GPGPU Volume Classification using SimpleOpenCL

Oscar Amoros[1], Sergio Escalera[1,2], Anna Puig[1], and Maria Salamó[1]

[1]*Dept. Matemàtica Aplicada i Anàlisi, Universitat de Barcelona, Spain* [2]*Centre de Visió per Computador, Spain*

[1]*Dept. Matemàtica Aplicada i Anàlisi, Universitat de Barcelona, Spain*

[2]*Centre de Visió per Computador, Universitat Autònoma de Barcelona, Spain*

E-mail: morousg@gmail.com, sescalera@cvc.uab.es, anna@maia.ub.es, maria@maia.ub.es

Abstract

In volume visualization, the definition of the regions of interest is inherently an iterative trial-and-error process finding out the best parameters to classify and render the final image. In this work, we present a general framework for training multi-class classifiers using Error-Correcting Output Codes. Moreover, we propose a GPGPU parallelization system using SimpleOpenCL, an Open-Source library we created to make easier the use of OpenCL. Results show accurate classification results as well as good speed ups.

Keywords: Volume visualization, Multi-class Classification, GPGPU, Parallel Computing, OpenCL.

1 Introduction

Knowledge expressiveness of scientific data is one of the most important visualization goals. The abstraction process the final user should carry out in order to convey relevant information in the underlying data use to be a difficult task. Automatic and user-guided segmentation strategies based on image processing are used to obtain classified data sets. In this paper, we propose a general framework of supervised statistical classification methods to label on-demand multiple regions of interest (see Fig. 1). The framework is composed by a pre-learning stage and an on-demand testing stage included in the renderer. The learning step gets a subset of pre-classified samples to train a set of Adaboost classifiers [1], which are codified as TFs, and combined in an ECOC design [2]. Next, the testing stage multi-classifies and labels a subset of volume classes based on user interaction. The label mapping defines clusters of the selected classes, and then it assigns optical properties and importance values to the final output classes to be visualizated. Moreover, we propose a GPGPU parallelization system using SimpleOpenCL, an Open-Source library we created to make easier the use of OpenCL. Results show accurate classification results as well as good speed ups. The rest of the paper goes as follow: Section 2 reviews the ECOC framework and presents our methodology for volume labelling. Section 3 explains implementation and the SimpleOpenCL library. Section 4 shows the experimental results, and finally, Section 5 concludes the paper.

2 Multi-class volume labelling

In this section, we present our automatic system for multi-class volume labelling.

2.1 ECOC overview

Given a set of N classes (volume structures or regions with certain properties) to be learnt in an ECOC framework, n different bi-partitions (groups of classes) are formed, and n binary problems over the partitions are trained. As a result, a codeword of length n is obtained for each class, where each position (bit) of the code corresponds to a response of a given classifier h (coded by +1 or -1 according to their class set membership, or 0 if a particular class is not considered for a given classifier). Arranging the codewords as rows of a matrix, we define a *coding matrix* M, where $M \in \{-1, 0, +1\}^{N\times n}$. Fig. 2(a) and (b) show a volume data set example and a coding matrix M, respectively. The matrix is coded using 15 classifiers $\{h_1, ..., h_{15}\}$ trained using a few voxel samples for each class of a 6-class problem $\{c_1, ..., c_6\}$ of respective codewords $\{y_1, ..., y_6\}$. The classifiers h are trained by considering the pre-labelled training data samples $\{(\rho_1, l(\rho_1)), ..., (\rho_k, l(\rho_k))\}$, for a set of k data samples (voxels in our case), where ρ is a data sample and $l(\rho_k)$ its label. For example, the first classifier h_1 is trained to discriminate c_1 against c_2, without taking into account the rest of classes. During the decoding or testing process, applying the n binary classifiers, a code X is obtained for each data sample ρ in the test set. This code is compared to the base codewords ($y_i, i \in [1, .., N]$) of each class defined in the matrix M, and the data sample is assigned to the class with the *closest* codeword. In fig. 2(b), code X is compared to the class codewords using the Hamming Decoding, and the test sample is classified by class c_1 with a measure of 0.5.

2.2 All-pairs multi-class learning

Given a set of pre-labelled samples for each volume structure, we use the one-versus-one ECOC design to train the set of all possible pairs of labels. An example for a 6-class foot problem is shown in Fig. 2(a) and (b). The positions of the matrix M coded by +1 are considered as one class for its respective classifier h_j, and the positions coded by -1 are considered as the other one (e.g. the first classifier is trained to discriminate c_1 against c_2).

2.3 ECOC submatrix definition

Given a volume that can be decomposed into N different possible labels, we want to visualize in rendering time the set of labels requested by the user. For this purpose, we use a small set of ground truth voxels features to train the set of $N(N-1)/2$

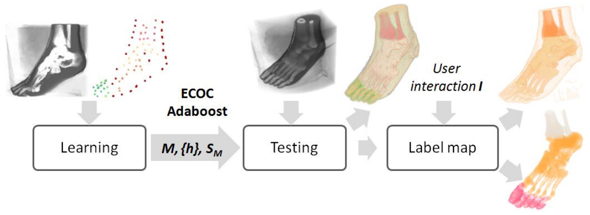
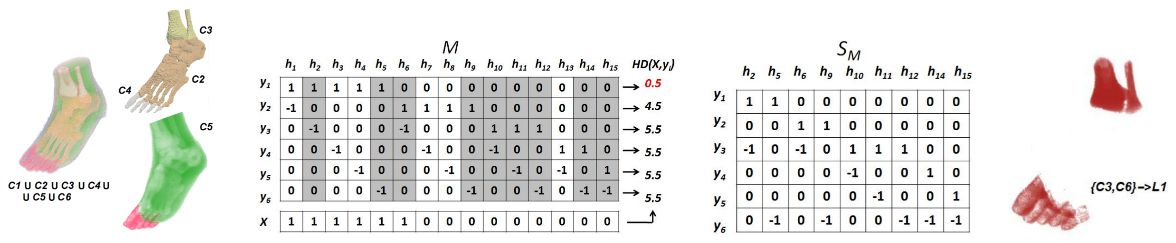


Figure 1: Visualization framework.

Adaboost binary problems that defines the one-versus-one ECOC coding matrix M of size $N \times n$. Then, let us define the interaction of the user as the set $I = \{I_0, .., I_z\} = \{\{c_i, .., c_j\}, ..\{c_k, .., c_l\}\}$, where $I, |I| \in \{1, .., N\}$ is the set of groups of labels selected by the user, and I_0 contains the background (always referred as c_1) plus the rest of classes not selected for rendering, $I_0 = \{c_i\}, \forall c_i \notin \{I_1, .., I_z\}, \bigcup_{c_i \in I} = \{c_1, .., c_N\}, \bigcap_{c_i \in I} = \emptyset$. Then, the submatrix $S_M \in \{-1, 0, +1\}^{N\times Z}$ is defined, where $Z \leq n$ is the number of classifiers selected from M that satisfies the constraint $h_i | \exists j, M_{ji} \in \{-1, 1\}, c_j \in I \setminus I_0$. For instance, in a 6-class problem of 15 one-versus-one ECOC classifiers (see Fig. 2(b)), the user defines the interaction $\{c_3, c_6\}$, resulting in the interaction model $I = \{\{c_1, c_2, c_4, c_5\}, \{c_3, c_6\}\}$ in order to visualize two different labels, one for background and other one for those voxels with label c_3 or c_6. Then, from the original matrix $M \in \{-1, 0, +1\}^{6\times 15}$, the submatrix $S_M \in \{-1, 0, +1\}^{6\times 9}$ is defined, as shown in Fig. 2(c). Note that the identifier i of each classifier h_i in S_M refers to its original location in M. Finally, we use the Loss-Weighted decoding (LW) [3] to obtain the final ECOC submatrix classification.

2.4 Label mapping

Given the submatrix S_M and the user interaction model I, after classification of a voxel ρ applying the LW decoding, the obtained label $c_i, i \in \{1, ..N\}$ is relabelled applying the mapping $L_M(I, c_i) = \begin{cases} l_1 & \text{if } c_i \in I_1 \\ ... & \\ l_z & \text{if } c_i \in I_z \end{cases}$, where $l_i, i \in \{1, .., z\}$ allows to assign RGBα or im-



M

	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}	h_{11}	h_{12}	h_{13}	h_{14}	h_{15}	$HD(X,y_i)$
y_1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	→ 0.5
y_2	-1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	→ 4.5
y_3	0	-1	0	0	0	-1	0	0	0	1	1	1	0	0	0	→ 5.5
y_4	0	0	-1	0	0	0	-1	0	0	-1	0	0	1	1	0	→ 5.5
y_5	0	0	0	-1	0	0	0	-1	0	0	-1	0	-1	0	1	→ 5.5
y_6	0	0	0	0	-1	0	0	0	-1	0	0	-1	0	-1	-1	→ 5.5
X	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	

S_M

	h_2	h_5	h_6	h_9	h_{10}	h_{11}	h_{12}	h_{14}	h_{15}
y_1	1	1	0	0	0	0	0	0	0
y_2	0	0	1	1	0	0	0	0	0
y_3	-1	0	-1	0	1	1	1	0	0
y_4	0	0	0	0	-1	0	0	1	0
y_5	0	0	0	0	0	-1	0	0	1
y_6	0	-1	0	-1	0	0	-1	-1	-1

(a) (b) (c) (d)

Figure 2: (a) True labels for a foot volume of six classes; (b) One-versus-one ECOC coding matrix M for the 6-class problem. An input test codeword X is classified by class c_1 using the Hamming Decoding; (c) Submatrix S_M defined for an interaction set $I = \{\{c_1, c_2, c_4, c_5\}, \{c_3, c_6\}\}$; (d) Visualization in the new label space.

portance values to all voxels that belong to the corresponding selected classes in I_i. As an example, applying the interaction model $I = \{\{c_1, c_2, c_4, c_5\}, \{c_3, c_6\}\}$ for the 6-class problem of Fig. 2(a), we obtain the submatrix S_M of Fig. 2(c). Applying the Loss-Weighted decoding over S_W, and the mapping function $\{\{c_1, c_2, c_4, c_5\}, \{c_3, c_6\}\} \longrightarrow \{0, 1\}$, the new volume representation is shown n Fig. 2(d).

3 Implementation

Here, we describe our proposed classifier representation, analyze its parallelization possibilities, and describe the novel SimpleOpenCL library.

3.1 Adaboost Look up table

We propose to define a new and equivalent representation of Adaboost classifier that facilitate the parallelization of the testing. We define the matrix V of size $3 \times |\rho| \cdot L)$, where $|\rho|$ corresponds to the dimensionality of the feature space and L the number of weak classifiers used in Adaboost training step. First row of V codifies the weight values of weak classifiers. In this sense, each position i of the first row of V contains the weight value for the feature $mod(i, |\rho|)$ if $mod(i, |\rho|) \neq 0$ or $|\rho|$, otherwise. The next weight value for that feature is found in position $i + |\rho|$. The positions corresponding to features not considered during training are set to zero. The second and third rows of V for column i contains the values of polarity and threshold used in a decision stump weak classifier training. Thus, in our proposal, each "weak classifier" is codified in a channel of a 1D-Texture.

3.2 GPU and Multi-CPU implementation

In order to make the application usable, we aim to improve the testing stage execution times as much as possible. The critical section in our application is a triple *for-loop* that traverses all the data. Each iteration is data independent with each other so they can be executed concurrently. Each iteration can be divided into two steps: a binary classification and a multi-class final decision. The latter is the multi-class decision made by using the ECOC matrix and the results of the first binary step.

