Análise de Paralelismo em Algoritmo de Método Numérico com OpenMP

Thiago P. M. Dardis[1]

[1]Departamento de Engenharia de Materiais – Universidade Federal de São Carlos (UFSCar)
São Carlos, SP – Brazil

thiagopmaffeid@gmail.com

Abstract. *The presente work aims to analyze the scalability of the execution of a numerial calculation algorithm, calculating a simple integral via the Trapezoid Method and analyzing the execution time of the code executed in an Intel Core i7-3770k CPU in a parallelized way via OpenMP and without parallelization. In addition, the execution on na Intel UHD Graphics P630 GPU via Devcloud was analyzed.*

Resumo. *O presente trabalho tem por objetivo analisar a escalabilidade da execução de um algoritmo de cálculo numérico, calculando uma integral simples via Método do Trapézio e analisando o tempo de execução do código em uma CPU Intel Core i7 – 3770k de forma paralelizada via OpenMP e sem paralelização. Além disso, analisou-se a execução em uma GPU Intel UHD Graphics P630 via Devcloud.*

1. Introdução

Dentre as principais ferramentas matemáticas, destaca-se o Cálculo Diferencial e Integral. Nessa temática, inúmeros métodos numéricos existem para que se possa calcular a integral de uma determinada função. Dentre eles, destaca-se o Método do Trapezio, que consiste no cálculo da área de um trapézio embaixo de uma determinada curva entre dois pontos.

Para se obter uma maior precisão da área aproximada embaixo da curva, maior é o número de trapézios necessários embaixo dela, ou seja, as partições. Entretanto, para funções mais complexas e aplicações que necessitem de alta velocidade de execução, a programação tradicional encontra seus limites.

Nesse contexto, surge o interesse em utilizar a programação paralela, que consiste em otimizar o tempo de execução de um determinado algoritmo por meio de estratégias de hardware, como executar múltiplas tarefas ao mesmo tempo.

Dentre as inúmeras maneiras de paralelizar um código, o presente trabalho utilizou do OpenMP, em virtude da facilidade da implementação e do potencial de ganho de tempo com essa técnica. O OpenMP é uma interface de programação (API), baseada no modelo de programação paralela de memória compartilhada para arquiteturas de múltiplos processadores (Lee and Eigenmann, 2013).

Mais precisamente, o OpenMP consiste em um modelo de execução tipo fork-join, em que se inicia a execução com apenas um processo (master thread) e, no ínicio

do construtor paralelo, cria-se um time de threads (fork), podendo se especificar a quantia visando uma maior otimização. Ao completar a execução, o time de threads se sincroniza em uma barreira implícita (join) e, logo após, apenas a master thread continua a execução (Ayguadé *et al.*, 2009).

Para fins didáticos, o presente trabalho utilizou a função seno ao quadrado de x para testar o algoritmo numérico e, então, utilizando a biblioteca <chrono>, obteve-se o tempo de execução para o cálculo da integral de 0 à 90 graus, somando 1.000.000 de partições, visando uma maior aproximação do valor real. Para a análise, executou-se o código em um device local com um Processador i7 – 3770k com 4 cores e 2 threads por cada, resultando em, no máximo, 8 threads para a execução paralela. Além disso, também utilizou-se a Devcloud, onde foi possível executar o código em uma GPU (Graphics Processing Unit) Intel UHD Graphics P630.

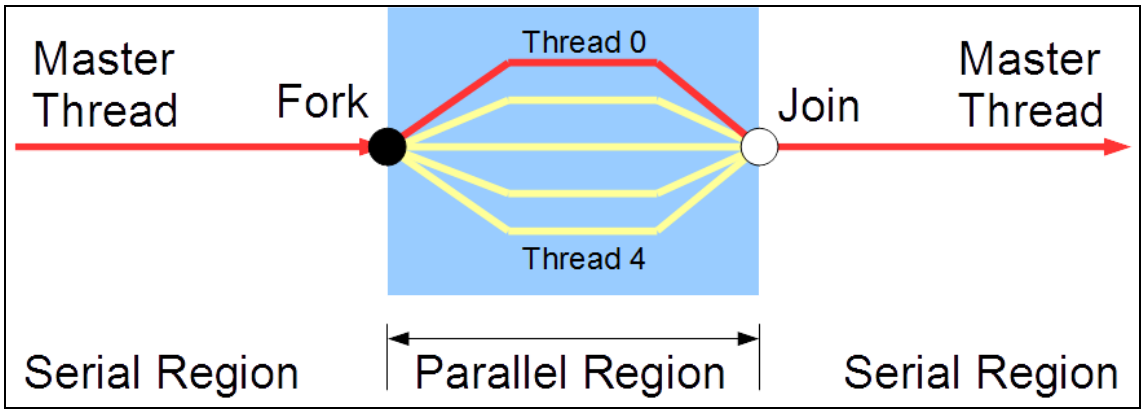

Figura 1. Representação do modelo "Fork-Join" do OpenMP

2. Resultados e Discussão

Para que se pudesse efetuar o cálculo da integral e, logo após, paralelizar o código, além de analisar o tempo, foi necessário importar as bibliotecas <math.h>, <chrono> e <omp.h>. Após executar o código por 5 vezes na CPU (Central Process Unit), utilizando a IDE Codeblocks, fez-se uma análise estatística e, então, obteve-se a média dos valores.

Logo após, paralelizou-se o trecho do código que consta na Figura 2 e repetiu-se os procedimentos estatísticos supramencionados. É válido ressaltar a simplicidade em termos de síntaxe para que se possa paralelizar um código utilizando o OpenMP, o que torna essa uma ferramenta altamente versátil.

```
std::chrono::steady_clock::time_point start(std::chrono::steady_clock::now());
#pragma omp parallel for num_threads (8)
for (i=0;i<n; i++)
{
    xb=xa+incr;

    I=I + (xb-xa)*(f(xa)+f(xb))/2.0f;

    xa=xb;
}
```

Figura 2. Paralelização do loop for com OpenMP e número de Threads de 8

Por fim, para que o código fosse executado na Devcloud, foi necessário instalar o Cygwin, um terminal linux e, com o comando ssh devcloud, conectar a rede. Logo após, foram criados três arquivos, um build.sh para se conectar ao servidor, um Makefile para compilar o code.cpp, onde consta o código. Logo após, executou-se em um código de uma GPU.

Mais especificamente ao código, algumas adaptações foram necessárias. Visando manter a homogeneidade dos parâmetros iniciais, fez-se necessário fixar os valores de 0 e 90 graus, já que, na Devcloud, não é possível utilizar os commandos cout e cin para obter os extremos de integração desejados pelo usuários. As alterações constam na Figura 3. E, após essas adptações, também executou-se o algoritmo 5 vezes para que se pudesse obter uma média do tempo de execução em uma GPU.

```
int main (int argc, char *argv[])
{
    double xa,xb;
    double I, incr;
    int i,n;
    cout<<"Bem vindo ao integrador via Metodo do Trapezio.\n\nVamos determinar o
s intervalos de integracao.\n";
    xa=0*PI/180;
    xb=90*PI/180;
n=1000000;
    incr = (xb-xa)/n;
    I=0.0f;
```

Figura 3. Adaptação necessária para execução do Código na Devcloud. Definiu-se os extremos de integração como 0 e 90 graus, além de 1000000 de partições

Os resultados das execuções na CPU, considerando o número de threads sendo 1 para a execução sequencial (sem paralelizar) e as execuções variando o número de threads de 2^n tendo n variando entre 1 e 3 constam na Figura 4. É interessante ressaltar que o decaimento do tempo de execução é exponencial ao aumento do número de threads. Além disso, em virtude das limitações de hardware, não foi possível executar um número de threads maior que 8. Por fim, é interessante ressaltar que, nesse caso, não observou-se o fenômeno de overhead, consequência da divisão elevada de tarefas.

