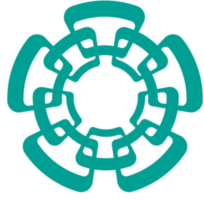


CENTER FOR RESEARCH AND ADVANCED STUDIES
FROM THE NATIONAL POLYTECHNIC INSTITUTE

Cinvestav Zacatenco

# A parallel texture analysis algorithm based on auto-mutual information for analyzing medical images

Thesis by:

**Lisa Pal**

in partial fulfillment of the
requirements for the degree of:

**Master of Science
in Computer Science**

Dr. Amilcar Meneses Viveros, Co-Director
Dr. Wilfrido Gómez Flores, Co-Director

Mexico City, Mexico November, 2020

© Copyright by
Lisa Pal
2020

*To my mother*

# Acknowledgements


- To Consejo Nacional de Ciencia y Tecnología (CONACyT), for the economic support provided to me during the period of my master's degree studies.
- To Centro de Investigación y Estudios Avanzados del Instituto Politécnico Nacional(CINVESTAV-IPN), for the opportunity of being part of their graduate studies.
- To CINVESTAV-Tamaulipas, for providing the appropriate infrastructure.
- To the Computer Science Department, for all the new experiences.
- To my thesis supervisors, Dr. Amilcar Meneses Viveros and Dr. Wilfrido Gómez Flores, for accepting me as their thesis student and guiding me through my work.
- To my thesis committee members, Dr. Cuauhtemoc Mancillas López and Dr. Mario Garza Fabre.
- To Eng. José de Jesús de Zapata Lara for helping me with technical problems.

# Contents

**Contents** i

**List of Figures** iii

**List of Tables** v

**List of Algorithms** vii

**Resumen** vii

**Abstract** ix

**1 Introduction** 1
- 1.1 Background and motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Research questions . . . 3
- 1.4 General and Particular Aims . . . 4
- 1.5 Research Methodology . . . 4
- 1.6 Organization of the thesis . . . 5

**2 Theoretical Framework** 7
- 2.1 Texture Analysis . . . 7
- 2.2 Ranklet Transform . . . 8
- 2.3 Auto-mutual Information . . . 10
- 2.4 Parallel Architectures . . . 13
  - 2.4.1 Flynn's taxonomy . . . 14
- 2.5 Parallel Programming Models . . . 15
  - 2.5.1 Shared Memory Model . . . 16
  - 2.5.2 Threads Model . . . 16
  - 2.5.3 Distributed/ Message Passing Model . . . 17
  - 2.5.4 Hybrid Model . . . 17
- 2.6 OpenMP . . . 18
  - 2.6.1 The Fork-Join Model . . . 19
  - 2.6.2 Thread Scheduling . . . 19
- 2.7 CUDA . . . 20
  - 2.7.1 Thread Hierarchy . . . 21

2.7.2 CUDA Memory Model . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.8 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

**3 State of the Art** **27**
3.1 Related works in image analysis . . . . . . . . . . . . . . . . . . . . . . 27
3.2 Related works in high performance computing . . . . . . . . . . . . . . . 29
3.2.1 Multicore processors . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.2.2 Graphics processing unit . . . . . . . . . . . . . . . . . . . . . . . 29
3.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

**4 Proposed approach** **31**
4.1 Sequential Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.2 Parallel Implementation on CPUs . . . . . . . . . . . . . . . . . . . . . . 36
4.3 Parallel Design on GPUs . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.3.1 Ranklets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.3.2 MI-based features . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.4 Heterogeneous implementation . . . . . . . . . . . . . . . . . . . . . . . 44
4.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

**5 Results** **47**
5.1 Infrastructure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.2 Timings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2.1 OpenMP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2.2 CUDA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
5.2.3 Heterogeneous implementation . . . . . . . . . . . . . . . . . . . . 59
5.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

**6 Conclusions** **63**
6.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

**Bibliography** **67**

# List of Figures


1.1 Running time of the AMI method as a function of the image area [1]; ROI-region of interest. . . . . 3

2.1 *Left:* Pixel intensity values in a region covered by a mask of resolution $r = 4$. *Right:* Pixel rank values, $\pi(p)$ of the left image [2]. . . . . 9
2.2 Three orientations for a $r = 4$ ranklet mask : horizontal (H), vertical (V) and diagonal (D), with their corresponding treatment (T) and control (C) subsets [1]. 10
2.3 Image displacements for computing the directional auto-mutual information. Dark gray and light gray lines mark the image displacements left/right (horizontal) and up/down (vertical), respectively. For an image of size $8 \times 8$, seven displacements are performed in the vertical and horizontal directions [1]. 13
2.4 Single Instruction, Single Data [3]. . . . . 14
2.5 Single Instruction, Multiple Data [3]. . . . . 14
2.6 Multiple Instruction, Single Data [3]. . . . . 15
2.7 Multiple Instruction, Multiple Data [3]. . . . . 15
2.8 Shared memory model. . . . . 16
2.9 Threads model. . . . . 17
2.10 Message passing model. . . . . 18
2.11 Hybrid model. . . . . 18
2.12 Fork-join model [4]. . . . . 19
2.13 Threads [5]. . . . . 21
2.14 Grid [5]. . . . . 22
2.15 Layout of a kernel call [5]. . . . . 23
2.16 CUDA memory model [6]. . . . . 24

3.1 Sequence of steps in texture classification using ranklets and SVM classifer. . 28

4.2 Each AMI feature corresponds to an element in the AMI feature vector. For every orientation and window size, two AMI features get computed. In this case, three orientations and four window sizes will produce a vector of size 24. 34
4.3 Loop hierarchy for the two stages of AMI. . . . . 36
4.4 Parallel regions for the AMI texture feature extraction stage. In the first parallel region, a single thread calculates the MI for every pair of subimages obtained from a displacement. In the second parallel region, a vector sum reduction obtaines the AMI feature. . . . . 37

4.5 For a $8 \times 8$ pixel image, we fix block size. As many blocks need to be launched to ensure there is one thread per pixel. In this figure, block dimensions were fixed to be (2,2) and therefore, a grid of 16 blocks will be launched. Threads are represented with the blue arrows, and blocks with the red squares. . . . . 40
4.6 Sub-images considered in downward displacements (yellow) and upward displacements (blue). No new images are created. . . . . . . . . . . . . . . . . . 42
4.7 Sequential addressing method for vector reduction. . . . . . . . . . . . . . . . 43
4.8 Heterogeneous approach: stage 1 is computed in a single GPU. The three resulting ranklet transforms are copied back to host. Stage 2 of the algorithm is computed in all the GPUs available. Each GPU computes one of horizontal, vertial and diagonal feature for a single window size. Lastly, the features are copied back to host. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

5.1 Average execution time for AMI OpenMP implementation for window resolution of 8, for all image sizes. This trendline was used to predict execution time of sequential implementation for image sizes 2048, 4096 and 8192. . . . . . . . 50
5.2 Average execution time for AMI sequential C++ and OpenMP implementations for a fixed window resolution of 8 and image sizes up to 1024. Relative standard deviation magnified by 5. . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.3 OpenMP speedup to C++ sequential code for all window resolutions and image sizes up to 1024 pixels per side. . . . . . . . . . . . . . . . . . . . . . . . 52
5.5 Semi-logarithmic plot of the measured execution times of the AMI for all implementations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.6 Execution times for AMI feature extraction using a different number of NVIDIA cards for images sizes 128 to 1024 (top) and from 2048 to 8192 (bottom). . . 60

# List of Tables

5.1 Execution times for the sequential and OpenMP implementations of AMI along with their corresponding percentage standard deviations. Values in red were predicted from the trend in Figure 5.1. The highest and lowest achieved speedups are highlighted in blue and pink respectively. . . . . . . . . . . . . . 50
5.2 Execution times for the MI kernel, the reduction kernel, and the total time for the computation of mutual features. Note that these times assume that a ranklet transformed image was given as an input, and therefore, excludes the computation time of ranklet transforms. . . . . . . . . . . . . . . . . . . . . 54
5.3 Execution times for computation of mutual information feature vector in OpenMP and CUDA. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.4 Execution times for ranklet transform with window resolutions 8 (left) and 32 (right) in OpenMP and CUDA. The lowest for each image size shown in bold. . 56
5.5 Ranklet execution times for CUDA implementation with different block configurations. The configurations yielding the best and worst times are highlighted in blue and pink respectively. . . . . . . . . . . . . . . . . . . . . . . 57
5.6 Execution times for the computation of AMI features for different block configurations and image size $1024 \times 1024$ (left) and $2048 \times 2048$ (right). The configurations yielding the best and worst times are highlighted in pink and blue respectively. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.7 Execution times of vector reduction for sequential and CUDA versions. The CUDA versions were executed with 128 and 256 bytes of shared memory. . . 59
5.8 Execution times and speedups using 1, 2 and 3 NVIDIA graphics cards. . . . 59

# Resumen

## Un algoritmo paralelo de análisis de textura basado en auto-información mutua para análisis de imágenes médicas

por

**Lisa Pal**
Cinvestav Zacatenco
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, 2020
Dr. Amilcar Meneses Viveros, Co-Director
Dr. Wilfrido Gómez Flores, Co-Director

El análisis de características de textura es una forma muy utilizada para obtener información a partir de imágenes médicas. Las aplicaciones van desde la segmentación de estructuras anatómicas específicas y la detección de lesiones, hasta la diferenciación entre tejido patológico y tejido sano. Un método de extracción de características de textura propuesto recientemente, llamado Auto Información Mutua (AMI, por sus siglas en inglés), demostró superar a otros nueve métodos en el contexto de la clasificación de lesiones mamarias en imágenes de ultrasonido. Sin embargo, tiene el factor limitante de que es doce veces más lento que ellos y no es escalable para su aplicación a tamaños de imágenes médicas más grandes, por ejemplo, mamografías. Hoy en día, las arquitecturas multinúcleo están ampliamente disponibles y generalmente brindan una gran aceleración cuando se trata de operaciones vectoriales. Sin embargo, una de las principales dificultades del uso de estas tecnologías suele consistir en problemas relacionados con la gestión de la memoria, como tener un espacio limitado para tipos específicos de memoria, o la brecha entre la velocidad del procesador y la memoria. Es importante implementar algoritmos de múltiples núcleos que sean escalables para poder usar tantos procesadores como estén disponibles. En este trabajo, proponemos varios diseños paralelos para el método AMI, utilizando OpenMP como una herramienta de paralelización de CPU de múltiples núcleos, y CUDA para la programación de unidades de procesamiento gráfico, con el objetivo de obtener vectores de características a partir de imágenes médicas de gran tamaño en un tiempo razonable. Como prueba de concepto, se procesarán mamografías con alta resolución espacial.

# Abstract

## A parallel texture analysis algorithm based on auto-mutual information for analyzing medical images

by

**Lisa Pal**
Cinvestav Zacatenco
Center for Research and Advanced Studies from the National Polytechnic Institute, 2020
Dr. Amilcar Meneses Viveros, Co-advisor
Dr. Wilfrido Gómez Flores, Co-advisor

The analysis of texture features is a widely used way to obtain information from medical images. The applications range from segmentation of specific anatomical structures and the detection of lesions, to differentiation between pathological and healthy tissue. This thesis focuses on the application to the breast lesion classification. A recently proposed texture feature extraction method, called Auto-Mutual Information (AMI), proved to outperform nine other methods in the context of breast lesion classification. However, it has the limiting factor that it is twelve times slower than them, and not scalable for its application to larger medical image sizes. Nowadays, multi-core architectures are widely available and generally provide a great speedup when it comes to vector operations. However, one of the main hardships of using these technologies often consist of issues related to memory management, such as having limited space for specific types of memory, or the poor processor-memory speed gap. It is important to implement multi-core algorithms that are scalable to be able to use as many processors as are available. In this work, we propose several parallel designs for the AMI algorithm, using OpenMP as a multi-core CPU parallelization tool, and CUDA for the programming of graphic processing units, with the aim of obtaining feature vectors from large medical images in a reasonable amount of time.

# 1 Introduction

## 1.1 Background and motivation

FEATURES are parts or patterns of an object in an image that help to identify it. A feature vector is used to characterize and numerically quantify the contents of an image. The definition of feature is very generic, as it will depend on the task at hand. For example, features used to recognize faces will be based on facial criteria, and will be obtained differently than features used to recognize human organs in an image. As you can see, there is a wide variety of contexts where feature extraction can be applied. In this work, we focus on medical images.

Generally, we want to extract features from an image to later identify a certain property of the image. The analysis of texture features is a widely used way to obtain information from medical images. The applications range from segmentation of specific anatomical structures and the detection of lesions, to differentiation between pathological and healthy tissue. This thesis focuses on the application to breast image analysis. Feature extraction still remains a major bottleneck for many implementations due to its high computational cost; this is especially the case for those algorithms that are the most robust to various image transformations [7]. The recently published work in [1] develops a new feature extraction method for breast lesion classification called Auto-Mutual Information (AMI). This method was compared with nine commonly used texture extraction methods proposed in literature for breast ultrasound analysis, outperforming them. This suggests that AMI could be further applied to other breast images obtained through a different imaging modality. One such

example would be the higher resolution mammographies. However, the AMI algorithm as it is currently implemented, would take an unreasonable amount of time to finish executing for such large images. Therefore, while AMI has been proven to be a suitable algorithm for breast image analysis, it is still lacking optimizations that could make it more widely applicable. Due to this, we will be considering the AMI as a starting point for our work.

## 1.2 Problem Statement

Image processing algorithms, in particular texture feature extraction, are often a compute intensive process, particularly in images with high spatial resolution. Let us consider the AMI algorithm, which is performed in two stages in cascade: (1) computing the ranklet transform, to obtain an intensity-invariant texture representation at different scales and orientations, and (2) extracting texture features by calculating the auto-mutual information on vertical and horizontal directions. Both of these stages encompass multiple operations (explained in Chapter 2) that depend on the image dimensions, making AMI a very compute intensive algorithm.

The AMI has been exclusively applied to breast ultrasound images (BUS), which are relatively small-sized images [1]. However mammographies are high resolution images, also widely used for detecting breast cancer. The sensitivity of this imaging modality depends significantly on age and breast density. According to a study conducted in [8], ultrasound had higher sensitivity than mammography in women younger than 45 years, whereas mammography had a higher sensitivity than ultrasound in women older than 60 years. This means that breast image analysis should be done on images obtained through different modalities, and of different spatial resolutions. BUS images are in the order of $2^8$ pixels in each dimension, whereas mammograms in the order of $2^{12}$ pixels in each dimension. In the original paper of AMI, the execution time was reported to be 12 times higher compared to the other feature extraction techniques. It was also experimentally demonstrated, for the case of BUS images, that from the total computation time, 90% was dedicated to auto-mutual information calculation, and the remaining time to compute ranklets. Figure 1.1 shows the computation time of AMI for BUS images. The graph only shows the processing time for up to $10^5$ pixels of the BUS image area, whereas mammograms consider $10^7$ pixels. Extrapolating these results to the case of other medical images with high spatial resolution, it follows that obtaining an image description in terms of AMI-based texture features is computationally inefficient.

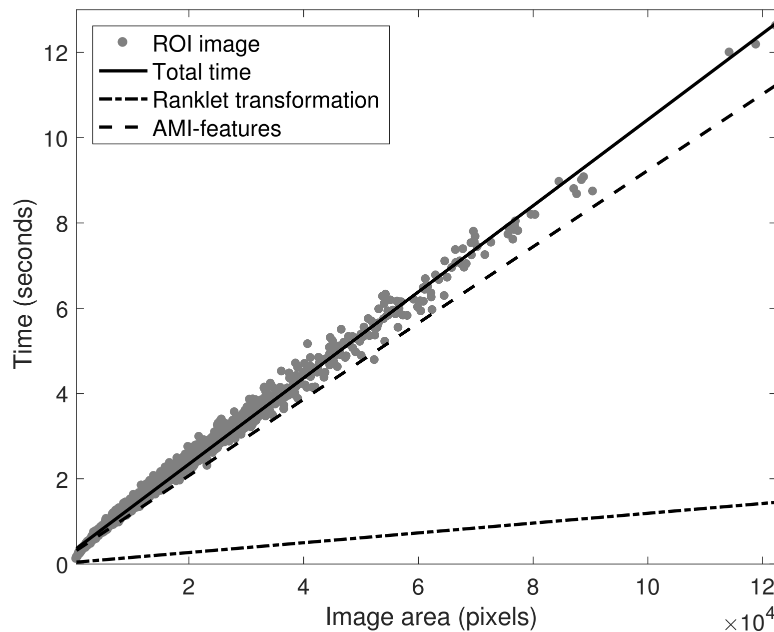



**Figure 1.1.** Running time of the AMI method as a function of the image area [1]; ROI- region of interest.

Image processing with parallel computing is an alternative way to solve problems that require large processing times or handle large amounts of data in an acceptable time. The main idea of parallel processing is to divide a problem into several independent subtasks and solve them concurrently, in a way that the total time can be divided among the total tasks, in the ideal case. Hence, this thesis addresses the design and implementation of a fully parallel version of the AMI-based texture description algorithm. For proof of concept, AMI-based texture features will be calculated from high-resolution mammograms.

## 1.3 Research questions

We seek to answer the following research questions by the end of this study:

- Is it possible to design a parallel version of the AMI algorithm?
- What is the parallelization scheme to apply to the AMI algorithm in order to efficiently use CPU and GPU devices?
- Is the use of high performance computing resources for the execution of the AMI algorithm justifiable?

- How much better does the parallel AMI algorithm perform in comparison to the sequential version?

## 1.4 General and Particular Aims

The general aim of this thesis is the following:

To design and implement a parallel version of a recently proposed texture extraction algorithm based on auto-mutual information for analysing medical images.

To achieve the general aim, the following particular aims were defined:

1. To select the technologies to be used to parallelise the auto-mutual information algorithm.
2. To define the parallelisation strategy.
3. To analyse the runtime of the sequential auto-mutual information algorithm and of the new parallelised version.

