Classificador de kernels para mapeamento em plataforma de computação híbrida composta por FPGA e GPP

Alexandre Shigueru Sumoyama

Alexandre Shigueru Sumoyama

Classificador de kernels para mapeamento em plataforma de computação híbrida composta por FPGA e GPP

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Vanderlei Bonato

USP – São Carlos
Julho de 2016

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

Sumoyama, Alexandre Shigueru
S634c Classificador de kernels para mapeamento em plataforma de computação híbrida composta por FPGA e GPP / Alexandre Shigueru Sumoyama; orientador Vanderlei Bonato. - São Carlos - SP, 2016.
88 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2016.

1. FPGA. 2. GPP. 3. DAMICORE. 4. Mapeamento de código. 5. classificador de *kernels*. I. Bonato, Vanderlei, orient. II. Título.

Alexandre Shigueru Sumoyama

Classifier of kernels for hybrid computing platform mapping composed by FPGA and GPP

Master dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in partial fulfillment of the requirements for the degree of the Master Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Vanderlei Bonato

USP – São Carlos
July 2016

Este trabalho é dedicado às todas pessoas que
contribuíram para a realização dessa dissertação.

AGRADECIMENTOS

A conclusão de meu trabalho de mestrado e confecção desta dissertação representam um grande marco em minha vida. Agradeço a todas as pessoas que de algum modo ajudaram na concretização deste trabalho.

À Universidade de São Paulo, e o Instituto de Ciências Matemáticas e de Computação, apresento meus agradecimentos pela oportunidade de desenvolvimento deste trabalho, bem como pelo conhecimento transmitido no decorrer do curso.

À CAPES pelo auxílio financeiro que permitiu que esse trabalho fosse realizado.

Ao professor Vanderlei Bonato, agradeço pela disponibilidade, dedicação, e auxílio prestado ao orientar meu trabalho.

À minha família, meus amigos e minha namorada, Flor Karina, agradeço pelo constante incentivo e pela confiança depositada em mim.

Finalmente, à todos que, diretamente ou indiretamente, contribuíram para a concretização deste trabalho, meus sinceros agradecimentos

“Se não puder voar, corra.
Se não puder correr, ande.
Se não puder andar, rasteje,
mas continue em frente de qualquer jeito.”
(Martin Luther King)

RESUMO

SUMOYAMA, A. S.. **Classificador de kernels para mapeamento em plataforma de computação híbrida composta por FPGA e GPP**. 2016. 88 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

O aumento constante da demanda por sistemas computacionais cada vez mais eficientes tem motivado a busca por sistemas híbridos customizados compostos por GPP (*General Purpose Processor*), FPGAs (*Field-Programmable Gate Array*) e GPUs (*Graphics Processing Units*). Quando utilizados em conjunto possibilitam otimizar a relação entre desempenho e consumo de energia. Tais sistemas dependem de técnicas que façam o mapeamento mais adequado considerando o perfil do código fonte. Nesse sentido, este projeto propõe uma técnica para realizar o mapeamento entre GPP e FPGA. Para isso, utilizou-se como base uma abordagem de mineração de dados que avalia a similaridade entre código fonte. A técnica aqui desenvolvida obteve taxas de acertos de 65,67% para códigos sintetizados para FPGA com a ferramenta *LegUP* e 59,19% para Impulse C, considerando que para GPP o código foi compilado com o GCC (GNU Compiler Collection) utilizando o suporte a OpenMP. Os resultados demonstraram que esta abordagem pode ser empregada como um ponto de decisão inicial no processo de mapeamento em sistemas híbridos, somente analisando o perfil do código fonte sem que haja a necessidade de execução do mesmo para a tomada de decisão.

Palavras-chave: FPGA, GPP, DAMICORE, Mapeamento de código, classificador de *kernels*.

ABSTRACT

SUMOYAMA, A. S.. **Classificador de kernels para mapeamento em plataforma de computação híbrida composta por FPGA e GPP**. 2016. 88 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

The steady increasing on demand for efficient computer systems has been motivated the search for customized hybrid systems composed by GPP (general purpose processors), FPGAs (Field-Programmable Gate Array) and GPUs (Graphics Processing Units). When they are used together allow to exploit their computing resources to optimize performance and power consumption. Such systems rely on techniques make the most appropriate mapping considering the profile of source code. Thus, this project proposes a technique to perform the mapping between GPP and FPGA. For this, it is applied a technique based on a data mining approach that evaluates the similarity between source code. The proposed method obtained hit rate 65.67% for codes synthesized in FPGA using LegUP tool and 59.19% for Impulse C tool, whereas for GPP, the source code was compiled on GCC (GNU Compiler Collection) using OpenMP. The results demonstrated that this approach can be used as an initial decision point on the mapping process in hybrid systems, only analyzing the profile of the source code without the need for implementing it for decision-making.

Key-words: FPGA, GPP, Code Mapping, DAMICORE, Data Mining, Classifier of Kernels.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CSP ..... *Communicating Sequential Process*
DAMICORE *DAta MIning of COde REpositories*
DES ..... *Data Encryption Standard*
DSP ..... *Digital Signal Processor*
FPGA .... *Field-Programmable Gate Array*
GPU ..... *Graphics Processing Unit*
LLVM .... *Low-Level Virtual Machine*
NCD ..... *Normalized Compression Distance*
NJ ..... Neighbor Joining
SIFT ..... *Scale-Invariant Feature Transform*
SMP ..... *Symmetric Multi-Processing*
ULP ..... *Ultra-Low Power*

SUMÁRIO

CAPÍTULO

1

INTRODUÇÃO

1.1 Contextualização e motivação

Muitas aplicações, chamadas de *supercomputing applications* ou *superapplications*, demandam poder computacional elevado com tendência de crescimento (KIRK; HWU, 2010). Diante deste panorama, mudanças severas nas arquiteturas dos computadores têm ocorrido. Os projetos começaram a utilizar paralelismo em larga escala, *cores* heterogêneos e aceleradores com foco principal no ganho de desempenho (BORKAR *et al.*, 2005; BORKAR; CHIEN, 2011).

O uso de aceleradores de hardware em sistemas computacionais é um recurso que permite o aumento de desempenho e, em muitos casos, a melhora da eficiência computacional em relação ao consumo de energia. A relação *trabalho realizado* $\times$ *consumo de energia* é um problema que tem recebido cada vez mais atenção da comunidade científica (XU, 2012), (HARING *et al.*, 2012) e (LIU *et al.*, 2009).

Em (KECKLER *et al.*, 2011), é apresentada uma lista dos 500 computadores mais eficientes da atualidade. Dentre eles, observa-se que os mais eficientes são aqueles que utilizam *Graphics Processing Unit* (GPU) como aceleradores. As GPUs tem se popularizado como um meio de computação de alto desempenho devido a sua alta produtividade, não exigindo que o usuário tenha conhecimentos avançados de hardware. Estas unidades de processamento permitem que sejam exploradas características de paralelismo onde cálculos matemáticos podem ser executados de forma mais eficiente do que em processadores de propósito geral.

Outra tecnologia adotada para o desenvolvimento de aceleradores de hardware é o *Field-Programmable Gate Array* (FPGA). Diferentemente das GPUs, a sua utilização eficiente requer conhecimento avançado de hardware. Dada a flexibilidade de um dispositivo FPGA, é possível obter um hardware acelerador customizado para cada tipo de aplicação. (JONES *et al.*, 2010) apresenta uma comparação de produtividade no desenvolvimento entre as tecnologias GPU e FPGA.

Processadores *Digital Signal Processor* (DSP) também desempenham um papel importante na construção de aceleradores. FPGAs mais atuais possuem diversos blocos de DSP implementados como hardware para realizar operações matemáticas básicas de forma eficiente permitindo um ganho significativo em relação à área, desempenho e consumo de energia.

Dependendo das características de código de cada aplicação, ela pode ter um desempenho melhor ou pior para um determinado acelerador. No trabalho de (CHE *et al.*, 2008) é traçado um perfil de aplicações que possuem melhor desempenho em GPUs e em FPGAs baseado no custo de desenvolvimento, desempenho e restrições de hardware. Por exemplo, códigos com o perfil do método de eliminação de Gauss tendem a ter um desempenho melhor em GPUs pelo fato de não existir dependência no fluxo de dados permitindo que a computação seja feita em paralelo. Por outro lado, códigos com o perfil do algoritmo *Data Encryption Standard* (DES) é mais adequado para FPGAs devido a grande quantidade de operações a nível de *bits*.

Códigos relacionados ao processamento de imagens também podem tirar proveito de arquiteturas FPGAs. No trabalho de (BONATO; MARQUES; CONSTANTINIDES, 2008) é proposta uma arquitetura de hardware paralela em FPGA para o processamento de um algoritmo denominado *Scale-Invariant Feature Transform* (SIFT) para a extração de características de imagens. Esta arquitetura gerada atua como um acelerador no processamento de imagem. Neste trabalho foi possível visualizar que o uso eficiente do hardware operando em modo *pipeline* pode prover desempenho várias vezes superior ao de um processador de propósito geral com um consumo de energia extremamente baixo.

Na área financeira, o banco JP Morgan trabalhou em conjunto com a empresa Maxeler Technologies em um projeto para acelerar o cálculo nas análises de risco de crédito no mercado de derivativos (WESTON *et al.*, 2012). Estas análises envolvem modelos matemáticos complexos e o uso intensivo de computação. Foi empregada uma tecnologia híbrida com a utilização de um *cluster* onde cada nó era composto por um processador Intel Xeon E5430 conectado a quatro FPGAs Xilinx Virtex6. A empresa Celoxica, conhecida pelo desenvolvimento da ferramenta Handel-C, também tem se especializado nos últimos anos em soluções para o mercado financeiro, desenvolvendo aceleradores baseados em FPGA.

A Intel lançou no mercado em 2010, em conjunto com a Altera, um processador da família Atom série E6x5C (Intel Corporation, 2010), que possui um FPGA da família Arria II GX embutido (Altera Corporation, 2010). Esta combinação permite o desenvolvimento de sistemas customizados com alto poder de processamento e baixo consumo de energia, já que esta família de processadores é classificada pela Intel como *Ultra-Low Power* (ULP). Em 2013, a Intel anunciou (Intel Corporation, 2013) um acordo com a Altera para desenvolver novos FPGAs utilizando a tecnologia de transistores *tri-gate* de 14nm da Intel. Em 2015, a Altera foi adquirida pela Intel (Intel Corporation, 2015). De acordo com a Intel, a tendência das novas tecnologias de CPU começarão a ser fabricadas com FPGA acoplada. O FPGA, nesse contexto, funcionará como um co-processador trabalhando em conjunto com a CPU (GPP) . Dessa maneira, é possível

dividir a carga de trabalho entre ambos de modo que o FPGA ataque seções críticas de uma aplicação que um processador convencional x86 não consiga executar de forma eficiente, obtendo uma melhor performance aliado com o consumo eficiente de energia.

O uso de um conjunto de dispositivos de *hardware* a fim de melhorar o desempenho das aplicações remete ao conceito de computação heterogênea. As aplicações desse conceito é comumente aplicado nas tecnologias *multicores* de GPP e também GPP-GPU. Essas tecnologias já estão consolidadas no mercado por proverem ambientes de desenvolvimento que permitem explorar seus respectivos recursos. Do lado do *multicores* para GPP, tem-se as ferramentas OpenMP e OpenMPI no quais são possíveis paralelizar uma aplicação em uma única GPP ou em um conjunto do mesmo, respectivamente. Em contrapartida, para CPU+GPU, podemos citar as ferramentas CUDA, exclusiva para GPU da NVIDIA, e o OpenCL para programação para qualquer modelo de GPU. Recentemente, foi desenvolvida uma plataforma de programação denominada AOCL(*Altera For OpenCL*). Essa ferramenta possui primitivas nativas do OpenCL e permite tanto a programação de aplicações para GPUs quanto para FPGAs.

A adoção de plataformas heterogêneas de processamento permite obter desempenho aliado com o consumo eficiente de energia, porém para obtê-los, é necessário explorar de maneira eficiente os recursos de hardware de cada tecnologia presente na plataforma. Um meio de realizá-lo é por meio do mapeamento da aplicação de modo a identificar os seus possíveis gargalos de computação, e dessa maneira, extrair características que permitam traçar o perfil de uma ou mais partes da aplicação que se adeque melhor a um determinado tipo de elemento de processamento.

O mapeamento das aplicações podem ser feito de duas maneiras: manual ou automática. De forma manual, o usuário/programador é responsável em inserir o conhecimento sobre a aplicação ou parte dela. Técnicas de *profiling*, por exemplo, podem auxiliar na identificação de possíveis gargalos da aplicação. Por outro lado, o mapeamento automático, o usuário/programador apenas insere os dados/códigos que se deseja classificar e o sistema retorna os mesmos já classificados, sem a necessidade de que usuário tenha de inserir o conhecimento sobre os dados.

Dessa maneira, o trabalho desenvolvido para esta dissertação consiste em mapear *kernels* ou trechos presentes em um código fonte de maneira a classificar automática e definir qual o dispositivo de processamento é mais adequado para execução de cada *kernel*, classificando-os em GPP ou FPGA. Para auxiliar no mapeamento dos trechos de código, foi aplicado uma ferramenta de mineração de dados, denominada DAMICORE (*DAta MIning of COde REpositories*), na qual permite encontrar a relação entre os códigos, baseando-se em uma métrica que mede a similaridade por meio da compactação de arquivos.

1.2 Objetivo

O intuito deste projeto é desenvolver um classificador de *kernel* para mapeá-los de modo automático em arquiteturas híbridas, especificamente para GPP e FPGA. Este classificador

determina o dispositivo mais adequado para cada *kernel*, baseando-se somente no código sem haver a necessidade de execução prévia dos mesmos. Isso reduz significativamente o tempo de configuração de uma máquina híbrida, principalmente no caso de FPGA onde o tempo de síntese para a geração do hardware é diversas vezes mais demorado do que a compilação do software para GPP.

1.2.1 Objetivos Específicos

- Aplicar uma técnica de agrupamento de dados para auxiliar o classificador a determinar a plataforma de processamento mais adequada.
- Adotar *benchmarks* para FPGA e GPP nos quais serão inseridos na técnica de agrupamento de dados e servirão como base de conhecimento para o classificador.
- Prover uma técnica que auxilie na divisão de processos em hardware e software em uma plataforma híbrida de processamento.

1.3 Justificativa

O mapeamento de uma aplicação é fundamental para viabilizar o uso de sistemas computacionais híbridos, principalmente para FPGA. A decisão de mapeamento pode ocorrer somente a partir da análise do código fonte ou somente do *profiling* da execução deste código ou ainda a combinação dessas informações. No entanto, nem sempre é uma alternativa adequada para quando há necessidade de geração do hardware, pois o tempo de síntese pode demorar muito e assim comprometer a vantagem de um hardware dedicado.

Além disso, o mapeamento automático de uma aplicação auxiliará o usuário na tomada de decisão inicial de implementar um *kernel* em software ou em hardware.

1.4 Organização

Este trabalho está organizado em cinco capítulos. Este capítulo introduziu a área de pesquisa, apresentando as lacunas na literatura que motivaram a realização deste trabalho. No Capítulo 2, apresenta revisão bibliográfica. O Capítulo 3 contém o método de classificação. O Capítulo 4 contém os resultados e análises da técnica. O Capítulo 5 apresenta a conclusão e os trabalhos futuros.

CAPÍTULO

2

REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta uma revisão bibliográfica sobre as técnicas de clusterização e ferramentas para síntese em FPGA e para compilação de programas para GPP *multi-core*. Também são apresentados alguns trabalhos relacionados sobre técnicas aplicadas em ambientes híbridos de processamento para explorar os seus recursos computacionais.

2.1 Algoritmos de Clusterização

Técnicas de clusterização ou agrupamento vêm se tornando importantes nos dias atuais, pois permitem classificar ou agrupar objetos em grupos pelas similaridades entre os objetos (BERKHIN, 2002). De acordo com (JAIN; DUBES, 1988), a clusterização é um método que utiliza o aprendizado não supervisionado no qual pertence ao contexto de aprendizado de máquina, como ilustra a Figura 1.

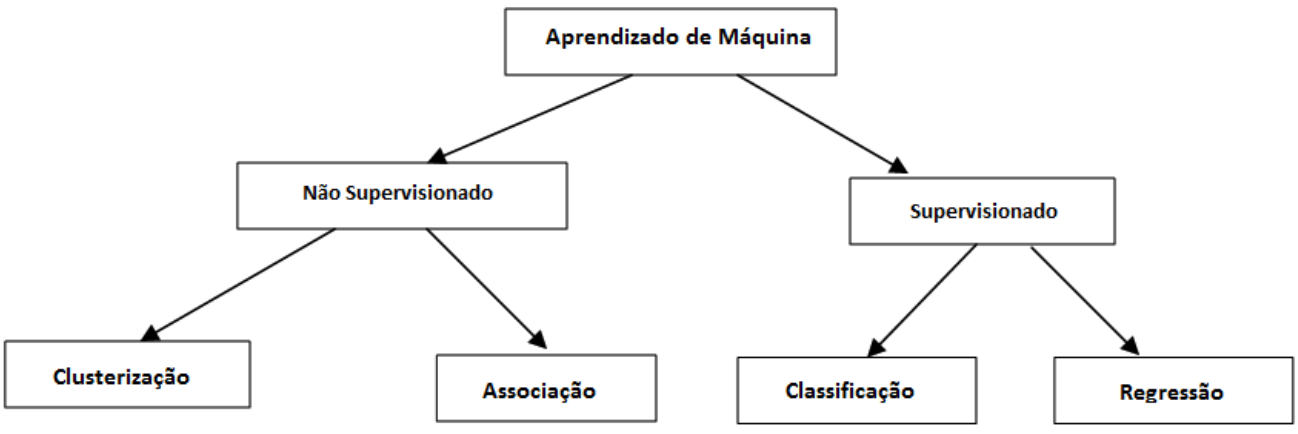

Figura 1 – Taxonomia de métodos de aprendizado de máquina adaptado de (KONONENKO; KUKAR, 2007)

Na prática, o uso desse tipo de técnica é empregado em aplicações de mineração de dados, como exploração de dados especifícos, recuperação de informações, mineração de textos, *marketing*, diagnósticos médicos, biologia computacional, entre outros (BERKHIN, 2002).

Na definição, segundo (KONONENKO; KUKAR, 2007), a taxonomia das técnicas de clusterização podem ser dividas em hierárquico e particionado.

- Clusterização hierárquica: constrói uma árvore de *clusters*, também conhecido por dendrograma. Todos os nós presentes no agrupamento contem nós “filhos” e nós “irmãos” particionados por um mesmo ponto coberto: um nó "pai"em comum. Essa estrutura se assemelha à uma árvore binária no qual permite explorar dados de diferentes níveis de granularidade. Os métodos hierárquicos podem ser categorizados em aglomerativo e divisivo.
  - Aglomerativo (*bottom-up*): inicia com um único ponto (objeto) e recursivamente mescla dois ou mais *clusters* apropriados de maneira que um grupo seja formado;
  - Divisivo (*top-down*): inicia com todos os objetos em um mesmo *cluster* e recursivamente o divide em grupos menores. O processo de divisão continua até que um critério de parada seja satisfeito;
- Clusterização particionada: constrói uma partição de um conjunto de dados contendo n objetos dentro de um conjunto de k *clusters*. Os métodos particionados realocam as instâncias, movendo-os de um *cluster* para outro, isto por meio de critérios definidos pelo usuário. Normalmente, a definição do número de *clusters* também é importante visto que a formação do agrupamento é moldada na quantidade de grupos definidos.

Os algoritmos de clusterização, como foi visto, realizam o procedimento de classificar os objetos pela similaridade em certos números de *clusters* (grupos, subconjuntos ou categorias), porém existem alguns procedimentos para realizar a classificação de um conjunto de dados. (XU; WUNSCH, 2005) descreve um processo de análises para algoritmos de clusterização que podem ser divido em quatro procedimentos básicos, como ilustra a Figura 2:

- Seleção e extração de características: a seleção de características distintivas de um conjunto de dados podem gerar recursos úteis para uma determinada aplicação. É um fator que define a eficácia do sistema, uma vez que as características sejam bem selecionadas. Isto pode diminuir a carga de trabalho e simplificar os processos subsequentes;
- Seleção do algoritmo de clusterização: esse passo é geralmente combinado com a seleção de uma medida de proximidade correspondente e a construção de uma função de critério. Os padrões são agrupados de acordo com as semelhanças dos objetos. Quase todos os algoritmos de clusterização são explicitamente ou implicitamente ligados a alguma

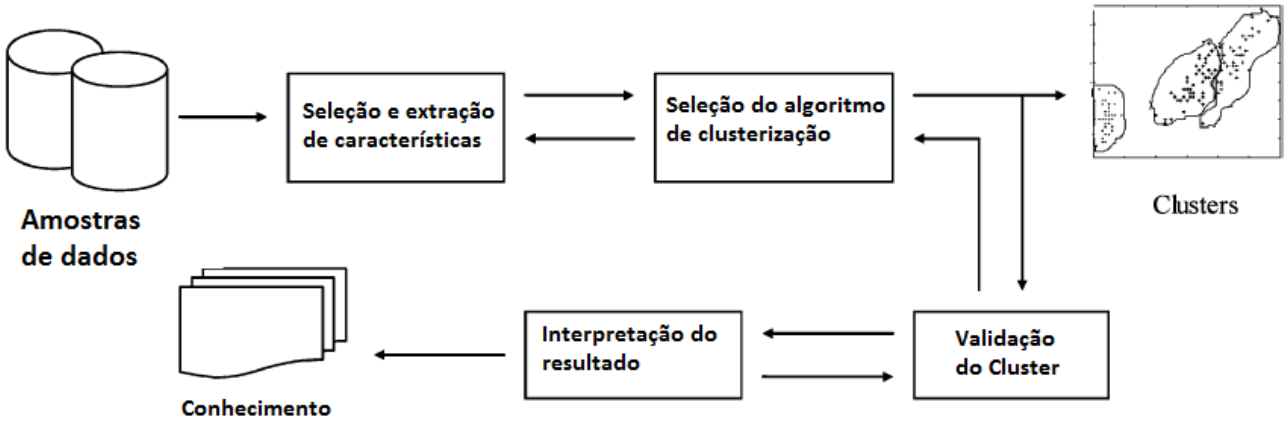

Figura 2 – Procedimento de clusterização. A análise de cluster típico consiste de quatro passos com um caminho de retorno. Estes passos estão intimamente relacionados uns aos outros e derivado de clusters (Adaptado de (XU; WUNSCH, 2005)).

definição da medida de proximidade, o que torna os algoritmos bem específicos à um domínio de aplicação.

- Validação: dado um conjunto de dados, cada algoritmo de clusterização pode gerar uma divisão sem importar se existe ou não uma estrutura. Além de diferentes abordagens que levam a diferentes *clusters*. E se tratando do mesmo algoritmo, a identificação de parâmetros ou a ordem de apresentação dos padrões de entrada podem afetar diretamente os resultados finais. Portanto, os critérios ou padrões adotados para a avaliação deve ser eficaz para fornecer o grau de confiança dos resultados obtidos do algoritmo utilizado.

- Resultados da interpretação: descreve o último objetivo no qual deve fornecer aos usuários soluções significativas a partir dos dados originais, para que possam efetivamente resolver os problemas encontrados. Além disso, análises e experimentos podem ser necessárias para garantir a confiabilidade do conhecimento extraído.

A seguir serão descritos dois exemplos de métodos de clusterização, o algoritmo de K-means e o DAMICORE, sendo este adotado para o desenvolvimento do trabalho

2.1.1 Algoritmo de K-Means

O algoritmo de K-means, também conhecido por K-médias, é uma clusterização particionada. K-means foi muito difundido devido a sua facilidade de implementação, pois permite agrupar um conjunto de dados em grupos seperados (BERKHIN, 2002).

O método de K-means opera sobre um conjunto de vetores dimensionais d, $d = \{x_i | i = 1, ..., N\}$, onde $x_i \in \Re^d$ representa os pontos de dados de i. O algoritmo inicia pela escolha de pontos k em d como representantes de k *clusters* iniciais ou centroides. A seleção das

sementes iniciais (representantes) é feita por meio de uma amostragem incluída a um conjunto de dados aleatórios, configurando eles como solução do cluster, um subconjunto do dado ou uma perturbação a média global no tempo de dados de k. Em seguida, o algoritmo itera entre dois passos até a convergência (WU *et al.*, 2007).

- Passo 1: Cada ponto de dado é atribuído ao centroide mais próximo. Isto resulta em um particionamento dos dados;
- Passo 2: Cada cluster representativo é realocado para o centro de todos os pontos de dados atribuídos. Se os pontos de dados vierem como uma medida de probabilidade, a realocação é feita por meio de uma média ponderada nos dados do particionamento.

O algoritmo converge quando as atribuições não sofrem mais alterações. Para exemplificar esse processo, têm-se a Figura 3. O exemplo ilustra um algoritmo no qual foi configurado o número de 4 *clusters* para serem agrupados. O algoritmo começa pela inicialização do conjunto de *clusters* e definição de quatro pontos iniciais (na imagem é representado pelo losango). Na iteração seguinte ocorre o passo 1, descrito anteriormente, no qual é responsável pela atribuição de cada ponto de dado ao centroide mais próximo. Nota-se que alguns dados, nessa iteração, já foram agrupados em *clusters*.

Em seguida, inicia-se o passo 2 no qual é calculado o centro de cada *cluster* formado. Nesse passo é possível notar o deslocamento dos centros de cada *cluster*. Após esta iteração, repete-se o Passo 1 e 2 novamente, chegando, assim, no ponto de convergência do algoritmo.

2.1.2 DAMICORE

DAta MIning of COde REpositories (DAMICORE) é uma ferramenta de mineração de dados baseada em uma métrica denominada *Normalized Compression Distance* (NCD) para encontrar a relação entre arquivos. O DAMICORE pode ser aplicado em códigos fontes de programas (funções/kernels) ou em representações simbólicas (características do programa). Isto permite realizar análises de grandes números de funções ou *kernels* para aplicações complexas a serem executadas em sistemas específicos (SANCHES; CARDOSO; DELBEM, 2011).

No processo de extração de características, O DAMICORE aplica a combinação de três técnicas para a elaboração de soluções: *Normalized Compression Distance* (NCD), *Neighbor Joining* (NJ) e *Fast Newman* (FN). Nas subseções a seguir serão abordadas as respectivas técnicas e o resultado final produzido com a combinação das mesmas.

2.1.2.1 Normalized Compression Distance

Normalized Compression Distance (NCD), em português Distância por Compressão Normalizada, é uma métrica que determina a semelhança entre arquivos (que nesta dissertação contém os códigos fontes dos *kernels* a serem classificados) com base no princípio da

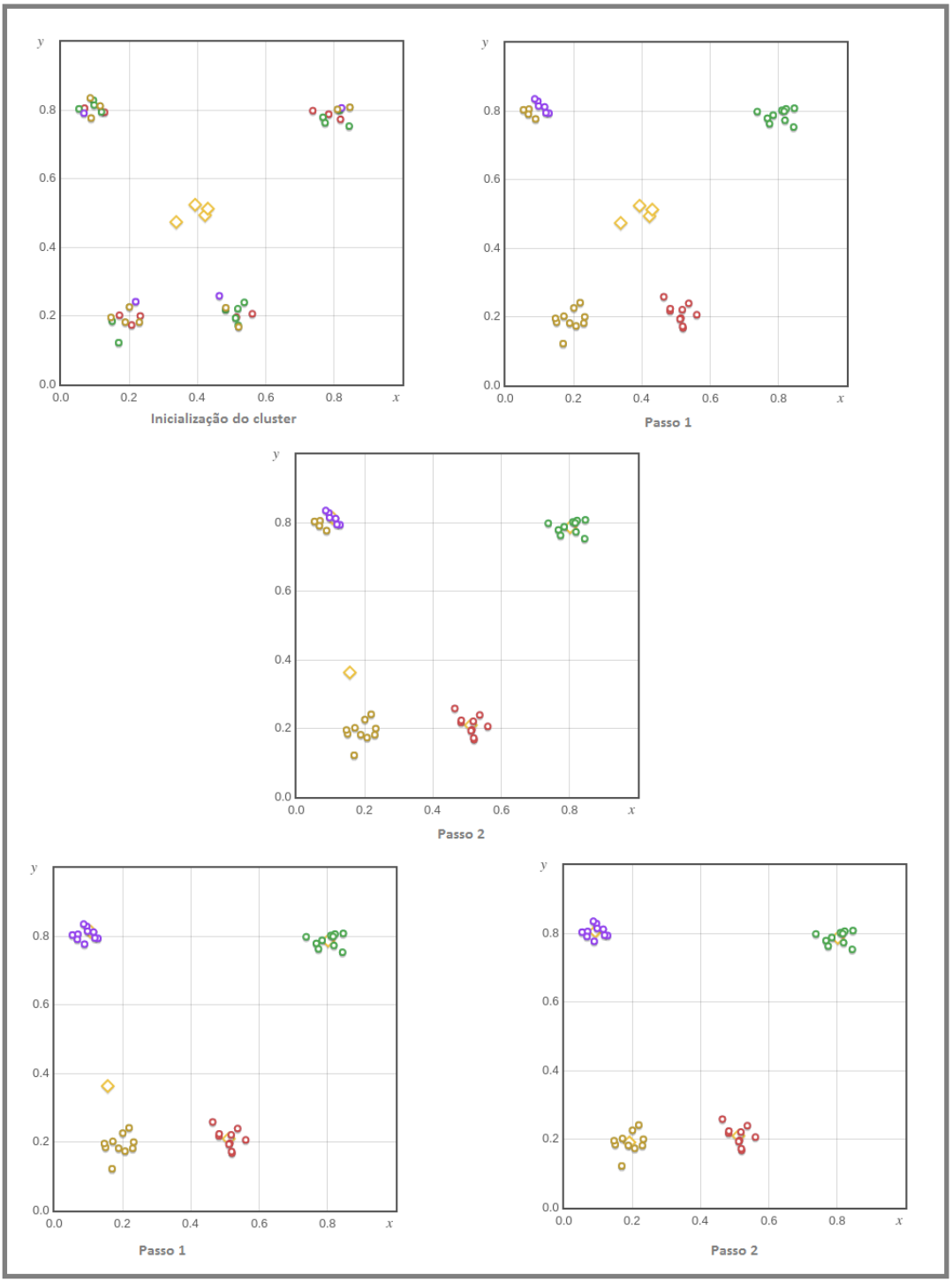

Figura 3 – Exemplo do algoritmo K-means (Hugo, 2015).

compactação de arquivos. A abordagem dessa técnica não requer conhecimento prévio sobre a quantidade de recursos utilizados para representação dos objetos analisados no domínio da aplicação (CILIBRASI; VITáNYI, 2005).

O NCD é baseado em uma métrica chamada NID (Normalized Information Distance) na qual é uma medida universal de distância que visa determinar a semelhança entre as variáveis ou objetos de acordo com a característica dominante compartilhada entre elas. O NID utiliza o conceito de complexidade de Kolmogorov para realizar o cálculo da distância, porém o NCD substitui o cálculo de Kolmogorov e aplica um algoritmo de compressão no qual o resultado é obtido por meio de uma aproximação. A equação do NCD é dada por:

$$D_{NCD}(X,Y) = \frac{C(XY) - min\{C(X), C(Y)\}}{max\{C(X), C(Y)\}}; \tag{2.1}$$

A equação representa o valor da distância entre X e Y. O valor corresponde a um número positivo e pode variar entre $[0; 1+\varepsilon]$, no qual representa a disparidade entre as variáveis X e Y e o parâmetro ε corresponde ao limitante superior para o erro do compressor utilizado. C (XY) representa o tamanho obtido da concatenação de X e Y seguida da compressão dos tamanhos de $C(X)$ e $C(Y)$ (CILIBRASI; VITáNYI, 2005).

No NCD pode-se aplicar diferentes compressores, tais como gzip, bzip, PPMZ, os mais comuns encontrados na literatura (ITO; ZEUGMANN; ZHU, 2010). Porém, o compressor adotado para este trabalho como padrão foi o zlib (GAILLY; ADLER, 2013).

A escolha do compressor pode impactar diretamente no cálculo de distância, pois dependendo da quantidade de dados e tipo, um compressor pode ser melhor que o outro.

No contexto do DAMICORE, o NCD funciona como um gerador de matrizes de distância para um conjunto de códigos, no qual permite verificar os níveis de semelhança entres os códigos e suas propriedades em comum (SANCHES; CARDOSO; DELBEM, 2011). Esta matriz de distância gerada servirá como entrada para a técnica *Neighbor Joining*, que será descrita a seguir.

2.1.2.2 Neighbor Joining

O Neighbor Joining (NJ) é um método de agrupamento para reconstrução de árvores filogenéticas, ou seja, cria uma estrutura em árvore que descreve o relacionamento entre objetos, isso corresponde a uma clusterização hierárquica . O princípio do NJ consiste em encontrar pares de unidade taxonômicas operacionais (OTUS ou vizinhos) (SAITOU; NEI, 1987). Estas unidades minimizam o comprimento total dos nós em cada estágio do agrupamento.

A construção da estrutura em árvore inicia com uma árvore-estrela (sem raiz, *unrooted tree*). Durante o processo, é realizado uma junção entre os nós da árvore-estrela, dentre os quais são substituídos por um nó adicional que representa um nó ancestral comum entre os nós. A cada junção, a estrutura da árvore perde dois nós, porém ganha um nó adicional no qual é

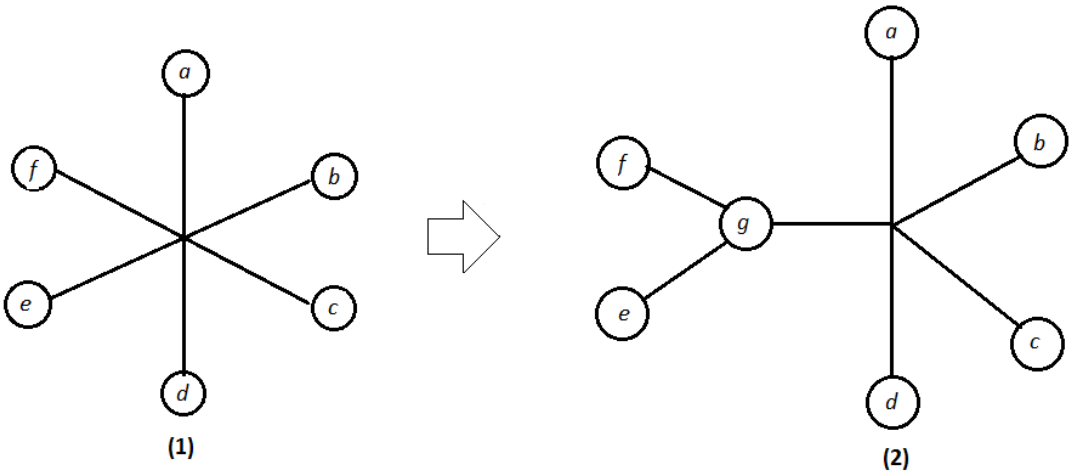

Figura 4 – Primeira iteração da técnica NJ: (1) Árvore-estrela e (2) Árvore após a inserção de um nó ancestral g

vinculado o comprimento do ramo que deve ser calculado. As iterações dessas junções são feitas sequencialmente de modo que ao final restem apenas dois nós na árvore (SOARES, 2014). A figura 4 ilustra como é feita a primeira iteração do método.

O processo de junções sucessivas possuem três etapas essenciais:

- Escolha dos dois nós na estrutura da árvore-estrela que sofrerá a junção. A escolha deve buscar na nova árvore criada um valor mínimo entre os nós. Este valor é calculado a partir de um somatório com base no comprimento entre os nós ramos da árvore. Dado os valores da matriz distância de entrada, a junção é calculada pela equação 2.2, onde D_{ij} representa o calculo da distância entre dois elementos do *cluster* no qual é utilizado a média da distância entre eles.

$$M_{ij} = D_{ij} - \frac{R_i + R_j}{n-2}; \tag{2.2}$$

$$\sum_{j,i\neq j} D_{ij}; \tag{2.3}$$

- Cálculo do comprimento dos ramos de dois nós escolhidos i e j para sofrer a junção, até a criação do nó ancestral g na árvore . As equações 2.4 e 2.5 representam este processo.

$$S_{iu} = \frac{D_{ij}}{2} + \frac{R_i + R_j}{2(n-2))}; \tag{2.4}$$

$$S_{ji} = \frac{D_{ij}}{2} + \frac{R_j + R_i}{2(n-2))}; \tag{2.5}$$

- Cálculo de uma nova matriz de distância, representado pela equação 2.6, uma vez que foi inserido um novo nó na árvore.

$$D'_{ku} = \frac{D_{ik} + D_{ij}}{2} - \frac{D_{ij}}{2}, \text{para todo nó } k \neq i \text{ e } j \tag{2.6}$$

No contexto do DAMICORE, o NJ recebe a matriz de distância gerada pelo método NCD e aplica as etapas descritas anteriormente para gerar a árvore de relacionamento da entrada de dados do DAMICORE. Porém, para extrair o potencial do *cluster* nesta topologia é necessário a aplicação da técnica *Fast Newman*

2.1.2.3 Fast Newman

O *Fast Newman* (FN) visa detectar e extrair a estrutura de comunidades de redes (NEWMAN, 2004). Neste caso, as redes representam as árvore filogenéticas geradas com o método obtidas do método NJ. O algoritmo FN lida com uma grande quantidade de dependências em árvores filogenéticas para encontrar as principais comunidades representativas de um sistema.

Dada uma rede, o algoritmo FN produz algumas divisões dos vértices da rede em comunidades. Esta divisão é definida por uma função Q, onde e_{ij} representa a comunidade i e j e descreve a fração de arestas que ligam aos vértices no grupo i àqueles no grupo j em relação a toda a rede. Define-se então, e_{ii} como o número de arestas dentro da comunidade i divido pelo total de arestas. Um algoritmo de detecção de comunidades deve maximizar a fração das arestas que os nós estão conectados dentro de uma mesma comunidade, ou seja, maximizar $\sum_i (e_{ij})$.

Porém, essa medida não permite avaliar de maneira adequada a qualidade das comunidades, pois o valor máximo é facilmente atingido quando todos os vértices pertencem a uma mesma rede. Para permitir uma melhor avaliação é inserido mais um componente $a_i \sum_j e_{ij}$ que corresponde a fração das arestas que conectam a pelo menos um vértice da comunidade i. Com isso, é definido o índice de modularidade Q que pondera as frações em questão. Ao final, têm-se a equação 2.7

$$Q = a_i \sum_j (e_{ij} - a_i^2); \tag{2.7}$$

O índice de modularidade permite obter todas as possíveis divisões da rede em comunidades. (NEWMAN, 2004) desenvolveu um método guloso com o intuito de identificar as comunidades em uma rede. Este método considera cada vértice da rede como uma comunidade e por meio de iterações repetidas as comunidades em pares considerando todas as possibilidades.

A técnica FN é relevante devido a dois aspectos: por permitir a escalabilidade de tratar a manipulação de muitas funções; e pelas operações sobre arranjos de dados em modelos de rede no qual permite o uso de árvores filogenéticas (SANCHES; CARDOSO; DELBEM, 2011).

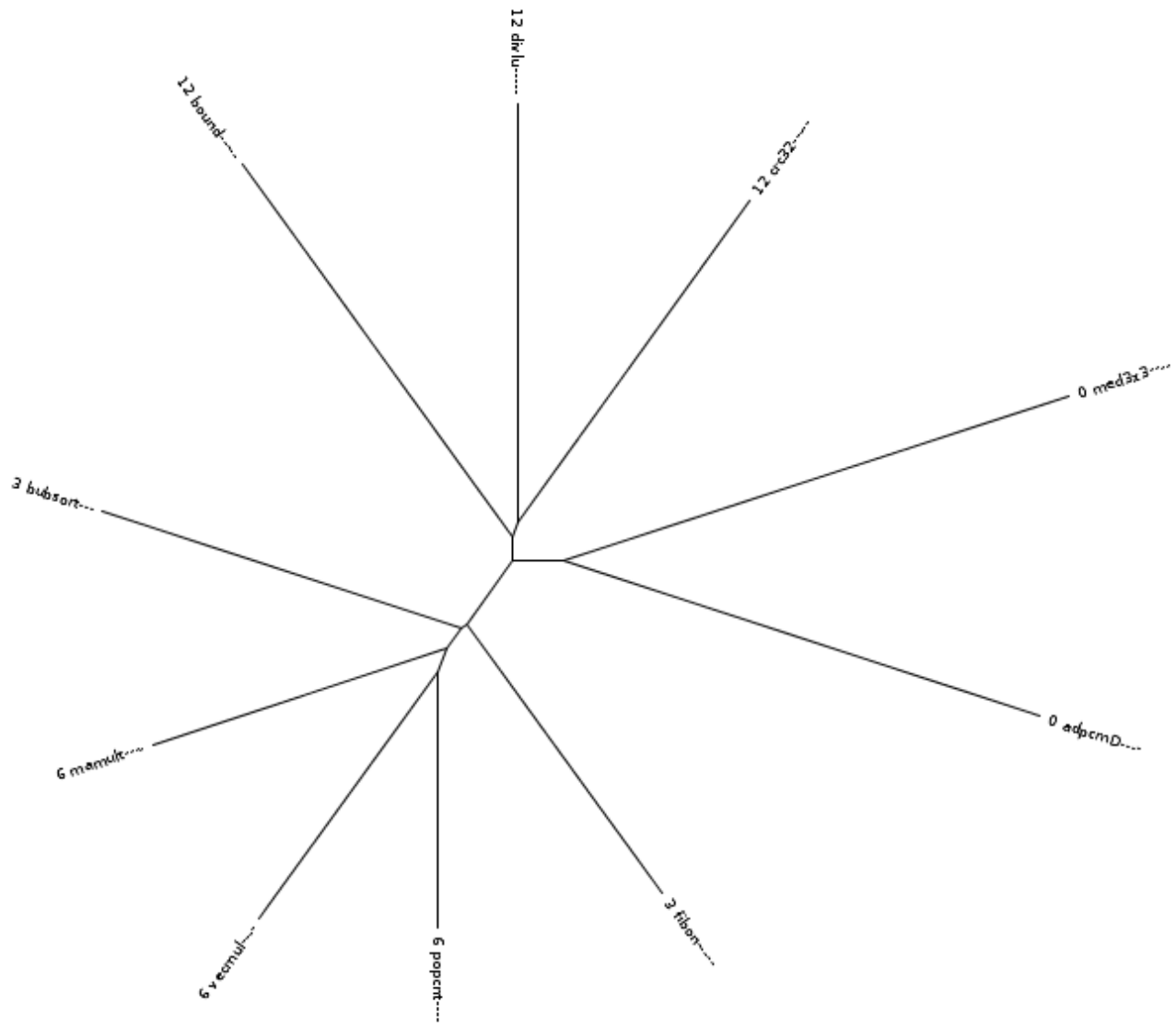

Figura 5 – Saída gerada pelo DAMICORE: aplicação das técnicas NCD, NJ e FN.

2.1.2.4 Saída do DAMICORE

Como foi explicado, o DAMICORE utiliza a combinação de 3 técnicas: NCD, NJ e FN. O NCD gera a matriz de distância entre as entradas; o NJ cria a árvore de relacionamento, baseando-se na matriz de distância; o FN, a partir do resultado provido do NJ, determina a organização do agrupamento com base nas comunidades geradas.

Por meio dessas três técnicas, o DAMICORE produz uma estrutura em árvore, conforme o exemplo da Figura 5. O NJ pode ser notado pela estrutura da árvore criada e o FN pelas comunidades formadas que é representado pelo npumero que inicia a identificação dos nós folhas. Por exemplo, têm-se as entradas com o nome med3x3 e adpcmD nos quais possuem o número ou identificador de comunidade 0.

Além de gerar árvore de modo gráfico, o DAMICORE gera um arquivo texto que

representa essa estrutura. O arquivo gerado, com base na Figura 5, é a seguinte:

```
1   (0 adpcmD-----:0.42822,((3 fibon------:0.28404,(((6 popcnt
      -----:0.21718,
2 6 vecmul-----:0.26066):0.02099,6 mamult-----:0.26401):0.02202,
3 3 bubsort---:0.32261):0.00605):0.06639,(12 bound------:0.39334,
4 (12 divlu------:0.35732,12 crc32------:0.33822):0.01383):0.02056)
      :0.04335,
5 0 med3x3-----:0.45413);
```

Inicialmente, o arquivo gerado pelo DAMICORE é composto por delimitadores: “(”, “,”, “)”; índices de identificação de comunidade; nomes das entradas de dados; informações de distância de uma entrada a outra ou a distância de uma comunidade ou grupo a outro e identificador de final de agrupamento “;”.

O delimitador “(” representa o inicio de um agrupamento ou uma comunidade, assim como o delimitador “)” representa o final para ambos os casos. O delimitador ","corresponde o incio de um nova comunidade ou a distância entre um item ao outro dentro na mesma comunidade.

As informações de índice de comunidade e nomes das entradas de dados aparecem sempre nessa ordem, respectivamente. Por exemplo, 0 *adpcmD*. O valor de distância pode ser identificado após o caractere entre os caracteres ":"e ",".

2.2 Ferramentas de síntese para FPGA e de compilação para processadores *multi-core*

Esta seção apresenta as ferramentas ImpulseC e LegUP para a plataforma FPGA nos quais permitem converter algoritmos descritos na linguagem C para RTL (Register Transfer Level), que representa o hardware do algoritmo num dado FPGA. E a ferramenta OpenMP para compilação em plataformas *multi-cores* (GPP). Os resultados obtidos do ImpulseC, LegUP e OpenMP são necessários para avaliar a eficácia do classificador de *kernels*

2.2.1 Impulse C

O Impulse C foi desenvolvido pela empresa *Impulse Accelerated Technologies* e permite desenvolver aplicações para software e hardware (FPGA), a partir de um subconjunto da linguagem C combinado com uma biblioteca de funções compatíveis a C com suporte a programação paralela (PELLERIN; THIBAULT, 2005). As bibliotecas contidas no Impulse C suportam FPGAs da família Altera com os processadores soft-core Nios e NiosII, bem como para a família da Xillinx, com os processadores Microblaze e PowerPC.

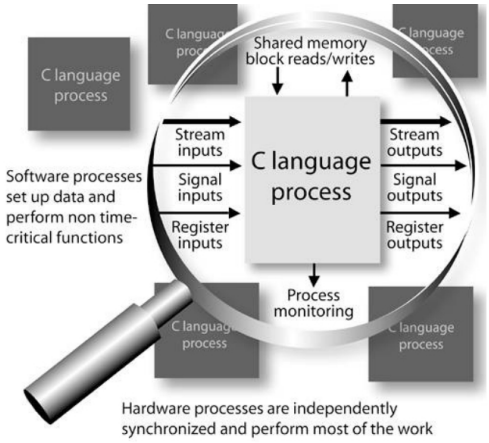

Figura 6 – Estrutura de programação no Impulse C (PELLERIN; THIBAULT, 2005).

A ferramenta Impulse C teve sua origem no laboratório de *Los Alamos National Laboratories* sob a direção de *Dr. Maya Gokhale*. Esta pesquisa, resultou na disponibilização do compilador *Stream-C* (GOKHALE *et al.*, 2000). As aplicações desenvolvidas em Stream-C geralmente atacam problemas voltados para criptografia de dados, processamento de imagem, astrofísica, entre outros.(PELLERIN; THIBAULT, 2005).

O intuito do Impulse C é permitir que a linguagem C seja utilizada para descrever uma ou mais unidades de processamento (chamados de processos) de aplicações paralelas que podem ser mapeadas em FPGA ou ser distribuídas entre recursos de hardware e software, incluindo micro-processadores embarcados ou DSPs.

A abordagem do Impulse C centra-se em mapear algoritmos para sistemas mistos composto por FPGA e GPP com o objetivo de criar implementações de processos em hardware que comuniquem (por meio de *streams*, sinais e/ou memórias) com processos implementados para software (PELLERIN; THIBAULT, 2005).

O modelo de programação do Impulse C é baseado em um modelo chamado de processo de comunicação sequencial (*Communicating Sequential Process* (CSP) (PELLERIN; THIBAULT, 2005). Esse modelo foi descrito por Charles Antony Richard Hoare em 1978. Segundo Hoare, é um modelo de programação que descreve padrões de interações entre componentes, cada componente é um processo que executa de modo independente (HOARE, 1978). Cada processo pode representar um programa em software (operações sequenciais) ou um programa em hardware, porém os processos são limitados a se comunicar um com os outros por meio de canais de *streams* e sinais, ou por memória compartilhada.

Com base na Figura 6, o Impulse C funciona basicamente como produtor e consumidor. Processos em software geralmente são responsáveis por gerar os dados da aplicação e transmiti-los por meio do *streams* para um outro processo. O resultado desse processo é consumido por processo em hardware no qual executa a computação da aplicação e a saída gerada por ser transmitida para outro processo em hardware ou em software consumirem.

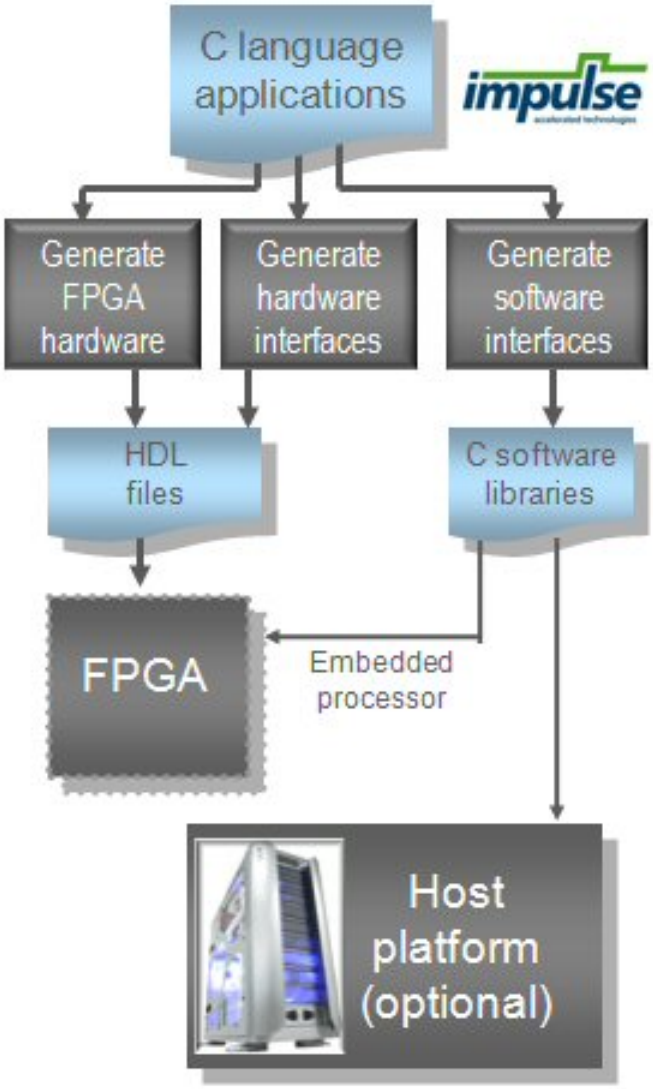

Figura 7 – Geração de código a partir de uma aplicação em linguagem C (IMPULSEC, 2010).

A geração de códigos no Impulse C é divido em três partes, conforme ilustrado pela figura 7:

- Geração de hardware para FPGA: converte os processos de hardware para linguagens de descrição de hardware Verilog ou VHDL;

- Geração de interfaces de hardware: cria as interfaces específicas de hardware para a plataforma FPGA, por exemplo FPGAs Xillinx ou Altera;

- Geração de interfaces de software: cria as interfaces de software que foi especificadas no código C. Também é gerado as interfaces para os processadores soft-core embarcados em FPGA (NiosII, por exemplo);

A ferramenta Impulse C também provê opções de otimização aplicadas na geração de códigos para o hardware. As otimizações podem ser inseridas na estrutura do código fonte em C ou configuradas na plataforma de programação da ferramenta antes de realizar a compilação. Dentre as opções de otimizações podemos citar:

- *Constant Propagation*: propaga os valores de contantes no hardware;
- *Scalarize array variables*: o compilador tenta substituir os vetores locais por variáveis de maneira que elas sejam implementadas com registadores, ao invés de memórias;
- *Unrolling*: o compilador desenrola o escopo do *loop* por inteiro, movendo certas expressões para fora da estrutura de repetição. Esta primitiva pode ser acionada por meio da inserção da instrução #pragma *CO UNROLL* na estrutura de repetição.
- *Pipelining*: permite que múltiplas iterações na estrutura de *loop* sejam executado em paralelo. Para habilitar a primitiva de *pipeling* na ferramenta é necessário inserir a instrução #pragma *CO PIPELINE* na estrutura de repetição que se deseja aplicá-lo.

Alguns cuidados devem ser tomados para a geração de aplicações na ferramenta Impulse C, pois alguns tipos de instruções descritas em linguagem C e de construções não são suportadas. Dentre elas podemos citar:

- Recursão: um processo de hardware não pode chamar ele mesmo.
- Chamada de funções, exceto as funções pré-definidas pelo Impulse C.
- Ponteiros: devem ser resolvidos em tempo de compilação.
- Apenas são permitidos o uso de inteiros e *arrays*.
- Sem suporte a variáveis globais, deve-se utilizar o canal de *streams*, de sinais ou memórias compartilhadas para realizar a comunicação entre processos.

2.2.2 LegUP

O *LegUP* é uma ferramenta desenvolvida por pesquisadores da Universidade de Toronto e permite que programadores possam utilizar a linguagem C para criar aplicações em FPGA (CANIS *et al.*, 2013a). A ferramenta *LegUP* se encontra na versão 4.0. Nas versões anteriores, oferecia suporte apenas para FPGAs Cyclone II e Stratix IV da Altera. Na atual versão, inclui FPGA Virtex 6 da família Xillinx e Stratix V da Altera.

O processo de desenvolvimento de um aplicação no *LegUP* é dividido em seis etapas (CANIS *et al.*, 2013b), conforme ilustrado pela Figura 8.

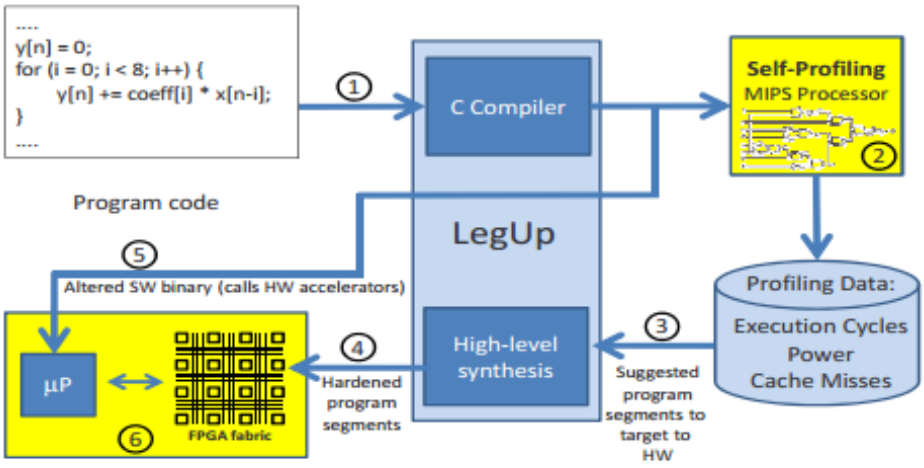

Figura 8 – Fluxo de projeto com LegUP (CANIS *et al.*, 2013a).

1. O projetista implementa a aplicação em software utilizando a linguagem C.
2. Executa a aplicação;
3. A partir disso, é criado um hardware *profiler* que identifica as seções críticas do código. Na ferramenta, o *profiling* é voltado para processadores MIPS (*Microprocessor without interlocked pipeline stages*)
4. Utilizando as informações de *profiling*, o projetista pode inserir marcações (primitivas de otimização) nas funções do programa para ser sintetizados no acelerador de hardware (FPGA).
5. O software original é recompilado com as funções aceleradas e modificadas com o código para iniciar o acelerador de hardware correspondente e passar alguns parâmetros das funções necessários para o hardware;
6. Execução do aplicação em GPP/FPGA.

Utilizando o *LegUP*, um programa pode ser particionado em uma porção em hardware e outra em software. A escolha do particionamento depende do objetivo do projetista. O *Legup*, além de prover um mecanismo de profiling, pode também estimar o *speedup* associado com a migração de um determinado segmento. Isso permite obter a informação se é melhor opção manter o programa em hardware ou em software.

A ferramenta *LegUP* possui como base o compilador do *Low-Level Virtual Machine* (LLVM). A aplicação em C é traduzida para representação intermediária do LLVM, no qual é analisada e modificada a partir de uma série de passos de otimizações do compilador. As otimizações que podem ser empregadas no *LegUP* são as técnicas de *pipeline* e *Unroll*, otimizações para eliminar a presença de códigos mortos na aplicação, propagação de constantes e *Multi-Pumping*.

O *Multi-Pumping* permite o compartilhamento de recurso entre as operações, mapeando múltiplas operações para uma única unidade funcional. Esta primitiva é particularmente efetiva para blocos DSPs da FPGA, em específico para as operações de multiplicação (CANIS; ANDERSON; BROWN, 2013).

Diferentemente do Impulse C onde as otimizações de *loop pipeline* e *unroll* são inseridas na estrutura do código, no *LegUP* é necessário inserir marcações (*labels*) na estrutura de repetição desejada e descrevê-las no arquivo de configuração do *LegUP* junto com a técnica que se deseja aplicar. Abaixo segue um exemplo, no qual é aplicado a técnica de *pipeline* a estrutura de repetição.

```
1        // marcações na estrutura de repetição
2    loop: for (i = 0; i < N; i++)
3
4
5      // habilitar loop pipeline
6    loop_pipeline "loop"
```

2.2.3 OpenMP

O *OpenMP* foi definido em 1990 pelo grupo *OpenMP Architecture Review Board (ARB)*. Porém só no final de 1997, um grupo formado por empresas tais como SUN Microsystems, IBM, Intel, dentre outros se uniram para criar um padrão para programação paralela para arquiteturas de memória compartilhada.

OpenMP é uma API (*Application Programming Interface*) de memória compartilhada, cujas características são baseadas em programação paralela. OpenMP é adequado para aplicações para arquiteturas *Symmetric Multi-Processing* (SMP), ou seja, processodores *multi-cores* (CHAPMAN *et al.*, 2008).

Esta API não é uma linguagem de programação, ela define notações que são inseridas em um programa sequencial que podem ser descritos em Fortran, C ou C++. Estas notações irão definir como a aplicação irá ser compartilhada entre as *threads* que executarão em diferentes processadores (*cores*). As inserções de notações em programas sequenciais trouxeram benefícios a arquitetura SMP dado que com um mínimo de modificações no código é possível explorar um nível considerável de paralelismo na aplicação.

O modelo de programação do *OpenMP* é baseado em *Multithreads*. Uma aplicação descrita no OpenMP começa executando com apenas uma *thread* até identificar o inicio de uma região paralela (identificação das diretivas da ferramenta). A partir desse ponto, as instruções são distribuídas para um conjunto de *threads* para realizar a computação de forma concorrente. Ao término, as ações de cada *thread* são sincronizadas e o fluxo de execução do código segue

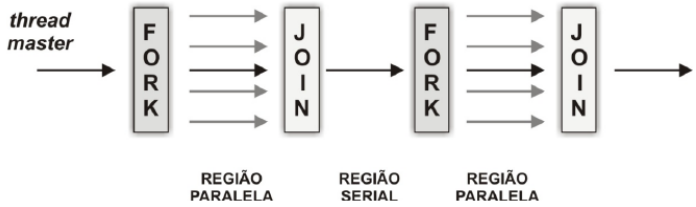

Figura 9 – Modelo de programação OpenMP

com apenas uma *thread* até identificar uma outra região paralela (CHAPMAN *et al.*, 2008). Esse modelo é conhecido como *fork-join*, conforme a Figura 9

As diretivas do *OpenMP* são geralmente aplicadas em estruturas de repetição, pois são os trechos com maiores *overheads* de uma aplicação sequencial e onde o paralelismo de instruções podem ser mais eficazes (CHAPMAN *et al.*, 2008). Dessa maneira, é possível repartir a mesma porção de carga de trabalho entre *threads* de processamento. Se considerarmos uma grande quantidade dados e iterações somados a quantidade de *threads* (núcleos de processamento), o ganho em tempo de execução de uma aplicação pode ser significativo.

2.3 Trabalhos Relacionados

Existem diversos trabalhos relacionados que utilizam plataformas heterogêneas de processamento. O uso de GPU está consolidado tanto para o meio científico quanto comercial. Isto se deve muito ao fato de que a programação para esses dispositivos são feitas em alto nível, como CUDA e OpenCL. Por outro lado os FPGAs, dispositivos que também apresentam vantagens na computação de alto desempenho, ainda necessitam de conhecimento avançado em hardware para utilizá-lo e explorá-lo de modo eficiente. Atualmente, os FPGAs estão sendo adotados em arquiteturas híbridas: GPP-FPGA, GPP-GPU-FPGA, GPP-FPGA (LIU; LUK, 2011; AUERBACH *et al.*, 2012; NESHATPOUR; MALIK; HOMAYOUN, 2015).

O uso desses tipos de arquiteturas, compostas por *n* tipos de processadores ou aceleradores de hardware, tem um porém. A questão é como explorar da melhor maneira possível os recursos providos por cada tecnologia de acordo com sua a aplicação.

Diversos trabalhos tentam explorar de modo eficiente os recursos de hardware visando obter desempenho e/ou uma melhor eficiência energética. (UKIDAVE; KAELI, 2013) demonstra um trabalho que aplica otimizações na estrutura do código fonte da aplicação, visando analisar o seu impacto no desempenho e no consumo de energia. O autor utilizou uma plataforma composta por GPP e GPU e aplicou as otimizações para em 3 casos de teste: aplicando *loop-unrolling* a estrutura de repetição da aplicação; modificações nas memórias locais de modo a reduzir a quantidade de acessos a memória global da GPU; e otimizações na representação dos dados, em específico para representação de números em ponto flutuante. Os resultados demonstraram que houve apenas melhoria no desempenho ao serem comparados com uma versão sem otimização,

por outro lado, o consumo de energia aumentou 27% devido a sobrecarga de instruções a cada núcleo de processamento da GPU.

Outra abordagem notada em trabalhos relacionados está na divisão e distribuição da carga de trabalho para os dispositivos de processamento na arquitetura. No trabalho de (MA *et al.*, 2012), o autor propõe um *framework* aplicando o método descrito em um dos seus cenários de teste de modo a criar um sistema que seja eficiente energeticamente. A arquitetura é composta por GPP e GPU e foi aplicada uma divisão e distribuição de carga de trabalho entre os dispositivos de modo que a quantidade de processos a serem executados em cada plataforma terminem quase ao mesmo tempo. Como resultado, a energia gasta pelos dispositivos que ficam ociosos e pela espera por dados são minimizados, porém isso afeta diretamente o desempenho da aplicação, tornando-o mais lento.

(LIU; LUK, 2011) também abordam aspectos de divisão e distribuição de carga trabalho para uma arquitetura heterogênea. O estudo realizado pelo autor utiliza uma arquitetura composta por GPP, GPU e FPGA e aplica 3 sistemas de métricas para os cenários de teste: *throughput*, eficiência energética e temperatura. Para todos os cenários são utilizados o *benchmark* Linpack [1] para realizar a avaliação e formulações matemáticas responsáveis por auxiliar a alocação de carga de trabalho para cada dispositivo. Como resultado, o sistema mostrou redução expressiva em relação ao consumo de energia, de aproximadamente 56,54%.

Utilizando ainda a mesma abordagem, (AUERBACH *et al.*, 2012) propõe um compilador, chamado *Liquid Metal* (Lime), no qual permite o uso de uma única linguagem de programação para sistemas computacionais heterogêneos, compostos por GPP, FPGA e GPU, para co-execução das operações. O intuito do autor é criar um ambiente que realize o particionamento dinâmico de um código fonte entre os elementos de processamento em tempo de execução, permitindo adaptação às mudanças nas cargas de trabalhos e disponibilidade de recursos dos dispositivos. A ferramenta desenvolvida pelo autor ainda não está funcionando por completo e está limitada a duas opções de ambiente de processamento: GPP + FPGA e GPP + GPU. Mesmo restrito, os resultados demonstraram que o compilador Lime obteve sucesso no processo de compilação empregando somente GPUs e somente FPGAs na co-execução de tarefas.

No trabalho de (NESHATPOUR; MALIK; HOMAYOUN, 2015) foi utilizado uma outra abordagem para o mapeamento da aplicação em dispositivos de processamento em GPP e FPGA, de modo que seja possível identificar os *overheads*. O autor desenvolveu um *framework*, denominado Hadoop, no qual é voltado para soluções de algoritmo *MapReduce*[2]. O Hadoop funciona basicamente como uma ferramenta de *profiling* no qual visa encontrar regiões críticas dentro de uma aplicação, ou seja, trechos onde ocorrem as incidências de *overheads*. A ferramenta utiliza uma análise compreensiva na comunicação e na computação de *overheads* para compreender como obter um melhor *speed-up* para cada aplicação ou *kernels* e dessa forma definir o que vai

[1] Linpack: *benchmark* que resolve um denso sistema de equações lineares.
[2] MapReduce: é um modelo de programação que lida com grandes quantidades de dados.

ser executado em GPP ou FPGA.

Conforme ilustrado anteriormente, os trabalhos são centrados em como realizar a divisão e distribuição de carga de trabalho em uma arquitetura híbrida. Na maioria dos casos a identificação do dispositivo ideal para um determinado trecho de código ouaplicação é feita manualmente pelo programador/usuário. O trabalho aqui desenvolvido visa justamente automatizar este processo de identificação, ou seja, o usuário apenas é responsável em inserir os dados a serem classificados.

2.4 Considerações Finais

Este capítulo apresentou uma revisão bibliográfica abordando os conceitos de algoritmos de agrupamento e alguns métodos aplicando os mesmos, descreveu duas ferramentas de síntese para FPGA e uma ferramenta de compilação para processadores *multi-core*, e, por fim, apresentou alguns trabalhos relacionados que visam explorar recursos computacionais de dispositivos de processamento em uma plataforma híbrida.

Na seção sobre técnicas de clusterização, foram abordados o algoritmo *K-Means* e a ferramenta DAMICORE. O K-means é uma técnica mais simples, pois em poucas etapas é possível construir um método que agrupe dados em *clusters*, baseando-se em algum critério. O DAMICORE por sua vez utiliza a junção de três técnicas (NCD, NJ,FN), isso o torna mais complexo e tem a capacidade de operar sobre grande quantidade de dados em um agrupamento, que facilita a análise entre grupos.

Na seção sobre as ferramentas de síntese para FPGA tais como *Impulse C* e *LegUP* foram são descritos seus respectivos modelos de programação e um conjunto de otimizações que podem ser empregados em uma aplicação. Essas otimizações são os fatores mais importantes para explorar as ferramentas de modo mais eficiente, obtendo assim melhores resultados para aplicações voltadas para FPGA. De outro lado, ilustrou-se a ferramenta de compilação para GPP no qual também é descrito seu modelo de programação e as diretivas de paralelização que são inseridas nas estruturas do código. No trabalho aqui desenvolvido, os resultados obtidos por meio dessas ferramentas são necessários para avaliar a eficácia o classificador de *kernels*.

Na seção de trabalhos relacionados, foi observado que existem maneiras de explorar os recursos computacionais em plataformas híbridas de processamento. Como vimos, a divisão de trabalho dos processos é feita por meio de mapeamento manual dos processos de modo a extrair características nas quais se adequem melhor a uma determinada plataforma, porém é o usuário realiza a tomada de decisão baseada nessas características. O trabalho aqui desenvolvido pode servir de base para a tomada de decisão de maneira automática.

CAPÍTULO

3

CLASSIFICADOR DE *KERNELS*

Neste capitulo será abordado todo o processo de desenvolvimento do trabalho, descrevendo aspectos de implementação do classificador, o uso das ferramentas descritas no capítulo anterior e o uso do DAMICORE nesse contexto.

3.1 O projeto

O intuito deste trabalho, como foi descrito anteriormente, consiste em classificar trechos de códigos (*kernels*) para plataformas híbridas de processamento, de modo que seja possível definir se um *kernel*, em específico, possui perfil mais adequado para FPGA ou GPP.

O projeto consiste em 6 etapas, ilustrado pela Figura 10.

1. Escolha e adequação de *benchmarks* aplicados as ferramentas *Impulse C*, *LegUP* e *OpenMP*: nessa etapa é extraído o resultado de execução de cada ferramenta para definir os *kernels* de referências na próxima etapa.

2. Definição de *kernels* de referência: a partir dos resultados providos de cada ferramenta, é realizado uma análise baseada no tempo de execução entre as combinações das plataformas: *Impulse C* com *OpenMP* e *LegUP* com *OpenMp*, para definir as referências de GPP e FPGA. As referências definidas desta etapa servirão de entrada para o DAMICORE.

3. Aplicação do DAMICORE: recebe como entrada os *kernels* de referência e a conjunto de *kernels* de entrada para gerar o agrupamento do DAMICORE que servirá como entrada para o método de interpretação.

4. Mapeamento dos *kernels*: é realizado a leitura dos dados gerados pelo DAMICORE de modo que todos os *kernels* e suas respectivas comunidades sejam identificados.

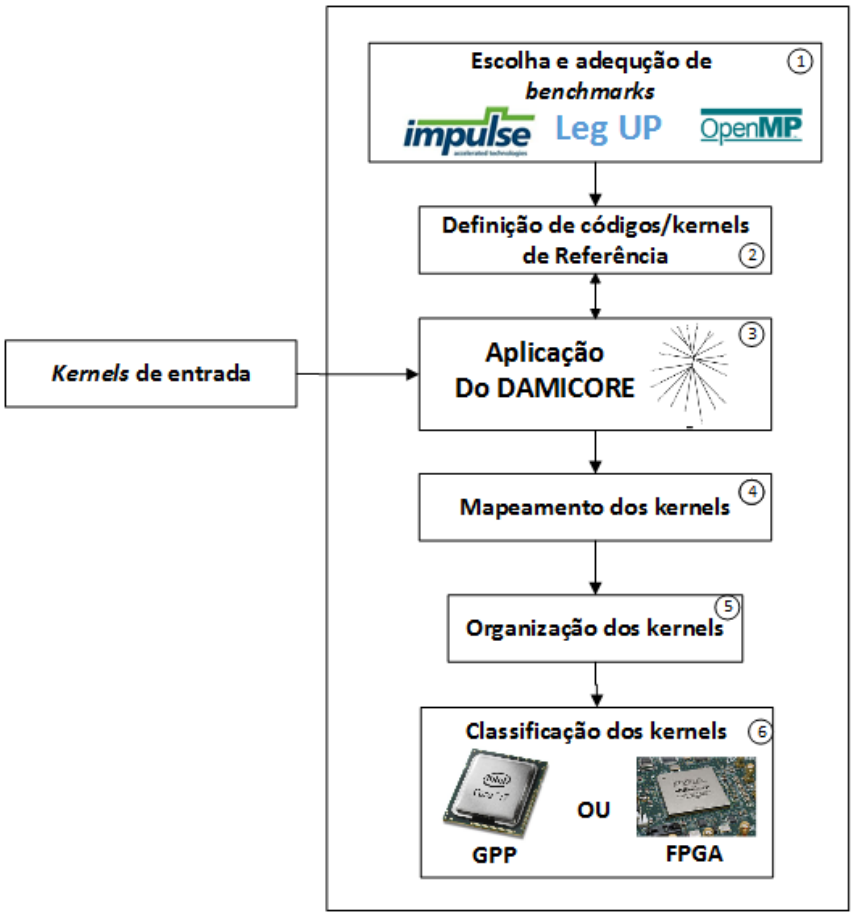

Figura 10 – Arquitetura do projeto.

5. Organização dos *kernels*: os *kernels* mapeados são organizados de maneira a prover a criação de estrutura que permita representar o agrupamento, ou seja, recriar uma estrutura semelhante a imagem gráfica gerada pelo DAMICORE.

6. Classificação dos *kernels*: nessa etapa é definido o perfil de cada *kernel* de entrada: GPP ou FPGA.

3.1.1 Escolha e adequação dos benchmarks

No trabalho foram escolhidos um conjunto de 30 *benchmarks* aplicados nas ferramentas *Impulse C*, *LegUP* e *OpenMP*, cujas implementações foram feitas em linguagem C e foram retiradas do repositório da empresa *Texas Instruments* [1]

Dentre os 30 *benchmarks* existem algoritmos de processamento de imagem, ordenação, algoritmos DSPs, entre outros. Conforme ilustrados pela tabela 1.

Durante a adaptação de cada *benchmark*, foram padronizadas as estruturas do código e as entradas de dados da aplicação a fim de maximizar a influência da lógica do algoritmo e

[1] http://www.ti.com/lsds/ti/processors/dsp/overview.page

Algoritmos
Adpcm Decoder
Adpcm Coder
Autocor
BubbleSort
Boundary
Change Brightness
Compositing
Conv3x3
Crc32
Divlu
Dotprod
FDCT
Fibonacci
Fibonacci_alt
GCD
InsertionSort
Matmul
Maximum
Median
Perimeter
Pix_sat
Popcount
Prime
Selection Sort
Sobel
SQRT
SSQRT
Vecsum
Vecmult

Tabela 1 – *Benchmarks* desenvolvidos.

minimizar a influência de como o algoritmo foi implementado, que depende das habilidades do programador.

Os resultados de cada ferramenta foram obtidos por meio da simulação do hardware no qual permitiu realizar uma estimativa do tempo de execução. Para a simulação utilizou-se a ferramenta *ModelSim* e junto com os dados de síntese foi possível medir o tempo com precisão. No caso do OpenMP foi utilizada uma diretiva de tempo do próprio compilador para realizar o calculo. Dessa maneira, gera-se o software para executar na GPP e com o *Impulse C* e o *LegUP* gera-se o hardware para a FPGA.

As aplicações que foram adaptadas o contexto do *Impulse C*, *LegUP* e *OpenMP* podem ter maneiras distintas de explorar as funcionalidades de cada ferramenta de modo a torná-las mais eficientes para, assim, obter melhores resultados. No entanto, para este projeto o intuito é

explorar a classificação a partir de implementações genéricas, porém com uma padronização da entrada do códigos e dados.

3.1.1.1 Impulse C

Nesta subseção será apresentado os resultados de síntese obtidos pelo Impulse C. As implementações foram sintetizadas para FPGA *Stratix* IV EP4SGX530KH40C2, da Altera.

No processo de implementação dos *benchmarks* foram utilizados apenas *streams* de dados. Em todas as aplicações têm-se o cenário de 2 processos em software e 1 em hardware. Os processos em software funcionam como um gerador de dados que são enviados por meio de *streams* para o processo em hardware e como receptor dos resultados providos do processo em hardware, também feito por meio de *streams*. Já o processo em hardware representa a implementação do *benchmark* em si.

Os resultados obtidos foram gerados para os seguintes cenários de teste: sem nenhuma otimização no código fonte em cada algoritmo; com uso das otimizações da própria ferramenta de *loop Unroll* e *loop Pipeline*, ambos com as técnicas *Scalarize array variables* e *Constant Propagation* habilitados.

O intuito destes cenários de teste é verificar como isso influência a geração de hardware. Dessa maneira, é possível analisar a eficiência da ferramenta em termos de tempo de execução e determinar para quais tipos de aplicação o *Impulse C* é mais eficiente. A tabela 2 ilustra os resultados obtidos de cada *benchmark* implementado.

A tabela 2 ilustra o tempo de execução, em milissegundos, de cada *benchmark* implementado. Nota-se que alguns tempos de execução não são informados devido as limitações da ferramenta. Essas limitações se encontram na maior proporção em algoritmos utilizando a técnica de *loop unroll* e com *arrays* de dados muito grande. Isto se deve porque a ferramenta, como citado no Capitulo 4, na seção que descreve a técnica de *unroll* aplicado pelo *Impulse C*, não possui parâmetros que indicam quantos *loops* serão desenrolados. Dessa maneira, os *loops* são desenrolados por completo e a quantidade de recursos de hardware replicados é alto, e dessa maneira, o compilador da ferramenta *Impulse C* não consegue gerar o hardware. Uma maneira de contornar essa situação é modificar o algoritmo de maneira que as estruturas de repetição sejam dividas em partes, limitando o número de iterações do *loop* mais interno (no qual é aplicado a otimização), porém para manter um padrão de desenvolvimento entre os códigos implementados em cada ferramenta e nos casos de testes, esta opção foi descartada.

Há casos em que os resultados estão zerados para os *benchmarks* implementados com a otimização de *Pipeline*, em específico os algoritmos: *Bubble Sort*, *Insertion Sort* e *Selection Sort*, pois a ferramenta de síntese não conseguiu gerar o hardware referente a eles.

Nota-se na tabela que em alguns casos os *benchmarks* sem otimização foram mais eficientes do que com otimizações. O motivo desse acontecimento se deve ao fato de que o

	Tempo (ms)		
Algoritmos	**Sem otimização**	**Unroll**	**Pipeline**
Adpcm Decoder	368	**-**	145
Adpcm Coder	479	-	160
Autocor	60	-	40
BubbleSort	14080	-	-
Boundary	338	439	87
Change Brightness	51	-	26
Compositing	91	-	31
Conv3x3	20	16	22
Crc32	106	71	58
Divlu	328	313	158
Dotprod	20	-	10
FDCT	35	28	27
Fibonacci_alt	12	-	10
Fibonacci	25	-	20
GCD	784	-	593
InsertionSort	8544	-	-
Matmul	35276	3350	3357
Maximum	236	-	97
Median	117	-	58
Perimeter	54	-	30
Pix_sat	58	-	34
Popcount	1094	166	1229
Prime	1410	-	14147
Selection Sort	69	-	-
Sobel	1259	-	809
SQRT	478	-	485
SSQRT	76	58	53
Vecsum	12	-	12
Vecmult	26	-	26
Viterbi	562	553	510

Tabela 2 – Resultados obtidos com o *Impulse C* em relação ao tempo de execução dos algoritmos em hardware com e sem otimização.

compilador da ferramenta não conseguiu sintetizar/paralelizar as otimizações de modo eficiente, gerando, assim, um hardware paralelo pode ser pior ou equiparado a uma implementação sequencial (sem otimização de paralelização). Em alguns casos, a frequência de *clock* do hardware paralelo gerado foi muito baixa, isto afetou fortemente o resultado.

Por meio da Figura 11 é possível ver melhor as disparidades entre as implementações. Pode-se observar que os algoritmos *Adpcm decoder*, *Boundary*,*Matrix Multiplication* e *Popcount*, foram os *benchmarks* que obtiveram uma melhora considerável no tempo de execução, comparando as versões com *Pipeline* e *Unroll* com a sem nenhuma otimização. Os tempos de execução dos *Benchmarks*, como *Vector multiplication*, *Vector Sum* e *Sqrt* são praticamente idênticos, ou seja, o compilador não conseguiu mapear as otimizações de modo eficiente durante a etapa de síntese do hardware. O pior caso ocorreu no algoritmo *Prime*, onde o resultado com a técnica de *pipeline* foi expressivamente bem pior que a versão sem otimização. Nos demais *Benchmarks*, a proporção de ganho das implementações com otimizações não são tão significantes. É importante ressaltar que os dados que não obtiveram resultados foram determinados os limites superiores no gráfico para representá-los.

3.1.1.2 LegUP

Nesta subseção será apresentado os resultados de síntese obtidos pelo LegUP. As implementações foram também sintetizadas para FPGA *Stratix* IV EP4SGX530KH40C2, da Altera. A adaptação dos *benchmarks* com o *LegUP* foi feita apenas com a combinação de otimizações *pipeline* e *unroll* aplicados nos *loops* mais internos do código fonte. O intuito não é realizar uma comparação entre as ferramentas, mas uma análise de comportamento das otimizações ao serem aplicadas no projeto desenvolvido. Os resultados providos dessa otimização está presente na tabela 3, a unidade de tempo é em milissegundos.

3.1.1.3 OpenMP

O processador utilizado para o contexto de GPP é um Intel i7@9200 de 2,67 GHz com 4 núcleos físicos de processamento e 4 núcleos emulados. Com base nas especificações do processador utilizado, os cenários de teste foram feitos para 1, 2 e 4 núcleos de processamento, sendo todos eles núcleos físicos. A opção de núcleos emulados não foi explorada.

No processo de implementação de cada código foram utilizadas diretivas de paralelização, em linguagem C, do *OpenMP*, aplicadas as estruturas de *loops*. Nessas estruturas são inseridas informações de quais variáveis do *loop* são privadas e a número de *cores* que será utilizado para o cenário de teste. Os dados de entrada de cada implementação foram declarados globalmente por meio de *arrays* no código fonte.

Por meio da tabela 4 é possível observar que os resultados de alguns *benchmarks* não são informados. Um dos motivos desse acontecimento ocorreu devido as dependências de dados na implementação dos algoritmos de modo que as operações não conseguiram ser paralelizadas.

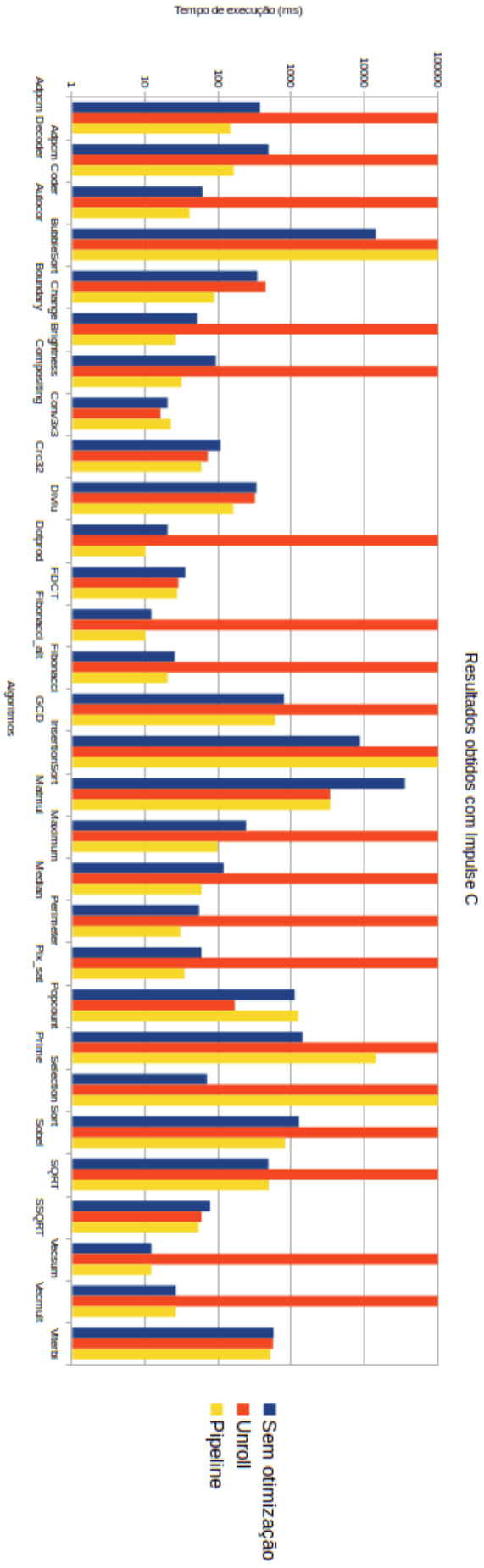

Figura 11 – Relação entre os resultados obtidos no *Impulse C*.

Algoritmos	Tempo de execução (ms)
Adpcm Decoder	60
Adpcm Coder	81
Autocor	8
BubbleSort	14815
Boundary	71
Change Brightness	4
Compositing	8
Conv3x3	2
Crc32	2
Divlu	259
Dotprod	8
FDCT	11
Fibonacci	16
Fibonacci alt	2
GCD	3187
Insertion Sort	11263
Matmul	2050
Maximum	3
Median	17
Perimeter	62
Pixsat	4
Popcount	117
Prime	978
Selection Sort	7327
Sobel	19
SQRT	1639
SSQRT	2
Vecsum	2
Vecmult	2
Viterbi	34

Tabela 3 – Resultados obtidos com o *LegUP*.

Em algumas implementações seriam necessárias modificações nas instruções do código de modo a permitir o uso das diretivas do *OpenMP*. Porém, tais modificações foram descartadas, pois alterando a estrutura do código fonte não haveria um padrão entre as implementações, e dessa maneira, a comparação dos resultados seria tendenciosa.

Existem algoritmos onde a versão sequencial (1 núcleo de processamento) foram melhores que as versões em paralelo (2 e 4 núcleos). Os resultados podem ter sido influenciados pelo: tempo de execução dos processos em cada *core*, comunicação e sincronização das operações entre os processos.

Nos casos em que os algoritmos possuem mais de um núcleo de processamento foram mais eficientes, pois o mapeamento das instruções realizado nos núcleos de processamento

Algoritmos	CPU (1 core)	CPU (2 cores)	CPU (4 cores)
Adpcm Decoder	327	-	-
Adpcm Coder	160	-	-
Autocor	7	4	5
BubbleSort	103	-	-
Boundary	1519	28045	28179
Change Brightness	13	13	21
Compositing	18	9	8
Conv3x3	6	9	9
Crc32	89	60	33
Divlu	420	599	634
Dotprod	7	10	5
FDCT	10	-	-
Fibonacci	8	6	5
Fibonacci alt	8	6	5
GCD	603	-	-
Insertion Sort	10	-	-
Matmul	1948	8713	16421
Maximum	189	431	959
Median	90	130	109
Perimeter	9	13	10
Pixsat	33	20	9
Popcount	1394	1109	1511
Prime	50	-	-
Selection Sort	30	-	-
Sobel	362	263	236
SQRT	57	29	20
SSQRT	112	-	-
Vecsum	12	8	11
Vecmult	2	3	3
Viterbi	440	-	-

Tabela 4 – Tempo de execução das implementações com o *OpenMP*.

permitiu uma melhor exploração do paralelismo dado que a quantidade de dados divididos entre os *cores* tornaram as instruções escaláveis dentro da região paralela.

A ilustração gráfica dos resultados das implementações de cada *benchmark* nos cenários de teste são demonstradas nas Figura 12. Conforme os casos citados anteriormente, é importante ressaltar alguns *benchmarks* implementados na ferramenta, como o *Bubble Sort*, onde a versão sequencial é melhor que a versão em paralelo. Algoritmos de ordenação como *Bubble Sort* possuem alta dependência de dados de uma iteração a outra, o que faz o resultado sequencial sobressair devido ao *overhead* da paralelização em processadores *multicore*. É importante observar que em implementações como *Vector Sum* e *Pix Sat*, a aplicação do paralelismo utilizando 2 *cores* foi melhor que 4 *cores*. Alguns fatores como a comunicação e sincronização das operações entre os *cores* influenciaram no resultado.

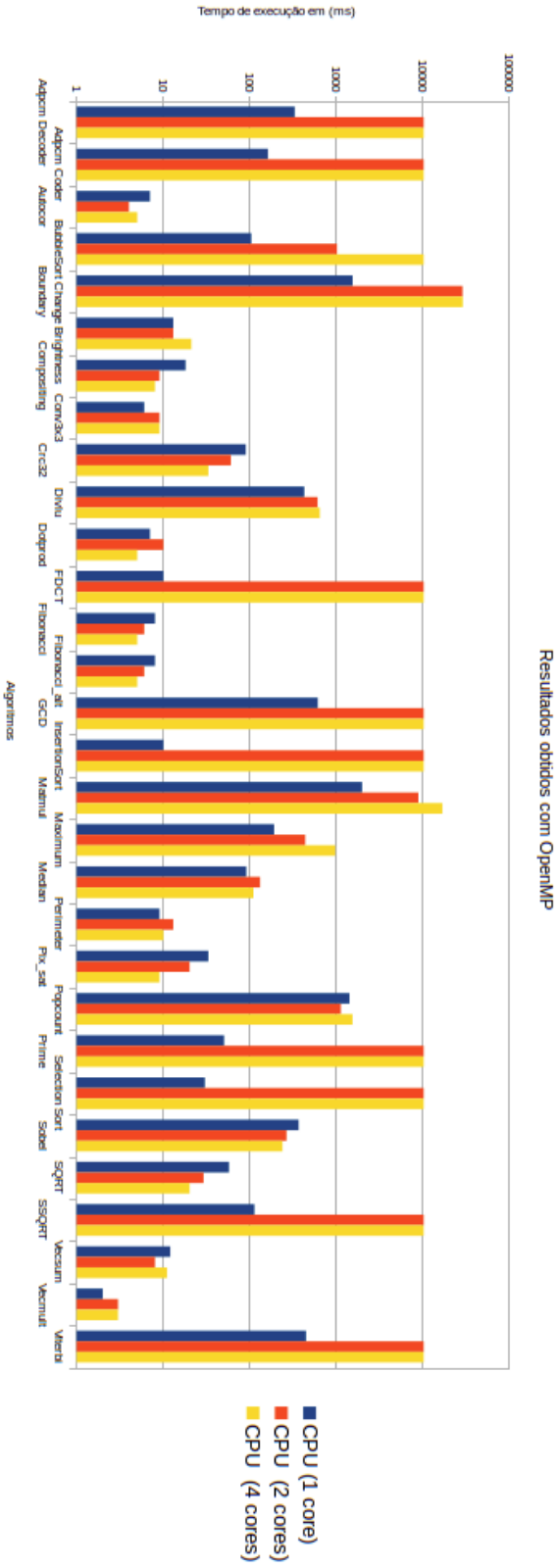

Figura 12 – Relação entre os tempos de execução dos algoritmos no *OpenMP*.

Um caso interessante a ser citado, refere-se ao algoritmo *Dotprod* onde há o seguinte cenário: a implementação para 1 *core* é melhor que 2 *cores*, porém é pior que a de 4. O resultado pode ter sido influenciado pela divisão de carga de trabalho entre os *cores*, acesso aos dados na memória e comunicação.

Os algoritmos *Prime*, *Selection Sort* e *Insertion Sort*, que não possuem valores para os cenários de teste para 2 e 4 *cores*, são as únicas implementações que não suportam a versão em paralelo.

3.1.2 Definição de kernels/códigos de referência

Após síntese ou compilação dos *algoritmo* em cada ferramenta descrita, inicia-se a etapa de seleção dos *kernels* de referência. Essa etapa é fundamental para etapa a classificação dos *kernels*.

A definição das referências é baseada na comparação do tempo de execução de cada *benchmark* nas ferramentas descritas e a distribuição dos 30 algoritmos aplicados no DAMICORE para definir o que é referência de hardware ou software. A justificativa de aplicar o DAMICORE nesta etapa é para garantir que as referências estejam bem distribuídas no agrupamento, não concentrando as referências em apenas um *cluster*.

As comparações são aplicadas para cenários entre hardware e software, ou seja, resultados providos do *Impulse C* versus *OpenMP* e do *LegUP* versus *OpenMP*. No total, são eleitas 10 referências de *benchmarks* para cada cenário descrito.

No primeiro cenário foram definidos 5 referências para FPGA e 5 para GPP . Para FPGA foram definidos os algoritmos *Adpcm Decoder*,*Boundary*, *Divlu*, *GCD* e *SSqrt*. Para GPP foram definidos, *Bubble Sort*, *Conv3x3*, *Dotprod*, *Matrix Multiplication* e *Vector Multiplication*. A escolha dessas referências foram definidas a partir do tempo de execução dos algoritmos citados e na distribuição do agrupamento gerado pelo DAMICORE, no qual é inserido todos *benchmarks* implementados. A Figura 13 ilustra a distribuição dos *kernels* de referência no *cluster* gerado, o que está em azul refere-se a FPGA e vermelho a GPP.

Diferentemente do Impulse C, não foi possível definir 5 *benchmarks* para FPGA e GPP, isto pela distribuição de todos os *benchmarks* no agrupamento gerado no DAMICORE a fim de evitar que várias referências de mesmo perfil ficassem no mesmo grupo, ilustrado pelo Figura 14 e pela comparação entre os resultados do *LegUP* e *OpenMP*. No total, têm-se 17 *benchmarks* com perfil de FPGA e 13 para GPP. Conforme a distribuição no agrupamento e nos resultados, foram definidos 3 referências para CPU: *Dotprod*, *Selection Sort* e GCD; e 7 para FPGA: ADPCM *Decoder*, *Boundary*, *Change Brightness*, Fibonacci alt, Sobel.

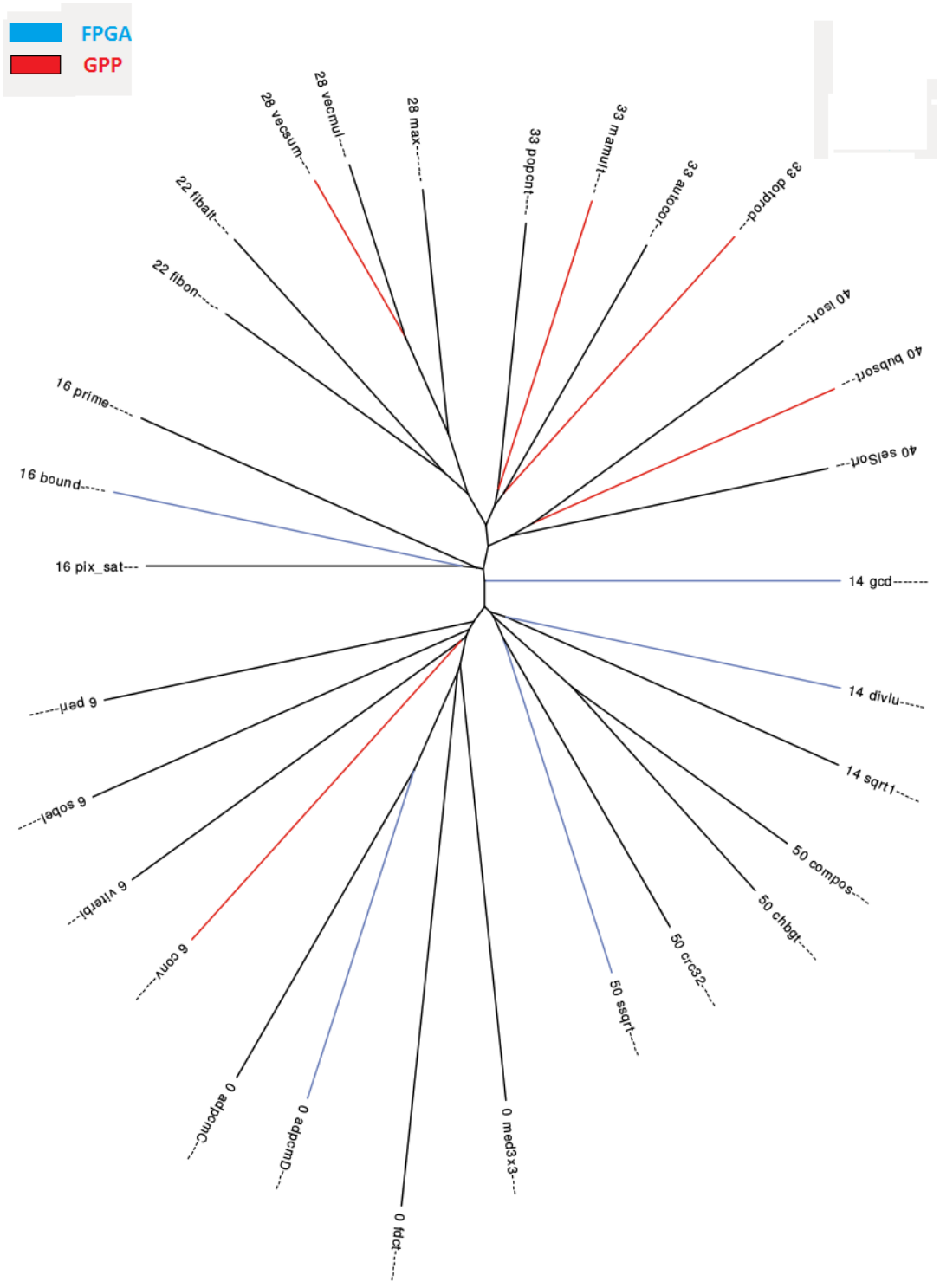

Figura 13 – Distruição dos *kernels* de referência no agrupamento do DAMICORE para as feramentas *Impulse C* e *OpenMP*.

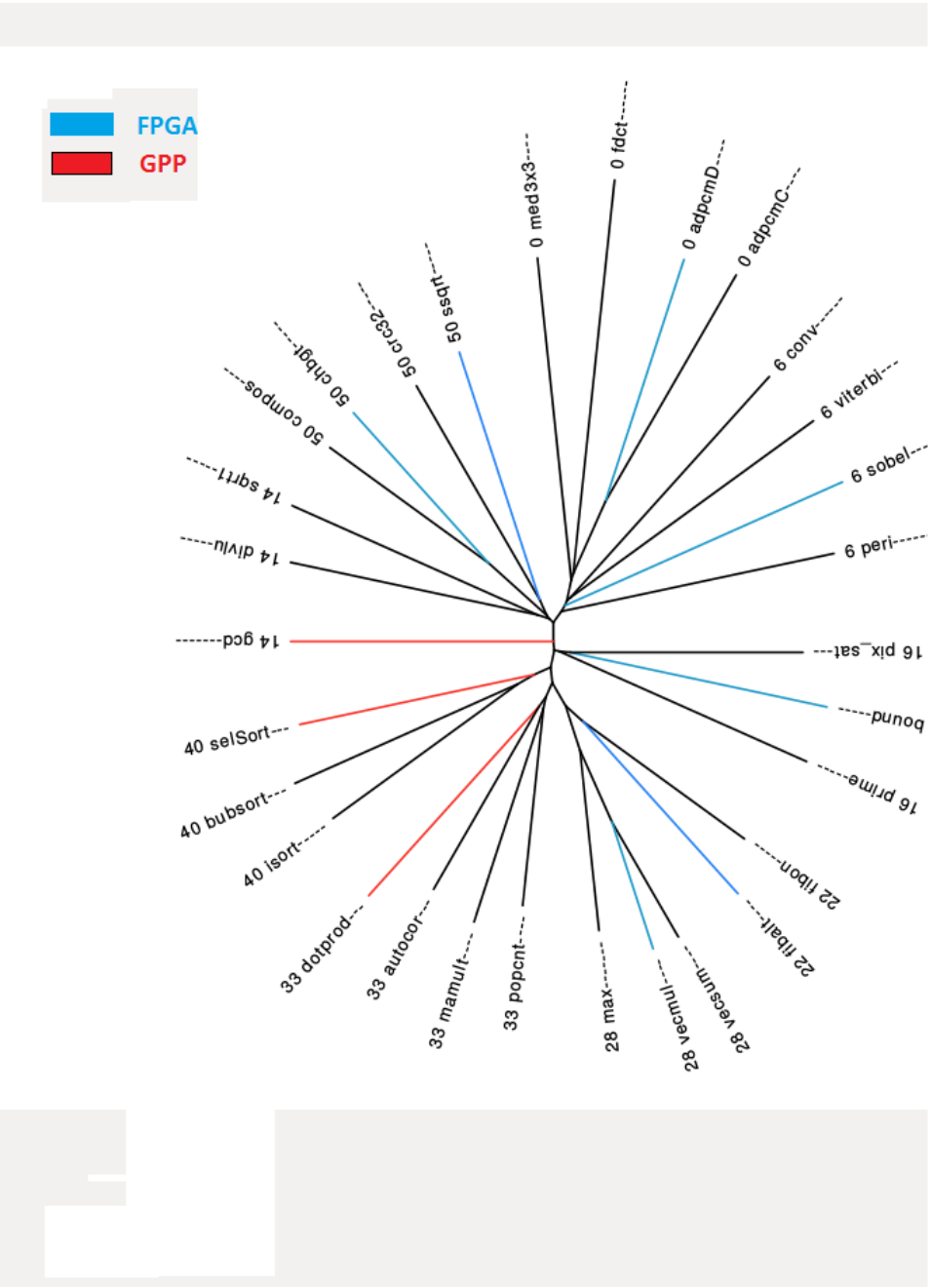

Figura 14 – Distruição dos *kernels* de referência no agrupamento do DAMICORE para as feramentas *LegUP* e *OpenMP*.

3.1.3 Aplicação do DAMICORE

Conforme visto na Figura 13 e Figura 14, o DAMICORE cria um arquivo em formato de uma notação, que representa a estrutura gráfica do agrupamento. Há informações de índice de identificação a qual grupo um item pertence, as respectivas distâncias entre cada elemento, além da descrição do nome abreviado de cada algoritmo (o formato desse arquivo foi explicado no Capítulo 2, na seção referente a descrição do DAMICORE). Este arquivo servirá de entrada para a próxima etapa.

3.1.4 Mapeamento da saída do DAMICORE

O mapeamento realizado no agrupamento consiste em extrair informações do arquivo gerado pelo DAMICORE para organizá-los de modo a identificar cada *benchmark* e a qual grupo ou comunidade o mesmo está associado.

```
1     ((0 popcnt----:0.23750,0 vecsum----:0.27516):0.03536,(4
      bubsort---:0.33009,
2    ((4 sobel-----:0.37479, 4 adpcmD----:0.44235):0.02423,
3    ((10 prime-----:0.38441,
4    10 fibalt----:0.43750):0.01614,(14 bound-----:0.39407,
5    (14 divlu-----:0.35407,14
6    crc32-----:0.34146):0.01310):0.01473):0.03191)
7    :0.04178):0.00229,0 fibon-----:0.28680);
8
```

Para demonstrar como o mapeamento foi implementado, temos um exemplo acima de um arquivo gerado pelo DAMICORE. O programa inicia identificando dois caracteres que correspondem a um delimitador de início, "(". O primeiro representa todo o *cluster* e o segundo o inicio de um *sub-cluster*. Nas próximas leituras são identificados o índice de um item do *cluster*, "0"no exemplo, e um critério de parada: espaço vazio, lido na próxima iteração. A seguir são feitas leituras sucessivas para a formação do nome do *benchmark*, tendo como critério de parada: ":". O caractere "-"significa que o total de caracteres disponíveis para a nome do *benchmark* não foi atingido.

O número em ponto flutuante que será lido na próxima iteração corresponde a distância que um item está de um outro no mesmo agrupamento ou que um *sub-cluster* está de um outro, porém esses dados não foram utilizados e consequentemente não foram inseridos na lista de *tokens*. Ao invés disso, foi adotado outro método para calcular a distância de um ponto ao outro (será explicado na subseção sobre o método de classificação dos *kernels*). Quando a iteração ler o caractere ",", este é armazenado na lista de *tokens*. Nas iterações seguintes, podem haver duas ações possíveis de leitura: identificar um outro item ou iniciar o mapeamento de um outro

sub-cluster. No caso do exemplo, representa o mapeamento de um *benchmark* presente no mesmo agrupamento.

Ao identificar o delimitador ")"existe também duas possibilidades de ações: ler na próximas iterações o valor da distância entre os *sub-clusters* ou identificar o delimitador de fim de arquivo ";". No caso do exemplo, é feita a leitura que identifica o valor da distância.

O fluxo de mapeamento segue até identificar todos os dados e encontrar o caractere de fim de arquivo. Caso encontre uma quebra de linha no arquivo durante as iterações, não é realizada nenhuma ação e o fluxo continua. No final teremos uma versão reduzida do arquivo gerado pelo DAMICORE. Isso servirá de entrada para o próxima etapa no qual a lista de caracteres formada é organizada de modo a criar uma estrutura em árvore

3.1.5 Organização dos Dados

A partir dos dados mapeados da etapa anterior, foi feito um método para organizar as informações de modo a criar uma estrutura em árvore que represente o formato do arquivo gerado. A criação da estrutura visa auxiliar o processo de classificação de *kernels*, provendo informações detalhadas de cada *kernel* tais como nome, identificação de comunidade e localização do item no *cluster*.

A estrutura implementada representa uma árvore b*[2] e seu processo de construção inicia pela leitura da lista de *tokens*, gerado na etapa anterior. O que define os níveis na árvore são os delimitadores, ou seja, os tokens "("e ")"nos quais demarcam o início e o fechamento de uma comunidade. Primeiramente, o método identifica o(s) *benchmark*(*s*) após a leitura do primeiro *token* de início até seu fechamento. Esse(s) *benchmark*(*s*) formarão os itens do nó raiz da árvore. Ao identificar um item, são inseridos o nome do *benchmark*, a identificação da comunidade e a qual nível da árvore o item está associado, e ponteiros para nós antecessores (apenas o nó raiz não possui) e sucessores associados também são inseridos.

Definido os itens do nó raiz, o método começa a construir os sub-níveis da árvore. O processo é similar ao passo descrito, inicia-se com a identificação do próximo *token* inicial de delimitação, mapeia todos os itens que estão no entre seu intervalo até identificar o *token* de delimitação final. Para ilustrar essa etapa, o exemplo contido na Tabela 5 demonstra uma árvore já mapeada com seus respectivos dados. Todavia esta estrutura em árvore não possui uma representação gráfica.

[2] Árvore b*: é uma técnica conhecida como *two-to-three-split*(divisão de dois para três) na qual apresenta mecanismo de que redistribui chaves durante o processo de inserção de dados em uma árvore. Isso permite adiar a divisão do agrupamento até que dois nós irmãos na árvore estejam complemente com dados. A partir disso, o conteúdo é redistribuído entre três nós.

Identificação de comunidade	Nome do Benchmark	Nível na árvore
0	Adpcm Decoder	2
0	Adpcm Coder	2
6	Fibonacci versão 1	4
6	Vector multiplication	6
6	Dotprod	6
6	Matrix Multiplication	5
4	Bubble Sort	3
14	Boundary	3
14	Divlu	4
14	Crc32	4
0	Median	1

Tabela 5 – Exemplo da organização dos dados e de sua inserção na estrutura da árvore criada.

3.1.6 Classificação dos Kernels

Esta etapa consiste em classificar os *kernels* de entrada de modo a definir qual plataforma (GPP ou FPGA) é mais adequada para o *kernel*. Isto é feito com base na identificação de *kernel(s)* de referência mais próximo ao *kernel* de entrada. O *kernel* de referência irá definir se o perfil é de GPP ou FPGA

Existe duas possibilidades de classificação quando um *kernel* de entrada está no mesmo *cluster* que um *kernel*(*s*) de referência e quando possui perfis de *kernels* de referência iguais para FPGA e GPP no *cluster*, isto inclui também quando não há nenhuma referência.

No primeiro caso, é atribuído a todos *kernels* de entrada presentes no cluster o perfil da(s) referência(s) dominante(s), ou seja, maior incidência de referências para FPGAs ou CPUs na comunidade. Essa definição é realizada sobre a estrutura em árvore criada na etapa passada. Caso não encontre nenhum ou tenha um número de perfis de *kernels* de referências iguais (Perfil de FPGA = Perfil de GPP), é realizado o segundo caso.

O segundo caso consiste em um método que identifica um *kernel* de referência de um outro *cluster* mais próximo a esse de modo que o perfil do outro agrupamento seja atribuído ao *kernel* de entrada em questão e ao restante da sua comunidade. Este método é baseado em um cálculo aplicado sobre níveis da estrutura em árvore e na lista de *kernels* de referência. O cálculo conta a quantidade de níveis percorridos na estrutura em árvore criada a partir do *kernel* de entrada identificado até encontrar todos *kernels* de referência com comunidade distinta a essa entrada. A menor distância calculada é a escolhida e sua identificação de comunidade é atribuída ao grupo. Caso o *cluster* ainda não foi mapeado com o seu respectivo perfil, é invocado o primeiro caso descrito novamente.

A atribuição para todos os *kernels* de entrada presentes em um *cluster* ou comunidade, a partir de apenas um *kernel*, visa diminuir o tempo de processamento visto que os processos que serão executados posteriormente são iguais para todos os membros da mesma comunidade.

3.2 Considerações Finais

Este capítulo demonstrou as seis etapas do desenvolvimento do classificador . O primeiro passo aborda os resultados de um conjunto de 30 *benchmarks* implementados nas ferramentas *Impulse C*, *LegUP* e *OpenMP*. Na segunda etapa, é descrita a definição dos *kernels* de referência nos quais são baseados nos resultados obtidos das ferramentas. No terceiro passo, é aplicado no DAMICORE os *kernel* definidos como referência e os *kernels* de entrada. No quarto passo, é realizado um mapeamento da representação gerada pelo DAMICORE que contem as informações do agrupamento gerado pelo passo anterior. Na quinta etapa, aborda aspectos da construção de uma representação em árvore para representar os dados mapeados pelo passo anterior. Por fim, a última etapa descreve o método desenvolvido para classificar os *kernels* de entrada em GPP ou FPGA.

CAPÍTULO

4

RESULTADOS E ANÁLISES

Este capítulo aborda os resultados provenientes dos casos de teste aplicados ao classificador de *kernels* e as análises realizadas sobre os resultados do caso de teste.

4.1 Resultados

Para a exploração do método de classificação foram criados 4 cenários de testes para a combinação das ferramentas Impulse C versus OpenMP e LegUP versus OpenMP, adotando 10 *benchmarks* de referência, conforme descrito anteriormente, sendo 5 referências de GPP e 5 referências para FPGA para a primeira combinação de ferramentas e, 3 referência de GPP e 7 de FPGA para segunda combinação.

Foram realizados 4 cenários de testes, aplicando 20 execuções de teste a cada cenário. Também foram reutilizados os *benchmarks* que não foram definidos como referência, devido ao conhecimento sobre os resultados de cada um nas ferramentas aplicadas.

Para o primeiro cenário foi inserido apenas 1 *benchmark* de entrada. Como restaram 20 *benchmarks*, não foi aplicado uma combinação dos mesmos, todos foram inseridos um a um nas execuções de teste desse cenário.

Para o segundo cenário foram inseridos 5 *benchmarks* de entrada. No terceiro cenário, 10 *benchmarks* de entrada e no quarto, 15 *benchmarks* de entrada. Para esses três últimos cenários os *benchmarks* foram escolhidos de maneira aleatória até completar a quantidade necessária de 20 execuções para cada caso de teste.

A métrica de avaliação para todos cenários é baseada em acertos e erros. Um acerto significa que o perfil atribuído pelo classificador ao dado de entrada é igual ao conhecimento prévio do perfil do mesmo sem aplicá-lo ao classificador (resultados obtidos das ferramentas). O erro é gerado quando estas duas asserções são diferentes.

Algoritmo inserido	**FPGA**	**GPP**	***Acerto e Erro***
AdpcmC		x	acertou
Autocor		x	acertou
BubbleSort		x	acertou
Change Brightness	x		errou
Compositing	x		errou
FDCT	x		errou
Fibonnaci_alt	x		errou
Fibonnaci		x	acertou
InsertionSort		x	acertou
Maximum		x	errou
Median	x		acertou
Perimeter	x		errou
Pix_sat	x		errou
Popcount	x		errou
Prime	x		errou
SelectionSort		x	acertou
Sobel	x		errou
SQRT1	x		errou
Vecsum		x	acertou
Viterbi		x	acertou

Tabela 6 – Resultados do cenário 1 para o conjunto de ferramentas *Impulse C* e *OpenMP*.

4.1.1 Impulse C versus OpenMP

Para o primeiro cenário de teste no qual foi inserido 1 *benchmarks*,os resultados obtidos podem ser observados na Tabela 6. Conforme ilustrado, é possível notar que a eficiência não é tão satisfatória, sendo que dentre as 20 execuções desse cenário, apenas 9 obtiveram êxito, isto representa apenas 45% de acerto.

No segundo cenário de teste, no qual é uma escolha aleatória de 5 *benchmarks* de entrada, foram feitas 20 avaliações. Em comparação ao primeiro cenário de teste, o resultado deste possui uma eficiência melhor, obtendo uma média de 75% de acertos e 25% de erros respectivamente, com base na Tabela 7. É possível notar que em nenhum cenário, os *benchmarks* aplicados ao projeto obtiveram 0 ou 1 acerto (0% e 20% respectivamente), a grande proporção se encontra em 3 e 5 acertos respectivamente (60% e 100%). Isto pode ser ilustrado de melhor forma nos gráficos 15 e 16, nas quais representam um histograma e descreve as informações de quantidade de acertos e erros

No terceiro cenário de teste, no qual foram aplicados 10 *benchmarks* de entrada, a média de acertos foi de 64%, baseando-se na tabela 8. Conforme os resultados, a maior concentração de acertos estão entre 60% a 70%, isto representa 6 e 7 acertos respectivamente. No total, foram 7 incidências para 6 acertos e 8 para 7 acertos. As incidências de acertos e erros podem ser visualizadas nos histogramas ilustrados pelas Figuras 17 e 18.

Cenário	Acerto FPGA	Acerto GPP	*Erro FPGA*	Erro GPP	Índice de acerto (%)	Índice de erro (%)
1	2	3	0	0	100	0
2	1	1	0	3	40	60
3	2	1	0	2	60	40
4	2	2	0	1	80	20
5	1	3	0	0	100	0
6	1	2	0	2	60	40
7	1	2	0	1	80	20
8	1	2	0	2	60	40
9	2	1	1	1	60	40
10	1	2	0	2	60	40
11	2	1	0	2	60	40
12	0	3	0	0	100	0
13	2	1	0	0	100	0
14	1	1	0	3	40	60
15	2	3	0	0	100	0
16	2	2	0	1	80	20
17	1	1	0	3	60	40
18	1	2	0	2	60	40
19	2	3	0	0	100	0
20	2	3	0	0	100	0
				Média	75	25

Tabela 7 – Resultados do cenário 2 para o conjunto de ferramentas *Impulse C* e *OpenMP*.

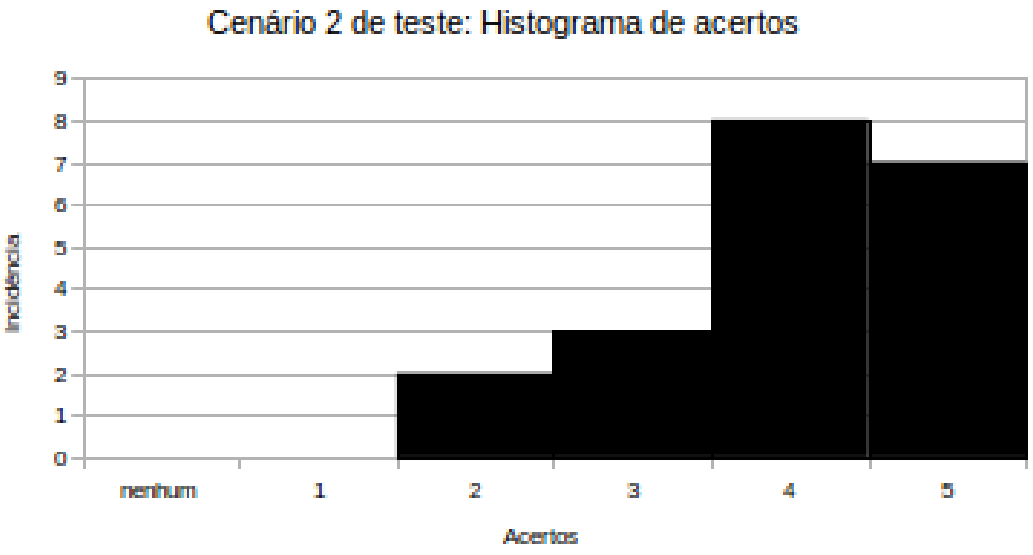

Figura 15 – Cenário 2 de teste: histograma de acertos nas ferramentas *Impulse C* e *OpenMP*.

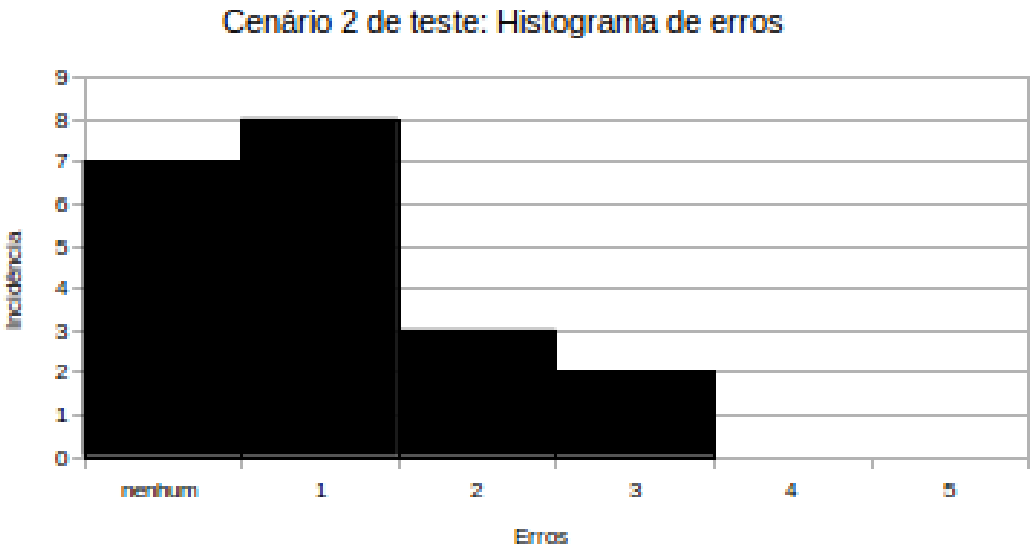

Figura 16 – Cenário 2 de teste: histograma de erros nas ferramentas *Impulse C* e *OpenMP*.

Cenário	**Acerto FPGA**	**Acerto GPP**	***Erro FPGA***	**Erro GPP**	**Índice de acerto (%)**	**Índice de erro (%)**
1	0	7	2	1	70	30
2	1	4	2	3	50	50
3	2	5	0	3	70	30
4	2	4	0	4	60	40
5	1	5	3	1	60	40
6	1	5	1	3	60	40
7	2	6	1	1	80	20
8	0	7	2	1	70	30
9	2	5	1	2	70	30
10	2	5	0	3	70	30
11	1	6	0	3	70	30
12	2	5	0	3	70	30
13	2	4	1	3	60	40
14	2	4	1	3	60	40
15	1	3	0	6	40	60
16	2	3	0	5	50	50
17	1	5	1	3	60	40
18	1	5	0	4	60	40
19	2	5	1	2	70	30
20	1	7	2	0	80	20
				Média	64	36

Tabela 8 – Resultados do cenário 3 para o conjunto de ferramentas *Impulse C* e *OpenMP*.

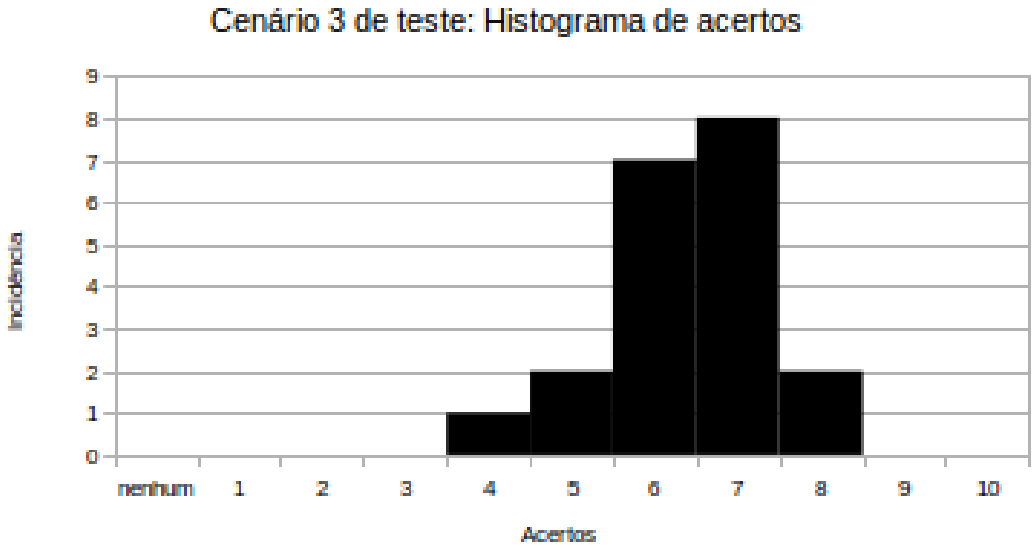

Figura 17 – Cenário 3 de teste: histograma de acertos nas ferramentas *Impulse C* e *OpenMP*.

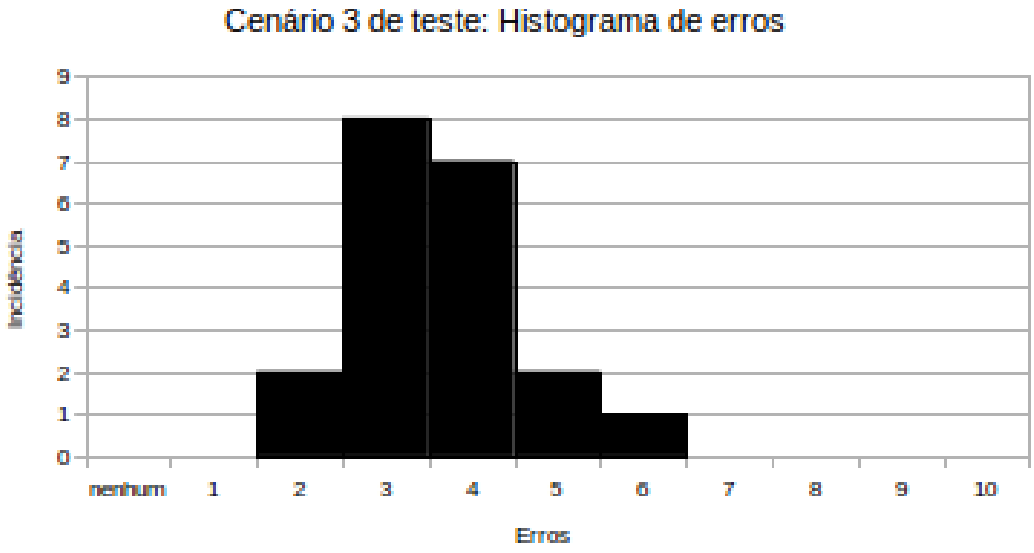

Figura 18 – Cenário 3 de teste: histograma de erros nas ferramentas *Impulse C* e *OpenMP*.

Cenário	Acerto FPGA	Acerto GPP	*Erro FPGA*	Erro GPP	Índice de acerto (%)	Índice de erro (%)
1	1	8	1	5	60	40
2	1	9	2	3	66,67	33,33
3	2	6	1	6	53,34	46,66
4	2	6	1	6	53,34	46,66
5	1	7	3	4	53,34	46,66
6	2	6	1	6	53,34	46,66
7	2	6	2	5	53,34	46,66
8	3	7	0	5	66,67	33,33
9	2	6	0	7	53,34	46,66
10	2	6	1	6	53,34	46,66
11	2	6	1	6	53,34	46,66
12	2	5	1	7	46,67	53,33
13	2	4	1	8	40	60
14	2	6	1	6	53,34	46,66
15	2	5	2	6	46,67	53,33
16	2	4	2	7	40	60
17	2	5	1	7	46,67	53,33
18	2	6	1	6	53,34	46,66
19	2	6	1	6	53,34	46,66
20	2	6	2	5	53,34	46,66
				Média	52,68	47,32

Tabela 9 – Resultados do cenário 4 para o conjunto de ferramentas *Impulse C* e *OpenMP*.

No quarto cenário, no qual foi aplicado 15 *benchmarks* de entrada, a média de acertos foi inferior que ao cenário 2 e 3, tendo como resultado 52,68% de eficiência, conforme a tabela 9. A maior concentração são de 8 acertos, contendo 13 incidências. Isto representa apenas 53,34% do total de 15 *benchmarks*. O panorama geral de acertos e erros podem ser ilustrado pelo histograma presente na figura 19 e figura 20.

Para visualizarmos de maneira geral todos os cenários de teste, foi feito um diagrama de caixas (boxplot) presente na figura 21 e figura 22 para ilustrar os intervalos de acertos e erros, descrevendo os valores mínimos e máximo, mediana e primeiro e terceiro quartis. Com base nesse diagrama, o melhor resultado foi referente ao cenário 2, o primeiro quartil varia de 60% a 70% para acertos e de 0% a 30% para erros; a mediana concentrou-se em 70% e o terceiro quartil, variando de 70% a 100%.

Paro o cenário 3, a variação de acertos para o primeiro quartil variou aproximadamente de 60% a 65%, e de erros em torno de 30% a 40%; a mediana concentrou-se em 65% para acertos e 35% para erros; e o terceiro quartil entre 65% a 70% para acertos e 35% a 40% para erros.

No ultimo cenário, a variação de acertos para o primeiro quartil variou entre 50% a 55% e a variação de erros entre 45% a 50% aproximadamente. A mediana concentrou-se em 55% aproximadamente para acertos e 45% para erros; o terceiro quartil não houve dados entre o

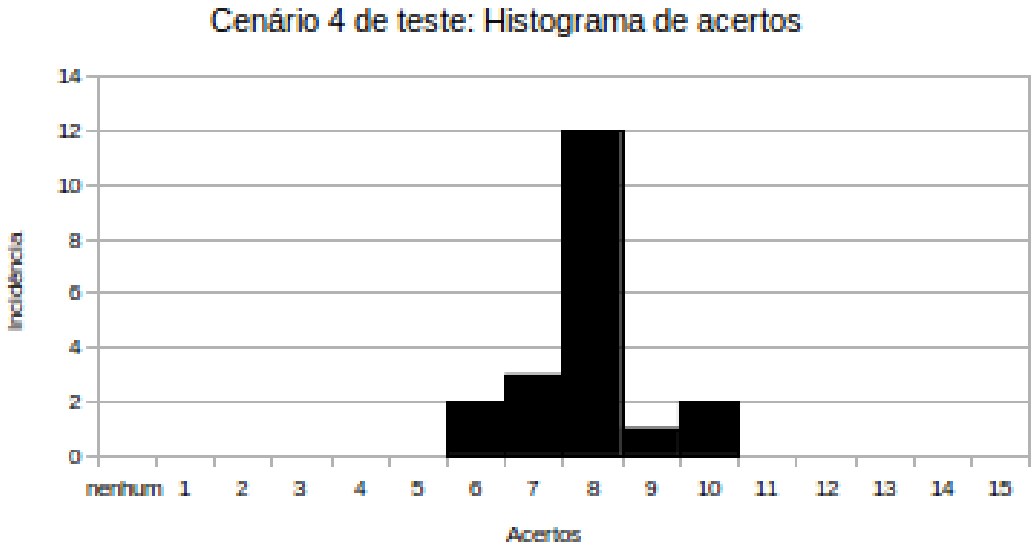

Figura 19 – Cenário 4 de teste: histograma de acertos nas ferramentas *Impulse C* e *OpenMP*.

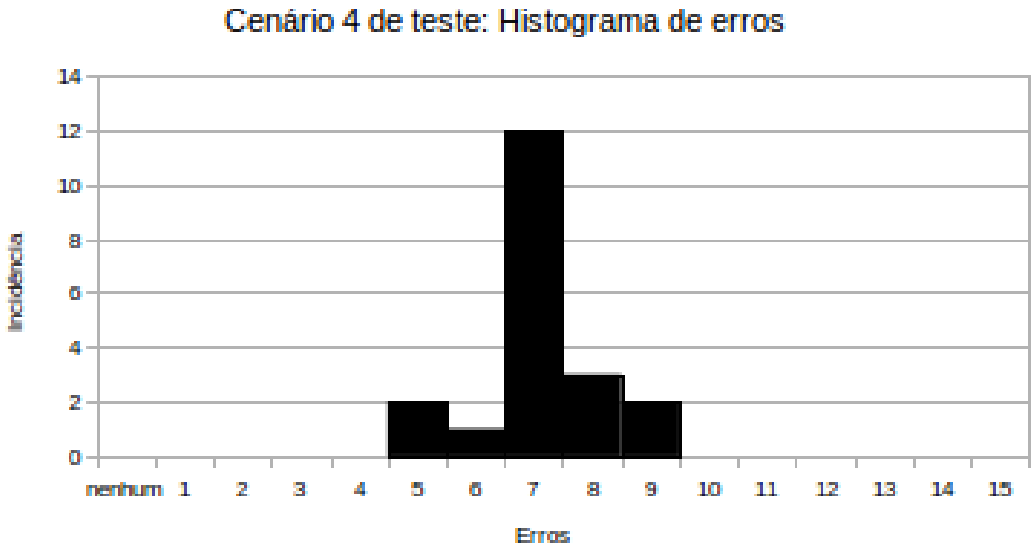

Figura 20 – Cenário 4 de teste: histograma de erros nas ferramentas *Impulse C* e *OpenMP*.

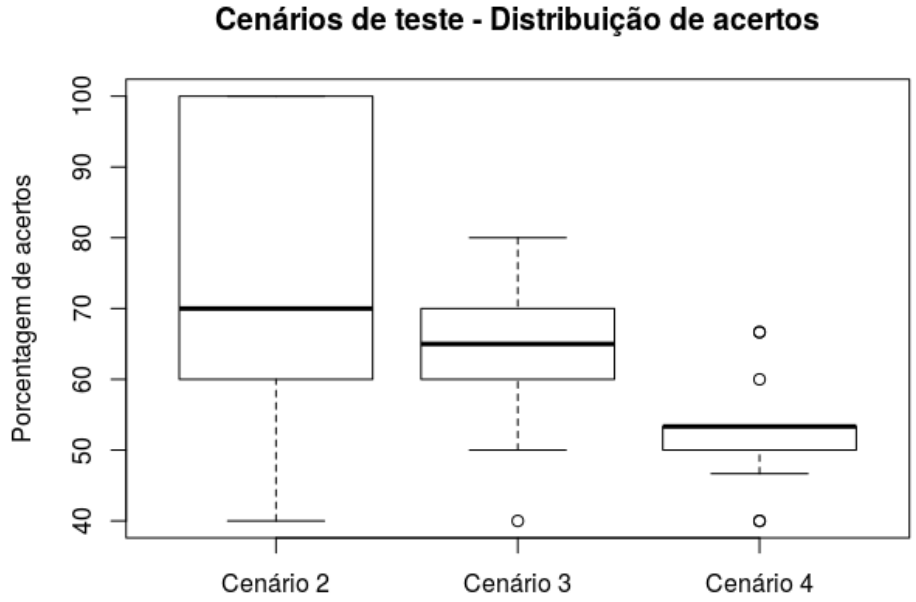

Figura 21 – Distribuição dos acertos para as ferramentas *Impulse C* e *OpenMP*.

intervalo para os acertos e para erros houve variação entre 45% a 50% aproximadamente. É importante ressaltar que o primeiro cenário não foi inserido no diagrama, porque foi aplicado apenas 1 entrada, ou seja, ou há apenas um acerto ou um erro (100% ou 0%).

Mesmo o cenário 2 possuindo uma grande variação entre acertos e erros, ainda assim é o melhor cenário. Como foi ilustrado na tabela de resultados, os maiores índices de acertos estão entre 80% e 100% (4 e 5 acertos) e de erros entre 0 a 20% (0 e 1 erros).

4.1.2 LegUP e OpenMP

Nesta subseção serão mostrados os resultados referentes aos quatro cenários de teste para as ferramentas LegUP versus OpenMP. O resultado obtido na primeiro cenário, no qual é feito a inserção de 1 algoritmo do conjunto restante de *benchmarks*, é ilustrado na tabela 10, conforme podemos observar o índice de acertos equivale a 55% enquanto os erros 45%, isto representa 11 acertos contra 9 erros.

No segundo cenário no qual são inseridos 5 *benchmarks* de entrada, o resultado foi melhor comparado com o primeiro cenário, que obteve uma média de 75% de acertos contra apenas 25% de erros, como podemos ver na tabela 11. Em mais da metade das execuções para este cenário, os resultados ficaram entre 80% e 100%, isso representa 4 e 5 acertos respectivamente. O índice de incidência de acertos e erros podem ser melhor visualizadas por meio das Figuras 23 e 24.

No terceiro cenário no qual são inseridos 10 benchmarks de entrada, os resultados foram inferiores ao segundo cenário, possuindo uma média de 67% de índice de acertos, conforme

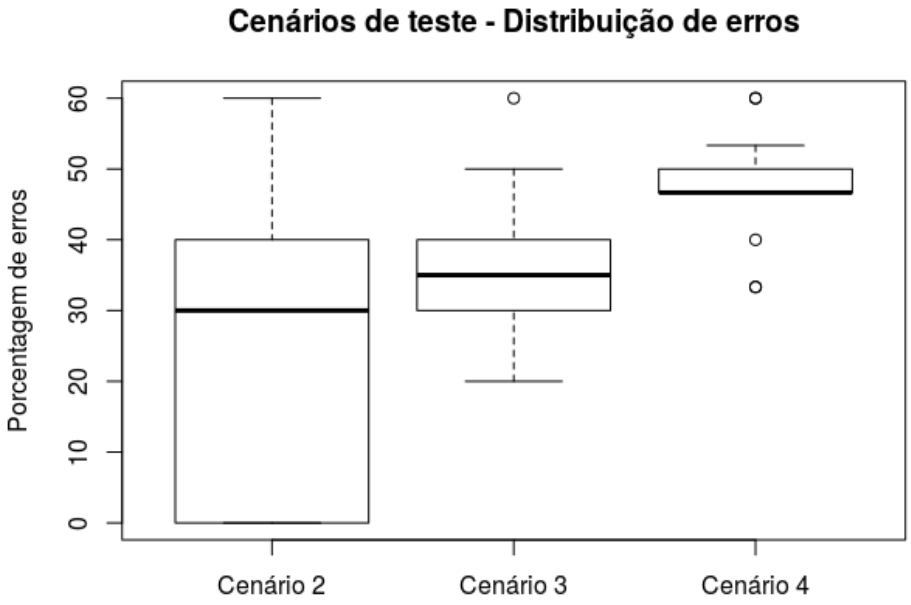

Figura 22 – Distribuição dos erros para as ferramentas *Impulse C* e *OpenMP*.

Algoritmo inserido	**FPGA**	**GPP**	***Acerto e Erro***
AdpcmC	x		acertou
Autocor	x		errou
BubbleSort		x	acertou
Compositing	x		errou
Conv3x3	x		acertou
crc32	x		acertou
divlu		x	errou
FDCT	x		errou
Fibonacci	x		errou
InsertionSort		x	acertou
Mamul(50x50)	x		errou
Maximum	x		acertou
Median	x		acertou
Perimeter	x		errou
Pix_sat		x	errou
Popcount	x		acertou
Prime		x	acertou
SQRT1		x	errou
Vecsum	x		acertou
Viterbi	x		acertou

Tabela 10 – Resultados do cenário 1 para o conjunto de ferramentas *LegUP* e *OpenMP*.

Cenário	Acerto FPGA	Acerto GPP	*Erro FPGA*	Erro GPP	Índice de acerto (%)	Índice de erro (%)
1	3	1	1	0	80	20
2	4	1	0	0	100	0
3	2	1	1	1	60	40
4	3	2	0	0	100	0
5	3	1	1	0	80	20
6	2	1	0	2	60	40
7	3	2	0	0	100	0
8	4	0	0	1	80	20
9	3	1	0	1	80	20
10	3	1	0	1	80	20
11	3	1	1	0	80	20
12	2	0	0	3	40	60
13	1	2	1	1	60	40
14	3	2	0	0	100	0
15	0	2	0	3	40	60
16	4	1	0	0	100	0
17	2	2	1	0	80	20
18	2	1	0	2	60	40
19	1	1	1	2	40	60
20	3	1	0	1	80	20
				Média	75	25

Tabela 11 – Resultados do cenário 2 para o conjunto de ferramentas *LegUP* e *OpenMP*.

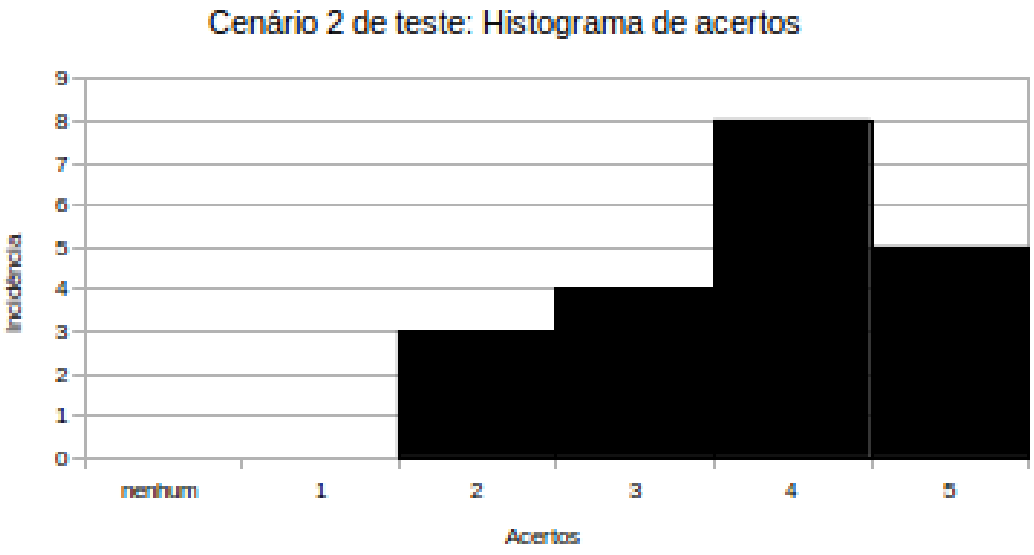

Figura 23 – Cenário 2 de teste: histograma de acertos nas ferramentas *LegUP* e *OpenMP*.

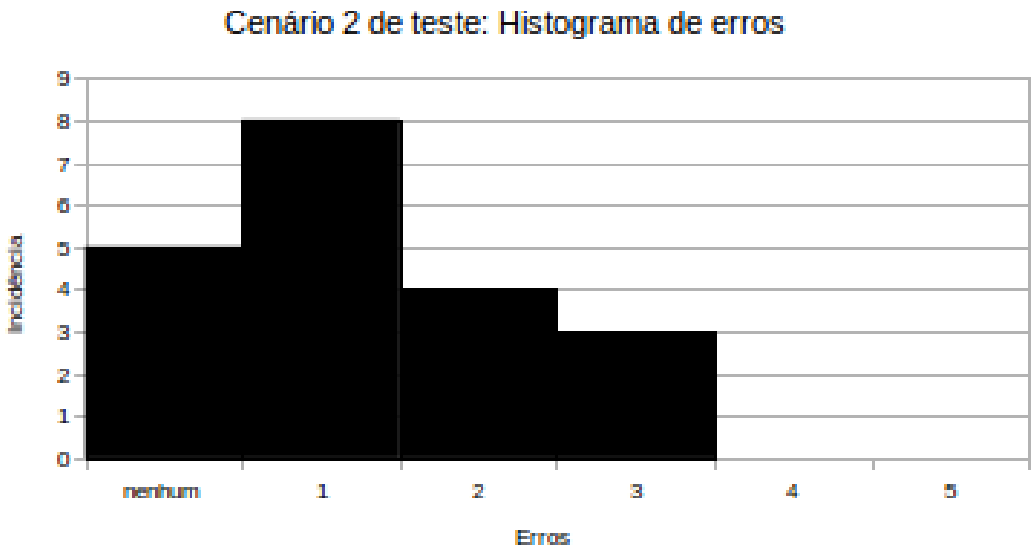

Figura 24 – Cenário 2 de teste: histograma de erros nas ferramentas *LegUP* e *OpenMP*.

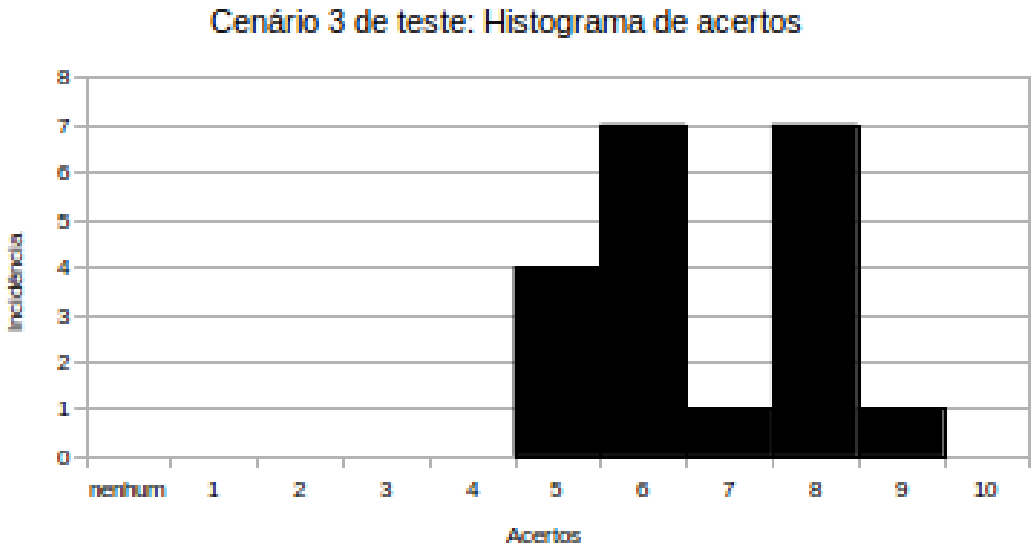

Figura 25 – Cenário 3 de teste: histograma de acertos nas ferramentas *LegUP* e *OpenMP*.

mostrado na tabela 12. Nota-se que a maior concentração de acertos está em 60% e 80%, isto representa 6 e 8 acertos respectivamente. Em nenhum caso, houve valores inferiores a 50%, embora houve 4 incidências nesse respectivo valor. Para obtermos um panorama de acertos e erros, os histogramas presentes na figuras 25 e 26 ilustram onde estão estas incidências.

No último cenário, os resultados obtidos a partir da inserção de 15 *benchmarks* de entrada foram inferiores ao terceiro cenário, com um média de 65,67% em acertos, isto representa aproximadamente 10 acertos de 15. Esse fato pode ser confirmado ao olharmos a tabela 13, onde o valor de 66,67% representa 10 acertos. Para uma melhor visualização do número de incidência

Cenário	Acerto FPGA	Acerto GPP	*Erro FPGA*	Erro GPP	Índice de acerto (%)	Índice de erro (%)
1	3	2	1	4	50	50
2	2	4	0	4	60	40
3	6	0	1	3	60	40
4	6	2	1	1	80	20
5	6	2	1	1	80	20
6	5	3	0	2	80	20
7	3	2	1	4	50	50
8	3	2	1	4	50	50
9	3	2	0	5	50	50
10	4	4	0	2	80	20
11	3	3	0	4	60	40
12	3	3	2	2	60	40
13	4	2	1	3	60	40
14	5	3	0	2	80	20
15	5	4	1	0	90	10
16	4	4	1	1	80	20
17	5	2	1	2	70	30
18	3	3	1	3	60	40
19	5	1	1	3	60	40
20	6	2	1	1	80	20
				Média	67	33

Tabela 12 – Resultados do cenário 3 para o conjunto de ferramentas *LegUP* e *OpenMP*.

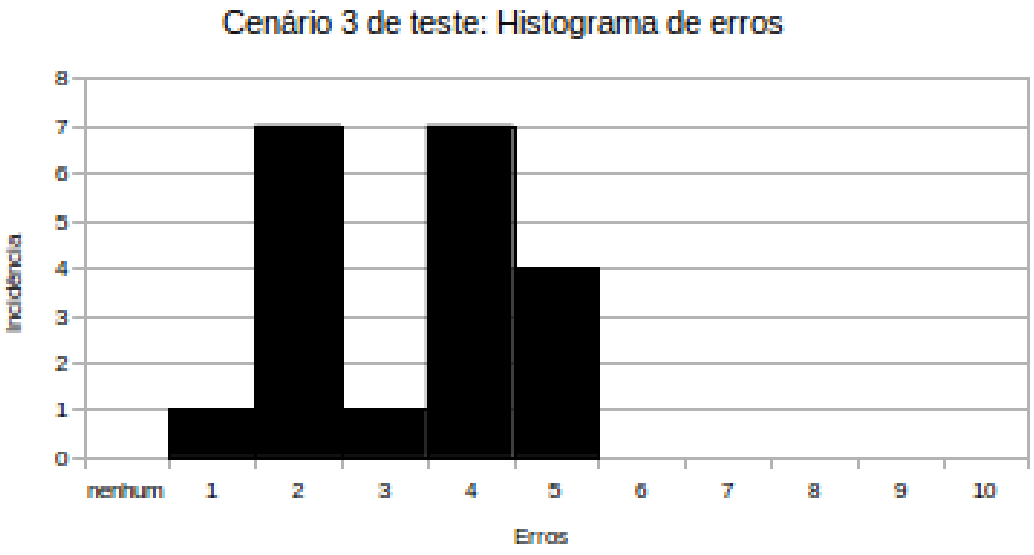

Figura 26 – Cenário 3 de teste: histograma de erros nas ferramentas *LegUP* e *OpenMP*.

Cenário	Acerto FPGA	Acerto GPP	*Erro FPGA*	Erro GPP	Índice de acerto (%)	Índice de erro (%)
1	7	4	0	4	73,34	26,66
2	5	4	2	4	60	40
3	7	3	0	5	66,67	33,33
4	7	4	1	3	73,34	26,66
5	5	5	2	3	66,67	33,33
6	5	5	1	4	66,67	33,33
7	5	5	0	5	66,67	33,33
8	5	3	2	5	53,34	46,66
9	8	5	1	1	86,67	13,33
10	7	3	0	5	66,67	33,33
11	6	2	1	6	53,34	46,66
12	5	5	1	4	66,67	33,33
13	5	5	1	4	66,67	33,33
14	6	4	1	4	66,67	33,33
15	6	5	0	4	73,34	26,66
16	8	2	1	4	66,67	33,33
17	4	3	2	6	46,67	53,33
18	6	4	2	3	66,67	33,33
19	7	2	1	5	60	40
20	6	4	1	4	66,67	33,33
				Média	65,67	34,33

Tabela 13 – Resultados do cenário 4 para o conjunto de ferramentas *LegUP* e *OpenMP*.

de acertos e erros, os histogramas presentes nas figuras 27 e 28 os ilustram.

Para observarmos melhor as variações de acertos e erros presentes em todos os cenários descritos, apresenta-se o diagrama de caixas presentes na Figura 29 e Figura 30. No cenário 2, o primeiro quartil variou entre 80% a 90% para acertos e entre 20% a 40% para erros; a mediana concentrou-se em 80% para acertos e 20% para erros; e o terceiro quartil entre 60% a 80% para acertos e 10% a 20% para erros

No terceiro cenário, o primeiro quartil variou entre 60% a 80% para acertos e para erros não houve variação; a mediana concentrou entre 60% para acertos e 40% para erros; e o terceiro quartil não houve variação para acertos e para erros, entre 20% a 40%

O melhor cenário para o conjunto de ferramentas *LegUP* e *OpenMP* foi o segundo, tendo como média 75% de acertos e 25% de erros nas 20 execuções.

4.2 Análises dos Resultados

Nesta seção serão analisados os resultados obtidos de classificação de cada execução dos cenários testes para o conjunto de ferramentas exploradas com o DAMICORE. Como podemos observar, o resultado dos cenários 1 de ambos foram ruins. Os motivos para esse caso

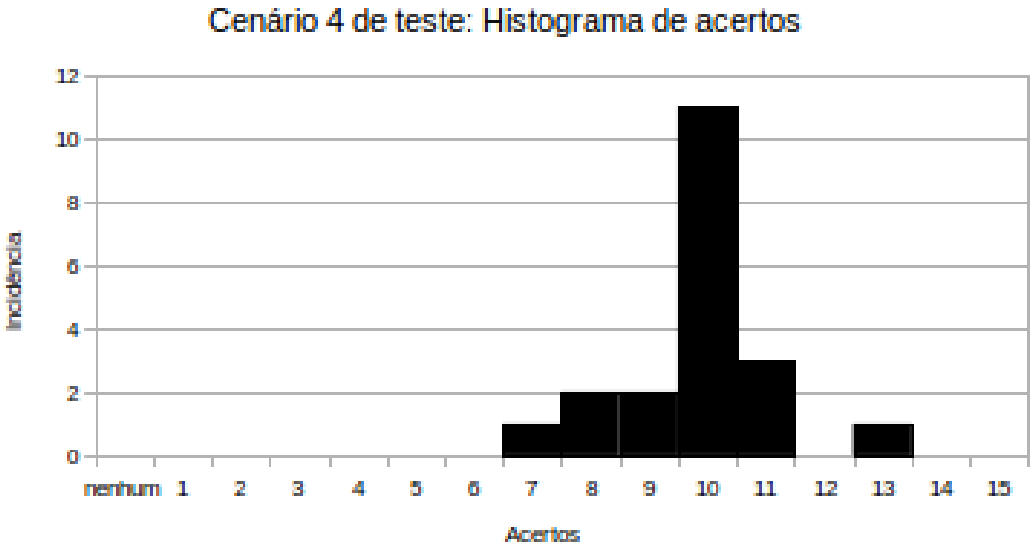

Figura 27 – Cenário 4 de teste: histograma de acertos nas ferramentas *LegUP* e *OpenMP*.

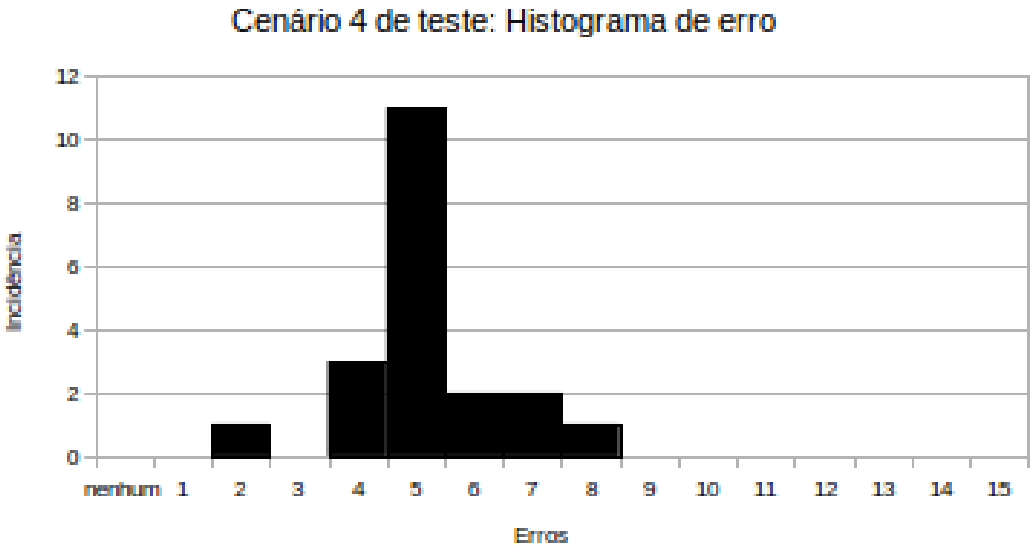

Figura 28 – Cenário 4 de teste: histograma de erros nas ferramentas *LegUP* e *OpenMP*.

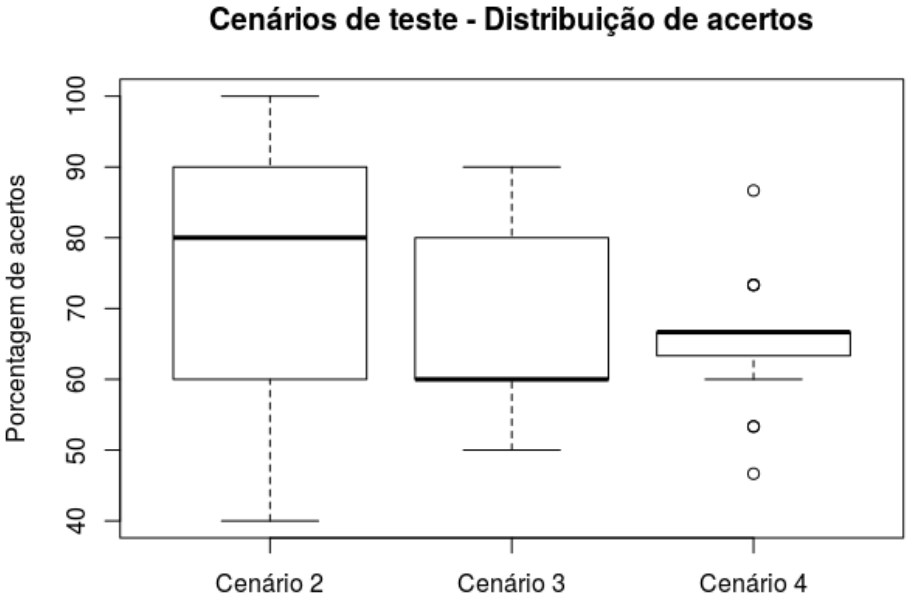

Figura 29 – Distribuição dos acertos para as ferramentas *LegUP* e *OpenMP*.

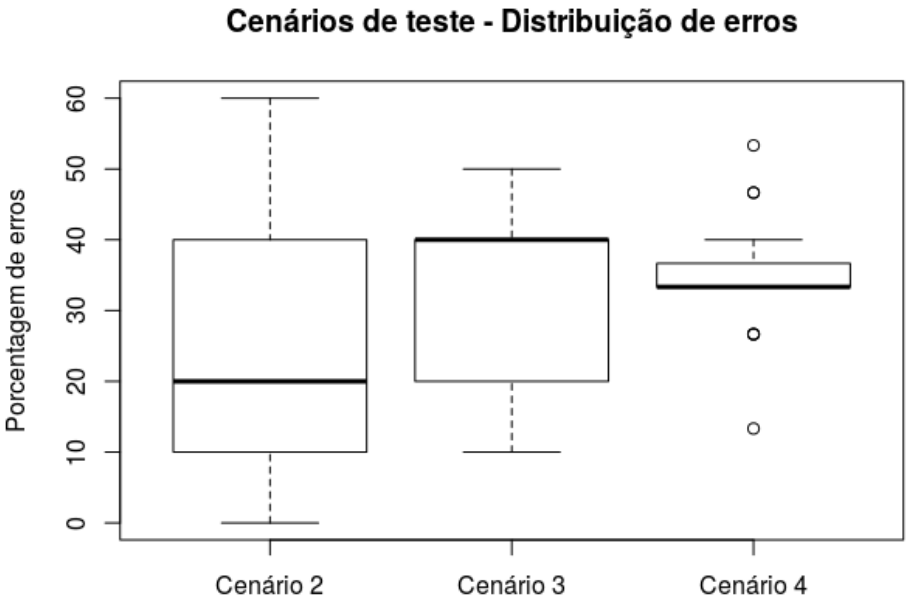

Figura 30 – Distribuição dos erros para as ferramentas *LegUP* e *OpenMP*.

são ocasionados pela quantidade de elementos para a formação do agrupamento. Com uma quantidade reduzida de elementos, o agrupamento formado é menor e o *kernel* pode ser associado a uma comunidade com um perfil totalmente diferente ao seu, ou seja, quando há poucos dados o DAMICORE pode encontrar uma pequena semelhança entre um elemento e outro e colocá-lo na mesma comunidade. Desse modo o erro é gerado.

Para os cenários 2, 3 e 4 dos testes tanto do *Impulse C* e *LegUP*, a principal influência são os resultados providos das ferramentas de síntese dado que nem sempre uma ferramenta consegue executar um algoritmo de maneira eficiente e prover um resultado melhor. Como vimos nos resultado de cada ferramenta, houveram casos em que a versão em hardware perdeu para a versão em software, e vice-versa. Isso pode gerar comunidades no agrupamento com perfis de elementos totalmente diferentes. Para o classificador, isso é o causador de erros dado que o classificador analisa a quantidade de referências dentro da comunidade, se possuir perfis diferentes de referência, o que possuir a maior incidência é o perfil atribuído aos *kernels* de entrada. Dessa maneira, pode ser atribuído um perfil diferente do esperado.

Outro fator, foi a formação de comunidades sem a presença de *benchmarks* de referência para os cenário 3 e 4. Dessa maneira, o classificador busca uma comunidade mais próxima e adiciona o perfil da mesma encontrada a comunidade em questão. Porém se esse grupo identificado possuir características totalmente diferentes do perfil que é esperado o erro é gerado.

A ferramenta DAMICORE também pode ter contribuído para a geração de erros em todos os cenários, devido ao compressor aplicado pelo NCD, o zlib. Há possibilidades que o compressor não consiga uma taxa de compressão/descompressão satisfatória, isto pode implicar no resultado final do classificador desenvolvido.

Um outro aspecto que deve-se levar em consideração é que o código fonte em alto nível, no caso a linguagem C, pode não conter elementos suficientes para o NCD do DAMICORE. Uma alternativa, poderia ser o uso de outras representações, por exemplo, considerar também as representações RTL e *assembly* deste códigos.

Conforme os resultados obtidos pela classificação de *kernels*, o melhor cenário de teste para ambas plataformas híbridas foram o segundo caso de teste, obtendo uma eficiência de 75% em acertos para as duas combinações de ferramentas *Impulse C* versus *OpenMP* e *LegUP* versus *OpenMP*.

4.3 Considerações finais

Este capítulo abordou os resultados obtidos pelo método de classificação e análises feitas com base nos resultados. Na seção de resultados foram ilustrados quatro casos de teste para o conjunto de ferramentas *Impulse C* versus *OpenMP* e *LegUP* versus *OpenMP*. Os resultados demonstraram para ambos casos que o segundo cenário de teste possuiu o maior índice de

acertos ao serem comparados com outros cenários. Nas análises realizadas foram descritos os motivos dos resultados gerados para cada caso de teste. Aspectos como o tamanho das comunidades, quantidade de *kernels* de entrada, formação de comunidades sem *kernel*(s) de referência e a influência dos resultados obtidos pelas ferramentas são os fatores que influenciaram nos resultados.

CAPÍTULO

5

CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo apresenta um resumo do trabalho realizado, uma conclusão sobre o classificador desenvolvido, suas contribuições e algumas direções para trabalhos futuros.

5.1 Resumo do trabalho

Este trabalho teve como objetivo prover método de classificação de *kernel* para ambientes híbridos de processamento que se baseia em uma técnica de mineração de dados, chamada DAMICORE, para encontrar a relação de similaridade entre *kernels*. Foram utilizadas as ferramentas *Impulse C* e *LegUP* para sintetizar *aplicações* para FPGA e a ferramenta OpenMP para compilação de programas voltados para GPP *multi-core*. A partir dos resultados providos de cada ferramenta, foi possível criar os casos de teste para avaliar a eficácia do classificador desenvolvido.

Durante o desenvolvimento, foram realizados quatro experimentos para os ambientes híbridos, onde para cada cenário foi aplicado uma variação no número de *kernels* de entrada inseridos ao classificador. A variação corresponde a inserção de 1, 5, 10 e 15 *kernels* de entrada ao método.

Para validar o classificador, compararam-se a taxa de acertos e erros dos *kernels* de entrada ao serem inseridos ao método de classificação com o conhecimento prévio dos mesmos. Os resultados demonstraram que o método é eficaz para alguns cenários de teste aplicados. Porém, o classificador depende principalmente da capacidade das ferramentas de síntese para FPGA, das escolhas dos *kernels* de referência e a quantidade de elementos no *cluster* para gerar um resultado satisfatório.

5.2 Conclusão do trabalho

Há diversos trabalhos na literatura que buscam obter um melhor aproveitamento dos recursos lógicos de um hardware, seja ele uma GPP, GPU ou FPGA. O trabalho desenvolvido não é diferente, o intuito é que o usuário ao utilizar a técnica tenha um método automático que auxilie na tomada de decisão para decidir se um *kernel* é mais adequado as plataformas GPP ou FPGA.

Em relação ao trabalho desenvolvido, pode-se concluir que o DAMICORE é uma opção de ferramenta válida para a classificação de *kernels* em arquiteturas híbridas, porém o usuário deve ter cuidado com os *kernels* de entrada, pois a ferramenta possui uma sensibilidade a todo o conteúdo existente no código fonte. Isso se deve ao fato de que a ferramenta aplica a métrica NCD para realizar a compressão dos dados e descobrir a sua relação. Por exemplo, o usuário pode utilizar declarações de variáveis com nomes genéricos ou esquecer comentários no código, esses fatores podem influenciar a geração do agrupamento dos dados.

Um outro ponto a ser levantado, é a definição dos *kernels* de referência. A escolha dessas referências é minuciosa dado que são elas que guiam o classificador a definir um perfil a um *kernel* de entrada. Se escolhermos uma referência sem aplicar um critério bem definido, isso pode implicar diretamente no resultado final.

Cabe ressaltar, que as referências são dependentes das ferramentas. Assim, caso o usuário queira utilizar uma outra combinação de ferramenta de síntese para hardware e de compilação para software um novo conjunto de referências deverá ser gerado. Porém, assim que definido este conjunto, as mesmas referências podem ser reutilizadas para classificar códigos de entrada de perfil desconhecido e assim possibilitar a automatização do processo de classificação.

É importante ressaltar que os resultados do classificador são também influenciados pelas ferramentas tanto para FPGA quanto para GPP. Como foi dito na seção de análise dos resultados do capítulo anterior, nem sempre a ferramenta consegue explorar de maneira eficiente uma aplicação e gerar um resultado melhor. No agrupamento, isso pode implicar na formação de comunidades com perfis de elementos totalmente diferentes.

Com base nos resultados providos do classificador nos casos de teste, foi demonstrado que esta abordagem pode ser empregada como um ponto de decisão inicial no processo de mapeamento em sistemas híbridos compostos por GPP e FPGA, somente analisando o perfil do código fonte sem que haja a necessidade do usuário executar o mesmo para a tomada de decisão.

5.3 Contribuições do trabalho

As principais contribuições desta pesquisa foram:

- Uma técnica de classificação automática de *kernels* para ambientes híbridos de processa-

mento compostos por GPP e FPGA de modo que o usuário não precise executar a aplicação a priori para definir o dispositivo mais adequado a plataforma de processamento.

- Um estudo que serviu de base para explorar a potencialidade do DAMICORE no tipo de problema abordado.
- Por fim, este trabalho demonstrou que é possível extrair informações de desempenho de um código fonte C a partir da similaridade do mesmo com outro código fonte de desempenho conhecido (*kernels* de referências) e que isso é fortemente influenciado pela capacidade de otimização das ferramentas de síntese.

5.4 Trabalhos Futuros

Findada a realização deste trabalho, ainda há espaço para a continuação e aperfeiçoamento do método. Portanto, nesta seção, são descritos algumas sugestões de trabalhos futuros.

- Estender o ambiente híbrido para outros dispositivos de hardware, como por exemplo GPU.
- Adicionar outras métricas de avaliação no classificador, como por exemplo informações sobre o consumo de energia.
- Aplicar outras abordagem para a classificação de *kernels* de modo a comparar a eficiência.
- Utilizar outras ferramentas de síntese para FPGA de modo a analisar o comportamento delas no classificador.
- Realizar a variação na quantidade de *kernels* de entrada e/ou *kernels* de referência para novos casos de teste para análise de comportamento do classificador.
- Avaliar a influência do modo de programar (ou do programador) no resultado do classificador.

REFERÊNCIAS

Altera Corporation. **Intel Atom Processor E6x5C Series**. 2010. <http://www.altera.com/devices/processor/intel/e6xx/proc-e6x5c.html>. [Online; acessado em 20-03-2013]. Citado na página 24.

AUERBACH, J.; BACON, D. F.; BURCEA, I.; CHENG, P.; FINK, S. J.; RABBAH, R.; SHUKLA, S. A compiler and runtime for heterogeneous computing. In: **Proceedings of the 49th Annual Design Automation Conference**. New York, NY, USA: ACM, 2012. (DAC '12), p. 271–276. ISBN 978-1-4503-1199-1. Disponível em: <http://doi.acm.org/10.1145/2228360.2228411>. Citado 2 vezes nas páginas 42 e 43.

BERKHIN, P. **Survey Of Clustering Data Mining Techniques**. [S.l.], 2002. Citado 3 vezes nas páginas 27, 28 e 29.

BONATO, V.; MARQUES, E.; CONSTANTINIDES, G. A parallel hardware architecture for scale and rotation invariant feature detection. **Circuits and Systems for Video Technology, IEEE Transactions on**, IEEE, v. 18, n. 12, p. 1703–1712, dez. 2008. ISSN 1051-8215. Citado na página 24.

BORKAR, S.; CHIEN, A. A. The future of microprocessors. **Commun. ACM**, ACM, New York, NY, USA, v. 54, n. 5, p. 67–77, maio 2011. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1941487.1941507>. Citado na página 23.

BORKAR, S. Y.; DUBEY, P.; KAHN, K. C.; KUCK, D. J.; MULDER, H.; PAWLOWSKI, S. S.; RATTNER, J. R. Platform 2015: Intel Processor and Platform Evolution for the Next Decade. **Intel White Paper**, July 2005. Disponível em: <ftp://download.intel.com/technology/computing/archinnov/platform2015/download/Platform_2015.pdf>. Citado na página 23.

CANIS, A.; ANDERSON, J. H.; BROWN, S. D. Multi-pumping for resource reduction in fpga high-level synthesis. In: **Proceedings of the Conference on Design, Automation and Test in Europe**. San Jose, CA, USA: EDA Consortium, 2013. (DATE '13), p. 194–197. ISBN 978-1-4503-2153-2. Disponível em: <http://dl.acm.org/citation.cfm?id=2485288.2485338>. Citado na página 41.

CANIS, A.; CHOI, J.; ALDHAM, M.; ZHANG, V.; KAMMOONA, A.; CZAJKOWSKI, T.; BROWN, S. D.; ANDERSON, J. H. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. **ACM Trans. Embed. Comput. Syst.**, ACM, New York, NY, USA, v. 13, n. 2, p. 24:1–24:27, set. 2013. ISSN 1539-9087. Disponível em: <http://doi.acm.org/10.1145/2514740>. Citado 3 vezes nas páginas 15, 39 e 40.

____. ____. **ACM Trans. Embed. Comput. Syst.**, ACM, New York, NY, USA, v. 13, n. 2, p. 24:1–24:27, set. 2013. ISSN 1539-9087. Disponível em: <http://doi.acm.org/10.1145/2514740>. Citado na página 39.

CHAPMAN, B.; JOST, G.; PAS, R. Van der; KUCK, D. J. **Using OpenMP : portable shared memory parallel programming**. Cambridge, Mass., London: The MIT Press, 2008. ISBN

978-0-262-53302-7. Disponível em: <http://opac.inria.fr/record=b1125568>. Citado 2 vezes nas páginas 41 e 42.

CHE, S.; LI, J.; SHEAFFER, J. W.; SKADRON, K.; LACH, J. Accelerating Compute-Intensive Applications with GPUs and FPGAs. **Symposium on Application Specific Processors**, IEEE Computer Society, Los Alamitos, CA, USA, p. 101–107, jun. 2008. Citado na página 24.

CILIBRASI, R.; VITáNYI, P. M. B. Clustering by compression. **IEEE Transactions on Information Theory**, v. 51, p. 1523–1545, 2005. Citado na página 32.

GAILLY, J.-l.; ADLER, M. **Zlib 1.2.8 manual**. [S.l.], 2013. Citado na página 32.

GOKHALE, M.; STONE, J.; ARNOLD, J.; KALINOWSKI, M. Stream-oriented fpga computing in the streams-c high level language. In: **Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on**. [S.l.: s.n.], 2000. p. 49–56. Citado na página 37.

HARING, R.; OHMACHT, M.; FOX, T.; GSCHWIND, M.; SATTERFIELD, D.; SUGAVANAM, K.; COTEUS, P.; HEIDELBERGER, P.; BLUMRICH, M.; WISNIEWSKI, R.; GARA, A.; CHIU, G.-T.; BOYLE, P.; CHIST, N.; KIM, C. The IBM Blue Gene/Q Compute Chip. **Micro, IEEE**, v. 32, n. 2, p. 48–60, abr. 2012. ISSN 0272-1732. Citado na página 23.

HOARE, C. A. R. Communicating sequential processes. **Commun. ACM**, ACM, New York, NY, USA, v. 21, n. 8, p. 666–677, ago. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359576.359585>. Citado na página 37.

Hugo. **PLUTO - An automatic parallelizer and locality optimizer for multicores**. 2015. <http://www.onmyphd.com/?p=k-means.clustering&ckattempt=1/>. [Online; acessado em 2015-11-30]. Citado 2 vezes nas páginas 15 e 31.

IMPULSEC. 2010. <http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm>. Accessed: 2014-09-30. Citado 2 vezes nas páginas 15 e 38.

Intel Corporation. **Intel Atom Processor E6x5C Series-Based Platform for Embedded Computing**. 2010. <http://download.intel.com/embedded/processors/prodbrief/324535.pdf>. [Online; acessado em 20-03-2013]. Citado na página 24.

____. **Altera to Build Next-Generation, High-Performance FPGAs on Intel's 14 nm Tri-Gate Technology**. 2013. [Online; acessado em 20-03-2013]. Citado na página 24.

____. **Intel Completes Acquisition of Altera**. 2015. <https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera>. [Online; acessado em 01-02-2016]. Citado na página 24.

ITO, K.; ZEUGMANN, T.; ZHU, Y. Algorithms and applications. In: ELOMAA, T.; MANNILA, H.; ORPONEN, P. (Ed.). Berlin, Heidelberg: Springer-Verlag, 2010. cap. Clustering the Normalized Compression Distance for Influenza Virus Data, p. 130–146. ISBN 3-642-12475-5, 978-3-642-12475-4. Disponível em: <http://dl.acm.org/citation.cfm?id=2167962.2167971>. Citado na página 32.

JAIN, A. K.; DUBES, R. C. **Algorithms for Clustering Data**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN 0-13-022278-X. Citado na página 27.

JONES, D.; POWELL, A.; BOUGANIS, C.; CHEUNG, P. Y. K. Gpu versus fpga for high productivity computing. In: **Field Programmable Logic and Applications (FPL), 2010 International Conference on**. [S.l.: s.n.], 2010. p. 119–124. ISSN 1946-1488. Citado na página 23.

KECKLER, S.; DALLY, W.; KHAILANY, B.; GARLAND, M.; GLASCO, D. GPUs and the Future of Parallel Computing. **Micro, IEEE**, v. 31, n. 5, p. 7–17, out. 2011. ISSN 0272-1732. Citado na página 23.

KIRK, D. B.; HWU, W.-m. W. **Programming Massively Parallel Processors: A Hands-on Approach**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN 0123814723, 9780123814722. Citado na página 23.

KONONENKO, I.; KUKAR, M. **Machine Learning and Data Mining: Introduction to Principles and Algorithms**. [S.l.]: Horwood Publishing Limited, 2007. ISBN 1904275214, 9781904275213. Citado 3 vezes nas páginas 15, 27 e 28.

LIU, L.; WANG, H.; LIU, X.; JIN, X.; HE, W. B.; WANG, Q. B.; CHEN, Y. GreenCloud: a new architecture for green data center. In: **Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session**. New York, NY, USA: ACM, 2009. p. 29–38. ISBN 978-1-60558-612-0. Citado na página 23.

LIU, Q.; LUK, W. Objective-driven workload allocation in heterogeneous computing systems. **2011 International Conference onField-Programmable Technology (FPT)**, IEEE, v. 0, p. 1–4, 2011. Citado 2 vezes nas páginas 42 e 43.

MA, K.; LI, X.; CHEN, W.; ZHANG, C.; WANG, X. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. **2012 41st International Conference on Parallel Processing**, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 48–57, 2012. ISSN 0190-3918. Citado na página 43.

NESHATPOUR, K.; MALIK, M.; HOMAYOUN, H. Accelerating machine learning kernel in hadoop using fpgas. In: **15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015**. [s.n.], 2015. p. 1151–1154. Disponível em: <http://dx.doi.org/10.1109/CCGrid.2015.165>. Citado 2 vezes nas páginas 42 e 43.

NEWMAN, M. E. J. Fast algorithm for detecting community structure in networks. **Phys. Rev. E**, American Physical Society, v. 69, p. 066133, Jun 2004. Disponível em: <http://link.aps.org/doi/10.1103/PhysRevE.69.066133>. Citado na página 34.

PELLERIN, D.; THIBAULT, S. **Practical Fpga Programming in C**. First. Upper Saddle River, NJ, USA: Prentice Hall Press, 2005. ISBN 0131543180. Citado 3 vezes nas páginas 15, 36 e 37.

SAITOU, N.; NEI, M. The neighbor-joining method: a new method for reconstructing phylogenetic trees. **Molecular biology and evolution**, SMBE, v. 4, n. 4, p. 406–425, 1987. Citado na página 32.

SANCHES, A.; CARDOSO, J.; DELBEM, A. C. B. Identifying Merge-Beneficial Software Kernels for Hardware Implementation. In: IEEE. **Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on**. [S.l.], 2011. p. 74–79. Citado 3 vezes nas páginas 30, 32 e 34.

SOARES, A. H. M. **Algoritmo de estimação de distribuição baseados em árvores filogenéticas**. Tese (Doutorado) — Universidade de São Paulo, http://www.teses.usp.br/teses/disponiveis/55/55134/tde-25032015-111952/pt-br.php, 10 2014. Citado na página 33.

UKIDAVE, Y.; KAELI, D. Analyzing optimization techniques for power efficiency on heterogeneous platforms. **2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum**, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 1040–1049, 2013. Citado na página 42.

WESTON, S.; SPOONER, J.; RACANIèRE, S.; MENCER, O. Rapid computation of value and risk for derivatives portfolios. **Concurrency and Computation: Practice and Experience**, John Wiley and Sons Ltd., Chichester, UK, v. 24, n. 8, p. 880–894, jun. 2012. ISSN 1532-0634. Citado na página 24.

WU, X.; KUMAR, V.; QUINLAN, J. R.; GHOSH, J.; YANG, Q.; MOTODA, H.; MCLACHLAN, G. J.; NG, A.; LIU, B.; YU, P. S.; ZHOU, Z.-H.; STEINBACH, M.; HAND, D. J.; STEINBERG, D. Top 10 algorithms in data mining. **Knowl. Inf. Syst.**, Springer-Verlag New York, Inc., New York, NY, USA, v. 14, n. 1, p. 1–37, dez. 2007. ISSN 0219-1377. Disponível em: <http://dx.doi.org/10.1007/s10115-007-0114-2>. Citado na página 30.

XU, R.; WUNSCH, D. Survey of clustering algorithms. **Neural Networks, IEEE Transactions on**, IEEE, v. 16, n. 3, p. 645–678, maio 2005. ISSN 1045-9227. Disponível em: <http://dx.doi.org/10.1109/tnn.2005.845141>. Citado 3 vezes nas páginas 15, 28 e 29.

XU, Z. Measuring Green IT in Society. **Computer**, IEEE Computer Society, Los Alamitos, CA, USA, v. 45, n. 5, p. 83–85, maio 2012. ISSN 0018-9162. Citado na página 23.