UNIVERSIDADE FEDERAL DE OURO PRETO

INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS

DEPARTAMENTO DE COMPUTAÇÃO

JOÃO VITOR CARDOSO DOS SANTOS COTTA

OFFLOADING COMPUTACIONAL EM ANDROID: ANÁLISE COMPARATIVA DE CÁLCULOS EXECUTADOS EM CPU VERSUS GPU
UMA ABORDAGEM PRÁTICA COM TENSORFLOW LITE E DELEGATES PARA OTIMIZAÇÃO DE TAREFAS VETORIAIS

Ouro Preto
2026

JOÃO VITOR CARDOSO DOS SANTOS COTTA

OFFLOADING COMPUTACIONAL EM ANDROID: ANÁLISE COMPARATIVA DE CÁLCULOS EXECUTADOS EM CPU VERSUS GPU
UMA ABORDAGEM PRÁTICA COM TENSORFLOW LITE E DELEGATES PARA OTIMIZAÇÃO DE TAREFAS VETORIAIS

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Eduardo Jose da Silva Luz
Coorientador: Guilherme Tavares de Assis

Ouro Preto
2026

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

João Vitor Cardoso dos Santos Cotta

Offloading Computacional em Android: Análise Comparativa de Cálculos Executados em CPU versus GPU: Uma abordagem prática com TensorFlow Lite e Delegates para otimização de tarefas vetoriais

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 25 de Fevereiro de 2026

Membros da banca

Eduardo Jose da Silva Luz (Orientador) - Doutor - Universidade Federal de Ouro Preto
Arthur Negrão de Faria Martins da Costa (Examinador) - Bacharel - Universidade Federal de Ouro Preto - PPGCC
Joubert de Castro Lima (Examinador) - Doutor - Universidade Federal de Ouro Preto

Eduardo Jose da Silva Luz, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 25/02/2026

Documento assinado eletronicamente por **Eduardo Jose da Silva Luz**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 04/03/2026, às 20:05, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **1062459** e o código CRC **CBCC503F**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

Dedico este trabalho integralmente à minha mãe, Chirley Cardoso, que abdicou de boa parte de seus sonhos em prol dos meus.

Agradecimentos

Sempre serei grato àqueles que, ao longo do tempo, enxergaram em mim um diamante a ser lapidado — não apenas pelo que eu era, mas pelo que ainda me tornaria. A todos que estiveram comigo nos momentos difíceis, oferecendo aconchego e amparo, deixo minha profunda gratidão. Àqueles que acreditaram no poder do estudo como caminho, que incentivaram, insistiram e nunca me deixaram desistir, meu reconhecimento sincero: foi esse apoio que me permitiu transpor as barreiras impostas pela vida. Aos que me tiraram sorrisos, meu sincero agradecimento: sem vocês, esse caminho rugoso teria sido muito mais difícil e turbulento. Aos que permaneceram, prometo ir mais longe, carregando o legado de vocês comigo, até a eternidade.

“Acceleration is not free. The cost of data movement and execution orchestration can outweigh computational gains when granularity is insufficient.” KIM et al., 2021

“For many edge workloads, the dominant cost is not computation itself, but the overhead of invoking accelerators and moving data across heterogeneous components.” VERMA et al., 2021

“Energy efficiency, not raw performance, ultimately determines the sustainability of continuous computation on mobile and edge devices.” MITTAL, 2015

Resumo

Aplicações móveis baseadas em monitoramento contínuo, como análise biométrica, fisiológica e sensorial, impõem desafios computacionais relevantes quando executadas diretamente na borda. Diferentemente de ambientes em nuvem, dispositivos móveis operam sob restrições severas de bateria, latência e disponibilidade de rede, o que torna inviável depender exclusivamente de processamento remoto ou de estratégias ingênuas de delegação computacional. Este trabalho apresenta uma pesquisa experimental em computação de borda voltada à otimização da execução de algoritmos matemáticos fundamentais em dispositivos Android. O foco está na análise de desempenho e eficiência energética de operações estatísticas e espectrais clássicas, especificamente a *Mean Absolute Deviation* (MAD) e a Transformada Rápida de Fourier (FFT), amplamente utilizadas em *pipelines* de processamento contínuo de dados sensoriais. Nessas aplicações, decisões sobre como estruturar e executar cada etapa do cálculo impactam diretamente a latência, o consumo energético e a viabilidade do sistema em execução contínua. A proposta investiga o uso de aceleração por meio de *delegates* do TensorFlow Lite (CPU, GPU e NNAPI), explorando diferentes níveis de granularidade computacional e estratégias de *batching*. O objetivo é avaliar em quais condições a delegação local para backends especializados se mostra vantajosa, considerando o tempo total de execução do *pipeline* e os custos internos associados à preparação e organização dos dados no *runtime*. Para isso, modelos otimizados no formato `.tflite` foram desenvolvidos e integrados a um aplicativo Android, permitindo uma avaliação sistemática de tempo de execução, *speedup* e comportamento energético em dispositivos com diferentes perfis de hardware. Os resultados demonstram que a delegação adequada das operações, combinada ao controle explícito da granularidade do *pipeline*, permite reduzir significativamente a latência e a demanda energética relativa, especialmente em cenários de execução contínua e em dispositivos de entrada. Tais descobertas reforçam a viabilidade de *pipelines* computacionais eficientes na borda e oferecem diretrizes práticas para o desenvolvimento de aplicações móveis mais responsivas, econômicas e resilientes em ambientes com restrições de energia, conectividade e heterogeneidade de hardware.

Palavras-chave: Computação em Borda; TensorFlow Lite; Delegação Computacional; Granularidade de Processamento; Eficiência Energética Relativa.

Abstract

Mobile applications based on continuous monitoring, such as biometric, physiological, and sensor data analysis, face significant computational challenges when executed directly at the edge. Unlike cloud-based environments, mobile devices operate under strict constraints on battery capacity, latency, and network availability, making exclusive reliance on remote processing or naive computation delegation strategies impractical. This work presents an experimental study on edge computing focused on optimizing the execution of fundamental mathematical operations on Android devices. The analysis targets classical statistical and spectral processing blocks — namely Mean Absolute Deviation (MAD) and Fast Fourier Transform (FFT) — which are widely employed in continuous sensor data processing pipelines. In such contexts, decisions on how computation stages are structured and executed directly impact latency, energy consumption, and system sustainability under continuous workloads. The proposed approach investigates hardware acceleration through TensorFlow Lite delegates (CPU, GPU, and NNAPI), examining the effects of computational granularity and batching-based execution strategies. The goal is to assess when local delegation to specialized backends is beneficial, considering the total pipeline execution time and the internal overheads associated with data preparation and runtime organization. Optimized `.tflite` models were developed and integrated into an Android application, enabling a systematic evaluation of runtime performance, speedup, and energy-related behavior across heterogeneous devices. Experimental results show that appropriate delegation combined with coarse-grained, granularity-aware pipeline design significantly reduces execution time and relative energy demand, particularly under continuous workloads and on entry-level hardware. These findings highlight the importance of careful granularity and delegation decisions and demonstrate the feasibility of efficient, low-latency, and energy-conscious computation at the mobile edge under real-world constraints.

Keywords: Edge Computing; TensorFlow Lite; Computational Delegation; Processing Granularity; Relative Energy Efficiency.

Lista de Figuras

Lista de Abreviaturas e Siglas

CPU	Central Processing Unit
DFT	Discrete Fourier Transform
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
GPGPU	General-Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
MAD	Mean Absolute Deviation
NNAPI	Neural Networks Application Programming Interface
NPU	Neural Processing Unit
SIMD	Single Instruction, Multiple Data
SoC	System-on-a-Chip
TCC	Trabalho de Conclusão de Curso
TFLite	TensorFlow Lite
VkFFT	Vulkan Fast Fourier Transform

Lista de Símbolos

N	Número de amostras do vetor
i	Índice da amostra
x_i	i-ésima amostra do sinal
μ	Média aritmética das amostras
Σ	Operador de somatório
$\lvert \cdot \rvert$	Operador de valor absoluto

Sumário

1 Introdução

A presente seção apresenta o contexto geral deste trabalho e os principais elementos que orientam sua realização. Inicialmente, na Seção 1.1, é apresentada a contextualização do tema e a motivação que levou ao desenvolvimento da pesquisa. Em seguida, na Seção 1.2, é descrito o problema de pesquisa que orienta o estudo. A Seção 1.3 apresenta os objetivos geral e específicos do trabalho, enquanto a Seção 1.4 descreve as hipóteses investigadas. Na Seção 1.5 é discutida a justificativa do estudo, destacando sua relevância acadêmica e prática. Por fim, a Seção 1.6 apresenta a estrutura do restante do trabalho.

1.1 Contextualização e motivação

Nas últimas décadas, smartphones e wearables passaram a oferecer capacidades computacionais cada vez maiores. O aumento progressivo do número de núcleos de CPU, a incorporação de GPUs integradas e, mais recentemente, de aceleradores especializados (como DSPs e NPUs) transformou esses dispositivos em pequenas plataformas de computação capazes de executar aplicações cada vez mais complexas. Esse avanço acompanha uma tendência mais ampla na computação, marcada pelo uso crescente de GPUs e arquiteturas heterogêneas. (OWENS et al., 2008; VOLKOV; DEMMEL, 2008).

Esse avanço possibilitou a consolidação de cenários de computação em borda, nos quais a coleta, o processamento e a interpretação de sinais passam a ocorrer diretamente no próprio aparelho, reduzindo a dependência de servidores na nuvem. O paradigma de *edge computing* é especialmente relevante em aplicações de monitoramento fisiológico, ergonomia industrial, segurança do trabalho e telemetria pessoal, nas quais frequentemente é necessário reagir em tempo quase real e, ao mesmo tempo, preservar a privacidade dos usuários (SHI et al., 2016; SATYANARAYANAN, 2017). Em muitos desses cenários, transmitir continuamente dados brutos para a nuvem é inviável, seja pelo custo energético, seja por limitações de conectividade ou por requisitos de confidencialidade.

Dentro desse cenário, operações clássicas de processamento de sinais, como MAD e FFT, continuam sendo amplamente utilizadas. Essas operações permitem extrair de medidas de variabilidade temporal e de informações espectrais a partir de séries multissensores, permitindo, por exemplo, a detecção local de padrões anômalos, a caracterização de eventos e o disparo de alertas sem a necessidade de transmitir dados brutos para fora do dispositivo. Estudos recentes em *wearables* reforçam que estatísticas simples por janela de tempo, combinadas a transformadas espectrais, seguem sendo blocos centrais na análise eficiente de dados sensoriais sob restrições de energia (AYACHI; BOUGUILA; AYED, 2022; ZHANG et al., 2021).

A execução dessas análises diretamente em *smartphones* e *wearables* contribui para a redução de latência, diminui o gasto energético associado à comunicação e permite que o sistema continue operando mesmo em ambientes com conectividade limitada ou inexistente. Porém, esses dispositivos operam sob restrições severas de energia, largura de banda de memória e capacidade de processamento (TZENG et al., 2012; MITTAL, 2015). Em cenários de uso contínuo, como monitoramento de turnos longos de trabalho, pequenas melhorias de eficiência computacional podem ter impacto acumulado ao longo do tempo. Na prática, isso pode resultar em maior autonomia de bateria e menor estresse térmico do dispositivo. Assim, a necessidade de otimizar ao máximo o processamento local emerge como um requisito prático, e não apenas como uma escolha de engenharia.

Nesse cenário, a decisão sobre onde executar cada etapa do processamento — seja na CPU, na GPU ou em aceleradores especializados acessados via NNAPI — não pode ser reduzida a uma simples comparação de latência. Trabalhos clássicos sobre computação em GPU mostram que arquiteturas altamente paralelas podem alcançar speedups superiores a 10× em relação a implementações equivalentes em CPU para determinadas classes de algoritmos, especialmente aqueles com alto grau de paralelismo de dados (OWENS et al., 2008; VOLKOV; DEMMEL, 2008). No entanto, em sistemas heterogêneos móveis, a escolha do dispositivo de execução envolve também considerações de eficiência energética, pois diferentes unidades de processamento apresentam perfis distintos de consumo e desempenho (TZENG et al., 2012).

Na prática, isso caracteriza um problema de otimização com múltiplos fatores envolvidos. Em aplicações de monitoramento contínuo, pequenas diferenças no perfil de execução podem se acumular ao longo do tempo, resultando em impactos relevantes no consumo energético e, consequentemente, na autonomia do dispositivo.

Trabalhos recentes indicam que o desempenho real de inferência e de operadores matemáticos em ambientes de borda depende fortemente das otimizações introduzidas pelos compiladores e da forma como CPU, GPU e NPU são organizadas dentro dos *pipelines* de execução (VERMA et al., 2021; HUANG et al., 2024; IGNATOV et al., 2025). O desempenho, portanto, não está vinculado apenas à seleção do algoritmo ou do modelo matemático, mas também à forma como o cálculo é estruturado, particionado e paralelizado. Aspectos como granularidade computacional (tamanho dos blocos de dados), fusão de operadores, reuso de buffers e encapsulamento das etapas dentro do grafo de execução tornam-se centrais para explorar o paralelismo disponível e reduzir gargalos de execução.

Nesse cenário, o avanço das arquiteturas heterogêneas em sistemas móveis, aliado ao surgimento de bibliotecas otimizadas como o TensorFlow Lite, viabiliza a delegação de cálculos para unidades de processamento alternativas — CPU, GPU ou NPU — por meio de *delegates*. Essa estratégia representa uma forma prática de *offloading* computacional local, no qual operações específicas são redirecionadas a aceleradores internos para reduzir tempo de execução ou consumo energético. Neste trabalho, o termo *offloading* é utilizado no sentido de delegar cálculos para

aceleradores internos do dispositivo. Também, reconhece-se que as ferramentas disponíveis atualmente não permitem observar diretamente o fluxo entre CPU, GPU e NPU em tempo real; os efeitos do offloading são aqui avaliados de forma indireta, com base em tempos de execução e perfis energéticos obtidos via instrumentação.

No caso específico das operações espectrais, estudos recentes investigam implementações de FFT de alto desempenho em NPUs e técnicas *in-place* voltadas para sistemas de borda, evidenciando que ainda há espaço para otimizações específicas em arquiteturas móveis e embarcadas (LI et al., 2024; VASILAKIS et al., 2025; ZHANG et al., 2021). Esses trabalhos indicam que a transferência de dados e o mapeamento das operações para o hardware adequado não são, por si só, positivos ou negativos: seus efeitos dependem do perfil da aplicação, do hardware disponível, do tamanho dos vetores e da forma como o *pipeline* é estruturado. Esse cenário reforça a necessidade de análises criteriosas que considerem diferentes estratégias de execução ao avaliar o comportamento de MAD e FFT em dispositivos Android reais, especialmente em contextos de monitoramento contínuo e restrições energéticas.

Afim de abordar esse problema, este trabalho não tem como foco avaliar modelos completos de aprendizado de máquina. O foco está, de forma deliberada, em operadores matemáticos fundamentais de processamento de sinais — especificamente MAD e FFT — analisados de maneira isolada e controlada. Essa escolha ajuda a entender com mais clareza os custos computacionais e energéticos desses blocos.

1.2 Problema de pesquisa

Grande parte da literatura atual sobre computação em borda foca na execução de redes neurais completas em dispositivos móveis. Existem diversos estudos e *benchmarks* específicos para TensorFlow Lite e para *System-on-Chips* (SoCs) usados em *smartphones*, com ênfase em cenários de inferência de visão computacional e processamento multimídia (Ignatov, Andrey et al., 2018; MLCommons Association, 2022; HASHEMI; FATEMI; BRETSCHNEIDER, 2015; KIM et al., 2021). Apesar desse avanço, ainda há poucas análises sistemáticas voltadas a blocos matemáticos fundamentais de processamento de sinais, como *Mean Absolute Deviation* (MAD) e *Fast Fourier Transform* (FFT), executados diretamente em ambientes Android reais, considerando simultaneamente tempo de execução e implicações energéticas.

Em muitos trabalhos, MAD, FFT e operações similares aparecem apenas como “detalhes de implementação” dentro de arquiteturas mais complexas, sem que seus custos individuais sejam isolados e quantificados em diferentes faixas de hardware (AYACHI; BOUGUILA; AYED, 2022; ZHANG et al., 2021). Na prática, esses blocos são justamente os responsáveis por varrer janelas de dados, calcular estatísticas, transformar sinais para o domínio da frequência e alimentar estágios posteriores de decisão. Quando o sistema precisa operar em regime contínuo, com janelas deslizantes e múltiplos sensores, pequenas ineficiências nesses operadores básicos se

propagam e passam a dominar o consumo de CPU, energia e tempo de resposta do aplicativo.

Também, a literatura recente em FFT e computação de alto desempenho mostra um esforço crescente para acelerar transformadas e operadores numéricos em arquiteturas especializadas, seja em GPUs (OWENS et al., 2008; VOLKOV; DEMMEL, 2008), em NPUs (LI et al., 2024) ou em métodos *in-place* projetados para cenários de borda e espaço (VASILAKIS et al., 2025). Esses trabalhos reforçam que o ganho de desempenho não depende apenas do algoritmo em si, mas também de como o cálculo é estruturado em nível de vetor, de memória e de paralelismo. Porém, boa parte dessas contribuições é avaliada em ambientes controlados ou plataformas específicas, e raramente traduzida para a realidade de aplicações Android que precisam conviver com multitarefa, limitações térmicas e variação de carga ao longo do dia.

