# Trabalho Prático de Computação Paralela

Parte 3

1st Carlos Silva
*Mestrado em Engenharia Informática*
*Universidade do Minho*
Braga, Portugal
PG57518

2nd Tiago Silva
*Mestrado em Engenharia Informática*
*Universidade do Minho*
Braga, Portugal
PG57614

***Abstract*—Nesta fase do trabalho, vamos implementar CUDA no código que fomos otimizando ao longo do projeto, com o principal objetivo de melhorar a performance do programa.**

## I. Introdução

O objetivo principal deste projeto é investigar e avaliar uma gama de estratégias e ferramentas visando a otimização de um algoritmo. Este projeto permitiu-nos testar vários modelos de paralelização, com o objetivo de melhorar a performance do nosso algoritmo.
Neste documento vamos falar de todas as fases do projeto, desde a primeira otimização do algoritmo sequencial, até à implementação de CUDA. Tal como otimizações feitas nas versões anteriores

## II. WA1 - Otimização da versão sequencial

A primeira fase deste trabalho prático consistiu em aplicar técnicas de otimização de maneira a reduzir o tempo de execução do código. Essas técnicas incluem: **mudança de algoritmo**, melhorias em **ILP** e **vetorização**.
Quando analisamos inicialmente o código, percebemos que a função *lin_solve* tem uma complexidade de **O(N³)** e **85,79%** do tempo de execução do código. Obtendo estes resultados.

TABLE I
Tabela de Performance Máquina Local

| Version | Time | CPI |
|---|---|---|
| base | 22.62s | 0,48 |

| #I | Cache Loads | Cache Misses |
|---|---|---|
| 166,903,979,997 | 70,087,724,072 | 2,344,329,597 |

### A. Otimizações implementadas

*1) Mudança de algoritmo:* : Implementamos a técnica de tilling que é frequentemente utilizada em multiplicação de matrizes, onde a matriz é dividida em sub-matrizes, denominadas por tiles. Em que em vez de se realizar a multiplicação elementar de uma matriz inteira de cada vez, o processo é feito em blocos, onde cada tile é carregada para cache e processada individualmente.
Tiling permite um melhor uso da cache, visto que matrizes grandes não cabem na cache do processador, o que provoca acessos frequentes à memória primária que torna o programa mais lento. Com tiling, blocos mais pequenos conseguem ser mantidos em cache, diminuindo assim os acessos à memória e ao mesmo tempo reduzindo o número de cache misses. Uma otimização realizada agora foi reduzir o tamanho das tiles de 16 para 4, para que estas caibam em cache.
Outra mudança que fizemos no algoritmo foi substituir a operação de divisão pela multiplicação do valor inverso, calculado inicalmente na função, visto que a multiplicação e uma operação mais leve.

*2) Melhorias em ILP:* : A melhoria implementada foi o uso da flag **-funrool-loops**, que permite desenrolar os loops automaticamente, reduzindo o overhead do controlo dos loops e aumentando o número de operações disponíveis para execução paralela.

*3) Hierarquia da memória:* : Mudamos a forma como IX é calculado de forma identica a trocar a ordem dos loops de maneira a melhorar a localidade espacial dos dados e reduzir as cache misses. No código original, o acesso à matriz não era contíguo na memória, o que resultava em um alto número de cache misses. Ao alteral a ordem dos loops, garatindo que os dados fosses acessados sequencialmente em linha (em vez de por coluna), o acesso à memória se tornou mais eficiente. Isso permite que os dados sejam carregados nas caches de forma mais eficaz, diminuindo o número de cache misses e melhorando o desempenho geral.

*4) Vetorização:* : Para fazer uso da vetorização usamos as flags **-ftee-vectorize** e **march=native**. A flag **-ftree-vectorize** foi usada para ativar a tree vectorizer, que permite automaticamente vetorizar loops transformando operações escalares em operações vetoriais. Essas operações vetoriais utilizam instruções SIMD, que podem melhorar a performance do código. Adicionalmente, usamos a flag **-march=native** que diz ao compilador para gerar o código otimizado de acordo com a máquina onde se está a correr, usando o conjunto de instruções que melhor faz uso de.

