CI-1164 - Introdução à Computação Científica

Relatório do Trabalho 2

Luan Machado Bernardt - GRR20190363
Lucas Müller - GRR20197160

August 14, 2021

Conteúdo

1 Introdução **3**

2 Especificações do Computador **4**

3 Como rodar o Experimento **4**

4 Otimizações **6**

5 Gráficos **9**

1 Introdução

Neste trabalho foram implementados algoritmos para cálculo de polinômios por Interpolação e Ajuste de Curvas, que utilizam fatoração LU. Também foram realizadas técnicas de otimização de código, apreendidas em aula, com o objetivo de melhorar a performance.

2 Especificações do Computador

Os testes foram executados na máquina i31 do DInf, cujas características, segundo o comando,

```
likwid-topology -c -g
```

são as seguintes:

- Nome do Processador: Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
- Tipo do Processador: Intel Coffeelake processor
- Sockets: 1
- Núcleos por Socket: 4
- Threads por Núcleo: 1
- Memória cache com 3 níveis
- L1: Tamanho da cache de 64kB e da linha 64B
- L2: Tamanho da cache de 256kB e da linha 64B
- L3: Tamanho da cache de 6MB e da linha 64B

3 Como rodar o Experimento

Para rodar o experimento deve-se seguir o seguintes passos:

1. Para compilar o programa em executável, é necessário utilizar o comando:

   ```
   make
   ```

   que serve para simplificar a compilação com o LIKWID, que está presente no arquivo make, porém é muito longa.

2. ```
   echo "performance" > /sys/devices
   /system/cpu/cpufreq/policy3/scaling_governor
   ```

   Fixa a frequência do processador, para que haja igualdade nos testes.

3. Para executar o teste deve-se chamar:

   ```
   python3 gera_entrada n |
   likwid-perfctr -C 3 -g $1 -m ./main
   ```

   Onde n é a quantidade de pontos que devem ser gerados pelo programa gera_entrada e $1 é algum grupo para medição, neste trabalho serão utilizados, FLOPS_DP, L3 e L2CACHE.
   ```

4. Após os testes, deve-se retornar o porcessador a frequência original:

```
echo "powersave" > /sys/devices
/system/cpu/cpufreq/policy3/scaling_governor
```

Para facilitar a execução dos testes, há a possibilidade de utilizar um script bash, que realiza todos os passos acima e executa o teste para n valores previamente definidos:

```
./perfctr $1
```

Onde $1 é um dos grupos de medição. Além disso os valores de n podem ser alterados.

4 Otimizações

Foram implementadas otimizações vistas em aula com o objetivo de melhorar a performance nas principais funções do trabalho, as quais serão vistas a seguir:

Struct: A estrutura de dados utilizada foi:

```
typedef struct {
    unsigned int n, m;
    double *A;
    double *L, *U;
    int *vetTroca;
    union { double *x, *B; };
} t_sist;
```

E para o programa foram criadas 3 variáveis t_sist, **SL**, que guarda os dados de entrada recebidos em stdin, **Int** que guarda os dados para a interpolação e **Ajc**, que armazena dados relacionados ao ajuste de curvas.

- n e m: são as variáveis que guardam as dimensões da matriz.
- A: armazena a matriz de forma linear. Para SL, a matriz lida da entrada, já para Int e Ajc, as matrizes criadas nas funções de Interpolação e Ajuste de Curvas respectivamente.
- L e U: para a Int e Ajc, guardam as matrizes resultantes da triangularização, enquanto para a de entrada não guarda nada e aloca apenas L.
- vetTroca: matriz alocada linearmente que possui a ordem das trocas a serem realizadas no vetor de termos independentes após a triangularização, é alocada apenas para Int e Ajc, pois é inicializada na primeira execução da triangularização, o que não ocorre para SL.
- x ou B: possuem duas utilidades, para SL ele serve pra guardar o vetor x, que possui os valores tabelados, enquanto para Int e Ajc, guarda o vetor B com termos independentes. Fez-se o uso de uma união, pois tanto x quanto B tem o mesmo tamanho e onde há um, não existe o outro.

