Escola Politécnica da Universidade de São Paulo

Engenharia de Telecomunicações

**Distorção digital de áudio utilizando GPU**

Alunos:

**Renan Ikeda Fernandes**

**Gustavo Faia Fagundes**

**Leonardo Posso Benetti**

Orientador:

**Guido Stolfi**

**São Paulo**

**2021**

**Sumário**

[**Resumo** 3](#_26in1rg)

[**Introdução** 3](#_lnxbz9)

[**Motivação** 3](#_35nkun2)

[**Processamento paralelo em GPU** 4](#_1ksv4uv)

[**Materiais** 4](#_44sinio)

[**Jetson** 4](#_2jxsxqh)

[**CUDA** 6](#_z337ya)

[**Áudio em programas** 8](#_3j2qqm3)

[**Metodologia** 11](#_1y810tw)

[**Requisitos** 19](#_4i7ojhp)

[**Bibliografia** 19](#_1ci93xb)

**Abstract**

Este projeto visou implementar distorção de sinais de áudio de forma inteiramente digital em tempo real, para alcançar este objetivo foi utilizado a GPU, possibilitando, então, o processamento paralelo a fim de melhorar a velocidade de processamento. Utilizou-se também a técnica de interpolação para aumentar a taxa de amostragem e filtragem a cada etapa de distorção para diminuir o aliasing proveniente das distorções. Como resultado, percebemos que as limitações para um processamento em tempo real são incompatíveis com o paralelismo fornecido pela GPU via API do CUDA. A despeito disso, foi demonstrado as vantagens da GPU frente a CPU em tempo de processamento e delay do processamento das amostras. Além disso, foi implementado um pipeline funcional, que além de aproveitar totalmente a capacidade de processamentos simultâneos na GPU, consegue distorcer um arquivo de áudio removendo aliasing provenientes de cada estágio de distorção. Finalmente, para demonstração, um sinal de sweep foi distorcido utilizando-se da arquitetura construída para remover o aliasing de uma distorção puramente digital.

# **1 Descrição do projeto**

O projeto teve como objetivo realizar a distorção de sinais de áudio de forma inteiramente digital em tempo real utilizando processamento paralelo em GPU e comparar com o processamento em CPU. Para isso, utilizamos o módulo Jetson TX2 da NVIDIA (Referência Jetson) que tem grande capacidade computacional, devido a presença de uma placa de vídeo dedicada à computação de uso geral, permitindo paralelismos massivos.

Inicialmente, a entrada de áudio do nosso sistema não estava implementada e era esperada a sua implementação durante o projeto. Entretanto, isso não foi possível devido a alguns problemas com o sistema operacional do Jetson. Portanto, decidimos demonstrar que a implementação em GPU é viável com processamentos offline, usando arquivos de áudio previamente gravados como entrada do sistema e escrevendo o resultado do sistema em um novo arquivo.

Em um sistema em tempo real, a leitura das amostras na correta taxa de amostragem é controlada pelo driver de áudio do sistema. Sendo assim, para justificar a viabilidade na GPU para processamentos em tempo real, utilizaremos como métrica o delay que o processamento gera desde a leitura da amostra do arquivo de entrada até a escrita da mesma processada no arquivo de saída, comparando seu resultado com seu tempo em CPU. Para melhorar a qualidade do sinal digital foi utilizado a sobreamostragem e filtros passa-baixas para evitar o rebatimento das frequências geradas pelas distorções não lineares aplicadas ao sinal de áudio.

# **2 Introdução**

## **2.1 Motivação**

A distorção de sinal de áudio não é estranha para nós. Qualquer som que chegue aos nossos ouvidos passará por alguma distorção, seja ela devido à fonte emissora, ao canal de comunicação pelo qual passou, à estrutura interna do nosso sistema auditivo ou todas elas juntas. Não é de se estranhar, então, que usamos da distorção para nosso próprio entretenimento, moldando o som, dando-o “sabores” distintos com finalidades diferentes. É comum, por exemplo, reconhecer um grupo musical ou instrumentista em específico através de quais efeitos de distorção estão presentes em uma música.

A distorção pode ser implementada de diversas formas, sejam elas de forma analógica, através de circuitos elétricos não lineares, de forma digital ou de forma mista. Em geral, ela consiste na deformação do espectro do sinal, de forma intencional ou não. Exemplos simples de distorção podem ser: a adição de harmônicas inexistentes no sinal original, aliasing, compensação ou defasagem de partes do espectro em relação a outras, etc.

Um exemplo de circuito analógico utilizado para a distorção de sinais de áudio é o pedal de guitarra. Sua vantagem frente ao processamento digital é que a discretização do sinal restringe a frequência máxima permitida em seu espectro, ao contrário do processamento analógico que, em teoria, permite a existência de infinitas harmônicas. Ao utilizar uma função não linear para causar distorção há o aparecimento de frequências harmônicas que podem estar acima da permitida para a discretização, gerando rebatimento e deteriorando o sinal. Para contornar esse problema seria necessário amostrar o sinal em taxas suficientemente altas, acima da usual utilizado em digitalização de áudio (48kHz), para que seja possível o aparecimento de harmônicas de alta frequência sem que haja o rebatimento. Infelizmente ao utilizar taxas maiores é preciso arcar com os custos de um melhor conversor analógico-digital (conversor AD) ou realizar o processo de interpolação do sinal. Ambos procedimentos geram a necessidade de lidar com o processamento extra devido a chegada de muito mais dados por período de tempo, o que não acaba sendo realizável utilizando uma CPU convencional.

Por esse motivo, neste trabalho busca-se utilizar GPU e processamento paralelo para implementar distorções de forma digital e em tempo real, devido à maior capacidade computacional desse ambiente.

## **2.2 Processamento paralelo em GPU**

Um programa de computador é composto por uma série de instruções. Nas arquiteturas mais tradicionais, quando o computador executa um programa ele carrega as instruções para uma memória volátil e as executa sequencialmente. A sequência das operações é importante dado que algumas delas do programa podem depender de resultados de operações anteriores. Entretanto, eventualmente podem haver operações que não possuem tal dependência. Com base nisso, começou-se a explorar o conceito de paralelismo no aumento da capacidade computacional.

A tendência foi a de aumentar o número de unidades de processamento, para que as operações não-sequenciais pudessem ser distribuídas entre diferentes unidades, alcançando um aumento na velocidade de execução dos programas. Nesse contexto as unidades de processamento gráfico (GPUs) lideram com a maior quantidade de unidades de processamento, sendo ambientes de computação paralela massiva (LAURI SAVIOJA).

Como as GPUs foram primeiramente projetadas para ser usadas com gráficos, foi necessária certa adaptação para que elas sirvam para computação de uso geral - ou general-purpose GPUs (GPGPU). Com base nisso, a NVIDIA desenvolveu uma API que possibilita utilizar suas placas de vídeo como GPGPUs, chamada CUDA - Compute Unified Device Architecture. Assim, podemos utilizar a arquitetura das GPUs para executar as partes paralelas dos programas em milhares de threads, diminuindo significativamente o tempo de execução.

De posse dessa poderosa ferramenta computacional, é possível alcançar a velocidade necessária para processar os dados em frequências elevadas com latências baixas a ponto de serem consideradas tempo real.

# **3 Materiais**

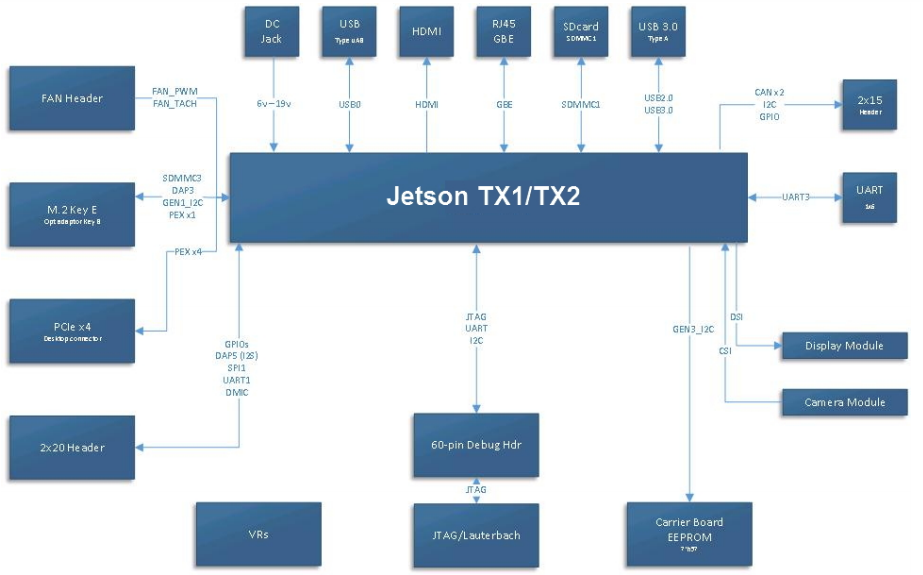
## **3.1 Jetson**

O projeto será desenvolvido utilizando o kit de desenvolvimento Jetson TX2 da NVIDIA, que adiciona integração ao módulo TX e facilita seu desenvolvimento tanto em hardware como em software. O Jetson TX é um sistema integrado em módulo, muito utilizado em desenvolvimento em Deep Learning e visão computacional, com grande poder computacional e baixo custo energético de 7.5W, dessa forma pode ser utilizado na ponta final do desenvolvimento de produtos. Seu poder computacional é composto de uma placa de vídeo Pascal de 256 núcleos e 2 processadores de 64-bits, um de quatro núcleos e outro de 2, além de 8 GB de memória RAM e uma memória flash de 32GB (Jetson TX2 Module, 2021)

| **GPU** | 256-core NVIDIA Pascal™ GPU architecture with 256 NVIDIA CUDA cores |
| --- | --- |
| **CPU** | Dual-Core NVIDIA Denver 2 64-Bit CPU Quad-Core ARM® Cortex®-A57 MPCore |
| **Memória** | 8GB 128-bit LPDDR4 Memory 1866 MHx - 59.7 GB/s |
| **Armazenamento** | 32GB eMMC 5.1 |
| **Alimentação** | 7.5W / 15W |

*Tabela 1: Especificações do módulo Jetson TX2*

Da perspectiva do usuário, é utilizado o sistema operacional da Linux, com possibilidade de utilização de entradas USB, HDMI, PCI, saídas de áudio, antenas e WiFi, permitindo sua utilização de forma fácil. O esquemático do kit de desenvolvimento e da arquitetura do módulo TX2 podem ser observados na Figuras 1 abaixo.



*Figura 1: Esquemático do kit de desenvolvimento do Jetson TX2* (Jetson Download Center, 2021)*.*

Devido às restrições de acesso ao Jetson pelos membros do grupo, decidimos configurá-lo para aceitar conexões via SSH (Secure Shell). Para isso, utilizamos a ferramenta disponibilizada pelo remote.it (site do remote.it) para fazer essa configuração. Com ela, temos acesso a um IP temporário de um proxy dos servidores do remote.it e um número de porta, com os quais conseguimos fazer as requisições SSH pela internet.

Com o servidor SSH aberto e de posse da senha de acesso ao Jetson, conseguimos interagir com um terminal remoto de comandos (CLI - Command Line Interface) e também fazer requisições SCP (Secure Copy Protocol) para transferência de arquivos. Ambos podem ser feitos por um terminal do windows, porém para as transferências de arquivos foi utilizado o programa WinSCP devido à interface gráfica mais conveniente (site do WinSCP).

## 

## **3.2 CUDA**

## **3.2.1 Sobre o CUDA**

O modelo de programação com CUDA foi desenvolvido para ter grandes similaridades com linguagens usuais de programação como C e C++, tendo abstrações de paralelismo - como hierarquia de grupo de threads, memória compartilhada e sincronização - expostas ao programador como extensões da linguagem (NVIDIA, 2021). O compilador NVCC (utilizada a versão 10.02.3) é o responsável por fazer essa interface entre a linguagem usual e a utilizada para tratar os paradigmas de paralelismo.

A figura X a seguir é referente à saída do programa deviceQuery que já vem na instalação do CUDA, e mostra as características da GPU utilizada.



## 

## **3.2.2 Acesso de memória**

## A NVIDIA usa a nomenclatura “host” e “device” para se referir ao processador da CPU e da GPU, respectivamente. A memória de cada um deles é separada, então para que seja possível que a GPU (device) manipule dados que estão na CPU, é necessário primeiramente que esses dados sejam transferidos para ela a partir da CPU. Posteriormente, para visualizar os resultados processados é necessário copiá-los da GPU de volta para a CPU. Esse processo, embora pareça simples, é um dos mais custosos em termos de tempo de processamento.

A função mais básica de cópia de memória é a cudaMemcpy, em que é especificado um endereço de memória do host, um endereço de memória do device, o tamanho dos dados que vão ser copiados e a direção, ou seja, do host para o device ou a direção contrária. Essa função é chamada durante a execução do programa no host e bloqueia os comandos subsequentes - o host não executa nenhum outro comando enquanto a cópia de memória não é concluída. Uma função parecida, mas que não bloqueia a execução do host é a cudaMemcpyAsync, que será discutida mais a fundo posteriormente neste relatório.

## **3.2.3 Paralelismo via Threads**

Como a API do CUDA foi desenvolvida para ser usada em um grande número de GPUs diferentes, com diferentes números de unidades de processamento, certos conceitos foram estabelecidos para que seja possível utilizar de maneira eficiente e padronizada essas diferentes arquiteturas. Um desses conceitos, é o de kernel. Um kernel se assemelha bastante com uma função nas linguagens C e C++ convencionais, porém possui a diferença que ele é executado pela GPU.

A GPU, por sua vez, consegue executar várias instâncias desse kernel simultaneamente, a partir de uma entidade abstrata chamada thread. Com isso, se cada thread operar em partes diferentes dos dados, obtém-se aumento na velocidade de processamento.

Para melhor organização de cada kernel, as threads são divididas em blocos (blocks) de threads e os blocos estão ordenados em uma grade (grid), como mostra a Figura 3.



*Figura 3: Organização das threads no CUDA* (NVIDIA, 2021)*.*

Dentro de cada bloco, as threads estão identificadas por um índice chamado “threadIdx”, que é um vetor com 3 coordenadas (do tipo dim3, definido pelo CUDA), que podem ser acessadas por “threadIdx.x”, “threadIdx.y” e “threadIdx.z”. Da mesma forma, dentro da grade os blocos estão identificados por “blockIdx”, também sendo um vetor de 3 coordenadas. Ao executar um programa que usa o CUDA, é necessário especificar o número de blocos, bem como o número de threads por bloco, que irão definir o número total de threads em uma execução:

Os kernels são especificados com um especificador “\_\_global\_\_”, que informa ao NVCC que essa função será chamada na CPU (host) mas será executada na GPU (device). Ele é executado por cada uma das threads em cada um dos blocos, totalizando execuções.

Para ter controle de cada thread separadamente, temos que gerar um identificador único para elas, a partir de seu “threadIdx” e “blockIdx”. Para isso, podemos acessar no kernel as variáveis gridDim e blockDim, ambas do tipo “dim3” citado anteriormente. Como o nome sugere, elas fornecem as dimensões da grade e do bloco, respectivamente. Ou seja, gridDim informa quantos blocos há em cada uma das dimensões da grade e blockDim informa quantas threads há em cada uma das dimensões do bloco. Por exemplo, na figura 3 gridDim e blockDim teriam as componentes:

De posse desses valores, podemos calcular um identificador único para as threads fazendo:

Com esse identificador, podemos fazer com que cada thread atue sobre uma porção específica dos dados, obtendo paralelismo à nível de dados. Como um exemplo simples disso, podemos pensar na soma de dois vetores de tamanho N: alocando N threads, cada thread pode ficar responsável por realizar o cálculo de apenas uma posição do vetor de resultados, dada pelo seu identificador. Assim, invés da CPU percorrer cada uma das posições dos vetores para realizar a soma, podemos fazer todas as somas ao mesmo tempo utilizando as threads, obtendo um aumento de velocidade.

Esse exemplo de soma vetorial foi utilizado para testar a programação com CUDA. Foram alocados 3 vetores de tamanho N e dois deles foram inicializados com valores aleatórios. Então o seguinte kernel foi invocado com N threads:

\_\_global\_\_ void vct\_add(float \*out, float \*a, float \*b, int n)

{

int i = threadIdx.x;

out[i] = a[i]+b[i];

}

Podemos observar que a variável “i” tem o mesmo valor de “id” para quando temos apenas um bloco na grade com threads apenas na dimensão x – as variáveis “threadIdx.y” e “threadIdx.z” são iguais a zero para todas as threads e também é igual a zero. Ou seja, “i” é um identificador único para cada thread, fazendo-a atuar apenas em uma determinada posição do vetor.

Vale ressaltar que a estrutura do CUDA permite ter no máximo 1024 threads por bloco. Portanto, o exemplo acima teria que ser dividido em mais de um bloco caso se queira alocar mais de 1024 threads para o processamento, mudando a maneira que se calcula a variável “i” para que ela calcule um identificador único para cada thread, como descrito anteriormente.

## **3.2.3 Paralelismo via Streams**

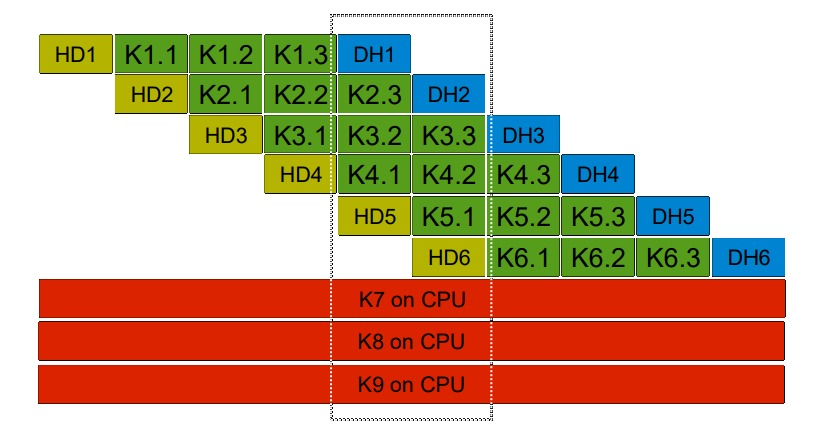
No entanto, paralelismo a nível de dados não é a única forma de paralelizar operações usando CUDA. Outro recurso importante em CUDA são Streams, que consistem em filas de trabalho a serem realizadas no device. O host coloca operações a serem realizadas para uma Stream específica em uma fila de trabalho e continua sua operação imediatamente, por sua vez o device agenda os trabalhos a serem realizados pelas Streams de acordo com os recursos disponíveis. Assim, cada Stream funciona individualmente de forma sequencial, ou seja, suas operações seguem uma fila ordenada (FIFO) e não podem acontecer ao mesmo tempo. Porém, operações em diferentes Streams podem acontecer ao mesmo tempo, de forma paralela.

Todas as funções do device são realizadas por algum Stream, mesmo que não especificado, seja ela uma cópia de memória ou um kernel específico. Quando a Stream não é referenciada, a função é executada pela Stream Default, uma Stream que contém propriedades de sincronismo diferente das outras Stream. A Stream Default não opera de forma concorrente com outras Streams, ou seja, funções executadas por ela só podem ser realizadas depois que todas funções agendadas para as outras Streams anteriormente foram executadas, da mesma forma outras Streams não podem operar enquanto a Stream Default está ativa. Esse comportamento entre a Stream Default e as demais é exclusivo para o sincronismo de operações entre Streams, mas pode ser evitado se as Streams declaradas pelo usuário forem criadas usando flags específicas.

A forma em que funções dentro do device podem ser realizadas simultaneamente consiste de um tipo de paralelismo denominado de pipeline. Dessa forma, além de kernel 's internamente poderem ter suas operações feitas de forma paralelas por threads, kernel´s diferentes podem estar sendo executados pelo device, alcançando um ganho de paralelismo ainda maior. Na Figura X um exemplo de pipeline é exibe como kernel´s diferentes são executados ao mesmo tempo usando Streams diferentes, inclusive a cópias de memórias (DH e HD na Figura).

A Nvidia disponibiliza uma ferramenta para a análise de programas feitos usando CUDA chamada de Visual Profiler. Além de métricas de eficiência de um kernel, a timeline do programa também é disponibilizada, podendo ser visualizado o ordenamento de funções executadas pelo device, em que tempo de execução foram realizadas, duração, kernels sendo realizados por Streams diferentes etc. No tópico Resultados exemplos de timelines geradas podem ser encontrados.

Como já mencionado anteriormente no tópico 3.2.2 cópias de memórias por padrão, usando a função cudaMemcpy são síncronas, bloqueando qualquer operação no host seguintes até a cópia de memória ser realizada. A despeito disso, a função cudaMemcpyAsync não possui esse caráter síncrono, podendo realizar cópias de memória assíncronas com o host, desde que a função seja especificada a uma Stream. Vale ressaltar que duas cópias de memórias feitas por Streams diferentes só podem ser executadas simultaneamente se a direção da cópia de memória seja diferente, uma do host pro device e a outra do device pro host.