### B. Resultados

Podemos ver que conseguimos um speedup de 9,16 vezes em relação ao código original.

TABLE II
TABELA DE PERFORMANCE MÁQUINA LOCAL

| Version | Time | CPI |
|---|---|---|
| final | 2,47s | 0,45 |

| #I | Cache Loads | Cache Misses |
|---|---|---|
| 20,056,180,168 | 8,426,405,558 | 220,462,068 |

## III. WA2 - OPENMP

A segunda fase do projeto tem como principal objetivo melhorar a performance do nosso programa, utilizando técnicas de paralelização e consequentemente, reduzindo o seu tempo de execução.
Começamos por identificar onde são os hotspots do programa. Em seguida, analisamos e avaliamos quais seriam as funções que beneficiariam ou não de paralelismo e com base nisso implementamos as nossas otimizações.

### A. Otimizações implementadas

Com o uso do gprof, foi identificado o hotspot do programa, a **lin_solve**. Foram então aplicadas técnicas de paralelização para diminuir o tempo de execução da mesma. Através do OpenMP, foi usado **#pragma omp parallel for reduction(max:max_c) schedule(static) collpase(2)**, em cada um dos dois nested loops presentes no red e no black, tendo ambos a mesma lógica. O **"parallel for"** paraleliza a execução dos loops, aproveitando múltiplas threads para processar diferentes partes da malha tridimensional. Isto é importante para processar de maneira mais eficiente a grande carga de volume. O **reduction** usado serve para caluclar o valor, neste caso de **max_c**, de forma segura e paralela pois cada thread mantêm o seu valor de max_c e no final as cópias locais são usadas para calcular o valor global máximo. Isto garante que nunca aconteçam *data races* para atualizar este valor. O uso de **scheduling**, neste caso **static**, funciona muito melhor que o **scheduling dynamic** pois o workload é muito parecido ou igual e a distribuição deste trabalho é feito de maneira mais uniforme pois este trabalho é de tamanho previsível. A cláusula **collapse** combina os 2 primeiros loops de índices i e j em um único loop. Isso melhora a distribuição de trabalho entre as threads. Apenas pode ser usado com **collpase(2)** pois o terceiro ciclo (k) precisa do valor dos 2 primeiros, isto é, é dependente do resultado de ambos. Embora a lin_solve fosse o hotspot óbvio, foram aplicadas, seguindo a mesma lógica, paralelização em outras partes do programa. Temos então presente em uso todas as diretivas faladas anteriormente faladas em outros loops. Mais uma vez, a utilizaçãp destas serve para, garantir que são processados em paralelo, para dividir de forma uniforme o workload das threads, e o **collapse(3)**, que serve para combinar os 3 loops em apenas 1 loop "conceptial", pois nestes já não existe dependência entre nehum deles. O reduction não foi usado pois não há atualização concorrente de variáveis compartilhadas, pois cada thread trabalha numa parte independente do problema.
Uma otimização feita em relação à entrega desta fase foi em vez de usar as diretivas de paralelismo em todos os loops removemo-las nas funções **add_source** e **set_bnd** pois paralelizar estas funções não compensa o overhead de gestão de threads, pois realizam cálculos muito simples e o número de elementos atualizados é pequeno.

### B. Análise de Escalabilidade / SpeedUP

Testes feitos nas máquinas com propriedade c=24, que são uma pool de 6 máquinas muito parecidas com a da partition

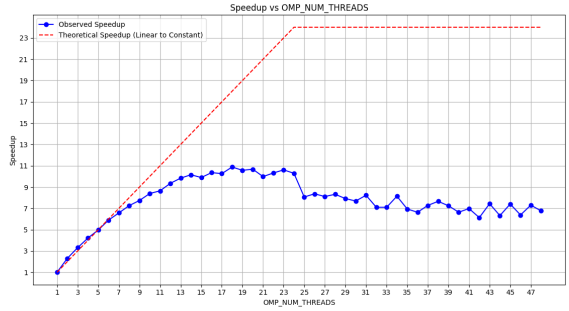


Fig. 1. Análise de SpeedUP

## IV. WA3 - IMPLEMENTAÇÃO EM CUDA

Nesta fase final, decidimos desenhar e implementar uma versão do nosso solver para aceleradores usando CUDA. Essa tecnologia permite explorar o poder de processamento das GPUs para realizar paralelização eficiente.

