Relatório sobre a utilização de CUDA para integração dupla

Otávio Augusto Teixeira
Luan Marques Batista

IBILCE / UNESP - CÂMPUS DE SÃO JOSÉ DO RIO PRETO

Introdução

O método dos trapézios é uma técnica amplamente utilizada para a aproximação numérica de integrais, permitindo a resolução de integrais simples e duplas de forma eficiente. Neste relatório, aplicaremos o método dos trapézios para calcular uma integral dupla da função $f(x,y) = sin(x^2 + y^2)$, definida sobre a região $[0, 1.5] \times [0, 1.5]$, utilizando a plataforma de programação paralela CUDA

CUDA (Compute Unified Device Architecture) é uma plataforma de programação paralela desenvolvida pela NVIDIA que permite aproveitar a potência de processamento das GPUs (Unidades de Processamento Gráfico) para realizar cálculos complexos de forma muito mais rápida do que com as tradicionais CPUs. Ao invés de serem utilizadas apenas para renderizar gráficos, as GPUs com suporte a CUDA podem ser programadas para executar uma ampla variedade de tarefas, como simulações científicas, aprendizado de máquina, criptografia e muito mais. Essa tecnologia revolucionou o campo da computação de alto desempenho, tornando possível resolver problemas antes considerados intratáveis.

Para esse estudo, utilizei uma placa de vídeo Nvidia, GTX 1660, versão notebook, e instalei a ultima versão do cuda que eu encontrei na Nvidia.

Neste estudo, serão realizados testes com as seguintes configurações:

- **Quantidade de blocos**: 10, 100, 1000.
- **Quantidade de intervalos no eixo x**: 10^3 , 10^4 , 10^5
- **Quantidade de intervalos no eixo y**: 10^3 , 10^4 , 10^5

Ao final, o relatório apresentará o tempo de execução para cada cenário, utilizando tabelas e gráficos para visualizar o impacto do paralelismo e da granularidade dos intervalos. O código fonte da aplicação será entregue junto ao relatório, garantindo que os experimentos possam ser replicados.

Resultados

O código utiliza CUDA para calcular a integral pelo método dos trapézios, dividindo o trabalho entre múltiplos blocos e threads de forma eficiente. Ele aceita como parâmetros de entrada a quantidade de blocos, o número de intervalos de discretização nos eixos xxx e yyy, e utiliza a biblioteca CUDA para realizar os cálculos paralelamente.

A divisão do trabalho ocorre da seguinte forma:

1. **Blocos e Threads**:
   Cada bloco contém 512 threads, sendo responsável por processar uma parte dos intervalos. O número total de threads em execução é determinado pela multiplicação do número de blocos pelo número de threads por bloco. Com isso, cada thread processa múltiplos pontos na discretização.
2. **Cálculo Paralelo**:
   Cada thread é responsável por calcular os valores da função em uma fração dos pontos da malha de discretização. Esses cálculos consideram o peso de cada ponto (bordas, cantos ou pontos internos) no método dos trapézios.
3. **Redução Parcial e Soma Global**:
   Os resultados parciais calculados por cada thread são acumulados em memória compartilhada dentro do bloco. Após a redução dos valores no nível do bloco, uma soma global é realizada usando a função **`atomicAdd`** para combinar os resultados de todos os blocos.

Código usado para buildar:

nvcc -o trabalho4 Trabalho4.cu

Código usado para executar:

./trabalho4 <num_blocos> <num_eixo_x> <num_eixo_y>

Tabela de tempos obtida após a execução do código para cada caso acima:
Obs: Cada caso foi testado 5 vezes e, após isso, peguei a mediana para melhor representação dos dados

	Quantidades de Blocos		
Quantidade de intervalos eixo x e y	10 blocos	100 blocos	1000 blocos
$x = 10^3$ $y = 10^3$	0.156s	0.167s	0.175s
$x = 10^3$ $y = 10^4$	0.194s	0.238s	0.234s

$x = 10^3$ $y = 10^5$	0.512s	0.837s	0.848s
$x = 10^4$ $y = 10^3$	0.172s	0.165s	0.161s
$x = 10^4$ $y = 10^4$	0.250s	0.235s	0.225s
$x = 10^4$ $y = 10^5$	0.849s	0.854s	0.848s
$x = 10^5$ $y = 10^3$	0.241s	0.214s	0.203s
$x = 10^5$ $y = 10^4$	0.863s	0.668s	0.637s
$x = 10^5$ $y = 10^5$	7.012s	5.326s	4.985s

