An Algorithmic Skeleton for Massively Parallelized Mean Shift Computation with Applications to GPU Architectures

Darius Malysiak
Hochschule Ruhr West
Computer Science Institute
Bottrop, Germany
darius.malysiak@hs-ruhrwest.de

Uwe Handmann
Hochschule Ruhr West
Computer Science Institute
Bottrop, Germany
uwe.handmann@hs-ruhrwest.de

***Abstract*—In this paper we discuss parallelization approaches for generic mean shift clustering. We provide an algorithmic skeleton which allows an easy creation of platform specific implementations, be it small scale systems as multicore CPUs, large GPUs or even distributed cluster systems. Additionally we provide an exhaustive runtime complexity analysis and various remarks for further research. In order to illustrate the practicability of our theoretic framework we discuss a GPU implementation which exhibits significant speedups for small and large scale datasets.**

***Keywords*-statistics, gpgpu, high performance computing, mean shift clustering, clustering, opencl, cuda**

I. Introduction and previous work

The mean shift algorithm as explained in e.g. [1], [2] or in the original paper [3], is a non-parametric method which allows one to approximate and select modes in a sample from some arbitrary distribution of multidimensional datapoints. It is widely used in the area of image processing as e.g. a clustering method in the HOG algorithm [4] or as tracking method in [5]. Although it exhibits a simple design, it may become unfeasible in certain situations if the sample size grows to large. One example is its application in the HOG algorithm, even if the detection system itself exhibits a high efficiency (e.g. [6]), it may produce a number of results for which the mean shift clustering would become the systems bottleneck. Modern multicore CPUs and GPUs provide excellent system architectures for parallelized mean shift execution. Yet up to this point we know of no existing theoretical analysis of the mean shifts parallelization potential. There already exist GPU accelerated implementations e.g. a naive version in the OpenCV library or a very specialized version for tracking applications in [7] using CUDA.

In this paper we will state a theoretical framework for the mean shift algorithm itself and analyze its applicability for small scale as well as large scale parallelization. Our analysis of runtime complexities respects generic system aspects as e.g. memory accesses and parallelization factors. The algorithmic skeletons can be used for further theoretical studies or as a guideline for designing platform specific implementations, be it GPUs or large cluster systems.

Section II introduces the classical mean shift algorithm and derives a naive parallelizable form. In section III this naive form will be extended to the final algorithmic skeleton which also represents the basis for our GPU formulation in section IV. Throughout these two section we provide a rigorous analysis of the skeletons. The corresponding results will be discussed in section V, followed by a conclusion of our work.

II. The Mean Shift Algorithm

We will now briefly discuss the original algorithm (with notation lend from [4]) and derive a massively parallelizable form. Let $S \subseteq \{x \in \mathbb{R}^n\}$ be a sample of n k-dimensional datapoints drawn from a distribution $\mathcal{D}$. For any given point $p \in S$ the mean shift algorithm will iteratively compute an approximation of the closest mode y_p to it. This is done via the following iteration equation

$$y_p = H_h(y_p) \sum_{i=1}^{n} \overline{\omega}_i(y_p) {H_i}^{-1} y_i \tag{1}$$

with

$$\overline{\omega}_i(y_p) = \frac{|H_i|^{-1/2} exp(-D^2[y_p, y_i, H_i]/2)}{\sum_{j=1}^{n} |H_j|^{-1/2} exp(-D^2[y_p, y_j, H_j]/2)} \tag{2}$$

$\biguplus_{i=1}^{n}\{y_i\} = S$ and

$$D^2[y_p, y_j, H_j] \quad := \quad (y_p - y_i)^T {H_i}^{-1} (y_p - y_i) \tag{3}$$

These expressions require a more detailed explaination, H_i is the so called uncertainty matrix and takes the role of an estimated covariance matrix. Thus $D^2[y_p, y_j, H_j]$ becomes the Mahalanabois distance between y_p and y_j. Note that in its original form the algorithm uses an undefined kernel $K(y_p, y_i)$, yet as the RBF kernel is most common, we fix K to be of this form throughout the paper. In addition to the generic kernel, the original algorithm does not utilize statistical methods to compute the distance between data

978-1-4799-5338-7/14/$31.00 ©2014 IEEE

points, thus it does not use any covariance matrices. The Matrix H_h is defined indirectly through

$$H_h^{-1}(y) = \sum_{i=1}^{n} \overline{\omega}_i(y) H_i^{-1} \quad (4)$$

In practical applications one usually has a closed expression to calculate H_i, i.e. $H_i = H_i(y_i)$.

Remark. *Note that if H_i is a diagonal matrix, H_h is one as well and can be calculated with high numerical stability.*

By rewriting eq. 1 and eq. 2 we obtain

$$y_p = H_h(y_p) \frac{\sum_{i=1}^{n} |H_i|^{-1/2} exp(-D^2[y_p, y_i, H_i]/2) H_i{}^{-1} y_i}{\sum_{j=1}^{n} |H_j|^{-1/2} exp(-D^2[y_p, y_j, H_j]/2)} \quad (5)$$

Which indicates that $|H_i|^{-1/2} exp(-D^2[y_p, y_i, H_i]/2)$ occurs exactly the same amount of times in nominator and denominator. With this observation one can significantly reduce the required calculations by simply evaluating the expressions in lockstep. The same holds for eq. 4, which has the form

$$H_h^{-1}(y_p) = \frac{\sum_{i=1}^{n} |H_i|^{-1/2} exp(-D^2[y_p, y_i, H_i]/2) H_i{}^{-1}}{\sum_{j=1}^{n} |H_j|^{-1/2} exp(-D^2[y_p, y_j, H_j]/2)} \quad (6)$$

Furthermore the denominators in eq. 5 and eq. 6 cancel each other out. This induces the natural algorithm 1. The

Algorithm 1 $approxMode$ (calculation of eq. 1)

Require: start point $y_p \in \mathbb{R}^k$, n

1: $y_{p,t} = 0; H_{h,t} = 0;\ e = 0;$
2: **for** i=0; $n-1$ **do**
3: calculate $H_i^{-1}(y_i)$
4: $e = |H_i|^{-1/2} exp(-D^2[y_p, y_i, H_i]/2);$
5: $H_{h,t}$ += $eH_i^{-1}(y_i)$
6: $y_{p,t}$ += $eH_i^{-1}(y_i) y_i;$
7: **end for**
8: $y_p = H_{h,t}^{-1} y_{p,t}$

runtime complexity is mainly determined by the calculation of $H_i^{-1}(y_i)$, which involves two separate operations, the calculation of $H_{h,t}^{-1}$ and the determinant $|H_i|$. The computation of $H_i^{-1}(y_i)$ also involves the determination of H_i itself. Let us denote the corresponding complexity classes by $\mathcal{O}_{H_i^{-1}}$ and $\mathcal{O}_{H_i}$, respectively. With $\mathcal{O}_{H_h^{-1}}$ we shall denote the complexity class of determining $H_{h,t}^{-1}$ and with $\mathcal{O}_{|H_i|}$ that of calculating the determinant. Differentiating between both matrix inversions allows to incorporate inversion algorithms which exploit certain features of the matrices and thus differ in their complexity. Yet one also has to account for the elementary matrix operation, i.e. addition, matrix-vector multiplication and scalar product computation. We express this through the classes $\mathcal{O}_+, \mathcal{O}_{m*v}, \mathcal{O}_{v*v}$ respectively.

Lemma 1. *The complexity for a single execution of alg. 1 in a general mean shift calculation is $\mathcal{O}(n \cdot (\lambda + \kappa + \xi + \gamma + \rho + \phi) + \gamma + \eta)$, with $\kappa \in \mathcal{O}_{H_i^{-1}}, \lambda \in \mathcal{O}_{H_i}, \eta \in \mathcal{O}_{H_h^{-1}}$, $\xi \in \mathcal{O}_{|H_i|}$, $\gamma \in \mathcal{O}_{m*v}$, $\phi \in \mathcal{O}_+$ and $\rho \in \mathcal{O}_{v*v}$.*

Proof: A total of n iterations will be executed. Each iteration requires (in following order): calculating H_i, inverting H_i, calculating the determinant $|H_i|$ and determining the Mahalanabois distance D^2. So far this yields $\lambda+\kappa+\xi+(\rho+\gamma)+\phi$. Scaling each component of H_i^{-1} by e is equally complex as adding two matrices, i.e. ϕ. Afterwards one has to account for updating $H_{h,t}$, multiplying y_i by eH_i^{-1} and updating $y_{p,t}$. This results in $\phi + \phi + \gamma + \rho$, as updating y_p has the same complexity as calculating the inner product. Summarized for we have $n \cdot (\lambda + \kappa + \xi + \gamma + \rho + \phi)$ which is followed by η for inverting H_h and γ for calculating y_p. ∎

Remark. *For a general approach, using state-of-the-art algorithms, this complexity transforms to $\mathcal{O}(n \cdot (\lambda + k^3))$.*

The amount of iterations of alg. 1 can be limited by an upper limit m_{it} and a distance threshold t_d for y_p between two consecutive iterations (see alg. 4 for details). Let $\mathbb{E}_{it}(S)$ be the expected amount of alg. 1 repetitions, which only depends on the current sample S. This implies an overall complexity of $\mathcal{O}(\mathbb{E} \cdot n^2 \cdot (\lambda + \kappa + \xi + \gamma + \rho + \phi) + n\gamma + n\eta)$, note that this expression refers to the approximation of all n modes.
As all mode approximations do not rely on each other it is possible to execute them in parallel. This would scale the complexity down by $1/z$ with z being the amount of parallel processing units (PPUs).

