Computação Paralela

3D Fluid Simulation

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

Filipe Pereira
Departamneto 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 centra-se na otimização de um código de simulação de dinâmica de fluidos em 3D, recorrendo a ferramentas de *profiling* para identificar os principais gargalos de desempenho. As técnicas de otimização implementadas incluem a reorganização de ciclos, *loop unrolling*, o particionamento em blocos e a vetorização, com o objetivo de reduzir o tempo de execução. A avaliação do desempenho foi realizada através da análise de métricas como o tempo de execução e a eficiência na utilização da cache. Estas melhorias permitiram otimizar o desempenho global do programa, tornando-o mais eficiente.**

I. Introdução

Este trabalho foi desenvolvido no âmbito da Unidade Curricular de Computação Paralela, com o objetivo de analisar e otimizar um código de simulação 3D de dinâmica de fluidos, de *Jos Stam*. Neste relatório são abordadas as técnicas de otimização aplicadas, visando melhorar o desempenho em termos de tempo de execução. Para tal, foram utilizadas ferramentas de análise e *profiling*, como o *gprof* e o *perf*, com o intuito de identificar os blocos de código mais críticos em termos de consumo de tempo. Após essa análise, foram implementadas otimizações para reduzir o tempo de execução, sem comprometer a integridade da simulação e preservando a legibilidade do código.

II. Análise de Código

A. Profiling

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***.

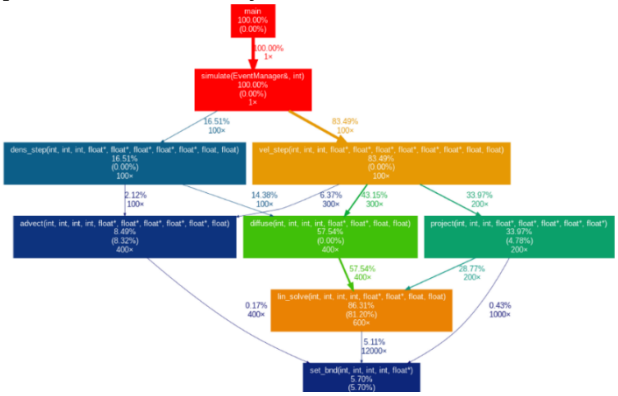

Figura 1- Call Graph obtido

III. Otimizações (1)

A. 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. Ao reorganizar a ordem dos ciclos (de **i,j,k** para **k,j,i**), o acesso aos elementos foi estruturado de forma a garantir que os elementos ***x[i][j][k]*** fossem lidos de maneira contígua na memória, melhorando a localidade espacial e reduzindo os *cache misses*. É importante salientar que esta otimização foi aplicada em todas as funções do ficheiro "***fluid_solver.cpp***".

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. O tamanho dos blocos foi definido como 4, devido à arquitetura do hardware que utiliza as extensões ***SSE (Streaming SIMD Extensions)***. Estas extensões operam com registos de 128 *bits*, permitindo o processamento simultâneo de 4 *floats* de 32 *bits*. Ao otimizar o código para processar 4 *floats* de cada vez, conseguimos tirar melhor partido da capacidade de processamento paralelo do hardware, aumentando o desempenho geral do programa.

Mesmo após esta alteração, a vetorização do código não foi possível devido à dependência de dados entre as iterações do ciclo. Como o valor de cada elemento depende dos seus vizinhos, os valores de ***x [i, j, k]*** são lidos de posições como ***x [i-1, j, k]***, ***x[i+1, j, k]***, etc., o compilador não consegue garantir que não haverá interferência entre as iterações durante a execução de instruções vetorizadas, o que impede a paralelização segura dessas operações.

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. Além disso, foi fundamental para o nosso entendimento do tema e permitiu-nos aplicar as técnicas estudadas nas aulas práticas, razão pela qual a incluímos em anexo.

IV. Otimizações (2)

A. *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. Esta técnica não apenas reduz significativamente o *overhead* do ciclo, diminuindo o número de instruções de controlo, mas também permite a paralelização das operações sem dependências, algo que não era viável na implementação anterior. Esta combinação de fatores contribuiu consideravelmente para a diminuição do tempo de execução.

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, com exceção o **i - 1** que é dependente, (ao longo do *loop i*) antes das operações de multiplicação e divisão. A vetorização foi realizada sem comprometer os valores finais, pois ocorre antes da paralelização das instruções subsequentes que utilizam os valores calculados. Esta técnica transforma operações escalares em operações vetoriais, permitindo que múltiplos dados sejam processados simultaneamente. Com a vetorização, conseguimos alcançar alguns ganhos de desempenho.

Para garantir que o nosso código foi corretamente vetorizado, utilizámos o "*Compiler Explorer*" para analisar o *assembly* gerado. Após esta análise, confirmámos que a vetorização foi efetivamente realizada. Abaixo segue um excerto do código assembly:

```
vmovups ymm8, YMMWORD PTR [r10+rax]
vmovups ymm10, YMMWORD PTR [r8+rax]
vaddps  ymm9, ymm8, YMMWORD PTR [r11+rax]
vaddps  ymm11, ymm10, YMMWORD PTR [r9+rax]
```

V. Flags Utilizadas

Para maximizar o desempenho do código, foram utilizadas várias flags de otimização durante a compilação com o g++:

- ***-O3***: Esta flag ativa a terceira e mais agressiva otimização do compilador. Ele tenta gerar o código mais rápido possível, ativando diversas técnicas de otimização.
- ***-funroll-loops***: Desenrola os loops, transformando-os em operações repetidas para evitar a sobrecarga do controlo de loops.
- ***-msse4***: Habilita o uso de instruções ***SSE4 (Streaming SIMD Extensions 4)***, otimizando operações de ponto flutuante e manipulação de dados vetoriais.
- ***-ffast-math***: Ativa otimizações agressivas para cálculos matemático melhorando a performance, especialmente em operações de ponto flutuante.
- ***-march=native***: Esta flag ajusta as instruções geradas especificamente para a arquitetura do processador local, utilizando todas as otimizações disponíveis para o hardware em uso.
- ***-flto***: Habilita a otimização em nível de link (link-time optimization), permitindo que o compilador realize otimizações adicionais no código durante o processo de linkagem, melhorando ainda mais o desempenho geral do programa.

VI. Conclusão

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. Embora a vetorização não tenha sido viável em todas as partes do código devido à dependência de dados, a vetorização, juntamente com a implementação do *loop unrolling* e a utilização de *flags* de compilação, contribuiu para maximizar o desempenho. Acreditamos ter cumprido todos os objetivos propostos para este trabalho. Este estudo reforça a importância da otimização de código na perspetiva do paralelismo e ilustra o impacto direto das técnicas de otimização de alto nível na eficiência computacional. Além disso, foi fundamental aplicar as técnicas aprendidas nas aulas e observar na prática os seus efeitos.