Relatório CI164 - Trabalho Prático (2/2)

Rafael Ravedutti Lucio Machado
Departamento de Informática – Universidade Federal do Paraná
rrlm13@inf.ufpr.br

1. Análise Geral

1.1 Relatório do Valgrind

==14902== Memcheck, a memory error detector
==14902== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==14902== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==14902== Command: ./gradSolver -r 512 graphics/valgrind.out
==14902==
[Saída do programa]
==14902==
==14902== HEAP SUMMARY:
==14902== in use at exit: 0 bytes in 0 blocks
==14902== total heap usage: 2 allocs, 2 frees, 2,113,536 bytes allocated
==14902==
==14902== All heap blocks were freed -- no leaks are possible
==14902==
==14902== For counts of detected and suppressed errors, rerun with: -v
==14902== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Não houve nenhum problema reportado pelo Valgrind.

1.2 Likwid-Topology

CPU type: AMD Interlagos processor
****
Hardware Thread Topology
****
Sockets: 1
Cores per socket: 6
Threads per core: 1

HWThread Thread Core Socket

0	0	0	0
1	0	1	0
2	0	2	0
3	0	3	0
4	0	4	0
5	0	5	0

Socket 0: (0 1 2 3 4 5)

****

Cache Topology

****

Level: 1

Size: 16 kB

Cache groups: (0) (1) (2) (3) (4) (5)

Level: 2

Size: 2 MB

Cache groups: (0 1) (2 3) (4 5)

Level: 3

Size: 8 MB

Cache groups: (0 1 2 3 4 5)

****

NUMA Topology

****

NUMA domains: 1

Domain 0:

Processors: 0 1 2 3 4 5

Relative distance to nodes: 10

Memory: 689.504 MB free of total 3678.97 MB

1.3 Maior sistema linear possível

Temos que 3678.97 MB = 3678970000 bytes

Embora nunca tenhamos toda a memória disponível, vamos calcular para este caso. O programa irá alocar memória para a matriz e os vetores dependentes de N pela seguinte fórmula (em bytes):

$$F(n) = 8(n^2 + 4n)$$

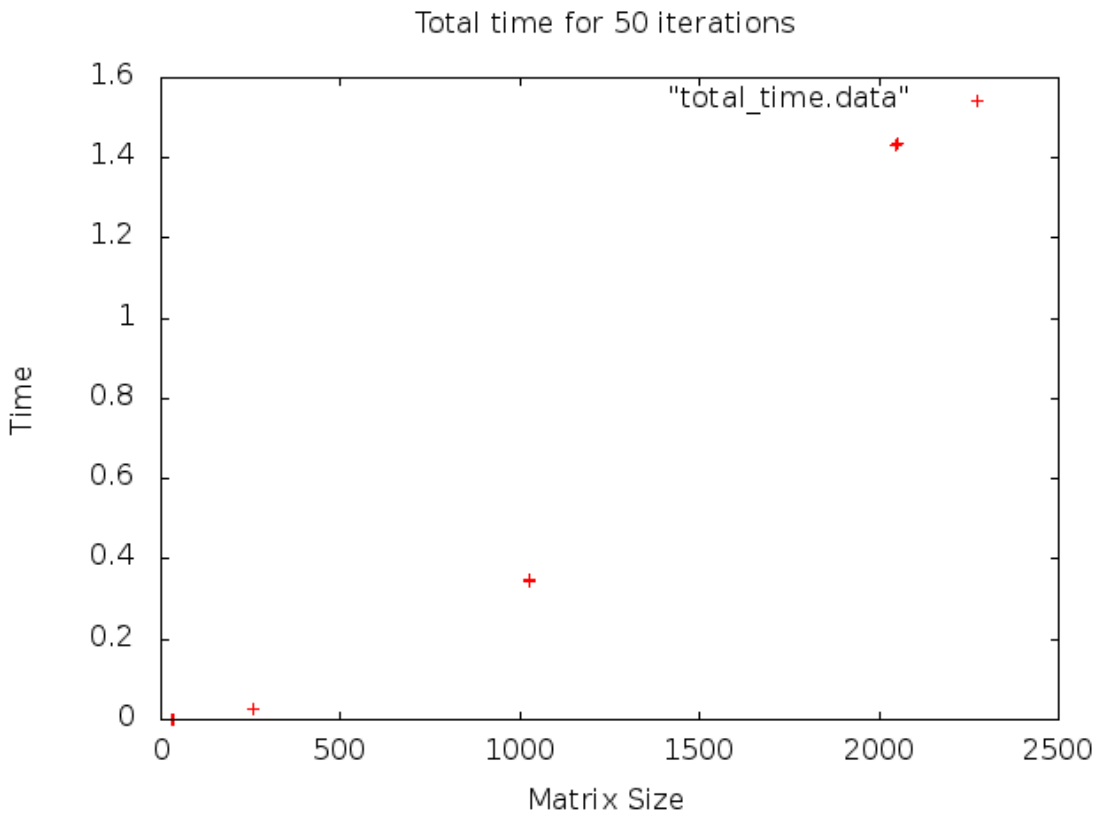
Onde n^2 é a quantidade de elementos da matriz A, e 4n são para os vetores b, x, residuo e direção. Multiplicamos a quantidade de elementos por 8 que é o tamanho em bytes ocupado por um double (pois cada elemento é um double). Portanto não podemos exceder este limite e o maior sistema linear será dado quando F(x) = 3678970000, ou seja:

$$8(n^2 + 4n) = 3678970000 \Rightarrow n^2 + 4n = 459871250$$

Resolvendo o sistema, temos como resultado $n \sim= 30325$

Isso significa que se tivessemos toda a memória disponível, então poderiamos resolver no máximo um sistema linear de dimensão 30325x30325 nesta máquina, dada sua limitação de memória.

1.4 Gráfico de tempo para 50 iterações



2. Análise de Funções

2.1 FLOPs em função do tamanho N da matriz

Lambda:

Temos um laço de 0 a N que possui um laço de 0 até i (onde i é o valor de iteração do primeiro laço) e executa mais 3 FLOPS (uma soma e duas multiplicações). No laço interno, temos 4 FLOPS por iteração (uma soma e três multiplicações), então temos $3N + 4(N(N-1) / 2)$

Temos mais dois laços de 0 a N que executam 2 FLOPS (uma soma e uma multiplicação), então temos mais 2N e mais 2 FLOPS para calcular alpha e beta. Contudo ficamos com a função:

$$F(n) = 4\frac{n(n-1)}{2} + 5n + 2 = 2n^2 + 3n + 2 \text{ FLOPS/iteração}$$

Resíduo:

Temos um laço de 0 a N que possui outro laço de 0 a N onde são executados 3 FLOPS (uma soma e duas multiplicações) e também (no primeiro laço) são executados mais 3 FLOPS (uma subtração, uma soma e uma multiplicação). Contudo ficamos com a função:

$$F(n) = 3n^2 + 3n \text{ FLOPS/iteração}$$

2.2 Memória utilizada em função do tamanho N da matriz

Como visto na seção 1.3, a quantidade de bytes alocados para matrizes e vetores dependentes de n (i.e. A, b, x, direção e resíduo) é dada pela seguinte função:

$$F(n) = 8(n^2 + 4n)$$

Além disso, o resto das variáveis declaradas (não alocadas dinamicamente) irão ocupar mais 224 bytes de memória, portanto podemos definir mais precisamente F como:

$$F(n) = 8(n^2 + 4n) + 224$$

2.3 Tempos de execuções

Lambda:

Versão nova

256: 0.00418433s (sem O3) e 0.00170185s (com O3)

257: 0.00467163s (sem O3) e 0.00163901s (com O3)

1024: 0.0262712s (sem O3) e 0.00987112s (com O3)

1025: 0.0261455s (sem O3) e 0.0114698s (com O3)

2048: 0.0898832s (sem O3) e 0.0353183s (com O3)

2049: 0.0826276s (sem O3) e 0.0364685s (com O3)

Versão antiga

256: 0.00584757s (sem O3) e 0.00233204s (com O3)

257: 0.00592496s (sem O3) e 0.00240595s (com O3)

1024: 0.0347276s (sem O3) e 0.0169681s (com O3)

1025: 0.0349079s (sem O3) e 0.0166864s (com O3)

2048: 0.117569s (sem O3) e 0.0435134s (com O3)

2049: 0.11927s (sem O3) e 0.0471167s (com O3)

Residuo:

Versão nova

256: 0.00208043s (sem O3) e 0.000493278s (com O3)

257: 0.00208091s (sem O3) e 0.000500055s (com O3)

1024: 0.0154761s (sem O3) e 0.00582406s (com O3)

1025: 0.013083s (sem O3) 0.00700858s (com O3)

2048: 0.0516177s (sem O3) e 0.0217941s (com O3)

2049: 0.0468468s (sem O3) e 0.0220392s (com O3)

Versão antiga

256: 0.00331835s (sem O3) e 0.000985933s (com O3)

257: 0.00335837s (sem O3) e 0.0010013s (com O3)

1024: 0.0201977s (sem O3) e 0.0107585s (com O3)

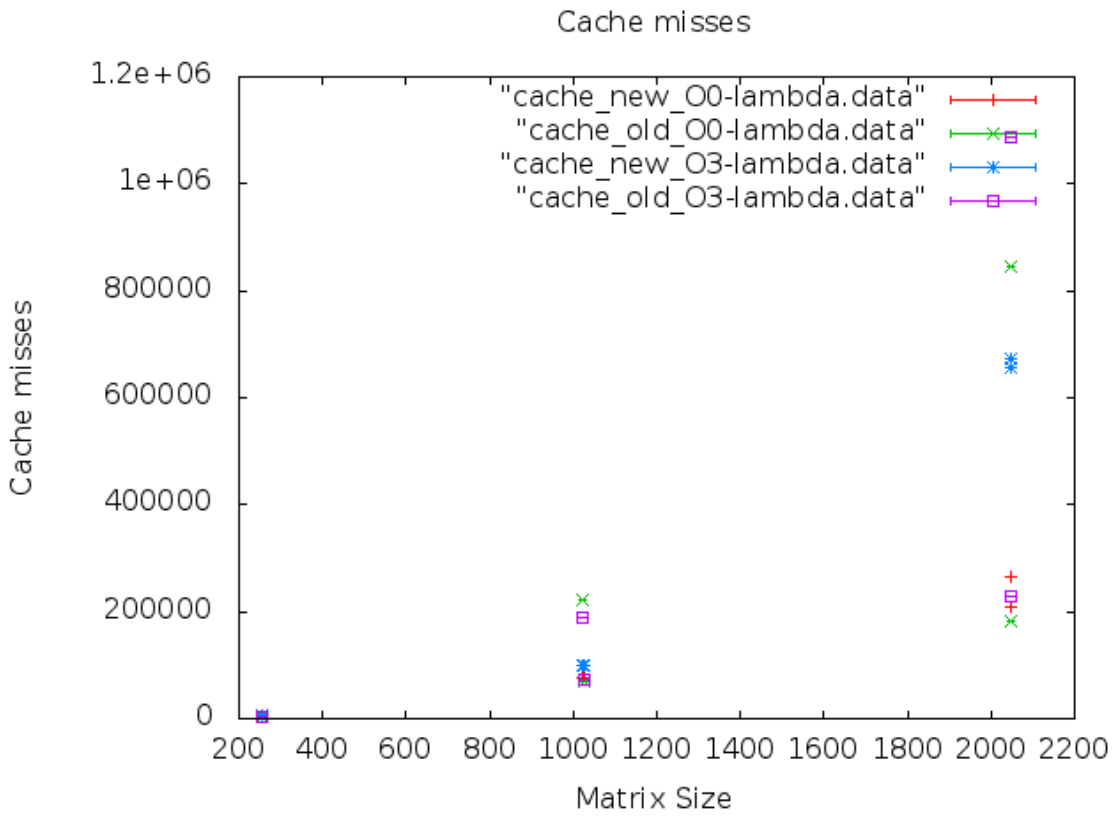
1025: 0.020273s (sem O3) e 0.0104007s (com O3)