## 1.5 Research Methodology

To achieve the objectives planned in this thesis, the following research methodology was followed:

1. **The study of the AMI algorithm proposed in the original paper and its implementation in C++.**
    - The state of the art regarding image processing and feature extraction was revised.
    - The working for AMI sequential algorithm was understood.
    - Sequential implementation of AMI in C++ was coded.
2. **The study of parallelization technologies and techniques with subsequent parallel AMI implementations.**
    - The OpenMP API was explored.
    - The CUDA API was learned.
    - General parallelization strategies and approaches were studied.

- AMI implementation in OpenMP was designed and implemented.
- AMI implementation in CUDA was designed and implemented.

3. **Evaluation of the methods proposed.**
   - Execution time measurements of the sequential, OpenMP and CUDA implementations were carried out.
   - A comparison of the improvements in performance was done.

## 1.6 Organization of the thesis

This thesis consists of six chapters. The remaining five are briefly described next:

- **Chapter 2 - Theoretical Framework:** in this chapter we explain all the background concepts that are required to understand the work being developed. Topics from both image analysis and parallel computing are explained, as well as more technical details about hardware architecture.
- **Chapter 3 - State of the Art:** we list related works that have been studied previously.
- **Chapter 4 - Proposed Approach:** in this chapter we give the in-depth details of our parallel implementation designs, touching on the architectural reasons of why certain design choices were made.
- **Chapter 5 - Results and Discussion:** a description of the tests conducted, how they were carried out, and their results, are reported. We give an insight on why such findings are reasonable.
- **Chapter 6- Conclusions:** we present the conclusions obtained in this thesis, as well as the aspects that could be improved in future work.

# 2

# Theoretical Framework

THIS thesis seeks to improve on the performance of the auto-mutual information algorithm proposed in [1], where it was initially applied to breast ultrasound images for breast lesion classification. Therefore, in this chapter, we first provide the details to understand how to obtain a feature vector from an image using AMI. Subsequently, the various parallel computing technologies and architectures are described, providing insight into how the different hardware components play a role in software optimization.

## 2.1 Texture Analysis

The terms image processing and image analysis are different. On the one hand, image processing is a type of signal processing, where both the input and output are images. It performs operations on an image in order to enhance it, mainly to transform the contrast, brightness, resolution, and noise level of an image. On the other hand, image analysis is a technique used to obtain quantitative data from an image, usually segmenting a digital image based on features such as colour, density, or texture. The output of image analysis is usually a feature vector, which is an abstract representation of an input image. A third task is image classification, where the output of the image analysis step is used to assign one text label to the input image, and the labels are taken commonly from a fixed set of categories.

One way to distinguish between different objects of an image is by their textures. Texture can be defined as a repeating pattern of local variations in image intensity. It cannot be defined for a single pixel, since it requires contextual information. Then, the objective of

texture analysis is to describe the spatial-local variations of intensity employing a vector of characteristics, which numerically describes texture attributes such as roughness, smoothness, silkiness, or bumpiness as a function of the spatial variation in pixel intensities.

Texture analysis can be performed in the spatial domain of the image. Such processing is done directly modifying the value of pixels, so larger images, with greater number of pixels, require a higher computational cost.

Texture analysis can be applied in different fields such as robotics, defence, geosciences, and medicine. In the area of medical imaging, texture analysis has been widely used to describe internal structures of human tissues, organs or pathological changes [9]. These could reveal, for example, if a tumour is malignant or benign.

Typically, texture classification systems follow two steps: (1) a feature extraction step applied to the image under study and (2) a classification step, where a texture class membership is assigned to it according to texture features. It is unlikely that the images to be classified were captured under the same illumination and viewpoint conditions as those used in the training step for classification. When texture analysis is performed, it is important to obtain texture features invariant to transformations of brightness and contrast of the input image. This procedure ensures a better generalization in recognition of textures. The ranklet transform of an image, discussed in the next section, achieves this goal.

## 2.2 Ranklet Transform

Transforms relate a function in one domain to another function in a second domain. A two dimensional digital image can be expressed as a function $f(x,y)$, where $x$ and $y$ are spatial coordinates. The value at every point $(x,y)$ represents the image intensity. Therefore, applying a transform to an image would convert it from its spatial domain, to a different domain such as frequency domain or Hough space. Viewing an image in another domain enables the identification of features that may not be easily detected in the original domain [10]. The ranklet transform of an image involves three phases: $(1)$ multi-resolution, $(2)$ orientation-selective, and $(3)$ non-parametric analysis [11]. It considers the relative rank of the pixels within a local region, instead of their grayscale values (Figure 2.1), making it an operator invariant to linear and non-linear monotonic intensity transformations to the original image [12]. The ranklet transform of a single image is done in the spatial domain, and it produces $n_R = n_r \times n_o$ ranklet images, where $n_r$ is the number of resolutions and $n_o$ is the number of orientations in which the analysis is conducted. Multi-resolution analysis implies defining a

set of resolution values, usually powers of two, that correspond to the linear sizes of ranklet masks. Orientation-selective analysis is done by dividing half of the pixels in each squared window into a *treatment* subset T and a *control* subset C.

$p$

| | | | |
|---|---|---|---|
| 12 | 81 | 98 | 125 |
| 25 | 243 | 196 | 114 |
| 210 | 9 | 203 | 165 |
| 178 | 112 | 48 | 181 |

$\pi(p)$

| | | | |
|---|---|---|---|
| 2 | 5 | 6 | 9 |
| 3 | 16 | 13 | 8 |
| 15 | 1 | 14 | 10 |
| 11 | 7 | 4 | 12 |

**Figure 2.1.** *Left:* Pixel intensity values in a region covered by a mask of resolution $r = 4$. *Right:* Pixel rank values, $\pi(p)$ of the left image [2].

For each of the horizontal (H) , vertical (V) and diagonal (D) orientations, T and C are defined differently, as in Figure 2.2. Finally, the non-parametric analysis is performed in the area covered by the ranklet mask to obtain the ranklet coefficients, computed as follows [13]:

$$R_j = \frac{\sum_{p \in T_j} \pi(p) - \frac{N}{4}(\frac{N}{2}+1)}{\frac{N^2}{4}} \tag{2.1}$$

where $j = H, V, D$ is the orientation, $\pi(p)$ represents the rank of the pixel in the subset $T_j$ and $N = r^2$ is the number of pixels in the neighbourhood. The obtained coefficients are in the range $[0, 1]$. $R_j$ is close to 1 the more pixels in the treatment group $T_j$ of the window neighbourhood have higher gray-scale value than the pixels in the control group $C_j$. Conversely, $R_j$ is close to 0 the more pixels in $C_j$ have higher gray-scale value than the pixels in $T_j$. If $R_V, R_H$ and $R_D$ are close to 0.5 , it means that there is no vertical, horizontal and diagonal global gray-scale value variation, respectively. At a given location of one of these ranklet images, the ranklet coefficient can be thought as a measure of the regularity of its neighborhood at that specific resolution and orientation, namely, a measure of the texture; the higher its absolute value, the more variable its neighborhood [12].

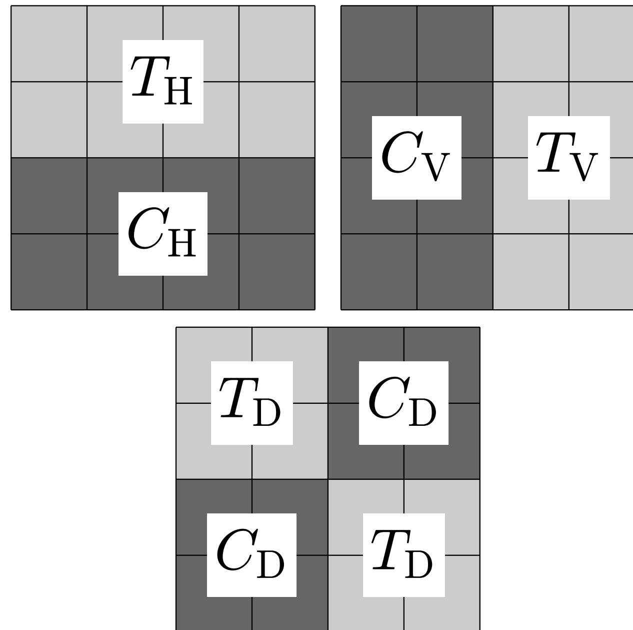



**Figure 2.2.** Three orientations for a $r = 4$ ranklet mask : horizontal (H), vertical (V) and diagonal (D), with their corresponding treatment (T) and control (C) subsets [1].

Since the computation of mutual information is defined for discrete variables, the ranklet coefficients are quantized [14]:

$$R'_j = \lfloor q \cdot R_j + 1 \rfloor \tag{2.2}$$

where $q$ is the desired number of quantization levels. This changes the ranklet coefficient range from $[0,1]$ to $[1,q]$. While the ranklet tranform, and the entire AMI algorithm can be adapted to different types of images, in this work we apply it to grayscale images. For grayscale images, the pixel value is a single number that represents the brightness of the pixel. The most common pixel format is the byte image, where this number is stored as an 8-bit integer giving a range of possible values from 0 to 255. Typically zero is taken to be black, and 255 is taken to be white. Values in between make up the different shades of gray. Therefore, during the quantization step, $q = 256$.

## 2.3 Auto-mutual Information

In information theory, entropy of a random variable is a function which attempts to characterize the “unpredictability” or amount of information of a random variable [15]. It

was first introduced by Claude Shannon in his paper from 1948 "A Mathematical Theory of Communication", that considered a data communication system composed of source data, a communication channel and a receiver. In this original work, entropy represented the mathematical limit on how well data from the source can be losslessly compressed onto a perfectly noiseless channel. Currently, it is most widely used in the area of machine learning, where entropy is a measure of the randomness in the information being processed. The higher the entropy, the harder it is to draw conclusions from that information. But entropy is not a vague concept. In particular, if a random variable $X$ can have values in a set $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, and is defined by the probability distribution $P(X)$, then the entropy of the random variable is:

$$H(X) = -\sum_{x \in \mathcal{X}} P(x) \log P(x) \tag{2.3}$$

The log in the above equation is most commonly taken to be to the base 2, which expresses the entropy in bits.

If several random elements are defined on the same probability space, we can consider joint entropy $H(X,Y)$ to be a measure of the uncertainty associated with a set of variables (Equation 2.4), and conditional entropy, $H(X|Y)$, to be the average of the entropies of the conditional distributions (indicated by $X|Y$) (2.5) [16].

$$H(X,Y) = -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x,y) \log p(x,y) \tag{2.4}$$

$$H(X|Y) = \sum_{x \in X} p(x) H(Y|X = x) \tag{2.5}$$

Closely related to entropy, is mutual information. This quantity measures a relationship between two random variables that are sampled simultaneously. In particular, it measures how much information is communicated, on average, in one random variable about another. Intuitively, it answers the question "how much does one random variable tell me about another?" [15]. The formal definition of mutual information of two random variables $X$ and $Y$, whose joint distribution is defined by $P(X,Y)$, is given by:

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} P(x,y) \log \frac{P(x,y)}{P(x)P(y)} \tag{2.6}$$

where $P(X)$ and $P(Y)$ are the marginal probabilities of $X$ and $Y$ respectively. Equation 2.6

can be expressed in terms of entropy of $X$, $Y$, and their conditional entropies as [17] :

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X) = H(X) + H(Y) - H(X,Y) \tag{2.7}$$

There is no upper bound for $I(X;Y)$, so for easier interpretation and comparisons, a normalized version of $I(X,Y)$ that ranges from 0 to 1 is desirable. Several normalizations are possible, but in this work we define normalized mutual information as [1]:

$$\widetilde{I}(X,Y) = \frac{I(X,Y)}{\sqrt{H(X)H(Y)}} \tag{2.8}$$

The Auto-Mutual Information (AMI) function, proposed in [1] measures the degree of similarity between successive displacements on an input image, which can be performed in the horizontal and vertical directions. The horizontal AMI of an input image $f(x,y)$ of size $M \times N$ is defined as:

$$A_x = \sum_{i=1}^{M-1} \widetilde{I}(f(k_l,y);f(k_r,y)), \quad \forall y : 1 \leqslant y \leqslant N \tag{2.9}$$

where $k_l = 1,2,\ldots,M-i$ and $k_r = i+1, i+2, \ldots M$ denote the left and right displacements respectively. Similarly, the AMI in the vertical direction is defined as:

$$A_y = \sum_{i=1}^{N-1} \widetilde{I}(f(x,k_u);f(x,k_d)), \quad \forall x : 1 \leqslant x \leqslant M \tag{2.10}$$

where $k_u = 1,2,\ldots,N-i$ and $k_d = i+1, i+2, \ldots, N$ denote the up and down displacements, respectively. Figure 2.3, taken from [1], shows the image displacements for computing the AMI. Let us consider the horizontal displacements. At every iteration, the left displacement gets rid of the first column of the image, and the right displacement gets rid of the last column, obtaining two sub-images. For an image with $M$ columns, there will be $M-1$ horizontal displacements, and $2 \times (M-1)$ sub images. For each sub-image pair, the normalized MI in Equation 2.8 is computed. The same process happens for vertical AMI, with the displacements being upwards and downwards.

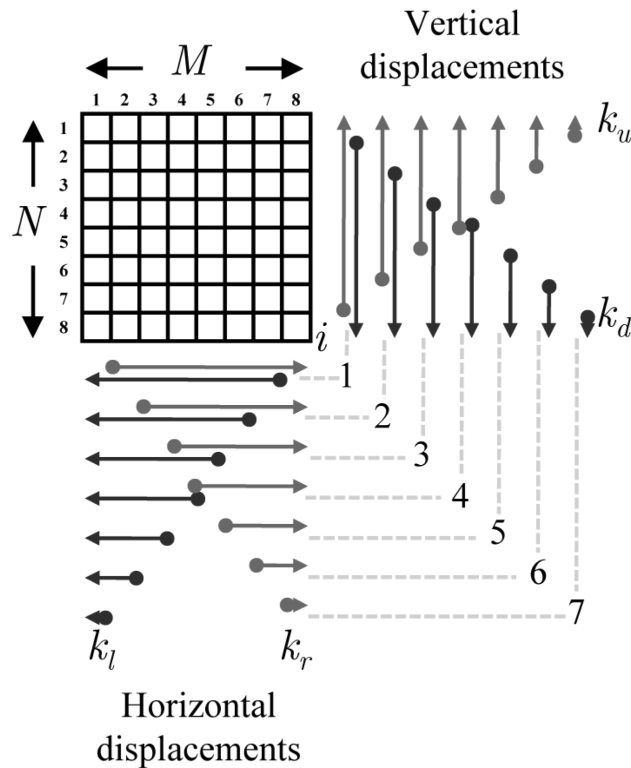



**Figure 2.3.** Image displacements for computing the directional auto-mutual information. Dark gray and light gray lines mark the image displacements left/right (horizontal) and up/down (vertical), respectively. For an image of size $8 \times 8$, seven displacements are performed in the vertical and horizontal directions [1].

## 2.4 Parallel Architectures

Parallel computing refers to the process of breaking down large problems into smaller, independent parts that can be executed simultaneously by multiple processing elements, the results of which are combined upon completion as part of an overall algorithm. The processing elements can be diverse, and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the previous components. In this chapter, we present the parallel computer classification, the theoretical limits of speedup using parallelism, parallel programming models, and explore some of the parallel computing APIs available.

### 2.4.1 Flynn's taxonomy

In 1966, Michael Flynn introduced a taxonomy of computer architectures to classify machines based on how many data items they can process concurrently and how many different instructions they can execute at the same time. The four classifications defined by Flynn are the following:

- Single Instruction, Single Data (SISD) : A machine that executes one instruction at a time, operating on a single data item.

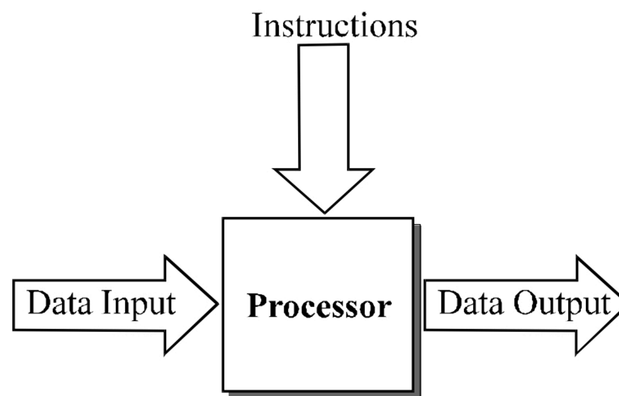



**Figure 2.4.** Single Instruction, Single Data [3].

- Single Instruction, Multiple Data (SIMD) : Each instruction is applied on a collection of items.

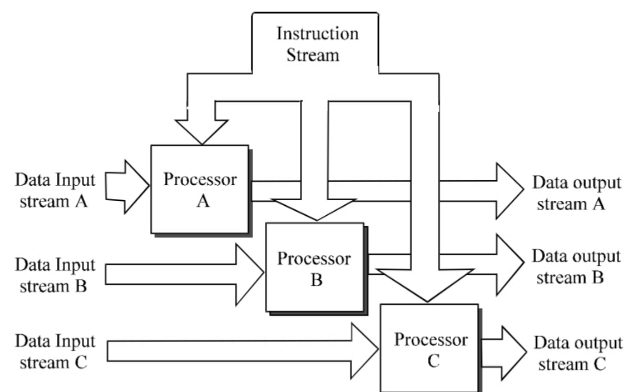



**Figure 2.5.** Single Instruction, Multiple Data [3].

- Multiple Instructions, Single Data (MISD) : Multiple instructions operate on one data stream. This is an uncommon architecture, but is used when fault tolerance is required in a system.

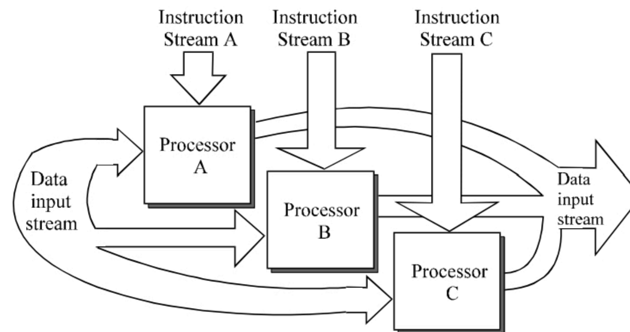



**Figure 2.6.** Multiple Instruction, Single Data [3].

- Multiple Instructions, Multiple Data (MIMD) : Multiple autonomous processors simultaneously execute different instructions on different data. Multicore machines follow this paradigm. For example, Intel Xeon Phi processors have more than 60 processing cores that can execute different instructions on different data.

