Computação Paralela
3dfluid - 3º Fase

Rúben Silva
pg57900
Mestrado Eng. Informática

Pedro Oliveira
pg55093
Mestrado Eng. Informática

Henrique Faria
a91637
Externo

I. Introdução

Através de ferramentas de análise de *profiling* é possível fazermos uma análise da performance do código. A partir dos resultados obtidos, procedemos, na fase 1, a várias otimizações a nível de hierarquia de memória (cache).

De forma a explorarmos um maior paralelismo, recorremos ao OpenMP de forma a otimizarmos o tempo de execução através do lançamento de várias *threads* (paralelismo de grão fino) e, ao mesmo tempo, estudarmos a escalabilidade da nossa solução, tendo em conta o balanceamento da carga, entre outras métricas.

Para esta fase final, iremos estudar e implementar uma solução em memória distribuída. Para esta fase final, iremos estudar e implementar uma solução em GPUs utilizando CUDA, explorando paralelismo massivo a partir da execução de milhares de threads simultaneamente, com especial atenção à eficiência da alocação de recursos, otimização e redução de latências associadas ao acesso à memória global.

II. Fase 1 - Sequencial

A. *Profiling*

O profiling foi utilizado como ferramenta essencial para analisar detalhadamente a performance do código sequencial. Este processo permitiu identificar as principais funções responsáveis pelo consumo de recursos computacionais e fornecer uma base para otimizações subsequentes. A análise foi conduzida utilizando o gprof2dot, que gerou snapshots dos Call-Graphs inicial e final, destacando os principais gargalos do sistema. A seguir, apresentamos os resultados e as melhorias implementadas com base nessa análise.

(Por falta de legibilidade dos *Grafos*, as funções com mais custosas estão descritas nas Tabelas abaixo dos respetivos diagramas)

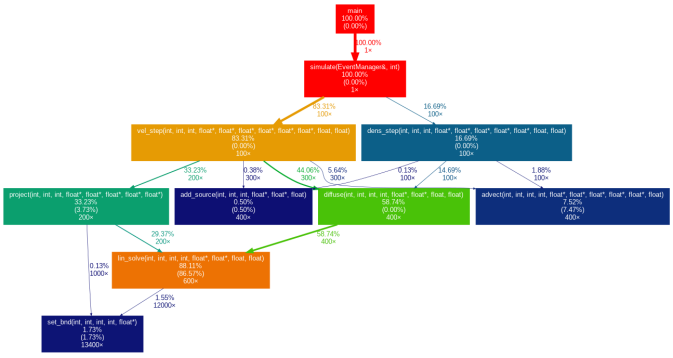

Fig. 1: Call-Graph Inicial

TABLE I: Call-Graph Inicial

Nome da Função	Utilização Relativa	Nº Chamadas
lin_solve	86.57%	600
advect	7.47%	400
project	3.73%	200
set_bnd	1.73%	13400
add_source	0.50%	400

Após a implementação de todas as otimizações, foi obtido o seguinte *Call-Graph*:

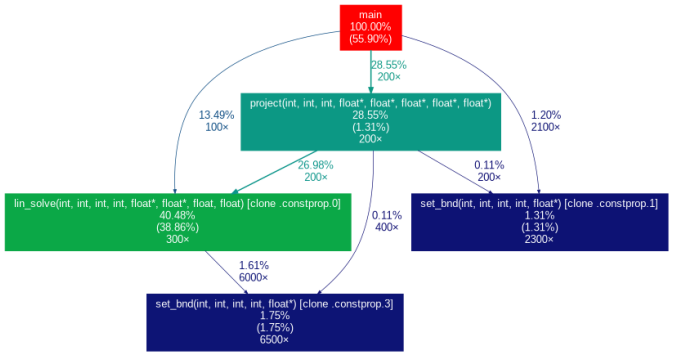

Fig. 2: Call-Graph Final

TABLE II: Call-Graph Final

Função	Utilização Relativa	Nº Chamadas
lin_solve	47.81%	300
main	44.74%	——
project	3.95%	200
set_bnd	3.07%	8800

B. Estimativa de Performance

Realizamos uma estimativa da complexidade de cada uma das funções, que podem ser vistas na seguinte tabela de forma a obtermos uma estimativa de performance do programa:

