UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"
FACULDADE DE CIÊNCIAS - CAMPUS BAURU
DEPARTAMENTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ANDRÉ LIBÓRIO DE BARROS FERRAZ

INVESTIGAÇÃO DE TÉCNICAS DE OTIMIZAÇÃO PARA ALGORITMOS DE APRENDIZADO DE MÁQUINA

BAURU
Julho/2022

ANDRÉ LIBÓRIO DE BARROS FERRAZ

INVESTIGAÇÃO DE TÉCNICAS DE OTIMIZAÇÃO PARA ALGORITMOS DE APRENDIZADO DE MÁQUINA

Trabalho de Conclusão de Curso do Curso de Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.
Orientador: Prof. Dr. João Paulo Papa
Coorientador: Prof. Dr. Alexandro José Baldassin

BAURU
Julho/2022

F381i

Ferraz, André Libório de Barros

Investigação de técnicas de otimização para algoritmos de aprendizado de máquina / André Libório de Barros Ferraz. -- Bauru, 2022

45 f. : il.

Trabalho de conclusão de curso (Bacharelado - Ciência da Computação) - Universidade Estadual Paulista (Unesp), Faculdade de Ciências, Bauru

Orientador: João Paulo Papa

Coorientador: Alexandro José Baldassin

1. Computação de alto desempenho. 2. Redes neurais. 3. Informática. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca da Faculdade de Ciências, Bauru. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

André Libório de Barros Ferraz

INVESTIGAÇÃO DE TÉCNICAS DE OTIMIZAÇÃO PARA ALGORITMOS DE APRENDIZADO DE MÁQUINA

Trabalho de Conclusão de Curso do Curso de Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.

Banca Examinadora

Prof. Dr. João Paulo Papa
Orientador
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Computação

Profª. Drª. Simone das Graças Domingues Prado
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Computação

Prof. Dr. Kelton Augusto Pontara da Costa
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Computação

Bauru, _____ de _____ de _____.

Dedico esta monografia a todos aqueles ao meu redor, que de qualquer forma contribuíram para chegar a este ponto. Em especial à minha família, amigos e professores, que sempre me apoiaram e me incentivaram a seguir meus sonhos.

Agradecimentos

Agradeço a minha família, em especial minha mãe Flávia, que sempre me apoiou e contribuiu para que meu dia a dia fosse o mais proveitoso apesar de todas as dificuldades e minha calopsita Luke, que sempre me faz companhia, em especial durante o trabalho nesta monografia.

Agradeço meus professores, que me ensinaram e deram oportunidades para meu crescimento, em especial a prof. Dra. Andréa Carla Gonçalves Vianna, que me incentivou a entrar no curso e, durante as disciplinas, melhorar muito minhas habilidades em programação, ao prof. Dr. João Paulo Papa, por todo o conhecimento e apoio dado no decorrer da graduação e por ter aceitado o pedido de orientação deste trabalho e ao prof. Dr. Alexandro José Baldassin por nestes últimos anos ter sido um grande orientador, e neste trabalho ter aceitado o pedido de co-orientação, por ter me ensinado muito sobre a vida acadêmica e me dado inúmeras oportunidades de aprendizado que me fizeram chegar a este ponto.

Gostaria também de agradecer a todos meus amigos e colegas que fizeram parte desse período da minha vida pelo os bons momentos que passamos e os que ainda virão.

Allons-y!

Resumo

Tendo em vista as inovações tecnológicas da última década, este trabalho busca, por meio de técnicas como vetorização utilizando AVX-512 e arcabouços computacionais para arquiteturas paralelas como o Galois, modificar algoritmos de aprendizado de máquina baseados em grafos, neste caso em particular, o OPF (*Optimum-Path Forest*) com a finalidade de melhorar o seu tempo de execução. Resultados apresentam ganhos significativos com o uso da tecnologia AVX-512, particularmente nas configurações com 1 *thread* de até 26,84% se comparado a versão com uso de AVX2 e 112,83% se comparado a versão não vetorizada. Quanto ao Galois, fora realizado um estudo inicial que avaliou o desempenho do MST (*Minimum Spanning Tree*) e os resultados preliminares apontam um *speedup* de até 6x com o *dataset* `Epinions`. No futuro, espera-se complementar a implementação do Galois para outros algoritmos de grafos baseados no OPF.

Palavras-chave: Computação de alto desempenho, Redes neurais, Informática.

Abstract

In view of the technological innovations of the last decade, this work seeks, through techniques such as vectorization using AVX-512 and computational frameworks for parallel architectures such as Galois, to modify graph-based machine learning algorithms, in this particular case, the OPF (Optimum-Path Forest) in order to improve its execution time. Results show significant gains with the use of AVX-512 technology, particularly in configurations with 1 thread up to 26.84% compared to the version using AVX2 and 112.83% compared to the non-vectored version. As for Galois, an initial study was carried out that evaluated the performance of the MST (Minimum Spanning Tree) and the preliminary results point to a speedup of up to 6x with the `Epinions` dataset. In the future, it is expected to complement the Galois implementation for other OPF-based graph algorithms.

Keywords: High-performance computing, Neural networks, Computing.

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface
AVX	Advanced Vector Extensions
IPC	Instructions per Cycle
KNN	K Nearest Neighbor
NUMA	Non-Uniform Memory Access
OPF	Optimum Path Forest
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Extensions

Sumário

1 Introdução

A computação é uma ciência que está em constante evolução. Cada nova tecnologia desenvolvida permite avanços desses sistemas computacionais, que se superam dia após dia, e são capazes de realizar processamentos com uma rapidez jamais vista. Hoje, a tecnologia é mais acessível do que nunca, mas para atingir esse patamar, fora necessário o desenvolvimento não apenas de *hardware*, com microprocessadores capazes de realizar bilhões de instruções por segundo, mas também, *softwares* que são capazes de se aproveitar desse poder de processamento. Com essa mentalidade, este trabalho busca unir esses dois pontos, permitindo que algoritmos de aprendizado de máquina façam um melhor uso do *hardware* e utilize novas tecnologias para atingir um melhor desempenho.

1.1 Problema

Algoritmos de aprendizado de máquina se mostram essenciais para o futuro da computação, seja por meio de predições inteligentes que auxiliam o processamento do *hardware* ou até mesmo usos para visão computacional, que buscam revolucionar dentre outros, o setor médico (ESTEVA et al., 2021) e automobilístico (SONI et al., 2021). Mas, para isso tornar-se uma realidade, é necessário tornar tais tecnologias acessíveis a todos. Por meio de otimizações de *software*, é possível obter um melhor desempenho sem elevar os custos com *hardware*.

1.2 Justificativa

Com base na constante evolução da tecnologia, este estudo busca, por meio de técnicas contemporâneas, otimizar algoritmos de aprendizado de máquina com a finalidade de otimizar tais execuções, possibilitando uma maior popularização de *softwares* que utilizem tais algoritmos. Essa otimização torna possível a execução desses *softwares* em um maior espectro de dispositivos, demandando uma menor capacidade de *hardware* destinado a este processamento no mesmo tempo de execução.

1.3 Objetivos

Este trabalho busca, por meio da compreensão das tecnologias envolvidas, ou seja, dos mecanismos que fazem parte de algoritmos de aprendizado de máquina e tecnologias de otimização, como paralelização e vetorização, criar, a nível de aplicação, novas versões do código que permitam uma melhora no seu tempo de execução.

Este trabalho é organizado da seguinte forma. A Seção 2 apresenta a fundamentação teórica das principais tecnologias utilizadas. A Seção 3 apresenta as principais linguagens de programação e bibliotecas utilizadas neste trabalho, a Seção 4 apresenta a metodologia utilizada para a caracterização e os resultados experimentais dos estudos que resultaram na implementação vetorial, por fim apresentada na Seção 5. A Seção 6 comenta sobre e apresenta testes realizados com o sistema Galois. Por fim, a conclusão é apresentada na Seção 7.

2 Fundamentação Teórica

Dentre os tópicos abordados neste trabalho, os seguintes se destacam devido a sua importância e, portanto, mostra-se interessante apresentar uma breve contextualização sobre essas tecnologias.

2.1 Paralelização

Um dos desenvolvimentos mais icônicos das últimas décadas é a criação e popularização dos processadores multi-núcleos. Dentre os fatores que determinavam um melhor microprocessador, a frequência e IPC passaram a ser métricas secundárias, dando lugar à quantidade de núcleos físicos presentes (VENU, 2011). A presença de múltiplos núcleos permite a execução de diversas instruções simultaneamente, uma tecnologia capaz de elevar o poder de processamento de maneiras nunca antes vistas, mas com o viés de requerer a otimização do software para tirar proveito de tais capacidades.

As otimizações requeridas pelos programas nem sempre são triviais. O maior problema das abordagens em sistemas paralelos é que o desempenho da aplicação pode chegar próximo, mas nunca consegue escalar linearmente com o número de núcleos disponíveis. Existem diversos vieses que impedem esse comportamento linearizado de desempenho mas, o principal deles, é a dificuldade de realizar a divisão igualitária da execução da aplicação entre as múltiplas threads criadas (balanceamento de carga).

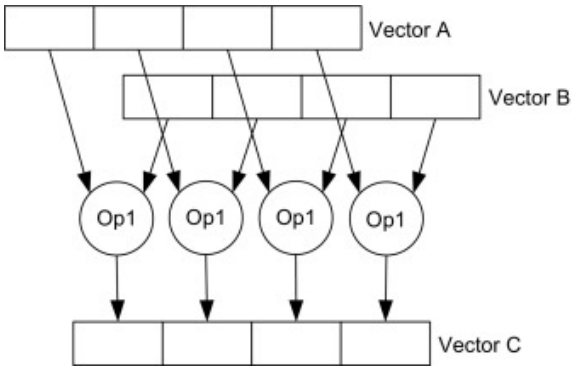
Atualmente, existem diversos facilitadores para a paralelização de código. No caso deste trabalho, ressalta-se a API OpenMP das linguagens C/C++ que permite por meio de *pragmas* a paralelização eficiente do código para execuções em CPUs, como foi utilizado no trabalho de Culquicondor et al. (2020).

2.2 Vetorização

Mesmo que o constante incremento de núcleos presentes em processadores seja evidentemente o principal fator de melhora de desempenho, outras tecnologias foram sendo desenvolvidas e incorporadas. Uma delas é o suporte para vetorização, comumente conhecido como SIMD (*Single Instruction Multiple Data*), advindo da Taxonomia de Flynn (FLYNN, 1966), cujo mecanismo básico de operações é apresentado na Figura 1, representando um exemplo no qual apenas um operador é aplicado a quatro operandos distintos em paralelo. Essa tecnologia, apesar de ter implementações nas mais diversas arquiteturas, desde x86 até ARM, possui nomenclaturas e funcionamentos claramente distintos. Aqui, o foco será na

implementação utilizada na arquitetura x86.

Figura 1 – Exemplo de operação de uma unidade SIMD.



Fonte: Science Direct - Single Instruction Multiple Data[1].

Instruções SIMD estão disponíveis em processadores Intel desde a criação do set de instruções MMX em 1997, com seus registradores de 64-bits dedicados. Hoje, processadores como o Intel Xeon Gold 5220 e o Intel Core i7 1065G7, utilizados para fins experimentais neste trabalho, possuem além de suporte para instruções MMX, suporte a tecnologias mais recentes, cronologicamente, SSE, com registradores de 128-bits, AVX, também referido como AVX2, com registradores de 256-bits e, por fim, AVX-512, com registradores de 512-bits. Este último, é o de maior interesse para os estudos a serem aqui apresentados. Ressalta-se ainda que para fins de simplicidade, neste trabalho a tecnologia AVX com registradores de 256-bits será referida como AVX2 e a sigla AVX será utilizada de maneira a se referir ao uso de quaisquer tecnologias dentre AVX2 e AVX-512.

