Computação Paralela

Fluid Simulation

Eduardo Cunha
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg55939@uminho.pt

Filipe Pereira
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg55941@uminho.pt

José Pacheco
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg55972@uminho.pt

***Abstract* — Este projeto foca-se na otimização de um código simulador de fluidos, abordando três níveis distintos de otimização. No primeiro nível, foram realizadas otimizações gerais, incluindo a reorganização da ordem dos loops, análise detalhada do código, redução de cache misses e aplicação de vetorização. No segundo nível, foi introduzida a implementação de OpenMP para explorar paralelismo a nível de CPU. Por fim, no terceiro nível, foi aplicada a tecnologia CUDA, utilizando GPUs para alcançar alto desempenho.**

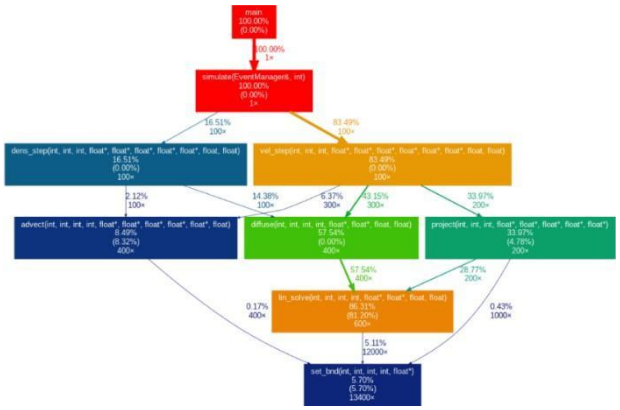
I. Introdução

Este trabalho foi desenvolvido no âmbito da Unidade Curricular de Computação Paralela, com o objetivo de explorar e implementar soluções avançadas para otimizar o desempenho de um caso de estudo previamente desenvolvido. O desenvolvimento focou-se na utilização de aceleradores, em particular GPUs, através da framework CUDA. Este método permitiu tirar partido do paralelismo massivo oferecido pelas GPUs, aumentando a eficiência na execução de tarefas computacionalmente intensivas.

Neste relatório, são apresentados e revistos os dois estágios anteriores do projeto, culminando numa síntese final que integra os resultados das várias fases de trabalho.

II. Fase 1 - Otimizações Iniciais

O *profiling* realizado com a ferramenta *gprof* revelou as funções mais dispendiosas em termos de tempo de execução no código. A função mais crítica é a ***lin_solve***, responsável por **86,31%** do tempo total de execução, dos quais 81,20% correspondem à sua própria execução, enquanto o restante resulta de chamadas a outra função. Estes resultados indicam claramente que as otimizações devem concentrar-se prioritariamente na função ***lin_solve***.



A. Otimizações (1)

Resultados obtidos antes e após as primeiras otimizações

	Antes	Depois
Instruções	166.610.376.661	242.007.753.342
Cache misses	2.305.138.098	219.096.537
Tempo Execução	27,97 s	3.9214 s

Uma das primeiras e principais otimizações implementadas foi a alteração da ordem dos ciclos nos cálculos que envolvem os elementos da matriz tridimensional. Inicialmente, a variação do índice k causava saltos significativos na memória, resultando em múltiplos cache misses, o que prejudicava o desempenho.

De seguida, implementámos uma abordagem por blocos com o objetivo de otimizar o uso da cache e reduzir ainda mais os cache misses durante o processamento. Ao dividir a matriz tridimensional em blocos mais pequenos, que podem ser completamente armazenados na cache, garantimos um acesso mais eficiente aos dados, ganhando localidade temporal e minimizando o número de acessos à memória principal.

Embora esta implementação por blocos não tenha sido a mais rápida que conseguimos, foi a que apresentou os melhores resultados na redução de cache misses. Isto pode ser explicado pela forma como a divisão em blocos melhora a localidade temporal de dados em conjunção com a localidade espacial atingida anteriormente.