*Figura X: Streams concorrentes (Referencia PDF* [*https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf*](https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf) *)*

## **3.2.4 Sincronismo entre CPU e GPU**

Uma das características do CUDA é que durante a execução do programa do host (CPU) são feitas chamadas à API do CUDA para execução dos kernels, mas o fluxo do host não é interrompido como acontece em uma função comum. No caso dos kernels, não é necessário esperar que sua execução acabe para realizar a próxima operação no host, devido ao fato de as operações da CPU e GPU serem realizadas em dispositivos diferentes.

Entretanto, podem haver operações no host que dependem de resultados gerados pelo device, então é conveniente que se tenha funções para permitir que o processamento do host seja bloqueado até que o processamento da GPU acabe. Nesse contexto, a função mais comum é a *cudaDeviceSynchronize*, que bloqueia a execução do host até que todos os processamentos da GPU se encerrem.

Em alguns casos, pode ser que uma operação do host dependa apenas do resultado de um kernel específico, sendo que a GPU pode estar processando diversos kernels diferentes. Sendo assim, não faria sentido esperar todos os processamentos da GPU acabarem, quando podemos esperar que apenas o kernel desejado para continuar a execução no host e deixar os outros ainda em execução. Por isso, existem algumas funções que realizam tal tarefa de sincronização específica, cada uma com suas particularidades.

A função cudaStreamSynchronize, por exemplo, bloqueia a execução do host apenas enquanto uma Stream específica não acabou seu processamento, possibilitando realizar as operações subsequentes no host depois que ela acabar mesmo que as outras Streams do device ainda não tenham terminado.

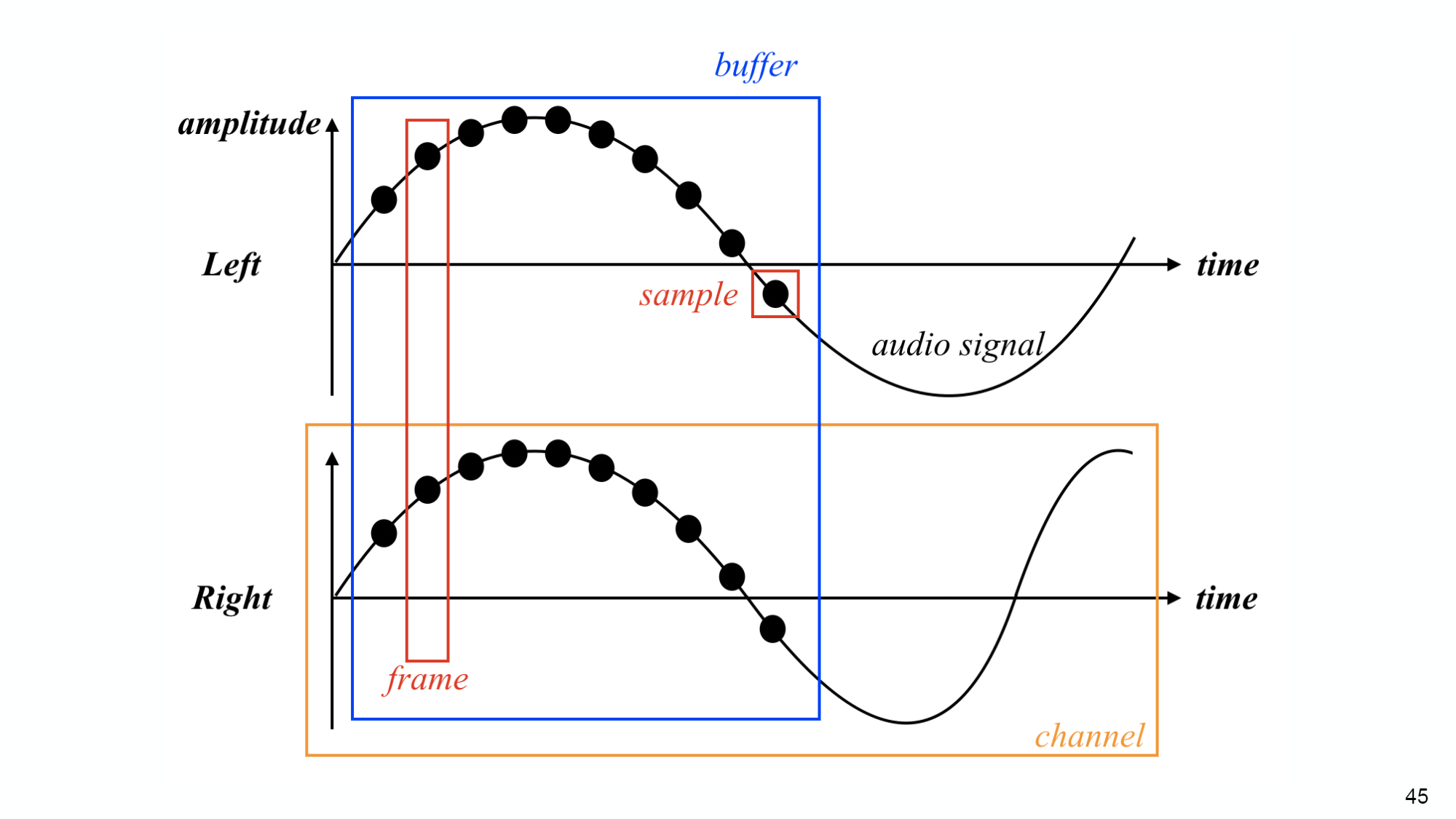
Pode-se definir também Eventos em CUDA, que são entidades que atrelamos à Streams no programa através da função cudaEventRecord. Os Eventos possuem diversas funções de sincronização, dentre as quais podemos citar a cudaEventSynchronize e cudaEventQuery. A primeira possui um comportamento equivalente à usar a cudaStreamSynchronize, bloqueando a execução do host até que a operação atrelada ao evento seja concluída. A cudaEventQuery, entretanto, não tem esse comportamento bloqueante. Ela é apenas uma “consulta” ao estado do evento, retornando se a função atrelada à ele ainda está em execução ou já foi concluída. Com isso, é possível escolher no host executar ou não certas operações, dependendo se a GPU já terminou o processamento atrelado ao Evento.

## **3.3 Áudio em programas**

**3.3.1 Conceitos Básicos**

Em um computador, sinais de áudio são compostos por amostras ordenadas do sinal de tempo contínuo. As principais características que os definem são a frequência de amostragem e o número de bits reservados para representar cada amostra. A frequência de amostragem está intrinsecamente ligada com a máxima frequência que o computador é capaz de representar sem que ocorra rebatimento, pelo critério de Nyquist. Já o número de bits de cada amostra define uma resolução para o sinal amostrado pois quanto mais bits menor o erro de quantização.

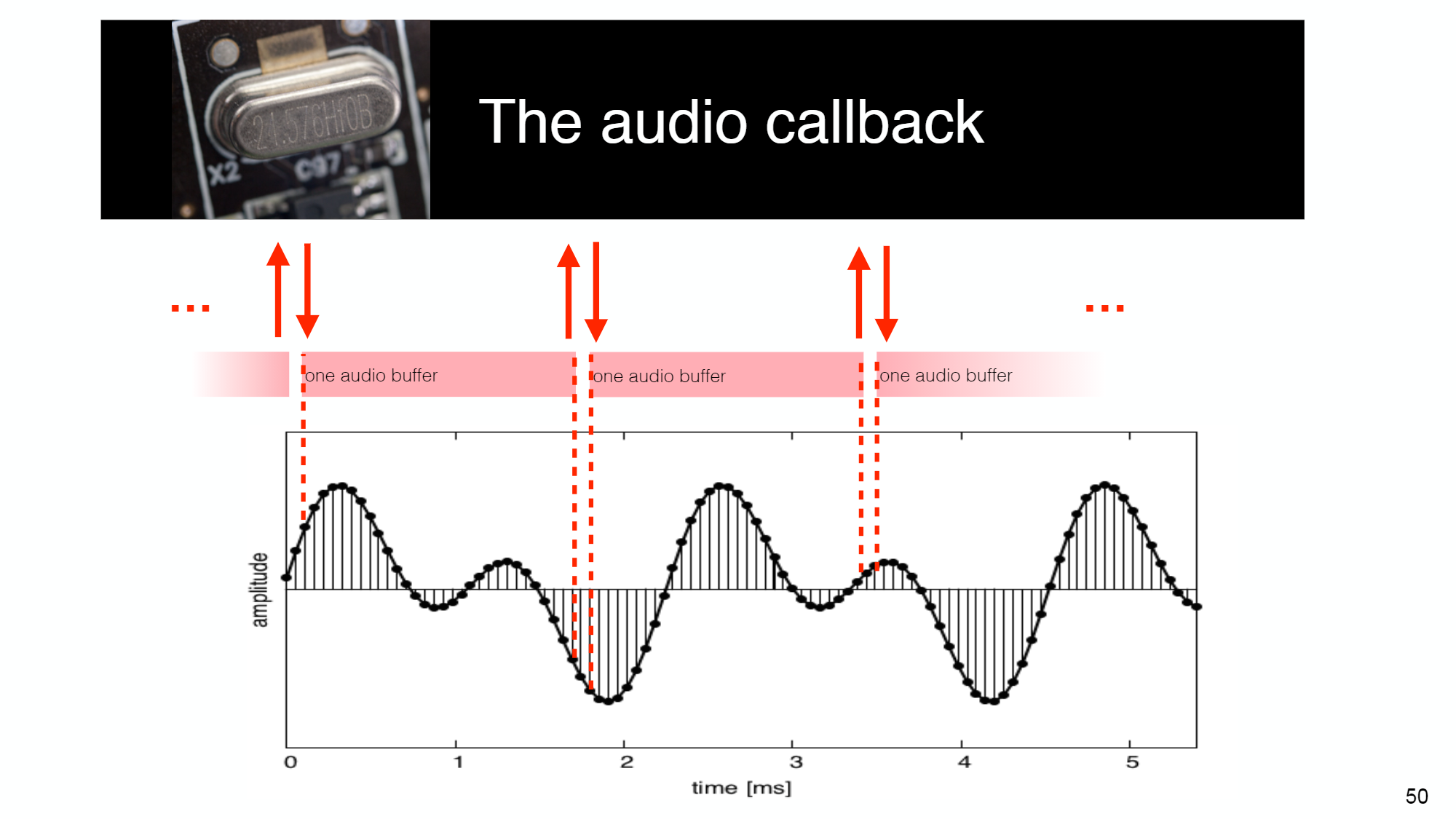
Quando o computador precisa mandar essas amostras para a saída de áudio, elas são organizadas em canais e frames. Canais se referem às diferentes saídas de áudio presentes no equipamento de som, como os canais “esquerda” e “direita” para um dispositivo estéreo. Já os frames se referem às amostras de um mesmo instante de diferentes canais, como mostra a Figura 4. Nela, também é possível visualizar o buffer, que é definido por uma certa quantidade de frames



*Figura 4: Organização das amostras de áudio no computador* (DOUMLER, 2019)*.*

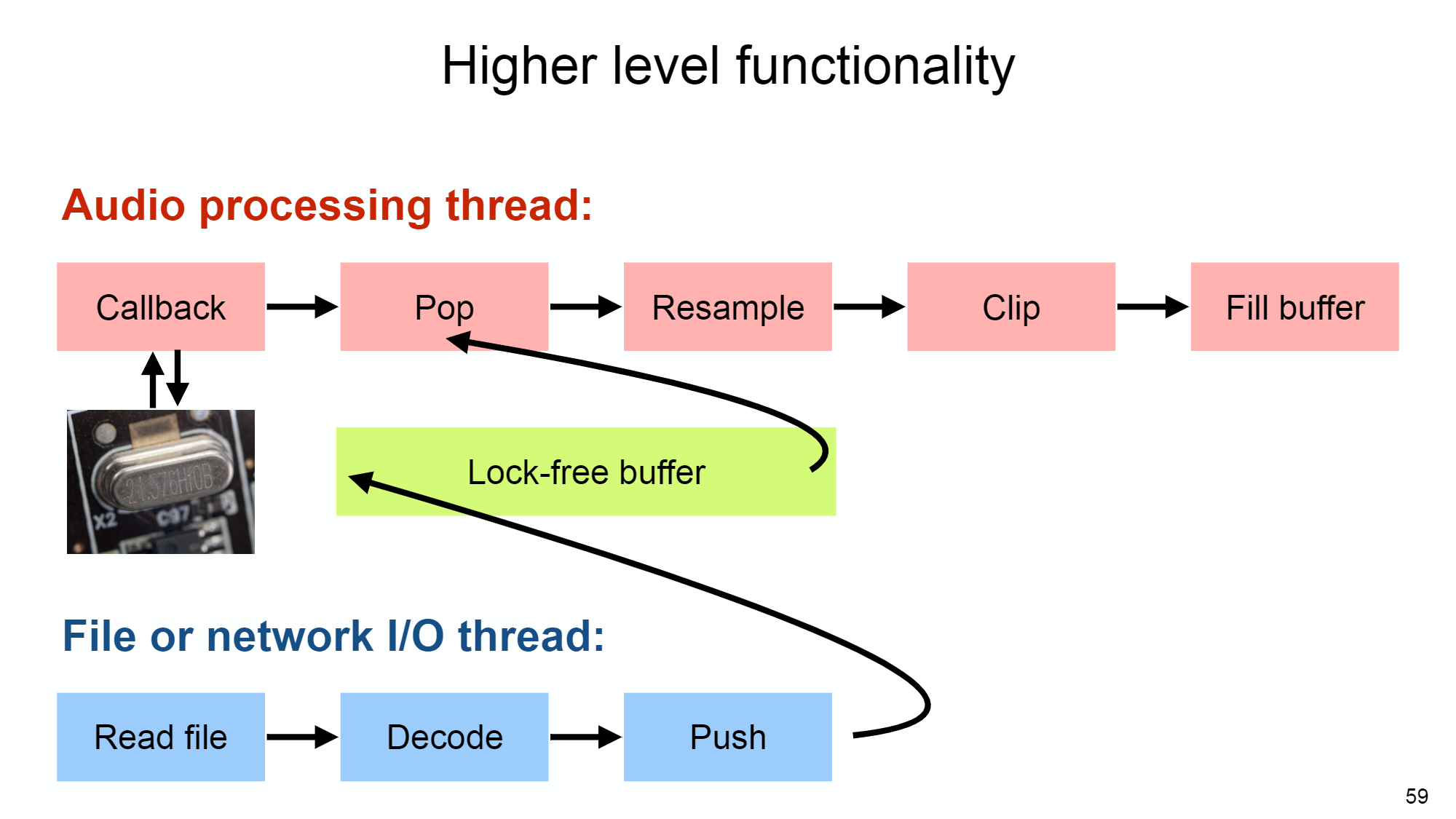
Para que o áudio seja reproduzido, o computador precisa utilizar a placa de som. Ela possui um temporizador que é capaz de gerar uma função de callback, responsável por pegar todas as amostras contidas em um buffer periodicamente e reproduzi-las nas saídas correspondentes aos diferentes canais. Como o ouvido humano possui grande sensibilidade para erros e descontinuidades em áudio, essa função de callback roda em uma thread de alta prioridade, para não ocorrer interrupções e atrasos na reprodução das amostras de um sinal de áudio.

Há um compromisso entre o tamanho do buffer utilizado e a latência do sistema, visto que a cada vez que a função de callback é executada as amostras do buffer são reproduzidas até o fim. Por exemplo, se temos uma frequência de amostragem de 48kHz e 256 amostras no buffer, a função de callback tem que ser chamada aproximadamente uma vez a cada 5 milissegundos (). Quanto menor o número de amostras no buffer, mais rápido o callback tem que ser chamado, exigindo mais da CPU do sistema, mas resultando numa latência menor. A Figura 5 mostra a atuação da função de callback na reprodução do sinal.



*Figura 5: Função de call-back* (DOUMLER, 2019)*.*

Para que um programa consiga ler um arquivo de áudio e reproduzi-lo, ele deve seguir os passos representados na figura 6, que consistem em decodificar o arquivo de áudio e colocar as amostras em uma fila FIFO (First In First Out) que será alimentada para que a função de callback consiga pegar as amostras do sinal nos momentos corretos, e fazer eventuais processamentos de adequação de taxa de amostragem e clipagem.



*Figura 6: Funcionamento da reprodução de áudio em programas* (DOUMLER, 2019)*.*

As figuras desta seção foram todas retiradas de uma palestra realizada pelo grupo “C++now”, que reúne especialistas em C++ para tratar de tópicos relevantes para o desenvolvimento da linguagem. Nesta palestra (DOUMLER, 2019), era discutida a possibilidade de gerar áudios a partir de programas em C++, porém a linguagem ainda não possui essa funcionalidade implementada na biblioteca padrão para escrever a função de callback, sendo necessário utilizar bibliotecas de terceiros para este fim.

Tais bibliotecas fazem interface com as APIs específicas de cada plataforma de computadores, tais como Mac, Windows e Linux. Com elas, é possível definir as características desejadas da função callback para que áudio seja reproduzido, como a taxa de amostragem, tamanho do buffer, entre outras.

Dentro do contexto deste trabalho, como estamos com uma plataforma Linux no Jetson, encontramos uma biblioteca chamada PortAudio, que pode utilizar da API ALSA (Advanced Linux Sound Architecture) para escrever as funções necessárias para reprodução de áudio. Com ela, foi possível reproduzir sinais simples com códigos em C, porém a impossibilidade de implementar uma entrada de áudio no nosso sistema fez com que não utilizássemos essa biblioteca no resultado final. Mesmo assim, para implementações futuras das ideias contidas neste relatório, pode-se encontrar informações sobre a PortAudio em (PortAudio Documentation).

A solução encontrada para ler arquivos de áudio pré gravados foi a biblioteca libsndfile - versão 1.0.31 (<https://github.com/libsndfile/libsndfile>). Com ela, conseguimos ler as amostras de arquivos no formato .wav e obter informações essenciais sobre ele, como a taxa de amostragem, número de canais em que o arquivo foi gravado e número de frames. Além disso, podemos também gravar novos arquivos a partir dos dados processados, permitindo ouvir e conferir os resultados obtidos.

Para ler as amostras do arquivo, utilizamos a função *sf\_read\_double()*, que coloca as amostras do arquivo em uma variável do tipo double em C++. No programa, essas variáveis lidas são números reais limitados ao intervalo de -1 e 1, representando as amplitudes do sinal de áudio.

**3.4 Configurações ambiente de desenvolvimento**

Essa seção tem como objetivo elencar as bibliotecas utilizadas, bem como o ambiente de desenvolvimento, para que seja possível a replicação dos resultados apresentados por um leitor interessado.

Conforme mencionado, o projeto foi realizado para execução no kit de desenvolvimento Jetson TX2, que apresentava como sistema operacional um Ubuntu 18.04.5 LTS (Bionic Beaver). Geralmente os kits de desenvolvimento do Jetson já vem com CUDA instalado, porém a instalação pode ser feita através do site da nvidia (https://developer.nvidia.com/cuda-downloads). A versão utilizada do CUDA foi a 10.02.3.

Foi utilizado em conjunto com o CUDA a biblioteca libsndfile, versão 1.0.31 (<https://github.com/libsndfile/libsndfile>). A comando necessário para compilar o programa resultante dessa combinação necessita, portanto, da inclusão dos headers e binários necessários para a libsndfile no compilador NVCC da NVIDIA:

nvcc -o <nome\_executável> <nome\_do\_script.cu> -I/usr/local/include -lsndfile

Como também foram desenvolvidos programas de teste, que rodam na CPU em C++, também foi necessário utilizar um compilador dessa linguagem. Assim, foi utilizado o G++ (Ubuntu/Linaro 7.5.0-3ubuntu1~18.04), versão 7.5.0. A compilação foi realizada de forma semelhante ao nvcc:

g++ -o <nome\_executável> <nome\_do\_script.cpp> -I/usr/local/include -lsndfile

Foi utilizado também o Visual Profiler, que parte das ferramentas da do nvidia-toolkit, para gerar visualizações das operações realizadas na GPU. Isso permitiu realizar tanto deputações no nosso código, quanto provas de conceito. Entretanto, a versão do CUDA instalada no Jetson não permitia o uso dessa ferramenta dentro, por incompatibilidades de arquitetura. A solução que encontramos foi gerar arquivos de “profiles” que podemos observar em outros computadores, com arquitetura diferente, que suportam a ferramenta. Para gerar esses arquivos utilizamos o seguinte comando:

sudo -E env "PATH=$PATH" nvprof -f --export-profile <saida.prof> <programa\_executável>

# **4 Metodologia**

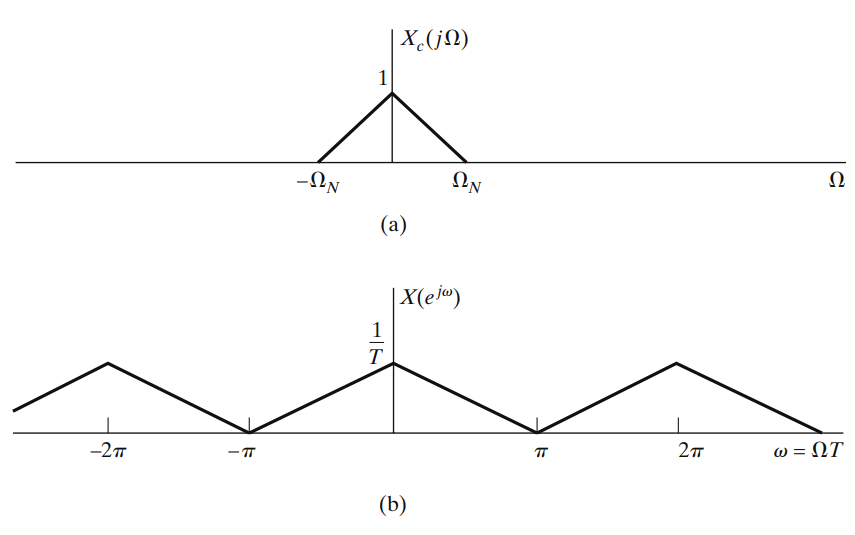
**4.1 Interpolação**

Como mencionado, o processo de distorção adiciona componentes espectrais no sinal, múltiplas das suas componentes fundamentais, que podem passar da frequência de Nyquist do sinal e gerar aliasing. A maneira mais direta de evitar esse fenômeno é aumentar a frequência de Nyquist, aumentando a frequência de amostragem do sinal, de forma que as harmônicas geradas pelas componentes do sinal não a ultrapasse ou se ultrapassar estejam em uma magnitude menor devido a distância espectral maior até a frequência de Nyquist interpolada.

