Trabalho 01

Cainã Figueiredo Pereira
April 16, 2021

Abstract

Este trabalho consiste no estudo de desempenho da operação $b_{nx1} = A_{nxn}.x_{nx1}$ (produto matriz-vetor), kernel de grande parte dos códigos de simulação computacional executados em diversos supercomputadores. Diversos fatores precisam ser considerados quando a produção de softwares com bons desempenhos é um dos objetivos principais. Para isso, é de suma importância que o programador possua conhecimentos sobre como certas estruturas de dados são representadas pelas diferentes linguagens de programação, o que irá lhe permitir uma adaptação de seus códigos para se conseguir o máximo de desempenho possível. Aqui, foram avaliados os impactos de uma programação que não leva em consideração aspectos como a organização dos dados na memória e o fluxo dos dados ao longo da hierarquia dos dispositivos de armazenamento. Os resultados mostraram que uma simples troca na ordem dos loops pode levar a impactos significativamente negativos no desempenho durante a execução.

1 Introdução

À medida que os softwares se tornam mais e mais complexos, seja através de requisitos que exigem a produção de imagens para jogos ultra-realistas em tempo real ou mesmo a realização de cálculos extremamente custosos para uma dada simulação, é certo que os seus desempenhos passam a ser uma preocupação durante o seu desenvolvimento. Diversos fatores precisam ser considerados quando a produção de softwares com bons desempenhos é um dos objetivos principais. Dentre eles, o programador pode lançar mão de certas práticas de programação que, de certa forma, permitem ao programa uma utilização mais adequada dos recursos computacionais, a exemplo da hierarquia de memórias. Em conjunto com essas boas práticas de programação, é de suma importância que o programador possua conhecimentos sobre como certas estruturas de dados são representadas pelas diferentes linguagens de programação, o que irá lhe permitir uma adaptação de seus códigos para se conseguir o máximo de desempenho possível.

Neste trabalho, diversos experimentos foram realizados a fim de analisar como essas práticas de programação podem, de fato, influenciar o desempenho percebido pelo usuário do programa. Diante disso, acompanhe mais detalhes através das próximas seções. A princípio, na seção 2, iremos descrever as especificações do projeto proposto. A seguir, as principais decisões que foram tomadas durante a execução do projeto são discutidas na seção 3. Os resultados experimentais são apresentados na seção 4 e, logo em seguida, discutidos na seção 5. Finalmente, este trabalho é concluído na seção 6 com uma breve consideração sobre todo o projeto.

Todos os arquivos e códigos gerados durante a realização deste projeto estão disponibilizados em um repositório do GitHub [1].

2 Descrição do projeto

Este trabalho consiste no estudo de desempenho da operação $b_{n\times 1} = A_{n\times n} \cdot x_{n\times 1}$ (produto matriz-vetor), kernel de grande parte dos códigos de simulação computacional executados em

[1] https://github.com/cainafigueiredo/Computacao$_A$lto$_D$esempenho/tree/main/Projeto1

diversos supercomputadores. Por isso, é essencial que essa operação seja muito bem implementada, ou seja, que a sua implementação leve em conta aspectos que impactam diretamente seu desempenho. Para exemplificar, uma simples troca na ordem de loops aninhados pode produzir significativas mudanças de desempenho. O algoritmo da operação que realiza o produto matriz-vetor pode ser encontrado no algoritmo 1.

Algorithm 1 Produto Matriz-Vetor

Entrada: $A_{n\times n}[n][n], x_{n\times 1}[n]$
Saída: $b_{n\times 1}[n]$

1: $b_{n\times 1}[n] = [0, ..., 0]$;
2: Para $i = 1$ até n faça
3: Para $j = 1$ até n faça
4: $b_{n\times 1}[i]$ += $A_{n\times n}[i][j] \cdot x_{n\times 1}[j]$