Interpolação: O cálculo da matriz da interpolação envolve elevar o valor de x[i] ao número correspondente à coluna j, para cada linha i. Como uma linha não depende da anterior, implementou-se a técnica de *unroll and jam*, percorre n linhas da matriz a cada laço de i, sendo n o fator de *unroll*, que nesse trabalho é 8. Essa técnica faz melhor uso da memória cache, pois carrega 8 linhas da matriz por vez.
Como visto acima cada valor da matriz, corresponde ao quadrado do valor da coluna anterior, sendo assim, para toda linha atribui-se 1 ao primeiro elemento da linha e x[i] ao segundo, e a partir do terceiro multiplica-se o valor da coluna anterior por x[i], ou seja, pelo elemento da segunda coluna na mesma linha, i.e.:

```
...
// Coluna 0
Int->A[Int->n*i] = 1.0;
...
// Coluna 1
Int->A[Int->n*i+1] = SL->x[i];
...
// Colunas 2..n-1
Int->A[Int->n*i+j] = Int->A[Int->n*i+1] * Int->A[Int->n*i+(j-1)];
...
```

Dessa forma, não é nessário utilizar nenhuma função potência para calcular cada elemento da matriz.

Ajuste de Curvas: Para a matriz do ajuste é necessário calcular $\sum_{k=0}^{n-1}(x[k]^i * x[k]^j)$, onde i é a linha e j a coluna, logo se o valor do somatório será o mesmo para toda posição da matriz com a mesma soma de i+j, então calcula-se a primeira linha da matriz e, a partir da segunda, copia-se o valor da diagonal superior direita, i-1 e j+1, sendo necessário calcular somente a última coluna de cada linha.

```
...
Ajc->A[Ajc->n*i+j] = Ajc->A[Ajc->n*(i-1)+(j+1)];
...
```

Como se sabe, potenciações de ponto flutuantes são operações caras, e mesmo que seja necessário calcular apenas um somatório por linha da matriz, serão calculadas 2n potências para cada coluna n da matriz, portanto se n for muito grande, muitas operações serão realizadas. Uma solução encontrada para esse problema, foi a implementação de uma *lookup table* que guarda todos os valores de $x[i]^j$, onde i é a linha e j a coluna, e ambos vão de 0 até n-1. Essa tabela é montada no começo da função, e é utilizada no cálculo da matriz do ajuste, i.e.:

```
...
//Primeira linha da matriz (i = 0)
for (unsigned int j=0; j < Ajc->n; ++j)
      for (unsigned int k=0; k < Ajc->n; ++k)
          Ajc->A[j] += lookup[Ajc->n*j+k];
...
```

Por meio da *lookup table* faz-se possível reutilizar valores que serão necessários várias vezes durante o programa, tendo o custo de calculá-los apenas um vez.

Triangularização: Um dos principais problemas da triangularização com pivoteamento parcial é que como há troca de linhas, deve-se trocar também a ordem dos termos independentes, mas como a ordem é sempre a mesma, pois a triangularização está sendo realizada sobre a mesma matriz, faz-se o uso de um vetor que guarda a ordem das trocas e os elementos que devem ser trocados,

desta maneira, só há a necessidade de triangularizar a matriz uma vez, haja vista que após a primeira execução é possível permutar os valores do vetor de termos independentes da maneira correta, fora da função de triangularização.

```
// versao nao otimizada
// era necessario triangularizar toda a matriz para trocar
// trecho executado para cada linha i da matriz
pivo = maxValue(copia,SL->n,i);
if (pivo != i) {
    trocaElemento(SL->B+i, SL->B+pivo); // troca termo independente
    trocaLinha(copia, i, pivo, SL->n);
    trocaLinha(SL->L, i, pivo, SL->n);
}

// versao otimizada
// ao inves de trocar, guarda as posicoes que devem ser trocadas
// trecho executado para cada linha i da matriz
pivo = maxValue(SL->U,SL->n,i);
if (pivo != i) {
    SL->vetTroca[2*i] = i;
    SL->vetTroca[2*i+1] = pivo;
    trocaLinha(SL->U, i, pivo, SL->n);
    trocaLinha(SL->L, i, pivo, SL->n);
}
```