Em primeiro momento, podemos imaginar que seria mais fácil apenas aumentar diretamente a taxa com que o sinal é amostrado. Entretanto, esse procedimento exige instrumentos de digitalização de sinais extremamente potentes, e clocks de sistemas bastante precisos, o que é inacessível financeiramente.

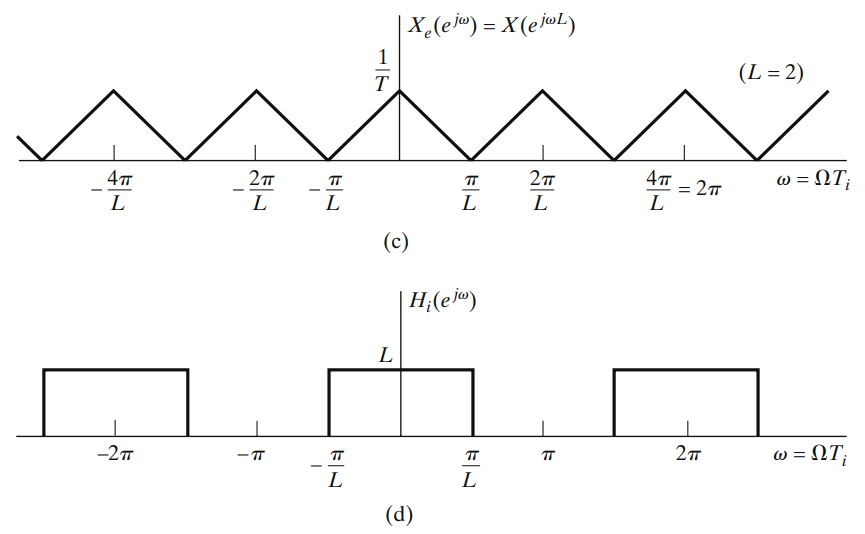
A melhor solução neste caso é aumentar a taxa de amostragem artificialmente, por meio do procedimento chamado interpolação. Ele consiste em inserir amostras novas entre as amostras originais do sinal, obtendo o valor delas a partir de uma operação matemática com as amostras originais do sinal. O resultado disso, efetivamente, é refletido como um aumento na taxa de amostragem proporcional ao número de amostras novas inseridas entre as amostras antigas do sinal. Para aumentar a taxa de amostragem por um fator , insere-se amostras novas entre cada uma das amostras antigas do sinal. O cálculo do valor correto dessas novas amostras segue o seguinte procedimento:

Partindo do sinal em tempo contínuo, faz-se a amostragem e obtém-se o espectro do sinal em tempo discreto, conforme a figura X. Note que o processo de amostragem multiplica o módulo do espectro por um fator de .



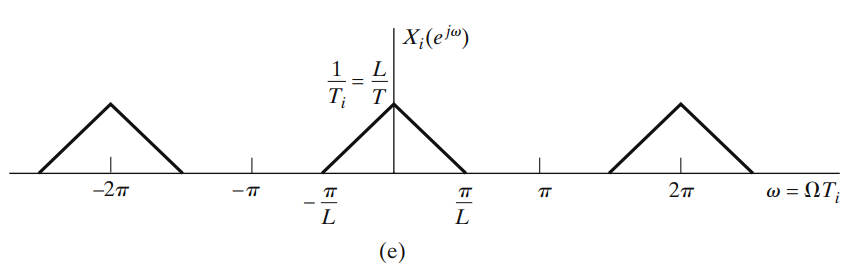
*Figura X: Espectro do sinal em tempo contínuo e tempo discreto* (OPPENHEIM e SCHAFER, 2012)*.*

Em seguida são inseridas amostras nulas entre as amostras do sinal, que causa uma “compressão” do espectro em tempo discreto por um fator de L. Portanto, para que as frequências discretas de 0 a contenham somente o espectro do sinal original (sem rebatimentos), seria necessário utilizar um filtro passa-baixas ideal, com ganho L e frequência de corte igual a .



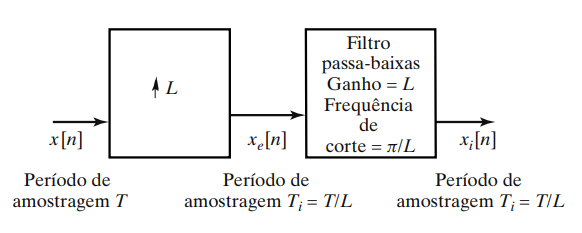
*Figura X: Alteração do espectro no processo de sobreamostragem em fator L e o filtro passa-baixas com corte em π/L* (OPPENHEIM e SCHAFER, 2012)*.*

O resultado desse procedimento é ter um sinal resultante com frequência máxima L vezes menor que a frequência máxima do sinal original.



*Figura X: Espectro final após a filtragem* (OPPENHEIM e SCHAFER, 2012)*.*

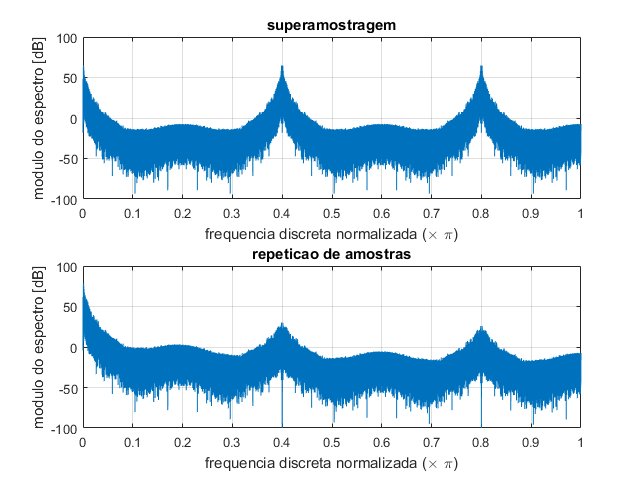
A figura X a seguir mostra um diagrama de blocos desse procedimento.



*Figura X: Diagrama de blocos interpolação* (OPPENHEIM e SCHAFER, 2012)*.*

O procedimento descrito acima teria resultados muito satisfatórios, se fosse possível ter comportamentos ideais no sistema. Porém, como sabemos, um filtro passa-baixas ideal não é realizável, sendo necessário utilizar filtros que possuem faixas de transição menos abruptas e ripples na faixa de passagem e de rejeição diferentes de 0. O filtro em si, portanto, causa certa distorção indesejada do sinal. Por esse motivo, utilizamos neste projeto filtro elípticos IIR para implementar o filtro de interpolação, devido à sua capacidade de proporcionar uma faixa de transição bem abrupta e atenuação na faixa de rejeição elevada para um número de coeficientes relativamente baixo. Além disso, filtros IIR são bem convenientes para processamento em tempo real devido à possibilidade de processar uma amostra por vez, ao invés de ter que esperar algumas amostras para o processamento poder começar como no caso de filtros FIR.

Outra escolha adotada no projeto foi implementar a interpolação primeiramente repetindo as amostras do sinal vezes, invés de acrescentar zeros entre as amostras do sinal. Esse procedimento melhora o espectro do sinal resultante pois as repetições do espectro do sinal original ficam com magnitude menor, como demonstrado na figura X, o que permite que o filtro passa-baixas seja menos restritivo.



*Figura X: Diferença de espectro entre os dois tipos de interpolação.*

**4.2 Distorção**

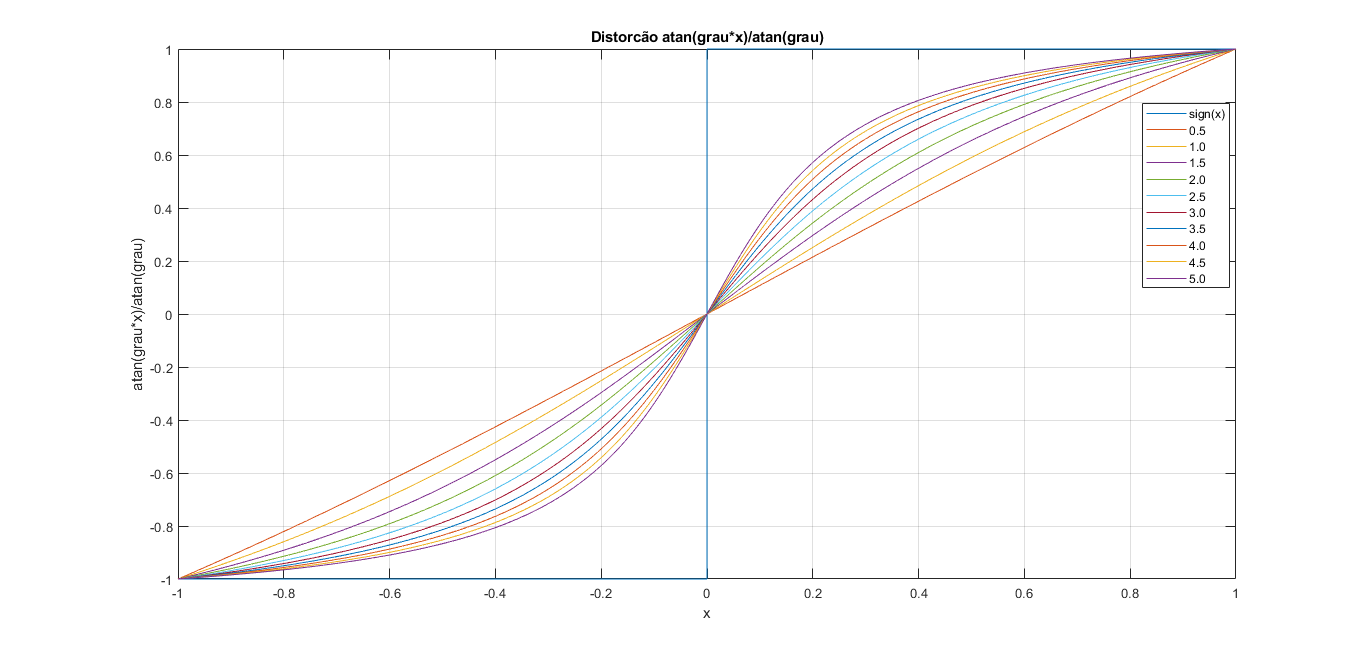
**4.2.1 Função não linear**

O cerne do projeto e processamento é a distorção. A distorção não será concentrada, isto é, não será realizada integralmente de uma única vez, sendo dividida em estágios de distorções mais leves. Após cada bloco de distorção leve há uma filtragem por um filtro passa-baixas, que tem função eliminar frequências geradas pela distorção que estão além da banda do sinal original, principalmente de frequências próximas às de Nyquist. Assim, conseguimos diminuir o efeito de intermodulação em distorções subsequentes das frequências altas, para que o aliasing gerado por elas seja desprezível. Portanto, a principal vantagem de se dividir a distorção em estágios repousa no fato de que para distorções mais leves, os parâmetros dos filtros se tornam cada vez menos restritivos, por isso os chamamos de filtros “suaves”, conseguindo-se evitar mais aliasing no sinal e atingindo uma melhor qualidade de som final.

Por mais que os filtros evitem em algum nível o aliasing gerado pela intermodulação das distorções, o aliasing de cada distorção é inevitável, apesar de poder ser atenuado com interpolações mais pesadas. A despeito disso, algum nível é tolerável (90 dB abaixo das frequências principais do sinal) pois os equipamentos de som não conseguem reproduzir sons com grandes diferenças de intensidade, passando-se como se o aliasing não existisse.

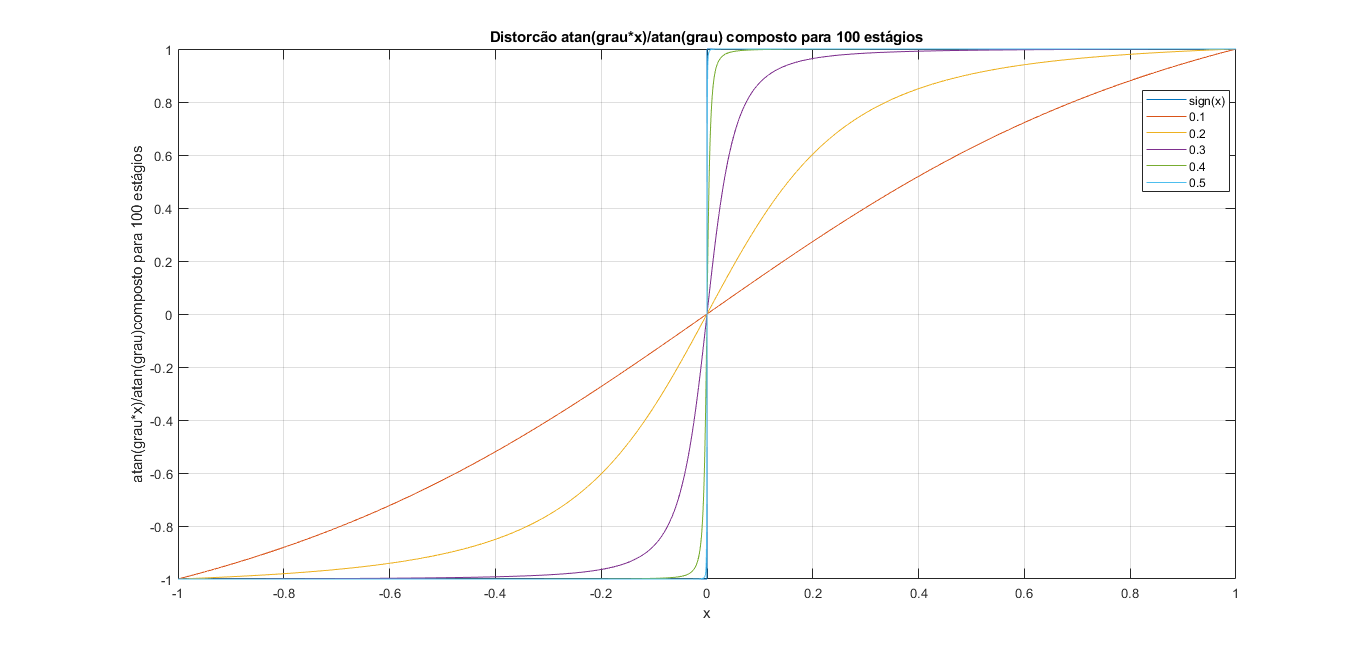
Inúmeras funções não lineares podem ser usadas no intuito desse projeto. Algumas funções podem ser listadas como polinômios, arco tangente, sinal, etc. No entanto, a função arco tangente apresenta algumas características convenientes para o projeto. A primeira vantagem é em nível de projeto, alterar o grau de distorção exercido por essa função em cada estágio reside em apenas uma mudança do seu domínio. Porém, devido ao fato da entrada e saída do sistema ser limitada entre +1 e -1, uma mudança na imagem da função também deve ser feita para valores limites (+1 e -1) serem mapeados nos limites da saída do sistema (+1 e -1), evitando assim ao máximo que o sinal não tenha que sofrer um processo de “clipping” no processo de escrita do arquivo de saída, o que acarretaria em mais aliasing no aúdio final. Denominando o grau de distorção como , a função não linear correspondente fica:

Para uma melhor visualização dessa função, inclusive para diversos , na Figura X é mostrado a imagem da função para o domínio limitado entre +1 e -1 , para vários graus de distorção e juntamente da função . A função foi tomada como o limite superior da distorção total que pode-se aplicar no sinal. Sua aplicação em um sinal de áudio corresponde sua deixar a forma mais próxima de uma onda quadrada, que contém exclusivamente os valores limites do sistema em considerado, +1 e -1. A sua aplicação em cascata não altera o resultado anterior, portanto uma distorção do tipo pode ser considerada como a distorção máxima que o sinal pode sofrer no sistema ao todo. Para maiores, então, mais próximo chega da função e tomando o chega-se de fato a função .

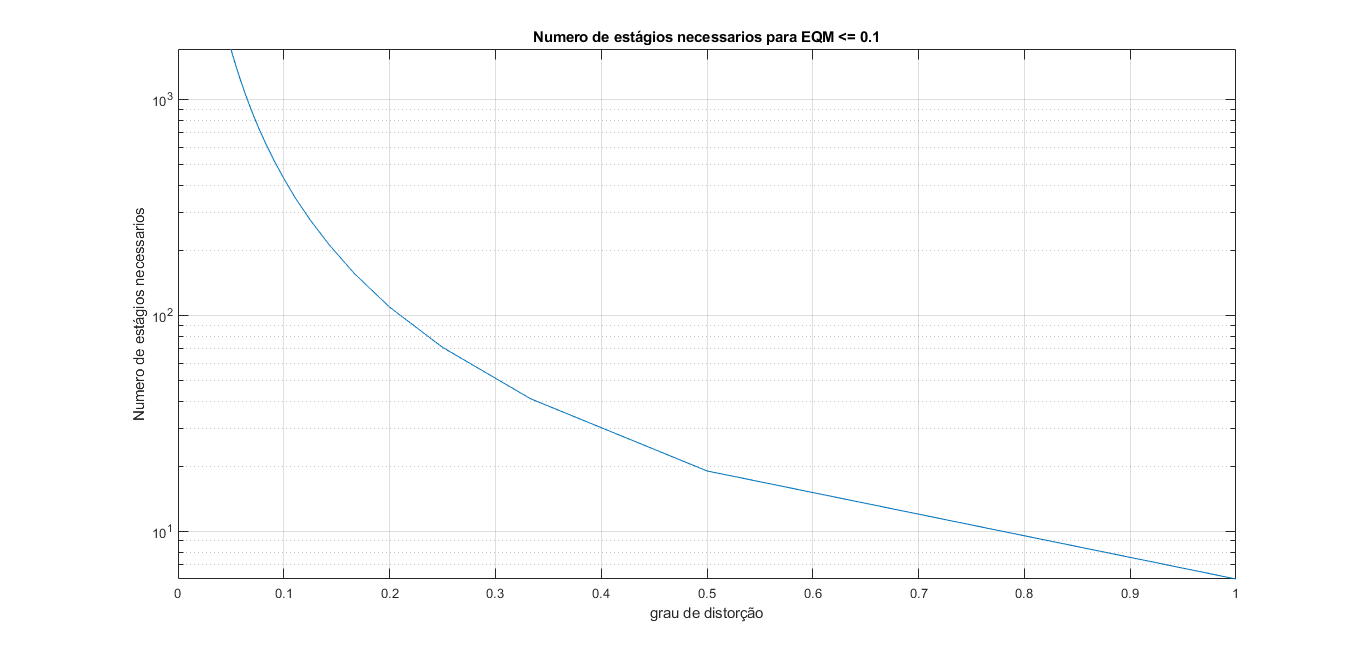


*Figura X: Função arco tangente para diversos valores de ⍺.*

A distorção total que será aplicada ao áudio depende do número de estágios de distorção em que o sinal irá passar, além do *⍺.* de cada estágio. Desprezando o efeito dos filtros passa-baixas que são presentes no final de cada estágio, a função equivalente que é aplicada ao sinal interpolado é o cascateamento do número de estágios totais da função não linear considerada. Ao todo o equivalente da composição de diversos arcos tangentes vai se aproximando da função como mostra a Figura X, feita para uma composição de 100 estágios, que equivale a uma distorção de grau maior (*⍺.)* ter sido aplicada. Assim, para uma distorção total de interesse, deve-se escolher para um certo *⍺*  um número de estágios totais que consiga fornecer essa distorção. Na Figura X é mostrado aproximadamente quantos estágios são necessários para alguns graus de distorção entre 0 e 1 chegar no mesmo nível de distorção total. O nível de distorção para o levantamento dos pontos foi considerado como erro quadrático médio entre a função composta por diversos arcos tangentes e a função ser menor que certo valor. No caso da Figura X, o número de estágios informado é o primeiro inteiro que fornece um erro quadrático médio menor ou igual a 0.1 .



