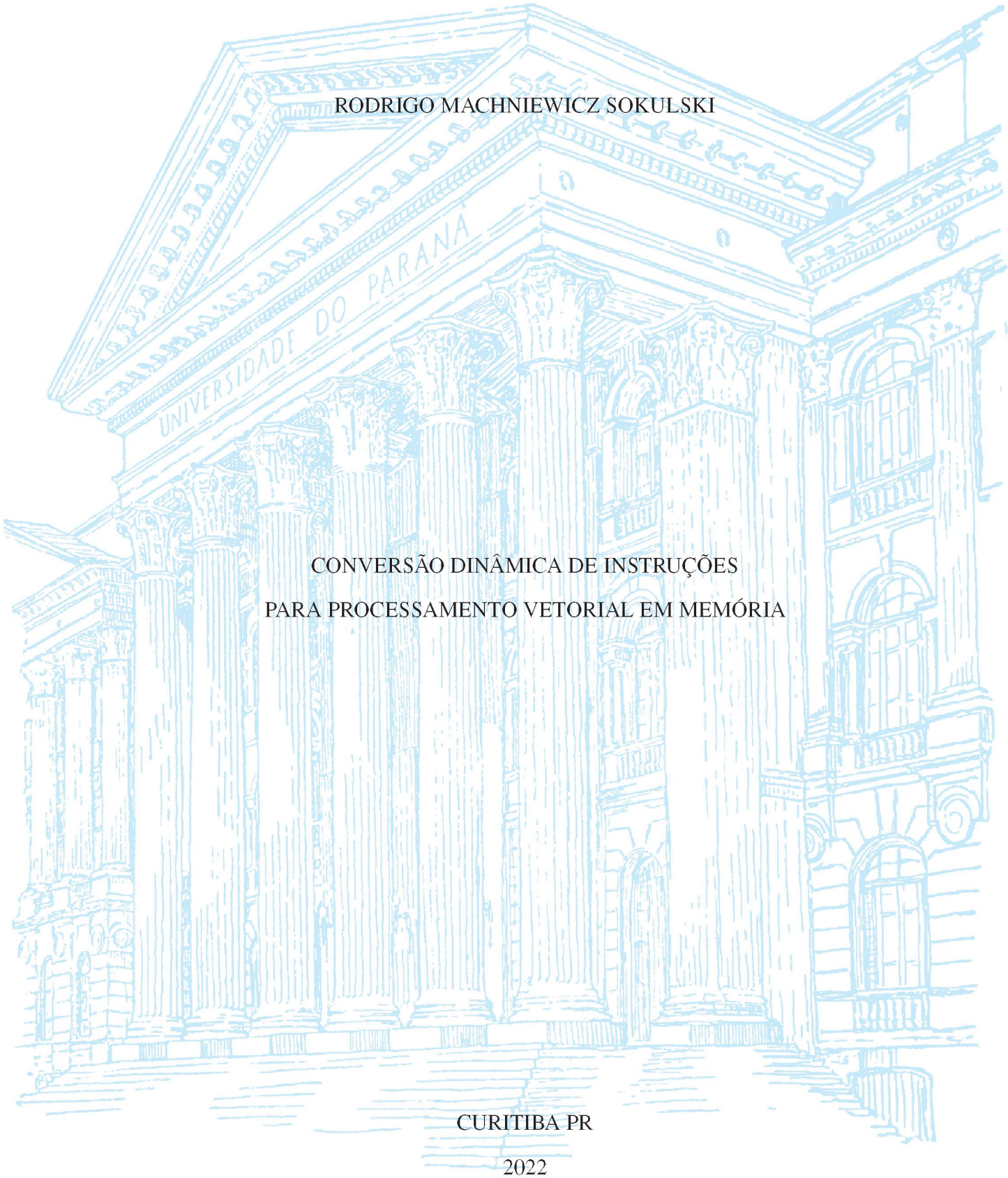
UNIVERSIDADE FEDERAL DO PARANÁ

RODRIGO MACHNIEWICZ SOKULSKI



CONVERSÃO DINÂMICA DE INSTRUÇÕES PARA PROCESSAMENTO VETORIAL EM MEMÓRIA

CURITIBA PR

2022

RODRIGO MACHNIEWICZ SOKULSKI

CONVERSÃO DINÂMICA DE INSTRUÇÕES PARA PROCESSAMENTO VETORIAL EM MEMÓRIA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Marco Antonio Zanata Alves.

CURITIBA PR

2022

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Sokulski, Rodrigo Machniewicz
Conversão dinâmica de instruções para processamento vetorial em memória / Rodrigo Machniewicz Sokulski. – Curitiba, 2022.
1 recurso on-line : PDF.

Dissertação (Mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: Marco Antonio Zanata Alves

1. Processamento eletrônico de dados. 2. Sistemas de memória de computadores. 3. Memória magnética (computadores). 4. Vetorização. I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Alves, Marco Antonio Zanata. IV. Título.

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **RODRIGO MACHNIEWICZ SOKULSKI** intitulada: **CONVERSÃO DINÂMICA DE INSTRUÇÕES PARA PROCESSAMENTO VETORIAL EM MEMÓRIA**, sob orientação do Prof. Dr. MARCO ANTONIO ZANATA ALVES, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 09 de Dezembro de 2022.

Assinatura Eletrônica
14/12/2022 15:32:55.0
MARCO ANTONIO ZANATA ALVES
Presidente da Banca Examinadora

Assinatura Eletrônica
14/12/2022 11:43:32.0
ANTONIO CARLOS SCHNEIDER BECK FILHO
Avaliador Externo (UNIVER. FEDERAL DO RIO GRANDE DO SUL)

Assinatura Eletrônica
16/12/2022 14:09:55.0
LUIS CARLOS ERPEN DE BONA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

A todos aqueles que, de alguma maneira, me ajudaram a chegar até aqui.

AGRADECIMENTOS

Agradeço aos meus pais Shirlei Terezinha Machniewicz e José Moacyr Gasparello Sokulski por todo o apoio e empenho durante minha formação, sem eles nada teria sido possível. Também ao meu irmão Renan Machniewicz Sokulski, por todas as conversas e companheirismo durante esta jornada.

Agradeço também a meu orientador Marco Antonio Zanata Alves, por me guiar todo esse caminho, desde a iniciação científica, sempre com paciência, palavras sábias, experiências de vida, empolgação e sugestões infindáveis de caminhos a seguir.

Da mesma maneira, agradeço a todos os meus colegas e amigos, pelos ótimos dias que passamos juntos, por suas sugestões, incentivos, colaboração e oportunidades abertas, vocês fizeram toda a diferença para este trabalho acontecer.

Por fim, gostaria de agradecer à Glenda Proença Train, que me incentivou a trilhar este caminho, me apoiando em todos os momentos deste trabalho. Sua companhia, seus risos e suporte foram essenciais durante os períodos mais sombrios da pandemia e continuam a iluminar meus dias. Além disso, suas revisões, críticas e prévias foram fundamentais, tanto para o desenvolvimento deste trabalho, quanto para meu desenvolvimento pessoal.

RESUMO

O rápido desenvolvimento dos sistemas computacionais nas últimas décadas levou ao surgimento de uma grande disparidade entre as velocidades de processamento e fornecimento de dados, ocasionando o problema conhecido como *Memory Wall*. Dispositivos *Near-Data Processing* (NDP), também denominados *Processing-In-Memory* (PIM), buscam mitigar esse problema adicionando unidades de processamento próximas aos dados ou junto aos circuitos que os armazenam. O surgimento das memórias 3D, com uma camada lógica capaz de suportar circuitos adicionais, fez emergir diversas novas propostas de dispositivos NDP. Dentre essas propostas, as mais eficientes executam instruções vetoriais próximas à memória. A *Vector-In-Memory Architecture* (VIMA) é um exemplo desse processamento vetorial. Essa arquitetura propõe a inclusão de algumas unidades funcionais vetoriais na camada lógica de memórias 3D, permitindo que instruções a serem executadas dentro da memória acessem o paralelismo interno dela. O uso de dispositivos NDP, em sua grande maioria, depende de conhecimentos específicos de programadores sobre *intrinsics* ou bibliotecas especializadas que acessam suas funcionalidades. Porém, essa dependência dificulta a adoção desses dispositivos, além de ser suscetível a erros. Outra abordagem seria o uso de compiladores capazes de fornecer conversões automáticas para NDP. Contudo, a recompilação de programas proprietários ou antigos pode ser impraticável, enquanto a manutenção de diferentes versões de código para as diversas variações de NDP pode ser inviável. De forma alternativa, extensões de *hardware* também podem ser usadas para essas conversões. Esses mecanismos são capazes de converter as instruções que seriam executadas pelo processador para instruções NDP durante a execução, não exigindo compilações especializadas ou conhecimentos adicionais por parte dos programadores. Porém, até onde sabemos, a tradução binária foi proposta apenas para NDPs com instruções escalares, dependendo do auxílio de bibliotecas especializadas para suas conversões. A contribuição deste trabalho é o Simple AVX to PIM Vectorizer (SAPIVe), uma extensão de *hardware* capaz de converter durante a execução as instruções vetoriais de arquiteturas com processadores tradicionais para instruções vetoriais de processamento em memória VIMA. Ao ser avaliado sobre programas sintéticos ideais para a conversão, o mecanismo proposto obtém ganhos de desempenho no tempo de execução de até $5x$, em relação a uma execução tradicional em CPU, sem perdas significativas para aplicações reais fora de seus padrões esperados de conversão. Tais ganhos podem aumentar em sistemas futuros, visto que o crescimento no tamanho do *Reorder Buffer* (ROB) dos processadores permite aumentar o número de conversões paralelas realizadas pelo SAPIVe.

Palavras-chave: Processamento próximo aos dados; Vetorização; Memórias Inteligentes;

ABSTRACT

The fast development of computer systems in recent decades has led to the rise of a significant disparity between the processing and data supply speeds, causing the problem known as Memory Wall. *Near-Data Processing* (NDP) devices, also known as *Processing-In-Memory* (PIM) devices, seek to mitigate this problem by adding processing units close to the data or close to the circuits that store it. The emergence of 3D memories, with a logical layer capable of supporting additional circuits, gave rise to several new proposals for NDPs devices. Among these proposals, the most efficient execute vector instructions close to memory. *Vector-In-Memory Architecture* (VIMA) is an example of this vector processing. This architecture offers the inclusion of some vector units in the logical layer of 3D memories, allowing instructions executed inside the memory to access its internal parallelism. The use of NDP devices, for the most part, depends on programmers' specific knowledge of intrinsics or specialized libraries that access their functionality. However, this dependency hinders these techniques adoption, in addition to being susceptible to errors. Another approach would be to use compilers capable of providing automatic conversions for NDP, however recompiling proprietary or old programs may be impractical, while maintaining different code versions for the many variations of NDP may be unfeasible. Alternatively, hardware extensions can also be used for these conversions. These mechanisms are capable of converting the instructions that would be executed by the processor to NDP instructions in runtime, not requiring specialized compilations or additional knowledge on the part of programmers. Nonetheless, as far as we know, binary translation only have been proposed for NDPs with scalar instructions, depending on the aid of specialized libraries for its conversions. The main contribution of this work is Simple AVX to PIM Vectorizer (SAPIVe), a hardware extension capable of converting in runtime the traditional vector instructions into VIMA memory vector instructions. When evaluated on synthetic programs that are ideal for conversion, the proposed mechanism obtains speedups of up to 5x, in relation to a usual *Central Processing Unit* (CPU) execution, without significant losses for real applications outside their expected instructions pattern standards for conversion. Such gains may increase in future systems since the size growth in processor *Reorder Buffer* (ROB) allows increasing the number of SAPIVe parallel conversions.

Keywords: Near-data processing; Vectorization; Smart memories;

LISTA DE FIGURAS

1.1 Classificação de *smart memories*.. . . . 19
1.2 Padrões de cópia de memória, operação em um vetor e operação entre dois vetores, elemento a elemento. . . . 20
1.3 Equivalência entre conjunto de iterações AVX e instrução VIMA. . . . 21

2.1 Tendência de melhoria na performance das CPUs e memórias entre 2004 e 2020. 23
2.2 Representação de memória 3D.. . . . 27
2.3 Componentes da VIMA e sua relação com a memória 3D. . . . 29
2.4 Passos para a execução de uma instrução VIMA. . . . 30

4.1 Conjuntos convertidos pelo SAPIVe. . . . 45
4.2 Equivalência entre códigos AVX-512 e VIMA-256 B.. . . . 45
4.3 Exemplo de conversão sobre um laço de cópia de memória.. . . . 46
4.4 Estado do mecanismo após encontrar o primeiro *load* de um conjunto. . . . 49
4.5 Vínculo de instruções na cópia de memória. . . . 50
4.6 Estado do mecanismo após o *store* de uma cópia de memória.. . . . 50
4.7 Vínculo de instruções na operação entre vetores.. . . . 51
4.8 Estado do mecanismo após o segundo *load* de um conjunto. . . . 51
4.9 Estado do mecanismo após a operação aritmética ou lógica de um conjunto. . . . 52
4.10 Estado do mecanismo após o *store* de um conjunto de operação entre vetores. . . 52
4.11 Leitura indevida de registradores de conjunto. . . . 53
4.12 Escrita indevida de registradores de conjunto. . . . 53
4.13 Diagrama de transição de estados da identificação de conjuntos de instruções pelo SAPIVe . . . 54
4.14 Equivalência entre instruções AVX contíguas e instruções VIMA. . . . 57
4.15 Validação de conversões no SAPIVe.. . . . 59
4.16 Problemas na conversão de instruções AVX com operações inibidas por máscaras. 62
4.17 Envio dos dados, da última iteração convertida, da VIMA para a CPU. . . . 63
4.18 Comportamento de conversões consecutivas do SAPIVe. . . . 64
4.19 Passos para a conversão de um laço. . . . 65

5.1 Efeito das conversões de AVX-256 e AVX-512 para VIMA-256 B nos algoritmos de cópia de memória e soma vetorial. . . . 71
5.2 Efeito das conversões de AVX-512 para VIMA-256 B, 512 B e 1024 B nos algoritmos de cópia de memória e soma vetorial. . . . 72

5.3 Efeito da agressividade na conversão de instruções AVX-512 para VIMA-512 B no algoritmo de cópia de memória. 74
5.4 Efeito da agressividade na conversão de instruções AVX-512 para VIMA-512 B no algoritmo de soma entre vetores. 75
5.5 Aceleração obtida pelo SAPIVe e por *intrinsics* sobre os algoritmos de cópia de memória e soma entre vetores, compilados com instruções AVX-512, utilizando instruções VIMA-512 B. 76
5.6 Aceleração obtida pelo SAPIVe e por *intrinsics* sobre os algoritmos de cópia de memória e soma entre vetores, compilados com instruções AVX-512, utilizando instruções VIMA-512 B, em um sistema aprimorado. 77
5.7 Aceleração obtida pelo SAPIVe e por *intrinsics* sobre os algoritmos de cópia de memória e soma entre vetores, compilados com instruções AVX-512, utilizando instruções VIMA-1024 B, em um sistema aprimorado. 78
5.8 Classificação de laços escalares por número de iterações. 81
5.9 Classificação de laços vetoriais (128 bits) por número de iterações. 81
5.10 Classificação de laços vetoriais (256 bits) por número de iterações. 82
5.11 Número de laços com diferentes tipos de instruções. 83
5.12 Laços apresentando um padrão de cópia de memória e padrões de operação sobre um e dois vetores. 83
5.13 Laço apresentando um padrão de cópia de memória junto a outra instrução que também acessa a memória. 84
5.14 Laço contendo um padrão de cópia de memória junto a outra operação que lê o valor copiado. 84
5.15 Laço contendo um padrão de cópia de memória formado em um dos caminhos de uma instrução condicional. 87

A.1 Falha de conversão durante a última iteração de um laço. 100
A.2 Aceleração por conversões de AVX-256 para VIMA-256 B no algoritmo de cópia de memória. 100
A.3 Acelerações por conversões de AVX-256 para VIMA-256 B, no algoritmo de cópia de memória, afetadas pela primeira instrução VIMA. 101
A.4 Início da primeira conversão de AVX-256 para VIMA-256 B no algoritmo de cópia de memória. 101
A.5 Conversões ideais de AVX-256 para VIMA-256 B no algoritmo de cópia de memória. 102
A.6 Conversões incompletas de AVX-256 para VIMA-256 B no algoritmo de cópia de memória. 103
A.7 Instruções atrasadas devido a conversões incompletas de AVX-256 para VIMA-256 B, no algoritmo de cópia de memória. 104
A.8 Aceleração por conversões de AVX-256 para VIMA-512 B no algoritmo de cópia de memória. 105

A.9 Aceleração por conversões de AVX-256 para VIMA-1024 B no algoritmo de cópia de memória.. 105
A.10 Aceleração por conversões de AVX-512 para VIMA-256 B no algoritmo de cópia de memória.. 106
A.11 Aceleração por conversões de AVX-512 para VIMA-512 B no algoritmo de cópia de memória.. 107
A.12 Aceleração por conversões de AVX-512 para VIMA-1024 B no algoritmo de cópia de memória.. 107
A.13 Aceleração por conversões de AVX-256 para VIMA-256 B no algoritmo de soma entre vetores. 108
A.14 Aceleração por conversões de AVX-256 para VIMA-512 B no algoritmo de soma entre vetores. 108
A.15 Aceleração por conversões de AVX-256 para VIMA-1024 B no algoritmo de soma entre vetores. 109
A.16 Aceleração por conversões de AVX-512 para VIMA-256 B no algoritmo de soma entre vetores. 110
A.17 Instruções atrasadas devido a conversões incompletas de AVX-512 para VIMA-256 B no algoritmo de soma entre vetores.. 111
A.18 Aceleração por conversões de AVX-512 para VIMA-512 B no algoritmo de soma entre vetores. 112
A.19 Aceleração por conversões de AVX-512 para VIMA-1024 B no algoritmo de soma entre vetores. 112
A.20 Risco de tentativa adicional de conversão. 113
A.21 Obtenção de ganhos em tentativas de conversão.. 113
A.22 Conversões ideais de AVX-256 para VIMA-512 B no algoritmo de soma entre vetores. 114
A.23 Potenciais ganhos de uma conversão adicional de AVX-256 para VIMA-512 no algoritmo de soma entre vetores. 114
A.24 Potenciais ganhos de uma conversão adicional de AVX-256 para VIMA-256 no algoritmo de soma entre vetores. 115
A.25 Conversões incompletas de AVX-256 para VIMA-512 B no algoritmo de soma entre vetores. 115
A.26 Potenciais perdas de uma conversão adicional de AVX-256 para VIMA-512 B no algoritmo de soma entre vetores. 116
A.27 Efeito da agressividade na conversão de AVX-256 para VIMA-256 B no algoritmo de cópia de memória. 117
A.28 Efeito da agressividade na conversão de AVX-256 para VIMA-512 B no algoritmo de cópia de memória. 118
A.29 Efeito da agressividade na conversão de AVX-256 para VIMA-1024 B no algoritmo de cópia de memória. 119
A.30 Efeito da agressividade na conversão de AVX-512 para VIMA-256 B no algoritmo de cópia de memória. 120

A.31 Efeito da agressividade na conversão de AVX-512 para VIMA-512 B no algoritmo de cópia de memória. 121
A.32 Efeito da agressividade na conversão de AVX-512 para VIMA-1024 B no algoritmo de cópia de memória. 122
A.33 Efeito da agressividade na conversão de AVX-256 para VIMA-256 B no algoritmo de soma entre vetores.. 123
A.34 Efeito da agressividade na conversão de AVX-256 para VIMA-512 B no algoritmo de soma entre vetores.. 124
A.35 Efeito da agressividade na conversão de AVX-256 para VIMA-1024 B no algoritmo de soma entre vetores. 125
A.36 Efeito da agressividade na conversão de AVX-512 para VIMA-256 B no algoritmo de soma entre vetores.. 126
A.37 Efeito da agressividade na conversão de AVX-512 para VIMA-512 B no algoritmo de soma entre vetores.. 127
A.38 Efeito da agressividade na conversão de AVX-512 para VIMA-1024 B no algoritmo de soma entre vetores. 128
A.39 Aceleração obtida pelo SAPIVe e por *intrinsics* sobre o algoritmo de cópia de memória compilado com instruções AVX-256. 129
A.40 Aceleração obtida pelo SAPIVe e por *intrinsics* sobre o algoritmo de cópia de memória compilado com instruções AVX-512. 130
A.41 Aceleração obtida pelo SAPIVe e por *intrinsics* sobre o algoritmo de soma vetorial compilado com instruções AVX-256.. 131
A.42 Aceleração obtida pelo SAPIVe e por *intrinsics* sobre o algoritmo de soma vetorial compilado com instruções AVX-512.. 132
A.43 Efeito da agressividade na conversão de AVX-256 para VIMA-256 B no algoritmo de cópia de memória, considerando um sistema expandido. 133
A.44 Efeito da agressividade na conversão de AVX-256 para VIMA-512 B no algoritmo de cópia de memória, considerando um sistema expandido. 134
A.45 Efeito da agressividade na conversão de AVX-256 para VIMA-1024 B no algoritmo de cópia de memória, considerando um sistema expandido. 135
A.46 Aceleração obtida pelo SAPIVe e por *intrinsics* sobre o algoritmo de cópia de memória compilado com instruções AVX-256, considerando um sistema expandido.. 136
A.47 Efeito da agressividade na conversão de AVX-512 para VIMA-256 B no algoritmo de cópia de memória, considerando um sistema expandido. 137
A.48 Efeito da agressividade na conversão de AVX-512 para VIMA-512 B no algoritmo de cópia de memória, considerando um sistema expandido. 138
A.49 Efeito da agressividade na conversão de AVX-512 para VIMA-1024 B no algoritmo de cópia de memória, considerando um sistema expandido. 139
A.50 Aceleração obtida pelo SAPIVe e por *intrinsics* sobre o algoritmo de cópia de memória compilado com instruções AVX-512, considerando um sistema expandido.. 140

A.51 Efeito da agressividade na conversão de AVX-256 para VIMA-256 B no algoritmo de soma entre vetores, considerando um sistema expandido.. 141
A.52 Efeito da agressividade na conversão de AVX-256 para VIMA-512 B no algoritmo de soma entre vetores, considerando um sistema expandido.. 142
A.53 Efeito da agressividade na conversão de AVX-256 para VIMA-1024 B no algoritmo de soma entre vetores, considerando um sistema expandido. 143
A.54 Aceleração obtida pelo SAPIVe e por *intrinsics* sobre o algoritmo soma entre vetores compilado com instruções AVX-256, considerando um sistema expandido.144
A.55 Efeito da agressividade na conversão de AVX-512 para VIMA-256 B no algoritmo de soma entre vetores, considerando um sistema expandido.. 145
A.56 Efeito da agressividade na conversão de AVX-512 para VIMA-512 B no algoritmo de soma entre vetores, considerando um sistema expandido.. 146
A.57 Efeito da agressividade na conversão de AVX-512 para VIMA-1024 B no algoritmo de soma entre vetores, considerando um sistema expandido. 147
A.58 Aceleração obtida pelo SAPIVe e por *intrinsics* sobre o algoritmo de soma entre vetores compilado com instruções AVX-512, considerando um sistema expandido.148

LISTA DE TABELAS

2.1 Largura de banda de memórias 3D e DDR.. 27
2.2 Configuração de sistema, baseada na microarquitetura Skylake. 32

3.1 Responsabilidade de conversão para o processamento em memória nos trabalhos relacionados 42

4.1 Tabela de transição de estados para a identificação de conjuntos de instruções correspondentes a uma instrução VIMA. 56

5.1 Configurações de compilação das aplicações ideais **memcpy** e **vecsum**. 70
5.2 Crescimento no tamanho do ROB. 76
5.3 Lista de aplicações avaliadas no SPEC-CPU 2017.. 79
5.4 Número de iterações necessárias para a formação de cada tipo de instrução VIMA 80
5.5 Problemas com padrões de cópia de memória em instruções escalares. 85
5.6 Problemas com padrões de operação entre vetores em instruções escalares.. . . . 85
5.7 Problemas com padrões de operação entre vetores em instruções escalares.. . . . 85
5.8 Problemas com padrões de cópia de memória e operação sobre um ou dois vetores em instruções sobre dados de 128 bits. 86
5.9 Problemas com padrões de cópia de memória e operação sobre um ou dois vetores em instruções sobre dados de 256 bits. 86
5.10 Problemas com padrões de cópia de memória e operação sobre um ou dois vetores em instruções sobre dados de 512 bits. 87

A.1 Risco e ganho médio de uma tentativa de conversão adicional. 116

Lista de Acrônimos

AGU	*Address Generation Unit*
API	*Application Programming Interface*
ASIC	*Application Specific Integrated Circuit*
AVX	*Advanced Vector Extension*
AVX2	*Advanced Vector Extension 2*
BLAS	*Basic Linear Algebra Subprogram*
BTB	*Branch Target Buffer*
CAIRO	*Compiler-Assisted technique and decision model for enabling InstRuction-level Offloading*
CFG	*Control Flow Graph*
CNN	*Convolutional Neural Network*
CPU	*Central Processing Unit*
DDR	*Double Data Rate*
DL	*Deep Learning*
DLP	*Data-Level Parallelism*
DNN	*Deep Neural Network*
DRAM	*Dynamic Random Access Memory*
EDVAC	*Electronic Discrete Variable Computer*
FeFET	*Ferroelectric Field Effect Transistor*
FSM	*Finite State Machine*
FU	*Functional Unit*
GCC	*GNU Compiler Collection*
GPU	*Graphics Processing Unit*
HBM	*High-Bandwidth Memory*
HBM2	*High-Bandwidth Memory* de segunda geração
HIVE	*HMC Instruction Vector Extensions*
HMC	*Hybrid Memory Cube*
HTM	*Hardware Transactional Memory*
HyTM	*Hybrid Transactional Memory*
IMA	*In-Memory Accelerator*
IoT	*Internet of Things*
IPC	*Instruction per Cycle*
ISA	*Instruction Set Architecture*
JIT	*Just-In-Time*
LLC	*Last Level Cache*
LLVM	*Low Level Virtual Machine*
LRU	*Least Recently Used*
LSTM	*Long Short-Term Memory*
MIMD	*Multiple Instruction-Multiple Data*
MISD	*Multiple Instruction-Single Data*
MOB	*Memory Order Buffer*
NCA	*Near-Cell Accelerator*
NDP	*Near-Data Processing*

NMA	*Near-Memory Accelerator*
OoO	*Out-of-Order*
OrCS	*Ordinary Computing Simulator*
PC	*Program Counter*
PIM	*Processing-In-Memory*
PRIMO	*Processing-In-Memory cOmpiler*
REP	*Repeat Prefix*
ReRAM	*Resistive Random-Access Memory*
ROB	*Reorder Buffer*
RVU	*Reconfigurable Vector Unit*
SAPIVe	Simple AVX to PIM Vectorizer
SIMD	*Single Instruction-Multiple Data*
SiNUCA	*Simulator of Non-Uniform Cache Architecture*
SISD	*Single Instruction-Single Data*
SLP	*Superword Level Parallelism*
SPMD	*Single-Program Multiple-Data*
SRAM	*Static Random Access Memory*
SSE	*Streaming SIMD Extensions*
STM	*Software Transactional Memory*
SVE	*Scalable Vector Extensions*
TLB	*Translation Look-aside Buffer*
TLP	*Thread-Level Parallelism*
TM	*Transactional Memory*
TSV	*Through-Silicon Via*
TSX	*Transactional Synchronization Extensions*
VIMA	*Vector-In-Memory Architecture*

SUMÁRIO

1 INTRODUÇÃO 17
2 CONCEITOS IMPORTANTES 22
2.1 GARGALO DE VON NEUMANN E O MEMORY WALL 22
2.2 VETORIZAÇÃO 23
2.3 *NEAR-DATA PROCESSING* (NDP) 26
2.4 VECTOR-IN-MEMORY ARCHITECTURE (VIMA) 27
2.4.1 Organização 28
2.5 ORDINARY COMPUTING SIMULATOR (ORCS) 30
2.6 TRANSACTIONAL MEMORY (TM) 32
3 TRABALHOS RELACIONADOS 34
3.1 VETORIZAÇÃO 34
3.1.1 Compiladores com informações de execução 34
3.1.2 *Softwares* Just-In-Time (JIT) 35
3.1.3 Extensões de *hardware* 36
3.1.4 Comparação com o SAPIVe 37
3.2 NEAR-DATA PROCESSING (NDP) 38
3.2.1 Conversão manual 38
3.2.2 Conversão automática 40
4 SIMPLE AVX TO PIM VECTORIZER (SAPIVE) 44
4.1 ALTERAÇÕES DA VECTOR-IN-MEMORY ARCHITECTURE (VIMA) 47
4.2 ETAPAS DA CONVERSÃO 48
4.2.1 Identificação de conjuntos de instruções para conversão 48
4.2.2 Máquina de estados para a identificação de conjuntos de instruções para conversão 54
4.2.3 Análise de conjuntos candidatos 56
4.2.4 Validação de conversão 59
4.2.5 Confirmação de conversão 61
4.2.6 Invalidação de conversão 62
4.3 OTIMIZAÇÕES 63
4.3.1 Conversões consecutivas 63
4.3.2 Lista de conversões falhas 64
4.3.3 Agressividade de adiantamentos 65
4.4 REQUERIMENTO DE *HARDWARE* 66
4.5 RESTRIÇÕES DO MECANISMO 67

5 RESULTADOS. 69
5.1 APLICAÇÕES IDEAIS. 69
5.1.1 Efeito do tamanho da instrução AVX. 70
5.1.2 Efeito do tamanho da instrução VIMA 71
5.1.3 Efeito da agressividade e do tamanho do laço 73
5.1.4 Comparação com implementação manual 74
5.1.5 Efeito de aprimoramentos no sistema. 75
5.2 APLICAÇÕES REAIS 78
5.2.1 Tamanho dos laços 80
5.2.2 Inadequações em laços 83
6 CONCLUSÕES 90
REFERÊNCIAS 91
APÊNDICE A – RESULTADOS ADICIONAIS DAS APLICAÇÕES IDEAIS 99
A.0.1 Comportamento do mecanismo. 99
A.0.2 Relação entre riscos e ganhos. 110
A.0.3 Efeito da agressividade 117
A.0.4 Comparação com implementação manual 121
A.0.5 Efeito de aprimoramentos no sistema. 124

1 INTRODUÇÃO

Operações sobre conjuntos cada vez maiores de dados, como o treinamento de grandes redes neurais (Mikolov et al., 2011) e análises de *big data* (Ji et al., 2012) são possíveis graças à crescente velocidade de processamento dos sistemas computacionais.

Porém, esse crescimento nos dados processados não é exclusivo de grandes sistemas computacionais. Dispositivos menores, como celulares e computadores pessoais também são afetados pela crescente exigência de processamento de dados em aplicações como *web browsers*, reprodutores de vídeos e classificadores de imagens (Boroumand et al., 2018).

Para acompanhar as altas demandas por processamento, os sistemas passaram a explorar diferentes paralelismos existentes nas aplicações, como processar requisições a bancos de dados de maneira independente (Pohl, 2020) ou operar paralelamente diversos valores durante a inferência de modelos de *Deep Learning* (DL) (Wang et al., 2019). Assim, esses paralelismos foram classificados em dois tipos: *Data-Level Parallelism* (DLP) e *Thread-Level Parallelism* (TLP). O DLP consiste em um mesmo conjunto de operações aplicado a diferentes dados de maneira simultânea e independente, enquanto o TLP consiste na possibilidade de executar diferentes tarefas, simultânea e independentemente (Hennessy e Patterson, 2017).

Diversas técnicas são implementadas para a exploração desses paralelismos, como as instruções vetoriais *Single Instruction-Multiple Data* (SIMD), que exploram o DLP. Essas instruções são capazes de operar sobre diversos dados simultaneamente, sendo implementadas na maioria dos processadores atuais. Desde sistemas embarcados com processadores como o Cortex-A73 com a extensão Neon (Arm, 2016), passando pelos computadores pessoais, com o Intel® Core™ i7-10700 e sua extensão *Advanced Vector Extension 2* (AVX2) (Intel, 2020). Até em supercomputadores, como o líder do TOP500 em novembro de 2020, o supercomputador Fugaku (TOP 500, 2020), que possui processadores Arm A64FX capazes de executar operações vetoriais com até 512 bits (FUJITSU, 2020).

O uso de instruções vetoriais depende de compiladores adaptados, rotinas *intrinsic*, anotações no código, camadas adicionais de *software* ou *hardwares* especializados, com cada uma dessas técnicas apresentando sua própria dificuldade.

As *intrinsics* e anotações de código, apesar de poderem explorar a maior parte do potencial de vetorização dos algoritmos, são limitadas pelo conhecimento e habilidade de seus utilizadores, exigindo mais tempo e esforço por parte do programador para seu uso e podendo levar a graves erros de implementação. Camadas adicionais de *software* podem ser utilizadas como compiladores *Just-In-Time* (JIT) (Plangger e Krall, 2016), vetorizando aplicações com informações disponíveis durante sua execução, sem depender dos esforços de programadores, porém essa alternativa pode apresentar sobrecustos para a execução, prejudicando sistemas de alto desempenho. Já a vetorização em *hardware* também é capaz de explorar oportunidades de vetorização não disponíveis durante a compilação (Pajuelo et al., 2002; Pohl, 2020), sem exigir esforços adicionais de programadores e evitando as perdas de desempenho geradas por camadas adicionais de *software*, mas com suas operações limitadas por tempo de processamento e sobrecustos de *hardware*.

Apesar das diferentes técnicas para a exploração de paralelismos, visto que a maior parte das arquiteturas atuais é baseada no modelo de von Neumann, para que muitas das suas operações seja efetuada, instruções e dados devem ser trafegados entre a memória e a *Central Processing Unit* (CPU). Isso gera um gargalo na interconexão entre esses dois componentes que é conhecido como Gargalo de von Neumann, onde o processador depende constantemente de

dados da memória para prosseguir sua execução. Essa característica é agravada pela grande diferença de latências, de um lado, temos baixas latências de processamento e, em outro, as altas latências de obtenção de dados da memória, o que é denominado *Memory Wall*. Assim, o sistema acaba limitado pela velocidade com que adquire seus dados, impedindo a plena utilização de seu potencial de processamento (Talati et al., 2019). Além disso, como cada instrução vetorial realiza operações com diversos dados, seu uso exige um alto fluxo contínuo de dados entre processador e memória, agravando o gargalo de von Neumann. O uso de *Processing-In-Memory* (PIM), também denominado *Near-Data Processing* (NDP), é capaz de estender a arquitetura de von Neumann ao levar unidades de processamento para mais próximo do local de armazenamento dos dados (Barbalace et al., 2017) e, de acordo com a distância entre esses elementos de processamento e os dados armazenados, adotamos a classificação de Santos et al. (2021):

- *Near-Cell Accelerator* (NCA): integram suas capacidades computacionais na própria estrutura de armazenamento (células de memória e circuitos de leitura e escrita) a partir de propriedades elétricas existentes em *arrays* de *Dynamic Random Access Memory* (DRAM) (Seshadri et al., 2017) ou em *crossbar arrays* de *memristors*, como o realizado em Li et al. (2019) e Jain et al. (2018). O que é ilustrado pela Figura 1.1(a).

- *In-Memory Accelerator* (IMA): incluem elementos de processamento no mesmo módulo que as células de memória se encontram, porém com componentes distintos para processamento e armazenamento. Um exemplo desta categoria, ilustrada pela Figura 1.1(b), é a extensão vetorial *HMC Instruction Vector Extensions* (HIVE), do trabalho Alves et al. (2016).

- *Near-Memory Accelerator* (NMA): possuem componentes de processamento próximos à memória, porém em módulos separados, como no *memory controller* (Hashemi et al., 2016). O que é ilustrado pela Figura 1.1(c).

Além desses, também usamos os termos *Near-Data Processing* (NDP) e *Processing-In-Memory* (PIM) como sinônimos para descrever de forma genérica o modelo de computação que aproxima a computação da memória. Como as alternativas *In-Memory Accelerator* (IMA) e *Near-Cell Accelerator* (NCA) evitam o tráfego de dados fora da memória, elas são capazes de minimizar os impactos do gargalo de von Neumann. Além disso, por sua maior proximidade com os dados, essas alternativas permitem reduzir a latência de acesso a eles, minimizando o *Memory Wall*.

Com o surgimento de memórias 3D, como o *Hybrid Memory Cube* (HMC) e a *High-Bandwidth Memory* (HBM), que apresentam uma camada lógica e diversas camadas DRAM empilhadas, dispositivos IMA capazes de serem implementados nessa camada lógica tornaram-se mais atrativos para NDP. Esses IMAs podem ser constituídos de CPUs ou *Graphics Processing Units* (GPUs), simplificadas para atender às restrições energéticas da camada lógica; *Application Specific Integrated Circuits* (ASICs) focados na aceleração de códigos comuns em determinadas bibliotecas ou; *Functional Units* (FUs), que além de apresentarem consumos energéticos mais baixos que CPUs e GPUs são mais flexíveis que os ASICs. Dentre as alternativas IMA compostas de FUs, algumas utilizam instruções vetoriais, vantajosas por seu uso eficiente da maior vazão de dados das memórias 3D (Alves et al., 2016; Santos et al., 2017; Tomé et al., 2018; Lee et al., 2018).

Assim como para as instruções vetoriais, o acesso a instruções de memória de grande parte das técnicas NDP exige o uso de instruções específicas, que podem ser inseridas pelo compilador, por *softwares* ou *hardwares* especializados durante a execução, ou por meio de *intrinsic functions* e bibliotecas.

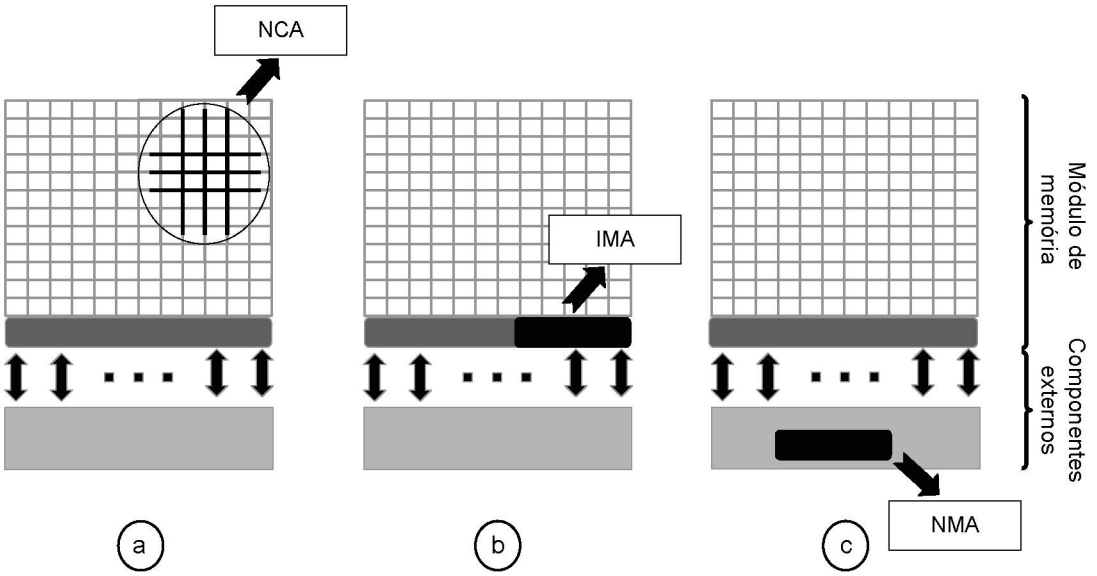


Figura 1.1: Classificação de *smart memories* de acordo com o posicionamento de seus elementos de processamento. a) Em *smart memories* que adotam *Near-Cell Accelerator* (NCA), o processamento é realizado por células de memória e seus circuitos de leitura e escrita. b) Em *smart memories* que adotam *In-Memory Accelerator* (IMA), o processamento é realizado no módulo de memória, porém em circuitos distintos aos de leitura e escrita dos dados. c) Em *smart memories* que adotam *Near-Memory Accelerator* (NMA), o processamento é realizado por unidades de processamento próximas à memória, porém fora de seu módulo. Fonte: Autor.

O uso de compiladores, *intrinsic functions* e bibliotecas é limitado por binários sem código-fonte aberto e, portanto, que não podem ser recompilados, além de exigir compiladores (Ahmed et al., 2019; Nai et al., 2017; Guo et al., 2015; Hadidi et al., 2017) e bibliotecas (Ahn et al., 2015a; Lee et al., 2019, 2018; Guo et al., 2014) específicos. Outra alternativa é uma compilação a partir das instruções do próprio binário (tradução binária estática), porém isso ainda exige compiladores especializados, além de não possuir a semântica completa do código fonte original, o que limita suas otimizações em relação aos compiladores tradicionais (Altman et al., 2000).

Já se realizado durante a execução (tradução binária dinâmica), por uma camada de *software*, como a de compiladores *Just-In-Time* (JIT) (Barrière et al., 2021), poderia gerar sobrecustos por sua compilação durante a execução, afetando aplicações que exigem alto desempenho. Porém, essa conversão também pode ser realizada em *hardware* (Ahn et al., 2015b). Apesar disso, todos os mecanismos de conversão em *hardware* encontrados (Ahn et al., 2015b; Nai et al., 2017) foram planejados para conversões simples entre instruções escalares da CPU e instruções escalares NDP, não podendo ser facilmente adaptados para NDPs vetoriais. Além disso, eles dependem do auxílio de bibliotecas especializadas, não removendo totalmente a carga dos programadores.

Assim, esta dissertação foi guiada pela seguinte pergunta de pesquisa: **apesar de suas limitações e *overheads*, a auto-vetorização em *hardware* para processamento em memória pode levar a ganhos de desempenho?** Para a responder, nesta dissertação se propõe uma extensão totalmente em *hardware* para vetorização de programas, durante a execução, utilizando instruções vetoriais IMA. O mecanismo proposto, denominado Simple AVX to PIM Vectorizer (SAPIVe), realiza conversões de laços contendo padrões de instruções *Advanced Vector Extension* (AVX) *memory-bound* pré-definidos para instruções vetoriais IMA, permitindo a aceleração de seu processamento. Para nossas conversões, utilizamos o dispositivo IMA *Vector-In-Memory Architecture* (VIMA), uma arquitetura de processamento em memória com instruções vetoriais

descrita na Seção 2.4. Cada instrução *Vector-In-Memory Architecture* (VIMA) carrega seus operandos e armazena seus resultados em memória, sem registradores auxiliares. Assim, para obter uma conversão equivalente, nosso mecanismo converte laços realizando cópias de memória, operações em um vetor e operações, elemento a elemento, entre dois vetores, padrões apresentados na Figura 1.2. Além disso, como as instruções VIMA visam utilizar de forma eficiente o paralelismo fornecido pelas memórias 3D, cada uma dessas instruções processa um volume de dados superior ao das instruções AVX. Por conta disso, diversas iterações de um laço AVX precisam ser unidas para a composição de uma única instrução VIMA, como apresentado na Figura 1.3.

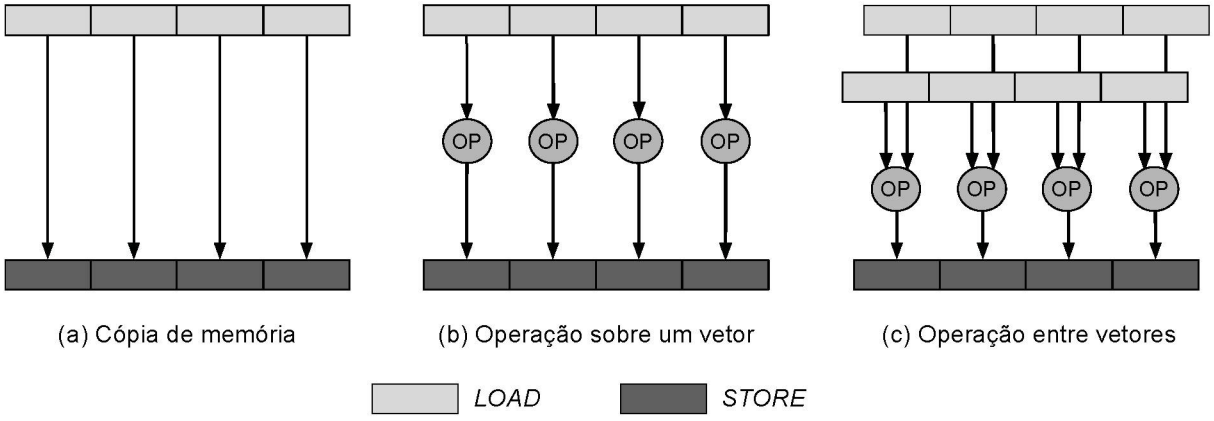


Figura 1.2: Padrões de instruções executadas pela VIMA. (a) apresenta uma cópia de memória, que consiste de uma instrução de *load* seguida de um *store*, armazenando os dados lidos em uma nova posição de memória. (b) ilustra uma operação sobre os elementos de um único vetor, como ocorre em uma operação com um imediato. (c) mostra um padrão de operação entre dois vetores, elemento a elemento, consistindo de dois *loads* que carregam dados para uma operação aritmética ou lógica, sendo que o resultado dessa operação é armazenado em memória por um *store*. Fonte: Autor.

Para nossas avaliações, o SAPIVe foi implementado no *Ordinary Computing Simulator* (OrCS) (Köhler, 2019), um simulador orientado a traços com acurácia de ciclos, descrito na Seção 2.5. Visto que os padrões (a) e (b), apresentados na Figura 1.2, realizam uma quantidade similar de acessos à memória, os resultados obtidos durante a conversão de ambos, com nosso mecanismo, são similares. Assim, por simplificação, nesta dissertação apresentamos apenas resultados referentes à conversão dos padrões (a) e (c). Com isso, para nossos experimentos, utilizamos traços de execução de duas aplicações sintéticas seguindo esses padrões, uma cópia de memória e uma soma vetorial. Os traços desses *kernels* foram extraídos utilizando o Pin (Intel, 2018), um instrumentador binário dinâmico da Intel. Além disso, para avaliar o SAPIVe em condições reais, utilizamos traços de execução pertencentes às aplicações presentes no SPEC-CPU 2017, contendo suas porções mais significativas, extraídos utilizando a tecnologia de PinPoints (Patil et al., 2004), presente no Pin.

Por simplicidade, consideramos o SAPIVe sobre um sistema com um único núcleo, executando uma única *thread*, com instruções VIMA geradas apenas por nosso mecanismo.

Nossos resultados com aplicações sintéticas mostram que, no melhor caso, o uso do SAPIVe pode levar a uma aceleração de 5x no tempo de execução, em relação a uma execução usual, com instruções AVX-512. Além de conseguir atingir 89% dos ganhos em aceleração obtidos pela utilização manual de rotinas *intrinsics* da VIMA. Nos demais laços onde a conversão é bem sucedida e o número de iterações AVX convertidas é suficiente, utilizando algumas otimizações, nosso mecanismo obtém uma aceleração mínima de quase 2x. Além disso, os

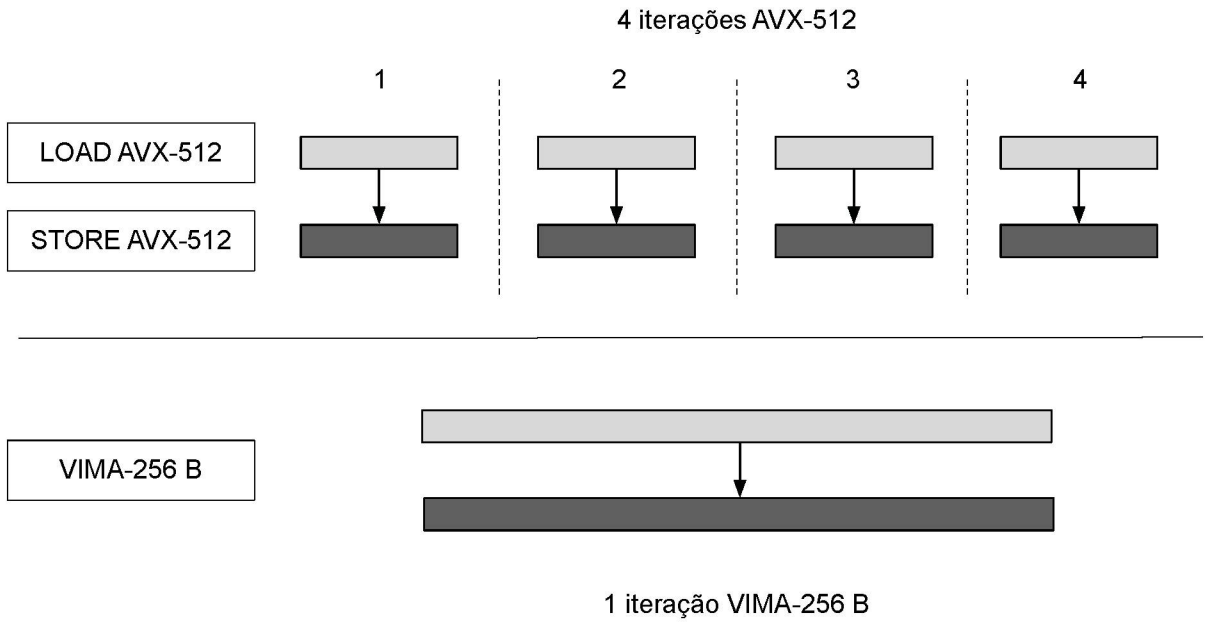


Figura 1.3: Equivalência entre conjunto de iterações AVX e instrução VIMA. Como cada operação VIMA opera sobre um grande conjunto de dados, apenas a união de várias iterações de instruções AVX é equivalente a ela. Nessa figura temos a equivalência de instruções AVX-512 e VIMA-256 B, que só é obtida pela união de 4 iterações AVX-512 (256 B/64 B). Fonte: Autor.

ganhos do SAPIVe podem ser ainda maiores ao consideramos sistemas futuros com maiores *buffers* e largura de cada estágio, chegando a alcançar o mesmo potencial das *intrinsics* VIMA.

Apesar desses ganhos, nas aplicações reais presentes no SPEC-CPU 2017, o SAPIVe não foi capaz de realizar conversões, não gerando ganhos ou prejuízos significativos. Isso, porque as restrições necessárias para a conversão de laços não são atendidas nessas aplicações, que possuem padrões de operações encadeadas complexos, e outras instruções que acabam impossibilitando nossa abordagem.

Com base em nossos resultados, conseguimos mostrar que, apesar de suas limitações e *overheads*, a auto-vetorização em *hardware* para processamento em memória pode levar a ganhos de desempenho em casos ideais, o que pode ser expandido em trabalhos futuros.

O restante deste documento está organizado da seguinte maneira: Capítulo 2 apresenta alguns conceitos importantes para a compreensão de nossa proposta; Capítulo 3 expõe alguns trabalhos relacionados ao nosso mecanismo, tanto em relação à sua conversão, que pode ser interpretada como um tipo de vetorização de instruções AVX para VIMA, quanto em relação às técnicas comumente adotadas para o uso do processamento em memória; Capítulo 4 descreve nosso mecanismo proposto, denominado Simple AVX to PIM Vectorizer (SAPIVe); Capítulo 5 apresenta os resultados da avaliação de nosso mecanismo sobre programas ideais e algumas aplicações reais presentes no SPEC-CPU 2017 e; Capítulo 6 traz a conclusão de nosso trabalho, juntamente a algumas ideias e desafios para trabalhos futuros.

2 CONCEITOS IMPORTANTES

Neste capítulo são apresentados conceitos fundamentais para esta dissertação, como algumas das limitações presentes nos sistemas computacionais baseados na arquitetura de von Neumann (von Neumann, 1993), vetorização, *Near-Data Processing* (NDP), a arquitetura de processamento em memória *Vector-In-Memory Architecture* (VIMA), o simulador *Ordinary Computing Simulator* (OrCS), utilizado em nossos experimentos, e memórias transacionais.

2.1 GARGALO DE VON NEUMANN E O MEMORY WALL

A grande maioria dos sistemas computacionais atuais é baseada na arquitetura de von Neumann (Silva Junior, 2020; von Neumann, 1993). Essa arquitetura possui como uma de suas principais características unidades de armazenamento e processamento de dados separadas, comunicando-se através de uma interconexão. A necessidade de transferências constantes de dados por essa interconexão age como um limitante ao desempenho do sistema, uma vez que essas transferências são realizadas de forma puramente sequencial. Essa limitação é atualmente denominada gargalo de von Neumann. Porém, quando a arquitetura de von Neumann foi proposta, como um rascunho para a construção do *Electronic Discrete Variable Computer* (EDVAC), ela considerava que a maior parte das operações aritméticas do sistema tinha um tempo de execução igual ao tempo de acesso à sua memória, tornando o gargalo insignificante (Knuth, 1970; Godfrey e Hendry, 1993). Com o desenvolvimento tecnológico, tubos de vácuo foram substituídos por tecnologias cada vez mais avançadas de transistores, e as memórias de linhas de atraso de mercúrio, foram substituídas por novas tecnologias *Dynamic Random Access Memory* (DRAM). Porém, com os avanços dessas tecnologias, o tempo de acesso à memória virou um empecilho ao alto desempenho dos sistemas, o que ficou conhecido como *Memory Wall*.

O *Memory Wall* é causado por dificuldades na obtenção de dados da memória na velocidade esperada pelo processador. De acordo com Preshing e Poley (2012) e K. Chang (2017), entre 2004 e 2011 a velocidade média de cada núcleo dos processadores aumentou 21% ao ano, ao mesmo tempo em que a largura de banda da DRAM aumentou em menos de 10% ao ano. Já a latência da DRAM foi melhorada em apenas 30% durante todo o período de 7 anos, resultando em uma média anual de 2%. Mesmo que só consideremos as *Central Processing Units* (CPUs) de alta performance para *desktops* lançadas entre 2005 e 2020, que de acordo com Lechner (2020), apresentaram uma tendência de aceleração de apenas 10% ao ano em cada thread, essa diferença continua significativa, visto que a latência ofertada pelas tecnologias DRAM melhoraram apenas por volta de 30% em todos esses 15 anos (K. Chang, 2017; Milan, 2019). Com isso, exemplificamos como cada vez mais os sistemas são limitados pela velocidade de fornecimento oferecida pela memória, o que agrava o *Memory Wall* e torna o gargalo de von Neumann mais significativo e incômodo.

O surgimento das memórias 3D levou a um expressivo aumento na largura de banda das memórias, porém sem trazer alterações em sua latência de fornecimento de dados pelas células DRAM (Milan, 2019), o que é ilustrado na Figura 2.1. Assim, para muitas aplicações com baixo reuso de dados ou com acessos a endereços muito distantes, onde as memórias cache não conseguem ser bem aproveitadas, esses problemas continuam significativos.

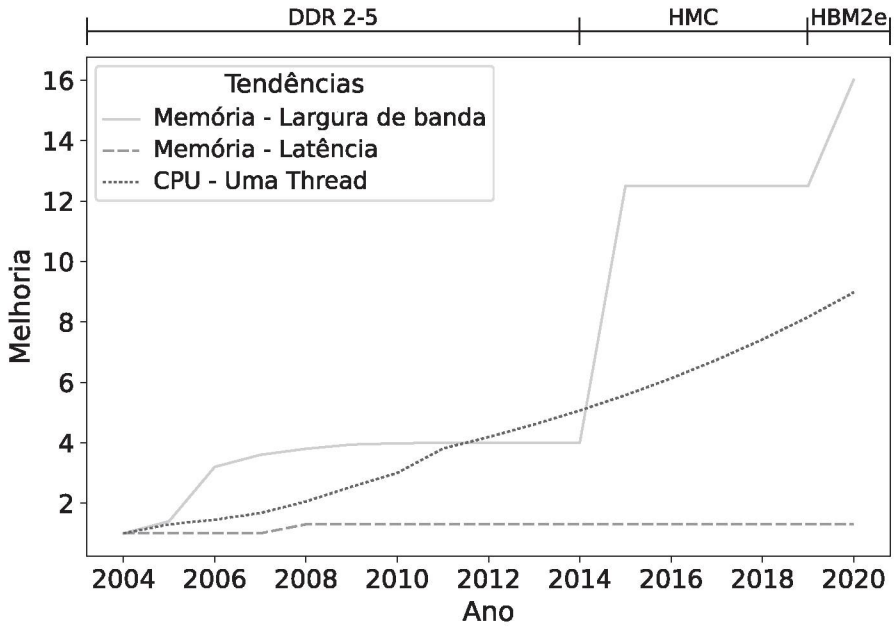


Figura 2.1: Tendência de melhoria na performance das CPUs (Preshing e Poley, 2012; Lechner, 2020) e memórias (K. Chang, 2017) entre 2004 e 2020. Consideramos memórias DDR de até quatro canais e, a partir de 2012, tendências de melhoria de 10% ao ano para a velocidade da CPU (Lechner, 2020). Fonte: Autor.

2.2 VETORIZAÇÃO

Como definido por Flynn (1966) em sua taxonomia, os sistemas computacionais podem ser categorizados da seguinte maneira:

- *Single Instruction-Single Data* (SISD): Sistemas que executam apenas um fluxo de instruções, sendo que cada instrução opera sobre um ou dois operandos, produzindo um resultado, como as instruções do IBM 700, usado como referência por Flynn.
- *Single Instruction-Multiple Data* (SIMD): Sistemas com várias unidades funcionais que executam, ao mesmo tempo, a mesma instrução sobre diferentes dados, exemplos dessa categoria são as extensões 3DNow! (Oberman et al., 1999), AVX (Intel, 2011) e Neon (Rickards, 2010).
- *Multiple Instruction-Single Data* (MISD): Sistemas onde várias instruções operam sobre o mesmo dado, como ocorre em arquiteturas sistólicas (Flynn e Rudd, 1996) e em processamento de sinais.
- *Multiple Instruction-Multiple Data* (MIMD): Sistemas com vários processadores, cada um executando um fluxo de instruções distinto sobre dados diferentes, como servidores com vários núcleos de processamento (Hennessy e Patterson, 2017).

Os sistemas modernos são uma mistura dessas várias categorias (Hennessy e Patterson, 2017), por exemplo, grandes servidores utilizam vários núcleos de processamento para distribuir sua carga, caracterizando-se como sistemas *Multiple Instruction-Multiple Data* (MIMD). Porém cada núcleo pode executar instruções escalares (*Single Instruction-Single Data* (SISD)) e vetoriais (*Single Instruction-Multiple Data* (SIMD)), compondo uma arquitetura mista.

Considerando essa arquitetura mista, esta dissertação adota o entendimento de que vetorizar um programa pode consistir tanto de transformar parte de suas instruções SISD em

Código 2.1: Laço realizando soma vetorial com instruções escalares. Fonte: Autor.

```
for (int i=0; i < SIZE; ++i) {
   RES[i] = V1[i] + V2[i];
}
```

Código 2.2: Laço realizando uma soma vetorial com o uso de instruções vetoriais, por meio de *intrinsics* AVX. Fonte: Autor.

```
for (int i=0; i <= SIZE-4; i += 4) {
   __m256i nV1 = _mm256_setr_epi64x (V1[i], V1[i+1], V1[i+2], V1[i+3]);
   __m256i nV2 = _mm256_setr_epi64x (V2[i], V2[i+1], V2[i+2], V2[i+3]);
   __m256i nRES = _mm256_add_epi64(nV1, nV2);
   _mm256_storeu_si256((__m256i *)(&RES[i]), nRES);
}
```

instruções SIMD, quanto transformar instruções SIMD já existentes em outras maiores. Ou seja, a vetorização seria qualquer conversão que substitua um conjunto de instruções por outro, onde cada nova instrução opera sobre um conjunto maior de dados que cada uma das substituídas.

A conversão de instruções SISD para SIMD, primeiro tipo de vetorização, pode ser observada nos algoritmos de soma vetorial dos Códigos 2.1 e 2.2. O algoritmo presente no Código 2.2 é uma versão vetorizada com as rotinas da biblioteca *intrinsics Advanced Vector Extension* (AVX) (Intel, 2021) do algoritmo do Código 2.1, onde as instruções de leitura, soma e escrita em memória foram vetorizadas. Como o número de elementos operados por cada instrução é alterado, o incremento utilizado para a variável de controle do laço também aumenta, porém esse aumento tem um efeito colateral. Se $SIZE$ não for um múltiplo de quatro, os $SIZE \ mod \ (4)$ elementos finais de $V1$ e $V2$ não serão processados. Portanto, o algoritmo do Código 2.3 apresenta a versão completa da vetorização, com um laço adicional tratando os elementos restantes de forma escalar.

A vetorização apresentada no exemplo anterior foi feita considerando um laço, porém ela também pode ocorrer em trechos de códigos sem laços. (Pohl et al., 2020). Nesses algoritmos, a vetorização consiste da união de diversas instruções que realizam a mesma operação, como ilustrado nos Códigos 2.4 e 2.5, onde o Código 2.5 é uma versão vetorizada do Código 2.4, reordenando e unindo as instruções de soma em instruções vetoriais AVX.

Código 2.3: Laço realizando uma soma vetorial com o uso de instruções vetoriais, por meio de *intrinsics* AVX, contendo um tratamento de resto escalar. Fonte: Autor.

```
int i;
for (i=0; i <= SIZE-4; i += 4) {
   __m256i nV1 = _mm256_setr_epi64x (V1[i], V1[i+1], V1[i+2], V1[i+3]);
   __m256i nV2 = _mm256_setr_epi64x (V2[i], V2[i+1], V2[i+2], V2[i+3]);
   __m256i nRES = _mm256_add_epi64(nV1, nV2);
   _mm256_storeu_si256((__m256i *)(&RES[i]), nRES);
}

// Cálculo escalar de partes restantes
for (; i < SIZE; ++i) {
   RES[i] = V1[i] + V2[i];
}
```

Código 2.4: Código escalar sem laços onde as instruções escalares de mesmo tipo, sem dependências (marcadas pelas setas), podem ser unidas em uma nova instrução vetorial. Fonte: Autor.

```
1 R1 = a + e; // <-
2 R5 = a / c;
3 R2 = b + f; // <-
4 R6 = g * d;
5 R3 = c + g; // <-
6 R4 = d + h; // <-
```

Código 2.5: Código vetorizado pela união de instruções escalares, independentes, realizando o mesmo tipo de operação (somas entre inteiros). Fonte: Autor.

```
1  __m256i P1 = _mm256_setr_epi64x(d, c, b, a);
2  __m256i P2 = _mm256_setr_epi64x(h, g, f, e);
3  __m256i RES = _mm256_add_epi64(P1, P2);
4
5  R5 = a / c;
6  R6 = g * d;
7
8  R1 = RES[3];
9  R2 = RES[2];
10 R3 = RES[1];
11 R4 = RES[0];
```

Essa tarefa requer análises de dependência mais simples que a vetorização direta de laços. Por conta disso, aliada a técnicas de *loop-unrolling*, pode simplificar a vetorização de laços para os compiladores (Porpodas e Jones, 2015).

