Processamento de áudio em tempo real em plataformas computacionais de alta disponibilidade e baixo custo.

André Jucovsky Bianchi

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa de Pós Graduação em Ciência da Computação
Orientador: Prof. Dr. Marcelo Gomes de Queiroz

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES

São Paulo, Agosto de 2013

Processamento de áudio em tempo real em plataformas computacionais de alta disponibilidade e baixo custo.

Esta é a versão original da dissertação elaborada pelo candidato André Jucovsky Bianchi, tal como submetida à Comissão Julgadora.

Agradecimentos

Ao Marcelo, meu valoroso orientador, à Foz, minha corajosa companheira, e à Lilian, minha generosa progenitora, agradeço por terem toda a paciência do mundo. A todo o pessoal do Grupo de Computação Musical, agradeço a companhia e a força que me deram para completar esta fase.

Resumo

BIANCHI, A. J. **Processamento de áudio em tempo real em plataformas computacionais de alta disponibilidade e baixo custo.**. 2012. 120 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2012.

Neste trabalho foi feita uma investigação sobre a realização de processamento de áudio digital em tempo real utilizando três plataformas com características computacionais fundamentalmente distintas porém bastante acessíveis em termos de custo e disponibilidade de tecnologia: Arduino, GPU e Android. Arduino é um dispositivo com licenças de hardware e software abertas, baseado em um microcontrolador com baixo poder de processamento, muito utilizado como plataforma educativa e artística para computações de controle e interface com outros dispositivos. GPU é uma arquitetura de placas de vídeo com foco no processamento paralelo, que tem motivado o estudo de modelos de programação específicos para sua utilização como dispositivo de processamento de propósito geral. Android é um sistema operacional para dispositivos móveis baseado no kernel do Linux, que permite o desenvolvimento de aplicativos utilizando linguagem de alto nível e possibilita o uso da infraestrutura de sensores, conectividade e mobilidade disponível nos aparelhos. Buscamos sistematizar as limitações e possibilidades de cada plataforma através da implementação de técnicas de processamento de áudio digital em tempo real e da análise da intensidade computacional em cada ambiente.

Palavras-chave: arduino, gpu, android, processamento de áudio em tempo real.

Abstract

BIANCHI, A. J. **Real time digital audio processing using highly available, low cost devices**. 2012. 120 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2012.

This text describes an investigation about real time audio signal processing using three platforms with fundamentally distinct computational characteristics, but which are highly available in terms of cost and technology: Arduino, GPU boards and Android devices. Arduino is a device with open hardware and software licences, based on a microcontroller with low processing power, largely used as educational and artistic platform for control computations and interfacing with other devices. GPU is a video card architecture focusing on parallel processing, which has motivated the stufy of specific programming models for its use as a general purpose processing device. Android is an operating system for mobile devices based on the Linux kernel, which allows the development of applications using high level language and allows the use of sensors, connectivity and mobile infrastructures available on devices. We search to systematize the limitations and possibilities of each platform through the implementation of real time digital audio processing techinques and the analysis of computational intensity in each environment.
Keywords: arduino, gpu, android, real time audio processing.

Sumário

Capítulo 1

Introdução

Este trabalho se situa no contexto de diversas pesquisas em diferentes áreas do conhecimento que têm motivações tanto técnicas quando estéticas para explorar as possibilidades de abordagem algorítmica do fenômeno sonoro. Uma primeira leitura do título pode dar a impressão de que seja demasiado ousado por tratar de temas de abordagem bastante difícil como "tempo real", "alta disponibilidade" e "baixo custo". Apesar disso, a proposta é que se compreenda cada um destes termos de forma a motivar uma interpretação específica do sentido da tecnologia e seu uso. Assim, faz-se necessário explicar cada termo separadamente.

Processamento de sinais é uma área da engenharia que lida com a captura, manipulação e emissão de sinais analógicos, e em particular sinais analógicos temporais, ou seja, funções que são definidas sobre uma variável real (contínua) que representa o tempo. A possibilidade de amostragem e captura de uma representação discreta (com perda de informação) dos sinais analógicos e o advento do computador, que permite a representação e manipulação rápida de símbolos discretos, deram origem à área de processamento de sinais digitais. Muitas das ferramentas do domínio analógico estão presentes no domínio digital com as devidas adaptações e restrições.

A noção de **tempo real** utilizada ao longo deste trabalho é bastante mais simples do que o problema usual da restrição de tempo real para aplicações críticas como por exemplo controle de voo ou de pressão de caldeira. Entendemos que um certo processamento de sinais é viável em tempo real quando existe uma limitação de atraso (constante) entre a entrada e a saída do sistema, permitindo em teoria uma produção de fluxo de saída ininterrupta. A motivação deste trabalho é desenvolver ferramentas para viabilizar a utilização de certos tipos dispositivos para processamento de áudio para fins técnicos, artísticos e educativos. Assim, a restrição do tempo real utilizada é "fraca": o atraso no processamento, e consequentemente na geração de novas amostras de áudio ou de um fluxo de características de um sinal de entrada, pode gerar artefatos audíveis, mas em geral não oferece perigo que não seja estético.

A ideia de **alta disponibilidade** está relacionada à facilidade de obtenção dos dispositivos computacionais utilizados e sua abundância no entorno dos cenários de utilização pretendidos. Isto pode ser compreendido tanto em termos de preço e presença significativa no mercado, como de possibilidade de reutilização e reciclagem de tecnologia usualmente considerada obsoleta. Uma outra palavra que pode descrever algo próximo é "ubiquidade": a característica de onipresença da tecnologia abordada nos cenários considerados.

Por fim, **baixo custo** é uma característica bastante relativa, que depende de características estatísticas do salário de uma determinada região (como média, desvio padrão e concentração de renda) e do custo (econômico, social e cultural) e poder computacional das tecnologias utilizadas para uma certa aplicação. No caso do processamento de áudio, em um extremo se encontram equipamentos profissionais como equipamentos profissionais de estúdio, computadores altamente especializados e processadores paralelos programáveis, e no outro extremo este trabalho propõe que se considerem dispositivos cada vez mais baratos e comuns como interfaces minimais para microcontroladores, dispositivos móveis programáveis e placas gráficas com processadores do tipo GPU.

Computação e música

Muitas são as possibilidades de uso da tecnologia na produção artística sonora como, por exemplo, o desenvolvimento de controladores de dispositivos sonoros, a criação de instrumentos expandidos, nos quais a utilização da tecnologia ajuda a gerar novos sons ou novas formas de interação, e o desenvolvimento de certa inteligência musical, através da detecção de características de um conjunto de sinais e atuação no ambiente como resultado de raciocínio computacional. A forma de utilização de tecnologia que interessa a este trabalho é o processamento de áudio em tempo real utilizando diferentes plataformas computacionais, de modo que se possa utilizar o resultado da computação em uma sessão ao vivo.

Quanto às plataformas computacionais existentes, os sistemas desenvolvidos para apresentação artística interativa geralmente se encontram entre dois extremos. Em um extremo estão os sistemas desenvolvidos com objetivo de auxiliar um trabalho de um artista específico. No outro extremo, estão sistemas dotados de alta flexibilidade que se propõem a auxiliar artistas com as mais diversas concepções estéticas. A utilização de plataformas com baixo custo e alta disponibilidade para este tipo de processamento pode aproximar artistas de novas possibilidades na construção de sistemas musicais de qualquer tipo. Para a escolha das plataformas computacionais a serem abordadas, algumas métricas qualitativas são a possibilidade de extensão das funcionalidades de cada plataforma, seu custo e licenças de uso associadas, e as possibilidades de integração e/ou comunicação remota destas três plataformas com ferramentas tradicionais para processamento de som em tempo real.

Diversas técnicas de processamento de áudio em tempo real têm sido utilizadas em apresentações artísticas. As possibilidades estéticas oferecidas pela utilização de instrumentos elétricos e eletrônicos surgem quando artistas entram em contato com estas ferramentas e adquirem conhecimento suficiente para realizar a exploração destes recursos. Alguns exemplos de uso artístico da tecnologia são a subversão de circuitos eletrônicos com o objetivo de criar novos instrumentos, como na prática denominada Circuit Bending[1], e o desenvolvimento de sistemas complexos dotados de autonomia musical e capacidade de interação, como os sistemas *Cypher* (Rowe, 1992a) e *Voyager* (Lewis, 2000). O tipo de apresentação artística interativa na qual artista e aparato tecnológico dividem o palco e interagem em tempo real muitas vezes ocorre como fruto de pesquisa realizada em trabalhos acadêmicos. Trabalhos recentes desta natureza podem ser encontrados nos anais de conferências organizadas pela *International Computer Music Association* (ICMA) e pela comunidade de pesquisa em *Sound and Music Computing* (SMC), nos anais do *Simpósio Brasileiro de Computação Musical* (SBCM), além de periódicos especializados como *Leonardo Music Journal* (LMJ), *Organized Sound* (OS) e *Computer Music Journal* (CMJ).

1.1 Objetivos

Tendo como base o que foi discutido na seção anterior, pode-se formular o objetivo principal deste trabalho como a investigação das possibilidades de implementação, execução e desempenho de diferentes algoritmos de processamento de áudio em tempo real em plataformas computacionais escolhidas por serem baratas e de fácil obtenção, além de possuírem características técnicas bastante distintas. As próximas seções detalham o conceito de processamento de áudio em tempo real, as plataformas computacionais abordadas e o tipo de análise de desempenho realizada em cada uma delas.

1.1.1 Processamento de áudio em tempo real

Informalmente, processamento de áudio digital é a manipulação de uma sequência de valores (igualmente espaçados no tempo e que representam a evolução temporal de um sinal de áudio) através da utilização de diversas ferramentas matemáticas. A cada valor desta sequência é dado o nome de **amostra**. Em geral, as amostras representam a amplitude de uma onda sonora em

[1] http://www.anti-theory.com/soundart/circuitbend/cb01.html

diferentes instantes, e a variação destes valores ao longo do tempo pode codificar sinais que são percebidos pelo aparelho auditivo humano como som. Um dos possíveis objetivos da manipulação do sinal digital é a captura de um conjunto de métricas matemáticas que podem ser relacionadas a descritores perceptuais como ritmo, altura musical, brilho e rugosidade. Outra possibilidade é a manipulação do sinal original com o objetivo de produzir um novo sinal com características alteradas, como por exemplo um novo período, uma nova altura musical ou um novo espectro com regiões intensificadas ou atenuadas.

No contexto do processamento em tempo real, supõe-se que o sinal digital não está completamente determinado no início da computação e, ao contrário, é disponibilizado para o algoritmo em uma taxa fixa, em geral relacionada à frequência de amostragem do sinal. Assim, os algoritmos utilizados em processamento em tempo real não conhecem as amostras do futuro e têm que se limitar às amostras já capturadas. Fica claro que, antes que sejam recebidas todas as amostras do sinal de entrada, não é possível gerar um fluxo de características sonoras ou um novo sinal de saída que dependam do sinal de entrada completo. A solução mais comum para esta restrição é a acumulação de um bloco de amostras consecutivas e a manipulação do sinal original considerando o bloco atual e eventualmente os blocos passados. Assim, é possível gerar os resultados da computação correspondente a cada bloco dentro de um período aceitável de forma que se possa repetir o processo enquanto houver novas amostras de entrada.

Dado um sinal digital amostrado em tempo real a uma frequência igual a R Hz, costuma-se supor que novas amostras são disponibilizadas para o algoritmo através de um *buffer* de entrada de tamanho N. O período correspondente a um bloco de N amostras é N/R segundos e por isso N amostras novas do sinal estão disponíveis para manipulação a cada N/R segundos. Para que se possa iniciar a computação sobre um novo bloco, o ideal é que a computação sobre o bloco anterior já tenha terminado. Além disso, para que seja possível emitir um novo sinal sem descontinuidades, é necessário que as N novas amostras sejam calculadas antes do final do período do bloco. Assim, o período de um bloco de N amostras corresponde também ao tempo máximo disponível para manipulação do bloco e cálculo do resultado do processamento.

Neste sentido, é possível definir o **ciclo DSP**, ou "ciclo de processamento do sinal digital", como uma iteração da seguinte sequência de eventos: (1) obtenção de um bloco de amostras do sinal de entrada, (2) processamento das amostras e geração das amostras de um novo sinal de saída ou de um fluxo de características sonoras associadas ao sinal de entrada, e (3) emissão dos resultados. Estes passos podem tomar diferentes formas dependendo do dispositivo que se esteja utilizando para realizar o processamento. Por exemplo, alguns dispositivos realizam a conversão entre sinais analógicos e digitais na entrada e na saída, enquanto que outros lidam somente com amostras digitais e dependem de outras interfaces para realizar a captura e emissão de amostras. Além disso, outros passos podem estar presentes no processo como por exemplo transferência de dados entre diferentes níveis de memória. Algumas destas diferenças e sua expressão nos dispositivos abordados neste trabalho serão consideradas mais adiante na Seção 2.1.1.

Até mesmo em dispositivos mais simples que possuem a capacidade de amostrar um sinal analógico, é comum que o processo de amostragem seja controlado por hardware específico e ocorra em paralelo com outros processos computacionais. Isto significa que, enquanto uma parte do hardware cuida de realizar a amostragem do sinal de entrada, outra parte do hardware pode iniciar um ciclo DSP sobre o último bloco completo capturado. No contexto de tempo real, o período do ciclo DSP deve ser menor ou igual ao período do bloco de amostras, de forma que o início de um novo ciclo DSP coincida com a emissão do resultado da computação do último bloco e com a disponibilidade de um novo bloco de amostras do sinal de entrada.

É importante reforçar que a restrição de que o período do ciclo DSP seja menor do que o período do bloco de amostras será satisfeita dependendo da quantidade de computação que se deseja realizar dentro do ciclo DSP e do poder computacional do dispositivo utilizado. Se a quantidade de computação desejada for muito grande ou o poder computacional do dispositivo utilizado for muito pequeno, o período do ciclo DSP poderá exceder o período do bloco de amostras. Se isto ocorrer, pode-se tentar interromper a computação do bloco atual e iniciar a do próximo, ou então

pode-se esperar que a computação termine causando um atraso na emissão do resultado e no início da computação do próximo bloco. Em ambos os casos ocorrem situações indesejadas: ausência e atraso no cálculo dos resultados da computação sobre o bloco atual, respectivamente. Tanto no caso do cálculo de características do sinal de entrada quanto no caso da síntese de um novo sinal pode haver falta ou atraso de dados com relação ao tempo real. No último caso, artefatos audíveis podem ser introduzidos no sinal.

Uma vez exposto informalmente o que se propõe por "processamento de áudio em tempo real", vale a pena repetir que este trabalho não tem como objetivo desenvolver técnicas de interrupção da computação ou recuperação de dados nos casos em que o período do ciclo DSP exceda o período do bloco de amostras. Ao contrário, este trabalho utiliza a medição do período do ciclo DSP em diferentes cenários para estabelecer se certas combinações de hardware e software são viáveis de serem executadas em tempo real ou não. Na Seção 2.2 será dada uma descrição mais formal e detalhada do problema, que fundamentará a metodologia, as implementações e os testes de desempenho.

1.1.2 Plataformas computacionais de alta disponibilidade e baixo custo

Já foi dito anteriormente que este estudo explora os limites e possibilidades de processamento de áudio digital em tempo real em dispositivos que possuem características computacionais bastante distintas, mas que são acessíveis para os usuários finais em termos de custo e tecnologia. Faltou, porém, explicitar quais são os dispositivos abordados e os motivos para tais escolhas.

A proposta do trabalho leva na direção de abordar plataformas que não tenham sido projetadas especificamente para o processamento de áudio, mas que também possam ser utilizadas para este fim. Para a escolha dos dispositivos, foi levado em conta não só o baixo custo de aquisição e facilidade de obtenção (em comparação com soluções profissionais), mas também outros aspectos relevantes como licença de utilização e nível de obsolescência, sendo que este último é levado em conta de forma oposta ao usual: quanto mais obsoleto, mais barato e abundante é o dispositivo em questão. É importante notar que a realidade quanto à disponibilidade das plataformas é diferente em contextos com diferentes recursos, legislação ou influência de outros fatores. Tendo em mente este sentido de disponibilidade, pretende-se aqui quantificar e qualificar as possibilidades de processamento de áudio digital em tempo real para o contexto, por exemplo, de apresentações artísticas ao vivo que disponham de um mínimo de investimento de recursos ou cujo contexto seja tal que estas tecnologias já estejam amplamente disponíveis como resultado de sua forte presença no mercado ou substituição por modelos mais novos.

Existem outras características que são desejáveis para a escolha das plataformas de estudo, como sua capacidade de interação com outros sistemas computacionais e adaptabilidade a diferentes cenários de utilização. Também é interessante que as plataformas possuam características bastante distintas umas das outras para motivar diferentes abordagens das técnicas de processamento e assim enriquecer o estudo.

As considerações acima, junto com a realidade do departamento de pesquisa no qual se desenvolveu este trabalho, levou à decisão de abordar três "famílias" de dispositivos que expressam diferentes frentes tecnológicas, expressas por representantes que foram considerados como significativos para cada uma delas. O primeiro tipo de dispositivo considerado é o microcontrolador, representado pela plataforma Arduino, uma combinação de hardware e software com licença aberta que compreende um microcontrolador e uma interface que possibilita a captura, processamento e síntese de sinais analógicos e digitais. O segundo tipo de dispositivo é o circuito do tipo GPU, disponível em grande parte das placas de vídeo comercializadas atualmente (e neste trabalho representado por placas gráficas da fabricante NVIDIA), que permite processamento paralelo e pode atuar como coprocessador de dados em conjunto com a arquitetura tradicional dos computadores pessoais. O terceiro tipo de dispositivo corresponde ao que se tem chamado de dispositivos móveis, abordados através da programação para o sistema operacional Android, que permite grande flexibilidade e abstração no desenvolvimento por se tratar de um sistema operacional completo e com partes de seu código publicado sob licenças abertas, além de apresentar abundância de sensores e

interfaces de conectividade.

Estes três tipos de dispositivos (microcontroladores, placas com GPU e dispositivos móveis) são máquinas Turing-completas e podem, portanto, computar tudo aquilo que é computável, respeitados os limites de memória de cada uma delas e de tempo dos usuários. Neste sentido, não são dispositivos desenvolvidos especificamente para o processamento de áudio, muito menos em tempo real. Para este fim, em geral são utilizados dispositivos específicos como por exemplo chips DSP como o modelo Blackfin da fabricante Analog Devices (ADBlackfin), que implementa uma plataforma RISC de 32 bits, ou processadores baseados em FPGA tais como a família Virtex-7 da fabricante Xilinx (XilVirt7), entre muitos outros. Avanços nas pesquisas e na indústria levaram a otimizações no desempenho computacional e consumo de energia nestas plataformas. Apesar disto, e em oposição aos tipos de dispositivos escolhidos para investigação neste trabalho, não são plataformas que estão imediatamente e abundantemente disponíveis para utilização no dia a dia.

Na próxima seção será feita uma descrição detalhada de cada um dos tipos de dispositivos estudados. No capítulo seguinte, uma visão geral da metodologia de análise de desempenho e das ferramentas matemáticas utilizadas fundamentará a abordagem subsequente dos dispositivos. Logo em seguida, três capítulos descreverão as possibilidades de programação e processamento de áudio em cada um dos tipos de dispositivo, bem como os resultados obtidos na análise de desempenho de cada um para o processamento de áudio em tempo real.

1.1.3 Análise de desempenho

Uma vez que os dispositivos escolhidos para investigação não foram desenvolvidos especificamente para processamento de áudio em tempo real, encontra-se na literatura pouca informação sobre suas possibilidades de uso para esta tarefa. A análise sistemática do desempenho destes sistemas computacionais provê uma ferramenta para escolha do equipamento a ser utilizado em função da necessidade dos usuários/artistas.

A capacidade computacional pode ser entendida como a quantidade de determinadas operações que é realizável em um certo espaço de tempo. As operações analisadas devem representar átomos do domínio de aplicação do problema a ser resolvido. Uma métrica bastante comum é a quantidade máxima de operações de número flutuante por segundo (*FLOPS*), intimamente relacionada à frequência do processador e as operações básicas implementadas em hardware. No caso do processamento de áudio, pode ser mais interessante observar como os dispositivos se comportam ao realizar tarefas comuns de manipulação de áudio tais como síntese aditiva, cálculo da FFT e da convolução no domínio do tempo. Como será visto na Seção 2.1.1, cada um destes algoritmos possui parâmetros que caracterizam tanto sua complexidade computacional quanto suas possibilidades de manipulação dos sinais digitais considerados. Quanto maior o número de osciladores, mais tempo é consumido na síntese aditiva, porém mais rico pode ser o espectro do sinal sintetizado. De forma similar, quanto maior o número de amostras consideradas na FFT, mais tempo toma-se para realizar a transformada e mais detalhado é o espectro resultante do cálculo. E assim por diante.

Para quantificar a capacidade computacional e os limites de processamento de sinais de áudio, é interessante utilizar os arcabouços e metodologias encontrados na literatura de forma a implementar diferentes algoritmos e técnicas de processamento de áudio digital em cada uma das plataformas escolhidas. Uma lista de algoritmos usuais de processamento de sinais foi levantada, e aqueles que foram utilizados para realizar a análise nas plataformas serão descritos no próximo capítulo.

1.2 Estudos de caso: três plataformas de processamento

Como visto nas seções passadas, três plataformas computacionais foram escolhidas para estudar o processamento de sinais digitais em tempo real e avaliar as restrições e possibilidades de implementação em relação às tecnologias disponíveis atualmente. A escolha das plataformas foi feita levando em conta seu baixo custo em relação a plataformas comerciais, a alta disponibilidade para usuários comuns e a existência de diferenças fundamentais de projeto, propósito e características

entre elas. Nas próximas seções, cada uma das plataformas escolhidas será descrita com o objetivo de dar uma visão sobre o histórico e as motivações do desenvolvimento e da escolha de cada uma delas para análise neste trabalho.

1.2.1 Microcontroladores: plataforma Arduino

O projeto **Arduino**[2] foi iniciado em 2005 com o objetivo de criar uma plataforma de desenvolvimento de projetos para estudantes mais barata dos que as disponíveis na época. Baseado na ideia de prover uma estrutura minimal para interface com um microcontrolador, o Arduino veio de uma ramificação do projeto Wiring[3], uma plataforma de desenvolvimento criada em 2003 com o objetivo de unir designers e artistas ao redor do mundo para compartilhar ideias, conhecimento e experiência, utilizando hardware e software com licenças abertas. O software utilizado pelo projeto Wiring, por sua vez, foi influenciado pela plataforma de desenvolvimento Processing[4], outro projeto aberto iniciado em 2001 por ex-integrantes do MIT Media Lab[5].

Tanto o hardware quanto o software do Arduino são publicados sob licenças de código aberto Source[6]. Os projetos originais das placas do tipo Arduino estão publicados sob a licença Creative Commons Attribution-ShareAlike 2.5[7], o que significa que o projeto do hardware não requer nenhuma permissão para o uso e permite trabalhos derivados tanto para uso pessoal quanto para uso comercial, contanto que haja crédito para o projeto oficial e que os projetos derivados sejam publicados sob a mesma licença. Já o software utiliza duas versões diferentes de licenças de software livre: o ambiente escrito em Java é liberado sob a licença GPL[8] e as bibliotecas em C/C++ do microcontrolador são liberados sob a licença LGPL[9]. Toda a documentação no sítio do projeto é publicada sob a licença Creative Commons Attribution-ShareAlike 3.0[10] e os trechos de código estão em domínio público. Atualmente, existem diversas empresas[11] e indivíduos fabricando o Arduino e, devido a este esquema de licenciamento, qualquer um com acesso às ferramentas adequadas pode construir uma placa a partir de componentes eletrônicos básicos.

A disponibilidade dos projetos das placas e do código fonte do compilador, somada a uma escolha adequada das licenças de distribuição, resultou em uma comunidade mundial que desenvolve plataformas de hardware e software que podem ser utilizadas para as mais diversas aplicações (respeitadas, é claro, as limitações do poder computacional dos microcontroladores utilizados). Também decorre destas escolhas a possibilidade de fabricação, industrial ou caseira, por um preço acessível. Hoje existem muitos outros projetos baseados no Arduino que utilizam microcontroladores do mesmo tipo ou outros modelos de fabricantes diferentes e com características distintas, mas totalmente compatíveis no nível do software[12]. Também é possível encontrar na internet uma variedade muito grande de módulos conectáveis às placas Arduino, tanto para comprar quanto para construir. Estes módulos têm como objetivo prover outras funções, desde as mais básicas como implementação de relógios digitais que possibilitam um controle mais fino da passagem do tempo[13], até funções mais avançadas como interconexão sem fio com outras plataformas[14].

Ao longo da existência do projeto Arduino, diversas versões do hardware foram desenvolvidas e publicadas[15]. Cada versão possui características diferentes como, por exemplo, tipo de conexão com

[2] http://arduino.cc/
[3] http://wiring.org.co/
[4] http://www.processing.org/
[5] http://www.media.mit.edu/
[6] http://www.opensource.org/docs/osd
[7] http://creativecommons.org/licenses/by-sa/2.5/
[8] http://www.gnu.org/licenses/gpl.html
[9] http://www.gnu.org/licenses/lgpl.html
[10] http://creativecommons.org/licenses/by-sa/3.0/
[11] http://www.arduino.cc/en/Main/Buy
[12] http://www.arduino.cc/playground/Main/SimilarBoards
[13] http://totusterra.com/index.php/2009/10/31/using-the-555-timer-as-an-external-clock
[14] http://arduino.cc/en/Main/ArduinoXbeeShield,http://jt5.ru/shields/cosmo-wifi/,http://jt5.ru/shields/cosmo-gsm/
[15] http://arduino.cc/en/Main/Hardware

o computador (USB, serial, FireWire), modelos diferentes do microcontrolador utilizado, formato, tamanho, facilidade de montagem (em termos de ferramentas e técnica necessárias), e até estética. Apesar disso, todos possuem a mesma interface básica e são compatíveis com os mesmos módulos e código.

O projeto Wiring, que deu origem ao projeto Arduino, possuía como objetivo o compartilhamento de ideias, experiências e conhecimento entre artistas e designers ao redor do mundo. O projeto Arduino, por sua vez, tem como objetivo, desde sua concepção, desenvolver uma plataforma educacional economicamente viável. A junção destas ideias resulta em uma característica fundamental do Arduino, que é a confluência de criatividade e técnica.

O arcabouço de desenvolvimento do projeto Arduino é distribuído livremente e a transferência do programa para o microcontrolador pode ser feita utilizando um cabo USB. Além disso, a existência de uma comunidade que dá suporte aliada a outros motivos estéticos, práticos, econômicos e técnicos, fizeram com que o Arduino se difundisse muito entre hobbystas e artistas[16,17]. Nos últimos anos, projetos utilizando o Arduino têm ocupado galerias, museus e diversos outros espaços, artísticos ou não (Gibb, 2010). Dada a grande disponibilidade de projetos compatíveis com Arduino (módulos de hardware e pedaços de código), a plataforma tem sido utilizada nos mais diversos contextos, como por exemplo para controlar os motores de impressoras 3D[18], compor novos instrumentos musicais [19,20,21], controlar aeromodelos[22], entre muitos outros usos.

Finalmente, é importante comentar que as pessoas que mantêm o projeto Arduino expressam uma preocupação com a desigualdade de condições de estudo e trabalho em diferentes partes do mundo. Um indivíduo ou empresa que queira utilizar o nome Arduino deve contribuir com o projeto de alguma forma: pagando taxas, liberando os projetos ou código desenvolvidos, documentando e dando suporte ao produto ou qualquer combinação dessas possibilidades. No sítio do projeto, é explicitado que uma parte da entrada de dinheiro é destinada a fomentar a computação em lugares onde os custos de hardware são proibitivos. Além disso, para utilizar a marca, é pedido ao fabricante que faça todo o esforço para ter o hardware montado sob condições de trabalho justas. Estes procedimentos, frutos de reflexão crítica sobre o papel e o impacto do projeto na sociedade, são importantes não só porque levam em conta o sistema de produção (de bens materiais e simbólicos) como um todo, mas também porque atribuem à ferramenta didática uma potencialidade de transformação da realidade.

Através da experimentação com captura, processamento e produção de sinais analógicos e digitais de áudio utilizando o Arduino, pretendemos quantificar a capacidade computacional e qualidade do processamento de sinais digitais em tempo real da plataforma. Como se trata de uma plataforma com baixo poder de processamento, grande parte dos projetos desenvolvidos com Arduino o utilizam como ferramenta de automação, usando suas entradas e saídas para captura de sinais de sensores e emissão de sinais de controle baseados em instruções computacionalmente simples. Apesar disto, é possível encontrar trabalhos específicos sobre processamento de sinais utilizando Arduino[23], e sua natureza lúdica reafirma o interesse na exploração da plataforma como ferramenta educacional e artística.

A plataforma

A plataforma desenvolvida pelo projeto Arduino consiste em um conjunto de hardware e software que, juntos, compõem uma interface simplificada para interação com um microcontrolador. Existem diversos projetos para placas de Arduino, mas todos possuem a mesma estrutura e as mesmas funcionalidades básicas. O hardware genérico é composto por um microcontrolador com

[16] http://www.arduino.cc/playground/Projects/ArduinoUser
[17] http://www.studiobricolage.org/arduino-1
[18] http://wiki.makerbot.com/thingomatic
[19] http://www.surek.co.uk/trampoline/
[20] http://xciba.de/pumpbeats/
[21] http://servoelectricguitar.com/
[22] http://aeroquad.com/
[23] http://interface.khm.de/index.php/lab/experiments/arduino-realtime-audio-processing/

memória flash programável e suporte a entradas e saídas analógicas e digitais. O software, por sua vez, é composto por um compilador e um ambiente de desenvolvimento integrado (IDE), desenvolvidos em Java[24] com bibliotecas em C, e um sistema de inicialização que roda no microcontrolador. O projeto da placa possui o mínimo necessário para alimentação e comunicação com o microcontrolador: reguladores, relógio de cristal, interface USB ou serial para conexão com um computador e uma interface de programação para substituir o sistema de inicialização.

Nas próximas seções, cada um dos componentes básicos de hardware e software do Arduino serão descritos com mais detalhes, para em seguida avaliar suas possibilidades e limitações de uso e estabelecer parâmetros de análise.

Uma interface com um microcontrolador

Um *microcontrolador* é um conjunto minimal de componentes para a execução de uma aplicação específica: processador, memória para armazenamento do programa (em geral flash), memória de acesso aleatório para armazenamento dos dados e interfaces de entrada e saída de dados. A diferença fundamental entre microcontroladores e microprocessadores (como, por exemplo, as unidades centrais de processamento dos computadores de mesa e notebooks) é que os microcontroladores possuem toda a estrutura necessária para a computação em um único chip, enquanto que microprocessadores utilizam memória de acesso aleatório externa ao chip para armazenamento de programa e dados. Além disso, microcontroladores possuem custo de produção mais baixo e menor consumo de energia pois, em geral, são menos flexíveis (em termos de aplicações), dispõem de menor capacidade computacional, além de serem desenvolvidos com tecnologias específicas, diferentes das do microprocessador[25].

A função básica do hardware do Arduino é disponibilizar uma interface mínima para a comunicação de um computador com um microcontrolador, e deste microcontrolador com outros módulos de hardware, sensores ou atuadores. Existem vários projetos diferentes de placas Arduino, que diferem principalmente no modelo de microcontrolador utilizado, interface de comunicação com o computador e disposição dos componentes na placa.

A escolha de uma marca e modelo de microcontrolador para um projeto deve levar em conta diversos fatores, como por exemplo a possibilidade de reprogramação (e a infraestrutura necessária para realizar este procedimento), as possibilidades de conexão com periféricos (USB, rede, módulos de PWM, de memória externa, etc), tensão de alimentação suficiente para controlar diretamente LEDs e outros periféricos, apresentação (em termos de necessidades relativas a manipulação, transporte, proteção, etc) e limites de memória para o programa e para os dados (muitas famílias de microcontroladores possuem limites muito pequenos, veja a tabela 1.1 com os limites dos modelos do Arduino).

O objetivo do projeto Arduino é, desde seu início, criar uma plataforma para desenvolvimento de projetos com microcontroladores que seja acessível (financeira e tecnologicamente) para estudantes, artistas, hobbystas e curiosos em geral. Esta ambição cria algumas necessidades em termos de funcionalidade da plataforma, o que por consequência gera restrições para a estrutura do microcontrolador a ser utilizado pelo projeto. Necessitava-se de um microcontrolador que fosse facilmente reprogramável a partir de uma interface disponível em qualquer computador pessoal e possuísse um bom balanço entre flexibilidade e padronização na interface com periféricos.

O projeto Arduino escolheu os microcontroladores da marca Atmel[26] da série megaAVR, mais especificamente os modelos ATmega168, ATmega328, ATmega1280 e ATmega2560. Os modelos diferem um do outro na frequência de operação, capacidade de memória (tanto para programa quanto para dados), e número de pinos de entrada e saída digitais e analógicas. Uma comparação entre as características de Arduinos que utilizam cada modelo de microcontrolador pode ser vista na tabela 1.1. Um mesmo modelo de microcontrolador é utilizado por vários modelos diferentes de

[24] http://www.oracle.com/technetwork/java/index.html
[25] http://www.engineersgarage.com/microcontroller
[26] http://www2.atmel.com/

Tabela 1.1: *Características dos processadores utilizados em diferentes modelos do Arduino*

	ATmega168	ATmega328	ATmega1280	ATmega2560
Frequência de operação	16 MHz	16 MHz	16 MHz	16 MHz
Memória para programa	16 KB	32 KB	128 KB	256 KB
Memória para dados	1 KB	2 KB	8 KB	8 KB
Entradas/saídas digitais	14	14	54	54
Saídas com PWM	6	6	14	14
Entradas analógicas	8	8	16	16

Arduino e a tabela mostra o valor máximo de cada característica encontrada entre os modelos que utilizam um mesmo microcontrolador.

A atualização do programa nos microcontroladores do Arduino é feita através da interface de desenvolvimento, que é executada em qualquer computador capaz de rodar aplicações Java e se comunica com a placa através de algum tipo de conexão padrão. Os primeiros projetos de Arduino possuíam interface serial com o computador, mas com o declínio da produção de placas-mãe com esta interface e a popularização do USB, os modelos mais novos do Arduino possuem interface USB. Também estão disponíveis projetos de módulos USB para a adaptação deste tipo de interface a modelos antigos ou para a inclusão de mais uma porta de comunicação nos modelos mais novos.

O projeto Arduino não teria tanto alcance se não incluísse uma interface de desenvolvimento que simplifica a escrita, compilação e carga do programa no microcontrolador. É possível obter, a partir do sítio oficial do projeto[27], o código fonte e versões compiladas para Windows, Mac OS X e GNU/Linux do software que unifica o processo de desenvolvimento para o Arduino em torno de apenas uma ferramenta. Com uma placa de Arduino conectada ao computador via USB e um ambiente Java configurado corretamente, é necessário apenas um clique para compilar o código e transferi-lo para a área do programa no microcontrolador. Através da abstração dos detalhes mais árduos da interação com o microcontrolador obtida pela utilização do mesmo software para programação, compilação e transferência do programa para a placa, o projeto atinge um grande nível de usabilidade. A plataforma se utiliza de um *bootloader* (um gerenciador de carregamento do sistema) gravado nas primeiras centenas de bytes do chip que permite que a substituição do programa seja feita sem a necessidade de hardware específico[28].

Extensões do Arduino

Um Arduino pode ser estendido através de placas que podem ser acopladas à placa principal. Estes módulos levam o nome de *shields*[29] (escudos) e compreendem tanto a integração de diversas tecnologias padrão como Ethernet, Xbee, cartões SD, sensores de temperatura, WiFi e GSM, como outras funcionalidades úteis como controle de motores[30], reprodução de MP3[31], ou mesmo um único escudo que disponibiliza acelerômetro, alto-falante, microfone, transmissor e receptor infravermelho, LED RGB, botões, potenciômetro e sensor de luz visível[32]. O sítio do Arduino aponta uma listagem extensa de escudos oficiais e não oficiais compatíveis com a plataforma[33].

Existem também muitas placas derivadas do Arduino, compatíveis apenas com o software. São clones genéricos, placas e interfaces para os mais diversos fins, desde implementações de pilotos automáticos para aeromodelos e tanques até versões que ocupam menos espaço ou podem ser

[27] http://arduino.cc/en/Main/Software
[28] http://arduino.cc/en/Tutorial/Bootloader
[29] http://arduino.cc/en/Main/ArduinoShields
[30] http://www.robotpower.com/products/MegaMoto_info.html
[31] http://www.roguerobotics.com/products/electronics/rmp3
[32] http://www.ruggedcircuits.com/html/gadget_shield.html
[33] http://shieldlist.org/

montadas em papel[34].

Deve-se observar que existe um balanço entre a técnica e infraestrutura disponível para a construção de uma placa, um escudo ou outros tipos de extensões e adaptações do Arduino e a qualidade do resultado do processamento, principalmente quando da captura e síntese de áudio. Variações de temperatura no ambiente ou ressonância com ondas de radiofrequência, por exemplo, podem interferir no resultado da conversão de sinais entre representações analógicas e digitais em montagens caseiras que não possuam cuidados especiais.

Entradas e saídas analógicas, frequência de operação e taxa de amostragem

Os microcontroladores da Atmel possuem algumas portas de entrada com conversores analógico-digitais com resolução de 10 bits (veja a tabela 1.1) que mapeiam voltagens entre dois valores de referência para inteiros entre 0 e 1023. A maioria dos modelos de microcontrolador do Arduino operam a uma taxa de 16 MHz, mas a leitura da entrada analógica utilizando a função da biblioteca demora cerca de 100 microssegundos, e portanto a taxa de amostragem de uma porta analógica nesse cenário atinge cerca de 10.000 Hz[35]. Esta restrição representa não só um limite do espectro de frequências representadas após a digitalização mas também estabelece a necessidade de utilização de um filtro externo no sinal de entrada para cortar frequências acima da taxa de Nyquist para que não haja aliasing (veja a Seção 2.2, mais adiante). Acessando-se o hardware diretamente é possível obter taxas de amostragem bem maiores, como será visto na Seção 3.2.2.

Com esta limitação na taxa de amostragem de um sinal de entrada, uma opção interessante para o processamento de áudio em tempo real é a reconfiguração dos relógios internos para diminuir a taxa de chamada da função de processamento e assim permitir um período maior de processamento entre a chegada de duas amostras consecutivas. Um projeto que implementa processamento de áudio em tempo real[36], por exemplo, realiza amostragem alternadamente em duas portas diferentes, uma com entrada de um sinal de áudio, e outra conectada a um sinal de controle. Para obter um tempo razoável de processamento, os relógios internos são configurados de forma que a amostragem de cada sinal (de áudio e de controle) é feito com uma taxa efetiva de 15 KHz. Supondo que os sinais de controle não precisam ser amostrados com taxa igual ao sinal de áudio e que a computação pode ser feita em blocos, é possível melhorar estes resultados.

Também é possível gerar saídas analógicas com resolução de 8 ou 16 bits através de circuitos PWM embutidos nos pinos de saída do processador. O sinal gerado por esta técnica pode ser utilizado para controle do nível de brilho de LEDs, controle de velocidade de motores e, com um pouco de boa vontade, geração de sinais de áudio analógicos. A frequência máxima obtida de um sinal PWM no Arduino utilizando-se a função de escrita analógica da biblioteca é 500 Hz[37], bastante baixa em relação ao espectro de frequências audíveis. Utilizando-se de algumas funções específicas do microcontrolador e fazendo a escrita diretamente na porta de saída, é possível obter frequências bem mais altas, como será descrito na Seção 3.2.3. Existem ainda escudos do Arduino tanto para gravar[38] quanto para reproduzir som com maior resolução e frequência (12 bits e 22 KHz, respectivamente, neste exemplo[39]). Assim, se a baixa fidelidade não é uma escolha estética, sempre há a possibilidade de extensão da plataforma utilizando hardware específico de forma a atender as necessidades do trabalho de cada artista.

Apesar disso, a opção deste trabalho é de utilizar o Arduino sem escudos, exatamente para determinar as possibilidades da plataforma em sua expressão mais básica. Este tipo de utilização e os resultados obtidos serão descritos no Capítulo 3.

[34] http://www.arduino.cc/playground/Main/SimilarBoards
[35] http://arduino.cc/en/Reference/AnalogRead
[36] http://interface.khm.de/index.php/lab/experiments/arduino-realtime-audio-processing/
[37] http://www.arduino.cc/en/Reference/AnalogWrite
[38] http://shieldlist.org/seeedstudio/music
[39] http://www.ladyada.net/make/waveshield/faq.html

1.2.2 Processadores gráficos: programação para GPUs utilizando CUDA

GPU é um acrônimo para **Graphics Processing Unit** (unidade de processamento de gráficos) e se trata de um tipo de circuito projetado especificamente para o processamento paralelo de imagens para a exibição na tela de um computador. O processamento de imagens, bi ou tridimensionais, possui uma série de características e necessidades específicas que foram determinantes para as escolhas feitas durante o desenvolvimento dos circuitos específicos para estes fins.

A computação paralela é fundamentada pela associação entre estruturas e operações matemáticas altamente paralelizáveis e os dispositivos capazes de realizar tais computações. O início do estudo formal e as primeiras propostas de circuitos ocorreram ainda na década de 1950 e resultaram tanto em propostas teóricas, como por exemplo a máquina universal sugerida por Holland (Holland, 1959), quanto na construção dos primeiros computadores paralelos, projetados pela IBM[40] (multiprogramação com apenas um processador[41]) e UNIVAC[42] (uso de dois processadores em paralelo), produzidos em 1959.

Nos anos seguintes, houve avanços importantes em pesquisas relativas a paralelismo e processamento de sinais, como por exemplo o desenvolvimento da *Transformada Rápida de Fourier* (Cooley e Tukey, 1965) em 1965 e a descrição, em 1966, da taxonomia utilizada até hoje para classificação das possíveis relações entre instruções e dados nos circuitos de computação paralela (Flynn, 1966), entre outros.

Naquele momento, toda infraestrutura computacional e pesquisas em andamento eram destinadas a aplicações científicas, militares e industriais e até o meio da década de 1960 os resultados de todas essas computações só podiam ser gravados em fita magnética, perfurados em cartões ou impressos em papel. Em 1967 surge o primeiro teletipo[43] que, através da simulação de um terminal de impressão, imprime os resultados em uma tela ao invés de imprimir no papel. Os primeiros computadores pessoais começaram a ser comercializados nos anos 1970 e em meados dos anos 1980 a indústria de hardware começou a produzir os primeiros modelos equipados com placas de vídeo, com instruções primitivas para auxiliar na produção de gráficos em duas dimensões[44,45,46].

No início da década de 1990, o desenvolvimento das placas de vídeo se alinha direta e definitivamente com as pesquisas na área da computação paralela. Ao longo de toda a década, intensifica-se o desenvolvimento de circuitos especializados para processamento de imagens tridimensionais com foco no mercado de usuários domésticos, impulsionado pelo mercado de jogos para computador. O modelo de *pipeline* desenvolvido para as placas de vídeo ao longo desta década apresenta, além de paralelismo de *dados*, paralelismo de *tarefas*. Os dados são divididos em blocos que são inseridos sequencialmente na pipeline e passam por uma série de estágios diferentes da computação (as tarefas). Nesta fila, a saída de um estágio é inserida diretamente na entrada do estágio seguinte. Na execução de cada tarefa sobre um bloco, a aplicação de uma mesma operação ocorre em paralelo em todos os dados do bloco. Além disso, a cada momento, toda a pipeline pode estar ocupada com blocos de dados em estágios diferentes da computação e todas estas tarefas são executadas também em paralelo (Owens *et al.*, 2008).

Algumas características são especialmente importantes para compreender a estrutura dos circuitos GPU atuais. Uma primeira característica é o alto requerimento computacional para o processamento de gráficos tridimensionais em tempo real: milhões de pixels têm que ser computados por segundo e a computação de cada pixel pode ser bastante complexa. Apesar deste alto requerimento computacional, existe também alto grau de paralelismo presente nas estruturas e operações matemáticas relacionadas. Uma transformação linear sobre um conjunto de vértices, por exemplo, é uma operação que pode ser realizada sobre vários elementos em paralelo. Outra característica ainda é a necessidade do resultado das computações ser recebido em tempo real, o que requer uma

[40] http://en.wikipedia.org/wiki/IBM_7030
[41] http://www.cs.clemson.edu/~mark/stretch.html
[42] http://en.wikipedia.org/wiki/UNIVAC_LARC
[43] http://en.wikipedia.org/wiki/Datapoint_3300
[44] http://tinyurl.com/3k6ek2x
[45] http://en.wikipedia.org/wiki/Commodore_Amiga#Graphics
[46] http://en.wikipedia.org/wiki/8514_%28display_standard%29

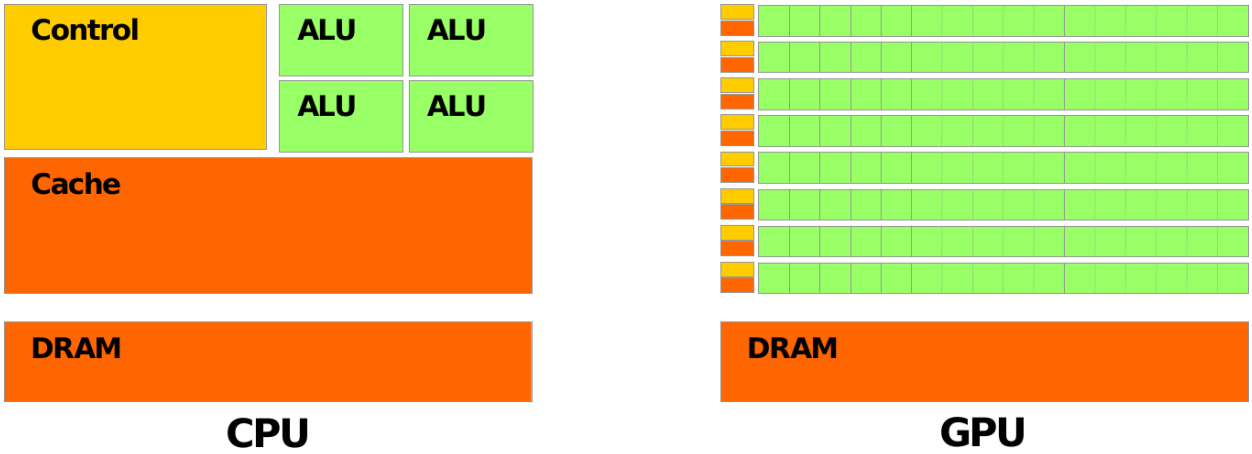

Figura 1.1: *Estrutura da CPU versus da GPU.*

alta taxa de fluxo de dados. Estas três características levaram o desenvolvimento dos circuitos de processamento de gráficos para um caminho bastante distinto do das CPUs (Owens *et al.*, 2008).

