GPU–Accelerated Texture Analysis Using Steerable Riesz Wavelets

***Abstract*—Visual pattern recognition is a key research topic in the field of image processing and computer vision with many applications including medical diagnosis, identification and classification tasks. Texture analysis based on steerable Riesz wavelets is powerful, but requires computing pixel–wise operations resulting in a run time in the order of days when large volumes of data are processed. To overcome this limitation we propose a Graphics Processing Unit (GPU) based solution. A standard CPU version is used as starting point for the development of baseline GPU versions. To further increase the performance, and to overcome compute and memory limitations we apply a series of optimization techniques, leading to five versions in total. The best performing GPU solution ensures a speed–up of 93x for the parallelized section of the application and of 29.6x for the entire application. Furthermore, we show that a higher Riesz order and/or a higher image resolution further increases the speed–up.**

Keywords*—Medical imaging, GPU, parallel processing, texture analysis.

I. INTRODUCTION

Textured pattern recognition is a key research topic in computer vision. One of the fundamental concepts in pattern recognition is the characterization of the local organization of image scales and directions to identify visually different patterns [1]. Although much research has been carried out on this topic [2], [3], [4], [5], [6], [7], it has proven difficult to elegantly exploit the potential of local attributes for classification. Depeursinge et al. proposed an iterative multi–scale and rotation–covariant texture learning approach using steerable Riesz wavelets [8], [9]. The framework allows learning local computational models of patterns. Subsequently, the models yield feature vectors that are optimally discriminant for a given problem. Classification accuracies of up to 98.4% was reported on texture databases such as Outex[1]. However, the high accuracy comes at the expense of an algorithm run time of the order of several days for large databases. To improve its usefulness for image analysis, parallelization of computations on graphics processing units (GPU) is proposed in this paper.

GPUs are dedicated video processors, frequently used to accelerate a wide range of compute–intensive applications [10]. Typically, these applications rely on heterogeneous CPU–GPU hardware configurations. Whereas the CPU runs the sequential part, the GPU runs the parallel part by calling specific functions, called kernels, that use a very large number of threads. The GPU is organized into streaming multiprocessors, whereas each multiprocessor contains several cores and different types of memory (shared memory, registers, etc.) [11] which depend on the GPU architecture. Since the GPU is especially well–suited to address problems that can be expressed as data–parallel computations with high arithmetic intensity (the ratio of arithmetic operations to memory operations) [12], our goal is to explore the GPU–based acceleration of classification tasks relying on steerable Riesz wavelets.

[1]http://www.outex.oulu.fi/, as of 7 May 2015.

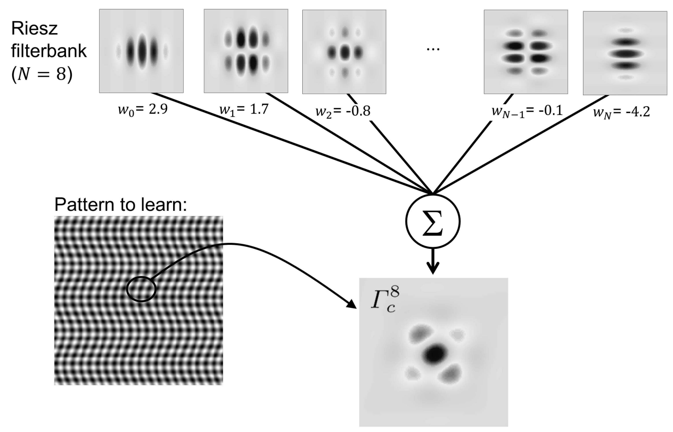


Fig. 1. An illustration of the process to determine the texture signature Γ_c^8 of a simple pattern (shown in the bottom–left corner) using an 8th–order Riesz filterbank (shown at the top) is presented here. Γ_c^8 is constructed as a weighted linear combination of Riesz filters and shown in the bottom–right corner. A quick visual inspection reveals that the reconstructed template is indeed similar to the basic pattern element in the original image [8].