One of the languages we have considered is ANSI C + OpenMP because not all computers have an OpenCL capable GPU. We have also considered testing Apple's GCD (Grand Central Dispatch) API, that passes the management of the threads to the system and takes in account the work load of the CPU cores to reduce context switching. At the programmer level GCD substitutes the threads with queues. It allows the programmer to only focus on deciding which part of the code will be synchronous or not regarding the main program, using lots of lightweight queues that will feed a few system controlled threads.

On the other hand, we have also analyzed OpenCL due to the huge speedups it can deliver when running on high end GPU's and the possibility of OpenGL integration. Nevertheless, productivity continues to be a major concern. Then, CUDA could be an option but we preferred to stick with OpenCL's compatibility and portability between GPU's and CPU's. We maintain a correlation of one Work Item per one sample. It would have been feasible to use more than a Work Item per sample in the above mentioned steps in order to achieve a more fine-grained parallelization and a more scalable code. Nevertheless, at the end of the two steps we only obtain a value per sample. As a consequence, the extra Work Items need to communicate through local memory and so, the performance is drastically reduced. In addition, maintaining the same Work Group and Work Item dimensions through all the process allowed us to code a single kernel, integrating also an initial gradient calculation step.

3.3 SimpleOpenCL library

SimpleOpenCL is an OpenSource library and Google code project [1] that we created. It is written in ANSI C with single device performance and portability goals in mind. The main goal of SimpleOpenCL has been reducing the code needed to run the experiments on the GPU with OpenCL, but also supports managing CPU devices. With OpenCL we can control the data flow between different kinds of computing devices that use different types of interconnect with the CPU and main memory, compile code for each of them, send the proper binary code for each of them, order execution etc. This is a very powerful tool since allows for computer architects to create any sort of accelerator and make it compatible with existing code. It is necessary because there are some technology constraints when increasing CPU performance. The tendency is to increase the parallelism and heterogeneity of the computing architectures, so OpenCL gives a way to recycle code that can be executed on all of them. OpenCL is an open standard managed by Khronos group [2]. OpenCL provides so much control over the system that it requires a lot of code. So with SimpleOpenC we are providing two levels of programming. The first level provides the simplest way to do an OpenCL program, but doesn't allow to do some very low level improvements and optimizations. For that purpose we provide a second level, where the programmer has more control, but also has to write more code, although less than with plain OpenCL.

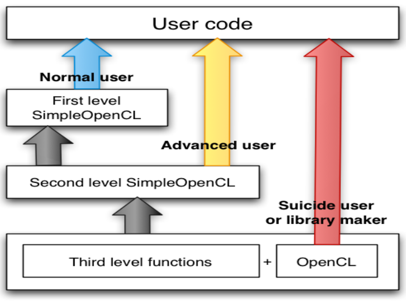


Figure 3: SimpleOpenCL programming levels.

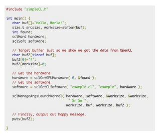
```
cl_event sclManageArgsLaunchKernel( sclHard hardware,
                                    sclSoft software,
                                    size_t *global_work_size,
                                    size_t *local_work_size,
                                    const char* sizesValues,
                                    ... );
```

Figure 4: SimpleOpenCL first level function example.

As seen in Fig. 3 the first level uses internally second level functions, and the second level uses both OpenCL and some utilities or third level functions. At the first level, we provide a function that allows to execute a kernel in a single line, passing standard C pointers as arguments, and some information about them in a printf fashion. That way, the function knows what to do with the parameters and automatically does all the repetitive work a developer should do with native OpenCL. This reduces lots of code into a single line.

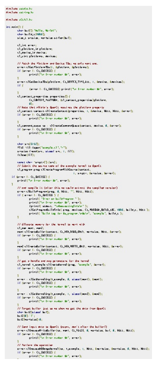
As seen in Fig. 4 we use a variable argument function where the variable arguments are preceded by a string whose contents are explained in the wiki manual of SimpleOpenCL OpenSource project web page. The function internally will create all the necessary buffers attached to the "hardware" context, copy the data, set all the kernel arguments, launch the kernel and capture an event on it to query the time it lasted, wait for the kernel to finish execution, update the host pointers that

[1]http://code.google.com/p/simple-opencl/
[2]http://www.khronos.org/



```
#include "simpleCL.h"

int main() {
   char buf[]="Hello, World!";
   size_t srcsize, worksize=strlen(buf);
   int found;
   sclHard hardware;
   sclSoft software;

   // Target buffer just so we show we got the data from OpenCL
   char buf2[sizeof buf];
   buf2[0]='?';
   buf2[worksize]=0;

   // Get the hardware
   hardware = sclGetGPUHardware( 0, &found );
   // Get the software
   software = sclGetCLSoftware( "example.cl", "example", hardware );

   sclManageArgsLaunchKernel( hardware, software, &worksize, &worksize,
                              " %r %w ",
                             worksize, buf, worksize, buf2 );

   // Finally, output out happy message.
   puts(buf2);

}
```

(a)



```
#include <stdio.h>
#include <string.h>

#include <CL/cl.h>

int main() {
        char buf[]="Hello, World!";
        char build_c[4096];
        size_t srcsize, worksize=strlen(buf);

        cl_int error;
        cl_platform_id platform;
        cl_device_id device;
        cl_uint platforms, devices;

        // Fetch the Platform and Device IDs; we only want one.
        error=clGetPlatformIDs(1, &platform, &platforms);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }
        error=clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, &devices);
        if (
                (error != CL_SUCCESS) printf("\n Error number %d", error);
        }
        cl_context_properties properties[]={
                CL_CONTEXT_PLATFORM, (cl_context_properties)platform,
                0};
        // Note that nVidia's OpenCL requires the platform property
        cl_context context=clCreateContext(properties, 1, &device, NULL, NULL, &error);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }
        cl_command_queue cq = clCreateCommandQueue(context, device, 0, &error);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }

        char src[8192];
        FILE *fil=fopen("example.cl","r");
        srcsize=fread(src, sizeof src, 1, fil);
        fclose(fil);

        const char *srcptr[]={src};
        // Submit the source code of the example kernel to OpenCL
        cl_program prog=clCreateProgramWithSource(context,
                                                  1, srcptr, &srcsize, &error);
    if (error != CL_SUCCESS) {
        printf("\n Error number %d", error);
    }
        // and compile it (after this we could extract the compiled version)
        error=clBuildProgram(prog, 0, NULL, "", NULL, NULL);
        if ( error != CL_SUCCESS ) {
                printf( "Error on buildProgram " );
                printf("\n Error number %d", error);
                fprintf( stdout, "\nRequestingInfo\n" );
                clGetProgramBuildInfo( prog, devices, CL_PROGRAM_BUILD_LOG, 4096, build_c, NULL );
                printf( "Build Log for %s_program:\n%s\n", "example", build_c );
        }

        // Allocate memory for the kernel to work with
        cl_mem mem1, mem2;
        mem1=clCreateBuffer(context, CL_MEM_READ_ONLY, worksize, NULL, &error);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }
        mem2=clCreateBuffer(context, CL_MEM_WRITE_ONLY, worksize, NULL, &error);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }
        // get a handle and map parameters for the kernel
        cl_kernel k_example=clCreateKernel(prog, "example", &error);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }
        error = clSetKernelArg(k_example, 0, sizeof(mem1), &mem1);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }
        error = clSetKernelArg(k_example, 1, sizeof(mem2), &mem2);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }
        // Target buffer just so we show we got the data from OpenCL
        char buf2[sizeof buf];
        buf2[0]='?';
        buf2[worksize]=0;

        // Send input data to OpenCL (async, don't alter the buffer!)
        error=clEnqueueWriteBuffer(cq, mem1, CL_FALSE, 0, worksize, buf, 0, NULL, NULL);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }
        // Perform the operation
        error=clEnqueueNDRangeKernel(cq, k_example, 1, NULL, &worksize, &worksize, 0, NULL, NULL);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
```

(b)

Figure 5: "Hello world" (a) SimpleOpenCL and (b) OpenCL.

have to be updated with device results and return an OpenCL event to query execution times. This behavior is what would be expected on a typical C sequential execution, but OpenCL provides more advanced execution options. We can enqueue several kernel executions on different queues for different devices without waiting for every single kernel. We can wait for all of them or for some of them in a given queue or for a given kernel to finish. All that possibilities are accessible through the second level functions. An example just to see the proportion of the difference between OpenCL and SimpleOpenCL is shown in Fig. 5. In fact, in the OpenCL version there are 15 extra missing lines and it doesn't include printing error names, that would add a huge function. The larger the code, the bigger the code reduction.

3.3.1 SimpleOpenCL future work

The development road for SimpleOpenCL is to provide a simplified set of advanced functionality on second level, to be used internally on first level. The proper management of memory transfers between host and device is a very important matter since every transfer has a minimum of something like 6000 cycles overhead when using PCIe devices like discreet GPU's. The ideal is to transfer data once, do all the processing needed and then read the results. Nevertheless, some advanced functionality to be added on the future is advanced memory transfer management. You can find an elaborated idea of it for CUDA on Google code project GMAC [3]. If there where an OpenCL version of GMAC it would be ideal to use it inside second or first level SimpleOpenCL functions since it resolves very low level performance and programability issues. The only problem is that GMAC may need to manually add support for each new device that appears. By now, SimpleOpenCL relies only on a working OpenCL 1.1 implementation regardless of the devices being used.

4 Simulations and Results

This section describes the experimental setup and shows the performance evaluation in terms of classification accuracy and execution time.

4.1 Setup

• **Data**: We used three data sets, *Thorax* data set of size 400×400×400 represents a MRI phantom human body; *Foot* and *Brain*[4] of sizes 128×128×128 and 256×256×159 are CT scans of a human foot and a human brain, respectively.

• **Methods**: We use the one-versus-one ECOC design, Discrete Adaboost as the base classifier, and we test it with different number of decision stumps. For each voxel sample ρ, we considered eight features: x, y, z coordinates, the respective gradients, g_x, g_y, g_z, the gradient magnitude, $|g|$ and the density value, v. The system is compared in C++, OpenMP, GCD, and OpenCL codes.

• **Hardware/Software**: For the CPU versions we used a Core i5 750 processor and for the GPGPU an NVIDIA GTX 470. The viewport size is 700×650. We used the MoViBio software by the GIE Group at the UPC university.

• **Measurements**: We compute the mean execution time from 500 code runs. For the accuracy analysis, we performed 50 runs of cross-validation with a 5% stratified samplings.

4.2 Classification accuracy analysis

Fig. 6 shows the classification accuracy of the framework for the data sets considering different number of decision stumps $\mathcal{M}$. The accuracies

[3]http://code.google.com/p/adsm/

[4]http://www.voreen.org, http://www.slicer.org/archives

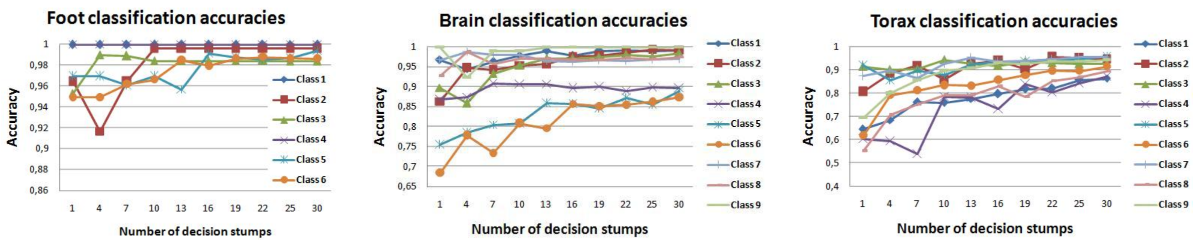


Figure 6: Classification accuracies for different number of decision stumps in the Adaboost-ECOC framework.

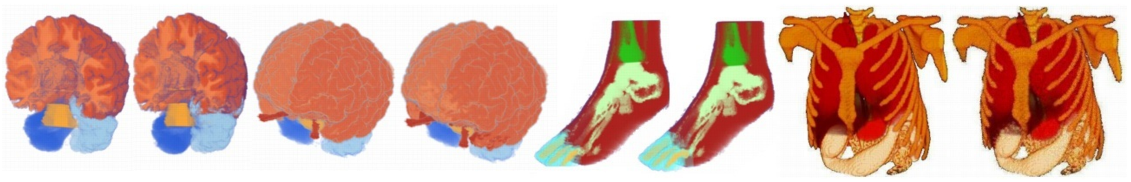


Figure 7: Comparison of the pre-labelled (left) and the classified data set (right) for the data sets.

are shown individually for each volume structure. From Fig 6 one can see that even for different complexity of volume structures, most of the categories obtain upon 90% of accuracy. Qualitative results are shown in Fig. 7.

4.3 Execution time analysis

We compared the time performance of our GPU testing with the CPU implementations based on sequential C-code, OpenMP, and GCD approaches. In table 1, the averaged time of the 500 executions. We employ different sized data sets with several number of classes to be labelled and distinct user-selections. Our proposed SimpleOpenCL-based optimization has an average of speed up of 109x, 31.11x, and 31.14x over the sequential C-coded algorithm, the OpenMP and the GCD based algorithms, respectively. In table 1, we can observe that time values are proportional to three features: the data set sizes, the number of classes,N, and the number of classifiers, Z, used for the selected classes. Note that we obtain real-time in the *Foot* data set.

5 Conclusions

We proposed o a two-level GPU-based labelling algorithm based on the novel SimpleOpenCL library that computes in time of rendering voxel labels using the ECOC framework with the Adaboost classifier. After a training step using few volume voxel features from different structures, the user is able to ask for different volume visualizations and optical properties. Additionally, to exploit the inherent parallelism of the proposal, we implemented the testing stage in C++, OpenMP, GCD, and GPU-OpenCL. Our results indicate that the proposal has the potential to deliver worthwhile accuracy and speeds up execution time.

Data set	N	Sel. classes	Z	CPU	OpenMP	GCD	OpenCL
Foot	3	2	2	0.387	0.111	0.111	0.008
	9	9	36	8.319	1.787	1.777	0.091
Brain	9	2	15	39.396	11.190	11.177	0.358
	9	9	36	96.859	27.642	27.557	1.263
Thorax	2	2	1	26.849	7.604	7.600	2.694
	9	9	36	971.915	270.751	269.751	7.763

Table 1: Testing step times in seconds for a C implementation running on CPU, parallel CPU implementations in OpenMP and GCD and GPGPU implementation in OpenCL C.

References

[1] J. Friedman, T. Hastie, R. Tibshirani, Additive logistic regression: a statistical view of boosting, Annals of Statistics 28.

[2] S. Escalera, D. Tax, O. Pujol, P. Radeva, R. Duin, Subclass problem-dependent design of error-correcting output codes, in: IEEE Transactions in Pattern Analysis and Machine Intelligence, Vol. 30, 2008, pp. 1–14.

[3] S. Escalera, O. Pujol, P. Radeva, On the decoding process in ternary error-correcting output codes, IEEE Transactions on Pattern Analysis and Machine Intelligence 32 (2010) 120–134.