Paralelização da Soma de Progressão Harmônica com OpenMP e CUDA

Pedro C. Coleone[1], Pietro M. Moralles[1], Carlos H. R. Matos Filho[1], Hélio C. Guardia[1]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCAR)
Caixa Postal 676 – 13565-905 – São Carlos – SP – Brazil

{pedrocoleone, morallespietro, carlosmatos}@estudante.ufscar.br

Abstract. *This article examines parallel implementations of the harmonic progression sum problem using OpenMP in a multiprocessor shared memory environment and CUDA for GPU processing. The performance of these implementations is evaluated through speedup and efficiency metrics, highlighting runtime gains that demonstrate the importance of parallelization techniques. Calculation partitioning and partial result aggregation were essential to achieve good speedup, especially with the GPU, by avoiding unnecessary data copies between GPU memory and RAM.*

Resumo. *Este artigo examina implementações paralelas do problema da soma de progressão harmônica, usando OpenMP em um ambiente multiprocessado com memória compartilhada e CUDA para processamento em GPU. O desempenho dessas implementações é avaliado por meio de métricas de speedup e eficiência, destacando ganhos nos tempos de execução que comprovam a importância das técnicas de paralelização. A partição de cálculos e a agregação de resultados parciais foram essenciais para alcançar um bom speedup, especialmente com a GPU, ao evitar-se cópias desnecessárias de dados entre a memória da GPU e a RAM.*

1. Introdução

A soma harmônica, fundamental na matemática, envolve a adição dos recíprocos dos números naturais [Knuth 1968]. Assim como toda operação computacionalmente intensiva, a redução de seu tempo de execução é relevante para o bom desempenho nas aplicações onde é utilizada. Este estudo explora abordagens computacionais para calcular essa soma, destacando o impacto da paralelização no desempenho dos cálculos. Para isso, são analisados recursos providos por OpenMP, para paralelização em sistemas multicore, e CUDA, para aproveitar o poder de processamento em GPU. A viabilidade da paralelização se dá pela independência das operações em cada termo da sequência harmônica, possibilitando distribuir o trabalho entre múltiplos processadores e realizar agregações de forma consistente, o que possibilita reduzir o tempo de execução.

2. Metodologia

Como referência para o problema da soma harmônica, utilizou-se o código fonte apresentado na 16ª Maratona de Programação Paralela [Bianchini and Pillon 2021]. Neste problema, que visa realizar as somas envolvidas mantendo a precisão até a 100ª casa

decimal, utiliza-se um vetor onde são salvos os valores de cada dígito. A partir do problema original, foram implementadas versões paralelas do algoritmo de soma de progressão harmônica, utilizando OpenMP e CUDA. Nesses códigos, uma seção relevante para paralelização é o *loop* principal da função **sum**, onde os cálculos dos dígitos são realizados.

Listagem 1. Código sequencial para cálculo da soma de progressão harmônica.

```
1: for (long unsigned int i = 1; i <= n; ++i) {
2:   long unsigned int remainder = 1;
3:   for (long unsigned int digit = 0; digit < d + 11 && remainder; ++digit) {
4:     long unsigned int div = remainder / i;
5:     long unsigned int mod = remainder % i;
6:     digits[digit] += div;
7:     remainder = mod * 10;
8:   }
9: }
```

Cada ciclo desse *loop* opera de maneira independente, sendo adequado para distribuição entre várias *threads*, o que acelera o cálculo ao explorar o paralelismo dos sistemas multicore. No entanto, para isso, é necessário replicar o vetor de somas parciais para cada tarefa e agregá-los de forma consistente. Quanto à complexidade do código, ela é *O(n.d)*, onde **n** é o número de iterações do primeiro *loop* e **d** é o limite do segundo *loop*.

Na solução adotada, há uma divisão das iterações no cálculo dos valores associados aos vários dígitos, utilizando a diretiva `parallel for`. Para evitar as operações de sincronização na agregação dos resultados, usou-se o operador de redução. Deste modo, cada *thread* possui sua própria instância do vetor de soma, sobre o qual realiza suas somas parciais, sendo que o resultado final é agregado pela operação de soma, posição a posição, ao final do *loop*. Após a paralelização, a complexidade passa ser de *O(n/p.d)*, sendo *p* = número de tarefas.