A. Optimization for diagonal regular covariance matrices and massive parallelization

With the above analysis it becomes obvious that for diagonal regular covariance matrices the stated complexity can be reduced dramatically. From now we assume $H_i(y_i)) = diag(\sigma_1(y_i), ..., \sigma_k(y_i))$, which implies $\mathcal{O}_{H_i^{-1}} = \mathcal{O}_{H_h^{-1}} = \mathcal{O}(k)$ (i.e. only the non-zero elements along the diagonal must be inverted) and $\mathcal{O}_{|H_i|} = \mathcal{O}(k)$ (summing up only diagonal elements).
In order to explain our approach for GPU architectures we must extend out approach from above in terms of parallel. Let us assume that in addition to our z processing unit, each of them incorporates $\tau \leq n$ primitive units (PUs) (i.e. units without complex mechanisms as e.g. instruction prefetching or branch prediction). We will now distribute the iterations of the for-loop in alg. 1 over the PUs. Note that if each PU executes a subset $I \subseteq \{1, ..., n\}$ of iterations, we would obtain the previously mentioned parallelization factor $1/z$ in the case of $|I| = n, \tau = 1$, in case of $|I| = 1, \tau = n$ we

would obtain a spectacular boost of $1/n$ per PPU (as all for-loop evaluation would be executed in parallel). We will elaborate with great detail on such cases in the following sections.
Using this approach alg. 1 becomes alg. 2, where the index pu indicates a variable local to each PU and the function 'getPUIdx()' returns the index of the PU. This formulation

Algorithm 2 $approxMode$ (Parallel calculation of eq. 1)

Require: start point $y_p \in \mathbb{R}^k$, load factor $f_{load} \leq \tau$, τ, n
1: $step = \lceil n/f_{load} \rceil$
2: $y_{p,t} = 0; H_{h,t} = 0;$
3: i_{pu} =getPUIdx(); $\quad \rightarrow$ PU local work begins here
4: $i_{start,pu} = i_{pu} \cdot step$
5: $i_{end,pu} = (i_{pu} + 1) \cdot step$
6: $y_{p,t,pu}$ = 0;$H_{h,t,pu}$ = 0; e_{pu} = 0;
7: **if** $i_{end,pu} - i_{start,pu} < step$ **then**
8: $\quad i_{end,pu} = i_{end,pu} - i_{start,pu}$
9: **end if**
10: **for** i=$i_{start,pu}$; $i_{end,pu}$ **do**
11: $\quad$ calculate $H_i^{-1}(y_i)$
12: $\quad e = |H_i|^{-1/2} exp(-D^2[y_p, y_i, H_i]/2);$
13: $\quad H_{h,t,pu}$ += $eH_i^{-1}(y_i)$
14: $\quad y_{p,t,pu}$ += $eH_i^{-1}(y_i)y_i;$
15: **end for**
16: parallel reduction of $H_{h,t,pu}$ to $H_{h,t}$
17: parallel reduction of $y_{p,t,pu}$ to $y_{p,t}$
18: **if** $i_{pu} == 0$ **then**
19: $\quad y_p = H_{h,t}^{-1} y_{p,t}$
20: **end if**

does not require $n = l \cdot \tau$, but implies that each PPU must calculate a complete evaluation of eq. 1 for only a single point y_p. The load factor f_{load} indicates how many iterations, i.e. $\lceil n/f_{load} \rceil$, should be handled by a single PU. This algorithm will be executed in parallel by each PU on a PPU, it does not require a specific execution paradigm as e.g. a lockstep execution or any specific system architecture as e.g. a GPU SIMD environment. Allthough one would benefit from this strategy for a large n, it yields a significant drawback for small n as well as for a small number of PPUs with many PUs. For small n, i.e. ($n < \tau$), one would like to be able to process multiple evaluations, i.e. for different y_p, on a single PPU in order to reduce underutilization. The same holds for a small number of PPUs, i.e. $z < n$. This issue can be solved through the concept of virtual PPUs which will be explained in the next section.

Remark. *If $\tau \nmid n$, the algorithm will underutilize the available PUs independent of the chosen value for f_{load}. Only for $f_{load} = \tau$ will all PUs be utilized and the amount of iterations per PU minimized.*

Let us now analyze the complexity which we have achieved with alg. 2. Constants will only be carried into the analysis if they are induced through this algorithm, i.e. if they haven't occured in alg. 1. This way a direct comparison of both approaches remains possible. Furthermore due to the needed parallel reduction we assume a uniform memory access (UMA, [8]) architecture with a fixed cost c for every data access on scalar elements.