TABLE III: Estimativa de Performance

Função	Complexidade Temporal
lin_solve	O(LINEARSOLVERTIMES×M×N×O)
advect	O(M×N×O)
project	O(M×N×O)
set_bnd	O(M×N+N×O+M×O)
add_source	O(M×N×O)

A análise apresentada na Tabela III fornece uma estimativa da complexidade temporal das principais funções do simulador de fluidos baseado no repositório 3dfluid. Estas funções desempenham papéis essenciais na simulação de fluidos. Estas estimativas permitem uma avaliação do impacto do tamanho do problema no desempenho do simulador. Por exemplo, aumentar o **SIZE** resulta no crescimento cúbico no custo computacional para funções como advect e project, enquanto o impacto de linsolve é exacerbado pelo número de iterações do solver.

Adicionalmente, a análise sugere potenciais áreas para otimização. A paralelização de funções como advect e project, pode melhorar significativamente o seu desempenho. Para linsolve, reduzir LINEARSOLVERTIMES ou empregar métodos iterativos mais eficientes pode trazer benefícios substanciais sem comprometer a precisão.

Dessa forma, a análise de performance não apenas identifica os gargalos computacionais do simulador, mas também aponta caminhos para aumentar o desempenho do simulador.

C. Optimizações:

1) Hierarquia de Memória: A hierarquia de memória tem impacto direto na performance do código, devido às diferenças de velocidade entre Cache e RAM. A técnica de Loop Tiling foi usada para segmentar ciclos em blocos menores, maximizando o uso da Cache e reduzindo acessos lentos à RAM, o que melhorou a eficiência do código.

Após testes, considerando o tamanho da Cache L1 (32KB) do processador do cluster, os melhores tamanhos de blocos (*Tile Size*) foram determinados para diferentes funções:

- *lin_solve* → **TILE = 6**
- *advect* → **TILE = 26**
- $project_{for1}$ → ***TILE = 16***
- $project_{for2}$ → ***TILE = 16***

Os valores são pequenos devido à natureza cúbica (**TILE**^{3}) do simulador, o que resultou no aumento exponencial no consumo da memória.

Além disso, a ordem dos loops foi ajustada para otimizar o reaproveitamento de dados na Cache. Ao mudar de "*M-N-O*" para "*N-O-M*", foi possível obter um ganho de cerca de 10% no tempo de execução. As siglas representam as dimensões da matriz M×N×OM×N×O, e a escolha da ordem pelas quais as percorremos revelou-se crucial para o desempenho.

2) Paralelismo no Nível de Instrução: As melhorias implementadas incluem o desenrolar automático de loops, reduzindo o tempo gasto em cada iteração, e a utilização de otimizações específicas para acelerar cálculos e operações.

Para além disso, foram exploradas instruções avançadas do processador e técnicas de otimização durante a ligação do código, o que resultou em um desempenho significativamente superior.

Flags utilizadas:

-Ofast -funroll-all-loops -march=native -flto

D. Resultado

Entre as otimizações aplicadas, destacam-se o uso da técnica de Loop Tiling, com tamanhos ajustados para cada função baseado na Cache L1, e a reordenação dos loops de "M-N-O" para "N-O-M", que proporcionou um ganho adicional de 10% no tempo de execução. Técnicas adicionais, como unroll automático de loops e otimizações específicas do compilador com flags avançadas (-Ofast, -funroll-all-loops, -march=native), também contribuíram para os resultados obtidos.

A análise detalhada do desempenho destacou a importância de priorizar a hierarquia de memória e identificou gargalos computacionais críticos. Essas melhorias não apenas reduziram o tempo de execução, mas também estabeleceram uma base sólida para as próximas fases, que explorarão paralelismo em memória compartilhada e distribuída.

TABLE IV: Resultados Sem Hierarquia de Memória

Métricas	No Flags	`-O2 -FRLoop -FtrVect -mavx`	`-O3 -FRLoop -FtrVect -mavx`	`-Ofast -FRLoop -MNative -flto`
Instructions	166613827657	16291008681	15645802313	16671358346
Cycles	94417810364	43400774402	34525319160	22778985010
L1dcLM	2420130899	2846690855	2594925840	2530349767
Cache-Misses	1026	1273	6025	1276
Cache-Reference	1336375896	2420720487	1287953269	1205093828
CPI	0,567	2,664	2,207	1,366
Run-Time (ms)	31.2821	13.5709	10.7519	7.072
Result	81981.3	81981.3	81981.3	81981.5

TABLE V: Resultados Com Hierarquia de Memória

Métricas	No Flags	`-O2 -FRLoop -FtrVect -mavx`	`-O3 -FRLoop -FtrVect -mavx`	`-Ofast -FRLoop -MNative -flto`
Instructions	168117489688	19328951330	17286351975	18894376660
Cycles	63205633692	21383833040	20042667776	12453730302
L1dcLM	243535495	244202794	243172253	245089457
Cache-Misses	737	859	3041	724
Cache-Reference	97980725	100185792	98714473	98948982
CPI	0.376	1,106	1.159	0,659
Run-Time (ms)	19.6830	6.5753	6.09	3.62
Result	81981.2	81981.2	81981.2	81981.5

Com as otimizações implementadas, o tempo de execução da simulação foi significativamente reduzido de 31,3s para 3,62s, resultando em uma melhoria de $\approx$ 89%. Além disso, a taxa de L1-dcache-misses foi minimizada para cerca de 3,4%, utilizando uma matriz tridimensional de 48³ partículas (N = 48).

III. Fase 2 - OpenMP

Nesta segunda fase, usámos o OpenMP para explorar o paralelismo da memória de forma a reduzir o tempo de execução do simulador com 8x mais partículas. Também foi substituído o solver pelo (Red-Black), fornecido pela equipa docente.

A. Análise

Foi realizada uma análise do código utilizando o *perf* de forma a que se pudesse identificar quais seriam os Hot-Spots do programa.

Overhead	Samples	Command	Shared Object	Symbol
79.56%	108824	fluid_sim	fluid_sim	[.] lin_solve(int, int,
14.72%	20078	fluid_sim	fluid_sim	[.] advect(int, int, in
2.86%	3904	fluid_sim	fluid_sim	[.] project(int, int, i
1.46%	2004	fluid_sim	fluid_sim	[.] main
0.80%	1096	fluid_sim	fluid_sim	[.] set_bnd(int, int, i
0.02%	28	fluid_sim	[unknown]	[k] 0xffffffff8ac011d3

Fig. 3: Profiling

Como se verifica na *Fig. 1*, a função *lin_solve* continua a ser o principal **Hot-Spot**, mas as outras funções têm de ser otimizadas.

B. Implementação

Inicialmente, melhoramos a função *lin_solve*. Posteriormente, de modo a conseguirmos explorar o paralelismo na função, procedemos à melhoria da Localidade *Espacial* e *Temporal* do novo solver, ajustando a ordem dos ciclos dos *"fors"* para para (*O-N-M*).

Primordialmente começamos por explorar, de uma forma básica, o Paralelismo. Contudo, para que fosse possível melhorar a escalabilidade, procedemos às seguintes alterações:

Os ciclos Red e Black foram inseridos numa secção **parallel** para minimizar o overhead do **Paralelismo**. Adicionou-se também a cláusula (**reduction(max: max_c)**), que permite calcular o valor máximo de forma segura entre múltiplas threads, assegurando que a comparação do valor máximo (max_c) ocorre sem erros de concorrência, minimizando assim o overhead das **Sincronizações**.

Para mitigar o **Load Imbalance**, foi aplicada a cláusula **schedule(static)** nos ciclos do algoritmo *Red-Black*. Esta configuração distribui as iterações dos ciclos entre as threads, ajustando a carga de trabalho em tempo real.

As restantes funções, Localidade Espacial e Temporal, já se encontravam muito bem otimizadas, tendo em conta os valores obtidos na primeira fase. Por este motivo, a exploração do paralelismo foi, somente, feita com a utilização do **parallel for schedule(dynamic)** para todos os ciclos "*fors*".

Toda a implementação na versão final é fruto de diversas tentativas diferentes de atingir a melhor performance.

C. Resultados

Os seguintes resultados foram obtidos no Cluster da Universidade do Minho, na partição de *cpar* que contém 20 PUs.