Tabela dos Speedups comparando a quantidade de blocos com os intervalos, a base do speedup foi 10 blocos do CUDA:

Speedups			
Quantidade de intervalos eixo x e y	10 blocos - Base	100 blocos	1000 blocos
$x = 10^3$ $y = 10^3$	1.0	0,93	0,89
$x = 10^4$ $y = 10^4$	1.0	1,06	1,11
$x = 10^5$ $y = 10^5$	1.0	1,31	1,40

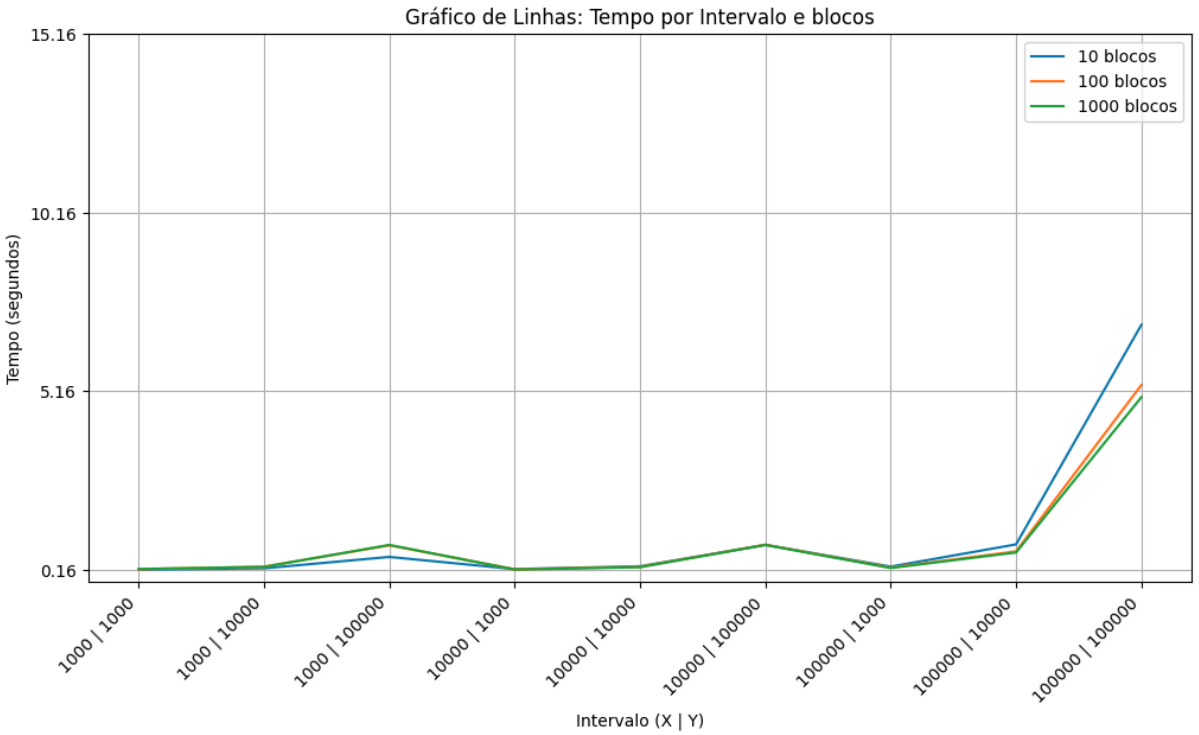
Análise das tabelas

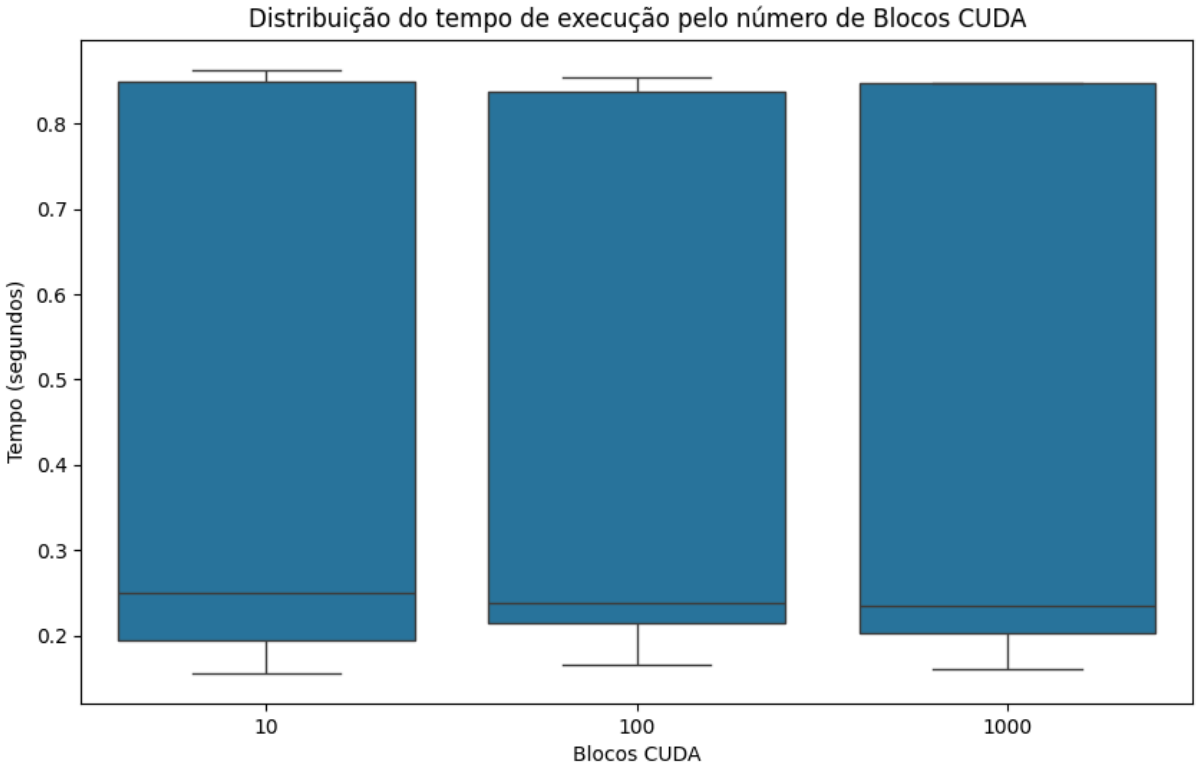
A análise das tabelas mostra que o desempenho do código CUDA varia significativamente com o aumento da quantidade de blocos e intervalos de discretização nos eixos x e y. No geral, tempos menores são obtidos para configurações com menos blocos e intervalos menores, como x=10^3 e y=10^3 . À medida que a granularidade da discretização aumenta (10^4 e 10^5), os tempos de execução crescem devido ao maior número de cálculos realizados por cada thread e ao aumento na quantidade de dados manipulados. No entanto, o uso de mais blocos (100 ou 1000) geralmente proporciona reduções no tempo em cenários de maior carga, indicando uma melhor distribuição do trabalho entre os recursos da GPU. Por exemplo, para x=10^5 e y=10^5 , o tempo diminui de 7,012s (10 blocos) para 4,985s (1000 blocos), evidenciando o benefício de maior paralelismo.

Os speedups revelam tendências semelhantes, com ganhos de desempenho notáveis para configurações mais complexas. Para discretizações 10^5 * 10^5 , o speedup cresce para 1,31 (100 blocos) e 1,40 (1000 blocos), demonstrando uma eficiência crescente com o aumento de blocos. Por outro lado, para casos menos exigentes, como x=10^3 e y=10^3 , os speedups são inferiores a 1, sugerindo que a sobrecarga associada a mais blocos pode não ser vantajosa quando a carga computacional por thread é baixa. Esses resultados mostram a importância de ajustar os parâmetros de blocos e intervalos para equilibrar eficiência e desempenho, aproveitando ao máximo os recursos da GPU.