Theorem 1. *Alg. 2 exhibits a complexity of $\mathcal{O}(\lceil n/f_{load} \rceil \cdot (5k + \lambda) + 2\log(\tau) \cdot k + 2(\sum_{i=1}^{\log_2 \tau} \frac{\tau}{2^i}) \cdot k \cdot c + \aleph(k, \tau) + k)$. $f_{load} = \tau$ is the optimal choice. If $f_{load} \nmid n$ then at least one PU will not be fully utilized. In case of $log_2(\tau) \in \mathbb{N}$ we obtain $\aleph(k, \tau) = 0$.*

Proof: All initializations can be done in constant time (line 1-9), the for loop involves exhibits $\lceil n/f_{load} \rceil$ iterations with the same complexity as in alg. 1. Due to the diagonal matrices the complexity for the algorithm until line 15 is $\lceil n/f_{load} \rceil \cdot (5k + \lambda)$, as the PUs will execute their local for-loops in parallel. The interesting part are the parallel reductions in line 16 and 17. We will state the complexity only for line 16 as the same arguments hold for line 17. There are $\log_2(\tau)$ iterations involved in the reduction, if τ is not a power of two the parallel reduction (PR) will only process $l < k$ data elements with l being the nearest power of two. The remaining elements must be processed otherwise, e.g. sequentially, which induces an additional complexity term $\aleph(k, \tau)$ (which equals 0 in case that $\log_2(\tau) \in \mathbb{N}$). Each of the active PUs in one of the PR iterations has to sum up k elements, i.e. add $H_{h,t,pu'}$ of an adjacent PU pu' to its own copy. Thus in total there are $\log(\tau) \cdot k$ calculations. Yet, since each PU can access the memory of another one in a uniform way, we have to account for that with $(\sum_{i=1}^{\log_2 \tau} \frac{\tau}{2^i}) \cdot k \cdot c$. Since this holds for the second PR as well, we obtain the factor 2. Finally a single matrix multiplication remains which has a complexity of k.
In case if $f_{load} < \tau$ we obtain $n/\lceil n/f_{load} \rceil < \tau$, which in turn implies the existence of at least one underutilized PU. Trivially this holds as well for the case $n < \tau$. Note that $f_{load} = \tau$ is not sufficient for full utilization. Observe that $(l \cdot \tau + p = n) \Rightarrow n/\lceil n/\tau \rceil < \tau$. Thus only if additionaly τ divides n we obtain full utilization. ∎

Remark. *In order to force $\aleph(k, \tau) = 0$ one could pad n to a power of 2 with numerically feasible dummy data. Furthermore one has to restrict the available number of PUs to a power of 2 closest to τ, i.e. set $f_{load} = 2^x < \tau$. Yet this in turn would imply underutilization, thus such a decision must be evaluated carefuly.*

Although it might seem that the new algorithmic skeleton is much more complicated than the first one, we will show that with a fitting adaptation for the system architecture it can provide a significant improvement to the naive approach. Yet before that, we will increase the flexibility of

alg. 2.

III. Virtual parallel processing units

In order to circumvent the drawbacks of alg.2 in the context of small data sets as well as in the case of only a few PPUs with many PUs (e.g. GPU architectures). We modify the algorithm only marginally by subdividing the PUs on a PPU into virtual PPUs (vPPUs). For this purpose we introduce a new variable f_{vppu} which states how many of the PUs should form a vPPU. Thus the load factor becomes local to each vPPU.

Algorithm 3 $approxMode$ (Parallel calculation of eq. 1)

Require: start points $y_{p,j} \in \mathbb{R}^k$, load factor $f_{load} \leq \tau/f_{vppu}$, τ, n, f_{vppu}
1: $step = \lceil n/f_{load} \rceil$
2: $y_{p,t} = 0$;$H_{h,t} = 0$;
3: i_{pu} =getPUIdx(); $\rightarrow$ PU local work begins here
4: $j_{pu} = \lfloor i_{pu}/f_{vppu} \rfloor$;
5: $i_{start,pu} = (i_{pu} \mod f_{vppu}) \cdot step$
6: $i_{end,pu} = ((i_{pu} \mod f_{vppu}) + 1) \cdot step$
7: $y_{p,t,pu}$ = 0;$H_{h,t,pu}$ = 0; e_{pu} = 0;
8: **if** $i_{end,pu} - i_{start,pu} < step$ **then**
9: $i_{end,pu} = i_{end,pu} - i_{start,pu}$
10: **end if**
11: **for** i=$i_{start,pu}$; $i_{end,pu}$ **do**
12: calculate $H_i^{-1}(y_i)$
13: $e = |H_i|^{-1/2} exp(-D^2[y_{p,j_{pu}}, y_i, H_i]/2)$;
14: $H_{h,t,pu}$ += $eH_i^{-1}(y_i)$
15: $y_{p,t,pu}$ += $eH_i^{-1}(y_i)y_i$;
16: **end for**
17: vPPU-local parallel reduction of $H_{h,t,pu}$ to $H_{h,t}$
18: vPPU-local parallel reduction of $y_{p,t,pu}$ to $y_{p,t}$
19: **if** $i_{pu} \mod f_{vppu} == 0$ **then**
20: $y_p = H_{h,t}^{-1} y_{p,t}$
21: **end if**

Remark. *In order to keep the analysis simple we assume that $f_{vppu} \mid \tau$, the reader interested in the generic case can generalize our analysis with additional complexity terms in the same way we did with $\aleph(k, \tau)$.*

