Relatório do Exercício-Programa 2 de MAC0219

Giovanna Uchôa Maia Gomes
14574998

Isabella Aparecida Costa Caselli
13686685

Muriel Toneli Latorre
14599710

24 de novembro de 2024

Resumo

Este é o relatório do Exercício-Programa 2 da disciplina Programação Concorrente e Paralela (MAC0219), da Universidade de São Paulo, ministrada pelo Prof. Alfredo Goldman. O programa consiste na paralelização de um código para o cálculo da Equação de Calor em Estado Estacionário utilizando CUDA. Neste relatório, está documentado os passos que foram seguidos em cada uma das partes do EP, explicitando quais foram os desafios encontrados, bem como as soluções desenvolvidas. Ainda, em anexo, é possível visualizar as três versões do código-fonte usadas em formato `.c`, `.cu` e `.cu`, respectivamente para cada parte, desenvolvidas de maneira colaborativa através da plataforma Github.

Conteúdo

1 Introdução

A proposta deste exercício é paralelizar o cálculo da Equação de Calor em Estado Estacionário usando CUDA, comparando a eficiência das versões paralelizadas com a implementação sequencial. Tal equação é descrita pela seguinte expressão de Laplace:

$$\frac{\delta^2 h}{\delta x^2} + \frac{\delta^2 h}{\delta y^2} = 0$$

A função $h(x, y)$ representa a distribuição de calor em estado estacionário e é uma solução da expressão anterior. Dessa forma, iremos considerar um espaço bidimensional com bordas de temperaturas fixas e utilizaremos o o método de diferenças finitas para aproximar a solução.

Em linhas gerais, a área do problema é discretizada em uma malha, onde cada ponto interno tem sua temperatura calculada como a média das temperaturas dos quatro pontos vizinhos. As bordas possuem temperaturas fixas, enquanto os pontos internos são ajustados iterativamente até atingir uma condição de convergência, ou seja, quando a diferença entre iterações sucessivas for suficientemente pequena.

Assim, o método é implementado em código sequencial, usando dois vetores: um para armazenar os valores atuais e outro para os novos valores calculados. Esse processo, conhecido como iteração de Jacobi, utiliza multiplicações em vez de divisões para otimizar o desempenho.

Dessa maneira, o Exercício-Programa foi segmentado em 3 tarefas:

1. **Tarefa 1:** Implementação das funções de inicialização e de iteração de Jacobi de maneira sequencial.
2. **Tarefa 2:** Alteração em código gerado na Tarefa 1 para que utilize CUDA como maneira de paralelizar o problema.
3. **Tarefa 3:** Alteração em código gerado na Tarefa 2 para que considere um corpo quente com temperatura fixa de 37°C.

Agora, vamos verificar os desafios e as soluções encontradas em cada tarefa. É importante ressaltar que a Tarefa 1 foi disponibilizada pelo Professor e, portanto, não será considerada neste presente relatório.

2 Tarefa 2

Para a Tarefa 2, iremos considerar o código sequencial disponibilizado e realizaremos modificações para que a implementação utilize CUDA para paralelizar o problema. Assim, vamos verificar os procedimentos realizados e os desafios enfrentados.

Inicialmente, foi necessário consolidar os conceitos associados ao CUDA e configurar um acesso a Rede Linux, disponível no Instituto de Matemática e Estatística, para termos acesso a uma GPU. Após, tornou-se possível iniciar o desenvolvimento do código paralelizado.

2.1 Paralelização com CUDA

As operações em CUDA são mais facilmente executadas em vetores unidimensionais. Assim, primeiramente, adaptamos as matrizes bidimensionais do código sequencial para que fossem representadas como vetores unidimensionais, mantendo a mesma localidade relativa entre os valores.

Com essa convenção, alteramos as funções `initialize` e `jacobi_iteration`, criadas na primeira tarefa, para acessarem os elementos como $g[i \times n + j]$ ao invés de $g[i][j]$. Além disso, a função `jacobi_iteration` foi renomeada para `jacobi_iteration_sequential`, concluindo assim a implementação da rotina de cálculo da distribuição de calor apenas no *host*.

Baseando-nos no exemplo `5_vector_add.cu`, desenvolvemos uma versão paralela da iteração de Jacobi. Assim, implementamos a função `__global__ jacobi_iteration`, que calcula o valor de um único ponto da matriz g, verificando previamente se a posição é válida. A atualização de cada ponto $g_{i,j}$ segue a equação:

$$g_{i,j} = 0.25 \times (h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1})$$

Uma dificuldade significativa foi a sincronização das *threads*. O método `_syncthreads` do CUDA apenas sincroniza *threads* dentro de um mesmo bloco, enquanto, para realizar a operação $h_{i,j} = g_{i,j}$, todas as posições de g precisam ser previamente calculadas. Para resolver esse problema, realocamos o laço de iteração para a função `main` e, dentro do laço, utilizamos chamadas ao kernel seguidas por `cudaDeviceSynchronize()` para garantir a finalização do processamento na GPU antes de prosseguir. Além disso, realizamos trocas de *buffers* entre as matrizes h e g a cada iteração. Com isso e seguindo as instruções dos slides de CUDA, o código está pronto para paralelização.