Em paralelo, estudos sobre compiladores, *runtimes* e *delegates* para inferência na borda mostram que a escolha de caminhos de execução, grafos e estratégias de delegação de operadores (CPU versus GPU versus NPU) altera significativamente o balanço entre custo de cálculo e *overhead* de movimentação de dados (VERMA et al., 2021; KIM et al., 2021; TensorFlow Team, 2024). Esses resultados ajudam a entender o comportamento de redes neurais completas, mas ainda deixam em aberto uma questão prática para a comunidade acadêmica e para desenvolvedores Android: qual é, na prática, o custo de executar operações clássicas de sinal, como MAD e FFT sobre janelas vetoriais, com diferentes *delegates* do TensorFlow Lite, e como esse custo varia com o tipo de hardware, o tamanho dos vetores e o padrão de uso (processamento contínuo em janelas simples ou em lote)?

Diante disso, o problema central deste trabalho pode ser resumido da seguinte forma: analisar empiricamente como diferentes estratégias de execução — CPU Kotlin versus delegados do TensorFlow Lite para CPU, GPU e NNAPI, com e sem *batching* — se comportam ao implementar MAD e FFT em sinais multissensores, levando em conta métricas de tempo de execução e implicações energéticas. Em especial, busca-se entender o impacto do tipo de hardware, do tamanho dos vetores e do uso de *batching* na eficiência global dos *pipelines*.

Uma hipótese adicional, explorada ao longo do texto, é que o desempenho percebido não depende apenas da escolha do *delegate*, mas também da *granularidade computacional* adotada. Aqui, essa granularidade envolve três aspectos interligados: (i) o tamanho das janelas de dados, (ii) a quantidade de janelas processadas por chamada (*lote*) e (iii) o quanto da lógica de processamento é encapsulado dentro do grafo TensorFlow Lite em vez de permanecer no código nativo. Entender essa interação é essencial para projetar *pipelines* de MAD e FFT que combinem baixa latência, uso eficiente de energia e robustez em cenários contínuos de computação em borda.

1.3 Objetivos

Objetivo geral: Avaliar o desempenho e o comportamento de *pipelines* de MAD e FFT em dispositivos Android, combinando implementações nativas em Kotlin com delegados do TensorFlow Lite, observando tempos de execução, efeitos do *batching* e estimativas de consumo energético sob diferentes granularidades de cálculo.

Objetivos específicos:

- Comparar o desempenho dos delegados CPU, GPU e NNAPI em relação ao *baseline* em CPU Kotlin na execução de MAD e FFT em sinais multissensores;
- Analisar como variações no tamanho do vetor afetam o tempo de processamento, o *throughput* por amostra e a estabilidade das estatísticas;
- Investigar o efeito do *batching* como estratégia para amortizar o custo fixo de transferência de dados entre CPU, memória principal e estruturas internas do TensorFlow Lite;
- Estudar a influência da granularidade computacional sobre o tempo total de execução, incluindo tamanho das janelas, número de janelas por chamada e encapsulamento do *pipeline* dentro do grafo;
- Realizar uma análise exploratória do impacto energético em diferentes perfis de hardware, correlacionando tempo de execução e estimativas de energia;
- Derivar recomendações práticas para o projeto de *pipelines* de processamento de sinais em dispositivos Android.

1.4 Hipóteses de pesquisa

- **H1**: A execução das operações MAD e FFT por meio dos delegados do TensorFlow Lite(CPU Delegate, GPU Delegate e NNAPI Delegate), que utilizam implementações otimizadas em bibliotecas nativas e aceleração por hardware quando disponível, apresenta tempo de execução significativamente menor quando comparada a uma implementação equivalente dessas operações realizada diretamente em CPU por meio de código Kotlin convencional, sem otimizações específicas ou uso de aceleradores.
- **H2**: Dispositivos de entrada tendem a apresentar ganhos relativos maiores ao utilizar *delegates* GPU/CPU, por conta das limitações de seus núcleos ARM e subsistemas de memória.
- **H3**: A granularidade computacional — tamanho das janelas, número de janelas por chamada e encapsulamento do pipeline — influencia diretamente o *speedup*, o tempo de execução e o consumo energético observado.

1.5 Justificativa

A motivação prática deste estudo está ligada à necessidade de soluções confiáveis para sistemas de monitoramento contínuo, em contextos nos quais decisões rápidas e baseadas em dados podem evitar situações críticas, por exemplo em aplicações nas áreas de saúde, ergonomia e segurança. Em muitos desses cenários, transmitir dados brutos continuamente para a nuvem não é viável, seja por limitações de conectividade, privacidade ou consumo de energia (SHI et al., 2016; SATYANARAYANAN, 2017). Assim, torna-se importante que operações básicas de processamento de sinais, como MAD e FFT, possam ser executadas localmente de forma eficiente nos próprios dispositivos (TZENG et al., 2012).

Ao focar nesses blocos fundamentais, o trabalho contribui com uma visão detalhada sobre seu custo computacional em Android, algo pouco abordado na literatura, apesar da relevância prática. Isso permite otimizar sistemas simples e sofisticados, ao entender melhor o impacto de cada etapa sobre o tempo e a energia.

Também, a análise de granularidade computacional busca preencher lacunas práticas: muitos estudos ignoram como tamanho de janelas, uso de *batching* e organização interna do pipeline impactam o custo real de execução. Ao tornar isso explícito em MAD e FFT, o trabalho fornece diretrizes úteis para desenvolvedores Android.

1.6 Estrutura do trabalho

O Capítulo 1 apresenta o contexto, motivação, problema, objetivos, hipóteses e justificativa.

O Capítulo 2 reúne a revisão bibliográfica sobre computação em borda, MAD, FFT, TensorFlow Lite e trabalhos relacionados.

O Capítulo 3 detalha a metodologia, ambiente experimental, arquitetura, modelos implementados, granularidade computacional e protocolos de teste.

O Capítulo 4 discute os resultados obtidos: tempos, *speedups*, energia e efeitos da granularidade em cada cenário e dispositivo.

Por fim, o Capítulo 5 sintetiza as conclusões, limitações e possíveis extensões futuras.

2 Revisão Bibliográfica

Neste capítulo são apresentados os principais conceitos e trabalhos que servem de base para este TCC. A ideia é contextualizar o uso de aceleração computacional em dispositivos móveis, com foco em comparações entre CPU e GPU, uso de *frameworks* como TensorFlow Lite, APIs gráficas, granularidade computacional e duas operações matemáticas centrais neste trabalho: MAD e FFT. Também são discutidos estudos sobre computação paralela, *offloading* computacional (isto é, a redistribuição seletiva de partes do cálculo entre CPU, GPU, NPU e outros aceleradores), heterogeneidade de hardware em aparelhos Android atuais, eficiência energética e o papel de compiladores e NPUs em cenários de inferência em borda.

2.1 Trabalhos relacionados

O uso de GPU para acelerar cálculos não é algo novo. Desde meados dos anos 2000, a comunidade começou a estudar o uso de placas gráficas para tarefas que vão além do *rendering*, no movimento conhecido como *General-Purpose computing on Graphics Processing Units* (GPGPU). Owens et al. (2008) mostram que o paralelismo massivo das GPUs traz ganhos importantes em algoritmos vetoriais, operações de matrizes e processamento de sinais (OWENS et al., 2008). Em paralelo, Volkov e Demmel (2008) apresentam *benchmarks* em que a GPU supera a CPU em tarefas como álgebra linear densa e transformadas rápidas, reforçando a ideia de que a GPU é naturalmente adequada para operações matemáticas bem estruturadas (VOLKOV; DEMMEL, 2008). Em linhas gerais, esses trabalhos fundam a visão de que partes intensivas do cálculo podem ser explicitamente transferidas para unidades altamente paralelas, desde que a granularidade do problema seja adequada.

Em dispositivos móveis, esse raciocínio começou a ser adaptado à realidade de *smartphones* e *tablets*. Tzeng et al. (2012) mostram que, em muitos casos, o uso da GPU embarcada pode reduzir tanto o tempo de execução quanto o consumo energético em algoritmos intensivos, especialmente quando o paralelismo de dados é bem explorado (TZENG et al., 2012). De forma complementar, Kim et al. (2021) discutem que a presença de memória unificada e o desenho do *System-on-Chip* influenciam diretamente o custo de movimentação de dados e o ponto de equilíbrio entre CPU e GPU em plataformas móveis (KIM et al., 2021). Trabalhos mais recentes, como Rathore et al. (2024), reforçam que a escolha do alvo de execução é sensível ao tipo de aplicação e ao padrão de acesso à memória em arquiteturas heterogêneas (RATHORE et al., 2024).

Com a popularização da inteligência artificial embarcada, *frameworks* como TensorFlow Lite passaram a oferecer uma camada mais amigável para aproveitar essa heterogeneidade de hardware. O trabalho AI Benchmark, de Ignatov et al. (2018), mostra ganhos em tempo de

execução, típicos entre 2× e 7× ao executar modelos com aceleração por GPU ou NNAPI em *smartphones* (Ignatov, Andrey et al., 2018). De forma parecida, CNNdroid (Hashemi et al., 2015) demonstra que redes convolucionais podem ser aceleradas em Android ao mapear as operações para execução paralela em GPU (HASHEMI; FATEMI; BRETSCHNEIDER, 2015). A própria equipe do TensorFlow relata que delegados GPU podem reduzir de forma significativa o tempo de inferência em modelos densos ou com muitas operações paralelizáveis (TensorFlow Team, 2019). Esses resultados consolidam a prática de *offloading* computacional seletivo: partes pesadas do cálculo são empurradas para aceleradores, enquanto a CPU coordena o fluxo.

Trabalhos mais recentes começam a olhar também para o papel dos compiladores e *runtimes* nesse cenário. Verma et al. (2021) comparam diferentes *deep learning compilers* (TVM, TFLite, Glow, entre outros) em cenários de inferência em borda (VERMA et al., 2021). Eles mostram que a escolha do compilador e das otimizações (fusão de operadores, organização dos tensores, quantização) pode alterar de forma expressiva a latência e o *throughput*, mesmo mantendo o mesmo modelo e o mesmo hardware. Assim, o desempenho observado em inferência na borda passa a depender não apenas do hardware ou do *delegate*, mas também das transformações do compilador e do *runtime* (p. ex., fusão de operadores e organização de tensores), que determinam o custo efetivo de execução e de movimentação de dados. Essa sensibilidade ao *runtime* é especialmente relevante para operações vetoriais curtas, em que *overheads* podem dominar o tempo total — exatamente o cenário investigado neste trabalho para MAD e FFT.

Seguindo essa linha, Huang et al. (2024) propõem um pipeline de *image super-resolution* voltado para dispositivos móveis, explorando de forma coordenada CPU, GPU e NPU (HUANG et al., 2024). O trabalho mostra que dividir as etapas do processamento entre diferentes unidades pode ajudar a equilibrar qualidade de imagem, tempo de resposta e consumo energético. Essa discussão dialoga diretamente com o problema deste TCC, que envolve decidir *onde* e *como* executar blocos como MAD e FFT em um pipeline móvel. Zhang et al. (2021) reforçam essa perspectiva especificamente para FFT, ao otimizar análises espectrais em dispositivos de borda com recursos limitados, mostrando que escolhas de arquitetura, *buffering* e organização de memória podem alterar significativamente o trade-off entre latência e energia (ZHANG et al., 2021).

No contexto mais amplo de IA móvel, Ignatov et al. (2025) analisam o *Mobile AI 2025 Challenge*, focado em *super-resolution* quantizada em NPUs de *smartphones* (IGNATOV et al., 2025). O relatório mostra que modelos quantizados, bem mapeados para as NPUs e com poucos *fallbacks* para CPU, conseguem *speedups* relevantes com menor consumo de energia, desde que sejam respeitadas limitações de memória e de formato numérico. Esses resultados dialogam com outros trabalhos como MLPerf Mobile e AIEnergy (MLCommons Association, 2022; ZHANG et al., 2024), reforçando o potencial de delegar blocos computacionalmente pesados, inclusive FFT e extratores de atributos, para NPUs ou GPUs em cenários de borda.

Um ponto complementar é o tratamento dos dados e a escolha de formatos numéricos.

Mittal (2015) revisa diferentes estratégias para reduzir consumo energético em sistemas embarcados, incluindo o uso de `float16`, quantização e compactação como formas de economizar memória e energia (MITTAL, 2015). *Benchmarks* mais recentes, como MLPerf Mobile v2.0 (2022) e AIEnergy (Zhang et al., 2024), confirmam que modelos otimizados, seja por quantização, seja por redução de precisão, tendem a rodar mais rápido e gastar menos energia em hardware móvel (MLCommons Association, 2022; ZHANG et al., 2024). Em paralelo, trabalhos como Ayachi et al. (2022) mostram que extratores estatísticos clássicos ainda desempenham um papel central em *wearables*, justamente por oferecerem boa relação custo-benefício entre informação extraída e energia consumida (AYACHI; BOUGUILA; AYED, 2022).

2.1.1 Heterogeneidade de dispositivos

Um desafio prático é que o "Android" não é uma plataforma única: a variedade de SoCs e combinações de CPU, GPU, NPU e DSP é enorme. Kim et al. (2021) discutem esse ponto ao introduzir o conceito de "AI Tax", mostrando que a mesma operação pode ter desempenho muito diferente dependendo do SoC (KIM et al., 2021). Essas diferenças estão relacionadas à microarquitetura, largura de banda de memória, frequências de operação, *drivers* e até da implementação dos *delegates*. Na prática, isso significa que resultados medidos em um único aparelho dificilmente representam o comportamento de todo o ecossistema Android.

Huang et al. (2024) ilustram bem esse problema ao mostrar que a melhor combinação entre CPU, GPU e NPU varia de acordo com o SoC: em alguns casos, a GPU apresenta o melhor equilíbrio; em outros, a NPU é claramente superior (HUANG et al., 2024). De forma semelhante, o desafio Mobile AI 2025, analisado por Ignatov et al. (2025), mostra que modelos quantizados rodando em NPUs diferentes têm variações grandes de desempenho e eficiência energética, mesmo quando o modelo de rede é o mesmo (IGNATOV et al., 2025). Zhang et al. (2021) acrescentam que, mesmo para FFTs relativamente "simples", a forma como o algoritmo é mapeado para cada SoC faz diferença: algumas arquiteturas se beneficiam mais de FFTs em blocos, outras favorecem variações com menos movimentação de memória (ZHANG et al., 2021).

Essa heterogeneidade não impacta apenas o tempo de execução. Ela também afeta o consumo de energia, o comportamento térmico e a estabilidade ao longo do tempo, fatores importantes para aplicações que precisam rodar continuamente, como monitoramento fisiológico ou telemetria em contexto industrial. Em um cenário de borda em que MAD e FFT são executados repetidamente, essas diferenças entre SoCs podem mudar o ponto em que vale a pena usar GPU, NPU ou ficar na CPU, bem como a estratégia de *offloading* mais adequada para cada classe de dispositivo.

2.1.2 Delegates no TensorFlow Lite e compiladores para borda

Modelos executados pelo TensorFlow Lite são representados internamente como grafos computacionais. Nesse tipo de representação, cada nó do grafo corresponde a uma operação (por exemplo, convolução, soma ou transformadas), enquanto as arestas representam tensores que transportam dados entre essas operações. Durante a execução, o mecanismo de inferência percorre esse grafo avaliando as operações de acordo com suas dependências. Essa estrutura permite que partes do grafo sejam delegadas para diferentes unidades de processamento disponíveis no dispositivo, como CPU, GPU ou aceleradores especializados, dependendo do suporte oferecido por cada *delegate* (ABADI et al., 2016; TensorFlow Team, 2024).

De acordo com a documentação oficial do TensorFlow Lite, a plataforma oferece vários *delegates*: CPU (via XNNPACK), GPU (OpenCL/Vulkan), NNAPI, Hexagon DSP, entre outros (TensorFlow Team, 2024). Cada *delegate* suporta apenas um subconjunto de operações. Quando uma operação não é suportada, o TFLite faz *fallback* para CPU, o que reduz o ganho total. Isso faz com que o desenho do grafo, a escolha dos operadores e do tipo numérico sejam decisões importantes para aproveitar bem cada *delegate* e minimizar *offloading* ineficiente (isto é, ida e volta desnecessária entre acelerador e CPU).

Verma et al. (2021) aprofundam essa discussão ao avaliar compiladores de *deep learning* focados em inferência em borda, como TVM, TFLite e outros (VERMA et al., 2021). Eles mostram que fusão de camadas, reordenação de operadores e escolha de *kernels* otimizados podem gerar diferenças grandes de desempenho, mesmo quando, no fim, tudo é executado pelo mesmo *delegate*. Em termos simples: o compilador e o *runtime* fazem parte do jogo tanto quanto o hardware, e podem ser determinantes para que o *offloading* seja vantajoso ou não.

Ignatov et al. (2025), no relatório do *Mobile AI 2025 Challenge*, complementam essa visão ao relatar a experiência de vários times que submeteram modelos de *super-resolution* quantizados para NPUs móveis (IGNATOV et al., 2025). Os resultados indicam que soluções que tiram melhor proveito do *delegate* NPU, evitam *fallbacks* e usam quantização de forma planejada conseguem latências menores e eficiência energética superior em relação a abordagens mais "genéricas". Esse tipo de resultado é relevante para este trabalho porque, embora aqui o foco não seja realizar o offloading, o problema é análogo: decidir quando faz sentido realizar o *offloading* operações como MAD e FFT para o grafo TFLite, e quando é melhor mantê-las em CPU Kotlin.

Além de impactar latência, a escolha do *delegate* altera o perfil de ativação do *System-on-Chip* (SoC) e, consequentemente, o consumo de energia e o comportamento térmico ao longo do tempo. Em linhas gerais, a CPU tende a apresentar maior densidade térmica durante execução contínua de operações vetoriais, pois depende de poucos núcleos (muitas vezes os *big cores*) operando em frequências elevadas para sustentar o *throughput*. Já GPUs e NPUs exploram paralelismo interno mais amplo e podem executar certas classes de operações com menor energia por operação, reduzindo o tempo efetivo de CPU ativa e redistribuindo a dissipação térmica para

outras unidades do chip. Assim, mesmo quando a latência observada é semelhante, o caminho de execução pode diferir substancialmente em termos de energia total consumida, picos térmicos e estabilidade sob regimes contínuos, motivando análises que considerem métricas além do tempo bruto (TZENG et al., 2012; ZHANG et al., 2024; KIM et al., 2021).

Por outro lado, a própria comunidade do TensorFlow aponta limitações importantes: operações como FFT ainda têm suporte incompleto no TensorFlow Lite (TensorFlow Contributors, 2024). Isso abre espaço para a combinação de TFLite com bibliotecas externas mais específicas, como a VkFFT, que implementa FFT sobre Vulkan, CUDA, HIP e outras APIs de mais baixo nível (TOLMACHEV, 2021). Em aplicações onde a FFT é um gargalo, como análise vibracional ou espectral em *wearables*, faz sentido usar TFLite para os blocos neurais e uma biblioteca dedicada para a FFT, especialmente quando se busca um *offloading* mais granular e adaptado ao perfil energético de cada dispositivo.

Essa limitação reforça a necessidade de uma caracterização empírica cuidadosa em dispositivos reais: mesmo quando um *delegate* não suporta integralmente uma operação, o comportamento de *fallback* para CPU, o custo de movimentação de tensores e o custo de ativação do *runtime* podem definir o ponto de equilíbrio entre manter o cálculo em CPU Kotlin ou encapsular etapas no grafo TFLite. Assim, medir esses custos sob diferentes granularidades e perfis de hardware é essencial para orientar decisões de projeto em *pipelines* contínuos baseados em MAD e FFT.

2.2 Granularidade computacional, MAD e FFT

Um recorrente tema na literatura analisada é a importância da granularidade computacional. Em geral, GPUs, NPUs e DSPs só mostram vantagem real quando há “trabalho suficiente” por chamada, ou seja, quando o custo fixo de preparar e transferir os dados é diluído em um volume razoável de operações. Se a granularidade for muito fina (janelas muito pequenas, poucas amostras por chamada), o tempo gasto apenas para acionar o *delegate* pode ser maior do que o tempo de execução do algoritmo na CPU (KIM et al., 2021; VERMA et al., 2021). Em termos de *offloading*, isso significa que é preciso cuidar não só de “para onde” o cálculo é enviado, mas também de “quanto” é enviado por vez.

É importante notar que o termo *granularidade computacional* pode ser interpretado em camadas complementares, que ajudam a explicar por que o mesmo *delegate* pode se comportar de forma distinta dependendo do desenho do pipeline. Neste trabalho, adota-se a seguinte leitura: (i) **granularidade algorítmica**, associada ao tipo de operação e ao seu padrão de paralelismo (por exemplo, MAD com operações elementares por amostra versus FFT com estágios de combinações e acessos mais estruturados); (ii) **granularidade de dados**, relacionada ao tamanho do vetor/janela e ao volume de amostras processadas por chamada (N pequeno tende a acentuar overheads; Tamanho de entradas maiores tendem a favorecer aceleradores); e (iii)

granularidade de execução, ligada à forma como o trabalho é empacotado e despachado para o *runtime*, incluindo *batching* (processar múltiplas janelas por invocação), reuso de buffers e o grau de encapsulamento do pipeline dentro do grafo TFLite (reduzindo idas e voltas entre CPU e acelerador). Essa distinção é útil porque permite separar efeitos de *custo fixo* (ativação do *delegate*, preparação e movimentação de tensores) e de *custo variável* (processamento em si), oferecendo uma base teórica direta para interpretar os resultados experimentais apresentados no Capítulo 4 (KIM et al., 2021; VERMA et al., 2021).

Essa discussão se conecta de forma direta ao problema deste TCC. Tanto MAD quanto FFT são aplicados em janelas deslizantes sobre sinais multissensores, e o projeto do pipeline envolve decidir o tamanho dessas janelas e quantas janelas serão processadas juntas em cada chamada (o *batching*), justamente para amortizar custos fixos de despacho e movimentação de tensores e tornar o *offloading* efetivamente vantajoso.

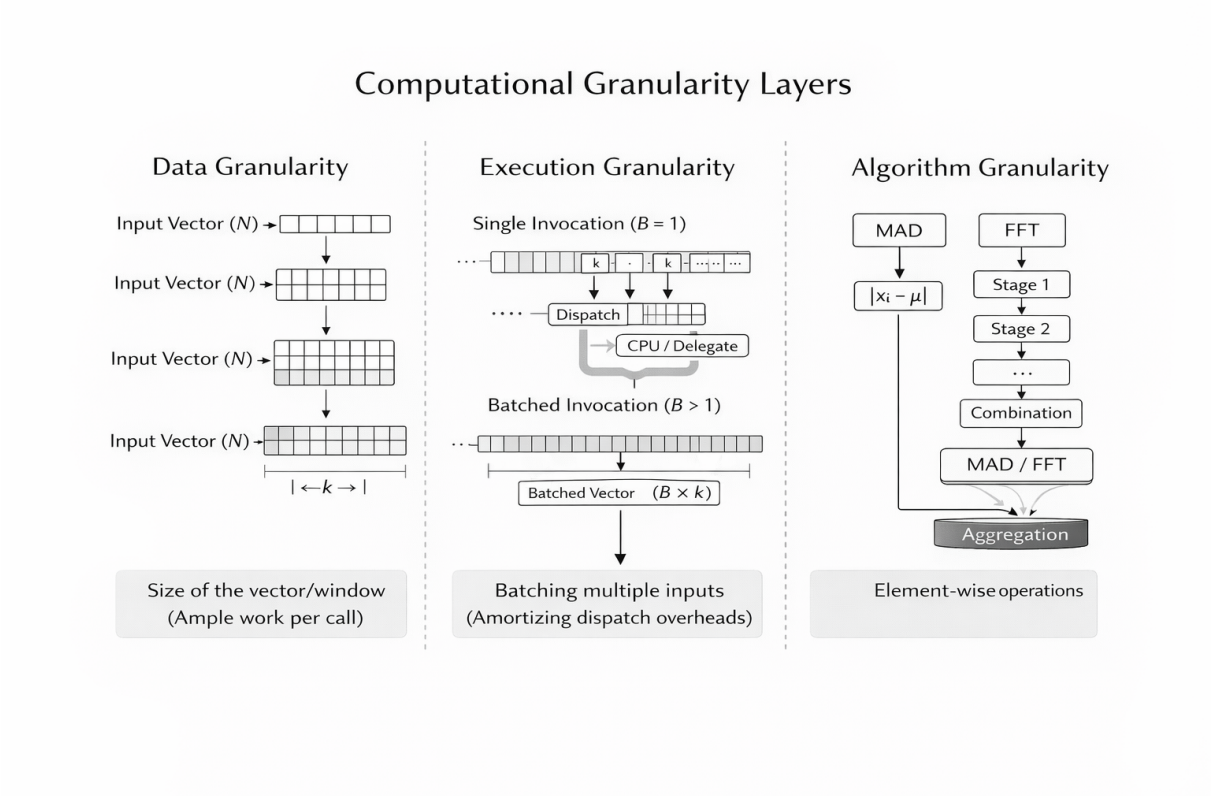

Figura 2.1 – Camadas de granularidade computacional adotadas neste trabalho. A granularidade de dados define o tamanho do vetor processado por execução; a granularidade algorítmica está relacionada à forma como o algoritmo se decompõe internamente em operações menores; e a granularidade de execução diz respeito à forma como o trabalho é empacotado e despachado para o *runtime*. Essas camadas atuam de forma complementar e influenciam diretamente os custos fixos e variáveis do processamento.

2.2.1 MAD — Mean Absolute Deviation

A *Mean Absolute Deviation* (MAD) é uma medida estatística de dispersão que quantifica a variabilidade de um conjunto de valores em relação à sua média. Diferentemente de outras medidas de variabilidade, como a variância ou o desvio padrão, a MAD utiliza apenas operações de subtração, valor absoluto e média aritmética, o que resulta em uma formulação computacionalmente simples. Por essa razão, ela é frequentemente utilizada em aplicações de processamento de sinais e análise de dados provenientes de sensores. Para um vetor de tamanho N, a MAD é dada por:

$$MAD = \frac{1}{N} \sum_{i=1}^{N} |x_i - \mu|,$$

em que μ é a média das amostras. O cálculo é composto por operações simples por amostra: subtração, valor absoluto e soma, seguidos de uma normalização. Isso torna a MAD naturalmente paralelizável, seja via instruções vetoriais na CPU (SIMD), seja em arquiteturas com paralelismo massivo como GPU ou NPU.

Em pipelines de *edge computing* voltados a sinais fisiológicos, vibração ou aceleração, a MAD costuma ser calculada em janelas deslizantes para extrair indicadores de variabilidade local, detectar eventos e alimentar classificadores posteriores. Ayachi et al. (2022), por exemplo, discutem estratégias de extração de atributos estatísticos em dados de vestíveis, destacando que medidas simples, quando bem projetadas, podem ser mais eficientes energeticamente do que modelos mais complexos, especialmente em tarefas que exigem monitoramento contínuo (AYACHI; BOUGUILA; AYED, 2022). Nesses casos, a granularidade, tamanho da janela e quantidade de janelas agrupadas por chamada, influencia diretamente o ponto de equilíbrio entre processar tudo na CPU (por exemplo, em Kotlin puro) ou delegar parte do trabalho para aceleradores via TFLite.

A literatura sobre compiladores e *delegates* reforça que, mesmo em operações simples, o *overhead* de empacotar dados em tensores e chamar o executor pode consumir boa parte do tempo total quando o problema é pequeno demais (VERMA et al., 2021; KIM et al., 2021). Em contrapartida, quando várias janelas e sensores são agrupados em lotes maiores, esse custo fixo é amortizado e o uso de GPU ou NPU passa a ser mais interessante, exatamente o tipo de situação que este trabalho busca medir na prática, ao comparar diferentes granularidades de MAD em dispositivos Android reais.

2.2.2 FFT — Fast Fourier Transform

A *Fast Fourier Transform* (FFT), com complexidade $O(N \log N)$, é um dos algoritmos clássicos em computação de alto desempenho e em GPGPU. Estudos sobre otimização de kernels numéricos mostram que arquiteturas GPU podem obter reduções significativas no tempo

de execução quando comparadas a implementações equivalentes em CPU, especialmente em algoritmos com paralelismo de dados estruturado, como a FFT. Volkov e Demmel (2008) demonstram que o desempenho dessas implementações depende fortemente da organização do acesso à memória e do aproveitamento do paralelismo disponível, destacando o papel de técnicas como *memory coalescing* na melhoria do throughput computacional (VOLKOV; DEMMEL, 2008). Em plataformas móveis, diversos trabalhos indicam que, para tamanhos médios e grandes de FFT, implementações em GPU tendem a apresentar menor tempo de execução do que implementações em CPU, principalmente quando há mecanismos eficientes de transferência e compartilhamento de memória entre os dispositivos de processamento.

Zhang et al. (2021) discutem especificamente a otimização de FFT em dispositivos de borda com recursos restritos, propondo ajustes em algoritmos e estruturas de dados para reduzir acessos à memória externa e melhorar a ocupação dos aceleradores (ZHANG et al., 2021). Os autores mostram que decisões de granularidade, tamanho dos blocos transformados, padrão de *batching* e forma de armazenamento dos dados, têm impacto direto na eficiência da FFT em cenários de borda.

Li et al. (2024) estudam em detalhes algoritmos de Transformada de Fourier de alta performance baseados em NPU para uso em *edge computing* (LI et al., 2024). Eles propõem duas abordagens: um algoritmo de DFT baseado em multiplicação de matrizes (MM-2DFT) e um algoritmo de FFT baseado em operações matriz-vetor (MV-2FFT). Em experimentos com *image filtering*, o MM-2DFT chega a superar uma FFT em GPU Tegra X2 para entradas pequenas, com *speedups* na faixa de 4× a 8×. Já o MV-2FFT tende a se aproximar do desempenho da GPU à medida que o tamanho do problema cresce, e em alguns casos é mais eficiente energeticamente por usar melhor as unidades de matriz/vetor da NPU.

Os autores também destacam que arquiteturas típicas de NPU (como a DaVinci, usada em aceleradores da Huawei) contam com unidades especializadas para operações em `float16`, o que favorece representações matriciais da DFT/FFT e a divisão do problema em blocos menores (LI et al., 2024). Isso é relevante para este trabalho, que avalia FFT em vetores de tamanhos 4 096, 8 192 e 16 384 e observa como essa escala afeta o ponto de equilíbrio entre CPU e aceleradores.

Vasilakis et al. (2025) trazem outra perspectiva ao propor melhorias em FFTs voltadas para aplicações de espaço e borda, com foco em métodos *in-place* que reduzem o uso de memória sem prejudicar a estabilidade numérica (VASILAKIS et al., 2025). Em plataformas com memória limitada, como sistemas embarcados e dispositivos projetados para missões espaciais, o custo de alocar e mover dados pode ser tão importante quanto a própria complexidade $O(N \log N)$ da FFT. Isso reforça a importância de pensar a granularidade não só em termos do tamanho da FFT, mas também de como os dados são organizados em memória e de como o *offloading* é estruturado.

Por fim, trabalhos sobre NPUs móveis e desafios de IA embarcada apontam que FFTs e outras operações espectrais muitas vezes aparecem como pré-processamento pesado antes de

redes neurais (LI et al., 2024; IGNATOV et al., 2025; ZHANG et al., 2021). Em cenários de monitoramento vibracional, análise de fadiga ou detecção de eventos em *wearables*, isso significa que uma parte considerável do custo total pode estar na própria FFT, justificando o foco deste TCC em comparar, de forma direta, CPU Kotlin, TFLite CPU, GPU e NNAPI especificamente para esse bloco.

2.3 Síntese

De forma resumida, a revisão apresentada neste capítulo destaca alguns pontos que são centrais para o desenvolvimento deste trabalho:

1. GPUs e NPUs oferecem grande potencial de paralelismo, mas só trazem ganhos expressivos quando a granularidade do problema é adequada e o custo de *offloading* (preparar, transferir e sincronizar dados) é compensado (OWENS et al., 2008; VOLKOV; DEMMEL, 2008; KIM et al., 2021; VERMA et al., 2021; RATHORE et al., 2024). Em problemas muito pequenos ou pouco agregados, a CPU pode continuar sendo a melhor opção.

2. MAD e FFT aparecem com frequência em aplicações de computação em borda, como monitoramento fisiológico, análise de vibração, ergonomia e *wearables*, e têm perfil naturalmente paralelizável, o que as torna boas candidatas à aceleração por GPU ou NPU. Ao mesmo tempo, trabalhos recentes destacam que a eficiência real dessas operações depende tanto do desenho das janelas e dos lotes quanto do mapeamento para o hardware (AYACHI; BOUGUILA; AYED, 2022; ZHANG et al., 2021; LI et al., 2024; VASILAKIS et al., 2025).

3. O ecossistema Android é altamente heterogêneo em termos de hardware (combinações distintas de CPU, GPU, NPU e DSP), o que torna importante medir o desempenho em mais de um aparelho e considerar que a melhor estratégia de *offloading* pode variar conforme o SoC (KIM et al., 2021; HUANG et al., 2024; IGNATOV et al., 2025).

4. TensorFlow Lite oferece *delegates* que, em muitos casos, trazem ganhos reais de desempenho, mas ainda apresenta limitações, como suporte parcial a FFT e *fallbacks* para CPU quando certas operações não são suportadas (TensorFlow Team, 2024; TensorFlow Contributors, 2024). Em alguns cenários, combinar TFLite com bibliotecas especializadas, como VkFFT, pode ser uma alternativa viável para blocos espectrais (TOLMACHEV, 2021).

5. Otimizações como uso de `float16`, quantização, *batching* e empacotamento cuidadoso de operações em grafos TFLite ajudam a reduzir tempo de execução e consumo de energia, principalmente quando combinadas com compiladores e *delegates* adequados (MITTAL, 2015; MLCommons Association, 2022; ZHANG et al., 2024; VERMA et al., 2021; IGNATOV et al., 2025). Esse tipo de otimização é particularmente relevante em pipelines contínuos que processam dados de sensores em tempo quase real.

6. Trabalhos recentes sobre FFT em NPUs, métodos *in-place* para dispositivos de borda e extração estatística em *wearables* mostram que detalhes de arquitetura (unidades de matriz/vetor, suporte a `float16`) e de organização da memória são tão relevantes quanto a complexidade assintótica do algoritmo (LI et al., 2024; VASILAKIS et al., 2025; ZHANG et al., 2021; AYACHI; BOUGUILA; AYED, 2022).

Apesar dos avanços apresentados, observa-se uma lacuna recorrente na literatura: a maioria dos estudos concentra-se em *benchmarks* de modelos completos de IA embarcada, enquanto a caracterização isolada de operadores matemáticos fundamentais — como MAD e FFT — ainda é menos explorada em Android, especialmente quando se considera simultaneamente tempo, granularidade e implicações energéticas em múltiplos perfis de hardware. Essa lacuna é particularmente relevante em cenários de monitoramento contínuo, nos quais esses blocos podem dominar o custo total do pipeline.

Esses pontos sustentam a proposta deste TCC: investigar, de forma sistemática, o comportamento de CPU Kotlin, TFLite CPU, GPU e NNAPI na execução de MAD e FFT em sinais multissensores, usando diferentes granularidades (tamanho da FFT, número de janelas por lote, maior ou menor encapsulamento no grafo TFLite) e três dispositivos Android que representam faixas distintas de hardware. A partir dessa análise, busca-se oferecer evidências quantitativas e diretrizes práticas tanto para a comunidade acadêmica quanto para desenvolvedores Android interessados em projetar pipelines de processamento em borda mais eficientes, robustos e energeticamente conscientes.

3 Metodologia

Este capítulo apresenta como o estudo foi conduzido na prática, descrevendo as etapas necessárias para avaliar o desempenho e o comportamento energético de *pipelines* de processamento estatístico (MAD) e espectral (FFT) em dispositivos Android. A ideia central foi observar como diferentes estratégias de execução — desde código nativo em Kotlin até delegados do TensorFlow Lite (CPU, GPU e NNAPI) — se comportam em cenários reais, variando granularidade, tamanho das janelas e características do hardware. Ao longo deste capítulo, são detalhados o tipo de pesquisa, o ambiente de desenvolvimento, os modelos utilizados, a arquitetura completa do sistema, os cenários experimentais, as métricas monitoradas e o procedimento adotado para conduzir os experimentos. Ressalta-se que os modelos TensorFlow Lite utilizados não envolvem aprendizado de máquina ou inferência treinável, mas grafos determinísticos que encapsulam operações matemáticas clássicas de processamento de sinais.

Para fins de transparência e reprodutibilidade científica, todo o código-fonte desenvolvido nesta pesquisa, incluindo implementações, scripts experimentais e configurações de execução, está disponível publicamente no repositório GitHub da monografia: <https://github.com/DoisDedin/GPUAndroidUp>.

3.1 Tipo de pesquisa

Este trabalho caracteriza-se como uma pesquisa aplicada, de natureza quantitativa e com delineamento experimental. O foco recai sobre a avaliação empírica de *pipelines* de processamento estatístico (MAD) e espectral (FFT) em dispositivos Android, comparando o desempenho de implementações nativas em CPU com delegados do TensorFlow Lite (CPU, GPU e NNAPI).

Os experimentos são conduzidos em condições controladas, com geração determinística de dados e repetição sistemática dos cenários. Isso permite variar fatores como tamanho das janelas, número de janelas por chamada (lote) e *delegate* utilizado, isolando os efeitos de transferência de memória, processamento e amortização por lotes — elementos centrais para sistemas de computação em borda orientados a sinais fisiológicos e multissensores. O controle experimental envolve a fixação do gerador de dados determinísticos, a repetição sistemática dos cenários e a execução isolada de cada *delegate*, evitando interferência entre caminhos de execução concorrentes.

Isso representa um estudo experimental controlado, voltado à caracterização de desempenho e comportamento energético, e não de um benchmark padronizado nem de um estudo de caso de aplicação final. As implementações avaliadas são propositalmente simplificadas, de modo a isolar o comportamento dos operadores MAD e FFT e reduzir interferências externas.

3.2 Ambiente de desenvolvimento

O ambiente de desenvolvimento foi montado para representar condições reais de execução de um aplicativo Android moderno.

- **Linguagem**: Kotlin, adotada como padrão no ecossistema Android.
- **Ferramentas**: Android Studio (versões Hedgehog/Koala), Android Gradle Plugin 8.6.0 e Gradle 8.7 (via *wrapper*).
- **Configuração do sistema**: *compileSdk* 35, *targetSdk* 34 e *minSdk* 31, garantindo acesso às bibliotecas Jetpack mais recentes.
- **Execução numérica em Android**: TensorFlow Lite 2.16.1, incluindo extensões para GPU e NNAPI, além da biblioteca JTransforms 3.1 como referência de FFT no *baseline* em CPU Kotlin.
- **Pipeline Python para geração de modelos**: Python 3.9+, TensorFlow 2.16, NumPy, SciPy, Matplotlib, Seaborn e Pandas, em ambientes virtuais isolados, utilizados para construir, validar e converter os modelos em formato `.tflite`.
- **Ferramentas de medição em Android**: Android Profiler, Logcat e coleta dos contadores internos de bateria (nWh, mAh, estado de carga, temperatura e modos de economia), além de registros estruturados expostos pelo sistema.

A descrição detalhada das ferramentas e versões visa garantir reprodutibilidade experimental, não implicando dependência conceitual dos resultados a versões específicas do ambiente.

Os testes foram realizados em três dispositivos físicos que representam categorias distintas de hardware:

- **Galaxy S21** — topo de linha, com SoC Exynos 2100;
- **Moto G84 5G** — intermediário, com Snapdragon 695;
- **Moto G04s** — entrada, com Spreadtrum/Unisoc T606.

Essa variedade mostra como delegados e granularidade interagem com limitações reais de CPU, GPU, memória, subsistemas térmicos e gerenciamento de energia, aproximando os experimentos de cenários típicos de uso em campo.

Para reduzir variabilidade associada a aquecimento e escalonamento dinâmico de frequência, os experimentos foram executados com os dispositivos em estado térmico estável, após períodos de resfriamento, e com o aplicativo em primeiro plano. Aplicações em segundo plano foram minimizadas sempre que possível durante a execução dos testes.

3.3 Modelos e funções avaliadas

O estudo se concentra em dois blocos funcionais centrais no processamento multissensorial: o cálculo de *Mean Absolute Deviation* (MAD) e a *Fast Fourier Transform* (FFT). Em ambos os casos, há uma versão de referência em Kotlin (*baseline*) e um modelo equivalente em TensorFlow Lite, permitindo comparar diretamente diferentes estratégias de execução.

Ressalta-se que as implementações de MAD e FFT adotadas neste estudo não têm como objetivo atingir o estado da arte em otimização algorítmica específica, mas sim manter uma estrutura funcional equivalente entre as diferentes estratégias de execução. Assim, busca-se comparar caminhos de execução (CPU Kotlin, TFLite CPU, GPU e NNAPI), e não algoritmos distintos.

3.3.1 Cálculo de Mean Absolute Deviation (MAD)

No domínio de acelerômetros, o pipeline de MAD adotado no estudo segue três etapas principais:

1. cálculo da magnitude tridimensional do acelerômetro a partir dos eixos x, y e z;
2. segmentação dos dados em janelas fixas de cinco segundos;
3. extração das estatísticas por janela: média, MAD, desvio padrão, mínimo e máximo.

Essa lógica está presente tanto na implementação nativa em Kotlin quanto no modelo TensorFlow Lite:

- **Implementação nativa Kotlin** — serve como *baseline* do estudo, implementando diretamente as operações de magnitude, agregações e estatísticas;
- **Modelo TFLite** — encapsula o pipeline completo em um grafo único, construído e exportado em precisão FP32 para manter compatibilidade com os *delegates* CPU, GPU e NNAPI sem perdas de fidelidade numérica.

O modelo `.tflite` é construído por scripts auxiliares em Python que replicam exatamente a lógica do pipeline, validam os resultados contra a versão em Kotlin e empacotam o artefato final nos *assets* do aplicativo para consumo direto pelo interpretador TFLite embarcado.

3.3.2 Transformada Rápida de Fourier (FFT)

O segundo bloco implementa a Transformada Rápida de Fourier com pesos dinâmicos, baseada em operações de *Real FFT* (RFFT). O pipeline de FFT realiza:

- RFFT em janelas de tamanhos variáveis, de 512 até 65 536 amostras por sensor (limite exposto na interface de experimentos);
- aplicação de pesos espectrais dinâmicos sobre os coeficientes no domínio da frequência;
- processamento simultâneo de dez sensores em paralelo;
- normalizações opcionais integradas ao grafo, gerando magnitudes ponderadas em FP32.

A versão nativa em Kotlin, via JTransforms 3.1, é utilizada tanto para validação numérica dos resultados quanto como referência de desempenho em CPU. Assim como no caso da MAD, o modelo FFT em TFLite é construído em Python e convertido para `.tflite`, preservando precisão FP32.

Porém, apesar de compartilharem o mesmo arquivo `.tflite`, os delegados GPU e NNAPI fazem *fallback* para XNNPACK no caso da FFT, pois o TensorFlow Lite ainda não oferece *kernels* FFT nesses *backends*. Na prática, isso significa que toda a execução efetiva ocorre na CPU, e que as diferenças observadas entre delegados para FFT decorrem principalmente de cópias de *buffers*, escolhas internas de *threads* e eventuais ajustes de precisão (FP16/FP32) em cada *runtime*.

Embora a FFT não seja efetivamente executada em GPU ou NPU nos delegados avaliados, sua inclusão nos experimentos é metodologicamente relevante para quantificar o custo real de encapsular operações espectrais no grafo TensorFlow Lite. Esses custos incluem preparação de tensores, cópia de buffers, gerenciamento de *threads* e ativação do *runtime*, que fazem parte do overhead observado por aplicações reais e impactam diretamente a decisão entre manter a FFT em código nativo ou integrá-la ao pipeline TFLite.

Esse comportamento não representa uma limitação do experimento, mas sim uma característica relevante do ecossistema atual do TensorFlow Lite em dispositivos Android, que impacta diretamente decisões práticas de projeto.

Versões experimentais do modelo com janelas de 128 k, 256 k e 524 k amostras continuam disponíveis em scripts Python, mas foram desativadas na interface Android devido às falhas recorrentes de alocação de memória observadas nos dispositivos de teste durante os experimentos conduzidos neste trabalho (ver Seção **??**), um comportamento consistente com as limitações de memória frequentemente relatadas em dispositivos móveis (KIM et al., 2021).

3.4 Arquitetura do sistema e integração com Android

A solução foi organizada em dois módulos principais dentro do ecossistema Android:

- **Biblioteca de suporte** — concentra os geradores de lotes determinísticos de acelerômetro, construtores de tensores (reais, imaginários, magnitudes e pesos), implementações *baseline*

em Kotlin, *wrappers* de inferência TFLite, padronização de *buffers* diretos e utilitários de medição de tempo;

- **Aplicativo Android** — expõe a interface gráfica, o *ViewModel* e os serviços responsáveis por orquestrar as execuções, acompanhar o progresso dos testes, registrar tempos e exportar resultados.

O fluxo executado em cada *benchmark* seguiu, em linhas gerais, as seguintes etapas:

1. definição dos parâmetros experimentais (tipo de operação, *delegate*, granularidade, tamanho da janela e número de iterações);
2. geração dos pacotes determinísticos de dados (modo *SINGLE* ou modo *BATCH*, com número configurável de pacotes por execução);
3. construção dos tensores de entrada de acordo com o modelo selecionado;
4. execução do cenário e registro dos tempos de transferência e de processamento;
5. exportação dos resultados em arquivos CSV e em arquivos de texto legíveis contendo resumos por cenário.

A Figura 3.1 ilustra, de forma resumida, a arquitetura geral do sistema de *benchmarks*, desde a definição dos cenários até a exportação dos resultados. Todo o ciclo ocorre dentro do ecossistema Android, respeitando ciclos de vida de *fragments*, serviços em segundo plano e permissões de acesso ao armazenamento para gravação dos arquivos de saída.

Essa arquitetura foi projetada para permitir controle explícito da granularidade computacional, possibilitando variar o tamanho das janelas e o número de janelas processadas por invocação sem alterar a lógica funcional dos operadores.

3.5 Cenários de teste e métricas

Esta seção descreve como os experimentos foram organizados e executados. Inicialmente, são apresentados os cenários de teste considerados neste trabalho. Em seguida, são descritas as métricas utilizadas para avaliar desempenho e consumo energético. Por fim, é detalhado o procedimento experimental adotado para garantir consistência e reprodutibilidade nas medições realizadas.

3.5.1 Cenários de teste

Cada cenário experimental avaliado neste estudo é definido de forma unívoca pela combinação de quatro fatores:

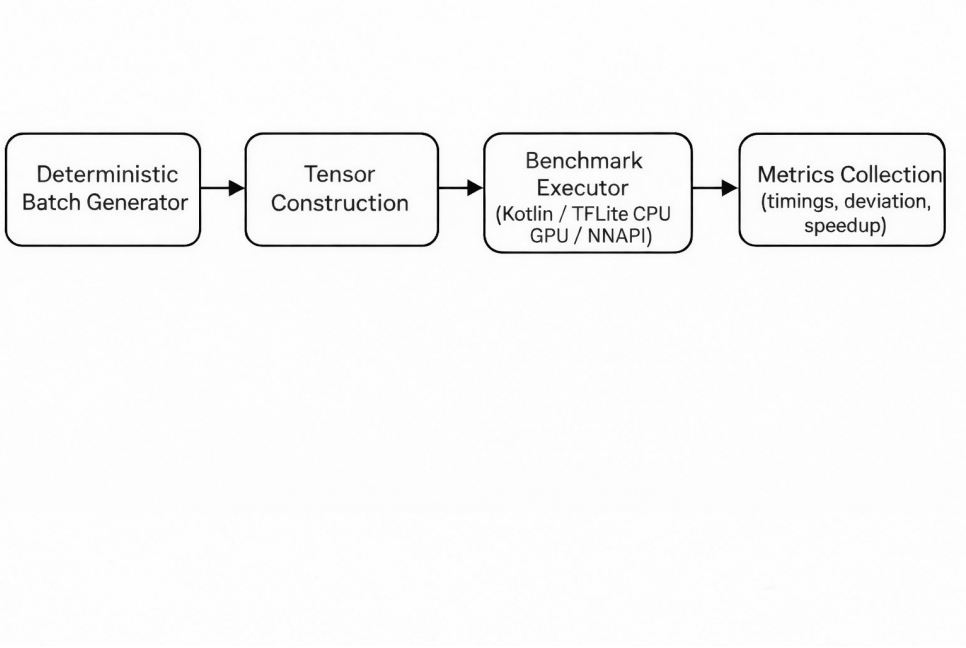

Figura 3.1 – Arquitetura geral do sistema de *benchmarks*: da definição dos cenários à exportação dos resultados.

- **Algoritmo**: Mean Absolute Deviation (MAD) ou Fast Fourier Transform (FFT);
- **Delegate**: CPU Kotlin (baseline), TFLite CPU (XNNPACK), TFLite GPU ou TFLite NNAPI;
- **Modo de lote**: *SINGLE* (1 pacote por execução) ou *BATCH* (N pacotes por execução);
- **Escala**: tamanho do vetor de entrada, com $N \in \{512, 1\,024, 2\,048, 4\,096, 8\,192, 16\,384, 32\,768, 65\,536\}$

O modo de lote (*BATCH*) define a quantidade de pacotes processados dentro de uma mesma execução do cenário, com B (número de pacotes) selecionado na interface do aplicativo entre $\{1, 4, 8, 12\}$. Na campanha principal analisada neste trabalho (Realizada em 09/12/2025), o modo *BATCH* foi fixado em $B = 12$ pacotes por execução. O modo de lote apenas repete pacotes completos dentro de uma mesma execução e não altera a quantidade de sensores contida em cada pacote.

Diferença estrutural entre MAD e FFT

A composição de um pacote difere entre os algoritmos:

- **MAD**: cada pacote contém dados de **1 sensor**.
- **FFT**: cada pacote contém dados de **10 sensores simultâneos**.

Assim, a leitura correta de cada cenário é:

- **MAD SINGLE**: 1 pacote $\times$ (1 sensor $\times N$ amostras);
- **MAD BATCH 12**: 12 pacotes $\times$ (1 sensor $\times N$ amostras);
- **FFT SINGLE**: 1 pacote $\times$ (10 sensores $\times N$ amostras);
- **FFT BATCH 12**: 12 pacotes $\times$ (10 sensores $\times N$ amostras).

Cobertura experimental

Para cada escala N, a suíte completa executa 16 cenários, resultantes de:

$$4 \text{ delegates} \times 2 \text{ algoritmos} \times 2 \text{ modos (SINGLE, BATCH)}.$$

Com oito escalas válidas (512 a 65 536), cada execução completa da suíte corresponde a 128 cenários executados por dispositivo.

3.5.2 Métricas analisadas

As métricas coletadas incluíram:

- **Tempo total de execução** por cenário;
- **Tempo de transferência de tensores** (entrada/saída) quando aplicável;
- **Tempo efetivo de processamento** no *delegate*;
- ***Throughput*** (amostras processadas por segundo);
- ***Speedup*** em relação ao *baseline* Kotlin;
- **Indicadores energéticos diretos** (nWh, mAh, estado de carga e temperatura), complementados por uma etiqueta qualitativa (`estimated_energy`) e pelo modo de economia de energia, para comparação relativa entre cenários;
- **Estabilidade estatística** das medidas (média, desvio padrão, valores mínimo e máximo por cenário).

Notas textuais complementam cada linha nos arquivos de saída com estatísticas adicionais de MAD ou resumos dos primeiros *bins* de FFT, auxiliando na conferência qualitativa dos resultados.

As métricas energéticas analisadas possuem caráter exploratório e comparativo, não substituindo medições diretas com instrumentação externa. O objetivo é identificar tendências relativas entre estratégias de execução sob condições equivalentes.

3.6 Procedimento experimental

O procedimento experimental inicia com a configuração dos *chips* de iteração e do tamanho de lote na interface, conforme o plano de testes. Em seguida, para cada dispositivo, executa-se a suíte completa que percorre todos os cenários de MAD e FFT nas versões *SINGLE* e *BATCH*, cobrindo as oito escalas válidas até 65 536 pontos. Na campanha principal (09/12), o modo *BATCH* foi fixado em 12 pacotes por execução.

Em uma das campanhas experimentais representativas, por exemplo, cada botão de cenário foi acionado cinco vezes seguidas (5×128 execuções), mantendo o gerador determinístico de dados, o que resultou em aproximadamente 640 linhas por dispositivo (além das linhas experimentais com janelas acima de 64 k no Moto G84). Tentativas com comprimentos superiores a 65 k amostras são realizadas apenas quando o pesquisador habilita manualmente os scripts Python/CLI, uma vez que os dispositivos testados tendem a esgotar a RAM antes de concluir esses comprimentos. Esse total decorre da execução de 16 cenários por escala (4 delegates, 2 algoritmos e 2 modos de processamento) ao longo de oito escalas válidas.

Os resultados são exportados automaticamente em arquivos CSV e textos resumidos por cenário. Após a coleta, scripts auxiliares em Python processam os CSVs para gerar gráficos comparativos, tabelas e descrições em linguagem natural, além de consolidações multi-dispositivo. Essa rotina se repete para cada unidade de hardware, armazenando os artefatos em diretórios organizados por dispositivo e data, com um CSV consolidado para comparações cruzadas. O mesmo protocolo é seguido para o teste energético, garantindo que cada conjunto de execuções tenha *logs* completos e consistentes para análise quantitativa e discussão.

A campanha principal analisada neste trabalho corresponde às execuções realizadas, que seguem exatamente esse protocolo e servem de base para os resultados apresentados no Capítulo 4.

3.7 Justificativa metodológica

A escolha da metodologia adotada neste capítulo foi guiada por diversos trabalhos discutidos na revisão bibliográfica (Capítulo 2). Em particular, estudos sobre computação em borda e aceleração por GPU/NPU indicam que:

- a granularidade computacional influencia diretamente o ganho obtido com aceleradores, exigindo janelas e lotes suficientemente grandes para amortizar o custo fixo de *offloading*;
- o TensorFlow Lite introduz custos fixos de preparação e transferência de tensores, que precisam ser medidos explicitamente;
- SoCs móveis apresentam grande heterogeneidade de desempenho, de modo que resultados em um único aparelho não são necessariamente representativos de todo o ecossistema

Android;

- a definição explícita de granularidade, tanto para operações estatísticas quanto espectrais, é essencial para avaliar quando o custo de *offloading* é compensado, especialmente em pipelines contínuos baseados em MAD e FFT.

Um ponto metodológico adicional diz respeito a uma possível alternativa aparentemente intuitiva, mas incorreta, de "executar o mesmo cenário (isto é, o mesmo grafo/mesma computação) em paralelo" em múltiplos alvos (CPU, GPU e NNAPI) para reduzir o tempo total. Essa estratégia não é adequada por três razões principais. Primeiro, os três caminhos executariam o mesmo processamento sobre os mesmos dados de entrada, o que implica duplicação de *buffers* e maior pressão sobre memória e subsistema térmico, elevando o consumo energético e podendo induzir *throttling*. Segundo, como o resultado final ainda depende do término completo da computação para ser utilizado, a latência percebida passa a ser limitada pelo tempo de conclusão do caminho mais lento (além do custo adicional de sincronização), anulando qualquer benefício prático de redundância. Terceiro, o TensorFlow Lite não particiona automaticamente um mesmo grafo para execução concorrente em diferentes aceleradores; em geral, seleciona-se um único *delegate* por configuração e ocorre *fallback* para CPU apenas quando operadores não são suportados pelo *delegate*. Assim, do ponto de vista experimental e de engenharia, comparar caminhos de execução requer selecionar um único *delegate* por execução e medir separadamente seus custos de preparação, transferência e processamento, evitando paralelismo redundante que apenas aumenta variância, consumo e temperatura.

Seguindo essas orientações, os experimentos deste TCC foram estruturados para produzir resultados comparáveis, reprodutíveis e coerentes com cenários reais de computação em borda, permitindo discutir, de forma fundamentada, como diferentes combinações de CPU Kotlin, TFLite CPU, GPU e NNAPI impactam o tempo de execução, o *throughput* e o comportamento energético de pipelines baseados em MAD e FFT em dispositivos Android.

4 Resultados

Este capítulo apresenta os resultados obtidos a partir da campanha de *benchmarks* conduzida com os pipelines de MAD e FFT descritos no Capítulo 3. São reportados os tempos de execução e *speedups* em relação ao baseline em CPU Kotlin, efeitos de granularidade (tamanho de janelas e uso de lotes), uma análise qualitativa de consumo energético com base em contadores internos do sistema operacional coletados durante os benchmarks e um resumo da precisão numérica observada entre os diferentes delegados. A discussão busca sempre relacionar os descobertas ao enquadramento conceitual de granularidade apresentado na Seção 2.2 e à heterogeneidade de hardware entre os dispositivos avaliados.

4.1 Configuração dos benchmarks

A campanha oficial de benchmarks, realizada em 09/12/2025, percorreu **128 cenários por dispositivo** (16 botões $\times$ oito escalas válidas). Os testes foram executados nos três aparelhos físicos descritos na Metodologia:

- **Galaxy S21**, com SoC Exynos 2100 (faixa topo de linha);
- **Moto G04s**, com Unisoc/Spreadtrum T606 (faixa de entrada);
- **Moto G84 5G**, com Snapdragon 695 (faixa intermediária).

Cada cenário contemplou comprimentos de 512, 1 024, 2 048, 4 096, 8 192, 16 384, 32 768 e 65 536 pontos por sensor, mantendo sempre 10 sensores simultâneos no caso da FFT. Como todos os dispositivos saturaram a RAM ao ultrapassar 65 k, as escalas de 128 k, 256 k e 524 k ficaram restritas a experimentos manuais (via scripts auxiliares) e não compõem a suíte oficial de comparação apresentada neste capítulo.

As execuções *SINGLE* processam um único pacote por iteração, enquanto as variantes *x12* reaproveitam os mesmos parâmetros para 12 pacotes consecutivos, preservando o determinismo do gerador de dados e permitindo avaliar o impacto do *batching*. Em cada dispositivo, *cada botão* foi acionado cinco vezes consecutivas (5 $\times$ 128 execuções), mantendo o gerador determinístico; isso resultou em 640 linhas por aparelho (e 816 no Moto G84, que inclui também as tentativas extremas acima de 65 k). Dentro de cada cenário, as 12 iterações internas são usadas para calcular média, desvio padrão e limites mínimos/máximos, reduzindo a variância das medidas.

Nesta seção (e em todo o capítulo), os resultados quantitativos são apresentados com base em `compute_ms`. Embora o aplicativo registre métricas auxiliares de tempo, o Android (e as bibliotecas usadas nesta implementação) não oferecem um mecanismo uniforme que separe

de forma confiável o tempo de *offloading* computacional (movimentação de dados, cópias e sincronizações) do tempo estrito de computação do grafo em cada backend. Por isso, quando este capítulo discute "tempo total de processamento" em gráficos por delegate, entende-se esse valor como o custo agregado do pipeline (*offloading* + computação) observado na execução, sem alegar uma decomposição precisa entre essas parcelas.

Todos os resultados foram consolidados em arquivos CSV e em gráficos gerados a partir dos benchmarks, evidenciando padrões coerentes com a revisão de granularidade (Seção 2.2): crescimento do tempo no baseline em CPU Kotlin e queda acentuada em `compute_ms` quando a granularidade é grossa (janelas maiores e *batching*), em especial no MAD.

Nos trechos a seguir, destacamos resultados para escalas representativas (especialmente 4 096 e 65 536 pontos), sem perder de vista que os mesmos padrões se repetem ao longo de toda a faixa $512 \rightarrow 65\,536$.

4.2 Resultados de MAD

Nesta seção são apresentados os resultados experimentais obtidos para o algoritmo *Mean Absolute Deviation* (MAD) nos dispositivos avaliados. Os resultados são organizados por dispositivo, permitindo observar como diferentes arquiteturas de hardware influenciam o desempenho das estratégias de execução consideradas. Em cada subseção são analisados os tempos de execução (`compute_ms`) obtidos para a implementação em CPU Kotlin e para os delegados do TensorFlow Lite (CPU, GPU e NNAPI), considerando diferentes tamanhos de entrada e modos de processamento (*SINGLE* e *BATCH*).

4.2.1 Galaxy S21

No Galaxy S21, o uso de `compute_ms` evidencia ganhos expressivos para MAD com TFLite CPU/GPU ao longo de toda a faixa $512 \rightarrow 65$ k. Em 4 096 amostras no modo *SINGLE*, a CPU Kotlin registrou 2,64 ms, enquanto os delegados TFLite CPU e GPU atingiram aproximadamente 0,078 ms (*speedup* de cerca de $34\times$); o NNAPI permaneceu em 1,36 ms. Em 65 536 amostras no modo *SINGLE*, a CPU chegou a 402 ms, contra 1,09–1,14 ms nos delegados CPU/GPU (aproximadamente 350–$370\times$) e 8,13 ms no NNAPI.

No modo *x12*, o contraste se acentua: na escala de 65 k, o tempo em CPU Kotlin ficou em torno de 5 176 ms, enquanto TFLite CPU/GPU concluíram o lote em aproximadamente 12,9 ms; o NNAPI permaneceu acima, com cerca de 89,8 ms.

As Figuras 4.1 e 4.2 sintetizam os tempos de execução e os *speedups* do MAD no Galaxy S21.

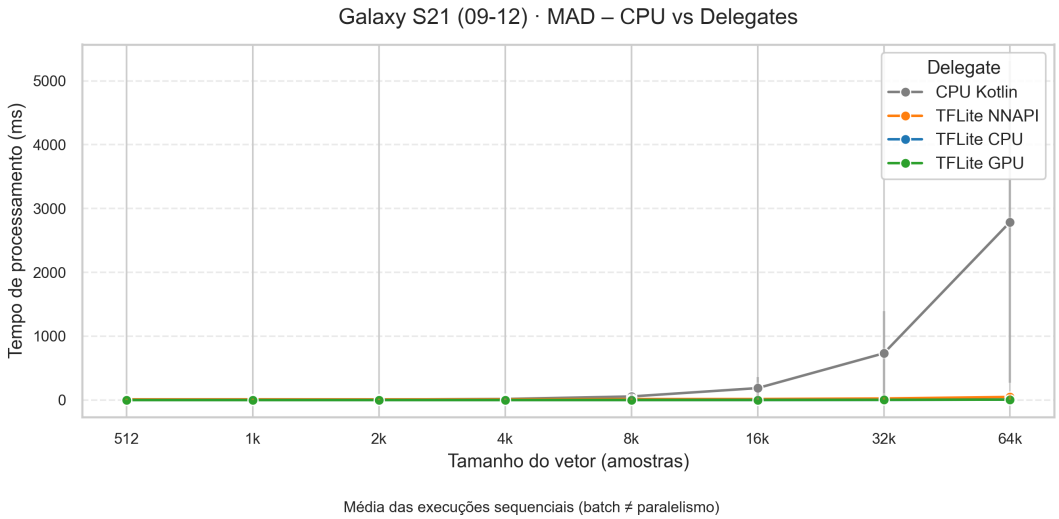

Figura 4.1 – Tempo de execução do MAD no Galaxy S21 para tamanhos representativos (e.g., 4 096 e 65 536 pontos), nos modos *SINGLE* e *x12*, comparando CPU Kotlin e delegados TFLite (CPU, GPU, NNAPI), reportado em `compute_ms`.

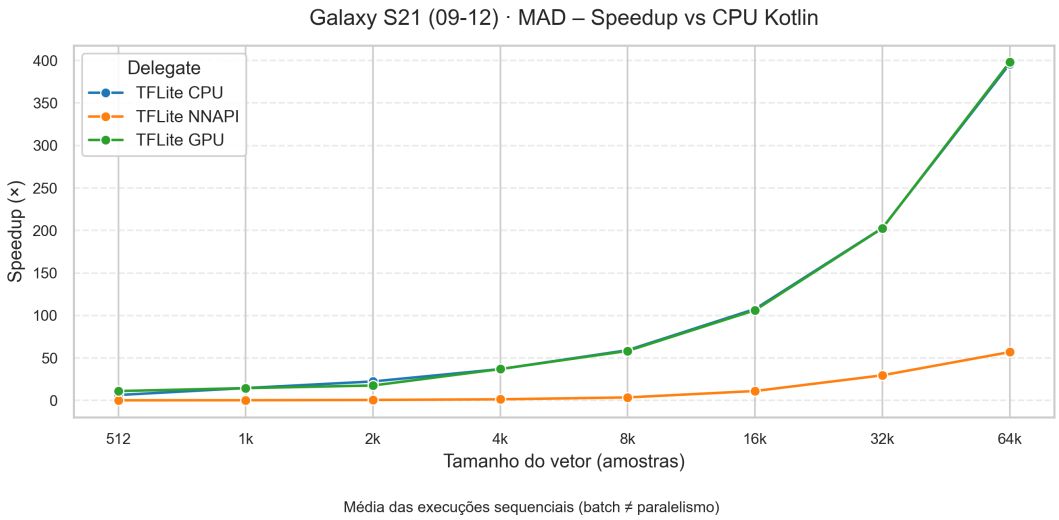

Figura 4.2 – *Speedup* do MAD no Galaxy S21 em relação ao baseline CPU Kotlin, para escalas representativas, nos modos *SINGLE* e *x12*, por delegado TFLite (baseado em `compute_ms`).

4.2.2 Moto G04s

No Moto G04s, o `compute_ms` destaca um contraste forte entre TFLite CPU/GPU e NNAPI. Em 4 096 amostras no modo *SINGLE*, a CPU Kotlin registrou 9,27 ms, enquanto TFLite CPU/GPU alcançaram aproximadamente 0,33 ms (ganho de 28–29×); o NNAPI ficou em 12,47 ms, pior que a CPU. Em 65 536 amostras no modo *SINGLE*, a CPU chegou a 999,9 ms e TFLite CPU/GPU ficaram em torno de 2,9 ms (cerca de 340×), com NNAPI em 55,7 ms.

No modo *x12*, 65 k passou de aproximadamente 12 023 ms (CPU) para cerca de 33,5 ms nos delegados CPU/GPU, enquanto o NNAPI permaneceu alto (cerca de 654 ms).

As Figuras 4.3 e 4.4 apresentam os tempos e *speedups* do MAD no Moto G04s.

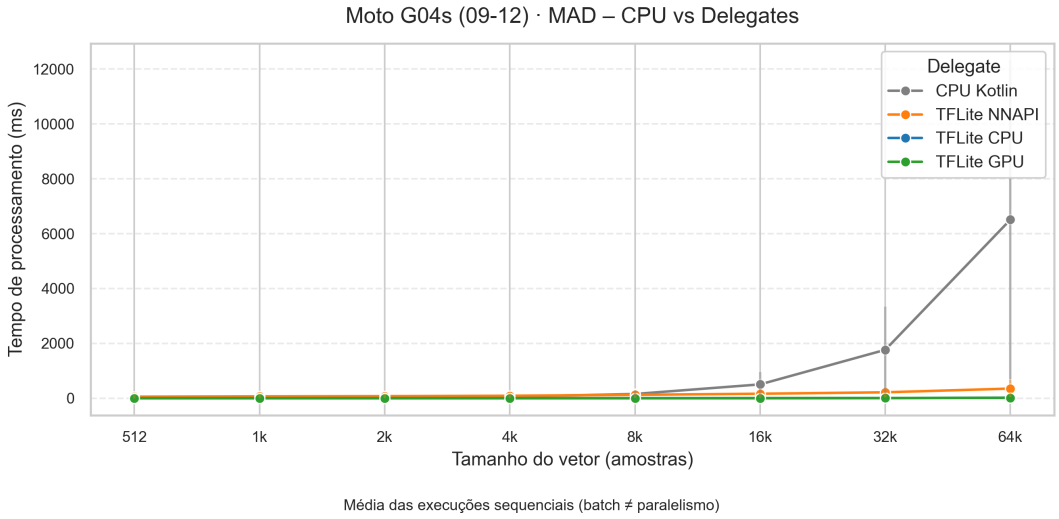

Figura 4.3 – Tempo de execução do MAD no Moto G04s para escalas representativas, nos modos *SINGLE* e *x12*, comparando CPU Kotlin e delegados TFLite (CPU, GPU, NNAPI), reportado em `compute_ms`.

4.2.3 Moto G84 5G

O Moto G84 5G apresentou comportamento estável entre os delegados. Em 4 096 amostras no modo *SINGLE*, a CPU Kotlin ficou em 4,19 ms e os delegados variaram entre 0,135–0,138 ms (cerca de 30×). Em 65 536 amostras no modo *SINGLE*, a CPU atingiu 450 ms, enquanto TFLite CPU e NNAPI ficaram em aproximadamente 1,90 ms e o GPU em cerca de 2,04 ms.

No modo *x12*, 65 k passou de aproximadamente 5 401 ms (CPU) para cerca de 22 ms nos três delegados.

As Figuras 4.5 e 4.6 consolidam os resultados de MAD no Moto G84 5G.

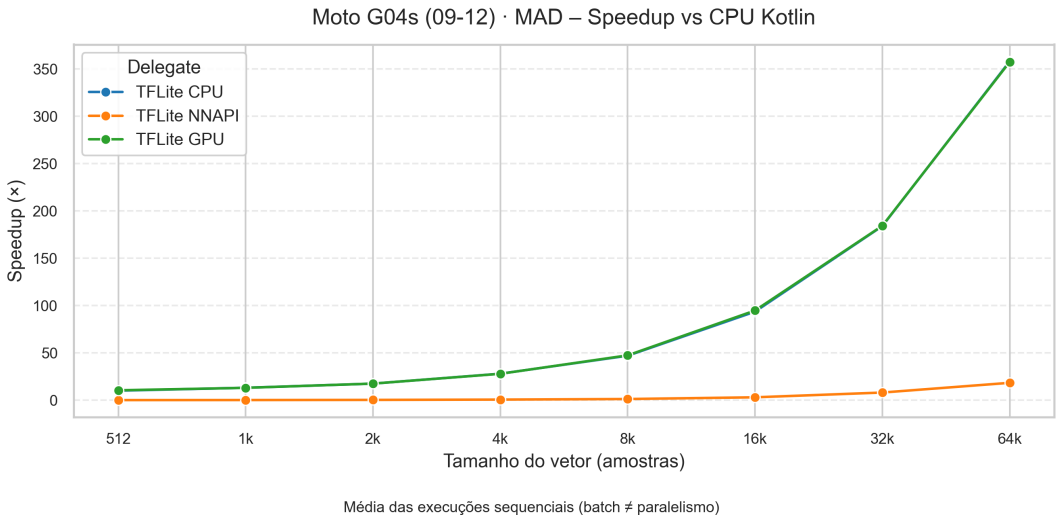

Figura 4.4 – *Speedup* do MAD no Moto G04s em relação ao baseline CPU Kotlin, para escalas representativas, nos modos *SINGLE* e *x12*, por delegado TFLite (baseado em `compute_ms`).

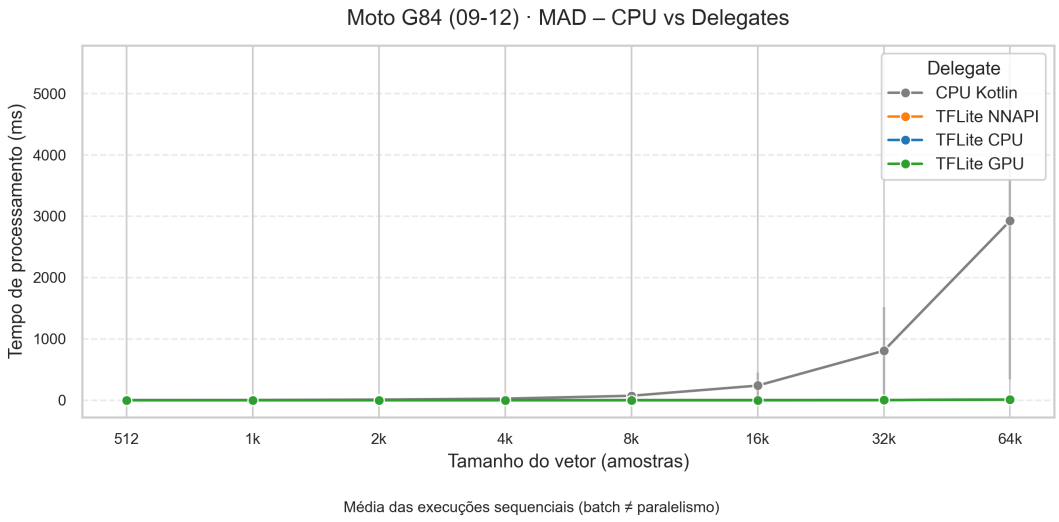

Figura 4.5 – Tempo de execução do MAD no Moto G84 5G para escalas representativas, nos modos *SINGLE* e *x12*, comparando CPU Kotlin e delegados TFLite (CPU, GPU, NNAPI), reportado em `compute_ms`.

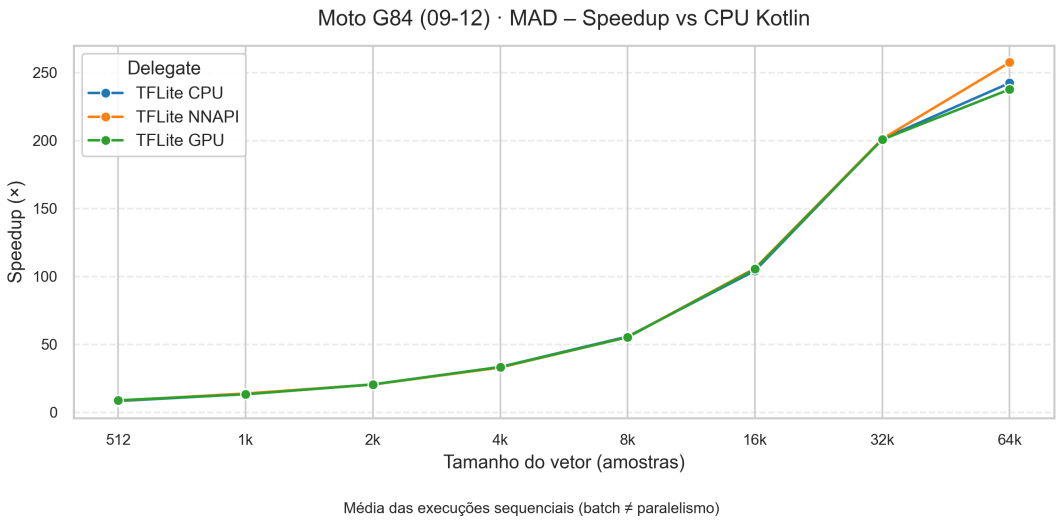

Figura 4.6 – *Speedup* do MAD no Moto G84 5G em relação ao baseline CPU Kotlin, para escalas representativas, nos modos *SINGLE* e *x12*, por delegado TFLite (baseado em `compute_ms`).

4.2.4 Tempo total de processamento por delegate (*offloading* + computação)

Para contextualizar a diferença prática entre delegados, foram mantidos gráficos que apresentam o **tempo total de processamento** observado em cada *backend* ao executar o pipeline, agregando custos de movimentação de dados, sincronizações e computação (i.e., *offloading* + cálculo). Reforça-se que, nesta instrumentação, o Android não fornece a separação dessas parcelas; portanto, os gráficos a seguir devem ser interpretados como o custo agregado do pipeline por delegate. As Figuras 4.7–4.9 apresentam esse tempo total para o MAD em cada dispositivo, comparando TFLite CPU, TFLite GPU e NNAPI.

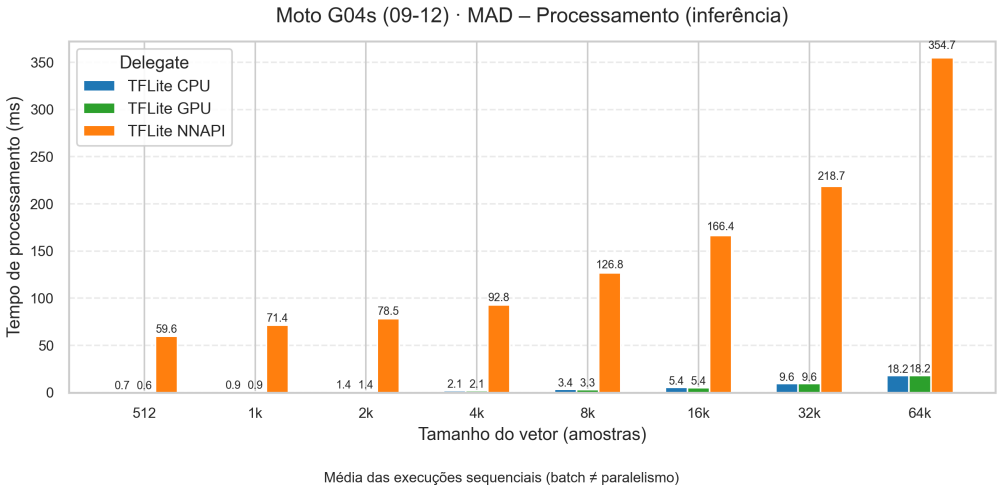

Figura 4.7 – Tempo total de processamento do pipeline de MAD no Moto G04s (offloading + computação), por delegate (TFLite CPU, TFLite GPU e NNAPI).

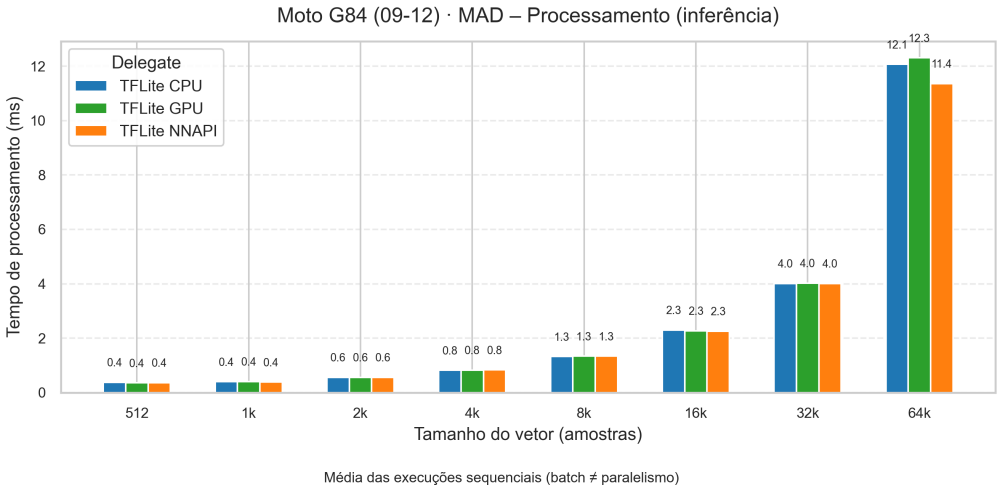

Figura 4.8 – Tempo total de processamento do pipeline de MAD no Moto G84 (offloading + computação), por delegate (TFLite CPU, TFLite GPU e NNAPI).

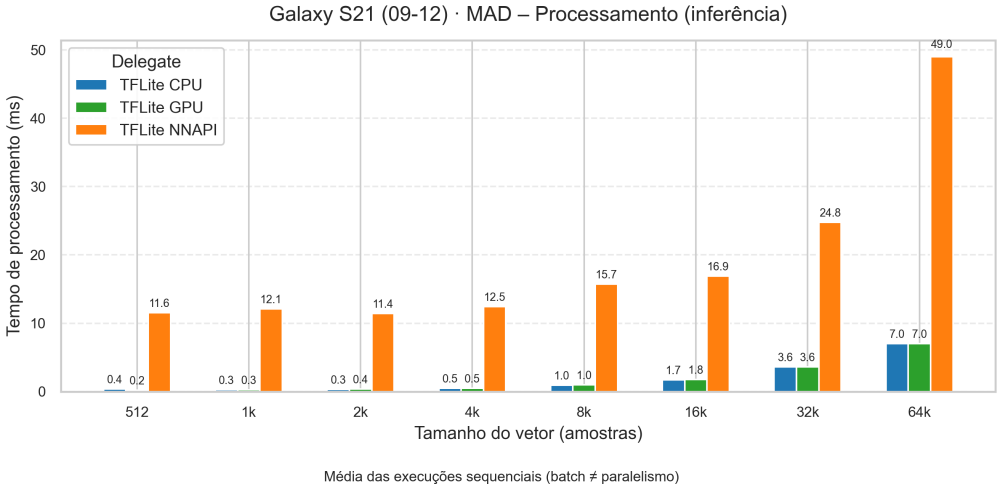

Figura 4.9 – Tempo total de processamento do pipeline de MAD no Galaxy S21 (offloading + computação), por delegate (TFLite CPU, TFLite GPU e NNAPI).

4.3 Resultados de FFT

Como discutido na Metodologia, a versão atual do TensorFlow Lite utilizada neste trabalho não disponibiliza *kernels* FFT dedicados para GPU ou NNAPI; assim, os delegados fazem *fallback* para o backend XNNPACK em CPU. Ao considerar `compute_ms`, observa-se que o TFLite CPU tende a ser a opção mais consistente, enquanto GPU e NNAPI podem introduzir *overhead* adicional, sobretudo em escalas menores, mesmo que o núcleo de computação permaneça na CPU.

4.3.1 Galaxy S21

No Galaxy S21, em 4 096 amostras no modo *SINGLE*, a CPU Kotlin ficou em 3,99 ms e o TFLite CPU em 0,93 ms, enquanto GPU e NNAPI foram mais lentos (6,21 ms e 7,10 ms). Em 65 536 amostras no modo *SINGLE*, a CPU chegou a 77,7 ms, o TFLite CPU a 19,4 ms, o GPU a 31,4 ms e o NNAPI a 63,2 ms. No modo *x12* para 65 k, a CPU ficou em aproximadamente 953 ms contra 213 ms no TFLite CPU.

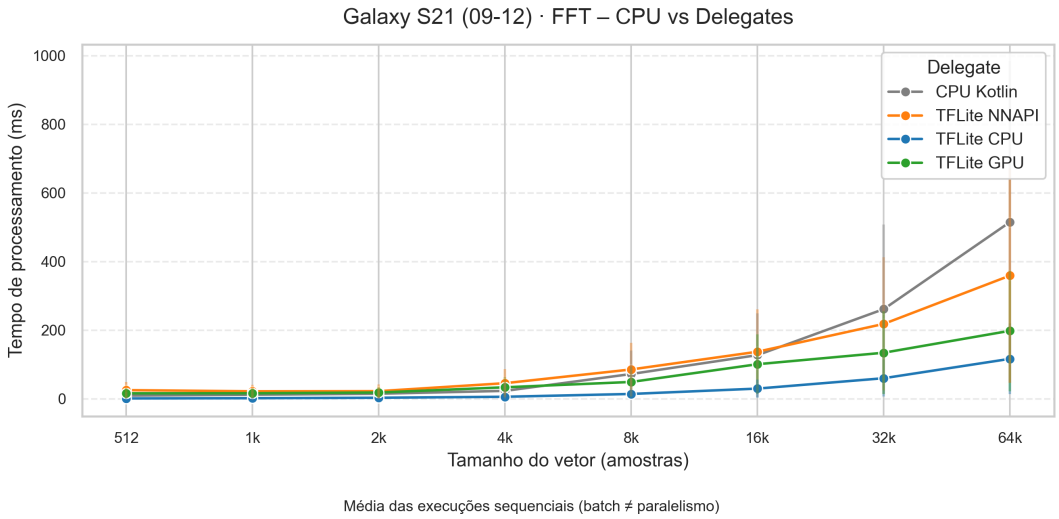

Figura 4.10 – Tempo de execução da FFT no Galaxy S21 para escalas representativas, nos modos *SINGLE* e *x12*, comparando CPU Kotlin e delegados TFLite (CPU, GPU, NNAPI), reportado em `compute_ms`.

4.3.2 Moto G04s

No Moto G04s, em 4 096 amostras no modo *SINGLE*, a CPU Kotlin ficou em 13,6 ms e o TFLite CPU em 3,00 ms, enquanto GPU (18,2 ms) e NNAPI (13,1 ms) não superaram o baseline. Em 65 536 amostras no modo *SINGLE*, a CPU chegou a 196 ms e o TFLite CPU a 45,6 ms, contra 95,0 ms no GPU e 91,7 ms no NNAPI. No modo *x12* para 65 k, a CPU ficou em aproximadamente 2 344 ms, e o TFLite CPU reduziu para cerca de 534 ms.

4.3.3 Moto G84 5G

No Moto G84, em 4 096 amostras no modo *SINGLE*, a CPU Kotlin ficou em 7,40 ms, o TFLite CPU em 1,29 ms e o NNAPI em 1,27 ms; o GPU ficou em 3,90 ms. Em 65 536 amostras no modo *SINGLE*, a CPU atingiu 174 ms, enquanto TFLite CPU/NNAPI ficaram em torno de 26–27 ms e o GPU em 43 ms. No modo *x12* para 65 k, a CPU ficou em aproximadamente 2 266 ms contra cerca de 309 ms no TFLite CPU e 307 ms no NNAPI.

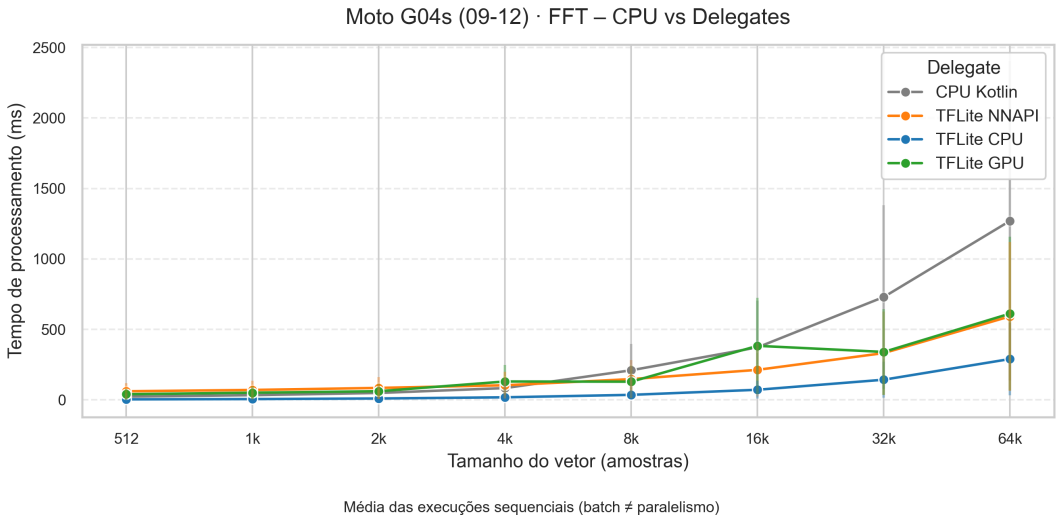

Figura 4.11 – Tempo de execução da FFT no Moto G04s para escalas representativas, nos modos *SINGLE* e *x12*, comparando CPU Kotlin e delegados TFLite (CPU, GPU, NNAPI), reportado em `compute_ms`.

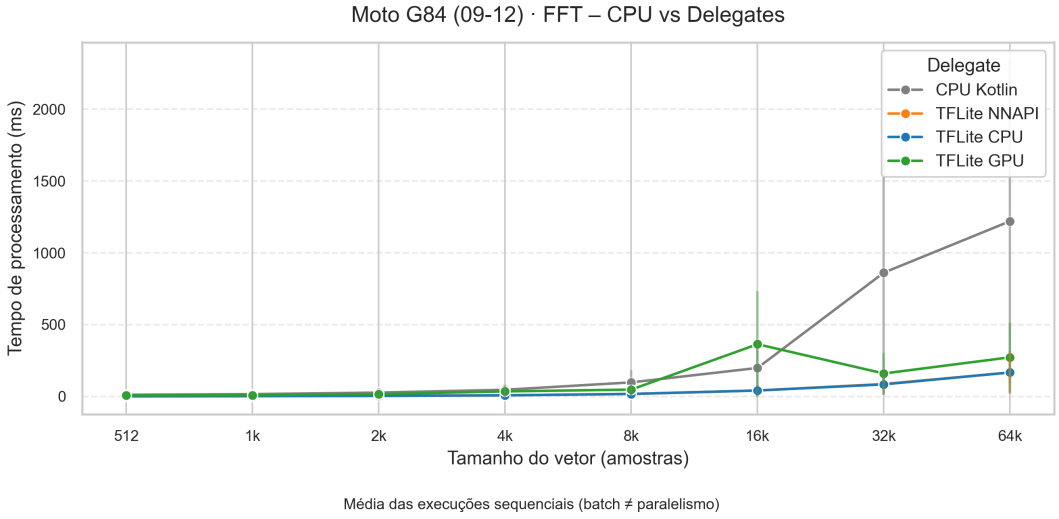

Figura 4.12 – Tempo de execução da FFT no Moto G84 5G para escalas representativas, nos modos *SINGLE* e *x12*, comparando CPU Kotlin e delegados TFLite (CPU, GPU, NNAPI), reportado em `compute_ms`.

4.3.4 Tempo total de processamento por delegate (offloading + computação)

De forma análoga ao MAD, foram mantidos gráficos que mostram o **tempo total de processamento** da FFT observado em cada delegate, englobando custos de coordenação, cópias e sincronizações além do cálculo (offloading + computação). As Figuras 4.13–4.15 apresentam esse tempo total para a FFT em cada dispositivo, comparando TFLite CPU, TFLite GPU e NNAPI. Em FFT, essa visualização é especialmente útil para contextualizar por que alguns delegates podem empatar (ou até piorar) em escalas menores: embora o cálculo principal utilize *fallback* para XNNPACK em CPU, o custo agregado do pipeline varia conforme o backend.

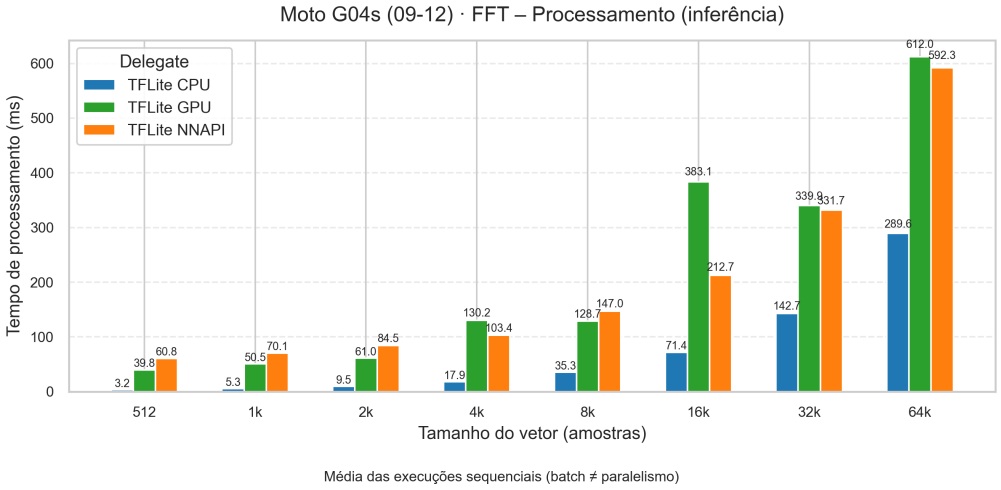

Figura 4.13 – Tempo total de processamento do pipeline de FFT no Moto G04s (offloading + computação), por delegate (TFLite CPU, TFLite GPU e NNAPI).

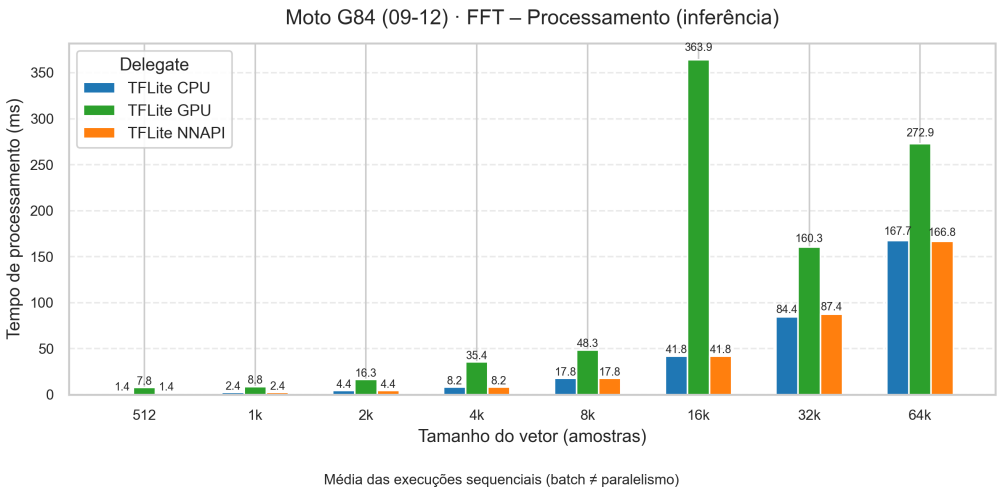

Figura 4.14 – Tempo total de processamento do pipeline de FFT no Moto G84 (offloading + computação), por delegate (TFLite CPU, TFLite GPU e NNAPI).

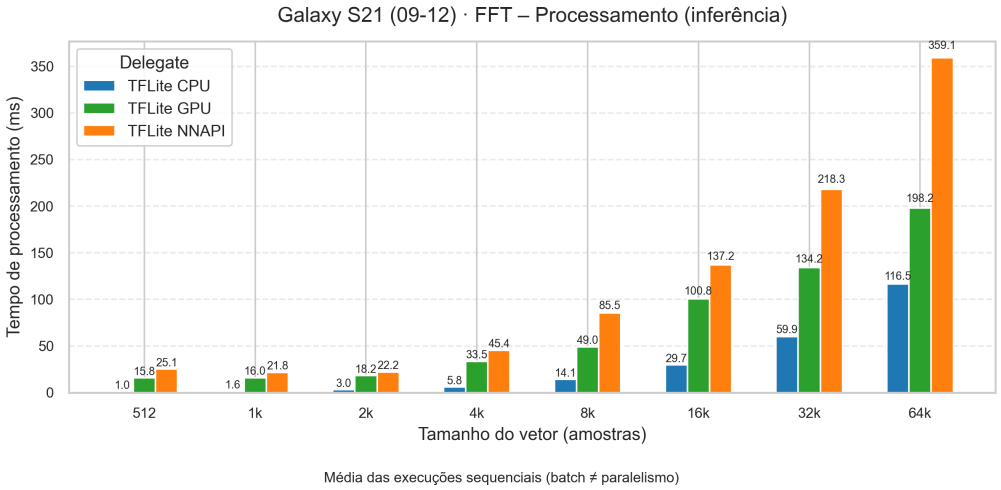

Figura 4.15 – Tempo total de processamento do pipeline de FFT no Galaxy S21 (offloading + computação), por delegate (TFLite CPU, TFLite GPU e NNAPI).

4.4 Resultados energéticos

Nesta campanha não foi executado o ciclo dedicado de 100 execuções do "teste energético" descrito na Metodologia. As etiquetas de consumo foram derivadas diretamente das leituras que o aplicativo registra junto a cada *benchmark* (12 execuções por cenário), incluindo:

- nível de bateria (em %);
- `energy_counter` (energia acumulada em nWh, quando disponível);
- `charge_counter` (carga em mAh);
- temperaturas inicial e final;
- estado do modo de economia de energia.

Esses registros estruturados alimentam diretamente as classificações presentes nos arquivos CSV e nos gráficos consolidados. Para cada combinação {dispositivo, delegado}, calcula-se a variação relativa de energia e de nível de bateria ao longo do conjunto de execuções, e a partir desse *delta* é atribuído um rótulo qualitativo de demanda energética, armazenado no campo `estimated_energy`.

Os rótulos seguem a seguinte interpretação:

- **Baixa**: menor queda relativa de energia entre os delegados naquele dispositivo;
- **Média**: queda intermediária, entre os extremos observados;
- **Alta**: maior queda relativa de energia para o mesmo cenário de teste.

Na prática, o ramo em CPU Kotlin tende a ser classificado como demanda **“Alta”**, o TFLite CPU como **“Média”**, o TFLite GPU como **“Baixa”** e o NNAPI como **“Baixa/Média”**, com pequenas variações entre aparelhos. Ou seja, as etiquetas não representam medições absolutas em mWh, mas um índice qualitativo comparando as estratégias de execução sob a mesma carga de trabalho e nas mesmas condições experimentais.

Mesmo com essa coleta enxuta, as leituras evidenciaram que delegar reduz o tempo ativo da CPU, o que resulta em menores quedas de bateria e temperaturas mais estáveis em sessões prolongadas. Esses efeitos permanecem coerentes com a análise baseada em `compute_ms`, embora a energia dependa também do custo total do pipeline (incluindo parcelas não isoladas pela plataforma).

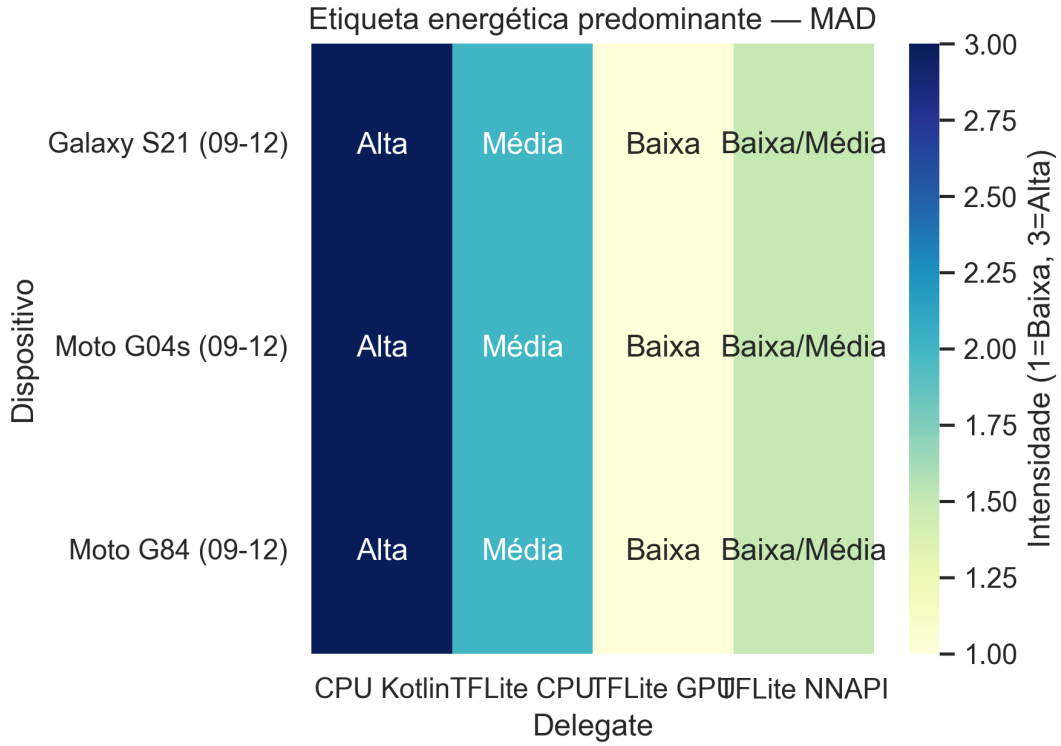

Figura 4.16 – Índice qualitativo de demanda energética para MAD, a partir do rótulo `estimated_energy`, comparando CPU Kotlin, TFLite CPU, TFLite GPU e TFLite NNAPI nos três dispositivos avaliados.

Os resultados indicam que, ao processar a mesma carga de trabalho em menos tempo e com menor atividade de CPU, os delegados tendem a reduzir o consumo relativo em comparação à implementação nativa em Kotlin. Ainda assim, é importante frisar que:

- os valores derivam exclusivamente de contadores internos do sistema operacional;
- não foi utilizada instrumentação externa (como medidores dedicados de corrente);

- variações muito pequenas de temperatura não foram consideradas em profundidade na análise.

Assim, os rótulos energéticos devem ser entendidos como um indicador qualitativo de ordem de grandeza e não como uma medição absoluta em mWh.

4.5 Precisão observada

Além dos tempos de execução e métricas energéticas, foi avaliada a consistência numérica dos resultados de FFT entre os diferentes delegados TFLite. Para isso, o mesmo teste `TfliteDelegatesInstrumentedTest`, usado como *gate* de regressão de desempenho, foi estendido com uma nova suíte de testes chamada `fft_precision_statistics_suite`.

O comando abaixo exemplifica a execução dessa rotina:

```
./gradlew :app:connectedAndroidTest \
  -Pandroid.testInstrumentationRunnerArguments.precisionRepeats=5
```

O teste percorre cada comprimento de 512 a 65 536 pontos cinco vezes, registra as diferenças em `precision_fft.csv` e imprime linhas de log no formato:

```
FFT_PRECISION_STATS: len=65536 (5x) | maxRel avg=11.104% std=12.779%
 min=3.605% max=36.572% | meanRel avg=0.007% | rmse avg=2.004928
```

As escalas até 16 k apresentaram médias relativas inferiores a 3% e desvios baixos; em 32 k e 65 k surgiram picos ocasionais (resposta direta da magnitude dos *bins* e do uso de FP32 em CPU), mas a média relativa permaneceu da ordem de 0,007%. Comprimentos acima de 65 k não foram incluídos porque os dispositivos não completaram as execuções por falta de memória.

Esses registros ajudam a explicar por que diferentes delegados exibem pequenas variações mesmo usando o mesmo arquivo `.tflite`: GPU e NNAPI podem forçar FP16 no pipeline interno, alguns *bins* passam a utilizar erro absoluto (um *guard rail* de 10^3) e, sobretudo, todo o grafo roda no CPU via XNNPACK, de modo que qualquer oscilação deriva de cópias e arredondamentos em cada backend. Apesar disso, os erros médios se mantêm baixos o suficiente para que as diferenças não afetem as conclusões de desempenho e nem comprometam a utilidade prática dos *pipelines* de FFT em cenários de computação em borda.

4.6 Discussão geral

Os resultados demonstram forte dependência do hardware-alvo e do delegate escolhido. Para MAD, TFLite CPU e GPU entregam ganhos muito altos em `compute_ms` em todas as

plataformas, enquanto o NNAPI apresenta variabilidade — chegando a ser mais lento que a CPU em escalas pequenas no Moto G04s, mas permanecendo competitivo em janelas grandes. Para FFT, o TFLite CPU é o mais consistente; GPU e NNAPI frequentemente adicionam *overhead* por causa do *fallback* para XNNPACK, o que explica resultados piores em escalas pequenas. O *batching* de 12 pacotes continua relevante, porém seu benefício aparece com mais clareza quando se compara `compute_ms` (núcleo do cálculo) com o tempo total de processamento observado por delegate (*offloading* + computação).

Embora este estudo tenha focado principalmente em métricas de desempenho, como tempo de execução e *speedup*, aspectos relacionados ao consumo energético e ao comportamento térmico também são relevantes em dispositivos móveis. Em particular, a utilização de aceleradores como GPU e NNAPI pode reduzir o tempo de execução de determinadas operações, mas também pode alterar o perfil de consumo energético do sistema, dependendo do padrão de uso e do custo de transferência de dados entre CPU e aceleradores. Trabalhos anteriores indicam que o desempenho observado nem sempre se traduz diretamente em maior eficiência energética em plataformas móveis heterogêneas (TZENG et al., 2012; KIM et al., 2021). A análise detalhada de consumo de energia e temperatura fica como uma direção de investigação futura deste trabalho.

5 Considerações Finais

Este capítulo apresenta uma síntese dos principais resultados obtidos ao longo do trabalho, bem como uma discussão sobre suas implicações no contexto de execução de pipelines de processamento de sinais em dispositivos móveis. Inicialmente, é realizada uma síntese dos resultados experimentais observados nos diferentes dispositivos e estratégias de execução avaliadas. Em seguida, são discutidas as limitações do estudo e possíveis direções para trabalhos futuros.

5.1 Síntese dos resultados

Este trabalho avaliou, de forma experimental, pipelines de *Mean Absolute Deviation* (MAD) e *Fast Fourier Transform* (FFT) em dispositivos Android reais, confrontando implementações em CPU Kotlin com delegates do TensorFlow Lite (CPU, GPU e NNAPI). Foram consideradas oito escalas de dados (512 $\rightarrow$ 65 536 amostras) e duas configurações de lote (modo *SINGLE* e modo *x12*). A suíte oficial executou cada botão cinco vezes consecutivas, com 12 iterações internas por cenário, produzindo 640 medições por dispositivo (e mais 176 linhas extras no Moto G84 devido às tentativas experimentais acima de 64 k, que falharam por falta de memória). Como o Android (e as bibliotecas utilizadas) não expõe métricas isoladas que separem, de forma padronizada e confiável, o custo de *offloading* do custo estrito de computação, esta versão da análise prioriza o tempo de computação (`compute_ms`); métricas auxiliares de tempo refletem majoritariamente custos de geração e empacotamento dos dados e não foram tratadas como decomposição precisa de execução.

De maneira geral, os resultados mostraram que:

- **MAD:** os ganhos em `compute_ms` foram de **dezenas a centenas de vezes** em todas as plataformas, com destaque para os delegates **TFLite CPU** e **TFLite GPU**. O NNAPI apresentou **variabilidade** e chegou a perder para a CPU em escalas pequenas no Moto G04s, reforçando que a escolha do delegate depende tanto do dispositivo quanto do regime de granularidade.
- **FFT:** o **TFLite CPU** foi a opção mais consistente. Em contrapartida, **GPU** e **NNAPI** frequentemente adicionaram *overhead* por conta do *fallback* para o backend XNNPACK (isto é, o núcleo da FFT continuou sendo executado em CPU), chegando a resultados piores que o baseline em escalas menores. Assim, mesmo compartilhando o mesmo arquivo `.tflite`, o comportamento efetivo de FFT variou por delegate e por faixa de escala, e o ganho real observado concentrou-se no caminho TFLite CPU.

- **Granularidade e batching:** o modo *x12* manteve boa eficiência por janela e reduziu variância nas medições. Embora o maior impacto do batching se manifeste quando se considera o custo total do pipeline (preparação + computação), a análise por `compute_ms` já indica que concentrar trabalho por chamada é um requisito prático para extrair ganhos consistentes ao delegar o processamento.
- **Energia e comportamento térmico:** mesmo com limitações de instrumentação, as leituras internas do sistema mostraram tendência de menor demanda energética relativa e maior estabilidade térmica quando o pipeline reduz o tempo ativo de CPU. Mesmo no caso da FFT, em que há *fallback* para CPU, o uso do *interpreter* TFLite mitigou picos e favoreceu um perfil mais estável nas três plataformas avaliadas.

No conjunto, os achados reforçam que delegar o pipeline de MAD para o TensorFlow Lite é eficaz e consistente quando a granularidade do processamento é adequada, enquanto, para FFT, o estado atual do ecossistema (ausência de kernels dedicados e dependência de *fallback*) limita o potencial de aceleração por GPU/NPU, tornando o TFLite CPU o caminho dominante em desempenho.

De forma mais ampla, os resultados indicam que a decisão de delegar computações em Android não deve ser guiada apenas por latência pontual: isso é um problema de otimização multiobjetivo que envolve granularidade, custo de preparação de dados, consumo energético e comportamento térmico. Nesse sentido, os delegates do TensorFlow Lite mostraram-se mais vantajosos quando o pipeline concentra trabalho suficiente por chamada, amortizando custos fixos e reduzindo o tempo ativo do sistema.

5.2 Limitações do estudo

Apesar dos resultados, é importante reconhecer limitações do estudo:

- Os experimentos foram realizados em apenas três smartphones, o que restringe a generalização para outros SoCs, combinações de CPU/GPU/NPU e versões de sistema operacional.
- O escopo focou em duas operações de referência (MAD e FFT), não abrangendo redes neurais completas ou pipelines híbridos mais longos, embora essas operações apareçam como blocos em diversas arquiteturas.
- Os sinais utilizados foram gerados sinteticamente a partir de um processo inspirado em acelerômetro; apesar de representarem padrões plausíveis, não substituem a variabilidade de dados coletados em campo.

- As métricas energéticas derivam de APIs e sensores internos do sistema, fornecendo estimativas consistentes, porém não equivalentes a medições com instrumentação externa dedicada (e.g., monitores de corrente).
- O gargalo estrutural associado à preparação de dados e cópias internas não foi atacado com técnicas mais agressivas, como *zero-copy*, buffers persistentes, fusão de operadores ou reorganização mais profunda do grafo.

5.3 Trabalhos futuros

Como continuidade, recomenda-se:

- Expandir os experimentos para outras operações relevantes em computação em borda, incluindo convoluções, filtros digitais, espectrogramas e modelos completos em TensorFlow Lite, aproximando os cenários de aplicações reais.
- Comparar a FFT do TFLite com bibliotecas especializadas, incluindo implementações otimizadas para GPU/Vulkan (por exemplo, VkFFT), avaliando desempenho, estabilidade e consumo energético.
- Ampliar o conjunto de dispositivos, contemplando SoCs com NPUs mais recentes, perfis energéticos distintos e versões recentes do Android, para observar como drivers, firmware e runtime impactam o desempenho.
- Integrar o pipeline a fluxos reais de sensores e sessões prolongadas, analisando execução contínua sob variação térmica e condições de uso normal.
- Investir na redução do custo de preparação e movimentação de dados, por meio de buffers persistentes, redução de cópias e reorganização do pipeline para concentrar computação por chamada.

Conclusão geral

Em síntese, este TCC mostrou que delegar operações de MAD e FFT para o TensorFlow Lite em dispositivos Android é uma estratégia viável para reduzir latência e carga na CPU, principalmente quando se trabalha com granularidade adequada e janelas processadas em lote. Contudo, os resultados também deixam claro que a efetividade do delegate depende do tipo de operação e do suporte real do runtime: no MAD, os ganhos em `compute_ms` foram amplos e consistentes; na FFT, o ganho mais confiável concentrou-se no TFLite CPU, enquanto GPU e NNAPI apresentaram *overhead* devido ao *fallback* para CPU.

Assim, mais do que “ativar aceleradores”, o que define o ganho real é o alinhamento entre algoritmo, granularidade e arquitetura de execução do pipeline. Em última instância, este trabalho reforça que a eficiência em computação em borda em Android emerge de decisões de projeto conscientes e mensuráveis, e não automaticamente do uso de GPU/NPU.

Referências

ABADI, M.; BARHAM, P.; CHEN, J.; CHEN, Z.; DAVIS, A.; DEAN, J.; DEVIN, M.; GHEMAWAT, S.; IRVING, G.; ISARD, M.; KUDLUR, M.; LEVENBERG, J.; MONGA, R.; MOORE, S.; MURRAY, D. G.; STEINER, B.; TUCKER, P.; VASUDEVAN, V.; WARDEN, P.; WICKE, M.; YU, Y.; ZHENG, X. Tensorflow: A system for large-scale machine learning. *arXiv preprint arXiv:1605.08695*, 2016. Disponível em: <https://arxiv.org/abs/1605.08695>.

AYACHI, F.; BOUGUILA, N.; AYED, Y. B. Statistical feature extraction for wearable sensor data: Efficiency and energy considerations. *Sensors*, v. 22, n. 9, p. 3412, 2022. Disponível em: <https://www.mdpi.com/1424-8220/22/9/3412>.

HASHEMI, S. M.; FATEMI, O.; BRETSCHNEIDER, T. Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android. In: *Proceedings of the 14th IEEE International Conference on Machine Learning and Applications (ICMLA)*. [s.n.], 2015. p. 847–852. Disponível em: <https://arxiv.org/abs/1511.07376>.

HUANG, K.; YIN, X.; GU, T.; GAO, W. Perceptual-centric image super-resolution using heterogeneous processors on mobile devices. In: *Proc. 30th Annual Int'l Conf. on Mobile Computing and Networking (MobiCom)*. [s.n.], 2024. p. 1361–1376. Disponível em: <https://dl.acm.org/doi/10.1145/3636534.3690698>.

IGNATOV, A.; PEREVOZCHIKOV, G.; TIMOFTE, R. et al. Quantized image super-resolution on mobile npus, mobile ai 2025 challenge: Report. In: *Proc. IEEE/CVF Conf. on Computer Vision and Pattern Recognition Workshops (CVPRW)*. [s.n.], 2025. p. 1908–1921. Disponível em: <https://openaccess.thecvf.com/content/CVPR2025W/MAI/papers/Ignatov_Quantized_Image_Super-Resolution_on_Mobile_NPUs_Mobile_AI_2025_Challenge_CVPRW_2025_paper.pdf>.

Ignatov, Andrey et al. *AI Benchmark: Running Deep Neural Networks on Android Smartphones*. 2018. <https://arxiv.org/abs/1810.01109>. Accessed: 2025-08-16.

KIM, J.; LEE, Y.; KANG, H.; HAN, S.; YOO, H. Mobile socs: Ai tax - evaluating the performance and cost of on-device intelligence. In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. [s.n.], 2021. p. 45–56. Disponível em: <https://projects.iq.harvard.edu/files/edge/files/mobile_soc_ispass_2021.pdf>.

LI, Q.; ZUO, D.; FENG, Y.; WEN, D. Research on high-performance fourier transform algorithms based on the npu. *Applied Sciences*, v. 14, n. 1, p. 405, 2024. Disponível em: <https://www.mdpi.com/2076-3417/14/1/405>.

MITTAL, S. A survey of techniques for improving energy efficiency in embedded computing systems. *ACM Computing Surveys*, v. 47, n. 2, p. 1–34, 2015. Disponível em: <https://www.researchgate.net/publication/255485741_A_survey_of_techniques_for_improving_energy_efficiency_in_embedded_computing_systems>.

MLCommons Association. *MLPerf Mobile Inference Benchmark v2.0*. 2022. <https://mlcommons.org/en/news/mlperf-mobile-v2-0/>. Accessed: 2025-08-16.

OWENS, J. D.; HOUSTON, M.; LUEBKE, D.; GREEN, S.; STONE, J. E.; PHILLIPS, J. C. Gpu computing. *Proceedings of the IEEE*, v. 96, n. 5, p. 879–899, 2008. Disponível em: <https://escholarship.org/uc/item/0cv1p1nc>.

RATHORE, S.; SAHA, R.; SHARMA, B.; POOJARY, P.; PATEL, H. Cpu vs. gpu: Performance comparison of opencl applications on a heterogeneous architecture. In: *2024 IEEE International Conference on High Performance Computing (HiPC)*. [s.n.], 2024. p. 1–10. Accessed: 2025-05-22. Disponível em: <https://www.researchgate.net/publication/385679921_CPU_vs_GPU_Performance_comparison_of_OpenCL_Applications_on_a_Heterogeneous_Architecture>.

SATYANARAYANAN, M. The emergence of edge computing. *Computer*, IEEE, v. 50, n. 1, p. 30–39, 2017. Disponível em: <https://www.researchgate.net/publication/312109957_The_Emergence_of_Edge_Computing>.

SHI, W.; CAO, J.; ZHANG, Q.; LI, Y.; XU, L. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, v. 3, n. 5, p. 637–646, 2016. Disponível em: <https://cse.buffalo.edu/faculty/tkosar/cse710_spring20/shi-iot16.pdf>.

TensorFlow Contributors. *Issues and Feature Requests — TensorFlow Lite GitHub*. 2024. <https://github.com/tensorflow/tensorflow/issues?q=is%3Aissue+tflite+fft>. Accessed: 2025-08-16.

TensorFlow Team. *TensorFlow Lite Now Faster with Mobile GPUs*. 2019. <https://blog.tensorflow.org/2019/01/tensorflow-lite-now-faster-with-mobile.html>. Accessed: 2025-05-22.

TensorFlow Team. *Performance with Delegates*. 2024. <https://www.tensorflow.org/lite/performance/delegates>. Accessed: 2025-08-16.

TOLMACHEV, D. *VkFFT: Vulkan/CUDA/HIP/OpenCL/Level Zero/Metal Fast Fourier Transform library*. 2021. <https://github.com/DTolm/VkFFT>. Accessed: 2025-05-22.

TZENG, S.; WEI, S.-C.; CHEN, Y.-C.; CHANG, Y.-C.; HSU, C.-H.; CHEN, L.-G. Energy and performance characterization of mobile heterogeneous computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 20, n. 12, p. 2226–2239, 2012.

VASILAKIS, C.; TSAGKAROPOULOS, A.; KOUTOULAS, I.; REISIS, D. Improving the fast fourier transform for space and edge computing applications with an efficient in-place method. *Software*, v. 4, n. 2, p. 11, 2025. Disponível em: <https://www.mdpi.com/2674-113X/4/2/11>.

VERMA, G.; GUPTA, Y.; MALIK, A. M.; CHAPMAN, B. Performance evaluation of deep learning compilers for edge inference. In: *Proc. 2021 IEEE Int'l Parallel Distributed Processing Symp. Workshops (IPDPSW)*. [s.n.], 2021. p. 858–865. Disponível em: <https://ieeexplore.ieee.org/document/9472197>.

VOLKOV, V.; DEMMEL, J. W. Benchmarking gpus to tune dense linear algebra. In: *Proceedings of the 2008 ACM/https://escholarship.org/uc/item/0cv1p1nc Conference on Supercomputing*. [s.n.], 2008. p. 1–11. Disponível em: <https://mc.stanford.edu/cgi-bin/images/6/65/SC08_Volkov_GPU.pdf>.

ZHANG, Y.; LI, X.; WANG, L.; CHEN, J. Efficient spectral analysis for edge computing: Fft optimization on resource-constrained devices. *IEEE Internet of Things Journal*, v. 8, n. 16, p. 12945–12957, 2021. Disponível em: <https://ieeexplore.ieee.org/document/9399476>.

ZHANG, Y.; ZHANG, B.; HU, X. L.; LI, M. Aienergy: An energy benchmark for ai-empowered mobile and iot devices. *IEEE Transactions on Mobile Computing*, v. 23, n. 2, p. 1234–1247, 2024. Disponível em: <https://www.researchgate.net/publication/393302600>.