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

THIAGO DADALT SOUTO

Predição de performance em ambientes multi-core com aceleradores compartilhados

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

Orientador: Prof. Dr. Antônio Carlos Beck Filho
Co-orientador: Prof. Dr. Luigi Carro

Porto Alegre
2018

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Souto, Thiago Dadalt
Predição de performance em ambientes multi-core com aceleradores compartilhados / Thiago Dadalt Souto. – 2018.
102 f.
Orientador: Antonio Carlos Schneider Beck Filho.
Co-orientador: Luigi Carro

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2018.

1.Aceleradores compartilhados. 2. Métrica 3.Arquiteturas *multi-core*. 4. Predição de performance I. Beck Filho, Antonio Carlos Schneider, orient. II. Carro, Luigi, coorient. III. Predição de performance em ambientes multi-core com aceleradores compartilhados.

AGRADECIMENTOS

Nesta etapa final da minha pesquisa, gostaria de expressar minha imensa gratidão a todas pessoas envolvidas diretamente ou indiretamente. Meus sinceros agradecimentos pelas conversas, apoio e incentivo ao longo desses 2 anos.

Muito obrigado a minha família, em especial aos meus pais, Luiz Augusto e Cristiani Beatriz, pelo exemplo, educação, amor e apoio, emocional e financeiro, no decorrer de toda minha vida. A minha irmã, Gabriele, exemplo de pesquisadora, e também incentivadora desse trabalho. Amo vocês.

A minha esposa Débora. Obrigado por estar ao meu em todos os momentos desta caminhada, pela compreensão, e incentivo. Teu apoio foi fundamental. Te amo!

Agradeço ao meu orientador, Antônio Carlos Schneider Beck Filho, pela confiança depositada em mim. Seu comprometimento, disponibilidade e compreensão foram essenciais para o desenvolvimento deste trabalho. Na minha opinião, um exemplo de dedicação à pesquisa brasileira, e uma prova do imenso potencial da nossa ciência.

Obrigado a todos meus amigos, pelo incentivo durante a produção deste trabalho. Ao P&D Schneider Electric agradeço à todos pelo tempo de convivência, e aprendizado, e em especial ao Alexandre Lorençato, que deu dicas valiosas para que esse trabalho fosse possível.

Aos colegas de laboratório, pelas frutíferas discussões que tornaram essa dissertação possível, e em especial agradeço imensamente o colega Marcelo Brandalero, pelo interesse, pela ajuda e pelos questionamentos feitos no decorrer deste trabalho, com certeza foram fundamentais nesta pesquisa. Muito obrigado.

Agradeço a Schneider Electric, e em especial ao Alexandre Saccol, por incentivar e viabilizar os meus estudos, através da flexibilidade que me perimitiu frequentar as aulas, reuniões do grupo de pesquisa, e realizar as atividades relacionadas à minha pesquisa.

RESUMO

Dois dos principais fatores do aumento da performance em aplicações *single-thread* – frequência de operação e exploração do paralelismo à nível das instruções – tiveram pouco avanço nos últimos anos devido a restrições de potência. Além disso, a exploração do paralelismo em nível de *threads* é limitada pelas porções intrinsecamente sequenciais das aplicações. Neste contexto, é cada vez mais comum a integração de aceleradores em arquiteturas *multi-core.* Esses sistemas exploram o que há de vantajoso em cada um dos componentes que os compõem: enquanto o arranjo *multi-core* explora o paralelismo em nível de *threads,* aceleradores em *hardware* executam funções com performance e eficiência energética ordens de grandeza maiores que uma possível execução nos *cores* de propósito geral.

No atual estado da arte, poucos trabalhos propuseram métricas que avaliam características outras que não a performance ou energia desses arranjos *multi-core* que compartilham aceleradores em *hardware* entre os elementos de processamento. Os MPSoCs, *Multi-processor System-on-Chips*, popularizaram a integração de aceleradores nessas arquiteturas, mas nenhum estudo foi realizado em termos da aceleração esperada com esses aceleradores, ou sobre o custo-benefício esperado da adição de novos aceleradores nessas arquiteturas. A ausência de métricas que avaliem outras características de arquiteturas *multi-core* heterogêneas pode limitar severamente o seu potencial.

Este trabalho tem por objetivo propor uma nova métrica para a integração de aceleradores compartilhados em arquiteturas *multi-core*, SACL (do inglês, *Shared Accelerator Concurrency Level*), descorrelacionada do paralelismo no nível das threads. Esta métrica avalia o percentual de uma aplicação em que blocos básicos aceleráveis executam simultâneamente em diferentes *threads* ativas no contexto, competindo assim pelo uso do acelerador. A partir disto, o projetista pode usar o valor obtido para prever a aceleração esperada para uma determinada aplicação, e também estabelecer o custo-benefício da adição (ou não) de novos aceleradores no sistema. Essa métrica é independente do acelerador, podendo ser utilizada tanto para aceleradores específicos como reconfiguráveis.

Palavras-chave: Aceleradores compartilhados; Arquiteturas *multi-core*; Métrica; Predição de performance.

Predicting performance in multi-core environment with shared accelerators

ABSTRACT

Two of the major drivers of increased performance in single-thread applications - increase in operation frequency and exploitation of instruction-level parallelism - have had little advances in the last years due to power constraints. In addition, the intrinsic sequential portions of application limit exploitation of thread-level parallelism. In this context, it is increasingly common the integration of hardware accelerator in multi-core architectures. These systems are able to exploit the most advantageous features of each composing component: while the multi-core configuration exploits thread-level parallelism, hardware accelerators execute functions with increased performance and energy efficiency when comparing to execution in a general purpose processors.

In the current state of art, very few works proposed metrics that evaluate characteristics other than performance or energy of these multi-core configurations that share hardware accelerators among processing elements. MPSoCs have popularized the integration of accelerators in these architectures, but there is no study realized in regard of expected speedup with these accelerators, or about the expected cost-benefit of the addition of more accelerators in these architectures. The absence of metrics that evaluate different characteristics of heterogeneous multi-core architectures may severely limit its potential.

The goal of this work is to propose a new metric for the integration of shared hardware accelerators in multi-core architectures, SACL, uncorrelated to the thread-level parallelism (TLP). This metric evaluates the percentual of an application that accelerable basic blocks simultaneously execute in different active threads in the context, thus competing to use the accelerator. With this metric, a designer can use the obtained value to predict the expected speedup for a specific application, and to establish the cost-benefit of adding new accelerators to the system. The proposed metric is independent of the accelerator type, and it can be used with specific or reconfigurable accelerators.

Keywords: Shared accelerators; Multi-core architectures; Metric; Performance prediction.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ALU	Atithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-Set Processor
ARM	Advanced RISC Machine
CGRA	Coarse-Grain Reconfigurable Array
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
EDP	Energy-Delay Product
FPGA	Field Programmable Gate Array
GPU	Graphical Processing Unit
ILP	Intruction-Level Parallelism
InO	In-Order
ISA	Instruction Set Architecture
LLC	Last Level Cache
LUT	Look Up Table
MPSoC	Multi-Processor System-on-Chip
OoO	Out-of-order
OS	Operating System
RISC	Reduced Instruction Set Computer
SoC	System-on-Chip
TDP	Thermal Design Power
TLP	Thread-Level Paralellism
TPU	Tensor Processing Unit
VLSI	Very-Large-Scale Integration

SUMÁRIO

1 INTRODUÇÃO

Nos últimos 20 anos, a performance dos microprocessadores multiplicou-se em mais de 1000 vezes. Esse aumento exponencial foi possível devido a três fatores-chave: possibilidade de chavear os transistores em frequências cada vez maiores, melhorias realizadas nas microarquiteturas dos processadores, e aumento do tamanho e níveis das memórias cache (BORKAR; CHIEN, 2011).

A contínua escalabilidade dos transistores foi estabelecida por Gordon Moore, no que ficou conhecida como a Lei de Moore. O enunciado dessa lei determina que a cada dois anos deveria haver uma redução de 30% nas dimensões dos transistores, e consequente redução de 50% na sua área, mantendo-se constante o campo elétrico em todos os pontos do transistor, de forma a manter-se a confiabilidade de operação. Essa diminuição do transistor foi o que permitiu o aumento da frequência de chaveamento em 40%, a cada geração. Além disso, a Lei de Dennard (DENNARD et al., 1974) estabelece que a densidade de potência permaneceria constante, pois tensão e corrente também eram reduzidas proporcionalmente às dimensões do transistor.

Microarquiteturas cada vez mais complexas foram propostas, aproveitando-se da crescente densidade de transistores. Técnicas como *pipeline*, predição de desvios, execução fora de ordem, superescalaridade, e especulação, aproveitaram a grande capacidade de integração de transistores permitida pela Lei de Moore para obter ganhos em performance cada vez maiores.

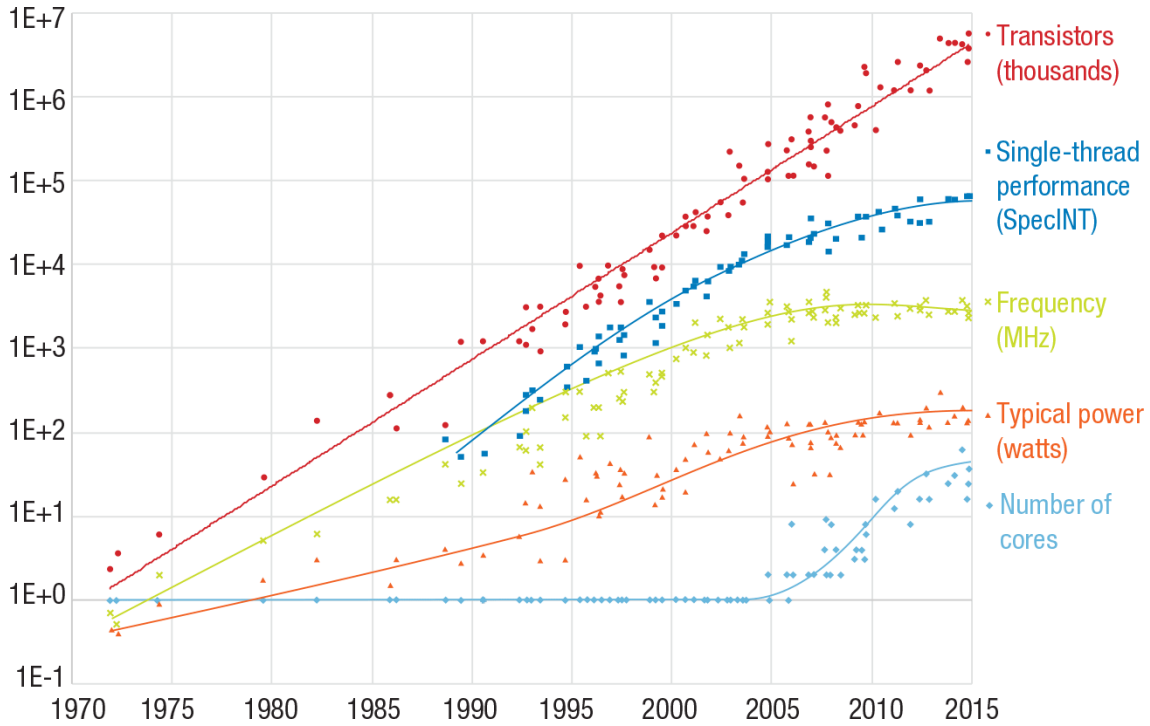
Por fim, o último fator-chave responsável pelos grandes ganhos de performance dos últimos 20 anos foram as memórias cache. As memórias DRAM evoluíram dramaticamente com a Lei de Moore, mas de forma diferente dos transistores. Ao passo que sua performance (tempo de acesso) aumentou gradativamente, sua capacidade dobrou a cada geração. O aumento mais lento no tempo de ciclo das memórias DRAM resultou em um gargalo, que poderia reduzir a performance geral do sistema. Entretanto, maneiras eficientes de se construir uma hierarquia de memórias permitiu que o maior tempo de ciclo de memórias DRAM fosse compensado por memórias *cache*. As memórias *cache* possuem diferentes níveis, e são mais próximas do processador, permitindo tempos de acesso menores que a DRAM, com a contrapartida de serem menores (BORKAR; CHIEN, 2011).

Esses três fatores-chave juntos entregaram um aumento de performance de três ordens de magnitude em 20 anos, mas novos desafios tornaram-se evidentes. A redução da tensão de

operação do transistor chegou a um limite, devido ao exponencial crescimento da corrente de fuga. A partir da tecnologia de 65nm, uma tensão mínima de operação por volta de 1V permaneceu constante, o que manteve a corrente de fuga sob controle, mas com o consequente fim da Lei de Dennard. Além disso, ao passo que a densidade de transistores dobrava a cada geração, o consumo de potência de cada dispositivo não reduzia na mesma proporção, resultando em um aumento na densidade de potência dissipada. A combinação de aumento exponencial da corrente de fuga, com a impossibilidade de redução da tensão de operação dos transistores, levaram a uma "barreira de potência": a frequência de operação encontrou um limite em 4 GHz.

A Figura 1.1 mostra que o desempenho *single-thread* dos processadores passou a crescer em taxas muito menores, devido ao crescimento exponencial da potência térmica de projeto (TDP, do inglês *Thermal Design Power*). Os ganhos de performance não poderiam mais ser obtidos através do aumento da frequência de chaveamento, nem através do projeto de complexas microarquiteturas. Projetistas passaram a valer-se apenas da capacidade de integração dos transistores para aumentar a performance dos processadores.

Figura 1.1 – 40 anos de tendências em processadores, em termos de número de transistores, performance *single-thread*, frequência de chaveamento, potência dissipada e número de *cores*



Fonte: ("40 Years of Microprocessor Trend Data")

1.1 A Era Multi-core

As restrições relativas ao aumento da frequência dos transistores, e ao projeto de estruturas microarquiteturais complexas, juntamente com demandas por performances maiores levaram arquitetos de processadores a desenvolverem os primeiros processadores *multi-core*: uma solução que integra múltiplos elementos de processamento com memória *cache* compartilhada em um mesmo componente (BORKAR et al., 2005). Juntamente com a exploração do ILP, processadores *multi-core*, utilizando-se de técnicas de programação paralela, passaram a explorar paralelismo em nível de *threads*, uma técnica que particiona uma aplicação em múltiplas *threads*, que são executadas simultaneamente nos múltiplos *cores* de um processador *multi-core*.

Explorar o paralelismo em nível de *threads* em aplicações típicas requer que o programador especifique como o problema é particionado entre as *threads* e defina exatamente a comunicação necessária entre elas – uma tarefa que é frequentemente não-trivial (BLAKE et al., 2010). O desempenho das aplicações *multithread* possui influência direta dos programadores, e como mostrado por (FATEHI; GRATZ, 2014), requer um balanceamento adequado das cargas de trabalho entre os diversos elementos de processamento do processador, pois, caso contrário, o custo da troca de contexto, intrínseco a arquiteturas *multi-core*, levará a retornos cada vez menores, conforme o número de *cores* aumenta.

A seção 2.1 desse trabalho é dedicada a explorar as arquiteturas *multi-core* em maiores detalhes.

1.2 Aceleração em Hardware

Apesar da crescente utilização de processadores *multi-core* para as mais diferentes aplicações, essas arquiteturas não afetam o desempenho de aplicações *single-thread*, ou de aplicações que, devido a sua natureza, não mostram ganhos significativos quando paralelizadas. Desta forma, surgiu a necessidade de criarem-se novas soluções para otimizar as aplicações que não se beneficiam de múltiplas *threads*.

Uma das formas encontradas é a aceleração em *hardware*. Em geral, esta técnica aumenta a performance através de unidades computacionais customizadas, barramentos de dados especializados para a movimentação de dados, e sequências de instruções com redução de custos (BORKAR; CHIEN, 2011). Um exemplo de acelerador em *hardware* utilizado em

larga escala são os TPUs (do inglês, *Tensor Processing Unit*), proposto por (JOUPPI et al., 2017). O TPU foi uma resposta da Google à crescente demanda de seus usuários por aplicativos de reconhecimento de voz, que baseados em redes neurais profundas (DNN, do inglês *Deep Neural Networks*), faziam extenso uso de multiplicação de matrizes. Esse acelerador foi incluído nos *datacenters* do Google em 2015, e executa DNNs de 15-30 vezes mais rápido, com 30-80 vezes maior eficiência energética, do que CPUs e GPUs contemporâneas em tecnologias de semicondutores similares.

A seção 2.2 desse trabalho é dedicado a explorar diferentes tipos de aceleradores em hardware, possíveis configurações, e princípios básicos.

1.3 MPSoC

Aceleradores em *hardware* passaram ainda a ser integrados a arquiteturas *multi-core*, juntamente com outros tipos de processadores programáveis, levando ao desenvolvimento de arquiteturas especializadas, os MPSoCs (do inglês, *Multiprocessor System-on-Chip*).

Essas arquiteturas caracterizam-se pela alta performance e alta eficiência energética, criando um ambiente de processamento altamente heterogêneo e eficiente. Esses sistemas exploram o que há de vantajoso em cada um dos componentes que os integram: enquanto o arranjo *multi-core* explora o paralelismo em nível de *threads*, aceleradores em *hardware* executam funções com performance e eficiência energética ordens de grandeza maiores que uma possível execução nos *cores* de propósito geral. Outros processadores programáveis, como DSPs e GPUs, executam aplicações propícias a essas arquiteturas, como processamento de sinais, e multimídia.

A inclusão de aceleradores em *hardware* em arquiteturas *multi-core* traz consigo uma série de desafios, como por exemplo, determinar a posição do acelerador relativa aos *cores*, a comunicação entre acelerador e *cores*, custo-benefício em termos de performance e área, alocação desses recursos. Além disso, como veremos no Capítulo 3, alguns trabalhos exploram o compartilhamento desses aceleradores entre os *cores*, propondo que essas arquiteturas podem atingir uma melhor utilização dos recursos, e consequentemente, um maior ganho de performance na execução de aplicações.

1.4 Objetivos da dissertação

Essa dissertação é uma análise do aumento de performance da execução de aplicações *multithread*, em arquiteturas *multi-core* heterogêneas com aceleradores compartilhados. Através de um simulador *cycle-accurate*, um *software* capaz de simular microarquiteturas de processadores ciclo-a-ciclo, oferecemos um meio para projetistas incluírem aceleradores em ambientes *multi-core*, indiretamente dando indícios de custo de área, energia e desempenho. Estabelecendo uma métrica que possa prever o ganho de performance de aplicações *multithread* executando nessas arquiteturas, podemos mostrar que ainda existe uma grande oportunidade de ganho de performance no compartilhamento de aceleradores entre elementos de processamento.

Os objetivos podem ser elencados da seguinte forma:

1. Proporemos uma métrica, descorrelacionada do paralelismo à nível de *threads*, que pode prever a aceleração potencial de uma aplicação *multithread,* em ambientes *multi-core*, com o compartilhamento de aceleradores em *hardware*;
2. Utilizando-se processadores *multi-core* com aceleradores reconfiguráveis compartilhados como estudo de caso, exploraremos o espaço de projeto disponível nessas arquiteturas, avaliando, através de um conjunto de *benchmarks*, a aceleração de diversas aplicações, e com diferentes arranjos de processador/aceleradores. Até o melhor do nosso conhecimento, nenhum trabalho explorou até o momento o compartilhamento de aceleradores de grão-grosso reconfiguráveis em arquiteturas *multi-core*;
3. Mostraremos que existe uma alta correlação entre o que definimos como oportunidade de aceleração de uma aplicação, e a métrica proposta. Isso significa que estabeleceremos um meio, em tempo de projeto, de o projetista definir o custo-benefício em termos de performance e área da inclusão de aceleradores no sistema;
4. Mostraremos que nossa métrica pode ser estendida para prever a performance não apenas de aceleradores reconfiguráveis, mas também de aceleradores específicos.

1.5 Estrutura da dissertação

Essa dissertação está organizada da seguinte forma: No Capítulo 2, nós apresentamos para o leitor o conhecimento-base utilizado nesse trabalho, que incluem as arquiteturas *multi-core*, a aceleração em hardware, e os MPSoCs. No Capítulo 3, discutimos alguns dos principais trabalhos relacionados ao tema dessa dissertação. Capítulo 4 apresenta o racional da métrica proposta, e sua definição matemática. Capítulo 5 introduz a metodologia utilizada para chegar ao modelo para integração de aceleradores em *hardware* em sistemas *multi-core*. Capítulo 6 discute os resultados obtidos, e o Capítulo 7 encerra esse trabalho com conclusões e trabalhos que podem ser realizados futuramente sobre o modelo proposto.

2 ARQUITETURAS MULTI-CORE E ACELERAÇÃO EM HARDWARE

No Capítulo 1 mostramos a evolução dos processadores nos últimos 30 anos, e o racional para cada uma das mudanças de paradigma que ocorreram. Nesse capítulo iremos aprofundar conceitos vistos no Capítulo 1, e que são centrais para essa dissertação: arquiteturas *multi-core*, aceleração em *hardware*, e arquiteturas MPSoC. Primeiramente, iremos apresentar a classificação mais utilizada para arquiteturas *multi-cores*, que se refere aos elementos de processamento de um processador *multi-core*, mais especificamente, à sua arquitetura (ISA) e à sua microarquitetura, também chamada organização do processador. Ao longo desse trabalho, utilizaremos ambos os termos de forma intercambiável. Posteriormente, iremos expandir o conceito de aceleração em *hardware*, e veremos também classificações comuns, e princípios básicos dessa técnica. Por fim, iremos abordar uma arquitetura que tem sido muito utilizada, e engloba conceitos de arquitetura multi-core e aceleração em hardware, chamada MPSoCs.

2.1 Arquiteturas Multi-core

As arquiteturas *multi-core* foram, como vimos no Capítulo 1, uma resposta ao fim da Lei de Dennard. A potência de pico dissipada por mm² de chip continuava crescendo, até que um limite de dissipação de potência por *chip* foi atingido. Isso fez com que projetistas passassem a propor arquiteturas com mais de um elemento de processamento em um mesmo *chip*: as arquiteturas *multi-core*.

A principal vantagem dessas arquiteturas é o aumento de performance depender do número de *cores*, e não do aumento da frequência, o que se traduz em um crescimento menor no consumo de potência (BLAKE et al., 2009). Entretanto, sabe-se que a programação paralela não é uma técnica trivial, e a extração de paralelismo não é automática como nos processadores superescalares.

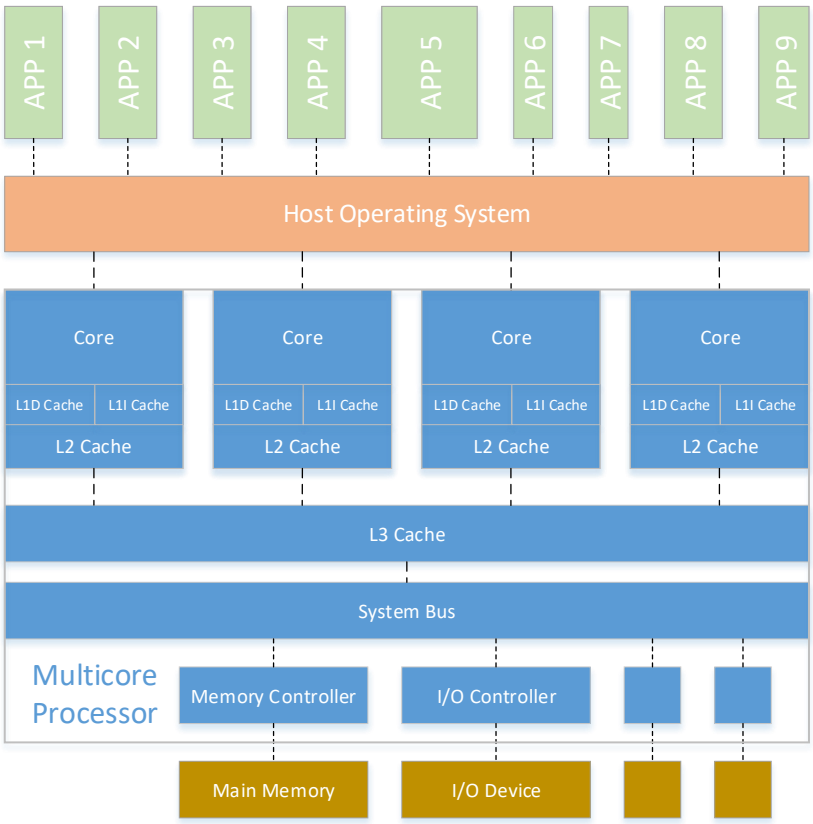
Muitos trabalhos foram realizados no sentido de explorar a melhor organização dos *cores* no processador. Processadores *multi-core* podem ser classificados como homogêneos e heterogêneos, e nas próximas seções iremos abordar as diferenças que existem entre essas arquiteturas, o contexto no qual foram propostas, e suas principais características.

2.1.1 Arquiteturas Multi-core Homogêneas

Em processadores *multi-core* homogêneos, a ISA de cada core é a mesma ISA do processador de um *core* único correspondente, apenas com algumas modificações para o suporte de paralelismo (BLAKE et al., 2009). Há poucos anos, conforme (FATEHI; GRATZ, 2014), essa era a única abordagem da indústria para o projeto de arquiteturas *multi-core*, e ainda é o tipo de sistema dominante nos mercados de computadores pessoais, *notebooks* e servidores (*mainframes*). Essas arquiteturas são comumente compostas de processadores superescalares (HENESSY; DAVID A. PATTERSON, 2011), o que permite que o paralelismo de instruções também possa ser ser apropriadamente explorado.

Quando um processador *multi-core* é composto por elementos de processamento que possuem a mesma arquitetura e microarquitetura, diz-se que os processadores apresentam arquitetura multi-core homogênea, como a ilustrada na Figura 2.1.

Figura 2.1 – Arquitetura multi-core homogênea genérica



Fonte: O autor

Na Figura 2.1 temos um processador *multi-core* homogêneo, com quatro elementos de processamento. Cada *core* possui memórias *cache* L1 de dados e instrução, e uma memória *cache L2*, privadas. A comunicação entre os elementos de processamento ocorre através da memória *cache* compartilhada L3. Ainda, cada uma das nove aplicações sendo executadas no processador é alocada pelo sistema operacional para cada um dos elementos de processamento. Se alguma dessas aplicações foi programada valendo-se de paradigmas de programação paralela, também é responsabilidade do sistema operacional alocar as diferentes *threads* nos *cores*.

Arquiteturas multi-core homogêneas são bastante adequadas para a exploração de TLP, mas não apresentam o mesmo potencial quando é necessária a execução de aplicações intrinsecamente sequenciais (HILL; MARTY, 2008). Isto pode ser constatado através da Lei de Amdahl (AMDAHL, 1967). Essencialmente, a Lei de Amdahl diz que o potencial ganho de aceleração na execução de um programa, devido a paralelização, não pode ser maior que o inverso da porção do programa que é instrinsicamente sequencial. Formalmente:

$$S(n) = \frac{1}{B + \frac{1}{n}(1 - B)}$$

Equação 2.1: Lei de Amdahl

Na Equação 2.1, $S(n)$ é o ganho de performance teórico na execução de um algoritmo utilizando-se n threads. B é a porção do algoritmo estritamente serial (não-paralelizável).

Muitos problemas clássicos resolvidos por computador tendem a ter limitação em relação a quantidade de paralelismo que pode ser efetivamente explorado, na forma de *threads* ou outras construções paralelas, limitando o potencial de performance que pode ser obtido apenas através da replicação de *cores* iguais em um processador. Arquiteturas que focam na exploração de TLP, acabam sacrificando performance para aplicações que possuem grandes partes sequenciais (PERICAS et al., 2007).

2.1.2 Arquiteturas Multi-core Heterogêneas em Organização e Homogêneas em ISA

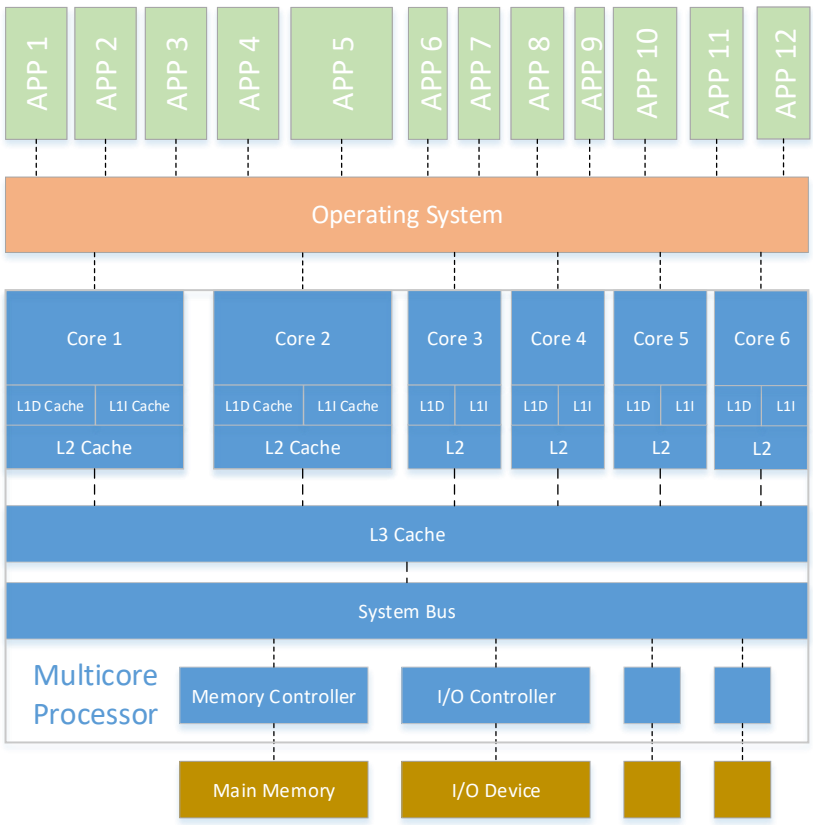
Sistemas homogêneos são ineficientes, pois são focados apenas em redução de custos e de complexidade de projeto, e que é insuficiente para aplicações *multithread* emergentes. Esses sistemas são ineficientes quando processando aplicações com comportamento heterogêneo

(SOUZA, 2016). Devido ao fato de os elementos de processamento serem iguais, cada um explora os mesmos níveis de paralelismo em nível de instrução, o que pode levar a desperdício de recursos em uma grande quantidade de aplicações.

