Domain transform acceleration for the GPU-based real-time planar near-field acoustic holography

Oznaya Angeles, M.E.

Award date:
2014

Domain Transform Acceleration for the GPU-Based Real-Time Planar Near-Field Acoustic Holography

Master thesis by:

Miguel Emilio Oznaya Angeles

Supervisors:

dr. ir. Rick Scholte
prof. dr. Henk Corporaal
prof. dr. Johan Lukkien

Mentors:

ir. Wouter Ouwens
ir. Gert-Jan van den Braak

Eindhoven University of Technology
Department of Mathematics and Computer Science
Master of Science in Embedded Systems

Presentation date: October 10, 2014
Publishing date: October 27, 2015

Abstract

Planar Near-Field Acoustic Holography (PNAH) is a method to approximate the pressure distribution on a desired plane in space employing pressure data measured on a parallel plane, separated by a known distance. When the targeted plane is the surface of an object of interest, the output of the PNAH provides insight on the dynamical behaviour of said object. These characteristics enable the PNAH to be employed as a vibration detection technique. Its advantages over other typically employed methods, such as accelerometer- or laser-based measurements, are low cost and increased spatial resolution and accuracy.

In the semiconductor industry, an important development driver is the constant need for enhanced circuit features at a lower cost. One of these developments, within the lithography machines, is based on the need to increase the wafer size from 300 mm. to 450 mm. in order to decrease the circuit production costs. One of the effects that this would have on the mentioned machines is that additional vibrations would appear on the wafers' mounting table. To counteract these vibrations, a proper controller needs to be implemented. The PNAH technique has been proposed as a vibration detector to provide the required input for such a controller.

A real-time implementation of the PNAH, running in a GPU platform, has already been presented, reaching a maximum throughput of 1 kHz. The objective of this project is to build up on such implementation in order to increase this value and improve it by at least 15%. To achieve this, three different variations within the PNAH algorithm are proposed and tested. These alternatives concern the time-frequency domain transform (Fourier Transform), as well as the time domain preprocessing stages; more specifically, they explore the advantages and limitations of employing a *naive* matrix multiplication paradigm to compute the domain transform. Additionally, the use of the raw binary measured data is proposed as an input to the PNAH algorithm to further increase the current throughput by bypassing the signal preprocessing stages. This raw input is also coupled with the use of the binary Walsh kernel functions; therefore proposing a different domain transform *route* (Walsh-Fourier Transform).

The obtained results show that using a matrix multiplication approach does have advantages over the *fast* version of this algorithm (FFT); however, they are limited to the amount of output information required by this procedure. Besides showing that the raw binary input is adequate for the PNAH algorithm, the expected execution time decreases were reached. Finally, the Walsh-Fourier Transform is shown to output correct results, but its implementation requires further optimizations for this potentially faster technique to fulfill the required real-time constraints.

Acknowledgements

This project represented an important challenge from which I learnt not only about the existence and inner workings of acoustic holography, but also about how an innovative technology can impact a well established product design process, provide didactic insight on physical phenomena or even influence artistic applications.

I would like to express my gratitude towards all the Sorama team. Specially, I want to thank Rick for granting me with this opportunity and for sharing his expertise on the subject, and also Wouter for being so patient with my beginner questions and for providing me with essential technical guidance along the way.

I want to thank dr. Corporaal because his constant supervision, advices and support made the project direction clearer for me. I would also like to thank dr. Lukkien for accepting the invitation to join the committee and for contributing with insightful mid-term feedback. Additionally, I want to recognize Gert-Jan's valuable technical support which helped me gain a better understanding in fundamental aspects related to the project.

A big 'thank you' goes to all my Mexican, Dutch and international friends who were always, personally or remotely, providing me with a strong support, words of wisdom and happy moments. Without these, this project would not have been the good experience it was.

Additionally, I want to thank CONACyT for giving me the opportunity of earning a degree in the Netherlands by providing the means to do so.

Finally, my biggest gratitude is towards my main drive for this project: all of my family. I want to thank Paty and Carlos for helping me conclude this vital stage. And above all, I want to thank a mi Madre, al Miguelito, a César y a la Nena (mentioned by order of appearance) for their undescribably indispensable support throughout this period of time. The entirety of this work is dedicated to you. Love you, guys.

Miguel Emilio Oznaya Angeles.

Eindhoven, October 2014.

Contents

Chapter one

Introduction

Sound is the pressure variation over time at a point in space. This variation requires a physical environment (liquid, solid or gas) to be transmitted through space; usually the transmission of this variation is regarded as a sound wave. This definition of sound might require slightly more time to understand when compared to the initial perception and idea we have of sound. Commonly, sound is used to transmit a message or information, however, sound can also be an undesired effect of certain process.

In the first case, sound is employed to carry out several interactions, the most important of these being human interaction. Examples of human communication are the use of an established language to transmit messages. Besides language, which can be thought as a structured protocol employing a finite number of sounds, music is also employed to communicate messages and emotions in a *more free* way when compared to language. Apart from human communication, sound also allows us to obtain feedback from the environment in which we are located: receiving information from a device or tool or providing warnings about a potential hazard, for example. In all these cases, sound is employed as an information transmission method.

As mentioned above, there are also cases in which sound is a collateral product causing negative effects on the environment. Think of a train braking when arriving to a platform, an airplane turbine or a coffee machine in duty. The negative effect of the undesired sound produced by these devices is so strong that they can easily be mentally *replayed* and associated with a somewhat unpleasant perception. However, comfort is not the only advantageous effect of a silent process. When a device or process emits a sound, some of its parts or elements move (vibrate) following a certain pattern. This movement causes the pressure in the medium surrounding the considered system to vary over time, thus emitting sound. In many cases, this movement is not intended nor taken into account. Therefore, it might interfere with the proper operation of the system, causing unexpected faults. In this way, either a silent system or a one emitting a predetermined or known sound is desired, since this could mean that the internal system vibrations have been taken into account.

Examples of systems where the undesired vibrations need to be controlled and counteracted are the ones relying on accurate positioning of a tool or another object. In these cases, such vibrations introduce errors which reduce the overall precision of the system. To minimize these errors, the vibrations have to be compensated for, typically with the implementation of a control loop. For the scope of this project, a particular case of such a system is used.

1.1 Problem Description

Within the semiconductor industry, a constant need for cheaper circuits exists. One of the critical steps in the creation of such a circuit is the lithography stage. Due to the frequent requirement of lowering costs, the lithography systems need to output more wafers in less time (the term *wafer* denotes the circle-shaped stack of layered materials onto which several copies of the circuit pattern are *printed*). One

of the proposed alternatives to achieve this is to increase the size of the employed wafers, from 300 mm. to 450 mm. in order to be able to obtain more chips per wafer.

In this way, the rate of output wafers per unit of time is increased, but the effects of doing so cannot remain unnoticed. One of the major consequences of this is that the *chuck*, which is the moving *table* on which the wafer is placed and printed, has to also increase its size and dynamical properties, therefore causing undesired vibrations. Being lithography the delicate process it is, such vibrations can prove disastrous for the output; a way of controlling these vibrations is required. Moreover, the controller has to be fast enough to accurately counteract these movements. This means that the vibration measurement system has to output results at a similar rate.

Typical methods of measuring the vibrations are placing accelerometers on the surface of interest or employing laser-based methods. Some of the disadvantages of these methods are the following:

- Accelerometers interfere with the vibration pattern.
- Laser-based methods are very expensive.
- Usually, vibrations are only measured at a small number of points in space, thus, resulting in aliasing.
- Analog preprocessing is required.

As an alternative to tackle these issues, an acoustic-based vibration method is proposed. Such a method is already implemented and in use by Sorama. Based in Eindhoven, this company developed the Sorama Cam, which consists of an array of 1024 small microphones which are able to measure the vibrations, posing the following advantages over the previously mentioned alternatives:

- Microphones perform a contactless measurement; the vibration pattern suffers no interference.
- The employed microphones are considerably cheaper.
- Configured in an array fashion, the microphones cover an extensive area sampling at 1024 points in space.
- The output of the microphones is a digital stream; no analog circuitry is required.

The basic idea is to *point* such camera towards the object of interest and measure the vibrations. Because of the existing distance between the surface of interest and the location of the sensors, the measured magnitudes do not represent the behaviour on the object. To obtain this information, Sorama uses Planar Near-Field Acoustic Holography (PNAH) algorithm to *trace* the measured vibrations from the measurement plane (Sound Cam location) to the source (object location). An illustration of such a process is depicted in Figure 1.1.

Back to the lithography system, a controller needs to have a throughput of at least 1kHz to successfully correct for the undesired vibrations. This means that Sorama's proposed method of vibration detection needs to output information at the same throughput.

1.2 Project Goals

A real-time execution of the PNAH algorithm, achieving a throughput of at least 1kHz was already implemented by Sorama. However, considering a case when this requirement is raised to at least 2kHz in the next five years, a yearly improvement of at least 15% is needed. Considering that this implementation is already accelerated via the use of a GPU, some changes in the algorithm need to be proposed to keep up with the required pace.

In this sense, the main goal of the project to, based on analysis of the current implementation's execution

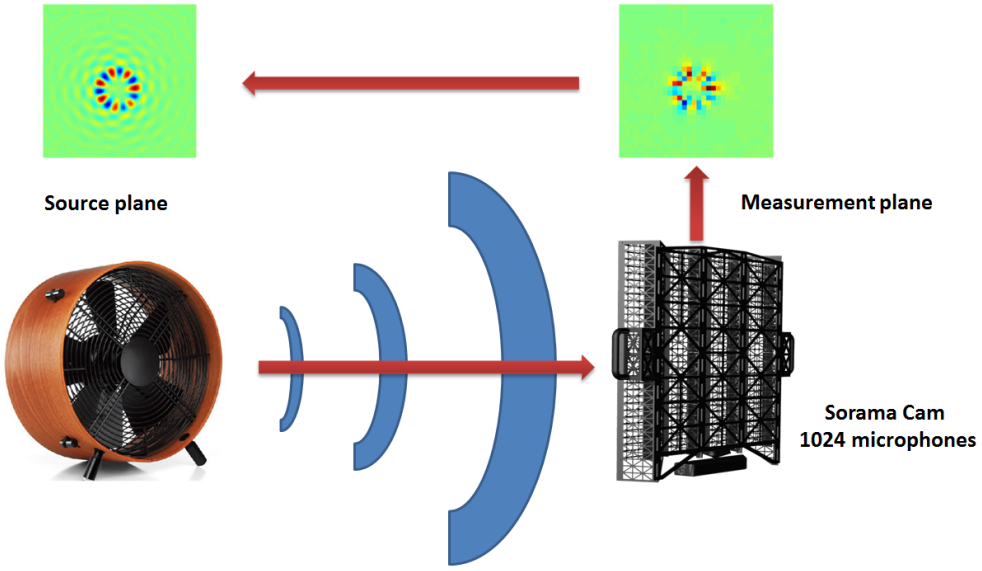


Figure 1.1: Vibration measurement employing the Sorama Cam and the PNAH technique.

times, propose and apply modifications to the PNAH algorithm that result in iterations being computed in less time, such that the yearly throughput improvement is met.

The remainder of this document is structured as follows. First, an introduction to the PNAH algorithm is provided. Then, its implementation in the current system, along with its metrics, issues and proposed approach to address the main goal are given. After this has been stated, the theory on the proposed alternatives are provided, followed by the way they are implemented in the system. Subsequently, the achieved results are presented together with an analysis on the observations. Finally, a conclusion is given, followed by some suggestions to serve as a future work proposal.

Chapter two

Planar Near-Field Acoustic Holography

The scope of this chapter is completely dedicated to the theoretical aspects of the Planar Near-Field Acoustic Holography (PNAH), as well as the steps required to get this technique working along with its most important considerations and issues to take into account.

In the first section, a basic review on Acoustic Holography and its variations is provided. Then the background concepts related to the Fourier-based Near-Field Acoustic Holography (NAH), the method employed by Sorama, are defined, making an emphasis on the PNAH. In the second section, the PNAH algorithm is described in a generalized way, abstracting out several implementation details. The objective of this approach is to explain in a broader way the different domains in which each processing stage takes place. More details on the actual implementation of the algorithm are provided in the upcoming chapter.

2.1 Acoustic Holography

Acoustic Holography (AH) is a technique to propagate the pressure distribution from one set of initial points in space to a desired set of final points. The locations of all the points within these two sets are known. Since the pressure in a region changes when an acoustic wave passes through it, the propagation of sound is used as a mean to achieve these reconstructions; thus, the *acoustic* qualifier is employed. Acoustic Holography is employed to either further- or inverse- propagate the sound waves. Through the forward AH it is possible to obtain the pressure distribution in a required region of space given that the conditions of an emitting source are known; in other words, the sound which is being propagated out of the source is reconstructed for a desired location. On the other hand, the inverse AH considers the case when the conditions in the source are unknown but the pressure field can be measured a certain distance away from it. Therefore, the pressure distribution at the source is computed based on the sound recorded at a known location. Despite the difference between the propagation directions, inverse and forward AH are ruled by the same equations.

For the interest of this project, which matches that of Sorama, only the inverse propagation is considered. The reason for this is that typically the objects under analysis exhibit a complex mechanical behaviour; thus, having a complete description of the pressure distribution within them is not achievable. A way to approximate such distribution is to measure the sound they emit and propagate it backwards to the source. Taking this into consideration, all the following references to AH only regard the Inverse Acoustic Holography.

2.1.1 Classification

Inverse Acoustic Holography (AH) backpropagates the pressure distribution employing different ways of representing this field. Following this criterion, there are two ways by which AH is computed: Space-based AH and Fourier-based AH. The space-based methods employ a spatial frequency representation of the pressure field, and by means of spatial convolutions the inverse propagation is achieved. On the other

hand, Fourier-based methods transform this spatial representation of the pressure field to the *K-space* domain, where a simple amplitude and phase shift is applied to achieve the same result (more information on the K-space domain is found in the next subsection). Due to the complexity and computationally intensity of the spatial convolutions, the Fourier-based methods are considered.

The Near-field classification criterion makes use of the concepts of *propagating waves* and *evanescent waves*, thus, a brief description of these is given first. The propagating waves are the ones propagating to the far-field, which means that their amplitude shows a no decay over distance. The evanescent waves, on the other hand and as their name suggest, are the ones whose amplitude decays exponentially as the wave propagates. After certain distance has been traveled, they are no longer distinguishable from noise. This is caused by adjacent regions in the object's surface having positive and negative velocity: they tend to cancel each other as they push against the fluid (air, in this case), failing to irradiate energy to the far-field. This condition, which gives rise to the creation of these waves, is also known as hydrodynamic short circuit [1]. Evanescent waves are therefore only found in the near-field.

Near-field AH (NAH) and Far-field AH (FAH) are AH techniques which base their calculations in the information measured in the mentioned fields. Both of them use data corresponding to the propagating waves, as these waves are found in both the near- and far-fields. However, the evanescent waves are only measurable in the near-field. In this sense, NAH achieves a much more detailed pressure distribution reconstruction that the one that could be obtained by beamforming (a FAH method), for example. Because a reconstruction as detailed as possible is required, the NAH method is considered from this point on.

2.1.2 Near-Field Acoustic Holography

Plane Wave Properties

Since the NAH algorithm is based on the analysis of plane waves, some of their basic properties are introduced first. Assume a planar wave, on an xy plane, propagating in a direction with an angle θ. Its wavelength, denoted as λ, projects a trace wavelength in both the x axis (λ_x) and the y axis (λ_y). The relationships between λ and its traces are:

$$\lambda = \lambda_x \cdot sin\theta, \tag{2.1}$$

and:

$$\lambda = \lambda_y \cdot cos\theta. \tag{2.2}$$

Figure 2.1a illustrates an example of such situation, where the λ projections (λ_x and λ_y) are depicted in thick lines on the $y = 0$ and $x = 0$ planes respectively. By looking at these projections and at Equations 2.1 and 2.2, it becomes evident that in general, the projections λ_x and λ_y can be larger than λ itself.

A planar wave's spatial frequency components, or wavenumber components, is expressed in terms of its wavelength's traces through the following expressions:

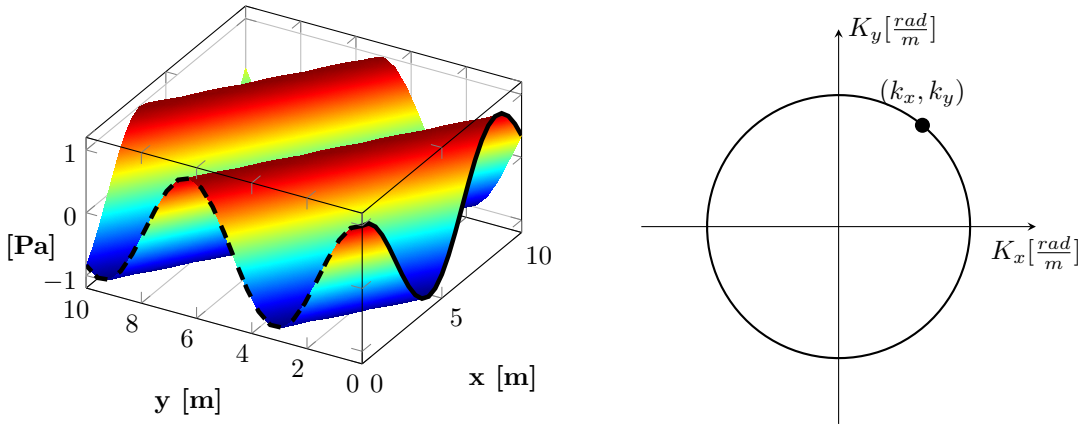
$$k_x = \frac{2\pi}{\lambda_x}, \tag{2.3}$$

and:

$$k_y = \frac{2\pi}{\lambda_y}. \tag{2.4}$$

These magnitudes, which are expressed in $[\frac{rad}{m}]$, are interpreted as the distance over which the phase of the wave increases by 2π when time is *stopped* [1]. Additionally, for a planar wave, the k_x and k_y components are used to compute the acoustic wavenumber:

$$k = \sqrt{k_x^2 + k_y^2}. \tag{2.5}$$



(a) Spatial frequency domain

(b) K-space domain and the radiation circle.

Figure 2.1: The same plane wave is represented in both the Spatial-Frequency and K-Space domains.

An example on the K-space magnitudes described in the last three equations is depicted in Figure 2.1b. The radiation circle's radius equals k, and the point, with coordinates (k_x, k_y) represents the exemplified planar wave. The waves lying within or on the radiation circle are propagating waves, whereas the ones outside of it are evanescent waves.

Theory

The theoretical foundations for the Fourier-based NAH were laid down in [1]. One of the most important assumptions made by NAH, which also adds the planar constraint for it (PNAH), is that a half source-free space is required. This means that the sources should be located in any xy plane which fulfills $z_s \leq 0$, and that all planes $z > 0$ are source free $(z, z_s \mid z \in \mathbb{R}^+ \bigwedge z_s \in \mathbb{R}_{\leq 0})$.

The acoustic wave equation is employed to model any infinitesimal pressure change relative to its equilibrium value:

$$\nabla^2 p(x, y, z, t) = \frac{1}{c^2} \cdot \frac{\delta^2 p(x, y, z, t)}{\delta t^2}, \tag{2.6}$$

where:

- $p(x, y, z, t)$ is the pressure value at a point in space (with cartesian coordinates x, y, z) at a given time t
- ∇^2 is the Laplacian operator defined as: $\nabla^2 \equiv \frac{\delta^2}{\delta x^2} + \frac{\delta^2}{\delta y^2} + \frac{\delta^2}{\delta z^2}$
- c is phase velocity of the wave. For sound this constant is $c_0 \approx 343 \frac{m}{s}$ in air at 20°C.

A physical interpretation of this equation, as provided in [2], is found in Figure 2.2. Considering pressure variations in a single spatial dimension, it is seen in this graphic that the sound pressure distribution can be analyzed either on time, provided a fixed position, or along the considered spatial dimension, provided a fixed time instant.

Sound linearity can be exploited to study its behaviour in the frequency domain. This imposes another important assumption on PNAH; namely, the sound sources should be stationary (i.e. producing a constant sound over time). To analyze the sound in this domain, Equation 2.6 needs to be expressed in terms of frequency. By applying the Fourier transform to it, the Helmholtz equation is derived:

$$\nabla^2 \overline{p}(x, y, z, \omega) + k^2 \cdot \overline{p}(x, y, z, \omega) = 0, \tag{2.7}$$

where:

- $\overline{p}(x, y, z, \omega)$ is the pressure value at a point in space for a certain angular frequency $\omega = 2\pi f$, where f is the sound wave frequency in Hz. In other words, this is the Fourier transform of $p(x, y, z, t)$.

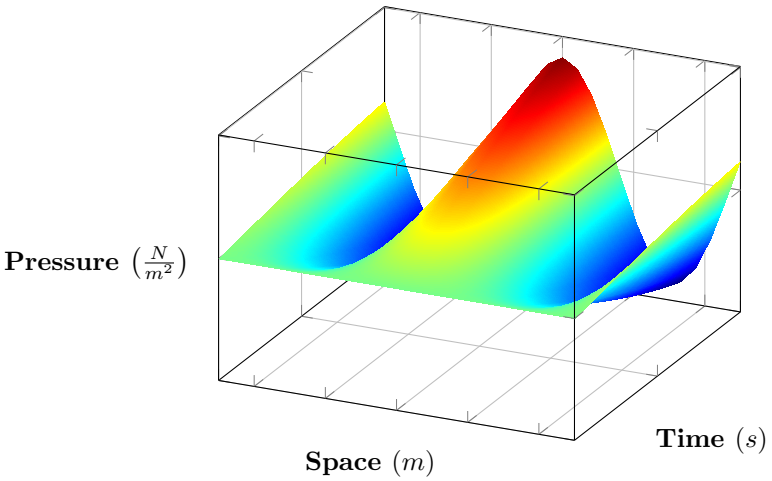


Figure 2.2: Spatial-temporal interpretation of the Wave Equation.

- $k = \frac{\omega}{c_0}$ is the acoustic wavenumber expressed in terms of the frequency.

A plane wave, described in terms of its spatial frequency $\overline{p}$, which could solve Equation 2.7 is:

$$\overline{p}(x, y, z, \omega) = A(\omega) \cdot e^{j(k_x x + k_y y + k_z z)}, \tag{2.8}$$

where:

- $A(\omega)$ is a frequency-dependent constant
- k_x, k_y and k_z are the wavenumber's components in each direction.

Equation 2.8 requires the following expression to be true:

$$k^2 = k_x^2 + k_y^2 + k_z^2. \tag{2.9}$$

The sensor spatial distribution provides the values of the k_x and k_y components, and k is obtained from the wavelength, which is derived from the frequency. In this sense k_z is chosen as the independent variable in Equation 2.9. Moreover, because of the source-free half-space assumption stated earlier, only the positive solutions for this variable are to be considered. Thus, the following expression is used, which in turn yields three different cases:

$$k_z = \sqrt{k^2 - k_x^2 - k_y^2}, \tag{2.10}$$

where:

a) $k^2 > k_x^2 + k_y^2 \iff$ Propagating wave

b) $k^2 = k_x^2 + k_y^2 \iff$ Wave travelling perpendicular to the plane (z direction)

c) $k^2 < k_x^2 + k_y^2 \iff$ Evanescent wave

In K-domain (reached after performing a second, $2D$ Fourier transform), the magnitude of k equals the radius of the *radiation circle*, which represents the region that marks the following differentiation criteria for the wavenumbers corresponding to the previous three cases:

a) Propagating waves lay within the radiation circle.

b) Perpendicular waves are exactly on the circle itself.

c) Evanescent waves lay outside the radiation circle.

Once the K-domain representation of the measured plane wave is obtained (via the spatial Fourier transform), it is backpropagated from the measurement plane ($z_h > 0$) to the source ($z = 0$) in this domain. The relation between these two planes is:

$$\tilde{p}(k_x, k_y, 0, \omega) = \tilde{p}(k_x, k_y, z_h, \omega) \cdot e^{-jk_z z_h}, \tag{2.11}$$

where $\tilde{p}(k_x, k_y, 0, \omega)$ is the K-domain representation of $\overline{p}(x, y, 0, \omega)$. An inverse spatial Fourier transform applied on $\tilde{p}$ yields the spatial frequency representation of the pressure distribution at the source plane $z = 0$.

2.2 PNAH Algorithm description

The algorithm to backpropagate the pressure distribution from the measured plane to the actual source begins with the data acquisition of the pressure variation at N_{ch} linearly spaced known points located in a single plane (where $N_{ch} \mid N_{ch} \in \mathbb{N}^+$ is the number of sensors collecting information). Such a linear spacing is required to properly compute the k_x and k_y components previously described. Since the measured pressure varies over time, and because the sensor location is known, these samples are considered to be in the *spatial-time* domain. As previously mentioned, a stationarity condition on the source is imposed with Equation 2.6, which might not be the case in several scenarios. For the algorithm to be able to cope with non-stationary sources, as proposed in [3], a stationarity assumption is made during all of the small time intervals serving as an input for the algorithm. Along this line, each of the N_{ch} sensors provides a number $N \mid N \in \mathbb{N}^+$ of input samples. These N samples are transformed from time- to frequency-domain.

Directly processing the N considered samples would be equivalent to employing a rectangular window in time domain, which has undesired effects in the frequency domain, such as spectral leakage. For this reason, weighting this input vector by a window function is required to *shape* the time domain signal [4]. Despite of altering the input time signals by applying a window on them, the corresponding frequency domain representation is not negatively affected because under the steady-state assumption on the input, the *least-affected* windowed samples still have enough information to yield an accurate representation of their relevant frequency components. Figure 2.3 illustrates the first stages of the PNAH algorithm.

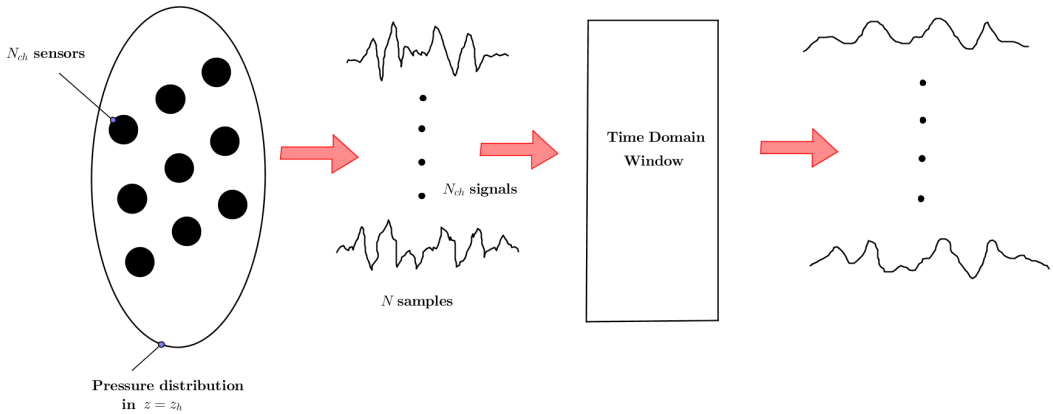


Figure 2.3: Spatial-time domain processing. The sensors collect pressure data at the measurement plane. This data is filtered.

Consider a single dataset consisting of N samples. Usually, after a domain transform is applied to these values, an equal number of frequency bins is obtained. However, not all of them are of interest for the user. In the case of the PNAH algorithm, typically only the pressure distribution related to a certain subset of frequencies is required to gain more insight on the behaviour of the analyzed system. Under this assumption, a subset of n frequencies is selected, where $n \mid n \in \mathbb{N}^+$, and usually $n \ll N$. The frequency components corresponding to each of these n bins are further processed, whereas the rest of them are discarded. Back to the time-to-frequency domain transform, this operation is applied to all of the N_{ch}

sets, each consisting of N samples. Because the spatial location of each sensor is known, the results of this transform are said to be in the *spatial frequency* domain. At this point, the spatial information of each sensor is assembled together. Every value out of the selected n frequency bins within the selected subset gets grouped together with the same bins corresponding to the rest of the spatially distributed sensors. As a result, a *hologram* consisting of N_{ch} *points* is created. Such hologram is a pressure distribution representation at a given frequency, and covers a determined region of space depending on the exact positions of the sensors on a plane. A total of n holograms, corresponding to the n frequency bins, are formed.

Once these holograms are created, they need to be taken to the K-domain, thus requiring another domain transform. To achieve this, an important preprocessing is required prior to each of the transformation of the n holograms. In the spatial-frequency domain, the finite physical region covered by the sensors is interpreted as a spatial truncation window [5]. Analogous to the previous domain transform, directly processing this spatially truncated window (hologram) would have negative effects on the K-domain representation (e.g. spectral leakage). Applying a non-rectangular window to the hologram aims at reducing the resulting discontinuities at the edges of the sampled region. Such discontinuities appear as an effect of the periodicity assumptions made by the Fourier transform algorithm and are erroneously interpreted as high wavenumbers in the K-space domain. An important difference between the time domain windowing and spatial domain windowing is the steady-state assumption. As mentioned in previous paragraphs, this assumption makes the spatial frequency domain representation *resilient* to the information loss incurred by the time domain windowing process. On the other hand, this assumption is not valid for the K-domain transform because the pressure distribution in the spatial-frequency domain is expected to vary depending on each sampled location. In other words, this distribution can follow any pattern along the finite measured region, and since no assumptions are made on the periodicity of this distribution, no information loss can be tolerated; every location contains valuable information. Since spatial frequency domain windowing slowly attenuates the measured values down to zero (at the edges), an extra step has to take place in order to avoid information loss due to windowing.

To keep an intact measured region after spatial windowing, the hologram needs to be extrapolated; the resulting hologram is interpreted as a measurement carried out in a larger region of space. In this way, the extrapolated information is weighted by the corresponding window coefficients, achieving two goals. First, the new holograms' values at the edges are zero and smoothly increase up to the original values corresponding to the measured region. This removes discontinuities and reduce spectral leakage in K-domain. Secondly, the information contained in the originally measured region remains intact, thus, an accurate K-domain representation of the wavenumbers contained in such a region is obtained. Once the hologram is extrapolated and windowed, the K-domain transform takes place. Figure 2.4 illustrates a general idea of the preprocessing done in the spatial frequency domain.