2.2 Configuração de Grade e Blocos

Iremos definir o número de threads por bloco como parâmetros ajustáveis. Para concluir a implementação, calculamos automaticamente o número de blocos b com base no valor de t recebido como entrada: $b = (n + t)/t$.

Assim, o número total de *kernels* é $(n + t)^2 > n$, garantindo uma quantidade mais que o suficiente para cobrir todos os elementos da matriz. Com isso, definimos grades e blocos bidimensionais e concluímos a implementação da iteração de Jacobi com suporte à execução na GPU.

2.3 Verificar a corretude

Após a efetuação de cálculos sem erros de compilação evidentes, foi desenvolvida uma função que verifica se os cálculos da CPU e GPU produziam o resultado correto. Essa etapa foi feita sem dificuldades.

2.4 Tempo de execução

Com o auxílio da função `clock_gettime()`, mensuramos o tempo dos seguintes eventos:

- Movimentação de dados do *host* para o *device*;
- Movimentação de dados do *device* para o *host*;

- Cálculo da iteração de jacobi feito na CPU;
- Cálculo da iteração de jacobi feito na GPU.

2.5 Cálculo de *speedup*

Calculamos o *speedup* considerando os resultados de desempenho de um algoritmo paralelizado utilizando CUDA em comparação com sua versão sequencial. A fórmula utilizada para o cálculo é dada por:

$$\text{speedup} = \frac{T_{sequencial}}{T_{paralelo}}$$

Foram testadas diferentes configurações de tamanho de entrada (n) e threads por bloco (t), variando também o número de blocos (b) automaticamente. Além disso, o número de iterações foi fixado como 1000 para todos os testes.

n	t	Tempo Sequencial (ms)	Tempo Paralelo (ms)	Speedup
10	8	0.57	5.63	0.18
10	16	0.55	5.53	0.18
10	32	0.57	5.60	0.18
100	8	46.75	5.97	7.83
100	16	47.44	6.11	7.77
100	32	48.75	6.69	7.29
1000	8	4765	62.60	76.06
1000	16	4718	60.77	77.63
1000	32	4712	81.43	57.87

Tabela 1: Tabela de tempos sequenciais e paralelos com o speedup correspondente para a Tarefa 2.

3 Tarefa 3

Para a Tarefa 3, criamos uma cópia do código desenvolvido na tarefa anterior e modificamos-o para computar a distribuição de calor dentro do quarto considerando um corpo quente com temperatura fixa de 37°C, em alguma posição fixada e com tamanho adequado.

Inicialmente, definimos `BODY_TEMP` como o valor 37.0 e definimos a coordenada do corpo como `BODY_X` e `BODY_Y`.

Em seguida, foi necessário alterar a função `jacobi_iteration` para que verificasse se dada posição era a coordenada do corpo ou um dos pontos de cálculo, o que foi feito com uma condicional. Aqui, um desafio enfrentado foi calcular a posição relativa do corpo no grid, o que pode ser feito com:

$$\text{X\_Relativo} = \frac{(\text{BODY\_X} * n)}{\text{ROOM\_SIZE}} \qquad \text{Y\_Relativo} = \frac{(\text{BODY\_Y} * n)}{\text{ROOM\_SIZE}}$$

De maneira análoga, a função `jacobi_iteration_sequential` também precisou ser alterada para que verificasse o ponto da iteração, tal como explicado anteriormente. Ainda, a função `initialize` precisou ser alterada para que inicializasse as paredes e a posição do corpo de forma adequada no grid.

No restante, o código manteve-se inalterado, uma vez que a lógica de resolução é a mesma implementada na tarefa anterior.

3.1 Comparativo entre versões

Primeiramente, iremos verificar os tempos de execução da versão sequencial e da versão paralelizada do algoritmo. Assim como na Tarefa 2, foram testadas diferentes configurações de tamanho de entrada (n) e threads por bloco (t), variando também o número de blocos (b) automaticamente. Além disso, o número de iterações foi fixado como 1000 para todos os testes.

n	t	Tempo Sequencial (ms)	Tempo Paralelo (ms)	Speedup
10	8	0.64	5.46	0.12
10	16	0.63	5.53	0.11
10	32	0.51	5.63	0.09
100	8	50.82	5.97	8.51
100	16	53.01	6.11	8.68
100	32	51.66	6.75	7.65
1000	8	5173	62.61	82.62
1000	16	5179	60.83	85.14
1000	32	5167	82.58	62.57

Tabela 2: Tabela de tempos sequenciais e paralelos com o speedup correspondente para a Tarefa 3.

Agora, considere a tabela abaixo, que compara os speedups de ambas tarefas.

n	t	Speedup Tarefa 2	Speedup Tarefa 3
10	8	0.18	0.12
10	16	0.18	0.11
10	32	0.18	0.09
100	8	7.83	8.51
100	16	7.77	8.68
100	32	7.29	7.65
1000	8	76.06	82.62
1000	16	77.63	85.14
1000	32	57.87	62.57

Tabela 3: Tabela comparativa dos speedups da Tarefa 2 e da Tarefa 3.