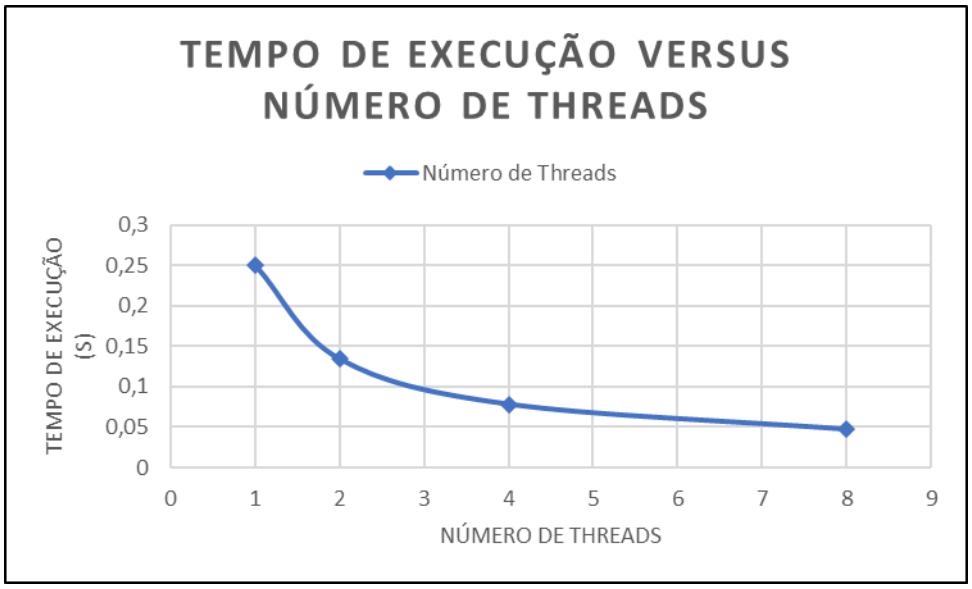

Figura 4. Gráfico do Tempo de Execução para número de threads variando de 2^0 até 2^3 com a paralelização OpenMP

Os resultados obtidos por todas as análises constam nas Tabelas 1 e 2. Além disso, calculous-se o Speedup, por meio do quociente entre o Tempo de execução sem paralelizar na CPU (referência) e o tempo analisado, tanto as execuções com threads variadas na CPU quanto a execução sequencial na GPU. Além do Speedup, também calculou-se a Eficiência, que seria o quociente entre o Speedup e o número de threads.

Tabela 1. Tempos de execução média para diferentes números de threads, bem como para o Código sem paralelizar. Além da Speedup e da Eficiência para cada caso de paralelismo.

	Sem paralelizar	Threads	OpenMP	Speedup	Eficiência
Tempo	0,251093	2	0,13437	1,868666	93,43%
Tempo	0,251093	4	0,07812	3,214185	80,35%
Tempo	0,251093	8	0,04687	5,357208	66,97%

Da análise dos resultados obtidos, é possível concluir que, para essa tarefa de integração, 8 threads obtiveram o melhor desempenho dentre as demais quantidades de threads e a GPU, apesar de não apresentar nenhum tipo de paralelismo, obteve uma excelente performance, ainda que menor que no caso da paralelização na CPU com 8 threads, conforme consta na Tabela 2. Tal fenômeno já era esperado, já que as GPUs possuem uma velocidade para execução de tasks muito maior quando comparada com CPUs.

Tabela 2. Tempos de execução médio para threads=8 executado na CPU e para o código sem paralelizar sendo executado na CPU e, também, na GPU. Também const ana tabela o Speedup comparando a execução paralela em CPU e em GPU

	CPU (local)			GPU	Speedup	
	Sem paralelizar	Threads	OpenMP	Sem Paralelizar	CPU	GPU
Tempo	0,251093	8	0,046869	0,06625318	5,35734	3,7899

3. Conclusão

Com base na discussão realizada na seção "Resultados e conclusão" é possível concluir que, em se tratando de paralelismo local, 8 threads desempenham de forma explêndida quando comparada com as demais quantias de threads analisadas e, além disso, não se observa o fenômeno de overhead nas circunstâncias em que o trabalho foi desenvolvido. Em oposição, há uma queda brusca da eficiência quando há o aumento do número de threads fato esse não tão importante nesta análise, pois objetiva-se uma maior velocidade na execução do código.

Por fim, quanto às GPUs, obteve-se resultados esperados que demonstram o grande potencial de aplicação desse tipo de microprocessador especializado em processor gráficos em aplicações mais diversas, como a do cálculo numérico. Dessa forma, conclui-se que as GPGPUs (General Purpose Graphics Processing Unit) possuem um amplo campo de atuação, já que deixam de possuir as aplicações específicas atribuídas às GPUs tradicionais.

Portanto, conclui-se que a implementação de paralelismo é altamente eficiente e, com o número de threads certo para uma determinada task, pode resultar em um speedup significativo. Fato esse que, aplicações específicas que dependam de uma baixa latência, além de uma alta velocidade de execução ou que sejam muito demoradas para serem executadas de forma sequencial, tornam-se campos pujantes para a aplicação do paralelismo.

4. Referências

Ayguadé, E. *et al.* (2009) 'The design of OpenMP tasks', *IEEE Transactions on Parallel and Distributed Systems*, 20(3), pp. 404–418. doi: 10.1109/TPDS.2008.105.

Lee, S. and Eigenmann, R. (2013) ‘OpenMPC: Extended OpenMP for efficient programming and tuning on GPUs’, *International Journal of Computational Science and Engineering*, 8(1), pp. 4–20. doi: 10.1504/IJCSE.2013.052110.