5 Gráficos

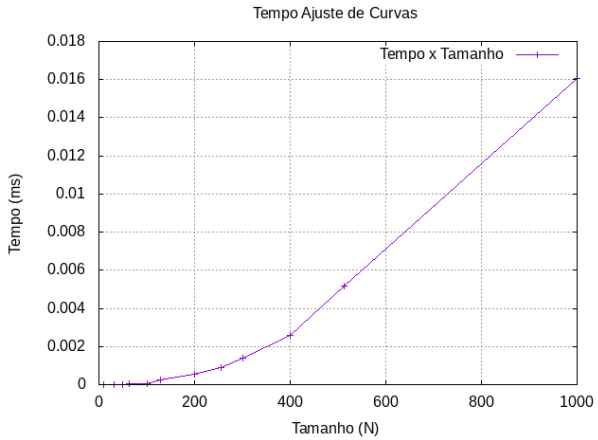

Figure 1: Runtime para Ajuste de Curvas

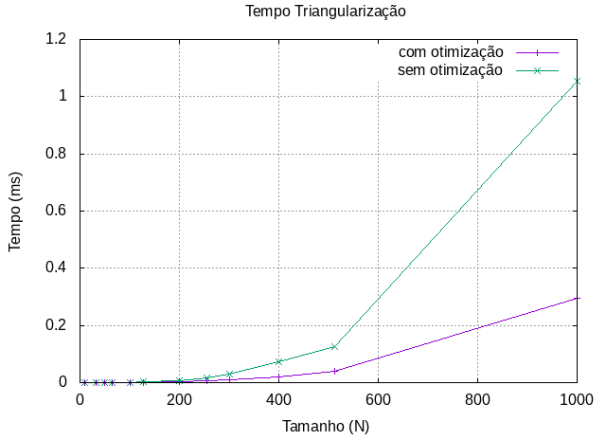

Figure 2: Runtime para Triangularização

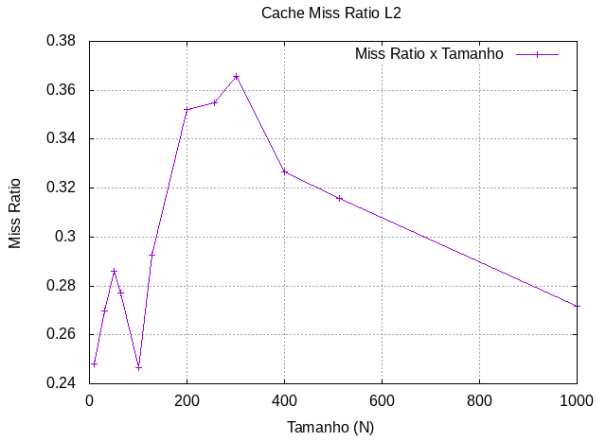

Figure 3: Cache Miss Ratio L2 para Ajuste de Curvas

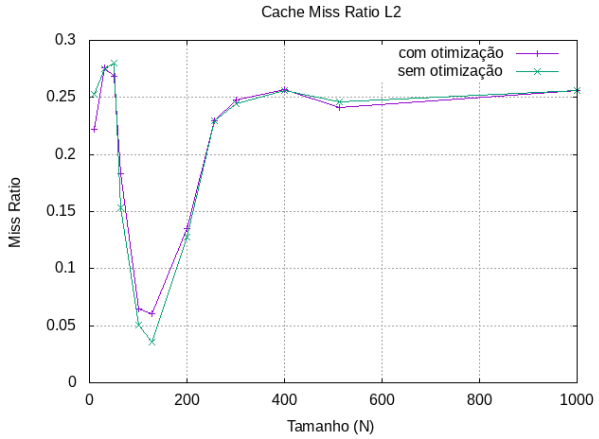

Figure 4: Cache Miss Ratio L2 para Triangularização