*Figura X: Função arco tangente aplicada 100 vezes para diversos valores 𝛼.*



*Figura X: Número de estágios necessários para um mesmo nível de distorção*

**4.2.2 Análise da função arco tangente**

A segunda vantagem de se escolher a função arco tangente reside em nível de análise. A função arco tangente pode ser facilmente decomposta uma soma de série de potências. Mais especificamente, pode-se mostrar como sendo:

Tendo, portanto, a decomposição da função não linear como a soma de polinômios de ordens ímpares crescentes, sendo o polinômio de primeiro grau a parcela linear da função, podemos analisar o efeito da aplicação dessa função em um tom simples. Assim, tendo assim um tom digitalizado de frequência genérica , , como entrada da função não linear, temos:

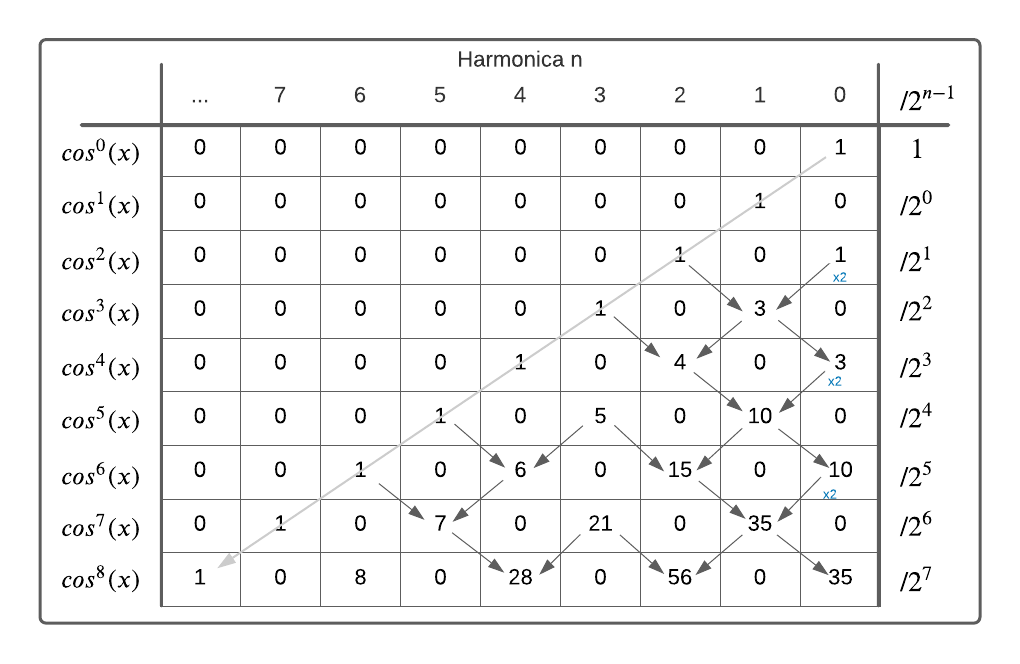
Potências de cossenos podem ser também decompostas como soma ponderadas de seus harmônicos. Mais especificamente, potências pares de cossenos podem ser decompostas como soma ponderadas dos harmônicos pares até o harmônico da mesma ordem da potência, junto de um nível DC. Por outro lado, harmônicos ímpares podem ser decompostos apenas como harmônicos ímpares até a ordem da potência, sem nenhum nível DC. Alguns exemplos de decomposição de potências ímpares de um cosseno :

O fator que multiplica cada harmônico de acordo com cada ordem da potência pode ser posto na forma matricial, com as colunas correspondendo aos harmônicos de (0 como nível DC) e as linhas correspondendo à potência do cosseno - . Dessa matriz, foi levantado um algoritmo que determina os coeficientes que multiplicam os harmônicos de para cada função , a partir dos coeficientes já conhecidos de potências baixas (ordem 2).

O algoritmo consiste em montar a matriz dos coeficientes das harmônicas da direita para a esquerda, começando do nível DC de . Os coeficientes nesta matriz serão considerados apenas o numerador de seu valor na forma obtida a partir da expansão de Taylor. Dessa forma, esses coeficientes terão que ser normalizados posteriormente pela respectiva potência de 2 da expansão - indicados à direita da figura X.

A partir do nível DC de forma-se uma linha formada por 1´s na diagonal secundária da matriz, formada pela enésima fundamental da enésima potência de . Sabendo-se que potenciais ímpares não possuem nível DC, a coluna mais à direita da matriz tem valor zero para potências ímpares. A partir do nível DC das harmônicas ímpares, formam-se diagonais secundárias constituídas apenas de 0 ́ s, pois sabe-se que harmônicos pares de potências ímpares e harmônicos ímpares de potências pares são 0.

O coeficiente da frequência fundamental (primeira harmônica) das potências ímpares são formados pela soma do coeficiente superior à esquerda com duas vezes o coeficiente superior à direita. O coeficiente referente ao nível DC desconhecido das potências pares são a cópia do coeficiente da fundamental da potência ímpar anterior. Os coeficientes das demais harmônicas são formados pela soma do valor superior à esquerda com o superior à direita, sem a necessidade de multiplicar o da direita por dois. A Figura X ilustra a aplicação do algoritmo para calcular os coeficientes dos harmônicos da oitava potência de um tom.



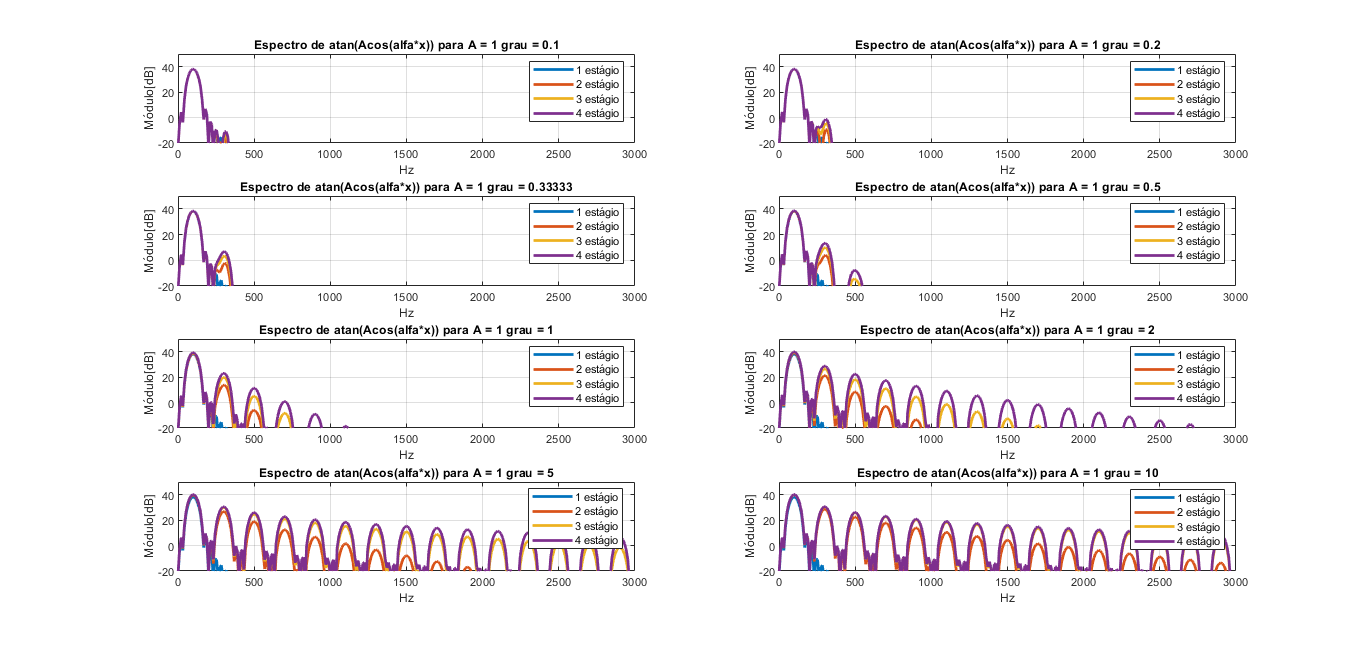
*Figura X: Tabela muito louca.*

Tendo em posse os valores dos coeficientes das harmônicas de cada potência, é possível calcular numericamente o coeficientes das harmônicas de . Em um programa no MATLAB calculou-se o valor em dB do coeficiente de cada harmônica de . Na Figura X é ilustrado o decaimento em dB das cem primeiras harmônicas geradas pela função aplicada a um tom de amplitude A. É observável que o decaimento da potência das harmônicas é maior para amplitudes de tons menores. Como só é constituído de harmônicos ímpares, a amplitude do tom só é elevado a potências ímpares, e no caso de A<0, o mesmo decaimento é observável para -1<A<0, pois o módulo dos coeficientes não mudam em relação a 1>A>0, apenas o seu sinal. Para um sinal formado por uma série de tons, a função não linear também gera harmônicos formados pelas combinações das frequências tonais e harmônicas do sinal.

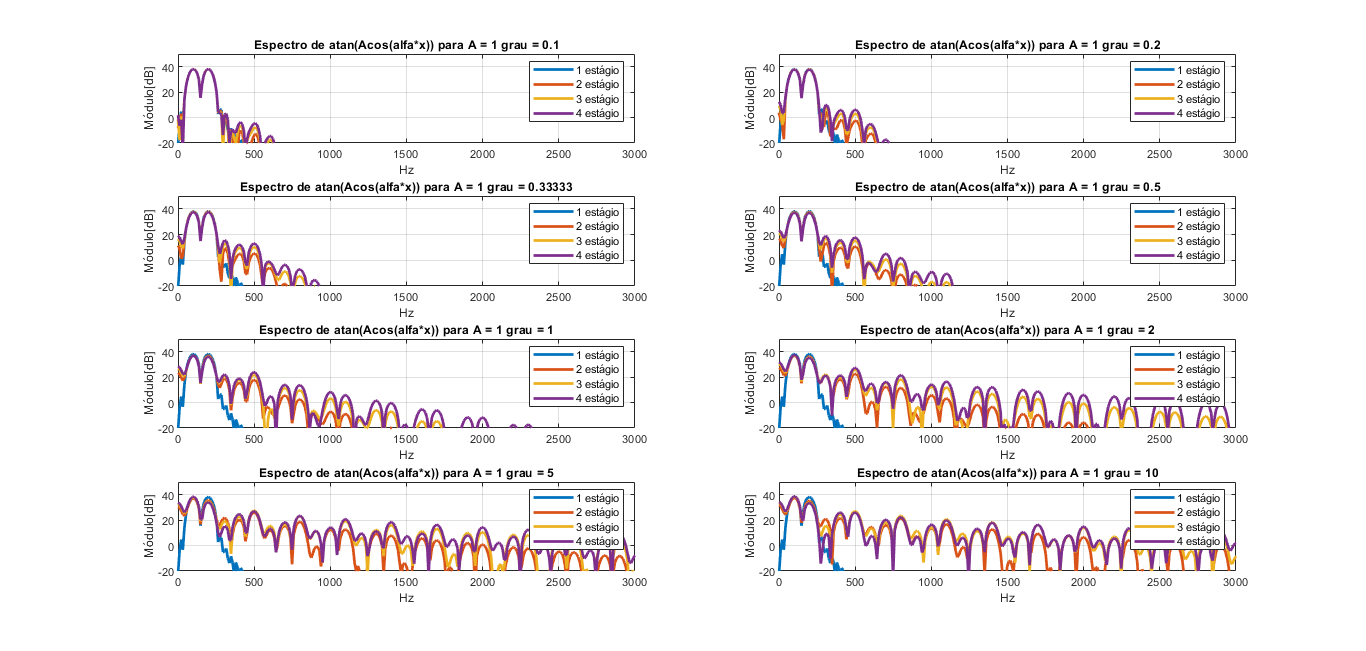


*Figura X: Decaimento em dB das primeiras 100 harmônicas para diferentes amplitudes A.*

Para verificar a análise feita, um tom de 100 Hz e amplitude A foi aplicada a uma cascata de um até quatro arcos tangentes para diversas amplitudes de tons e amplitude variável. Na Figura X encontra-se o módulo do espectro da saída da cascata em dB. Observe que harmônicas ímpares estão presentes. Além de graus de distorção maiores causarem harmônicas de amplitudes maiores, cascatas maiores também aumentam a potência das harmônicas devido a intermodulação da distorção. De forma semelhante, mas para dois tons simultâneos de 100 e 200 Hz de mesma amplitude levantamos a Figura X. Agora, além das harmônicas ímpares dos tons de 100 e 200 Hz, temos a combinação de suas harmônicas, por exemplo, 800 Hz.



*Figura X: Módulo do espectro de saída para 1 a 4 estágios de distorção para um tom de 100Hz.*

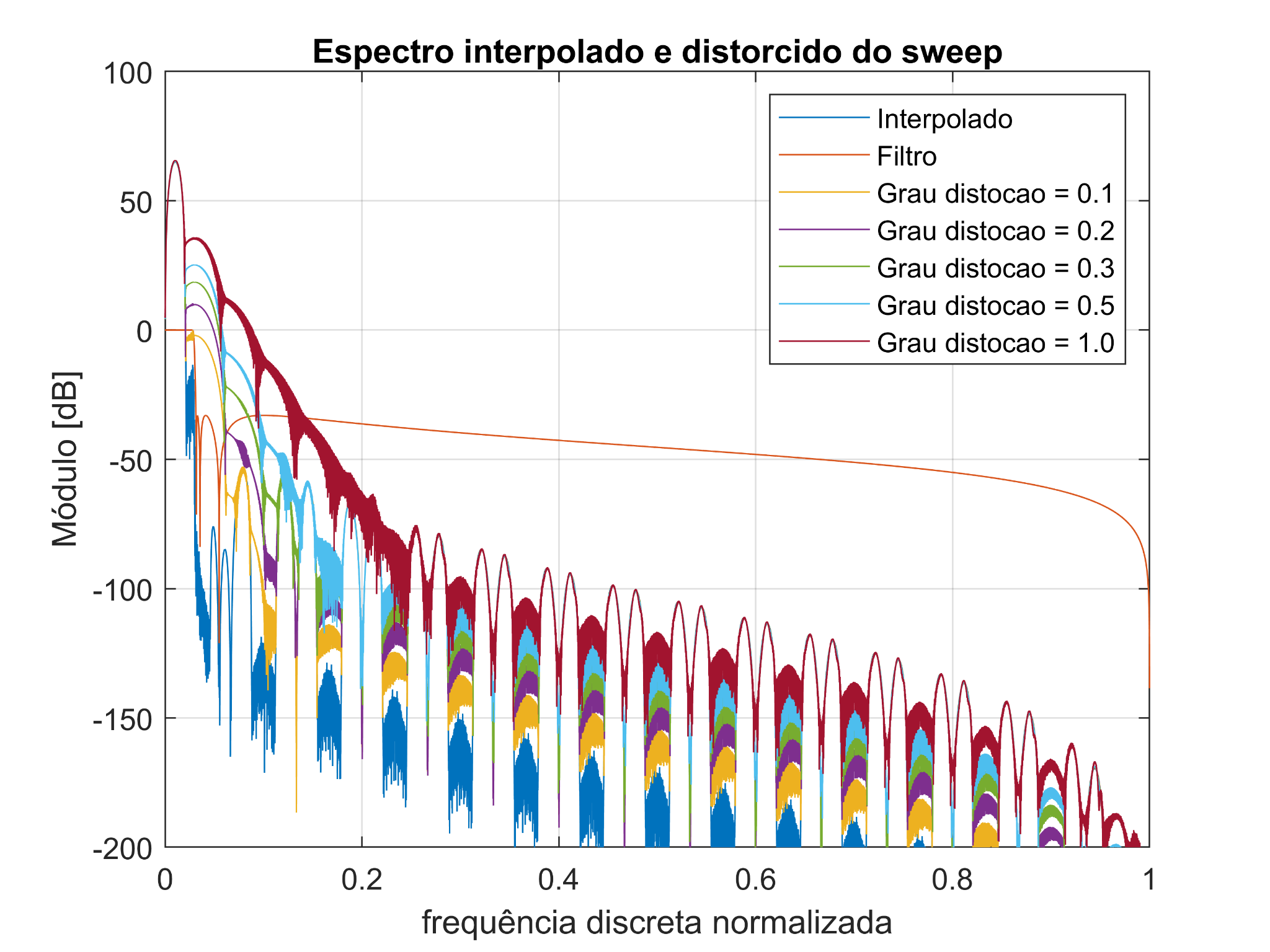


*Figura X: Módulo do espectro de saída para 1 a 4 estágios de distorção para um tom de 100Hz e 200Hz simultâneos.*

**4.2.3 Análise do aliasing de cada estágio**

Como explicado anteriormente, a distorção deve ser prosseguida de um filtro com intuito evitar que intermodulação de frequências geradas pela distorção além da banda do sinal causem mais aliasing. Em todo estágio, após a distorção, aliasing é introduzido no sinal em algum nível. Frequências rebatidas na faixa de corte do filtro serão eliminadas. Frequências rebatidas na faixa de passagem do filtro são inevitáveis de serem filtradas. No entanto, espera-se manter esse aliasing inevitável abaixo de 90 dB da frequência mais proeminente presente na banda do sinal, a fim de tornar esse termo desprezível em relação aos demais.

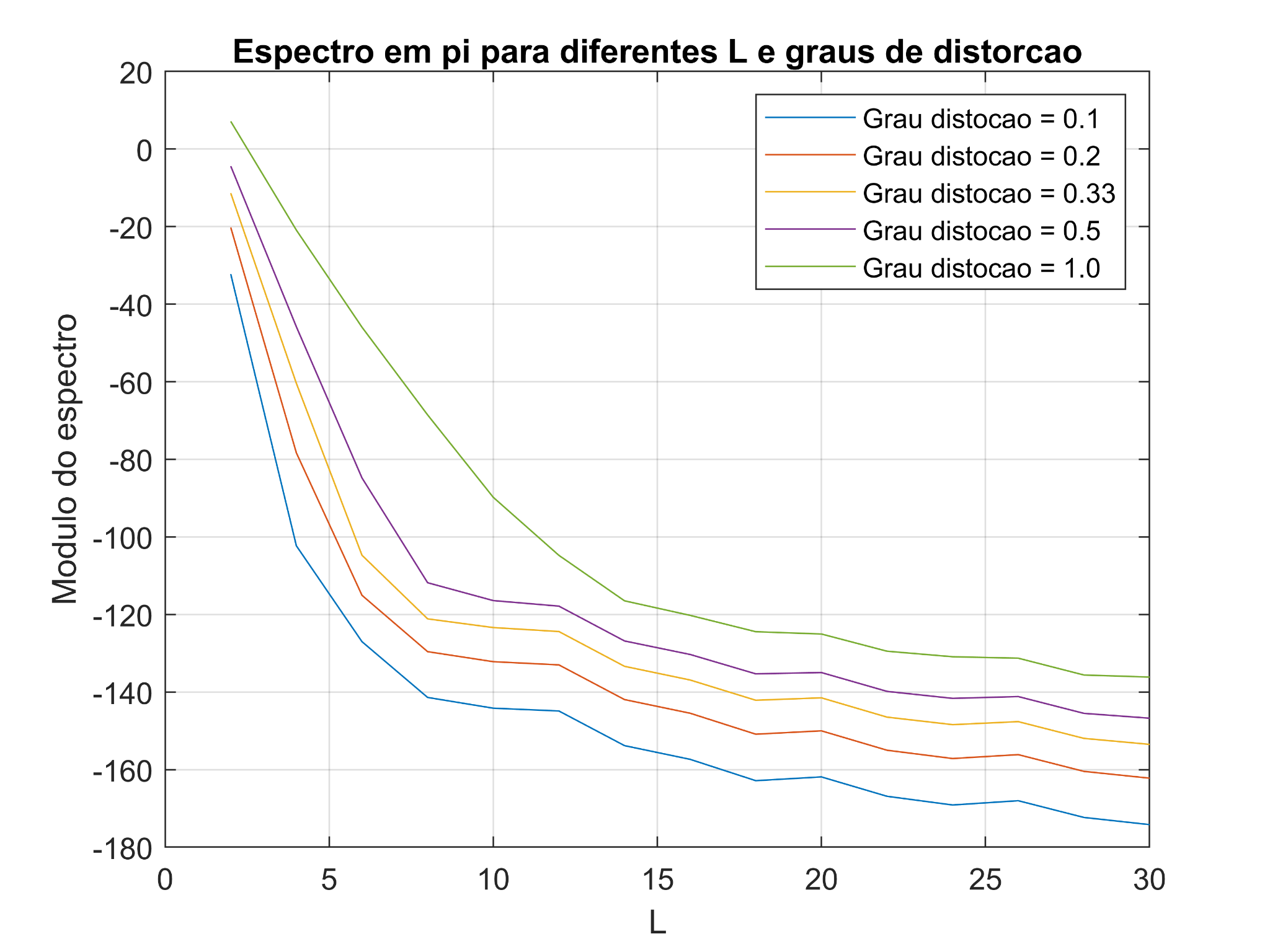
Um sinal sweep variando de 1 Hz até 15000 Hz em 100 ms considerando a taxa de amostragem como 48kHz foi gerado usando o software Audacity. O sinal de sweep foi considerado por ser um sinal “rico” em frequências, podendo conter frequências baixas (próximo de zero) até próximo da frequência máxima permitida pela taxa de amostragem em poucos segundos. Esse áudio foi interpolado, repetindo L vezes a amostra, com fatores de interpolação variando de 2 até 30 (L), usando uma cascata de três filtros passa baixa elíptico com ripple de 0.01 dB na faixa de passagem e 33 dB de atenuação na região de corte. O filtro foi dividido em uma cascata de três filtros nessa análise para que fosse possível criar filtros com faixas de transição cada vez menores com o aumento de L, sendo esta uma banda de Hz . A faixa de passagem do filtro é de 0 até Hz com o corte localizado na frequência de Nyquist do sistema original, ou seja, Hz. O sinal interpolado então foi aplicado à distorção com *⍺* variando entre 0.1 e 1. Na Figura X, é mostrado o módulo do espectro do sinal interpolado para *L=30* e diversos graus de distorção.



