Optimização 3dfluid: CUDA

Gonçalo Brandão
Departamento Engenharia Informática
Universidade do Minho
Braga, Portugal
pg57874@alunos.uminho.pt

Henrique Pereira
Departamento Engenharia Informática
Universidade do Minho
Braga, Portugal
pg57876@alunos.uminho.pt

Maya Gomes
Departamento Engenharia Informática
Universidade do Minho
Braga, Portugal
pg57891@alunos.uminho.pt

FASE 1:

Na primeira fase do trabalho, caracterizou-se a otimização do `fluid_solver` para `size 42`. No profiling inicial, identificou-se que a função `lin_solve` era responsável por 85,53% do tempo de execução do programa, devido ao seu papel central nos cálculos. Essa função apresentou desafios em termos de localidade espacial e temporal, com várias dependências de dados, pois, na `lin_solve`, o valor atual é dependente de seu i, do i anterior $(i-1)$ e do próximo $(i+1)$.

A macro `IX(i, j, k)`, por exemplo, evidencia como a estrutura de memória afeta a localidade, sendo i mais próximo no espaço, seguido de j e k. Para melhorar a eficiência, implementaram-se alterações como a reorganização das variáveis nos laços `for`, priorizando k no ciclo externo para reduzir cache misses. Além disso, a técnica de *blocking* aumentou a localidade espacial e temporal ao dividir o problema em blocos menores, ajustados para maximizar a utilização da cache.

Na primeira fase, também abordou-se como as várias *flags* de compilação influenciam a velocidade e precisão do código, explorando-as da forma mais pertinente.

FASE 2:

Nesta fase, realizamos a otimização de um solver em OpenMP para melhorar o desempenho do nosso problema de dinâmica de fluidos. O novo solver introduzido eliminou dependências entre elementos pares e ímpares, utilizando o método *red-black solver*. Também incluiu uma condição de paragem antecipada na função `lin_solve`, quando esta converge antes de 20 iterações. O tamanho do problema foi aumentado de 42 para 84.

Inicialmente, identificou-se que a função `lin_solve` representava 42,65% do tempo de execução e, mais uma vez, foi destacada como o principal *hotspot*. Foi observado que as operações no ciclo interno eram independentes, permitindo a paralelização. No entanto, detectaram-se possíveis *data races* na variável `max_c`, que foram solucionadas com a cláusula `#pragma omp reduction(max:max_c)`, assegurando que o maior valor de `max_c` fosse mantido ao final da execução paralela. Além disso, variáveis como `old_x` e `change` foram configuradas como `firstprivate` para evitar interferências de "lixo" na memória.

Duas zonas paralelas foram implementadas para cálculos alternados dos elementos pares e ímpares. Para otimizar os *loops*, utilizou-se `#pragma omp for` com `collapse` nos dois ciclos mais externos e `nowait` para evitar sincronizações desnecessárias entre *threads*. O escalonamento estático (`schedule static`) foi escolhido, garantindo menor *overhead*, enquanto o escalonamento dinâmico mostrou-se ineficiente devido à uniformidade do tempo de execução das iterações.

Após a otimização da função `lin_solve`, novos *hotspots* como `project`, `advect`, e `set_bnd` foram identificados e otimizados. No final, foi alcançado um *speed-up* de 8,73 com 16 *threads*, abaixo do ideal teórico devido à execução sequencial de certas operações, especialmente na função `set_bnd`. A otimização demonstrou uma redução significativa no tempo de execução, com gráficos de escalabilidade indicando o impacto positivo das estratégias implementadas.

I. FASE: 3 - INTRODUÇÃO

O objetivo deste relatório é analisar a implementação sequencial e paralela, além de explorar novos ambientes de programação voltados para o aproveitamento do paralelismo. As melhorias obtidas com o uso deste ambiente serão avaliadas de acordo com análises de escalabilidade, desempenho, balanceamento de carga, entre outros aspectos. Nesta fase, o tamanho do `SIZE` foi aumentado de 84 para 168.

Assim, no inicio desta fase, o grupo enfrentou a decisão de qual metodo utilizar para melhorar o tempo de execução das fases anteriores. Entre realizar as otimizações em OPENMP (Com um teto maximo de classificação de 13 valores), CUDA ou MPI. O grupo decidiu utilizar CUDA.

O CUDA (Compute Unified Device Architecture) é uma plataforma de computação paralela e um modelo de programação desenvolvido pela NVIDIA para unidades de processamento gráfico (GPUs). Ele permite a execução de programas em GPUs para acelerá-los.

II. CUDA

Com uma nova metodologia de programação, desta vez, orientada a GPUS, o grupo começou por reformular o seu trabalho, adicionando Kernels a todas as áreas que nas fases anteriores já tinham sido classificadas como fortes candidatas a correr em paralelo.

Após esta reformulação do nosso código base, o grupo percebeu que estava a perder muito tempo com a alocação de memória na GPU e a transferência

de dados para a mesma. Assim, alteramos também as funções `allocate_data`, `clear_data`, `free_data`, `apply_events` e `sum_density`, fazendo com que a alocação e a cópia de dados para a GPU fossem realizadas de forma dinâmica.

III. Exploração

A. Identify

No início da análise da nossa obordagem CUDA, voltamos a fazer um *profiling* do mesmo. Analisando o seu *nvprof*, percebemos que a `lin_solve` representava 42,65% do tempo de execução, sendo o nosso *hotspot* e o ponto onde decidimos começar a implementação de diretivas de paralelismo no nosso código.

B. Analyse

Analisando a `lin_solve` percebemos que esta desperdissava muito tempo nos seus kernels, pois estavamos a fazer operações atômicas na memória global, reduzindo o paralelismo e diminuido o desempenho.