A vetorização pode ficar a cargo do programador, do compilador, de sistemas *Just-In-Time* (JIT) ou do *hardware* que executa a aplicação (Pajuelo et al., 2002). A vetorização realizada pelo programador possui o maior potencial de ganho de desempenho, visto que, por conhecer o contexto da aplicação, em determinadas situações consegue otimizar mais profundamente que mecanismos automáticos. Porém essa forma de vetorização pode ser complexa e difícil de ser realizada manualmente, além de precisar ser atualizada para novas *Instruction Set Architectures* (ISAs), aproveitando o potencial dos novos sistemas, o que pode torná-la inviável (Pohl, 2020). Já a vetorização realizada por compiladores é restrita por desconhecer o contexto da aplicação, assim sua otimização está limitada a códigos genéricos que funcionam em qualquer contexto, fazendo com que muitas vezes apresente desempenho pior que a vetorização manual (Pajuelo et al., 2002; Pohl et al., 2020). Além disso, o programa deve ser compilado para cada variação de arquitetura, para utilizar as melhores extensões SIMD de qualquer sistema (Pohl, 2020; Pajuelo et al., 2002).

Mecanismos de vetorização durante a execução, além de apresentarem as facilidades da vetorização por compiladores, possuem acesso a informações de contexto não disponíveis ao compilador, simplificando diversas análises de dependência e otimizações (Maleki et al., 2011). Porém essas técnicas são limitadas por seu tempo de execução e recursos utilizados, visto que não devem afetar o desempenho das aplicações (Hallou et al., 2015; Pajuelo et al., 2002).

Nem sempre uma vetorização traz ganhos para determinada aplicação, uma vez que movimentações desnecessárias entre registradores vetoriais e escalares, padrões complexos de acesso à memória, dentre outros, podem levar a perdas de desempenho (Porpodas e Jones, 2015). Compiladores do estado da arte usam modelos de custo para definir quando realizar

ou não determinada vetorização, enquanto na vetorização manual essa decisão fica a cargo do programador (Porpodas e Jones, 2015; Pohl et al., 2020). Já na vetorização durante a execução, essa decisão ocorreria por restrições impostas a instruções selecionadas para vetorização ou algoritmos simples, devido às suas restrições de tempo.

Com base nisso, o Simple AVX to PIM Vectorizer (SAPIVe), proposto nessa dissertação, realiza vetorizações em *hardware*, durante a execução, unindo instruções AVX de diversas iterações de um laço para a composição de uma instrução *Vector-In-Memory Architecture* (VIMA), que opera, em memória, os dados de todas as instruções AVX unidas.

2.3 *NEAR-DATA PROCESSING* (NDP)

Surgindo como uma extensão à arquitetura de von Neumann para a mitigação do *Memory Wall*, o uso de *Near-Data Processing* (NDP) permite que operações sejam executadas de maneira mais próxima aos dados, evitando o sobrecusto da movimentação de dados pela hierarquia de caches e interconexões (Silva Junior, 2020).

Assim como Santos et al. (2021), neste trabalho, separamos o conceito de NDP em três categorias: *Near-Cell Accelerator* (NCA), *In-Memory Accelerator* (IMA) e *Near-Memory Accelerator* (NMA), de acordo com o posicionamento dos componentes de processamento em relação aos dados armazenados. Além disso, adotamos os termos NDP e *Processing-In-Memory* (PIM) como sinônimos para descrever o modelo de computação que aproxima a computação da memória (englobando NCA, IMA e NMA).

O NCA integra suas capacidades computacionais na própria estrutura de armazenamento a partir de propriedades elétricas de *Dynamic Random Access Memories* (DRAMs) (Seshadri et al., 2017) ou de *crossbar arrays* de *memristors* (Li et al., 2019). O IMA insere elementos como *Central Processing Units* (CPUs) e *Graphics Processing Units* (GPUs) simplificadas, *Application Specific Integrated Circuits* (ASICs) ou *Functional Units* (FUs) dentro do módulo de memória, porém fora de suas estruturas de armazenamento, como em Silva Junior (2020) e Jain et al. (2018), tendo sido impulsionado pelo surgimento das memórias 3D. Essas memórias possuem uma camada lógica que permite uma integração mais simples com IMAs, fomentando a criação de diversas técnicas para esse propósito (Alves et al., 2016; Santos et al., 2017; Tomé et al., 2018; Lee et al., 2018; Nai et al., 2017; Guo et al., 2015; Ahn et al., 2015a). Já o NMA consiste na adição de elementos de processamento próximos à memória, porém fora de seu módulo, e pode ser utilizado por componentes com requisitos de energia e área superiores aos suportados pela camada lógica de memórias 3D (Silva Junior, 2020), como CPUs e GPUs mais complexas integradas junto ao controlador de memória.

Hybrid Memory Cube (HMC) e *High-Bandwidth Memory* (HBM) são exemplos de memórias 3D que utilizam a comunicação entre diversas camadas de DRAM e uma camada lógica, como ilustrado na Figura 2.2, para fornecer altas vazões de dados ao sistema. Suas camadas DRAM empilhadas são conectadas por uma tecnologia denominada *Through-Silicon Via* (TSV), formando conjuntos de uma camada lógica e várias camadas de DRAM empilhadas denominados *vaults*. Os *vaults* são capazes de operar independentemente, de maneira semelhante a canais de DRAM, aumentando a largura de banda e paralelismo do sistema (Jeddeloh e Keeth, 2012; Silva Junior, 2020; Cordeiro et al., 2017; Akin et al., 2015). As camadas DRAM de cada *vault* são divididas em *banks*, também capazes de operar de forma independente, fornecendo maior paralelismo de acordo com o escalonamento de suas requisições (Silva Junior, 2020; Santos et al., 2017). Os dados dessas memórias 3D são acessíveis através de vias capazes de enviar e receber dados de qualquer *vault* em alta frequência e com pouca interferência (Cordeiro

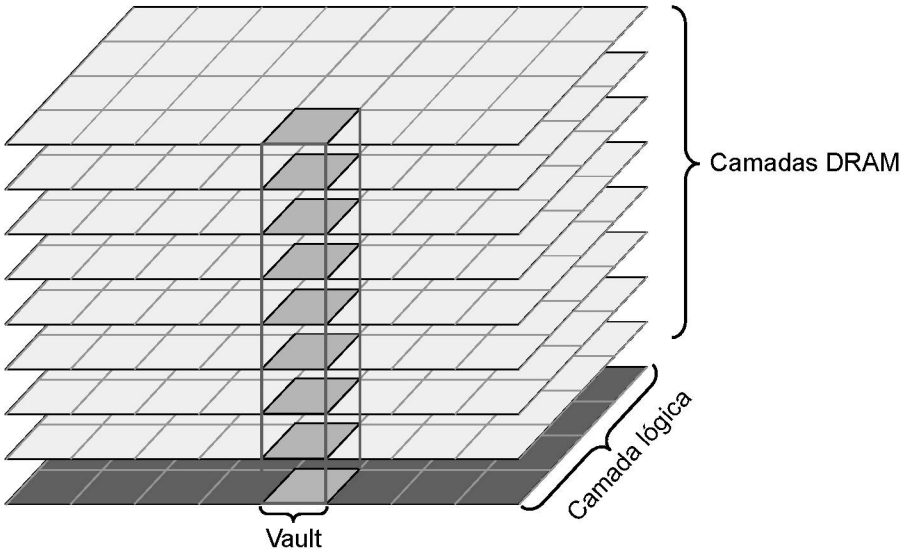


Figura 2.2: Representação de memória 3D composta de camadas de DRAM empilhadas sobre uma camada lógica. Essas camadas são separadas em blocos denominados *vaults*, onde cada camada é conectada verticalmente com as demais por *Through-Silicon Vias* (TSVs) (não representadas nesta imagem). Fonte: Autor.

et al., 2017). Com isso, esses dispositivos conseguem disponibilizar larguras de banda superiores às atingidas pelas memórias *Double Data Rate* (DDR), como apresentado na Tabela 2.1.

Tabela 2.1: Largura de banda de memórias 3D e algumas versões DDR considerando 4 canais, adaptada de Cordeiro (2020).

Versão	Largura de banda
DDR4	102 GiB/s
DDR5	166 GiB/s
HMC	320 GiB/s
HBM2e	410 GiB/s

De forma a possibilitar a execução de instruções em memória, o acesso ao HMC é feito com um protocolo simples, que permite leituras e escritas de tamanhos variados entre 16 e 256 bytes e a execução de operações aritméticas e binárias de 8 ou 16 bytes por sua camada lógica (Jeddeloh e Keeth, 2012; Cordeiro et al., 2017). Assim como ocorre com as instruções presentes no HMC, a camada lógica das memórias 3D pode ser adaptada para a inclusão de diversos aceleradores e técnicas IMA mais complexas, como as propostas em Lee et al. (2018), Alves et al. (2016) e Alves et al. (2022), sendo, a última, o alvo para as conversões de nosso mecanismo.

2.4 VECTOR-IN-MEMORY ARCHITECTURE (VIMA)

As diversas alternativas de *Near-Data Processing* (NDP) podem ser divididas em duas categorias principais: soluções de propósito geral ou aceleradores.

Soluções de propósito geral, como as instruções nativas do *Hybrid Memory Cube* (HMC) (Nai e Kim, 2015), são projetadas para funcionar em qualquer contexto, utilizando processadores completos ou simplificados (Silva Junior, 2020; Alves et al., 2016; Tomé et al., 2018; Ahn et al., 2015b). Já os aceleradores são implementados tendo em vista aplicações

específicas e suas operações mais custosas, de forma que não são versáteis para situações que fujam a seu propósito inicial (Guo et al., 2014; Lee et al., 2018; Guo et al., 2015; Ahn et al., 2015a).

As soluções de propósito geral também podem ser classificadas em dois tipos, soluções escalares (como a utilizada em Hadidi et al. (2017)) ou vetoriais (como a apresentada em Silva Junior (2020)). As escalares utilizam instruções simples, que atuam sobre um ou dois operandos (*Single Instruction-Single Data* (SISD)). Já as vetoriais possuem instruções lógicas e aritméticas que operam sobre grandes volumes de dados paralelamente (*Single Instruction-Multiple Data* (SIMD)), aproveitando melhor a grande vazão das memórias 3D (Alves et al., 2016).

Dentre as várias técnicas NDP vetoriais existentes, essa dissertação utiliza uma variação do mecanismo *HMC Instruction Vector Extensions* (HIVE) (Alves et al., 2016) desenvolvida pelo nosso laboratório, denominada *Vector-In-Memory Architecture* (VIMA) (Alves et al., 2022). Essa variação consiste na troca do banco de registradores existente no HIVE por uma cache, permitindo uma interface de programação mais simples, reúso de dados, exceções precisas, *multi-theading* e um design expansivo. Neste trabalho adotaremos essa proposta por suas várias vantagens em relação ao design anterior (HIVE), pela facilidade de inclusão de suas instruções no *pipeline* do processador *host*, não exigindo configurações detalhadas, e pela existência de um ambiente de simulação já pronto para futuros experimentos.

2.4.1 Organização

A implementação da *Vector-In-Memory Architecture* (VIMA) consiste basicamente em adicionar algumas unidades funcionais vetoriais, uma cache de dados e um sequenciador de instruções à camada lógica da memória 3D, conforme ilustrado na Figura 2.3. Suas instruções passam pelo *pipeline* como qualquer outra requisição à memória, sendo adicionadas ao *Memory Order Buffer* (MOB) e aguardando uma resposta da VIMA antes do *commit*. Por sua vez, ao receber requisições de memória, a VIMA verifica se é alguma de suas operações e, se for, a armazena no sequenciador de instruções até que esteja pronta para execução. Por outro lado, caso a requisição não seja uma operação VIMA, ela é repassada e atendida pela memória 3D. Quando uma instrução VIMA termina, o mecanismo envia um sinal de confirmação para o processador, indicando sucesso ou exceção, e permitindo o *commit* da instrução no pipeline.

A VIMA suporta instruções ARM Neon (Rickards, 2010) para números inteiros e de ponto flutuante, onde cada operação é definida por 2 ou 3 operandos, um endereço de origem e um de destino ou dois endereços de origem e um de destino. Para as avaliações dessa dissertação, consideramos uma memória 3D com 32 *vaults*, 8 *banks* independentes por *vault* e *row buffers* de tamanho 256 B. Por conta disso, as instruções VIMA consideradas operam sobre vetores de dados de 256 B a 8 KiB. O menor tamanho corresponde ao fornecido por 1 *bank* de um único *vault* da memória 3D, o que corresponderia ao menor paralelismo dessa estrutura. Já os 8 KiB correspondem ao paralelismo completo de *vaults* desse dispositivo, quando cada um dos 32 *vaults* envia 256 B de dados paralelamente. Apesar disso, tanto a VIMA quanto o Simple AVX to PIM Vectorizer (SAPIVe) poderiam ser adaptados a outras configurações.

As operações da VIMA podem ser realizadas sobre operandos de 32 ou 64 bits. Isso implica em 2048 operações de 32 bits ou 1024 operações de 64 bits para cada vetor de 8 KiB. Essas operações poderiam ser realizadas de forma totalmente paralela, porém isso exigiria um grande número de unidades funcionais, acarretando na necessidade de grandes conexões entre a cache e as unidades de processamento da VIMA. Assim, a VIMA contém apenas 256 unidades vetoriais paralelas, reduzindo a quantidade de conexões enquanto mantém uma alta vazão de dados em suas operações. Como dois vetores são enviados simultaneamente (dois operandos),

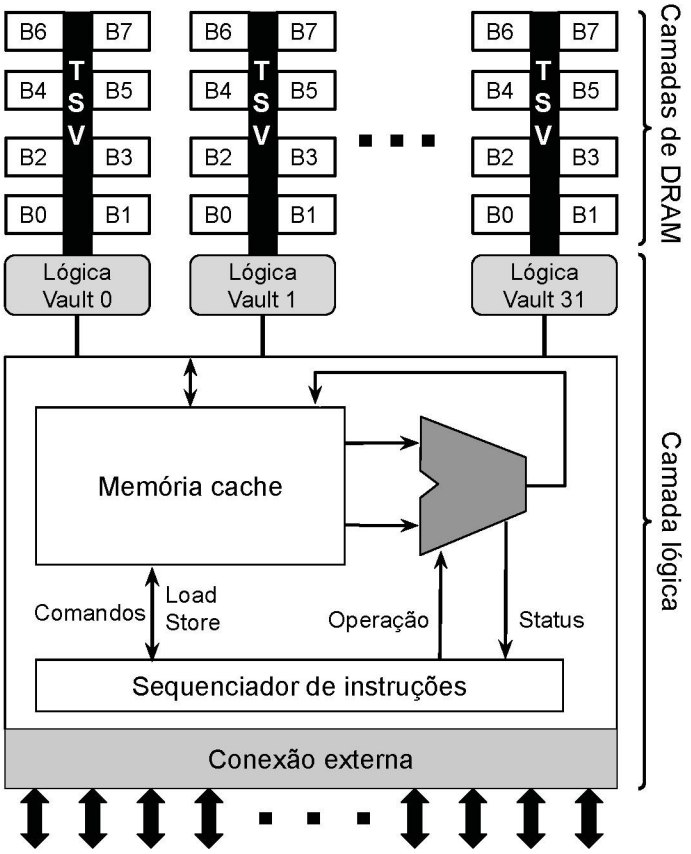


Figura 2.3: Componentes da VIMA e sua relação com a memória 3D. Por ser implementado junto à camada lógica da memória 3D, esse mecanismo tem acesso direto aos dados obtidos pelos *vaults*. Ele é composto por um sequenciador de instruções, uma memória cache e um conjunto de unidades funcionais paralelas, permitindo processamento vetorial próximo à memória. Fonte: Autor.

a cache da VIMA contém duas portas, permitindo uma execução em pipeline pelas unidades funcionais.

Para o uso da VIMA, suas instruções precisam ser adicionadas à *Instruction Set Architecture* (ISA) do sistema, como ocorre com o conjunto de instruções *Advanced Vector Extension* (AVX). Cada instrução VIMA executa seguindo os passos ilustrados na Figura 2.4. Assim como as demais requisições à memória, cada instrução VIMA é buscada, decodificada e renomeada (1), sendo inserida no *Reorder Buffer* (ROB) e no MOB (2), onde ocupa uma entrada. Em seguida, os endereços de acesso à memória dessa instrução são obtidos pela *Address Generation Unit* (AGU), no estágio de execução (3). Isso permite que o MOB envie uma requisição de operação para a memória 3D, que é interceptada pela VIMA (4). Essa requisição é armazenada no sequenciador de instruções até que seus operandos estejam disponíveis na cache de dados (5). Quando os dados estiverem disponíveis e não existirem instruções VIMA anteriores para serem executadas, a operação é realizada e seu resultado é armazenado na cache dados (6). Após o termino da execução, a VIMA envia um sinal de estado para a *Central Processing Unit* (CPU), informando sobre exceções ou sobre o sucesso da operação (7). Assim, caso a operação tenha sido bem sucedida, a CPU pode realizar o *commit* da instrução VIMA (8). Tais instruções podem ser geradas através do uso de bibliotecas *intrinsics*, com vetorizações feitas pelo próprio compilador, ou por tradução binária dinâmica, com uma camada de *software* (*Just-In-Time* (JIT))

ou uma extensão de *hardware* especializada, o que é realizado pelo mecanismo proposto por esta dissertação.

A VIMA apresenta uma memória cache de dados de 64 KiB. Essa memória permite o armazenamento de 256 vetores de 256 B ou 8 vetores com 8 KiB, sendo que o tamanho das instruções utilizadas pelo dispositivo não pode ser alterado durante a execução. Com essa memória cache, o mecanismo é capaz de manter um desempenho próximo ao da arquitetura HIVE, ao mesmo tempo em que fornece um melhor reúso de dados, uma interface de programação mais acessível, exceções precisas, capacidade de ser estendida de forma simples e suporte a processamento *multi-threading* (Alves et al., 2022; dos Santos, 2022).

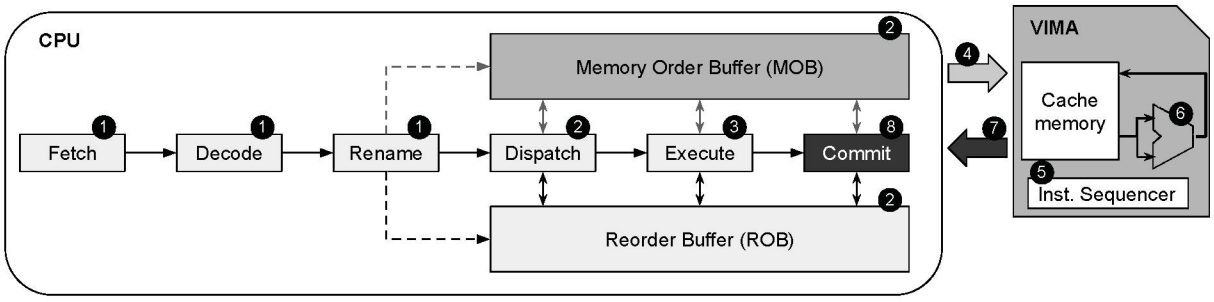


Figura 2.4: Passos para a execução de uma instrução VIMA. Cada instrução VIMA é buscada, decodificada e renomeada, sendo inserida no ROB e no MOB. Em seguida, a AGU obtém os endereços de acesso à memória dessa instrução no estágio de execução, pemitindo que ela seja enviada pelo MOB para a memória 3D. Na VIMA, essa requisição de operação é adicionada no sequenciador de instruções, executando quando seus dados se tornam disponíveis na cache de dados desse dispostivo. Ao fim da execução na VIMA, um sinal de status é enviado para a CPU, permitindo a finalização da instrução. Fonte: Autor.

Para permitir a geração de exceções precisas, a VIMA executa suas operações em ordem. Apesar disso, a VIMA realiza a requisição dos dados necessários a cada instrução o mais cedo possível, idealmente no momento em que a instrução é adicionada ao sequenciador de instruções, que possui um limite para 16 instruções VIMA concorrentes, nesta dissertação.

Para manter a coerência de memória do sistema, o protocolo de coerência adotado precisa ser adaptado para identificar as operações da VIMA e eliminar das caches dos processadores as páginas de memória que serão operadas, fazendo *write-back* de páginas sujas e invalidando todas as páginas usadas. Apesar disso, nesta dissertação, estamos negligenciando o tempo necessário, a cada instrução VIMA, para esses *write-backs* e invalidações.

Ao final de cada instrução, seu resultado, que está armazenado em um *buffer*, é inserido na cache da VIMA. Visto que todas as operações da VIMA possuem ao menos um acesso à memória, esses acessos passam pelo *Translation Look-aside Buffer* (TLB) para verificações de permissão de acesso, como as demais operações de memória executadas em *huge pages*. Apesar de serem tratadas como operações de memória, as instruções VIMA não passam pela hierarquia de caches (apenas pelo diretório de coerência), sendo enviadas de forma mais rápida ao módulo de memória 3D.

2.5 ORDINARY COMPUTING SIMULATOR (ORCS)

Como apresentado por Jain (1991), as principais formas de se avaliar um sistema computacional são através de medições em uma implementação existente, modelagem analítica e simulações. Considerando os altos custos envolvidos na construção de um novo sistema e suas dificuldades na avaliação de parâmetros, devido a influências dos processos utilizados, as medições são um método pouco utilizado para a avaliação de novas propostas de arquitetura. Já

a modelagem analítica, embora exija poucos recursos, pode simplificar efeitos importantes dos sistemas, levando a resultados imprecisos. Por conta disso, as simulações são frequentemente adotadas para avaliações de projetos de novas arquiteturas (Santos et al., 2017; Jain et al., 2018; Silva Junior, 2020; Ahn et al., 2015a,b) e são a forma de avaliação utilizada neste trabalho.

O simulador *Ordinary Computing Simulator* (OrCS) (Köhler, 2019), utilizado para nossos experimentos, é uma versão simplificada do *Simulator of Non-Uniform Cache Architecture* (SiNUCA) (Alves et al., 2015), capaz de executar códigos x86_32, x86_64 e *Vector-In-Memory Architecture* (VIMA). Esse simulador foi desenvolvido internamente pelo nosso laboratório de pesquisa e utiliza traços de execução, que são registros de uma instância de execução de uma aplicação e podem ser gerados manualmente ou por uma ferramenta denominada *Trace Generator*. Portanto ele executa traços gerados em uma máquina real com uma carga de trabalho real sem a influência do sistema operacional ou outros processos. Os traços são simulados de forma comportamental com precisão de ciclo, onde cada componente é modelado para executar suas operações com base no ciclo de clock. O *Trace Generator* utilizado é baseado no instrumentador binário dinâmico Pin, disponibilizado pela Intel (Intel, 2018), que permite o monitoramento e a execução de códigos externos durante a execução de um programa. Com o Pin, o *Ordinary Computing Simulator* (OrCS) é capaz de simular a execução completa de aplicações, porém essa simulação nem sempre é desejável, visto que demanda muito tempo e contém porções de código pouco significativas para o comportamento da aplicação, como sua inicialização. Assim, por meio da tecnologia de PinPoints (Patil et al., 2004), presente no Pin, o *Trace Generator* é capaz de selecionar as partes mais significativas de determinado programa, como seus principais laços, para encurtar o tempo de simulação enquanto mantém as características principais de cada programa.

Para simplificar suas simulações, o OrCS não executa a aplicação em um ambiente simulado, mas apenas modela as latências microarquiteturais e o comportamento de cada um de seus componentes durante a execução. Além disso, ele não modela o sistema operacional, apresentando simulações determinísticas com precisão de ciclos.

As simulações do OrCS são feitas considerando um *pipeline* superescalar configurável de 6 estágios básicos com execução fora de ordem. A funcionalidade principal de cada estágio é brevemente resumida a seguir. Mais detalhes podem ser encontrados em Alves (2014).

- ***Fetch*:** realiza a busca de instruções (*opcodes*) da aplicação, no caso do OrCS, a busca por instruções do traço simulado.
- ***Decode*:** decodifica os *opcodes* em *micro-ops*.
- ***Rename*:** identifica dependências reais entre as *micro-ops*, associando-as a registradores físicos, e as insere no *Reorder Buffer* (ROB). Além disso, *micro-ops* que realizam acessos à memória são adicionadas ao *Memory Order Buffer* (MOB).
- ***Dispatch*:** as *micro-ops* são encaminhadas para suas respectivas unidades funcionais para execução.
- ***Execute*:** as unidades funcionais realizam a execução das *micro-ops*. Além disso, são resolvidas dependências com *micro-ops* seguintes, permitindo que executem.
- ***Commit*:** conclui os *opcodes*, em ordem, ao fim de sua execução.

Para as avaliações deste trabalho, o OrCS foi configurado de acordo com a Tabela 2.2, que tem como base a microarquitetura Skylake (Doweck et al., 2017).

Devido a limitações de tempo, não implementamos um *prefetcher* no OrCS.

Tabela 2.2: Configuração de sistema, baseada na microarquitetura Skylake (Doweck et al., 2017), utilizada para os experimentos deste documento.

Processador *Out-of-Order* (OoO)					
Largura de *issue* - 6; # Entradas de Buffers: 40-*fetch*, 128-*decode*; 224-ROB;					
Tamanho do MOB: 72-*read*, 56-*write*; 1-*load* por ciclo, 1-*store* por ciclo;					
***Functional Units* (FUs):**					
ALU	4 FUs	1 ciclo;	Bit_Manipulation	2 FUs	1 ciclo;
DIV	1 FU	10 ciclos;	FP_Mov	1 FU	1 ciclo;
Shift	2 FUs	1 ciclo;	Vec_Add	2 FUs	1 ciclo;
Shuffle	1 FU	1 ciclo;	Vec_Shift	2 FUs	1 ciclo;
Slow_Int	1 FU	1 ciclo;	Vec_Mul	2 FUs	1 ciclo;
SIMD_Misc	1 FU	1 ciclo;	Vec_ALU	3 FUs	1 ciclo;
***Branch predictor*:**					
Piecewise (Jimenez, 2005). 4096 entradas no *Branch Target Buffer* (BTB);					
Cache L1 - Instruções					
32 KiB, 8 vias, 4 ciclos; linha-64 B; política *Least Recently Used* (LRU);					
Cache L1 - Dados					
32 KiB, 8 vias, 4 ciclos; linha-64 B; política LRU;					
Cache L2					
1 MiB, 16 vias, 12 ciclos; linha-64 B; política LRU;					
***Last Level Cache* (LLC)**					
16 MiB, 16 vias, 44 ciclos; linha-64 B; política LRU;					
***Dynamic Random Access Memory* (DRAM)**					
8 KiB *row buffer*; 4 GiB; DRAM@1666 MHz; Latência de instruções: 1 ciclo da CPU;					
Bursts de 8 B a uma frequência relativa de 2.5:1 entre o processador e o barramento;					
Política *Open-row*; Latências CAS, RP, RCD, RAS, CWD: 9-9-9-24-7 ciclos, respectivamente;					

2.6 TRANSACTIONAL MEMORY (TM)

A conversão de instruções realizada pelo Simple AVX to PIM Vectorizer (SAPIVe) consiste em identificar conjuntos de instruções *Advanced Vector Extension* (AVX), presentes em laços, que possam ser substituídos por uma instrução *Vector-In-Memory Architecture* (VIMA). Após a identificação de um desses conjuntos, o mecanismo gera uma instrução VIMA correspondente a futuras iterações dele, o que é denominado conversão especulativa. Assim, é possível que após uma instrução VIMA ser processada, a conversão correspondente a ela se prove incorreta. Para contornar esse problema, nosso mecanismo armazena os resultados de cada instrução VIMA em um *buffer* de escritas, com propriedades similares a uma *Transactional Memory* (TM).

As TMs, propostas por Lomet (1977) surgiram como uma alternativa para simplificar a construção de aplicações paralelas, fornecendo abstrações de alto nível para sua escrita e sincronização (Larus e Kozyrakis, 2008; Fangueiro, 2017). Para isso, esses sistemas permitem a execução paralela de seções críticas de aplicações na forma de transações, ou seja, grupos de instruções que executam paralelamente de forma atômica sobre dados compartilhados (Larus e Kozyrakis, 2008; Fangueiro, 2017). Neles, duas propriedades são garantidas entre cada transação: Atomicidade e Isolamento.

A atomicidade garante que, independentemente de quantas operações uma transação executar, seus resultados apenas estarão visíveis às demais após sua execução ser bem sucedida,

o que é denominado *commit*, ou serão todos descartados em caso de falha, o que é denominado *abort*. Já pela propriedade de isolamento, temos que uma transação não pode interferir nos resultados de outra durante sua execução, ou seja, as instruções de uma transação não aparentam estar entrelaçadas com as instruções de outras, obtendo os mesmos resultados de uma execução serial (Herlihy e Moss, 1993). Caso essas propriedades não sejam satisfeitas por alguma transação, seus resultados são descartados pela TM e novas tentativas de execução podem ser feitas (Larus e Kozyrakis, 2008; Fangueiro, 2017; Castro et al., 2019).

Por esse tratamento otimista de conflitos, o uso de transações é mais simples e pode levar a ganhos de desempenho, em relação ao uso de *locks*, em aplicações com estruturas de dados compartilhadas (Diegues et al., 2014; Fangueiro, 2017; Castro et al., 2019). Devido a isso, transações em *hardware* já foram implementadas em processadores comerciais, como com as *Transactional Synchronization Extensions* (TSX) (Yoo et al., 2013) da Intel.

De acordo com Cascaval et al. (2008) e Fangueiro (2017), dependendo dos recursos utilizados em sua implementação, uma memória transacional é classificada em uma das seguintes três categorias: (i) *Hardware Transactional Memory* (HTM), caso seja implementada apenas em *hardware*; (ii) *Software Transactional Memory* (STM), caso seja implementada apenas em *software*; ou (iii) *Hybrid Transactional Memory* (HyTM), caso seja implementada a partir de uma combinação entre *hardware* e *software*.

Visto que o objetivo deste trabalho é a proposta de um mecanismo de conversão dinâmica totalmente em *hardware*, adotaremos os termos *Transactional Memory* (TM) e *Hardware Transactional Memory* (HTM) como sinônimos pelo restante deste documento.

Como o SAPIVe precisa armazenar resultados especulativos, adaptamos a ideia de uma memória transacional em um *buffer* de escritas na VIMA. Assim, o resultado das conversões especulativas realizadas pelo SAPIVe pode ser salvo na memória, ou descartado, pelo *buffer* de escritas, conforme cada conversão seja, ou não, bem sucedida.

3 TRABALHOS RELACIONADOS

Esta dissertação propõe um mecanismo de conversão (vetorização) em *hardware* de instruções *Advanced Vector Extension* (AVX) para instruções *In-Memory Accelerator* (IMA) da *Vector-In-Memory Architecture* (VIMA). Assim, a fundamentação teórica deste trabalho pode ser dividida em duas partes: vetorização dinâmica de aplicações e técnicas para o uso de *Near-Data Processing* (NDP). Essas partes são apresentadas nas seções a seguir.

3.1 VETORIZAÇÃO

A ideia de vetorizar programas de maneira dinâmica não é nova e diversos trabalhos já apresentaram propostas para isso, focadas principalmente no contexto de processamento tradicional em *Central Processing Unit* (CPU). Essas técnicas podem ser divididas de acordo com sua arquitetura alvo e a forma de obtenção da vetorização (componente em *hardware* ou um *software* especializado).

Alguns trabalhos, como Kumar et al. (2013) e Kumar et al. (2016) exploram a vetorização dinâmica em sistemas com co-design entre *hardware* e *software*. Esses sistemas são vantajosos pela possibilidade de inserção de camadas adicionais de *software* com algoritmos específicos de vetorização dinâmica e um melhor controle e tratamento de vetorizações incorretas por ambos, *hardware* e *software*. Porém, ao contrário destes mecanismos, a proposta desta dissertação visa uma implementação em *hardware* para processadores convencionais, sem a necessidade de camadas de *software* específicas que poderiam degradar o desempenho e aumentar a complexidade do sistema.

As seções seguintes apresentam trabalhos com técnicas aplicáveis a processadores convencionais, indo das mais dependentes do compilador às independentes de *softwares*.

3.1.1 Compiladores com informações de execução

Alguns trabalhos realizam re-compilações utilizando informações da execução para aprimorar as vetorizações de um programa, como os apresentados a seguir:

Holewinski et al. (2012) propõem um método para identificar oportunidades de vetorização não aproveitadas pelo compilador. Para isso, utilizam o *Low Level Virtual Machine* (LLVM) (Lattner e Adve, 2004) para instrumentar o código avaliado durante a compilação. Assim, durante a execução, esse programa gera um grafo de dependência entre suas instruções executadas. Esse grafo é então usado para identificar conjuntos de instruções independentes durante essa execução, considerando possíveis reordenamentos que mantenham as dependências do grafo. Dentre esses conjuntos, são identificadas operações que mantém acessos com *stride* fixo à memória, representando potenciais oportunidades de paralelização *Single Instruction-Multiple Data* (SIMD), que podem ser consideradas em futuras otimizações.

Rane et al. (2015) avaliam laços que não puderam ser vetorizados pelo compilador, apenas pela análise estática, gerando recomendações para permitir essas vetorizações. Para isso, os relatórios gerados pelo compilador são avaliados em busca de laços que poderiam se beneficiar de uma melhor vetorização. Dentre esses laços, aqueles mais siginificativos para o tempo de execução, de acordo com informações de *profiling*, são instrumentados. Assim, durante a execução dessa aplicação, são obtidas métricas relevantes para auxiliar nas vetorizações do compilador, como o número de iterações executadas pelos laços e o alinhamento dos endereços

acessados por eles. Logo antes do fim dessa execução, as métricas coletadas são avaliadas, gerando recomendações de pragmas, diretivas e alterações do código para auxiliar no processo de vetorização pelo compilador.

Watkinson et al. (2019) avaliam diversos algoritmos de aprendizado supervisionado para identificar oportunidades de vetorização perdidas pelo compilador. Esses algoritmos utilizam informações obtidas por contadores de *hardware* durante a execução, como o *Instruction per Cycle* (IPC), o número de instruções de desvio e os misses do primeiro nível de cache de cada laço. Com isso, conseguem prever quando um laço pode ser vetorizado, seja pelo compilador ou manualmente, além de identificar os potenciais ganhos de cada uma dessas conversões, o que pode ser utilizado tanto como guia para otimizações manuais quanto para futuras melhorias em compiladores.

Diferentemente desses trabalhos anteriores, o Simple AVX to PIM Vectorizer (SAPIVe) visa vetorizar aplicações sem execuções prévias, de forma que mesmo um programa executado uma única vez no sistema possa obter ganhos em seu desempenho. Apesar disso, as informações obtidas pelos sistemas de *profiling* propostos por esses artigos podem ser adaptadas e utilizadas para otimizar conversões realizadas por nosso mecanismo de forma ortogonal a nossa abordagem.

3.1.2 *Softwares* Just-In-Time (JIT)

Algoritmos em *software* também podem ser usados durante a execução para identificar oportunidades de vetorização e aplicar essas otimizações, como os seguintes:

Yardimci e Franz (2008) utilizam uma fina camada de *software* como máquina virtual. Antes da primeira execução, em uma etapa estática, o programa alvo é convertido em uma representação intermediária de alto nível que facilita traduções e *profilings*. Sobre essa representação, durante a execução, otimizações e análises são realizadas em superblocos, ou seja, grupos de blocos básicos com apenas um bloco de entrada para a execução e diversos blocos de saída. Uma vez que um laço com superblocos recorrentes é identificado, o mecanismo passa a extrair dados de *profiling* como o número de iterações e os padrões de acesso à memória desse conjunto de superblocos.

Assim, dependendo das características do laço e das capacidades computacionais disponíveis, esse conjunto de superblocos pode ser paralelizado em diversos níveis, como threads simultâneas, processos ou instruções vetoriais. Esses diversos níveis de paralelização oferecem um paralelismo superior ao de nosso mecanismo, que visa apenas instruções vetoriais. Assim, o uso de camadas de *software* auxiliares e representações de mais alto nível poderia ser um caminho adotado futuramente.

Nakamura et al. (2011) propõem o *Selftrans*, capaz de fazer análises sobre códigos binários x86, realizando vetorizações durante a execução. Esse mecanismo monitora todas as aplicações, em busca de processos com um grande tempo de execução. Em seguida, ele realiza o *disassembling* das instruções necessárias para a construção de um *Control Flow Graph* (CFG), identificando as funções da aplicação. Com isso, a função mais custosa é identificada, tendo todas as suas instruções traduzidas do binário. Por fim, o laço mais custoso dessa função é avaliado em busca de possíveis vetorizações de buscas lineares, blocos básicos (*Superword Level Parallelism* (SLP)) e operações executadas sobre endereços com *stride* fixo (*Data-Level Parallelism* (DLP)). Assim, o mecanismo descrito nesse artigo realiza vetorizações não suportadas pelo nosso, como buscas lineares. Isso, porque essas buscas não realizam escritas na memória, o que é um requisito de nossas vetorizações. Porém ele exige a construção de representações intermediárias antes do início de suas otimizações, gerando *overheads* inexistentes no SAPIVe.

Nuzman et al. (2011) descrevem uma técnica de compilação em partes que utiliza a vetorização de compiladores tradicionais com o conhecimento de uma compilação *Just-In-*

Time (JIT) sobre a plataforma. Assim, criam uma vetorização portável entre plataformas com diferentes tamanhos de instrução SIMD. Para isso, a vetorização realizada pelo compilador é genérica, deixando anotações sobre o procedimento realizado, como o tamanho original dos dados processados e informações sobre seu alinhamento. Durante a execução, essas informações são utilizadas para realizar realinhamentos e para converter as vetorizações genéricas nas melhores vetorizações disponibilizadas pela arquitetura. Porém, caso essa arquitetura não apresente instruções SIMD, essa etapa também é responsável por desfazer as vetorizações e executar seu processamento escalarmente. Ao contrário dessa abordagem, nosso mecanismo busca vetorizar códigos sem nenhum tipo de anotação preexistente. Podendo funcionar independentemente do compilador adotado, contanto que instruções vetoriais AVX sejam utilizadas.

Hallou et al. (2016) propõe duas técnicas para melhorar o uso de instruções vetoriais pelos laços de uma aplicação. A primeira é capaz de converter vetorizações *Streaming SIMD Extensions* (SSE) existentes nesses laços em instruções AVX, operando sobre dados de maior tamanho. Isso é realizado com conversões praticamente diretas entre cada instrução SSE e AVX. Apesar disso, a técnica precisa garantir a manutenção da corretude realizando conversões entre registradores das duas versões, ajustando o *stride* das variáveis de controle do laço, gerenciando iterações finais quando a conversão não for possível, dentre outros ajustes. Já a segunda técnica possibilita a vetorização de instruções escalares de laços em equivalentes vetoriais durante a execução. Para isso, o código binário é convertido em uma representação em alto nível, onde o framework Polly (Grosser et al., 2012) realiza vetorizações, antes da reescrita do binário em execução. Em comparação a esse mecanismo, devido às limitações de uma extensão de *hardware*, o SAPIVe visa realizar vetorizações com o mínimo de informações sobre o fluxo de controle da aplicação. Apesar disso, assim como Hallou et al. (2016), nosso mecanismo substitui vetorizações existentes na aplicação por vetorizações mais avançadas, no caso por instruções em memória.

3.1.3 Extensões de *hardware*

Por fim, apesar de menos comuns, extensões de *hardware* podem ser utilizadas para realizar vetorizações dinâmicas, sendo que as mais relevantes encontradas são apresentadas a seguir.

Pajuelo et al. (2002) propõem um mecanismo para identificar *loads* com *stride* fixo, ou seja, instruções que carregam dados a distâncias fixas um do outro, e os vetorizar especulativamente. Após isso, vetoriza em cascata todas as instruções cujos operandos são resultado dessas instruções já vetorizadas, com exceção de instruções de escrita, que não são vetorizadas devido à natureza especulativa do mecanismo. Nosso mecanismo supera a limitação de vetorizar operações de escrita presente nesse artigo, porém está limitado a conversões mais restritas, devido à ausência de registradores na VIMA. Apesar disso, acredita-se que essa limitação pode ser superada pela adição de alguns *buffers* intermediários na VIMA em trabalhos futuros.

Kalathingal et al. (2016) apresentam um mecanismo capaz de vetorizar *threads* de aplicações *Single-Program Multiple-Data* (SPMD). Nessas aplicações, diferentes threads trabalham sobre dados distintos, frequentemente executando o mesmo conjunto de instruções. Assim, o mecanismo proposto unifica essas instruções de diferentes threads em instruções vetoriais compartilhadas, otimizando a utilização dos recursos do processador. Diferentemente dessa abordagem, nossa proposta visa aplicações com uma única *thread* para evitar problemas com acesso compartilhado à memória e consistência. Porém a abordagem de unir instruções de diferentes *threads* também poderia ser adaptada para geração de instruções em memória como um trabalho futuro.

Call Barreiro (2014) apresentam um mecanismo focando na obtenção de ganhos energéticos pela vetorização dinâmica. Esse mecanismo acumula operações do mesmo tipo para serem executadas de forma vetorial, degradando a performance por suas dependências, mas reduzindo o consumo energético por diminuir o número de instruções executadas. Essa técnica vetoriza apenas instruções aritméticas, executando *loads* e *stores* de maneira escalar. Apesar de eliminar as vetorizações especulativas que ocorrem com nosso mecanismo, essa abordagem não poderia ser implementada para instruções em memória por não vetorizar *loads* e *stores*.

Clark et al. (2007) descrevem uma técnica para tornar códigos SIMD genéricos. Nesse processo, o compilador converte instruções SIMD do programa em instruções escalares equivalentes, que são convertidas novamente para vetoriais pelo *hardware* durante a execução. Assim, o sistema é capaz de fazer a tradução para a vetorização mais adequada disponível. Essa técnica de revetorização é específica para as conversões realizadas na primeira etapa do processo (transformação de vetorial para escalar), visto que os padrões escalares devem seguir um conjunto fixo de regras, facilitando a reconversão para SIMD. Diferente dessa proposta, nosso mecanismo visa a vetorização sem pré-auxilio de *software*, de forma que seja independente de qualquer *software* adicional.

Stephens et al. (2017) apresentam a *Scalable Vector Extensions* (SVE) da ARM. Uma extensão da arquitetura ARM com um conjunto de instruções vetoriais agnóstico em relação ao tamanho dos vetores sobre os quais irá operar. Dessa forma, uma mesma vetorização pode ser utilizada por diversos sistemas, otimizados a propósitos diferentes e com instruções vetoriais de tamanhos diferentes. Também são propostas novas técnicas de compilação para essa vetorização, de forma que o peso dessas mudanças não atinja o programador. Diferentemente de nosso trabalho, a vetorização realizada por eles parte de um grande vetor, quebrando em menores conforme disponível pelo *hardware*, enquanto partimos de vetores AVX menores para a construção de um grande vetor VIMA. Além disso, buscamos realizar conversões diretamente em *hardware*, sem o auxílio de novas compilações. Para isso, partimos da extensão AVX, já disponível em diversos sistemas, evitando a recompilação necessária para o uso da SVE. Além de que realizamos conversões para instruções em memória, as quais possuem mais restrições, sobrecustos e ganhos diferentes das instruções vetoriais tradicionais.

3.1.4 Comparação com o SAPIVe

Diferentemente dos trabalhos anteriores, que realizam vetorizações para a CPU, realizamos vetorizações para instruções NDP. Essa vetorização para instruções em memória, utilizando a VIMA apresenta novos desafios, como a necessidade de realizar endereçamento direto na memória a cada operação, sem o auxílio de registradores, processamento em ordem e menos eficiente para operações limitadas pela CPU e novos custos para a transferência de resultados de uma operação em memória para uma instrução executada na CPU.

Além disso, em ambos os casos dependentes de *softwares* (compilação e JIT, apresentados nas Seções 3.1.1 e 3.1.2), a vetorização depende de programas específicos, exigindo certa familiaridade entre o desenvolvimento do *software* de vetorização (ou *profiling*) e o *hardware* alvo a ser utilizado. Em nosso mecanismo removemos essa dependência, assim simplificando a adoção e tornando o mecanismo opaco para qualquer utilizador.

Dentre as alternativas focadas em *hardware*, apresentadas na Seção 3.1.3, nenhuma pôde ser diretamente adaptada para o uso de instruções vetoriais IMA. Apesar disso, todas oferecem ideias interessantes e promissoras para a otimização da conversão proposta nesta dissertação.

3.2 NEAR-DATA PROCESSING (NDP)

Seguindo a nomenclatura adotada por Santos et al. (2021), técnicas para processamento próximo à memória, genericamente denominadas NDP, podem ser divididas em *Near-Memory Accelerator* (NMA), onde unidades de processamento são adicionadas fisicamente próximas ao armazenamento, *In-Memory Accelerator* (IMA), onde unidades de processamento são adicionadas no mesmo módulo do armazenamento, e *Near-Cell Accelerator* (NCA), onde o processamento de dados ocorre dentro dos circuitos de armazenamento.

Todas as formas de processamento NDP dependem de *Application Programming Interfaces* (APIs) especializadas, compiladores capazes de converter partes do programa para instruções NDP ou *hardwares* e *softwares* dedicados a essa conversão (Singh et al., 2019). Trabalhos utilizando cada uma dessas técnicas de conversão são apresentados nas seções seguintes.

3.2.1 Conversão manual

Uma prática comum por sua simplicidade, em trabalhos que utilizam NDP, é a criação de bibliotecas para o acesso às funcionalidades de processamento implementadas. Apesar de mais simples, essa alternativa geralmente exige mais conhecimentos do programador, o que dificulta seu uso. Algumas técnicas desse tipo são apresentadas a seguir.

Guo et al. (2015) propõe o uso do processamento em memória para acelerar operações realizadas em bibliotecas de alta performance, como a *Basic Linear Algebra Subprogram* (BLAS) (bla, 2002). Para isso, adicionam uma nova camada ao *Hybrid Memory Cube* (HMC) contendo múltiplos grupos de aceleradores, capazes de realizar operações comuns nessas bibliotecas, como a multiplicação de uma matriz por um vetor, além de operações mais complexas com o encadeamento desses aceleradores. Essas operações de memória são acessíveis por meio de um compilador, capaz de converter as chamadas a procedimentos e alocações de memória usadas com esses bibliotecas por versões equivalentes usando os aceleradores. Diferente desse trabalho, nosso mecanismo realiza essa conversão durante a execução, para todo o código da aplicação. Apesar disso, o uso de APIs e *softwares* de conversão proposto por Guo et al. (2015) poderia ser adaptado em um trabalho futuro como uma maneira de aumentar as oportunidades de conversão encontradas pelo SAPIVe.

Ahn et al. (2015a) propõe a utilização de dispositivos IMA como aceleradores para o processamento de grafos. Para isso, acrescenta processadores simples e prefetchers em cada vault do HMC e estabelece um protocolo de comunicação entre os vaults. Esses aceleradores são programados com uma biblioteca específica, que permite o envio de operações para a memória. Diferentemente desse trabalho, nosso mecanismo converte essas instruções para o processamento em memória durante a execução, sem a necessidade de alterações no código da aplicação. Além disso, Ahn et al. (2015a) utiliza instruções escalares em memória, algo não realizado nesta dissertação, porém que poderia ser implementado em trabalhos futuros para aumentar nossa gama de oportunidades de conversão.

Ahn et al. (2015b) estende a *Instruction Set Architecture* (ISA) com instruções denominadas *PIM-enabled* que são inseridas no código pelo programador. Essas instruções são operações simples (eg. um add atômico) que podem ser executadas tanto pela memória quanto pelo processador. Durante a execução da aplicação, o *hardware* decide, com base no local em que se encontram os dados dos operandos da instrução, onde ela será executada. Apesar de ter sido aplicado para operações escalares, essa decisão de onde executar cada parte da aplicação de acordo com o posicionamento de seus dados poderia ser um incremento interessante para nosso mecanismo em trabalhos futuros. Dessa forma, caso os dados necessários para a execução da

maior parte das iterações de determinado laço já estejam presentes em cache, nosso mecanismo pode aproveitar o processamento mais rápido do processador, caso contrário, o processamento em memória seria utilizado.

Nai et al. (2017) otimiza o processamento de grafos transferindo operações atômicas para a memória. Sua técnica consiste de uma biblioteca para manipulação de grafos especializada e uma pequena extensão no processador. Essa extensão converte instruções atômicas simples do programa para instruções HMC 2.0, caso acessem dados de uma região de memória especial do sistema. Assim, reduzindo a penalidade de seus acessos irregulares aos dados. Já nesta dissertação, além de evitar o uso de bibliotecas específicas para o processamento em memória, também utilizamos instruções vetoriais em memória. Apesar disso, o uso de instruções escalares em memória proposto por Nai et al. (2017) poderia ser utilizado em trabalhos futuros na tentativa de acelerar determinadas aplicações que não puderam ser vetorizadas por nosso mecanismo.

Han et al. (2017) propõe uma arquitetura de processamento em memórias *Resistive Random-Access Memory* (ReRAM) para acelerar algoritmos que percorrem grafos. Essa arquitetura tenta minimizar a movimentação de dados com um esquema de compressão e mapeamento eficiente de partes do grafo para a memória. Além disso, é capaz de realizar expansões paralelamente em múltiplos bancos de memória durante suas buscas. Diferentemente de nosso mecanismo, essa técnica é acessível por meio de uma API específica, sendo limitada a problemas sobre grafos. Por conta disso, consegue ser muito eficiente nesses problemas, porém acaba não abordando programas mais genéricos. Apesar disso, o uso de bibliotecas especializadas auxiliares poderia contribuir para aumentar a capacidade de conversão de nosso mecanismo em trabalhos futuros, mantendo sua simplicidade de padrões.

Cordeiro et al. (2017) propõe uma metodologia para a simulação de execução de códigos em arquiteturas NDP ainda não existentes. Essa técnica consiste no uso de *intrinsics*, contendo código x86, que representam funções de processamento NDP. O código contendo essas *intrinsics* é compilado para a arquitetura real (x86) e, por meio do instrumentador binário dinâmico Pin da Intel (Intel, 2018) e do gerador de traços do *Simulator of Non-Uniform Cache Architecture* (SiNUCA) (Alves et al., 2015), é gerado um traço de execução correspondente para simular a execução na nova arquitetura. Nesse traço, as *intrinsics* x86 são substituídas por operações correspondentes dentro da *smart memory*, o que é interpretado de acordo pelo simulador. Tal técnica poderia ser facilmente adaptada para fornecer bibliotecas *intrinsics* para a programação de novas arquiteturas que utilizem unidades funcionais NDP. Enquanto esse trabalho foca no uso de *intrinsic routines* para o uso do processamento em memória, nossa proposta visa a conversão automática de instruções de forma transparente aos desenvolvedores, simplificando seu uso. Apesar disso, permitir o uso do processamento em memória tanto por nossa extensão como por *intrinsics*, é um possível trabalho futuro.

Lee et al. (2018) anexa aceleradores à camada lógica do HMC para aumentar o desempenho de buscas por similaridade. Esses aceleradores utilizam instruções específicas que estão disponíveis a partir de uma biblioteca. Já em nosso mecanismo, buscamos evitar qualquer alteração no código da aplicação, seja pelo usuário ou por um compilador. Além disso, visamos sistemas de processamento em memória mais genéricos, que poderiam ser utilizados por qualquer aplicação com operações elemento a elemento sobre vetores. Enquanto Lee et al. (2018) propõe uma técnica de processamento em memória focada em acelerar apenas buscas por similaridade.

Zhang et al. (2018) propõe um processador criptográfico reconfigurável NDP em dispositivos *Internet of Things* (IoT). Isso permite a adoção de técnicas criptográficas mesmo sob as restrições existentes nesses dispositivos. O processador criptográfico é capaz de executar operações IMA e *Near-Cell Accelerator* (NCA), tornando seu processamento eficiente. A utilização fica a cargo do programador, por meio do envio de comandos para um decodificador

mapeado em memória. Diferentemente de nossa proposta, Zhang et al. (2018) visa sistemas com altas restrições energéticas, de forma que suas operações em memória são especializadas apenas para as mais comuns em operações criptográficas. Além disso, o código fonte dessas operações precisa ser alterado manualmente para a inserção das instruções de memória, porém essas alterações poderiam ser realizadas por nossa extensão, em um trabalho futuro.

Liu et al. (2018) tenta simplificar o processo de implementação de redes neurais com o processamento em memória, utilizando dispositivos NDP com funcionalidades fixas (aceleradores) e programáveis (propósito geral). Para isso, estende o modelo de programação OpenCL e utiliza um sistema durante a execução. O mesmo código em OpenCL é compilado para CPU, NDPs programáveis e NDPs fixos. Durante a execução, é feito um *profiling* da primeira iteração da rede neural que é usado para selecionar a versão mais eficiente de cada parte do código. Essa versão é executada nas próximas iterações da rede, otimizando o uso do sistema. Apesar de nosso trabalho não depender do auxílio de compiladores, a técnica de *profiling* utilizada por Liu et al. (2018) poderia servir como base para melhorias na seleção de instruções para o processamento em memória de nosso mecanismo.

Lee et al. (2019) propõe uma arquitetura de NDP sobre o *High-Bandwidth Memory* de segunda geração (HBM2), adicionando elementos de lógica dentro de cada um dos bancos desse dispositivo. Além disso, apresenta uma visão geral de como os componentes de um sistema, desde a aplicação, passando pelo sistema operacional, controladores de memória e o dispositivo NDP podem funcionar juntos para a obtenção de alta performance com baixos custos de implementação. Essa arquitetura trata instruções NDP como comandos convencionais à memória, sendo capaz de responder a requisições de leitura e escrita enquanto realiza processamento. Para seu uso é fornecida uma biblioteca, deixando a utilização do NDP para o programador. As principais diferenças desse trabalho com esta dissertação estão na localização de suas unidades de processamento em memória e no uso do IMA, que fazemos de forma automatizada.

Li et al. (2019) propõe o uso de *crossbar arrays* de *memristors* (usando ReRAMs) durante as etapas de treinamento e inferência em redes recorrentes *Long Short-Term Memory* (LSTM). Com isso, as multiplicações de matrizes da rede são efetuadas dentro da própria memória, reduzindo o tráfego de dados entre a memória e a unidade de computação principal utilizada (CPU ou *Graphics Processing Unit* (GPU)). Para o uso dessa técnica NCA, foi construída uma biblioteca em MATLAB, permitindo a criação de diferentes configurações para LSTMs totalmente conexas. Diferentemente de Li et al. (2019), nosso mecanismo visa otimizar operações de qualquer aplicação. Além disso, nosso mecanismo foi projetado para a execução de instruções na camada lógica de memórias DRAM 3D, ao invés de *crossbar arrays* de *memristors*. Porém a avaliação do SAPIVe sobre redes neurais poderia ser analisada em trabalhos futuros.

3.2.2 Conversão automática

Apesar da simplicidade na disponibilização de APIs, a exigência de conhecimentos específicos acaba dificultando seu uso. Assim, diversos trabalhos focando técnicas NDP independentes de desenvolvedores, apenas por *hardware*, *software* ou pelo compilador, também foram propostos e os mais relevantes encontrados são apresentados a seguir. Além desses, também são incluídos trabalhos onde a dependência de APIs é opcional, sendo dispensável para o uso das técnicas NDP.

Akin et al. (2015) utiliza processamento IMA para reorganizar dados dentro de memórias 3D. Por meio de alterações no *memory controller*, esse componente passa a identificar o comportamento dos acessos à memória, reorganizando o endereço físico dos dados para aproveitar melhor o paralelismo fornecido pelas memórias 3D (*intra/inter vault* e a localidade do *row buffer*). Além disso, também fornece uma *software stack* que permite o envio de operações

de reorganização especificadas por programadores. Diferentemente de nosso trabalho que envia operações para a memória, esse visa reorganizar os dados presentes na memória para permitir um uso mais eficiente de seu paralelismo. Essa ideia poderia ser adaptada como uma extensão de nosso mecanismo, em um trabalho futuro, acelerando mesmo operações que não puderem ser levadas para a memória.

Hadidi et al. (2017) propõe a técnica *Compiler-Assisted technique and decision model for enabling InstRuction-level Offloading* (CAIRO), que identifica, durante a compilação, instruções atômicas e conjuntos de instruções de *load*, operação e *store*, que poderiam ser substituídos por instruções atômicas do dispositivo IMA HMC. Apenas instruções de CPU e GPU, em algoritmos de grafos, são selecionadas nesta primeira etapa. Após a identificação de candidatos, a técnica avalia se a substituição de cada um deles levaria a ganhos ou perdas de desempenho para a aplicação. Essa avaliação é feita utilizando um método de *profiling*, que obtém a taxa estimada de *misses* na *Last Level Cache* (LLC), a densidade de candidatos próximos, a densidade de instruções atômicas próximas e a largura de banda média durante toda a execução, para cada candidato. Os dados obtidos pelo *profiling* são analisados com base em parâmetros do sistema, obtidos por meio de *micro-bechmarks*, e é estimada a aceleração obtida pela substituição de cada candidato. Aqueles com uma conversão vantajosa para a aplicação são convertidos para instruções atômicas HMC. Apesar de sua similaridade com nosso mecanismo, devido aos padrões convertidos por ambos, essa técnica é aplicada durante a compilação, utilizando dados obtidos por um mecanismo de *profiling*. Além disso, diferentemente do SAPIVe, CAIRO converte instruções escalares da CPU para instruções escalares IMA, de forma que cada instrução ou conjunto de instruções da CPU é equivalente a uma instrução de memória. Por outro lado, realizamos a conversão de instruções AVX de 256 e 512 bits para instruções em memória de maior tamanho, exigindo diversas iterações das instruções AVX para cada instrução IMA. Apesar dessas diferenças, as métricas utilizadas pelo CAIRO para a seleção de candidatos promissores poderiam ser adaptadas para nosso mecanismo. Além disso, a conversão de instruções vetoriais da CPU para instruções VIMA de mesmo tamanho, apesar de ser menos eficiente, poderia simplificar as conversões de nosso mecanismo em trabalhos futuros. Essa conversão pode ser guiada pelas mesmas métricas de densidade de candidatos e taxa de *misses* na LLC, reduzindo conversões prejudiciais.

Ahmed et al. (2019) propõe um compilador capaz de vetorizar programas para NDP, sem exigir anotações no código ou instruções especiais. Para isso, o *Processing-In-Memory cOmpiler* (PRIMO) vetoriza aplicações tanto com instruções vetoriais da CPU, quanto com instruções vetoriais processadas pela memória, utilizando o maior tamanho de instrução vetorial possível. Como as instruções vetoriais da memória, utilizadas nesse trabalho, processam dados maiores que as presentes na CPU, permitindo o uso de operações de diversos tamanhos, porções de código vetorizadas para esses grandes tamanhos são executadas em memória. Além disso, esse trabalho utiliza como dispositivo IMA o *Reconfigurable Vector Unit* (RVU) (Santos et al., 2017), com uma unidade de processamento vetorial diferente em cada *vault* da memória 3D, contendo seus próprios registradores e a capacidade de operar independentemente. Assim, o PRIMO também seleciona a unidade IMA mais adequada para o processamento de seus dados, de acordo com as dependências entre instruções enviadas para a memória. Com isso, tenta reduzir o número de dependências entre instruções de dispositivos IMA distintos, ao mesmo tempo em que maximiza o paralelismo desses dispositivos. As principais diferenças com nosso trabalho são: o momento onde as vetorizações são realizadas, permitindo um maior conhecimento sobre o comportamento do algoritmo como um todo por parte do PRIMO, o que simplifica suas conversões; sua capacidade de efetivamente substituir instruções no código da aplicação, reduzindo o número de instruções processadas e; o uso de diversos dispositivos IMA com capacidade de operar paralelamente em cada *vault* da memória 3D e sobre diferentes tamanhos de

instrução de memória em uma execução. Diferentemente desse trabalho, utilizamos o dispositivo VIMA, visto que, o uso de registradores pelo RVU pode inviabilizar o uso ideal de *multi-threading* em IMA para trabalhos futuros. Além disso, o PRIMO não poderia ser adaptado de forma direta para VIMA por ela não usar registradores, apenas endereçamento direto na memória. Apesar disso, o uso de preditores do número de iterações nos laços de uma aplicação poderia aproximar nosso mecanismo das conversões realizadas pelo PRIMO, permitindo a seleção de diferentes tamanhos de instruções IMA, de acordo com o número de iterações previsto para cada laço.

Long et al. (2019) utiliza um sistema em *software* durante a execução capaz de carregar a descrição de *Deep Neural Networks* (DNNs), assim como modelos pré-treinados, e converter esses modelos em alto nível para a execução em memória. O trabalho propõe uma arquitetura IMA baseada em *Ferroelectric Field Effect Transistor* (FeFET), a qual apresenta menores correntes de fuga e maior densidade computacional em relação a tecnologias utilizando *Dynamic Random Access Memories* (DRAMs) e *Static Random Access Memories* (SRAMs) convencionais. Já nosso trabalho realiza vetorizações em programas genéricos para instruções em memórias 3D. Apesar disso, o uso de uma arquitetura IMA baseada em FeFET por nossa extensão poderia ser avaliado em um trabalho futuro.

Wang et al. (2019) fornece um framework que escalona tarefas presentes em *Convolutional Neural Networks* (CNNs) para o processamento em memória. Nele, os dados intermediários e das camadas convolucionais são alocados segundo um modelo de programação dinâmica, enquanto as tarefas da rede são escalonadas considerando a utilização do sistema e possibilidades de execução concorrente de convoluções. Com isso o paralelismo da arquitetura é maximizado, minimizando seu tráfego de dados. Suas principais diferenças em relação à nossa proposta são ser restrito a CNNs e ser realizado em *software*, enquanto adotamos conversões genéricas em *hardware*.

A Tabela 3.1 apresenta um resumo dessas técnicas com os mecanismos utilizados durante a conversão de suas instruções para o processamento em dispositivos NDP.

Tabela 3.1: Responsabilidade de conversão para o processamento em memória em cada trabalho relacionado. Fonte: Autor.

Trabalhos	**API**	**Compilação**	***Software* JIT**	***Hardware***
(Guo et al., 2015)	•		•	
(Ahn et al., 2015a)	•			
(Ahn et al., 2015b)	•			•
(Nai et al., 2017)	•			•
(Han et al., 2017)	•			
(Cordeiro et al., 2017)	•			
(Lee et al., 2018)	•			
(Zhang et al., 2018)	•			
(Liu et al., 2018)	•		•	
(Lee et al., 2019)	•			
(Li et al., 2019)	•			
(Akin et al., 2015)	•			•
(Hadidi et al., 2017)		•		
(Ahmed et al., 2019)		•		
(Long et al., 2019)			•	
(Wang et al., 2019)			•	
Esta dissertação				•

Diferentemente de todos os trabalhos apresentados, o mecanismo proposto nesta dissertação, denominado SAPIVe, visa fazer toda a tradução para o processamento em memória com *hardware*, reduzindo a dependência de *software* ou treinamentos de pessoal.

4 SIMPLE AVX TO PIM VECTORIZER (SAPIVE)

A contribuição desta dissertação é uma extensão de *hardware* para tradução binária dinâmica capaz de converter, durante a execução, múltiplas instruções *Advanced Vector Extension* (AVX), de 256 e 512 bits, para instruções *Vector-In-Memory Architecture* (VIMA) de 256 B a 8 KiB, denominada Simple AVX to PIM Vectorizer (SAPIVe). O mecanismo proposto considera um sistema com um único núcleo, executando uma única *thread*, com instruções VIMA geradas apenas por nosso mecanismo. Um projeto visando um sistema mais robusto exigiria um tratamento mais completo em relação a sincronizações e coerência entre componentes, o que pode ser abordado em trabalhos futuros.