Nesse contexto, novas arquiteturas capazes de explorar o TLP como os processadores multi-core homogêneos, mas que também fossem energeticamente mais eficientes, passaram a ser desenvolvidas.

Uma das primeiras arquiteturas heterogêneas foi proposta por (KUMAR et al., 2003). Nesse trabalho, foi proposta uma arquitetura de processadores multi-core heterogêneos (em microarquitetura), mas homogêneos em arquitetura (mesma ISA), como o da Figura 2.2, na tentativa de melhorar a eficiência de sistemas multi-core processando aplicações altamente heterogêneas. Nesse trabalho, os autores replicaram diferentes tamanhos de um processador Alpha, e as *threads* são alocadas de acordo com suas necessidades. Ficou evidente o grande potencial de sistemas heterogêneos para economia de energia, utilizando elementos de processamento mais simples quando a demanda por performance da aplicação é pequena, em vez de utilizar *cores* complexos, em geral superescalares.

Figura 2.2 – Arquitetura multi-core heterogênea genérica



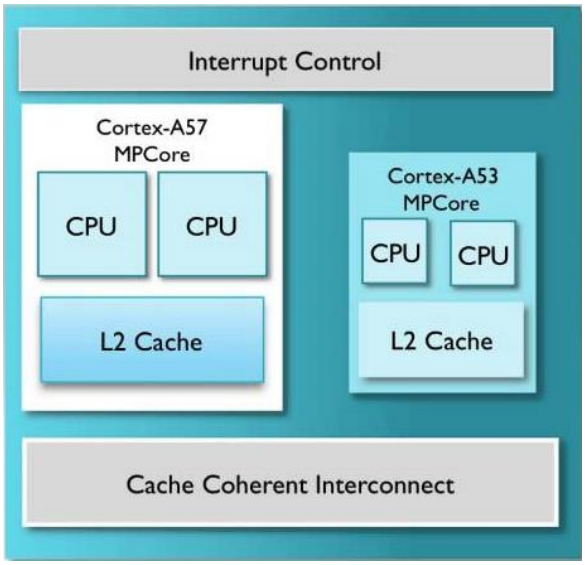
Fonte: O autor

Nesse caso, diferentemente da Figura 2.1, os elementos de processamento são iguais em relação à arquitetura, mas diferentes em relação à microarquitetura, caracterizando a heterogeneidade do processador (KUMAR et al., 2003). Em outras palavras, um sistema é dito heterogêneo quando os elementos de processamento não são todos iguais entre si. O sistema operacional, nesse caso, é responsável por gerenciar as aplicações e *threads* em *cores* diferentes. Ou seja, o sistema operacional é responsável por eficientemente explorar as diferentes microarquiteturas, a depender da aplicação a ser executada.

A heterogeneidade também pode ser explorada em tempo de execução, como proposto por (SRINIVASAN et al., 2013). Nesse trabalho, os autores propuseram um processador multi-core superescalar com execução fora de ordem (OoO), no qual os elementos de processamento são capazes de modificarem-se para *cores* menores, com execução em ordem (InO), a depender de mudanças nos requisitos de potência e performance. Quando uma *thread* exibe um alto ILP a ser explorado, existe ganho de performance ao se utilizar o processador superescalar para executá-la. Entretanto, quando o processador está parado devido a alguma dependência (dados ou controle), ou aguardando uma operação de alta latência (acesso à memória principal), esse processador está desperdiçando potência estática. Para reduzir o consumo de potência nessas fases, alguns blocos do *core* podem ser desligados em tempo de execução, além de reduzir o tamanho das estruturas de decodificação das instruções. A vantagem dessa arquitetura é evitar o *overhead* criado pela troca de *threads* entre os *cores*, e reduzir o consumo de energia. Entretanto, quando o processador está operando no modo InO, muitos recursos são desligados, desperdiçando área do chip.

A tecnologia big.LITTLE (“big.LITTLE Technology, 2016”) da ARM é um exemplo de uma arquitetura *multi-core* heterogênea, largamente utilizada em uma grande quantidade de processadores embarcados, e que de fato popularizou os processadores heterogêneos. Essa arquitetura consiste da combinação de elementos de processamento de alta performance, como o Cortex-A57, com elementos de processamento de alta eficiência energética, como o Cortex-A53. Essa combinação cria um ambiente heterogêneo, em que a aplicação pode ser executada pelos *cores* que mais se adequam às suas necessidades. A Figura 2.3 mostra o diagrama de blocos de um processador ARM big.LITTLE.

Figura 2.3 – Diagrama de blocos de um processador ARM Cortex-A9, que utiliza elementos de processamento heterogêneos para o processamento de aplicações que requerem alta eficiência energética



Fonte: (“big.LITTLE Technology”, 2016)

Esse sistema é composto de dois conjuntos distintos de *cores* superescalares de mesma ISA, capazes de explorar diferentes níveis de paralelismo em nível de instrução, e cada um com diferentes requisitos de potência. Um escalonador *online* é responsável por alternar entre os elementos de processamento de acordo com as necessidades da aplicação, reduzindo drasticamente o consumo de energia quando as aplicações entram em um estágio de baixa demanda. A grande vantagem dessa arquitetura é a capacidade de manter a compatibilidade de *software*, pois ambos conjuntos de processadores possuem a mesma ISA. O processador ARM Cortex-A9, que emprega essa organização, é na verdade um MPSoC, um sistema com arquitetura altamente heterogênea, que veremos em maior detalhe na seção 2.3.

Algumas características dos sistemas heterogênos justificam sua crescente utilização. Primeiramente, um processador multi-core heterogêneo pode alocar a aplicação para o *core* que melhor se adequa às suas características. Em segundo lugar, um processador heterogêneo pode cobrir melhor as demandas das cargas de trabalho de uma determinada aplicação. Por exemplo, aplicações com baixo TLP e alto ILP podem ser alocadas nos *cores* maiores, superescalares com execução OoO; já aplicações com alto TLP podem ser mais eficientemente executados em um número maior de *cores* mais simples. Entretanto, para aplicações *single-thread*, e aplicações com grandes porções intrinsecamente sequenciais, as arquiteturas heterogêneas compartilham a mesma limitação das arquiteturas homogêneas: o potencial de ganho de performance das aplicações é limitado pela Lei de Amadahl.

2.2 Aceleração em Hardware

As arquiteturas multi-core foram capazes de manter os ganhos de performance dos processadores, mesmo que em um nível menor, após o fim da Lei de Dennard. Entretanto, como visto nas seções 2.1.1 e 2.1.2, esses ganhos estão limitados às porções paralelizáveis das aplicações. Uma técnica que passou a ser utilizada para aumentar a performance de execução de aplicações intrinsecamente sequenciais é a aceleração em *hardware*. Aceleradores em *hardware* são porções de *hardware* dedicadas, projetadas para a execução de operações específicas. Utilizando-se implementações em *hardware* para execução, o custo intrínseco das arquiteturas *multi-core* de propósito geral é evitado, o que resulta em ganho de performance e eficiência energética (HOU et al., 2011).

A funcionalidade implementada por um determinado acelerador pode ser realizada tanto através de componentes de *hardware* específicos para uma determinada aplicação, como em (JOUPPI et al., 2017), quanto por *hardwares* dinamicamente customizáveis, conhecidos como aceleradores reconfiguráveis. Esses aceleradores podem ser baseados em FPGAs como em (VAHID; STITT; LYSECKY, 2008), ou em outras estruturas como em (RUTZIG; BECK; CARRO, 2013).

Alguns exemplos de periféricos/aceleradores comumente integrados ao processador incluem rasterizadores gráficos, aceleradores de codecs (áudio, imagem e vídeo), controladores de memória, e etc. Tais componentes podem ter um grande impacto na performance geral do sistema. Nas seções 2.2.1 e 2.2.2 iremos discutir os tipos de aceleradores integrados à processadores: aceleradores específicos e aceleradores reconfiguráveis.

2.2.1 Aceleradores específicos

Aceleradores específicos são arquiteturas projetadas para executar funções específicas. A funcionalidade implementada pode abranger tanto tarefas simples (como operações de multiplicar-acumular), tarefas de complexidade moderada (por exemplo, FFT ou DCT), até tarefas altamente complexas, como a implementação de algoritmos de encriptação/desencriptação e algoritmos de codificação/decodificação de vídeo. Os TPUs, abordados no Capítulo 1, são aceleradores específicos para a execução de DNNs, e um exemplo de ganho de performance que esses componentes de *hardware* podem prover.

Aceleradores dessa classe são comumente encontrados na forma de ASICs (do inglês, *Application Specific Integrated-Circuits*) e ASIPs (do inglês, *Application Specific Intruction-Set Processors*).

2.2.2 Aceleradores reconfiguráveis

Aceleradores reconfiguráveis apresentam arquiteturas que podem se adaptar para implementar em *hardware* diferentes funções, e diversos trabalhos mostram, como (HARTENSTEIN, 2001) e (TODMAN et al., 2005), que essa técnica é capaz de aumentar a performance e eficiência de processadores, em significativas ordens de grandeza. Nessa seção, detalharemos esses aceleradores, apresentando os princípios básicos, a granularidade possível para o componente reconfigurável, e as formas de acomplamento do acelerador ao processador. Isso porque, posteriormente, utilizaremos um acelerador reconfigurável como estudo de caso, para avaliação da métrica proposta.

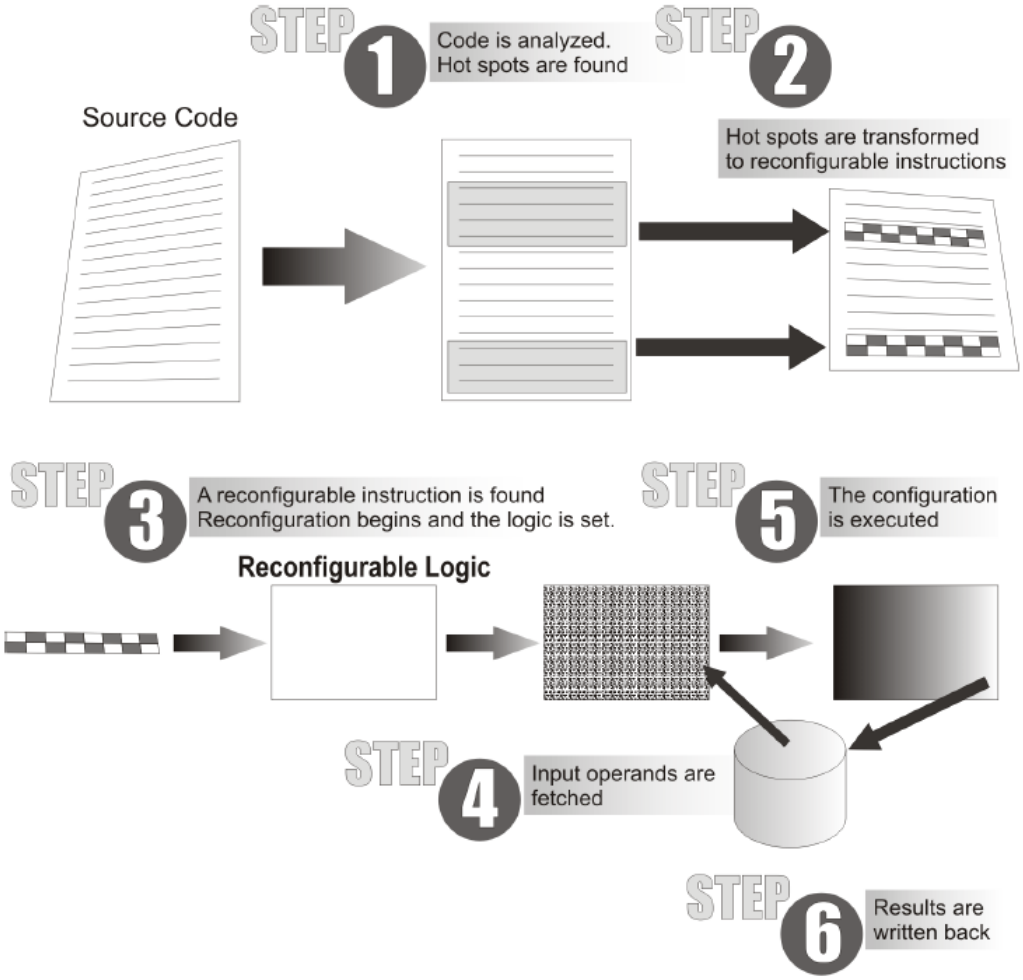
Diferentemente dos aceleradores específicos (seção 2.2.1), além desses sistemas oferecerem ganhos em performance e em eficiência energética de execução, também são flexíveis, podendo ser utilizados para acelerar uma grande gama de aplicações. Entretanto, como mostrado por (BECK; LANG LISBÔA; CARRO, 2013), os ganhos são menores, quando comparados com circuitos dedicados, como ASIPs e ASICs.

2.2.2.1 Princípios Básicos

Sistemas reconfiguráveis são comumente compostos de uma lógica reconfigurável, um controlador, responsável por gerenciar as reconfigurações e as comunicações, e uma memória de contexto, que armazena as configurações para a lógica reconfigurável (SOUZA, 2016). Os blocos do sistema reconfigurável são acoplados ao processador de propósito geral, que executa as regiões do software que não podem ser aceleradas, enquanto a lógica reconfigurável executa as outras regiões, chamadas *kernels*. Uma desvantagem desses sistemas é que os blocos que os compõem podem aumentar bastante a área total utilizada pelo processador.

A Figura 2.4 mostra os seis passos comumente implementados por sistemas reconfiguráveis.

Figura 2.4 - Passos básicos seguidos por um acelerador reconfigurável para execução em hardware de determinados trechos de código



Fonte: (BECK; LANG LISBÔA; CARRO, 2013)

1) Análise do Código: Em um primeiro momento, o código-fonte é analisado de forma a identificar os *kernels* da aplicação (trechos da aplicação que podem ser transformados para execução na lógica programável). Essa análise é normalmente realizada em um traço de código gerado durante execução anterior.
2) Transformação do Código: Após a identificação dos trechos de código que podem ser executados na lógica reconfigurável, as instruções de código dessas regiões são substituídas por instruções reconfiguráveis. Esse processo é realizado pelo controlador do sistema reconfigurável.
3) Reconfiguração: Durante a execução do programa que tem instruções reconfiguráveis, quando uma nova configuração está pronta para ser executada na lógica reconfigurável, as unidades programáveis da lógica são reorganizadas para executar o *kernel*. Um

conjunto de bits de configuração, chamados contexto de configuração, são carregados de uma memória especial: a memória de reconfiguração. O tempo necessário para carregar a configuração da memória, e configurar a lógica reconfigurável é chamado de tempo de reconfiguração.

4) Carregar contexto de entrada: Para executar uma dada instrução reconfigurável, um conjunto de entradas é necessário. Esses dados podem ser obtidos de um arquivo de registradores, memória compartilhada, ou através de um protocolo de comunicação implementado para um barramento;
5) Execução: Após a configuração das unidades programáveis da lógica reprogramável, a execução da instrução é iniciada. Essa execução é mais eficiente que no processador, pois agora existe um circuito especializado que executa em *hardware* a instrução.
6) *Write-Back*: Os resultados da execução da lógica reconfigurável são escritos em um registrador ou em uma memória compartilhada.

Os passos 3 a 6 são repetidos enquanto as instruções reconfiguráveis são encontradas no código, até o final da aplicação.

2.2.2.2 Granularidade

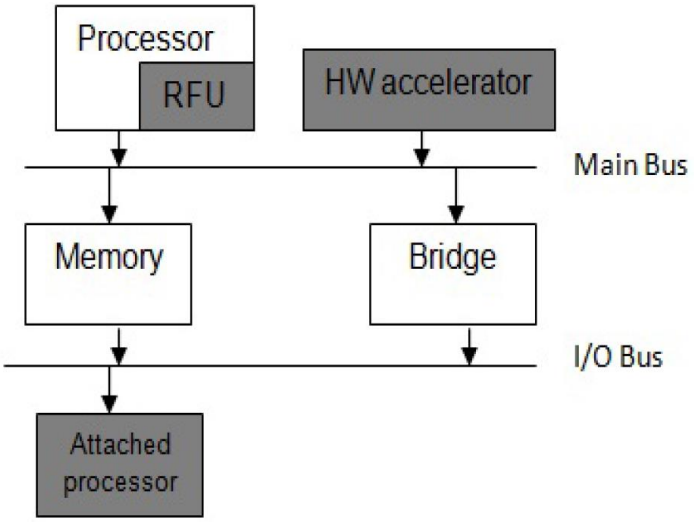
A granularidade de uma lógica reconfigurável define o nível de manipulação de dados da unidade reconfigurável. Para lógicas de grão-fino, os menores blocos que podem ser programados são geralmente portas lógicas (e.g. LUTs de FPGAs). Essa granularidade é mais eficiente para operações em nível de bit. De acordo com (SOUZA et al., 2016), arquiteturas de grão-fino podem levar a maiores níveis de aceleração, mas sua aplicação em geral é limitada a aplicações que possuem poucos kernel que são repetidos em uma grande porção do código. Isso deve-se ao fato de o tempo de reconfiguração dessas arquiteturas ser muito grande, inviabilizando reconfigurações frequentes.

Por outro lado, lógicas reconfiguráveis de grão-grosso possuem blocos configuráveis maiores (por exemplo, ALUs), sendo mais adequados para operações paralelas de bits, como *bytes* e *words*. Essas arquiteturas são capazes de acelerar aplicações inteiras pois possuem um reduzido tempo de configuração, quando comparado com as lógicas de grão-fino.

2.2.2.3 Acoplamento do acelerador ao processador

A posição da lógica reconfigurável relativa ao processador afeta diretamente a performance do sistema e o tipo de aplicações que podem se beneficiar do hardware reconfigurável. A lógica reconfigurável pode ser colocada em três posições relativas ao processador, conforme mostra a Figura 2.5.

Figura 2.5 – Formas de acoplamento possíveis de uma lógica reconfigurável a um processador principal



Fonte: (BARAT; LAUWEREINS; DECONINCK, 2002)

- Processador anexado: a lógica reconfigurável é colocada em um barramento de entrada/saída (por exemplo, barramento PCI);
- Co-processador: a lógica reconfigurável é colocada próxima ao processador. A comunicação é realizada utilizando-se um protocolo similar àqueles utilizados por co-processadores;
- Unidade funcional reconfigurável (RFU): a lógica é colocada dentro do processador. O decodificador de instruções envia as instruções para a unidade reconfigurável como se fosse uma unidade funcional padrão.

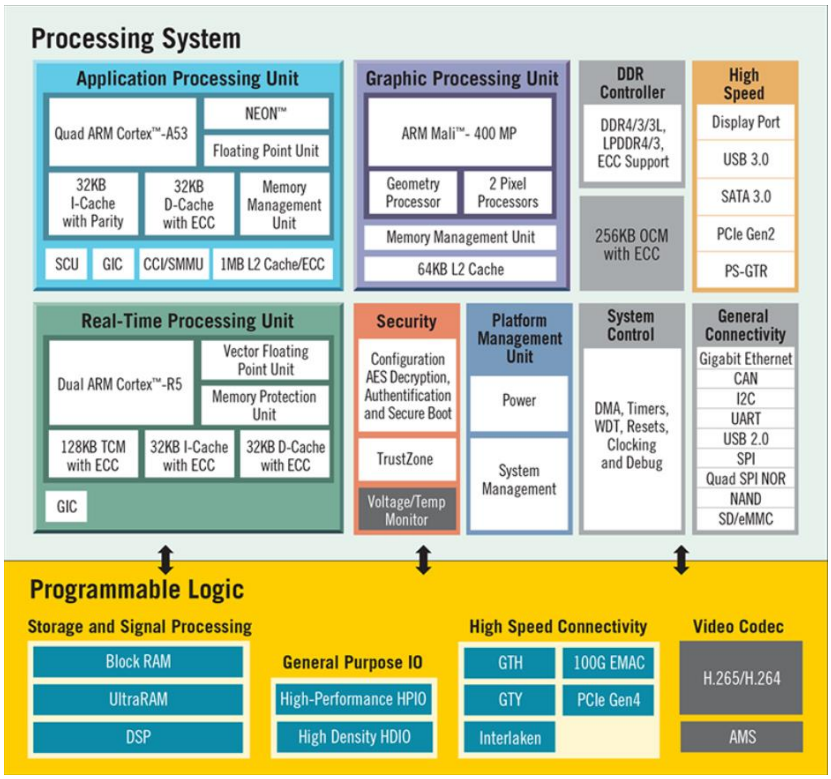
Em relação aos dois primeiros esquemas de acoplamento, chamados fracamente acoplados (do inglês *loosely-coupled*), a melhoria de performance tem que compensar o custo necessário para a transferência de dados.

Em relação ao esquema de unidade funcional reconfigurável integrada, chamado fortemente acoplado (do inglês *tightly-coupled*), os custos de comunicação são praticamente não-existentes, e é mais fácil obter um aumento de performance em uma maior gama de aplicações. Entretanto, para esse esquema, os custos de projeto são muito maiores (BARAT; LAUWEREINS; DECONINCK, 2002).

2.3 MPSoC

Multiprocessadores SoCs são uma importante classe de sistemas VLSI (*Very Large Scale Integration*). É um sistema que incorpora a maior parte dos componentes necessários para uma aplicação (WOLF; JERRAYA; MARTIN, 2008), e utiliza múltiplos processadores programáveis como componentes de sistema. Esses sistemas podem ser compostos por arquiteturas *multi-core* homogêneas (seção 2.1.1) ou arquiteturas *multi-core* heterogêneas 2.1.2, com um ou mais aceleradores integrados ao mesmo *chip*. Um MPSoC é ilustrado na Figura 2.6.

Figura 2.6 – Arquitetura MPSoC. Detalhe nos diferentes tipos de processadores programáveis integrados em um mesmo *System-on-Chip*



Fonte: (“Zynq UltraScale+ Product Brief”, 2016)

Como vemos na Figura 2.6, o MPSoC comercial Zynq UltraScale+ é composto de um processador de propósito geral Quad ARM Cortex-A53, que possui quatro *cores*. Além disso, ainda são integrados ao mesmo *chip* outros processadores programáveis, como uma GPU, para processamento gráfico, e uma unidade de processamento de tempo real. Aceleradores específicos para critptografia e codificação/decodificação também são incluídos, além de uma lógica programável (FPGA).

Os MPSoCs são principalmente utilizados em aplicações altamente heterogêneas, em que a flexibilidade dos diferentes arranjos de elementos de processamento podem traduzir-se em performance e eficiência. Entretanto, o fato dos MPSoC serem heterogêneos não somente em organização (microarquitetura), mas também em arquitetura, significa que cada unidade de processamento tem seu próprio conjunto de intruções (ISA). Diferentes conjuntos de instruções para cada elemento de processamento implicam em cada um possuir uma diferente interface entre *hardware*/*software*. Isso traz consigo as seguintes consequências: ou o programador responsável pelo *software* gerencia manualmente o uso de cada um dos aceleradores; ou ferramentas especializadas e compiladores devem ser distribuídos pelo fabricante, o que aumenta significativamente o tempo para novos projetos irem ao mercado.

2.4 Aceleradores compartilhados em ambientes multi-core

Essa dissertação propõem o estudo do espaço de projeto de aceleradores compartilhados em ambientes *multi-core*, e uma métrica para a predição de performance desses sistemas

Para a exploração do espaço de projeto utilizaremos, conforme visto na seção 2.1, uma arquitetura *multi-core* homogênea, com oito elementos de processamento. Os elementos de processamento dessa arquitetura compartilham aceleradores reconfiguráveis (seção 2.2.2), mais especificamente, o CGRA, visto em maior detalhe na seção 3.2. O trabalho de (BRANDALERO; BECK, 2017) foi inicialmente proposto para processadores *single-core*. Nessa dissertação, como estudo de caso para validação da métrica proposta, utilizaremos o CGRA em um processador *multi-core*, em que o acelerador reconfigurável pode ser compartilhado entre os elementos de processamento. O sistema completo, processador *multi-core* com aceleradores integrados, caracteriza um MPSoC (seção 2.3).

Utilizamos esse sistema como um estudo de caso, de forma a avaliar a métrica proposta no Capítulo 4. Maiores detalhes sobre o CGRA e o processador *multi-core* utilizado nesse estudo de caso são fornecidos no Capítulo 5.

3 TRABALHOS RELACIONADOS

Nesse capítulo, iremos apresentar e discutir trabalhos na literatura relacionados a métricas utilizadas para avaliar processadores multi-core heterogêneos, e a arquiteturas *multi-core* heterogêneas com aceleradores em hardware compartilhados.

Na primeira parte desse capítulo iremos mostrar que existem poucas métricas disponíveis, além de métricas de performance e eficiência-energética, para a avaliação de processadores *multi-core*. Especialmente para o caso de arquiteturas *multi-core* que integram aceleradores. Na segunda parte desse capítulo iremos apresentar uma breve visão geral do que existe na literatura relativo a aceleradores reconfiguráveis. Na terceira, e última parte desse capítulo, iremos abordar as arquiteturas *multi-core* que integram aceleradores, de propósito específico ou geral, compartilhados entre os elementos de processamento.

3.1 Métricas utilizadas para avaliar processadores multi-core

Nessa seção iremos revisar métricas propostas na literatura para avaliação de processadores multi-core. As métricas foram divididas em quatro seções: Métricas de paralelismo, métricas de performance, métricas de eficiência-energética, e métricas para processadores heterogêneos, em que discutimos principalmente o trabalho de (TOMUSK, 2016).

3.1.1 Métricas de paralelismo

(FLAUTNER et al., 2000) propôs uma métrica, capaz de descrever com acurácia o fator de utilização dos recursos paralelos para um processador *multi-core*: a TLP (*Thread-Level Parallelism*). De acordo com (BLAKE et al., 2010), essa métrica é uma indicação da eficiência de utilização dos recursos em um sistema multi-core.

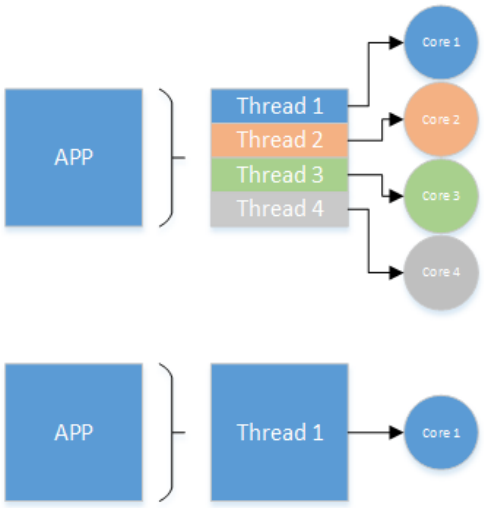
A TLP é calculada somando-se as frações de tempo ($c_i s$) em que exatamente $i = 0, ..., n$ (em que n é o número de *threads* no contexto) *threads* estão executando concorrentemente. Esse número é então dividido pela fração de tempo em que as *threads* permaneceram ativas, para se obter a TLP. Formalmente, o cálculo da TLP é realizado através da Equação 3.1.

$$TLP = \frac{\sum_{i=1}^{n} c_i i}{1 - c_0}$$

Equação 3.1: Fórmula utilizada para o cálculo da TLP de uma aplicação *multithread*

Um valor de TLP baixo não necessariamente indica que a performance ou responsividade do sistema é baixa, apenas provê informação sobre qual porção do sistema está inativa. Alternativamente, a TLP indica a taxa de uso de processador *multi-core* em relação à aplicação. Na Figura 3.1, ilustra-se como a exploração da TLP é benéfica para a performance, quando a aplicação é paralelizável.

Figura 3.1 – Exploração do paralelismo em nível de *threads*, utilizando-se múltiplos elementos de processamento, para aumentar a performance de aplicações.



Fonte: O autor.

No caso do processador de *core* único, o aplicativo a ser executado não é dividido em diversas threads, e o tempo de execução das quatro tarefas é o somatório dos tempos de execução individuais de cada tarefa:

$$TempoExecução \approx \sum_{i=1}^{4} Tarefa_i$$

Equação 3.2: Tempo de execução, em um processador *single-core*, de um determinado conjunto de tarefas

No segundo caso, de um processador multi-core, cada tarefa é transformada em uma thread, que pode ser alocada para um core diferente. Nesse caso, o tempo de execução total das quatro tarefas corresponde ao maior tempo de execução entre as tarefas:

$$TempoExecução \approx MAX_i(Tarefa_i)$$

Equação 3.3: Tempo de execução, em um processador *multi-core*, de um determinado conjunto de tarefas

3.1.2 Métricas de performance

Com a grande disseminação dos processadores *multi-core* (BLAKE et al., 2010), o surgimento de novas tecnologias, como o DVFS, e novas arquiteturas (processadores *multi-core* heterogêneos com ISA única), novas metodologias para avaliar esses sistemas passaram a ser desenvolvidas.

Um dos primeiros trabalhos a avaliar a performance de aplicações paralelas em processadores multi-core foi o de (TULLSEN et al., 1995). Nele, os autores advogaram que a performance deveria ser medida a partir da soma das instruções por ciclo (IPC, do inglês Instruction Per Cycle) de cada aplicação (ou *thread*) executando no processador multi-core. (SNAVELY, A.; TULLSEN, D. M., 2000) verificaram que essa métrica favorecia aplicações (ou *threads*) que possuem alto IPC, e propuseram uma nova métrica, que leva em conta a soma ponderada da aceleração de cada aplicação. Na mesma linha, (LUO; GUMMARAJU; FRANKLIN, 2001) propuseram que a performance deveria ser medida através da média harmônica das acelerações de cada aplicação, em vez de utilizar a soma ou média aritmética. Como a média harmônica tende a ser menor quando existe muita variância nos dados, os autores consideram que essa métrica traz uma noção maior de imparcialidade.

De acordo com (EECKHOUT; EYERMAN, 2014), existe um certo consenso de que a utilização de métricas ponderadas para a avaliação de performance de aplicações paralelas em processadores *multi-core* é a melhor opção. Tanto o trabalho de (SNAVELY, A.; TULLSEN, D. M., 2000), quanto de (LUO; GUMMARAJU; FRANKLIN, 2001), encaixam-se nessa categoria. Em ambos os casos, o IPC de cada aplicação (ou *thread*) é primeiramente dividido pelo IPC da aplicação quando executada isolada de outras aplicações.