### A. Otimizações implementadas

Partimos do código do WA2, mas com uma alteração no tamanho da grelha (de 84×84×84 para 168×168×168). Na função main, modificámos as alocações de dados para que a memória de todas as variáveis estivessem em GPU. Também implementámos as funções clear_data (que as inicializa a zero na GPU) e free_data (que liberta a memória GPU antes alocado, no final da execução).

A ideia principal desta terceira parte foi alocar os recursos na placa gráfica logo no arranque, inicializá-los a zero e, depois, chamar os kernels CUDA, passando-lhes os ponteiros para essas áreas de memória no device. Isso reduz bastante a movimentação de dados entre host (CPU) e device (GPU), o que só é feito de forma exceptional — por exemplo, no início e no fim de cada iteração de **lin_solve**. Já no **fluid_solver**, mantivemos todas as funções essenciais, agora em CUDA, chamando kernels específicos de forma a maximizar a paralelização.

A maior parte das funções é uma tradução bastante fiel do OpenMP para CUDA. A **lin_solve**, porém, é onde reside a maior complexidade. Inicialmente, a função em OpenMP utilizava diretivas paralelas para distribuir as iterações dos loops sobre os índices i e j, atualizando apenas as células "red" e "black" alternadamente para evitar dependências entre elas. Na versão CUDA, essa lógica foi dividida em dois kernels distintos: um para as células "red" e outro para as "black".

Cada kernel mapeia cada ponto 3d para uma thread específica na GPU, garantindo que apenas as células apropriadas sejam atualizadas em cada step. Para a convergência, foi implementada uma função auxiliar **atomicMaxFloat** que permite calcular de forma eficiente o máximo das diferenças entre as iterações, utilizando operações atômicas para evitar data races. Em cada kernel, fazemos primeiro uma redução em **memória partilhada (shared memory)** para obter o valor máximo de variação naquele bloco; em seguida, o resultado parcial é combinado através de um **atomicMaxFloat**, atualizando o valor global na GPU.

O controlo com critério de paragem continua no host (CPU): após cada par de chamadas (red e black), copiamos o valor global de max_c da GPU para o CPU, e verificamos se este é menor que o limite de tolerância (ou se atingimos o máximo de iterações). Caso ainda não tenhamos convergido, repetimos o processo (kernel red + kernel black + boundary conditions), garantindo sempre a sequência correta de atualizações entre as células "vermelhas" e "pretas".

O uso do 8x8x8 = 512 threads ataca melhor o problema em 3D, que é um múltiplo do tamanho de warp (32) e adequado para a máquina. Isto permite uma distribuição equilibrada das threads na GPU, garantindo que todos os recursos sejam utilizados de forma optimizada Em termos de escalabilidade, a migração para CUDA oferece uma melhoria significativa no desempenho, especialmente para volumes maiores, onde o poder de paralalelismo da GPU pode e é aproveitado. Enquanto o OpenMP paraleliza os loops sobre os poucos cores de CPU disponíveis, a GPU lida com milhares de threads simultaneamente, resultando em acelerações consideráveis. No entanto, é importante notar que para problemas de tamanho pequeno o overhead de configuração e transferência de dados para a GPU é uma grande desvantagem e não deverá ser utilizado. Um bloco de 256 threads vai ter, em média, menos overhead de arranque do que um bloco de 512 threads, mas na prática, essa diferença não será assim tão notável e estes blocos muito grandes (512 ou 1024 threads) podem esgotar recursos mais depressa, reduzindo o número de blocos em paralelo, mas podem também ser vantajosos para certos padrões de memória. Em suma, esta abordagem espelha a implementação OpenMP, mas tira proveito do modelo de execução em GPU e do paralelismo massivo que o CUDA oferece. Blocos mais pequenos (por exemplo, 64 ou 128 threads) nem deverão pensados pois deixaram a GPU subutilizada. Reparamos que o uso de thread por Bloco que melhor obtia resultados era os de 8x8x8.