II. METHODS

A. Texture Analysis

There exist a family of basic image filters, of which Riesz filters are a member, that can be used to reproduce any image pattern using a proper weighting of the filters being used. One may also do the reverse and decompose any pattern using the same filters. The weighting parameters needed to recompose the pattern may then be calculated and would exactly characterize the image in question. For any new pattern, the responses of aligned Riesz filters may be calculated and subsequently compared with those of known pattern classes via a classifier in order to determine if both depict the same pattern. This concept is at the basis of the classification task being carried out in this paper.

Nth–order Riesz wavelets are used here to learn texture signatures or, in other words, the basic pattern underlying a target texture. At the top of Fig. 1, a set of 8th–order Riesz filters are shown. When they are combined using learned weights $\boldsymbol{w}_c$, the basic pattern contained in the test image shown on the bottom–left of the figure is recreated and is shown on the bottom–right. The first step in the process of texture learning involves the convolution of Riesz filters with the training images containing the texture of interest in order

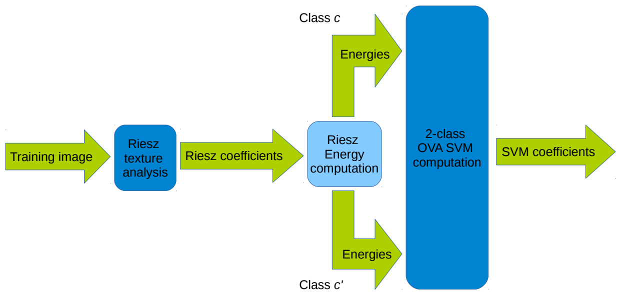


Fig. 2. Training images containing a target pattern are first convolved with Riesz filters to obtain Riesz coefficients. The Riesz energies are calculated from them and separated according to classes c and c'. They are then fed to a two–class SVM classifier using a OVA strategy to obtain the class–specific weights $\boldsymbol{w}_c$ of the linear combination of the Riesz filters for building the signature Γ_c. The process is repeated for all classes c to obtain C sets of coefficients.

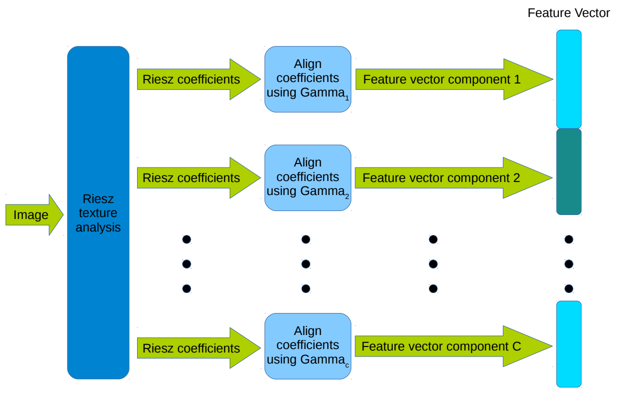

Feature Vector
Riesz coefficients
Align coefficients using Gamma$_1$
Feature vector component 1
Riesz coefficients
Align coefficients using Gamma$_2$
Feature vector component 2
Image
Riesz texture analysis
Riesz coefficients
Align coefficients using Gamma$_c$
Feature vector component C

Fig. 3. Riesz coefficients for training and test images are calculated and locally aligned using each texture signature, Gamma$_c$, in turn. The resulting coefficients for each Gamma$_c$ are concatenated to form the final feature vector for classification.

to obtain Riesz coefficients. The respective energies contained within them are then calculated. Subsequently, the energies of the coefficients of each Riesz filter are used to create Support Vector Machine (SVM) models (i.e., a separating hyperplane with the direction vector $\boldsymbol{w}_c$) while following a one–versus–all (OVA) supervised learning strategy [8]. The coefficients that the SVMs assigned to each Riesz filter (i.e., $\boldsymbol{w}_c$) correspond to a first estimate of the signature for that pattern. A new feature space is built based on the response of the multi–scale Riesz coefficients steered for each pixel to maximize the local weighted sum of Riesz coefficients. These maximum local responses are used to build the final feature vector that can be used for texture classification. Fig. 2 provides a visual representation of the texture analysis workflow that has just been described. Next, the process is repeated C times (i.e. for all textures classes in the database) and all C texture signatures are stored.