Gráficos

Os gráficos abaixo foram plotados usando seaborn, matplotlib, pandas e numpy. As imagens do gráfico estão junto com o código fonte e os resultados do benchmark





Com base nos resultados obtidos e analisando as tabelas e gráficos, torna-se evidente que, para casos que demandam alto poder de processamento, a tecnologia CUDA supera todas as outras soluções apresentadas até agora, alcançando tempos mínimos de 4,98 segundos para o caso de $10^5 * 10^5$ intervalos. Ao comparar as configurações com 10, 100 e 1000 blocos, percebe-se que, em cenários menores, o aumento no número de blocos gera um overhead significativo, o que impacta

negativamente o desempenho. Isso também afeta a representação no gráfico boxplot de tempo por blocos, já que, na maioria dos casos, os resultados das três amostras são bastante similares, elevando o terceiro quartil e dificultando a visualização clara das diferenças.

No entanto, ao analisar as medianas, fica evidente que a configuração com 1000 blocos apresenta o menor tempo em praticamente todos os cenários, especialmente nos casos mais complexos. Essa configuração mostra-se mais eficiente para distribuir a carga de trabalho e minimizar o tempo total de execução, comprovando a eficácia do aumento de paralelismo com o uso de mais blocos em problemas de alta carga computacional.

Comparação entre CUDA e MPI

Neste tópico, irei comparar os tempos de execução das tecnologias CUDA e MPI nos cenários de melhor caso, menor caso e caso médio. Decidi não incluir o OpenMP nesta análise, pois ele foi executado em um desktop com maior poder computacional em relação ao meu notebook. Dessa forma, a exclusão do OpenMP garante que as amostras permaneçam na mesma escala e proporcionem uma comparação mais justa entre CUDA e MPI.

	Quantidade cores / blocos		Speedup	
Casos	8 MPI cores	1000 Blocos	MPI - base	CUDA
Maior $(10^5 * 10^5)$	55.789s	4.987s	1.0	11,18
Médio $(10^4 * 10^4)$	0.701s	0.225s	1.0	3,11
Menor $(10^3 * 10^3)$	0.114s	0.175s	1.0	0,65

A análise comparativa entre CUDA e MPI revela diferenças significativas no desempenho, especialmente em cenários de alta carga computacional. No pior caso $(10^5 * 10^5)$, o CUDA demonstrou uma vantagem esmagadora, completando a execução em apenas 4,987 segundos, enquanto o MPI levou 55,789 segundos, resultando em um speedup impressionante de 11,18 vezes em favor do CUDA. Este

resultado destaca a superioridade do paralelismo massivo da GPU em lidar com grandes volumes de cálculos.

Nos cenários de carga média $(10^4 * 10^4)$ e baixa $(10^3 * 10^3)$, a diferença de desempenho é menos pronunciada. Para a carga média, o CUDA obteve um speedup de 3,11, processando o caso em 0,225 segundos contra os 0,701 segundos do MPI. Entretanto, no menor caso, o MPI foi mais eficiente, com um tempo de execução de 0,114 segundos em comparação aos 0,175 segundos do CUDA, resultando em um speedup de 0,65 a favor do MPI. Esses resultados mostram que o CUDA se destaca principalmente em casos grandes, enquanto o overhead associado ao uso da GPU pode torná-lo menos eficiente em cenários menores, onde a comunicação e inicialização pesam mais no desempenho.

Conclusão

A utilização da tecnologia CUDA para resolver problemas de integração dupla mostrou-se altamente eficiente, especialmente em cenários que demandam alto poder de processamento. A análise dos resultados destacou a capacidade das GPUs de lidar com grandes volumes de dados e distribuir as tarefas de forma paralela, aproveitando ao máximo os recursos disponíveis. A escolha de parâmetros adequados, como a quantidade de blocos e threads, revelou-se fundamental para equilibrar desempenho e eficiência, demonstrando a flexibilidade e o potencial de otimização oferecidos pela plataforma CUDA.

Ao comparar CUDA com outras tecnologias, ficou evidente a superioridade do paralelismo massivo da GPU em casos complexos. Este estudo não apenas reforça a importância do uso de GPUs em problemas computacionalmente intensivos, mas também destaca a necessidade de ajustes finos nos parâmetros para obter os melhores resultados. Conclui-se, portanto, que CUDA é uma solução robusta e eficiente para aplicações que exigem alto desempenho, oferecendo benefícios significativos em termos de velocidade e escalabilidade para cálculos científicos e computacionais.