2.3 Aprendizado de Máquina

Da mesma forma que a paralelização e a vetorização tiveram um avanço considerável nas últimas décadas, outra tecnologia que se beneficiou de tais avanços é o aprendizado de máquina, com os primeiros estudos realizados por Thearling (THEARLING, 1996). Somente com alto poder computacional passou a ser viável cogitar utilizar as técnicas de aprendizado de máquina já existentes com maior quantidade de dados e mecanismos mais complexos como o aprendizado de máquina profundo, o que também permitirá um grande avanço da tecnologia nos próximos anos (SZE et al., 2017). Hoje, ainda que algoritmos de classificação tradicionais como KNNs (THEARLING, 1996) ainda sejam utilizados, a tecnologia evoluiu, criando tecnologias como o aprendizado de máquina profundo, que, fazendo o uso de processamento heterogêneo, conseguiu revolucionar o setor de automação (BOJARSKI et al., 2016), saúde (CIREŞAN et al., 2013) e visão computacional (HUANG et al., 2017).

[1] Disponível em: <https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data>. Acesso em: 11 mai. 2022.

Ainda que seja um conceito relativamente simples, grafos vêm sendo utilizados nos últimos anos como uma abordagem promissora a ser aplicada em algoritmos de aprendizado de máquina, como o OPF (PAPA; FALCAO; SUZUKI, 2009) e aprendizado de máquina profundo (KIPF; WELLING, 2016). Tais mecanismos de grafos aplicados ao aprendizado de máquina passam a ser interessantes, uma vez que suas aplicações passam a ser capazes de resolver problemas que visam a classificação dos elementos por meio de classes não separáveis com formas arbitrárias, algo que até mesmo uma rede neural artificial baseada em perceptron (ANN-MLP) não consegue realizar (KUBAT, 1999). O trabalho realizado por Culquicondor et al. (2020) realizou uma implementação paralelizada do classificador OPF supervisionado, por meio de ferramentas como OpenMP e técnicas de vetorização, denominada Parallel OPF, ou simplesmente, POPF.

Visto que existe uma convergência entre trabalhos de processamento paralelo e, mais recentemente, aprendizado de máquina profundo, este trabalho visa aplicar métodos de otimização para algoritmos de aprendizado de máquina. Para isso, serão investigados tecnologias recentes para vetorização e métodos para exploração de paralelismo em máquinas com memória compartilhada. Dentre eles, serão contemplados vetorização utilizando AVX-512 (CORNEA, 2015) e paralelização utilizando arcabouços para computação paralela como o uso da biblioteca Galois (KULKARNI et al., 2007). Como existe uma gama elevada de algoritmos de aprendizado de máquina, o foco será na otimização de algoritmos baseados em OPF. Com base na investigação de vetorização e paralelização do OPF, busca-se compreender, de forma geral, maneiras de otimizar algoritmos também baseados em grafos, que abrangem até mesmo a mais recente ramificação de aprendizado de máquina, ou seja, aprendizado de máquina profundo, por meio das *Graph Convolutional Networks*, em tradução livre, Redes Convolucionais em Grafos (KIPF; WELLING, 2016).

3 Ferramentas

Para o desenvolvimento deste trabalho foi necessário o uso de algumas ferramentas que englobam linguagens de programação e bibliotecas especializadas. Ao se tratar do desenvolvimento de um estudo em ambiente computacional, fica clara a necessidade do domínio de linguagens de programação com a finalidade de criar *softwares* especializados, que visam a simplicidade de execução e desempenho.

3.1 *Bash Script*

Bash script é a linguagem do terminal Linux, portanto, ao se utilizar distribuições Linux para realizar os experimentos aqui apresentados, a linguagem torna-se bastante útil. Com ela é possível automatizar processos de execução, e até mesmo realizar instalações de *datasets* de modo bastante eficiente. No caso da implementação do OPF, o *bash script* foi utilizado de maneira a concatenar os resultados obtidos e facilitar a produção de gráficos e análises de resultados. O novo uso dessa ferramenta se mostra limitado devido a já existente automatização de execução no próprio POPF (CULQUICONDOR et al., 2020) realizada nesta mesma linguagem. Mesmo assim, o conhecimento da linguagem se mostrou útil até mesmo para modificar certas variáveis e manipular a execução dos experimentos para fins de diagnóstico.

3.2 Linguagem C

A linguagem de programação C é fundamental para este trabalho pois, além do OPF ter sido construído em sua maioria nessa linguagem, ela permite que sejam escritos códigos mais próximos do baixo nível. Seu uso permite realizar otimizações em nível de *hardware* como a própria vetorização utilizando AVX2 e AVX-512, bastante abordada neste trabalho. Uma alternativa seria escrever o código na linguagem de montagem, cuja programação não é tão trivial. A paralelização na linguagem C foi realizada em experimentos como nas Seções 4 e 5 por meio da biblioteca OpenMP, que apresentou resultados promissores por permitir um excelente desempenho em CPU (LIBÓRIO; BALDASSIN, 2021).

3.3 Linguagem Python

A linguagem de programação Python é amplamente utilizada para fins de aprendizado de máquina, por ser extremamente funcional devido a sua natureza de ser uma linguagem de alto nível. Neste trabalho a linguagem foi utilizada devido à sua biblioteca `matplotlib`, para fins de interpretação gráfica dos resultados obtidos, uma vez que a linguagem é extremamente

versátil e permite com certa facilidade a interpretação de dados e criação de gráficos de fácil visualização.

4 Vetorização

A vetorização é uma maneira encontrada para otimizar processamento vetorial, de maneira que, com um conjunto de registradores especializados em cada um dos núcleos de um processador, uma única instrução possa ser aplicada a todo o vetor de dados. O AVX foi uma tecnologia popularizada pela Intel em 2011 com a arquitetura Sandy Bridge, com a finalidade de contribuir com a inovação tecnológica iniciada pela criação do conjunto de instruções MMX em 1997. Diferentemente do MMX, que contém registradores de 64-bits, o AVX2 tem suporte para registradores de até 256-bits. Em 2016 houve um outro incremento nessa tecnologia com a arquitetura Skylake da Intel, originando o AVX-512, com registradores de 512-bits, que elevaram o potencial da tecnologia.

Seus usos podem vir a se mostrar interessantes nas mais diversas aplicações. Inicialmente o uso do AVX-512 se limitava a experimentos científicos, majoritariamente utilizado em aplicações como a apresentada neste estudo, ou seja, em aplicações relacionadas a inteligência artificial. Hoje, novas aplicações para essa tecnologia estão presentes até mesmo no âmbito comercial de modo a melhorar o desempenho de aplicações. Um exemplo recente de tal implementação é no emulador *open-source* do console Playstation 3 da Sony, o RCPS3, que ao simular com mais eficiência as SPUs (*Synergistic Processor Unit*) da arquitetura CELL, foi possível observar em um processador com arquitetura Intel Alder Lake um ganho de 23% no desempenho se comparado a versão anterior que utilizava AVX2 como método principal de vetorização (JESTADT, 2022).

A implementação da tecnologia AVX-512 em microprocessadores x86 tem sido limitada a processadores Intel e seu futuro ainda é questionado no ambiente *desktop*, devido a mais recente linha de processadores Intel Core de geração 12 possuir uma revisão mais atualizada que retira o suporte da tecnologia devido a sua arquitetura heterogênea (ALCORN, 2022). A AMD por outro lado, busca implementar, de maneira inédita, a tecnologia AVX-512 em toda sua futura geração de processadores com a nova arquitetura Zen 4 (TYSON, 2022). É também bastante provável que, incentivado pela concorrência, a Intel volte a implementar AVX-512 também nos processadores *desktop*. Com isso, fica claro que existe um futuro para AVX-512 até mesmo fora do âmbito de microprocessadores destinados a sistemas escaláveis como servidores, no qual sua implementação já é indispensável para alguns *workloads* (SHABANOV; RYBAKOV; SHUMILIN, 2019).

Neste caso inicial, o trabalho busca por meio da vetorização do algoritmo de multiplicação matricial, verificar os benefícios em desempenho que podem vir do uso das tecnologias de vetorização AVX2 e AVX-512, assim como a utilização do OpenMP para implementar o paralelismo.

No contexto de sistemas com múltiplas *threads* como o aqui utilizado, torna-se importante comentar sobre o fator NUMA (*Non-Uniform Memory Access*). Tal característica se refere ao comportamento obtido pelo acesso a memória RAM em sistemas que possuem fisicamente um ou mais processadores. Neles, determinados núcleos possuem canais de memória RAM dedicados, resultando que o acesso a outros canais de memória deva ser realizado de modo indireto, aumentando a latência de acesso e podendo influenciar o desempenho da aplicação.

A implementação da tecnologia AVX-512 é feita com o uso dos registradores `zmm`, mas seu uso não é algo tão trivial quanto seu conceito. As instruções devem ser específicas para cada tecnologia específica e programadas sob a linguagem C/C++, neste caso, por meio de *intrinsics* (INTEL, 2022) ou até mesmo utilizando a linguagem de montagem (*Assembly*), que permite um maior nível de otimização para o código. Aqui, atem-se a utilização da linguagem de programação C para todos os testes realizados.

O algoritmo apresentado no Pseudo-código 1 utiliza a estrutura tradicional de multiplicação matricial, ou seja, a operação é composta por três laços da estrutura de repetição *for*, de maneira que a multiplicação das matrizes quadradas A e B resulte na matriz C, de mesmas dimensões. No caso da vetorização, é utilizado o suporte de funções intrínsecas fornecidas pelo compilador para realização da multiplicação vetorial. Esta operação é mostrada de forma simplificada entre as linhas 8 e 11 do pseudo-código.

Para realizar a multiplicação vetorial, a quantidade de dados que deve ser carregada depende do tipo do dado e também da largura do vetor. No exemplo, usa-se dados de 64 bits (*double*) e a largura do vetor de 256 (AVX2) ou 512 (AVX-512). Assim, o número de colunas da matriz B (linha 5) é divido pela largura do vetor (LV) correspondente: 4 (256/64) para AVX2 e 8 para AVX-512 (512/64).

Pseudo-código 1 – Multiplicação de matrizes com AVX

```
1  inicio
2    // matrizes: A, B e C
3    // vetores AVX: Aavx, Bavx, Cavx, aux
4    para linha de 0 ate numero de linhas de A faca
5      para coluna de 0 ate numero de colunas de B/LV faca
6        Cavx = 0;
7        para i de 0 ate numero de colunas de A faca
8          Aavx = conteudo da matriz em A[linha][i];
9          Bavx = conteudo da matriz em B[i][coluna];
10         aux = multiplicacao dos vetores Aavx e Bavx;
11         Cavx = soma elementos de aux e Cavx;
12       fimpara
13       armazena dados do vetor Cavx em C[i][j];
14     fimpara
15   fimpara
16 fim
```

4.1 Metodologia

O sistema utilizado como padrão para a execução dos experimentos realizados neste trabalho é equipado com dois processadores Intel Xeon Gold 5220, totalizando 36 núcleos físicos e 72 lógicos, com o sistema operacional CentOS 7.7. Foi utilizado também o compilador GCC 7.3.1, as variáveis de ambiente CFLAGS com o *switch* de otimização -O3 para fins de otimização de compilação e, além da *flag* para o uso das instruções AVX2 e AVX-512, a variável *march* foi utilizada para maximizar o uso do *hardware* utilizado. Foi empregada a política de alocação de *threads close*, que busca utilizar *threads* fisicamente mais próximas, ou seja, no mesmo *socket*, aumentando a localidade dos dados e possibilitando uma menor latência no acesso à memória compartilhada.

Para assegurar a consistência dos testes, o tempo de execução das operações é calculado por meio da função *gettimeofday*, da biblioteca `time.h` (RATHORE; KUMAR, 2014). Os valores apresentados neste estudo são baseados na média de 30 execuções realizadas para cada configuração apresentada, com valores de ponto flutuante de precisão dupla (*double*), com intervalos de confiança de 95%. O *speedup* foi obtido dividindo-se o tempo de execução sequencial pelos tempos com diferentes números de *threads*.

Para os experimentos, foram utilizadas matrizes quadradas com tamanhos de 4000x4000 elementos, escolhidas pelo tempo razoável de execução, adequadas a produção dos experimentos.

4.2 Resultados

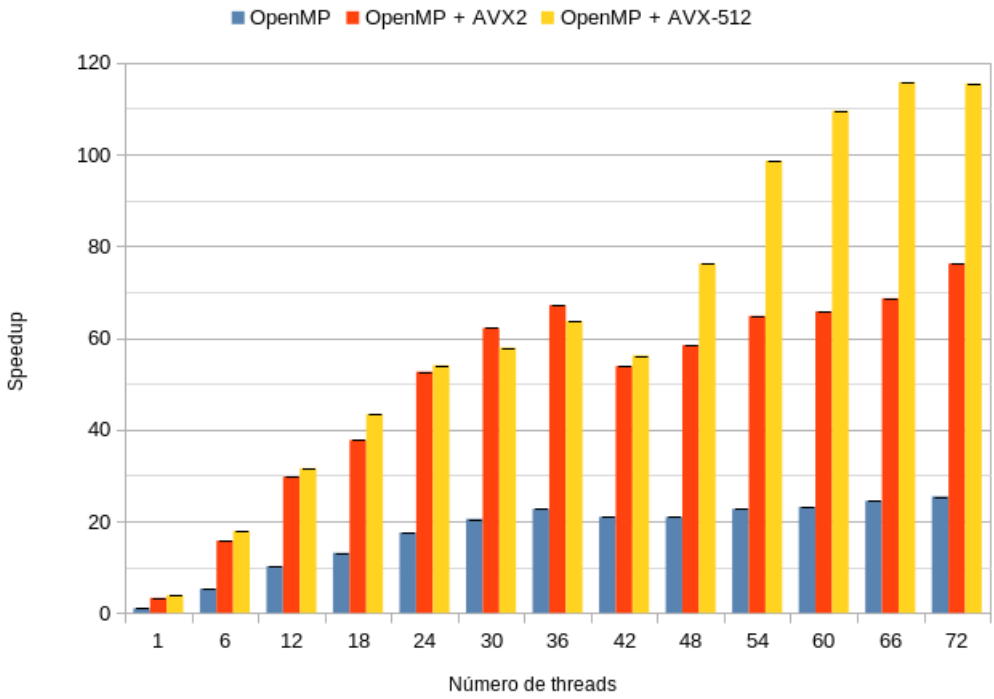
Os resultados da Figura 2 apresentam o *speedup* do código paralelizado, seguindo as configurações OpenMP, OpenMP+AVX2, OpenMP+AVX-512, para as matrizes de 4000x4000 elementos. O número de *threads* varia de 1 até 72.

É possível observar que, mesmo no caso com apenas 1 *thread*, existe uma diferença considerável de desempenho entre as três configurações. Utilizando a configuração sem vetorização como base, observa-se um *speedup* de 3,3x em relação ao AVX2 e 3,9x em relação ao AVX-512. Isso mostra que, mesmo obtendo um melhor desempenho, não consegue atingir o *speedup* teórico de 16x.

Os resultados com múltiplas *threads* evidenciam que, em geral, as versões do código que também utilizam vetorização tem desempenho maior do que a versão apenas com OpenMP. Há uma pequena queda de desempenho entre 36 e 42 *threads* devido ao fator NUMA já que, na configuração com 42 *threads*, os núcleos do outro *socket* começam a ser utilizados. Outro comportamento anômalo se refere às configurações de 30 e 36 *threads*. Nesses casos, o desempenho do AVX2 é ligeiramente superior ao AVX-512. Observa-se também, que a partir de 48 *threads*, o desempenho do AVX-512 é bastante superior ao do AVX2.

Ainda que os resultados apresentados na Figura 2 apresentem uma melhora significativa

Figura 2 – Gráfico de *speedup* global das operações de multiplicação matricial comparando o desempenho das tecnologias de vetorização.



Fonte: Elaborada pelo autor.

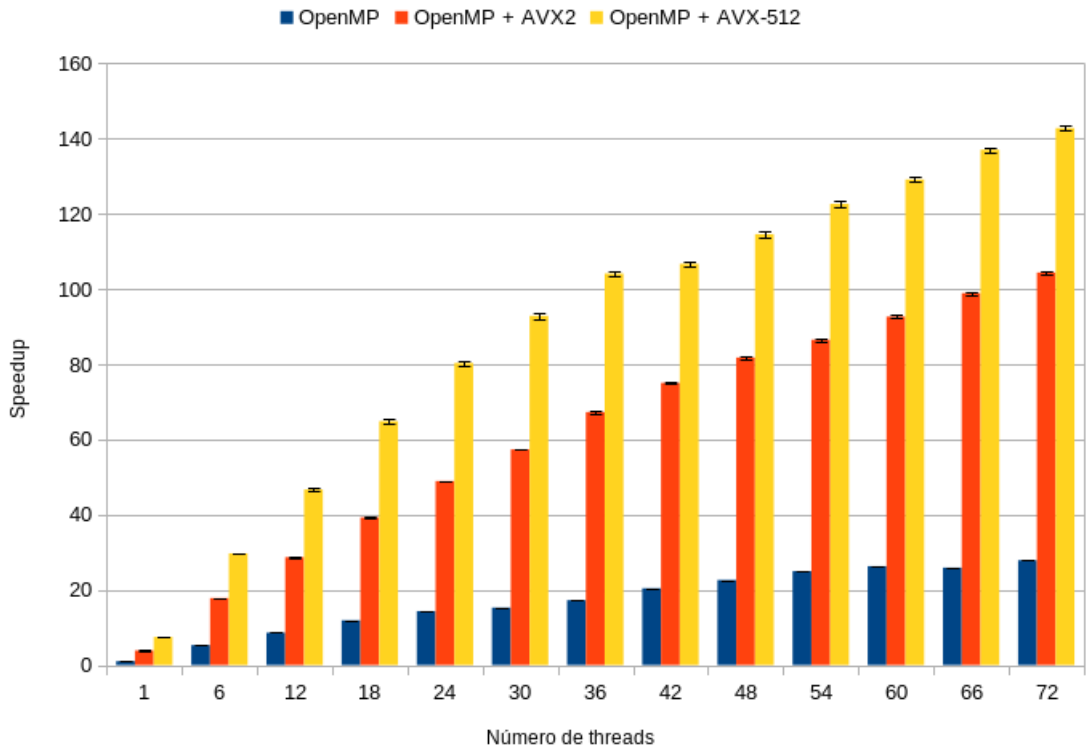
de desempenho de maneira geral, mesmo que este experimento se limite apenas ao estudo da tecnologia de vetorização, é de extrema importância ressaltar que otimizações neste código podem trazer ainda maiores benefícios ao comportamento observado.

O Pseudo-código 2 apresenta a otimização trazida pela utilização da matriz transposta de B (Bt) no cálculo da multiplicação matricial, cujos resultados são apresentado pela Figura 3. Isso é realizado de forma a otimizar a localidade de acesso no laço interno, resultando em um *speedup* mais próximo do esperado, sem as anomalias apresentadas pela Figura 2.

Pseudo-código 2 – Multiplicação de matrizes com AVX utilizando a otimização da matriz transposta Bt

```
1  inicio
2    // matrizes: A, Bt e C
3    // vetores AVX: Aavx, Bavx, Cavx, aux
4    para linha de 0 ate numero de linhas de A faca
5      para coluna de 0 ate numero de colunas de Bt/LV faca
6        Cavx = 0;
7        para i de 0 ate numero de colunas de A faca
8          Aavx = conteudo da matriz em A[linha][i];
9          Bavx = conteudo da matriz em Bt[coluna][i];
10         aux = multiplicacao dos vetores Aavx e Bavx;
11         Cavx = soma elementos de aux e Cavx;
12       fimpara
13       armazena dados do vetor Cavx em C[i][j];
14     fimpara
15   fimpara
16 fim
```

Figura 3 – Gráfico de *speedup* global das operações de multiplicação matricial comparando o desempenho das tecnologias de vetorização utilizando a otimização trazida pelas operações com matrizes transpostas.



Fonte: Elaborada pelo autor.

Os dados obtidos com 1 *thread* apresentam, ao comparar com a implementação sem vetorização, um desempenho relativo com a implementação AVX2 de 3,87x, e com AVX-512, de

expressivos 7,45x. Ao comparar esses dados com os obtidos anteriormente, é possível observar que o *speedup* agora fica muito mais próximo do máximo teórico. As configurações vetorizadas com mais *threads* também são beneficiadas pelo uso da matriz transposta, a configuração com 72 *threads* atinge 104,40x utilizando AVX2 e 142,82x utilizando AVX-512, se comparado aos experimentos sem o uso da transposta, com desempenho de respectivamente com AVX2 de 76,22x e com AVX-512 de 115,45x.

Portanto, pode-se afirmar que tais inconsistências observadas no caso da Figura 2 são advindas de limitações trazidas pela arquitetura do sistema utilizado, onde os resultados com a otimização de acesso à memória cache apresentam o *speedup* com comportamento de crescimento linear esperado.

É importante ressaltar, mesmo em algoritmos simples como este de multiplicação matricial, que o código ainda é passível de otimização. Mesmo que a implementação das instruções de vetorização já tragam uma melhora considerável de desempenho, uma otimização como é o caso desta com a matriz transposta, pode acentuar ainda mais os benefícios trazidos pela utilização de tecnologias como AVX2 e AVX-512.

5 OPF

O OPF (*Optimum-Path Forest*) é um algoritmo de aprendizado de máquina baseado em grafos. Nele, os nodos de um grafo são representados por um vetor de característica. Para calcular a semelhança entre os nodos, uma função para cálculo da distância entre cada par é utilizada. Tendo em vista que essa função é uma das partes do algoritmo que demanda maior tempo de computação, sua otimização passa a ser de extrema relevância. Assim, este trabalho busca substituir a técnica desenvolvida utilizando AVX2 realizada no trabalho de Culquicondor et al. (2020), por outra utilizando a mais recente AVX-512 e fazer uma análise dos resultados.

O cálculo da distância Euclidiana é uma abordagem bastante popular para o cálculo da distância entre dois pontos distintos, e possui um bom desempenho se comparado a outras abordagens (FAISAL; ZAMZAMI; SUTARMAN, 2020), por isso, é um excelente candidato para a vetorização. A Equação 5.1 apresenta o cálculo utilizado.

$$d(u,v) = D * log(1 + ||u - v||^2) = D * log(1 + \sum_{n=1}^{f}(ui - vi)^2) \tag{5.1}$$

onde D é uma constante, f o número de características, e u_i e v_i representam o i-ésimo componente dos vetores u e v, respectivamente. A somatória realizada no final da Equação 5.1 é uma operação realizada com vetores e pode se beneficiar da vetorização.

Para realizar a implementação, inicialmente, tornou-se interessante realizar o desenvolvimento de uma versão isolada do algoritmo de cálculo da distância euclidiana, com a finalidade de comparar os resultados obtido com e sem o uso de vetorização AVX e verificar sua corretude.

5.1 Experimentos iniciais

Com a finalidade de realizar a implementação vetorial do cálculo da distância euclidiana utilizando AVX-512, foram desenvolvidos três variações da implementação, sem vetorização, com AVX2 e com AVX-512, este último, representado no Pseudo-código 3.

A implementação AVX-512 apresentada faz o uso de *intrinsics*, que são instruções para as linguagens C/C++ utilizadas para o uso dos registradores vetoriais AVX. A estrutura de uma *intrinsic* se inicia por `_m` e separa os termos com por `_`. O primeiro dado passado é o tamanho do registrador vetorial. O segundo membro ou é vazio, ou pode acrescentar características para o processamento, como o uso de máscaras. O terceiro membro é aonde a operação é determinada, do tipo `load` por exemplo, realiza o carregamento da memória para os registradores, do tipo `add`, realiza adição, e `store` armazena os dados em um vetor AVX

para a memória. Por fim, o último membro da expressão descreve o tipo de variável que será utilizada, exemplos são `ps` para valores (*float*) e `pd` para inteiros.

Pseudo-código 3 – Implementação do cálculo da distância euclidiana utilizando AVX-512

```
1  inicio
2      //carrega os dados do vetor1 e vetor2 em respectivos vetores AVX-512
3      vetor1avx = _mm512_load_pd(&vetor1[i]);
4      vetor2avx = _mm512_load_pd(&vetor2[i]);
5
6      //realiza as operacoes de subtracao, multiplicacao e incremento
           final
7      vetorAux = _mm512_sub_pd(vetor1avx, vetor2avx);
8      vetormul = _mm512_mul_pd(vetorAux, vetorAux);
9      vetFinalAvx = _mm512_add_pd(vetormul, vetormul);
10
11     //armazena os valores finais presentes no vetor AVX-512 vetFinalAvx
           em vetorFinal
12     _mm512_storeu_pd(&vetorFinal[i], vetFinalAvx);
13 fim
```

Os resultados se mostraram corretos em todas as três versões desenvolvidas, ainda que tenham sido utilizados valores inteiros para simplificar a verificação dos resultados, diferentemente dos valores *float* utilizados no OPF.

Com os dados confirmados, torna-se possível realizar a implementação propriamente dita no algoritmo POPF e avaliar seu desempenho.

5.2 Implementação

Após o sucesso da implementação apresentada na Seção 5.1, torna-se viável utilizar estes mecanismos no próprio POPF. Este, por sua vez, já contém uma versão paralelizada do código original e uma implementação vetorial utilizando AVX2. Portanto, este trabalho visa atualizar essa implementação, utilizando o mais recente AVX-512. Para isso, fora utilizada uma implementação distinta da apresentada na Seção 5.1 a fim de manter a coerência com o formato dos dados de entrada e saída.

Inicialmente, o código para a implementação fora criado conforme o Pseudo-código 4, que passa a levar em consideração a entrada e saída de dados do POPF (linhas 1–2). O laço de repetição (linhas 4–13) permite utilizar o vetor de 512 bits para realizar a acumulação das diferenças entre os vetores (última parte da Equação 5.1). Cada característica do vetor é representado por um *float*, de 32 bits. Logo, é possível realizar 16 operações simultâneas usando a vetorização com o AVX-512.

Finalizando o pseudo-código está representado o processo de redução e armazenamento

dos dados resultantes (linhas 15–25). A redução dos valores dos vetores 512 bits permite o registro do valor acumulado (`vetAccAvx`) para um valor de 32 bits (`dist`), que é então retornado pela função.

A forma utilizada para redução foi primeiro separar a parte alta e baixa do vetor de 512 bits em dois vetores de 256 bits (linhas 15 e 16), então é feita a soma desses dois vetores (linha 17). O vetor de 256 bits é dividido em dois vetores de 128 bits, e é criado um vetor (tmp) que inverte essas duas partes (linha 18). É feita a soma desses dois vetores de 256 bits (linha 19), fazendo com que o vetor `vetAvxFinal` de 256 bits se comporte como dois vetores de 128 bits repetidos. Então, são feitas somas horizontais, representada pela *intrinsic* `hadd` (linhas 20–21), esse processo é utilizado para somar os valores próximos e alocar seus resultados de maneira que o vetor final possua apenas valores iguais. O vetor é então armazenado em tmp2 (linha 23) e o primeiro desses valores é utilizado para incrementar a distância (dist), que é o valor de saída da função.

Pseudo-código 4 – Implementação inicial do cálculo da distância euclidiana no POPF usando AVX-512

```
1  Entrada: vetor1, vetor2, qtdFeatureDataset
2  Saida: dist
3  inicio
4      for (i=0; i <= qtdFeatureDataset-16; i+=16){
5          //carregamento do trecho do vetor de 16 elementos
6          vetorAvx1 = _mm512_load_ps(&vetor1[i]);
7          vetorAvx2 = _mm512_load_ps(&vetor2[i]);
8
9          //calculo da distancia euclidiana
10         vetSubAvx = _mm512_sub_ps(vetorAvx1, vetorAvx2);
11         vetMulAvx = _mm512_mul_ps(vetSubAvx, vetSubAvx);
12         vetAccAvx = _mm512_add_ps(vetAccAvx, vetMulAvx);
13     }
14     //processo de reducao e armazenamento do resultado
15     __m256 low  = _mm512_castps512_ps256(vetAccAvx);
16     __m256 high = _mm256_castpd_ps(_mm512_extractf64x4_pd (
           _mm512_castps_pd(vetAccAvx),1));
17     vetAvxFinal = _mm256_add_ps(low, high);
18     __m256 tmp = _mm256_permute2f128_ps(vetAvxFinal,vetAvxFinal, 0x1);
19     vetAvxFinal = _mm256_add_ps(vetAvxFinal, tmp);
20     vetAvxFinal = _mm256_hadd_ps(vetAvxFinal, vetAvxFinal);
21     vetAvxFinal = _mm256_hadd_ps(vetAvxFinal, vetAvxFinal);
22
23     float tmp2[8] __attribute__((aligned(32)));
24     _mm256_store_ps(tmp2, vetAvxFinal);
25     dist += tmp2[0];
26 fim
```

Durante a implementação, um empecilho encontrado foi a maneira a qual estava sendo realizado o alinhamento de memória. Para realizar operações vetoriais com o AVX-512, fora necessário modificar o POPF a fim de que as alocações de memórias nos vetores de características estivessem alinhados em páginas de 64 bytes. Isso foi feito por meio de chamadas POSIX `memalign`. Outra alternativa que poderia solucionar tal problema seria a utilização da *intrinsic* `loadu` que, ao contrário do `load` aqui utilizado, permite o carregamento de dados não alinhados. Porém, devido a penalidades de desempenho que poderia vir a apresentar, a abordagem de já alocar a memória de forma alinhada se mostrou uma melhor opção.

Os resultados passaram a se mostrar corretos, porém, o problema de desempenho tornou-se aparente uma vez que vários testes indicaram que a implementação anterior em AVX2 ainda apresentava melhores resultados. A primeira tentativa para solucionar tal problema foi a mudança das linhas 10–12 do Pseudo-código 4 para a linha 10 do Pseudo-código 5. Esta simples mudança permite utilizar a otimização do próprio compilador GCC, o que mostrou-se beneficial para o desempenho global da aplicação.

Pseudo-código 5 – Implementação inicial otimizada do cálculo da distância euclidiana no POPF usando AVX-512

```
1  Entrada: vetor1, vetor2, qtdFeatureDataset
2  Saida: dist
3  inicio
4      for (i=0; i <= qtdFeatureDataset-16; i+=16){
5          //carregamento do trecho do vetor de 16 elementos
6          vetorAvx1 = _mm512_load_ps(&vetor1[i]);
7          vetorAvx2 = _mm512_load_ps(&vetor2[i]);
8
9          //calculo da distancia euclidiana
10         vetAccAvx = vetAccAvx + ((vetorAvx1 - vetorAvx2) * (vetorAvx1 -
               vetorAvx2));
11     }
12     //processo de reducao e armazenamento do resultado
13     __m256 low  = _mm512_castps512_ps256(vetAccAvx);
14     __m256 high = _mm256_castpd_ps(_mm512_extractf64x4_pd (
           _mm512_castps_pd(vetAccAvx),1));
15     vetAvxFinal = _mm256_add_ps(low, high);
16     __m256 tmp = _mm256_permute2f128_ps(vetAvxFinal,vetAvxFinal, 0x1);
17     vetAvxFinal = _mm256_add_ps(vetAvxFinal, tmp);
18     vetAvxFinal = _mm256_hadd_ps(vetAvxFinal, vetAvxFinal);
19     vetAvxFinal = _mm256_hadd_ps(vetAvxFinal, vetAvxFinal);
20
21     float tmp2[8] __attribute__((aligned(32)));
22     _mm256_store_ps(tmp2, vetAvxFinal);
23     dist += tmp2[0];
24 fim
```

Para realizar o uso da tecnologia AVX é necessário preencher todo o vetor. A mudança da implementação com AVX2 para AVX-512 implica que a quantidade de *features*, ou seja, características do *dataset* necessário para o uso do cálculo vetorial passou de 8 para 16. Para contornar o problema, fora utilizada uma implementação híbrida, com estruturas condicionais que avaliam a situação, determinando o cálculo com AVX-512 como de maior prioridade, seguido pelo cálculo utilizando AVX2, e por fim, no caso de não haver um número de características restantes adequado para preencher um vetor de 512 ou 256 bits, a versão do algoritmo sem vetorização é utilizada. Tal mecanismo pode ser observado nas linhas 5 e 31 do Pseudo-código 6.

Pseudo-código 6 – Implementação final otimizada do cálculo da distância euclidiana no POPF usando AVX-512

```
1  Entrada: vetor1, vetor2, qtdFeatureDataset
2  Saida: dist
3  inicio
4      qtdFeatureCopia = qtdFeatureDataset
5      if (qtdFeatureCopia>=16){
6        qtdFeatureCopia=qtdFeatureCopia-16;
7        for (i=0; i <= qtdFeatureCopia-16; i+=16){
8
9          //carregamento do trecho do vetor de 16 elementos
10         vetorAvx1 = _mm512_load_ps(&vetor1[i]);
11         vetorAvx2 = _mm512_load_ps(&vetor2[i]);
12
13         //calculo da distancia euclidiana
14         vetAccAvx = vetAccAvx + ((vetorAvx1 - vetorAvx2) * (vetorAvx1 -
               vetorAvx2));
15         }
16
17         //processo de reducao e armazenamento do resultado
18         __m256 low  = _mm512_castps512_ps256(vetAccAvx);
19         __m256 high = _mm256_castpd_ps(_mm512_extractf64x4_pd (
               _mm512_castps_pd(vetAccAvx),1));
20         vetAvxFinal = _mm256_add_ps(low, high);
21         __m256 tmp = _mm256_permute2f128_ps(vetAvxFinal,vetAvxFinal, 0x1
               );
22         vetAvxFinal = _mm256_add_ps(vetAvxFinal, tmp);
23         vetAvxFinal = _mm256_hadd_ps(vetAvxFinal, vetAvxFinal);
24         vetAvxFinal = _mm256_hadd_ps(vetAvxFinal, vetAvxFinal);
25
26         float tmp2[8] __attribute__((aligned(32)));
27         _mm256_store_ps(tmp2, vetAccAvxFinal);
28         dist += tmp2[0];
29     }
30
31     if (qtdFeatureCopia>=8 && qtdFeatureCopia<16){
32       qtdFeatureCopia=qtdFeatureCopia-8;
```

```
33          for (i=0; i <= qtdFeatureCopia-8; i+=8){
34
35            //carregamento do trecho do vetor de 8 elementos
36            vetorAvx1 = _mm256_load_ps(&f1[i]);
37            vetorAvx2 = _mm256_load_ps(&f2[i]);
38
39            //calculo da distancia euclidiana
40            vetAccAvx = vetAccAvx + ((vetorAvx1 - vetorAvx2) * (vetorAvx1 -
                  vetorAvx2));
41          }
42
43          //processo de reducao e armazenamento do resultado
44          __m256 tmp = _mm256_permute2f128_ps(vetAccAvx,vetAccAvx, 0x1);
45          vetAccAvx = _mm256_add_ps(vetAccAvx, tmp);
46          vetAccAvx = _mm256_hadd_ps(vetAccAvx, vetAccAvx);
47          vetAccAvx = _mm256_hadd_ps(vetAccAvx, vetAccAvx);
48
49          float tmp2[8] __attribute__((aligned(32)));
50          _mm256_store_ps(tmp2, vetAccAvx);
51          dist += tmp2[0];
52        }
53        //caso o numero de features nao seja suficiente para usar vetor AVX
54        for (i=0; i < qtdFeatureCopia; i++)
55            dist += (vetor1[i]-vetor2[i])*(vetor1[i]-vetor2[i]);
56    fim
```

5.3 Metodologia

O sistema e configurações de compilação se repetem do apresentado na Subseção 4.1, com a distinção de que os valores aqui apresentados são baseados na média de 5 execuções realizadas para cada configuração apresentada, também obtidos por meio da função *gettimeofday*, da biblioteca `time.h`, utilizando valores de ponto flutuante (*float*).

As bases de dados, ou *datasets*, aqui utilizadas foram também utilizadas no artigo referente ao POPF (CULQUICONDOR et al., 2020) para fins de validação dos resultados (LICHMAN, 2013).

5.4 Resultados

Uma consideração necessária ao utilizar cálculos vetoriais é quanto a quantidade de características do *dataset* em questão. Para determinar se uma tecnologia de vetorização pode ser utilizada pelo *dataset*, deve-se realizar o cálculo apresentado pela Inequação 5.2. Nele, o resultado deve ser maior ou igual a 1 para que o cálculo possa ser vetorizado e a parte decimal do valor obtido é referente aos valores que não serão vetorizados pela tecnologia. Em casos

de *datasets* como o `MiniBooNE`, com tamanhos de *feature* igual a 50 por exemplo, 48 dessas características conseguem preencher vetores AVX-512 (48/16=3), permitindo o cálculo de forma vetorizada. Já para as características restantes, os cálculos necessários são realizadas de maneira tradicional, ou seja, sem a vetorização aplicada. Idealmente, espera-se que a tal valor obtido pela Inequação 5.2 seja inteiro, neste caso, múltiplo de 1, para obter um maior desempenho.

$$(featuresDataset * tamanhoDado)/tamanhoVetorAVXEmBytes \geq 1 \tag{5.2}$$

Assim, dentre os *datasets* utilizados no POPF (CULQUICONDOR et al., 2020), pode-se observar na Tabela 1 apenas o `MiniBooNE`, `SDD` e `Letter` são interessantes para este estudo, pois podem apresentar benefícios do uso da tecnologia AVX-512. A tabela contém na primeira coluna o nome do *dataset*, a segunda o número de instâncias (nodos do grafo) e a terceira o número de características de cada vetor.

Tabela 1 – Descrição dos *datasets* utilizados neste trabalho

Dataset	**Instâncias**	**Características**
MiniBooNE	130.064	50
SDD	58.509	48
Letter	20.000	16

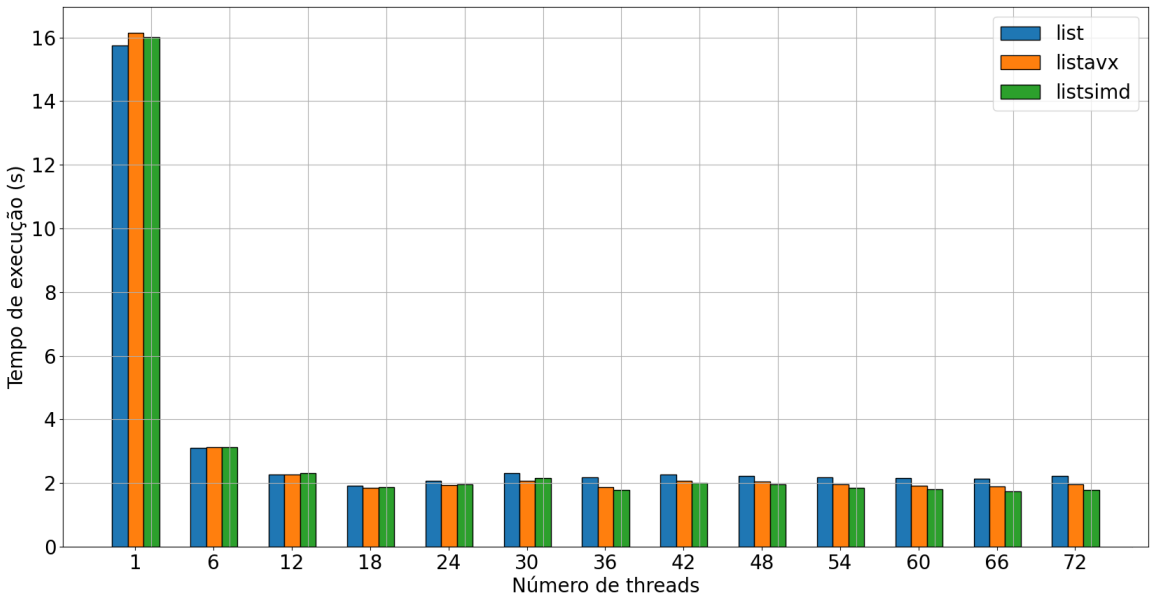
Fonte: Elaborada pelo autor.

Durante a realização dos experimentos foi possível observar que a aplicação OPF não possui ganhos significativos após 24 *threads*, conforme pode ser observado na Figura 4, que faz parte dos estudos iniciais relacionados à confirmação do desempenho do POPF. Por fins de simplicidade, apenas o *dataset* `Letter` é apresentado, mas tal comportamento é também observado nos demais.

Os testes iniciais buscaram analisar apenas os ganhos trazidos pelo uso da vetorização. Para isso, o POPF foi executado de forma sequencial, ou seja, com 1 *thread* e seu tempo de execução medido nas três configurações, sem vetorização, com AVX2 e também com AVX-512, cujos resultados são apresentados na Tabela 2.

A primeira coluna contém o nome do *dataset*, a segunda o número de características de cada vetor, e então os tempos nas configurações sem técnicas de vetorização (NoVect), AVX2 e AVX-512. As últimas colunas apresentam os ganhos com a vetorização AVX-512 comparada ao sequencial (sem vetorização) e AVX2. Como esperado, o ganho obtido é diretamente relacionado ao número de características do *dataset*. Seguindo a lógica apresentada na Inequação 5.2, é possível observar que o `SDD` deveria apresentar o melhor ganho de desempenho, uma vez que o cálculo de distância completo pode ser realizada com três operações vetoriais $(3 * 16)$. O ganho da vetorização obtido com o `MiniBooNE` é relativamente menor porque como o número de características não é múltiplo de 16, parte do vetor (no caso, 2 características) precisa ser

Figura 4 – Gráfico de tempo utilizado para confirmação dos resultados do artigo POPF (CULQUI-CONDOR et al., 2020) para as configurações do tipo `list` com o *dataset* `Letter`.



Fonte: Elaborada pelo autor.

onde `list` representa a configuração aqui denominada NoVect, mas no contexto do POPF, se trata da versão que substituiu a fila de prioridade por uma estrutura de lista linear, `listavx` incorpora as instruções vetoriais AVX2 e `listsimd` é a que utiliza a auto-vetorização por meio da cláusula `SIMD` do OpenMP.

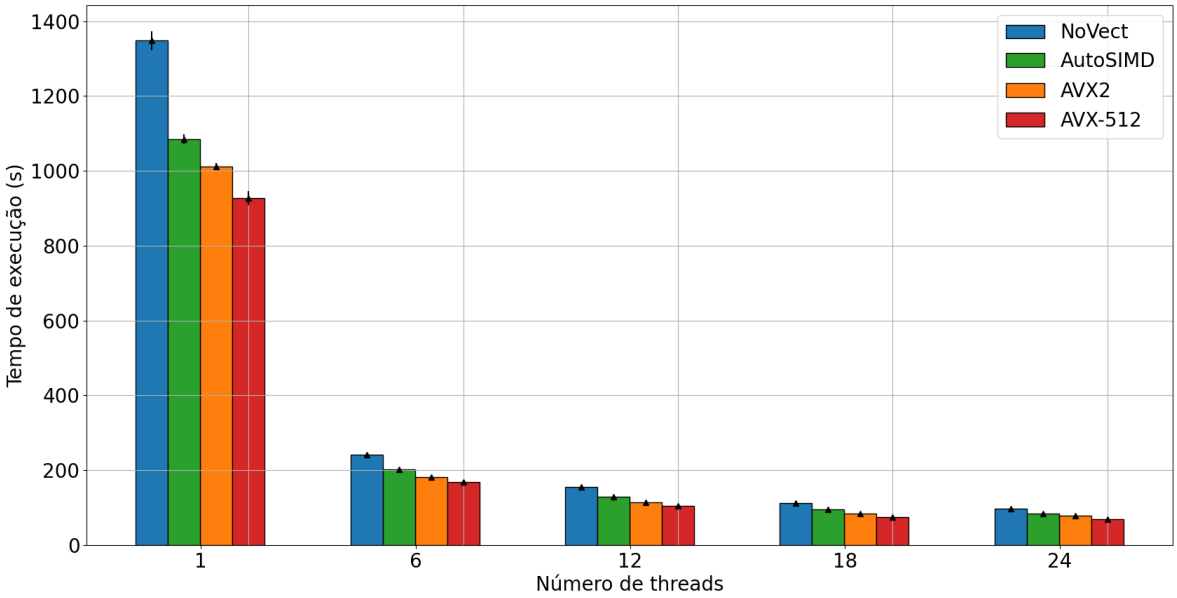
Tabela 2 – Tempo total de execução em segundos do POPF para a configuração de 1 *thread*.

Dataset	**Caract.**	**NoVect**	**AVX2**	**AVX-512**	**NoVect/AVX-512**	**AVX2/AVX-512**
MiniBooNE	50	1348,50	1011,60	927,68	45,36%	9,05%
SDD	48	120,84	90,78	73,49	64,43%	23,53%
Letter	16	15,75	16,15	13,10	20,23%	23,28%

Fonte: Elaborada pelo autor.

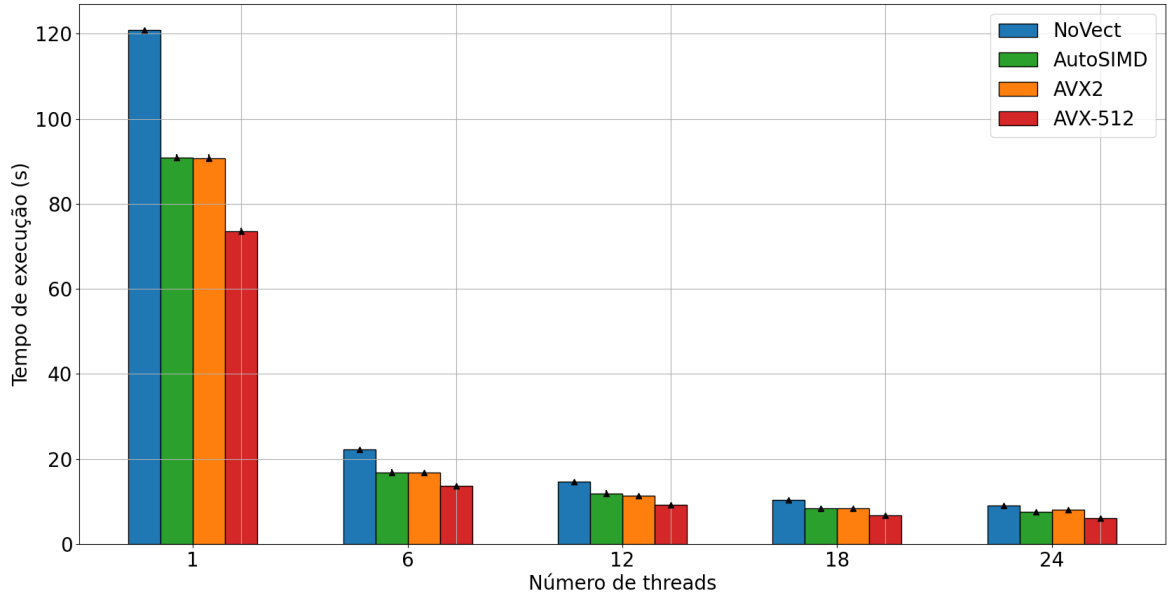
calculado de forma sequencial. Finalmente, o *dataset* `Letter` apresenta ganhos mais modestos. Com o AVX2, na verdade, há uma leve perda devido ao *overhead* necessário para preparar os operandos para a vetorização.

Figura 5 – Gráfico de tempo da versão final, comparando as execuções iniciais do POPF com a implementação de AVX-512 no *dataset* `MiniBooNE`.



Fonte: Elaborada pelo autor.

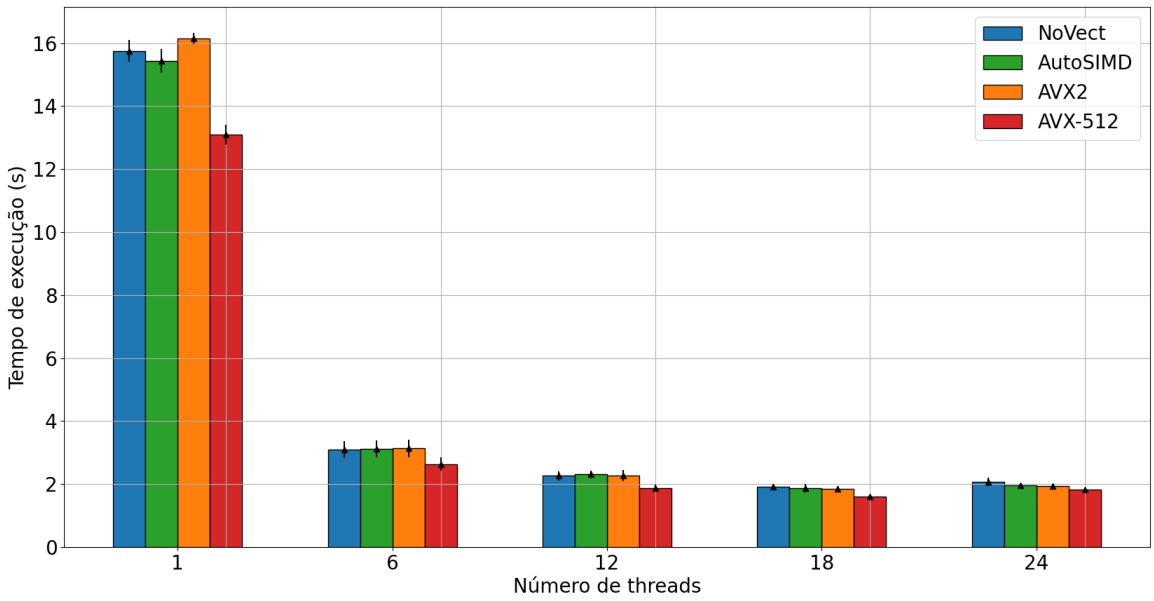
Figura 6 – Gráfico de tempo da versão final, comparando as execuções iniciais do POPF com a implementação de AVX-512 no *dataset* `SDD`.



Fonte: Elaborada pelo autor.

Os experimentos seguintes mostram o comportamento das execuções conforme o número de *threads* varia. Para esses experimentos, além da versão original não vetorizada (`NoVect`), do AVX2 e do AVX-512, também é usado uma versão obtida com o auto-vetorização por meio da cláusula `SIMD` do OpenMP (`AutoSIMD`). Os resultados para os *datasets* apresentados na Tabela 2 estão ilustrados nas Figuras 5, 6 e 7.

Figura 7 – Gráfico de tempo da versão final, comparando as execuções iniciais do POPF com a implementação de AVX-512 no *dataset* `Letter`.



Fonte: Elaborada pelo autor.

Ao analisar os resultados de 1 a 24 *threads* é possível observar que o ganho com a vetorização se torna menos relevante conforme o número de *threads* aumenta. Tal comportamento pode ser explicado pelo fato de que a paralelização das *threads* já proporciona um bom ganho no desempenho, reduzindo o impacto da vetorização. Em particular, o ganho do AVX-512 quando comparado ao AVX2 é bem pequeno para o `MiniBooNE` a partir de 6 *threads* como mostrado na Figura 5. Os gráficos também mostram um bom desempenho da versão com auto-vetorização do OpenMP, principalmente no caso do `SDD` e `Letter`. Mesmo assim, a vetorização com o AVX-512 possui em geral o melhor desempenho em todas as configurações.

5.5 Verificação de desempenho

Para validar os resultados obtidos anteriormente na Seção 5.4 de maneira a não limitar os testes a uma única arquitetura de microprocessadores, os testes foram também realizados em um sistema mais acessível. Esse sistema, no formato *ultrabook* (HERMERDING et al., 2011), possui uma implementação menos robusta da tecnologia AVX-512 em seus núcleos Intel Sunny Cove se comparado a implementações disponíveis no mercado de servidores (SUNNY. . . , 2022).

5.5.1 Metodologia

O sistema utilizado é equipado com um processador Intel Core i7 1065G7 de 4 núcleos físicos e 8 núcleos lógicos, 16GB de memória RAM e o sistema operacional Pop_OS 22.04

LTS, com o compilador GCC na versão 11.2.0. Os demais detalhes se repetem e podem ser encontrados na Subseção 5.3.

5.5.2 Resultados

Os dados obtidos nos experimentos com este sistema IceLake estão apresentados na Tabela 3. A primeira coluna apresenta o nome do *dataset*, a segunda o número de características de cada vetor, e então os tempos NoVect (não vetorizada), AVX2 e AVX-512. A última coluna apresenta os ganhos com a vetorização AVX-512 comparada a versão sem vetorização e com AVX2. Os resultados mostram que o ganho de desempenho é presente mesmo em uma arquitetura com implementação AVX-512 menos robusta, e com valores bastante expressivos, principalmente no caso do *dataset* `SDD`, onde foi possível observar um ganho de desempenho de 112,83% ao comparar a configuração sequencial e AVX-512.

Tabela 3 – Tempo total de execução em segundos do POPF para a configuração de 1 *thread* no sistema IceLake.

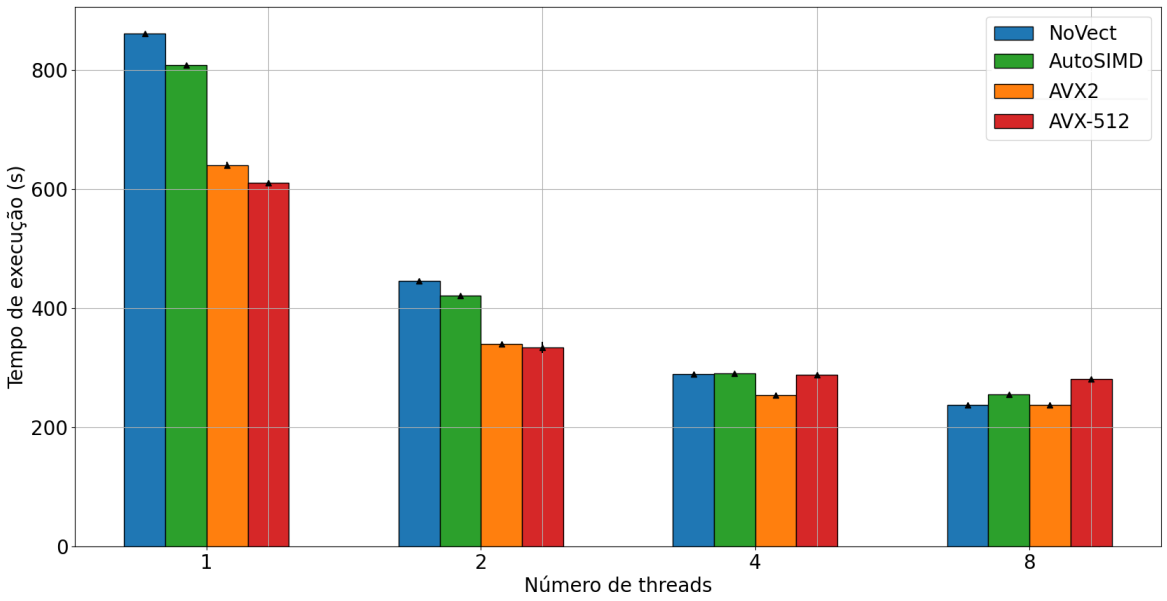
Dataset	**Caract.**	**NoVect**	**AVX2**	**AVX-512**	**NoVect/AVX-512**	**AVX2/AVX-512**
MiniBooNE	50	861,63	640,05	611,26	40,96%	4,71%
SDD	48	56,23	32,52	26,42	112,83%	23,09%
Letter	16	6,03	6,38	5,07	18,93%	25,84%

Fonte: Elaborada pelo autor.

Essa melhoria de desempenho ao comparar com resultado obtido no sistema anterior, aqui referido como CascadeLake, se deve às melhorias trazidas pela arquitetura Intel Ice Lake mais recente (2019), uma vez que a arquitetura Intel Cascade Lake (2019) é a terceira revisão da arquitetura Intel Skylake X, lançada em 2017. Essas melhorias incluem, mas não se limitam, a maiores frequências de *clock*, maior IPC e uma melhor implementação AVX-512.

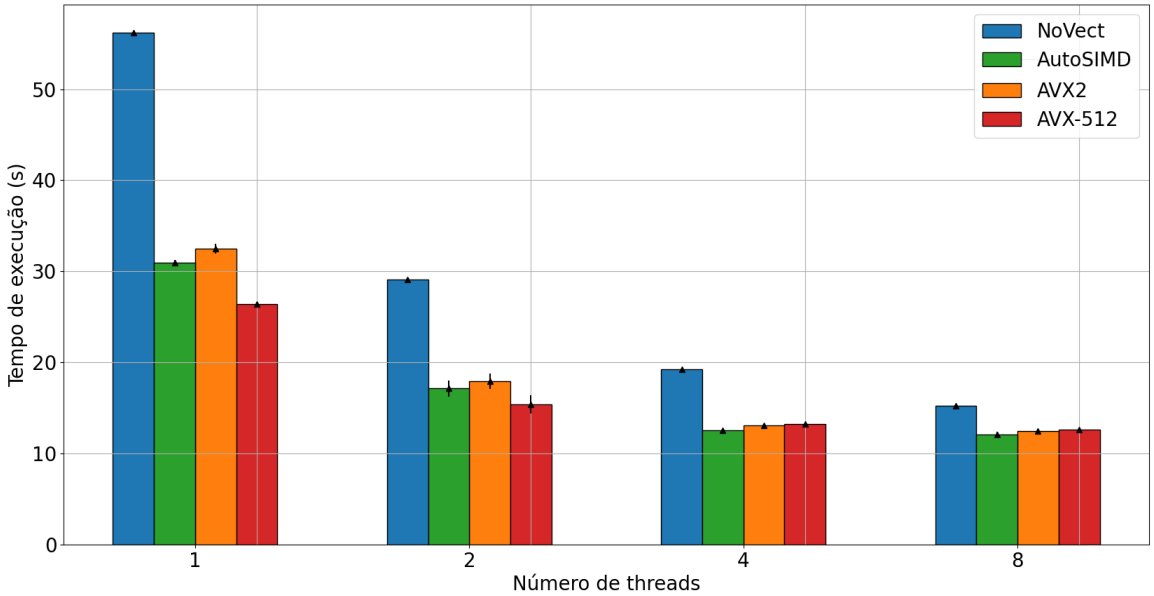
Foram realizados experimentos com mais *threads*, apresentados nas Figuras 8, 9 e 10, apresentando as configurações NoVect, AutoSIMD, obtida com o auto-vetorização por meio da cláusula `simd` do OpenMP e as duas implementações AVX2 e AVX-512. Pelo comportamento apresentado nos gráficos é possível observar as limitações da implementação AVX-512 nesse sistema ao comparar as execuções com 1 e 2 *threads* com as demais. Nos casos com 4 *threads*, é possível observar inconsistências, no *dataset* `MiniBooNE` a implementação com AVX-512 mostra piores resultados que com AVX2, no *dataset* `SDD`, a versão com AVX-512 apresenta um pior desempenho até mesmo do que com AVX2 e AutoSIMD e no *dataset* `Letter`, as configurações AutoSIMD e AVX2 apresentam inconsistências e a execução não vetorizada apresenta o melhor desempenho. As configurações com 8 *threads* são apresentadas apenas com a finalidade de apresentar o comportamento no caso de uso da tecnologia *hyper-threading*. Pelos dados obtidos é possível observar que essas operações pouco se beneficiam da utilização dos núcleos lógicos nessa arquitetura.

Figura 8 – Gráfico de tempo da versão final, comparando as execuções iniciais do POPF com a implementação de AVX-512 no *dataset* `MiniBooNE` com o sistema IceLake.



Fonte: Elaborada pelo autor.

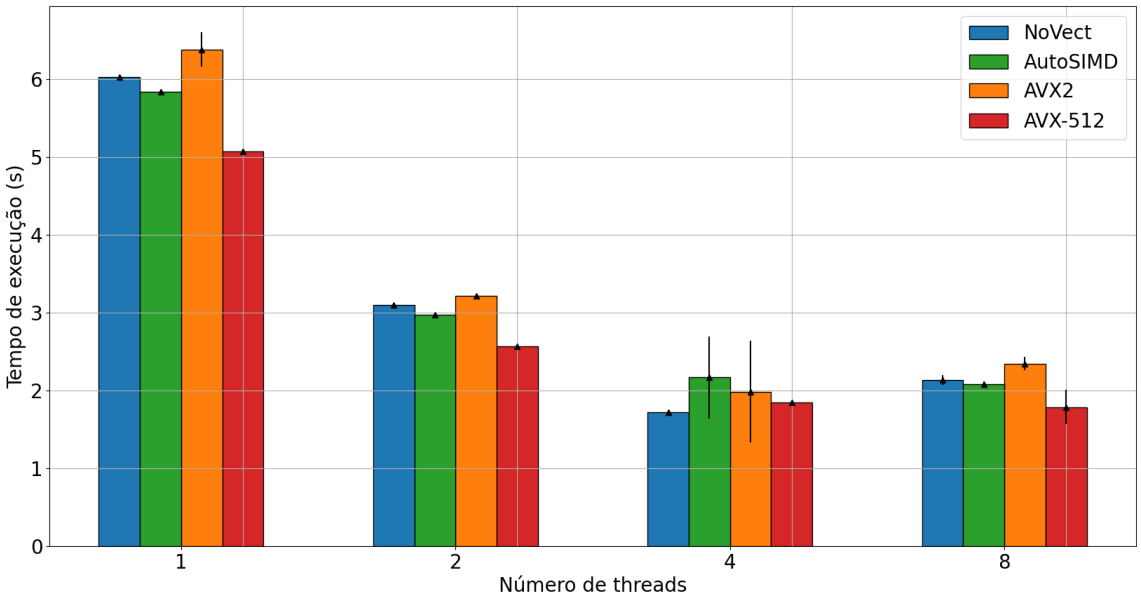
Figura 9 – Gráfico de tempo da versão final, comparando as execuções iniciais do POPF com a implementação de AVX-512 no *dataset* `SDD` com o sistema IceLake.



Fonte: Elaborada pelo autor.

Após uma inspeção mais detalhada com o software de diagnóstico Intel VTune Profiler, fora relatado que não existem fatores limitantes no sistema durante a execução de tais experimentos, limitados apenas pela própria capacidade de processamento. Mas tal comportamento, pode-se dar a limitação da potência energética utilizada pelo processador, de 15 Watts, que causa uma variação significativa na frequência do processador durante os *workloads* com 4 núcleos ou mais, explicando as oscilações no *dataset* `Letter` e o baixo desempenho de maneira

Figura 10 – Gráfico de tempo da versão final, comparando as execuções iniciais do POPF com a implementação de AVX-512 no *dataset* `Letter` com o sistema IceLake.



Fonte: Elaborada pelo autor.

geral nas configurações com 4 e 8 *threads*.

Mesmo usando um sistema equipado com um processador voltado para o mercado de dispositivos portáteis, o sistema IceLake é capaz de executar a aplicação de maneira adequada, visto que o comportamento de obter os maiores ganhos com poucos núcleos por meio da vetorização se assemelham ao o sistema CascadeLake.

6 Galois

O sistema Galois é um arcabouço computacional que permite explorar o paralelismo de dados amorfos em algoritmos irregulares na linguagem C++ sem a necessidade de uma programação paralela explícita (KULKARNI et al., 2007). Neste caso em específico, ele foi escolhido devido a seu bom desempenho em algoritmos baseados em grafos (WHANG et al., 2015) o qual também é utilizado pelo OPF. Tal uso abrange, mas não se limita a MST (*Minimum Spanning Tree*) e Djikstra.

Embora esse desenvolvimento seja um ponto de grande interesse para o OPF, como grande parte do trabalho foi focado na vetorização, e portanto, não foi possível conduzir toda a ideia original de reimplementação do OPF. Assim, este trecho do estudo foca apenas na análise de desempenho de um algoritmo MST, o Boruvka, assim como verificar a viabilidade de implementação no POPF.

Os testes são realizados utilizando o algoritmo de Boruvka que, assim como Prim que é utilizado no OPF, por meio de um grafo conectado e não direcionado, é capaz de encontrar uma árvore geradora mínima, de modo a fazer um estudo inicial e verificação de desempenho do algoritmo utilizando Galois por múltiplos *threads*.

6.1 Metodologia

Os testes foram realizados utilizam uma versão do algoritmo de cálculo de árvore geradora mínima Boruvka integrada ao sistema Galois. Os *datasets* utilizados foram o R-MAT (CHAKRABARTI; ZHAN; FALOUTSOS, 2004) e o Epinions (LESKOVEC; KREVL, 2014).

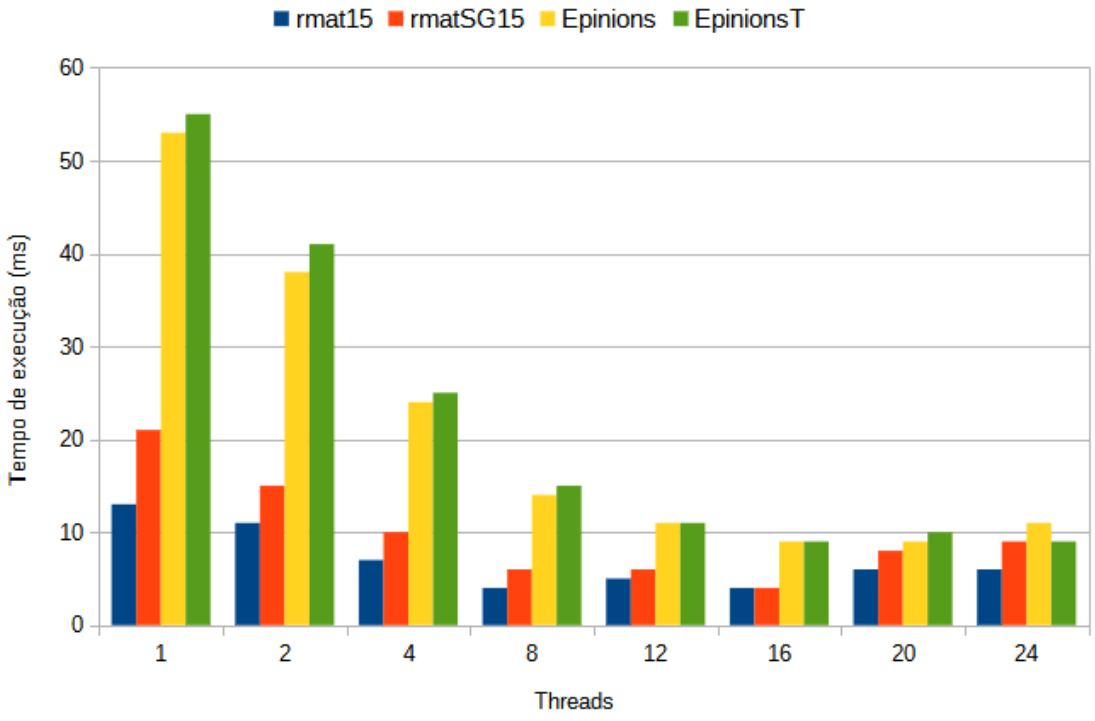
Os valores apresentados nesta seção foram obtidos utilizando as ferramentas do próprio Galois e são baseados na média de 3 execuções realizadas para cada configuração apresentada. Os dados consistem apenas nos valores médios devido a pouca variabilidade nos resultados individuais. O *speedup* foi obtido dividindo-se o tempo de execução sequencial de cada configuração pelos tempos com diferentes números de *threads*.

6.2 Resultados

Dentre os *dataset* utilizados tem-se o `rmat15`, que corresponde ao conjunto de dados R-MAT e, seguido pelo `rmatSG15`, sua versão com dados simétricos, em terceiro o *dataset* `Epinions` e por fim, sua versão transposta, representada por `EpinionsT`. Ressalta-se ainda que os *datasets* estão disponíveis como parte do repositório público do Galois (ISS, 2022).

Os resultados obtidos na Figura 11 apresentam que de maneira geral, o Galois consegue ter melhoras de desempenho com até 16 *threads* para todos os quatro *datasets* utilizados. Ressalta-se ainda que os *datasets* não apresentam comportamentos significativos após 24 *threads*, portanto, para fins de visualização, esses resultados não foram apresentados.

Figura 11 – Gráfico de tempo de execução comparando as configurações do algoritmo Boruvka utilizando Galois.



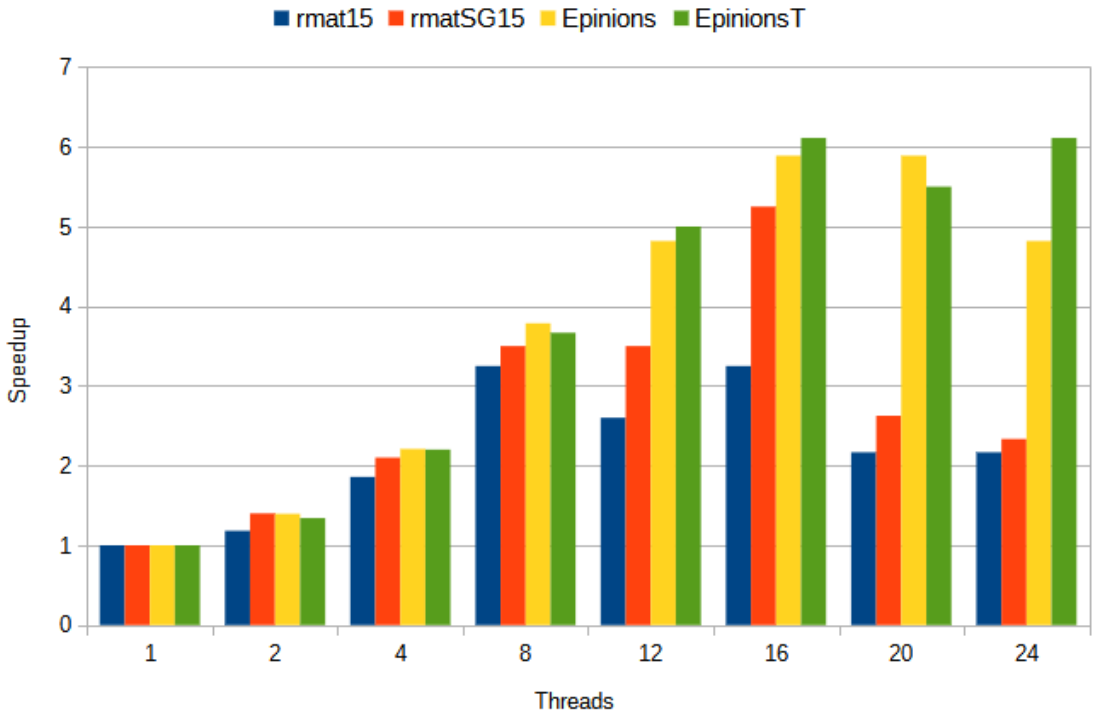
Fonte: Elaborada pelo autor.

Na Figura 12 é possível observar que o *speedup* se mantém bastante similar entre os *datasets* até 8 *threads*, em valores entre 3x e 4x, embora haja melhora de desempenho significativa, os ganhos ficam longe do *speedup* teórico de 8x.

A partir de 12 *threads*, o comportamento dos *datasets* começam a se distinguir. Os relativos ao Epinions (`Epinions` e `EpinionsT`) apresentam um melhor desempenho de maneira geral, atingindo um *speedup* de aproximadamente 6x em ambos os *datasets* com 16 *threads*. Com os *datasets* relativos ao R-MAT (`rmat15` e `rmatSG15`) o *speedup* máximo se mantém com 16 *threads*, mas para o `rmat15` é limitado em 3,25x, demonstrando um pior comportamento do que `rmatSG15`, o qual obteve uma melhora de até 5,25x. Para todos os *datasets* analisados, após o ponto de desempenho máximo, é notado um período curto de estabilidade pelas próximas configurações de *threads*, seguido por uma queda de desempenho, cujos valores se mantém com pouca oscilação até o uso dos 72 *threads* disponíveis no sistema.

Neste contexto, é possível citar estudos relacionados, como o de Mariano, Proença e Sousa (2015), que também utiliza um sistema Intel para os testes que, apesar de mais antigo, obtém resultados de até 6,06x com 8 *threads* com o *dataset* `road-network` utilizando o Galois,

Figura 12 – Gráfico de *speedup* comparando as configurações do algoritmo Boruvka utilizando Galois.



Fonte: Elaborada pelo autor.

um resultado bastante promissor se comparado ao maior valor obtido nessa configuração, com 3,79x aqui observado com o *dataset* `Epinions`.

Portanto, é possível observar que a utilização do Galois é bastante dependente do *dataset*. Mas tendo em vista seu potencial, mesmo apresentando um desempenho subótimo nos experimentos aqui realizados, seria de interesse realizar a implementação no POPF e avaliar as possíveis reduções no tempo de execução.

7 Conclusão

Com o intuito de realizar otimizações para algoritmos de aprendizado de máquina com base em grafos, a ideia de utilizar novas tecnologias de vetorização foi confirmado benéfico quanto ao desempenho do algoritmo POPF aqui utilizado. Para isso, três implementações principais abordando vetorização foram criadas, a do cálculo matricial, do cálculo da distância Euclidiana e, por fim, a implementação propriamente dita no POPF.

A primeira, buscou compreender os ganhos que AVX-512 poderia trazer em uma situação ideal, aqui ilustrada pelo algoritmo de multiplicação matricial, onde ganhos obtidos com 1 *thread* sobre a configuração não vetorial foram de 645% e sobre AVX2, de 92%. Tais resultados apresentaram ganhos significativos e, no caso do desempenho relativo sobre AVX2, próximo do ganho de desempenho máximo teórico de 100%.

A segunda implementação foi realizada com certa simplicidade, verificando os resultados obtidos pelo novo algoritmo vetorizado do cálculo da distância Euclidiana, o que fora confirmado, e serviu como intermediário para a implementação no POPF.

Por fim, a terceira corresponde à implementação no POPF. Os resultados obtidos na Seção 5.4 confirmam que a tecnologia AVX-512 é capaz de trazer benefícios a implementação POPF. Comparando com resultados obtidos com 1 *thread* entre as implementações não vetorizada e AVX-512, foi possível observar ganhos no sistema principal (CascadeLake) entre 20,23% e 64,43%, e ao comparar as implementações AVX-512 e AVX2, ganhos entre 9,05% e 23,53%. A Seção 5.5 complementa a anterior, de modo a apresentar outro sistema com suporte a AVX-512 e verificar os resultados obtidos anteriormente. Mesmo se tratando de um dispositivo portátil e com uma implementação vetorial menos robusta, observou-se resultados bastante positivos. No caso com 1 *thread*, comparando as configurações sem vetorização e AVX-512 foi constatado ganhos de desempenho entre 18,93% e 112,83%, já comparando entre as implementações vetoriais AVX2 e AVX-512, foram observado ganhos entre 4,71% e 26,84%.

Assim, torna-se possível concluir que a implementação encontrou sucesso, com um ganho bastante significativo de desempenho global com a implementação AVX-512.

Por fim, realizou-se um estudo inicial quanto ao arcabouço computacional Galois. O estudo, baseado em testes com *datasets* `R-MAT` e `Epinions` mostra que é capaz de melhorar o desempenho de algoritmos como o Boruvka utilizando até 16 *threads*. Mas ao considerar outros estudos relacionando Galois e algoritmos baseados em grafos, é possível observar que o arcabouço computacional é capaz de produzir resultados bastante próximos do ideal teórico em sistemas multinúcleo. Seria de interesse para trabalhos futuros realizar a implementação do Galois para outros algoritmos de grafos baseados no OPF e analisar os ganhos obtidos.

Referências

ALCORN, P. *Intel Nukes Alder Lake's AVX-512 Support, Now Fuses It Off in Silicon*. 2022. [Online; Acesso em 27 de Junho de 2022]. Disponível em: <https://www.tomshardware.com/news/intel-nukes-alder-lake-avx-512-now-fuses-it-off-in-silicon>.

BOJARSKI, M.; TESTA, D. D.; DWORAKOWSKI, D.; FIRNER, B.; FLEPP, B.; GOYAL, P.; JACKEL, L. D.; MONFORT, M.; MULLER, U.; ZHANG, J.; ZHANG, X.; ZHAO, J.; KAROL, Z. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

CHAKRABARTI, D.; ZHAN, Y.; FALOUTSOS, C. R-mat: A recursive model for graph mining. In: SIAM. *Proceedings of the 2004 SIAM International Conference on Data Mining*. [S.l.], 2004. p. 442–446.

CIREŞAN, D. C.; GIUSTI, A.; GAMBARDELLA, L. M.; SCHMIDHUBER, J. Mitosis detection in breast cancer histology images with deep neural networks. In: SPRINGER. *International conference on medical image computing and computer-assisted intervention*. [S.l.], 2013. p. 411–418.

CORNEA, M. Intel avx-512 instructions and their use in the implementation of math functions. *Intel Corporation*, p. 1–20, 2015.

CULQUICONDOR, A.; BALDASSIN, A.; CASTELO-FERNÁNDEZ, C.; CARVALHO, J. P. de; PAPA, J. P. An efficient parallel implementation for training supervised optimum-path forest classifiers. *Neurocomputing*, Elsevier, v. 393, p. 259–268, 2020.

ESTEVA, A.; CHOU, K.; YEUNG, S.; NAIK, N.; MADANI, A.; MOTTAGHI, A.; LIU, Y.; TOPOL, E.; DEAN, J.; SOCHER, R. Deep learning-enabled medical computer vision. *NPJ digital medicine*, Nature Publishing Group, v. 4, n. 1, p. 1–9, 2021.

FAISAL, M.; ZAMZAMI, E.; SUTARMAN. Comparative analysis of inter-centroid k-means performance using euclidean distance, canberra distance and manhattan distance. In: IOP PUBLISHING. *Journal of Physics: Conference Series*. [S.l.], 2020. v. 1566, n. 1, p. 012112.

FLYNN, M. Very high-speed computing systems. *Proceedings of the IEEE*, v. 54, n. 12, p. 1901–1909, 1966.

HERMERDING, J.; DISTEFANO, E.; HILL, T.; SHAH, K. R.; SRINIVASAN, V. Ultrabook™: Doing more with less. In: *2011 International Conference on Energy Aware Computing*. [S.l.: s.n.], 2011. p. 1–4.

HUANG, G.; LIU, Z.; MAATEN, L. V. D.; WEINBERGER, K. Q. Densely connected convolutional networks. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2017. p. 4700–4708.

INTEL. *Intel Intrinsics Guide*. 2022. [Online; Acesso em 13 de Julho de 2022]. Disponível em: <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>.

ISS. *Galois*. GitHub, 2022. [Online; Acesso em 21 de Julho de 2022]. Disponível em: <https://github.com/IntelligentSoftwareSystems/Galois>.

JESTADT, M. *Why Is AVX 512 Useful for RPCS3?* 2022. [Online; Acesso em 03 de Julho de 2022]. Disponível em: <https://whatcookie.github.io/posts/why-is-avx-512-useful-for-rpcs3/>.

KIPF, T. N.; WELLING, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

KUBAT, M. Neural networks: a comprehensive foundation by simon haykin, macmillan, 1994, isbn 0-02-352781-7. *The Knowledge Engineering Review*, Cambridge University Press, v. 13, n. 4, p. 409–412, 1999.

KULKARNI, M.; PINGALI, K.; WALTER, B.; RAMANARAYANAN, G.; BALA, K.; CHEW, L. P. Optimistic parallelism requires abstractions. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2007. (PLDI '07), p. 211–222. ISBN 9781595936332. [Online; Acesso em 20 de Julho de 2022]. Disponível em: <https://doi.org/10.1145/1250734.1250759>.

LESKOVEC, J.; KREVL, A. *SNAP Datasets: Stanford Large Network Dataset Collection*. 2014. [Online; Acesso em 20 de Julho de 2022]. Disponível em: <http://snap.stanford.edu/data>.

LIBÓRIO, A.; BALDASSIN, A. Análise de desempenho do cálculo matricial em sistemas paralelos utilizando openmp. In: SBC. *Anais da XII Escola Regional de Alto Desempenho de São Paulo*. [S.l.], 2021. p. 13–16.

LICHMAN, M. *UCI Machine Learning Repository*. 2013. [Online; Acesso em 18 de Julho de 2022]. Disponível em: <http://archive.ics.uci.edu/ml>.

MARIANO, A.; PROENCA, A.; SOUSA, C. D. S. A generic and highly efficient parallel variant of boruvka's algorithm. In: *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. [S.l.: s.n.], 2015. p. 610–617.

PAPA, J. P.; FALCAO, A. X.; SUZUKI, C. T. Supervised pattern classification based on optimum-path forest. *International Journal of Imaging Systems and Technology*, Wiley Online Library, v. 19, n. 2, p. 120–131, 2009.

RATHORE, Y.; KUMAR, D. Performance evaluation of matrix multiplication using openmp for single dual and multi-core machines. *IOSR Journal of Engineering (IOSRJEN)*, v. 4, p. 56–59, 2014.

SHABANOV, B.; RYBAKOV, A.; SHUMILIN, S. Vectorization of high-performance scientific calculations using avx-512 intruction set. *Lobachevskii Journal of Mathematics*, v. 40, p. 580–598, 05 2019.

SONI, A.; DHARMACHARYA, D.; PAL, A.; SRIVASTAVA, V. K.; SHAW, R. N.; GHOSH, A. Design of a machine learning-based self-driving car. In: *Machine Learning for Robotics Applications*. [S.l.]: Springer, 2021. p. 139–151.

SUNNY Cove: Intel's Lost Generationl. 2022. [Online; Acesso em 15 de Julho de 2022]. Disponível em: <https://chipsandcheese.com/2022/06/07/sunny-cove-intels-lost-generation/>.

SZE, V.; CHEN, Y.-H.; EMER, J.; SULEIMAN, A.; ZHANG, Z. Hardware for machine learning: Challenges and opportunities. In: IEEE. *2017 IEEE Custom Integrated Circuits Conference (CICC)*. [S.l.], 2017. p. 1–8.

THEARLING, K. Massively parallel architectures and algorithms for time series analysis. *Lectures in Complex Systems, Addison-Wesley*, 1996.

TYSON, M. *AMD Ryzen 7000: Up to 16 Cores, AVX-512 Support at Launch*. 2022. [Online; Acesso em 25 de Junho de 2022]. Disponível em: <https://www.tomshardware.com/news/amd-ryzen-7000-zen4-avx512>.

VENU, B. *Multi-core processors - An overview*. arXiv, 2011. [Online; Acesso em 19 de Julho de 2022]. Disponível em: <https://arxiv.org/abs/1110.3535>.

WHANG, J. J.; LENHARTH, A.; DHILLON, I. S.; PINGALI, K. Scalable data-driven pagerank: Algorithms, system issues, and lessons learned. In: SPRINGER. *European Conference on Parallel Processing*. [S.l.], 2015. p. 438–450.