Universidade Estadual de Campinas
Instituto de Computação

Casio Pacheco Krebs

Integração e análise de desempenho de arquiteturas RISC-V paralelas

CAMPINAS
2023

Casio Pacheco Krebs

Integração e análise de desempenho de arquiteturas RISC-V paralelas

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Lucas Francisco Wanner
Coorientador: Prof. Dr. Guido Costa Souza de Araújo

Este exemplar corresponde à versão final da Dissertação defendida por Casio Pacheco Krebs e orientada pelo Prof. Dr. Lucas Francisco Wanner.

CAMPINAS
2023

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

K871i

Krebs, Casio Pacheco, 1997-
Integração e análise de desempenho de arquiteturas RISC-V paralelas / Casio Pacheco Krebs. – Campinas, SP : [s.n.], 2023.

Orientador: Lucas Francisco Wanner.
Coorientador: Guido Costa Souza de Araújo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Processamento paralelo (Computadores). 3. Computação de alto desempenho. I. Wanner, Lucas Francisco, 1981-. II. Araújo, Guido Costa Souza de, 1962-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações Complementares

Título em outro idioma: Integration and performance analysis of parallel RISC-V architectures
Palavras-chave em inglês:
Computer architecture
Parallel processing (Electronic computers)
High performance computing
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Lucas Francisco Wanner
Rodolfo Jardim de Azevedo
Mateus Beck Rutzig
Data de defesa: 03-04-2023
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0003-1719-8546
- Currículo Lattes do autor: https://lattes.cnpq.br/4374078124009506

Universidade Estadual de Campinas
Instituto de Computação

Casio Pacheco Krebs

Integração e análise de desempenho de arquiteturas RISC-V paralelas

Banca Examinadora:

- Prof. Dr. Lucas Francisco Wanner
  Universidade Estadual de Campinas
- Prof. Dr. Rodolfo Jardim de Azevedo
  Universidade Estadual de Campinas
- Prof. Dr. Mateus Beck Rutzig
  Universidade Federal de Santa Maria

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 03 de abril de 2023

Resumo

O uso de arquiteturas vetoriais e matriciais tem o potencial de aceleração proporcional à quantidade de unidades de processamento primitiva e permitem reduzir a sobrecarga na cache de instruções. Neste cenário, acelerar rotinas de multiplicação e acumulação (MAC) sobre estruturas regulares, por meio do processamento paralelo de dados em estruturas de *hardware* dedicadas, tornou-se um objetivo desejado, tanto pela indústria, quanto pelo ambiente acadêmico. No entanto, a ativação dessas estruturas não são triviais, necessitando de intervenções por parte do programador no código base da aplicação, caso contrário, o código não estará atingindo o máximo de eficiência promovido pelo sistema. Neste trabalho, são investigados o coprocessador vetorial Hwacha e o acelerador matricial Gemmini, concomitantemente com o processador RISC-V superescalar BOOM. A fim de eliminar a dependência de intervenções no código base e do conhecimento das rotinas de ativação, a ferramenta de reescrita de código automática SMR foi estendida, a partir do desenvolvimento de novas bibliotecas, que resumem as rotinas de preparação e movimentação de dados, combinados com as instruções de ativação do Hwacha e do Gemmini, nos padrões de execução GEMV e GEMM. Com o uso da ferramenta SMR acordado com a plataforma Verilator, foi criado um ecossistema de simulação, onde o desempenho separado do Hwacha e do Gemmini foram analisados, e na sequência, comparados com o processador RISC-V BOOM, a partir de sete aplicações do conjunto de Álgebra Linear do *benchmark* Polybench. Com esse ecossistema de simulação, acreditamos disponibilizar uma ferramenta capaz de ativar essas estruturas de aceleração sem a necessidade de intervenções no código base da aplicação.

Abstract

The use of vector and matrix architectures has the potential for acceleration proportional to the number of primitive processing units and allows reducing the overhead in the instruction cache. In this scenario, accelerating multiplication and accumulation (MAC) routines on regular structures, through parallel processing of data in dedicated hardware structures, has become a desired objective, both by industry and by the academic environment. However, the activation of these structures is not trivial, requiring interventions by the programmer in the base code of the application, otherwise the code will not be reaching the maximum efficiency promoted by the system. In this work, the Hwacha vector coprocessor and the Gemmini matrix accelerator are investigated, concomitantly with the BOOM superscalar RISC-V processor. In order to eliminate the dependency on interventions in the base code and knowledge of activation routines, the SMR automatic code rewriting tool was extended, starting with the development of new libraries, which summarize the preparation and data movement routines, combined with the Hwacha and Gemmini activation instructions in the GEMV and GEMM runtime patterns. With the use of the SMR tool agreed with the Verilator platform, a simulation ecosystem was created, where the separate performance of the Hwacha and the Gemmini were analyzed, and then compared with the RISC-V BOOM processor, from seven applications of the Linear Algebra set of the Polybench benchmark. With this simulation ecosystem, we believe we can provide a tool capable of activating these acceleration structures without the need for interventions in the application's base code.

Lista de Figuras

Sumário

Capítulo 1

Introdução

Operações de multiplicação e acumulação (MAC) envolvendo estruturas regulares, como matrizes e vetores, compõem uma das mais fundamentais rotinas existentes em problemas computacionais. Aplicações envolvendo aprendizado de máquina, como redes neurais e redes convolucionais, podem ter operações com matrizes, comprometendo cerca de 70% do total de ciclos de computação durante a etapa de treinamento [Qin et al., 2020].

Como resultado da crescente demanda por essas rotinas, despertou um grande interesse em desenvolver bibliotecas [Oliphant, 2006; Xianyi and Kroeker, 2020] e algoritmos [Strassen et al., 1969; Coppersmith and Winograd, 1987; Williams, 2012, 2014] que buscam acelerar operações de multiplicação de Matriz-Matriz e Matriz-Vetor, também denominado de GEMM e GEMV, respectivamente. No entanto, por serem soluções baseadas em *software*, tendem a serem limitadas pelo núcleo, que precisa cumprir uma demanda de área e energia.

Historicamente, a melhoria da capacidade de processamento foi motivada principalmente pela Lei de Moore [Schaller, 1997] e pela escala de Dennard [Dennard et al., 1974]. Contudo, Dennard [Dennard et al., 2007] e outros pesquisadores já identificaram que os circuitos com transistores atingiram um ponto de fabricação crítico, onde o aumento exponencial da corrente de fuga impede o seguimento da escala original, gerando um aumento na densidade de potência e no custo energético total do *chip*. Nesse cenário, um alvo de pesquisa para continuar aumentando o desempenho, é o desenvolvimento de arquiteturas que buscam atender a um propósito específico da aplicação.

Uma prática comum dos projetistas é utilizar essas arquiteturas dedicadas como aceleradores dentro do *chip*, operando em conjunto com o processador de propósito geral, visando melhorar o desempenho final do *chip*, acelerando as rotinas extremamente intensivas em computação, e consequentemente, liberando o processador para continuar realizando computação útil. A implementação em hardware dessas rotinas, também permite avanços na eficiência energética dos projetos. Devido, não só pela simplificação das estruturas de controle e computação, mas também pela possibilidade de paralelização na etapa de processamento dos dados.

As aplicações que se baseiam em processamento de operações GEMM e GEMV, são alvo de otimizações por diversos projetos de aceleradores [Lee et al., 2015; Chen et al., 2016; Jouppi et al., 2017; Moss et al., 2018; Nvidia, 2017, 2018, 2020; Genc et al., 2021]. As arquiteturas desses aceleradores são baseadas no modelo de processamento *Single Ins-*

truction, Multiple Data (SIMD), onde a unidade de processamento primitiva é replicada, mas conectada a uma mesma unidade de controle, permitindo assim, que os aceleradores executem a mesma instrução sobre diferentes dados de forma simultânea.

Nesse intuito, apesar de terem o mesmo objeto, diversas topologias de aceleradores foram desenvolvidas, onde destacam-se as arquiteturas vetoriais [Lee et al., 2015; Lomont, 2011; Intel, 2022] e as sistólicas [Chen et al., 2016; Qin et al., 2020; Liu et al., 2020; Genc et al., 2021]. No entanto, a integração entre o *software* e o *hardware* é uma tarefa complexa no desenvolvimento dessas arquiteturas dedicadas.

Com um conjunto de instruções (ISA) que estende as instruções nativas do processador, é necessário por parte do programador, conhecimento prévio das funcionalidades do sistema para realizar adaptação do código base da aplicação para à sua ativação. Dessa forma, o desenvolvimento de bibliotecas de alto nível para facilitar seu uso, faz-se recomendado, uma vez que diferentes projetos podem estar presente em um mesmo SoC, e quando não acionadas, uma fração do *hardware* é subutilizada.

Nesse cenário, esta proposta buscou implementar e avaliar o coprocessador vetorial Hwacha [Lee et al., 2015] e o acelerador de processamento matricial Gemmini [Genc et al., 2021], visando disponibilizar diretrizes para os seus desenvolvimentos. Uma vantagem desses projetos, é a sua integração a framework de descrição de *hardware* Chipyard [Amid et al., 2020], que fornece ferramentas para implementar e avaliar componentes SoC (em português, sistema-em-um-chip). Por ter a natureza de código aberto, apresenta suporte a uma ampla variedade de processadores, incluindo o processador RISC-V superescalar BOOM [Zhao et al., 2020].

Visando oferecer suporte de alto nível, a interface de programação das arquiteturas, foi estendido no fluxo de compilação do *Source Matching and Rewriting* (SMR) [Espindola et al., 2023], uma ferramenta que realiza a correspondência e reescrita de código. Para esse fim, foi desenvolvido bibliotecas otimizadas que realizam as rotinas dessas arquiteturas, permitindo que o Hwacha e o Gemmini possam ser utilizados sem a necessidade de conhecimento do compilador, das funções de ativação, e principalmente, sem precisar modificar o código base da aplicação. Por fim, o desempenho das bibliotecas desenvolvidas foi comparado com a solução em *software* OpenBLAS [Xianyi and Kroeker, 2020], sendo executada no processador superescalar BOOM.

Sumarizando, este trabalho apresenta como principais contribuições:

- Preparação de um ambiente experimental para integração do acelerador Gemmini e do coprocessador Hwacha com um processador RISC-V superescalar, via interface RoCC (*Rocket Custom Coprocessor Interface*).
- Desenvolvimento de bibliotecas para integração e abstração das arquiteturas paralelas em aplicações de GEMV e GEMM.
- Integração das bibliotecas no fluxo do SMR, uma ferramenta para reescrita automática de código.
- Análise comparativa de desempenho entre o processador superescalar BOOM e arquiteturas vetoriais e matriciais.

Este texto está organizado conforme segue: o Capítulo 2, apresenta operações clássicas de multiplicação envolvendo vetores e matrizes, assim como aplicações com uso intensivo dessas operações; o Capítulo 3, apresenta os detalhes das arquiteturas dos componentes utilizados no estudo, como as arquiteturas de processamento paralelo e o processador de propósito geral; o Capítulo 4, apresenta trabalhos que englobam o contexto de aceleração por arquiteturas de processamento paralelo dedicada; no Capítulo 5, é apresentado o fluxo de processamento do SMR, combinado com a biblioteca de ativação das arquiteturas em seu fluxo; os resultados são explorados no Capítulo 6 e, a conclusão, é apresentada no Capítulo 7.

Capítulo 2

Referencial Teórico

2.1 Multiplicação Matriz-Matriz

Descrito pela primeira vez pelo matemático James Joseph Sylvester, em 1812, e divulgado em 1858, pelo matemático Arthur Cayley [Cayley, 1858], o termo matriz é utilizado para representar uma estrutura de dados dispostas em linhas e colunas. Comumente, os dados de uma matriz são representados entre colchetes, como mostrado abaixo:

$$\begin{bmatrix} C_{0,0} & C_{0,1} & \dots & C_{0,N-1} \\ C_{1,0} & C_{1,1} & \dots & C_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ C_{M-1,0} & C_{M-1,1} & \dots & C_{M-1,N-1} \end{bmatrix} =$$

$$\begin{bmatrix} A_{0,0} & A_{0,1} & \dots & A_{0,K-1} \\ A_{1,0} & A_{1,1} & \dots & A_{1,K-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{M-1,0} & A_{M-1,1} & \dots & A_{M-1,K-1} \end{bmatrix} \cdot \begin{bmatrix} B_{0,0} & B_{0,1} & \dots & B_{0N-1} \\ B_{1,0} & B_{1,1} & \dots & B_{1N-1} \\ \vdots & \vdots & \ddots & \vdots \\ B_{K-1,0} & B_{K-1,1} & \dots & B_{K-1,N-1} \end{bmatrix}$$

A multiplicação entre duas matrizes é definida se, e somente se, o número de colunas da matriz **A** for igual ao número de linhas da matriz da **B**. Dessa forma, dada uma matriz **A** do tipo [MxK], onde M é o número de colunas e K o número de linhas, e uma matriz **B** do tipo [KxN], o produto da matriz **A** pela matriz **B** é a matriz **C** do tipo [MxN], onde o elemento (**i,j**) da matriz resultante é calculado pelo somatório dos produtos obtidos pela multiplicação dos elementos da linha **i**, da matriz **A**, pelos elementos da coluna **j**, da matriz **B**, como mostrado pela equação 2.1.

$$C_{i,j} = A_{i,0}B_{0,j} + A_{i,1}B_{0,j} + ... + A_{i,K-1}B_{K-1,j} = \sum_{l=0}^{k-1} A_{i,l}B_{l,j} \tag{2.1}$$

Uma abordagem ingênua, para implementar a multiplicação entre matrizes, consiste em manter dois loops aninhados, a fim de acessar cada posição da matriz resultante, e um loop interno para se movimentar pela linha **i**, da matriz **A** e pela coluna **j**, da matriz **B**, realizando a multiplicação entre os elementos e a acumulação na posição de destino.

Essa implementação ingênua apresenta complexidade cúbica ($\mathcal{O}(n^3)$), e pode ser vista na Figura 2.1.

```
1 for (int i = 0; i < M; i++)
2     for (int j = 0; j < N; j++){
3         C[i][j] = 0;
4         for (int l = 0; l < K; l++)
5             C[i][j] += A[i][l] * B[l][j];
6     }
```

Figura 2.1: Código de multiplicação Matriz-Matriz

A generalização desse tipo de multiplicação recebe o termo GEMM (abreviação do ingles *General Matrix Multiplication*), e ocorre quando são adicionadas duas etapas de multiplicação por escalares. A primeira multiplica o resultado do produto da matriz **A** com a matriz **B**, com uma variável *alpha*, e a segunda, multiplica o valor previamente armazenado na matriz **C** com uma variável *beta*, com função de um acumulador. Apesar das multiplicações extras, a operação de GEMM apresenta complexidade cúbica. A função geral por ser vista na equação 2.2.

$$C := alpha * op(A) * op(B) + beta * C \tag{2.2}$$

Onde op(**A**) e op(**B**) representam as orientações de como as matrizes devem ser acessadas na multiplicação, isto é, sua orientação normal ou transposta. Desse modo, a operação pode ser executada de quatro formas, que podem ser vistas abaixo:

$$C = alpha * A * B + beta * C \tag{2.3}$$
$$C = alpha * A^t * B + beta * C \tag{2.4}$$
$$C = alpha * A * B^t + beta * C \tag{2.5}$$
$$C = alpha * A^t * B^t + beta * C \tag{2.6}$$

Algumas termologias são empregadas para identificar o tipo das variáveis utilizadas nas operações, para isso, é adicionado uma letra no início da operação. Os termos empregados podem ser visualizados abaixo:

- S: número real de precisão simples;
- D: número real de precisão dupla;
- C: número complexo de precisão simples;
- Z: número complexo de precisão dupla.

2.2 Multiplicação Matriz-Vetor

A partir da definição apresentada na seção anterior, um caso específico ocorre quando uma das matrizes apresenta apenas uma coluna, onde essa matriz recebe o nome de vetor. A multiplicação entre uma matriz do tipo [MxN] e um vetor de dimensão N, acarreta criação de um novo vetor de dimensão M, como mostrado abaixo:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \cdot\cdot \\ y_{M-1} \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & \cdots & A_{0,N-1} \\ A_{1,0} & A_{1,1} & A_{1,2} & \cdots & A_{1,N-1} \\ A_{2,0} & A_{2,1} & A_{2,2} & \cdots & A_{2,N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{M-1,0} & A_{M-1,1} & A_{M-1,2} & \cdots & A_{M-1,N-1} \end{bmatrix} . \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdot\cdot \\ x_{N-1} \end{bmatrix}$$

O produto Matriz-Vetor é definido se, e somente se, o número de colunas da matriz **A** é igual ao número de linhas do vetor **x**. Analogamente à multiplicação entre matrizes, o produto entre uma matriz e um vetor é calculado a partir do somatório dos produtos obtidos pela multiplicação dos elementos da linha **i**, pelo elemento **i** do vetor, conforme mostrado pela equação 2.7.

$$y_i = \sum_{j=0}^{N} A_{i,j} x_j \tag{2.7}$$

Nesse caso, a abordagem ingênua consiste em implementar um loop que percorre as posições do vetor resultante, e um loop interno para se movimentar pela linha **i**, da matriz **A** e pelas posições do vetor **x**, onde o resultado de cada multiplicação é acumulada na posição **i**, no vetor de destino **y**. Essa implementação ingênua pode ser vista na Figura 2.2.

```
1 for (int i = 0; i < M; i++){
2     y[i] = 0;
3     for (int j = 0; j < N; j++)
4         y[i] += A[i][j] * x[j];
5 }
```

Figura 2.2: Código de multiplicação Matriz-Vetor

A generalização desse tipo de multiplicação recebe o termo GEMV (abreviação do inglês *General Matrix-Vector Multiply*), e de forma análoga à operação GEMM, ocorre quando são adicionadas às etapas de multiplicação por *alpha* e *beta*. A função geral pode ser vista na equação 2.8.

$$y = alpha * op(A) * x + beta * y \tag{2.8}$$

2.3 BLAS

As abordagens descritas anteriomente realizam a leitura sequencial dos elementos, em seguida multiplicam e, por fim, acumulam o resultado no endereço de destino até completarem todos os produtos parciais. Em arquiteturas que apresentam hierarquia de memória, com vários níveis de cache e memória DRAM desacoplada, as estratégias podem apresentar perda de desempenho à medida que as matrizes envolvidas cresçam o suficiente ao ponto de não poderem ser completamente alocadas na cache, uma vez que os níveis superiores apresentam uma maior latência de acesso. Dessa forma, para multiplicação com matrizes muito grandes, o desempenho fica limitado pela largura de banda da memória DRAM.

Visando disponibilizar abordagens mais eficientes para tais operações, bibliotecas como *Basic Linear Algebra Subprograms* (BLAS) [Lawson et al., 1979], e suas derivações como o GOTOBLAS2 [Center, 2020], o OpenBLAS [Xianyi and Kroeker, 2020] e o NVBLAS [NVIDIA, 2022] foram desenvolvidas. Essas bibliotecas fornecem uma interface de programação, onde são disponibilizadas rotinas comuns de Algebra Linear, para realizar operações com vetores e matrizes. As bibliotecas visam aumentar a taxa de reutilização dos dados existentes na cache, otimizando assim os acessos à memória.

A implementação em baixo nível dessas bibliotecas, busca abstrair as rotinas em subrotinas menores, permitindo assim que os desenvolvedores se concentrem em otimizar um problema específico, visando atingir o algoritmo ótimo. Mas também, muitas bibliotecas buscaram otimizar as rotinas utilizando recursos de uma arquitetura específica, como por exemplo, ativando registradores vetoriais e programando os dados para aproveitarem instruções vetoriais, dessa forma, várias bibliotecas baseadas na interface BLAS são desenvolvidas com o foco em otimizar um conjunto de arquiteturas específicas.

Apesar do suporte restrito, a padronização das chamadas das funções permite que os usuários desenvolvam programas indiferentes à biblioteca BLAS que está sendo consumida, dessa forma, a etapa de migração para uma nova arquitetura necessita apenas da troca da biblioteca e não da modificação do código base.

A interface de programação fornecida pelas bibliotecas derivadas do BLAS, tem suas rotinas divididas em três conjuntos, denominados níveis, representando o grau de complexidades dos algoritmos. O nível 1, define o conjunto de funções de álgebra linear referentes à operações vetoriais que normalmente levam tempo linear ($\mathcal{O}(n)$), como, por exemplo, produtos escalares e adição vetorial, de formato: $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$. O principal benefício apresentado pelas bibliotecas atuais, nesse nível, se deve à ativação das instruções de processamento vetorial.