De maneira relativa, os speedups de ambas versões apresentam valores próximos, o que indica uma eficiência similar para os dois algoritmos. De fato, as implementações são praticamente iguais, com alterações apenas no corpo quente do quarto, o que mostrou não apresentar grande impacto no tempo de execução.

Por fim, vamos verificar os diagramas resultantes de cada algoritmo. Para exemplificar, vamos considerar valores fixados em número de pontos como 100, limite de iterações como 1000 e número de threads por bloco como 32.

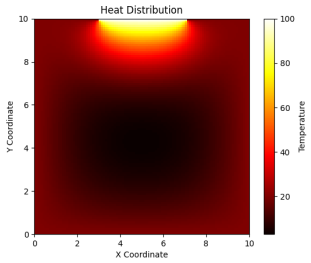

Figura 1: Diagrama da Tarefa 2

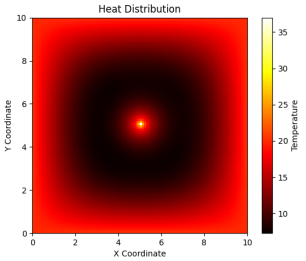

Figura 2: Diagrama da Tarefa 3, com posição central do corpo

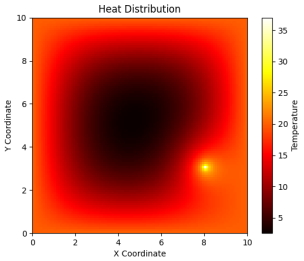

Figura 3: Diagrama da Tarefa 3, com posição arbitrária do corpo

É interessante notar que, na versão da Tarefa 2, a distribuição do calor pelo grid ocorre de forma mais intensa, uma vez que o corpo quente apresenta maior extensão e maior temperatura. Além disso, a temperatura da parede apresenta interferência menor na distribuição do calor na região perto do corpo, mas com crescente relevância para pontos distantes.

Já na versão da Tarefa 3, o corpo quente apresenta um impacto pequeno na distribuição, enquanto que as paredes apresentam impacto significativo. Tal comportamento era esperado, uma vez que o corpo quente, nesta versão, apresenta dimensões menores e temperatura mais amena. Ainda, tal comportamento repete-se independentemente da posição do corpo, como exemplificado pelas figuras 2 e 3.

4 Conclusão

Vamos considerar cada tarefa individualmente, com exceção da Tarefa 1, disponibilizada pelo Professor.

Para a Tarefa 2, as dificuldades encontradas durante o desenvolvimento foram superadas com base em exemplos e materiais de estudo, permitindo que as alterações necessárias fossem realizadas de forma progressiva. A principal adaptação foi a transformação de matrizes em vetores unidimensionais e a reorganização do código sequencial para suportar a paralelização com CUDA.

Os resultados mostraram que, para entradas pequenas ($n = 10$), a sobrecarga de inicialização e movimentação de dados na GPU faz com que a versão sequencial seja mais eficiente. No entanto, à medida que o tamanho da entrada aumenta ($n = 100$ e $n = 1000$), a versão paralela começa a superar a sequencial, alcançando speedups significativos, especialmente com configurações adequadas de blocos e threads. Observou-se ainda que, quando o número total de threads lançadas ($b^2 \times t^2$) é muito superior ao número de elementos na matriz (n^2), ocorre um desperdício de recursos computacionais, aumentando o tempo de execução da versão paralela. Assim, o ajuste cuidadoso dos parâmetros de paralelização é essencial para maximizar o desempenho.

Em resumo, a versão paralela mostrou-se vantajosa para entradas maiores, ou seja, a adaptação do algoritmo sequencial para suportar CUDA foi concluída com sucesso.

Já para a Tarefa 3, a alteração necessária para implementar o novo algoritmo mostrou-se simples, uma vez que já possuíamos um código resultante da Tarefa 2. Assim, as mudanças necessárias foram locais, como redefinição de variáveis e inclusão de condicionais para verificação de posição do corpo quente. Além disso, o código em paralelo mostrou-se mais eficiente do que a versão sequencial, o que era esperado, com um ganho em speedup crescente com o aumento do tamanho de entrada.

Ainda, ambos códigos da Tarefa 2 e Tarefa 3 apresentaram speedup similar, o que também era esperado, uma vez que as alterações de um algoritmo ao outro são mínimas. Ademais, os diagramas de cada tarefa apresentaram diferenças no impacto do corpo na distribuição do calor, mas também apresentaram similaridades, com uma maior quantidade de calor ao redor do corpo e com uma gradual redução com o distanciamento da fonte de calor, sendo que as paredes possuíram impacto considerando os pontos distantes do corpo.

Em conclusão, as tarefas foram realizadas de maneira completa e eficiente, atingindo os objetivos de desenvolvimento e o enfrentamento dos desafios da linguagem e da Equação de Calor em Estado Estacionário.