Como as instruções VIMA executam em memória, esperamos reduzir os custos relacionados a energia e tempo de execução originados da movimentação de dados entre a memória principal e a *Central Processing Unit* (CPU). Além disso, ganhos secundários podem ser obtidos por nossas conversões. Esses ganhos podem ser explicados pela execução em memória que evita que dados sejam armazenados nas *caches* da CPU, reduzindo o *cache pollution*, ou seja, a remoção de dados úteis para instalação de dados *dead-on-arrival* (que serão utilizados apenas uma única vez).

Essa redução do *cache pollution* aumenta a porcentagem de dados encontrados nas *caches*, acelerando a execução de instruções dependentes deles. Porém a redução de dados nas *caches* não é sempre benéfica, visto que futuras instruções que acessem os resultados não instalados nas *caches* serão afetadas por toda a latência da hierarquia de memória até a *Dynamic Random Access Memory* (DRAM).

Neste contexto de processamento em memória, o mecanismo SAPIVe realiza a conversão de instruções AVX-256 e AVX-512 para instruções VIMA, durante a execução da aplicação. Como apresentado na Seção 2.4, a VIMA não possui um banco de registradores. Por conta disso, cada uma de suas instruções carrega um ou dois dados da memória, opcionalmente realiza uma operação, e salva seu resultado em um endereço da memória. Assim, as instruções VIMA podem ser divididas em três categorias, ilustradas na Figura 4.1: (i) cópias de memória, realizando apenas um *load* e um *store*; (ii) operações sobre um único vetor, realizando um *load*, uma operação, como uma inversão bit a bit ou operação com imediato, e um *store* e; (iii) operações entre dois vetores, elemento a elemento, que realizam dois *loads*, uma operação e um *store*. Como o gargalo das instruções convertidas para a VIMA é o acesso à memória, instruções das categorias (i) e (ii) apresentam resultados similares com nosso mecanismo. Isso, porque ambas acessam a mesma quantidade de dados da memória, diferentemente das instruções da categoria (iii). Assim, por simplicidade, nos resultados desta dissertação, consideramos apenas as categorias (i) e (iii) em nossas conversões. Além disso, visto que cada instrução VIMA opera sobre um conjunto de dados de tamanho maior que o operado por instruções da CPU, restringimos nossas conversões a laços, de forma que diversas iterações de um mesmo laço possam compor uma instrução VIMA.

Após identificar um laço contendo instruções AVX que podem ser convertidas para uma instrução VIMA, o SAPIVe gera uma instrução VIMA correspondente a iterações futuras desse laço com instruções AVX. Porém, essa conversão pode se provar incorreta em algumas situações, como quando o número de iterações futuras não for suficiente para sua conclusão, o que torna nossas conversões especulativas. Para a garantir a corretude da execução mesmo perante a essas falhas, mantemos as instruções sendo convertidas no *Reorder Buffer* (ROB) durante todo o processo de conversão. Dessa forma, caso ocorra uma falha, podemos descartar o resultado gerado pela VIMA e executar em CPU as instruções já presentes no ROB.

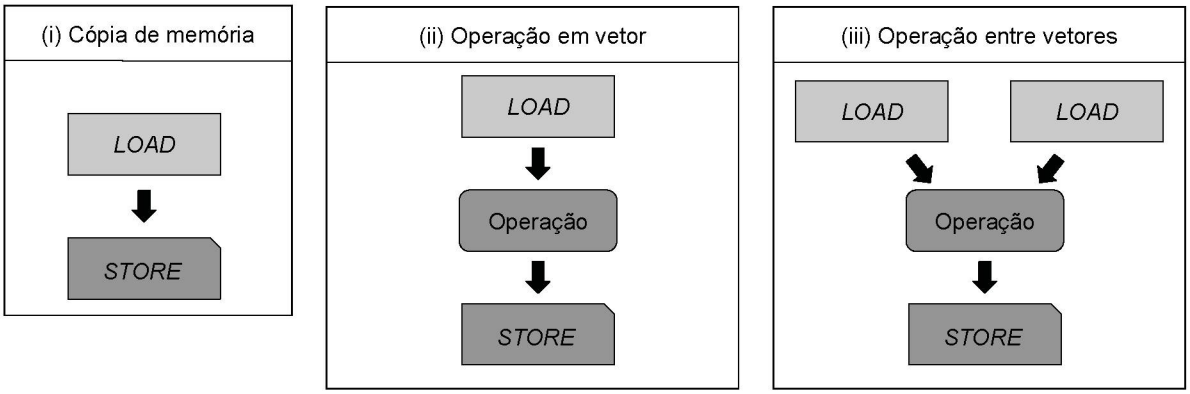


Figura 4.1: Conjuntos de cópia de memória, operação em um vetor e operação entre dois vetores convertidos pelo SAPIVe para instruções *In-Memory Accelerator* (IMA) da VIMA. Fonte: Autor.

Devido a esse acúmulo de instruções, optamos por converter apenas instruções AVX-256 e AVX-512, visto que o maior tamanho de seus dados exige um número menor de iterações para compor cada instrução VIMA. Dessa maneira, cada conversão acumula menos instruções no ROB, permitindo um maior número de conversões simultâneas e conversões para instruções VIMA maiores.

Alternativamente, nosso mecanismo poderia armazenar apenas o *Program Counter* (PC) da primeira instrução de cada conversão, realizando um *flush* no *pipeline* e restaurando o PC durante falhas de conversão. Porém, essa abordagem tornaria a recuperação mais lenta após cada conversão falha. A avaliação dessa alternativa é considerada para trabalhos futuros.

```
// Soma vetorial usando AVX-512
for (int i = 0; i <= v_size-16; i += 16) {
    V_a = _mm512_load_epi32 (&data_a[i]);
    V_b = _mm512_load_epi32 (&data_b[i]);
    V_c = _mm512_add_epi32 (V_a, V_b);
    _mm512_store_epi32 (&data_c[i], V_c);
}
```

(a) Soma vetorial em AVX-512.

```
int i;
// Conversão para VIMA-256 B
for (i = 0; i <= v_size - (16*4); i += 16 * 4)
{
_vim256_iadds(&data_a[i],&data_b[i],&data_c[i]);
}

// Final do laço usando AVX-512
for (; i <= v_size-16; i += 16) {
    V_a = _mm512_load_epi32 (&V_a[i]);
    V_b = _mm512_load_epi32 (&V_b[i]);
    V_c = _mm512_add_epi32 (V_a, V_b);
    _mm512_store_epi32 (&data_c[i], V_c);
}
```

(b) Vetorização da soma vetorial de AVX-512 para VIMA-256 B.

Figura 4.2: Equivalência entre códigos AVX-512 e VIMA-256 B. Ambos realizam uma soma, elemento a elemento, entre os vetores de inteiros de 4 B, data_a e data_b, armazendo o resultado em data_c. Em (a) é apresentado o código em AVX-512, enquanto em (b) temos o código vetorizado de AVX-512 para VIMA-256 B. Visto que cada iteração com a VIMA-256 B é equivalente a 4 iterações AVX-512 (256/64 = 4), o número de iterações entre (a) e (b) é alterado. Além disso, em (b), um laço para o cálculo do resto foi adicionado, para caso o número de iterações AVX-512 em (a) não seja múltiplo do tamanho dos dados operados pelas instruções VIMA (256 B). Fonte: Autor.

Uma ideia da equivalência entre instruções AVX em laços e instruções VIMA, que permite as conversões realizadas por nosso mecanismo, é apresentada na Figura 4.2. Nessa figura, o Código (a) apresenta um laço contendo instruções AVX-512 que somam, elemento a elemento, dois vetores, data_a e data_b, armazenando o resultado em data_c. Já o Código (b) apresenta o

mesmo laço, porém substituindo algumas das iterações com instruções AVX-512 por instruções VIMA-256 B. A *instrinsic* VIMA utilizada nessa substituição é a _vim64_iadds, que realiza uma soma, elemento a elemento, entre os dados presentes nos endereços de memória passados por seus dois primeiros operandos, e armazena o resultado no endereço de memória passado pelo terceiro operando. Visto que cada instrução VIMA-256 B opera sobre dados de 256 B, e cada instrução AVX-512 opera sobre 64 B, temos que uma iteração com instruções da VIMA-256 B equivale a 4 iterações AVX-512, exigindo uma alteração no passo do laço e, consequentemente, em seu número de iterações. Além disso, caso o tamanho dos vetores operados pelas instruções AVX-512 não seja múltiplo de 256 B, um laço secundário precisará terminar a execução das operações sem o uso da VIMA-256 B.

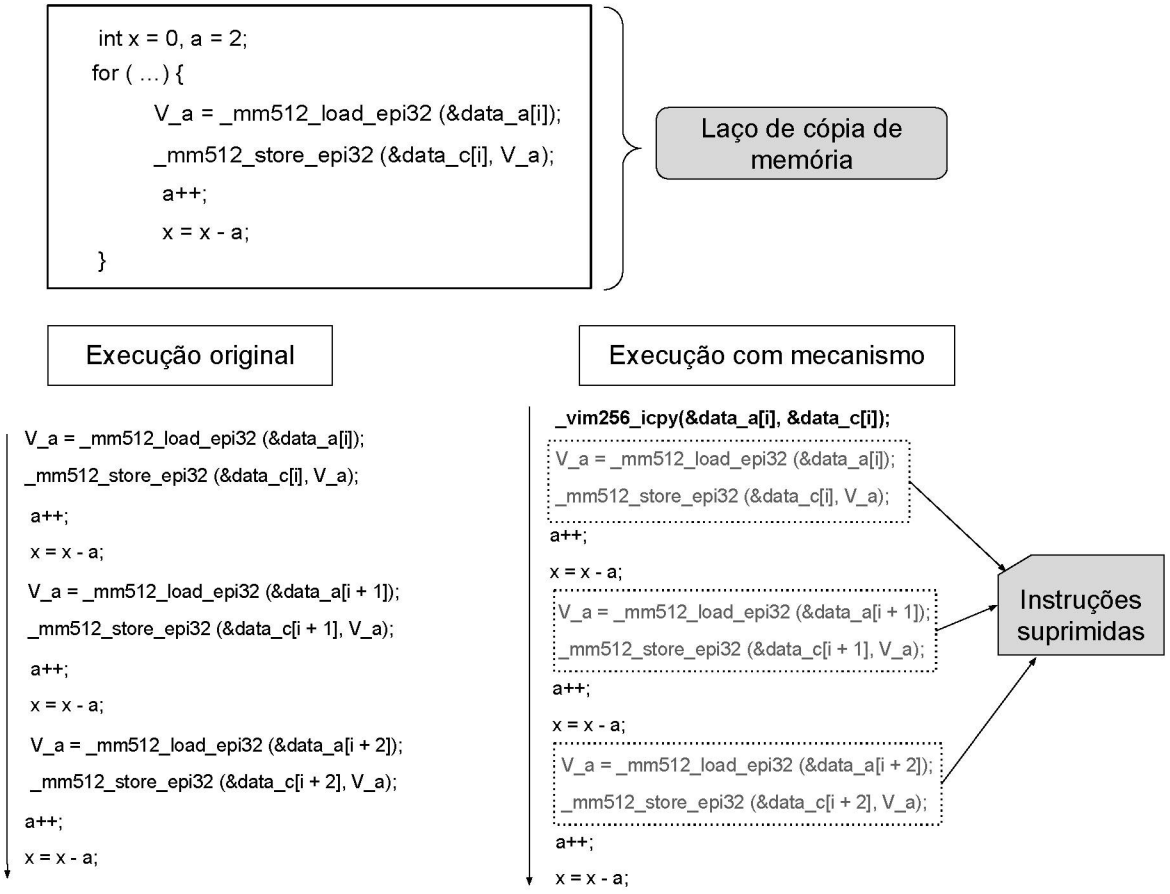


Figura 4.3: Exemplo de conversão realizada por nosso mecanismo sobre um laço de cópia de memória. As conversões do SAPIVe incluem uma instrução VIMA (_vim256_icpy) correspondente a algumas iterações do laço AVX no *pipeline*. Ao mesmo tempo, as instruções AVX substituídas têm sua execução suprimida, até que o mecanismo confirme se a conversão é realmente válida ou não. Fonte: Autor.

Esse mesmo processo de conversão é replicado por nosso mecanismo, porém sem alterar o número de iterações do laço inicial, apenas unindo instruções AVX de diferentes iterações em uma única instrução VIMA, como ilustrado pela Figura 4.3. Como o tamanho dos dados processados por instruções AVX é menor que o processado por instruções VIMA, podemos dizer que a conversão de instruções AVX para VIMA é um processo de vetorização.

As vetorizações em *hardware*, de nosso mecanismo, seguem as seguintes etapas de conversão:

- Identificação: cada instrução passando pelo estágio *rename* do *pipeline* é avaliada em busca de conjuntos de instruções, contidos em laços, realizando cópia de memória, operação em um vetor ou operação entre vetores, elemento a elemento.

- Análise: são obtidas informações sobre a execução do conjunto encontrado, como os endereços de acesso à memória, necessárias para a criação de uma instrução VIMA equivalente ao conjunto.

- Validação: uma instrução VIMA, correspondente ao conjunto, inicia uma execução especulativa, ao mesmo tempo em que as iterações do laço substituídas por ela são analisadas.

- Confirmação: se a execução da instrução VIMA for bem sucedida e as verificações da CPU comprovarem a equivalência entre as iterações do laço e essa instrução, o resultado da VIMA é salvo em definitivo na memória e as instruções convertidas são removidas do ROB, completando a conversão.

- Invalidação: se a execução da instrução VIMA falhar, ou a conversão do laço se provar impossível, o resultado da VIMA é descartado e as instruções que seriam convertidas são liberadas para execução na CPU.

Cada conversão é vinculada a um identificador único, incrementado para cada nova conversão. Isso permite a identificação de instruções vinculadas a cada conversão e de instruções pertencentes a conversões mais recentes, para a execução de invalidações em cascata.

Nosso mecanismo altera apenas a execução das instruções pertencentes aos conjuntos convertidos, as demais instruções presentes no ROB são executadas de maneira usual. As alterações necessárias para o uso do SAPIVe, seu funcionamento e limitações são descritos nas seções seguintes.

4.1 ALTERAÇÕES DA VECTOR-IN-MEMORY ARCHITECTURE (VIMA)

Por realizarmos conversões que podem precisar ser desfeitas (especulativas), nosso mecanismo depende da possibilidade de armazenar resultados temporários das operações VIMA. Esses resultados precisam ser descartados ou salvos em definitivo, de acordo com o resultado da conversão. Para permitir esse comportamento, adaptamos a *cache* existente na VIMA, aproximando-a de uma *cache* transacional.

Para isso, separamos essa *cache* em duas partes. A primeira parte é utilizada para o armazenamento de dados carregados para a leitura, durante a obtenção dos operandos da VIMA. Já o espaço da segunda parte é substituído por um *buffer* de escritas, utilizado para armazenar o resultado de cada operação VIMA.

Consideramos que o tempo de acesso, escritas e leituras a esses componentes é o mesmo que o tempo original de acesso à *cache* da VIMA. Além disso, a busca por operandos ocorre simultaneamente em ambos espaços e, caso um dado sendo escrito esteja presente na *cache* de leituras, ele é realocado para o *buffer* de escritas. Com isso, evitamos que outra operação obtenha esse dado antes que a conversão que o gera seja confirmada.

Garantimos as propriedades transacionais no *buffer* de escritas (atomicidade e isolamento), com 1 bit adicional de controle em cada uma de suas entradas Sempre que uma instrução VIMA escrever em uma entrada, ela tem seu bit de controle definido para 1. Após o sucesso da conversão AVX para VIMA ser confirmado, esse bit é zerado. Dessa maneira, dados com o bit de controle ativo representam dados temporários, que não podem ser lidos ou escritos por

outras instruções VIMA e nem mesmo escritos em memória, sendo mantidos dentro do *buffer*. Caso ocorra uma tentativa de acesso a um dado temporário, o que pode ser causado por uma instrução executada após o fim do laço, antes da conversão ser concluída, a instrução VIMA que escreveu esse dado temporário é invalidada e seus resultados são eliminados. Além disso, a CPU é informada do problema, invalidando a conversão que gerou a instrução VIMA e todas as próximas conversões.

Visto que a VIMA executa instruções de maneira sequencial, e em ordem, em relação às demais instruções de memória da aplicação, podemos garantir a atomicidade de suas operações. Como impedimos a leitura e sobrescrita de linhas com dados temporários, também garantimos a propriedade de isolamento para o *buffer* de escritas, permitindo que execute operações de maneira transacional.

Para evitar que duas conversões em sequência tentem escrever no mesmo dado do *buffer* de escritas, gerando invalidações, nosso mecanismo alinha as escritas de suas conversões com o início das linhas de *cache* VIMA, sendo que o tamanho dessas linhas é o mesmo da operação VIMA realizada. Caso isso não seja possível, por simplificação, a conversão é invalidada.

4.2 ETAPAS DA CONVERSÃO

Esta seção descreve as etapas envolvidas na conversão de conjuntos de instruções AVX-256 e AVX-512, presentes em laços, para instruções IMA da VIMA.

4.2.1 Identificação de conjuntos de instruções para conversão

A primeira etapa do processo de conversão consiste na busca por laços contendo conjuntos de instruções que podem ser convertidos para instruções VIMA. Esses conjuntos de instruções podem ser formados de três maneiras, ilustradas na Figura 4.1: (i) uma operação de *load* com seu resultado armazenado em memória por um *store*; (ii) uma operação de *load* com seu resultado utilizado por uma operação aritmética/lógica que tem o resultado salvo por um *store* (iii) duas operações de *load* com seus resultados utilizados em uma operação aritmética/lógica que tem seu resultado salvo por um *store*.

Para identificar esses conjuntos, o SAPIVe avalia cada operação que passa pelo estágio *rename* do *pipeline*. Esse estágio foi escolhido por ser o primeiro a possuir acesso às instruções decodificadas e aos registradores físicos utilizados por cada operação, essenciais para as avaliações realizadas pelo nosso mecanismo. Além disso, ele possibilita um menor atraso nas decisões do mecanismo, se comparado aos estágios seguintes do *pipeline*.

Para identificar dependências entre instruções, em busca de conjuntos que podem ser convertidos para instruções VIMA, o SAPIVe utiliza um vetor com 2 bits de controle em cada uma de suas entradas, denominado **vetor de dependências**. Cada posição desse vetor corresponde a um registrador físico, sendo atualizada sempre que uma instrução renomeada escreve nesse registrador, e assumindo os seguintes valores:

0. Valor inicial das entradas;
1. Um *load* AVX, de 256 ou 512 bits escreveu no registrador físico associado a esta entrada;
2. Uma *operação* AVX, sobre os registradores escritos por um ou dois *loads* AVX, escreveu no registrador físico associado a esta posição;
3. Outra instrução escreveu no registrador físico associado a esta posição.

Além disso, 3 bits de controle compõe o **construtor**, o qual é utilizado para controlar uma *Finite State Machine* (FSM) (apresentada na Seção 4.2.2) identificando que partes de um conjunto de instruções para tradução foram encontradas. Instruções independentes, que não influenciam na tradução, entre as instruções desse conjunto não alteram o **construtor**.

A construção de conjuntos é iniciada quando um *load* AVX, de 256 ou 512 bits, é renomeado. Nisso, a informação sobre o tamanho dessa operação (versão AVX) é salva e uma potencial conversão se inicia. Para essa nova conversão, um identificador único é criado. Esse identificador consiste do valor presente em um contador com *overflow*, incrementado a cada conversão bem sucedida ou falha. Esse valor é vinculado a cada instrução de um conjunto e à sua instrução VIMA equivalente. Assim, o tamanho desse contador deve ser suficiente para identificar unicamente cada conversão paralela realizada pelo SAPIVe.

A posição do vetor de dependências, correspondente ao registrador físico escrito pelo *load* renomeado recebe o valor 1 (Indicando um possível início de conjunto). Por fim, o construtor é incrementado para 1, indicando que um primeiro *load* AVX foi encontrado, como apresentado na Figura 4.4.

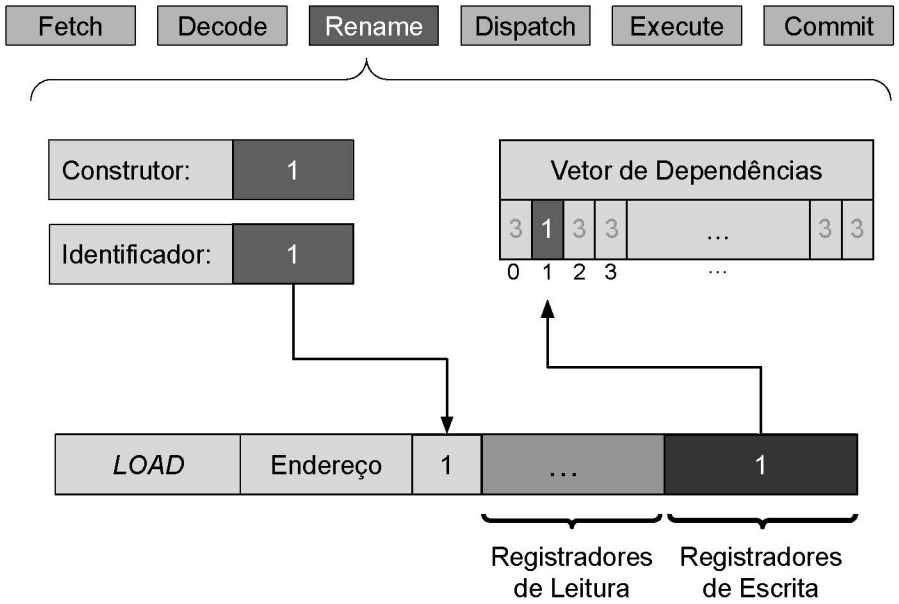


Figura 4.4: Estado das estruturas do mecanismo após encontrar o primeiro *load* de um conjunto. Para essa a nova conversão, o contador de identificadores é incrementado, criando um novo identificador único, que é associado ao *load*. Além disso, o vetor de dependências recebe 1 na posição correspondente ao registrador físico que será escrito pelo *load* e o construtor recebe 1, indicando que o primeiro *load* de um conjunto foi encontrado. Fonte: Autor.

Após o primeiro *load*, pelo construtor estar em 1, o mecanismo passa a buscar por operações que leiam o registrador que armazenou o valor buscado na memória por esse *load* (registrador de escrita), indicado pelo vetor de dependências, além de outras instruções de *load* AVX. Caso ele encontre um *store*, salvando os dados carregados pelo *load*, encontramos um conjunto de cópia de memória, como exemplificado na Figura 4.5. Assim, o construtor é atualizado para 4, indicando o fim do conjunto, e o identificador único da conversão é vinculado ao *store*, como apresentado na Figura 4.6.

Por outro lado, se encontrar uma operação, o mecanismo assume que encontrará o conjunto de operação sobre um vetor, enquanto que se encontrar um segundo *load*, ele assume uma operação entre dois vetores, o que é exemplificado na Figura 4.7. No caso de uma operação entre dois vetores, ao encontrar o segundo *load*, construtor é atualizado para 2, indicando que a próxima instrução encontrada será uma operação aritmética/lógica, e o vetor de dependências

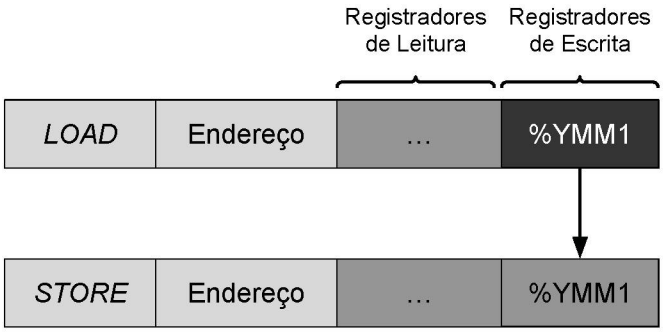


Figura 4.5: Vínculo de instruções no conjunto de cópia de memória buscado pelo mecanismo. Esse conjunto é formado por uma instrução de *load* e um *store*. Apenas o *store* utiliza os dados carregados pelo *load*. Esses dados são armazenados em memória pelo *store*, caracterizando uma cópia de memória. Fonte: Autor.

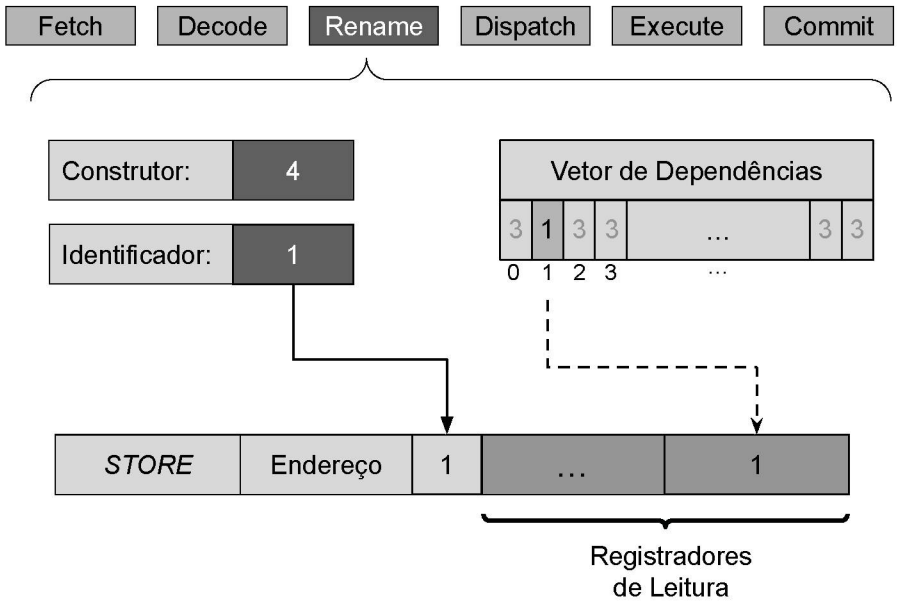


Figura 4.6: Estado das estruturas do mecanismo após encontrar o *store* de um conjunto de cópia de memória. O identificador único da conversão é associado ao *store* e o construtor recebe 4, indicando que o conjunto está completo. Fonte: Autor.

tem a posição correspondente ao registrador físico de escrita do segundo *load* marcada com 1. Além disso, o identificador único da conversão é associado a ele, como apresentado na Figura 4.8.

Com o construtor possuindo o valor 2, o mecanismo procura operações aritméticas ou lógicas operando sobre os dados dos *loads* encontrados, o que é indicado pelo vetor de dependências. Se uma dessas operações for encontrada, como apresentado na Figura 4.9, o construtor é incrementado para 3, enquanto o vetor de dependências recebe o valor 2 nas posições vinculadas aos registradores de escrita da operação. Assim, o mecanismo passa a buscar por um *store* que armazene os dados dessa operação, ou seja, armazene os dados do registrador físico marcado com 2 no vetor de dependências. Se esse *store* for encontrado, como apresentado na Figura 4.10, o construtor recebe o valor 4, indicando que um dos 3 tipos de conjunto para conversão, ilustrados na Figura 4.1, foi identificado. Além disso, o identificador único da conversão é vinculado ao *store*.

Ao encontrar um conjunto completo, o mecanismo passa a procurar por novas iterações desse conjunto. Encontrar pela segunda vez o mesmo *load* indica a presença de um laço ao redor

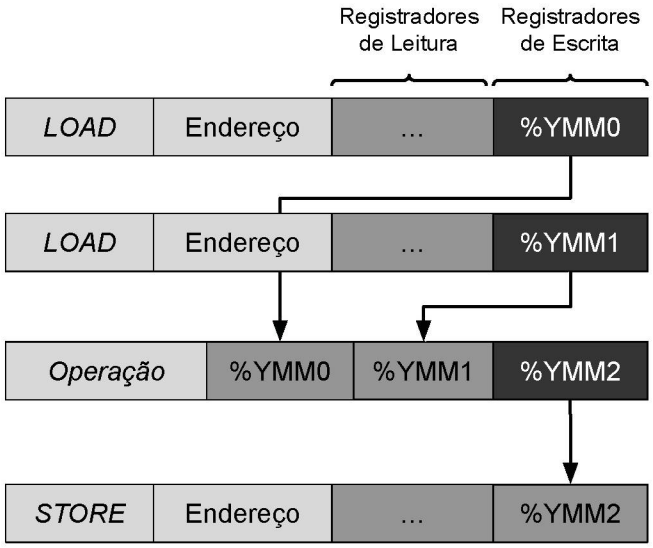


Figura 4.7: Vínculo de instruções no conjunto de operação, elemento a elemento, entre vetores buscado pelo mecanismo. Esse conjunto é formado por duas instruções de *load*, uma operação aritmética/lógica e um *store*. Apenas a operação aritmética/lógica reliza a leitura dos resultados carregados pelos *loads*. O resultado dessa operação é lido por apenas um *store*. Sendo armazenado em memória por ele. Fonte: Autor.

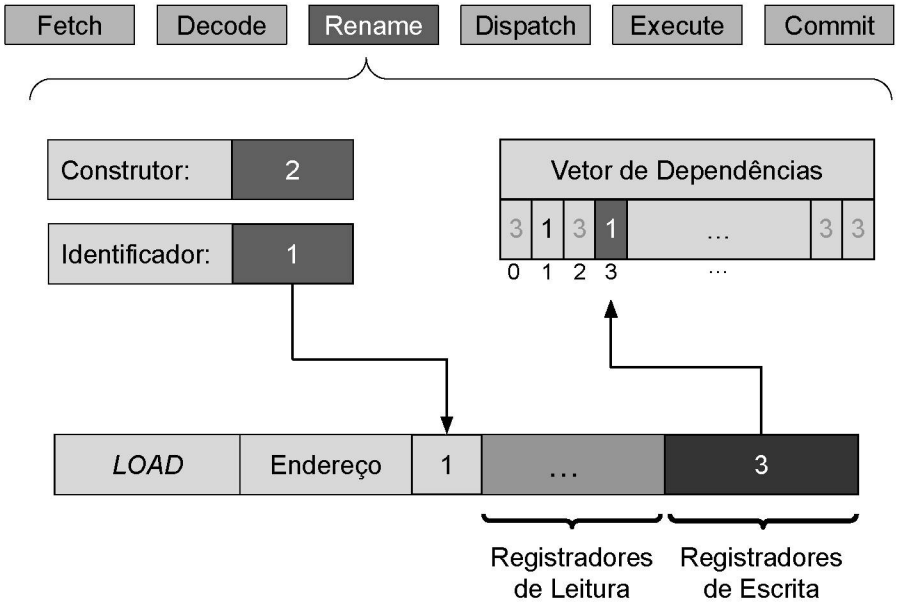


Figura 4.8: Estado das estruturas do mecanismo após encontrar o segundo *load* de um conjunto. Quando esse *load* é encontrado, o identificador único da conversão é associado a ele. Além disso, o vetor de dependências recebe 1 na posição correspondente ao registrador físico no qual ele escreve e o construtor recebe 2, indicando que o segundo *load* do conjunto de operação entre dois vetores foi encontrado. Fonte: Autor.

do conjunto. Nessa segunda identificação, as instruções encontradas são novamente vinculadas ao identificador único da conversão.

Cada nova iteração, encontrada indiretamente pela repetição dos endereços das instruções do conjunto, incrementa um contador de iterações. Visto que as instruções do conjunto recebem o identificador único da conversão, quando a *Address Generation Unit* (AGU) calcula seus endereços de acesso à memória, eles são informados ao SAPIVe. O processo de identificação do conjunto e incremento de iterações continua até o fim da conversão, servindo tanto para contabilizar quantas iterações se passaram desde o início da conversão, quanto para confirmar que

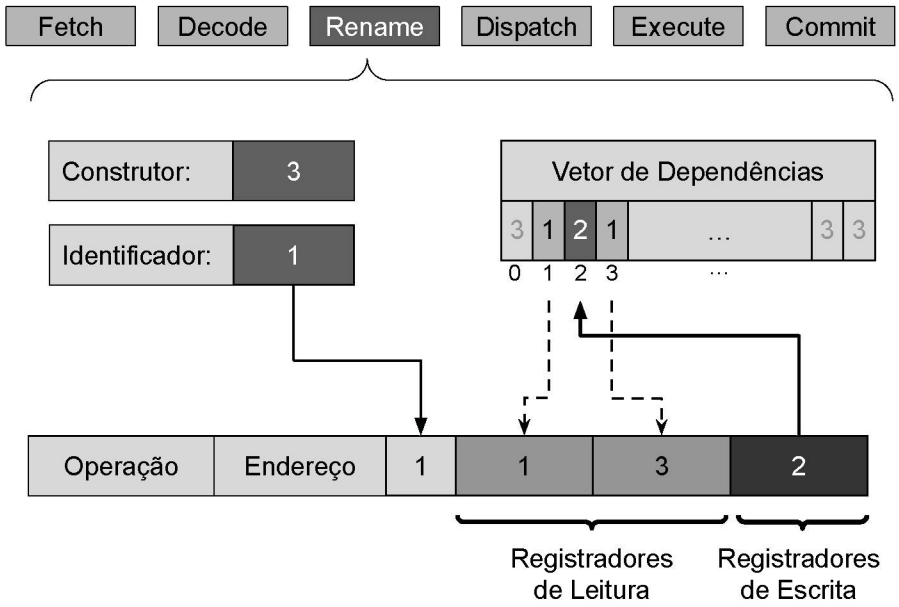


Figura 4.9: Estado das estruturas do mecanismo após encontrar a operação aritmética ou lógica de um conjunto. Quando essa operação é encontrada, o vetor de dependências recebe 2 na posição correspondente ao registrador físico no qual ela escreve e o construtor recebe 3, indicando que a operação do conjunto de operação entre dois vetores foi encontrada. Fonte: Autor.

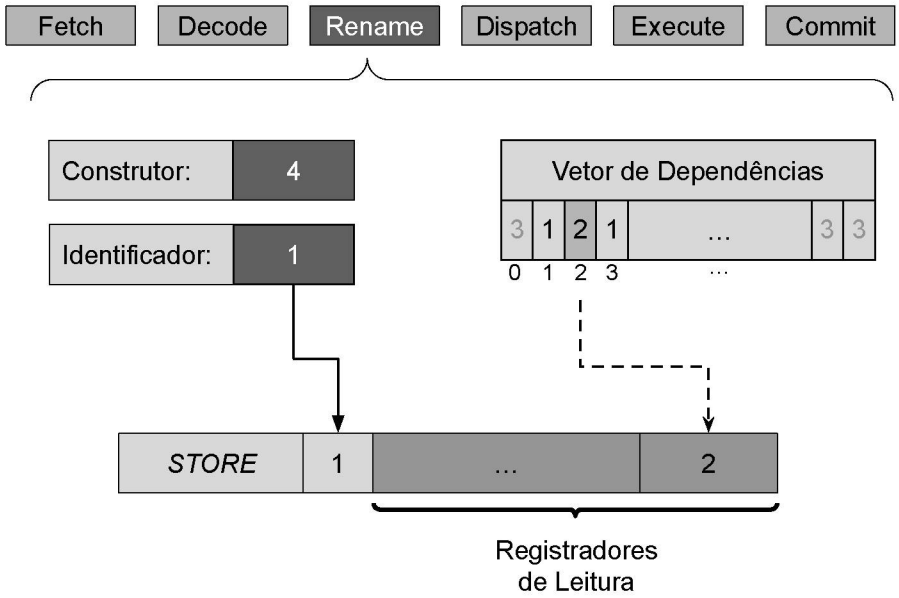


Figura 4.10: Estado das estruturas do mecanismo após encontrar o *store* de um conjunto de operação entre vetores. Quando esse *store* é encontrado, o construtor recebe 4, indicando que o fim do conjunto foi encontrado. Além disso, o identificador único da conversão é vinculado a essa instrução. Fonte: Autor.

o conjunto encontrado se mantém. Ao mesmo tempo em que a terceira iteração é contabilizada, o mecanismo inicia a etapa de análise.

Durante a etapa de identificação, tanto ao encontrar conjuntos, quanto ao contabilizar iterações, três situações podem invalidar uma conversão, interrompendo o processo de conversão e incrementando o identificador único, de forma a diferenciar conversões futuras das invalidadas. A primeira situação de invalidação consiste de uma instrução inesperada ler um registrador físico marcado com 1 ou 2 no vetor de dependências, como apresentado na Figura 4.11, onde

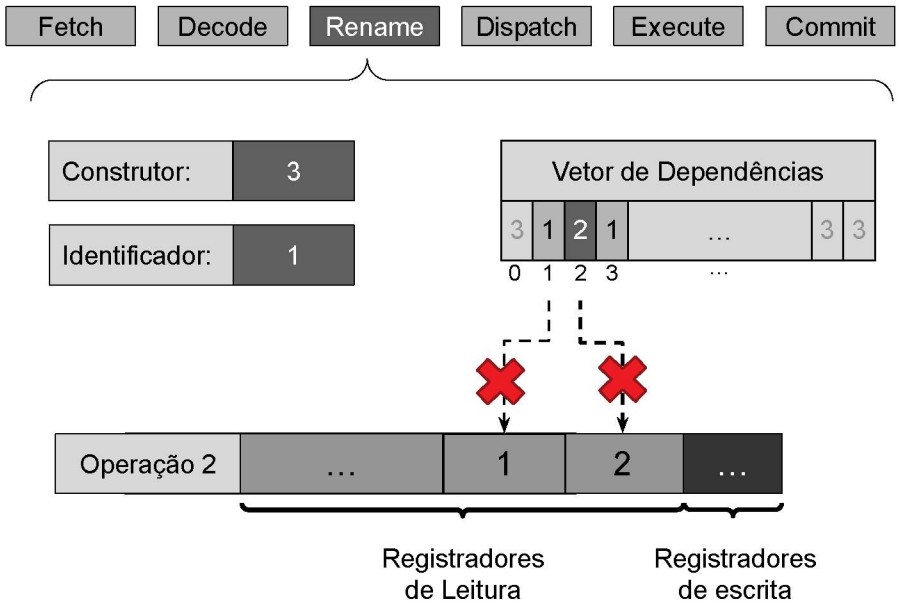


Figura 4.11: Ilustração de uma operação lendo o resultado dos registradores escritos pela operação aritmética/lógica e por um dos *loads* de um conjunto de operação entre vetores. Uma instrução externa ao conjunto não pode ler registradores escritos por instruções da conversão. Caso isso ocorra, a conversão segue para a etapa de invalidação. Fonte: Autor.

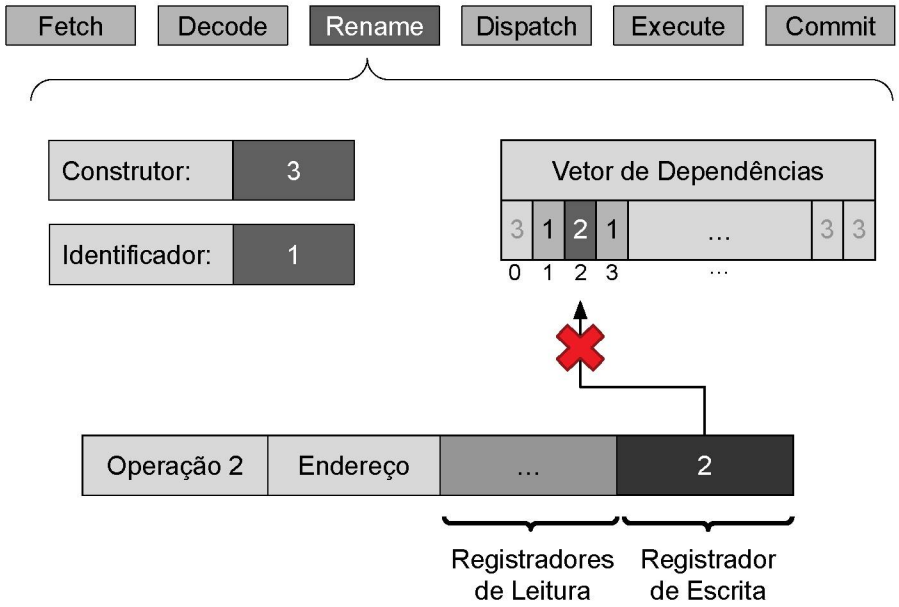


Figura 4.12: Ilustração de uma segunda operação escrevendo sobre resultado dos registradores escritos pela operação aritmética/lógica de um conjunto de operação entre vetores, antes do *store* ser encontrado. Uma instrução externa ao conjunto não pode sobrescrever registradores escritos por instruções da conversão no meio do conjunto, visto que o valor real dos registradores utilizado pelo *store* acabaria sendo o da segunda e não da primeira operação. Apesar disso, também não podemos alterar o conjunto para utilizar a segunda operação ao invés da primeira, já que é a primeira que lê o resultados dos *loads* presentes no conjunto. Assim, caso isso ocorra, a conversão segue para a etapa de invalidação. Fonte: Autor.

uma segunda operação lê o registrador escrito pela operação aritmética/logica e por um dos *loads* do conjunto. Isso, porque quando o SAPIVe executa um conjunto em memória, apenas o valor parcial que estaria nos registradores de escrita da última iteração das instruções do conjunto é levado para dentro dos registradores. Dessa forma, os valores parciais das demais

iterações não ficam disponíveis para instruções da CPU. Assim, qualquer instrução dependente desses resultados não poderá executar corretamente. O mesmo ocorre ao procurar conjuntos, em sobrescritas do conteúdo de registradores físicos marcados com 1 ou 2 no vetor de dependências, visto que é a indicação de que a instrução marcada anteriormente como parte do conjunto não pertence a ele. Um exemplo dessa situação é apresentado na Figura 4.12, onde uma operação de fora do conjunto sobrescreve o valor da operação anterior, enquanto um *store* é buscado.

A segunda situação de invalidação ocorre se existirem instruções, fora da conversão, que realizem acessos à memória. Esse caso leva à invalidação porque não podemos garantir que a execução das leituras e escritas trará os mesmos resultados, caso seus endereços sejam conflitantes com os das instruções convertidas para VIMA. Isso, porque a execução na VIMA incorre no resultado de várias iterações da CPU armazenando seu resultado de forma atômica e independente de outros *loads* e *stores* de cada iteração.

Por fim, nosso terceiro caso diz que uma invalidação também pode ocorrer caso nenhum conjunto seja encontrado ou o conjunto seja alterado durante uma conversão, devido ao fim de um laço ou um salto. Para prevenir a terceira situação de invalidação, além de observar mudanças no conjunto, caso uma instrução de salto seja encontrada entre o primeiro *load* e o *store* de uma das iterações da conversão, essa conversão passa para a etapa de invalidação.

4.2.2 Máquina de estados para a identificação de conjuntos de instruções para conversão

Para realizar a busca por padrões de instruções correspondentes a uma instrução VIMA, o SAPIVe utiliza uma FSM, ilustrada na Figura 4.13, que apresenta os seguintes estados:

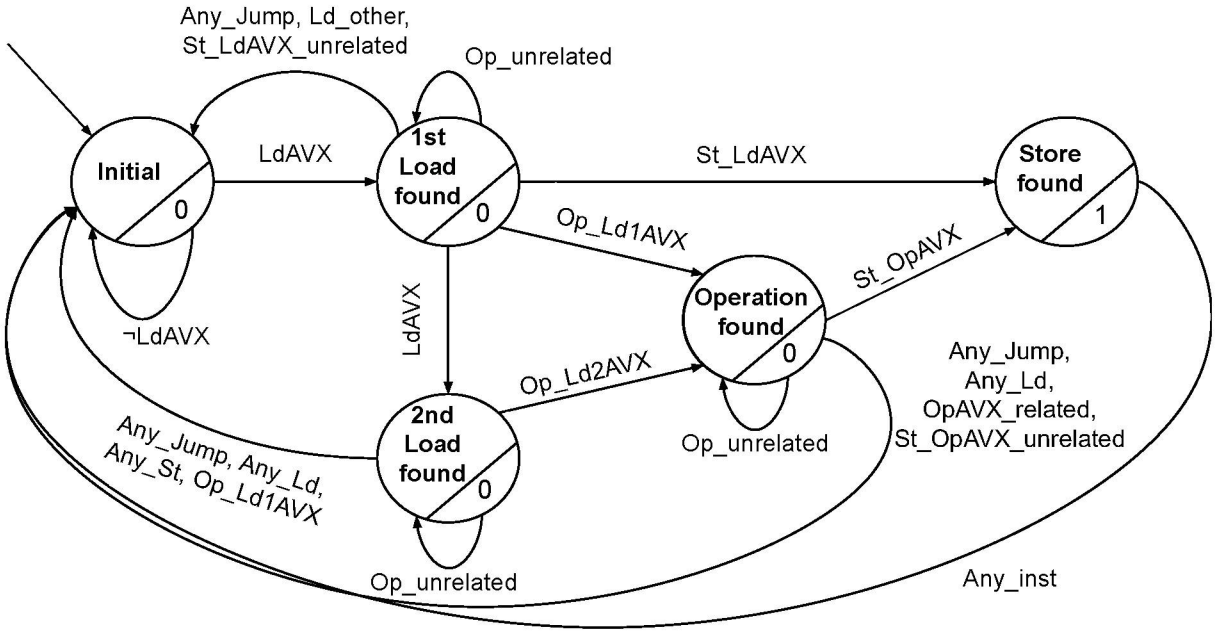


Figura 4.13: Diagrama de transição de estados correspondente à etapa de identificação de conjuntos de instruções que podem ser convertidos para instruções VIMA pelo SAPIVe. Nessa FSM, cada estado representa uma parte da obtenção do padrão, iniciando no estado Initial, onde nenhuma instrução de um conjunto foi encontrada e chegando ao estado Store found, que indica a obtenção bem sucedida de um conjunto. Como saída, essa máquina apresenta um bit indicando quando um conjunto completo foi (1) ou não (0) encontrado. Visto que nosso mecanismo não se propõe a identificar dois padrões no mesmo laço, mesmo que exista um *load* AVX após o *store* de um padrão, a FSM volta ao estado Initial, reiniciando o processo. Fonte: Autor.

Initial: Estado inicial, nenhuma instrução correspondente a um padrão foi encontrada;

1st Load found: Uma primeira instrução de *load* AVX, de 256 ou 512 bits, foi encontrada;

2nd Load found: Uma segunda instrução de *load* AVX, de 256 ou 512 bits, foi encontrada;

Operation found: Uma operação AVX utilizando o(s) resultado(s) do(s) *load(s)* AVX anteriores foi encontrada;

Store found: Um *store* AVX utilizando o resultado calculado pela operação ou pelo primeiro *load* AVX foi encontrado, finalizando a identificação de um conjunto.

Essa máquina apresenta um bit de saída indicando se um padrão foi completamente encontrado (1) ou não (0). Já o alfabeto dela é composto pelas entradas {Any_inst, Any_Jump, Any_Ld, LdAVX, Ld_other, Op_Ld1AVX, Op_Ld2AVX, OpAVX_related, Op_unrelated, Any_St, St_LdAVX, St_LdAVX_unrelated, St_OpAVX, St_OpAVX_unrelated}, que representam diferentes instruções identificadas na renomeação, e são descritas a seguir:

Any_inst - Qualquer instrução.

Any_Jump - Instrução de salto.

Any_Ld - Qualquer operação de *load*, independentemente do tamanho de seus dados carregados.

LdAVX - Instrução de *load* AVX de 256 ou 512 bits.

Ld_other - Instrução de *load* que não opera sobre dados de 256 nem 512 bits.

Op_Ld1AVX - Operação aritmética/lógica AVX sobre o registrador escrito por apenas um *load* da conversão.

Op_Ld2AVX - Operação aritmética/lógica AVX sobre os registradores escritos por dois *loads* da conversão.

OpAVX_related - Operação aritmética/lógica lendo ou escrevendo um registrador escrito por qualquer instrução da conversão.

Op_unrelated - Operação aritmética/lógica não relacionada àquelas instruções do conjunto já identificadas, o que é avaliado comparando seus registradores de leitura e escrita com o vetor de dependências.

Any_St - Qualquer operação de *store*.

St_LdAVX - *Store* AVX sobre o registrador escrito pelo *load* AVX já encontrado.

St_LdAVX_unrelated - Instrução realizando *store* de um dado que não foi gerado pelo *load* AVX encontrado.

St_OpAVX - *Store* AVX sobre o registrador escrito pela operação aritmética/lógica AVX, do conjunto, já encontrada.

St_OpAVX_unrelated - Instrução realizando *store* de um dado que não foi gerado pela operação aritmética/lógica do conjunto sendo identificado.

Com base nesse alfabeto, as transições de estado dessa máquina são apresentadas na Tabela 4.1.

Tabela 4.1: Tabela de transição de estados para a identificação de conjuntos de instruções correspondentes a uma instrução VIMA. Os estados iniciais são indicados por cada linha (I) enquanto os estados finais são indicados por cada coluna (F). Fonte: Autor.

I \ F	Initial	1st Load found	2nd Load found	Operation found	Store found
Initial	¬ LdAVX	LdAVX			
1st Load found	Any_Jump, Ld_other, St_LdAVX_unrelated	Op_unrelated	LdAVX	Op_Ld1AVX	St_LdAVX
2nd Load found	Any_Jump, Any_Ld, Any_St, Op_Ld1AVX		Op_unrelated	Op_Ld2AVX	
Operation found	Any_Jump, Any_Ld, OpAVX_related, St_OpAVX_unrelated			Op_unrelated	St_OpAVX
Store found	Any_inst				

4.2.3 Análise de conjuntos candidatos

Durante a etapa de análise, as verificações da primeira etapa do mecanismo continuam a ocorrer, podendo gerar invalidações. Enquanto isso, cada uma das instruções das duas primeiras iterações do conjunto identificado executam pelo *pipeline*. Por estarem vinculadas a identificadores únicos, quando os *loads* e *stores* calculam seus endereços de acesso à memória, o SAPIVe é informado.

Com base nesses endereços, a etapa de análise tem dois objetivos: identificar se o acesso à memória de cada *load* e *store* do conjunto é adequado para a conversão para VIMA e gerar uma instrução VIMA correspondente às iterações do conjunto que ainda não passaram pelo *rename*.

Para que uma instrução que acessa a memória seja adequada à conversão para a VIMA, os endereços acessados pelas iterações dela precisam ser contíguos. Assim, diversas iterações consecutivas dessa mesma instrução AVX podem ser unidas em uma única instrução VIMA, que também processa dados contíguos em memória, como ilustrado pela Figura 4.14. Além disso, para *stores*, deve ser possível alinhar seu endereço de acesso à memória com as linhas de *cache* da VIMA, evitando falso compartilhamento, ou seja, que conversões consecutivas sejam invalidadas por escrever na mesma linha do *buffer* de escritas.

Código 4.1: Código em C++ de algoritmo utilizado para a identificação de acessos contíguos à memória nas duas primeiras iterações de uma instrução.

```
1  bool contiguo(uint64_t endereco_0, uint64_t endereco_1, uint32_t AVX_SIZE) {
2
3      uint64_t maior = (endereco_0 > endereco_1) ? endereco_0 : endereco_1;
4      uint64_t menor = (endereco_0 > endereco_1) ? endereco_1 : endereco_0;
5
6      if (maior == menor + AVX_SIZE) {
7          /* Acessos contíguos */
8          return true;
9      } else {
10         /* Acessos esparsos */
11         return false;
12     }
13
14 }
```

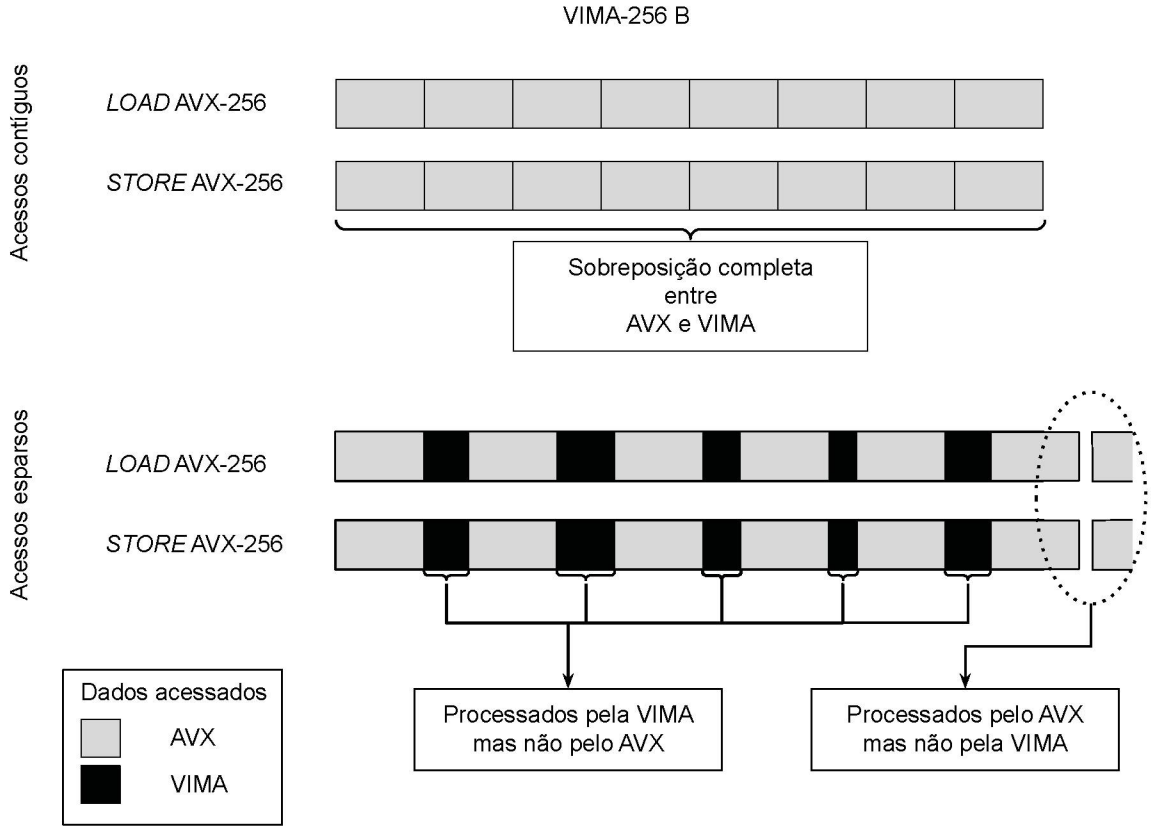


Figura 4.14: Ilustração apresentando a equivalência entre instruções AVX contíguas e instruções VIMA, nos acessos à memória realizados durante uma cópia de memória. Temos que cada instrução VIMA-256 B opera a mesma quantidade de dados que 8 instruções AVX-256. Em uma cópia de memória, se um *load* e um *store* em AVX-256 operam sobre dados contíguos, seu comportamento é o mesmo de uma instrução VIMA-256 B, que também acessa dados contíguos. Isso permite a substituição das instruções AVX pela VIMA, já que os dados lidos e escritos serão os mesmos. Porém, se os dados acessados pelas instruções AVX não forem contíguos, sendo então denominados acessos esparsos, a equivalência com as instruções VIMA é perdida, visto que dados diferentes seriam lidos e escritos por cada tipo de instrução. Fonte: Autor.

Para verificar se uma instrução apresenta acessos contíguos à memória, utilizamos os endereços de acesso à memória das duas primeiras iterações do conjunto, enviados pela AGU, como apresentado no Código 4.1. Esse algoritmo recebe o endereço das duas primeiras iterações de uma instrução, após seu cálculo pela AGU, além do tamanho dos dados (em bytes) acessados por essa instrução, retornando *true* caso os acessos sejam contíguos e *false* caso contrário. Assim, caso uma das instruções não apresente contiguidade, a conversão segue para a etapa de invalidação e o processo de conversão reinicia sua busca por conjuntos.

Por outro lado, se todas as instruções de acesso à memória do conjunto realizam acessos contíguos nessas duas primeiras iterações, o SAPIVe inicia a geração de uma instrução VIMA correspondente. Sabendo quantas iterações já passaram pelo estágio de *rename*, e que os acessos à memória das instruções do conjunto presentes nessas iterações são contíguos, o mecanismo é capaz de prever quais os endereços acessados por essas instruções durante qualquer iteração k pela seguinte fórmula:

$$Addr_k = Addr \pm ((k-1) * AVX_SIZE)$$

Onde $Addr_k$ é o endereço de memória acessado por uma instrução da iteração k, maior que zero. $Addr$ é o endereço acessado por essa instrução durante sua primeira iteração, informado pela AGU no início da etapa de análise. AVX_SIZE corresponde ao tamanho dos dados processados pelo tipo de instrução AVX convertida. Somando ou subtraindo caso os endereços acessados a cada iteração sejam crescentes ou decrescentes, respectivamente.

Com base no endereço de *store* acessado pela primeira iteração ($endereco_0$), no contador de iterações da etapa de identificação ($iteracao_atual$), no log_2 do tamanho dos dados operados pelas instruções AVX de origem (LOG_AVX_SIZE), no tamanho dos dados operados pelas instruções de destino ($VIMA_SIZE$) da conversão e sabendo se os endereços acessados a cada iteração são crescentes ou decrescentes ($crescente$) nosso mecanismo tenta encontrar qual a próxima iteração (p) que terá o endereço de seu *store* alinhado às linhas de *cache* da VIMA, o que é apresentado no Código 4.2.

Código 4.2: Código em C++ representando o algoritmo utilizado para a identificação da próxima iteração de uma conversão que apresenta um *store* alinhado à *cache* da VIMA.

```
int8_t proxima_alinhada (uint64_t endereco_0,
                         uint64_t iteracao_atual,
                         uint64_t LOG_AVX_SIZE,
                         uint64_t VIMA_SIZE,
                         bool crescente) {
    uint64_t endereco_atual;    // Endereço da iteração atual
    uint64_t proximo_alinhado;  // Possível próximo endereço alinhado
    uint64_t iteracao_alinhada; // Iteração do endereço alinhado
    uint64_t aux;

    /* Endereço da iteração atual */
    endereco_atual = (crescente)
                   ? endereco_0 + (iteracao_atual << LOG_AVX_SIZE)
                   : endereco_0 - (iteracao_atual << LOG_AVX_SIZE);

    /* Possível próximo endereço alinhado */
    proximo_alinhado = (crescente)  ? endereco_atual + VIMA_SIZE
                                    : endereco_atual - (1 << LOG_AVX_SIZE);

    proximo_alinhado &= ~(VIMA_SIZE - 1);

    /* Verifica se poderia alinhar */
    aux = (crescente) ? (proximo_alinhado - endereco_0)
                      : (endereco_0 - proximo_alinhado);

    if ((aux & ((1 << LOG_AVX_SIZE) - 1)) != 0) {
        /* Não pode ser alinhado */
        return -1;
    } else {

        /* Iteração do endereço alinhado*/
       iteracao_alinhada = aux >> LOG_AVX_SIZE;

        return iteracao_alinhada;
    }
}
```

A partir dos endereços dessa iteração p, o mecanismo gera uma instrução VIMA, equivalente a p e a suas iterações seguintes, iniciando a etapa de validação. Caso não seja

possível alinhar o *store*, a conversão é invalidada e o processo de identificação de conjuntos é reiniciado.

O número de iterações (N) equivalente à instrução VIMA gerada depende do tamanho dos dados processados pela instrução VIMA ($VIMA_SIZE$) e pelas instruções AVX convertidas (AVX_SIZE), sendo definido por:

$$N = VIMA_SIZE / AVX_SIZE$$

A instrução VIMA gerada nesta etapa é vinculada ao identificador único da conversão e é inserida no estágio *rename* do *pipeline*, assim, iniciando sua execução antes das instruções AVX substituídas passarem por esse estágio. Por conta disso, invalidações ocorridas após o envio dessa instrução devem ser informadas à VIMA, para que o resultado de suas operações seja descartado.

Visto que o tamanho dos dados operados por cada instrução AVX e VIMA são potências de 2, as multiplicações e divisões necessárias para os cálculos dos endereços acessados por cada iteração e do número de iterações necessárias para uma conversão podem ser realizadas apenas com operações de *shift*.

4.2.4 Validação de conversão

Durante a etapa de validação uma instrução VIMA, correspondente a N iterações futuras do laço (de p a $p + N - 1$), é executada pela VIMA. Após sua execução, a VIMA armazena o resultado obtido em seu *buffer* de escritas e informa a CPU sobre o sucesso ou falha da execução da instrução VIMA, junto a algumas informações adicionais, e aguarda o resultado das verificações da conversão na CPU, caso ainda não tenha recebido.

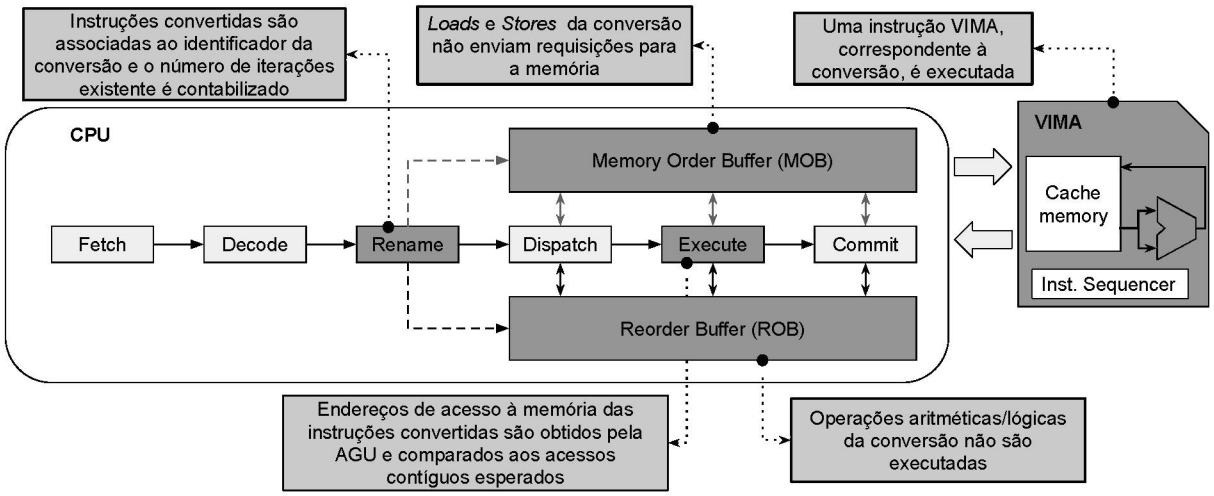


Figura 4.15: Ilustração indicando o funcionamento da etapa de validação do SAPIVe, assim como os estágios do *pipeline* envolvidos nessa tarefa. Fonte: Autor.

Ao mesmo tempo, as instruções AVX correspondentes às N iterações passam pelo *rename*. De forma similar ao ocorrido durante a identificação, essas instruções são associadas ao identificador único da conversão em andamento e à sua iteração correspondente. Assim, após o cálculo realizado pela AGU, seus endereços de acesso à memória serão informados ao mecanismo.

Conforme passam pelo *rename*, as instruções convertidas recebem um tratamento diferenciado pelo *pipeline*. Enquanto as operações aritméticas/lógicas são mantidas travadas

no ROB, os *loads* e *stores* alocam entradas correspondentes no *Memory Order Buffer* (MOB) e realizam o cálculo de seus endereços na AGU, porém sem enviar quaisquer requisições à hierarquia de memória. Dessa forma, essas instruções convertidas são mantidas travadas no ROB e no MOB até a confirmação ou invalidação da conversão. Essas etapas necessárias para a validação são ilustradas e resumidas na Figura 4.15.

Para a confirmação de uma conversão, a etapa de validação verifica duas condições. A primeira é que as N iterações equivalentes à instrução VIMA executada devem ter sido inseridas no ROB, o que é verificado avaliando o valor contido no contador de iterações da etapa de identificação. Caso um laço contenha menos de N iterações, haverá um momento em que a cabeça do ROB será uma instrução dessa conversão, ao mesmo tempo em que não existirá espaço no ROB para completar as N iterações, invalidando a conversão.

Como cada instrução convertida está vinculada ao identificador único de sua conversão, sempre que um endereço de acesso à memória dessas instruções é calculado ele é informado ao mecanismo. Com isso, e com a informação da iteração correspondente a cada instrução AVX, o SAPIVe é capaz de verificar a segunda condição, se os endereços de acesso à memória previstos para a instrução VIMA na etapa anterior se mantém, ou seja, se os acessos à memória continuam contíguos. Isso é apresentado no Código 4.3, que recebe o endereço de aceso à memória de uma instrução convertida ($endereco_agu$), após seu cálculo pela AGU, a iteração na qual essa instrução seria executada ($iteracao$), o endereço acessado pela primeira iteração dela ($endereco_0$), o log_2 do tamanho dos dados operados pelas instruções AVX de origem (LOG_AVX_SIZE) e uma flag indicando se os endereços acessados por cada uma das iterações dessa instrução são crescentes ($crescente$).

Para instruções AVX com máscaras, a conversão também deve garantir que a operação se aplica a todos os elementos, ou seja, a máscara não é utilizada para impedir nenhuma operação nessas instruções. Isso, porque as instruções atuais da VIMA não possibilitam o uso de máscaras, assim, se algumas operações de uma instrução AVX são inibidas por uma máscara, a conversão correspondente a essa instrução é aplicada incorretamente, como ilustrado pela Figura 4.16. Devido a isso, caso operações de uma instrução AVX sendo convertida sejam inibidas, a conversão deve ser invalidada.

Código 4.3: Código em C++ representando o algoritmo utilizado para verificar se os acessos à memória de uma instrução convertida continuam contíguos, com base em um novo endereço de acesso à memória calculado pela AGU.

```
bool mantem_contiguo (uint64_t endereco_agu,
                      uint64_t iteracao,
                      uint64_t endereco_0,
                      uint64_t LOG_AVX_SIZE,
                      bool crescente) {

    uint64_t endereco_esperado; // Endereço esperado para a iteração atual

    /* Endereço esperado para a iteração */
    endereco_esperado = (crescente)
                        ? endereco_0 + (iteracao << LOG_AVX_SIZE)
                        : endereco_0 - (iteracao << LOG_AVX_SIZE);

    /* Compara o endereço esperado com o obtido */
    if (endereco_agu == endereco_esperado) {
        return true;
    } else {
        return false;
    }
}
```

Se esses requisitos forem cumpridos, a CPU completa o necessário para a conversão, informando a VIMA sobre a equivalência e aguardando a confirmação de sucesso ou falha da instrução VIMA, caso ainda não tenha recebido.

Por outro lado, caso o número de iterações existentes no laço não seja suficiente para equivaler à instrução VIMA ou o *stride* dos acessos à memória seja alterado durante a conversão, essa conversão deve ser invalidada pelo mecanismo, informando à VIMA sobre a falha para que seu resultado seja descartado.

Caso a instrução VIMA tenha executado sem exceções e a CPU confirme a equivalência da conversão, a etapa de confirmação é iniciada.

4.2.5 Confirmação de conversão

Se todas as iterações convertidas se mostram equivalentes à instrução VIMA, tendo esta completado com sucesso, o mecanismo entra na etapa de confirmação. Ao fim da etapa de validação, a CPU informa à VIMA sobre o sucesso da verificação de equivalência, assim o resultado armazenado no *buffer* de escritas da VIMA é salvo definitivamente em memória. Já a VIMA informa a CPU sobre o sucesso na execução da instrução VIMA equivalente, permitindo a entrada na etapa de confirmação. Além disso, junto a essa mensagem, a VIMA envia à CPU os dados que teriam sido gerados pelo(s) *load(s)* e pela operação aritmética/lógica caso a última iteração AVX fosse executada pela CPU. Assim, na etapa de confirmação, esses dados são posicionados nos registradores de escrita dessas operações, como ilustrado pela Figura 4.17, evitando problemas de dependências com as próximas instruções. Em seguida, o identificador único de conversões é incrementado, iniciando uma nova conversão a partir da etapa de identificação. Por fim, as instruções do conjunto presas no ROB e MOB tem suas dependências resolvidas e são liberadas para o *commit*.

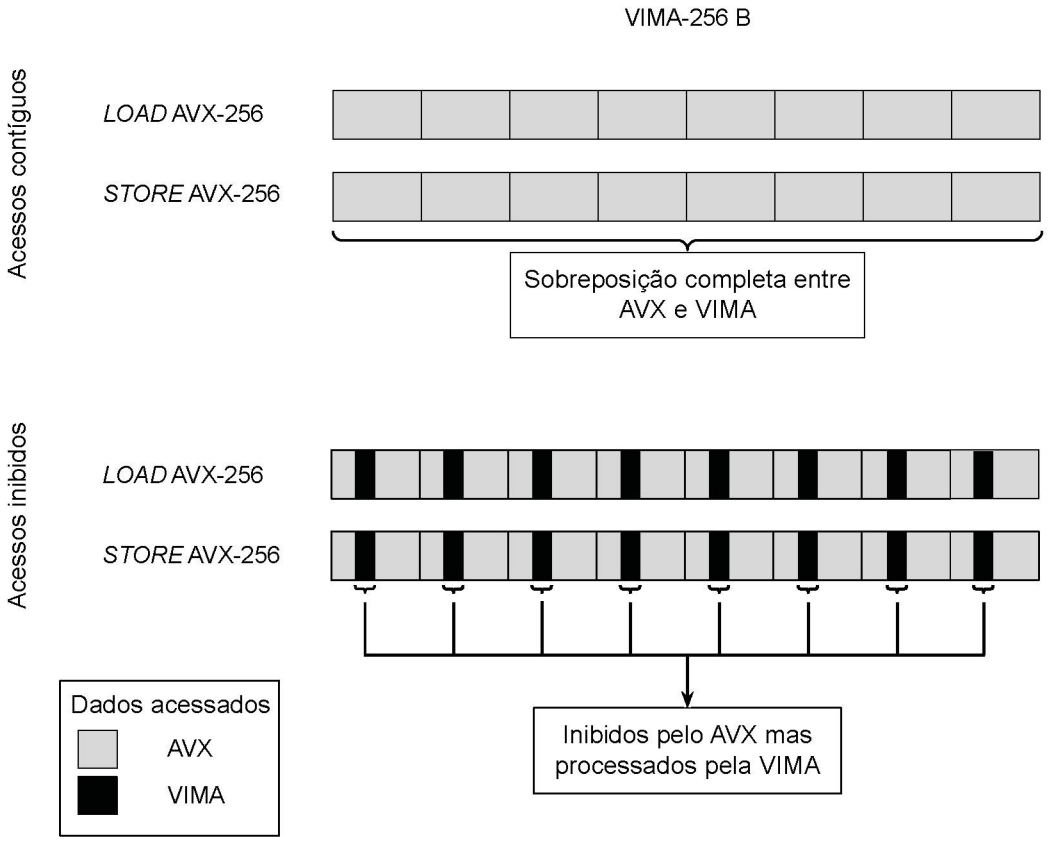


Figura 4.16: Ilustração apresentando problemas na conversão entre instruções AVX com operações inibidas por máscaras e instruções VIMA, considerando os acessos à memória realizados durante uma cópia de memória. Temos que cada instrução VIMA-256 B opera a mesma quantidade de dados que 8 instruções AVX-256. Em uma cópia de memória, se um *load* e um *store* em AVX-256 não utilizam máscaras para inibir o carregamento e o armazenamento de dados, seu comportamento é o mesmo de uma instrução VIMA-256 B. Porém, se uma máscara impedir a cópia de algum dos dados, a equivalência com as instruções VIMA, que não apresenta máscaras, se perde, visto que mais dados são lidos e escritos pela VIMA. Fonte: Autor.

4.2.6 Invalidação de conversão

Caso alguma das etapas do processo de conversão tenha problemas, o mecanismo entrará na etapa de invalidação. Essa etapa consiste em destravar as entradas do ROB e dos MOBs de leitura e escrita, de forma que as instruções que seriam substituídas pela conversão invalidada executem normalmente no processador. Além disso, caso a invalidação seja causada pela CPU, como em um laço sem iterações suficientes para a conversão, ou uma exceção, a CPU deve informar a VIMA, de forma que esta elimine a instrução VIMA e o resultado associado à conversão invalidada do *buffer* de escritas. Se a invalidação for causada pela instrução VIMA, o mesmo processo acontece, porém com a VIMA alertando a CPU e descartando seu resultado antes da CPU liberar a execução das instruções convertidas.

Mais de uma conversão pode estar ativa ao mesmo tempo, assim, caso uma conversão seja invalidada, todas as conversões com um identificador único mais recente que ela também serão invalidadas garantindo a semântica correta da execução.

Após uma invalidação, o identificador único de conversões é incrementado. Isso evita que instruções pertencentes a conversões invalidadas enviem seu resultado erroneamente para

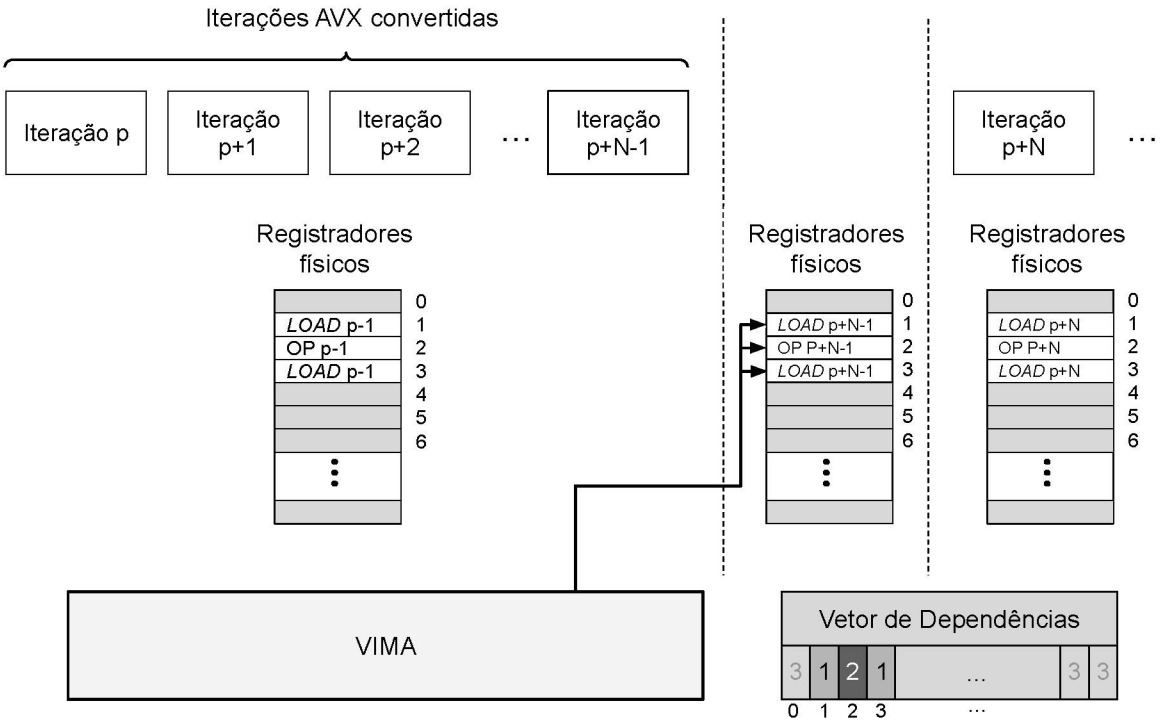


Figura 4.17: Ilustração representando o envio dos dados, da última iteração convertida, da VIMA para a CPU. Enquanto uma conversão das iterações p a $p + N - 1$ acontece, seu processamento é enviado para a memória. Dessa forma, ao fim da conversão, o banco de registradores da CPU está desatualizado, visto que nenhuma das instruções convertidas é executada na CPU (apenas os cálculos de endereços são realizados ali). Para compensar isso, junto à sua mensagem de sucesso ou falha na execução, a VIMA envia os dados que teriam sido carregados e gerados pelas instruções convertidas durante a última iteração. Caso a conversão seja bem sucedida, esses dados são posicionados nos registradores adequados, utilizando as informações coletadas pelo vetor de dependências. Com isso, a execução pode seguir normalmente após a conversão. Fonte: Autor.

as novas conversões. Além disso, o mecanismo reinicia a etapa de identificação, buscando por conjuntos para uma nova conversão.

4.3 OTIMIZAÇÕES

Além do comportamento básico descrito nas seções anteriores, o SAPIVe também conta com otimizações importantes, descritas a seguir.

4.3.1 Conversões consecutivas

Laços que podem ser convertidos possuem um grande número de iterações, além de realizarem acessos contíguos à memória. Porém, após todas as etapas de conversão, apenas poucas iterações AVX são convertidas para VIMA.

Considerando instruções AVX-512 para VIMA-256 B, apenas 4 iterações podem ser substituídas por cada instrução VIMA. Contudo, desde o início da etapa de identificação até a primeira iteração convertida, diversas iterações são executadas. Isso também ocorre entre conversões, onde diversas iterações precisam ser executadas em CPU, obtendo informações já existentes sobre os endereços de acesso à memória, contiguidade e conjuntos de instruções.

Assim, otimizamos o SAPIVe para converter iterações de forma contígua. Ou seja, ao final de uma conversão, iniciamos a próxima imediatamente, sem necessitar aguardar

novamente pelas etapas de identificação e análise. Isso é ilustrado pela Figura 4.18, onde temos o comportamento original do mecanismo, convertendo algumas iterações para a VIMA e executando outras na CPU, em comparação a uma conversão consecutiva, que busca enviar todas as iterações, após a primeira conversão, de um conjunto para a VIMA.

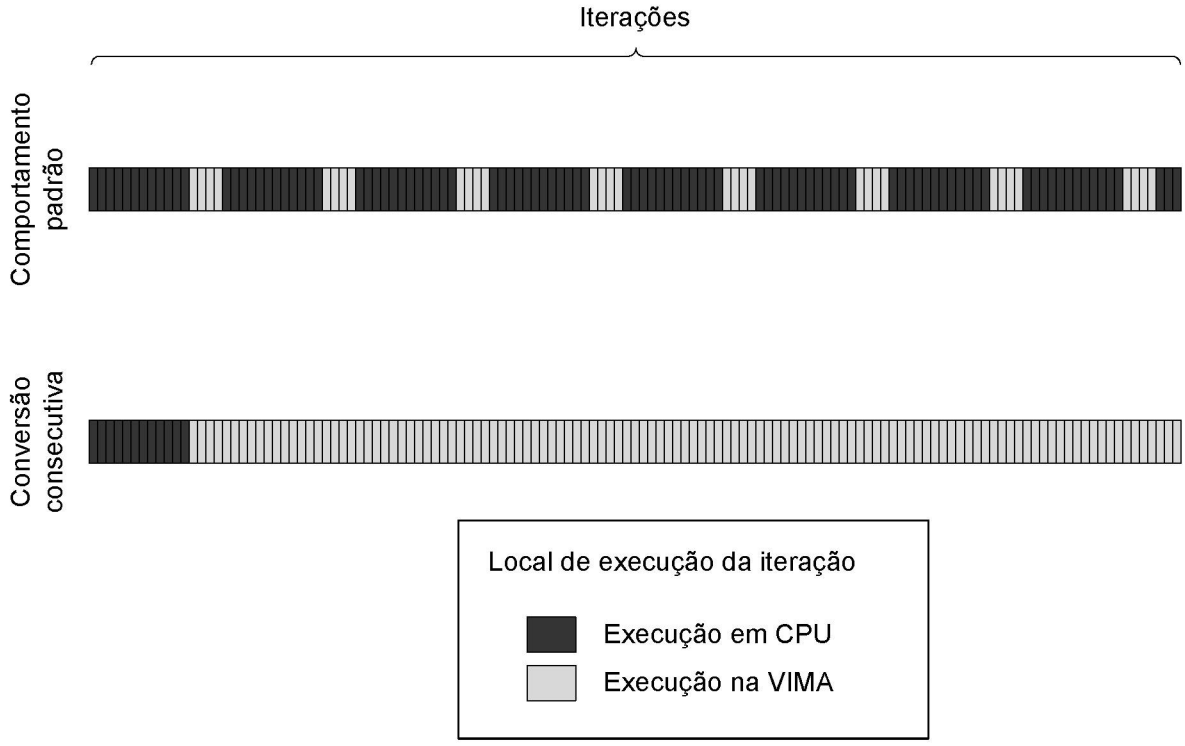


Figura 4.18: Ilustração representando as pausas entre conversões existentes na versão original de nosso mecanismo, em comparação a uma conversão com a otimização de conversões consecutivas. A cada conversão, a versão original reinicia o processo de conversão, que leva diversas iterações para ser completado, antes de lançar uma nova instrução VIMA para um pequeno número de iterações. Já na versão otimizada, a partir da primeira conversão, todas as iterações são convertidas consecutivamente. Assim, aproveitando de maneira mais eficiente o processamento em memória. Fonte: Autor.

Para permitir conversões consecutivas, quando a última iteração de uma conversão passa pelo *rename*, uma nova conversão correspondente às próximas iterações é lançada especulativamente.

Temos que a etapa de identificação da conversão anterior continua contabilizando o número de iterações a passarem pelo ROB, os acessos à memória são contíguos e o endereço de *store* já está alinhado com a *cache* da VIMA, devido à conversão anterior. Assim, a nova conversão inicia já na etapa de validação, após lançar sua instrução VIMA correspondente. Visto que essa nova conversão possui um identificador único diferente da conversão anterior, os dados de cada uma podem ser diferenciados pelo SAPIVe.

Esse processo se repete até o fim do laço, quando não houver iterações suficientes para completar uma conversão. Nesse processo, cada conversão é lançada antes mesmo de sua primeira iteração passar pelo estágio de *rename*, assim, mesmo laços com um número exato de iterações para nossas conversões têm uma última conversão invalidada (conversão sem iterações).

4.3.2 Lista de conversões falhas

Para evitar que a execução seja travada desnecessariamente por invalidações frequentes da mesma conversão, nosso mecanismo possui a lista de conversões falhas, ou seja, uma lista

de conversões a serem evitadas. Caso um laço tenha sua conversão invalidada, sem que ao menos uma conversão tenha sido realizada com sucesso, o mecanismo evita outras tentativas de conversão desse mesmo laço usando essa lista.

A lista de conversões falhas armazena o endereço da instrução do primeiro *load* de cada conjunto que falhou em ser convertido, assim o mecanismo não tentará novas conversões desses mesmos conjuntos. Esse vetor é acessado no início da etapa de Identificação (ver Seção 4.2.1), no momento em que o primeiro *load* de um possível conjunto é encontrado. Além disso, o tamanho ideal dessa lista depende da aplicação executada, sendo que quanto maior o número de conversões falhas em determinado trecho de uma aplicação, mais entradas serão necessárias. Apesar disso, o estudo do espaço de projeto do tamanho dessa estrutura foge ao escopo desta dissertação e é considerado um trabalho futuro.

Para nossos experimentos, consideramos uma lista com oito entradas acessada em um único ciclo.

4.3.3 Agressividade de adiantamentos

Em relação ao uso da VIMA com *intrinsics*, as conversões realizadas por nosso mecanismo são afetadas por dois gargalos. O primeiro é o tempo para processar as instruções da conversão e calcular seus endereços de acesso à memória. Esse tempo é determinado por características do *pipeline* e da aplicação. O segundo é o tempo de execução das instruções VIMA. Uma maneira de reduzir o segundo é realizando adiantamentos das conversões especulativas, como apresentado na Figura 4.19.

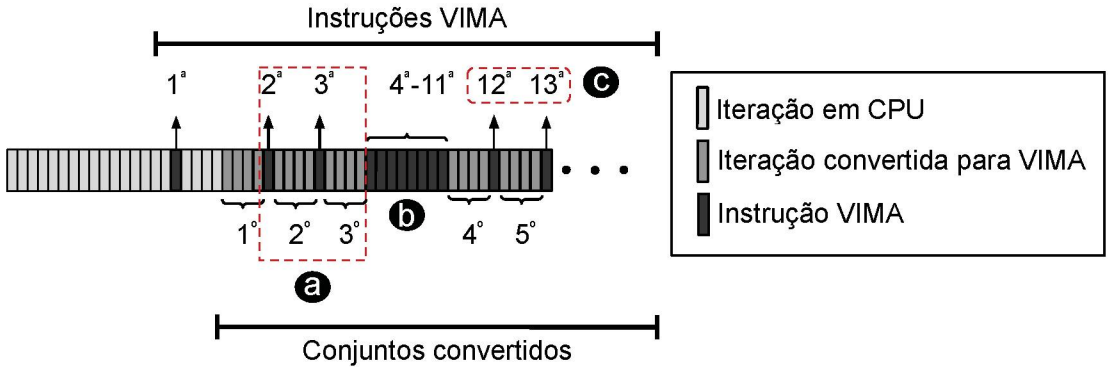


Figura 4.19: Passos para a conversão de um laço. (a) representa a otimização de conversões consecutivas. (b) ilustra o início do adiantamento de conversões especulativas para VIMA. (c) adianta novas conversões especulativas para VIMA, após uma conversão utilizar uma especulação adiantada anteriormente.

Para isso, após a realização de duas conversões consecutivas (a), nosso mecanismo adianta conversões, inserindo instruções especulativas VIMA adicionais, que correspondem a conversões futuras, no *pipeline* (b). Esse adiantamento é denominado agressividade. Assim, quanto mais agressivo o nosso mecanismo, maior o número de adiantamentos de conversão, enquanto que um mecanismo com agressividade igual a 0, não realiza nenhum adiantamento.

As instruções VIMA adiantadas são executadas em paralelo com as verificações das conversões em andamento, reduzindo o tempo de espera entre conversões e acelerando a busca de dados em memória.

Para aproveitar essa agressividade, cada nova conversão verifica se a instrução VIMA mais antiga, dentre aquelas sendo adiantadas, correspondente a ela. Caso exista, a instrução VIMA desse adiantamento é vinculada à conversão, que continua sua execução de forma usual. Além disso, como um espaço para adiantamentos fica livre, o SAPIVe aproveita para avançar

mais conversões especulativas, lançando uma nova instrução VIMA correspondente à próxima conversão futura (c).

Se não existir um adiantamento correspondente, todas os existentes são invalidados, visto que o laço correspondente a eles já foi finalizado. Além disso, é criada uma nova instrução VIMA para a conversão atual.

Em caso de invalidação, os adiantamentos em andamento também são invalidados, liberando espaço para novas conversões e adiantamentos atualizados.

A agressividade utilizada é limitada pelo tamanho do *buffer* de escritas presente na VIMA, e pelo número de conversões consecutivas permitido pelo ROB e MOB. Isso, porque todas as conversões e adiantamentos em andamento compartilham as entradas existentes nesses *buffers*.

Em caso de falta de espaço, a *cache* de leituras da VIMA não impede novos adiantamentos, visto que não precisa manter dados temporários e exige apenas os dados da instrução atual para permitir a execução.

4.4 REQUERIMENTO DE *HARDWARE*

Para realizar suas operações, o SAPIVe necessita de algumas estruturas auxiliares, sendo as principais apresentadas nesta seção.

Primeiramente, nosso mecanismo precisa de quatro vetores (conversões falhas, dependências, conversões em andamento, adiantamentos), para identificar e coordenar suas conversões, impedindo invalidações frequentes.

A lista de conversões falhas (implementada em nossos experimentos como um vetor de 8 posições) impede novas tentativas de conversão de conjuntos que foram invalidados sem ao menos uma conversão bem sucedida. Cada posição desse vetor possui um campo (8 bytes), armazenando o endereço do primeiro *load* dessas conversões falhas, além de um bit de validade (em nosso exemplo, essa estrutura ocuparia 65 bits$*$8 = 65 bytes).

O vetor de dependências serve para rastrear as dependências entre instruções durante a identificação de conjuntos, ele possui uma posição associada a cada registrador físico do sistema. Cada posição desse vetor possui um único campo (2 bits), indicando que instrução escreve sobre seu registrador associado (em nosso exemplo de arquitetura inspirada no x86, com 256 registradores físicos, essa estrutura ocuparia 512 bits de armazenamento).

O vetor de conversões em andamento armazena informações de cada conversão sendo realizada, permitindo verificações paralelas de diferentes conversões (as instruções de cada conversão possuem um ponteiro para sua posição associada neste vetor), ele apresenta uma posição para cada conversão simultânea que o SAPIVe pode realizar. Em nosso exemplo de ROB com 224 entradas (Tabela 2.2), sabendo que são necessárias 4 iterações das instruções AVX de maior tamanho (512 bits) para cada instrução VIMA de menor tamanho (256 B) e considerando que cada iteração apresenta 10 micro-ops, no pior caso apenas 6 conversões poderiam ocupar o ROB simultaneamente ($\lceil 224/(4 * 10) \rceil$). Assim, nossos experimentos consideraram um vetor de conversões em andamento com 8 entradas, sendo que cada posição desse vetor possui os seguintes campos (requerindo, em nosso exemplo, 251 bits por entrada e 251 bytes considerando as 8 entradas):

- Iteração Inicial (4 bytes): Caso um laço apresente conversões consecutivas, a iteração inicial de cada conversão será diferente.

- Primeiro *Load* (8 bytes): Endereço de memória acessado na primeira iteração convertida do primeiro *load* do conjunto.

- Segundo *Load* (8 bytes): Endereço de memória acessado na primeira iteração convertida do segundo *load* do conjunto (se existir).
- *Store* (8 bytes): Endereço de memória acessado na primeira iteração convertida do *store* do conjunto.
- Instrução VIMA Completa (1 bit): Bit indicando que a instrução VIMA correspondente a esta conversão foi executada com sucesso.
- Conversão Completa (1 bit): Bit indicando que a conversão foi bem sucedida e as instruções associadas a esta entrada podem seguir diretamente para o *commit*.
- Conversão Invalidada (1 bit): Bit indicando que a conversão associada a essa entrada foi invalidada, o que permite a execução convencional das instruções vinculadas a ela.
- Registradores Escritos (3 bytes): Contém 3 valores (cada um ocupando 1 byte), indicando os registradores físicos que seriam escritos pelas instruções convertidas.

Um vetor similar (adiantamentos) é utilizado para armazenar os adiantamentos nas conversões do SAPIVe. Cada uma de suas posições contém os mesmos campos do vetor de conversões em andamento, porém trocando os bits de Conversão Completa e Conversão Invalidada por campos contendo o endereço de cada instrução do conjunto convertido (em nossos experimentos ocupando 8 bytes $*4$). Com isso, o SAPIVe garante que cada adiantamento não seja vinculado a uma conversão incorreta. Assim como o vetor de conversões em andamento, o vetor de adiantamentos contém uma entrada para cada adiantamento realizado (o que, em nosso exemplo, ocupa 4 bytes para iteração inicial, 8 bytes para o endereço do primeiro *load*, 8 bytes para o endereço do segundo *load*, 8 bytes para o endereço acessado pelo *store*, um bit para indicar que a instrução VIMA associada está completa, 3 bytes para indicar os registradores escritos pelas instruções convertidas e 8 bytes para cada um dos endereços das 4 instruções convertidas, totalizando 505 bits por entrada e 2020 bytes considerando as 32 entradas de nossos experimentos).

Dessa maneira, em nosso exemplo de uma arquitetura inspirada no x86 com 256 registradores internos e 224 entradas no ROB, as principais estruturas do SAPIVe precisariam de menos de 3 KiB de armazenamento.

Para sua operação, o SAPIVe também precisa de uma unidade de comparação, um somador/subtrator e uma unidade de *shift*, para avaliar se os acessos à memória das instruções do conjunto são contíguos durante as iterações e identificar qual a próxima iteração convertida com seu *store* alinhado à *cache* da VIMA. Além disso, 7 comparadores são necessários para avaliar se uma nova conversão apresenta as mesmas instruções convertidas e os mesmos endereços acessados que o adiantamento mais antigo do vetor de adiantamentos.

4.5 RESTRIÇÕES DO MECANISMO

Consideramos que as instruções de memória possuem o mesmo processo de tradução de qualquer outra requisição de leitura ou escrita, por meio do *Translation Look-aside Buffer* (TLB). Apesar disso, os vetores são alinhados em páginas com limites de 1 GiB e alocados em uma grande página (huge pages), de tal forma que a tradução dessa grande página cobre o vetor inteiro. Porém, devido a restrições de tempo para a implementação, nossas simulações não consideram o tempo adicional exigido pelo uso de vetores desalinhados entre duas páginas, considerando que um acesso entre duas páginas será invalidado.

Visto que nosso mecanismo visa aplicações genéricas e seus vetores podem estar desalinhados entre si, buscamos alinhar apenas as escritas, e não as leituras, de nossas conversões às linhas de *cache* da VIMA.

Também consideramos que tanto a *cache* de leitura da VIMA quanto seu *buffer* de escritas são acessados simultaneamente, possuindo o mesmo tempo de acesso da *cache* original, não interferindo no caminho crítico da VIMA. Não consideramos o tempo de transferência de dados entre a *cache* de leituras e o *buffer* de escritas da VIMA, no caso de leituras de dados presentes no *buffer* de escritas ou escritas de dados presentes na *cache* de leituras.

5 RESULTADOS

Esta seção apresenta uma avaliação do SAPIVe sobre aplicações ideais e algumas aplicações reais.

5.1 APLICAÇÕES IDEAIS

O SAPIVe tem como foco a conversão de instruções presentes em laços de cópia e operações aritméticas/lógicas, elemento a elemento, em um vetor ou entre dois vetores. Visto que cópias de dados e operações em um único vetor realizam uma quantia similar de acessos à memória, ambos apresentam resultados similares com nosso mecanismo. Assim, não apresentamos resultados referentes a operações em um único vetor. Dessa forma, para avaliar os efeitos das conversões do SAPIVe, esta seção apresenta uma análise dos resultados desse mecanismo em dois casos ideais, os Códigos 5.1 e 5.2.

Código 5.1: Algoritmo para cópia de dados entre dois vetores de tamanho VEC_SIZE. Consiste de um laço que realiza a cópia, elemento a elemento, do vetor *origem* para o vetor *destino*. Fonte: Autor.

```
1   for (uint64_t i=0; i < VEC_SIZE; ++i) {
2      destino[i] = origem[i];
3   }
```

Código 5.2: Algoritmo para soma vetorial entre dois vetores de tamanho VEC_SIZE. Consiste de um laço que realiza a soma, elemento a elemento, dos vetores $origem1$ e $origem2$, armazenando o resultado de cada operação em uma posição distinta do vetor *destino*. Fonte: Autor.

```
1   for (uint64_t i=0; i < VEC_SIZE; ++i) {
2      destino[i] = origem1[i] + origem2[i];
3   }
```

O primeiro algoritmo avaliado (Código 5.1) consiste em um laço realizando cópia de memória entre dois vetores, e será referenciado como **memcpy** no restante deste trabalho. Já o segundo (Código 5.2), carrega dados de dois vetores e os soma, armazenando o resultado em memória, sendo denominado **vecsum**. Visto que nenhuma dessas aplicações apresenta reúso de dados, ambas são casos ideais para o processamento em memória, já que esse tipo de processamento evita que dados não reutilizados precisem trafegar por interconexões até a CPU e sejam instalados na hierarquia de caches. Além disso, diferentemente de uma aplicação real, esses casos ideais apresentam apenas um laço, dedicado somente às operações que podem ser convertidas. Isso simplifica as conversões do nosso mecanismo, evitando invalidações e aumentando o impacto de seus ganhos, que seriam menos significativos em aplicações reais, visto que apenas partes dos códigos poderiam ser convertidas.

Para os experimentos desta seção, os algoritmos foram compilados em duas versões, uma utilizando AVX-256 e, a outra, AVX-512. Para isso, usamos as configurações de compilação apresentadas na Tabela 5.1.

Como os algoritmos implementados não fazem parte de uma aplicação real, o resultado das operações deles não é utilizado em outras partes do código. Assim, o uso da *flag* de compilação $-O3$ identifica esses resultados desnecessários e os remove do binário, inviabilizando nosso experimento. Para evitar esse comportamento, utilizamos a *flag* $-O2$, que além de não realizar a remoção, ainda permite o uso das instruções AVX necessárias para nossos experimentos.

Além disso, para garantir resultados consistentes, em todos os experimentos, sobre cada combinação de algoritmo e versão do AVX, os mesmo endereços de acesso à memória foram utilizados (mesmos endereços de alocação para cada vetor).

Tabela 5.1: Configurações de compilação das aplicações ideais **memcpy** e **vecsum**. Cada versão de AVX exigiu o uso de um conjunto diferente de *flags*, além disso, limitamos as otimizações do compilador à *flag* $-O2$, visto que o uso da *flag* $-O3$ causou a remoção de iterações não utilizadas de nossos experimentos. Fonte: Autor.

Versão AVX	***Flags* de compilação**
AVX-256	-ftree-loop-vectorize -O2 -DNOINLINE -static -mavx2 -mno-avx512f -march=native
AVX-512	-ftree-loop-vectorize -O2 -DNOINLINE -static -mavx512f -march=native
Compilador	gcc 7.5.0

Para os experimentos, avaliamos os efeitos do uso de instruções VIMA de diversos tamanhos. O menor tamanho utilizado foi 256 B, visto que é o menor tamanho disponível na arquitetura de memória 3D considerada (como apresentado na Seção 2.4). Já o maior foi 1024 B, visto que tamanhos maiores não seriam possíveis devido ao tamanho de ROB avaliado (apresentado na Tabela 2.2). Isso, porque o SAPIVe precisa armazenar todas as instruções convertidas no ROB antes de completar a conversão. Ao realizarmos conversões de instruções AVX-256 para instruções VIMA de tamanho 1024 B, 32 iterações são armazenadas (1024/32). Considerando que cada iteração ocupa em torno de 10 entradas do ROB, isso já pode exceder o tamanho do ROB de 224 entradas definido na Seção 2.5. Ao aumentarmos ainda mais o tamanho da operação VIMA, nem mesmo as conversões de AVX-512 poderiam ser realizadas, o que definiu nosso limite.

Visto que existem diversos parâmetros que contribuem para os ganhos e perdas de nosso mecanismo, nossas avaliações sobre essas aplicações ideais são apresentadas em etapas. Primeiramente avaliamos o impacto da conversão de instruções AVX-256 e AVX-512 para instruções VIMA-256 na Seção 5.1.1. Com isso, buscando entender qual versão é mais adequada para as conversões realizadas pelo SAPIVe.

Em seguida, na Seção 5.1.2, avaliamos os ganhos obtidos durante a conversão para diversos tamanhos de instrução VIMA, em busca da conversão mais eficiente nas aplicações avaliadas.

Com os melhores resultados dos experimentos anteriores, mostramos a relação entre ganhos e perdas de nosso mecanismo, utilizando diferentes agressividades, em relação a laços de diferentes tamanhos na Seção 5.1.3.

Além disso, comparamos os ganhos do SAPIVe em relação ao uso manual de rotinas *intrinsic* da VIMA na Seção 5.1.4.

Por fim, na Seção 5.1.5, extrapolamos algumas configurações do sistema avaliado para entender como o aumento no tamanho dos *buffers* e da largura dos estágios do *pipeline* simulado afetariam nossos resultados.

Outros resultados relacionados podem ser encontrados no Apêndice A.

5.1.1 Efeito do tamanho da instrução AVX

Para a formação de cada instrução VIMA, pelo SAPIVe, um número suficiente de instruções AVX, equivalente à instrução VIMA precisa ser acumulado no ROB. Quanto maior o tamanho dos dados processados por cada instrução AVX, menor o número de iterações AVX

necessárias para a conversão, como apresentado na Tabela 5.4. Por conta disso, a conversão de instruções AVX-512 para instruções VIMA exige apenas metade do número total de iterações exigido por instruções AVX-256.

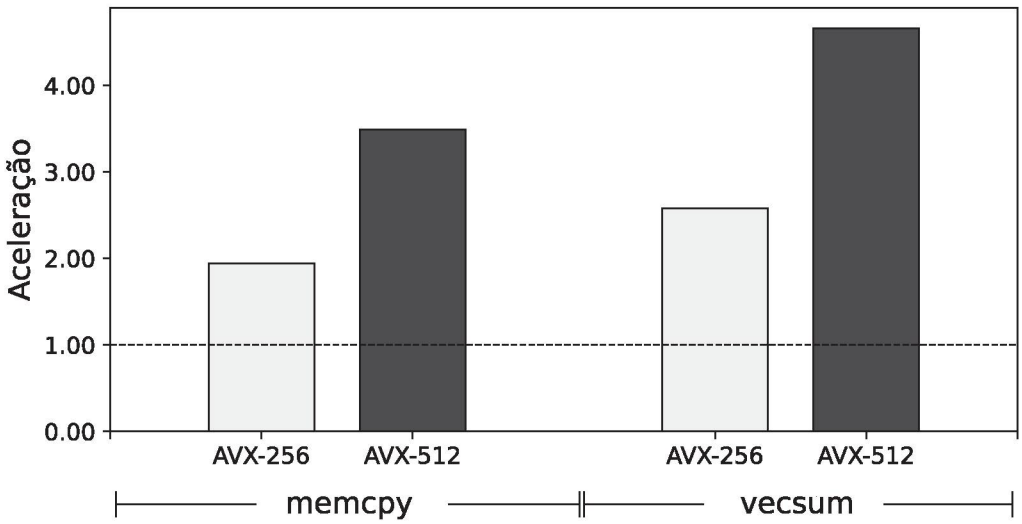


Figura 5.1: Acelerações obtidas pelo SAPIVe sobre os algoritmos de cópia de memória (memcpy) e soma vetorial (vecsum), apresentados nos Códigos 5.1 e 5.2, respectivamente. Cada algoritmo foi compilado em duas versões, uma com AVX-256 e outra com AVX-512. Ambas são convertidas pelo SAPIVe para instruções VIMA-256 B. Fonte: Autor.

A Figura 5.1 apresenta a aceleração obtida pelo SAPIVe ao realizar a conversão de instruções AVX-256 e AVX-512 para o menor tamanho de instrução VIMA (256 B). À direita estão dispostos os resultados da soma vetorial (vecsum), enquanto à esquerda os resultados da cópia de memória (memcpy). Os resultados são apresentados em relação ao tempo de execução usual em CPU das aplicações, sem nenhuma conversão para instruções VIMA, representado em 1. Acelerações maiores que 1 indicam uma redução no tempo de execução da aplicação com o uso do SAPIVe. Por outro lado, acelerações inferiores a 1 indicam degradações no tempo de execução. Cada aplicação foi executada utilizando vetores de 8 MB, o suficiente para os ganhos e perdas do SAPIVe estabilizarem. Para esse experimento, utilizamos uma agressividade de 8 conversões adiantadas, visto que potências de 2 maiores tendem a ter efeitos negativos sobre nosso mecanismo, por ocuparem muitas entradas no ROB, atrasando confirmações de conversões, como apresentado no Apêndice A.

O uso de instruções AVX-512 para as conversões permitiu praticamente dobrar os ganhos obtidos por nosso mecanismo. Isso, porque o uso de instruções AVX sobre dados maiores leva cada conversão a utilizar um número menor de entradas do ROB. Assim, para um mesmo tamanho de ROB, praticamente o dobro de conversões paralelas podem ser realizadas.

Com isso, o SAPIVe é capaz de confirmar simultaneamente um número maior de conversões na CPU, além de aproveitar melhor o paralelismo existente na memória 3D utilizada pela VIMA.

5.1.2 Efeito do tamanho da instrução VIMA

O uso de instruções VIMA maiores permite operações mais eficientes na VIMA, reduzindo o tráfego de dados entre a CPU e a memória e aproveitando o paralelismo interno

da VIMA. Porém, instruções VIMA maiores também exigem um acúmulo de mais iterações AVX no ROB para cada conversão, por exemplo, são necessárias 8 iterações AVX-256 para gerar uma instrução VIMA-256 B, enquanto seriam necessárias 16 iterações para gerar uma instrução VIMA-512 B. Esse maior número de iterações reduz a possibilidade de conversões paralelas e aumenta as perdas causadas por falhas, visto que mais instruções que estavam acumuladas precisam ser executadas, o que é detalhado no Apêndice A.

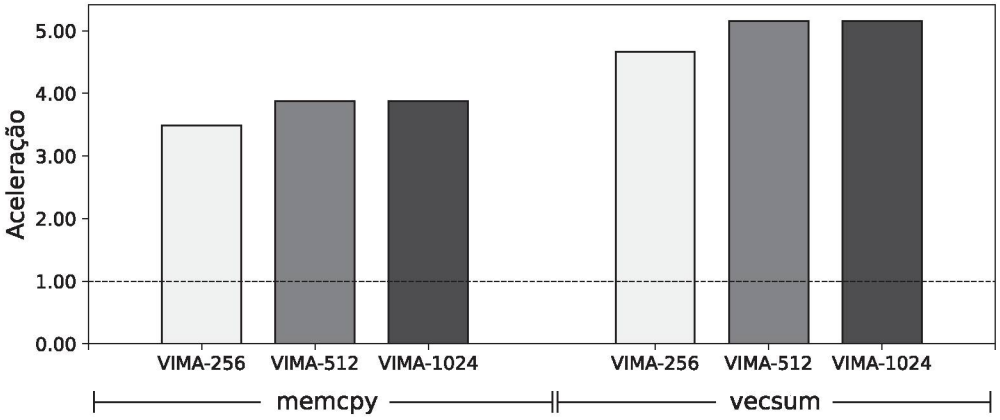


Figura 5.2: Acelerações obtidas pelo SAPIVe sobre os algoritmos de cópia de memória (memcpy) e soma vetorial (vecsum), apresentados nos Códigos 5.1 e 5.2, respectivamente. Foi utilizada a versão compilada com instruções AVX-512 de cada algoritmo, visto que essa versão obteve os melhores resultados na Seção 5.1.1. Ambos os algoritmos foram convertidos pelo SAPIVe para instruções VIMA-256 B, 512 B e 1024 B. Fonte: Autor.

Considerando o melhor tamanho de instrução AVX para nossas conversões (AVX-512), a Figura 5.2 apresenta os ganhos de desempenho obtidos pelo SAPIVe durante a conversão para VIMA-256 B, VIMA-512 B e VIMA-1024 B. À direita estão dispostos os resultados da soma vetorial (vecsum), enquanto à esquerda os resultados da cópia de memória (memcpy). Os resultados são apresentados em relação ao tempo de execução usual em CPU das aplicações, sem nenhuma conversão para instruções VIMA, representado em 1. Acelerações maiores que 1 indicam uma redução no tempo de execução da aplicação com o uso do SAPIVe. Por outro lado, acelerações inferiores a 1 indicam degradações no tempo de execução. Cada aplicação foi executada utilizando vetores de 8 MB, o suficiente para os ganhos e perdas do SAPIVe estabilizarem. Para esse experimento, utilizamos uma agressividade de 8 conversões adiantadas, visto que potências de 2 maiores tendem a ter efeitos negativos sobre nosso mecanismo, por ocuparem muitas entradas no ROB, atrasando confirmações de conversões, como apresentado no Apêndice A.

Observamos que, para ambas as aplicações, o aumento no tamanho da conversão não gerou diferenças significativas de desempenho entre o uso de instruções VIMA de 512 e 1024 bytes, levando a variações menores que 1% entre os resultados. Isso ocorre por conta do custo da confirmação de cada conversão, limitações no número de conversões paralelas, devido ao tamanho do ROB e pelo tamanho dos dados operados por cada instrução VIMA.

Como a agressividade utilizada é a mesma para todos os tamanhos de instrução VIMA, conversões agressivas para VIMA-256 B permitem um processamento paralelo de apenas metade dos dados operados por instruções VIMA-512 B. Isso limita os dados processados pela VIMA, não extraindo todo o potencial desse dispositivo.

Com o aumento para instruções VIMA-512 B, o SAPIVe é capaz de enviar o dobro dos dados, agressivamente, de maneira simultânea para o processamento na VIMA. Porém, visto que

o dobro de entradas no ROB são utilizadas por cada conversão, apenas metade do número de conversões pode ser confirmado paralelamente.

Quando o tamanho das instruções VIMA chega a 1024 B, o ROB simulado de 224 entradas (ver Tabela 2.2) suporta a confirmação de apenas uma conversão por vez. Assim, só quando todas as instruções dessa conversão são confirmadas, sendo adicionadas ao ROB e tendo o stride das instruções de acesso à memória avaliados, suas entradas no ROB são liberadas para a próxima conversão. Isso mantém muitas entradas paradas no ROB, o que atrasa a confirmação de cada conversão, degradando o desempenho do sistema.

Por conta disso, mesmo aproveitando de maneira mais eficiente o processamento da VIMA, conversões para instruções VIMA-1024-B não são capazes de atingir ganhos maiores que conversões para instruções VIMA-512 B. Visto que essas duas conversões apresentam ganhos parecidos, sendo que o uso de instruções VIMA-512 B exige menos iterações por conversão, utilizamos a conversão para instruções VIMA-512 como o padrão adotado para as próximas seções.

5.1.3 Efeito da agressividade e do tamanho do laço

A agressividade do SAPIVe determina quantas conversões especulativas ele adianta simultaneamente. Uma agressividade 0 indica que apenas as conversões com alguma iteração no ROB terão uma instrução correspondente executando na VIMA. Por outro lado, uma agressividade N indica que, além das conversões com alguma iteração no ROB, N conversões futuras são adiantadas, tendo uma instrução VIMA em execução.

Conforme a agressividade aumenta, o SAPIVe é capaz de utilizar melhor o paralelismo interno da memória 3D. Por outro lado, essa maior agressividade exige que um número maior de entradas no ROB seja dedicado a instruções VIMA de conversões futuras, reduzindo o espaço disponível para a confirmação das conversões atuais.

Além da agressividade utilizada, o número de iterações presente no laço convertido também tem um grande efeito sobre nossos ganhos. Laços pequenos, com um número de iterações insuficiente para o início de nossas conversões, não são afetados pelo SAPIVe. Já laços com tamanho suficiente para o início das conversões podem apresentar perdas de desempenho devido à inserção da instrução VIMA correspondente à conversão e à execução tardia de instruções correspondentes a uma conversão incorreta. Assim, levando a degradações que chegam a 25% do tempo de execução do laço, como apresentado no Apêndice A. Além disso, conforme o tamanho dos laços aumenta, essas perdas são intensificadas ou compensadas, pelo efeito das conversões.

A fim de avaliar o impacto desses dois fatores, as Figuras 5.3 e 5.4 apresentam a aceleração obtida utilizando diferentes agressividades em laços com um número variado de iterações, sobre os algoritmos de cópia de memória (memcpy) e soma vetorial (vecsum), respectivamente. Os resultados são apresentados em relação ao tempo de execução usual em CPU das aplicações, sem nenhuma conversão para instruções VIMA, representado em 1. Acelerações maiores que 1 indicam uma redução no tempo de execução da aplicação com o uso do SAPIVe. Por outro lado, acelerações inferiores a 1 indicam degradações no tempo de execução. As aplicações avaliadas foram compiladas com instruções AVX-512, visto que essa versão gerou os melhores resultados com nosso mecanismo. Além disso, consideramos conversões para VIMA-512 B, visto que essa versão de instruções VIMA apresenta resultados similares aos obtidos com VIMA-1024 B, exigindo um menor número de iterações para cada conversão.

Quanto maior o número de iterações em um laço, mais significativos os ganhos obtidos por nosso mecanismo. Isso, porque a cada conversão bem sucedida, nosso mecanismo acelera a execução da aplicação, compensando o tempo até o início das conversões e a degradação inicial

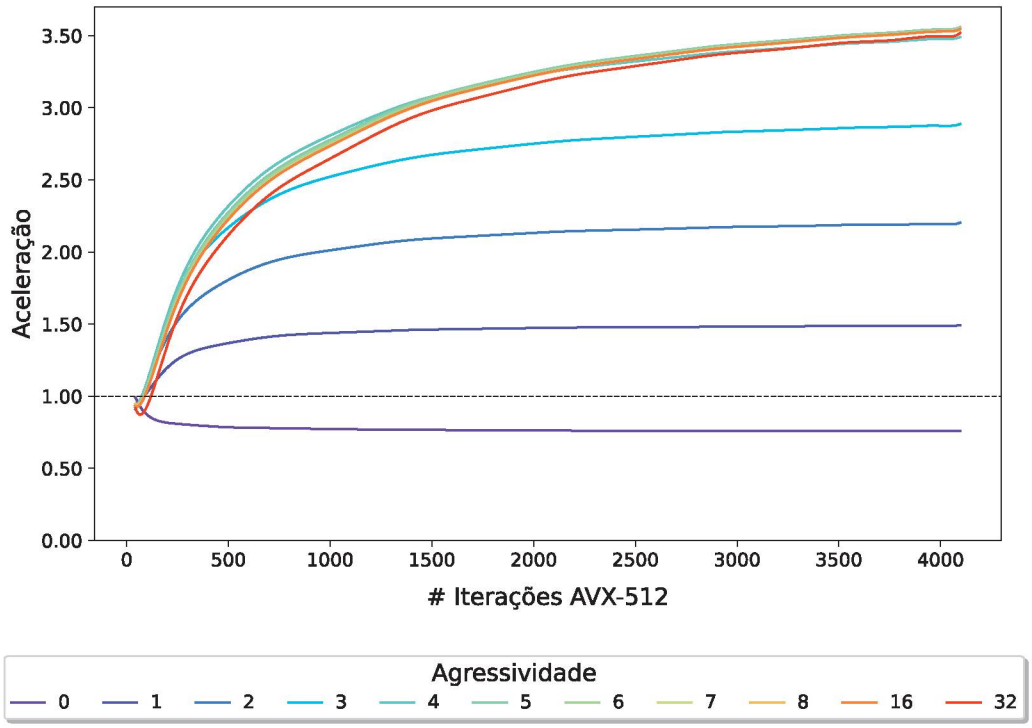


Figura 5.3: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-512 B, utilizando diversas agressividades em laços de tamanho crescente (horizontal). Os resultados são apresentados em relação a uma execução convencional em CPU. Fonte: Autor.

da execução causada pelo mecanismo. Porém, após certo número de conversões, esses ganhos estabilizam.

Um comportamento parecido pode ser observado com a agressividade. Conforme ela aumenta, os ganhos de desempenho do SAPIVe são intensificados, devido a um melhor uso do potencial existente na VIMA. Apesar disso, após certa agressividade, passamos a perder desempenho em ambas as aplicações, o que acontece porque o grande número de entradas no ROB dedicadas a armazenar os adiantamentos de conversões passa a dificultar a confirmação das conversões atuais.

5.1.4 Comparação com implementação manual

Como apresentado na Seção 5.1.3, nosso mecanismo é limitado tanto pela agressividade utilizada, quanto pelo tempo de processamento das conversões pela CPU. Nesta seção, avaliamos como o SAPIVe se comporta quando comparado ao uso manual das instruções VIMA, por meio de rotinas *intrinsic*. Para isso, examinamos a aceleração obtida com o uso de nosso mecanismo e com o uso de *intrinsics*, em relação à execução usual na CPU das aplicações avaliadas. As aplicações avaliadas foram compiladas com instruções AVX-512, visto que essa versão gerou os melhores resultados com nosso mecanismo. Além disso, consideramos conversões para VIMA-512 B, visto que essa versão de instruções VIMA apresenta resultados similares aos obtidos com VIMA-1024 B, exigindo um menor número de iterações para cada conversão.

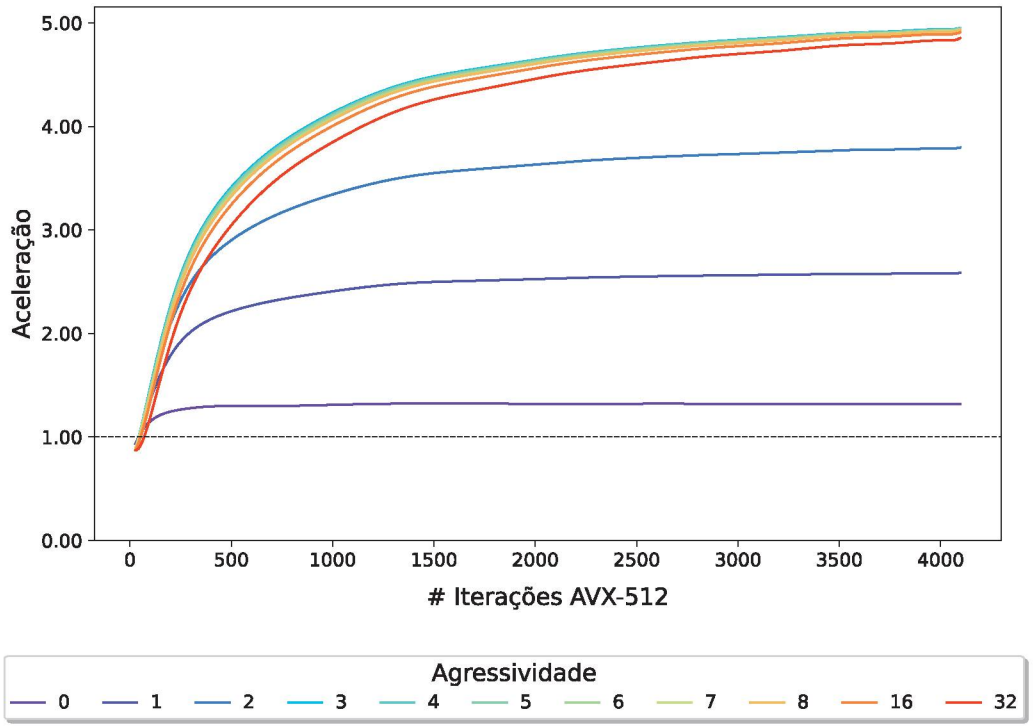


Figura 5.4: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-512 B, utilizando diversas agressividades em laços de tamanho crescente (horizontal). Os resultados são apresentados em relação a uma execução convencional em CPU. Fonte: Autor.

Nossos testes consideraram operações sobre vetores de 8 MB, tamanho suficiente para nosso mecanismo e as *intrisics* atingirem seus ganhos ou perdas máximas de desempenho.

A Figura 5.5 apresenta os resultados obtidos. Em cima estão dispostos os resultados da cópia de memória (memcpy), enquanto, em baixo, os resultados da soma vetorial (vecsum). Os resultados são apresentados em relação ao tempo de execução usual em CPU das aplicações, sem nenhuma conversão para instruções VIMA, representado em 1. Acelerações maiores que 1 indicam uma redução no tempo de execução da aplicação com o uso do SAPIVe. Por outro lado, acelerações inferiores a 1 indicam degradações no tempo de execução.

Assim como apresentado em resultados anteriores, após certa agressividade os resultados do SAPIVe tendem a manter ou piorar seus ganhos, o que ocorre com as agressividades 5 e 3, nas aplicações avaliadas. Apesar disso, o melhor resultado de nosso mecanismo tem ganhos próximos à metade dos obtidos com a implementação manual utilizando rotinas *intrinsic* da VIMA. Isso ocorre pela necessidade de confirmação das conversões realizadas por ele, limitada pelo tamanho do ROB. Devido a essa limitação, o SAPIVe não é capaz de utilizar completamente o potencial presente na VIMA.

5.1.5 Efeito de aprimoramentos no sistema

As conversões realizadas pelo SAPIVe exigem a retenção de instruções no *pipeline*. Isso limita o desempenho de nossas conversões, visto que apenas um pequeno número delas pode ser mantido no ROB e verificado simultaneamente.

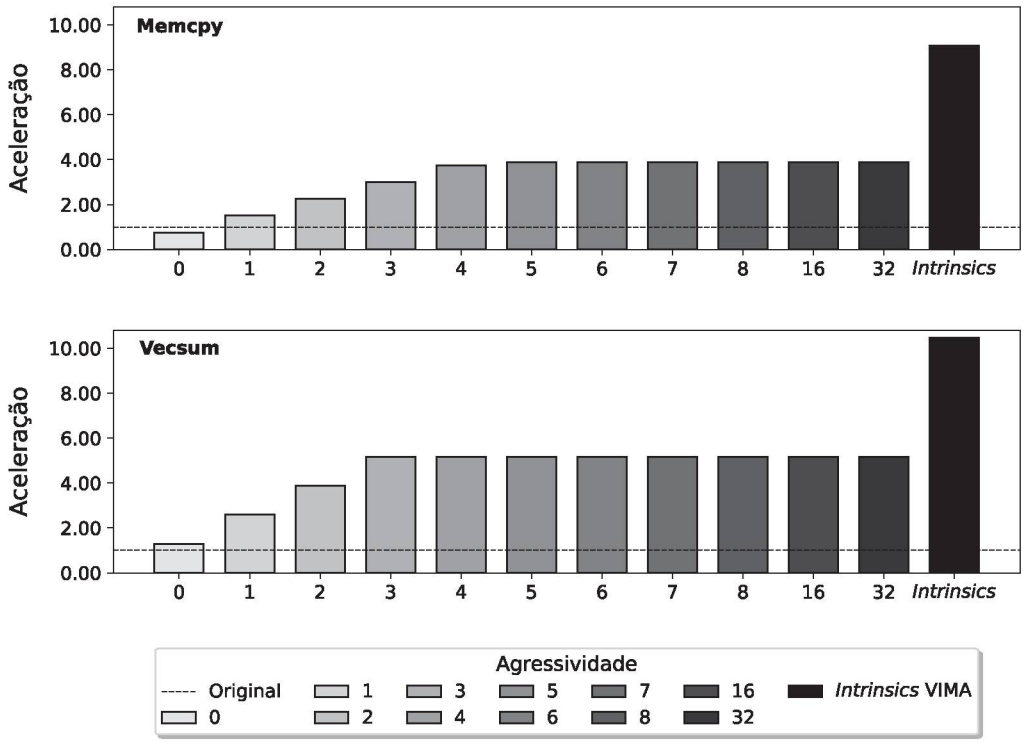


Figura 5.5: Aceleração obtida pelo SAPIVe e por uma implementação manual, utilizando rotinas *intrisic*, dos algoritmos de cópia de memória (memcpy) e soma entre vetores (vecsum), apresentados nos Códigos 5.1 e 5.2. Ambos os algoritmos foram compilados com instruções AVX-512, sendo utilizadas instruções VIMA-512 B para as conversões do SAPIVe e para a implementação manual. Os resultados são apresentados em relação ao tempo de execução convencional em CPU. Os algoritmos avaliados processam vetores de 8 MiB, tamanho suficiente para os ganhos em aceleração da implementação manual e de nosso mecanismo estabilizarem. Fonte: Autor.

Apesar disso, recentes versões de processadores têm mostrado uma expansão em suas estruturas. A Tabela 5.2 apresenta um vislumbre dessa tendência, exibindo o tamanho do ROB em algumas arquiteturas da Intel, de acordo com seu ano de lançamento.

Tabela 5.2: Diferentes arquiteturas de processadores da Intel, seu ano de lançamento e número de entradas presente no ROB. Dados retirados de Jon Stokes (2008), David Kanter (2010), David Kanter (2012), Ian Cutress (2019) e Ian Cutress and Andrei Frumusanu (2021). Notamos que, conforme o tempo passa, o número de entradas tende a aumentar, o que pode indicar sistemas com estruturas ainda maiores no futuro.

Arquitetura	**Ano**	**Tamanho do ROB**
Nehalem	2008	128
Sandy Bridge	2011	168
Haswell	2013	192
Skylake	2018	224
Sunny Cove	2019	352
Golden Cove	2021	512

Para avaliar os efeitos que essa constante melhoria teria sobre o SAPIVe, aumentamos o tamanho de todos os *buffers* do sistema e a largura de todos os estágios do *pipeline* em quatro vezes. Em nossas avaliações, esse aumento não gerou alterações significativas no tempo de execução do sistema base (sem nosso mecanismo de conversão), o que indica um gargalo em

outras partes do sistema, para aplicações limitadas pelo tempo de acesso à memória, porém esse estudo foge ao escopo desta dissertação.

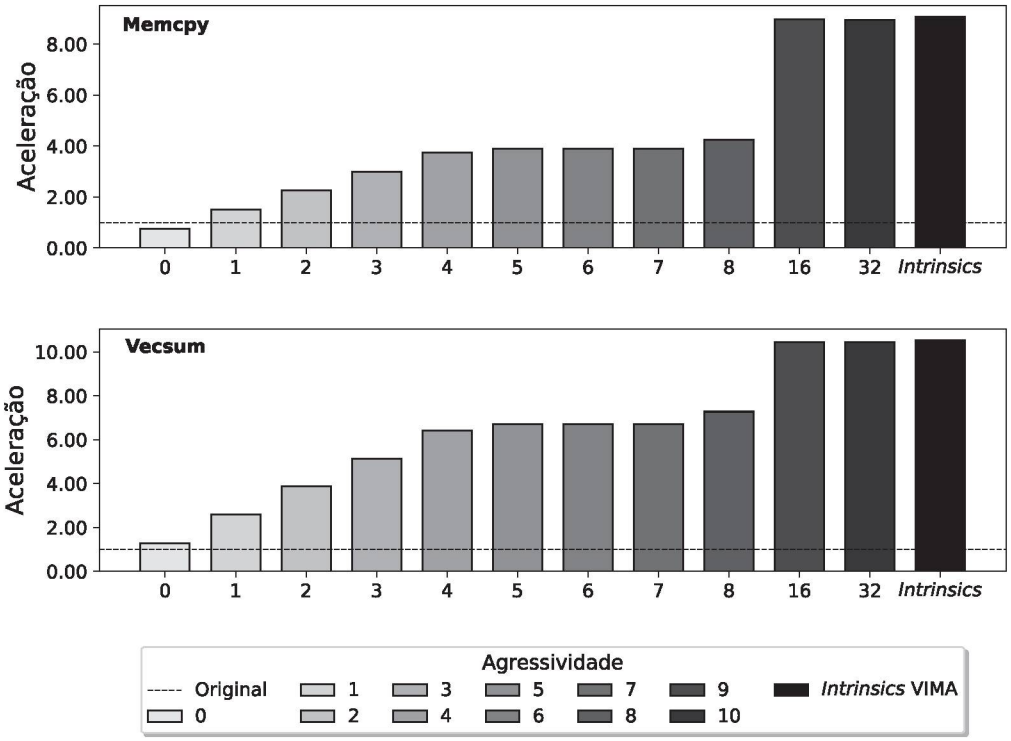


Figura 5.6: Aceleração obtida pelo SAPIVe e por uma implementação manual, utilizando rotinas *intrisic*, dos algoritmos de cópia de memória (memcpy) e soma entre vetores (vecsum), apresentados nos Códigos 5.1 e 5.2. Ambos os algoritmos foram compilados com instruções AVX-512, sendo utilizadas instruções **VIMA-512 B** para as conversões do SAPIVe e para a implementação manual. Os resultados são apresentados em relação ao tempo de execuções convencionais em CPU. Os algoritmos avaliados processam vetores de 8 MiB, tamanho suficiente para os ganhos em aceleração da implementação manual e de nosso mecanismo estabilizarem. Os resultados apresentados consideram um sistema aprimorado, com *buffers* e estágios quatro vezes maiores. Fonte: Autor.

As Figuras 5.6 e 5.7 mostram os ganhos de desempenho do SAPIVe e do uso manual de rotinas *intrinsic* VIMA nesse sistema aprimorado, utilizando instruções VIMA-512 B e VIMA-1024 B, respectivamente. Os resultados são apresentados em relação ao tempo de execução usual em CPU das aplicações, sem nenhuma conversão para instruções VIMA, representado em 1. As aplicações avaliadas foram compiladas com instruções AVX-512, visto que essa versão gerou os melhores resultados com nosso mecanismo. Além disso, consideramos conversões para VIMA-512 B e VIMA-1024 B, visto que, devido ao tamanho de ROB utilizado anteriormente, essas versões de instruções VIMA apresentaram resultados similares na análise da Seção 5.1.2.

Apesar da expansão das estruturas do *pipeline* não ser capaz de aumentar os ganhos de desempenho obtidos pelo uso de rotinas *intrinsic* VIMA, ela permitiu que os ganhos de desempenho do SAPIVe alcançassem aqueles obtidos pelas *intrinsics*.

De maneira parecida com o apresentado na Seção 5.1.3, a conversão de instruções da cópia de memória (memcpy) para VIMA-1024 B apresenta degradações no desempenho a partir da agressividade 7. Isso pode ser causado pela interferência das instruções VIMA adicionais na capacidade do ROB de conter conversões, reduzindo o paralelismo nas validações de conversão realizadas pela CPU. Apesar disso, comprovar essa teoria exige estudos adicionais

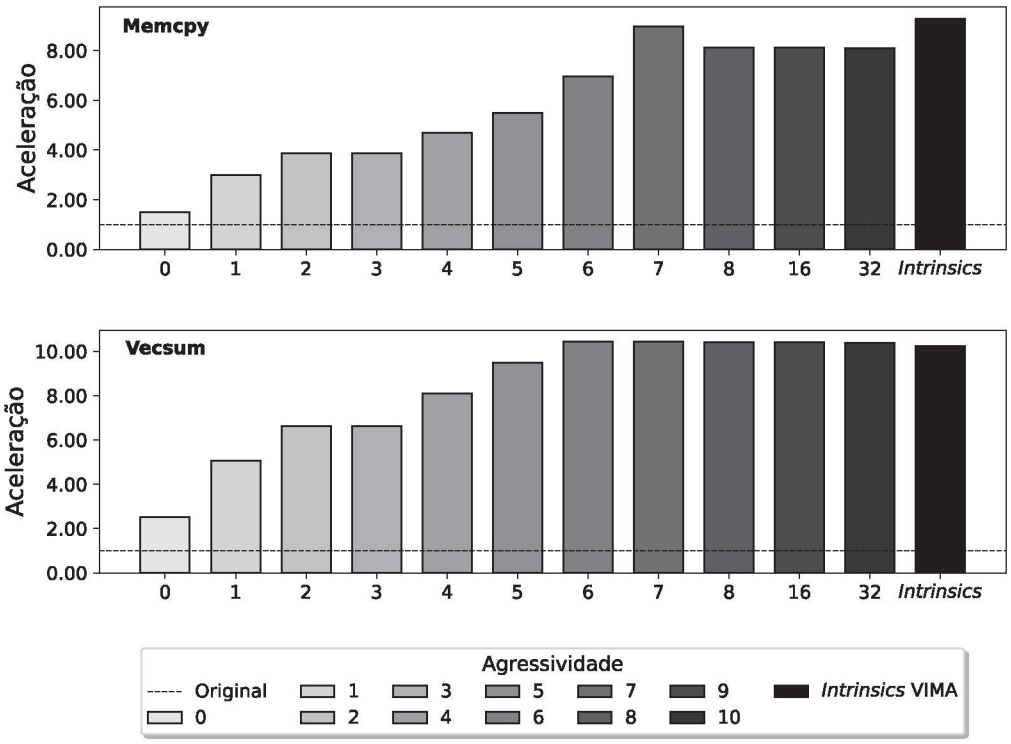


Figura 5.7: Aceleração obtida pelo SAPIVe e por uma implementação manual, utilizando rotinas *intrisic*, dos algoritmos de cópia de memória (memcpy) e soma entre vetores (vecsum), apresentados nos Códigos 5.1 e 5.2. Ambos os algoritmos foram compilados com instruções AVX-512, sendo utilizadas instruções **VIMA-1024 B** para as conversões do SAPIVe e para a implementação manual. Os resultados são apresentados em relação ao tempo de execuções convencionais em CPU. Os algoritmos avaliados processam vetores de 8 MiB, tamanho suficiente para os ganhos em aceleração da implementação manual e de nosso mecanismo estabilizarem. Os resultados apresentados consideram um sistema aprimorado, com *buffers* e estágios quatro vezes maiores. Fonte: Autor.

não englobados nesta dissertação.

5.2 APLICAÇÕES REAIS

Para entender os efeitos do uso do SAPIVe em aplicações reais, o avaliamos sobre alguns *benchmarks* presentes no SPEC-CPU 2017 (Bucek et al., 2018). Esses *benchmarks* foram compilados com o *GNU Compiler Collection* (GCC) na versão 7.4.0, utilizando as *flags* padrão de cada aplicação, além de outras habilitando a vetorização.

Devido a restrições de tempo, em nossas análises utilizamos traços de execução com 200 milhões de instruções, obtidos de algumas aplicações presentes no SPEC-CPU 2017. Esses traços foram gerados utilizando o *framework* PinPoints-3.11 (Patil et al., 2004) presente no instrumentador binário dinâmico Pin (Intel, 2018), da Intel. Esse *framework* é baseado no dispositivo SimPoint (Calder et al., 2005) e permite a seleção de porções mais significativas do código de uma aplicação para a geração de traços. Ao serem simulados, esses traços apresentam um comportamento similar ao da aplicação inicial, porém com um número reduzido de instruções, o que acelera o tempo de simulação. Por conta disso, os resultados apresentados nessa seção são apenas um vislumbre do comportamento real das aplicações avaliadas, servindo de indicativo

para suas características mais marcantes e, por conta disso, relevantes, para um mecanismo de conversões para a VIMA.

Apesar disso, em algumas aplicações, por conta de problemas com o PinPoints para a geração de traços contendo instruções que realizam *Gather* e *Scatter*, restringimos classes específicas de instruções durante a compilação, utilizando as seguintes flags: -O3 -march=native -mno-avx2 -mno-avx512f -mno-avx512pf -mno-avx512vl -fno=unsafe-math-optimizations.

Dessa forma, vetorizações ainda são realizadas, principalmente pelo uso de instruções *Streaming SIMD Extensions* (SSE) e AVX, apesar de isso reduzir o número de oportunidades de conversão para nosso mecanismo, que converte apenas AVX-256 e AVX-512.

A lista de aplicações compiladas com e sem essas restrições é apresentada na Tabela 5.3. Apenas as aplicações pop2 e cactuBSSN não foram avaliadas, devido a problemas na geração de seus traços. Além disso, utilizamos a versão *SPECspeed* das aplicações, visto que ela executa uma cópia de cada *benchmark*, o que é ideal para nossas conversões, limitadas para aplicações *single-thread*, e possui informações suficientes para nossas análises.

Tabela 5.3: Aplicações *SPECspeed* do SPEC-CPU 2017 avaliadas nessa seção. Todas as aplicações foram compiladas habilitando vetorizações até AVX-512. Por conta de problemas durante a geração de traços, a compilação de metade das aplicações precisou ser restringida, inibindo o uso de classes de instruções com *Gaters* e *Scatters*. Fonte: Autor.

Compilação normal	**Compilação com restrições**
bwaves	cam4
exchange2	deepsjeng
imagick	fotonik3d
lbm	gcc
mcf	leela
omnetpp	nab
x264	perlbench
xalancbmk	rooms
xz	wrf

Infelizmente, nessas aplicações, nosso mecanismo não foi capaz de realizar conversões, assim, não obteve ganhos ou perdas de desempenho significativas no tempo de execução. Isso, porque que os laços presentes nessas aplicações não se mostraram adequados para nossas conversões. Logo, esta seção apresenta os principais motivos pelos quais os laços dessas aplicações não puderam ser convertidos, indicando potenciais caminhos a seguir em trabalhos futuros.

As subseções seguintes apresentam características dos laços presentes em algumas aplicações do SPEC-CPU 2017, relevantes para possíveis conversões para IMA, comparando-as com os padrões buscados por nosso mecanismo e suas restrições. Para identificar os laços presentes nos traços avaliados, primeiramente encontramos todas as instruções de salto com destino a um endereço de memória menor ou igual ao do próprio salto. Após descartarmos os saltos correspondentes a chamadas e retornos de rotinas, consideramos as instruções executadas entre cada salto e seu destino como um laço. Em seguida, laços com as mesmas instruções iniciais e finais, ocorrendo sem instruções externas entre eles foram unidos como iterações de um mesmo laço Sendo que, em laços com instruções de controle de fluxo, como a estrutura if/else, consideramos apenas o primeiro caminho tomado durante a execução como o corpo do laço.

Como os traços não representam o fluxo de execução completo de uma aplicação, podem haver saltos inesperados entre instruções, dificultando a correta identificação de laços que executam apenas uma iteração. Devido a isso, selecionamos apenas laços com duas ou

mais iterações. Apesar dessa restrição, consideramos laços com apenas uma instrução, como instruções com o prefixo *Repeat Prefix* (REP), visto que apresentam operações que podem ser executadas em dispositivos IMA, como a cópia de dados entre endereços da memória.

Devido às restrições impostas por nossa compilação, nas próximas seções expandimos nossa busca por padrões de cópia de memória e soma vetorial, não apenas para instruções AVX sobre dados de 256 e 512 bits, mas também para instruções escalares, SSE e AVX de 128 bits.

5.2.1 Tamanho dos laços

Para que um mecanismo consiga converter instruções vetoriais de um laço em instruções VIMA, esse laço precisa de iterações suficientes para compor a instrução VIMA. A Tabela 5.4 apresenta uma relação entre o tamanho dos dados processados por uma instrução de CPU e o número correspondente de iterações que precisam ser acumuladas para sua conversão em diversos tamanhos de instrução VIMA.

Tabela 5.4: Número de iterações necessárias para a formação de cada tipo de instrução VIMA, de acordo com o tamanho dos dados processados por uma instrução da CPU. Fonte: Autor.

	32 bits	64 bits	128 bits	256 bits	512 bits
VIMA-256 B	64	32	16	8	4
VIMA-512 B	128	64	32	16	8
VIMA-1024 B	256	128	64	32	16

Examinamos a porcentagem de laços, presente em cada aplicação avaliada do SPEC-CPU 2017, que se enquadra em cada tipo de conversão. Para isso, separamos os laços, de acordo com seu tipo principal de instrução, em quatro categorias: laços contendo apenas instruções escalares, ou seja, sem nenhuma instrução com operandos de 128 bits ou mais; laços contendo instruções vetoriais, em sua maioria sobre operandos de 128 bits; laços contendo instruções vetoriais, em sua maioria sobre operandos de 256 bits e; laços contendo instruções vetoriais, em sua maioria sobre operandos de 512 bits. Em caso de empate, selecionamos o maior tamanho de dado utilizado. Laços externos e internos são contabilizados separadamente, sendo que as instruções do laço interno não interferem na classificação do laço externo. Cada execução distinta de um mesmo laço é contabilizada separadamente, assim um laço de 10 iterações que é executado 3 vezes durante a aplicação conta como 3 laços distintos de tamanho 10. Visto que nem todas as aplicações avaliadas contêm laços com todos os tamanhos de dados apresentados, as aplicações que não possuem nenhum laço com determinado tipo de instrução não são exibidas no respectivo gráfico.

A Figura 5.8 apresenta a porcentagem de laços, contendo apenas instruções escalares, com menos de 4 iterações, entre 4 e 8 iterações, entre 8 e 16 iterações, entre 16 e 32 iterações, entre 32 e 64 iterações e com mais de 64 iterações. De acordo com a Tabela 5.4, apenas laços de instruções escalares com mais de 32 iterações, considerando instruções sobre dados de 64 bits, poderiam ser convertidos para uma instrução de memória com 256 B, o menor tamanho suportado por nossa arquitetura VIMA. Assim, apenas as aplicações mcf, fotonik3d, leela, wrf, deepsjeng, xalancbmk e gcc apresentam mais de 20% de seus laços podendo ser convertidos.

A Figura 5.9 apresenta a porcentagem de laços contendo instruções vetoriais, em sua maioria sobre operandos de 128 bits, com menos de 4 iterações, entre 4 e 8 iterações, entre 8 e 16 iterações, entre 16 e 32 iterações, entre 32 e 64 iterações e aqueles com mais de 64 iterações. De acordo com a Tabela 5.4, precisamos de mais de 16 iterações para converter essas instruções vetoriais para uma instrução de memória com 256 B. Assim, apenas as aplicações omnetpp,

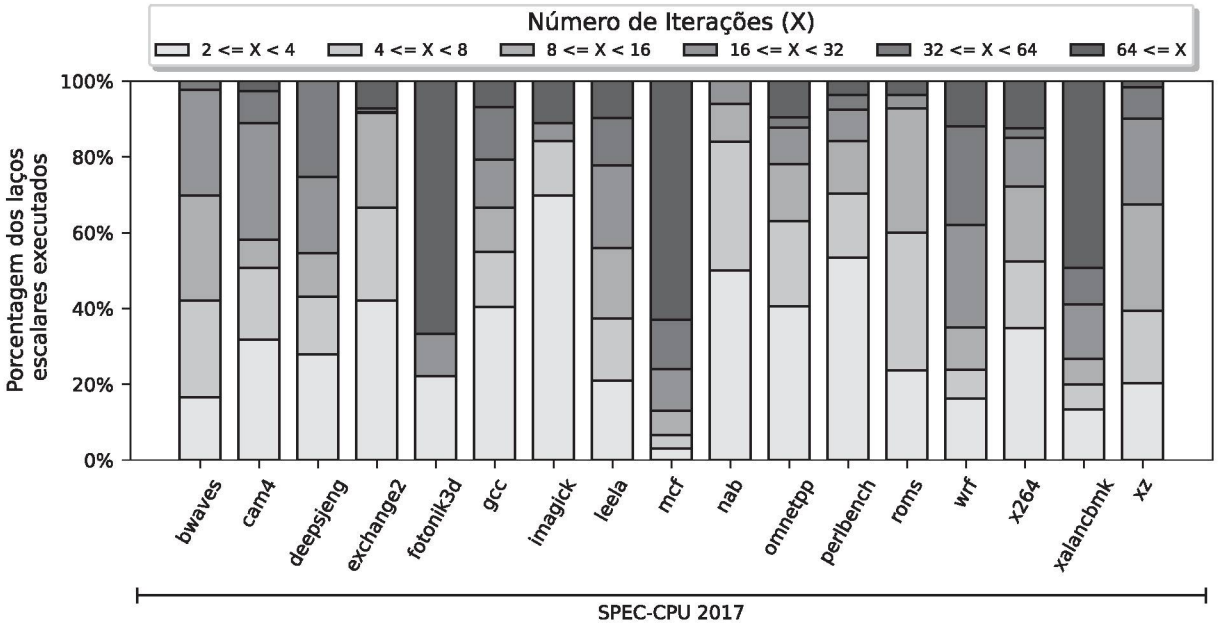


Figura 5.8: Porcentagem de laços, presente em cada aplicação do SPEC-CPU 2017 avaliada, contendo menos de 4, entre 4 e 8, entre 8 e 16, entre 16 e 32, entre 32 e 64 e acima de 64 iterações. Esses valores foram escolhidos por representar o número de iterações necessário para diversos tipos de conversão para VIMA. Os laços apresentados contém apenas instruções escalares, ou seja, não possuem instruções operando sobre dados de 128 bits ou mais. Aplicações sem nenhum laço nessa categoria foram removidas. Fonte: Autor.

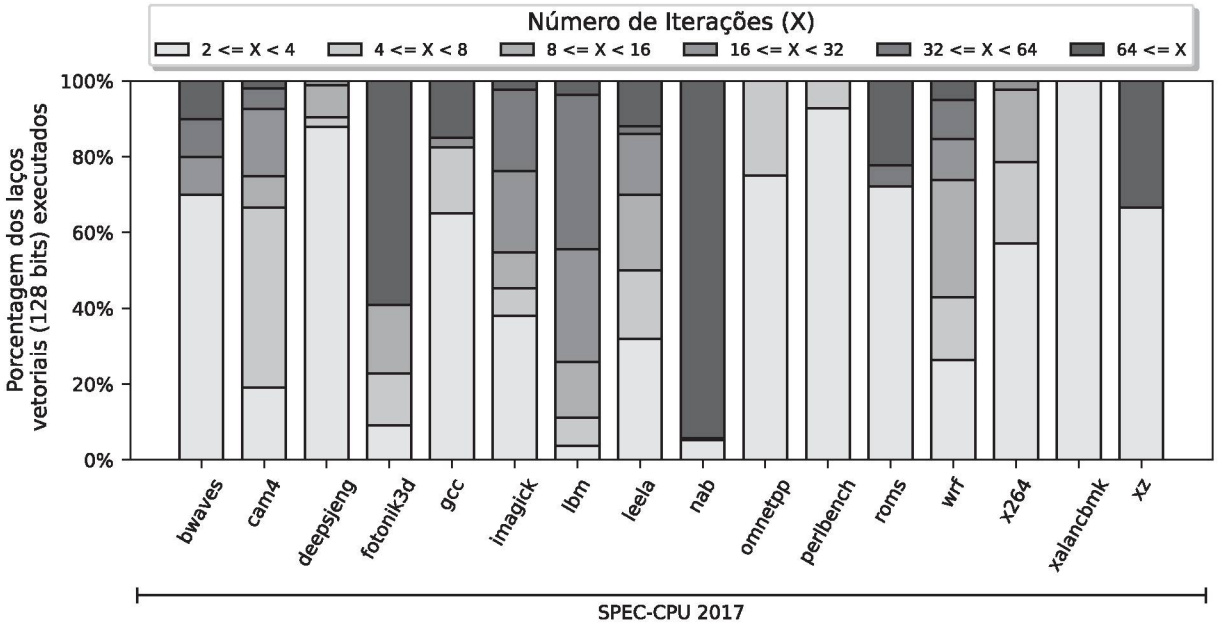


Figura 5.9: Porcentagem de laços, presente em cada aplicação do SPEC-CPU 2017 avaliada, contendo menos de 4, entre 4 e 8, entre 8 e 16, entre 16 e 32, entre 32 e 64 e acima de 64 iterações. Esses valores foram escolhidos por representar o número de iterações necessário para diversos tipos de conversão para VIMA. Os laços apresentados contém ao menos uma instrução vetorial, ou seja, com operandos sobre dados maiores ou iguais a 128 bits, sendo que a maior parte dessas instruções vetoriais opera dados de 128 bits. Aplicações sem nenhum laço nessa categoria foram removidas. Fonte: Autor.

deepsjeng, xalancbmk, perlbench, x264 e gcc não apresentaram oportunidades de conversão para a VIMA-256 B em mais de 20% de seus laços executados, sem considerar as aplicações sem laços deste tipo.

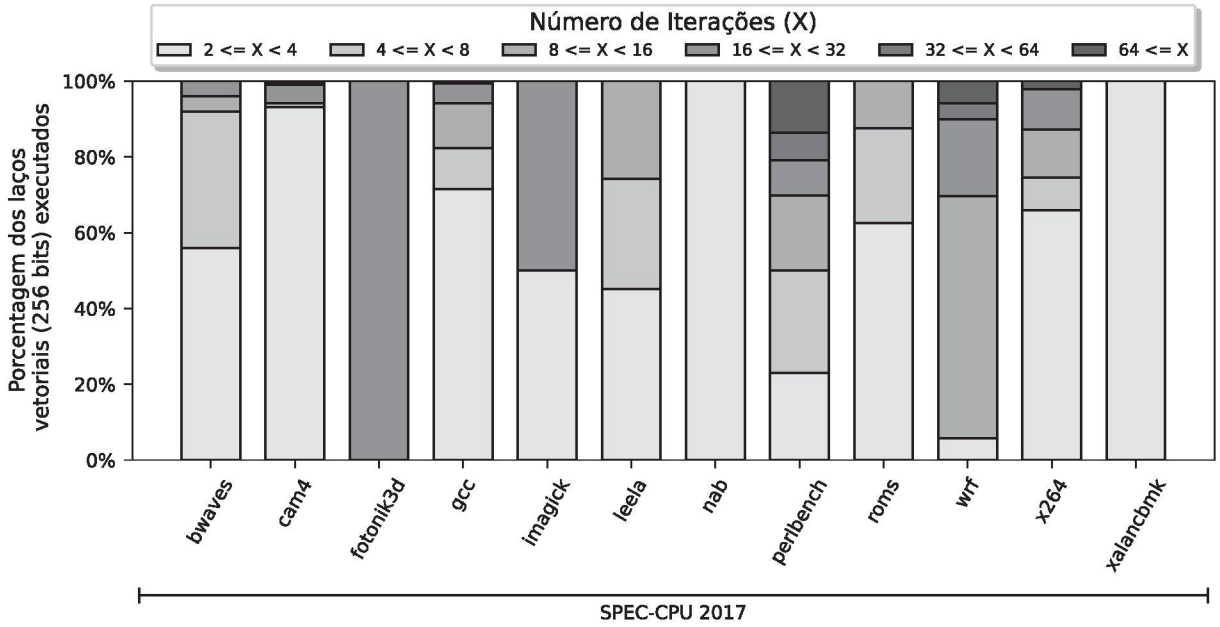


Figura 5.10: Porcentagem de laços, presente em cada aplicação do SPEC-CPU 2017 avaliada, contendo menos de 4, entre 4 e 8, entre 8 e 16, entre 16 e 32, entre 32 e 64 e acima de 64 iterações. Esses valores foram escolhidos por representar o número de iterações necessário para diversos tipos de conversão para VIMA. Os laços apresentados contém ao menos uma instrução vetorial, ou seja, com operandos sobre dados maiores ou iguais a 128 bits, sendo que a maior parte dessas instruções vetoriais opera dados de 256 bits. Aplicações sem nenhum laço nessa categoria foram removidas. Fonte: Autor.

A Figura 5.10 apresenta a porcentagem de laços contendo instruções vetoriais, em sua maioria sobre operandos de 256 bits, com menos de 4 iterações, entre 4 e 8 iterações, entre 8 e 16 iterações, entre 16 e 32 iterações, entre 32 e 64 iterações e aqueles com mais de 64 iterações. De acordo com a Tabela 5.4, precisamos de mais de 8 iterações para converter essas instruções vetoriais para uma instrução de memória com 256 B. Por conta disso, apenas as aplicações bwaves, nab, xalancbmk, roms, gcc e cam4 não apresentam oportunidades de conversão em mais de 20% de seus laços, sem considerar as aplicações sem laços deste tipo.

Por fim, apenas a aplicação x264 apresentou laços com a maior parte das instruções vetoriais operando sobre dados de 512 bits. Mesmo assim, todos os laços apresentados executaram entre 8 e 32 iterações, sendo alvos possíveis para conversão. Além disso, é possível que outras aplicações também apresentem esse tipo de laço, porém, de maneira menos significativa, portanto, não aparecendo em nossos traços.

Para obter uma perspectiva sobre a relevância de cada tipo de laço, nas aplicações avaliadas do SPEC-CPU 2017, o número de execuções de laços com instruções de cada tamanho é apresentado na Figura 5.11. Nesse gráfico, são separados laços escalares, ou seja, que não contém instruções que processam dados maiores ou iguais a 128 bits. Laços vetoriais de 128 bits, que contém a maior parte de suas instruções vetoriais sobre dados de 128 bits. Laços vetoriais de 256 bits, que contém a maior parte de suas instruções vetoriais sobre dados de 256 bits. Além dos laços vetoriais de 512 bits, que contém a maior parte de suas instruções vetoriais sobre dados de 512 bits. Visto que a disparidade entre os dados apresentados é grande, o número de execuções dos laços é apresentado em escala logarítmica. Como apresentado nesses resultados, para a maior parte das aplicações, o número de execuções de laços com instruções vetoriais é bem menor que o de laços sem essas instruções. Devido a isso, nessas aplicações, para que um mecanismo de conversão consiga ganhos significativos de desempenho, os laços escalares podem ser bons alvos de conversão para instruções IMA.

Apesar de algumas aplicações apresentarem condições promissoras para a conversão, com laços grandes e muito executados, a maior parte dessas oportunidades se concentra em laços com instruções que operam sobre dados de até 128 bits, um tamanho insuficiente para as conversões do SAPIVe. Além disso, mesmo em laços com instruções que operam sobre dados maiores, problemas relacionados às instruções adjacentes aos conjuntos que poderiam ser convertidos impedem nossas conversões, o que é explorado na próxima seção.

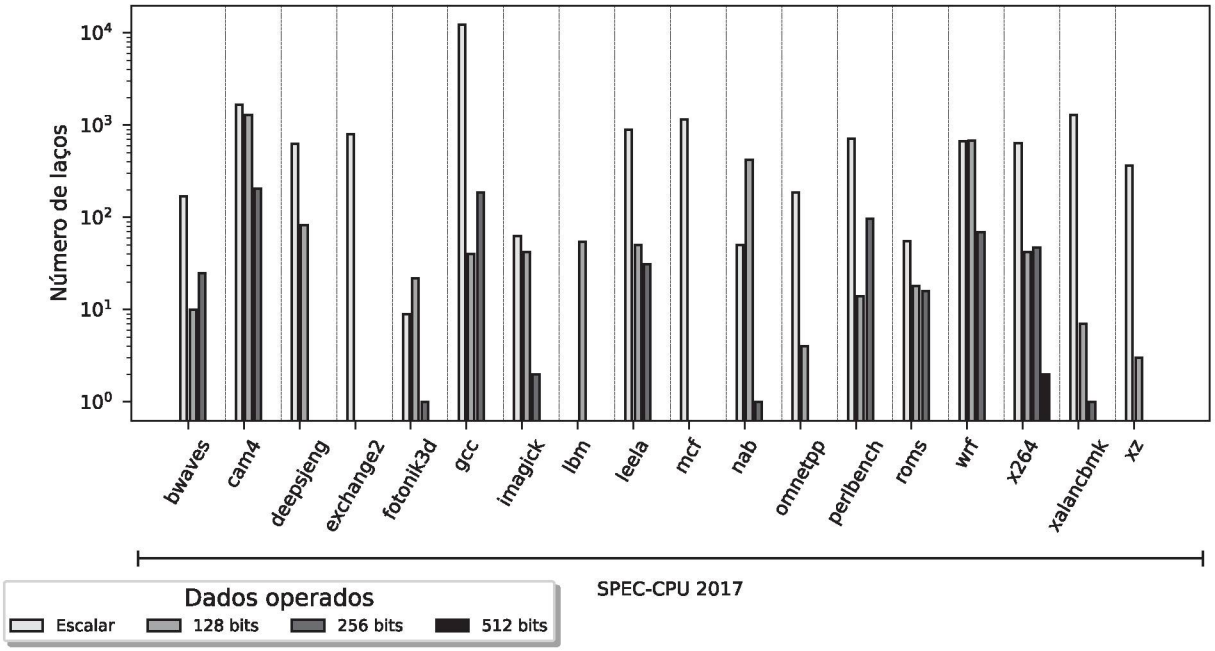


Figura 5.11: Número de laços executados por tipo presente em cada aplicação do SPEC-CPU 2017 avaliada. Os laços foram divididos com base em suas instruções em quatro categorias: laços contendo apenas instruções escalares, ou seja, sem nenhuma instrução com operandos de 128 bits ou mais; laços contendo instruções vetoriais, em sua maioria sobre operandos de 128 bits; laços contendo instruções vetoriais, em sua maioria sobre operandos de 256 bits e; laços contendo instruções vetoriais, em sua maioria sobre operandos de 512 bits. Os resultados são apresentados em escala logarítmica devido à sua grande disparidade. Fonte: Autor.

5.2.2 Inadequações em laços

Para garantir a semântica da execução durante as conversões, o SAPIVe precisa que os laços convertidos apresentem algumas características: (i) um dos padrões de conversão (cópia de

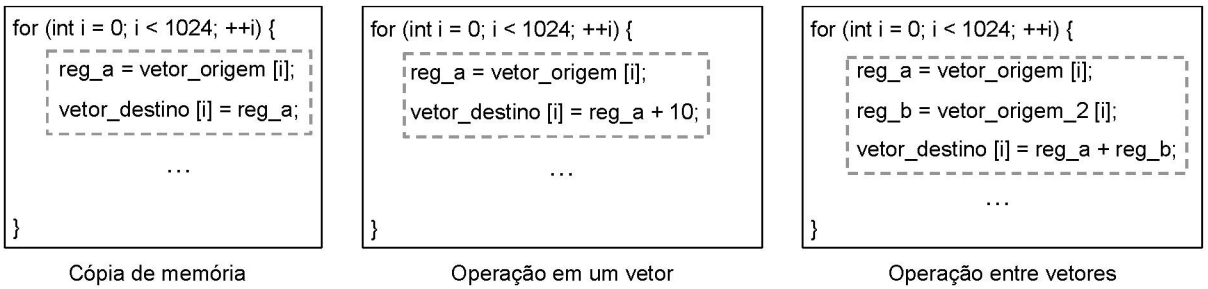


Figura 5.12: Laços apresentando um padrão de cópia de memória e um padrão de operação, elemento a elemento, sobre um e dois vetores. Todos os padrões são alvos de conversão pelo SAPIVe, por possuirem equivalência com instruções VIMA. Fonte: Autor.

memória ou operação, elemento a elemento, sobre um ou dois vetores, ilustrados com instruções

escalares na Figura 5.12), formado por instruções AVX-256 ou AVX-512, deve existir; (ii) nenhuma instrução do laço, fora as que estão sendo convertidas para VIMA, pode acessar a memória, o que acontece de forma incorreta no algoritmo da Figura 5.13; (iii) nenhuma instrução do laço, fora as que estão sendo convertidas para VIMA, pode ler um registrador com um dado escrito por uma instrução da conversão, o que acontece de forma incorreta no algoritmo da Figura 5.14, e; (iv) não podem existir estruturas de controle de fluxo (condicionais, rotinas e laços) entre as instruções convertidas, o que acontece de forma incorreta no algoritmo da Figura 5.15.

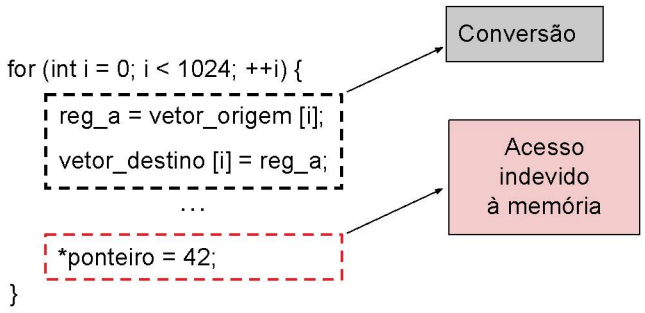


Figura 5.13: Laço contendo um padrão de cópia de memória e outra operação acessando a memória. Devido à operação adicional que acessa a memória, o padrão não pode ser vetorizado para instruções de memória. Isso, porque caso o padrão seja vetorizado e processado em memória, o resultado de diversas de suas iterações utilizaria o valor inicial do "vetor_origem", que pode ter sido alterado no decorrer dessas iterações pelo "ponteiro". Fonte: Autor.

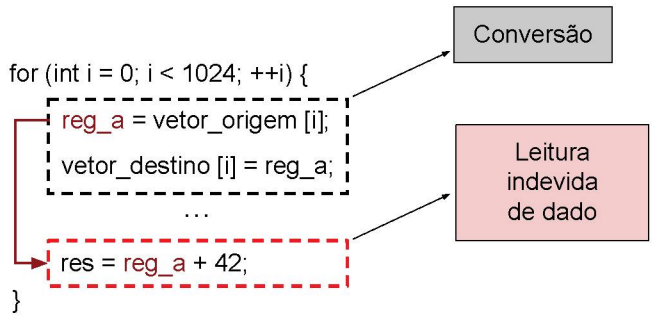


Figura 5.14: Laço contendo um padrão de cópia de memória e outra operação que lê o resultado de uma das instruções pertencentes à cópia de memória. Devido à operação adicional que lê o resultado de uma das instruções do padrão, ele não pode ser vetorizado para instruções de memória. Isso, porque caso o padrão seja executado em memória, o resultado das instruções convertidas não estaria disponível em CPU para as demais instruções do laço. Ou seja, o valor contido no registrador correspondente a "reg_a"estaria desatualizado, resultando em um cálculo incorreto para a soma "res = reg_a + 42". Fonte: Autor.

Por conta disso, laços aninhados e contendo chamadas de rotinas apresentam dificuldades em sua conversão. Nos laços aninhados, as instruções dos laços internos e externos não poderiam ser unidas para formar um padrão para a conversão, visto que seu número de execuções difere. Apesar disso, as instruções dos laços internos podem prejudicar a formação de um padrão no laço externo, visto que poderiam realizar acessos à memória, além de leitura e escrita de registradores. Já as chamadas de rotinas em laços, apesar de permitirem a formação de padrões entre as instruções do laço e da rotina, são um impeditivo para conversões, visto que representam um desvio de fluxo na aplicação e podem implicar em acessos à memória, ao salvar e recuperar o contexto.

As Tabelas 5.8, 5.9 e 5.10, apresentam o número total de padrões, de cópia de memória (CP) e operação, elemento a elemento, sobre um (O1V) ou dois vetores (O2V), formados por

Tabela 5.5: Número de padrões de cópia de memória (CP) que puderam ser formados nos laços presentes nos traços avaliados das aplicações do SPEC-CPU 2017, utilizando instruções que operam sobre dados escalares. Também são apresentados o número total de laços avaliados e a percentagem desses padrões que não poderia ser convertida pela existência de instruções com acesso à memória, instruções lendo registradores escritos pelas convertidas e estruturas de controle de fluxo (chamadas a rotinas, laços internos ou condicionais) no laço. Apenas aplicações com ao menos um padrão são exibidas. Fonte: Autor.

	bwaves	cam4	deepsjeng	exchange2	fotonik3d	gcc	imagick	leela	mcf	nab	omnetpp	perlbench	roms	wrf	x264	xalancbmk	xz
Padrão encontrado	CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	CP
# Padrões encontrados	71	151	1502	148	11	1525	118	459	54	82	41	746	128	132	716	135	109
# Laços avaliados	76	2580	288	570	30	6480	78	482	67	64	82	503	65	819	534	445	169
% Padrões com LOAD não permitido	97%	93%	100%	100%	100%	99%	99%	99%	100%	97%	100%	99%	98%	100%	99%	100%	100%
% Padrões com STORE não permitido	97%	90%	100%	97%	100%	98%	95%	100%	100%	97%	100%	99%	98%	100%	98%	100%	99%
% Padrões com salto não permitido	97%	90%	100%	94%	100%	98%	99%	92%	100%	97%	100%	99%	98%	100%	88%	100%	100%
% Padrões com leitura de registradores da conversão	5%	26%	38%	43%	9%	51%	37%	42%	29%	15%	46%	35%	24%	37%	40%	37%	70%

Tabela 5.6: Número de padrões de operação sobre um vetor (O1V) que puderam ser formados nos laços presentes nos traços avaliados das aplicações do SPEC-CPU 2017, utilizando instruções que operam sobre dados escalares. Também são apresentados o número total de laços avaliados e a percentagem desses padrões que não poderia ser convertida pela existência de instruções com acesso à memória, instruções lendo registradores escritos pelas convertidas e estruturas de controle de fluxo (chamadas a rotinas, laços internos ou condicionais) no laço. Apenas aplicações com ao menos um padrão são exibidas. Fonte: Autor.

	bwaves	cam4	deepsjeng	exchange2	fotonik3d	gcc	imagick	leela	mcf	nab	omnetpp	perlbench	roms	wrf	x264	xalancbmk	xz
Padrão encontrado	O1V	O1V	O1V	O1V	O1V	O1V	O1V	O1V	O1V	O1V	O1V	O1V	O1V	O1V	O1V	O1V	O1V
# Padrões encontrados	6	90	569	260	19	413	22	153	12	2	6	278	5	55	537	46	179
# Laços avaliados	76	2580	288	570	30	6480	78	482	67	64	82	503	65	819	534	445	169
% Padrões com LOAD não permitido	100%	96%	100%	93%	100%	99%	100%	100%	100%	100%	100%	100%	100%	100%	100%	97%	100%
% Padrões com STORE não permitido	100%	94%	100%	90%	100%	92%	100%	100%	100%	100%	100%	99%	100%	100%	99%	100%	98%
% Padrões com salto não permitido	100%	91%	100%	92%	100%	96%	100%	100%	100%	100%	100%	98%	60%	47%	100%	100%	100%
% Padrões com leitura de registradores da conversão	0%	16%	61%	13%	0%	13%	27%	24%	83%	100%	16%	28%	20%	14%	41%	52%	48%

Tabela 5.7: Número de padrões de operação entre dois vetores (O2V) que puderam ser formados nos laços presentes nos traços avaliados das aplicações do SPEC-CPU 2017, utilizando instruções que operam sobre dados escalares. Também são apresentados o número total de laços avaliados e a percentagem desses padrões que não poderia ser convertida pela existência de instruções com acesso à memória, instruções lendo registradores escritos pelas convertidas e estruturas de controle de fluxo (chamadas a rotinas, laços internos ou condicionais) no laço. Apenas aplicações com ao menos um padrão são exibidas. Fonte: Autor.

	bwaves	cam4	deepsjeng	exchange2	fotonik3d	gcc	imagick	leela	mcf	nab	omnetpp	perlbench	roms	wrf	x264	xalancbmk	xz
Padrão encontrado	O2V	O2V	O2V	O2V	O2V	O2V	O2V	O2V	O2V	O2V	O2V	O2V	O2V	O2V	O2V	O2V	O2V
# Padrões encontrados	10	53	325	75	23	93	1	16	14	6	3	7	20	76	509	11	218
# Laços avaliados	76	2580	288	570	30	6480	78	482	67	64	82	503	65	819	534	445	169
% Padrões com LOAD não permitido	100%	100%	100%	100%	100%	94%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
% Padrões com STORE não permitido	100%	100%	100%	92%	100%	93%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
% Padrões com salto não permitido	100%	100%	100%	100%	100%	94%	100%	100%	100%	100%	100%	100%	80%	100%	55%	100%	100%
% Padrões com leitura de registradores da conversão	20%	52%	83%	16%	52%	52%	100%	100%	85%	16%	100%	71%	35%	88%	76%	81%	100%

Tabela 5.8: Número de padrões de cópia de memória (CP) e operação sobre um (O1V) ou dois vetores (O2V) que puderam ser formados nos laços presentes nos traços avaliados das aplicações do SPEC-CPU 2017, utilizando instruções que operam sobre dados de 128 bits. Também são apresentados o número total de laços avaliados e a percentagem desses padrões que não poderia ser convertida pela existência de instruções com acesso à memória, instruções lendo registradores escritos pelas convertidas e estruturas de controle de fluxo (chamadas a rotinas, laços internos ou condicionais) no laço. Apenas aplicações com ao menos um padrão são exibidas. Fonte: Autor.

	bwaves	cam4	deepsjeng	fotonik3d	gcc	imagick	leela	nab	perlbench	roms	roms	wrf	x264	x264
Padrão encontrado	CP	CP	CP	CP	CP	CP	CP	CP	CP	O2V	CP	CP	O1V	CP
# Padrões encontrados	1	22	95	8	39	8	27	2	8	6	4	128	2	61
# Laços avaliados	76	2580	288	30	6480	78	482	64	503	65	65	819	534	534
% Padrões com LOAD não permitido	0%	86%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
% Padrões com STORE não permitido	0%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
% Padrões com salto não permitido	0%	40%	100%	100%	100%	100%	100%	100%	100%	0%	25%	6%	0%	100%
% Padrões com leitura de registradores da conversão	0%	40%	0%	0%	2%	0%	22%	0%	0%	0%	25%	2%	100%	1%

Tabela 5.9: Número de padrões de cópia de memória (CP) e operação sobre um (O1V) ou dois vetores (O2V) que puderam ser formados nos laços presentes nos traços avaliados das aplicações do SPEC-CPU 2017, utilizando instruções que operam sobre dados de 256 bits. Também são apresentados o número total de laços avaliados e a percentagem desses padrões que não poderia ser convertida pela existência de instruções com acesso à memória, instruções lendo registradores escritos pelas convertidas e estruturas de controle de fluxo (chamadas a rotinas, laços internos ou condicionais) no laço. Apenas aplicações com ao menos um padrão são exibidas. Além disso, apenas são apresentados dados relacionados à cópia de memória, visto que não foram encontrados padrões de operações vetoriais, elemento a elemento. Fonte: Autor.

	bwaves	cam4	gcc	imagick	leela	nab	perlbench	roms	wrf	x264	xalancbmk
Padrão encontrado	CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	CP
# Padrões encontrados	60	96	128	4	53	1	22	32	48	30	4
# Laços avaliados	76	2580	6480	78	482	64	503	65	819	534	445
% Padrões com LOAD não permitido	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
% Padrões com STORE não permitido	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
% Padrões com salto não permitido	100%	100%	93%	0%	84%	100%	63%	100%	83%	73%	0%
% Padrões com leitura de registradores da conversão	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

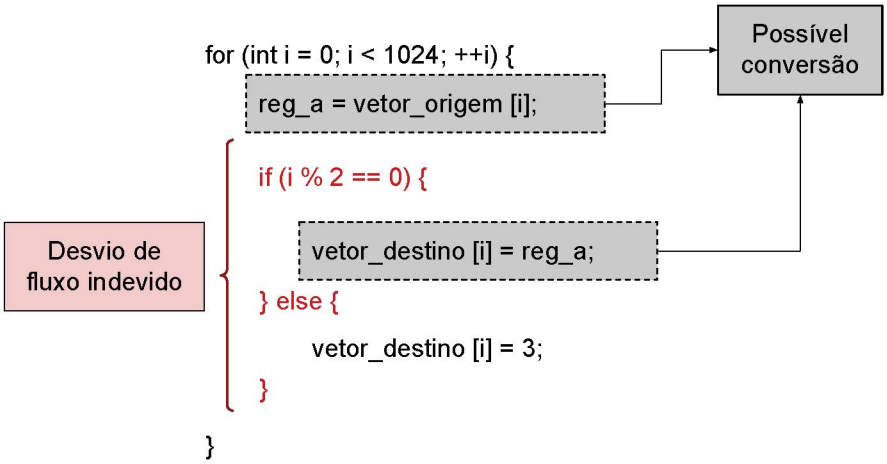


Figura 5.15: Laço contendo um padrão de cópia de memória formado em um dos caminhos de uma instrução condicional. Devido à instrução condicional, esse padrão não pode ser vetorizado para instruções de memória. Isso, porque para que ele seja vetorizado, todas as iterações convertidas precisam ser executadas. Apesar disso, nesse laço, o padrão é executado apenas em suas iterações pares. Assim, caso seja vetorizado para a memória, a instrução de memória resultante calcularia incorretamente o resultado das iterações ímpares. Fonte: Autor.

Tabela 5.10: Número de padrões de cópia de memória (CP) e operação sobre um (O1V) ou dois vetores (O2V) que puderam ser formados nos laços presentes nos traços avaliados das aplicações do SPEC-CPU 2017, utilizando instruções que operam sobre dados de 512 bits. Também são apresentados o número total de laços avaliados e a percentagem desses padrões que não poderia ser convertida pela existência de instruções com acesso à memória, instruções lendo registradores escritos pelas convertidas e estruturas de controle de fluxo (chamadas a rotinas, laços internos ou condicionais) no laço. Apenas aplicações com ao menos um padrão são exibidas. Além disso, apenas são apresentados dados relacionados à cópia de memória, visto que não foram encontrados padrões de operações vetoriais, elemento a elemento. Fonte: Autor.

	bwaves	cam4	lbm	x264	xalancbmk
Padrão encontrado	CP	CP	CP	CP	CP
# Padrões encontrados	5	98	1	1	3
# Laços avaliados	76	2580	1	534	445
% Padrões com LOAD não permitido	100%	91%	100%	0%	66%
% Padrões com STORE não permitido	100%	100%	100%	0%	66%
% Padrões com salto não permitido	100%	90%	100%	0%	100%
% Padrões com leitura de registradores da conversão	100%	25%	0%	0%	66%

instruções com dados vetoriais de 128, 256 e 512 bits, respectivamente, que pôde ser encontrado nos traços do SPEC-CPU 2017 utilizados. Já os padrões encontrados com instruções escalares de cópia de memória são apresentados na Tabela 5.5, enquanto que os padrões escalares de operação sobre um e dois vetores são apresentados nas Tabelas 5.6 e 5.7, respectivamente.

Cada tabela também apresenta o número de laços avaliados em cada aplicação, e a porcentagem dos padrões encontrados que não poderia ser convertida por nosso mecanismo devido às características (ii), (iii) e (iv). A característica (ii) foi dividida em duas categorias, instruções realizando leituras (*LOAD*) e escritas (*STORE*) à memória.

Apenas aplicações com algum padrão válido detectado foram apresentadas. Visto que a abordagem de utilização de traços visa manter as porções mais significativas de cada aplicação, nossos resultados e as dificuldades encontradas são um vislumbre do comportamento encontrado na execução completa das aplicações. Por conta disso, é possível que os padrões buscados existam em outros pontos da execução das aplicações do SPEC, porém não foram encontrados por não representarem as porções mais significativas de seus códigos. Além disso, apenas as instruções executadas durante a primeira iteração de cada laço foram analisadas. Dessa forma, em laços que apresentam desvios de fluxo, os padrões e características encontrados podem ser alterados, durante a execução de diferentes fluxos. Por conta disso, nesse experimento, apresentamos a percentagem total dos laços com os padrões buscados, que possui alguma instrução de desvio de fluxo, independentemente de sua localização em relação às instruções de seu padrão.

Observamos que, nas aplicações avaliadas, apenas as instruções escalares formaram uma quantidade significativa dos padrões de operação em um ou entre dois vetores. Assim, mecanismos para a conversão desse padrão devem ser desenvolvidos com foco nessas instruções. Além disso, para os padrões vetoriais, mesmo a cópia de memória foi encontrada em apenas uma pequena parte dos laços avaliados. Isso, porque a maior parte dos laços contendo instruções vetoriais, presentes no SPEC, apresenta diversas instruções em série entre cada *load* e *store*, evitando a comunicação frequente com a memória. Como os laços são desenrolados, diversas instruções AVX realizam *load* e *store* de diferentes partes dos dados a cada iteração, reduzindo o número de padrões simples que procuramos e fazendo com que, individualmente, essas instruções deixem de acessar dados contíguos entre as iterações. Visto que o SAPIVe não é capaz de unir diversos *loads* e *stores* de uma mesma iteração, isso impede nosso mecanismo de encontrar um padrão de acessos contíguos à memória, inviabilizando suas conversões.

Assim, mecanismos de conversão para instruções vetoriais precisam ser capazes de lidar com dados temporários entre cada instrução de memória, correspondentes a cada uma das operações em série presentes entre *loads* e *stores*, algo inviável sem a adoção de mais alterações no funcionamento da VIMA. Além disso, para quase todos os padrões, existem outras instruções realizando acessos à memória e estruturas de controle de fluxo no mesmo laço. Isso pode ser consequência, do processo de vetorização realizado pelo compilador (desenrolando e unindo laços) ou da complexidade dos laços principais já existente nessas aplicações.

Apesar dos problemas com acessos à memória e desvios de fluxo, apenas os padrões de operação entre vetores com instruções escalares e a cópia de memória sobre dados de 512 bits, têm outras instruções lendo seus registradores na maior parte das ocorrências. Isso pode indicar que, em instruções vetoriais, um mesmo dado é raramente utilizado por mais de uma instrução, porém é algo que precisa ser mais profundamente estudado.

Podemos notar que existe um padrão de cópia de memória com instruções AVX-512, na aplicação x264, que não apresenta problemas para a conversão. Porém, o endereço de destino dessa cópia não pôde ser alinhado com a cache da VIMA, impossibilitando sua conversão por nosso mecanismo. Outra aplicação com um padrão interessante, porém que não poderia ser convertido pelo SAPIVe por suas instruções de 128 bits, é o bwaves. Apesar desse padrão não apresentar outros acessos à memória ou desvios em seu laço, ele é executado um número muito pequeno de vezes, o que também impede sua conversão.

Como apresentado em nossas análises, um mecanismo de conversão eficiente de instruções vetoriais da CPU, para instruções vetoriais IMA, deve ser capaz de converter laços pequenos, superar as barreiras de acessos à memória de instruções de fora da conversão, tratar dependências, entre instruções da conversão e externas, gerenciar acessos não-contíguos à memória e padrões complexos de instruções encadeadas.

Considerando o SAPIVe, a adaptação da VIMA para suportar operações sobre dados menores e o armazenamento apenas do PC da instrução inicial de cada conversão, ao invés de todas as instruções convertidas no ROB, poderiam permitir a conversão de instruções escalares, que representam a maior parte das oportunidades de conversão. Além disso, a adição de um registrador para o armazenamento de dados temporários na VIMA habilitaria nossas conversões para laços com instruções encadeadas e em algumas situações de dependência, onde instruções de fora do conjunto também poderiam ser levadas para a memória, formando padrões mais complexos.

Para lidar com laços desenrolados, uma alternativa seria identificar, durante a primeira iteração de cada laço, quais intruções realizam acessos à memória contíguos entre si, tratando-as como uma única instrução pelo SAPIVe. Já para as demais instruções acessando a memória, seria necessário identificar quando seus acessos não se sobrepõe a nenhuma conversão, permitindo conversões no mesmo laço que outros acessos à memória. Por fim, a conversão de padrões sem acessos a dados contíguos dependeria de suporte fornecido pela VIMA, o que poderia ser feito por meio de instruções especiais com máscaras, que seriam ajustadas de acordo com os acessos à memória das instruções convertidas.

6 CONCLUSÕES

Com o surgimento de memórias 3D, como o *Hybrid Memory Cube* (HMC) e o *High-Bandwidth Memory* (HBM), surgiram diversas técnicas IMA capazes de aumentar o desempenho de aplicações *memory-bound*. Porém, seu uso depende de bibliotecas adaptadas, anotações de código, rotinas *intrinsic* ou compiladores especializados, alternativas que podem não ser viáveis para códigos legados ou proprietários. Esta dissertação propôs o SAPIVe, um mecanismo em *hardware* para a conversão de operações AVX *memory-bound*, presentes em laços de aplicações, em instruções IMA da VIMA. Por meio da conversão de instruções durante sua execução, esse dispositivo é opaco aos desenvolvedores, podendo usufruir das vantagens oferecidas pelo processamento em memória sem qualquer alteração ou recompilação das aplicações.

O SAPIVe foi implementado sobre o simulador *Ordinary Computing Simulator* (OrCS), e avaliado em duas aplicações ideais *memory-bound*, uma cópia de dados entre vetores e uma soma vetorial. Nessas aplicações, fomos capazes de obter uma aceleração de até 5 vezes, sem qualquer sobrecusto para os desenvolvedores ou necessidade de alteração em *softwares* existentes. Além disso, mostramos que em sistemas futuros, com maiores *buffers* e estágios mais largos, o SAPIVe pode ser capaz de se equiparar ao uso de *intrinsics* em aplicações ideais. Com base em nossos resultados, conseguimos mostrar que, apesar de suas limitações e sobrecustos, a auto-vetorização em *hardware* para processamento em memória pode levar a ganhos de desempenho em casos ideais.

Também avaliamos nosso mecanismo sobre as aplicações do *benchmark* SPEC-CPU 2017, porém, devido aos laços mais complexos existentes nessas aplicações, não fomos capazes de realizar conversões. Então apresentamos um breve estudo das oportunidades de conversão para a VIMA existentes nessas aplicações, avaliando a utilização de instruções vetoriais e destacando dificuldades relevantes para a conversão dessas aplicações, que devem ser consideradas em futuras técnicas de conversão dinâmica.

Assim, trabalhos futuros podem adaptar o SAPIVe para a detecção de padrões mais complexos de instruções vetoriais encadeadas, além de aprimorar a detecção de conflitos no acesso à memória (desambiguadores de acesso à memória) e adotar preditores de iterações em laços. Com isso, permitindo a conversão de laços com poucas iterações e tornando as conversões desse mecanismo mais genéricas e efetivas para aplicações reais.

Devido às restrições de tempo, não realizamos uma avaliação dos gastos e ganhos energéticos do SAPIVe, nem consideramos o tempo necessário para as invalidações realizadas pelo protocolo de coerência a cada instrução VIMA. Tópicos que podem ser abordados em trabalhos futuros, junto a uma comparação de nosso mecanismo com um sistema de *prefetch* de dados nas *caches* da CPU. Além desses, algumas adaptações do SAPIVe também podem ser avaliadas futuramente, como a adição de uma cache de conversões, acelerando o início de conversões conhecidas e o armazenamento do PC da primeira instrução de cada conversão, evitando o acúmulo das instruções sendo convertidas no ROB e liberando espaço para mais conversões paralelas e conversões para instruções VIMA maiores.

REFERÊNCIAS

(2002). An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151.

Ahmed, H., Santos, P. C., Lima, J. P. C., Moura, R. F., Alves, M. A. Z., Beck, A. C. S. e Carro, L. (2019). A compiler for automatic selection of suitable processing-in-memory instructions. Em *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, páginas 564–569.

Ahn, J., Hong, S., Yoo, S., Mutlu, O. e Choi, K. (2015a). A scalable processing-in-memory accelerator for parallel graph processing. Em *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, páginas 105–117.

Ahn, J., Yoo, S., Mutlu, O. e Choi, K. (2015b). Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. Em *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, páginas 336–348.

Akin, B., Franchetti, F. e Hoe, J. C. (2015). Data reorganization in memory using 3d-stacked dram. *SIGARCH Comput. Archit. News*, 43(3S):131–143.

Altman, E., Kaeli, D. e Sheffer, Y. (2000). Welcome to the opportunities of binary translation. *Computer*, 33:40 – 45.

Alves, M. A. Z. (2014). *Increasing Energy Efficiency of Processor Caches via Line Usage Predictors*. Tese de doutorado, Universidade Federal do Rio Grande do Sul, Porto Alegre - Brasil.

Alves, M. A. Z., Diener, M., Santos, P. C. e Carro, L. (2016). Large vector extensions inside the hmc. Em *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, páginas 1249–1254.

Alves, M. A. Z., Santos, S., Cordeiro, A. S., Moreira, F. B., Santos, P. C. e Carro, L. (2022). Vector in memory architecture for simple and high efficiency computing.

Alves, M. A. Z., Villavieja, C., Diener, M., Moreira, F. B. e Navaux, P. O. A. (2015). Sinuca: A validated micro-architecture simulator. Em *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, páginas 605–610.

Arm (2016). Cortex-a73. `https://developer.arm.com/ip-products/processors/cortex-a/cortex-a73`. Acessado em 16/02/2021.

Barbalace, A., Iliopoulos, A., Rauchfuss, H. e Brasche, G. (2017). It's time to think about an operating system for near data processing architectures. Em *HotOS '17: Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, páginas 56–61, Whistler BC - Canada.

Barrière, A., Blazy, S., Flückiger, O., Pichardie, D. e Vitek, J. (2021). Formally verified speculation and deoptimization in a jit compiler. *Proc. ACM Program. Lang.*, 5(POPL).

Boroumand, A., Ghose, S., Kim, Y., Ausavarungnirun, R., Shiu, E., Thakur, R., Kim, D., Kuusela, A., Knies, A., Ranganathan, P. e Mutlu, O. (2018). Google workloads for consumer devices: Mitigating data movement bottlenecks. Em *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, página 316–331, New York, NY, USA. Association for Computing Machinery.

Bucek, J., Lange, K.-D. e v. Kistowski, J. (2018). Spec cpu2017: Next-generation compute benchmark. Em *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, página 41–42, New York, NY, USA. Association for Computing Machinery.

Calder, B. et al. (2005). Simpoint 3.0: Faster and more flexible program analysis.

Call Barreiro, A. (2014). Dynamic vectorization of instructions. Dissertação de Mestrado, Universitat Politècnica de Catalunya, Barcelona, Spain.

Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S. e Chatterjee, S. (2008). Software transactional memory. *Communications of the ACM*, 51(11):40–46.

Castro, D., Romano, P. e Barreto, J. (2019). Hardware transactional memory meets memory persistency. *Journal of Parallel and Distributed Computing*, 130:63–79.

Clark, N., Hormati, A., Yehia, S., Mahlke, S. e Flautner, K. (2007). Liquid simd: Abstracting simd hardware using lightweight dynamic mapping. Em *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, páginas 216–227.

Cordeiro, A. S. (2020). Porting machine learning algorithms to vector-in-memory architecture. Dissertação de Mestrado, Pós-Graduação em Informática - Universidade Federal do Paraná, Curitiba - PR.

Cordeiro, A. S., Kepe, T. R., Tomé, D. G., de Almeida, E. C. e Alves, M. A. Z. (2017). Intrinsics-hmc: An automatic trace generator for simulations of processing-in-memory instructions. *XVIII Simpósio em Sistemas Computacionais de Alto Desempenho - WSCAD*.

David Kanter (2010). Intel's sandy bridge microarchitecture. `https://www.realworldtech.com/sandy-bridge/5/`. Acessado em 12/06/2022.

David Kanter (2012). Intel's haswell cpu microarchitecture. `https://www.realworldtech.com/haswell-cpu/3/`. Acessado em 12/06/2022.

Diegues, N., Romano, P. e Rodrigues, L. (2014). Virtues and limitations of commodity hardware transactional memory. Em *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, página 3–14, New York, NY, USA. Association for Computing Machinery.

dos Santos, S. R. (2022). *Enabling multi-threaded execution and improved memory access in fine-grain near-data processing systems*. Tese de doutorado, Universidade Federal do Paraná, Curitiba - Brasil.

Doweck, J., Kao, W.-F., Lu, A. K.-y., Mandelblat, J., Rahatekar, A., Rappoport, L., Rotem, E., Yasin, A. e Yoaz, A. (2017). Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2):52–62.

Fangueiro, N. H. N. R. E. (2017). *Exploiting Hardware Transactional Memory to Accelerate Concurrent Spatio-Temporal Indexes*. Tese de doutorado, ULisboa - Universidade de Lisboa, Lisboa - Portugal.

Flynn, M. J. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909.

Flynn, M. J. e Rudd, K. W. (1996). Parallel architectures. *ACM Comput. Surv.*, 28(1):67–70.

FUJITSU (2020). Fujitsu processor a64fx. `https://www.fujitsu.com/global/products/computing/servers/supercomputer/a64fx/`. Acessado em 16/02/2021.

Godfrey, M. D. e Hendry, D. F. (1993). The computer as von neumann planned it. *IEEE Annals of the History of Computing*, 15(1):11–21.

Grosser, T., Größlinger, A. e Lengauer, C. (2012). Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.*, 22.

Guo, Q., Alachiotis, N., Akin, B., Sadi, F., Xu, G., Low, T., Pileggi, L., Hoe, J. e Franchetti, F. (2014). 3d-stacked memory-side acceleration: Accelerator and system design. Em *2nd Workshop on Near-Data Processing (WoNDP)*.

Guo, Q., Low, T., Alachiotis, N., Akin, B., Pileggi, L., Hoe, J. C. e Franchetti, F. (2015). Enabling portable energy efficiency with memory accelerated library. Em *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, páginas 750–761.

Hadidi, R., Nai, L., Kim, H. e Kim, H. (2017). Cairo: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory. Em *ACM Transactions on Architecture and Code Optimization*.

Hallou, N., Rohou, E. e Clauss, P. (2016). Runtime vectorization transformations of binary code. *International Journal of Parallel Programming*, 45(6):1536–1565.

Hallou, N., Rohou, E., Clauss, P. e Ketterlin, A. (2015). Dynamic re-vectorization of binary code. Em *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, páginas 228–237.

Han, L., Shen, Z., Shao, Z., Huang, H. H. e Li, T. (2017). A novel reram-based processing-in-memory architecture for graph computing. Em *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, páginas 1–6.

Hashemi, M., Khubaib, Ebrahimi, E., Mutlu, O. e Patt, Y. N. (2016). Accelerating dependent cache misses with an enhanced memory controller. Em *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, página 444–455. IEEE Press.

Hennessy, J. L. e Patterson, D. A. (sixth edition, 2017). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

Herlihy, M. e Moss, J. E. B. (1993). Transactional memory: Architectural support for lock-free data structures. Em *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, página 289–300, New York, NY, USA. Association for Computing Machinery.

Holewinski, J., Ramamurthi, R., Ravishankar, M., Fauzia, N., Pouchet, L.-N., Rountev, A. e Sadayappan, P. (2012). Dynamic trace-based analysis of vectorization potential of applications. *SIGPLAN Not.*, 47(6):371–382.

Ian Cutress (2019). Sunny cove microarchitecture: Going deeper and wider. https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3. Acessado em 12/06/2022.

Ian Cutress and Andrei Frumusanu (2021). Golden cove microarchitecture (p-core) examined. https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/3. Acessado em 12/06/2022.

Intel (2011). Introduction to intel® advanced vector extensions. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html.

Intel (2018). Pin - a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

Intel (2020). Intel® core™ i7-10700 processor. https://ark.intel.com/content/www/us/en/ark/products/199316/intel-core-i7-10700-processor-16m-cache-up-to-4-80-ghz.html. Acessado em 16/02/2021.

Intel (2021). Intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons.

Jain, S., Ranjan, A., Roy, K. e Raghunathan, A. (2018). Computing in memory with spin-transfer torque magnetic ram. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):470–483.

Jeddeloh, J. e Keeth, B. (2012). Hybrid memory cube new dram architecture increases density and performance. Em *2012 Symposium on VLSI Technology (VLSIT)*, páginas 87–88.

Ji, C., Li, Y., Qiu, W., Awada, U. e Li, K. (2012). Big data processing in cloud computing environments. Em *2012 12th International Symposium on Pervasive Systems, Algorithms and Networks*, páginas 17–23.

Jimenez, D. A. (2005). Piecewise linear branch prediction. *SIGARCH Comput. Archit. News*, 33(2):382–393.

Jon Stokes (2008). What you need to know about intel's nehalem cpu. https://arstechnica.com/gadgets/2008/04/what-you-need-to-know-about-nehalem/. Acessado em 12/06/2022.

K. Chang, K. (2017). *Understanding and Improving the Latency of DRAM-Based Memory Systems*. Tese de doutorado, Carnegie Mellon University, Pittsburgh - USA.

Kalathingal, S., Collange, S., Swamy, B. N. e Seznec, A. (2016). Dynamic inter-thread vectorization architecture: Extracting dlp from tlp. Em *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, páginas 18–25.

Knuth, D. E. (1970). Von neumann's first computer program. *ACM Comput. Surv.*, 2(4):247–260.

Kumar, R., Martínez, A. e González, A. (2016). Assisting static compiler vectorization with a speculative dynamic vectorizer in an hw/sw codesigned environment. *ACM Trans. Comput. Syst.*, 33(4).

Kumar, R., Martínez, A. e González, A. (2013). Speculative dynamic vectorization to assist static vectorization in a hw/sw co-designed environment. Em *20th Annual International Conference on High Performance Computing*, páginas 79–88.

Köhler, R. (2019). Aceleração de cache misses prováveis através de requisições paralelas. Dissertação de Mestrado, Pós-Graduação em Informática - Universidade Federal do Paraná, Curitiba - PR.

Larus, J. e Kozyrakis, C. (2008). Transactional memory. *Communications of the ACM*, 51(7):80–88.

Lattner, C. e Adve, V. (2004). Llvm: a compilation framework for lifelong program analysis & transformation. Em *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, páginas 75–86.

Lechner, M. (2020). Evolution of single-threaded x86 cpu performance. `https://mlech26l.github.io/pages/2020/12/17/cpus.html`.

Lee, V. T., Mazumdar, A., del Mundo, C. C., Alaghi, A., Ceze, L. e Oskin, M. (2018). Application codesign of near-data processing for similarity search. Em *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, páginas 896–907.

Lee, W. J., Kim, C. H., Paik, Y., Park, J., Park, I. e Kim, S. W. (2019). Design of processing-"inside"-memory optimized for dram behaviors. *IEEE Access*, 7:82633–82648.

Li, C., Wang, Z., Rao, M., Belkin, D., Song, W., Jiang, H., Yan, P., Li, Y., Lin, P., Hu, M., Ge, N., Strachan, J. P., Barnell, M., Wu, Q., Williams, R. S., Yang, J. J. e Xia, Q. (2019). Long short-term memory networks in memristor crossbar arrays. *Nature Machine Intelligence*, 1(1):49–57.

Liu, J., Zhao, H., Ogleari, M. A., Li, D. e Zhao, J. (2018). Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. Em *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, páginas 655–668.

Lomet, D. B. (1977). Process structuring, synchronization, and recovery using atomic actions. *ACM SIGSOFT Software Engineering Notes*, 2(2):128–137.

Long, Y., Kim, D., Lee, E., Saha, P., Mudassar, B. A., She, X., Khan, A. I. e Mukhopadhyay, S. (2019). A ferroelectric fet-based processing-in-memory architecture for dnn acceleration. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, 5(2):113–122.

Maleki, S., Gao, Y., Garzar´n, M. J., Wong, T. e Padua, D. A. (2011). An evaluation of vectorizing compilers. Em *2011 International Conference on Parallel Architectures and Compilation Techniques*, páginas 372–382.

Mikolov, T., Deoras, A., Povey, D., Burget, L. e Černocký, J. (2011). Strategies for training large scale neural network language models. Em *2011 IEEE Workshop on Automatic Speech Recognition Understanding*, páginas 196–201.

Milan, R. (2019). *Memory bandwidth and latency in HPC: system requirements and performance impact*. Tese de doutorado, UPC - Universitat Politècnica de Catalunya, Barcelona - Espanha.

Nai, L., Hadidi, R., Sim, J., Kim, H., Kumar, P. e Kim, H. (2017). Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. Em *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, páginas 457–468.

Nai, L. e Kim, H. (2015). Instruction offloading with hmc 2.0 standard: A case study for graph traversals. Em *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, página 258–261, New York, NY, USA. Association for Computing Machinery.

Nakamura, T., Miki, S. e Oikawa, S. (2011). Automatic vectorization by runtime binary translation. Em *2011 Second International Conference on Networking and Computing*, páginas 87–94.

Nuzman, D., Dyshel, S., Rohou, E., Rosen, I., Williams, K., Yuste, D., Cohen, A. e Zaks, A. (2011). Vapor simd: Auto-vectorize once, run everywhere. Em *International Symposium on Code Generation and Optimization (CGO 2011)*, páginas 151–160.

Oberman, S., Favor, G. e Weber, F. (1999). Amd 3dnow! technology: architecture and implementations. *IEEE Micro*, 19(2):37–48.

Pajuelo, A., González, A. e Valero, M. (2002). Speculative dynamic vectorization. *SIGARCH Comput. Archit. News*, 30(2):271–280.

Patil, H., Cohn, R. et al. (2004). Pinpointing representative portions of large intel ® itanium ® programs with dynamic instrumentation. Em *Int. Symp. on Microarchitecture*.

Plangger, R. e Krall, A. (2016). Vectorization in pypy's tracing just-in-time compiler. Em *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '16, página 67–76, New York, NY, USA. Association for Computing Machinery.

Pohl, A. (2020). *Efficient Utilization of Vector Extensions in Microprocessors*. Tese de doutorado, Technische Universität Berlin, Berlin - Germany.

Pohl, A., Cosenza, B. e Juurlink, B. (2020). Vectorization cost modeling for neon, avx and sve. *Performance Evaluation*, 140-141:102106.

Porpodas, V. e Jones, T. M. (2015). Throttling automatic vectorization: When less is more. Em *2015 International Conference on Parallel Architecture and Compilation (PACT)*, páginas 432–444.

Preshing, J. e Poley, H. (2012). A look back at single-threaded cpu performance. `http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/`.

Rane, A., Krishnaiyer, R., Newburn, C. J., Browne, J., Fialho, L. e Matveev, Z. (2015). Unification of static and dynamic analyses to enable vectorization. Em Brodman, J. e Tu, P., editores, *Languages and Compilers for Parallel Computing*, páginas 367–381, Cham. Springer International Publishing.

Rickards, I. (2010). Arm architecture & neon. `http://mercury.pr.erau.edu/~siewerts/cs332/documents/ARM-ASM/ARM-Tutorials/lect.10.arm_soc.pdf`. Acessado em 16/02/2021.

Santos, P. C., Alves, M. A. Z. e Carro, L. (2017). Hmc and ddr performance trade-offs. Em Götz, M., Schirner, G., Wehrmeister, M. A., Al Faruque, M. A. e Rettberg, A., editores, *System Level Design from HW/SW to Memory for Embedded Systems*, páginas 159–171, Cham. Springer International Publishing.

Santos, P. C., Moreira, F. B., Cordeiro, A. S., Santos, S. R., Kepe, T. R., Carro, L. e Alves, M. A. Z. (2021). Survey on near-data processing: Applications and architectures. *Journal of Integrated Circuits and Systems*, 16(2):1–17.

Santos, P. C., Oliveira, G. F., Tomé, D. G., Alves, M. A. Z., Almeida, E. C. e Carro, L. (2017). Operand size reconfiguration for big data processing in memory. Em *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, páginas 710–715.

Seshadri, V., Lee, D., Mullins, T., Hassan, H., Boroumand, A., Kim, J., Kozuch, M. A., Mutlu, O., Gibbons, P. B. e Mowry, T. C. (2017). Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. Em *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, páginas 273–287.

Silva Junior, P. (2020). *Improving efficiency of general Purpose computer systems by adopting processing-in-memory architecture*. Tese de doutorado, UFRGS - Universidade Federal do Rio Grande do Sul, Porto Alegre - Brazil.

Singh, G., Chelini, L., Corda, S., Awan, A. J., Stuijk, S., Jordans, R., Corporaal, H. e Boonstra, A. (2019). Near-memory computing: Past, present, and future. *CoRR*, abs/1908.02640.

Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., Reid, A., Rico, A. e Walker, P. (2017). The arm scalable vector extension. *IEEE Micro*, 37(2):26–39.

Talati, N., Ben-Hur, R., Wald, N., Haj-Ali, A., Reuben, J. e Kvatinsky, S. (2019). mmpu—a real processing-in-memory architecture to combat the von neumann bottleneck. *Applications of Emerging Memory Technology. Springer Series in Advanced Microelectronics*, 63:191–213.

Tomé, D. G., Santos, P. C., Carro, L., Almeida, E. C. e Alves, M. A. Z. (2018). Hipe: Hmc instruction predication extension applied on database processing. Em *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, páginas 261–264.

TOP 500 (2020). Top500 list - november 2020. `https://www.top500.org/lists/top500/list/2020/11/`. Acessado em 16/02/2021.

von Neumann, J. (1993). First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, 15(4):27–75.

Wang, Y., Chen, W., Yang, J. e Li, T. (2019). Exploiting parallelism for cnn applications on 3d stacked processing-in-memory architecture. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):589–600.

Wang, Y. E., Wu, C., Wang, X., Hazelwood, K. M. e Brooks, D. (2019). Exploiting parallelism opportunities with deep learning frameworks. *CoRR*, abs/1908.04705.

Watkinson, N., Shivam, A., Chen, Z., Veidenbaum, A., Nicolau, A. e Gong, Z. (2019). Using hardware counters to predict vectorization. Em Rauchwerger, L., editor, *Languages and Compilers for Parallel Computing*, páginas 3–16, Cham. Springer International Publishing.

Yardimci, E. e Franz, M. (2008). Dynamic parallelization and vectorization of binary executables on hierarchical platforms. *J. Instr. Level Parallelism*, 10.

Yoo, R. M., Hughes, C. J., Lai, K. e Rajwar, R. (2013). Performance evaluation of intel® transactional synchronization extensions for high-performance computing. Em *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA. Association for Computing Machinery.

Zhang, Y., Xu, L., Dong, Q., Wang, J., Blaauw, D. e Sylvester, D. (2018). Recryptor: A reconfigurable cryptographic cortex-m0 processor with in-memory and near-memory computing for iot security. *IEEE Journal of Solid-State Circuits*, 53(4):995–1005.

APÊNDICE A – RESULTADOS ADICIONAIS DAS APLICAÇÕES IDEAIS

O SAPIVe tem como foco a conversão de instruções presentes em laços de cópia e operações aritméticas/lógicas, elemento a elemento, sobre um vetor ou entre dois vetores. Este apêndice apresenta uma análise mais aprofundada dos resultados obtidos pelo SAPIVe sobre os dois casos ideais avaliados na Seção 5.1, os Códigos 5.1 e 5.2.

O primeiro código, denominado **memcpy**, consiste em um laço realizando cópia de memória entre dois vetores. Já o segundo código, denominado **vecsum**, realiza a soma entre dois vetores, elemento a elemento.

Para os experimentos deste apêndice, todos os algoritmos foram compilados em duas versões, uma utilizando AVX-256 e outra AVX-512, mantendo as configurações de compilação apresentadas na Tabela 5.1.

Assim como na Seção 5.1, para garantir resultados consistentes, em todos os experimentos, sobre cada combinação de algoritmo e versão do AVX, os mesmo endereços de acesso à memória foram utilizados.

As avaliações deste apêndice são divididas em quatro etapas. Primeiro, realizamos uma análise dos efeitos da conversão realizada pelo SAPIVe, de acordo com o número de iterações de um laço. Em seguida, avaliamos o impacto da agressividade sobre as conversões. Então, a aceleração obtida é comparada com o uso de instruções VIMA de forma manual, por meio de rotinas *intrinsic*. Por fim, extrapolamos algumas configurações do sistema avaliado para entender como o aumento no tamanho dos *buffers* e da largura dos estágios do *pipeline* afetariam nossos resultados. Essas avaliações são apresentadas nas próximas seções.

A.0.1 Comportamento do mecanismo

Quanto maior o número de iterações presentes em um laço, maior o número de conversões possíveis para o SAPIVe. Laços pequenos, com iterações insuficientes para chegar ao fim da etapa de análise (descrita na Seção 4.2.3), não apresentam ganhos ou perdas significativas de desempenho ao executar com nossa extensão. Laços com tamanho suficiente para chegar ao fim da etapa de análise, que insere uma instrução VIMA correspondente, estão sujeitos ao sobrecusto dessa instrução. Já laços maiores, com um número suficiente de iterações para iniciar as conversões da Validação (descrita na Seção 4.2.4), podem resultar em conversões bem sucedidas, levando a possíveis melhorias no tempo de execução. Como ilustrado na Figura A.1, após a última conversão bem sucedida, não existem iterações restantes suficientes para compor uma nova instrução VIMA. Assim, a próxima conversão falha, devido à falta de iterações. Durante esse processo, as iterações finais do laço são mantidas travadas no ROB, até a invalidação se confirmar. No momento em que a invalidação ocorre, as instruções dessas iterações são liberadas para execução. Porém, essa espera até executar pode degradar o tempo de execução. Apesar disso, conforme o número de iterações de um laço aumenta, mais conversões podem ser realizadas, mitigando os efeitos da invalidação final.

Memcpy - AVX-256

A Figura A.2 apresenta o comportamento de nosso mecanismo ao realizar conversões para VIMA-256 B, sobre o algoritmo **memcpy** (Código 5.1), compilado para AVX-256. Cada ponto representa a aceleração obtida para um diferente número de iterações AVX-256, em relação ao nosso *baseline*, o código executado de maneira convencional pela CPU. O número máximo

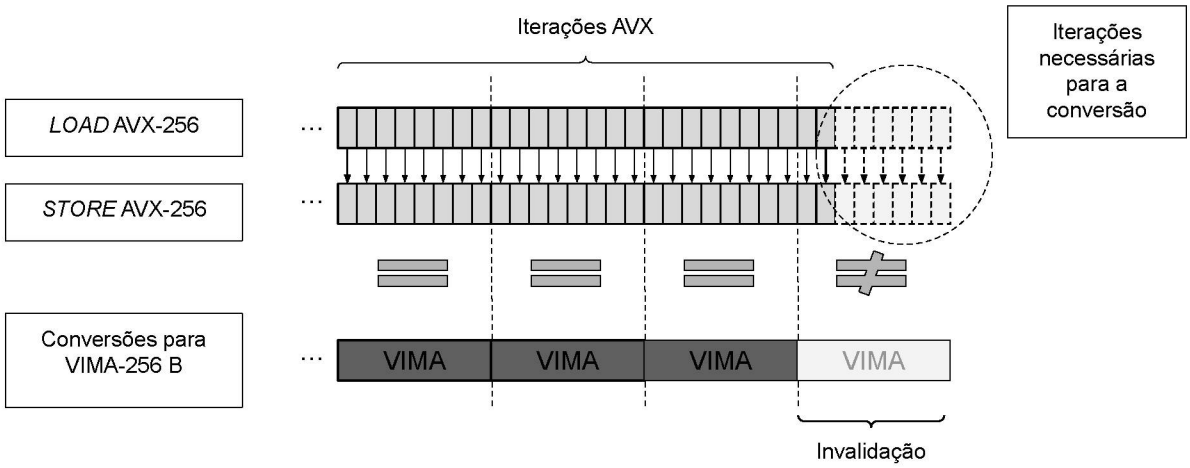


Figura A.1: Falha de conversão durante a última iteração de um laço contendo instruções AVX que realizam uma cópia de memória. Quando a última iteração de uma conversão é aquirida pelo *rename*, o mecanismo inicia a próxima conversão. Caso não existam iterações suficientes para completar essa nova conversão, ela segue para a etapa de invalidação. Fonte: Autor.

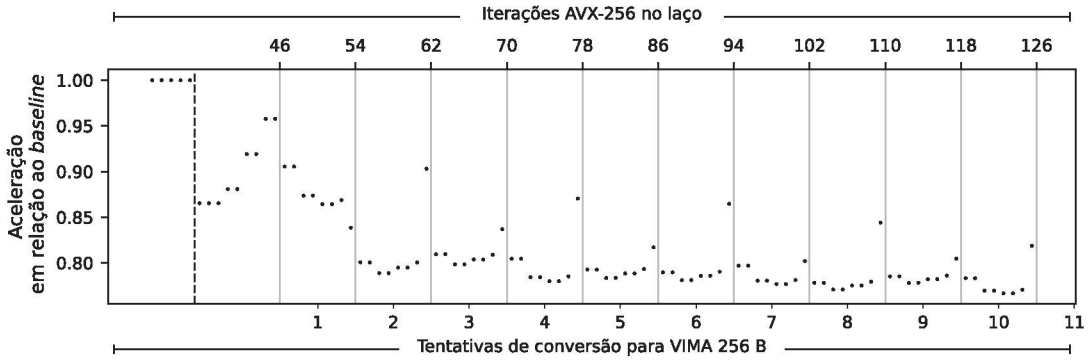


Figura A.2: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-256 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-256 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-256 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

de iterações AVX foi selecionado de forma que nosso mecanismo pudesse gerar ao menos 10 conversões VIMA bem sucedidas. Pontos acima de 1.00 indicam ganhos de desempenho em relação ao *baseline*, enquanto aqueles abaixo de 1.00 indicam perdas de desempenho. Por exemplo, um ponto em 2.00 indicaria que nosso mecanismo reduziu pela metade o tempo de execução da aplicação.

Inicialmente, nosso mecanismo não gera alterações no desempenho, por conta disso, os primeiros pontos se mantém em 1. A primeira divisão vertical tracejada indica a primeira inserção de uma instrução VIMA, gerada por nosso mecanismo no *pipeline*. Essa instrução é inserida ao final da etapa de análise (descrita na Seção 4.2.3) e inicia sua execução antes do mecanismo obter a próxima iteração AVX com *store* alinhado à cache da VIMA. Notamos que,

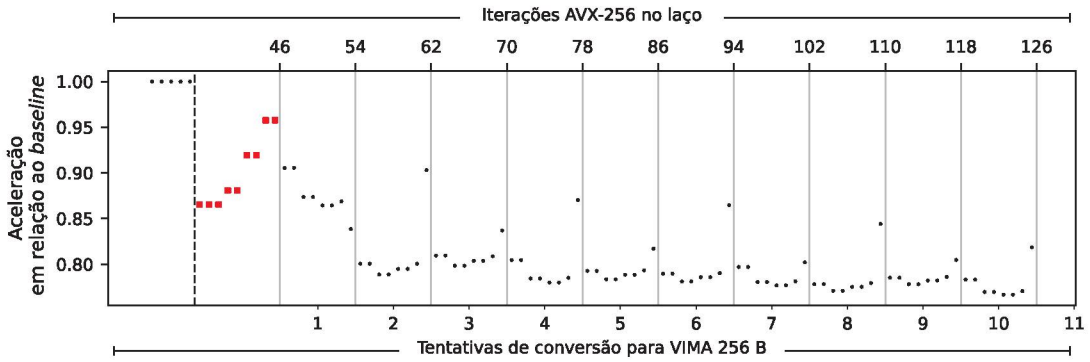


Figura A.3: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-256 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. Os pontos destacados correspondem a possíveis degradações no desempenho causadas pela primeira instrução VIMA, inserida na iteração indicada pela linha vertical pontilhada. Fonte: Autor.

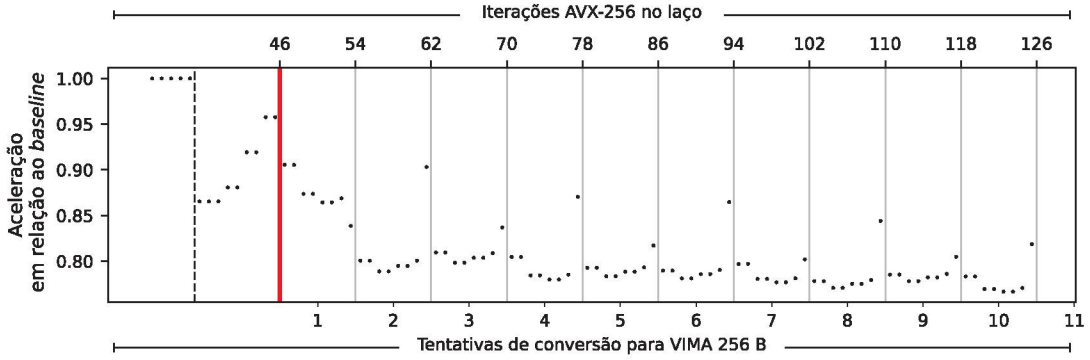


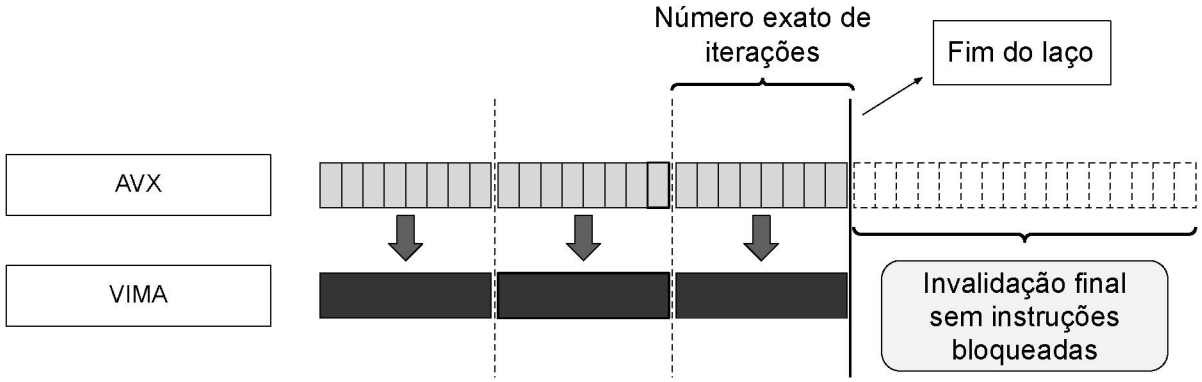
Figura A.4: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-256 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha destacada aponta o início da primeira conversão, ou seja, o momento no qual o mecanismo começa a coletar iterações para formar sua instrução VIMA. Fonte: Autor.

quando isso acontece, há uma perda de desempenho por conta do sobrecusto causado por essa instrução adicional, o que é destacado na Figura A.3.

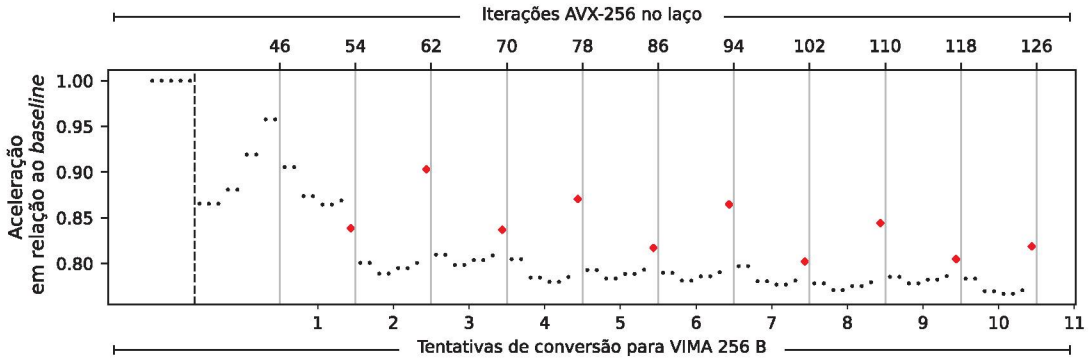
A próxima iteração a passar pelo *rename*, com seu *store* alinhado à cache da VIMA, é indicada pela divisão vertical destacada na Figura A.4. Cada próxima divisão vertical sólida representa o início de uma nova tentativa de conversão do mecanismo.

Após o início de uma dessas tentativas, as próximas iterações são acumuladas no ROB, sem executar, apenas calculando seus endereços de acesso à memória. Se houverem iterações suficientes para completar a conversão, as instruções bloqueadas são substituídas pelo resultado da VIMA, o que acontece nos casos evidenciados na Figura A.5, que também ilustra conversões com o número exato de iterações. Porém, se não houverem iterações suficientes para completar a conversão, ela é invalidada e as instruções que ficaram retidas no ROB são liberadas para execução. Isso é destacado na Figura A.6, que ilustra a falta de iterações para uma conversão e evidencia os casos onde instruções ficaram travadas por conta de uma invalidação.

Quanto mais instruções são acumuladas em uma tentativa de conversão, maior o número de instruções que tem sua execução atrasada caso uma invalidação ocorra. Isso pode ser observado

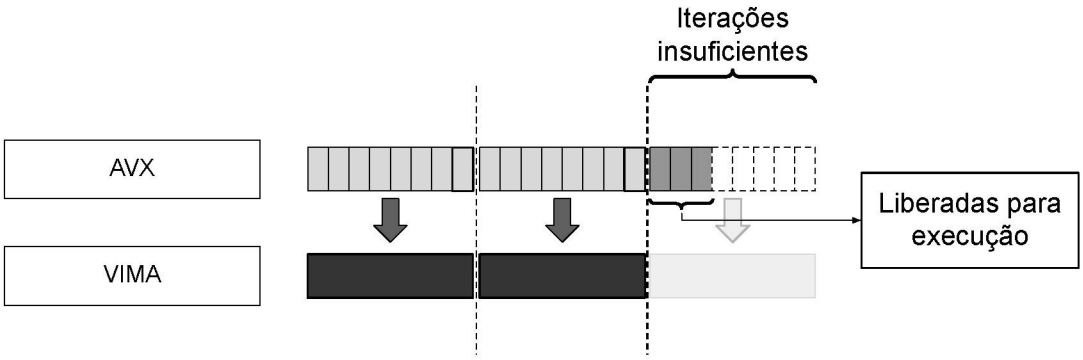


(a)

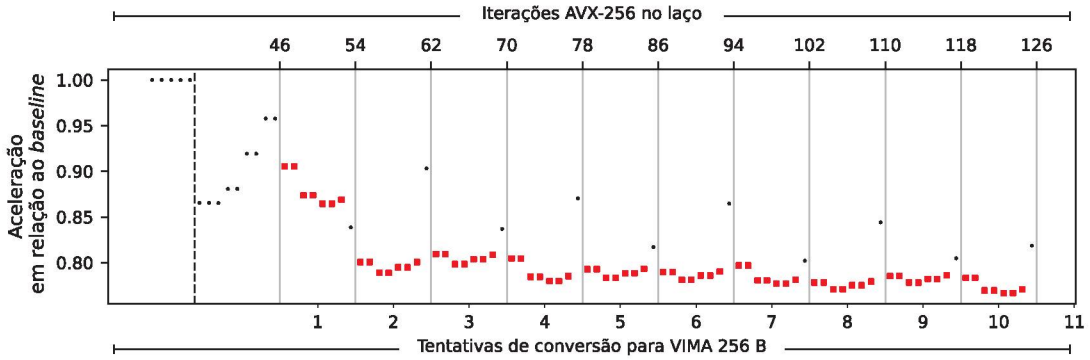


(b)

Figura A.5: Em (a) temos uma ilustração de conversão com número ideal de iterações, nenhuma iteração adicional fica travada por conta de uma conversão incorreta. Em (b) é exibida a aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-256 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. Os pontos em destaque indicam o número ideal de iterações representado por (a), ou seja, momento em que o número de iterações acumuladas é o exato para formar a instrução VIMA da conversão. Fonte: Autor.



(a)



(b)

Figura A.6: Em (a) temos uma ilustração de conversão com número insuficiente de iterações. Em (b) é exibida a aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-256 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. Os pontos em destaque indicam a ocorrência de iterações insuficientes para a conversão, o que é representado por (a). Fonte: Autor.

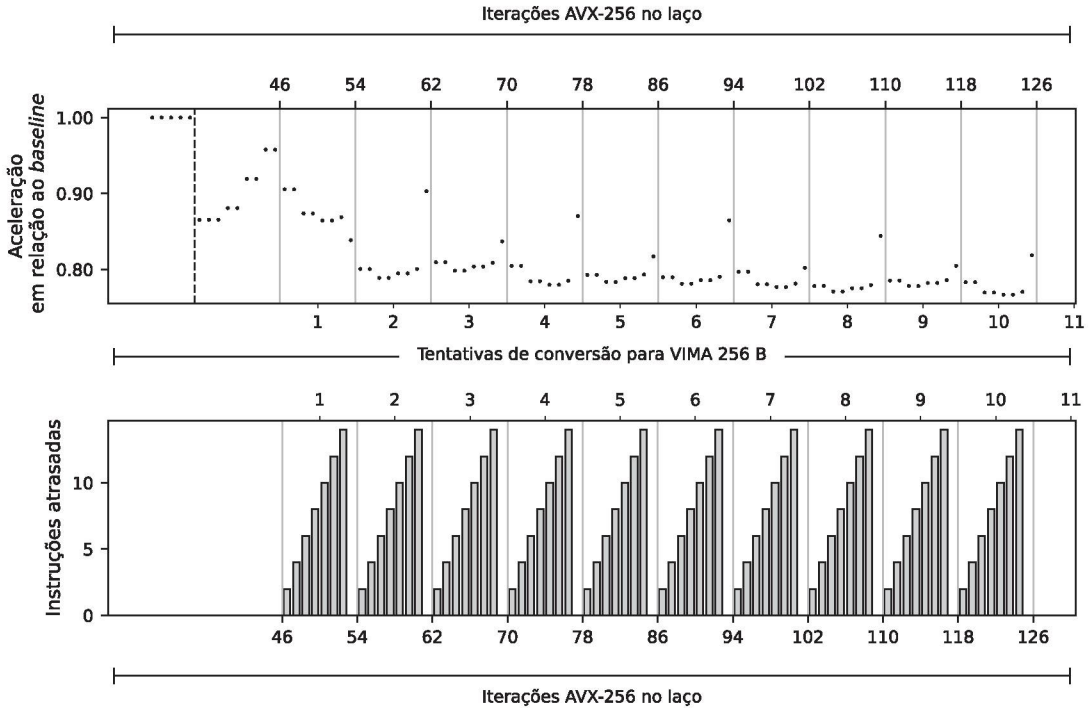


Figura A.7: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo, e o número de instruções com sua execução bloqueada no ROB devido a tentativas falhas de conversão para VIMA. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-256 B, por nosso mecanismo, durante a execução. A aceleração é apresentada no gráfico superior, onde cada ponto a representa para um número diferente de iterações AVX-256 no programa convertido. O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. O número de instruções com sua execução atrasada em uma tentativa falha de conversão é apresentado no gráfico inferior. Foram avaliadas iterações AVX-256 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-256 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

pela Figura A.7, que compara o gráfico de aceleração, com o número de instruções atrasadas caso a execução fosse encerrada em cada ponto. Para isso, o segundo gráfico dessa figura apresenta, na horizontal, o número de iterações contido no experimento. Já na vertical, temos o número de instruções atrasadas caso o laço fosse encerrado, ocasionando uma falha de conversão. Podemos notar que, conforme cada conversão é realizada, o número de instruções acumuladas aumenta, sendo reduzido sempre que uma conversão é bem sucedida.

A conversão de instruções AVX-256 para VIMA-256 B, nessa aplicação, não leva a ganhos de desempenho, porém o entendimento desse fenômeno requer estudos adicionais.

A Figura A.8 apresenta a aceleração obtida por nosso mecanismo durante a conversão do Código 5.1, compilado com instruções AVX-256, para VIMA-512 B. Podemos observar um padrão parecido com o apresentado nas conversões para VIMA-256 B. Nesse caso, as conversões também apresentam um desempenho menor que o da execução usual.

O problema do tempo de execução só é resolvido com a conversão para instruções VIMA-1024 B, apresentada na Figura A.9, que fornecem ganhos em relação às requisições AVX individuais. Com isso, começamos a gerar ganhos de desempenho a partir da segunda conversão, melhorando a cada iteração adicional, o que mostra a grande vantagem de instruções VIMA de

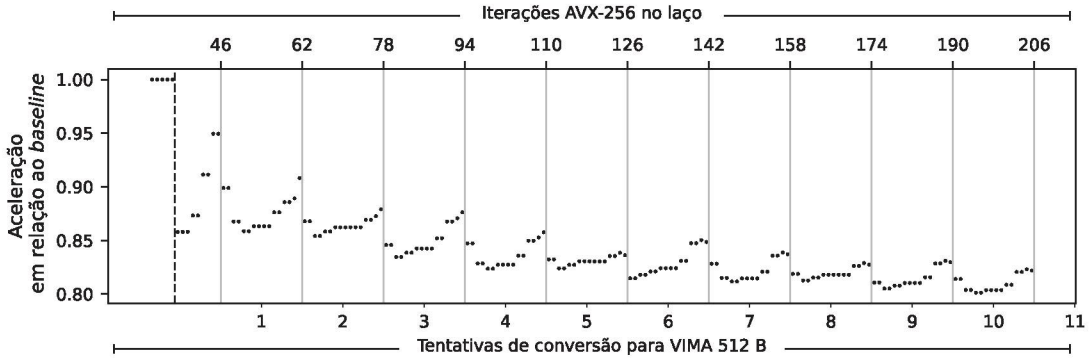


Figura A.8: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-512 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-256 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-512 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

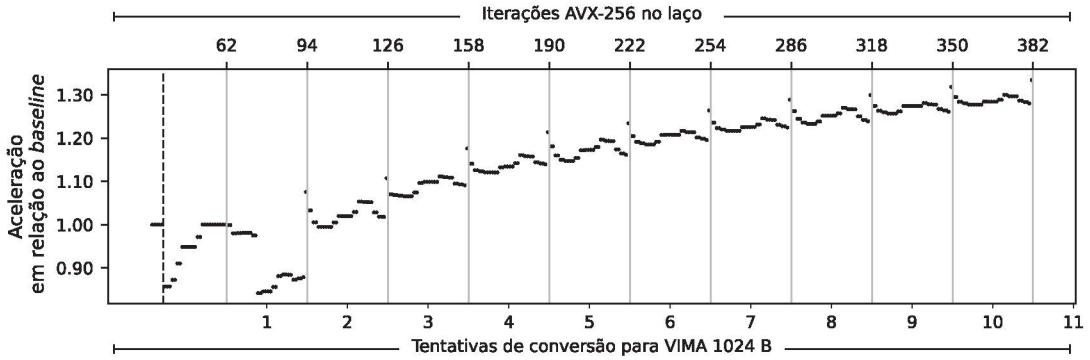


Figura A.9: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-1024 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-256 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-1024 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

maior tamanho.

Memcpy - AVX-512

A Figura A.10 apresenta o comportamento de nosso mecanismo ao realizar conversões para VIMA-256 B, sobre o algoritmo **memcpy** (Código 5.1), compilado para AVX-512. Cada ponto representa a aceleração obtida pelo SAPIVe para um número de iterações AVX-512, em relação ao nosso *baseline*, o código executado de maneira convencional pela CPU. O número máximo de

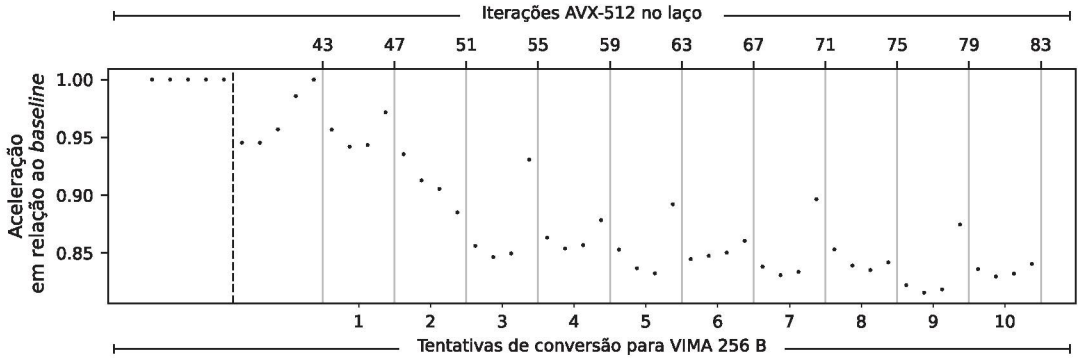


Figura A.10: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-512, sendo convertido para VIMA-256 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-512 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-512 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-256 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

iterações AVX apresentado foi selecionado de forma que nosso mecanismo pudesse gerar ao menos 10 conversões VIMA bem sucedidas.

A primeira divisão vertical tracejada indica a primeira inserção de uma instrução VIMA, gerada por nosso mecanismo no *pipeline*. Essa instrução é inserida ao final da etapa de análise (descrita na Seção 4.2.3) e inicia sua execução antes do mecanismo obter a próxima iteração AVX com *store* alinhado à cache da VIMA. Notamos que, quando isso acontece, há uma perda de desempenho por conta do sobrecusto causado por essa instrução adicional.

Cada divisão vertical sólida representa o início de uma tentativa de conversão do mecanismo.

Da mesma forma que com instruções AVX-256, a conversão de instruções AVX-512 para VIMA-256 B, nessa aplicação, não leva a ganhos de desempenho.

A Figura A.11 apresenta a aceleração obtida por nosso mecanismo durante a conversão do Código 5.1, compilado com instruções AVX-512, para VIMA-512 B. Podemos observar um padrão parecido com o apresentado nas conversões para VIMA-256 B. Nesse caso, as conversões também apresentam um desempenho menor que o da execução usual.

O problema do tempo de execução só é resolvido com a conversão para instruções VIMA-1024 B, apresentado na Figura A.12, que fornecem ganhos em relação às requisições AVX individuais. Com isso, começamos a gerar ganhos de desempenho a partir da segunda conversão, melhorando a cada iteração adicional, o que mostra a grande vantagem de instruções VIMA de maior tamanho, visto que o mesmo ocorre para a conversão de instruções AVX-256, nessa aplicação.

Vecsum - AVX-256

A Figura A.13 apresenta o comportamento de nosso mecanismo ao realizar conversões para VIMA-256 B, sobre o algoritmo **vecsum** (Código 5.2), compilado para AVX-256. Cada ponto representa a aceleração obtida pelo SAPIVe para um número de iterações AVX-256, em relação ao nosso *baseline*, o código executado de maneira convencional pela CPU. O número máximo de

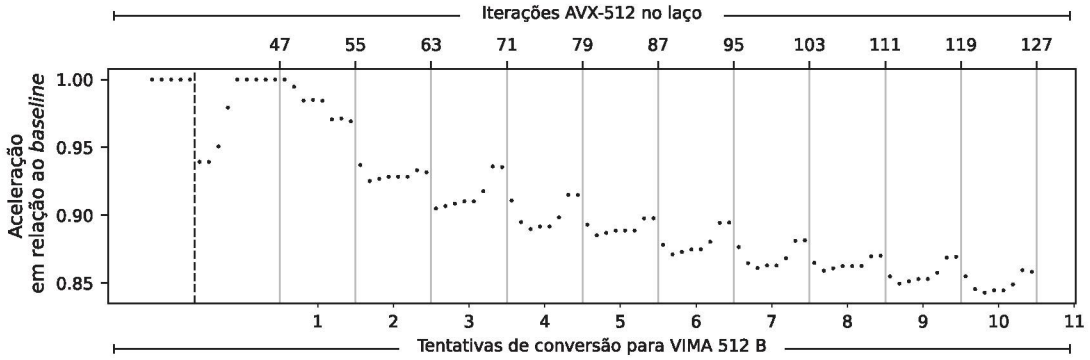


Figura A.11: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-512, sendo convertido para VIMA-512 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-512 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-512 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-512 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

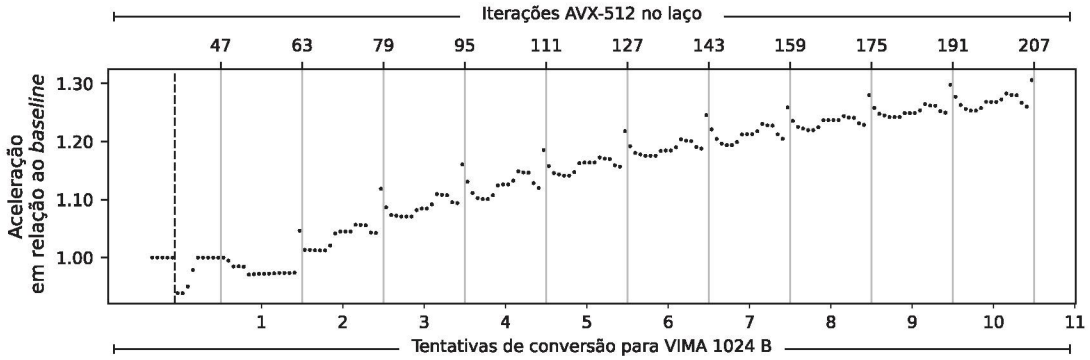


Figura A.12: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-512, sendo convertido para VIMA-1024 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-512 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-512 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-1024 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

iterações AVX apresentado foi selecionado de forma que nosso mecanismo pudesse gerar ao menos 10 conversões VIMA bem sucedidas.

A primeira divisão vertical tracejada indica a primeira inserção de uma instrução VIMA, gerada por nosso mecanismo no *pipeline*. Essa instrução é inserida ao final da etapa de análise (descrita na Seção 4.2.3) e inicia sua execução antes do mecanismo obter a próxima iteração AVX com *store* alinhado à cache da VIMA. Notamos que, diferentemente da cópia de memória, quando isso acontece, não há uma perda de desempenho pelo sobrecusto causado por essa instrução adicional. Isso pode ser causado pela maior pressão exercida na memória por essa

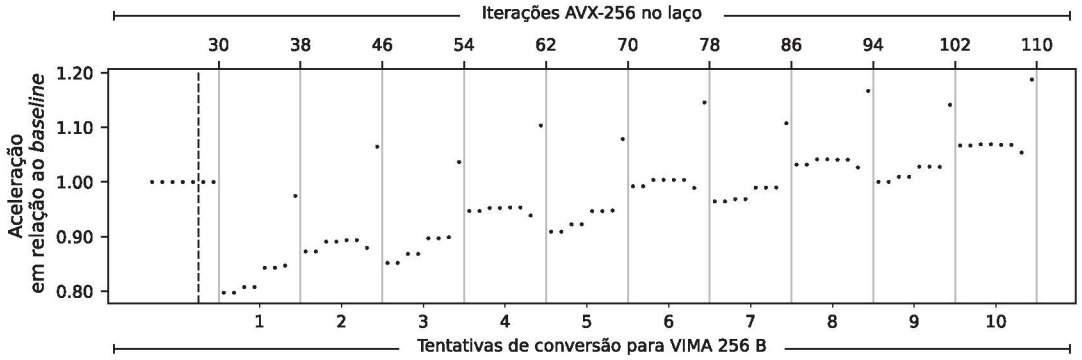


Figura A.13: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-256 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-256 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-256 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

aplicação, ocultando o custo da instrução adicionada ao ROB, porém uma compreensão acurada dos motivos dessa diferença exige estudos mais aprofundados e está além desta dissertação.

Cada divisão vertical sólida representa o início de uma tentativa de conversão do mecanismo.

Diferentemente da cópia de memória, a conversão de instruções AVX-256 para VIMA-256 B já leva a ganhos de desempenho, o que pode ser atribuído ao maior número de dados operado por cada uma de suas iterações (um *load* adicional).

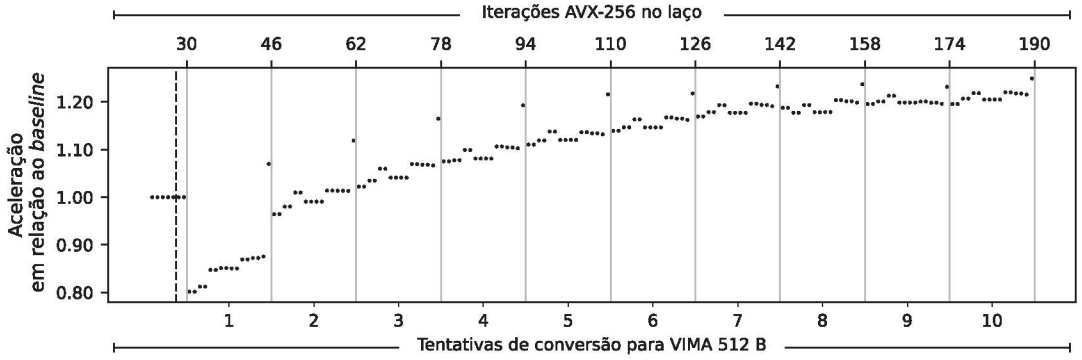


Figura A.14: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-512 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-256 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-512 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

A Figura A.14 apresenta a aceleração obtida por nosso mecanismo durante a conversão do Código 5.2, compilado com instruções AVX-256, para VIMA-512 B. Podemos observar um padrão parecido com o apresentado nas conversões para VIMA-256 B, porém com um crescimento mais acentuado e menores quedas no desempenho durante falhas de conversão.

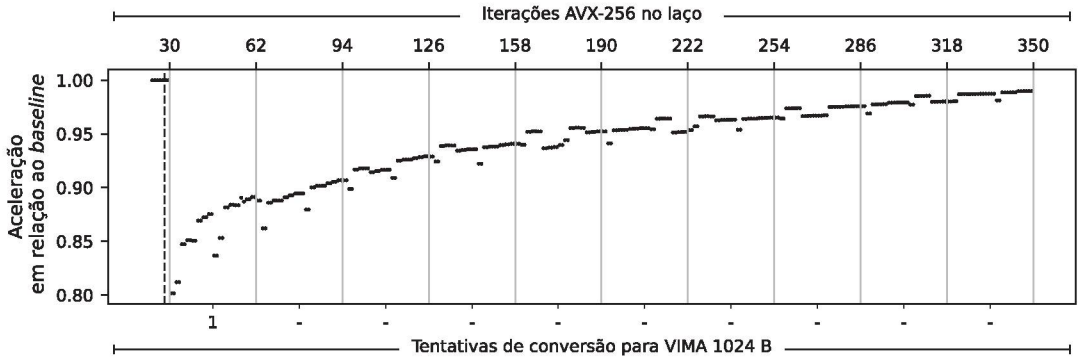


Figura A.15: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-1024 B, pelo SAPIVe durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-256 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-1024 B. Como o número de entradas no ROB de nosso processador adotado não é suficiente para realizar essas conversões, a primeira tentativa de conversão falha. Além disso, devido à nossa lista de conversões falhas, novas tentativas de conversão não são realizadas. Por conta disso, após a conversão incorreta, novas iterações gradualmente mitigam as perdas do mecanismo. Fonte: Autor.

A Figura A.15 apresenta a aceleração obtida por nosso mecanismo durante a conversão do Código 5.2, compilado com instruções AVX-256, para VIMA-1024 B. Podemos observar que, diferentemente da cópia de memória, para a soma vetorial, apenas a conversão para VIMA-1024 B não foi capaz de obter ganhos de desempenho. Isso, porque para esse tamanho de conversão, o número de instruções que precisariam ser travadas no ROB é superior ao tamanho de ROB utilizado. Dessa forma, após a primeira conversão falhar, o mecanismo não realiza novas conversões. Essa grande perda de desempenho inicial é gradualmente mitigada pela execução convencional das iterações restantes do laço, então laços maiores sofrem menos com a conversão falha (por isso a aproximação de 1.00).

Vecsum - AVX-512

A Figura A.16 apresenta o comportamento de nosso mecanismo ao realizar conversões para VIMA-256 B, sobre o algoritmo **vecsum** (Código 5.2), compilado para AVX-512. Cada ponto representa a aceleração obtida pelo SAPIVe para um número de iterações AVX-512, em relação ao nosso *baseline*, o código executado de maneira convencional pela CPU. O número máximo de iterações AVX apresentado foi selecionado de forma que nosso mecanismo pudesse gerar ao menos 10 conversões VIMA bem sucedidas.

A primeira divisão vertical tracejada indica a primeira inserção de uma instrução VIMA, gerada por nosso mecanismo no *pipeline*. Essa instrução é inserida ao final da etapa de análise (descrita na Seção 4.2.3) e inicia sua execução antes do mecanismo obter a próxima iteração AVX com *store* alinhado à cache da VIMA. Notamos que, quando isso acontece, pode ocorrer uma perda de desempenho por conta do sobrecusto causado por essa instrução adicional, porém isso não ocorre com essa aplicação.

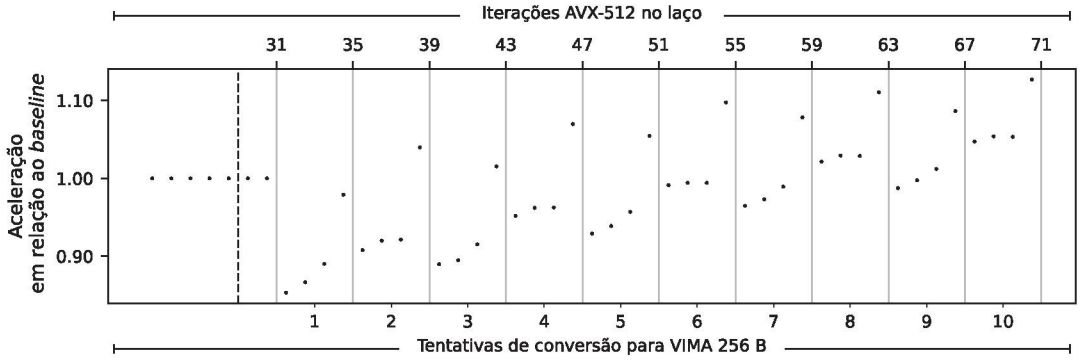


Figura A.16: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-512, sendo convertido para VIMA-256 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-512 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-512 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-256 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

Cada divisão vertical sólida representa o início de uma tentativa de conversão do mecanismo.

A Figura A.18 apresenta a aceleração obtida por nosso mecanismo durante a conversão do Código 5.2, compilado com instruções AVX-512, para VIMA-512 B. Podemos observar um padrão parecido com o apresentado nas conversões para VIMA-256 B. Nesse caso, as conversões também apresentam melhorias em relação ao tempo de execução da aplicação.

A Figura A.19 apresenta a aceleração obtida por nosso mecanismo durante a conversão do Código 5.2, compilado com instruções AVX-512, para VIMA-1024 B.

Todas as conversões de AVX-512, nesta aplicação, foram capazes de levar a ganhos de desempenho, sendo que esses ganhos aumentam para tamanhos maiores de instrução VIMA, de forma similar ao observado nas conversões de AVX-256 para VIMA-256 B e VIMA-512 B. Mesmo com a falha de conversão das instruções AVX-256 para VIMA-1024 B, devido a limitações do ROB, a conversão de AVX-512 para VIMA-1024 B pôde ser realizada, gerando ganhos de desempenho logo após a primeira conversão bem-sucedida. Esses ganhos rápidos, observados também nas Figuras A.9 e A.12, demonstram as vantagens da utilização de instruções VIMA maiores, devido ao seu melhor uso do paralelismo fornecido pelas memórias 3D e à menor necessidade de comunicação pela interconexão entre a CPU e a memória.

A.0.2 Relação entre riscos e ganhos

Esta seção estabelece uma relação entre os riscos e ganhos de cada nova conversão. Para cada conversão adicional, o risco é dado pela perda de desempenho causada por uma invalidação. Essa invalidação pode tanto ser gerada pelo fim do laço, quanto por algum desvio de fluxo, acesso à memória ou alteração indevida no padrão durante a conversão, como ilustrado pela Figura A.20. Já o ganho ocorre quando atingimos a iteração necessária para a nova conversão ser completada, sem que uma invalidação tenha ocorrido, como ilustrado pela Figura A.21.

Para calcular os riscos e ganhos médios de cada tipo de conversão avaliado na Seção A.0.1, utilizamos como referência cada ponto onde a conversão foi bem sucedida, o que é destacado,

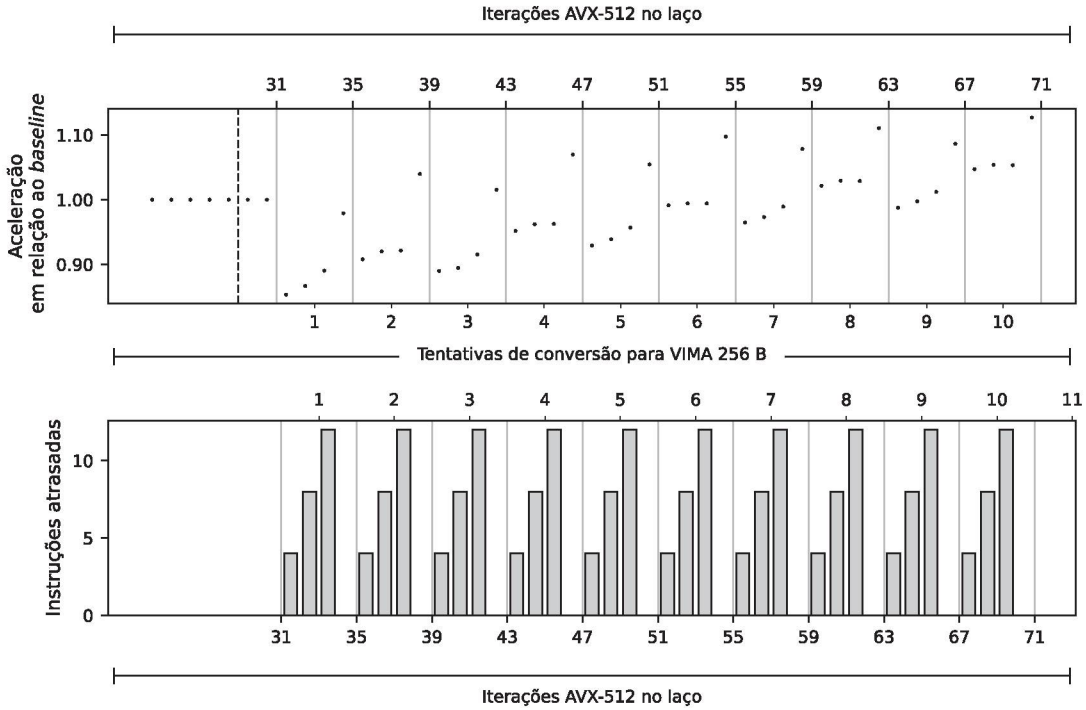


Figura A.17: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo, e o número de instruções com sua execução bloqueada no ROB devido a tentativas falhas de conversão para VIMA. O algoritmo avaliado foi compilado com instruções AVX-512, sendo convertido para VIMA-256 B, por nosso mecanismo, durante a execução. A aceleração é apresentada no gráfico superior, onde cada ponto a representa para um número diferente de iterações AVX-512 no programa convertido. O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. O número de instruções com sua execução atrasada em uma tentativa falha de conversão é apresentado no gráfico inferior. Foram avaliadas iterações AVX-512 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-256 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

para conversões entre instruções AVX-256 e VIMA-512 B do algoritmo de soma vetorial, na Figura A.22. Assim, após cada conversão bem-sucedida, os ganhos de uma nova conversão são dados pela porcentagem de melhora entre a aceleração dessa conversão e a aceleração da próxima conversão bem-sucedida, o que é ilustrado pela Figura A.23, onde as setas representam os ganhos para cada nova conversão. Observe que, em algumas situações, esses ganhos podem ser negativos, como ocorre durante a conversão de AVX-256 para VIMA-256 B, nesse mesmo algoritmo, o que é apresentado na Figura A.24.

Da mesma maneira, os riscos de uma nova conversão são dados pela porcentagem de piora entre conversões bem sucedidas, com um número exato de iterações, e as próximas conversões falhas, destacadas na Figura A.25, para conversões entre AVX-256 e VIMA-512 B no algoritmo de soma vetorial. Como cada conversão bem sucedida é seguida por diversas iterações, até formar a próxima conversão bem sucedida, optamos por avaliar o risco como a porcentagem de piora entre cada conversão bem sucedida e a média das conversões subsequentes com um número insuficiente de iterações, antes da próxima conversão bem sucedida. Isso é apresentado na Figura A.26, em conversões de AVX-256 para VIMA-512 B no algoritmo de soma vetorial. Nessa figura, as médias de conversões com um número insuficiente de iterações são apresentadas

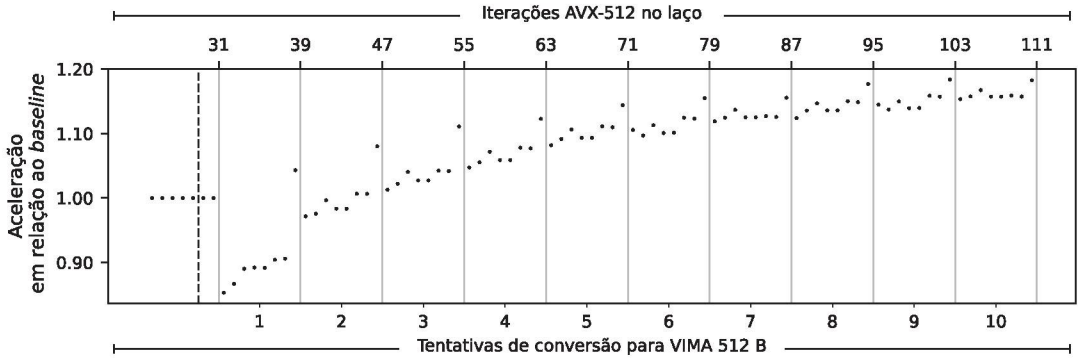


Figura A.18: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-512, sendo convertido para VIMA-512 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-512 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-512 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-512 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

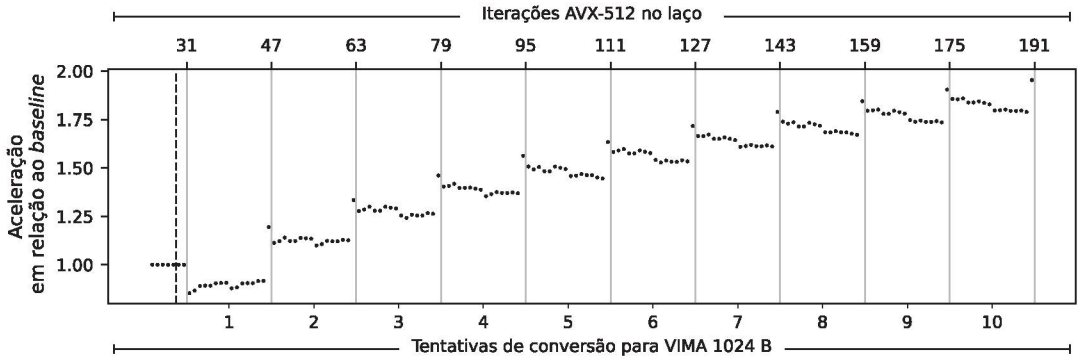


Figura A.19: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-512, sendo convertido para VIMA-1024 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-512 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-512 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-1024 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Fonte: Autor.

por linhas horizontais, sendo que as setas representam os riscos de uma próxima conversão para cada conversão bem sucedida.

Visto que os ganhos e perdas variam de conversão para conversão, apresentamos sua média, nas primeiras 10 tentativas de conversão (9 conversões bem sucedidas), na Tabela A.1. Temos que, aplicações com um ganho positivo são aquelas que ganhariam desempenho com nosso mecanismo, após certo número de iterações, sem os adiantamentos de uma maior agressividade. Já naquelas com ganhos negativos, o SAPIVe apenas degrada o tempo de execução, caso adiantamentos não sejam realizados.

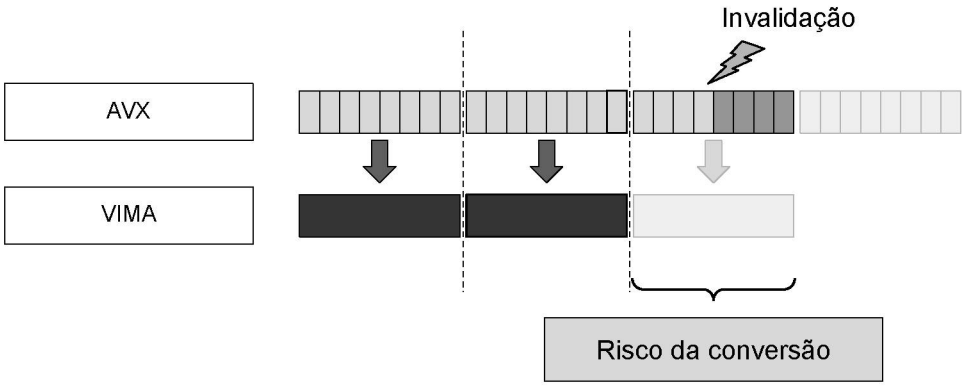


Figura A.20: O risco de uma nova tentativa de conversão é o fim de laço ou uma invalidação durante as instruções AVX da conversão. Fonte: Autor.

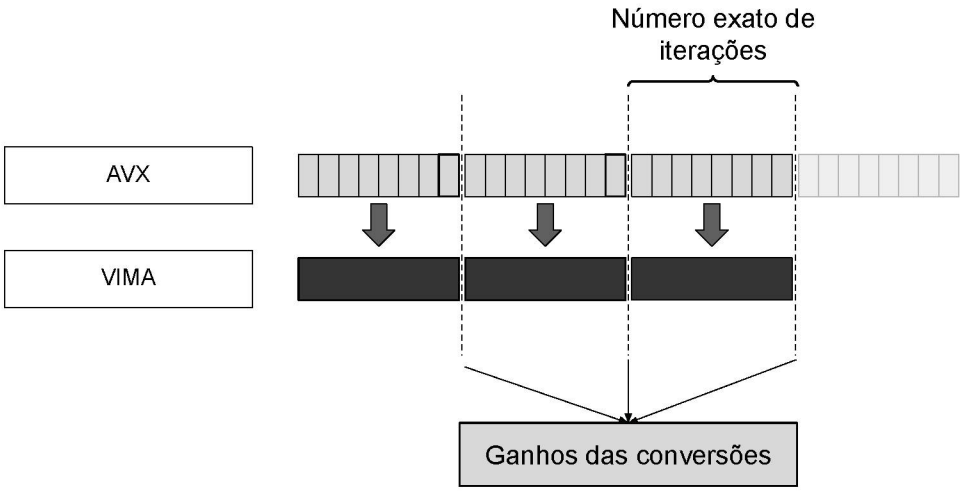


Figura A.21: O ganho de uma nova tentativa de conversão é obtido quando o número de iterações necessárias para a conversão é encontrado, sem que tenha ocorrido uma invalidação. Fonte: Autor.

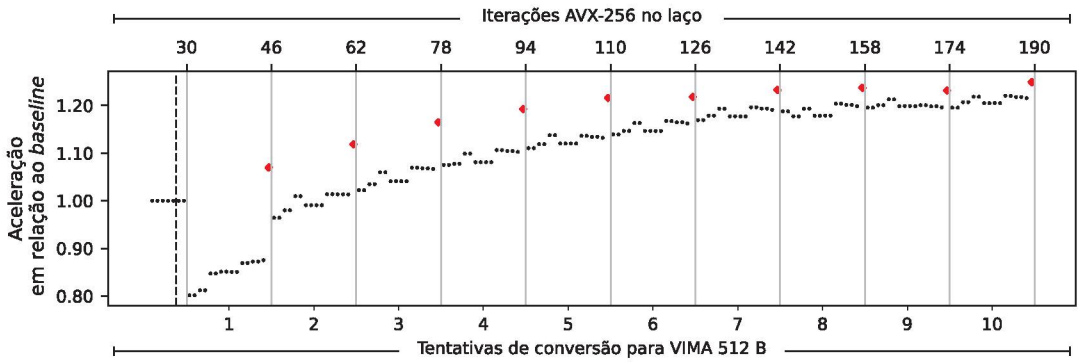


Figura A.22: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-512 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-256 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-512 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Os pontos em destaque representam conversões bem sucedidas com o número exato de iterações. Fonte: Autor.

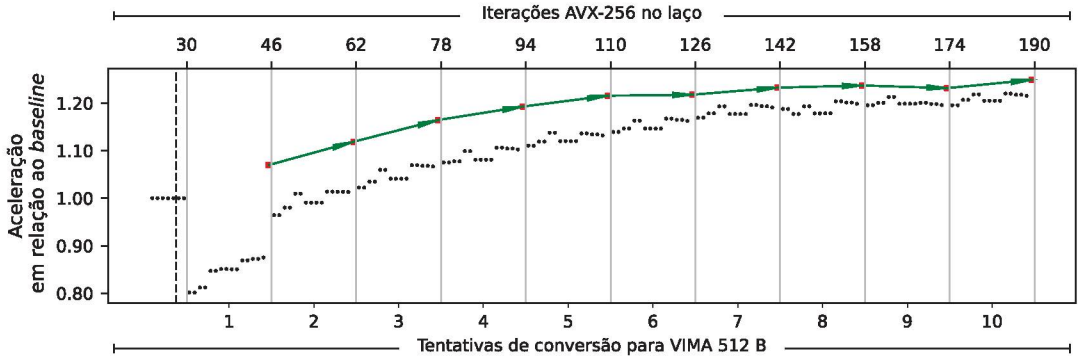


Figura A.23: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-512 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-256 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-512 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Os pontos em destaque representam conversões bem sucedidas com o número exato de iterações. As setas entre eles indicam os potenciais ganhos ao se fazer uma conversão adicional. Fonte: Autor.

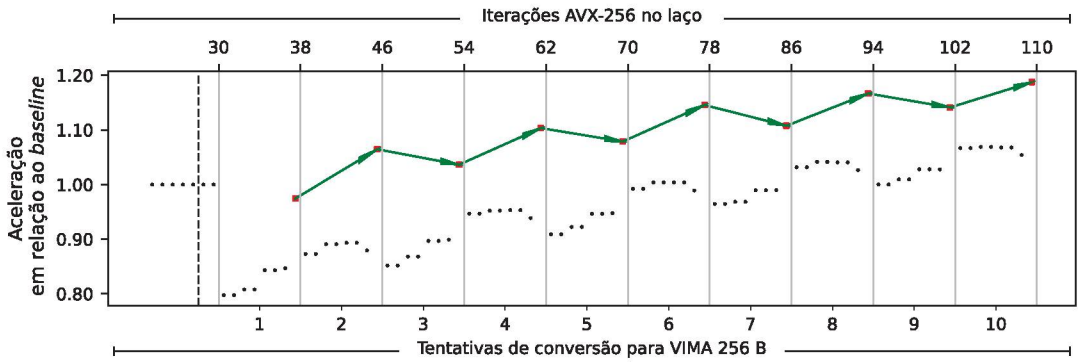


Figura A.24: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-256 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-256 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-256 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Os pontos em destaque representam conversões bem sucedidas com o número exato de iterações. As setas entre eles indicam os potenciais ganhos ao se fazer uma conversão adicional. Algumas conversões adicionais podem levar a perdas de desempenho. Fonte: Autor.

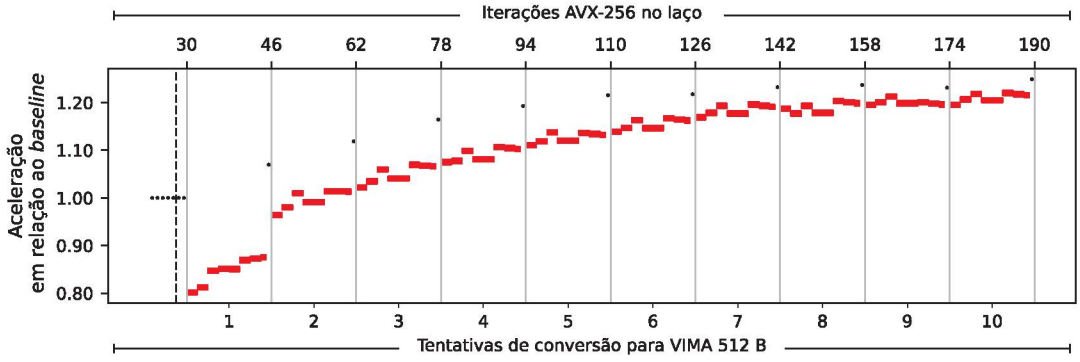


Figura A.25: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-512 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-256 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-512 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Os pontos em destaque representam tentativas de conversão com um número insuficiente de instruções. Fonte: Autor.

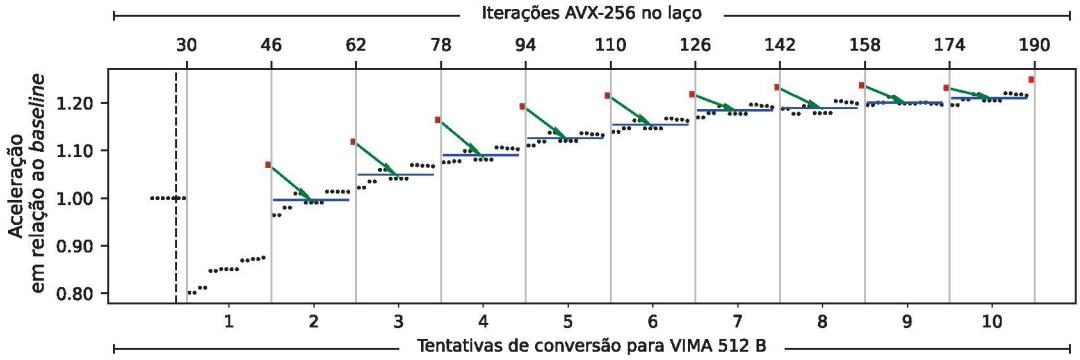


Figura A.26: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo. O algoritmo avaliado foi compilado com instruções AVX-256, sendo convertido para VIMA-512 B, pelo SAPIVe, durante a execução. Cada ponto apresenta a aceleração obtida para um número diferente de iterações AVX-256 no programa convertido (horizontal). O *baseline* considerado é uma execução convencional em CPU. A linha vertical tracejada indica o número de iterações necessário para que o mecanismo lance a instrução VIMA correspondente à primeira tentativa de conversão. As linhas verticais contínuas indicam o início das iterações correspondentes a cada conversão. Foram avaliadas iterações AVX-256 suficientes para nosso mecanismo realizar 10 conversões bem sucedidas para VIMA-512 B. Visto que uma tentativa de conversão apenas é bem sucedida se existirem iterações AVX em número suficiente, o número de conversões bem sucedidas é sempre uma conversão menor que o de tentativas realizadas, neste experimento. Os pontos em destaque representam conversões bem sucedidas com o número exato de iterações. As linhas horizontais representam a média de perdas por um número insuficiente de iterações, entre cada tentativa de conversão bem sucedida. As setas entre eles indicam as potenciais perdas ao se fazer uma conversão adicional que precise ser invalidada. Fonte: Autor.

Visto que nenhuma conversão da soma entre vetores, compilada com instruções AVX-256, para VIMA-1024 B pôde ser realizada, não há como calcular sua média de ganhos e perdas. Além disso, podemos notar que os riscos tendem a ser maiores que os ganhos durante essas conversões, sendo que a situação só é invertida com o uso de instruções VIMA-1024 B.

Tabela A.1: Risco médio de uma tentativa de conversão invalidada em relação ao ganho obtido por uma conversão bem sucedida. Ganhos negativos representam uma degradação no desempenho pelas conversões do SAPIVe. Visto que a aplicação **vecsum** compilada com AVX-256 não pôde ser convertida para VIMA-1024 B devido a limitações no tamanho do ROB, não há ganhos ou perdas para esse experimento. Fonte: Autor.

Aplicação	**Versão AVX**	**Versão VIMA**	**Perda média conversão interrompida**	**Ganho médio conversão bem sucedida**
memcpy	256	256	6.53%	-0.10%
		512	2.99%	-1.09%
		1024	2.99%	2.43%
	512	256	4.79%	-1.47%
		512	2.66%	-1.33%
		1024	2.40%	2.51%
vecsum	256	256	11.05%	2.32%
		512	4.57%	1.75%
		1024	–	–
	512	256	8.04%	1.63%
		512	3.09%	1.41%
		1024	4.83%	5.66%

A.0.3 Efeito da agressividade

A seção anterior apresentou o comportamento do SAPIVe no início de suas conversões. Nesta seção, avaliamos o efeito de diversas agressividades sobre nosso mecanismo, a fim de entender como essa tecnologia poderia acelerar as conversões, considerando agressividades de 0 a 8, 16 e 32 instruções VIMA.

Nossas avaliações consideraram laços de até 4096 iterações AVX, tamanho suficiente para que as curvas de crescimento de desempenho se estabilizem. Os resultados obtidos são apresentados a seguir.

Memcpy - AVX-256

A Figura A.27 apresenta a aceleração obtida com diversas agressividades durante a conversão do Código 5.1, compilado em AVX-256, para VIMA-256 B. Os resultados mostram que com um adiantamento de duas conversões nosso mecanismo começa superar a execução usual em CPU. Esse desempenho chega próximo ao ápice a partir de um adiantamento de 4 conversões, quando atinge uma aceleração próxima a 1,75x.

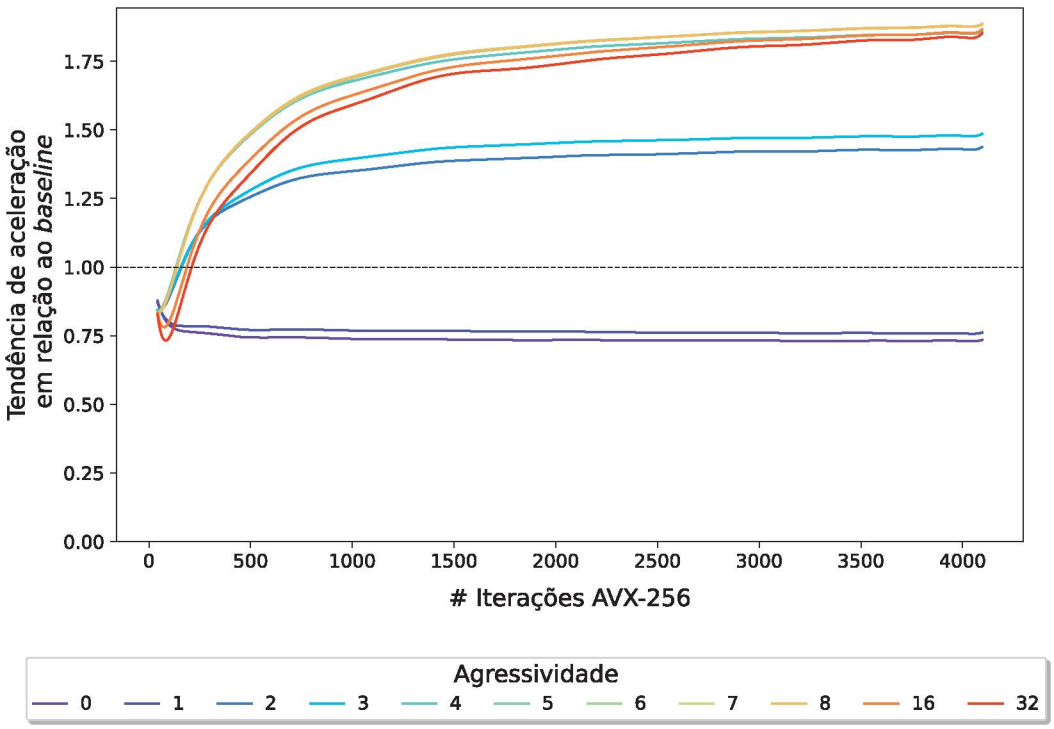


Figura A.27: Tendência de aceleração do algoritmo de cópia de memória, **memcpy**, apresentado no Código 5.1, obtida por nosso mecanismo em conversões de AVX-256 para VIMA-256 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU (*baseline*). Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Adotamos como maior teste um laço de 4096 iterações, visto que é o suficiente para a estabilização das curvas de crescimento dos ganhos de cada agressividade avaliada. Fonte: Autor.

Quando consideramos a conversão para VIMA-512 B, como apresentado na Figura A.28, temos que mesmo um adiantamento de uma conversão por vez já é capaz de gerar ganhos. Além disso, um desempenho similar ao máximo é atingido com uma agressividade menor, de apenas 2 adiantamentos.

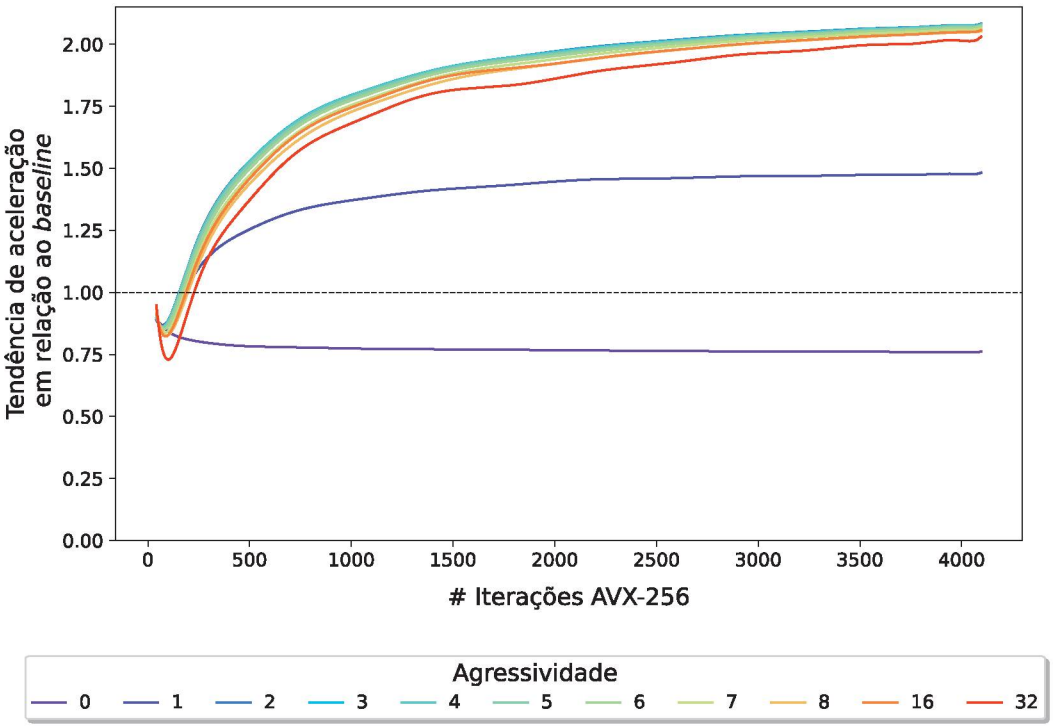


Figura A.28: Tendência de aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo em conversões de AVX-256 para VIMA-512 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU (*baseline*). Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Adotamos como maior teste um laço de 4096 iterações, visto que é o suficiente para a estabilização das curvas de crescimento dos ganhos de cada agressividade avaliada. Fonte: Autor.

A tendência de atingir o potencial máximo com uma menor agressividade é mantida nas conversões para VIMA-1024 B, como apresentado na Figura A.29. Essa conversão, além de obter ganhos de aceleração com e sem adiantamentos, ainda atinge ganhos próximos ao máximo utilizando apenas uma conversão adiantada.

Assim como nos resultados da seção anterior, conforme o tamanho da instrução VIMA resultante da conversão aumenta, os ganhos máximos também aumentam.

Além disso, as maiores agressividades (16 e 32 instruções), além de não levarem a ganhos de desempenho, ainda causaram algumas perdas em aceleração. Isso, porque com o ganho máximo do mecanismo atingido, novos adiantamentos apenas ocupam posições no ROB, atrasando a formação de novas conversões.

Memcpy - AVX-512

As Figuras A.30, A.31 e A.32 apresentam a aceleração obtida pelo uso de diversas agressividades durante a conversão do Código 5.1, compilado em AVX-512, para VIMA-256 B, VIMA-512 B e VIMA-1024 B, respectivamente. Diferentemente de suas versões em AVX-256, essas apresentaram uma maior variação de ganhos de desempenho, de acordo com a agressividade utilizada. Com isso, a aproximação do desempenho máximo de nosso mecanismo só foi atingida com adiantamentos de 8, 4 e 2 conversões, respectivamente. Apesar disso, a aceleração obtida foi maior, ultrapassando 3x nos três casos.

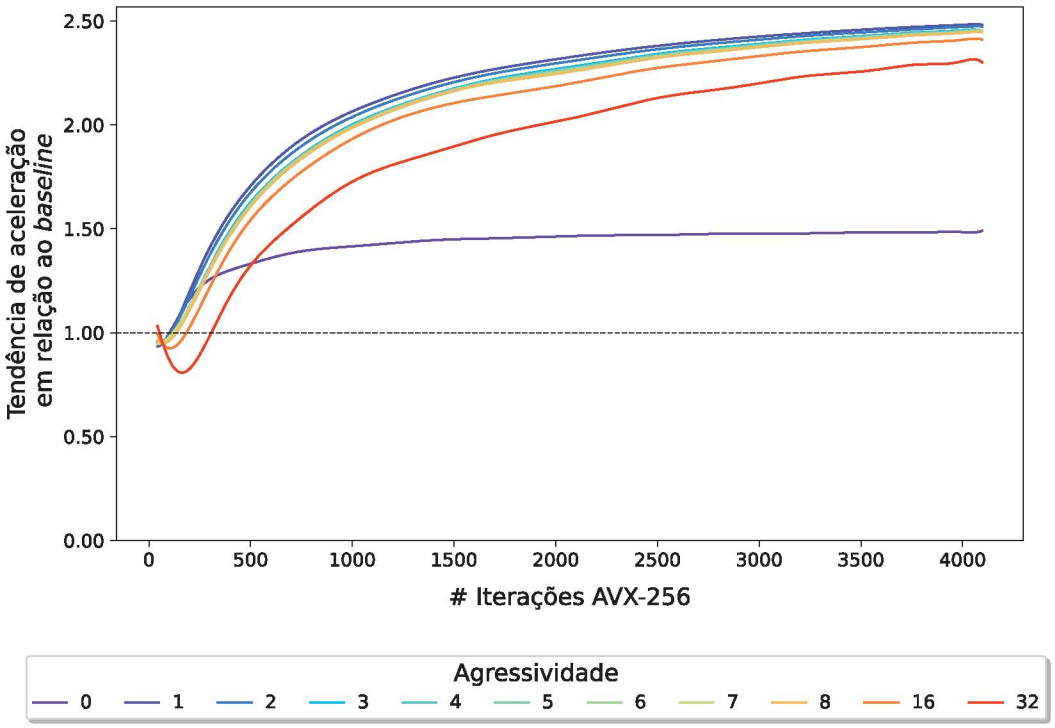


Figura A.29: Tendência de aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo em conversões de AVX-256 para VIMA-1024 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU (*baseline*). Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Adotamos como maior teste um laço de 4096 iterações, visto que é o suficiente para a estabilização das curvas de crescimento dos ganhos de cada agressividade avaliada. Fonte: Autor.

Notamos que, diferentemente da conversão de instruções AVX-256, nesse caso, nosso mecanismo precisa de agressividades maiores para atingir os ganhos máximos. Isso ocorre porque menos iterações AVX-512 são necessárias para a formação de cada instrução VIMA, reduzindo o gargalo dos cálculos da CPU. Assim, os adiantamentos se tornam mais efetivos, visto que são utilizados mais rapidamente.

Pela menor ocupação no ROB para cada conversão, o uso de agressividades maiores (16 e 32 instruções) não levou a degradações tão grandes quanto as vistas para o AVX-256. Apesar disso, não obtivemos melhores acelerações com essas agressividades maiores.

Vecsum - AVX-256

As Figuras A.33, A.34 e A.35 apresentam a aceleração obtida pelo uso de diversas agressividades durante a conversão do Código 5.2, compilado em AVX-256, para VIMA-256 B, VIMA-512 B e VIMA-1024 B, respectivamente. Os resultados mostram que a conversão para VIMA-256 B exige um adiantamento de 2 conversões para chegar próximo a seu potencial máximo, enquanto a conversão para VIMA-512 B já se aproxima dos melhores resultados com o adiantamento de apenas uma conversão. Da mesma forma que ocorre com a cópia de memória, o uso de instruções VIMA de maior tamanho reduz a necessidade de maiores agressividades para obter maiores ganhos. Isso, porque a própria instrução VIMA funciona como um adiantamento, operando sobre dados futuros na conversão especulativa.

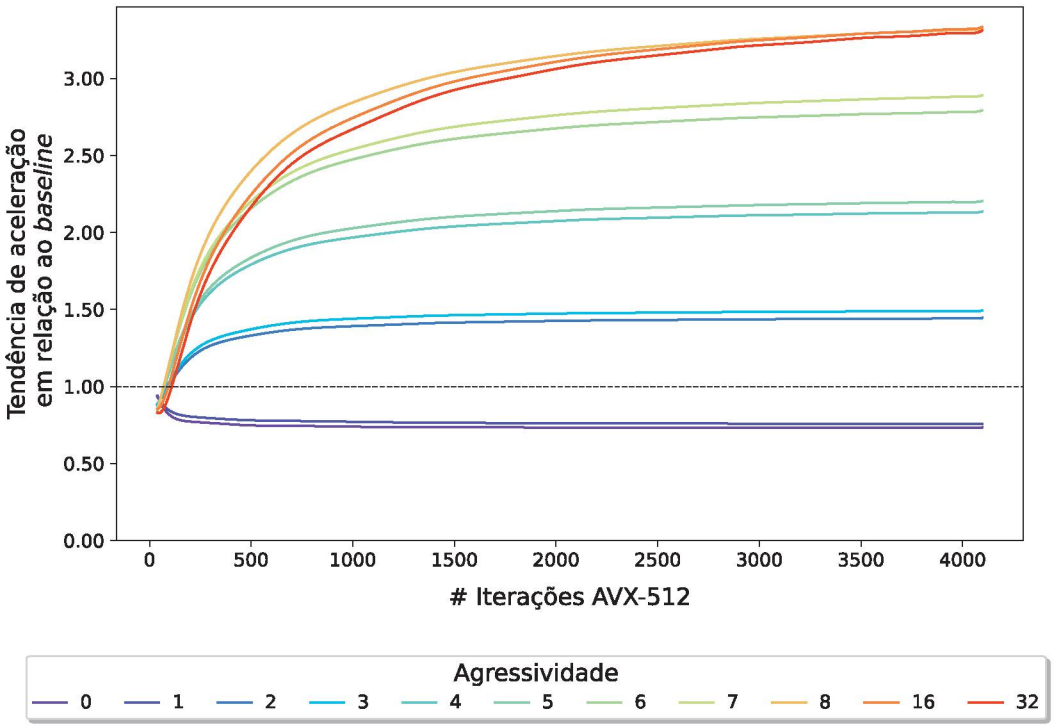


Figura A.30: Tendência de aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-256 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU (*baseline*). Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Adotamos como maior teste um laço de 4096 iterações, visto que é o suficiente para a estabilização das curvas de crescimento dos ganhos de cada agressividade avaliada. Fonte: Autor.

Como apresentado anteriormente, a conversão do laço **vecsum** compilado com AVX-256 para VIMA-1024 B não foi possível devido a limitações com o tamanho do ROB simulado. Por conta disso, independentemente da agressividade utilizada, essa conversão perde desempenho em laços pequenos. Apesar disso, essa perda é mitigada conforme outras iterações são executadas em laços maiores, ficando bem próxima e até ultrapassando o tempo de execução original. Essa aceleração é causada pela influência da instrução VIMA, da tentativa falha de conversão, à memória o que precisa ser melhor estudado em trabalhos futuros. Além disso, ela não deve ser considerada uma melhora real no desempenho dessa aplicação, visto que os efeitos dessa instrução poderiam ter sido prejudiciais em outras situações.

Diferentemente da cópia de memória, com instruções AVX-256, a soma vetorial, com instruções AVX-256, não apresentou perdas significativas de desempenho com grandes agressividades. O que pode ser causado pela maior pressão de memória exercida por essa aplicação, que oculta os efeitos negativos do maior número de adiantamentos no ROB.

Vecsum - AVX-512

As Figuras A.36, A.37 e A.38 apresentam a aceleração obtida pelo uso de diversas agressividades durante a conversão do Código 5.2, compilado em AVX-512, para VIMA-256 B, VIMA-512 B e VIMA-1024 B, respectivamente. Como aconteceu com o aumento no tamanho das instruções AVX no algoritmo **memcpy**, o uso de instruções AVX-512 permite um maior ganho de desempenho com adiantamentos.

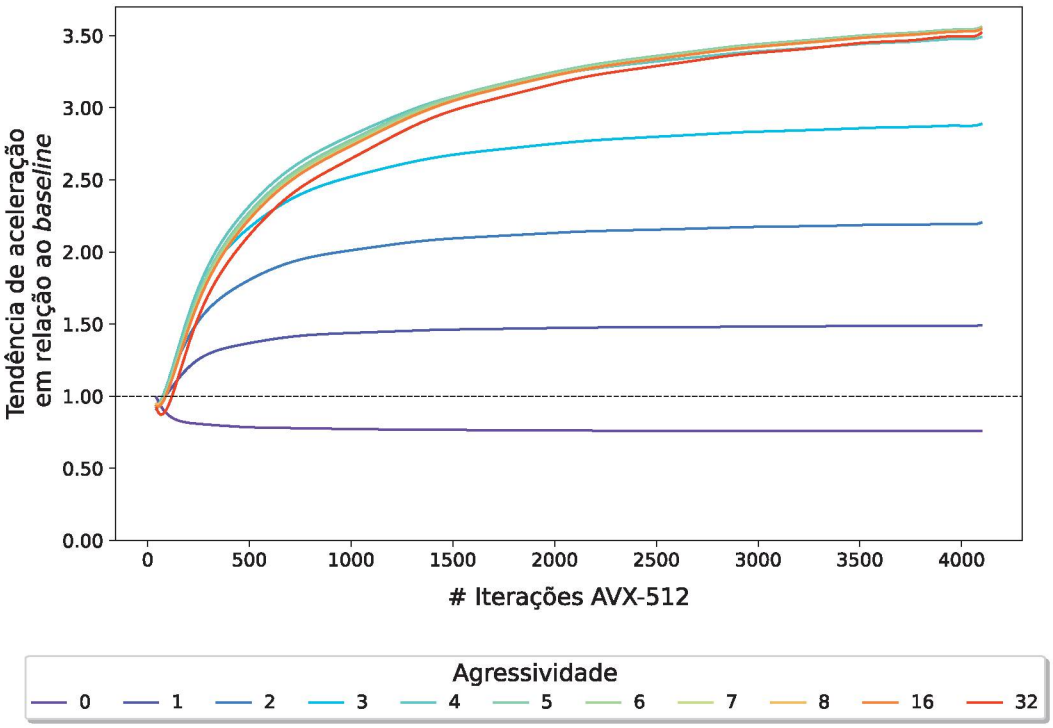


Figura A.31: Tendência de aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-512 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU (*baseline*). Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Adotamos como maior teste um laço de 4096 iterações, visto que é o suficiente para a estabilização das curvas de crescimento dos ganhos de cada agressividade avaliada. Fonte: Autor.

Com isso, a agressividade necessária para a obtenção do limite de ganhos aumentou, junto à aceleração obtida com nossas conversões. Esse aumento no limite é devido o menor número de instruções AVX-512 necessárias para formar cada instrução VIMA. Isso leva a uma necessidade de confirmar menos informações para cada conversão. Além disso, com o uso dos adiantamentos, nossas conversões são limitadas não pelo tempo de execução da VIMA, mas pelo tempo de todos os endereços das instruções convertidas serem calculados pela CPU. Portanto, a redução no número de instruções convertidas, devido ao uso do AVX-512, reduz esse gargalo, levando a maiores acelerações.

A.0.4 Comparação com implementação manual

Como apresentado na Seção A.0.3, nosso mecanismo é limitado tanto pela agressividade utilizada, quanto pelo tempo de processamento das conversões na CPU. Nesta seção, avaliamos como o SAPIVe se comporta quando comparado ao uso manual das instruções VIMA, por meio de rotinas *intrinsic*. Para isso, examinamos a aceleração obtida com o uso de nosso mecanismo e com o uso de *intrinsics*, em relação à execução usual na CPU (*baseline*) das aplicações avaliadas. Nossos testes consideraram operações sobre vetores de 8 MB, tamanho suficiente para nosso mecanismo e as *intrisics* atingirem seus ganhos/perdas máximas de desempenho. Cada teste considerou todas as variações de agressividade avaliadas na Seção A.0.3.

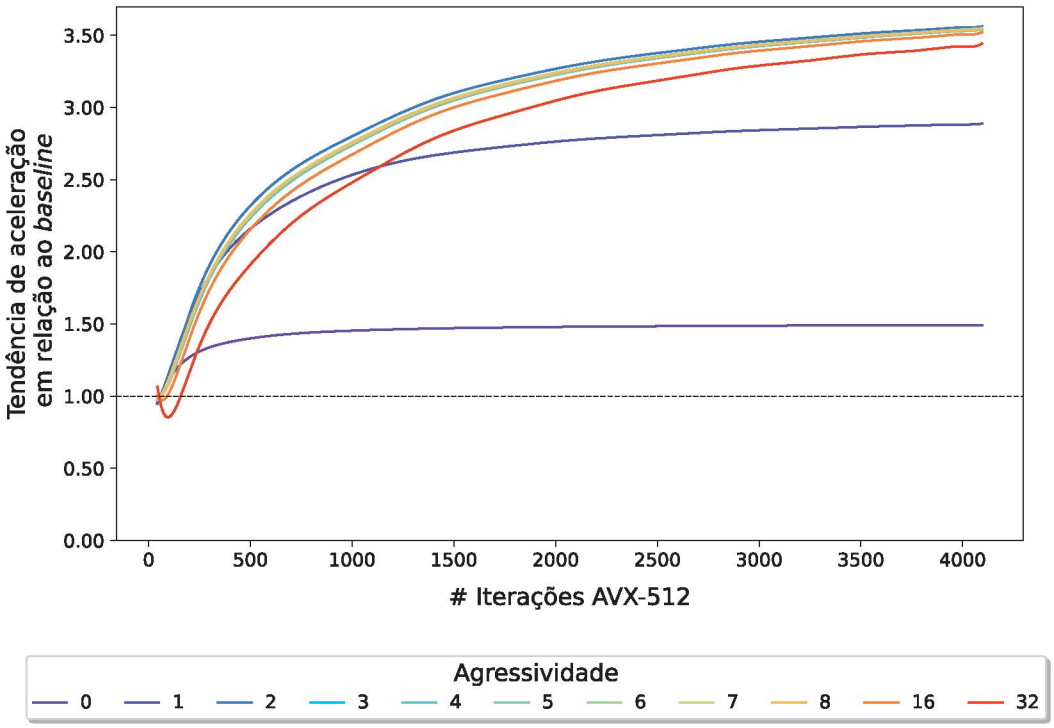


Figura A.32: Tendência de aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-1024 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Adotamos como maior teste um laço de 4096 iterações, visto que é o suficiente para a estabilização das curvas de crescimento dos ganhos de cada agressividade avaliada. Fonte: Autor.

Memcpy - AVX-256

A Figura A.39 apresenta a aceleração obtida pelo SAPIVe e pelo uso de *intrinsics* para o **memcpy** (Código 5.1), compilado com instruções AVX-256. Os resultados são apresentados, de cima para baixo, utilizando instruções VIMA-256 B, VIMA-512 B e VIMA-1024 B, respectivamente.

Podemos observar que nosso mecanismo obtém acelerações próximas a 49% das obtidos pelo uso de *intrinsics* com a VIMA-256 B, próximas a 24% com a VIMA-512 B e próximas a 29% com a VIMA-1024 B. Como apresentado anteriormente, os ganhos do SAPIVe são limitados pelo tempo de processamento das muitas iterações AVX que precisam confirmar a conversão. Por possuir um número significativamente menor de instruções, a implementação manual da VIMA não é afetada por esse gargalo, o que a permite atingir maiores acelerações.

Memcpy - AVX-512

A Figura A.40 apresenta a aceleração obtida pelo SAPIVe e pelo uso de *intrinsics* para o Código 5.1, dessa vez compilado com instruções AVX-512. Os resultados são apresentados, de cima para baixo, utilizando instruções VIMA-256 B, VIMA-512 B e VIMA-1024 B, respectivamente.

Podemos observar que nosso mecanismo obtém acelerações próximas a 88% das obtidas pelo uso de *intrinsics* com a VIMA-256 B, próximas a 43% com a VIMA-512 B e a 42% com a VIMA-1024 B.

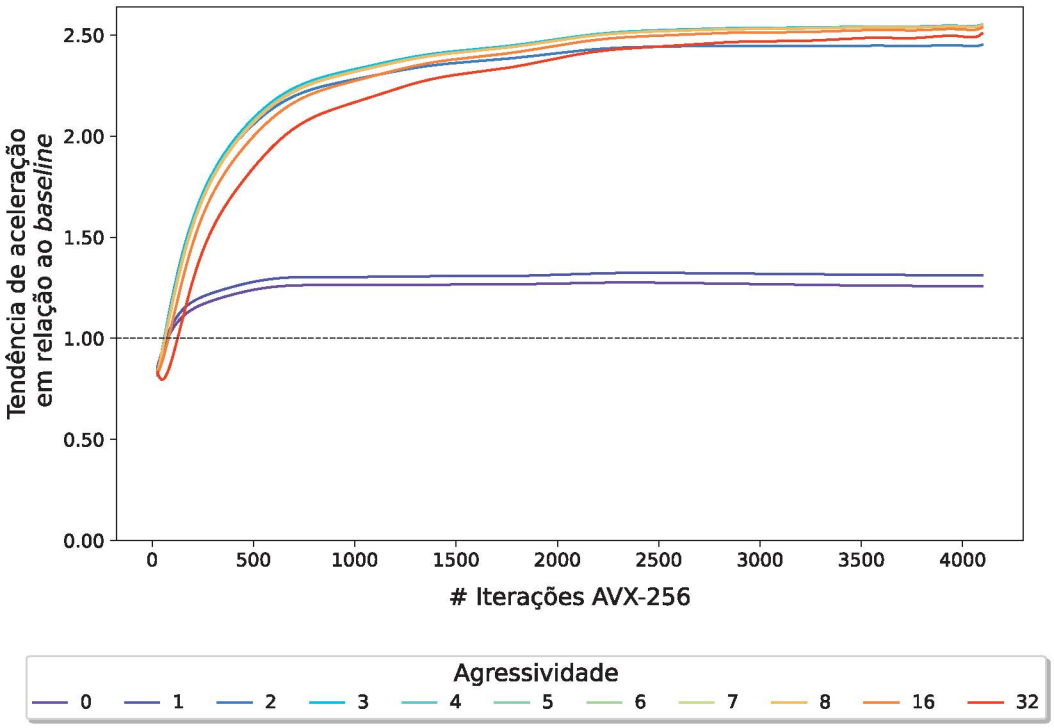


Figura A.33: Tendência de aceleração do algoritmo **vecsum**, de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-256 para VIMA-256 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU (*baseline*). Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Adotamos como maior teste um laço de 4096 iterações, visto que é o suficiente para a estabilização das curvas de crescimento dos ganhos de cada agressividade avaliada. Fonte: Autor.

Vecsum - AVX-256

A Figura A.41 apresenta a aceleração obtida pelo SAPIVe e pelo uso de *intrinsics* para o algoritmo **vecsum** (Código 5.2), compilado com instruções AVX-256. Os resultados são apresentados, de cima para baixo, utilizando instruções VIMA-256 B, VIMA-512 B e VIMA-1024 B, respectivamente.

Podemos observar que nosso mecanismo obtém ganhos próximos a 38% dos obtidos pelo uso de *intrinsics* com a VIMA-256 B e 24% com a VIMA-512 B. Visto que o SAPIVe não foi capaz de converter essa aplicação para VIMA-1024 B, todas as barras dele estão próximas a 1 no gráfico correspondente a essa conversão.

Vecsum - AVX-512

A Figura A.42 apresenta a aceleração obtida pelo SAPIVe e pelo uso de *intrinsics* para o **vecsum** (Código 5.2), dessa vez compilado com instruções AVX-512. Os resultados são apresentados, de cima para baixo, utilizando instruções VIMA-256 B, VIMA-512 B e VIMA-1024 B, respectivamente.

Podemos observar que nosso mecanismo obtém ganhos próximos a 68% dos obtidos pelo uso de *intrinsics* com a VIMA-256 B, próximos a 49% com a VIMA-512 B e próximos a 50% com a VIMA-1024 B.

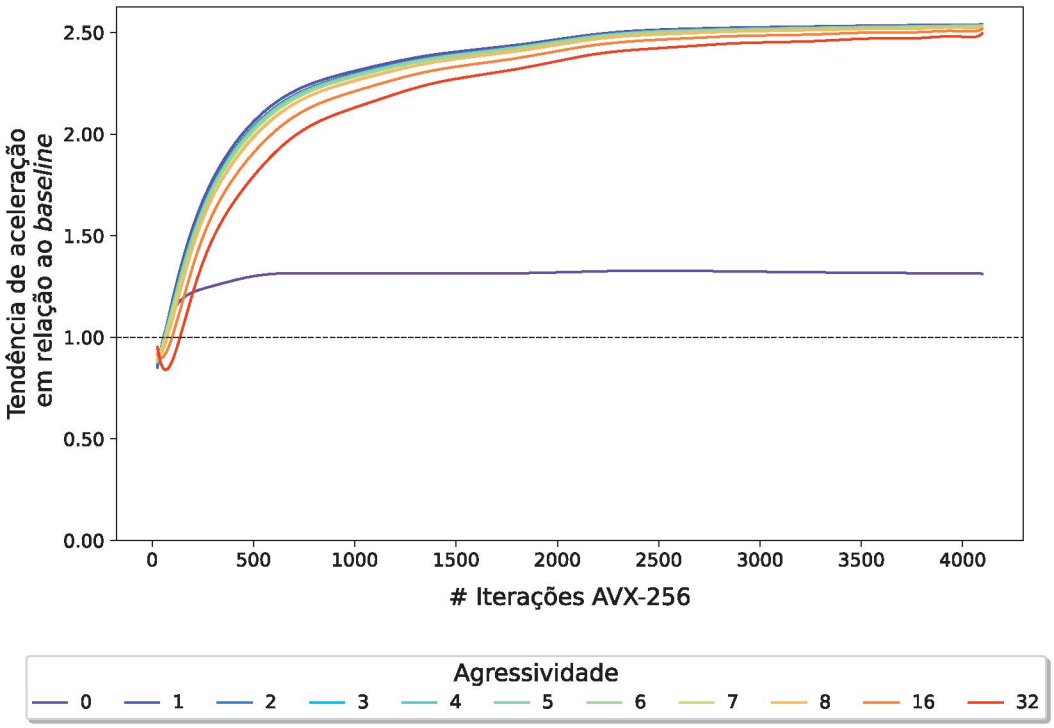


Figura A.34: Tendência de aceleração do algoritmo **vecsum**, de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-256 para VIMA-512 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU (*baseline*). Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Adotamos como maior teste um laço de 4096 iterações, visto que é o suficiente para a estabilização das curvas de crescimento dos ganhos de cada agressividade avaliada. Fonte: Autor.

Em ambas as aplicações avaliadas, nossos ganhos são mais próximos aos obtidos pelo uso de *intrinsics* com a VIMA-256 B, visto que essa versão exige que menos instruções sejam bloqueadas no ROB pelo SAPIVe. Além disso, aumentar a agressividade além de 8 não gerou ganhos significativos em aceleração para nenhuma combinação de aplicação e AVX. Isso, porque o tamanho do ROB limita o tempo de processamento de nossas conversões, limitando os ganhos com a VIMA.

Outro ponto interessante é o aumento no desempenho do SAPIVe em relação às *instrinsics* quando consideramos instruções AVX-512. Isso ocorre porque um número menor de instruções precisa ser confirmado por conversão, em relação ao AVX-256, reduzindo o gargalo do processamento em CPU para validar cada conversão.

A.0.5 Efeito de aprimoramentos no sistema

As conversões realizadas pelo SAPIVe exigem a retenção de instruções no *pipeline*. Isso limita o ganho de desempenho, visto que apenas um pequeno número delas pode ser mantido no ROB e verificado simultaneamente.

Apesar disso, recentes versões de processadores têm mostrado uma expansão em suas estruturas. A Tabela 5.2 apresenta um vislumbre dessa tendência, exibindo o tamanho do ROB em algumas arquiteturas da Intel, de acordo com seu ano de lançamento.

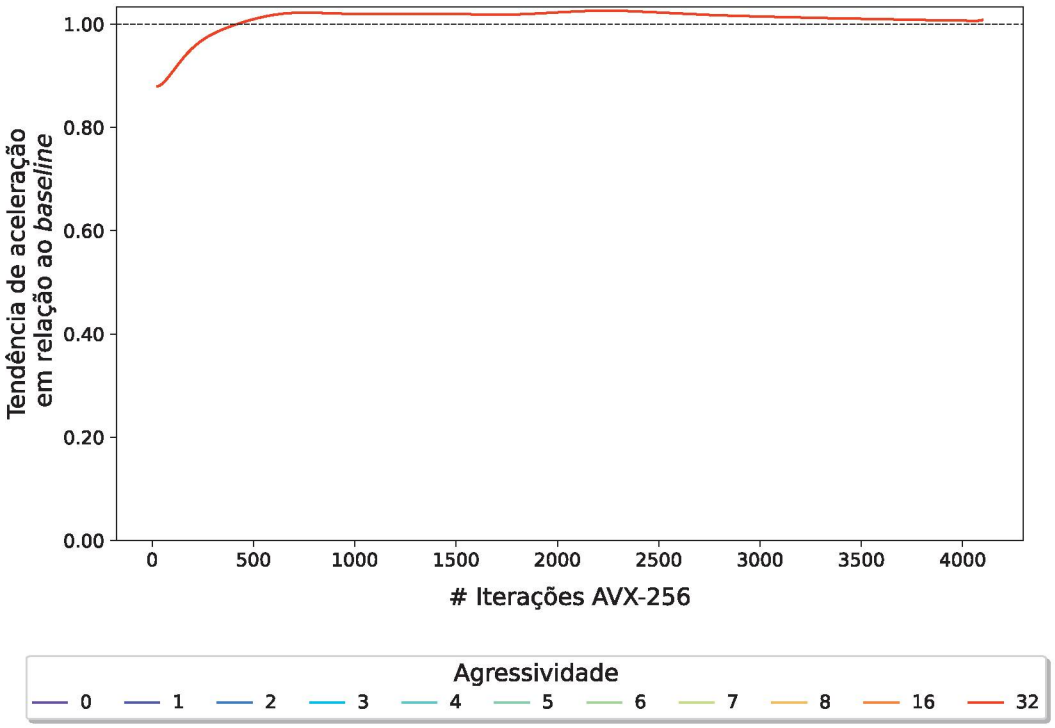


Figura A.35: Tendência de aceleração do algoritmo **vecsum**, de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-256 para VIMA-1024 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU (*baseline*). Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Observe que esse algoritmo não pôde ser convertido, por conta disso, após a primeira invalidação, ele é adicionado à lista de conversões falhas. Assim, nosso mecanismo não tenta novas conversões e a agressividade não gera qualquer alteração. Fonte: Autor.

Para avaliar os efeitos que essa constante melhoria teria sobre o SAPIVe, aumentamos o tamanho de todos os *buffers* do sistema e a largura de todos os estágios do *pipeline* em quatro vezes. Em nossas avaliações, esse aumento não gerou alterações significativas no tempo de execução sem nosso mecanismo de conversão, o que indica um gargalo em outras partes do sistema, para aplicações limitadas pelo tempo de acesso à memória, porém esse estudo foge ao escopo desta dissertação.

Os tópicos seguintes apresentam os ganhos no tempo de execução, obtidos pelo SAPIVe em um sistema expandido, em relação a execuções sem nosso mecanismo, sobre esse mesmo sistema. Não limitamos o número de conversões que podem ser realizadas simultâneamente com os conjuntos do ROB, por exemplo, com um ROB de 896 ($224 * 4$) entradas, podemos realizar até 23 conversões de AVX-512 para VIMA-256 B ($256/64 = 4$ iterações por conversão), considerando 10 entradas no ROB por iteração ($\lceil\lceil(896/10)\rceil/4\rceil$).

Memcpy - AVX-256

As Figuras A.43, A.44 e A.45 apresentam a aceleração obtida pelo SAPIVe em relação ao novo sistema, variando a agressividade, durante a conversão do Código 5.1, compilado em AVX-256, para instruções VIMA-256 B, VIMA-512 B e VIMA-1024 B, respectivamente.

Podemos notar que o maior tamanho das estruturas levou a maiores ganhos de desempenho para todas as versões avaliadas. Esses ganhos foram superiores 3,5x em todos os

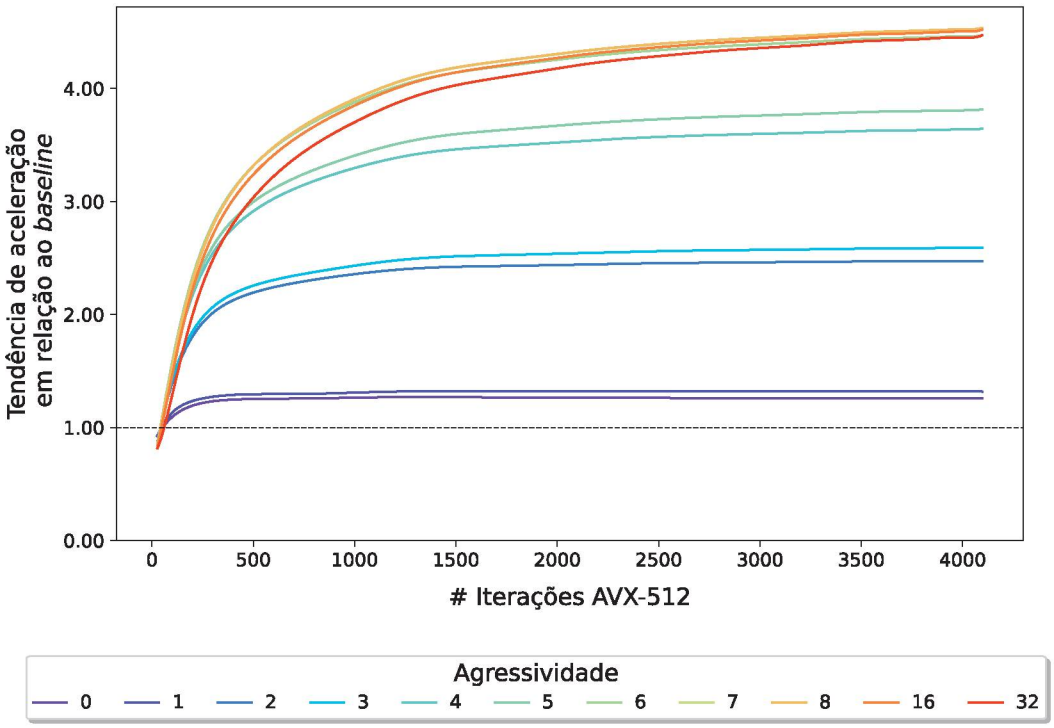


Figura A.36: Tendência de aceleração do algoritmo **vecsum**, de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-256 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU (*baseline*). Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Adotamos como maior teste um laço de 4096 iterações, visto que é o suficiente para a estabilização das curvas de crescimento dos ganhos de cada agressividade avaliada. Fonte: Autor.

casos, chegando a quase duas vezes os ganhos obtidos para o sistema original, apresentado na Tabela 2.2. Além disso, conforme conversões para instruções VIMA de maior tamanho foram utilizadas, o uso de agressividades maiores passou a ser prejudicial ao sistema. Isso ocorre porque conversões para instruções VIMA maiores exigem que mais entradas do ROB sejam bloqueadas por conversão. Assim, menos conversões paralelas podem ser realizadas. Por conta disso, cada entrada do ROB ocupada por adiantamentos degrada ainda mais o desempenho do mecanismo. Porém, estudos mais aprofundados precisam ser realizados para a compreensão total deste fenômeno.

A Figura A.46 apresenta uma comparação dos novos ganhos do SAPIVe em relação ao uso de rotinas *intrinsic*. O eixo horizontal de cada gráfico indica a agressividade utilizada pelo SAPIVe e a implementação utilizando *intrinsics*. No eixo vertical temos os ganhos obtidos por cada versão em relação ao *baseline*, um sistema com a largura de todos os *buffers* e de todos os estágios do *pipeline* quatro vezes o tamanho original. Tanto nosso mecanismo quanto as *intrinsics* também são executados considerando esse sistema expandido. Os diferentes gráficos nessa figura representam diferentes tamanhos de instrução VIMA utilizada para as conversões e *intrinsics*, o que é indicado ao lado esquerdo de cada gráfico. Consideramos uma operação sobre vetores de 8 MB, da mesma forma que em experimentos anteriores.

Como esperado, a expansão das estruturas do *pipeline* permitiu a obtenção de maiores ganhos de desempenho pelo SAPIVe, chegando a 99%, 50% e 52% da aceleração obtida por *intrinsics*, considerando instruções VIMA de 256 B, 512 B e 1024 B, respectivamente. Isso,

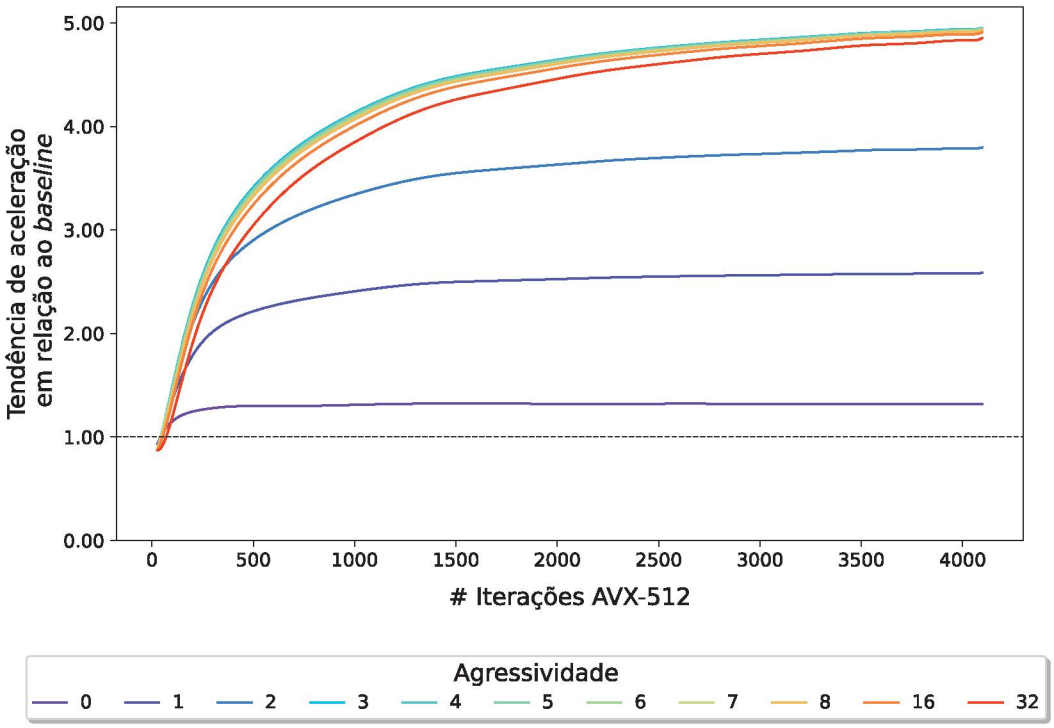


Figura A.37: Tendência de aceleração do algoritmo de cópia de memória, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-512 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU (*baseline*). Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Adotamos como maior teste um laço de 4096 iterações, visto que é o suficiente para a estabilização das curvas de crescimento dos ganhos de cada agressividade avaliada. Fonte: Autor.

porque com a expansão das estruturas, um número maior de conversões simultâneas pode ser validado, permitindo ao mecanismo aproveitar de maneira mais eficiente os adiantamentos. Por conta disso, ganhamos mais desempenho que no *pipeline* original, podendo aproveitar os ganhos dos adiantamentos com 16 e 32 instruções VIMA.

Apesar disso, os ganhos proporcionais das conversões em relação às *instrinsics* foram reduzidos conforme utilizamos instruções VIMA maiores. Isso pode ser explicado pelo maior número de instruções retidas por conversão em instruções VIMA maiores, o que reduz as vantagens do *pipeline* mais largo e reduz o aproveitamento de maiores agressividades.

Memcpy - AVX-512

As Figuras A.47, A.48 e A.49 apresentam a aceleração obtida pelo SAPIVe, em relação ao novo sistema, variando a agressividade, durante a conversão do Código 5.1, compilado em AVX-512, para instruções VIMA-256 B, VIMA-512 B e VIMA-1024 B, respectivamente.

Podemos notar que o maior tamanho das estruturas levou a ganhos de desempenho para todas as versões avaliadas, praticamente dobrando os ganhos obtidos nas conversões para AVX-512 B e VIMA-1024 B, em relação aos ganhos obtidos para o sistema original, descrito na Tabela 2.2.

A Figura A.50 apresenta uma comparação dos novos ganhos do SAPIVe em relação ao uso de rotinas *intrinsic*. O eixo horizontal de cada gráfico indica a agressividade utilizada pelo SAPIVe e a implementação utilizando *intrinsics*. No eixo vertical temos os ganhos obtidos por

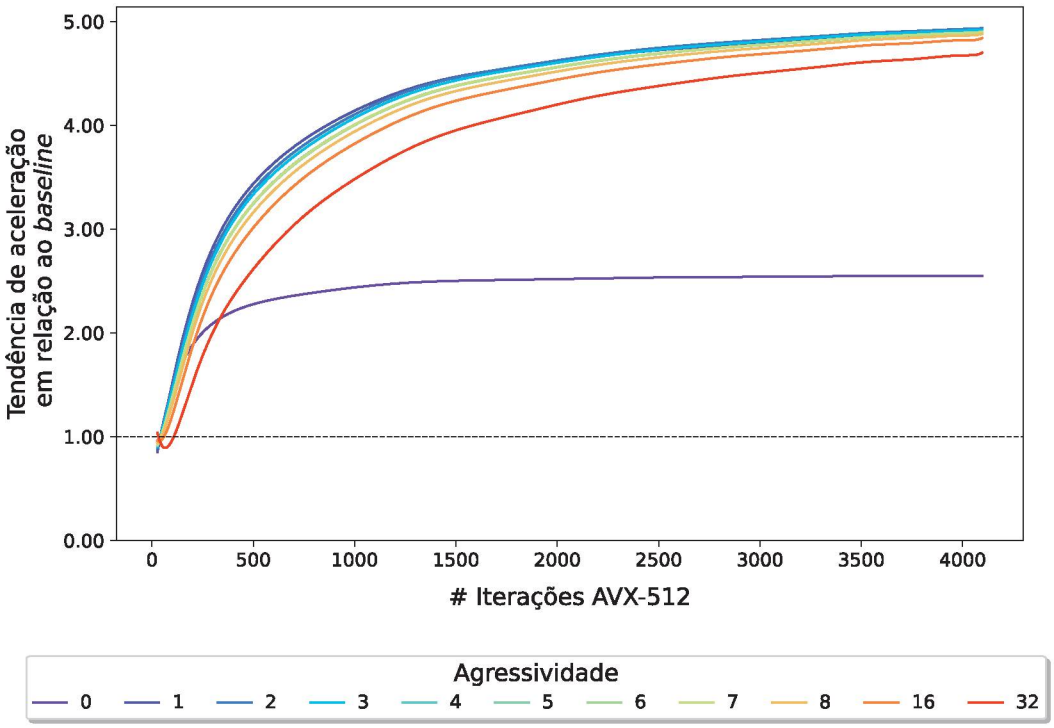


Figura A.38: Tendência de aceleração do algoritmo de cópia de memória, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-1024 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU (*baseline*). Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Adotamos como maior teste um laço de 4096 iterações, visto que é o suficiente para a estabilização das curvas de crescimento dos ganhos de cada agressividade avaliada. Fonte: Autor.

cada versão em relação ao *baseline*, um sistema com a largura de todos os *buffers* e de todos os estágios do *pipeline* quatro vezes o tamanho original. Tanto nosso mecanismo quanto as *intrinsics* também são executados considerando esse sistema expandido. Os diferentes gráficos nessa figura representam diferentes tamanhos de instrução VIMA utilizada para as conversões e *intrinsics*, o que é indicado ao lado esquerdo de cada gráfico. Consideramos uma operação sobre vetores de 8 MB, da mesma forma que em experimentos anteriores.

Como esperado, a expansão das estruturas do *pipeline* permitiu a obtenção de maiores ganhos de desempenho pelo SAPIVe, chegando a 99%, 98% e 96% da aceleração obtida por *intrinsics*, considerando instruções VIMA de 256 B, 512 B e 1024 B, respectivamente. Apesar dos três tipos de conversão obterem ganhos, a conversão para VIMA-1024 B apresenta degradações em agressividades maiores que 7. Possivelmente, nesse caso, instruções VIMA adicionais interferem na capacidade do ROB de conter conversões, com isso reduzindo o paralelismo nas validações da CPU. Apesar disso, comprovar essa teoria exige estudos adicionais não englobados nesta dissertação.

Vecsum - AVX-256

As Figuras A.51, A.52 e A.53 apresentam a aceleração obtida pelo SAPIVe em relação ao novo sistema, variando a agressividade, durante a conversão do Código 5.2, compilado em AVX-256, para instruções VIMA-256 B, VIMA-512 B e VIMA-1024 B, respectivamente.

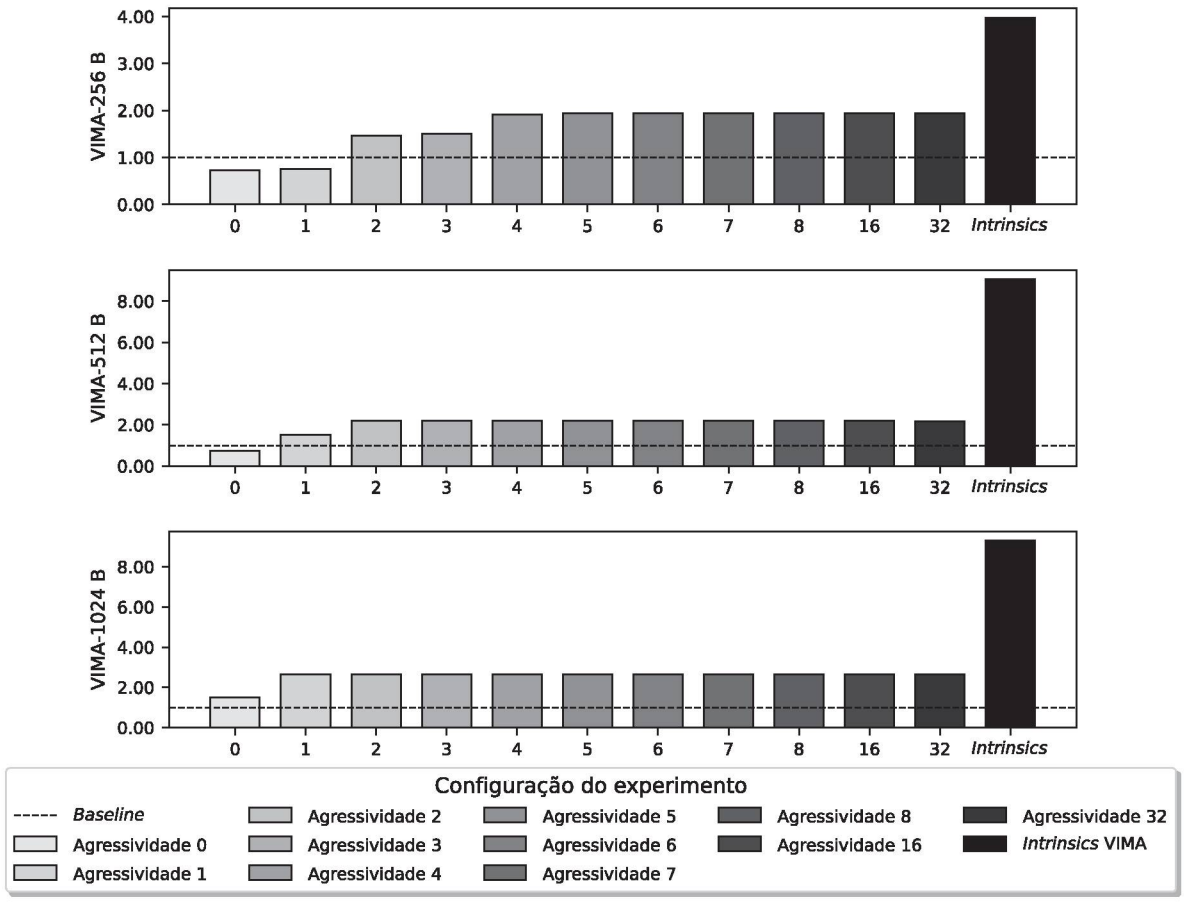


Figura A.39: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1 e compilado com instruções AVX-256, obtida pelo SAPIVe e por uma implementação manual, utilizando rotinas *intrinsic*, em relação a uma execução convencional em CPU (*baseline*). Para nosso mecanismo, são apresentados resultados com diversas agressividades, em conversões de AVX-256 para VIMA. A implementação manual utiliza instruções VIMA do mesmo tamanho das geradas por nosso mecanismo. Cada gráfico apresenta os resultados de aceleração obtidos utilizando um diferente tamanho de instrução VIMA, sendo o mais acima correspondente ao uso de instruções VIMA-256 B, o central ao uso de instruções VIMA-512 B e o mais abaixo correspondendo a instruções VIMA-1024 B. O algoritmo avaliado processa vetores de 8 MiB, tamanho suficiente para os ganhos em aceleração da implementação manual e de nosso mecanismo estabilizarem. Na horizontal, cada gráfico apresenta as diversas variações de nosso mecanismo e o uso de *intrinsics*. Já a vertical de cada gráfico contém a aceleração obtida sobre o *baseline* por cada experimento. Fonte: Autor.

Podemos notar que o maior tamanho das estruturas levou a maiores ganhos de desempenho para todas as versões avaliadas. Esses ganhos praticamente dobraram em relação aos obtidos no sistema original. Além disso, com um maior ROB foi possível converter **vecsum** compilado com AVX-256 para VIMA-1024 B, levando a grandes ganhos de desempenho.

A Figura A.54 apresenta uma comparação dos novos ganhos do SAPIVe em relação ao uso de rotinas *intrinsic*. O eixo horizontal de cada gráfico indica a agressividade utilizada pelo SAPIVe e a implementação utilizando *intrinsics*. No eixo vertical temos os ganhos obtidos por cada versão em relação ao *baseline*, um sistema com a largura de todos os *buffers* e de todos os estágios do *pipeline* quatro vezes o tamanho original. Tanto nosso mecanismo quanto as

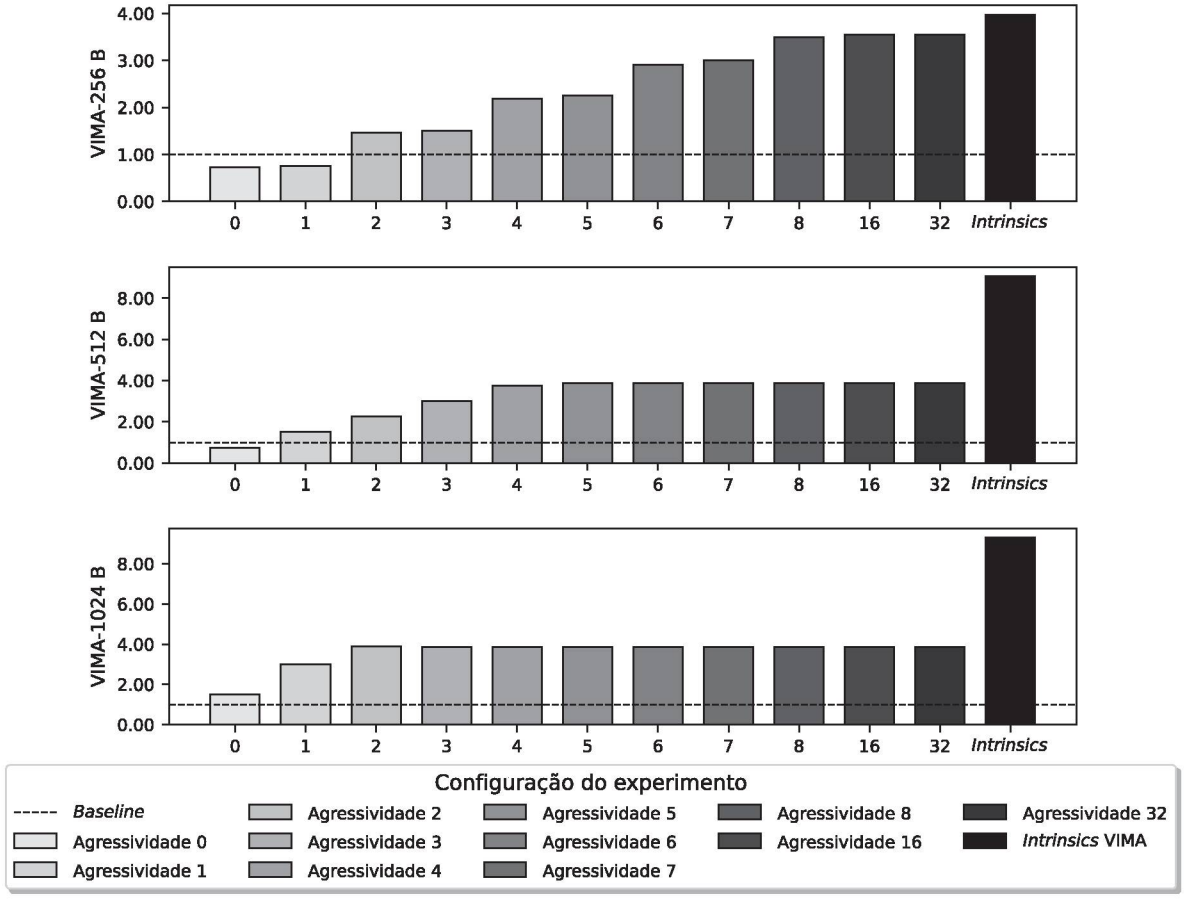


Figura A.40: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1 e compilado com instruções AVX-512, obtida pelo SAPIVe e por uma implementação manual, utilizando rotinas *intrinsic*, em relação a uma execução convencional em CPU (*baseline*). Para nosso mecanismo, são apresentados resultados utilizando diversas agressividades, em conversões de AVX-512 para VIMA. A implementação manual utiliza instruções VIMA do mesmo tamanho das geradas por nosso mecanismo. Cada gráfico apresenta os resultados de aceleração obtidos utilizando um diferente tamanho de VIMA, sendo o mais acima correspondente ao uso de instruções VIMA-256 B, o central ao uso de instruções VIMA-512 B e o mais abaixo correspondendo a instruções VIMA-1024 B. O algoritmo avaliado processa vetores de 8 MiB, tamanho suficiente para os ganhos em aceleração da implementação manual e de nosso mecanismo estabilizarem. Na horizontal, cada gráfico apresenta as diversas variações de nosso mecanismo e o uso de *intrinsics*. Já a vertical de cada gráfico contém a aceleração obtida sobre o *baseline* por cada experimento. Fonte: Autor.

intrinsics também são executados considerando esse sistema expandido. Os diferentes gráficos nessa figura representam diferentes tamanhos de instrução VIMA utilizada para as conversões e *intrinsics*, o que é indicado ao lado esquerdo de cada gráfico. Consideramos uma operação sobre vetores de 8 MB, da mesma forma que em experimentos anteriores.

Como esperado, a expansão das estruturas do *pipeline* permitiu a obtenção de maiores ganhos de desempenho pelo SAPIVe, chegando a 99%, 63% e 64% da aceleração obtida por *intrinsics*, considerando instruções VIMA de 256 B, 512 B e 1024 B, respectivamente. Da mesma forma que para a cópia de memória, conforme o tamanho das instruções VIMA aumenta, uma parte maior do ROB é ocupada por cada conversão, reduzindo o paralelismo nas conversões

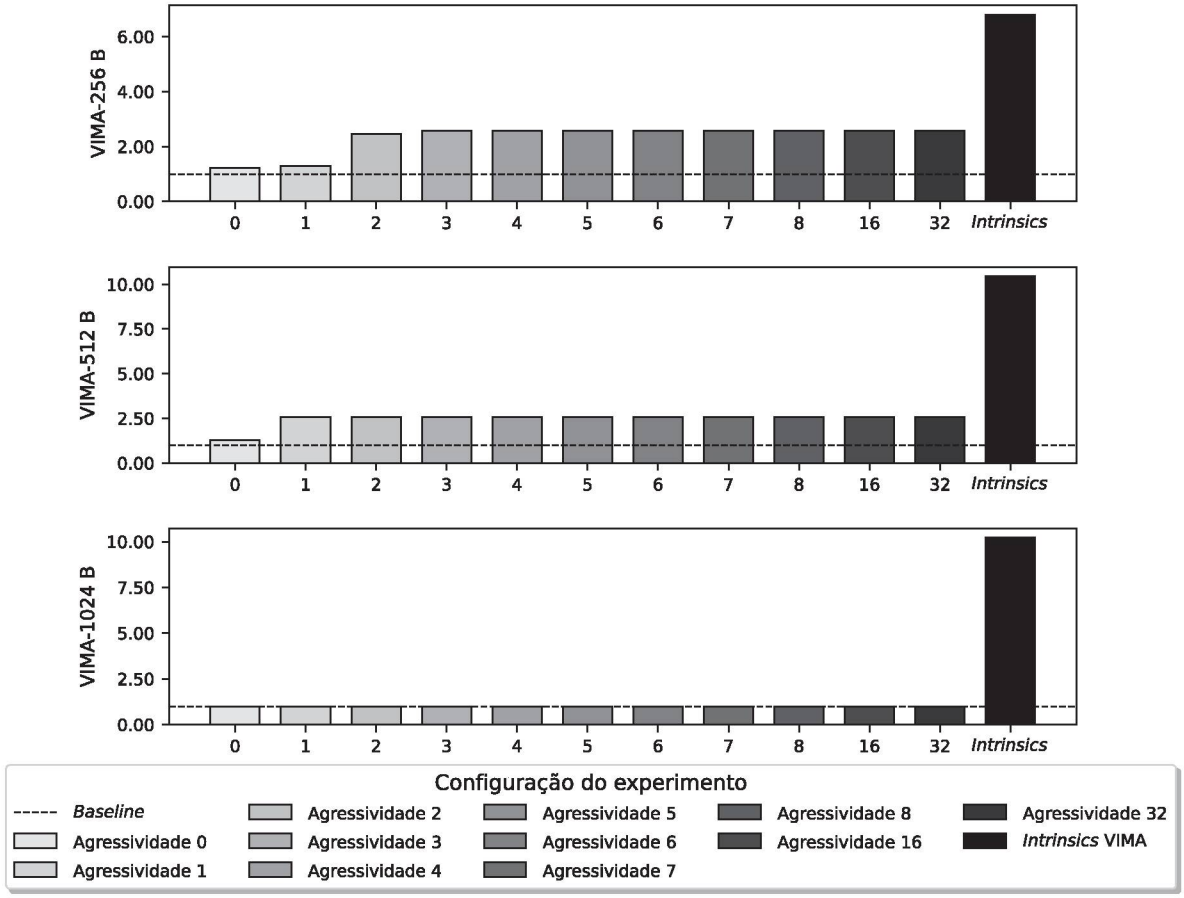


Figura A.41: Aceleração do algoritmo de soma vetorial, apresentado no Código 5.2 e compilado com instruções AVX-256, obtida pelo SAPIVe e por uma implementação manual, utilizando rotinas *intrinsic*, em relação a uma execução convencional em CPU (*baseline*). Para nosso mecanismo, são apresentados resultados utilizando diversas agressividades, em conversões de AVX-256 para VIMA. A implementação manual utiliza instruções VIMA do mesmo tamanho das geradas por nosso mecanismo. Cada gráfico apresenta os resultados de aceleração obtidos utilizando um diferente tamanho de VIMA, sendo o mais acima correspondente ao uso de instruções VIMA-256 B, o central ao uso de instruções VIMA-512 B e o mais abaixo correspondendo a instruções VIMA-1024 B. O algoritmo avaliado processa vetores de 8 MiB, tamanho suficiente para os ganhos em aceleração da implementação manual e de nosso mecanismo estabilizarem. Na horizontal, cada gráfico apresenta as diversas variações de nosso mecanismo e o uso de *intrinsics*. Já a vertical de cada gráfico contém a aceleração obtida sobre o *baseline* por cada experimento. Fonte: Autor.

verificadas pela CPU e reduzindo os ganhos em relação ao uso de *intrinsics*. Porém, no **vecsum**, o aumento no tamanho das instruções VIMA não leva a um redução de ganhos com agressividades maiores, o que pode estar relacionado à maior pressão de memória por cada conversão, visto que dois vetores são lidos ao invés de um. Assim, mesmo com um número menor de conversões podendo ser verificado, pela grande quantidade de adiantamentos, essa redução é ocultada pelo maior gargalo na memória. Porém, mais estudos são necessários para o esclarecimento dessa variação.

Vecsum - AVX-512

As Figuras A.55, A.56 e A.57 apresentam a aceleração obtida pelo SAPIVe em relação ao novo

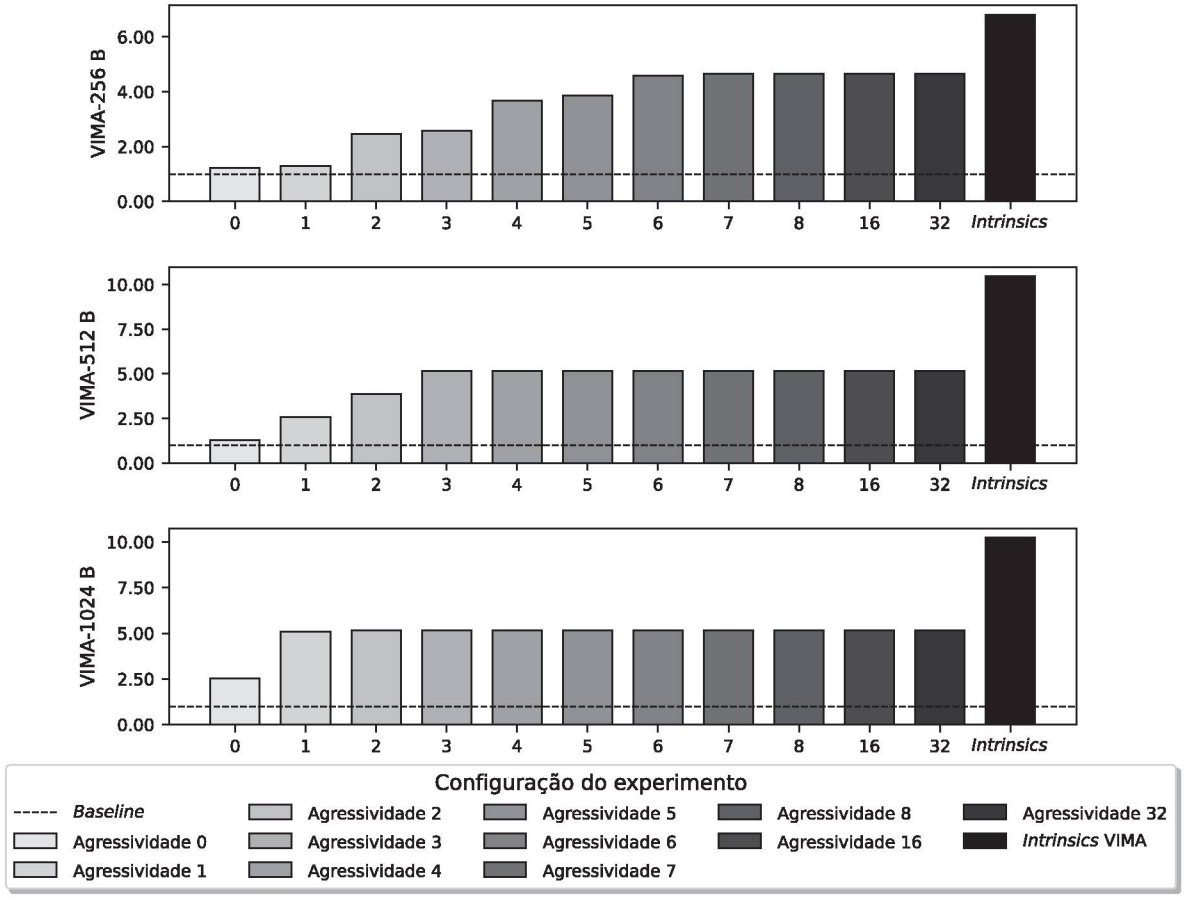


Figura A.42: Aceleração do algoritmo de soma vetorial, apresentado no Código 5.2 e compilado com instruções AVX-512, obtida pelo SAPIVe e por uma implementação manual, utilizando rotinas *intrinsic*, em relação a uma execução convencional em CPU (*baseline*). Para nosso mecanismo, são apresentados resultados utilizando diversas agressividades, em conversões de AVX-512 para VIMA. A implementação manual utiliza instruções VIMA do mesmo tamanho das geradas por nosso mecanismo. Cada gráfico apresenta os resultados de aceleração obtidos utilizando um diferente tamanho de VIMA, sendo o mais acima correspondente ao uso de instruções VIMA-256 B, o central ao uso de instruções VIMA-512 B e o mais abaixo correspondendo a instruções VIMA-1024 B. O algoritmo avaliado processa vetores de 8 MiB, tamanho suficiente para os ganhos em aceleração da implementação manual e de nosso mecanismo estabilizarem. Na horizontal, cada gráfico apresenta as diversas variações de nosso mecanismo e o uso de *intrinsics*. Já a vertical de cada gráfico contém a aceleração obtida sobre o *baseline* por cada experimento. Fonte: Autor.

sistema, variando a agressividade, durante a conversão do Código 5.2, compilado em AVX-512, para instruções VIMA-256 B, VIMA-512 B e VIMA-1024 B, respectivamente. Podemos notar que, da mesma forma que nos demais experimentos, a maior largura das estruturas permite que os ganhos apresentados continuem crescendo com um número maior adiantamentos.

A Figura A.58 apresenta uma comparação dos novos ganhos do SAPIVe em relação ao uso de rotinas *intrinsic*. O eixo horizontal de cada gráfico indica a agressividade utilizada pelo SAPIVe e a implementação utilizando *intrinsics*. No eixo vertical temos os ganhos obtidos por cada versão em relação ao *baseline*, um sistema com a largura de todos os *buffers* e de todos os estágios do *pipeline* quatro vezes o tamanho original. Tanto nosso mecanismo quanto as

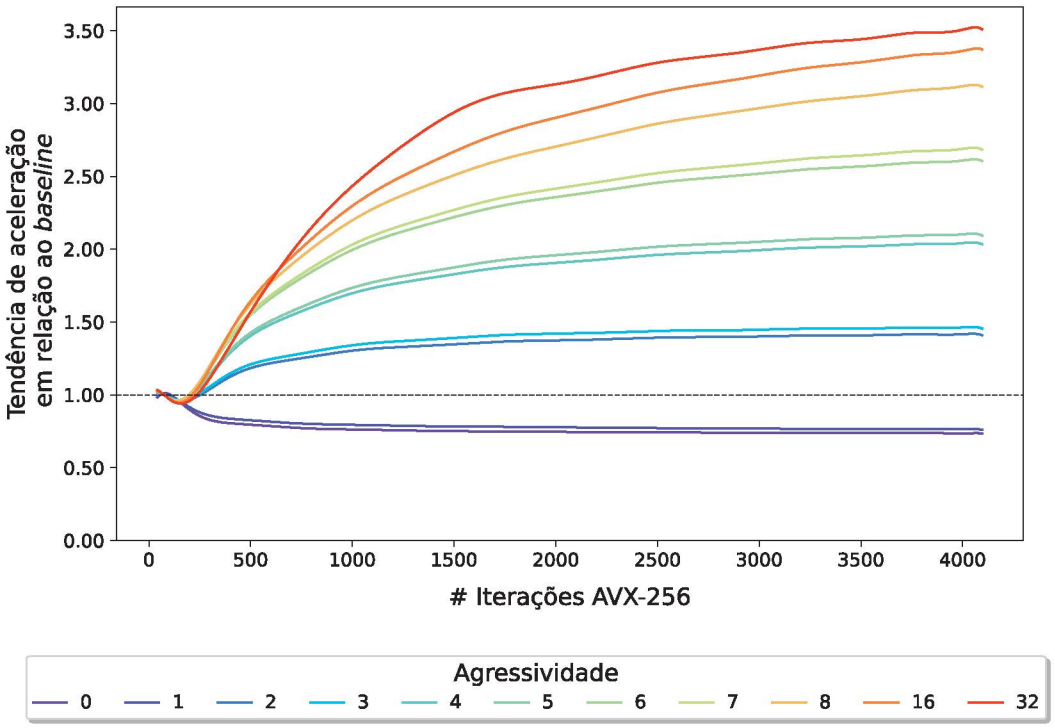


Figura A.43: Tendência de aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida pelo SAPIVe em conversões de AVX-256 para VIMA-256 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU(*baseline*). Tanto o mecanismo quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Fonte: Autor.

intrinsics também são executados considerando esse sistema expandido. Os diferentes gráficos nessa figura representam diferentes tamanhos de instrução VIMA utilizada para as conversões e *intrinsics*, o que é indicado ao lado esquerdo de cada gráfico. Consideramos uma operação sobre vetores de 8 MB, da mesma forma que em experimentos anteriores.

Como esperado, a expansão das estruturas do *pipeline* permitiu a obtenção de maiores ganhos de desempenho pelo SAPIVe, chegando a 99% os ganhos obtidos por *intrinsics*, considerando instruções VIMA de 256 B e 512 B, e 101% considerando instruções AVX-1024 B, respectivamente. Essa aceleração superior à obtida pelo uso de *intrinsics* pode ter ocorrido pelo envio conjunto de diversas instruções *vima* durante o primeiro adiantamento. Porém, estudos mais aprofundados são necessários para compreender esse fenômeno.

Esses resultados indicam o potencial de nosso mecanismo (SAPIVe) em sistemas futuros, aproximando-se cada vez mais, e podendo até mesmo superar, implementações manuais utilizando *intrinsics* VIMA.

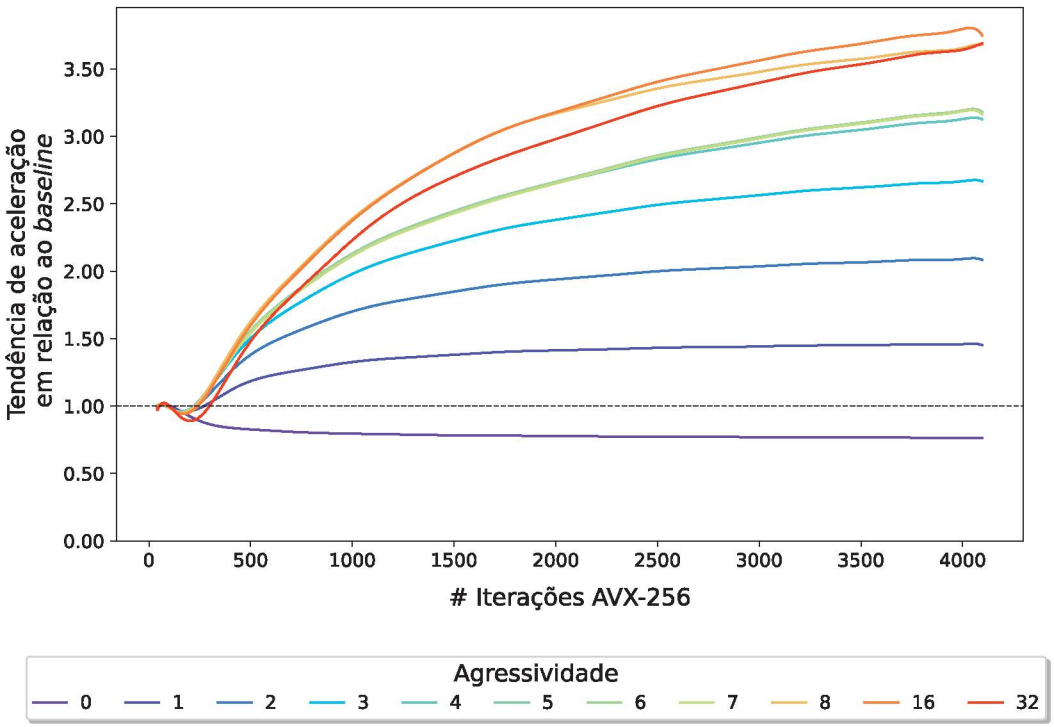


Figura A.44: Tendência de aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida pelo SAPIVe em conversões de AVX-256 para VIMA-512 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU(*baseline*). Tanto o mecanismo quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Fonte: Autor.

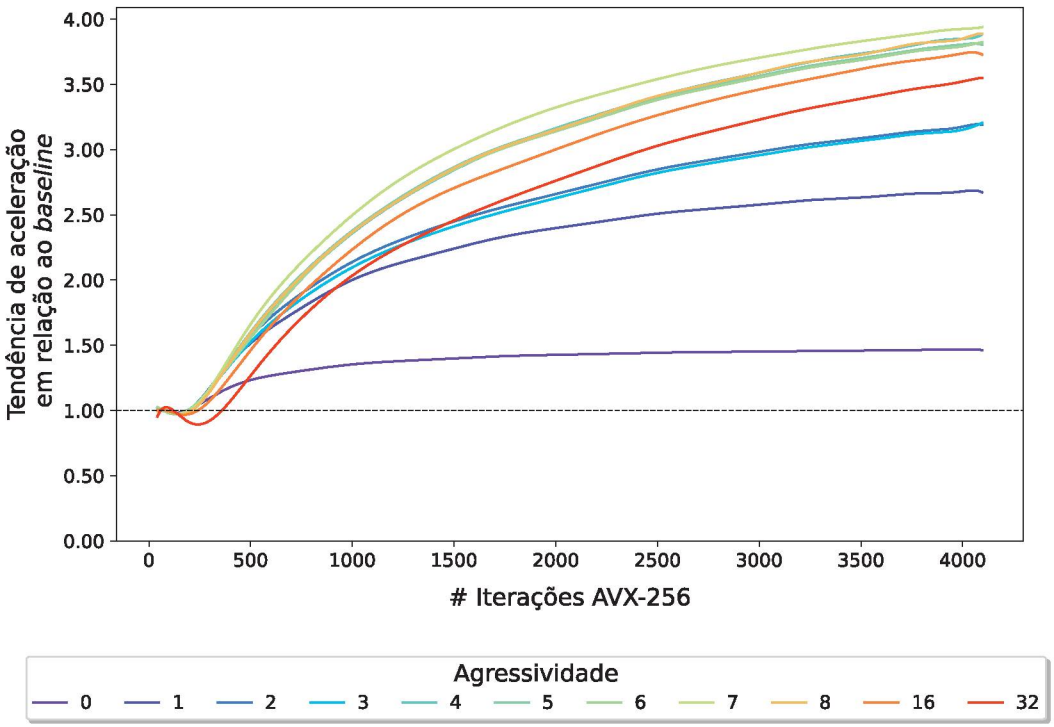


Figura A.45: Tendência de aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida pelo SAPIVe em conversões de AVX-256 para VIMA-1024 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU(*baseline*). Tanto o mecanismo quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Fonte: Autor.

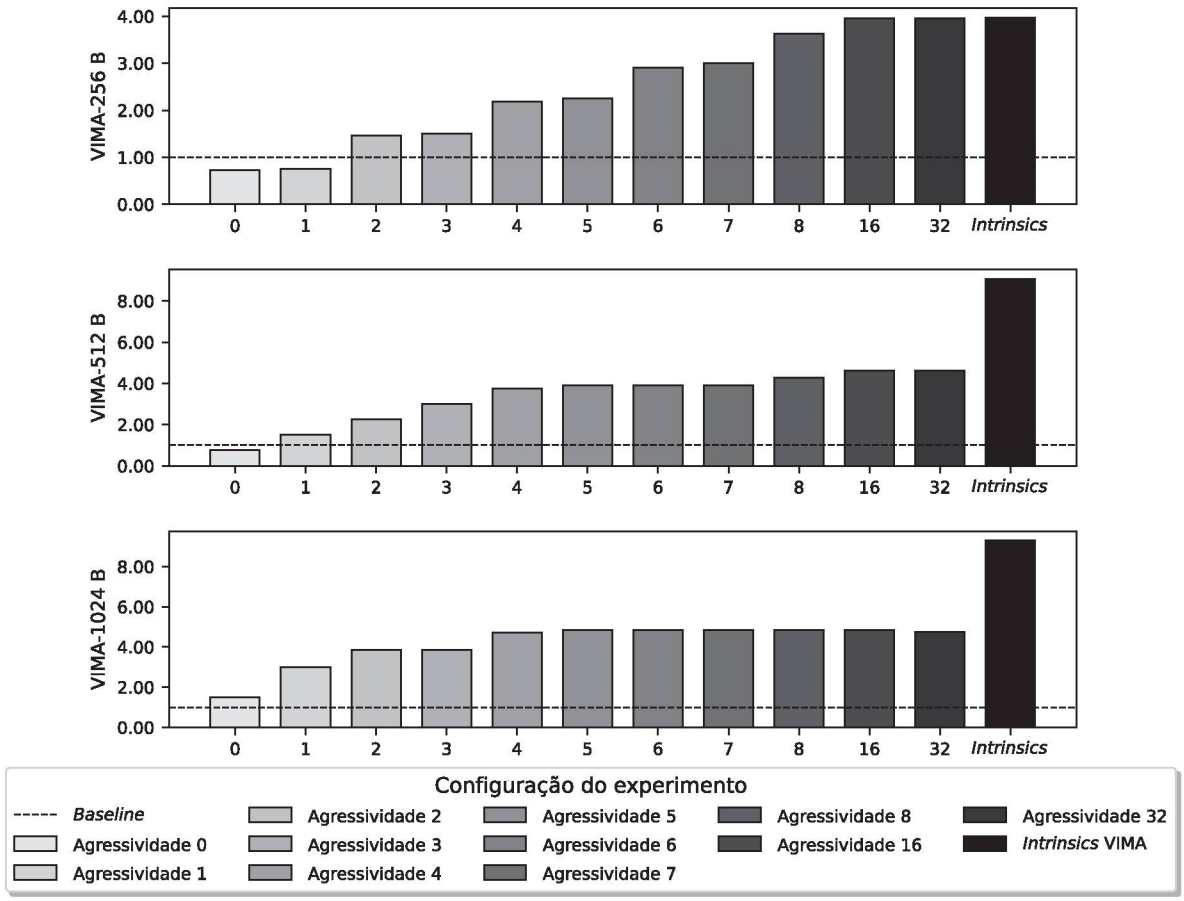


Figura A.46: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1 e compilado com instruções AVX-256, obtida pelo SAPIVe e por uma implementação manual, utilizando rotinas *intrinsic*, em relação a uma execução convencional em CPU (*baseline*). Tanto o mecanismo e as *intrinsics*, quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Para nosso mecanismo, são apresentados resultados utilizando diversas agressividades, em conversões de AVX-256 para VIMA. A implementação manual utiliza instruções VIMA do mesmo tamanho das geradas por nosso mecanismo. Cada gráfico apresenta os resultados de aceleração obtidos utilizando um diferente tamanho de VIMA, sendo o mais acima correspondente ao uso de instruções VIMA-256 B, o central ao uso de instruções VIMA-512 B e o mais abaixo correspondendo a instruções VIMA-1024 B. O algoritmo avaliado processa vetores de 8 MiB, assim como nos experimentos com o sistema original. Fonte: Autor.

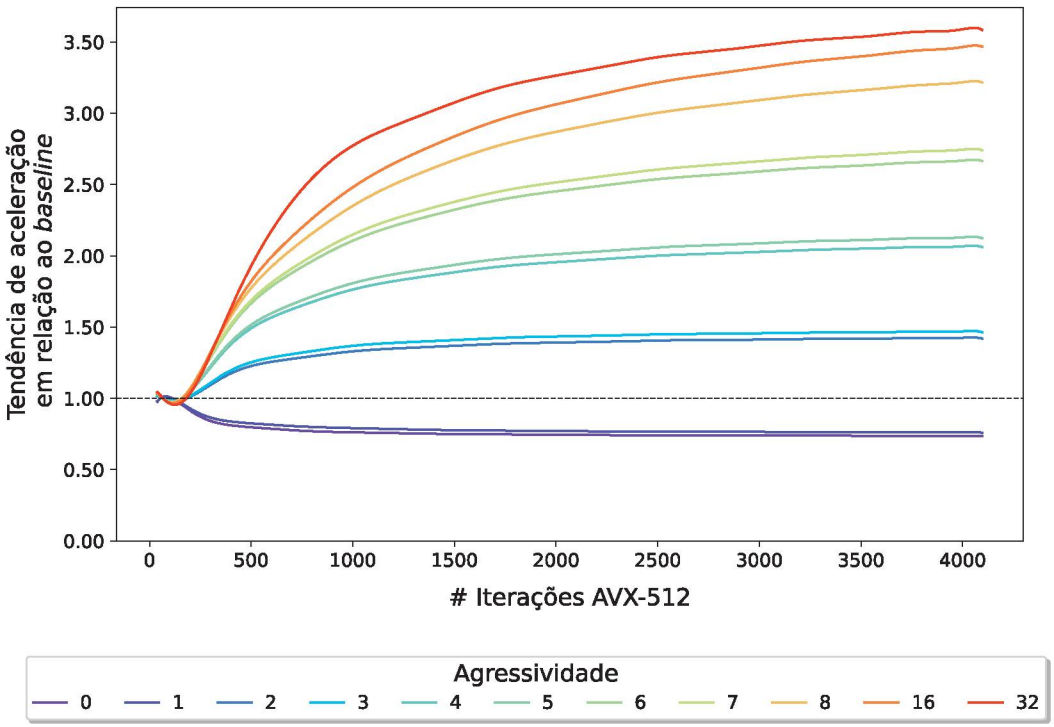


Figura A.47: Tendência de aceleração do algoritmo de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-256 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU(*baseline*). Tanto o mecanismo quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Fonte: Autor.

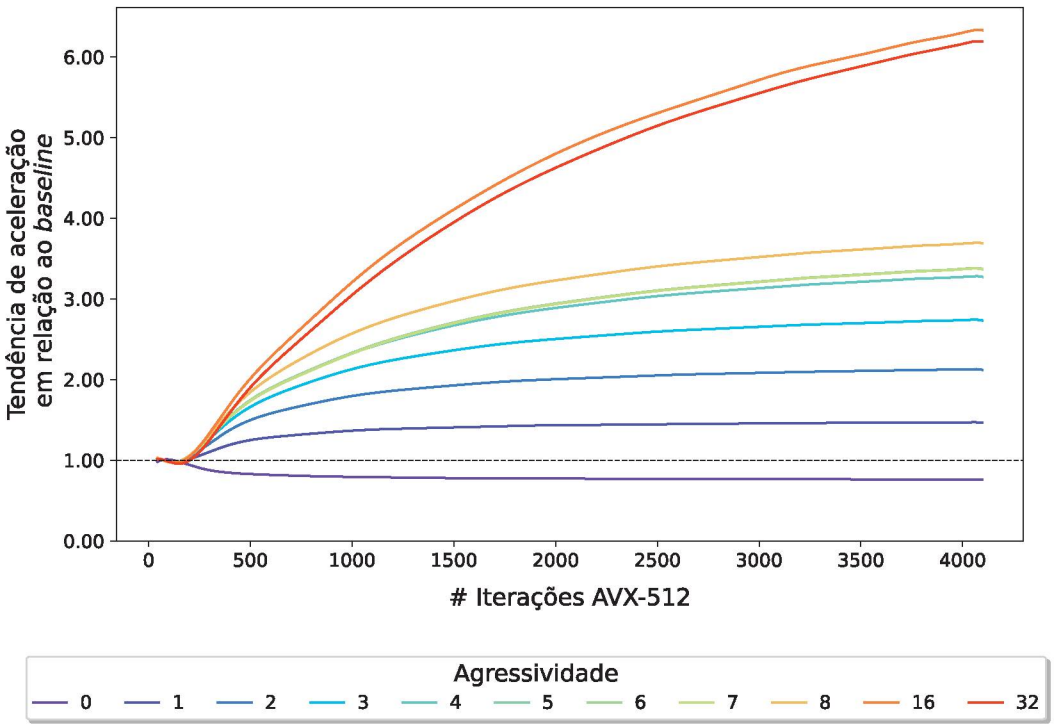


Figura A.48: Tendência de aceleração do código de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-512 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU(*baseline*). Tanto o mecanismo quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Fonte: Autor.

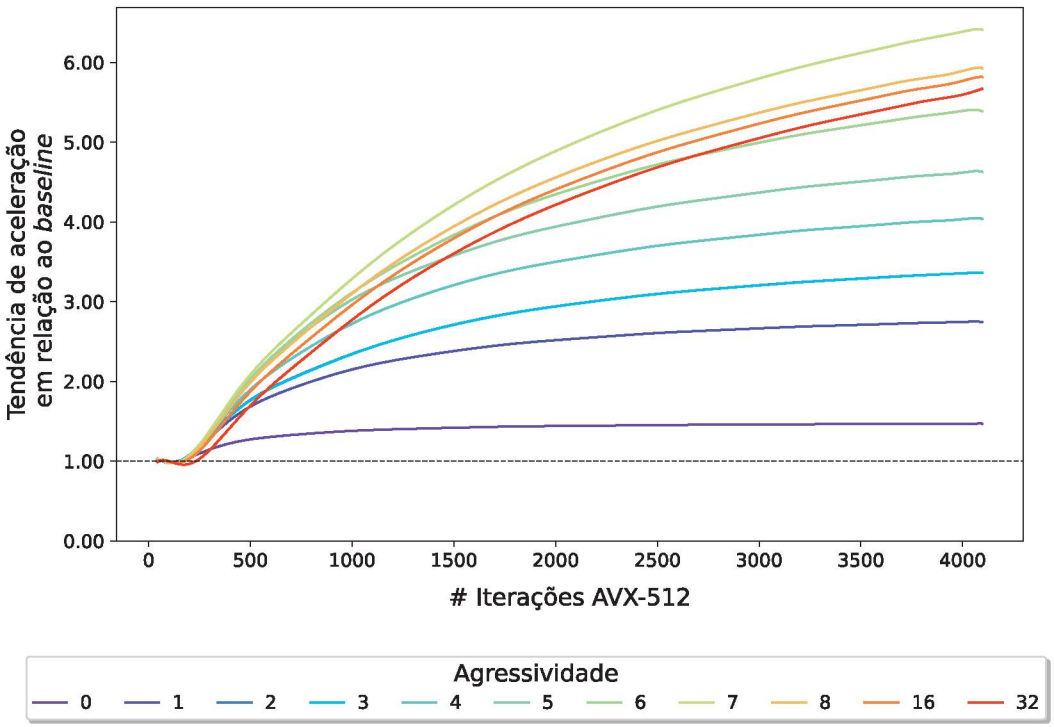


Figura A.49: Tendência de aceleração do código de cópia de memória, apresentado no Código 5.1, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-1024 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU(*baseline*). Tanto o mecanismo quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Fonte: Autor.

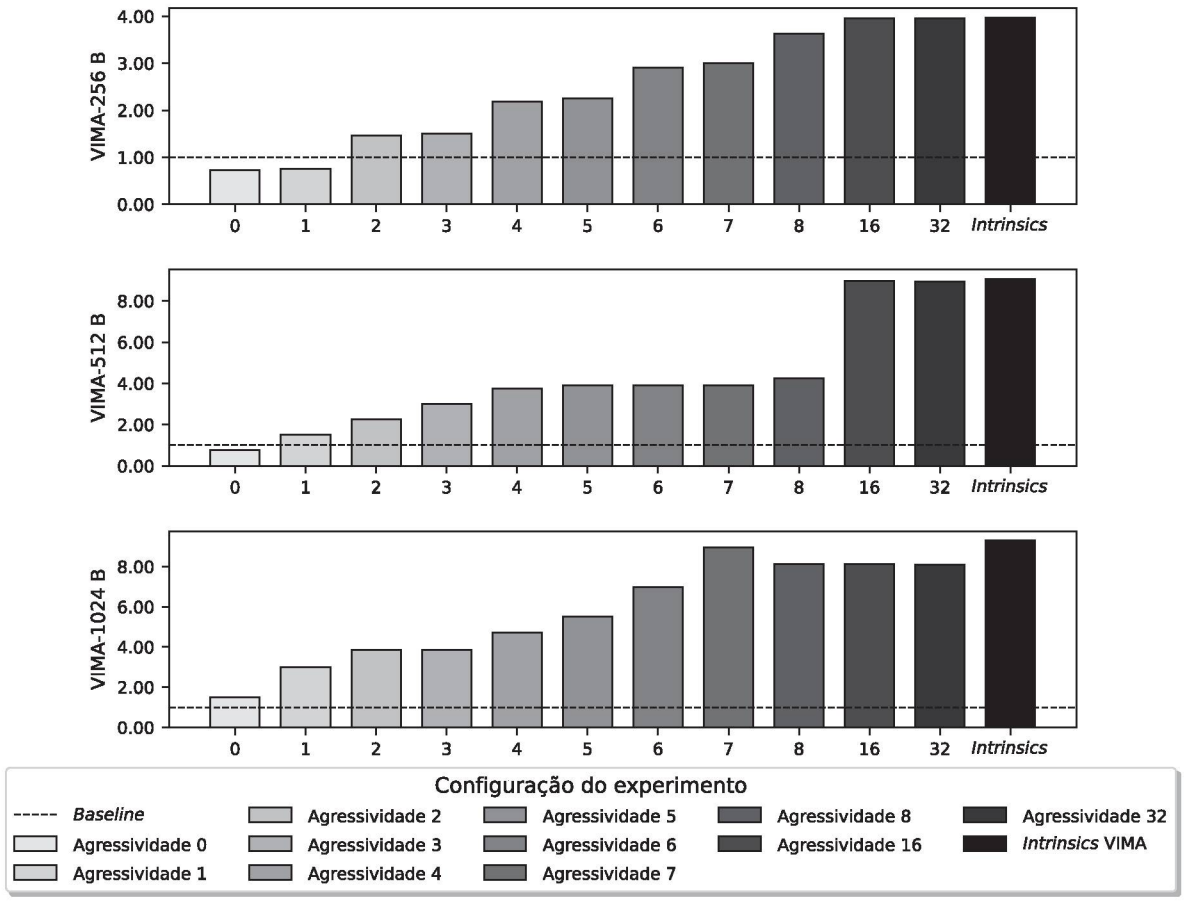


Figura A.50: Aceleração do algoritmo de cópia de memória, apresentado no Código 5.1 e compilado com instruções AVX-512, obtida por nosso mecanismo e por uma implementação manual, utilizando rotinas *intrinsic*, em relação a uma execução convencional em CPU (*baseline*). Tanto o mecanismo e as *intrinsics*, quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Para nosso mecanismo, são apresentados resultados utilizando diversas agressividades em conversões de AVX-512 para VIMA. A implementação manual utiliza instruções VIMA do mesmo tamanho das geradas por nosso mecanismo. Cada gráfico apresenta os resultados de aceleração obtidos utilizando um diferente tamanho de VIMA, sendo o mais acima correspondente ao uso de instruções VIMA-256 B, o central ao uso de instruções VIMA-512 B e o mais abaixo correspondendo a instruções VIMA-1024 B. O algoritmo avaliado processa vetores de 8 MiB, assim como nos experimentos com o sistema original. Fonte: Autor.

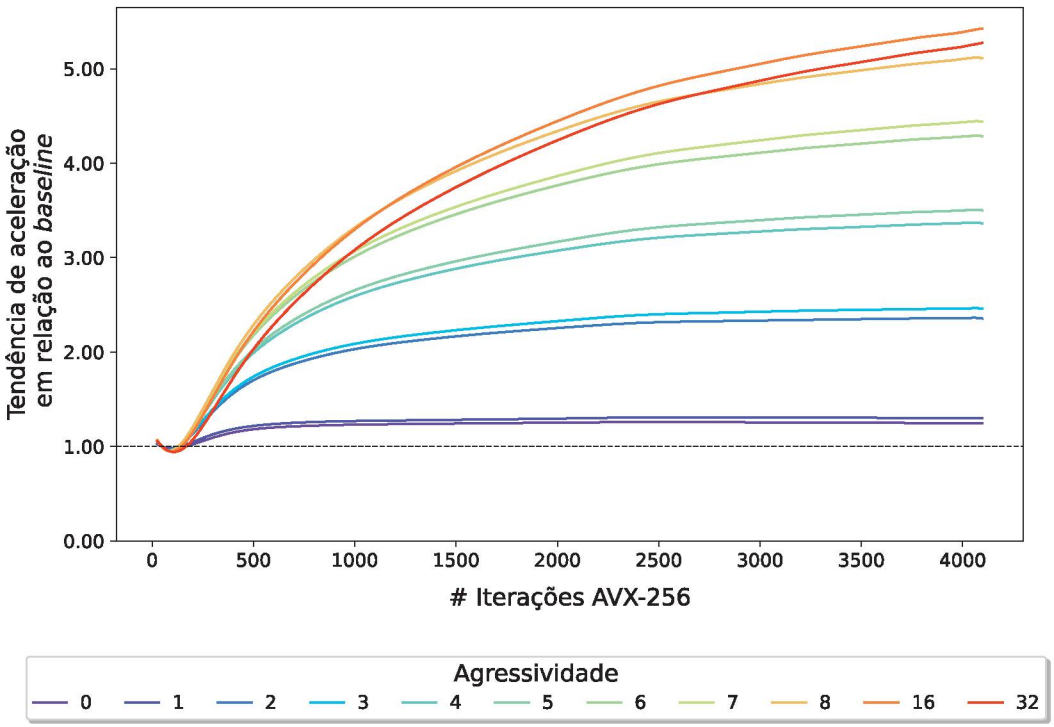


Figura A.51: Tendência de aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-256 para VIMA-256 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU(*baseline*). Tanto o mecanismo quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Fonte: Autor.

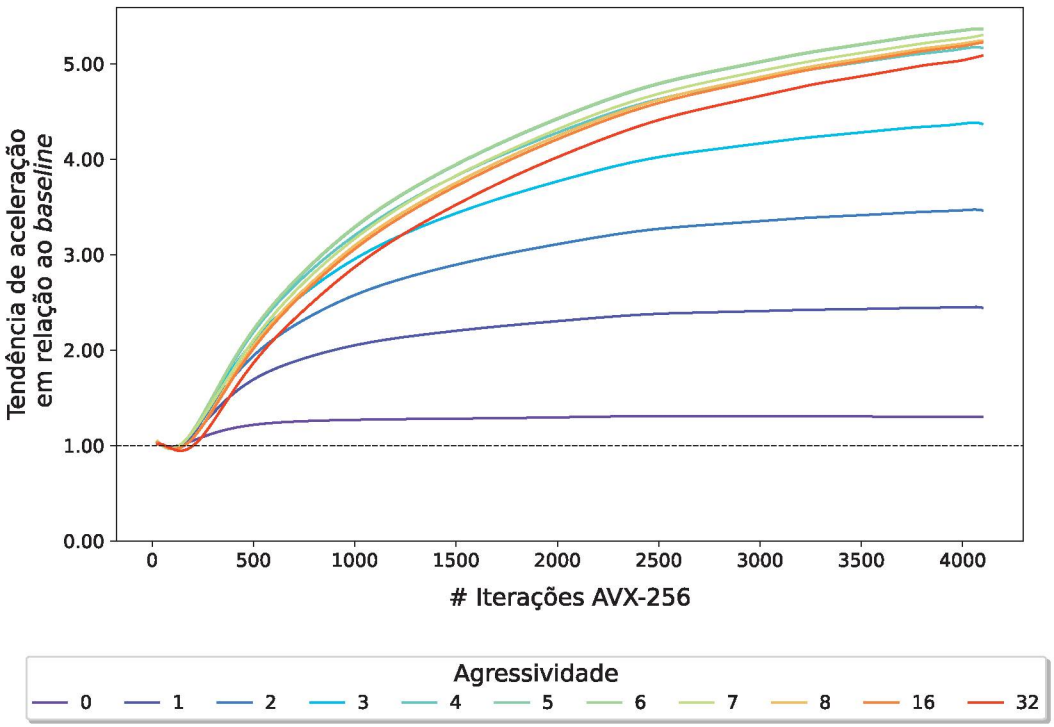


Figura A.52: Tendência de aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-256 para VIMA-512 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU(*baseline*). Tanto o mecanismo quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Fonte: Autor.

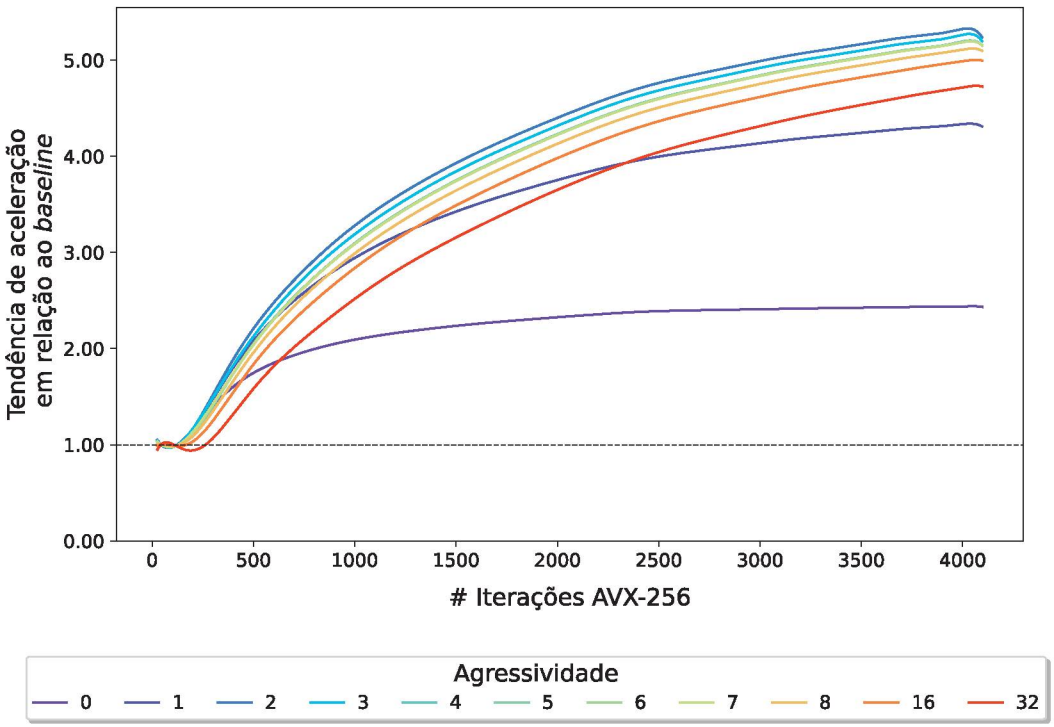


Figura A.53: Tendência de aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-256 para VIMA-1024 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU(*baseline*). Tanto o mecanismo quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Fonte: Autor.

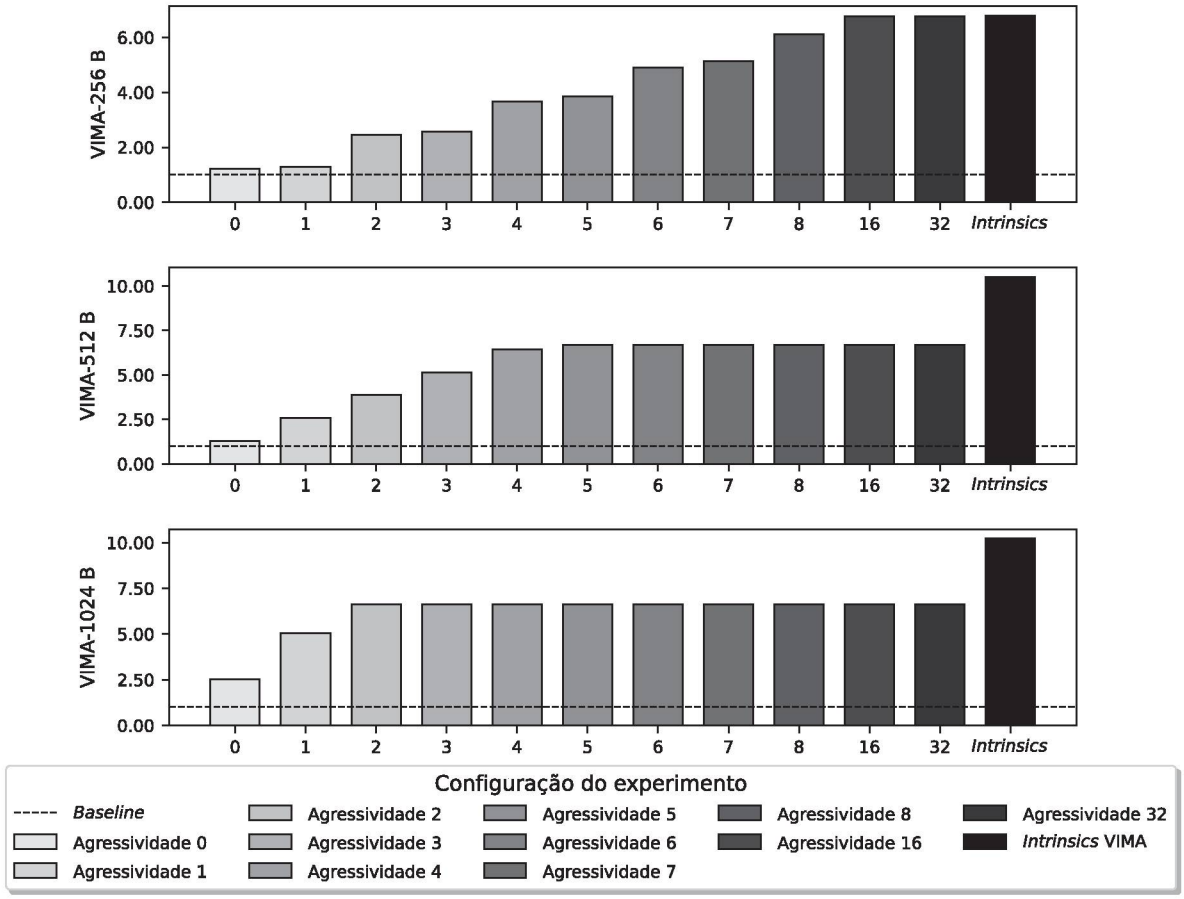


Figura A.54: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2 e compilado com instruções AVX-256, obtida pelo SAPIVe e por uma implementação manual, utilizando rotinas *intrinsic*, em relação a uma execução convencional em CPU (*baseline*). Tanto o mecanismo e as *intrinsics*, quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Para nosso mecanismo, são apresentados resultados utilizando diversas agressividades, em conversões de AVX-256 para VIMA. A implementação manual utiliza instruções VIMA do mesmo tamanho das geradas por nosso mecanismo. Cada gráfico apresenta os resultados de aceleração obtidos utilizando um diferente tamanho de VIMA, sendo o mais acima correspondente ao uso de instruções VIMA-256 B, o central ao uso de instruções VIMA-512 B e o mais abaixo correspondendo a instruções VIMA-1024 B. O algoritmo avaliado processa vetores de 8 MiB, assim como nos experimentos com o sistema original. Fonte: Autor.

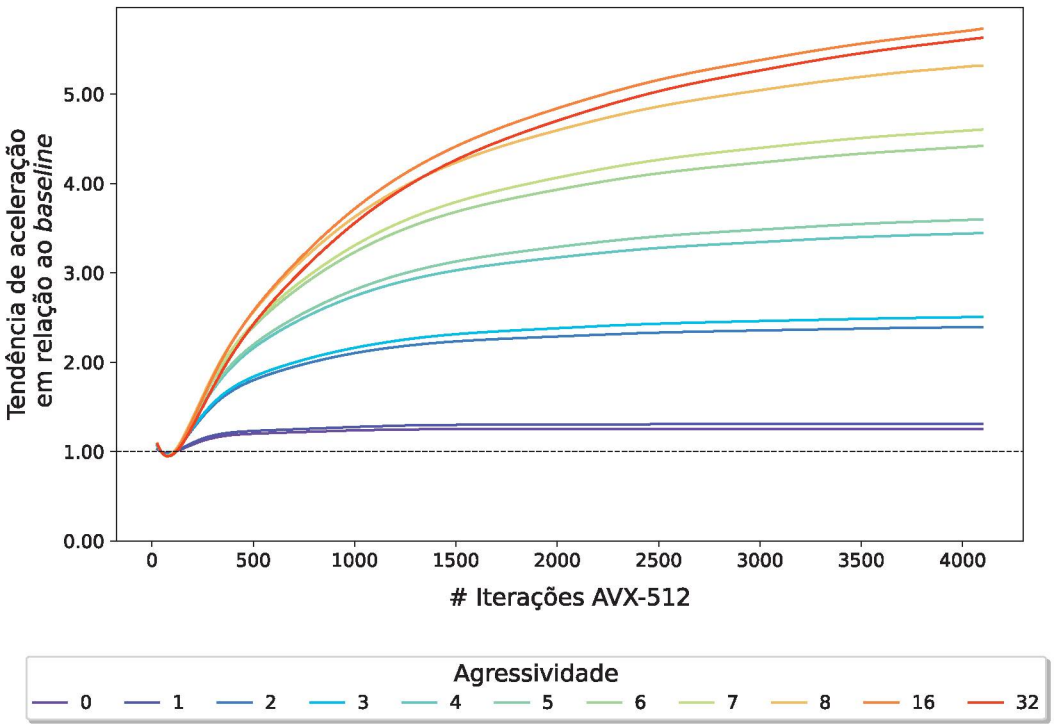


Figura A.55: Tendência de aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-256 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU(*baseline*). Tanto o mecanismo quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Fonte: Autor.

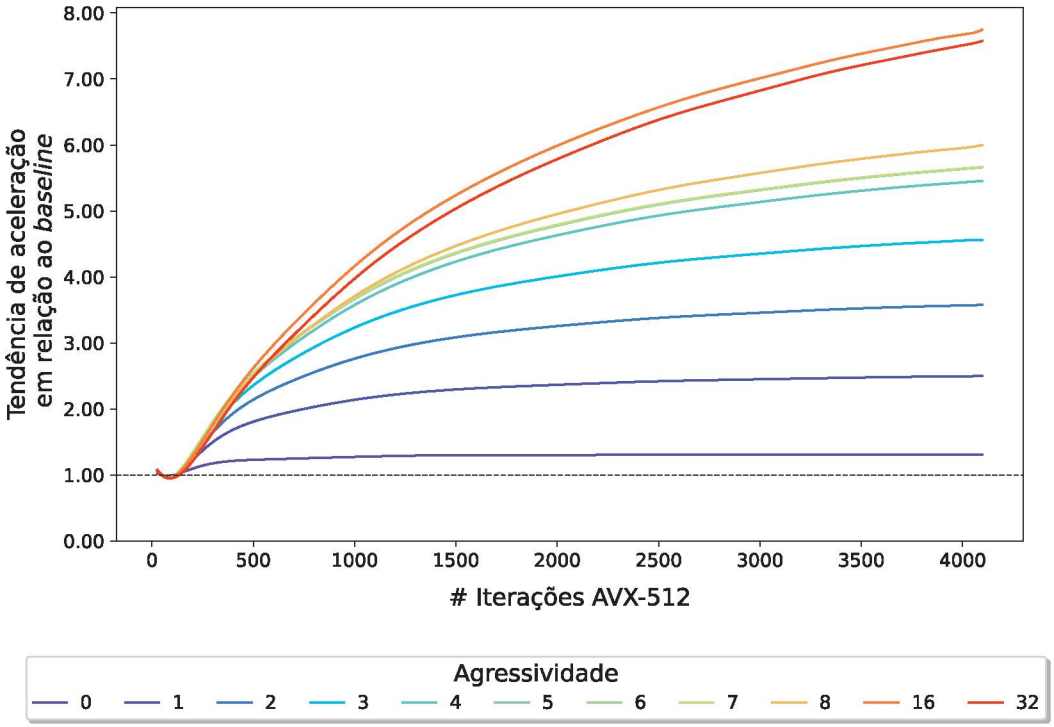


Figura A.56: Tendência de aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-512 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU(*baseline*). Tanto o mecanismo quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Fonte: Autor.

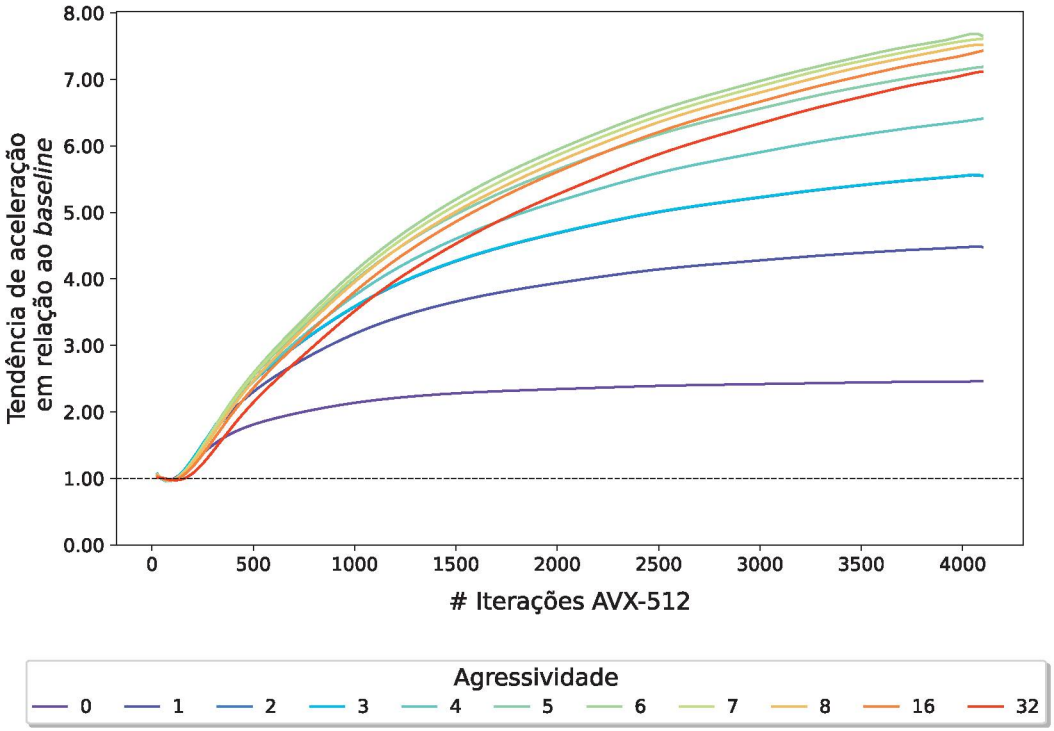


Figura A.57: Tendência de aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2, obtida por nosso mecanismo em conversões de AVX-512 para VIMA-1024 B, utilizando diversas agressividades, em relação a uma execução convencional em CPU(*baseline*). Tanto o mecanismo quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Cada agressividade foi avaliada para laços AVX com número de iterações crescente (horizontal). Fonte: Autor.

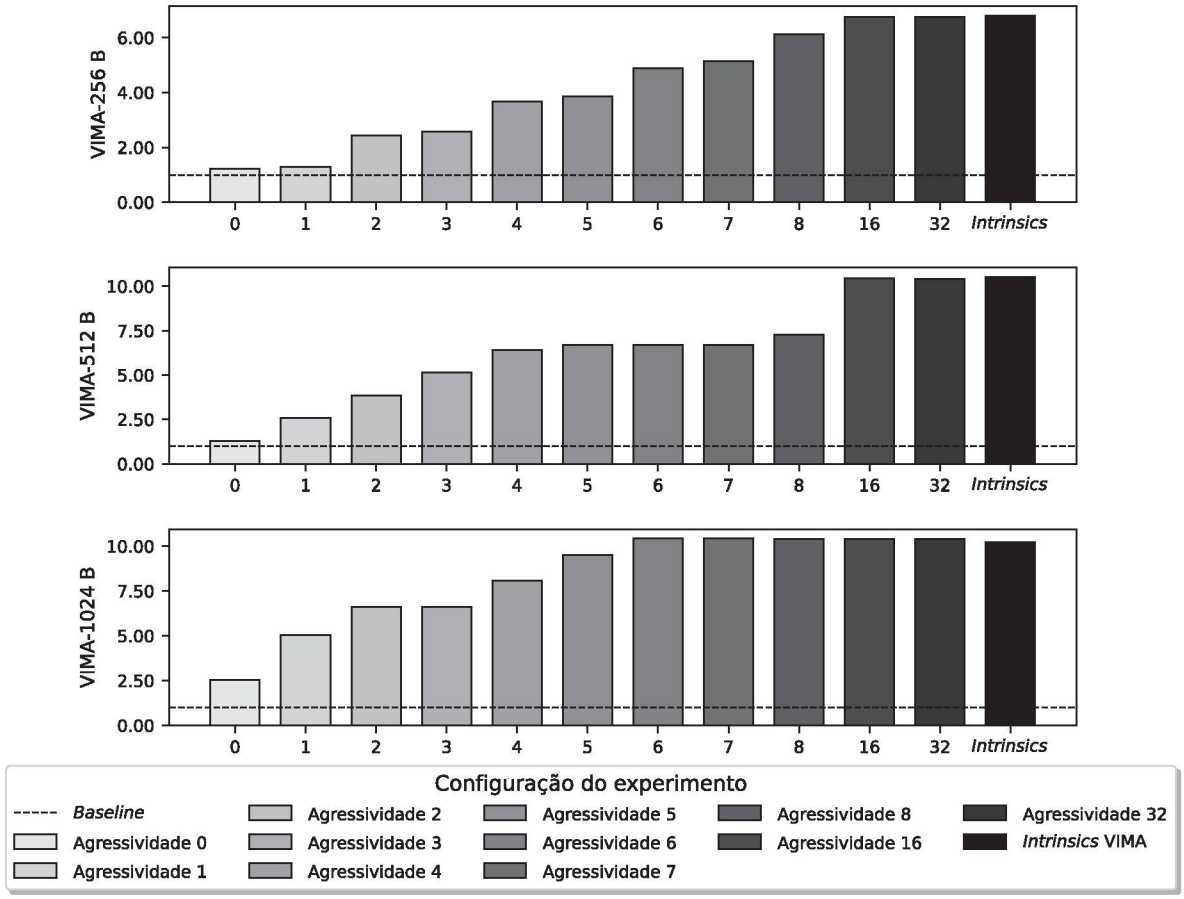


Figura A.58: Aceleração do algoritmo de soma entre vetores, apresentado no Código 5.2 e compilado com instruções AVX-512, obtida pelo SAPIVe e por uma implementação manual, utilizando rotinas *intrinsic*, em relação a uma execução convencional em CPU (*baseline*). Tanto o mecanismo e as *intrinsics*, quanto o *baseline* são avaliados em um sistema com *buffers* e estágios do *pipeline* quatro vezes mais largos. Para nosso mecanismo, são apresentados resultados utilizando diversas agressividades, em conversões de AVX-512 para VIMA. A implementação manual utiliza instruções VIMA do mesmo tamanho das geradas por nosso mecanismo. Cada gráfico apresenta os resultados de aceleração obtidos utilizando um diferente tamanho de VIMA, sendo o mais acima correspondente ao uso de instruções VIMA-256 B, o central ao uso de instruções VIMA-512 B e o mais abaixo correspondendo a instruções VIMA-1024 B. O algoritmo avaliado processa vetores de 8 MiB, assim como nos experimentos com o sistema original. Fonte: Autor.