Emprego da tecnologia AVX-512 para aceleração do algoritmo POPF

André Libório[1], Alexandro Baldassin[1], João Paulo Papa[1]

[1]Universidade Paulista Júlio de Mesquita Filho (UNESP) - Brasil

{andre.lb.ferraz, alexandro.baldassin, joao.papa}@unesp.br

Abstract. *With the popularization of the AVX-512 vectoring technology in the last decade, it has become attractive to check its performance in new applications. This article presents a study of the use of the AVX-512 technology in a graph-based machine learning algorithm, Parallel Optimum-Path Forest (POPF). The experiments conducted show a performance gain of up to 64% compared to the original, unvectorized, version and up to 23% against AVX2. The performance gains were more discrete in scenarios where multithreading is also employed, but the AVX-512 still displayed the best results overall.*

Resumo. *Com a popularização da tecnologia de vetorização AVX-512 na última década, tornou-se interessante verificar seu desempenho em novas aplicações. Este artigo apresenta um estudo sobre o uso da tecnologia AVX-512 em um algoritmo de aprendizado de máquina baseado em grafos, o Parallel Optimum-Path Forest (POPF). Os experimentos conduzidos mostram um ganho de desempenho de até 64% em relação à versão original, sem vetorização, e até 23% em relação ao AVX2. Os ganhos em desempenho foram mais discretos em cenários em que multithreading também é utilizado, mas mesmo assim a versão com AVX-512 atingiu os melhores resultados no geral.*

1. Introdução

Praticamente todos os processadores de propósito geral atuais possuem algum suporte para vetorização. Esta característica permite que certas aplicações possam ser otimizadas ao utilizarem o processamento do tipo *Single-Instruction Multiple Data* (SIMD) [Kretz 2015]. A Intel começou a adicionar instruções vetoriais a partir da linha Pentium, inicialmente com a tecnologia *MultiMedia EXtensions* (MMX), depois *Streaming SIMD Extensions* (SSE) e, mais recentemente, o *Advanced Vector Extensions* (AVX). A última versão, lançada em 2016 com a arquitetura *Knights Landing*, é conhecida como AVX-512. Estas instruções permitem o cálculo com operandos de até 64 bytes.

O uso eficiente da tecnologia AVX-512 depende do *workload* e portanto o ganho de desempenho final nem sempre corresponde às expectativas. Em um trabalho anterior [Culquicondor et al. 2020], a vetorização com o AVX2 proporcionou ganhos significativos para alguns *workloads* processados com o classificador *Optimum Path Forest* OPF [Papa et al. 2009]. Neste artigo, pretende-se avaliar a eficiência do AVX-512 no mesmo contexto[1] e realizar uma comparação com o AVX2. Os resultados apresentam ganhos significativos nos casos com *datasets* que se beneficiam dessa nova tecnologia de vetorização.

[1]Disponível em: https://github.com/andreliborio98/popfavx512

Este trabalho se inicia com um resumo dos principais conceitos sobre vetorização e algoritmos de aprendizado de máquina, como o classificador OPF na Seção 2, seguida por uma análise do cálculo da distância utilizado no OPF e da implementação em código na Seção 3, a metodologia na Seção 4, a avaliação experimental na Seção 5 e, por fim, a conclusão na Seção 6.

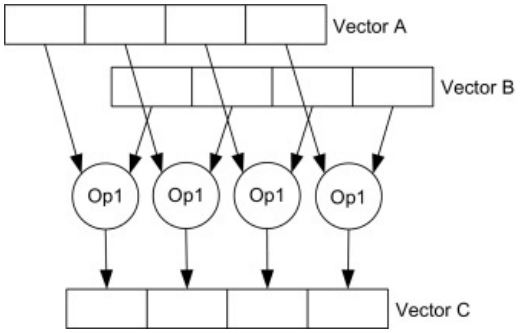
2. Contextualização

Esta seção apresenta os dois conceitos principais explorados neste artigo: a vetorização e o aprendizado de máquina.