As funções de nível 2 são destinadas às operações de Matriz-Vetor, onde se encontra a operação de GEMV, e de forma geral, apresentam tempo quadrático ($\mathcal{O}(n^2)$. A implementação das rotinas desse nível podem ser apresentadas em termos de funções de nível 1. As funções de nível 3, são operações como o produto Matriz-Matriz, onde se encontra a operação de GEMM, e normalmente apresentam tempo cúbico ($\mathcal{O}(n^3)$). Tais funções também podem ser implementadas a partir de chamadas das rotinas dos níveis inferiores.

Essas bibliotecas buscam acelerar a multiplicação de matrizes otimizando os acessos na cache, para isso, os dados das matrizes são divididos em blocos que possam ser alocados

nos diferentes níveis de cache, permitindo assim, que os blocos sejam processados com alta taxa de acerto das caches devido o reaproveitamento de dados. Por exemplo, uma multiplicação envolvendo matrizes de [1024x1024] pode ser dividida em uma sequência de multiplicações de matrizes de [64x64]. As operações de nível 2 também podem ser subdivididas, caso a matriz envolvida seja grande o suficiente.

2.4 Convolução

Dentro do campo da matemática, o termo convolução é designado para representar a operação entre duas funções reais que produzem uma terceira função. Formalmente, convolução dentre uma função **f**: $\mathbb{R} \rightarrow \mathbb{R}$ e uma função **g**: $\mathbb{R} \rightarrow \mathbb{R}$, resulta em uma função simbolizada por **(f * g)**, denominado de mapa de recursos. No campo discreto, a convolução é caracterizada pela soma dos produtos parciais resultantes entre a superposição das duas funções após uma ser refletida em torno do eixo y e deslocada, conforme mostrado pela equação 2.9.

$$(f * g)[i] = \sum_{l=-\infty}^{\infty} f[l]g[i-l] \tag{2.9}$$

A equação 2.9 engloba operações com funções unidimensionais, nesse caso, a operação também pode ser chamada de convolução 1D. Caso as funções envolvidas se estendam ao longo de um campo bidimensional, a operação é denominada de convolução 2D. A equação 2.10 mostra a convolução bidimensional considerando uma matriz **A** de dimensão [MxN] e uma matriz **G**, de dimensão [RxP].

$$(A * G)[i,j] = \sum_{k=0}^{R-1} \sum_{l=0}^{P-1} A_{i-k,j-l} G_{k,l} \tag{2.10}$$

O algorítimo ingênuo para a execução de uma convolução 2D, consiste na implementação de quatro loops, dois loops aninhados para percorrer as posições na matriz **A**, seguido por dois loops aninhados para percorrer as posições na matriz **G**, realizando o processo de multiplicação dos dados sobrepostos e acumulações na posição de destino. Dessa forma, a aplicação ingênua da convolução 2D se torna uma operação muito cara computacionalmente, apresentando complexidade de $\mathcal{O}(RxP)$ para cada posição da matriz **A**. Essa implementação pode ser vista na Figura 2.3.

2.4.1 Aplicação em algoritmos

No campo da Inteligência Artificial e de Aprendizado de Máquina, a Convolução 2D tem um papel fundamental na implementação dos algoritmos de processamento de imagens e, também, nos de inferência de Rede Neural Convolucional (CNN) [O'Shea and Nash, 2015], para fins de classificação e reconhecimento de padrões e objetos em imagens digitais.

```
1 for (int i = 0; i < M; i++)
2     for (int j = 0; j < N; j++){
3         C[i][j] = 0;
4         for (int k = 0; k < R; k++)
5             for (int l = 0; l < P; l++)
6                 C[i][j] += A[i-k][j-l] * G[k][l];
7     }
```

Figura 2.3: Código de convolução 2D

Aplicações de processamento de imagens como o filtro de Sobel [Kanopoulos et al., 1988] e semelhantes, para detecção de borda, algoritmos de Realce como filtros passa-baixa e passa-alta, e também algoritmos de rotação, reflexão ou dimensionamento em imagens, denominadas de transformações afins, são implementadas a partir de uma pequena matriz carregada com o respectivo peso do filtro, denominada de máscara ou kernel.

Por exemplo, para a realização da detecção de borda a partir do filtro de Sobel, são aplicadas duas matrizes de kernel, uma para detecção de borda na vertical (G_x) e a outra para a horizontal (G_y). O filtro usa kernels [3×3] que são convoluídos com a imagem original. Esses kernels podem ser visualizados abaixo:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Para realizar o processamento, as duas matrizes de kernels são propagadas pela imagem, também representada por uma matriz, onde cada posição representa o valor do pixel. Os dados sobrepostos são convoluidos e o resultado das duas convoluções são ponderados, a fim de gerar o novo pixel da imagem final, que é classificado como parte de uma borda ou não. Dessa forma, para cada pixel são realizados dois procedimentos de convolução 2D.

CNN compõem um subconjunto da área de Aprendizado de Máquina, que atualmente está recebendo bastante destaque. Tradicionalmente, os algoritmos de CNN são compostos por três camadas: uma camada convolucional, uma camada de *pool* e uma camada totalmente conectada (FC).

A camada convolucional é o bloco de construção central de uma CNN, o qual é o responsável pelo maior volume computacional em uma CNN. Nessa camada é aplicado o kernel que representam o padrão do objeto desejável, movendo-se pelos pixels da imagem, verificando se o padrão está presente. Uma CNN pode ter várias camadas convolucionais, cada uma destinada a detectar os diferentes padrões da imagem de entrada. A saída final após todas as interações é conhecida como mapa de recursos. Por fim, a imagem é convertida em valores numéricos, o que permite que na camada FC, seja interpretada e extraído todos os padrões identificados na imagem.

2.4.2 Conversão para chamadas blas

A convolução 2D pode ser reduzida para operações de multiplicação entre matrizes, dessa forma, os programadores podem otimizar os códigos a partir do uso de bibliotecas equivalentes ao BLAS. A ideia principal é transformar a imagem de entrada em uma representação baseada em "Coluna" de tal forma que as chamadas do GEMM para o produto escalar possam ser usadas para encontrar a saída final de convolução.

O algorítimo apresentado por Kumar Chellapilla, em 2006, denominado de *im2col* [Chellapilla et al., 2006], introduziu a ideia de expandir todas as janelas possíveis na memória e, em seguida, realizar o produto escalar como uma multiplicação de matrizes. Para esse fim, é identificada a quantidade de dados na matriz de kernel, que representa a altura da nova matriz de entrada, enquanto, a largura da nova matriz é igual ao número de vezes que o kernel de convolução desliza sobre a imagem de entrada. Os blocos de dados são tirados primeiro de cima para baixo e depois da esquerda para a direita da matriz.

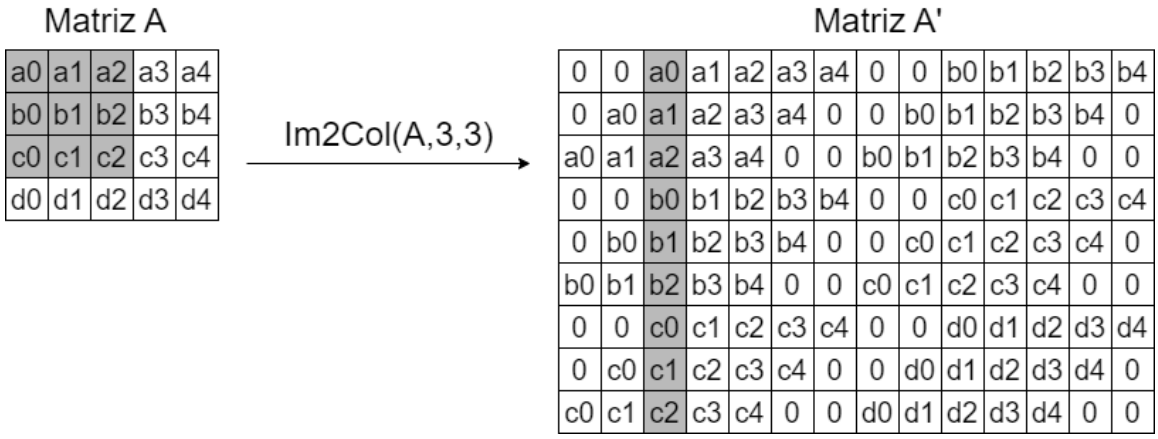

Figura 2.4: Representação da matriz de entrada antes e após o procedimento de *im2col*, com kernel de dimensão [3x3]

Na Figura 2.4, é demonstrado o procedimento de *im2col*, com kernel de dimensão [3x3]. Após o procedimento, os pesos das matrizes de kernels são diretamente transformados em linhas. Por exemplo, para kernel de tamanho [3x3], dará uma nova matriz **G** de tamanho [1x9], onde 3 x 3 = 9. Dessa forma, a nova matriz de entrada e o novo kernel podem ser processados diretamente com uma operação de GEMM.

Para convolução com apenas um kernel, o procedimento resultará em um vetor com dimensão equivalente ao número de vezes que o kernel de convolução desliza, sendo então convertido para as posições finais, com um procedimento análogo, denominado de *col2im*. As técnicas de *im2col* e *col2im* não melhoram a complexidade da convolução, mas melhora o desempenho em tempo real, pois permitem o uso de chamadas GEMM, às custas de mais uso de memória.

2.5 Aceleração em hardware

As arquiteturas vetoriais e matriciais têm sido bastante difundidas pela sua capacidade de explorar o paralelismo na execução de aplicações MAC. Como apresentado na seção 2.4, a implementação de uma convolução 2D é muito cara computacionalmente, podendo atingir complexidade quadrática. Trabalhos como o Decaf [Donahue et al., 2014] mostram que as operações de convolução em uma CNN ocupam mais de 90% do tempo de execução, dessa forma, é um dos principais alvos para a aceleração de hardware.

As instruções para ativação dessas arquiteturas têm um potencial de aceleração proporcional a quantidades de unidades de processamento primitiva, além disso, permite reduzir a carga na cache de instruções e a pressão do pipeline, por reduzir a quantidade de instruções que precisam ser carregadas, decodificadas e emitidas.

O trabalho apresentado por Hasan Genc [Genc et al., 2021], mostra que quando a carga computacional é transferida para o acelerador, o núcleo presente no projeto tem menor impacto na computação da aplicação, permitindo assim, reduzir a área do projeto a partir da redução da complexidade do núcleo, mas preservando o desempenho da aplicação.

O mesmo trabalho apresenta uma arquitetura matricial de um array sistólico de 16x16 unidades de processamento (o conceito e funcionamento de um array sistólico será explorado na seção 3.5.1) acelerando CNNs. Quando aplicado na rede neural ResNet50, a arquitetura proposta atingiu um speedup de 189 vezes e quando aplicado na rede AlexNet, o speedup apresentado foi de 216 vezes, valores atingidos, acelerando apenas as operações de GEMM.

Apesar de permitir ganho de desempenho, o uso de estruturas para processamento paralelo apresentam desvantagens quando comparado com a execução tradicional [Cebrian et al., 2020]. Com o aumento da quantidade de unidades de processamento, aplicações originalmente limitadas pela ULA podem se tornar limitantes pela memória, uma vez que o paralelismo aumenta a pressão sobre o cache de dados e ao barramento de memória, pois exigem mais largura de banda para manter os PEs carregados.

Capítulo 3

Integração entre Processador e Aceleradores na Framework Chipyard

O ecossistema Chipyard é uma framework para descrição de hardware, de código aberto, que permite projetar, avaliar e por fim, implementar componentes SoC (em português, sistema-em-um-chip), baseados no conjunto de instruções (em inglês *Instruction set Architecture* (ISA)) RISC-V [Amid et al., 2020]. Desenvolvido pelo grupo Berkeley Architecture Research (BAR) da Universidade da Califórnia (UCB) a partir do projeto *Rocket Chip Gererator* [Asanović et al., 2016], é escrito na linguagem de descrição de hardware *Chisel*, uma extensão da linguagem de programação Scala, que fornece primitivas para descrever geradores de circuitos complexos e parametrizáveis, utilizados para sintetizar códigos Verilog.

O Chipyard oferece suporte aos geradores do processador superescalar BOOM [Zhao et al., 2020], do coprocessador Hwacha [Lee et al., 2015] e do acelerador Gemmini [Genc et al., 2021], que serão utilizados neste trabalho, além de estruturas periféricas, como interconexões e cache. Esses aceleradores são linkados com o núcleo BOOM a partir da interface de conexão RoCC, integrada ao ecossistema Chipyard.

3.1 RISC-V

RISC-V é um conjunto de instruções (ISA) projetado para atender arquiteturas RISC (Computador de conjunto de instruções reduzidas, em português). Inicialmente, desenvolvido pelo grupo *Berkeley Architecture Research* [Waterman and Asanovic, 2019], foi planejado para ser utilizado como ferramenta de ensino, mas devido seu caráter aberto, chamou a atenção tanto da indústria quanto rede acadêmica, possibilitando o projeto de novas arquiteturas sem a necessidade de pagamento de royalties, permitindo a criação de diversos projetos [Conti et al., 2017; Flamand et al., 2018; Zhang et al., 2018; Pullini et al., 2019; Mashimo et al., 2019]. Atualmente é administrado pela RISC-V *International*, responsável por gerenciar a documentação de distribuição e definir alterações na ISA.

A base para o conjunto de instruções RISC-V é composto por quatro formatos de instruções, nominados de R-*type*, I-*type*, S-*type* e B-*type*, e duas variações com operador imediato, nominadas de U-*type* e J-*type*, que podem ser vistos na Figura 3.1. Esses forma-

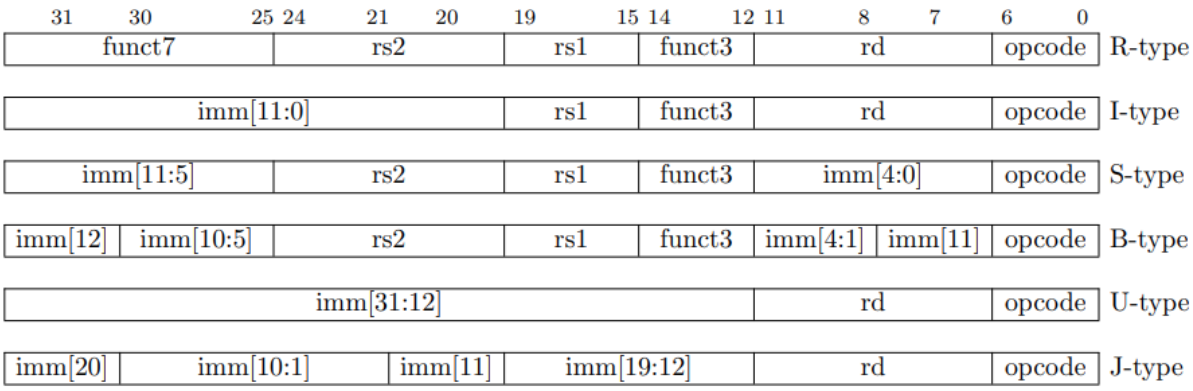

Figura 3.1: Formatos básicos de instrução RISC-V. Fonte [Waterman and Asanovic, 2019]

tos foram definidos de forma que tanto os campos de registradores de origem (*rs1* e *rs2*) quanto para o destino (*rd*) sejam mantidos na mesma posição para todos os formatos, a fim de simplificar o hardware de decodificação. Pela mesma forma, os imediatos foram colocados em direção aos bits mais significativos.

O RISC-V ISA apresenta duas variantes primárias referentes ao comprimento do espaço de endereçamento, representado pela letra "I", onde RV32I e RV64I representa o endereçamento de 32-bits ou 64-bits, respectivamente. Comumente o termo XLEN é utilizado para indicar o comprimento dos bits.

O formato rígido das instruções permitem que a ISA seja estendida com instruções específicas, a fim de reduzir o consumo de energia, tamanho do código ou o uso de memória [Waterman and Asanovic, 2019]. Algumas das extensões suportadas mais comuns podem ser vistas abaixo:

- M: Extensão padrão para multiplicação e divisão de inteiros;
- A: Extensão padrão para instruções atômicas;
- F: Extensão padrão para ponto flutuante de precisão simples;
- D: Extensão padrão para ponto flutuante de precisão dupla;
- C: Extensão padrão para instruções compactadas;

Por convenção, a letra "G"é utilizada para simplificar a combinação das extensões "IMAFD", dessa forma o termo RV64G, representa uma arquitetura de 64-bits com suporte as 4 extensões simultâneas.

3.2 Processador Berkeley Out-of-Order Machine (BOOM)

A plataforma Berkeley Out-of-Order Machine (BOOM) oferece um gerador de processador com execução fora de ordem [Zhao et al., 2020]. Atualmente em sua terceira versão, o núcleo BOOM conta com um estágio de previsão de desvio condicional, escalonamento

dinâmico de instruções, suporte a instruções RoCC e opera sobre o conjunto de instruções RV64GC. O projeto conta com uma execução dividida em 10 etapas, no entanto, algumas etapas são executados de forma paralela, resultando em um *pipeline* de 7 estágios de execuções e a etapa de *Commit* executada de forma assíncrona, que pode ser visto na Figura 3.2.

Na etapa de desenvolvimento, o gerador permite que sejam feitas mudanças em seus parâmetros estruturais, podendo modificar o tamanho da cache L1, número de registradores físicos e os estágios de *Fetch* e de *Decode* também podem ser configurados, modificando a quantidade de instruções lidas a partir da cache de instruções, e a quantidade de instruções decodificadas em um único ciclo, a fim de aumentar a quantidade de recursos do hardware utilizados.

Nos estágios de *Decode/Rename*, *Rename/Dispatch* e *Issue*, ocorre a execução do algoritmo de escalonamento dinâmico. Uma técnica baseada em *hardware* que realiza o despacho das instruções à medida que se tornam disponíveis os recursos para sua execução como, por exemplo, os dados de entrada, saída e unidades de execução. A fim de permitir que instruções que não tenham dependências de dados possam ser executadas paralelamente, garantindo assim, a consistência no resultado da execução do programa e apresentando ganho de desempenho.

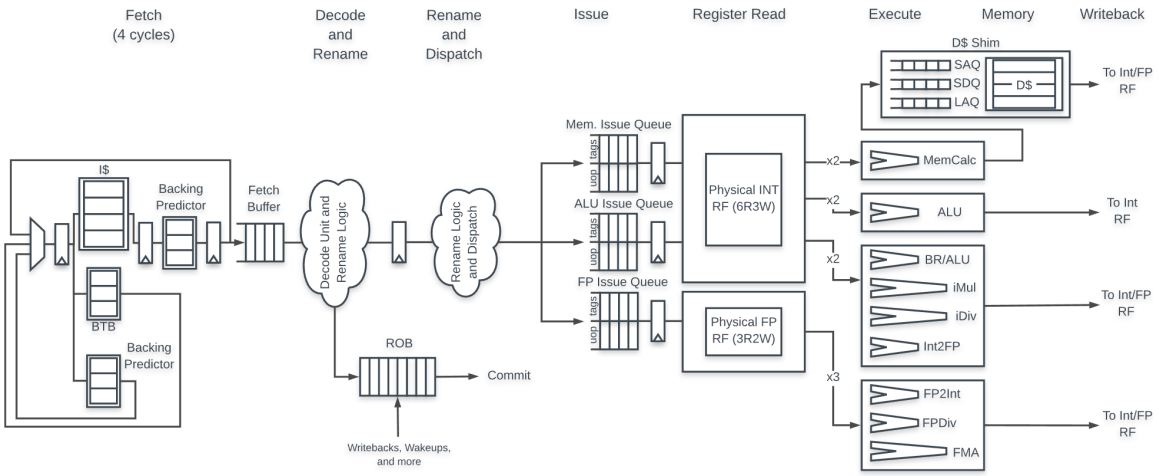

Figura 3.2: Esquema do *pipeline* do processador Berkeley Out-of-Order Machine (BOOM) [Celio et al., 2021]

3.2.1 Predição de desvio

A etapa de predição de desvio é implementada no estágio de *Fetch*, utilizando dois níveis de predição de desvio. O primeiro, denominado de preditor de próxima linha, chamado de NLP (do inglês *Next-Line Predictor*), opera com base em um *Buffer* de destino de ramificação (do inglês *Branch Target Buffer* (BTB)), onde é armazenado a decisão padrão com base no endereço nas instruções de desvios já analisadas, sendo que o objetivo desse nível de preditor é de manter um circuito de baixa complexidade que apresenta alta precisão para execução de laços de repetições.

Para execuções com desvios condicionados mais complexos, o BOOM utiliza um segundo preditor que supervisiona as decisões mantidas pelo BTB. Denominado de Preditor de Apoio (do inglês, *Backing Predictor* (BPD)), o modelo do preditor instanciado é definido pelo projetista, que pode escolher entre 5 preditores, como, por exemplo, o Gshare e o preditor de história global TAGE [Seznec, 2011]. Devido a maior complexidade do BPD, o *Fetch* prolonga-se por 4 ciclos, no entanto, é iniciado em paralelo ao BTB.