Local alignment of Riesz coefficients of a training or test image is then carried out for the training images using, one–by–one, the saved signatures Gamma$_c$. The resulting C aligned coefficients are then concatenated into a multi–dimensional feature space. A new SVM classifier is trained using the feature vectors of the training images. Finally, for any new pattern to be classified, its feature vector is computed as just described and based on this feature vector, the SVM classifier is used to assign the new pattern to one of the previously–learnt patterns. In the rest of this section, more details for the above process is provided.

1) ***Learning the texture signatures:*** The images are first convolved with Riesz filters in order to obtain the corresponding Riesz coefficients. For 2–D images, the Nth–order Riesz transform $\boldsymbol{\mathcal{R}}^N$ yield $N+1$ components as [13]:

$$\boldsymbol{\mathcal{R}}^N\{f\}(x)=\left(\mathcal{R}^{(0,N)}\{f\}(\boldsymbol{x}),\ldots,\mathcal{R}^{(n,N-n)}\{f\}(\boldsymbol{x}),\ldots,\mathcal{R}^{(N,0)}\{f\}(\boldsymbol{x})\right)^{\mathbf{T}}. \quad (1)$$

The local response of each component $\mathcal{R}^{(n,N-n)}$ of an image $f(\boldsymbol{x})$ rotated by an arbitrary angle θ is derived analytically from the above components of the Riesz filterbank via a steering matrix $\boldsymbol{A}^\theta$ as:

$$\boldsymbol{\mathcal{R}}^N\{f^\theta\}(\mathbf{0}) = \boldsymbol{A}^\theta\boldsymbol{\mathcal{R}}^N\{f\}(\mathbf{0}). \quad (2)$$

Multi–scale versions of these filterbanks are created by combining the Riesz transform with Simoncellis multi–resolution framework.

For classification, a weighting of the energies of the responses of the Riesz components $E(\mathcal{R}^{(n,N-n)}\{f\}(x))$ is required. The energies associated with each of the components is calculated. The goal is to build an optimal multi–scale texture signature Γ_c^N of class c from a linear combination of the Riesz components as:

$$\Gamma_c^N=w_1(\mathcal{R}^{(0,N)})_{s_1}+w_2(\mathcal{R}^{(1,N-1)})_{s_1}+\cdots+w_{J(N+1)}(\mathcal{R}^{(N,0)})_{s_J}. \quad (3)$$

where $s_j, j=1,...,J$ is the scale index and $\boldsymbol{w}$ contains the weights of the Riesz components. In Fig. 1, Γ_c^8 for a given texture of class c, order $N=8$ and a fixed scale is shown.

The training images available have C different but known textures. First, they are separated according to their class c, where $c \in \{1,2,...,C\}$. For each class c, there is a set c' such that $c' \in \{1,2,...,C\}-\{c\}$. Two–class SVMs are used C times to find the optimal separation between the Riesz energies of class c and c', which corresponds to weights $\boldsymbol{w}$ in Eq. (3).

2) ***Getting Texture Features for Training and Test Images:*** In this step, the Riesz coefficients obtained in the previous step are steered at each pixel position in order to locally alignment them.