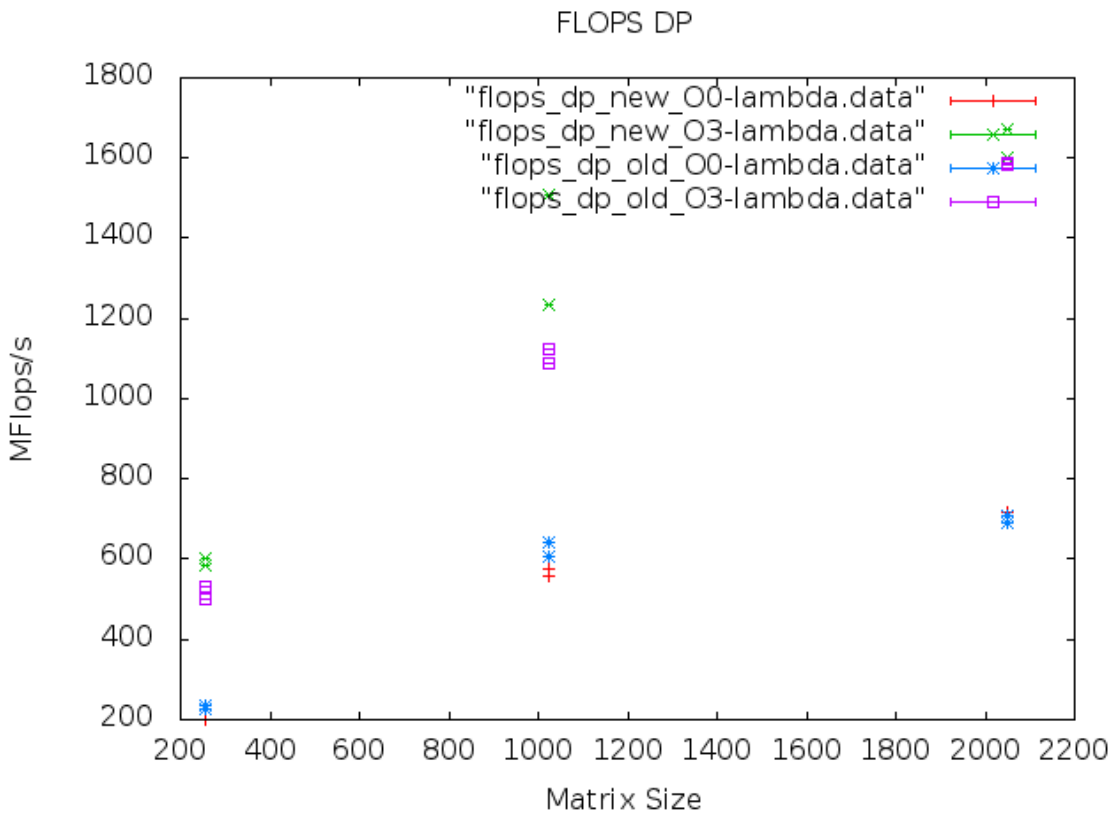
2048: 0.0741258s (sem O3) e 0.0278479s (com O3)

2049: 0.0746235s (sem O3) e 0.0279071s (com O3)

