Optimização 3dfluid

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

I. Análise

Iniciamos a resolução do nosso problema com o profiling do programa que tinhamos em mão.

Decidimos começar por gerar um call-graph de modo a perceber como cada função afetava o tempo de execução greal. Utilizamos o gprof para gerar o ficheiro gmon.out que continha os dados para o profiler e tambem gerar o main.gprof para o seu respetivo relatório. Por fim, utilizamos a ferramenta, gprof2dot para termos uma representação visual do nosso call-graph.

Através da análise do *call-graph*, percebemos que o programa passa 87,87% do tempo total de execução na função `lin_solve`, sendo que 85,53% do tempo corresponde à execução interna da função, e 2,61% do tempo é gasto na execução da função `set_bnd`.

Assim podemos concluir que o `lin_solve` será o nosso hotspot e deverá ser o ponto centrar da nossa análise.

Analisando a função lin_solve, percebemos que esta função recebe 8 argumentos, sendo que os argumentos M,N, e O são utilizados como definição de tamanho dos ciclos for, os apontadores *x e *x0 são atulizados no hotstop e a, b e c, são constantes utilizadas para cálculos, a e c, ou para passar à função auxiliar, b.

Analisando o codigo, percebemos que estamos perante 4 ciclos for, sendo o ciclo mais externo defino pelo macro #define LINEARSOLVERTIMES 20, deste modo tem tamanho constante. Já os 3 seguintes são definidos pelo tamanho das variáveis, definidas na main com SIZE igual a 42, logo o nosso programa, no hotstop irá realizar 20 * 42³ iterações.

Também vemos a presença repetida do macro IX(i,j,k) definido como

```
1 #define IX(i, j, k) ((i) + (M + 2) * (j) + (M + 2) *
      (N + 2) * (k))
```

Listing 1. Definição da macro IX

Analisado o macro percebemos que em termos de localidade, percebemos que o i é a variável com a maior localidade espacial, seguida de j que será calculada a (M + 2) * j e por fim k que será guardado em (M + 2) * (N + 2) * k.

Analisando o ponto critico do código percebemos que precisamos de obter por cada ciclo 3 versões da mesma variável, por exemplo, para i, vamos precisar de i, i -1 , i +1. Esta dependência de dados do próximo e do anterior, leva a uma barreira muito grande no processo de vetorização.

II. Optimize

Antes de começar as otimizações decidimos definir os valores que queríamos tabelar no sentido de perceber melhor todas as alterações que fazíamos ao código. Focamo-nos em ciclos, numero de instruções, cache-misses, L1-dcache-loads, L1-dcache-loads-misses, L1-dcache-store, L1-dcache-store-misses e tempo de compilação. Com estas métricas podemos obter facilmente o CPI e a % de cache misses, #I e o TCC, sendo assim mais fácil percebermos exatamente quais eram os efeitos de cada alteração.

De seguida, realizamos testes no código não alterado para utilizarmos como valores de referência, usando apenas a compilação sem qualquer flag e com -O2.

Version	Time	CPI	#I	L1_DMiss	%cache misses
Sem flags	53,78	0,537	1,66e10^{10}	2,31e10^{9}	3,29
-O2	21,520	0,531	1,88e10^{10}	2,32e10^{9}	31,79

Pela analise que tínhamos realizado no profiling, percebemos que a variável que seria a causa do maior cache misses era o k, logo esta deveria-se encontrar no ciclo mais exterior. Por sua vez a variável i, que é a que apresenta menor taxa de cache misses, devia ser colocada no ciclo mais interior. Esta alteração dos ciclos leva a um aumento da localidade espacial do nosso código, já que a probabilidade de existir na cache a próxima variável que o meu código precisa aumenta consideravelmente.

De seguida, percebemos que o nosso macro, IX(i, j , k) iria precisar do valor de 42³ * 3 variáveis, todos os valores de i, j e k mais os respetivos valores a seguir e anteriores. Por isso, a implementação de blocos iria trazer bastante localidade espacial e temporal ao nosso código, sendo este o próximo passo tomado.

Version	Time	CPI	#I	L1_DMiss	%cache misses
-O2	8,909	1,043	2,61e10^{10}	4,98e10^{8}	5,05

Com os blocos implementados, temos de descobrir qual é o tamanho otimizado para os blocos, sendo que o tamanho dos ciclos for é de 42. O tamanho ideal será um numero divisível por este, sendo as nossas melhores opções 2, 3 e 6. Contudo, na fase de testes fizemos uma má interpretação do tamanho do bloco, achávamos que era 44, o que levou à utilização do tamanho de bloco 4 em todos os nossos testes com diferentes flags.

Version	Time	CPI	#I	L1_DMiss	%cache misses
-O2(2)	9,0475	0,765	$3{,}81e10^{10}$	$4{,}80e10^{8}$	2,87
-O2(3)	8,656	0,938	$2{,}96e10^{10}$	$4{,}89e10^{8}$	3,87
-O2(4)	8,909	1,043	$2{,}61e10^{10}$	$4{,}98e10^{8}$	5,05
-O2(6)	9,010	1,210	$2{,}41e10^{10}$	$4{,}74e10^{8}$	5,30
-O2(8)	9,236	1,335	$2{,}24e10^{10}$	$4{,}86e10^{8}$	5,97

Analisando a tabela percebemos que blocos de proporção mais pequenas, como 2 e 3, possuem um maior número de instruções e por sua vez, menos cache-misses, pois a localidade espacial e temporal está maximizada, principalmente em blocos de tamanho 2. Contudo, o aumento do número de instruções leva a um maior TCC do programa, sendo que esta é a métrica que mais estamos a valorizar.