Este projeto contém certas especificações que tiveram de ser atentidas durante a sua realização. A primeira delas diz respeito às linguagens de programação em que as soluções deveriam ser implementadas, as quais foram definidas as linguagens C e Fortran. Para a compilação, nenhuma flag de otimização pôde ser utilizada. Como é possível alterar as ordens dos loops no algoritmo 1 sem muito esforço, ambas as soluções devem ser avaliadas em ambas as linguagens de programação. Essas avaliações consistem em análises gráficas do tempo necessário para realizar *somente* as operações de produto matriz-vetor em função das dimensões da matriz $A_{n\times n}$ e do vetor $x_{n\times 1}$, ou seja, em função do valor do parâmetro n. Os coeficientes de ambos os vetores $x_{n\times 1}$ e $b_{n\times 1}$, assim como da matriz $A_{n\times n}$ devem ser armanezados em variáveis de dupla precisão, tais como *double* e *REAL(8)*, onde tanto os coeficientes do primeiro vetor quanto da matriz são inicializados de forma aleatória. Enfim, os valores de n devem ser avaliados dentro do intervalo $0 < n \leq n_{max}$, onde n_{max} é um parâmetro desconhecido, a princípio, e por isso a sua determinação é deixada como parte da concretização deste projeto. No entanto, não há uma especificação de quais valores de n devem ser avaliados para posteriores representações em gráficos.

Em suma, nas próxima seção iremos apresentar as soluções para os seguintes itens:

1. Qual a dimensão máxima, n_{max}, para que o armazenamento da matriz e dos vetores apresentados previamente, cujos coeficientes aleatórios são representados como variáveis de dupla precisão, seja possível de ser realizado na memória RAM disponível?
2. Como a ordem dos loops do algoritmo 1 e as linguagens de programação, C e Fortran, afetam o desempenho da operação produto matriz-vetor para diferentes dimensões n?

3 Decisões de projeto

Nesta seção, serão discutidas as principais decisões tomadas durante a realização do projeto. Além das linguagens de programação especificadas, ou seja, C e Fortran, a linguagem Python foi usada para duas tarefas bem específicas: 1) requisitar ao sistema operacional a memória disponível e 2) gerar visualizações dos resultados dos experimentos. Na seção seguinte, abordaremos mais detalhes sobre como se deu a determinação do valor máximo do parâmetro n e onde o Python se encaixa nessa etapa.

3.1 Determinação da dimensão máxima

O objetivo principal deste trabalho é analisar o impacto das diferentes representações de arrays na memória e de programas que levam em conta essas características. Sendo assim, é importante garantir que todos os nossos dados estarão armazenados na memória e, portanto, não estarão se aproveitando de técnicas que permitem extrapolar o espaço da RAM através do uso do disco. Sendo assim, a determinação de n_{max}, ou seja, o valor máximo de n tal que os dados armazenados caibam na RAM, representa uma etapa importante.

$$n_{max}{}^{*} = \lfloor(-1 \pm \sqrt{1 + \frac{S}{B}})\rfloor \tag{1}$$

Em um cenário ideal, onde toda a RAM estaria disponível para a alocação desses dados [2], bastaria usar a equação 1, onde B é o número de bytes ocupados por cada coeficiente da matriz ou dos vetores e S é a quantidade de RAM disponível. No entanto, esse cenário não existe e nem mesmo é possível estimar um valor razoável para n_{max} devido à existência de outros processos que estão compartilhando esse recurso e até mesmo devido à necessidade de RAM disponível para um correto funcionamento do sistema operacional.

Diante disso, uma abordagem adotada foi realizar uma busca binária num espaço em que n pode assumir valores de 1 a $n_{max}{}^{*}$, porém o primeiro desafio com essa abordagem foi em como obter um feedback a respeito do sucesso ou não da alocação dos dados com os diferentes valores de n. Quando usada uma simples declaração de arrays, o programa tinha a sua execução interrompida devido ao levantamento de exceções do tipo *Segmentation Fault*. Por outro lado, quando funções de alocação dinâmica foram utilizadas, observou-se que era possível alocar mais espaço que a RAM disponível [3] e, portanto, o feedback dessas funções não eram adequadas para o nosso propósito - pelo menos não para esse valor máximo ideal $n_{max}{}^{*}$. Enfim, após algumas tentativas, chegamos à solução final para uma aproximação do valor máximo n_{max}: realizar a busca binária no parâmetro n, porém com o espaço reduzido ao intervalo $[1, n_{free}]$, onde n_{free} é também obtido com a equação 1, porém considerando S como a RAM livre no momento da execução. Assim, o feedback das funções de alocação se tornam mais confiáveis, pois sabemos que os dados não ocuparão mais do que a memória livre.

