel, using:

- casual order: the pixels are shuffled randomly.
- raster order: the order obtained by scanning the image by rows starting from the top one.
- "blocks" order: obtained by splitting the image in blocks of 8×8 pixels and scanning those blocks by rows starting from the top-left one.

The higher speedups have been obtained by supplying the points in the "blocks" order, supporting the fact that threads within the same block converging in a similar number of iterations are crucial to this CUDA Mean Shift implementation. A good order for the input points depends on the task to which the Mean Shift procedure is applied, in the case of image segmentation, pixels localized in a squared block are near in position and, with suitable small blocks, color, so there is a high chance that the threads of the block processing them will converge to the same $y*$ (in the limit), taking a similar processing time.
Overall the simple kernel has showed to be the fastest implementation when running with 64 threads per block (100% theoretical occupancy) but it is worth noting that in the case of casual order, for the images bigger than 142×142 pixels, the performances of the simple kernel (using $blockDim.x = 64$) degrades. On the device used to test the kernels, 142×142 is the maximum squared images size for which the grid fits the GPU at once, without blocks waiting to be scheduled, since a GeForce GTX

Figure 1: Average execution time of the mean shift procedure on the image segmentation task using $h = 0.07$. Each chart compares the different implementations and launch configurations on a different image size. The suffix "-32" or "-64" indicates the block size for the kernel launch.



1060 is equipped with 10 streaming multiprocessors and a multiprocessor can run a maximum of 2048 threads in parallel, for a total of 20480 threads. Above this number of pixels there are waiting blocks even in the 100% occupancy case and the improvement of the "raster" and "blocks" orders over the "casual" one grows with the image size as the more blocks need to be scheduled, the more important is the uniformity of the threads execution time.

The tiled implementation seems to perform almost

Input order	*Kernel launch parameters*		*Average speedup wrt image size*				
	blockDim.x	Occupancy	128 x 128	142 x 142	176 x 176	256 x 256	512 x 512
raster	32	0.5	54.5	55.4	67.3	76	83.9
	64	1.0	**55.8**	**63.9**	**72.8**	**82.4**	**95.6**
casual	32	0.5	46.7	50.3	59.5	62.0	65.3
	64	1.0	**53.4**	**58.5**	**64.5**	**68.0**	**75.2**
blocks	32	0.5	55.8	57.2	71.1	84.0	94.1
	64	1.0	**56.5**	**65.3**	**76.5**	**91.9**	**108.3**

Table 3: Average speedups for the "simple" kernel on the image segmentation task using $h = 0.02$.

Input order	*Kernel launch parameters*		*Average speedup wrt image size*				
	blockDim.x	Occupancy	128 x 128	142 x 142	176 x 176	256 x 256	512 x 512
raster	32	0.5	39.8	41.6	46.9	52.2	57.6
	64	1.0	**46.5**	**47.3**	**49.5**	**53.5**	**65.4**
casual	32	0.5	30.5	37.0	32.6	35.7	35.4
	64	1.0	**39.9**	**42.6**	**34.2**	**39.7**	**41.5**
blocks	32	0.5	44.6	45.2	52.0	58.0	62.9
	64	1.0	**52.6**	**57.3**	**56.8**	**64.2**	**72.6**

Table 4: Average speedups for the "simple" kernel on the image segmentation task using $h = 0.1$.

similar or slightly better when launched with 32 threads per block instead of 64, meaning that the increase from 50% to 66% theoretical occupancy does not bring an improvement and blocks holding less resources can balance this relatively small gap in occupancy.

Tab 3 and Tab 4 reports the speedups obtained by the simple kernel with different values of h. In the case of $h = 0.02$ the speedups are noticeably higher, this is probably due to the fact that with a lower value of h there are less data points contributing to the mean shift vector, so the threads converge faster and, overall, the execution times are more uniform.

References

[1] M. Á. Carreira-Perpiñán. A review of mean-shift algorithms for clustering. *CoRR*, abs/1503.00687, 2015.

[2] D. Comaniciu and P. Meer. Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619, 2002.