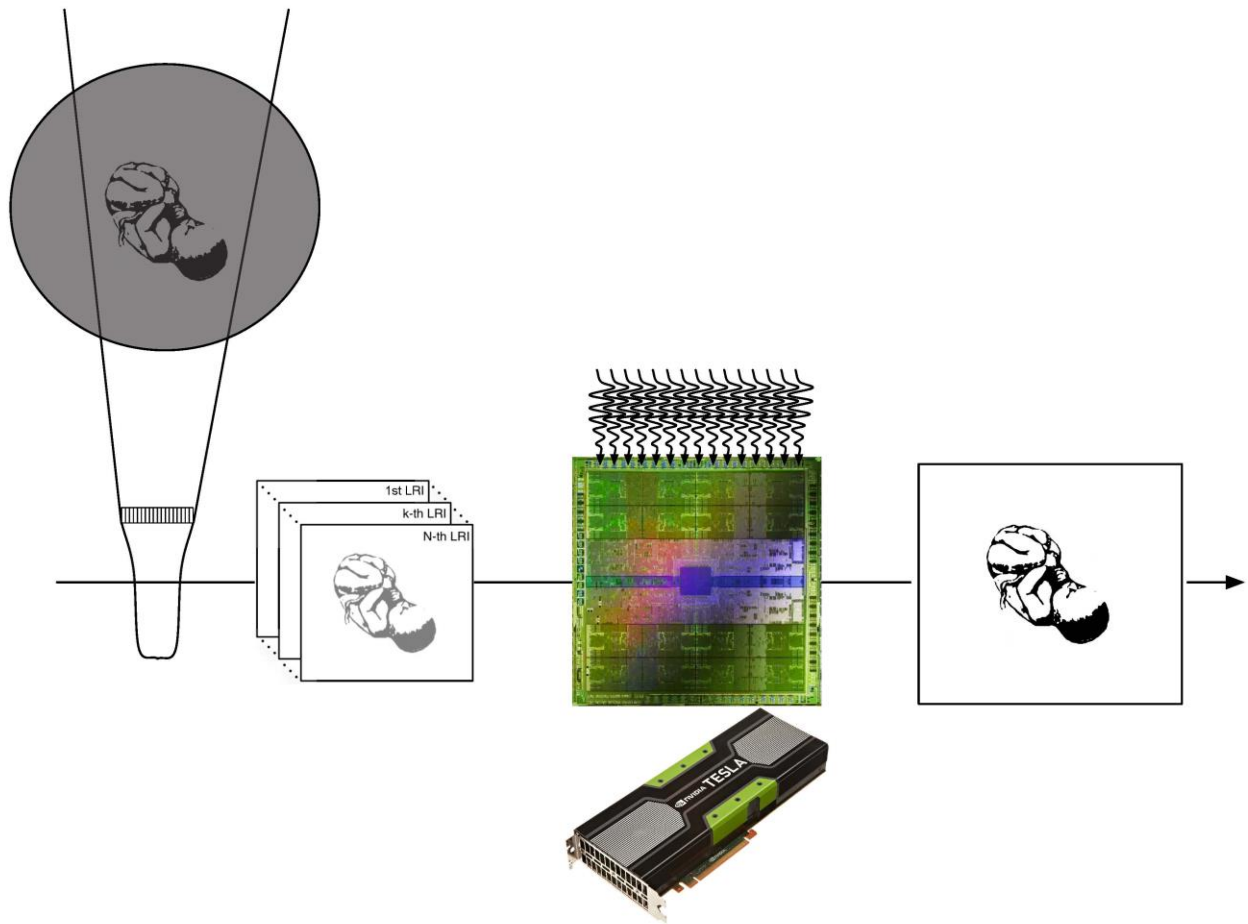


João Leonel Almeida Amaro

Synthetic Aperture beamforming processing on GPUs using OpenCL

Coimbra, Setembro de 2013

Universidade de Coimbra

Synthetic Aperture beamforming processing on GPUs using OpenCL

João Leonel Almeida Amaro

Dissertação para obtenção do Grau de Mestre em
Engenharia Electrotécnica e de Computadores

Júri

Presidente:	Doutor Jaime Batista dos Santos
Orientador:	Doutor Gabriel Falcão Paiva Fernandes
Vogais:	Doutor Vítor Manuel Mendes da Silva

Setembro de 2013

Agradecimentos

Gostaria de começar por agradecer ao meu orientador, o Professor Gabriel Falcão pelo seu acompanhamento constante do meu trabalho e pelo tempo dispensado a solucionar problemas e a propor ideias para melhorar o projecto. Gostaria de dar uma nota de destaque também ao João Andrade e ao Luís Oliveira, colegas de laboratório que se dispuseram a partilhar o seu conhecimento comigo nas variadas vertentes, e assim suavizar a minha curva de aprendizagem; e pela companhia proporcionada nas longas horas passadas no laboratório. Gostaria também de agradecer ao Professor Marco Gomes pelas diversas ocasiões em que interrompi o seu trabalho com dúvidas, e ele se dispôs a me auxiliar, indicando-me sempre o caminho certo a seguir. Gostaria de agradecer também a todos os meus amigos, a alguns pelo apoio proporcionado pelo facto de estarem perto, a outros pela paciência e compreensão pelo pouco tempo que aloquei para eles; por sempre terem manifestado interesse no meu trabalho, e estarem dispostos a ouvir-me explicá-lo exaustivamente. Agradeço também à minha mãe e ao meu pai, pois como família, sempre me apoiaram e aturaram, sem nunca deixar de zelar pelo meu bem-estar. Ao Diogo e ao Xico, pois como bons irmãos, me proporcionaram momentos de descontracção e humor quando chegava a casa. À minha namorada, Carina, por todo o amor e apoio que me deu durante todo este ano, muitas vezes pondo de lado os seus interesses pelos meus, e por me encorajar e disciplinar nos momentos em que a exaustão se começava a instalar. A todos vocês, não existem palavras grandes o suficiente para descrever o meu sentimento de gratidão.

Abstract

Synthetic Aperture (SA) Beamforming techniques represent the future of medical ultrasound imaging. The image quality and low number of artifacts introduced make it an optimal replacement for current B-scan pulse-echo imaging systems. Nonetheless, the computational workload introduced by such techniques is very high. This work suggests Graphics Processing Units (GPUs) as the solution to this problem, by implementing portable parallel kernels, under the Open Computing Language (OpenCL) framework, of a delay-and-sum approach of the SA beamforming technique. The GPU was the choice given its architecture, that perfectly suits parallel tasks by dividing the labour between several work-items, and therefore, its characteristics, advantages and disadvantages are discussed. The OpenCL framework allows for the use of one code in multiple devices, be it GPUs, Central Processing Units (CPUs) or even Field-Programmable Gate Arrays (FPGAs), and its workings are also examined.

Keywords

SA, Ultrasound Imaging, OpenCL, GPU, multiple GPU, Parallel Programming

Resumo

As técnicas de *SA Beamforming* representam o futuro da ultrasonografia. A qualidade de imagem melhorada e a baixa incidência de artefactos de imagem, fazem delas óptimas substitutas para os sistemas actuais, baseados em técnicas B-scan por recepção de eco. Ainda assim, a carga de computação introduzida por este tipo de técnicas é bastante elevada. Este trabalho surere as GPUs como uma possível solução para este problema, ao implementar kernels paralelos portáveis, sob a *framework* OpenCL, de uma aproximação *delay-and-sum* das técnicas de *SA Beamforming*. A GPU é escolhida dada a natureza da sua arquitectura, que se adequa perfeitamente a tarefas paralelas ao dividir o trabalho entre diversos *work-items*, e por conseguinte, as suas características, vantagens e desvantagens são abordadas. A *framework* OpenCL possibilita a utilização de um código em múltiplos dispositivos de computação *manycore*, tais como GPUs, CPUs ou ainda FPGAs, e a sua operação é igualmente discutida.

Palavras Chave

SA, Ultrasonografia, OpenCL, GPU, múltiplas GPU, Programação Paralela

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Objectives 3

1.3 Main contributions 3

1.4 Dissertation outline 3

2 Ultrasound Theory and Imaging Algorithms **5**

2.1 Theoretical Model 6

2.2 System Geometry and Operation Fundamentals 6

2.2.1 Probe Topology 8

2.2.1.A Beamsteering 10

2.2.1.B Beamforming 11

2.3 Ultrasound Wave Generation 12

2.3.1 Amplitude 12

2.3.2 Frequency and Wavelength 13

2.3.3 Waveform 13

2.4 Beamline based Pulse-Echo Imaging (B-scan) 14

2.4.1 Data Acquisition 14

2.4.2 Signal pre-processing 16

2.4.2.A Filtering 16

2.4.2.B Time-Gain Compensation (TGC) 16

2.4.2.C Analog to Digital (A/D) conversion and Envelope Extraction 16

2.5 SA Beamforming Imaging 17

2.5.1 Data Acquisition 18

2.5.2 Signal Processing 18

2.5.3 LRI Formation 19

2.5.4 HRI Compounding 20

2.6 SA Beamforming vs. B-scan techniques 21

2.7 Conclusions 21

3 Parallel Computing Frameworks and Manycore Devices 23

3.1 The OpenCL Parallel Programming Model 24
3.1.1 Platform Model 24
3.1.2 Runtime Model 25
3.1.2.A *Command-queues* 26
3.1.2.B Memory Model and Allocation 26
3.1.2.C Kernel Work Size and Execution 28
3.1.2.D Optimization Strategies 28
3.2 The multicore CPU 30
3.2.1 Control Unit 31
3.2.2 Arithmetic and Logic Unit (ALU) 31
3.2.3 Memory Interface and Caches 32
3.2.3.A Dynamic Random Access Memory (DRAM) and Read Only Memory (ROM) 32
3.2.3.B Caches 33
3.3 The manycore GPU 34
3.3.1 Streaming Processors (SPs) 34
3.3.2 Memory-Hierarchy 35
3.3.3 Mapping the GPUs Memory Regions to the OpenCL Address Spaces 38
3.4 Conclusion 39

4 Parallel Synthetic Aperture Beamforming on Manycore Devices 41

4.1 Outlining 42
4.1.1 SA Beamforming Kernels 42
4.1.2 Simulation Apparatus 46
4.1.3 Performance Metrics 47
4.1.3.A Throughput and Overall processing Time 48
4.1.3.B Memory Transfer Rates (Average and Slowest) 48
4.1.3.C Kernel Occupancy 48
4.2 Experimental Results 48
4.2.1 Single GPU scenario 48
4.2.1.A Advanced Micro Devices (AMD) 49
4.2.1.B NVIDIA 49
4.2.2 Multiple GPUs scenario 49
4.2.2.A Dual AMD 49

4.2.2.B Dual NVIDIA 50
4.2.2.C Hybrid AMD/NVIDIA 50
4.3 Analyzing the obtained results 50
4.4 Conclusions 54

5 Conclusions **55**
5.1 Future Work 56

A Appendix A **59**

List of Figures

2.1 Single transducer lying on the x-axis, and Region of Interest (ROI) in the x-z plane. 7
2.2 Single transducer operating in active mode, with echoes occurring in the scattering points of the medium. 7
2.3 Single transducer operating in passive mode, receiving the echoes, and outputting an electrical signal. 8
2.4 Array of transducers operating in active mode, and arbitrary point *R*. . . . 9
2.5 Array of transducers performing lateral steering. 10
2.6 Array of transducers performing angular steering. 10
2.7 Example of beamforming for three different scenarios. [1, pp. 183] 11
2.8 Gaussian modulated sinusoidal pulse waveform. 14
2.9 Basic B-scan Geometry. 15
2.10 Typical wave transmission/reception in SA Beamforming. 18
2.11 Hilbert FIR filter design. 19

3.1 OpenCL Device Query. 25
3.2 OpenCL Queue Creation. 26
3.3 OpenCL Buffer Creation and Mapping/Enqueuing. 27
3.4 OpenCL Address Space. [2] 27
3.5 OpenCL Kernel Execution/Memory Operations Overlapping. [3] 29
3.6 Intel Haswell architecture die. [4] 30
3.7 AMD Piledriver architecture die. [5] 31
3.8 Current generation quadcore CPU basic diagram. Intel and AMD CPUs differences are highlighted. 32
3.9 Random Access Memory (RAM) stick from Corsair. [6] 33
3.10 GCN Compute Unit Architecture. [7] 35
3.11 GCN Basic Memory Model. [7] 36
3.12 Kepler Stream Multiprocessor (SMX) model. [8] 37
3.13 Kepler Generic Architecture. [8] 38

4.1 Kernel A block diagram and algorithm structure on the GPU. 43
4.2 Kernel B block diagram and algorithm structure on the GPU. 44
4.3 Structural outline of the two parallel approaches followed in this work. . . 45
4.4 Reconstructed image of an in-vivo carotid, computed in the simulations. . 47
4.5 Throughput comparison between the different scenarios and versions. . . 51
4.6 Images reconstructed with various lateral resolutions to evaluate its effect on image quality. Highlighted in red are the regions where the differences can be seen. In subfigures b) and c), the incidence of image artifacts is much lower than that of subfigure a), which has lower lateral resolution. . 53

List of Tables

4.1 Host Platform Specs. 47
4.2 Results for the single AMD scenario. 49
4.3 Results for the single NVIDIA scenario. 49
4.4 Results for the dual AMD scenario. 50
4.5 Results for the dual NVIDIA scenario. 50
4.6 Results for the multiple hybrid GPU scenario. 51

List of Algorithms

List of Algorithms

List of Acronyms

A/D	Analog to Digital
ALU	Arithmetic and Logic Unit
AMD	Advanced Micro Devices
API	Application Programming Interface
BU	Branch Unit
CPU	Central Processing Unit
CU	Compute Unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
FPS	frames-per-second
FSB	Front-Side Bus
GCN	Graphic Core Next
GPR	General-Purpose Register
GPU	Graphics Processing Unit
HRI	High-Resolution Image
LDS	Local Data Shared Memory

LRI	Low-Resolution Image
MTR	Memory Transfer Rate
N/A	Non-Applicable
NDE	Non-Destructive Evaluation
OpenCL	Open Computing Language
OS	Operating System
PCIe	Peripheral Component Interconnect Express
RAM	Random Access Memory
ROI	Region of Interest
ROM	Read Only Memory
SA	Synthetic Aperture
SDK	Software Development Kit
SFU	Special Function Unit
SMX	Stream Multiprocessor
SP	Streaming Processor
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SNR	Signal-to-Noise Ratio
SRAM	Static Random Access Memory
TGC	Time-Gain Compensation
TP	Thread Processor
VLIW	Very Long Instruction Word

1

Introduction

Contents

1.1 Motivation 2
1.2 Objectives 3
1.3 Main contributions 3
1.4 Dissertation outline 3

Since the inception of the medical imaging field, the ultrasonography has been shown to be a utmost useful tool. Its applications range from diagnosis to the control of biological processes such as pregnancies, ovary cysts, vascular structures' blood flow, and also operates as a guiding system in certain types of biopsies and other surgical procedures. Because of its application range, ultrasound systems are widely used, but they require the use of large amounts of very complex hardware, such as Digital Signal Processors (DSPs) and arrays of Field-Programmable Gate Arrays (FPGAs). The complexity resides not only in its physical nature, but also in the way they are programmed to perform the desired task. On top of that, if we consider, for example FPGAs, they represent significant costs, which means that current ultrasound systems have ample space for improvement, in every way. The maintenance of such devices is not trivial, and can represent additional costs.

1.1 Motivation

In medical imaging, the most common ultrasound technology is the beamline based B-scan, a technique that displays a 2-D image of the Region of Interest (ROI). To do so, it is required to perform a full sweep of the entire array of transducers in order to be able of generating a single image. Additionally, since the transmitted wave is a beamline, it is very common to obtain image artefacts, a highly negative aspect, considering the sensitive nature associated with medical imaging.

Alternatively, Synthetic Aperture (SA) Beamforming (a technique developed in the 1950s, but only in recent years being used for medical applications) emulates a spherical wave by performing a relative delay transmission of several of the transducers, minimizing image artefacts. A given image is the compounding of a full sweep of the transducer array. To obtain a new image, only a new firing is required, thus addressing the ultra-high frame-rate issue. Of course, the computational workload brought on by this technique is several times higher than that of the conventional beamline method, which represents a significant problem. As previously stated, the computational workload is already an undesirable aspect of current systems because of the required hardware resources to perform the imaging.

There are currently two frameworks that allow the programmer to program parallel tasks to be implemented on Graphics Processing Units (GPUs): the Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL). The former is only compatible with NVIDIA GPUs, although it is highly optimized for these architectures. OpenCL is an open standard, meaning all the major brands include in their GPUs support for this Application Programming Interface (API). As an added functionality, Central

Processing Units (CPUs), DSPs and other types of computing devices nowadays support OpenCL. The downside to this interoperability is that OpenCL was not designed towards a specific architecture, meaning a certain task in OpenCL will most likely come second to CUDA, in terms of performance. To compensate this, the manufacturers have been developing architecture-oriented functionalities that leverage the performance to peaks similar to those obtained with CUDA (then again, sacrificing the interoperability of OpenCL code).

1.2 Objectives

The core of this thesis work is to take advantage of this parallel processing potential to implement a time-domain version of the SA Beamforming imaging algorithm, using GPUs and CPUs under the C++ OpenCL API for parallel applications. The final goal consists of proving the feasibility of a medical ultrasound imaging system with increased portability, achieving ultra-high frame-rates and improved image quality than current systems, while using GPUs and CPUs; ubiquitous devices one might find in its very own computer.

1.3 Main contributions

To enhance current medical ultrasound imaging systems, massive computing power is required. Therefore, GPUs and CPU are proposed in this work, and they show it is possible to achieve portability over different many-core systems, that are capable of running OpenCL kernels. The system provides real-time capability, achieving near 350 frames-per-second (FPS), and it also features adaptability to the hardware characteristics, ensuring proper performance.