Para a solução mencionada acima, foi preparado um script em Python para executar uma chamada ao sistema utilizando o comando '*cat /proc/meminfo*' e em seguida extrair o campo que informa a quantidade de bytes livre. Esse valor é então escrito em um arquivo e, posteriormente, lido pelos programas em ambas as linguagens C e Fortran durante a execução da busca binária. Após a busca ser concluída, temos uma aproximação para n_{max} durante a respectiva execução.

3.2 Implementações da operação Produto Matriz-Vetor

Para esta etapa, apenas foram seguidas as especificações do projeto, ou seja, foram implementadas duas versões da operação matriz-vetor para cada uma das linguagens (C e Fortran). A primeira versão é apenas uma tradução do algoritmo 1 para essas linguagens, enquanto que a segunda ver-

[2] Estamos nos referindo a uma matriz quadrada $A_{n\times n}$ e dois vetores n-dimensionais, x_n e b_n.

[3] Através de algumas consultas em fóruns, concluiu-se que isso ocorre devido a um mecanismo que o SO utiliza, denominado 'Memory Overcommitment'.

são possui como diferença apenas a ordem dos loops. Os tempo de execução começa a ser contado imediatamente antes da chamada deste procedimento e finaliza assim que ele acaba.

3.3 Geração de gráficos

Para esta tarefa, ambas as implementações em C e Fortran geram arquivos CSV contendo duas colunas, *n* e *time(s)*, e com as amostras nas linhas. Esses arquivos são lidos por um script em Python e esse gera os gráficos. Essa escolha se deu devido às facilidades que o Python oferece e porque essa tarefa não é o foco das análises deste trabalho.

4 Resultados experimentais

Os experimentos que serão descritos nesta seção foram realizados em um ambiente Ubuntu 18.04.5 LTS com uma RAM de 4GB e um processador Intel Core i3-2310M. Exceto por processos de segundo plano, todas as demais aplicações foram finalizadas antes da execução dos experimentos a fim de economizar memória e processamento para a obtenção das nossas amostras. Para as implementações em C, em ambas as versões da operação Produto Matriz-Vetor, o parâmetro n_{max} foi igual a 11564. Por outro lado, em ambas as implementações em Fortran, para ambas as versões da mesma operação, o parâmetro n_{max} foi igual a 3135. O motivo para essa diferença não foi descoberto. Feito isso, foram obtidas N amostras de pares (n_i, T_i), onde T_i é o tempo necessário para a realização da operação e $n_i = 2^{i-1}$, tal que $i \in [1, N]$ e $n_N \leq n_{max}$.

O gráfico da figura 1 mostra o tempo, em segundos, necessário para executar a operação Produto Matriz-Vetor para os diferentes valores de n amostrados. Cada curva representa uma implementação em uma dada linguagem e em uma das duas versões, conforme discutido na seção 3.2. No eixo x, temos o $\log_2(n)$ e no eixo y, a quantidade de segundos para a conclusão da operação em consideração.

5 Discussão

Infelizmente, o número de amostras representadas na figura 1 é limitado, em especial no caso das execuções das implementações em Fortran. Apesar disso, podemos supor que as curvas para o Fortran seriam semelhantes às do C se extrapolássemos as regiões amostradas, pois as curvas se assemelham bastante. No entanto, perceba que para uma mesma linguagem, uma simples troca na ordem dos loops faz com que os tempos necessários para a execução da mesma operação sejam muito diferentes, atingindo a uma diferença com um fator de 4x nas últimas amostras das implementações em C, por exemplo. Pelo comportamento dessas curvas, é possível afirmar que essa diferença tende a se tornar cada vez mais significativa.