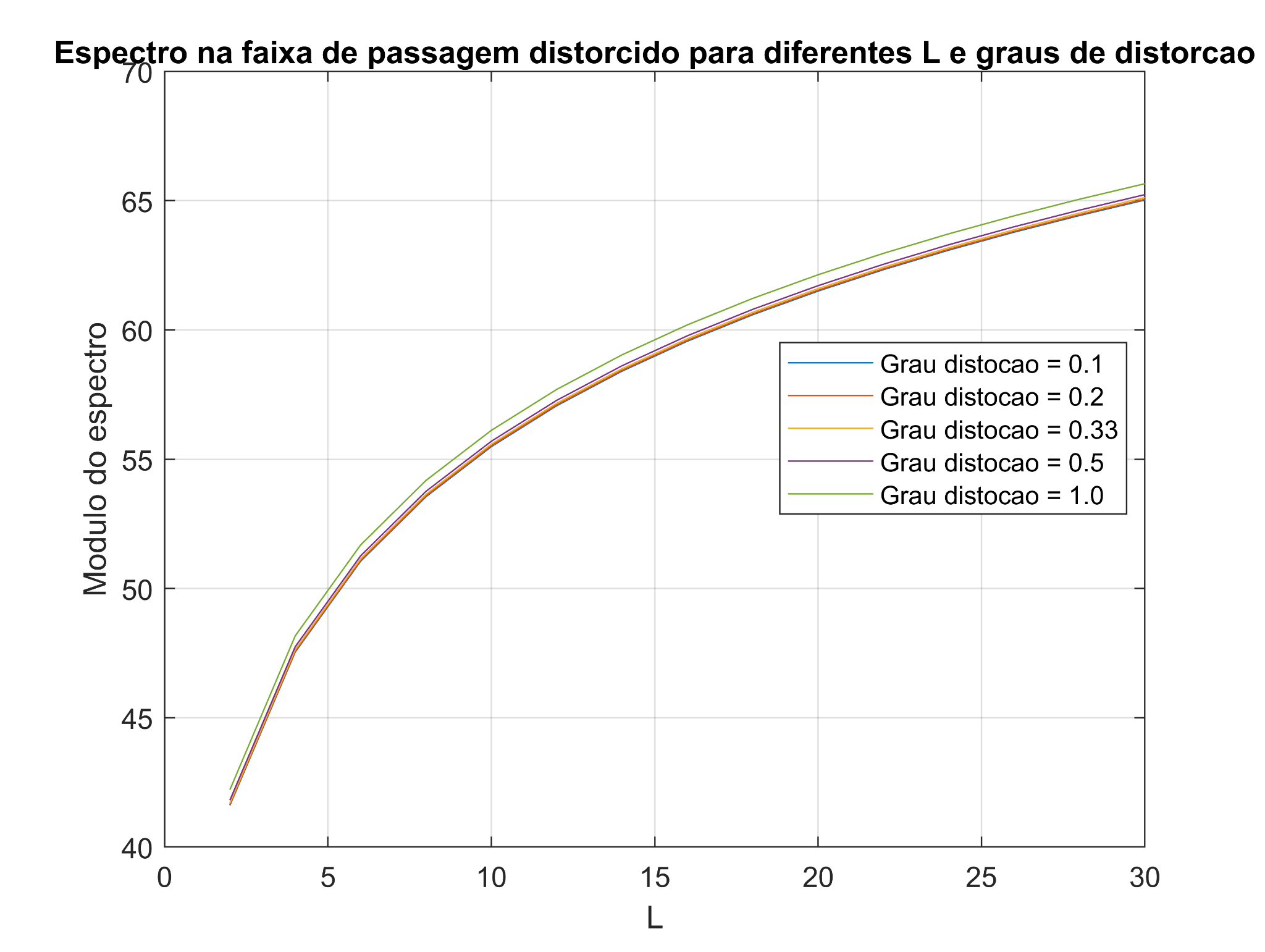
*Figura X: Espectro do sweep interpolado e distorcido*

A fim de estimar o nível de aliasing que está sendo introduzido na banda de passagem do sinal após uma distorção, procuramos estimar o máximo nível de sinal introduzido pela distorção próxima a frequência de Nyquist do sistema interpolado. Primeiramente, foi realizada diferença entre os espectros do sinal interpolado e o sinal interpolado e distorcido. Em segundo lugar, procurou-se o máximo módulo entre as frequências discretas e (Nyquist) no resultado dessa subtração. Essa estimativa permite que se tenha uma ideia do máximo nível de aliasing que está sendo introduzido pela distorção, pois supõe-se que a diferença entre o espectro com e sem distorção seja componentes introduzidos pelo processo de distorção. Como discutido anteriormente, espera-se a potência das harmônicas geradas pela função arco tangente sejam decrescentes, portanto frequências geradas além de devem ter potência menor do que a frequência mais potente gerada próximo a . Também tendo em posse do espectro distorcido, foi medido o módulo máximo do espectro na região de passagem do filtro. Na Figura X e Y estão apresentados a estimativa da frequência máxima introduzida próximo a frequência de Nyquist do sistema e o máximo na banda de passagem do módulo do espectro do sinal distorcido.

Os dois máximos analisados até então são tratados como apenas estimativas. Após a aplicação da distorção, frequências rebatidas podem estar presentes em qualquer parte do espectro. Como não é possível determinar de antemão o aliasing, a estimativa da frequência máxima criada próxima a contém influência de frequências rebatidas nessa região, o que acontece também em grau menor, devido a maior distância, da máxima frequência na região de passagem do sinal.



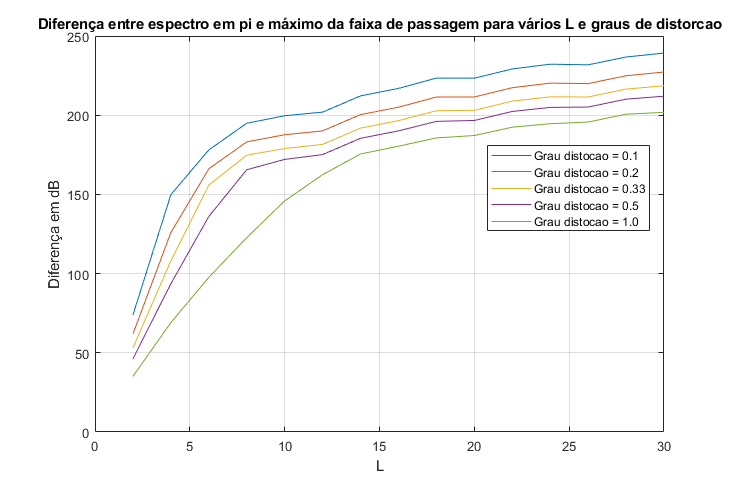
*Figura X: Espectro do sweep na frequência discreta π*



*Figura X: Máximo do módulo do espectro do sweep na faixa de passagem*

Fazendo a diferença entre os valores medidos, podemos estimar a diferença, em dB, entre o módulo da frequência mais significativa do sinal e aliasing introduzido na banda de passagem no sinal. Na Figura X, é apresentada a estimativa do aliasing gerado por uma distorção na banda de passagem para diversos fatores de interpolação e graus de distorção. Tendo em vista o objetivo de manter o aliasing nessa faixa pelo menos 90 dB da frequência mais proeminente, pelo gráfico concluímos que precisamos de pelo menos fatores de interpolação maiores do que 6 para um grau de distorção unitário para atingir essa meta em um estágio. Essa análise se restringe apenas a um estágio, no entanto como desejamos usar L consideravelmente maiores do que 6, por exemplo 30, e graus de distorção menores do que o unitário, mas em um número de estágios elevado, consideramos que a análise pode ser estendida para o processamento todo, desde que o filtro logo após a distorção atenue suficientemente as frequências na sua faixa de corte.

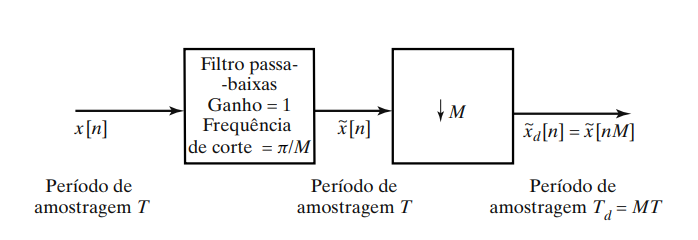
A fim de se obter uma estimativa mínima do corte do filtro após a distorção, tomamos como base a Figura X para um . O módulo do espectro do sinal interpolado distorcido para frequências discretas além de 0.1 em comparação com o módulo do espectro do sinal apenas interpolado para é de pelo menos 20 dB. Portanto, atenuações maiores do que essas devem ser usadas. A frequência discreta 0.1 foi considerada pois o filtro após distorção pode não ser tão restritivo em relação a sua faixa de transição, podendo deixar passar frequências além da frequência de Nyquist do sistema não interpolado. Isso se deve ao fato de que o filtro de decimação tem por si só a função de eliminar tais frequências. Frequências próximas a passagem também são tão críticas para o aliasing em relação às frequências presentes na passagem, devido a banda alta existente devido a interpolação até a frequência de Nyquist do sistema interpolado.



*Figura X: Diferença do módulo do espectro máximo na faixa de passagem e seu valor em π*

**4.3 Decimação**

Por fim, para conservar as frequências originais do sinal de áudio na entrada do Jetson, devemos decimá-lo até a taxa original. Isso porque o sinal que será enviado ao dispositivo de saída de áudio será reconstruído na conversão digital-analógica assumindo a mesma taxa de amostragem da entrada, portanto o espectro deve ser reduzido por um fator L também. O processo de decimação por um fator M, nesse caso M = L, envolve uma pré filtragem por um filtro passa baixas com corte em para evitar aliasing no processo, já que toda frequência após esse valor será rebatida para a região 0 a . Um diagrama de blocos de um processo de decimação se encontra na Figura 14.

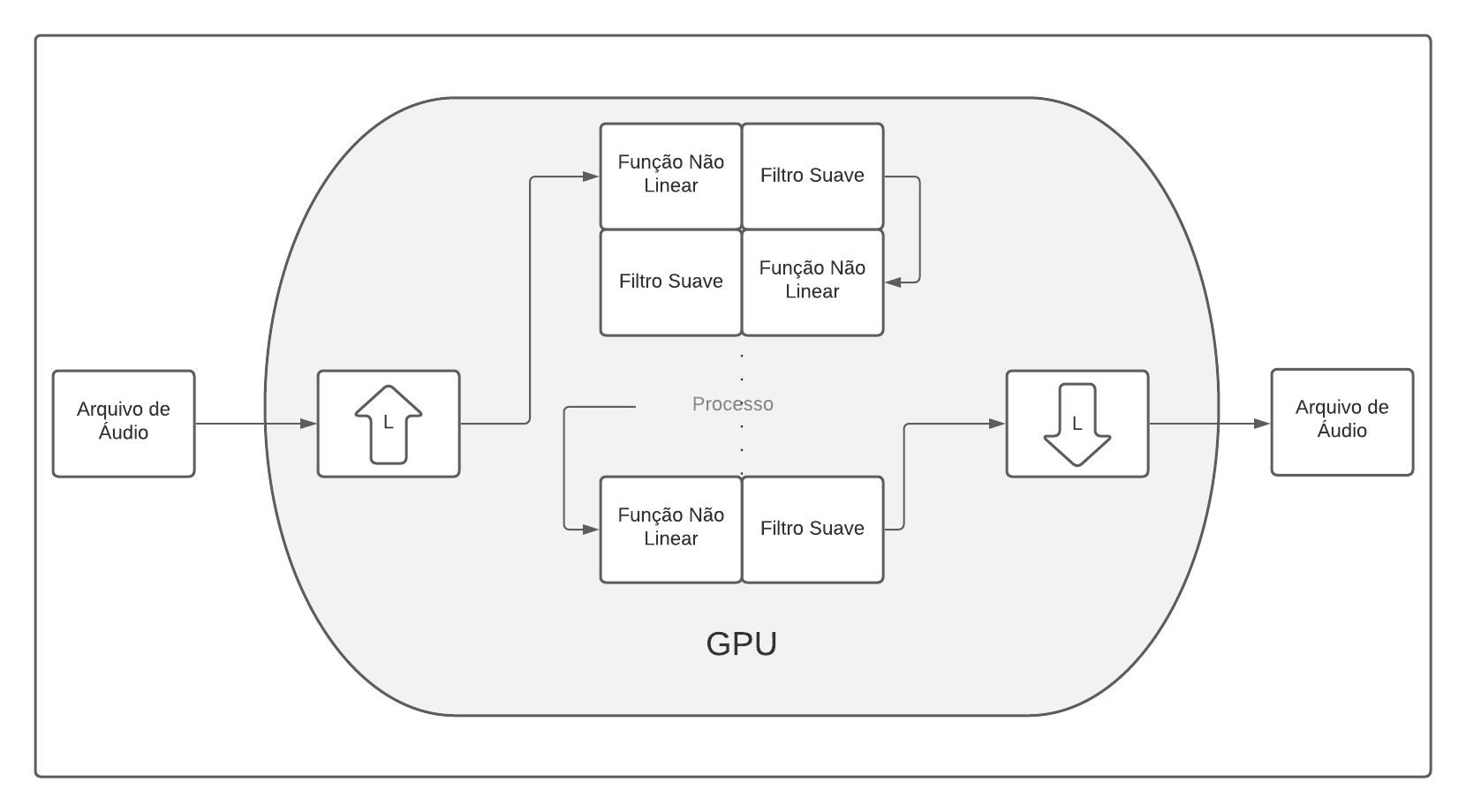


*Figura 14: Diagrama de blocos da decimação* (OPPENHEIM e SCHAFER, 2012)*.*

Durante o processo de filtragem pelo passa-baixas o sinal pode obter valores maiores que o unitário, o que geraria inconsistências na conversão para um sinal de áudio, já que, como discutido anteriormente, só permite valores reais entre -1 e 1. Por esse motivo, é utilizado comando SFC\_SET\_CLIPPING da biblioteca sndfile, que faz a *clipping* do sinal (trunca valores maiores que 1 e -1 para esses limites) na escrita do arquivo de áudio. Apesar desse processo introduzir mais distorção no processo, aumentando o nível de aliasing no sinal final, ainda sim é muito conveniente.

**4.4 Esquemático do processo**

Considerando o que foi apresentado até agora, Interpolação, Distorção e Decimação pode-se montar um esquemático do processamento que o sinal de áudio deve passar. Uma amostra por vez será enviada a GPU, será interpolada considerando a repetição de seu valor e passando por um filtro IIR passa baixas, as L amostras da interpolação passarão por uma sequência de distorções e filtragens também por um filtro IIR passa baixas e por fim as L amostras serão decimadas usando um filtro IIR passa baixas, resultando em uma amostra na saída do sistema.

****

*Figura X: Esquemático do processo completo de distorção.*

**4.5 Pipeline**

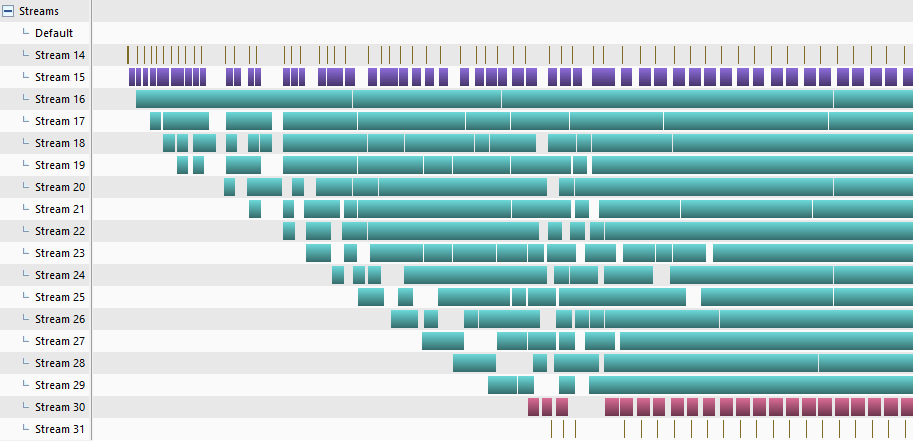
É possível notar que as operações descritas até então possuem bastante dependência entre si. É natural perceber que a interpolação só pode começar após a amostra ter sido recebida, que os estágios de distorção dependem sempre do estágio anterior ter sido concluído e que a amostra a ser decimada depende de todo processo de distorção ter se encerrado. Com essa visão em mente, o aproveitamento do paralelismo em nível de dados fica inviável. Entretanto, como o número de operações que desejamos fazer é muito grande, podemos dividi-las e obter um paralelismo em nível de tarefas, o pipeline.

Se dividirmos o processamento todo das amostras entre diferentes Streams, com cada uma realizando uma parte menor das operações, conseguimos com que cada Stream necessite de uma capacidade computacional menor, como visto na Figura X anteriormente (Figura das streams metodologia), diminuindo seu tempo de processamento.

A maneira que encontramos de fazer essa divisão foi separar uma Stream para fazer a interpolação e alguns estágios de distorção, uma Stream que faz alguns estágios de distorção e uma decimação, e outras Streams que fazem apenas estágios de distorção. Desenvolvemos o programa para que o número de estágios de distorção realizados em cada uma dessas Streams seja configurável, para que pudéssemos encontrar a melhor combinação.

O processo então consiste na chegada da amostra na Stream de interpolação (que faz a interpolação e certo número de estágios de distorção), que gera L amostras de saída que serão distorcidas pelas Streams subsequentes de distorção. Cada Stream de distorção alimenta suas amostras de resultado para a próxima Stream de distorção, seguindo assim por certo número de estágios. Finalmente, a última Stream de distorção alimenta suas saídas para a Stream de decimação, que gera uma amostra de saída. Quebrar a distorção em várias streams menores é vantajoso, pois o processamento é feito a partir de uma sequência de distorções leves seguidas de filtros passa-baixas, dessa forma encurtar o tamanho de cada kernel de distorção para um número menor de operações garante um fluxo contínuo de amostras na pipeline.

Notamos, portanto, que a leitura das amostras de entrada depende apenas de a Stream de interpolação ter acabado. Entretanto, as operações de interpolação são dependentes entre si, pois são realizadas com filtros IIR, que dependem das amostras anteriores processadas, para que seus vetores de condições iniciais sejam atualizados, impossibilitando realizar mais de uma interpolação ao mesmo tempo. As demais Streams tem um comportamento semelhante: as de distorção dependem apenas das Streams de distorção anteriores e a de decimação só depende da última distorção. Isso possibilita que o processamento de diferentes amostras seja realizado ao mesmo tempo na GPU, causando teoricamente apenas um delay no sistema, que se for limitado a valores pequenos pode ser imperceptível.



*Figura X: Paralelismo 16 streams e 100 distorções por stream.*

Na Figura X obtida no Visual Profiler (Referencia Prof) é possível observar a metodologia de pipeline empregada, em que a stream 14 representa o lançamento do kernel da cópia de memória do Host para o Device, a stream 15 representa a interpolação, as streams de 16 a 29 os kernels de distorção, stream 30 a decimação e 31 a cópia Device para o Host. Para garantir que os kernels sejam lançados na ordem correta utiliza-se o cudaEventQuery, para verificar se o evento da stream que o kernel depende já foi concluído.

Nota-se que esse processo permite aumentar a velocidade com que se obtêm as amostras da entrada, quando comparado com realizar o processamento completo de cada amostra antes de obter a próxima. Ele também permite que um grande número de streams processem dados simultaneamente como visto na Figura X.

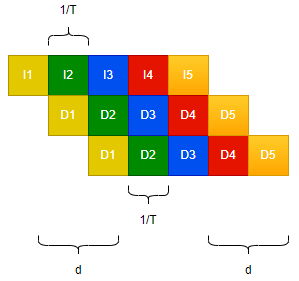
**4.6 Processamento em tempo real**

Estabelecido a estrutura do projeto, a viabilidade do processamento em tempo real depende de 2 pontos fundamentais: o período entre as cópias de memória do Host para o Device das amostras do sinal, e o tempo entre esta cópia e sua respectiva cópia de volta para o Host após o seu processamento.

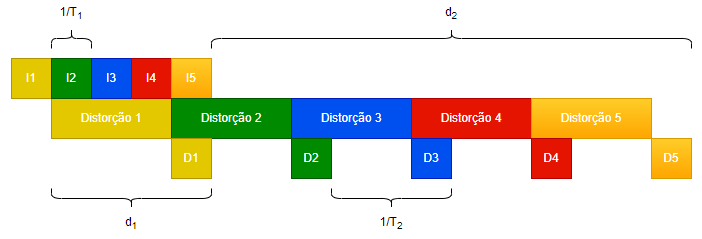
No primeiro caso, a cópia para a GPU precisa acontecer na mesma taxa de amostragem do sinal, ou seja 48kHz (ou superior), de forma que, se a pipeline ocorrer de forma correta (não tenha atrasos entre os kernels), o sinal processado seja escrito no arquivo de saída com a mesma taxa de amostragem, mas com certo atraso entre a entrada e saída.