For every image I at each position $\boldsymbol{x}_p$ (i.e., each pixel), the initial Riesz coefficients for a certain pattern are steered to the maximize the response of Γ_c^N in Eq. (3). The response of the signature steered by θ is:

$$\Gamma_c^{N,\theta} = \boldsymbol{w}^T\boldsymbol{A}^\theta\boldsymbol{\mathcal{R}}^N. \quad (4)$$

At the position $\boldsymbol{x}_p$, the θ_{dom} angle that maximizes the response of $\Gamma_c^{N,\theta}$ is:

$$\theta_{dom}(\boldsymbol{x}_p) := \underset{\theta\in[0,\pi]}{\arg\max}(\boldsymbol{w}^T\boldsymbol{A}^\theta\boldsymbol{\mathcal{R}}^N\{f\})(x_p). \quad (5)$$

A matrix $\Theta(\boldsymbol{x})$ is obtained for all $\boldsymbol{x}_p$. Subsequently, Riesz coefficients from all scales are steered using a unique angle matrix $\Theta(\boldsymbol{x})$. The above can be done analytically for each pattern [8]. The locally aligned Riesz coefficients for each pattern are then concatenated as shown in Fig. 3 in order

TABLE I. PERCENTAGE OF TIME SPENT BY EACH MAJOR COMPONENT WITHIN CPU BASED APPLICATION.

Component	%Time
Generation of Riesz energies	0.03
Creating matrices and computing SVM coefficients for ONE versus ALL (others) comparison	0.001
Alignment of signatures (see Equation 5)	99.289
Evaluation using SVM and post-processing	0.67

to obtain the final feature vector corresponding to image I. Finally, a new SVM classifier is trained using the computed feature vectors and it can subsequently be used for the final classification of patterns.

Table I outlines the main components of the algorithm and the corresponding percentage of the total run time. This analysis indicates that alignment of signatures is by far the most compute intensive operation of the application.

Therefore, the block diagram presented in Fig. 4 illustrates the most time–consuming workflow component within the algorithm. First, for each image pixel the computation of the maximum response of a given texture signature requires solving a Nth–order trigonometric polynomial (i.e., the lines of the steering matrix $\boldsymbol{A}_{\theta}$, see [8]). A Nth–order equation is then analytically solved for each pixel of the image. After performing a series of mathematical computations on the real solutions of the Nth–order correlated polynomial the responses of Γ_c^N are calculated. The first part of the computation involves applying trigonometric functions (arc-tangent, sine and cosine) on each real root, followed by element-wise multiplications on the resulting Mth–order vectors, where $M \leq N$ represents the number of real roots, while the second part involves the use of $4 \times (N+1)^3$ mathematical operations (element-wise multiplications and additions) at polynomial root level. The maximum is subsequently extracted and the rotation angles are derived from it.

Riesz coefficients alignment averaged 99.28% of the total run time, of which 0.43% corresponds to forming polynomials, 3.29% to solving polynomials, 3.61% to computing trigonometric operations, 92.43% to calculating responses and 0.24% to extracting the maximum response.

Since Riesz coefficients alignment is computationally expensive and all computations can be performed independently for all pixels, the GPU becomes a well-suited option for greatly decreasing run time and significantly improving the usefulness of the Riesz–based texture classification proposed by Depeursinge et al.

B. Baseline GPU–based Implementations

We first introduce a baseline GPU based implementation of the texture learning approach. Texture analysis is mostly used in medical diagnosis and clinical research, hence, due to the strict accuracy requirements, all computations are performed in double precision. The GPU based implementation (called *GPUBase*) covers the learning/detection stage, representing the actual computation of the feature vector. Since computing the responses Γ_c^N takes around 99.28% of the execution time, it represents the main focus of the parallelization activities.

A significant amount of data is required to reside on the GPU (requiring expensive copy operations), whereas the

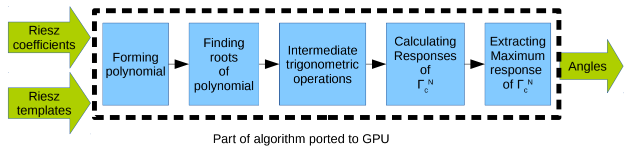

Riesz coefficients
Riesz templates
Forming polynomial
Finding roots of polynomial
Intermediate trigonometric operations
Calculating Responses of Γ_c^N
Extracting Maximum response of Γ_c^N
Angles
Part of algorithm ported to GPU

