UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

LUCAS GILBERTO KERN

Estudo da Eficiência Energética de Processadores Gráficos em Dispositivos Móveis

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Antonio Carlos Schneider Beck Filho

Porto Alegre, Dezembro de 2015

AGRADECIMENTOS

Agradeço ao meu orientador, Professor Antonio Carlos, pelo auxílio e aconselhamento durante a realização deste trabalho.

Agradeço, também, à minha mãe, Suzana, meu pai, Gilberto e à minha noiva, Francis, pela força e pelo encorajamento que recebi durante toda minha trajetória no curso e por sempre acreditarem em mim.

Por fim, agradeço a todos professores, funcionários e colegas da Universidade Federal do Rio Grande do Sul, em especial ao envolvidos nos cursos de Ciência da Computação e Engenharia da Computação, por criarem este maravilhoso ambiente no qual vivenciei grande parte dos últimos cinco anos.

SUMÁRIO

RESUMO

Este trabalho avalia o impacto do uso de processadores gráficos em sistemas embarcados. Processadores gráficos são sinônimos de alta performance em diversas aplicações. Desde jogos até simulações complexas, eles são utilizados para alcançar rapidamente algo que um processador de propósito geral demoraria dezenas de vezes mais para realizar. Para dispositivos móveis, que também os usam, isso não é diferente. Processadores gráficos, porém, também são conhecidos como grandes consumidores de energia, algo que não pode ser desprezado ao se trabalhar com uma plataforma móvel.

Desta maneira, este trabalho compara o consumo energético de aplicações, constituintes de um conjunto de benchmarks, desenvolvidas para dispositivos móveis ao utilizarem processadores gráficos para executarem suas principais rotinas. Para isto, cada um dos benchmarks em questão é executado em uma plataforma embarcada, havendo três versões distintas de cada um dos programas. A primeira versão, a versão base, é uma versão escrita na linguagem usual da plataforma, no caso, Java. A segunda versão, que usa a primeira como base, substitui os trechos mais executados da mesma por um código equivalente, escrito, porém, na linguagem nativa da plataforma, C++. A terceira, e mais importante para o trabalho, substitui os trechos mais executados da versão base por código a ser executado em um processador gráfico do dispositivo.

Para avaliar o impacto utilizamos duas métricas diferentes: desempenho e consumo energético. Comparamos as três versões de cada um dos benchmarks entre si, para encontrar as vantagens e desvantagens de cada uma das implementações. Por fim, analisamos e justificamos as variações de desempenho e consumo encontradas entre diferentes benchmarks e diferentes versões.

Palavras-chave: Processadores Gráficos. Sistemas Embarcados. Android. Consumo Energético. Benchmark.

Study of the Energy Efficiency of Graphics Processing Units in Mobile Devices

ABSTRACT

This work evaluates the impact of the usage of graphics processing units in embedded systems. Graphics processing units are usually linked to high performance systems for many different applications. From games to complex simulations, they are used to accomplish tasks in which a common central processing unit could take much longer to finish. In embedded systems, it's no different. However, graphics processing units aren't known for being energy-efficient, a major drawback that can't be ignored in a mobile platform.

With that in mind, this work compares the energy consumption from a set of benchmarks developed for a mobile platform when utilizing GPUs to run their main routines. In order to accomplish that, each of the benchmarks is executed in an embedded system, with three different implementations. The first implementation is written in the platform natural language, Java. The second, based on the first implementation, replaces the algorithm main routines for routines running in the platform native language, C++. The third and most important implementation replaces those main routines for routines executed in the device's GPU.

Two different metrics are used to evaluate the implementations: pure performance and energy consumption. We compared all three implementations of each benchmark among themselves, in order to find benefits and drawbacks from each of them. Finally, we analyzed the variations found in performance and power consumption between the implementations, trying to explain the reasons behind each of them.

Keywords: Graphic Processing Units. Embedded Systems. Android. Energy Consumption. Benchmark.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Interface de Programação de Aplicações
CPU	Unidade de Processamento Central (Central Processing Unit)
CUDA	Compute Unified Device Architecture
DFT	Transformada Discreta de Fourier (Discrete Fourier Transform)
DVM	Dalvik Virtual Machine
FFT	Transformada Rápida de Fourier (Fast Fourier Transform)
GPU	Unidade de Processamento Gráfico (Graphics Processing Unit)
JVM	Máquina Virtual Java da Oracle
JNI	Interface Nativa Java (Java Native Interface)
PC	Computador Pessoal
SIMD	Single Instruction, Multiple Data
SoC	System-on-Chip
UFRGS	Universidade Federal do Rio Grande do Sul

1 INTRODUÇÃO

Processadores gráficos são sinônimos de alta performance em diversas aplicações. Desde jogos até simulações complexas, eles são utilizados para alcançar rapidamente algo que um processador de propósito geral demoraria dezenas de vezes mais para realizar. Para dispositivos móveis, que também os usam, isso não é diferente.

Processadores gráficos, porém, também são conhecidos como grandes consumidores de energia, geralmente necessitando de recursos extras para alcançarem o ápice de seu desempenho. Estes recursos extras, que geralmente se refletem no aumento da potência disponível para o sistema, não podem ser facilmente alcançados em sistemas embarcados, que, em grande parte, contam com uma bateria de capacidade limitada.

Além do mais, o tempo útil de bateria de dispositivos móveis, principalmente aparelhos celulares, vem diminuindo ao longo dos anos, chegando a patamares em que os mesmos necessitam serem carregados constantemente. Com esse cenário, avaliar o impacto do uso de processadores gráficos nos mesmos é importantíssimo, pois possibilita haver mais uma visão de uma possível causa de problemas nesse âmbito.

A meta deste trabalho é, portanto, mostrar o comportamento do uso de processadores gráficos em diversos cenários, analisando as variações em performance e gasto energético dos programas ao se adicionar e remover esse uso. Temos o intuito de investigar o impacto do uso de processadores gráficos em sistemas embarcados, mostrando seu impacto em desempenho quanto em consumo energético.

De maneira geral, acredita-se que processadores gráficos sempre trarão um aumento na performance de aplicações, mas ao custo de um elevado consumo energético. Existem aplicações, porém, que o uso dos mesmos não deve beneficiá-las, devido à alta dependência entre dados, mostrando a ligação entre a possibilidade paralelização de dados e as vantagens do uso de processadores gráficos.

Para realizar a análise do consumo energético de processadores gráficos em dispositivos móveis, criamos um conjunto de benchmarks, produto de um subconjunto de benchmarks desenvolvido pela empresa SPEC. Este subconjunto foi adaptado para ser executado em um dispositivo móvel, uma vez que a versão original dos mesmos foi criada para analisar performances de computadores pessoais.

Com esse subconjunto em mãos, cada benchmark foi divido em três diferentes versões, contendo uma versão, a original, desenvolvida pela SPEC, com todo o código executado como um programa normal na plataforma; uma segunda versão, desenvolvida em Sartor et al. (2013),

com a maior parte do código executado de forma nativa na plataforma; e, por fim, criado para este estudo, uma terceira versão, com a maior parte do código executado em um processador gráfico.

Com a existência de três versões, analisamos, além do impacto da inclusão do uso do processador gráfico na aplicação, também o impacto do ambiente natural de execução do sistema, ambos em questões de performance e consumo energético. Feita essa comparação poderemos obter melhores conclusões do impacto da adição do uso do processador gráfico perante o melhor cenário de implementação puro do processador principal.

O trabalho está dividido em cinco capítulos principais. O primeiro capítulo, de fundamentação teórica, mostra informações e conhecimentos necessários para se compreender os capítulos seguintes. Após isso temos o capítulo de trabalhos relacionados, que mostra outras pesquisas realizadas na área de processadores gráficos e na área de dispositivos móveis. Então temos o capítulo de implementação, que mostra detalhes do processo de implementação da pesquisa, explicando como cada uma das medições foi realizada, além de ressaltar dados importantes para a análise de resultados. O capítulo de resultados mostra graficamente e analisa os dados obtidos através do processo de implementação da pesquisa, tanto na área de performance, quanto na área de consumo energético. Como último capítulo, temos o capítulo de conclusões e trabalhos futuros mostra conclusões retiradas a partir da análise de resultados, além de mostrar futuros passos para se continuar a pesquisa mais profundamente. Por fim, mostramos as fontes das referências utilizadas, tanto como base de pesquisa, quanto mencionadas ao longo dela.

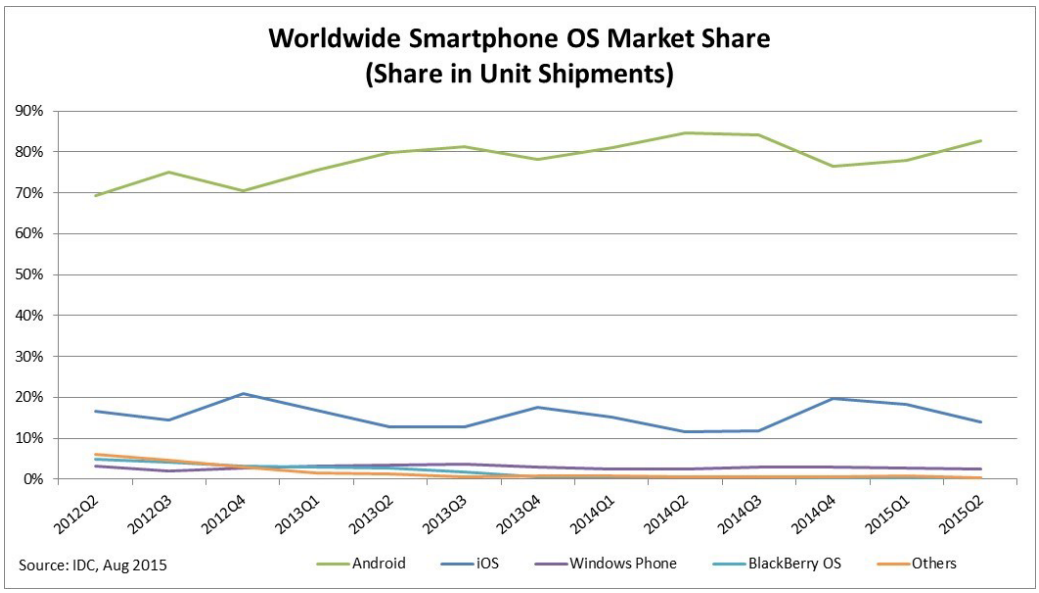
2 FUNDAMENTAÇÃO TEÓRICA

2.1 Android

Android é um sistema operacional com código aberto para dispositivos móveis (OPEN HANDSET ALLIANCE, 2015), desenvolvido pelo consórcio OHA (Open Handset Alliance), liderado pela Google e composto por companhias móveis, companhias de fabricação de semicondutores, companhias de desenvolvimento de software e companhias comerciais. Entre elas temos a própria Google, ARM, Intel, Samsung, HTC, Motorola, Qualcomm, Nvidia e diversas outras (OPEN HANDSET ALLIANCE, 2015).

Atualmente, cerca de 80% dos smartphones do mundo (INTERNATIONAL DATA CORPORATION, 2015) utiliza alguma versão do sistema Android como sistema operacional, como podemos ver na figura 2.1.

Figura 2.1 – Divisão do mercado de smartphones mundial



Fonte: IDC: Smartphone OS Marketshare 2015, 2014, 2013, and 2012[1]

2.1.1 Pilha Android[1]

A Figura 2.2 mostra detalhes da pilha Android. A pilha toda se baseia em um núcleo Linux – camada (0) da figura – que interage com o hardware. Esse núcleo contém todos drivers

[1] Disponível em: <http://www.idc.com/prodserv/smartphone-os-market-share.jsp>; Acesso em nov. 2015

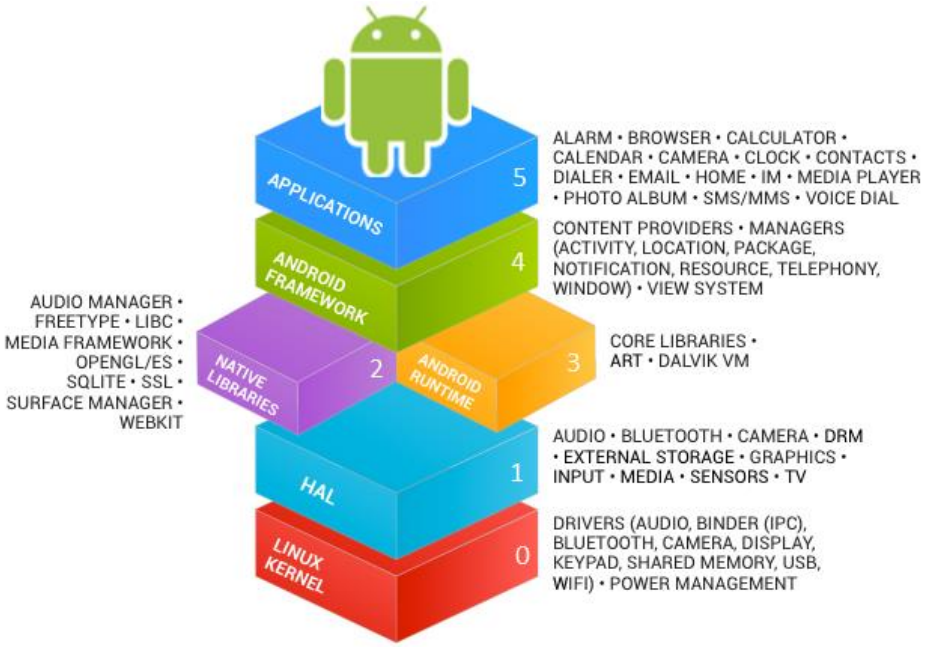
essenciais para o funcionamento do dispositivo. Em suas primeiras versões, até a versão Ice Cream Sandwich, esse núcleo Linux era baseado no núcleo da versão 2.6 do Linux. Versões mais recentes do Android são baseadas no núcleo da versão 3.x do Linux, com algumas alterações específicas no mesmo feitas pela Google. As bibliotecas que interagem com o núcleo, contidas na camada (1), consistem em bibliotecas nativas, escritas em C ou C++. Essas bibliotecas são utilizadas e otimizadas principalmente para tarefas que requerem uso intenso da CPU ou da GPU.

2.1.2 Máquinas Virtuais Java

Para o desenvolvimento de aplicativos Android, normalmente se é utilizado a linguagem Java, que possui suporte natural, fornecido pela própria pilha Android na camada (3) da mesma. Porém, existe uma diferença do Java que é executado em um dispositivo Android para o que é executado, por exemplo, em um computador pessoal. Ao oferecer suporte ao Java na pilha Android, a Google decidiu criar sua própria versão da Máquina Virtual Java (JVM) da Oracle, a Máquina Virtual Dalvik (DVM). O propósito de se existir uma máquina virtual para execução de código é oferecer uma experiência independente de plataforma para o desenvolvedor de

aplicativos, abstraindo detalhes de hardware e sistema operacional através da máquina virtual, aumentando significativamente a portabilidade de aplicações desenvolvidas para a mesma.

Figura 2.2 – A pilha Android



Fonte: The Android Source Code | Android Source Code Project[2].

A Máquina Virtual Dalvik (Dalvik), que aparece na camada (3) da pilha Android, possui uma arquitetura baseada em registradores (OPEN HANDSET ALLIANCE, 2015), diferente da Máquina Virtual Java comum, que, por sua vez, possui uma arquitetura baseada em pilha. A Dalvik foi desenvolvida com o propósito de ser executada em sistemas com quantidades bastante reduzidas de recursos, principalmente memória, além se suportar a criação de várias instâncias da mesma. Assim, cada aplicação disparada possui uma cópia da Dalvik, na qual a mesma é executada. A vantagem dessa construção consiste principalmente na segurança, pois esse sistema provê isolamento e controle de memória de programas de maneira automática.

De forma geral, uma arquitetura baseada em registradores terá melhor performance do que uma arquitetura baseada em pilhas. Esse efeito pode ser observado quando se compara eficiência de instruções de cada uma das máquinas virtuais, quando temos um número significativamente menor de instruções na máquina virtual baseada em registradores para executar um código de alto nível contra um número maior de instruções na máquina virtual

[2] Disponível em: <http://source.android.com/source/index.html>; Acesso em nov. 2015

baseada em pilha, mesmo que ao custo de maiores tamanhos de palavras de instruções. Com instruções maiores, porém, uma arquitetura baseada em registradores leva mais tempo para executar cada instrução em comparação com uma arquitetura baseada em pilha. Porém, o produto entre tempo por instrução e o número de instruções executadas é menor em arquiteturas baseadas em registradores, ao compararmos com arquiteturas baseadas em pilha, o que significa que uma arquitetura baseada em registradores demorará menos tempo para executar uma aplicação (EHRINGER, 2012).

2.1.2.1 Overhead

Independente da escolha, tanto a JVM quanto a Dalvik são máquinas virtuais, portanto, adicionam mais um nível de abstração entre o aplicativo e a CPU que executa o mesmo. Mesmo com otimizações e melhorias, a execução de um aplicativo em ambas será mais lenta do que a execução de uma aplicação escrita na linguagem nativa do sistema (C ou C++ no caso de dispositivos Android). Essa "taxa" que é paga por se usar a máquina virtual e suas ferramentas e facilidades é chamada, aqui, de overhead. Esse overhead pode ser diminuído por diversas técnicas, desde o uso de caches de instruções ou até pela escrita de pedaços código na linguagem nativa do dispositivo. Essa escrita de trechos de código, abordada com mais detalhes na seção 2.1.3, também possui suas desvantagens, como o custo para a cópia de contexto. Na próxima seção veremos maneiras de se diminuir o impacto desse overhead no desempenho das aplicações.

2.1.3 Interface Nativa Java

A Interface Nativa Java, ou JNI (*Java Native Interface*), é uma interface disponibilizada para dispositivos Android para execução de códigos em linguagem nativa dentro de aplicações escritas em linguagem natural da plataforma.

Ao se utilizar a JNI, o desenvolvedor precisa especificar métodos, ou até classes, como nativas no código original de sua aplicação e então implementá-los na linguagem nativa da plataforma, utilizando bibliotecas específicas para tal implementação e tratando o envio e recebimento de contextos entre ambas partes da aplicação. Utilizando a JNI, criamos uma espécie de divisão no programa, pois ambas partes possuem estruturas de dados diferentes para dados, isso é, a representação de dados na linguagem nativa é, muitas vezes, diferente da representação de dados na linguagem original, necessitando realizar conversões quando existe

essa discrepância. Enquanto o uso de código nativo pode apresentar um ganho de performance no programa, não é aconselhado utilizá-lo, uma vez que código nativo geralmente não é seguro (*unsafe code*), possui menor portabilidade e mais difícil de se detectar erros de forma geral. O uso da JNI é apenas aconselhado em casos de código legado, como APIs e bibliotecas escritas em C ou C++, ou para tarefas de processamento intenso (OPEN HANDSET ALLIANCE, 2015).

2.2 Processadores Gráficos

O termo Processador Gráfico, ou GPU, surgiu no final do século passado, em 1999, quando o departamento de marketing da Nvidia chamou seu último lançamento, a GeForce 256, de "O primeiro processador gráfico do mundo", em tradução livre. Desde então, GPUs têm se tornado cada vez mais populares nos mais diversos dispositivos gráficos.

Por definição, uma GPU é um circuito eletrônico desenvolvido especialmente para manipular e alterar posições de memória rapidamente para acelerar o processo de criação de imagens para visualização em um display. Diferente de CPUs, GPUs possuem um conjunto de instruções especializado para o processamento de imagens e manipulação de elementos de computação gráfica, geralmente do estilo SIMD, explorando o paralelismo entre dados em suas operações sobre píxeis.

Essa especialização ocorre através de uma estrutura altamente paralela, geralmente sendo composta de dezenas, ou até centenas, de processadores estruturados especialmente para manipular grandes quantidades de memória rapidamente. Em um computador pessoal, processadores gráficos estão geralmente presentes em formas de placas de extensão de vídeo, mas também podem existir diretamente na placa-mãe, ou, até mesmo, dentro do mesmo chip do processador principal.

2.2.1 Processadores Gráficos para Propósito Geral

Enquanto a inspiração para GPUs veio apenas da necessidade de se existir um processador dedicado a gerar imagens para diversos dispositivos, a ideia de se possuir uma placa com diversos processadores, que facilitam o uso de diretivas de processamento paralelo, agradou muitos mais grupos do que o imaginado. Cientistas de diversas áreas começaram a utilizar, e continuam até hoje, utilizando processadores gráficos como se os mesmos fossem

processadores de propósito geral. Assim, ao invés de utilizar suas instruções especiais de manipulação de memória e gerar imagens, eles utilizam os processadores gráficos para realizar cálculos paralelos muito mais rapidamente do que se conseguiria em processadores de propósito geral, graças a presença de dezenas de núcleos de processamento extra nessa nova opção.

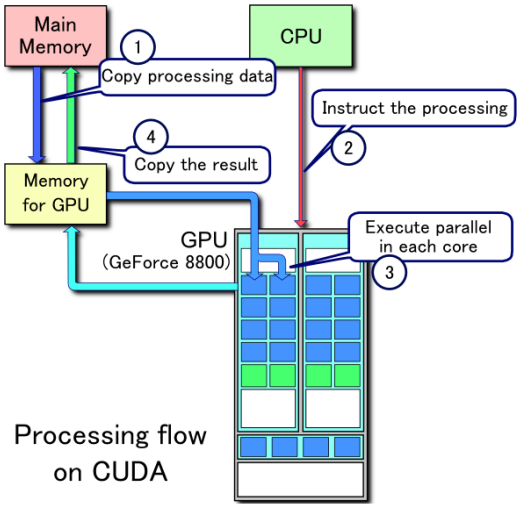
Inicialmente era necessário se utilizar diretivas disponíveis nas APIs especiais para utilização de processadores gráficos, como DirectX ou OpenGL. Essas APIs, porém, trabalham com imagens e não com dados. Com isso era necessário transformar dados em "pseudo-imagens", enviando-os para a memória da GPU como imagens para então enviar as operações que se desejava realizar sobre os mesmo como se fossem operações do processador de vértices para só então ser possível realizar os cálculos desejados.

Com o crescimento desse método de uso para processadores gráficos, em 2007, a Nvidia lançou a plataforma CUDA (*Compute Unified Device Architecture*), que define uma API, exclusiva para processadores gráficos da Nvidia, para programação paralela, oferecendo funções similares àquelas já conhecidas em programação C ou C++, porém, agora, com esse código sendo executado em paralelo em diversos núcleos (chamados de *CUDA cores*).

O procedimento para se realizar um processamento utilizando CUDA se dá em quatro passos, como podemos ver na figura 2.3. Primeiramente, como a GPU possui memória diferenciada da memória principal do sistema, é necessário copiar todos os dados necessário para o processamento para a memória do processador gráfico. Após realizada a cópia, o processador principal envia para a GPU o que deve ser processado, isso inclui lista de instruções e quantidade de threads. Após recebidos os dados necessário, a GPU executa, sem uma ordem estrita (respeitando suas dependências, porém), todas as threads requisitadas. Como último

passo, que é opcional, é necessário copiar as informações de volta para a memória principal do programa.

Figura 2.3 – Fluxo de processamento em CUDA



Fonte: Tosaka – Wikimedia Commons[3]

2.2.2 CUDA Cores

Inicialmente, processadores gráficos possuíam diversos tipos diferentes de processadores internamente: processadores de vértices, processadores de texturas, entre outros. Com a criação da plataforma CUDA, porém, esse cenário foi mudado. Cada um dos diversos processadores distintos que antes habitavam uma GPU foi transformado em um núcleo CUDA, ou CUDA core.

Estes CUDA cores possuem a capacidade combinada de todos os processadores os quais eles substituíram, além de novas habilidades para processamento de propósito geral, um dos objetivos da plataforma CUDA. A principal vantagem em se existirem CUDA cores, ao invés de processadores específicos, é o poder que o desenvolvedor ganha ao ter a opção de apenas escalonar tarefas a serem executa das pelos núcleos de execução, sem se preocupar com a quantidade de processadores específicos disponíveis em certo ponto da aplicação. Com isso, é possível diminuir a quantidade de cores inativos em um certo momento, pois cada núcleo pode executar qualquer tarefa pendente, considerando que existam tarefas a serem executadas.

[3] Disponível em: < https://commons.wikimedia.org/wiki/File:CUDA_processing_flow_(En).PNG>; Acesso em nov. 15

Também, como a distribuição de tarefas entre os núcleos é feita em tempo de execução, a arquitetura é escalável, pois, ao se aumentar a quantidade de núcleos, o poder de processamento também aumenta sem necessitar nenhuma alteração no código escrito. Assim, novas gerações de processadores podem executar o mesmo código de maneira mais eficiente.

2.2.3 Processadores Gráficos em Dispositivos Móveis

Nos últimos anos a tendência do uso de processadores gráficos chegou para dispositivos móveis. Impulsionado pela necessidade de performance em diversas atividades realizadas nesses dispositivos, desde interfaces mais fluídas, até melhor desempenho para jogos móveis, a presença de uma GPU no dispositivo se torna importante para garantir a impressão de alta performance para o usuário.

Assim como no mercado de processadores gráficos para PCs, a Nvidia também está presente para dispositivos móveis, com a série de SoC Tegra. Estes chips são compostos de processador de propósito geral e processador gráfico. Nas versões que precedem e incluindo Tegra 4, os processadores gráficos presentes nesses SoCs são feitos especialmente para dispositivos móveis, portanto os avanços presentes nas arquiteturas de GPUs para PCs não estão presentes nesses dispositivos. Isso mudou, porém, a partir da Tegra K1, que possui a arquitetura codinome Kepler, a mesma arquitetura de diversos processadores gráficos para PCs, da linha GTX 600, GTX 700 e alguns modelos da série 800.

Com essa mudança, a Nvidia deixa claro sua preocupação com o avanço de processadores gráficos para dispositivos móveis. Além disso, também mostra interesse no controle do consumo de energia para sua série de placas para embarcados, baseando toda a estratégia de marketing do seu SoC na eficiência do mesmo com baixo custo energético. (NVIDIA, 2015)

2.3 Benchmarks

Para se conseguir uma boa estimativa de comparação de consumo de energia e tempo de execução entre distintos cenários, foram utilizados seis diferentes benchmarks. Todos esses benchmarks são algoritmos que são utilizados pela empresa norte-americana SPEC (Standard Perfomance Evaluation Corporation), uma empresa dedicada em “produzir, estabelecer, manter e endossar um padrão” para benchmarks relevantes para a mais nova geração de computadores

(SPEC, 2015). Os benchmarks selecionados fazem parte do conjunto SPEC MPI® 2007, criado para avaliar processamento paralelo, operações com ponto flutuante e tarefas com computação intensiva (SPEC, 2015), adaptados para execução em Java pela SciMark, desenvolvida pelo Insituto Nacional de Padrões e Tecnologia (NIST) dos Estados Unidos.

Dentre estes seis benchmarks escolhidos, quatro deles apresentam cenários com os quais a introdução de um processador gráfico pode beneficiar bastante o algoritmo, devido a média ou alta paralelização de seus algoritmos. Já os outros dois casos apresentam algoritmos que possuem alta dependência de dados, isso é, cada passo do algoritmo depende da resolução do passo anterior, fazendo com que os mesmos não possam desfrutar completamente do ganho de paralelização adquirido com o processador gráfico.

2.3.1 Algoritmo de Compressão de Lempel-Ziv Modificado

Esse algoritmo de compressão, criado em 1984, é um algoritmo baseado em dicionários. Normalmente é um algoritmo utilizado quando se deseja realizar uma compressão sem perda (*lossless*). Possui uma taxa de compressão de 33% a 25% do tamanho original do arquivo. É utilizado, por padrão, nos formatos de imagem TIFF e GIF

A ideia do algoritmo é encontrar padrões com o maior tamanho possível dentro do alvo de compressão e substitui-lo por um único símbolo. Ele realiza isso através da criação de um dicionário, adicionado ao arquivo gerado, que mapeia símbolos encontrados para um único símbolo de saída, encontrando conjuntos de símbolos de maior tamanho possível.

Enquanto o processo de descompressão é altamente paralelizável, por ser apenas leitura em um dicionário e substituição de símbolos por frases, o processo de compressão possui alta dependência entre dados, pois o algoritmo precisa encontrar as maiores frases mais comuns na lista de símbolos alvo, além de necessitar alterar entradas do dicionário conforme o processo de compressão vai evoluindo.

2.3.2 Transformada Rápida de Fourier

A transformada rápida de Fourier é um algoritmo criado para realizar o cálculo da transformada discreta de Fourier, ou sua inversa. A transformada de Fourier tem como principal função a conversão de sinais entre domínios distintos, tendo, como exemplo mais famoso, a conversão de sinais no domínio do tempo para o domínio frequência e vice-versa. Este processo

de conversão é bastante custoso, sendo de ordem $O(n^2)$ para o algoritmo original da DFT, mas tem seu custo reduzido para O(n*log n) quando calculado através da transformada rápida de Fourier.

A utilização da FFT abrange as mais diversas áreas, desde processamento de sinais, processamento de imagens e até matemática pura. Com um uso tão abrangente, existir uma ferramenta para se realizar um cálculo rapidamente é de extrema importância.

O algoritmo da FFT é reconhecido por sua eficiência por utilizar uma estratégia de divisão e conquista para se obter o resultado de uma DFT. Para tal, dada uma DFT longa de N pontos, cada passo do algoritmo reduz essa DFT em duas DFTs menores, de tamanhos N_1 e N_2. Esse processo se repete recursivamente até se alcançar um N_k de tamanho primo, o qual é resolvido através de uma DFT comum. Quando um passo não for decomponível por 2, os elementos restantes até a próxima potência de 2 são preenchidos por 0s.

Devido a essa natureza de dependências entre os diversos passos do cálculo da FFT, a paralelização alcançada não é considerável, uma vez que diversas threads esperarão resultados de threads em níveis mais baixos na árvore de divisão e conquista, threads essas executando operações complexas, que definem a DFT.

2.3.3 Fatoração LU

A fatoração LU, do inglês *Lower* e *Upper*, é um método de análise numérica que fatora uma dada matriz em duas matrizes, uma composta apenas de elementos abaixo da diagonal principal e outra por elementos acima da diagonal principal. A figura 2.4 mostra um exemplo de como a fatoração se apresenta. Para realizar essa fatoração, é importante que a matriz em questão seja quadrada.

Figura 2.4 – Modelo de fatoração LU

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

Fonte: Elaborada pelo autor

A fatoração LU é amplamente utilizada em métodos computacionais que envolvem matrizes. Estes métodos incluem, mas não se limitam a, soluções de sistemas de equações

lineares com o mesmo número de variáveis e expressões, inversões de matrizes e cálculo de determinantes de matrizes.

A decomposição se dá através de um algoritmo que calcula individualmente cada elemento de cada uma das novas matrizes e é facilmente paralelizável. Porém, há uma pequena dependência de dados entre alguns elementos, diminuindo o ápice de paralelização do processo. Cada elemento (i,j), tanto da matriz L, quanto da matriz U, que não seja 0 ou 1 por definição, depende do resultado dos elementos (g,j) e (i,h), onde $g \in [1...i)$ e $g \in [1...j)$.

2.3.4 Integração pelo Método de Monte Carlo

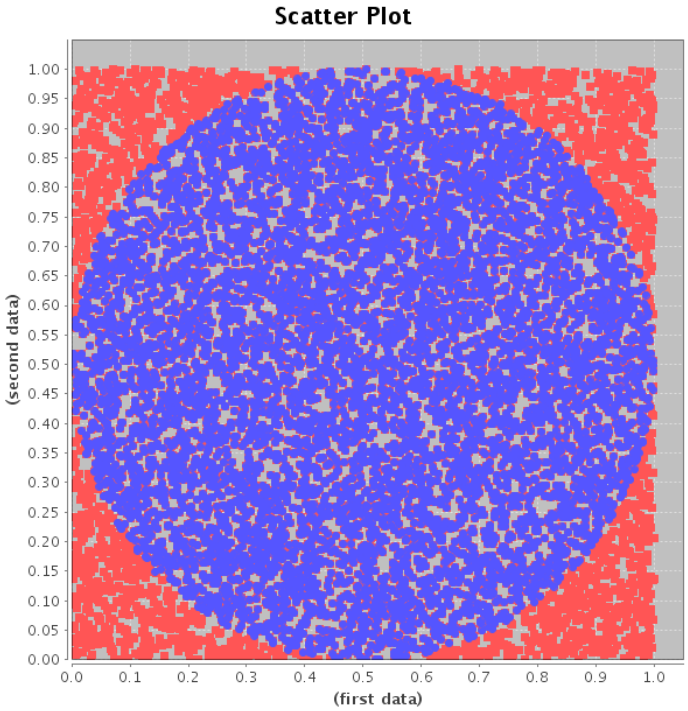
Método de Monte Carlo é a nomenclatura utilizada para métodos estatísticos de rápida integração. Estes métodos se baseiam em gerar massivas amostragens aleatórias para se obter resultados numéricos. Esta classe de métodos é popularmente utilizada para obtenção de aproximações numéricas de funções em que não se é viável, ou até mesmo impossível, se obter uma solução analítica ou determinística.

A classe métodos de Monte Carlo utilizada compreende aqueles que resolvem integrações. Para tal, se define uma curva, ou curvas, as quais se deseja descobrir a integral. Define-se então o intervalo de integração desejado e se calcula a área n-dimensional na qual este intervalo é presente, chamado de zona de interesse. Com esses dados em mãos, se inicia um processo de geração de pontos aleatórios no espaço n-dimensional das curvas: quanto maior a quantidade de pontos, maior a precisão do método. Para cada um desses pontos, se calcula se o mesmo está dentro da zona de interesse. Caso o mesmo esteja na zona de interesse, o ponto é marcado como válido.

Feito este processo, calcula-se a porcentagem de pontos dentro da zona de interesse e, a partir desse valor, estima-se a área n-dimensional do cálculo. A figura 2.5 mostra um exemplo

de cálculo finalizado para se estimar o valor de pi, a partir da área de um círculo. Os pontos vermelhos estão fora da zona de interesse, enquanto os roxos estão dentro da zona.

Figura 2.5 – Estimação de pi pelo método de integração de Monte Carlo



Fonte: ProbaPerception[4]

2.3.5 Método de Sobre-Relaxação Sucessiva

Este é um método utilizado para resolução de sistemas de equações lineares, que serve como uma alternativa para o método de Gauss-Seidel para o problema. O método parte de um sistema de equações no formato Ax = b, onde A é uma matriz quadrada de tamanho n por n e x e b vetores de tamanho n por 1. A figura 2.6 mostra a estrutura de problema analisado.

A partir desse sistema, a matriz A é decomposta em três outras matrizes: D, L e U, sendo que A = D + L + U. D é uma matriz composta apenas pelos elementos da diagonal principal da matriz A, enquanto L e U são matrizes estritamente triangulares, formadas pelas partes

[4] Disponível em: < http://probaperception.blogspot.com.br/2012/10/estimation-of-number-pi-monte-carlo.html>; Acesso em nov. 15

inferiores e superiores, respectivamente, da matriz A, em relação a diagonal principal. A figura 2.7 mostra a decomposição da matriz A em D, L e U.

Figura 2.6 – Estrutura do problema SOR

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Fonte: Elaborada pelo autor

Figura 2.7 – Decomposição da matriz A

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} + L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix} + U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

Fonte: Elaborada pelo autor

Com essa fatoração, o problema pode então ser reescrito como visto na figura 2.8.

Figura 2.8 – Problema reescrito

$$(D + \omega L)\mathbf{x} = \omega\mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x}$$

Fonte: Elaborada pelo autor

Temos adicionado um fator ω, que é, enquanto ω > 1, chamado de fator de relaxação. Deve-se então iniciar a iteração com um valor arbitrário para o vetor x do lato direito da equação. Com esses dados em mão, inicia-se um processo iterativo que realizará o cálculo do vetor x real. Cada etapa dessa iteração utiliza o valor de x do lado esquerdo do passo anterior, isso é, o valor calculado pelo passo anterior, como x no lado direito, justificando a necessidade de um valor inicial para o x da direita. O processo se repete até a convergência do sistema ser encontrada.

Enquanto o algoritmo iterativo não possui muitas alternativas de paralelização entre seus passos, tendo em vista que um passo necessita do anterior para ser iniciado, o cálculo realizado

em cada passo pode ser bastante paralelizado, por ser um cálculo de matrizes sem operações complexas.

2.3.6 Multiplicação de Matrizes Esparsas

Este benchmark realiza a multiplicação entre duas matrizes quadradas de lado 1000, porém, aproximadamente 5000 elementos em cada matriz não é 0, classificando ambas matrizes como matrizes esparsas. Cada linha da matriz possui aproximadamente 5 elementos que não são 0, igualmente espaçados um com os outros desde o início de cada linha até a diagonal principal. Devido a construção especial das matrizes, o armazenamento das mesmas não é feita de forma convencional, mas sim através do uso de n listas, uma representando cada linha das matrizes. Devido a essa construção, o algoritmo realiza a multiplicação apenas das linhas e colunas não nulas, uma vez que os resultados com uma linha ou uma coluna nula é trivial.

Com a verificação de linhas e colunas não nulas, o algoritmo se torna algo um pouco mais complexo do que uma simples multiplicação de matrizes, fazendo ele um pouco mais lento de ser processado, porém, com um número muito menor de dados para ser processado.

3 TRABALHOS RELACIONADOS

Estudos sobre eficácia, tanto energética, quanto em termos de desempenho, para processadores gráficos são bastante comuns de forma geral. Motivados principalmente pelos próprios fabricantes, é comum aparecerem artigos e estudos por volta da data de lançamento de uma nova arquitetura para mostrar sua eficácia. Por sua vez, análises de consumo energético para dispositivos móveis é também são um assunto comum para pesquisas na área de sistemas embarcados. Isso ocorre pela importância do problema de maximização da vida útil de um dispositivo sem abrir mão da performance do mesmo, motivado, fortemente, pelo desejo do usuário de se ter um dispositivo de alto desempenho, mas com um tempo de bateria aceitável.

Mesmo com ambas áreas de pesquisa com diversas amostras de cada caso, para as mais diversas arquiteturas e modelos, estudos que analisam o impacto da adição de processadores gráficos em dispositivos móveis é praticamente nulo. Não há, em fontes de fácil acesso, dados sobre uma análise, mesmo que superficial, no tema. Por sua vez, o tema possui suma importância, pois uma análise de casos que revele pontos nos quais se há ganho substancial de desempenho, com pouco ou nenhum aumento no consumo do dispositivo pode motivar a criação de melhores algoritmos e implementações para essa delicada plataforma.

Mostramos então trabalhos relacionados, seja na análise energética de dispositivos móveis, seja na análise energética de processadores gráficos.

3.1 Análises de Consumo de Energia de Processadores Gráficos

3.1.1 Statistical Power Consumption Analysis and Modeling for GPU-based Computing

Em Ma et al. (2009), os autores realizam um estudo com o intuito de modelar o consumo de energia de uma dada GPU de maneira estatística, sem o conhecimento prévio do processador gráfico em questão. Essa estimativa é feita através de execuções de específicos programas de estilo GPGPU no processador alvo e medindo o consumo da placa.

Após adquiridas as curvas de consumo para cada aplicação testada, é feito uma aproximação, através de métodos numéricos, do consumo de energia por instrução em alto nível

na GPU. Essa estipulação é feita uma vez que os autores têm conhecimento dos algoritmos executados e da curva de consumo de cada algoritmo na GPU.

Enquanto os resultados, expostos no próprio artigo, são satisfatórios, os próprios autores comentam da necessidade de um número muito grande de dados para servirem como treinamento do modelo estatístico criado. Também não há como, em métodos estatísticos em geral, uma maneira de predizer quantos dados seriam necessários para se treinar completamente o modelo para uma GPU ou arquitetura específica.

3.1.2 An Integrated GPU Power and Performance Model

Em Hong Kim (2010), os autores propõem um modelo empírico para predizer, tanto consumo, quanto performance de aplicações GPGPU, buscando poder reduzir o consumo de aplicações de modo geral controlando a quantidade de núcleos ativos no processador gráfico de forma otimizada. A vantagem do modelo criado, em comparação aos outros existentes, é que o modelo não necessita de medida de hardware, apenas o código em questão.

O aplicativo desenvolvido, o IPP, analisa todos os kernels enviados para a GPU, realizando então cálculos de performance e consumo de energia para cada kernel. Com estes dados, o programa otimiza o desempenho do programa por watt, gerando um modelo de alocação de threads e blocos para a GPU.

A principal limitação do IPP está nos métodos que o mesmo usa, focados na execução massiva da GPU. Com isso, aplicações com foco grande no fluxo de controle são dificilmente modeladas.

3.2 Análises de Consumo de Energia Java e JNI

3.2.1 AndroProf

Em Sartor et al. (2013), os autores criam uma ferramenta de *profiling* para a plataforma Android. Baseada na ferramenta de emulação QEMU, parte da SDK do Android, é criada uma extensão para a ferramenta para a coleta e análise de dados. A ferramenta é multiplataforma, suportando a emulação das arquiteturas ARM e MIPS executando Android e, parcialmente, x86.

A ferramenta fornece ao usuário ferramentas para extração de dados após realizadas simulações na versão alterada do QEMU. Dentre estes dados disponíveis estão a obtenção da

lista de instruções básicas executada durante a emulação, informações sobre quais são os *basic blocks* da aplicação e estimativas de consumo de energia pela aplicação.

3.2.2 Who Killed My Battery: Analyzing Mobile Browser Energy Consumption

Em Thiagarajan et al. (2012) os autores analisam o consumo de energia de uma das principais tarefas realizadas com *smartphones*, o acesso a páginas web. O estudo evidencia a falta de conhecimento quanto ao impacto do acesso de páginas web em dispositivos móveis, além da falta de otimização, tanto em questão de transmissão de dados, quanto em consumo de energia de diversas páginas no modo *mobile*.

É feita a medição minuciosa do consumo energético de diversas páginas *mobile*, utilizando-se hardware externo para tal. Após a medição é feita a análise da fonte do consumo de cada página, entre javascript, css, imagens e transmissão de dados, para mapear as causas do consumo medido.

Os trabalhos sobre dispositivos móveis exibem a preocupação existente com o consumo de energia realizado por aplicações em dispositivos móveis, mas sem nunca abrir mão da busca por um melhor desempenho nos dispositivos. Pensamento que nos leva ao elemento do processador gráfico, como mencionado anteriormente, sinônimo de aumento de desempenho em diversas plataformas, uma adição importante para se atingir um patamar de performance desejado.

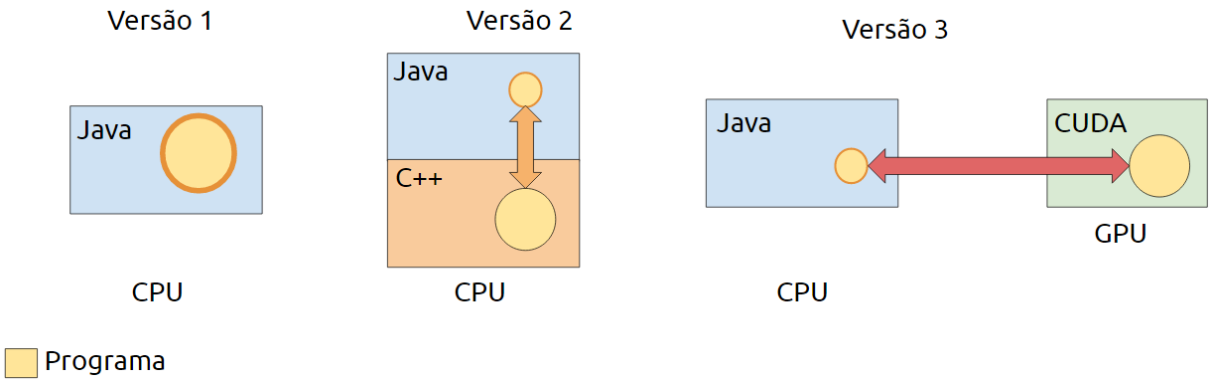
Porém, como podemos ver nos trabalhos sobre processadores gráficos, mesmo em cenários em que o consumo de energia não é algo limitante da plataforma, já existe essa preocupação com o consumo de GPUs de maneira geral. Esses exemplos exibem a importância do nosso estudo, onde ambas áreas se encontram, em que as preocupações com o consumo da GPU sejam bastante acentuadas, visto a plataforma em que a mesma é adicionada.

4 IMPLEMENTAÇÃO

Para se alcançar os resultados desejados, foi montado o esquema da figura 4.1. Através desse esquema, para cada algoritmo do conjunto de benchmarks, temos três versões diferentes de programas: a versão original, a versão em JNI e a versão em CUDA. Começa-se com a versão original, implementada em Sartor et al. (2013), que faz a adaptação de diversos benchmarks SPEC, originalmente em C++, para Java. A partir dessa versão, AndroProf (2013) faz a análise de quais são os *basic blocks* de cada programa, isso é, quais são os métodos que mais gastam trabalho da CPU durante a execução dos mesmos e converte esses métodos para a JNI, que, aqui aparecem como a segunda versão de cada benchmark. A importância de se analisar a segunda versão dos programas é analisar o quanto do ganho de desempenho e energia é devido a eliminação do overhead da Dalvik através da JNI.

A terceira versão, por sua vez, reutiliza a implementação em Java da segunda versão, porém, a parcela de código que antes era executada via JNI, reescrita para esse estudo, é executada em um processador gráfico, em CUDA.

Figura 4.1 – Esquema de implementação dos benchmarks



Fonte: Elaborada pelo autor

É importante lembrar que, como o espaço de endereçamento de memória da GPU é completamente diferente da CPU, os benchmarks da versão três possuem um alto custo de troca de contexto entre a CPU e a GPU, pois é necessário fazer a cópia de todos os dados utilizados pela GPU para a mesma. A versão dois também há uma pequena troca de contexto entre o programa em Java e em C++ devido ao fato de que as linguagens não necessariamente possuem a mesma estrutura de dados para armazenar os dados utilizados pelo programa, necessitando realizar essa tradução de estruturas em alguns casos. A versão um, por sua vez, não necessita

de nenhuma troca de contexto ou tradução. Este, porém, paga o preço por estar sendo executado em uma máquina virtual, graças ao overhead causado pela mesma.

Para se realizar a medição para a versão três, como não havíamos disponível nenhuma ferramenta para medição de consumo de energia ou desempenho do processador gráfico do dispositivo móvel, foi necessário realizar uma adaptação para se chegar em valores aproximados. Para gerar esses resultados, a parte do programa em Java é executada em um dispositivo móvel, mas a parte do programa em CUDA é executada em um computador comum com uma placa de vídeo GeForce GT9500, com 32 CUDA cores. A escolha dessa placa de vídeo foi feita pelo fato de a mesma não ser um modelo de última geração, contrabalanceando o fato da mesma estar sendo utilizada para simular um processador gráfico de um dispositivo móvel. Ainda assim, a mesma possui o mesmo conjunto de instruções de placas de vídeo móveis disponíveis pela Nvidia, possibilitando um cálculo mais preciso de consumo energético.

A escolha de se realizar essa medição em um computador pessoal ao invés de em dispositivo móvel foi movida pelo fato de, no momento de realização da implementação, não existir uma plataforma disponível para simulação Android que possua um processador gráfico. Existem plataformas da Nvidia para simulação de dispositivos móveis, porém as mesmas não suportam Android.

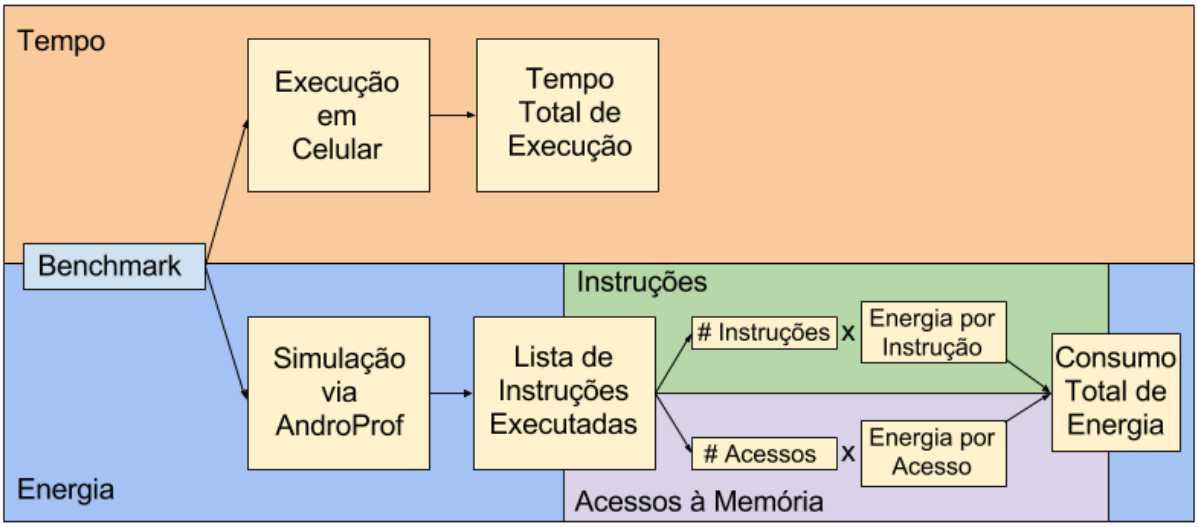
4.1 Processo de Medição – Versões 1 e 2

A figura 4.2 mostra a base do processo realizado para obtenção das medições. O processo se focou em adquirir duas medidas de cada versão dos benchmarks: tempo de execução e consumo de energia. A escolha de realizar a medição do tempo foi para que se fosse possível observar também a variação de performance em cada um dos casos, além da variação de consumo energético.

Para as versões 1 e 2, a obtenção de resultados foi idêntica. Todo código em ambas versões foi executado em um smartphone Samsung Galaxy SM-G900M (S5) e a partir dessa execução, o tempo executado foi obtido através de funções da biblioteca System.nanoTime. Para o cálculo do consumo energético foi utilizado a ferramenta AndroProf (SARTOR et al., 2015) para simular a execução do programa. Ao se utilizar a ferramenta, a mesma disponibiliza,

ao final da execução, a lista de instruções que seriam executadas durante a simulação pelo processador do dispositivo.

Figura 4.2 – Processo de obtenção de medições – versões 1 e 2



Fonte: Elaborada pelo autor

Conseguidas a lista de instruções executadas, a ferramenta foi novamente utilizada para realizar o cálculo de consumo de energia. Para tal foi considerado o custo médio por instrução de um processador ARM A9 (BLEM et al., 2013).

Para obtenção do consumo de energia total, porém, é necessário levar em conta os acessos a memória principal, feito por instruções de leitura e escrita. Para estimar o consumo por acesso, utilizamos dados obtidos de MURALIMANOHAR (2015), com configurações similares àquelas encontradas no dispositivo físico utilizado, isso é, 2GB RAM, 8 bancos, 64kbytes de bloco e arquitetura de 28nm, chegando ao valor médio de 2,51nJ por acesso.

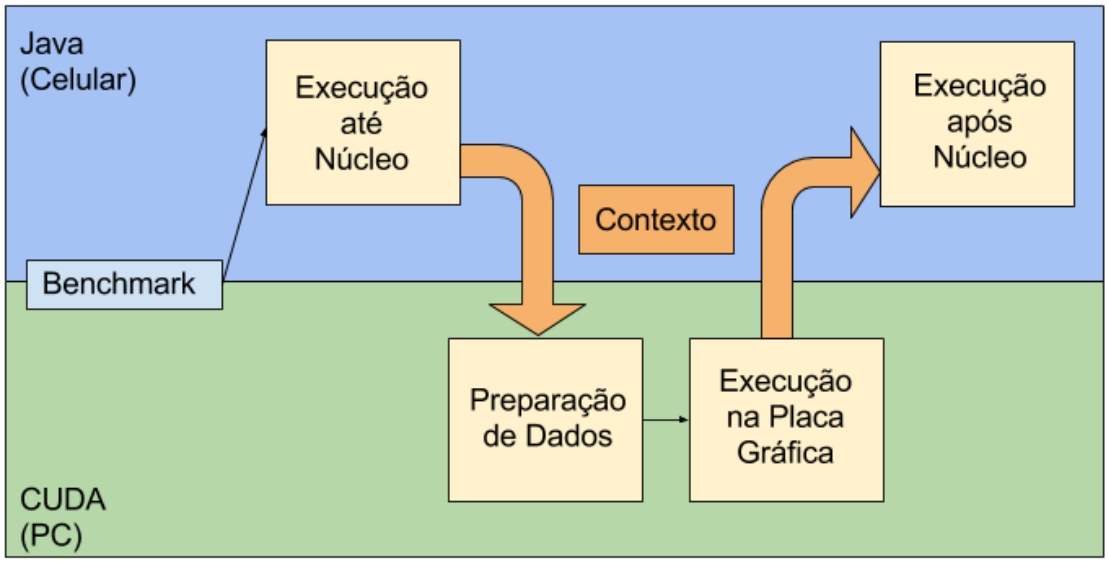
4.2 Processo de Medição – Versão 3

A medição da versão 3 foi feita de maneira diferente das versões 1 e 2, como podemos ver na figura 4.3. A parte do benchmark realizada em Java utilizou a mesma técnica das versões 1 e 2 para a parte do programa em Java.

Já para a terceira parte, havia um problema: estimar o custo pela transferência de contexto entre a CPU e a GPU. Esse problema ocorre por que, dada uma simulação da versão 3, pedaços de código da mesma são simulados em duas plataformas diferentes. A parte em Java é simulada pelo AndroProf e a parte em CUDA simulada pelo Multi2Sim (UBAL et al., 2012), com os

resultados finais somados após todo o processo de simulação. O Multi2Sim é uma ferramenta de simulação de aplicações, análoga ao AndroProf, mas que possui suporte a simulação de programas em CUDA.

Figura 4.3 – Processo de execução da versão 3 dos benchmarks



Fonte: Elaborada pelo autor

Não existem, porém, dados que mostrem os gastos necessários para o envio dos dados do processador principal do dispositivo móvel, representado pela simulação realizada no AndroProf, para o processador gráfico, representado pela simulação realizada no Multi2Sim. Resta, então, como a única alternativa, estimar esse valor. Essa estimativa foi realizada através de dois cenários para cada simulação e execução da versão três de cada benchmark.

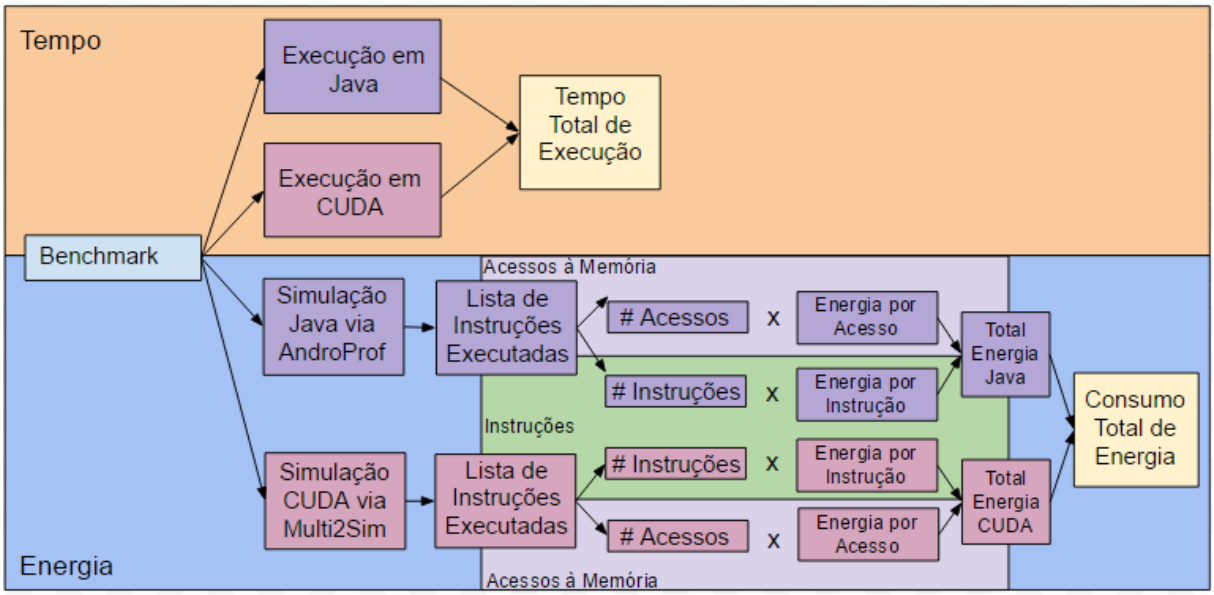
No primeiro cenário, a passagem de contexto é feita através de um processo que, com certeza, terá uma performance e gasto energético maior do que o caso real. A troca de contexto foi simulada através de uma leitura de arquivo com o contexto necessário pelo programa. Para obter tal contexto, foi alterado o código em Java, para, na hora de chamar o método em questão, o contexto necessário pelo mesmo é salvo em um arquivo de texto. Com esse arquivo em mãos, o programa em CUDA, além de executar o núcleo de execução como requisitado, também faz todo o processo de leitura de arquivo. Este processo serviu também para validar o resultado final, ao escrever o resultado do processamento final em arquivo.

No segundo cenário, a passagem de contexto é feita de maneira similar a troca de contexto realizada na JNI. Para tal, a versão Java agora invoca um método nativo, mas o mesmo é um método vazio, realizando apenas o retorno para a função. A partir desse resultado em Java,

é adicionado, então, o valor da simulação e da execução do processador gráfico (simulado anteriormente utilizando o Multi2Sim), ignorando a leitura de arquivo de contexto, realizada no primeiro cenário.

Com ambos cenários, estimamos que o valor real deve estar em algum ponto entre eles. Podemos afirmar isso por dois motivos. Uma leitura de arquivo com certeza será mais lenta e gastará mais energia do que uma troca de contexto entre GPU e CPU. E uma troca de contexto JNI, devido ao fato de ambas partes possuírem o mesmo endereçamento de memória, será mais rápida e consumirá menos energia do que uma troca de contexto entre GPU e CPU, que necessita realizar a movimentação de todo contexto necessário para as tarefas da GPU para a memória da mesma. Pode-se também inferir que o resultado, ao longo da evolução dos processadores gráficos irá, mais e mais, tender a se aproximar do resultado do segundo cenário. Empresas, como Nvidia e AMD, estão anunciando suporte em suas plataformas para memória compartilhada entre CPU e GPU, para diminuir os custos existentes em se manipular os mesmos dados em ambas. Com essas mudanças futuras, resultados de uma simulação em que há uso de CPU e GPU serão mais dependentes dos algoritmos utilizados e não tão dependentes de detalhes de hardware.

Figura 4.4 – Processo de obtenção de medições – versões 3-1 e 3-2



Fonte: Elaborada pelo autor

Para aquisição de valores de tempo para os programas em CUDA, executados em um processador gráfico em um PC, foi mensurada através de tempos fornecidos por resultados da

biblioteca time.h. Os trechos de código executados na GPU nesses benchmarks, foram executados em uma GeForce GT 9500, com 32 CUDA cores, como mencionado anteriormente.

Para os valores de consumo energético, foi realizado um esquema similar ao utilizado nas versões 1 e 2 dos benchmarks. Foi utilizada a ferramenta Multi2Sim para obter a lista de instruções executadas pela GPU nos trechos não executados em um dispositivo móvel. Para instruções da GPU, foram utilizados valores médios por instrução, obtidos em (HONG et al., 2010).

4.3 Núcleos de Execução

Para um melhor entendimento dos resultados, é importante analisar as diferenças entre cada um dos benchmarks, principalmente em seus núcleos de execução e contextos necessários para execução dos mesmos. Os núcleos de execução de cada benchmark são os trechos de código mais executados pelos mesmos, segmentos de código pelos quais o processador dedica a maior parte do seu tempo a executar. A informação sobre os principais *basic blocks* (que compõe os núcleos) de cada benchmark é extraída utilizando a ferramenta AndroProf, que lista e agrupa os *basic blocks* em sua interface. Esse agrupamento mostra a quantidade de vezes que cada *basic block* de cada benchmark foi executado.

A partir desses *basic blocks* , é possível encontrar quais são os métodos mais executados pelo benchmark, que chamamos aqui de núcleos de execução. Esse foi o processo utilizado por Sartor et al. (2013) para a obtenção dos núcleos de execução traduzidos para JNI e utilizados, como critério de comparação, como versão 2. Os núcleos de execução utilizados para a

implementação da versão 3 são estes mesmos, para que se possa realizar a comparação entre o desempenho e consumo entre as diferentes plataformas de execução.

A implementação destes núcleos de execução em CUDA, para cada um dos benchmarks, possui tamanho médio de 400 linhas de código por benchmark, dependendo do tamanho e da complexidade do contexto de cada um dos benchmarks.

4.4 Comparação de Contexto

Dentre os seis benchmarks, eles podem ser classificados em três categorias quanto a espécie de contexto passado entre CPU e GPU, chamadas aqui por: Contexto Unitário, Contexto Simples, ou matricial, e Contexto Complexo.

A categoria de Contexto Unitário compreende apenas um benchmark, Monte Carlo. O contexto enviado para a GPU desse benchmark é apenas a quantidade de pontos que o mesmo deve criar. Isso ocorre porque todo o método de cálculo do algoritmo é feito ao redor de se gerar uma grande quantidade de pontos aleatórios em um espaço n-dimensional, não necessitando mais nenhuma informação vinda da CPU. Os dados enviados de volta para a CPU também são mínimos: a quantidade de pontos internos a zona de interesse do algoritmo, novamente, um número único.

A categoria de Contexto Simples, também podendo ser chamada de Contexto Matricial, possui três algoritmos: LU, SOR e Sparse. Elas recebem esse nome, pois os únicos contextos enviados ou recebidos pela CPU são matrizes. Para os benchmarks LU e SOR, as matrizes estão em um formato padrão, construídas como um vetor de vetores, ambas de tamanho quadrado, de lado 1000. Já o benchmark Sparse possui uma matriz quadrada de lado 10.000 como contexto. Como se trata de uma matriz esparsa, a representação da matriz é feita por um vetor de listas. Cada posição desse vetor representa uma linha da matriz original e cada elemento de cada lista representa uma posição não-nula da matriz original, representada na lista pelo seu valor e posição na linha. O dado enviado como resposta pela GPU é o resultado da multiplicação no mesmo formato, um vetor de listas, representando a matriz resultante da multiplicação.

A última categoria, chamada de Contexto Complexo, compreende os dois últimos algoritmos: Compress e FFT. Ambos algoritmos possuem diversos dados que necessitam ser enviados para a GPU, não apenas o dado na qual a GPU irá trabalhar, mas também dados de configuração muito mais complexos do que os outros benchmarks. Além disso, o processamento na GPU não é feito de forma única. Enquanto os benchmarks anteriores

realizavam todo o processamento na GPU com apenas uma troca de contexto, os algoritmos dessa categoria realizam mais trocas de contexto, sendo duas trocas para o benchmark Compress e diversas trocas durante a execução do FFT.

5 RESULTADOS

Para avaliar o desempenho de cada uma das versões dos benchmarks, foram realizadas medições de tempo e de consumo de energia, como descrito no capítulo de implementação. Todas as execuções em dispositivo móvel foram realizadas em um celular Samsung Galaxy S5, rodando o sistema operacional Android KitKat (4.4) no Modelo G900M, que possui um SoC Qualcomm MSM8974AC Snapdragon 801, que é composto por um processador Krait 400 quad-core com frequência de operação de 2.5GHz, 2GB de RAM, 16GB de armazenamento, alimentado por uma bateria íons-lítio de 2800mAh. As execuções em PC foram realizadas em um sistema com uma placa de vídeo Nvidia GeForce 9500GT, com a frequência de operação de 1400MHz e 32 CUDA cores com a frequência de operação em 550MHz.

5.1 Tempo de Execução

A tabela 5.1 mostra os resultados da média de tempo de execução entre dez execuções dos benchmarks nos sistemas acima citados. Os cenários de execução da versão 3, como mencionados na seção 4.2, são representados aqui como versões 3-1 e 3-2, para o cenário no qual a troca de contexto é realizada via leitura de arquivo e para o cenário que a troca de contexto é tratada como uma troca JNI, respectivamente. A tabela 5.2 mostra os mesmos resultados da tabela 5.1, porém normalizados quanto ao tempo de execução da versão 1, para melhor exibir a variância de tempo de execução entre as versões de cada benchmark. A figura 5.1, por sua vez, apresenta estes resultados normalizados da tabela 5.2 de forma gráfica.

Tabela 5.1 – Tempo médio de execução dos benchmarks

Nome do Benchmark	Tempo Versão 1 (s)	Tempo Versão 2 (s)	Tempo Versão 3-1 (s)	Tempo Versão 3-2 (s)
Compress	1278	1214	1478	1415
FFT	97	91	145	132
LU	715	684	601	556
Monte Carlo	1412	1285	849	846
SOR	460	432	384	351
Sparse	602	557	502	481

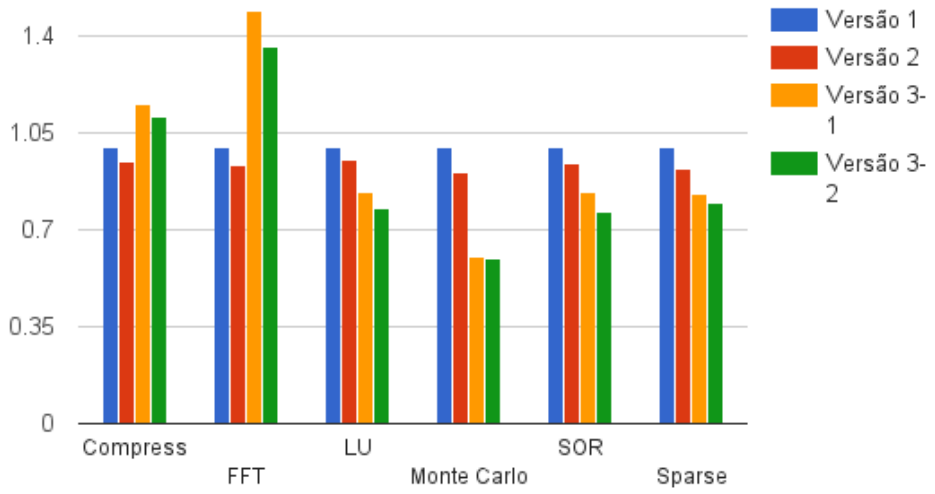
Fonte: Elaborada pelo autor

Tabela 5.2 – Normalização do tempo médio de execução dos benchmarks

Nome do Benchmark	Versão 1 Normalizada	Versão 2 Normalizada	Versão 3-1 Normalizada	Versão 3-2 Normalizada
Compress	1.0	0.95	1.16	1.11
FFT	1.0	0.94	1.49	1.36
LU	1.0	0.96	0.84	0.78
Monte Carlo	1.0	0.91	0.60	0.60
SOR	1.0	0.94	0.83	0.76
Sparse	1.0	0.93	0.83	0.80

Fonte: Elaborada pelo autor

Figura 5.1 – Representação gráfica da tabela 5.2



Fonte: Elaborada pelo autor

A primeira observação quanto aos resultados é o aumento de tempo de execução em ambas versões 3 nos benchmarks Compress e FFT. Esse comportamento ocorre devido à natureza dos algoritmos dos benchmarks em questão. O algoritmo de Compressão, como discutido na seção 2.3, é um algoritmo de difícil paralelização. Assim, a quantidade de CUDA cores ativos, dado um certo instante de execução, é menor do que o desejável, devido à grande quantidade de dependências entre cada núcleo de execução do mesmo.

Com isso, o algoritmo acaba sendo executado quase sequencialmente, pois a maioria das suas threads está trancada, esperando o resultado de outra ser finalizado. Devido a isso, sua execução será mais lenta quando executado dentro da GPU do que quando executado na CPU. Esse comportamento ocorre já que o poder de processamento de um núcleo de uma GPU é bastante inferior se comparado ao de uma CPU, mesmo no caso de um processador de um

dispositivo móvel. Essa perda de eficiência, porém, só não é maior pois a outra parte do algoritmo, a parte de descompressão, é altamente paralelizável, pois realiza apenas leituras em dicionário e substituições de símbolos.

Por sua vez, o algoritmo de FFT não possui essa vantagem, pois todo seu algoritmo sofre com alta dependência de dados. Essa dependência faz com que seus resultados, quando executados pela GPU, levem uma quantidade maior de tempo ao comparado com a quantidade de tempo necessária para execução do mesmo apenas pela CPU do dispositivo. Existem, porém, implementações diferentes para esse algoritmo, que o otimizam para o alto processamento paralelo, como é o caso da biblioteca cuFFT (NVIDIA, 2015), desenvolvida pela Nvidia para CUDA. A análise desta versão poderá ser feita em estudos futuros.

Os algoritmos LU, SOR e Sparse possuem resultados similares. Isso ocorre devido à similaridade de paralelização dos mesmos. Todos apresentam operações sobre matrizes, casos clássicos de paralelização que, mesmo apresentando alguma dependência de dados, como no caso do algoritmo LU; ou ainda diferente representação de dados, como no caso do Sparse; possuem grande ganho com o aumento da quantidade de núcleos de processamento.

O algoritmo que mais se beneficiou do uso da GPU foi o Monte Carlo. A causa desse grande ganho de desempenho (aproximadamente 40% nas versões 3, em relação à versão 1) se dá principalmente da alta paralelização do mesmo. Cada thread executada pela GPU é, em grande parte, independente de todas outras threads em execução. A única dependência que cada thread apresenta é na hora de retornar o resultado, quando há uma adição simples em uma variável compartilhada. Essa ação, porém, possui suporte da própria plataforma, na forma de instruções atômicas, para controle de acesso. Com isso, esse algoritmo mostra todo o potencial do processador gráfico ao ser adicionado com os dados do dispositivo móvel.

Todavia, esse ganho poderia ser ainda maior. O ganho foi delimitado pela quantidade relativamente baixa de CUDA cores disponíveis no processador gráfico, podendo escalar quase

que linearmente com o aumento da quantidade e da frequência dos núcleos disponíveis no sistema.

A tabela 5.3 apresenta a normalização do tempo médio de execução do segundo cenário da versão 3, normalizado pelo primeiro cenário da versão.

Tabela 5.3 – Normalização do tempo médio de execução para a versão 3

Nome do Benchmark	Cenário 2 / Cenário 1
Compress	0.96
FFT	0.91
LU	0.93
Monte Carlo	1.00
SOR	0.91
Sparse	0.96

Fonte: Elaborada pelo autor

A normalização é realizada para se exibir a diferença entre os métodos de passagem de contexto entre a GPU e a CPU. Como cenário 1, temos a transferência de contexto realizada via leitura de arquivo, isso é, de uma maneira muito mais lenta do que ocorreria em um cenário natural, que seria a passagem de dados entre a CPU e a GPU. Já no cenário 2 temos essa mesma transferência, simulada agora, porém, por uma chamada JNI. Este processo, considerando o que existe hoje, é mais rápido do que uma passagem normal de dados entre CPU e GPU, já que, tanto a execução normal, quanto a execução via JNI, são feitas no mesmo processador e com o mesmo endereçamento de memória. Os resultados mostrados na tabela 5.3 exibem o aumento de velocidade do cenário 2 em relação ao cenário 1. Benchmarks com pequeno contexto, como o Monte Carlo (o contexto passado é apenas a quantidade de amostras a serem realizadas), praticamente não possuem diferenças entre os tempos de execução. Já em benchmarks como SOR e LU, que possuem matrizes com mais de 1.000.000 de elementos como contexto, e o benchmark de FFT, que possui uma estrutura complexa de elementos enviados como contexto (além de realizar essa troca de contexto diversas vezes), são os algoritmos que mais apresentam redução no tempo de execução comparando os cenários, chegando a quase 10% de redução de tempo no melhor caso.

5.2 Energia

A tabela 5.4 possui os resultados da média de consumo de energia entre dez execuções dos benchmarks nos sistemas acima citados. As versões 3-1 e 3-2 representam os diferentes cenários da versão 3, considerando a abertura de arquivo e a troca de contexto JNI, respectivamente, como trocas de contexto entre a CPU e a GPU. A tabela 5.5 exibe os mesmos

resultados da tabela 5.4, mas normalizados em função do resultado da versão 1, para melhor mostrar a variância do consumo de energia em cada benchmark. Finalmente, a figura 5.2 apresenta os resultados normalizados da tabela 5.5 de forma gráfica.

Tabela 5.4 – Média do consumo de energia dos benchmarks

Nome do Benchmark	Consumo Versão 1 (J)	Consumo Versão 2 (J)	Consumo Versão 3-1 (J)	Consumo Versão 3-2 (J)
Compress	202.64	184.74	195.43	190.42
FFT	14.42	13.39	14.35	13.35
LU	101.86	94.15	115.41	107.40
Monte Carlo	208.37	189.41	274.84	271.35
SOR	84.65	77.84	96.28	89.73
Sparse	106.04	97.13	115.39	113.14

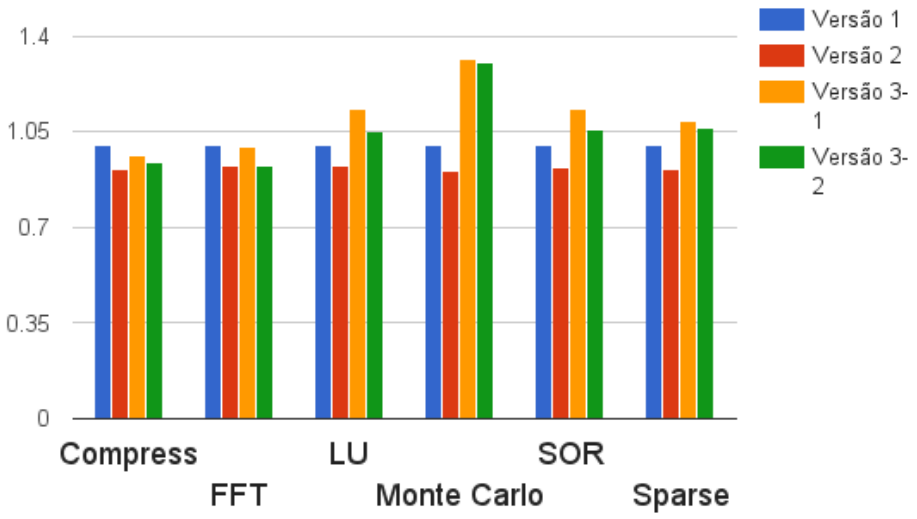
Fonte: Elaborada pelo autor

Tabela 5.5 – Normalização da média de consumo de energia dos benchmarks

Nome do Benchmark	Consumo Versão 1	Consumo Versão 2	Consumo Versão 3-1	Consumo Versão 3-2
Compress	1.0	0.91	0.96	0.94
FFT	1.0	0.93	1.00	0.93
LU	1.0	0.92	1.13	1.05
Monte Carlo	1.0	0.91	1.32	1.30
SOR	1.0	0.92	1.14	1.06
Sparse	1.0	0.92	1.09	1.07

Fonte: Elaborada pelo autor

Figura 5.2 – Representação gráfica da tabela 5.5



Fonte: Elaborada pelo autor

Para esses testes de consumo de energia, o resultado esperado era um aumento de energia consumida para todos os benchmarks na versão três. Esse aumento, porém, apenas ocorreu caso comprarmos as versões 3 com a versão 2, ou seja, a execução dos núcleos de execução na GPU contra a execução via JNI. Contudo, quando ao compararmos com a versão 1, alguns casos apresentaram um consumo de energia menor quando executados pela GPU do que quando executados completamente dentro da Dalvik.

Como analisados na seção anterior, os benchmarks Compress e FFT são os algoritmos nos quais a GPU não consegue prover à aplicação um ganho de desempenho, apresentando, inclusive, perda significativa. Dito isso, a execução das aplicações é feita de maneira similar àquela realizada no processador principal do dispositivo, quase de forma sequencial. Considerando que um núcleo CUDA tem menor desempenho e, também, menor consumo do que um núcleo do processador principal, o resultado, além de perda de desempenho, é um menor consumo de energia.

Os benchmarks de LU, SOR e Sparse possuem uma curva de gasto de energia similar, assim como apresentaram no quesito de desempenho. Todos os três benchmarks mostram um aumento de aproximadamente 11% em relação a versão 1, executada na Dalvik. Esse agrupamento ocorre, como visto na seção anterior, pela similaridade entre os contextos e execuções realizados na GPU, assim como o nível de paralelização possível. Devido a essas similaridades, todos os três possuem uma curva de consumo energético semelhante.

O benchmark com maior gasto, como o esperado, é o Monte Carlo, que possui, também, o maior ganho de desempenho. A grande possibilidade de paralelização, que tornou possível

seu elevado ganho de desempenho, agora é causa do grande aumento de consumo de energia exibido pelo benchmark.

As tendências apresentadas pela diferença de desempenho entre os cenários da versão 3 é vista novamente quando comparamos a diferença de consumo energético entre os cenários, como vemos na tabela 5.6.

Tabela 5.6 – Normalização do consumo médio de energia entre os cenários da versão 3

Nome do Benchmark	Cenário 2 / Cenário 1
Compress	0.97
FFT	0.93
LU	0.93
Monte Carlo	0.99
SOR	0.93
Sparse	0.98

Fonte: Elaborada pelo autor

Benchmarks com grande quantidade de contexto para ser enviado da CPU para GPU, como LU e SOR, ou com várias trocas de contexto, como o FFT, são aqueles que mais se beneficiam da melhora de processo de troca de contexto, apresentando um gasto quase 10% menor de energia. Outros casos, como Compress e o Sparse, não apresentam tanta mudança no consumo de energia ao se trocar o método de troca de contexto. O benchmark Monte Carlo, como possui quantidade desprezível de contexto, apresenta, também nesse caso, uma mudança mínima no consumo de energia entre os cenários da versão 3.

6 CONCLUSÃO E TRABALHOS FUTUROS

Processadores gráficos estão presentes cada vez mais em celulares e outros sistemas embarcados com interface gráfica. Com esse trabalho exibimos o impacto que o uso desse poderoso componente pode ter em nossas aplicações. Mostramos, também, que usar esse hardware especializado em qualquer cenário nem sempre pode resultar em bons resultados, deixando a critério do desenvolvedor o uso correto e apropriado dessa plataforma, que utilizada de forma errônea, pode, ou diminuir o desempenho de aplicações, ou aumentar de maneira significante o consumo de energia de um dispositivo, um preço que nem sempre pode ser pago quando falamos em dispositivos com uma fonte limitada de energia.

É importante que pesquisadores e empresas procurem as consequências do uso de processadores gráficos em dispositivos móveis, para que possam então desenvolver ferramentas melhores que, por sua vez, façam o uso dessa grande peça que temos a nossa disposição, mas de maneira consciente, uma vez que o uso descontrolado da mesma pode causar resultados catastróficos para a bateria e para o desempenho do dispositivo.

Para resultados mais conclusivos e definitivos para a questão de análise de energia de processadores gráficos seria necessário a existência de uma plataforma de simulação própria para tais testes. A Nvidia possui uma plataforma para tal, como citada no texto, porém, a mesma não possui, no presente momento, suporte para simulações com sistema Android, um grande problema que deve ser solucionado no futuro. O uso de diferentes plataformas de simulação e execução pode haver gerado algum ruído em meio aos dados da pesquisa, por isso, o ideal para este trabalho seria realiza-lo toda na mesma plataforma de simulação e na mesma plataforma de execução, para obtenção de resultados que adequem uns com os outros, pois os mesmos viriam da mesma fonte.

Outra comparação pertinente nessa mesma linha de pensamento seria o quanto custaria para se fazer o processamento de forma remota, enviando todo contexto via rede, seja Wi-Fi ou 3G/4G, para uma unidade de processamento com um processador gráfico, que, por sua vez, realizaria este processamento e enviaria o resultado de volta para o dispositivo móvel. Tal abordagem, porém, deve levar em consideração diversas variáveis, como disponibilidade e velocidade de rede, que fogem do escopo deste trabalho.

REFERÊNCIAS

OPEN HANDSET ALLIANCE. **ART and Dalvik | Android Source Code Project**. Disponível em: <http://source.android.com/devices/tech/dalvik/index.html>. Acesso em: novembro, 2015

OPEN HANDSET ALLIANCE. **JNI Tips | Android Developers**. Disponível em: <http://developer.android.com/training/articles/perf-jni.html>. Acesso em: novembro, 2015

OPEN HANDSET ALLIANCE. **Android Open Source Project.** Disponível em <http://source.android.com/>. Acesso em: novembro, 2015.

OPEN HANDSET ALLIANCE. **The Android Source Code | Android Source Code Project**. Disponível em: <http://source.android.com/source/index.html>. Acesso em: novembro, 2015.

LUIZ SARTOR, ANDERSON, ULISSES BRISOLARA CORREA, AND ANTONIO CARLOS SCHNEIDER BECK. AndroProf: A Profiling Tool for the Android Platform**.** ***Computing Systems Engineering (SBESC), 2013 III Brazilian Symposium on***, pp. 23-28. IEEE, 2013.

BLEM, EMILY, JAIKRISHNAN MENON, AND KARTHIKEYAN SANKARALINGAM. A detailed analysis of contemporary ARM and x86 architectures. ***UW-Madison Technical Report*** (2013).

MURALIMANOHAR, NAVEEN. **CACTI 6.0 Download Page**. Disponível em: <http://www.cs.utah.edu/~rajeev/cacti6/>. Acesso em: novembro, 2015.

NVIDIA. **cuFFT :: CUDA Toolkit Documentation**. Disponível em: <http://docs.nvidia.com/cuda/cufft/>. Acesso em: novembro, 2015

EHRINGER, 2012. EHRINGER, D. **The Dalvik Virtual Machine Architecture**, março 2012.

INTERNATIONAL DATA CORPORATION, **IDC: Smartphone OS Marketshare 2015, 2014, 2013, and 2012**. Disponível em: <http://www.idc.com/prodserv/smartphone-os-market-share.jsp>. Acesso em: novembro, 2015.

MA, XIAOHAN, MIAN DONG, LIN ZHONG, AND ZHIGANG DENG. Statistical power consumption analysis and modeling for GPU-based computing**.** ***Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)***. 2009.

UBAL, RAFAEL, BYUNGHYUN JANG, PERHAAD MISTRY, DANA SCHAA, AND DAVID KAELI. "Multi2Sim: a simulation framework for CPU-GPU computing."

Proceedings of the 21st international conference on Parallel architectures and compilation techniques, pp. 335-344. ACM, 2012.

OPEN HANDSET ALLIANCE. **Open Handset Alliance Members**. Disponível em: <http://www.openhandsetalliance.com/oha_members.html>. Acesso em: novembro, 2015.

HONG, SUNPYO, AND HYESOON KIM. An integrated GPU power and performance model**. *ACM SIGARCH Computer Architecture News***, vol. 38, no. 3, pp. 280-289. ACM, 2010.

STANDARD PERFORMANCE EVALUATION CORPORATION, **SPEC - Standard Performance Evaluation Corporation**. Disponível em: <https://www.spec.org/>. Acesso em: novembro, 2015.

STANDARD PERFORMANCE EVALUATION CORPORATION, **SPEC MPI® 2007**. Disponível em: <https://www.spec.org/mpi2007/> Acesso em: novembro, 2015.

THIAGARAJAN, NARENDRAN, GAURAV AGGARWAL, ANGELA NICOARA, DAN BONEH, AND JATINDER PAL SINGH. Who killed my battery?: analyzing mobile browser energy consumption**. *Proceedings of the 21st international conference on World Wide Web***, pp. 41-50. ACM, 2012.

NVIDIA, **The GPU - NVIDIA Tegra K1 Preview & Architecture Analysis**. Disponível em: <http://www.anandtech.com/show/7622/nvidia-tegra-k1/3>. Acesso em: novembro, 2015