Thus the runtime complexity of alg. 3 is given by

Theorem 2. *Alg. 3 exhibits a complexity of $\mathcal{O}(\lceil n/f_{load} \rceil \cdot (5k + \lambda) + 2\log(\tau') \cdot k + 2(\sum_{i=1}^{\log_2 \tau'} \frac{\tau'}{2^i}) \cdot k \cdot c + \aleph(k, \tau') + k)$. $f_{load} = \tau/f_{vppu} =: \tau'$ is the optimal choice. If $(f_{load}/f_{vppu}) \nmid n$ then at least one PU will not be fully utilized. In case of $log_2(\tau') \in \mathbb{N}$ we obtain $\aleph(k, \tau') = 0$.*

Proof: Follows directly from theorem 1 and remark III. ■

Remark. *The complexity of alg.3 may seem identical to that of alg. 2, yet alg. 3 handles a set $\{y_{p,j}\}$ of data points!*

A general measure for PPU (and PU) underutilization can not be stated without defining the system architecture to use. An example for that would be a modern GPU, which exhibits a very specific scheduling policy with respect to the amount of PU local memory requirements. Whereas a cluster system may distribute the computation in an arbitrary node topology. Only by defining such parameters one can truly assess the term utilization. We will give a detailed example in section IV where we discuss the implementation for GPU architectures.
One should note that alg. 3 does not impose any restriction onto the matrix structures, amount of data points or system architecture. It rather depicts a flexible skeleton for mean shift computation which can be easily parallelized for arbitrary systems. We now state the complete mean shift algorithm. Alg. 4 is self-explanatory to the larger part. For every given data point y_i we approximate a corresponding mode $y_{p,i}$. The for-loop iterations are independent to each other, thus they (this includes $approxMode$ and $dist$) can be arbitrarily distributed among available PPUs. The algorithm $dist$ is discribed in alg. 5, its purpose is to calculate the Mahalanabois distance between two data points. The while-loop terminates if either the distance threshold t_d or the maximal number of iterations m_{it} has been reached. Afterwards the modes are grouped into ϵ-regions $\{\tilde{y}_p\}$ through algorithm $groupModes(\{y_{p,i}\}, \epsilon)$, which is stated in listing 6. This algorithm is inherently sequential and thus can't be effectively parallelized. Yet in the context of alg. 4 it can be exchanged for more efficient variants. Our evaluations in section V are based on alg. 6, as its runtime complexity

Algorithm 4 Mean shift clustering

Require: set of n data points $y_i \in \mathbb{R}^k$, t_d, ϵ, m_{it}
1: **for** i=0; $n-1$ **do** $\rightarrow$ arbitrary iteration-distribution over PPUs possible
2: $y_{p,i,prev} = 0$; $y_{p,i} = y_i$; $t_{d,i} = 0$, $c = 0$;
3: **while** $t_{d,i} > t_d \mid\mid c < m_{it}$ **do**
4: $y_{p,i,prev} = y_p$;
5: $y_{p,i} = approxMode(y_{p,i}, ...)$;
6: $t_{d,i} = dist(y_{p,i,prev}, y_{p,i}, ...)$;
7: $c+ = 1$;
8: **end while**
9: **end for**
10: $\{\tilde{y}_p\} = groupModes(\{y_{p,i}\}, \epsilon)$

Algorithm 5 $dist$

Require: data points $y_i, y_j \in \mathbb{R}^k$, covariance matrix H_i
1: $d = D^2[y_j, y_i, H_i]$

Algorithm 6 $groupModes$

Require: data points $y_{p,i} \in \mathbb{R}^k$, ϵ
1: $c = 0$;$exists = 0$;
2: **for** i=0; $n-1$ **do**
3: $exists = 0$;
4: **for** j=0; c **do**
5: **if** $dist(\tilde{y}_{p,j}, y_{p,i}, ...) < \epsilon$ **then**
6: $exists = 1$;
7: BREAK;
8: **end if**
9: **end for**
10: **if** $exists == 1$ **then**
11: add $y_{p,i}$ to list of $\tilde{y}_p$;
12: **end if**
13: **end for**

has proven to be feasible for our application. The signature wildcards "..." indicate that arbitrary algorithms can be inserted at that position as long as their signatures are identical up to the wildcard.
We conclude the section with following

Theorem 3. *Alg. 4 exhibits a complexity of* $\mathcal{O}(\mathbb{E}(S)/(z \cdot \tau/f_{vppu})(\lceil n/f_{load}\rceil \cdot (5k+\lambda) + 2\log(\tau')\cdot k + 2(\sum_{i=1}^{\log_2 \tau'} \frac{\tau'}{2^i}) \cdot k \cdot c + \aleph(k,\tau') + 2k) + \zeta)$. *With* $\zeta \in \mathcal{O}_{group}$ *being a function in the complexity class of alg. 6 and* z *the amount of PPUs.*

Proof: Almost every part follows from theorem 2. Alg. 3 allows to compute up to $z \cdot \tau/f_{vppu}$ approximation steps in parallel. Whereas one final exection of alg. 6 is required, thus the term $+\zeta$. ■

IV. Application to GPU Computation

We begin by stating our last assumption regarding matrix structures, let $H_i(y_i) = diag(\sigma_1 \exp(y_i^k), ..., \sigma_{k-1}\exp(y_i^k), \sigma_k)$ with σ_i being fixed parameters. Thus H_i is assumed to be diagonal and depends only on the last element of y_i (a^l denotes the l-th vector component).