As CPUs são otimizadas para alto desempenho de código sequencial, com muitos transistores dedicados à obtenção de paralelismo no nível das instruções. Por outro lado, a natureza altamente paralelizável da computação gráfica permite à GPU utilizar os recursos adicionais diretamente para computação, atingindo intensidade aritmética mais alta com o mesmo número de transistores. Por causa de diferenças fundamentais entre as arquiteturas (veja a Figura 1.1), a velocidade de crescimento da capacidade computacional das placas gráficas é muito mais alta que a das CPUs (Owens *et al.*, 2007). Em Agosto de 2013, quando da elaboração deste texto, a melhor placa gráfica comercializada pela NVIDIA atingia pico de processamento de mais de 1 TFLOPS[47], enquanto que a melhor CPU Intel disponível não ultrapassa 100 GFLOPS[48].

De volta à história, em 1999 o termo GPU era utilizado pela primeira vez pelas fabricantes de placas de vídeo para designar um modelo de processamento paralelo e sua implementação em placas com circuitos especializados para o processamento de transformações lineares, cálculos de iluminação e renderização de gráficos tridimensionais.

No início da década de 2000, os primeiros modelos de GPU possuíam funções fixas em muitos de seus estágios de computação e os programadores tinham de gastar tempo desenvolvendo estratégias para adaptar a estrutura de seus problemas às possibilidades computacionais oferecidas pelo hardware. Ao longo da década, ocorreram grandes avanços nas pesquisas sobre programação de propósito geral com dispositivos gráficos, o que reafirmou o reconhecimento de sua aplicabilidade científica. Neste contexto, a arquitetura das GPUs foi sendo transformada, e desde então o desafio dos fabricantes tem sido atingir um bom balanço entre prover acesso de baixo nível ao hardware para obtenção de melhor desempenho e desenvolver ferramentas e linguagens de alto nível para permitir flexibilidade e produtividade na programação. Hoje o que está disponível são dispositivos com unidades completamente programáveis que suportam operações vetoriais de ponto flutuante, acompanhadas de linguagens de alto nível e arcabouços de programação que abstraem e facilitam ainda mais a tarefa de programação para GPU.

O domínio numérico da computação gráfica é prioritariamente o de ponto flutuante e por isso os primeiros modelos de GPU não possuíam, por exemplo, suporte a aritmética de inteiros e operações do tipo *bitwise*. Uma outra dificuldade das primeiras gerações era a inflexibilidade do acesso aleatório para escrita na memória, pois a posição final de cada vértice na tela, que corresponde ao endereço na memória onde cada pixel resultante será gravado, era definida num estágio inicial da computação e não podia ser alterada posteriormente. Estas questões foram superadas nos últimos anos através da implementação do suporte a diversos tipos de operação e pesquisa científica

[47] http://www.nvidia.com.br/object/workstation-solutions-tesla-br.html
[48] http://www.intel.com/support/processors/xeon/sb/CS-020863.htm

sobre formas de utilizar o hardware gráfico. Atualmente, já existem implementações de algoritmos criptográficos complexos que rodam inteiramente em GPU fazendo uso intenso de aritmética de inteiros[49], além de estudos sobre a eficiência de operações paralelas como *gather* e *scatter* utilizando GPUs (He *et al.*, 2007).

Já há alguns anos a maioria absoluta dos computadores pessoais novos (mais de 90% dos desktops e notebooks) e uma grande parte dos dispositivos móveis (PDAs, telefones celulares, etc) incluem, integrado à placa mãe, um processador que atua exclusivamente na aceleração do processamento de gráficos[50]. Esta alta disponibilidade da GPU nos computadores modernos tem permitido enorme avanço nas pesquisas e sua utilização como um dispositivo genérico de coprocessamento, não apenas para aceleração do processamento de imagens, mas para propósitos mais gerais. É possível utilizar a infraestrutura da GPU para cálculos diversos, utilizando paradigmas de computação paralela (como MIMD e SIMD) através do modelo de processamento de fluxos de dados, como será visto na seção 4.1.2 (Venkatasubramanian, 2003). Já não é novidade o uso da GPU para a construção de supercomputadores com milhares de processadores em paralelo a um custo acessível para utilização científica[51], e uma busca rápida revela uma quantidade muito grande de problemas mapeados para resolução nesta infraestrutura paralela.

Neste contexto, um novo termo aparece para designar a utilização da GPU para propósitos outros que não o processamento de vídeo para o qual a plataforma foi inicialmente desenvolvida. O termo **GPGPU**, acrônimo para *General Purpose computation on GPU*, faz referência a este tipo de utilização de circuitos GPU para propósitos gerais. No campo da Computação Musical, já é possível encontrar menções à utilização da GPU para, por exemplo, processamento de áudio tridimensional e auralização através de técnicas como HRTF[52] (Gallo e Tsingos, 2004).

Mais recentemente, outras soluções têm sido propostas com foco na integração entre as arquiteturas da GPU e CPU[53,54,55]. De qualquer forma, tanto as contribuições metodológicas trazidas ao longo do desenvolvimento da GPU quanto os arcabouços teóricos e computacionais desenvolvidos são de extrema valia para a computação paralela em geral. Além disso, as possibilidades computacionais trazidas pela GPU e sua curva de crescimento muito mais rápida do que a da CPU tradicional constituem ainda mais motivos para continuar o estudo deste tipo de plataforma.

Este trabalho descreve algumas possibilidades e limites do processamento de áudio digital em tempo real utilizando GPUs. Para isto, foram investigadas possibilidades de paralelismo de algoritmos de processamento de áudio e foram implementadas soluções utilizando os arcabouços disponíveis. Além disso, também foi feita a integração da GPU com um programa com licença livre bastante utilizado para processamento de áudio digital em tempo real chamado Pure Data[56]. Tudo isto será abordado em detalhes no Capítulo 4.

1.2.3 Dispositivos móveis: sistema operacional Android

Com a evolução da computação e da engenharia, estão à disposição dispositivos computacionais altamente complexos que cabem na palma da mão. **Android** é o nome de um sistema operacional para dispositivos móveis, desenvolvido a partir de 2003 por empresa homônima. Em 2005, sistema e empresa foram comprados pela Google Inc., multinacional americana que atua em diversas áreas tecnológicas que formam base para sua atividade principal em termos de receita: o mercado de publicidade eletrônica.

O sistema operacional Android é composto por um kernel fortemente baseado no kernel do Linux[57], um conjunto de drivers para muitos modelos de aparelhos, uma série de programas utilitários

[49] http://http.developer.nvidia.com/GPUGems3/gpugems_ch36.html
[50] http://computershopper.com/feature/the-right-gpu-for-you
[51] http://fastra.ua.ac.be/
[52] *Head Related Transfer Functions*
[53] http://www.nvidia.com/object/pr_nexus_093009.html
[54] http://www.amd.com/us/Documents/49282_G-Series_platform_brief.pdf
[55] http://software.intel.com/file/37300
[56] http://puredata.info/
[57] http://kernel.org/

para o gerenciamento do sistema operacional e uma vasta gama de aplicações para o usuário final (Hall e Anderson, 2009). Apesar de se tratar de um conjunto de programas que compõem um sistema operacional completo, o nome Android é também comumente utilizado para fazer referência aos dispositivos móveis distribuídos com este sistema operacional.

O projeto Android baseia seus aplicativos na linguagem Java, o que permite a utilização do mesmo código binário em diferentes arquiteturas. Também disponibiliza bibliotecas para interface com funcionalidades que estão se tornando cada vez mais comuns em dispositivos móveis, como processamento gráfico otimizado, conexão via bluetooth, 3G, GSM, WiFi, câmera, GPS, bússola e acelerômetro.

A licença principal utilizada no projeto Android é a *Apache Software License 2.0*[58], considerada pela Free Software Foundation[59] como uma licença de software livre[60] compatível com a versão 3 da GPL[61]. Apesar disto, outras licenças estão presentes no projeto, como é caso dos patches do kernel do Linux, que têm obrigatoriamente que ser licenciados sob a GPL versão 2.0[62].

Discussões recentes têm levantado preocupações quanto à possível infração de alguns termos de licenças livres na utilização de código do kernel do Linux no sistema operacional Android. Alguns exemplos são a negação por parte da empresa Google da distribuição do código fonte do seu produto até o lançamento de versões mais novas do sistema[63,64] (inclusive colocando em cheque a "liberdade" da licença Apache como um todo) e a possível infração ao importar partes de arquivos de cabeçalho do kernel do Linux e licenciar o código derivado com licenças não previstas pela GPL[65].

Outros sistemas operacionais têm sido desenvolvidos como ramificações do Android, com o objetivo de priorizar outras questões. O projeto Replicant[66], por exemplo, é uma distribuição do sistema operacional Android com 100% do código publicado sob licenças livres. O sistema Cyanogenmod[67] é uma variante que se propõe a aumentar o desempenho e confiabilidade no sistema. Também é possível instalar o sistema Debian Linux em dispositivos que suportam o Android[68], ou modificar o sistema para utilizar soluções de segurança como criptografia de disco[69], por exemplo.

Ao longo deste trabalho, sempre que se falar em Android se estará referindo a todas as plataformas que compreendem o ecossistema complexo descrito nesta seção. As técnicas e metodologias desenvolvidas não dependem de um tipo de instalação específica, dos modelos dos dispositivos móveis ou do sistema operacional em si. Ao contrário, a ideia é analisar as possibilidades de uso de plataformas móveis em geral como dispositivos para processamento de áudio e contemplar sistemas nos quais seja possível capturar áudio, receber sinais de outros sensores e aparelhos, executar código desenvolvido pelo usuário, e retornar o resultado do processamento destes sinais através de saídas de áudio ou conexões do tipo WiFi, bluetooth, infravermelho ou quaisquer outras que estejam disponíveis. As implementações e resultados obtidos nos sistemas Android será o assunto do Capítulo 5.

1.3 Trabalhos relacionados

Nesta seção, serão revisadas algumas referências teóricas relevantes para os estudos realizados neste trabalho. Primeiro são feitos alguns comentários sobre a evolução do processamento de sinais em tempo real. Em seguida, será abordada a investigação de limites inferiores para a realização de

[58] http://www.apache.org/licenses/LICENSE-2.0
[59] http://www.fsf.org/
[60] http://www.fsf.org/about/what-is-free-software
[61] http://www.gnu.org/licenses/license-list.html,
[62] http://www.gnu.org/licenses/old-licenses/gpl-2.0.html
[63] http://www.theregister.co.uk/2011/05/10/android_ice_cream_sandwich/
[64] http://www.itworld.com/open-source/164153/how-google-can-delay-android-source-code-releases
[65] http://ebb.org/bkuhn/blog/2011/03/18/bionic-debate.html
[66] http://replicant.us/about/
[67] http://www.cyanogenmod.org/about
[68] http://lanrat.com/android/debian
[69] https://github.com/guardianproject/LUKS/wiki

processamento de sinais digitais em tempo real, que é interessante para estabelecer um patamar de análise para o Arduino enquanto ferramenta dotada de baixo poder computacional. Em seguida, será analisada a evolução de técnicas paralelas para processamento digital em geral para fundamentar o estudo da plataforma GPGPU. Por fim, são levantadas algumas possibilidades de uso de aparelhos Android para o processamento em tempo real.

Os três capítulos seguintes deste texto terão uma estrutura parecida, pois tratam de problemas parecidos porém não idênticos. Cada capítulo contará a metodologia utilizada para abordar uma das plataformas de estudo, e em seguida apresentará os resultados obtidos naquela plataforma. Este é o preço que se pagará neste trabalho por uma abordagem tríplice, e é o preço que o leitor pagará se se propuser a ler o texto do começo ao fim. Apesar disso, em sendo as plataformas e portanto os assuntos tão distintos, espera-se que o leitor possa encontrar novidades instigantes a cada capítulo, assim como eu encontrei.

1.3.1 Evolução e exemplos de processamento de sinais em tempo real

A evolução dos circuitos especializados em processamento de sinais digitais se deu, em geral, através da inclusão de características que facilitam a computação dos algoritmos de processamento de sinais digitais. Um exemplo é a inclusão do produto interno como operação básica, o que permite a computação de filtros de resposta impulsiva finita de forma muito mais rápida. Outros fatores fundamentais para a qualidade e velocidade dos circuitos especializados para processamento de sinais digitais como os que estão disponíveis hoje são a associação de múltiplas unidades de execução (como por exemplo a possibilidade da execução do cálculo do produto interno em paralelo com o processamento lógico e aritmético), o esforço para atingir eficiência no acesso à memória, o desenvolvimento de formatos de dados específicos para obter fidelidade numérica, a utilização de estágios sequenciais de processamento (*pipelines*) e o desenvolvimento de conjuntos de instruções especializados (Eyre e Bier, 2000).

Enquanto a produção de hardware segue linhas industriais, a produção de software para processamento de áudio em tempo real pode ser dividida em duas frentes: técnica e experimental. De um lado, podem ser encontrados programas de propósito geral que implementam modelos computacionais para processamento em tempo real, como CSound (Vercoe e Ellis, 1990), Pure Data (Puckette, 1996) e outros. Do outro lado, encontram-se as produções artísticas que expressam concepções estéticas específicas, explorando ideias de acompanhamento ao vivo e interatividade na apresentação artística, como por exemplo os sistemas Voyager (Lewis, 2000) e Cypher (Rowe , 1992b), já citados anteriormente.

Limites inferiores para o processamento de sinais digitais em tempo real

Ao lidar com processamento de sinais digitais em tempo real, uma questão fundamental que surge é sobre o poder computacional necessário para realizar com efetividade o processamento em questão, de forma a obter os resultados dentro de um prazo razoável. Para isto, é necessário quantificar a velocidade e o desempenho do hardware utilizado e a relação destes com as necessidades de cada tipo de processamento proposto.

Por meio da representação esquemática de filtros digitais através de grafos dirigidos e associando a cada vértice um custo relacionado ao tempo de processamento de cada tipo de operação realizada, é possível obter transformações do grafo que mantém a equivalência funcional do filtro e assim determinar determinar o período mínimo (ou a frequência máxima) de amostragem associado a uma certa classe de filtros digitais (Renfors e Neuvo, 1981). Neste sentido, também pode-se seguir o caminho inverso e perguntar qual é o poder computacional mínimo para permitir a aplicação de um certo filtro a um sinal digital, se a taxa de amostragem é dada. Em outras palavras, se se deseja uma taxa de amostragem fixa, qual é o máximo de tempo que uma certa plataforma pode gastar em cada operação para que o cálculo seja realizado sem diminuir a taxa de geração de amostras? E, se a taxa de geração de amostras for prejudicada, quais são os prejuízos sonoros levando-se em conta os detalhes, por exemplo, do tipo de conversão digital-analógica utilizada? Esta abordagem

permite determinar, para uma plataforma com uma dada capacidade computacional, os tipos de processamento de sinais digitais que podem ser implementados para utilização em tempo real.

Além de estudar o custo computacional associado a certas classes de filtros digitais, também existem iniciativas no sentido de projetá-los tendo como objetivo obter filtros com baixo custo computacional. É possível encontrar desde trabalhos que fazem uso de técnicas de inteligência artificial e teoria dos grafos para desenvolver filtros de resposta impulsiva finita de baixa complexidade (Redmill e Bull, 1997), passando pelo desenvolvimento de filtros baseados em subconvoluções paralelas (Gray, 2003), até trabalhos que comparam diferentes formas de implementação de paralelismo no cálculo de filtros avaliando o desempenho de cada uma (Deepak *et al.*, 2007).

Conectividade, sensores e interatividade em apresentações artísticas

A realização de processamento de áudio em tempo real em plataformas móveis com capacidade de interconexão abre muitas possibilidades de colaboração para produção artística. A conectividade, aliada à abundância de tipos de sensores diferentes (acelerômetro, giroscópio, luz, campo magnético, orientação, pressão, proximidade, temperatura, entre outros) resulta numa ferramenta bastante rica para exploração e experimentação.

Assim, o processamento de áudio em dispositivos móveis pode ser compreendido neste contexto como uma ferramenta tecnológica para produção artística dotada de capacidade de interconexão com outros dispositivos e interação com o ambiente através de sensores e atuadores. A flexibilidade trazida por plataformas como o Android nas quais é possível desenvolver programas utilizando linguagens de alto nível aproxima os artistas em potencial das possibilidades trazidas pelo instrumento.

No contexto artístico, é interessante compreender conectividade e mobilidade não só por suas características desejáveis, como a possibilidade de comunicação instantânea entre múltiplos pontos dispersos no espaço, mas também através das limitações inerentes a estas características. Se uma volta em torno da terra à velocidade da luz demora cerca de 130 milissegundos, então a latência na comunicação planetária é inevitável. Nesse sentido, encontramos estudos que, ao desconstruir a ideia da conectividade como solução tecnológica universal para todos os problemas, sugere que a latência e outras características normalmente compreendidas como indesejáveis na comunicação podem ser aproveitadas como material estético (Shroeder *et al.*, 2007).

1.3.2 Paralelismo no processamento de sinais digitais

Muitas técnicas têm utilizado paralelismo no processamento de sinais, sempre explorando características paralelas inerentes a alguns tipos de estruturas de dados, equações e algoritmos. Dentre as plataformas escolhidas para análise, a GPU é a que desperta maior interesse em relação ao estudo do paralelismo no processamento de sinais, pois trata-se essencialmente de um processador paralelo. Dispositivos móveis mais recentes, inclusive alguns capazes de suportar o sistema operacional Android, também possuem circuitos do tipo GPU, apesar de que com capacidade computacional reduzida devido à necessidade de economia de energia.

Nesta seção, são descritos alguns estudos sobre paralelismo no processamento de sinais, com o objetivo de desenvolver um ferramental que ajudará a analisar as possibilidades que uma plataforma de processamento paralelo pode trazer para o processamento de sinais digitais em tempo real.

Paralelismo na transformada de Fourier

No início do século XIX, Jean Baptiste Joseph Fourier sugeriu que a solução para a equação do calor em um meio sólido poderia ser expressa como uma combinação linear de soluções harmônicas (Fourier, 1807). Mais de vinte anos depois, foi provado que a mesma ideia vale para uma classe mais geral de sinais (Dirichlet, 1829). Desses trabalhos surgiu a "Transformada de Fourier", uma operação matemática inversível que decompõe um sinal em suas componentes de frequência.

Até a metade do século XX, os conceitos desenvolvidos a partir da transformada de Fourier já haviam ganhado aplicação nas mais diversas áreas do conhecimento científico, como por exemplo

cálculo da periodicidade da orientação do spin de certos cristais, monitoramento sísmico remoto (no contexto de facilitar acordos sobre banimento de testes nucleares após a Segunda Guerra Mundial), melhoramentos da capacidade de detecção acústica de submarinos a distância, filtros digitais, processamento de fala, música e imagens, análise espectral de dados de interferômetro para determinação do espectro infravermelho de planetas, estudos na área de oceanografia, entre muitas outras (Cooley, 1987). Este cenário propiciou o desenvolvimento de muitas variantes do algoritmo original de cálculo dos coeficientes da transformada. O algoritmo ingênuo consome tempo proporcional a N^2 se N é o tamanho da entrada, e o amadurecimento desta linha de pesquisa deu origem em 1965 à `FFT` (Fast Fourier Transform), a transformada "rápida" de Fourier, que diminui o consumo de tempo para algo proporcional a $N.\log(N)$ (Cooley e Tukey, 1965).

Mais de 10 anos depois foi proposta uma adaptação da `FFT` para processamento paralelo em uma máquina desenvolvida especialmente para este propósito, de forma que a operação seja dividida em níveis e cada nível possua uma quantidade pequena de operações elementares que possam ser realizadas ao mesmo tempo (Pease, 1968). O resultado é um algoritmo que realiza três tipos de operações paralelas (a diferença entre pares de elementos adjacentes da entrada, uma permutação "ideal" das linhas de uma matriz, e a habilidade de multiplicar todos os elementos da entrada por um mesmo fator) e cuja execução é da ordem de duas vezes mais rápida do que a da `FFT` tradicional.

A manipulação do espectro de um sinal digital pode ser feita utilizando-se a Transformada de Fourier para trabalhar diretamente no domínio da frequência, ou através do cálculo da convolução do sinal digital com a resposta impulsiva de um certo filtro no domínio do tempo. O estudo do cálculo da convolução levou a uma outra abordagem bastante comum no processamento digital de sinais, que é a divisão do sinal de entrada em blocos de tamanho fixo e o desenvolvimento de algoritmos para realização de cálculos em blocos (Oppenheim *et al.*, 1999).

Durante a década de 60 e início dos anos 70, muita discussão foi feita em torno da eficiência e estabilidade de algoritmos de cálculo de convolução em blocos. Esta discussão culminou no desenvolvimento de técnicas para a aplicação de filtros recursivos utilizando operações em blocos, que permitem a execução em paralelo do processamento de cada bloco (Burrus, 1972).

A seguir, será visto como o paralelismo se desenvolve no contexto do hardware específico para processamento de sinais digitais.

Circuitos digitais para processamento paralelo

Como visto na seção 1.3.1, o desenvolvimento dos projetos de processadores de sinais digitais tem sido motivado pelas características dos algoritmos disponíveis para realizar o trabalho. Com a implementação de paralelismo nas arquiteturas ocorre o mesmo. Existem diversos tipos de paralelismo possíveis de serem explorados em um sistema como, por exemplo, paralelismo de dados, de instruções ou de tarefas. A combinação do nível de exploração de cada tipo de paralelismo deve levar em conta um balanço entre rapidez de hardware, flexibilidade de software e domínio de aplicação. Essa abordagem tem dado origem a diversas implementações distintas de circuitos paralelos para processamento de sinais em tempo real (Sernec *et al.*, 2000).

Como será visto mais adiante na seção 4.1.2, a estrutura do processamento de sinais é convenientemente capturada pelo modelo de processamento de fluxos de dados. Este modelo surge para satisfazer a necessidade de especificação formal do paralelismo inerente a soluções algorítmicas para problemas em diversos domínios, tais como mineração de dados, processamento de imagens, simulações físicas, e muitos outros (Owens *et al.*, 2007). O processamento gráfico não só se encaixa neste modelo mas também é influenciado por ele. O fato de que o desenvolvimento das arquiteturas tem sido fortemente influenciado pelo modelo de processamento de fluxos de dados pode ser observado pelas escolhas estratégicas recentes das fabricantes de placas de vídeo. As arquiteturas mais novas mostram uma tendência a unificar as unidades programáveis, em direção a um modelo menos específico do que somente para processamento de imagens tridimensionais. A ideia do processamento de fluxos de dados é complementada pelo desenvolvimento de arcabouços que implementam este modelo, como será visto na seção 4.1.3.

1.3.3 Processamento de sinais nas plataformas abordadas

Como foi visto na seção 1.2, a escolha das plataformas para este estudo foi feita tendo em mente algumas características em comum, como alta disponibilidade e (relativo) baixo custo, mas também considerando as particularidades de cada uma em termos de tecnologia computacional, bem como a riqueza de possibilidades de exploração resultante dessas particularidades. Talvez pela diferença de idade entre as três opções levantadas (20 anos de desenvolvimento de placas de vídeo aceleradas contra pouco mais de 6 anos dos projetos Arduino e Android), ou talvez por assimetrias no interesse acadêmico sobre cada uma, a impressão é de que há muito mais material disponível sobre GPGPU do que sobre as outras duas plataformas.

Enquanto a pesquisa em Arduino parece interessante no sentido lúdico, didático, experimental e artístico, para o Android já é possível encontrar uma gama maior de aplicações de processamento de sinais digitais. Já o processamento paralelo na GPU é um fato e uma tendência, tanto no âmbito acadêmico quanto no de mercado. Essas diferenças se refletem neste texto na medida em que o tema GPGPU ganha mais espaço por sua complexidade, idade, interesse científico e consequente abundância de material.

Nas próximas seções, veremos alguns estudos que abordam soluções relacionadas às propostas deste trabalho.

Processamento de sinais no Arduino

O Laboratório para Ciência da Computação Experimental da Academia de Artes Midiáticas de Colônia[70] possui um estudo (não publicado em periódico) sobre a utilização do Arduino para processamento de sinais em tempo real[71]. Pequenas adaptações do hardware são feitas para possibilitar a entrada e saída de sinais analógicos utilizando as conversões ADC e DAC disponíveis no microcontrolador do Arduino e alguns tipos de filtro são implementados. Este estudo é um bom ponto de partida para implementações como as que estamos propondo neste trabalho.

Outra iniciativa interessante é a da utilização do Arduino como placa de som, através da implementação de um driver para Linux utilizando a infraestrutura do projeto ALSA[72] (Dimitrov e Serafin , 2011b). O resultado fica aquém das placas de som comerciais, principalmente com relação à resolução dos sinais capturados e emitidos (8 bits no Arduino contra 32 bits de placas comerciais), mas o trabalho abre caminho para uma implementação completa e funcional de uma placa de som com licença aberta. Um estudo subsequente analisa questões relacionadas à entrada e saída de áudio em placas de som e propõe, como complementação do estudo anterior, um projeto de uma placa complementar para o Arduino que cuidaria exclusivamente da entrada e saída de sinais (Dimitrov e Serafin, 2011a).

Processamento de sinais em hardware gráfico

Uma das técnicas básicas de realização de processamento de sinais digitais é a Transformada Rápida de Fourier (*Fast Fourier Transform* – `FFT`) da qual falaremos com mais detalhes na seção 1.3.2. É possível encontrar estudos discutindo detalhes da implementação da `FFT` em placas gráficas do tipo GPU (Moreland e Angel, 2003), incluindo comparações com implementações altamente otimizadas para a CPU tradicional, como a biblioteca `FFTW`[73], escrita em C e publicada sob licença livre. Um outro estudo implementa a Transformada Discreta do Cosseno (*Discrete Cosine Transform* – `DCT`), outra técnica de análise e representação de sinais digitais, e deriva uma relação para o desempenho ótima na divisão de operações entre GPU e CPU (Mohanty, 2009). Ainda neste caminho, encontramos também a implementação para GPU da Transformada Discreta de Wavelets (*Discrete Wavelet Transform* – `DWT`), mais uma ferramenta para análise de sinais digitais (Wong *et al.*, 2007). Na direção de utilização da GPU como dispositivo para processamento

[70] http://interface.khm.de/
[71] http://interface.khm.de/index.php/lab/experiments/arduino-realtime-audio-processing/
[72] http://www.alsa-project.org/
[73] http://www.fftw.org/

de áudio, outro estudo avalia possibilidades de espacialização de áudio em tempo real, aplicando HRTFs com ajuda de funções nativas de reamostragem de texturas (Gallo e Tsingos, 2004).

No sentido de utilizar a GPU como dispositivo para processamento de propósito geral, ou seja, para outros tipos de computação que não somente o processamento de gráficos tridimensionais, é possível encontrar estudos em diversas direções. Desde simples multiplicações de matrizes e implementações de solucionadores do problema de decidir a linguagem `3-SAT` (Thompson *et al.* , 2002), passando por análises de imagens médicas e simulações físicas e biológicas (Owens *et al.* , 2008, 2007), até soluções para recuperação de informação (Govindaraju *et al.*, 2005), são muitas as iniciativas de adaptação de problemas para soluções utilizando GPU.

Processamento de sinais em dispositivos móveis

Com o objetivo de trazer para os dispositivos móveis uma ferramenta muito utilizada para o processamento de sinais digitais em computadores pessoais, o Grupo de Tecnologia Musical da Universidade Pompeu Fabra de Barcelona[74] realizou uma adaptação do Pure Data para a arquitetura PocketPC (Geiger, 2003). Analisando a qualidade sonora, a capacidade computacional e o desempenho do sistema na arquitetura escolhida, este estudo abre caminho para a utilização de sistemas móveis no processamento em tempo real com flexibilidade comparável à dos computadores pessoais.

Um outro estudo aborda uma aplicação bastante distinta das propostas neste texto, mas utiliza técnicas que são de interesse para nosso trabalho. Visando a implementação de sistemas de vigilância em dispositivos móveis, um estudo do Departamento de Engenharia da Informação da Universidade de Modena e Reggio Emilia na Itália discute as possibilidades de processamento de imagens e transmissão de sinais via internet utilizando dispositivos móveis (Cucchiara e Gualdi, 2010).

A discussão sobre as ferramentas, conceitos e atividades envolvidas na criação de música levou ao desenvolvimento de um aplicativo de mixagem de sons específico para a plataforma Android, chamado mixDroid (Flores *et al.*, 2010). Neste trabalho, o aplicativo serve como prova de conceito sobre associações entre conceitos da área de Ciências Humanas e a prática de fazer música com auxílio de ferramentas computacionais.

Otimizações no roteamento do sinal de áudio entre as camadas de mídia do sistema Android foram propostas com o objetivo de diminuir o uso de energia e economizar bateria (Pathak, 2011). Ao criar um gerenciador de rota de sinal e usar funcionalidades de compressão de áudio em hardware e software, os autores daquele trabalho conseguiam reduzir em 20% o consumo de energia para algumas tarefas específicas de processamento de áudio, quando comparado com o sistema de roteamento padrão da versão do Android considerada. O mergulho nas entranhas do sistema é diferente da abordagem do presente trabalho, que deseja obter indicadores de desempenho no nível da aplicação.

Esforços recentes têm feito a união de programas de processamento em tempo real tradicionais, como Csound (YI e LAZZARINI, 2012) e Pure Data (Brinkmann, 2012). Ambos os trabalhos fazem uso da JNI[75] para misturar código em C/C++ com código em Java. Sua abordagem é diferente daquela feita aqui, na qual é utilizado somente código em Java, para implementar uma interface de usuário e um modelo de DSP minimais, e o uso de código nativo representa um passo adiante no desenvolvimento e medição de desempenho.

O uso de código nativo não implica automaticamente em melhor desempenho, pois aumenta a complexidade da aplicação e possui um custo associado a chamadas ao código nativo. Existem trabalhos que têm como objetivo encontrar diferenças de desempenho entre Java e código nativo em diferentes cenários (Lin *et al.*, 2011). Mesmo assim, para o processamento de sinais em tempo real não foi possível encontrar uma implementação e comparação, e este passo pode ser considerado um trabalho futuro de bastante interesse.

[74] http://www.mtg.upf.edu/
[75] https://developer.android.com/sdk/ndk/overview.html

Capítulo 2

Metodologia e fundamentação teórica

O objetivo deste trabalho é obter informações sobre o desempenho das plataformas computacionais apresentadas na Seção 1.2, quando utilizadas para o processamento de áudio em tempo real. Por causa de diferenças de propósito, projeto e arquitetura entre as plataformas escolhidas para análise, é difícil (senão impossível e talvez até irrelevante) estabelecer uma comparação puramente quantitativa entre as três. O Arduino, por exemplo, desperta interesse por suas limitações de velocidade e memória que o permitem operar com consumo de energia bastante baixo, enquanto que a GPU tem como objetivo fundamental a aceleração da computação através da utilização de paralelismo, além de dispor de diversos níveis de memória, ao custo de consumir centenas de Watts de energia. Já a mobilidade e possibilidade de comunicação com outros dispositivos são características fundamentais do Arduino e dos dispositivos Android, mas não da GPU. Por estes motivos, é necessário desenvolver uma metodologia geral de análise de desempenho que possa ser usada nas três plataformas mas que deixe espaço para que as particularidades de cada uma sejam levadas em consideração.

Como é possível, então, avaliar o desempenho de uma ferramenta de forma mais abstrata que traga resultados esclarecedores sobre o uso no processamento de áudio em tempo real em plataformas com características tão distintas? Este trabalho pretende ajudar a esclarecer esta questão através da investigação, em cada plataforma, das possibilidades de implementação e da observação de métricas de desempenho de uma série de algoritmos frequentemente utilizados em rotinas mais complexas de processamento de áudio.

A primeira seção deste capítulo apresenta a noção de desempenho utilizada ao longo do trabalho, introduz rapidamente os algoritmos e métricas propostos, e investiga algumas diferenças entre as plataformas analisadas que influenciarão a abordagem e as implementações em cada uma. A seção seguinte apresenta com mais detalhes os algoritmos de processamento de áudio implementados e descreve como cada um pode ser usado como ferramenta genérica para a análise de desempenho. Toda esta discussão fornecerá a base para os capítulos seguintes, nos quais cada plataforma será abordada de forma a contemplar suas especificidades, utilizando diferentes subconjuntos e implementações dos algoritmos apresentado neste capítulo.

2.1 Análise de desempenho em diferentes plataformas computacionais

A análise de desempenho de uma certa plataforma computacional pode ser compreendida como a medição da quantidade de operações que podem ser realizadas por tal ferramenta por unidade de tempo. Num nível mais baixo, seria possível definir **desempenho** como, por exemplo, o número máximo de operações de ponto flutuante por segundo. Por outro lado, se a ferramenta computacional em questão não possuir uma implementação de ponto flutuante em hardware, pode ser mais interessante avaliar outros tipos de operação, como aritmética de inteiros ou operações sobre bits.

Pelo fato de poderem ser executados sobre blocos de amostras (veja a Seção 1.1.1), a complexidade dos algoritmos de processamento de áudio sempre pode ser descrita como uma função de pelo

menos um parâmetro relevante para as computações envolvidas, como será visto na Seção 2.1.1. O tempo que uma plataforma demora para calcular um algoritmo para um certo valor de parâmetro e o valor máximo de um parâmetro para o qual um algoritmo é viável em tempo real são informações que, do ponto de vista do usuário interessado em processamento sonoro, podem descrever melhor o desempenho da plataforma ao executar aquele algoritmo do que, por exemplo, uma medida da quantidade de operações de ponto flutuante executadas por unidade de tempo.

As três plataformas de processamento de interesse deste trabalho operam em três frentes computacionais bastante distintas, como foi visto na Seção 1.2. Sendo plataformas com características e possibilidades tão distintas, também são distintas as formas de programação e os tipos de problema que podem ser resolvidos em cada uma delas. É claro que, por serem dispositivos Turing completos, todos podem, em princípio, resolver qualquer problema para o qual se possa escrever um algoritmo. Apesar disso, cada uma conta com tamanhos diferentes de memórias com diferentes velocidades de acesso e diferentes conjuntos de operações básicas. Por isso, apesar da proposta deste trabalho ser implementar os mesmos algoritmos para avaliar as três plataformas, em cada uma delas é necessário utilizar uma abordagem um pouco diferente.

2.1.1 Algoritmos e métricas para análise de desempenho

Neste trabalho, a análise do desempenho das plataformas computacionais apresentadas na Seção 1.2 é feita através da implementação de uma série de algoritmos frequentemente utilizados para o processamento de áudio em tempo real. A ideia básica é, através da implementação, execução e medição do tempo gasto para diversas instâncias destes algoritmos, fornecer meios de comparação entre os diferentes dispositivos considerados. Para isto, foram escolhidos três algoritmos básicos, significativos por sua presença frequente em rotinas comuns de processamento de áudio, e um algoritmo mais complexo que resulta da composição de dois dos algoritmos mais básicos.

A **FFT**, apelido da *Transformada Rápida de Fourier*, é um algoritmo bastante utilizado para viabilizar o acesso à informação sobre as componentes de frequências presentes em sinais digitais. Ela opera sobre blocos de amostras e fornece uma descrição espectral suficiente para reconstruir o sinal original. Sua origem e implementação serão descritos na Seção 2.2.1. A operação de **convolução** e suas implementações algorítmicas são bastante utilizadas no processamento em tempo-frequência, como por exemplo na implementação de filtros de resposta impulsiva finita (filtros FIR). Seus efeitos no domínio do tempo e das frequências e suas duas formas, circular e linear, serão descritas na Seção 2.2.2. A **síntese aditiva**, por sua vez, é uma técnica de construção de um sinal a partir de conjuntos de osciladores mais básicos que podem ser implementados utilizando a função seno ou uma tabela contendo um período amostrado de uma função periódica qualquer. Detalhes destas implementações serão descritas na seção 2.2.3. Por fim, o ***Phase Vocoder*** é um algoritmo que permite a representação de um sinal arbitrário através de parâmetros para um conjunto de osciladores que podem ser manipulados para obtenção de efeitos diversos. Uma das formas de implementar o *Phase Vocoder* é através da análise via FFT e ressíntese via síntese aditiva, e será descrita na Seção 2.2.4.

Cada algoritmo mencionado acima possui um parâmetro em sua expressão matemática que influencia a complexidade computacional do cálculo de uma amostra: o tamanho do bloco de amostras no caso da FFT, a ordem do filtro FIR no caso da convolução no domínio do tempo e o número de osciladores no caso da síntese aditiva. Para valores fixos de cada um destes parâmetros, a complexidade computacional do cálculo de um bloco de amostras depende apenas do tamanho do bloco. Mas se considerarmos cada algoritmo dividido em *instâncias*, sendo que cada instância representa um valor distinto para os parâmetros descritos, então é possível formular a seguinte questão: qual é a maior instância de certo algoritmo que pode ser computada em tempo real?

Medição do tempo médio de execução de uma instância de um algoritmo sobre um bloco de amostras

Todo algoritmo de processamento de áudio que trabalha sobre um conjunto de amostras tem sua complexidade computacional parametrizada pelo número de amostras consideradas. O período teórico do ciclo DSP no processamento em tempo real é determinado pelo tamanho do bloco de amostras considerado e pela taxa de amostragem do sistema, e representa um limite para o tempo de execução do processamento do bloco (veja a Seção 1.1.1). Nesse sentido, dada uma certa instância de um algoritmo, a comparação do tempo de execução do algoritmo sobre um bloco de amostras com o período teórico do ciclo DSP permite decidir se aquela instância é viável em tempo real no dispositivo considerado ou não.

A medição do tempo de execução pode ser realizada através da marcação dos instantes de início e término do algoritmo. Dependendo da infraestrutura utilizada para o processamento, pode ser interessante incluir nesta medição outras tarefas comuns inerentes a sistemas de processamento de áudio em tempo real, como por exemplo o tempo de leitura e escrita de amostras no hardware de captura e emissão de áudio, tempo de conversão de tipo de representação numérica das amostras (PCM para ponto flutuante e vice-versa), tempo de transferência de memória (caso a memória do dispositivo seja separada da memória principal), entre outros.

Instância máxima de um algoritmo viável para um certo tamanho de bloco de amostras

A segunda métrica relevante neste contexto é a instância máxima viável de um certo algoritmo para um certo tamanho de bloco de amostras. Em outras palavras, o que se quer determinar neste caso são os valores máximos dos parâmetros para os quais cada algoritmo pode ser executado em tempo real, dada uma certa taxa de amostragem, em uma determinada plataforma. Este valor pode ser obtido através da execução de cada algoritmo repetidas vezes, cada vez utilizando valores maiores para os parâmetros que influenciam sua complexidade. Através da marcação do tempo necessário para execução de cada uma das instâncias e comparação de cada resultado com o valor teórico do período do ciclo DSP é possível determinar se o parâmetro ainda pode ser incrementado ou se a instância máxima já foi atingida. O conjunto de resultados de parâmetros máximos para diferentes instâncias de diferentes algoritmos permite se ter uma ideia da intensidade computacional máxima que pode ser obtida para o processamento de áudio em tempo real num dado dispositivo.

2.1.2 Diferenças nas abordagens de cada plataforma

As características de cada plataforma abordada (suas limitações, as bibliotecas disponíveis e as formas de programação) influencia nas possibilidades de implementação. No caso do Arduino, o foco é na operação do microcontrolador; para a GPU o ponto interessante é o modelo de "terceirização" da computação paralelizável utilizando uma placa para computação paralela; e para o sistema Android a grande motivação é o fato de ser um sistema operacional que roda em uma quantidade grande de diferentes marcas e modelos de dispositivos móveis.

Desafios comuns com soluções diferentes

Alguns desafios são comuns a todas as plataformas. Quais são, por exemplo, as possibilidades de implementação de uma função de manipulação de blocos de amostras que seja executada periodicamente em tempo real? No Arduino, a leitura e escrita das amostras são feitas diretamente nos registradores do processador, sem hardware específico para intermediar um *buffer* de áudio e controlar a taxa de leitura e escrita das amostras. Como veremos no Capítulo 3, a opção feita neste caso foi por utilizar a interrupção de um contador de 8 bits para ajudar nesta tarefa. Já na GPU, este controle foi feito pelo agendamento já implementado no software Pure Data (Puckette, 1996) e as rotinas de processamento paralelo foram implementadas como um tipo de *plugin*. No Capítulo 4 esta opção será abordada com mais detalhes. No capítulo 5 será explicado como e por que, no sistema Android, a opção foi por utilizar as funções de agendamento do sistema operacional. Neste

caso, o objeto principal de estudo é exatamente o sistema operacional então nada seria mais natural do que utilizar sua infraestrutura para executar a tarefa.

Desafios específicos de cada plataforma

Outros desafios, porém, são específicos a cada tipo de dispositivo. Para o Arduino, por exemplo, uma implementação direta dos algoritmos descritos na seção anterior demonstrou que, além da plataforma possuir poder computacional bastante limitado, as operações nativas do microcontrolador de 8 bits não são suficientes para executar o processamento em tempo real com frequências de amostragem que possam representar uma parte significativa do espectro audível. Por não possuir operações de ponto flutuante implementadas em hardware, foi necessário buscar alternativas de implementação que utilizassem operações mais simples (como aritmética de inteiros e operações sobre bits).

Para a programação em GPU a situação é diferente. Além de focar em algoritmos que permitem implementações de paralelismo, as placas com este tipo de processador incluem otimizações de hardware e software em diferentes estágios do circuito. Diversas aplicações já foram descritas utilizando mapeamento de problemas gerais para o domínio do processamento gráfico nestes diferentes estágios. As bibliotecas para programação em GPU já possuem implementações altamente eficientes da FFT, de forma que reimplementar este algoritmo não faria sentido. Neste caso, a opção foi por medir a velocidade da FFT presente na biblioteca para diferentes tamanhos de blocos, e implementar um algoritmo mais pesado como o *Phase Vocoder* que consiste, como poderá ser visto na Seção 2.2.4, de uma análise do sinal através de FFT e uma ressíntese utilizando osciladores senoidais. Assim, foi possível se ter uma ideia de como as placas se comportam ao executar tanto algoritmos básicos quanto processamentos mais pesados, uma vez que o foco das GPUs é exatamente explorar ao máximo a intensidade computacional de algoritmos paralelizáveis.

Para o sistema Android, a possibilidade de escrever um programa que pode ser executado em qualquer aparelho permite a implementação de um software "genérico" de análise de desempenho, que possa coletar dados do aparelho e da execução de alguns algoritmos e enviar relatórios via email para análise posterior. Esta abordagem permite uma análise do desempenho tanto pelo desenvolvedor do software quanto pelo utilizador do aparelho, que ao utilizar o aplicativo tem a oportunidade de explorar a capacidade computacional do seu dispositivo para uma série de tarefas relacionadas a aplicações de Computação Musical.

Pelos motivos descritos acima, a quantidade e forma de implementação dos algoritmos que serão abordados na próxima seção varia em cada plataforma. Espera-se que os motivos das opções feitas em cada caso fiquem claros ao longo do texto, após a descrição dos detalhes de cada arquitetura e das formas de programação para cada plataforma. A próxima seção descreve o arcabouço teórico que fundamenta a implementação dos algoritmos de processamento de áudio utilizados para análise de desempenho.

2.2 Algoritmos e técnicas de manipulação de áudio

Como foi visto na Seção 1.1.1, para que a operação em tempo real seja viável, o dispositivo utilizado para processamento deve terminar a computação do bloco de amostras dentro do período teórico do ciclo DSP. Na prática, o período real disponível é menor do que o teórico e os indicadores de intensidade computacional de um dispositivo ao executar cada algoritmo durante um ciclo DSP variam de acordo com o algoritmo e a implementação.

Como mencionado na Seção 2.1.1, quatro algoritmos bastante utilizados na manipulação de fluxos de áudio foram escolhidos para implementação nas plataformas para viabilizar a análise de desempenho: Transformada Rápida de Fourier, Convolução, Síntese Aditiva e *Phase Vocoder* (Oppenheim *et al.*, 1999; Zölzer, 2002). Nas próximas seções será descrito o arcabouço teórico que fundamenta cada um destes algoritmos e serão levantadas algumas perguntas naturais sobre a viabilidade de certas tarefas que surgem no contexto do processamento em tempo real.

Sinais analógicos, sinais digitais e amostragem

Um **sinal analógico** é um processo físico que depende do tempo e pode ser modelado por uma função real sobre uma variável real t que representa o tempo. No caso de sinais de áudio, é comum que esta função represente a pressão sonora, o valor da corrente elétrica que corre por um fio ou a posição da membrana de um alto falante num momento específico. É raro que um sinal analógico possa ser representado por uma expressão analítica, pois a maioria dos sinais analógicos são muito complexos e possuem ruído. Apesar de muitas ferramentas matemáticas se aplicarem a sinais contínuos, atualmente grande parte do processamento de sinais é executada em computadores, pois são dispositivos flexíveis e rápidos. Apesar disso, por trabalharem com símbolos discretos, computadores não conseguem representar sinais analógicos em sentido estrito (Broughton e Bryan, 2011).

No contexto deste trabalho, um **sinal digital** é definido como uma função que mapeia um subconjunto sequencial dos números inteiros em um conjunto finito de símbolos. Um sinal analógico pode ser digitalizado por um processo de **amostragem**, que aproxima valores do sinal analógico tomados em intervalos igualmente espaçados no tempo. Dado um sinal analógico $x(t)$ definido em um intervalo $a \leq t < b$ e um inteiro N que representa a quantidade de amostras a serem obtidas do sinal analógico, o **intervalo de amostragem** é definido como $\Delta t := (b - a)/N$. A medição de $x(t)$ em pontos igualmente espaçados por Δt dá origem a pontos $x(n) = x(a + n\Delta t)$, para $n = 0, 1, \ldots, N - 1$.

O passo final do processo de amostragem chama-se **quantização** e consiste em aproximar os valores $x(n)$ para representá-los por um conjunto finito de símbolos. Uma forma bastante comum de realizar a quantização é dividir a imagem do sinal analógico em 2^b partes (que podem ser representados por um certo número b de bits) e utilizar alguma regra de arredondamento do valor de cada $x(n)$ para obter o símbolo mais próximo. Esse arredondamento gera um **erro de quantização**, que implica em perda de informação em relação ao sinal analógico. O resultado final da amostragem de um sinal analógico $x(t)$ é um vetor $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1})$ de tamanho N, onde cada x_n pertence, no caso do exemplo acima, ao conjunto de 2^b símbolos utilizados para arredondar o valor das medições do sinal analógico. Ao longo deste trabalho será permitido um abuso de notação e algumas vezes será dito que uma amostra digital pertence ao conjunto $\mathbb{R}$ ou $\mathbb{C}$, quando na verdade o processo que ocorre é de quantização desta amostra em um conjunto finito, como por exemplo 8 bits PCM ou 32 bits ponto flutuante. As questões relacionadas à qualidade numérica de representação e de resultados de computação com valores quantizados são fundamentais para muitas áreas da computação mas estão fora do escopo deste trabalho.