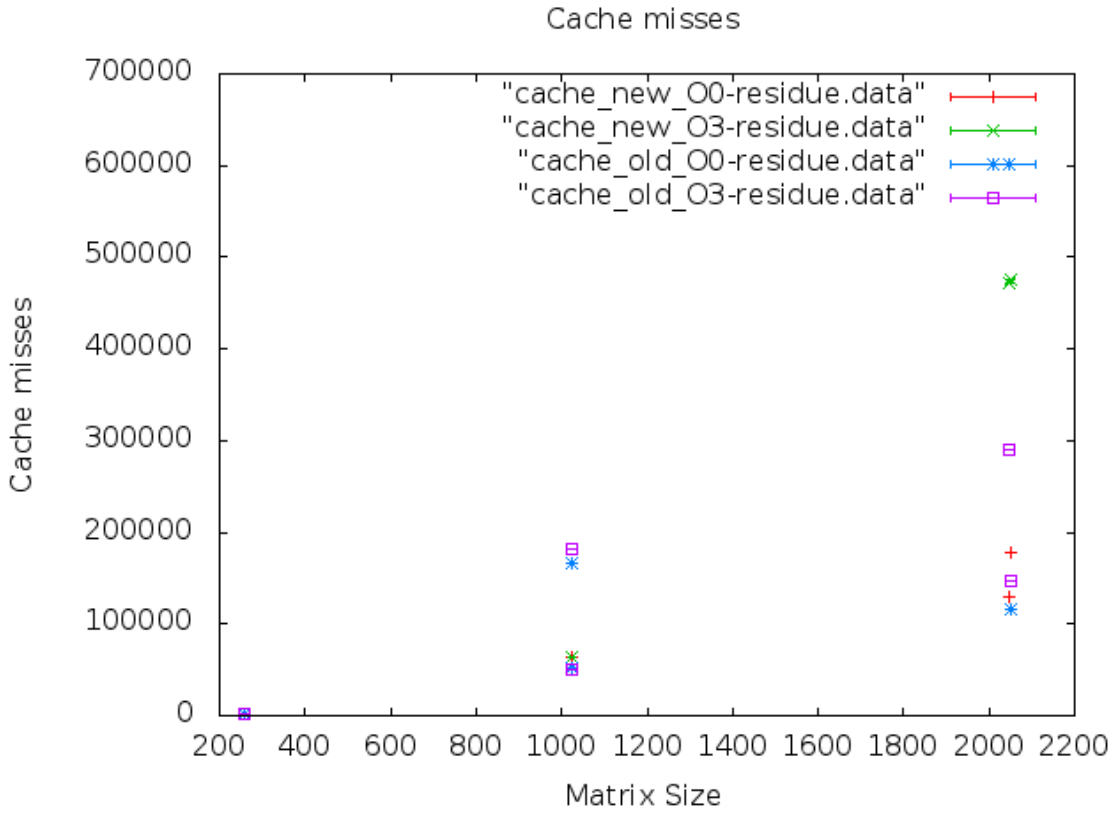
2.4 Gráficos

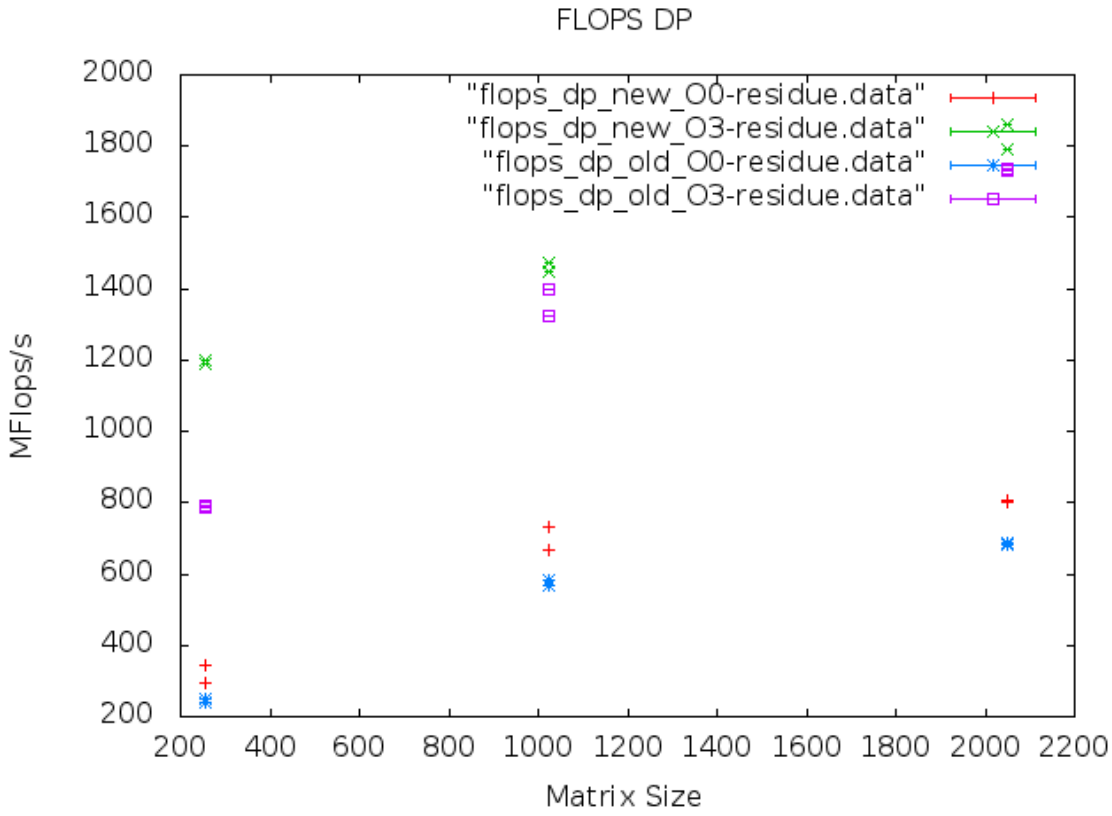
Lambda:


Cache misses
"cache_new_O0-lambda.data"
"cache_old_O0-lambda.data"
"cache_new_O3-lambda.data"
"cache_old_O3-lambda.data"
Cache misses
Matrix Size
1.2e+06
1e+06
800000
600000
400000
200000
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200


FLOPS DP
"flops_dp_new_O0-lambda.data"
"flops_dp_new_O3-lambda.data"
"flops_dp_old_O0-lambda.data"
"flops_dp_old_O3-lambda.data"
MFlops/s
Matrix Size
1800
1600
1400
1200
1000
800
600
400
200
200
400
600
800
1000
1200
1400
1600
1800
2000
2200

Resíduo:


Cache misses
"cache_new_O0-residue.data"
"cache_new_O3-residue.data"
"cache_old_O0-residue.data"
"cache_old_O3-residue.data"
Cache misses
Matrix Size
0
100000
200000
300000
400000
500000
600000
700000
200
400
600
800
1000
1200
1400
1600
1800
2000
2200


FLOPS DP
"flops_dp_new_O0-residue.data"
"flops_dp_new_O3-residue.data"
"flops_dp_old_O0-residue.data"
"flops_dp_old_O3-residue.data"
MFlops/s
Matrix Size
200
400
600
800
1000
1200
1400
1600
1800
2000
2200

Nos gráficos de cache para a versão antiga do programa, ocorre um salto grande entre os números que são potências de 2 e seus sucessores, por exemplo, de 2048 para 2049, temos um valor muito maior de cache misses (e consequentemente demoramos muito mais) em 2048 (entre 1000000 e 1200000) do que para 2049 (entre 200000 e 400000). Isto acontece porque estes tamanhos são múltiplos do número de blocos da cache, e consequentemente, eles serão levados ao mesmo índice em todas as linhas da matriz, gerando um cache miss todas as vezes que formos carregar um novo bloco da matriz para a cache. Em outras palavras, só estamos utilizando nestes casos um bloco da cache devido aos endereços de memória de cada bloco da matriz e por isso aumentamos o cache miss significativamente.

No programa novo, isso foi corrigido detectando quando o tamanho da matriz é uma potência de 2, a partir disto, alocamos mais um espaço de coluna na memória para ajeitar os endereços dos elementos da matriz. Assim, quando muda-se a linha da matriz, os próximos endereços já serão armazenados em outros índices da cache. Na versão compilada com a opção O3 este problema não foi otimizado.

Quanto a taxa de FLOPS do programa, a nova versão melhorou em relação a antiga, isto ocorreu pois foram realizados uma série de Loop Unrollings e isto também permitiu o uso da vetorização SIMD (i.e. várias instruções do programa foram executadas em paralelo). Nas versões compiladas com a opção O3 o aumento de instruções de operações em ponto flutuante (FLOPS) foram aumentadas mais ainda. Além da vetorização, os Loop Unrollings também fizeram com que algumas operações fossem executadas 4 vezes menos do que antes (isto porque as iterações são feitas de 4 em 4, e então armazenamos os valores das 4 iterações para essas operações, ao invés de 1 em 1).