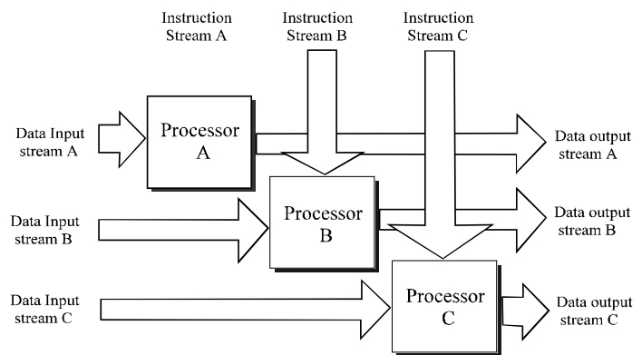



**Figure 2.7.** Multiple Instruction, Multiple Data [3].

## 2.5 Parallel Programming Models

Just like there are several different classes of parallel hardware, which we overviewed in Section 2.4.1, there are different models of parallel programming. Parallel programming models provide an abstract view of how processes can be run on an underlying memory and hardware architecture. These models do not depend specifically on a particular architecture,

so they could be implemented on any type of hardware. Although, specific hardware can make the job easier at times. Below, we give a brief overview of the parallel programming models.

### 2.5.1 Shared Memory Model

In this programming model, the shared memory can be accessed by multiple processes, which can read and write to it asynchronously. In such a way, processes can communicate with each other. An advantage of shared memory model is that data can be accessed equally by all processes, so it is not necessary to explicitly specify the communication of data between tasks. Various mechanisms such as locks and semaphores control the access to shared memory to avoid race conditions and deadlocks. Another advantage is that, on the same machine, memory communication is faster on shared memory model than message passing model. However, T. Ngo and L. Snyder [18] conducted experiments on shared-memory machines that indicated that programs written in a non-shared memory model generally offer better data locality.

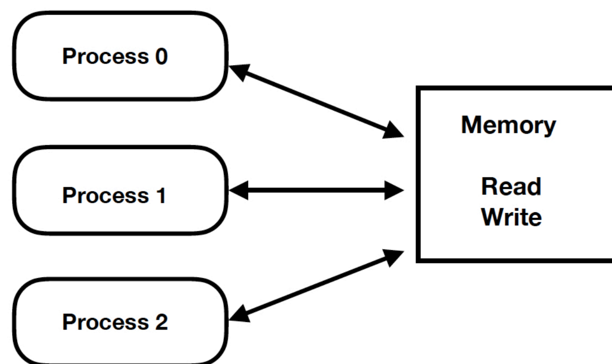



**Figure 2.8.** Shared memory model.

### 2.5.2 Threads Model

Multithreading programming model allows multiple threads to exist within the context of one process, sharing the process's resources but being able to execute independently with their own local data. In the threads model of parallel programming, a single, compute-intensive process can have multiple lighter weight concurrent execution paths. Shared-memory approach to parallel programming must provide a means for starting up threads,

assigning work to them, and coordinating their accesses to shared data, including ensuring that certain operations are performed only one thread at a time [19]. Thread implementations include POSIX threads, OpenMP and CUDA threads for GPUs.

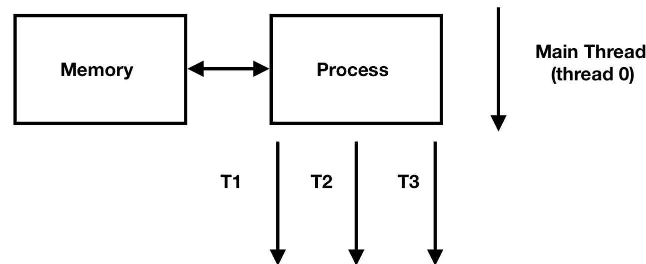



**Figure 2.9.** Threads model.

### 2.5.3 Distributed/ Message Passing Model

Message passing model (Figure 2.10)is closely associated with the distributed memory architecture of parallel computers. A distributed memory computer is effectively a collection of separate computers, each called a node, connected by some network cables. Tasks use their own local memory for computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines [20]. The only way to communicate information between nodes is to send data over the network using a “message". It is not possible for one node to directly read or write to the memory of another node, i.e, there is no sort of shared memory. The most widely used implementation of message-passing is the Message Passing Interface (MPI).

### 2.5.4 Hybrid Model

Hybrid programming model (Figure 2.11) combines several parallel programming models for the purpose of using both shared memory and distributed memory. Combining message passing model with threads model allows for threads to perform computationally intensive kernels using local, on-node data, while processes in different nodes communicate over the network.

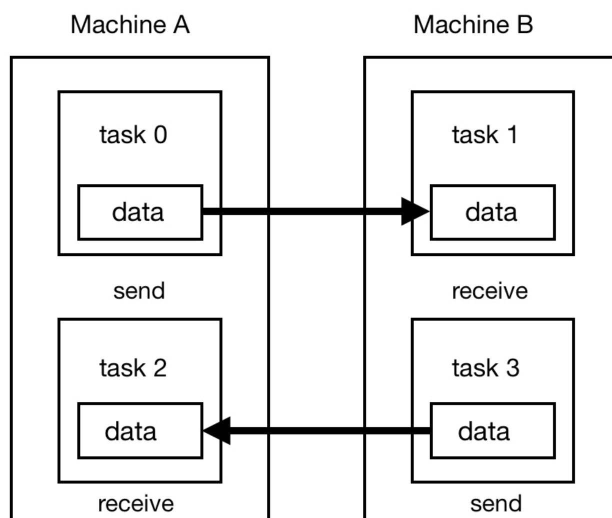



**Figure 2.10.** Message passing model.

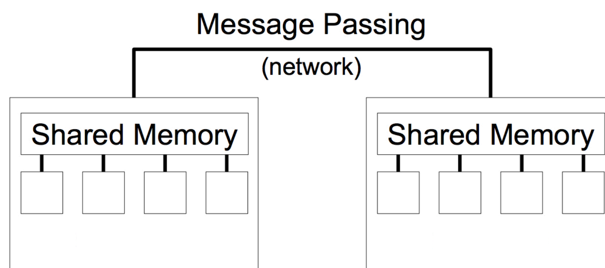



**Figure 2.11.** Hybrid model.

## 2.6 OpenMP

The OpenMP (Open Multi-Processing) API supports multi-platform shared memory parallel programming in C/C++ and Fortran. It was jointly designed by a group of major hardware and software vendors in 1997. Unlike other programming paradigms, OpenMP permits an incremental approach to parallelizing code, meaning we can parallelize portions of the program in successive steps. OpenMP hides the low-level details of creating independent threads and how to assign work to them. The programmer's job then, is to describe the parallelization with high-level constructs, simplifying her work . The OpenMP API comprises a set of compiler directives, runtime library routines, and environment variables to specify shared memory parallelism that allows the user to [21]:

- Create teams of threads for parallel execution.

- Specify how to share work among the members of a team.
- Declare both shared and private variables.
- Synchronize threads and enable them to perform certain operations without interference of other threads.

### 2.6.1 The Fork-Join Model

The fork-join model is a way of setting up and executing parallel programs. In OpenMP, the programmer first needs to specify the parallel region by inserting a *parallel* directive immediately before the code that is to be executed in parallel. Just like in a sequential program, it starts as a single thread of execution. The thread that executes this code is referred to as the *initial* or *master thread*. When the OpenMP parallel construct is encountered by the master thread, it creates a team of threads, called fork, and collaborates with other threads in the fork to execute code dynamically enclosed by the construct [21]. At the end of the parallel region, these threads "join", or merge, back into a single master thread, which continues the program execution sequentially. Parallel sections may themselves create more parallel sections, creating nested forks. Figure 2.12, taken from [4], depicts the fork-join model.

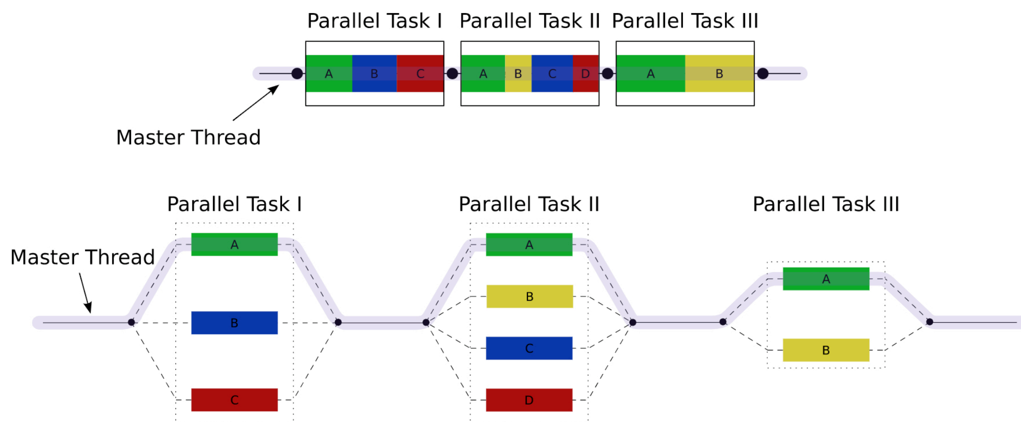



**Figure 2.12.** Fork-join model [4].

### 2.6.2 Thread Scheduling

Controlling how loop iterations are distributed over threads can have a major impact in the performance of the program. This can be done with the schedule clause : `schedule (type,`

`[chunk_size])`. The chunk_size parameter is optional, and its interpretation depends on the chosen schedule. The scheduling types are as follows [21]:

- Static: OpenMP divides the iterations into chunks of size *chunk_size*, distributing the chunks to threads in a round-robin fashion. If no *chunk_size* is specified, the iterations are divided equally among threads, and each thread gets at most one chunk.
- Dynamic: each thread executes a "chunk-sized" chunk of iterations and, when it has finished, requests another chunk until there are no more chunks to work on. *Chunk_size* is 1 if not specified.
- Guided: like in dynamic scheduling, iterations are assigned to threads as the threads request them. However, its difference with dynamic scheduling lies in the size of the chunks. The size of a chunk is proportional to the number of unassigned iterations divided by the number of threads. This means that the size of the chunks decrease. The minimum size of a chunk is set by *chunk_size*. *Chunk_size* is 1 if not specified.
- Runtime: the decision regarding which type of scheduling to use is made at runtime. The schedule and optional chunk size are set through the `OMP_SCHEDULE` environment variable.
- Auto: delegates the scheduling type decision to the compiler and/or runtime system [22].

## 2.7 CUDA

While OpenMP provides a simple way to execute shared memory parallelism, more efficient and lower-level parallel code is possible. However, this often involves re-designing the algorithm. This is achievable using Graphics Processing Units (GPUs). While there are GPUs from several companies, such as AMD, Nvidia's GPUs along with their proposed parallel computing platform and programming model, CUDA (Compute Unified Device Architecture), is the dominating choice when it comes to GPU computing.

In the CUDA programming model the CPU handles the serial code execution. When you come to a computationally intense section of code the CPU hands it over to the GPU to make use of the huge computational power it has. The CPU is called the "host", and it is where the GPU or "device" code is called from. Functions that run in the device are knows as *kernels*. The CPU and GPU have separate memory spaces, meaning you cannot access CPU parameters

in the GPU code and vice versa. Thefore processing flow of a CUDA program involves copying data from host to device memory, invoking kernel on device, copying data back from GPU to CPU memory, and releasing GPU memory and resetting it.

The CUDA programming model has three key abstractions: a hierarchy of thread groups, shared memories, and barrier synchronizations. We will dive into these concepts next.

### 2.7.1 Thread Hierarchy

As we mentioned before, a CUDA kernel is a function that runs on the GPU. Kernels are executed by threads. A thread is an abstract entity that represents the execution of the kernel (Figure 2.13). Each thread has a thread ID that is used to compute memory addresses and make control decisions.

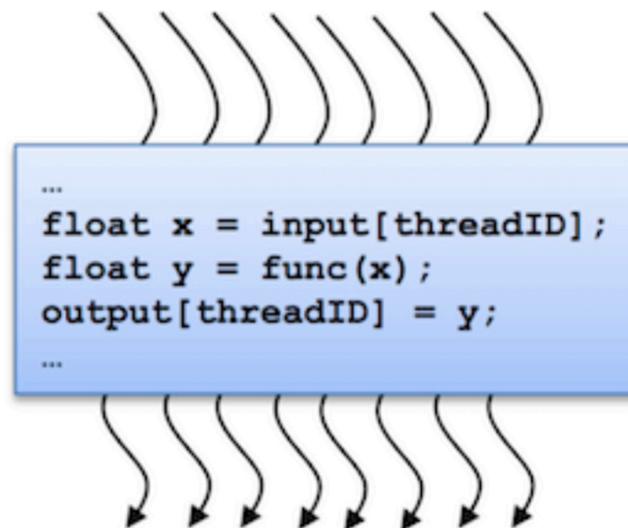



**Figure 2.13.** Threads [5].

Threads are grouped into thread blocks. Threads of the same block have access to a shared memory and their execution can be synchronized to coordinate memory accesses. Thread blocks must execute independently, i.e, the computations carried out in one thread block should not depend on the computations outside of that block. This allows thread blocks to be scheduled across any number of cores, making the code scalable with the hardware being used. Furthermore, thread blocks are arranged into grids of up to three dimensions.

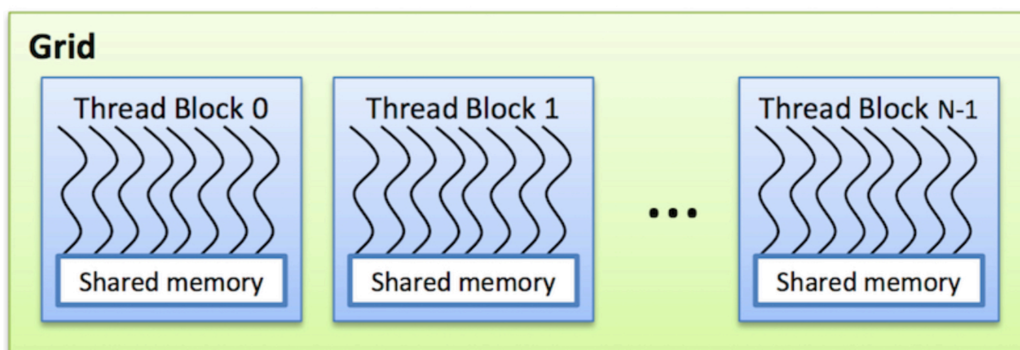



**Figure 2.14.** Grid [5].

### 2.7.2 CUDA Memory Model

CUDA enabled GPUs provide different types of memory with different properties such as access latency, address space, scope and lifetime. For each different memory type, the tradeoffs must be considered when designing the algorithm for CUDA kernel. Incorrectly using available memory in an application can obstruct the maximum performance that can be achieved. The different memories , in order from fastest to slowest access, are the following: register file, shared memory, constant memory, texture memory, local memory and global memory. Intuitively, we would think that always using the fastest memories, such as register files and shared memory, would give us the best performance. However, characteristics such as the scope and lifetime of memory should be also considered while choosing the best form of memory for the different sections of your application. Memory size may limit the use that can be given to it as well. The key points of each type of memory are described next.

- **Register memory:** Scalar variables that are declared in the scope of a kernel function and are not decorated with any attribute are stored in register memory by default. Registers reside "on-chip", meaning it is inside the main processor of the graphics card. The only other on-chip memory is shared memory. Register memory access is very fast, but the number of registers that are available per block is limited. Arrays that are declared in the kernel function are also stored in register memory but only if access to the array elements are performed using constant indexes. Register variables are private to the thread. Threads in the same block will get private versions of each register variable. Register variables only exists as long as the thread exists [6]. Read/write operations to registers do not need to be synchronized. In most cases, accessing a register consumes zero clock cycles per instruction. However, delays can occur due to read after write dependencies and bank conflicts. The latency of read after write dependencies is roughly 24 clock cycles. In addition to the

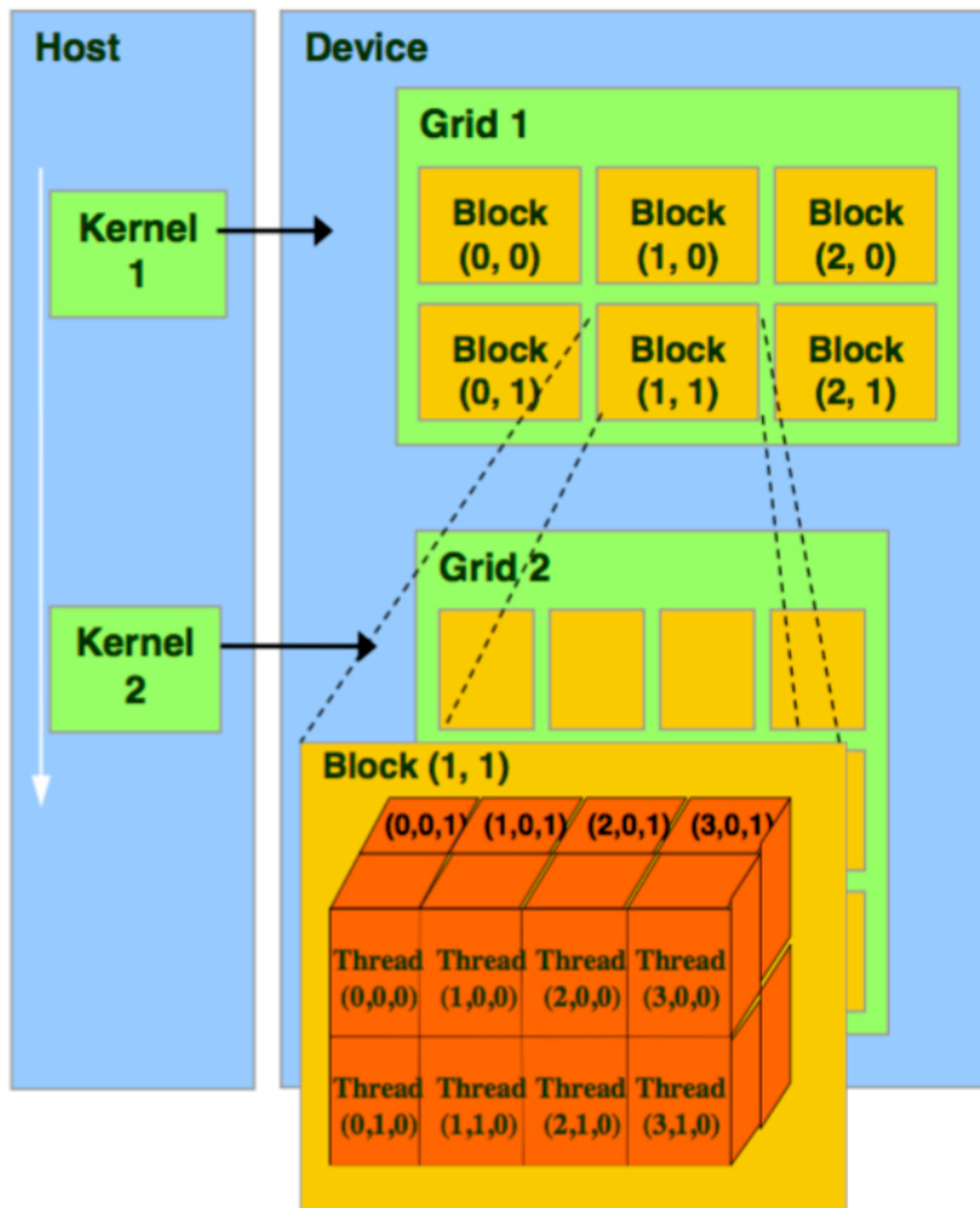



**Figure 2.15.** Layout of a kernel call [5].

read after write latency, register pressure can severely detract from the performance of the application. Register pressure occurs when there are not enough registers available for a given task. When this occurs, the data is "spilled over" using local memory [23].

- **Local memory:** Local memory is not a physical type of memory, but an abstraction of global memory. Any variable that cannot fit into the register space will get spilled-over into local memory by the compiler. Automatic variables that are large structures or arrays are also

typically placed in local memory. Like registers, local memory is private to the thread and it resides off-chip, meaning that it is as expensive to access as global memory. Read/write operations to local memory variables do not need to be synchronized.

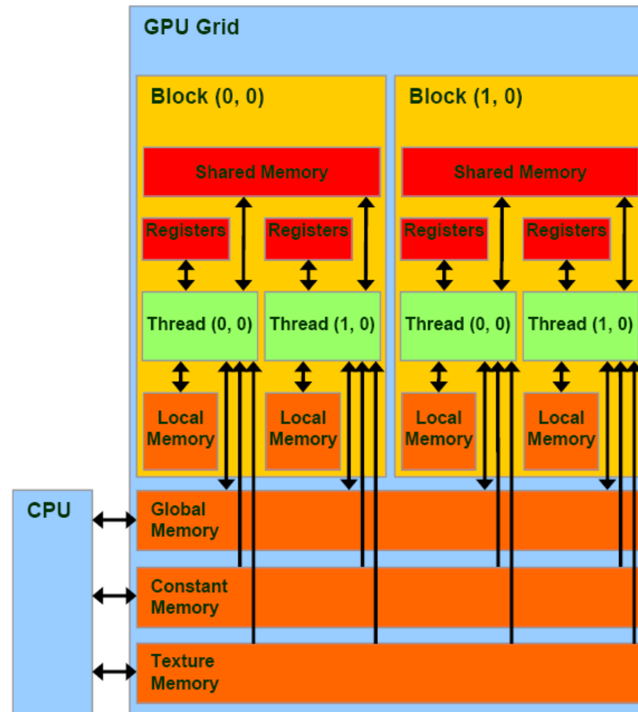



**Figure 2.16.** CUDA memory model [6].

- **Shared memory:** Shared memory is a small amount of on-chip memory, ranging from 16 kB to 48 kB. If used optimally, shared memory speed is almost as fast as register speeds. Shared memory is allocated and used per block, meaning that all threads within the same thread block share the same allocation of shared memory, and each block allocation is not visible to other blocks in the grid. Modifications to shared memory must be synchronized unless you guarantee that each thread will only access memory that will not be read from or written to by other threads in the block. Since access to shared memory is faster than accessing global memory, it is more efficient to copy global memory to shared memory to be used within the kernel but only if the number of accesses to global memory can be reduced within the block [6].

  A common problem arises when memory is shared: with all memory available to all threads, there will be many threads accessing the data simultaneously. To alleviate this potential bottleneck, shared memory is divided into 32 logical banks. Successive sections of memory

are assigned to successive banks. Therefore, any memory load or store of n addresses that spans n distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is n times as high as the bandwidth of a single bank. If multiple addresses of a memory request map to the same memory bank, the accesses are serialized. The hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory requests [24]. When there are no bank conflicts present, shared memory performance is comparable to register memory.

- **Constant memory:** 64 kB of constant memory is used for read-only data that will not change throughout the kernel execution. Constant memory shares device memory along with global memory, but unlike it, is cached on-chip. Constant memory cannot be written from within the kernel, and has the lifetime of the application. Using constant memory instead of global memory can reduce memory bandwidth if all threads in a warp access the same value simultaneously [23].

- **Texture memory:** Like constant memory, it is a read-only memory in the device that is cached on-chip. Texture caches are designed particularly for memory access patterns exhibiting plenty of cache locality [25].

- **Global memory:** Global memory resides in device memory and is accessed via 32-, 64-, or 128-byte memory transactions [26]. The access latency to global memory is very high, about 100 times slower than shared memory, but there is much more global memory than shared memory up, to about 6 GB depending on the specific graphics card. Global memory is both read and write capable and has the lifetime of the application. Since thread execution cannot be synchronized across multiple blocks, access to global memory is synchronized by dividing the problem into several kernels and syncronizing on the host after every kernel invocation. An efficient CUDA program seeks to reduce the amount of global memory accesses, most commonly done by figuring out a different data access pattern that can make use of shared memory.

## 2.8 Summary

In this chapter, we provide the background required to understand the concepts referred to on this thesis. In the beginning, we describe the ideas related to the image analysis aspect, such as texture analysis, and a step-by-step explanation of how the particular algorithm we

are improving works. Next, we describe the concepts related to parallel computing, including parallel architectures, parallel programming models and notions specific to the different APIs used for the implementation component of this work.

# 3
# State of the Art

IN this section, we first give a brief overview of the literature related to texture extraction methods based on ranklets, to make texture features invariant to monotonic intensity changes of the input image. Next, we explore a few instances where high-performance computing (HPC) has been used in the context of image processing.

## 3.1 Related works in image analysis

Ranklets were first introduced in 2002. Smeraldi [11] proposed a family of multi-scale, orientation-selective, non-parametric features modelled on Haar wavelets. These ranklets show wavelet-style directional selectivity and are therefore well suited to characterize extended patterns with complex geometry. In this work, ranklets were used for face detection. Smeraldi showed that ranklets outperform other algorithms such as Haar wavelets, SNoW (Sparse Network of Windows), and linear SVMs.

Massoti *et al.* [12] introduced ranklets in the texture classification domain by proposing a texture classification method invariant to linear and non-linear greyscale transformation. They achieved invariance to 90-degree rotations by averaging, for each resolution, correspondent vertical, horizontal and diagonal texture features. The texture feature vector obtained is assigned a texture class membership using a support vector machine (SVM) classifier. Compared to three recent methods found in literature and evaluating them on the same datasets, the proposed method performs better. Also, invariance to linear and non-linear monotonic grey-scale transformations and 90-degree rotations are evidenced by training the

SVM classifier on texture feature vectors created from the original images, then testing it on texture feature vectors formed from contrast-enhanced, gamma-corrected, histogram-equalized, and 90-rotated images. Figure 3.1 shows the steps in texture classification using ranklets and SVM classifier.

A year later, Massotti *et al.* [27] proposed grey-scale invariant ranklet texture features for false-positive reduction (FPR) in computer-aided detection (CAD) of breast masses. They added a new false positive reduction module to the previous version of their CAD system. This new FPR module was based on texture features computed on ranklet images corresponding to the regions of interest. Free-response receiver operating characteristic (FROC) curve analysis of the two CAD systems demonstrates that the new approach achieves a higher reduction of false-positive marks when compared to the previous one.

Yang *et al.* [13] developed a robust computer aided diagnosis (CAD) system based on texture analysis. First, they extracted grey-scale invariant features from ultrasound images with multi-resolution ranklet transform. On the resulting GLCM-based texture features they applied linear support vector machines for discriminating between malign and benign tumours. Their experiments confirmed that they produced a stable and robust tumour diagnosis method.

Finally, Gómez *et al.* [1] described a novel texture extraction method to classify breast lesions, combining the ranklet transform and mutual information. Since the breast ultrasound (BUS) images were obtained from different scanners, they varied in intensity. For this purpose, the ranklet transform was applied. Next, the AMI based texture features were extracted in the horizontal and vertical directions from each ranklet image. Finally, a support vector machine (SVM) was used to classify breast lesions into benign and malignant classes. The AMI method performed better than nine other methods it was compared to, validating that it is suitable for breast cancer classification.

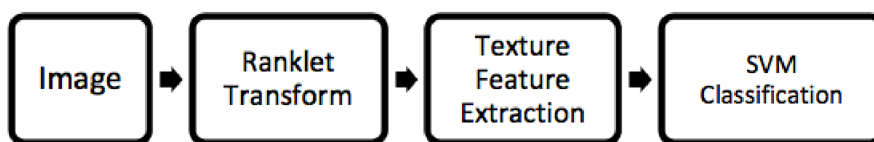



**Figure 3.1.** Sequence of steps in texture classification using ranklets and SVM classifer.

## 3.2 Related works in high performance computing

### 3.2.1 Multicore processors

A multicore CPU is a single computing component with two or more independent actual central processing units, called "cores". Pthreads, OpenMP (Open Multi-Processing), TBB (Threading Building Blocks), and Cilk are application programming interfaces (API) to use the capacity of a multicore CPU efficiently [28].

Kegel *et al.* [29] parallelized a numerical algorithm for 3D image reconstruction obtained by Positron Emission Tomography (PET). They compared two parallel programming approaches for multicore systems, OpenMP and TBB, concluding that OpenMP can parallelize the algorithm with very little program redesign, while with TBB it needs to be re-written. Also, the best runtime results were obtained when using OpenMP.

Saxena *et al.* [30] designed some parallel image processing algorithms including segmentation, noise reduction, features calculation, and histogram equalization using medical images obtained with CT, PET, and MRI. These were implemented in OpenMP, and it was seen that the parallel implementation was 2.5x faster than the sequential implementation.

### 3.2.2 Graphics processing unit

A graphics processing unit (GPU) is a single instruction and multiple data stream (SIMD) architecture where the same instruction is performed on all data elements in parallel [28]. At the same time, the pixels of an image can be considered as separate data elements. So, GPU is a suitable architecture to process data elements of an image in parallel [31]. Over the past few years, the performance of GPU has been improving at a much faster rate than the performance of CPUs. In 2007, NVIDIA's most advanced GPU provided six times the peak performance of Intel's most advanced CPU [32].

Kalaiselvi *et al.* [33] proposed a per-pixel threading (PPT) method for processing a slice and per-slice threading (PST) for an MRI volume that can be implemented in a GPU using CUDA.

Li *et al.* [34] proposed an effective noise reduction method for brain MRI's, where they improved the collateral filter algorithm with parallel computing in a GPU using CUDA.

Alsmirat *et al.* [35] accelerated the execution times of a type of segmentation algorithms based on Fuzzy C-means. Using GPUs, they achieve a performance improvement of 8.9 times, maintaining the same performance accuracy.

Elahi *et al.* [36] improved the computational implementations of Confocal Microwave Imaging algorithms, which are used for the early detection of breast cancer, using GPUs. These algorithms are easy to implement using general-purpose computing platforms. However, a large number of radar signals, high sampling rate and reconstruction algorithms slow down the image formation process. With the GPU implementation, the computational time was reduced by a factor of 250 compared to sequential implementation and a factor of 110 compared to a parallel CPU implementation.

Ramkumar *et al.* [7] improved on the efficiency of the KAZE image feature detection and description algorithm with both a CUDA implementation and a parallel CPU implementation. They achieved a tenfold speedup using a single GPU in comparison to the implementatio for a 16-threaded CPU.

In his master's thesis, Ansaloni [37] implements a multicore version of the ranklet transform using OpenMP and subsequently implements a GPU version using CUDA. In his implementation, the CPU version is faster for images of size up to 4096 pixels, while the GPU version is faster for larger images.

## 3.3 Summary

From our revision of the state of the art, it can be observed that the several image classification schemes based on the ranklet transform have outperformed previous ones. While these methods have been successfully applied for image classification, they focused solely on the accuracy of their methods, leaving behind the efficiency. The works that focused on medical image classification used ultrasound images. However, using these algorithms on larger medical images, which carry more detailed information due to their high resolutions, may be unviable.

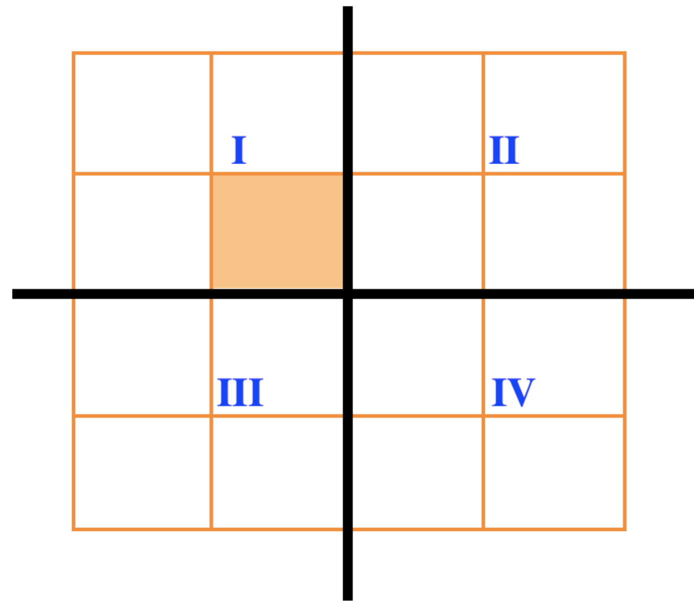
In addition, high performance computing is widely used in image processing tasks, such as image reconstruction and feature detection, since they are so compute intensive. In all cases, the parallel implementation is more efficient than the sequential implementation, reaching up to a 250 factor speedup in some cases. While in [37], there seems to be a parallel ranklet implementation, the code is not provided anywhere. In addition, according to our revision of the literature, the most time intensive section of the AMI has not been parallelized in other works yet.

# 4
# Proposed Approach

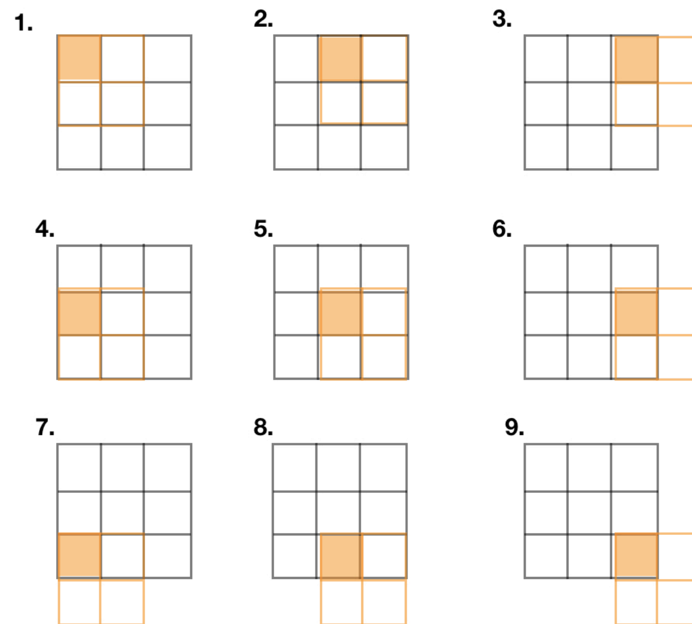
In this chapter, we present the AMI sequential implementation which was done in the C++ language, using a few functions from the computer vision library OpenCV to simplify basic operations like reading images or obtaining a section of the images. While the main idea of the AMI algorithm was discussed in Chapter 2, Sections 2.2 and 2.3, several details which will be covered now were left out, since they are implementation-specific. Subsequently, we describe the parallel design proposed for implementation in CPU using OpenMP and in GPU using CUDA. Lastly, we present a heterogeneous version that uses both approaches.

## 4.1 Sequential Implementation

The AMI algorithm consists of two stages: the ranklet transform, and the auto-mutual information features extraction. As it is commonly the case with transforms in image analysis, we use sliding window operations to obtain the ranklet transform of a given image. For a single window, we calculate the new value for the pixel located at the middle of the window using Equation 2.1, as in Figure 4.1a. Since we preserve the total number of pixels in the image, we do this window operation $N \times M$ times, where $N$ and $M$ are the input image dimensions. We obtain the new pixel values in the ranklet-transformed image one by one. The window slides from top left to bottom right, according to Figure 4.1b.




(a)




(b)

**Figure 4.1.** (a) A window of resolution of $r = 4$. Note that since the window resolution is always taken to be a power of 2 (i.e, is even), we take the middle pixel to be the rightmost pixel at the bottom of the first quadrant. (b) Shows how an $r = 2$ window slides, from top right to bottom left. Every coefficient is computed one at a time.