Numa continuação de análise, desta vez das demais funções migradas para CUDA, assume-se que os kernels estão apenas a trabalhar sobre apontadores, por isso alocação de memória (e free's) não serão das responsabilidade destas funções. Começando pela **add_source**, como estamos a trabalhar sobre um único array de tamanho fixo (size), justifica-se o uso de kernel 1D, onde cada thread trata de 1 elemento. Como justificação do uso de 256 threads por bloco, num kernel tão simples, um aumento para 512 poderá até ter resultados indesejados, devido a overhead.

Quanto ao **set_bnd**, optámos por um kernel 3D, onde cada thread irá receber um trio de coordernadas e aplica consoante correspondência. Quanto ao blockDim, o uso de 8x8x8 (512 threads) dá se a natureza de de uma, na nossa análise, boa decisão para divisão deste problema 3D. A diffuse mantêm-se igual, pois não existe qualquer oportunidade de aproveitamente de paralelismo A advect, ajustan corretamente os deslocamentos com base nas dimensões do grid e realiza a interpolação trilinear para atualizar os valores advectados. A escolha das dimensões dos blocos (8, 8, 8) é adequada, como já antes explicado para estes tipos de problemas.

A implementação da função **project_cuda** utiliza dois kernels distintos (**compute_divergence e update_velocity**) por motivos de separação de responsabilidades e dependências de dados., mantendo a mesma lógica antes existente. A escolha de blocos tridimensionais de 8x8x8 threads resulta em 512 threads por bloco, o promove uma distribuição equilibrada das threads, maximizando a utilização dos recursos da GPU e garantindo acessos eficientes à memória. A migração das funções **dens_step e vel_step** para CUDA foi feita de maneira muito direta, substituindo as chamadas das versões CPU pelas em CUDA, pois vemos estas apenas como orquestradoras. Esta abordagem mantém a estrutura lógica.

**Makefile**

Inspirada na Makefile do PL10, mantendo a mesma estrutura, verificamos que os tempos de exeucação aumentavam significamente com a substituição da flag O2 por O3.

### *B. Testes e análise de resultados*

Para analisar a performance da implementação em Cuda foram realizados alguns testes. Começamos por ver como o tamanho das matrizes afeta o desempenho do programa. De seguida usamos a ferramenta **nvprof** para analisar quais são as operações mais pesadas. Corremos o código no cluster e numa máquina local (NVIDIA GeForce GTX 1650)

**Análise de Escalabilidade**

TABLE III
TABELA DO TAMANHO DAS MATRIZES E SEUS TEMPOS DE EXECUÇÃO CLUSTER

| Tamanho da Matriz | Tempo de Execução |
|---|---|
| 42 | 613,48ms |
| 84 | 4,03s |
| 168 | 37,46s |

TABLE IV
TABELA DO TAMANHO DAS MATRIZES E SEUS TEMPOS DE EXECUÇÃO MÁQUINA LOCAL (NVIDIA GEFORCE GTX 1650)

| Tamanho da Matriz | Tempo de Execução |
|---|---|
| 42 | 1,087s |
| 84 | 3,003s |
| 168 | 17,93s |
| 336 | 182,913 |