B. Otimizações (2)

Resultados obtidos antes e após as otimizações finais

	Antes	Depois
Instruções	242,007,753,342	16,300,149,128
Cache misses	219,096,537	219,331,485
Tempo Execução	3.9214 s	2.6961 s

Após a remoção da implementação por blocos, a primeira otimização aplicada foi o *loop unrolling*, que consiste em expandir a iteração de um ciclo. A seguir, a partir do código com o *loop unrolling* aplicado, conseguimos vetorizar a parte inicial do código. Essa otimização foi viabilizada através da utilização de um *array* auxiliar que calcula todos os vizinhos. A vetorização foi realizada sem comprometer os valores finais, pois ocorre antes da paralelização das instruções subsequentes que utilizam os valores calculados.

C. Conclusão da 1º fase

Através de uma análise detalhada do perfil de execução, realizada com ferramentas de profiling, foi possível identificar as funções mais dispendiosas em termos de tempo e aplicar otimizações direcionadas, que se mostraram muito eficazes na melhoria do desempenho do Código.

III. Fase 2 - Open MP

Para esta fase do projeto foi introduzida uma nova função, lin_solve, baseada num algoritmo red-black. Além disso, o

tamanho dos dados foi aumentado de SIZE=42 para SIZE=84.

A. *Paralelização com OpenMP*

A análise inicial do profiler identificou os principais hot spots: a função lin_solve (79,93% do tempo total de execução), e a função advect (11,99%).

Na função lin_solve, os dois for loops foram paralelizados independentemente utilizando a diretiva #pragma omp parallel for schedule(static) private(old_x, change) reduction(max:max_c).

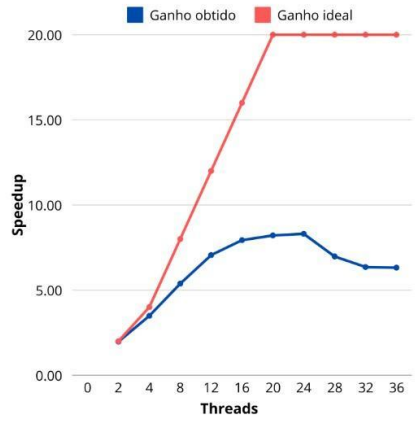
Utilizou-se reduction(max), que cria cópias da variável max_c para cada thread, agregando os valores no final para obter o maior. É mais eficiente que métodos manuais, como critical ou atomic, graças à sincronização interna.

Analisámos os escalonamentos estático e dinâmico. O escalonamento estático reduz o overhead na gestão das threads. O escalonamento dinâmico proporciona um maior equilíbrio de carga entre as threads. Como o trabalho entre threads é sempre semelhante, o escalonamento estático apresentou o melhor desempenho.

Para otimizar ainda mais o código, foram aplicados outros pragmas, ao longo do programa, para explorar ao máximo o potencial de paralelização, como pragma omp for schedule(static), tendo sempre em atenção a ausência de data races.

B. *Análise de Escalabilidade*

O melhor tempo obtido foi de 1,48s com 20 threads. A partir das 21 threads o tempo de execução não diminui mais e, eventualmente, aumenta. Isto ocorre devido a diversos fatores como, overhead de paralelização relativo às trocas de contexto entre threads e à sincronização, a limitação imposta pela Lei de Amdahl, onde as partes sequenciais do código limitam os ganhos de desempenho, e também o aumento da competição pelo acesso à memória, que provoca atrasos significativos devido à limitação na largura de banda e na latência da memória.



Relativamente a esta fase, é importante destacar que tivemos a infelicidade de não reparar num detalhe na função **set_bnd**. Ao contrário das restantes funções, nesta as variáveis do loop são definidas fora do mesmo. Como resultado, esquecemos de aplicar a diretiva ***#pragma private***, um erro subtil que resultou numa data race, precisamente o problema que pretendíamos evitar.

IV. FASE 3 - CUDA

A fase final do projeto foi conduzida com recurso a aceleradores, em particular GPUs, através da framework CUDA, aproveitando o paralelismo massivo intrínseco a estas tecnologias. O trabalho envolveu o design e a implementação de uma nova versão do caso de estudo, com o objetivo principal de reduzir o tempo de execução, complementado por uma análise detalhada do desempenho e uma série de testes para confirmar os ganhos obtidos e avaliar a escalabilidade da solução.