Perda de informação, frequência de amostragem e o teorema de Nyquist-Shannon

O erro introduzido pelo processo de quantização causa uma perda de informação em relação ao sinal analógico original: o valor de um certo $x(n)$ não pode ser recuperado a partir do conhecimento da amostra x_n correspondente. Há ainda outro tipo de informação que pode ser perdida no processo de amostragem, relacionada ao tamanho do intervalo de amostragem Δt.

A **frequência de amostragem**, definida como $R := 1/\Delta t$, pode ser compreendida como o número de amostras tomadas do sinal analógico por unidade de tempo. O teorema de Nyquist-Shannon (Oppenheim *et al.*, 1999) diz que uma condição suficiente para que uma função contínua possa ser reconstruída perfeitamente a partir de uma representação discreta de pontos igualmente espaçados é que não seja composta de frequências maiores do que $R/2$ Hz. Por outro lado, se uma função contínua possui em sua composição frequências maiores do que $R/2$ Hz e é amostrada com uma frequência de amostragem menor ou igual a R Hz, então ocorre um processo chamado **aliasing**, no qual todas as componentes com frequências que excedem $R/2$ Hz são representadas como componentes com frequências mais baixas, causando distorção no sinal amostrado (Oppenheim *et al.*, 1999). Assim, para evitar distorção, a amostragem a R Hz de um sinal analógico contendo frequências maiores do que $R/2$ Hz deve ser precedida de uma etapa analógica de **filtragem**, ou seja, de remoção destas componentes, a fim de garantir que o sinal a ser amostrado

satisfaça o critério de Nyquist-Shannon e que não ocorra *aliasing*.

2.2.1 Transformada Rápida de Fourier

Um sinal digital periódico de período N pode ser completamente descrito por N componentes senoidais com frequências harmonicamente relacionadas. A **Transformada de Fourier Discreta** (em inglês *Discrete Fourier Transform*, ou **DFT**) é uma função sobre um vetor de tamanho N que corresponde a uma mudança de base em um determinado espaço vetorial e permite o cálculo das N componentes de frequência do sinal (Broughton e Bryan, 2011). As N componentes são igualmente espaçadas no espectro e limitadas superiormente pela metade da frequência de amostragem (veja a Seção 2.2). A DFT de $\mathbf{x}$ é, portanto, definida como o vetor $\mathbf{X} = (X_0, X_1, \ldots, X_{N-1}) \in \mathbb{C}^N$ onde a $k-$ésima componente é calculada da seguinte forma:

$$X_k = \sum_{n=0}^{N-1} x_e^{\frac{-2\pi i}{N} nk}, \text{ para } k = 0, \ldots, N-1.$$

A DFT é uma transformação linear e inversível. Dado um vetor $\mathbf{X} = (X_0, X_1, \ldots, X_{N-1}) \in \mathbb{C}^N$, a **Transformada Discreta de Fourier Inversa**, ou **IDFT** de $\mathbf{X}$, é o vetor $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1})$ com componentes dadas por:

$$x_k = \frac{1}{N} \sum_{m=0}^{N-1} X_m e^{\frac{2\pi i}{N} mk}, \text{ para } k = 0, \ldots, N-1.$$

A **Transformada Rápida de Fourier** (em inglês *Fast Fourier Transform*, ou **FFT**) é uma implementação eficiente da DFT que diminui sua complexidade de $O(N^2)$ para $O(N \log(N))$, onde N é o número de amostras no domínio do tempo ou, de forma equivalente, o número de índices de frequência que descrevem o espectro do sinal após a computação da Transformada (Press *et al.* , 1992). O algoritmo da FFT tira vantagem de redundância e simetria presentes em passos intermediários do cálculo e é usado em muitos outros algoritmos de processamento de sinais. O esquema geral da FFT pode ser visto na Figura 2.1. A implementação eficiente da transformada inversa (IDFT), usando dos mesmos mecanismos de divisão e conquista usados pela FFT, é denotada por **IFFT**.

O tamanho do bloco é um parâmetro fundamental da FFT que determina sua complexidade. Existem, inclusive, algoritmos que otimizam o cálculo para diferentes tipos de tamanho: blocos de tamanho potência de 2, tamanho primo, etc. O algoritmo de Cooley e Tukey (Cooley e Tukey, 1965) possui uma derivação matemática e uma implementação razoavelmente simples para tamanhos de bloco que sejam potências de 2. Neste caso, o cálculo das componentes X_k da Transformada de Fourier pode ser reduzido ao cálculo de duas transformadas com a metade do tamanho da seguinte forma:

$$
\begin{aligned}
X_k &= \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N} nk} \\
&= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{\frac{-2\pi i}{N}(2m)k} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{\frac{-2\pi i}{N}(2m+1)k} \\
&= \underbrace{\sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{\frac{-2\pi i}{N}(2m)k}}_{E_k} + e^{\frac{-2\pi i}{N}k} \underbrace{\sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{\frac{-2\pi i}{N}(2m)k}}_{O_k} \\
&= E_k + e^{\frac{-2\pi i}{N}k} O_k
\end{aligned}
$$

Se a definição da DFT for estendida para todos os inteiros, é fácil ver que ela se torna uma função periódica de período N:

$$
\begin{aligned}
X_{k+N} &= \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N} n(k+N)} \\
&= \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N} nk} \underbrace{e^{\frac{-2\pi i}{N} nN}}_{1} \\
&= \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N} nk} \\
&= X_k
\end{aligned}
$$

Da periodicidade das FFTs de tamanho $\frac{N}{2}$ vale que $E_{k+N/2} = E_k$ e $O_{k+N/2} = O_k$, para $k = 1 \ldots \frac{N}{2} - 1$. Assim, a transformada de tamanho N pode ser escrita da seguinte forma:

$$
X_k = \begin{cases} E_k + O_k e^{\frac{-2\pi i}{N} k} & \text{se } k < \frac{N}{2} \\ E_{k-\frac{N}{2}} + O_{k-\frac{N}{2}} e^{\frac{-2\pi i}{N}(k-\frac{N}{2})} & \text{se } k \geq \frac{N}{2} \end{cases}
$$

Após o cálculo das duas transformadas menores, o valor é combinado para se obter o valor da transformada maior, como pode ser visto na figura 2.1.

A restrição de tempo do processamento em tempo real é linear: o período máximo permitido para emissão de novas amostras corresponde a N/R segundos (veja a Seção 1.1.1). Por outro lado, a complexidade computacional da FFT é $O(N \log(N))$ e, portanto, para um certo dispositivo computacional sempre existe algum valor de N para o qual o tempo de computação da FFT excede o período teórico do ciclo DSP. Neste contexto, a dúvida é somente qual é o tamanho máximo de uma FFT que pode ser computada em tempo real para um certo dispositivo. Veremos mais adiante que esta dúvida pode ser sanada através da implementação da FFT e medição do tempo de execução nos diferentes dispositivos. No Capítulo 3, a limitação de poder computacional do Arduino nos forçará a diminuir significativamente a frequência de amostragem de forma a aumentar o período do ciclo DSP e viabilizar a computação da FFT. Já no Capítulo 4 a existência de centenas de processadores que operam em paralelo na GPU deixará de fazer a diferença quando o tamanho dos blocos estiver na casa dos milhares. Por fim, o Capítulo 5 ilustrará a diversidade da capacidade computacional dos dispositivos que rodam Android, e consequentemente as diferentes limitações de tamanho máximo de FFT que conseguem computar em tempo real.

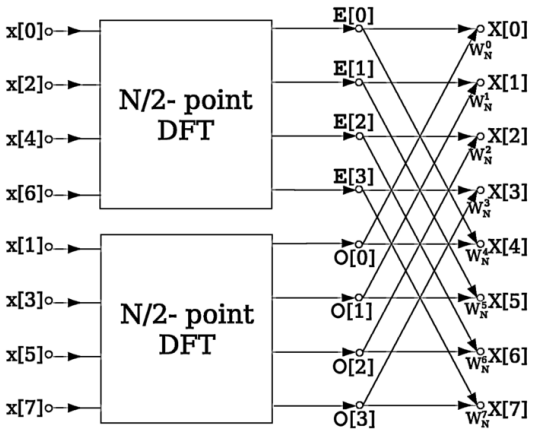

Figura 2.1: *A FFT utiliza uma abordagem de divisão e conquista e salva resultados intermediários para acelerar o cálculo do espectro do sinal. A figura mostra o esquema do cálculo de uma FFT de 8 pontos e como os resultados são mapeados para os índices de frequência.*

2.2.2 Convolução no domínio do tempo

A **convolução** é uma operação definida sobre funções contínuas bastante utilizada nos campos da probabilidade, estatística, visão computacional, processamento de sinais e equações diferenciais. Também definida no caso discreto, a convolução é operação básica para representação e manipulação de sinais em diferentes bases, dando origem às transformadas de Fourier, de Wavelets, entre outras.

A operação de **convolução circular** é definida sobre dois vetores $\mathbf{x}, \mathbf{y} \in \mathbb{C}^N$ e resulta num novo vetor $\mathbf{w} \in \mathbb{C}^N$ com componentes w_r dados por:

$$w_r = \sum_{k=0}^{N-1} x_k y_{(r-k) \pmod N}.$$

A convolução circular é denotada por $\mathbf{w} = \mathbf{x} * \mathbf{y}$ e é fácil ver que possui as propriedades de linearidade, comutatividade, e associatividade. Além disso, pode ser formulada matricialmente e é periódica se os vetores $\mathbf{x}$ e $\mathbf{y}$ forem estendidos periodicamente, com período N, para todos os valores inteiros (Broughton e Bryan, 2011).

O **teorema da convolução** diz que a operação de convolução circular de dois sinais no domínio do tempo corresponde à operação de multiplicação ponto a ponto dos espectros destes sinais no domínio das frequências. Mais formalmente, sejam $\mathbf{x}$ e $\mathbf{y}$ dois vetores em $\mathbb{C}^N$ com DFTs, respectivamente, $\mathbf{X}$ e $\mathbf{Y}$, e seja $\mathbf{w} = \mathbf{x} * \mathbf{y}$, com DFT $\mathbf{W}$. Então, vale que:

$$W_k = X_k Y_k, \text{ para } 0 \leq k \leq N-1.$$

Isso permite ver que a convolução circular pode ser implementada de forma eficiente através da expressão $w = IFFT(FFT(x) .* FFT(y))$, onde $.*$ denota o produto dos vetores componente a componente. Essa implementação é chamada de convolução rápida, e tem custo computacional $O(N \log(N))$ ao invés de $O(N^2)$, associado ao cálculo direto da expressão de w_r, $r = 0, 1, \ldots, N-1$.

Convolução circular e filtros FIR

O resultado visto acima permite utilizar a convolução circular no domínio temporal para obter um novo sinal cujo espectro seja a multiplicação dos espectros de dois sinais de entrada. Dado um vetor $\mathbf{h}$ com DFT $\mathbf{H}$, se a convolução de $\mathbf{h}$ com um sinal de entrada $\mathbf{x}$ resulta na modificação do espectro $\mathbf{X}$ de forma controlada, torna-se interessante chamar $\mathbf{h}$ de **filtro**. Quanto menor for o número de coeficientes de $\mathbf{h}$ diferentes de zero, menor é o custo computacional da implementação do filtro diretamente através da expressão da convolução no domínio do tempo. O maior coeficiente

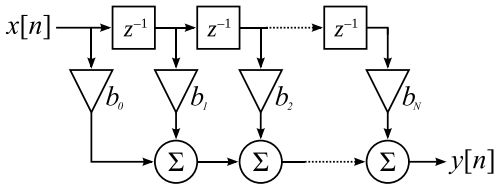

Figura 2.2: *Esquema geral para a implementação de filtros FIR utilizando convolução no domínio do tempo: o sinal da entrada $x[n]$ é convolvido com os coeficientes b_i, para $i = 1 \ldots N$, que caracterizam a resposta impulsiva do filtro, e gera um sinal de saída $y[n]$.*

não nulo define a **ordem** do filtro e determina o número de amostras "do passado" que serão utilizadas para calcular uma nova amostra de saída. Assim, características interessantes para sinais candidatos a filtro são possuir espectro conhecido e parametrizável, para permitir o controle dos efeitos no sinal de entrada, e representação temporal com poucos coeficientes, de forma a viabilizar uma implementação com baixa complexidade computacional no domínio do tempo.

A implementação da convolução diretamente no domínio do tempo é uma técnica amplamente utilizada em uma série de algoritmos de computação musical, sendo particularmente eficiente quando a ordem do filtro é pequena. Uma classe de filtros bastante estudada e que possui as características acima são os *filtros de resposta impulsiva finita* (ou **filtros FIR**), cujo esquema geral pode ser visto na Figura 2.2. A equação geral do cálculo do sinal $\mathbf{y}$ resultante da filtragem de um sinal de entrada $\mathbf{x}$ de tamanho N por um filtro FIR $\mathbf{h}$ de ordem K, implementado utilizando a convolução no domínio do tempo, é escrita da seguinte forma:

$$y_n = \sum_{k=0}^{K} h_n x_{n-k}, \;\; n = 0, \ldots, N-1.$$

Note que nesta formulação, o cálculo das primeiras $K-1$ amostras necessitarão de valores de x_n para n negativo. Uma solução comum é utilizar valores nulos nestes casos, o que corresponde à interpretação da convolução linear que será discutida a seguir, através da implementação conhecida como convolução linear rápida.

Convolução linear e convolução linear rápida

Uma outra operação, que está relacionada à convolução circular mas possui complexidade computacional mais baixa e fornece um resultado um pouco diferente, é a chamada **convolução linear**, que essencialmente se difere da convolução circular pelo anulamento dos termos x_{n-k} quando $n-k<0$ (ao invés de considerá-los iguais a $x_{(n-k) \,(\mathrm{mod}\ N)}$). Dados dois sinais $\mathbf{x}$ e $\mathbf{h}$ de tamanho N, a convolução linear pode ser computada eficientemente da seguinte forma:

1. Estenda os sinais $\mathbf{x}$ e $\mathbf{y}$ com zeros à direita até o tamanho $2N$.
2. Compute a FFT de $2N$ pontos dos dois sinais.
3. Multiplique os espectros calculados para obter $Y_n = X_n H_n$, para $n = 0, \ldots, 2N-1$.
4. Compute a IFFT de $2N$ pontos do sinal $\mathbf{Y} = (Y_0, \ldots, Y_{2N-1})$ para obter $\mathbf{y}$ com coeficientes y_n para $n = 0, \ldots, Y_{2N-1}$.

Como é baseada nos algoritmos da FFT e IFFT, o cálculo da convolução linear rápida possui complexidade computacional igual a $O(2N \log(2N))$, que é essencialmente $O(N \log(N))$. Apesar disso, o vetor obtido não corresponde ao sinal cujo espectro é a multiplicação dos espectros dos sinais de entrada de tamanho N, pois o sinal $\mathbf{y}$ obtido pela convolução linear rápida possui tamanho

$2N$ e seu espectro é a multiplicação dos espectros dos sinais de entrada estendidos com zeros até o tamanho de $2N$.

A implementação da convolução linear diretamente no domínio do tempo, que possui custo computacional $O(N)$ por amostra, tem a vantagem de permitir a implementação de filtros com coeficientes que variam no tempo, além de permitir a interpretação de $\mathbf{x}$ como fluxo de entrada (de tamanho arbitrário) e $\mathbf{h}$ como resposta impulsiva de um filtro de tamanho N. A complexidade computacional do cálculo da implementação da convolução no domínio do tempo depende, portanto, do tamanho do sinal de entrada e da ordem do filtro utilizado. Se a ordem do filtro é constante, então a complexidade é linear no tamanho do sinal de entrada. Neste sentido, a dúvida relevante para este trabalho é sobre o tamanho máximo de um filtro que pode ser aplicado a um sinal de entrada em tempo real em cada dispositivo considerado. A exemplo do que foi comentado sobre a FFT na Seção 2.2.1, a resposta dependerá da natureza de cada dispositivo. No Capítulo 3, por exemplo, será visto que a restrição dos filtros a uma família bastante específica permite aumentar consideravelmente a ordem de alguns filtros implementados no Arduino.

Janelamento, processamento em blocos, e efeitos no espectro

No processamento de áudio digital em tempo real, supõe-se que o sinal digital é obtido e/ou gerado em blocos de amostras que representam seções do sinal correspondentes a intervalos de tempo iguais (veja a Seção 1.1.1). Por este motivo, o sinal completo nunca está totalmente disponível antes do final da execução do processamento. As únicas partes do sinal que estão disponíveis para processamento são o bloco atual e os blocos passados, limitados pelo tamanho da memória do dispositivo utilizado. O arcabouço teórico que fundamenta a manipulação do sinal em blocos é chamado **janelamento**, que pode ser entendido como um "recorte" do sinal digital de forma que se considere somente um pedaço de tamanho fixo por vez.

Em sua forma mais simples, o janelamento pode ser compreendido como a multiplicação ponto a ponto do sinal digital original por uma vetor que vale 1 nos índices que correspondem ao bloco considerado e zero em todos os outros pontos. Como a multiplicação no domínio do tempo corresponde à operação de convolução no domínio das frequências (veja a Seção 2.2.2), o janelamento introduz uma distorção no sinal que é quantificável. O efeito do janelamento no domínio do tempo corresponde, no domínio das frequências, à convolução do espectro do sinal original com o espectro da janela utilizada. A janela retangular, por sua descontinuidade acentuada (em termos discretos) nos pontos onde começa e termina, possui um espectro relativamente rico em relação a outras janelas, mais frequentemente utilizadas, que possuem transições mais suaves nas extremidades.

Considere que $\mathbf{x} \in \mathbb{C}^N$ seja o sinal original e que $\mathbf{X} \in \mathbb{C}^N$ seja o vetor que representa suas componentes em frequência calculadas pela FFT (veja a Seção 2.2.1). Considere também que $\mathbf{w} \in \mathbb{C}^N$ seja uma janela com $\mathbf{w}_k = 0$ para $k < m$ e $k \geq m+M$ para uma certa escolha de m e M, e que $\mathbf{W} \in \mathbb{C}^N$ seja seu espectro. Considere, ainda, o sinal $\tilde{\mathbf{x}} = (\mathbf{w}_m\mathbf{x}_m, \mathbf{w}_{m+1}\mathbf{x}_{m+1}, \ldots, \mathbf{w}_{m+M-1}\mathbf{x}_{m+M-1}) \in \mathbb{C}^M$ que corresponde à multiplicação ponto a ponto de $\mathbf{x}$ com $\mathbf{w}$, considerando somente os M valores dentro da janela. Qual é, então, a relação entre os espectros de $\mathbf{x}$ e $\tilde{\mathbf{x}}$? Suponha que $N = qM$ para algum q inteiro. Então, a FFT $\tilde{\mathbf{X}} \in \mathbb{C}^M$ do sinal $\tilde{\mathbf{x}}$ é dada por $\tilde{\mathbf{X}}_s = \dfrac{e^{2\pi i m s/M}}{N}(\mathbf{X} * \mathbf{W})_{qs}$, para $s = 0, 1, \ldots, M-1$ (Broughton e Bryan, 2011).

Sobreposição de blocos e janelamento deslizante

Mesmo em processamentos que não sejam realizados em tempo real pode haver motivos para realizar o janelamento do sinal. Um exemplo são processamentos em tempo-frequência utilizando a FFT (vista na Seção 2.2.1). Uma transformada do sinal completo captura uma descrição das frequências que compõem o sinal como um todo e muitas vezes pode não capturar com precisão aquelas frequências que estão presentes somente em uma parte do sinal. Através do janelamento, é possível analisar seções menores do sinal e observar eventos *transientes*, que ocorrem somente em regiões pequenas do sinal (veja mais sobre isso nas Seções 2.2.3 e 2.2.4). A resolução da análise, ou

seja, o número de componentes igualmente espaçadas entre 0 e R/2 Hz que podem ser representadas por uma FFT feita sobre um bloco de amostras, é determinada pelo tamanho do bloco.

Há portanto uma certa dualidade entre a resolução nos domínios do tempo e das frequências. Diminuir o tamanho dos blocos com o objetivo de aumentar a resolução no domínio do tempo implica necessariamente em diminuir a resolução, ou a quantidade de componentes calculadas pela FFT, no domínio das frequências. Uma forma de obter um balanço entre a resolução nos dois domínios é permitir a sobreposição de blocos no domínio do tempo. A sobreposição permite que a resolução no domínio das frequências seja mantida, pois o tamanho de cada bloco é o mesmo, e que a resolução no domínio do tempo seja aumentada, pois a cada passo o início do novo bloco considerado é menos distante no tempo do início do bloco anterior (Zölzer, 2002). A realização da operação de janelamento sobre um sinal de forma que janelas sucessivas sejam consideradas a cada iteração (com ou sem sobreposição) é chamada **janelamento deslizante**.

No caso de processos de síntese que utilizam janelamento deslizante é necessário tomar uma decisão sobre a forma de combinar as partes dos blocos de amostras sobrepostas em blocos adjacentes. Uma técnica bastante utilizada é chamada ***overlap-add***. Se o janelamento foi feito utilizando uma janela com extremidades suaves, o procedimento de *overlap-add* consiste apenas em somar as amostras que se sobrepõem nos diferentes blocos, para obter a amostra final.

2.2.3 Síntese aditiva

A **síntese aditiva** é um processo de construção de um sinal através da soma de uma série de sinais periódicos mais simples. As componentes mais simples em geral não são percebidas individualmente, mas contribuem para a formação do sinal resultante. Esta técnica tem sido muito utilizada para a obtenção de novos sons e também para a ressíntese de sinais após terem passado por algum tipo de análise e processamento (Moore, 1990). As componentes básicas da síntese aditiva são funções periódicas, governadas por funções de amplitude e fase independentes, às quais são dadas o nome de **osciladores** (veja a Figura 2.3). Quanto maior o número de osciladores utilizados, mais ricos em informação podem ser os sinais gerados. Mas quanto maior o número de parcelas que têm que ser calculadas, também é maior a quantidade de recursos computacionais necessários para o cálculo da forma de onda sintetizada.

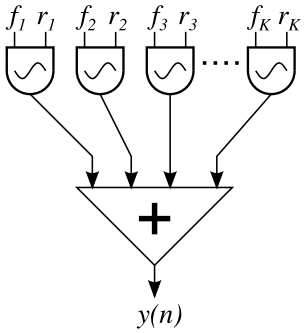

Figura 2.3: *Síntese aditiva: diversos osciladores governados por funções de amplitude e fase independentes são combinados para formar um sinal mais complexo.*

O teorema de Fourier diz que é possível sintetizar qualquer sinal periódico através de um conjunto (possivelmente infinito) de sinais senoidais harmonicamente relacionados (Moore, 1990). Apesar disso, é possível utilizar a síntese aditiva para gerar sinais não periódicos com banda de frequências limitada, através de um conjunto finito de osciladores. A ideia é utilizar o janelamento deslizante (veja a Seção 2.2.2) para sintetizar blocos de amostras que podem ser combinados de forma a compor um sinal mais complexo.

Considere inicialmente um conjunto de K frequências constantes f_k expressas em Hz e um conjunto de K funções de amplitude $r_k(n)$ que variam com o tempo; neste caso, a soma de K

osciladores senoidais que gera um sinal de áudio a uma frequência de amostragem de R Hz é dada pela seguinte expressão:

$$y(n) = \sum_{k=1}^{K} r_k(n) \sin\left(\frac{2\pi f_k n}{R}\right), \quad n \geq 0.$$

A expressão acima considera que as frequências dos osciladores são constantes. Para implementar corretamente osciladores cujas frequências podem variar com o tempo, é necessário adaptar a expressão considerando-se *valores instantâneos* de frequência. Nesse sentido, define-se a **frequência instantânea** como uma quantidade proporcional ao valor de incremento da fase (θ) da função seno, e assim a expressão da síntese aditiva para frequências que mudam ao longo do tempo corresponde a:

$$y(n) = \sum_{k=1}^{K} r_k(n) \sin\left(\theta_k(n)\right),$$

onde

$$\begin{aligned} \theta_k(n+1) &= \theta_k(n) + \Delta\theta_k(n), \text{ e} \\ \Delta\theta_k(n) &= \frac{2\pi}{R}\Delta f_k(n), \end{aligned}$$

na qual $\Delta\theta_k(n)$ é a frequência instantânea, expressa em radianos, do $k-$ésimo oscilador e $\Delta f_k(n)$ é a frequência instantânea do $k-$ésimo oscilador expressa em Hz (Moore, 1990).

Implementação utilizando consulta a tabela

Existem diversas implementações de aproximações da função seno, como por exemplo através do cálculo da expansão da série de Taylor ou como uma composição de frações, que convergem para o valor procurado com possibilidade de aproximação arbitrária. Outra opção é utilizar as funções trigonométricas de cálculo de seno que em geral estão presentes nas APIs dos dispositivos. Nem sempre estas opções são as mais econômicas em termos de custo computacional. Diferentes implementações possuem diferentes qualidades de aproximação e diferentes custos computacionais. Em determinados tipos de plataformas pode ser mais fácil computar uma solução em paralelo, enquanto que em outras um cálculo de uma série com muitas parcelas pode ser inviável.

Uma alternativa menos custosa em geral é a utilização de uma tabela pré-calculada contendo exatamente um período da função seno amostrada em intervalos iguais. O valor da fase do oscilador pode ser mapeado e modulado para o tamanho da tabela e índices fracionários podem ser consultados realizando algum tipo de interpolação no momento da consulta utilizando os valores dos índices inteiros mais próximos.

Na implementação por consulta a tabela é utilizada uma tabela $\mathbf{s} = (s_0, \ldots, s_{S-1})$ de tamanho S contendo um período da função seno amostrado em S pontos. Os índices da tabela podem ser previamente calculados de acordo com a seguinte expressão:

$$s_i = \sin(2\pi i/S), \text{ para } i = 0, \ldots, S-1.$$

O intervalo $[0, 2\pi[$ pode ser mapeado para o intervalo $[0, S[$ e, caso o resultado seja um valor fracionário, diferentes técnicas podem ser utilizadas para obter um valor intermediário entre dois índices inteiros da tabela $\mathbf{s}$. Dado um valor θ_k entre 0 e 2π para a fase do $k-$ésimo oscilador (constante, por enquanto), é possível obter índices j_k para consulta à tabela através da seguinte relação:

$$j_k = \frac{\theta_k S}{2\pi}.$$

Assim, substituindo θ_k na relação acima pelo valor dos argumentos do seno para cada oscilador no cálculo da síntese aditiva com frequências constantes, é possível obter K índices para consulta à tabela, um para cada oscilador, dados por $j_k = f_k nS/R$. O cálculo da síntese aditiva de K osciladores, utilizando a notação $\mathbf{s}[l]$ para expressar a consulta à tabela no índice (possivelmente fracionário) l, fica da seguinte forma:

$$y(n) = \sum_{k=1}^{K} r_k(n)\mathbf{s}[f_k nS/R].$$

No caso de frequências f_k que variam com o tempo, a frequência instantânea deve ser levada em conta e o cálculo da variação do índice $\Delta j_k(n)$ fica:

$$\Delta j_k(n) = \frac{\Delta\theta_k(n)S}{2\pi} = \Delta f_k(n)\frac{S}{R}.$$

O algoritmo completo de síntese aditiva com frequências que podem variar no tempo, utilizando osciladores calculados por consulta a tabela fica, portanto, assim:

$$\begin{aligned} y(n) &= \sum_{k=1}^{K} r_k(n)\text{consulta}(\mathbf{s}, L_k(n)), \\ L_k(n+1) &= L_k(n) + I_k(n) \pmod S, \end{aligned}$$

onde

- $y(n)$ é a saída do oscilador.
- $r_k(n)$ é a amplitude do oscilador k que varia ao longo do tempo.
- $\mathbf{s}$ é uma tabela de tamanho S contendo exatamente um período de uma forma de onda de período 2π amostrada nos pontos $0, 2\pi/S, 4\pi/S, \ldots, (S-1)2\pi/S$.
- $L_k(n)$ é o valor do índice da tabela relacionado ao $k-$ésimo oscilador no instante n.
- consulta$(\mathbf{s}, l)$ é uma função de consulta (que será comentada a seguir) que obtém um valor fracionário l da tabela $\mathbf{s}$.
- $I_k(n)$ é o incremento do índice de consulta a tabela no instante n dado por $I_k(n) = f_k(n)S/R$, onde $f_k(n)$ é a frequência a ser gerada pelo $k-$ésimo oscilador no instante n, S é o tamanho da tabela, e R é a taxa de amostragem.

Possíveis formas de consulta a tabela

Em geral, o resultado da expressão $i_k = L_k(n)$ é um índice fracionário para a maioria dos valores de k e n, e portanto alguma decisão tem que ser tomada sobre a forma de consultar a tabela para obter um valor intermediário $\tilde{s}_k$ que esteja entre índices inteiros. Algumas possibilidades imediatas são:

- **Consulta truncada:**

$$\tilde{s}_k = \mathbf{s}\left[\lfloor i_k \rfloor\right].$$

- **Consulta arredondada:**

$$\tilde{s}_k = \begin{cases} \mathbf{s}\left[\lfloor i_k \rfloor\right], & i_k - \lfloor i_k \rfloor \leq 0.5, \\ \mathbf{s}\left[\lceil i_k \rceil\right], & \text{caso contrário.} \end{cases}$$

- **Interpolação linear:** toma-se os dois índices inteiros mais próximos de i_k e calcula-se um valor intermediário utilizando interpolação linear:

$$\begin{aligned} d &= i_k - \lfloor i_k \rfloor, \\ i_0 &= \lfloor i_k \rfloor \pmod{S}, \\ i_1 &= i_0 + 1 \pmod{S}, \\ \tilde{s}_k &= (\mathbf{s}[i_1] - \mathbf{s}[i_0])d + \mathbf{s}[i_0]. \end{aligned}$$

- **Interpolação cúbica:** toma-se quatro índices inteiros próximos de i_k e calcula-se um valor intermediário utilizando interpolação cúbica:

$$\begin{aligned} d &= i_k - \lfloor i_k \rfloor, \\ i_0 &= \lfloor i_k \rfloor - 1 \pmod{S}, \\ i_1 &= i_0 + 1 \pmod{S}, \\ i_2 &= i_0 + 2 \pmod{S}, \\ i_3 &= i_0 + 3 \pmod{S}, \\ \tilde{s} &= -\frac{d(d-1)(d-2)\mathbf{s}[i_0]}{6} + \frac{(d+1)(d-1)(d-2)\mathbf{s}[i_1]}{2} \\ &\quad -\frac{(d+1)d(d-2)\mathbf{s}[i_2]}{2} + \frac{(d+1)d(d-1)\mathbf{s}[i_3]}{6}. \end{aligned}$$

As opções acima são frequentemente utilizadas no cálculo de índices fracionários, e cada uma possui um custo computacional inversamente proporcional à qualidade numérica que proporciona no resultado do valor calculado. É interessante notar que qualquer esquema de consulta (mesmo a mais simples, com índice truncado) pode ter sua qualidade numérica melhorada arbitrariamente aumentando-se o tamanho da tabela. O tamanho da memória de certos dispositivos pode representar um limite para este procedimento, como no caso do Arduino.

Análise de desempenho na síntese aditiva

Uma medida simples de desempenho na execução da síntese aditiva pode ser obtida incrementando-se o número de osciladores e realizando-se uma comparação entre o tempo gasto na síntese para diferentes quantidades de osciladores e o período teórico do ciclo DSP. Desta forma pode-se descobrir qual é a maior quantidade de osciladores que pode ser utilizada em tempo real em uma certa plataforma computacional, dada uma implementação de síntese aditiva.

Fica claro que, além do número de osciladores, diferentes formas de implementação de consulta a tabela também possuem custos computacionais distintos. Nos testes realizados nas diferentes plataformas que serão descritos nos próximos capítulos, sempre que possível serão realizadas comparações entre diferentes implementações. Além disso, para plataformas mais limitadas, a utilização de frequências constantes ou variantes com o tempo também pode causar diferenças significativas no desempenho.

2.2.4 Phase Vocoder

O ***Phase Vocoder***, ou simplesmente PV, é uma técnica para representar um sinal através de um conjunto de osciladores cujas amplitudes e frequências instantâneas variam com o tempo (Dolson , 1986). O número de osciladores utilizados é igual à metade do tamanho do bloco de amostras usado na fase de análise, de onde se extraem os parâmetros que controlam estes osciladores. Após a fase de análise e estimação de parâmetros, é possível reconstruir o sinal através dos parâmetros

obtidos, possivelmente alterando os valores dos parâmetros para obter diversos efeitos, como por exemplo modificações independentes na altura sonora e na duração do sinal, dispersão, mutação entre dois sinais, separação de componentes estáveis e transientes, robotização, ente outros (Zölzer, 2002).

No contexto do processamento em tempo real, o sinal de entrada é analisado em blocos de amostras, possivelmente com sobreposição, e os parâmetros são utilizados para sintetizar novos blocos de amostras que devem ser computados antes do término do período teórico do ciclo DSP. Existem várias possibilidades de implementação do *Phase Vocoder*. Uma forma de implementação que utiliza técnicas descritas anteriormente neste capítulo é a utilização da FFT para estimação das frequências instantâneas (amplitude e fase) e uso da síntese aditiva para ressíntese do sinal sonoro a partir dos parâmetros obtidos e eventualmente manipulados.

Em princípio, as frequências de análise da FFT são fixas: cada X_k calculado pela fórmula da FFT corresponde à frequência de análise igual a $\frac{kR}{N}$ Hz. Apesar disso, as frequências instantâneas de cada bloco de amostras podem ser estimadas através das diferenças de fase que cada componente de análise apresenta entre um bloco e outro. Se o fator de sobreposição dos blocos é igual a M e a variação da fase inicial do $k-$ésimo oscilador entre dois blocos consecutivos é de $\Delta_\phi(k)$, a $k-$ésima frequência "real" contida naquele bloco pode ser estimada por:

$$\tilde{f}_k = \left(\frac{M\Delta_\phi(k)}{2\pi} + k \right) \frac{R}{N}.$$

As frequências instantâneas calculadas de acordo com a expressão acima podem ser modificadas (ou não) e utilizadas para alimentar diretamente o algoritmo da síntese aditiva visto na Seção 2.2.3. A amplitude de cada oscilador na ressíntese pode ser estimada como sendo igual a duas vezes a amplitude obtida pela FFT para considerar a componente de frequência negativa, que será contabilizada no mesmo oscilador.

Neste contexto, o *Phase Vocoder* é simplesmente a concatenação dos algoritmos FFT e síntese aditiva, eventualmente separados por algum tipo de processamento no domínio das frequências. Em dispositivos com maior poder computacional, como é o caso da GPU, o Phase Vocoder figura como o algoritmo computacionalmente mais pesado que, quando executado utilizando diferentes tamanhos de bloco, pode ser utilizado para análise do desempenho no processamento em tempo real.

Capítulo 3

Processamento de áudio em tempo real em Arduino

Neste capítulo será abordado o processamento de áudio em tempo real utilizando o **Arduino Duemilanove**, um dos modelos descritos na seção 1.2.1. O projeto do Arduino Duemilanove é baseado no microcontrolador **ATmega328P**[1] da marca Atmel, série megaAVR. Na primeira seção, será dada uma visão geral sobre a plataforma e seu uso. Na segunda seção, uma descrição das estruturas principais do microcontrolador ajudará a esclarecer algumas das possibilidades de processamento de áudio em tempo real, permitindo assim o projeto de programa que realiza as tarefas básicas do processamento de áudio. Por fim, serão descritos os detalhes específicos dos experimentos feitos em Arduino e serão apresentados os resultados obtidos quanto ao desempenho no processamento de áudio em tempo real.

O aparelho utilizado nesta pesquisa foi gentilmente disponibilizado pelo grupo MOBILE[2], da ECA/USP, por todo o período que durou esta investigação.

3.1 Programando para Arduino

Com o objetivo de tornar a interação com o microcontrolador mais palatável, um dos focos do projeto Arduino é a manutenção de uma plataforma gráfica de desenvolvimento própria. Esta plataforma facilita a edição de arquivos de código fonte, a compilação e carga do programa no microcontrolador, e o monitoramento de eventual comunicação serial estabelecida com o dispositivo. Apesar disso, esta plataforma gráfica é bastante simples e não possui funcionalidades que contribuam para uma investigação sistemática que dependa de automatização da carga de programa e coleta de resultados de experimentos.

Nesta seção, veremos quais são os detalhes relevantes da compilação de programas para a linguagem de máquina do ATmega328P (ou mais genericamente para qualquer modelo da família AVR), da carga do programa na área de memória de programa do microcontrolador, e da comunicação serial com o dispositivo. As informações aqui contidas devem ser suficientes para viabilizar a interação automatizada com qualquer modelo de Arduino.

3.1.1 Estrutura de um programa e bibliotecas

Um programa que será executado no microcontrolador de um Arduino deve ter ao menos duas funções definidas: `setup()` e `loop()`. A função `setup()` é chamada no início da execução do programa (após a alimentação com energia ou uma reinicialização do sistema), e é usada para inicializar variáveis, configurar os modos dos pinos de entrada e saída, inicializar bibliotecas, configurar os relógios, etc. Em seguida, a função `loop()` é executada consecutivamente até o fim da operação do aparelho.

[1] http://www.atmel.com/devices/ATMEGA328P.aspx
[2] http://www.cmu.eca.usp.br/mobile/

Uma série de bibliotecas estão disponíveis para possibilitar a interface com dispositivos de entrada e saída como interface *serial*, *Ethernet*, visores de cristal líquido, matrizes de LEDs, entre outros[3]. Também é possível encontrar bibliotecas para tarefas comuns como controle dos relógios, troca de mensagens com o computador, entre Arduinos ou com dispositivos de outras naturezas, e até mesmo para tarefas mais avançadas como o gerenciamento de servidores HTTP.

3.1.2 Compilação

A ferramenta `avr-gcc`[4] é uma versão do compilador `gcc` adaptada para traduzir código escrito em C/C++ para objetos binários que podem ser carregados e executados em microcontroladores da linha megaAVR da Atmel. Esta ferramenta está disponível nos repositórios das distribuições GNU/Linux mais comuns e é o programa utilizado para compilação pela interface gráfica de programação do Arduino.

A utilização da IDE gráfica do Arduino permite a escrita de programas mais enxutos (com extensão `.ino`) que, no momento da compilação, são encaixados em arquivos de código fonte completos (com extensão `.c`) incluindo cabeçalhos, definições de tipos e funções, chamadas das funções `setup()` e `loop()`, etc, antes de serem processados pelo `avr-gcc`. Um arquivo `Makefile` apropriado permite a utilização direta do `avr-gcc` para compilação sem a necessidade da utilização da ferramenta gráfica.

3.1.3 Cópia do programa para o microcontrolador

Uma das grandes vantagens do Arduino como plataforma de prototipação e experimentação com microcontroladores é que toda a interface para carga do programa no microcontrolador já vem embutida no hardware e no software. O microcontrolador possui a capacidade de autoprogramação e isto, junto com um programa de boot adequado, permite que um arquivo binário contendo um programa compilado possa ser carregado no microcontrolador através de uma conexão USB.

Uma vez que o Arduino esteja conectado ao computador via interface USB, a carga do programa consiste em dois passos: (1) envio de um sinal para reinicialização do microcontrolador, e (2) escrita do programa na memória flash. Em GNU/Linux, cada uma destas operações pode ser feita utilizando-se, respectivamente, as ferramentas `stty` e `avrdude`. Abaixo pode-se ver um exemplo de chamada dos dois comandos para um Arduino que utiliza o modelo de microcontrolador ATmega328P:

```
1 PORT=/dev/ttyUSB0
2 ARDUINO_DIR=/usr/share/arduino
3 stty -F ${PORT} hupcl
4 avrdude -V -F -C ${ARDUINO_DIR}/hardware/tools/avr/etc/avrdude.conf \
5   -p atmega328p -P ${PORT} -c arduino -b 57600
6   $
```

3.1.4 Comunicação serial

O microcontrolador possui uma interface serial para comunicação com outros dispositivos[5] que, nesta investigação, é utilizada somente para obter o resultado das medições feitas durante os experimentos. A forma mais simples de transmitir e receber dados é a partir da porta USB, a mesma interface utilizada para gravar o programa no microcontrolador.

[3] http://arduino.cc/en/Reference/Libraries
[4] http://www.avrfreaks.net/wiki/index.php/Documentation:AVR_GCC
[5] http://arduino.cc/en/Reference/Serial

Para utilização desta interface serial, é necessário incluir no código fonte do programa o cabeçalho `#include <HardwareSerial.h>` (a ferramenta gráfica do Arduino já faz isso de forma transparente) e definir uma taxa de transmissão de bits no código com a chamada da função `Serial.begin(57600)` (onde o valor 57600 é apenas um exemplo e pode ser substituído por algumas outras taxas disponíveis, de acordo com a documentação).

Em um computador com GNU/Linux, pode-se utilizar o programa `minicom`, que também está disponível nos repositórios das principais distribuições, para enviar e receber dados via comunicação serial com o Arduino. Um exemplo de linha de comando para uma taxa de transferência de 57600 bits/s e utilizando a interface USB pode ser visto abaixo:

```
1 minicom -b 57600 -D /dev/ttyUSB0
```

3.2 Processamento de áudio em tempo real em Arduino

Como será visto na Seção 3.2.2, a taxa máxima de transferência do mecanismo de comunicação serial não é suficiente para alimentar o microcontrolador na velocidade necessária para a manipulação em tempo real de sinais de áudio com frequências de amostragem que possam representar parte significativa do espectro audível. Portanto, para realizar o processamento de áudio em tempo real sem o uso de hardware adicional, o microcontrolador deve ser configurado de forma que seja possível realizar as três operações básicas do processamento de áudio em tempo real: amostragem de um sinal analógico de entrada de forma a obter um sinal digital, manipulação periodica do sinal digital capturado, e conversão do sinal de volta para o domínio analógico. Cada uma destas tarefas pode ser realizada de diversas formas, e para este estudo a escolha foi usar as funcionalidades intrínsecas ao microcontrolador.

A maioria dos modelos do Arduino utiliza microcontroladores da linha megaAVR da marca Atmel[6]. O modelo que foi utilizado neste trabalho possui um microcontrolador modelo ATmega328P, que conta com uma unidade central de processamento de 8 bits do tipo RISC com capacidade de operar a até 16 MHz, e dispondo de 32 KB de memória para armazenamento do programa e 2 KB de memória SRAM.

Nas próximas seções, veremos com detalhes quais são as características do microcontrolador ATmega328P que possibilitam a realização de processamento de sinais de áudio em tempo real nos termos descritos acima. Como veremos mais adiante, o programa desenvolvido para análise de desempenho configura o microcontrolador através da manipulação de valores de seus registradores. Por este motivo, o programa pode ser utilizado somente nos microcontroladores cujas instruções de configuração são compatíveis com as do ATmega328P. Apesar disso, a lógica de operação por trás do programa utiliza funcionalidades que estão presentes em grande parte dos microcontroladores, e portanto após a leitura deste capítulo espera-se que se tenha informação suficiente para alterar facilmente o programa para qualquer propósito, inclusive adaptando-o a outros modelos de microcontrolador. Neste capítulo, sempre que for feita uma referência ao *microcontrolador*, deve-se entender que se trata de um modelo específico, o ATmega328P.

3.2.1 Elementos do microcontrolador

Para saber como configurar o dispositivo para o processamento de áudio em tempo real, é necessário um entendimento geral do funcionamento interno do microcontrolador. Um microcontrolador da série Atmel megaAVR é composto de diversos elementos, alguns fundamentais para nossa investigação, que serão cobertos brevemente nesta seção.

[6] http://www.atmel.com/products/microcontrollers/avr/megaavr.aspx

CPU, Registros e interrupções

A CPU do microcontrolador possui uma unidade de lógica aritmética que dispõe de centenas de *registros*: porções de memória que armazenam dados utilizados na computação e determinam o fluxo de execução do programa. No caso do ATmega328P, 32 destes registros podem ser usados para computação de propósito geral, enquanto que os outros são reservados e desempenham funções específicas. Além disso, neste modelo não há representação de números com ponto flutuante em hardware, de forma que somente operações sobre inteiros e valores fracionários de ponto fixo são executadas rapidamente, enquanto que operações mais complexas têm que ser emuladas via software.

Uma *interrupção* é uma tentativa de desvio do fluxo corrente de execução através da alteração de determinados valores em registros específicos. Para os propósitos deste trabalho, as interrupções são de extremo valor pois são elas as estruturas de baixo nível que permitem que um certo código seja executado com uma frequência fixa (ao menos se supusermos que a frequência do relógio é realmente constante em relação ao tempo real). Este funcionamento será visto com detalhes mais adiante.

Relógios e pré-escalonadores

Diversos *relógios* provêm frequências de operação para as diferentes partes do microcontrolador. São basicamente emissores ou divisores de sinais de onda quadrada que determinam a frequência de operação da CPU, do sistema ADC, do acesso às memórias e de outros componentes do microcontrolador. Possíveis fontes de frequência para relógios são osciladores RC (resistor/capacitor) e de cristal.

Um conceito útil associado aos relógios é o de um *pré-escalonador*. Pré-escalonadores são divisores da frequência dos relógios que, ou diminuem de fato a frequência de um determinado relógio, ou ao menos permitem a operação de alguns componentes em frequências que são uma fração da frequência de certos relógios.

O *relógio do sistema* (system clock) provê a frequência base de operação do sistema. Outros relógios importantes são o *relógio de Entrada/Saída* (I/O clock), o *relógio ADC* (ADC clock), e os *relógios de contadores*, todos estes usados para estabelecer a frequência de operação da maioria dos mecanismos de entrada e saída. É possível escolher quais relógios devem estabelecer a frequência de operação de diferentes partes do sistema, assim como selecionar valores de pré-escalonador de forma independente. Neste estudo, foi feito uso do pré-escalonador dos relógios associados aos contadores para controlar a frequência PWM utilizada como base para o mecanismo DSP, como será visto na Seção 3.2.3.

Temporizadores ou contadores

Um *temporizador*, também chamado de *contador*, é um registro cujo valor é automaticamente incrementado de acordo com algum relógio e um valor de pré-escalonador. Um certo contador tem um tamanho fixo em bits e pode ter várias interrupções associadas a seu comportamento. Quando um contador atinge seu valor máximo, envia uma interrupção de *overflow* (transbordamento), que pode ser configurada para causar a execução de uma função, e volta a contar a partir do zero.

Contadores são importantes no contexto de processamento de sinais em tempo real no Arduino pois provêm uma forma natural de executar várias das tarefas necessárias na sequência do processamento digital de sinais. Exemplos destas tarefas são o lançamento periódico de uma função para amostragem do sinal de entrada (que enche um *buffer* de entrada) e a emissão de uma forma de onda PWM (que será abordada com mais detalhes na Seção 3.2.3) que, após uma etapa de filtragem analógica, permite conversão do sinal digital para analógico. O microcontrolador ATmega328P possui dois contadores de 8 bits e um contador de 16 bits. Cada um deles possui um conjunto diferente de funcionalidades mas todos são capazes de realizar PWM.

Pinos de entrada e saída

Microcontroladores podem receber e emitir sinais digitais através de *pinos de entrada e saída* que, no caso das placas Arduino, são convenientemente distribuídas na montagem física de forma que é fácil encaixar neles outros componentes e placas. A leitura e escrita nestes pinos é feita de acordo com frequências governadas por diferentes relógios (I/O, ADC e outros).

Em princípio, os pinos do microcontrolador são projetados para funcionar com sinais binários, representados por duas voltagens de referência (0 V e 5 V). Apesar disso, os pinos de entrada e saída vêm equipados com mecanismos úteis para amostrar sinais de entrada com tensão variante (entre os dois valores de referência), e também para emitir formas de onda que, após serem filtradas no domínio analógico, geram sinais que podem ser conectados diretamente a outros dispositivos que trabalham com áudio analógico. Estes mecanismos são, respectivamente, o conversor analógico-digital (ADC) e a modulação por largura de pulso (PWM), que serão vistos nas próximas seções.

Tipos de memória

O microcontrolador possui três áreas de memória distintas para armazenamento do programa e dados, e a tabela seguinte resume as diferentes características e propósitos de cada uma delas (retirada de ATmega328P datasheet):

Tipo	Tamanho (KB)	Persistência de dados	Tempo de escrita (clock ticks)	Duração (ciclos de escrita/leitura)
Flash	32	sim	1	10,000
SRAM	2	não	2	indeterminado
EEPROM	1	sim	30	100,000

Em geral, a memória Flash armazena o programa, a memória SRAM guarda dados voláteis utilizados durante a computação, e a memória EEPROM é usada para armazenamento de longo prazo entre sessões de trabalho. Note que a quantidade de memória SRAM representa um limite importante para muitos algoritmos DSP. Uma tabela pré-calculada com 512 amostras de 8 bits representando um período da função seno, por exemplo, representa 25% de todo o espaço disponível para trabalho. Assim, se o espaço de memória estiver escasso para uma certa aplicação, uma opção interessante pode ser armazenar dados pré-calculados diretamente na memória do programa (ou seja, incluir estes dados diretamente no código fonte).

3.2.2 Entrada de áudio: ADC

Existem várias formas de alimentar o microcontrolador com dados, sendo as mais simples a utilização dos mecanismos embutidos de comunicação serial (via pinos de entrada ou porta USB) e conversão analógico-digital (somente via pinos de entrada). A comunicação serial abstrai a transferência de dados digitais através da utilização de *buffers*, enquanto que o mecanismo ADC permite a amostragem de um valor analógico de tensão entre dois valores de referência, usando 8 ou 10 bits de resolução na conversão.

A velocidade máxima de transferência do mecanismo de comunicação serial é 115200 bits/s, o que é suficiente para transferir 14400 amostras de 8 bits em um segundo para o aparelho. Para processamento de áudio esta taxa é baixa, uma vez que corresponde a uma frequência de amostragem de 14400 Hz e à possibilidade de representação de componentes senoidais com frequências de até 7000 Hz. O mecanismo de conversão analógico-digital dos pinos de entrada permite obter frequências de amostragem mais próximas das utilizadas para representação de áudio. Por este motivo, no cenário escolhido para os testes, o sinal analógico de entrada passa por uma etapa de centralização (para garantir que toda sua amplitude, de 0 V a 5 V, seja mapeada corretamente para o domínio de representação em 8 bits) e outra de filtragem (veja a Seção 2.2), ambas analógicas, e em seguida é conectado diretamente ao pino de entrada do microcontrolador, como pode ser visto

na Figura 3.1. Assim, nenhum dispositivo externo tem que ser utilizado para amostrar o sinal, e é possível estudar o desempenho do aparelho levando em conta a amostragem, um passo crucial da cadeia de processamento digital de áudio.

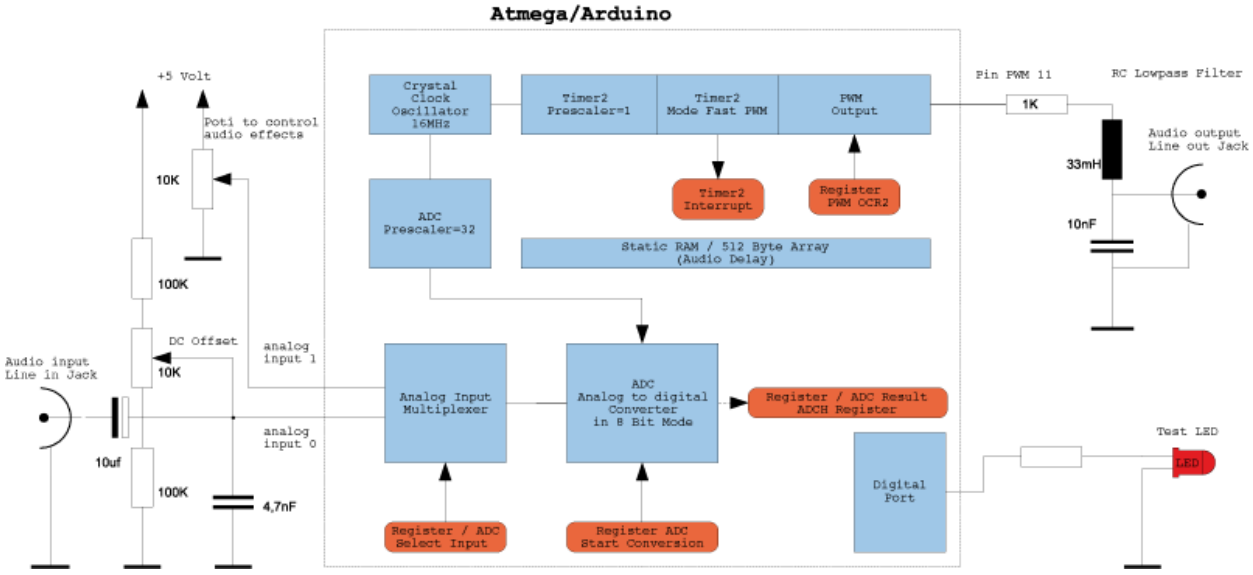

Figura 3.1: *Esquema do circuito utilizado para entrada e saída de áudio no Arduino. Na parte esquerda inferior, o sinal de áudio é conectado à entrada analógica 0 através de um circuito utilizado para calibrar o sinal de entrada. Na parte esqueda superior, um potenciômetro é ligado à porta analógica 1 que pode ser consultada eventualmente para fornecer um sinal de controle. Na parte direita superior, a saída PWMé ligada à entrada de áudio de um fone de ouvido ou caixa de som através de um filtro RLC.*

Quando uma conversão é disparada, o mecanismo ADC utiliza um circuito do tipo *sample-and-hold* que congela a tensão de entrada e a mantém em um nível constante até o fim da conversão. Esta tensão constante é então sucessivamente comparada com valores de referência para obter cada bit da aproximação, do mais significativo para o menos significativo. Se uma conversão mais rápida é desejada a precisão pode ser sacrificada, sendo que os primeiros 8 bits podem ser lidos antes que os últimos 2 sejam computados. Segundo o manual do microcontrolador (ATmega328P datasheet), o tempo de conversão toma algo entre 13 e 250 μs, dependendo de diversos parâmetros de configuração que influenciam a precisão do resultado.

O mecanismo ADC possui um relógio dedicado, o que garante que a conversão pode ocorrer independentemente das outras partes do microcontrolador. Além disso, a conversão pode ser iniciada manualmente (sob demanda), ou automaticamente (uma nova conversão é iniciada assim que a última tenha terminado).

3.2.3 Saída de áudio: PWM

Uma vez que o sinal de entrada tenha sido amostrado e processado, uma forma de convertê-lo de volta para o domínio analógico é utilizar o mecanismo de *pulse-width-modulation* (PWM) atrelado aos contadores e a alguns pinos de saída do microcontrolador, seguido de um estágio de filtragem analógica. Uma onda PWM codifica um certo valor na largura de um pulso quadrado. Para fazer isto, um *ciclo de trabalho* (*duty-cycle*) é definido como a porcentagem de tempo que a onda quadrada está em seu valor máximo em relação ao período total entre os diferentes pulsos (veja as Figuras 3.2 e 3.3). A codificação de um valor x tal que $x_1 \leq x \leq x_2$ corresponde à imposição de um *duty-cycle* com porcentagem igual a $\frac{x-x_1}{x_2-x_1}$.

O estágio final de filtragem é necessário para remover componentes de alta frequência presentes no espectro da onda de pulsos quadrados e reconstruir um sinal com banda limitada. Em nosso caso, esta filtragem é feita por um circuito RC integrador simples que está posicionado entre o pino de saída e um alto-falante comum (veja a Figura 3.1) (Nawrath).

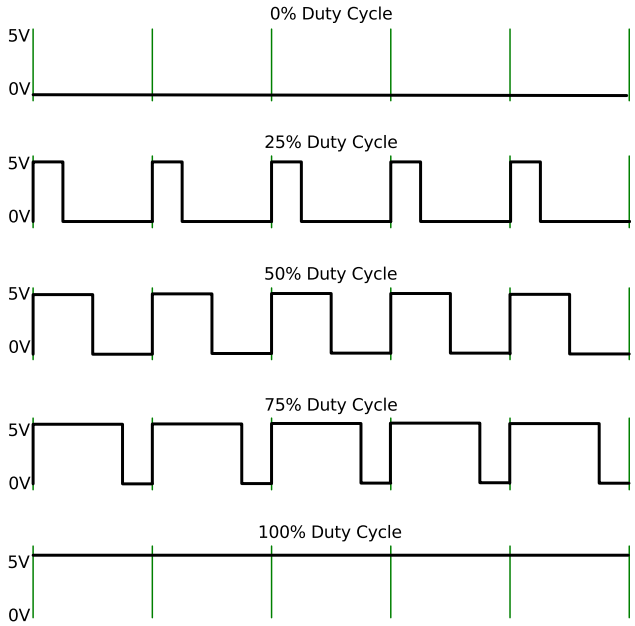

Figura 3.2: *Exemplos de ondas PWM com diferentes* duty-cycles*. Neste exemplo, o alinhamento à esquerda representa o modo* Fast PWM*.*

O mecanismo PWM pode operar em diferentes modos, que variam de acordo com a forma como o valor de referência a ser codificado se relaciona com o contador digital para gerar os valores de saída da onda modulada. No modo *Fast PWM*, o sinal de saída vale 1 no início do ciclo e 0 sempre que o valor de referência se torna menor do que o valor do contador (veja a Figura 3.4). Este modo possui a desvantagem de gerar pulsos quadrados alinhados à esquerda do ciclo PWM. Um outro modo chamado *Phase correct* está disponível e soluciona este problema com o custo de cortar a frequência de geração do sinal pela metade. Ele funciona fazendo o contador contar do valor máximo de volta ao zero ao invés de simplesmente reiniciar o valor do contador quando este atinge seu valor máximo (veja a Figura 3.5).

A frequência de saída do sinal PWM é uma função do relógio selecionado para ser usado como entrada para o contador, do valor de pré-escalonador selecionado, do tamanho do contador (em bits) e do modo PWM escolhido. Para um contador de b bits com frequência de relógio igual a f_{clock} Hz e valor de pré-escalonador igual a p, um contador operando em modo *Fast PWM* atinge seu valor máximo com uma frequência de $\frac{f_{clock}}{p \times 2^b}$ Hz. Este mecanismo provê uma forma não só de gerar um sinal de saída com frequência constante, mas também de usar a mesma infraestrutura para agendar ações periódicas, como por exemplo a obtenção de novos valores do mecanismo ADC e a sinalização de quando existe um novo bloco de amostras pronto para ser processado.

Note também que o tamanho do contador determina a resolução (em amplitude) do sinal de saída, pois o *duty-cycle* dos pulsos quadrados corresponde à razão entre o valor atual do contador e seu valor máximo possível. Veremos mais sobre a escolha de parâmetros para PWM na seção 3.2.5.

3.2.4 Processamento em tempo real

A maior restrição do processamento em tempo real é, como visto na Seção 1.1.1, a quantidade de tempo disponível para computação das amostras de saída: elas devem estar prontas para serem consumidas pelo hardware de reprodução (ou pelas próximas etapas de processamento) a tempo, caso contrário podem ser introduzidos artefatos sonoros indesejados no sinal de saída.

Para implementar este comportamento no microcontrolador, é necessário encontrar uma forma de (1) acumular amostras de entrada em um *buffer*, (2) agendar uma chamada periódica a uma função que irá manipular as amostras neste *buffer*, e (3) emitir as amostras modificadas, tudo isto com uma frequência fixa. Os componentes estão à mão: ADC para fazer a leitura de um sinal

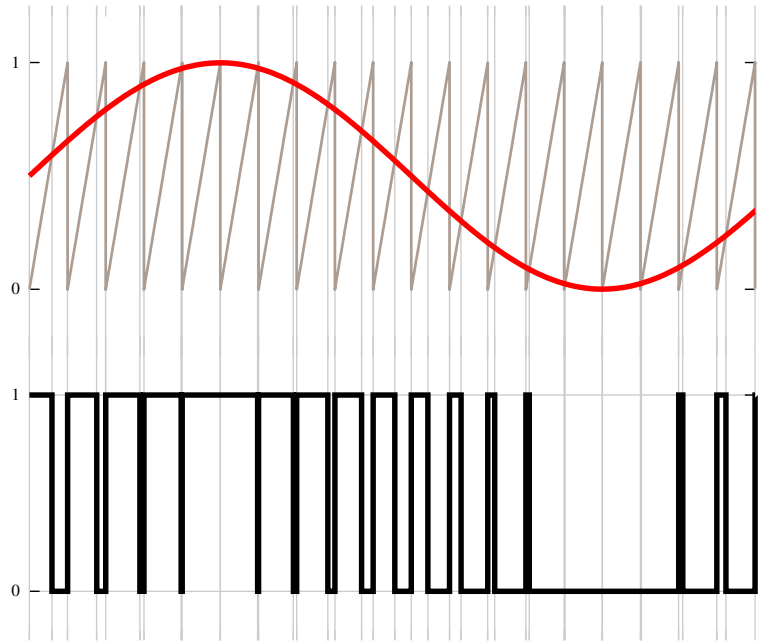

Figura 3.3: *Exemplo de codificação PWM. Na parte de cima, o sinal vermelho corresponde ao sinal que se deseja codificar, e o sinal cinza corresponde à onda dente de serra com os valores de referência (que no microcontrolador são fornecidos pelos contadores). Na parte de baixo, o sinal codificado em pulsos quadrados com larguras distintas. Note como o sinal de baixo alterna entre os valores extremos sempre que os sinais de cima se cruzam.*

de entrada, contadores e suas interrupções para executar rotinas periódicas, e PWM para emitir o sinal resultante. Adicionalmente, é possível utilizar a função `loop()` (descrita na Seção 3.1.1) para realizar o processamento do bloco de amostras sempre que um novo bloco esteja disponível.

Como vimos na Seção 3.2.3, o mecanismo PWM provê uma frequência de interrupção por *overflow* que pode ser utilizada para agendar uma função para execução periódica. No programa desenvolvido para os testes, utilizamos este mecanismo para ler periodicamente amostras de entrada do sistema ADC e acumulá-las em um *buffer* de entrada, e em seguida escrever no *buffer* de saída PWM as amostras computadas no último ciclo DSP. Nesta mesma função, sempre que o *buffer* está cheio e um novo bloco de amostras está pronto para ser processado, uma variável é alterada e a função `loop()` começa a trabalhar nas amostras.

3.2.5 Implementação

Para juntar todos os elementos descritos na seção anterior e obter um sistema de processamento de áudio em tempo real que inclui amostragem e emissão de sinal, basta apenas escolher os parâmetros certos para configurar as diferentes partes do microcontrolador.

Parâmetros ADC

Segundo a especificação do microcontrolador, o processo completo de conversão ADC demora cerca de 14.5 ciclos do relógio ADC. Se a frequência do relógio da CPU é de 16 MHz e o valor do pré-escalonador ADC é p, então o período do relógio ADC é de $p/16$ μs e o período de conversão é de $T_{\text{conv}} = 14.5 \times p/16$ μs. Os valores teóricos para o período de conversão T_{conv} (para todos os valores de pré-escalonador disponíveis) e os resultados $\tilde{T}_{\text{conv}}$ da medição do período de conversão podem ser vistos na tabela abaixo. A última coluna corresponde às frequências de conversão medidas $\tilde{f}_{\text{conv}} = 1/\tilde{T}_{\text{conv}}$.

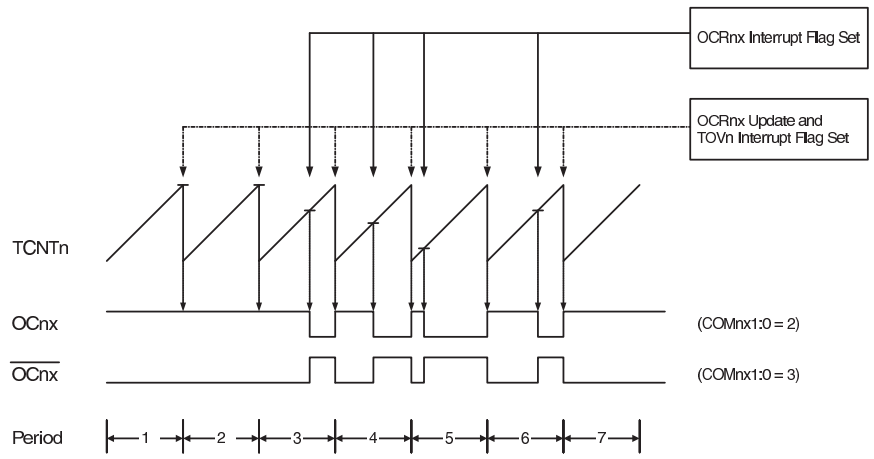

Figura 3.4: *Evolução temporal dos valores dos registros do mecanismo PWM funcionando no modo* Fast PWM*. O registro* `TCNTn` *corresponde ao valor do contador, e o registro* `OCnx` *contém o valor do pino de saída. Note como mudanças no valor de referência determinam o* duty-cycle *de cada período de onda.*

Pré-escalonador ADC	T_{conv} (μs)	$\tilde{T}_{\text{conv}}$ (μs)	$\tilde{f}_{\text{conv}}$ (≈KHz)
2	1.8125	12.61	79.302
4	3.625	16.06	62.266
8	7.25	19.76	50.607
16	14.5	20.52	48.732
32	29	34.80	28.735
64	58	67.89	14.729
128	116	114.85	8.707

Estas medições foram feitas utilizando a função `micros()` da API do Arduino, que tem uma resolução de cerca de 4 μs. Isto pode explicar parte da diferença entre os valores medidos e os valores esperados para os valores mais baixos de pré-escalonador. Na medição foi utilizada uma aproximação de 8 bits, e para uma aproximação de 10 bits pode-se esperar um aumento significativo (por volta de 25%) no tempo de conversão.

PWM

A partir do que foi visto na Seção 3.2.3, em uma CPU operando a 16 MHz, um contador de 8 bits e valor de pré-escalonador igual a p possui uma frequência de *overflow* de $f_{\text{overflow}} = 10^6/p \times 2^4$ Hz. Abaixo podemos ver uma tabela com as frequências de incremento f_{incr} e de interrupção por *overflow* f_{overflow} para todos os valores possíveis de pré-escalonador de um contador de 8 bits:

Pré-escalonador PWM	f_{incr} (KHz)	f_{overflow} (Hz)
1	16.000	62500
8	2.000	7812
32	500	1953
64	250	976
128	125	488
256	62,5	244
1024	15,625	61

A escolha dos valores de pré-escalonador PWM e ADC determinam a frequência de amostragem do sistema DSP implementado. Se o pré-escalonador ADC for configurado de forma que o período

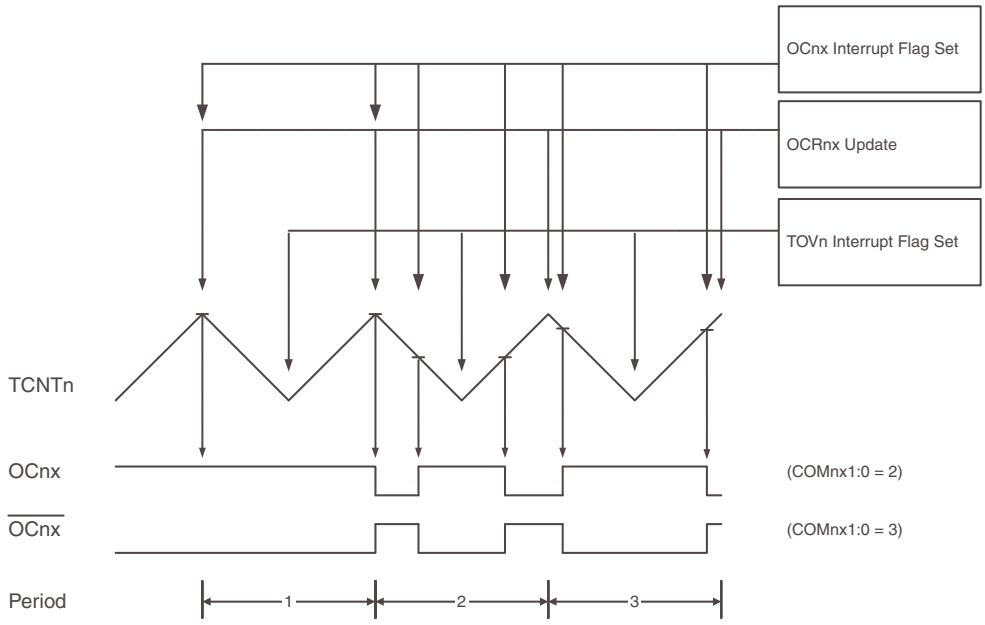

Figura 3.5: *Evolução temporal dos valores dos registros do mecanismo PWM funcionando no modo* Phase Correct*. O registro* `TCNTn` *corresponde ao valor do contador, e o registro* `OCnx` *contém o valor do pino de saída. Note como mudanças no valor de referência determinam o* duty-cycle *de cada período de onda.*

de conversão ADC seja menor do que a frequência de interrupção por overflow do PWM, e se a leitura da entrada for sincronizada com a escrita na saída, então a frequência de interrupção por overflow do PWM corresponde exatamente à frequência de amostragem do sistema DSP. Isto será visto com mais detalhes na próxima seção.

Para o mecanismo PWM, a escolha foi utilizar o modo Fast-PWM, com um contador de 8-bits com valor de pré-escalonador igual a 1. Isto fornece uma taxa de amostragem de 62500 Hz, suficiente para representar o espectro audível. É possível diminuir artificialmente esta frequência executando o ciclo DSP completo (amostragem, processamento e emissão do sinal) não em todas mas apenas em uma fração das interrupções. Nos testes executados, a escolha foi de cortar a frequência de amostragem pela metade, para 31250 Hz, correspondendo a um período de amostra de 32 μs. Esta escolha privilegia a vantagem de se ter disponível o dobro do tempo de computação, vantagem esta aparentemente maior do que a desvantagem de se perder os últimos 4,4 KHz do espectro audível.

Juntando os pedaços

Tendo escolhido o tamanho do contador e do pré-escalonador PWM, falta apenas escolher os valores dos parâmetros ADC. Como já foi dito, é suficiente escolher valores que garantam que o período de conversão ADC é menor do que o período desejado de uma amostra. Para os testes, foi escolhido utilizar a conversão de 8 bits para obter uma resolução igual à da saída PWM e um período menor de conversão (comparado com o período da conversão de 10 bits). Também foi escolhido um valor de pré-escalonador ADC igual a 8, correspondendo a um período de conversão medido de $19,76\mu$s que, quando comparado com o período de amostra igual a 32 μs garante que a conversão terminará antes que o mecanismo ADC tenha que iniciar uma nova conversão, deixando tempo livre para outros processamentos.

Abaixo, pode-se ver uma reprodução do pedaço de código correspondente à função de interrupção por overflow, executada sempre que o contador é reiniciado, que corresponde ao mecanismo controlador do ciclo DSP no sistema implementado. O vetor `x` é o *buffer* de entrada, `ADCH` é uma variável que aponta para o registro ADC que contém a amostra de entrada, `OCR2A` aponta para o registro de saída PWM e `y` é o vetor de saída. Note que boa parte do código corresponde a cálculos

de índices dos vetores.

```
1  /**************************************************************************
2   * Funcao de interrupcao por overflow do Timer 2.
3   *
4   * As variaveis `writeind` e `readind` sao inteiros de 8 bits, portanto seu
5   * valor maximo e' 255. Essa propriedade e' utilizada para fazer a leitura e
6   * escrita circular dos vetores de entrada e saida, `x` e `y`, cujo tamanho e'
7   * exatamente 256.
8   *
9   **************************************************************************/
10 ISR(TIMER2_OVF_vect)
11 {
12     /*
13      * Divisao da frequencia de amostragem por 2.
14      */
15     static boolean div = false;
16     div = !div;
17     if (div){
18
19         /*
20          * Leitura da entrada com conversao de tipo. ADCH e' o registrador que
21          * contem o resultado da ultima conversao ADC.
22          */
23         x[writeind] = (int16_t) 127 - ADCH;
24
25         /*
26          * Escrita da amostra de saida no registrador PWM, com ajuste de
27          * offset e conversao de tipo.
28          */
29         OCR2A = (uint8_t) 127 + y[(writeind-BLOCK_SIZE)];
30
31         /*
32          * Verifica se um novo bloco de amostras esta cheio, calculando a
33          * seguinte condicao: (writeind mod BLOCK_SIZE) == 0.
34          */
35         if ((writeind & (BLOCK_SIZE - 1)) == 0) {
36           readind = writeind - BLOCK_SIZE;
37           dsp_block = true;
38         }
39
40         /*
41          * Incremento do indice de escrita.
42          */
43         writeind++;
44
45         /*
46          * Inicia a proxima conversao.
47          */
48         sbi(ADCSRA,ADSC);
49     }
50 }
```

Note que no passo 3 (linhas 11-14) é feito um teste para determinar se o valor do índice de entrada é um múltiplo do tamanho do bloco e, em caso positivo, o valor do índice de leitura `rind` e da variável `dsp_block` são atualizados para sinalizar que existe um novo bloco de amostras disponível para processamento. Enquanto isso, a função `loop()` é executada concorrentemente e eventualmente iniciará o trabalho nas amostras. Ao final, o índice dos *buffers* é incrementado (linhas 16 e 17) e um registro é alterado para iniciar uma nova conversão ADC (linha 19).

O código a seguir mostra uma implementação de algoritmo simples de convolução no domínio

do tempo dentro da função `loop()`. Cada implementação substitui somente o pedaço de código referente ao processamento do bloco de amostras.

```
/**
 * LIMIT(x)
 * Limita 'x' ao intervalo [-127,127].
 */
#define LIMIT(x) \
  if (x < -127)  \
    x = -127;    \
  if (x > 127)   \
    x = 127;

/**
 * TMOD(x, index, len)
 * Leitura circular do indice 'index' de um vetor 'x' de tamanho 'len',
 * potencia de 2.
 */
#define TMOD(x, index, len) x[(index)&(len-1)]

void loop()
{
  /* aguarda um novo bloco de amostras */
  while (!dsp_block);

  /* variaveis para contagem de tempo */
  static unsigned long elapsed_time = 0;
  unsigned long start_time = micros();

  /* processamento do bloco */
  uint16_t maxind = readind+BLOCK_SIZE;
  for (uint8_t n = readind; n < maxind; n++) {

    int16_t yn = 0;
    uint8_t order = 0;
    for (uint8_t i = 0; i <= order; i++) {
        yn += TMOD(x, n, BUFFER_SIZE) * 0.33;
    }

    LIMIT(yn);
    TMOD(y, n, BUFFER_SIZE) = yn;
  }
  elapsed_time += micros() - start_time;
  count++;

  dsp_block = false;
}
```

3.3 Resultados e discussão

A infraestrutura descrita na seção anterior foi utilizada para executar os algoritmos de convolução no domínio do tempo, síntese aditiva e FFT, descritos na Seção 2.2. As próximas seções descrevem particularidades nas implementações destes métodos no Arduino, bem como os diversos resultados obtidos variando detalhes de implementação.

3.3.1 Síntese aditiva

O primeiro experimento pretende determinar o número máximo de osciladores que podem ser utilizados para realizar síntese aditiva em tempo real no modelo de Arduino testado. A cada ciclo DSP, o algoritmo de síntese aditiva é executado utilizando um número determinado de osciladores, e a média do tempo de síntese é calculada sobre 1000 medições.

Durante a implementação do experimento, uma primeira tentativa utilizando a função `sin()` da API mostrou-se inviável em tempo real. Por causa disso, a alternativa foi focar em implementações com osciladores por consulta a tabela. Os tamanhos de bloco utilizados foram de 32, 64 e 128 amostras, e a utilização de blocos maiores não foi possível por causa da limitação de memória do dispositivo. O número de osciladores foi incrementado até que o período do ciclo DSP fosse excedido.

Estruturas de laço

O primeiro resultado obtido tem a ver com o uso de estruturas de laço. Pelo fato de laços em geral utilizarem incremento e testes de valores de variáveis em cada iteração, o uso de uma estrutura de laço pode ter forte influência na quantidade máxima de osciladores que podem ser utilizados na síntese aditiva em tempo real no Arduino.

Qualquer algoritmo de processamento de áudio que trabalhe sobre um bloco de amostras possui ao menos uma estrutura de laço, aquela que itera sobre todas as amostras do bloco. Este laço pode ser eliminado, porém ao custo de ter que recompilar o código sempre que o tamanho do bloco é alterado, o que não é muito conveniente.

Em geral, outros laços são utilizados, como por exemplo na síntese aditiva ao calcular o valor dos diversos osciladores e somar seus resultados em uma variável. A alternativa de remover este laço interno, escrevendo explicitamente cada parcela da soma dos osciladores, foi investigada, e as Figuras 3.6 e 3.7 mostram o valor máximo de osciladores computável em tempo real fazendo uso de um laço, e fazendo uso de código repetido, respectivamente. Ao remover o laço interno, foi possível aumentar a quantidade de osciladores de 8 para até 13 ou 14, dependendo do tamanho do bloco.

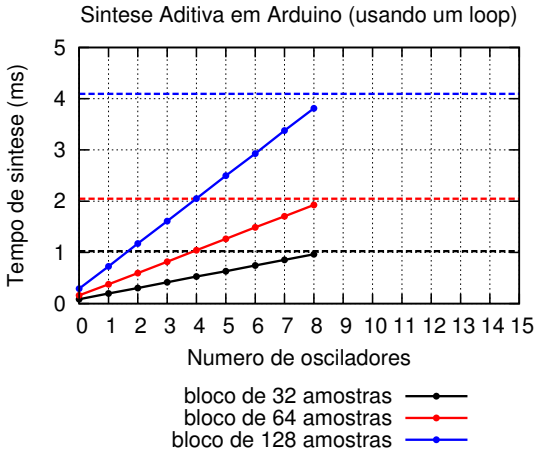

Figura 3.6: *Tempo de síntese para diferentes números de osciladores e tamanhos de bloco usando laços e* bit-shifting*. As linhas pontilhadas mostram o limite de tempo imposto pelo período teórico do ciclo DSP.*

Cálculo do valor dos osciladores

Fica evidente que as menores diferenças de implementação podem ter um impacto grande no desempenho dos algoritmos. O cálculo do valor dos osciladores é uma parte fundamental da síntese aditiva, e portanto é interessante verificar o impacto que diferentes restrições causam no desempenho geral do algoritmo. Dois parâmetros são utilizados para calcular o valor de cada oscilador: amplitude e fase. Em uma implementação com consulta a tabela, a fase é determinada através da atualização do índice de leitura da tabela com os valores do seno, e em seguida a amplitude é multiplicada pelo valor resultante da consulta.

Operações de ponto flutuante são extremamente custosas no microcontrolador utilizado, e portanto foram comparadas 3 formas distintas de realizar a multiplicação da amplitude: (1) utilizando

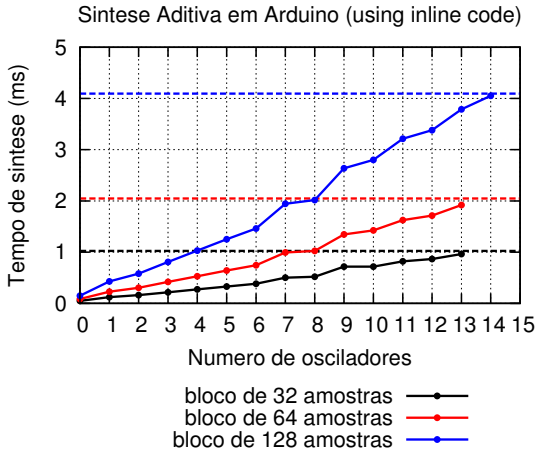

Figura 3.7: *Tempo de síntese para diferentes números de osciladores e tamanhos de bloco, usando código* inline.

uma multiplicação e uma divisão por inteiros, (2) usando apenas uma divisão por inteiros, e (3) usando uma operação de *bit-shifting* com valor variável para permitir a multiplicação e divisão por potências de 2. Cada uma destas três abordagens impõe restrições diferentes para os valores de amplitude que podem ser utilizados nos osciladores, o que também restringe o conjunto de sinais que podem ser gerados por cada uma delas. Apesar disso, o interesse deste experimento é evidenciar como pequenas escolhas de implementação podem alterar o desempenho, mesmo que isto também implique em restrições no tipo de sinal gerado.

A Figura 3.8 mostra o tempo utilizado pela síntese aditiva usando as variantes descritas acima. Ao utilizar operações de mais baixo nível (que permitem resultados menos precisos) e código sem laços, foi possível aumentar o número de osciladores de 3 (quando utilizando 2 operações sobre inteiros e laços) para 15 (ao usar *bit-shifting* variável e código sem laços).

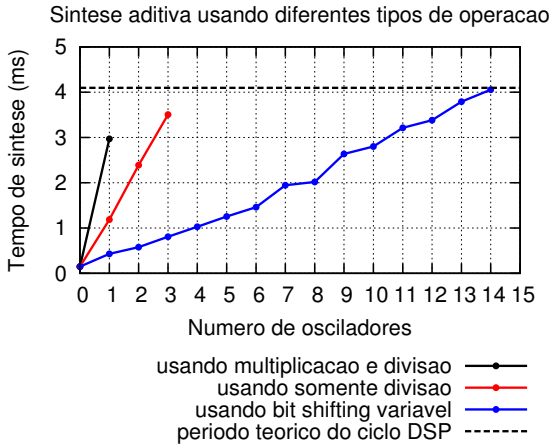

Figura 3.8: *Tempo de processamento para o algoritmo de síntese aditiva com tamanho de bloco de 128 amostras, usando diferentes números e tipos de operação.*

3.3.2 Convolução no domínio do tempo

O segundo experimento tenta esclarecer qual é o tamanho máximo de um filtro FIR que pode ser aplicado em tempo real sobre um sinal de entrada utilizando o algoritmo de convolução no domínio do tempo. Seguindo as lições aprendidas no primeiro experimento, o laço de filtragem

foi implementado utilizando tipos diferentes de operação para multiplicar cada coeficiente pelo valor da amostra correspondente: (1) usando uma multiplicação e uma divisão de inteiros, (2) usando *bit-shifting* variável, e (3) usando *bit-shifting* constante. Os resultados para cada uma destas implementações podem ser vistos nas Figuras 3.9, 3.10 e 3.11 respectivamente. Este experimento foi executado com uma taxa de amostragem de 31250 Hz e tamanhos de bloco de 32, 64, 128 e 256 amostras.

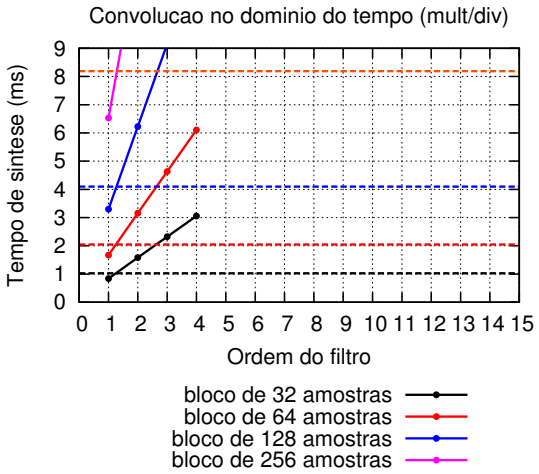

Figura 3.9: *Convolução no domínio do tempo usando duas operações sobre inteiros.*

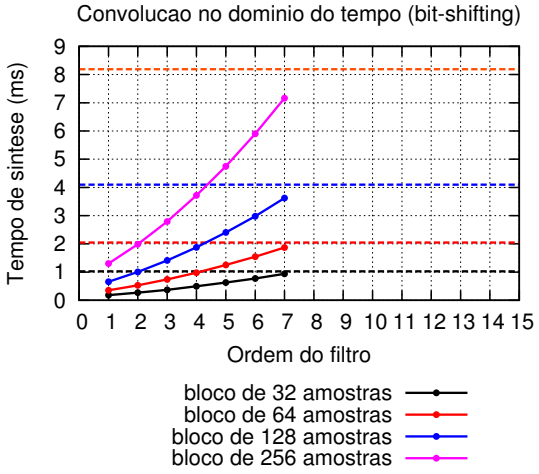

Figura 3.10: *Convolução no domínio do tempo usando* bit-shifting *variável.*

Os resultados novamente mostram que pequenas diferenças de implementação causam grandes diferenças no poder computacional. Na utilização de divisão de inteiros, a ordem máxima obtida para o filtro foi 1, enquanto usando um *bit-shifting* variável a ordem máxima foi de 7 e com *bit-shifting* constante pudemos atingir uma ordem de 13 ou 14, dependendo do tamanho do bloco, comparável ao número máximo de osciladores da síntese aditiva.

Um exemplo de algoritmo de conjunto de filtros que pode ser implementado utilizando somente operações de *bit-shifting* são os chamados filtros *Moving Average* que calcula médias de um número de amostras que seja potência de 2. Na Figura 3.12 é possível ver a resposta em frequência deste tipo de filtro para ordem 1, 2, 4 e 8.

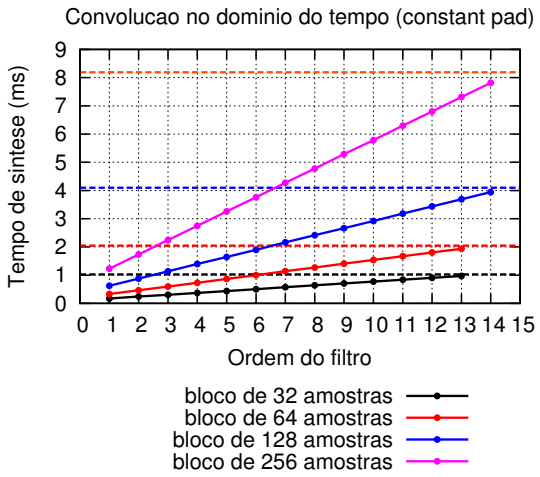

Figura 3.11: *Convolução no domínio do tempo usando* bit-shifting *constante.*

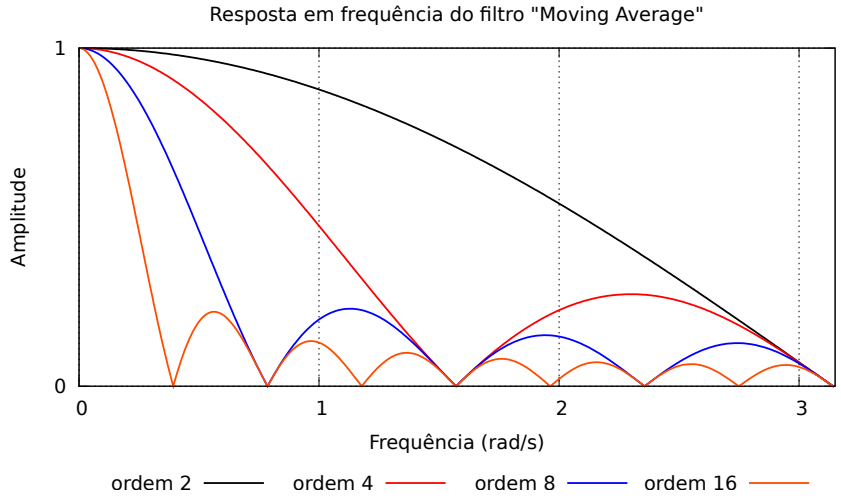

Figura 3.12: *Resposta em frequência de filtros* Moving Average *com ordem igual a diferentes potência de 2.*

3.3.3 FFT

O terceiro experimento procura saber qual é o comprimento máximo de uma FFT que pode ser computado em tempo real dentro do Arduino, considerando uma FFT por bloco de processamento. Neste caso, a escolha foi de avaliar uma implementação padrão da FFT, sem modificações ou adaptações às especificidades do microcontrolador.

Na primeira tentativa ficou evidente que calcular uma FFT usando a taxa de amostragem utilizada nos outros experimentos (31250 Hz) é inviável, e por isso foi necessário alterar os parâmetros de configuração do microcontrolador para obter um período de ciclo DSP mais longo (para o mesmo número de amostras) até que o cálculo da FFT fosse viável. Medindo-se o tempo necessário para computar a FFT de um certo número (fixo) de amostras, é possível determinar qual a frequência de amostragem máxima para viabilizar uma FFT por bloco; no caso de um bloco de 256 amostras este tempo é de aproximadamente 428,27 μs, o que permite uma frequência máxima de aproximadamente 2335 Hz. Assim, aumentando o valor do pré-escalonador PWM para 32, foi possível atingir uma taxa de amostragem de aproximadamente 1953 Hz, de modo a viabilizar ao menos este tamanho de bloco.

A Figura 3.13 mostra o tempo da análise da FFT a uma frequência de amostragem de 1953 Hz para diferentes tamanhos de bloco. Pode-se ver que neste cenário o tamanho máximo de bloco para o qual a FFT pode ser computada em tempo real no mecanismo DSP implementado foi de 256 amostras. É claro que este era o esperado uma vez que a frequência de amostragem escolhida era pequena o suficiente apenas para garantir que uma FFT de 256 amostras fosse viável em tempo real. Note que, neste cenário, apesar de ser possível executar uma FFT para blocos de tamanho menor ou igual a 256 amostras, não resta muito tempo do período do ciclo DSP para utilizar estes resultados para alguma outra computação.

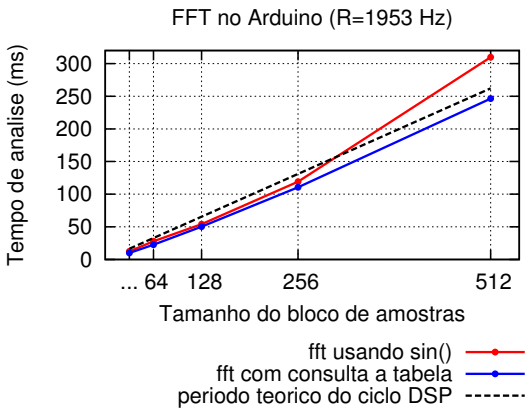

Figura 3.13: *FFT sendo executada a 1953 Hz para diferentes tamanhos de bloco. A curva vermelha indica o tempo utilizado por uma implementação utilizando a função* `sin()` *da API, e a curva azul indica a utilização de consulta a tabela.*

3.3.4 Discussão

Dos resultados dos experimentos, fica óbvio que pequenos detalhes de implementação, como a escolha dos tipos de dados e quantidade e tipo de operações utilizadas, fazem uma grande diferença na quantidade de computação, como descrito nas Seções 3.3.1 e 3.3.2. Multiplicação e divisão de inteiros, por exemplo, tomam um tempo cerca de duas vezes maior do que soma de inteiros. A quantidade de laços também mostrou influenciar no desempenho. Na Seção 3.3.1 foi possível praticamente dobrar o número de osciladores que podem ser utilizados apenas substituindo um laço por código *inline*.

Estes experimentos podem servir de ilustração para o tipo de preocupação que deve-se ter em mente ao implementar tarefas de processamento de áudio em tempo real no Arduino, além de servir como referência para as limitações na complexidade destas tarefas quando o funcionamento em tempo real é desejado.

Capítulo 4

Processamento de áudio em tempo real em GPU

No capítulo anterior, as limitações do Arduino serviram para evidenciar as dificuldades e possibilidades de lidar com processamento de áudio em tempo real num nível relativamente baixo, quando comparado com o que será visto neste e no próximo capítulos. Dos dispositivos estudados neste trabalho, as placas de processamento do tipo GPU se encontram num extremo oposto ao do Arduino em termos de poder computacional (número de processadores, frequência de operação, quantidade e velocidade de acesso à memória), consumo de energia, custo, entre outros fatores. Na GPU, centenas de processadores bem escalonados possibilitam o paralelismo no níveis das instruções e dos dados.

Na Seção 4.1, uma descrição da evolução da *pipeline* gráfica e da construção do paradigma de *programação de propósito geral para GPU* introduz o tema e prepara terreno para que seja possível, na Seção 4.2, explicar como esta estrutura pode ser aproveitada no processamento de áudio em tempo real. Algumas das técnicas e algoritmos apresentados no Capítulo 2 são intrinsecamente paralelizáveis, e a apresentação e discussão dos resultados da exploração deste paralelismo são o tema da Seção 4.3.

4.1 Programação de propósito geral usando GPU

Como comentado na Seção 1.2.2, no início do desenvolvimento das placas gráficas a existência de funções fixas para propósitos específicos em diversos estágios da *pipeline* forçava a necessidade de adaptação da solução de um problema a uma estrutura bastante engessada. A evolução das técnicas de processamento gráfico ao longo de mais de 10 anos trouxe bastante flexibilidade para a utilização deste tipo de circuito para a computação de propósito geral. A utilização de hardware especializado para implementação de funções fixas específicas para o processamento de imagens tridimensionais é então complementada com a implementação de paralelismo de dados e de tarefas. Assim, várias tarefas podem ser executadas ao mesmo tempo, cada uma realizando uma mesma operação em paralelo em todo um conjunto de dados. Isto resulta em uma arquitetura que consegue atingir alta intensidade computacional e alta taxa de fluxo de dados.

Com o amadurecimento do campo de pesquisa, as técnicas se tornaram mais sofisticadas e as comparações com os trabalhos fora do campo da GPU mais rigorosas. No nível do software, esta maturidade é evidenciada pela construção de aplicações reais nas quais a GPU demonstra possuir grandes vantagens. No nível do hardware, houve a transformação da GPU em um processador paralelo totalmente programável com funções fixas adicionais para propósitos especiais, como por exemplo alguns tipos de transformação linear e funções trigonométricas. Os conjuntos de funcionalidades e de instruções dos processadores de vértices e de fragmentos (veja a Seção 4.1.1) têm aumentado e convergido. Por causa disso, a *pipeline* baseada em tarefas paralelas tem tido uma tendência a ser remodelada para uma estrutura baseada em uma única unidade programável unificada, de forma que é possível atingir níveis mais complexos de paralelismo de tarefas e de dados.

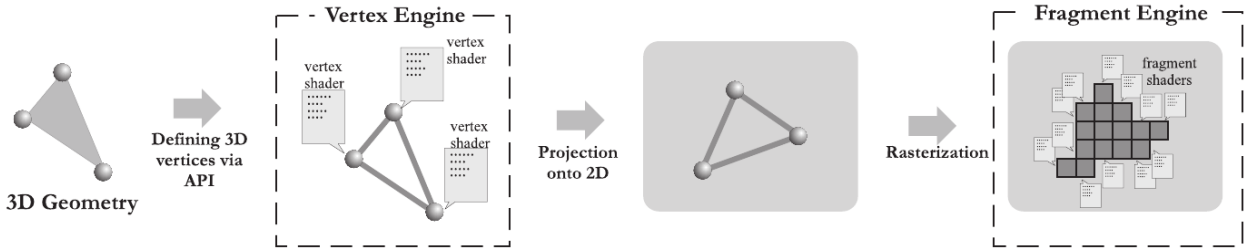

Figura 4.1: *Operações envolvidas na* pipeline *gráfica tradicional. Uma descrição abstrata de uma cena tridimensional passa primeiro pelo processamento de vértices, em seguida pela projeção da cena em uma janela bidimensional e finalmente pelo cálculo dos fragmentos que são transformados em pixels.*

Isso faz com que, cada vez mais, os programadores possam se concentrar em apenas uma unidade programável contando ainda com técnicas básicas de computação paralela como *map*, *reduce*, *scatter*, entre outras (Owens *et al.*, 2008, 2007).

Mesmo com esses avanços, programar para a GPU não se trata somente de aprender uma nova linguagem, mas de utilizar um modelo de computação diferente do tradicionalmente utilizado na programação para CPU, de forma a adaptar o problema à arquitetura utilizada. Para compreender este modelo de computação, é necessário ter uma ideia do funcionamento da *pipeline* de uma GPU.

4.1.1 Processamento gráfico tradicional

O processamento de gráficos tridimensionais requer grande capacidade computacional paralela e alta taxa de fluxo de dados para exibição dos resultados da computação em tempo real. Para atender a esta demanda, a *pipeline* tradicional para processamento de gráficos consiste em uma sequência de estágios de computação, ao longo dos quais certas funções fixas são aplicadas aos dados de entrada. Tipicamente, o primeiro estágio recebe conjuntos de triângulos e texturas, os estágios intermediários realizam transformações nestes dados e o último estágio gera imagens para exibição na tela (veja a Figura 4.1). Os estágios de computação da *pipeline* tradicional são:

- **Processamento de vértices:** Os vértices dados como entrada para a GPU são mapeados para uma posição na tela, de acordo com interações com as fontes de luz da cena e propriedades óticas dos objetos. A partir dos vértices, são construídos triângulos, primitivas fundamentais suportadas pelo hardware das GPUs.

- **Geração de fragmentos (ou "rasterização"):** Consiste no mapeamento dos triângulos construídos na etapa anterior para pixels. Cada triângulo gera um *fragmento* (outro tipo primitivo) para cada pixel que cobre na tela. A cor de cada pixel pode ser computada a partir de vários fragmentos.

- **Processamento de fragmentos:** Cada fragmento gerado é processado utilizando texturas armazenadas na memória para determinar sua cor final.

- **Composição da imagem:** Os fragmentos são combinados para determinar a cor final de cada pixel. Em geral, mantém-se a cor do fragmento mais próximo da tela.

Os estágios de processamento de vértices e de fragmentos são altamente paralelizáveis pois a computação de um pixel associado a um vértice não depende de outros pixels, assim como a determinação da cor final de um fragmento não depende de informação sobre outros fragmentos. Nas primeiras gerações de GPUs, as operações disponíveis nesses estágios não eram programáveis, mas apenas configuráveis. Era possível, por exemplo, escolher a posição e cor dos vértices e fontes de luz, mas não o modelo de iluminação que determinava sua interação. O passo chave para a generalização do uso da GPU está na ideia da substituição das funções fixas nesses estágios por programas especificados pelo programador para serem aplicados em cada vértice e fragmento.

Enquanto as primeiras gerações de GPUs podiam ser descritas como uma *pipeline* de funções fixas com adição de elementos programáveis, as novas gerações são melhor caracterizadas como *pipelines* programáveis com suporte de unidades de funções fixas (veja as Figuras 4.2 e 4.3) (Owens *et al.*, 2008).

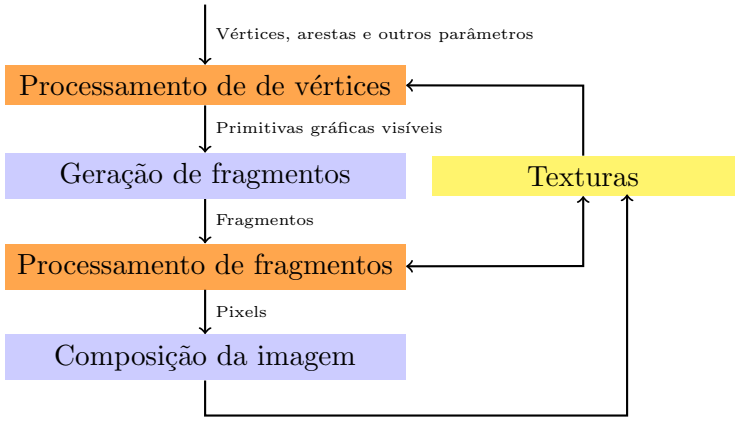

Figura 4.2: *Esquema da* pipeline *gráfica tradicional. Cada estágio possui uma função específica e não pode ser programado arbitrariamente. Note os tipos de dados que fluem de um estágio para o outro, e os possíveis sentidos de fluxo de dados no acesso à memória de texturas*

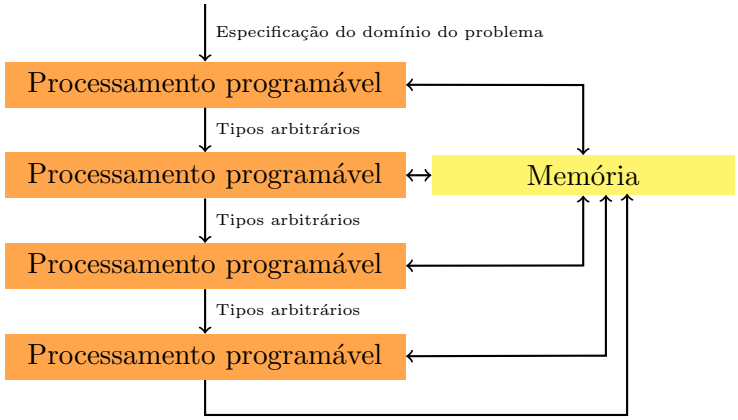

Figura 4.3: *Tendência da estrutura da* pipeline *para processamento de propósito geral. A execução de cada estágio pode ser programada arbitrariamente e a memória pode ser lida e escrita livremente em cada estágio*

Na *pipeline* gráfica, o conjunto de vértices de entrada e a memória de texturas podem ser acessados e alterados em cada estágio de processamento. Por causa do paralelismo de tarefas, o desempenho da *pipeline* depende sempre da tarefa mais lenta. Este fato, somado à natureza das operações de cada estágio, tornou possível flexibilizar muito mais o processamento de fragmentos do que o processamento de vértices. Apesar disso, a tendência observada é de convergência dos conjuntos de instruções e funcionalidades, e unificação da unidade programável.

O desenvolvimento deste tipo de arquitetura de *pipeline* para processamento gráfico e o aumento da flexibilidade das unidades programáveis motivou o desenvolvimento de um modelo de programação que tem como objetivo generalizar a expressão do paralelismo e das estruturas de comunicação presentes em estruturas computacionais similares à do processamento gráfico. Este modelo se baseia na ideia de processamento de "fluxos" de dados e será descrito na próxima seção.

4.1.2 Processamento de fluxos de dados

Num primeiro momento, o modelo de programação de propósito geral utilizando GPU consistia numa subversão do modelo de computação gráfica. O programador especificava uma primitiva

geométrica que cobrisse o domínio de computação de interesse e a submetia à estrutura da *pipeline* para manipular os dados, tendo muitas vezes que se utilizar de truques para realizar operações que não eram suportadas ou cujo consumo de tempo não era otimizado. Com os avanços das arquiteturas e dos modelos de computação paralela, hoje o programador define o domínio de interesse como uma grade estruturada de *threads* e determina operações matemáticas arbitrárias e acesso aleatório de leitura e escrita em memória global com bastante flexibilidade (Owens *et al.*, 2008).

Como visto na Seção 4.1.1, a *pipeline* gráfica é tradicionalmente estruturada como uma sequência de estágios de computação conectados por um fluxo de dados que atravessa todos os estágios. Esta estrutura, consequente das propriedades matemáticas do processamento tridimensional e das possibilidades e limitações da construção de processadores e memória, é expressada por um modelo computacional chamado **processamento de fluxos de dados** (em inglês, *stream processing*), que captura a localidade do processamento e expõe o paralelismo e alguns padrões de comunicação de uma aplicação (Kapasi *et al.*, 2003).

No modelo de processamento de fluxos de dados, todos os dados são representados através de **fluxos** (*streams*), definidos como conjuntos ordenados de dados do mesmo tipo. A computação nos fluxos de dados é feita através das chamadas **funções de *kernel***, funções que operam em um ou mais fluxos de entrada e que possuem um ou mais fluxos de dados como saída. A saída de um *kernel* deve ser uma função somente de sua entrada. Além disso, dentro de uma função de *kernel* a computação sobre um elemento do fluxo não deve depender da computação de outros elementos. Este modelo permite que a computação de um *kernel* sobre vários elementos de um fluxo de dados seja realizada em paralelo e que várias funções de *kernel* sejam concatenadas formando uma cadeia de composição de funções sobre um ou mais fluxos (Owens, 2005).

A estrutura da *pipeline* gráfica descrita na seção 4.1.1 pode ser vista como uma restrição do modelo de fluxo de dados. Sob o modelo de processamento de fluxos de dados, a implementação de uma *pipeline* gráfica deste tipo envolveria simplesmente a escrita de um *kernel* para cada estágio de processamento da *pipeline* e a conexão da saída de uma função de *kernel* com a entrada de outra, na ordem apresentada.

A eficiência do modelo de processamento de fluxos de dados está não só na possibilidade de uma função de *kernel* processar diversos elementos do fluxo em paralelo, mas também no fato de que diversas funções de *kernel* podem ser calculadas em paralelo permitindo a implementação de paralelismo de tarefas. Além disso, um bom balanço entre funções de *kernel* completamente programáveis e funções fixas (que dependem do domínio do problema e podem ser implementadas em hardware especializado) pode aumentar ainda mais a eficiência da computação.

O estudo do processamento de fluxos de dados é interessante ainda pois representa um modelo alternativo de computação que pode levar em conta as diferenças da velocidade de avanço das tecnologias de processadores e de memórias. A eficiência de um processador é medida em termos de capacidade computacional (operações lógicas ou aritméticas por unidade de tempo), e a eficiência das memórias de acesso aleatório é medida em termos de banda (quantidade de dados transferida por unidade de tempo) e latência (tempo de percurso de um bloco de dados desde a origem até o destino). Enquanto a capacidade computacional dos processadores aumenta cerca de 71% a cada ano, a banda de transferência das memórias de acesso aleatório cresce somente 25%, e a latência diminui apenas 5%. As questões "computação *versus* comunicação", "latência *versus* banda" e "consumo de energia" são hoje questões centrais para o desenvolvimento de processadores (Owens, 2005).

Estudos sobre a eficiência e complexidade de modelos de processamento de fluxos podem ser encontrados desde a década de 1980, quando do aparecimento das primeiras placas de vídeo com aceleração para o processamento de gráficos. Já naquela época foi mostrado, por exemplo, que ordenação é um problema difícil neste modelo computacional. Também foram estabelecidos limites inferiores e superiores para diversos outros problemas além de ser dada uma caracterização do poder computacional em função do número de registros e operações disponíveis (Fournier e Fussell, 1988). Mais recentemente, foi mostrado também que o número de "passagens" para a computação da mediana em um modelo de processamento de fluxos de dados é proporcional a $O(\log N)$, em

oposição à complexidade de $O(N)$ no modelo tradicional, se N é o tamanho da entrada (Guha *et al.*, 2003).

É interessante notar que a *pipeline* gráfica tradicional dos anos 90, que veio como solução para as necessidades do processamento de imagens tridimensionais (alta intensidade computacional, alto grau de paralelismo e alta taxa de fluxo de dados), acabou motivando esta abordagem mais abstrata através do modelo de processamento em fluxos de dados, que por sua vez voltou a influenciar o projeto e a produção das GPUs em direção à computação de propósito geral. O balanço entre o uso de funções fixas ou funções de *kernel* completamente programáveis, que também pode ser vista como um balanço entre o número e organização de transistores utilizados para controle da computação ou para o processamento propriamente dito, continua sendo uma questão central no desenvolvimento da GPU. De qualquer forma, a abstração e eficiência trazidas pelo modelo de processamento de fluxos de dados tem permitido a adaptação de diversos problemas para o domínio de aplicação da GPU.

4.1.3 Plataformas e arcabouços

Diversas iniciativas teóricas e práticas foram desenvolvidas com o objetivo de permitir a especificação de programas no modelo de processamento de fluxos de dados (Buck *et al.*, 2004; Kapasi *et al.*, 2002; Patrick *et al.*, 2003). Apesar disso, duas abordagens fomentadas pela indústria de placas GPU têm hoje maior adesão: CUDA[1], arquitetura desenvolvida pela Nvidia, e OpenCL[2], um arcabouço genérico com padrão aberto, mantido por um consórcio de empresas sem fins lucrativos.

Mais recentemente, o uso da GPU especificamente para processamento de áudio tem sido estudado (Gallo e Tsingos, 2004; Moreland e Angel, 2003; Savioja *et al.*, 2011). Por exemplo, ao medir o desempenho da GPU contra o da CPU, Tsingos et. al. mostraram que para diversas aplicações é possível conseguir aceleração com fatores de 5 a 100 vezes (Tsingos *et al.*, 2011). Ali, são estudadas duas abordagens diferentes para implementações de aplicações DSP comuns: o mapeamento de problemas para a *pipeline* gráfica, e a computação de propósito geral para GPU.

Uma alternativa, com a qual esta pesquisa manteve um contato mais próximo ao longo de seu desenvolvimento, corresponde à ideia de implementar as funcionalidades do CUDA no software Pure Data[3](também conhecido pelo apelido "Pd"), amplamente utilizado para processamento de áudio em tempo real, para possibilitar a interação com placas GPU da Nvidia (Henry, 2011). Uma vez que esta estrutura esteja implementada, será possível utilizar a interface gráfica do Pd para especificar rotinas arbitrárias de processamento de áudio na GPU. No presente trabalho, uma variação deste esquema é utilizada para permitir o uso da GPU através de *externals* (*plugins*) de Pd.

A seguir, será feita uma breve descrição de cada plataforma mencionada nesta seção, e os detalhes da utilização do CUDA com Pure Data serão abordados mais adiante na Seção 4.2.1.

CUDA

A arquitetura CUDA, lançada em 2007 pela Nvidia com o objetivo de unificar a forma de desenvolvimento para sua linha de placas GPU, é composta por um modelo de programação e uma plataforma com extensões para algumas linguagens de programação (como C, C++ e Fortran). Por ter sido desenvolvida especificamente para as linhas de placas da Nvidia, a arquitetura CUDA pode acessar funções especializadas destas placas que outros arcabouços mais genéricos talvez não ofereçam suporte.

[1] http://www.nvidia.com/object/cuda_home_new.html
[2] http://www.khronos.org/opencl
[3] http://www.puredata.info/

OpenCL

OpenCL é um dos exemplos de arcabouços mais genéricos que CUDA, que pode expressar computação envolvendo não só CPU e GPU, mas também DSPs (veja a Seção ??) e outros tipos de processadores. Adotado como padrão pela indústria, diversas fabricantes oferecem hoje suporte a OpenCL, como é o caso da própria Nvidia e de outras como Intel, AMD, Altera, Samsung, e ARM. A arquitetura CUDA inclui compiladores para código escrito em OpenCL, apesar de que podem existir funcionalidades das GPUs da Nvidia que só são acessíveis com utilização de CUDA.

Pure Data

Pure Data (Puckette, 1996), também conhecido por **Pd**, é um software para processamento de áudio em tempo real, distribuído sob licenças livres, que permite a montagem de diagramas gráficos para processamento de fluxos de dados. Em uma tela, a princípio vazia, uma série de objetos que consomem, transformam e produzem sinais de áudio, vídeo e controle podem ser organizados e conectados entre si. Ao conectar estes objetos e permitir a troca de sinais entre eles, é possível implementar algoritmos arbitrários de processamento de sinais. Além disso, o Pd também pode ser facilmente estendido através de bibliotecas escritas em C, gerando espécies de *plugins* que no contexto do Pd são chamados *externals*.

O Pd processa sinais em blocos de amostras: enquanto os dados chegam, são armazenados em um *buffer* de tamanho N, que uma vez cheio pode ser manipulado de forma a realizar o processamento descrito pelo diagrama desenhado pelo usuário. O tamanho do bloco pode ser definido pelo usuário e o tamanho padrão é de 64 amostras. Como visto na Seção 1.1.1, se a frequência de amostragem do ambiente é de R Hz, espera-se que o Pd produza N novas amostras em intervalos de N/R segundos, caso contrário o tempo lógico (correspondente à primeira amostra que refletirá a computação presente) ficará atrasado em relação ao tempo real.

A partir do diagrama desenhado, o Pd consegue ordenar uma lista de rotinas de processamento e executá-las em ordem de anti-dependência sobre o conjunto de dados, produzindo assim a saída desejada. Estas funções de processamento podem executar qualquer tarefa que possa ser escrita em C, e por isso são de especial interesse para este estudo pois o fluxo de execução pode ser divergido para utilização da GPU no processamento dos dados.

Todos os modelos de placa GPU disponíveis para o grupo de Computação Musical do IME, e assim para esta investigação, eram da marca Nvidia, e portanto a opção foi de utilizar a extensão CUDA C para escrever *externals* de Pd que exportassem dados e computação para a GPU. A forma como isto foi feito será discutida em detalhes na Seção 4.2.

4.1.4 Métricas fundamentais

A terminologia usual chama de **computador hospedeiro** (*host computer*) o aparelho no qual a placa GPU está instalada, e de **dispositivo** (*device*) a placa paralela em si. No momento da escrita deste trabalho, a maioria das arquiteturas de GPU ainda possui uma memória própria, de forma que os dados sempre têm que ser copiados entre a memória principal do computador hospedeiro e a do dispositivo para viabilizar a utilização dos processadores paralelos. Esta transferência de dados consome tempo não negligenciável, e por isto tem de ser levada em conta no estudo do desempenho das placas GPU para processamento em tempo real. As métricas fundamentais consideradas neste trabalho para o estudo do desempenho e viabilidade de ambientes que utilizam a GPU para processamento em tempo real de áudio são descritas a seguir.

Tempo de transferência de memória

A GPU processa dados que residem em sua própria memória, que geralmente está separada da memória do computador hospedeiro (por exemplo, em uma placa separada conectada por alguma interface como PCI Express). Por esta razão, a transferência de memória pode representar um

gargalo que deve ser levado em conta ao desenvolver aplicações paralelas que utilizem a GPU: geralmente, quanto menor a quantidade de dados transferidos, melhor (Cebenoyan, 2004).

Tempo de execução de funções de *kernel*

Uma vez que os dados foram colocados na memória da GPU, o hospedeiro pode realizar chamadas de funções de *kernel* para serem executadas pela GPU. O tempo de execução de um *kernel* é uma métrica importante para medir o desempenho de computações executadas na GPU. Em nossos testes, consideramos o tempo de execução de *kernel* como o tempo total utilizado por todas as instruções executadas na GPU, depois que a memória foi transferida a partir do hospedeiro, e antes que ela seja transferida de volta.

Tempo de viagem de ida e volta

O tempo de viagem de ida e volta corresponde ao tempo total tomado para transferir dados do hospedeiro para o dispositivo, operar nos dados usando a GPU, e transferir a memória de volta. Este é o valor mais importante que deve ser levado em conta e comparado com o período teórico do ciclo DSP, para estabelecer a viabilidade da utilização da GPU em aplicações em tempo real.

Na próxima seção, será descrito o esquema utilizado para implementar e realizar a medição de tempo destas tarefas utilizando o Pd e a GPU.

4.2 Processamento de áudio em tempo real usando CUDA e GPUs da Nvidia

Para utilizar a generalidade da GPU e medir seu desempenho no processamento de áudio em tempo real, a abordagem utilizada neste trabalho é de exportar o cálculo paralelizável para a GPU a partir do Pd. Nesta seção, primeiro será descrito como foi modelada e programada a interação entre o Pure Data e a GPU através do arcabouço CUDA da Nvidia. Em seguida, será abordado como os algoritmos de processamento de áudio apresentados na Seção 2.2 podem ser acelerados através do paralelismo. Ao final do capítulo, os resultados obtidos serão apresentados.

Uma abordagem similar foi feita por Savioja *et al.* (2011), ao analisar o desempenho da GPU para síntese aditiva, FFT e convolução no domínio do tempo. Este trabalho difere daquele em dois pontos. Em primeiro lugar, aqui é feito o uso do Pure Data como ambiente computacional para interação com a API da GPU, possibilitando uma visão do uso do paralelismo em uma plataforma amplamente utilizada por potenciais interessados (compositores e intérpretes de música mediada por tecnologia). Adicionalmente, no presente trabalho os tempos de transferência de memória são considerados separadamente, de forma a viabilizar a comparação com o tempo de processamento.

4.2.1 Interface com a placa de vídeo utilizando o Pd

O modelo de exportação de computação desenvolvido para realização dos testes que serão descritos na Seção 4.3 utiliza a GPU junto com o Pd de forma síncrona. Em cada ciclo DSP, o Pd transfere uma porção de memória para a GPU, requisita a execução de uma série de funções de *kernel* sobre esta porção de dados, aguarda o seu término, e transfere os dados de volta. O modelo geral de comunicação e computação pode ser visto na Figura 4.4.

O objetivo de utilizar o Pure Data para a alimentação e controle da computação na GPU é utilizar toda uma infraestrutura de captura, processamento e emissão de amostras em tempo real que já está disponível e é bastante utilizada pela comunidade de artistas e pesquisadores da área de Computação Musical. Para medir o tempo de transferência de memória e de execução dos programas usando Pd e GPU, foi desenvolvido um *external* que realiza as chamadas de sistema para a GPU e mantém um controle do tempo entre as operações. O *external* se comporta como um objeto comum do Pd que recebe sinais de entrada e produz sinais de saída, enquanto a computação de fato é delegada à GPU.

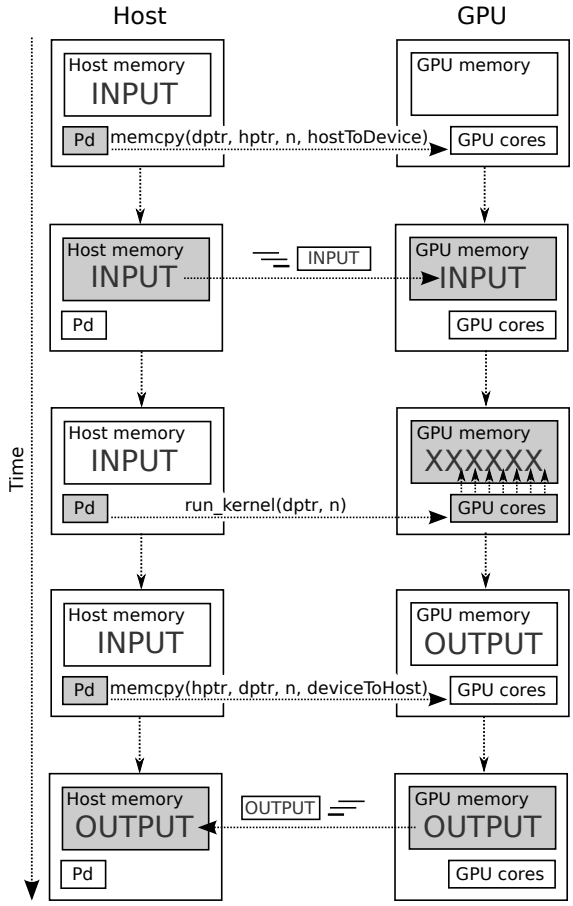

Figura 4.4: *Utilização da GPU pelo Pd durante um ciclo DSP. Os blocos de cor cinza indicam as partes ativas em cada passo. Primeiro, o Pd faz uma chamada de sistema para realizar a transferência de uma porção da memória do hospedeiro para a GPU. Após a transferência da memória, o Pd lança uma ou mais funções de* kernel *para operar sobre a memória do dispositivo. Finalmente, o Pd requisita que os dados sejam transferidos de volta e assim adquire acesso ao resultado da computação.*

O Pure Data pode ser executado com as opções `-nogui` e `-batch` para, respectivamente, executar sem interface gráfica e operar de forma a executar as computações o mais rápido possível, sem aguardar o período real do ciclo DSP para iniciar um novo ciclo. Todos os testes foram executados em 3 cenários: (1) Pd em tempo real com interface gráfica, (2) Pd em tempo real sem interface gráfica (com a opção `-nogui`) e (3) Pd sem interface gráfica e com execução da computação o mais rápido possível (com as opções `-nogui` e `-batch`). Foi possível verificar que a presença ou ausência da interface gráfica e utilização das opções tempo real ou batch têm influência mínima nos resultados obtidos nos testes. Assim, a utilização do Pd em apresentações ao vivo utilizando os sistemas ADC e DAC para entrada e saída tem resultados comparáveis aos experimentos aqui apresentados, que utilizam o Pd com as duas opções descritas acima e com entrada e saída de sinais via arquivos WAV, por conveniência e melhor possibilidade de automação dos experimentos.

Os *externals* de Pd (veja a Seção 4.1.3) são objetos binários compilados como bibliotecas dinâmicas, de forma que podem ser associados ao programa principal em tempo de execução. O código que define um *external* tem de seguir uma série de convenções, em sua maioria as mesmas utilizadas por objetos definidos no código fonte do Pd. O código fonte do Pd é estruturado seguindo o modelo de orientação a objetos, e assim a classe que define um *external* deve prover métodos para configuração da classe, criação de um novo objeto dentro de uma tela do Pd, e processamento dos fluxos que recebe e emite (veja a Seção 4.2.4).

Na implementação utilizada nos testes, a placa GPU é configurada durante a criação do objeto.

Também neste estágio, ambos o hospedeiro e o dispositivo são configurados: variáveis para medição do tempo são criadas e a memória para os cálculos é alocada. As linguagens utilizadas foram CUDA C[4] e C padrão. Foi utilizada também a biblioteca CUFFT[5], uma implementação da FFT desenvolvida e mantida pela Nvidia, compatível com a FFTW, por sua vez uma coleção de rotinas em C bastante conhecida e amplamente utilizada[6].

4.2.2 Paralelismo no processamento de áudio

O cálculo de uma amostra do sinal de saída nos algoritmos básicos para manipulação de áudio descritos na Seção 2.2 depende apenas de valores do sinal de entrada e dos valores que controlam os parâmetros utilizados nos cálculos. Por não depender do valor de outras amostras do sinal de saída, tais algoritmos podem ser paralelizados de forma a produzir todas as amostras de um mesmo bloco em paralelo (veja a Seção 4.1.2). Abaixo, segue uma descrição sucinta das possibilidades de paralelização de alguns daqueles algoritmos.

Transformada de Fourier paralela

O cálculo da fase e amplitude de cada um dos N coeficientes da transformada de Fourier depende do mesmo conjunto de dados: as N amostras do sinal no domínio do tempo. Como não dependem uns dos outros, o cálculo de cada coeficiente pode ser executado em paralelo. Assim, em cada ciclo DSP podem ser lançadas N funções que serão executadas paralelamente na CPU para calcular cada um dos coeficientes da transformada ao mesmo tempo.

Como descrito na seção anterior, a biblioteca CUFFT provê uma forma de especificar e executar Transformadas de Fourier multidimensionais paralelas utilizando CUDA em placas da Nvidia. A aceleração obtida pelas implementações altamente especializadas desta já foi comprovada em trabalhos anteriores (Merz, 2009; Radhakrishnan, 2007) e neste trabalho a biblioteca foi utilizada para implementar um *external* e medir a relação entre o tempo consumido pelo cálculo da transformada, o tempo de transferência de memória, e o período teórico do ciclo DSP.

Convolução paralela

O resultado da convolução de dois sinais é um terceiro sinal cujo valor de cada amostra depende apenas dos sinais de entrada. Para calcular a convolução de dois blocos de tamanho N em paralelo, a mesma lógica descrita acima pode ser aplicada e N funções paralelas podem ser executadas sem necessidade de que uma aguarde o término da outra para poder realizar sua tarefa.

Exemplos da utilização de CUDA para o cálculo de convoluções paralelas constam dos códigos de exemplo distribuídos com o próprio CUDA, e a aceleração resultante também foi medida e comparada com o cálculo na CPU (Podlozhnyuk, 2007). Neste trabalho, um *external* de Pure Data que realiza a convolução de dois sinais de entrada foi utilizado para comparar o desempenho nas três placas disponíveis, como será exposto na Seção 4.3.

Síntese aditiva paralela

Na síntese aditiva, cada amostra da saída é resultado da soma de K osciladores, cujos valores não dependem de nada além de parâmetros para magnitude e fase de cada oscilador. Assim, o cálculo do valor de uma amostra da saída não depende do cálculo do valor de outras amostras, e portanto cada amostra pode ser calculada em paralelo com as outras. Além disso, os valores dos próprios osciladores também podem ser calculados em paralelo, uma vez que não dependem uns dos outros. Assim, N funções podem ser lançadas em paralelo para calcular o valor de cada uma das amostras de saída, e em seguida K funções podem ser lançadas para calcular os novos valores dos osciladores em paralelo.

[4] http://developer.nvidia.com/cuda-toolkit
[5] http://developer.nvidia.com/cufft
[6] http://www.fftw.org/

Como a GPU é um ambiente com bastante poder computacional, a implementação da síntese aditiva foi composta com uma análise prévia do sinal utilizando a FFT paralela para implementar o Phase Vocoder, como descrito na Seção 2.2.4. O tempo de execução da FFT paralela foi analisado separadamente do tempo da síntese aditiva, como será visto na Seção 4.3.

4.2.3 Especificidades da GPU

Como comentado na Seção 4.1, o balanço entre funções fixas e unidades programáveis é fundamental para o aumento da generalidade e desempenho das GPUs. Ao longo do desenvolvimento dos dispositivos gráficos, algumas funções fixas foram cristalizadas por terem importância fundamental para auxiliar a computação ao longo dos estágios da *pipeline* da GPU. É o caso, por exemplo, das funções trigonométricas e da memória de texturas (e suas operações associadas, como leitura interpolada de índices fracionários). A natureza destas funções coincide com algumas operações básicas utilizadas nos algoritmos de processamento de áudio, e assim sua utilização pode aumentar o desempenho da GPU no processamento de áudio em tempo real.

Na implementação do Phase Vocoder para GPU, mais especificamente no estágio de ressíntese implementado utilizando síntese aditiva, o uso da leitura interpolada de índices fracionários da memória de texturas e da função trigonométrica `sin()`, embutidas no hardware da GPU, oferece possibilidade imediata de comparação com a técnica de cálculo de uma função senoidal baseada em interpolação (linear ou cúbica, no caso desta exploração), através da leitura de índices fracionários de uma tabela contendo os valores de um período da função seno.

4.2.4 Implementação

Para implementar a utilização do Pd com GPU, o primeiro passo foi desenvolver um modelo de *external* que inicializa a placa GPU e controla a transferência de memória e chamadas de funções de *kernel* de acordo com os ciclos DSP do Pd. Em seguida, este modelo foi copiado para implementar os diferentes algoritmos testados (FFT e Phase Vocoder), e um conjunto de *shell-scripts* foi escrito para auxiliar a execução automatizada dos testes. Esta seção descreve alguns detalhes importantes da estrutura de testes desenvolvida.

Estrutura (orientada a objetos) de um *external*

Um diagrama desenhado no Pd é chamado *patch*. Os objetos gráficos que podem ser utilizados para compor um diagrama de processamento em um *patch* podem ser de um de três tipos: *abstractions*, *built-ins* ou *externals*. Abstrações são outros *patches* criados com o Pd e encapsulados em um objeto, e podem possuir entradas e saídas de fluxos de áudio e controle através, respectivamente, dos pares de objetos (`inlet~`, `outlet~`) e (`inlet`, `outlet`). *Built-ins* são objetos binários escritos em C e compilados junto com o arquivo binário principal do Pd. Por sua vez, os *externals* também são objetos escritos em C, mas compilados como bibliotecas dinâmicas que podem ser carregadas em tempo de execução.

Os *externals* devem aderir ao modelo de orientação a objetos utilizado em todo o código do Pd, e devem possuir um conjunto minimal de funções para viabilizar a criação e operação de objetos gráficos na construção de um *patch*. As seguintes componentes são o mínimo necessário para codificar um *external* que processa áudio:

- **Estrutura de dados:** Uma estrutura de dados que representa o objeto do *external*. Uma porção de memória contendo uma instância desta estrutura é disponibilizada para o método de processamento de áudio (veja abaixo) junto com os sinais processados. Ela deve conter campos para armazenar os parâmetros de controle que o objeto pode receber, ponteiros para regiões de memória alocadas dinamicamente e outros valores dos quais o objeto necessite para sua operação.

- **Método de criação de um novo objeto:** Este método deve criar uma estrutura de dados do tipo descrito no item anterior, inicializar seus valores e retornar o valor do ponteiro para a estrutura criada.
- **Método de processamento de áudio:** Este é o método de manipulação dos sinais de áudio. Ele recebe as configurações atuais do Pd e um ponteiro para a estrutura de dados que representa a instância atual do objeto do Pd e pode trabalhar em cima dos *buffers* com as amostras de áudio para produzir o efeito desejado.
- **Método de inicialização do processamento:** Executado sempre que o processamento de áudio é iniciado, este método inclui o método acima em uma fila de métodos que serão executados para processar os sinais ao longo do diagrama.
- **Método de configuração de classe:** A quantidade e tipos de entradas e saídas do objeto, o método de criação de um novo objeto e o método de inicialização do processamento são configurados por este método.

Inicialização da placa GPU

A inicialização da placa GPU consiste na configuração do número da placa a ser utilizada (pode haver diversas placas num mesmo hospedeiro) e alocação e inicialização de memória para computação no dispositivo. Estas tarefas podem ser realizadas no método de criação de um novo objeto, tomando os devidos cuidados para evitar que duas instâncias de um mesmo *external* não interfiram na computação um do outro.

Organização do código e compilação

Para melhor organização do código é interessante encapsular toda a parte que lida com a GPU (funções de *kernel* e funções que façam chamadas a funções de *kernel* de forma que sejam especificadas em arquivos com extensão `.cu` e que possam ser compiladas pelo compilador do CUDA. Assim, a parte do código que lida somente com funções escritas em C pode continuar sendo compilada pelo compilador convencional, enquanto que todas as partes que lidam com código escrito em CUDA-C podem ser compiladas pelo `nvcc`, compilador específico distribuído junto com o arcabouço CUDA.

Funções que lidam com a GPU

As chamadas de funções que lidam com a GPU são, em geral, de três tipos: alocação ou liberação de memória, funções de *kernel* disponíveis na biblioteca CUDA, ou funções de *kernel* especificadas pelo usuário. Na implementação desenvolvida, a alocação de memória na GPU é realizada na criação do objeto, e a transferência de dados e as chamadas às funções de *kernel* são feitas a cada ciclo DSP no método de processamento.

4.3 Resultados e discussão

Foram utilizados dois ambientes de testes, com um total de três modelos distintos de placas Nvidia GPU. O primeiro ambiente é um computador `Intel(R) Core(TM) i7 CPU 920 @2.67GHz` com 8 cores e 6 GB RAM, rodando `Ubuntu GNU/Linux 11.10` com versão de *kernel* `3.0.0-32-generic`, e equipado com dois modelos de placa Nvidia GPU: Geforce GTX 275 e Geforce GTX 470. O segundo ambiente de testes é um computador `Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz`, com 12 cores e 24 GB RAM, rodando `Ubuntu GNU/Linux 12.10` com versão de *kernel* `3.5.0-24-generic`, equipado com uma placa Nvidia Quadro GF 100 GL. Foram utilizadas a versão 5.0 da plataforma CUDA e a versão 0.44-0 do Pure Data para rodar os *externals* desenvolvidos.

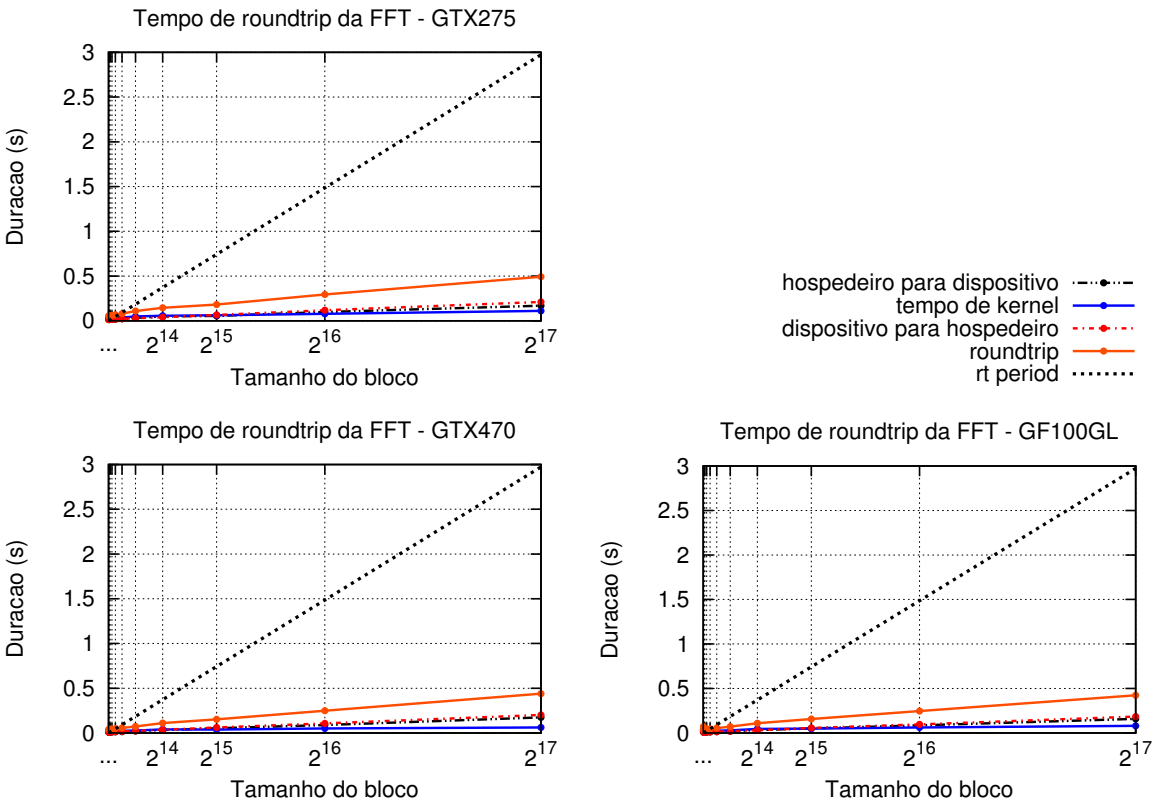

Figura 4.5: *Tempo de transferência de memória e de execução da FFT para diferentes tamanhos de bloco em diferentes modelos de placa de vídeo GPU.*

Um resumo das características de cada placa pode ser visto na próxima tabela:

Modelo	Cores	Mem (MB)	Mem BW (GB/s)
GTX 275	240	896	127.0
GTX 470	448	1280	133.9
GF 100	256	2000	89.6

Para avaliar o desempenho do esquema utilizado para DSP em tempo real usando a GPU, o primeiro passo foi implementar um *external* de FFT utilizando a biblioteca CUFFT, que transfere dados para a GPU, executa o algoritmo sobre destes dados, e finalmente transfere os dados de volta para a memória do computador. Os resultados para diferentes tamanhos de bloco podem ser vistos nas Figuras 4.5 e 4.6, e serão discutidos na próxima seção. Bastante tempo de computação ainda fica disponível após o cálculo da FFT, mesmo considerando o tempo de transferência de memória. Para verificar a possibilidade de uso da GPU em tarefas mais intensas, também foram implementados os algoritmos de convolução e Phase Vocoder.

A implementação da convolução é imediata: toma-se dois sinais de entrada conectados a um objeto do Pd e realiza-se a convolução dos dois sinais calculando cada amostra da saída em paralelo, como descrito na Seção 4.2.2. Como a entrada do algoritmo são dois sinais de áudio, a quantidade de memória transferida aqui é o dobro da quantidade transferida no caso anterior do cálculo da FFT de apenas um sinal. O resultado do tempo total de execução, somando o tempo de transferência de memória e o tempo de execução da função de *kernel* que realiza a convolução pode ser visto na Figura 4.7.

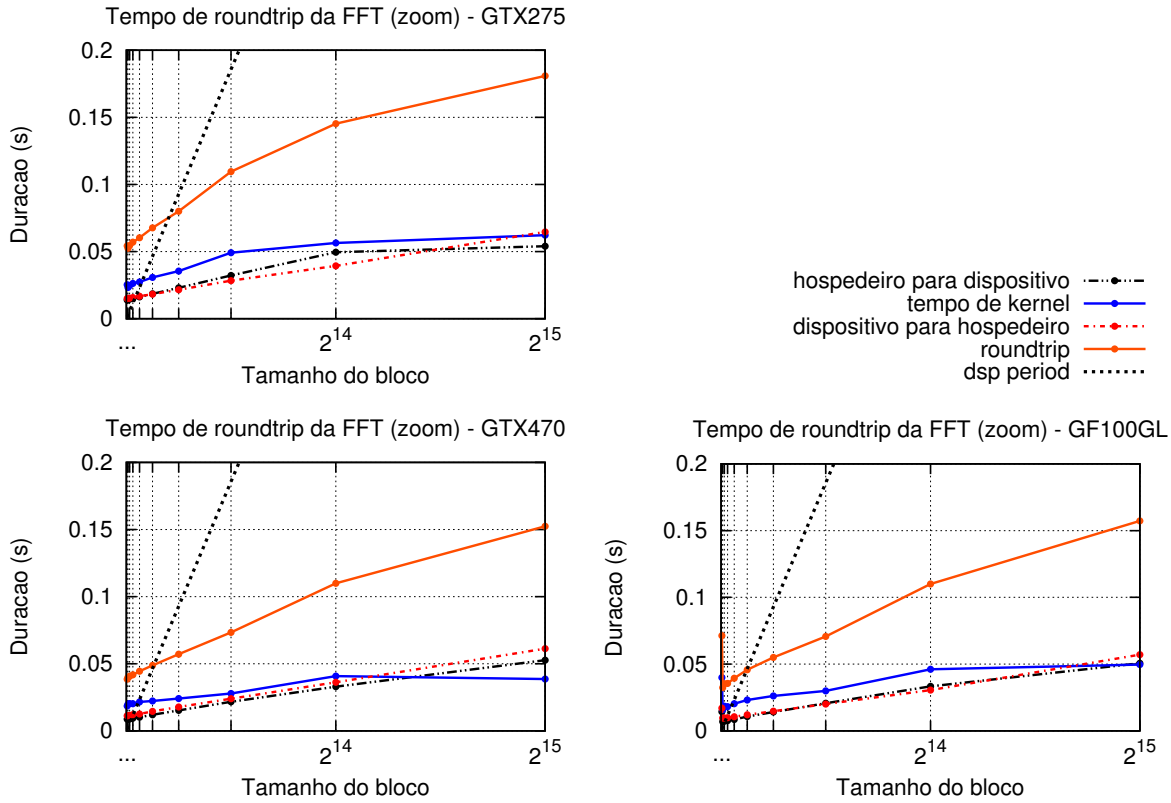

Figura 4.6: *Porção inicial do gráfico anterior do tempo de transferência de memória e de execução da FFT para diferentes tamanhos de bloco em diferentes modelos de placa de vídeo GPU.*

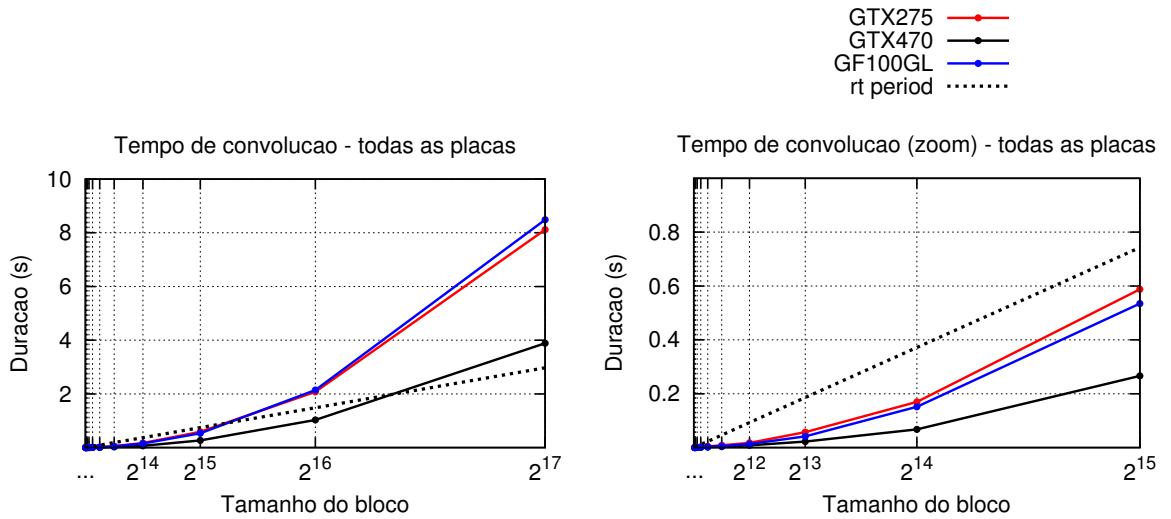

Figura 4.7: *Tempo de viagem de ida e volta para o algoritmo de convolução em todas as placas GPU.*

Uma implementação do Phase Vocoder para a GPU pode utilizar paralelismo de duas formas. Primeiro, pode estimar as amplitudes e frequências instantâneas para cada oscilador fazendo uso da FFT paralela. Em seguida, como o resultado de cada amostra sintetizada não depende do cálculo do valor de outras amostras de saída, o Phase Vocoder pode realizar uma síntese aditiva em paralelo para cada amostra de saída de um bloco DSP. Assim, uma implementação do Phase Vocoder na GPU transfere a mesma quantidade de dados entre o computador hospedeiro e o dispositivo que o algoritmo da FFT paralela, mas é composta de mais chamadas a funções paralelas e mais computação dentro de cada função.

A parte do código do Phase Vocoder que implementa a síntese aditiva é computacionalmente intensa e bastante sensível em relação ao método utilizado para obter o valor de cada oscilador senoidal, como observado por Savioja et al. (Savioja *et al.*, 2011). Nos testes realizados, foram comparadas 5 implementações distintas:

1. Consulta a tabela com interpolação cúbica utilizando 4 pontos.
2. Consulta a tabela com interpolação linear utilizando 2 pontos.
3. Consulta a tabela com índice truncado (sem interpolação). Note que a qualidade numérica de uma consulta truncada pode ser melhorada aumentando-se o tamanho da tabela (e a GPU, diferentemente do que foi visto com o Arduino no capítulo anterior, possui memória suficiente para tabelas grandes).
4. Primitiva trigonométrica da GPU. A função `sinf()` da API do CUDA computa um número de ponto flutuante com precisão dupla.
5. Primitiva de consulta a índices fracionários na memória de textura. A GPU se encarrega de realizar e retornar um valor interpolado linearmente.

Os resultados para tempos de transferência de memória e tempo de *kernel* da síntese aditiva dos testes com o Phase Vocoder paralelo podem ser vistos nas Figuras 4.8 e 4.9, e também serão discutidos na próxima seção.

Cada algoritmo (FFT, convolução e Phase Vocoder) foi executado por um período igual a 100 blocos DSP para tamanhos de bloco iguais a 2^i, para $6 \leq i \leq 17$, e em seguida foram calculados os tempos médios para a transferência de dados (de ida e volta) e para a execução das funções de *kernel* relativas a cada algoritmo.

O maior tamanho de bloco considerado, de $2^{17} = 131.072$ amostras, corresponde a um período de por volta de 3 segundos de áudio. Esta escolha de período de um ciclo DSP pode parecer exagerada para utilização em tempo real, mas a latência correspondente pode ser compensada pela escolha de um fator de sobreposição grande de forma a manter o tamanho do bloco (e portanto a resolução espectral associada) e obter maior resolução temporal. O tempo de execução do Phase Vocoder para blocos de amostras de tamanho maior do que 2^{17} excede, para todas as implementações, o período do bloco DSP correspondente, de forma que este tamanho de bloco é suficiente para prover limitantes superiores para a viabilidade da computação como função do tamanho do bloco em todos os modelos de GPU testados.

Comparando as figuras que descrevem apenas a FFT com as que descrevem a implementação completa do Phase Vocoder, é possível ver que há uma diferença notável, de algumas ordens de magnitude, entre o tempo utilizado para rodar cada um destes algoritmos. Comparando o tempo tomado pelos dois algoritmos para um mesmo modelo de placa, é possível ver que a FFT toma tempo comparável ao tempo de transferência de memória, da ordem de décimos de milissegundos, enquanto que a implementação completa do Phase Vocoder toma vários segundos para tamanhos de bloco grandes. Isto indica que centenas de FFTs poderiam ser executadas em um ciclo DSP, enquanto que apenas alguns ciclos de análise e síntese de Phase Vocoder poderiam ser executados na mesma quantidade de tempo.

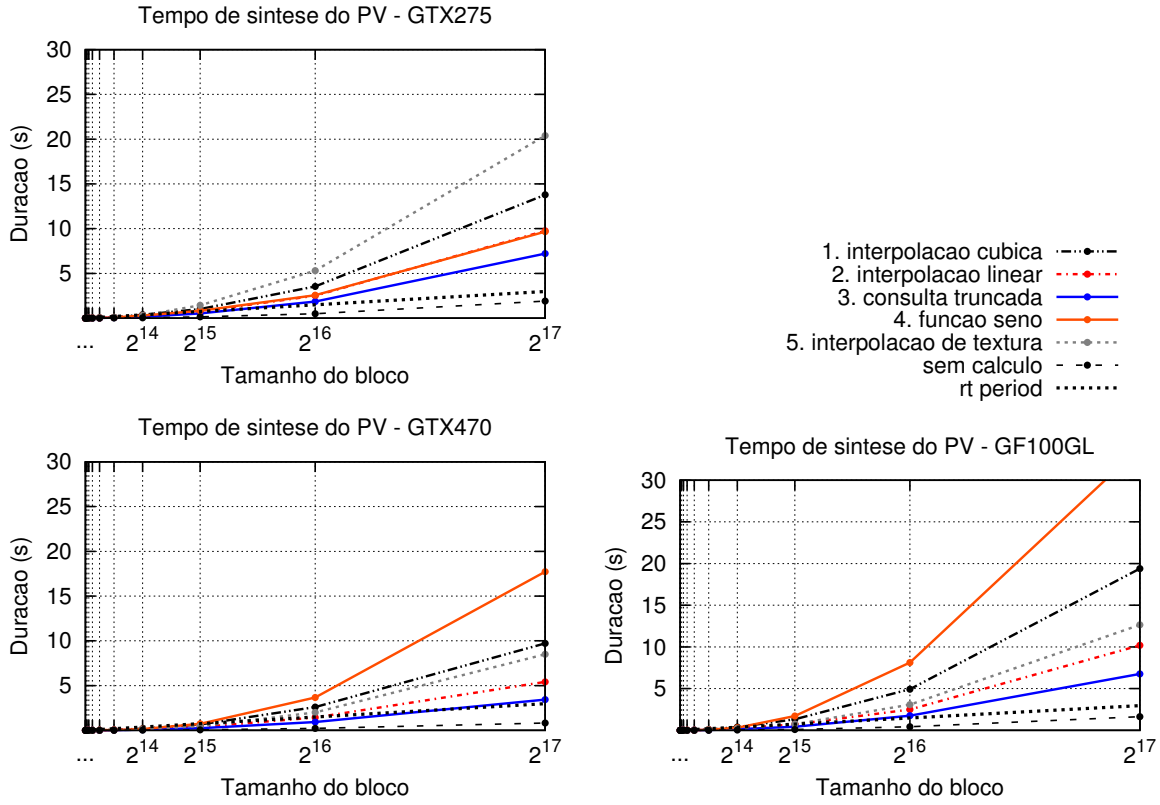

Figura 4.8: *Tempo de transferência de memória e de execução da síntese aditiva do Phase Vocoder para diferentes tamanhos de bloco em diferentes modelos de placa de vídeo GPU.*

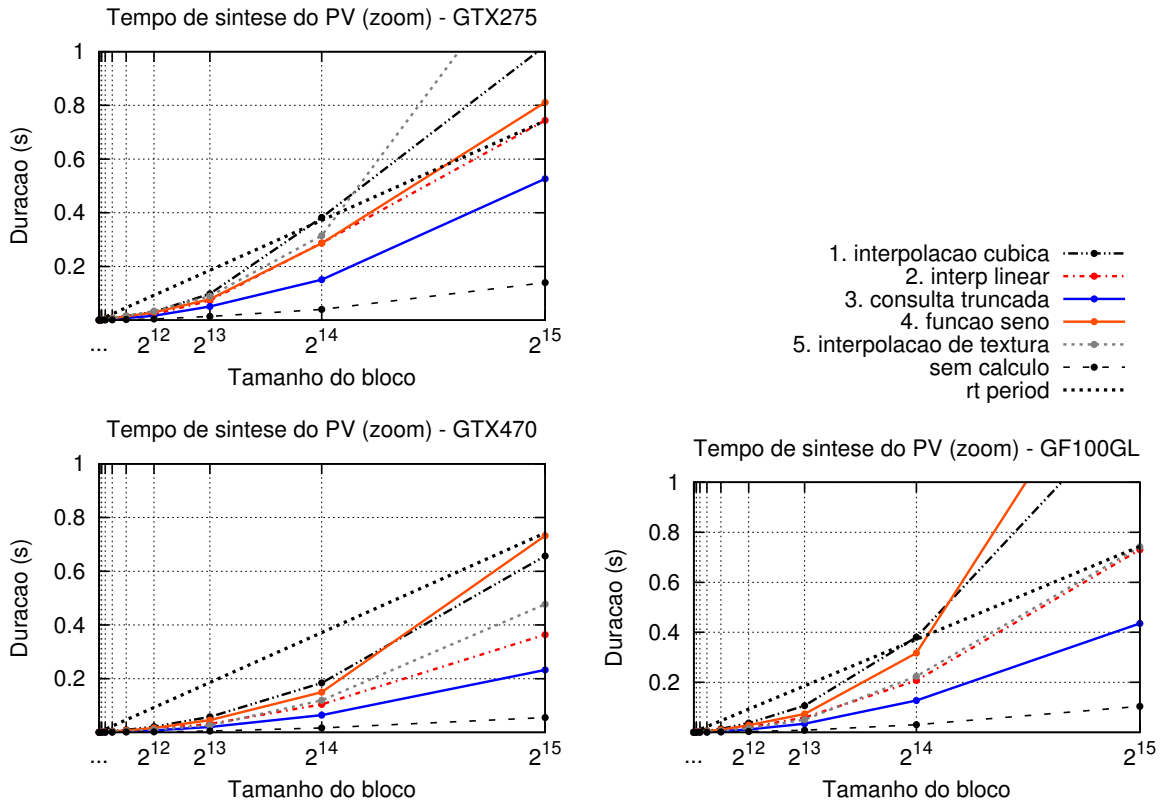

Figura 4.9: *Porção inicial do gráfico anterior do tempo de transferência de memória e de execução da síntese aditiva do Phase Vocoder para diferentes tamanhos de bloco em diferentes modelos de placa de vídeo GPU.*

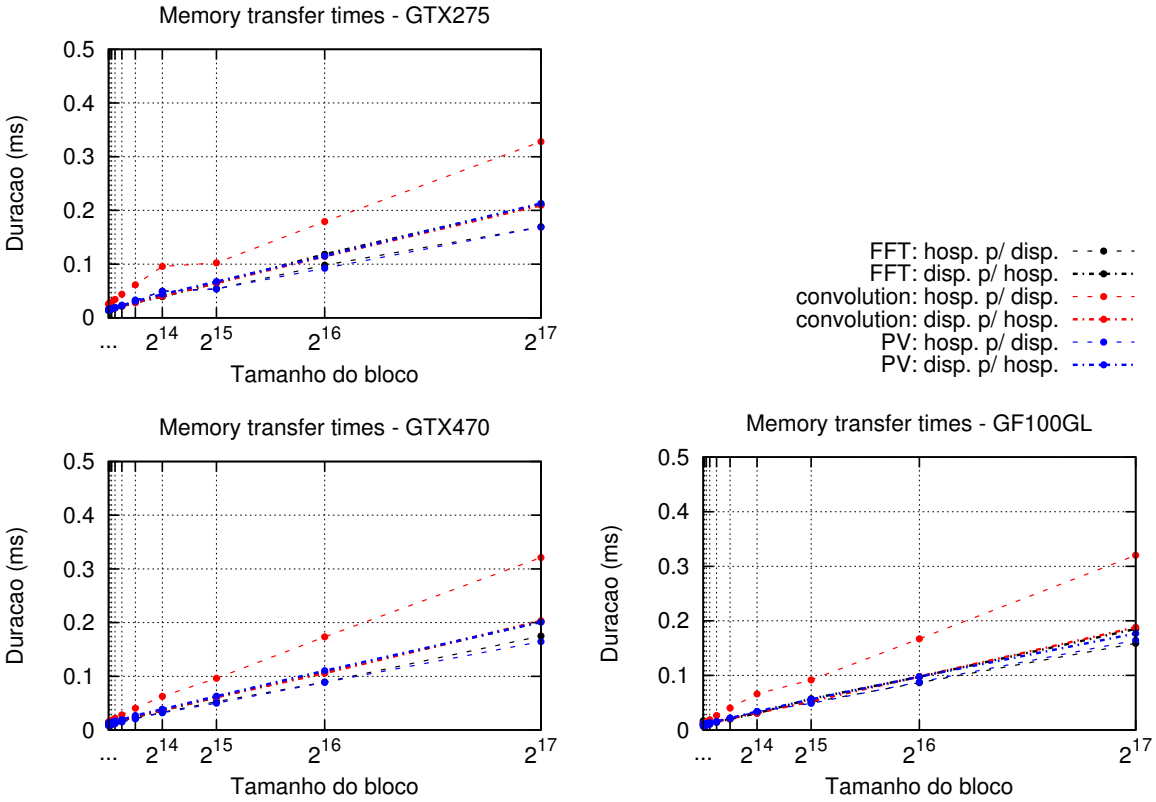

Figura 4.10: *Tempo de transferência de memória para os algoritmos FFT, convolução e PV, para cada uma das placas.*

4.3.1 Tempo de transferência de memória

Os resultados para os tempos de transferência de memória para cada algoritmo em cada uma das placas GPU podem ser vistos na Figura 4.10. É possível observar que o tempo tomado para transferir uma certa quantidade de memória do hospedeiro para o dispositivo e de volta é aproximadamente linear em relação ao tamanho do bloco. Isto parece razoável uma vez que a banda de transferência entre o hospedeiro e a GPU indicada pelo fabricante é constante (veja a tabela na Seção 4.3). Apesar disso, é preciso notar que a relação entre banda de transferência e velocidade de transferência pode depender da arquitetura e implementação da hierarquia de memórias.

Também foi possível determinar que os tempos de transferência de memória são bastante próximos para as implementações de FFT, convolução e Phase Vocoder, se divididos pelo número de amostras transferidas. Isto indica que o número de chamadas a funções de *kernel* ou o tempo que estas funções tomam ao trabalhar em cima dos dados parece não ter influência na velocidade de transferência de memória.

Finalmente, deve-se notar que a diferença de tempo de transferência de memória nos três modelos é bastante baixa, e em todos os cenários a transferência de volta toma um pouco mais de tempo do que a de ida. Novamente, pequenas diferenças entre os modelos são esperadas pois, apesar de serem de famílias diferentes, todos os modelos possuem banda de transferência bastante alta quando comparadas com a quantidade de memória transferida de fato nos testes (um máximo de cerca de 4 MB em cada ciclo DSP).

4.3.2 FFT

O algoritmo original da FFT consome tempo proporcional a $N \log(N)$, onde N é o número de amostras. A versão paralela do algoritmo da FFT consome tempo proporcional a $N \log(N)/p$, onde p é o número de processadores utilizados (Cui-xiang *et al.*, 2005). Como o número de threads

paralelas lançadas em cada ciclo DSP para blocos grandes é da ordem de milhares, enquanto que o número de processadores é da ordem de centenas (ou seja, $N \gg p$), o número p de processadores pode ser entendido como uma constante que depende somente da placa GPU a qual o usuário tem acesso. Assim, a tendência esperada é que o tempo do *kernel* da FFT em algum momento ultrapasse o tempo de transferência de memória de acordo com o crescimento do tamanho dos blocos. Apesar disso, pode-se observar nas Figuras 4.5 e 4.6 que, para todos os tamanhos de bloco testados, o tempo de viagem de ida e volta relativo à FFT é bastante pequeno quando comparado com o período teórico do ciclo de processamento em tempo real (cerca de 1/6), deixando bastante espaço para a realização de outras computações.

4.3.3 Convolução

A curva do tempo da função de *kernel* que realiza a convolução dos dois sinais de entrada, observado na Figura 4.7, parece esboçar a complexidade computacional quadrática do algoritmo de convolução. Este resultado é esperado uma vez que, para blocos grandes, o argumento de considerar o número de processadores como uma constante que depende da placa utilizada, apresentado acima para a FFT, vale para qualquer outro algoritmo. É interessante ver que o tempo de processamento é reduzido para cerca de metade na GTX470, em relação às outras duas placas, o que se justifica exatamente pelo número de processadores presente nesta placa (pouco menos de duas vezes a quantidade presente nas outras duas), o que é talvez o fator mais significativo na caracterização da tal constante que varia de placa para placa.

4.3.4 Phase Vocoder

Em relação à implementação do Phase Vocoder, como observado na última seção, foi possível determinar que a parte que consome o maior tempo da GPU é a parte do cálculo dos valores dos osciladores na síntese aditiva. Para se ter uma ideia de quantos recursos tal computação consome, foram comparadas 5 implementações distintas do cálculo dos osciladores, descritas no início desta seção. Três delas utilizam consulta a uma tabela de 1024 pontos contendo um período completo de um sinal senoidal, e duas delas utilizam funções embutidas na GPU para ajudar nos cálculos.

Nas Figuras 4.8 e 4.9 também estão desenhados o período do ciclo DSP para cada tamanho de bloco, e os resultados de tempo para uma implementação de "controle" para funcionar como uma base de comparação. Esta implementação de controle contém toda a parte de análise do Phase Vocoder (que contém por sua vez uma chamada da função de *kernel* que executa a FFT para estimação da fase e amplitude dos osciladores) e também o laço principal da síntese, mas sem o cálculo dos valores dos osciladores (o laço apenas soma um número constante em cada rodada).

Em relação às implementações (1), (2) e (3), é possível ver que se comportam da forma esperada, ou seja, o tempo de cálculo cresce proporcionalmente de acordo com o número de operações envolvidas: consulta truncada à tabela é mais rápida que consulta com interpolação linear, que por sua vez é mais rápida que consulta com interpolação cúbica. Consistentemente, todas elas são mais rápidas na placa GTX 470, seguidas por um comportamento parecido nas placas GTX 275 e GF 100 GL.

A implementação (4), que utiliza a função seno da API, toma aproximadamente o mesmo tempo que a implementação (2) na placa GTX 275, mas tem um desempenho pior nas placas GTX 470 e GF 100 GL, atingindo nestas um resultado intermediário entre as implementações (1) e (2) até blocos de tamanho 2^{14}, e piorando consideravelmente para o bloco de tamanho 2^{17}.

Em relação à implementação (5), que utiliza leitura interpolada de texturas, seu comportamento é um pouco difícil de explicar. Na placa GTX 275, possui o pior desempenho de todos os métodos, tomando cerca de 40% mais tempo do que a implementação com interpolação cúbica, a segunda mais cara. Por outro lado, nas placas GTX 470 e GF 100 GL, a implementação (5) possui um desempenho comparável com todos os outros, sendo inclusive mais rápida do que a implementação de interpolação cúbica. Pode-se supor que este comportamento deve ter algo a ver com diferenças

na implementação (em hardware) da leitura da memória de texturas nos diferentes modelos de placa.

Destes gráficos pode-se ver que, para os modelos GTX 275 e GF 100 GL, blocos de tamanho maior ou igual a 2^{16} demoram mais tempo para serem computados do que o tempo disponível para aplicações em tempo real, independente da implementação utilizada. Um resultado similar pode ser visto no modelo GTX 470, para blocos com 2^{17} amostras.

Também pode-se observar que os tempos do Phase Vocoder crescem superlinearmente para todas as implementações. Uma vez que o número de osciladores na síntese aditiva corresponde a cerca da metade do número de amostras de um bloco, uma complexidade computacional quadrática é esperada. O grau de paralelismo trazido pela GPU não é suficiente para produzir diferenças nos perfis dos vários métodos de consulta a tabela implementados, e diferenças em escala são explicadas pela constante oculta na notação O.

Dados os resultados apresentados, é possível concluir que pequenas diferenças de implementação podem ter resultados significativos no tempo de execução de um *kernel* na GPU. Não está claro, por exemplo, o quanto a qualidade numérica da função seno *built-in* é melhor do que a da interpolação cúbica de 4 pontos, mas está claro que escolhas conscientes devem ser feitas para poder utilizar o potencial total da GPU para blocos grandes.

Também é possível concluir que, se um ciclo DSP for restringido a poucas transferências de memória em cada direção, então não há a necessidade de se preocupar com o tempo de transferência de memória uma vez que sua magnitude é da ordem de décimos de milissegundos, enquanto que os períodos dos blocos DSP são da ordem de diversos milissegundos, mesmo para os menores tamanhos de bloco considerados.

Analisando os resultados de tempo da análise e síntese do Phase Vocoder, é possível obter o tamanho máximo dos blocos para os quais o uso de cada implementação de oscilador é viável, para cada modelo de placa considerado. O número máximo de amostras atingido para cada cenário está resumido na tabela a seguir:

modelo \ implementação	1	2	3	4	5
GTX 275	2^{14}	2^{15}	2^{15}	2^{15}	2^{14}
GTX 470	2^{15}	2^{16}	2^{16}	2^{15}	2^{15}
GF 100 GL	2^{14}	2^{14}	2^{15}	2^{14}	2^{15}

Capítulo 5

Processamento de áudio em tempo real em Android

Nos capítulos anteriores, bastante ênfase foi dada ao hardware utilizado e a programação em cada ambiente foi utilizada como ferramenta para viabilizar a utilização de cada placa. No caso do Arduino, apesar de haver modelos com diferentes implementações em hardware, o objetivo foi olhar para o microcontrolador como uma unidade de processamento de áudio em tempo real e descobrir seus pontos fortes e fracos para esta tarefa. No caso da GPU, a discussão sobre processamento de propósito geral forneceu insumo à abordagem, mas a implementação foi executada em diferentes modelos de placa de uma fabricante específica. Este capítulo, por sua vez, descreve a abordagem do sistema operacional Android para processamento de áudio em tempo real. Como um dos grandes objetivos do sistema Android é ser executável num conjunto muito grande de dispositivos, o foco se volta para o software.

Na Seção 5.1, a programação para o sistema operacional Android será descrita em linhas gerais. Na Seção 5.2, as possibilidades de entrada e saída de áudio e de agendamento de execução serão descritas, com objetivo de montar uma infraestrutura mínima para viabilizar o processamento de áudio em tempo real. Esta infraestrutura permitiu o desenvolvimento de um aplicativo que foi executado em diversos aparelhos e colheu dados sobre o desempenho de diferentes algoritmos de processamento de áudio em tempo real. Estes algoritmos e os resultados serão discutidos na Seção 5.3.

5.1 Programação para Android

O desenvolvimento de aplicações para o sistema Android é relativamente simples, pois utiliza uma API em Java bem definida, convenções simples e bem documentadas para descrição dos recursos utilizados, além de poder ser realizado a partir da ferramenta Eclipse, uma IDE para Java com licença livre. Esta seção descreve o esquema geral de programação para Android e o arcabouço de desenvolvimento utilizado.

5.1.1 Organização do sistema operacional

O sistema operacional Android é organizado em quatro camadas, como pode ser visto na Figura 5.1: aplicações (*applications*); arcabouço para aplicações (*application framework*); bibliotecas e tempo de execução (*libraries and Android runtime*); e, finalmente, o kernel do Linux (*Linux kernel*).

Na camada mais próxima dos usuários estão as aplicações (envio de mensagens, calendário, navegador de internet, contatos, etc), escritas em Java (com possíveis requintes em C utilizando a JNI[1]) e desenvolvidas de acordo com algumas convenções que permitem o intercâmbio de funcionalidades entre aplicações distintas. As aplicações são desenvolvidas utilizando um arcabouço de desenvolvimento em Java, conceitualmente posicionado numa camada imediatamente abaixo da

[1] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

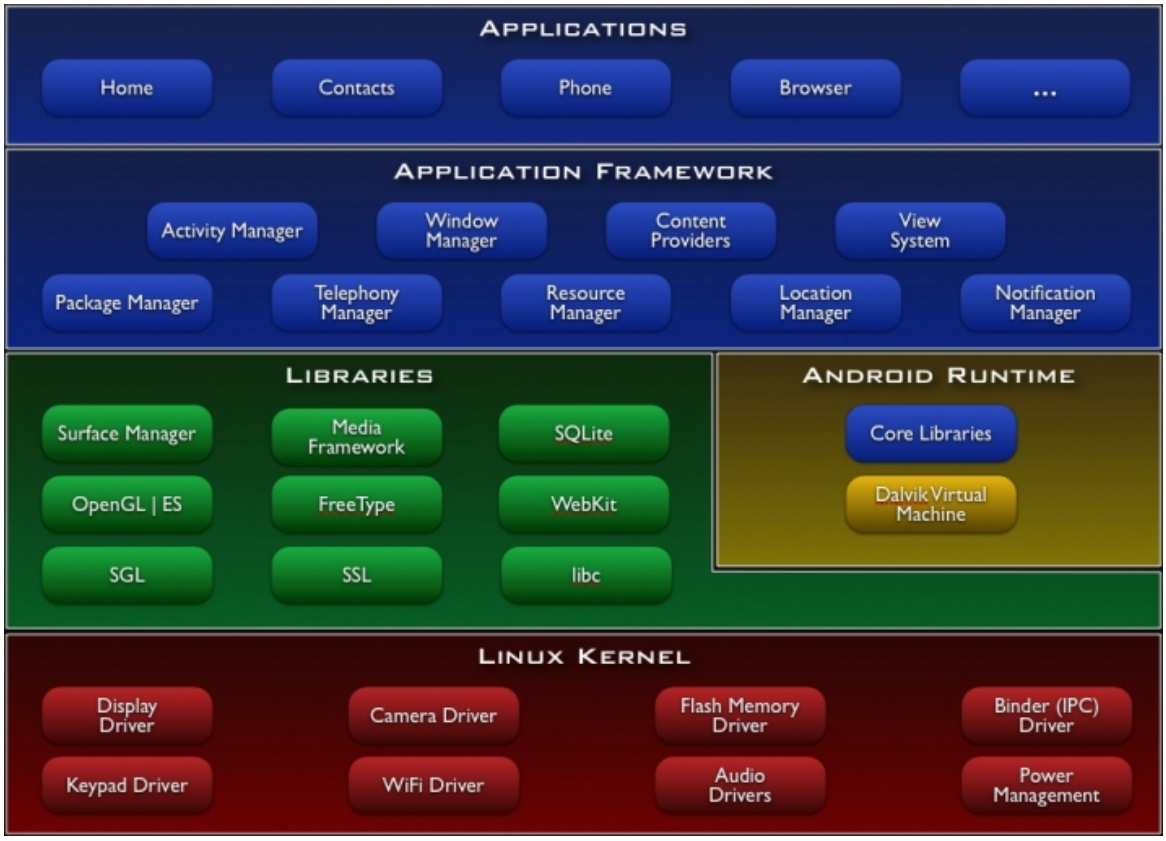

Figura 5.1: *A organização do sistema operacional Android: uma pilha de software.*

camada de aplicações, composto por um conjunto de classes que provê interface com as funcionalidades do aparelho e com outras aplicações. As bibliotecas para desenvolvimento e sua documentação estão disponíveis para download na internet[2,3]. Para desenvolvimento de aplicações para Android é necessário, portanto, conhecer as convenções e o arcabouço de desenvolvimento disponíveis.

De acordo com as convenções de desenvolvimento para o Android, toda aplicação deve definir quais recursos do sistema está preparada para utilizar, como por exemplo entrada e/ou saída de som, internet, bluetooth, etc. No momento da instalação de uma certa aplicação no sistema, o usuário do aparelho é inquirido sobre a concessão de permissão para cada um dos recursos aos quais a aplicação dá suporte. O sistema Android se utiliza, então, da infraestrutura de processos e usuários fornecida pelo sistema Linux para garantir que cada aplicação seja executada dentro de um contexto limitado, de forma que um eventual mal funcionamento da aplicação não comprometa nada além dela mesma. Cada aplicação no sistema é, portanto, executada dentro de um processo diferente, lançado por um usuário (de sistema) específico, criado para aquela aplicação no momento da instalação, com permissões para acessar apenas os recursos que o usuário (do aparelho) escolheu. Desta forma, a aplicação possui acesso a uma porção de memória limitada e não possui mais permissões do que o estritamente necessário para seu funcionamento (ou até eventualmente menos, se o usuário do aparelho assim o desejar).

Além de definir quais recursos do sistema está preparada para utilizar, uma aplicação também pode definir quais recursos disponibiliza para o sistema, de forma que outras aplicações possam utilizar esses recursos. Por exemplo, uma aplicação de processamento digital de sinais pode fornecer filtros de áudio e vídeo que podem ser utilizados por outras aplicações no momento em que forem processar conteúdo multimídia. Esta reutilização de recursos é gerenciada pelo arcabouço de desenvolvimento (representado pela segunda camada, de cima para baixo, na Figura 5.1).

[2]http://developer.android.com/sdk/index.html
[3]http://developer.android.com/reference/packages.html

Descendo mais uma camada no modelo de organização do sistema Android, encontra-se um conjunto de bibliotecas que são utilizadas por diversos componentes do sistema e que também estão disponíveis para o desenvolvedor de aplicações através do arcabouço para desenvolvimento de aplicações da camada imediatamente acima. Estão incluídas nesta camada bibliotecas de sistema, gravação e reprodução de mídia, processamento de imagens em duas e três dimensões, suporte a alguns tipos de banco de dados, entre outras funcionalidades.

Conceitualmente posicionados na mesma camada, bibliotecas operam junto com código "em tempo de execução". Como as aplicações são desenvolvidas em Java, o código gerado não é específico para uma arquitetura e necessita de uma máquina virtual para ser executado. O sistema Android utiliza uma máquina virtual própria[4], otimizada para execução em aparelhos móveis, de forma que múltiplas instâncias podem rodar ao mesmo tempo de forma eficiente. Como foi descrito acima, cada aplicação é executada dentro de um processo próprio. Cada processo executa uma instância diferente desta máquina virtual, que depende da infraestrutura do kernel do Linux para gerenciamento de memória de baixo nível e criação de *threads* de execução.

Finalmente, no último nível se encontra o kernel do Linux, que funciona como uma ponte entre o hardware e as outras camadas de software, provendo serviços para os outros níveis tais como gerenciamento de memória e de processos, segurança, rede e drivers.

As aplicações e o arcabouço de desenvolvimento para aplicações são, geralmente, escritos em Java. As bibliotecas de sistema e a máquina virtual são escritas em C/C++. O kernel do Linux é escrito em C. Com exceção do Kernel do Linux, que é publicado sob a GPL versão 2.0, o resto do código fonte do Android é, em sua maioria, licenciado sob a Apache Software License 2.0[5] e pode ser obtido no sítio do sistema Android[6].

5.1.2 Estrutura das aplicações

Nesta seção, os tópicos fundamentais para o desenvolvimento de aplicações no Android serão descritos de forma resumida. Informações mais detalhadas podem ser encontradas no sítio do produto[7].

Como descrito anteriormente, as aplicações para Android são escritas em Java utilizando um arcabouço de bibliotecas específico. Um kit de desenvolvimento, disponível para download no sítio do Android[8], pode ser utilizado para compilar e empacotar os binários junto com outros arquivos eventualmente necessários para distribuição. Após instalada, cada aplicação é executada em um ambiente seguro e controlado, utilizando um usuário do sistema específico (um usuário diferente é criado para cada aplicação instalada), de forma que é executada em um processo e máquina virtual próprios, com o mínimo de permissões necessárias para seu funcionamento. As permissões para uso de recursos do sistema devem ser dadas pelo usuário do aparelho no momento da instalação.

As aplicações no sistema Android são divididas em diversos *componentes*, definidos como "pontos através dos quais o sistema pode acessar a aplicação". Cada componente pode ser de um dentre quatro tipos: *atividade* (nome dado a cada uma das telas de interface com usuário), *serviço* (operações sem interface), *provedor de conteúdo* (provê gerenciamento de persistência de dados) e *receptor de mensagens difundidas* (que fornece meios para a aplicação responder a mensagens do sistema). Qualquer aplicação pode iniciar componentes de outra aplicação e eventualmente receber de volta o resultado da execução daquele componente.

Uma aplicação não possui permissão para executar os componentes de outra aplicação diretamente, e portanto a conexão entre componentes de aplicações distintas deve ser intermediada pelo sistema. O acesso a componentes dos tipos atividade, serviço e receptor de mensagens são feitos através de *mensagens de intenção* assíncronas, enviadas pela aplicação ao sistema. O acesso a componentes do tipo provedor de conteúdo é feito através de um objeto específico, chamado *resolvedor*

[4] http://code.google.com/p/dalvik/
[5] http://www.apache.org/licenses/LICENSE-2.0
[6] http://source.android.com/
[7] http://developer.android.com/guide/topics/fundamentals.html
[8] http://developer.android.com/sdk/index.html

de conteúdo.

Com o objetivo de prover para o sistema todas as informações necessárias para seu correto funcionamento, uma aplicação deve incluir um arquivo chamado *arquivo de manifesto*. O arquivo de manifesto declara para o sistema os componentes existentes na aplicação, identifica as permissões necessárias (como acesso à internet ou acesso de leitura aos contatos), declara a versão mínima das bibliotecas requeridas, as funcionalidades de hardware e software (como acesso ao microfone ou comunicação via bluetooth), outras bibliotecas necessárias, entre outras informações.

Por fim, uma aplicação pode ser constituída de mais do que simplesmente código em Java: pode conter, por exemplo, imagens, arquivos de áudio, vídeo, certificados criptográficos, etc. Uma aplicação Android permite que esses recursos sejam definidos separadamente do código e empacotados junto com a aplicação compilada[9]. Esta característica permite maior flexibilidade na manutenção de recursos (separada da manutenção do código), além de possibilitar a otimização dos recursos para mais de uma configuração de dispositivo.

5.1.3 Arcabouço de desenvolvimento

A ferramenta básica para desenvolvimento no Android é a IDE Eclipse[10]. O Eclipse permite a instalação de plugins a partir da configuração de repositórios arbitrários, e desta forma é possível instalar o plugin Android Development Tools[11] (ADT) que configura a ferramenta para o desenvolvimento de aplicações Android. Esta configuração consiste no gerenciamento de kits de desenvolvimento para as diferentes versões do sistema operacional, que determinam uma série de relações entre diferentes partes da aplicação e proveem a API com todas as funcionalidades disponíveis no sistema operacional. Com isto, é possível estruturar uma aplicação da forma descrita na seção anterior, compilar o código fonte para *bytecode* Java, e executar a aplicação em um emulador ou em um dispositivo real. Existem versões do Eclipse empacotadas pelos desenvolvedores do Android de forma que já vêm prontas com todo o arcabouço necessário para o desenvolvimento[12].

Cada versão do sistema operacional Android é compatível com a API das versões anteriores. Isso significa que um aplicativo desenvolvido para uma certa versão pode ser executado em versões mais novas do sistema. A Figura 5.2 mostra a evolução temporal da porcentagem de aparelhos rodando cada versão do sistema operacional. No momento da escrita deste texto, a versão mínima do sistema operacional que deve ser considerada para que um aplicativo possa ser executado em quase a totalidade dos dispositivos é a 2.1.

Uma outra extensão interessante é o kit de desenvolvimento para "código nativo". No jargão específico da linguagem Java, código nativo é aquele escrito em C/C++ e que pode interagir com código em Java através de uma interface da máquina virtual Java chamada JNI (Java Native Interface). A utilização de código nativo no Android é feita a partir do NDK (Native Development Kit), mantida pelos desenvolvedores do Android e que pode ser encontrado também no site oficial do sistema[13]. Neste trabalho não é utilizado o NDK, mas já existe pesquisa em andamento no Grupo de Computação Musical do IME/USP com a utilização desta ferramenta para comparação, por exemplo, do tempo de execução de diversos algoritmos escritos em Java com os mesmos algoritmos escritos em C/C++.

5.2 Processamento de áudio em tempo real em Android

Esta seção descreve os itens disponíveis na infraestrutura de desenvolvimento da plataforma Android para lidar com captura, manipulação periódica e emissão de sinais de áudio, tendo como objetivo enfatizar a operação em tempo real. Serão descritas as classes que permitem a interface com dispositivos de captura e emissão de áudio, bem como agendamento para execução de métodos

[9] http://developer.android.com/guide/topics/resources/index.html
[10] http://www.eclipse.org
[11] http://developer.android.com/tools/sdk/eclipse-adt.html
[12] http://developer.android.com/sdk/index.html
[13] http://developer.android.com/tools/sdk/ndk/index.html

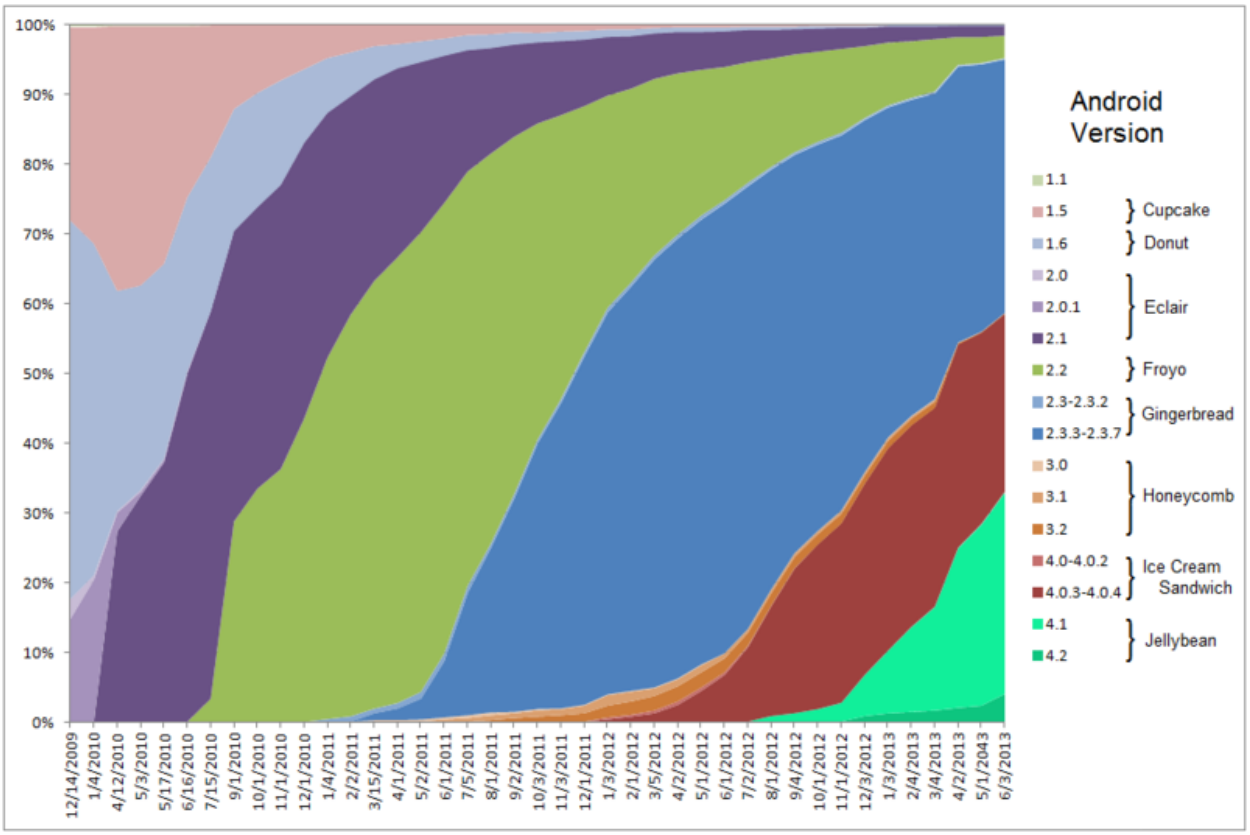

Figura 5.2: *Histórico de porcentagem das distribuições de Android (Wikipedia).*

periódicos. Ao final da seção será descrito o aplicativo desenvolvido, que utiliza esta infraestrutura para prover um ambiente de realização de processamento em tempo real, bem como automatização de testes.

5.2.1 Fontes de sinais de áudio

O nível de abstração proporcionado pela utilização da API em Java em um sistema operacional baseado em Linux (e portanto nos padrões POSIX[14]) permite o acesso a diversas fontes de sinais de áudio. Algumas limitações são impostas pelo esquema de permissões do sistema mas, como veremos, não impedem a implementação de alternativas para acesso às diversas fontes de sinais.

Possíveis fontes de sinais de áudio são: arquivos de áudio (representados por diversos tipos de codificação), microfones (às vezes há mais de um microfone em um dispositivo) e sinais transmitidos por rede (Wi-Fi, 3G, etc). Cada tipo de fonte será descrita separadamente a seguir.

Obtendo sinais de áudio do microfone

O acesso ao sinal de áudio capturado pelo microfone é feito através da classe `AudioRecord`[15], que permite a configuração de diversos parâmetros para acesso a um *buffer* com os valores das amostras. Ao instanciar um objeto desta classe, devem ser informados a forma de acesso ao microfone (detalhada a seguir), a taxa de amostragem desejada, o número de canais, o formato das amostras de áudio (8 ou 16 bits) e o tamanho do *buffer* onde serão escritas as amostras.

O microfone de um aparelho móvel é geralmente utilizado para capturar a voz do usuário em diferentes situações como, por exemplo, durante uma chamada, durante a gravação de lembretes ou para processamento por aplicativos de reconhecimento de voz. É por causa dessa diversidade de

[14] http://standards.ieee.org/develop/wg/POSIX.html
[15] https://developer.android.com/reference/android/media/AudioRecord.html

aplicações que é necessário informar ao construtor da classe `AudioRecord` a forma de acesso a cada microfone. As opções possíveis são valores constantes da classe `MediaRecorder.AudioSource`[16], estão sujeitas a disponibilidade em cada modelo de aparelho, e são as seguintes:

- `MIC`: Áudio bruto do microfone.
- `CAMCORDER`: Áudio bruto capturado de um microfone com a mesma orientação da câmera (se disponível).
- `VOICE_DOWNLINK`: Áudio bruto recebido durante uma chamada.
- `VOICE_UPLINK`: Áudio bruto enviado durante uma chamada.
- `VOICE_CALL`: Soma dos sinais de áudio brutos enviado e recebido durante uma chamada.
- `VOICE_COMMUNICATION`: Áudio do microfone, pré-processado para comunicação por voz, ou seja, submetido a processos de cancelamento de eco, controle de ganho automático, entre outros.
- `VOICE_RECOGNITION`: Áudio do microfone, pré-processado para reconhecimento de voz.

Um exemplo de como pode ser feita a instanciação de um objeto da classe `AudioRecord` pode ser visto a seguir:

```
1 AudioRecord recorder = new AudioRecord(
2   AudioSource.MIC,                // - Fonte de audio.
3   44100,                          // - Taxa de amostragem (Hz).
4   AudioFormat.CHANNEL_IN_MONO,    // - Configuracao de numero de canais.
5   AudioFormat.ENCODING_PCM_16BIT, // - Codificacao das amostras.
6   AudioRecord.getMinBufferSize(   // - Tamanho do buffer de audio (neste caso, o
7     44100,                        //   minimo para a configuracao atual).
8     AudioFormat.CHANNEL_IN_MONO,
9     AudioFormat.ENCODING_PCM_16BIT));
```

Instanciação de `AudioRecord`

Vale comentar que a configuração de gravação com taxa de amostragem igual a 44.100 Hz, número de canais igual a 1 e codificação em PCM 16 bits é a única que tem garantia de funcionamento em todos os dispositivos. Note também que existe um tamanho mínimo para o *buffer* de áudio (utilizado internamente pelo gravador), que pode ser obtido através do método estático `AudioRecord.getMinBufferSize()`. Apesar da existência de um tamanho mínimo para este *buffer* interno de áudio, é importante saber que não há garantia de gravação sem perda de amostras se o sistema estiver com muita carga.

A leitura do valor das amostras é feita através do método de instância `AudioRecord.read()`, que recebe como argumentos um *buffer* para onde devem ser copiadas as amostras, o valor do índice a partir do qual os dados devem ser escritos no *buffer* e a quantidade de amostras a serem lidas do microfone. As amostras são codificadas como áudio PCM e portanto são representadas por símbolos do tipo `byte` (com valores entre -128 e 127) ou `short` (com valores entre -32768 e 32767), e portanto algum cuidado tem que ser tomado quando do processamento destas amostras para que não haja problemas com os limites de cada tipo de representação.

Um exemplo de método para leitura dos valores de amostras do microfone e escrita em um *buffer* para posterior processamento pode ser visto abaixo:

[16] https://developer.android.com/reference/android/media/MediaRecorder.AudioSource.html

```
1 public void readLoop(short[] inputBuffer, int readBlockSize) {
2   while (isRunning) {
3     recorder.read(
4       inputBuffer,                              // - buffer para as amostras.
5       ((j++) % buffer.length) * readBlockSize, // - indice para escrita no buffer.
6       readBlockSize);                           // - tamanho do bloco lido.
7   }
8 }
```

Note que seguidas leituras de blocos de tamanho `readBlockSize` serão feitas, até que a variável de controle `isRunning` seja tornada falsa para interromper a leitura. O valor de `readBlockSize` será discutido na seção 5.2.2, assim como a questão da conversão de representação PCM para ponto flutuante e vice-versa.

Gravando o sinal do microfone em um arquivo

Existe uma opção de mais alto nível para capturar, codificar (com perdas) e armazenar o sinal do microfone, sem no entanto permitir o acesso ao valor das amostras em tempo real. A classe `MediaRecorder`[17], disponível na API do Android, pode ser utilizada para gravar o sinal do microfone em um arquivo codificado em AMR (otimizado para codificação de fala), MPEG4 ou 3GPP (utilizado em plataformas com acesso a redes 3G). Estes arquivos podem ser posteriormente abertos para leitura e processamento.

Algumas desvantagens desta abordagem são que todos os formatos são proprietários, o que limita as possibilidades de uso, além de utilizarem modelos de compactação com perdas. A alternativa para armazenar o sinal de áudio bruto é fazer a leitura da entrada com a classe `AudioRecord`, da forma descrita anteriormente, e implementar algum formato de representação sem perdas, como por exemplo FLAC ou WAV.

Obtendo sinais de áudio a partir de arquivos

A API do Android não oferece suporte à leitura e decodificação de arquivos de áudio ou vídeo para acesso e manipulação do valor das amostras do sinal. A alternativa que resta aos desenvolvedores de aplicações é implementar bibliotecas específicas para os tipos de arquivo que utilizam.

Para o contexto deste trabalho foi desenvolvida uma classe chamada `AudioStream`, na qual foi encapsulado todo o acesso a sinais de áudio. O acesso ao sinal de áudio do microfone é feito através de uma subclasse chamada `MicStream`, e o acesso a sinais armazenados em arquivos WAV, por sua vez, é feito através de uma outra subclasse chamada `WavStream`.

Uma vez que todo o acesso a um sinal de áudio está encapsulado, é possível responsabilizar a classe `AudioStream` também pela manipulação e reprodução do sinal de áudio. As próximas seções discutem a forma de implementação deste mecanismo utilizada neste trabalho.

5.2.2 Agendamento dos ciclos de processamento

Para implementar um ambiente de processamento de sinais é necessário representar no sistema algumas características do sinal como taxa de amostragem, número de canais e tipo de codificação das amostras. Além disso, o próprio processamento também possui parâmetros tais como tamanho do bloco de processamento e fator de sobreposição (que serão abordados mais adiante). Com estes parâmetros, é possível determinar o período do ciclo de processamento e agendar uma função de manipulação para execução em intervalos periódicos (veja a Seção 1.1.1), acessando diferentes seções de um *buffer* circular.

Na Seção 3.2.5 foi descrito como o uso da frequência de transbordamento de um contador de 8 bits pode ser utilizado no Arduino para controlar a execução da função de amostragem, manipulação

[17] https://developer.android.com/reference/android/media/MediaRecorder.html

e emissão do sinal de áudio. Já na Seção 4.2.4 vimos como utilizar a infraestrutura de tempo real do Pure Data para controlar a manipulação periódica de amostras usando a GPU. No caso do sistema Android, a forma escolhida para viabilizar a manipulação periódica de sinais de áudio foi a utilização de funções de agendamento.

Uma função de manipulação agendada pode não ser necessária em um contexto no qual o mecanismo de leitura das amostras já possui um controle de tempo real (como é o caso da leitura feita a partir do microfone), pois as amostras são entregues para processamento exatamente com a mesma taxa que devem ser devolvidas para reprodução (supondo que a frequência de amostragem de entrada e saída seja a mesma). Apesar disso, quando se deseja realizar processamento em tempo real de um sinal armazenado em arquivo, todas as amostras estão disponíveis desde o começo, e portanto é necessário que o controle do período do ciclo de processamento seja independente da disponibilidade de amostras na fonte do sinal. Se este cuidado não for tomado, a alteração nos parâmetros do processamento num certo instante pode afetar amostras que correspondem a pontos ainda muito distantes no futuro.

Desta forma, a opção feita foi por agendar uma função de manipulação que acessa periodicamente um *buffer* circular que contém o sinal a ser modificado. Um *buffer* circular é um vetor de tamanho finito que tem este nome por causa da forma como o acesso aos seus índices é feito. As amostras digitais de áudio geradas pelo sistema em intervalos de tempo igualmente espaçados são gravadas sequencialmente no *buffer*, e o incremento dos índices de leitura e escrita é feito de forma modular (utilizando como módulo o tamanho do *buffer*).

Cabe comentar que, segundo a documentação do Android, não existe garantia sobre a precisão do agendamento feito através das classes disponíveis na API, de forma que pode ocorrer flutuação no período de execução dos métodos agendados dependendo da carga do sistema [18]. Infelizmente, sem utilizar recursos das camadas mais baixas do sistema (para os quais seriam necessários privilégios especiais), as classes da API são as únicas opções que existem para a realização do agendamento.

Bloco de processamento e fator de sobreposição

O bloco de processamento, neste caso, corresponde à seção do *buffer* circular que será considerada em cada ciclo de processamento, e seu tamanho (em número de amostras) depende do algoritmo que será utilizado para manipulação do sinal e do efeito pretendido. Como visto na Seção 1.1.1, para um bloco de processamento com período igual a N amostras e uma taxa de amostragem de R Hz, um atraso mínimo de $\frac{N}{R}$ segundos entre a entrada do sinal original e a saída do sinal modificado é inevitável, pois este é o tempo mínimo necessário para acúmulo de N amostras.

O *fator de sobreposição* é a relação entre o tamanho do bloco de processamento e o incremento no índice de leitura do *buffer* circular a cada ciclo de processamento. Existem diversas razões para permitir um incremento no índice de leitura diferente do tamanho do bloco de processamento. Para processamentos no domínio da frequência, por exemplo, a ideia de usar um fator de sobreposição diferente de 1 é obter uma maior resolução temporal de análise do sinal sem perder resolução espectral. Por outro lado, existem processamentos puramente temporais que também podem se beneficiar da sobreposição de blocos de processamento, como por exemplo *Time Stretching* ("esticamento" temporal) e *Pitch Shifting* (alteração de altura musical) executados no domínio do tempo utilizando *overlap-add* (Zölzer, 2002).

Assim, para um fator de sobreposição igual a M, o $j-$ésimo ciclo de processamento altera a seção do *buffer* correspondente às amostras do intervalo $\left[j\frac{N}{M}, j\frac{N}{M} + N - 1\right]$. Isto também determina o período do ciclo de processamento, que é dado por $T = \frac{N}{MR}$. É necessário garantir que as amostras de áudio correspondentes ao $j-$ésimo ciclo de processamento já tenham sido capturadas antes do início do ciclo. Para isto, a leitura da entrada deve ser feita em blocos de tamanho N/M e deve-se aguardar que o *buffer* possua pelo menos N amostras antes de que o primeiro ciclo de processamento seja iniciado.

[18] https://developer.android.com/reference/java/util/Timer.html

Desta forma, uma vez determinados a taxa de amostragem, o tamanho do bloco de processamento e o fator de sobreposição, pode-se agendar uma função de processamento que deve será executada, idealmente, nos valores de tempo $t_j = \frac{N}{R} + j(\frac{N}{MR})$ segundos para $j \in \mathbb{N}$. A saída destas funções deve estar disponível para reprodução imediata antes do início do próximo ciclo, e portanto o tempo efetivamente disponível para a manipulação do sinal é menor do que período do ciclo de processamento.

O agendamento pode ser feito utilizando-se a classe `ScheduledExecutorService`[19] da API do Android, que permite a indicação do período do agendamento em nanossegundos e assim possibilita um agendamento mais preciso. No trecho de código abaixo, `scheduler` é uma variável do tipo `ScheduleExecutorService` e é agendada de acordo com o cenário descrito acima:

```
1 // agenda a funcao de processamento periodica
2 public void scheduleDspCallback() {
3   System.gc(); // forca o recolhimento de lixo
4   try {
5     scheduler = Executors.newScheduledThreadPool(1);
6     dspTask = scheduler.scheduleAtFixedRate(
7       dspCallback,                  // - funcao de processamento.
8       (float) N/R*Math.pow(10,9),   // - atraso para a primeira execucao.
9       (float) N/(MR)*Math.pow(10,9),// - intervalo entre execucoes.
10      TimeUnit.NANOSECONDS);        // - unidade de tempo utilizada.
11  } catch (Exception e) {
12    e.printStackTrace();
13  }
14 }
```

Agendamento da função de processamento

A função de processamento agendada corresponde, na verdade, ao método `run()` de um objeto do tipo `Runnable`[20], que será visto com mais cuidado na próxima seção.

5.2.3 Manipulação e reprodução do sinal

A classe `AudioTrack`[21] permite a escrita de valores de amostras (codificadas em PCM 8 bits ou 16 bits) em um *buffer* para reprodução pelo hardware de áudio. Esta classe possui dois modos de operação, *static* e *streaming*, que diferem quanto à quantidade de áudio que será reproduzida e as condições de transferência do sinal da camada de aplicação para o hardware do aparelho. Apesar do modo *static* oferecer menor latência, para que seja possível escrever dados para reprodução de forma contínua é necessária a utilização do modo *streaming*. A escrita das amostras para reprodução é feita através do método `write()`, que será visto adiante. Um objeto da classe `AudioTrack` pode ser instanciado da seguinte forma:

```
1 AudioTrack track = new AudioTrack(
2   AudioManager.STREAM_MUSIC,      // - stream de audio a ser utilizado.
3   44100,                          // - taxa de amostragem.
4   AudioFormat.CHANNEL_OUT_MONO,   // - numero de canais.
5   AudioFormat.ENCODING_PCM_16BIT, // - codificacao das amostras.
6   audioStream.getMinBufferSize(), // - tamanho do buffer.
7   AudioTrack.MODE_STATIC);        // - modo de operacao
```

Instanciação de `AudioTrack`

[19] https://developer.android.com/reference/java/util/concurrent/ScheduledExecutorService.html
[20] https://developer.android.com/reference/java/lang/Runnable.html
[21] https://developer.android.com/reference/android/media/AudioTrack.html

Até agora já foi discutido como ler amostras de um sinal de áudio com `AudioRecord`, como agendar a execução de uma função de processamento com `ScheduleExecutorService`, e como preparar o sistema para reprodução utilizando `AudioTrack`. O que falta agora para completar o desenvolvimento de uma infraestrutura para processamento de sinais em tempo real é apenas a manipulação de fato do sinal e a escrita das amostras resultantes para reprodução.

Na seção 5.2.2, foi dado um exemplo de como agendar uma função de processamento utilizando um objeto do tipo `Runnable` chamado `dspCallback`. O código a seguir descreve uma possível implementação para este objeto, usando um *buffer* de entrada e outro de saída para permitir a sobreposição de blocos durante o processamento:

```
short inputBuffer;  // - buffer circular de amostras de entrada.
short outputBuffer; // - buffer circular de amostras processadas.
int j;              // - indice de leitura dos buffers circulares.
int L = inputBuffer.length;
Runnable dspCallback = new Runnable() {
  public void run() {
    double performBuffer[] = new double[L];
    // 1. conversao das amostras de (PCM) short para double
    for (int i = 0; i < N; i++)
      performBuffer[i] = (double)
        inputBuffer[(j*N/M + i) % L] / Short.MAX_VALUE;
    // 2. executa o algoritmo escolhido
    dspAlgorithm.perform(performBuffer);
    // 3. conversao das amostras de double para (PCM) short
    for (int i = 0; i < N; i++) {
      if (i >= N-N/M) // previne overlap-add nos ultimos indices
        outputBuffer[(j*N/M + i) % L] = 0;
      outputBuffer[(j*N/M + i) % L] += (short) // overlap-add
        (performBuffer[i] * Short.MAX_VALUE);
    }
    // 4. escrita das amostras para reproducao
    track.write(outputBuffer, ((j++)*N/M) % L, N);
  }
};
```

Classe que implementa o ciclo de processamento

O método `run()` do objeto acima consiste em, essencialmente, quatro passos: (1) conversão das amostras de PCM para ponto flutuante para permitir sua manipulação numérica; (2) chamada do método `perform()` do algoritmo escolhido para manipulação das amostras; (3) conversão de ponto flutuante de volta para PCM; e (4) enfileiramento das amostras para reprodução. Se o fator de sobreposição for diferente de 1, então algum cuidado tem que ser tomado com a sobreposição das amostras de saída. No exemplo acima, é feita uma soma simples dos sinais sobrepostos, mas outras técnicas podem ser implementadas para obter outros efeitos ou algoritmos mais eficientes.

5.2.4 Implementação

Para investigar as possibilidades do uso do Android para processamento de áudio em tempo real, o ambiente foi programado para poder executar algoritmos arbitrários sobre um fluxo de áudio dividido em blocos de N amostras, permitindo a variação do algoritmo ao longo da execução. O software desenvolvido é uma aplicação Android[22] (usando a API nível 7[23]) que consiste em uma interface gráfica e um modelo de objetos para processamento de áudio que mantém um registro do tempo que certas tarefas específicas tomam enquanto são executadas.

[22] http://www.ime.usp.br/~ajb/DspBenchmarking.apk

[23] A API nível 7 é compatível com Android OS versão 2.1 ou superior.

A interface gráfica permite tanto a utilização ao vivo do processamento de áudio quanto a execução de testes automatizados para avaliação do desempenho do processamento. Amostras de áudio podem ser obtidas diretamente do microfone ou a partir de arquivos WAV, e o bloco DSP pode ser configurado para tamanhos 2^i para $0 \leq i \leq 13$. O limite superior é configurável porém, sob uma taxa de amostragem igual a 44.1 KHz, um bloco de $N = 2^{13}$ amostras corresponde a uma latência de 186 μs, o que é razoavelmente perceptível.

A opção por não executar os testes para blocos maiores se justifica pela limitação do tempo disponível para execução dos testes (em alguns aparelhos os testes já demoram cerca de uma hora). Além disso, com os dados colhidos já foi possível obter os parâmetros máximos para a convolução e síntese aditiva, e também esboçar uma comparação entre os diferentes dispositivos quanto à execução da FFT.

Como a configuração dos aparelhos que rodam o sistema Android é bastante heterogênea e o aplicativo é executado na camada mais alta do sistema de forma concorrente com diversos outros processos, uma métrica interessante de ser investigada é a quantidade de tempo que de fato está disponível para manipulação das amostras do sinal de áudio. Quanto menos poderoso o dispositivo (em termos de velocidade de processamento, memória, etc), mais tempo a infraestrutura que viabiliza o processamento (incluindo leitura e emissão de amostras) demora para ser executada, e menos tempo está disponível para o processamento propriamente dito. Além desta métrica, é necessário investigar, a exemplo do que foi feito nas outras plataformas, a quantidade de tempo de computação necessário para realizar tarefas comuns de processamento. Para obter todos estes valores, o programa mantém um registro dos instantes de leitura e escrita de amostras e do tempo de execução dos algoritmos DSP. Com esta informação, é possível ter uma ideia do desempenho do dispositivo e do sistema desenvolvido, comparando o tempo tomado pelas distintas tarefas DSP com o tempo disponível para computação de um bloco de N amostras.

Para a investigação do sistema Android, três cenários de processamento foram desenvolvidos. No primeiro cenário, nenhuma modificação no sinal é realizada, somente o tempo entre a entrada e saída de amostras é medido. No segundo cenário, um algoritmo de reverberação simples é computado usando um filtro passa tudo; e num terceiro cenário uma FFT do sinal é calculada para um bloco de amostras. Em cada cenário, o programa mantém um registro de todo o período do ciclo DSP (incluindo a conversão entre representações PCM e ponto flutuante, código para acompanhamento de tempo, e enfileiramento de amostras para reprodução), e também do tempo de execução do algoritmo DSP.

A seguir, será apresentado o modelo orientado a objetos desenvolvido para a medição do desempenho de algoritmos genéricos de DSP em tempo real em ambiente Android. Em seguida, serão descritos os fluxos de dados de parâmetros e finalmente os resultados obtidos para análise.

Modelo orientado a objetos para processamento de áudio

O diagrama de classes para as partes principais do modelo orientado a objetos desenvolvido podem ser vistos na Figura 5.3. O aplicativo é composto de duas atividades: uma para performance ao vivo e uma para medição automática de desempenho. A classe `LiveActivity` permite ao usuário escolher a fonte do sinal de áudio e também alterar as configurações do processamento, como tamanho de bloco, algoritmo e parâmetros de processamento (veja a Figura 5.4). Alternativamente, a classe `TestActivity` executa um conjunto de testes automatizados e gera um arquivo de relatório com os resultados. Ambas as atividades estendem uma classe abstrata mais geral chamada `DspActivity`.

Neste modelo, as atividades DSP são responsáveis por diversas tarefas, sendo uma delas manter a interface gráfica atualizada com informações sobre o desempenho do dispositivo. Uma instância da *thread* `SystemWatch` reúne informações sobre o dispositivo e entrega para o objeto do tipo `DspActivity`, que por sua vez altera a interface gráfica de forma que reflita o estado do sistema. Além disso, a atividade pode combinar estes valores com parâmetros adquiridos a partir de sliders, botões e outros *widgets* visuais na interface gráfica, passando seus valores para a *thread* DSP, descrita a seguir.

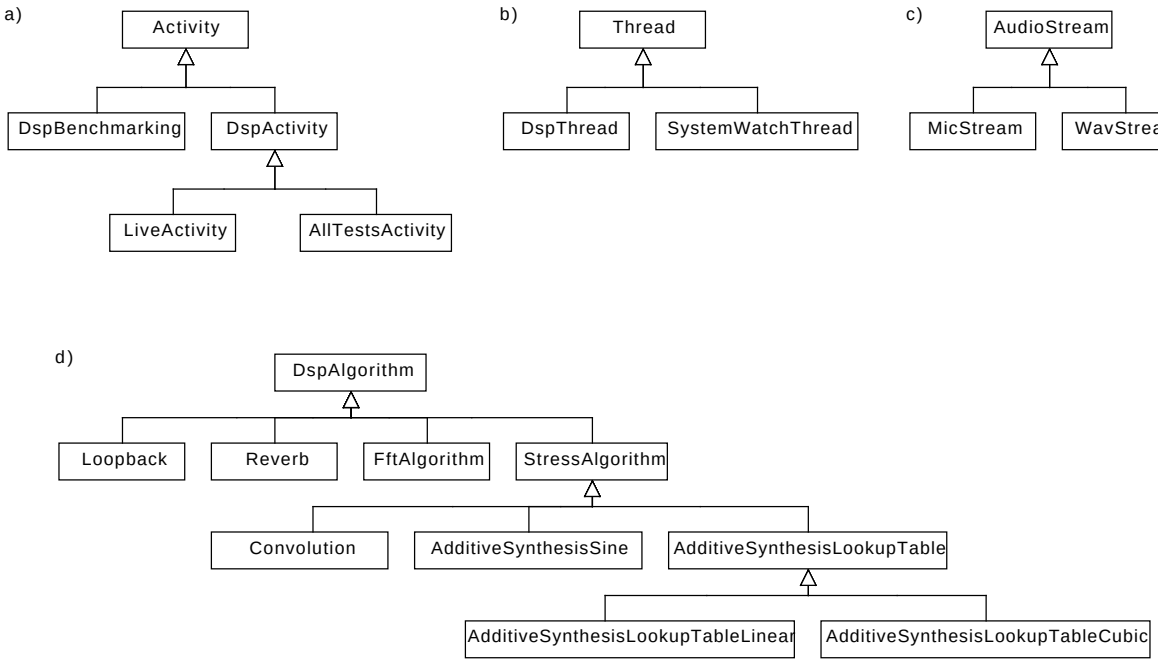

Figura 5.3: *Diagrama de classes para as principais partes do nosso modelo de objetos: (a) o programa possui três interfaces gráficas (ou "atividades"), a principal (`DspBenchmarking`), que dá acesso às outras duas, uma para uso "ao vivo" (`LiveActivity`) e outra para teste de desempenho (`TestActivity`); (b) duas* threads *concorrentes tomam conta do processamento de sinais (`DspThread`) e da reunião de informações sobre o sistema (`SystemWatchThread`); (c) sinais de áudio podem ser obtidos a partir do microfone (`MicStream`) ou de arquivos WAV (`WavStream`); (d) fluxos de áudio podem ser modificados por diversos algoritmos que estendem uma mesma classe abstrata (`DspAlgorithm`).*

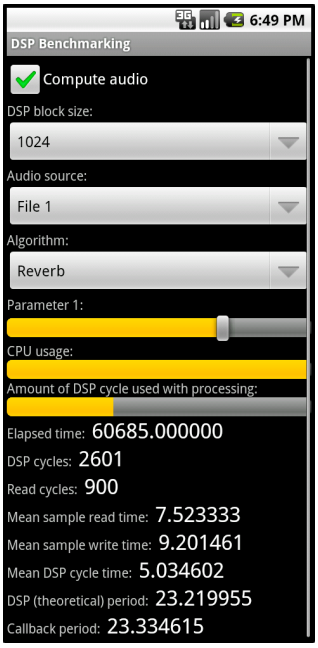

Figura 5.4: *A interface gráfica controlando um processamento ao vivo. O usuário pode escolher o tamanho do bloco DSP, a fonte do sinal de áudio (microfone ou arquivos WAV predefinidos) e o algoritmo DSP que será executado. Além disso,* widgets *gráficos podem fornecer parâmetros de entrada explícitos, enquanto estatísticas visuais e numéricas fornecem informações sobre o estado do sistema.*

Sendo executada concorrentemente com a atividade principal e a *thread* `SystemWatch`, uma instância da classe `DspThread` agenda um método para ser executado a cada ciclo DSP (veja a Figura 5.5). A manipulação do sinal é definida por subclasses da classe abstrata `DspAlgorithm`. O método agendado lê amostras de áudio de um *buffer* que contém o sinal de entrada e escreve o resultado do processamento em um *buffer* de saída. O período do ciclo DSP é $\Delta_N = N/R$ segundos, onde N é o período do bloco DSP (em amostras) e R é a frequência de amostragem (em Hz), e corresponde ao máximo período de tempo que o método agendado pode demorar para escrever as novas amostras na fila de reprodução do hardware de áudio, sob a restrição de operação em tempo real.

Todas as classes que estendem `DspAlgorithm` têm que implementar um método chamado `perform()` que age sobre um *buffer* para gerar o efeito desejado. A instância de `DspThread` sendo executada é responsável pelo instanciamento do algoritmo selecionado (seja pelo usuário ou pelo teste em questão) e também por seu agendamento, de forma que o método seja executado a cada ciclo DSP para modificar os dados no *buffer*.

Os algoritmos podem acessar e produzir diversos parâmetros. Sliders na interface gráfica e valores de sensores (câmera, acelerômetro, proximidade, etc) podem ser utilizados como entrada através de métodos específicos. Parâmetros definidos pelo usuário, como por exemplo coeficientes derivados da análise de blocos de áudio, podem ser periodicamente emitidos como strings ou sobre uma conexão Wi-Fi ou bluetooth ou escritos em arquivos.

Como visto na Seção 5.2.1, é relativamente simples utilizar a API do Android para gravar amostras a partir do microfone do dispositivo, instanciando a classe `AudioRecord`. Por outro lado, as classes de gerenciamento de mídia presentes na API do Android só podem ser utilizadas para tocar arquivos inteiros (usando a classe `MediaPlayer`) ou adicionar efeitos sonoros à cadeia de saída de áudio (usando a classe `AudioEffect`), e não podem ser utilizadas para acessar (ou seja, ler e modificar) o sinal digital no nível do aplicativo. Por causa destas restrições, foi necessário implementar uma classe leitora de WAV baseada na especificação do cabeçalho padrão[24] e em configurações de profundidade de bits e frequência de amostragem específicas para os arquivos de teste utilizados.

A lista a seguir resume a funcionalidade das principais classes do modelo DSP:

- **Atividades (`LiveActivity`, `TestActivity`):** Interface gráfica para controle do processamento, e gerenciamento de *threads*.
- **DSP thread (`DspThread`):** entrada e saída de sinais, agendamento de métodos e registro de tempo. Algumas partes importantes desta classe são:
  - Uso de `MicStream` para gravação a partir do microfone e de `WavStream` para obtenção de amostras de arquivos WAV.
  - Uso da classe `AudioTrack` para escrever no *buffer* de saída de áudio após o processamento (esta é uma operação não bloqueante).
  - Conversão de amostras a partir do tipo 16-bits para representação em ponto flutuante e vice-versa. Alguns algoritmos de DSP podem ser implementados sobre inteiros, mas para muitas aplicações a representação em ponto flutuante é necessária, e então a opção feita foi por considerar esta como uma tarefa comum que necessita ser levada em conta para medição do desempenho em ambientes que se propõem a executar algoritmos arbitrários de processamento de áudio.
  - Agendamento de um método `perform()` de uma subclasse de `DspAlgorithm` para ser executado a cada ciclo DSP usando o mecanismo de agendamento de `AudioStream`.
  - Passagem de parâmetros da interface gráfica para o algoritmo DSP e vice-versa.

[24] http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html

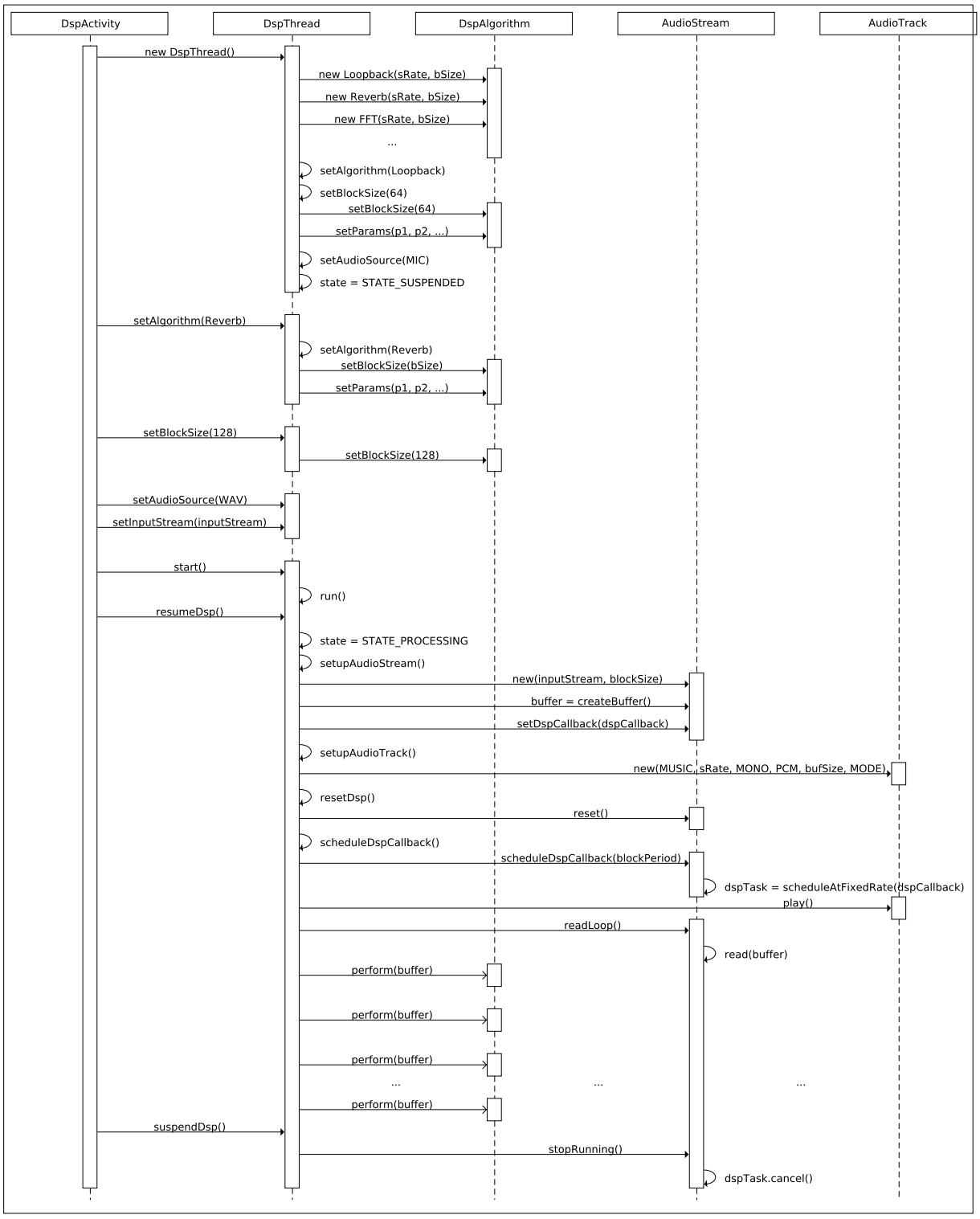

Figura 5.5: *Diagrama de sequência da parte principal do modelo DSP orientado a objetos em funcionamento. A atividade inicia uma* thread *DSP, que em seguida instancia subclasses de* `DspAlgorithm` *e* `AudioStream`*. A* thread *DSP então agenda um método* `perform()` *que é chamado a cada ciclo DSP, e inicia a leitura de amostras do* buffer *de entrada. Chamadas subsequentes, periódicas e assíncronas, ao método* `perform()` *do algoritmo DSP modificam o sinal de entrada e escrevem o resultado no* buffer *de saída.*

- ***Thread* de monitoramento de sistema (`SystemWatch`):** obtém informações sobre o sistema (uso da CPU e valores de sensores) e alimenta a interface gráfica e os algoritmos DSP.
- **Algoritmos (subclasses de `DspAlgorithm`):** Definição dos algoritmos DSP. Possuem um método com assinatura definida para realizar computações em bloco durante ciclos DSP.

Como toda a infraestrutura descrita acima foi implementada sobre a pilha de software do Android, ou seja, como uma aplicação Android escrita em Java, preocupações comuns relacionadas a gerenciamento de memória tiveram que ser abordadas. Objetos nunca são criados a não ser que sejam realmente necessários: é possível, por exemplo, suspender o processamento DSP para mudar suas características (tamanho do bloco, algoritmo, etc) sem encerrar a *thread* DSP, e em seguida reiniciá-la através do reagendamento do método `perform()`. O coletor de lixo também deve ser usado com cautela para garantir que não haverá vazamento de memória e ao mesmo tempo evitar qualquer interferência na frequência ou período de execução do método DSP.

Usando o modelo descrito nesta seção, foram desenvolvidas duas formas de medir o desempenho de cada dispositivo, que serão descritas nas próximas seções.

5.3 Resultados e discussão

Como o poder computacional dos dispositivos com sistema Android são muito diversos, torna-se interessante fornecer meios de avaliação de desempenho para processamento de áudio em tempo real para qualquer dispositivo que execute o sistema. A programação para sistemas Android permite a execução de um aplicativo em qualquer dispositivo, desde que o aplicativo tenha sido desenvolvido com uma versão da API menor ou igual a do dispositivo em questão. Utilizando a infraestrutura descrita na Seção 5.2, foi desenvolvido um aplicativo que oferece a possibilidade tanto de execução "ao vivo", fornecendo em tempo real dados sobre o desempenho do dispositivo, quanto de realização de uma bateria de testes automatizados e envio dos resultados via email para o desenvolvedor do aplicativo.

Esta seção descreve os resultados obtidos por uma rodada de testes em aparelhos de voluntários. Na Seção 5.3.1 é descrito como o experimento foi projetado para ser executado em diversos aparelhos sem tomar muito tempo e de forma que os resultados pudessem ser recuperados para análise. A Seção 5.3.2 descreve os algoritmos executados e os primeiros resultados obtidos. Na Seção 5.3.3 são discutidos os resultados dos testes de estresse dos aparelhos. Finalmente, a Seção 5.3.4 discute os resultados e aponta para direções de estudos futuros.

5.3.1 Projeto de experimentos

Como discutido anteriormente, a análise de desempenho é feita por um aplicativo iniciado pelo usuário. A interação do usuário nos testes automatizados é mínima, necessitando apenas de um clique para iniciar os testes e outro clique para enviar os resultados dos experimentos via email. No momento dos experimentos, o Grupo de Computação Musical do IME/USP contava com poucos aparelhos com Android disponíveis para desenvolvimento e testes, e assim a opção foi enviar um chamado à participação para estudantes e professores.

Desta forma, sabia-se que a maioria dos dispositivos experimentados seria de uso estritamente pessoal, e portanto foi necessário manter o tempo de experimento em um limite razoável. O experimento foi projetado em duas fases. Na primeira fase, diferentes algoritmos com parâmetros fixos são executados para diferentes tamanhos de bloco. A medição do tempo de execução permite determinar a porcentagem do período teórico do ciclo DSP que é ocupada pelo processamento do sinal de áudio. Na segunda fase, algoritmos que possuem parâmetros que podem ser aumentados, de forma a aumentar a intensidade computacional, são executados para diferentes parâmetros e diferentes tamanhos de bloco. Assim, pode-se estimar o parâmetro máximo viável para um certo algoritmo em um certo dispositivo (veja a Seção 2.1.1).

A busca pelo número máximo do parâmetro de um algoritmo é feita em duas fases distintas. Durante a primeira fase, o valor do parâmetro é aumentado exponencialmente, até que o tempo médio de execução do algoritmo exceda o período teórico do ciclo DSP. Neste ponto, é possível estabelecer duas invariantes: (1) que o dispositivo é capaz de executar tal algoritmo com um certo parâmetro igual a m em tempo real, e (2) que o dispositivo não consegue executar o mesmo algoritmo com parâmetro igual a M, onde $M = 2m$ neste momento. Uma vez que este ponto tenha sido atingido, uma busca binária recursiva é iniciada sobre os valores $k = (M + m)/2$, aumentando e diminuindo os valores de m e M dependendo do desempenho do dispositivo em cada estágio, para finalmente convergir no número K que é o valor máximo do parâmetro para o qual o dispositivo consegue computar o algoritmo durante um ciclo DSP.

O tempo necessário para, por exemplo, estimar o tamanho máximo K de um filtro para um certo tamanho de bloco é limitado pelo logaritmo de K por causa da busca binária. Apesar disto, o número de medições do tempo gasto para o cálculo (para as quais a média é calculada) tem que ser escolhido com cuidado, para manter um tempo de experimentação razoável. Para reduzir o tempo dos testes, cada rodada de um algoritmo foi limitada a 100 ciclos DSP. O resultado é um experimento que automaticamente percorre todos os algoritmos, roda por cerca de 60 minutos, e ao final envia um relatório por email com os resultados para o autor. Para melhorar a qualidade dos resultados e análises, as seguintes instruções foram enviadas para os usuários por email e repetidas na tela no momento da execução dos testes:

> **Bem vindo/a ao DSP Benchmarking**
>
> Muito obrigado por aceitar ajudar este trabalho de mestrado e submeter seu aparelho com Android a uma bateria de testes de processamento de áudio em tempo real.
>
> Por favor, encerre todos os aplicativos que estiverem ativos no telefone e coloque-o no "Modo Avião" antes de executar os testes. Seu dispositivo ficará mudo durante os testes.
>
> O código fonte deste aplicativo e informações sobre como verificar a autenticidade deste pacote estão disponíveis no seguinte endereço:
>
> http://www.ime.usp.br/∼ajb/android

Ao final do período especificado, resultados de um total de 35 dispositivos foram recebidos. A lista de dispositivos que participaram desta pesquisa pode ser vista na Tabela 5.1.

5.3.2 Medição de tempo de diferentes algoritmos

A cada ciclo DSP, um método agendado é executado e processa o bloco atual de amostras do sinal de áudio. Este método inclui a execução de rotinas para medição de tempo, conversão entre a representação em `shorts` e `doubles` (ida e volta), execução do método `perform()` do algoritmo DSP e escrita das amostras para o *buffer* de saída. Para os propósitos desta investigação, chamamos o tempo utilizado para realizar este conjunto de operações de *tempo de callback*. O tempo de callback é, então, o tempo necessário por um conjunto minimal de operações DSP para executar qualquer algoritmo desejado, e assim pode ser comparado com o período teórico do ciclo DSP para determinar a viabilidade de certas operações.

Se o tempo do callback DSP eventualmente exceder o período teórico do ciclo DSP, então o sistema não consegue gerar amostras em tempo real, ou seja, entregar o resultado da computação para reprodução antes que o ciclo DSP termine. Como em dispositivos móveis há, em geral, restrições significativas com relação ao poder computacional, a primeira questão que surge é se o modelo de processamento desenvolvido para este trabalho, que é baseado em Java, inclui código para registro de tempo, converte amostras de um tipo para outro, depende do agendamento presente na API do

Modelo	Fabricante	Modelo da CPU	CPU	RAM (MB)	Versão (API)
GT-P1000L	Samsung	Galaxy Tab	1 GHz	512	2.2 (8)
LG-P500h (1)	LG	Optimus One	600 MHz	512	2.3.3 (10)
LG-P500h (2)	LG	Optimus One	600 MHz	512	2.3.3 (10)
GT-I9000B	Samsung	Galaxy S	1 GHz	512	2.3.3 (10)
LG-P698f (1)	LG	Optimus Net Dual	800 MHz	512	2.3.4 (10)
LG-P698f (2)	LG	Optimus Net Dual	800 MHz	512	2.3.4 (10)
R800i	Sony Ericsson	Xperia PLAY	1 GHz	512	2.3.4 (10)
GT-I8150B	Samsung	Galaxy W	1.4 GHz	512	2.3.5 (10)
XT320	Motorola	Defy Mini	600 MHz	512	2.3.6 (10)
GT-S5830i	Samsung	Galaxy Ace	800 MHz	278	2.3.6 (10)
GT-S5360L	Samsung	Galaxy Y	830 MHz	290	2.3.6 (10)
GT-I8530	Samsung	Galaxy Beam	Dual-core 1 GHz	768	2.3.6 (10)
A750	Lenovo	A750	1 GHz	512	2.3.6 (10)
GT-S5360B	Samsung	Galaxy Y	830 MHz	290	2.3.6 (10)
MB860	Motorola	ATRIX 4G	Dual-core 1 GHz	1024	2.3.6 (10)
Blade	ZTE	Blade	600 MHz	512	2.3.7 (10)
Transformer TF101	Asus	Transformer TF101	Dual-core 1GHz	1024	4.0.3 (15)
NB0026	Multilaser	Vibe	1 GHz	512	4.0.4 (15)
GT-S7562	Samsung	Galaxy S Duos	1 GHz	768	4.0.4 (15)
LG-P990	LG	Optimus 2X	Dual-core 1 GHz	512	4.0.4 (15)
ST25a	Sony	Xperia U	Dual-core 1 GHz	512	4.0.4 (15)
MZ607	Motorola	XOOM 2 Media Edition	Dual-core 1.2 GHz	1000	4.0.4 (15)
Transformer TF101	Asus	Transformer TF101	Dual-core 1 GHz	1024	4.1.1 (16)
MB526	Motorola	Defy+	1 GHz	512	4.1.2 (16)
GT-I8190L	Samsung	Galaxy SIII Mini	Dual-core 1 GHz	1000	4.1.2 (16)
LT26i	Sony-Ericsson	Xperia S	Dual-core 1.5 GHz	10224	4.1.2 (16)
GT-I9300 (1)	Samsung	Galaxy SII	Quad-core 1.4 GHz	1024	4.1.2 (16)
GT-I9300 (2)	Samsung	Galaxy SII	Quad-core 1.4 GHz	1024	4.1.2 (16)
GT-I9300 (3)	Samsung	Galaxy SII	Quad-core 1.4 GHz	1024	4.2.2 (17)
Nexus 7 (1)	Asus	Google Nexus 7	Quad-core 1.2 GHz	1024	4.2 (17)
Nexus 7 (2)	Asus	GoogYle Nexus 7	Quad-core 1.2 GHz	1024	4.2.2 (17)
GT-I9000B	Samsung	Galaxy S	1 GHz	512	4.2.2 (17)
MK16i	Sony-Ericsson	Xperia Pro	1 GHz	512	4.2.2 (17)
Galaxy Nexus 4	Samsung	Galaxy Nexus	Dual-core 1.2 GHz	1024	4.3 (18)
Nexus 4	LG	Nexus 4	Quad-core 1.5 GHz	2048	4.3 (18)

Tabela 5.1: *Tabela de dispositivos testados. Note que existem alguns modelos repetidos: dois LG-P500h (mesma versão da API), dois LG-P698f (mesma versão da API), três GT-I9300 (dois com versão 4.1.2 e um com versão 4.2.2) e dois Nexus 7 (um com versão 4.2 e outro com versão 4.2.2).*

Android, e opera com o mesmo nível de prioridade que outros aplicativos comuns, pode de fato ser utilizado em tempo real.

Para responder a esta questão, o primeiro passo foi executar 3 algoritmos simples e tirar a média do período de execução dos callbacks DSP para diferentes tamanhos de bloco. Estes algoritmos são:

- **Loopback:** um método `perform()` vazio, que retorna imediatamente sem alterar as amostras no *buffer*. Esta implementação é utilizada para estabelecer o overhead intrínseco ao modelo apresentado na Seção 5.2.4
- **Reverb:** um filtro IIR de ordem 3 que emite o sinal gerado pela fórmula $y(n) = -gx(n) + x(n-m) + gy(n-m)$ (Oppenheim *et al.*, 1999).
- **FFT:** Uma implementação em Java do algoritmo da FFT. (Cooley e Tukey, 1965; Sedgewick e Schidlowsk , 1998).

Com estas três implementações, é possível ter uma ideia de como é realizar processamento de áudio em tempo real em diferentes combinações de hardware e software rodando Android. Os resultados serão vistos a seguir.

Análise de desempenho dos algoritmos

Nas Figuras 5.6, 5.7 e 5.8, é possível ver os resultados dos algoritmos de, respectivamente, loopback, reverb e FFT, rodando em dispositivos distintos com diferentes versões da API. Em todas as figuras são apresentados somente os resultados para blocos grandes (de 512 até 8192 amostras) por serem consistentes com os resultados para blocos pequenos. Em todas as figuras também pode-se ver o período teórico do ciclo DSP, que corresponde ao tempo máximo que um método pode demorar para escrever novas amostras na fila do hardware de saída, sob as restrições de tempo real.

É interessante notar que o padrão das curvas para o algoritmo loopback não parece condizer com a quantidade de computação realizada por este algoritmo (ou seja, nenhuma) quando comparado com os algoritmos reverb e FFT. Para blocos de tamanho grande, o padrão observado para os dois últimos parece respeitar a quantidade de computação realizada, enquanto que para alguns dispositivos o algoritmo loopback apresenta maior tempo de execução. Pode-se supor que este padrão esteja relacionado a algum comportamento do escalonador de processos do sistema operacional, que talvez realize a alocação de frações de tempo da CPU de forma mais otimizada quando existe realmente computação sendo executada.

Uma vez que loopback e reverb tomam tempo linear com respeito ao tamanho do bloco, os padrões que ocorrem nos primeiros dois gráficos são esperados. Mesmo assim, estes gráficos são úteis para hierarquizar os dispositivos com respeito a seu poder computacional, e também para fornecer uma avaliação precisa do tempo disponível para computação extra para cada dispositivo. A complexidade computacional da FFT explica a curva vista na Figura 5.8. Espera-se que, para blocos de tamanho grande, a FFT em tempo real se torne inviável para todos os dispositivos.

Em geral, o mecanismo de processamento em tempo real desenvolvido deixa bastante espaço para os algoritmos de processamento. O filtro reverberante é viável para todos os dispositivos e tamanhos de bloco, e a FFT é viável para uma parcela grande (com exceção de apenas 4 modelos com API versão 2 e apenas um modelo com API versão 4).

Em relação à diferença entre níveis da API é possível observar que, de modo geral, existem mais aparelhos com CPUs mais rápidas e com múltiplos cores que rodam a versão 4.x do Android. Assim, o tempo menor de processamento nestes aparelhos em relação àqueles que rodam a versão 2.x é esperado.

5.3.3 Estimação de parâmetros máximos

Supondo que a média dos tempos de callback obtidos realmente deixe espaço para mais computação, então a próxima pergunta natural é: quanta computação (concorrente) pode ser de fato

executada dentro do callback, mantendo-se a geração de amostras em tempo real? Para responder a esta questão, os dispositivos foram levados ao limite através da execução do algoritmo de convolução e de quatro variantes da síntese aditiva. Ao comparar o tempo de callback para diferentes tamanhos de bloco (no caso da convolução) e para diferentes números de osciladores (no caso da síntese aditiva) com o período teórico do ciclo DSP, foi possível estimar o tamanho máximo de uma convolução e a quantidade máxima de osciladores que podem ser computados em tempo real.

A medição foi realizada para diferentes tamanhos de bloco (de 2^4 até 2^{13} amostras). Espera-se que o parâmetro máximo seja mais ou menos o mesmo para diferentes tamanhos de bloco, pois apesar de blocos maiores corresponderem a períodos maiores entre callbacks, eles também possuem mais amostras para serem processadas, e assim estes dois fatores se cancelam mutuamente.

Ao longo do experimento, pôde-se observar que, eventualmente, valores pequenos de tamanho de bloco resultavam em valores máximos de parâmetros inconsistentes. Isto ocorreu pois, para blocos pequenos, o período teórico do ciclo DSP também é pequeno, e algumas vezes o sistema operacional, ao executar tarefas administrativas concorrentemente (como recolhimento de lixo), ocupa parte significativa da CPU influenciando no resultado do teste. Este comportamento é minimizado para blocos com maior número de amostras. Para evitar este problema, foi calculada uma primeira média dos valores máximos dos parâmetros para diferentes tamanhos de bloco, e os valores que desviavam de mais do que 2 desvios-padrão foram descartados. Assim, uma segunda média foi calculada somente com os valores considerados consistentes.

Os resultados da estimação do tamanho máximo de uma convolução viável em tempo real para os diferentes dispositivos pode ser visto na Figura 5.9. A figura mostra a média dos tamanhos máximos de convolução que cada dispositivo consegue processar em tempo real obtida ao realizar a medição para diferentes tamanhos de bloco.

A exemplo do que foi feito na GPU e descrito na Seção 4.3, aqui também foram implementadas diferentes formas de cálculo de osciladores, utilizando: (1) a função `sin()` da API, (2) consulta a tabela com índice truncado, (3) consulta a tabela com interpolação linear, e (4) consulta a tabela com interpolação cúbica. O método de estimação do número máximo de osciladores utilizado foi o mesmo descrito acima para a convolução, e os resultados podem ser vistos na Figura 5.10. Em geral, os resultados são os esperados: pode-se calcular mais osciladores em tempo real utilizando consulta truncada do que com interpolação linear, e mais osciladores utilizando consulta com interpolação linear do que com interpolação cúbica. Em geral também, a utilização da função seno da API Java permite o cálculo de um número de osciladores que fica entre os parâmetros obtidos com a consulta com interpolação cúbica e com interpolação linear. Em alguns dispositivos, o resultado da consulta truncada se inverte com o da consulta com interpolação linear, e pode-se supor que este comportamento se deva exatamente à natureza não determinística do escalonamento de processos no sistema operacional, aliada à quantidade de operações que o sistema operacional executa concorrentemente ao aplicativo de testes e à prioridade do aplicativo no escalonamento, que não difere dos outros aplicativos executados no nível de usuário sem permissões especiais.

5.3.4 Discussão

Foi desenvolvido um modelo de processamento de áudio em tempo real em Java para dispositivos Android que permite executar testes automatizados em qualquer dispositivo que rode a API 7 ou mais nova. Com esta aplicação, foi possível obter estatísticas de tempo de algoritmos de processamento de áudio usuais, além de estressar os dispositivos para medir a viabilidade de certas tarefas sob as condições de tempo real. Com isto, foi possível obter limites superiores para a computação usando este modelo DSP, mas os resultados poderiam ser melhorados uma vez que este modelo não utiliza privilégios especiais do sistema, nenhum código nativo (em C/C++) e poucas otimizações de código.

Ao longo deste trabalho, foi possível encontrar pouca documentação sobre processamento de áudio em tempo real utilizando a API do Android no nível da aplicação, então espero que este trabalho possa suprir parte desta demanda ao distribuir o código fonte (sob licença livre) e a documentação do modelo (ambos podem ser obtidos em http://www.ime.usp.br/~ajb/android). Além

disso, este trabalho foi desenvolvido com a expectativa de que seja útil para outros pesquisadores e artistas que queiram obter informações importantes que possam ajudar na escolha de hardware para ser utilizado em apresentações em tempo real. Ao prover uma forma sistemática de obter medidas de desempenho relacionadas a tarefas usuais de processamento de áudio e limites superiores em filtros FIR (que podem ser entendidos como um modelo DSP geral em termos de complexidade computacional), espero ter atingido este objetivo.

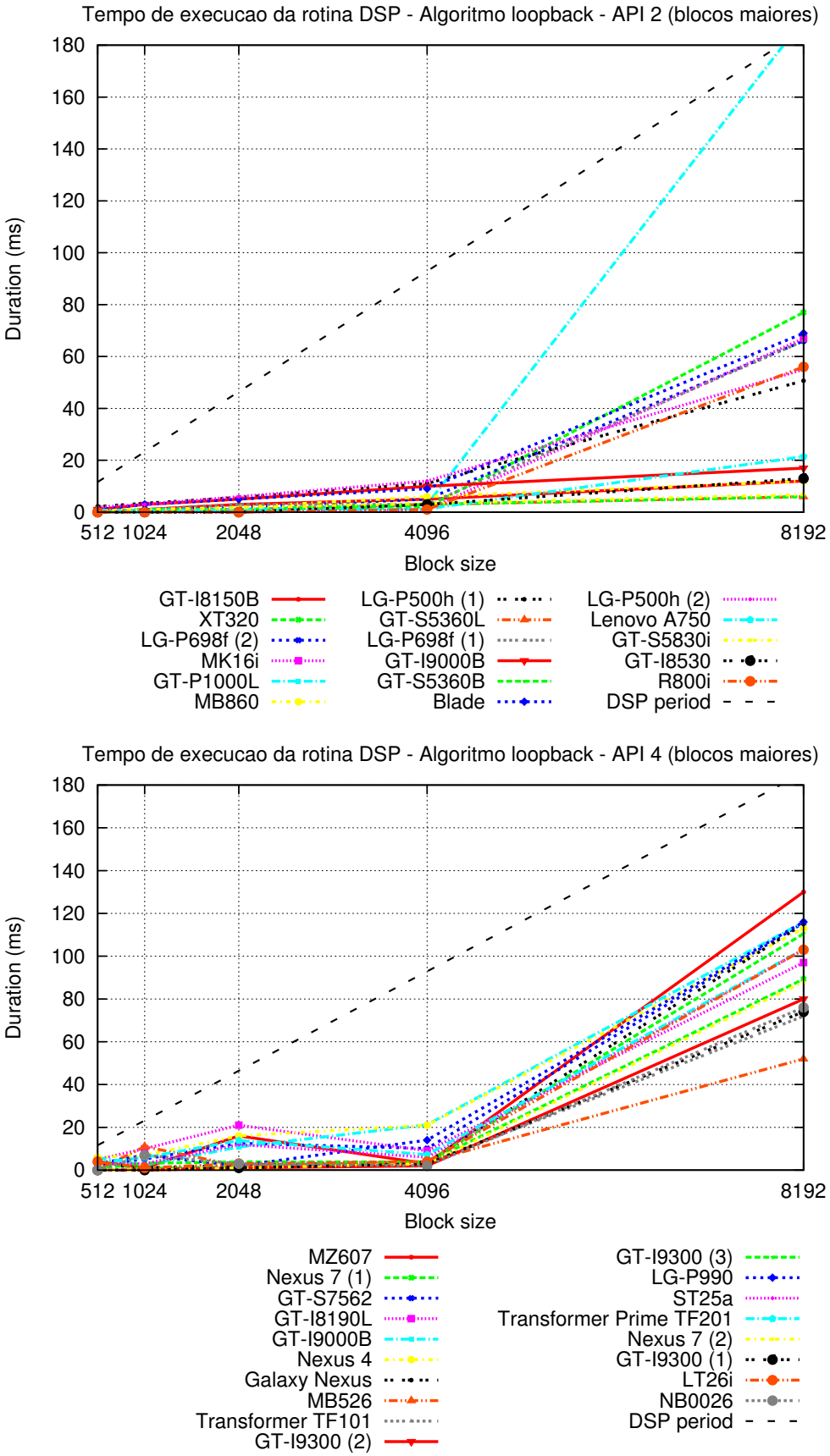

Figura 5.6: *Comparação entre tempo de execução do algoritmo de "loopback" para blocos maiores (512 a 2048 amostras) em dispositivos com API versão 2.X (figura de cima) e 4.X (figura de baixo).*

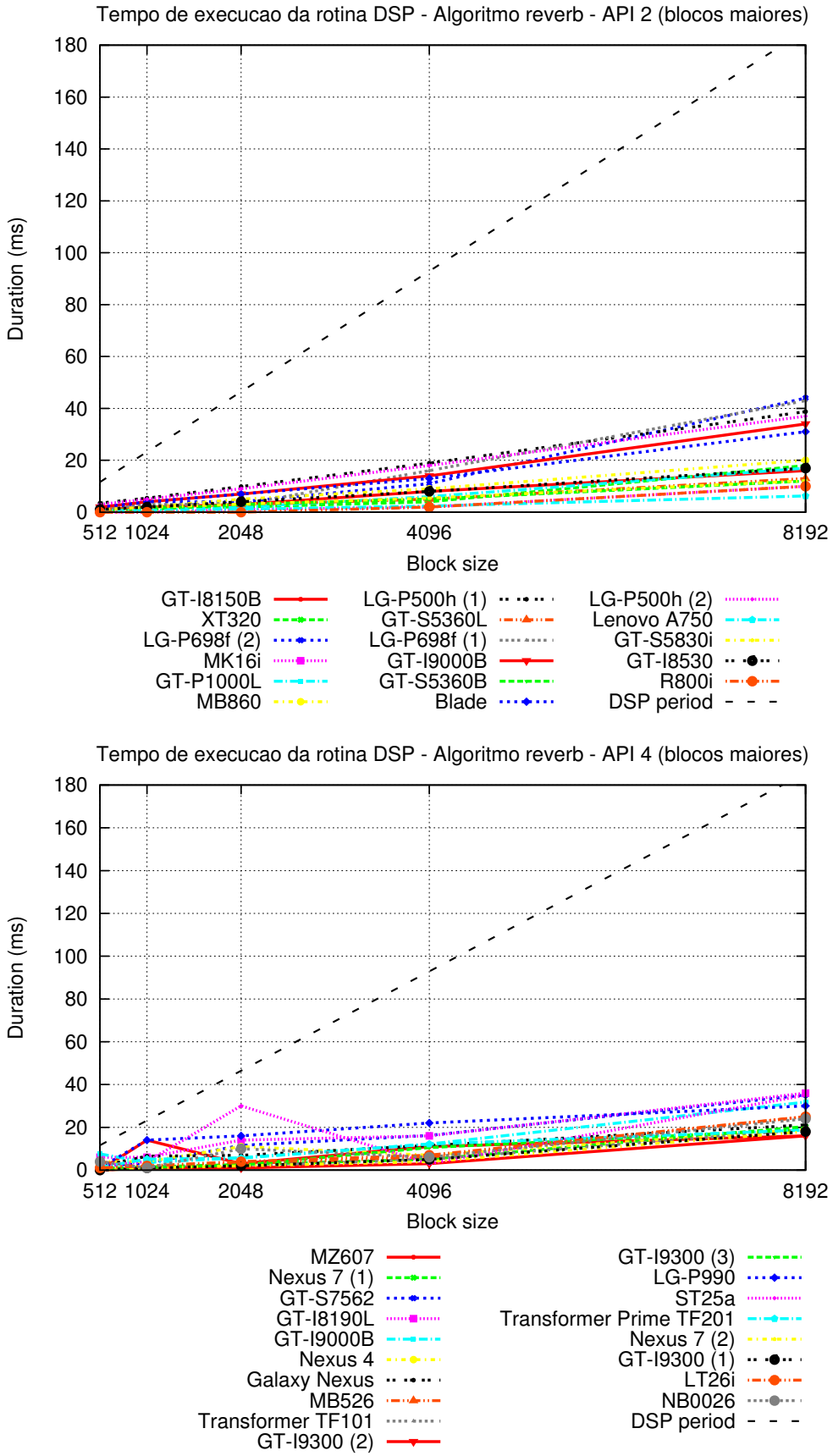

Figura 5.7: *Comparação entre tempo de execução do algoritmo de reverberação para blocos maiores (512 a 2048 amostras) em dispositivos com API versão 2.X (figura de cima) e 4.X (figura de baixo).*

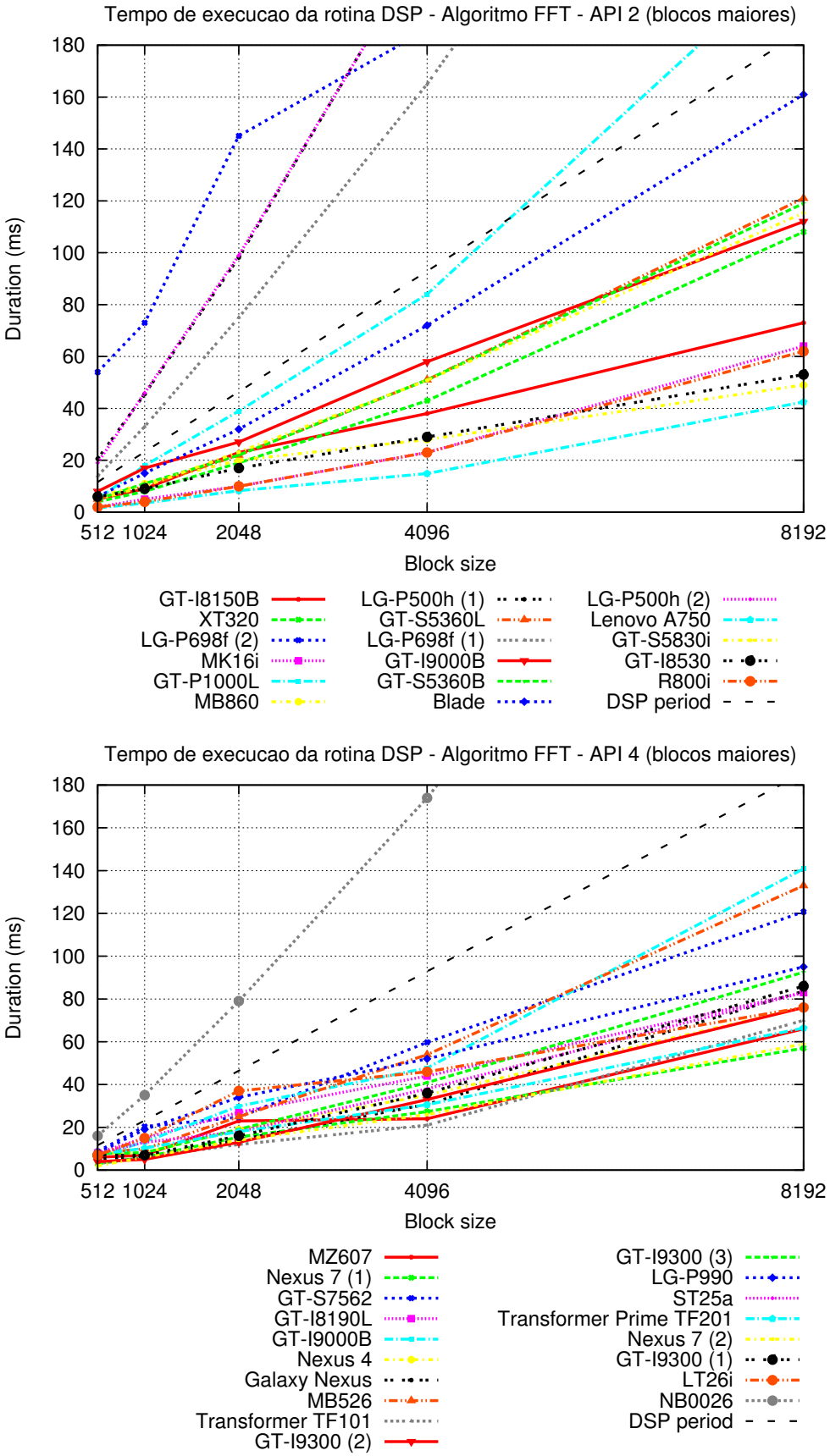

Figura 5.8: *Comparação entre tempo de execução da FFT para blocos maiores (512 a 2048 amostras) em dispositivos com API versão 2.X (figura de cima) e 4.X (figura de baixo)*

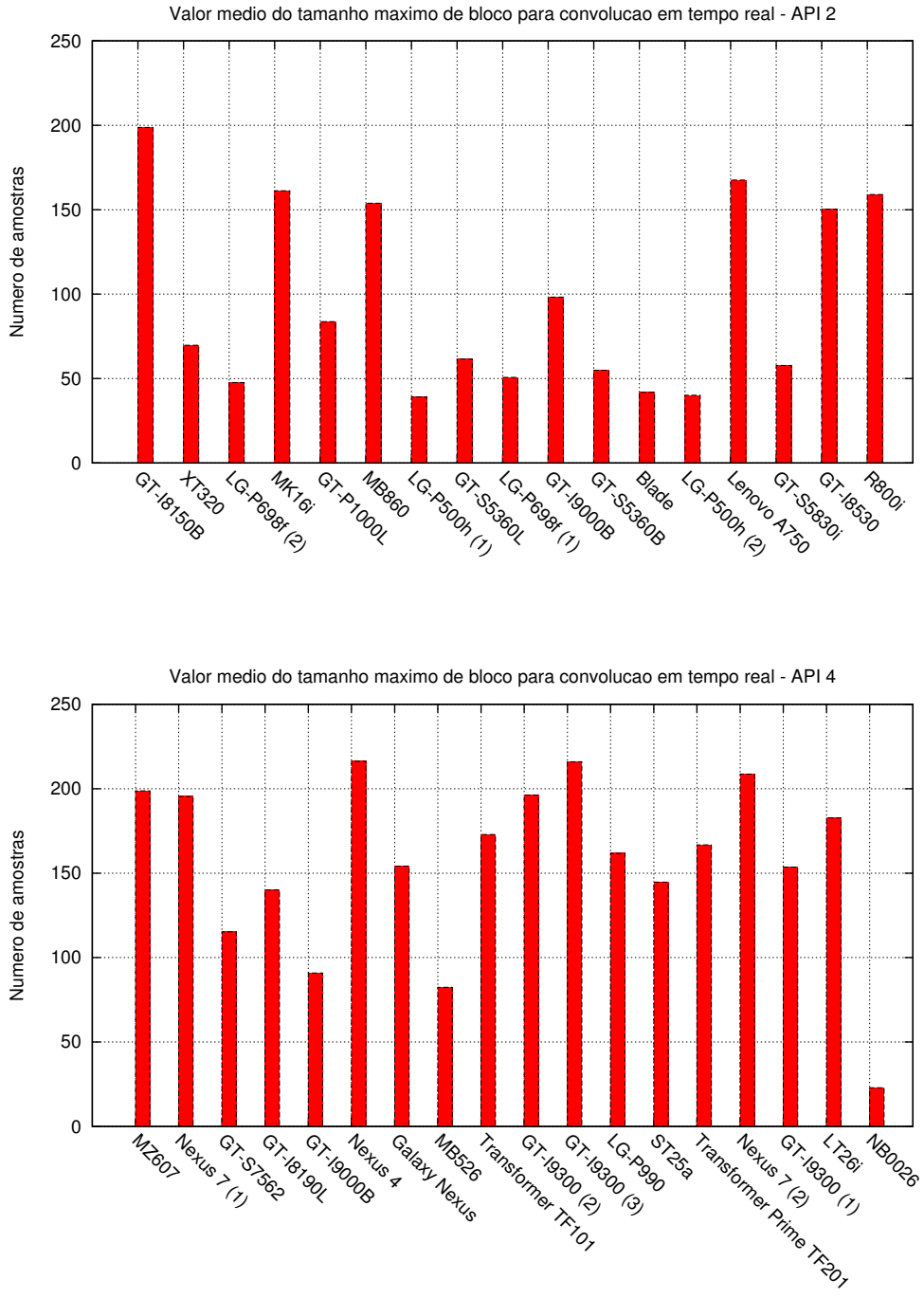

Figura 5.9: *Comparação entre valores médios do tamanho máximo de bloco para convolução em diferentes dispositivos.*

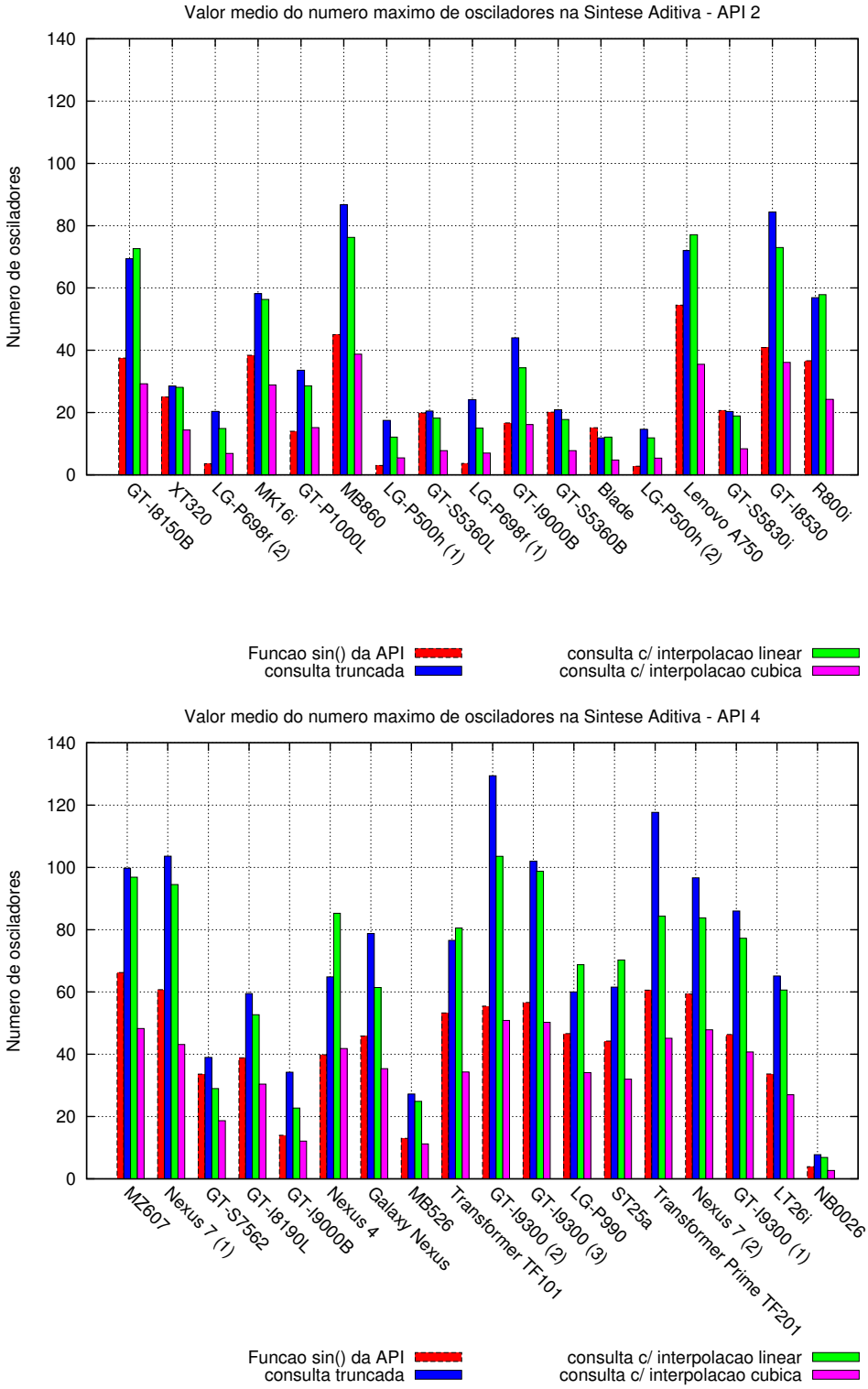

Figura 5.10: *Comparação entre valores médios do número máximo de osciladores na síntese aditiva para diferentes implementações e diferentes dispositivos.*

Capítulo 6

Conclusão

Na Seção 1.1 foram expostos os seguintes objetivos principais para este trabalho: explorar técnicas de processamento de áudio em tempo real e utilizar tecnologia barata e acessível para tal. Em uma sociedade na qual o cidadão e o consumidor são duas caras da mesma moeda, a utilização de tecnologia de baixo custo e a resistência à tendência de obsolescência precoce dos dispositivos computacionais têm, cada vez mais, importância fundamental na educação, na arte e na ciência. A reutilização e reaproveitamento da tecnologia imediatamente disponível pode significar uma economia de recursos (em termos de dinheiro, equipamento, meio ambiente, etc) que nem sempre é facilmente percebida por causa do desconhecimento que o cidadão-consumidor tem sobre os meios de produção e suas possibilidades de escolha.

As implementações realizadas em Arduino, expostas no Capítulo 3, mostram que com um pouco de boa vontade e algumas restrições no domínio de aplicação é possível realizar processamento de áudio em tempo real até mesmo em microcontroladores com poder computacional bastante baixo. Aplicações em locais remotos que disponham de fontes de energia intermitentes (como por exempo solar ou eólica) e utilizem bateria para armazenamento podem se beneficiar do baixo consumo de energia do Arduino.

Atualmente, as placas do tipo GPU têm presença significativa mesmo em dispositivos móveis, que em geral possuem na bateria uma de suas maiores limitações. Ao observar o aumento do número de computadores de mesa, notebooks, tablets e telefones celulares que vêm de fábrica com processadores gráficos programáveis, é possível prever que em pouco tempo as GPUs estarão presentes mesmo nos aparelhos de mais baixo custo. As possibilidades de integração da GPU com software livre e sua utilização para tarefas rotineiras de processamento de áudio demonstradas no Capítulo 4 podem aumentar a utilização deste tipo de circuito para fins artísticos e científicos.

O sistema operacional Android, por sua vez, figura como uma tendência crescente e possui, no momento da escrita deste texto, mais de 60% do mercado global de *smartphones*. Este é um motivo para abordar a plataforma mas não para se restringir a ela. Alternativas existem para desenvolvimento de aplicativos que podem ser compilados para diferentes plataformas, algumas inclusive baseadas em tecnologias bem estabelecidas como HTML5 e Javascript, que podem gerar código para Android, iPhone e Blackberry, entre outros. Ao utilizar estes aparelhos para fins como o processamento de áudio em tempo real, como feito no Capítulo 5, pode-se diminuir a necessidade de fabricação e obtenção de novos dispositivos para aplicações específicas.

Vê-se, portanto, que a implementação de técnicas de processamento de áudio em tempo real em dispositivos "não convencionais" é, não só possível, como bastante frutífera em termos de pesquisa científica e de provimento de alternativas para utilização imediata da tecnologia que já está disponível. Ao contrário de identificar novas tecnologias para cenários de aplicação, este trabalho propôs e, espero, mostrou que muitas vezes a tecnologia necessária já está disponível apesar de não estar devidamente evidenciada.

6.1 Artigos publicados

A pesquisa apresentada neste relatório gerou alguns artigos publicados em um evento nacional e três conferências internacionais, relacionados abaixo.

III Workshop em Música Ubíqua (UbiMus)

De 4 a 6 de maio de 2012, ocorreu no IME/USP o III Workshop de Música Ubíqua, no qual pesquisadores interessados em aplicações sonoras e musicais de tecnologia da informação se reuniram para compartilhar propostas e resultados de projetos de pesquisa nesta área. Naquele evento foi apresentado um artigo com o conteúdo relativo à Seção 5.2 deste texto, contendo os resultados de uma pesquisa preliminar para implementação do aplicativo de processamento em tempo real em sistemas Android (Bianchi, 2012).

Sound and Music Computing (SMC) 2012

Os resultados de uma primeira rodada de testes com o aplicativo desenvolvido para Android foram apresentados e publicados nos anais do evento Sound and Music Computing Conference 2012, que ocorreu de 11 a 14 de Julho de 2012 em Copenhagen, Dinamarca (Bianchi e Queiroz , 2012b). Naquela etapa, apenas 11 aparelhos haviam sido avaliados e os resultados obtidos ainda não contavam com as implementações de convolução e síntese aditiva. Além disso, o código do aplicativo e do sistema de testes foi consideravelmente melhorado e documentado desde então.

International Computer Music Conference (ICMC) 2012

As medições de tempo de transferência de memória e tempo de execução dos diversos algoritmos de processamento de áudio na GPU foram apresentados e publicados nos anais do evento International Computer Music Conference 2012, que ocorreu de 9 a 14 de setembro de 2012 em Ljubljana, Eslovênia (Bianchi e Queiroz, 2012a). Naquele momento, o trabalho ainda não contava com a implementação da convolução no domínio do tempo e contava somente com dois modelos de placa de vídeo.

Sound and Music Computing (SMC) 2013

As implementações e resultados obtidos com as implementações de convolução, síntese aditiva e FFT no Arduino foram apresentados e publicadas nos anais da edição de 2013 da Sound and Music Computing Conference, realizada de 30 de julho a 3 de agosto de 2013 em Estocolmo, Suécia (Bianchi e Queiroz, 2013).

6.2 Trabalhos futuros

Existem diversas possibilidades de trabalhos futuros nas plataformas abordadas. Abaixo estão listadas algumas possibilidades que já estão em andamento ou que surgiram ao longo da pesquisa.

Algumas possibilidades de extensão imediata do trabalho realizado no Arduino foram levantadas. A primeira seria o uso da conversão ADC de 10 bits, adaptação dos testes para operações sobre 2 bytes e comparação do desempenho com a implementação atual. Pode-se esperar um custo computacional bastante maior por causa da natureza de 8 bits do processador. Uma outra posibilidade é a determinação do ruído introduzido no sinal pelo processo de conversão ADC e síntese utilizando PWM. Além disso, a utilização de outros modelos de Arduino pode trazer à luz diferenças significativas entre poder computacional e possibilidades de aplicação.

Em relação à GPU, diversas possibilidades surgiram:

- Um do trabalhos atuais no Grupo de Computação Musical do IME/USP lida com a distribuição de áudio em redes de computadores. É interessante analisar a possibilidade de

terceirizar o processamento em tempo real através da rede para máquinas remotas que possuam placas GPU, e verificar se existe vantagem na aceleração da computação dados diversos cenários de latência na rede.

- Uma técnica comum para aumentar o número de frequências observadas numa Transformada de Fourier e ao mesmo tempo manter uma resolução razoável no domínio do tempo é permitir a sobreposição de blocos de amostras. A implementação desta técnica pode trazer melhores resultados em termos de qualidade de áudio ao diminuir o tempo de computação das funções de *kernel* e realizar um maior número de funções num mesmo intervalo de tempo.

- É possível utilizar as possibilidades de execução assíncrona da GPU para desacoplar a transferência de memória e a execução da função de kernel do ciclo DSP do Pd. Seria possível, por exemplo, utilizar um modelo de produtor/consumidor para alimentar buffers intermediários que poderiam ser reproduzidos independentemente do controle da computação. Esta técnica também poderia aumentar o desempenho do cálculo na GPU.

- Como foi comentado na Seção 4.1.3, a avaliação do desempenho do Pd com a GPU pode ser de extremo valor para programadores de externals de Pd e também para usuários finais uma vez que o suporte à GPU esteja embutido nas funções básicas do Pd. A transferência da infraestrutura de medição desenvolvida aqui para o código do Pd seria uma forma de prover esta funcionalidade.

Parcerias têm se desenvolvido em torno do aplicativo Java desenvolvido para Android. A última rodada de testes, lançada em Julho de 2013, incluiu contribuições de colegas que ajudaram com melhorias no código e incluíram testes envolvendo outros algoritmos como as Transformadas Discretas do Coseno, do Seno e de Hartley, usando bibliotecas conhecidas como JTransforms e FFTW. Um trabalho em andamento trata da inclusão de algoritmos que exploram paralelismo, de forma que seja possível medir a aceleração em dispositivos com múltiplas unidades de processamento. Trabalhos futuros em Android incluem a exploração da GPU, cada vez mais presente nos dispositivos móveis, e a utilização da bilioteca libpd[1] para interface com Pure Data.

Além destas, existem muitas outras possibilidades de extensão do trabalho aqui apresentado, não restritas às três plataformas abordadas, sendo que duas direções principais merecem ser destacadas. A primeira direção é a da utilização do que geralmente é considerado lixo computacional para a fabricação de novos dispositivos para aplicações científicas, artísticas e educacionais. A abundância de material de difícil descarte, que muitas vezes possui em sua composição metais pesados e mesmo assim não conta com política de reciclagem consistente em muitos países, pode por outro lado significar abundância de material para construção de novas máquinas para resolução de problemas específicos.

Uma outra direção interessante é a de aumentar o domínio de aplicação das tecnologias aqui estudadas, de forma a abarcar mais do que simplesmente processamento de áudio em tempo real. Como foi dito anteriormente no texto, os dispositivos aqui utilizados podem executar qualquer algoritmo e só dependem das limitações de memória (do dispositivo) e tempo (do usuário). Nesse sentido, aplicações destas tecnologias e da metodologia aqui apresentada permitiriam a exploração daquilo que seria considerado lixo computacional em outros âmbitos artísticos (teatro, dança, artes visuais), na educação (no ensino de música ou de computação, entre muitas outras disciplinas), em jogos, etc.

Assim, concluo este texto com a esperança de ter contribuído para desfazer alguns preconceitos comuns sobre a possibilidade de uso de plataformas não convencionais para a realização de processamento de áudio em tempo real. Tenham elas pouca capacidade computacional, latência na transferência de memória ou possibilidades restritas de modificação, os fatos de estarem altamente disponíveis e possuírem custo relativamente baixo são motivações para que sejam consideradas com seriedade para as mais distintas aplicações.

[1] http://puredata.info/downloads/libpd/

Referências Bibliográficas

ADBlackfin () ADBlackfin. Analog Devices - Processadores Blackfin. http://www.analog.com/en/processors-dsp/blackfin/products/index.html, 2013. [Online; accessed 22-Aug-2013]. Citado na pág.

ATmega328P datasheet () ATmega328P datasheet. Atmel ATmega48A/48PA/88A/88PA/168A/328/328P datasheet. http://www.atmel.com/Images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet.pdf, 2013. [Online; accessed 07-Apr-2013]. Citado na pág.

Bianchi (2012) André J. Bianchi. Processamento de áudio em tempo real em sistemas Android. Citado na pág.

Bianchi e Queiroz (2012a) André J. Bianchi e Marcelo Queiroz. Measuring the performance of realtime dsp using pure data and gpu. *Proceedings of the International Computer Music Conference 2012*, páginas 124–127. Citado na pág.

Bianchi e Queiroz (2012b) André J. Bianchi e Marcelo Queiroz. On the performance of realtime dsp on android devices. *Proceedings of the 9th Sound and Music Computing Conference*, páginas 113–120. Citado na pág.

Bianchi e Queiroz (2013) André J. Bianchi e Marcelo Queiroz. Real time digital audio processing using arduino. *Proceedings of the Sound and Music Computing Conference 2013*, páginas 538–545. Citado na pág.

Brinkmann (2012) Peter Brinkmann. *Making Musical Apps*. O'Reilly Media. Citado na pág.

Broughton e Bryan (2011) S.A. Broughton e K.M. Bryan. *Discrete Fourier Analysis and Wavelets: Applications to Signal and Image Processing*. Wiley. ISBN 9781118030707. URL http://books.google.com.br/books?id=ViG_gc8F-RAC. Citado na pág.

Buck *et al.* **(2004)** Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston e Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23:777–786. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1015706.1015800. URL http://doi.acm.org/10.1145/1015706.1015800. Citado na pág.

Burrus (1972) C. Burrus. Block realization of digital filters. *Audio and Electroacoustics, IEEE Transactions on*, 20(4):230 – 235. ISSN 0018-9278. doi: 10.1109/TAU.1972.1162387. Citado na pág.

Cebenoyan (2004) Cem Cebenoyan. *Graphics Pipeline Performance*, páginas 473–486. Addison-Wesley Professional. URL http://download.nvidia.com/developer/GPU_Gems/Sample_Chapters/Graphics_Pipeline_Performance.pdf. Citado na pág.

CMJ () CMJ. Computer Music Journal. http://www.computermusicjournal.org/, 2013. [Online; accessed 22-Aug-2013]. Citado na pág.

Cooley (1987) J. W. Cooley. How the fft gained acceptance. Em *Proceedings of the ACM conference on History of scientific and numeric computation*, HSNC '87, páginas 97–100, New

York, NY, USA. ACM. ISBN 0-89791-229-2. doi: http://doi.acm.org/10.1145/41579.41589. URL http://doi.acm.org/10.1145/41579.41589. Citado na pág.

Cooley e Tukey (1965) James Cooley e John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301. Citado na pág.

Cucchiara e Gualdi (2010) Rita Cucchiara e Giovanni Gualdi. *Mobile Video Surveillance Systems: An Architectural Overview*, volume 5960, páginas 89–109. Springer Berlin / Heidelberg. Citado na pág.

Cui-xiang *et al.* (2005) Zhong Cui-xiang, Han Guo-qiang e Huang Ming-He. Some new parallel Fast Fourier Transform algorithms. Em *Parallel and Distributed Computing, Applications and Technologies, 2005. PDCAT 2005. Sixth International Conference on*, páginas 624–628. doi: 10.1109/PDCAT.2005.224. Citado na pág.

Deepak *et al.* (2007) G. Deepak, P.K. Meher e A. Sluzek. Performance characteristics of parallel and pipelined implementation of fir filters in fpga platform. Em *Signals, Circuits and Systems, 2007. ISSCS 2007. International Symposium on*, volume 1, páginas 1 –4. doi: 10.1109/ISSCS.2007.4292697. Citado na pág.

Dimitrov e Serafin (2011a) Smilen Dimitrov e Stefania Serafin. *An Analog I/O Interface Board for Audio Arduino Open Sound Card System*, páginas 290–297. Padova University Press. Citado na pág.

Dimitrov e Serafin (2011b) Smilen Dimitrov e Stefania Serafin. *Audio Arduino - an ALSA (Advanced Linux Sound Architecture) audio driver for FTDI-based Arduinos*, páginas 211–216. University of Oslo and Norwegian Academy of Music. ISBN ISSN 2220-4792. Citado na pág.

Dirichlet (1829) Peter G. Dirichlet. Sur la convergence des séries trigonométriques qui servent à représenter une fonction arbitraire entre des limites données. *Journal für die reine und angewandte Mathematik*, 4:157–169. URL http://arxiv.org/abs/0806.1294. Citado na pág.

Dolson (1986) Mark Dolson. The phase vocoder: A tutorial. *Computer Music Journal*, 10(4):14–27. Citado na pág.

Eyre e Bier (2000) J. Eyre e J. Bier. The evolution of dsp processors. *Signal Processing Magazine, IEEE*, 17(2):43 –51. ISSN 1053-5888. doi: 10.1109/79.826411. Citado na pág.

Flores *et al.* (2010) Luciano Flores, Ro Miletto, Marcelo Pimenta, Eduardo Mir e Damián Keller. Musical interaction patterns: Communicating computer music knowledge in a multidisciplinary project, 2010. Citado na pág.

Flynn (1966) M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901 – 1909. ISSN 0018-9219. doi: 10.1109/PROC.1966.5273. Citado na pág.

Fourier (1807) J. Fourier. Mémoire sur la propagation de la chaleur dans le corps solides. *Noveau bulletin des sciences par la société philomatique de paris*, 6(10):215. Citado na pág.

Fournier e Fussell (1988) Alain Fournier e Donald Fussell. On the power of the frame buffer. *ACM Trans. Graph.*, 7:103–128. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/42458.42460. URL http://doi.acm.org/10.1145/42458.42460. Citado na pág.

Gallo e Tsingos (2004) Emmanuel Gallo e Nicolas Tsingos. Efficient 3d audio processing on the gpu. Em *Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors*. ACM. URL http://www-sop.inria.fr/reves/Basilic/2004/GT04. Citado na pág.

Geiger (2003) G Geiger. Pda: Real time signal processing and sound generation on handheld devices. Em *Proceedings of the International Computer Music Conference ICMC*. San Francisco: ICMA. URL http://quod.lib.umich.edu/cgi/p/pod/dod-idx?c=icmc;idno=bbp2372.2003.054. Citado na pág.

Gibb (2010) A. M. Gibb. *NEW MEDIA ART, DESIGN, AND THE ARDUINO MICROCONTROLLER: A MALLEABLE TOOL*. Tese de Doutorado, Pratt Institute. Citado na pág.

Govindaraju *et al.* (2005) Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin e Dinesh Manocha. Fast computation of database operations using graphics processors. Em *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA. ACM. doi: http://doi.acm.org/10.1145/1198555.1198787. URL http://doi.acm.org/10.1145/1198555.1198787. Citado na pág.

Gray (2003) A.A. Gray. Parallel sub-convolution filter bank architectures. Em *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, volume 4, páginas IV–528 – IV–531 vol.4. doi: 10.1109/ISCAS.2003.1205971. Citado na pág.

Guha *et al.* (2003) Sudipto Guha, Shankar Krishnan, Kamesh Munagala e Suresh Venkatasubramanian. Application of the two-sided depth test to csg rendering. Em *Proceedings of the 2003 symposium on Interactive 3D graphics*, I3D '03, páginas 177–180, New York, NY, USA. ACM. ISBN 1-58113-645-5. doi: http://doi.acm.org/10.1145/641480.641513. URL http://doi.acm.org/10.1145/641480.641513. Citado na pág.

Hall e Anderson (2009) Sharon P. Hall e Eric Anderson. Operating systems for mobile computing. *J. Comput. Small Coll.*, 25:64–71. ISSN 1937-4771. URL http://portal.acm.org/citation.cfm?id=1629036.1629046. Citado na pág.

He *et al.* (2007) Bingsheng He, Naga K. Govindaraju, Qiong Luo e Burton Smith. Efficient gather and scatter operations on graphics processors. Em *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, páginas 1 –12. doi: 10.1145/1362622.1362684. Citado na pág.

Henry (2011) Charles Z. Henry. Pdcuda: an implementation with the cuda runtime api. Em *PDCON*. Citado na pág.

Holland (1959) John Holland. A universal computer capable of executing an arbitrary number of sub-programs simultaneously. Em *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '59 (Eastern), páginas 108–113, New York, NY, USA. ACM. doi: http://doi.acm.org/10.1145/1460299.1460311. URL http://doi.acm.org/10.1145/1460299.1460311. Citado na pág.

ICMA () ICMA. International Computer Music Association. http://www.computermusic.org/, 2013. [Online; accessed 22-Aug-2013]. Citado na pág.

Kapasi *et al.* (2003) U.J. Kapasi, S. Rixner, W.J. Dally, B. Khailany, Jung Ho Ahn, P. Mattson e J.D. Owens. Programmable stream processors. *Computer*, 36(8):54–62. Citado na pág.

Kapasi *et al.* (2002) Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens e Brucek Khailany. The Imagine stream processor. Em *Proceedings 2002 IEEE International Conference on Computer Design*, páginas 282–288. Citado na pág.

Lewis (2000) George E. Lewis. Too Many Notes: Computers, Complexity and Culture in Voyager. *Leonardo Music Journal*, 10:33–39. URL http://www.mitpressjournals.org/doi/abs/10.1162/096112100570585. Citado na pág.

Lin *et al.* **(2011)** Cheng-Min Lin, Jyh-Horng Lin, Chyi-Ren Dow e Chang-Ming Wen. Benchmark dalvik and native code for android system. Em *Innovations in Bio-inspired Computing and Applications (IBICA), 2011 Second International Conference on*, páginas 320 –323. doi: 10.1109/IBICA.2011.85. Citado na pág.

LMJ () LMJ. Leonardo Music Journal. http://www.leonardo.info/lmj/about.html, 2013. [Online; accessed 22-Aug-2013]. Citado na pág.

Merz (2009) H. Merz. Cufft vs fftw comparison. Relatório técnico, University of Waterloo. Citado na pág.

Mohanty (2009) S.P. Mohanty. Gpu-cpu multi-core for real-time signal processing. Em *Consumer Electronics, 2009. ICCE '09. Digest of Technical Papers International Conference on*, páginas 1 –2. doi: 10.1109/ICCE.2009.5012160. Citado na pág.

Moore (1990) F. Richard Moore. *Elements of computer music*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-252552-6. Citado na pág.

Moreland e Angel (2003) Kenneth Moreland e Edward Angel. The fft on a gpu. Em *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, páginas 112–119, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. ISBN 1-58113-739-7. URL http://portal.acm.org/citation.cfm?id=844174.844191. Citado na pág.

Nawrath () M. Nawrath. [Online; accessed 07-Apr-2013]. Citado na pág.

Oppenheim *et al.* **(1999)** Alan V. Oppenheim, Ronald W. Schafer e John R. Buck. *Discrete-time signal processing (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-754920-2. Citado na pág.

OS () OS. Organised Sound. http://journals.cambridge.org/action/displayJournal?jid=OSO, 2013. [Online; accessed 22-Aug-2013]. Citado na pág.

Owens *et al.* **(2008)** J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone e J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879 –899. ISSN 0018-9219. doi: 10.1109/JPROC.2008.917757. Citado na pág.

Owens (2005) John Owens. Streaming architectures and technology trends. Em *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA. ACM. doi: http://doi.acm.org/10.1145/1198555.1198766. URL http://doi.acm.org/10.1145/1198555.1198766. Citado na pág.

Owens *et al.* **(2007)** John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn e Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113. ISSN 1467-8659. doi: 10.1111/j.1467-8659.2007.01012.x. URL http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x. Citado na pág.

Pathak (2011) Kailash Pathak. Efficient audio processing in android 2.3. *Journal of Global Research in Computer Science*, 2(7):79–82. Citado na pág.

Patrick *et al.* **(2003)** William Dally Patrick, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, François Labonté, Jung ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju e Ian Buck. Merrimac: Supercomputing with streams, 2003. Citado na pág.

Pease (1968) Marshall C. Pease. An adaptation of the fast fourier transform for parallel processing. *J. ACM*, 15:252–264. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321450.321457. URL http://doi.acm.org/10.1145/321450.321457. Citado na pág.

Podlozhnyuk (2007) Victor Podlozhnyuk. Image convolution with cuda. Relatório técnico, NVidia Corporation. Citado na pág.

Press *et al.* **(1992)** William H. Press, Saul A. Teukolsky, William T. Vetterling e Brian P. Flannery. Numerical recipes in C: The art of scientific computing. second edition, 1992. Citado na pág.

Puckette (1996) Miller Puckette. Pure data: another integrated computer music environment. Em *in Proceedings, International Computer Music Conference*, páginas 37–41. Citado na pág.

Radhakrishnan (2007) Arjun Radhakrishnan. *Signal processing on a graphics card - An analysis of performance and accuracy.* Tese de Doutorado, University of Cape Town. Citado na pág.

Redmill e Bull (1997) D.W. Redmill e D.R. Bull. Design of low complexity fir filters using genetic algorithms and directed graphs. Em *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1997. GALESIA 97. Second International Conference On (Conf. Publ. No. 446)*, páginas 168 –173. doi: 10.1049/cp:19971175. Citado na pág.

Renfors e Neuvo (1981) M. Renfors e Y. Neuvo. The maximum sampling rate of digital filters under hardware speed constraints. *Circuits and Systems, IEEE Transactions on*, 28(3):196 – 202. ISSN 0098-4094. doi: 10.1109/TCS.1981.1084972. Citado na pág.

Rowe (1992a) R Rowe. Machine listening and composing with cypher. *Computer Music Journal*, 16(1). Citado na pág.

Rowe (1992b) R Rowe. Machine listening and composing with cypher. *Computer Music Journal*, 16(1). Citado na pág.

Savioja *et al.* **(2011)** Lauri Savioja, Vesa Välimäki e Julius O. Smith. Audio signal processing using graphics processing units. *J. Audio Eng. Soc*, 59(1/2):3–19. Citado na pág.

SBCM () SBCM. Simpósio Brasileiro de Computação Musical. http://compmus.ime.usp.br/sbcm/, 2013. [Online; accessed 22-Aug-2013]. Citado na pág.

Sedgewick e Schidlowsky (1998) Robert Sedgewick e Michael Schidlowsky. *Algorithms in Java, Third Edition, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd ed. ISBN 0201361205. Citado na pág.

Sernec *et al.* **(2000)** R. Sernec, M. Zajc e J. Tasic. The evolution of dsp architectures: towards parallelism exploitation. Em *Electrotechnical Conference, 2000. MELECON 2000. 10th Mediterranean*, volume 2, páginas 782 – 785 vol.2. doi: 10.1109/MELCON.2000.880050. Citado na pág.

Shroeder *et al.* **(2007)** F. Shroeder, Alain Renaud, P. Rebelo e F. Gualda. Addressing the network: Perfomative strategies for playing apart. Em *Proceedings of International Computer Music Conference 2007*. International Computer Music Association. URL http://eprints.bournemouth.ac.uk/9450/. Citado na pág.

SMC () SMC. http://smcnetwork.org/. , 2013. [Online; accessed 22-Aug-2013]. Citado na pág.

Thompson *et al.* **(2002)** Chris J. Thompson, Sahngyun Hahn e Mark Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. Em *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, páginas 306–317, Los Alamitos, CA, USA. IEEE Computer Society Press. ISBN 0-7695-1859-1. URL http://portal.acm.org/citation.cfm?id=774861.774894. Citado na pág.

Tsingos *et al.* **(2011)** Nicolas Tsingos, Wenyu Jiang e Ian Williams. Using programmable graphics hardware for acoustics and audio rendering. *J. Audio Eng. Soc*, 59(9):628–646. Citado na pág.

Venkatasubramanian (2003) Suresh Venkatasubramanian. The graphics card as a streaming computer. *CoRR*, cs.GR/0310002. Citado na pág.

Vercoe e Ellis (1990) B. L. Vercoe e D. P. Ellis. Real-time csound: software synthesis with sensing and control. Em *International Computer Music Conference Glasgow 1990 Proceedings*, páginas 209–211. Citado na pág.

Wikipedia () Wikipedia. Android historical version distribution. http://www.atmel.com/Images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet.pdf, 2013. [Online; accessed 07-Apr-2013]. Citado na pág.

Wong *et al.* (2007) Tien-Tsin Wong, Chi-Sing Leung, Pheng-Ann Heng e Jianqing Wang. Discrete wavelet transform on consumer-level graphics hardware. *Multimedia, IEEE Transactions on*, 9 (3):668 –673. ISSN 1520-9210. doi: 10.1109/TMM.2006.887994. Citado na pág.

XilVirt7 () XilVirt7. Xilinx - Processadores Xilinx 7. http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/index.htm, 2013. [Online; accessed 22-Aug-2013]. Citado na pág.

YI e LAZZARINI (2012) Steven YI e Victor LAZZARINI. Csound for android. *Linux Audio Conference 2012*. Citado na pág.

Zölzer (2002) Udo Zölzer. *DAFX:Digital Audio Effects*. John Wiley & Sons. ISBN 0471490784. URL http://www.worldcat.org/isbn/0471490784. Citado na pág.