Quando uma instrução de desvio condicional é encaminhada a partir da cache de instruções, é realizado um acesso ao BTB em busca de uma correspondência, e em caso positivo, o valor do contador de programa é redirecionado após apenas 1 ciclo, e em caso de falha na busca, uma nova entrada é alocada no BTB para essa instrução e o BPD calibra a decisão salva no BTB no último ciclo do estágio *Fetch*.

3.2.2 Dependência de dados

Dependência de dados caracterizam-se quando duas ou mais instruções ativas no *pipeline* realizam requisições ao mesmo registrador, gerando inconsistência no resultado caso sua execução seja realizada em ordem oposta. Tais dependências podem ser distribuídas em três classificações como *Read after write (RAW)*, *Write after read (WAR)* e *Write after write (WAW)* [Hennessy and Patterson, 2017].

A primeira dependência refere-se a uma situação, em que uma determinada instrução requisita o dado de um registrador que ainda não foi gerado por outra instrução, realizando uma leitura incorreta do dado, pela primeira instrução. Servindo como base o exemplo na Figura 3.3, a instrução *I2*, só deverá ser despachada após a execução da instrução *I1*, devido à dependência no registrador *R1*. Enquanto que a segunda, refere-se a uma situação em que um registrador tem seu valor alterado antes que outra instrução possa acessá-lo, realizando uma leitura incorreta do dado, pela segunda instrução. No exemplo apresentado, a instrução *I3*, só deverá ser despachada após a execução da instrução *I2*, devido à dependência no registrador *R2*. Por fim, a terceira, refere-se a uma situação em que duas ou mais instruções tentam modificar o valor de um mesmo registrador, e caso a escrita seja feita na ordem inversa, o valor final do registrador será comprometido. No exemplo apresentado, a instrução *I5*, só deverá ser despachada após a execução da instrução *I4*, devido à dependência no registrador *R3*.

O primeiro computador a apresentar um algoritmo de escalonamento dinâmico para realizar o tratamento de *Data Hazard* foi o CDC 6600 [Thornton, 1980], por meio de uma estrutura de *Scoreboard*, onde os registradores de destino e de entrada eram armazenados em tabelas, no entanto, essa estrutura gerava muitos comandos de *"stall"* durante o tratamento de dependência para garantir a integridade dos dados. Anos mais tarde, a IBM anunciou o System/360 Model 91 [Anderson et al., 1967], como um competidor do CDC 6600, introduzindo o algorítimo de Tomasulo, onde apresentou o conceito de renomeação de registradores para prevenir *Data Hazard* dos tipos *WAR* e *WAW*, mapeando os registradores conflitantes para outros registradores físicos [Tomasulo, 1967]. Para facilitar a visualização da técnica de remapeando dos registradores, na Figura 3.3, pode ser vista a sequência de instruções criadas após a etapa de remapeamento.

```
1   I1: R1 <= Lw address(0)              R1 <= Lw address(0)
2   I2: R5 <= R2 + R1                    R5 <= R2 + R1
3   I3: R2 <= R6 + R7          =>        T1 <= R6 + R7
4   I4: R3 <= R8 * R9                    R3 <= R8 * R9
5   I5: R3 <= R8 + R10         =>        T2 <= R8 + R10
```

Figura 3.3: Exemplo de instruções antes e após a etapa de renomeação de registradores, para corrigir dependência de dados dos tipos *WAR* e *WAW*

3.2.3 Escalonamento dinâmico de instruções

Para realizar o processo de escalonamento dinâmico, o núcleo BOOM divide o processo em duas etapas, a primeira de renomeação de registradores e a segunda de despacho dinâmico. Para realizar o processo de renomeação de registradores, o núcleo apresenta projeto design de "renomeação explícita", onde mantém: um banco de registro físico (em inglês, *Physical Register File* (PRF)), responsável por armazenar explicitamente os nomes dos registradores físicos direcionados para cada registrador lógico decodificado das instruções; tabelas de mapa de renomeação (abreviado em inglês, *Map Tables*), que mantém salvo o estado dos registradores mapeados para cada ramificação especulativa identificada; e por fim, duas tabelas, sendo que a primeira armazena os registradores físicos que não estão alocados, denominada de *Freelist*, enquanto que a segunda armazena os status dos registradores físicos, denominada de *busy table*.

Dessa forma, ao passar para o estágio de *Decode*, a controladora realiza uma requisição de endereçamento para cada registrador vinculado à instrução a ser decodificada. No caso dos registradores de origem é realizado uma requisição ao PRF, buscando a TAG do registrador físico na *Map Table* da ramificação correspondente. No caso do registrador de destino, a fim de eliminar qualquer dependência de dados, sempre é realizado uma requisição no *Freelist* solicitando uma realocação do registrador, dessa forma todas as instruções que tenham o registrador *rd* ativo, acarretará em uma modificação no *Map Table*. Devido ao formato rígido das instruções do padrão RV64G ISA, os *Map Table* podem ser acessados durante o estágio de *Decode*, isso permite que seja combinado com o estágio de *Rename*.

O processo de despacho dinâmico, realizado nos estágios de *Dispatch* e *Issue*, é responsável por analisar as instruções decodificadas, identificando os recursos e despachar para execução à medida que se tornam disponíveis. Para isso são mantidos um buffer de reordenamento (em inglês, *Reorder Buffer* (ROB)) e um *Issue Slot* para cada ramo de execução (inteiro, ponto flutuante ou acesso à memória).

O ROB opera similarmente a uma lista circular, como pode ser visto na Figura 3.4, e é composto principalmente pelo campo *Val* responsável por indicar se a informação na linha é válida, o campo *Bsy*, que indica que o registrado de destino se mantém bloqueado, isto é, a instrução ainda está em execução, o campo *Exc*, por indicar se é uma exceção, caso seja, quando chegar no topo do ROB é lançada a exceção no *pipeline*, o campo *Uopc*, que armazena o pacote contendo a instrução, os registradores de destino e origem, e por fim, *Brmask* que indica qual conjunto especulativo a instrução está associada.

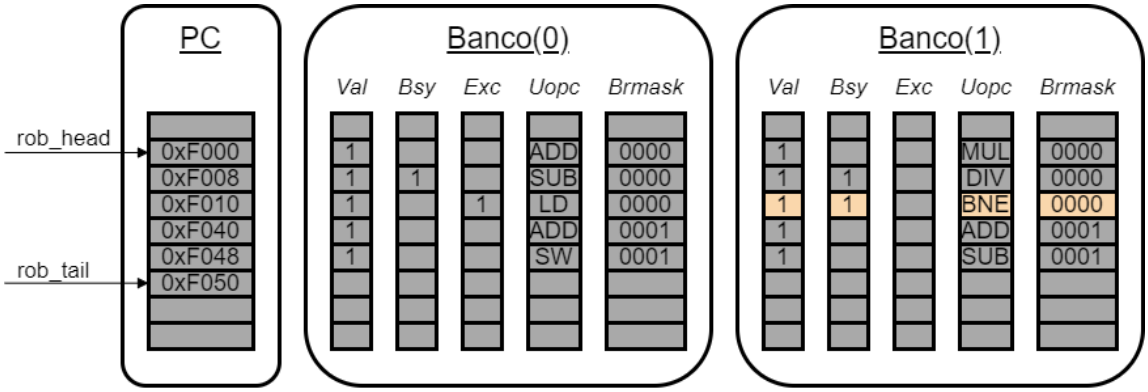

Figura 3.4: Diagrama do *Reorder Buffer* de dois bancos

Quando uma instrução é emitida do estágio de *Decode/Rename*, é gerada uma nova entrada no ROB, que será adicionada no final da lista, indicado pelo endereço apontado por *rob_ tail*, onde o mesmo será realocado para o próximo endereço circular. Quando a instrução apontada por *rob_ head* não está mais ocupada, ela pode ser confirmada, ou seja, seu estado pode ser enviado para a memória. Após a emissão, a instrução é removida do ROB e o endereço *rob_ head* reajustado. Uma característica das instruções RoCC é o fato de não suportarem execução especulativa, dessa forma o projeto BOOM apenas emite para execução, quando estiverem na posição apontada pelo *rob_ head*.

Após ser gerada uma nova entrada no ROB, a instrução é repassada para a Unidade de Emissão (em inglês *Issue Unit*). Como pode ser visto na Figura 3.2, o núcleo BOOM particiona o *pipeline* em 3 ramos após as filas de emissões (em inglês *Issue Queues*), isolando a execução entre instruções de operandos inteiros, de ponto flutuante ou instruções de acesso à memória. Esses 3 *Issue Queues* operam de forma paralela, sendo responsáveis por reter as instruções até todos os operandos estiverem prontos e a unidade de execução livre. Cada *Queues* associa um bit a cada operando, para representar o estado do registrador, quando todos os bits de estados estiverem prontos, será emitido o sinal de solicitação ao componente de execução apropriado e aguardará ser emitido.

3.3 Rocket Custom Coprocessor Interface (RoCC)

O *Rocket Custom Coprocessor Interface* (RoCC) é uma interface que estende o núcleo no SoC, permitindo uma comunicação desacoplada entre o núcleo e os coprocessadores [Asanović et al., 2016]. No caso específico desse trabalho desenvolvido, os coprocessadores atuaram como aceleradores, por apresentarem arquiteturas especializadas em multiplicação paralela. A interface RoCC proporciona uma conexão simples e direcional, e como pode ser visto na Figura 3.5, é composta principalmente por quatro canais de comunicação, dois para troca de dados entre o núcleo e o coprocessador, e dois canais para troca de dados entre o coprocessador e o sistema de memória.

A comunicação entre o núcleo e o coprocessador é baseada no algoritmo FIFO utilizando sinais pronto e válido. Dessa forma, para iniciar uma instrução de comando, o núcleo aciona o sinal válido e disponibiliza os dados, em seguida, espera até que o coprocessador eleve o sinal de pronto, onde a transferência é considerada aceita se ambos sinais

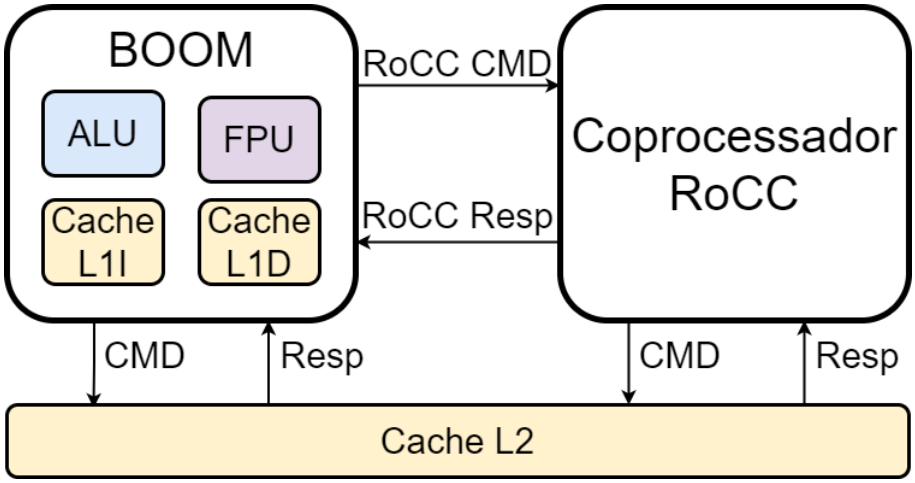

Figura 3.5: Diagrama do núcleo BOOM com um coprocessador conectados pela interface RoCC

estiverem no nível lógico 1, no mesmo ciclo de clock. Para a resposta do coprocessador é mantida uma estrutura análoga, mas em sentido oposto.

Uma vez que essa interface foi projetada para estender o núcleo Rocket, um processador de execução em ordem de *pipeline* clássico de 5 estágios, a resposta ocorre na mesma ordem das requisições recebidas. Como nesse trabalho desenvolvido será utilizado o núcleo BOOM, isso significa que, ao ser executado instruções de comando RoCC em sequência, terão suas emissões realizadas em ordem, pois o ROB só poderá confirmar e emitir a próxima instrução, a medida que identificar o comando de resposta por parte do coprocessador.

As instruções de comando apresentam comprimento de 32-bits e assemelham-se ao formato R-*type*, como mostrado na Figura 3.6, no entanto, os 3-bits do campo *funct3* são reservados como Bits de validade, indicando se os registradores de origem e de destino devem ou não ser lidos. Para redirecionar a instrução para a porta RoCC, presente no núcleo, o campo *opcode* é fixado entre 0 e 3, valores reservados pelo RISC-V ISA para instruções customizadas [Waterman and Asanovic, 2019]. Como as instruções customizadas não são expostas através do *GNU binutils assembler*, é necessário descrever essa codificação diretamente em linhas de código *assembly*.

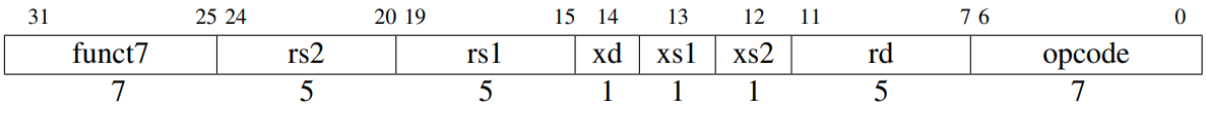

Figura 3.6: Formato da instrução de controle da interface RoCC

A interface de memória disponibilizada pelo RoCC, permite que o coprocessador se comunique diretamente com o barramento de memória, dentro do ecossistema Chipyard, o que representa uma conexão direta à cache L2, dessa forma evitando uma sobrecarga à controladora de memória do núcleo. Ao contrário das instruções de comando, as instruções de acesso à memória não podem ser confirmadas em ordem, pois o coprocessador não pode manter o cache aguardando, portanto, ele deve aceitar a resposta sem a possibilidade de

adiar a transação. Isso significa que o coprocessador deve aceitar todas as respostas da memória assim que a linha válida for alta. Com esta interface, a cache L2 conduz os dados e os sinais válidos e o coprocessador só precisa verificar se a linha válida está alta e aceitar os dados recebidos.

Além dos quatro canais principais, a interface RoCC também fornece canais para permitir funcionalidades avançadas como, por exemplo, conectar diretamente o *Page Table Walker* (PTW) ou a Unidade de Ponto Flutuante (FPU) do núcleo com o coprocessador, também pode permitir link direto com o sistema de memória externa. Esses canais extras fazem parte da chamada interface RoCC estendida.

3.4 Coprocessador Hwacha

A plataforma Hwacha fornece um gerador de um coprocessador vetorial, parametrizável de código aberto[Lee et al., 2015]. O projeto é integrado ao ecossistema Chipyard, sendo desenvolvido com base em uma arquitetura de acesso à memória e execução desacopladas [Smith, 1984], permitindo assim que o Hwacha realize requisições ao barramento de memória em paralelo com a ativação da unidade de execução vetorial. A arquitetura foi projetada para permitir que o desenvolvedor configure, em tempo de compilação, a precisão da operação de ponto flutuante e a opcionalmente a execução predicada das instruções.

O coprocessador apresenta uma interface de *Frontend* independente, que realiza um estágio de *Fetch* das instruções vetoriais, fazendo com que o núcleo possa ativá-lo com apenas uma instrução, associada a um valor de contador de programa (PC) que aponta para o início do bloco de instruções vetoriais. O estágio de *Fetch* independente, permite que o núcleo continue realizando trabalho útil, enquanto o Hwacha esteja executando as operações vetoriais.

Como mostrado na Figura 3.7, o Hwacha é composto por 4 componentes principais, sendo uma cache L1 exclusiva, armazenando apenas as instruções vetoriais, uma Unidade de *Runahead* Vetorial (VRU), as unidades responsáveis por realizar a execução das operações primitivas, denominadas de Pistas Vetoriais, por fim, uma Unidade Escalar que realiza o estágio de *Fetch* das instruções vetoriais e, é responsável pela ativação das Pistas Vetoriais. A comunicação com o núcleo é realizada por uma unidade RoCC, que distribui as instruções para a Unidade Escalar e a VRU.

A estrutura das Pistas Vetoriais é dividida em dois componentes, a Unidade de Execução Vetorial (em inglês, *Vector Execution Unit* (VXU)), e a Unidade de Memória Vetorial (em inglês, *Vector Memory Unit* (VMU)). O VXU engloba os arquivos de registro vetorial (VRF) e predicado (PRF) e as unidades primitivas de processamento, separadas em unidades de meia precisão, precisão simples e dupla. O VXU é organizado em quatro bancos, onde cada banco é capaz de realizar uma operação primitiva e contém uma SRAM que forma uma parte do VRF e do PRF. Uma barra cruzada conecta os bancos às unidades funcionais, compartilhando as mesmas linhas de operando, predicado e resultado, enquanto que o VMU é responsável por controlar o movimento de dados entre a VXU e a cache L2, carregando o vetor e alimentando as unidades de processamento.

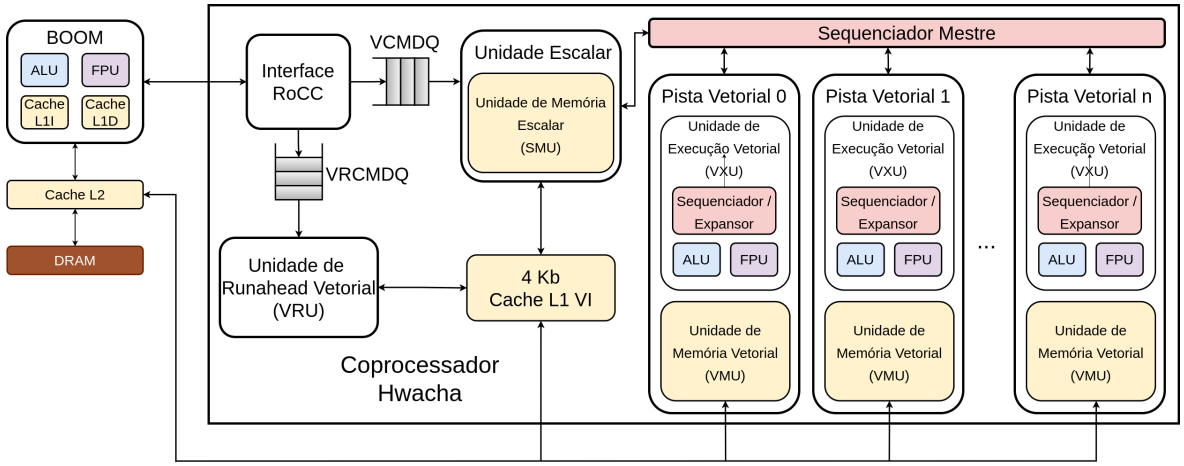

Figura 3.7: Visão geral do coprocessador Vetorial Hwacha.

3.4.1 Fluxo de dados

O modelo de programação utilizado para a ativação do coprocessador, separa as instruções em dois blocos. O primeiro bloco denominado de *thread* de controle, é executado pelo núcleo, onde são mapeadas as instruções escalares, responsáveis pela etapa de configuração a Unidade Vetorial do Hwacha, além das instruções padrão do núcleo. O segundo, onde são mapeadas apenas as instruções de execução da Unidade Vetorial, é responsável pelo fluxo de execução e é denominado de *thread* de trabalho.

A unidade RoCC distribui as instruções da *thread* de controle proveniente do núcleo para a Unidade Escalar e a VRU, através do Vector Command Queue (VCMDQ) e do Vector Runahead Command Queue (VRCMDQ), respectivamente. Após ser decodificada a instrução para ativação, a Unidade Escalar inicia o estágio de *Fetch* das instruções vetoriais, a partir do valor de PC informado, continuando até atingir a instrução que sinaliza o final do bloco de busca.

No estágio de *decode*, a Unidade Escalar despacha as instruções para o sequenciador mestre, em seguida os dados dos vetores são despachados para cada Pista Vetorial. O sequenciador mestre emite operações para todas as pistas de forma síncrona, no entanto, cada pista é executada de forma desacoplada uma da outra. Em caso do comprimento vetor ser maior que o número de bancos para processamento, a Unidade de Memória Escalar (em inglês, *Scalar Memory Unit* (SMU)) irá carregar o vetor completo e salvar em registradores vetoriais, para isso é mantido um registrador de comprimento do vetor configurável que informa quantos dados devem ser carregados a partir da memória. Por fim, o sequenciador mestre irá separar os dados em blocos contínuos para serem despachados para as Pistas Vetoriais.