1.4 Dissertation outline

This thesis is structured in five chapters. Following the introduction, Chapter 2 will focus on the basic principles of ultrasound imaging, particularly in medical applications, as well as the signal processing involved in the SA beamforming method. In chapter 3, GPU architecture will be discussed, followed by an explanation of the workings of the OpenCL framework. Chapter 4 will feature the experimental results obtained along the work developed. Finally, in Chapter 5, we will discuss the conclusions of this work, while also providing a path for future work in this field.

2

Ultrasound Theory and Imaging Algorithms

Contents

2.1 Theoretical Model 6
2.2 System Geometry and Operation Fundamentals 6
2.3 Ultrasound Wave Generation 12
2.4 Beamline based Pulse-Echo Imaging (B-scan) 14
2.5 Synthetic Aperture (SA) Beamforming Imaging 17
2.6 SA Beamforming vs. B-scan techniques 21
2.7 Conclusions 21

This chapter discusses the basic principles that serve as foundation for ultrasound imaging in medical applications, in general. Nonetheless, the reader is expected to have a certain knowledge of wave propagation and signal processing theory. If that's not the case, the reader can refer itself to [9] [1]
SA Beamforming and Beamline-based B-scan methods are exploited in more detail, the former due to its relevance regarding this thesis, the latter as a term of comparison against the first, and for currently being the most common choice in medical ultrasound imaging systems.

2.1 Theoretical Model

Since ultrasound imaging is modelled by wave propagation theory, it is important to establish the particular conditions that describe the problem. Radar and Sonar ultrasound imaging evaluate distant objects that lie in the Fraunhofer region, or Far-field, as opposed to Non-Destructive Evaluation (NDE) and medical ultrasound imaging, whose ROI mainly lie on the Fresnel Region, or Near-field [10, pp. 413].

2.2 System Geometry and Operation Fundamentals

In ultrasound imaging, transducers are used to transmit acoustic signals, and also to receive the reflections generated by discontinuities in the medium. An acoustic wave is called an ultrasound wave when its frequency is high enough such that the wave is above the human audible range. Usually, this threshold varies from person to person, but it's located around $20kHz$ [1]. This chapter has the purpose of conveying the reader with a basic knowledge of the fundamentals of ultrasound imaging, particularly beamline based B-scan and medical SA beamforming. Let's consider the scenario represented in Fig 2.1:

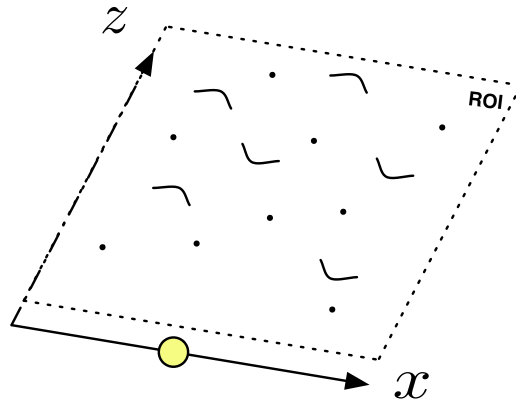


Figure 2.1: Single transducer lying on the x-axis, and ROI in the x-z plane.

A single circular electro-acoustic transducer, to which we can refer as aperture [AFT] positioned along the x-axis is evaluating a ROI placed in the x-z plane. When the transducer is in active mode, it is excited by an electrical impulse, causing it to transmit an acoustic wave into the ROI, as seen of Fig 2.2. This acoustic wave propagates through the medium, and when a discontinuity is hit, the structures have different acoustic impedances. This, of course, means that the discontinuity is also characterized by a reflection coefficient, and that wave reflections will occur in this point.

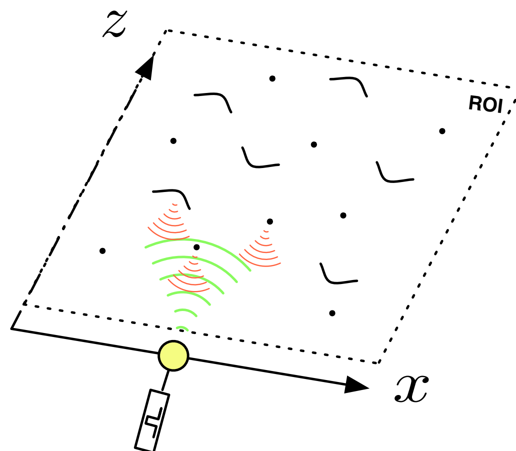


Figure 2.2: Single transducer operating in active mode, with echoes occurring in the scattering points of the medium.

When operating in passive mode, the transducer can be viewed as a receiver, which is to say, given a received echo, the transducer will generate an electrical signal, whose amplitude will depend on the amplitude and frequency of the acoustic signal.

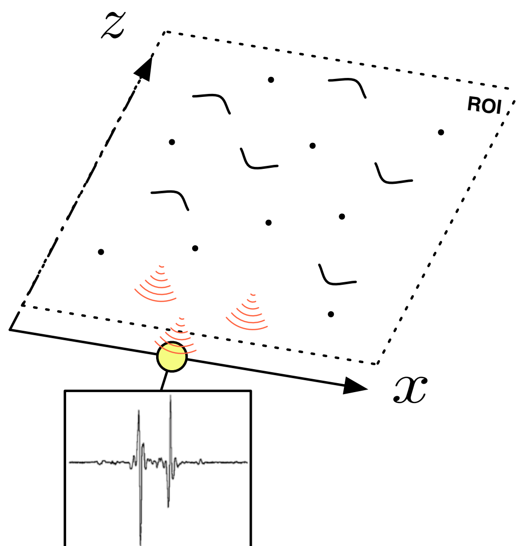


Figure 2.3: Single transducer operating in passive mode, receiving the echoes, and outputting an electrical signal.

2.2.1 Probe Topology

In medical ultrasound imaging, there are many probe topologies. On the course of this work, the preferred topology is the linear transducer phased array. This topology allows for a number of techniques, such as beamsteering and beamforming, that endow the system with more powerful information [1, pp.173-181].

Still considering the scenario described by Fig. 2.1, and given the fact that the wave is acoustic, its effect in the field manifests itself in the form of a pressure.

Using the Huygens principle, that states that every point in a wavefront is a source of wavelets [11, pp. 412], and if the condition defined in equation 2.1 is met, we can treat the aperture, whose diameter is represented by a, as a circular diffraction slit. In this specific case, this aperture corresponds to the transducer surface.

$$a < \lambda, \tag{2.1}$$

λ representing the wavelength of the transmitted wave.
The pressure generated due to the acoustic perturbation induced by the source (the transducer) in the arbitrary point R can therefore be calculated by:

$$P(x,z) = \iint_{surface} A_0(x_0,z_0) \frac{e^{j(wt - kd(x_0,z_0,x,z) + \varphi(x_0,z_0))}}{d(x_0,z_0,x,z)} ds, \tag{2.2}$$

where $A_0(x_0,z_0)$ and $\varphi(x_0,z_0)$ represent the reference amplitude and phase of the infinitesimal field source point located at coordinates (x_0,z_0), $d(x_0,z_0,x,z)$ is the distance between the source point and R, and k is the wave number.

To compute the pressure at an arbitrary point R due to the perturbation caused by an array of transducers (see Fig 2.4) with length L comprised of N individual transducers, the superposition theorem is employed [1, pp. 173], and equation 2.2 becomes:

$$P(x,z) = \sum_{i}^{N} \iint_{surface-i} A_{0i}(x_0,z_0) \frac{e^{j(wt-kd(x_0,z_0,x,z)+\varphi_i(x_0,z_0))}}{d(x_0,z_0,x,z)} ds_i. \tag{2.3}$$

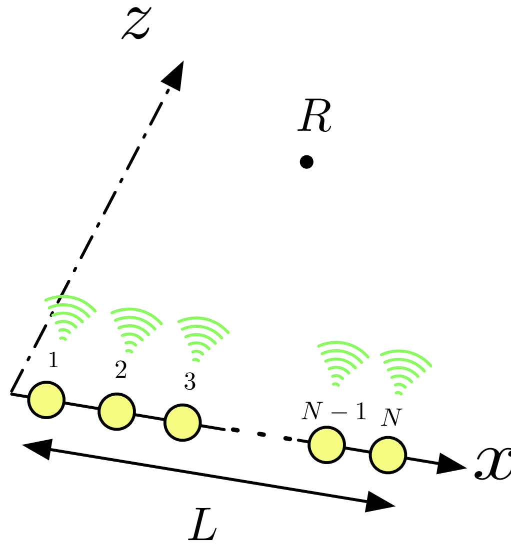


Figure 2.4: Array of transducers operating in active mode, and arbitrary point R.

The position of each transducer, x_i, is given by:

$$x_i = (i-1) \times \Delta_x - \frac{L}{2}. \tag{2.4}$$

The dimensions and spacing of the transducers (represented by Δ_x in Eq. 2.4) are of critical importance, as an incorrect design will give way to catastrophic results, with the appearance of grating lobes in the radiation pattern [1, pp. 175]. The condition to prevent this undesired phenomenon is:

$$\Delta x \leq \lambda. \tag{2.5}$$

To conclude this section, we establish that the system will operate in *strip-map mode*, as the ROI will be swept along the x-axis, and the reconstructed image will be shaped like

a ”strip”.

2.2.1.A Beamsteering

Beamsteering is the process of electronically change the radiation pattern of an antenna, in this case, of the array. The simplest way of achieving this is to perform lateral steering, where in the i-th firing, transducers $i \times n$ to $i \times n + m$ are used in the transmission, whereas in the $i+1$-th firing, the transmitting transducers are $(i+1) \times n$ to $(i+1) \times n + m$ [1, pp. 176]. This process is represented in Fig. 2.5.

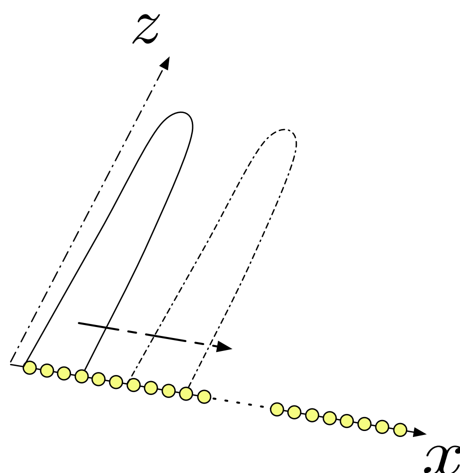


Figure 2.5: Array of transducers performing lateral steering.

Another common type of beamsteering is the angular steering:

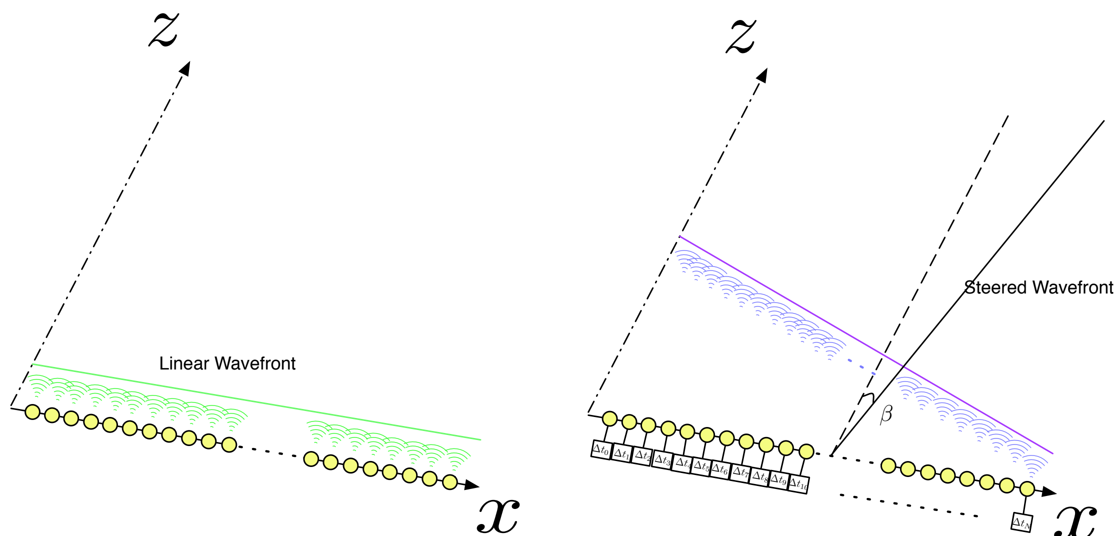


Figure 2.6: Array of transducers performing angular steering.

Until now, we have only considered beams perpendicular to the array of transducers, but through angular steering, it is possible to tilt the transmission, even if only up to a certain angle. This is possible by introducing relative delays to each transducer element, so that the waves from each transducer element reach the dotted line in Fig. 2.6 at the same time [1, pp. 178]. Taking equation 2.4, the expression for each transducers' relative delay is:

$$\Delta_i = [(i-1) \times \Delta x - \frac{L}{2}] \times \frac{sin(\beta)}{v_p}, \tag{2.6}$$

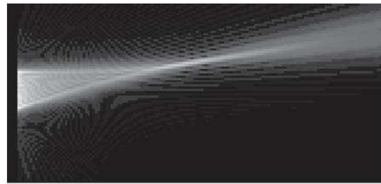
where β corresponds to the angle between the z-axis and the direction of propagation of the wavefront. This angle is limited, under penalty of appearing grating lobes. Therefore, in addition to equation 2.5, the design of the system must also obey:

$$\beta_{max} = \sin^{-1}(\frac{\lambda}{\Delta x} - 1). \tag{2.7}$$

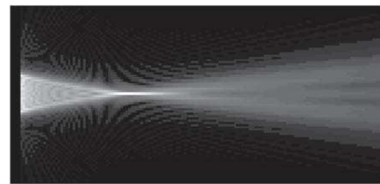
2.2.1.B Beamforming

By combining the aforementioned methods of beamsteering and by shaping the acoustic beam, we arrive at a process called beamforming, whose main advantage is that of achieving beam focusing [1, pp. 182]. Consider a new arbitrary point over the z-axis, lying at a distance F from the axis' origin. The pulse (traveling with a propagation speed of v_p) emitted from the central transducer will reach the focal point after:

$$\Delta t = \frac{F}{v_p}. \tag{2.8}$$



(a) $\beta = -10°; F = 80$mm



(b) $\beta = 0°; F = 50$mm

(c) $\beta = 20°; F = 90$mm

Figure 2.7: Example of beamforming for three different scenarios. [1, pp. 183]

Fig. 2.7 represents three different scenarios where the beam is angled with different degrees β, and whose focal point lies at different distances F.

To achieve focusing, the pulses emitted by all the transducers must reach the focal point at the same time. With x_i representing the position of the i-th transducer:

$$\frac{\sqrt{x_i^2 + F^2}}{v_p} = \frac{F}{v_p}, \tag{2.9}$$

which yields:

$$\Delta t_i = \frac{F - \sqrt{x_i^2 + F^2}}{v_p}, \tag{2.10}$$

where Δt_i stands for the relative time delay of the i-th transducer. It is possible that equation 2.10 outputs a negative time delay. In such event, the solution is to simply shift the reference transducer to that whose distance to the focal point is shorter (alternatively, the one with the most negative time delay).
Finally, to achieve beamforming, it is necessary to combine the focusing effect described above with the lateral and angular steering concepts. This is achieved by combining Equations 2.10 and 2.6.

2.3 Ultrasound Wave Generation

Acoustic transducer elements are composed of a group piezoelectrical crystals, that vibrate when subjected to an electric current. This vibration generates an acoustic wave [1]. This wave can be characterized by its amplitude, waveform, frequency and wavelength. To choose a pulse for transmission, it is important to first understand how each of these characteristics affect the global performance of the system.

2.3.1 Amplitude

Depending on the characteristics of the input signal, the amplitude of the acoustic wave can vary. The maximum achievable amplitude is limited by the piezoelectric crystals. The intensity of a wave is directly proportional to the square of the amplitude. When the intensity of a propagating wave is high, non-linear effects such as the appearance of harmonic waves start to appear.

2.3.2 Frequency and Wavelength

Frequency and wavelength are interdependent, since one can be obtained from the other by the following relation:

$$\lambda = \frac{v_p}{f}, \tag{2.11}$$

where f is the frequency in Hz, λ represents the wavelength in meters and v_p corresponds to the propagation speed, in m/s.

The frequency of the transmitted pulse directly affects the spatial resolution of the reconstructed image. On the other hand, when the frequency is raised, so is the attenuation, defined by the absorption of a part of the wave's energy by the medium, and also dependent on the distance covered by the wave. The following equations give a simplified view of the total attenuation:

$$\alpha_c = \alpha_0 + \alpha_1 f^y, \tag{2.12}$$

$$A(x, f) = \alpha_c(f) \times r, \tag{2.13}$$

with x representing the distance from the transducer, α_c representing the attenuation coefficient and α_0 and α_1 are constants depending on the physical parameters of the medium. Given the fact that in medical imaging, the ROI is mainly comprised of soft tissues, it is assumed that $y = 1$ and thus, the relation between the attenuation coefficient and the frequency is linear. In different scenarios, y can assume any value between 1 and 2, depending on the nature of the tissues under evaluation.

This puts the choice of the frequency of operation into a new perspective, since higher frequencies will yield lower distances of penetration, but better resolution, so high frequencies are most suited for the evaluation of superficial tissue. On the contrary, lower frequencies, while achieving lower spatial resolutions, can travel a greater distance, and are best suited for the appraisal of deeper structures [12].

2.3.3 Waveform

The nature of medical applications of ultrasound imaging dictate the requirement for short, time-domain wise, pulses, or broadband pulses. Although there are many possibilities that fulfil these needs, it is very common to use the Gaussian modulated sinusoidal pulse [1, pp. 99] [13], as seen in Fig. 2.8, and defined by:

$$g(t) = e^{-\beta t^2} \cos 2\pi f_c t, \tag{2.14}$$

where f_c stands for the center frequency of the pulse, and β represents its width.