Remark. *With this assumption we obtain* $\mathcal{O}_{H_i} = \mathcal{O}(k)$. *Thus the complexity of alg.3 becomes* $\mathcal{O}(\lceil n/f_{load}\rceil \cdot (6k) + 2\log(\tau')\cdot k + 2(\sum_{i=1}^{\log_2 \tau'} \frac{\tau'}{2^i}) \cdot k \cdot c + k)$ *for padded data sets of size* n.

The concept of PPU/PU can be directly translated to GPU architectures as SMX/SP (Cuda) or compute unit/processing element (OpenCL). But as mentioned before, many aspects about thread scheduling and memory architectures must be considered to utilize the efficiency of our approach. These techniques are well beyond the scope of this paper, the interested reader might refer to [9],[10],[11],[12] or [13] for further details.

Implementing alg. 6 was done in two succeeding stages using the OpenCL language, allowing its execution on multicore CPUs, NVidia GPUs as well as AMD GPUs. First we ported the algorithm naively, i.e. not considering coalesced global memory access and local bank conflicts. The second implementation utilizes preloading of datapoints into local memory and advanced synchronization mechanisms. In both cases the grouping of approximated modes was done on the host side using the system CPU. Due to the page limit of this paper we can describe our optimization techniques for the second variant only briefly. Alg. 7 will preload the datapoints

Algorithm 7 $approxModes$ OpenCL implementation structure

Require: start points $y_{p,j} \in \mathbb{R}^k$, load factor $f_{load} \leq \tau/f_{vppu}$, τ, n, f_{vppu}, preload_size
1: init variables;
2: **for** x=0; preload_iters **do**
3: preload batch of datapoints into local memory
4: Synchronization s_1
5: **for** i=$i_{start,pu}$; $i_{end,pu}$ **do** $\rightarrow$ lockstep exection
6: calculate $H_i^{-1}(y_i)$
7: $e = |H_i|^{-1/2} exp(-D^2[y_{p,j_{pu}}, y_i, H_i]/2)$;
8: $H_{h,t,pu}$ += $eH_i^{-1}(y_i)$
9: $y_{p,t,pu}$ += $eH_i^{-1}(y_i)y_i$;
10: **end for**
11: Synchronization s_2
12: vPPU-local parallel reduction of $H_{h,t,pu}$ to $H_{h,t}$
13: vPPU-local parallel reduction of $y_{p,t,pu}$ to $y_{p,t}$
14: **end for**
15: **if** $i_{pu} \mod f_{vppu} == 0$ **then**
16: $y_p = H_{h,t}^{-1} y_{p,t}$
17: **end if**

in sets of fixed size into local memory. As the local memory is available to all workitems in a workgroup, this might yield a speedup if multiple modes $\{y_{p,j}\}$ are approximated by one workgroup. Otherwise the workitems would access high-latency (i.e. $\approx$ 700 clock cycles) global memory. One should note the two distinct strategy elements, first the use of low latency memory and second the encapsulation of multiple node approximations on one workgroup. If one would choose to approximate a single mode per workgroup, there would be no gain from local memory. More or less surprisingly this approach yields little to no advance for computing devices capable of global memory broadcasts and lockstep execution, as the instructions are executing in lockstep. I.e. when all processing elements access the same memory position (local or global memory), this request will be delivered as an efficient broadcast to them. In case of a large number of scheduled workitems and no preloading, the global broadcast latencies could be effectively hidden and the preload approach would become contraproductive due to the imposed synchronization mechanisms. These

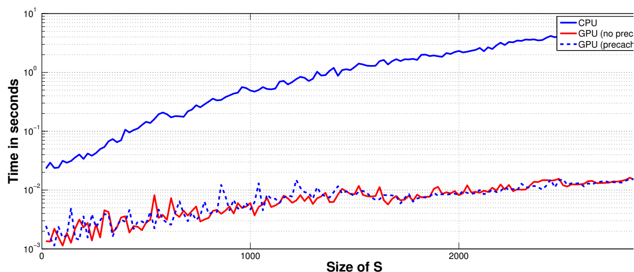

Figure 1. Execution speeds of alg. 4 and 7 on CPU and GPU. The GPU exhibits a speedup of up to ≈ 176 compared to the execution on CPU side.

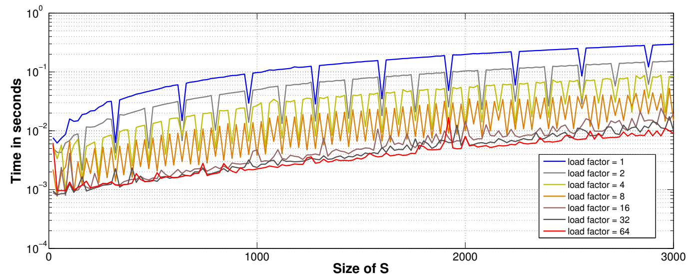

Figure 2. Execution speeds of alg. 4 and 7 on CPU and GPU. The GPU exhibits a speedup of up to ≈ 176 compared to the execution on CPU side.

synchronization elements are depicted as s_1 and s_2, they involve atomic integer operations and a semaphore shared among the workgroups workitems. Furthermore changes in the algorithms design easily lead to thread divergence and thus, in the context of s_1/s_2, to barrier divergence [14], [15] as well.

Another critical point of alg. 7 is the required amount of memory per workgroup. Multiple workgroups can be scheduled for execution on a compute unit, yet it is the amount of inbound workgroups which helps to mask the latencies of memory accesses. The number of inbound workgroups is determined by the available memory of a compute unit, the lower the memory consumption of a workgroup the more of them can be kept inbound. Thus if one decides to consume too much local memory for preloading, this number will be significantly reduced.

These aspects will become visible within our results in the next section. Yet, from a general perspective alg. 7 allows one to achieve high throughput in case of memory types with large latencies (e.g. zero-copy host RAM), additionally it provides the possibility to implement access patterns to local memory preloading if no local memory broadcasts are available. Furthermore our algorithm can be split across multiple GPUs in a single system.

V. Results

We evaluated our algorithms on a Radeon7970 which provides 32 compute units (PPUs) with a total of 2048 stream cores (PUs), 32KB of local memory per workgroup and 3GB of global memory. Thus each PPU holds 64 PUs. The host system provided a Core-i7 3820 3.6GHz CPU, 64GB RAM and was running ArchLinux 3.15.5-1 with the SimpleHydra SDK [16]. The parameters for the algorithm were identical for all our experiments; $\epsilon = 0$, $m_{it} = 100$, $t_d = 10^{-5}$. The choice for $\epsilon = 0$ will become clearer when we discuss our applied error measure. Due to the dynamic nature (with respect to e.g. temperature) of GPUs we averaged our results over 20 repetitions without changing the data. We generated synthetic samples S of varying sizes where each sample contained 3-dimensional integer tuples.

The graphs in Fig. 1 depict the execution speeds for CPU and GPU. It becomes clear that the GPU provides a significant boost of up to ≈ 176 compared to the CPU. Furthermore one can observe that the preloading strategy of alg. 7 does not yield any improvement. The overhead for caching and synchronization outweights the potential speed gain. We did not conduct the experiment for the CPU as it does not feature any form of shared memory (i.e. OpenCL shared memory requests are translated to global memory requests).

Fig. 2 shows the previously discussed effect of underutilization for various load factors. The optimal load factor should be 64 as the GPU provides 64 PUs per PPU, this becomes apparent through the shrinking average distance between consecutive load factors. The execution times seem to converge towards that of $f_{load} = 64$. Another interesting aspect is visualized in the equidistant "drops" along a graph for a single load factor. A close look at the x-axis (Fig. 3) reveals that, for e.g. $f_{load} = 1$, a significant boost of execution speed occurs every increment of 64 samples. This is explained by the fact that for $f_{load} = 1$ every compute unit will handle 64 mode approximations, thus at e.g. $64 \cdot k$ all PUs in exactly k PPUs (if available) will execute the same instruction in lockstep. This allows for coherent memory access in form of broadcasts, i.e. every PU in a PPU will request the same address in lockstep with all other active PUs on the GPU. For element counts which are not a multiple of 64, this situation does not occur. The same holds for loadfactors below 16, above 16 the effect of latency hidding takes over and masks memory access with a huge number of threads. One should note that the effect of underutilization and favorable memory access pattern are two distinct phenomena, which can be seen in Fig. 2 where "drop-free" line segments as well as "drops" are positioned significantly apart in a consistent way.

One critical point is the numerical stability of the GPU implementation as is does not utilize the same numerical precision as the CPU version. In order to measure the error between both implementations we appplied a simple

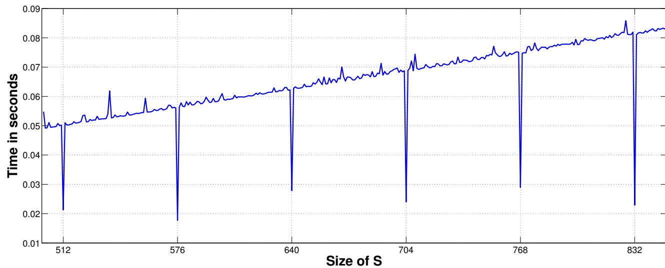

Figure 3. Execution speeds of alg. 4 on a GPU with a step size of 1 and load factor $f_{load} = 1$. At equidistant positions of 64 elements a significant boost of execution speed occurs. They are the result of favorable memory access patterns in situations of fully utilized PPUs.