Notice that positions in the filter may be outside positions in the image, particularly if the center pixel is on the border of the image. To avoid fetching arbitrary values that may be stored in memory for such positions, we use a common technique in image processing called padding. It consists of assuming the border pixels in the image are surrounded by additional rows and columns of a certain value. In our case, we padded the image on all sides with as many rows and columns as the largest window we are considering, with intensity values of 255.

Altogether, the breakdown of tasks required for the ranklet transform are as follows:

- Read an image into the OpenCV Mat format, which was chosen to simplify a few of the tasks in the later steps.
- Iterate through all the window sizes to do the steps below (loop A).
- Pad the image according to the largest window size we are considering.
- Iterate through all the pixels in the image, at each position to calculate its new ranklet value (loop B). Each iteration corresponds to a single window slide.
- Calculate the ranklet value for current pixel by sorting all pixels in the current window, calculating the rank sum of pixels in the treatment group, and applying Equation 2.1 (loop C). We calculate the horizontal, vertical, and diagonal values of the pixel within the same function call using Mann-Whitney statistics (given by Equation 2.1), as per the description of orientation-selectivity in the original paper on ranklet [11].

We have labelled the points that required iterations as loops A, B, and C and will be referring to them in later sections. Loop C does not refer to one particular loop, but to any iterations required for the last point. For example, the quick sort implementation has several associated loops, as does the computation of $\pi(p)$ in the treatment group.

In the computation of ranklet transform, the only instances in which OpenCV functions were used were to read our input image, to pad and flatten out the images, both before ranklet function, and in creating the three extra Mat variables to store the H, V, and D transformed images. As an extra step, our implementation for displaying the transformed images in a separate window also used OpenCV.

The second stage in AMI is the computation of mutual information texture features. Let us remind ourselves that the ultimate goal of the algorithm is to obtain a feature vector. The vector will be of size $\#windows \times \#orientations \times 2$, the last 2 corresponding to the fact that

we compute MI features with horizontal 2.9 and vertical 2.10 directions. Figure 4.2, illustrates how many features are obtained for the case where we use four different window sizes.

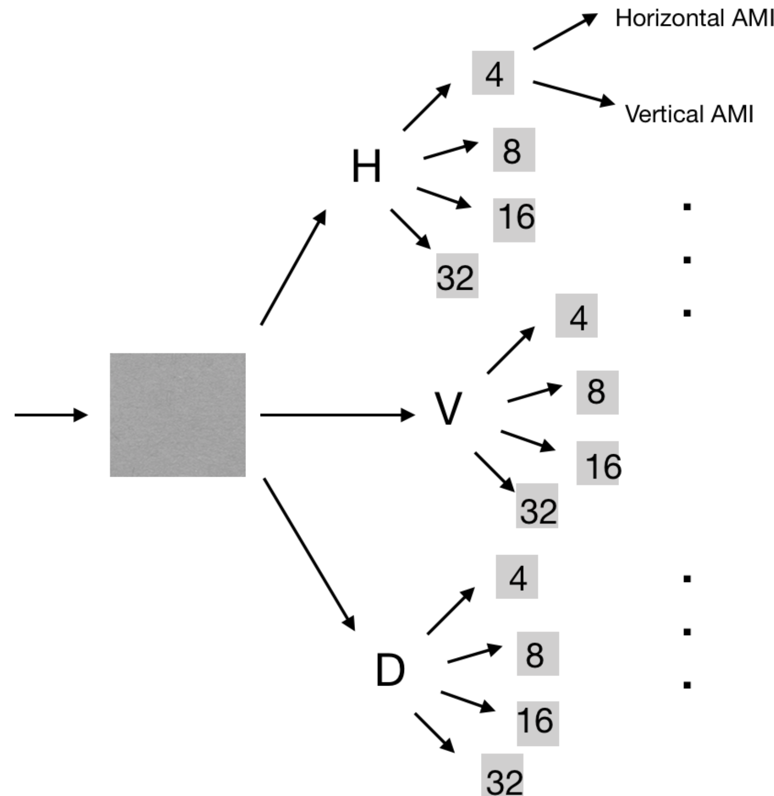



**Figure 4.2.** Each AMI feature corresponds to an element in the AMI feature vector. For every orientation and window size, two AMI features get computed. In this case, three orientations and four window sizes will produce a vector of size 24.

Once we have obtained the ranklet transform of an image, we have to quantize it according to Equation 2.2 to obtain an image in the greyscale intensity range. The mutual information of two random variables is given by Equation 2.6. In the case where the inputs for auto-mutual computation are two images, the random variables considered will be pixel intensity values. To compute $P(x, y)$, we build a joint histogram table. Given two images of the same size, the first dimension of the table logs the intensities for the first image and the second dimension logs the intensities for the second image. At location $(i, j)$, the joint histogram stores how many pixels we have encountered that have intensity $i$ in the first image and intensity $j$ in the second image. If the input images are quantized, $i, j \in [0, 255]$. The marginal probabilities can be then easily obtained from the joint histogram by summing all the elements in a single column to calculate $P(x)$, and all the elements in a row to calculate $P(y)$. The joint histogram

will store as many elements as pixel values there are in the input image sizes.

The final feature is the normalized mutual information, given by Equation 2.8. Once we have the marginal probabilities, this is easy to calculate using the formula for entropy given in Equation 2.3. All of the above is only to compute the mutual information between two images. The auto-mutual information will repeat this as many times as there are rows in the image to obtain a single horizontal feature, and as many times as there are columns to obtain a single vertical feature. This is a key concept, because notice that for an image of size $1024 \times 1024$, there will be 1024 joint histograms created to to obtain a horizontal feature, and then again 1024 to obtain a vertical feature. A single joint histogram is an array of doubles of size $256 \times 256$. We will come to this idea in the later sections that discuss the parallel implementation.

Altogether, the breakdown of tasks required for the obtention of the texture feature vector are as follows:

- Quantize the input image of size $N \times M$ . This image is the result of the ranklet transform in stage 1.
- Iterate through the $N - 1$ rows of the image to obtain two sub-images each time. One sub-image will discard the top row and the other one will discard bottom row (loop D).
- For the current pair of sub-images, compute:
  - Joint probability table (loops E).
  - Mutual information (loops F).
  - Entropy (loops G).
  - Normalized mutual information.
- Sum the normalized mutual information for every single displacement, to obtain a single auto-mutual information feature (loop H).
- Transpose the image and repeat all of the steps above to obtain the horizontal AMI feature.

We have labelled the points that required iterations as loop(s) D, E, F, G, and H, and will be referring to them in later sections. As in the ranklet sections, loops E, F, and G refer to all the iterations required to complete that particular step. In loop D, we used the OpenCV built-in function `Mat::colRange(int starcol, int endcol)` to discard sections of a given image. We are creating a new matrix by borrowing data from an existing matrix. An

important thing to note is that sub images obtained in this manner, may be stored in memory with gaps at the end of each row. To ensure we are picking data that actually belongs to the subimage, we use the built-in function `Mat::isContinuous()` that reports whether the matrix is continuous or not. If the matrix is not continuous, we call the `Mat::clone()` function that creates a deep copy of the mat, and calls `Mat::create` to create a new matrix which is guaranteed to be continuous. This is not done initially so as to not make unnecessary memory copies, and to avoid thoughtlessly calling clone(), which is known to be an expensive function. Lastly, we make use of OpenCV on our last point, to transpose the input image using `transpose(Mat input, Mat transpose)`.

## 4.2 Parallel Implementation on CPUs

Running our algorithm in parallel using OpenMP is relatively simple, since there are multiple sections that can be done independently, and whose results can be subsequently combined. The inherent parallelism of our problem can be exploited by running these sections in parallel regions. The following loops were run in parallel for each stage:

- Ranklet transform: loop B.
- MI feature extraction: loops D and H.

While there were multiple iterative operations for both of the stages, discussed in the previous section, we only parallelized a small portion of them. Two main aspects were considered for making this choice: nested parallelism and task granularity. In stage 1, loops A, B and C are triply nested (Figure 4.3a). Creating nested parallel regions adds overhead. If there is enough parallelism at the outer level and the load is balanced, generally it will be more efficient to use all the threads at the outer level of the computation than to create nested regions at the inner levels [38].

**(a)** Stage 1: Ranklet transform. **(b)** Stage 2: MI feature extraction.

**Figure 4.3.** Loop hierarchy for the two stages of AMI.

The sequential execution time of code has to be several times this to make it worthwhile parallelizing. One example of this in the ranklet stage is when we are sorting the pixels of a window in loop C. The window sizes are of resolutions $r = 4$ up to $r = 32$, or 16 to 1024 pixels. For most resolutions in this range, it is not worth sorting in parallel. However, if the algorithm used larger window size, maybe the cost of launching a parallel region would far exceed the overhead. In any case, this can only be found out experimentally.

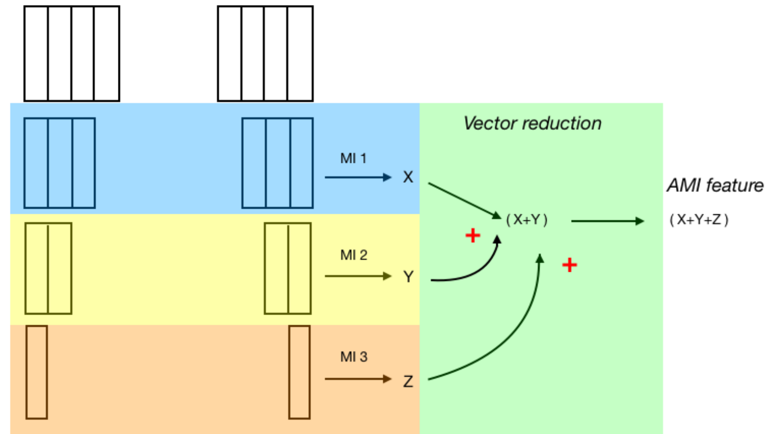



**Figure 4.4.** Parallel regions for the AMI texture feature extraction stage. In the first parallel region, a single thread calculates the MI for every pair of subimages obtained from a displacement. In the second parallel region, a vector sum reduction obtaines the AMI feature.

This situation also links to the idea of task granularity. Granularity is the amount of real work in the parallel task. If granularity is too fine, then performance can suffer from communication overhead. If granularity is too coarse, then performance can suffer from load imbalance. The goal is to determine the right granularity (usually larger is better) for parallel tasks, while avoiding load imbalance and communication overhead, to achieve the best performance [39]. Iterating through all the window sizes, loop A, would assign to each thread high granularity tasks. And, as discussed above, loop C would assign very finely grained tasks to them. Therefore, we chose to only put loop B, calculating every pixel value of the ranklet image, into a critical region. In stage 2, the maximum level of nesting is two (see Figure 4.3b). We chose to parallelize the two outermost loops. Tasks in loop G are finely grained once we have computed the joint probability tables in loop E, as discussed in our previous section. Filling up the joint probability tables in parallel would also be a bold choice of parallelism. However, this would introduce nested parallelism. We avoided this, and parallelized larger tasks: (1) obtaining the normalized mutual information for every pair of subimages, and (2) adding these normalized mutual informations to obtain a single auto-

mutual information feature. The latter was done using the OpenMP reduction clause (see Figure 4.4).

One upside of this implementation using OpenMP is that if the parallel sections of the algorithm are identified from the beginning, very little rewriting of the sequential code is required, sustaining code reusability.

## 4.3 Parallel Design on GPUs

The AMI implementation in the GPU was written in C++ using the CUDA API. Given that a GPU is a massively parallel hardware, we can expect our algorithm to benefit from its use. However, there are a few extra steps that may hinder the overall performance gain. We followed closely on the parallelization scheme used in the multi-CPU version of the AMI. One key difference concerns memory management. The CPU has a large amount of system memory with three levels of cache. Using the shared memory model of OpenMP, all threads have access to store and retrieve variables to memory, although they also have a thread-private memory. So, for this implementation, memory management is not an issue. However, for the GPU implementation, there are several types of memory, discussed in the *CUDA Memory Model* subsection of Chapter 2. For each different memory type, the tradeoffs must be considered when designing a CUDA program. Since the purpose of our work is to be able to apply the AMI algorithm to large images in a reasonable amount of time, it is not a surprise that one of the main challenges is to deal with size of the data itself. The size of the images used range from 20 KB, for the images with 128 pixels in side, to 65 MB for the largest images of $8192 \times 8192$ pixels. We must assure that all images fit in device memory during the allocation of resources. Register and shared memory are the only on-chip memories and would be the location that would provide quickest access to the images. Register space is used for automatic storing of scalar variables done by the compiler, so we cannot use it. Likewise, we cannot use shared memory because it has a size limit of 48 KB in the card we are using. Constant and texture memory are both device memories, like global memory, but they have their own on-chip caching systems. Ideally, we would use them, but we run again into the memory limit of 64 KB for constant memory and of $65 \times 32$ KB for 2D texture memory. This leaves us with having to allocate our input image into global memory, whose size of 6 GB provides more than enough space for our data, but whose access is the slowest.

Every CUDA program has some generic steps that need to be completed before actually carrying out the main operations that the multi-processor nature of the GPU is used for, and

as such, we have followed them in our implementation [40]:

- Declare and allocate host and device memory.
- Initialize host data.
- Transfer data from the host to the device.
- Execute kernels.
- Transfer results from the device to the host.

These steps set up the communication between the host and device. As we will see, calling the kernel itself is often a highly expensive operation. Next, we provide some details of the implementation of the two stages of the AMI algorithm.

### 4.3.1 Ranklets

To compute the three ranklet transforms of an image, a similar parallelization scheme to the OpenMP one was used. In our previous implementation, there would be as many threads as CPU cores available. Now, we can launch as many threads as allowed by the NVIDIA card, which is 1024 threads per block and 3D grids of 65,535 blocks per dimension. The largest images we are manipulating have $8192 \times 8192 = 67,108,864$ pixels. Given the hardware limitations, we can still assign to one thread the task of computing a single pixel value in the transformed image. In fact, since we compute the new pixel values for the H, V and D transforms simultaneously, a single thread will compute three different pixel values. To achieve this, we have to launch as many threads as there are pixels in the image. The number of threads per block is variable, but we have to launch enough blocks to cover the image area. Given this natural analogy between grid and image, we launch a 2D grid with the same dimensions as the image. For example, say we choose to launch a 2D block of size $4 \times 4$. Then, we declare a `dim3` type grid, `grid(img.cols/block.x, img.rows/block.y)`, that will launch enough threads to cover the entire image, where `img.cols` is the number of columns in the input image, `img`, and `img.rows` is the number of rows it has. Note that this expression for grid declaration is the same no matter the size of the 2D block.

As before, we originally read the image into the OpenCV type Mat, however we copy this data into device as unsigned chars, which is 1 byte in size and can hold exactly the number of values in the greyscale range. Once the kernel is launched, we map threads to pixels in row order fashion.

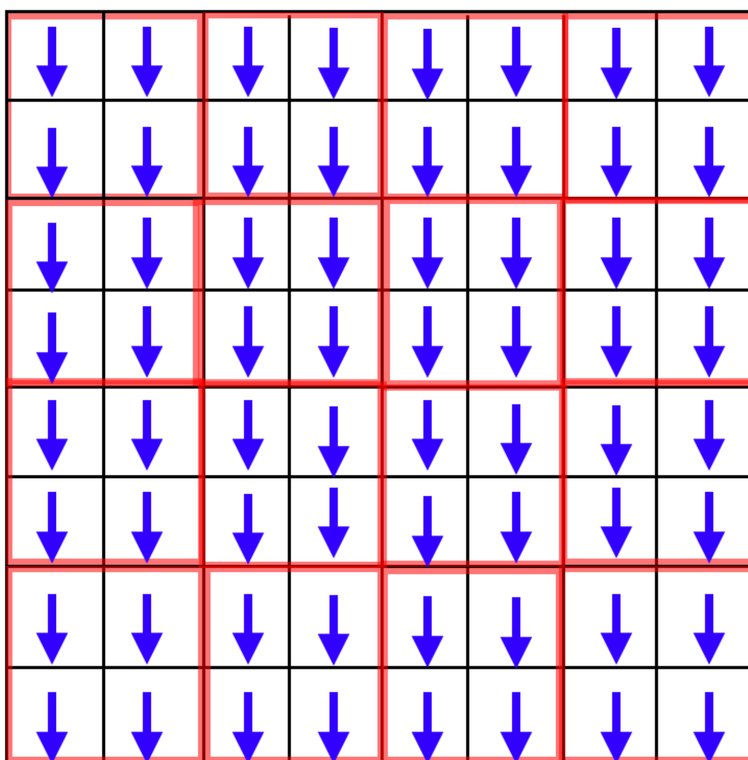


**Figure 4.5.** For a $8 \times 8$ pixel image, we fix block size. As many blocks need to be launched to ensure there is one thread per pixel. In this figure, block dimensions were fixed to be (2,2) and therefore, a grid of 16 blocks will be launched. Threads are represented with the blue arrows, and blocks with the red squares.

Each thread now computes the new pixel value throughout a series of function calls with the __*device*__ qualifier, which means they can be called only from within the device, and not from host. One important aspect that was considered was the concept of memory coalescing. Given that we are storing our image in the slowest memory, global memory, we should try to retrieve image information in an efficient manner. Memory coalescing refers to the idea that threads that run simultaneously access memory locations that are near to each other. In CUDA, threads in a warp execute the same instruction. If all threads in the warp accessed consecutive locations in the DRAM, the request to access these locations could be done at once, achieving memory coalescing. Due to this, we used row-major access pattern in our pixel to thread mapping.