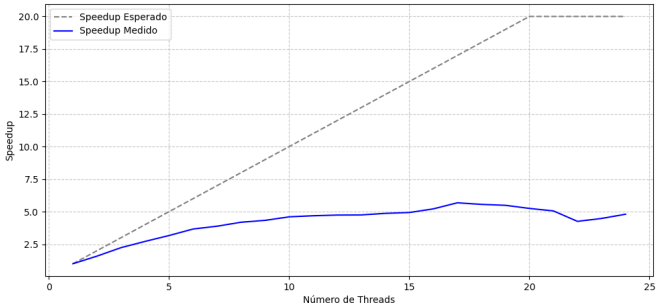

Fig. 4: Evolução do Speedup

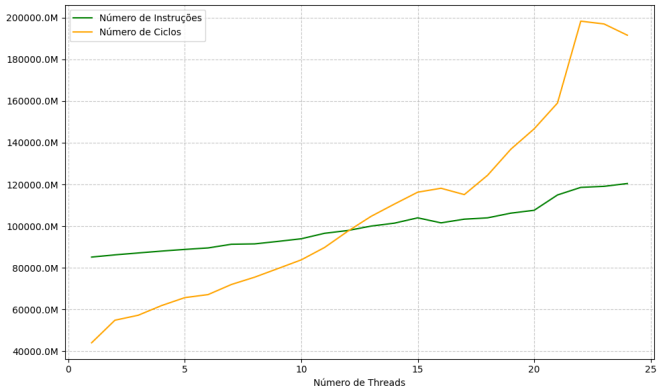

Fig. 5: Evolução do nº de Instruções e Ciclos

Foi obtida uma melhoria de ≈ 6x de Speedup. Também é notório o aumento acentuado do número de ciclos e um aumento mais ligeiro do número de instruções.

A Versão Sequencial teve um runtime de ≈ 12.8s e a Versão Paralela, com o melhor runtime de ≈ 2.35s.

É ainda notável que acima das 20 Threads os valores de ambos os gráficos ultrapassam o que seria expectável. Isto, resulta, claramente, da existência de só 20 PUs.

IV. Fase 3 - CUDA

A. Introdução

O CUDA, "Compute Unified Device Architecture", é uma arquitetura de computação paralela desenvolvida pela NVIDIA. Esta tecnologia permite acelerar o processamento utilizando GPUs (Unidades de Processamento Gráfico). De forma simples, o CUDA possibilita aos programadores tirar partido do poder de processamento das GPUs para executar cálculos em paralelo, sendo particularmente útil em tarefas relacionadas com gráficos, simulações físicas e outras computações científicas.

Como observado na fase anterior (Fase 2), através da análise do código sequencial, a função que consome mais tempo computacional é a *lin_solve* (representando ≈ 80% do tempo de execução). Por isso, é nesta função que iremos aplicar técnicas de paralelismo (tal como fizemos com o OpenMP) utilizando o modelo de programação CUDA.

Devido à natureza intensiva do simulador, onde cada função é chamada frequentemente, tornou-se essencial converter grande parte do código para CUDA. Esta abordagem tem como objetivo minimizar as trocas de dados entre o dispositivo (GPU) e o host (CPU), uma vez que estas foram identificadas como o principal bottleneck ao longo do desenvolvimento desta fase.

B. Implementação

Inicialmente, definimos **N=168** e alocámos os vetores necessários diretamente na GPU utilizando o *cudaMalloc*. Posteriormente, transferimos os dados para a GPU através do *cudaMemcpy*. Este fluxo de operações assegura que os cálculos principais ocorrem no dispositivo, reduzindo significativamente a necessidade de comunicação com o host durante a execução.

Este passo estratégico revelou-se crucial para mitigar o impacto negativo da latência e da largura de banda limitada associadas às transferências de dados entre o host e a GPU. Desta forma, conseguimos não só melhorar a eficiência computacional, como também otimizar o desempenho global do simulador.

Todas as funções no ficheiro *fluid_solver.cu* foram alteradas, mantendo exatamente a mesma estrutura do código original. As modificações restringiram-se apenas ao essencial para que o código

funcionasse em CUDA, garantindo uma transição direta e alinhada com o design inicial.