Além disso, perceba que as ordens dos loops que levam às curvas com maiores tempos em cada uma das linguagens, representadas pelas curvas verde e laranja, são diferentes. Para compreender esse fenômeno, precisamos considerar questões de mais baixo nível, principalmente com respeito a métodos de armazenamento de arrays muldimensionais na RAM e à dinâmica dos dados ao longo da hierarquia de memória. Em se tratando do primeiro ponto levantado, há duas variantes que permitem a organização de arrays muldimensionais, que são: 1) row-major (linhas são armazenadas em espaços de memória contíguos) e 2) column-major (colunas são armazenadas em espaços de memória contíguos). A linguagem C utiliza a primeira abordagem, enquanto que

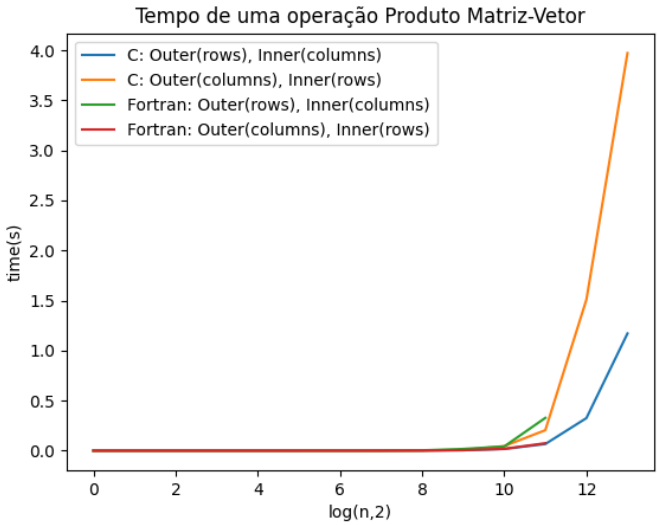

Figure 1: Tempo para a execução completa da operação Produto Matriz-Vetor para as suas diferentes implementações.

o Fortran faz uso da segunda [4].

A forma com que essas linguagens implementam esse armazenamento se torna relevante ao considerarmos que o acesso a dados passa por uma hierarquia de dispositivos de armazenamento, a qual utiliza diversas técnicas para melhorar o desempenho. Dentre essas técnicas, procura-se manter mais próximo do núcleo de processamento aqueles dados com maiores potenciais de serem acessados em um futuro próximo, o que nos introduz ao conceito de *taxa de cache hit*, ou seja, a taxa com que um acesso a um determinado dado é concluído ainda no nível da memória cache. Aliado a isso, os conceitos de *localidade temporal* e *localidade espacial*, em especial esse último, possibilitam uma maior taxa de cache hit. Essas técnicas buscam trazer para mais perto da CPU não só o dado requisitado, mas também uma quantidade de dados que estão armazenados imediatamente após a sua posição. Isso significa que toda a linha de um array bidimensional, por exemplo, pode subir um nível na hierarquia de memória na abordagem row-major com uma única operação de acesso, enquanto que o oposto ocorreria na abordagem column-major, i.e., vários elementos da mesma coluna são armazenados na cache em uma única operação. Esse é o motivo pelo qual a implementação no C com loop externo ao longo das colunas e a implementação no Fortran com loop externo ao longo das linhas (curvas laranja e verde, respectivamente) atingiram uma performance pior se comparadas às implementações da curva azul e vermelha. Essas duas últimas se aproveitam dos dados que foram antecipadamente carregados na memória cache, aumentando assim a taxa de cache hit e, consequentemente, obtendo um melhor desempenho.

[4] https://www.mathworks.com/help/coder/ug/what-are-column-major-and-row-major-representation-1.html

6 Conclusão

Neste trabalho, foram avaliados os impactos de uma programação que não leva em consideração aspectos como a organização dos dados na memória e o fluxo dos dados ao longo da hierarquia dos dispositivos de armazenamento. Para isso, uma simples tarefa que realiza uma operação de produto matriz-vetor foi implementada em duas diferentes linguagens de programação, C e Fortran, e em duas variantes obtidas a partir da alteração da ordem dos loops nesse procedimento.

Diversas amostras do tempo necessário para a execução dessa operação foram obtidas para diferentes valores do parâmetro ajustável n, respeitando sempre um limite máximo n_{max}. Os resultados mostraram que uma simples troca na ordem dos loops pode levar a impactos significativamente negativos no desempenho durante a execução, o que é uma consequência direta de uma violação de heurísticas que executam em níveis mais baixos, como a carga antecipada de dados contíguos para a cache.

Em suma, conclui-se que é essencial que o programador esteja ciente de como a linguagem de programação usada implementa certas estruturas de dados, de forma a poder assim, através de sua programação, maximizar a probabilidade dos dados necessários para a sua aplicação serem encontrados ainda no nível da memória cache.