Os blocos de maiores proporções fazem menos acessos à memoria porque como os blocos são maiores é aproveitado todos os i, j e k que estão em cache, aumentando a percentagem de cache misses, contudo como realizam menos instruções, melhoramos consideravelmente o tempo de execução do programa.

Qualquer alteração ao código em relação à vetorização não foi implementada pelos motivos descritos na analise do código realizada no inicio do projeto.

A. Implementação de Flags

Após as otimizações de código, passamos a otimizá-lo utilizando flags de compilação. A primeira flag que testamos foi a flag de -funroll-loops vs -funroll-all-loops. A flag de loop unrolling é a -funroll-loops, onde o compilador analisa o ciclo e confirma se fazer loop unroll do mesmo é seguro e caso isso aconteça o compilador usa essas oportunidades para dar unroll de um loop. Já o funroll-all-loops, faz com que o compilador procure ainda mais oportunidades para fazer unroll de ciclos no seu código Assembly.

O loop unrolling, à priori, traz algumas consequências como o aumento de CPI e número de misses e uma diminuição do número de instruções.

Version	Time	CPI	#I	L1_DMiss	%cache misses
-O2 -funroll -loops	8,452	0,851	$3{,}02e10^{10}$	$4{,}92e10^{8}$	4,91
-O2 -funroll -all-loops	7,994	0,853	$3{,}02e10^{10}$	$4{,}87e10^{8}$	4,85

É de salientar que a implementação do unroll de loops não trouxe a totalidade dos resultados esperados, foi verificado um aumento do CPI e do número de misses do nosso programa, como esperado, contudo a diminuição do número de instruções não se fez sentir em todos os testes realizados. A perceção obtida pelos elementos do grupo foi que o loop unroll apenas diminuía o número de instruções de forma consistente quando o código não estava extremamente otimizado. Concluindo então que o tamanho dos blocos teria impacto nas otimizações do loop unrolling. Contudo, mesmo não diminuindo o número de instruções o -funroll-all-loops melhorou consideravelmente o desempenho do nosso código.

Mesmo já tendo analisado que o ponto crítico do nosso código não seria vectorizável, decidimos testar na mesma a flag -ftree-vectorize e concluir como esta se relacionava com a -funroll-all-loops.

Version	Time	CPI	#I	L1_DMiss	%cache misses
-O2 (4)	8,909	1,043	$2{,}61e10^{10}$	$4{,}98e10^{8}$	5,05
-O2 -ftree -vectorize	8,484	1,042	$2{,}59e10^{10}$	$4{,}86e10^{8}$	4,95
-O2 -ftree -vectorize -funroll -all-loops	8,564	0,850	$3{,}01e10^{10}$	$4{,}84e10^{8}$	4,85

Após a análise das flags de loop unrolling e vetorização iremos tentar otimizar ao máximo o nosso programa, até agora utilizamos o -O2 como flag base de comparação, agora iremos testar o -O3 e o -Ofast.

Em relação ao -O2, o -O3 aumenta o inlining, reduzindo o overhead da função e também em mais agressivo em otimizações como loop unrolling. Já o -Ofast em relação ao -O3, utiliza todas as otimizações do -03 contudo é menos rigoroso com os cálculos e exceções em operações matemáticas, podendo causar pequenos desvios numéricos. Contudo, o -Ofast diminui substancialmente o tempo de execução.

Version	Time	CPI	#I	L1_DMiss	%cache misses
-O2 -funrolls -all-loops	7,9946	0,853	$3{,}02e10^{10}$	$4{,}87e10^{8}$	4,85
-O3 -funroll -all-loops	8,534	0,913	$2{,}91e10^{10}$	$4{,}95e10^{8}$	3,80
-Ofast -funroll -all-loops	5,146	0,517	$3{,}16e10^{10}$	$4{,}92e10^{8}$	3,77
-Ofast -funroll -all-loops -ftree-vectorize	5,178	0,515	$3{,}15e10^{10}$	$4{,}92e10^{8}$	3,77

Todos estes dados foram obtidos com blocos de tamanho 4, contudo, decidimos testar os blocos de tamanho 3 e 6, pois estes são divisores de 42, logo deviriam produzir melhores resultados do que blocos de tamanho 4.

Version	Time	CPI	#I	L1_DMiss	%cache misses
-Ofast -funroll -all-loops (3)	5,250	0,510	$3{,}14e10^{10}$	$4{,}98e10^{8}$	3,73
-Ofast -funroll -all-loops (4)	5,146	0,517	$3{,}16e10^{10}$	$4{,}92e10^{8}$	3,77
-Ofast -funroll -all-loops (6)	4,707	0,685	$2{,}23e10^{10}$	$4{,}80e10^{8}$	5,02

Analisando os dados da tabela percebemos que o melhor tempo de execução obtido neste projeto foi de 4,707 segundos, com a alteração da ordem dos ciclos for, implementação de blocos de tamanho 6 e das flags de compilação -Ofast e -funroll-all-loops. Esta solução apresenta uma percentagem de cache misses de 5,02%, sendo que a melhor % de cache misses no trabalho pratico foi de 2,87% com a utilização de blocos de tamanho 2.

Também é de salientar realizamos outro call-graph para a versão com -Ofast -funroll-all-loops (4) e obtendo um uma percentagem de execução dentro da lin_solve de 36,26%, uma melhoria significativa aos, 85,53% obtidos inicialmente.