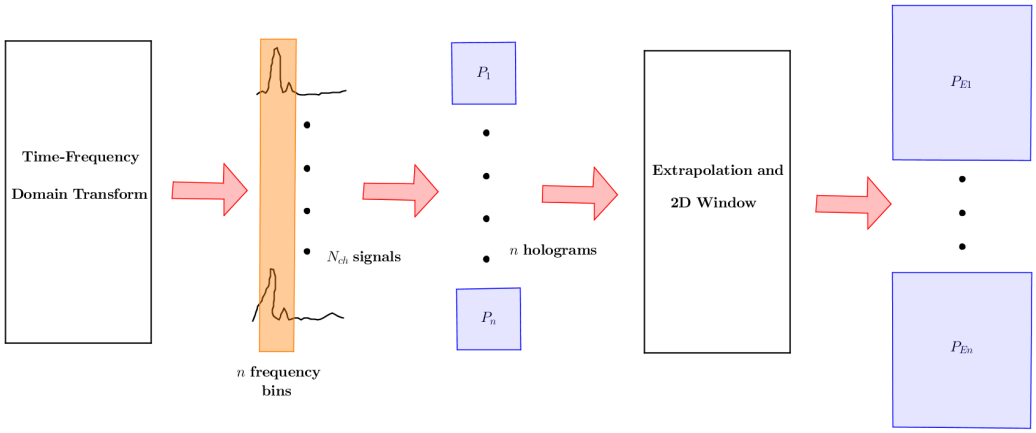


Figure 2.4: Spatial-frequency domain processing. From the resulting frequency spectra, n frequency bins are selected, from which n holograms are created. These holograms are then extrapolated and windowed.

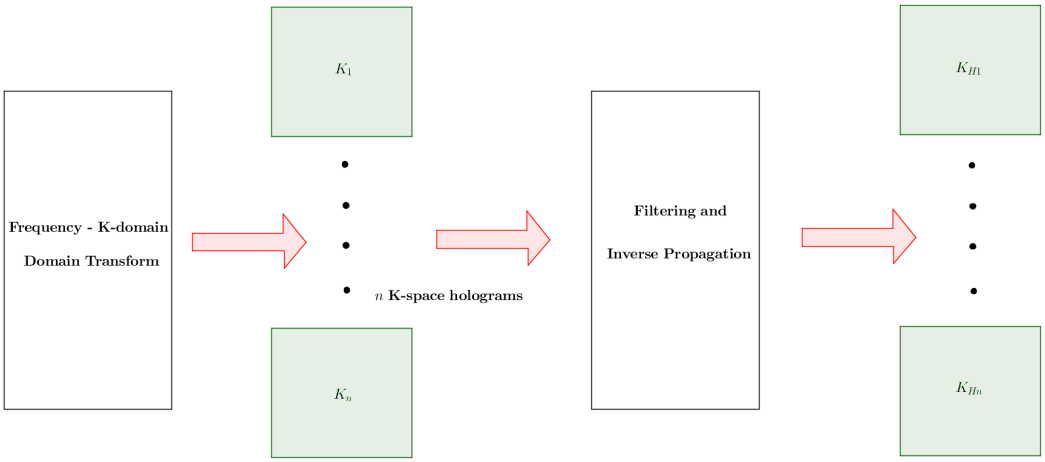


Figure 2.5: K-domain processing. After the domain transform, the high wavenumbers are filtered to avoid noise blow-up. Then, the information is backpropagated to the source plane.

The K-domain representation enables the measured pressure distribution to be easily backpropagated to the source plane by means of a phase shift and an amplitude factor correction. In the case of propagating waves (whose wavenumbers are located within the radiation circle in the K-domain representation), this factor is equal to one, thus, the amplitude is kept the same and only a phase shift is applied. In the case of evanescent waves, this factor is an exponential function depending on how *further away* from the radiation circle the evanescent waves are. Due to this exponential factor, measured noise whose K-domain representation also lies outside this circle is exponentially amplified. This potential noise blow-up represents an important problem, also extensively covered in [5]. To tackle this issue, a filtering stage is required prior to backpropagation; this filter should leave the propagating waves intact, while being able to discern between the evanescent waves and the measured noise. Achieving this, however, is very complicated; typically, such a filter represents a tradeoff between reconstruction detail and noise blowup. The *right* settings in this stage are very important for a correct interpretation of the results, and they usually depend on the system being analyzed and the measurement conditions. After the filter is applied, the phase shift and corresponding amplitude correction is performed. The result of this, is the K-domain representation of the pressure distribution at the source plane. These last stages are depicted in Figure 2.5.

Finally, to display the results in an understandable way, this K-domain representation needs to be transformed back to the spatial frequency domain. Therefore, a final, inverse domain transform is executed. The result is a hologram depicting the pressure distribution at the source. Because of the extrapolation done earlier in the algorithm, this hologram can also be thought of as covering a larger area than that of the actually measured. Thus, only the relevant area needs to be taken into account. Figure 2.6 contains a depiction of the last stage of the PNAH algorithm.

This chapter concludes after providing a brief description of both the PNAH theory and algorithm. The following chapter provides the implementation details that were abstracted out in this introduction, as well as the current performance metrics and proposed approaches to achieve the established goals.

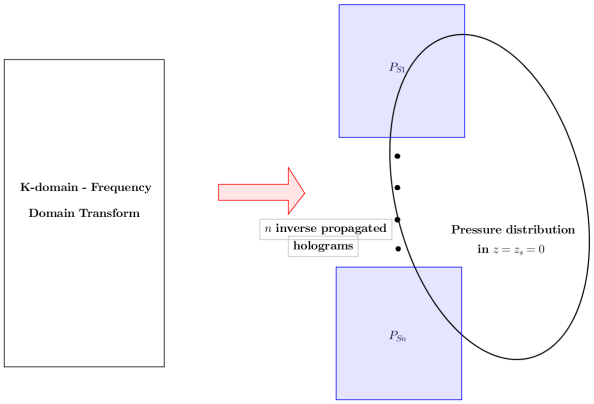


Figure 2.6: An inverse domain transform results in the n backpropagated holograms being represented in the spatial-frequency domain. In this domain, the holograms, which describe the pressure distribution at the source plane, are easier to interpret.

Chapter three

System Description

The objective of this chapter is to describe the current real-time implementation of the PNAH algorithm in terms of its *inner* functionality and achieved performance. Based on the measured performance, the taken approaches to address these issues are presented.

The remaining of this chapter is organized as follows: first, the implementation of each of the steps mentioned in the previous chapter, with additional details, is described. Then, the execution times of this algorithm is analyzed. Finally, based on these, the proposed approaches to reduce the execution times are given.

3.1 PNAH Implementation

The current implementation of the real-time PNAH algorithm is presented in this section. The set of steps comprising this algorithm is referred to as the *nominal implementation* [6] in the remainder of this document.

The execution of this algorithm begins with the data acquisition. The pressure variation is measured through its propagation by acoustic waves; thus, the sound in the z_h plane is measured employing a total of N_{ch} microphones. The hardware employed to do this task is the Sorama Cam, which is a microphone array consisting of $N_{ch} = 1024$ MEMS microphones arranged in a 32×32 grid. Each of these 1024 microphones is separated from its neighbours, in both X and Y directions, by two centimeters. The surface covered by the array is enough to span the target scanning area of the considered litography system; in this sense, this number of sensors is required. A picture of the Sorama Cam can be seen in Figure 3.1. Each of the microphones includes a $\Sigma\Delta$ analog-to-digital converter, meaning the the microphones' output is in the digital domain (more details on the $\Sigma\Delta$ converters follow in the subsequent section). Every sensor outputs a binary stream, where each bit corresponds to a measured sample. The microphones' sampling frequency, and, in consequence, the bitstream frequency, equals to $F_s = 1.5$ MHz.

The samples from all $N_{ch} = 1024$ channels are fed to an FPGA (whose characteristics are found in Appendix A), which applies a decimation filter on the samples for each channel. The objective of this filtering stage is to reduce the input sampling rate, to increase the accuracy of a single sample and to low-pass filter the input. This low-pass filtering is required because the $\Sigma\Delta$ AD converter has a noise-shaping property which *pushes* the quantization noise (error between the quantized and real magnitudes) to the high-frequency regions of the spectrogram [7]. The decimation filter consists of two different stages: a cascaded integrator-comb (CIC) filter followed by a compensation filter. The CIC filters, initially introduced in [8], belongs to the family of *moving-average* filters and quickly became popular because of its economic multiplier-less structure which enabled processing at a very high input rate. The advantages of this structure do not come without tradeoffs. CIC filters have two considerable issues: a non-flat frequency response and frequency aliasing. The non-flat frequency response is caused because CIC filters essentially apply a rectangular window in time domain, which causes a *sinc* function-like frequency response. On the other hand, aliasing is an effect of downsampling. To tackle

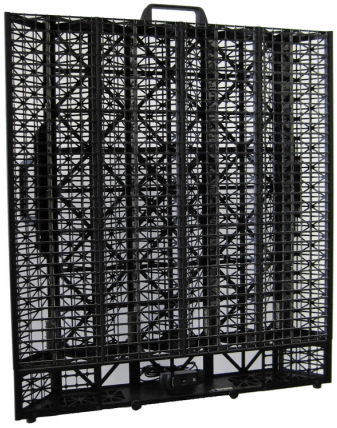


Figure 3.1: Sorama Cam - 32×32 microphone array.

these two problems, the compensation filter, which is actually an FIR filter, flattens (compensates) the frequency response and filters out any introduced aliased spectra. Since the decimation factor is $D = 32$, the output of the FPGA is a total of N_{ch} streams of 32-bit samples with a frequency of $f_s = \frac{1.5}{32}$ MHz $= 46.875$ kHz.

From this point on, the rest of the algorithm is executed in a GPU platform (whose characteristics can be found in Appendix A). Each of the following stages is implemented via an *OpenCL kernel*. OpenCL is a framework, developed by the Khronos Group, which is compatible with different hardware platforms and allows the execution of routines where parallelism is explicitly included by the programmer [9]. On the other hand, a kernel is a piece of code following the OpenCL specification that executes in parallel within the specified hardware platform. An OpenCL kernel can be thought of as the set of computations that are applied to a single element within a set.

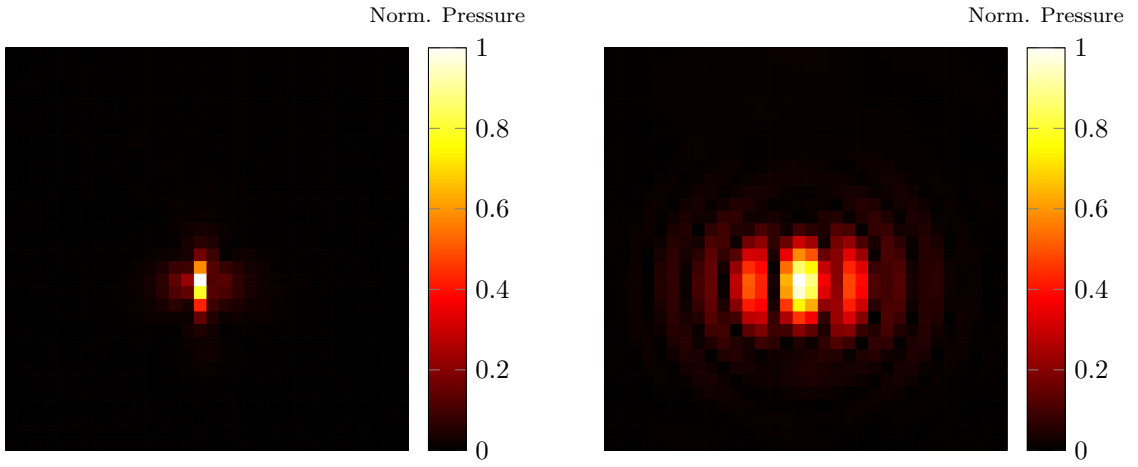
Resuming the algorithm description: for each of the $N_{ch} = 1024$ microphones, a total of $N = 1024$ samples are selected to be processed by the next stages. Considering the sampling frequency f_s, $N = 1024$ samples cover a time interval of approximately 0.022 s. The reason why N has this value is justified by three different reasons:

- As shown in [3], $N = 1024$ represents a time interval small enough in which the assumption of a stationary source can be considered valid. With this number, PNAH provides accurate results when analyzing transient phenomena.
- The achieved frequency resolution (the whole spectrum divided in a total of N frequency bins) is small enough to contain the *spectral leakage* effects under accepted levels.
- The FFT (upcoming step) executes faster and more efficiently when datasets whose size equals a power of two are considered.

Spectral leakage is a problem which arises after a domain transform mainly due to two different situations:

- The periodicity assumption by the domain transform algorithm causes that, for example, if the first sample of the set has a different value than that of the last one, an inexistant high frequency component is introduced, which *steals* energy from the remaining frequency bins.
- If the number of available frequency bins is too small, the energy corresponding to frequency components which are not properly represented might *leak* to neighboring frequency bins.

To address this issue, an FIR filter applied through a Hann window is employed to slowly attenuate the edges down to zero. Since both the window and its derivative are continuous, it shows a desirable behaviour on frequency: a faster mainlobe falloff and decreased sidelobe level [10]. Once this window is applied, the filtered time-domain data gets processed by the FFT algorithm to be transformed to the spatial frequency domain.



(a) Measured input hologram at $z = z_h$.

(b) Backpropagated output hologram at $z = z_s = 0$.

Figure 3.2: Normalized pressure distribution holograms of a tuning fork ($A4,\ 440$ Hz).

The domain transform is implemented through the *clFFT*, which is a software library written with the OpenCL specification that executes the FFT algorithm. This library, which was recently made open-source, exploits OpenCL's explicit parallelism capabilities to efficiently execute the FFT in hardware platforms containing several processing elements such as GPUs [11].

As only a certain subset of n frequencies is selected, for each of the N_{ch} channels, the desired n frequency bins are put together with the same bins from the remaining channels to form the hologram representing the pressure distribution for a given frequency. Figure 3.2a illustrates an example of a hologram *taken* at the measurement plane z_h. A total of n of these holograms are selected; the rest of the information outputted by the Fourier transform is discarded.

As described in the previous chapter, these holograms require to be extrapolated and windowed prior to being transformed to the K-domain. To address the extrapolation stage, a method presented in [5] and denoted as Linear Predictive Border Padding (LPBP) was initially used to compute signal values outside of the known area based on the measured data. This method first extrapolates the data in one direction (processing either rows or columns), and then the other direction is calculated. To increase the achievable parallelization in a GPU device, a Planar LPBP (PLPBP) was proposed in [6]. PLPBP computes the extrapolated values in both directions (X and Y) in a single calculation; thus, the columns do not depend on the values of the rows, or viceversa. Even though PLPBP has increased complexity, it achieves a better efficiency, for a limited subset of holograms, in parallel platforms such as the considered GPU. Once the n holograms have been extrapolated, a $2D$ Tukey window is applied on them such that the originally measured data remains intact, whereas the added samples slowly attenuate to zero. As a consequence, spectral leakage induced by edge discontinuities is avoided.

After these two preprocessing stages, the $2D$ FFT algorithm, also implemented with the clFFT library, is applied to the n resulting holograms. The result of this process is a wavenumber representation of all the propagating and evanescent waves and noise measured. The reason that the backpropagation cannot be directly performed is that noise whose wavenumbers lie outside the radiation circle would be exponentially amplified. For this reason, a low-pass filter must be applied; such filter must keep data within the radiation circle unaffected, and set a selection criteria for wavenumbers outside this region. This filter is determined via a Cut-Off and Slope iteration filter (COS), as proposed in [5], previous to the beginning of the PNAH algorithm execution in the GPU. Once it is derived, it is used to essentially window the K-domain representation for each iteration. The correct selection of this filter's parameters is crucial, since this represents a trade-off between reconstruction accuracy and amplified noise.

Once this filter is applied on all n holograms, the backpropagation is done via Equation 2.11. In this expression it is observed that a real k_z (propagating wave) causes a phase shift, whereas a complex k_z (evanescent wave) results in an exponential increase in their amplitude.

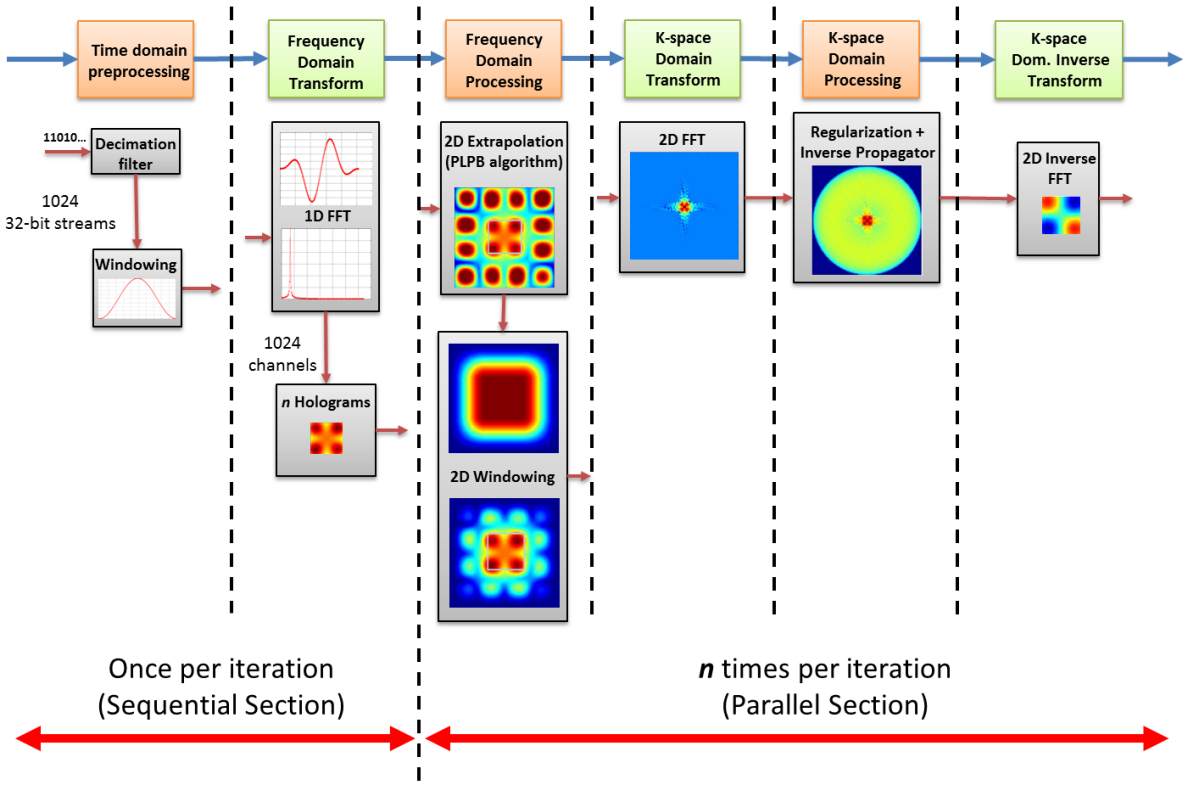


Figure 3.3: PNAH Stages executed per iteration.

After backpropagating the n holograms, a $2D$ inverse FFT (clFFT) is executed to obtain the spatial frequency representation of the pressure distribution in the plane of interest (usually $z = 0$). Because the holograms' dimensions were previously increased by the extrapolation process, the original measured region has to be cropped out of the n inverse FFT outputs. These holograms represent the output of the real-time PNAH algorithm. An example of such a backpropagated result is observed in Figure 3.2b. If the output resolution requires to be improved for interpretability, an additional *zero-padding* stage takes place previous to the $2D$ inverse FFT. This process simply consists of adding zeroes around the K-domain hologram, thus increasing the coarse resolution imposed by the relatively large space between the linearly spaced sensors.

To conclude this section, a summary of the stages comprising the real-time PNAH algorithm is found in Figure 3.3, where as the mapping of these to the hardware is depicted in Figure 3.4.

3.2 Performance Metrics

As previously mentioned, a GPU-based implementation of the real-time PNAH is presented in [6], where the goal is to implement a system that achieves a throughput of at least 1kHz. The cited source states that this goal is achieved when a total of $n \leq 10$ desired holograms are computed. In this sense, the execution times for the whole PNAH algorithm are shown in Figure 3.5a for a different number of n required holograms. Additionally, to get a better understanding of the execution times of each stage of the algorithm in terms of the total time, a relative plot containing this information is seen in Figure 3.5b.

By observing Figure 3.5b, it becomes clear that the hologram extrapolation (through PLPBP) is the routine which takes most of the time per iteration (around 63% of it); an important part of the work in [6] was dedicated to reduce this percentage. Besides PLPBP, the three domain transforms, specially the first $1D$ FFT, as well as the time domain filtering are the ones standing *next in line* as the stages which consume the most time. In the depicted cases, the $1D$ time-frequency FFT consumes between

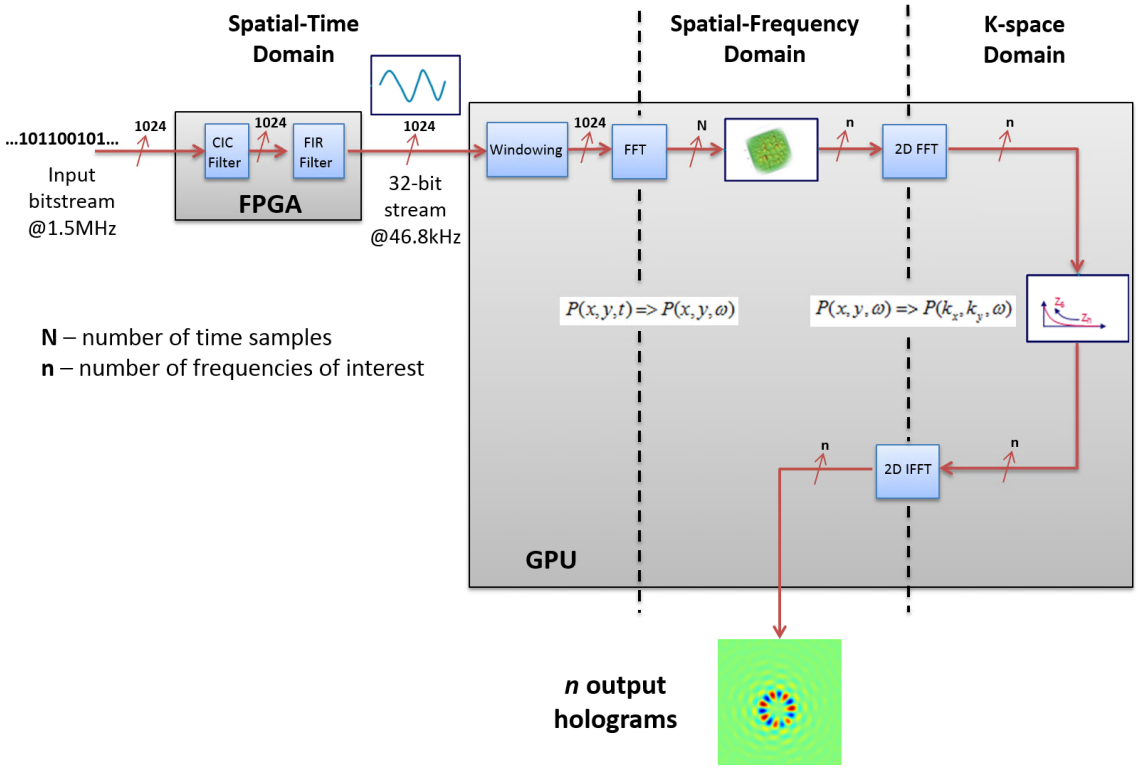
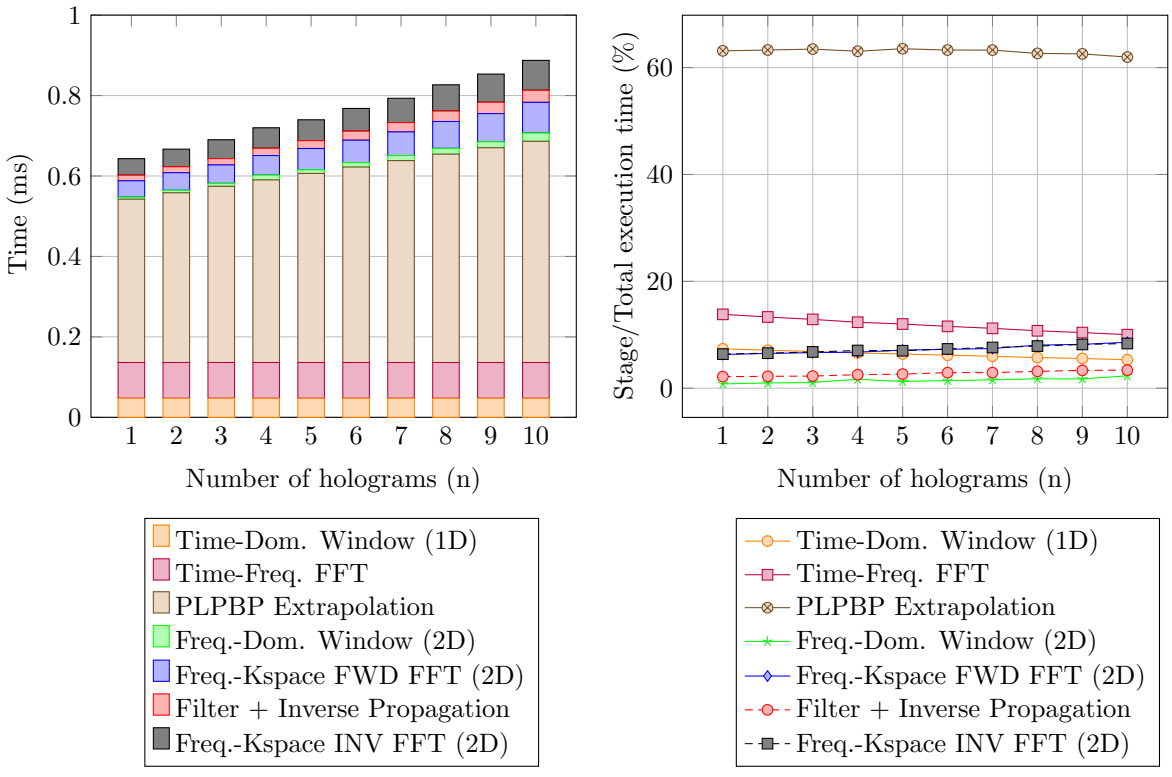


Figure 3.4: PNAH execution in hardware (*Nominal Implementation*).



(a) Absolute execution time.

(b) Per-stage relative execution time.

Figure 3.5: Execution time of a single iteration of the current PNAH implementation for a different number of holograms.

17% and 10% of the execution time, whereas the addition of both forward and inverse $2D$ FFTs consume between 11% and 16% of the time. Both the latter transforms, appearing at approximately the same points in the referred figure, exhibit a tendency to increase proportional to the number n of output holograms; the $2D$ transforms might pose a larger bottleneck than the $1D$ transforms in the future, that is, when more than 10 holograms are considered. For the time being, the time-to-frequency transform is, time-wise, more relevant since it takes more time than either the forward or backward $2D$ FFT.

3.3 Proposed Modifications

As mentioned above, important optimizations on the complex $2D$ extrapolation process were reported in [6]; thus, the proposed approach in this work is to address the other stages which, besides PLPBP, consume most of the time. Since the first domain transform, along with the processing required previous to it, consume a considerable amount of time, they are addressed together. In this sense, there are three proposed modifications to some stages of the PNAH algorithm in order to achieve the project goals. These approaches are discussed in the following paragraphs, but before describing them, it is worth mentioning that they are classified in two categories, depending on the format of the input data required by each. Therefore, they are regarded as Decimated-and-Filtered Input or Raw-Bitstream Input data approaches. Each of the three approaches is implemented employing the OpenCL framework in the GPU platform.

3.3.1 Decimated-and-Filtered Input

The approach falling within this category suffers does not alter the current dataflow. The microphone data input follows the same path as the current implementation (decimation filter in the FPGA + time domain windowing within the GPU). However, the stage which is modified is the $1D$ domain transform: instead of employing the FFT algorithm, the DFT approach is employed.

n Dot Products

The Discrete Fourier Transform (DFT) is a method to compute the Fourier coefficients of an input signal; because of this, its output is exactly the same as that of the FFT. Their main difference is that the complexity of the DFT is $O(N^2)$, where as the FFT's is $O(N \cdot logN)$, where N is the number of input samples. This is caused by the way each method computes the results. DFT is essentially a matrix-vector multiplication. In this context, the Fourier transform matrix, denoted as W_F and with a size of $N \times N$, consists of N rows containing N *twiddle factors*. The twiddle factors are complex numbers representing single samples of the complex sinusoids. Vector X has a size of $N \times 1$ and contains the input samples. The product of the Fourier matrix and the input vector takes N^2 multiplications and $N \cdot (N-1)$ additions.

On the other hand, the FFT is a divide-and-conquer based approach in which the data gets reordered and grouped in *butterfly* patterns such that the staged-multiplication in every regrouping phase saves several operations (both multiplications and additions) [12]. This causes its complexity to be considerably low when compared to the DFT, which in turn also popularized it. More information on the FFT algorithm follows.

Up to a sufficiently small number of desired frequencies (n_x), computing these through a modified DFT method, named as *n dot products*, should need less time than executing the FFT algorithm. This number is approximated by looking at the estimated number of operations incurred by each algorithm. The n dot products approach requires around $n \cdot N$ operations, whereas the FFT algorithm executes a quantity of operations proportional to $N \cdot log_2 N$. Therefore, this approach should be beneficial when the following constraint holds:

$$n \cdot N < N \cdot log_2 N, \tag{3.1}$$

and n_x is found when the following holds:

$$n_x = log_2 N. \tag{3.2}$$

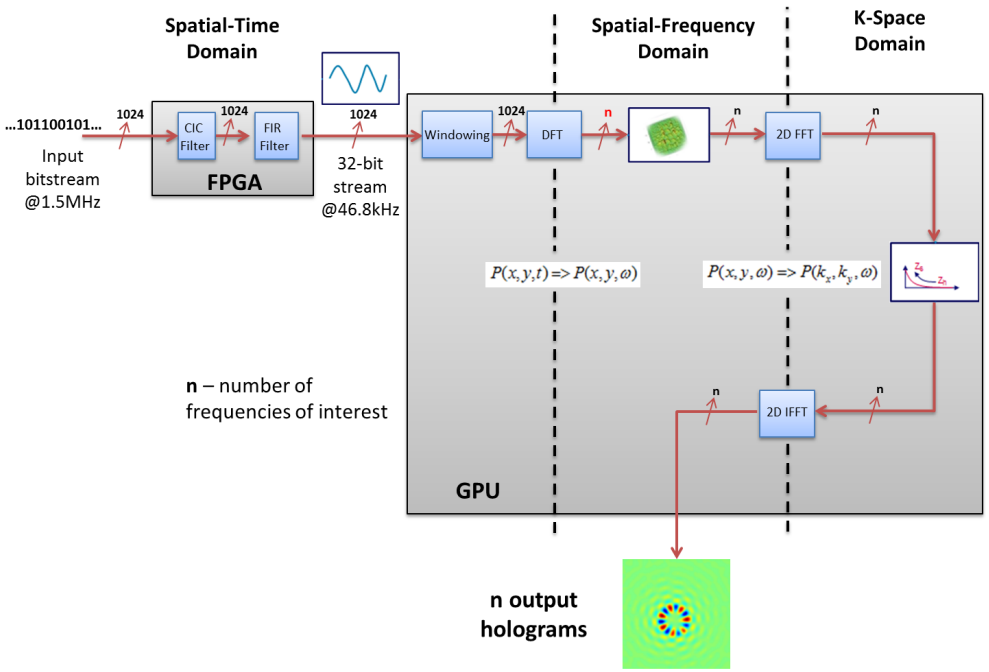


Figure 3.6: Modifications for the n *Dot Products* approach. Note that the output of the *DFT* block is a set of only n frequency bins per channel, instead of all the N frequency bins from the nominal approach.

Only the needed n (where $n \leq n_x$) results are computed. There are two important aspects to consider when employing this approach. First, the efficiency of computing the n dot products is considerably low taking into account that only n frequency bins are calculated ($n \ll N$). However, for the scope of this project, the execution time is more relevant than the computational efficiency, and since high computational efficiency does not necessarily mean low execution time, this approach is valid. Additionally, the n dot products approach requires less memory at the output when compared with the FFT. Second, this approach is only applicable when the expression $n \leq n_x$ is true. It is important to find the value of n_x because, if $n > n_x$, then it is faster to execute the FFT algorithm instead and discard the unneeded results.

As a conclusion, the n dot products approach is expected to have a positive impact on the PNAH execution times as long as the condition $n \leq n_x$ is true, and the smaller the value of n, the better. Figure 3.6 depicts the modifications for this approach.

3.3.2 Raw-Bitstream Input

Since the last two approaches rely on digital signal processing (DSP) being performed on binary streams, initially, some aspects related with this topic are given.

Historically, one of the main factors which popularized DSP techniques applied on bitstreams was the use of the Direct Stream Digital (DSD) method employed by Sony and Philips to reconstruct audio signals from digital streams of data. The objective of this approach was to move away from the standard CD-format (16-bit resolution and sampling frequency $f_s = 44.1$kHz). The reason to do this was that the analog circuitry to do the proper filtering was to slow and expensive [13]. In general, since digital circuitry has become cheaper and faster than its analog counterpart, there has been a tendency to *push* the digital domain closer to the system front-end, which was typically an analog circuitry domain.

As mentioned earlier in this document, the output of the MEMS microhpones comprising the Sorama Cam is a digitally-encoded stream representing the measured sound. These devices oversample their input to economically avoid signal distortion caused by aliasing and to enable resolution increasing on a later decimation stage. Avoiding aliasing is achieved because sampling at frequencies higher than that of Nyquist's widens the transition band of the subsequent filter. On the other hand, increased resolution

is obtained by decimation, which can be thought of as a basic form of averaging: the number of binary 1's in a signal is proportional to its value in a certain period of time [14]. Table 3.1 contains an example where this concept is illustrated.

16 1-bit values																**Decimation**	**Average**
1	0	1	0	0	0	1	0	1	1	0	0	0	1	0	1	16:1	$\frac{7}{16}$

Table 3.1: Signal averaging based on the decimation process.

A similar bitstream is the output of the ΣΔ analog-to-digital conversion process, although it is worth mentioning that binary 0's represent a sampled amplitude of -1, and binary 1's represent measured amplitudes of $+1$. The ΣΔ converter is based on Δ-modulation. This modulation quantizes the signal change instead of encoding the absolute value at each sample: the system tries to predict whether the next sample's value will be larger of smaller. The output of this modulation is a binary stream, where a $+1$ means that the sample increased its value with respect to the previous one, whereas a -1 concerns the contrary case. The problem with this modulation is that in case of rapidly-rising or -falling (high frequent) signals, the output shows a slow response since it cannot *catch up* with the signal value: Δ-modulation shows slope overloading. To correct for this issue, an integrator (represented by Σ) is placed before the modulator to *smoothen* the modulator; therefore, the ΣΔ convention. The main difference between these, is that ΣΔ-modulation encodes the integral of the signal, making its behaviour frequency-independent [14]. Additionally, its internal structure *shapes* the quantization noise, sending most of it to the high-frequency bands, which are anyway filtered out.

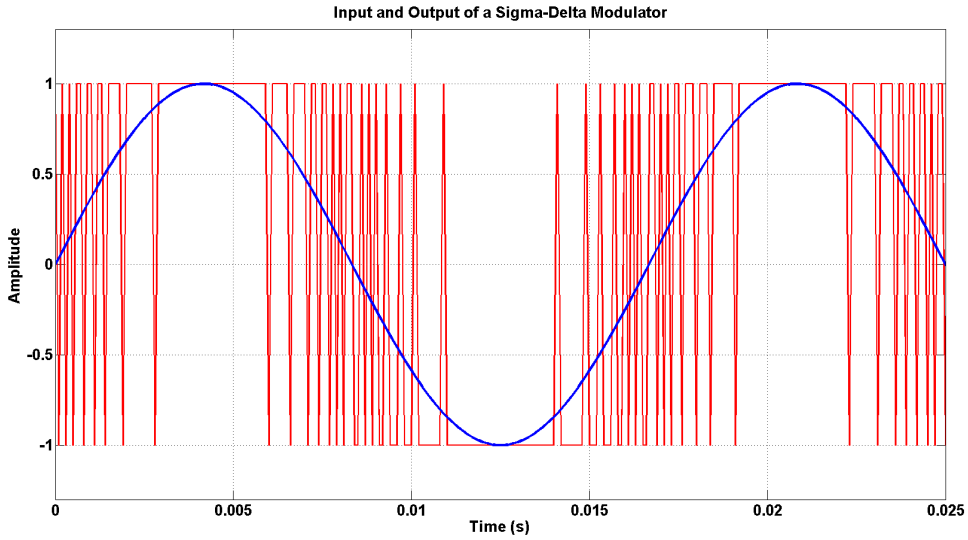


Figure 3.7: ΣΔ-modulator input (blue) and output (red).

Figure 3.7 depicts an example of the output of a ΣΔ-modulator when a sinusoidal wave serves as its input. The noise shaping effect is observed mainly in the areas where the input approximates a value of zero. Here, the output switches quickly between the -1 and $+1$ full scales, therefore introducing high frequency components. This behaviour is attenuated as the input approaches its either crest or valley. Despite these introduced high frequencies, the underlying low-frequencies are easily appreciated.

From the previous paragraphs, an important aspect needs to be summarized in order to justify the remaining two proposed approaches for this project. Namely, as exemplified in Table 3.1 and illustrated in Figure 3.7, the average of the output for a certain period of time is proportional to the input value within the same interval.

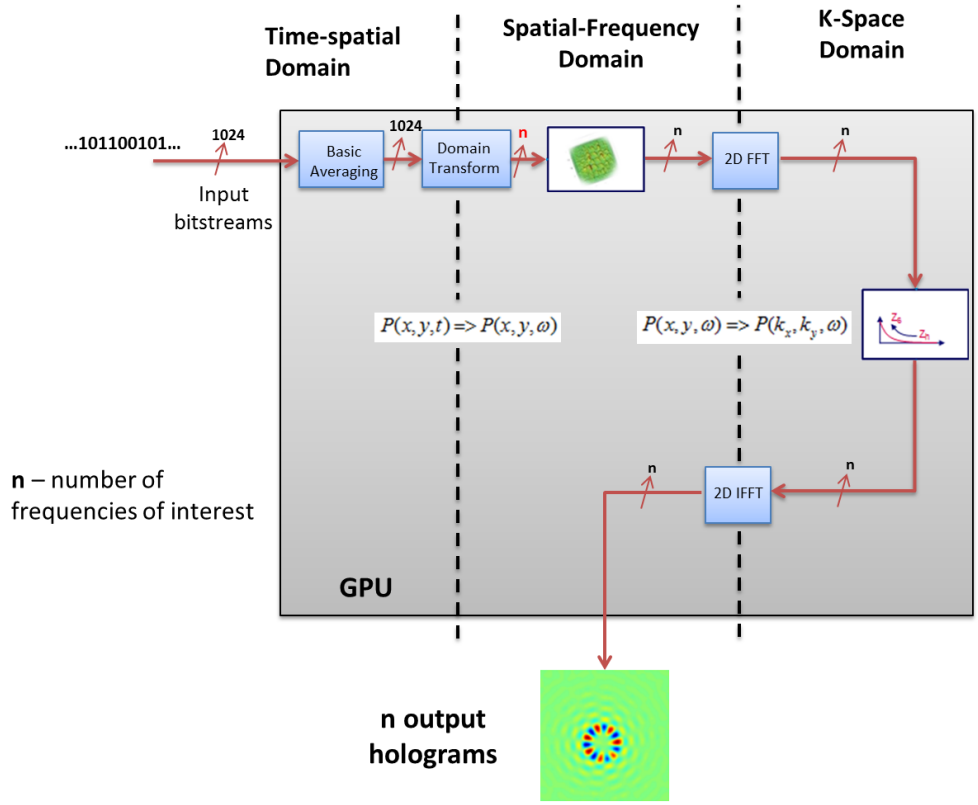


Figure 3.8: Modifications for the *Basic Averaging* approach.

Basic Averaging

The raw bitstream input can be decimated in a very basic way just by defining groups consisting of a number D of bit samples and averaging over these periods. The average can be carried out by counting the number of binary 1's and dividing the result by D, as exemplified in Table 3.1. The benefits of this approach are expected to have less impact than the other two proposed because only the decimation filter would be bypassed. However, along with the next approach, it is beneficial in terms of system integration since the whole dataflow would take place only in the GPU platform. Figure 3.8 depicts the modifications for this approach.

Walsh-Fourier Transform

Because the average, over a given time interval, of the $\Sigma\Delta$-modulator's output is proportional to the input signal's value, and taking into account that averaging is a basic form of low-pass filtering, applying a domain transform on the modulator's output itself would be similar to transforming an unfiltered dataset. Despite of high frequency noise being added, the frequency range of interest (typically $[0, 10\text{k}]$ Hz) remains well below the beginning of the region where quantization noise is *pushed* to as a consequence of the $\Sigma\Delta$-modulator's noise shaping property. Having in mind that the currently implemented decimation ratio is $D = 32$ and the number of 32-bit samples is $N = 1024$, a domain transform on $N_b = D{\cdot}N = 32768$ 1-bit samples should be performed in order to cover the same time interval. The objective of this approach is to reduce the PNAH execution time as a consequence of:

- Bypassing the decimation filter and time-domain windowing
- Computations applied directly on binary values
- Computing only the n desired frequency bins

Initially, the increased number of binary samples to process would have a negative impact on the Fourier transform regarding execution time. This effect is caused because both the number of multiplications

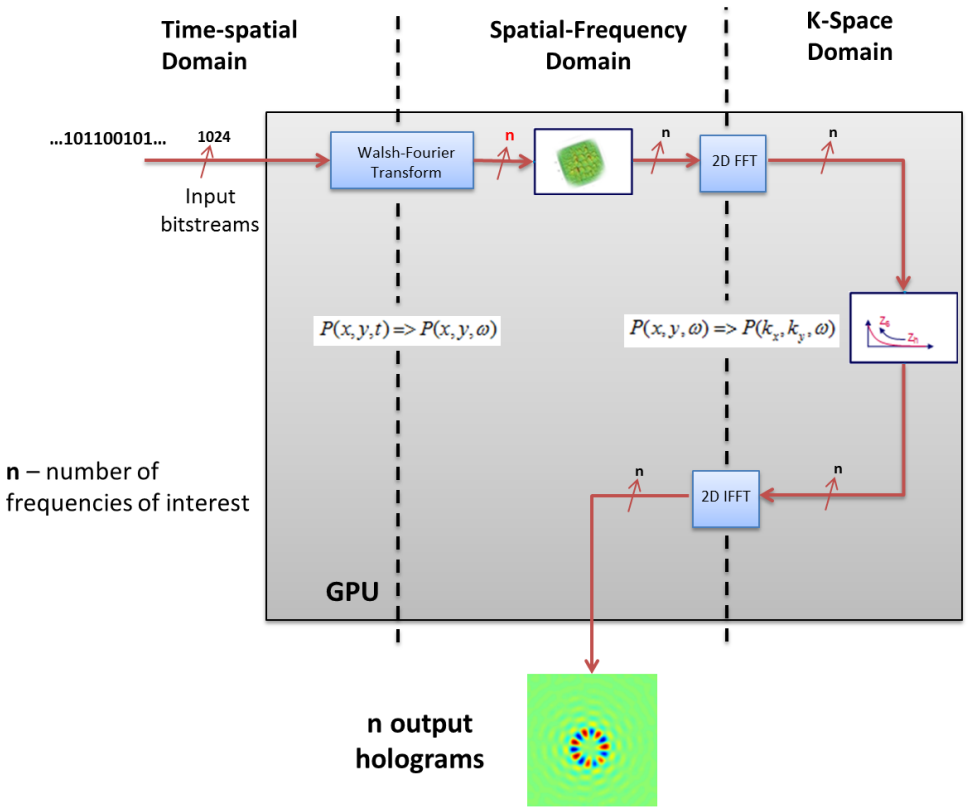


Figure 3.9: Modifications for the *Walsh-Fourier Transform* approach.

and memory requirements are increased by a factor of D. As for the operations, each of the N_b samples need to be multiplied by its corresponding twiddle factor. On the other hand, the required memory is increased since the product of multiplying each of the binary -1 or $+1$ values by the twiddle factors needs to be represented with a precision higher than 1 bit (usually 32 bits). The execution time of the $1D$ FFT stage was measured in a small experiment, where this algorithm was applied on $N_{ch} = 1024$ sets of $N_b = 32768$ samples and took approximately 9.6ms, roughly 108 times as much time as compared to the nominal implementation.

An alternative to solve the increased time issue is to compute some of the Fourier coefficients (the n required coefficients) via the Walsh transform. More details on this transform follow in subsequent chapters; for now it should suffice to mention that, due to the nature of the Walsh functions, this transform is well suited to operate on binary streams. Figure 3.9 depicts the modifications for this approach.

Having described the current system implementation and the proposed solutions to achieve the project goals, this chapter concludes. Since the first two proposed approaches employ the Fourier transform, the next chapter is dedicated to this transform. After it, the upcoming chapter provides more information on the Fourier coefficients via the Walsh transform (namely, the Walsh-Fourier transform).

Chapter four

Fourier Transform

The PNAH algorithm is referred to as a Fourier-based method because the pressure field backpropagation is carried out in the K-space domain, which is reached via a Fourier transform. One of the most important reasons to do it this way is because backpropagation in K-space is done with a multiplication, whereas in the spatial frequency domain, a convolution is required. The latter operation requires a considerable number of operations, making it an unviable solution. Since the inverse propagated information is not directly interpretable in the K-domain, it still has to be transformed back to the spatial frequency domain. Therefore, both forward and backward Fourier transforms are required. Additionally, a third Fourier transform, which actually comes first in the dataflow, is required to bring the measured time-dependent pressure distribution to the frequency domain.

It is reasonable to state that the PNAH algorithm heavily relies on the Fourier transform. An important reason for this is that the Fourier domain provides a direct physical interpretation: after the transform, the input data is expressed in terms of basic sinusoidal waves, which provide both amplitude and phase information of the input's projection onto the kernel functions.

This direct interpretation has also boosted the development of optimizations to efficiently compute the result of a discretized Fourier transform. For example, take the FFT algorithm, the most popular method to compute these results. This algorithm has been subject to a lot of research and optimizations regarding its inner data orderings and permutations. Furthermore, several platform-specific enhancements have taken place such that, when mapped to said hardware architectures, the FFT yields its results in less time and/or requiring less memory space when compared to other base implementation. As expected, most of these specific optimizations are based on assumptions regarding the computer's memory management and native instruction sets.

Since the amount of time and effort along the years spent on the enhancement of the FFT has been too high, it is difficult to expect considerable gains on execution time by modifying or implementing an own version of this algorithm; this might even be counterproductive. Additionally, computing only a subset n of desired frequencies through the FFT algorithm is in principle not an option due to the FFT's internal function. As a workaround for this issue, FFT pruning can be proposed, but its effects become considerable only when a large number N of samples is considered [15]. This number is larger than the currently employed one.

An alternative to this issue is proposed in this chapter, but first an introduction to the related theory, Fourier kernel functions and transform is provided.

4.1 Background concepts and definition

4.1.1 Complex Sinusoids

The Fourier vector space is based on a set of orthogonal functions onto which a given input function is projected, therefore obtaining a frequency domain representation. This set of functions is infinite, and each of its elements is described via the Euler's equation:

$$e^{j\cdot 2\pi ft} = cos(2\pi ft) + \big(j \cdot sin(2\pi ft)\big), \tag{4.1}$$

where:

- f is the sinusoid frequency in Hz., and $f \in \mathbb{R}$
- t is a certain time instant in seconds, and $t \in \mathbb{R}_{\geq 0}$
- j is the imaginary unit $j = \sqrt{-1}$.

4.1.2 Fourier Series

When an input function $x(t)$ is projected onto this vector space, it is represented in terms of a possibly infinite set of basic complex sinusoids described by the Euler formula, which is [16]:

$$x(t) = a_0 + \sum_{n=1}^{\infty} \Big(a_n \cdot cos(2\pi ftn)\Big) + \sum_{n=1}^{\infty} \Big(b_n \cdot sin(2\pi ftn)\Big), \tag{4.2}$$

where the coefficients $\{a_0, a_n, b_n\}$, known as the Fourier coefficients, are computed with the following expressions:

$$a_0 = \frac{1}{T}\int_T x(t)dt, \tag{4.3}$$

$$a_n = \frac{2}{T}\int_T x(t) \cdot cos(2\pi ftn)dt, \tag{4.4}$$

$$b_n = \frac{2}{T}\int_T x(t) \cdot sin(2\pi ftn)dt, \tag{4.5}$$

where T is the period $T = \frac{1}{f}$.

The complex Fourier coefficient c_n is related to the previously introduced ones by the following expression:

$$c_n = \frac{a_n + j \cdot b_n}{2}, \tag{4.6}$$

and is calculated with the following expression, which is derived employing Equations 4.1, 4.4, 4.5 and 4.7:

$$c_n = \frac{1}{T}\int_T x(t) \cdot e^{j\cdot 2\pi ft}dt, \tag{4.7}$$

In this sense, the complex Fourier series is expressed as:

$$x(t) = \sum_{n=-\infty}^{\infty} \Big(c_n \cdot e^{j\cdot 2\pi ft}\Big) \tag{4.8}$$

4.2 Fourier Transform

In practical applications, an infinite representation is not possible. Initially, the input function is discretized in a total of N sampled points. Assuming a sampling frequency of fs Hz, the samples equal the signal values $x(t_i)$ at time $t_i = \frac{i}{f_s}$, where $i \in \mathbb{N}_{\geq 0} \bigwedge i < N$. The interval between each sample is $T_s = \frac{1}{f_s}$. An assumption about signal periodicity is made here: it is assumed that outside the time interval covered by the N points, the signal is periodic.

An approximation of the projected sampled signal is achieved by computing only a finite subset of the infinite set of complex Fourier coefficients denoted in Equation 4.8. These coefficients correspond to the sinusoid with the fundamental frequency and to those with a harmonic frequency. A harmonic of a certain frequency is an integer multiple of it. On the other hand, the fundamental frequency is determined by the number of input samples taken and the sampling frequency at which these were taken.

The frequency band $[0, fs]$ is divided into N *slices*, or frequency bins. Frequency $f_0 = 0$ corresponds to the DC component, or the signal offset with respect to the x axis. Frequency $f_1 = \frac{fs}{N}$ Hz. corresponds to the fundamental frequency, and the rest of the bins, with frequencies $\{\frac{2 \cdot fs}{N}, \frac{3 \cdot fs}{N}, \ldots, \frac{(N-1) \cdot fs}{N}\}$, are its harmonics.

All of these N frequencies are used together with Equation 4.1 to create N complex sinusoids with different frequencies. Moreover, each of these N sinusoids is sampled at a total of N time instants $t = t_i$. Therefore, a complex $N \times N$ matrix, denoted as FT_N, is created containing the sampled harmonically-related complex sinusoids. This matrix is the Fourier transform matrix, and these samples are often referred to as the twiddle factors.

4.2.1 Discrete Fourier Transform

The Discrete Fourier Transform (DFT), denoted as F_N, of a sampled input X_N of size $N \times 1$ is then computed by multiplying the twiddle factor FT_N matrix above described by the input column vector:

$$F_N = FT_N \times X_N \tag{4.9}$$

where each element i of the resulting column vector $F(i)_N$ represents the complex Fourier coefficient c_i from Equation 4.7.

4.2.2 Fast Fourier Transform

As observed from Equation 4.9, the DFT has complexity $O(N^2)$ since each of the N output elements requires N processing stages. The complete operation requires N^2 multiplications and $N \cdot (N-1)$ additions. As this approach is very expensive, most of the times the Fast Fourier Transform (FFT) algorithm is employed. The *fast* qualifier refers to this algorithm having a complexity proportional to $O(N \cdot log_2 N)$.

The way the twiddle factor FT_N matrix is organized allows the use of a *divide-and-conquer* paradigm, which in general is exploited by all FFT algorithms. Considering inputs whose size is a power of two ($N = 2^m$), the main steps to be followed by the FFT algorithm are the following [17]:

1) Divide the problem in two subgroups, each with half the size as the original.
2) Recursively employ the same algorithm to solve for each subgroup. Perform this step until the size of the subgroups is equal to one.
3) The solution to the original problem is obtained by combining the solutions of all subgroups.

The FFT algorithm is actually a *family* of algorithms; all of its members follow the divide-and-conquer paradigm. However, there are some variations regarding the ways the input or output results are stored,

and how the intermediate results are managed, to mention some of them. These specific *tunings* might be beneficial for specific kinds of hardware architectures, for example.

The idea of dividing the initial problem into subgroups leads to a scenario where, at its lowest level, two samples are weighted by a given coefficient (twiddle factor) and then added to produce two outputs. Because of the pattern shape described by this computation dataflow, this operation is usually referred to as a *butterfly*.

When the lowest level of the tree has been reached, the computations consist on grouping larger numbers of samples together and perform the butterflies on them, until the point where the original problem size is reached. It is because of this *divide* paradigm that a tree structure is followed, therefore accounting for the $log_2 N$ factor in the algorithm's complexity.

Once the final stage of the computation is achieved, the results are usually *scrambled* when compared to the ordering the input had at the beginning. To account for this, there is usually a bit-reversal stage prior to the input or after the output has been computed. Since bit-reversing either one of them is computationally intense, some modifications to the FFT algorithms have been proposed, such as the Stockham Autosort framework (employed by the clFFT library [11]), to avoid this step [12].

When an FFT algorithm is executed in a parallel platform, such as the considered GPU, all elements within a certain *divide*-level (tree depth) are computed in parallel. Since all the tree levels are required to compute the output, pruning the results of the FFT (in order to extract a few n desired coefficients) does not result in considerable gains, because computing the rest of the undesired outputs comes *for free*.

4.3 Proposed Application

There is a number $n_x \in \mathbb{N}^+$ for which employing the matrix multiplication paradigm of the DFT requires less computation time than executing the FFT algorithm. More specifically, computing only a subset n of desired frequencies via dot products, although less computationally efficient, takes less time.

The proposed approach to this is to create an efficient method to compute the n dot products between the n sampled complex sinusoids and the input vector. The dot product operation consists of two steps:

1) Perform an element-wise multiplication between both vectors.
2) Perform a reduction operation (add all the multiplication results together).

The idea is to efficiently implement these operations in the GPU and measure its effect on a different number of n computed frequencies. More details on the implementation of the proposed GPU kernel follows in the next chapters, along with the obtained results.

Chapter five

Walsh-Fourier Transform

As mentioned in previous chapters, after the one-dimensional time-to-frequency domain transform, the PNAH algorithm selects only a few n Fourier frequency bins of interest, discarding the rest of them. Typically, $n \ll N$, where N represents the transform size. Because only n frequency bins are required, and to take advantage of the binary nature of the microphone raw data, the Walsh-Fourier method is proposed. This method, introduced in [18] and later retaken in [19] expresses a certain subset of Fourier coefficients in terms of Walsh coefficients. Essentially, it applies the Walsh transform to the input, and then, through a linear combination of some of the resulting Walsh coefficients, the required Fourier coefficients are computed. One constraint for this method is that the input data can only be real numbers; thus, it is only suitable for the first domain transform. Details on the causes of this constraint are provided in the subsequent sections.

One of the most relevant motivations that the cited sources had to propose this approach was that, in those years (end of the 60's, mid-70's), multiplications were hardware- and timewise very expensive. This approach would reduce the amount of required multiplications to compute the Fourier coefficients because the Walsh transform is based on additions and substractions. Additionally, this would be a reasonable way of pruning the Fourier transform results for applications requiring a small number n of Fourier coefficients. Such pruning is not efficient when the FFT algorithm is being employed. Nowadays, multiplication is no longer an expensive operation. However, because of the nature of the Walsh functions, multiplication of the raw binary data with said functions is reduced to a bitwise XOR operation; more details on this follow. Using this method requires the decimation and filtering stages to be bypassed. As a consequence, the overall execution time can be reduced. The inherent tradeoffs of the Walsh-Fourier transform are discussed in the upcoming chapters. Since this method clearly relies on the Walsh transform, an introduction to it is initially provided.

The Walsh transform is a nonsinusoidal orthogonal transform. As it name suggests, it is based on the Walsh functions, which take up only two different values, namely: $\{+1, -1\}$. This transform keeps certain analogy with the Fourier transform in the sense that both of them represent a given function in terms of a set of orthogonal functions. However, as opposed to the latter where a frequency domain representation is obtained, the Walsh transform expresses the result in a *sequency* domain; the proper introduction to this concept is found in the upcoming section.

As mentioned above, the binary nature of the Walsh kernel functions enables this transform to be carried out without multiplications, a formerly expensive operation. This was the reason why the Walsh transform was very popular between the 1960's and 1980's. Eventually, the frequent hardware improvements, the further optimizations to the FFT algorithm, and the direct physical interpretation of the frequency spectrum drew the attention back to the Fourier transform. Yet, current applications of the Walsh transform are found in biosignal compression and processing, communication protocols, and in automatic test pattern generation (ATPG) for integrated digital circuits, for example. In the first case, the acquired biosignal, typically an electrocardiogram, is processed into a filtered version of the signal by keeping only the Walsh coefficients where most of the signal's energy is stored, while still allowing pattern recognition. Regarding communication applications, the Walsh transform is used in the Code-division-multiple-access (CDMA) protocol to encode the different messages to be transmitted.

Finally, in automated test pattern generation (ATPG) techniques aiming to verify the functional correctness of digital circuits, the Walsh transform is employed to generate binary input vectors which have higher probability of causing faulty behaviour in said systems [20].

The remainder of this chapter is organized as follows. First, the necessary concepts and definition of the Walsh transform is provided. Then, its relation with the Fourier transform, namely, the Walsh-Fourier transform, is described, and finally, its proposed application on the raw data is given.

5.1 Background concepts and definition

5.1.1 Sequency

Regarding the sinusoidal kernel functions within the context of the Fourier transform, the term *frequency* is employed when referring to periodic functions whose zero-crossings over time follow an homogeneous distribution [16]. This frequency parameter can be thought of as the amount of full cycles (or the number of zero-crossings divided by two) achieved by said periodic sinusoids in a given period of time.

The generalized frequency, or *sequency*, is a broadened definition of frequency in the sense that the function's zero-crossings need not necessarily be uniformly spaced in time, nor necessarily periodic. As frequency uses Hertz (Hz) to express its units, sequency has employed the term *zps* to express the zero-crossings per second. Making use of this, a waveform's sequency is the rounded-up result (ceiling) of the *zps* divided by two. A rectangular waveform is characterized by its sequency just as a sinusoid is described by its frequency. Figure 5.1 illustrates an example of this characterization with a given waveform.

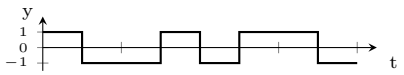


Figure 5.1: Rectangular waveform with a total of 5 *zps*; thus, a sequency of $s = 3$.

5.1.2 Rademacher Functions

The Walsh functions, which serve as the kernels for the Walsh transform, are generated from a set of incomplete orthonormal functions, known as the Rademacher functions. These functions, while also being rectangular waveforms taking up the values $\{+1, -1\}$, differ from the Walsh functions in that they are periodic pulses with a total of 2^{i_R-1} cycles in the interval $[0, 1)$. Here the index i_R uniquely characterizes each given Rademacher function $\text{RAD}(i_R, t)$, where t represents the time. The exception to the pulse periodicity is the first function $\text{RAD}(0, t)$, which is simply the unit pulse. To better illustrate these functions, Figure 5.2 contains the first five of them. It is worth noting that the characterizing index i_R does not represent the Rademacher function's sequency, nor its number of zero crossings.

One way of generating any i_R Rademacher function is by employing the following formula:

$$\text{RAD}(i_R, t) = sgn\left[sin\left(\left(2^{i_R+1}\pi t\right) + \frac{\pi}{N}\right)\right], \tag{5.1}$$

where:

- The total number of samples in the function is N, and $N = 2^m$
- The Rademacher characterizing index i_R satisfies $i_R \in \mathbb{N}^0 \bigwedge i_R \leq m$
- t is one of the N *slices* in which the $[0, 1)$ interval is divided

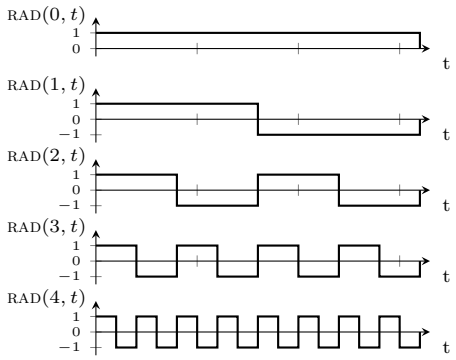


Figure 5.2: Rademacher functions

- The $sgn(h)$ function is defined as $sgn(h) = \begin{cases} 1, & \text{if } h \geq 0 \\ -1, & \text{if } h < 0 \end{cases}$

5.1.3 Walsh Functions

Before describing the process of generating the Walsh functions out of the Rademacher functions, an introduction to the ordering and notation employed by the former ones is provided.

Function notation and matrix ordering

Just as the Rademacher functions are characterized by the i_R index, the Walsh functions are characterized by the i_W index, and denoted by the $\text{WAL}(i_W, t)$ notation. It is important to mention that there are three different ways of grouping the Walsh functions within the transform matrix. For the scope of this project, only one ordering, the *Sequency* ordering, is considered. Consequently, the previously introduced notation in any case refers to the Walsh functions arranged in the sequency ordering. More details on this follow; however, for the sake of completeness, the remaining two orderings are briefly described first.

The *Dyadic* ordering is achieved when the i_W sequency-ordered Walsh function is *moved* to position i_{W_D}, creating a new matrix arrangement. The index i_{W_D} is computed by first writing index i_W in base two. Then, the Gray-encoding is applied to this binary representation, and finally, this new binary number gets converted to decimal basis.

The *Hadamard* ordering is obtained when the i_W sequency-ordered Walsh function is *placed* in position i_{W_H}, thus creating this new matrix ordering. The index i_{W_H} is calculated by expressing index i_W in binary and applying the bit-reversal algorithm to this representation. Afterwards, Gray-encoding is performed on the result, and finally, a conversion from binary to decimal is carried out.

A *Sequency*-ordered Walsh transform matrix, as it name suggests, is the one where the Walsh functions are arranged in such a way that the sequency of a given function is greater than or equal to that of its preceding function. Additionally, the index i_W represents the number of zero-crossings in the $[0, 1)$ time interval. With this in mind, the following case formula is employed to calculate the sequency of the ${i_W}^{th}$ Walsh function:

$$s_{i_W} = \begin{cases} 0, & \text{if } i_W = 0 \\ \frac{i_W}{2}, & \text{if } i_W \text{ even} \\ \frac{i_W+1}{2}, & \text{if } i_W \text{ odd} \end{cases} \tag{5.2}$$

The fact that, except for $i_W = 0$, there are always two Walsh functions mapping to the same sequency arises the use of the $\text{SAL}(s_{i_W}, t)$ and $\text{CAL}(s_{i_W}, t)$ notation when employing the sequency ordering. This notation was coined as an abbreviation for *sine Walsh* and *cosine Walsh* to emphasize their resemblance with the trigonometric sinusoidal fucntions. For a given sequency s_{i_W}, its corresponding odd i_W index

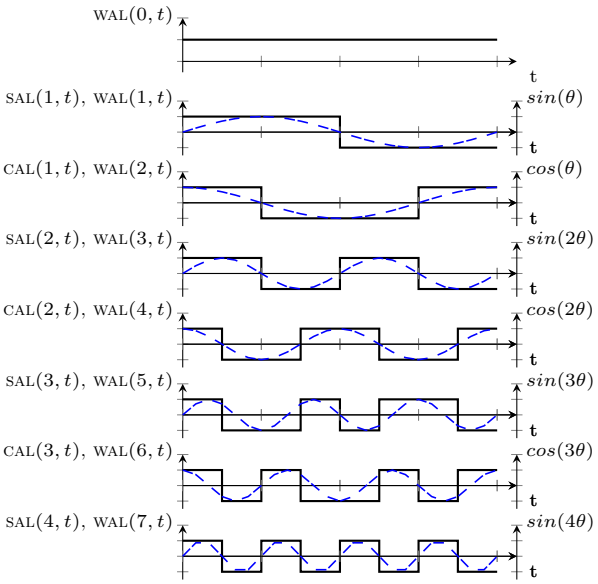


Figure 5.3: Sequency-ordered Walsh functions and their trigonometric equivalents ($\theta = 2\pi t$ and $t \in [0, 1)$).

(the odd Walsh functions) are the $\text{SAL}(s_{i_W}, t)$ functions, whereas the $\text{CAL}(s_{i_W}, t)$ functions represent the even ones. This interpretation of Equation 5.2 is depicted in Figure 5.3, where also the newly introduced notations are employed.

Function generation

The Walsh functions are generated by multiplying a subset of the required m Rademacher functions (where $N = 2^m$ and N is the total number of input samples for the Walsh transform). Given a certain i_W Walsh function index, the method to determine which Rademacher functions to multiply to obtain $\text{WAL}(i_W, t)$ is the following. [21]

First, the index i_W is converted to its binary representation and Gray-encoded using a total of m bits. Secondly, each of the m Rademacher functions with index i_R gets assigned to one of the m bits of the Gray-coded representation. Rademacher function $\text{RAD}(m, t)$ is assigned to the MSB, whereas $\text{RAD}(1, t)$ to the LSB. Finally, for each $\text{WAL}(i_W, \text{T})$ function, the Rademacher functions whose corresponding bit is 0 are dismissed, and the rest of them are multiplied to obtain the required Walsh function. Table 5.1 depicts this process for $N = 8$. Here, it is seen that, for example, $\text{WAL}(4, t)$ is generated by multiplying functions $\text{RAD}(2, t)$ and $\text{RAD}(3, t)$. As expected, all resulting Walsh functions have the same range ($\{-1, 1\}$) as the Rademacher functions.

Walsh Function	i_W **index**	**Gray-encoded indexes (Binary)**			**Walsh Function**
		$\text{RAD}(3, t)$	$\text{RAD}(2, t)$	$\text{RAD}(1, t)$	
$\text{WAL}(0, t)$	0	0	0	0	
$\text{WAL}(1, t)$	1	0	0	1	$\text{SAL}(1, t)$
$\text{WAL}(2, t)$	2	0	1	1	$\text{CAL}(1, t)$
$\text{WAL}(3, t)$	3	0	1	0	$\text{SAL}(2, t)$
$\text{WAL}(4, t)$	4	1	1	0	$\text{CAL}(2, t)$
$\text{WAL}(5, t)$	5	1	1	1	$\text{SAL}(3, t)$
$\text{WAL}(6, t)$	6	1	0	1	$\text{CAL}(3, t)$
$\text{WAL}(7, t)$	7	1	0	0	$\text{SAL}(4, t)$

Table 5.1: Generating Walsh functions through the Rademacher functions

5.1.4 Walsh Transform

Once the Walsh functions have been generated, the Walsh transform matrix is created. When the N Walsh functions (each having a length of N points) are arranged in a sequency ordering, such a matrix (referred to as W_N) is formed. Now an input vector can be transformed into the sequency domain. The Walsh transform (WT_N) is the scaled product of this matrix and the input column vector. Each of the resulting N coefficients represents the magnitude of a certain sequency component contained within the input vector. The Walsh transform is therefore calculated as follows:

$$WT_N = \frac{1}{N} W_N X, \tag{5.3}$$

where WT_N is the column vector with the N Walsh coefficients (output of the Walsh transform), W_N represents the matrix with all N $\text{WAL}(k,t)$ functions and X is the input vector.

5.2 Sequency-to-Frequency Domain Transform

The goal of the Walsh-Fourier (sequency-to-frequency domain) transform, presented in [18] and [19], is to express the Fourier coefficients (a_x and b_x) in terms of the Walsh coefficients, denoted as $A(i)$ (where $i \in \mathbb{N}^0 \bigwedge i < N$). By achieving this representation, it is possible to know which Walsh coefficients are needed to compute the required Fourier coefficients. Recalling the previous chapter, it is important to remember that the output of any Fourier transform algorithm (e.g. the FFT) is a set of N complex Fourier coefficients denoted as c_x. The relationships between these coefficients are the following:

$$a_x = c_x + \overline{c_x}, \tag{5.4}$$

$$b_x = \frac{c_x - \overline{c_x}}{j} \tag{5.5}$$

$$c_x = \frac{a_x + b_x j}{2} \tag{5.6}$$

where $\overline{g}$ is the complex conjugate of the complex number g and j is the imaginary unit $j = \sqrt{-1}$.

Now, the sequency-to-frequency domain transform is described, while providing an example along the explanation. First, assume a discretized time-dependent function $f(i)$ with a total of N sampled points, where $i \in \mathbb{N}^0 \bigwedge i < N$. The time span covered by the N sampled points depends on the sampling frequency of the function f; however, for the sake of simplicity, this time period is normalized to one, such that these N points cover the time interval $\{0, \frac{1}{N}, \frac{2}{N}, \ldots, \frac{N-1}{N}\}$. Moreover, assume all $f(i)$ sampled points are arranged in a column vector.

Each of these $f(i)$ points is expressed in terms of its first N Fourier coefficients (a_x and b_x). In other words, the expression to compute said points can be thought of as the *modified* partial Fourier series expansion of this function. The reason why the adjective '*modified*' is employed has to do with the formula terms being reordered to match the Walsh sequency ordering. More details on this rearrangement follow. To avoid repeated appearances of the 2π factor, a variable replacement takes place: $\omega_i = \frac{2\pi i}{N}$. Now, function f is expressed in terms of ω_i, and the formula to compute the sampled point $f(\omega_i)$ is:

$$f(\omega_i) = a_0 + \sum_{k=1}^{\frac{N}{2}-1} \Big(b_k sin(k\omega_i) + a_k cos(k\omega_i) \Big) + a_{\frac{N}{2}} cos\left(\frac{N}{2}\omega_i\right). \tag{5.7}$$

Introducing the example, where the sample size is chosen to be $N = 8$, Equation 5.7 looks like:

$$f(\omega_i) = a_0 + \sum_{k=1}^{3} \Big(b_k sin(k\omega_i) + a_k cos(k\omega_i) \Big) + a_4 cos(4\omega_i), \tag{5.8}$$

where the value for a given $f(\omega_i)$ is computed.

Since it is of the method's interest to employ the Fourier coefficients a_x and b_x as variables, instead of considering the $f(\omega_i)$ column vector, an $N \times N$ matrix, denoted as F_N is considered. Each row in this matrix represents now an *instance* of Equation 5.7 for a given value of ω_i. Additionally, every element within a certain row represents the value by which the appearances of the Fourier coefficients a_x and b_x, in the same equation, are multiplied. It is important to respect the order of the vector elements within the given row: they follow the same order as the appearances of the Fourier coefficients as written down in Equation 5.7. Further on with the previously introduced example, consider the first row of matrix F_8. In this case, where $i = 0$, the values for this row are found by first computing ω_i ($\omega_0 = 0$) and substituting it in each separate term of Equation 5.8. Table 5.2 explains this procedure in a clearer way.

Fourier coeffs.	a_0	b_1	a_1	b_2	a_2	b_3	a_3	a_4
Row terms	1	$sin(\omega_0)$	$cos(\omega_0)$	$sin(2\omega_0)$	$cos(2\omega_0)$	$sin(3\omega_0)$	$cos(3\omega_0)$	$cos(4\omega_0)$
Final values	1	0	1	0	1	0	1	1

Table 5.2: Filling up a row of the F_8 matrix, with $N = 8$ and $\omega_0 = 0$.

As stated before, two remarks must be made regarding the *modified* partial Fourier series expression in Equation 5.7. Considering the trigonometric functions within the scope of the summation, the coefficient corresponding to the sine term appears *before* the one of the cosine term. This arrangement is adopted to match the sequency-ordered Walsh matrix, where the $\text{SAL}(k,t)$ function appears *before* the $\text{CAL}(k,t)$ function. As a consequence, the elements within each row of the F_N matrix must respect this ordering. Secondly, and contrary to the cited sources, the last term corresponds to the cosine term instead of the sine term. The reason for this is that for all possible values of i (where the constraints $i \in \mathbb{N}^0 \bigwedge i < N$ still hold), $sin(\frac{N}{2}\omega_i) = 0$. This would force both the Walsh $A(N-1)$ and Fourier $a(\frac{N}{2})$ coefficients to be zero, which would cause some problems in the upcoming matrix inversion and would trim out relevant information. To avoid this, the cosine function was employed.

The Walsh transform matrix (W_N) is also a square matrix of size N. For the case of the current example, the matrix W_8 is the following:

$$W_8 = \begin{bmatrix} +1 & +1 & +1 & +1 & +1 & +1 & +1 & +1 \\ +1 & +1 & +1 & +1 & -1 & -1 & -1 & -1 \\ +1 & +1 & -1 & -1 & -1 & -1 & +1 & +1 \\ +1 & +1 & -1 & -1 & +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 & +1 & -1 & -1 & +1 \\ +1 & -1 & -1 & +1 & -1 & +1 & +1 & -1 \\ +1 & -1 & +1 & -1 & -1 & +1 & -1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & +1 & -1 \end{bmatrix} \tag{5.9}$$

The next step is to compute the Walsh transform of the sampled data. Since the assumed input function is expressed in terms of the Fourier coefficients, the Walsh coefficients (output of this transform) are also expressed in terms of the Fourier coefficients. This means that, instead of the transform output being a column vector, it is also an $N \times N$ matrix, denoted by A_N. This notation might seem to conflict with the one employed in Equation 5.3. However, even though both A_N and WT_N represent the Walsh coefficients, WT_N is a column vector containing the resulting values of the Walsh coefficients. The Walsh transformed data, in terms of the Fourier coefficients, is given by:

$$A_N = \frac{1}{N}(W_N)(F_N), \tag{5.10}$$

The k-th row of the A_N matrix has the $A(k)$ Walsh coefficient expressed as a linear combination of a subset of the Fourier coefficients a_x and b_x. Finally, to achieve the desired goal of the Walsh-Fourier transform, the A_N matrix needs to be inverted. As a result, the A_N^{-1} matrix has the a_x and b_x Fourier coefficients expressed in terms of the Walsh coefficients. Please note that the rows in this matrix follow the same order as the appearances of the Fourier coefficients in Equation 5.7. Returning to the example,

the matrix A_8^{-1} can be interpreted as follows:

$$\begin{bmatrix} a_0 \\ b_1 \\ a_1 \\ b_2 \\ a_2 \\ b_3 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} A(0) \\ 1.2071A(1) - 0.5A(2) - 0.5A(5) - 0.2071A(6) \\ 0.5A(1) + 1.2071A(2) - 0.2071A(5) + 0.5A(6) \\ A(3) - A(4) \\ A(3) + A(4) \\ 0.2071A(1) + 0.5A(2) + 0.5A(5) - 1.2071A(6) \\ 0.5A(1) - 0.2071A(2) + 1.2071A(5) + 0.5A(6) \\ A(7) \end{bmatrix} \tag{5.11}$$

When the input to the Fourier transform is real, the last $\frac{N}{2} - 1$ output complex coefficients are discarded, since they are only the complex conjugates of the first half. This property is clearly exploited in this approach. In the example where a real input of length $N = 8$ is assumed, an expected output of the Fourier transform would be the complex coefficients $\{c_0, c_1, \ldots, c_7\}$. However, out of these N coefficients, only five of them ($[c_0, c_4]$) contain relevant information because the following expressions are true: $c_1 = \overline{c_5}$, $c_2 = \overline{c_6}$ and $c_3 = \overline{c_7}$. Even though all N coefficients of the Walsh-Fourier transform are required, the amount of output information is the same as compared to the Fourier transform on real data, since the *middle* $N - 2$ Walsh-Fourier outputs make up the *relevant* $\frac{N}{2} - 1$ Fourier coefficients. In summary, matrix A_N^{-1} has enough information to compute the first and relevant $\frac{N}{2} + 1$ Fourier frequency bins, which corresponds to one half of the sampling frequency ($\frac{F_S}{2}$).

Finally, depending on which are the desired Fourier frequency bins (coefficients), a set of Walsh coefficients needs to be computed. Given a couple of required a_x and b_x Fourier coefficients, their corresponding rows in the A_N^{-1} matrix must be analyzed. More details on this row mapping are provided in the following section. In this way, it is known which Walsh coefficients need to be computed first: they are the ones corresponding to the non-zero elements of such rows. Considering the example in Equation 5.11, if the Fourier coefficient c_2 is required, the rows corresponding to a_2 and b_2 must be taken into account. From these two rows, it is seen that the Walsh coefficients $A(3)$ and $A(4)$ need to be obtained first by computing the dot products between the time domain input and the functions $\text{WAL}(3, t)$ and $\text{WAL}(4, t)$ respectively. Then, a linear combination of $A(3)$ and $A(4)$ must be carried out to obtain both a_2 and b_2 Fourier coefficients, which finally lead to c_2.

5.3 Proposed application

Taking into account that goal is to reduce the execution times by bypassing the decimation filter and time-domain windowing, the straightforward approach would be to apply the Fourier transform (with the FFT algorithm) on the raw bitstreams. However, this method is not efficient, since the FFT algorithm would take a considerably increased amount of time. This is caused by the increased number of input samples that are taken as an input by the Fourier transform: more twiddle factors have to be computed/fetched, and the number of multiplications is increased.

To address these issues, the Walsh-Fourier transform is proposed, and is referred to in the remainder of this document as the Walsh-Fourier approach. The basic idea, as described in the previous paragraphs, is to compute certain Walsh coefficients such that, through a linear combination of these, the desired Fourier complex coefficients are obtained. This indirect Fourier frequency calculation might sound more troublesome. Nevertheless, the advantages of this method lay on the facts that the kernel functions are binary. This means that storing one sample of such a signal requires only one bit, and that multiplications of binary signals are reduced to a bitwise XOR operation.

To incur in the least modifications to the PNAH algorithm as possible, the input time span for the Walsh-Fourier approach should be the same as the one corresponding to the nominal approach. Considering a single channel (microphone), the latter approach applies the Fourier transform to 1024 input samples. These 32−bit samples have a frequency of 46.875 kHz, thus, a time interval of approximately 0.022s is considered (to comply with the transient condition). The raw microphone bitstream has a frequency of 1.5MHz, which means that a total of 32768 1−bit input samples should be processed to cover the same

time interval.

5.3.1 Binary representation conventions

Along this chapter, it has been assumed that the time domain binary signals would have the same range as the Rademacher and Walsh functions, namely, an image consisting of the values $\{-1, +1\}$. However, the microphones output a binary signal with values $\{0, 1\}$, where binary 0 corresponds to -1, and binary 1 to $+1$. This *convention* is referred to as $C^{+1\to 1_b}_{-1\to 0_b}$. For the results of the XOR operation to be correctly interpreted as a multiplication, the Walsh functions have to follow a different convention. As implemented in [22], the $+1$'s of the Walsh functions must be represented as a binary 0, whereas their -1's should become binary 1. This assignment is denoted as $C^{+1\to 0_b}_{-1\to 1_b}$.

The reason for this is that the bitwise XOR operation between binary variables x and y ($\{0,1\}$) can be thought of as a multiplication of binary variables X and Y ($\{-1,+1\}$) with an additional -1 factor: $x \bigoplus y = -XY$. This -1 factor is the reason of the counterintuitive assignment required by the Walsh function. Additionally, this must also be taken into account when multiplying binary-represented Rademacher functions to obtain the desired Walsh functions. Rademacher functions are chosen to be encoded using the $C^{+1\to 1_b}_{-1\to 0_b}$ convention. Thus, when multiplying an odd number of times (equal to multiplying an even number of functions), an additional XOR with binary 1 should be considered to achieve the required $C^{+1\to 0_b}_{-1\to 1_b}$ convention for the Walsh functions. Table 5.3 contains a simple multiplication to better illustrate these issues, along with the conventions employed. Here, the convention adopted by both x and X is the same as the one followed by the microphone raw data, where as the one employed by y and Y is the one required by the Walsh functions.

Binary XOR				**Multiplication**		
$C^{+1\to 1_b}_{-1\to 0_b}$ x	$C^{+1\to 0_b}_{-1\to 1_b}$ y	$C^{+1\to 1_b}_{-1\to 0_b}$ $x \bigoplus y$		$C^{+1\to 1_b}_{-1\to 0_b}$ X	$C^{+1\to 0_b}_{-1\to 1_b}$ Y	$C^{+1\to 1_b}_{-1\to 0_b}$ $X \cdot Y$
0	0	0	=	-1	+1	-1
0	1	1		-1	-1	+1
1	0	1		+1	+1	+1
1	1	0		+1	-1	-1

Table 5.3: Operation comparison between XOR and multiplication.

After the time domain signal has been multiplied by a certain Walsh function, the result of every element has to be added in an accumulator. Together with the multiplication, this reduction makes up the dot product between these two vectors, which in this case represents the Walsh coefficient. When the signals are in the $\{-1, +1\}$ range, the results are added without any further consideration; this is not the case when the signals are represented with the values $\{0, 1\}$. A final *translation* between these two representations has to take place. This last consideration, along with the previously described operations and conventions is summarized with the following expression:

$$A(k) = \Big(2 \cdot OneCount\big(v_N(k)\big)\Big) - N, \tag{5.12}$$

where:

- $A(k)$ represents the k-th Walsh coefficient.
- $OneCount(h)$ is a function that, given a vector h whose range is $\{0, 1\}$, outputs the number of binary 1's contained in it.
- $v_N(k)$ is a vector of size N, following the $C^{+1\to 1_b}_{-1\to 0_b}$ convention, containing the result of the element-wise multiplication of the time domain input X and the Walsh function $\text{WAL}(k, t)$, both of size N as well. Vector X has the $C^{+1\to 1_b}_{-1\to 0_b}$ convention, where as $\text{WAL}(k, t)$ is represented in the $C^{+1\to 0_b}_{-1\to 1_b}$ convention.

5.3.2 Matrix calculations

The implementation of the Walsh-Fourier approach using 32768 samples requires the computation of matrix A^{-1}_{32768}, which in turn requires matrices F_{32768} and W_{32768}. These matrices are very lengthy to compute, let alone to multiply. However, it is only a one-time procedure since once A^{-1}_{32768} is obtained, all the Fourier coefficients are expressed in terms of the Walsh coefficients, and thus can be saved for later references. Additionally, not the entirety of this matrix is required. Because of the 32768 input samples, this matrix has enough information to compute the first 16385 Fourier frequency bins. Remembering that the sampling frequency of the raw data is $F_{S_b} = 1.5$MHz, a frequency range from 0 to approximately 750.045kHz could be reconstructed. Yet, considering a much lower range of frequencies of interest, say from 0 to 10kHz, approximately the first 220 Fourier frequency bins would be usable. This means that only the first 440 rows of the A^{-1}_{32768} would be of interest.

Given a desired Fourier frequency bin number $Bin_F \mid Bin_F \in \mathbb{N}^0 \bigwedge Bin_F \leq \frac{N}{2}$, the index of its corresponding rows in the A^{-1}_N matrix (denoted as i_{a_k} and i_{b_k} for the a_k and b_k coefficients, respectively) is computed according to three different cases. In the first two, a single index (i_{a_k}) suffices because the values for these Fourier frequency bins are real numbers. In case one, $Bin_F = i_{a_k} = 0$, whereas in case two, where $Bin_F = \frac{N}{2}$, $i_{a_k} = N - 1$. In the third case, where $Bin_F \in [1, \frac{N}{2} - 1]$, the Fourier frequency bins contain complex numbers, thus, both a_k and b_k are required. For these bins, the following expressions are employed:

$$i_{a_k} = 2 \cdot Bin_F \tag{5.13}$$

$$i_{b_k} = 2 \cdot Bin_F - 1 \tag{5.14}$$

Rows i_{a_k} and i_{b_k} (or only i_{a_k}, depending on the case) of matrix A^{-1}_N contain the *weighting* values for each of the relevant Walsh coefficients. For $N = 32768$, the number of non-zero elements contained in each of the rows corresponding to the above mentioned frequency range is *too high* (typically between 1k and 20k coefficients whose absolute value is larger than $1 \cdot 10^{-6}$). Instead of computing all these Walsh coefficients to obtain a single Fourier frequency bin, a selection is needed. For the same frequency range, the coefficients with an absolute value larger than 0.1 is arbitrarily chosen. In this case the complex numbers computed from the a_k and b_k coefficients are an approximation to the complex coefficients obtained from the Fourier transform since there is some error inherent to the coefficient selection. Figure 5.4 illustrates the variation of this amount per frequency bin. Additionally, two more aspects are deduced from looking at this figure. First, there are some Fourier frequencies which require considerably fewer Walsh coefficients to be computed, such as $\approx 91, \approx 182, \approx 365, \approx 730, \approx 1460$ and ≈ 2930Hz, which in this case, require only 10 Walsh coefficients. Second, a tendency is observed where, the higher the Fourier frequency, the larger the amount of Walsh coefficients required to approximate it.

5.3.3 Implementation

Two variations of the Walsh-Fourier approach, along with the details described so far were implemented in the GPU platform, and the output of this stage was fed to the PNAH algorithm. Further details on the GPU implementation itself, as well as on the results obtained, are provided in the upcoming chapters. In the next paragraphs, a description of these two slight variations on the approach is found.

Initially, the set of the n desired Fourier frequency bins requires a bigger set of n_W Walsh coefficients (where $n_W \mid n_W \in \mathbb{N}^+ \bigwedge n < n_W \ll N$). The indexes of these Walsh coefficients are not known, and they have to be selected. This process is carried out with the information contained in the A^{-1}_N matrix and follows the procedure described in the paragraphs above. This is necessary only once, but it must be done before the Walsh-Fourier transform can take place.

Remember that each Walsh coefficient has a unique Walsh function associated to it. In the case where $N = 32768$, storing N Walsh functions, each with N 1-bit samples would require a large space when considering the size of the *fast* global memory (128 MB for N Walsh functions, when such a memory has a size of 64 kB. More details on this follow). Additionally, only n_W out of N would be required. Therefore it is reasonable to work with only the required Walsh functions instead. It is here where the two variations of the approach differ. On the one hand, it is possible to precompute the required Walsh

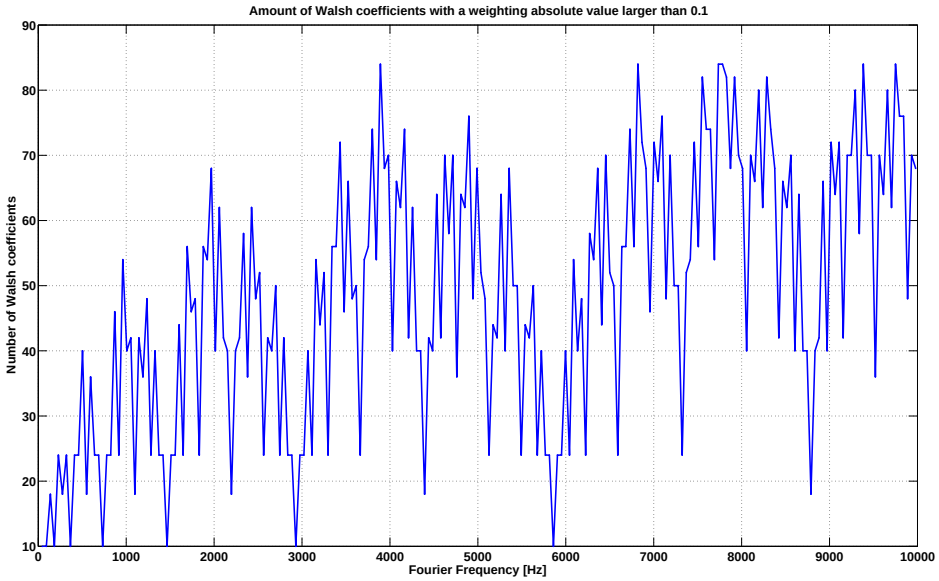


Figure 5.4: Number of Walsh coefficients required per Fourier frequency bin.

functions and feed this as an input to the Walsh-Fourier transform. This is referred to as the *Precomputed Walsh Functions* approach. On the other hand, the required Walsh functions can be computed from the m base Rademacher functions (where $N = 2^m$). This is referred to as the *Walsh-function Computation* approach.

The Precomputed Walsh Functions approach is faster because the dot product (XOR + reduction) between these functions and the time domain input is executed right away. Nevertheless, it is not flexible to changes in the set of n Fourier frequency bins, since if this is the case, the provided Walsh functions would no longer be the appropriate ones. Additionally, all the required Walsh functions should be stored in memory. On the other side, the Walsh-function Computation approach is more flexible in this sense because any of them are computed based on the m Rademacher functions, which in turn, are the only ones needed to be stored in memory. Compared to the previous approach, its main disadvantage lies on the fact that the computation of the Walsh functions can become complex mainly because of the Gray-code conversion and the constant Rademacher function fetching.

After the introduction to the Walsh-Fourier transform and the description of the details surrounding its implementation, this chapter concludes. The obtained results regarding this approach are discussed in the following chapters.

Chapter six

Implementation

This chapter describes the details regarding how the proposed approaches are implemented in the GPU platform, whose characteristics are found in Appendix A. In general, the code executing within this device describes the calculations to be done on the information of a single dataset: a data parallelism model is followed. In the scope of this project, such a dataset represents the measured information acquired from each of the N_{ch} pressure sensors (microphones) of the Sorama Cam. In this way, the platform efficiently exploits the parallelism inherent to the PNAH algorithm.

For this application the PNAH algorithm aims for a real-time implementation at a *high-as-possible* throughput. Because of this, the computation parallelism that can be achieved with the employed hardware needs to be exploited to meet the execution time deadlines. As previously mentioned, the three different proposed approaches for this project, namely the n Dot Products, the Basic Averaging, and the Walsh-Fourier Transform are also implemented in the GPU platform. These three methods concern the transform from spatial-time to spatial-frequency domains. The basic (*slow* or *naive*) method to perform the domain transform of a single dataset X ($N \times 1$ matrix) consists on multiplying it by an $N \times N$ transform matrix. Because of the nature of the PNAH algorithm, only a subset n of the total results are required. In this sense, the required transformed results are found by employing a transform matrix with a reduced size of $n \times N$; this means that n dot products need to be computed. The three proposed approaches employ the same computation approach and for this reason, all three are based on a similar code template that performs the described computations.

With this in mind, this chapter is organized as follows. First, a description of the implementation through the OpenCL framework of the dot product code template (OpenCL *kernel*) is provided. In this section, some general aspects regarding the architecture assumed by OpenCL are included, followed by the explained dataflow within the OpenCL kernel. Secondly, having the code template in mind, the modifications done for specifically each of the three approaches are detailed.

6.1 OpenCL Kernel - Dot Product Template

The general dataflow implemented by the proposed kernel is provided in this section. However, prior to this, a few basic concepts belonging to the OpenCL framework are introduced. After this, the description of the kernel implementation is outlined employing the proper OpenCL terms and concepts.

