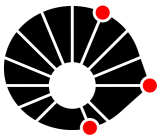

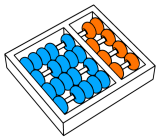

Revisão do Estado da Arte da Linearização de Vetores Matriciais

C. Carneiro

Relatório Técnico - IC-PFG-23-18
Projeto Final de Graduação
2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Revisão do Estado da Arte da Linearização de Vetores Matriciais

César Guedes Carneiro

Dezembro 2023

Resumo

Acesso à tensores multi-dimensionais representam uma parcela significativa do tempo de execução de um programa, particularmente em programas científicos nas áreas de *High-Performace Computing* (HPC) e *Machine Learning* (ML), onde o tamanho dos tensores é muito grande. Linearizar tensores de modo a torná-los unidimensionais é reconhecidamente uma técnica poderosa para aumentar a localidade de seus elementos na hierarquia da memória, reduzindo assim a latência do seu acesso. Porém, a linearização é uma técnica muito pouco usada historicamente, sendo sua aplicação restrita a implementações manuais de GEMMs. Isso se dá porque, dado o padrão de acesso dos tensores, nem todas as linearizações produzem uma melhoria de desempenho quando realizada. Recentemente, o estado da arte nesta área vem focando em maneiras de generalizar o uso da linearização, como é o caso do GPAT que faz uma análise da viabilidade em algumas aplicações. Com essas novas ferramentas, é possível realizar a aplicação geral da linearização em tempo de compilação, a fim de melhorar a performance do treino de algoritmos de *Machine Learning* (ML), assim como acelerar a execução de programas em *High-Performace Computing* (HPC).

Introdução

Durante o treinamento de modelos de aprendizado de máquina, são realizadas diversas convoluções, e em cada uma delas um filtro é aplicado a cada canal da entrada. A aplicação desses filtros requer que o algoritmo itere sobre cada canal usando laços, e essa etapa representa uma parte significativa do tempo de treinamento do modelo. Para otimizar esse processamento, é necessário transformar a entrada de forma a reduzir o número de acessos à memória. As transformações mais amplamente estudadas atualmente são a fragmentação (*Tiling*), que divide vetores grandes em blocos menores que se encaixam em cada nível da hierarquia da memória, e a linearização (*Packing*), que cria uma cópia desses blocos enquanto reorganiza seus elementos na ordem em que são utilizados nas computações, resultando em maior localidade e aumentando a probabilidade de que os dados acessados subsequentemente pertençam à mesma página de memória. A fragmentação é amplamente utilizada em compiladores atuais. No entanto, a linearização ainda é utilizada apenas em aplicações manuais de Multiplicação Geral de Matrizes (GEMM) [1], devido à sua maior complexidade de implementação e à dificuldade de avaliar os possíveis ganhos.

O propósito central deste trabalho consiste em explorar a abordagem da linearização, compreendendo seu funcionamento, benefícios, e perspectivas futuras. Nesse contexto, realizaremos uma abrangente revisão bibliográfica, examinando uma variedade de artigos que abordam a linearização, além de investigar outras estratégias correlatas, como a fragmentação.

Conforme anteriormente discutido, a linearização comumente se restringe a implementações manuais de GEMM. Com esse contexto em mente, empregaremos um exemplo específico de linearização aplicada em uma operação GEMM para elucidar seu funcionamento e analisar seu impacto na performance. Este exemplo foi extraído do trabalho de Rohwedder et al. [2].

Algoritmo 1 Exemplo de GEMM

```
1:
2: for i ← 0 até 49 do                          // A_{80 × 100 × 50}
3:     for j ← 0 até 59 do                      // A_{80 × 100 × 1}
4:         for k ← 0 até 79 do                  // A_{1 × 100 × 1}
5:             for l ← 0 até 99 do              // A_{1 × 1 × 1}
6:                 a ← load A[k][l][i]
7:                 b ← load B[l][k][j]
8:                 prod ← mul a, b
9:                 c ← load C[i][j]
10:                sum ← add c, prod
11:                store sum, C[i][j]
12:            end for
13:        end for
14:    end for
15: end for
```

O Algoritmo 1 contém uma implementação de uma GEMM genérica, onde os co-

mentários à direita indicam a forma do conjunto de trabalho de A fora do aninhamento do loop e dentro de cada loop, ou seja, indica quais laços iteram sobre A e como essa iteração é feita.