### 4.3.2 MI-based features

After obtaining the three ranklet transforms, we must obtain a feature from each of them. Previously, we brought the transformed images back to host, so to use them again, we need to allocate device space for them and repeat the steps we mentioned every CUDA program requires. It is important to notice that what we are doing is parallelizing the computation of a feature from a single image, and not the obtention of multiple features simultaneously, which we explore in the next section. In the case of the ranklet transform, we followed a similar parallelization scheme to our OpenMP implementation, where loops D and H were run in parallel. As before, the first step is to make sure the images we will use fit in device memory. Previously, we decided to put them in global memory. During the MI stage, at every iteration of loop D, two sub images are considered. One of them discards the top row and the other one discards the bottom row. For each of these pair of sub-images, we compute the joint probability table. Let us remind ourselves that the joint probability table is a $256 \times 256$ table of doubles that will hold the probability of having a value $X$ in image 1 and having a value $Y$ in image 2 at the same position. Let us consider an image with $N$ rows and $M$ columns. Because discarding one row will change $M$ values in the histogram, we compute a new joint histogram for every single subimage generated. This means that to compute a single feature, we create $2 \times N - 1$ joint histogram tables, each of which take up $256 \times 256 \times \mathtt{sizeof(double)} = 524,288$ bytes, or 524 KB. This joint histogram needs to be written to and read from, and will not fit in our 48 KB shared memory. Now, let's remind ourselves that if we parallelize loop D, those $2 \times N - 1$ tables will be existing at the same time, requiring $524 \times 2 \times N - 1$ KB. Clearly, we have no other choice but to use global memory for this particular parallelization scheme.

We launch two kernels in this stage: one to get the normalized mutual information vector (let's call it MI kernel), and another one to sum this vector in the commonly know vector reduction operation (let's call it reduction kernel). For the MI kernel, the number of threads are variable, but we must launch as many blocks as needed to have row-1 threads. In this way, one thread will compute a single element of the vector to be reduced. In the ranklet stage, we used a 2D grid, but here the nature of the problem does not require us to use an extra dimension since we only consider displacements on either horizontal or vertical dimensions at once. Before doing the kernel call, one important extra step is required. Notice that there is a great deal of work assigned to every thread. Each of them does the work in loops E through G. Once in the MI kernel, they are completed through a series of functions calls with the *__device__* qualifier. There are several intermediate variables, like the histogram table, marginal probabilities and joint probabilities, that declare

temporary arrays. Doing so using `malloc()` and `free()` will store such variables in the thread's heap. We need to make sure there is enough heap space for each thread before the kernel call, and if not, allocate it by using `cudaThreadSetLimit(enum cudaLimit limit, size_t value)`. For our largest images, we allocated up to 4 GB of this as `cudaThreadSetLimit(cudaLimitMallocHeapSize, 1024ULL*1024ULL*1024ULL*4ULL)`.

Now that we have done all our memory requirement considerations, we finally launch our MI kernel. Every thread considers two sub images simply by pointing to the appropriate memory location, rather than newly creating two sub-images. For example, thread 0 will consider the image starting at row 1, and the image starting at row 0 up to row $N-1$. Thread 1 will consider one image starting at row 2, and another one starting at row 0 and ending at row $N-2$. This is illustrated in Figure 4.6, and is one considerable difference we made to the OpenMP version, where creating completely new sub images would not bring with it memory problems.

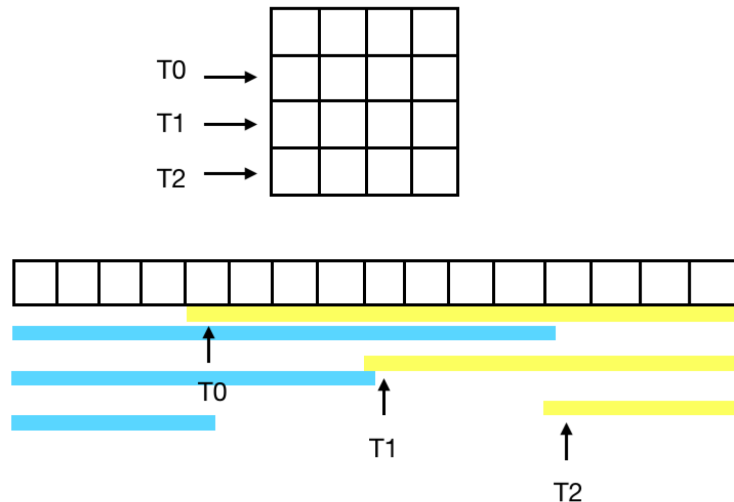



**Figure 4.6.** Sub-images considered in downward displacements (yellow) and upward displacements (blue). No new images are created.

Parts of the code required to implement MI feature extraction were taken from [41]. The functions to compute joint probabilities and mutual information were used as is for the OpenMP implementation. Other code optimization, not necessarily related to the use of CUDA, were done for our GPU implementation, namely, reducing the number of intermediate variables in the code, which would take up more thread heap space and could lead to the inability of launching enough threads to cover all rows in the input image. In the original

code, the joint probability function only gives us a joint histogram table, and to find the marginal and joint probabilities, we have to read from the table again to fill up newly created vectors. Our implementation did both of these in one go, avoiding creating the $P(a)$, $P(b)$ 1D arrays and the $P(a, b)$ 2D array. Since these quantities are computed row-1 times, this optimization provided a huge improvement in memory usage. In fact, without it, our algorithm ran out of memory for images as small as $1024 \times 1024$.

At the end of the MI kernel, we have a vector of normalized mutual information that needs to be summed, where one element corresponds to a pair of sub-images. For this, we launch a vector reduction kernel. This is a well studied problem, which has several implementations. For this work, we chose to do the vector reduction using shared memory. This implementation does per-block reductions, having per-block partial sums that are combined into a final total sum. The shared memory will hold as many partial sums as threads per block are launched. To avoid warp divergence and having idle threads, we make sure that sequential threads are doing the sums. This avoids otherwise using the modulo operator, which is an expensive operation, as well as shared memory bank conflicts. When a thread wants to get a word from shared memory, we go to the shared memory bank. If all threads access different banks, they can access them at the same time. If they all access same data, we can use the broadcast method. If different threads access the same bank but different data, this access will be serialized. The sequential addressing method we used is shown in Figure 4.7.

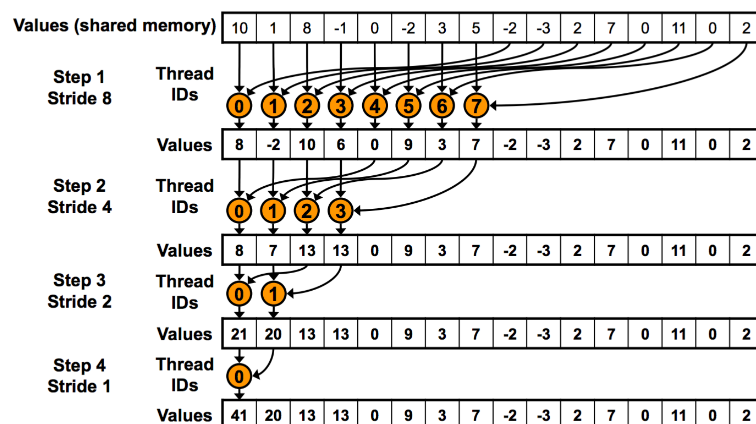



**Figure 4.7.** Sequential addressing method for vector reduction.

## 4.4 Heterogeneous implementation

Heterogeneous computing refers to the use of different types of processing units. In our case, we combine CPUs and GPUs. In our previous scheme for the MI feature extraction stage, we used GPU computing power to obtain a single feature at once. Now, we use a varying number of cards to obtain multiple features at once. With OpenMP, we create a host thread for each available CUDA device. Subsequently, the host thread spawns multiple device threads that will each execute the kernel. This is a context aware implementation, meaning that we will use as many devices as available. If we have a single device available, the features will be computed sequentially. Note that for the ranklet stage we used a single card, since previously we computed the H, V, and D transforms in one go. It would be possible to assign one card to each orientation, but this would mean a lot of duplicate code without much gain from it. Therefore, the ranklet transform is done in a single card and its resulting images brought back to host. In the MI stage, every transformed image is copied to the respective card that will obtain its MI feature. Figure 5.6 shows this setup.

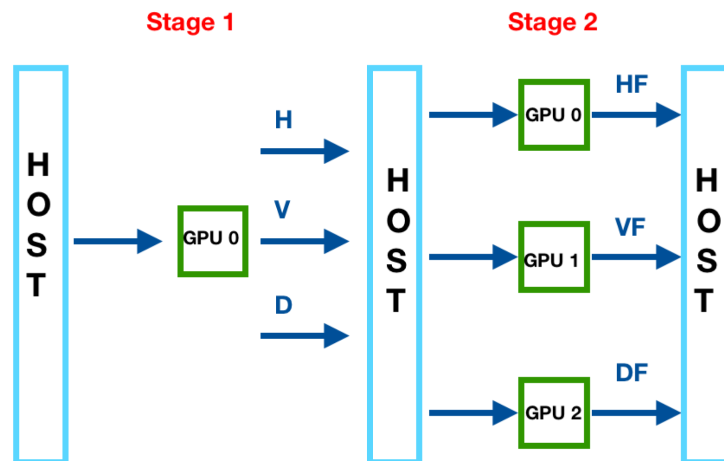



**Figure 4.8.** Heterogeneous approach: stage 1 is computed in a single GPU. The three resulting ranklet transforms are copied back to host. Stage 2 of the algorithm is computed in all the GPUs available. Each GPU computes one of horizontal, vertial and diagonal feature for a single window size. Lastly, the features are copied back to host.

## 4.5 Summary

In this chapter, we presented a detailed approach of the sequential, the two parallel, and the heterogeneous implementations of AMI. We provided the details regarding the design and programming choices made for each case, and the advantages and disadvantages resultant of them. We highlight the parallel regions in the original algorithm, and point out which of them were chosen to be executed simultaneously in the proposed schemes.

# 5

# Results and Discussion

In this chapter, we describe the tests conducted on the three implementations described in Chapter 4, in order to compare their speedups to the sequential version of AMI. We measured the OpenMP performance using one computing node containing two Xeon processors with six cores each. For the CUDA implementation, three different Tesla C2070 NVIDIA graphics cards were used. The dataset was composed of seven sets of fifty squared, 8-bit images each, taken from the Mammographic Image Analysis Society (MIAS) dataset, obtained from their public repository [1]. The original images are of size $1024 \times 1024$, but to test our algorithm on images of different sizes than the ones contained in MIAS, we used MATLAB `imresize()` method. Images of both smaller and larger dimensions than the original ones were obtained using the default nearest-neighbour interpolation method.
To guarantee a greater statistical validity of the measurements, for each image size, the measurements of execution time were done 50 times on 50 different images, using the same window resolution. The images are of size 128, 256, 512, 1024, 2048, 4096 and 8192 pixels per side. Each of the seven sets contained 50 images of the corresponding sizes, for a total of 350 images. The window sizes used for the ranklet transform were of resolution 4, 8, 16, and 32.

[1] https://www.repository.cam.ac.uk/handle/1810/250394

## 5.1 Infrastructure

For the development of this work, a server with the following characteristics was used:

- Operating system: Ubuntu 16.04.6 LTS
- RAM: 24 GB
- **CPU:**

| Model | Intel(R) Xeon(R) CPU X5675 |
|---|---|
| Processor base frequency | 3.07GHz |
| Number of cores | 12 |
| Cores per socket | 6 |
| Cache memory | 12 MB |

- **GPU:**

| Number of cards | 3 |
|---|---|
| Model | Tesla C2070 |
| Number of multiprocessors | 14 |
| Shader clock rate | 1147 MHz |
| Compute capability | 2.0 |
| Global memory | 5428224.00 KB |
| Constant memory | 64.00 KB |
| Shared memory per block | 48.00 KB |
| Registers available per block | 32768 |
| Warp size | 32 |
| Maximum number of threads per block | 1024 |
| Maximum number of threads per multiprocessor | 1536 |
| Maximum number of warps per multiprocessor | 48 |
| Maximum grid size | (65535,65535,65535) |
| Maximum block dimension | (1024,1024,64) |

The programs were written in the C++ language, along with the CUDA version 8.0 and OpenMP version 4.0 APIs.

## 5.2 Timings

In this section, we present the execution times obtained for the three different implementations, along with several graphs for their comparison. As mentioned, for every image size, the execution time was calculated for 50 different images. The reported times are the average of them.

### 5.2.1 OpenMP

Time measurements for OpenMP related code were done using the double precision built-in function `omp_get_wtime()`, that returns elapsed wall clock time in seconds. The execution times for the different image sizes for the C++ sequential and the OpenMP parallel implementations are shown in Table 5.1. A fixed window resolution size of 8 was employed. The percentage standard deviations that arise from the 50 different set of values measured for a particular image size are also reported. The sequential measurements were done experimentally only up to image sizes of 1024 pixels per side, which already took the considerable time of 3 min per image. Notice that, since the measurement was done for 50 images, this experiment took us 2.5 hours. Timing for a single 2048 image did not finish even after an 24 hours of execution. For this reason, the execution time of the rest of the larger image sizes were predicted from the data trend line. The trend line used was not that set by the data points of the previous sequential measurements, but the one set by the OpenMP data points (see Figure 5.1). This was done because the tendency followed by all seven data points was more meaningful than to extend the trend set by only the first four data points. Using the trend set by only four sequential points, in fact, indicated that the sequential implementation for images 4096 and 8192 took less time than the OpenMP implementation, which we know is not a sensible prediction. The best line fit that was followed to obtain the sequential execution times for images of 2048 up to 8192 pixels per size, shown in red in Table 5.1, is given by the quadratic equation $y = 0.003x^2 - 0.8748x + 327.47$ seconds, shown in Figure 5.1. The speedups to the sequential version went from 8.29 up to 15.62 times. Notice that the speedup values do not follow any particular trend, for example, a linear trend, where the speedup would increase by a constant factor proportional to the increase in image size. It is expected that, the larger the image, the greater the speedup. However, for the last two images sizes, there is a decline in speedup. This “plateau” may indicate that the resources are best utilized for the 2048 images. In OpenMP, tasks are distributed evenly among the threads. A few of them finish early and a few of them lag. It is possible that for 2048 images, this gap is the smallest.

For these tests, the variable `OMP_NUM_THREADS` was set to 12. This variable controls the upper limit of the size of the thread team spawned for parallel regions, however, the runtime could pick a smaller number of threads.

| Image size (pixels per side) | Sequential (s) | Sequential SD (%) | OpenMP (s) | OpenMP SD (%) | Speedup |
|---|---|---|---|---|---|
| **128** | 3.2519 | 1.248500876 | 0.392 | 4.974489796 | 8.295663265 |
| **256** | 8.7337 | 2.449133815 | 1.0066 | 3.069739718 | 8.676435526 |
| **512** | 33.6264 | 2.382651726 | 3.892 | 2.083761562 | 8.63987667 |
| **1024** | 179.1006 | 1.062754675 | 21.1715 | 0.953168174 | 8.459513969 |
| **2048** | 11118.7916 | 0.320859747 | 148.3701 | 0.562107864 | 15.6218972 |
| **4096** | 47075.9372 | 0.093827422 | 1648.6 | 0.142860609 | 15.22441495 |
| **8192** | 194487.7004 | 0.027307296 | 14137 | 0.513546014 | 15.09285216 |

**Table 5.1.** Execution times for the sequential and OpenMP implementations of AMI along with their corresponding percentage standard deviations. Values in red were predicted from the trend in Figure 5.1. The highest and lowest achieved speedups are highlighted in blue and pink respectively.

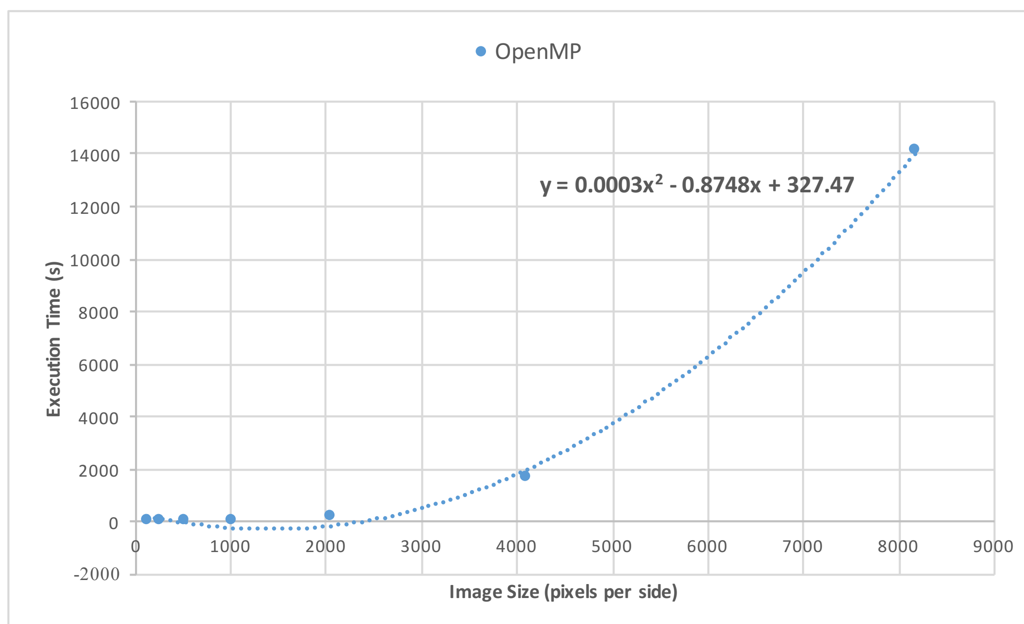



**Figure 5.1.** Average execution time for AMI OpenMP implementation for window resolution of 8, for all image sizes. This trendline was used to predict execution time of sequential implementation for image sizes 2048, 4096 and 8192.