Listagem 2. Trecho relevante do código paralelizado com OpenMP.

```
1: #pragma omp parallel for reduction(+:digits[:d+11])
2: for (long unsigned int i = 1; i <= n; ++i) {
3:   long unsigned int remainder = 1;
4:   for (long unsigned int digit = 0; digit < d + 11 && remainder; ++digit) {
5:     long unsigned int div = remainder / i;
6:     long unsigned int mod = remainder % i;
7:     remainder = mod * 10;
8:   }
9: }
```

Ao aplicar a paralelização com CUDA, é essencial o tratamento de zonas críticas para garantir a integridade dos dados. O trecho de código exemplifica a função *kernel* `sum()`, que calcula a soma da progressão harmônica de forma paralela. Uma otimização relevante que se pode observar no código é o uso de uma área de memória compartilhada entre as *threads* de cada bloco para a redução da sobrecarga de acesso à memória global.

Listagem 3. Trecho relevante (*kernel*) do código CUDA para a paralelização.

```
1: __global__ void sum(unsigned long long int *digits, long unsigned int d, long unsigned int n)
2: {
3:   int idx = threadIdx.x + blockIdx.x * blockDim.x;
4:   extern __shared__ unsigned long long int shared_digits[];
5:   for (unsigned int digit = threadIdx.x; digit < d + 11; digit += blockDim.x)
6:     shared_digits[digit] = 0;
7:   __syncthreads();
```

```
8:  if (idx < n) {
9:      unsigned long long int remainder = 1;
10:
11:     for (unsigned int digit = 0; digit < d + 11 && remainder; ++digit) {
12:         unsigned long long int div = remainder / (idx + 1);
13:         unsigned long long int mod = remainder % (idx + 1);
14:         atomicAdd(&shared_digits[digit], div);
15:         remainder = mod * 10;
16:     }
17:  }
18:  __syncthreads();
19:  for (unsigned int digit = threadIdx.x; digit < d + 11; digit += blockDim.x)
20:      atomicAdd(&digits[digit], shared_digits[digit]);
21: }
```

Além de considerar o dimensionamento correto dos blocos de *threads* e a alocação de recursos na GPU, é essencial analisar o hardware e as características do problema para otimizar os parâmetros de tamanho de bloco e número de blocos. Quanto à complexidade do código CUDA, ela depende da quantidade de *threads* executadas simultaneamente na GPU. Embora os *loops* sejam paralelizados, a sincronização implícita entre as *threads* e a sobrecarga das operações *atomicAdd* podem resultar em uma complexidade aproximada de *O(n * d)*, considerando um número suficiente de *threads* para compensar essa sobrecarga.

3. Resultados

Os testes foram realizados em um ambiente de hardware com configuração específica, variando o número de *threads* ou blocos conforme necessário. Os parâmetros de execução, como o tamanho da sequência e a precisão dos resultados, foram ajustados para avaliar o desempenho em diferentes cenários. Os resultados apresentados aqui foram obtidos para cálculos de 10.000.000 elementos e precisão de 100 casas decimais.

A análise dos resultados revela a vantagem das implementações paralelas. Enquanto a execução sequencial demandou cerca de 19,85 segundos, a versão OpenMP concluiu em aproximadamente 1,01 segundos, e a implementação CUDA foi ainda mais eficiente, com apenas 0,18 segundos. Essa diferença destaca os benefícios da paralelização para otimizar o desempenho em cargas computacionais intensivas.