6.1.1 OpenCL

OpenCL is a programming framework for general purpose parallel programming across hardware platforms such as CPUs and GPUs, among others [23]. The OpenCL execution model is comprised by a *host* which is connected to a set of *devices*; these devices are usually the hardware containing multiple processing units: multicore CPUs or GPUs, for example. The host executes a main application, from which a number of commands are issued to one or more of the devices connected to the host. For the sake of

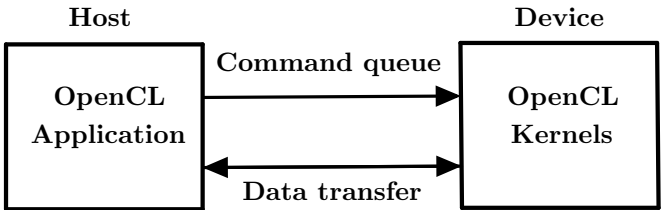


Figure 6.1: Host-Device interaction within the OpenCL framework.

simplicity, and to address the current hardware configuration, a single device is considered from this point on. Furthermore, such device is assumed to be the mentioned GPU. The host can be thought of as a manager issuing instructions to the worker: the device.

A coarse summary of the interactions within such a model is described as follows. As previously mentioned, the host runs an OpenCL application. When a section of this application, within which data parallelism can be exploited, is reached, the host issues a command for the device to execute such section. After this, the host can either resume its activities or wait for the issued command to be completed. The command is queued in the *command queue*, which is fed to the device in order for it to receive the issued instructions. The device is in charge of internally scheduling the code execution, carrying it out and notifying the host about its completion. This code, referred to as OpenCL *kernel*, specifies the computations to be performed for a single *piece* of data. Usually, after the execution of a kernel has been finished, the host application can request the transmission of the results from the device memory, or can issue a command to execute a different kernel, for example. In any case, the control is returned to the host application. Figure 6.1 depicts in general the communication between host and device.

The OpenCL framework provides an abstract memory model; its mapping to the physical one is done exclusively by the hardware device itself. Since both the soft- and hardware models are relevant, two memory architectures are described: the virtual memory model assumed by the OpenCL framework and then the physical memory model implemented by the GPU. Afterwards, the relationship between both is provided.

Virtual memory model

When a kernel is assigned to a specific device, OpenCL sees the device as a hierarchically arranged *space* in which the kernel code is executed. Within the device, there are two grouping hierarchies. Namely, a device contains a certain number of *work groups*, and each work group consists of a given number of *work items*. For both hierarchy levels, their elements can be organized in $1D$, $2D$ or $3D$ grids. For example, a device can be organized in a 3×2 two-dimensional grid of work groups, or a work group can be organized in a $3 \times 2 \times 4$ three-dimensional grid of work items. For each dimension, a given identification index is provided. However, for the sake of simplicity and for the scope of this project, only $1D$ grids at both hierarchy levels are employed.

Starting from the lowest level, the work items are the elements in charge of executing the code in the kernel, and can also be thought of as individual processing threads. The work items have its own *private memory* space, where all the variables corresponding to each thread are stored. This private memory is the fastest and smallest of the rest. Within their corresponding work group, the work items communicate with each other via the *local memory*, which is a memory accessible only by the items within a group. Although larger, this memory is slower than private memory. It is important to explicitly include a synchronization mechanism when fetching data from this memory after writing to it: since threads execute in parallel, no assumptions can be made regarding the order of memory accesses. The work items in a given work group are identified by their *local id* $L_{id} \mid L_{id} \in \mathbb{N}_{\geq 0} \bigwedge L_{id} < L_{Max}$, where L_{Max} is the maximum number of work items allowed per work group. The size of a work group is denoted as L_{Size}. These concepts are illustrated in Figure 6.2 (right).

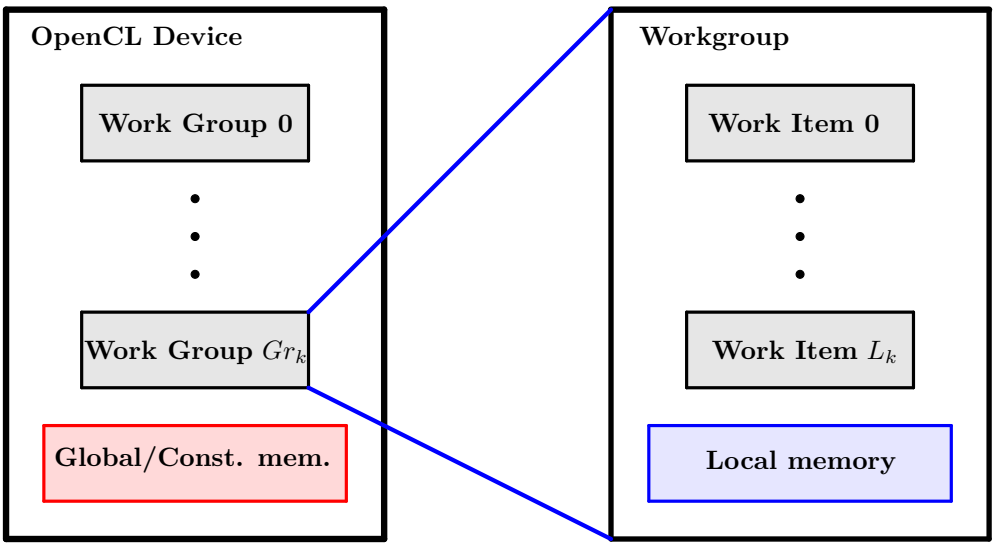


Figure 6.2: Virtual OpenCL memory architecture. A device (left) consists of one or more workgroups, and each of these (right) is comprised by one or more work items.

At the upper hierarchy level, the device is organized into one or more work groups. Each group is identified by their *group id* $Gr_{id} \mid Gr_{id} \in \mathbb{N}_{\geq 0} \bigwedge Gr_{id} < Gr_{Max}$, where Gr_{Max} is the maximum number of work groups allowed per device. The size of a work group is denoted as Gr_{Size}. These groups communicate with each other through the *global memory*, which is the largest and slowest memory space. Typically, the kernel inputs and/or outputs are placed in global memory, since it is this location to/from which the data is copied from/to the host. Besides the global memory, there is also the *constant memory*, which is globally accessible too. The difference between these is that the latter behaves similarly to a read-only memory along the kernel execution, and it is also optimized for data broadcasting. Figure 6.2 (left) depicts these relationships.

It is important to mention that all work items within a device is also identified by their *global id* where $G_{id} \mid G_{id} \in \mathbb{N}_{\geq 0} \bigwedge G_{id} < G_{Max}$, where G_{Max} is the maximum number of work items allowed per device. Additionally, this global id is also be computed as: $G_{id} = \left(Gr_{id} \cdot L_{size}\right) + L_{id}$.

Physical memory model

This subsection only concerns the physical memory model of the employed GPU device, whose characteristic are found in Appendix A. This AMD GPU implements the Graphics Core Next (GCN) architecture, which is a RISC SIMD architecture replacing the previous VLIW4 SIMD one of earlier devices. This architecture improves the device performance for general purpose applications.

The device is comprised by a collection of compute units (CUs). These units represent the basic computational building blocks of the GCN architecture since each of them implements the provided instruction set. Within each CU, one scalar and four vector (SIMD) units are present. Each of the four SIMD units has 16 processing elements (PEs) or ALUs, which can apply the same operation across 16 elements [24]. Using the notation employed by AMD, each of these processing elements is referred to as a Stream Processor (SPs), and there are 64 SPs per compute unit.

Virtual-Physical Memory Relationships

At the *upper* level of the virtual memory hierarchy, an OpenCL workgroup is usually mapped to a compute unit. In this sense, the workgroup employs all the CU's resources, such as the local shared memory, for example. Additionally, the GCN architecture enables each SIMD unit to work on separate *wavefronts*; thus, improving latency hiding. A wavefront is a group of threads (work items) which execute the same instruction. Work items, which are mapped to the stream processors, are instantiated in integer

multiples of the wavefront size (64).

6.1.2 Kernel description

Now that the basic OpenCL-related concepts are introduced, the general structure of the code employed to perform the dot product operation are described in terms of these concepts. First of all, it is important to remember that the data corresponding to a total of $N_{ch} = 1024$ microphones needs to be processed. Each microphone generates its own, independent data; thus, they are processed in parallel. In this way, and having in mind that only $1D$ arrays of both workgroups and work items are considered, 1024 work groups are instantiated: the device is represented by a 1×1024 matrix (vector) of work groups. Each work group shares the global memory, which is where the input and output data is placed. This arrangement is depicted in Figure 6.3.

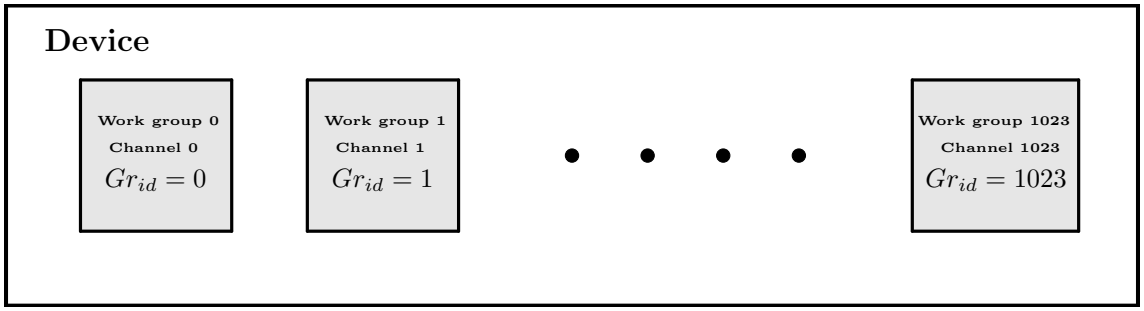


Figure 6.3: Device arranged in a $1 \times N_{ch}$ matrix of work groups ($N_{ch} = 1024$).

The maximum possible size of a work group is of 256 work items in the X direction. Furthermore, the use of this size is suggested in order to optimize the device efficiency; therefore, each work group is comprised of a 1×256 matrix of work items. These work items share the local memory, where the intermediate results of each dot product are stored. Figure 6.4 illustrates this structure.

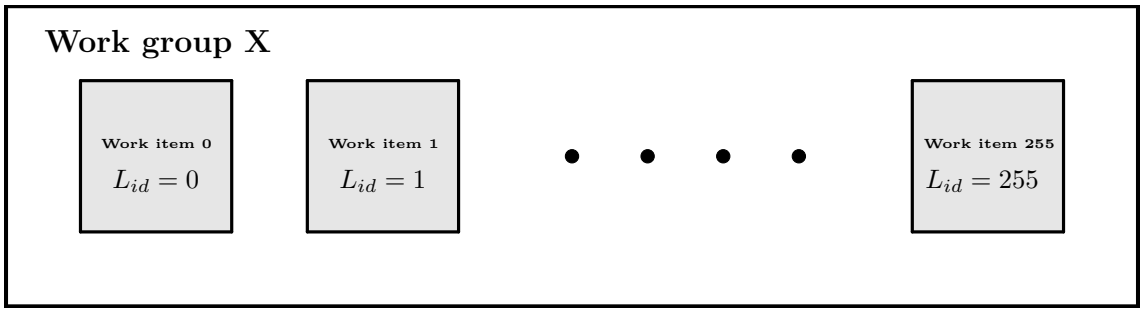


Figure 6.4: Workgroup, corresponding to microphone X, arranged in a 1×256 matrix of work items.

The work group corresponding to any of the N_{ch} input channels needs to read a total of N input samples. Since a work group size of 256 is assumed, each individual work item has to read $N_{WI} = \frac{N}{256}$ samples from global memory. For example, in the *nominal* case where $N = 1024$, each work item fetches four time samples. Once the work items have their corresponding input, the information regarding which frequency bin to compute is required. This information specifies the vector with which the dot product with the time input is applied. In general, two different approaches are used here. In the first one, this frequency-dependent vector is also fetched from global memory. This case is the simplest, since the work items needs not compute the required N_{WI} vector elements, but just fetch them from (global) memory. On the second case these elements need to be computed; this is done using the work item local id (L_{id}) and the number of samples that each work item has to read (N_{WI}). Figure 6.5 illustrates how each work items fetches its corresponding time samples, and transform function (as in the first case described).

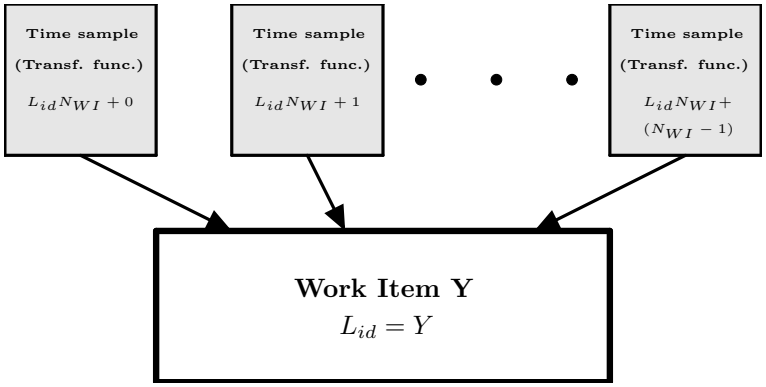


Figure 6.5: Work item fetching its corresponding time samples (and transform functions if needed). To increase efficiency, the elements are loaded as a $<$ DataType $>$ N_{WI} vector (float4, for example).

Once all work items have their corresponding time domain input and transform function samples, an element-wise multiplication between them is performed. After this, the sum of all of these products (reduction) has to take place. First, since all work items have a total of N_{WI} results, these are summed up, so that all work items have one result. The reduction operation yields a final result per work group. Therefore, the work items employ the local memory to communicate with each other: initially, all work items submit their own result to an array of intermediate results located in the local memory (LDS). All subsequent intermediate results are also located in this memory; the array location of these results is the same as the work item index L_{id}.

The reduction stage follows a *divide-and-conquer* paradigm. Figure 6.6 illustrates this approach. As just mentioned, all 256 work items *upload* their results to their corresponding location within the array located in LDS memory. To describe the following reduction process, assume a variable $A_{WI} = 256$, denoting the *number of active work items* in the reduction stage. This variable gets halved for each reduction iteration; therefore, in the first loop, it gets a value of $A_{WI} = 128$. The work items whose index is less than A_{WI} are the active items in each recursion step; their corresponding boxes in the referred figure are colored in black. For each recursion step, the work items whose local index fulfills $L_{id} < A_{WI}$ fetch the results corresponding to work items $L_{id} + A_{WI}$ (colored in red) and add them to their own. After this, these work items update their result within the array in LDS memory and the following recursion step is ready to start again by halving variable A_{WI}. The boundary condition for this recursion is: $A_{WI} > 1$. At the end of the recursion stage, work item $L_{id} = 0$ has the final dot product for the given transform function, which in most cases represents the desired frequency bin.

This template can be further optimized. Two dot products can be computed at the same time adding a *mirrored* version of this structure. This is helpful when an even number of frequency components needs to be computed, or when two dot products are required to yield a result (as is the case for the Walsh-Fourier approach, for example). This structure is illustrated in Figure 6.7, where the *first half* of the work group is responsible for computing one dot product, and the *second half* of the work group is responsible for the other one. As expected, this approach requires that, prior to the beginning of the reduction stage, two different transform functions have been computed or fetched, and that the time domain input has been element-wise multiplied by both these functions. Furthermore, it also makes use of two different arrays in the LDS memory, each of them holding the results of the corresponding element-wise multiplications and being accessed only by its corresponding set of work items. For this mirrored modification, the condition along the reduction recursion discerning between which work items perform an addition and which remain idle changes depending on the work items' local id L_{id}. For an $L_{id} < 128$, work items fulfilling $L_{id} < A_{WI}$ fetch results from location $L_{id} + A_{WI}$ and add them to their own, just as described in the previous paragraph. However, if $L_{id} \geq 128$, work items fulfilling the condition $L_{id} > (255 - A_{WI})$ are the ones fetching results from $L_{id} - A_{WI}$ and operating on them. For the second half of the work group, the work items actively performing the reduction in each stage are colored in red (Figure 6.7), where as the ones in black remain idle. Additionally, their arrows depicting the dataflow are shown in red. For both cases, the boundary condition $A_{WI} > 1$ stays the same, since the fact that A_{WI} is halved for each recursion step remains unchanged. At the end of this reduction recursion, work items with $L_{id} = 0$ and $L_{id} = 255$ have each a different dot product result.

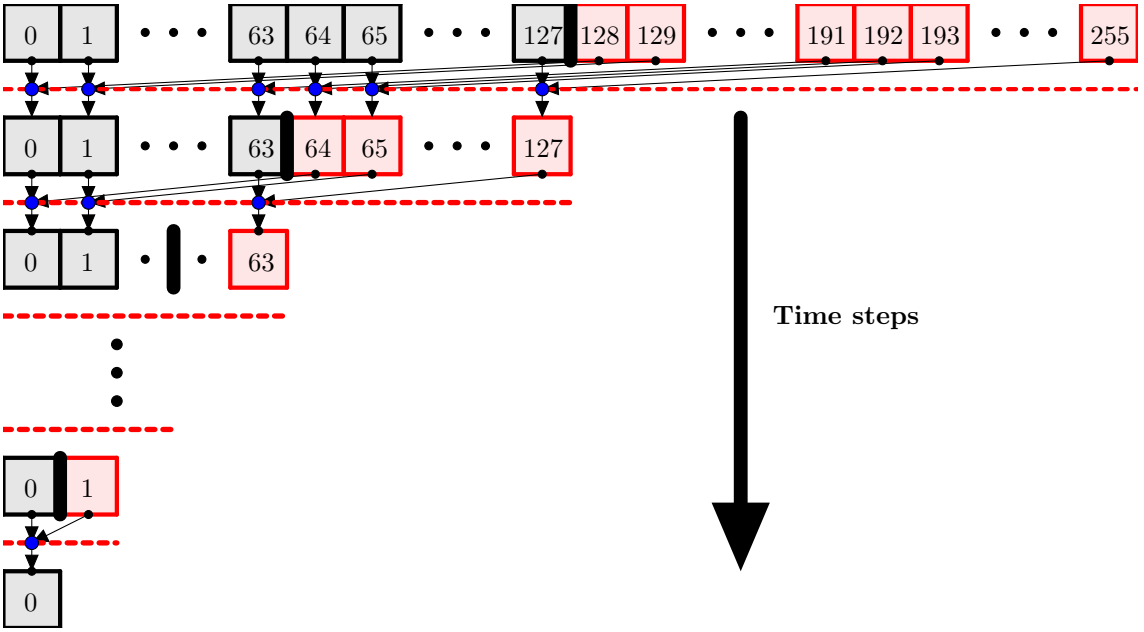


Figure 6.6: Reduction within a work group. Each work item is represented by a box with its corresponding index. In each stage, only work items colored in black execute operations, whereas the ones in red remain temporarily idle. The blue dots represent an addition.

Writing to local memory (LDS memory) takes more time than doing so to private memory (register file); therefore, reducing these events as much as possible reduces execution times. For both mirrored and unmirrored approaches, an additional optimization addressing this aspect can be carried out. Starting from the element-wise multiplication, and along all reduction recursion stages, the work items fulfilling the operating condition ($L_{id} < A_{WI}$ and $Lid > (255 - A_{WI})$ for the first and second half, respectively) only store their corresponding results in private memory, whereas the ones not fulfilling these conditions, submit their results to the LDS memory array(s). This method saves $\sum_{i=1}^{8} \left(\frac{1}{2} \cdot \frac{256}{2^i}\right)$ and $\sum_{i=1}^{8} \frac{256}{2^i}$ LDS memory accesses for the unmirrored and mirrored versions respectively.

6.2 Approach-specific Kernel Adaptations

Now that the kernel template has been outlined, the specific approach-dependent variations to the kernel are further discussed. These modifications mostly concern the transform function samples computation; the reduction loop, either mirrored or unmirrored, is kept intact.

6.2.1 n Dot Products

For the *n Dot Products* approach, the outlined kernel suffers the least modifications. As previously stated, depending on whether the number of desired frequency bins (n) is even or odd, the mirrored or unmirrored template are employed. Since the number of decimated (32-bit) samples that this approach considers for the domain transform is $N = 1024$, each work item within a workgroup has to fetch $N_{WI} = \frac{1024}{256} = 4$ time domain samples.

Additionally, regarding the transform functions (twiddle factors), both fetching and computing them were tried out. The best (smallest) execution times were achieved by computing said factors. More details on this difference follow in the subsequent chapter.

Considering the approach where the transform functions are computed, and given a certain frequency bin index $FB_s \mid FB_s \in \mathbb{N}_{\geq 0} \bigwedge FB_s < N$, a work item with local id L_{id} has to compute a total of

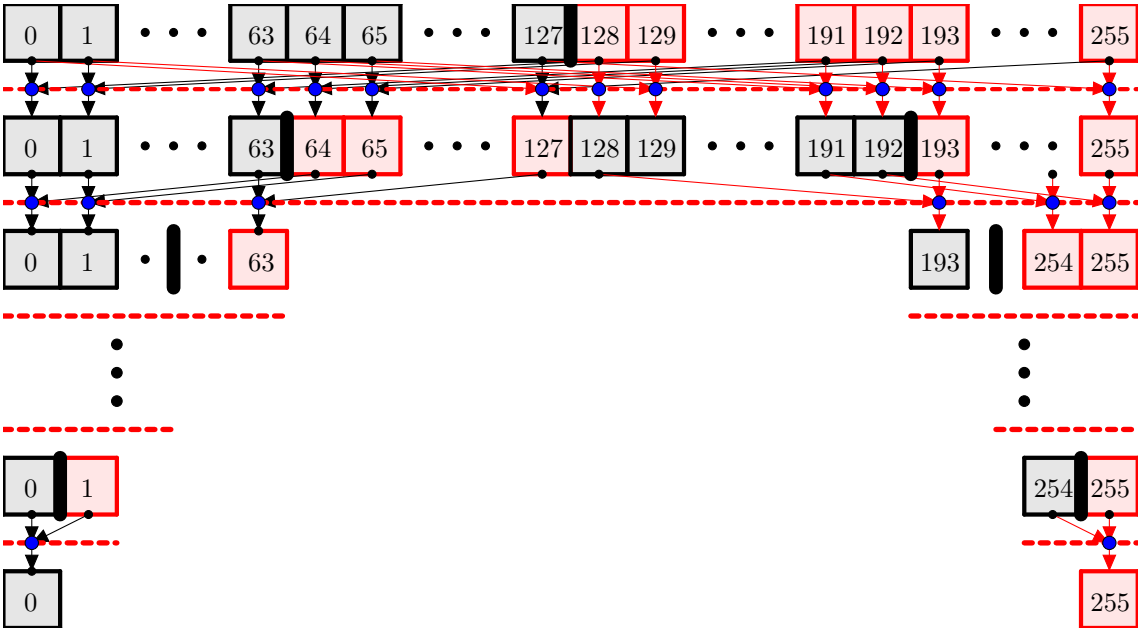


Figure 6.7: Mirrored reduction within a workgroup. For the *first* (left) half of the workgroup, work items whose boxes are colored in black remain active, whereas the ones corresponding to the boxes in red remain idle. This color convention is reversed for the *second* (right) half of the workgroup.

$N_{WI} = 4$ different twiddle factors. As mentioned before, a twiddle factor is a complex number in the form of $cos(\theta) - j \cdot sin(\theta)$. Therefore, a cosine and a sine have to be computed using all N_{WI} different arguments. Each argument is computed as follows:

$$\theta = TS_{idx} \cdot FB_s \cdot \frac{2\pi}{N} \tag{6.1}$$

where:

- $TS_{idx} = L_{id} \cdot N_{WI} + h$, where $h \in \{0, 1, \ldots, (N_{WI} - 1)\}$.
- FB_s is one of the n desired frequency bins.

In this way, N_{WI} arguments are computed, which in turn are used for the same number of twiddle factors corresponding to a single frequency bin. This process is done for all n desired frequency bins. In total, a work item computes $n \cdot N_{WI}$ arguments, and $2 \cdot n \cdot N_{WI}$ sinusoidal samples.

6.2.2 Basic Averaging

A bitstream input is considered for this method. As mentioned in chapters three and six, a total of $N = 32768$ 1-bit samples are required to execute the domain transform. Therefore, a work item executing this approach needs to fetch $N_{WI} = \frac{32768}{256}$ binary samples for a work group to process all elements. Since these binary samples are sequentially *packed* in 32-bit chunks, a work item fetches four 32-bit chunks. In other words, the same amount of information is being employed as the previous approach. The only difference is that it is arranged differently.

Within a work item, the *basic averaging* is done for every 32-bit chunk. As described in Table 3.1, a *population count* method is employed. This method, implemented via the V_BCNT_U32_B32 bit count instruction of the corresponding GPU device architecture [25], counts and returns the number of *set* (binary 1) bits in a 32-bit unsigned integer (which only represents the sequentially packed 32 1-bit samples). This result follows the $C^{+1 \to 1_b}_{-1 \to 0_b}$ convention, introduced in the previous chapter and illustrated in Table 5.3. The result of this population count method is processed in a similar way as done by Equation 5.12, however, with a few modifications:

$$Avg(TS_{idx}) = \Big(2 \cdot OneCount\big(x(TS_{idx})\big)\Big) - 32, \tag{6.2}$$

where:

- $Avg(TS_{idx})$ is the averaged result corresponding to a 32-bit chunk whose index is TS_{idx}
- $x(TS_{idx})$ is the input 32-bit chunk with index TS_{idx}

After this basic averaging process has been made, the corresponding twiddle factors for the computation of the required Fourier frequency bins takes place just as described in the section above.

6.2.3 Walsh-Fourier Transform

The Walsh-Fourier Transform is also carried out with the raw binary input. For this method, two variations were implemented: one where the precomputed Walsh functions are given as an input, and the other one where these are computed based on the Rademacher functions. For the latter one, the process of synthesizing the Walsh functions was already described in the previous chapter; thus, it is not further mentioned. Additionally, the fastest execution was achieved when the Walsh functions were precomputed. For these reasons, the following description assumes that the Walsh functions are already computed and within the work items' memory.

Just as the *basic averaging* alternative, the input data is sequentially packed in chunks of 32-bits. This is also the case for the Walsh functions. Since the same number of $N = 32768$ 1-bit samples is required, a total of $N_{WI} = 128$ binary samples are needed as well. Therefore, four 32-bit chunks of the raw binary input and four 32-bit chunks of each of the required Walsh functions are processed by every work item.

In this case, the element-wise multiplication related to the previous two Fourier-based approaches is replaced by a bitwise XOR. A total of 128 individual XOR operations have to be applied. However, this operation is applied on all 32 elements of a chunk via the V_XOR_32 instruction. Furthermore, when considering vectors of four unsigned integers (again, only representing the sequentially packed 1-bit samples), these 128 XOR operations are executed with a single instruction.

The binary 1's in the result need to be counted employing the same method as proposed in Equation 6.2. This is done for the four chunks within a single work item. After these four results are added, the reduction loop takes place. It is important to remember that for the Walsh-Fourier transform, both Fourier coefficients a_n and b_n are required to compute a c_n complex coefficient. To speed up this process, the mirrored reduction loop depicted in Figure 6.7 is employed. In this structure, the *first half* of the workgroup is in charge of reducing the results for the a_n coefficient, whereas the *second half* of it reduces the results for the b_n coefficient.

As a final extra stage for this Walsh-Fourier transform, when both a_n and b_n are computed, they are weighted by their corresponding weights in matrix A_N^{-1}, which is exemplified in Equation 5.11. Once all the required coefficients are weighted, the required Fourier coefficient c_n is computed. A total of n of these coefficients are calculated following the same process.

With these approach-specific descriptions, this chapter concludes. Now that the kernel structure for all the proposals has been detailed, the results achieved with them are provided in the next chapter.

Chapter seven

Results and Analyses

Now that the main ideas and details regarding the proposed solutions have been provided, along with their implementation details in the employed GPU, the achieved results are discussed. For all the solutions proposed, first the achieved results are provided, followed by an analysis on this data. The obtained data presented in this chapter is compared to the *nominal* implementation. The metrics to be compared are the approach execution time and the resulting inverse propagated holograms obtained based on the input holograms generated by the proposed approaches.

The reasons why the nominal approach is taken as a reference are the following. Regarding the execution times, the nominal approach, as shown in [6], is currently the implementation where a single iteration of the PNAH algorithm takes the least time (at most 1 ms. per iteration, generating at most 10 output holograms). Therefore, the comparisons made on this aspect are against the *state of the art*. The reasoning is not as straightforward when justifying the use of the nominal approach as a baseline when the hologram *correctness* is to be analyzed. Previous related work, such as [5] and [6], have measured hologram differences employing simulated pressure distributions as baselines; in other words, the baselines are not directly a result from sampling, decimating, filtering and transforming the measured time samples. A similar case where the spatial frequency representation of the measurement plane is used as an input for the PNAH algorithm cannot be assumed for this project, since the focus of the acceleration is on the time-frequency domain transform itself. In case simulations were to be employed, a sound field simulation would be required. In order for it to be an accurate simulation, the source impulse response should be computed and extrapolated [26]. This requires some initial information on the targeted system, and a way of obtaining such data is actually employing the PNAH algorithm itself. With this reasoning, the proposed approaches were directly tested on real data instead.

Moreover, relying on the validation of methods such as the PNAH via comparison against simulations, Sorama has successfully implemented said techniques on practical scenarios, confirming the applicability of the results through different experimental cases. These *real life* implementations have been done following the nominal data acquisition protocol, where the data has been decimated, windowed and transformed to form the hologram. Therefore, the correctness of the input hologram creation method through this time-domain method is assumed. As a consequence, this assumption has been made along this project, and for error-measuring purposes, this is not an exception. For the scope of this project the mentioned assumption suffices since it is not part of the objective to prove that the nominal data acquisition protocol produces the same hologram as the ones simulated. Finally, it is important to bear in mind that the nominal approach does not achieve a perfect reconstruction but an accurate approximation instead. This fact opens the possibility of the binary-based representations being a better approximation than the ones obtained from the nominal approach; this possibility is one of the important contributions of this project.

Previous to presenting the results, the metrics measurement methods (error and performance) are provided. Once this has been done, the results and an analysis for each of the proposed approaches are presented. Beginning with the developed kernel, this template is tested to get some information on how *well* does it perform when compared to other implementations. More specifically, the general matrix-matrix multiplication (GEMM) section of the clBLAS library is used. clBLAS is a library, based

on the OpenCL framework, which contains optimized routines to compute basic linear algebra operations in OpenCL devices. clBLAS is a *sibling* library of clFFT; both of them are part of clMath, the open-source contribution of AMD to efficient libraries implementing mathematical computations through the OpenCL framework. The objective of this first comparison is to select the best option to carry out the proposed dot products, which is later compared with the FFT employed in the nominal approach.

After this kernel template has been tested, the *n Dot Products* approach is presented. This is the one closest to the nominal implementation since it performs the $1D$ domain transform employing the decimated and filtered data. Because of this, the output holograms have no difference, nor in amplitude nor in phase, with the holograms that result from applying the FFT algorithm. As a consequence, the focus lays on the execution times alone at first. The objective of this is to also provide an initial idea on the advantages and limitations in which a DFT-like dot product domain transform incurs when compared with the full FFT.

Then, the *Basic Averaging* approach is measured. The execution time metrics for this approach are given first. As mentioned before, the advantages of this method regarding the mentioned aspect are due to the fact that the raw bitstream is employed (bypassing decimation filtering and time domain windowing). However, because of the different preprocessing done on the signal, the significative results concern the differences between the holograms generated by this process and the ones created through the nominal implementation.

A similar case is that of the *Walsh-Fourier Transform* method, which first presents the execution time results, but where the more significant results regard the hologram errors.

7.1 Metrics Measurement

Before providing the results, the criteria employed to evaluate the outputs are provided.

7.1.1 Execution Time and Performance

The required information to measure these metrics is obtained with the AMD CodeXL Profiler version 1.4.5724.0 (as specified in Appendix A). The execution time is a direct output of this tool; however, the performance, which is measured employing the *Roofline Model*, requires additional information (obtained from the same profiler) to be computed.

Before introducing the roofline model, it is worth noting that an additional criterion, namely the *Kernel Occupancy*, is employed to further explain some behaviours shown in these models. The kernel occupancy, also provided by the mentioned profiler, is defined as the ratio (percentage) between the number of active work items at any instant, achieved by the profiled kernel, and the theoretical number of work items that can be active at the same time in the device. This relationship mostly depends on the amount of both local and private memory employed by the analyzed kernel. The device has a limited amount of physical memory; thus, if a kernel requires *too much* memory, only a limited amount of work items can be active at the same time. In this case, the remaining work items need to be queued prior to execution. On the other hand, when a kernel requires a *small* amount of memory, the theoretical maximum is achieved; therefore, a larger or the largest possible number of work items run at the same time.

Roofline Model

The *Roofline Model*, introduced in [27], is a visual performance model employed to provide general insight on how to improve a routine's efficiency by increasing the number of floating point operations per unit of time. This model helps to quickly discern, for example, whether a routine is memory- or computation-bounded (more details on these concepts are found in subsequent paragraphs). The Roofline model consists of a $2D$ plot, where the *operational intensity* is plotted on the X axis, and the attainable device performance, in $\left[\frac{FLOPS}{s}\right]$, is plotted on the Y axis. Both axes are displayed in a base-2 logarithmic scale. The operational intensity is the ratio representing the operations executed per byte of memory

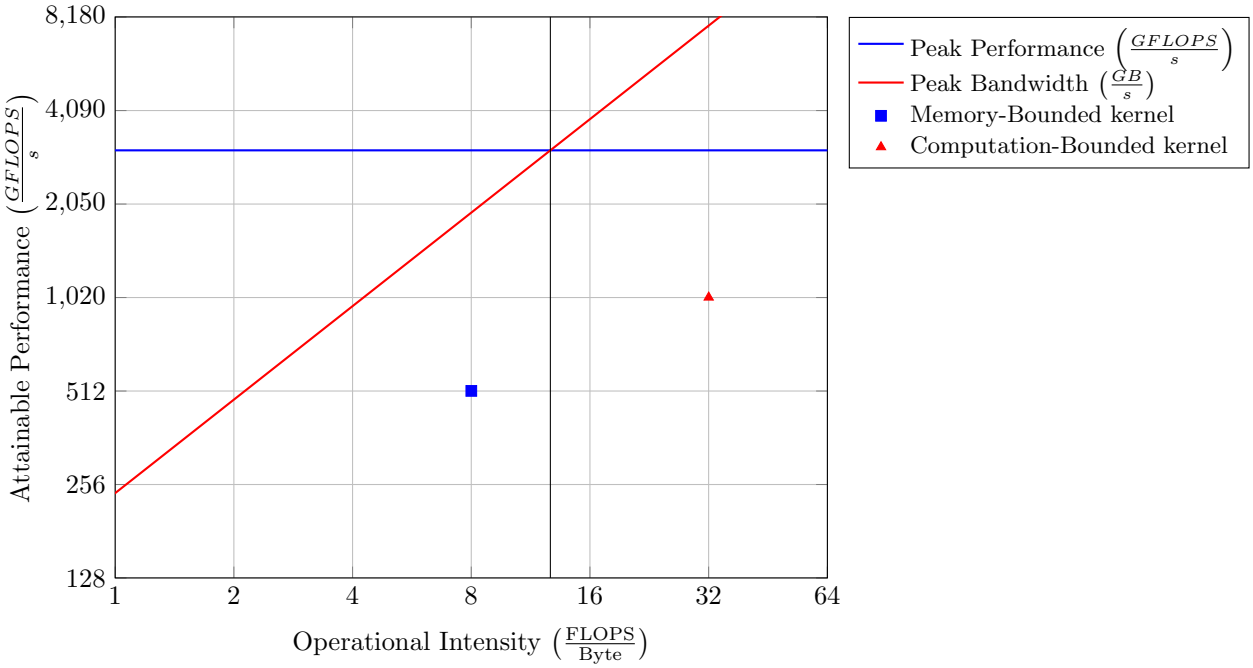


Figure 7.1: Roofline model example displaying the characteristics of two different hypothetical kernels running on the GPU hardware described in Appendix A (Memory bandwidth of 240 $\frac{GB}{s}$ and peak performance of 3046.4 $\frac{GFLOPS}{s}$).

traffic. A condition on this traffic is that the data is fetched not from any cached memory, but from the external memory. In the case of the considered GPU platform, these data transmissions are the ones between the processing elements and the global memory.

The roofline model gets its name because of the roofs below which the *points* corresponding to the execution of a piece of code (OpenCL kernel in this case) are located. Such a point describes the code execution in terms of achieved performance. First, there is a constant, horizontal roof, representing the peak performance of the physical device. This value, given in terms of $\left[\frac{FLOPS}{s}\right]$ and obtained from the hardware specifications, does not depend on the operational intensity, since the device has a limited computation capacity. Secondly, there is a sloping roof corresponding to the peak memory bandwidth achieved by the device. This roof is obtained by multiplying the device's memory bandwidth (constant value in terms of $\left[\frac{bytes}{s}\right]$ and obtained from the hardware specifications as well) by the corresponding operational intensity, thus yielding a result in terms of $\left[\frac{FLOPS}{s}\right]$. In this sense, for every value of operational intensity, the valid roof is the minimum value of the device's peak performance and peak memory bandwidth. A vertical line is drawn at the point where both roofs intersect; the points laying to the left of this line are memory-bounded, whereas the ones laying to its right are computation-bounded. A piece of code (OpenCL kernel) whose corresponding point is in the memory-bounded region is characterized by having a high percentage of its execution time waiting on the memory fetch unit. On the other hand, the computation-bound region typically represents the other case, where the system's processing elements are in use the majority of the CPU time, causing the memory-fetch unit to be rarely used.

Figure 7.1 illustrates a roofline model example of the considered hardware. Two example points are depicted, where each of them represent the performance characteristics of a given code (OpenCL kernel) within the modeled device. The coordinates of a given kernel's point are obtained from the output of the profiler when executing this code. The operational intensity (X axis) is computed by dividing the total amount of bytes transferred to and from the global memory by the total number of instantiated work items. The achieved performance (Y axis) is computed by multiplying the total number of work items by the number of floating-point operations executed by each of them, and then dividing this product by the execution time.

In this sense, this model provides visual insight on how efficiently are the hardware resources being used, as well as on which actions can be taken to improve this efficiency, which could lead to execution time reductions. For example, if a kernel is in the memory-bounded region, improving the temporal locality of the memory accesses can help reduce this number. On the other hand, if the kernel is in the computation-bounded region, employing a different computation paradigm, such as a *divide-and-conquer*, will alleviate the low efficiency.

7.1.2 Hologram Errors

The differences between the output source plane holograms generated based on the input holograms created via the nominal approach and the proposed approaches need to be measured. The reason for this is that the relevant metric is how accurate can a reconstruction be achieved using a certain input, not the input itself. Therefore, two quantities describing the error per reconstructed hologram are computed. First, the Root Mean Squared Reconstruction Error (RMSRE) is used to provide a relative quantitative error following the next expression:

$$RMSRE_{A_i} = \sqrt{\frac{1}{UV} \cdot \sum_{u=1}^{U} \sum_{v=1}^{V} \frac{(|p_{Nom}(u,v)| - |p_{Ai}(u,v)|)^2}{|p_{nom}(u,v)|^2}}, \tag{7.1}$$

where:

- A_i where $i \in \{2, 3\}$ denotes approach A_2 as the Basic Averaging and approach A_3 as the Walsh-Fourier Transform. A_1, the n Dot Products approach, does not modify the holograms (w.r.t. the *nominal* approach); thus, errors are not computed for it.
- U and V are the dimensions of the reconstructed hologram ($U = V = 32$).
- $p_{A_i}(u, v)$ is the spatial-frequency representation of the pressure distribution at the source plane at point (u, v) as reconstructed using one of the proposed approaches.
- $p_{nom}(u, v)$ represents the same as $p_{A_i}(u, v)$, but reconstructed based on a nominal hologram.

The second error measurement is the Normalized Sum of Absolute Differences (NSAD) given by:

$$NSAD_{A_i} = \frac{1}{UV} \cdot \sum_{u=1}^{U} \sum_{v=1}^{V} \Big(\frac{|p_{nom}(u,v)|}{max|p_{nom}|} - \frac{|p_{A_i}(u,v)|}{max|p_{A_i}|} \Big), \tag{7.2}$$

where:

- $max|p_{A_i}|$ represents the maximum absolute value of the source plane hologram for approach A_i.
- Similar situation with $max|p_{nom}|$, but employing a nominal hologram as an input.

The NSAD error provides an absolute error which provides a measurement on visual differences because it is normalized to the maximum value of a hologram, which is the value employed to define the colors assigned to all the remaining elements of the hologram. This is an important metric, since much insight on a system is gained via visual inspection.

When attempting to compare the inverse propagated holograms obtained through the proposed Basic Averaging and Walsh-Fourier Transform approaches, a difference to remark is that results obtained by these methods are scaled by a certain factor when compared with the results obtained via the nominal holograms. This is due to the fact that the magnitudes of the $1D$ transform input signals being different across each approach. To properly compare these holograms, the *normalized holograms* are compared instead. All elements of a hologram are divided by its maximum magnitude:

$$p_{norm}(u, v) = \frac{p(u, v)}{max(p)}, \tag{7.3}$$

where p_{norm} is the inverse propagated normalized hologram. This is done for all the resulting holograms to be compared (nominal implementation, Basic Averaging and Walsh-Fourier Transform).

After this step, the errors described in Equations 7.1 and 7.2 are computed for a proper comparison between these holograms. It is worth mentioning that since the normalized holograms are being compared, the 'Normalized' qualifier in the NSAD error is redundant, and only the element-wise difference between both holograms is considered (a value of 1 becomes the denominator for both terms). Now that the error metrics have been described, the results, followed by an analysis, are presented.

7.2 Kernel Template

The family of GEMM functions pertaining to the clBLAS library is used to compare the execution of the kernel template created for this project. Existing Basic Linear Algebra Subprogram (BLAS) libraries for GPU platforms exist, such as CUBLAS and MAGMA. Since CUBLAS is aimed for the CUDA framework (NVIDIA devices), the clBLAS library was selected for this comparison due to the fact that it is optimized for AMD GPU hardware [28]. This was the reason to choose it over MAGMA.

In this case, the n dot products are viewed as a matrix multiplication: the transform matrix A with size $n \times N$ contains the $N = 1024$ twiddle factors for all n desired frequency bins, whereas the time domain input matrix B with size $N \times N_{ch}$ contains the time samples for all $N_{ch} = 1024$ channels. In this sense, matrix $C = A \times B$ is the $n \times N_{ch}$ matrix with the transform results. Instantiating these matrices with the corresponding data, the clBLAS function clblasCgemm() is employed, since it multiplies general rectangular matrices with float complex elements [28].

Regarding the proposed dot product template, the kernel is instantiated as follows. The device contains a total of N_{ch} workgroups which each process the dataset for a single microphone. All workgroups have 256 work items in charge of processing $N = 1024$ time samples; thus, each work item fetches $N_{WI} = \frac{1024}{256} = 4$ samples. The work items compute the required twiddle factors based on their local indexes. This means that no precomputed transform function samples need to be fetched and that the only global memory accesses involved are the ones to read the input and to write the output. The *global size* of this kernel is of $256 \cdot 1024 = 262144$ work items.

It is important to notice that three different variations of the proposed kernel template were made. The first one follows the reduction proposed in Figure 6.6, where only one output is generated after each reduction cycle. The second one follows the mirrored reduction proposed in Figure 6.7, where two outputs are generated after each reduction cycle. Finally, the third one is an *optimized* version of the mirrored reduction, where four outputs are generated after each reduction cycle. Because of their characteristics, the first one is employed when the number of required output holograms (n) is an odd number, the second one when n is even, and the third one when n is a multiple of four. A differentiation between these three version is made in the following results.

Results

Dummy test data (random float numbers ranging between $[-1, 1]$) was employed as the time domain input to the performed tests, where the number of desired frequencies was $n \in [1, 10]$ together with $n \in \{12, 14, 16, 18, 20, 22, 30, 40, 44, 50\}$. The results are observed in Figure 7.2. Additionally, the roofline model for both these methods was computed and is seen in Figure 7.3.

Analysis

From the results concerning the proposed kernel in Figure 7.2 (bars in different tones of black), it is seen that, in general, the kernels computing an even number of outputs (mirrored reduction) perform better than the ones computing an odd number of outputs. This means that it is results better to compute an even number of outputs, even if an odd number is required. Additionally, it is seen that the optimized version for multiples of four (black bars) performs better up until when $n = 16$. This can be noted by looking at the results when $n = 20$ and $n = 22$; when $n = 22$, the execution time is smaller than when $n = 20$, which is contradictory. The reason for this is that the *optimized* version for multiples of four employs a considerably larger amount of local memory (LDS). When $n = 4$, this version has a kernel

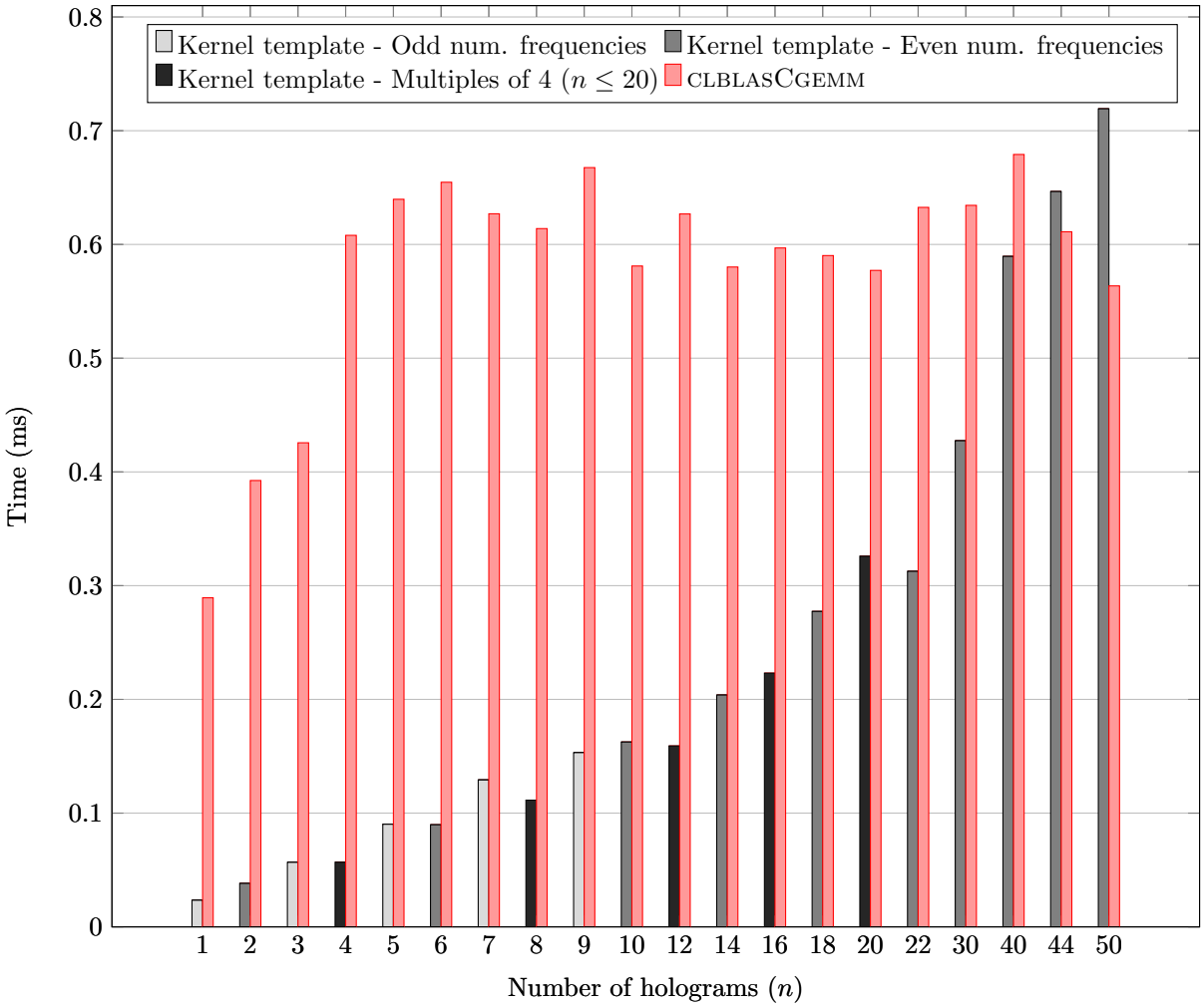


Figure 7.2: Execution time of the Dot Product kernel template vs. clBLAS implementation.

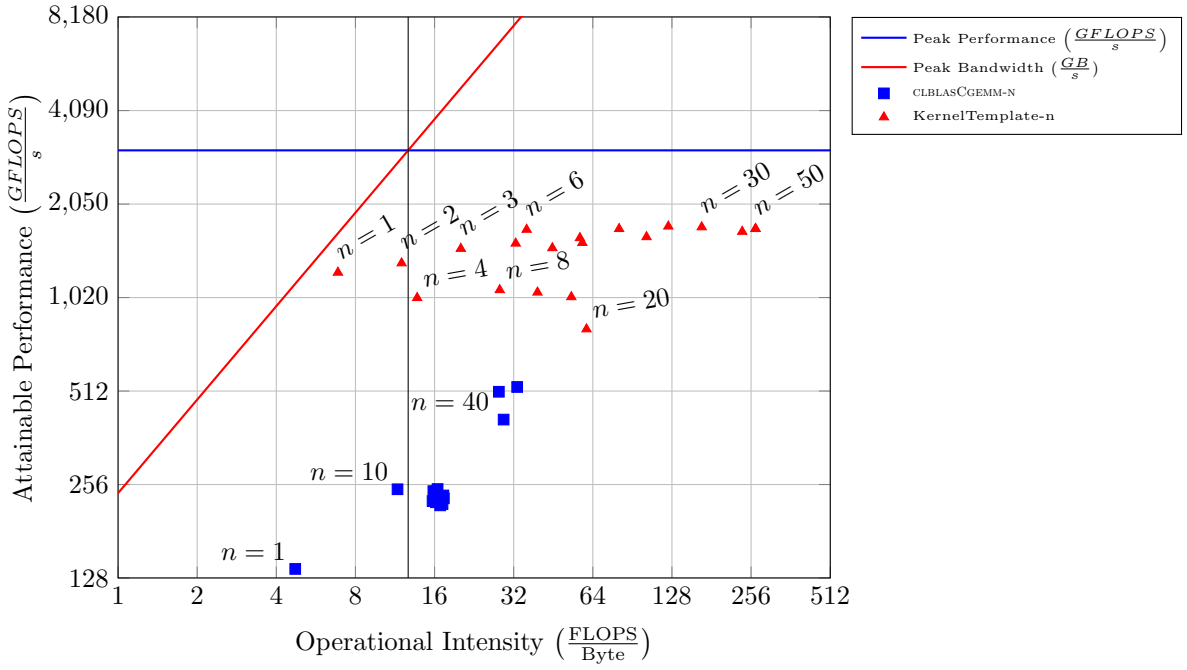


Figure 7.3: Roofline model of the Dot Product kernel template and clBLAS implementation. The number of desired frequency bins is $n \mid n \in [1, 10] \bigwedge n \in \{12, 14, 16, 18, 20, 22, 30, 40, 44, 50\}$.

occupancy of 100%; however, for larger integer multiples, the efficiency slowly decreases to 20% at $n = 16$. Actually, for $n > 20$, this *optimized* version cannot execute due to the large requirements in terms of LDS memory. This constantly decreasing kernel occupancy contrasts with the one corresponding to the other two kernel versions, which is always 100%, independent of the variable n.

From Figure 7.2 it is also seen that the kernel template executes in less time than the clBLAS implementation up until when $n = 44$. Here, the dot product kernel already takes more time, and from this point on it is beneficial to employ the clBLAS library. Limiting the range of required frequency bins ($n \leq 12$), the savings achieved with the proposed kernel template are considerably larger, and around a factor of 10 for when $n = 1$, for example. One cause of this is the low kernel occupancy achieved by this library, which remains at 20% for all n. However, the main reason for this difference is that previous knowledge on the contents of the transform matrix is used in the proposed kernel, where as this is not the case for the clBLAS implementation.

More specifically, the proposed kernel computes the required twiddle factors, whereas the clblasCgemm function fetches them from the device's global memory. The cost incurred by these extra memory accesses is what decreases the execution time when employing the proposed template, specially when $n \leq 12$. However, this is the same reason why when $n \geq 44$, this same approach requires more time than the clBLAS library. When the size of the memory fetch is larger, the accesses are coalesced and *more data* can be retrieved with a single fetch. This causes the cost of fetching the twiddle factors to be less than the cost of computing them, therefore, reducing the execution time.

It is shown in the roofline model in Figure 7.3 that the kernel template achieves a considerably better GFLOPS metric; however, the operational intensity increases proportionally with the number of desired frequency bins n, causing the increase in execution time for this approach. The fact that the points corresponding to the kernel template start to *move* to the right as the number of desired frequencies increases is expected. This region corresponds to *computation-bounded* kernels, where the amount of computation done on a single item is too high. This approach employs the DFT-based method which has a complexity of $O(N^2)$; the *right-shift* of these points is a consequence. This is not the case for the clBLAS implementation, where a *step* pattern can be deduced from the *clusters* in which the points are located. The GFLOPS attainable by this library are low compared with the kernel implementation for this range of n. Nevertheless, the *right-shift* towards the computation-bounded area is *slower*, thus limiting the increase in execution time.

Usually, software libraries, such as clBLAS, are already optimized for the target systems or frameworks and further optimizations are not guaranteed to provide valuable gains. Despite of this, when certain assumptions on the problem are made, such as a limited range of the n value, gains can be achieved within the assumed region. These general libraries cannot assume such situations since this would limit their generic employability.

7.3 1D Time-Frequency Domain Transform

In order to obtain experimental data to compare, four different datasets were collected by measuring acoustic sources with the Sorama Cam in a semi-anechoic environment:

- Dataset 1 ($D1$) - Pistophone emitting a frequency of 1 kHz.
- Dataset 2 ($D2$) - Pistophone emitting a frequency of 1 kHz (different location on the plane).
- Dataset 3 ($D3$) - Vibrating plate.
- Dataset 4 ($D4$) - Power drill.

From the corresponding measurements performed, the raw binary streams of all microphones were extracted. For each originally measured dataset, an additional one was derived by applying the decimation and time-domain window filter to it; in this sense, two different *versions* existed for each dataset. The decimated and filtered version is employed with the n Dot Products approach. Since this approach does

not change the hologram with respect to the nominal implementation, the ones obtained by this method are also labeled as the *nominal* holograms when referring to the hologram errors. On the other hand, the raw bitstreams originally measured are used with the Basic Averaging and Walsh-Fourier Transform approaches.

Initially, the execution times corresponding to the time domain preprocessing and $1D$ domain transform of the PNAH nominal implementation are included. These numbers, contained in Table 7.1, represent the baseline against which the rest of the approaches' execution times are compared to. As a side note, the *Decimation Filter* stage comprises the entirety of the computations executed by the FPGA platform. In this sense, this time is only approximated due to the fact that several factors such as memory buffering, protocol drivers and queuing cannot be fully predicted. Considering an upper limit of 100 cycles to process the dataset corresponding to a single channel, the total execution time of this stage is the approximated to 0.001 ms.

N = 1024	**Exec. time (ms)**
Decimation Filter	≈ 0.001
$1D$ Windowing	0.048
FFT	0.089
Total	0.138

Table 7.1: Baseline execution time metrics for the $1D$ Time-Frequency Domain Transform.

7.3.1 n Dot Products

This approach only represents a modification to the domain transform algorithm itself; the decimation and $1D$ window filtering are the same as compared to the nominal approach. Taking this into account, only the execution times corresponding to the Dot Products and the FFT is considered since the holograms obtained from this method are the same as the ones generated by the nominal implementation.

Results

The execution times of the Dot Product approach, normalized with the FFT execution times, are provided in Figure 7.4. The roofline model comparing both the FFT algorithm versus the DFT-based one is included in Figure 7.5.

Analysis

The kernel characteristics previously pointed out become more apparent in Figure 7.4. In the cases where *optimized* mirrored version was used ($n = 4$ and $n = 8$), the execution times show a sort of *stepped* behaviour in the sense that, with respect to their predecessors ($n = 3$ and $n = 7$), this metric does not increase as could be inferred. Additionally, it is seen that the difference in execution time between the cases when $n = 5$ and $n = 6$ is minimal. Once again, this is due to the mirrored pattern proposed in Figure 6.7.

The Dot Products approach yields gains in execution times when $n \in [1, 5]$. Based in Equations 3.1 and 3.2, which estimate the limits of this approach's applicability, and considering that $N = 1024$, it would be expected that $n_x = 10$, meaning that this approach would remain beneficial in a range $n \in [1, 10]$. Nevertheless, these experiments show that this is not the case by a factor of 2 (Figure 7.4).

One of the causes for this difference is the fact that the clFFT library does not incur in any twiddle factor computation, whereas the proposed approach does compute the require twiddle factors. The clFFT-generated kernel writes the basic twiddle factors, in a permutated fashion, in the *constant memory*. When these complex values are required, they are broadcasted to all the work items who requested them. Moreover, the permuted order in which these factors are stored allows an ordered access to this memory, since it follows the memory access pattern characteristic of the Stockham implementation of the FFT

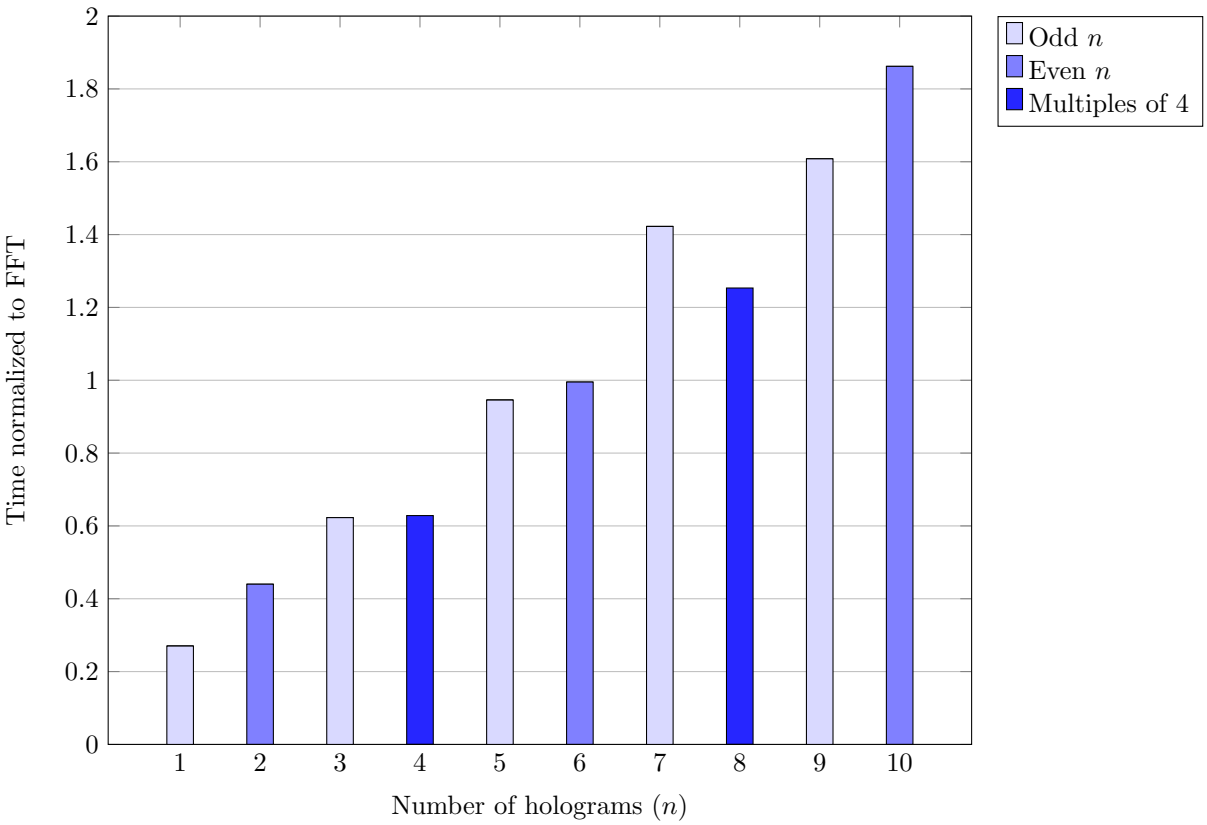


Figure 7.4: Dot Product execution times, normalized to the FFT execution time.