Fig. 4. Part of the workflow around the pixel–wise local alignment of Riesz coefficients using the Riesz templates, Gamma$_c$, is represented here. Riesz coefficients and templates are first used to form the polynomial to be solved in order to obtain the matrix of angles for local alignment. Next, the roots of the polynomial are found. Following some intermediate trigonometric operations, the responses of the templates are calculated. Finally, the maximal response is determined and the matrix of angles for local alignment is derived from it. Since this part of the algorithm is the most time–consuming and massively parallel (it must be computed for each image pixel independently), it is mapped on to the GPU for acceleration.

number of actual computations required for computing the responses Γ_c^N is small. Hence, since the compute intensity is significantly lower than the capabilities of current GPUs, the memory access latency cannot be hidden, but only mitigated with large data caches. This limits the throughput and we identify as main approach for performance improvement an increase in compute intensity [14]. To increase compute intensity the implementation covers also the computation of polynomial roots and the maximum response extraction, along with a series of other operations that include trigonometric operations applied to the computed roots.

In the execution configuration the threads and the thread–blocks are organized into 2–D structures, with a total number of threads equal to $dimX \times dimY$, where $dimX$ and $dimY$ represent the height and width attributes for the image. Each thread sets a number of N values in the feature vector V, where N is the Riesz order.

The *GPUBase* version is divided into two parts: the computation of polynomial roots and the feature vector computation. In the first part, each thread stores all $N+1$ coefficients of the associated polynomial, and all real roots are computed. To determine these roots, we combine Newton's method that allows us to approximate one root of a polynomial with Horner's method for polynomial long division [15]. In the second stage, the feature vector is built by regional averaging of the energies. Each thread updates locations (based on the number of real roots found by the thread) from the feature vector. Therefore, a repetitive loop structure is used to browse through the maximum N features handled by a thread. Each feature is computed iteratively, based on the previously determined values. This introduces three additional repetitive loop structures, leading to a significant increase in the amount of work performed by each thread.

Next, the local orientations of each template are optimized to maximize their response, which is carried out by aligning Riesz components based on the dominant orientation of the signatures [8]. For the *GPUBase* version these steps are included in the kernel, since no additional data are required.

C. Enhanced GPU–based Implementations

Since data in the *GPUBase* version are stored in the global memory, and the performance of the kernel is primarily

```
// global memory buffers
cosT[n * dimx * dimy];
sinT[n * dimx * dimy];
cossin[n*(n+1)* dimx * dimy];
template[n+1];
steermat[(n+1)*(n+1)*(n+1)];
orig[(n+1)* dimx * dimy];

void GPUBase(...)
{
...
R=roots(...);
for (int i=0; i<roots; i++)
  {
   tha[idx] = atan(R[idx]);
   cosT[idx] = cos(tha[idx]);
   sinT[idx] = sin(tha[idx]);
  }
for (int i=0; i<n+1; i++)
 for (int j=0; j<roots; j++)
   {
      cossin[index] =
       pow(cosT[idx],(n-i))*
       pow(sinT[idx],i);
   }
for(int i=0; i<n+1; i++)
 for(int j=0; j<n+1; j++)
  for(int k=0; k<n+1; k++)
   for(int l=0; l<roots; l++)
    {
     V[Vidx] += template[i]*
        steermat[steerIdx]*
        cossin[cossinIdx]*
        orig[origIdx];
    }
...
}
```

(a)

```
void GPUReg(...)
{
...
R=roots(...);
for (int i=0; i<roots; i++)
 {
  tha[idx] = atan(R[idx]);
  double cosT = cos(tha[idx]);
  double sinT = sin(tha[idx]);
 }
for (int j=0; j<n+1; j++)
  {
   double cossin =
     pow(cosT, (order-j))*
     pow(sinT, j);

   for (int k=0; k<n+1; k++)
    for (int l=0; l<n+1; l++)
     {
      V[Vidx] += template[i]*
        steermat[steerIdx]*
        cossin[cossinIdx]*
        orig[origIdx];
     }
  }
...
}
```

(b)

Fig. 5. Simplified kernel code for (a) *GPUBase*, and (b) *GPUReg*.

affected by the global memory bandwidth, we first address this aspect.