Algoritmo 2 Exemplo de GEMM com uso da Linearização

```
1: for i ← 0 até 49 do
2:     A' ← alloc(1 × 80 × 100)                    // A'1 × 80 × 100
3:     for m ← 0 até 79 do
4:         for n ← 0 até 99 do
5:             tmp ← load A[m][n][i]
6:             store tmp, A'[0][m][n]
7:         end for
8:     end for
9:     for j ← 0 até 59 do                         // A'1 × 80 × 100
10:        for k ← 0 até 79 do                     // A'1 × 1 × 100
11:            for l ← 0 até 99 do                 // A'1 × 1 × 1
12:                a ← load A'[0][k][l]
13:                b ← load B[l][k][j]
14:                prod ← mul a, b
15:                c ← load C[i][j]
16:                sum ← add c, prod
17:                store sum, C[i][j]
18:            end for
19:        end for
20:    end for
21: end for
```

O Algoritmo 2 mostra a linearização aplicada no problema do Algoritmo 1, sem o uso da fragmentação. À direita é mostrado o conjunto de trabalho de A'. Os tensores de entrada são $A_{80 \times 100 \times 50}$, $B_{100 \times 80 \times 60}$ e $C_{50 \times 60}$, onde C foi inicializado com zeros. Podemos ver que do Algoritmo 1 para o 2 superficialmente não ocorreram mudanças de performance - a complexidade continua igual. Isso se dá pelo fato de que o ganho de performance da linearização vem nos acessos aos dados dos tensores.

A leituras dos dados, feitas nas linhas 6, 7 e 9 do Algoritmo 1, envolvem a tradução de endereços virtuais para endereços físicos, que é realizada por meio de uma tabela de páginas. Essa tabela, contendo informações sobre a localização dos dados na memória principal, pode levar a um aumento no tempo de acesso devido à necessidade de duas etapas: uma para obter o endereço físico e outra para acessar os dados. Para otimizar o desempenho do acesso à memória, muitos sistemas fazem o uso do *Translation-Lookaside Buffer (TLB)*, uma cache especializada que armazena traduções recentes de endereços virtuais para endereços físicos. O TLB atua como um mecanismo de armazenamento temporário dessas traduções, evitando consultas frequentes à tabela de páginas na memória principal.

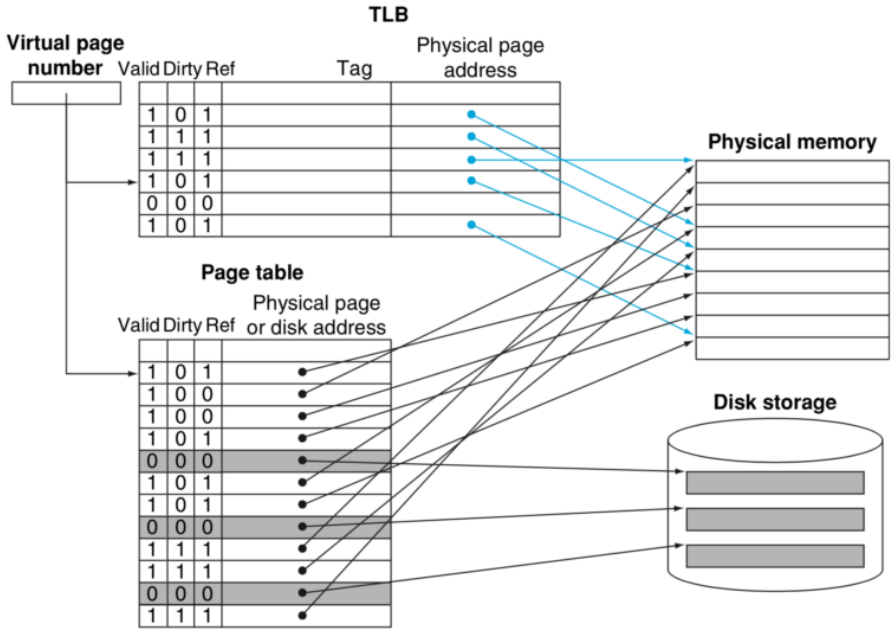

Figura 1: O TLB atua como uma cache da tabela de páginas apenas para as entradas que fazem o mapeamento para páginas físicas. Adaptado de [3].