Um sequenciador interno é responsável por monitorar o progresso de cada operação ativa dentro de cada pista específica. Esse sequenciador realiza o despacho dinâmico das instruções, no entanto, ao contrário do núcleo BOOM, não apresenta uma etapa de realocação do registrador, mantendo as instruções paradas até que seus registradores estejam desocupados. Após liberado, as instruções seguem ao expansor, que encaminha as instruções para as respectivas unidades de execução. Por fim, os resultados são depositados em

filas, onde o sequenciador mestre monitora a retirada de forma assíncrona, e sequencializa os dados para serem salvos corretamente no registrador vetorial.

3.4.2 Conjunto de instruções

As instruções da *thread* de controle Hwacha, são encaminhadas pelo núcleo, dessa forma, seguem o padrão de instruções de comando da interface RoCC, discutido na Seção 3.3. A API de controle é dividida em três tipos de instruções. O primeiro referente as instruções de Configuração do Vetor, responsável por configurar os parâmetros de execução do VXU. O segundo tipo é para as instruções de Movimento de Dados, que transferem vetores e valores escalares, entre a *thread* de controle e a *thread* de trabalho. Por fim, a instrução de Busca do Vetor, que informa o valor de PC que inicia a sequência das instruções da *thread* de trabalho.

São duas às instruções de Configuração do Vetor representadas pelos nomes *VSETCFG* e *VSETVL*. A primeira é responsável por configurar a Unidade Vetorial, alocando os registradores a serem utilizados na operação, para isso apenas o registrador *rs1* é valido. O padrão utilizado no registrador *rs1*, pode ser visto abaixo.

- *rs1*[8:0]: Número de registradores de meia precisão;
- *rs1*[13:9]: Número de registradores predicados;
- *rs1*[22:14]: Número de registradores de precisão simples;
- *rs1*[31:23]: Número de registradores de precisão dupla;

A instrução *VSETVL* configura o comprimento do vetor, para isso o registrador *rs1* é responsável por informar o valor requisitado, e o valor do comprimento do vetor alocado é salvo no registrador *rd*. Dessa forma, caso seja requisitado um valor maior que o disponível na arquitetura, o valor de retorno será menor que o requisitado, necessitando de uma nova chamada por parte do programador para processar os dados faltantes.

A instrução de Busca do Vetor é representada pelo nome *vf* nela, o endereço de PC referente ao bloco destinado às instruções vetoriais é transmitido pelo registrador apontado em *rs1*, e os valores campos *funct7* e *rd* são combinados, formando um valor imediato de 12-bits, que após decodificado, é somado com o endereço informado para encontrar o endereço final.

Por fim, as instruções *VMCS* e *VMCA* compõem as instruções do tipo Movimento de Dados. A instrução *VMCS* move o conteúdo do registrador *rs1* no processador de controle para um registrador de vetor compartilhado (*vs*). A instrução *VMCA* move o conteúdo do registrador *rs1* no processador de controle para um registrador de endereço vetorial (*va*).

Como as instruções da *thread* de trabalho não são repassadas através da interface RoCC, elas não tem seu comprimento limitado, dessa forma apresentam formato de 64-bits. As instruções são divididas em 5 formatos, que podem ser visualizadas na Figura 3.8.

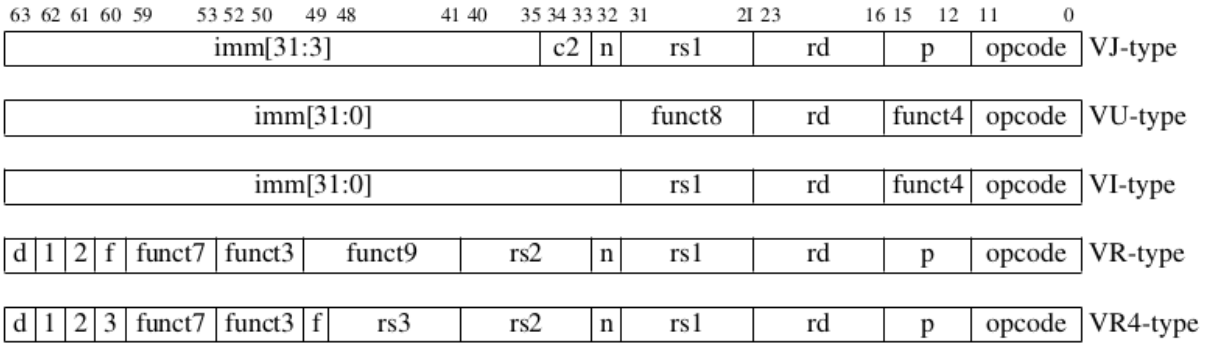

Figura 3.8: Formatos das instruções da *thread* de trabalho [Lee et al., 2015]

Como a ISA da *thread* de trabalho é composta por mais de 200 instruções, a fim de simplificar a apresentação, abaixo constam apenas as instruções que serão utilizadas nesse trabalho:

- *vstop*: Delimita o final do bloco de instruções vetoriais;
- *vlw*: Carrega um bloco da memória, a partir do endereço apontado por *va*, no registrador vetorial (*vv*) informado;
- *vsw*: Carrega os dados armazenado no registrador vetorial (*vv*) informado, no endereço de memória apontado por *va*;
- *vfmadd.s*: Realiza a multiplicação e acumulação dos dados armazenados no registrador vetorial (*vv*) informado, pelo escalar apontado em *vs*.

Na Figura 3.9 é apresentado um exemplo de código para a ativação do coprocessador Hwacha, onde é realizado o processo de multiplicação e acumulação de um vetor por um escalar, análogo ao apresentado na Seção 2.2. Para realizar a multiplicação, inicialmente a variável escalar é copiada para o registrador de vetor compartilhado (linha 1), dentro do escopo do loop, o comprimento do vetor é configurado (linha 3), e os endereços dos vetores carregados (linhas 4 e 5), por fim a instrução de Busca do Vetor é acionada para ativar o Hwacha (linha 6), encerrando a etapa de configuração. No bloco de instruções da *thread* de trabalho, os vetores são carregados a partir do sistema de memória e carregados nos registradores vetoriais (linhas 11 e 12), em seguida é realizado o procedimento de multiplicação e acumulação (linha 13), por fim, o resultado do registrador é salvo no sistema de memória (linha 14).

3.5 Acelerador Gemmini

A plataforma Gemmini fornece um gerador full-stack de um acelerador para aplicações de Redes Neurais Profundas (em inglês, *Deep Neural Network* (DNN)), de código aberto, com total integração ao ecossistema Chipyard [Genc et al., 2021]. Foi desenvolvido baseado no modelo de Arrays Sistólicos (SA), onde cada elemento de processamento (PE) consegue executar operações de multiplicação e de acumulação (MAC) dos produtos parciais.

```
1     vmcs vs0, a1                          // vs0 = a
2 Loop:
3     vsetvl t0, a0                         // a0 = DIM
4     vmca va0, a2                          // va0 = &x[0]
5     vmca va1, a3                          // va1 = &y[0]
6     vf thread_trabalho                    // Acionar Hwacha
7     sub a0, a0, t0                        // a0 -= retorno do
        vsetvl
8     bnez a0, Loop
9
10 thread_trabalho:
11    vlw vv0, va0                          // vv0 = x[i]
12    vlw vv1, va1                          // vv1 = y[i]
13    vfmadd.s vv1, vs0, vv0, vv1           // vv1 += a*vv0;
14    vsw vv1 ,va1                          // y[i] = vv1
15    vstop
```

Figura 3.9: Código de multiplicação entre vetores mapeados para o coprocessador Hwacha

Como pode ser visto na Figura 3.10, o Gemmini apresenta um Scratchpad que armazena explicitamente blocos das matrizes de entrada e de saída do SA, a fim de facilitar a reutilização de dados, e um acumulador equipado com unidades somadoras, para reduzir a necessidade de escrita dos resultados parciais no Scratchpad. Ambas estruturas são implementadas em SRAM, a fim de prover alta largura de banda. O acelerador, também, inclui estruturas auxiliares capazes de realizar operação de transposição e de multiplicação por um escalar antes de carregar a matriz no SA, dessa forma, o Gemmini permite que sejam aceleradas operações de multiplicação de matrizes, de expressão C = alpha * A * B + beta * D. Por fim, é oferecido suporte a funções de ativação como ReLU ou ReLU6, após a etapa de acumulação [Genc et al., 2021].

O gerador Gemmini inclui dois DMAs, um responsável por copiar as matrizes de entrada da cache L2 para os Scratchpad e outro, por mover a matriz resultante do Scratchpad para a cache L2. Ambos DMAs, operam com endereços virtuais e compartilham o acesso a um TLB para traduzi-los em endereços físicos. A fim de mascarar a latência de memórias de hierarquias inferiores, o acelerador permite que a movimentação de dados seja realizada em paralelo com a execução do SA, para isso, é implementado um ROB, análogo ao apresentado na Seção 3.2.3, responsável por garantir a resposta em ordem.

Para facilitar a ativação do acelerador, a plataforma oferece suporte a bibliotecas de baixo nível, escritas em linguagem C, fornecendo ao programador funções comuns em operação de DNN como multiplicação e adições de matrizes, convoluções (com ou sem pooling), entre outros. A API é gerada em conjunto com a arquitetura, baseado em seus parâmetros como, por exemplo, o tamanho do Scratchpad e do Acumulador, dimensão dos arrays sistólicos. A biblioteca também busca atender os casos onde as dimensões das matrizes são maiores que as dimensões dos SA, para isso, é realizado o processo de *tiling* da multiplicação das matrizes, visando otimizar a movimentação de dados entre a cache e o Scratchpad, maximizando a reutilização de dados.

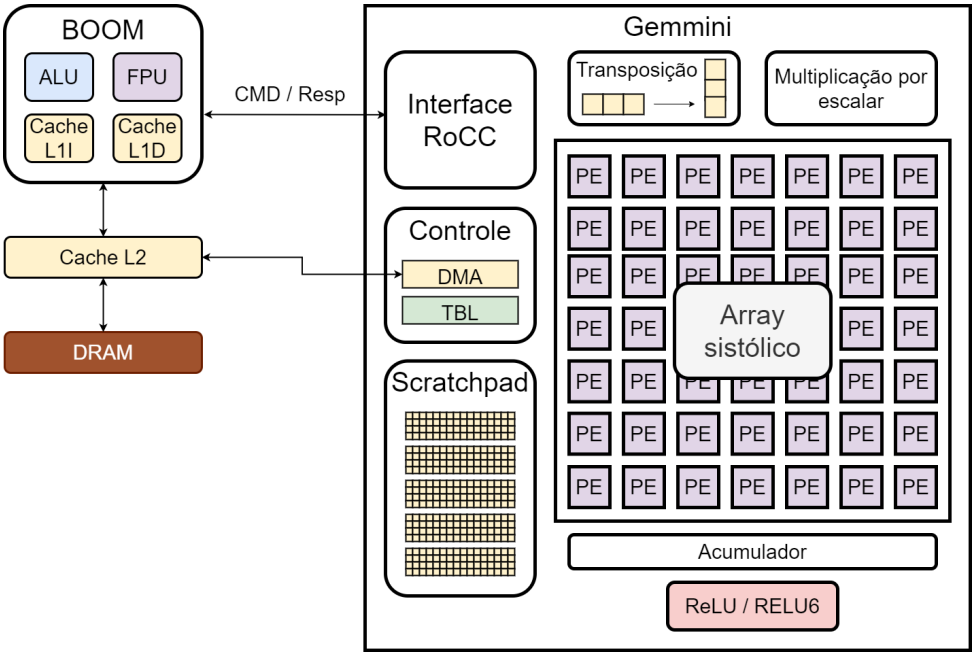

Figura 3.10: Visão geral do Acelerador Gemmini

3.5.1 Array sistólico

Arrays sistólicos [H.T.Kung, 1979, 1982] são estruturas de hardware, onde cada unidade primitiva de processamento, é replicada e distribuídas em formato de matriz. Essas unidades são denominadas de células ou de elementos de processamento (PE) e apresentam conexões com seus vizinhos imediatos, sendo capazes de realizar a mesma operação com dados diferentes de forma paralela. Os SA buscam aumentar o paralelismo de computação, permitindo que os resultados parciais de uma operação sejam repassados diretamente às outras unidades por meio dos registradores locais, dessa forma os dados fluem entre as unidades, sem precisar realizar operações de leitura e escrita na memória, o que o torna muito eficiente e facilmente escalável.

O fluxo de controle mais difundido em arquitetura SA, consiste em transmitir duas matrizes simultaneamente através do array. Nesse caso, uma matriz é carregada a partir da primeira linha e transmitida no sentido de cima para baixo, com uma posição deslocada em cada linha, enquanto outra matriz é carregada a partir da primeira coluna e transmitida no sentido da esquerda para a direita. A cada ciclo de clock, os dados são propagados nos sentidos específicos, a fim de que as entradas de operandos do PE sejam carregadas. Os dados sobrepostos são multiplicados e os resultados parciais acumulados, os valores finais da multiplicação são encontrados quando uma linha e uma coluna tenham percorrido, por inteiro, uma unidade de processamento.

O Gemmini implementa duas hierarquias de SA, onde a camada interna é composta por um conjunto de PEs, e cada PE é diretamente conectado aos elementos adjacentes, este conjunto é chamado de *Tile*, como pode ser visto na Figura 3.11. A camada externa, denominada de *Mesh*, é formada pela combinação de *Tiles*, ligados por meio de registradores de *pipeline* explícitos. Tanto os PEs quanto os *Tiles*, apenas podem se comunicar com os vizinhos imediatos. O gerador oferece suporte a dois tipos de fluxo de dados: es-

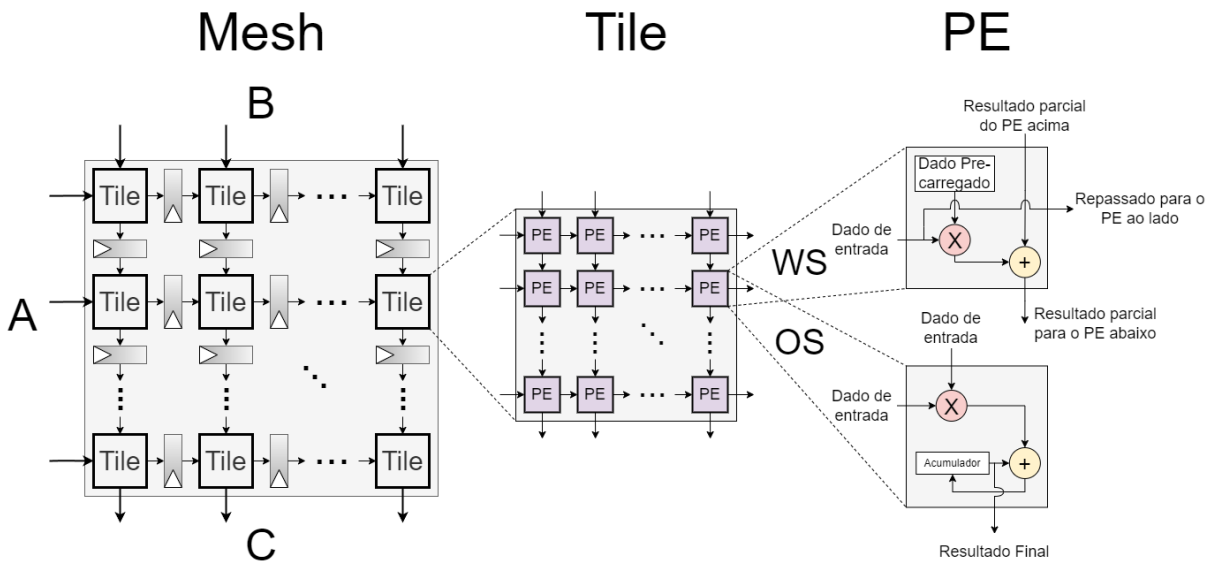

Figura 3.11: Visão geral do SA e unidades MACs do Acelerador Gemmini

tacionário de peso (**WS**) ou de saída (**OS**). Após realizar as operações de MAC, os PEs podem armazenar os resultados parciais no acumulador ou repassar para os PEs vizinhos, dependendo do fluxo de dados escolhidos, usando apenas comunicação local entre registradores.

O fluxo de dados **WS** é projetado para minimizar os acessos à matriz de pesos no Scratchpad e, consequentemente, minimizar acessos a cache. Neste fluxo, a matriz de peso (**B**) é lida e os dados pré-carregados no PE correspondente, permanecendo estacionário enquanto a outra matriz (**A**) é propagada pelo SA, a fim de executar o maior número possível de operações que usam o mesmo peso. Assim, é indicado quando são realizadas múltiplas operações utilizando uma mesma matriz de entrada.

Para implementar o fluxo **WS**, inicialmente, cada PE do Gemmini realiza a operação de multiplicação, em seguida, soma com o resultado recebido do PE acima, concluindo assim a operação de MAC, por fim, o resultado é repassado para o PE da linha abaixo. Os resultados emitidos pelos PEs da última linha são encaminhados ao Scratchpad. Na Figura 3.12, é apresentado o diagrama de execução de uma multiplicação entre matrizes 2x2, para que cada elemento da matriz propagada (**A**) atinja o PE no ciclo de clock correto, o Gemmini apresenta registradores de deslocamento na entrada das linhas. Em caso de operação no formato C = A * B + D, a matriz **D** é propagada e somada apenas na primeira linha.

O fluxo de dados **OS** é projetado para minimizar a escrita de dados no Scratchpad. Para isso, ambas as matrizes são propagadas para manter o acúmulo de somas parciais no mesmo endereço de ativação do SA. Neste fluxo, os PEs encaminham diretamente o resultado da multiplicação ao acumulador correspondente, após a propagação total das matrizes, os valores armazenados nos acumuladores são salvos nos Scratchpad.

Na Figura 3.13, também é apresentado o diagrama de execução de uma multiplicação entre matrizes 2x2, mas no fluxo de dados **OS**. Para realizar uma operação no formato C = A * B + D, a matriz **D** é pré-carregado no acumulador. Esse fluxo é indicado quando são realizadas consecutivas multiplicações de matrizes, onde os resultados dessas multipli-

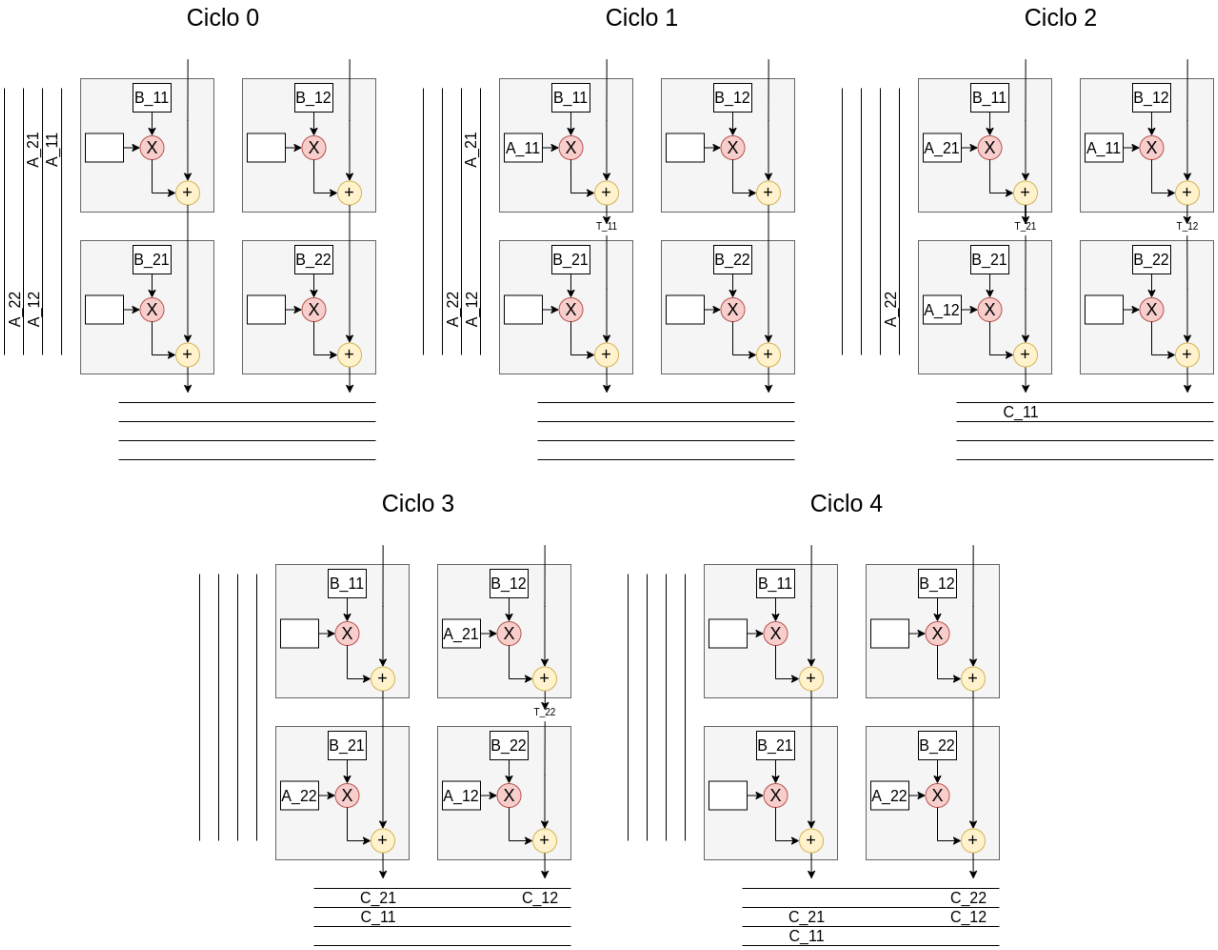

Figura 3.12: Diagrama de execução de uma multiplicação de matrizes 2x2 (A × B = C), utilizando o fluxo estacionário de peso (WS), onde T_xx representa o resultado parcial transmitido

cações são somados entre si, nesse caso, o fluxo permite que os resultados intermediários sejam acumulados até que o resultado final seja calculado.

3.5.2 Sistema de controle

O Gemmini opera como um acelerador RoCC, utilizando a interface RoCC para se comunicar com o Core, através de instruções RISC-V customizadas. Como apresentado na Seção 3.3, essa interface é composta por 4 canais de comunicação, para isso, o sistema de controle do Gemmini é subdividido em 3 controladoras autônomas, permitindo que as etapas de execução e de acesso à memória operem de formas desacopladas.

A primeira controladora é responsável por gerenciar tanto a execução dos SAs quanto a ativação das estruturas auxiliares, onde essa controladora permite que o programador defina o *pipeline* de execução, indicando qual fluxo de dado a ser utilizado, quais serão as matrizes propagadas pelo SA, e quais funções auxiliares que deverão ser ativadas. A segunda, é responsável por controlar o fluxo de dados provenientes da cache e, por fim, a terceira, por salvar o resultado do Scratchpad na cache.

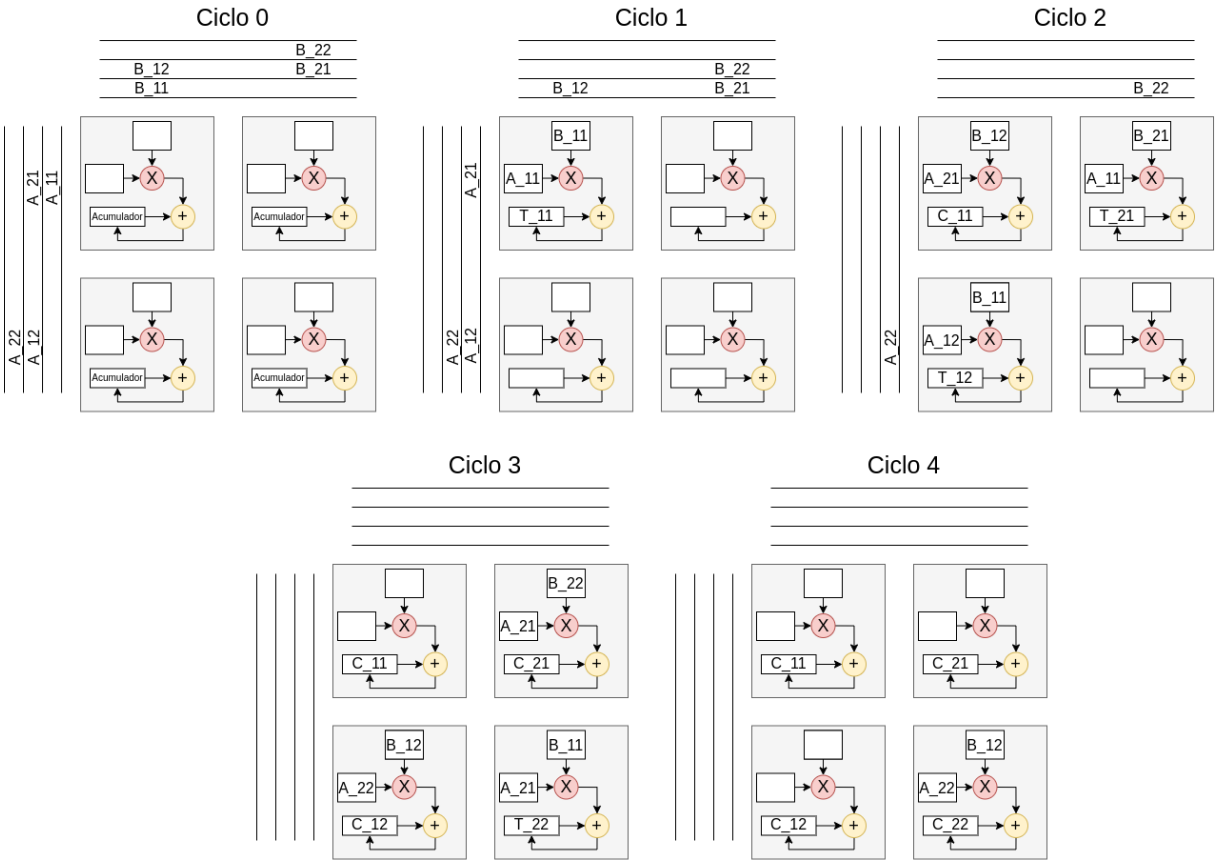

Figura 3.13: Diagrama de execução de uma multiplicação de matrizes 2x2 (A × B = C), utilizando o fluxo estacionário de saída (OS), onde T_xx representa o resultado parcial acumulado

Cada controladora é ativada por um conjunto de instruções especificas. Após serem emitidas pelo núcleo, as instruções são decodificadas e encaminhadas para a controladora específica. Dessa forma, a ISA do Gemmini é dividida em instruções de configuração, instruções de ativação do SA e instruções de movimentação de dados.

Como apresentado na Seção 3.3, as instruções seguem o formato R-*type*, no entanto, o Gemmini opera apenas com os registradores de origem ativos, para isso, o campo *funct3* é sempre fixado em 011. Para repassar as informações ao acelerador, os registradores *rs1* e *rs2* seguem um padrão específico para cada tipo de instrução, para isso, a API do Gemmini fornece funções em linguagem C para construir as codificações necessárias, chamando essas instruções.

Das funções de instruções de configuração existentes na API do Gemmini podemos destacar o gemmini_extended_config_ex(), responsável por configurar o *pipeline* de execução. Abaixo consta os padrões exigidos nos registradores:

- *rs1*[2]: Estacionário de peso (1) ou de saída (0);
- *rs1*[4:3]: Ativar função ReLU, ReLU6;
- *rs1*[8]: Booleano, matriz A transposta?
- *rs1*[9]: Booleano, matriz B transposta?

- *rs1*[31:16]: Dimensão da matriz A;
- *rs1*[63:32]: número de bits pelos quais o resultado acumulado é deslocado para a direita ao sair do acumulador;
- *rs2*[31:0]: número de bits pelo qual o resultado acumulado é deslocado para a direita ao sair da matriz sistólica;
- *rs2*[63:32]: número de bits pelo qual 6 deve ser deslocado para a esquerda antes de aplicar ReLU6;
- *funct7*: 0;

Para as funções de movimentação de dados existentes na API do Gemmini, podemos destacar o gemmini_mvin(), responsável por mover uma matriz do tamanho do SA da cache para o Scratchpad e o gemmini_mvout(), por mover a matriz do Scratchpad para a cache. A API também fornece funções para mover blocos de dados de tamanho arbitrário. Abaixo consta os padrões exigidos:

- *rs1*[63:0]: Endereço DRAM virtual para carregar no Scratchpad;
- *rs2*[31:0]: Endereço do Scratchpad;
- *rs2*[47:32]: Número de colunas;
- *rs2*[63:48]: Número de linhas;
- *funct7*: 2 (mvin) ou 3 (mvout);

Devido ao comprimento da instrução, a etapa de ativação é dividida em duas instruções, que devem ser acionadas em sequência. Dessa forma, primeiro deve ser acionada a função gemmini_extended_preload(), responsável por carregar a matriz C e a matriz D no SA, em seguida pela função gemmini_extended_compute_preloaded() ou gemmini_extended_compute_accumulated(), onde essas funções são responsáveis por carregar as matrizes A e B e por iniciar a multiplicação no fluxo **WS** ou **OS**, respectivamente. Para os casos onde a matriz D não é requisitada, o Gemmini permite que seja repassado o endereço *NULL*, dessa forma Gemmini mapeia os pesos do SA para o valor 0. Abaixo consta os padrões exigidos para a função gemmini_preload():

- *rs1*[31:0]: Endereço do Scratchpad da matriz D (saída estacionária), ou matriz B (peso estacionário);
- *rs1*[47:32]: Número de colunas da matriz D / B;
- *rs1*[63:48]: Número de linhas da matriz D / B;
- *rs2*[31:0]: Endereço do Scratchpad da matriz C;
- *rs2*[47:32]: Número de colunas da matriz C;

- *rs2*[63:48]: Número de linhas da matriz C;
- *funct7*: 6

Abaixo consta os padrões exigidos para as funções gemmini_extended_compute_preloaded() e gemmini_extended_compute_accumulated():

- *rs1*[31:0]: Endereço do Scratchpad da matriz A;
- *rs1*[47:32]: Número de colunas da matriz A;
- *rs1*[63:48]: Número de linhas da matriz A;
- *rs2*[31:0]: Endereço do Scratchpad da matriz B (saída estacionária), ou a matriz D (peso estacionário);
- *rs2*[47:32]: Número de colunas da matriz B / D;
- *rs2*[63:48]: Número de linhas da matriz B / D;
- *funct7*: 4 (preloaded) ou 5 (accumulated)

A fim de facilitar a escrita do código, a API também fornece versões comprimidas das funções. Essas versões reduzem a necessidade de informar todos os campos, por realizar a chamada das funções estendidas implicitamente, completando os campos comumente menos utilizados.

```
1  // Copiar matrizes de entrada no Scratchpad
2  gemmini_mvin(A, A_sp_end);
3  gemmini_mvin(B, B_sp_end);
4  gemmini_mvin(D, D_sp_end);
5
6  // Configurar o Pipeline
7  gemmini_config_ex(OS, 0, 0, 0, 0);
8
9  // Ativar o SA
10 gemmini_preload(D_sp_end, C_sp_end);
11 gemmini_compute_accumulated(A_sp_end, B_sp_end);
12
13 // Copiar resultado na Cache
14 gemmini_mvout(C, C_sp_end);
```

Figura 3.14: Código de multiplicação de matriz sendo executada pelo acelerador Gemmini

Na Figura 3.14 é apresentado um exemplo, onde é executada a operação de multiplicação de matriz com o formato C = A * B + D, análogo ao apresentado na Seção 2.1, utilizando as funções de ativação disponibilizadas pela biblioteca nativa do Gemmini. Inicialmente é utilizado a função gemmini_mvin() para carregar as matrizes de entrada no Scratchpad, em seguida é configurado o *pipeline* de execução, para isso é utilizado a função

gemmini_config_ex(). Para o exemplo, os dois primeiros zeros representam a não ativação das funções ReLU e ReLU6, e os dois últimos zeros a não ativação da etapa de transposição de ambas as matrizes de entrada. Para realizar o início a computação das matrizes são utilizadas as funções gemmini_preload() e gemmini_compute_accumulated(), por fim, é realizado a cópia do resultado no Sistema de Memória, com a função gemmini_mvout().

Capítulo 4

Trabalhos relacionados

Com a difusão de aplicações que podem ser convertidas para padrões GEMM e GEMV, criou-se uma demanda por arquiteturas vetoriais e sistólicas pelo seu alto nível de paralelismo. No entanto, as arquiteturas vetoriais são bem mais difundidas que as sistólicas, pois atualmente, a maioria das CPUs comerciais apresentam suporte a instruções SIMD de comprimento fixo. Neste capítulo serão mostradas um pouco das histórias e, também, o estado da arte dessas arquiteturas, assim como trabalhos que visam otimizar limitações na sua execução.

4.1 Arquiteturas vetoriais

As primeiras arquiteturas vetoriais, que apresentavam o conceito de elementos de processamento em paralelo, começaram a surgir na década de 60. Uma das características dos primeiros projetos, era a comunicação entre os PEs através do barramento de memória, como o projeto Illiac IV [Hord, 2013] que apresentava 64 unidades de processamento. Mais adiante, projetos como Cray-1 [Russell, 1978], modificou a ativação das unidades, apresentando o conceito de registradores vetoriais, onde os dados temporários ficariam salvos nos registradores, beneficiando o reaproveitamento de dados. Ao longo dos anos foi observado que dobrar o comprimento dos registradores vetoriais seria mais energeticamente eficiente do que dobrar a quantidade de núcleos do SoC [Hennessy and Patterson, 2017].

A implementação de processamento vetorial mais difundida é baseada em instruções SIMD que estende a ISA do núcleo. Em 1996, a Intel lançou o Pentium MMX, o primeiro processador a suportar instruções SIMD, por meio da extensão MMX, que oferecia registradores de 64-bits, permitindo processar até quatro operações de 16-bits, ou até oito operações de 8-bits [Peleg et al., 1997]. Apesar de proporcionar ganhos de desempenho, a extensão apenas oferecia suporte a operações aritméticas com inteiros.

A geração seguinte da extensão integrada ao Pentium III, lançado em 1999, foi denominada de SSE (*Streaming SIMD Extensions*). A nova arquitetura proporcionava oito novos registradores de 128-bits, onde cada registrador podia armazenar até quatro operandos de 32-bit, com suporte a ponto flutuante de precisão simples, simultaneamente. Em 2000, a extensão SSE2 foi apresentada, oferecendo suporte a novas instruções para operação de precisão dupla, no entanto, foi mantido o comprimento dos registradores em 128-bits.

Inicialmente proposta em 2008 pela Intel, e comercialmente implementada em 2011 nos processadores da família *Sandy Bridge*, o conjunto de instruções AVX (do inglês *Advanced Vector Extensions*), estende a ISA da arquitetura x86 adicionando instruções de 256-bits, destinadas a principalmente operações matemáticas com 8 operandos de precisão simples ou 4 operandos de precisão dupla [Lomont, 2011].

Por fim, em 2016 foi apresentada a extensão AVX-512, adicionando suporte até 32 registradores de 512-bits, permitindo assim, 16 operandos de precisão simples ou 8 operandos de precisão dupla [Intel, 2022]. Historicamente, processadores produzidos pela AMD também ofereceram suporte a essas extensões. Vale destacar que, conforme apresentado na seção 2.3, bibliotecas como OpenBLAS [Xianyi and Kroeker, 2020], permite realizar processamento vetorial a partir da utilização de instruções AVX, quando compilada para atender arquiteturas x86.

O trabalho desenvolvido por Juan M. Cebrian [Cebrian et al., 2020] realiza uma análise de escalabilidade e eficiência energética da extensão AVX-512, comparando-as com a execução escalar, a partir de *benchmarks* do conjunto ParVec. Para o kernel Blackscholes, uma aplicação *CPU-bound*, a extensão AVX-512 conseguiu reduzir o número de instruções executadas em 19x e apresentar um speedup de 11x, permitindo uma eficiência energética de 9x melhor. Para kernel Streamcluster, uma aplicação *Memory-bound*, a extensão conseguiu reduzir o número de instruções executadas em 5x e apresentar um speedup de 2,4x, permitindo uma eficiência energética de 3x melhor.

4.2 Arquiteturas sistólicas

Após ser apresentada no final dos anos 70 [H.T.Kung, 1979], as arquiteturas sistólicas começaram a se destacar. Os primeiros projetos visavam solucionar sistemas lineares triangulares [H.T.Kung, 1982; Gentleman and Kung, 1982], calcular convoluções [Kung and Song, 1981], fatoração de matriz [Schreiber, 1983], entre outros algoritmos de Álgebra Linear. Com crescente demanda por aplicações envolvendo aprendizado de máquina, as arquiteturas sistólicas recuperaram recentemente popularidade, uma vez que os kernels são altamente suscetíveis à aceleração usando matrizes sistólicas.

Algumas arquiteturas sistólicas foram desenvolvidas com propósito comercial, onde pode-se destacar a Unidade de Processamento Tensor (em inglês *Tensor Processing Unit* (TPU)) desenvolvida pela Google [Jouppi et al., 2017]. Essa arquitetura apresenta uma matriz sistólica de multiplicadores contendo 256x256 unidades que executam operações MACs com operandos inteiros de 8-bits, combinado com uma unidade de acumuladores. A capacidade de executar até 64k de operações por ciclo tornou seu uso extremamente requisitado em cluster para aplicações de processamento em nuvem.

Também com objetivos de atender a *Data Center*, a Nvidia lançou em 2017 a arquitetura Pascal [Nvidia, 2017], para acelerar o processamento gráfico. A arquitetura apresenta até 672 unidades de matrizes sistólicas com capacidade de acelerar matrizes de dimensão 4x4 cada. A natureza do processamento gráfico permite que cada unidade de SA seja executada em paralelo. Sua arquitetura sucessora, denominada de Turing [Nvidia, 2018], expandiu as operações dos PEs, permitindo que fossem executadas operações com inteiros

de 32-bits, ao contrário da geração anterior, a densidade da malha de PEs, também foi aumentada para 8x8. Em 2020 foi apresentada a arquitetura denominada de Ampere [Nvidia, 2020], que elevou o número de PEs para uma malha de 16x16 e acrescentou unidades de pontos flutuantes de 64-bits.

4.2.1 Projetos acadêmicos

Reduzir a quantidade de acessos ao sistema de memória, reutilizando os dados pré-carregados no SA é altamente desejável, visto que a DRAM apresenta grande latência, resultando em processamento ocioso, quando é necessário recarregar os PEs. Por esse fim, o projeto de devolvido por Yu-Hsin Chen *et al*, apresenta o acelerador Eyeriss, que implementa um novo fluxo de dados para arquiteturas sistólicas, denominado de Linha Estacionária, com acrônimo em inglês de RS, que busca combinar os fluxos **WS** e **OS**, para equilibrar os benefícios de ambos, visando reduzir consumo energético da aplicações [Chen et al., 2016].

O fluxo apresentado consiste em carregar uma linha da matriz de pesos nos PEs e mantê-los estacionários enquanto a linha da outra matriz é propagada pelo SA, propagando os resultados parciais, porém, para aumentar a reutilização de dados, os resultados da propagação são carregados em acumuladores e mapeados os casos onde os pesos armazenados possam ser reaproveitados por outras etapas da convolução, acumulando os resultados de cada etapa. Com essa maior reutilização de dados, o projeto alcançou uma eficiência energética de até 2,5 vezes melhor, quando comparado com os fluxos **WS** e **OS**.

Alguns projetos visam aumentar a utilização dos PEs em casos onde são realizadas operações MAC sobre matrizes ou vetores esparsas (i.e. estruturas de dados com excesso de elementos nulos). O Eyeriss v2 [Chen et al., 2019] melhorou o Eyeriss original, apresentando uma abordagem otimizada para essas estruturas, que permite mapear e distribuir os pesos em qualquer PEs existentes no SA.

A fim de otimizar o fluxo RS, antes de ser carregado os pesos no PEs, os dados da matriz são analisados, identificando os elementos não nulos, a fim de gerar o vetor análogo à linha da arquitetura original. Para garantir a corretude dos resultados, cada PE é associado a um buffer para salvar o resultado no endereço correto. Quando comparado com o projeto original, a nova versão apresenta um Speedup de 5,3x com uma redução de energia de 3,9x.

O acelerador SIGMA [Qin et al., 2020] modifica a controladora e a interconexão, e realiza uma etapa de *bitmap* sobre ambas as matrizes de entrada, para primeiramente descartar da matriz propagada, as linhas inteiras nulas, e identificar todos os elementos nulos da matriz estacionária, em seguida os elementos não nulos são identificados. Dessa forma, quando carregados no PE, seus resultados são redirecionados para acumuladores que tenham o mesmo *ID*, permitindo assim, que o dado possa ser carregado em qualquer PE do SA implementado.

Essa abordagem permitiu que operações de GEMM com matrizes compostas por dados 80% espaços, atingissem utilização média de até 82% dos PEs, contra uma utilização média de 12,5% de uma abordagem tradicional, no entanto, devido a maior complexidade na controladora e na interconexão, o projeto apresenta uma sobrecarga de área de 37,7%.

O trabalho apresentado por Zhi-Gang Liu [Liu et al., 2020], fornece uma nova arquitetura de SA, denominado de *Systolic Tensor Array*, dedicada a um caso específico de matrizes esparsas, denominado de bloco delimitador de densidade (DBB), onde as matrizes apresentam um limite de elementos não nulos por coluna.

O objetivo da abordagem é reduzir a necessidade de implementação física das unidades PEs, uma vez que apenas uma fração delas serão ativadas na multiplicação. Por exemplo, se em uma coluna com 8 elementos, tiver apenas 2 elementos não nulos, a unidade de SA necessita apenas 2 unidades PEs em vez de 8, representando uma redução de 75% no hardware MAC. Uma limitação do projeto se deve ao fato de que após a sintetização da arquitetura, ela necessita de etapa de *tile*, externo ao acelerador, para executar multiplicação com matrizes que apresentam um limite de elementos não nulos por coluna maior que o projetado.

Como uma forma de facilitar o projeto de arquiteturas sistólicas, os pesquisadores também propuseram sistemas metodológicos e algoritmos para gerar automaticamente arquiteturas sistólicas diretamente dos algoritmos que deveriam acelerar. Como, por exemplo, a ferramenta PolySA [Cong and Wang, 2018], que analisa modelos poliédricos para tentar encontrar o mapeamento ótimo entre um algoritmo sequencial e um conjunto de PEs paralelos.

4.3 Síntese dos Trabalhos Relacionados

No decorrer deste capítulo, foram apresentados alguns projetos desenvolvidos que trabalham com arquiteturas relacionadas ao presente projeto. Para facilitar esta análise, a Tabela 4.1 foi criada, destacando alguns fatores importantes para análise e comparação entre os trabalhos:

- Tipo de arquitetura utilizada (Vetorial ou Matricial);
- Se a abordagem visada em seu desenvolvimento é comercial ou acadêmica;
- Se a unidade utilizada como base para a comparação de desempenho é o núcleo ou outro acelerador;
- Se a ferramenta desenvolvida permite a ativação automática sem a necessidade de modificação no código base por parte do programador.

Ao realizar a síntese dos trabalhos relacionados, é possível observar que o trabalho desenvolvido se diferencia dos demais por realizar a comparação de desempenho considerando os dois tipos de estruturas de processamento paralelo de dados estudados, e também, por permitir que a ativação dessas estruturas sejam realizadas sem a necessidade de modificação do código base da aplicação, facilitando assim, a implementação tanto do coprocessador vetorial Hwacha, quanto do acelerador matricial Gemmini.

Tabela 4.1: Tabela de comparação entre trabalhos

	Arquitetura		**Abordagem**			
Trabalhos	**V**	**M**	**A**	**C**	**Comparação**	**Ativação automática**
[Peleg et al., 1997]	X			X	Núcleo	
[Lomont, 2011]	X			X	Núcleo	
[Intel, 2022]	X			X	Núcleo	
[Cebrian et al., 2020]	X		X		Núcleo	
[Jouppi et al., 2017]		X		X	Núcleo	
[Nvidia, 2017, 2018, 2020]		X		X	Núcleo	
[Chen et al., 2016]		X	X		Acelerador	
[Chen et al., 2019]		X	X		Acelerador	
[Qin et al., 2020]		X	X		Acelerador	
[Liu et al., 2020]		X	X		Acelerador	
[Cong and Wang, 2018]		X	X		Núcleo	
Trabalho desenvolvido	X	X	X		Núcleo e Acelerador	X

A: *Projeto acadêmico*, **C**: *Implementação comercial*, **V**: Vetorial, **M**: Matricial

Capítulo 5

Integração no fluxo SMR

Códigos paralelos são difíceis de serem gerados automaticamente pelo compilador, uma vez que, para identificar regiões com potencial de paralelismo, como, por exemplo, transformar um loop em uma instrução vetorial, é necessário realizar um mapeamento das instruções do loop, a fim de identificar possíveis dependências de dados entre as iterações[Li et al., 2005]. Como consequência, a paralelização automática de código escrito de forma ingênua é geralmente limitada pela largura de banda, conforme apresentado na seção 2.3. Assim, nesses casos, são necessárias modificações no código base da aplicação, para permitir que o compilador alcance novas possibilidades de otimizações.

O objetivo deste capítulo é apresentar o *Source Matching and Rewriting* (SMR) [Espindola et al., 2023], uma ferramenta de código aberto orientada ao usuário, que realiza a reescrita de código, a partir da correspondência em código MLIR [Lattner et al., 2020, 2021]. Entretanto, o SMR não inclui integração com as plataformas estudadas nesse projeto. Portanto, a seguir também serão apresentadas as bibliotecas criadas que mapeiam as chamadas de GEMV e GEMM nos padrões da biblioteca OpenBLAS [Xianyi and Kroeker, 2020], para códigos de ativação do coprocessador Hwacha e do acelerador Gemmini.

5.1 Matching and Rewriting (SMR)

O SMR busca permitir que um usuário sem grande experiência com o fluxo de um compilador, consiga mapear e substituir trechos de códigos, permitindo otimizar a execução de algoritmos. Em [Espindola et al., 2023], foi demonstrada a ferramenta rescrevendo trechos dos kernels do programa Polybench [Narayan and Pouchet, 2012], mapeando para chamadas da biblioteca OpenBLAS correspondente, acelerando a execução de aplicações de álgebra linear, em troca de um pequeno aumento no tempo de compilação.

A ferramenta opera a partir de uma pequena linguagem declarativa, denominada de PAT, onde são descritos explicitamente a função de correspondência e a função de reescrita. Como pode ser visto na Figura 5.1, uma descrição de PAT é dividida em duas seções. A primeira seção (linhas 1 a 5), apresenta a declaração da função de correspondência, combinado com seus argumentos. A segunda seção (linhas 5 a 9), apresenta a função de reescrita que substituirá o trecho de código, caso haja uma correspondência. Em ambas seções é declarado a linguagem de programação utilizadas (lingA e lingB). O SMR não

avalia a corretude da função de reescrita, dessa forma, cabe ao programador garantir a correta execução das funções.

```
1 lingA {
2     fun (tipo1 arg1, tipo2 arg2, ...) {
3         correspondenciaA
4     }
5 } = lingB {
6     fun (tipo1 arg1, tipo2 arg2, ...) {
7         reescritaB
8     }
9 }
```

Figura 5.1: Visão geral da estrutura de um código PAT

Na Figura 5.2 é apresentado o fluxo de compilação do SMR. Para sua inicialização, dois arquivos de entrada são fornecidos, o código inicial e o arquivo PAT. Os arquivos de entrada são reduzidas aos seus dialetos MLIR correspondentes (por exemplo, FIR em caso de Fortran [Flang Compiler, 2019] ou CIL em caso C [Compiler Tree Technologies, 2021]), em seguida, são gerados gráficos de dependência de controle (CDG) do código de entradas e da função de correspondências, e essa etapa busca identificar estruturas que tenham fluxos de controle semelhantes ao desejado.

Na fase seguinte, os fragmentos de códigos candidatos da etapa anterior têm seus Gráficos de Dependência de Dados (DDGs) construídos. Essa nova etapa busca verificar se os tipos de dados utilizados pelos candidatos são isomórficos com o DDG gerado a partir do arquivo PAT. Isso garante que os nomes dados das variáveis não interferem nas comparações.

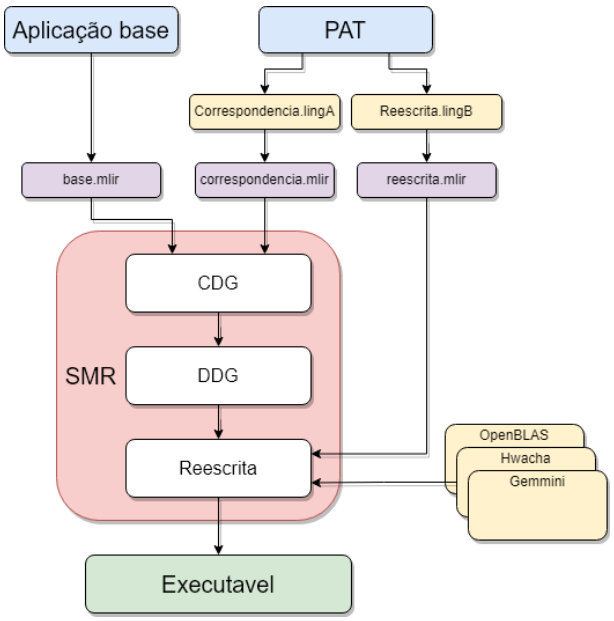

Figura 5.2: Fluxo SMR extendido com as bibliotecas desenvolvidas

O resultado de uma correspondência retorna uma bijeção, onde após a etapa de comparação, o SMR remove do código MLIR dos trechos marcados e substituiu pelo código MLIR da função de reescrita correspondente. A última etapa do fluxo do SMR consiste em gerar o arquivo executável otimizado a partir do código modificado.

Uma das principais vantagens do uso do SMR é que as funções do arquivo PAT e o arquivo de entrada são reduzidos para a linguagem intermediária MLIR. Isso permite que a função de reescrita possa ser desenvolvida em uma linguagem de programação diferente da utilizada no código inicial. Dessa forma, o seu uso torna-se útil no trabalho desenvolvido, por permitir que kernels que realizam multiplicação do tipo Matriz-Matriz e Matriz-Vetor, desenvolvidos em diferentes linguagens, sejam otimizados, reescrevendo a multiplicação por funções de ativação das plataformas Hwacha e Gemmini. Além disso, o SMR também ajuda a gerar um ecossistema padrão para a execução das arquiteturas estudadas,

Para permitir que a ferramenta SMR consiga acionar as plataformas externas é necessário que a função de reescrita existente no arquivo PAT execute os comandos de ativação das arquiteturas. Para facilitar a integração, foram desenvolvidas bibliotecas que sintetizam as etapas de movimentação de dados, configuração e ativação, em chamadas sintaticamente idênticas às operações de SGEMM e SGEMV existentes na biblioteca OpenBLAS. Dessa forma, na etapa de geração do executável, apenas o link utilizado pelo SMR para chamadas de bibliotecas externas é substituído, redirecionando para as novas bibliotecas desenvolvidas.

5.2 Integração do Hwacha no SMR

O projeto Hwacha conta com uma biblioteca, desenvolvida em C, para realizar processo de *tiling* entre multiplicações de matrizes [UC Berkeley Architecture Research, 2022]. Para executar uma multiplicação do formato C = A x B + C, a biblioteca busca abastecer o Hwacha, carregando uma linha da matriz **C** no registrador vetorial de destino, uma linha da matriz **B** no registrador vetorial de origem e apenas um dado da matriz **A** no registrador de vetor compartilhado por vez. Dessa forma, a linha da matriz **B** é multiplicada paralelamente com o dado da matriz **A** e acumulada na linha da matriz **C**. Nesse caso, os dados da matriz A, operam analogamente aos pesos de cada PE utilizados em um SA. Por fim, os endereços são reajustados para atender as próximas linhas.

Para o caso onde a largura das matrizes **C** e **B** são maiores que o comprimento do registrador vetorial, uma etapa adicional é implementada. Nesse caso, é realizado o processamento apenas das posições cobertas pelo registrador vetorial. Após a execução da multiplicação, o indicador da coluna é reajustado. O processo de multiplicação pode ser visto na Figura 5.3.

A biblioteca, no entanto, não oferece suporte as execuções da multiplicação com matriz transposta ou a multiplicação das matrizes por um escalar, dessa forma, a fim de acelerar operações de SGEMM e SGEMV existentes na biblioteca OpenBLAS, foi desenvolvido uma biblioteca que realiza a multiplicação por um escalar e mapeia as orientações de uma

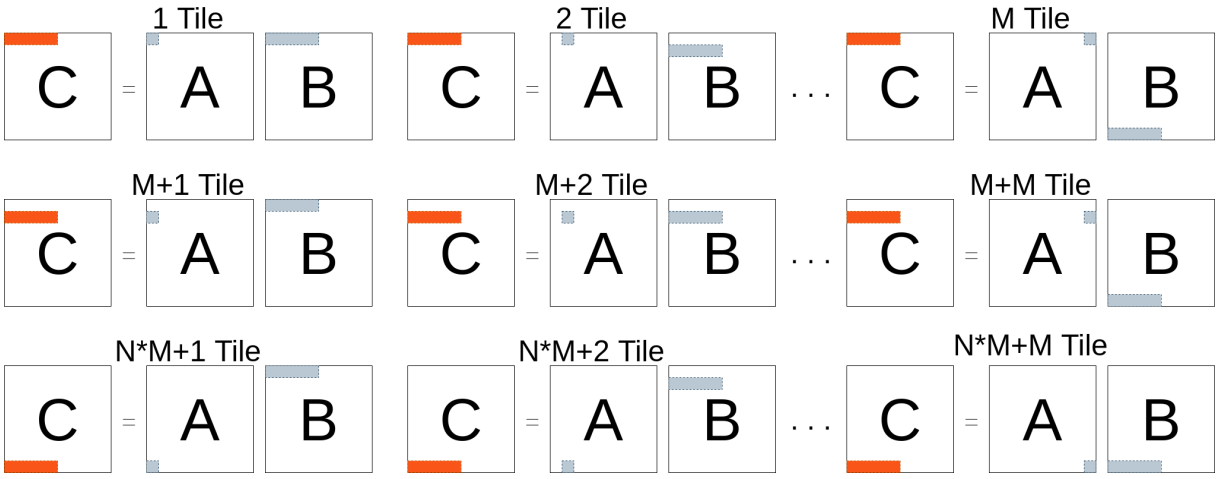

Figura 5.3: Diagrama de execução de multiplicação Matriz-Matriz paralelizada pelo coprocessador vetorial Hwacha. Orientação *Row-major*

chamada ao OpenBLAS e realiza a conversão das orientações e das dimensões para as chamadas do Hwacha.

Ocorre um gargalo importante nesse tipo de operação, em relação ao método de armazenamento dos dados. Enquanto o SMR foi desenvolvido visando aplicações do Polybench, baseados na linguagem Fortran, que assume a orientação *Column-major*, a biblioteca básica oferecida pelo Hwacha utiliza orientação *Row-major*, proveniente da linguagem C. Dessa forma, uma característica da biblioteca desenvolvida é que os comandos de transposição ocorrem de forma inversa ao requisitado.

Para oferecer suporte a multiplicação entre matrizes, são realizados três fluxos de execução. O primeiro fluxo é referente ao caso onde ambas as matrizes de entradas devem ser computadas na orientação padrão, nesse caso, as ordens das matrizes **A** e **B** são invertidas, com base na propriedade de multiplicação entre matrizes, onde a transposta de **A**x**B** e equivalente a **Bt**x**At**. O segundo fluxo é referente ao caso onde ambas as matrizes de entradas devem ser computadas na orientação transposta, nesse caso, as ordens das matrizes **A** e **B** não são alteradas e uma etapa adicionar de transposição é realizada pelo núcleo com o resultado da multiplicação. Por fim, para os casos onde apenas uma matriz deve ser transposta, essa etapa é realizada pelo núcleo na matriz identificada, e a multipliicação é ralizada na ordem invertida.

Como o coprocessador não realiza multiplicação das matrizes por um escalar, esse processo dever ser realizado pelo núcleo, portanto, a fim de mascarar a latência do acesso à memória, a multiplicação é realizada apenas na linha em que é encaminhada para o registrador vetorial. Para reduzir o impacto da memória quando executado multiplicação com matrizes muito grandes, também foi implementado o procedimento de divisão das matrizes em blocos, análogo ao apresentado na seção 2.3.

Em relação à operação de SGEMV que apresenta fórmula de y = alpha*A*x + beta*y, sendo **A** uma matriz MxN, **x** um vetor de comprimento N e, **y** um vetor de comprimento M, para garantir a integridade do resultado, a coluna da matriz **A** precisa ser carregada para o registrador vetorial, conforme mostrado na Figura 5.4, dessa forma, o vetor **x** opera como a matriz de peso, sendo carregado apenas uma posição no registrador de

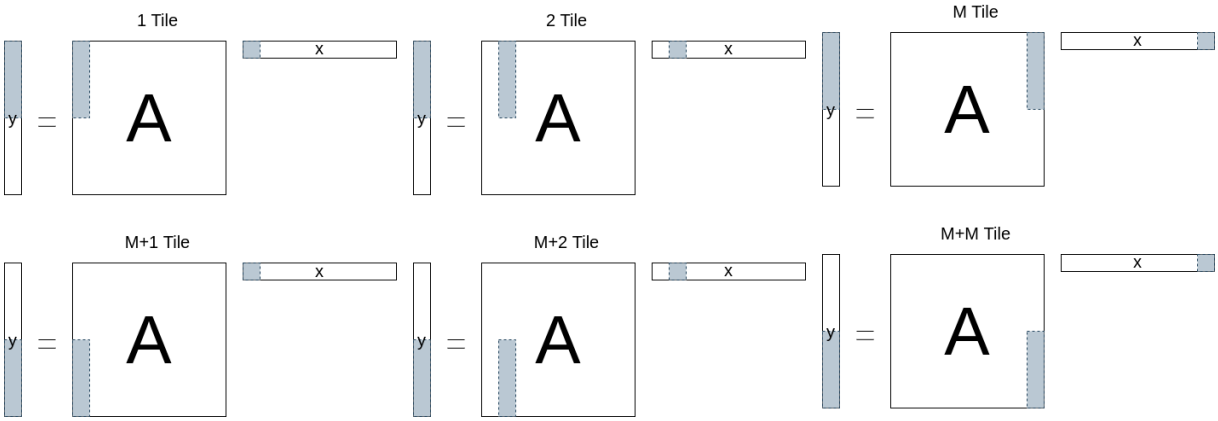

Figura 5.4: Diagrama de execução de multiplicação Matriz-Vetor paralelizada pelo coprocessador vetorial Hwacha. Orientação *Row-major*

vetor compartilhado, nesse momento, a etapa de multiplicação por um escalar ocorre, reduzindo assim, o gargalo que o núcleo gera na operação.

Quando requisitada uma operação com matriz transposta, uma etapa adicional é exigida. Para garantir que a leitura dos dados ocorram em endereços consecutivos, o núcleo é requisitado para acessar apenas os endereços da coluna referente a interação atual, salvando em um vetor auxiliar, que será indexado ao coprocessador para executar a multiplicação. A etapa adicional, representa uma adição de [NxM] novos acessos de leitura e [NxM] novos acessos de escrita ao sistema de memória, portanto, a integração da transposição na etapa da ativação do Hwacha, tem como objetivo, evitar múltiplos acessos a DRAM.

5.3 Integração do Gemmini no SMR

Como apresentado na Seção 3.5, a biblioteca nativa do Gemmini oferece suporte de operações de expressão C = alpha * A * B + beta * D. No entanto, uma limitação da sua integração no fluxo do SMR é causada pelo fato da biblioteca oferecer suporte a transposição apenas de uma das matrizes **A** e **B**, por chamada, não permitindo que a matriz **D** seja transposta. Portanto, para garantir a integralidade das operações, é necessário que o resultado da multiplicação entre as matrizes **A** e **B** esteja disposta de forma transposta em relação à orientação *Row-major*. Essa limitação é superada utilizado a propriedade transposição de multiplicação de matrizes, onde a transposta da multiplicação entre duas matrizes é igual à multiplicação da transposta de cada uma das matrizes.

Para acelerar operações de SGEMM, são realizados três fluxos de execução, onde no primeiro, as matrizes **A** e **B** originalmente não-transpostas realizam uma chamada ao Gemmini com as ordens das matrizes **A** e **B** invertidas, para garantir que o resultado seja transposto para somar com a matriz **D**, no segundo fluxo, com apenas uma das matrizes originalmente transpostas é realizada uma chamada ao Gemmini com as ordens das matrizes **A** e **B** invertidas e com a ativação do processo de transposição da matriz originalmente transposta e, por fim, o terceiro fluxo onde as matrizes **A** e **B** são originalmente transpostas realizam uma chamada ao Gemmini com as ordens das matrizes **A** e

B originais, como apresentado na seção anterior, nesse fluxo tambem é adicionado uma etapa de transposição, do resultado da multiplicação, realizado pelo núcleo.

Durante o desenvolvimento da função para acelerar operações de SGEMV, foi observado que o uso de vetores auxiliares que armazenam a multiplicação de *alpha* pelo vetor x e a multiplicação de *beta* pelo vetor y, aceleraram a operação, em comparação à multiplicação realizada no processo de movimentação dos dados pelo Gemmini, devido ao fato do processador BOOM conseguir despachar multiplas instruções de ponto flutuante por ciclo. Portanto, foi implementado um único fluxo, onde são utilizadas matrizes auxiliares com a multiplicação dos vetores pelos escalares e com a ativação da função de transposição inversa à chamada original do OpenBLAS.

Para reduzir a exigência no barramento de memória, em caso de *beta* igual a 0, apenas é repassado ao Gemmini a indicação que a matriz **D** é nula, eliminando assim a necessidade de acesso à memória, nesse caso o Gemmini mapeia os pesos do SA para o valor 0.