3.1.3 Métricas de eficiência-energética

Além das métricas de performance, as métricas de consumo de energia também foram extensivamente estudadas nas últimas duas décadas. A maior parte dessas métricas é derivada do produto energia-atraso (ED, do inglês *Energy-Delay*), primeiramente proposta por (GONZALES; HOROWITZ, 1996). Multiplicando-se energia e tempo (execução), e otimizando-se o produto, é possível balancear e avaliar a combinação ótima para uma determinada aplicação. (BROOKS et al., 2000) propôs variações à métrica ED, incluindo o produto potência-atraso, que resulta em uma métrica que dobra o peso dado ao tempo de execução da aplicação (ED^2). Alguns outros trabalhos, como em (ALIOTO et. al., 2012), exploram diferentes pesos para os parâmetros de energia e atraso.

Outra métrica comum de energia é a EPI, energia por instrução, proposta por (ANNAVARAM, 2006). Essa métrica avalia o custo energético de executar uma determinada instrução. Desde que a primeira métrica que relacionava energia e atraso foi proposta, ela se tornou a métrica mais utilizada para medição de eficiência de execução de *benchmarks* na literatura, tanto para processadores unicore, como processadores multi-core homogêneos.

3.1.4 Métricas para processadores *multi-core* heterogêneos

Como vimos na seções 3.1.2 e 3.1.3, muitos trabalhos foram desenvolvidos no sentido de desenvolver métricas e metodologias para a avaliação de performance e eficiência-energética de processadores *multi-core*. Poucos trabalhos na literatura, entretanto, preocuparam-se em propor métricas que avaliem outros aspectos de processadores *multi-core*. Esse fato é ainda mais aparente no caso de arquiteturas heterogêneas. De acordo com (TOMUSK; DUBACH; O'BOYLE, 2014), a falta de métricas para avaliar processadores heterogêneos pode limitar severamente o potencial dessas arquiteturas. As métricas atuais são bastante úteis para a avaliação de processadores *single-core*, e até mesmo *multi-core* homogêneos. Entretanto, para a avaliação de processadores *multi-core* heterogêneos, os *cores* precisam ser considerados como um conjunto, em vez de considerar os *cores* individuais. Sem a existência de métricas apropriadas é difícil estabelecer metas de projeto para os processadores, e comparar duas arquiteturas heterogêneas distintas.

Para endereçar esse problema, em (TOMUSK; DUBACH; O'BOYLE, 2016) são propostas quatro novas métricas para quantitativamente avaliar conjuntos de *cores* heterogêneos. Ou seja, as métricas propostas procuram avaliar o comportamento de *cores* coletivamente, em vez de considerar os aspectos qualquer *core* individualmente.

A primeira métrica proposta é a não-uniformidade localizada. Dado um conjunto de cores que representam pontos ótimos (Eficiência de Pareto) de potência-performance, a ideia dessa métrica é medir o quão uniformemente um subcojunto de *cores* está distribuído ao longo da fronteira de Pareto, avaliando assim, a flexibilidade proporcionada pelo conjunto, em termos de potência-performance. A segunda métrica proposta é a *Gap Overhead*. Essa métrica avalia qual é o desperdicio, em termos de tempo de execução, da seleção *cores*, em relação a toda curva de Pareto de possíveis *cores*. Em outras palavras, essa métrica mede o quanto se renuncia em termos de desempenho, em troca de eficiência energética, e vice-versa. A terceira métrica proposta é a *Set Overhead*. Enquanto a métrica *Gap Overhead* quantifica quanto de um recurso, como tempo de execução, é desperdiçado pelo fato de o processador heterogêneo não implementar todos os *cores* ótimos da fronteira de Pareto, *Set Overhead* estende essa métrica para comparar os *cores* em dois processadores heterogêneos diferentes. *Set Overhead* quantifica o custo de se usar um conjunto de *cores* em detrimento a outro. A última métrica proposta pelos autores é a Generalidade. Essa métrica avalia se os *cores* selecionados são relevantes para qualquer tipo de *benchmark*, ou se alguns desses cores são especializados para tipos específicos de *benchmarks*. A métrica de generalidade pode ajudar o projetista a determinar até que ponto adicionar um *core* de diferentes tipos ao processador heterogêneo pode beneficiar a performance de todos *benchmarks*. E, ainda, pode ser utilizada como uma métrica para comparar a especialização de processadores heterogêneos distintos.

O trabalho de Tomusk é inovador no sentido de buscar novas métricas para avaliar as arquiteturas heterogêneas, especialmente para os casos de aplicações que requerem alta eficiência energética, combinada com performance, como no caso dos *smartphones* e outros dispositivos portáteis. Essas métricas exploram principalmente arquiteturas heterogêneas. (TOMUSK, 2016) procura avaliar o conjunto de *cores*, em termos de várias diferentes métricas, para conjuntos de *cores* de mesma arquitetura, mas de diferentes microarquiteturas, como a arquitetura big.LITTLE vista na seção 2.1.2.

3.2 Aceleradores reconfiguráveis

Muitas arquiteturas reconfiguráveis foram propostas na literatura. O sistema Transmeta Crusoe (DEHNERT et al., 2003) utiliza um processador VLIW para executar instruções x86, através de um sistema de tradução binária. O sistema avaliava o código x86 em tempo de execução, e o transformava em instruções VLIW, salvando essa transformação em uma cache. O processador Warp (LYSECKY; STITT; VAHID, 2006) utiliza um FPGA simplificado acoplado a um ARM7 e a um pequeno processador CAD. Enquanto o programa executa no processador principal, o processador CAD analisa e transforma sequências inteiras em configurações FPGA. Essas configurações são utilizadas posteriormente para executar no FPGA os blocos de instruções transformados. O CCA (CLARK et al., 2004) é uma matriz configurável de unidades funcionais que também é acoplada a um processador ARM. Em tempo de execução, os kernels da aplicação são descobertos através de um analisador de grafos, e transformados em instruções para o CCA. O sistema DIM (BECK et al., 2008) é um CGRA acoplado a um processador MIPS, que utiliza-se de um componente de hardware simplificado para a tradução binária. Kernels da aplicação são transformados em configurações para o CGRA, e salvos em uma cache especial.

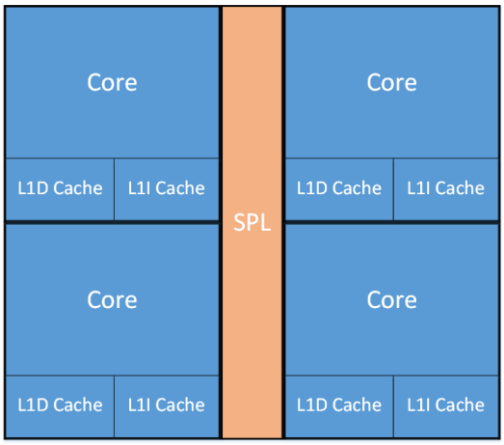
Nessa dissertação iremos focar em aceleradores reconfiguráveis propostos no contexto de processadores *multi-core*, em que esses aceleradores podem ser compartilhados entre os *cores*. A seção 3.2.1 realiza um levantamento desses sistemas, e a seção 3.2.2 foca no acelerador utilizado posteriormente como estudo de caso, para avaliação da métrica proposta.

3.2.1 Aceleradores reconfiguráveis compartilhados em sistemas *multi-core*

Na área de arquiteturas *multi-core* heterogêneas com aceleradores integrados, ainda poucos trabalhos exploram o compartilhamento de aceleradores entre os *cores*. Nessa seção iremos explorar esses trabalhos e avaliar as principais diferenças.

Um dos primeiros trabalhos propostos de inclusão de aceleradores em arquiteturas *multi-core*, e que abordaram o compartilhamento do acelerador entre os *cores* foi o de (GARCIA; COMPTON, 2008). Neste trabalho, os autores propuseram uma forma de compartilhar *kernels* reconfiguráveis entre os *cores*, de forma a aumentar a utilização do *hardware*. A Figura 3.2 ilustra essa arquitetura.

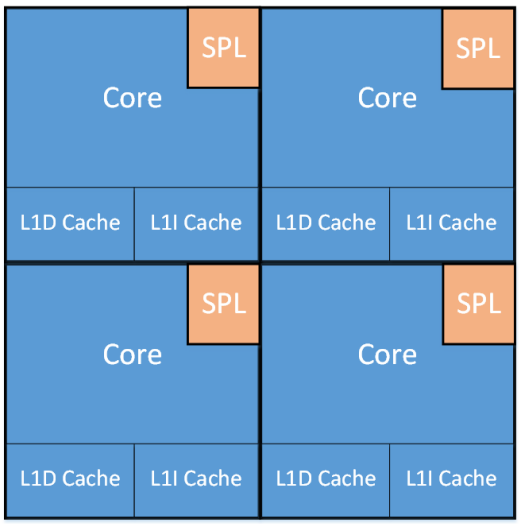
Figura 3.2 – *Cores* de um processador *multi-core* compartilhando uma lógica reconfigurável



Fonte: O autor.

Em (CHEN; PITTMAN; FORIN, 2010), os autores investigam os benefícios da inclusão de suporte à ISA reconfigurável em um processador *multi-core*, e concluem que a combinação das técnica de aplicações paralelizadas, com suporte à ISA customizada, provê ganhos de performance maior que o produto das duas técnicas utilizadas isoladamente. A arquitetura proposta contém um conjunto de *cores* homogêneos, e diferentemente de (GARCIA; COMPTON, 2008), cada *core* tem um *hardware* reconfigurável dedicado, como mostra a Figura 3.3, que pode ser explorado para a implementação de instruções customizadas. Dessa forma, é possível criar um ambiente heterogêneo em tempo de execução, estendendo-se a arquitetura de cada *core* com diferentes instruções customizadas.

Figura 3.3 – Processador *multi-core* em que cada *core* possui um *hardware* reconfigurável privado



Fonte: O autor.

A maior parte dos trabalhos, entretanto, passou a utilizar a arquitetura em que os *cores* compartilham um mesmo *hardware* reconfigurável. Como mostrado por (CHEN; MITRA, 2011), essa arquitetura pode atingir uma melhor utilização dos recursos, e consequentemente um maior ganho de performance na execução de aplicações. Além disso, um *hardware* reconfigurável compartilhado pode acomodar instruções customizadas muito maiores para tarefas individuais, do que seria possível com lógicas reconfiguráveis privadas aos *cores*. Esse tipo de arquitetura foi proposta em trabalhos como (CHEN; MITRA, 2011), (WATKINS; ALBONESI, 2010), e (BOUTHAINA et al., 2013).

Em (WATKINS; ALBONESI, 2010), os autores propõem uma arquitetura *multi-core* heterogênea, que consite de múltiplos *clusters* de *cores*, que compartilham uma lógica reconfigurável, chamada ReMAP. O gerenciamento de múltiplas *threads* que podem ser executadas nos *clusters* é feito de maneira dinâmica. O acelerador em *hardware* é uma estrutura reconfigurável, chamada Lógica Programável Especializada (SPL, do inglês *Specialized Programmable Logic*) que pode ser compartilhada entre múltiplas *threads*. O acelerador reconfigurável pode ser fisicamente particionado entre os *cores*, e é acessado de maneira multiplexada (*round-robin*). O particionamento dinâmico do acelerador reconfigurável, entre as *threads* que o compartilham, juntamente com o escalonamento das *threads* nos diferentes *clusters* reconfiguráveis, é feito de maneira inteligente, através do algoritmo proposto. Essa arquitetura oferece um compartilhamento limitado, já que a estrutura reconfigurável deve ser dividida em partes iguais, além de ter sido projetado para CMPs, que são inerentemente homogêneos.

Em (LEE et al., 2010) um sistema com arquitetura homogênea e organização heterogênea é proposto, em que um FPGA é utilizado com lógica reprogramável, possibilitando suporte à execução de múltiplas *threads*.

Em (CHEN; MITRA, 2011) é proposta uma nova abordagem para a otimização do tempo de execução e área. Neste trabalho, um acelerador em *hardware* reconfigurável é compartilhado entre os *cores*, e instruções customizadas são associadas com diferentes versões. Um algoritmo que escolhe a versão apropriada das instruções customizadas, e o seu ponto de reconfiguração foi desenvolvido de forma a minimizar o tempo de execução. Na arquitetura proposta, o acelerador reconfigurável é integrado ao caminho de dados do processador, ou seja, é um acelerador TCA (do inglês *Tightly-Coupled Accelerator*). Os autores utilizam a plataforma *Stretch* para validação dos resultados, um *software* configurável para arquitetura de processadores.

O trabalho de (BOUTHAINA et al., 2013) é semelhante a (CHEN; MITRA, 2011), mas os autores consideraram também o compartilhamento da mesma instrução customizada entre os diferentes *cores*. Os autores propõem um MPSoC heterogêneo, em que os aceleradores são compartilhados entre os diferentes *cores* de forma inteligente. Diferentemente ainda de (CHEN; MITRA, 2011), a proposta é validada em uma plataforma FPGA reconfigurável. Os conflitos que decorrem das tentativas simultâneas de acessar um mesmo acelerador, por parte de múltiplos *cores*, são resolvidos através de *mutexes*, presentes no escalonador proposto pelos autores.

Em (LIU et al., 2017) é apresentado um método para caracterizar e otimizar aceleradores em *hardware*, integrados no MPSoC através da memória principal. Os autores propõem um método de exploração do espaço de projeto para MPSoCs, que consiste de duas etapas. Inicialmente, cada acelerador é individualmente caracterizado através de uma exploração do espaço de projeto de várias sínteses em alto-nível (HLS, do inglês *High-Level Synthesis*) de um mesmo acelerador, de forma a encontrar projetos Pareto-ótimos. Em um segundo momento, o espaço de projeto em nível de sistema é explorado usando os aceleradores ótimos obtidos na primeira etapa, encontrando configurações com *trade-offs* entre área e performance.

O trabalho de (COTA et al., 2015) propõe um estudo sobre aspectos práticos da integração de aceleradores em *hardware* em ambientes *multi-core* heterogêneos. Esses aspectos práticos incluem a invocação dos aceleradores, e sua interação com com outros componentes da arquitetura, como os elementos de processamento e memórias *cache*.

Os aceleradores em hardware nesse caso são específicos para cada *benchmark*, e são implementados seguindo três diferentes modelos de acoplamento: TCAs (*Tightly-Coupled Accelerators*), em que a implementação do acelerador ocorre diretamente no *core*; LCA (*Loosely-Coupled Accelerator*) com DMA para a LLC (do inglês *Last-Level Cache*), em que a comunicação entre o *core* e o acelerador ocorre a partir do último nível de memória *cache*; e LCA com DMA para a DRAM, em que a comunicação entre o *core* e o acelerador ocorre a partir da memória principal.

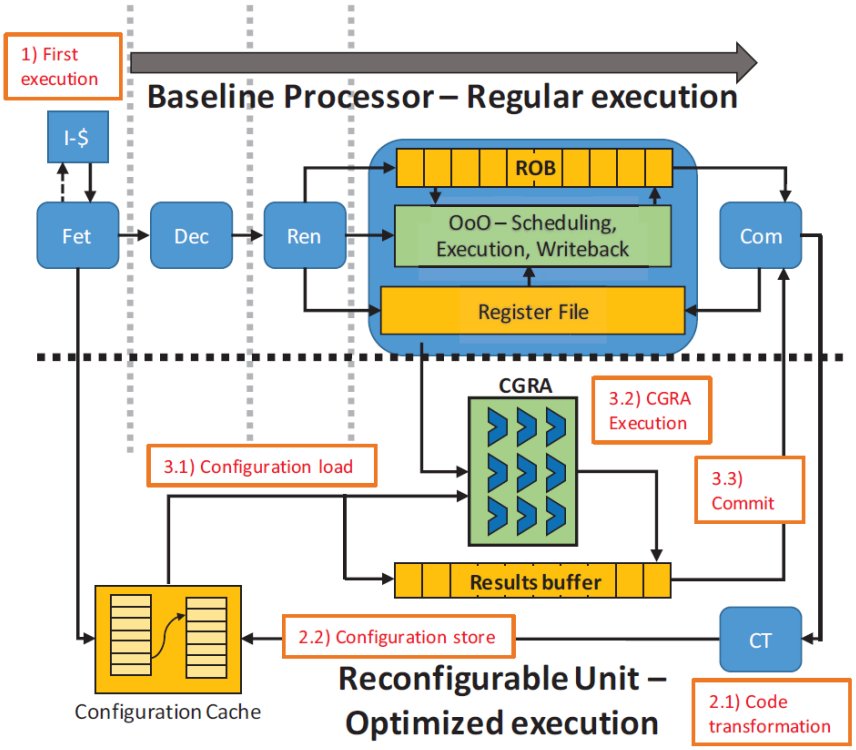
Através da avaliação dos aceleradores projetados, os autores concluem que a aceleradores LCA possuem maior potencial de performance do que aceleradores TCA. Entretanto, alterações profundas na organização do processador são necessárias para que esse potencial seja convertido em performance.

3.2.2 Acelerador reconfigurável utilizado como estudo de caso

O sistema CReAMS (RUTZIG; BECK; CARRO, 2013) acopla uma matriz reconfigurável homogênea de unidades funcionais com um processador principal, para compor cada *core* do sistema. A arquitetura proposta mantém compatibilidade binária, da mesma forma que multiprocessadores superescalares, mas utilizam um circuito especial de tradução. O componente de *hardware* de tradução transforma as sequências de instruções em configurações para a lógica reconfigurável, que podem ser armazenadas e reutilizadas no futuro, evitando que o código seja analisado múltiplas vezes. CReAMS mostrou ser sempre melhor, em termos de performance e eficiência energética, do que um processador multi-core superescalar, considerando a mesma quantidade de transistores em diferentes cenários. A arquitetura CReAMS proposta é composta por 8 DAPs (*Dynamic Adaptive Processor*), que exploram o paralelismo em nível de *thread*. Um DAP é uma arquitetura transparente reconfigurável de grão-grosso, fortemente acoplada ao processador. Existe ainda uma memória de 512KB compartilhada entre os DAP. Essa estrutura de acelerador foi utilizada nessa dissertação para validação e comprovação de resultados.

(BRANDALERO; BECK, 2017) também utiliza uma matriz reconfigurável de grão-grosso (CGRA, do inglês *Coarse-Grained Reconfigurable Array*), mas acoploada a um processador x86 superescalar. Nesse trabalho é proposta uma nova organização do processador, que reduz a utilização de estruturas complexas de *hardware* responsáveis pela decodificação e escalonamento de sequências de instruções repetidas, salvando esse processamento em uma memória *cache* especial, e reutilizando-o posteriormente. O CRGA é utilizado para implementar esse reúso. A Figura 3.4 mostra a organização proposta. No Capítulo 5 dessa dissertação, como estudo de caso, utilizamos a organização proposta por (BRANDALERO; BECK, 2017) para avaliar e validar a métrica proposta, e dessa forma, uma maior ênfase é dada nesse sistema.

Figura 3.4 – Visão-geral do acelerador reconfigurável acoplado à um processador x86



Fonte: (BRANDALERO; BECK, 2017)

A execução nesse sistema segue o seguinte fluxo:

- Primeira Execução: a primeira execução de uma *stream* de instruções segue o fluxo normal de execução em uma arquitetura x86. Esse processador possui uma microarquitetura superescalar, que permite a execução fora-de-ordem de instruções, além de implementar técnicas que permitem *read-after-write* sem necessidade de acessar a memória *cache*;
- Transformação do Código: Em paralelo com a execução normal da arquitetura x86 superescalar, o módulo CT processa sequências de microoperações que serão transformadas em uma configuração do CGRA. Esse módulo implementa um algoritmo responsável por alocar as microoperações no CGRA, e para maximizar o ILP das aplicações, implementa especulação de controle e de dependência de memória;
- Armazenamento da Configuração: Uma configuração para o CGRA é gerado e armazenado em uma *cache* de configuração, juntamente com o PC da primeira microoperação do BB que será executado na lógica reprogramável. Cada configuração

contém as operações das unidades funcionais, e os registradores de destino de cada microoperação na sequência, para que o arquivo de registradores possa ser atualizado posteriormente;

- Carrega a Configuração: Quando o processador detecta um novo BB durante a execução regular, a unidade busca procura na *cache* de configuração se já existe esse BB armazenado. Se o bloco não é encontrado, então ainda não foi transformado, e precisa ser buscado na *cache* de instrução, e é executado regularmente pelo processador. Caso contrário, a configuração CGRA associada é carregada. O fato de executar no CGRA o mesmo BB economiza todo o custoso processo de decodificação e escalonamento de instruções;
- Execução no CGRA: A execução no CGRA começa quando a configuração é completamente carregada. O BB é executado na lógica reconfigurável de grão-grosso, composto por unidades funcionais. O resultado temporário da execução no CGRA é armazenado temporariamente no *buffer* de resultados;
- Final da execução: Uma microoperação de sincronização inserida no ROB marca o final da execução do BB, e o estágio de *commit* do processador começa a realizar o *commit* das microoperações do *buffer* de resultados.

A arquitetura proposta pode reduzir o consumo de energia de um processador x86 superescalar em até 31,4% e melhorar a performance em até 32,6%, adicionando somente 12,5% a mais em área.

3.3 Considerações

Como vimos na seção 3.1, poucos trabalhos propuseram métricas para avaliar características diferentes de performance e energia, de sistemas *multi-core*. (TOMUSK, 2016) propôs novas métricas para arquiteturas *multi-core* homogêneas em arquitetura e heteogêneas em organização, mas não levou em conta a integração de aceleradores nesses sistemas.

Essa dissertação propõe uma nova métrica para avaliar sistemas *multi-core* com aceleradores compartilhados entre os cores (MPSoCs, seção 2.3). Essa métrica visa auxiliar na integração de aceleradores nesses sistemas, e a partir dela, o projetista é capaz de prever a aceleração possível para uma determinada aplicação, e estimar o custo-benefício da adição de

novos aceleradores no sistema. Até o melhor do nosso conhecimento, nenhum trabalho explorou métricas para avaliar essa característica de MPSoCs.

Para avaliar essa nova métrica, utilizamos um sistema como estudo de caso, composto de um processador *multi-core*, e um acelerador reconfigurável (seção 3.2), mais especificamente, um CGRA, visto em detalhes na seção 3.2.2. Esse acelerador é integrado ao sistema, e configurações com o compartilhamento do CGRA entre os *cores*, e de CGRAs dedicados foram utilizados para a avaliação da métrica. Nós mostramos que, enquanto a nossa métrica é descorrelacionada do paralelismo no nível de *threads* (seção 3.1.1), ela apresenta uma forte correlação com a performance (seção 3.1.2) do sistema avaliado, quando um acelerador compartilhado é estendido para múltiplos aceleradores dedicados.

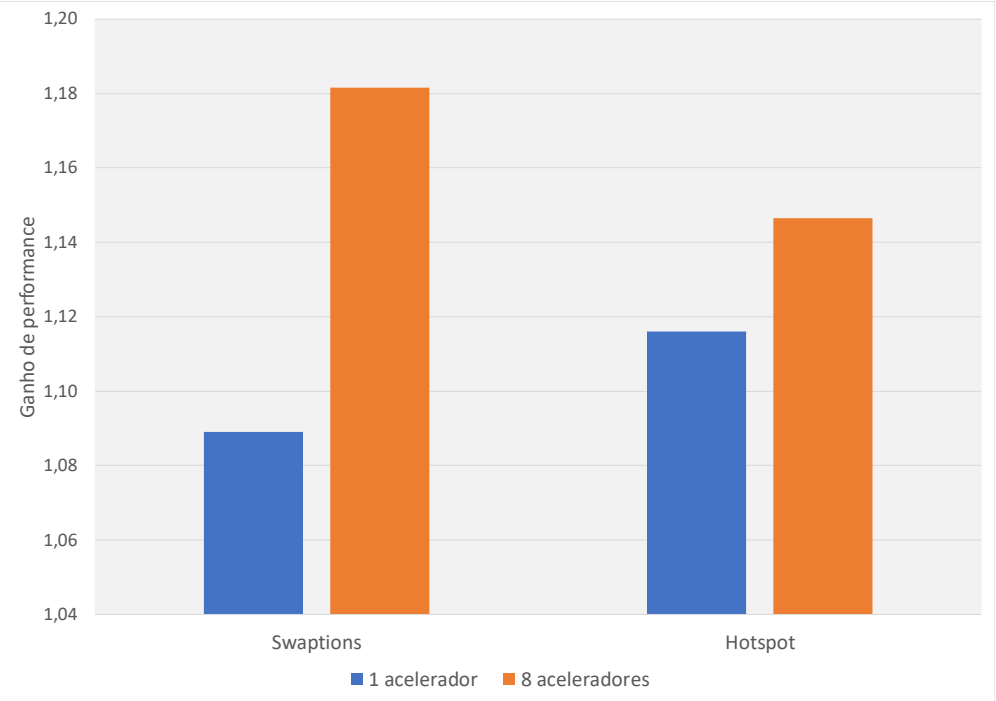
Mostraremos ainda que apesar da utilização de um acelerador reconfigurável no nosso estudo de caso, nossa métrica pode ser facilmente estendida para avaliação de aceleradores específicos (seção 2.2.1)

4 A MÉTRICA PROPOSTA

De forma a encontrar um modelo que possa servir de metodologia para a inclusão de aceleradores em ambientes *multi-core*, esse trabalho propõe uma métrica, SACL (do inglês, *Shared Accelerator Concurrency Level*), ou Nível de Concorrência por Acelerador Compartilhado. Essa métrica avalia o percentual total do tempo de execução em que *threads* estariam competindo pelo uso de um acelerador, de propósito específico ou reconfigurável. O racional por trás dessa métrica é que avaliando o quanto diferentes *threads* estão competindo pelo uso de um acelerador, podemos estimar o ganho de performance possível com a adição de novos aceleradores, em diferentes configurações, e otimizar a quantidade de aceleradores necessária em um sistema para atingirmos um determinado ganho de performance.

A TLP de uma aplicação mede o nível de utilização de recursos em um sistema *multi-core*. Entretanto, essa métrica fornece limitada informação sobre o potencial de compartilhamento de aceleradores entre os *cores*. A Figura 4.1 ilustra resultados obtidos com nossos experimentos, que motivam a necessidade da proposição de uma nova métrica para essa finalidade. Assume-se aqui, um sistema *multi-core* genérico, em que os elementos de processamento compartilham um acelerador em *hardware*.

Figura 4.1 – Aceleração de diferentes aplicações usando um acelerador em *hardware* compartilhado, e um acelerador em *hardware* por *thread*



Fonte: O autor.

A Figura 4.1 mostra a aceleração alcançada ao extendermos um sistema 8-*core* com um único acelerador em *hardware*, para um sistema com um acelerador em *hardware* por *core*. Apesar de ambas as aplicações apresentarem TLPs similares, ambas apresentam comportamentos muito diferentes no contexto de aceleradores em *hardware* compartilhados. Enquanto *Swaptions* apresenta um ganho de performance menor que *Hotspot* no cenário com um acelerador compartilhado (8,91% vs 11,60%, respectivamente), quando utilizam-se aceleradores dedicados o ganho de performance de *Swaptions* é aumentado em 2x com relação ao caso de um acelerador, atingindo 18,16%, enquanto *Hotspot* tem uma aceleração de apenas 1,26x, atingindo 14,65%.

Para determinar os impactos na performance do compartilhamento ou implementação de aceleradores dedicados, nós propomos a SACL.

De forma a definirmos a métrica nessa dissertação, primeiramente iremos discutir o racional por trás de sua definição, e depois iremos apresentar sua definição matemática. Esse Capítulo é finalizado com exemplos numéricos de cálculo da SACL, para três cenários distintos. A metodologia utilizada para sua obtenção é discutida no Capítulo 5, e os resultados obtidos são discutidos no Capítulo 6.

4.1 Racional da métrica

A métrica proposta refere-se ao tempo de execução de uma aplicação, em que diferentes *threads* estariam tentando acessar um mesmo recurso (acelerador em *hardware*). Em outras palavras, dado um conjunto de *threads* de uma mesma aplicação, queremos determinar qual o percentual total de execução, no qual essas *threads* competem pela utilização de um mesmo recurso.

O acelerador pode ser tanto de propósito específico, como reconfigurável, de forma a mantermos a métrica tão genérica quanto possível. Os blocos básicos das threads são divididos entre blocos básicos aceleráveis (BBA) e blocos básicos não-aceleráveis (BBNA). No caso de um acelerador de propósito específico, blocos básicos não-aceleráveis são simplesmente aqueles para os quais o *hardware* não foi projetado para executar. No caso de um acelerador reconfigurável, blocos básicos não-aceleráveis são aqueles que não tem sua execução suportada no acelerador reconfigurável, ou aqueles que sua execução no acelerador não compensa, por exemplo, blocos básicos que acessam frequentemente a memória.

Para não haver desbalanceamento na aceleração das *threads*, o acelerador é compartilhado entre os *cores* através de um algoritmo de escalonamento. Cada *core* ganha acesso ao acelerador conforme a política estabelecida pelo algoritmo, e após o final de execução do bloco básico no acelerador, o escalonador passa a buscar, em tempo de execução, o próximo BBA a ser executado no acelerador em *hardware*. De forma a maximizar a utilização desse acelerador, nós escolhemos para o nosso estudo de caso o algoritmo de escalonamento *First-Come First-Serve* (SILBERSCHATZ; GALVIN; GAGNE, 2008), ou FCFS, de forma que toda vez que o acelerador se torna disponível, a próxima *thread* com um BBA pode utilizá-lo. Maiores detalhes sobre o algoritmo de escalonamento, e sobre o compartilhamento do acelerador, são fornecidos no Capítulo 5.