2.1. Vetorização

Mesmo que o constante incremento de núcleos presentes em processadores seja evidentemente o principal fator de melhora de desempenho, outras tecnologias foram desenvolvidas e incorporadas. Uma delas é o suporte para vetorização, comumente conhecida como SIMD (Single Instruction/Multiple Data), advindo da taxonomia de Flynn [Flynn 1966], cujo mecanismo básico de operações é apresentado na Figura 1. 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 uma unidade SIMD, neste caso, realizando um exemplo com apenas um operador aplicado a quatro operandos distintos em paralelo.



Fonte: [Cardoso et al. 2017]

Instruções SIMD estão disponíveis em processadores Intel desde a criação do conjunto de instruções MMX em 1997, com seus registradores de 64-bits dedicados. Hoje, processadores como o Intel Xeon Gold 5220, utilizado para fins experimentais neste trabalho, possuem além do suporte a registradores MMX, suporte a tecnologias mais recentes, cronologicamente, SSE, com registradores de 128-bits, AVX, também chamado de 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.

2.2. 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 1996]. Somente com alto poder computacional passou a ser viável cogitar a utilização de técnicas de aprendizado de máquina já existentes com dados mais robustos 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] sejam utilizados, a tecnologia evoluiu, criando mecanismos sofisticados 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].

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 et al. 2009] e aprendizado de máquina profundo [Kipf and Welling 2016]. Tais mecanismos de grafos aplicados ao aprendizado de máquina passam a ser interessantes, uma vez que suas aplicações são 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. [Culquicondor et al. 2020] desenvolveu uma paralelização do OPF, por meio de ferramentas como OpenMP e técnicas de vetorização, denominada Parallel OPF (POPF).

No OPF, 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. Essa função é reconhecidamente uma das partes do algoritmo que demanda maior tempo de computação. Logo, sua otimização é de extrema relevância. O trabalho de [Culquicondor et al. 2020] desenvolveu uma técnica de vetorização para o cálculo de distância usando o AVX2. O objetivo deste artigo é realizar a vetorização com o AVX-512 e fazer uma análise de seu ganho.

3. Cálculo da distância

Este trabalho busca realizar a implementação da tecnologia AVX-512 no algoritmo POPF, de maneira a verificar o desempenho e compará-lo a implementações anteriores. Há várias formas de calcular a distância entre os vetores no POPF. A mais utilizada é o logaritmo da distância Euclidiana, apresentada pela Equação 1.

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

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 soma realizada no final da Equação 1 é uma operação sobre vetores e pode se beneficiar do hardware vetorial.

Para a implementação da vetorização usando o AVX-512 foram utilizadas as operações *intrinsics* do compilador GCC, que permitem especificar as instruções AVX-512 diretamente. O pseudo-código relativo à operação principal do cálculo da distância é mostrado na Listagem 1. O laço das linhas 4–11 utiliza vetores de 512 bits para realizar a acumulação das diferenças entre os vetores (última parte da Equação 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. O restante do pseudo-código, representado entre as linhas 12–23, faz a redução dos valores presentes no vetor de 512 bits que registra o valor acumulado (`vetAccAvx`) para um valor de 32 bits (`dist`), que é então retornado pela função do POPF. 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 13 e 14), então é feita a soma desses dois vetores (linha 15). 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 16). É feita a soma desses dois vetores de 256 bits (linha 17), fazendo com que o vetor de 256 bits se comporte como dois vetores de 128 bits repetidos. Então, são feitas somas horizontais, representada pela *intrinsic* `hadd` (linhas 18–19); 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 22) e o primeiro desses valores é utilizado para incrementar a distância (dist), que é o valor de saída da função.

Listing 1. Implementação do cálculo da distância euclidiana no OPF usando AVX-512

```
1  Entrada: vetor1, vetor2, tamanhoFeatureDataset
2  Saida: dist
3  inicio
4      for (i=0; i <= tamanhoFeatureDataset-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
```

Uma dificuldade encontrada na implementação do código deveu-se ao alinhamento de memória necessária para a realização das operações vetoriais com o AVX-512. Assim, foi necessário modificar o POPF para que as alocações de memórias dos vetores de características estivessem alinhados em páginas de 64 bytes. Para isso, utilizou-se a chamada POSIX `memalign`. Uma outra alternativa considerada foi utilizar a *intrinsic loadu*, que realiza 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.

4. Metodologia experimental

O sistema utilizado para a execução dos experimentos realizados 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 nível de otimização -O3 para fins de otimização de compilação, a *flag* para o uso das instruções AVX2 e AVX-512 e 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 and Kumar 2014]. Os valores apresentados neste estudo são baseados na média de 5 execuções realizadas para cada configuração apresentada, com valores de ponto flutuante (*float*), com intervalos de confiança de 95%.

Para realizar os testes foram utilizados até 24 *threads* para os experimentos, uma vez que não existe ganho de desempenho aparente com um número maior de *threads*. As bases de dados, ou *datasets*, aqui utilizadas são as mesmas do artigo referente ao POPF [Culquicondor et al. 2020] para fins de validação dos resultados [Lichman 2013].

5. Resultados experimentais

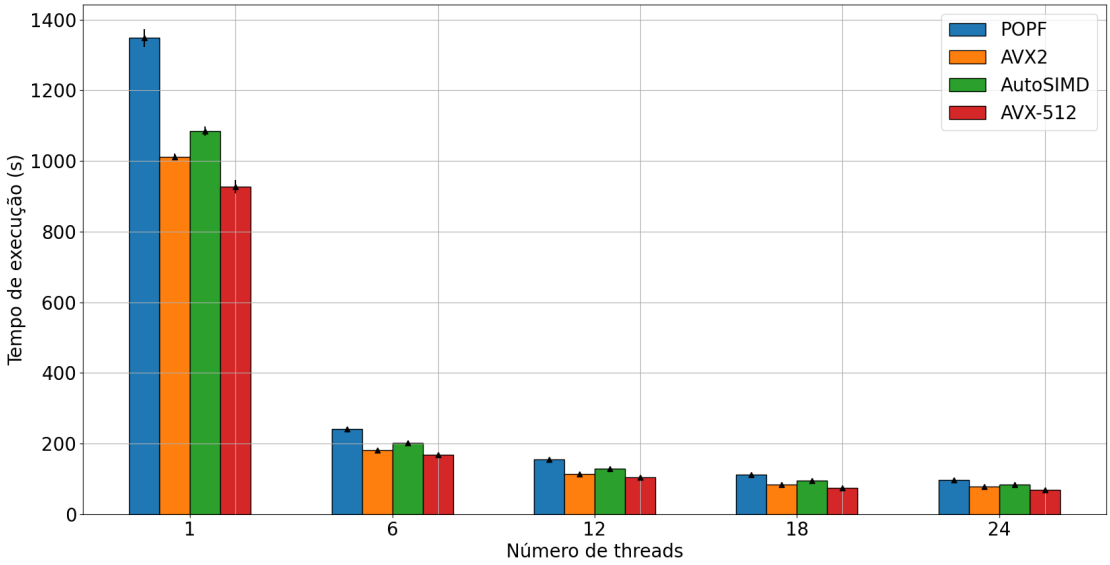
O primeiro conjunto de experimentos procurou quantificar somente o ganho com a vetorização. Para isso, o POPF foi executado de forma sequencial e mediu-se o tempo (sem vetorização), o tempo com AVX2 e também com AVX-512. Os resultados estão mostrados na Tabela 1. A primeira coluna contém o nome do *dataset*, a segunda o número de características de cada vetor e então os tempos sequencial, AVX2 e AVX-512. As últimas colunas apresentam os ganhos com a vetorização AVX-512 comparada ao sequencial e AVX2. Apenas os *datasets* com pelo menos 16 características são utilizados, dado que um valor menor que este não poderia ser usado com o AVX-512.

Como esperado, o ganho obtido é proporcional ao número de características. No `SDD`, a operação do cálculo de distância pode ser realizada com três operações vetorais ($3*16$). 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 calculado de forma sequencial. Finalmente, o dataset `Letter` apresenta ganhos mais modestos em relação ao sequencial. Com o AVX2, na verdade, há uma leve perda devido ao *overhead* necessário para preparar os operandos para a vetorização.

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

Dataset	Caract.	Sequencial	AVX2	AVX-512	Seq/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%

Figura 2. Tempo de execução com o `MiniBooNE`.



O segundo conjunto de experimentos mostra o comportamento das execuções conforme o número de *threads* é aumentado. Para esses experimentos, além da versão original (`POPF`), do AVX2 e do AVX-512, também é usado uma versão obtida com a auto-vetorização por meio da cláusula `simd` do OpenMP (`AutoSIMD`). Os resultados para os *datasets* apresentados na Tabela 1 estão ilustrados nas Figuras 2, 3 e 4.

Primeiramente, nota-se que o ganho com a vetorização é menos relevante conforme mais *threads* são utilizadas. Isso 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 2. 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.

6. Conclusão

Este trabalho apresentou um estudo de vetorização por meio da tecnologia AVX-512 aplicada ao módulo de cálculo da distância do algoritmo POPF. Foi apresentada uma implementação utilizando-se de operações com *intrinsics* do AVX-512 e o desempenho comparado com a versão original (sem vetorização) e AVX2 foi analisado. Como resultado, notou-se um ganho de até 64% em relação à versão original e de 23% em relação ao AVX2 em um cenário com apenas uma thread. Em um ambiente *multithreading*, o ga-

Figura 3. Tempo de execução com o `SDD`.

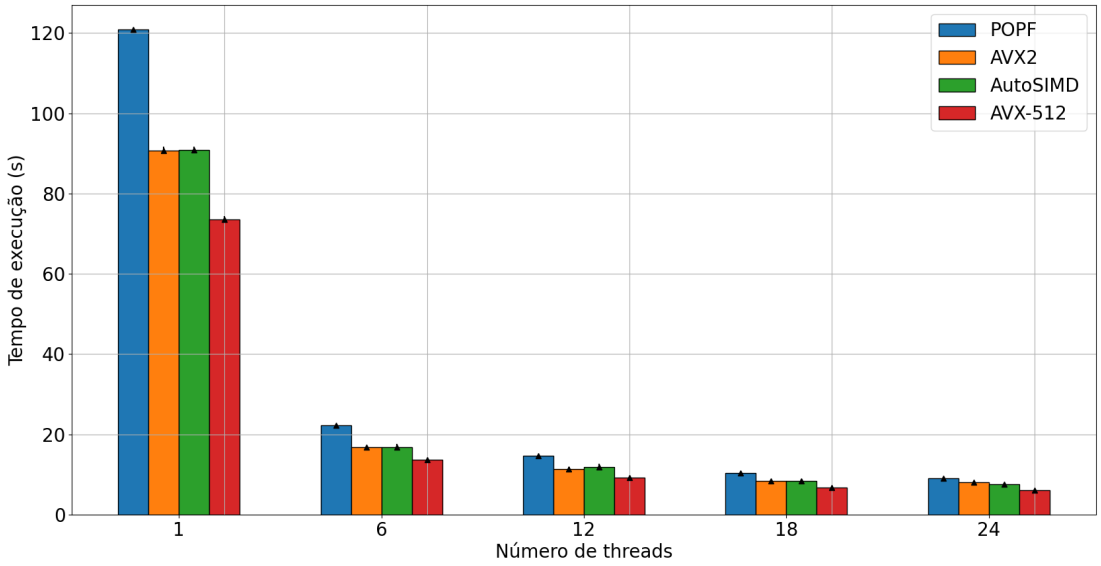
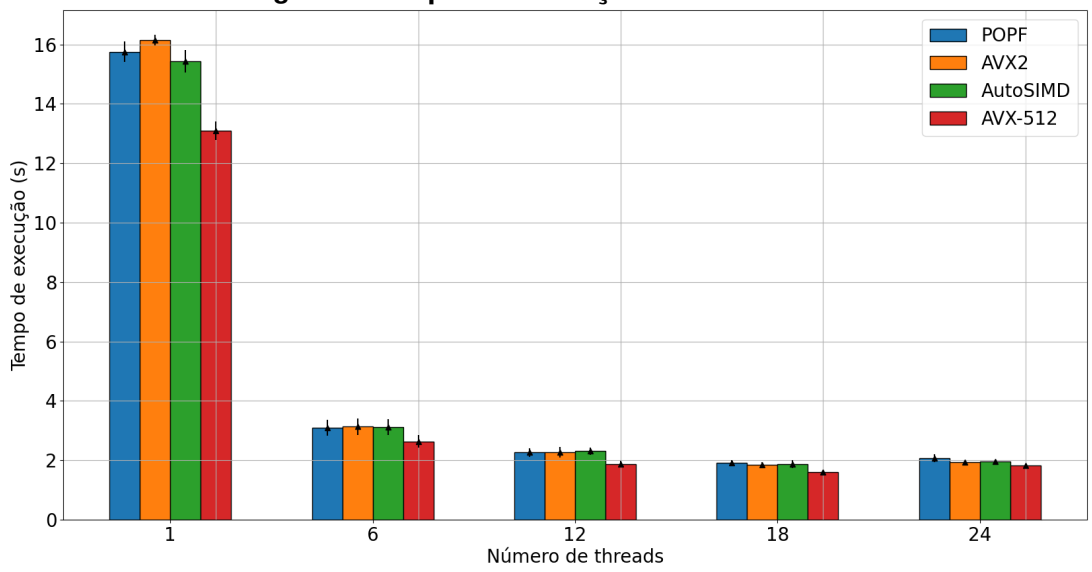


Figura 4. Tempo de execução com o `Letter`.



nho relativo é menor mas mesmo assim a vetorização com AVX-512 apresentou o melhor resultado no geral.

Agradecimentos. Os autores agradecem à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processo nº 2018/15519-5.

Referências

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., et al. (2016). End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*.

Cardoso, J. M., Coutinho, J. G. F., and Diniz, P. C. (2017). Chapter 2 - high-performance embedded computing. In Cardoso, J. M., Coutinho, J. G. F., and Diniz, P. C., editors, *Embedded Computing for High Performance*, pages 17–56. Morgan Kaufmann, Boston.

Cireşan, D. C., Giusti, A., Gambardella, L. M., and Schmidhuber, J. (2013). Mitosis detection in breast cancer histology images with deep neural networks. In *International conference on medical image computing and computer-assisted intervention*, pages 411–418. Springer.

Culquicondor, A., Baldassin, A., Castelo-Fernández, C., de Carvalho, J. P., and Papa, J. P. (2020). An efficient parallel implementation for training supervised optimum-path forest classifiers. *Neurocomputing*, 393:259–268.

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

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708.

Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Kretz, M. (2015). *Extending C++ for explicit data-parallel programming via SIMD vector types*. PhD thesis, Frankfurt am Main, Johann Wolfgang Goethe-Univ., Diss., 2015.

Kubat, M. (1999). Neural networks: a comprehensive foundation by simon haykin, macmillan, 1994, isbn 0-02-352781-7. *The Knowledge Engineering Review*, 13(4):409–412.

Lichman, M. (2013). UCI machine learning repository.

Papa, J. P., Falcao, A. X., and Suzuki, C. T. (2009). Supervised pattern classification based on optimum-path forest. *International Journal of Imaging Systems and Technology*, 19(2):120–131.

Rathore, Y. and Kumar, D. (2014). Performance evaluation of matrix multiplication using openmp for single dual and multi-core machines. *IOSR Journal of Engineering (IOSRJEN)*, 4:56–59.

Sze, V., Chen, Y.-H., Emer, J., Suleiman, A., and Zhang, Z. (2017). Hardware for machine learning: Challenges and opportunities. In *2017 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8. IEEE.

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