O segundo caso precisa garantir que o tempo de processamento de cada amostra dure no máximo 30ms para que o atraso entre a entrada e saída não seja perceptível para o usuário. Para isso, é preciso tomar um número de distorções máximo, que faça com que os kernels de distorção não sejam longos demais que ultrapassem o tempo permitido.

Também foi observado que os kernels precisam ter durações próximas, para evitar fenômenos indesejados. Um deles é que quando um kernel é muito mais lento, ele pode limitar a taxa de escrita na saída de áudio. O outro é que kernels que sejam muito mais rápidos que outros conseguem ser executados várias vezes antes que o kernel mais lento seja executado, gerando um acúmulo de tarefas para o próximo kernel, que não consegue acompanhar a velocidade de processamento. Isso gera uma fila de processos, que acaba limitando a quantidade de operações que podem estar sendo executadas na GPU, perdendo-se o pipeline. Nas Figuras X e Y abaixo é possível verificar a diferença entre o caso ideal, em que todas os kernels têm o mesmo tamanho (3 Streams, uma interpolação I e duas distorções D), com o caso em que há um kernel muito maior, aumentando o delay de saída (d1 em comparação a d5) conforme as amostras processadas aumentam, como também diminuindo a taxa de amostras na saída em relação a taxa da entrada (1/T2 em comparação a 1/T1).



*Figura X: Pipeline com streams ideal.*



*Figura X: Pipeline com streams caso não ideal.*

**4.7 Métricas do projeto**

Dessa forma definimos as métricas utilizadas neste projeto:

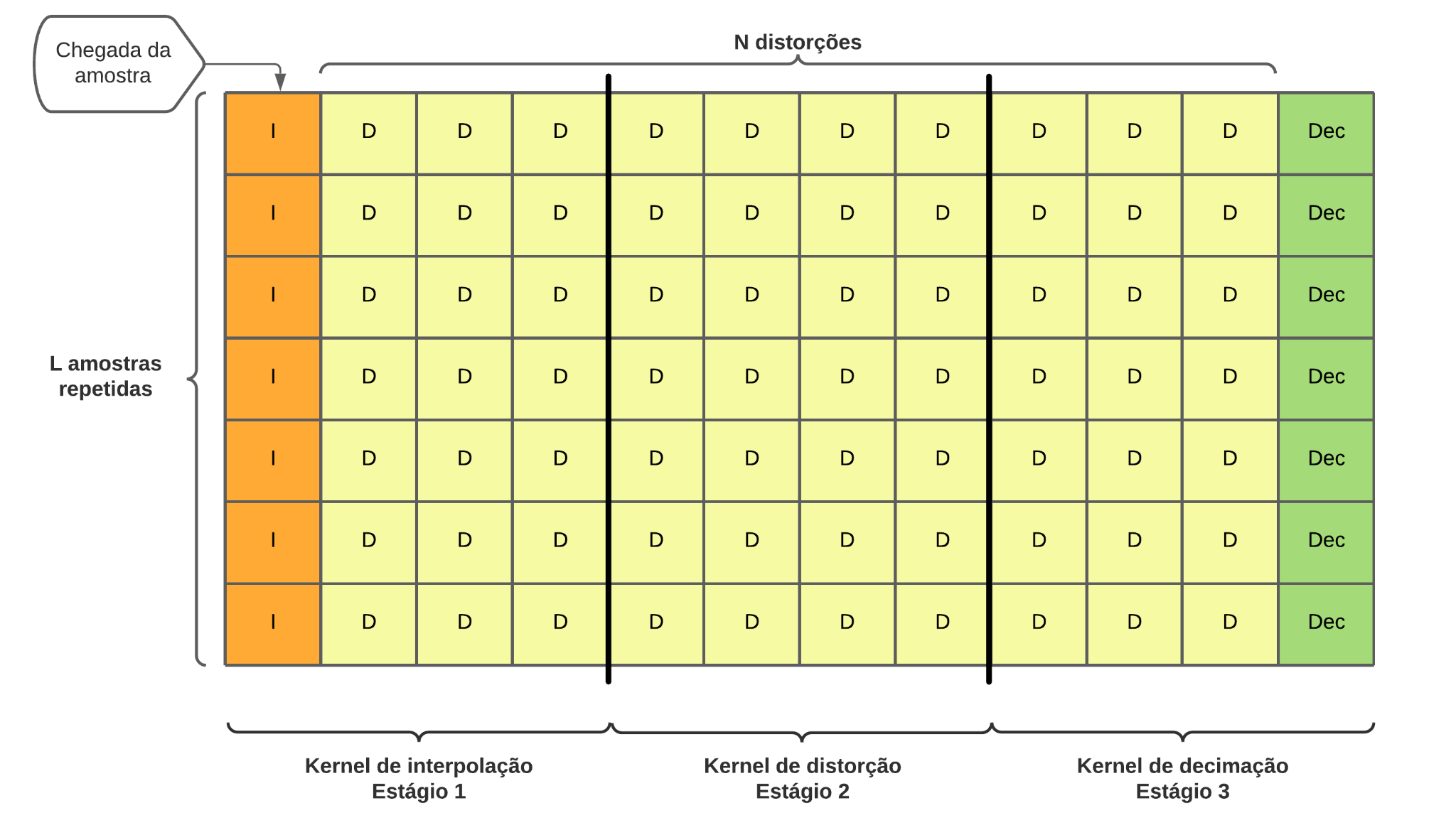
* Nível de aliasing: Percepção qualitativa do aliasing presente no sinal após o processamento.
* Delay de processamento: Tempo entre a cópia da amostra para a GPU e sua cópia de volta a CPU após processamento.
* Tempo de processamento: Tempo total de processamento de um sinal de áudio.

O nível de aliasing é testado a partir de um sinal de sweep de 0 a 15kHz processado utilizando a sobreamostragem e as filtragens entre distorções, comparando com o mesmo sinal aplicando somente as distorções. O delay de processamento é obtido pelas medições de tempo no programa de distorção final e comparando os tempos entre a GPU e CPU, da mesma forma é medido o tempo de processamento total.

# **5 Programas**

**5.1 Estrutura Geral do Processamento**

Devido ao procedimento de interpolação, cada amostra do sinal de áudio original gera a necessidade de processamento de L amostras, sendo L o fator de interpolação. Cada uma dessas amostras precisa passar pelo filtro de interpolação (passa-baixas), por todas operações de distorção e pelo filtro de decimação (também um passa-baixas). Portanto, podemos representar o processamento de L amostras conforme a figura X. Nela, cada linha representa uma amostra que precisamos processar, e as colunas representam as operações necessárias nessas amostras. Nota que as amostras acrescentadas ao sinal original são repetições da amostra anterior, como descrito na seção 4.1.

*Figura X: Processamento L amostras.*

Observando o processamento da primeira amostra (primeira linha da Figura X), podemos notar que cada operação depende apenas da anterior da linha ter acabado. As operações das demais amostras (linhas), entretanto, dependem tanto da operação anterior da linha, quanto da mesma operação da amostra (linha) anterior ter acabado, devido aos vetores que armazenam as condições iniciais dessas operações precisarem ser atualizados a cada amostra.

Isso levou à decisão de dividir o processamento em diferentes blocos, que serão executados por diferentes kernels na GPU, que foram denominados “estágios”. Cada um desses estágios executa certo número de operações necessárias para o processamento das amostras e fornece sua saída para o próximo estágio. Nomeamos o kernel que executa a filtragem da interpolação de “kernel de interpolação”, os kernels que realizam apenas distorções de “kernel de distorção” e o que realiza a filtragem de decimação de “kernel de decimação”.

Com essa divisão, desenvolvemos o programa com a possibilidade de controlar o número de operações que cada kernel faz com as seguintes variáveis:

* N\_distorcoes\_case0: número de operações de distorção realizadas pelo kernel de interpolação
* N\_distorcoes\_case1: número de operações de distorção realizadas pelo kernel de decimação
* N\_distorcoes\_case2: número de operações de distorção realizadas pelo kernel de distorção

Além de calcular o número de estágios que queremos utilizar com a variável N\_estágios. Com isso, o número total de operações de distorção é calculado como:

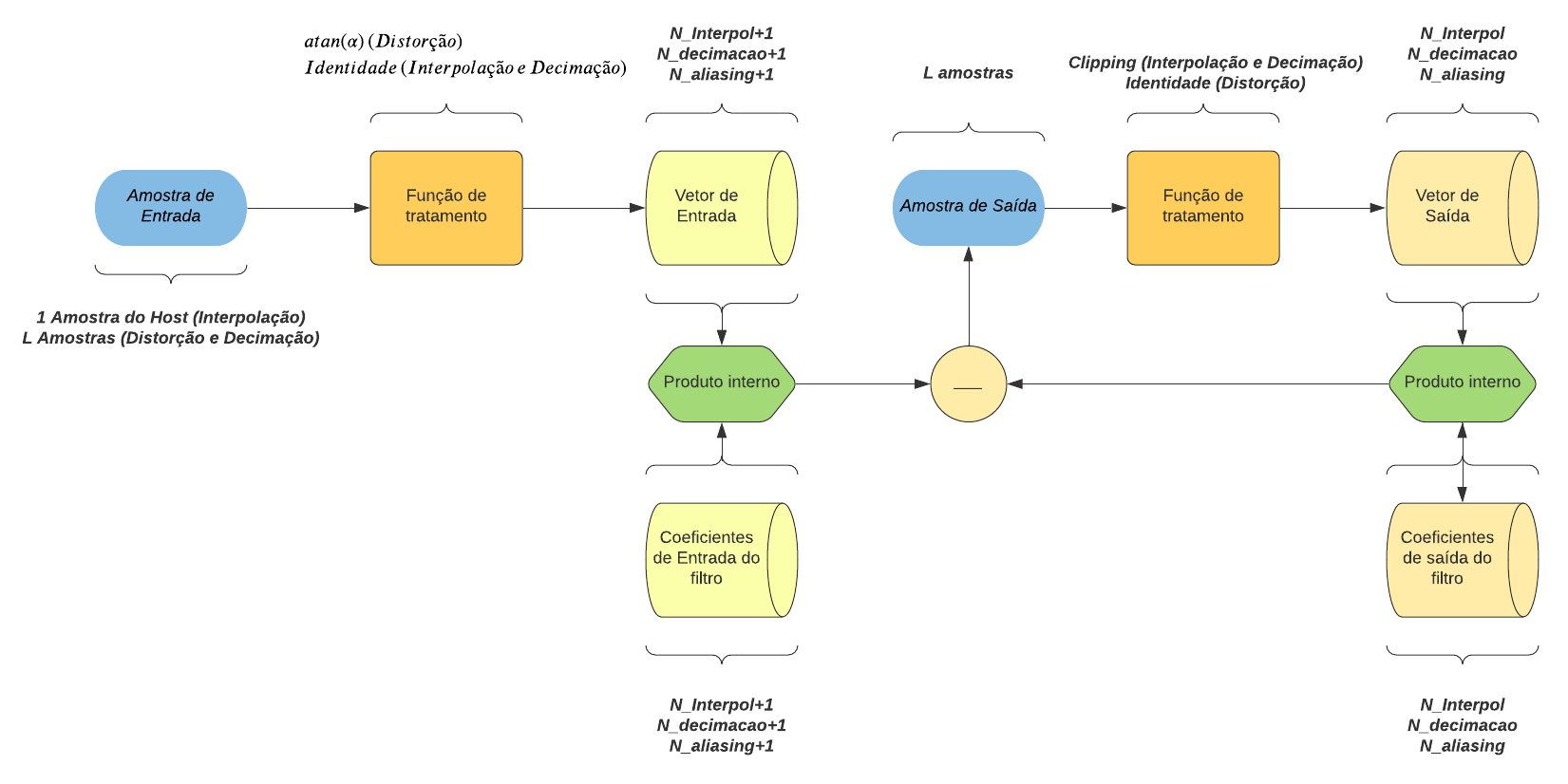
Devido às observações apontadas na seção 4.6 quanto ao tamanho dos kernels, é importante que eles possuam aproximadamente o mesmo tempo de execução. Com isso em mente, considerando que as operações que precisam ser realizadas são de complexidade parecida - filtragem IIR, decidimos realizar o mesmo número de distorções em cada um dos kernels.

**5.2 Estrutura básica da filtragem**

Todos os kernels possuem filtragem por filtros IIR. A diferença entre o processamento da interpolação, a distorção filtrada e a decimação repousa no tratamento da amostra de entrada e saída, além de basearem-se em coeficientes de filtros diferentes. Na Figura X está mostrada a estrutura da filtragem IIR, enfatizando as diferenças no tratamento da amostra de entrada e saída pela interpolação, distorção filtrada e decimação. Para simplicidade de notação, a ordem do filtro de interpolação foi denominada pela variável N\_interpol, a do filtro depois da distorção de N\_aliasing e do filtro de decimação de N\_decimacao. Os vetores de entrada e saída consistem nas condições iniciais do filtro no instante considerado, ou seja, as amostras de entrada e saída passadas ao instante considerado. Eles possuem dimensões indicadas na figura.

A estrutura da filtragem IIR começa com o recebimento da amostra a ser filtrada. No caso da interpolação apenas uma amostra é recebida, enquanto na distorção filtrada e decimação são recebidas L amostras. Na interpolação a amostra recebida é usada como entrada L vezes, para começar o processo de aumento de taxa de amostragem. Cada amostra recebida na distorção também é passada pela função não linear. O vetor de entradas passadas é deslocado, tendo todas suas posições atualizadas e então a sua posição referente a amostra mais nova deslocada para que a nova amostra ocupe seu lugar. É realizado então o produto interno do vetor de entradas com os coeficientes do filtro de entrada, e também o do do vetor de saídas passadas do filtro com os coeficientes de saída do filtro. A diferença entre os produtos internos consiste na saída do processo de filtragem. Esse procedimento tem objetivo de implementar a função característica do filtro IIR:

A nova saída na interpolação e decimação tem seu valor conferido para que o módulo da amostra não ultrapasse o limite da saída do sistema entre 1 e -1, fazendo o clipping da amostra. Na interpolação, o clipping é implementado usando condicionais (diretivas *if* )*,* enquanto na decimação o processo é feito pela biblioteca *sndfile* usando o comando SFC\_SET\_CLIPPING. O vetor de saída então é deslocado da mesma forma que o de entrada, recebendo a saída mais atual para ocupar a posição deslocada do vetor. O processo se repete L vezes, cada vez com uma nova amostra de entrada.



*Figura X: Estrutura filtragem IIR*

As condições dos filtros, os vetores de entrada e saída, precisam ter seus valores atualizados antes de receberem as próximas L amostras. Esse fato implica que não podem haver várias interpolações ou mesmos estágios de distorção ou várias decimações ocorrendo concomitantemente por Streams diferentes.

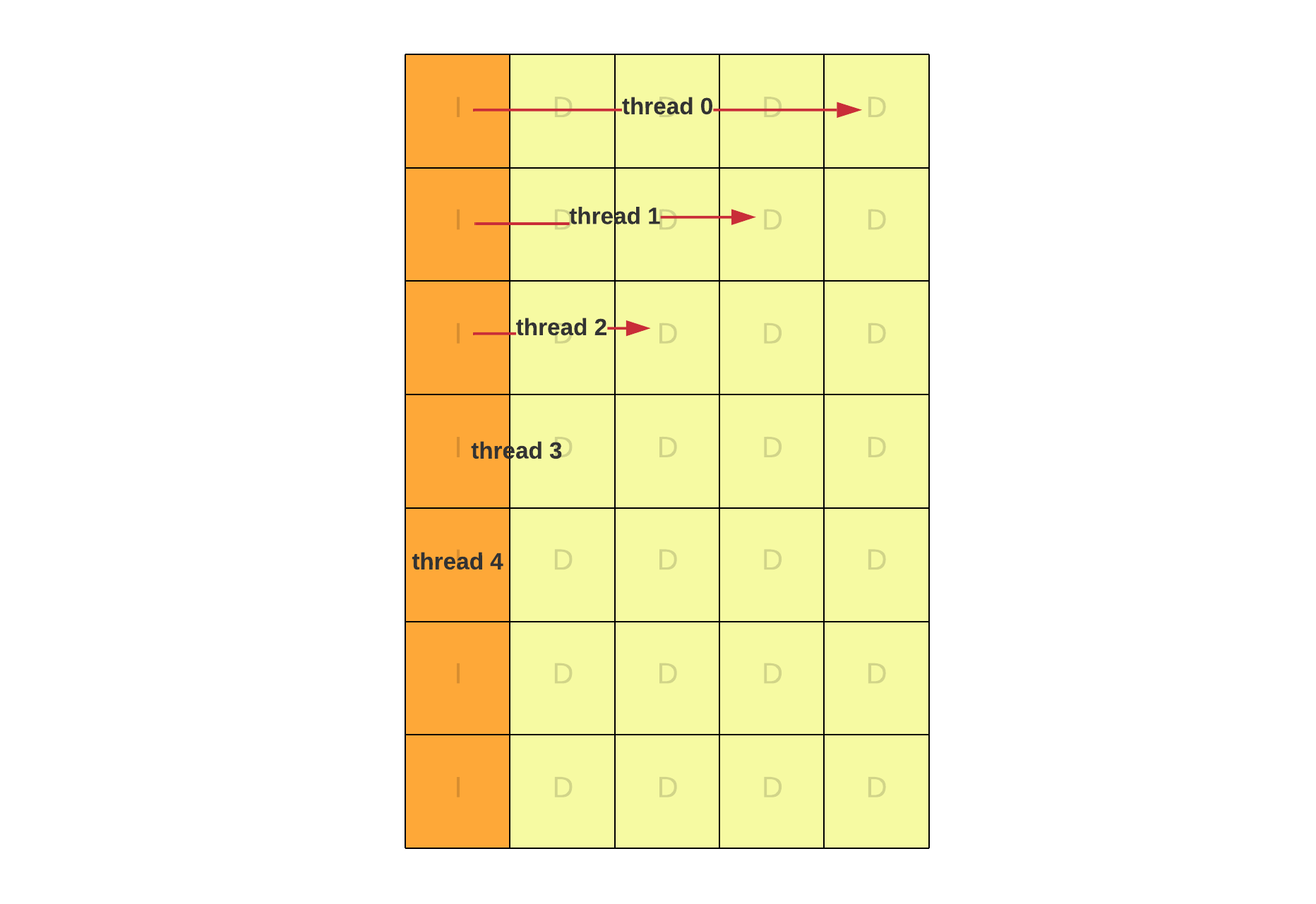
**5.3 Paralelismo interno do kernel**

Como dito anteriormente, criamos três estruturas diferentes de kernels. O kernel de interpolação consiste de N\_filtragens\_interpol filtragens pelo filtro de interpolação e N\_operacoes\_case0 distorções com filtragem, o de distorção possui N\_operacoes\_case2 distorções com filtragem, e o de decimação possui N\_operacoes\_case1 distorções com filtragem e N\_filtragens\_decimacao filtragem pelo filtro de decimação. Em qualquer um desses casos, uma filtragem ou mais serão realizadas para garantir a eliminação de artefatos indesejados.

Portanto, se o kernel realizar N operações de filtragem, em L amostras, NL operações de filtragens devem ser realizadas. No caso de processamento sequencial, teríamos um delay do kernel de NL vezes o tempo de uma filtragem. No entanto, nos três kernels, L amostras podem estar sendo processadas simultaneamente, por threads diferentes. Quando uma amostra acaba seu processamento em uma thread, a próxima amostra pode começar seu processamento em outra thread.

Para implementar o paralelismo dentro do kernel, então, cada thread foi atribuída a processar uma amostra, sendo necessário, portanto, L threads. O kernel possui um contador para cada thread, indicando qual operação do kernel cada thread está processando na iteração. O contador é inicializado de forma que apenas a primeira thread, que processa a primeira amostra comece o processamento na primeira iteração. Na segunda iteração, a primeira thread começa o segundo estágio enquanto a segunda começa o primeiro. A cada iteração as threads são sincronizadas para que todas realizem apenas um estágio por iteração. O loop termina então quando a última thread realiza o último estágio.

Dessa forma de processamento, o processo pode ser realizado mais rapidamente do que em NL vezes o tempo de execução de uma filtragem. A última thread deve esperar apenas L operações para começar o seu processamento, demorando N operações para terminar, resultando em L+N operações a serem concluídas em frente aos NL operações no caso em série de uma CPU.

****

*Figura X:*

**5.4 Paralelismo entre Streams**

Para garantir o paralelismo entre Streams conforme descrito na seção 4.5 utilizou-se os eventos do CUDA. Cada estágio ao acabar de ser executado ativa um evento. O Host pode através da função cudaEventSynchronize adicionar execução de Streams na fila de execução da GPU quando o evento tiver acontecido. Inicialmente, era de posse desse fato que tentamos desenvolver o sincronismo entre estágios de execução. No entanto, mesmo que a execução fosse em ordem correta, o hardware demonstrou incapacidade de adicionar mais do que cinco kernels em uma fila, travando a execução do programa quando apenas cinco estágios estivessem ativos. Para conseguir mais estágios ativos ao mesmo tempo, mudamos a função de sincronismo para cudaEventQuery. Dessa forma, ao invés de criar uma fila de execução na GPU dependente de eventos, decidimos usar condicionais no host para apenas demandar a execução de um estágio quando um evento fosse alcançado, possibilitando que mais Streams ocorressem concomitantemente.