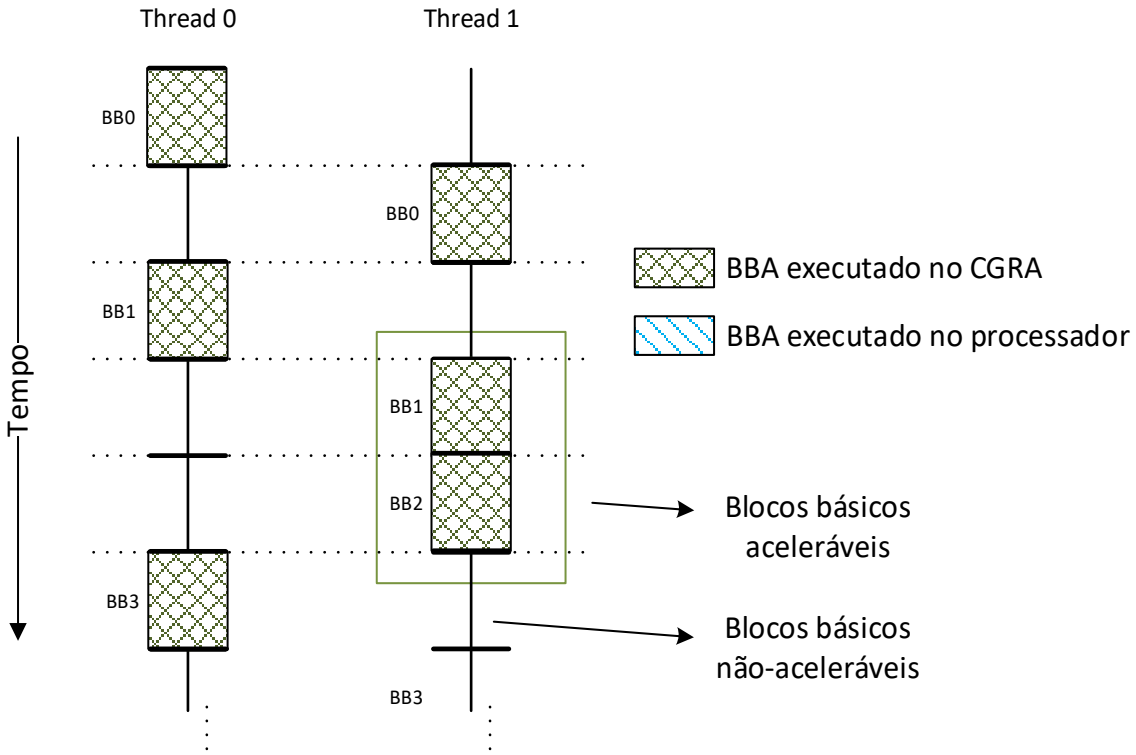
Inicialmente, iremos abordar o cenário com duas threads, e avaliaremos três casos diferentes: nos dois extremos, ou seja, quando não existe nenhuma sobreposição entre BBAs nas *threads*, quando existe sobreposição total entre BBAs nas *threads*, e os casos intermediários de sobreposição entre BBAs. Posteriormente, discutiremos os casos de quatro e oito threads (seção 4.1.2).

4.1.1 Cenário com duas *threads*

Na Figura 4.2 um processador *multi-core* está executando uma aplicação, em que o paralelismo no nível das *threads* é explorado através dos múltiplos elementos de processamento desse processador. Os *cores* desse processador compartilham um acelerador, que pode ser um acelerador de propósito específico ou um acelerador reconfigurável, capaz de acelerar os blocos básicos destacados.

No caso da Figura 4.2, não existe conflito entre blocos básicos aceleráveis executando ao mesmo tempo em ambas as *threads*. Para esse caso, um acelerador compartilhado entre as *threads* é suficiente para que todos os blocos básicos aceleráveis da aplicação sejam executados no acelerador. Ou seja, esse é o melhor caso para o compartilhamento de um acelerador entre os *cores*. Ainda, não existe oportunidade de ganho de performance com a inclusão de mais um acelerador no sistema. Ao fazer isso, estaríamos apenas desperdiçando recursos, pois ambos os aceleradores ficariam ociosos um alto percentual do tempo total de execução da aplicação.

Figura 4.2 – *Threads* sem blocos básicos aceleráveis executando ao mesmo tempo (sobreposição). Blocos básicos destacados com caixas denotam BBAs, e blocos básicos sem detaque corresponem a BBNAs.

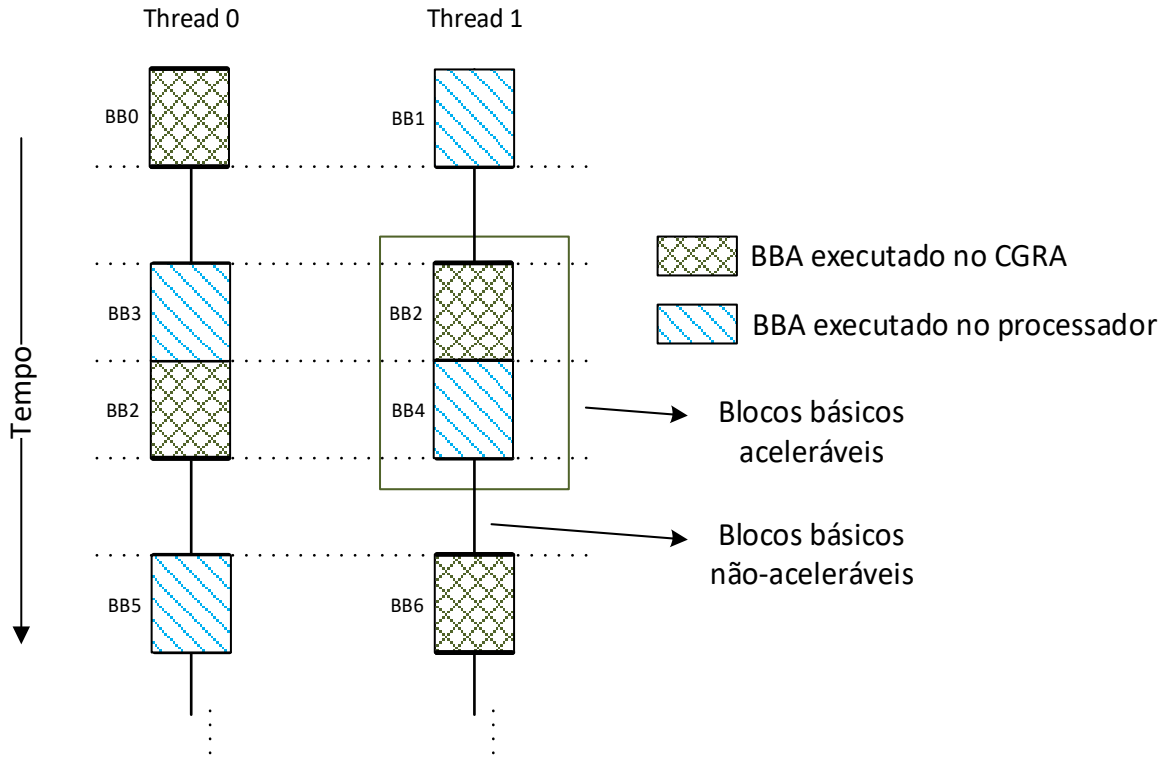


Fonte: O autor.

Nossa métrica pode quantificar a ausência de sobreposição temporal de BBAs na aplicação, e através da avaliação do caso da Figura 4.2, definimos nossa métrica SACL de forma que seu valor seja 0 nesse cenário.

Por outro lado, a Figura 4.3 ilustra a situação em que os blocos básicos aceleráveis das *threads* são executados simultaneamente. Nesse caso, existe ganho de performance com apenas um acelerador, mas um alto percentual de BBAs da aplicação não é executado no acelerador, casos dos BBs 1, 3, 4 e 5 da Figura 4.3. Isso porque os blocos básicos que poderiam ser acelerados acabam executando ao mesmo tempo. Assim, limitado pelo compartilhamento do acelerador, o potencial de ganho de performance da aplicação é limitado. Entretanto, há uma grande oportunidade de aceleração para essa aplicação se mais um acelerador for incluído no sistema.

Figura 4.3 – *Threads* apenas com blocos básicos aceleráveis executando ao mesmo tempo (sobreposição). Blocos básicos destacados com caixas denotam BBAs, e blocos básicos sem detaque corresponem a BBNAs.



Fonte: O autor.

Logo, a situação anterior representa um cenário de pior caso para o compartilhamento de um acelerador, no sentido de desperdiçar um alto potencial de aceleração. O percentual de execução de blocos básicos aceleráveis da aplicação que executam simultaneamente em ambas as *threads* é alto, limitando o ganho possível. Por outro lado, a oportunidade de ganho de performance com a inclusão de outro acelerador é alta nessa mesma situação.

Nossa métrica pode quantificar a sobreposição temporal de todos BBAs na aplicação, e através da avaliação da Figura 4.3, definimos nossa métrica SACL de forma que seu valor seja 1 para esse caso.

As Figuras 4.2 e 4.3 mostram os extremos que podem ocorrer, em relação à sobreposição de BBAs entre as *threads*. Em aplicações reais, a sobreposição de BBAs ocorre em maiores ou menores níveis.

Além dos extremos, nossa métrica é capaz de quantificar esses diferentes níveis de blocos básicos aceleráveis executando simultâneamente nas diversas *threads* do sistema, como um percentual do tempo total de execução da aplicação. A SACL foi definida de forma a assumir valores intermediários para esses casos, com valores maior quando existe maior

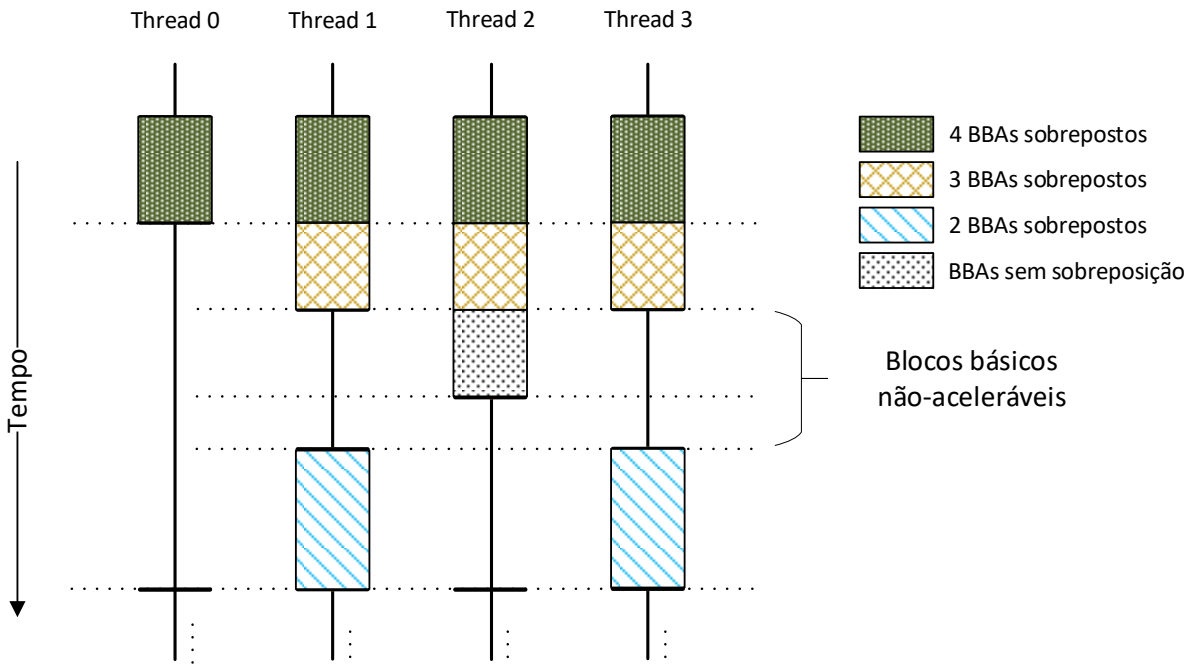
sobreposição de BBAs, e valores menores quando não há tantos BBAs executando simultaneamente nas *threads*.

4.1.2 Cenário com múltiplas *threads*

Para o caso em que a aplicação é dividida em mais que duas *threads*, o racional é análogo. A métrica deve prever, entretanto, que quanto maior o número *threads* executando blocos básicos aceleráveis ao mesmo tempo, maior é o peso dado a esses blocos básicos na nossa métrica. Por exemplo, em um sistema *multi-core* com oito *cores*, se oito *threads* executam ao mesmo tempo um bloco básico acelerável, o peso desse bloco básico na métrica deve ser maior, quando comparado à situação de apenas duas *threads* executarem simultaneamente um bloco básico acelerável. Isso sugere um componente ponderado na definição matemática da métrica, que veremos na seção 4.2, similar ao que ocorre com a TLP (Equação 3.1).

A Figura 4.4 ìlustra o caso em que uma aplicação é divida em quatro *threads*, e mostra os casos possíveis de sobreposição de BBAs nessa situação.

Figura 4.4 – *Threads* com blocos básicos aceleráveis e não-aceleráveis, destacando os casos de sobreposição de BBAs que podem ocorrer com 4 *threads*



Fonte: O autor.

Na Figura 4.4, os blocos básicos destacados com retângulos correspondem a BBAs que executam simultaneamente; e blocos básicos sem destaque representam BBNAs. Neste cenário com quatro *threads*, cinco casos podem ocorrer: 4 BBAs sobrepostos entre todas *threads*, 3 BBAs sobrepostos entre três *threads*, 2 BBAs sobrepostos entre duas *threads*, 1 BBA executando sem sobreposição com BBAs de outras *threads*, e o caso em que nenhum BBA executa em qualquer *thread* em um dado instante de tempo.

Para oito *threads*, as sobreposições de BBAs que podem ocorrer são análogas.

4.2 Definição matemática da métrica

A métrica aqui proposta então procura medir o percentual de sobreposição entre blocos básicos aceleráveis em relação ao total de blocos básicos da aplicação. A métrica deve levar em conta não somente sobreposição total entre blocos básicos aceleráveis, mas também sobreposições parciais, de forma a capturar melhor a característica geral da aplicação. Intuitivamente, conforme mostrado na seções 4.1.1 e 4.1.2, quanto maior for o percentual de sobreposição de blocos básicos aceleráveis, maior é o potencial de ganho de performance da aplicação, com a adição de aceleradores no sistema. Por outro lado, quanto menor o percentual de sobreposição de blocos básicos aceleráveis, maior a oportunidade de utilização de apenas um acelerador compartilhado, com ganhos em termos de área e energia.

Para o cálculo da SACL de uma aplicação definimos um processo iterativo que verifica cada bloco básico pertencente a uma determinada *thread* i. Esse processo avalia a sobreposição dos BBAs entre as *threads* ativas no contexto. Para um bloco básico acelerável da *thread* i, executando em um instante de tempo t, verifica-se para toda *thread* j, sendo j diferente de i, se existe um bloco básico acelerável executando no mesmo instante de tempo t. Se existe tal bloco básico na *thread* j, incrementa-se um contador que verifica o número de *threads* executando BBAs no instante de tempo t. Repete-se, então, esse processo para as n *threads* ativas no instante de tempo t.

Ao final desse processo, temos o número total de blocos básicos aceleráveis, que foram executados apenas na *thread* i, sem que outro BBA estivesse sendo executado em qualquer outra *thread*; o número total de blocos básicos aceleráveis, que foram executados, para qualquer instante de tempo, em duas *threads*; e assim por conseguinte, até o número total de blocos básicos aceleráveis, para qualquer instante de tempo, que foram executados nas n *threads* simultaneamente.

Definimos então, através da Equação 4.1, a SACL de uma *thread* i.

$$SACL_i = \frac{\left(\sum_{j=2}^{n} c_j j\right)}{n}$$

Equação 4.1: Cálculo da SACL para cada uma das n *threads* individuais

Na Equação 4.1, $SACL_i$ é a SACL para a thread i, c_j é a fração, em relação ao número total de blocos básicos da thread n, de BBAs que estão sendo simultaneamente executados em i *threads*, e n corresponde ao número total de *threads*. Observa-se que o percentual de BBAs da *thread* i que não executa simultaneamente com BBAs de qualquer outra *thread*, não é incluído no cálculo da métrica. $SACL_i$ é calculada para cada thread i da aplicação, e a SACL final é dada pela Equação 4.2.

$$SACL_{app} = \frac{\sum_{i=1}^{n} SACL_i}{n}$$

Equação 4.2: Cálculo da SACL da aplicação

A Equação 4.2 corresponde à SACL final da aplicação, que nada mais é que a média aritmética simples da SACL individual de cada *thread*. Em aplicações altamente homogêneas, as SACLs individuais de cada *thread* serão bastante semelhante entre si, o que dispensaria a utilização de uma média para agregar s valores das SACLs individuais. Por outro lado, aplicações heterogêneas podem ter *threads* em que o valor das SACLs são bastante diferentes entre si, e nesse caso, uma média é necessária para agregar esses valores em um único valor final.

Com as equações 4.1 e 4.2 podemos observar que quanto maior o número de *threads* executando BBAs simultaneamente, maior o peso que esse bloco básico terá na métrica, sendo o oposto também verdade. Ainda, a métrica proposta abrange os casos extremos vistos nas seções 4.1.1 e 4.1.2. Ou seja, da forma como foi definida, a SACL pode variar entre 0 e 1. Um valor de 0 significa que não existem BBAs entre todas as *threads* que executam simultaneamente; analogamente, uma SACL de 1 significa que todos os BBAs entre todas as *threads* executam simultaneamente.

Deve ficar claro ainda que, da maneira que nossa métrica é proposta, ela é válida para qualquer tipo de acelerador que seja incluso em um sistema, podendo ser calculada tanto para aplicações inteiras, como kernels de aplicações.

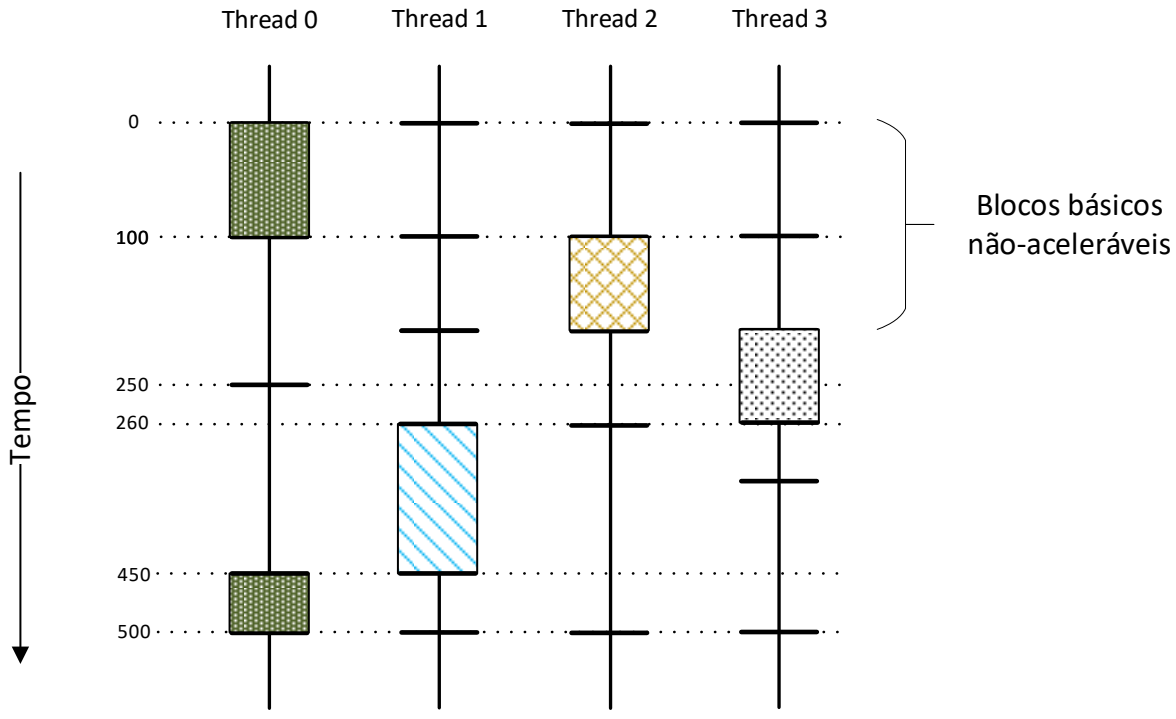
4.3 Exemplos de cálculo da SACL

Nesta seção iremos mostrar, na prática, o cálculo da SACL para três diferentes casos: o melhor caso para o compartilhamento do acelerador, em que não há sobreposição dos BBAs, o pior caso para o compartilhamento do acelerador, em que há sobreposição total dos BBAs entre as *threads*, e o caso em que a SACL possui um valor intermediário entre os casos-limite.

4.3.1 Cálculo da SACL para o melhor caso de compartilhamento do acelerador

Como vimos na seção 4.1, o melhor caso para o compartilhamento do acelerador é quando não há sobreposição de BBAs entre as *threads.* A Figura 4.5 ilustra esse caso.

Figura 4.5 - Cálculo da SACL para o melhor caso de compartilhamento de um acelerador



Fonte: O autor.

Como vimos na seção 4.2, para calcularmos a SACL da aplicação, inicialmente calculamos inicialmente as SACLs individuais para cada *thread.* Para isso, procuramos pelo início de cada bloco básico da *thread* 0, de acordo com o tempo em que começou sua execução, e verificamos se existem blocos básicos aceleráveis nas outras *threads*, que estejam sobrepostos com os BBAs da *thread 0.*

O primeiro bloco básico analisado começa no tempo 0 na *thread 0.* Para considerarmos que existe um bloco básico sobreposto na *thread 1*, a diferença entre o tempo de início do bloco básico da *thread 0,* e o tempo de início do bloco básico na *thread 1*, deve ser menor que o o tempo de execução total do bloco básico da *thread 0*. Para esse caso, as *threads* 1, 2 e 3 possuem blocos básicos com início da execução também no tempo 0, indicando sobreposição, mas esses blocos básicos são não-aceleráveis, e logo, não são contabilizados na nossa métrica. O próximo BBA da *thread* 0 começa a ser executado no tempo 450, mas também não está sobreposto com BBAs em nenhuma outra *thread*. O valor de $SACL_0$, que indica a SACL da *thread 0,* é então 0.

Seguimos a análise da *thread 1*, para o cálculo de $SACL_1$. Nesta *thread,* o único BBA começa a ser executado no tempo 260, e da mesma forma que ocorreu na *thread 0,* esse BBA não está sobreposto com BBA de nenhuma outra *thread*. Logo, $SACL_1$ também é 0. Ao observar a Figura 4.5 vemos que o mesmo ocorre nas *threads* 2 e *3*, em que não existem BBAs sobrepostos com BBAs das outras *threads*. Temos então que $SACL_2$ e $SACL_3$ também têm valor 0. A Equação 4.2 mostra que para o cálculo da SACL final da aplicação, fazemos a média aritmética das SACLs individuais. Nesse caso, a SACL da aplicação também é 0.

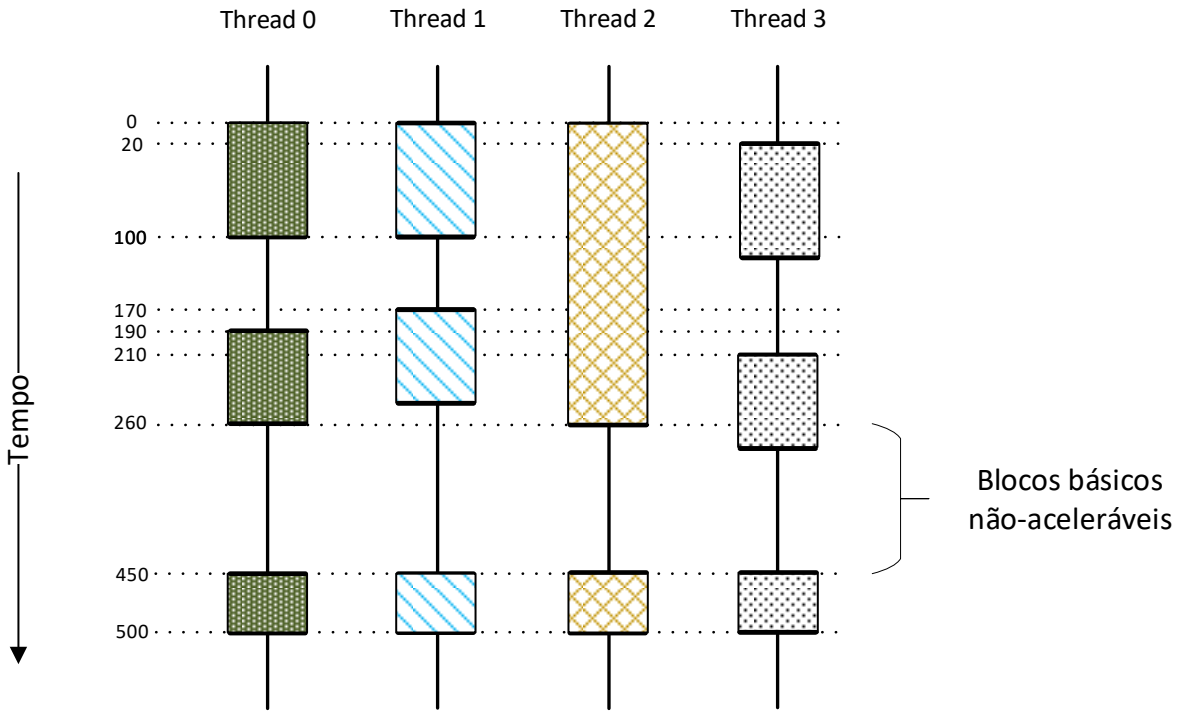
4.3.2 Cálculo da SACL para o pior caso de compartilhamento do acelerador

A Figura 4.6 indica o pior caso de compartilhamento do acelerador, no qual os BBAs das diferentes *threads* estão totalmente sobrepostos.

O primeiro bloco básico analisado começa no tempo 0 na *thread 0*. No mesmo instante de tempo, BBAs nas *threads* 1 e 2 têm sua execução iniciada. O BBA da *thread* 3, que começa a executar no instante de tempo 20, também é considerado sobreposto, pois a diferença entre o início do bloco básico da *thread* 0 e o início bloco básico da *thread* 3 é menor que o tempo total de execução do BBA da *thread 0*. Considerando essa metodologia para avaliar blocos básicos sobrepostos, verificamos que os três BBAs da *thread* 0 executam simultaneamente com BBAs

das três outras *threads*. Para a *thread* 0 temos então, considerando um total de três BBAs, a seguinte SACL individual:

Figura 4.6 Cálculo da SACL para o pior caso de compartilhamento de um acelerador



Fonte: O autor.

$$\begin{cases} c_2 = 0 \\ c_3 = 0 \\ c_4 = 1 \end{cases}$$

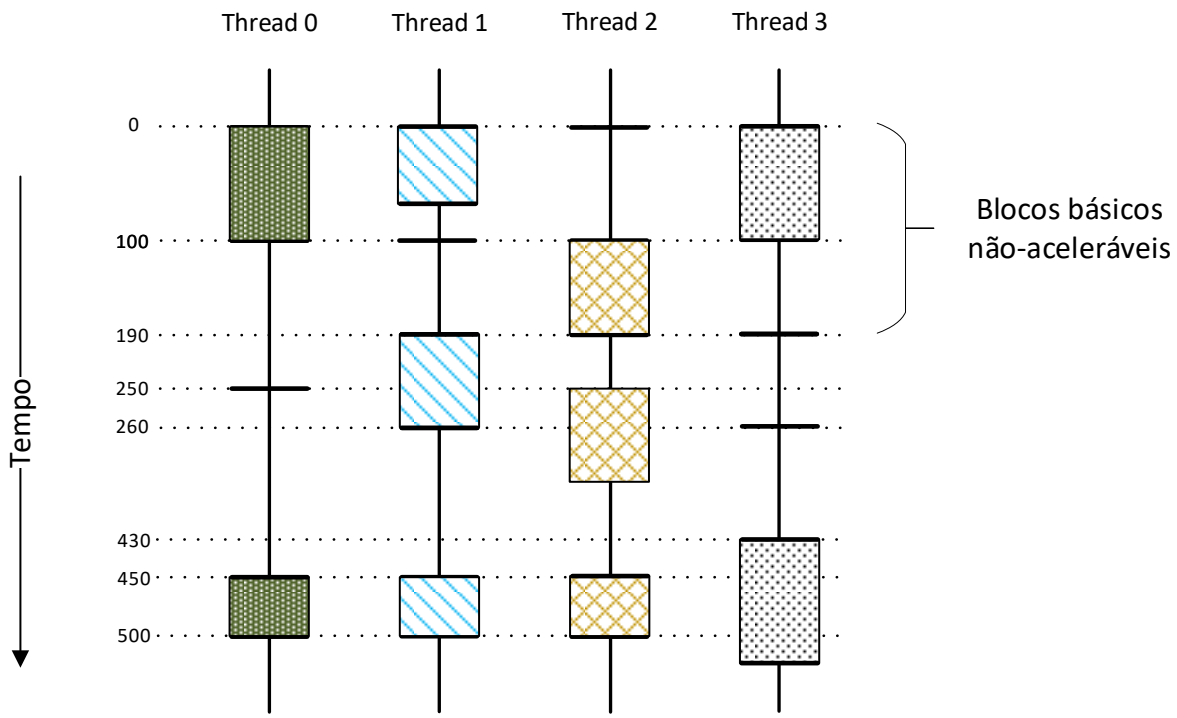
$$SACL_0 = \frac{2c_2 + 3c_3 + 4c_4}{4} = 1$$

Pode-se verificar através da Figura 4.6 que o mesmo ocorre nas outras *threads*: todos BBAs de cada *thread* executam simultaneamente com três outros BBAs das outras *threads*. A SACL individual de cada *thread* é então, 1, e assim o é a SACL da aplicação.

4.3.2 Cálculo da SACL para um caso intermediário de compartilhamento do acelerador

A Figura 4.7 indica um caso intermediário de compartilhamento do acelerador, em que os BBAs das diferentes *threads* estão sobrepostos de diferentes formas.

Figura 4.7 - SACL para um caso intermediário de compartilhamento de um acelerador



Fonte: O autor.