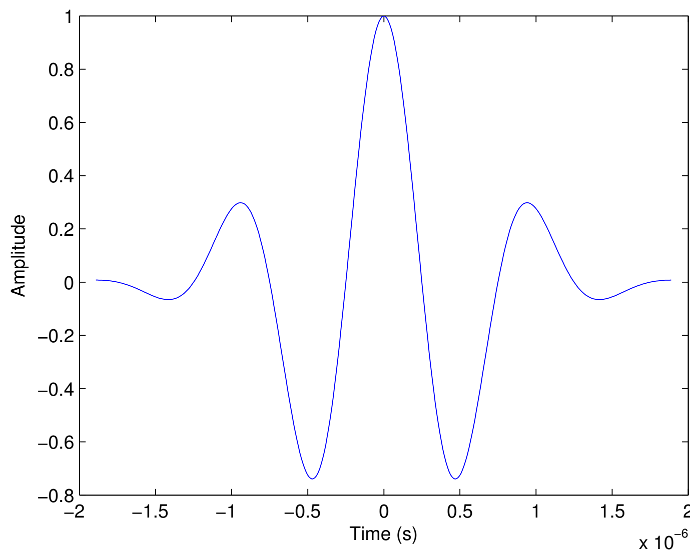


Figure 2.8: Gaussian modulated sinusoidal pulse waveform.

2.4 Beamline based Pulse-Echo Imaging (B-scan)

The B-scan is widely used throughout the world, gathering recognition as the most common method of medical ultrasound imaging method.

2.4.1 Data Acquisition

Pulse-Echo Imaging systems are based on the transmission of a broadband pulse into the medium, and the the reception of the echoes, either by the transmitting transducers or a set of transducers exclusively dedicated to signal reception. These echoes are originated by the transmitted signal traversing discontinuities in the medium, so that the difference in the acoustic impedance and consequent reflection coefficient, provide that a fraction of the signal energy is reflected. This reflection is an angle-related function, and thus, the echo propagates in different directions. Current B-scan imaging systems already perform transmit beamforming, so unlike primordial systems, where only echoes travelling along the beamline were used, current systems provide much accurate information [14, pp. 93]. Each receiving transducer will sample one A-line, and the final image is reconstructed from the A-lines from every receiving transducer of the array. After this data acquisition

step, the signal must endure a signal processing routine before it displays the reconstructed image.

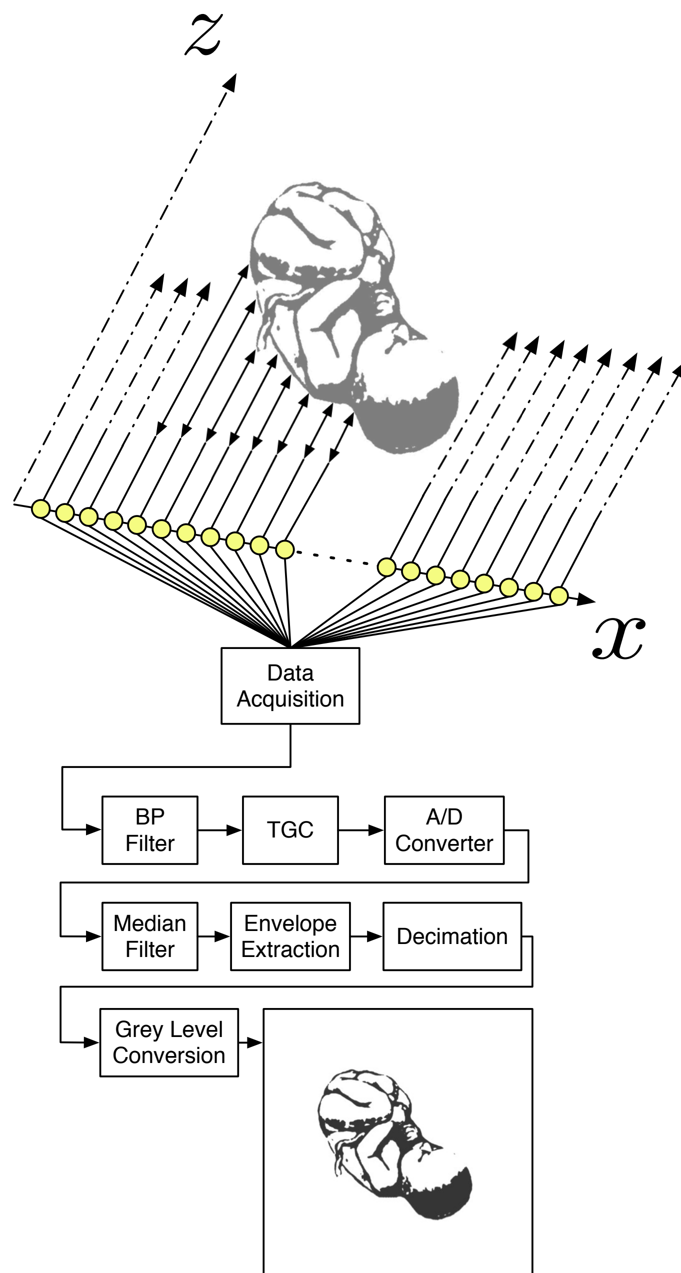


Figure 2.9: Basic B-scan Geometry.

2.4.2 Signal pre-processing

2.4.2.A Filtering

The signal acquired by the receiving transducers includes not only the desired echoes, but also undesired noise. Consequently, this signal must undergo filtering, to eliminate as much as possible the noise (there are many sources to this noise, but in the scope of this work, they are irrelevant). The first step is high-pass filtering, since the frequency of operation is in the order of MHz, and lower frequencies carry no information. Before the Analog to Digital (A/D) conversion can take place, low-pass filtering must also be applied to the signal, mainly to limit the maximum frequency to the Nyquist frequency (otherwise, the A/D conversion would cause aliasing of the signal). These two filtering steps could also be grouped in one filter, a band-pass filter [1, pp. 208].

2.4.2.B Time-Gain Compensation (TGC)

Due to the effects of attenuation, the amplitude of the signal that reaches deeper structures is significantly lower when compared to more superficial tissue. The following assumptions are made:
-The propagation speed , v_p, remains constant throughout the medium.
-The reflectivity of the scatterers is relatively equal for all the ROI.
-The attenuation coefficient, $\alpha_c(f)$, is also constant. This assumption is accepted if there are no bones or air in the acoustic path.

Thus, a gain factor is applied to the signal to mitigate the effects of the attenuation in the reconstructed image, and guarantee that equal structures at different depths appear the same, in the end of the image reconstruction. This process is denominated TGC [1, pp. 205].

2.4.2.C A/D conversion and Envelope Extraction

The signal is now digitized and passes through a median filter, to smooth out any isolated peaks. In an imaging system, only the changes in texture represent useful information, and additional information, such as the representation of the pulse's waveform is dispensable, so the next step is to extract the signal's envelope, $\varsigma(t)$ [1, pp. 208]. To achieve this, the Hilbert transform, whose operator is represented by $H\{\}$. The frequency response of the Hilbert transform is given by:

$$H(j\omega) = -j \times sign(\omega). \tag{2.15}$$

As can be seen from Eq. 2.15, the Hilbert transform shifts all the negative frequencies by 90°, and all the positive frequencies by −90°. The following equation is the next step in the process. The resulting signal $\hat{S}(t)$ is known as the ”analytic representation” of the signal $S(t)$. Its properties are object of discussion in [15].

$$\hat{S}(t) = S(t) + j \times H\{S(t)\}, \tag{2.16}$$

$$\varsigma(t) = |\hat{S}(t)|. \tag{2.17}$$

Finally, the envelope of the original signal $S(t)$ is extracted by computing the absolute value of $\hat{S}(t)$.

Since the acquired signal was a function of time, decimation is required to migrate from a time scale to a distance scale. Finally, defining a vector of N grey levels, whose minimum and maximum are G_{min} and G_{max}, respectively, the signal is quantized, by using the following relation:

$$\breve{\varsigma}(t) = \begin{cases} G_{min} & \text{, if } \varsigma(t) < G_{min} \\ G_{max} & \text{, if } \varsigma(t) > G_{max} \\ \text{round}(N \times \frac{\varsigma(t) - G_{min}}{G_{max} - G_{min}}) & \text{, otherwise} \end{cases} \tag{2.18}$$

2.5 SA Beamforming Imaging

SA Beamforming Imaging is a method of performing ultrasound imaging that, unlike other methods, such as the one described in the previous section, take advantage of the additional information provided by the echoes’ phase, being recognized as a coherent method [14, pp. 3].

Aside from receive beamforming, this method also performs transmit beamforming. To achieve this, during the process of constructing an image, reconstructed images of subregions of the ROI, also known as a Low-Resolution Image (LRI), are also computed, and then compounded into an High-Resolution Image (HRI) [16].

There are several imaging reconstruction algorithms, but the scope of this thesis’ work will only broach the Delay-and-Sum procedure.

2.5.1 Data Acquisition

In SA Beamforming Imaging, the way data acquisition is performed differs from the one discussed in Sec. 2.4.
On the m-th firing, a set of transducers ranging from m to $m+M$ will emulate the transmission of a spherical wave, as displayed in the following figure:

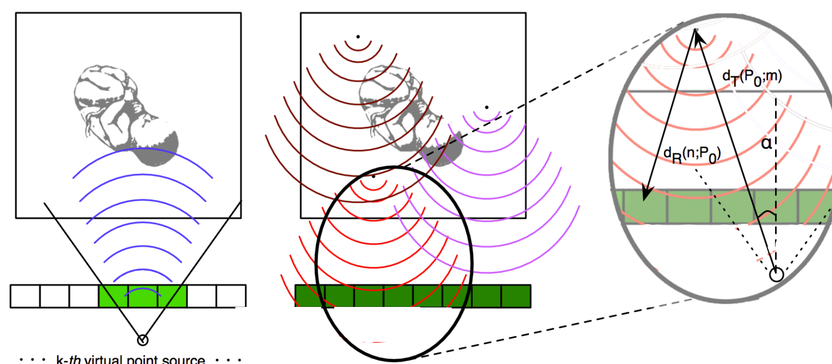


Figure 2.10: Typical wave transmission/reception in SA Beamforming.

When the ROI is illuminated, as discussed in the previous sections, discontinuities in the acoustic impedance, characterized by scattering structures, such as tissues, different cells, organs, etc., part of the wave's energy will be reflected. The echo's are then sampled by the entire (or just part) of the array of transducers, and then processed.

2.5.2 Signal Processing

After the data acquisition is completed, similarly to the Pulse-Echo technique, the signal is passed through two filtering steps, to minimize noise and consequently increase the Signal-to-Noise Ratio (SNR), then TGC is applied and the signal suffers A/D conversion. The following step again requires the transforming of the signal using the Hilbert transform. As can be seen from Eq. 2.16, the resulting signal is a complex one, and further exploring its spectral frequency, it is obvious that the original content is preserved. Furthermore, this signal includes phase information of the original data, making it a useful tool, signal processing-wise [17] [18]. Since the main focus of this thesis focuses over SA beamforming techniques, it is also important to shed some light over how the analytic representation was calculated. Eq. 2.15 provides insight over the frequency response of the Hilbert Transform. To apply this transform without crossing over to the frequency-domain, the signal was convolved with a Finite Impulse Response (FIR) filter with L taps, whose impulse response is given by:

$$h[l] = \begin{cases} 0 & \text{, for even} l \\ \frac{2}{\pi \times l} & \text{, for odd} l \end{cases} \tag{2.19}$$

The FIR filter coefficients were obtained using the least-square method, that approximates the frequency response of the desired filter, by minimizing the error between the ideal response and the approximated one. This was performed in Matlab, using the following commands:

```
d = fdesign.hilbert(100,0.1); %(1st argument is the order of the filter
   %2nd represents the width of the transition band)
designmethods(d);
hd = design(d, 'firls');
hilbertCoeff = hd.Numerator;
```

Figure 2.11: Hilbert FIR filter design.

$$a_{n,m}[g] = \sum_{l=1}^{L} h[l] \times x_{n,m}[g-l], \tag{2.20}$$

In Eq. 2.20, a represents the result of the FIR filtering operation, and x the acquired discrete signal.

2.5.3 LRI Formation

After the analytic data samples are computed, the time scale is mapped into a physical scale.
The requirements to perform such task include computing, the following quantities are defined [17]

$$\tau_{n,m}[P_0] = \frac{d_T(P_0;m) + d_R(P_0;n)}{v_p}, \tag{2.21}$$

where $\tau_{n,m}$ represents the focusing delay from the m-th firing to the n-th receiving transducer,P_0 refers to the pixel for which the focusing delay is being calculated. $d_T(P_0;m)$ stands for the distance between virtual source point m and pixel P_0. $d_R(P_0;n)$ represents the distance between said pixel and the n-th receiving transducer. Fig. 2.10 also illustrates this concept.

The focusing delay allows us to calculate other two quantities required to perform the scale migration:

$$k = \lfloor f_s \tau_{n,m}[P_0] \rfloor, \tag{2.22}$$

$$\chi = 1 + k - f_s \tau_{n,m}[P_0]. \tag{2.23}$$

k represents the depth sample number associated with P_0, f_s is the sampling frequency, and χ is the interpolating weight between proximal depth samples.

To perform the LRI calculation, each receiving channel contribution is taken into account, but it is obvious that the receiving channels closer to the transmission channels have more important information, and therefore an arbitrary window function ψ will be applied:

$$\alpha_{n,m}[P_0] = \psi \times a_{n,m}[k] + (\psi - 1) \times a_{n,m}[k+1]. \tag{2.24}$$

Given an arbitrary pixel P_0, its m-th LRI value is defined in the following way:

$$L_m[P_0] = \sum_{n=1}^{N} \psi_n \alpha_{n,m}[P_0], \tag{2.25}$$

where $L_m[P_0]$ is the value of pixel P_0 of the m-th LRI.

2.5.4 HRI Compounding

Assuming that a full sweep of the transducer array corresponds to M firings, let's define the compounding of an HRI as:

$$H[P_0] = \sum_{m=1}^{M} L_m[P_0], \tag{2.26}$$

where $H[P_0]$ corresponds to the value of pixel P_0. When a new LRI is computed, the oldest LRI in the compounding frame is discarded, and the new one takes its place. Thus, it is possible to convert Eq. 2.26 into a more efficient [17]:

$$H_i[P_0] = H_{i-1}[P_0] + L_i[P_0] - L_{i-M}[P_0]. \tag{2.27}$$

The process of compounding several LRIs to form an HRI corresponds to performing transmit beamforming, allowing for a more general focusing of the ROI, and improving the lateral resolution of the final image.

2.6 SA Beamforming vs. B-scan techniques

After conveying the reader with the working principles of both techniques, it is now time to discuss the advantages/disadvantages of the SA technique in comparison to the beamline-based technique:

<table>
<tr><th>Advantages</th><th>Disadvantages</th></tr>
<tr><td>Ultra-High Frame-Rate Capability</td><td>Higher Computational Complexity</td></tr>
<tr><td>Transmit Beamforming Capability</td><td rowspan="2">Current Hardware is not ready to implement SA beamforming techniques</td></tr>
<tr><td>Improved Overall Image Quality</td></tr>
<tr><td>Reduced Image Artefacts Appearance</td><td></td></tr>
</table>

It would easily come to the mind of the reader that the preferred option should be the SA Beamforming technique, given its heightened image quality and frame-rate performance, but the disadvantages should also be taken seriously, since the computational workload required to implement the technique is significantly higher than that of typical B-scan techniques.

2.7 Conclusions

Concluding this chapter, it is expected that the readers now understand the principles behind beamline-based B-scan and SA Beamforming techniques, and can perform an assessment of the requirements and challenges faced by ultrasound systems engineers.
It is the author's belief that SA Beamforming represents the future of medical ultrasound imaging systems, and that a parallel approach to the signal processing routine discussed in 2.5.2 and the use of programmable multi-core devices are the keys to address the disadvantages exposed in 2.6 and therefore, to achieve a working system, that can run on the standard personal computer or notebook, with improved image quality and real-time capability.

3

Parallel Computing Frameworks and Manycore Devices

Contents

3.1 The Open Computing Language (OpenCL) Parallel Programming Model 24

3.2 The multicore CPU 30

3.3 The manycore GPU 34

3.4 Conclusion 39

The rise and evolution of semiconductor technology over the last decades provided computing devices, such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs) with growing computing capabilities. In recent years, physical limitations, especially tied to the core clock speeds and memory wall, implied that the scale of CPU cores could not increase, and thus the advent of multi-core CPUs became alternative. The initial role of the GPU was to perform multimedia tasks (video playback, graphics rendering) on the computer. The parallel nature of these devices, in conjunction with their multiple cores, and the development of programming constructs and interfaces lead to the rise of parallel programming, which allowed raising the throughput performance peak in other areas of general purpose computing. Since then, scientists in many fields have been employing the various parallel computing frameworks to implement compute-intensive algorithms.
In this chapter, the OpenCL framework is also discussed, as well as a brief discussion of current CPU and GPU architectures, emphasizing the latter, exposing through the section the main differences between Advanced Micro Devices (AMD) and NVIDIA GPUs.

3.1 The OpenCL Parallel Programming Model

OpenCL birthed from the industry's effort to design a general purpose parallel computing open standard. The OpenCL Application Programming Interface (API) currently supports both C and Fortran programming languages. This effort results in a wide range of compatible multi-core devices available in the market. Although CPUs are able to run OpenCL kernels, the framework is optimized for GPUs.
A single kernel code, running efficiently on several devices is one of the main goals of OpenCL, therefore, there is a low-level of hardware abstraction.

In this section, for illustrative purposes the figures will be related to a sample program whose objective is to perform the parallel summation of two single floating-point matrices (A and B) and store the result in matrix C. The computing workload associated with this algorithm is not very high, but it works perfectly as a demonstrative program.

3.1.1 Platform Model

In an OpenCL program, there must be distinction between the host code and the device code. The first will query the available parallel platforms and respective devices, allocate device memory, and overall perform flow control. The latter is the parallel section of the code, the kernel itself, and runs on the queried devices.
The mere presence of an OpenCL supported device in the machine is not sufficient to guarantee that the device is able to run OpenCL kernels. In order to achieve that, the

vendor's OpenCL Software Development Kit (SDK) must be installed in the machine. Fig. 3.1 illustrates the process of querying the machine for OpenCL supported devices:

```
//OpenCL Device Query Code
char pbuff[100]; //Char Buffer to hold the name of the devices
cl_int err; //Error Controlling Variable
cl_uint numPlatforms;
cl_uint numDevices;

//First call to get the number of available platforms
err = clGetPlatformIDs(0,NULL,&numPlatforms);
//Allocate space for all the available platforms
cl_platform_id* platforms = (cl_platform_id *)malloc(numPlatforms*sizeof(cl_platform_id));
//Query the platforms
err = clGetPlatformIDs(numPlatforms,platforms,NULL);
//Choose the first (Assuming error-free execution, for space constraint reasons)
cl_platform_id cPlatform = platforms[0];

//For the chosen platform, query the number of available devices
err = clGetDeviceIDs(cPlatform,CL_DEVICE_TYPE_ALL,0,NULL,&numDevices);
//Allocate space for all the available devices
cl_device_id *devices = (cl_device_id *)malloc(numDevices*sizeof(cl_device_id));
//Query the devices
err = clGetDeviceIDs(cPlatform,CL_DEVICE_TYPE_ALL,numDevices, devices, NULL);

for(int i=0; i<numDevices; i++)
{
//Get the info of each device, and print it iteratively
err = clGetDeviceInfo(devices[i], CL_DEVICE_NAME, sizeof(pbuff), pbuff, NULL);
printf("\%s\setminus n",pbuff);
}

cl_device cDevice = devices[0];
cl_context cContext = clCreateContext(0, 1, &cDevice, NULL, NULL, &err);
/*...*/

//Free the allocated memory
free(platforms);
free(devices);
```

Figure 3.1: OpenCL Device Query.

3.1.2 Runtime Model

The fact that OpenCL can be run in different types of devices, adds a new layer of complexity to the programming process. While NVIDIA only provides OpenCL support for their GPUs, Intel provides support for proprietary devices (CPUs and GPUs), AMD provides support for all devices except NVIDIA and Intel integrated GPUs. To generate a kernel executable, each device may require different binaries. For that reason, and because

the device query and selection is only performed at runtime, the building, compiling, linkage and executable generation of the *kernel* can only occur during runtime.
Generally OpenCL *kernels* are either stored in a separate file (the common extension is **.cl**) or directly stored in a string in the main C/C++ file. [19] [20]

3.1.2.A *Command-queues*

All OpenCL objects, either being *devices*, *platforms*, *queues*, or *programs* share the fact that all operations performed on them must be enqueued in a *command-queue*. This *command-queue* is bound to a specific *device*, and by default the operations are performed in-order, and synchronization is implicitly guaranteed. It is possible to alter this behaviour, allowing for out-of-order queue execution. However, the synchronization between commands must be ensured by using *events*. This is required, for example, when implementing overlapping the memory transfers and *kernel* executions (see Sec. 3.1.2.D). Fig. 3.2 illustrates the creation of a *command-queue* with default options (the third argument is empty).

```
/* ... */
cl_command_queue cQueue = clCreateCommandQueue(cContext, cDevice, NULL, &err);

/* ... */
```

Figure 3.2: OpenCL Queue Creation.

3.1.2.B Memory Model and Allocation

Given the interoperability between different devices, even when running the OpenCL kernels on CPUs, there must be separation between host memory and device memory. To run an OpenCL *kernel*, the programmer must ensure that the required input and output *device buffers* are allocated, and properly data-filled before the *kernel* execution can occur. The fact that OpenCL is supposed to run on a multitude of *devices*, introduces a new level of complexity, since different *devices* have different memory regions (more on this on Sec. 3.3.2) and sizes, and thus, the programmer must also imbue the program with the necessary conditions to attain resource adaptability.

There are two ways to perform memory transfers: either synchronously by queueing the transfer, or asynchronously by mapping the buffer with pointers in the *device/host*. Fig. 3.3 shows the creation of the synchronous and asynchronous *buffers*.

```
/* ... */
float array[5] = {1.0,2.1,3.2,4.3,5.4} //input data to be transferred to the buffers

cl_mem cBuffer_1, cBuffer_2;
//cBuffer_1 will be created with queueable data transfers
cBuffer_1 = clCreateBuffer(cContext,CL_MEM_READ_ONLY,sizeof(cl_float)*5,NULL,&err);
err = clEnqueueWriteBuffer(cQueue,cBuffer_1,CL_FALSE,0,sizeof(cl_float)*5,array,NULL, ...
... NULL,NULL);

//cBuffer_2 will be created for mapping in the device
cBuffer_2 = clCreateBuffer(cContext,CL_MEM_READ_ONLY | CL_USE_HOST_PTR, ...
... sizeof(cl_float)*5, NULL,&err);
clEnqueueMapBuffer(cQueue,cBuffer_2,CL_FALSE,CL_MAP_READ,0,sizeof(float)*5,NULL,NULL, ...
... NULL,&err);

/* ... */
```

Figure 3.3: OpenCL Buffer Creation and Mapping/Enqueuing.

After execution, the output buffer is read from the device.

OpenCL defines several address spaces, where the buffers can be allocated or mapped, according to the desired usage and requirements. These are *__global*, *__local*, *__private* and *__constant*. The *__global* is accessible to all work-items in the context, and it is also the largest (and slowest) address space. *__local* is specific to each work-group, and thus only the work-items of a given work-group can read/write from/to this memory region. *__constant* is a read-only memory region, accessible by every work-item, faster than global memory, and ideal for variables initialized in the host, that won't suffer changes throughout the kernel execution. Finally, *__private* is specific to each work-item. It is excellent to store temporary data, as it is the fastest type of memory.

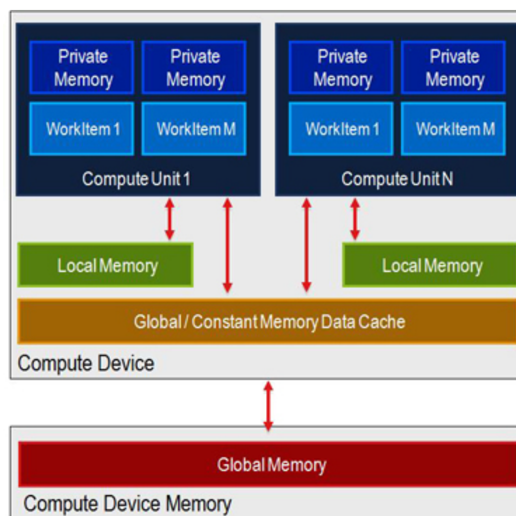


Figure 3.4: OpenCL Address Space. [2]

3.1.2.C Kernel Work Size and Execution

A parallel implementation of a serial algorithm consists of the division of the work into small work units. For the example summation program considered, each work unit corresponds to an element of the buffer. In OpenCL, each of these small units is called *work-item*. A *work-group* is a group of *work-items*. Depending on the nature of the algorithm and the *device* capabilities, the *work-group* might have 1, 2 or even 3 dimensions. The optimal solution regarding the number of *work-items* per *work-group* must be determined by experiment (although OpenCL profilers use a few metrics to determine the point of "optimal" performance, the proposed solutions often return non-optimal results). The size of a *work-group* is called the *local work size*. The total number of *work-items* required by the algorithm determines the *global work size*, whose value must always be a multiple of the *local work size* (if more than one dimension is used, all the dimensions must be multiples).

After the input *buffers* are filled, and the work dimensions are set, the *kernel* is enqueued and executed.

3.1.2.D Optimization Strategies

The main goal of parallel computing consists of using the resources in an efficient way. Therefore, when designing an OpenCL *kernel*, there are a few rules that can significantly improve the program's performance:

- Thread Divergence: A wavefront is the execution of *N* OpenCL *work-items* in parallel, on AMD GPUs. On NVIDIA GPUs, while the exact concept is not replicated, because of the different nature of the architecture, the basic principle applies, but it is called a warp. When the *kernel* is being executed, consecutive *work-items* are grouped for execution. When a divergent condition is present (such as an if statement), the wavefront/warp will have to check both paths of execution, resulting in an additional cycle. In the limit all threads/work-items can follow a different path and execution serializes. Therefore, whenever possible, divergent branches should be eliminated.

- Coalesced Memory Accesses: Most current computers are equipped with dual channel Random Access Memory (RAM), that ensure that the width of the memory bus to the CPU is 128-bit. Similarly, the Peripheral Component Interconnect Express (PCIe) 3.0 currently used by most mid-level GPUs also feature a bus width of 128-bit. When the *kernel* is running, and the work-items in a warp/wavefront perform a memory load, the device is smart enough to detect if the addresses being read by consecutive work-items are consecutive or separated by a constant stride multiple of 16 or 32. When such event occurs, such as in the matrix summation sample, the program coalesces multiple memory accesses into

a single access, thus largely reducing the number of memory operations/transactions.

- Use of registers: The number of private registers (See 3.3.2) depends on the device, but they can be used to store temporary data, or to reduce the number of accesses to the global memory.

- Asynchronous Memory Transfers: Frequently, the main performance bottleneck in OpenCL applications are the memory transfers. By using two *queues* bound to the same device, additional buffers and events to guarantee synchronization between queues, it is possible to overlap memory transfers with kernel execution, increasing the overall throughput of the program.

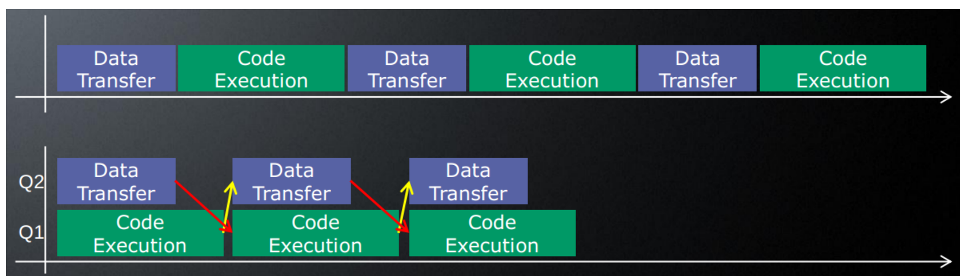


Figure 3.5: OpenCL Kernel Execution/Memory Operations Overlapping. [3]

With this exposition, the reader is expected to better understand the advantages and potential behind the use of parallel programming, particularly with the OpenCL framework, allowing for one code to run on multiple devices, crossing different architectures, different vendors and even different types of devices.

3.2 The multicore CPU

In 1965, Gordon Moore predicted that, until 1975, every two years the number of transistors per integrated circuit would double [21]. Little did he know back then, that his predictions were not only spot-on for that span of time, but it also set objectives for the computing hardware and semiconductor industry, and these goals are still being sought as of today.
CPUs are just ”victim” of this growth, and if in 1980, the Intel 80186 featured 29000 transistors, in 1993 the original Pentium processor already had more than 3 million transistors. To further increase the performance of CPUs to meet the ever-increasing industry needs, the clock speed of the transistors also increased. But physical limitations regarding heat dissipation, and the fact that the growth in clock speed had stalled, led to the pursuit of new ways to increase the performance. Thus, the rise of the multicore CPU era. Multicore CPUs provide parallel resources, and the power constraints are not so constrictive, making these the viable solution, to keep supplying the users with performance increments. Fig. 3.6 and Fig. 3.7 show the die of the Haswell and Piledriver architectures from Intel and AMD, respectively.

Let us now discuss the general architecture of multicore CPUs, starting with the Control Unit:

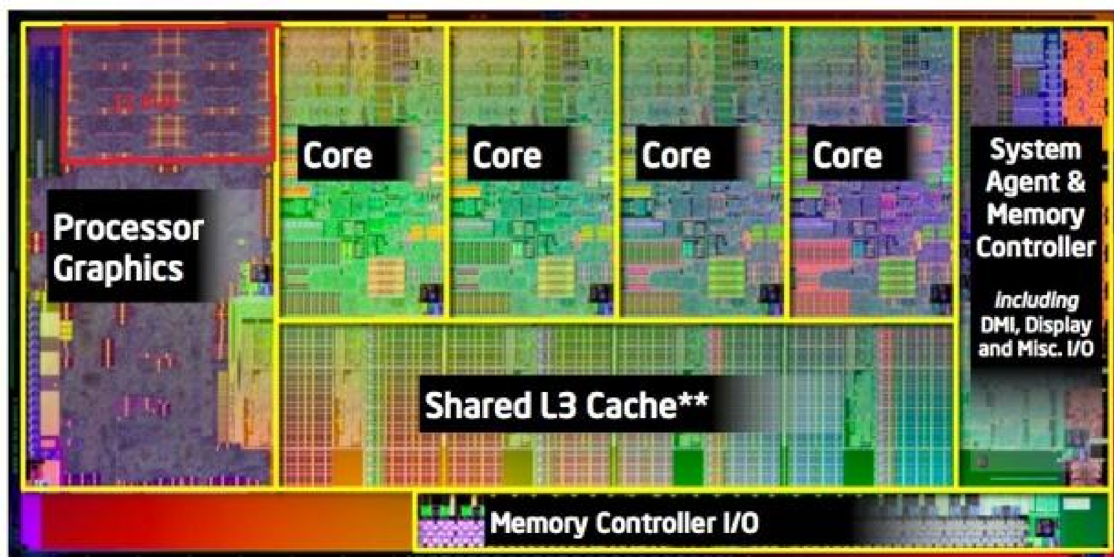


Figure 3.6: Intel Haswell architecture die. [4]

3.2.1 Control Unit

The control unit of a core, as its name says, corresponds to the group of circuits in the chip that is responsible for controlling the fetching of instructions from memory, decoding the instruction and memory addresses of the operands for the operation, redirects the data to the arithmetic and logic units, so that the desired operation may be performed. It also controls the flow of data between all the devices in the computer. The instruction set of the CPU is also implemented in the control unit.

3.2.2 Arithmetic and Logic Unit (ALU)

The ALU is the group of logic circuitry present in each core that is responsible for performing every arithmetic operations like additions, subtractions, multiplications, bit shifting, logic operations such as AND, OR, NOT and boolean comparisons. Operations performed by ALUs are the basis for more complex instructions. Current CPUs' ALUs provide support for floating-point operations, which leads to the understandable conclu-

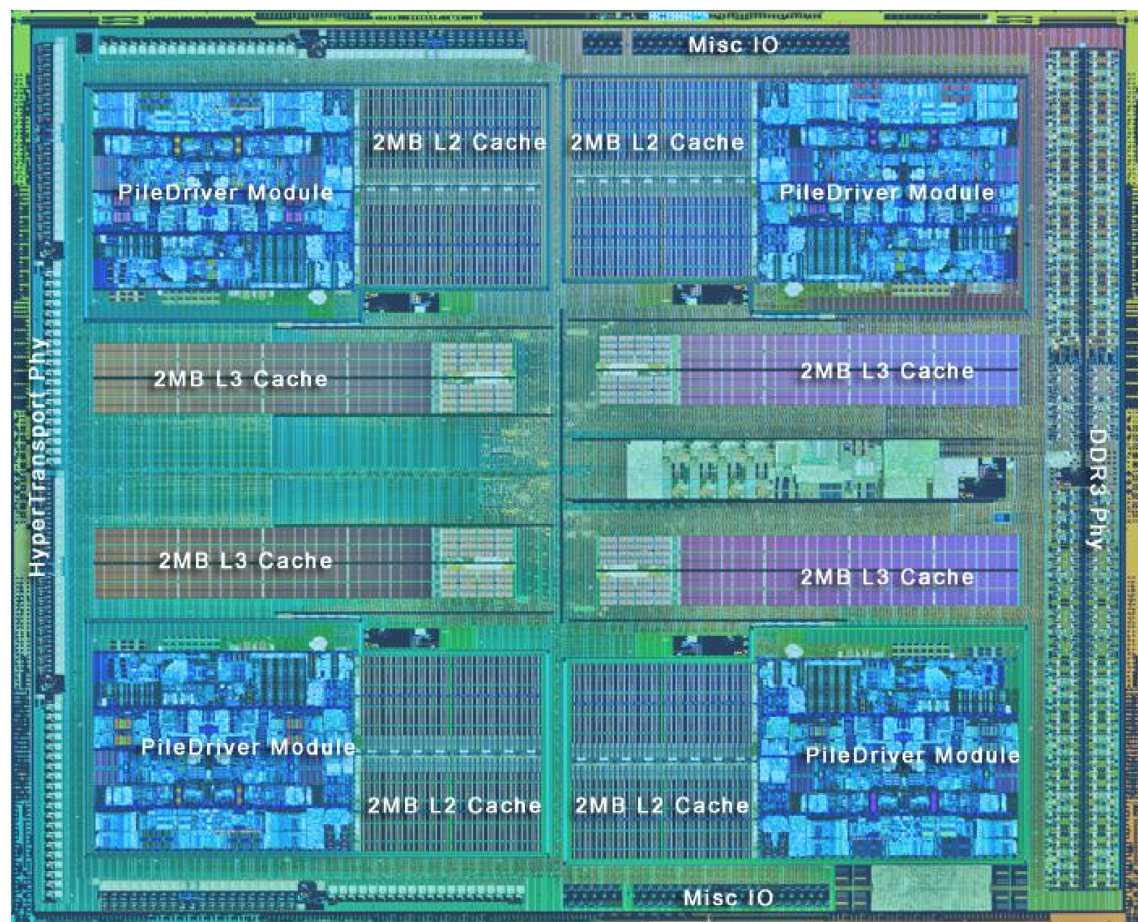


Figure 3.7: AMD Piledriver architecture die. [5]

sion that, nowadays, ALUs are components of very complex design, and this design is different between different CPUs models.

3.2.3 Memory Interface and Caches

The system memory is basically comprised of three types of memory: Dynamic Random Access Memory (DRAM), Read Only Memory (ROM) and cache memory. While the caches are a feature of the CPU, DRAM and ROM are not located in the chip itself, as they are placed on the motherboard. Here, an appointment must be made regarding a major difference between the two main CPU vendors, Intel and AMD. Until recent years, Intel CPUs communicated with the DRAM and ROM through the motherboard memory controller, the Front-Side Bus (FSB). But recently, Intel improved its system, implementing the QuickPath InterConnection technology. This technology, as all AMD CPUs, features native memory controllers. For the sake of retro compatibility, since there are still many computers using FSB, its principles are briefly discussed below.