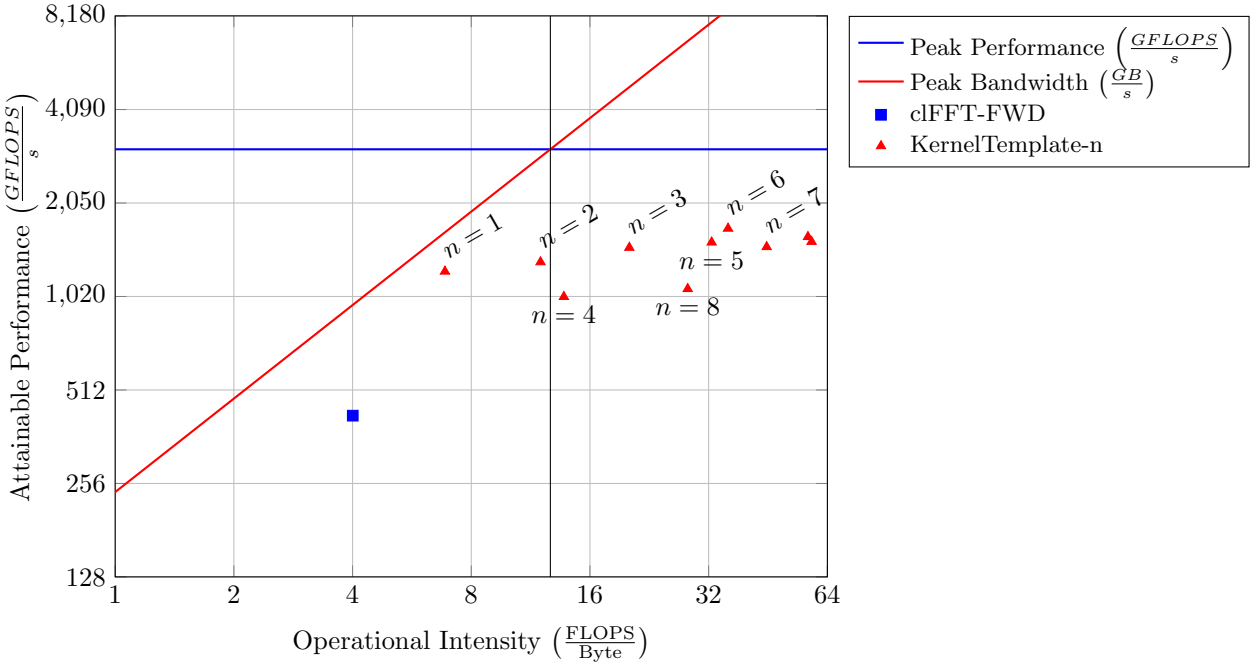


Figure 7.5: Roofline model of the Dot Product kernel and the clFFT implementation. The number of desired frequency bins ranged in the values $n \in [1, 10]$ for the Dot Product kernel.

algorithm, which is the one used by the clFFT library. As a final remark, the *divide-and-conquer* paradigm is such that, within a certain tree level (except in the leaves), more than one twiddle factor is employed by different *butterflies*; therefore, an efficient reuse of the fetched data takes place. This cannot be the case for this DFT-based implementation since each of the n different frequency bins require different twiddle factors that, in the general case, are not reusable by the remaining frequency bins.

In order to explore the benefits of twiddle factor fetching over computing, a version of this proposed approach, where the twiddle factors were also fetched from constant memory was implemented. The frequent fetches required by this algorithm's paradigm increased the average execution times by a factor of approximately 20 since the memory fetches increased proportionally with the number of holograms (n). This caused the algorithm to be memory-bounded and decreased its operational intensity, which in turned decreased the efficiency achieved by the device while executing this kernel. For these reasons, this implementation was discarded.

Regarding the roofline model from Figure 7.5, a clear distinction can be seen between the cases where $n = 4$ and $n = 8$ (*optimized* mirrored version) and the rest. A more subtle difference is also noticeable between the mirrored and unmirrored versions when $n = 5, n = 6$ and $n = 7$. Nevertheless, it is seen that the quickly increasing operational intensity of the dot product approach in general is the cause of it being beneficial for only a limited number of required frequency bins. The easiest option to solve this issue, when considering larger number of required frequencies, is changing the algorithm paradigm, or in other words, to employ the FFT algorithm. Nevertheless, when $n \in [1, 5]$, the Dot Product approach is recommended.

7.3.2 Basic Averaging

This approach uses the raw binary data to perform a basic averaging/decimation based on a *population count* function to count the number of binary 1's. Once this is done, the data is transformed via the Dot Product approach or the FFT algorithm. In this case, to further employ the proposed kernel template, as well as to avoid the overhead of launching a different kernel for the clFFT library, the Dot Product approach was included in the same kernel as the population count-based decimation.

Results

Within the kernel deployed for this approach, the proposed dot product takes place right after the population count-based decimation has been executed. In this sense, the execution times obtained from the profiler are the sum of the last approach plus the population-count based decimation:

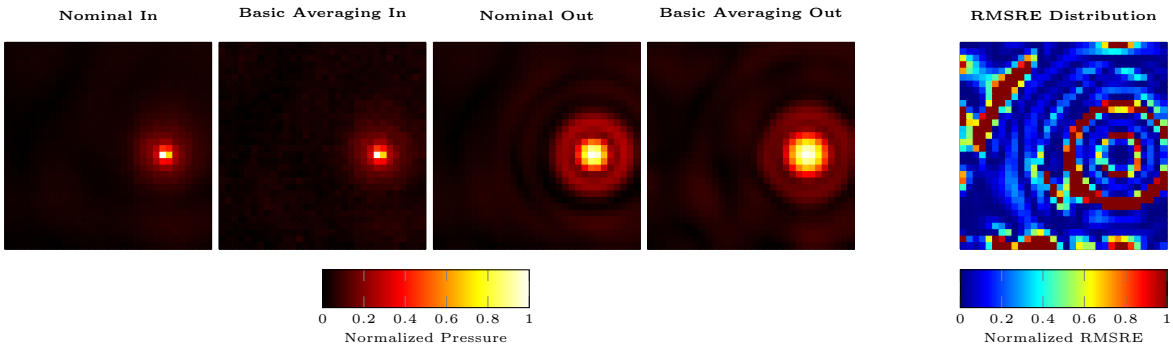
$$t_{A2} = t_{A1} + t_{pc}, \tag{7.4}$$

where:

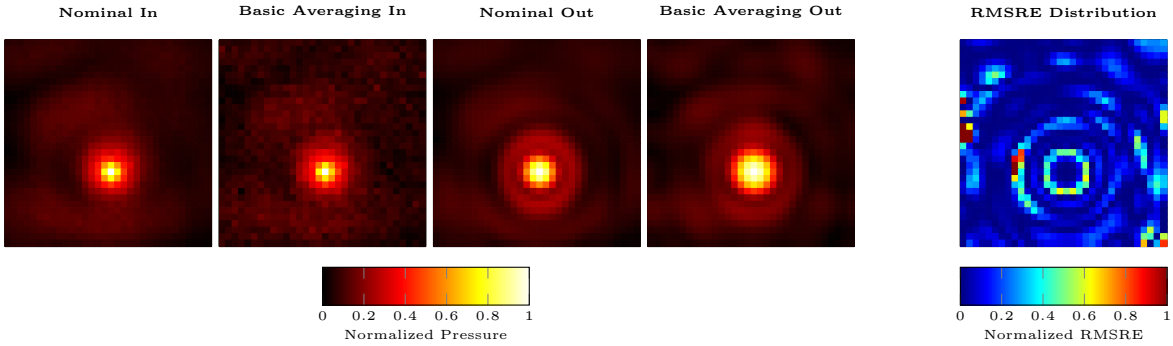
- t_{A2} denotes the execution time for Approach 2 (Basic Averaging).
- t_{A1} is the execution time for Approach 1 (Dot Product approach, as shown in the last section).
- t_{pc} is the execution time of the population count-based decimation by itself.

With this expression, the execution time of the basic decimation is computed by substracting the two outputs from the profiler. After performing a set of measurements where the n variable was in the range of $[1, 10]$, and using the data from the previous approach, the average execution time of the population count-based decimation was found to be $t_{pc} = 0.005$ ms.

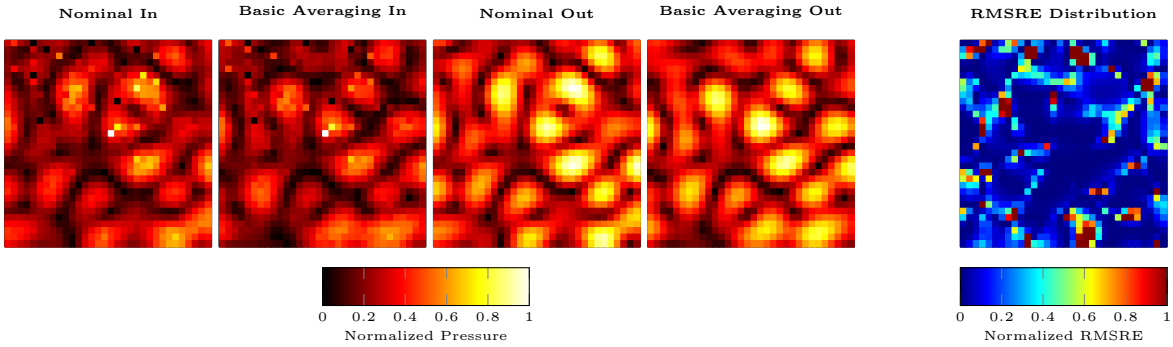
Regarding the holograms, an example of the input and backpropagated holograms employing the nominal and Basic Averaging approaches are seen in Figure 7.6a and in Figure 7.6b.



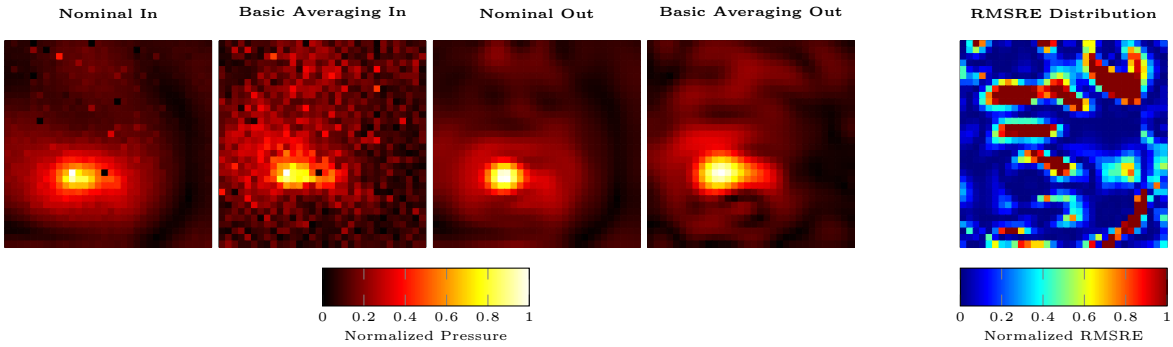
(a) Dataset $D1$ - 1000 Hz - RMSRE of 14.3% and NSAD of 2.05% (Output holograms).



(b) Dataset $D2$ - 1000 Hz - RMSRE of 7.5% and NSAD of 1.2% (Output holograms).



(c) Dataset $D3$ - 220 Hz - RMSRE of 11.7% and NSAD of 0.9% (Output holograms).



(d) Dataset $D4$ - 824 Hz - RMSRE of 19.2% and NSAD of 7.1% (Output holograms).

Figure 7.6: Input and Output holograms comparison between the Nominal implementation and the Basic Averaging. The normalized pressure is employed and four different datasets (Subfigures *a*, *b*, *c* and *d*) are analyzed. For all datasets, the K-space filter parameters are: cut-off wavenumber $k_{co} = 50\frac{rad}{m}$ and slope of 0.3. The RMSRE error distribution (normalized to the corresponding RMSRE value) between both output holograms is shown at the right.

Analysis

From the nominal results included in Table 7.1, it is seen that the achieved population count-based decimation ($t_{pc} = 0.005$ ms) performs 5 times slower than the decimation filtered currently carried out in the FPGA. However, this method eliminates the need of a time-domain window. In fact, applying such a window considerably distorts the frequency spectra and the relevant information for a given frequency bin is lost. By removing the time-domain windowing step, the achieved execution times are reduced. Additionally, different advantages are exposed, which might not be considered when only looking at this figure itself.

With this approach, a known value for the decimation stage execution time is obtained. This is not the case for the current system, where an approximation of this time is used. Measuring the exact time required by the FPGA-based decimation filter, on practical cases, turns out to be a complex task since there are many factors which play a role, such as sample buffering or FTDI controllers, for example. Therefore, the approximations are estimated based on the model characteristics, such as the length of the FIR filter within the decimation process.

On another aspect and as mentioned before, this proposed approach causes a higher system integration, since the need for the external FPGA disappears. This has positive effects on the overhead of transmitting the data from this device to the employed GPU, and is beneficial when considering issues as system cost, for example.

One of the most important aspects to remark about this approach is the validity of the generated holograms. As seen from the data in Figures 7.6a,7.6b,7.6c and 7.6d, the obtained backpropagated holograms have a small visual difference; the pressure distribution at the source plane is well interpreted and considered a reliable representation. The error distribution on the right of these figures shows that the errors concentrate on the zero crossings. Since this is a relative error, this behaviour is expected.

7.3.3 Walsh-Fourier Transform

This last proposed method is implemented in such a way that an approximation of the Fourier coefficients is achieved via the Walsh-Fourier Transform. As seen in Figure 5.4, the number of required Walsh coefficients varies depending on the frequency. Furthermore, for this implementation, only the 40 *most important* Walsh coefficients were employed. This represents a tradeoff between execution time and hologram error: the larger the number of used Walsh coefficients, the longer the execution time and the smaller the hologram error. And viceversa.

Results

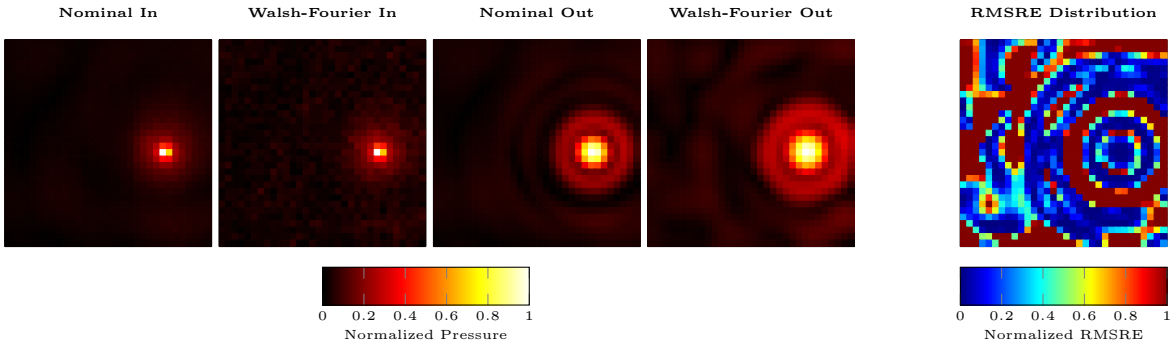
Regarding execution times, the implementation employing 40 precomputed Walsh functions took a total of 0.32 ms. to create a single hologram. Additionally, the implementation computing the required Walsh functions required 1.25 ms.

The results for the same datasets as in the previous section, but employing the Walsh-Fourier approach are seen in Figures 7.7a,7.7b,7.7c and 7.7d.

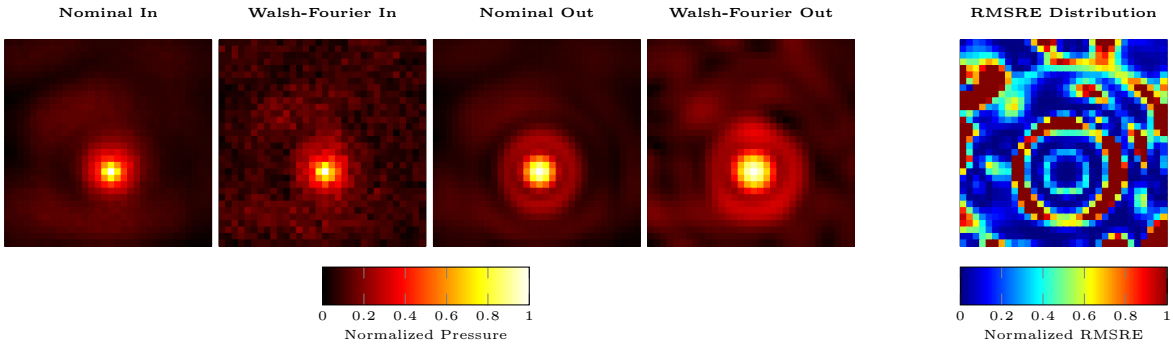
Analysis

This approach's execution times (0.32 ms. with precomputed Walsh functions and 1.25 ms. computing these ones) are far too high to be considered as a viable option at the moment.

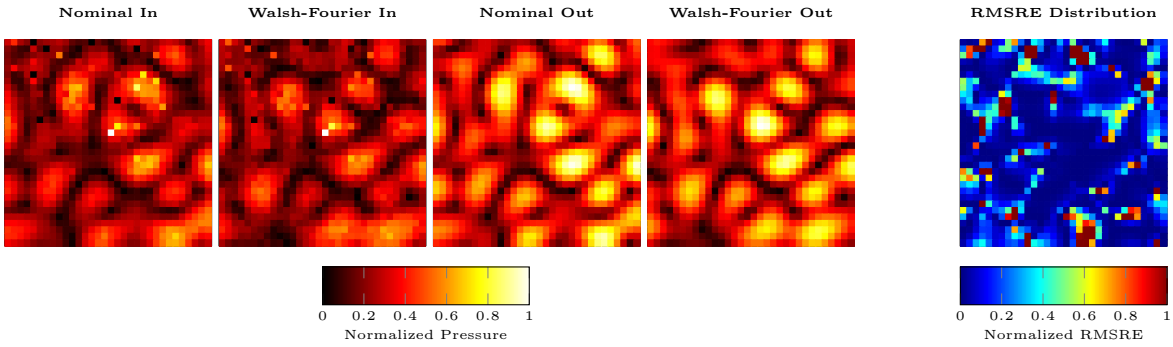
The fastest implementation, the one which employs the precomputed Walsh functions, fetches them from the global memory, which is the slowest memory in the device. Moreover, the slower implementation (Walsh function computation) is also memory-bounded. However, the reason is slightly different. These functions are computed based on the Rademacher functions. These basic functions are stored in the constant memory because they fit here (15 Rademacher functions $\times$ 4096$\frac{bytes}{function} <$ 64 kB of the constant memory). Even though the constant memory is optimized for broadcasting, its amount of accesses is



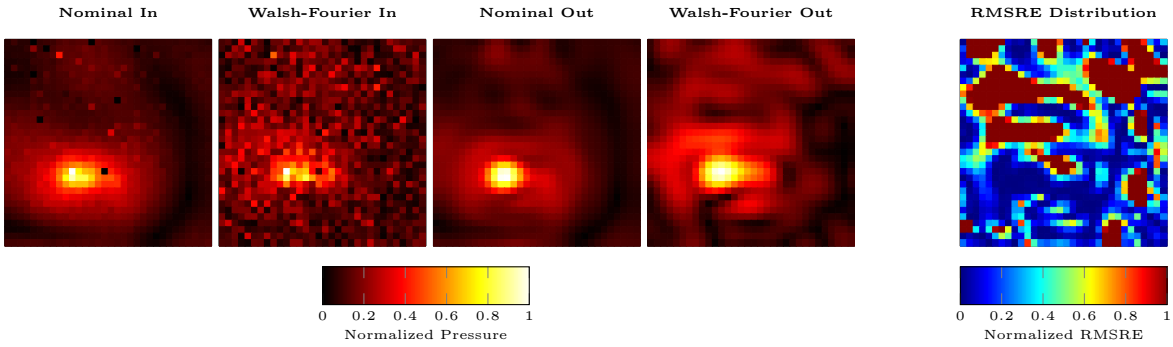
(a) Dataset $D1$ - 1000 Hz - RMSRE of 25.3% and NSAD of 4.26% (Output holograms).



(b) Dataset $D2$ - 1000 Hz - RMSRE of 21.1% and NSAD of 7.6% (Output holograms).



(c) Dataset $D3$ - 220 Hz - RMSRE of 17.4% and NSAD of 5.9% (Output holograms).



(d) Dataset $D4$ - 824 Hz - RMSRE of 30.7% and NSAD of 9.4% (Output holograms).

Figure 7.7: Input and Output holograms comparison between the Nominal implementation and the Walsh-Fourier Transform approximation. The normalized pressure is employed and four different datasets (Subfigures a, b, c and d) are analyzed. For all datasets, the K-space filter parameters are: cut-off wavenumber $k_{co} = 50\frac{rad}{m}$ and slope of 0.3. The RMSRE error distribution (normalized to the corresponding RMSRE value) between both output holograms is shown at the right.

too high. This increased access number is caused because each Walsh function requires, on average, between three and four Rademacher functions. Additional, costly extra operations, such as the Gray code conversion, take place. These are the main reasons why the execution time of this implementation is too high.

The computation of the dot product itself is not the problem in these implementations; the issue is loading the required Walsh functions. A proposed solution for this is exploiting the periodicity of the Rademacher functions to speed up the Walsh function computation, or to exploit the properties of the XOR truth table. For the first alternative, the value for a Rademacher function can be computed based on a work item's local index and Rademacher index, for example. In the case of the last proposed solution, it is useful to refer to Table 5.3, containing the XOR truth table. When an binary input time signal is XOR'ed with a binary 0, the sample is not altered. However, when it is XOR'ed with a binary 1, the result is *flipped* or *toggled*. Since the Walsh-Fourier transform approach is based on *population count*, an alternative could be to simply calculate the number of binary 1's that a Walsh function has, and with this information, compute how many binary samples in the time domain input will be flipped. In this sense, the current result of XOR'ing both functions is obtained based on the original population counts, avoiding the need of further XORs and, most importantly, of fetching the Walsh function. The concept of the Walsh function's *sequency* is employed in this alternative since it has a direct relationship to (it represents the proportion between) the number of binary 1's and 0's in a function. Further research into this topic is needed.

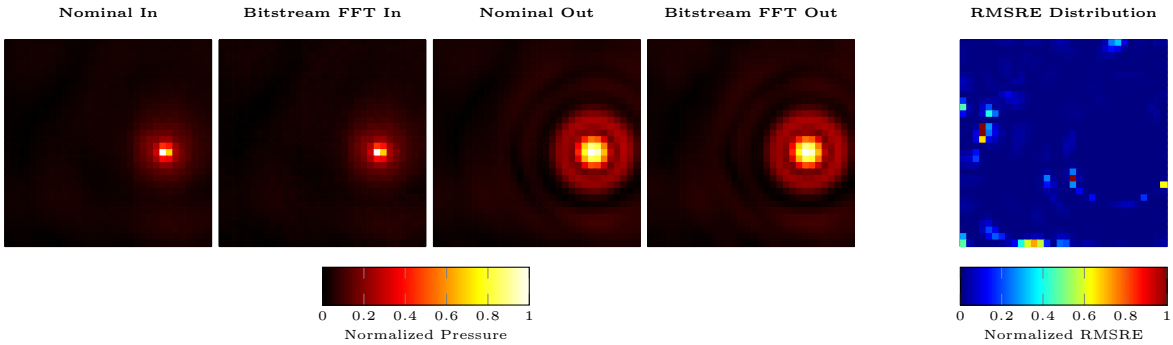
Regarding the obtained holograms, even though the visual interpretation is of good use (as pointed out by the small NSAD errors), there is a considerable RMSRE error. The reason for this is the *small* number (40) of Walsh coefficients computed to approximate the *real* Fourier coefficients. As mentioned before, this number represents a tradeoff between execution time and hologram differences. Increasing this number reduces these errors; furthermore, when this number is sufficiently large, the output of this approach is the same as applying the Fourier transform directly on the entire raw bitstream. As an example of what this approach could achieve, Figures 7.8a,7.8b,7.8c and 7.8d are given. Here, it is seen (as confirmed by the very small NSAD error) that the visual output is close to identical to that of the nominal approach. In fact, when looking closely, one could argue that more, smaller details can be observed in the Bitstream-based output hologram. It can be the case that these details were discarded in one of the filtering stages of the nominal approach, for example. This can explain the RMSRE error obtained with these holograms.

One last advantage of this approach is that the interpretability of the resulting hologram is higher when analyzing low frequencies. The nominal decimation filter adds a large DC offset to the input time-domain signals. Moreover, said DC offset is different depending on the channel (one column has a larger DC offset than the other). When performing the time-to-frequency domain transform, the energy contained in the DC component bin is very high. The amount of energy leaking to the neighboring frequency bins also increases, therefore distorting the original information (example in Figure 7.9a). On the other hand, the raw bitstream has a DC offset of zero, and its corresponding DC component bin in frequency domain has, consequently, less energy. This makes the information in the neighboring bins still be interpretable. An example of this situation can be seen in Figure 7.9b.

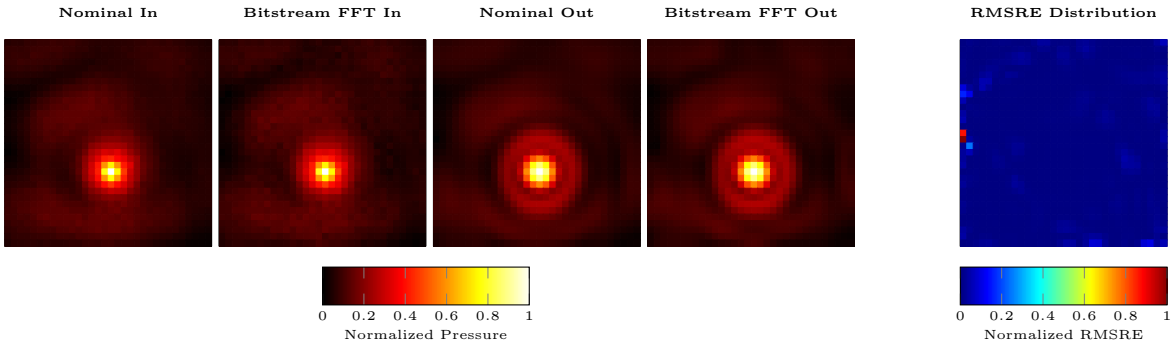
7.4 2D Frequency-Kspace Domain Transform

Results

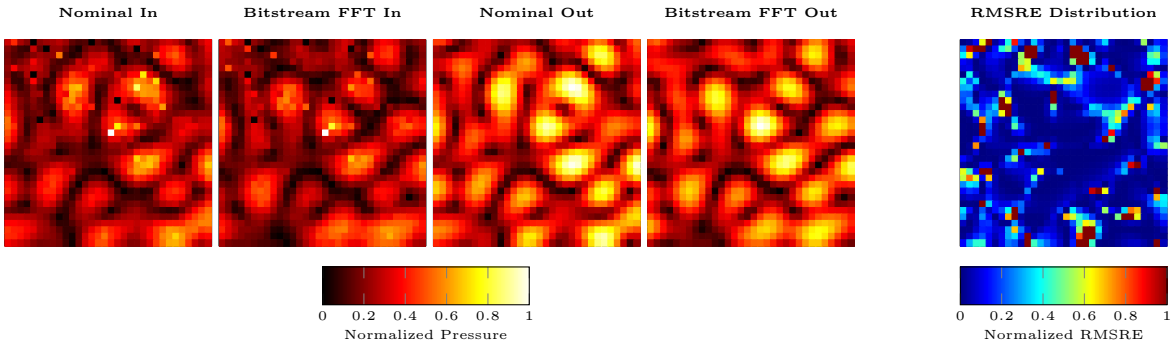
The $2D$ spatial frequency to K-space domain transform is carried out by a $2D$ FFT implemented by the clFFT library. The execution times of the $2D$ forward transform, for different number n of holograms, is shown in Figure 7.10, whereas the roofline model for the same configurations of the $2D$ FFT is shown in Figure 7.11. The FFTs executed for each hologram consisted on transforming a 96×96 complex matrix, representing the 32×32 complex matrix representing *original aperture size*, and the extrapolated data surrounding this originally measured area.



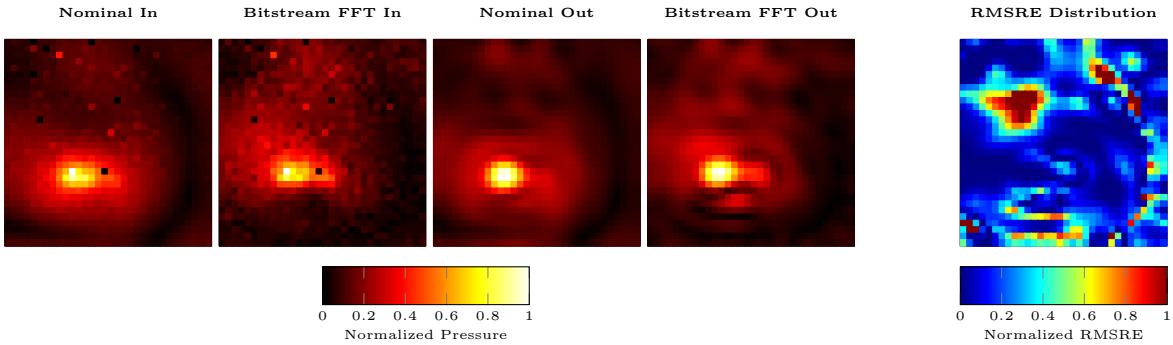
(a) Dataset $D1$ - 1000 Hz - RMSRE of 1.2% and NSAD of 0.01% (Output holograms).



(b) Dataset $D2$ - 1000 Hz - RMSRE of 2.7% and NSAD of 0.006% (Output holograms).

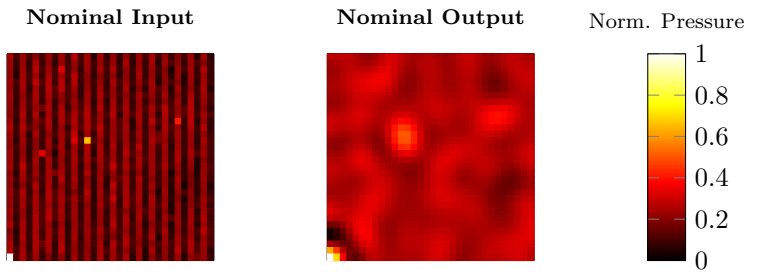


(c) Dataset $D3$ - 220 Hz - RMSRE of 6.2% and NSAD of 0.9% (Output holograms).

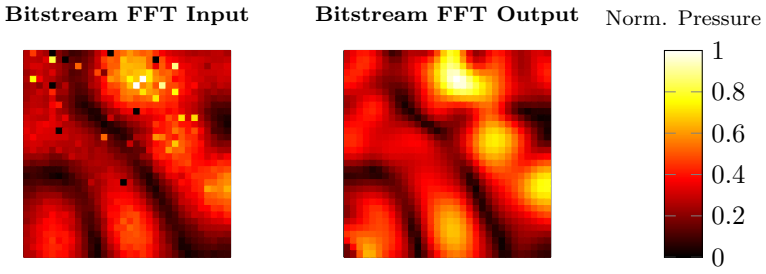


(d) Dataset $D4$ - 824 Hz - RMSRE of 8.4% and NSAD of 1.1% (Output holograms).

Figure 7.8: Input and Output holograms comparison between the Nominal implementation and the **exact** Walsh-Fourier Transform approach, which is equivalent to applying the Fourier Transform on the raw bitstream. The normalized pressure is employed and four different datasets (Subfigures *a*, *b*, *c* and *d*) are analyzed. For all datasets, the K-space filter parameters are: cut-off wavenumber $k_{co} = 50\frac{rad}{m}$ and slope of 0.3. The RMSRE error distribution (normalized to the corresponding RMSRE value) between both output holograms is shown at the right.



(a) $D3$ - Nominal approach.



(b) $D3$ - Exact Walsh-Fourier Transform

Figure 7.9: Dataset $D3$ is depicted in both subfigures (a and b) for a frequency of 55 Hz., which corresponds to the second frequency bin. Spectral leakage of the DC component (first frequency bin) in (a) causes information loss for low frequencies. This undesired effect does not appear in (b) since the leaked energy is less. The K-space filter parameters are: cut-off wavenumber $k_{co} = 50\frac{rad}{m}$ and slope of 0.3.

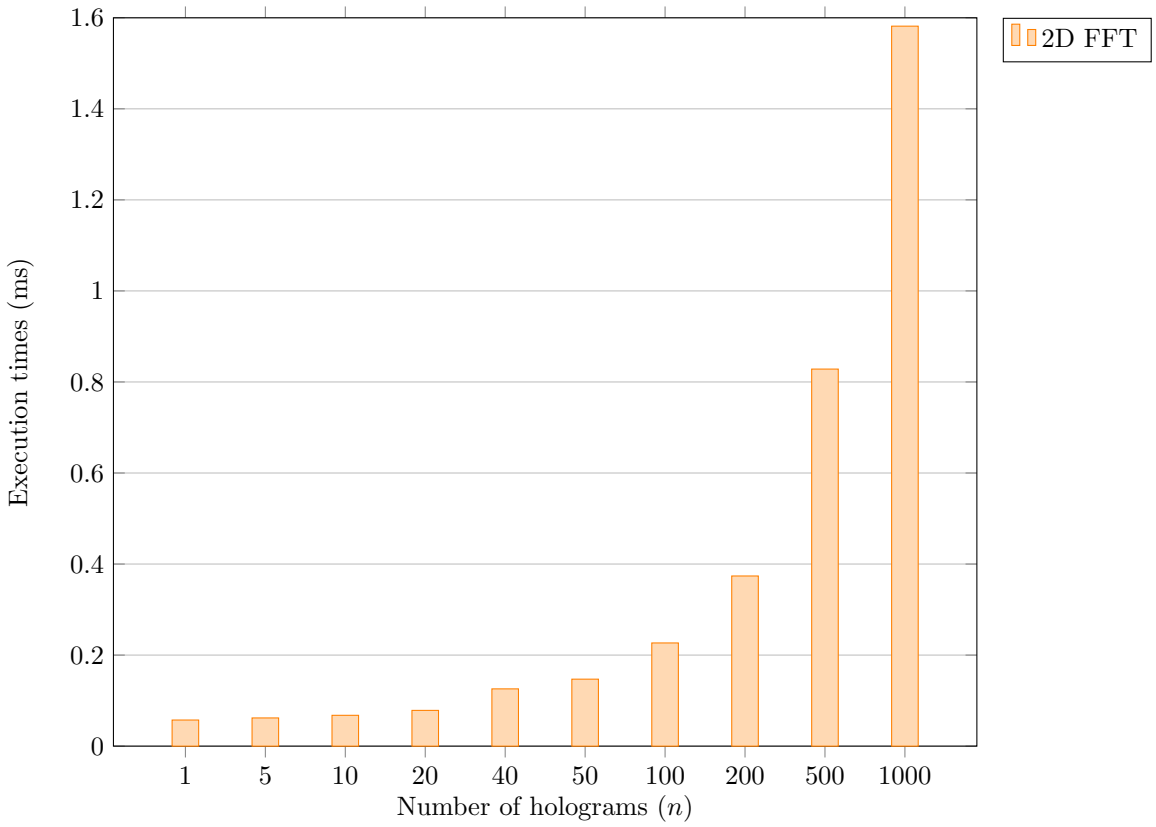


Figure 7.10: Execution time of the forward $2D$ FFT.

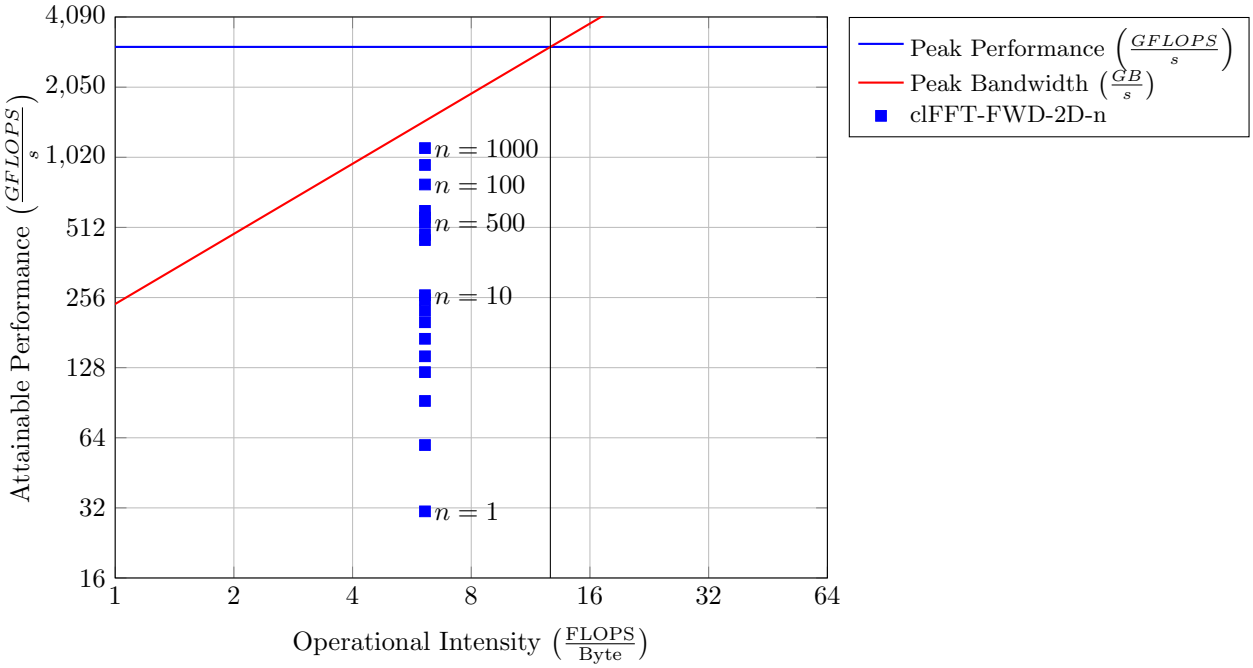


Figure 7.11: Roofline model of the $2D$ FFT for $n \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 100, 200, 500, 1000\}$ holograms.

Analysis

The clFFT $2D$ Fourier transform is an algorithm which performs considerably well in terms of execution times. It is seen from these results, for example, that applying the forward $2D$ FFT on 20 96×96 holograms is slightly faster than executing a $1D$ FFT on the datasets of $N_{ch} = 1024$ channels, each consisting of $N = 1024$ samples. This is one of the reasons why the focus in this project was set on the first FFT.

Proposing an alternative to reduce execution times on the $2D$ FFT is not straightforward. A n *Dot Product*-similar approach to the $2D$ Fourier transform would not even yield the small gains obtained in the $1D$ case, for example. The reason for this is that the complexity of such algorithm is no longer proportional to $O(N^2)$, but to $O(N^4)$, and this would eliminate the possibility of even marginal gains. A *fast* algorithm, such as the FFT, is the best option to perform this domain transform, since for its $2D$ version has a complexity of $O(N^2 \cdot log_2 N^2)$.

As seen in the roofline model in Figure 7.5, the GFLOPS attained by the $2D$ FFT increase along with the number of computed holograms. Nevertheless, this does not mean that the algorithm executes faster, as illustrated in the data from Figure 7.10. If the number n of holograms is such that the $2D$ FFT represents a bottleneck for the PNAH algorithm, an option could be to employ more than one GPU device to execute the PNAH algorithm from the $2D$ FFT onwards.

This alternative is proposed for the $2D$ FFT and not for the $1D$ one because in the latter case, batch size and the dataset size is such that the $1D$ FFT executes in an amount of time small enough such that, if it were split into more devices, the gains would be marginal.

In case a technique such as *zero-padding* would be employed to improve the output results' readability, the hologram size would increase. In this sense, the $2D$ FFT can achieve a better performance, however, the execution time would also increase. Considering this scenario, an alternative could be to implement the $2D$ Sparse FFT algorithm, proposed by [29]. This algorithm has considerable benefits in terms of time complexity when the proportion of non-zero coefficients is *very small* ($\approx 5\%$) when compared to the number of zero coefficients. Nevertheless, this number of non-zero coefficients might be too low when compared with the considered case of the inverse $2D$ FFT.

7.5 Entire PNAH Algorithm

In this section, the execution times of the entire PNAH algorithm are considered, where the n Dot Products and Basic Averaging approaches have been considered. Since the Walsh-Fourier Transform is still too slow to be considered for this real-time application, it has not been included.

Results

The n Dot Products approach is beneficial only for the case when the number of desired frequencies is $n \leq 5$. For larger values of n, the implementation presented in [6] should remain the same. In this sense, the execution times for the cases where $n \leq 5$ is shown in Table 7.2. Additionally, since this approach causes no difference in the input holograms, the inverse propagated pressure distributions have no error compared to the nominal approach.

The Basic Averaging approach causes a very small increase (almost negligible) of the execution time caused by the GPU-based equivalent of the decimation filter. Nevertheless, this method decreases the overall execution time due to the elimination of the time domain windowing step. Since the implemented Basic Averaging uses the n Dot Product approach, as the domain transform, the gains presented in Table 7.2 are also limited for $n \leq 5$. If this Basic Averaging were to be coupled with the FFT algorithm instead, the gains on the PNAH algorithm would only be the remotion of the $1D$ windowing. In this case, this would represent a 7.4% of execution time reduction in the best case (where $n = 1$). Even though this reduction is constant, the relative percentage decreases because when increasing n, the execution time does as well.

Regarding the last approach, the current GPU implementation of the Walsh-Transform is not yet suited to fulfill the required throughput of at least 1kHz for the PNAH algorithm to be executed in real-time. This is because creating a single hologram from the time input takes around 0.32ms, or $\frac{1}{3}$ of the time required for a single iteration. Further suggestions on this approach follow in the next chapter.

Number of holograms (n)		**1**	**2**	**3**	**4**	**5**
Exec. time (ms)	**Nominal**	0.643	0.666	0.691	0.721	0.742
	n Dot Products	0.577	0.619	0.658	0.687	0.733
	Basic Averaging	0.535	0.577	0.615	0.645	0.689
Relative gain (%)	**n Dot Products**	10.26	7.05	4.77	4.71	1.21
	Basic Averaging	16.79	13.36	11	10.54	7.14

Table 7.2: Total execution time for a single iteration of the PNAH algorithm. The relative gains are computed with respect to the nominal implementation.

Analysis

It is shown that reducing the PNAH algorithm execution times on the real-time implementation requires not only addressing the domain transform, but also proposing a modification on the time domain preprocessing stages. Disregarding the number of desired holograms, this preprocessing stage always takes place and represents a constant additional time.

From the n Dot Product approach, it was seen that keeping the same time domain preprocessing algorithm, the gains in the execution time are only obtained considering limited cases where the number of desired holograms is $n \leq 5$. To achieve larger gains, a modification to this preprocessing stage was proposed for the other two alternatives.

With the Basic Averaging approach, a larger gain was achieved since the need for the time domain windowing is removed. Additionally, a higher integrated system can be conceived, since there is no need

for additional external components: the entire algorithm is executed in the GPU platform. Although, it is important to consider whether the controller system, the one receiving the backpropagated holograms, is able to correctly operate with the holograms product of this method, where some differences exist when compared to the ones generated by the nominal approach.

Finally, even though the current implementation of the Walsh-Fourier Transform method has still a large room for improvement, it has a lot of potential due to the simplicity of the operations which this method requires. Even though a larger number of Walsh coefficients are required to improve the output's accuracy (currently using 40), as seen in Figure 5.4, for certain frequency components this method can yield greater advantages because of the small number of coefficients required. Similar to the Basic Averaging method, the Walsh-Fourier Transform removes the need for time domain windowing, which yields additional gains in execution time. Finally, the backpropagated holograms generated using this method tend to become more accurate as the number of Walsh coefficients increases. These tendency leads to what would be the results of applying the FFT directly on the raw bitstream; an example of this is provided in Figure 7.8a. These results raise a question on whether it would be preferable to use the binary input to get a more accurate representation of the pressure distribution at the source plane.

Chapter eight

Closure

This final chapter presents the conclusions of the work described in this document, and afterwards concludes with a couple of suggestions to continue the development along the direction proposed.

8.1 Conclusion

The real-time implementation of the PNAH algorithm presented in [6] currently achieves a throughput of 1kHz for at most 10 holograms. This same cited source optimized the $2D$ extrapolation method and proposed the use of the PLPBP algorithm. In this way, the most time consuming stage within the PNAH algorithm was addressed. To explore additional ways of reducing the execution times, the attention was drawn to the domain transforms employed by this algorithm, specifically, to the spatial time-frequency transform, usually executed via the FFT.

First, a modified DFT algorithm was proposed to reduce the execution times of the domain transform itself. Since this approach yielded limited gains when compared to the heavily optimized clFFT library implementing the FFT, some proposals were done in order to modify the time domain preprocessing stage. These alternatives operate on the raw bitstreams directly from the sensors (microphones). This implies that the decimation filter and time windowing are removed. Together with these discarded stages, computing only the required frequency bins, as initially proposed with the modified DFT algorithm, yields larger gains.

Even though the execution time reductions achieved in this project are modest, for a specific case, the improved throughput by 15% was achieved. However, the main contributions of this work are:

- The viability of PNAH analysis based on binary-generated input holograms is shown
- The analyses made on the domain transforms which exhibit potential execution time reductions.

The Basic Averaging and the Walsh-Fourier Transform approaches showed that an *accurate* representation of the source plane pressure distribution is achieved employing the raw, unfiltered input. There are some differences in the holograms presented as example; however, there is still room for further improvement of the proposed techniques. This is specially the case for the Walsh-Fourier Transform, where improving the approximation (increasing the number of Walsh coefficients) tends to resulting holograms which could arguably contain more details compared to the nominal output holograms.

Two final points are made as conclusion of this project:

- **Execution times:** For a number of desired holograms $n \leq 5$, the n Dot Product approach is suggested. Otherwise, the FFT implemented via the clFFT library is the best option. For $2D$ domain transforms, a DFT-like approach would worsen the execution times; thus, the clFFT library should be used in this case as well, independent of the n.

- **Binary input:** A domain transform applied directly on the raw bitstream yields accurate results on the source plane representation of a pressure distribution. Since employing the Fourier transform would be too costly because of the use of a larger number of time samples (even with the FFT), two alternatives to do so are presented and shown to generate significative results.

8.2 Future Work

The clFFT library remains the best option to compute the $2D$ forward and backward FFTs. As mentioned before, if a bottleneck appears at this stage, the possibility exists to split the workload in different GPU devices since the holograms are independent of each other. In this way, the execution times are reduced.

Additional improvements in the Walsh-Fourier Transform are suggested. The current GPU implementation can be improved by reducing the memory accesses to fetch the Walsh (or Rademacher) functions; as a proposed solution to this, consider the mentioned possibility of exploiting the XOR truth table. Moreover, instead of completely bypassing the FPGA device for this approach, this platform can be used to compute the Walsh-Fourier Transform. This was not currently possible since the memory capabilities of the device would limit the number of Walsh coefficients used to approximate an output, and the number of outputs. The reason for this is that each Walsh function requires 4096 bytes of memory, and accurate approximations can require tenths of these functions for one single hologram, which could mean hundreds for several holograms. A device with greater capabilities could be a candidate for such an implementation. Furthermore, if an optimization to the Walsh-Fourier transform as proposed is achieved such that the whole process is carried out based on the original population counts, this process can be executed in the current platform and the addition of Walsh coefficients to the approximation would be easier and faster. The main objective of this alternative is to reduce the PNAH execution times by enabling the FPGA to output the spatial-frequency domain holograms, instead of a time-domain output which still needs to be preprocessed.

Finally, it is worth mentioning that, following Amdahl's law, a lower bound for the execution time of the PNAH algorithm computing n output holograms is the execution time when $n = 1$. The PNAH algorithm is comprised by a sequential and a parallel section. The sequential section (time domain preprocessing and $1D$ domain transform) is independent from the number of required holograms. The parallel section is formed by the following computation stages (spatial-frequency and K-space domains). This parallel section can in theory be executed by a separate device for each hologram; this is the reason why, when $n = 1$, the lower limit on execution time is achieved. The best execution time when $n = 1$ achieved in this work is approximately 0.53 ms, which represents a current throughput close to 2 kHz.

A Walsh-Fourier Transform implemented in the FPGA device could improve the execution times, and in consequence the throughput, in approximately 15% as seen in the relative execution times in Figure 3.5b. However, the largest bottleneck of the PNAH algorithm is still the spatial-frequency domain extrapolation stage: this process incurs in several memory accesses to estimate the new values. Future GPU devices where memory access latency is reduced, or bandwith improved, will bring considerable improvements to the real-time implementation of PNAH.

Bibliography

[1] E. G. Williams, *Fourier Acoustics: Sound Radiation and Nearfield Acoustical Holography.* Academic Press, 1999.

[2] R. Scholte, "Improved Source Localization Techniques in Planar Near-Field Acoustic Holography," Master's thesis, University of Twente, 2004.

[3] P. van Dalen, "Transient Planar Near-field Acoustic Holography," Master's thesis, Eindhoven University of Technology, 2012.

[4] P. P. Vaidyanathan, *Multirate Systems and Filter Banks.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[5] R. Scholte, "Fourier Based High-resolution Near-Field Sound Imaging," Ph.D. dissertation, Eindhoven University of Technology, 2008.

[6] W.J.N. Ouwens, *GPU-accelerated Real-Time Transient Planar Near-field Acoustic Holography*, Master's paper, Eindhoven University of Technology, 2013.

[7] B. Baker, "How delta-sigma ADCs work," Texas Instrument Inc., Tech. Rep., 2011.

[8] E. Hogenauer, "An Economical Class of Digital Filters for Decimation and Interpolation," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 29, no. 2, pp. 155–162, Apr 1981.

[9] Khronos Group - OpenCL, https://www.khronos.org/opencl/.

[10] F. Harris, "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform," *Proceedings of the IEEE*, vol. 66, no. 1, pp. 51–83, Jan 1978.

[11] clFFT Libraries Website, https://github.com/clMathLibraries/clFFT.

[12] C. V. Loan, *Computational Frameworks for the Fast Fourier Transform.* Cornell University, 1992.

[13] P. N. Derk Reefman, "Why Direct Stream Digital is the best choice as a Digital Audio Format," *Audio Engineering Society*, 2001.

[14] S. Park, *Principles of Sigma-Delta Modulation for Analog-to-Digital Converters.* Motorola, 1993.

[15] FFTW Website - Pruned FFTs, https://www.fftw.org/pruned.html.

[16] N. Ahmed, *Orthogonal Transforms for Digital Signal Processing.* Springer-Verlag, 1975.

[17] E. Chu and A. George, *Inside the FFT Black Box.* CRC Press, 2000.

[18] K.-h. Siemens and R. Kitai, "Digital Walsh-Fourier Analysis of Periodic Waveforms," *Instrumentation and Measurement, IEEE Transactions on*, vol. 18, no. 4, pp. 316–321, Dec 1969.

[19] Y. Tadokoro and T. Higuchi, "Discrete Fourier Transform Computation via the Walsh Transform," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 26, no. 3, pp. 236–240, Jun 1978.

[20] N. Yogi and V. Agrawal, "Application of Signal and Noise Theory to Digital VLSI Testing," in *VLSI Test Symposium (VTS), 2010 28th*, April 2010, pp. 215–220.

[21] B. Jacoby, "Walsh Functions: A Digital Fourier Series," *The BYTE Book of Computer Music*, 1977.

[22] A. R. M. Zulfikar; Abbasi, Shuja A.; Alamoud, "FPGA Based Walsh and Inverse Walsh Transforms for Signal Processing," *Electronics & Electrical Engineering*, vol. 18, 2012.

[23] Khronos Group, "OpenCL Specification v1.2," https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[24] Advanced Micro Devices, *AMD Graphics Cores Next (GCN) Architecture*, https://github.com/AMD-FirePro/SDK/tree/master/documentation, 2012.

[25] ——, *Southern Islands Series Instruction Set Architecture*
, https://github.com/AMD-FirePro/SDK/tree/master/documentation, 2012.

[26] N. Hahn, "Sound Field Simulation Using Extrapolated Loudspeaker Impulse Responses," in *Audio Engineering Society Conference: 52nd International Conference: Sound Field Control - Engineering and Perception*, Sep 2013.

[27] P. Williams, Waterman, "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures," 2008.

[28] Advanced Micro Devices, *OpenCL BLAS Manual*, http://clmathlibraries.github.io/clBLAS/, 2013.

[29] Andre Rauh, Gonzalo Arce, "Sparse 2D Fast Fourier Transform," *Proceedings of the 10th International Conference on Sampling Theory and Applications*, 2013.

Appendix A - HW and SW Specs

GPU Device	
Description	**Value**
Device name	AMD Radeon HD 7900 Series (Tahiti 7950)
Architecture name	Graphics Core Next (GCN)
Clock freq.	850 MHz
Memory bandwidth	240 $\frac{GB}{s}$
Number of Compute Units	28
Stream Processors per C.U.	64
Wavefront size	64
Total Stream Processors	1792
Global memory size	3 GB
Constant memory size	64 kB
Local memory size	32 kB

FPGA Device	
Description	**Value**
Device name	XC3SD3400A Spartan-3A DSP FPGA
DSP48As units	126
Distributed RAM bytes	47744
Block RAM bytes	290304

CPU	
Description	**Value**
Processor	Intel Core i7-3770 @3.4 GHz
OS	Windows Server 2012 R2 64-bits
RAM	16 GB
Profiler	AMD CodeXL Profiler (version 1.4.5724.0)

Appendix B - Sorama Cam Mapping

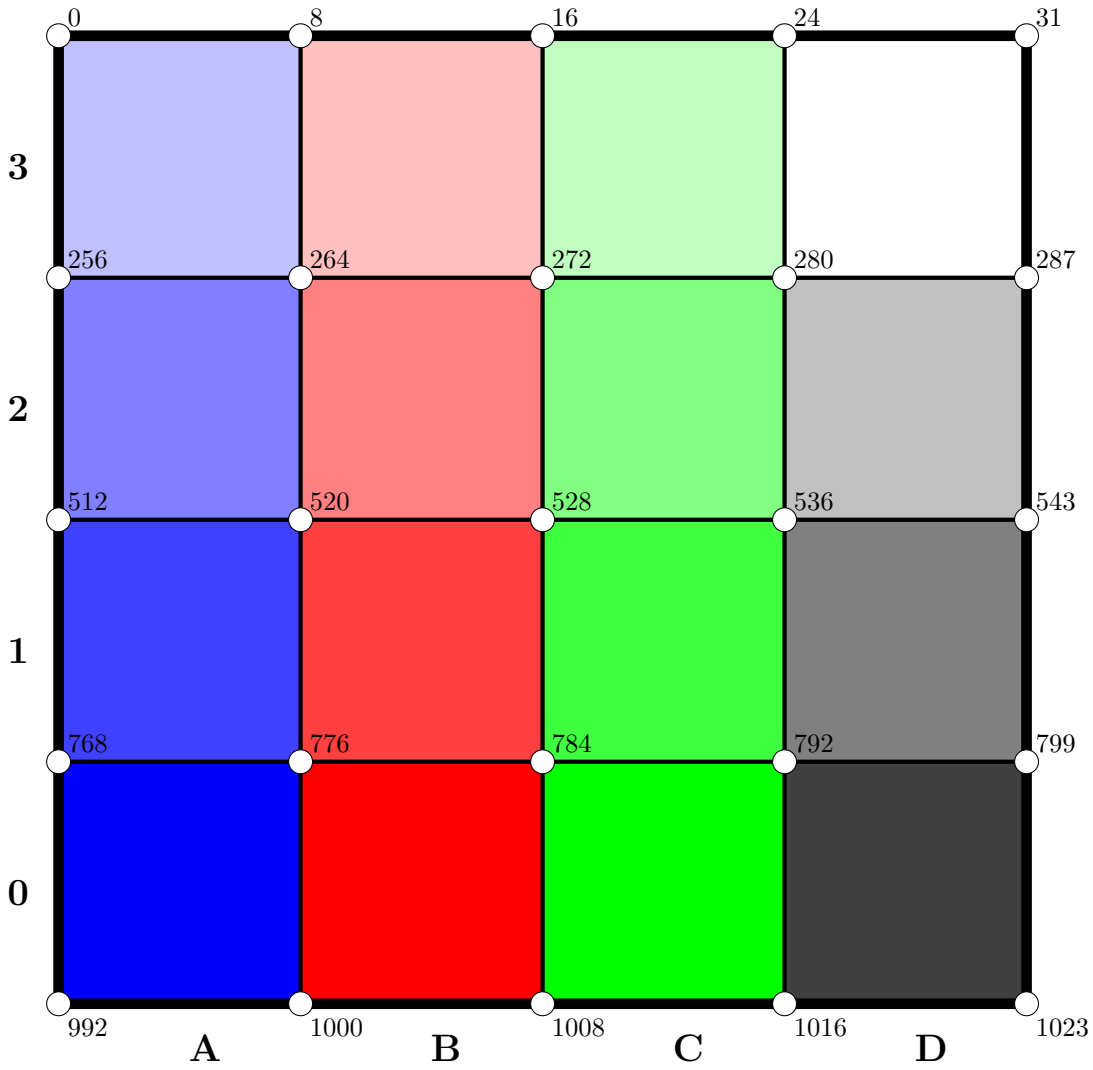


Figure 10.1: Sorama Cam formed by 16 different 8×8 microphone arrays. Some microphones are shown (white points) along with their corresponding camera channel number. The depicted point of view is from the back (the microphones are *pointing* towards inside the depicted plane).

This section describes the Sorama Cam MEMS microphone mapping, with the objective of detailing how the raw bitstream was obtained from a modified firmware loaded into the FPGA Platform. The data output of such a firmware follows the ordering here described. This data sequence was taken into account when creating the proper data acquisition software.

The Sorama Cam, illustrated in Figure 10.1, consists of 16 individual arrays, each labeled with their corresponding column letter and row number. The order in which the data from the arrays is fetched follows a columnwise (left to right) order, and within each column, it follows an ascending order (zero to three). In this sense, the array sequence is: $\{A0, A1, \ldots, A3, B0, B1, \ldots, D3\}$. This, along with the microphone numbering depicted in the same figure, constitutes the mapping at the *camera level*.

For each individual array, a different microphone mapping (*array level*) is employed and is depicted in Figure 10.2. In this case, the data is retrieved row-wise from left to right, and from the top down. This can be inferred from the few numbers next to their corresponding microphones depicted in the referred figure.

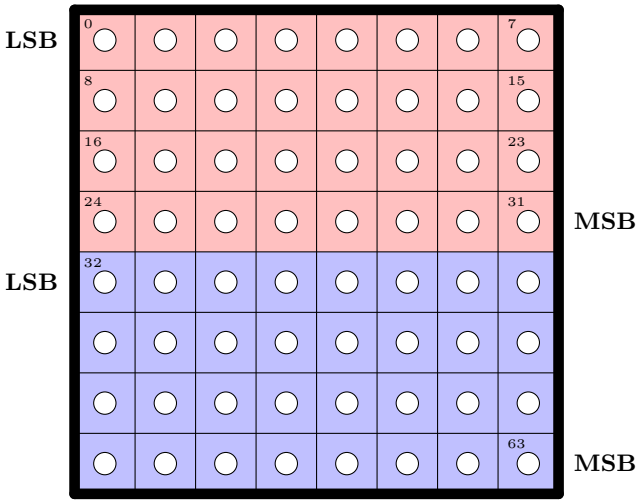


Figure 10.2: Single 8×8 microphone array. The depicted point of view is from the back (the microphones are *pointing* towards inside the depicted plane).

For the raw data acquisition, each microphone generates a 1-bit sample; therefore, two 32-bit chunks contain one time sample for every microphone in a single array. Following the array level mapping just described, the *first* 32-bit chunk of data contains the 1-bit samples corresponding to microphones $[0, 31]$ (top four rows, shaded in red in Figure 10.2), whereas the *second* 32-bit data chunk contains the binary samples of microphones $[32, 63]$ (lower four rows, shaded in blue). Taking into consideration that the system follows the *little endian* convention, the least significant bit (LSB) of every 32-bit chunk read corresponds to microphone 0 or 32, where as the most significant bit (MSB) belongs to microphone 31 or 63.

The raw binary data acquisition software combines the mapping at these two levels (camera and array) to correctly assign the bits to their corresponding microphone. In this case, the first 32 32-bit data chunks contain the first bit sample for all the microphones. Combining both mappings, it is seen that every two 32-bit data chunks belong to the the arrays following the previously described sequence: $\{A0, A1, \ldots, A3, B0, B1, \ldots, D3\}$. Once the mapping is done, the microphones acquire a new global (camera-level) index, as illustrated in Figure 10.2. For example, microphone 0 of array $A0$ has a global index of 768, microphone 0 of array $A3$ has global index 0 and microphone 63 of array $D0$ has global index 1023.