Alterações Realizadas: Adição de Kernels CUDA, em cada função, foram adicionados os kernels necessários para realizar os cálculos paralelos. Estes kernels mantêm a lógica original da função, permitindo uma execução eficiente na GPU, tendo em conta que a *lin_solve* era a mais intensiva computacionalmente e foi necessário um olhar mais criterioso que será explicado a seguir.

Com estas adaptações, o código está agora preparado para explorar o paralelismo da GPU sem comprometer a clareza e a organização do código original.

No final da execução, na função *simulate*, realizámos um último *cudaMemcpy* para transferir os resultados calculados na GPU de volta para o host, garantindo a correta visualização dos mesmos.

Na implementação CUDA, utilizámos uma grid tridimensional com blocks de "8x8x8" threads, totalizando 512 threads por bloco. Essa configuração foi escolhida para maximizar o paralelismo e a eficiência, equilibrando a utilização de recursos da GPU.

Cada thread foi responsável por processar uma parte do domínio tridimensional, sendo indexado por threadIdx e blockIdx. A grid foi dimensionada dinamicamente para se adaptar às dimensões individuais de cada função, garantindo uma escalabilidade superior, por exemplo, na função *lin_solve* usamos "BLOCK-1" devido a se mostrar mais eficiente.

No *lin_solve*, foi implementada uma estratégia otimizada baseada no conceito de reduction para calcular apenas as partículas "red" ou "black". Essa abordagem foi projetada para evitar **idle threads** dentro de warps, um problema que ocorre quando alguns threads dentro de um warp não realizam cálculos devido a condições como "if phase % 2 == 0". Quando isso acontece, os threads inativos aguardam enquanto os outros são executados, causando ineficiência devido ao divergence.

Ao distribuir dinamicamente os cálculos entre os threads, garantindo que todos os threads de um warp contribuam para a computação, maximizamos a utilização do paralelismo intrínseco da GPU. Isso eliminou avaliações redundantes e minimizou o impacto de warp divergence, resultando em uma execução mais eficiente e em melhor desempenho geral.

Também foi incluída uma verificação para garantir que apenas os threads localizados em regiões relevantes do domínio realizem cálculos. Isso é importante para evitar o processamento por *idle threads*, que podem surgir devido à maneira como os warps são organizados em GPUs.

Essa abordagem assegura que apenas os threads internos ao bloco e dentro das fronteiras do domínio contribuam para os cálculos. Com isso, minimizamos o desperdício de recursos computacionais, reduzindo o impacto de warp divergence (quando threads dentro de um warp seguem diferentes caminhos de execução). Dessa forma, a execução é otimizada, garantindo que o paralelismo da GPU seja explorado de maneira mais eficiente.

Estas melhorias foram aplicadas exclusivamente ao *lin_solve*, pois, durante os testes, não apresentaram benefícios significativos de desempenho em outras funções do programa.

C. Resultados

Inicialmente, apresentamos a evolução do tempo de execução em função do valor de BLOCK. Como o contexto é tridimensional, o número total de threads por bloco é dado por BLOCK × BLOCK × BLOCK, o que faz com que o valor escale muito rapidamente. Valores abaixo de BLOCK = 8 não produzem resultados suficientemente corretos, enquanto valores iguais ou superiores a BLOCK = 11 geram erros ao transferir dados para a GPU, uma vez que a arquitetura Kepler suporta no máximo 1.024 threads por bloco. Assim, todas as análises consideram configurações entre BLOCK = 8 e BLOCK = 10, garantindo resultados corretos e dentro das limitações da GPU.

Com base na análise realizada, concluímos que o tamanho ideal para garantir a precisão necessária na simulação de partículas fluídicas é **BLOCK = 8**. Este valor proporciona um equilíbrio perfeito entre desempenho, precisão e conformidade com as limitações da arquitetura Kepler.

TABLE VI: *BLOCK* - Runtime

BLOCK	Threads por bloco	Resultado	Runtime (s)
7	343	Incorreto	N/A
8	512	Correto	10.1
9	729	Correto	11
10	1.000	Correto	13.2
11	1.331	Erro	N/A

Com base nos dados apresentados na tabela *II*, foi realizado um profiling do uso da GPU utilizando a ferramenta **nvprof** e resultando na seguinte figura:

Time(%)	Time	Calls	Avg	Min	Max	Name
75.70%	7.49883s	14254	526.09us	520.36us	553.19us	lin_solve_kernel(int
8.62%	854.15ms	400	2.1354ms	2.1139ms	2.1879ms	advectKernel(int, int
5.67%	562.00ms	200	2.8100ms	2.7956ms	2.8280ms	project_kernel_2(int
3.26%	323.18ms	400	807.94us	786.25us	833.58us	addSourceKernel(int,
3.06%	302.92ms	200	1.5146ms	1.5044ms	1.5210ms	project_kernel_1(int
2.52%	249.19ms	8527	29.223us	14.785us	108.35us	set_bnd_kernel(int,
0.60%	59.252ms	7135	8.3040us	863ns	6.7061ms	[CUDA memcpy HtoD]
0.57%	56.846ms	7135	7.9670us	1.2480us	6.0435ms	[CUDA memcpy DtoH]
0.00%	73.569us	20	3.6780us	3.3280us	4.8000us	apply_events_kernel(
94.77%	9.94035s	14270	696.59us	5.8540us	11.654ms	cudaMemcpy
2.80%	293.86ms	608	483.33us	110.56us	216.72ms	cudaMalloc
1.74%	182.56ms	24001	7.6060us	5.5690us	627.62us	cudaLaunchKernel
0.68%	71.188ms	608	117.09us	83.200us	2.0474ms	cudaFree
0.01%	722.06us	1	722.06us	722.06us	722.06us	cuDeviceTotalMem
0.00%	385.77us	101	3.8190us	360ns	152.87us	cuDeviceGetAttribute
0.00%	32.913us	1	32.913us	32.913us	32.913us	cuDeviceGetName
0.00%	20.003us	1	20.003us	20.003us	20.003us	cuDeviceGetPCIBusId
0.00%	2.9130us	3	971ns	426ns	1.6750us	cuDeviceGetCount
0.00%	1.7560us	2	878ns	436ns	1.3200us	cuDeviceGet
0.00%	1.0910us	1	1.0910us	1.0910us	1.0910us	cuDeviceGetUuid

Fig. 6: nvprof - BLOCK=8

Tendo em conta os dados da tabela acima, verifica-se que a função *lin_solve* consome ≈ 76% do tempo total, enquanto a call à API *memcpy* representa ≈ 95% desse tempo. Este elevado consumo de tempo pelo *memcpy* pode ser explicado pela transferência de grandes volumes de dados para a VRAM no início da execução. Após essa transferência inicial, a GPU pode realizar movimentações internas de dados entre a VRAM e a cache, o que contribui significativamente para o tempo total de execução, mesmo não ocorrendo novas chamadas explícitas de *memcpy* durante o processamento.

Para testar a nossa implementação, iremos variar o valor de **N** e comparar os resultados com a versão sequencial e a paralelizada utilizando OpenMP e CUDA.

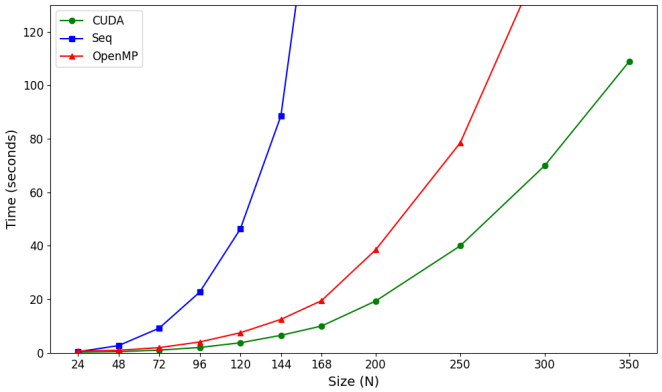

Fig. 7: CUDA vs Seq vs OMP

Os resultados apresentados na Figura 7 evidenciam o impacto da escalabilidade das diferentes abordagens computacionais (CUDA, OpenMP e sequencial) em função do tamanho do problema (N). Observa-se que a implementação sequencial apresenta crescimento exponencial no tempo de execução, ultrapassando rapidamente o limite prático de dois minutos $N > 168$, devido à sua incapacidade de paralelizar as operações. Por outro lado, a implementação com OpenMP, apesar de explorar paralelismo em nível de CPU, enfrenta limitações no número de threads disponíveis e na sobrecarga de sincronização, resultando em tempos significativamente maiores para $N > 200$.

Em contraste, a abordagem baseada em CUDA apresenta uma escalabilidade superior, com tempos de execução substancialmente menores, mesmo para $N = 300$. Esta vantagem deve-se à arquitetura de memória hierárquica e ao massivo paralelismo das GPUs, que permitem executar milhares de threads simultaneamente. Assim, o CUDA demonstra ser uma solução mais eficiente para problemas de larga escala, minimizando o tempo de execução mesmo em cenários com grandes volumes de dados, enquanto as abordagens sequenciais e de CPU enfrentam gargalos intrínsecos de escalabilidade.

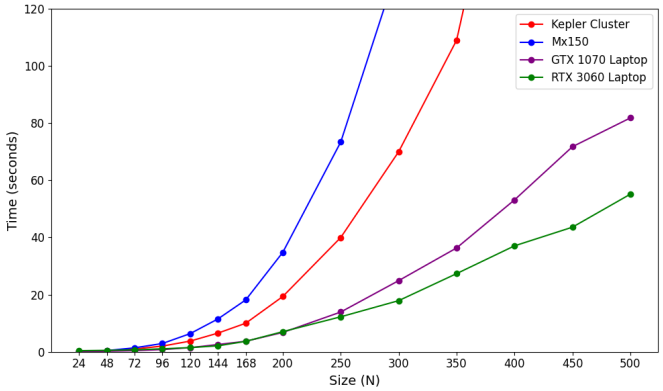

Fig. 8: GPU Benchmarks

Ao comparar as diferentes placas gráficas, podemos observar um padrão claro de que, à medida que o tamanho da matriz tridimensional aumenta (ou seja, o número de partículas e interações entre elas aumentam), o tempo de execução também aumenta. Esse comportamento é esperado, uma vez que simulações de partículas em 3D demandam mais cálculos à medida que o número de partículas cresce.

No entanto, o impacto do hardware é evidente. A GTX 1070 Laptop e a RTX 3060 Laptop são mais eficientes, exibindo tempos de execução significativamente menores, especialmente em matrizes grandes. Já o Kepler Cluster e a MX150 começam a apresentar dificuldades de desempenho conforme o tamanho da matriz aumenta, com tempos de execução que se passam dos 120 segundos, especialmente para matrizes de tamanho N $>$ 350.

D. Melhorias Possíveis

Embora não tenhamos alcançado o desempenho esperado utilizando GPUs, acreditamos que existem melhorias que poderiam ser implementadas, como por exemplo, o uso de memória compartilhada, uma região de memória de acesso rápido e baixa latência, que é compartilhada entre os threads de um bloco no kernel CUDA. Não conseguimos de maneira alguma implementar e que justificasse a sua implementação

V. Conclusão

Durante os três trabalhos realizados neste semestre, explorámos diferentes técnicas de paralelismo para otimização de código e pudemos verificar, na prática, a melhoria significativa em relação ao código original fornecido pelos professores. No primeiro trabalho, aplicámos várias técnicas, como alterações no algoritmo, melhorias no ILP, otimização da hierarquia de memória e vetorização, o que resultou em uma melhoria considerável no tempo de execução. No segundo trabalho, ao adicionar paralelismo com OpenMP ao código já otimizado, conseguimos alcançar tempos de execução ainda mais rápidos.

No terceiro trabalho, aprofundámos os conhecimentos no modelo de programação CUDA. Embora não tenhamos superado a versão com OpenMP, conseguimos superar a versão sequencial, o que demonstrou o potencial de aceleração oferecido pela CUDA. A complexidade dessa tarefa motivou-nos a estudar mais profundamente este modelo, com o objetivo de alcançar a melhor solução possível.

Além disso, é importante destacar que, ao longo de todos os trabalhos, foi possível identificar não só os benefícios do paralelismo, mas também os desafios inerentes à implementação de tais soluções em diferentes plataformas. O impacto das trocas de dados entre CPU e GPU, a escolha de técnicas de paralelismo adequadas e o gerenciamento eficiente de recursos de memória foram fatores cruciais para o desempenho final. Assim, este processo proporcionou uma compreensão mais aprofundada das limitações e das melhores práticas de otimização, além de servir como base para futuras investigações na área de computação paralela.