1) Register Usage: Registers are fast on chip memory, having almost no latency when read/write operations are performed. We use registers to store data that are otherwise repeatedly loaded: for the *GPUBase* version this strategy is applied for the values computed through trigonometric and associated operations. In each of the above mentioned cases, each thread requires loading and storing data from/to global memory. Since several intermediate computations are performed, by placing the partial results into registers, and merging the repetitive loop structures, a new enhanced implementation is obtained (*GPUReg*). Instead of having 7 repetitive instructions and 3 additional buffers stored in the global memory, we use only 4 repetitive instructions and 3 registers (Fig. 5). To further reduce global memory access, we introduce another version (*GPURegGlM*), in which we store the arc-tangent value for a root in an additional register. First the data are read from the register and stored into the global memory as it is required in the end, then, for the sine and cosine operations, the data cached in the register are used.

2) Shared Memory Usage: Beside registers, GPUs provide another fast on–chip memory that can be used for sharing data at block level, with low latency: shared memory. While optimizations with shared memory should in theory improve the performance in memory–bound applications, its effectiveness may be limited by the constraints imposed by the GPU compute architecture and limited on–chip memory capacity [16]. Specifically, since the storage of the majority of the data in shared memory could lead to an exceeding of the maximum amount of available memory when higher Riesz order and/or higher image resolution are employed, the utility of this type of memory is limited herein.

The starting point for the new kernel is the *GPUReg* version. Only one data set (describing the weights of the Riesz components based on SVM coefficients) is small enough to not exceed the maximum size of the shared memory. For the first strategy, *GPUtShM*, threads within a block read the values they handle from global memory and store them in the shared memory. The drawback is that we write in the shared memory only if the local index is less than $(N+1)$. Thus, if N is much lower than the total number of threads within a block, only a small number of threads writes in the shared memory array, leading to divergent branches and hence to serialization. Since access to shared memory is faster than global memory access, shared memory can also be used to store data that cannot be put into registers and are repeatedly accessed by a thread [17]. For this approach (*GPUShMTha*) we use again the *GPUReg* implementation as starting point.

We observe that there are data that are accessed multiple times by each thread. In *GPUReg* it was pre–allocated in the global memory such that each thread had a certain number of locations for storing its values. Because the values are computed entirely on the GPU and because the same data are not shared among multiple threads, we only change the memory space in which data are stored without any need of synchronization.

TABLE II. EXECUTION TIMES [S] OF IMPLEMENTATIONS FOR A SINGLE TIME STEP, WHEN RIESZ ORDER AND IMAGE SIZE ARE SET TO 8 AND 128X128 RESPECTIVELY.

Method	**Execution time [s]**	**Method speed–up**
GPUBase	0.088 ± 0.009	56-68x
GPUReg	0.058 ± 0.006	85-103x
GPURegGlM	0.065 ± 0.005	78-89x
GPUShM	0.105 ± 0.010	47-56x
GPUShMTha	0.071 ± 0.012	65-91x
CPUBaseline	5.413 ± 0.055	

III. RESULTS

We evaluate the different texture learning strategies using a hardware configuration based on an Intel Core i7 3.8 GHz processor with 64GB of memory and a NVIDIA GeForce GTX TITAN Black (Kepler architecture) GPU, configured with 48KB of shared memory and 16KB of L1 cache, compute capability 3.5 and the CUDA toolkit version 6.0. The overall workflow of the application is implemented in Matlab, and the parallelizable components are implemented in C++ for the CPU based version, and in CUDA for the GPU-based versions. We first analyse the execution times for a configuration with a Riesz order of 8 and an image size of 128×128 (the CPU C++ based version, *CPUBaseline*, was considered alongside the 5 GPU based versions). The results are shown in Table II.