A. *Paralelismo em GPUs (CUDA)*

O paralelismo em GPUs (Graphics Processing Units) tira partido da sua arquitetura altamente paralela para acelerar tarefas intensivas em processamento. Diferentemente das CPUs, com poucos núcleos otimizados para desempenho sequencial, as GPUs possuem milhares de núcleos menores, capazes de executar múltiplas operações em simultâneo.

A framework CUDA (Compute Unified Device Architecture), desenvolvida pela NVIDIA, permite programar GPUs de forma eficiente. Com CUDA, os dados são divididos em blocos e threads, permitindo a cada thread resolver uma parte do problema, explorando o paralelismo massivo e reduzindo significativamente o tempo de execução.

B. *Implementação de código CUDA*

Inicialmente, as funções do ficheiro ***fluid_solver.cpp***, agora convertido em ***fluid_solver.cu,*** foram adaptadas para suportar a execução em CUDA. Esta transição exigiu a implementação de kernels específicos para cada função, sendo que, nas funções mais complexas, foi necessária a criação de múltiplos kernels para assegurar uma divisão eficiente das tarefas e uma utilização otimizada dos recursos da GPU. As adaptações foram realizadas com o máximo rigor, garantindo a total compatibilidade com CUDA e tirando partido do paralelismo massivo característico da arquitetura das GPUs.

Relativamente à criação inicial dos kernels, a implementação das funções revelou-se, em geral, bastante direta. Contudo, a função ***lin_solver*** apresentou um desafio adicional devido à complexidade envolvida na redução eficiente do valor máximo das diferenças entre iterações. Para superar esta dificuldade, optou-se pela utilização da abordagem ***Static Unroll***, uma técnica discutida nas aulas e detalhada nos materiais teóricos, que demonstrou ser altamente eficaz para operações deste tipo em ambientes CUDA.

No contexto da GPU, o Static Unroll é uma técnica de otimização que divide o processo de redução em várias fases, maximizando o desempenho. Estas fases incluem:

1. Partição Inicial em Memória Partilhada:

Cada thread transfere o valor correspondente da memória global para a memória partilhada (***sdata[]***), que é consideravelmente mais rápida. Este passo inicial minimiza a latência nas fases subsequentes da redução, melhorando o desempenho global.

2. Redução Hierárquica:

A redução é realizada de forma iterativa, onde cada thread combina os seus valores com os de outra thread, num esquema de divisão e conquista. Em cada iteração, o conjunto

de dados é dividido ao meio, reduzindo gradualmente o número de elementos até restar apenas um.

- Para tamanhos de blocos maiores, são utilizadas barreiras de sincronização (***__syncthreads()***), assegurando que todas as *threads* terminam as suas operações antes de prosseguir para a próxima etapa.
- Quando o número de threads desce para menos de 32, a redução prossegue sem barreiras, através da função ***warpReduceMax***, que utiliza primitivas ao nível do warp (como ***fmaxf***) para garantir maior rapidez e eficiência. Além disso, o algoritmo recorre ao desenrolamento do ciclo (loop unrolling), eliminando instruções condicionais e aproveitando ao máximo a execução paralela intrínseca dos warps. Esta abordagem combina otimização de desempenho com a minimização da sobrecarga computacional.

3. Escrita do Resultado:

No final do processo de redução, o valor máximo calculado por cada bloco é armazenado na memória global, permitindo a sua disponibilização para cálculos agregados subsequentes. A redução global, responsável por combinar os máximos de todos os blocos, é realizada posteriormente no CPU utilizando a função ***find_max***. Este passo garante que o valor máximo global é obtido de forma eficiente, integrando os resultados parciais produzidos pela GPU.

C. Gargalos ao nível da Memória

Contudo, apesar das transformações substanciais aplicadas ao código, os testes iniciais demonstraram que os ganhos de desempenho ficaram aquém das expectativas. Uma análise detalhada utilizando o ***nvprof*** revelou que uma parte considerável do tempo total de execução era consumida pela alocação e transferência de dados entre o host e o device, resultado da necessidade de copiar frequentemente grandes volumes de dados. Este gargalo de desempenho reduzia significativamente o impacto positivo das otimizações introduzidas nos kernels.