Resultados de Execução e Performance

Tempo total de execução: 1m28.280s

Top 5 Atividades de GPU

Nome	Time (%)	Time (s)	Calls	Avg (ms)	Max (ms)
red_phase_kernel	58.01	73.7499	7117	10.362	11.750
black_phase_kernel	40.09	50.9660	7117	7.1612	7.3895
set_bnd_kernel	0.71	0.89681	8517	0.1053	0.1197
advect_kernel	0.49	0.61986	400	1.5497	1.6352
update_velocity	0.33	0.41974	200	2.0987	2.1200

TABLE I
Top 5 atividades de GPU com suas métricas de desempenho.

Top 3 Chamadas de API

Nome	Time (%)	Time (s)	Calls	Avg (ms)	Max (ms)
cudaMemcpy	99.44	127.080	7218	17.606	27.709
cudaMalloc	0.26	0.32786	708	0.4631	231.24
cudaLaunchKernel	0.19	0.24107	32568	0.0074	0.6367

TABLE II
Top 3 chamadas de API com suas métricas de desempenho.

C. Select

Com base na análise dos dados obtidos, identificamos a necessidade de implementar uma estratégia de redução para otimizar o desempenho. Inicialmente, decidimos explorar duas abordagens: a redução interleaved e a redução sequencial. O objetivo foi avaliar qual das duas melhor se adequaria às características específicas do nosso problema.

1) Comparação de Desempenho: Redução Interleaved vs Sequencial:

Métrica	Interleaved	Sequencial	Diferença (%)
Tempo Total GPU (s)	49.06	33.76	-31.2%
black_phase_kernel (ms)	3.3214	2.2122	-33.4%
red_phase_kernel (ms)	3.2319	2.1947	-32.1%
set_bnd_kernel (ms)	105.48	105.22	-0.25%
advect_kernel (ms)	1.5519	1.5491	-0.18%
update_velocity (ms)	2.1082	2.1028	-0.26%
Real Time (s)	53.75	38.51	-28.3%

TABLE III
Comparação de desempenho entre as abordagens de redução interleaved e sequencial.

2) Análise dos Resultados: Os resultados mostram uma clara vantagem do método de redução sequencial sobre o interleaved para esta simulação de dinâmica de fluidos. A redução sequencial apresenta uma diminuição de 31,2% no tempo total de execução na GPU e uma redução de aproximadamente 33% nos tempos médios dos kernels principais (`black_phase_kernel` e `red_phase_kernel`).

Estsa melhoria pode ser atribuída a uma maior eficiência na redução sequencial para este caso específico, possivelmente devido à menor complexidade de sincronização ou ao tamanho dos dados processados, onde a sobrecarga do interleaved não se justifica. No entanto, para casos com maior volume de dados ou maior necessidade de paralelismo, o método interleaved pode apresentar vantagens.

3) Data Races: Para lidar com as *data races*, utilizamos memória compartilhada e operações atômicas. Cada bloco de threads armazena os valores máximos locais na memória compartilhada (*shared memory*), que são reduzidos hierarquicamente através de uma redução em árvore binária (*tree-based reduction*). Para garantir que apenas uma *thread* por bloco atualize a variável global `maxChange`, empregamos a função atômica `atomicMax`, que assegura exclusividade no acesso à variável durante as operações de escrita, evitando conflitos de acesso entre diferentes *threads*.

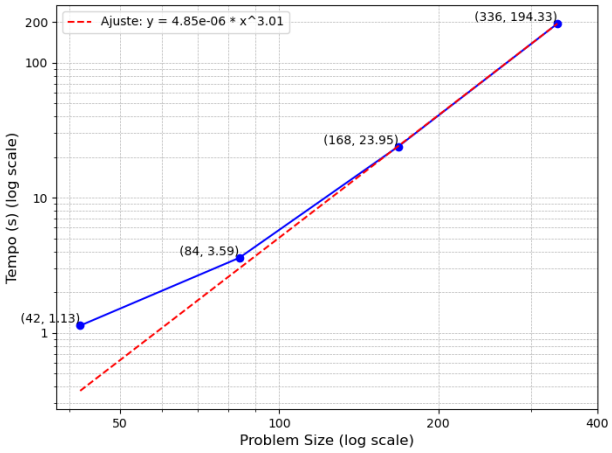
D. Performance Analise

1) Problem size: Testando a nossa solução para vários problem sizes obtemos os seguintes resultados:

Problem Size (x)	Execution Time (y) [s]
42	1.131
84	3.588
168	23.951
336	194.329

TABLE IV
EXECUÇÃO DO ALGORITMO PARA DIFERENTES TAMANHOS DE PROBLEMA.

Com estes resultados obtemos a seguinte tabela:



Analisando a relação entre os pontos, percebemos que o tempo de execução possui, aproximadamente, uma relação cúbica, pois $y = x^3 + b$. Esta relação é importante, pois sugere que a complexidade do nosso problema é de $O(n^3)$. Tal comportamento se confirma, já que esta é a complexidade dos kernels da `lin_solve`, que correspondem a $O(n^3)$, sendo o seu tempo de execução dos kernels é distribuído da seguinte forma: 47,43% no `black_phase_kernel` e 47,21% no `red_phase_kernel`.

2) Número de Threads: Seguidamente, decidimos analisar como o numero de Threads iriam influenciar o nosso tempo de execução. Decidimos apenas alterar o numero de Threads na função `lin_solve` pois este é o hotspot.

Número de Threads	128	256	512	1024
Tempo de Execução (s)	19.935	20.647	22.483	23.951

TABLE V
RELAÇÃO ENTRE O NÚMERO DE THREADS E O TEMPO DE EXECUÇÃO.