All GPU–based versions lead to a significant speed–up compared to the CPU–based version. The best performance is obtained for *GPUReg*, leading to a significant reduction of the execution time (98.9%), as compared to *CPUBaseline*. We display in Table III the resource allocation for the various GPU–based versions. We chose the total number of threads

TABLE III. RESOURCE ALLOCATION FOR THE GPU BASED VERSIONS ON THE GTX TITAN BLACK CARD.

Method	**Threads per block**	**Blocks per SM**	**Reg. per thread**	**Shared memory per block [bytes]**	**Total number of 64 bit global load instr.**	**Total number of 64 bit global store instr.**	**Instr. executed**
GPUBase	1024	16	42	-	38437470	7890346	26679619
GPUReg	1024	16	46	-	23794569	7628589	22694175
GPUReGlM	1024	16	46	-	23784178	7628589	22694175
GPUShM	256	64	42	72	16220106	7628589	23550904
GPUShMTha	256	64	43	16384	23767794	7618198	23638963

TABLE IV. EXECUTION TIMES [S] OF THE ENTIRE MATLAB RIESZ–BASED TEXTURE CLASSIFICATION WHERE THE PARALLELIZED PARTS IMPLEMENTED IN C/C++ AND CUDA WERE INTEGRATED.

		Execution time [s]	
Component	**Sub-component**	**MATLAB + CPU**	**MATLAB + GPU**
Generation of Riesz energies		78.884	78.884
Creating matrices and computing SVM coefficients		3.32	3.32
Alignment of signatures	Pre-processing	1574.06	1574.06
	Alignment of Riesz coefficients	249431.04	2897.058
	Post-processing	2340.09	1729.721
Evaluation using SVM		1729.721	1729.721
Total		255157.115	8623.133

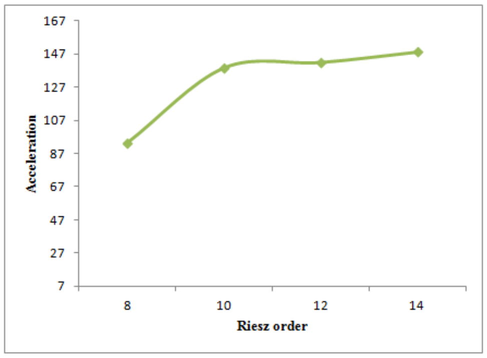
to be equal to $image\ width \times image\ height$. Regarding the distribution of threads and blocks: while for most of the versions we adopted a standard number of 1024 threads per block, for the versions that use shared memory the number of threads is limited by the maximum size of this type of memory.

The *GPUBase* version is bandwidth limited since there is a large global memory access requirement. *GPUReg* improves data reuse and reduces global memory accesses by employing additional registers. Computational time is 34% smaller than for the *GPUBase* version. The *GPURegGlM* version continues to reduce global memory load operations by using an additional register and as a result the execution time compared to the baseline *GPUBase* is decreased by 26%, but it increases slightly when compared to *GPUReg*: the larger number of registers limits the number of blocks of threads that can run simultaneously.

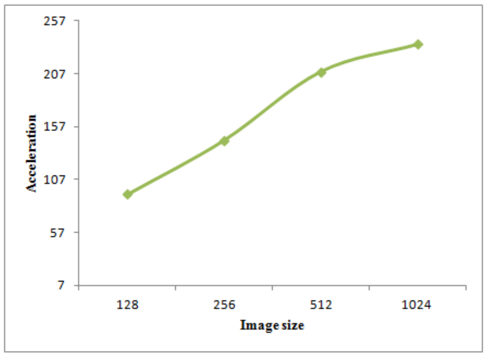
Next, shared memory is used in two versions to reduce latencies and global bandwidth usage: when data are not shared between threads but are repeatedly used by a thread (*GPUShMTha*), or when multiple threads share the same data (*GPUShM*). The first version only performs 19.3% faster than *GPUBase*. The kernel is limited by the shared memory size and therefore occupancy decreases. The second implementation performs slower than the baseline *GPUBase* version due to the massive warp serialization requirements.

