UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GUILHERME MENEGUZZI MALFATTI

TÉCNICAS DE AGRUPAMENTO DE DADOS PARA COMPUTAÇÃO APROXIMATIVA

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Antonio Carlos S. B. Filho
Co-orientador: Prof. Dr. Luigi Carro

Porto Alegre
2017

CIP - Catalogação na Publicação

Malfatti, Guilherme Meneguzzi
Técnicas de agrupamento de dados para computação aproximativa / Guilherme Meneguzzi Malfatti. -- 2017.
90 f.

Orientador: Antonio Carlos Schneider Beck Filho.
Coorientador: Luigi Carro.

Dissertação (Mestrado) -- Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2017.

1. Computação Aproximativa. 2. Clusterização de Dados. 3. Redes Neurais. 4. Alta performance. I. Beck Filho, Antonio Carlos Schneider, orient. II. Carro, Luigi, coorient. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Profa. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Profa. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. João Luiz Dihl Comba
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Nesta etapa final da minha pesquisa, gostaria de expressar minha imensa gratidão a todas pessoas envolvida diretamente ou indiretamente. Meus sinceros agradecimentos pelas conversas, apoio e total incentivo oferecido no decorrer da pesquisa.

Muito obrigado a minha família, em especial aos meus pais, **Renato** e **Maria**, pelo exemplo, educação, amor e apoio no decorrer de toda minha vida. Obrigado por serem meu porto seguro.

A minha namorada **Débora**. Obrigado por estar sempre ao meu lado nos momentos difíceis desta caminhada, sempre incentivando e provando que os problemas enfrentados eram menores que realmente pareciam ser. Tuas palavras de incentivo e companhia foram um excelente combustível para que eu pudesse seguir em frente.

Agradeço ao meu orientador, **Antônio Carlos Schneider Beck Filho** e Co-orientador **Luigi Carro**, pela confiança depositada em mim. Seu comprometimento e disponibilidade foram essenciais para o desenvolvimento deste trabalho. Sem os seus direcionamentos e apontamentos "cirúrgicos", este trabalho não teria sido possível, meus sinceros agradecimentos.

Obrigado a todos meus amigos, pelo incentivo durante a produção deste trabalho. Aos colegas de laboratório, pelo tempo investido para ler, revisar e comentar parte desta dissertação, em especial agradeço o colega e amigo **Marcelo Brandalero**, pelo interesse, pela ajuda e pelos questionamentos feitos no decorrer deste trabalho, com certeza foram fundamentais nesta pesquisa. Muito obrigado.

Agradeço as empresas **Hewlett-Packard e Compasso Tecnologia LTDA**, por incentivar os meus estudos e pela flexibilidade que eu pudesse frequentar as aulas e reuniões do grupo de pesquisa.

RESUMO

Dois dos principais fatores do aumento da performance em aplicações *single-thread* – frequência de operação e exploração do paralelismo no nível das instruções – tiveram pouco avanço nos últimos anos devido a restrições de potência. Neste contexto, considerando a natureza tolerante a imprecisões (i.e.: suas saídas podem conter um nível aceitável de ruído sem comprometer o resultado final) de muitas aplicações atuais, como processamento de imagens e aprendizado de máquina, a computação aproximativa torna-se uma abordagem atrativa. Esta técnica baseia-se em computar valores aproximados ao invés de precisos que, por sua vez, pode aumentar o desempenho e reduzir o consumo energético ao custo de qualidade.

No atual estado da arte, a forma mais comum de exploração da técnica é através de redes neurais (mais especificamente, o modelo *Multilayer Perceptron*), devido à capacidade destas estruturas de aprender funções arbitrárias e aproximá-las. Tais redes são geralmente implementadas em um hardware dedicado, chamado acelerador neural. Contudo, essa execução exige uma grande quantidade de área em chip e geralmente não oferece melhorias suficientes que justifiquem este espaço adicional.

Este trabalho tem por objetivo propor um novo mecanismo para fazer computação aproximativa, baseado em reúso aproximativo de funções e trechos de código. Esta técnica agrupa automaticamente entradas e saídas de dados por similaridade, armazena-os em uma tabela em memória controlada via software. A partir disto, os valores quantizados podem ser reutilizados através de uma busca a essa tabela, onde será selecionada a saída mais apropriada e desta forma a execução do trecho de código será substituído. A aplicação desta técnica é bastante eficaz, sendo capaz de alcançar uma redução, em média, de 97.1% em *Energy-Delay-Product* (EDP) quando comparado a aceleradores neurais.

Palavras-chave: Computação Aproximativa; Clusterização de Dados; Redes Neurais; Alta performance.

Data Grouping Techniques For Approximate Computing

ABSTRACT

Two of the major drivers of increased performance in single-thread applications - increase in operation frequency and exploitation of instruction-level parallelism - have had little advances in the last years due to power constraints. In this context, considering the intrinsic imprecision-tolerance (i.e., outputs may present an acceptable level of noise without compromising the result) of many modern applications, such as image processing and machine learning, approximate computation becomes a promising approach. This technique is based on computing approximate instead of accurate results, which can increase performance and reduce energy consumption at the cost of quality.

In the current state of the art, the most common way of exploiting the technique is through neural networks (more specifically, the Multilayer Perceptron model), due to the ability of these structures to learn arbitrary functions and to approximate them. Such networks are usually implemented in a dedicated neural accelerator. However, this implementation requires a large amount of chip area and usually does not offer enough improvements to justify this additional cost.

The goal of this work is to propose a new mechanism to address approximate computation, based on approximate reuse of functions and code fragments. This technique automatically groups input and output data by similarity and stores this information in a sofware-controlled memory. Based on these data, the quantized values can be reused through a search to this table, in which the most appropriate output will be selected and, therefore, execution of the original code will be replaced. Applying this technique is effective, achieving an average 97.1% reduction in Energy-Delay-Product (EDP) when compared to neural accelerators.

Keywords: Approximate Computing; Data Clustering; Neural networks; High Performance.

LISTA DE FIGURAS

Figura 1.1 Visão geral da técnica desde a anotação do código até execução final da aplicação........... 14
Figura 2.1: Exemplo de uma rede neural dividida em camadas de entrada (Input Layer), interna (Hidden) e saída (OutPut). 18
Figura 2.2: Esquema de unidade McCulloch – Pitts. 19
Figura 2.3: Número de ocorrências para cada entrada quando a função seno é executada na aplicação FFT e função aproximada. 20
Figura 2.4: Técnica de Loop Perforation........ 21
Figura 2.5: Resultado da aplicação da técnica Loop Perforation x(y %), onde x é o speedup e y a porcentagem de erro. 22
Figura 2.6: Hardware reconfigurável da transformação de Parrot. 24
Figura 2.7: Benchmarks utilizados na avaliação da técnica e taxa de erro. 24
Figura 2.8: Melhorias em performance e consumo energético. 25
Figura 2.9.: Configuração do hardware para uma LUT sequencial. 26
Figura 2.10: Resultados da técnica fuzzy memoization. 27
Figura 2.11: Resultados apresentando a diferença entre a comunicação tradicional e a comunicação aproximada. 28
Figura 3.1: Apresentação de uma rede neural Multilayer Perceptron, enfatizando a quantidade de multiplicações em cada neurônio. 30
Figura 3.2: Gráfico apresentando dois pontos aleatórios no plano 36
Figura 3.3: Gráfico apresentando a mudança da posição de cada centróide. 37
Figura 3.4: Gráfico apresentando a mudança de alguns pontos no plano para outro cluster. 37
Figura 3.5: Pseudocódigo do algoritmo K-means Cluster. 38
Figura 3.6: Mapeamento dos clusters de entrada para as respectivas saídas. 39
Figura 3.7: Resultado da clusterização hierárquica. 40
Figura 3.8: Mapeamento do agrupamento hierárquico. 41
Figura 3.9: Fluxo para obter o valor mais próximo em memória (lookups tables). 43
Figura 3.10: Representação comum de trechos de códigos distintos. 45
Figura 3.11: Diagrama de sequência da técnica proposta. 46
Figura 4.1: Acessos a memória cache L1, L2, L3 e memória principal, normalizado em relação ao baseline. 57
Figura 4.2: Aumento da quantidade de instruções executadas e ciclos relativo ao algoritmo baseline, quando executado as abordagens RN e EDCAC em software. 58
Figura 4.3: Arquitetura do coprocessador. 60
Figura 4.4: Quantidade de ciclos para as abordagens EDCAC e RNs em hardware, normalizados com respeito a execução do baseline (algoritmo original executado em software). 66
Figura 4.5: Speedup total de cada aplicação utilizando EDCAC. 67
Figura 4.6: Consumo de energia para ambas abordagens, EDCAC e RN, dividindo entre consumo da memória e lógica, normalizado com respeito ao total de consumo da RN. 68
Figura 4.7: Área utilizada pela implementação individual de cada benchmark. 69
Figura 4.8: Análise de espaço e de projeto do benchmark FFT. 70
Figura 4.9: Análise de espaço e de projeto do benchmark Inversek2j 71
Figura 4.10: Análise de espaço e de projeto do benchmark Jmeint. 71
Figura 4.11: Análise de espaço e de projeto do benchmark JPEG. 72
Figura 4.12: Análise de espaço e de projeto do benchmark K-means........ 72
Figura 4.13: Análise de espaço e de projeto do benchmark Sobel. 73
Figura 4.14: Taxa de erro alcançado por EDCAC com quantidade distintas de clusters. 74

LISTAGEM

Listagem 3.1: Sintaxe da anotação pragma 31
Listagem 3.2: Implementação do hotspot original do benchmark FFT.. 33
Listagem 3.3: Código original retirado do benchmark JPEG.. 46
Listagem 3.4: Funções dct() e quantization() originais 46
Listagem 3.5: Exemplo da aplicação JPEG com aproximação, substituindo as funções dct() e quantization() 49
Listagem 3.6: Código utilizado pela transformação em redes neurais. 51

LISTA DE TABELAS

Tabela 4.1: Benchmarks utilizadas na avaliação, com características de cada função aproximada, dados de treinamento e resultados da aproximação. 53
Tabela 4.2: Configuração do simulador Gem5 56
Tabela 4.3: Configuração de memória utilizada por cada aplicação. 64
Tabela 4.4: Dados absolutos para os cenários citados nesta seção. 65
Tabela A.1: Consumo de potência de cada unidade em ponto flutuante utilizadas pelo hardware de cálculo da distância de Manhattan, bem como dados de área e temporização. 84
Tabela B.1: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de entrada do benchmark FFT. 85
Tabela B.2: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de saída do benchmark FFT 86
Tabela B.3: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de entrada e saída do benchmark Inversek2j. 86
Tabela B.4: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de entrada do benchmark Jmeint. 87
Tabela B.5: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de saída do benchmark Jmeint. 87
Tabela B.6: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de entrada e saída do benchmark JPEG. 88
Tabela B.7: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de entrada do benchmark K-means. 88
Tabela B.8: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de saída do benchmark K-means. 89
Tabela B.9: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de entrada do benchmark Sobel. 89
Tabela B.10: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de saída do benchmark Sobel. 90

LISTA DE ABREVIATURAS E SIGLAS

CIC: Clusters of Input Clusters 40
CPU: Central Processing Unit 76
DRAM: Dynamic random access memory 22
EDCAC: Enhanced Data Clusteringfor Approximate Computing 13
FPGA: Field-programmable gate array 25
GPU: Graphics Processing Unit 25
IC: Input Clusters 39
ILP: Instruction Level Parallelism 12
MLP: Multi Layer Perceptron 18
MPEG: Moving Picture Experts Group 23
O: Output 39
PEs: Processing Elements 62
PSNR: Peak signal-to-noise ratio 28
TLP: *Thread-Level-Parallelism* 12
UPs: Unidades de processamento 59
VLSI: Very-large-scale integration 25

SUMÁRIO

RESUMO 4
ABSTRACT 5
LISTA DE FIGURAS 6
LISTAGEM 7
LISTA DE TABELAS 8
LISTA DE ABREVIATURAS E SIGLAS 9
1 INTRODUÇÃO 12
1.1 Estrutura da Dissertação 14
2 PRINCÍPIOS E TRABALHOS RELACIONADOS 16
2.1 Redes Neurais Artificiais (Multilayer Perceptron) 17
2.2 Trabalhos relacionados 19
2.2.1 Camada de algoritmo 21
2.2.2 Camada de arquitetura 22
2.2.3 Camada de circuito 25
2.2.4 Outros trabalhos 27
3 EDCAC 29
3.1 A Técnica 29
3.1.1 Fase de Programação 31
3.1.2 Compilação 33
3.1.3 Execução 42
3.2 Modelo de Implementação 45
3.3 Considerações 51
4 AVALIAÇÃO EM SOFTWARE E HARDWARE DO FRAMEWORK EDCAC 52
4.1 Benchmarks 52
4.2 Implementação em Software 55
4.2.1 Metodologia 55
4.3 Implementação em Hardware 59
4.3.1 Arquitetura do coprocessador 59
4.3.2 Metodologia 61
4.4 Considerações finais 74
5 CONCLUSÃO 76
5.1 Trabalhos Futuros 77
REFERÊNCIAS 79

A. APÊNDICE A 84
B. APÊNCIDE B 85

1 INTRODUÇÃO

As constantes melhorias relacionadas ao aumento da performance dos processadores de propósito geral sempre se basearam em: aumentar o número de transistores disponíveis no chip; melhorar a organização do hardware e explorar paralelismo a nível das instruções (ILP, do inglês *Instruction-Level-Parallelism*). Estas técnicas tornaram-se bastante agressivas devido à cessação da escala de Dennard (BOHR, 2007) (que basicamente observava a voltagem e a corrente proporcionais ao tamanho linear dos transistores, que por consequência, a potência seria proporcional à área), e ao aumento da complexidade do circuito, que cresce quadraticamente em relação ao aumento linear da exploração de ILP (OLUKOTUN; HAMMOND, 2005). Porém, o emprego de tais técnicas torna-se muito caro. Arquitetos também exploram paralelismo no nível de threads (TLP, do inglês *Thread-Level-Parallelism*) através de processadores *multicore*. Contudo, estas alterações não afetam o desempenho de aplicações *single thread*, ou de aplicações que, devido a sua natureza, não mostram ganhos significativos quando paralelizadas. Desta forma, é necessário criar novas soluções para otimizar as aplicações que não se beneficiam de múltiplas *threads*.

Considerando que algumas aplicações naturalmente toleram “um certo” grau de imprecisão em seus resultados (ESMAEILZADEH et al., 2012a; ST AMANT et al., 2014; XU et al., 2016), a computação aproximativa é um paradigma que introduz a qualidade como uma nova métrica na exploração de espaço e projeto de microprocessadores. Neste nicho que, apesar de específico, abrange uma quantidade significativa de aplicações, é possível explorar esta característica: engenheiros podem trocar precisão por melhor desempenho, área e energia. Desta forma, computação aproximativa surge como uma abordagem atrativa em aplicações *single thread*, antes técnicas estagnadas.

Atualmente, programas com as características citadas acima utilizam técnicas baseadas na substituição da execução de regiões de código, os quais são tolerantes a imprecisão pela invocação de um hardware dedicado que implementa uma rede neural (RN) (CHEN et al., 2012; ESMAEILZADEH et al., 2012a; GRIGORIAN et al., 2015; GRIGORIAN; REINMAN, 2015; MOREAU et al., 2015; YAZDANBAKHSH et al., 2015). O programador identifica estas regiões durante o desenvolvimento e provê dados de treinamento para que o framework automaticamente seja capaz de treinar a RN a fim de substituir a execução destes segmentos de código pela execução da RN.

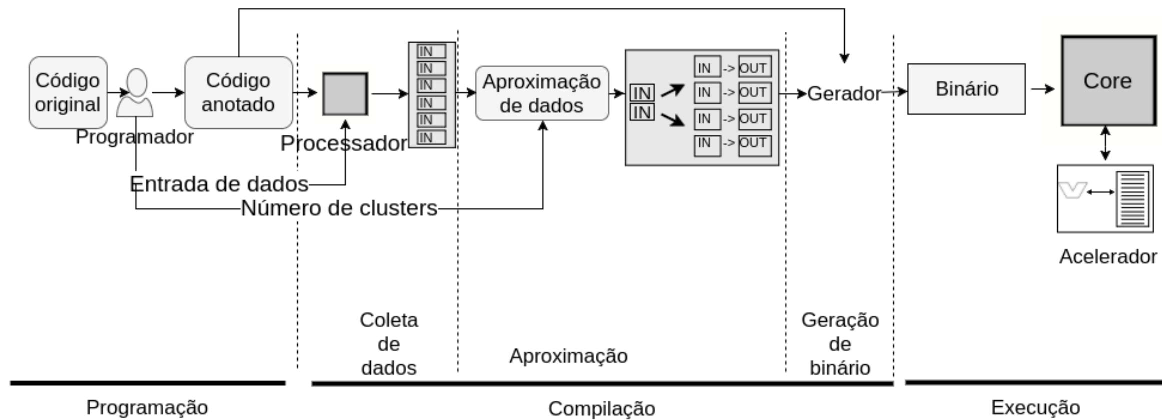
Embora trabalhos anteriores tenham mostrado que a utilização de RN é eficaz na implementação de funções aproximadas, experimentos discutem a eficiência do uso de RN, pois é necessário uma grande quantidade de operações em ponto flutuante e altos custos de roteamento de informações entre as camadas, transferindo a informação de um neurônio para a camada seguinte, que, obviamente impactam na performance e energia, tornando o custo da sua implementação em *hardware* quase proibitivo. Desta forma, considerando que o objetivo é prover uma versão aproximada de um código exato, este trabalho apresenta uma alternativa: a quantização e armazenamento de entradas e saídas de trechos de código. Isto é, a mesma função que seria aproximada com RN é pré-computada e armazenada em uma tabela, permitindo a substituição de operações caras e complexas por um simples acesso à memória, resultando em ganhos potencias em performance, energia e área.

Desta maneira, este trabalho apresenta uma alternativa às RNs: o framework EDCAC (do inglês *Enhanced Data Clustering for Approximate Computing*). Conforme apresentado na Figura 1.1~~Figura 1.1~~, ele é capaz de substituir a computação original de um trecho de código pré-selecionado por uma *lookup table* controlada por software a fim de selecionar o resultado mais apropriado baseado na entrada original e nas informações pré-computadas. Semelhante à abordagem com RN em (ESMAEILZADEH et al., 2012a), o programador deve identificar a região de código que seja possível aproximar, como também prover dados de treinamento da aplicação. Então, em tempo de compilação, o framework monitora o comportamento das entradas e saídas de código a ser aproximado e utiliza um algoritmo de agrupamento de dados a fim de selecionar as entradas e saídas mais representativas. Estes dados serão salvos juntamente com o código binário da aplicação. Em seguida, em tempo de execução, estes valores são carregados em memória controlada via software e, sempre que a região do código aproximado é utilizada, é executada uma busca nestas memórias, para que o valor de saída mais similar ao original substitua a computação (i.e., execução das instruções originais daquele trecho de código no processador). Assim, aumenta-se o desempenho e reduz-se o consumo energético; mas ao mesmo tempo perde-se em precisão.

Este trabalho avalia performance, área, energia e qualidade da técnica proposta utilizando o conjunto de benchmarks AxBench(YAZDANBAKHSH et al., 2017), através de medidas fornecidas pela síntese dos circuitos utilizando as ferramentas Cadence RTL Compiler (CADENCE) e CACTI (MURALIMANOHAR et al., 2009), e as compara com os resultados obtidos utilizando a técnica de RN. Utilizando o framework EDCAC, é possível alcançar resultados similares em qualidade aos trabalhos citados previamente, fornecendo um

Energy-Delay Product (EDP) 97.1% melhor que RN e menor consumo de área na maioria dos benchmarks.

Figura 1.1 Visão geral da técnica desde a anotação do código até execução final da aplicação.



Fonte: O autor.

1.1 Estrutura da Dissertação

Para oferecer um entendimento geral e uma melhor leitura, este trabalho está dividido em cinco capítulos, que oferecem uma visão de todos os conceitos utilizados, detalhes do projeto e a implementação da nova abordagem. O capítulo dois apresenta uma visão geral de computação aproximativa, como princípios, definição de conceitos e técnicas relacionadas a computação aproximativa a nível de software, hardware e outros sistemas, frisando o nível da arquitetura, que é onde este trabalho se aplica.

Posteriormente, o capítulo três apresenta um estudo sobre a proposta de implementação utilizada nesta dissertação, a fundamentação teórica da técnica utilizada, a geração de dados aproximados e a busca otimizada em memória. Para cada um destes passos as soluções foram estudadas e apresentadas.

O capítulo quatro apresenta detalhes do projeto, como a proposta de hardware, bem como um experimento comparando o estado atual da arte utilizando redes neurais, o qual originou a proposta deste trabalho. Ao final do capítulo é apresentado tanto a avaliação em software como em hardware para validar a técnica proposta. É neste capítulo que o desempenho em termos de performance e energia são coletados e avaliados.

Por fim, no capítulo cinco é apresentada a conclusão do estudo realizado nesta dissertação, bem como as contribuições obtidas no decorrer do trabalho. Além disso, esse capítulo apresenta indicativos para trabalhos futuros relacionados à computação aproximativa, em que abordagem proposta pode ser ampliada e adaptada de tal forma a resolver outros problemas ou obter melhores resultados.

2 PRINCÍPIOS E TRABALHOS RELACIONADOS

De maneira genérica, o termo computação aproximativa pode ser definido como a ação de trocar algumas operações precisas por operações imprecisas e potencialmente de menor custo. Neste paradigma, observamos que há uma grande quantidade de processamento de dados semelhantes, os quais podem ser ignorados para que somente os mais significativos sejam processados.

O espaço de projeto de sistemas computacionais é tipicamente guiado por três métricas: desempenho, área e energia. Para algumas aplicações (e.g.: processamento de áudio e vídeo, jogos, *machine learning*, etc.), o resultado final de toda computação não precisa ser exato. Assim, computação aproximativa torna-se um novo paradigma que, baseado na observação de que diversas aplicações são naturalmente tolerantes a imprecisões, introduz um novo eixo neste espaço de projeto: a qualidade. Desta maneira, pode-se optar por sacrificar a qualidade da aplicação em troca de aumento de desempenho ou redução no consumo energético.

Computação aproximativa tem sido utilizada em uma grande variedade de domínios tolerantes a erros. De fato, existe uma oportunidade emergente para novas arquiteturas e novas técnicas, devido a sinergia entre aplicações que toleram computação imprecisa. Abaixo lista-se alguns dos domínios mais utilizados:

- Processamento de sinais
- Robótica
- Jogos
- Compressão de dados
- *Machine learning*
- Processamento de imagens

Aqui está uma definição de computação aproximativa que este trabalho utiliza:

"Computação aproximativa é a ideia que sistemas computacionais podem optar por ter 100% de acurácia ou sacrificar a mesma em troca de eficiência. Isso inclui qualquer técnica onde o sistema intencionalmente expõe falhas para a camada de aplicação para conservar recursos computacionais. "

Esta definição é claramente ampla o suficiente para incluir ideias que existiram desde os primórdios da computação até os dias atuais.

A granularidade na qual a computação aproximativa se aplica não é intuitiva, porém essencial para ter sucesso nesta abordagem. A aproximação aplicada em instruções e palavras individuais da memória é conhecida como granularidade fina (e.g., (ESMAEILZADEH et al., 2012b)) enquanto costuma-se chamar de granularidade grossa quando se aproxima todo o algoritmo ou parte de uma função (e.g., (ESMAEILZADEH et al., 2012a)).

A granularidade de uma técnica afeta a sua generalidade e sua capacidade de eficiência. A granularidade fina é mais genérica. Uma unidade de multiplicação aproximada, por exemplo, pode ser utilizada para todas as multiplicações de um programa, mas o ganho em eficiência é fundamentalmente limitado aos multiplicadores. Técnicas de aproximação que trabalham com granularidade grossa, por outro lado, têm um potencial de ganho maior, pois, além do possível controle do local em que a mesma será aplicada, uma parte maior do programa será afetada. Desta forma, é possível ter um controle maior sobre a aplicação e, por consequência, potenciais ganhos em aumento de desempenho.

Com base nisso, este capítulo apresenta uma introdução a redes neurais, as quais são amplamente utilizadas no atual estado-da-arte, bem como outras técnicas utilizadas em trabalhos relacionados, que aplicam computação aproximativa para criar novas análises de custo-benefício entre diversas métricas. Mesmo que estas técnicas relatem com sucesso a utilização de redes neurais para fins de aproximação, o maior desafio de redução energética e melhorias em performance não foram obtidos de uma forma ótima. Neste contexto, a técnica proposta neste trabalho possui um potencial de ganho mais agressivo em comparação a redes neurais.

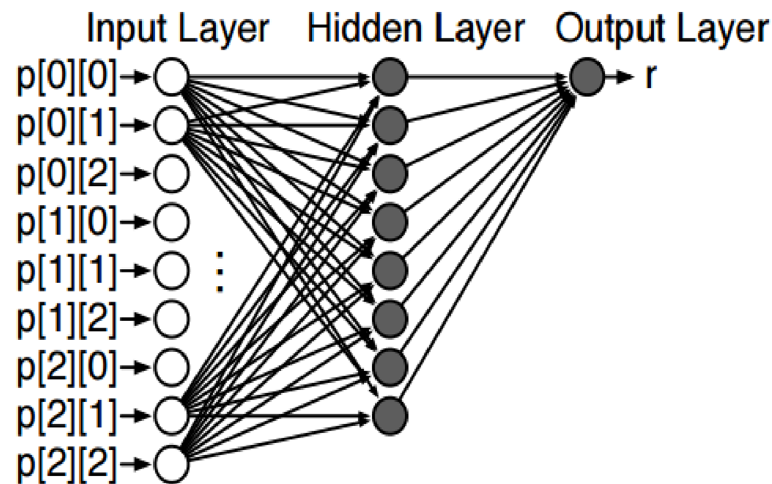
2.1 Redes Neurais Artificiais (Multilayer Perceptron)

Redes neurais artificiais são modelos computacionais inspirados no sistema nervoso central de um animal, em particular o cérebro, as quais são capazes de realizar tanto aprendizado de máquina quanto reconhecimento de padrões (HERTZ et al., 1991). O primeiro trabalho relacionado com o desenvolvimento de um modelo matemático para imitar o sistema nervoso com redes neurais foi publicado em 1943 por McCulloch e Pitts (MCCULLOCH; PITTS, 1943).

Como mostrado na Figura 2.1~~Figura 2.1~~, em alto nível, uma rede neural é um conjunto de neurônios artificiais interligados através de pesos adaptáveis e divididos em camadas: de

entrada, que recebem estímulos externos; internas ou ocultas, utilizadas para processamento; e de saída, que se comunicam com o exterior. A forma de organizar neurônios em camadas é conhecida como *Multi Layer Perceptron* (MLP) e foi concebida para resolver problemas complexos.

Figura 2.1: Exemplo de uma rede neural dividida em camadas de entrada (Input Layer), interna (Hidden) e saída (OutPut).



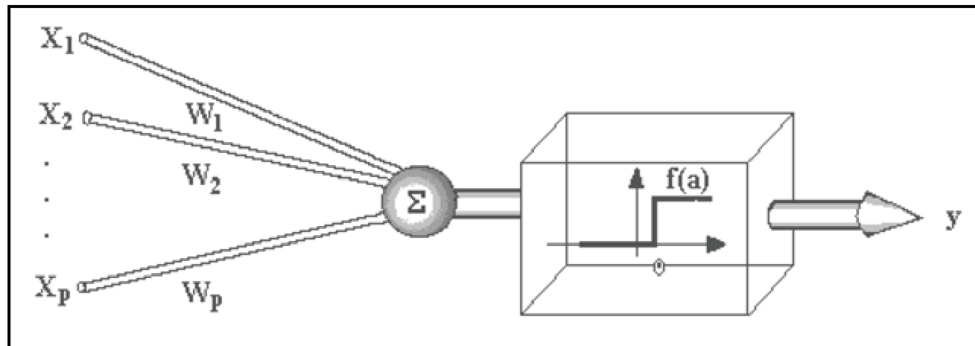
Fonte: (ESMAEILZADEH et al., 2012a)

Uma unidade de processamento, ou neurônio, somente faz operações em sua unidade local, com devidos valores recebidos pelas conexões de outros neurônios. Sendo assim, o aprendizado de uma rede neural se dá através da interação entre as unidades de processamento.

A operação de cada unidade de processamento, proposta por McCulloch e Pitts em 1943, é mostrada na Figura 2.2~~Figura 2.2~~, onde:

- Sinais são aplicados à entrada;
- Cada sinal é multiplicado por um peso que irá influenciar na saída de cada unidade;
- Uma função de ativação é calculada sobre a soma dos sinais multiplicados pelos pesos, o que produz um nível de atividade;
- Se o nível estiver acima de um certo limite (*threshold*) a unidade irá produzir uma resposta.

Figura 2.2: Esquema de unidade McCulloch – Pitts.



Fonte: (MCCULLOCH; PITTS, 1943)

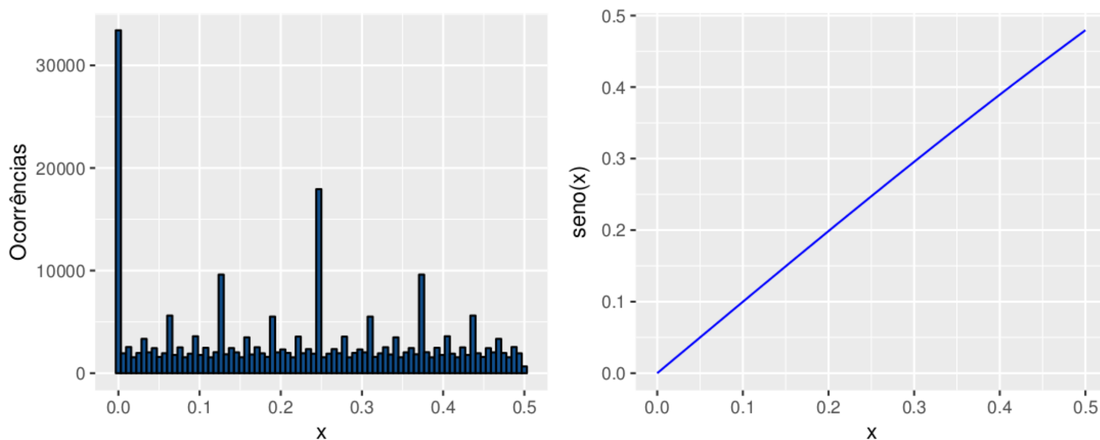
A propriedade mais importante em uma rede neural é a capacidade de aprendizado do seu ambiente. Isto é feito com o ajuste de seus pesos, através de um processo chamado de treinamento. Existem muitos tipos de algoritmos de aprendizado, que diferem na forma como os pesos são modificados. Em técnicas de aprendizado supervisionado, que tipicamente são usadas para treinar redes do tipo MLP, são apresentados um conjunto de treinamento consistindo de pares de entradas e correspondentes saídas que a rede deve aprender. O algoritmo mais comum é conhecido como *backpropagation* (RUMELHART et al., 1985). O processo de treinamento é iterativo, de forma que os pesos são ajustados a cada iteração para diminuir o erro. O algoritmo prossegue até que um máximo de iterações seja atingido ou a taxa de erro médio por iteração seja menor que um *threshold* previamente determinado. Um estudo aprofundado sobre algoritmos de aprendizado e estruturas de redes neurais pode ser encontrado em (FREEMAN; SKAPURA; MITCHELL, 1997; THEODORIDIS; KOUTROUMBAS, 2003; DUDA et al., 2012).

2.2 Trabalhos relacionados

Como já discutido, computação aproximativa surgiu como um novo paradigma para modelar sistemas com melhorias relacionados à performance computacional e consumo energético, baseado na observação de que algumas aplicações apresentam tolerância ou pela própria natureza em que apresentam resultados imprecisos. Esta característica pode surgir de muitos fatores como ruídos nos dados de entradas (em filtros e redes neurais, por exemplo), habilidade limitada do ser humano de percepção (para vídeo e áudio) e cálculos

probabilísticos (em aplicações para inteligência artificial). Uma discussão mais detalhada é encontrada em (HAN; ORSHANSKY, 2013). É importante lembrar que as principais aplicações atuais de alta performance estão exatamente neste domínio, como *machine learning* e processamento de áudio e vídeo. Para estas aplicações, é observado que regiões de código frequentemente executadas apresentam um comportamento similar e que execuções com entradas semelhantes normalmente produzem saídas semelhantes.

Figura 2.3: Número de ocorrências para cada entrada quando a função seno é executada na aplicação FFT e função aproximada.



Fonte: O autor

Para ilustrar este princípio, é apresentado na Figura 2.3~~Figura 2.3~~ o comportamento da função seno na aplicação *Fast-Fourier Transform* (FFT), a qual é amplamente utilizada em aplicações para processamento de sinais. No lado direito, é apresentado o gráfico da função seno e no lado esquerdo a frequência de cada chamada da função. Claramente, algumas entradas são mais frequentes que outras e, como a função tem comportamento relativamente regular, a substituição de uma entrada por outra similar irá introduzir apenas uma pequena margem de erro no resultado final. Embora esta demonstração seja simples, no decorrer deste trabalho será apresentado que este mesmo comportamento surge também em outras aplicações. Assim, será mostrado que, com o armazenando de uma quantidade finita de valores de entrada e saída em uma tabela, é possível substituir toda execução da região por uma busca em tabela que dinamicamente seleciona o valor de saída mais adequado a fim de minimizar a degradação na qualidade final.

A computação aproximativa pode ser implementada em camadas distintas no nível sistêmico. Trabalhos anteriores (XU et al., 2016) classificam abordagens para computação aproximativa nas seguintes três camadas: camada de algoritmos, arquitetura e circuito.

2.2.1 Camada de algoritmo

Neste caso, as alterações para computação aproximativa ocorrem diretamente na modelagem da especificação, sem nenhum suporte do hardware. Trabalhos como (MENGTE et al., 2010; NIU et al., 2011; MISAILOVIC et al., 2012; RENGANARAYANA et al., 2012) apresentam um estudo para eliminar o overhead em sincronização de programas *multi-thread.* Já em (HOFFMANN et al., 2011), é apresentada uma forma de identificar e ajustar parâmetros que consigam controlar a qualidade final da aplicação. (MISAILOVIC et al., 2010; SIDIROGLOU-DOUSKOS et al., 2011) apresenta uma técnica capaz de trocar qualidade por performance, transformando execuções de *loops* em um subconjunto de suas iterações, chamada de *loop perforation*.

Figura 2.4: Técnica de Loop Perforation.

```
for (i = 0; i < b; i++) { ... }
                ↓
for (i = 0; i < b; i += n) { ... }
```

Fonte: (SIDIROGLOU-DOUSKOS et al., 2011)

Figura *2.4*~~Figura *2.4*~~ apresenta a implementação em alto nível da técnica de *loop perforation*, cujo objetivo é reduzir a quantidade de operações que o programa necessita para produzir um resultado. A técnica pode ser utilizada em *loops* que não causem falhas na execução e que o resultado final não seja tão aproximado a ponto de não ser aceitável. Os resultados apresentados pelo autor mostram que é possível acelerar aplicações, bem como manter uma porcentagem de erro em torno de 10%, como apresentado na Figura 2.5~~Figura 2.5~~.

Figura 2.5: Resultado da aplicação da técnica Loop Perforation x(y %), onde x é o speedup e y a porcentagem de erro.

Application	Training				Production			
	2.5%	5%	7.5%	10%	2.5%	5%	7.5%	10%
x264	2.38 (2.5%)	2.66 (5%)	3.17 (6.53%)	3.25 (9.31%)	2.34 (5.15%)	2.53 (6.08%)	3.12 (8.72%)	3.19 (10.3%)
bodytrack	3.44 (2.23%)	6.32 (4.36%)	6.89 (6.19%)	6.89 (6.19%)	2.70 (4.00%)	4.93 (6.12%)	4.811 (6.58%)	4.811 (6.58%)
swaptions	5.08 (1%)	5.08 (1%)	5.08 (1%)	5.08 (1%)	5.05 (0.2%)	5.05 (0.2%)	5.05 (0.2%)	5.05 (0.2%)
ferret	1.02 (0.2%)	1.03 (4%)	1.03 (4%)	1.16 (10%)	1.002 (0.15%)	1.02 (0.23%)	1.02 (0.23%)	1.07 (7.90%)
canneal	1.14 (4.38%)	1.18 (4.43%)	1.913 (7.14%)	1.913 (7.14%)	1.14 (4.38%)	1.14 (4.38%)	1.46 (7.88%)	1.46 (7.88%)
blackscholes	33 (0.0%)	33 (0.0%)	33 (0.0%)	33 (0.0%)	28.9 (0.0%)	28.9 (0.0%)	28.9 (0.0%)	28.9 (0.0%)
streamcluster	5.51 (0.54%)	5.51 (0.54%)	5.51 (0.54%)	5.51 (0.54%)	4.87 (1.71%)	4.87 (1.71%)	4.87 (1.71%)	4.87 (1.71%)

Fonte: (SIDIROGLOU-DOUSKOS et al., 2011)

2.2.2 Camada de arquitetura

Neste caso, a aplicação é acelerada a partir de alguma modificação em hardware e, geralmente, com suporte do compilador. Diversos trabalhos aplicam técnicas de aproximação em memória, na qual a ideia geral é gastar menos energia em operações para armazenar ou acessar informações (Chang, Mohapatra, and Roy 2011; Liu et al. 2009; Lucas et al. 2014; Shoushtari, BanaiyanMofrad, and Dutt 2015).

Em (SHOUSHTARI et al., 2015), os autores propõem remover ou diminuir a proteção (do inglês guard-banding) tipicamente utilizada nas memórias para diminuir os efeitos de variações de manufatura, a fim de garantir 100% de integridade da informação, considerando que 100% de precisão não é necessário para todas as aplicações. O autor apresenta a possibilidade de diminuição do consumo de energia estática em torno de 74%, utilizando esta abordagem.

Em (LIU et al., 2009), os autores apresentam um estudo considerando que, nos dispositivos móveis, mesmo quando estão em modo suspenso (do inglês *sleep mode*), uma quantidade de energia significante é consumida, já que os dados da memória DRAM precisam ser atualizados periodicamente. Assim, os autores propõem uma técnica chamada Flicker, na qual permite desenvolvedores especificar dados críticos e não críticos de um programa. Os dados críticos são atualizados na mesma frequência, enquanto os não críticos são atualizados com uma frequência menor, em que se traduz em economia de energia ao custo de um aumento modesto de dados corrompidos (indiretamente adicionando imprecisão no resultado final).

Em (LUCAS et al., 2014), os autores apresentam uma extensão ao trabalho Flicker, como citado acima, chamada de Sparkk, na qual propõem alterações para que seja possível a atualização de informações “manualmente” e em partes separadas da memória. Sparkk utiliza essa abordagem para ter um controle maior sobre as memórias (granularidade fina), a fim de obter ganhos computacionais.

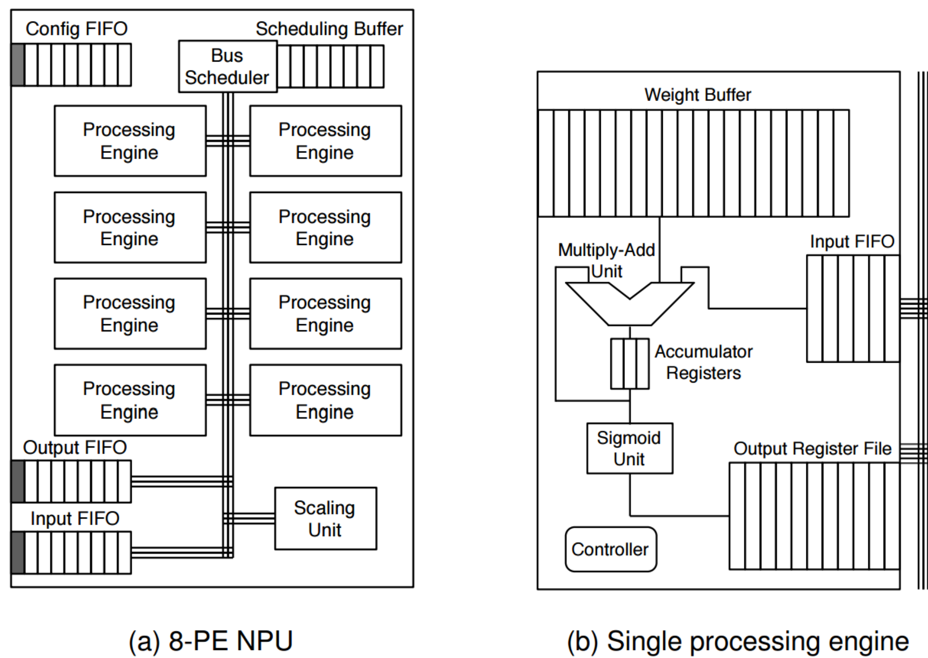
Em (CHANG et al., 2011), os autores apresentam uma abordagem com relação ao escalonamento de voltagem em memórias SRAM. Atuando principalmente na compressão de dados através do padrão MPEG-4, a proposta é um array de memória híbrido, sendo uma mistura de SRAMs de 6 e 8 transistores. A premissa fundamental do trabalho é que sistema visual humano é mais sensível à pixels com uma luminância de ordem superior. Desta forma, é implementada uma política de armazenamento em que os pixels de maior luminância são armazenados em uma memória robusta de 8T, enquanto os pixels com menor luminância são armazenadas em memórias convencionais de 6T, facilitando um aumento agressivo de escalonamento de voltagem. O autor apresenta a possibilidade de obter uma diminuição de 32% de consumo de energia com memórias híbridas, ao custo de uma degradação insignificante na saída do algoritmo.

A camada de arquitetura também envolve alterações no algoritmo, o qual é ajustado conforme a estrutura do hardware. A abordagem mais comum consiste em desenvolver um hardware capaz de virtualizar uma RN para executar regiões de código com maior eficiência (Chen et al. 2012; Grigorian, Farahpour, and Reinman 2015; McAfee and Olukotun 2015; Moreau et al. 2015; Yazdanbakhsh et al. 2015). Neste caso, a RN é utilizada essencialmente como um acelerador que substitui a execução original do código por uma execução aproximada da rede neural.

Já (ESMAEILZADEH et al., 2012a) avalia o desempenho de uma abordagem baseada em aprendizado com o objetivo de aproximar aplicações. O autor apresenta uma transformação que seleciona e treina uma rede neural a fim de substituir regiões de códigos passíveis de serem aproximadas, chamada de transformação de Parrot. Após a fase de treinamento, o compilador substitui o código original pela invocação de um hardware dedicado, chamado de unidade de processamento neural (NPU, do inglês *neural processing unit*). Como apresentado na Figura *2.6*~~Figura *2.6*~~, a NPU contém oito unidades de processamento idênticas, a fim de executar a rede neural. Cada neurônio da RN é representado por uma das oito unidades de processamento. Quando uma rede neural possui

mais de oito neurônios, estes são calculados conforme a configuração de cada RN através da transição de informações entre neurônios físicos e memórias da NPU.

Figura 2.6: Hardware reconfigurável da transformação de Parrot.



Fonte: (ESMAEILZADEH et al., 2012a)(ESMAEILZADEH et al., 2012a)

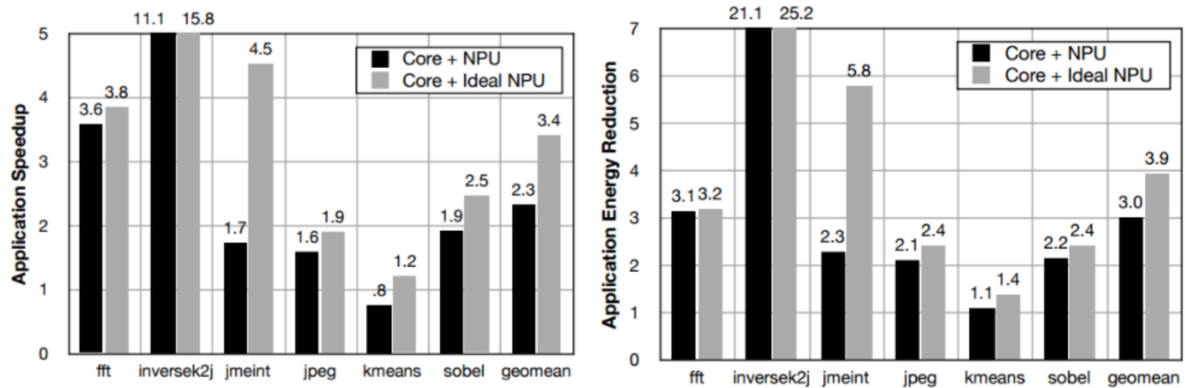
A Figura 2.7~~Figura 2.7~~ apresenta os benchmarks utilizados para validação da técnica, bem como a topologia escolhida para cada rede neural e o erro obtido através da transformação de Parrot. Dentre os resultados, o principal aponta para o aumento de desempenho da aplicação bem como a diminuição do consumo energético, como apresentado na Figura 2.8~~Figura 2.8~~.

Figura 2.7: Benchmarks utilizados na avaliação da técnica e taxa de erro.

	Description	Type	Evaluation Input Set	# of Function Calls	# of Loops	# of ifs/ elses	# of x86-64 Instructions	Training Input Set	Neural Network Topology	NN MSE	Error Metric	Error
fft	Radix-2 Cooley-Tukey fast Fourier	Signal Processing	2048 Random Floating Point Numbers	2	0	0	34	32768 Random Floating Point Numbers	1 -> 4 -> 4 -> 2	0.00002	Average Relative Error	7.22%
inversek2j	Inverse kinematics for 2-joint arm	Robotics	10000 (x,y) Random Coordinates	4	0	0	100	10000 (x,y) Random Coordinates	2 -> 8 -> 2	0.00563	Average Relative Error	7.50%
jmeint	Triangle intersection detection	3D Gaming	10000 Random Pairs of 3D Triangle Coordinates	32	0	23	1,079	100000 Random Pairs of 3D Triangle Coordinates	18 -> 32 -> 8 -> 2	0.00530	Miss Rate	7.32%
jpeg	JPEG encoding	Compression	220x200-Pixel Color Image	3	4	0	1,257	Three 512x512-Pixel Color Images	64 -> 16 -> 64	0.00890	Image Diff	9.56%
kmeans	K-means clustering	Machine Learning	220x200-Pixel Color Image	1	0	0	26	50000 Pairs of Random (r, g, b) Values	6 -> 8 -> 4 -> 1	0.00169	Image Diff	6.18%
sobel	Sobel edge detector	Image Processing	220x200-Pixel Color Image	3	2	1	88	One 512x512-Pixel Color Image	9 -> 8 -> 1	0.00234	Image Diff	3.44%

Fonte: (ESMAEILZADEH et al., 2012a)

Figura 2.8: Melhorias em performance e consumo energético.



Fonte: (ESMAEILZADEH et al., 2012a)

Os trabalhos (YAZDANBAKHSH et al., 2015) e (MOREAU et al., 2015) mantêm a mesma metodologia apresentada em (ESMAEILZADEH et al., 2012a), embora o primeiro execute RN em GPU, o segundo utiliza FPGA.

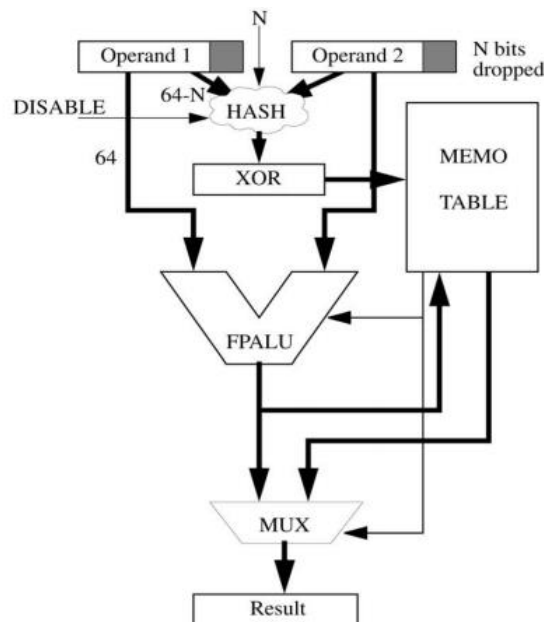
2.2.3 Camada de circuito

Nesta situação, o hardware é modificado transparentemente para a aplicação. Trabalhos anteriores (VERMA et al., 2008; ZHU et al., 2009; GUPTA et al., 2011, 2013; KEDEM et al., 2011; KAHNG; KANG, 2012; MIAO et al., 2012; WEBER et al., 2013; YE et al., 2013; SHAFIQUE et al., 2015), com foco em processadores VLSI (do inglês, *Very-Large-Scale Integration*), utilizam células imprecisas, em que somadores com uma quantidade menor de transistores são utilizados. Esta abordagem introduz erros. Entretanto, quando utilizada somente para os bits menos significativos (LSB, do inglês Least Significant Bits) (exemplo, um somador), a qualidade pode continuar sendo aceitável e com ganhos em performance computacional. Outra técnica é *voltage over-scaling* (MOHAPATRA et al., 2011), na qual introduz-se erros de temporização (tempo de propagação de um sinal para chegar de um ponto A até um ponto B) ao custo de aproximação, com ganhos significativos em energia . Uma

estratégia para aproximação em unidade de ponto flutuante é apresentada em (TONG et al., 2000; YEH et al., 2007), na qual propõe-se adaptar a largura da mantissa.

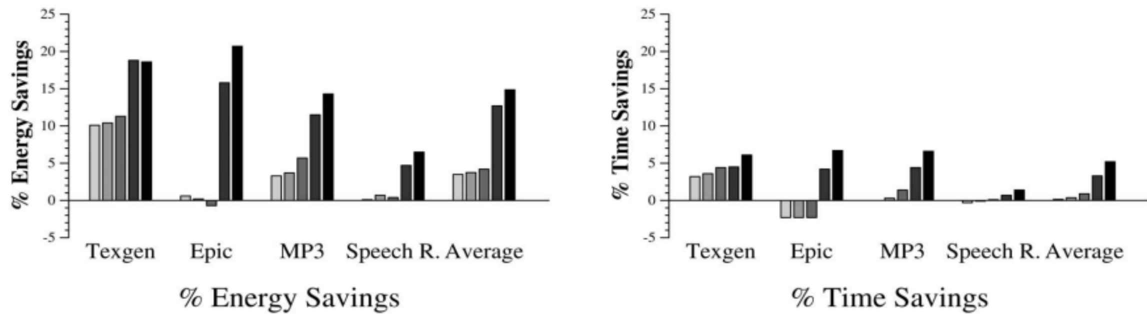
Já (ALVAREZ et al., 2005) apresentam a técnica denominada *Fuzzy memoization*, que é aplicada em unidades de ponto flutuante a fim de obter melhorias em performance e energia, através do reúso de instâncias de operações já realizadas. Performance e consumo energético podem ser melhorados ao custo de perda de precisão em algumas operações. Esta técnica expande a ideia de que entradas similares produzem saídas similares. A Figura *2.9*~~Figura *2.9*~~ apresenta o esquema utilizado para implementar a técnica, na qual N bits menos significativos são removidos da mantissa antes de serem utilizados para acessar uma tabela que contém o resultado de execuções anteriores. Com isso, não somente os operandos iguais, mas também os similares, irão gerar um *hit* na tabela. Os resultados apresentados pelos autores mostram que a técnica é capaz de acelerar aplicações, bem como diminuir o consumo energético, como apresentado na Figura *2.10*~~Figura *2.10*~~.

Figura 2.9.: Configuração do hardware para uma LUT sequencial.



Fonte: (ALVAREZ et al., 2005)

Figura 2.10: Resultados da técnica fuzzy memoization.



Fonte: (ALVAREZ et al., 2005)

Em (GRIGORIAN et al., 2015), o autor propõe uma arquitetura heterogênea, composta por RN (aproximativo) e aceleradores precisos. A RN é utilizada para aceleração quando a qualidade pode ser trocada por performance e eficiência energética. Por outro lado, esta arquitetura permite ao usuário especificar um limite mínimo de qualidade para cada aplicação. Se este limite for atingido, a aplicação migra para o acelerador preciso, mantendo a qualidade como esperada.

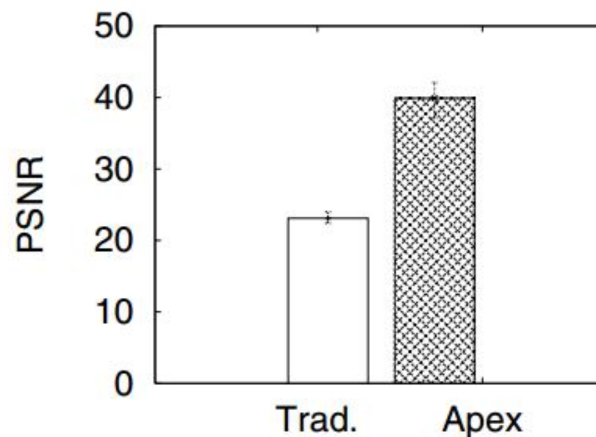
Mesmo que todos os trabalhos citados acima tenham apresentado resultados positivos utilizando RN como um paradigma de aproximação, o objetivo principal de ter reduções agressivas em consumo energético não é alcançado. Isto acontece, pois, a execução de uma RN contém uma grande quantidade de dados que devem ser lidos da memória e uma grande quantidade de operações em ponto flutuante que devem ser executadas.

2.2.4 Outros trabalhos

Enquanto otimizações nos níveis da arquitetura e do software dominam o campo de propostas para aproximação, alguns trabalhos recentes exploram os mesmos *trade-offs* em outros componentes. Uma pesquisa recente apresentou BlinkDB(AGARWAL et al., 2013), uma base de dados que pode retornar valores de uma query com um certo grau de imprecisão requerido pela aplicação. Em sistemas distribuídos ou supercomputadores, técnicas de aproximação visam a evitar redundância, como apresentado em (HO et al., 2012).

O trabalho (SEN et al., 2010) apresenta oportunidades claras de melhorias em comunicação via rede, explorando a característica de que comunicação sem fio é propensa a erros. O autor defende que quando uma informação é decodificada erroneamente pelo receptor, esta é uma informação aproximada. Baseado nesta propriedade, o trabalho propõe um método que explora a estrutura destes erros para fornecer nativamente uma proteção dos bits transferidos, sem nenhum custo adicional de rede. O trabalho enfatiza que a comunicação aproximativa é particularmente útil em aplicações de mídia, as quais podem beneficiar-se do método proposto. Desta forma, o autor implementa um sistema *end-to-end* de entrega de mídia, denominado de Apex. A Figura *2.11*~~Figura *2.11*~~ apresenta que Apex é capaz de prover uma melhoria de qualidade de vídeo de 5 a 20 dB PSNR (do inglês, Peak signal-to-noise ratio).

Figura 2.11: Resultados apresentando a diferença entre a comunicação tradicional e a comunicação aproximada.



Fonte: (SEN et al., 2010)

Embora todos os trabalhos relacionados apresentem resultados positivos, tanto em aumento de performance, diminuição do consumo energético e melhorias em qualidade do resultado, utilizando técnicas de clusterização de dados e armazenamento em memória, potencialmente tem-se ganhos semelhantes ou superiores aos citados acima. Esta dissertação estende os trabalhos relacionados, pois a utilização de computação aproximativa como uma forma de alcançar melhorias em performance, área ou energia é diretamente atrelada com a proposta deste trabalho. Detalhes da técnica proposta serão apresentadas no capítulo 3.

3 EDCAC

EDCAC (do inglês, *Enhanced Data Clustering for Approximate Computing)* é um framework capaz de substituir a computação de um trecho de código pré-selecionado por uma consulta em tabela controlada por software com o propósito de melhorar a performance e o consumo energético. Para isto, o framework reúsa em tempo de execução o resultado mais próximo ao original, baseado nas entradas do trecho de código (sem modificação) e em saídas pré-computadas. Nesta seção, será apresentada a proposta de implementação da técnica de reúso aproximativo de funções ou trechos de códigos.

Nota-se, também, que o EDCAC, através de aproximação, faz com que uma técnica de reúso consiga ser utilizada de maneira eficiente. Embora técnicas de reúso em geral sejam promissoras, sempre existiu um problema de execução: as tabelas necessárias para o armazenamento de dados de entrada e saída se tornam excessivamente grandes e, por consequência, lentas e caras. Como será demonstrado, é possível reduzir o custo destas tabelas através de aproximação.

3.1 A Técnica

Como discutido previamente, a grande maioria das soluções relacionadas com computação aproximativa utilizam redes neurais, pois elas têm a capacidade de aprendizado de seu ambiente, sendo este supervisionado ou não. Embora seja uma técnica promissora e amplamente difundida na indústria e na área acadêmica, experimentos realizados discutem a eficácia da aplicação destas técnicas e sugerem diversos questionamentos, tais como:

Técnicas utilizando redes neurais são as melhores soluções para programas aproximativos?

Redes neurais oferecem o melhor aproveitamento de recursos computacionais?

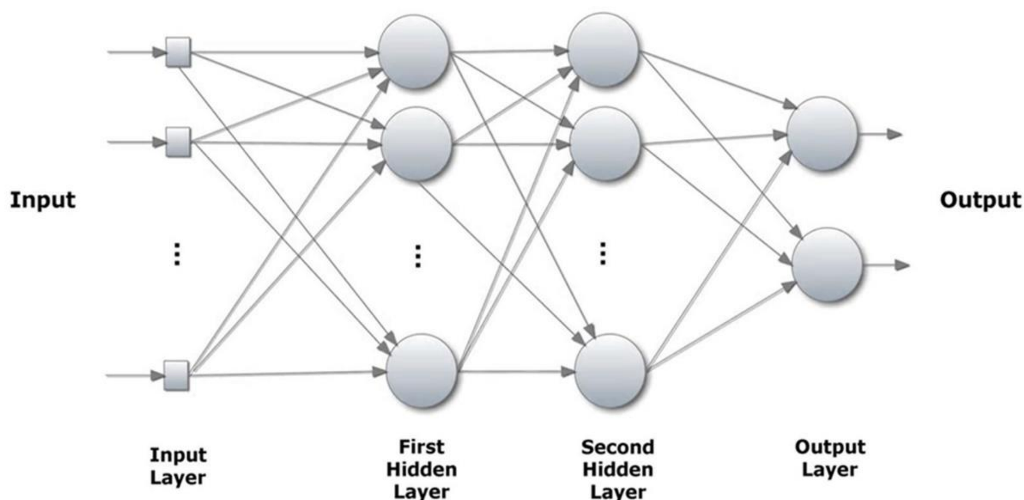
Como já discutido, para o treinamento da rede neural, é utilizada uma quantidade finita de pares de entrada e saída de cada região de código. Estes dados podem ser facilmente armazenados em uma tabela estática e reusados quando necessário. Porém, neste caso, a tabela se tornaria enorme e ineficaz, pois a quantidade de dados armazenados nela seria gigantesca. Em contrapartida, a técnica proposta – que será discutida posteriormente –

consegue diminuir o tamanho da tabela a ponto da quantidade de dados nela contidos não ser mais um problema.

Na prática, a execução de uma rede neural é basicamente constituída de inúmeras multiplicações. No caso de uma rede Multilayer Perceptron, a quantidade de multiplicações é proporcional à quantidade de neurônios em cada camada. Assim, todo e qualquer neurônio de uma camada contém uma conexão com todos os neurônios da camada seguinte, contendo uma multiplicação em cada conexão, como mostrado na Figura 3.1~~Figura 3.1~~.

Considerando o fato que existem várias operações de carga-multiplicação-acumula (do inglês, *load-multiply-acumulate*) em uma rede Multilayer Perceptron, o que queremos é substituir todas estas por um simples acesso à memória através de uma *lookup table* para obter potenciais ganhos em performance, consumo energético e área.

Figura 3.1: Apresentação de uma rede neural Multilayer Perceptron, enfatizando a quantidade de multiplicações em cada neurônio.



Fonte: (FOROOZESH et al., 2013)

Em alto nível, uma aplicação que utiliza esta abordagem substitui a execução da função original pela busca otimizada do dado aproximado na tabela de reúso. Primeiramente, a aplicação executa no processador principal e o hardware adicional é invocado executando a técnica aproximativa quando necessário (i.e., no momento da execução do *hotspot*). Na *Figura 1.1*~~*Figura 1.1*~~ é mostrada uma visão geral da abordagem, na qual são definidas 3 etapas principais: 1) fase de programação, na qual o programador anota o código de interesse; 2) compilação, em que o compilador seleciona os dados a serem aproximados e substitui o

código original pelo reúso da tabela; 3) execução, na qual a nova aplicação é executada com o reúso aproximado de dados.

3.1.1 Fase de Programação

Durante a modelagem do software, o programador necessita selecionar as funções que são suscetíveis a serem aproximadas, já que tolerância à aproximação é uma propriedade semântica do programa, sendo de responsabilidade do programador garantir que a execução aproximada não comprometa a confiabilidade da aplicação como um todo. Esta é uma prática comum na literatura relacionada à computação aproximativa (STANLEY-MARBELL; MARCULESCU, 2006; KRUIJF, DE et al., 2010). Mais detalhes sobre o modelo de programação podem ser encontrados na seção 3.2.

Neste trabalho, a transformação do código original para uma representação aproximada é aplicada a funções inteiras. Para identificar a função a ser aproximada, são utilizadas anotações no código fonte, similares à forma como foram utilizadas em (ESMAEILZADEH et al., 2012a) (e.g., C++11[[annotation]]).

Nesta etapa o programador anota o código fonte, selecionando qual região é tolerante a aproximação. Estas anotações vão informar ao compilador a região a ser aproximada bem como indicar as entradas e saídas, como mostrado na Listagem *3.1*~~Listagem *3.1*~~.

Listagem 3.1: Sintaxe da anotação pragma.

```
1.  #pragma approximation(input, "fft", [1] x)
2.  void fftSinCos(float x, float* s, float* c) {
3.      *s = sin(-2 * PI * x);
4.      *c = cos(-2 * PI * x);
5.  }
6.  #pragma approximation(output, "fft",
7.  [2], [s, c])
```

O programador é responsável por assegurar que a função seja pura (mais detalhes na próxima subseção) e apresente tamanho fixo de entradas e saídas (caso seja um ponteiro, este deve apontar para um arranjo com tamanho fixo, e caso a função necessite retornar ou receber múltiplos valores, é possível receber e retornar um array de tamanho fixo ou uma estrutura em C). Esta é a única etapa em que é necessária intervenção humana. Após, a transformação para o código aproximativo é automática e transparente, não necessitando de nenhuma intervenção.

3.1.1.1 Critérios para escolha da região aproximada

A região do código a ser aproximada deve satisfazer três critérios simples, porém de grande importância para uma boa performance: o trecho a ser aproximado deve 1) ser executado frequentemente; (i.e., ser um *hotspot*); 2) tolerar imprecisões em sua computação; 3) apresentar entradas e saídas bem definidas (e.g., tamanhos fixos de entradas e saídas).

A transformação pode ser aplicada a uma grande quantidade de código, desde pequenas funções a algoritmos inteiros. A região a ser aproximada não deve ter nenhuma interferência externa, modificando o estado atual do programa. A região do código pode conter chamadas de funções, loops e fluxos complexos. Neste último caso, maiores ganhos são possíveis, pois substitui-se um código complexo e caro de ser executado por uma estrutura simples e barata.

Um bom exemplo de aplicação é a FFT, que computa a transformada de Fourier e é largamente utilizada em processamento digital de sinais. O código alvo deste benchmark é composto principalmente por operações aritméticas e um fluxo com pouca complexidade, porém é executado inúmeras vezes, como mostrado na Listagem *3.2*~~Listagem *3.2*~~. Observando a linha 10, nota-se que o código é executado inúmeras vezes durante o tempo de vida da aplicação. Neste caso, o hotspot em evidência é um bom candidato para aplicar a técnica de computação aproximativa.

Listagem 3.2: Implementação do *hotspot* original do benchmark FFT.

```
1.  for(i = 0, N = 1 << (i + 1); N <= K ; i++, N = 1 << (i + 1))
2.      {
3.          for(j = 0 ; j < K ; j += N)
4.          {
5.              step = N >> 1 ;
6.              for(k = 0; k < step ; k++)
7.              {
8.                  arg = (float)k / N ;
9.
10.                 fftSinCos(arg, &fftSin, &fftCos);
11.
12.                                 ...
13.
14. void fftSinCos(float x, float* s, float* c) {
15.     *s = sin(-2 * PI * x);
16.     *c = cos(-2 * PI * x);
17. }
```

Cabe ao programador assegurar que o código executado no coprocessador não influencie a execução da aplicação como um todo, pois a taxa de erro pode ser muito alta ou podem ocorrer erros catastróficos. Para ser possível aproximar uma função, as restrições acima citadas devem ser respeitas. As mesmas podem ser identificadas estaticamente, como visto na Listagem *3.2*~~Listagem *3.2*~~ (onde o trecho de código respeita todas restrições impostas, recebendo e retornando valores fixos, não envolvendo nenhuma interferência externa e executado inúmeras vezes).

3.1.2 Compilação

O framework processará o código anotado na etapa anterior e automaticamente produzirá um código binário com a região de código substituída por *lookups tables* em uma

memória dedicada. Uma vez o código anotado, o compilador irá aplicar a aproximação em três etapas, (1) coleta e observação; (2) aproximação de dados; (3) geração do novo binário.

3.1.2.1 Coleta e Observação

A fim de analisar os dados da região alvo, a aplicação é executada com entradas fornecidas pelo programador, monitorando as entradas e saídas da região selecionada e armazenando-as localmente. Embora estas informações possam gerar uma tabela potencialmente grande, o que tornaria esta abordagem ineficaz, o próximo passo utiliza estas informações para reduzir o tamanho da tabela.

Na primeira fase, o compilador irá coletar valores de entrada da região a ser aproximada. Estes pares de entrada permitem o compilador identificar os dados mais significativos para a região critica.

A cada nova chamada da região crítica, o compilador armazena localmente os dados coletados executando o programa repetidamente utilizando entradas de testes selecionadas pelo programador. A saída desta etapa é um conjunto de dados de entrada contendo todos os valores originais do código anotado.

As entradas obtidas pelo compilador fazem parte de uma suíte de treinamento selecionadas randomicamente. O exemplo da aplicação FFT mostrado na Listagem *3.1*~~Listagem *3.1*~~ provê 11264 observações. Estas amostragens serão utilizadas no próximo passo a fim de agrupar e identificar os dados mais significativos através de *clusterização*, como será discutido posteriormente.

3.1.2.2 Aproximação dos dados

Como já observado, tabelas muito grandes são caras e lentas. O objetivo, aqui, é diminuir o tamanho das tabelas ao custo de acurácia utilizando o algoritmo K-means, explicado na sequência. O framework EDCAC explora a propriedade que entradas similares têm a tendência de produzir saídas similares.

O K-means é um algoritmo bem conhecido de agrupamento de dados, o qual é capaz de particionar N observações em K clusters. É um método simples de classificar dados em

grupos definidos *a priori*. O algoritmo agrupa observações próximas, representando-as através de uma média dos pontos que a ele pertencem, resultando em uma divisão do espaço de dados em um Diagrama de Voronoi (SACK; URRUTIA, 1999).

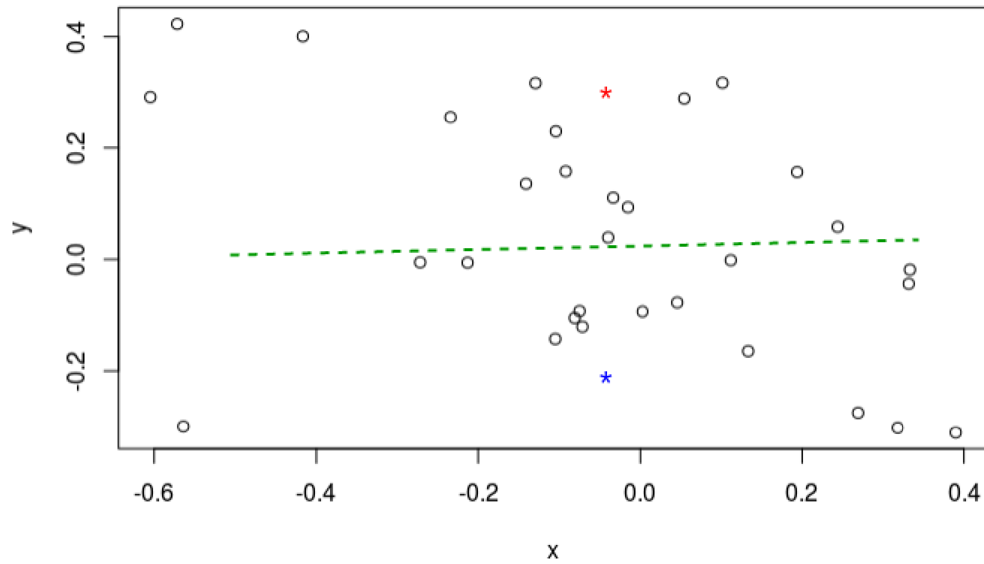
O algoritmo depende de um esquema iterativo, que pode ser resumido pelos seguintes passos:

1. Defina K elementos representando os clusters. Uma das formas de iniciar o processo é inserir aleatoriamente os valores de cada K, para que em seguida seja possível iniciar as iterações;
2. Atribua cada uma das amostragens ao cluster mais próximo, determinado por uma métrica de distância, (e.g., Distância Euclidiana);
3. Após atribuir todas as amostragens, é recalculada a nova posição de cada K clusters através da média dos valores nele contido;
4. Repita o passo 2 e 3 até convergir (i.e., os centróides não se movimentarem mais). Também é possível definir uma quantidade máxima de iterações, em casos que a precisão possa interferir nos resultados.

Para entender melhor o funcionamento do algoritmo, será apresentado um exemplo passo a passo que mostra como os dados convergem para um resultado. Neste exemplo, o valor K será igual a dois, que se traduz em dois centróides (pontos centrais de cada grupo), desta forma indicando a similaridade entre as amostragens.

A Figura *3.2*~~Figura *3.2*~~ apresenta dois pontos aleatórios criados no gráfico, um vermelho e outro azul e uma linha tracejada que mostra a divisão dos clusters, sendo assim os pontos acima da linha pertencem ao grupo vermelho e os abaixo da linha pertencem ao grupo azul. Esta imagem apresenta os passos 1 e 2 do algoritmo, em que é definido aleatoriamente dois pontos no gráfico 2D e cada observação é atribuída para o cluster mais próximo.

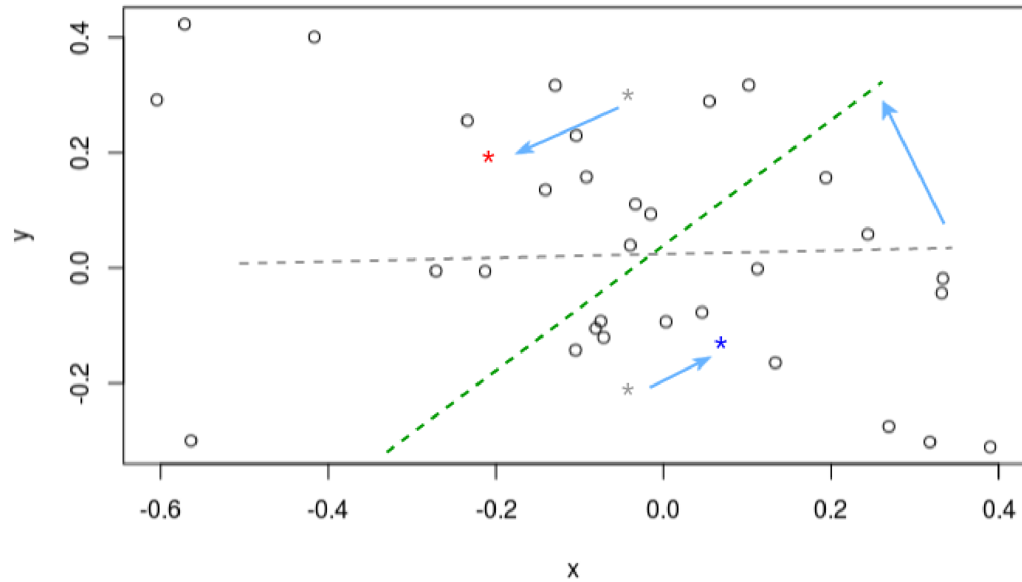
Figura 3.2: Gráfico apresentando dois pontos aleatórios no plano.



Fonte: O autor

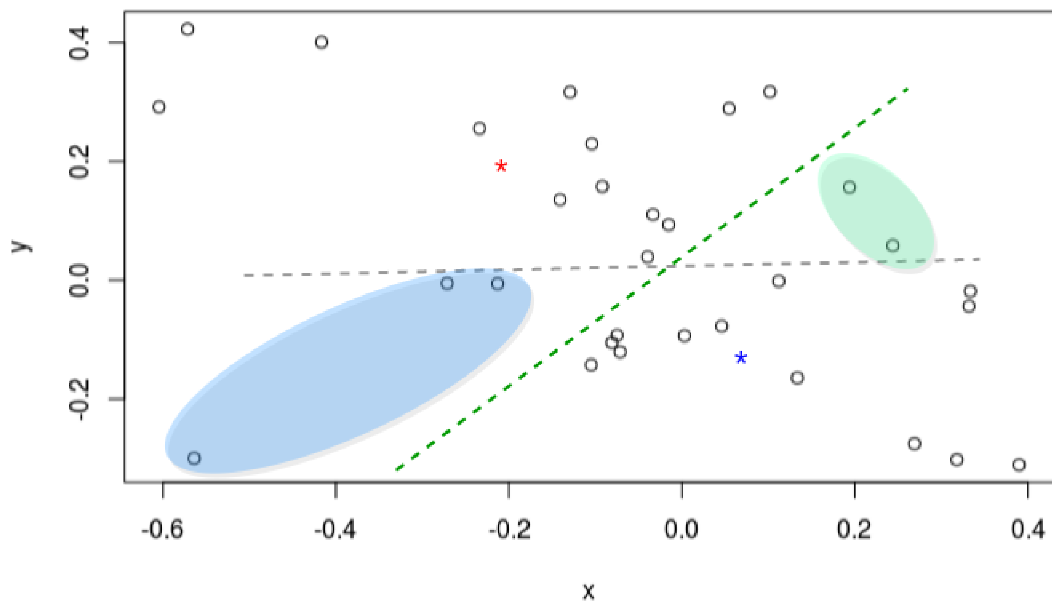
A próxima iteração do algoritmo calcula a média de todas as amostragens atreladas a cada centróide. Com isso, a posição do centróide potencialmente muda para o novo valor a ele atribuído, que é a média dos valores nele contido. A mudança de posição do centróide pode acarretar na mudança de algumas amostragens que fazem parte daquele grupo. Podemos analisar este comportamento na Figura *3.3*~~Figura *3.3*~~ e na Figura *3.4*~~Figura *3.4*~~.

Figura 3.3: Gráfico apresentando a mudança da posição de cada centróide.



Fonte: O autor

Figura 3.4: Gráfico apresentando a mudança de alguns pontos no plano para outro cluster.



Fonte: O autor

Como apresentado na Figura *3.4*~~Figura *3.4*~~, observa-se que após a iteração do cálculo da média de cada centróide alguns pontos mudaram de cluster: os pontos destacados em azul passaram a fazer parte do cluster vermelho e os pontos destacados em verde passaram a fazer parte do cluster azul.

A Figura *3.5*~~Figura *3.5*~~ também apresenta um pseudocódigo do algoritmo *K-means*, em que todas as etapas discutidas acima são apresentadas.

Figura 3.5: Pseudocódigo do algoritmo *K-means* Cluster.

Algorithm 1 K-Means Clustering

```
1: Inicialização dos clusters
2: while i <= I OR Nenhuma mudança entre clusters do
3:     for Todos N elementos do
4:         Adiciona elemento para o cluster mais próximo
5:         Cálculo do centróide de cada cluster
6:     end for
7: end while
```

Fonte: O autor

Não existe uma forma exata de definir qual é o resultado correto do algoritmo k-means, pois depende da maneira que o conjunto de dados é interpretado por cada métrica utilizada e da inicialização randômica de cada K clusters. Da mesma forma não existe uma quantidade exata de clusters. Um estudo mais aprofundado sobre este assunto é encontrado em (RAY; TURI, 1999).

Como uma observação final, o algoritmo k-means pode ser aplicado múltiplas vezes em um conjunto de dados, em um processo semelhante ao *hierarchical clustering* (JOHNSON, 1967), em que cada amostragem é colocada em seu próprio cluster, e depois são identificados os dois clusters mais próximos e combinados em um único cluster. Este processo é repetido até que todas as amostragens estejam em um único cluster. Desta forma, os centróides podem ser adaptados para uma hierarquia de clusters, a fim de reduzir a quantidade de cálculos da distância ao classificar uma nova entrada.

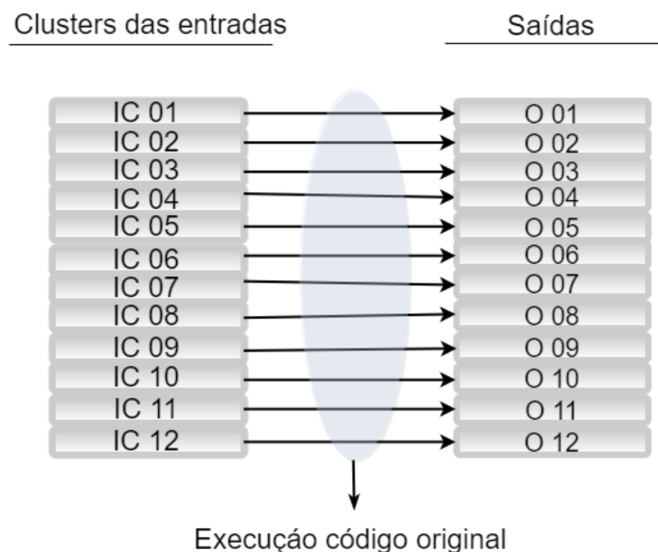
Com base no exposto acima, a hierarquia proposta neste trabalho executa o algoritmo K-means duas vezes, sendo que a segunda execução utiliza os centróides gerados pela primeira execução, desta forma criando um novo conjunto de centróides e gerando uma

hierarquia entre os clusters. A seguir, explica-se como, especificamente, o algoritmo K-means foi aplicado no contexto deste trabalho.

Primeiro, o algoritmo k-means é aplicado ao conjunto de dados gerados pela etapa de *coleta e observação de dados* (nota-se que o conceito de dados de entrada é abstrato – mais especificamente, cada dado de entrada é composto pelo número de variáveis que a função a ser aproximada recebe como entrada). O resultado desta etapa de aproximação é um conjunto de K clusters representando entradas da região do código que são similares aos originais, chamadas de clusters das entradas (IC, do inglês, *input clusters*). Cada IC é representado por seu centróide. Se a função a ser aproximada recebe, por exemplo, três variáveis de entrada, o centróide será constituída por três valores.

Com o centróide de cada IC em mãos, cada respectiva saída é gerada. Para isto, o código original é executado uma única vez para cada IC com seu respectivo centróide como parâmetro de entrada. O resultado desta execução é um mapeamento de ICs para suas saídas correspondentes, as quais chamamos de saída (O, do outputs), apresentadas na Figura *3.6*~~Figura *3.6*~~. Assim, a lista de saída é composta dos K resultados (ou valor de retorno) da função a ser aproximada usando como parâmetro os K centróides de IC, achados anteriormente.

Figura 3.6: Mapeamento dos clusters de entrada para as respectivas saídas.

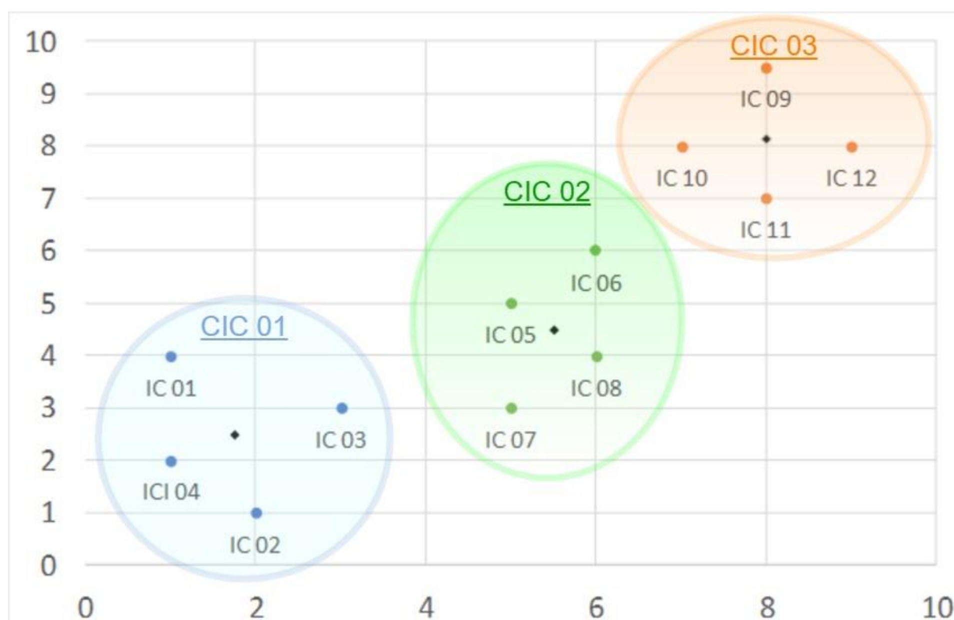


Fonte: O autor

O algoritmo k-means é executado mais uma vez considerando o conjunto de ICs como novo conjunto de dados em um processo conhecido como *Hierarchical clustering* (já discutido previamente) no qual o resultado é um novo conjunto de K´ clusters em uma hierarquia de entradas de dados que estão mapeadas para as a saídas, onde $K' < K$. Estes novos clusters são chamados de cluster de clusters das entradas (CIC, do inglês *clusters of input clusters*). A nova quantidade de clusters gerada nesta etapa está diretamente relacionada com a quantidade de cluster definidos na etapa de geração de ICs.

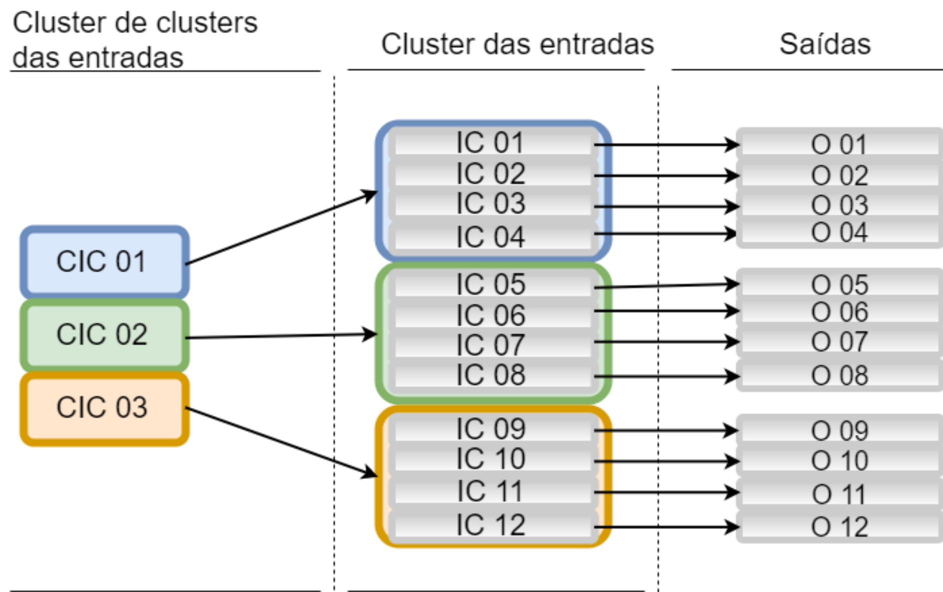
O resultado deste processo é um conjunto de CICs conectados a múltiplos ICs, como mostrado na Figura *3.7*~~Figura 3.7~~ e Figura *3.8*~~Figura 3.8~~. Desta maneira, conseguimos criar uma estrutura que é capaz conectar CICs a uma fração menor de ICs que por sua vez estão mapeados para sua respectiva saída (O). Este processo visa a otimizar a busca do valor aproximado, reduzindo a quantidade de acessos sequenciais à memória.

Figura 3.7: Resultado da clusterização hierárquica.



Fonte: O autor

Figura 3.8: Mapeamento do agrupamento hierárquico.



Fonte: O autor

A fim de obter melhores resultados com o uso de técnicas de clusterização de dados, as amostragens são normalizadas entre zero e um. O propósito da normalização é minimizar os problemas oriundos da dispersão entre variáveis, assim beneficiando algumas ferramentas de modelagem (e.g., redes neurais e clusterização). A Equação 3.1~~Equação 3.1~~ demonstra a fórmula utilizada na normalização de dados, onde x é o valor em questão e min e max são os valores mínimo e máximo do conjunto de dados respectivamente.

$$y = \frac{x - min}{max - min}$$

Equação 3.1: Fórmula utilizada para normalização de dados.

Para ambas execuções do algoritmo k-means, foi utilizada a distância Euclidiana (DEZA; DEZA, 2009) como métrica para definir a similaridade entre as amostragens, a qual é definida pela Equação *3.2*~~Equação *3.2*~~.

$$\text{Distancia_Euclidiana}(x, y) = \sqrt{\sum_{i=1}^{n} (y_i - x_i)^2}$$

Equação 3.2: Fórmula da Distância Euclidiana.

No contexto do EDCAC, a quantidade de K clusters é definida pelo programador. Este número deve ser o menor possível para alcançar a qualidade desejada, pois quanto maior o número de K, haverá dados mais próximos aos originais, que se traduz em uma qualidade melhor no resultado final da aplicação, porém da mesma maneira, haverá mais entradas na tabela, que resultará em um custo maior em área, energia e tempo de acesso.

3.1.2.3 Geração do binário

Após a etapa de geração de dados aproximados, o compilador gera um binário substituindo a execução original por instruções que vão inicializar e acessar a tabela de reúso ao invés de executar o código original. O binário executa no processador e invoca o acelerador de hardware ao invés de executar o código original quando necessário. A comunicação com o acelerador de hardware ocorre via mapeamento de memória controlada via software.

Três estruturas de dados são armazenadas ao longo da geração do binário: os valores dos *cluster of input clusters* (CIC), os *inputs clusters* (IC) e as saídas correspondentes – *outpus* (O).

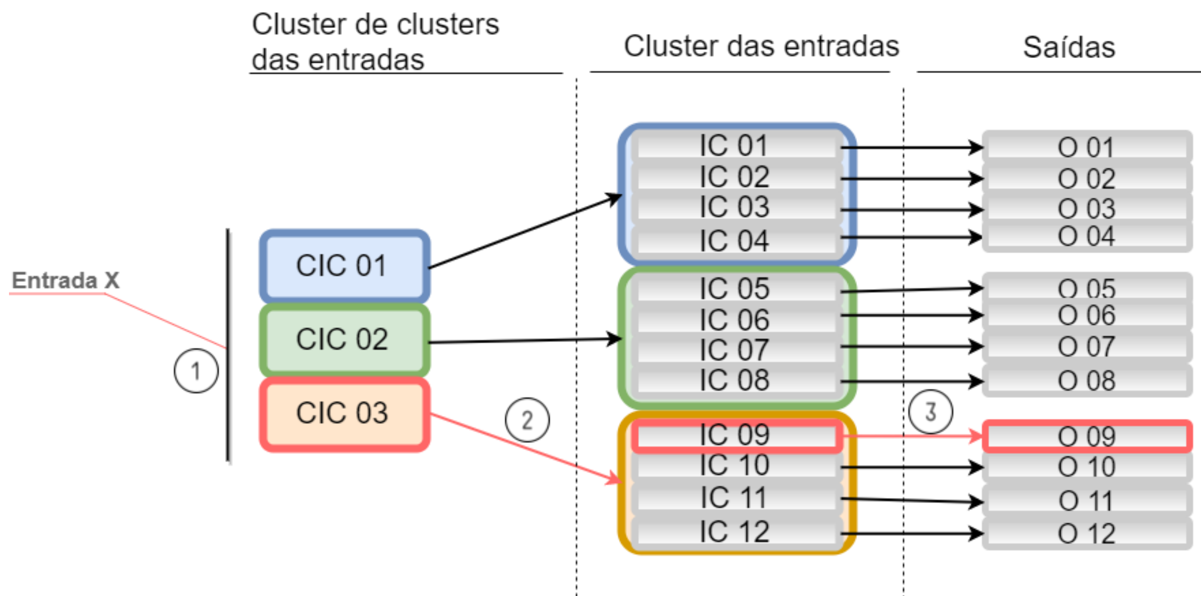
3.1.3 Execução

Quando a região do código é invocada com uma entrada x, as informações armazenadas nas estruturas de memórias discutidas acima são acessadas em três etapas como apresentado nos passos abaixo e na Figura *3.9*~~Figura *3.9*~~:

1. Todos os clusters na tabela de CIC são lidos e o cluster que possuir a menor distância para x é selecionado. Este CIC aponta para uma região da tabela de ICs, que contém um subconjunto de ICs;

2. Todos os valores da região de ICs que estão associados com o CIC encontrado na etapa anterior são lidos e o valor que possuir a menor distância para x é selecionado;
3. O valor O ligado com o IC selecionado na etapa 2 é selecionado e retornado como resultado da região aproximada.

Figura 3.9: Fluxo para obter o valor mais próximo em memória (lookups tables).



Fonte: O autor

Desta forma, é possível identificar dentre N valores armazenados nas tabelas de reúso qual é o mais próximo ao original de uma forma otimizada. Como discutido anteriormente, acessando a tabela em dois níveis, através de CICs e ICs, ao invés de utilizar somente ICs, é possível reduzir a um número médio de acessos de k para aproximadamente $k' + \frac{k}{k'} + 1$, onde $K' \approx \sqrt{k}$.

Para exemplificar, utilizaremos 256 clusters para gerar a tabela de ICs. Neste exemplo, temos 256 acessos sequenciais para definir qual é a entrada mais próxima e um acesso na tabela de O para de fato reutilizar o valor. Porém, utilizando a memória hierárquica, teremos 16 acessos na tabela de CIC e idealmente 16 acessos na tabela de ICs, mais um acesso na tabela de O, totalizando 33 acessos sequencias. Neste caso, temos uma redução de 88% de acessos à memória e, desta forma, resolvendo o problema de tabelas extremamente grandes e lentas.

Uma vez que a distância euclidiana é computacionalmente cara para calcular vetores grandes, a solução adotada foi trocar a métrica de distância utilizada na etapa de busca do valor mais próximo para a distância de Manhattan (definida como na Equação *3.3*~~Equação *3.3*~~), pois esta não envolve nenhuma multiplicação ou operação de raiz quadrada.

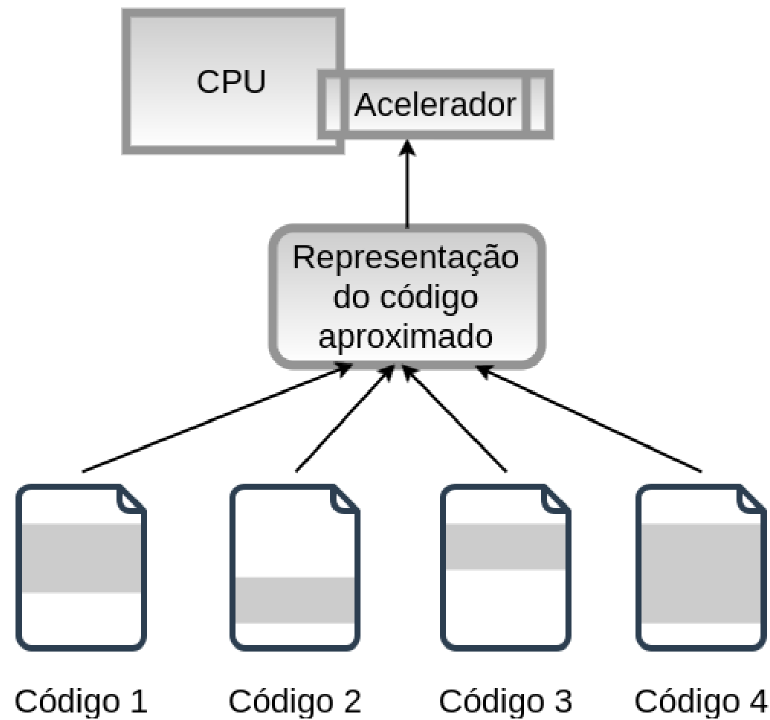
$$Distancia_de_Manhattan(x, y) = \sum_{i=1}^{n} |y_i - x_i|$$

Equação 3.3: Fórmula para calcular a distância de Manhattan.

Quando a aplicação inicia a execução, a estrutura de dados é carregada para as memórias via software, utilizando instruções especiais de manipulação de memórias. A partir do momento que as estruturadas de dados estejam carregadas em memória, o fluxo de execução é o mesmo, porém utilizando a busca em memória ao invés de utilizar o código original.

Na Figura *3.10*~~Figura *3.10*~~, é apresentada em forma gráfica e de alto nível a capacidade de converter regiões de códigos distintas em uma única representação comum. Desta forma, podemos acelerar as mais diversas aplicações, desde as mais simples que envolvem somente operações aritméticas com custo computacional baixo até as aplicações com fluxos mais complexos. A implementação desta abordagem em hardware possui um potencial promissor, pois o maior *overhead* de abordagens anteriores utilizando RNs é roteamento dos dados e a execução de operações complexas. Entretanto, utilizando a técnica apresentada neste capítulo, conseguimos abstrair este *overhead*. Desse modo, alcança-se resultados positivos em relação a performance e consumo energético, conforme será apresentado na seção 4.3.2.1.

Figura 3.10: Representação comum de trechos de códigos distintos.



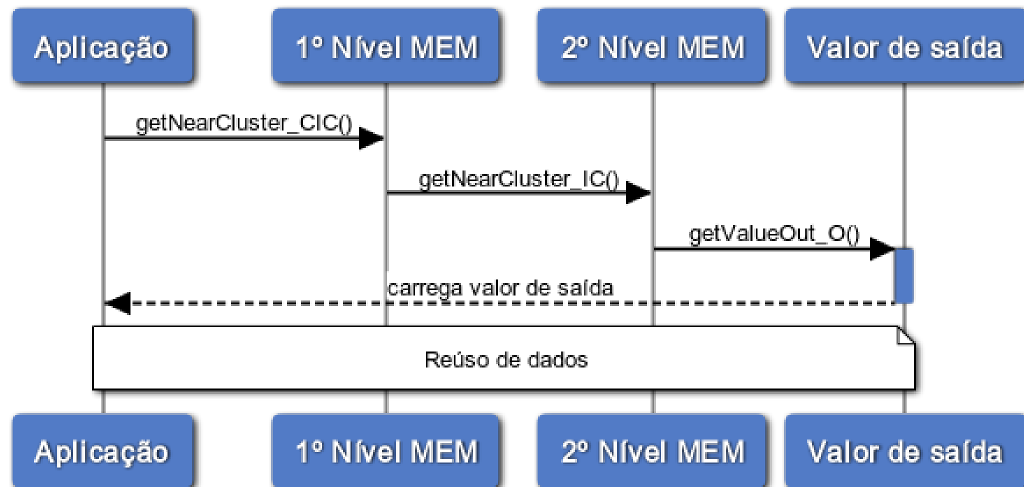
Fonte: O autor

3.2 Modelo de Implementação

Nesta seção, serão apresentados os modelos de implementação e exemplos de códigos utilizados para implementar EDCAC e RNs. O propósito de ambas implementações é provar que, em ambientes de simulação iguais, nossa técnica apresenta ganhos superiores a técnica de redes neurais, otimizando o uso de recursos computacionais disponíveis.

A implementação atual visou simular todo o fluxo da técnica, desde a pesquisa sequencial das informações nos níveis da memória, até o retorno do valor mais próximo, para ser propriamente reusado. A Figura 3.11~~Figura 3.11~~ apresenta um diagrama de sequência contendo o passo-a-passo da busca sequencial do valor aproximado, onde o primeiro nível representa a tabela de memória contendo CICs, o segundo nível ICs e o valor de saída representa Os.

Figura 3.11: Diagrama de sequência da técnica proposta.



Fonte: O autor

As Listagem 3.3~~Listagem 3.3~~ e Listagem 3.4~~Listagem 3.4~~ apresentam o código original utilizado na aproximação. A aplicação alvo deste exemplo é o benchmark JPEG, o qual contém duas funções (dct() e quantization()) que implementam a transformada discreta de cosseno e chamadas de funções e loops, respectivamente.

Listagem 3.3: Código original retirado do benchmark JPEG.

```
1.  dct(Y1);
2.  quantization(Y1, ILqt);
```

Listagem 3.4: Funções dct() e quantization() originais.

```
1.  void dct (INT16 *data)
2.  {
3.      UINT16 i;
4.      INT32 x0, x1, x2, x3, x4, x5, x6, x7, x8;
5.      static const UINT16 c1=1420;    /* cos PI/16 * root(2)  */
6.      static const UINT16 c2=1338;    /* cos PI/8 * root(2)   */
7.      static const UINT16 c3=1204;    /* cos 3PI/16 * root(2) */
8.      static const UINT16 c5=805;     /* cos 5PI/16 * root(2) */
9.      static const UINT16 c6=554;     /* cos 3PI/8 * root(2)  */
10.     static const UINT16 c7=283;     /* cos 7PI/16 * root(2) */
11.     static const UINT16 s1=3;
```

```
12.  static const UINT16 s2=10;
13.  static const UINT16 s3=13;
14.  for (i=8; i>0; i--)
15.  {
16.     x8 = data [0] + data [7];
17.     x0 = data [0] - data [7];
18.     x7 = data [1] + data [6];
19.     x1 = data [1] - data [6];
20.     x6 = data [2] + data [5];
21.     x2 = data [2] - data [5];
22.     x5 = data [3] + data [4];
23.     x3 = data [3] - data [4];
24.     x4 = x8 + x5;
25.     x8 -= x5;
26.     x5 = x7 + x6;
27.     x7 -= x6;
28.     data [0] = (INT16) (x4 + x5);
29.     data [4] = (INT16) (x4 - x5);
30.     data [2] = (INT16) ((x8*c2 + x7*c6) >> s2);
31.     data [6] = (INT16) ((x8*c6 - x7*c2) >> s2);
32.     data [7] = (INT16) ((x0*c7 - x1*c5 + x2*c3 - x3*c1) >> s2);
33.     data [5] = (INT16) ((x0*c5 - x1*c1 + x2*c7 + x3*c3) >> s2);
34.     data [3] = (INT16) ((x0*c3 - x1*c7 - x2*c1 - x3*c5) >> s2);
35.     data [1] = (INT16) ((x0*c1 + x1*c3 + x2*c5 + x3*c7) >> s2);
36.     data += 8;
37.  }
38.  data -= 64;
39.  for (i=8; i>0; i--)
40.  {
41.     x8 = data [0] + data [56];
42.     x0 = data [0] - data [56];
43.     x7 = data [8] + data [48];
44.     x1 = data [8] - data [48];
45.     x6 = data [16] + data [40];
```

```
46.         x2 = data [16] - data [40];
47.         x5 = data [24] + data [32];
48.         x3 = data [24] - data [32];
49.         x4 = x8 + x5;
50.         x8 -= x5;
51.         x5 = x7 + x6;
52.         x7 -= x6;
53.         data [0] = (INT16) ((x4 + x5) >> s1);
54.         data [32] = (INT16) ((x4 - x5) >> s1);
55.         data [16] = (INT16) ((x8*c2 + x7*c6) >> s3);
56.         data [48] = (INT16) ((x8*c6 - x7*c2) >> s3);
57.         data [56] = (INT16) ((x0*c7 - x1*c5 + x2*c3 - x3*c1) >> s3);
58.         data [40] = (INT16) ((x0*c5 - x1*c1 + x2*c7 + x3*c3) >> s3);
59.         data [24] = (INT16) ((x0*c3 - x1*c7 - x2*c1 - x3*c5) >> s3);
60.         data [8] = (INT16) ((x0*c1 + x1*c3 + x2*c5 + x3*c7) >> s3);
61.         data++;
62.     }
63. }
64.
65. void quantization(INT16* const data, UINT16* const quant_table_ptr) {
66.     INT16 i;
67.     INT32 value;
68.
69.     for (i = 63; i >= 0; i--) {
70.         value = data[i] * quant_table_ptr[i];
71.         value = (value + 0x4000) >> 15;
72.         Temp[zigzagTable[i]] = (INT16) value;
73.     }
74. }
```

Para exemplificar as modificações necessárias, a fim de permitir que uma aplicação possa ser corretamente aproximada dentro da técnica proposta, foi projetado e implementado o código abaixo, apresentado na Listagem *3.5*~~Listagem *3.5*~~.

Listagem 3.5: Exemplo da aplicação JPEG com aproximação, substituindo as funções dct() e quantization().

```
1.  for (int i = 0; i < BLOCK_SIZE; ++i) {
2.      aux[i] = normalization(dataIn[i], dt_min[i], dt_max[i]);
3.  }
4.
5.  int index_second_level = getClusterFirstLevel(aux);
6.
7.  switch ( index_second_level ) {
8.    case 0 :
9.      index_mem_out = getValueSecondLevel(aux, mem_second_level_0);
10.     for(int i = 0; i < BLOCK_SIZE; ++i)
11.     {
12.         Temp[i] = out_mem0[index_mem_out][i];
13.     }
14.     break;
15.   case 1 :
16.     index_mem_out = getValueSecondLevel(aux, mem_second_level_1);
17.     for(int i = 0; i < BLOCK_SIZE; ++i)
18.     {
19.         Temp[i] = out_mem1[index_mem_out][i];
20.     }
21.     break;
22.   case 2 :
23.     index_mem_out = getValueSecondLevel(aux, mem_second_level_2);
24.     for(int i = 0; i < BLOCK_SIZE; ++i)
25.     {
26.         Temp[i] = out_mem2[index_mem_out][i];
27.     }
28.     break;
29.   case 3 :
30.     index_mem_out = getValueSecondLevel(aux, mem_second_level_3);
31.     for(int i = 0; i < BLOCK_SIZE; ++i)
32.     {
```

```
33.        Temp[i] = out_mem3[index_mem_out][i];
34.     }
35.     break;
36. }
```

A aplicação JPEG, em específico, contém a mesma quantidade de entradas e saídas, ambas contendo um vetor de 64 posições.

Na linha 2 da Listagem 3.5Listagem 3.5 é feita a normalização dos dados de entrada entre zero e um, através da função *normalization()*, como já discutido previamente.

A linha 5 chama a função *getClusterFirstLevel*(), passando como parâmetro o valor original. Esta função fará a busca sequencial do cluster mais próximo ao valor original no primeiro nível de memória (CIC) e retornará a posição deste cluster. Esta posição indicará em qual fração no segundo nível (IC) deverá ser buscado o valor aproximado, apresentado a partir da linha 7, onde é feito um *switch case* para definir onde devemos pesquisar o valor mais próximo. Nesta etapa, é chamada a função *getValueSecondLevel*(), passando o vetor de dados originais como parâmetro de entrada. Esta função irá retornar o índice que contém o valor mais próximo ao original e atrelado a cada índice do segundo nível de memória, onde temos a memória de saída (com O), que contém o valor aproximado a ser reutilizado pela aplicação. Como exibido na linha 12, é utilizado a variável *index*_mem_out que contém o índice correto para buscar o valor na memória de saída, representada pelo vetor out_*mem**. Ambas as funções *getValueSecondLevel*() e *getClusterFirstLevel*() utilizam a distância de Manhattan como medida para identificar qual é o valor mais próximo ao original. Com as modificações descritas acima, a aplicação está apta a ser executada dentro de uma classe de aplicações aproximativas para o modelo proposto.

Para a implementação da rede neural, foi utilizado parte do trabalho disponibilizado por (ESMAEILZADEH et al., 2012a). O código utiliza chamadas da rede neural para substituir a execução do código original. A Listagem 3.6Listagem 3.6 apresenta o código utilizado em (ESMAEILZADEH et al., 2012a). Neste exemplo, é implementado o algoritmo Inversek2j, que tem como entrada e saída um vetor de duas posições. Na linha 6 é feita a leitura do arquivo de treinamento da rede neural e na linha 7 a execução da rede propriamente dita.

Listagem 3.6: Código utilizado pela transformação em redes neurais.

```
1.  ann_type *parrotOut;
2.  fann_type parrotIn[2];
3.  parrotIn[0] = dataIn[0];
4.  parrotIn[1] = dataIn[1];
5.
6.  static struct fann *ann = fann_create_from_file("inversek2j.nn");
7.  parrotOut = fann_run(ann, (fann_type*)parrotIn);
8.
9.  dataOut[0] = parrotOut[0];
10. dataOut[1] = parrotOut[1];
```

3.3 Considerações

Este capítulo apresentou o framework EDCAC, com a finalidade de oferecer uma alternativa ao estado da arte atual (redes neurais). Será demonstrado que nossa abordagem pode substituir uma região de código com sucesso, sendo capaz de converter diferentes regiões de código, desde fluxos simples a complexos, adicionando um grau de imprecisão da mesma maneira como já vem sendo feito no estado atual da arte.

4 AVALIAÇÃO EM SOFTWARE E HARDWARE DO FRAMEWORK EDCAC

Este capítulo apresenta duas avaliações (assim como a metodologia e benchmarks usados) da técnica. A primeira, em software, visa comparar o desempenho do EDCAC e RN quando implementadas e executadas em um processador de propósito geral. A segunda é feita em hardware, comparando as duas técnicas e considerando que há um hardware dedicado para cada uma delas. Um ponto importante é a capacidade de adaptação que a técnica proporciona, tanto em termos de software como em hardware, viabilizando uma exploração de espaço de projeto potencialmente grande.

4.1 Benchmarks

Na Tabela 4.1~~Tabela 4.1~~, são listados os benchmarks utilizados para avaliação da técnica proposta. Para isso, foi utilizado um conjunto de seis aplicações da suíte de benchmarks Axbench (YAZDANBAKHSH et al., 2017). Todos os benchmarks foram escritos em C e C++, e foram selecionados pela utilidade em aplicações gerais e tolerância na imprecisão dos resultados. Utilizamos as seguintes aplicações: FFT, Inversek2j, Jmeint, JPEG, K-means e Sobel. Juntos, estes *benchmarks* representam uma ampla gama de domínios de aplicações, incluindo processamento de sinais digitais, robótica, jogos, compressão de imagens, *machine learning* e processamento de imagens. Estas classes de aplicações também já foram utilizadas em trabalhos anteriores (ALVAREZ et al., 2005; ESMAEILZADEH et al., 2012a; YAZDANBAKHSH et al., 2017). Na mesma tabela, também é listado o conjunto de dados de entrada utilizados para avaliar performance, energia e precisão do resultado final (conjuntos estes diferentes dos utilizados na fase geração de dados aproximados). Para aplicações que utilizam imagens como entrada, foram utilizadas diferentes imagens para avaliação. Já para aplicações com conjunto de dados randômicos, foram utilizados conjuntos de dados randômicos diferentes.

Tabela 4.1: Benchmarks utilizadas na avaliação, com características de cada função aproximada, dados de treinamento e resultados da aproximação.

Aplicação	Tipo	Dataset de validação	# Hotspot	Dataset de treinamento	Métrica de erro	Tamanhos de entradas e saídas	Hierarquia da memória	# clusters	% Erro EDCAC	% Erro RN	Topologia da RN
fft	Processamento de sinais	32768 números randômicos em ponto flutuante	11264	2048 números randômicos em ponto flutuante	Erro médio relativo	1 – 2	16–8-1	128	6.88%	6.24%	1-4-4-2
Inversek2j	Robótica	10000(x,y) coordenadas randômicas	100000	10000(x,y) Coordenadas randômicas	Erro médio relativo	2 – 2	16–16-1	256	9.39%	9.98%	2-8-2
Jmeint	Jogos 3D	10000 pares randômicos de coordenadas de um triangulo 3d	10000	100000 coordenadas randômicas de pares de um triangulo 3D	Taxa de erro	18 – 2	4–8-1	32	34.24%	30.16%	18-32-8-2
Jpeg	Compressão	Imagem 220x200-Pixel	4096	Dez Imagens 512x512	Image Diff	64 – 64	4–4-1	16	8.79%	9.00%	64-16-64
Kmeans	Machine Learnig	Imagem 220x200-Pixel	1572864	Uma Imagens 512x513	Image Diff	6 – 1	16–32–1	512	11.31%	6.18%	6-8-4-1
Sobel	Processamento de imagens	Imagem 220x200-Pixel	262144	Uma Imagens 512x514	Image Diff	9 - 1	16–32-1	512	14.55%	15.00%	9-8-1

Fonte: O autor

O código de cada benchmark foi identificado como descrito na seção 3.2, no qual as funções identificadas respeitam as restrições discutidas neste capítulo. Existem muitas formas para identificar o código a ser aproximado, dentre elas destaca-se que o código deve ser utilizando frequentemente para tirar mais proveito da técnica. Para propósitos de avaliação, utiliza-se as mesmas anotações utilizadas em trabalhos anteriores, (ESMAEILZADEH et al., 2012a; YAZDANBAKHSH et al., 2017).

Na maioria dos benchmarks, o código aproximado contém fluxos de controle complexos, incluindo condicionais, *loops* e chamadas de funções. No Jmeint, por exemplo, existem muitas chamadas de funções e inúmeras instruções condicionais; o JPEG contém a transformada discreta de cosseno e fases de quantização, que contém chamadas de funções e *loops*; já no FFT, Inversek2j e Sobel, o código alvo consiste principalmente de operações aritméticas e fluxos de controle simples; e no benchmark k-means, o código aproximado é executado com alta frequência. Neste, como a aproximação é feita no cálculo da distância Euclidiana, o fluxo é simples e com apenas operações aritméticas. Nossa técnica é capaz de abstrair toda a complexidade, independente da função a ser aproximada.

Para gerar os dados aproximados, utilizamos entradas típicas para cada benchmark (e.g., imagens ou números randômicos) baseando-se no trabalho (ESMAEILZADEH et al., 2012a), tanto para treinamento como para avaliação. Para benchmarks que utilizam entradas randômicas, é definido um intervalo aceitável pela aplicação, e assim são gerados valores randômicos neste intervalo. Para os benchmarks que utilizam imagens como entrada, é definido um conjunto de imagens dentro da quantidade necessária para a geração dos dados para treinamento e outro para a avaliação da técnica.

Tabela *4.1*~~Tabela *4.1*~~ mostra a quantidade e o tipo de entrada de cada benchmark, utilizando diferentes imagens e conjunto de dados randômicos, definidos pelas colunas "*Dataset de validação*" e "*Dataset de treinamento*".

A coluna "*Hierarquia da memória*", na Tabela *4.1*~~Tabela *4.1*~~, mostra a hierarquia utilizada por cada aplicação, construída como discutido na seção 3.1~~3.1~~. O benchmark Inversek2j, por exemplo, precisa de 33 acessos à memória para buscar o valor mais próximo ao original. Este benchmark utiliza 256 clusters na primeira fase de geração de dados (IC) e 16 clusters para gerar o primeiro nível da hierarquia de memória (CIC). Idealmente, cada cluster do primeiro nível irá conter 16 valores. Sendo assim, teremos 16 acessos no primeiro nível, 16 acessos no segundo nível para definir o valor mais próximo contido na fração de

clusters identificado no primeiro nível e mais um acesso na memória que contém o valor de saída (O), contabilizando 33 acessos a memória para busca do valor mais próximo.

Para a avaliação do erro, são utilizadas métricas específicas para cada benchmark, como mostrado na coluna "*Métrica de Erro*". A avaliação do resultado final é feita através da comparação dos resultados obtidos com a aplicação sem utilizar a técnica de aproximação (i.e., resultados originais) e dos resultados obtidos utilizando nossa técnica. No FFT e Inversek2j, que geram valores numéricos em suas saídas, é medido o erro médio de cada valor. Já para o Jmeint, que calcula se dois triângulos tridimensionais se cruzam, o erro é medido pela taxa de acerto (triângulos se cruzam ou não). Para as aplicações que produzem imagens em suas saídas, como JPEG, Sobel e K-means, é utilizada a diferença entre a imagem original e a aproximada. Na coluna "% *Erro EDCAC*" é mostrado a porcentagem de erros de cada aplicação, considerando a métrica de avaliação.

4.2 Implementação em Software

4.2.1 Metodologia

No intuito de avaliar o efeito de performance da nossa técnica e da utilização de RNs também em software, utilizamos o simulador Gem5 (BINKERT et al., 2011), que é capaz de simular diversas arquiteturas, bem como sistemas completos. O simulador foi configurado para simular a recente microarquitetura Intel Haswell, como mostrado na Tabela *4.2*~~Tabela *4.2*~~.

Os benchmarks foram compilados utilizando o compilador g++ versão 4.9.2, com a flag de otimização -O3, a fim de habilitar otimizações agressivas pelo compilador. O código baseline reportado nos experimentos é a execução do código original sem nenhuma alteração, da mesma forma que feito com as aplicações que incluem as técnicas aproximadas.

Tabela 4.2: Configuração do simulador Gem5.

Core		Memória Cache
Architecture	X86-64	L1 Inst 32KB, 32B per line, 8-way, 1 cycle latency
Issue Width	8	L1 Data 32KB, 32B per line, 8-way, 2 cycle latency
Issue Ports ALU	6	L2 Cache 256KB, 32B per line, 8-way, 6 cycle latency
Issue Ports MUL	2	L3 Cache 2Mb, 32B per line, 16-way, 10 cycle latency
Issue Ports LOAD	2	
ROB Entries	192	
Instruction Qeue Entries	60	

Fonte: O autor

Para a implementação que utiliza RNs, o código original foi substituído pela execução de uma rede neural implementada utilizando a biblioteca FANN, sendo reaproveitado parte do trabalho de (ESMAEILZADEH et al., 2012a) e (YAZDANBAKHSH et al., 2017), no qual é disponibilizada a implementação da técnica de RNs em software. Para cada benchmark, foi utilizada uma configuração diferente da rede neural seguindo as informações disponibilizadas por (ESMAEILZADEH et al., 2012a). Logo, cada benchmark teve sua própria rede neural treinada separadamente. O EDCAC também possui uma implementação customizada para cada benchmark, semelhante ao código apresentado na seção 3.2.

Todas as informações relacionadas a cada técnica utilizada foram obtidas através de estatísticas fornecidas pelo simulador gem5, como: 1) Acesso a memória, que é a operação de ler e escrever informações na memória principal e memórias cache; 2) Micro-instruções, também conhecidas como micro-ops (quantidade de micro-operações de uma instrução complexa na arquitetura x86); e 3) Performance, que é a quantidade total de ciclos que uma determinada tarefa leva para ser realizada pelo sistema computacional.

Durante a simulação, o benchmark sobel apresentou erros na execução devido à necessidade de bibliotecas específicas que o simulador não suportava. Para manter a consistência dos experimentos e utilizar apenas aplicações da suíte Axbench, decidimos omitir resultados do sobel ao invés de modificar a aplicação.

Um recurso importante a ser mensurado é o número de acessos à memória, mesmo que a comparação seja feita com o somatório de todos os acessos a memórias, visto que geralmente uma quantidade menor de acessos resulta em menor custo em energia e melhor

desempenho. A Figura *4.1*~~Figura *4.1*~~ exibe dados referentes a estatísticas obtidas através do simulador Gem5 em relação a acessos a memória. Os resultados apontam um aumento potencial em performance computacional utilizando EDCAC, uma vez que a redução de acessos é agressiva em relação a RNs. Todos os benchmarks diminuem significativamente o número de acessos à memória em relação a RN. Por exemplo, o Jmeint mostra em torno de 100 vezes menos acessos à memória.

Figura 4.1: Acessos a memória cache L1, L2, L3 e memória principal, normalizado em relação ao baseline.

Fonte: O autor

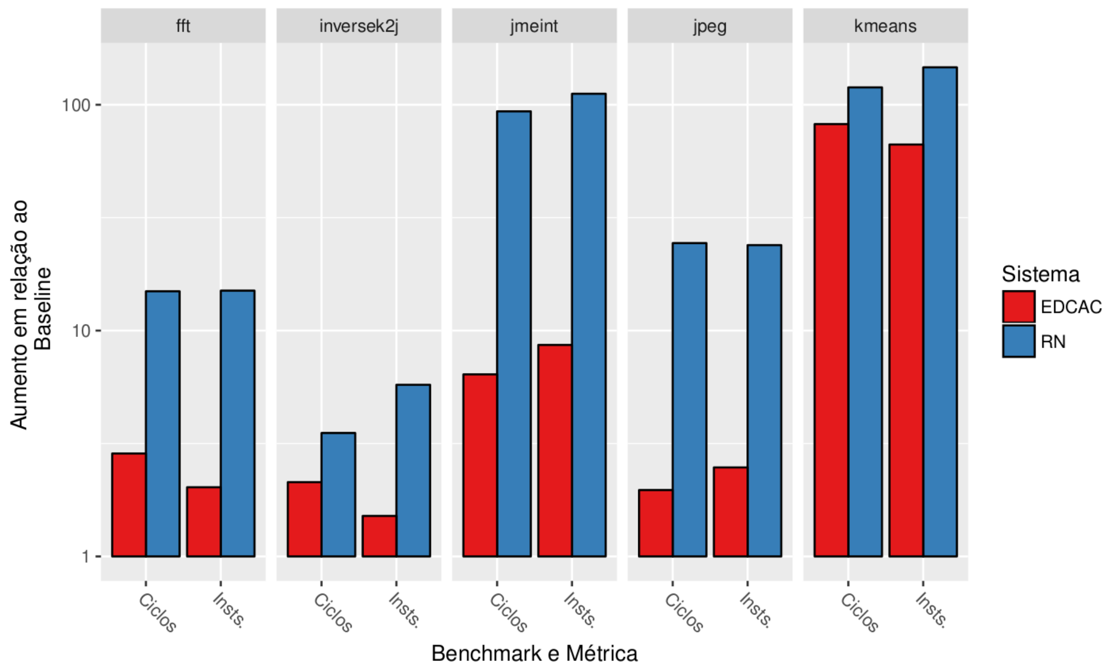
A Figura *4.2*~~Figura *4.2*~~ apresenta o número de instruções executadas no processador e o número de ciclos, ambos normalizados com relação ao algoritmo original (baseline, sem aproximação). Uma RN é comumente constituída por multiplicadores, somadores e uma função de ativação para cada neurônio (a sigmoide). Logo, as operações envolvem calcular o valor para cada neurônio, definir os pesos das arestas e manipular os valores obtidos de cada neurônio, resultando em operações complexas.

Os resultados mostram que a abordagem de EDCAC possui uma performance melhor que a implementação com RN em todas aplicações. A principal razão é que, utilizando EDCAC, executa-se 79% menos instruções e as operações não são complexas como em RNs,

o que mostra um grande potencial de utilizar EDCAC devido a sua redução agressiva em recursos computacionais, que por sua vez permite ganhos em performance e, principalmente, em diminuição no consumo energético.

No entanto, ambas soluções implementadas em software são inferiores ao *baseline*, quando consideramos a performance. Este comportamento é a razão pela qual todas as soluções propostas em RNs possuem um hardware dedicado como acelerador, e também a motivação para criar um hardware dedicado para EDCAC, como será discutido nas próximas seções. Assim, esta análise mostra que, apesar de ambos serem mais caros que a aplicação original, nossa abordagem é superior em termos de performance quando comparado com RNs. Logo, a implementação de um hardware dedicado para EDCAC tende a ser superior a uma rede neural dedicada quando implementada em HW em condições parecidas.

Figura 4.2: Aumento da quantidade de instruções executadas e ciclos relativo ao algoritmo baseline, quando executado as abordagens RN e EDCAC em software.



Fonte: O autor

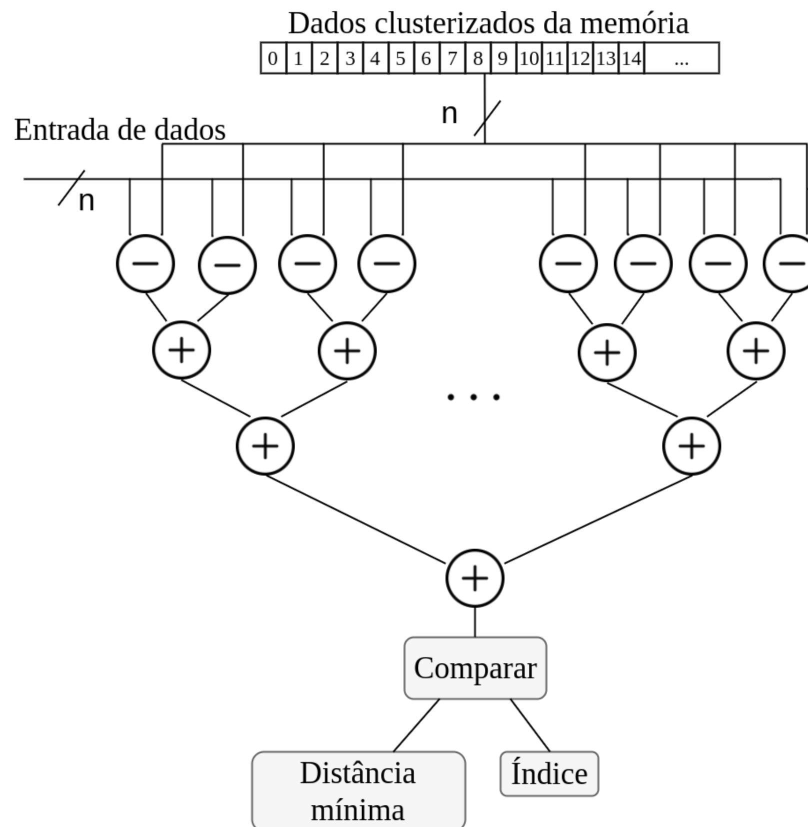
4.3 Implementação em Hardware

4.3.1 Arquitetura do coprocessador

Nesta seção, é apresentada a arquitetura proposta, capaz acelerar a execução dos mais diversos comportamentos na aplicação original, e também de reproduzir diferentes trechos de códigos, aceitando quantidades diferentes de entradas e saídas de dados. A solução é composta por três memórias, uma para os CICs, uma para os ICs e outra para Os, todas com a largura da linha de memórias compatível com as entradas e saídas de cada aplicação. O hardware para calcular a distância Manhattan entre dois vetores do tipo *float* é baseado em um *pipeline*, e é totalmente paralelo, utilizando uma árvore de somadores, conforme mostrado na Figura 4.3~~Figura 4.3~~.

Das aplicações executadas neste trabalho, foi identificado que o número máximo de Unidades de Processamento (UPs) necessárias para calcular a distância é de 127, definidas pelo benchmark JPEG, que contém a maior quantidade de valores no vetor de entrada (64 valores). Após experimentos realizados, foi identificado que a quantidade de UPs no primeiro nível da árvore deve ser idêntica à quantidade de parâmetros de entrada de cada função aproximada, para que o hardware planejado seja o mais rápido possível e o cálculo da distância de Manhattan seja feito totalmente em paralelo.

Figura 4.3: Arquitetura do coprocessador.



Fonte: O autor

O esquema apresentado na figura Figura *4.3*~~Figura *4.3*~~ é um hardware projetado com o intuito de proporcionar o maior desempenho possível em termos de performance. Tal arquitetura permite que a distância de Manhattan seja calculada em um ciclo após o preenchimento do pipeline de somadores.

Da mesma maneira que o autor apresenta a técnica utilizando redes neurais (ESMAEILZADEH et al., 2012a), na qual é possível ter variações no hardware - tais como aumentar a quantidade de neurônios físicos -, apresenta-se duas análises com relação a diminuição da quantidade de UPs do hardware, bem como a largura das estruturas de cada memória

4.3.1.1 Quantidade de UPs

Como serão discutidos nos resultados de performance na seção 4.3.2.1, todos os benchmarks apresentam um aumento de performance relevante. Com isto, existe uma margem com a qual podemos diminuir a quantidade de UPs, a fim de poupar área. Porém, ao diminuir a quantidade de UPs, a consequência é que a quantidade de ciclos para calcular a distância de Manhattan aumentará em relação à nova quantidade de UPs.

4.3.1.2 Largura das memórias

Com relação à estrutura de memórias, o hardware é projetado para que a largura seja a quantidade de entradas de cada *hotspot.* Dito isto, existe a possibilidade de diminuir a largura da memória, resultando no aumento da quantidade de leituras necessárias para recuperar o valor aproximado e, por consequência, o aumento da quantidade de ciclos para execução da técnica. Contudo, da mesma forma discutida anteriormente, visto que as aplicações mostram um aumento de performance relevante, esta é uma outra possível alteração no intuito de diminuir área (e.g., grande parte da área consumida por uma memória é devido ao roteamento entre as células, quanto maior for a largura maior será o roteamento) em troca de desempenho.

Muitas são as possíveis alterações para explorar maior desempenho, ganhos em energia ou mesmo em área. Os resultados obtidos nesta dissertação foram baseados em um hardware como descrito nesta seção, utilizando o cálculo da distância totalmente paralelo e com a largura de memórias necessárias para cada aplicação. A exploração do espaço de projeto investigando diferentes trade-off será analisada em trabalhos futuros.

4.3.2 Metodologia

Para avaliação do efeito da performance e consumo energético, foi desenvolvido um modelo de performance, área e energia, para ambas abordagens EDCAC e RN. Este modelo foi baseado em medidas fornecidas pelas ferramentas Cadence RTL Compiler (para unidades de ponto flutuante, utilizando *verilog*) e CACTI (para modelagem das memórias). Foi considerado um processador base executando a uma frequência de 2.5 GHz, do qual foram

derivadas as latências em ciclos para todas as estruturas. Toda a análise considera uma tecnologia de 65nm.

Para modelar a implementação da técnica de RNs, utilizamos a descrição do trabalho (ESMAEILZADEH et al., 2012a), com oito neurônios físicos, chamados de PEs (do inglês, *Processing Elements*). A configuração da rede neural é enviada em tempo de execução para o hardware dedicado, utilizando virtualização para redes que necessitam uma quantidade maior de oito neurônios. Os pesos e entradas de cada neurônio são armazenados em memórias, e multiplexados em tempo de execução de cada neurônio. O barramento de conexão entre os PEs não é especificado no trabalho original. Por isso, foi considerado que todos estão conectados através de um barramento de 32-bit, com uma área insignificante e que a transação dos dados é executada em 2 ciclos do processador.

O EDCAC foi modelado considerando os requisitos de cada benchmark individualmente e compondo três memórias – uma para armazenar os CICs, uma para ICs e outra para Os –, ambas com a largura de linhas compatíveis com o tamanho de cada entrada e saída da aplicação. O hardware para computar a distância de Manhattan é composto por um pipeline em paralelo, utilizando uma árvore de somadores, como mostrado na Figura 4.3~~Figura 4.3~~. Considerando todos os cenários possíveis, com quantidade de clusters distintas para cada benchmark, foi selecionado para cada aplicação aquela quantidade que é capaz de prover um nível de qualidade mais próximo ao trabalho (ESMAEILZADEH et al., 2012a).

Desta forma, os parâmetros primários para modelagem de performance são os seguintes:

- N_{FPU}: É a quantidade de unidades de ponto flutuante necessárias para realizar o cálculo da distância de Manhattan.
- N_{Access}: É a quantidade necessária de acessos a memória para definir o valor mais próximo ao original, sendo relativo a quantidade de k clusters para cada benchmark.

O tempo gasto para buscar o valor aproximado na tabela de reúso é:

$$T_{Total} = (N_{Access} * T_{Men_CIC} + T_{dist}) + (N_{Access} * T_{Men_IC} + T_{dist}) + T_{Men_O}$$

Onde:

- T_{Men_CIC}: É o tempo de acesso à memória que contém os pares de entradas aproximados clusters dos clusters.
- T_{Men_IC}: É o tempo de acesso à memória que contém os pares de entradas aproximados das entradas dos clusters.
- T_{Men_O}: É o tempo de acesso à memória que contém os pares de saída aproximados.
- T_{dist}: É o tempo para calcular a distância de Manhattan entre dois vetores, relativo ao tempo para as unidades de pontos flutuantes (N_{FPU}). Aqui é considerado o tempo de cada unidade funcional mais o tempo de carregar o pipeline de somadores.

Para avaliação de energia, foi modelado o consumo energético para a quantidade mínima necessária de cada benchmark (i.e., cada benchmark com apenas a quantidade de unidade de pontos flutuantes e memória mínima necessária). Para o cálculo de energia, foram considerados os seguintes fatores: 1) acessos a memórias; 2) tempo que a lógica combinacional fica ligada; e 3) consumo estático de cada componente. São parâmetros primários do modelo:

- N_{Access}: É a quantidade necessária de acessos a memória para definir o valor mais próximo ao original. É relativo a quantidade de clusters para cada benchmark.

A energia gasta para efetuar a busca do valor aproximado é:

$$E_{Total} = E_{LeakagePower} + (N_{Access} * E_{Read_memory}) + (T_{Logic} * E_{Logic})$$

Onde:

- $E_{LeakagePower}$: É a energia consumida estaticamente quando o processador está ligado, mas não executando nenhuma tarefa (corrente de fuga).
- E_{Read_memory}: É considerada a energia gasta para cada acesso a memória.
- T_{Logic}: É considerado o tempo em que as unidades de processamento estão executando uma tarefa.
- E_{Logic}: É a energia gasta quando o hardware está executando uma determinada tarefa.

A mesma abordagem utilizada na modelagem de energia foi utilizada para modelar a área, no intuito de prover dados precisos, utilizando CACTI e valores providos pela síntese

lógica como base para o desenvolvimento deste modelo. São parâmetros primários do modelo:

A área gasta para efetuar a busca do valor aproximado:

$$A_{Total} = A_{Memories} + (N_{Logic} * A_{Logic})$$

Onde:

- $A_{Memories}$: É considerado a área de todas as memórias necessárias para a implementação da técnica, a área é relativa a quantidade de clusters.
- N_{Logic}: É a quantidade de unidades funcionais utilizadas para cada aplicação.
- A_{Logic}: É a área utilizada por cada unidade funcional.

Para EDCAC, a mesma configuração de memória foi utilizada para implementação em hardware e software, como apresentado na Tabela 4.3~~Tabela 4.3~~.

Tabela 4.3: Configuração de memória utilizada por cada aplicação.

Aplicação	Tamanho Entrada (B)	Tamanho Saída (B)	1º Memória			2º Memória			Memória de saída		
			Linhas	Latência (ns)	Área (mm²)	Linhas	Latência (ns)	Área (mm²)	Linhas	Latência (ns)	Área (mm²)
FFT	4	8	16	0.31	0.001	128	0.40	0.005	128	0.41	0.009
Inversek2j	8	8	16	0.33	0.003	256	0.49	0.017	256	0.49	0.017
Jmeint	72	4	4	0.38	0.136	32	0.38	0.143	32	0.37	0.002
JPEG	256	256	4	0.62	1.549	16	0.62	1.548	16	0.62	1.549
Kmeans	24	4	16	0.34	0.018	512	0.62	0.092	512	0.50	0.016
Sobel	36	4	16	0.35	0.038	512	0.63	0.145	512	0.50	0.016

Fonte: O Autor

4.3.2.1 Resultados

Considerando todos os cenários possíveis para EDCAC, com quantidade de clusters distintas, foi selecionado para cada benchmark a quantidade que alcance um nível de qualidade similar ao de redes neurais. A Tabela *4.4*~~Tabela *4.4*~~, apresenta resultados absolutos da execução, os quais serão discutidos em mais detalhes a seguir.

Tabela 4.4: Dados absolutos para os cenários citados nesta seção.

Aplicação	EDCAC clusters	RN PEs	Performance		Área		Energia		Erro	
			EDCAC	RN	EDCAC	RN	EDCAC	RN	EDCAC	RN
FFT	128	8	32	76	0.03	0.87	$1.54 * 10^{-10}$	$9.49 * 10^{-9}$	6.9	6.2
Inversek2j	256	8	62	84	0.06	0.87	$6.38 * 10^{-10}$	$10.7 * 10^{-9}$	9.4	9.9
K-means	512	8	106	140	0.20	0.87	$40.7 * 10^{-10}$	$18.1 * 10^{-9}$	11.3	6.2
Sobel	512	8	112	124	0.30	0.87	$63.1 * 10^{-10}$	$16.1 * 10^{-9}$	14.6	15.0
Jmeint	32	8	45	520	0.49	0.87	$30.8 * 10^{-10}$	$72.9 * 10^{-9}$	34.2	30.2
JPEG	16	8	54	1060	5.40	0.87	$128 * 10^{-10}$	$152 * 10^{-9}$	8.8	9.0

Fonte: O autor

A Figura 4.4~~Figura 4.4~~ apresenta o número de ciclos gastos executando o hotspot de cada benchmark, utilizando ambas técnicas de RNs e EDCAC. Os resultados estão normalizados em respeito à execução do hotspot em software com o algoritmo original. As aplicações estão ordenadas pela complexidade na técnica de RNs (complexidade medida através da quantidade de operações MAC). Para melhor entendimento do custo envolvido na técnica de RNs, foi feita a decomposição do número total de ciclos em custo computacional de cada neurônio, o custo de roteamento dos resultados e o roteamento dos pesos de cada neurônio.

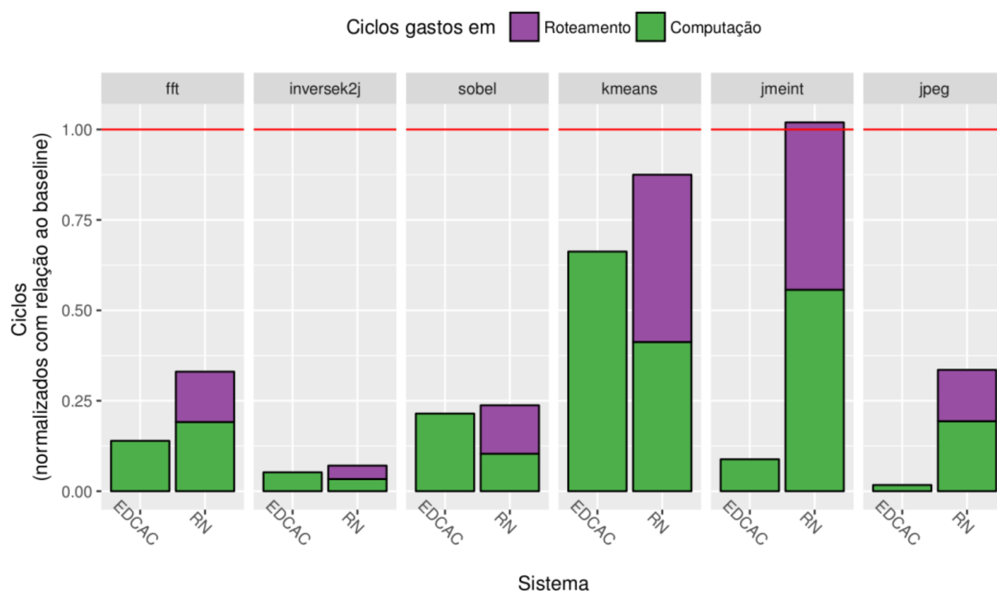
Como mostrado na Figura 4.4~~Figura 4.4~~, EDCAC apresenta performance superior em todos os benchmarks, quando comparado com RNs. Desta forma, confirma-se nossa hipótese inicial, de que é possível substituir o custo de operações complexas e roteamento da RN por simples acessos a memória, proporcionando aceleração na execução. Podemos observar que nos benchmarks de menor complexidade na rede neural – como Inversek2j, Sobel e K-means – a quantidade de ciclos que o EDCAC consome para computar o hotspot é maior que a quantidade de ciclos utilizada em RNs. No entanto, RNs apresentam uma grande quantidade de ciclos gastos em roteamento, o qual é inexistente em nossa abordagem, já que todos os valores lidos da memória são diretamente atribuídos ao pipeline do hardware dedicado.

Para as aplicações mais complexas, como Jmeint e JPEG, a técnica proposta, além de evitar os altos custos de roteamento das RNs, ainda escala com maior eficiência quando o tamanho das entradas de cada aplicação aumenta. Enquanto em RNs o número de operações para computação e roteamento aumentam linearmente, nosso custo em computação escala

com o logaritmo da largura da entrada de dados, pois o cálculo da distância entre os clusters é feito em paralelo e utilizando uma árvore de somadores. É possível argumentar que aumentando a quantidade de PEs na técnica de RNs, tal custo computacional poderia diminuir, no entanto esta estratégia não traria melhorias relacionadas ao custo em roteamento, que por si só já é superior ao total de ciclos consumidos por EDCAC.

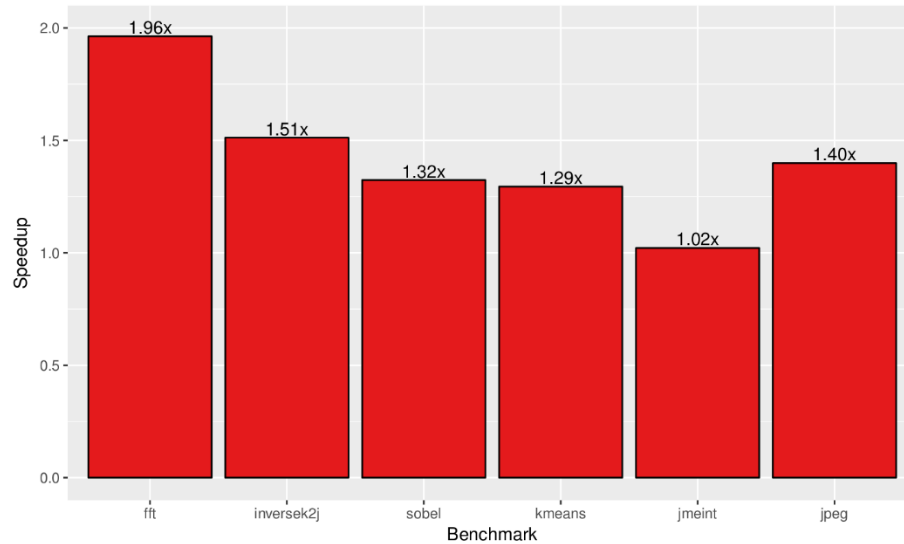
Adicionalmente, a Figura 4.5~~Figura 4.5~~ apresenta o *speedup* total de toda aplicação (com respeito à execução do baseline), no qual EDCAC atinge entre 1.02x e 1.96x de melhorias. Este *speedup* da aplicação depende do *speedup* de cada hotspot e da frequência que ele é executado. Por exemplo, embora o benchmark FFT não apresente o melhor speedup de hotspot (quando comparando com os outros benchmarks), este hotspot representa em torno de 56% do tempo total de execução da aplicação. Este alto reúso do hotspot faz com que o FFT tenha o melhor *speedup* de aplicação entre os benchmarks avaliados.

Figura 4.4: Quantidade de ciclos para as abordagens EDCAC e RNs em hardware, normalizados com respeito a execução do baseline (algoritmo original executado em software).



Fonte: O autor

Figura 4.5: Speedup total de cada aplicação utilizando EDCAC.

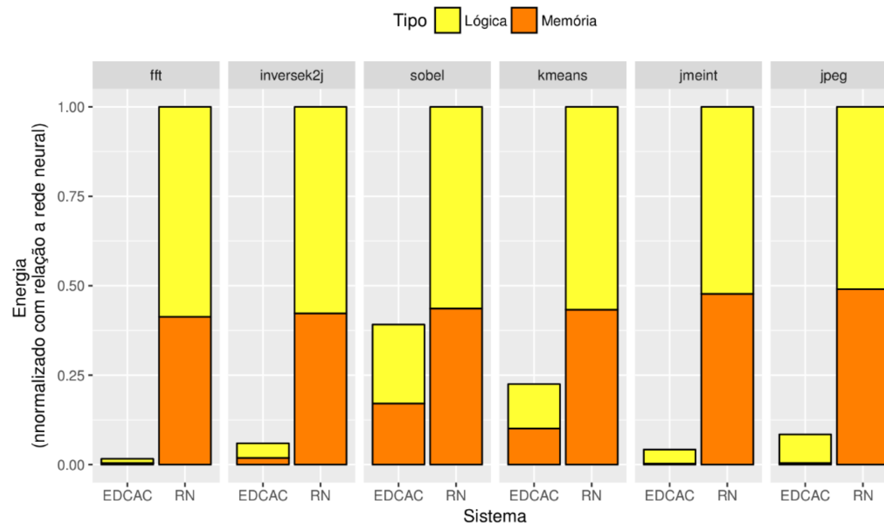


Fonte: O autor

A Figura 4.6~~Figura 4.6~~ apresenta o consumo energético para ambas as abordagens EDCAC e RN, agora normalizadas com respeito a implementação da RN. Como é possível visualizar, há redução de consumo energético em todos os benchmarks, com uma média de 85% de redução em todas as aplicações quando comparado com RN. A maior redução acontece no benchmark FFT (98,4%), pois o cálculo da distância euclidiana é muito simples. O FFT possui apenas uma entrada, que por sua vez, necessita de uma memória com uma largura menor e somente uma subtração para o cálculo da distância.

Para algumas aplicações, o número de unidades lógicas EDCAC é menor que em RN e com isto, a potência de EDCAC é menor. Contudo, mesmo nos casos em que o EDCAC necessita de mais unidades lógicas devido ao tamanho das entradas de cada aplicação (e.g. JPEG), o *speedup* provido pela técnica é capaz de neutralizar o aumento da potência do hardware, no qual resulta em baixo consumo de energia.

Figura 4.6: Consumo de energia para ambas abordagens, EDCAC e RN, dividindo entre consumo da memória e lógica, normalizado com respeito ao total de consumo da RN.

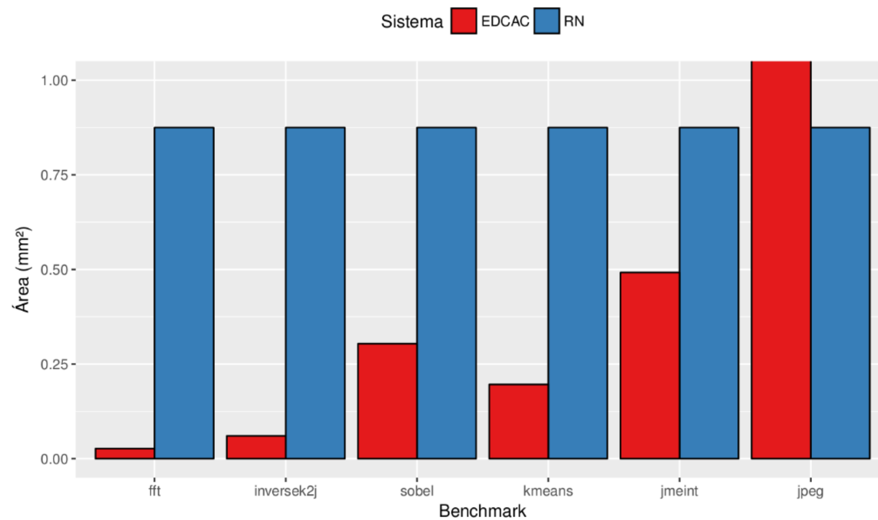


Fonte: O autor

A Figura 4.7~~Figura 4.7~~ apresenta a área total consumida por EDCAC e RN. Para todas as aplicações, o mesmo hardware com oito PEs foi utilizado para RN, portanto a sua área não mudou. A área total para EDCAC é variável, de acordo com a entrada de dados de cada aplicação e o número total de clusters. Estes parâmetros afetam diretamente o tamanho da memória, pois a variação ocorre na quantidade de linhas da memória (número de clusters) e na largura de cada linha (número de entradas). Além disso, o hardware para calcular a distância é afetado de maneira similar. O tamanho das entradas e saídas de cada benchmark bem como a quantidade de clusters é apresentado na Tabela *4.1*~~Tabela *4.1*~~ nas colunas "Tamanhos de Entradas e Saídas" e em "# Clusters", respectivamente.

Em EDCAC, a área consumida é menor que RN em todos benchmarks, exceto no JPEG. Neste exemplo, a grande quantidade de entradas e saídas do *hotspot* demanda uma memória de largura muito grande (64 floats), bem como uma estrutura em árvore de 64 unidades de largura para calcular a distância. No entanto, como visto acima, o JPEG possui uma performance muito melhor que RN no EDCAC. Portanto, neste caso, existe uma margem para troca de performance por um hardware com consumo menor em área. Por exemplo, seria possível reduzir a largura da memória e aumentar o número de acessos e ciclos para calcular a distância.

Figura 4.7: Área utilizada pela implementação individual de cada benchmark.



Fonte: O autor

A fim de medir a qualidade da técnica, foi utilizada a mesma métrica de erro utilizada em (ESMAEILZADEH et al., 2012a), a qual é mostrada na coluna "*Métrica de erro*" na Tabela *4.1*~~Tabela *4.1*~~. Foram comparadas as saídas do algoritmo original com a do algoritmo aproximado utilizando três métricas distintas (erro médio relativo, taxa de erro e diferenças entre imagens) dependendo do tipo da aplicação. A coluna "% Erro EDCAC" apresenta a porcentagem de erro de cada aplicação.

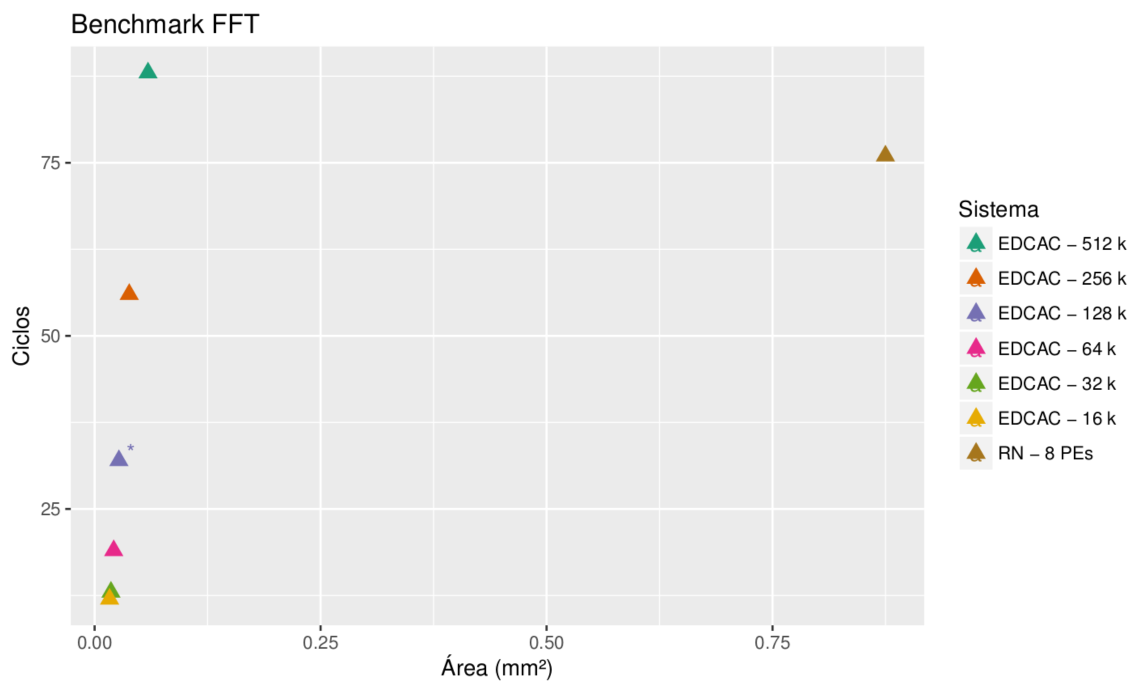
As aplicações utilizadas com EDCAC contém em média um erro entre 6% e 15% de acurácia, similar a medida de qualidade de outros trabalhos (BAEK; CHILIMBI, 2010; MISAILOVIC et al., 2010; SAMPSON et al., 2011; ESMAEILZADEH et al., 2012a, 2012b), que apresentam erros em torno de 10%. Uma funcionalidade do framework EDCAC é que o programador pode configurar facilmente os *trade-offs* entre performance e qualidade, simplesmente ajustando a quantidade de K clusters no algoritmo. Quanto maior a quantidade de clusters que a aplicação tiver, maior é a quantidade de memória que deverá ser utilizada para armazenar os valores aproximados, contudo maior é a computação requerida durante o processo de calcular a distância entre os valores.

A fim de compreender melhor a capacidade do framework de trocar qualidade por uma melhor performance computacional, será apresentado uma análise de espaço e projeto individual para cada benchmark, comparando EDCAC com a técnica de RNs.

Figura *4.8*~~Figura *4.8*~~, Figura *4.9*~~Figura *4.9*~~, Figura *4.10*~~Figura *4.10*~~, Figura *4.11*~~Figura *4.11*~~, Figura *4.12*~~Figura *4.12*~~ e Figura *4.13*~~Figura *4.13*~~, apresentam uma análise de espaço de projeto entre performance e área utilizada por cada configuração de EDCAC, variando a quantidade de clusters entre 16 e 512 clusters. Primeiro, é observado que em todas as análises a grande maioria das configurações de EDCAC possui uma quantidade de ciclos menor que RNs. Nesta análise, vale lembrar que diminuindo a quantidade de clusters consequentemente teremos uma performance melhor, porém não necessariamente teremos uma qualidade aceitável. Detalhes referentes à quantidade de clusters vs qualidade serão discutidos posteriormente.

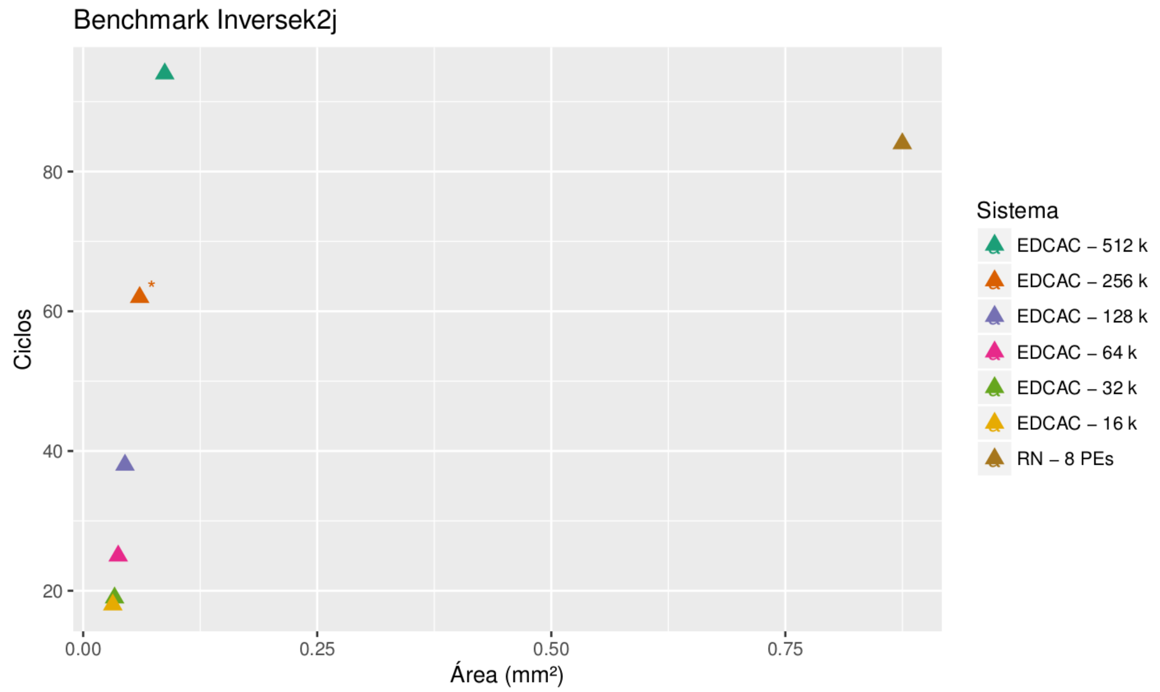
Também é observado que a área utilizada por cada aplicação é menor em todas as configurações de EDCAC, exceto para a aplicação JPEG, pois a utilização de memórias tem uma largura consideravelmente maior que as demais, como já discutido anteriormente.

Figura 4.8: Análise de espaço e de projeto do benchmark FFT.



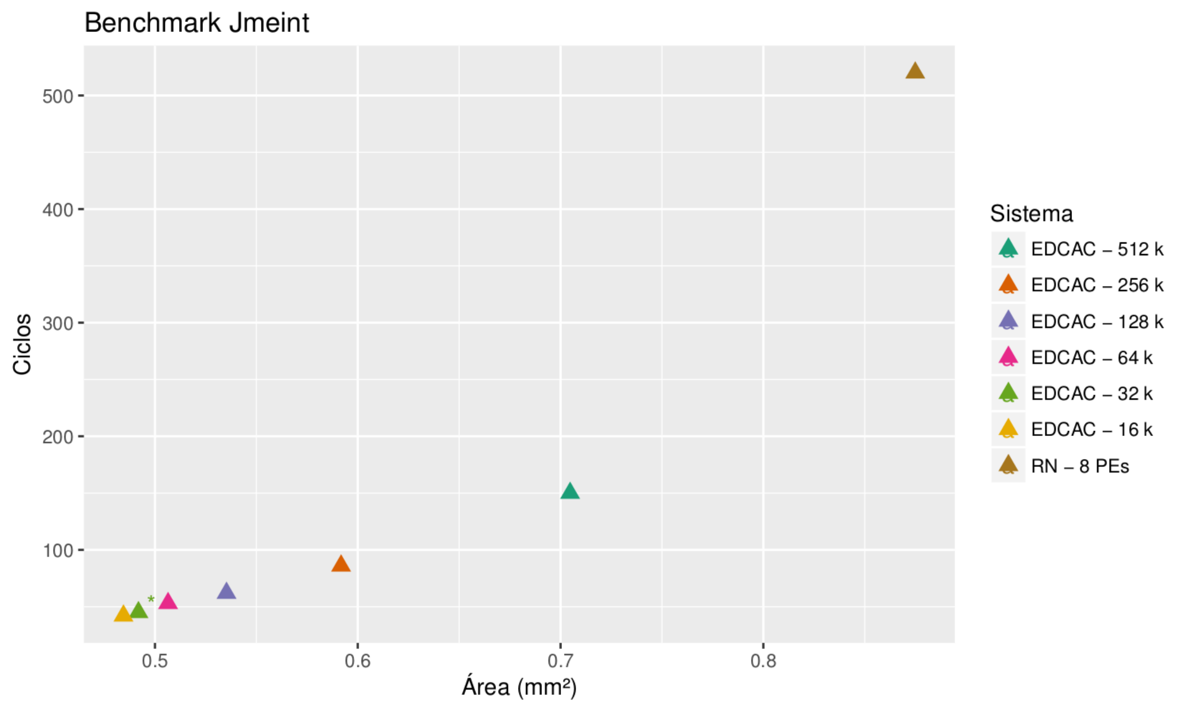
Fonte: O autor.

Figura 4.9: Análise de espaço e de projeto do benchmark Inversek2j.



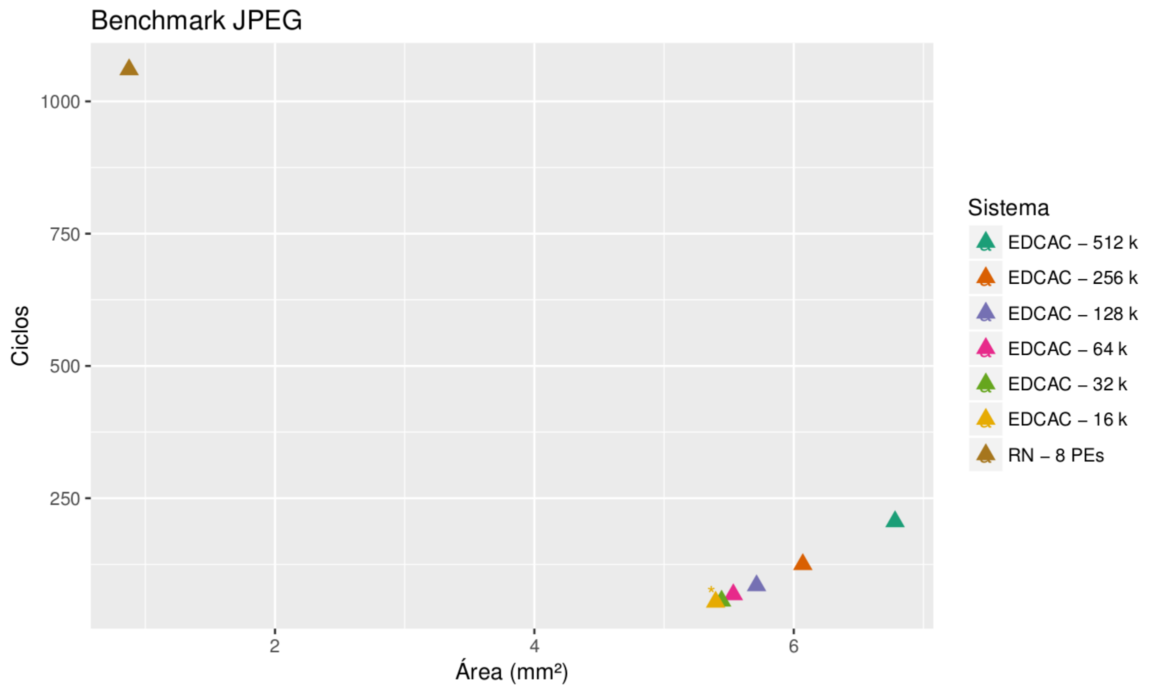
Fonte: O autor.

Figura 4.10: Análise de espaço e de projeto do benchmark Jmeint.



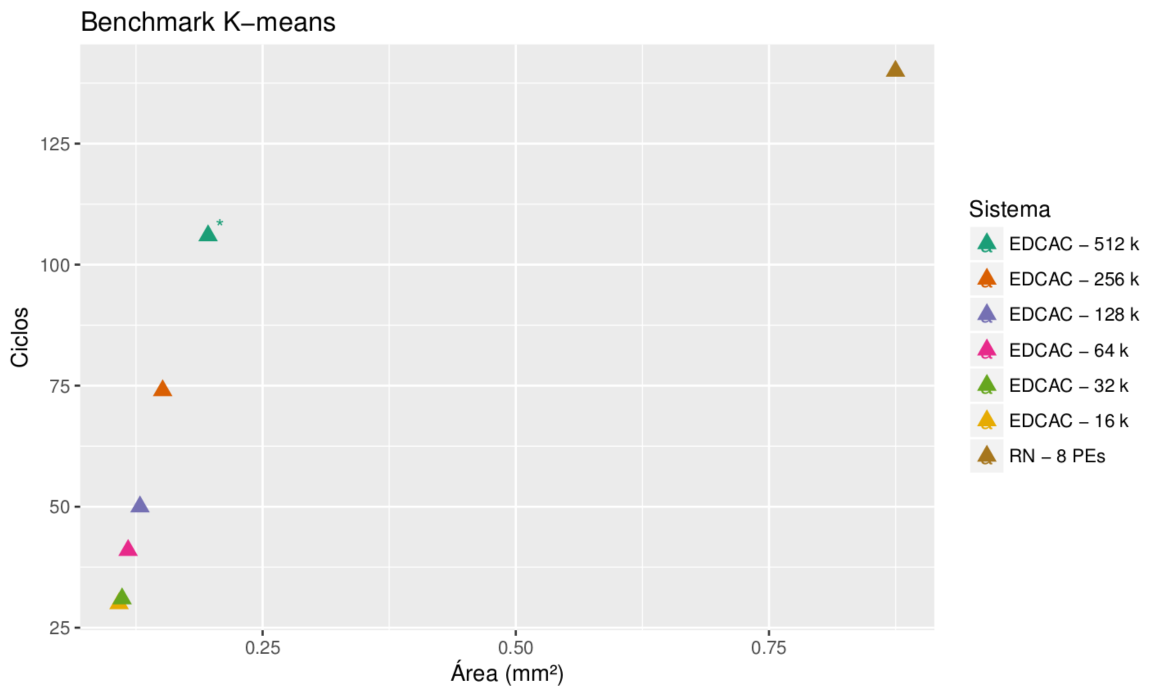
Fonte: O autor.

Figura 4.11: Análise de espaço e de projeto do benchmark JPEG.



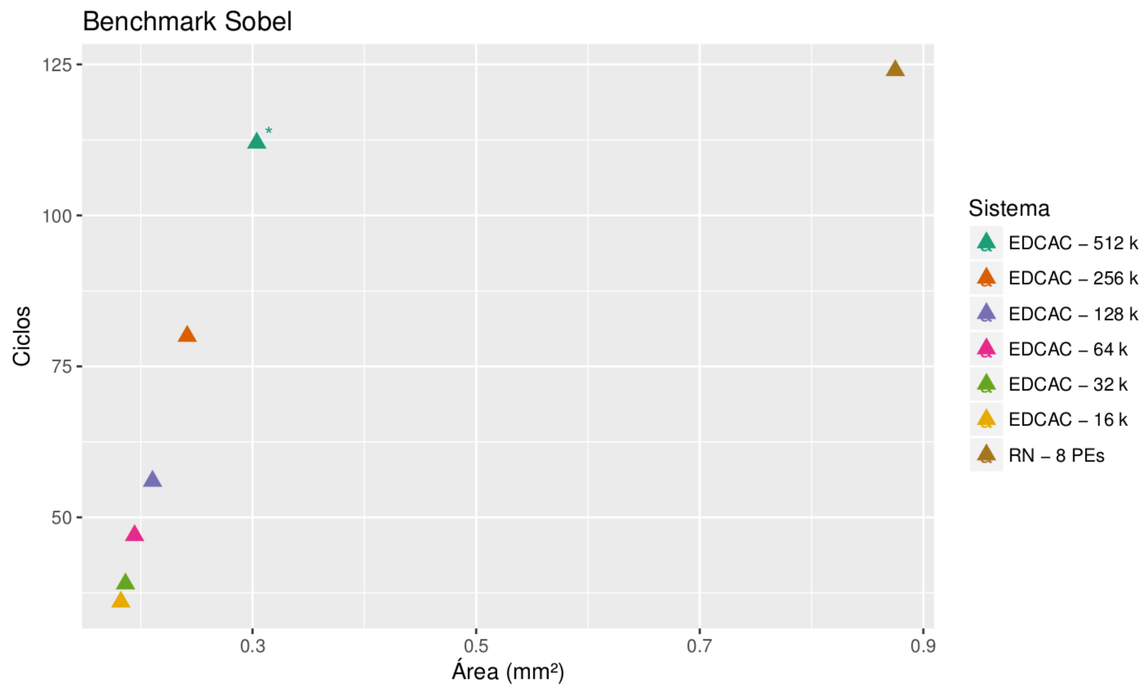
Fonte: O autor.

Figura 4.12: Análise de espaço e de projeto do benchmark K-means.



Fonte: O autor.

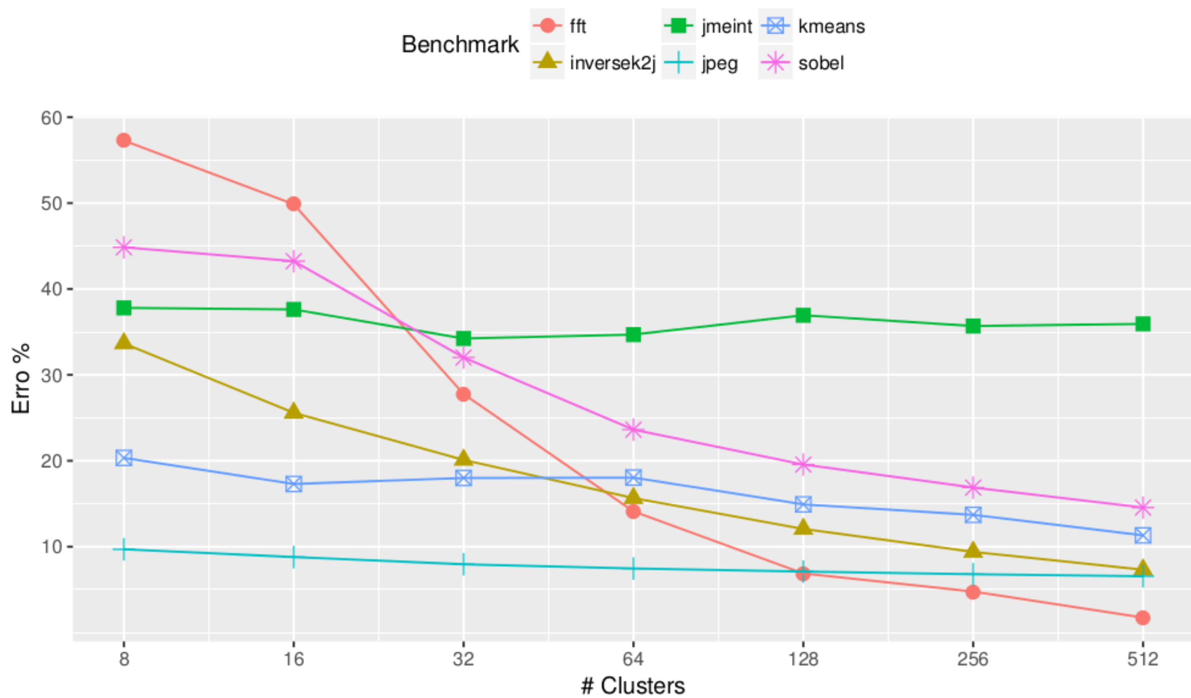
Figura 4.13: Análise de espaço e de projeto do benchmark Sobel.



Fonte: O autor.

Para entender melhor a variação em qualidade, é apresentado na Figura 4.14~~Figura 4.14~~ a porcentagem de qualidade no resultado de cada aplicação, quando a mudança do número de clusters ocorre entre 8 e 512. Para muitas aplicações como FFT, JPEG, e Sobel, a qualidade aumenta rapidamente quando a quantidade de clusters dobra. A aplicação Jmeint apresenta um comportamento diferente dentre todas as aplicações, pois nenhuma configuração é capaz de alcançar uma taxa de erro aceitável, sendo todas acima de 30%. A razão pela qual a aplicação apresenta este comportamento é devido ao resultado ser um tipo booleano, o que faz com que ele se torne muito sensível a variações, já que o resultado pode assumir apenas dois valores (verdadeiro ou falso). Vale lembrar que a técnica utilizada com RNs apresenta o mesmo comportamento com o benchmark Jmeint.

Figura 4.14: Taxa de erro alcançado por EDCAC com quantidade distintas de clusters.



Fonte: O autor

4.4 Considerações finais

Este capítulo demonstrou o potencial que a técnica de aproximação de dados tem quando substitui uma região de código “comportada” pela utilização de dados aproximados. A aproximação de dados é capaz de converter diferentes regiões de códigos para uma representação comum. Utilizando técnicas de *Machine Learning,* tais como clusterização de dados, como uma forma de representar estes dados comuns, criou-se uma nova classe de aceleradores de hardware que oferecem ganhos em energia e performance, que até então nunca foram exploradas. Toda técnica de aproximação introduz um grau de erro em cada aplicação. O erro inserido neste trabalho é menor ou semelhante a trabalhos anteriores. Providenciando esta solução *end-to-end*, as aplicações avaliadas neste trabalho obtiveram um *speedup* médio de 1.41x e consumo energético 85% menor que as RNs.

Tradicionalmente, implementações em hardware para obter maior eficiência sempre foram desenvolvidas utilizando redes neurais. Esta nova abordagem está alinhada com os trabalhos que vêm sendo desenvolvidos atualmente, utilizando componentes dedicados como aceleradores de software.

5 CONCLUSÃO

Este trabalho propôs uma nova técnica de computação aproximativa para uso em processadores de propósito geral utilizando técnicas de *Machine Learning,* como K-means cluster, até então nunca utilizada para este propósito. Determinou-se quais técnicas e métodos seriam aplicados no intuito de aumentar a performance e diminuir o consumo energéticos de aplicações de propósito geral. Para isso, seis aplicações dos mais diversos domínios foram utilizadas para análise da técnica proposta, que apesar da pequena quantidade de aplicações, as mesmas representam uma quantidade significativa em aplicações de propósito geral.

Os resultados das avaliações realizadas através de simulações foram comparados com trabalhos anteriores, que utilizavam redes neurais para computação aproximativa, permitindo constatar que a aplicabilidade do EDCAC é viável, considerando os benefícios em termos de performance e energia. Todas as aplicações apresentaram um speedup positivo, no entanto a aplicação que obteve maior desempenho foi a FFT, alcançando 1.96x de speedup, quando comparado com a aplicação baseline. Já em consumo energético notou-se que todas aplicações alcançaram um consumo energético 50% menor quando comparado com RN.

Com base nos resultados obtidos, foi possível verificar que a técnica não adiciona nenhum tipo de *overhead* relevante, seja em termos de acessos a memória ou de utilização CPU (processamento). Dessa forma, é correto assumir que a utilização e implementação da nova abordagem pode trazer tanto benefícios em termos de performance como de energia, no que diz respeito à utilização de técnicas de Machine Learning em conjunto com armazenamento em memória. As avaliações realizadas demonstram que com o avanço da tecnologia é possível integrar uma grande quantidade de memórias em um microprocessador e, desta forma, a adição de memória não é um problema significativo.

Por fim, a abordagem mostrou que embora os aceleradores aproximativos em hardware tradicionalmente sejam implementados utilizando redes neurais, é possível desenvolver com sucesso uma nova forma de fazer computação aproximativa através de clusterização de dados e reúso.

5.1 Trabalhos Futuros

Trabalhos sobre aproximação de dados e aceleradores de hardware provém resultados satisfatórios em performance e benefícios relacionados com consumo energético, porém pesquisas adicionais devem abordar algumas limitações, como: 1) aplicabilidade, 2) anotação do código, 3) controle de erro e 4) conjunto de dados para treinamento.

Aplicabilidade: a técnica utilizada nesta pesquisa não é capaz de aproximar qualquer área de código. Como mencionado na seção 3.1.1.1, a região de código deve satisfazer os seguintes critérios:

- A região deve ser executada com uma grande frequência para beneficiar-se da aceleração;
- A função deve tolerar um certo grau de imprecisão sem causar falhas catastróficas;
- E por fim, a região deve conter um número fixo de pares de entrada e saídas, os quais possam ser identificados estaticamente.

Embora estes critérios sejam a base para identificar regiões de códigos factíveis a aproximação, esta não garante 100% de precisão do resultado, como exemplo o Jmeint, que mesmo respeitando os critérios de aproximação, não provê resultados ideais. Desta forma, não existe um critério simples que identifique certa região do código a ser aproximada.

Trabalhos posteriores devem melhorar a forma de identificação estática da funcionalidade ou o comportamento de uma função que tenha tendência de ser aproximada.

Anotação do código: neste trabalho e em trabalhos anteriores, o programador necessita manualmente identificar a área a ser aproximada. Trabalhos futuros devem explorar técnicas para que o compilador tenha total liberdade de identificar qual função é passível de ser aproximada.

Controle de erro: resultados obtidos nesta pesquisa sugerem que a técnica de aproximação tem total capacidade de aproximar regiões de códigos com precisão, no entanto existe a possibilidade de algumas execuções do código aproximado não retornem o valor ideal, resultando em uma qualidade abaixo da média. Em outras palavras, sem que a aplicação seja executada com todas as possibilidades de entrada, é impossível identificar o pior cenário.

Por esta razão, trabalhos futuros devem explorar técnicas para evitar estes desvios. Uma possível solução seria desenvolver um “preditor” de erro, se a previsão for maior que um *threshold* previamente definido, o código original é executado sem utilizar o acelerador de hardware, caso contrário é utilizado e a aproximação é aplicada.

Conjunto de dados para o treinamento: como em (ESMAEILZADEH et al., 2012a) utilizando redes neurais, é necessário um conjunto de dados separados para treinar a rede neural ou gerar dados aproximados. Trabalhos futuros devem explorar um comportamento encontrado em todos os benchmarks, em que o código aproximado é executado inúmeras vezes e desta forma aplicar um esforço para desenvolver um novo método de treinamento *online*, ou seja, um tempo de execução de cada benchmark, de forma que seja possível dinamicamente gerar os dados aproximados necessários e assim ter dados prontos para utilização.

Para trabalhos futuros, é importante pensar em aprimorar todas as fases citadas neste trabalho, desde a intervenção do programador para identificar o código a ser aproximado até o algoritmo para a busca do valor aproximado. Neste contexto, todas as ideias propostas visam ao aumento do desempenho de um processador de propósito geral, assim como disponibilizam novas técnicas/abordagens para a comunidade acadêmica.

REFERÊNCIAS

AGARWAL, S.; MOZAFARI, B.; PANDA, A.; et al. BlinkDB: queries with bounded errors and bounded response times on very large data. Proceedings of the 8th ACM European Conference on Computer Systems. **Anais...** . p.29–42, 2013.

ALVAREZ, C.; CORBAL, J.; VALERO, M. Fuzzy memoization for floating-point multimedia applications. **IEEE Transactions on Computers**, v. 54, n. 7, p. 922–927, 2005. IEEE.

BAEK, W.; CHILIMBI, T. M. Green: a framework for supporting energy-conscious programming using controlled approximation. ACM Sigplan Notices. **Anais...** . v. 45, p.198–209, 2010.

BINKERT, N.; BECKMANN, B.; BLACK, G.; et al. The gem5 simulator. **ACM SIGARCH Computer Architecture News**, v. 39, n. 2, p. 1–7, 2011. ACM.

BOHR, M. A 30 year retrospective on Dennard's MOSFET scaling paper. **IEEE Solid-State Circuits Society Newsletter**, v. 12, n. 1, p. 11–13, 2007. IEEE.

CADENCE, R. T. L. Compiler User's Manual. . Cadence.

CHANG, I. J.; MOHAPATRA, D.; ROY, K. A priority-based 6T/8T hybrid SRAM architecture for aggressive voltage scaling in video applications. **IEEE transactions on circuits and systems for video technology**, v. 21, n. 2, p. 101–112, 2011. IEEE.

CHEN, T.; CHEN, Y.; DURANTON, M.; et al. BenchNN: On the broad potential application scope of hardware neural network accelerators. Workload Characterization (IISWC), 2012 IEEE International Symposium on. **Anais...** . p.36–45, 2012.

DEZA, M. M.; DEZA, E. Encyclopedia of distances. **Encyclopedia of Distances**. p.1–583, 2009. Springer.

DUDA, R. O.; HART, P. E.; STORK, D. G. **Pattern classification**. John Wiley & Sons, 2012.

ESMAEILZADEH, H.; SAMPSON, A.; CEZE, L.; BURGER, D. Neural acceleration for general-purpose approximate programs. Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture. **Anais...** . p.449–460, 2012a.

ESMAEILZADEH, H.; SAMPSON, A.; CEZE, L.; BURGER, D. Architecture support for disciplined approximate programming. ACM SIGPLAN Notices. **Anais...** . v. 47, p.301–312, 2012b.

FOROOZESH, J.; KHOSRAVANI, A.; MOHSENZADEH, A.; MESBAHI, A. H. Application of Artificial Intelligence (AI) modeling in kinetics of methane hydrate growth.

American Journal of Analytical Chemistry, v. 2013, 2013. Scientific Research Publishing.
FREEMAN, J. A.; SKAPURA, D. M. Neural networks: algorithms, applications and programming techniques. 1991. **Reading, Massachussets: Addison-Wesley**.
GRIGORIAN, B.; FARAHPOUR, N.; REINMAN, G. BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing. High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on. **Anais...** . p.615–626, 2015.
GRIGORIAN, B.; REINMAN, G. Accelerating divergent applications on simd architectures using neural networks. **ACM Transactions on Architecture and Code Optimization (TACO)**, v. 12, n. 1, p. 2, 2015. ACM.
GUPTA, V.; MOHAPATRA, D.; PARK, S. P.; RAGHUNATHAN, A.; ROY, K. IMPACT: imprecise adders for low-power approximate computing. Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design. **Anais...** . p.409–414, 2011.
GUPTA, V.; MOHAPATRA, D.; RAGHUNATHAN, A.; ROY, K. Low-power digital signal processing using approximate adders. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 32, n. 1, p. 124–137, 2013. IEEE.
HAN, J.; ORSHANSKY, M. Approximate computing: An emerging paradigm for energy-efficient design. Test Symposium (ETS), 2013 18th IEEE European. **Anais...** . p.1–6, 2013.
HERTZ, J. A.; KROGH, A.; PALMER, R. G. Introduction to the Theory of Neural Computation, volume 1 of Santa Fe Institute Studies in the Sciences of Complexity: Lecture Notes. , 1991. Addison-Wesley, Redwood City, CA.
HO, C.-H.; KRUIJF, M. DE; SANKARALINGAM, K.; et al. Mechanisms and evaluation of cross-layer fault-tolerance for supercomputing. Parallel Processing (ICPP), 2012 41st International Conference on. **Anais...** . p.510–519, 2012.
HOFFMANN, H.; SIDIROGLOU, S.; CARBIN, M.; et al. Dynamic knobs for responsive power-aware computing. ACM SIGPLAN Notices. **Anais...** . v. 46, p.199–212, 2011.
JOHNSON, S. C. Hierarchical clustering schemes. **Psychometrika**, v. 32, n. 3, p. 241–254, 1967. Springer.
KAHNG, A. B.; KANG, S. Accuracy-configurable adder for approximate arithmetic designs. Proceedings of the 49th Annual Design Automation Conference. **Anais...** . p.820–825, 2012.
KEDEM, Z. M.; MOONEY, V. J.; MUNTIMADUGU, K. K.; PALEM, K. V. An approach to energy-error tradeoffs in approximate ripple carry adders. Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design. **Anais...** . p.211–216, 2011.
KRUIJF, M. DE; NOMURA, S.; SANKARALINGAM, K. Relax: An architectural

framework for software recovery of hardware faults. **ACM SIGARCH Computer Architecture News**, v. 38, n. 3, p. 497–508, 2010. ACM.

KUMAR, A.; RABAEY, J.; RAMCHANDRAN, K. SRAM supply voltage scaling: A reliability perspective. Quality of Electronic Design, 2009. ISQED 2009. Quality Electronic Design. **Anais...** . p.782–787, 2009.

LIU, S.; PATTABIRAMAN, K.; MOSCIBRODA, T.; ZORN, B. G. Flicker: Saving Refresh-Power in Mobile Devices through Critical Data Partitioning. Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),. **Anais...** , 2009.

LUCAS, J.; ALVAREZ-MESA, M.; ANDERSCH, M.; JUURLINK, B. Sparkk: Quality-scalable approximate storage in DRAM. The Memory Forum. **Anais...** . p.1–9, 2014.

MCAFEE, L.; OLUKOTUN, K. EMEURO: A framework for generating multi-purpose accelerators via deep learning. Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. **Anais...** . p.125–135, 2015.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, v. 5, n. 4, p. 115–133, 1943. Springer.

MENGTE, J.; RAGHUNATHAN, A.; CHAKRADHAR, S.; BYNA, S. Exploiting the forgiving nature of applications for scalable parallel execution. Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on. **Anais...** . p.1–12, 2010.

MIAO, J.; HE, K.; GERSTLAUER, A.; ORSHANSKY, M. Modeling and synthesis of quality-energy optimal approximate adders. Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on. **Anais...** . p.728–735, 2012.

MISAILOVIC, S.; SIDIROGLOU, S.; HOFFMANN, H.; RINARD, M. Quality of service profiling. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. **Anais...** . p.25–34, 2010.

MISAILOVIC, S.; SIDIROGLOU, S.; RINARD, M. C. Dancing with uncertainty. Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability. **Anais...** . p.51–60, 2012.

MITCHELL, T. M. Machine learning. 1997. **Burr Ridge, IL: McGraw Hill**, v. 45, n. 37, p. 870–877, 1997.

MOHAPATRA, D.; CHIPPA, V. K.; RAGHUNATHAN, A.; ROY, K. Design of voltage-scalable meta-functions for approximate computing. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011. **Anais...** . p.1–6, 2011.

MOREAU, T.; WYSE, M.; NELSON, J.; et al. SNNAP: Approximate computing on programmable socs via neural acceleration. High Performance Computer Architecture

(HPCA), 2015 IEEE 21st International Symposium on. **Anais...** . p.603–614, 2015.

MURALIMANOHAR, N.; BALASUBRAMONIAN, R.; JOUPPI, N. P. CACTI 6.0: A tool to model large caches. **HP Laboratories**, p. 22–31, 2009.

NIU, F.; RECHT, B.; RÉ, C.; WRIGHT, S. J. H. A lock-free approach to parallelizing stochastic gradient descent. arXiv preprint. **arXiv preprint arXiv:1106.5730**, 2011.

OLUKOTUN, K.; HAMMOND, L. The future of microprocessors. **Queue**, v. 3, n. 7, p. 26–29, 2005. ACM.

RAY, S.; TURI, R. H. Determination of number of clusters in k-means clustering and application in colour image segmentation. Proceedings of the 4th international conference on advances in pattern recognition and digital techniques. **Anais...** . p.137–143, 1999.

RENGANARAYANA, L.; SRINIVASAN, V.; NAIR, R.; PRENER, D. Programming with relaxed synchronization. Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability. **Anais...** . p.41–50, 2012.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. **Learning internal representations by error propagation**. 1985.

SACK, J.-R.; URRUTIA, J. **Handbook of computational geometry**. Elsevier, 1999.

SAMPSON, A.; DIETL, W.; FORTUNA, E.; et al. EnerJ: Approximate data types for safe and general low-power computation. ACM SIGPLAN Notices. **Anais...** . v. 46, p.164–174, 2011.

SEN, S.; GILANI, S.; SRINATH, S.; SCHMITT, S.; BANERJEE, S. Design and implementation of an approximate communication system for wireless media applications. ACM SIGCOMM Computer Communication Review. **Anais...** . v. 40, p.15–26, 2010.

SHAFIQUE, M.; AHMAD, W.; HAFIZ, R.; HENKEL, J. A low latency generic accuracy configurable adder. Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE. **Anais...** . p.1–6, 2015.

SHOUSHTARI, M.; BANAIYANMOFRAD, A.; DUTT, N. Exploiting partially-forgetful memories for approximate computing. **IEEE Embedded Systems Letters**, v. 7, n. 1, p. 19–22, 2015. IEEE.

SIDIROGLOU-DOUSKOS, S.; MISAILOVIC, S.; HOFFMANN, H.; RINARD, M. Managing performance vs. accuracy trade-offs with loop perforation. Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. **Anais...** . p.124–134, 2011.

ST AMANT, R.; YAZDANBAKHSH, A.; PARK, J.; et al. General-purpose code acceleration with limited-precision analog computation. **ACM SIGARCH Computer Architecture**

News, v. 42, n. 3, p. 505–516, 2014. ACM.

STANLEY-MARBELL, P.; MARCULESCU, D. A programming model and language implementation for concurrent failure-prone hardware. Proceedings of the 2nd Workshop on Programming Models for Ubiquitous Parallelism, PMUP06. **Anais...** , 2006.

THEODORIDIS, S.; KOUTROUMBAS, K. Pattern recognition. , 2003. Academic press.

TONG, J. Y. F.; NAGLE, D.; RUTENBAR, R. A. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 8, n. 3, p. 273–286, 2000. IEEE.

VERMA, A. K.; BRISK, P.; IENNE, P. Variable latency speculative addition: A new paradigm for arithmetic circuit design. Proceedings of the conference on Design, automation and test in Europe. **Anais...** . p.1250–1255, 2008.

WEBER, M.; PUTIC, M.; ZHANG, H.; LACH, J.; HUANG, J. Balancing adder for error tolerant applications. Circuits and Systems (ISCAS), 2013 IEEE International Symposium on. **Anais...** . p.3038–3041, 2013.

XU, Q.; MYTKOWICZ, T.; KIM, N. S. Approximate computing: A survey. **IEEE Design & Test**, v. 33, n. 1, p. 8–22, 2016. IEEE.

YAZDANBAKHSH, A.; MAHAJAN, D.; ESMAEILZADEH, H.; LOTFI-KAMRAN, P. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. **IEEE Design & Test**, v. 34, n. 2, p. 60–68, 2017. IEEE.

YAZDANBAKHSH, A.; PARK, J.; SHARMA, H.; LOTFI-KAMRAN, P.; ESMAEILZADEH, H. Neural acceleration for gpu throughput processors. Proceedings of the 48th International Symposium on Microarchitecture. **Anais...** . p.482–493, 2015.

YE, R.; WANG, T.; YUAN, F.; KUMAR, R.; XU, Q. On reconfiguration-oriented approximate adder design and its application. Proceedings of the International Conference on Computer-Aided Design. **Anais...** . p.48–54, 2013.

YEH, T.; FALOUTSOS, P.; ERCEGOVAC, M.; PATEL, S.; REINMAN, G. The art of deception: Adaptive precision reduction for area efficient physics acceleration. Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on. **Anais...** . p.394–406, 2007.

ZHU, N.; GOH, W. L.; YEO, K. S. An enhanced low-power high-speed adder for error-tolerant application. Integrated Circuits, ISIC'09. Proceedings of the 2009 12th International Symposium on. **Anais...** . p.69–72, 2009.

A. APÊNDICE A

Tabela A.1~~Tabela A.1~~ apresenta dados de potência e área e temporização de cada componente utilizada na arquitetura do hardware, como subtratores, e multiplicadores. Estas informações foram obtidas através da síntese lógico utilizando ferramentas Cadence RTL Compiler.

Tabela A.1: Consumo de potência de cada unidade em ponto flutuante utilizadas pelo hardware de cálculo da distância de Manhattan, bem como dados de área e temporização.

Unit	Period(ns)	Cells	Area(um2)	Leakage Power (nW)	Dynamic Power (nW)	Power
FP-Add	1043	2610	11729	2075.8	8816958	8819033
FP-Mul	1147	1147	21556	3668.339	14069963	14073631

Fonte: O autor

B. APÊNCIDE B

Este apêndice contém informações detalhadas dos componentes utilizados no modelo proposto nesta dissertação. A potência, área e tempo de acesso de cada memória apresentado nas tabelas a seguir apresentam informações dos benchmarks FFT, Inversek2j, Jmeint, JPEG, K-means e sobel, estas informações foram extraídas utilizando a ferramenta CACTI (Muralimanohar et al., 2009). As tabelas apresentam informações das memórias utilizadas para armazenar vetores de entrada e saída de cada aplicação, é apresentado informações variando entre 16 e 512 linhas cada memória, pois estas foram as informações utilizadas para definir a configuração apresentada neste trabalho.

Tabela B.1: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de entrada do benchmark FFT.

# Lines	Line Size (B)	# Rd Ports	Access Time(ns)	Cycle Time (ns)	Area (mm2)	Leakage Power	Read Energy
16	4	1	0.311047	0.21269	0.001056583	0.000105208	1.28135E-012
32	4	1	0.366548	0.23112	0.001629256	0.000202641	1.56241E-012
64	4	1	0.373431	0.232211	0.002719238	0.000376304	1.99568E-012
128	4	1	0.405052	0.274056	0.004846811	0.000728329	2.58543E-012
256	4	1	0.410872	0.274499	0.008438782	0.00136083	3.46168E-012
512	4	1	0.500226	0.337568	0.015805822	0.00307651	4.76519E-012

Fonte: O autor

Tabela B.2: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de saída do benchmark FFT.

# Lines	Line Size (B)	# Rd Ports	Access Time(ns)	Cycle Time (ns)	Area (mm2)	Leakage Power	Read Energy
16	8	1	0.329461	0.247115	0.002720413	0.000192543	2.46984E-012
32	8	1	0.386799	0.267382	0.003678185	0.000364372	2.97459E-012
64	8	1	0.393775	0.268566	0.005623395	0.000696457	3.80941E-012
128	8	1	0.407242	0.274499	0.009245029	0.00137394	4.94158E-012
256	8	1	0.493061	0.326936	0.017073724	0.00265783	7.09021E-012
512	8	1	0.501859	0.338177	0.030478484	0.00598267	9.15953E-012

Fonte: O autor

Tabela B.3: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de entrada e saída do benchmark Inversek2j.

# Lines	Line Size(B)	# Rd Ports	Access Time(ns)	Cycle Time(ns)	Area(mm2)	Leakage Power	Read Energy
16	8	1	0.329461	0.247115	0.002720413	0.000192543	2.46984E-012
32	8	1	0.386799	0.267382	0.003678185	0.000364372	2.97459E-012
64	8	1	0.393775	0.268566	0.005623395	0.000696457	3.80941E-012
128	8	1	0.407242	0.274499	0.009245029	0.00137394	4.94158E-012
256	8	1	0.493061	0.326936	0.017073724	0.00265783	7.09021E-012
512	8	1	0.501859	0.338177	0.030478484	0.00598267	9.15953E-012

Fonte: O autor

Tabela B.4: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de entrada do benchmark Jmeint.

# Lines	Line Size(B)	# Rd Ports	Access Time(ns)	Cycle Time(ns)	Area(mm2)	Leakage Power	Read Energy
16	72	1	0.383758	0.428598	0.1361404	0.00164959	2.09991E-011
32	72	1	0.440675	0.448445	0.1429658	0.00313221	2.5083E-011
64	72	1	0.459478	0.457583	0.1564747	0.00612854	3.32414E-011
128	72	1	0.545146	0.509145	0.1831728	0.0120663	4.94605E-011
256	72	1	0.662946	0.614742	0.2360731	0.0238518	8.18549E-011
512	72	1	0.905494	0.82685	0.3416436	0.0474212	1.46657E-010

Fonte: O autor

Tabela B.5: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de saída do benchmark Jmeint.

# Lines	Line Size(B)	# Rd Ports	Access Time(ns)	Cycle Time(ns)	Area(mm2)	Leakage Power	Read Energy
16	8	1	0.329461	0.247115	0.002720413	0.000192543	2.46984E-012
32	8	1	0.386799	0.267382	0.003678185	0.000364372	2.97459E-012
64	8	1	0.393775	0.268566	0.005623395	0.000696457	3.80941E-012
128	8	1	0.407242	0.274499	0.009245029	0.00137394	4.94158E-012
256	8	1	0.493061	0.326936	0.017073724	0.00265783	7.09021E-012
512	8	1	0.501859	0.338177	0.030478484	0.00598267	9.15953E-012

Fonte: O autor

Tabela B.6: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de entrada e saída do benchmark JPEG.

# Lines	Line Size(B)	# Rd Ports	Access Time(ns)	Cycle Time(ns)	Area(mm2)	Leakage Power	Read Energy
16	256	1	0.620979	1.31339	1.548946	0.00530036	7.40011E-011
32	256	1	0.695334	1.35068	1.571905	0.0100132	8.83748E-011
64	256	1	0.739284	1.38496	1.61713	0.0194701	1.17113E-010
128	256	1	0.851284	1.46286	1.706757	0.0383288	1.74492E-010
256	256	1	0.982974	1.58234	1.884991	0.0759564	2.89205E-010
512	256	1	1.22586	1.79479	2.240708	0.15121	5.18644E-010

Fonte: O autor

Tabela B.7: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de entrada do benchmark K-means.

# Lines	Line Size(B)	# Rd Ports	Access Time(ns)	Cycle Time(ns)	Area(mm2)	Leakage Power	Read Energy
16	24	1	0.343598	0.282792	0.01839137	0.000608611	7.12861E-012
32	24	1	0.397488	0.299611	0.02083107	0.00115994	8.52813E-012
64	24	1	0.416782	0.30924	0.02570998	0.00229371	1.1318E-011
128	24	1	0.502236	0.360587	0.03529075	0.00450632	1.67998E-011
256	24	1	0.619501	0.465651	0.05409708	0.00884161	2.77198E-011
512	24	1	0.624319	0.466625	0.09158447	0.0171325	3.25515E-011

Fonte: O autor

Tabela B.8: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de saída do benchmark K-means.

# Lines	Line Size(B)	# Rd Ports	Access Time(ns)	Cycle Time(ns)	Area(mm2)	Leakage Power	Read Energy
16	4	1	0.311047	0.21269	0.001056583	0.000105208	1.28135E-012
32	4	1	0.366548	0.23112	0.001629256	0.000202641	1.56241E-012
64	4	1	0.373431	0.232211	0.002719238	0.000376304	1.99568E-012
128	4	1	0.405052	0.274056	0.004846811	0.000728329	2.58543E-012
256	4	1	0.410872	0.274499	0.008438782	0.00136083	3.46168E-012
512	4	1	0.500226	0.337568	0.015805822	0.00307651	4.76519E-012

Fonte: O autor

Tabela B.9: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de entrada do benchmark Sobel.

# Lines	Line Size(B)	# Rd Ports	Access Time(ns)	Cycle Time(ns)	Area(mm2)	Leakage Power	Read Energy
16	36	1	0.35411	0.31754	0.03779317	0.000881712	1.06043E-011
32	36	1	0.408428	0.334788	0.04134472	0.00167872	1.26749E-011
64	36	1	0.427367	0.344062	0.04841144	0.00330384	1.68069E-011
128	36	1	0.512906	0.395494	0.06233182	0.00649915	2.4973E-011
256	36	1	0.630386	0.500771	0.08978176	0.0127998	4.12616E-011
512	36	1	0.63523	0.501772	0.14461184	0.0248711	4.84047E-011

Fonte: O autor

Tabela B.10: Área, potência e tempo de acesso da memória utilizada para armazenar o vetor de saída do benchmark Sobel.

# Lines	Line Size(B)	# Rd Ports	Access Time(ns)	Cycle Time(ns)	Area(mm2)	Leakage Power	Read Energy
16	4	1	0.311047	0.21269	0.001056583	0.000105208	1.28135E-012
32	4	1	0.366548	0.23112	0.001629256	0.000202641	1.56241E-012
64	4	1	0.373431	0.232211	0.002719238	0.000376304	1.99568E-012
128	4	1	0.405052	0.274056	0.004846811	0.000728329	2.58543E-012
256	4	1	0.410872	0.274499	0.008438782	0.00136083	3.46168E-012
512	4	1	0.500226	0.337568	0.015805822	0.00307651	4.76519E-012

Fonte: O autor