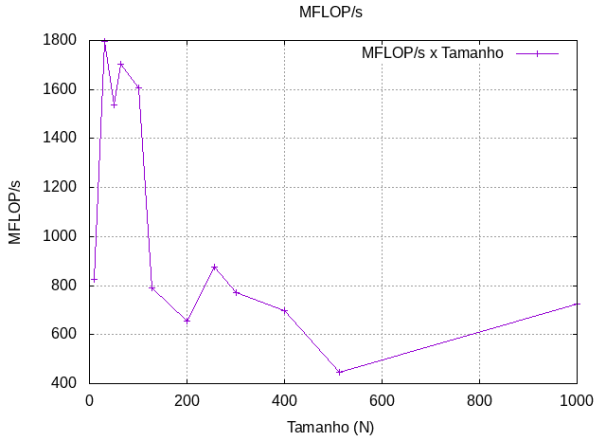

Figure 5: MFLOPS/s para Ajuste de Curvas

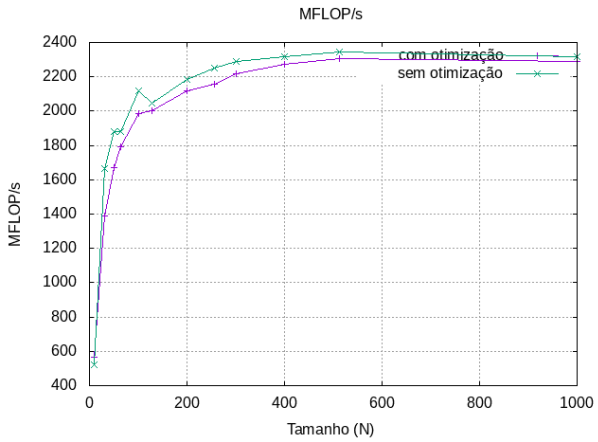

Figure 6: MFLOPS/s para Triangularização

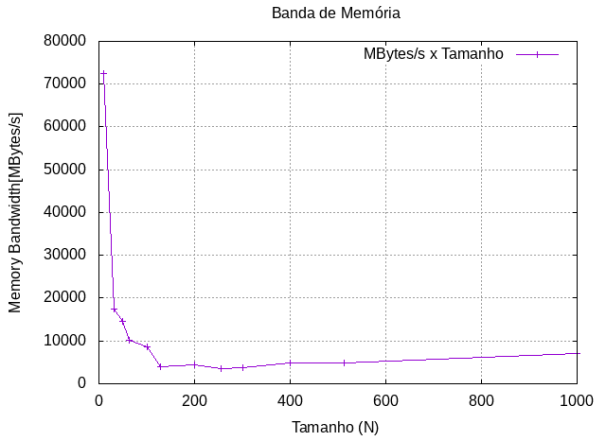

Figure 7: Banda de Memória para Ajuste de Curvas

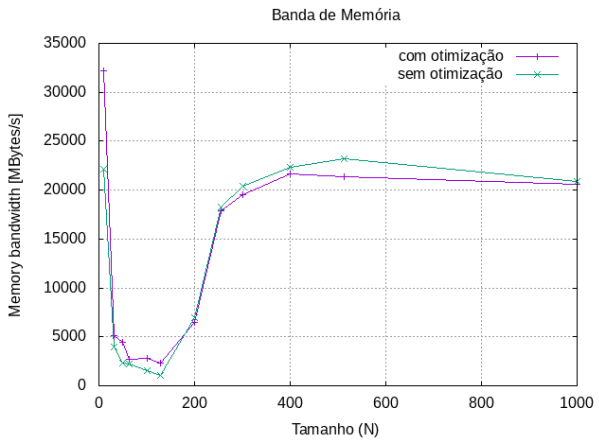

Figure 8: Banda de Memória para Triangularização