Capítulo 6

Evolução e Avaliação das arquiteturas

O objetivo deste capítulo é descrever a metodologia de avaliação do coprocessador Hwacha e do acelerador Gemmini. Primeiro são apresentadas as características do SoC utilizadas como base para a evolução das arquiteturas, em seguida são descritas as configurações de *hardware* específicas. Por fim, é discutido o desempenho obtido a partir do *benchmark* Polybench, comparando com a execução da biblioteca OpenBLAS.

6.1 Definição e configuração da plataforma de avaliação

Os experimentos foram realizados através do simulador Verilator [Verilator, 2022], um gerador de simulador *full-cycle* que utiliza o código Verilog, para emular o comportamento da arquitetura. Sua natureza de código aberto e seu alto desempenho, por permitir que a simulação ocorra de forma paralelizada, tornou seu uso amplamente difundido. O Verilator é integrado ao Chipyard, permitindo assim, que a simulação ocorra sem a necessidade de sintetizar o projeto em uma FPGA (em inglês *Field Programmable Gate Array*).

Para gerar o simulador, foi utilizado como base o processador BOOM em sua configuração *"MEGA"*, também denominada de 4-*wide* (4-*wide* significa que no estágio de *Fetch* são lidas 4 instruções por ciclo), instanciada pela classe *WithNMegaBooms*, dentro do ecossistema Chipyard. O núcleo é combinado com 32 Kb de Cache L1 8-way, e o SoC apresenta 512 Kb de uma Cache L2 8-way, baseado no módulo Sifive Inclusive-Cache[Specification, 2018], ambas memórias configuradas com blocos de 64 bytes. Para simular uma DRAM, dentro do ambiente gerado pelo Verilator, foi instanciado a classe *BlackBoxSimMem*, com capacidade de 2 GB, que modela uma DRAM com padrão de acesso de latência fixa.

Uma DRAM é uma memória desacoplada do SoC, composta por um arranjo de células dispostas em formato de matriz. Quando é realizado um acesso, a controladora conecta a linha de destino ao barramento de transferência, sendo assim, quando for realizado acessos a endereços contínuos, a linha estará ativa permitindo que os demais dados sejam carregados diretamente.

Dessa forma, para aproximar-se do comportamento real de uma DRAM, o valor da frequência do núcleo precisou ser fixada em 1GHz, valor extraído a partir de diversos projetos que realizaram a sintetização do processador BOOM em FPGAs [Genc et al.,

2021; Zhao et al., 2020; Gonzalez et al., 2021; Kim et al., 2017]. Por fim, para os parâmetros da DRAM, foi utilizado como base os valores extraídos da memória da MICRON DDR3 SG15 que opera a 1333 MHz, e podem ser vistos abaixo:

- Taxa de transferência: 1333 MT/s;
- Tamanho do bloco de memória: 8 bytes;
- Tempo necessário para transferir um bloco: 750 ps;
- Tempo mínimo necessário para que uma linha fique ativa para garantir que os dados possam ser acessados: 51000 ps.

Logo, o tempo mínimo necessário para acessar e transferir 64 Byte (tamanho de um bloco da Cache utilizada no projeto) é de 56250 ps, que representa um atraso de 56,25 ciclos quando observado pelo núcleo operando a 1GHz, sendo assim, para todos os experimentos foi aplicado um atraso fixo de 60 ciclos.

6.2 Aplicações

As aplicações escolhidas neste trabalho pertencem ao *benchmark* Polybench[Narayan and Pouchet, 2012], disponível em: `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/polybench-fortran.html`. As sete aplicações escolhidas pertencem ao conjunto de Álgebra Linear, e foram definidas para variáveis de ponto flutuante de precisão simples. As aplicações podem ser visualizadas abaixo:

- **gemm:** Realiza o procedimento de multiplicação entre matrizes, com ativação da etapa de multiplicação por escalares. Operação de formato $C = alpha * A * B + beta * C$;
- **2mm:** Realiza duas operações de multiplicação entre matrizes, onde os dados não são reaproveitados, sem a ativação da etapa de multiplicação por escalares;
- **3mm:** Realiza três operações de multiplicação entre matrizes, onde os resultados das duas primeiras multiplicações são as entradas da terceira multiplicação, sem a ativação da etapa de multiplicação por escalares;
- **syrk:** Realiza uma operação de multiplicação entre matrizes, com formato $C = alpha * A * A^t + beta * C$;
- **atax:** Realiza duas operações de multiplicação entre um vetor e uma matriz, primeiramente, na orientação normal seguida pela orientação transposta, onde o vetor resultante da primeira operação é o vetor de entrada da segunda operação;
- **bicg:** Realiza duas operações de multiplicação entre um vetor e uma matriz, primeiramente, na orientação normal seguida pela orientação transposta, onde as operações são independentes;

- **mvt:** Realiza duas operações de multiplicação Matriz-Vetor, primeiramente, na orientação normal seguida pela orientação transposta, com ativação do escalar *beta*, onde as operações são independentes.

Para simular uma operação de multiplicação entre matrizes de ponto flutuante de precisão simples, análogo a operação SGEMM, foi utilizado o kernel *gemm* do programa Polybench, que realiza uma única operação de multiplicação. As variáveis escalares *alpha* e *beta* também foram ativadas, dessa forma, é analisado o cenário onde são exigidos todos os recursos da aplicação. A fim de simular uma operação de SGEMV, foi desenvolvido um novo kernel, que pode ser visualizado na Figura 6.1, baseado no kernel *mvt*, visto que o *mvt* realiza dois procedimentos de multiplicação matriz-vetor independentes, em sequência.

O executável de cada aplicação foi gerado através da ferramenta SMR. Conforme apresentado na seção 5.1, para a execução do fluxo, é necessário descrever explicitamente a função de correspondência e a função de reescrita, dessa forma, foi desenvolvida a descrição que mapeia os códigos de multiplicações Matriz-Matriz ingênua e Matriz-Vetor ingênua, para chamadas conforme a sintaxe exigida pelas operações de SGEMM e SGEMV. Em todos os casos, no executável é realizado o procedimento de limpeza das caches antes de iniciar a etapa de computação, permitindo assim avaliar com a mesma condição a movimentação de dados por parte das arquiteturas.

```
1  subroutine kernel_GEMV(M, x, y, a, alpha, beta)
2      implicit none
3
4      DATA_TYPE, dimension(M, M) :: a
5      DATA_TYPE, dimension(M) :: x
6      DATA_TYPE, dimension(M) :: y
7      DATA_TYPE :: alpha, beta
8      integer :: M
9      integer :: i, j
10
11     do i = 1, M
12       y(i) = (y(i)*beta)
13       do j = 1, M
14         y(i) = y(i) + (alpha * a(j, i) * x(j))
15       end do
16     end do
17 end subroutine
```

Figura 6.1: Kernel para execução de operação GEMV. Linguagem Fortran

O cálculo de desempenho da arquitetura por aplicação é dado por instruções de ponto flutuante por segundo (FLOPS, do inglês *Floating Point operations per second*), permitindo assim, mensurar a vazão das bibliotecas desenvolvidas e compará-los com a execução da biblioteca OpenBLAS.

Primeiramente, é realizada a multiplicação da quantidade de ciclos gastos na simulação do Verilator (N_c), após o processo de limpeza do cache, pelo inverso da frequência de

operação (N_f), a partir disso, calcula-se o tempo de execução (T), conforme a Equação 6.1. Em seguida, o desempenho final, em FLOPS, é dado pela divisão entre o tempo de execução e a quantidade de operações de ponto flutuante executadas pela aplicação (N_o), como pode ser visto na Equação 6.2.

$$T = N_c \frac{1}{N_f} \tag{6.1}$$

$$FLOPS = \frac{T}{N_o} \tag{6.2}$$

6.3 Avaliação Hwacha

Para realizar o estudo de aceleração por parte do coprocessador Hwacha, primeiro foi avaliado a escalabilidade quando modificado o número de Pistas Vetoriais, em seguida foram avaliados gargalos no projeto da biblioteca de ativação, à medida que as dimensões dos dados de entrada crescem. A partir do modelo original apresentado em [Lee et al., 2015], foram detalhados, na Tabela 6.1, alguns dos principais parâmetros influenciam no seu desempenho.

Tabela 6.1: Parâmetro Hwacha

Parâmetro	**Especificação**
Frequência	1 GHz
Pistas Vetoriais	1/2/4
N° Bancos de execução por Pistas Vetoriais	4
Hwacha L1 VI	4 Kb
N° Registradores vetoriais	256
N° Registradores de endereço	32
Comprimento máximo do vetor	2048

Foram montados três cenários de execução para avaliar a escalabilidade da arquitetura do acelerador. Para cada cenário, com o auxílio da plataforma Verilator, foram gerados simuladores com diferentes quantidades de Pistas Vetoriais, variando em 1, 2 e 4, devido à imposição do gerador em aplicar valores em potência de 2.

Na Figura 6.2 é demonstrado o desempenho das arquiteturas propostas. É possível verificar que para a aplicação SGEMV, inicialmente não são observados ganhos de desempenho ao modificar a quantidade de Pistas Vetoriais, pois primeiramente os dados são carregados a partir da DRAM, dessa forma, latência da memória oculta o tempo de execução das multiplicações. Também deve ser considerado o fato que ao adicionar novas Pistas Vetoriais, o sequenciador mestre do Hwacha mantém os dados parados até que todas as unidades tenham terminado de computar os dados para salvá-los em ordem nos registradores vetoriais. No entanto, quando executado com matrizes [512x512] ou maiores, a biblioteca desenvolvida permite que sejam reaproveitados os dados existentes na cache, reduzindo a latência média, assim, elevando a vazão de dados.

Em relação ao SGEMM, devido à característica da aplicação de realizar múltiplos acessos ao mesmo endereço, a biblioteca desenvolvida permite explorar melhor o rea-

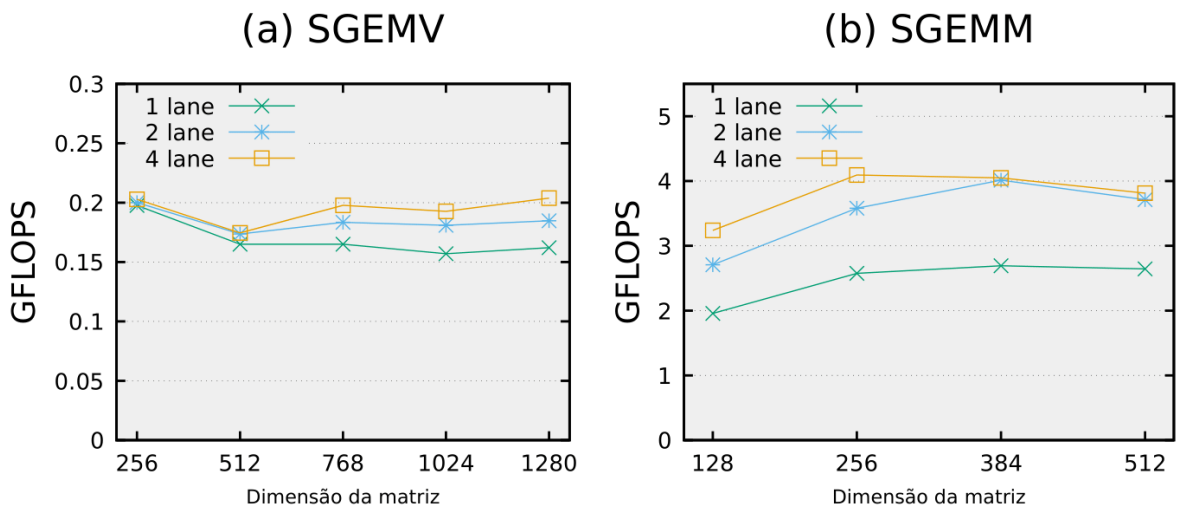

Figura 6.2: Número de milhões de instruções de ponto flutuante por segundo executadas pelo Hwacha nas aplicações SGEMV e SGEMM, variando a quantidade de Pistas Vetoriais

proveitamento de dados, dessa forma, a vazão observada é maior quando comparada ao SGEMV.

Para a execução com apenas uma Pista Vetorial, a vazão observada quando executado com matrizes [512x512], representa 66,1% do potencial teórico da arquitetura, uma vez que cada Pista Vetorial é composta por 4 unidades de computação, permitindo assim até 4 GFLOPS de computação ativa. O gráfico também demonstra que quando elevado o número de pistas para 2 e 4, o desempenho de ambas configurações convergem entre si, apresentando uma diferença de desempenho de 2,7%, reduzindo a vazão para 46,4% e 23,8%, respectivamente, demonstrando que ao elevar o número de pistas a aplicação começou a ser limitada pela memória, uma vez que para apenas duas pistas vetoriais, aproximou-se do limiar da biblioteca.

A unidade de memória do Hawcha (SMU), impõe uma limitação à implementação do kernel SGEMV, por apenas emitir requisições ao barramento de memória em endereços consecutivos. Dessa forma, para realizar operação com matriz transposta, à biblioteca requisita ao núcleo, a etapa de transposição dos dados. Para mensurar o impacto que os acessos adicionais na memória causam na aplicação SGEMV, foram executadas chamadas para a função com e sem orientação transposta, utilizando apenas uma Pista Vetorial.

Quando acionado a transposição pelo núcleo, são adicionados [2xNxM] novos acessos ao sistema de memória. Na Figura 6.3 é possível verificar que os acessos adicionais convergem para uma redução de 32,7% na vazão da aplicação. Analisando isoladamente a multiplicação com matriz de [1280x1280], a execução na orientação normal apresentou 10108079 ciclos simulados, enquanto, com multiplicação transposta na orientação apresentou 15029687, representando um aumento de 4921608 ciclos, isso posto, considerando que [2x1280x1280] novos acessos a DRAM representariam 12288000 ciclos de penalidades, a estratégia adotada na biblioteca, de apenas salvar os dados da orientação transposta em vetores para então serem carregados nos registradores vetoriais, permitiu reduzir a penalidade, uma vez que, a cache L2 utilizada não é capaz de alocar completamente a matriz utilizada.

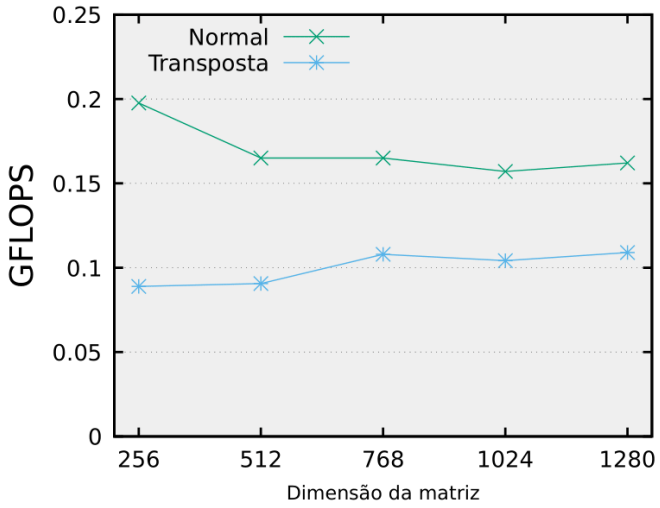

Figura 6.3: Número de milhões de instruções de ponto flutuante por segundo na aplicação SGEMV, utilizando orientação normal e transposta da matriz de entrada

6.4 Avaliação Gemmini

Para realizar o estudo do acelerador Gemmini, foi avaliado o impacto que os fluxos de dados apresentam nas aplicações SGEMM e SGEMV. Para a realização da avaliação foi utilizado como base os principais parâmetros da implementação sintetizada em [Genc et al., 2021], no entanto, a implementação original conta com um array sistólico de 16x16 de unidades de computação por inteiro de 8-bits, dessa forma, para realizar os experimentos foi modificada a unidade de processamento para suportar operações de ponto flutuante de 32-bits, através da classe *GemminiFPConfigs*. Os principais parâmetros podem ser vistos na Tabela 6.2.

Tabela 6.2: Especificação Gemmini

Parâmetro	**Especificação**
Frequência	1 GHz
Nº de *Tiles*	1
Tile	8x8 PEs
Precisão	FP 32-bits
Scratchpad	256 Kb
Acumulador	64 Kb

Como apresentado na seção 3.5.1, o fluxo de dados **WS** mantém a matriz **B** estacionada nos PEs correspondentes, visando minimizar os acessos ao sistema de memória, enquanto o fluxo de dados **OS**, é projetado para reaproveitar os endereço de destino, a fim de aumentar a acumulação dos dados. Dessa forma, devido à característica dos experimentos de realizar apenas uma etapa de propagação, o fluxo **WS** se mostrou melhor para realização das operações de SGEMV e SGEMM. Como demonstrado na Figura 6.4.

Também é possível constatar que, devido à presença do Scratchpad no Gemmini de 256 KB, a vazão observada na operação SGEMM quando aplicada matrizes quadradas de [128x128] se diferencia dos demais, isso se deve ao fato que o Scratchpad não é comple-

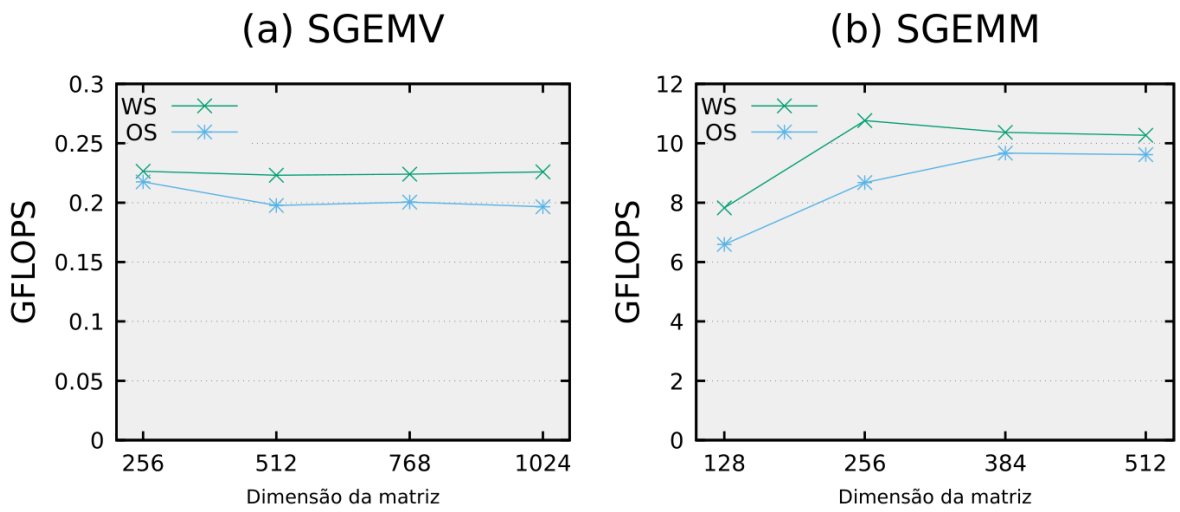

Figura 6.4: Número de milhões de instruções de ponto flutuante por segundo executadas pelo Gemmini nas aplicações SGEMV e SGEMM, variando o fluxo de dados

tamente explorado, pois é grande o suficiente para armazenar todos os dados de entrada e de saída, simultaneamente. No entanto, quando executado com matrizes [256x256] ou maiores, a biblioteca consegue alocar todas as posições do Scratchpad, permitindo assim aumentar a vazão de dados, por reaproveitar os dados armazenados no Scratchpad, que apresenta baixa latência de acesso, por ser implementado em SRAM.

6.5 OpenBLAS

Para gerar a biblioteca OpenBLAS, foi utilizado a versão 0.3.20, disponível em `https://github.com/xianyi/OpenBLAS/releases/download/v0.3.20/OpenBLAS-0.3.20.tar.gz`, configurada para a arquitetura *RISCV64_GENERIC*. Para comparação, foi utilizado o coprocessador Hwacha em sua configuração com apenas uma Pista Vetorial, e o acelerador Gemmini configurado para ser executado com o fluxo WS. As implementações dos kernels SGEMV e SGEMM, da biblioteca OpenBLAS, foi utilizada como linha de base para ambas arquiteturas.

Na Figura 6.5a é avaliado o desempenho da aplicação SGEMV quando executada a partir da biblioteca OpenBLAS e das bibliotecas desenvolvidas. O Hwacha apresentou um speedup que convergiu para 1,7x, enquanto o Gemmini apresentou um speedup de 2,1x, mesmo apresentando 64 PEs. Devido à sua estrutura rígida, o array sistólico do Gemmini é subutilizado, tendo apenas 8 PEs ativados durante a execução, pois apenas a primeira linha do SA é carregada com os dados do vetor, para garantir que os dados da matriz sejam acumulados nas posições corretas, uma vez que as somas parciais são repassadas pelas colunas durante a execução.