Next, we determined the computation time of the entire application, when using the best performing GPU based version *GPUReg* and the CPU based version *CPUBaseline*. The results are displayed in Table IV. **The computation time decreases from 70.87 hours to 2.39 hours (speed–up of 29.58x)**.

The Riesz order and the image size have a considerable impact on the execution time results, since they affect the level of parallelism. We considered the best performing GPU based implementation and determined the speed–up of the parallelizable part for: (a) four different Riesz orders (8, 10, 12 and 14) with image size set constant at 128x128, and (b) four different image sizes (128×128, 256×256, 512×512, 1024×1024) with Riesz order $N = 8$ (Fig. 6). When the Riesz order increases from 8 to 14 the speed–up changes from 93x to 148x. Similarly, as the image size increases to 1024×1024, the speed–up increases to 235x. The results indicate that once the image resolution increases beyond a certain threshold the speed–up curve flattens since the occupation of the GPU decreases substantially (due to the larger number of registers).



(a)



(b)

Fig. 6. Comparison of speed–up obtained with the best GPU based implementation over the CPU based implementation with different (a) Riesz orders and (b) image dimensions.

IV. CONCLUSIONS

In this paper we introduced several GPU–based implementations for iterative texture learning. To obtain the best speed–up on the GPU, starting from a baseline version, we have applied a series of optimization techniques to overcome compute and memory limitations. Overall the implementation with the optimized register usage performed best, reducing the execution time from 70.87 to 2.39 hours.

The results obtained for different optimization techniques indicate that it is difficult to predict apriori the most successful strategy, and extensive tests are required to determine the best approach.

Future work will focus on a more efficient usage of the GPU cache memory and the development of a framework for using a multi–GPU strategy. Such a framework is required for high Riesz orders and/or large images, and should automatically distribute the workload to the various GPUs in a cluster.

Acknowledgment

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007–2013) in the context of the MD–Paedigree project under grant agreement 600932, and the Swiss National Science Foundation under grant PZ00P2_154891.

References

[1] On the existence of neurones in the human visual system selectively sensitive to the orientation and size of retinal images. *Journal of Physiology*, 203(1):237–260, 1969.

[2] An information processing approach to understanding the visual cortex. Technical report, Massachusetts Institute of Technology, 1980.

[3] Is local dominant orientation necessary for the classifcation of rotation invariant texture. In *Neurocomputing*, volume 116, pages 182–191, 2013.

[4] Textural features for image classification. *IEEE Transactions on Systems, Man and Cybernetics*, 3(6):610 – 621, 1973.

[5] Visual pattern discrimination. *IRE Transactions on Information Theory*, 8(2):84–92, 1962.

[6] Textural features corresponding to visual perception. *IEEE Transactions on Systems, Man and Cybernetics*, 8(2):460–473, 1978.

[7] Local features and kernels for classification of texture and object categories: A comprehensive study. *International Journal of Computer Vision*, 73(2):213–238, 2007.

[8] Rotation–covariant texture learning using steerable riesz wavelets. *IEEE Transactions on Image Processing*, 23(2):898–908, 2014.

[9] Rotation–covariant visual concept detection using steerable riesz wavelets and bags of visual words. In *SPIE Wavelets and Sparsity XV*, volume 8858, 2013.

[10] *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier, 2010.

[11] NVIDIA Corporation. *NVIDIA Kepler GK110 Architecture Whitepaper*, 2010.

[12] NVIDIA Corporation. *CUDA, Compute unifed device architecture Programming Guide*, v5.5 edition, 2013.

[13] Steerable pyramids and tight wavelet frames in $L_2(R^d)$. *IEEE Transactions on Image Processing*, 20(10):2705–2721, 2011.

[14] Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1522–1523, 2014.

[15] *Numerical analysis*. Springer, 1997.

[16] Optimizing stencil computations for nvidia kepler gpus. *International Workshop on High-Performance Stencil Computations*, pages 1–7, 2014.

[17] Double precision stencil computations on kepler gpus. *System Theory, Control and Computing*, pages 123 – 127, 2014.