Trabalho 2 - Introdução a Computação Científica

Giordano Henrique Silveira - GRR20197154

Gabriel Razzolini Pires de Paula - GRR20197155

Fevereiro 2023

1 Introdução

O objetivo deste trabalho foi otimizar o trabalho 1, além de medir, utilizando o **likwid**, o quanto as otimizações afetaram o desempenho do nosso código.

No trabalho anterior, foi implementado o método dos gradientes conjugados para a resolução de um sistema linear. Ele é um método iterativo que envolve várias multiplicações de matrizes e vetores. Portanto, para sistemas muito grandes, o método tende a piorar. Para este trabalho, era necessário otimizar duas funcionalidades: o cálculo do resíduo e o método dos gradientes conjugados com pré-condicionadores. E comparar as duas versões.

1.1 Arquivos

Antes de continuar, é importante explicar como estão organizados os arquivos do diretório. Nele serão encontrados: um arquivo **makefile**, um **shell script**, um **python script**, arquivos **.c** e **.h** terminados com o sufixo **v1** e **v2**, além deste relatório em pdf.

Os arquivos terminados em **v1** são referentes ao trabalho 1 - sem otimização, e os terminados em **v2** são referentes ao trabalho 2 - com otimização. Com relação ao código deste trabalho, não há necessariamente as mesmas funções, visto que era somente necessário otimizar parte do código.

O **makefile** compila ambas versões e gera dos arquivos executáveis: **cgSolver_v1** e **cgSolver_v2**. O primeiro é para a versão não otimizada e o segundo é para a versão otimizada. O arquivo em python trata dos arquivos **.csv**, gerados pelo **shell script**, e dos gráficos **.png** para realizar a comparação.

Entretanto, para executar tudo é necessário somente executar o **shell script**: **create_likwid_files.sh**. Ele irá, automaticamente, chamar o makefile, executar e gerar os .csv para cada tamanho especificado pelo professor, além de chamar o script em python. Para excutar o shell script use:

```
./create_likwid_files.sh
```

1.2 Execução

Após criados os arquivos executáveis, dentro do shell script, eles são rodados da seguinte maneira:

```
likwid-perfctr -O -C 3 -g ${k} -m ./cgSolver_v1 -n $s \

    -i 150 -k 7 -p 1 -o saida_{versão}.txt -d diagonal > ${s}_${k}_{versão}.csv
```

onde **-C** é o core da CPU onde programa será executado, **-g** é grupo do **likwid**, **-m** é o arquivo executável, **-n** é o tamanho da matriz, **-i 150** representa que metódo executará 150 vezes, **-k 7** dizendo que a matriz será uma **7-diagonal**, **-p 1** que é o métodos dos gradientes conjugados com pré-condicionador, **-o** é o arquivo contendo a saída formatada e **-d** é a flag criada para transformar uma matriz em diagonal dominante para a aplicação do método.

Sobre a máquina que os programas foram testados, a máquina utilizada foi a **H58** do laboratório 1 e 2 do departamento de informática - DINF. Uma máquina de 4 núcleos, 8 GB de memória RAM, 32 kB na cache L1, 256 kB na cache L2 e 6 MB na cache L3, que é compartilhada.

2 Otimizações

No trabalho 1 foi usado para representar os coeficientes do sistema linear uma matriz onde todos os dados, inclusive os nulos, são armazenados. Continua sendo uma matriz na versão dois, porém são armazenado somente as colunas onde os elementos não são nulos. Por exemplo, seja uma matriz de tamanho **1000** e **7-diagonal**, será alocado para ela um espaço de **1000x7**, **1000 linhas** e **7 colunas**. Na coluna central, ou seja, a coluna $\lfloor k/2 \rfloor$, onde k é o número de *k-diagonais*, será armazenado os elementos da diagonal principal. As colunas menores que $\lfloor k/2 \rfloor$ são as colunas onde estão os elementos das diagonais inferiores e as colunas maiores que $\lfloor k/2 \rfloor$ guardam os elementos das diagonais superiores. Deste modo, é guardado somente os dados que serão realmente usados.