O primeiro BBA da *thread 0* tem sua execução iniciada no tempo 0. Neste mesmo instante de tempo, as *threads* 1 e 3 também têm BBAs iniciando sua execução. Assim, o BBA da *thread 0* executa simultâneamente com outros dois BBAs. O segundo BBA da *thread 0* inicia sua execução no tempo 450. As *threads* 1 e 2 também tem BBAs cuja execução inicia no tempo 450. A *thread* 3 também possui um BBA, mas sua execução inicia no tempo 430. Como a diferença de início de execução desses BBAs (20) é menor que o tempo total de execução bloco básico da *thread 0* (50), o BBA da *thread* 3 também é considerado sobreposto. O cálculo da SACL individual da *thread 0*, considerando que essa *thread* tem 4 blocos básicos no total, fica da seguinte forma:

$$\begin{cases} c_2 = 0 \\ c_3 = 0{,}5 \\ c_4 = 0{,}5 \end{cases}$$

$$SACL_0 = \frac{2.c_2 + 3.c_3 + 4.c_4}{4} = \frac{1{,}5 + 2}{4} = 0{,}62$$

A *thread 1* tem seis blocos básicos no total, sendo três BBAs. O primeiro BBA executa simultaneamente com BBAs das *threads* 0 e 3. O segundo BBA executa simultaneamente com um BBA das *thread* 2, e o terceiro BBA executa simultaneamente com BBAs das outras três *threads* ativas. O cálculo da SACL individual da *thread 1* fica da seguinta forma:

$$\begin{cases} c_2 = 0{,}33 \\ c_3 = 0{,}33 \\ c_4 = 0{,}33 \end{cases}$$

$$SACL_1 = \frac{2.c_2 + 3.c_3 + 4.c_4}{4} = \frac{0{,}66 + 0{,}99 + 1{,}32}{4} = 0{,}74$$

O cálculo das SACLs individuais das *threads* 2 e 3 é feito de forma análoga. Os valores obtidos são os seguintes: $\text{SAC}L_2 = 0{,}49$ e $\text{SAC}L_3 = 0{,}87$. De acordo com a Equação 4.2, a SACL da aplicação é encontrada através da média aritmética das SACLs individuais: $\text{SAC}L_{app} = 0{,}68$.

5 METODOLOGIA

Durante este trabalho, algumas ferramentas e simuladores foram utilizados, além da criação de diversos *scripts*. Nesta seção iremos explorar a infraestrutura utilizada para a obtenção dos resultados desse trabalho, que serão apresentados no capítulo seguinte, bem como a metodologia utilizada. Inicialmente, iremos apresentar o simulador gem5, vastamente utilizado na literatura para a pesquisa de arquiteturas e microarquiteturas de processadores. Então, iremos apresentar os *scripts* customizados, desenvolvidos especialmente para extrair algumas características das simulações obtidas com o gem5. Posteriormente, cobrimos os *benchmarks* utilizados nesse trabalho. Por fim, apresentaremos as configurações de sistema (processador/acelerador) implementadas para obtenção dos resultados, e avaliação da métrica.

5.1 O simulador gem5

O gem5 (BINKERT et al., 2011) é um simulador altamente configurável, largamente utilizado no meio acadêmico para a simulação de arquitetura e microarquitetura, com precisão de ciclo. As aplicações são executadas no modo de emulação de chamadas de sistema (SE, do inglês *Syscall Emulation Mode*). O modelo DerivO3CPU é um modelo bastante acurado de um processador superescalar com execução fora de ordem, e forma a base das simulações. Os *scripts* padrão de simulação do gem5 foram utilizados, com algumas variações de parâmetros microarquiteturais para refletir a proposta dessa dissertação.

O suporte a aplicações paralelas no gem5 é dado por uma biblioteca própria para execução de programas *multi-thread*, chamada m5threads, que é basicamente uma versão reduzida do pthreads. O gem5 possui dois modos de simulação possíveis: emulação de chamadas de sistema (SE, do inglês *syscall emulation*) e emulação completa do sistema (FS, do inglês *full-system simulation*). O modo SE simula a execução do programa no processador escolhido, e emula todas as chamadas de sistema.

Cada *benchmark* simulado foi desenvolvido utilizando-se as interfaces de programação paralela OpenMP (DORTA et al., 2005) ou PThreads (BUTHENHOF, 1997), que permitem uma alocação dinâmica das *thread* ao longo da execução. Particularmente, a biblioteca OpenMP ainda fornece métodos que descobrem, em tempo de execução, o número de elementos de processamento do processador. Dessa forma, essa biblioteca pode tirar vantagem de todos os recursos disponíveis, até mesmo quando a plataforma muda, sem necessidade de alteração no

código, ou recompilação. Os detalhes dos *benchmarks* utilizados são descritos na seção 5.3. Quando o gem5 executa uma aplicação, um arquivo de saída chamado *trace file* é gerado, e o mesmo contém detalhes de todas as instruções executadas em cada uma das *threads* criadas ao longo da aplicação. A Figura 5.1 mostra um excerto de um *trace file* do gem5.

Figura 5.1 - Excerto do trace file obtido do simulador gem5

```
47175000: system.cpu1 T0 : @worker+164    : addsd       ecx, eax
47175000: system.cpu1 T0 : @worker+164.0  :   ADDSD_XMM_XMM : maddf   %xmm1_low, %xmm1_low, %xmm0_low : FloatAdd :  D=0x4026000000000000
47175000: system.cpu1 T0 : @worker+168    : jnle        0xfffffffffffffe6
47175000: system.cpu1 T0 : @worker+168.0  :   JNLE_I : rdip   t1, %ctrl153,  : IntAlu :  D=0x000000000040100a
47175000: system.cpu1 T0 : @worker+168.1  :   JNLE_I : limm   t2, 0xfffffffffffffe6 : IntAlu :  D=0xfffffffffffffe6
47175000: system.cpu1 T0 : @worker+168.2  :   JNLE_I : wrip   , t1, t2 : IntAlu :
47176000: system.cpu1 T0 : @worker+144    : mov rdx, DS:[8*rax + r8]
47176000: system.cpu1 T0 : @worker+144.0  :   MOV_R_M : ld   rdx, DS:[8*rax + r8] : MemRead :  D=0x00000000006d7830 A=0x6d7b18
47176000: system.cpu1 T0 : @worker+148    : movsd       eax, DS:[rcx + rdx]
47176000: system.cpu1 T0 : @worker+148.0  :   MOVSD_XMM_M : ldfp   %xmm0_low, DS:[rcx + rdx] : FloatMemRead :  D=0x3ff0000000000000 A=0x6d78a8
47176000: system.cpu1 T0 : @worker+153    : mulsd       eax, DS:[8*rax + rdi]
47176000: system.cpu1 T0 : @worker+153.0  :   MULSD_XMM_M : ldfp   %ufp1, DS:[8*rax + rdi] : FloatMemRead :  D=0x3ff0000000000000 A=0x6d6b68
47176500: system.cpu2 T0 : @_IO_file_xsputn+135.2  :   POP_R : mov   r12, r12, t1 : IntAlu :  D=0x00000000004a3124
47176500: system.cpu2 T0 : @_IO_file_xsputn+137    : pop         r13
47176500: system.cpu2 T0 : @_IO_file_xsputn+137.0  :   POP_R : ldis   t1, SS:[rsp] : MemRead :  D=0x00007ffff7fbe6c8 A=0x7ffff7fbe140
47176500: system.cpu2 T0 : @_IO_file_xsputn+137.1  :   POP_R : addi   rsp, rsp, 0x8 : IntAlu :  D=0x00007ffff7fbe148
```

Fonte: O autor.

Cada linha do *trace file* é composta pelo ciclo no qual a instrução terminou de ser executada, pelo *core* no qual a instrução foi executada, e o PC da instrução. Além disso, o *trace* detalha a instrução e microoperações executadas em cada ciclo de execução do processador superescalar. Para cada microoperação ainda existe a informação dos registradores utilizados, e também o tipo de instrução executada.

5.2 Scripts customizados

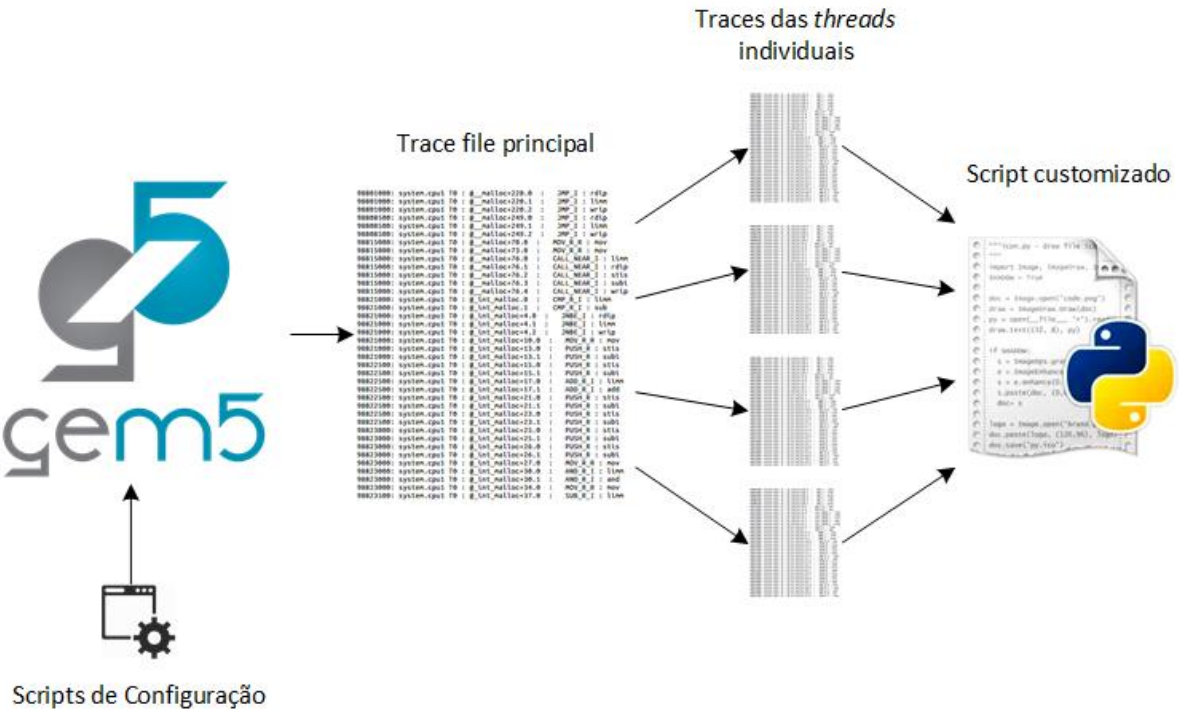
Para a obtenção dos resultados desse trabalho, diversos *scripts* foram desenvolvidos, com o objetivo de extrair informações do *trace file* obtido do simulador gem5. Nessa seção, iremos apresentar nossa metodologia utilizada para obtenção, a partir do *trace file*, das seguintes características da aplicação:

- Separação do *trace file* principal em n arquivos, em que n é o número de *threads* utilizadas para executar a aplicação, de forma a separar as instruções executadas em cada uma das *threads;*
- O número de ciclos que cada bloco básico leva para executar no processador superescalar;
- O tamanho médio dos blocos básicos, em número de microoperações, de cada *thread*, e por conseguinte, da aplicação;

- Obtenção da SACL a partir do *trace file* do gem5.

A Figura 5.2 mostra a visão geral da metodologia utilizada para a obtenção da métrica proposta nessa dissertação.

Figura 5.2 – Visão geral da metodologia para extração das características de interesse de cada *benchmark*



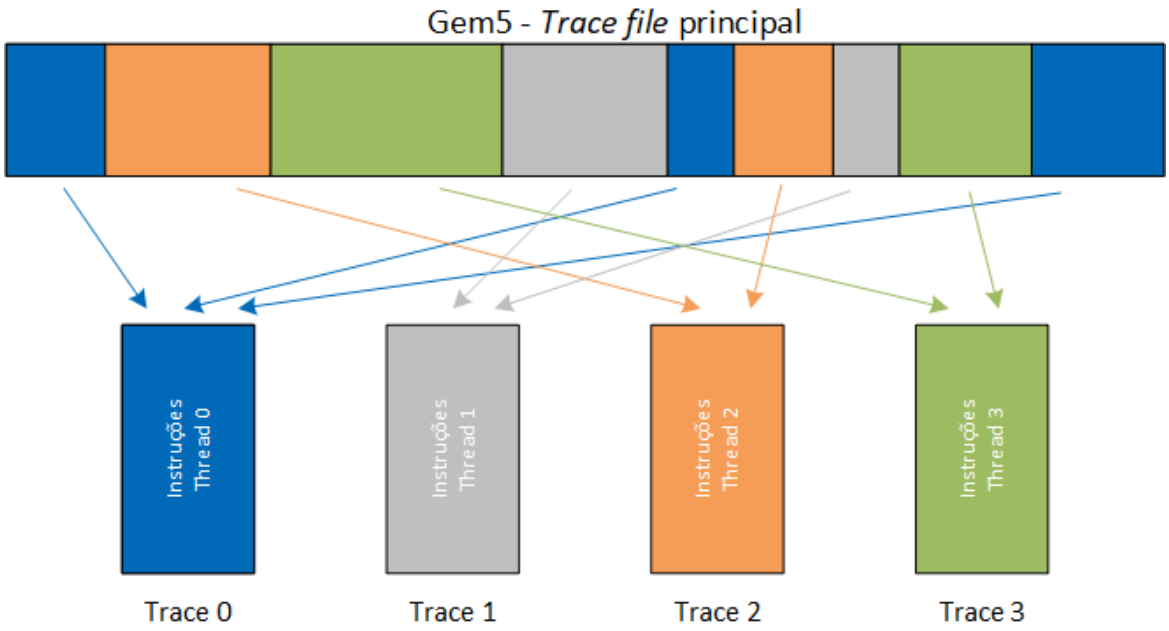
Fonte: O autor

Nas próximas seções apresentamos a metodologia para obtenção de cada uma dessas características listadas.

5.2.1 *Trace files* individuais para cada *thread*

Como já discutido na seção 5.1, o simulador gem5 é responsável por gerar um *trace file* contendo detalhes de todas instruções de todas *threads.* Entretanto, para a análise do arquivo, e posterior obtenção da métrica SACL proposta, é necessário que o *trace file* principal seja separado de acordo com as *threads* às quais as instruções pertencem. Para isso, utilizou-se um *script* que lê o arquivo principal e o separa em diversos arquivos. A Figura 5.3 ilustra esse processo.

Figura 5.3 – *Trace file* principal obtido do gem5 é separado em vários *traces* individuais, um para cada *thread*



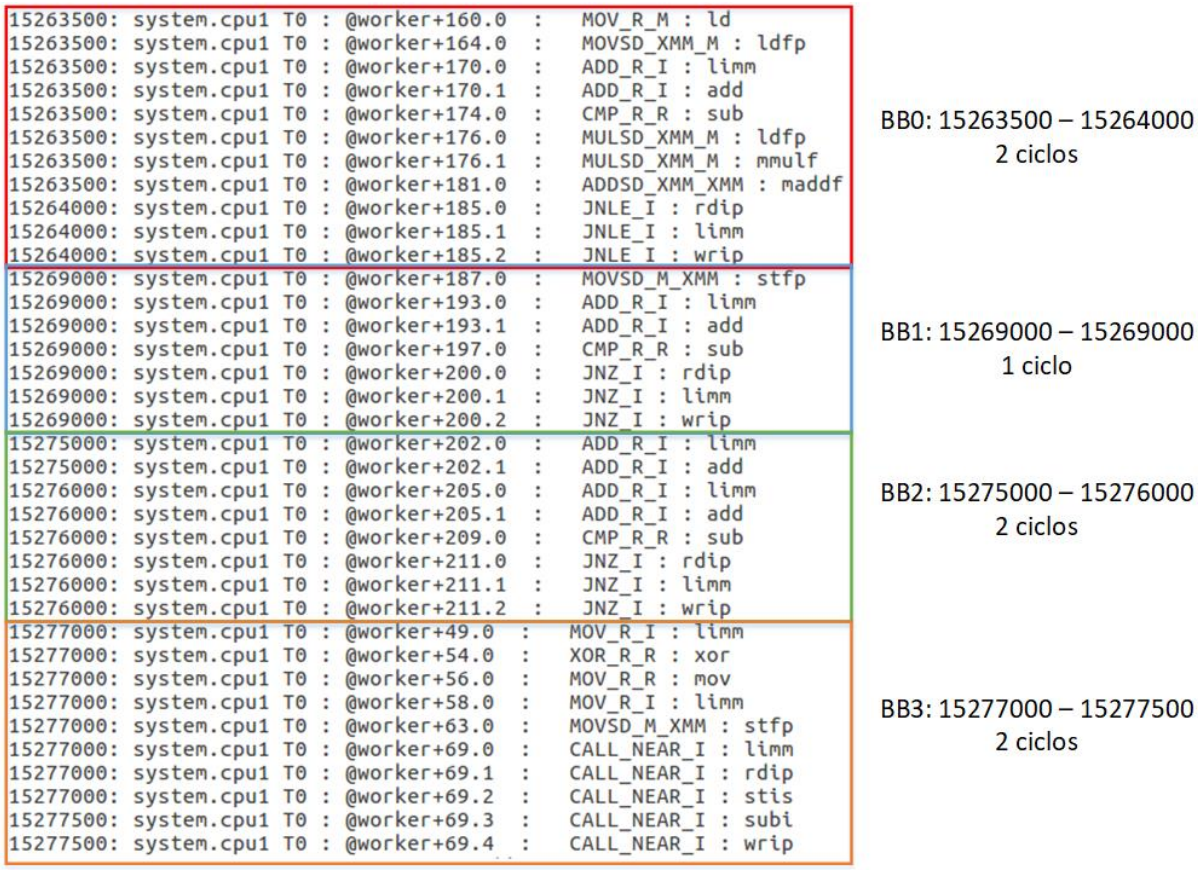
Fonte: O autor.

Posteriormente, as instruções de cada um dos arquivos são agrupadas em blocos básicos. De acordo com (SHERWOOD et al., 2001), um bloco básico é uma seção do código que é executada do início ao final com apenas uma entrada e uma saída. O final de um bloco básico é marcado por uma instrução de desvio (*branch* ou *jump*), ou por uma chamada de função (*call*). Dessa forma, um *script* é responsável por analisar os *traces* individuais de cada uma das *threads* executadas em uma aplicação, e verificar as instruções para agrupá-las em blocos básicos.

5.2.2 Tempo de execução dos blocos básicos da aplicação

A Figura 5.4 mostra como os blocos básicos são divididos em cada um dos *traces* obtidos a partir do *trace file* principal. Para sua posterior identificação e comparação, de cada bloco básico são armazenadas as instruções (e microoperações) que os compõem, e os ciclos de execução em que cada bloco básico começou e terminou. Com essa informação é possível calcular o tempo de execução, em número de ciclos, que o processador levou para executar um determinado bloco básico.

Figura 5.4 – Excerto do trace file obtido do simulador gem5, com destaque para como é obtido o tempo de execução (em número de ciclos) de cada bloco básico da aplicação.



Fonte: O autor.

No excerto do *trace file* da *thread* 1 ilustrado na Figura 5.4 existem quatro blocos básicos distintos. De acordo com o padrão utilizado pelo gem5, cada incremento de 500 no valor mostrado na coluna da esquerda, corresponde a um incremento de um ciclo de CPU no processador superescalar. Com isso é possível determinar o início e o final do bloco básico, e estimar o número de ciclos necessários para sua execução.

Além disso, podemos observar que a microoperação *wrip* é responsável por dividir os diferentes blocos básicos. Dentro de uma instrução de desvio ou de salto, essa microoperação é a última a ser executada, e sua função é escrever no ponteiro de instruções o novo endereço da memória *cache* de instruções onde está a próxima microoperação a ser executada.

5.2.3 Tamanho médio dos blocos básicos da aplicação

De forma semelhante, o processo para obtenção do tamanho médio dos blocos básicos da aplicação pode ser observado pela Figura 5.4. O mesmo script que determina o número de ciclos que cada bloco básico leva para executar no processador, também avalia o número de microoperações contidas em cada bloco básico. Para cada *thread*, soma-se o número total de microoperações e divide-se pelo número total de blocos básicos daquela *thread*. Para obtenção do tamanho médio dos blocos básicos de toda aplicação, simplesmente soma-se o tamanho médio obtido por *thread*, e divide-se pelo número de *threads* da aplicação. Formalmente:

$$\overline{BB_{Total}} = \overline{BB_1} + \overline{BB_2} + \cdots + \overline{BB_n}$$

Equação 5.1: Tamanho médio, em número de microoperações, dos blocos básicos de uma aplicação

Na Equação 5.1, cada termo representa o tamanho médio dos blocos básicos por *thread*, e n é o número de threads da aplicação.

5.2.4 Obtenção da SACL

A SACL, formalmente apresentada no Capítulo 4, também é obtida a partir do *trace file* do gem5 através de um *script* customizado.

O *script* implementa o seguinte algoritmo, para o cálculo da SACL:

- Determina BBs de cada *thread*: Inicialmente, o algoritmo que calcula a SACL da aplicação varre cada *thread*, marcando cada um dos BBs, armazenando o ciclo em que a primeira e última instruções do BB foram executadas;
- Determina BBs aceleráveis: De acordo com as restrições impostas na seção 4.1, o algoritmo determina quais blocos básicos da aplicação são aceleráveis, e quais são não-aceleráveis;
- Procura BBs aceleráveis executando simultâneamente: O algoritmo verifica, para todas as *threads* ativas no contexto, quantos blocos básicos aceleráveis executam concorrentemente, para cada ciclo de execução do *trace file*. Para nossa métrica, definimos que existe concorrência de execução quando a diferença entre o início de

blocos básicos aceleráveis é menor ou igual ao tempo de execução médio, em número de ciclos, dos blocos básicos da aplicação;

- Cálculo da SACL individual de cada *thread*: Para cada *thread*, calcula-se a SACL individual, através da implementação da Equação 4.1;
- Cálculo da SACL da aplicação: Através da implementação da Equação 4.2, calcula-se a SACL final da aplicação.

A escolha pela granularidade em blocos básicos, para o cálculo da SACL, traz a vantagem de ser menos intensiva computacionalmente, em comparação com uma granularidade em nível das instruções. Entretanto, traz a desvantagem de perder em precisão em comparação com um método que utilize-se das instruções para o cálculo da métrica.

5.3 Benchmarks

Os *benchmarks* utilizados para a avaliação da métrica proposta, e para os experimentos de ganho de performance do Capítulo 5, fazem parte de dois conjuntos de *benchmarks* bastante utilizados na literatura: PARSEC e RODINIA.

Os processadores *multi-core* se tornaram as formas dominantes de processadores de propósito geral (BLAKE et al., 2009). Essa transição criou uma mudança disruptiva, pois pela primeira vez ganhos de performance substanciais não poderiam ser obtidos sem modificação profunda no código-fonte das aplicações. Cada vez mais, a responsabilidade dos ganhos de performance recai sobre os programadores, que devem desenvolver *softwares* de maneira instrinsecamente paralela. Diversos *benchmarks* passaram a ser propostos para a avaliação de arquiteturas *multi-core*, e dentre eles, o PARSEC (BIENIA et al., 2008) ganhou destaque por incluir aplicações de múltiplas *threads* (algo inovador na época), que não eram focadas em computação de alta-performance, e com características diversas.

Os *benchmarks* do PARSEC utilizados nesse trabalho são os seguintes:

- *Blackscholes:* Essa aplicação é muito utilizada no mercado financeiro, para o cálculo de preço de opções utilizando-se equações diferenciais parciais;
- *Swaptions:* Essa aplicação é utilizada no mercado financeiro para o cálculo de um portfólio de opções em uma troca, chamados *swaptions*. Swaptions utiliza-se da simulação de Monte-Carlo (MC) para o cálculo dos preços;

- *Canneal:* Utilizado para minimizar o custo de roteamento no projeto de chips.

Além desses, ainda escolhemos a multiplicação de matrizes para fazer parte dos *benchmarks* desse trabalho por ser muito utilizada para as mais variadas aplicações, e por exibir um comportamento bastante particular. A multiplicação de matrizes é altamente paralelizável, com um alto potencial de exploração de TLP, e suas threads são altamente homogêneas. As aplicações do PARSEC, e a multiplicação de matrizes, utilizam a biblioteca PThreads de paralelismo.

O outro conjunto de *benchmarks* utilizado nesse trabalho é o RODINIA (CHE et al., 2009). Proposto com a ideia de guiar estudos em arquiteturas *multi-core* heterogêneas, fornecendo implementações de cada uma das aplicações para processadores multi-core e GPUs. A maior parte dos *benchmarks* anteriores, como o PARSEC, focava em fornecer aplicações seriais ou paralelas para arquiteturas de CPU convencionais, e o RODINIA foi proposto com a ideia de ocupar o nicho de aplicações para o estudo de arquiteturas heterogêneas contendo aceleradores.

Os *benchmarks* do RODINIA utilizados nesse trabalho foram os seguintes:

- SRAD (do inglês, Speckle Reducing Anisotropic Diffusion): É um algoritmo de difusão baseado em equações diferenciais parciais, e é utilizado para remover pontos indesejados em imagens sem sacrificar características importantes. SRAD é muito utilizado em aplicações que demandam processamento de imagens, como imagens ultrasônicas e de radares;
- HotSpot (HS): É uma ferramenta de simulação térmica usada para estimar a temperatura de um processador baseado em suas características arquiteturas e medidas de potências simuladas;
- Back Propagation (BP): É um algoritmo de *machine learning* que treina os pesos das conexões entre nós em uma rede neural;
- K-Means (KM): É um algoritmo de clusterização utilizado extensivamente em mineração de dados. Esse algoritmo identifica pontos relacionados associando cada ponto com o cluster mais próximo, computando novos centróides para cada cluster, e iterando até a convergência;
- Particle Filter (PF): É um modelo probabilístico para o rastreamento de objetos em um ambiente ruidoso, usando um dado conjunto de amostras de partículas;

- PathFinder: É um algoritmo de programação dinâmica para encontrar o menor caminho em um grid 2D, linha por linha, através da escolha do menor peso acumulado;
- Myocyte: É uma aplicação de simulação utilizada na medicina para modelar miócitos cardíacos, e simular seu comportamento;
- k-Nearest Neighbour (k-NN): É uma aplicação de mineração de dados, que é utilizada para encontrar os k vizinhos mais próximos em um conjunto de dados não estruturado;
- LavaMD: É uma aplicação de dinâmica molecular, que cálcula o potencial e realocação de partículas dentro de um espaço 3D.

As aplicações do RODINIA utilizados foram implementadas utilizando-se a biblioteca OpenMP de paralelismo. A Tabela 5.1 resume os benchmarks utilizados nessa dissertação.

Tabela 5.1 – Os *benchmarks* utilizados para a obtenção dos resultados. Foram utilizados *benchmarks* dos conjuntos PARSEC e Rodinia, implementados em PThreads e OpenMP respectivamente

Aplicação	Benchmark	Biblioteca de Paralelismo
Blackscholes	PARSEC	PThreads
Swaptions	PARSEC	PThreads
Canneal	PARSEC	PThreads
MxM	N.A.	PThreads
Kmeans	Rodinia	OpenMP
LavaMD	Rodinia	OpenMP
Backpropagation	Rodinia	OpenMP
Pathfinder	Rodinia	OpenMP
Srad	Rodinia	OpenMP
Myocyte	Rodinia	OpenMP
NN	Rodinia	OpenMP
Particle Filter	Rodinia	OpenMP
Hotspot	Rodinia	OpenMP

Fonte: O autor.

Os *benchmarks* foram compilados utilizando o compilador g++ versão 4.9.2, com a flag de otimização -O3, a fim de habilitar otimizações agressivas pelo compilador. Essa versão de compilador foi utilizada devido à problemas de compatibilidade com o simulador. O código utilizado nos experimentos é a execução do código original no gem5, sem nenhuma alteração. Para a obtenção dos resultados dessa dissertação considerou-se apenas o *kernel* das aplicações, omitindo-se blocos básicos e instruções relacionados com entrada e saída de dados.

Na seções 5.3.1 e 5.3.2, apresentamos o comportamento dinâmico dos *benchmarks*, assim como o tamanho médio de seus blocos básicos. Dessa forma, mostramos que escolhemos um conjunto de aplicações bastante diverso para avaliarmos com a SACL.

5.3.1 Cobertura dos blocos básicos em relação à aplicação

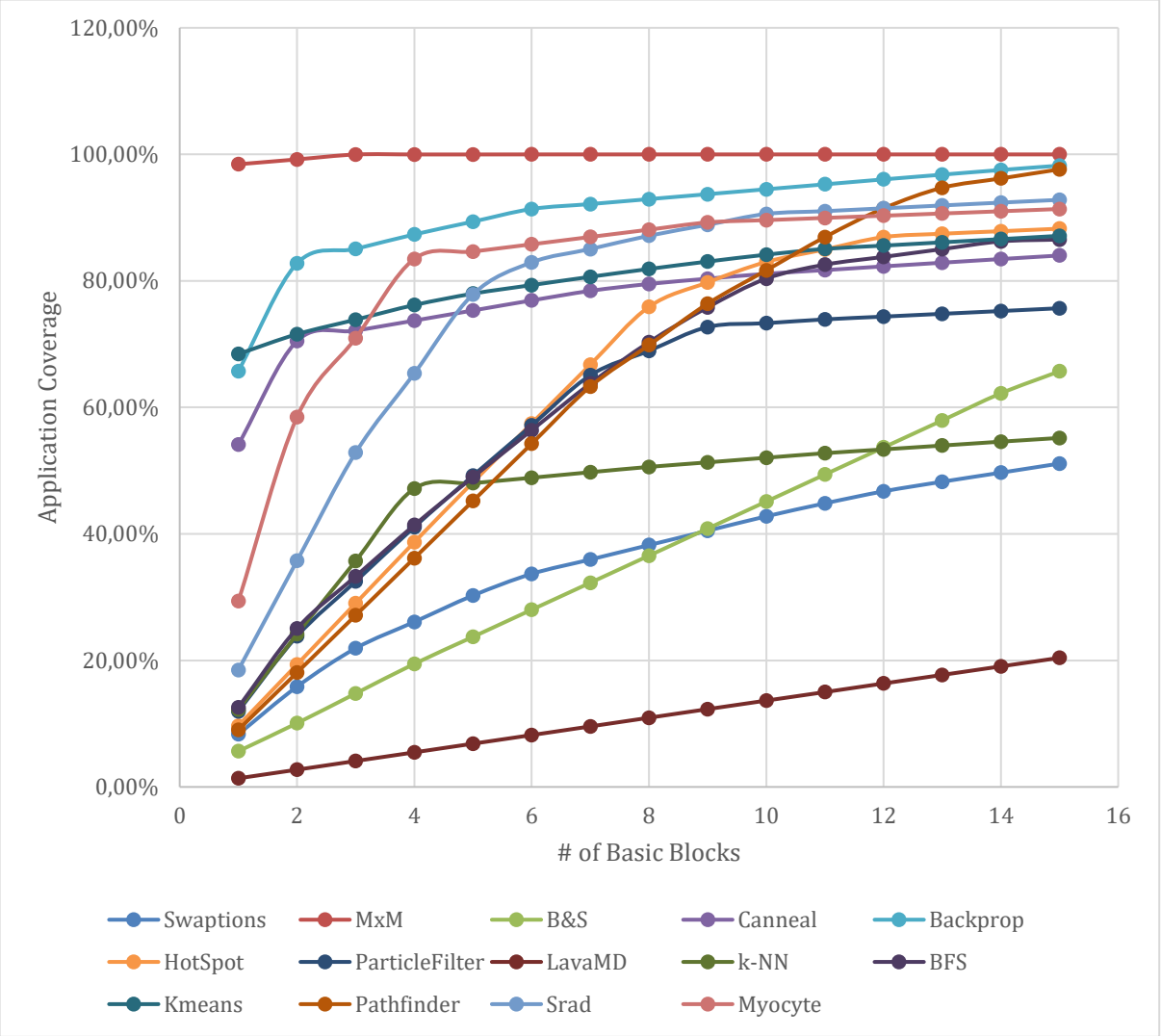
O comportamento dinâmico dos *benchmarks* indica qual o percentual total de uma aplicação é representado por um determinado número de blocos básicos. Essa caracterização dos *benchmarks* também é obtida a partir do *trace file* do gem5.
O script analisa os *traces* individuais de cada *thread*, e mantém uma lista com os quinze blocos básicos mais executados de cada *thread* da aplicação. O percentual da execução que cada número de blocos básicos representa é obtido através divisão pelo número total de blocos básicos de cada *thread*.

Na Figura 5.5, ordenamos os *benchmarks* incrementado o número de blocos básicos, e sua contribuição para o tempo de execução (cobertura), e mostramos quantos blocos básicos são necessários para atingirmos uma determinada taxa de cobertura.

Algumas aplicações, como MxM, possuem *kernels* bastante distintos, e um único bloco básico cobre 98% da aplicação. Por outro lado, aplicações como LavaMD possuem muitos *kernels* distintos (e em geral, pequenos em número de microoperações), e o bloco básico mais executado cobre apenas 1,37 do total da aplicação. Podemos ver que os *benchmarks* escolhidos cobrem um espectro bastante amplo de aplicações.

Do ponto de vista de aceleradores, é bastante benéfico quando poucos blocos básicos cobrem altos percentuais da aplicação. Em um acelerador reconfigurável essa característica reduz os custos de reconfiguração. Já para aceleradores específicos, isso significa que apenas um pode trazer uma maior aceleração, quando comparado a um caso em que uma aplicação é composta de vários diferentes *kernels*. Nesse caso, ou aumenta-se o número de aceleradores, aumentado o custo, ou a aceleração é baixa.

Figura 5.5 – Cobertura da aplicação, representando o comportamento dinâmico dos *benchmarks*



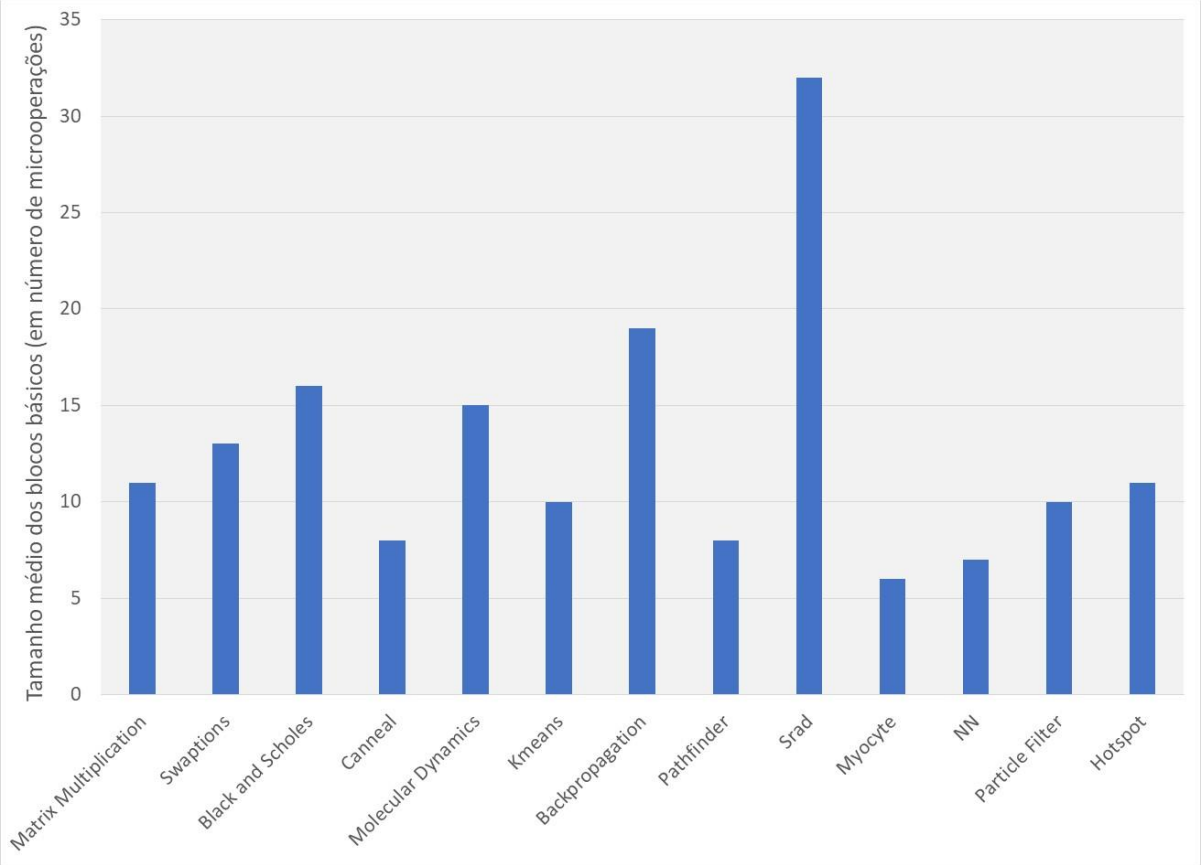
Fonte: O autor.

5.3.2 Tamanho médio dos blocos básicos dos *benchmarks*

O tamanho médio dos blocos básicos das aplicações, em número de microoperações, obtidos de acordo com a seção 5.2.3, são apresentados na Figura 5.6.

De acordo com (BRANDALERO; BECK, 2017) aplicações com blocos básicos menores são tipicamente mais difíceis de acelerar, pois precisam de um melhor mecanismo de predição de controle, como é o caso em aplicações com muitos *kernels*.

Figura 5.6 – Tamanho médio, em número de microoperações, dos blocos básicos que compõem cada um dos *benchmarks* utilizados neste trabalho



Fonte: O autor.

5.4 Configuração do sistema – Processador e Acelerador Reconfigurável

Neste trabalho, além de propormos uma nova métrica – Capítulo 4 – também realizamos uma exploração do espaço de projeto disponível em arquiteturas *multi-core* com aceleradores compartilhados. Assim, avaliamos o ganho de performance dessas arquiteturas heterogêneas, em relação a um processador *multi-core baseline*. Como o trabalho de (BRANDALERO; BECK, 2017) propunha a utilização do acelerador CGRA para o caso de um processador *single-core*, toda a implementação da infraestrutura utilizada para o compartilhamento de um acelerador entre múltiplos *cores* precisou ser realizada.

Nas próximas seções, nós apresentamos as configurações escolhidas para esse sistema, tanto em termos de processador *baseline*, acelerador reconfigurável escolhido, e combinações processador/acelerador.

5.4.1 O processador

O processador *baseline* utilizado nesse trabalho é um superescalar 8-*issue*, com arquitetura x86. Seus parâmetros são apresentados na Tabela 5.2. Essa organização é bastante semelhante à moderna microarquitetura Intel Haswell, presente em um dos últimos processadores Intel Core i7, que é implementado em 22nm (HAMMARLUND et al., 2014). Entretanto, devido a limitações do simulador, a memória *cache* de último nível (LLC) é a L2, não havendo L3 na hierarquia de memória. A *cache* L1 é privada a cada um dos *cores* do processador, e é dividida entre *cache* de instruções e *cache* de dados. A *cache* L2 é compartilhada entre os *cores*.

Tabela 5.2 – Parâmetros do processador *baseline*

Pipeline:	8-issue com execução fora-de-ordem, com 4 portas ALU, 2 portas de multiplicação, 2 portas de load e 1 porta de store. Fila de instruções: 60 µops. Buffer de Load: 72 µops. Buffer de Store: 42 µops. Entradas no ROB: 192 µops.
Caches L1 D+I:	32 kB cada cache L1 (D+I), 2 ciclos de latência (hit)
Cache L2	256 kB, 8 ciclos de latência (hit)

Fonte: O autor.

Para a exploração do espaço de projeto utilizou-se um processador *multi-core*, com microarquitetura Haswell, e 8 *cores.* A Tabela 5.3 apresenta a latência das operações executadas no processador superescalar.

Tabela 5.3 – Latência das unidades funcionais do processador superescalar OoO

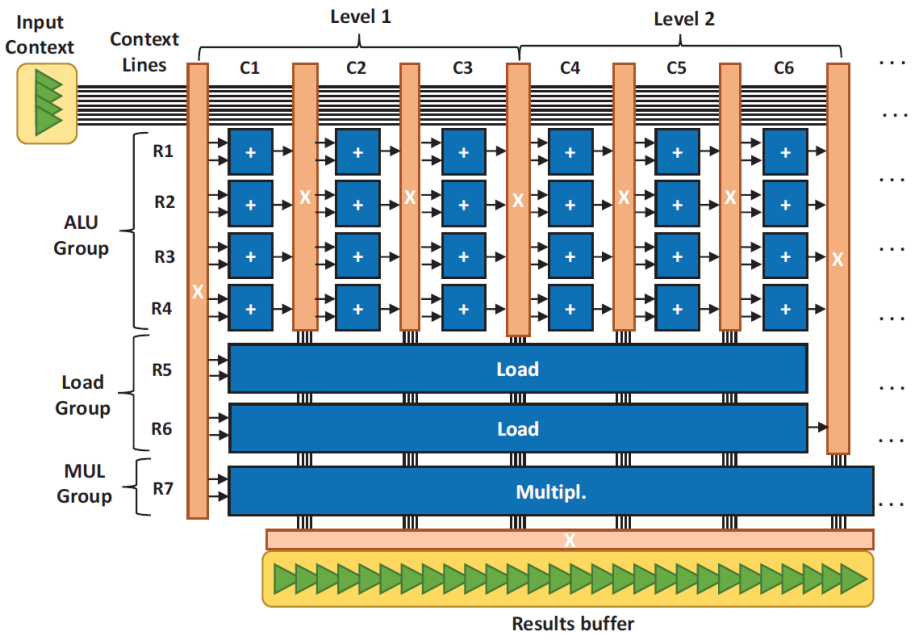
Unidade Funcional (FU)	Latência
ALUs	1 ciclo
Multiplicadores	3 ciclos
Unidades de Load	2 ciclos
Unidades de Store	1 ciclo

Fonte: O autor.

5.4.2 O acelerador

Para a obtenção dos resultados de ganho de performance, e estudo de caso da métrica proposta, o acelerador escolhido é o acelerador reconfigurável CGRA descrito na seção 2.2.2.3 desse trabalho, proposto por (BRANDALERO; BECK, 2017) para uma arquitetura x86 com um *core* único. Nesse trabalho, expandimos a ideia inicial para a utilização do CGRA em arquiteturas *multi-core*, em que os diferentes elementos de processamento podem compartilhar o CGRA, a depender de um algoritmo de alocação de recursos descrito na seção 5.4.4. A Figura 5.7 apresenta o CGRA maiores detalhes.

Figura 5.7 – CGRA (*Coarse-Grain Reconfigurable Array*) em detalhe



Fonte: O autor.

O acelerador reconfigurável é uma matriz heterogênea de unidades funcionais (FU), composta de lógica combinacional e dividida em linhas e colunas. Dados propagam da esquerda para a direita, de forma que cada FU ocupa uma linha, e uma sequência de colunas, dependendo da latência da operação. As unidades mais simples da matriz são as unidades lógica/aritméticas (ALUs), que levam um terço de ciclo de um processador superescalar na nossa implementação, e correspondem a uma única coluna. As latências das operações, e distribuição das unidades funcionais podem ser customizadas dependendo da tecnologia de implementação, e das restrições de projeto. A latência das unidades funcionais da nossa implementação são descritas na Tabela 5.4.

Tabela 5.4 – Latência das unidades funcionais do CGRA

Unidade Funcional (FU)	Latência
ALUs	1/3 ciclo
Multiplicadores	3 ciclos
Unidades de Load	2 ciclos
Unidades de Store	1 ciclo

Fonte: O autor.

Em nossa implementação do CGRA, um bloco básico é considerado acelerável pelo algoritmo quando ele atende a três requisitos básicos:

- O bloco básico não possui instruções que não são suportadas pelo acelerador reconfigurável;
- O bloco básico não possui instruções não-suportadas, e uma configuração correspondente já existe na Cache de Configuração do acelerador;
- O CGRA executa a configuração em menor número de ciclos, quando comparado com a execução do bloco básico, ou conjunto de blocos básicos, no processador superescalar.

Se algum desses requisitos não é atendido, esses blocos básicos são sempre executados no processador superescalar, e configurações não são criadas para o CGRA. O tempo de reconfiguração do CGRA foi considerado nulo na implementação.

Para maximizar o ILP, o CGRA ainda explora uma forma de especulação de controle. O algoritmo especula em uma sequência de blocos básicos que serão executados mapeando múltiplos blocos básicos na mesma configuração, permitindo exploração de ILP entre fronteiras de controle. O número de blocos básicos por configuração é chamado de *trace length*, que define o quão agressiva é a especulação. Essa característica do acelerador é particularmente interessante no caso de compartilhamento do acelerador entre múltiplos *cores*, como veremos no Capítulo 6. Baseado no trabalho de (BRANDALERO; BECK, 2017), definimos um *trace length* constante, de 6 blocos básicos por configuração.

O acelerador reconfigurável e a *cache* de configuração possuem áreas de 4,18mm² e 1,34mm² respectivamente. Um *core* superescalar com a organização apresentada na seção 5.4.1 ocupa uma área de 13,94mm². A Tabela 5.5 apresenta o acréscimo de área resultante da adição de um, dois, quatro e oito aceleradores ao processador *multi-core*. O acréscimo de área foi calculado em relação a um processador Intel i7-5960X, que possui 8 *cores*, apresenta uma organização Haswell, e ocupa uma área de 355mm² ("Intel Core i7 Processor Family for LGA2011-v3 Socket Datasheet").

Tabela 5.5 – Acréscimo de área resultante da adição de 1, 2, 4 e 8 aceleradores a um processador com 8 *cores* com organização Haswell

Número de CGRAs acrescidos ao processador superescalar	*Overhead* de Área
1	1,55%
2	3,11%
4	6,22%
8	12,44%

Fonte: O autor.

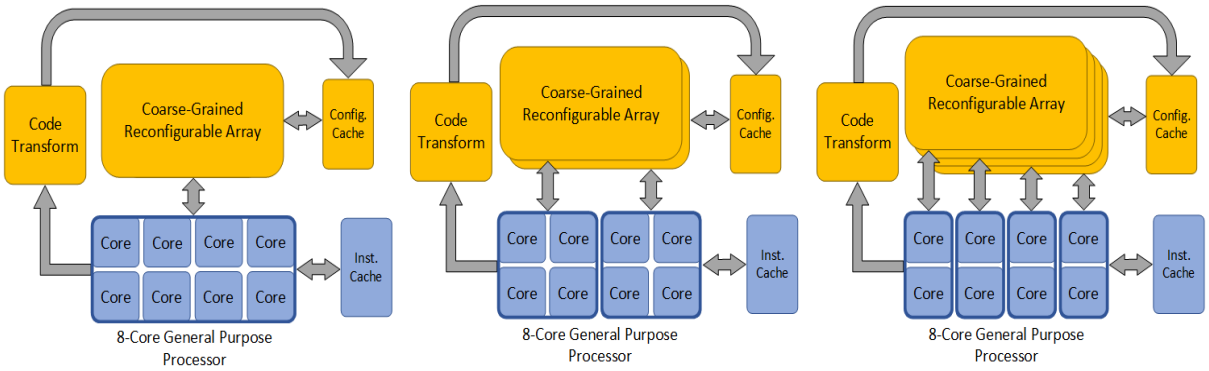
5.4.3 Configurações do sistema

Os resultados de ganho de performance foram obtidos através de configurações pré-definidas do sistema, para uma análise apropriada do espaço de projeto disponível para o compartilhamento de aceleradores em ambientes *multi-core*.

O processador foi mantido constante, sendo o processador *multi-core* com arquitetura x86, organização Haswell, e com 8 *cores*, apresentado na seção 5.4.1. Foram criadas quatro configurações distintas: processador compartilhando 1, 2 e 4 aceleradores CGRA, além da

configuração de implementação de aceleradores dedicados para cada *core*. A Figura 5.8 mostra as configurações de sistema implementadas, em que existe compartilhamento do acelerador, e a Figura 5.9 mostra a configuração implementada em que existe um acelerador dedicado por *core*.

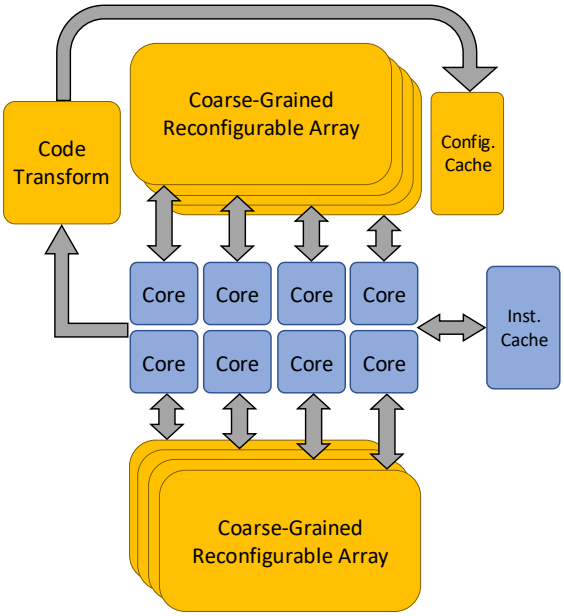
Figura 5.8 – Configurações implementadas em que existe compartilhamento do acelerador reconfigurável entre os *cores* do processador



Fonte: O autor.

A implementação dos blocos de Transformação de Código (CT) e *Cache* de Configuração (CC) foi realizada em alto nível, e não foram modelados os acessos simultâneos que podem ocorrer a esses elementos.

Figura 5.9 – Configuração implementada em que existe um acelerador reconfigurável dedicado para cada *core* do processador



Fonte: O autor.

5.4.4 Escalonamento do acelerador implementado

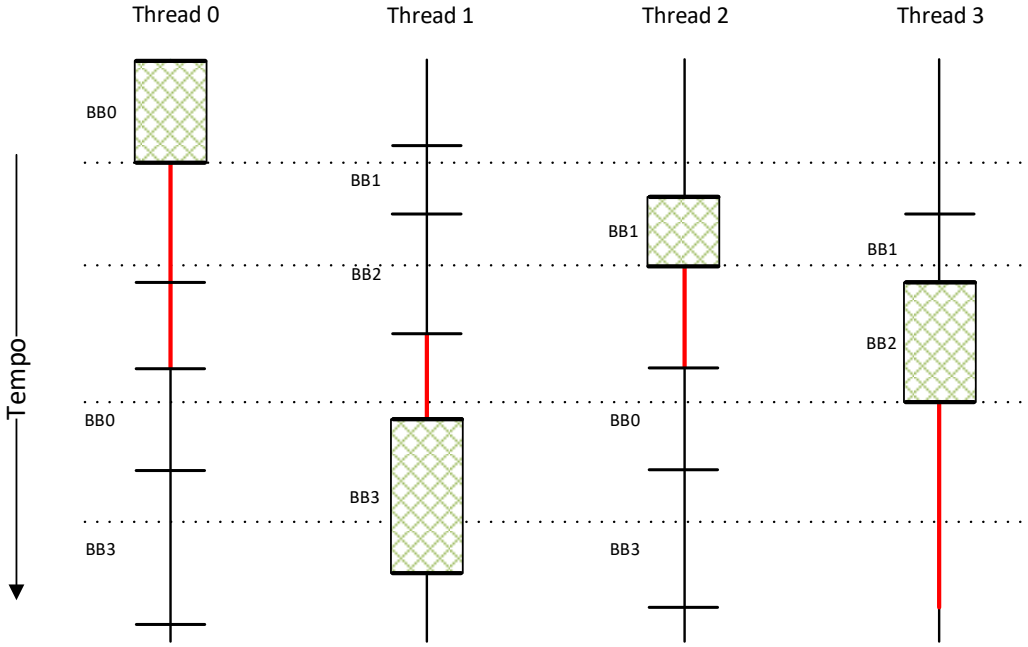
O compartilhamento do acelerador entre os *cores* foi implementado através do algoritmo de alocação *First-Come First-Serve* (SILBERSCHATZ; GALVIN; GAGNE, 2008). Em tempo de execução o algoritmo é responsável por varrer as *threads* ativas da aplicação, e encontrar o próximo bloco básico que pode ser acelerado pelo acelerador reconfigurável. Como o objetivo desse trabalho não era avaliar o desempenho de escalonadores específicos, escolhemos um de fácil implementação, que pudesse maximizar a utilização do acelerador. A performance do algoritmo FCFS mostou-se melhor que a utilização de um escalonamento *Round-Robin*. Nesse caso, como o acelerador permanece alocado um intervalo de tempo fixo para uma determinada *thread*, ocorre que a *thread* pode estar executando naquele momento um bloco básico (ou conjunto de blocos básicos) não-acelerável, o que acaba sub-utilizando o acelerador.

O acelerador só pode acelerar um bloco básico a partir do seu ponto de entrada. O PC da primeira instrução do bloco básico é armazenado na Cache de Configuração (CC), juntamente com o número de ciclos que o CGRA leva para executar a configuração referente ao bloco básico analisado, ou ao grupo de bloco básicos, caso exista especulação de controle. Ao encontrar um bloco básico acelerável, o acelerador é reconfigurado com a configuração correspondente, e o ganho de performance é a diferença entre número de ciclos que um bloco básico leva para executar no processador superescalar e no acelerador reconfigurável.

Ao final da execução no CGRA, o acelerador vai para o estado disponível, e o algoritmo passa a buscar o próximo bloco básico que pode ser executado no CGRA, com a configuração correspondente. A Figura 5.10 ilustra o processo de compartilhamento de um acelerador entre 4 *threads*.

Na Figura 5.10, a *thread* 0 está utilizando o acelerador para executar BB0. Ao final da execução, o algoritmo passa a analisar qual o próximo bloco básico, e *thread* correspondente, que irá utilizar o acelerador. Como o próximo bloco básico da *thread* 0 não é acelerável, o acelerador deve esperar até o início do bloco básico 1 na *thread* 2 para ser utilizado novamente. Pode-se perceber aqui que, quanto maior o número de blocos básicos não-aceleráveis, mais ocioso ficará um recurso (acelerador) do processador.

Figura 5.10 – Escalonamento do acelerador entre múltiplos elementos de processamento, utilizando um algoritmo FCFS.



Fonte: O autor.

Para o caso em que o número de *threads* é igual ao número de aceleradores disponíveis, cada acelerador é alocado para uma *thread*. Nesse caso, a TLP da aplicação está diretamente ligada ao tempo de ociosidade dos aceleradores.

Para as outras configurações previstas na seção 5.4.3, em que existem 8 *threads* e 2 ou 4 aceleradores, os aceleradores são alocados para conjuntos de threads. Ou seja, quando existe uma configuração com 8 *threads* e 2 aceleradores, para cada conjunto de 4 *threads* é alocado um acelerador; no caso de uma configuração com 8 *threads* e 4 aceleradores, para cada conjunto de 2 *threads* é alocado um acelerador.

5.5 Ganho de performance em aplicações multithread

Neste trabalho, trabalhamos somente com aplicações *multithread*. O cálculo do ganho de performance de aplicações *multithread* diferem das aplicações de *thread* única, principalmente porque a maior parte dessas aplicações utilizam barreiras de sincronismo. Além disso, o tempo de execução da aplicação deve levar em conta a última *thread* a finalizar. Nessa seção vamos analisar o ganho de performance de aplicações *multithread* com e sem barreiras

de sincronização. Essa metodologia de cálculo é posteriormente utilizada para o cálculo de ganho de performance no Capítulo 6.

Em geral, o cálculo de ganho de performance de uma aplicação, de acordo com (HENESSY; PATTERSON, 2011) é dado por:

$$S = \frac{T_{w0}}{T_w}$$

Equação 5.2: Cálculo de aceleração de uma arquitetura modificada (com inclusão de aceleradores) em relação a uma arquitetura *baseline*

Na Equação 5.2, S é o ganho de performance da aplicação, T_{w0} é o tempo de execução da aplicação sem aceleração, e T_w é o tempo de execução da aplicação com aceleração. Devido à nossa implementação do acelerador, os resultados da execução da aplicação são obtidos em número de ciclos, e a Equação 5.2 permanece válida. Para o ganho de performance percentual temos:

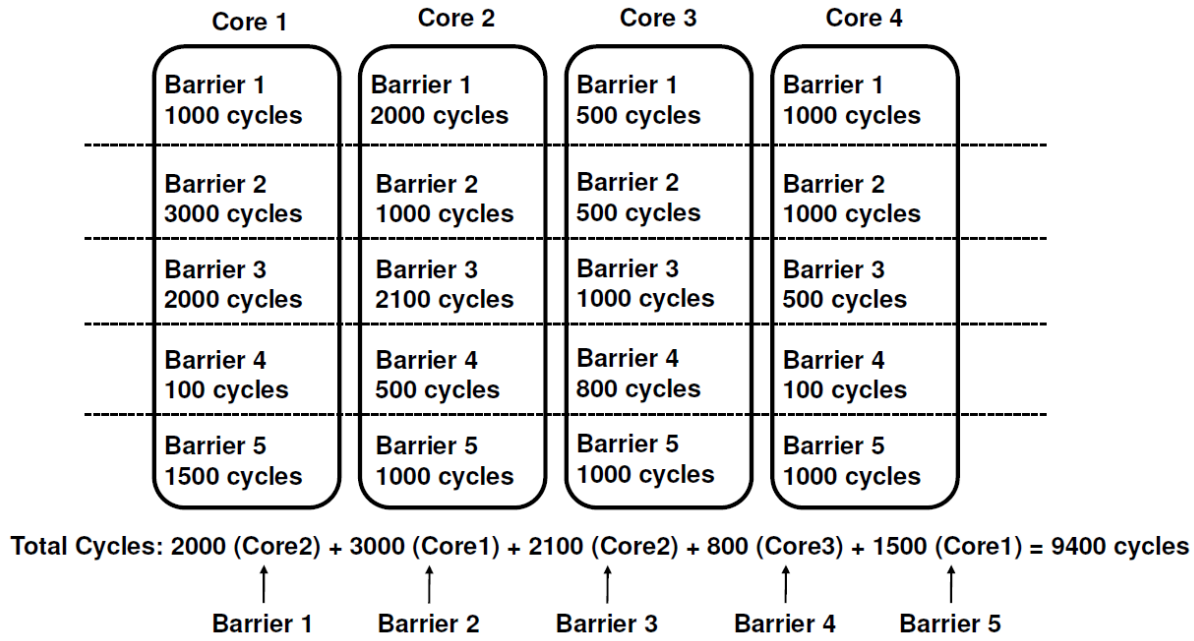
$$S(\%) = \left(\frac{numCiclos_{w0}}{numCiclos_w} - 1\right).100$$

Equação 5.3: Cálculo do ganho de performance percentual de uma aplicação, executada em duas diferentes arquiteturas

Em aplicações em que um determinado número de tarefas deve ser completado antes que a aplicação possa continuar, barreiras de sincronização são utilizadas. No caso da biblioteca PThreads, as barreiras devem ser explicitamente inseridas no código, através de métodos da biblioteca. No caso da OpenMP, essas barreiras são gerenciadas pela biblioteca, não dependendo da intervenção do programador. O método especifica o número de *threads* que devem sincronizar na barreira, e a execução só é resumida quando a última *thread* alcança a barreira.

A Figura 5.9 ilustra o processo para calcular o número de ciclos total que uma aplicação multithread com pontos de sincronização levou para exececutar.

Figura 5.11 – A cada barreira, o *core* que levou o maior número de ciclos é considerado, pois os outros devem aguardar na barreira o *core* mais lento alcançar o ponto de sincronização.



Fonte: (SOUZA, 2016)

Como realizamos a implementação da aceleração em *hardware* baseada no *trace file* do gem5, um *script* é responsável por encontrar a nova posição das barreiras de sincronismo, e calcular a aceleração das aplicações. Isso porque, as *threads* não são aceleradas uniformemente, tanto pelas características das aplicações, como pelo compartilhamento dos aceleradores, as barreiras de sincronismo de cada *thread* sofrem um deslocamento diferentes entre si. A partir da obtenção das novas barreiras de sincronismo, pós-aceleração, o *script* calcula o tempo de execução da aplicação. A aceleração final da aplicação é dada então pela Equação 5.2.

5.6 Correlação

No capítulo 6, utilizamos a correlação entre conjuntos de dados para as mais diversas finalidades. A correlação utilizada nessa dissertação é o Coeficiente de Pearson, ou simplesmente, Coeficiente de Correlação, definida na Equação 5.4.

$$r = \frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{S_x}\right)\left(\frac{y_i - \bar{y}}{S_y}\right)$$

Equação 5.4: Coeficiente de Pearson

Assumindo dois conjuntos de dados *x* e *y*, x_i e y_i são os *i* dados de seus respectivos conjuntos. $\bar{x}$ e $\bar{y}$ correspondem às médias aritméticas dos conjuntos de dados *x* e *y*. S_x e S_y são os devios padrões dos conjuntos de dados, e *n* é o número de dados de cada um dos conjuntos (os dois conjuntos devem ter o mesmo número de dados).

O coeficiente de correlação varia entre -1 e 1. O valor de 1 implica que uma equação linear crescente descreve a relação entre os conjuntos de dados, de forma que quando o valor dos dados no eixo x aumentam, o valor dos dados no eixo y aumentam proporcionalmente. O valor de -1 implica que uma equação linear decrescente descreve a relação entre os conjuntos de dados, mas quando os valores no eixo x aumentam, os valores para o eixo y descrescem. Um valor de zero, ou próximo a esse valor, indica que não existe uma relação entre os dados nos conjuntos.

6 RESULTADOS

Neste capítulo, apresentamos os resultados obtidos para a métrica proposta no Capítulo 4. Inicialmente, mostraremos que não existe correlação entre o paralelismo no nível das threads e nossa métrica proposta, validando o fato de que ambas não avaliam a mesma característica da aplicação. Na seção 6.2 iremos apresentar os resultados de ganhos de performance obtidos com o compartilhamento de um acelerador reconfigurável, entre múltiplos elementos de processamento, nas diversas configurações propostas na seção 5.4.3. Essa seção é utilizada como um estudo de caso, para uma posterior correlação entre a nossa métrica com o que definimos como oportunidade de aceleração de uma aplicação. Por fim, analisamos o impacto das diferentes latências de LLC na performance das aplicações, e na métrica proposta.

6.1 TLP e SACL

A TLP é a principal métrica utilizada para a avaliação do nível de utilização de recursos paralelos em sistemas *multi-core*. Após a obtenção da métrica proposta para o conjunto de *benchmarks* é imperativo que a correlação entre a TLP e SACL seja estabelecida. Isso porque caso haja uma forte correlação entre as métricas, seja positiva ou negativa, um projetista que quisesse integrar aceleradores em uma determinada arquitetura poderia simplesmente utilizar a TLP, uma métrica já existente e consolidada na literatura. Já a ausência de correlação mostraria que a métrica proposta, apesar de semelhante à TLP em termos de definição matemática, mede uma característica diferente da aplicação.

A Tabela 6.1 relaciona os valores obtidos para a TLP e SACL de cada um dos *benchmarks* relacionados na seção 5.3. Para o conjunto de *benchmarks* utilizado, o coeficiente de correlação (Equação 5.4) entre TLP e a métrica proposta é de 0,44, o que pode ser considerada uma fraca correlação entre os dois conjuntos de dados. Uma fraca correlação significa que não existe uma clara associação entre os conjuntos, tanto em questão de magnitude, como em questão de direção da relação entre ambos.

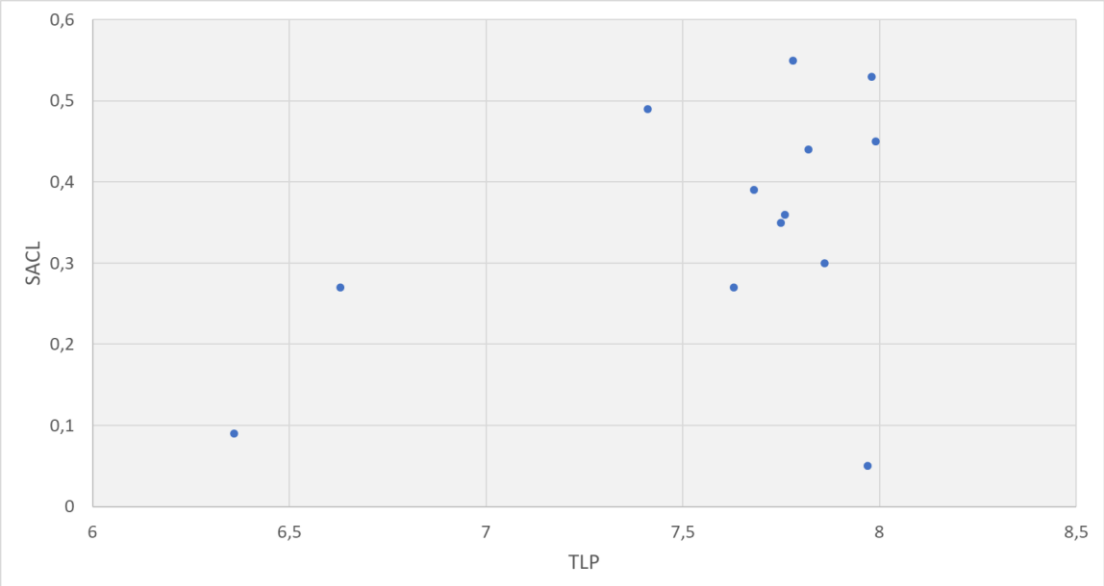
Tabela 6.1 – Valores de TLP e SACL calculados para o conjunto de *benchmarks*

Benchmark	TLP	SACL
Blackscholes	7,98	0,53
Swaptions	6,63	0,27
Canneal	7,99	0,45
MxM	7,68	0,39
Kmeans	7,41	0,49
Molecular Dynamics	7,78	0,55
Backpropagation	7,63	0,14
Pathfinder	7,75	0,35
Srad	7,97	0,05
Myocyte	7,76	0,36
NN	7,82	0,44
Particle Filter	7,86	0,30
Hotspot	6,36	0,09

Fonte: O autor.

A Figura 6.1 mostra o gráfico de dispersão dos dados obtidos para a SACL (eixo-y) e TLP (eixo-x).

Figura 6.1 – Dispersão dos valores obtidos para SACL e TLP



Fonte: O autor.

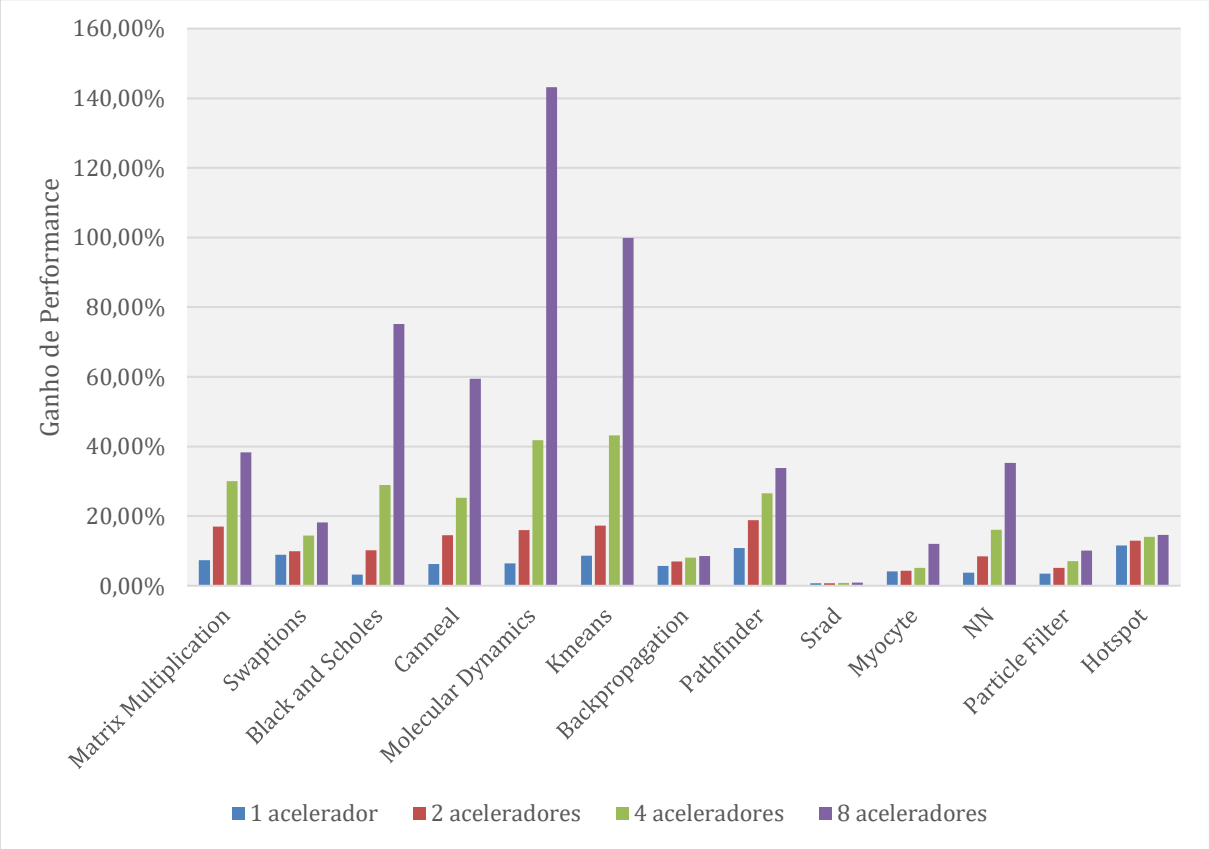
Fica claro a partir da Figura 6.1 que não existe uma linha de tendência que possa ser traçada para os dados da dispersão, que os coloque em uma ordem estritamente monotônica. Assim, com o valor da correlação obtido para os dados de SACL e TLP, em conjunto com o gráfico de dispersão desses dados, nós assumimos que as métricas não avaliam a mesma característica da aplicação.

6.2 Avaliação de Ganho de Performance

Para validarmos a utilidade da métrica proposta, executamos as aplicações listadas na seção 5.3 no sistema da seção 5.4, para todas as configurações implementadas, de acordo com a seção 5.4.3, e obtivemos dados de aceleração para cada *benchmark*. Utilizaremos os dados obtidos na seção 6.3, em que mostraremos a correlação entre a oportunidade de aceleração de um *benchmark* (uma métrica definida neste trabalho, calculada a partir dos dados de aceleração) e a métrica proposta.

Figura 6.2 apresenta o ganho de desempenho (em número de ciclos) obtida com o compartilhamento do acelerador reconfigurável entre os 8 *cores* do processador. Para cada *benchmark*, foi realizada a comparação de ganho de performance para 1, 2, 4 e 8 aceleradores incluídos no sistema. O eixo-X mostra os *benchmarks* avaliados, para cada número de aceleradores no sistema, e o eixo-Y é o ganho de performance percentual obtido para cada *benchmark*, e cada configuração de sistema.

Figura 6.2 – Comparação de performance para cada *benchmark*, para um sistema com 1, 2, 4 e 8 aceleradores, compartilhados pelos *cores* de um processador *multi-core*

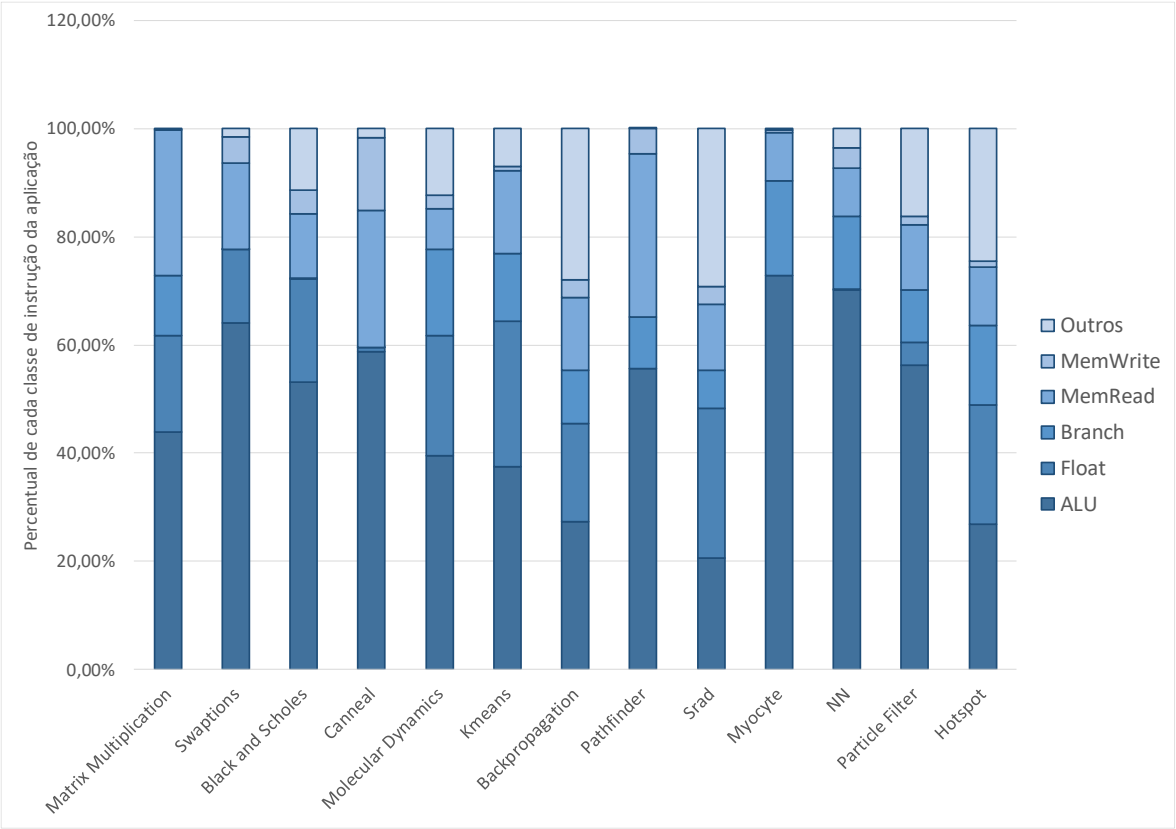


Fonte: O autor.

O *benchmark* que teve a maior aceleração quando compartilhando apenas um acelerador entre os *cores* foi Hotspot, atingindo um ganho de 11,60%. Já o *benchmark* com menor aceleração quando compartilhando um acelerador entre os *cores* foi Srad, com apenas 0,71% de ganho no tempo de execução. Entretanto, o cenário é diferente quando consideramos a configuração com 8 aceleradores. Nesse caso, Molecular Dynamics, com uma aceleração de 143,18%, é o *benchmark* com o maior ganho entre os avaliados, e Srad, com aceleração de apenas 0,87% também apresenta o menor ganho nessa configuração.

A Figura 6.3, em conjunto com a Figura 6.4, nos ajuda a entender a aceleração das aplicações avaliadas. A Figura 6.3 mostra o percentual das classes de instruções executadas no processador, em relação ao total de instruções executadas. No eixo-X temos os *benchmarks* avaliados, e no eixo-Y, o percentual de cada uma das classes consideradas: instruções executadas na ALU, instruções do tipo *float*, instruções de desvio, operações de memória, leitura e escrita, e outras instruções. Já a Figura 6.4 mostra o percentual do total de blocos básicos que foram executados no processador e no CGRA, para cada configuração considerada.

Figura 6.3 – *Breakdown* das classes de operações executadas para cada um dos *benchmarks* avaliados

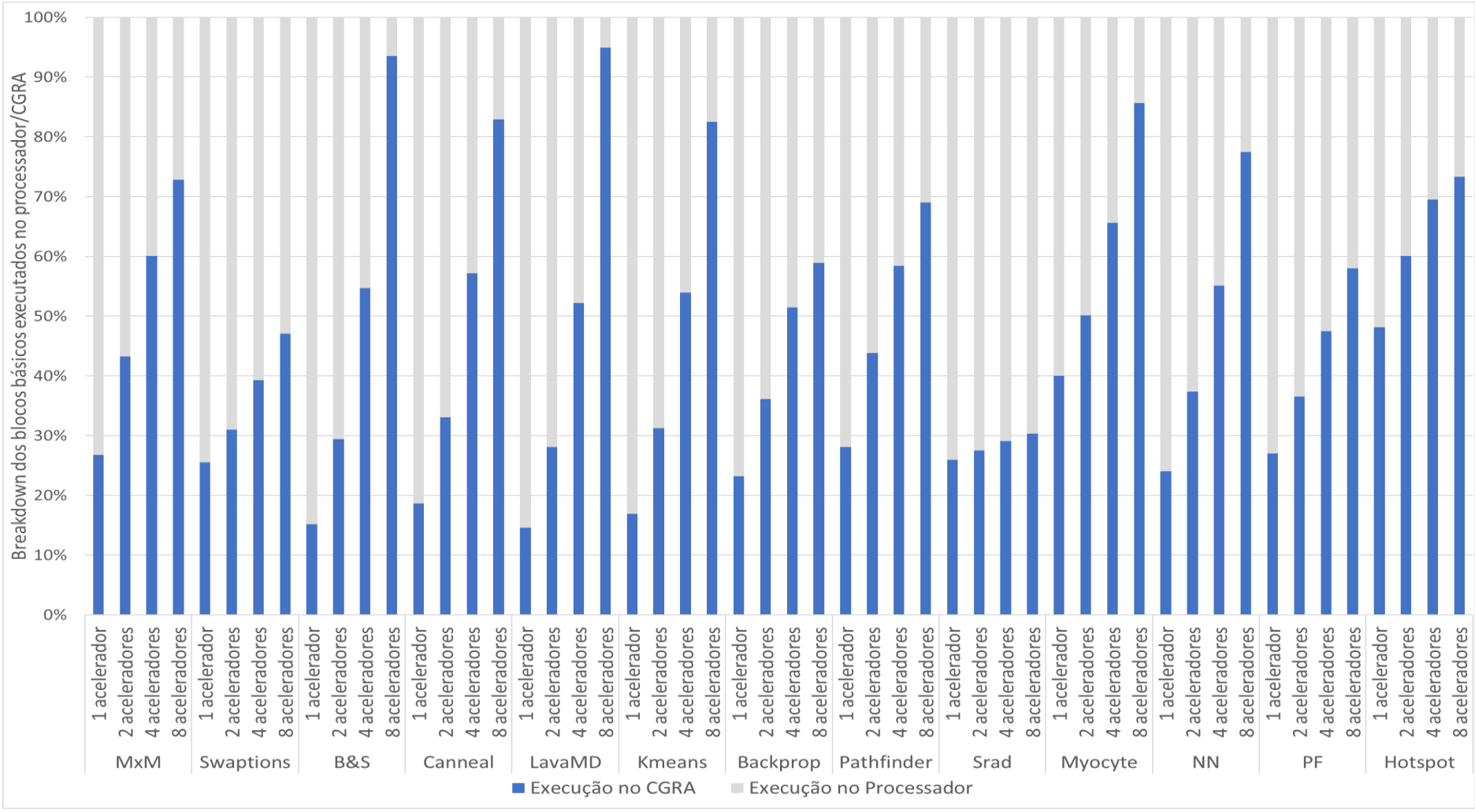


Fonte: O autor.

No geral, as aplicações com altos ganhos de performance apresentam uma combinação de alto percentual de instruções executadas na ALU, e um baixo percentual de acesso a memória. O CGRA, como já mostrado por (BRANDALERO; BECK, 2017), leva vantagem sobre o processador superescalar exatamente em aplicações com alto percentual de instruções ALU, pois são instruções executadas em um terço de ciclo do processador. Já aplicações com alto percentual de instruções de acesso memória, em geral, apresentam baixa aceleração, pois esses acessos não podem ser acelerados, tornando-se caminhos críticos da aplicação.

Por exemplo, como vimos na Figura 6.2, o *benchmark* Srad apresentou a pior aceleração entre os avaliados. Podemos ver na Figura 6.3 que apenas 20,62% das instruções são do tipo ALU, e cerca de 15% das instruções são de acesso a memória, entre leituras e escritas. Soma-se a isso o fato de que, como vemos na Figura 6.4, um baixo percentual da aplicação é executado no CGRA, para qualquer número de aceleradores avaliados. Dois motivos principais explicam isso: um alto percentual de acessos a memória resultam em *cache miss*, e muitas instruções do *benchmark* Srad não são suportadas para execução no acelerador.

Figura 6.4 – *Breakdown* do número de blocos básicos, em relação ao total de blocos básicos da aplicação, executados no processador *baseline*, e executados no acelerador reconfigurável CGRA, para cada um dos *benchmarks*, e para cada uma das configurações (1, 2, 4 e 8 aceleradores)



Fonte: O autor

As aplicações com alta aceleração, como *LavaMD*, *BlackScholes*, e *Kmeans*, por outro lado, apresentam altos percentuais de instruções do tipo ALU, e baixos percentuais de instruções de acesso à memória. Em *LavaMD*, apenas 7,56% das instruções acessam a memória, o mais baixo percentual desse tipo de instrução entre todos os avaliados. Além disso, essas aplicações atingem um alto percentual de execução no CGRA: mais de 85% do total dos blocos básicos são executadas no acelerador, indicando que essas aplicações possuem baixo *cache miss*.

Outra característica que podemos avaliar em relação aos resultados apresentados na Figura 6.2, em conjunto com a Tabela 6.1, é que não existe uma clara correlação entre o ganho de performance e a TLP das aplicações, exceto quando os *cores* do processador compartilham apenas um acelerador. Entretanto, mesmo nesse caso, a correlação é de -0,70. Ou seja, em aplicações com baixo TLP, em que existe uma menor competição pelo acelerador, existe um certo nível de correlação com a aceleração. Mesmo nesse caso, entretanto, a correlação não é alta. O *benchmark Kmeans,* por exemplo, está entre as aplicações com maior paralelismo à nível de *threads*, e ainda sim, apresenta uma alta aceleração. *Srad*, por outro lado, apresenta alto TLP (7,97), e mesmo assim, apresenta os menores níveis de aceleração para todas configurações.

Para os outros casos, com 2, 4 e 8 aceleradores, a correlação é muito próxima de zero. Nesses casos, não é possível inferir nenhum tipo de relação entre aceleração e TLP da aplicação, em um processador *multi-core*, com aceleradores compartilhados.

Por fim, na Tabela 6.2, apresentamos a média de aceleração para o conjunto de *benchmarks,* para 1, 2, 4 e 8 aceleradores.

Tabela 6.2 – Média de aceleração do conjunto de *benchmarks* para cada uma das configurações avaliadas.

1 acelerador	2 aceleradores	4 aceleradores	8 aceleradores
6,23%	10,94%	20,12%	42,25%

Fonte: O autor.

A partir das Figuras 6.2, 6.3 e 6.4, em conjunto com a Tabela 6.1, podemos sumarizar as seguintes conclusões:

- A aceleração das aplicações, utilizando-se um acelerador CGRA compartilhado, é dependente de haver um alto percentual de instruções do tipo ALU, e um baixo percentual de instruções de acesso à memória;
- A aceleração das aplicações depende do percentual de acessos à memória que resultaram em *cache miss*. Quanto maior o percentual de *cache miss*, menor necessariamente será a aceleração da aplicação. Isso não é específico para o nosso sistema, sendo uma característica comum a sistemas com aceleradores;
- A aceleração das aplicações não depende do paralelismo em nível de *threads* (TLP), exceto no caso em que os *cores* do processador compartilham apenas um acelerador. Nesse caso, existe um certo nível de correlação entre ambas.

6.3 SACL e a oportunidade de aceleração

Nesta seção iremos correlacionar a SACL com o que nós definimos como oportunidade de aceleração. A oportunidade de aceleração de uma aplicação é o ganho de performance médio adicional obtido, cada vez que o número de aceleradores do sistema é dobrado. Formalmente:

$$OA = \frac{\sum_{i=0}^{(\log_2 n)-1} S_{(2^i)(2^{i+1})}}{\log_2 n}$$

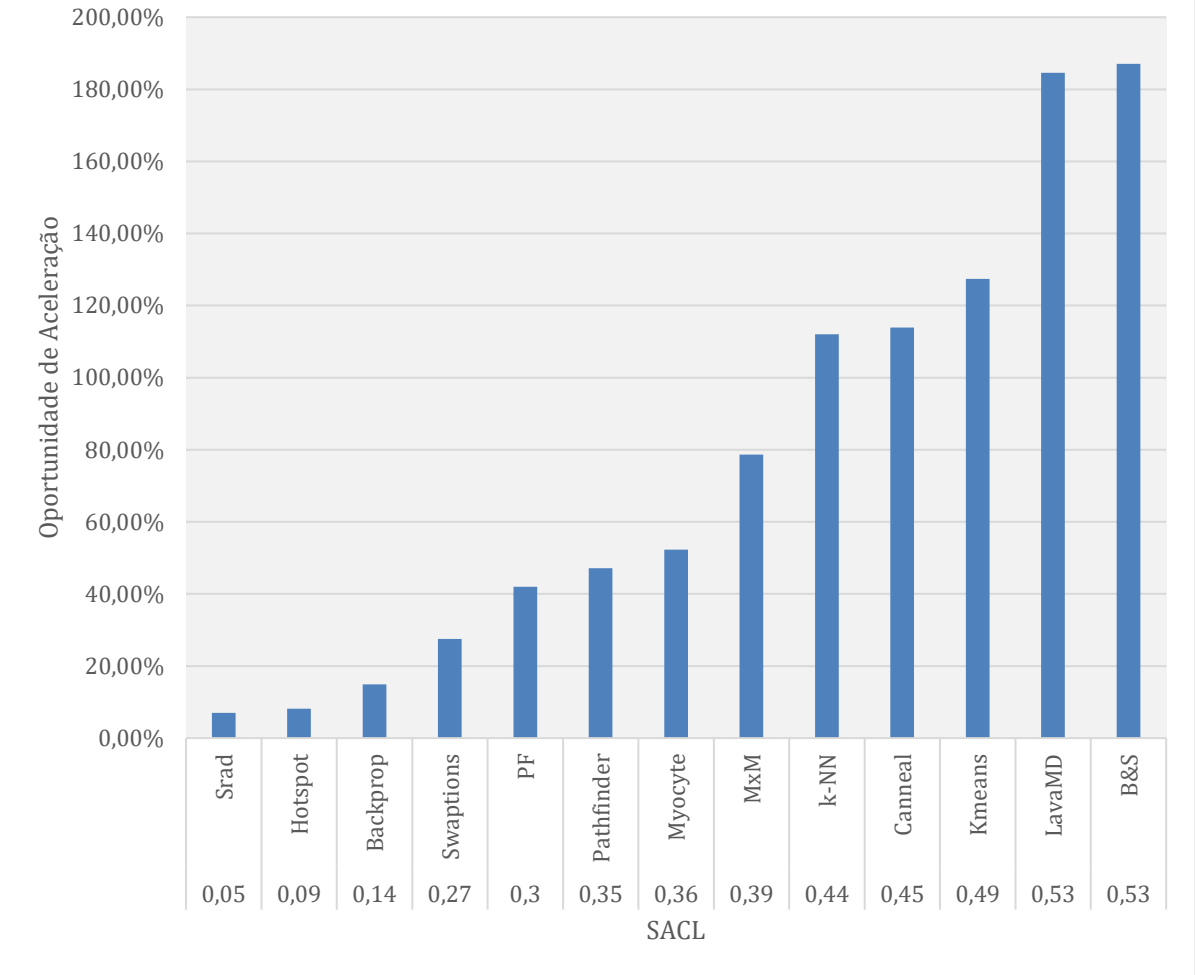
Equação 6.1: Oportunidade de Aceleração de aplicaçòes *multithread* executando em arquiteturas *multi-core* com aceleradores compartilhados

A OA considera apenas potência de dois número de aceleradores no sistema. Na Equação 6.1, n é o número de *threads* ativas, e cada termo $S_{(2^i)(2^{i+1})}$ é a aceleração obtida quando o número de aceleradores é aumentado de 2^i para 2^{i+1}. O somatório é então dividido pelo número de vezes que é possível dobrar o número de aceleradores, de acordo com o número de *threads*, e considerando que n é o maior número de aceleradores possível no sistema. A oportunidade de aceleração não deve ser confundida com o ganho de desempenho absoluto das aplicações, característica avaliada na seção 6.2. A OA foi uma maneira que encontramos de

agrupar os dados de desempenho obtidos com as configurações propostas, para posteriormente, correlacionarmos com a métrica proposta. Por exemplo, uma aplicação pode ter uma alta oportunidade de aceleração, mas o ganho de performance absoluto da aplicação pode ser baixo.

A partir da definição, podemos ver que OA indica quanto, efetivamente, um sistema se beneficia ou não da integração de aceleradores adicionais no sistema. Por exemplo, considerando o caso com 8 *threads*, se não existe ganho de performance quando aumenta-se o número de aceleradores de 1 para 2, 2 para 4, e 4 para 8, então a OA da aplicação é 0% (ou 1x). Se, por outro lado, a cada vez que o número de aceleradores no sistema é dobrado resulta uma aceleração de 100% (2x), então OA é 100%.

Figura 6.5 – Oportunidade de Aceleração para o um conjunto de *benchmarks*, ordenados em ordem crescente de SACL



Fonte: O autor.

A Figura 6.5 apresenta os resultados obtidos em forma gráfica. Nesta Figura, comparamos a oportunidade de aceleração de cada uma das aplicação, com a nossa métrica proposta, SACL. Neste gráfico, o eixo-X mostra os *benchmarks* avaliados, que estão dispostos em em ordem crescente de SACL: o *benchmark* mais da esquerda (Srad) apresentou a menor SACL entre os avaliados, enquanto o mais da direita (*Black&Scholes*) apresentou a maior SACL. O eixo-Y mostra a oportunidade de aceleração, calculada de acordo com a Equação 6.1, em percentual, de cada um dos *benchmarks*.

Fica evidente, através da Figura 6.5, que existe uma alta correlação entre a oportunidade de aceleração e a SACL. Isso significa que aplicações com baixo SACL apresentam uma baixa oportunidade de aceleração. *Benchmarks* como *Srad*, *Backpropagation,* e *Hotspot*, apresentam os menores valores de SACL entre os avaliados, e como vimos na Figura 6.2, são os *benchmarks* que menos se beneficiam da inclusão de mais aceleradores no sistema. Nesses *benchmarks* é mais vantajoso compartilhar um acelerador, economizando em termos de área e energia, do que adicionar aceleradores que não terão grande impacto na aceleração. Por outro lado, *benchmarks* como *Black&Scholes* e LavaMD possuem alta SACL, e ao mesmo tempo, uma alta oportunidade de aceleração, 187,5% e 184,59%, respectivamente. Nesses casos, através da SACL, o projetista pode estimar que a aceleração da aplicação, quando comparado com aplicações de baixo SACL, será maior com a adição de aceleradores no sistema.

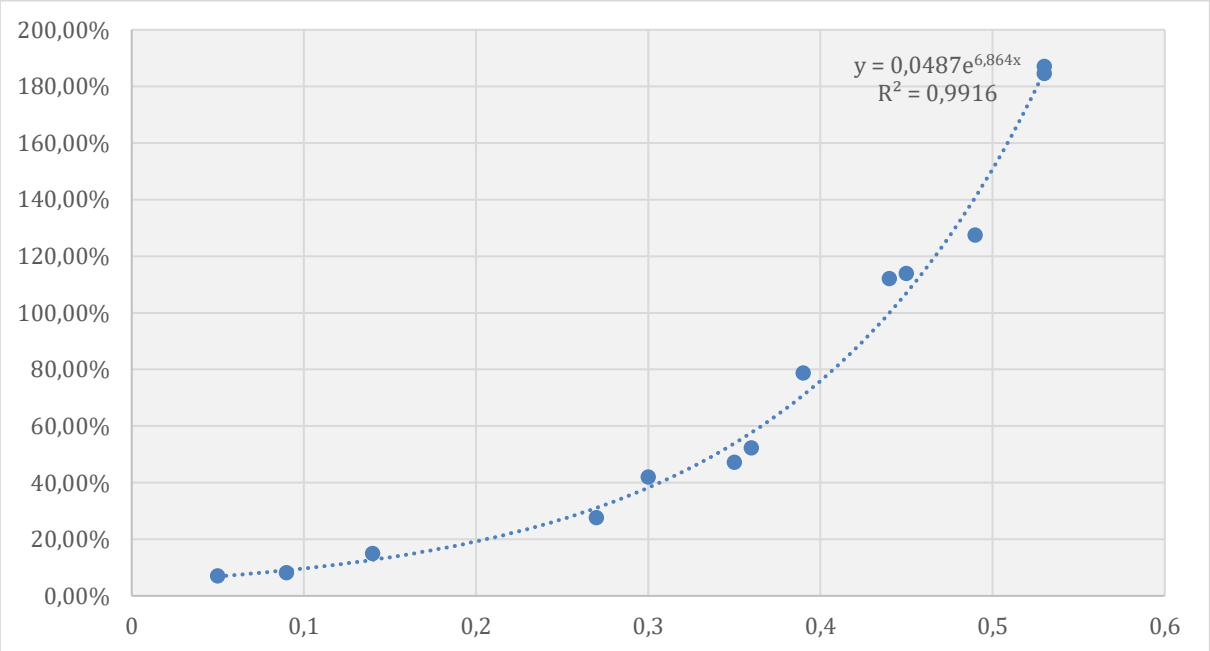
Outro exemplo bastante interessante diz respeito aos *benchmarks Hotspot* e *Swaptions*. Ambos possuem TLPs bastante similares, 6,36 e 6,63, respectivamente, mas apresentam SACLs bastante distintos, 0,09 e 0,27, respectivamente.

De acordo com o que foi apresentado no Capítulo 4, *Hotspot* possui menos blocos básicos aceleráveis executando concorrentemente, o que o torna uma aplicação com potencial de compartilhamento de um acelerador. Nesse caso, isso fica ainda mais evidente. O ganho de performance de *Hotspot* com um acelerador compartilhado é de 11,60%. *Swaptions* por outro lado, que apresenta maior SACL, apresenta uma aceleração de 8,91% quando executando nessa mesma configuração. Como *Swaptions* possui mais blocos básicos aceleráveis executando ao mesmo tempo, o ganho de performance com apenas um acelerador é mais limitado que no caso de *Hotspot*. Por outro lado, a sua oportunidade de aceleração é maior. Como vemos na Figura 6.5, *Swaptions* tem uma oportunidade de aceleração de 18,16%, enquanto *Hotspot*, de apenas 8,14%. Isso nos mostra que a SACL além de avaliar uma característica da aplicação distinta da TLP, ainda se apresenta como uma ferramenta válida para projetistas inserirem aceleradores em arquiteturas *multi-core*.

Como podemos observar também na Figura 6.5, *LavaMD* possui uma oportunidade de aceleração menor que *Black&Scholes*, mas possui uma SACL de mesmo valor. De acordo com nossa avaliação, a pequena diferença pode ser explicada pela granularidade escolhida para o cálculo da SACL. Como a média do tempo de execução dos blocos básicos é levado em conta no cálculo da métrica, isso pode levar a algumas distorções a depender das características da aplicação. Ainda, conforme explicitamos anteriormente, a oportunidade de aceleração e a SACL não são métricas de ganho de performance. *LavaMD* e *Black&Scholes* possuem o mesmo valor de SACL, e valores de oportunidade de aceleração bastante próximos, entretanto, seus ganhos de performance são totalmente distintos. A aceleração de *LavaMD* quando um acelerador é compartilhado entre os *cores* é de 6,40%, e de *Black&Scholes*, para o mesmo caso é de 3,21%. Para o caso em que aceleradores dedicados são implementados para cada *core*, a aceleração de *LavaMD* e *Black&Scholes* são, respectivamente, 143,18% e 75,14%. Ressaltamos assim, que a SACL não é uma métrica para avaliar a aceleração absoluta das aplicações, e sim para avaliar o custo benefício do compartilhamento, ou da implementação dedicada, de aceleradores entre os *cores* de um processador.

A Figura 6.6 apresenta a correlação existente entre oportunidade de aceleração, e a SACL, através de uma linha de tendência.

Figura 6.6 – Oportunidade de Aceleração para o um conjunto de *benchmarks*, ordenados em ordem crescente de SACL



Fonte: O autor.

A Figura 6.6 mostra que existe, de fato, uma alta correlação entre a oportunidade de aceleração dos *benchmarks* e a SACL. A correlação calculada é de 0,99, e a linha de tendência mostra que essa correlação é exponencial.

Por fim, estabelecemos a média aritmética da oportunidade de aceleração e da SACL para o conjunto de *benchmarks* avaliados. Com esses dados, podemos comparar mais facilmente o impacto das diferentes latências de memória LLC para o conjunto de *benchmarks.* A Tabela 6.3 apresenta esses dados.

Tabela 6.3 – Média da SACL e média da oportunidade de aceleração para o conjunto de *benchmarks* avaliados

Média da SACL	Média da Oportunidade de Aceleração
0,34	77,13%

Fonte: O autor.

6.4 Impacto da latência da LLC na aceleração e na SACL das aplicações

Para investigarmos o impacto da latência da LLC na aceleração das aplicações e na SACL, utilizamos as configurações propostas em 5.4.3, para outras duas latências de LLC: 1 ciclo e 200 ciclos de latência da LLC, em caso de *cache hit*. Define-se a latência de LLC o tempo que o processador leva para ler um dado da LLC, quando um *cache hit* ocorre neste nível de hierarquia de memória. Os *cores* de nosso processador *multi-core* compartilham a mesma LLC, e apenas um pode acessá-la em um determinado instante de tempo, o que impacta na performance geral do sistema, especialmente quando existe uma alta competição por este recurso. Quanto maior a competição para acessar a LLC, maior o tempo que os *cores* ficam aguardando, o que também impacta no acelerador reconfigurável compartilhado. Como nossa métrica mede a simultaneidade de execução de blocos básicos aceleráveis em diferentes *threads*, uma maior latência de LLC tende a espaçar temporalmente esses blocos básicos, reduzindo a competição pelo acelerador.

Após a obtenção dos dados de aceleração para diferentes latências, comparamos com os resultados da seção 6.2 e 6.3, que foram obtidos com a latência original de LLC do processador, 8 ciclos. Dessa forma, podemos avaliar o impacto da latência da LLC na aceleração, e na métrica proposta, quando aumentamos o diminuímos a latência da LLC. Nessa seção, em vez

de apresentarmos os resultados para cada um dos *benchmarks* individuais, iremos apresentar os resultados médios para o conjunto de *benchmarks.*

Na Tabela 6.4, apresentamos a média de ganho de performance para o conjunto de *benchmarks,* para 1, 2, 4 e 8 aceleradores, para as diferentes latências de LLC.

Tabela 6.4 – Comparação da média de ganho de performance do conjunto de *benchmarks* para cada uma das configurações avaliadas, e diferentes latências de memória LLC

	1 acelerador	2 aceleradores	4 aceleradores	8 aceleradores
1 ciclo	5,99%	10,40%	19,62%	43,07%
8 ciclos	6,23%	10,94%	20,12%	42,25%
200 ciclos	5,97%	9,14%	14,45%	26,30%

Fonte: O autor.

A partir da Tabela 6.4, inicialmente vemos que a aceleração do conjunto de *benchmarks* para as latências de LLC de 1 ciclo e 8 ciclos são bastante próximas. Para os casos em que existe o compartilhamento de aceleradores, a aceleração do conjunto de *benchmarks* é menor para o caso de uma latência de LLC de 1 ciclo. E para o caso em que existe um acelerador para cada *thread* no sistema, a aceleração é maior para o caso de 1 ciclo de latência de LLC.

No caso de uma latência de LLC de 200 ciclos, a média da aceleração do conjunto de *benchmarks* é claramente menor que para os casos de 1 ciclo e 8 ciclos de latência de LLC. Isso é explicado pelo fato de que conforme a latência é aumentada, mais tempo a aplicação gasta quando ocorre um cache miss, e esse tempo não é acelerável, independente do acelerador. Assim, a aceleração é menor;

Na Tabela 6.5, apresentamos a comparação entre a média da SACL do conjunto de *benchmarks,* e a média da oportunidade de aceleração do mesmo conjunto, para diferentes latências de memória LLC.

Tabela 6.5 – Comparação da média da SACL do conjunto de *benchmarks* com a média da oportunidade de aceleração do mesmo conjunto, para diferentes latências de memória LLC

	Média da SACL	Média da Oportunidade de Aceleração
1 ciclo	0,34	77,13%
8 ciclos	0,35	78,98%
200 ciclos	0,27	48,87%

Fonte: O autor.

A partir da Tabela 6.5 podemos reforçar aquilo que foi visto na seção 6.3: a SACL tem uma correlação direta com a oportunidade de aceleração, e pode funcionar muito bem para projetistas que procuram integrar aceleradores em arquiteturas *multi-core*. Fica claro que quanto maior a latência de LLC, menor é a média da SACL do conjunto de *benchmarks*, e consequentemente, menor é a média da oportunidade de aceleração. De forma análoga, vimos uma direta correlação entre menores latências de LLC e maiores SACLs.

Isso mostra a generalidade da nossa métrica. Ela pode ser utilizada para comparar *benchmarks multithread* executados em sistemas com diferentes latências de LLC, e ainda assim, a métrica fornece um panorama adequado dos custos-benefícios da inclusão (ou não) de mais aceleradores no sistema.

7 CONCLUSÃO

Este trabalho propôs uma nova métrica, SACL, capaz de prever a aceleração de aplicações em arquiteturas *multi-core* com aceleradores em *hardware* compartilhados. Para isso, utilizamos um conjunto de 13 aplicações de dois dos conjuntos de *benchmarks* mais conhecidos da literatura para a avaliação de arquiteturas *multi-core*, PARSEC e Rodinia. Essas aplicações cobrem um amplo espectro de domínios, e foram utilizados para avaliar a métrica proposta.

Nossos resultados mostram que existe correlação entre SACL e o que definimos como a oportunidade de aceleração de uma aplicação. A oportunidade de aceleração é uma medida da média do quanto a aceleração de uma aplicação aumenta, cada vez que o número de aceleradores do sistema é multiplicado por dois. Nossa métrica avalia o percentual de uma aplicação em que blocos básicos aceleráveis executam simultâneamente em diferentes *threads* ativas no contexto. A partir de SACL, o projetista pode usar o valor obtido para prever a aceleração esperada para uma determinada aplicação, e também estabelecer o custo-benefício da adição (ou não) de novos aceleradores no sistema.

Com base nos resultados obtidos, foi possível verificar que a métrica é capaz de prever, com um alto grau de fidelidade, a aceleração potencial de aplicações executando em arquiteturas *multi-core* com aceleradores compartilhados, para configurações com 1, 2, 4 e 8 aceleradores. Aplicações com altos valores de SACL traduziram-se em altas oportunidades de aceleração, ou seja, aplicações que beneficiam-se muito da inclusão de novos aceleradores no sistema. Já aplicações com baixo SACL traduziram-se em baixas oportunidades de aceleração, ou seja, aplicações que não se beneficiam, ou beneficiam-se pouco, da adição de novos aceleradores no sistema. Nessas aplicações, uma grande parte do ganho total potencial (configuração com 8 aceleradores) já é obtido com apenas um acelerador compartilhado.

Avaliamos ainda o impacto da latência da LLC na aceleração das aplicações, e na SACL (e consequentemente, na oportunidade de aceleração das aplicações). Com essa avaliação, chegamos às seguintes conclusões:

1) Quanto maior a latência de LLC, menor é a aceleração das aplicações. Isso é explicado pelo fato de que conforme a latência é aumentada, mais tempo a aplicação gasta quando ocorre um *cache miss*, e esse tempo não é acelerável, independente do acelerador. Assim, a aceleração é menor;

2) Maiores latências de LLC traduziram-se diretamente em menores SACL, e consequentemente, em menores oportunidades de aceleração. De forma análoga, vimos uma direta correlação entre menores latências de LLC e maiores SACL.

Por fim, nossos resultados mostraram que a métrica proposta apresenta uma alta generalidade, podendo avaliar o seguinte:

1) *Benchmarks* com quaisquer características podem ser avaliados com a nossa métrica proposta. Sejam homogêneos, heterogênos, com baixa ou alta TLP, SACL foi capaz de avaliar adequadamente a aceleração de aplicações *multithread* executando em arquiteturas *multi-core* com aceleradores compartilhados;
2) Nossa métrica é independente do acelerador utilizado. Da forma como foi proposta, aceleradores de propósito específico, ou reconfiguráveis, podem ser utilizados para prever a aceleração potencial de aplicações;
3) Nossa métrica pode ser utilizada para comparar *benchmarks* executados em sistemas com diferentes latências de LLC.

7.1 Trabalhos Futuros

Como vimos, poucos trabalhos propuseram métricas para avaliar características da execução de aplicações em arquiteturas *multi-core*, outras que não sejam relacionadas com performance e energia. Esse trabalho mostrou que métricas podem ser benéficas para projetistas avaliarem, em tempo de projeto, a quantidade de aceleradores necessários em um sistema, e a aceleração esperado para uma determinada aplicação. Muito ainda pode ser explorado em relação a métrica proposta nesse trabalho:

- Arquiteturas: nesse trabalho optamos pela arquitetura x86 para a avaliação da métrica SACL. Entretanto, seu conceito é genérico o suficiente para que seja utilizada na avaliação de aplicações executando em qualquer arquitetura;
- Aceleradores: o estudo de caso desse trabalho foi realizado utilizando-se um acelerador reconfigurável de grão-grosso. Entretanto, a métrica foi proposta de uma maneira genérica. Logo, investigar os benefícios da métrica para avaliar a integração

de outros tipos de aceleradores em arquiteturas *multi-core* é um trabalho futuro necessário, para confirmar a generalidade da métrica nesse aspecto;

- Níveis de memória *cache*: a arquitetura escolhida para esse trabalho inclui apenas dois níveis de memória, *cache* L1 privada à cada um dos *cores*, e *cache* L2 compartilhada entre os *cores*. Um futuro trabalho é avaliar o impacto de diferentes níveis de memória tanto na aceleração das aplicações, como na SACL e oportunidade de aceleração. Dessa forma, seria possível avaliar se a SACL fornece um panorama adequado de oportunidade de aceleração mesmo entre aplicações executadas em sistemas com diferentes níveis de memória cache;
- Latências de memória LLC: para trabalhos futuros, outras latências de memória LLC poderiam ser utilizadas, e ainda, avaliar o impacto da latência da DRAM na aceleração e na oportunidade de aceleração das aplicações.

REFERÊNCIAS

RUPP, Karl. **40 Years of Microprocessor Trend Data**. Disponível em: <https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data>. Acesso em: 13 ago. 2018.

ALIOTO, M.; CONSOLI, E.; PALUMBO, G. From energy-delay metrics to constraints on the design of digital circuits. **International Journal of Circuit Theory and Applications**, v. 40, n. 8, p. 815-834, ago. 2012.

AMDAHL, G. M. Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. '67 American Federation of Information Processing Societies (AFIPS) joint Computer Conference, New Jersey, 1967. **Anais**… p. 483-485, 1967.

ANNAVARAM, M. Energy per Instruction trends in Intel Microprocessors. **Intel Technology Magazine**, 2006.

BARAT, F.; LAUWEREINS, R.; DECONINCK, G. Reconfigurable Instruction Set Processors from a Hardware/Software Perspective. **IEEE Transactions on Software Engineering**, v. 28, n. 9, p. 847-862, set. 2002.

BIENIA, C.; KUMAR, S.; SINGH, J. P.; LI, K. The PARSEC Benchmark suite: Characterization and architectural implications. 17th International Conference on Parallel Architectures and Compilation Techniques (PACT), Toronto, 2008. **Anais**… p. 72-81, 2008.

MEI, B.; LAMBRECHTS, A.; VERKEST, D.; MIGNOLET, J. Y.; LAUWEREINS, R. Architecture Exploration for a Reconfigurable Architecture Template. **IEEE Design & Test of Computers**, v. 22, n. 2, p. 90-101, abr. 2005.

BECK, A. C. S.; RUTZIG, M. B.; GAYDADJIEV, G.; CARRO, L. Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications. 2008 Design, Automation and Test in Europe (DATE), Munich, 2008. **Anais**… p. 1208-1213, 2008.

big.LITTLE Technology. Disponível em: <https://www.arm.com/products/processors/technologies/biglittleprocessing.php>. Acesso em: 19 fev. 2018.

BINKERT, N.; BECKMANN, B.; BLACK, G.; REINHARDT, S. K.; SAIDI, A.; BASU, A.; HESTMESS, J.; HOWER, D. R.; KRISHNA, T.; SARDASHTI, S.; SEN, R.; SEWELL, K.; SHOAIB, M.; VAISH, N.; HILL, M. D.; WOOD, D. A. The gem5 simulator. **ACM SIGARCH Computer Architecture News**, v. 39, n. 2, p. 1-7, ago. 2011.

BORKAR, S. Y. et al. Platform 2015: Intel Processor and Platform Evolution for the Next Decade. [S.l.], 2005.

BORKAR, S.; CHIEN, A. A. The Future of Microprocessors. **Communications of the ACM**, New York, v. 54, n. 5, p. 67-77, mai. 2011.

BOUTHAINA, D.; BAKLOUTI, M.; NIAR, S.; ABID, M. Shared Hardware Accelerator Architectures for Heterogeneous MPSoCs. 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), Darmstadt, 2013. **Anais**… p. 1-6, 2013.

BLAKE, G.; DRESLINSKI, R. G.; MUDGE, T.; FLAUTNER, K. Evolution of Thread-Level Parallelism in Desktop Applications. **ACM SIGARCH Computer Architecture News – ISCA '**10, Saint Malo, v. 38, n. 3, p. 302-313, jun. 2010.

BLAKE, G.; DRESLINSKI, R. G.; MUDGE, T. A Survey of Multi-core Processors. **IEEE Signal Processing Magazine**, v. 26, n. 6, p. 26-37, out. 2009.

BRANDALERO, M.; BECK, A. C. S. A mechanism for energy-efficient reuse of decoding and scheduling of x86 instruction streams. 2017 Conference on Design, Automation & Test in Europe, Lausanne. **Anais**… p. 1472-1477, 2017.

BROOKS, D. M.; BOSE, P.; SCHUSTER, S.; JACOBSON, H. M.; KUDVA, P.; BUYUKTOSUNOGLU, A.; WELLMAN, J. D.; ZYUBAN, V. V.; GUPTA, M.; COOK, P. W. Design and modeling challenges for next-generation microprocessors. **IEEE Micro**, v. 20, n. 6, p. 26-44, nov. 2000.

BUTENHOF, D. R. **Programming with POSIX Threads**. 1. ed., Nashua, NH: Addison-Wesley, 1997.

CHE, S.; BOYER, M.; MENG, J.; TARJAN, D.; SHEAFFER, J. W.; LEE, S.; SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. IEEE International Symposium on Workload Characterization (IISWC), Austin, 2009. **Anais**… p. 44-54, 2009.

CHEN, Z.; PITTMAN, R. N.; FORIN, A. Combining Multi-core and Reconfigurable Instruction Set Extensions. 18th Annual ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA), Monterey, 2010. **Anais**… p. 33-36, 2010.

CHEN, L.; MITRA, T. Shared Reconfigurable Fabric for Multi-Core Customization. 48th Design Automation Conference (DAC), New York, 2011. **Anais**… p. 830-835, 2011.

CLARK, N.; KUDLUR, M.; PARK, H.; MAHLKE, S.; FLAUTNER, K. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. 37th International Symposium on Microarchitecture (MICRO), Portland, 2004. **Anais**... p. 30-40, 2004.

COTA, E. G.; MANTOVANI, P.; GUGLIELMO, G. D.; CARLONI, L. P. An Analysis of Accelerator Coupling in Heterogeneous Architectures. 52nd Annual Design Automation Conference (DAC), San Francisco, 2015. **Anais**… p. 202-1 – 202-6, 2015.

DEHNERT, J.C.; GRANT, B. K.; BANNING, J. P.; JOHNSON, R.; KISTLER, T.; KLAIBERT, A. MATTSON, J. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO), San Francisco, 2003. **Anais**… p. 15-24, 2003.

DENNARD, R. et al. Design of ion-implanted MOSFETs with very small physics dimensions. **IEEE Journal of Solid State Circuits**, v. 9, n. 5, p. 256–268, out. 1974.

DORTA, A. J.; RODRIGUEZ, C.; SANDE, F.; GONZALES, A. G. The OpenMP Source Code Repository. 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing, Lugano, 2005. **Anais**… p. 244-250, 2005.

EYERMAN, S.; EECKHOUT, L. Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance. **IEEE Computer Architecture Letters**, v. 13, n. 2, p. 93-96, mai. 2014.

FATEHI, E.; GRATZ, P. V. ILP and TLP in Shared Memory Applications: A Limit Study. 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), Edmonton, 2014. **Anais**… p. 24-27, 2014.

FLAUTNER, K.; UHLIG, R.; REINHARDT, S.; MUDGE, T. Thread-level parallelism and interactive performance of desktop applications. 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Cambridge, 2000. **Anais**… p. 129-138, 2000.

GARCIA, P.; COMPTON, K. Kernel sharing on reconfigurable multiprocessor systems. 2008 International Conference on Field-Programmable Technology (FPT), Taipei, 2008. **Anais**… p. 225-232, 2008.

GONZALEZ, R.; HOROWITZ, M. Energy dissipation in general purpose microprocessors. **IEEE Journal of Solid-State Circuits**, v. 31, n. 9, p. 1277-1284, set. 1996.

GREENHALGH, P. **big.LITTLE Technology: The Future of Mobile**. White Paper, ARM Ltd. Publicação eletrônica, 2013.

HAMMARLUND, P.; MARTINEZ, A. J.; BAJWA, A. A.; HILL, D. L.; HALLNOR, E.; JIANG, H.; DIXON, M.; DERR, M.; HUNSAKER, M.; KUMAR, R.; OSBORNE, R. B.; RAJWAR, R.; SINGHAL, R.; D'SA, R.; CHAPPEL, R.; KAUSHIK, S.; CHENNUPATY, S.; JOURDAN, S.; GUNTHER, S.; PIAZZA, T.; BURTON, T. Haswell: The Fourth-Generation Intel Core Processor. **IEEE Micro**, v. 34, n. 2, p. 6-20, mar. 2014.

HARTENSTEIN, R. A decade of reconfigurable computing: a visionary retrospective. 2001 Conference on Design, Automation and Test in Europe (DATE), Munich, 2001. **Anais**… p. 642-649, 2001.

HENESSY, J. L.; DAVID A. PATTERSON. Computer Architecture: A Quantitative Approach. 5. ed. Morgan Kaufmann, 2011.
HILL, M. D.; MARTY, M. R. Amdahl's Law in the Multi-core Era. **Computer**, v. 41, n. 7, p. 33-38, jul. 2008.

HOLT, W. M. Moore's Law: A path going forward. 2016 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, 2016. **Anais**… p. 8-13, 2016.

HOU et al. Efficient Data Streaming with On-chip Accelerators: Opportunities and Challenges. 17th International Symposium on High Performance Computer Architecture (HPCA), San Antonio, 2011. **Anais**… p.312-320, 2011.

HUBNER, M.; BECKER, J. Multiprocessor System-on-Chip: Hardware Design and Tool Integration. 1. ed. Springer, 2011.

Intel Core i7 Processor Family for LGA2011-v3 Socket Datasheet. Disponível em: < https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/core-i7-lga2011-3-datasheet-vol-1.pdf >. Acesso em: 02/09/2018.

ITRS Roadmap 2.0. Disponível em: <https://www.semiconductors.org/clientuploads/Research_Technology/ITRS/2015/0_2015%20ITRS%202.0%20Executive%20Report%20(1).pdf>. Acesso em: 01/07/2018.

JOUPPI, N. P. In-Datacenter Performance Analysis of a Tensor Processing Unit. 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, 2017. **Anais**… p. 1-12, 2017.

KUMAR, R. et al. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. Annual IEEE/ACM International Symposium on Microarchitectures (MICRO), San Diego, 2003. **Anais**… p. 81-92, 2003.

KUMAR, R.; TULLSEN, D. M.; JOUPPI, N. P.; RANGANATHAN, P. Heterogeneous Chip Multiprocessors. **Computer**, v. 38, n. 11, p. 32-38, nov. 2005.

LEE, J.; WU, H.; RAVICHANDRAN, M.; CLARK, N. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. 37th Annual International Symposium on Computer Architecture (ISCA), Saint-Malo, 2010. **Anais**… p. 270-279, 2010.

LUO, K.; GUMMARAJU, J. FRANKLIN, M. Balancing thoughput and fairness in SMT processors. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Tucson, Arizona, 2001. **Anais**… p. 164-171, 2001.

MICHAUD, P. Demystifying multi-core throughput metrics. **IEEE Computer Architecture Letters**, v. 12, n. 2, p. 63-66, ago. 2012.

LYSECKY, R.; STITT, G.; VAHID, F. Warp Processors. 41st annual Design Automation Conference (DAC), San Diego, USA, 2004. **Anais**… New York: ACM, 2004, p. 659-681.
MOORE, G. E. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. **IEEE Solid-State Circuits Society Newsletter**, v. 11, n. 3, p. 33-35, set. 2006.

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

PERICAS, M.; CRISTAL, A.; CAZORLA, F. J.; GONZALEZ, R.; JIMENEZ, D. A.; VALERO, M. A Flexible Heterogeneous Multi-Core Architecture. 16th International Conference on Parallel Architecture and Compilation Techniques (PACT), Brasov, 2007. **Anais**… p. 13-24, 2007.

RUTZIG, M. B.; BECK, A. C. S.; CARRO, L. A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Simultaneous ILP and TLP Exploitation. 2013 Conference on Design, Automation and Test in Europe (DATE), Grenoble, 2013. **Anais**… p. 1559-1564, 2013.

SHAFIQUE, M.; GARG, S. Computing in the Dark Silicon Era: Current Trends and Research Challenges**. IEEE Design & Test**, v. 34, n. 2, p. 8–23, abr. 2017.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. Operating System Concepts. 8. ed. Wiley, 2008.

SNAVELY, A. TULLSEN, D. M. Symbiotic jobscheduling for a simultaneous multithreaded processor. 9th International conference on Architectural support for programming languages and operating systems (ASPLOS), Cambridge, 2000. **Anais…** p. 234-244, 2000.

SOUZA, J. D. **A Reconfigurable Heterogeneous Multi-core System with Homogeneous ISA**. 2016. 127 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2016.

SOUZA, J. D.; CARRO, L. RUTZIG, M. B.; BECK, A. C. S. A Reconfigurable Heterogeneous Multi-core with a Homogeneous ISA. 2016 Conference on Design, Automation & Test in Europe (DATE), Dresden, Germany, 2016. **Anais**… p. 1598-1603, 2016.

SRINIVASAN, S. et al. A Study on Polymorphing Superscalar Processor Dynamically to Improve Power Efficiency. IEEE Computer Society Annual Symposium on VLSI, Natal, 2013. **Anais**... p. 46–51, 2013.

TODMAN, T. J.; CONSTANTINIDES, G. A.; WILTON, S. J. E.; MENCER, O.; LUK, W.; CHEUNG, P. Y. K. Reconfigurable computing: architectures and design methods. **IEE Anais – Computer and Digital Techniques**, v. 152, n. 2, p. 193-207, jul. 2005.

TOMUSK, E.; DUBACH, C.; O'BOYLE, M. Measuring Flexibility in Single-ISA Heterogeneous Processors. 23rd International conference on Parallel architectures and compilation (PACT), Edmonton, 2014. **Anais**… p. 495-496, 2014.

TOMUSK, E.; DUBACH, C.; O'BOYLE, M. Four Metrics to Evaluate Heterogeneous Multi-cores. **ACM Transactions on Architecture and Code Optimization** (TACO), v. 12, n. 4, p. 1-25, 2016.

TOMUSK, E. **Heterogeneous Processor Composition: Metrics and Methods**. 2016. 266 f. Tese (Doutorado em Ciência da Computação) – Escola de Informática, Universidade de Edinburgo, Edinburgo, 2016.

TULLSEN, D. M.; EGGERS, S. J.; LEVY, H. M. Simultaneous Multithreading: Maximizing on-chip parallelism. 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, 1995. **Anais**… p. 392-403, 1995.

VAHID, F.; STITT, G.; LYSECKY, R. Warp Processing: Dynamic Translation of Binaries to FPGA Circuits. **Computer**, v. 41, n. 7, p. 40-46, jul. 2008.

WATKINS, M. A.; ALBONESI, D. H. ReMAP: A Reconfigurable Heterogeneous Multi-core Architecture. 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Atlanta, 2010. **Anais**… p. 497-508, 2010.

WOLF, W. The Future of Multiprocessor Systems-on-Chips. 41st Annual Design Automation Conference (DAC), San Diego, 2004. **Anais**… p. 681-685, 2004.

WOLF, W.; JERRAYA, A. A.; MARTIN, G. Multiprocessor system-on-chip (MPSoC) technology. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 27, n. 10, p. 1701–1713, 2008.

XILINX. **Zynq Ultrascale+ MPSoC Product Brief**. Disponível em: <https://www.xilinx.com/support/documentation/product-briefs/zynq-ultrascale-plus-product-brief.pdf>. Acesso em: 24/08/2018.