Para a aplicação SGEMM, todos os 64 PEs do Gemmini podem ser ativados, permitindo assim ganhos de desempenho próximos às 19x, quando comparado com a implementação do OpenBLAS, enquanto o Hwacha em sua configuração de uma Pista Vetorial apresentou ganhos próximos a 4,9x quando aplicadas matrizes de dimensão [768x768].

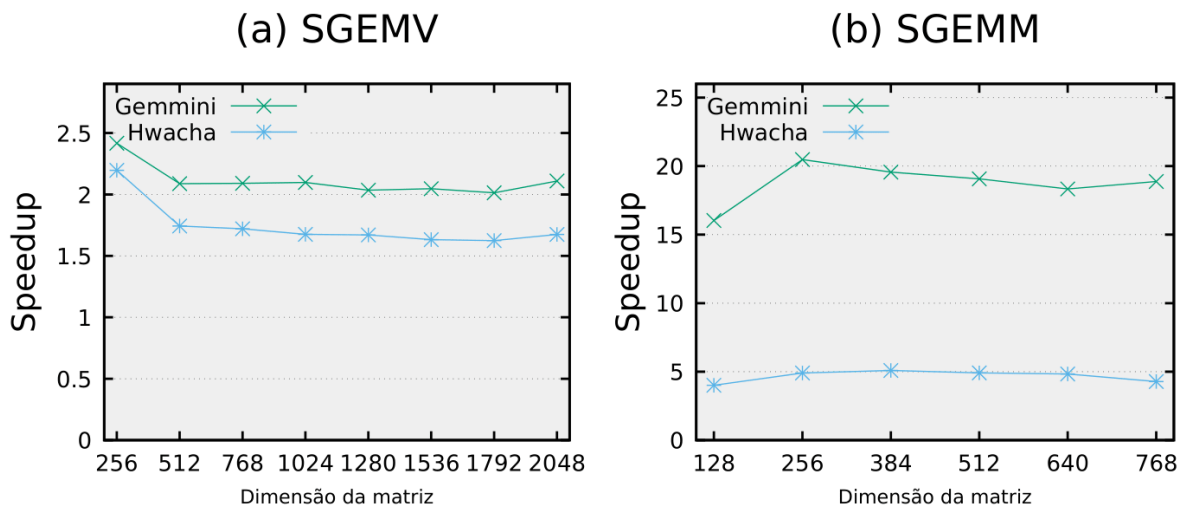

Figura 6.5: Desempenho da execução dos kernel SGEMM e SGEMV, quando executadas pelo Hwacha e Gemmini, em comparação a execução pelo núcleo

Como mostrado na Figura 6.2, o Hwacha não apresenta grandes ganhos ao aumentar o número de Pista Vetorial, dessa forma, o maior desempenho do Gemmini provém principalmente pelo uso do Scratchpad, que permite ser abastecida com dados da cache L2 enquanto executa as operações de MAC, e pela comunicação por registradores locais internos do SA, permitindo maior vazão de dados.

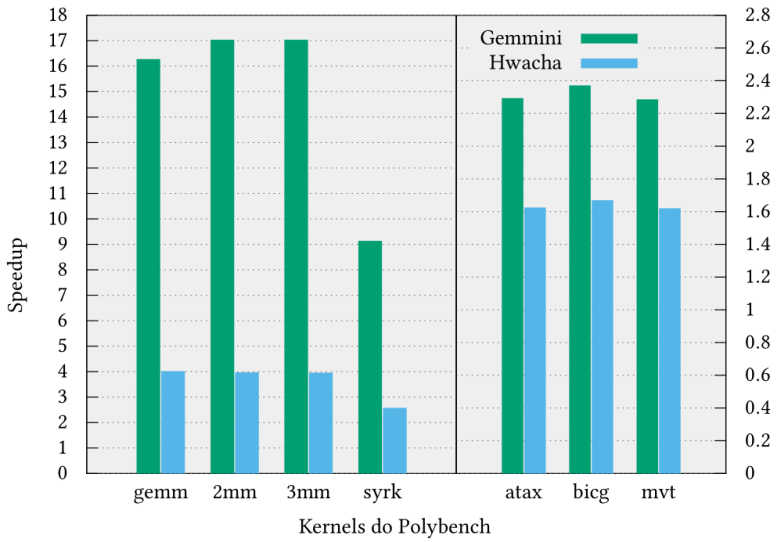

Figura 6.6: Desempenho dos kernels do Polybench, quando executadas pelo Hwacha e Gemmini, em comparação a execução pelo núcleo

Os resultados do Polybench, quando aplicado o conjunto de dados *"SMALL_DATASET"*, podem ser vistos na Figura 6.6. Nos kernels *gemm*, *2mm* e *3mm*, a reescrita aplicada pelo SMR envolve a substituição do kernel por uma ou mais execuções da chamada SGEMM, por conta disso, o Gemmini destaca-se com speedups que variam entre 16x e 17x mais rápidas que o OpenBLAS, enquanto o Hwacha apresenta ganhos próximos a 4x.

Para o kernel *syrk*, a ferramenta SMR aplica uma reescrita abaixo do ideal. Apesar de ter uma aplicação nativa na biblioteca OpenBLAS, o kernel é convertido para duas chamadas de SGEMM, reduzindo os ganhos para a marca de 9,1x com o Gemmini e de 2,6x para o Hwacha.

Com relação aos kernels *atax*, *bicg* e *mvt*, pequenos speedups foram alcançados, ficando na marca de 2,28-2,37x quando executado no Gemmini e, devido a sua limitação de transposição já apresentada, os ganhos apresentados pelo Hwacha atingiram a marca de 1,61-1,67x.

6.6 Limitações Experimentais

Algumas limitações foram observadas durante o processo de simulação das aplicações e da geração do simulador por parte do Verilator. Durante toda a etapa experimental, foi utilizado um computador equipado com um Core i9-9900KF de 16 *threads* operando à 3.6GHz, combinado com 32GB de memória RAM, e para complementar a memória foi separado 32GB do disco rígido para operar como memória de SWAP, totalizando 64GB de memória mapeável. O sistema operacional Linux 20.04 LTS, foi utilizado durante a experimentação.

Na etapa de geração do simulador, o total de memória do sistema foi um limitante observado. Quando configurada uma arquitetura composta pelo núcleo BOOM, em sua opção *4-wide*, e o acelerador Gemmini composto por um SA 8x8 de multiplicação de ponto flutuante de 32-bits, recebeu picos de memória de 40GB (32GB de RAM + 8GB de SWAP), no entanto, o Verilator não permitiu gerar a mesma configuração com um SA de 16x16, alegando falta de memória. A mesma mensagem foi observada quando foi configurada uma arquitetura composta pelo núcleo BOOM, em sua opção *4-wide*, e o coprocessador Hwacha com 8 pistas vetoriais, impedindo assim, análise com configurações superiores as utilizadas.

Outra limitação observada, consiste na consideração ao tempo de simulação. Para realizar a simulação de uma operação SGEMM com matrizes de [768x768] através da biblioteca OpenBLAS, foram necessárias aproximadamente 38 horas para executar completamente a operação, utilizando as 16 *threads* disponíveis do processador. Dessa forma, é possível prever que para simular a operação *gemm* do Polybench com conjunto de dados *"STANDARD_DATASET"*, levaria ao menos 90 horas para sua simulação e até o triplo desse valor para simular a operação *3mm*. Sendo assim, para simular todas as aplicações sendo executadas pela biblioteca OpenBLAS, com esse conjunto de dados, seriam necessários mais de 6 semanas de execução ininterrupta, uma vez que esse conjunto trabalha com matrizes de [1024x1024] para operações de tipo GEMM e matrizes de [4000x4000] para operações de tipo GEMV.

Um objetivo desejável que não pode ser realizado é a análise de área das arquiteturas estudadas, uma vez que o Verilator não fornecesse ferramentas para tal análise. Embora tenham sido encontrados trabalhos relacionados que tratam da análise de área das arquiteturas em questão, as métricas utilizadas nesses trabalhos não podem ser comparadas diretamente porque não foram sintetizadas com a mesma tecnologia. Dessa forma, a fim

de dimensionar as áreas do coprocessador Hwacha e do acelerador Gemmini, foi utilizado como referência de área o núcleo utilizado nos projetos, onde foi possível visualizar que o núcleo Rocket combinado com uma cache L1 de 16 Kb, foi um padrão nas implementações.

Os trabalhos indicaram que, para uma Pista Vetorial, o Hwacha apresenta uma área 80% maior do que o núcleo de referência, quando configurado com as mesmas especificações apresentadas na Tabela 6.1. Já para o acelerador Gemmini, não foram encontrados trabalhos com configurações semelhantes às apresentadas na Tabela 6.2, em relação ao Array sistólico. No entanto, utilizando o processador Rocket como referência, foi possível verificar que o acelerador configurado com um Array sistólico de 16x16 com suporte apenas a operações de inteiros de 8 Bits, apresentou uma área 4,7 vezes maior do que o processador.

Capítulo 7

Considerações Finais

A crescente demanda por aplicações dependentes de processamento sobre estruturas de dados vetoriais e matriciais, motivaram a indústria e o ambiente acadêmico a buscarem soluções mais eficientes para esse padrão de computação. Nesse contexto, estruturas de *hardware* especializadas em oferecer processamento paralelo sobre tais dados, surgiram para contornar soluções em *software*, limitadas pelas atuais demandas de área e energia do núcleo, no entanto, precisam de conhecimento prévio por parte do programador para sua completa utilização.

Esta proposta buscou o estudo e análise de arquiteturas RISC-V de processamento paralelo para aceleração de aplicações GEMM e GEMV, através da implementação do coprocessador vetorial Hwacha e do acelerador matricial Gemmini. Para isso, foram apresentados os seus fluxos de controle e de dados, com suas limitações estruturais.

Para a análise das plataformas foram criadas bibliotecas individuais que mapeiam os padrões de códigos de ativação e movimentação de dados, para chamadas de rotinas GEMM e GEMV em funções sintaticamente idênticas às chamadas de bibliotecas análogas ao BLAS.

A adoção da ferramenta de reescrita de código em nível de compilação SMR, disponibilizou um ambiente de compilação que elimina a necessidade de conhecimento das etapas de ativação de cada arquitetura estudada, por parte do usuário final, permitindo que o coprocessador vetorial Hwacha e o acelerador matricial Gemmini fossem compatíveis com aplicações desenvolvidas em Fortran sem a necessidade de reescrita das mesmas.

Em relação ao desempenho final, a implementação do coprocessador vetorial Hwacha permitiu observar ganhos de até 4 vezes em aplicações de SGEMM e ganhos de 1,6 vezes em aplicações exigentes de SGEMV, sobre a solução de *software* OpenBLAS. Quanto ao acelerador matricial Gemmini, foram observados ganhos na marca de 17 e 2,3 vezes para as mesmas aplicações. Os experimentos mostraram que quando o SA é subutilizado a sua eficiência geral é reduzida. No entanto, ficou evidente que a maior limitação de utilizar arquiteturas especializadas para a aceleração das rotinas SGEMV e SGEMM, se deve ao fato que seu uso impõe uma grande demanda ao sistema de memória, dessa forma, a aplicação é limitada pela latência da memória DRAM.

Como trabalho futuro, sugere-se implementar essas arquiteturas em FPGAs para aprofundar a análise no sistema de memória e aceleração da execução dos kernels, permitindo assim, a aplicação em conjunto de dados maiores.

Referências Bibliográficas

Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Jonathan Bachrach, Sophia Shao, Borivoje Nikolić, and Krste Asanović. Invited: Chipyard - an integrated soc research and implementation environment. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi: 10.1109/DAC18072.2020.9218756.

D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967. doi: 10.1147/rd.111.0008.

Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

Arthur Cayley. A memoir on the theory of matrices. *Philosophical transactions of the Royal society of London*, 148:17–37, 1858.

Juan Cebrian, Lasse Natvig, and Magnus Jahre. Scalability analysis of avx-512 extensions. *The Journal of Supercomputing*, 76, 03 2020. doi: 10.1007/s11227-019-02840-7.

Chris Celio, Jerry Zhao, Abraham Gonzalez, and Ben Korpan. Riscv-boom documentation. https://docs.boom-core.org/, Apr 2021.

Texas Advanced Computing Center. Gotoblas2, 2020. URL `https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2`.

Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft, 2006.

Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual*

International Symposium on Computer Architecture (ISCA), pages 367–379, 2016. doi: 10.1109/ISCA.2016.40.

Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

Compiler Tree Technologies. CIL, feb 2021. URL `https://github.com/compiler-tree-technologies/cil`.

Jason Cong and Jie Wang. Polysa: Polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018. doi: 10.1145/3240765.3240838.

Francesco Conti, Robert Schilling, Pasquale Davide Schiavone, Antonio Pullini, Davide Rossi, Frank Kağan Gürkaynak, Michael Muehlberghuber, Michael Gautschi, Igor Loi, Germain Haugou, et al. An iot endpoint system-on-chip for secure and energy-efficient near-sensor analytics. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(9):2481–2494, 2017.

Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987.

Dennard, Robert, Gaensslen andFritz, Yu, Hwa-Nien, Rideout, Leo, Bassous, Ernest, and Andre Leblanc. Design of ion-implanted mosfets with very small physical dimensions. *IEEE Journal of Solid-Circuits*, pages 256–267, 1974.

Dennard, Robert H, Cai, Jin, Kumar, and Arvind. A perspective on today's scaling challenges and possible future directions. *Solid-State Electronics*, 51(4):518–525, 2007.

Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *International conference on machine learning*, pages 647–655. PMLR, 2014.

Vinicius Espindola, Luciano Zago, Hervé Yviquel, and Guido Araujo. Source matching and rewriting for mlir using string-based automata. *ACM Trans. Archit. Code Optim.*, 20(2), mar 2023. ISSN 1544-3566. doi: 10.1145/3571283. URL `https://doi.org/10.1145/3571283`.

Eric Flamand, Davide Rossi, Francesco Conti, Igor Loi, Antonio Pullini, Florent Rotenberg, and Luca Benini. Gap-8: A risc-v soc for ai at the edge of the iot. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–4. IEEE, 2018.

Flang Compiler. F18 LLVM Project (fir-dev branch), nov 2019. URL `https://github.com/flang-compiler/f18-llvm-project/tree/fir-dev`.

Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 769–774, 2021. doi: 10.1109/DAC18074.2021.9586216.

W Morven Gentleman and HT Kung. Matrix triangularization by systolic arrays. In *Real-time signal processing IV*, volume 298, pages 19–26. SPIE, 1982.

Abraham Gonzalez, Jerry Zhao, Ben Korpan, Hasan Genc, Colin Schmidt, John Wright, Ayan Biswas, Alon Amid, Farhana Sheikh, Anton Sorokin, Sirisha Kale, Mani Yalamanchi, Ramya Yarlagadda, Mark Flannigan, Larry Abramowitz, Elad Alon, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. A 16mm2 106.1 gops/w heterogeneous risc-v multi-core multi-accelerator soc in low-power 22nm finfet. In *ESSCIRC 2021 - IEEE 47th European Solid State Circuits Conference (ESSCIRC)*, pages 259–262, 2021. doi: 10.1109/ESSCIRC53450.2021.9567768.

John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 6th edition, 2017.

R Michael Hord. *The Illiac IV: the first supercomputer*. Springer Science & Business Media, 2013.

Charles E. Leiserson H.T.Kung. Systolic arrays (for vlsi). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for industrial and applied mathematics Philadelphia, PA, USA, 1979.

Hsiang-Tsung H.T.Kung. Why systolic architectures? *Computer*, 15(01):37–46, 1982.

Intel. *Intel® Architecture Instruction Set Extensions and Future Features*. Intel, 2022.

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay

Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, jun 2017. ISSN 0163-5964. doi: 10.1145/3140659.3080246. URL `https://doi.org/10.1145/3140659.3080246`.

N. Kanopoulos, N. Vasanthavada, and R.L. Baker. Design of an image edge detection filter using the sobel operator. *IEEE Journal of Solid-State Circuits*, 23(2):358–367, 1988. doi: 10.1109/4.996.

Donggyu Kim, Christopher Celio, David Biancolin, Jonathan Bachrach, and Krste Asanović. Evaluation of risc-v rtl with fpga-accelerated simulation. In *The First Workshop on Computer Architecture Research with RISC-V (CARRV 2017)*, 2017.

HT Kung and Siang Wun Song. A systolic 2-d convolution chip. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1981.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law. In *CoRR*, 2020.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO 2021 - Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 2–14, 2021. ISBN 9781728186139. doi: 10.1109/CGO51591.2021.9370308.

Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, and Krste Asanović. The hwacha microarchitecture manual, version 3.8.1. Technical Report UCB/EECS-2015-263, EECS Department, University of California, Berkeley, Dec 2015. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-263.html`.

Man-Lap Li, R. Sasanka, S.V. Adve, Yen-Kuang Chen, and E. Debes. The alpbench benchmark suite for complex multimedia applications. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 34–45, 2005. doi: 10.1109/IISWC.2005.1525999.

Zhi-Gang Liu, Paul N. Whatmough, and Matthew Mattina. Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile cnn inference. *IEEE Computer Architecture Letters*, 19(1):34–37, 2020. doi: 10.1109/LCA.2020.2979965.

Chris Lomont. Introduction to intel advanced vector extensions. *Intel white paper*, 23, 2011.

Susumu Mashimo, Akifumi Fujita, Reoma Matsuo, Seiya Akaki, Akifumi Fukuda, Toru Koizumi, Junichiro Kadomoto, Hidetsugu Irie, Masahiro Goshima, Koji Inoue, et al. An open source fpga-optimized out-of-order risc-v soft processor. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 63–71. IEEE, 2019.

Duncan J.M Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H.W. Leong. A customizable matrix multiplication framework for the intel harpv2 xeon+fpga platform: A deep learning case study. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 107–116, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356145. doi: 10.1145/3174243.3174258. URL `https://doi.org/10.1145/3174243.3174258`.

Mohanish Narayan and Louis-Noel Pouchet. PolyBench/Fortran 1.0, 2012. URL `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/polybench-fortran.html`.

Nvidia. *Nvidia Tesla V100 GPU Architecture*. Volta Architecture Whitepaper, 2017.

Nvidia. *Nvidia Turing GPU Architecture*. Nvidia Turing Architecture Whitepaper, 2018.

Nvidia. *Nvidia A100 Tensor Core GPU Architecture*. Nvidia Ampere Architecture Whitepaper, 2020.

NVIDIA. Nvblas library: User guide. https://docs.nvidia.com/cuda/nvblas/, 2022.

Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

Alex Peleg, Sam Wilkie, and Uri Weiser. Intel mmx for multimedia pcs. *Communications of the ACM*, 40(1):24–38, 1997.

Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, and Luca Benini. Mr. wolf: An energy-precision scalable parallel ultra low power soc for iot edge processing. *IEEE Journal of Solid-State Circuits*, 54(7):1970–1981, 2019.

Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70, 2020. doi: 10.1109/HPCA47549.2020.00015.

Richard M Russell. The cray-1 computer system. *Communications of the ACM*, 21(1): 63–72, 1978.

Robert Schaller. Moore's law: Past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997.

Robert Schreiber. A systolic architecture for singular value decomposition. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1983.

André Seznec. A new case for the tage branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 117–127, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450310536. doi: 10.1145/2155620.2155635. URL `https://doi.org/10.1145/2155620.2155635`.

James E. Smith. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.*, 2(4):289–308, nov 1984. ISSN 0734-2071. doi: 10.1145/357401.357403. URL `https://doi.org/10.1145/357401.357403`.

SiFive TileLink Specification. Sifive tilelink specification, 2018. URL `https://www.sifive.com/`.

Volker Strassen et al. Gaussian elimination is not optimal. *Numerische mathematik*, 13 (4):354–356, 1969.

J. E. Thornton. The cdc 6600 project. *Annals of the History of Computing*, 2(4):338–348, 1980. doi: 10.1109/MAHC.1980.10044.

R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, pages 25 – 33, 1967.

UC Berkeley Architecture Research. riscv-benchmarks (hwacha branch), set 2022. URL `https://github.com/ucb-bar/riscv-benchmarks/tree/master/hwacha`.

Verilator. Verilator, Open-source tool for Verilog HDL simulation, out 2022. URL `https://www.veripool.org/verilator/`.

Andrew Waterman and Krste Asanovic. The risc-v instruction set manual, volume i: User-level isa. https://riscv.org/, Dec 2019.

Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 887–898, 2012.

Virginia Vassilevska Williams. Multiplying matrices in o (n2. 373) time. *preprint*, 2014.

Zhang Xianyi and Martin Kroeker. OpenBLAS: An optimized BLAS library, 2020. URL `https://www.openblas.net/`.

Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind Arvind. Composable building blocks to open up processor design. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 68–81. IEEE, 2018.

Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, May 2020.