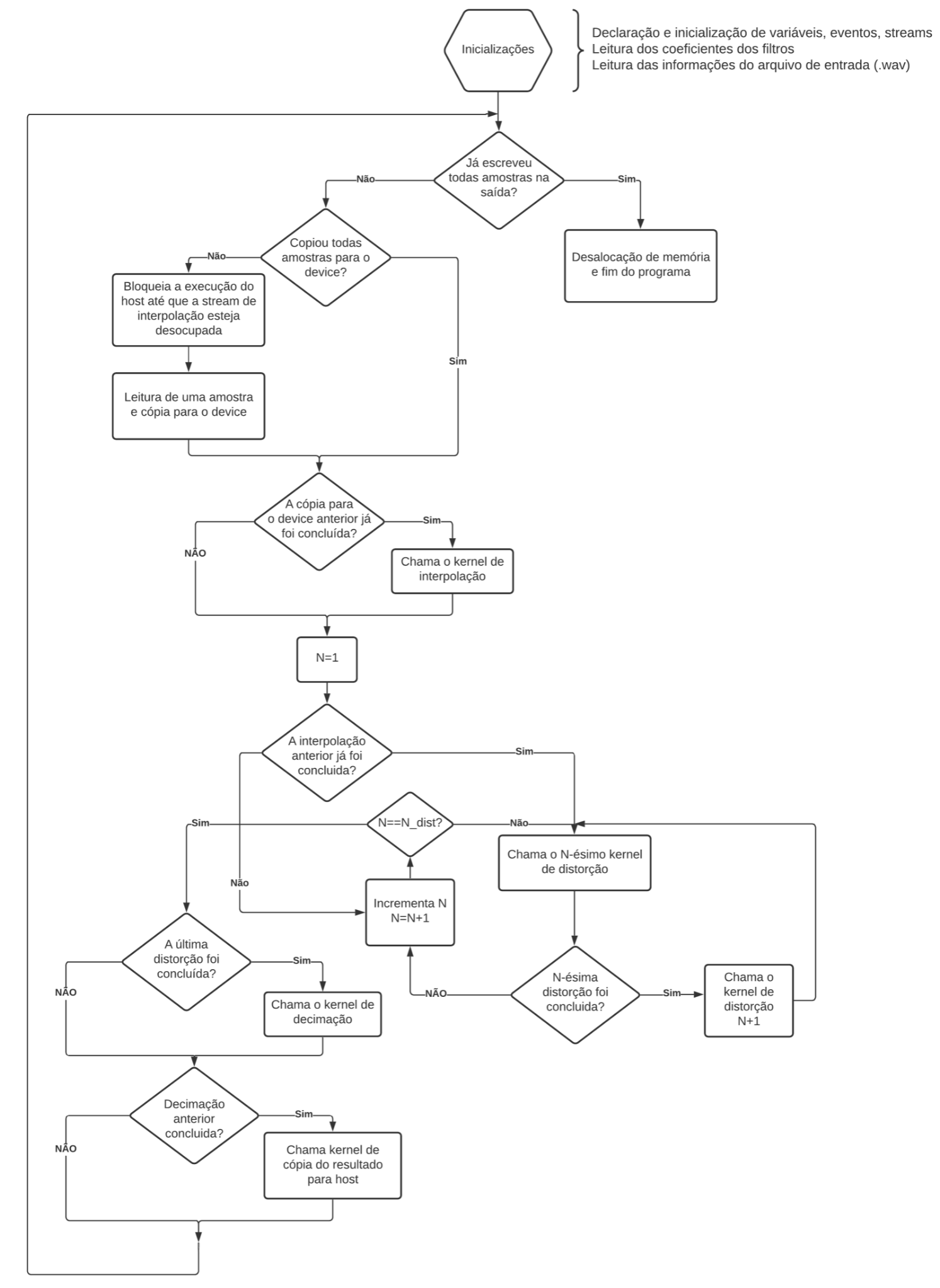
Com o uso da nova função de sincronismo cudaEventQuery, uma nova variável de sincronismo no host também precisou ser criada, event\_ok. Sem esta nova variável, o host poderia lançar kernels antes da hora correta, pois eventos em Cuda tem como seu estado default o estado de sucesso, sendo que seu valor apenas muda para “executando” quando o host pede essa execução à GPU, assim variáveis booleanas foram necessárias para indicar que o host já passou por certa região do código.

Como resultado, conseguimos com que diversas streams mantivessem suas execuções em um mesmo momento, de forma sincronizada. Aproveitando melhor as capacidades computacionais da GPU. Apesar de ter sido possível executar mais do que 100 streams diferentes, o hardware em si da GPU apenas consegue manter 16 Streams ativas ao mesmo tempo. A Figura X do tópico 4.5 da metodologia mostra um exemplo de timeline da execução do programa feito, com 16 Streams ativas na maioria do tempo.

**5.5 Programa principal**

Com base nas ideias apresentadas nas seções 5.1 a 5.4, foi desenvolvido um programa que segue o fluxograma da figura X. Note que ele é composto por dois laços principais: um responsável por controlar o número de amostras lidas e outro responsável por chamar o número correto de kernels de distorção.

O tamanho do arquivo lido é obtido através de funções da libsndfile, e o resultado do processamento é escrito em um novo arquivo de áudio de saída.



**5.6 Programa Auxiliares**

**5.6.1 Programas CPU**

Para ter bases de comparação com o resultado do nosso projeto, foram desenvolvidos dois programas que rodam na CPU. Um deles implementa o mesmo processo que fazemos na GPU, só que sem os paralelismos e o segundo implementa o processo direto de aplicação de distorções, sem fazer interpolação ou filtragem.

A partir do programa que não possui paralelismos, podemos ter uma referência de qual o aumento de velocidade de processamento estamos obtendo com a utilização da GPU, além de poder conferir os resultados obtidos ao ouvir os arquivos de áudio resultantes do processamento.

Já a partir do programa que não possui o processo de interpolação com distorções suaves, podemos observar a vantagem que o processo do programa principal gera no resultado final, produzindo uma distorção com menos aliasing e, portanto, de melhor qualidade.

**5.6.2 Programas de Estatísticas**

Foram desenvolvidos programas que, em essência, funcionam da mesma forma que os programas citados anteriormente, porém possuem iteradores para que diferentes configurações sejam executadas mais de uma vez, para podermos levantar dados e gerar estatísticas sobre eles. Os parâmetros que achamos relevantes para nosso processo foram o delay máximo do sistema e o tempo total de execução. Com os quais podemos realizar operações de média para diminuir eventuais flutuações nesses valores.

Medimos o delay máximo do sistema como o maior tempo entre a leitura de uma amostra e a eventual escrita de sua amostra correspondente na saída. Já o tempo de processamento foi considerado como o tempo que os programas levam para acabar de processar um arquivo de áudio completo. Para parametrizar os testes, escolhemos um arquivo de áudio de mesmo tamanho para ser executado em todos os programas e utilizando o mesmo número total de operações de distorção .

No caso do programa da GPU, há diversas combinações que podemos fazer de número de estágios e número de operações por estágio para atingir o número de operações de distorção final. Isso faz com que o programa de testes da GPU leve mais tempo para ser concluído que os programas de teste da CPU, porém ele possibilita encontrar a combinação ótima desses parâmetros para um delay máximo escolhido. Os programas utilizados de medição de tempos da GPU e CPU se encontram em anexo à entrega do projeto.

**5.6.3 Programas MATLAB**

Foram realizados programas no MATLAB para suporte e planejamento do programa principal. O primeiro programa feito tinha em vista analisar a distorção causada pela função não linear arco tangente, sozinha e em cascata, tendo em vista também quantas distorções aproximadamente são necessárias para cada grau de distorção para causar o mesmo nível de distorção no final. Os resultados desse programa estão no tópico 4.2.1.

Um programa para aplicar o algoritmo que calcula os coeficientes das harmônicas da potência de um tom e assim calcular numericamente os harmônicos gerados pela aplicação da função não linear nesse tom, com resultados sendo apresentados no tópico 4.2.2. Outro programa foi feito para analisar o espectro da aplicação do arco tangente em um tom e na combinação de dois tons, com resultados apresentados no final da seção 4.2.2. Por fim, para a distorção, os resultados da seção 4.2.3 advém de um programa que faz previsões do aliasing gerado por uma distorção do arco tangente.

Para ajudar a depurar o programa principal, um programa foi gerado no MATLAB que realiza as mesmas operações do programa principal a não ser do paralelismo. Com esse programa pudemos comparar os resultados do programa principal, comparando espectros e ondas no tempo. Vale ressaltar que nesse programa contém também as especificações dos filtros e o seu projeto, salvando os coeficientes dos filtros em arquivos texto para serem lidos pelo programa principal na GPU. Os programas em Matlab utilizados se encontram em anexo à entrega do relatório.

# **6 Resultados**

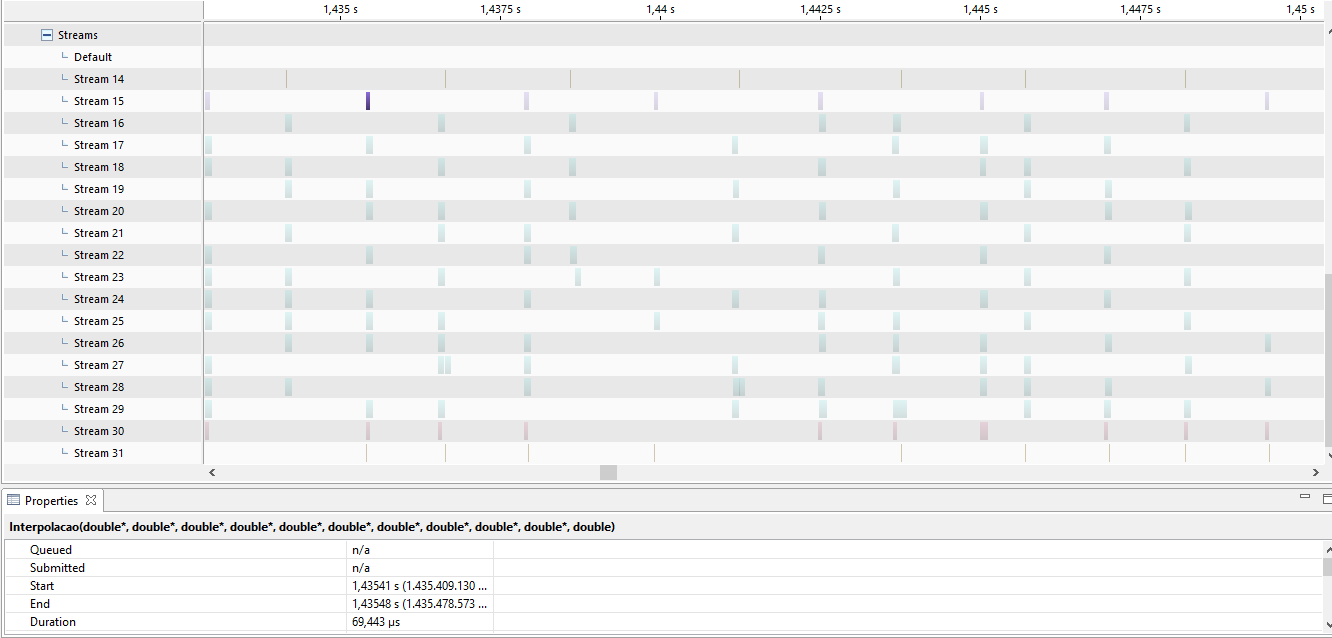
**6.1 Atraso entre leitura e escrita de amostras**

**6.1.1 Considerações sobre Tempo Real**

Conforme discutido na seção 4.6, para um correto funcionamento em tempo real, nosso sistema deveria ser capaz de ler amostras da entrada em um tempo menor ou igual ao período de amostragem do áudio. Entretanto, a cada nova amostra de chegada temos que lançar um kernel de interpolação e só podemos processar outra amostra quando a interpolação da amostra anterior já terminou (devido à filtragem IIR). Isso impõe a restrição que o kernel de interpolação tenha que ser executado muito rapidamente, a menos do período de amostragem do áudio de entrada. Além disso, ainda com base nas discussões apresentadas em 4.6, todos os kernels do sistema teriam que ter aproximadamente o mesmo tamanho, impondo a restrição de tempo de execução em todos os kernels do processamento.

Ao tentar implementar essa ideia, percebemos que quando os kernels ficam muito pequenos, o tempo entre o fim da execução de um e o começo da execução de outro começa a ser mais relevante para o atraso do sistema do que o próprio kernel. Além disso, o tamanho mínimo de kernel que conseguimos gerar - para o mínimo de operações necessárias para uma amostra, ou seja, apenas a filtragem de interpolação - já fica maior que o período de amostragem do áudio, aproximadamente 70 μs como mostra a Figura X .

O fenômeno do aparecimento do atraso entre as execuções dos kernels acontece porque a GPU é projetada para fazer operações complexas, de grande capacidade computacional. Sua excelência se mostra quando fazemos operações desta natureza. Quando tentamos fazer operações muito simples, a simples chamada da API do CUDA pode levar mais tempo que a própria execução do kernel, quebrando a ideia de pipeline desejada no sistema.



(Imagem do pipeline ruim)

Nota-se, então, que para ter um pipeline mais eficiente precisamos aumentar o número de operações que cada kernel realiza, o que impossibilita o processamento em tempo real do sistema dado que isso implica que o tempo de execução do kernel ficaria cada vez maior em relação ao período de amostragem e ainda pelo fato que cada kernel processa apenas uma amostra do sinal original por vez (L amostras do sinal interpolado). Uma das opções para contornar esse problema seria a leitura de mais amostras de uma vez, o que permitiria que os kernels de interpolação referentes a essas amostras fossem executados com menos atrasos entre eles. Porém, sempre que precisássemos copiar essas amostras para a GPU, teríamos certo atraso no lançamento dos kernels, resultando numa flutuação do tempo de processamento das amostras ao longo do sinal de entrada. Isso geraria uma variação também na taxa de escrita da saída, que por fim também impossibilita a satisfação das requisições para termos o processamento em tempo real.

Com base nessa discussão, concluímos que a maneira como implementamos nosso sistema não seria capaz de atingir os requisitos necessários para processamento em tempo real. Algumas ideias que tivemos, mas não conseguimos implementar, que talvez poderiam satisfazer esses requisitos foram: possibilidade de escrita das amostras diretamente na memória da GPU ou possibilidade de leitura de um buffer grande de amostras ao invés da leitura de uma amostra por vez.

A ideia de ler um buffer de amostras ao invés do uso de uma amostra por vez também remonta a possibilidade do processamento ser realizado com filtros FIR. Filtros FIR dependem apenas de um certo número de amostras passadas para o processamento de cada amostra, assim se o tamanho do buffer lido for suficientemente grande, seria possível processar várias amostras ao mesmo tempo, sem ter que esperar a interpolação de uma amostra ter acabado para que a próxima seja executada, por exemplo. Essa ideia não foi implementada por completo devido a limitações de tempo do projeto, porém vale comentar que um kernel que implementa de forma paralela a interpolação polifásica de um buffer de entrada foi completado e testado para apenas uma Stream .

Fizemos pesquisas às documentações do CUDA e aos fóruns da comunidade, porém sempre encontramos que o procedimento para processamento em GPU é o mesmo: a amostra tem que ser copiada do host para o device, e depois de processada copiada do device ao host. Se conseguíssemos, de alguma forma, que essas amostras fossem escritas e lidas diretamente do device, o processamento certamente seria mais rápido e as requisições para tempo real poderiam ser alcançadas, pois as cópias entre CPU e GPU levam um tempo considerável de processamento.

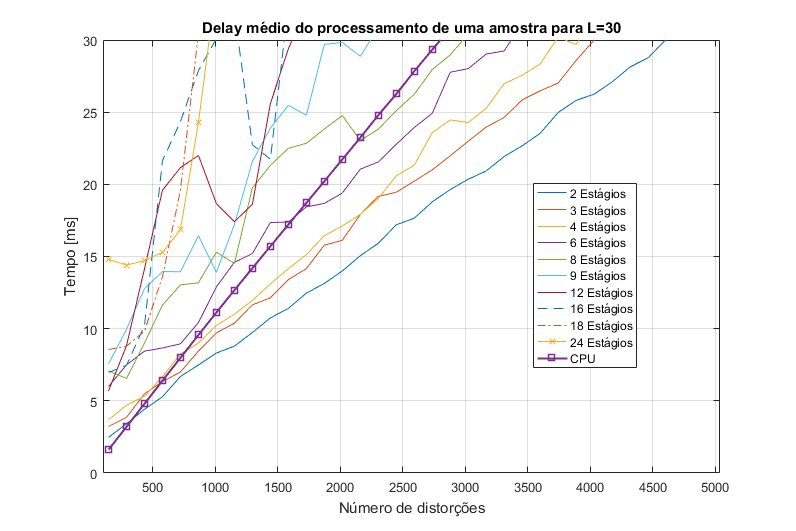
A cópia de um buffer maior de amostras para a GPU pode ser mais interessante, visto que os kernels de interpolação poderiam executar mais operações, porém isso não foi implementado pois a ideia que se imaginou do sistema é que a chegada de amostras no conversor A/D geraria uma interrupção que faria a cópia dessa amostra para a GPU. Portanto, a GPU receberia sempre uma amostra por vez.

Apesar disso, decidimos testar nossos algoritmos sem considerar que a entrada e saída deveriam ter a taxa de amostragem do áudio, para observar tanto a qualidade do processo de distorção do áudio com as interpolações quanto o ganho obtido em tempo de processamento quando usamos a GPU, comparando-o com a CPU. Os resultados desses testes são mostrados nas seções a seguir.

**6.1.2 Análise do Delay do Sistema**

Para avaliar o input delay do nosso sistema, ou seja, o tempo que ele leva entre receber uma amostra e entregar a mesma processada, decidimos criar programas que medem esse intervalo de tempo e os escrevem em arquivos de saída. Para a CPU, a medição desse tempo é bem direta, visto que as tarefas são feitas sequencialmente. Para esse caso, então, a medição foi feita através da contagem do número de clocks que o sistema demora desde o começo até o fim de cada amostra. Já para a GPU, foi preciso criar Events do CUDA, que são marcados a cada leitura e escrita de amostra, e usar a função cudaEventElapsedTime para medir o tempo entre essas tarefas.

Visto que a GPU possui diversas configurações diferentes para realizar um mesmo número de distorções, foi necessário analisar o delay para vários números de estágio. Na Figura X está apresentado o delay médio apresentado pelo sistema para o processamento de uma amostra, frente ao delay médio pelo mesmo procedimento feito na CPU, usando um fator de interpolação de 30 vezes. A média foi feita a partir do delay de mil amostras de um arquivo de áudio lançado 35 vezes, portanto cada ponto sendo a média de 35000 delays. A Figura X abaixo foi limitada para um delay de 30 ms, que foi o valor estabelecido para considerarmos o processamento em tempo real.



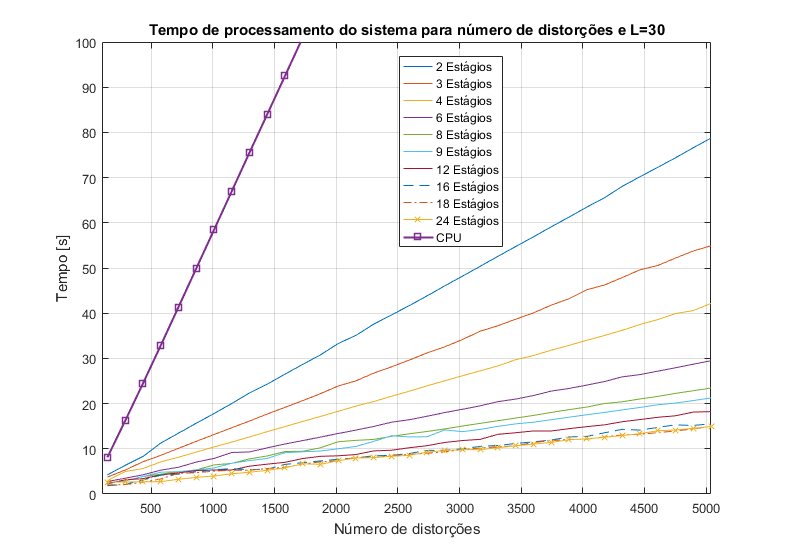
É visível que o delay médio cresce não apenas com o número de distorções, mas também com o número de estágios lançados. Em um caso ideal de pipeline, onde os estágios de distorção pudessem ser lançados instantaneamente após o estágio anterior ter terminado para um mesmo número de distorções, o delay médio seria igual para números de estágios diferentes vezes o tempo de processamento de cada estágio, pois a amostra tem que passar por todos os estágios até chegar na saída. Entretanto, na prática, existe um tempo para que o host determine que o estágio de distorção terminou a sua execução, demande o próximo estágio à GPU e o processamento do novo estágio se inicie, acumulando um delay a mais no processamento de cada amostra. Assim, por mais que mais estágios tragam um maior aproveitamento dos recursos, ele também representa um compromisso tendo em face delays toleráveis para um sistema em tempo real.