Quando um programa acessa um valor em um tensor, o processador verifica se a tradução necessária está presente no TLB. Caso encontre, ocorre um "TLB hit,"acelerando o acesso aos dados. Por outro lado, se a tradução não estiver no TLB, ocorre um "TLB miss,"e o processador precisa buscar a tradução na tabela de páginas, adicionando uma etapa extra ao acesso aos dados. A importância de reduzir "TLB misses"está relacionada à exploração da localidade de referência, conceito que sugere que, uma vez traduzido um endereço virtual e acessados os dados, é provável que esses dados sejam necessários novamente em breve. Portanto, ao armazenar traduções recentes no TLB, a probabilidade de "TLB misses"é reduzida, melhorando a eficiência no acesso aos dados do tensor e, por conseguinte, a performance do algoritmo. E é nesse contexto que se da a melhora de performance da linearização, ao reorganizar os elementos dos blocos criados pela fragmentação na ordem em que serão utilizados, aumenta-se consideravelmente a localidade.

Fundamentação Teórica

Bondhugula et al.[4] explora a geração de código de alto desempenho em MLIR, apresentando um estudo de caso inicial com GEMM. O estudo concentra-se em aspectos como memrefs, o dialeto affine e utilitários poliédricos, destacando o papel da infraestrutura do compilador na geração de código competitivo com bibliotecas desenvolvidas manualmente. A exploração se estende ao impacto de várias técnicas de otimização no desempenho da arquitetura de computadores, com um foco específico em padrões de acesso à memória e layouts de referência de memória.

O artigo se aprofunda em técnicas destinadas a aprimorar o desempenho de códigos envolvendo arrays multidimensionais em Computação de Alto Desempenho (HPC), com um foco principal no caso específico da multiplicação de matrizes. Uma técnica crucial destacada é a linearização, envolvendo uma transformação na representação do tensor original, criando um novo tensor com elementos reorganizados em buffers contínuos. Essa representação compactada permite um acesso mais rápido aos elementos do tensor, reduzindo conflitos de cache, "TLB misses", e aprimorando a eficiência do prefetching de hardware.

Juntamente com outro método, a fragmentação [1], essa técnica proporciona benefícios adicionais para a multiplicação de matrizes, como a melhoria na utilização de cache, a redução do tempo de execução e, consequentemente, o aprimoramento do desempenho do código. Os autores demonstram que desabilitar a linearização durante o processo de unroll-and-jam leva a uma perda significativa de desempenho. Eles comparam o desempenho com e sem a linearização, mostrando que a linearização é crucial para alcançar alto desempenho.

O artigo esclarece a aplicação prática dessas técnicas e apresenta resultados de testes comparando o desempenho de códigos com e sem essas otimizações. Por fim, o estudo conclui que a aplicação da linearização e da fragmentação pode melhorar significativamente o desempenho de programas em HPC que utilizam a multiplicação de matrizes. Ao abordar sistematicamente layouts de referência de memória, layouts de mapas afins e otimização de padrões de acesso à memória, o artigo destaca a importância dessas técnicas na obtenção de uma multiplicação de matrizes eficiente, contribuindo assim para avanços nas aplicações de Computação de Alto Desempenho.

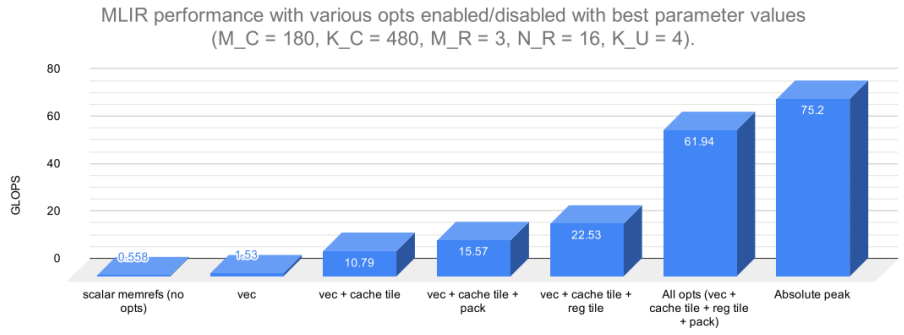

Figura 2: Performance de um algoritmo de GEMM com diferentes modificações no compilador MLIR. Adaptado de [4].

Lam et al. [5] discute várias técnicas para melhorar a performance, com foco em cache e hierarquia de memória. Ela mostra que o desempenho do cache é altamente dependente do tamanho do problema e do tamanho do bloco. Além disso, ela sugere que o uso da técnica de fragmentação é uma maneira eficaz de otimizar a performance de algoritmos, reduzindo a latência média de acesso à memória e o número de referências feitas a níveis mais lentos da hierarquia de memória.

Nesse estudo, é mostrado como aumentar o tamanho dos blocos pode melhorar o

desempenho com o aumento do número de reutilizações dos blocos na cache, no entanto, o aumento excessivo do tamanho do bloco pode levar a um aumento no número de faltas de cache devido ao aumento do número de blocos de dados que não possam ser armazenados na cache. Dessa forma, a escolha do tamanho de bloco é uma decisão crítica que depende da capacidade da cache e das características do hardware subjacente. A fixação de tamanhos de bloco maiores que o tamanho da cache ou menor que o tamanho de um único elemento de matriz pode prejudicar o desempenho. Um tamanho ideal do bloco pode permitir que o algoritmo maximize a reutilização dos dados armazenados na cache e, assim, minimizar o número de faltas de cache.

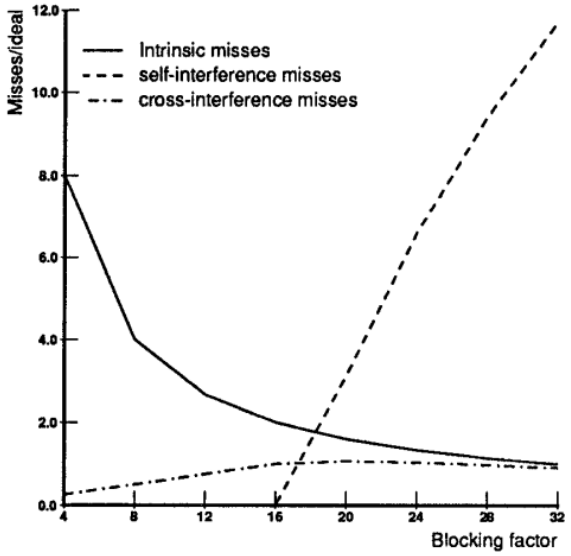

Figura 3: Estimativa de erros por número de blocos. Adaptado de [5].

Outra abordagem discutida no artigo é a reordenação dos blocos. Uma técnica eficaz para melhorar o desempenho do cache, feita para eliminar a interferência, ou seja, para evitar que dados de diferentes matrizes atrapalhem uns aos outros enquanto são armazenados na cache. As técnicas de reordenação incluem a mudança da ordem de acesso aos elementos da matriz, bem como a transposição da matriz. Em muitos casos, a reordenação dos blocos em si não é suficiente para reduzir o número de misses de cache, especialmente quando há reuso de blocos em diferentes partes da matriz. Por isso, também pode ser utilizada a técnica de cópia dos blocos [6] quando o mesmo é reutilizado várias vezes. A cópia de um bloco de dados pode ser usada para minimizar o número de misses de cache e, assim, melhorar o desempenho do algoritmo. No entanto, a utilização da técnica de cópia dos blocos pode ter desvantagens em relação aos custos de "overhead"associados à cópia, especialmente quando o tamanho do bloco é grande em relação à capacidade da memória cache. Além disso, a cópia dos blocos pode ser inadequada em situações em que o bloco reutilizado se move constantemente pela matriz e não ocupa um local fixo.

O estudo realizado por Temam et al. [7] apresenta que o custo/benefício da cópia de dados deve ser cuidadosamente analisado, pois nem sempre o "overhead"gerado pela cópia

é compensado pela melhoria na performance. O texto indica que, considerando tanto os custos quanto os benefícios da cópia, nem sempre vale a pena copiar tudo, já que o custo pode superar os benefícios. Neste sentido, o artigo propõe uma abordagem em tempo de compilação chamada "selective copying", que visa determinar quais e quando copiar os dados, analisando os conflitos da cache. Essa estratégia é baseada em uma análise cuidadosa dos custos e benefícios de copiar os dados em cada caso específico, levando em conta as interferências que ocorrem na cache.

Também se tratando de copia, Esseghir et al. [8] propõe a otimização de cópias de dados em programas como uma técnica para reduzir interferências na cache. Essa técnica utiliza uma combinação de dois métodos conhecidos: o particionamento do espaço de iteração do programa e a otimização de cópias de dados. O particionamento de espaço de iteração divide o espaço de iteração do código em subespaços menores, enquanto a otimização de cópias é responsável por fazer cópias de dados para reduzir a interferência na cache. Em conjunto, essas técnicas podem reduzir a quantidade de falhas de cache, melhorando a eficiência do programa.

O artigo feito por Goto el al. [1] discute a técnica de linearização como uma maneira de melhorar o desempenho de GEMMs. O linearização pode ser usada para rearranjar os dados das matrizes A e B de tal forma que os dados usados em operações consecutivas sejam contíguos na memória, aumentando assim a eficiência do acesso à memória e o desempenho da GEMM. No entanto, o custo do empacotamento deve ser cuidadosamente orquestrado para que valha a pena. O artigo detalha como implementar o linearização em duas variantes da GEMM (gepp e gepm) e discute suas implicações em termos de uso da TLB e tamanho do cache L2. Além disso, o linearização pode ser usada em conjunto com a fragmentação, já que a fragmentação divide as matrizes originais em submatrizes menores e o linearização pode ser usada para rearranjar os dados dessas submatrizes em buffers que permanecem no cache L2 ou na TLB, melhorando o desempenho da GEMM.

LoopStack [9] é um compilador específico para operações com tensores, que permite compilar redes neurais inteiras e gerar código para diversos conjuntos de instruções, incorporando otimizações frequentemente ausentes em outros compiladores de aprendizado de máquina. Esse compilador possui otimizações manuais e automáticas. As otimizações manuais consistem em estratégias de programação de loop bem estabelecidas, como variação de ordem de loop, desenrolamento de loop, fragmentação dos dados e a linearização. A etapa de ajuste manual do LoopStack usa uma interface de linha de comando baseada em texto que permite ao usuário definir o tamanho dos loops e outras opções de otimização.

Por outro lado, as otimizações automáticas do LoopStack são divididas em duas categorias: otimização baseada em IA e otimização baseada em heurística. A otimização baseada em IA é um processo de aprendizado de máquina que ajusta automaticamente as estratégias de programação de loop com base em uma análise do desempenho de iterações anteriores. A otimização baseada em heurística usa métodos heurísticos para realizar ajustes automatizados, como identificar e simplificar cargas independentes, dividir loops aninhados e reescrever instruções em forma matricial. Ambos os métodos de otimização

automática são baseados em uma combinação de técnicas de busca local e global.

O LoopStack usa a linearização para otimizar o uso de registradores de processador, reduzindo a quantidade de acesso à memória global. Isso é feito pela manipulação de matrizes e tensores de entrada, combinando múltiplos elementos em um único registro e realizando operações em lote nesses registros. A linearização também é usada para combinar vários loops em uma única iteração, elevando um lote inteiro de elementos de entrada para cache ou registradores. Porém, o tamanho e os parâmetros da linearização devem ser personalizados manualmente para otimizar o desempenho em cenários específicos.

Grosser et al. [10] discute o uso do Polly, um otimizador poliédrico, para melhorar o desempenho de representações intermediárias de baixo nível. Ele se concentra em técnicas como modelagem de funções de acesso a dados exatas, representação de uma parte de controle estática (SCoP) usando uma descrição poliédrica baseada em conjuntos e mapas inteiros fornecidos pela biblioteca `ISL` e utilização da evolução escalar (SCEV) para descrever o deslocamento do endereço. Além disso, o artigo menciona o uso do algoritmo PLuTo para maximizar a localidade de dados e a capacidade de fragmentação para código sequencial. Ele também destaca o potencial do Polly em otimizar representações intermediárias de baixo nível para melhorias de desempenho.

O artigo demonstra a eficácia do Polly e do PLuTo em melhorar o desempenho. O uso do algoritmo PLuTo para tiling contribui para melhorias de desempenho maximizando a localidade de dados e a capacidade de fragmentação para código sequencial. O artigo apresenta comparações de aceleração de pollycc com e sem fragmentação, mostrando melhorias significativas de desempenho para vários benchmarks, mesmo em tamanhos de dados pequenos. Quanto a vetorização, o artigo discute o impacto da geração de código SIMD no desempenho, indicando que incluir SIMDização no agendamento poliédrico poderia melhorar ainda mais o desempenho em certos casos. Em geral, o artigo destaca a importância das técnicas de fragmentação e vetorização, facilitadas pelo Polly e PLuTo, em melhorar o desempenho de representações intermediárias de baixo nível.

Rohwedder et al. [2] discute uma técnica chamada GPAT, que visa otimizar os padrões de acesso à memória em programas de computador por meio do empacotamento de estruturas de dados. A técnica envolve várias fases, incluindo a identificação de candidatos a empacotamento, a estimativa do impacto do empacotamento nas entradas da tabela de tradução (TLB) e a realização de uma análise de custo-benefício para cada candidato. O benefício de um candidato é determinado pela redução no número de entradas da TLB, enquanto o custo é estimado com base na sobrecarga de empacotamento e no tamanho da estrutura de dados empacotada. O GPAT classifica os candidatos a empacotamento com base em sua relação benefício-custo, fornecendo um meio de selecionar os mais benéficos. O artigo inclui exemplos e listagens de código para ilustrar os conceitos e o impacto do empacotamento nos padrões de acesso à memória.

O estudo explora os detalhes da técnica GPAT, sua implementação e os fatores envolvidos na seleção dos candidatos a empacotamento ideais para melhorar o desempenho do programa, especialmente no contexto da otimização do acesso à memória. Ele fornece

uma compreensão abrangente dos desafios e considerações no empacotamento de dados para a otimização da hierarquia de memória.

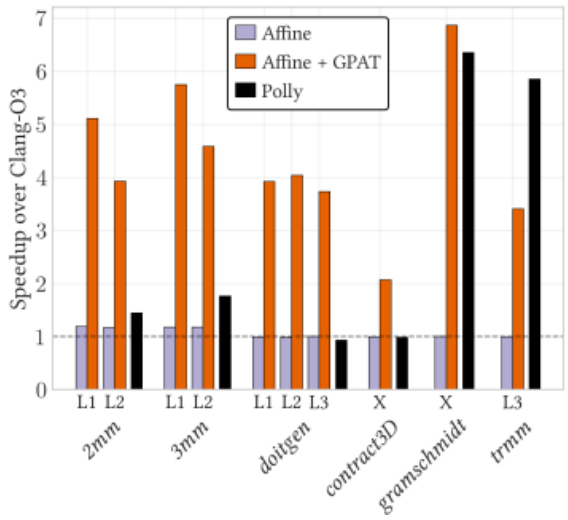

Figura 4: Comparação da performance do Affine, Affine + Gpat, e Polly comparado com Clang-O3 utilizando diferentes niveis de cache. Adaptado de [2].

Os resultados obtidos por Rohwedder são impressionantes. Em todos os modelos utilizados nos testes, que servem como uma sólida referência para os principais modelos de IA, a combinação de Affine + GPAT demonstra uma melhoria significativa no desempenho em comparação com o Clang-O3. Essa melhoria chega a ser mais do que dobrada em muitos casos. Vale ressaltar a notável abrangência do GPAT, que proporciona ganhos de desempenho em todos os modelos testados. Enquanto o Polly mostra ser mais eficaz para o trmm e equivalente ao gramschmidt, não consegue aprimorar consideravelmente o desempenho em outros cenários.

Conclusão

Os artigos mencionados na seção "Fundamentação Teórica"abordam a técnica de linearização de tensores, com foco na otimização do processamento de modelos de aprendizado de máquina, especialmente na multiplicação geral de matrizes (GEMM). A linearização envolve a reorganização dos elementos dos tensores para melhorar a localidade dos dados e, consequentemente, a eficiência no acesso à memória. Os estudos destacam a importância da linearização e sua aplicação prática, demonstrando que a técnica pode significativamente melhorar o desempenho de programas em HPC que utilizam a multiplicação de matrizes. Além disso, ressaltam a relevância da linearização e da fragmentação na obtenção de uma multiplicação de matrizes eficiente, contribuindo para avanços nas aplicações de Computação de Alto Desempenho. Os resultados apresentados evidenciam a importância dessas técnicas na otimização do desempenho de algoritmos, especialmente em relação à hierarquia de memória e ao acesso aos dados.

Os artigos revisados fornecem uma visão abrangente e fundamentada sobre a linearização de tensores e seu impacto na eficiência computacional, destacando sua relevância e potencial para aplicações práticas em modelos de aprendizado de máquina e Computação de Alto Desempenho. Eles demonstram que a linearização, juntamente com outras estratégias correlatas, como a fragmentação, pode proporcionar benefícios significativos para a otimização do desempenho de algoritmos, especialmente na multiplicação de matrizes. Através de estudos de caso e análises comparativas, os artigos evidenciam a importância da reorganização dos elementos dos tensores para melhorar a localidade dos dados, reduzir conflitos de cache, "TLB misses"e aprimorar a eficiência do prefetching de hardware, resultando em uma melhoria substancial no desempenho de programas em HPC que utilizam a multiplicação de matrizes.

Em resumo, os artigos revisados fornecem uma base teórica sólida e exemplos práticos que sustentam a importância da linearização de tensores e sua aplicação na otimização do desempenho de algoritmos, especialmente em relação à hierarquia de memória e ao acesso aos dados. Eles destacam a relevância dessas técnicas para avanços nas aplicações de Computação de Alto Desempenho e modelos de aprendizado de máquina, evidenciando seu potencial para melhorar significativamente a eficiência computacional e o desempenho de programas em HPC que utilizam a multiplicação de matrizes.

Referências

[1] Kazushige Goto and Robert A van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):1–25, 2008.

[2] Caio Salvador Rohwedder, Nathan Henderson, João P. L. De Carvalho, Yufei Chen, and José Nelson Amaral. To pack or not to pack: A generalized packing analysis and transformation. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23)*, page 14, Montréal, QC, Canada, February 25 - March 1 2023. ACM.

[3] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, March 2017.

[4] Uday Bondhugula. High Performance Code Generation in MLIR: An Early Case Study with GEMM, March 2020. arXiv:2003.00532 [cs].

[5] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGPLAN Notices*, 26(4):63–74, April 1991.

[6] David Gannon and William Jalby. The influence of memory hierarchy on algorithm

organization: Programming ffts on a vector multiprocessor. In *Characteristics of Parallel Algorithms*. MIT Press, 1987.

[7] O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 410–419, New York, NY, USA, December 1993. Association for Computing Machinery.

[8] Karim Esseghir. Improving data locality for caches. Master's thesis, Rice University, 1993.

[9] Bram Wasti, José Pablo Cambronero, Benoit Steiner, Hugh Leather, and Aleksandar Zlateski. LoopStack: a Lightweight Tensor Algebra Compiler Stack, May 2022. arXiv:2205.00618 [cs].

[10] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, December 2012. Publisher: World Scientific Publishing Co.