Também foram aplicados os conceitos, quando possível, de *Unroll* e *Jam* para permitir que as operações *SIMD* fossem usadas. As flags: **-O3 -mavx2 -march=native** também foram usadas na compilação, permitindo que o compilador consiga fazer as melhores otimizações possíveis durante todo código na fase de compilação. Essas flags de compilação foram usadas tanto no trabalho 1, quanto no trabalho 2. Essas foram as principais otimizações arquitetadas para esta versão 2 do trabalho.

3 Comparação

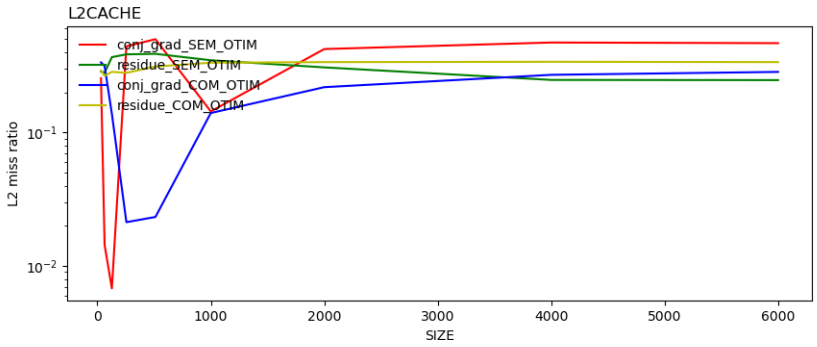
Certo, foram aplicadas otimizações durante todo o código, logo, a performance foi, de algum modo, afetada. Para medir o quanto a performance foi afetada, foram usados alguns grupos do **Likwid** para retirar algumas métricas e realizar

a comparação, são esses grupos e métricas: o **L2CACHE** para ver o número de **cache miss**, a **L3** usada para medir o **memory bandwidth**, a **FLOPS_DP** para medir as operações de **ponto flutuante** e o **FLOPS_AVX** para medir as operações de pontos flutuantes, porém em utilizando instruções **AVX**, instruções essas que foram projetadas para acelerar operações vetoriais. Além de medir o **tempo de execução** de cada programa.

Para cada grupo foi gerado um .csv com o nome: **GRUPO_LIKWID.csv**. Destes arquivos são gerados os gráficos usados para as comparações. No eixo **y** desses gráficos estão representados as métricas e no eixo **x** estão representados os tamanhos. Os tamanhos variam de **32** a **6000**. Existem 4 linhas nesses gráficos todas de cores diferentes. A linha em vermelho representa o métodos dos gradientes conjugados com pré-condicionadores sem otimização, em azul o mesmo método com otimização, em verde o calculo do resíduo sem otimização e, em amarelo, o resíduo com otimização.

3.1 Cache miss

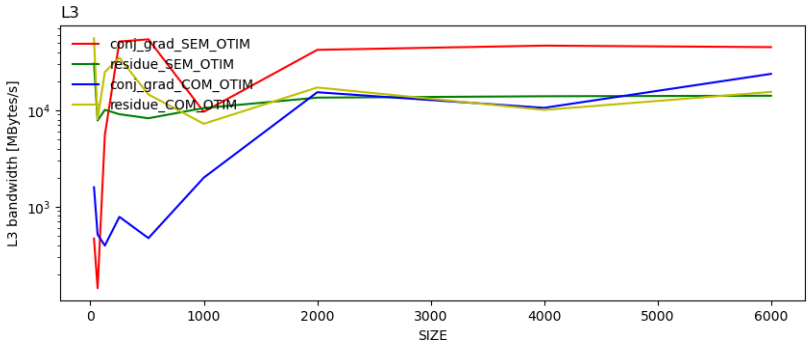
Cache miss é um evento que ocorre quando um processador ou um dispositivo de armazenamento em cache tenta acessar um item de dados que não está armazenado em sua cache. Quanto mais cache miss, mais o computador terá que acessar a memória principal para pegar os dados que não estão na cache. Acessar a memória principal é uma operação custosa. Ela demanda mais ciclos de processamento, além de ter o tempo da memória para acessar esses dados. Portanto, quanto menos cache miss, melhor. Quanto à isso:



como apontado pelo gráfico, embora a otimização piore um pouco o cache miss no cálculo com resíduo, no cálculo do método a otimização melhora o cache miss consideravelmente á medida que o tamanho da matriz aumenta. Visto que é o método que demanda mais processamento, a otimização melhora o cache miss. Apesar de não ser uma mudança muito grande.

3.2 Memory bandwidth

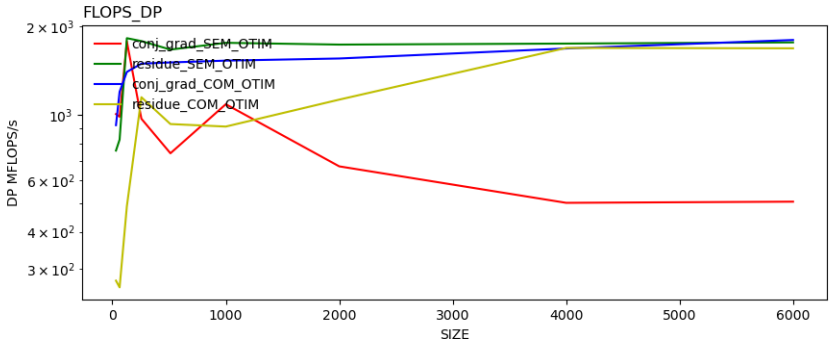
A largura de banda da memória (memory bandwidth) se refere à quantidade de dados que podem ser transferidos entre a CPU e a memória em um determinado período de tempo. Portanto, para aplicações de computação a largura de banda da memória é um fator crítico no desempenho do sistema. Uma largura de banda de memória maior permite que a CPU acesse mais rapidamente e manipule grandes quantidades de dados, o que pode levar a um processamento mais rápido e eficiente. Quanto a isso temos o seguinte:

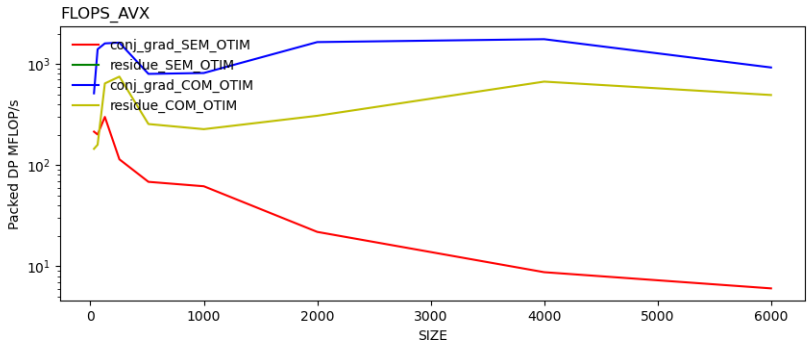


Não a uma melhora significativa no Memory bandwidth no trabalho otimizado, porém não há uma piora significativa também. Ela se mantém constante. Algo em torno de 10^4 [Mbytes/s].

3.3 Operações em ponto flutuante

Em geral, quanto mais FLOPS um sistema de computação científica for capaz de realizar, melhor será seu desempenho em aplicações que envolvem operações em ponto flutuante. Isso ocorre porque muitas aplicações científicas exigem um grande número de operações em ponto flutuante para serem executadas. Então, abaixo estão os gráficos contendo o FLOPS e o FLOPS AVX.





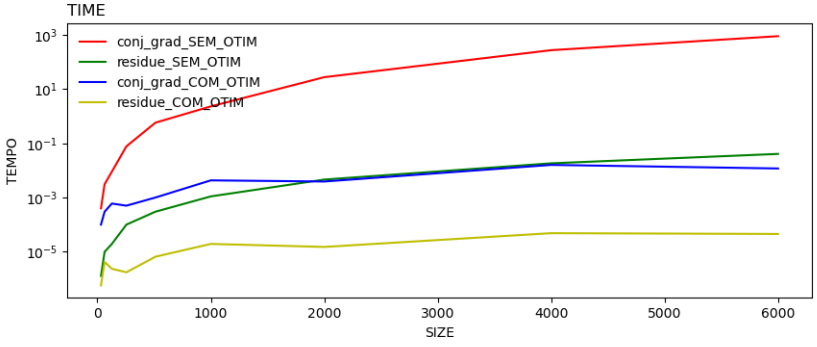
Como mostrados pelos gráficos, há uma melhora significativa tanto em FLOPS, quando em FLOPS AVX.

Para o FLOPS normal, usando tamanhos grandes de matrizes, as implementações dos resíduos se mantém constantes. Não há mudança significativa. Contudo, o método dos gradientes conjugados com otimização melhora em 10 vezes em relação ao método sem otimização.

Para o FLOPS AVX, a melhora é muito mais significativa. O cálculo do resíduo sem otimização não aparece no gráfico, ou seja, ele não utiliza operações AVX. Para o método dos gradientes otimizado, a melhora é, no mínimo, 10 vezes maior, isso para tamanho grandes de matrizes. As otimizações feitas foram eficientes, principalmente, para esta métrica.

3.4 Tempo

Por fim, mas tão importante quanto, o tempo. Quanto menos tempo executar um programa melhor. Consequentemente, temos:



Neste gráfico é visível que o o gradiente conjugado otimizado é, no mínimo, 10^6 vezes mais rápido que o não otimizado, não só isso, o cálculo do resíduo otimizado é 10 vezes mais rápido que o não otimizado. O ganho das otimizações, pelo menos em tempo, é bem alto, principalmente no cálculo do método.

4 Corretude

Apesar da mudanças significativas em todas as métricas usadas, não foi possível manter a mesma corretude do algoritmo da versão um do trabalho. Para o primeiro trabalho, para uma matriz de 6000x6000, o resíduo final fica em torno de $1e^{-13}$, porém para o mesmo tamanho de matriz no trabalho otimizado, ele fica em torno de 10^1. É muito mais alto. Porém, haja vista a dificuldade de mexer somente com as diagonais e realizar contas em cima dela, é um resíduo parcialmente aceitável. Isto é possível ver nos arquivos **saida_v1.txt** e **saida_v2.txt**, onde contém aquela saída formatada pedida para o trabalho 1. No arquivo **saida_v1.txt** contém a saída da versão 1 do trabalho e no arquivo **saida_v2.txt** contém a saída da versão 2 do trabalho.

É provável que esse aumento ocorra ao não gerar um sistema equivalente para o cálculo do método. Para a aplicação do método, é necessário transformar o sistema linear para que os coeficientes sejam simétricos e positivos definidos. Isso é feito ao fazer a seguinte conta:

$$A^t * A = A^t * b,$$

onde A é a matriz dos coeficientes, A^t é a matriz dos coeficientes transposta e b são os termos independentes.

Esta conta, usando as somente as diagonais, torna-se complicada de fazer. É provável que achamos um sistema que fosse parcialmente parecido, visto que se não fosse nada parecido o resíduo daria muito maior que 10^1.

5 Conclusão

Apesar da corretude não se manter no mesmo nível, ele realiza todas as contas e cálculos com vetores. E nesses cálculos o programa foi otimizado.

O *Unroll* e *Jam* tem a sua participação neste trabalho permitindo o uso de operações SIMD e melhorando o FLOPS, o FLOPS AVX, além de preencherem melhor o pipeline do processado. O tempo e gasto de memória, são melhorados, principalmente, por não armazenamos a matriz inteira, armazenamos um fatia muito menor. Ao invés de processar uma matriz de **6000x6000**, o que daria mais de 1 MB de dados, processamos uma matriz de **6000x7**, bem menos que 1 MB de dados.

Neste trabalho, o que consideramos mais importante no desenvolvimento foi melhorar o desempenho, utilizar instruções SIMD, melhorar o tempo, melhorar o cache miss e aumentar as operações em ponto flutuante. A corretude foi a segunda coisa mais importante no desempenho. Tentamos ao máximo manter uma corretude equivalente, mas o mais baixo que conseguimos chegar foi 10. E, tendo em vista todos os resultados apresentados na versão 2, nós conseguimos melhorar o desempenho. Ela é consideravelmente melhor do que a versão 1.