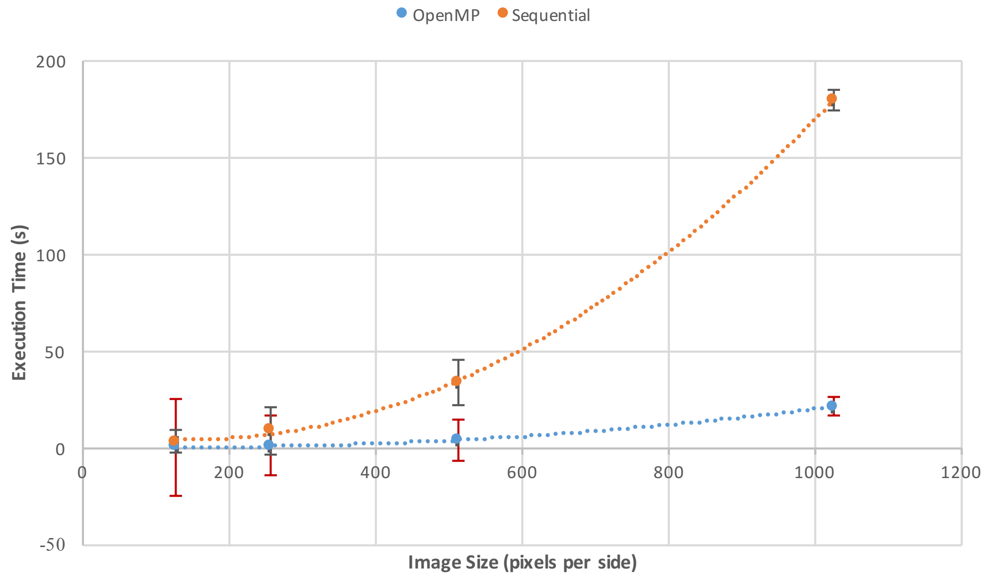



**Figure 5.2.** Average execution time for AMI sequential C++ and OpenMP implementations for a fixed window resolution of 8 and image sizes up to 1024. Relative standard deviation magnified by 5.

Figure 5.2 shows the AMI execution time for the sequential and OpenMP implementations for the data points obtained experimentally for a window resolution of 8. Note that this includes both stages of the algorithm: the ranklet transforms and the mutual information-based feature extraction. The execution time increases with increasing image size, but the rate of increase in case of the OpenMP implementation is considerably slower. For these sets of image sizes, the speedup is rather consistent, with an average of 8.5178 times increase in performance. The relative standard deviation is negligible. We know that theoretically, the timing of every image of the same size takes the same number of operations, and each operation should take the same duration since these do not depend on the pixel value and there is no further simplification made in the algorithm for specific cases, such as when a value is 0. However, it is hard to ignore the fact that in the graph, where the standard deviation has been magnified by 5 for visual aid, we see that the the SD decreases with increasing image size. One possible explanation can be attributed to the Intel Turbo Boost technology. The clock frequency changes based on the number of threads running, being at the lowest value when all threads are running. When the CPU gets heavy load processes, frequent changes in CPU clock speed may interfere with system performance. It is possible that the initial use of OpenMP threading created this spike in execution time deviation, since the clock frequency experienced a sudden change. As the later data points were obtained, the clock frequency

remained at a stable value.

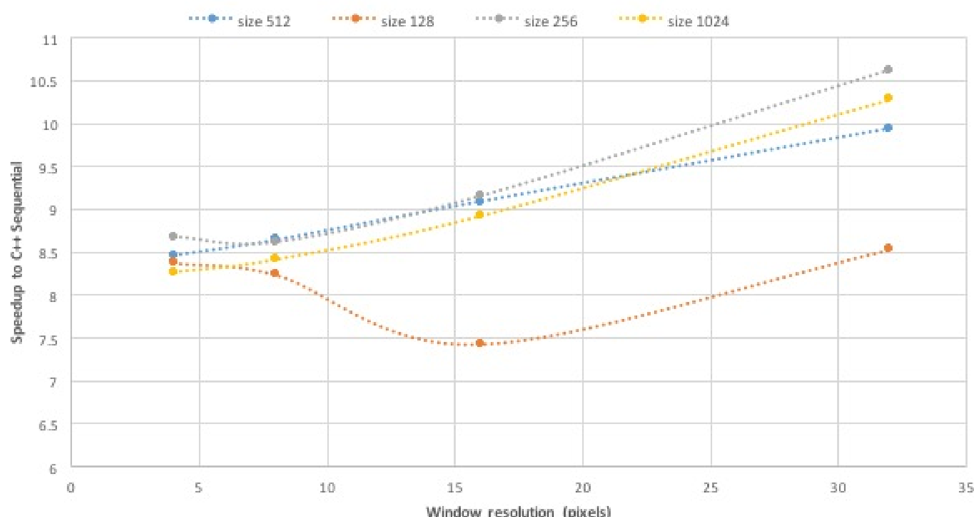



**Figure 5.3.** OpenMP speedup to C++ sequential code for all window resolutions and image sizes up to 1024 pixels per side.

Figure 5.3 shows the OpenMP speedup to sequential code for all window resolutions and images up to 1024 pixels per side. The speedup is simply the execution time achieved by OpenMP implementation divided by execution time taken by C++ sequential implementation. We can observe a couple of things from the graph. Firstly, as window resolution increases, the speedup increases. This is unexpected, since window size affects performance during the ranklet transform for (1) ranking the pixels and (2) summing up the pixels in the treatment group, none of which were run in parallel. Secondly, we notice that the speedup is not increasing in accordance with image size, i.e., it is not always the case that with increasing image size, we get an increase in performance by using OpenMP. This never happens for any window resolution. We do expect to get the least improvement with the smallest image size, which we can conclude experimentally as well. However, we obtain the best speedups for image sizes of 256 pixels per side, followed by the $512 \times 512$ sized images. This could simply mean that resources are the most ideally allocated for particular image sizes.

Next, we explore the different scheduling clauses for image sizes of $512 \times 512$ and a window resolution of 32 pixels per side for the two stages of the AMI algorithm. We note that for both the ranklet transform and the obtention of AMI features, the dynamic scheduling is the best option. Dynamic scheduling is optimal when every iteration has a different workload. In this sense, it is reasonable that this is the best scheduling type for stage 2, since every iteration in

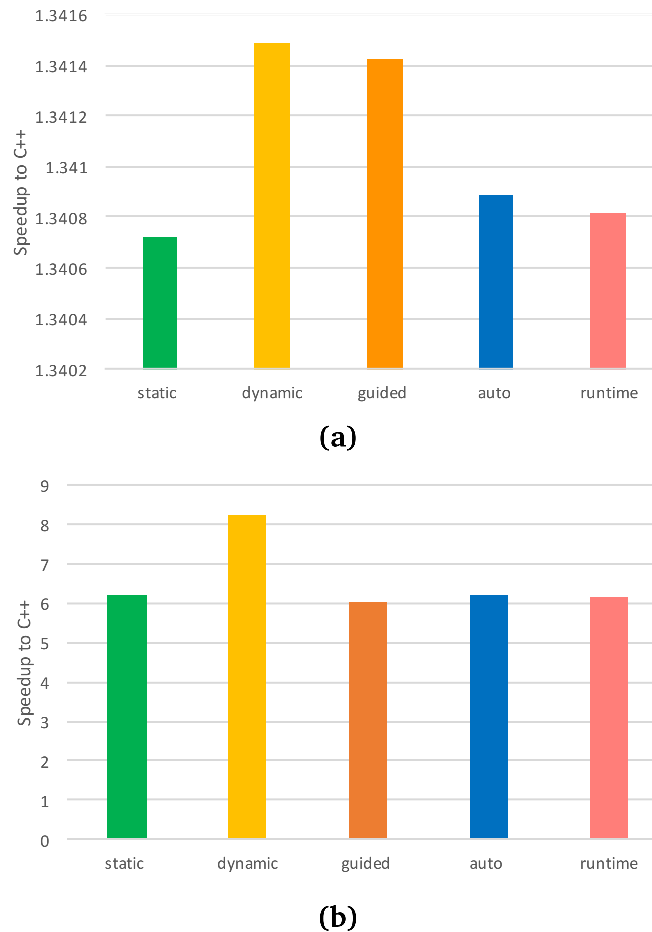



**Figure 5.4.** Speedup to C++ sequential code for different thread scheduling type for ranklet (a) and AMI feature extraction (b). All measurements taken for a $512 \times 512$ size image, and window $r = 32$.

loop D handles sub-images of different sizes. However, for stage 1, the amount of work for every window is the same. We would expect static scheduling to give the best results since every thread computes a pixel value using the same window size, and the workload is split evenly between threads. Remember that we are using 12 Xeon cores, with 6 cores per socket. There is a chance that the inter-socket communication during static scheduling overhead exceeds the overhead caused by dynamic scheduling. Also for stage 1, guided scheduling performs the second best, as it is very similar to the dynamic technique. The only difference is that the chunk size changes as the program goes on.

### 5.2.2 CUDA

In this section, we look at the results for our GPU implementation. Time measurements for CUDA related code were done using cuda events. One `cudaEvent_t` variable was declared as a time stamp before the code to be measured, and one as a time stamp after the code was completed. The function `cudaEventElapsedTime` measures the elapsed time in milliseconds between those two points. There is an important matter regarding the timings done in this manner. Cuda events are used to time code execution related to CUDA activities such as kernel launches. However, there is a big time overhead to simply call the wrapper function. Since the wrapper function is called from host, we make use of `omp_get_wtime()` again.

The timings reported as "total" include the overhead of the wrapper function that has all the memory allocations, kernel calls and device synchronization. The kernel timings are the ones measured from within the wrapper call, using cuda events.

| Image size (pixels per side) | MI kernel (ms) | MI kernel SD (%) | Reduction kernel (ms) | Reduction kernel SD (%) | Total AMI execution time (s) | AMI SD (%) |
|---|---|---|---|---|---|---|
| 128 | 281.55592 | 1.151051564 | 0.01776192 | 1.392889748 | 5.5654429 | 0.170331516 |
| 256 | 370.4116 | 1.735438326 | 0.01859424 | 1.600495637 | 5.645183 | 0.155511319 |
| 512 | 724.49304 | 2.067125624 | 0.01882784 | 3.613071225 | 6.0101403 | 0.273215223 |
| 1024 | 2671.7627 | 2.39858739 | 0.01918944 | 3.191336954 | 7.9560995 | 0.831810411 |
| 2048 | 14891.5 | 1.507120984 | 0.02051392 | 7.355214842 | 20.179801 | 1.112389459 |
| 4096 | 107018.39 | 0.838625193 | 0.02891968 | 11.94802772 | 112.30107 | 0.800140213 |
| 8192 | 811241.4375 | 0.45126461 | 0.031594 | 2.506767961 | 816.5374688 | 0.44875804 |

**Table 5.2.** Execution times for the MI kernel, the reduction kernel, and the total time for the computation of mutual features. Note that these times assume that a ranklet transformed image was given as an input, and therefore, excludes the computation time of ranklet transforms.

For the CUDA version, timings were done separately for stage 1 and stage 2 of AMI. Table 5.2 reports the execution times for different parts of stage 2, the computation of AMI features, for all image sizes. This includes the timings for the MI kernel, reduction kernel and total time. Notice that the timing of the kernels is given in milliseconds, while the total time to compute the features is given in seconds. This indicates how little time the computations actually take relative to the overhead of the "setup" that needs to be done to use the GPU, which includes mostly transfer of the data in our case. This supports the idea that GPUs are best suited for very compute intensive algorithms as long as the overhead of data transfer does not overshadow it.

Table 5.3 shows the total execution times for the feature extraction stage of OpenMP as well, for comparison. We see a performance acceleration of 0.125 to 17.25 times for the

smallest to largest images to the multi-CPU version and of up to 22.5 times to the sequential version, where largest image considered was of size $1024 \times 1024$. As expected, there is a better speedup for the larger the images. Figure 5.5 shows a semi-log graph of the execution times of stage 2 of AMI for all implementations.

| Image size (pixels per side) | MI execution times (s) | |
|---|---|---|
| | OpenMP | CUDA |
| 128 | **0.6964** | 5.5654429 |
| 256 | **0.9026** | 5.645183 |
| 512 | **3.644** | 6.0101403 |
| 1024 | 20.3881 | **7.9560995** |
| 2048 | 145.4976 | **20.179801** |
| 4096 | 1637 | **112.30107** |
| 8192 | 14089 | **816.5374688** |

**Table 5.3.** Execution times for computation of mutual information feature vector in OpenMP and CUDA.

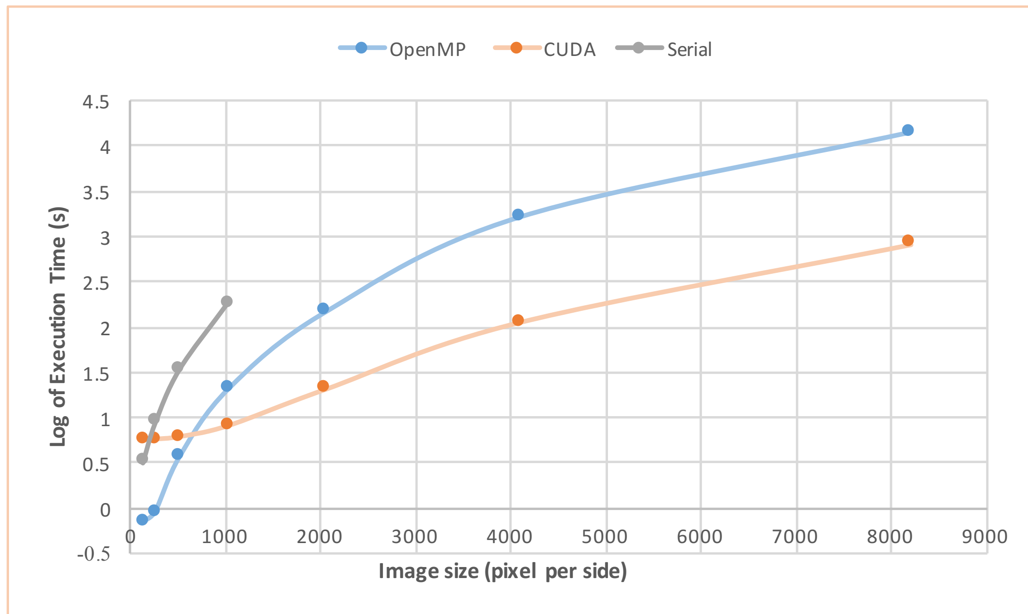



**Figure 5.5.** Semi-logarithmic plot of the measured execution times of the AMI for all implementations.

Next, we measured the performance of stage 1 implemented in CUDA. Table 5.4 shows the timings of the ranklet transform for both parallel implementations for all image sizes

and window resolutions of 8 and 32 respectively. For window with $r = 8$, we notice that the OpenMP version outperforms the CUDA version, while for $r = 32$, there is a mere 0.03 times improvement for 1024 pixels per side sized images. This means running stage 1 of AMI on GPUs is not worth it.

| Image size (pixels per side) | Ranklet execution times (s) | |
|---|---|---|
| | OpenMP | CUDA |
| 128 | **0.0467** | 5.370548 |
| 256 | **0.1125** | 5.425767 |
| 512 | **0.2911** | 5.710389 |
| 1024 | **0.9219** | 6.526797 |
| 2048 | **2.8735** | 10.087327 |
| 4096 | **11.5678** | 23.157837 |
| 8192 | **47.8826** | 72.636063 |

| Image size (pixels per side) | Ranklet execution times (s) | |
|---|---|---|
| | OpenMP | CUDA |
| 128 | **1.0963** | 8.689476 |
| 256 | **5.0269** | 13.159889 |
| 512 | **21.5576** | 27.674815 |
| 1024 | 89.134 | **83.731253** |
| 2048 | 329.376 | **317.217902** |
| 4096 | 1358.05 | **1330.899136** |
| 8192 | 5860.92 | **5813.523042** |

**Table 5.4.** Execution times for ranklet transform with window resolutions 8 (left) and 32 (right) in OpenMP and CUDA. The lowest for each image size shown in bold.

All of our previous measurements were done with the best block and grid configurations, which we discuss now. Recall that for the ranklet stage, we used a 2D block and grid configuration. Table 5.5 shows the execution times using different block sizes. As mentioned in Chapter 4, we always launched enough blocks to cover all pixels in the image. So, for a block configuration of (2,2) more blocks were launched than for a (4,4) block. Through the empirical process of performance tuning we found that the highest execution times were obtained for a thread block size of (4,4), while the best configuration was that of (8,8). It is generally advised that the block size be a multiple of the warp size, to increase occupancy.

Occupancy is a metric that quantifies the ability of a given kernel to supply enough parallel work to achieve a good enough performance. It is the ratio of active CUDA warps to the maximum warps that each SM can execute concurrently. The higher the occupancy, the more effectively is the GPU being used since the latency of stalled warps is hidden. Our best configuration agrees with this idea, since 64 threads are launched per block, and this size is a multiple of 32. We achieve an effective warp scheduling as such.

| Thread block size (dim3) | Image size (pixels per side) | Ranklet execution time (s) |
|---|---|---|
| (2,2) | 128 | 5.97764 |
| | 256 | 7.671878 |
| | 512 | 14.431341 |
| | 1024 | 41.764556 |
| (4,4) | 128 | 9.503035 |
| | 256 | 17.602349 |
| | 512 | 48.000193 |
| | 1024 | 170.323756 |
| (8,8) | 128 | 5.491805 |
| | 256 | 5.85972 |
| | 512 | 7.129224 |
| | 1024 | 12.361647 |
| (16,16) | 128 | 5.567827 |
| | 256 | 5.998324 |
| | 512 | 7.599683 |
| | 1024 | 13.675148 |
| (32,32) | 128 | 5.626153 |
| | 256 | 6.101879 |
| | 512 | 7.800798 |
| | 1024 | 14.352045 |

**Table 5.5.** Ranklet execution times for CUDA implementation with different block configurations. The configurations yielding the best and worst times are highlighted in blue and pink respectively.

We would naively think that the more threads our program has, the better the performance would be. However, our findings disagree. Let us remember that we stored our input images in global memory, which means retrieving our image data will be cached into L2. We strongly rely in this caching mechanism, since in this implementation we did not make use of shared memory. A sound explanation to why, for example, a (32,32) block configuration performs worse than the (8,8) blocks despite both having sizes that are multiple of the warp size would be that the more threads are running concurrently, the more values are attempting to be retrieved from the L2 cache. This introduces a bigger chance of a cache miss, having to read the missing value again from global memory. At the same time, this thread would be evicting

a value that another thread could be possibly needing soon after. Having less threads per block also increases the locality of the data being read.