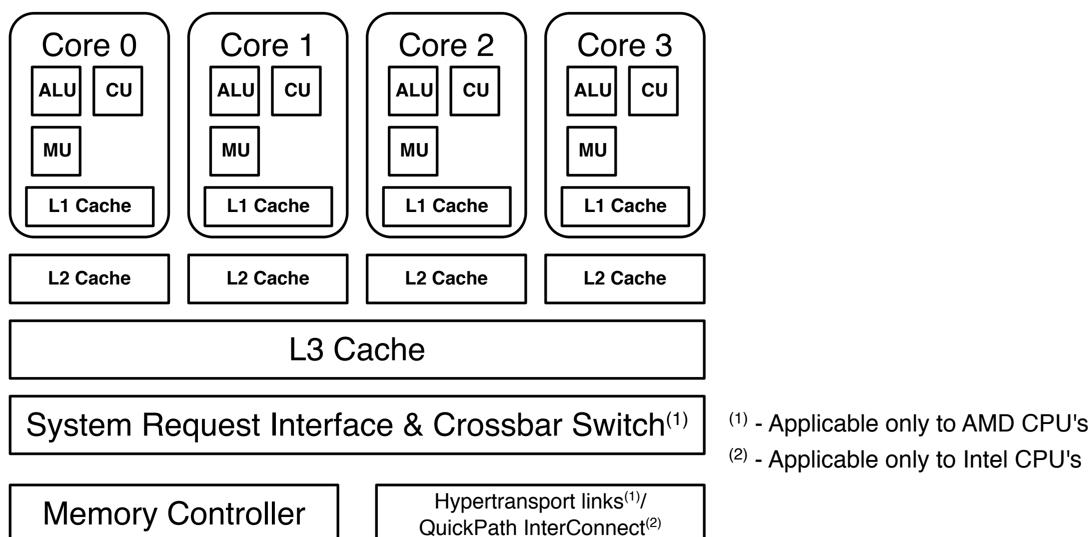


Figure 3.8: Current generation quadcore CPU basic diagram. Intel and AMD CPUs differences are highlighted.

3.2.3.A DRAM and ROM

DRAM memory is generally the main region of the system. When running a program, the program's instructions and data are loaded into the DRAM, and then loaded as they are needed by the CPU. Afterwards, if needs be, the CPU can also store data in DRAM. ROM, on the contrary, is read-only, so the CPU can only fetch data from the

memory. Another interesting difference between the two memory regions is that ROM is not volatile.

ROM memory is filled by default with special instruction sets that are important to load the Operating System (OS) when booting the machine.

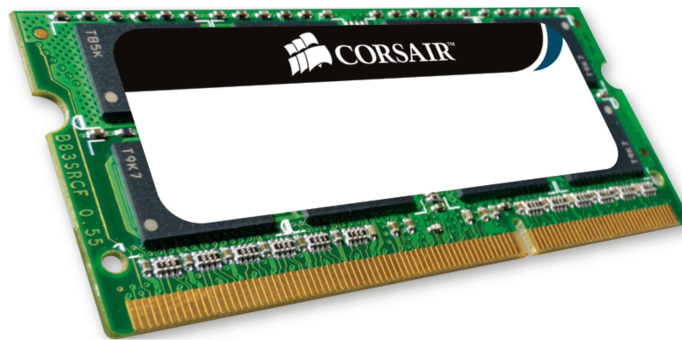


Figure 3.9: RAM stick from Corsair. [6]

Regarding the FSB referenced above, its most important characteristic is its speed, as it determines the maximum bandwidth that memory transfers between the motherboard and the Intel CPU occur. Meanwhile, in recent years, some Intel CPUs have already dropped the FSB, in favour of the QuickPath InterConnect technology, providing these CPUs with a memory interface very similar to that of AMD's. In these systems, the maximum achievable bandwidth is limited only by the specifications of the RAM modules.

3.2.3.B Caches

The cache is a hardware block of fast memory placed in the CPU chip, whose main objective (as with all other caches) is to mask the memory latency induced by accesses to the main memory. To achieve that, the cache stores accessed data and instructions, based on the premise that it is likely that this set of data will be used again. Therefore, instead of loading it again from RAM, the CPU simply fetches it from the cache, whose latency is significantly lower than that of RAM memory accesses. There are currently three levels of cache: $L1$, $L2$ and $L3$. The $L1$ cache is the fastest cache. It resides on-chip, and consists on a large group of flip-flops etched into the die. It is classified as an Static Random Access Memory (SRAM) memory. Due to space limitations, the size of the $L1$ cache is small. Current systems only have $64kB$. This limited space led to the use of another cache level, specifically $L2$. In earlier systems, the $L2$ cache was placed directly in the motherboard or as "cache sticks", but recent advances have seen the $L2$ placed in the CPU. In current multicore processors, one of two choices is possible: either the $L2$ cache is shared between cores (with one for all, or one for a pair of cores), or each core

has its own $L2$ cache. Again, $L2$ is also considered SRAM, but its slower than $L1$. Typical $L2$ caches have $256kB$. Finally, the $L3$ cache is bigger than the $L2$, but also slower and resides in the motherboard. Current CPUs feature $L3$ caches sized $6MB$.

3.3 The manycore GPU

In the first years of the 21st century, when the CPU processing power improvement started to saturate, GPU appeared as a good alternative to provide additional processing power, by way of new programming models and parallel implementations of algorithms whose nature allowed such approach. Image processing algorithms in particular, can be highly parallelizable. In the following section, the basic architecture of the GPU is discussed, and the differences between the two main vendors - NVIDIA and AMD are highlighted. The GPU improvement between generations is commandeered by the computer gaming industry. The competition between different companies to fulfil the client's wish for realistic games, has pushed GPUs even further.

3.3.1 Streaming Processors (SPs)

The basic processing unit in a GPU is a SP. A few years ago, NVIDIA created the designation CUDA core. AMD created its own designation: the Thread Processor (TP). Both can be regarded as single processing units. While the CUDA core is comprised of a single SP, the TP features five SP (four capable of performing simple operations, one Special Function Unit (SFU) and one Branch Unit (BU). Of course, they operate in different ways. When performing scalar operations, only one SP in the TP is being used, while the other four are idle. The only exception to this behaviour, is when four SP can be used at the same time, as long as the instruction is issued in a Very Long Instruction Word (VLIW) format. When this happens, the TP performs simple arithmetic operations over scalar types (such as $int4$, $float4$), and thus features a better performance than its NVIDIA counterpart. In the CUDA core, the only SP is obviously used.

In the latest AMD architecture, a group of 16 TP is called a Single Instruction Multiple Data (SIMD) engine. When 4 SIMD engines are grouped, a Compute Unit (CU) is obtained. Its NVIDIA equivalent (a group of CUDA cores) is regarded as a Stream Multiprocessor (SMX). A Kepler GPU's SMX packs 192 CUDA cores.

Regarding OpenCL kernel execution, both vendors follow a Single Instruction Multiple Thread (SIMT) policy. NVIDIA introduced the warp, where a single instruction is executed in 32 threads in parallel. AMD GPUs consider wavefronts, which are groups of

64 parallel threads.

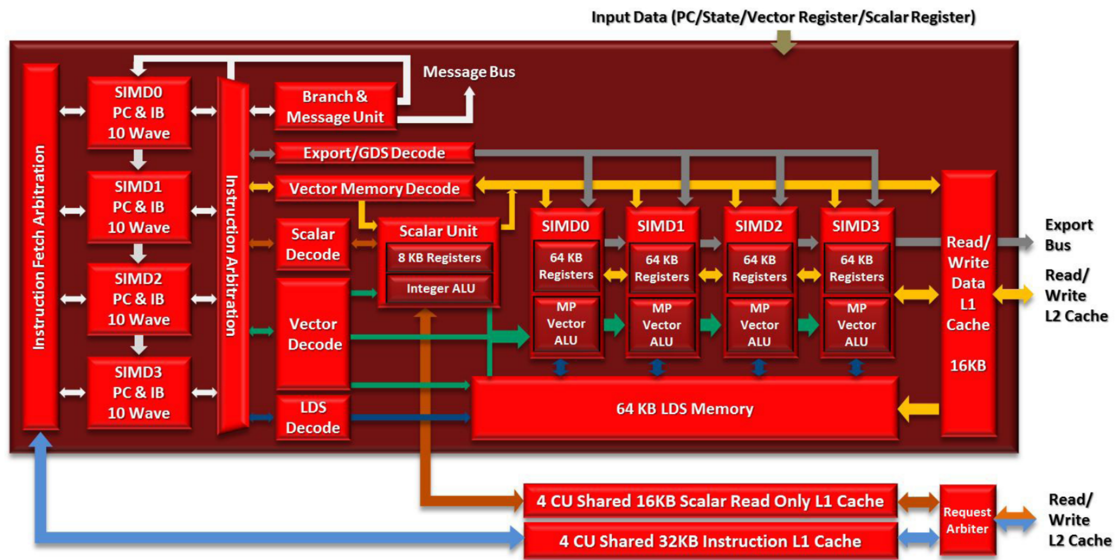


Figure 3.10: GCN Compute Unit Architecture. [7]

3.3.2 Memory-Hierarchy

The memory architecture differs greatly for AMD and NVIDIA GPUs, and also between different device generations.

In the new AMD generation of GPUs, the Graphic Core Next (GCN) architecture, each SIMD engine is endowed with $64kB$ of private registers. In a given CU, the 4 SIMD engines share an $L1$ data cache, of size $16kB$. Also present is the Local Data Shared Memory (LDS), which can either be accessed directly by the SIMD engines, or load data into the $L1$ cache, when this is deemed advantageous. The LDS block is usually $64kB$ long. Additionally, the CU has two auxiliary shared $L1$ caches: the scalar read-only and the instruction, with sizes $16kB$ and $32kB$, respectively. These are also shared between the 4 SIMD engines [7]. Since scalar operations are commonly used for flow control purposes, there is little need to write the results back into the cache, therefore the decision for a read-only cache. Regarding the instruction cache, there is little more to say about it, as it stores frequently used instructions to mask global memory latency. Fig. 3.10 illustrates this architecture.

Shared by all the CUs are multiple instances of $L2$ cache (one for each memory channel), each of them sized between $64kB$ and $128kB$. The role of the $L2$ cache is to keep the coherency of the application, by grouping the data from every CU. The fact that it is lo-

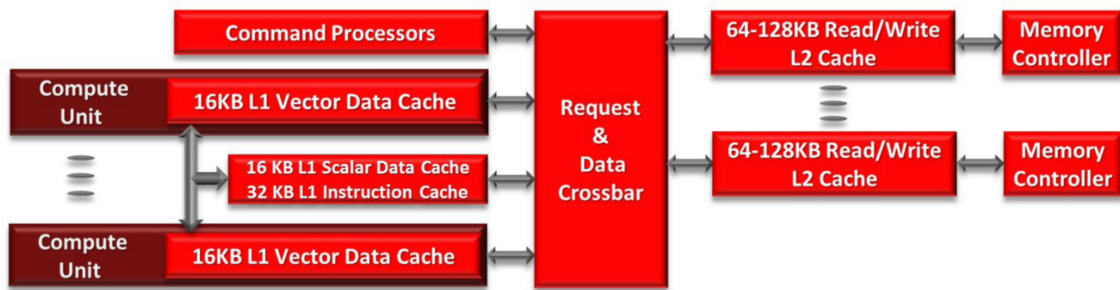


Figure 3.11: GCN Basic Memory Model. [7]

cated on-chip is a great advantage, since it is significantly faster than the previously used off-chip memory. As evidenced in Fig. 3.11, the CU and respective caches communicate with the $L2$ slices through a switching ”cross-bar” fabric.

The current generation of NVIDIA GPUs are based on a refresh of the Kepler architecture. In this architecture, each SMX has $64kB$ of on-chip memory, that can be configured as $48kB$ of shared memory and $16kB$ of $L1$ cache, or the other way around, to best suit the applications' needs. Similar to AMD, NVIDIA GPUs also feature a read-only $L1$ data cache, but unlike its competitor, NVIDIA's cache is $48kB$ long [8]. The $L2$ cache again acts as a coherency point, and the big difference to AMD is the fact that here, this cache is a single $1536kB$ block.

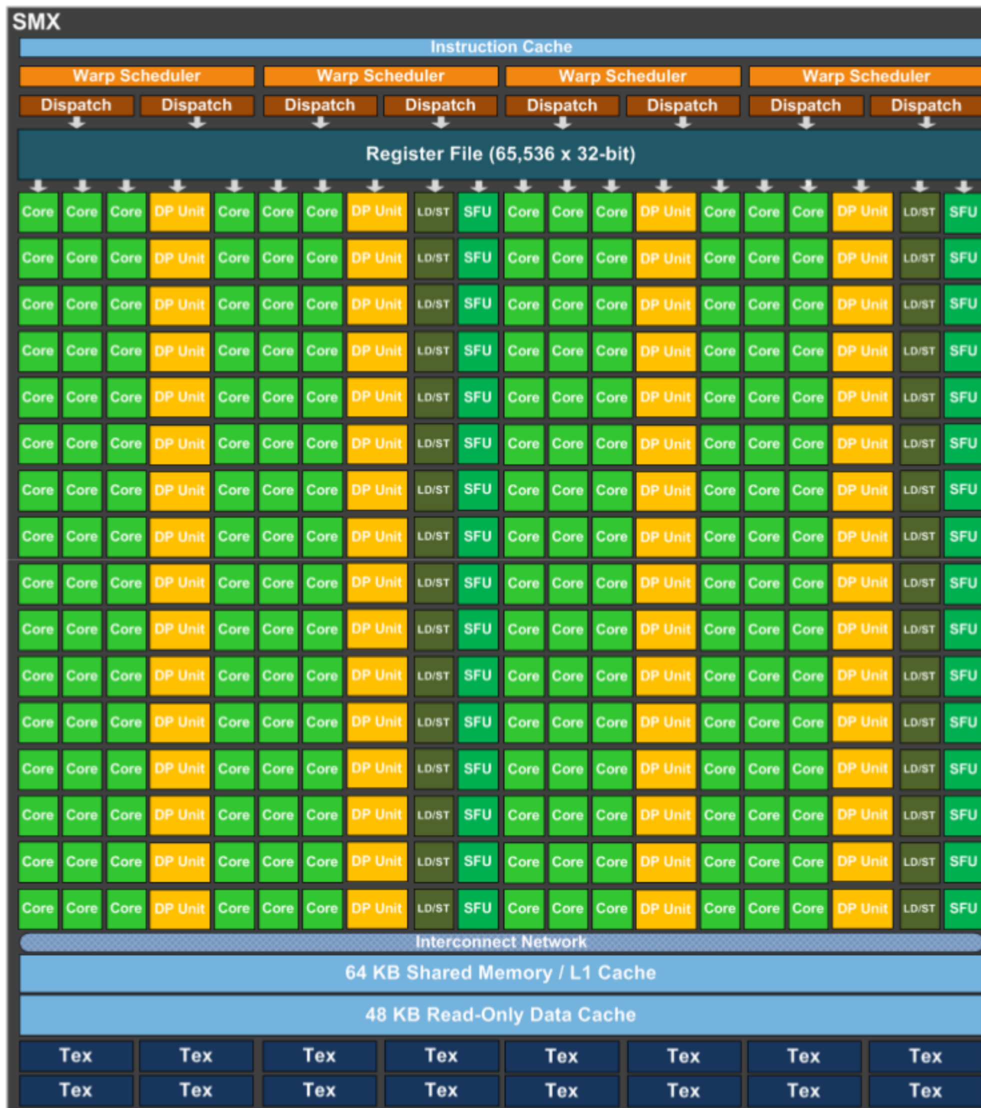


Figure 3.12: Kepler SMX model. [8]

Figure 3.13: Kepler Generic Architecture. [8]

3.3.3 Mapping the GPUs Memory Regions to the OpenCL Address Spaces

As discussed in 3.1.2.B, OpenCL defines four address spaces, each of which has its own advantages and disadvantages. Considering their characteristics, it is easy to map these address spaces to specific memory regions in the current generation of GPUs from AMD (GCN) and NVIDIA (Kepler), as exposed in the following table:

	AMD	**NVIDIA**
__global	$L2$ Cache	$L2$ Cache
__local	LDS	Shared Memory
__constant	Read-Only Data $L1$ Cache	Read-Only Data $L1$ Cache
__private	General-Purpose Registers (GPRs) ($256kB/CU$)	GPRs ($256kB/SMX$)

3.4 Conclusion

This chapter conveys the reader with a basic knowledge of OpenCL's inner workings and principles, and the main families of multicore and manycore devices, namely CPUs and GPUs, upon which the framework can be used. The reader is now able to assess the advantages and disadvantages of the powerful programming model that OpenCL represents, as well as to identify some of its potential applications.Under the context of this work, the next chapters show how Synthetic Aperture (SA) Beamforming Imaging kernels can be developed in order to conveniently exploit the computational power that such parallel architectures can offer.

4

Parallel Synthetic Aperture Beamforming on Manycore Devices

Contents

4.1 Outlining 42
4.2 Experimental Results 48
4.3 Analyzing the obtained results 50
4.4 Conclusions 54

During the course of the following chapter, the simulation scenario and performance metrics are outlined, followed by the description and analysis of the results obtained.

4.1 Outlining

This section describes how the delay-and-sum approach of the Synthetic Aperture (SA) beamforming technique will be implemented in the Graphics Processing Unit (GPU). This is followed up by a discussion of the platform used for the simulations and all the metrics used for performance evaluation purposes.

4.1.1 SA Beamforming Kernels

To design an Open Computing Language (OpenCL) application, it is necessary to first discuss which steps to parallelize. Reaching back to Chapter 2, it is clear that before performing the image reconstruction step, the signal must undergo an additional filtering step (Hilbert transform FIR filter), so as to compute the analytic representation. This is illustrated in the Fig. 4.1.

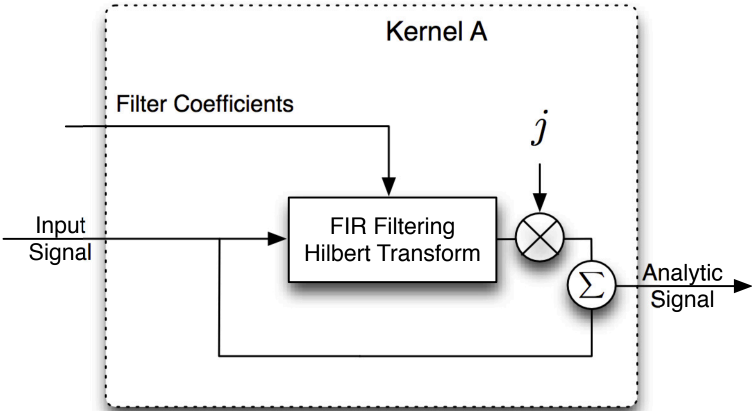
After computing the analytic representation of the acquired signal, the image reconstruction algorithm is now performed. This algorithm is the main focus of this work, since it features the biggest computing complexity in the whole process. Fig. 4.2 shows how the algorithm is mapped in the GPU.

During the course of this project, two parallel approaches to the imaging algorithm were followed. They are both illustrated in Fig. 4.3. In the 1st approach, the kernels are launched independently, meaning that when operating with multiple GPUs, each will be assigned one of the kernels, and their execution is sequential. With the use of pthreads, the 2nd approach introduces a new level of parallelism, and each GPU will execute the two kernels sequentially, while the other is concurrently performing the same tasks on another set of data. A direct consequence of this, is that there is need for strict synchronization between threads to prevent calculations on already processed data, and wrong increments on shared variables.

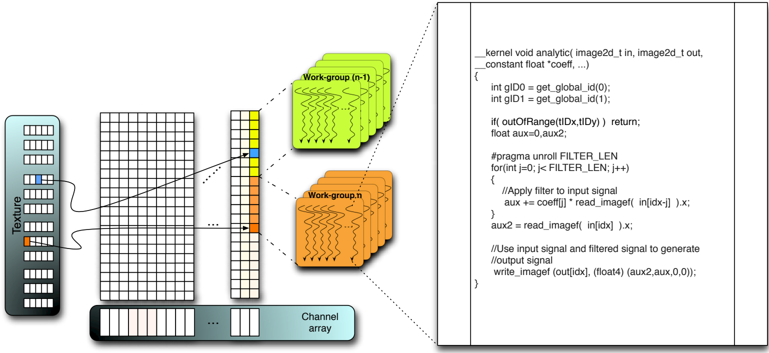
The experimental results were obtained for four versions of the application, based on the approaches outlined in Fig. 4.3. Both v1.0 and v1.1 are based on Approach 1. Similarly, v2.0 and v2.1 are based on Approach 2.

v1.0

- Full C console. Kernel A and B are both executed on GPUs.



(a)



(b)

Figure 4.1: Kernel A block diagram and algorithm structure on the GPU.

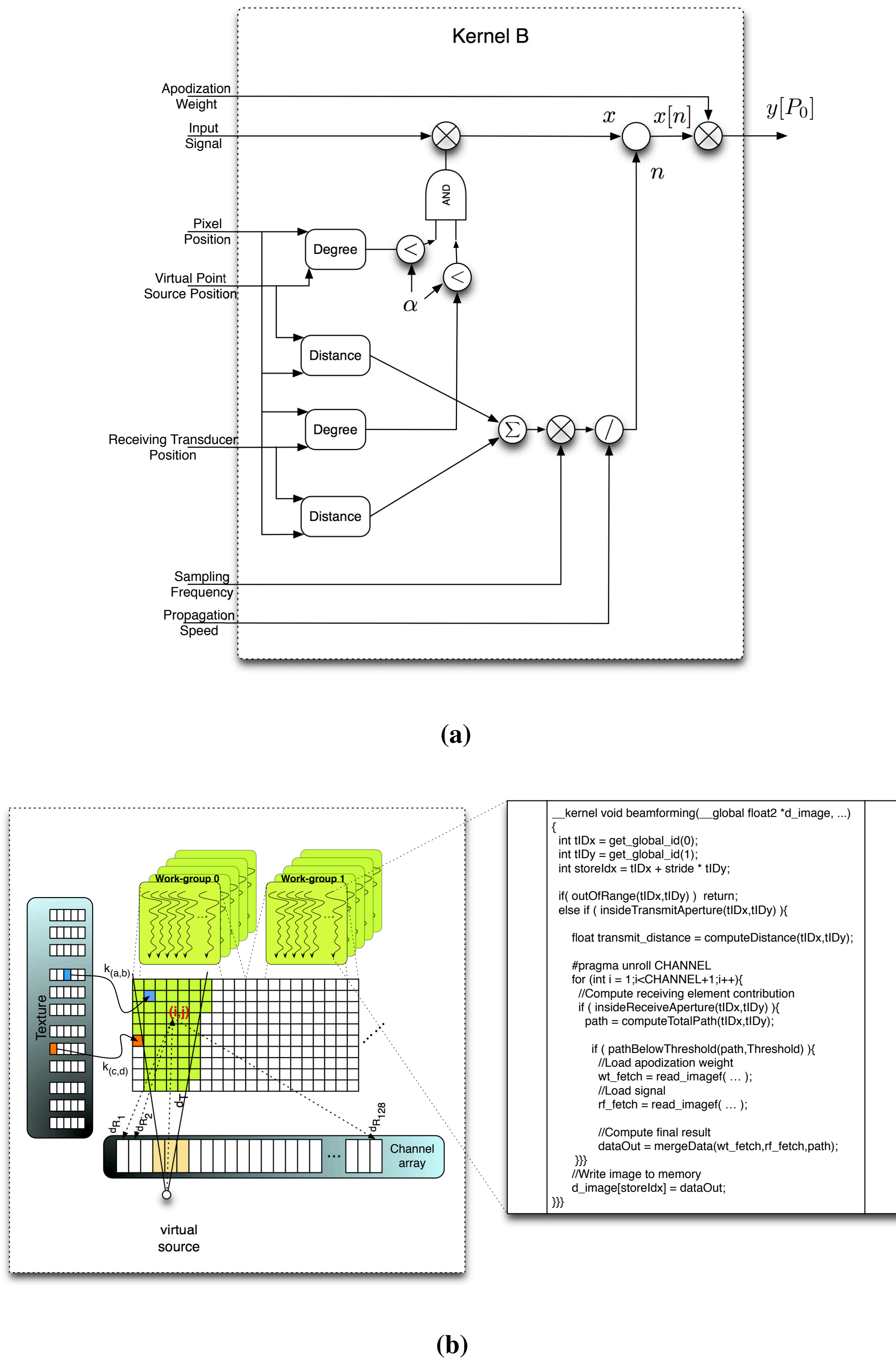
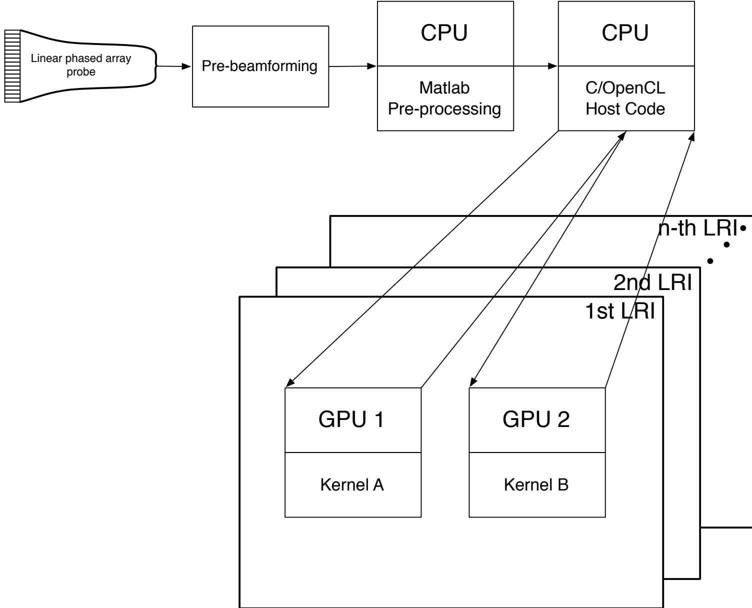
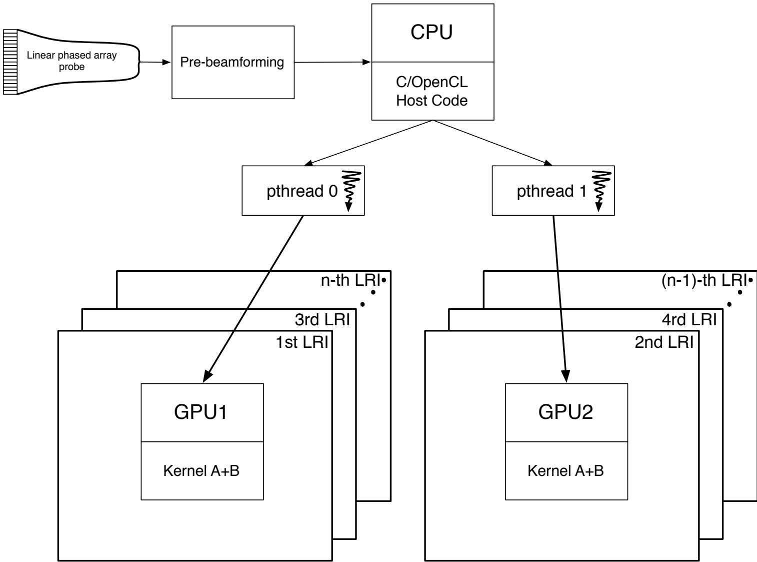


Figure 4.2: Kernel B block diagram and algorithm structure on the GPU.



(a) Approach 1: Both GPUs compute the LRI cooperatively. The LRI computation is sequential.



(b) Approach 2: LRIs are computed in parallel, with the use of pthreads. Each GPU is assigned an LRI for computation

Figure 4.3: Structural outline of the two parallel approaches followed in this work.

v1.1

- Optimized host code.

v2.0

- Memory transfers are blocking, meaning that when enqueued, the host code is blocked until execution is completed.

v2.1

- Vector types (cl_float2, cl_float4) used both for memory transfer and for vectorial arithmetic operations.
- Two command queues for the same device: one for memory transfers, the other for kernel calls.
- Memory API calls are non-blocking (the host does not wait for the enqueued command to be completed before it advances).

Due to the fact that the different GPUs have different resources and capabilities, some scenarios can not handle the resource requirements introduced by all the versions. When this occurs, the corresponding cells in the tables are filled with Non-Applicable (N/A).

4.1.2 Simulation Apparatus

The experimental apparatus used in this work consists on a Asus P6X58D Premium. It features an Intel Core i7 950 Central Processing Unit (CPU), with $8MB$ of $L3$ cache, and clocked at $3.07GHz$. The machine is also equiped with $2x3GB$ of DDR3 Random Access Memory (RAM), clocked at $1600MHz$. During the course of the simulations, several GPUs are used, namely two Advanced Micro Devices (AMD) Radeon HD6970 (24 Compute Unit (CU) and $2GB$ GDDR5 memory), an NVIDIA Tesla C1060 (30 Stream Multiprocessor (SMX) and $4GB$ GDDR3 memory) and one NVIDIA GTX680 (8 SMX and $2GB$ GDDR5 memory). The machine runs on Windows 7 Ultimate x64, and the project was created under Microsoft Visual Studio 2010 Ultimate. OpenCL support was added with the installation of the AMD and NVIDIA SDK's for OpenCL. Also, in order to perform application profiling, AMD CodeXL and NVIDIA Visual Profiler were used, providing valuable information regarding kernel and memory metrics. The simulation consists on a set of bundles of raw data, acquired using a Sonix-RP research scanner equipped with a pre-beamformer data acquisition tool [17]. The 2-cycle $10MHz$ sinusoid pulse is repeated with frequency $5kHz$. There are 97 transmit positions, each comprised of a 64 transducer aperture, with the virtual point sources apart by $0.3mm$. On receiving,

each of the 128 transducers are operating in passive mode, with a sampling frequency of $40MHz$. Each of the 97 data bundles will be processed into an LRI. The group of LRIs will be compounded into an High-Resolution Image (HRI). The Region of Interest (ROI) is comprised of . The reconstructed image is $512x255pixels$, and can be seen in Fig. 4.4.

<table>
<tr><th colspan="3">Host Platform</th></tr>
<tr><td>CPU</td><td colspan="2">Intel Core i7 950</td></tr>
<tr><td>Motherboard</td><td colspan="2">Asus PX58D Premium</td></tr>
<tr><td>Memory</td><td colspan="2">$3 \times 2GB$ DDR3</td></tr>
<tr><td></td><td colspan="2">Multiple GPUs setup</td></tr>
<tr><td rowspan="6">GPUs</td><td>2× AMD Radeon HD6970</td><td></td></tr>
<tr><td>NVIDIA GTX680</td><td>NVIDIA Tesla C1060</td></tr>
<tr><td>AMD Radeon HD6970</td><td>NVIDIA GTX680</td></tr>
<tr><td colspan="2">Single GPU setup</td></tr>
<tr><td colspan="2">AMD Radeon HD6970</td></tr>
<tr><td colspan="2">NVIDIA GTX680</td></tr>
</table>

Table 4.1: Host Platform Specs.

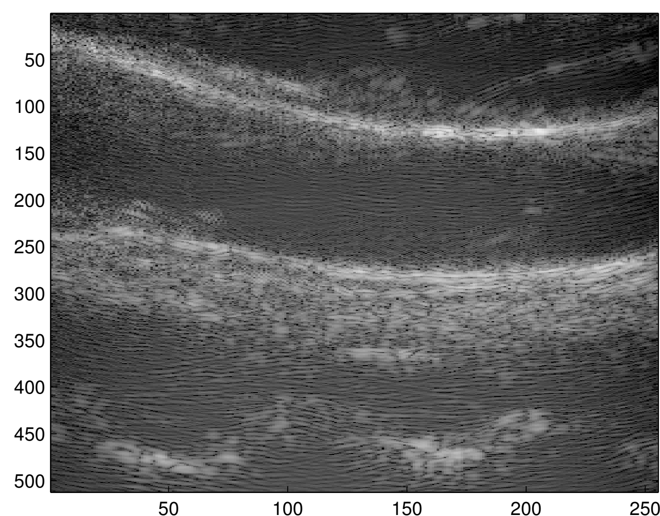


Figure 4.4: Reconstructed image of an in-vivo carotid, computed in the simulations.

4.1.3 Performance Metrics

To evaluate correctly each scenario, they need to be evaluated using different metrics. Let us then define these metrics.

4.1.3.A Throughput and Overall processing Time

Throughput is defined as the relation between the output of a given system and the time it takes for it to be computed. In the context of this work, throughput shall be defined as the number of frames-per-second (FPS) computed under each scenario. Obviously, the throughput is directly related to the overall processing time of the simulation.

4.1.3.B Memory Transfer Rates (Average and Slowest)

Given the considerable amount of data transfers involved in such application, it is only natural that the memory performance of the devices is evaluated. This is achieved by assessing the average read/write transfer rate of the computing devices. The slowest memory transfer is also inspected, since it provides useful information regarding hanging points in the simulation.

4.1.3.C Kernel Occupancy

In parallel computing frameworks such as Compute Unified Device Architecture (CUDA) and OpenCL, kernel occupancy is a very important performance metric. Given the differences between vendors, there are some differences between kernel occupancy in an NVIDIA GPU and an AMD GPU. For NVIDIA devices, kernel occupancy is the ratio between the number of warps each multiprocessor is using concurrently and the maximum number of warps the device is able to handle. In AMD devices, which use wavefronts instead of warps, the ratio is between the number of in-flight wavefronts per CU and the maximum number of wavefronts [22] [23].

The kernel occupancy can be affected by several factors. Either the work group size, the number of registers being used per warp/wavefront, and the amount of local memory used by each workgroup can be limiting factors. With the help of CodeXL and NVIDIA Visual Profiler, the limiting factor(s) can be identified and tweaked to ensure optimal kernel occupancy. Depending on the nature of the algorithm being used, it is possible that the maximum achievable kernel occupancy is below 100%. Nonetheless, although the kernel occupancy ensures optimal utilization of the GPU resources, a higher kernel occupancy does not always ensure higher throughput [24].

4.2 Experimental Results

4.2.1 Single GPU scenario

In this section, the results from the scenarios featuring a single GPU acting as a computing device are explored.

4.2.1.A AMD

In this scenario, a single AMD Radeon HD6970 is used to run the simulation described in Sec.4.1.2.

Source Code Version	Throughput (FPS)	Average Memory Transfer Rate (MTR) (GB/s)
v1.0	0.5532	0.45
v1.1	16.93	0.441
v2.0	98.7	3.03
v2.1	209.5	4.69

Source Code Version	Kernel A Occupancy (%)	Kernel B Occupancy (%)
v1.0	100%	100%
v1.1	100%	100%
v2.0	100%	100%
v2.1	100%	100%

Table 4.2: Results for the single AMD scenario.

4.2.1.B NVIDIA

The computing device now being used is the NVIDIA GTX680:

Source Code Version	Throughput (FPS)	Average MTR (GB/s)
v1.0	7.5397	4.5
v1.1	17.71	N/A
v2.0	106.9	10.52
v2.1	119.4	10.46

Source Code Version	Kernel A Occupancy (%)	Kernel B Occupancy (%)
v1.0	100%	94%
v1.1	100%	94%
v2.0	94%	100%
v2.1	94%	100%

Table 4.3: Results for the single NVIDIA scenario.

4.2.2 Multiple GPUs scenario

In this subsection, different combinations of multiple GPUs scenarios are tested: Two AMD Radeon HD 6970, an NVIDIA GTX680 with an NVIDIA Tesla C1060, and finally, an AMD Radeon HD6970 with an NVIDIA GTX680.

4.2.2.A Dual AMD

The first scenario is comprised of the two AMD Radeon HD6970.

Source Code Version	Throughput (FPS)	Average MTR (GB/s)
v1.0	0.5415	0.385
v1.1	13.87	0.384
v2.0	171.96	2.867
v2.1	350.82	3.04

Source Code Version	Kernel A Occupancy(%)	Kernel B Occupancy(%)
v1.0	100%	100%
v1.1	100%	100%
v2.0	100%	100%
v2.1	100%	100%

Table 4.4: Results for the dual AMD scenario.

4.2.2.B Dual NVIDIA

The current scenario features the NVIDIA GTX680 and the NVIDIA Tesla C1060.

Source Code Version	Throughput (FPS)	Average MTR (GB/s)
v1.0	14.10	2.82
v1.1	21.16	N/A
v2.0	166.4	10.22
v2.1	186.9	10.04

Source Code Version	Kernel A Occupancy (%)	Kernel B Occupancy (%)
v1.0	50%(1)	100%(2)
v1.1	50%(1)	100%(2)
v2.0	94%(2)/75%(1)	100%(2) /100%(1)
v2.1	94%(2)/75%(1)	100%(2) /100%(1)

(1) - Tesla C1060;(2) - NVIDIA GTX680

Table 4.5: Results for the dual NVIDIA scenario.

4.2.2.C Hybrid AMD/NVIDIA

This final scenario features one GPU from each vendor: the AMD Radeon HD6970 and the NVIDIA GTX680.

4.3 Analyzing the obtained results

When inspecting the output of the profiling tools used in the various version/scenario combination, and whose results are displayed in the previous sections' tables, several facts become clear.

First, a note must be made regarding the NVIDIA profiling tool. Due to the fact that NVIDIA has its own parallel computing framework, CUDA, there is not great interest by NVIDIA to facilitate the usage of OpenCL. For example, NVIDIA Nsight, the profiling

Source Code Version	Throughput (FPS)	Average MTR (GB/s)
v1.0	3.119	5.44[(2)]/0.222[(3)]
v1.1	16.99	N/A
v2.0	194.1	N/A[(2)]/3.28[(3)]
v2.1	291.4	N/A[(2)]/4.181[(3)]

Source Code Version	Kernel A Occupancy (%)	Kernel B Occupancy (%)
v1.0	94%[(2)]	100%[(3)]
v1.1	94%[(2)]	100%[(3)]
v2.0	100%[(3)]/94%[(2)]	95.2%[(3)]/100%[(2)]
v2.1	100%[(3)]/94%[(2)]	100%[(3)]/94%[(2)]

[(2)] - NVIDIA GTX680;[(3)] - AMD Radeon HD6970

Table 4.6: Results for the multiple hybrid GPU scenario.

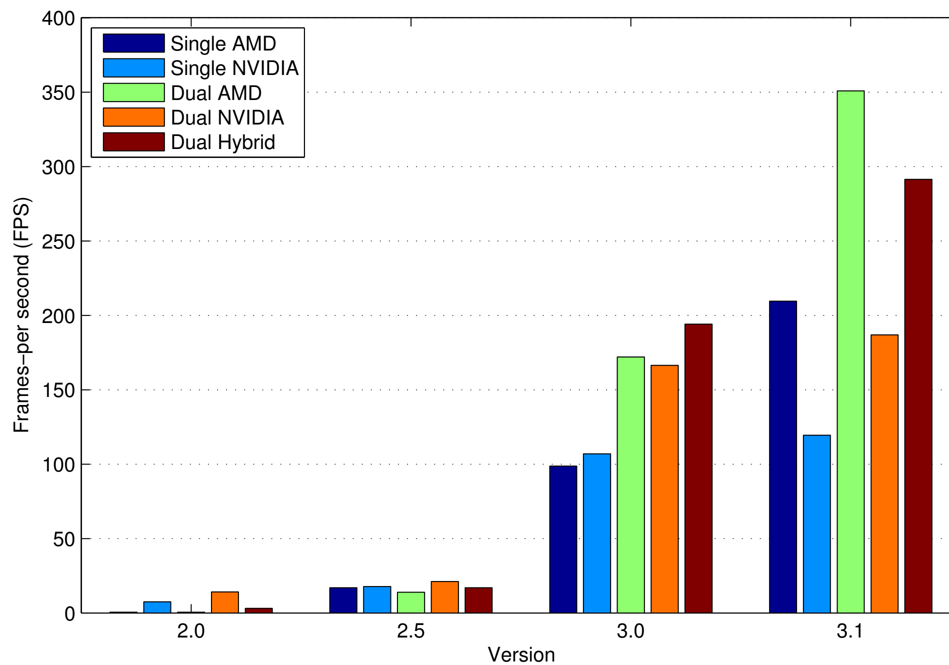


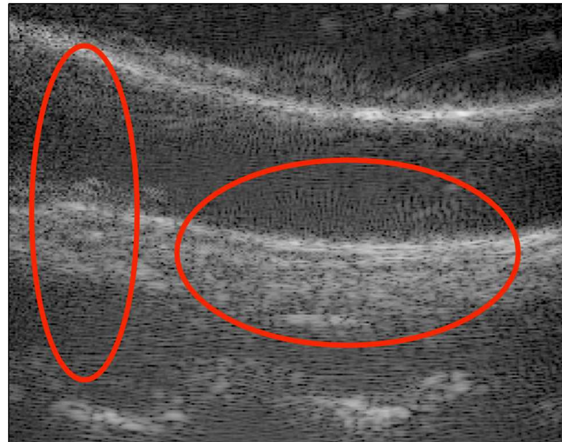
Figure 4.5: Throughput comparison between the different scenarios and versions.

tool used in this work, only provides basic OpenCL profiling capabilities. Also, the latest graphic drivers have been reported to halve the OpenCL performance of retail GPUs, such as the GTX680. Nonetheless, NVIDIA has a few perks. Starting with v1.0 of the source code, when comparing the performance of the NVIDIA GTX680 with the AMD Radeon HD6970 (single GPU scenario), the difference regarding throughput is very large. When comparing the timelines of both cases, it is clear that the reason for such result, resides in the fact that NVIDIA caches the kernels' binaries. Thus, when a new frame is being processed, the application simply fetches the kernel binary from the cache. On the contrary, AMD devices do not have any kind of binary caching, resulting in a much greater processing time. Secondly, when inspecting all the scenarios, NVIDIA GPUs tended to perform better memory transfer-wise.

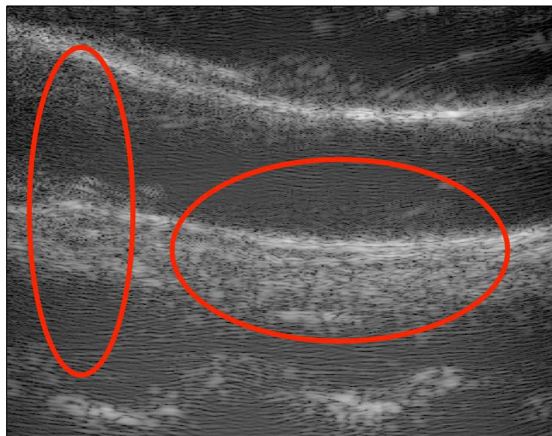
When comparing both the AMD and NVIDIA single and multiple GPU scenarios (Tables 4.2,4.3,4.4,4.5 and 4.6), in versions 1.0, 1.1 and 3.0 NVIDIA fares better in terms of throughput, but the same can not be said regarding version 3.1. The reason for such event resides in the fact that this version uses vectorial data types and operations. As introduced previously in Chapter 3, AMD Thread Processors (TPs) feature four basic operations Streaming Processor (SP), but they are only used simultaneously when vectorial operations are used.

In version 3.1, although the overlapping of memory transfers and kernel executions is implemented, current AMD and NVIDIA OpenCL drivers enforce kernel execution calls to be blocking. This means that until the kernel returns, the host code does not advance. Of course, with the use of explicit synchronization, the improvement in performance is low. The immediate solution is to use two host threads to control the execution, but this would require very rigorous synchronization. By alternative, when the memory transfers and kernel executions are blocking, the need for explicit synchronization between threads disappears, and the overlap occurs. Compared to the implementation of version 3.1, the performance improvement is considerably larger. To illustrate these claims, a special scenario is used, adapted from version 3.0. Thus, two AMD GPUs are used, and for each one, two pthreads are launched.

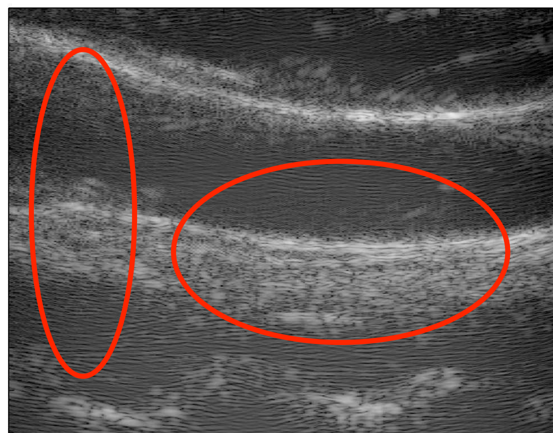
Finally, given the throughputs achieved, the lateral resolution of the image was changed, to provide proof of its influence on the overall image quality, particularly on the field of image artefacts. Fig. 4.6 shows the same image with three different lateral resolutions, and highlights the regions where the differences are more visible.



(a) Simulation with 256*x*255 pixels.



(b) Simulation with 512*x*255 pixels.



(c) Simulation with 1024*x*255 pixels.

Figure 4.6: Images reconstructed with various lateral resolutions to evaluate its effect on image quality. Highlighted in red are the regions where the differences can be seen. In subfigures b) and c), the incidence of image artifacts is much lower than that of subfigure a), which has lower lateral resolution.

4.4 Conclusions

The results obtained in the current chapter, show promising signs that GPUs programmed under the OpenCL framework can deliver a real-time medical ultrasound imaging system. Nonetheless, there is still work being done, specifically by addressing some of the problems encountered during the course of this thesis' work, namely the current implementation of the OpenCL drivers by the vendors, that make it impossible for the clear implementation of the memory transfers and kernel execution overlaps. These results also make it clear that the portability provided by OpenCL sacrifices performance, and the only way to address this, is to adapt the code to the specific architecture of the computing device.

5

Conclusions

Contents

5.1 Future Work 56

While the Synthetic Aperture (SA) Beamforming kernels developed during the course of this work did achieve the proposed mark of 350 frames-per-second (FPS), there is still potential for major improvements, whose result will yield much better results, potentially over 1000 FPS. By comparing the various scenarios: single Graphics Processing Unit (GPU), multiple GPUs and multiple Central Processing Unit (CPU) cores, several facts arise. The use of texture images yields major speed-ups in GPUs, but current CPU architecture is not optimized for such structures, thus making the CPU a poor candidate in the pool of many-core computing devices. When using two different GPUs, the performance is only slightly better than that of the single-GPU scenario (in some cases, performing even worse). The reason for this is the introduced overhead in synchronizing data and sharing the Peripheral Component Interconnect Express (PCIe) bus. The main bottleneck of the implementation is still fairly obvious to pinpoint: memory transfers. The solution to minimize this bottleneck is not obvious: the bandwidth attributed the device allocates to perform a given memory transfer is a log-like function of the size of the data to be transferred. While the parallel processing of Low-Resolution Images (LRIs) would increase the volume of data to be transferred, and maximize the allocated bandwidth, its effect on kernel execution would be nefarious, with additional execution branches. On the other hand, converting the data from *float* to *half* would decrease the allocated bandwidth.

5.1 Future Work

As stated above, the results of this work are not final, as there is still ample space for improvement. There are many challenges in the design of portable parallel kernels, the biggest being how to properly address the differences of the architectures of the many-core devices. Additionally, only now are SA beamforming techniques blooming in the ultrasound imaging field. The delay-and-sum approach exploited in this thesis is still a work in progress, but there is already work in progress regarding the design of frequency-domain imaging reconstruction algorithms that can benefit from a parallel approach using Open Computing Language (OpenCL) on many-core devices.

Bibliography

[1] H. Azhari, *Basics of biomedical ultrasound for engineers*, 2010.

[2] R. Farber, "Part 2: Opencl - memory spaces," 2010. [Online]. Available: http://www.codeproject.com/Articles/122405/Part-2-OpenCL-Memory-Spaces

[3] O. Rosenberg, "Opencl do's and dont's," 2011.

[4] [Online]. Available: http://techreport.com/r.x/core-i7-4770k/haswell-die.jpg

[5] [Online]. Available: http://www.hardwarebenchnews.com/wp-content/uploads/2012/11/FX_8350_Review_Piledriver_die.jpg

[6] [Online]. Available: http://www.corsair.com/us/memory-by-product-family/value-select-memory-upgrades/vs8gsdskit800d2.html

[7] AMD, "White Paper — AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE," 2012.

[8] NVIDIA, "Whitepaper - NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.

[9] S. Haykin and B. Van Veen, *Signals and systems Second Edition*, 2003.

[10] L. J. Ziomek, *Acoustic Field Theory and Space-Time Signal Processing*, 1995.

[11] M. Bom and E. Wolf, "Principles of optics," *Pergamon Oxford (Ed.)*, 1980.

[12] L. J. Cutrona, "Comparison of sonar system performance achievable using synthetic-aperture techniques with the performance achievable by more conventional means," *The Journal of the Acoustical Society of America*, vol. 58, no. 2, p. 336, 1975. [Online]. Available: http://link.aip.org/link/JASMAN/v58/i2/p336/s1&Agg=doi

[13] T. Stepinski, "An Implementation of Synthetic Aperture Focusing Technique in Frequency Domain," *IEEE Transactions on Ultrasonics Ferroelectrics and Frequency Control*, vol. 54, no. 7, pp. 1399–1408, 2007. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/17718329

[14] D. Hawkins, "Synthetic Aperture Imaging Algorithms: with application to wide bandwidth sonar," no. October, 1996. [Online]. Available: http://ir.canterbury.ac.nz/handle/10092/1082

[15] M. Feldman, *Hilbert Transform Applications in Mechanical Vibration*. Chichester, UK: John Wiley & Sons, Ltd, Mar. 2011.

[16] A. J. Hunter, B. W. Drinkwater, and P. D. Wilcox, "The wavenumber algorithm for full-matrix imaging using an ultrasonic array." *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, vol. 55, no. 11, pp. 2450–62, Nov. 2008. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/19942535

[17] J. Amaro, G. Falcao, B. Y. S. Yiu, and A. C. H. Yu, "Portable Parallel Kernels For High-Speed Beamforming In Synthetic Aperture Ultrasound Imaging," 2013.

[18] B. S Yiu, I. H Tsang, and A. H Yu, "GPU-based beamformer: Fast realization of plane wave compounding and synthetic aperture imaging." *IEEE Transactions on Ultrasonics Ferroelectrics and Frequency Control*, vol. 58, no. 8, pp. 1698–1705, 2011. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/21859591

[19] NVIDIA, "OpenCL Programming Guide for the CUDA Architecture," 2009.

[20] AMD, "OpenCL Programming Guide AMD Accelerated Parallel Processing," no. December, 2012.

[21] G. E. Moore, "Readings in computer architecture," M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, ch. Cramming more components onto integrated circuits, pp. 56–59.

[22] [Online]. Available: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/archived-tools/amd-app-profiler/user-guide/app-profiler-kernel-occupancy/

[23] [Online]. Available: parallel.vub.ac.be/~jgcornel/Occupancy.html

[24] V. Volkov, "Better performance at lower occupancy," *Proceedings of the GPU Technology Conference, ...*, 2010. [Online]. Available: http://people.sc.fsu.edu/~gerlebacher/gpus/better_performance_at_lower_occupancy_gtc2010_volkov.pdf

Appendix A

PORTABLE PARALLEL KERNELS FOR HIGH-SPEED BEAMFORMING IN SYNTHETIC APERTURE ULTRASOUND IMAGING

Joao Amaro, Gabriel Falcao*, Billy Y. S. Yiu‡, and Alfred C. H. Yu‡*

*Instituto de Telecomunicações, University of Coimbra, Portugal
‡Medical Engineering Program, University of Hong Kong, Hong Kong SAR

ABSTRACT

In medical ultrasound, synthetic aperture (SA) imaging is well-considered as a novel image formation technique for achieving superior resolution than that offered by existing scanners. However, its intensive processing load is known to be a challenging factor. To address such a computational demand, this paper proposes a new parallel approach based on the design of OpenCL signal processing kernels that can compute SA image formation in real-time. We demonstrate how these kernels can be ported onto different classes of parallel processors, namely multi-core CPUs and GPUs, whose multi-thread computing resources are able to process more than 250 fps. Moreover, they have strong potential to support the development of more complex algorithms, thus increasing the depth range of the inspected human volume and the final image resolution observed by the medical practitioner.

***Index Terms*—** Synthetic aperture, Ultrasound medical imaging, Beamformer, OpenCL, GPU

1. INTRODUCTION

Ultrasound medical imaging systems are nowadays a fundamental tool for helping medical practitioners performing a reliable non-invasive diagnosis procedure. Based on the processing and analysis of pulse-echo signals, current ultrasound imaging systems are complex from a hardware perspective due to the use of array transducers that inherently involve multi-channel processing. They are supported by extensive microelectronics systems such as filed-programmable gate arrays (FPGA) and digital signal processors (DSP) [1].

As the theoretical principles of advanced ultrasound image formation paradigms have become more mature in recent years, there is a growing level of interest in realizing them in practice. Of particular interest is the real-time execution of those algorithms, which represents a key factor in ultrasound imaging regarding its bedside clinical role.

One such advanced ultrasound technique is synthetic aperture beamforming, which transmits unfocused pulses form distinct lateral positions [2]. Each pulse generates echoes that are received by all channels in the sensor to form a low-resolution image (LRI) per each instance of pulse-echo sensing, which is accomplished by performing delay-and-sum beamforming at each pixel position. Then the sum of a predefined set of LRIs can be used to form high-resolution images (HRI) [3]. Naturally, these operations are computationally demanding. Additionally, unlike previous ultrasound techniques that use the same set of focusing delays, synthetic aperture beamforms each pixel based on different sets of varying focus delays [2], which is more complex to do and demands higher processing capabilities.