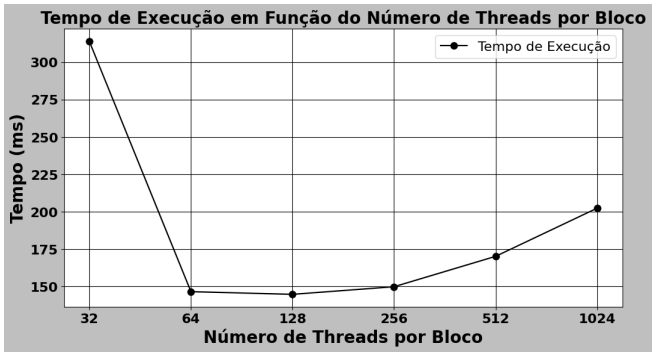

Figura 1. Tempo de execução com CUDA, em função do número de *threads* por bloco

Como se observa na Figura 1, o número de *threads* por bloco influencia diretamente a maneira como o dispositivo GPU organiza e executa as *threads*, enquanto o número de blocos determina como o problema é dividido entre os multiprocessadores

da GPU. Nas medidas de tempo realizadas, que incluem não só o processamento mas também cópias em memória de e para a GPU, os melhores desempenhos form obtidos com 64, 128 e 256 *threads* por bloco, o que corresponde, respectivamente, a 2, 4 e 8 rodadas de escalonamento de *warps* dentro de cada SM. Os experimentos com CUDA correspondem aos valores médios da execução com uma GPU NVIDIA T4 utilizada na plataforma Colab.

Na Figura 2, vê-se os resultados coletados para o programa OpenMP, obtidos a partir da média de nove execuções em um computador com processador Intel Xeon Silver 4208 com 16 cores físicos e 32 cores lógicos. Dada a granularidade das operações paralelas realizadas por cada *thread*, o pico de desempenho correspondeu ao número de processadores (*cores*/*threads*) disponíveis no sistema utilizado, sem benefícios com o uso de mais *threads* lógicas.

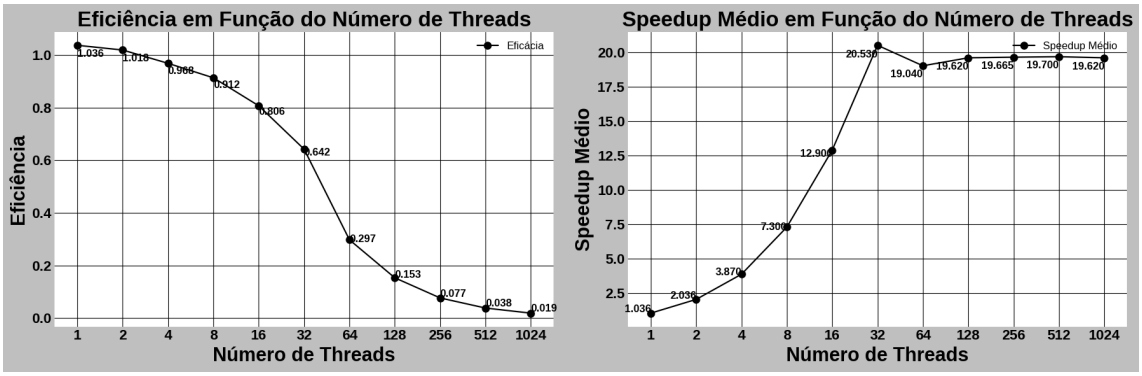

Figura 2. *Speedup* e Eficiência da implementação paralela utilizando OpenMP

4. Conclusão

As implementações paralelas com OpenMP e CUDA mostraram melhorias no desempenho em comparação com a abordagem sequencial. Os resultados enfatizaram a importância das otimizações de código e ajustes nas técnicas de paralelização, especialmente com CUDA, ao evitar cópias em memória e usar uma área de memória compartilhada entre as *threads* de cada bloco, o que reduziu a competição pelo acesso à memória compartilhada da GPU. Na implementação com OpenMP, o uso do operador de redução, realizando a agregação de somas parciais, também proporcionou melhores resultados do que o acesso concorrente a um vetor compartilhado.

Referências

Documentação oficial do CUDA Toolkit. NVIDIA.

Documentação oficial do OpenMP. OpenMP Architecture Review Board.

Bianchini, C. and Pillon, M. A. (2021). 16th marathon of parallel programming: Rules for remote contest. In *Proceedings of the 16th Marathon of Parallel Programming*, pages 1–1, Brazil. SBAC-PAD & WSCAD.

Knuth, D. E. (1968). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley.