Explorando Paralelismo no Problema da Subsequência de Soma Máxima

Lucas Sciarra Gonçalves, Murilo Miranda, Hélio Guardia

Departamento de computação– Universidade Federal de São Carlos (UFSCAR)

{lucassciarra, murilo.miranda}@estudanfe.ufscar.br

Abstract. *This paper explores parallelization techniques applied to an algorithm addressing the problem of computing the maximum sum of an increasing subsequence of length K within a vector of N elements. The main challenges discussed revolve around determining the optimal parallelization strategy and managing the overheads introduced by the implementations. Consequently, versions utilizing OpenMP and CUDA were developed, with the latter exhibiting more efficient performance, attributed to the enhanced functionality provided by its architecture.*

Resumo. *Este artigo explora formas de paralelização sobre um algoritmo para o problema da soma máxima de uma subsequência crescente de tamanho K em um vetor de N elementos. A dificuldade de encontrar a melhor estratégia de paralelismo, e lidar com os overheads que as implementações geram são os pontos principais abordados. Por isso, foram feitas as versões em OpenMP e em CUDA, onde esta última teve um desempenho mais rentável, por conta de funções mais eficientes que a arquitetura proporciona.*

1. Introdução

A eficiência em otimização combinatória é vital na computação, com aplicações em análise de dados, IA, reconhecimento de padrões [Bentley 1984], biologia computacional, tais como identificação de genes [Lima 2015]. Este trabalho aborda a soma máxima de uma subsequência crescente de tamanho k em um array $[A_1, A_2, \ldots, A_n]$, onde $S_1 \leq S_2 \leq \ldots \leq S_k$, uma variação do problema "Longest Increasing Subsequence" [Fredman 1975]. A partir de um algoritmo sequencial que utiliza técnicas de programação dinâmica [SBAC-PAD WSCAD 2019], avaliamos otimizações com OpenMP e CUDA, propondo uma solução eficaz e *insights* para problemas semelhantes.

2. Análise da Implementação Sequencial

O algoritmo sequencial em C utiliza programação dinâmica para calcular a soma máxima de uma subsequência de tamanho k. A abordagem define a matriz $dp[i][j]$ como a soma máxima de uma subsequência de tamanho j terminando em i, atualizando os valores com a transição $dp[i][l+1] = max\ (dp[i][l+1], dp[j][l] + arr[i])$ *para* $arr[j] < arr[i]$. A complexidade temporal é $O(n^2 \cdot k)$, reduzida para $O(n^2)$ se k for constante, enquanto o espaço requerido é $O(n \cdot k)$. Essa complexidade torna a solução inviável para $n > 10^6$. Assim, a exploração de paralelismo no problema torna-se uma abordagem promissora, especialmente com o uso de tecnologias como OpenMP e CUDA [Colichio et Al]. Essas ferramentas permitem otimizar a execução, aproveitando a capacidade de processamento paralelo de CPUs e GPUs, respectivamente, o que pode resultar em ganhos significativos de desempenho para instâncias computacionalmente intensivas.

Código 1: Implementação do algoritmo sequencial

```
int MaxIncreasingSub(int arr[], int n, int k) {
  for (int i = 1; i < n; i++) {
    // #pragma omp parallel for
    for (int j = 0; j < i; j++)
      if (arr[j] < arr[i])
        for (int l = 1; l < k; l++)
          if (dp[j][l] != -1)
           // #pragma omp critical
            dp[i][l+1]=(dp[i][l+1]>dp[j][l] + arr[i]) ? dp[i][l+1] :
 (dp[j][l]+arr[i]);

    int ans = -1;
    for (int i = 0; i < n; i++)
        if (ans < dp[i][k]) ans = dp[i][k];
    return (ans == -1) ? 0 : ans;}
```

3. Análise das Implementações Paralelas

Ao analisar o código, foi definida como estratégia a paralelização do *loop* intermediário (*loop* externo *j*), pois o *loop* mais externo (*i*) apresenta dependência de dados entre suas iterações, enquanto o *loop* mais interno (*k*) geralmente envolve poucas iterações, tornando-o menos vantajoso para paralelização. Cada iteração de *j* é limitada pelo tamanho de *i* (já que ele itera de 0 até *i*-1) e, como está dentro do *loop i*, o seu número de iterações irá crescendo cada vez mais. Porém, mesmo com valores pequenos nas primeiras iterações de *j*, esta ainda é uma opção válida a ser paralelizada. O grande problema está na dependência de dados no acesso à matriz *dp[i][l+1]*, de forma que se faz necessária a atualização concorrente da variável, para que não haja *race conditions,* gerando valores incorretos. A versão com OpenMP usa *#pragma omp parallel for*. Contudo, a contenção no acesso à memória compartilhada devido à seção crítica (*#pragma omp critical*), necessária para atualizar *dp[i][l+1]*, introduz *overhead*, reduzindo drasticamente o desempenho e a escalabilidade.

Na versão CUDA, cada *thread* processa um índice *j* para um dado *i*, com o *kernel computeDP* sendo invocado iterativamente. As iterações de *j* (com $j < i$) são distribuídas entre *threads*, mas isso limita o paralelismo ao tamanho de *i* e pode causar *race conditions* na atualização do *array dp*. Para evitar isso, utiliza-se *atomicMax*, garantindo atualizações seguras em *dp[i *(k + 1)+(l + 1)]* com mínimo *overhead*.

Uma otimização importante envolve o uso de memória compartilhada e centralização das atualizações atômicas em uma única *thread* por bloco. Com 64 *threads* por bloco, os dados de *arr[j]* e *dp[j*(k+1)+l]* são carregados na memória compartilhada, reduzindo acessos à memória global. As *threads* atualizam o array *shared_results* com *atomicMax*, e a *thread* 0 aplica as atualizações finais em memória global, reduzindo a concorrência e melhorando a eficiência. Apesar dos ganhos, a abordagem exige cuidado com o uso de memória. A memória compartilhada, limitada a aproximadamente 48 KB por bloco, comporta bem casos de até k=100, mas pode ser insuficiente para *k* muito altos, tornando-a menos escalável que a implementação original. Nesses casos, é necessário reduzir o número de *threads* por bloco ou reestruturar o *kernel*. Ainda assim,

a combinação de memória compartilhada e atualizações atômicas centralizadas oferece bom desempenho e corretude para a maioria dos cenários práticos.

Código 2: Implementação do Kernel em CUDA (sem *shared memory*)

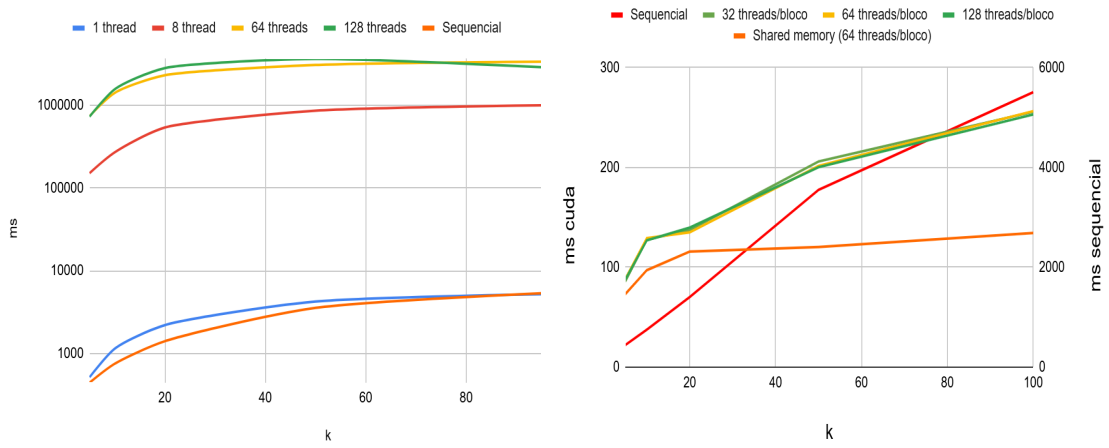
```
__global__ void computeDP(int *arr, int *dp, int n, int k, int i) {
  int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (j >= i) return;
    if (arr[j] < arr[i])
        for (int l = 1; l <= k - 1; l++)
            if (dp[j * (k + 1) + l] != -1) {
                int new_val = dp[j * (k + 1) + l] + arr[i];
                atomicMax(&dp[i * (k + 1) + (l + 1)], new_val); }

  //A chamada do kernel
  int threadsPerBlock = 32;
  int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
  for(int i = 1; i < n; i++) {
    computeDP<<<blocksPerGrid, threadsPerBlock>>>(d_arr,d_dp,n,k,i);
        cudaDeviceSynchronize();}
```

4. Resultados

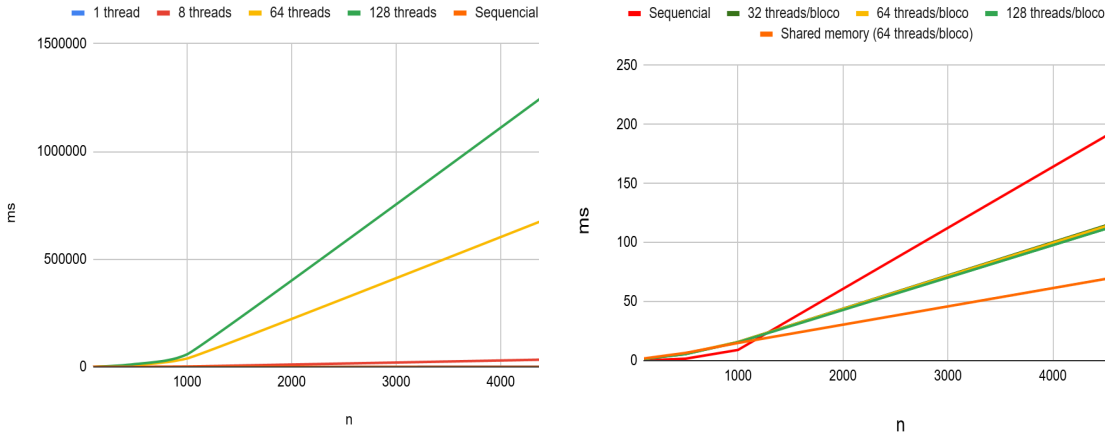
A implementação $O(n^2.k)$ foi avaliada em dois ambientes: uma CPU Intel Xeon E7-4870 (4 *sockets*, 10 núcleos/*socket*, 2 *threads*/núcleo, 256GB RAM) com OpenMP, e uma GPU Tesla T4 (16GB VRAM) no Google Colaboratory com CUDA. Com tamanho fixo e subsequência variável, o tempo sequencial cresce linearmente, enquanto OpenMP sofre com sobrecarga de seções críticas, perdendo desempenho com mais *threads*. CUDA tem tempos menores, especialmente para n maior, pelo melhor uso do paralelismo massivo. Com subsequência fixa e n variável, o tempo sequencial cresce quadraticamente, OpenMP mostra pouca vantagem ou piora, e CUDA mantém alta eficiência. A variação no tamanho do grid não trouxe ganhos relevantes, pois o gargalo está no crescimento de i.

Figura 1: (a) Tempos das versões Sequencial x OpenMP, (b) Sequencial x CUDA, com N = 5000 e K variável



(a) **(b)**

Figura 2: (a) Tempos das versões Sequencial x OpenMP, (b) Sequencial x CUDA, com K = 10 e N variável



(a) **(b)**

5. Conclusão

A solução sequencial é eficiente ao podar implicitamente caminhos não promissores, garantindo ótimo desempenho prático. Contudo, dentre as paralelizações, CUDA sobressai ao explorar o paralelismo massivo de GPUs, superando tanto a versão sequencial quanto a implementação OpenMP, especialmente para números com muitos dígitos. OpenMP, embora mais simples, sofre com paralelismo limitado e overhead de seções críticas, tornando-se menos eficiente que o sequencial. Já CUDA exige cuidado com memória e sincronização, mas se mostra altamente escalável em *hardware* adequado. Com o uso de *shared memory*, há redução dos acessos à memória global, ao concentrar os valores temporários e os resultados parciais dentro de cada bloco, é possível minimizar a contenção e acelerar as operações atômicas.

6. Referências

Bentley, J. (1984). Programming pearls: Algorithm design techniques. Communications of the ACM, 27(9), 865-873.

Colichio, H., Pimenta, Y., Borges, P., Menecucci, M., Barros, L., & Guardia, H. (2023). Ordenação paralela com OpenMP e CUDA. In Anais da Escola Regional de Alto Desempenho de São Paulo (ERAD-SP). https://sol.sbc.org.br/index.php/eradsp/article/view/28722

Fredman, M. L. (1975). On computing the length of longest increasing subsequences. Discrete Mathematics, 11(1), 29-35. https://doi.org/10.1016/0012-365X(75)90103-X

Lima, A. C. de. (2015). Soluções para os problemas da soma máxima e do k-ésimo menor elemento de uma sequência usando o modelo BSP/CGM. Universidade Federal de Mato Grosso do Sul, Campo Grande, MS, Brasil.

International Symposium on Computer Architecture and Simpósio em Sistemas Computacionais de Alto Desempenho High Performance Computing. 14th marathon of parallel programming sbac-pad wscad.