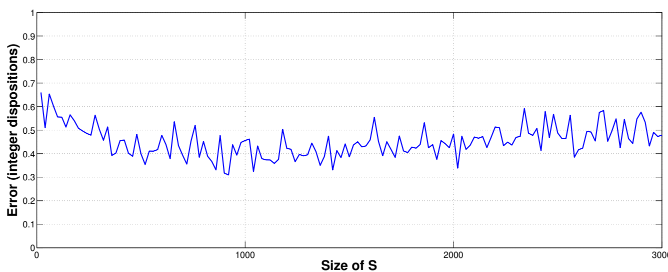

Figure 4. $E_{CPU-GPU}$ for alg. 4, the maximal error 0.7 clearly shows a difference in numerical behaviour. Yet the error remains negligible for the results even for large samples.

measure

$$E_{CPU-GPU} = \left(\sum_{i=1}^{|S|} |y_{p,i,CPU} - y_{p,i,GPU}|_{pos} \right) / |S| \quad (7)$$

with

$$|a, b|_{pos} = \sum_{i=1}^{k} |a^i - b^i| \quad (8)$$

being the amount of integer dispositions along all k dimensions. The results are depicted in Fig. 4, where it becomes visible that the error, although existing, becomes negligible as it never crosses 0.7 (even for large samples).

VI. Conclusion

In this paper we discussed the classic mean-shift algorithm for mode approximation. On one side we restricted ourself to gaussian kernels while on the other side we introduced a generic distance metric through covariance matrices. We provided an algorithmic skeleton in order to analyze the algorithms potential for arbitrary parallelization. This skeleton gave rise to an efficient (non-trivial) GPU implementation which yielded a speedup of up to 176. Preload strategies showed no benefit as the induced management overhead simply outweighted the fast memory access. Current computation devices (e.g. cell phones, car computers or mainstream PCs) are designed with an increasing focus on parallel computation. Our work provides a significant benefit for all algorithms which utilize mean shift methods, this includes especially methods for image processing. An example would be the HOG algorithm which can be accelerated up to 60ms for 1600x1200px images, yet its robustness relies (among other things) on the mean shift based approximation of modes. For CPU based implementations or naive parallelization approaches one would not be able to keep the 60ms. Which can be critical for e.g. realtime applications.
With the introduction of abstract terms as vPPUs or PUs one can embed the algorithm into different system structures, be it multicore CPUs or large distributed cluster systems. The stated generic runtime complexities enable one to estimate the gain for various system structures. Although we have idealized certain aspects for the runtime analysis, the algorithms themself were stated in a generic way, remarks have hinted towards the corresponding complexity analysis.
Future work should include analysis of the algorithms applicability for different structures than GPUs, e.g. ARM CPUs, Xeon Phi or distributed systems for large scale problems as well as an evaluation of it in specific image processing tasks (e.g. tracking problems on mobile devices). Furthermore it remains an open question which strategies should be followed for generic problems as e.g. non-diagonal covariance matrices. One could for example outsource the matrix inversions to other system components.
We provide our implementation per email request and hope that our work, besides being practical, will inspire other researchers.

References

[1] D. Comaniciu, "An algorithm for data-driven bandwidth selection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 25, no. 2, pp. 281–288, Feb 2003.

[2] D. Comaniciu and P. Meer, "Mean shift: a robust approach toward feature space analysis," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 24, no. 5, pp. 603–619, May 2002.

[3] K. Fukunaga and L. Hostetler, "The estimation of the gradient of a density function, with applications in pattern recognition," *Information Theory, IEEE Transactions on*, vol. 21, no. 1, pp. 32–40, Jan 1975.

[4] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," *CVPR*, vol. 1, pp. 886–893, 2005.

[5] S. Avidan, "Ensemble tracking," *2013 IEEE Conference on Computer Vision and Pattern Recognition*, vol. 2, pp. 494–501, 2005.

[6] S. Hommel, D. Malysiak, and U. Handmann, "Model of human clothes based on saliency maps," in *Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on*, Nov 2013, pp. 551–556.

[7] P. Li and L. Xiao, "Mean shift parallel tracking on gpu," *Pattern Recognition and Image Analysis*, p. 120?127, 2009.

[8] M. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, no. 9, pp. 948–960, Sept 1972.

[9] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2010.69

[10] AMD, "Amd accelerated parallel processing opencl programming guide," 2013.

[11] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[12] NVidia, "Opencl programming guide for the cuda architecture," 2012.

[13] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[14] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: A verifier for gpu kernels," *SIGPLAN Not.*, vol. 47, no. 10, pp. 113–132, Oct. 2012. [Online]. Available: http://doi.acm.org/10.1145/2398857.2384625

[15] E. Bardsley and A. F. Donaldson, "Warps and atomics: Beyond barrier synchronization in the verification of gpu kernels," 2013.

[16] D. Malysiak and U. Handmann, "An efficient framework for distributed computing in heterogeneous beowulf clusters and cluster-management," in *Computational Intelligence and Informatics (CINTI), 2014 IEEE 15th International Symposium on*, Nov 2014.