Para resolver este problema, foi efetuada uma reformulação profunda no ficheiro ***main.cpp***, também convertido para ***main.cu***. A principal melhoria consistiu na criação direta de variáveis no espaço de memória do device, eliminando assim as transferências contínuas entre o host e o device. Estas variáveis passam agora a ser inicializadas no início da execução e mantidas ao longo de todo o processo, sendo atualizadas de forma eficiente nas funções do ***lin_solver.cu***, que, por sua vez, foram otimizadas para evitar alocações desnecessárias e reutilizar estruturas de dados existentes sempre que possível. No final de todos os cálculos, os valores das densidades são copiados para o host, onde o cálculo final da densidade, tal como anteriormente, continua a ser realizado pelo CPU.

D. Melhoria na Grid

Adicionalmente, foram implementadas otimizações na configuração dos parâmetros de execução do CUDA. O número de threads por bloco e o número de blocos por grid foram ajustados na função **lin_solve** (de 8×8×8 para 16×16×1) com o objetivo de maximizar o desempenho, com base em testes empíricos e uma análise detalhada das especificações da GPU utilizada. A configuração 16×16 favorece uma utilização mais eficiente do warp, pois permite que as threads acessem a memória de maneira mais sequencial e coerente, otimizando o acesso à memória e melhorando a execução paralela. Essa abordagem possibilitou uma exploração mais eficiente dos recursos disponíveis, garantindo uma distribuição equilibrada das tarefas.

A aplicação destas estratégias resultou numa redução substancial do tempo de execução, traduzindo-se numa melhoria significativa da escalabilidade da solução.

E. Possíveis Melhorias

Uma possível melhoria significativa para otimizar o código seria migrar todas as operações para CUDA, aproveitando ao máximo a capacidade de paralelismo da GPU. Isso inclui a implementação do cálculo da densidade (no ***main.cu***) e da função ***find_max*** (no ***fluid_solver.cu***) diretamente na GPU, ao invés de depender do CPU para essas operações.

F. Makefile

O Makefile foi desenvolvido com base nos modelos fornecidos nas aulas práticas de CUDA. Ele está configurado para usar o compilador nvcc, a ferramenta padrão para compilar código CUDA. O Makefile oferece duas opções principais de compilação: release e debug.

- **release:** Configurado com uma série de flags de otimização, como o uso de operações matemáticas rápidas (--use_fast_math), otimização agressiva do código (-O3), e a especificação da arquitetura da GPU (-arch=sm_35). Essas configurações têm como objetivo maximizar o desempenho do código durante a execução em GPU.
- **debug:** Inclui opções voltadas para depuração, como a ativação de mensagens de erro detalhadas (-Wall) e suporte para profiling e rastreamento de desempenho (-pg), facilitando a identificação de falhas e a análise do comportamento do programa.

G. Set de testes

Para avaliar o desempenho e a escalabilidade da solução implementada em CUDA, foi desenvolvido um conjunto de testes automatizados. Este conjunto baseia-se num script em Bash (***"weak_scalability.sh"***) que executa o programa com diferentes tamanhos (***SIZE***), variando entre 32 e 288, com incrementos de 32.

Em cada iteração, o código é recompilado e executado, permitindo uma análise pormenorizada do impacto das otimizações em diversas cargas de trabalho. Durante os testes, foram registadas métricas de desempenho, como o tempo de execução e o perfil de execução dos kernels, proporcionando uma visão clara dos ganhos alcançados com as alterações efectuadas no código.

H. Resultados

Os testes realizados foram fundamentais para validar as otimizações implementadas ao longo do projeto e demonstrar a eficiência da versão paralelizada do código utilizando CUDA. A análise comparativa baseou-se em dois cenários distintos: a execução no **cluster** e numa máquina com uma **GPU RTX 3050**, permitindo avaliar diferenças de desempenho entre estas arquiteturas.

No estado inicial, sem qualquer otimização, a execução do programa para um tamanho **SIZE=168** demorava 10m32s no cluster. Este valor reflete o desempenho inicial do código sem qualquer paralelização ou utilização de GPUs.

Inicialmente, o conjunto de testes foi definido com os tamanhos **SIZE = 32, 64, 96, 128, 160, 192, 224, 256, e 288**. Posteriormente, foram incluídos os tamanhos **SIZE = 42, SIZE = 84** e **SIZE = 168**, que correspondem aos valores solicitados para a resolução do trabalho. A tabela abaixo apresenta os tempos de execução (em segundos) nas duas plataformas:

	Cluster	Máquina RTX 3050
SIZE= 32	2.661s	0.78s
SIZE= 64	3.490s	1.39s
SIZE= 96	5.206s	2.79s
SIZE= 128	10.214s	5.42s
SIZE= 160	18.597s	9.40s
SIZE= 192	30.759s	15.38s
SIZE= 224	50.169s	23.30s
SIZE= 256	1m16.397s	32.83s
SIZE= 288	1m49.559s	49.08s

	Cluster	Máquina RTX 3050
SIZE= 42	2.823s	0.91s
SIZE= 84	4.479s	1.92s
SIZE= 168	19.319s	9.94s

Como era de esperar, os resultados evidenciam uma clara vantagem na execução na Máquina RTX 3050 em comparação com o Cluster, resultado da superioridade da sua arquitetura. Em média, a Máquina RTX 3050 registou uma redução de cerca de 60% no tempo de execução face ao Cluster.

A solução, como se pode observar facilmente, apresenta uma relação linear com o parâmetro size, ou seja, à medida que o size aumenta, o tempo de execução também cresce. A solução em GPUs destacou-se como a mais escalável, alcançando os menores tempos de execução. No entanto, para tamanhos excessivamente grandes, demonstra dificuldades em manter a escalabilidade. Além disso, os nossos testes evidenciaram que máquinas mais avançadas conseguem suportar sizes ainda maiores de forma mais eficiente.

Importa realçar que, para garantir a compatibilidade e maximizar o desempenho na Máquina RTX 3050, a flag -arch=sm_35 foi ajustada para -arch=sm_86.

V. TRABALHO GLOBAL (3 FASES)

O projeto foi desenvolvido em três fases distintas, cada uma com foco em diferentes estratégias para otimizar o desempenho do código e reduzir o tempo de execução.

Na primeira fase, a abordagem principal consistiu na reorganização dos ciclos que manipulam a matriz tridimensional, com o objetivo de reduzir as falhas de cache. A solução final envolveu o desenrolamento de loops (loop unrolling) e o uso de um array auxiliar, permitindo alguma vetorização. Esta fase foi particularmente interessante, pois exigiu criatividade na implementação das soluções de otimização, sendo um dos momentos mais desafiadores e cativadores do projeto.

Na segunda fase, introduzimos a paralelização utilizando OpenMP, o que permitiu paralelizar todo o código. Esta fase foi relativamente simples, mas enfrentamos um desafio técnico quando uma variável foi esquecida, resultando em um data race. Além disso, houve um erro de indentação no nosso Makefile, o que impossibilitou a avaliação correta dos tempos de execução durante os testes.

Na terceira e última fase, implementamos uma versão otimizada do código utilizando GPUs e a framework CUDA, explorando o paralelismo massivo dessas arquiteturas. A adaptação inicial das funções revelou que as transferências de memória entre o host e o device representavam um obstáculo significativo. Para resolver esse problema, foi implementada uma memória global para as variáveis, o que resultou na maior redução do tempo de execução entre todas as fases do projeto.

Embora não seja possível comparar diretamente os resultados das três fases, pois houve uma mudança de paradigma entre a primeira e a segunda fase devido à alteração do algoritmo do solver para um modelo mais facilmente paralelizável, é evidente que as tecnologias empregadas em cada fase contribuíram para melhorias contínuas no desempenho. OpenMP revelou-se uma excelente solução de paralelização, mais poderosa que ajustes no código e potencialmente mais eficiente do que a vetorização. O uso de GPUs com CUDA, por sua vez, mostrou ser muito mais eficiente do que outras tecnologias, embora exija um investimento considerável em tempo e esforço para a sua implementação.

Consideramos que todos os objetivos propostos foram cumpridos com sucesso, e o trabalho foi de grande importância para o nosso aprofundamento no tema, além de ser bastante interessante de desenvolver.