This argument for less threads blocks creating less cache misses and increasing the locality of data being read can be more clearly observed in the results for the execution times obtained for different block configurations in stage 2 of AMI, Table 5.6. Let us remember that here, we use 1D blocks, and launch a grid with as many blocks as necessary to cover the number of rows of the image minus one. The best configuration is solely of 4 threads per block, and the worst one is of 256 threads per block.

| Threads/Block | MI execution time | |
|---|---|---|
| | Kernel (ms) | Total (s) |
| 2 | 2954.7536 | 8.2442 |
| 4 | 2608.7138 | 7.89452 |
| 8 | 3049.5715 | 8.32692 |
| 16 | 3882.5214 | 9.1678 |
| 32 | 4503.4028 | 9.7751 |
| 64 | 4837.6191 | 10.12865 |
| 128 | 5037.6953 | 10.3303 |
| 256 | 6785.3037 | 12.04553 |

| Threads/Block | MI execution time | |
|---|---|---|
| | Kernel (ms) | Total (s) |
| 2 | 16164.2392 | 21.4385 |
| 4 | 14513.0205 | 19.7449 |
| 8 | 14467.915 | 19.7833 |
| 16 | 19613.986 | 24.9039 |
| 32 | 20688.359 | 25.9491 |
| 64 | 22631.68 | 27.9113 |
| 128 | 25844.6835 | 31.1231 |
| 256 | 27583.19727 | 32.8615 |

**Table 5.6.** Execution times for the computation of AMI features for different block configurations and image size $1024 \times 1024$ (left) and $2048 \times 2048$ (right). The configurations yielding the best and worst times are highlighted in pink and blue respectively.

Finally, the last parameter we tuned was the size of the shared memory for the reduction kernel. Since shared memory is assigned per block, its size represents the number of threads we assign per block. Therefore, a shared memory of 128 bytes represents a launch configuration of 128 threads/block and 256 bytes represents 256 threads/block. Table 5.7 shows the execution times of vector reduction for the sequential and CUDA versions. For image size of 128 pixels per side, we obtain a vector of 127 elements, meaning that we cannot use a shared memory of 256 bytes, so the first corresponding row is blank. Using vector reduction to sum the feature vector only improves the sequential time up to 3 seconds in the best case. Nonetheless, our CUDA MI feature extraction timings in Table 5.2 consider this method rather than sequentially summing the elements to obtain the AMI value.

| Image size (pixels per side) | Sequential (s) | CUDA | | | |
|---|---|---|---|---|---|
| | | Shared memory = 128 | | Shared memory = 256 | |
| | | Kernel (ms) | Total (s) | Kernel (ms) | Total (s) |
| 128 | 5.55529 | 0.018304 | 5.5597 | - | - |
| 256 | 5.6699 | 0.017696 | 5.66279 | 0.01885 | 5.648317 |
| 512 | 5.9956 | 0.018144 | 5.9837 | 0.01846 | 5.9812 |
| 1024 | 7.89452 | 0.01795 | 7.8789 | 0.01952 | 7.888 |
| 2048 | 19.7849 | 0.02026 | 19.9619 | 0.02217 | 19.938 |
| 4096 | 111.8641 | 0.02605 | 111.8207 | 0.0304 | 111.517 |
| 8192 | 815.5116 | 0.0329 | 812.6229 | 0.03267 | 813.79 |

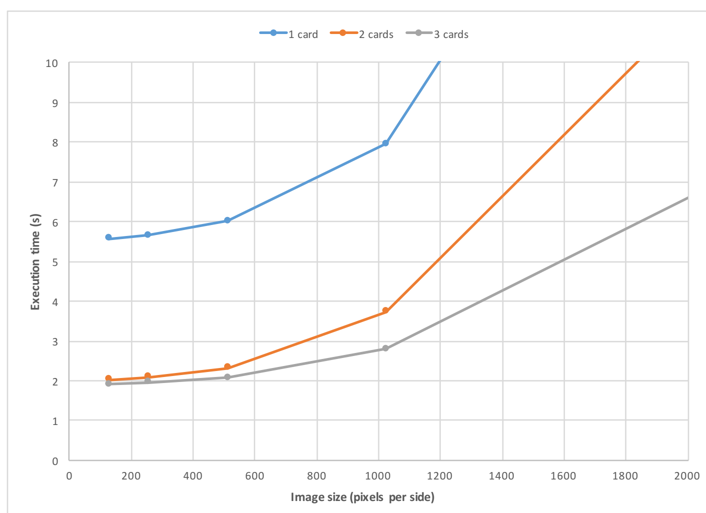
**Table 5.7.** Execution times of vector reduction for sequential and CUDA versions. The CUDA versions were executed with 128 and 256 bytes of shared memory.

### 5.2.3 Heterogeneous implementation

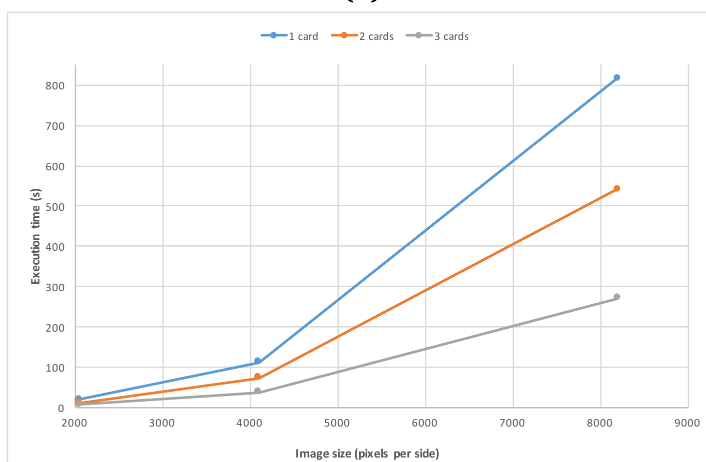
In this section, the results for the heterogeneous implementation are presented. Table 5.8 shows the execution times and speedups using one, two and three NVIDIA cards. We see around a 2x speedup using two cards and 3x speedup using three cards. This is expected since we are allocating the same task load in each card, and there is no communication between one another, making each of them work independently to obtain a feature. Figures 5.6a and 5.6b show the execution times for the calling of MI kernel for images of 128 to 1024 pixels per size (top) and 2048 to 8192 pixels per side (bottom).

| Image size (pixels per side) | Execution times (s) | | | | |
|---|---|---|---|---|---|
| | 1 card | 2 cards | 3 cards | Speedup with 2 cards | Speedup with 3 cards |
| 128 | 5.5654429 | 2.019954 | 1.915759 | 2.755232495 | 2.905085086 |
| 256 | 5.645183 | 2.084816 | 1.953476 | 2.707760781 | 2.889814362 |
| 512 | 6.0101403 | 2.316768 | 2.076941 | 2.594191693 | 2.893746284 |
| 1024 | 7.9560995 | 3.726553 | 2.79557 | 2.134975539 | 2.845966833 |
| 2048 | 20.179801 | 11.647076 | 6.763333 | 1.732606622 | 2.983706554 |
| 4096 | 112.30107 | 72.993183 | 37.597865 | 1.538514494 | 2.986900187 |
| 8192 | 816.5374688 | 540.518915 | 270.560829 | 1.510654754 | 3.017944141 |

**Table 5.8.** Execution times and speedups using 1, 2 and 3 NVIDIA graphics cards.




**(a)**




**(b)**

**Figure 5.6.** Execution times for AMI feature extraction using a different number of NVIDIA cards for images sizes 128 to 1024 (top) and from 2048 to 8192 (bottom).

For an implementation where the ranklet was done in CUDA, we could use the fact that the transformed images, which are the input for the feature extraction step, are already located in device memory. Recall that in our implementation, we compute the ranklet transform in a single card. We would have to transfer two of the ranklet images to the memory of the remaining devices. It is possible that a card has access to the memory of a peer card. This can be checked with the function `cudaDeviceCanAccessPeer( int* canAccessPeer, int device, int peerDevice)`. In our case, device 1 can access the memory of device 0, so the only task left to do is transfer the remaining ranklet image to device 2. For this, we have no choice than to first, bring it back to host, to then allocate it

to device 2. This whole process would save us a device-to-host transfer and a host-to-device transfer, both of which we have found out cause a significant overhead. This approach is left as a work for the future.

## 5.3 Summary

In this chapter, we showed the results obtained from the benchmarking of the implementations described in Chapter 4. For the OpenMP version, we found an increase in performance of up to 15 times, corresponding to images of size $2048 \times 2048$. For the CUDA implementation, we obtained an improvement of 22.5 times in the best case, for the feature extraction stage. For the heterogeneous implementation, we get the expected acceleration to be proportional to the number of cards used.

# 6 Conclusions

The recently proposed AMI algorithm has been proven to be a successful feature extraction method based on the experiments conducted in BUS images, where its advantage over other texture analysis techniques was demonstrated. Despite its superiority in terms of classification performance, the application of AMI is not feasible on larger medical images, since it takes place in the spatial domain. This issue motivated the investigation of parallel schemes for this algorithm. As such, the proposed approaches in this work were tested on mammography images, since they are commonly of high spatial resolution and therefore their analysis usually takes an excessive amount of time.

In this thesis, we implemented four versions the Auto-Mutual Information algorithm: a sequential version in C++, a parallel version using multiple CPUs, a parallel version using GPU and a heterogeneous version using both of the previous options. Along the way, we discussed the design choices made and dug into the limitations that come as a result of them. The OpenMP implementation achieved up to 15 times speedup to the sequential version. The CUDA version achieved up to 22.5 times speedup, which was further reduced thrice in the heterogeneous version using 3 NVIDIA cards. The implementations completed in this work are capable of scaling to multiple nodes and cards. However, note that for the GPU

implementation this does not mean it can be applied to arbitrarily larger sizes. There is a limit to the image sizes that can be used without running into memory issues, determined by the global memory size. The proposed approaches can be potentially applied to other types of high resolution medical images, such as MRI and CT scans, or even to other fields where images processed are usually very large, like satellite images. However, the suitability of AMI for features obtained in contexts other than BUS images still remain to be determined.

The main contributions of this thesis are the following:

- Implementation of AMI in C++, using OpenCV.
- Design and implementation of AMI with OpenMP.
- Design and implementation of AMI in GPUs using CUDA, without using third party computer vision libraries.
- Implementation considering a heterogeneous approach using OpenMP and CUDA.
- Performance improvement to the sequential version for all three cases.

## 6.1 Future Work

The approaches implemented thus far show a considerable improvement in performance compared to the state of the art. However, designing a different parallelization scheme, and adding on to the current one could further improve the work in terms of efficiency. The following suggestions are made:

- Our GPU implementation stored images in global memory, from which all read and write operations were done. A scheme where shared memory is used in intermediate steps would additionally reduce the execution time.
- In our heterogeneous implementation, we used three cards for the feature extraction step. For this, we transferred the transformed ranklet images from one card to the host, then transferred them back to the different cards. We could use advantage of peer-to-peer transfers, where one card can access the memory of another card. As we saw in the results section, a large portion of execution time is spent on host-device communication. Reducing the number of images that have to go through this type of transfer will show an increase of performance.

- In Chapter 5, Table 5.3, the execution times for the feature vector extraction are shown to be better for image sizes of $128 \times 128$ up to $512 \times 12$ when using the OpenMP version than in the CUDA version. A smart implementation that chooses the best method according to the size of the input image could be developed.

- It is hard to ignore the fact that the equipment used in this work can be considered obsolete, and a lot of new technology currently exists. GPUs of compute capability 3.5 or higher have the ability of "dynamic parallelism", which allows to launch new grids from the GPU. This would allow for "nested parallelism" schemes in our design. For example, in the MI feature extraction, not only could we do the image sliding operations in parallel, but parallelize the histogram computation for each sub-image. A parallel histogram scheme was proposed in [42] and could be used as a nested parallel section, in combination with the GPU scheme proposed in this work.

# Bibliography

[1] W. Gómez-Flores, A. Rodríguez-Cristerna, and W. C. D. A. Pereira, "Texture analysis based on auto-mutual information for classifying breast lesions with ultrasound," *Ultrasound in Medicine & Biology*, vol. 45, no. 8, p. 2213–2225, 2019.

[2] W. Gomez Flores, "Descripcion con rasgos de textura."

[3] E. Alpha, O. Awodele, and S. Kuyoro, "The improvement of parallel scientific computing by the application of complex algorithm," Feb 2015.

[4] W. U. A1, "Fork–join model," Jan 2020.

[5] "Cuda," 2017.

[6] Jeremiah, "Cuda memory model," Nov 2013.

[7] B. Ramkumar, R. Laber, H. Bojinov, and R. S. Hegde, "Gpu acceleration of the kaze image feature extraction algorithm," *Journal of Real-Time Image Processing*, Nov 2019.

[8] E. Devolli-Disha, S. Manxhuka-Kërliu, H. Ymeri, and A. Kutllovci, "Comparative accuracy of mammography and ultrasound in women with breast symptoms according to age and breast density," *Bosnian Journal of Basic Medical Sciences*, vol. 9, no. 2, p. 131–136, 2009.

[9] A. Illanes, N. Esmaeili, P. Poudel, S. Balakrishnan, and M. Friebe, "Parametrical modelling for texture characterization—a novel approach applied to ultrasound thyroid segmentation," *Plos One*, vol. 14, no. 1, 2019.

[10] R. C. Gonzalez and R. E. Woods, *Digital image processing*. Pearson, 2018.

[11] F. Smeraldi, "Ranklets: orientation selective non-parametric features applied to face detection," *IEEE Comput. Soc*, vol. 3, no. 4, pp. 379–82, 2002.

[12] M. Masotti and R. Campanini, "Texture classification using invariant ranklet features," *Pattern Recognition Letters*, vol. 29, no. 14, p. 1980–1986, 2008.

[13] M.-C. Yang, W. K. Moon, Y.-C. F. Wang, M. S. Bae, C.-S. Huang, J.-H. Chen, and R.-F. Chang, "Robust texture analysis using multi-resolution gray-scale invariant features for breast sonographic tumor diagnosis," *IEEE Transactions on Medical Imaging*, vol. 32, no. 12, p. 2262–2273, 2013.

[14] W. Gomez, W. C. A. Pereira, and A. F. C. Infantosi, "Analysis of co-occurrence texture statistics as a function of gray-level quantization for classifying breast ultrasound," *IEEE Transactions on Medical Imaging*, vol. 31, no. 10, p. 1889–1899, 2012.

[15] E. G. Learned-Miller, "Entropy and mutual information," Sep 2013.

[16] Y. Xie, "Information theory, ece 587 lecture notes," Sep 2012.

[17] T. M. Cover and J. A. Thomas, *Elements of information theory*. Wiley-India, 2010.

[18] T. Ngo and L. Snyder, "Data locality on shared memory computers under two programming models," 07 1993.

[19] G. R. Andrews, *Foundations of multithreaded, parallel, and distributed programming*. Addison-Wesley, 2000.

[20] B. Barney.

[21] B. Chapman, G. Jost, and R. V. d. Pas, *Using OpenMP: portable shared memory parallel programming*. MIT Press, 2008.

[22] J. Speh, "Openmp: For scheduling," Jun 2016.

[23] J. McKennon, "Gpu memory types - performance comparison," Aug 2013.

[24] "Bank conflicts in shared memory in cuda: Shared memory in cuda in detail: Shared memory and bank conflict ion in cuda."

[25] B. Goossens, J. De Vylder, and D. Van Haerenborgh, "Cuda - exploiting constant memory," Jun 2020.

[26] "Cuda c programming guide."

[27] M. Masotti, N. Lanconelli, and R. Campanini, "Computer-aided mass detection in mammography: False positive reduction via gray-scale invariant ranklet texture features," *Medical Physics*, vol. 36, p. 311–316, Jul 2009.

[28] S. Aydin, R. Samet, and O. F. Bay, "Real-time parallel image processing applications on multicore cpus with openmp and gpgpu with cuda," *The Journal of Supercomputing*, vol. 74, p. 2255–2275, Dec 2017.

[29] P. Kegel, M. Schellmann, and S. Gorlatch, "Using openmp vs. threading building blocks for medical imaging on multi-cores," *Lecture Notes in Computer Science Euro-Par 2009 Parallel Processing*, p. 654–665, 2009.

[30] S. Saxena, S. Sharma, and N. Sharma, "Image registration techniques using parallel computing in multicore environment and its applications in medical imaging: An overview," *2014 International Conference on Computer and Communication Technology (ICCCT)*, 2014.

[31] E. Smistad, A. C. Elster, and F. Lindseth, "Gpu accelerated segmentation and centerline extraction of tubular structures from medical images," *International Journal of Computer Assisted Radiology and Surgery*, vol. 9, p. 561–575, Jan 2013.

[32]

[33] T. Kalaiselvi, P. Sriramakrishnan, and K. Somasundaram, “Performance of medical image processing algorithms implemented in cuda running on gpu based machine,” *International Journal of Intelligent Systems and Applications*, vol. 10, p. 58–68, Aug 2018.

[34] C.-Y. Li and H.-H. Chang, “Cuda-based acceleration of collateral filtering in brain mr images,” *Eighth International Conference on Graphic and Image Processing (ICGIP 2016)*, Aug 2017.

[35] M. A. Alsmirat, Y. Jararweh, M. Al-Ayyoub, M. A. Shehab, and B. B. Gupta, “Accelerating compute intensive medical imaging segmentation algorithms using hybrid cpu-gpu implementations,” *Multimedia Tools and Applications*, vol. 76, p. 3537–3555, Jan 2016.

[36] M. A. Elahi, A. Shahzad, M. Glavin, E. Jones, and M. O’Halloran, “Gpu accelerated confocal microwave imaging algorithms for breast cancer detection,” in *2015 9th European Conference on Antennas and Propagation (EuCAP)*, pp. 1–2, April 2015.

[37] Ansaloni and Pietro, “Analisi di immagini con trasformata ranklet: ottimizzazioni computazionali su cpu e gpu,” Master’s thesis.

[38] S. Microsystems, “Openmp api user’s guide.”

[39] “Granularity and parallel performance,” Jan 2012.

[40] M. Harris, “An easy introduction to cuda c and c,” Sep 2020.

[41] H. Peng.

[42] C. Ledig and C. Chefd’hotel, “Efficient computation of joint histograms and normalized mutual information on cuda compatible devices,” *Siemens Corporate Research, Princeton, NJ, U.S.A.*