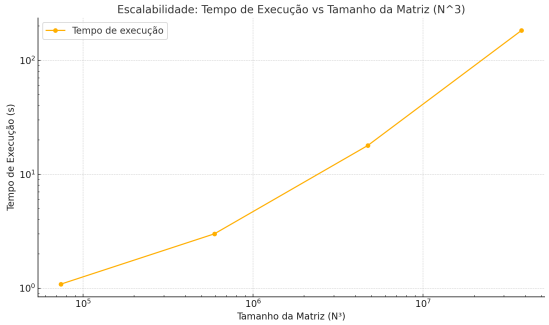


Fig. 2. Análise de Escabilidade Máquina Local

Podemos ver que, na máquina local a escabilidade não é boa quanto mais aumentamos o tamanho da matriz, mesmo se mantendo próxima de linear com valores de N mais pequenos, podemos ver que a relação entre $N^3$ (tamanho das matrizes) e o tempo de execução vai aumentando entre cada iteração o que indica que a escabilidade não é forte, visto que não se mantém constante com o crescimento do tamanho das matrizes. Isto deve-se devido ao facto de que a implementação de kernel gerá cargas desbalanciadas e usa operações atómicas, com valores mais altos de N esses desbalanciamentos também aumentam e também o número de operações atómicas que não podem ser paralelizadas.

Embora as GPU's sejam feitas para este tipos de trabalhos, em que existe grande paralelização, chegamos a valores de tempos de execução que não representam a teoria.
Talvez por bottlenecks por nos impostos, com deficiências na implementação em CUDA.

**Análise das operações mais pesadas**

TABLE V
ANÁLISE DAS OPERAÇÕES MAIS PESADAS

| Operação | Tempo Total | Percentagem tempo | Chamadas |
|---|---|---|---|
| lin_solve_kernel_black | 7,01724s | 39,15% | 7127 |
| lin_solve_kernel_red | 7,00829s | 39,10% | 7127 |
| set_bnd_kernel | 2.01004s | 11,21% | 8527 |
| memcpy DtoH | 763,15ms | 4,26% | 7228 |
| update_velocity | 372,29ms | 2,08% | 200 |

Existem algumas possíveis razões para o domínio completo de tempo de execução observado pelos "lin_solve_kernel_red", "lin_solve_kernel_black" e "set_bnd_kernel". Primeiro, a lin_solve utiliza um método iterativo, com dependências ,que exige múltiplas iterações para convergir. Isto resulta em números elevados de acessos memória global, pois cada atualização depende de valores nas células vizinhas, tornando o kernel sensível à largura de banda da memória. Além disso, a separação em duas fases, red e black, implica o lançamento de dois kernels distintos para cada iteração, aumentando o overhead de sincronização e comunicação entre os kernels. Já a função set_bnd_kernel, responsável pela aplicação das condições de contorno, também apresenta acessos intensivos à memória global, muitas vezes de forma não coalescente e que talvez podesse ser melhor implementada. Essa falta de coalescência nos acessos pode impactar negativamente o desempenho, já que a GPU não consegue agrupar eficientemente as os reads e writes que acontecem. Outra razão poderá ser a de, em problemas de menor escala, as operações podem não explorar devidamente os recursos da Gráfica, resultando em um mau uso desta mesma. Para concluir, a combinação de overheads de sincronização, dependências em memória global e possíveis ineficiências no paralelismo/más implementações das funções faz com que lin_solve(os dois kernels) e set_bnd dominem o tempo total de execução do programa.

## V. CONCLUSÃO

Ao longo das três etapas deste projeto realizadas neste semestre, conseguimos analisar a importância do paralelismo na otimização de código. Utilizando diferentes técnicas que aprendemos, conseguimos verificar e observar na prática o quanto o desempenho do código inicial, fornecido pelos professores, foi aprimorado.

Na **primeira fase**, as técnicas aplicadas no código (alterações no algoritmo, melhorias em ILP, hierarquia de memória, e vetorização) resultaram em um tempo de execução muito melhor em comparação ao código original, alcançando assim nosso objetivo principal.

Na **segunda fase**, ao aplicar **OpenMP** ao código desenvolvido na primeira fase, conseguimos obter um desempenho ainda melhor em termos de tempo de execução.

Por fim, na **terceira tarefa**, exploramos a plataforma de computação paralela **CUDA**. Apesar de não termos superado a versão com **OpenMP**, conseguimos, mesmo assim, superar a versão sequencial. Além disso, a complexidade dessa tarefa motivou-nos bastante a estudar mais a fundo esta plataforma, com o objetivo de buscar a melhor solução possível.