Although originally dedicated to image rendering, graphics processing units (GPU) have been recently introduced as powerful parallel accelerators for general-purpose computing [4, 5]. They have been shown to be well-suited to the real-time realization of synthetic aperture imaging algorithms [6]. Unlike conventional approaches that exploit the compute unified device architecture (CUDA) interface [7] which is limited to execute only in NVIDIA GPUs, in this article we propose using OpenCL [8, 9], a more generic programming model, and show that it supports the execution of these parallel kernels on a wide variety of multi- and many-core systems [10]. Depending on the specificities of the system (e.g., number of array transducers, image resolution, etc.), OpenCL allows targeting synthetic aperture kernels to the most appropriate heterogeneous computational environment [11] that is capable of supplying the necessary processing power. In this article we report experimental results obtained by running the same kernel on Intel CPUs and ATI or NVIDIA GPUs. We also show how these OpenCL-based signal processing kernels were developed in order to extract parallelism from the architecture.

2. SYNTHETIC APERTURE IMAGING

We first discuss the theory associated with synthetic aperture imaging. We start by identifying and elaborating on the different phases of the algorithm.

This work is funded in part by the Portuguese Foundation for Science and Technology (FCT) project PEst-OE/EEI/LA0008/2011, as well as the Hong Kong Innovation and Technology Fund (ITS/292/11).

2.1. Signal transmission characteristics

The transmitting elements used in current transducers emit a linear signal. The synthetic aperture algorithm is based on the emission of a spherical wave from each transmission source. To minimize changes in the hardware, we can use the current transducers, but the signal is emitted as in figure 1 to emulate a spherical wave. The point behind the element array represents the epicenter of the spherical wave, or the virtual source point.

2.2. Signal reception

As previously mentioned, the signal propagates in the form of a spherical wave both in transmission and in reception (after reflection in the scattering medium). So, we must take into account the relative delay of the signal received from a reflection in a pixel in relation to the neighboring receiving elements.

2.3. LRI calculation

To compute each pixel of a LRI, we now have to account for the contribution of each receiving element, with a delay-and-sum procedure. This is basically a linear interpolation of the pixels' position neighboring analytic data samples $\alpha_{n,m}$:

$$\alpha_{n,m}(P_0) = \lambda a_{n,m}(k) + [1-\lambda]a_{n,m}(k+1), \quad (1)$$

where n corresponds to the $n-th$ receive channel and m represents the $m-th$ transmitting virtual source point. Each of the receiving elements contribution is passed through a window function ω_n. To find the depth sample number k, we must first consider the focusing delay $\tau_{n,m}(P_0)$ and the interpolation weight λ:

$$k = \lfloor f_s \tau_{n,m}(P_0) \rfloor, \quad (2)$$

$$\lambda = 1 + k - f_s \tau_{n,m}(P_0). \quad (3)$$

The focusing delay in SA imaging is calculated in the following way:

$$\tau_{n,m}(P_0) = \frac{d_T(P_0;m) + d_R(P_0;n)}{c_0}, \quad (4)$$

where $d_T(P_0;m)$ represents the distance between the transmitting position and the position of pixel P_0, and $d_R(P_0;n)$ the distance between this position and the nth receiving element, while c_0 is the speed in the scattering medium.

Finally, the value for pixel P_0 of the mth LRI can be obtained by:

$$L_m(P_0) = \sum_{n=1}^{N} \omega_n \alpha_{n,m}(P_0), \quad (5)$$

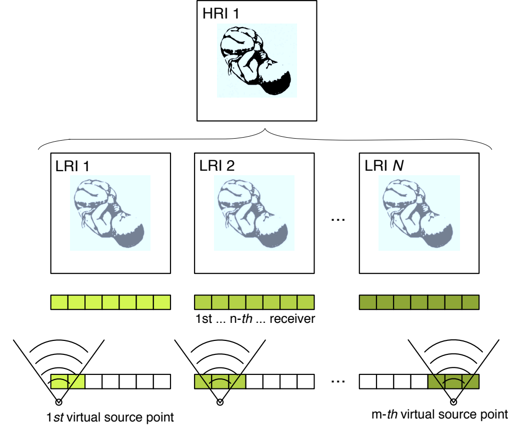


Fig. 1. Pulse-echo system based on synthetic aperture and the multi-channel generation of LRI and HRI.

2.4. HRI compounding

After computing a new LRI, we replace the oldest LRI in the compounding frame, and after recursive summation of the entire frame (with size N), we are able to form a new HRI. This can be modeled as a relation between the new HRI and the previous one:

$$H_i(P_0) = H_{i-1}(P_0) + L_i(P_0) - L_{i-M}(P_0). \quad (6)$$

3. PARALLEL PORTABLE KERNELS FOR SA

Historically, developing an algorithm for running on a multicore system was considered nontrivial. Recently, parallel programming models were developed to allow programming some specific architectures in particular [7]. Later, the introduction of a broadly accepted programming model compatible with different multi- and many-core architectures was made possible with the arrival of OpenCL [8]. OpenCL provides a framework that allows signal processing programmers to develop code once and execute it on a variety of multicore systems such as CPUs or GPUs. Using a C/C++ environment, the programmer instructs the compiler how a code section should be parallelized. Parallelization is organized by the programmer in work-groups, where each work-group dispatches a certain predefined number of work-items. At runtime, the program inspects the compute resources available on the platform, compiles the source code according to it and launches execution. At the end, processed data is sent back to the host system that orchestrates execution [9].

The parallel algorithm developed exploits thread-level parallelism to perform the calculation of new LRIs. Processing is performed on a pixel-per-pixel basis for all channels of the system. The processing unity with smaller granularity-level is the work-item.

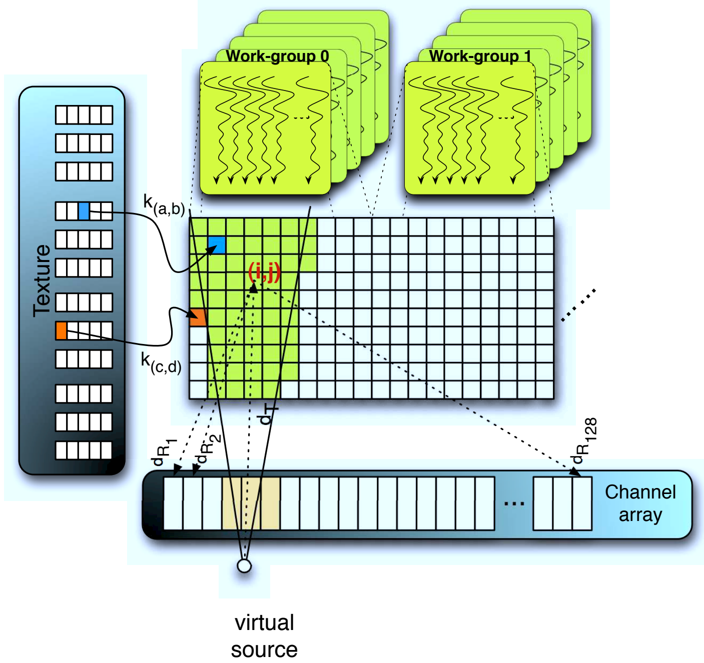


Fig. 2. Calculation of pixel (i, j) for the nth LRI. Illustration of the influence of a virtual source point in the calculation of the pixels of an LRI. Pixel with index (i, j) is processed by a work-item in parallel with the processing of other pixels in the same work-group.

3.1. Parallel calculation of LRIs

Figure 2 describes how pixels are influenced by each virtual source. The processing is performed in parallel by all work-items and one of two situations occur: either the pixel is under the influence of a virtual source or it isn't. This verification implies the use of divergent (conditional) instructions, which penalizes performance. However, parallelization is achieved since each work-item processes in parallel one of the pixels that are under the influence of a same virtual point (for those who are outside, the work-item returns execution). Every two work-groups are able to perform the parallel processing of a complete LRI.

Pulse-echo distances are represented by d_T and d_R, which are used to calculate the delay $\tau_{n,m}$ as shown in (4) and finally obtain the depth sample number k indicated in (2).

The level of parallelism achieved increases (until a certain limit) with the number of compute resources available. In the case of a GPU, the processing of hundreds of pixels in parallel is possible.

3.2. Using texture memory to accelerate computation

On the GPU, texture memory has latency times similar to those of global memory. The main advantage of using this type of memory lies on the level 1 cache capabilities associated with textures. Not only data used is maintained for reuse, but also do its immediate neighboring elements, which is useful in this particular algorithm. Therefore, analytical data is loaded into textures before the kernel is launched. Then, data is accessed by each work-item at element in the position given by depth sample k previously calculated, in order to produce $\alpha_{n,m}$ [6]. This procedure basically consists of a weighted summation of interpolated channel-domain samples for all N array channels as described in (1).

In the case the OpenCL kernel is running on a CPU, where texture memory does not exist, this functionality is obtained using software emulation. However, it is worth noting that the use of textures in this scenario is not recommended mainly due to efficiency reasons.

4. EXPERIMENTAL RESULTS

4.1. Apparatus

The experimental results were obtained using the OpenCL C API, interfaced with Matlab's MEX-function, and the C console compiled with Visual Studio 2010. The computer has a quad-core Intel Core i7 950 @3.07 GHz, 3GB of RAM memory, running Windows 7 Ultimate x86. The OpenCL devices are an ATI Radeon HD6970 (Cayman) with 1536 shaders, a NVIDIA Tesla C1060 with 240 cores. We must note that the relation between performance vs. number of compute units is not the same for both vendors, as NVIDIA indicates fewer units. As an additional test, the algorithm was also run on the CPU, to further demonstrate the computing potential of the GPUs.

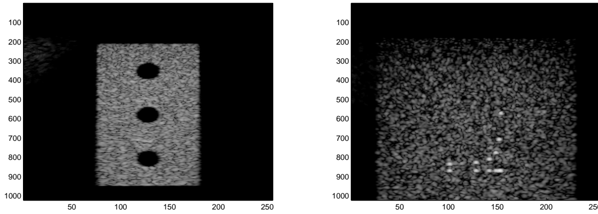
A Sonix-RP research scanner equipped with a pre-beamformed data acquisition tool was used to collect the dataset processed in the host. Ultrasound parameters are: frequency – 10 MHz; transmit pulse shape – 2-cycle sinusoid; pulse repetition frequency – 5 kHz. The synthetic aperture implementation is based on a scanner front-end that was reprogrammed to fire according to a virtual point source configuration. It uses 97 point sources in total (0.3 mm laterally spaced apart, 20 mm axially behind field of view), each formed from a 64-channel aperture. It performs one firing from each virtual point source, swept from left to right side. The data acquisition is based on 128 channels received in parallel using the pre-beamformed data acquisition tool, with 40 MHz sampling and 12-bit resolution.

4.2. High frame-per-second LRI throughput calculation

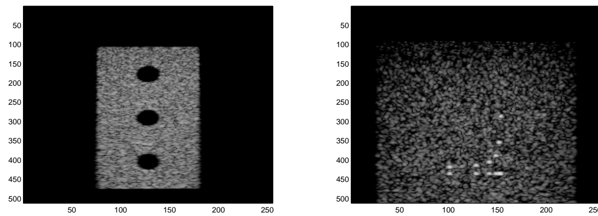
To perform the experimental results we have run the same OpenCL kernel on 3 different platforms: one multi-core CPU and two many-core GPUs. We used 2 datasets which were obtained using synthetic aperture beamforming: dataset 1 (DS1) consists of a Perforated Plate, while dataset 2 (DS2) shows random Dots. The selected aperture is 256 for all experiments. Table 1 indicates the total computation times for gen-

Table 1. Computation times for two datasets: Perforated Plate (DS1) and Dots (DS2) generating images with resolution 512×255 pixels.

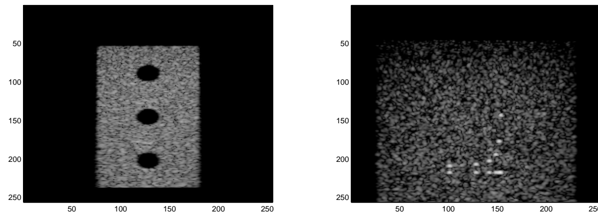
Platform	Radeon HD6970 GPU		Tesla C1060 GPU		Intel i7 950 CPU	
Dataset 512×255 pixels	DS1	DS2	DS1	DS2	DS1	DS2
Total memory operations time	695 ms	695 ms	1.068 s	1.058 s	187 ms	182 ms
Total kernel execution time	403 ms	382 ms	1.067 s	670 ms	24.349 s	20.221 s



(a) 1024×255 pxl. images gen. on ATI Radeon HD 6970



(b) 512×255 pxl. images gen. on Intel i7 950



(c) 256×255 pxl. images gen. on NVIDIA Tesla C1060

Fig. 3. Perforated Plate (left) and Dots (right) images generated from datasets obtained using synthetic aperture beamforming. The images were generated by running the same OpenCL kernel on three different multi-core platforms.

erating 97 LRIs. They include data transfers between host and device, and kernel execution times on device. It can be seen that the CPU platform presents lower transfer times, which is natural since it is not limited by the PCIe bus, but it also shows considerably higher kernel processing times due to a lower number of processing cores available. The speedup obtained from CPU to ATI GPU execution ranges from 22 to 30, while against the NVIDIA GPU it approximates $52 \sim 60$ times. The Radeon GPU is capable of processing more than 253 frames per second (fps), as the inspection of table 1 indicates, achieving 1.35 GFLOPS for DS1 with 512×255 pixels.

Figure 3 demonstrates the code portability concept by showing that all generated images are equivalent and that the main difference is computation time. Many-core systems with superior capabilities, such as number of cores, higher clock frequencies or memory bandwidth would run the OpenCL kernel even faster, thus processing more fps.

5. RELATION TO PRIOR WORK

To our knowledge, this work is perhaps the first attempt to investigate the feasibility of adopting a hardware-flexible parallel processing approach to execute synthetic aperture beamforming operations at real-time throughput. Based on devising portable software kernels using the OpenCL framework, our approach inherently differs from previous CUDA-based synthetic aperture beamformers that can only work on vendor-specific hardware (NVIDIA) [6]. Also, it is not the same as other solutions that attempt to use FPGAs [12], computer clusters [13], and DSP platforms [14] for synthetic aperture image computing purposes. From an ultrasound system design standpoint, our OpenCL-based solution should be more favorable than others as its code portability allows the beamforming kernel to be hosted on a wide variety of computing hardware, from multi-core CPUs to many-core GPUs and even FPGAs. This would provide ultrasound system designers with more flexibility in designing novel hardware architecture for synthetic aperture ultrasound imaging.

6. CONCLUDING REMARKS

With the availability of code-portable parallel processing kernels to handle beamforming operations, it becomes more feasible to pursue practical realization of synthetic aperture ultrasound imaging whose image formation principles are known to be computationally demanding. This work is therefore expected to contribute to the latest developments in advanced ultrasound system design. It is worth noting that, besides synthetic aperture beamforming, our OpenCL-based parallel processing approach may be extended to other computing operations in synthetic aperture imaging. For instance, in the delay-and-sum process, it may be of interest to include an adaptive apodization module that applies signal-dependent channel weighting. Such an adaptive beamforming strategy is well considered to be computationally demanding as well. Parallel processing, especially portable ones that can be executed on a variety of computing hardware, may provide an answer to this technical hurdle.

7. REFERENCES

[1] G. York and Y. Kim, "Ultrasound processing and computing: Review and future directions," *Annu. Rev. Biomed. Eng.*, vol. 1, pp. 559–588, 1999.

[2] J. A. Jensen, S. I. Nikolov, K. L. Gammelmark, and M. H. Pedersen, "Synthetic aperture ultrasound imaging," *Ultrasonics*, vol. 44, Supplement, pp. e5–e15, 2006.

[3] S. I. Nikolov, K. L. Gammelmark, and J. A. Jensen, "Recursive ultrasound imaging," in *Proc. IEEE Ultrasonics Symp.* IEEE, 1999, pp. 1621–1625.

[4] T. P. Chen and Yen-Kuang Chen, "Challenges and opportunities of obtaining performance from multi-core CPUs and many-core GPUs," in *Proc. of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'09)*, April 2009, pp. 613–616.

[5] H.K.-H. So, Junying Chen, B.Y.S. Yiu, and A.C.H. Yu, "Medical ultrasound imaging: To GPU or not to GPU?," *IEEE Micro*, vol. 31, no. 5, pp. 54–65, 2011.

[6] B.Y.S. Yiu, I.K.H. Tsang, and A.C.H. Yu, "GPU-based beamformer: Fast realization of plane wave compounding and synthetic aperture imaging," *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on*, vol. 58, no. 8, pp. 1698–17052, August 2011.

[7] CUDA Developer NVIDIA, "CUDA 5.0," Nov. 2012.

[8] Khronos Group, "OpenCL 1.2," Nov. 2011.

[9] B. R. Gaster, H. Lee, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, Morgan Kaufmann, 2012.

[10] G. Falcao, V. Silva, L. Sousa, and J. Andrade, "Portable LDPC Decoding on Multicores Using OpenCL," *IEEE Signal Processing Magazine*, vol. 29, no. 4, pp. 81–109, July 2012.

[11] S. Singh, "Computing without processors," *Communications of the ACM*, vol. 54, no. 8, pp. 46–54, August 2011.

[12] J. A. Jensen, M. Hansen, B. G. Tomov, S. I. Nikolov, and H. Holten-Lund, "System architecture of an experimental synthetic aperture real-time ultrasound system," in *Proc. IEEE Ultrason. Symp.*, 2007, pp. 636–640.

[13] F. Zhang, A. Bilas, A. Dhanantwari, K. N. Plataniotis, R. Abiprojo, and S. Steriopoulos, "Parallelization and performance of 3D ultrasound imaging beamforming algorithms on modern clusters," in *Proc. ACM Int. Conf. Supercomput.*, 2002, pp. 294–304.

[14] C. R. Hazard and G. R. Lockwood, "Theoretical assessment of a synthetic aperture beamformer for real-time 3-d imaging," *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on*, vol. 46, pp. 972–980, 1999.