De fato, para um número de estágios suficientemente elevado, o delay a mais do sistema para o lançamento de novos estágios supera as vantagens do paralelismo em relação a CPU. Pela Figura X, concluímos que a desvantagem do número de estágios supera as vantagens para mais do que 8 estágios. Para restrições de tempo real de delays até 30 ms, usando uma interpolação de 30 vezes, pela curva, é possível perceber que a CPU encontra seu limite em aproximadamente 2750 distorções, enquanto a GPU consegue realizar para dois estágios apenas, um pouco mais do que 4500 distorções, claramente uma vantagem em relação a CPU.

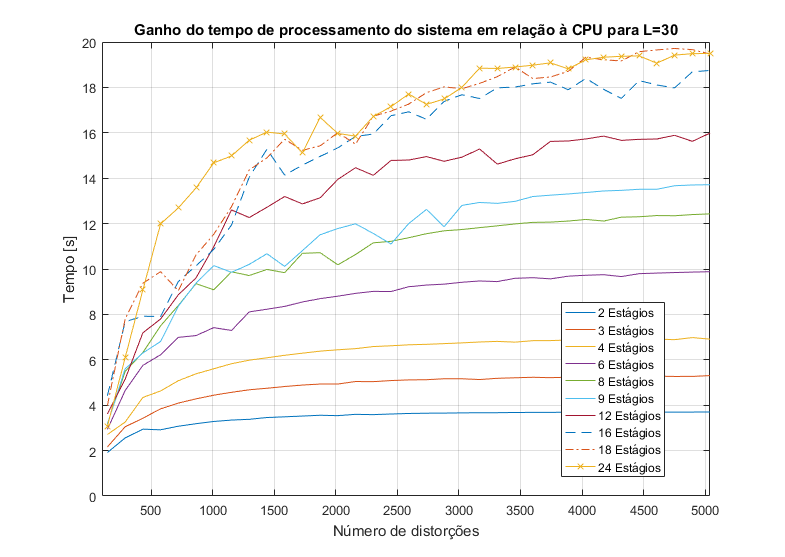
Por fim, considerando a discussão feita na seção 6.1.1, se pudéssemos para essa arquitetura realizar a cópia de memória de entrada e saída da GPU a taxa de amostragem do sistema, o único atraso referente ao processamento de uma amostra seria o seu delay de processamento. Nesse caso, para L=30, conseguiríamos para poucos estágios, uma vantagem em relação a CPU, podendo, em tempo real, realizar mais operações de distorção.

**6.2 Análise do tempo de processamento offline**

Dado que o processamento em tempo real foi considerado impraticável no tópico 6.1.1, tornou-se necessário considerar uma métrica para avaliar a eficiência do processamento da GPU em relação a CPU com o processamento feito de forma totalmente offline. Assim, foram levantadas curvas do tempo de processamento médio de um arquivo fixo, para diversas configurações de estágios, além do tempo de processamento médio na CPU, mostradas na Figura X. A média foi considerada com o processamento de cinco vezes o mesmo arquivo, contendo aproximadamente 105 ms de áudio.



Diferentemente do comportamento da curva ao se medir o delay do sistema, seção 6.1.2, pode-se perceber que para um mesmo número de distorções, mais estágios garantem um tempo de processamento menor do áudio, transparecendo as vantagens do maior aproveitamento dos recursos da GPU com mais estágios. Apesar disso, qualquer número de estágios escolhido demonstrou superioridade em relação ao processamento sequencial, para qualquer número de distorções. Para evidenciar a superioridade do tempo de processamento, calculamos o ganho - razão entre o tempo apresentado pela CPU pelo tempo da GPU - apresentado na Figura X. Dessa Figura é possível perceber que para aproximadamente 5 mil distorções e L=30, a GPU consegue um processamento até aproximadamente vinte vezes mais rápido do que a CPU devido ao paralelismo de operações.



Existe um limite do hardware em relação a sua compute capability em relação ao número de Streams concomitantes no pipeline. Devices com compute capability 6.2, como o caso do módulo Jetson TX2 (tabela da seção 3.2.1), permitem até que 16 kernels estejam em operação ao mesmo tempo (primeira linha da tabela 15 de [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index. html#features-and-technical-specifications\_\_technical-specifications-per-compute-capability](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications__technical-specifications-per-compute-capability)). Assim, por mais que mais do que 16 estágios sejam lançados, como no caso de 18 e 24 estágios nas figuras, ao mesmo tempo apenas 16 ocorrerão, ou seja, os demais estágios ocorrerão em série. Esse comportamento é visível nas Figuras X e Y: para além de 16 estágios, a mesma performance aproximadamente é alcançada para um mesmo número de distorções, revelando um limite de capacidade superior do hardware em questão.

**6.3 Áudios Processados**

**6.3.1 Sweep**

Para testar a funcionalidade do programa em seus processos de filtragem e interpolação, usamos um sinal de sweep digital gerado pelo auxílio do software Audacity. O sinal foi gerado para variar sua frequência de 1 a 15 kHz em 5 segundo e gerado em 48 kHz. O mesmo sinal foi processado na CPU, sem usar interpolação, apenas aplicação a distorção repetidamente, para fins comparativos. O sinal de sweep, por conter diversos tons fundamentais, é muito passível de conter frequências rebatidas em potência suficiente para a audição. Assim, é possível comparar a eficiência do programa em reduzir efeitos de aliasing por meios audíveis.

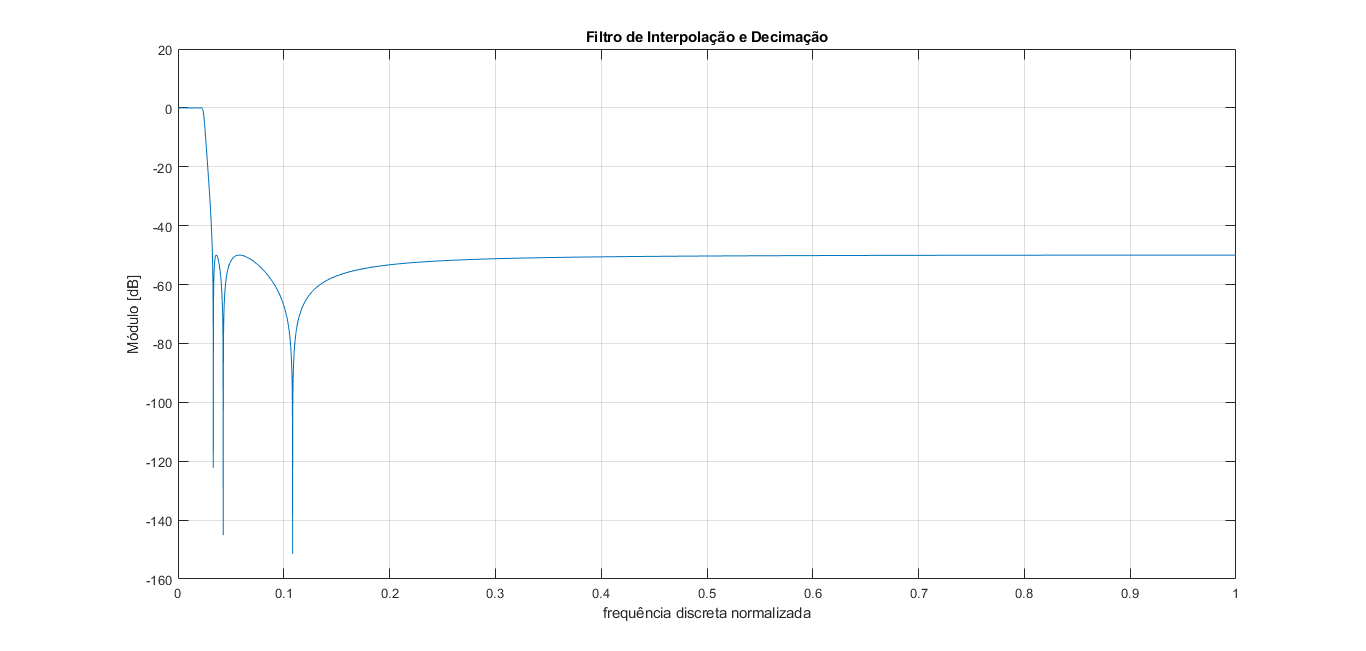
Visando então a comparação sonora, decidimos por usar 2000 distorções de , já que essa distorção causa aliasing sonoro considerável no áudio distorcido sem interpolação. Para gerar resultados satisfatórios para um nível de distorção considerado satisfatório por nós, para um fator de interpolação de 30 vezes, projetamos os filtros, de forma heurística, com as seguintes especificações:

| *Interpolação e Decimação:*  Ripple da faixa de passagem: 0.025 dB  Atenuação na faixa de corte: 50 dB  Frequência de passagem: (16 kHz)  Frequência de corte: (24 kHz)  Ordem do filtro: 6 | *Pós distorção:*  Ripple da faixa de passagem: dB  Atenuação na faixa de corte: 100 dB  Frequência de passagem: (16 kHz)  Frequência de corte: (50,400 kHz)  Ordem do filtro: 7 |
| --- | --- |

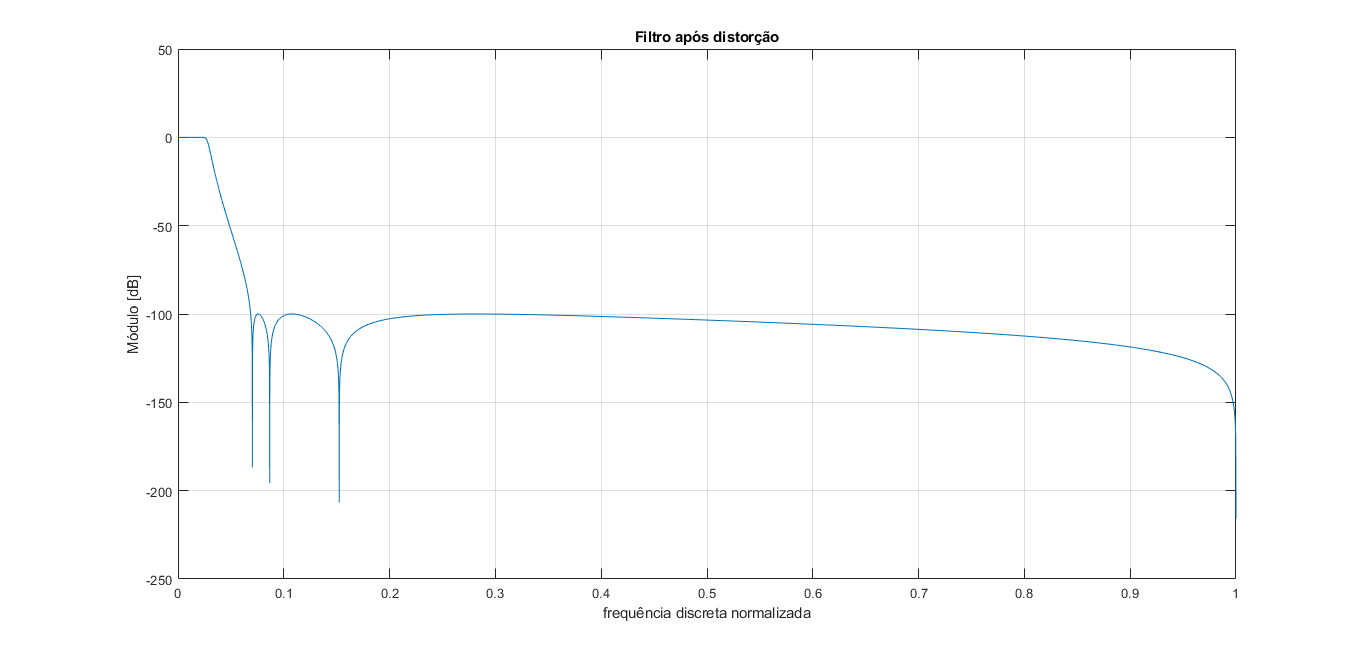
Os filtros de interpolação e decimação contém a faixa de passagem mais estreita, enquanto os filtros após a distorção podem ser mais permissivos como discutido no tópico 4.2.3. Além disso, sua faixa de passagem diminui com o aumento de L. Para L=30, não encontramos atenuações da faixa de rejeição satisfatórias para a pequena faixa de transição correspondente a esse fator de interpolação. Assim, foi decidido separar o filtro em uma série de filtros e especificações mais brandas em cascata. O ripple foi fixado em 0.025 dB, resultando em 0.1 dB de ripple da cascata (desvio de ) e a atenuação do corte em 50 dB, gerando 200 dB finais.

Para o filtro após a distorção, como o grau de distorção aplicada é pequeno (<0.1), aproximamos a cascata de distorções e filtragens como apenas de filtragens, assim para dB em cascata de 2000 filtros gera aproximadamente um ripple de 1 dB (desvio de ). Na faixa de rejeição, 100 dB de corte gerou resultados agradáveis e assim foi mantido. Como faixa de passagem como último parâmetro a ser escolhido, para demais parâmetros fixos, o valor acima de 24 kHz encontrado satisfatório foi de 50,400 kHz.

As máscaras dos filtros finais se encontram as figuras X e Y a seguir:

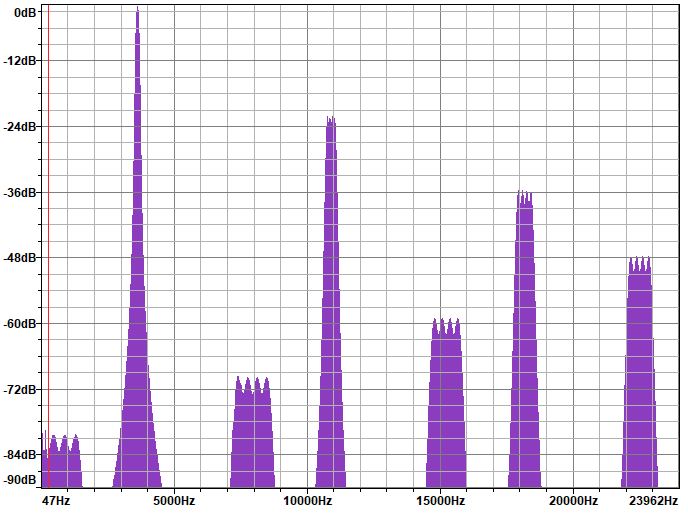


*Figura X: Filtro de interpolação e decimação.*

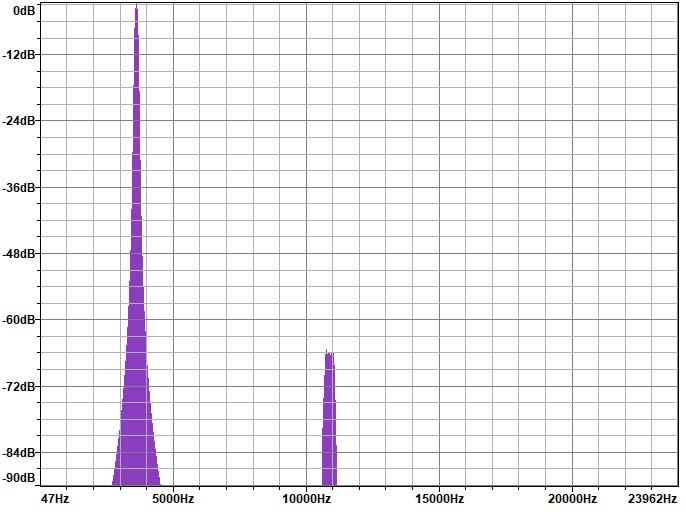


*Figura X: Filtro de distorção*

Nas Figuras X e Y estão apresentados espectros da mesma parte do áudio do sweep distorcido sem interpolação e o sweep distorcido no jetson usando os filtros mencionados.



*Figura X: Espectro do sweep em um intervalo em torno de 1.2s distorcido sem interpolação e filtragem.*



*Figura X: Espectro do sweep em um intervalo em torno de 1.2s distorcido com interpolação e filtragem.*

Podemos aproximar o trecho do sweep observado como uma composição de tons em torno de 3,5 kHz, que é a primeira raia observada. Na figura X, do áudio processado sem a interpolação, é possível ver claramente que as harmônicas do sinal distorcido sofrem aliasing - as harmônicas principais parecem uma escada descendente para a direita, até que depois de 24 kHz as harmônicas rebatidas fazem uma escada descendente para a esquerda.

Já no sinal processado com interpolação notamos que as componentes rebatidas desapareceram, além de que as componentes não rebatidas tiveram um decréscimo de amplitude, devido à retirada do aliasing que estava presente nessas raias no sinal sem interpolação. Além disso, temos que a faixa de transição do nosso filtro de interpolação começa nos 16 kHz, o que causa uma distorção não desejada do sinal, diminuindo ainda mais a amplitude das harmônicas depois dessa frequência.

**6.3.2 Sinal de guitarra gravado**

Para ver o efeito do processamento em um sinal real de guitarra, utilizamos um sinal gravado a 48 kHz e ouvimos ele antes e depois de processar. As configurações de processamento foram as mesmas utilizadas na seção anterior (6.3.1), que conseguiu eliminar o aliasing do sweep.

# **7 Conclusão**

Ao rever todos os passos necessários para realizar este projeto, podemos avaliar que sua execução foi bem desafiadora. Desde a familiarização com o ambiente de desenvolvimento do Jetson, até a adaptação com os paradigmas de programação de CUDA e com novas bibliotecas de áudio. Avaliamos que houve uma quantidade considerável de aprendizados.

Notamos também que as expectativas quanto aos resultados estavam mais altas do que fomos capazes de realizar. O fato de não ter sido possível implementar a entrada de áudio no Jetson já causou uma grande mudança no produto final que iríamos conseguir entregar. Somado a isso, ao tentar fazer uma avaliação “offline” da viabilidade do projeto, fomos surpreendidos com as limitações do CUDA para quanto ao processamento em tempo real, que também se afasta bastante do resultado que gostaríamos.

Apesar disso, depois de muita dedicação, foi possível levantar justificativas coerentes para o não atendimento de nossas especificações, além de termos tentado elaborar outros caminhos a partir dos quais elas poderiam ser satisfeitas. Mostramos também que conseguimos implementar um pipeline com uma ocupação considerável da GPU - máximo de Streams possíveis - que conseguiu grandes ganhos em relação ao processamento em CPU, para um grande número de estágios de distorção.

Conseguimos bons resultado na comparação dos processamentos da GPU frente a CPU, utilizando a sobreamostragem e técnicas de paralelismo atingimos nosso objetivo de utilizar um número alto de etapas de processamento com distorções leves, que juntamente com a sobreamostragem, reduz o aliasing presente nas distorções digitais de sinais. Dessa forma, conseguimos atingir as especificações de manter o delay de processamento abaixo de 30ms e aliasing abaixo de 80dB, sendo assim, este método se mostrou viável para o processamento digital de sinais de áudio.

# **8 Bibliografia**

DOUMLER, T. Audio in standard C++. **Github**, 2019. Disponivel em: <https://github.com/boostcon/cppnow\_presentations\_2019/blob/master/05-08-2019\_wednesday/Audio\_in\_Standard\_Cpp\_\_Timur\_Doumler\_\_cppnow\_05082019.pdf>. Acesso em: 05/07/2021 Julho 2021.

FRANKLIN, D. NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge. **Developer NVIDIA**, 2021. Disponivel em: <https://developer.nvidia.com/blog/jetson-tx2-delivers-twice-intelligence-edge/>. Acesso em: 05/07/2021 Julho 2021.

JETSON Download Center. **Developer NVIDIA**, 2021. Disponivel em: <https://developer.nvidia.com/embedded/downloads>. Acesso em: 07/05/2021 Julho 2021.

JETSON Modules. **Developer NVIDIA**, 2021. Disponivel em: <https://developer.nvidia.com/embedded/jetson-modules>. Acesso em: 05/07/2021 Julho 2021.

JETSON TX2 Module. **Developer NVIDIA**, 2021. Disponivel em: <https://developer.nvidia.com/embedded/jetson-tx2>. Acesso em: 05/07/2021 Julho 2021.

KEEN, R. G. The Technology of the Fuzz Face. **Geoflex**, 1998. Disponivel em: <http://www.geofex.com/article\_folders/fuzzface/fffram.htm>. Acesso em: 05/07/2021 Julho 2021.

NVIDIA. CUDA C++ Programming Guide. **Docs NVIDIA**, 2021. Disponivel em: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Acesso em: 05/07/2021 Julho 2021.

OPPENHEIM, A. V.; SCHAFER, W. Processamento em tempo discreto de sinais. 3. ed. [S.l.]: Pearson Education, 2012. Cap. 4, p. 688.

PORTAUDIO Documentation. **PortAudio**. Disponivel em: <http://www.portaudio.com/>. Acesso em: 2021 Julho 2021.

L. Savioja, V. Välimäki, and JU. O.. Smith, "Audio Signal Processing Using Graphics Processing Units," J. Audio Eng. Soc., vol. 59, no. 1/2, pp. 3-19, (2011 January.). doi: