Programação de Alto Desempenho

Atividade 2 - Otimizando o desempenho de códigos para afinidade de memória

Lucas Santana Lellis - 69618
PPGCC - Instituto de Ciência e Tecnologia
Universidade Federal de São Paulo

I. Introdução

Nesta atividade foram realizados experimentos relacionados com a otimização do desempenho de algoritmos quanto à afinidade de memória. Cada experimento foi realizado 5 vezes, e os resultados apresentados são a média dos resultados obtidos em cada um deles.

Todos os programas foram feitos em C, com otimização -O3, utilizando a biblioteca PAPI para estimar o tempo total de processamento, quantidade de chache misses em memória cache L2 (PAPI_L2_DCM), e o total de operações de ponto flutuante (PAPI_DP_OPS).

As especificações da máquina utilizada estão disponíveis na Tabela I.

Tabela I: Especificações da Máquina

CPU	Intel Core i5 - 3470
Cores	4
Threads	4
Clock	3.2 GHz
Cache L3	6144 KB
Cache L2	256 KB * 4
Hardware Counters	11
RAM	8 Gb
SO	Fedora 24
Kernel	4.7.2-201.fc24.x86_64
GCC	6.1.1 20160621

II. Experimento 1 - Multiplicação Trivial

Nesse experimento foi implementado o algoritmo tradicional para multiplicação de matrizes, sem blocagem, para verificar a diferença no desempenho causada pela mudança da hierarquia dos laços: IJK, IKJ, JIK, JKI, KIJ e KJI.

Na Tabela II temos um resumo dos experimentos realizados, onde percebemos a clara vantagem que as hierarquias IKJ e KIJ têm sobre as demais. Desses resultados podemos perceber que o modelo KIJ é mais eficiente para matrizes de 128x128, enquanto o modelo IKJ se sobressai nas demais.

Em experimentos anteriores, o PAPI chegou a falhar na contagem de operações de ponto flutuante, uma vez que era utilizada a propriedade "PAPI_FP_OPS". Sabe-se que esta propriedade apresenta problemas em contar operações vetoriais, e por este motivo, acredita-se que o compilador otimizou automaticamente os laços dos testes KIJ e IKJ, uma vez que o GCC possui a função de auto-vetorização [1]. Isso se confirma ao analisar a grande queda no número de ciclos por instrução.

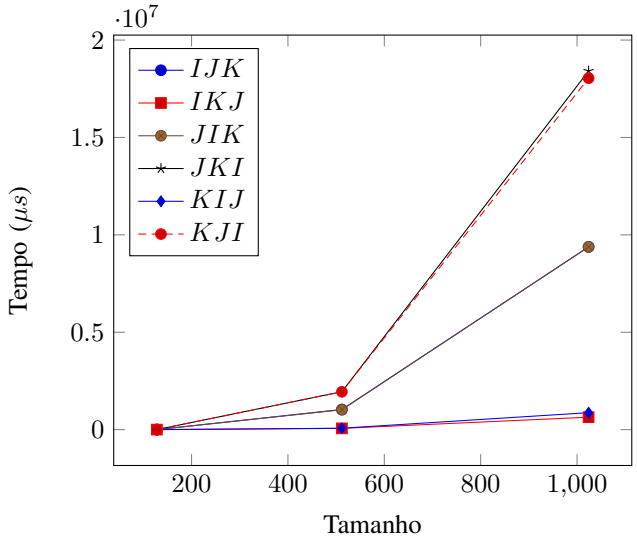
Percebe-se que o número de cache misses do cache L2 não foi menor do que nos outros exemplos, mas existe a possibilidade destes estarem sendo contados excessivamente quando se faz uso de vetorização [2].

Ainda assim, percebe-se que os piores resultados apresentavam um número de cache misses maior em uma ordem de grandeza do que as outras, e neste caso, esse é possivelmente o motivo do maior tempo de execução.

Tabela II: Multiplicação de matrizes trivial comutando hierarquia de laços.

Size	Mode	Tempo(μs)	L2_DCM	MFLOPS	CPI
128	IJK	2296.0	30720.8	2541.476	0.45
128	IKJ	1217.4	52699.4	4644.762	0.43
128	JIK	2283.4	65147.2	2411.176	0.46
128	JKI	2940.2	65014.6	1972.994	0.47
128	KIJ	1143.8	43128.8	4199.338	0.43
128	KJI	2963.8	54645.4	1909.844	0.47
512	IJK	1034952.0	17134397.4	1866.768	3.29
512	IKJ	65857.2	2488558.2	5247.794	0.41
512	JIK	1020162.0	2679476.4	1657.918	3.22
512	JKI	1946646.8	18970861.0	2136.658	4.99
512	KIJ	72242.6	7947405.4	5022.500	0.45
512	KJI	1943300.8	18098968.4	2129.632	4.97
1024	IJK	9379871.2	89582052.6	1647.380	3.79
1024	IKJ	639567.6	27410800.2	4053.790	0.49
1024	JIK	9384713.4	53961401.0	1391.234	3.73
1024	JKI	18413863.2	283493657.0	2064.984	5.84
1024	KIJ	870798.6	77637102.8	3217.820	0.68
1024	KJI	18044042.4	413583035.2	2022.272	5.84

Figura 1: Comparação do tempo de execução entre as diferentes hierarquias.



III. Experimento 2 - Multiplicação com Blocagem

Nesse experimento foi implementado o algoritmo para multiplicação de matrizes com blocagem, para verificar a

diferença no desempenho causada pela mudança do tamanho do bloco para 2, 4, 16 e 64. A implementação foi feita com base na hierarquia de laços IKJ, que obteve os melhores resultados em matrizes maiores.

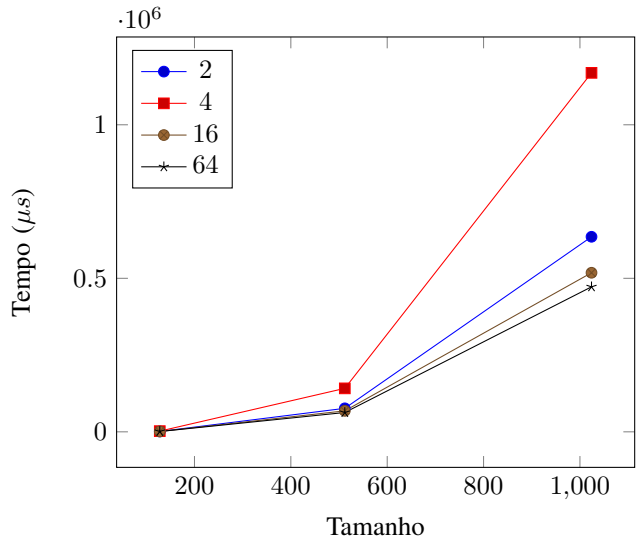
Na tabela III também percebe-se uma forte relação entre os melhores tempos e o número de cache misses. Neste caso, o bloco de tamanho 64x64 se mostrou o mais eficiente em todos os casos, embora se equipare com o segundo colocado, que é o bloco de tamanho 16x16.

Percebe-se nesse experimento que o tempo de execução não foi diretamente influenciado pelo número de cache misses da memória cache L2, e que esses resultados não estão sujeitos à possivel vetorização de operações. Com uma análise extendida seria possível identificar outros fatores, como a variação no número de cache misses da memória cache L3.

Tabela III: Multiplicação de matrizes IKJ com blocagem, variando tamanho dos blocos.

Size	Block	Tempo(μs)	L2_DCM	MFLOPS	CPI
128	2	1277.4	21547.0	3442.836	0.41
128	4	2272.6	10016.2	1916.454	0.54
128	16	1088.0	12574.6	4083.400	0.37
128	64	1073.4	6076.4	4890.978	0.37
512	2	76549.2	723738.2	3677.434	0.41
512	4	141571.4	360047.6	1981.468	0.53
512	16	68315.4	476030.0	4203.178	0.37
512	64	62944.0	549661.4	5665.736	0.35
1024	2	635263.8	5170774.2	3533.186	0.42
1024	4	1169158.4	1593589.4	1920.372	0.54
1024	16	518078.8	2729243.0	4356.296	0.37
1024	64	472215.2	2770415.0	5546.164	0.33

Figura 2: Comparação do tempo de execução entre os diferentes tamanhos de blocos.



IV. Experimento 3 - Multiplicação de Strassen

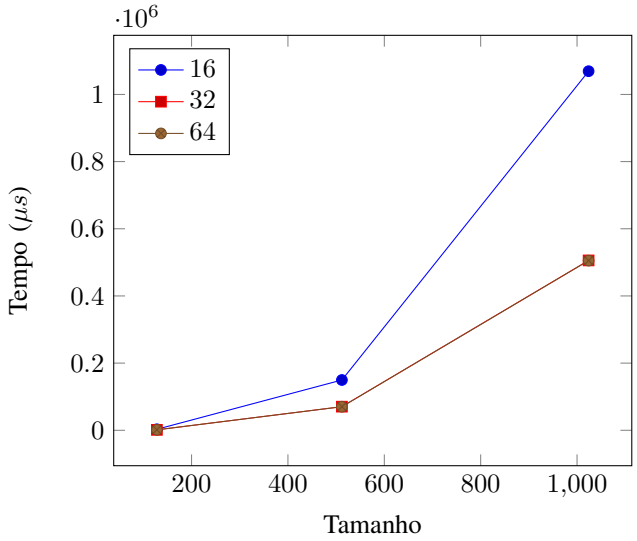
Nesse experimento foi implementado o algoritmo de Strassen, de forma que a matriz é particionada e realocada em matrizes menores utilizando uma técnica de divisão e conquista. Nessa implementação específica o algoritmo possui uma particularidade, pois quando se obtém uma matriz suficientemente pequena, é realizada uma multiplicação trivial IKJ [3].

Neste experimento, variamos o tamanho da matriz do caso base, que corresponde à segunda coluna da Tabela IV. Percebemos então um empate técnico entre a matriz de tamanho 32x32 e a de tamanho 64x64. Dessa vez, não necessariamente os melhores resultados foram os com o menor número de cache misses, uma vez que cada chamada recursiva requer alocação dinâmica de matrizes, que pode provocar tais variações nos resultados.

Tabela IV: Multiplicação de Matrizes de Strassen, variando tamanho da matriz do caso base.

Size	Block	Tempo(μs)	L2_DCM	MFLOPS	CPI
128	16	3003.2	50554.8	1153.724	0.41
128	32	1299.4	33814.8	2833.442	0.40
128	64	1394.2	21002.8	3093.936	0.47
512	16	149722.4	3752694.2	1172.518	0.43
512	32	70324.6	2768805.0	2642.726	0.42
512	64	70155.8	2113807.8	3082.758	0.48
1024	16	1069186.4	28045284.6	1158.834	0.43
1024	32	505758.0	20928380.6	2590.934	0.44
1024	64	505214.2	16202391.8	3014.232	0.49

Figura 3: Comparação do tempo de execução entre os diferentes tamanhos minimos.



V. Experimento 4 - BLAS

Nesse experimento foi utilizada a função cblas_dgemm do BLAS para realizar a multiplicação de duas matrizes, e obtemos assim os resultados da Tabela V.

Tabela V: Multiplicação de matrizes da biblioteca BLAS

Size	Tempo(μs)	L2_DCM	MFLOPS	CPI
128	587.0	5632.0	6487.864	0.42
512	23404.0	328827.8	11351.898	0.36
1024	181309.8	830260.2	11711.924	0.36

Comparando os melhores resultados dos 4 experimentos (considerando os testes com matrizes de tamanho 1024x1024), fazemos então a comparação da Tabela VI, onde percebemos

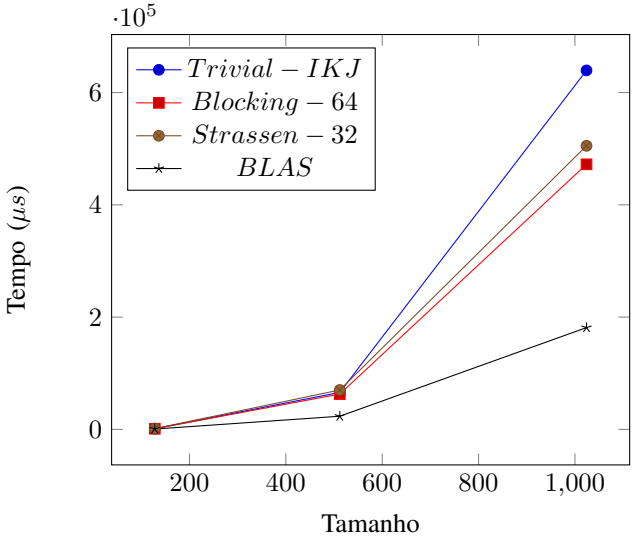
a clara superioridade do BLAS/ATLAS sobre todas as outras tentativas, tal fato fica ainda mais evidente na Figura 4.

Essa diferença de desempenho era esperada, uma vez que a BLAS é implementada com um alto nível de otimização, que além da afinidade de cache, utiliza recursos do processador como registradores vetoriais ou instruções SIMD [4].

Tabela VI: Comparação dos melhores resultados em multiplicação de matrizes de tamanho 1024x1024

Algorithm	Tempo(μs)	L2_DCM	MFLOPS	CPI
Trivial	639567.6	27410800.2	4053.790	0.49
Blocking	472215.2	2770415.0	5546.164	0.33
Strassen	505214.2	16202391.8	3014.232	0.49
BLAS	181309.8	830260.2	11711.924	0.36

Figura 4: Comparação entre os melhores resultados dos quatro experimentos com relação ao tempo.



VI. Experimento 5 - Fusão de Laços

Nesse experimento foi feita a comparação do desempenho da técnica de fusão de laços, realizando operações sobre dados de um vetor de 1000000 de elementos. Na Tabela VII, a primeira linha representa o algoritmo sem fusão de laços, e a segunda, o algoritmo com fusão de laços.

Percebe-se então que o resultado permanece muito semelhante, e apesar de apresentar um maior número de cache misses, a operação sem fusão de laços apresentou-se ligeiramente mais eficiente.

Tabela VII: Técnica de fusão de laços.

Size	Mode	Tempo(μs)	L2_DCM	MFLOPS	CPI
1000000	Sem Fusao	131142.8	17302.2	519.388	2.96
1000000	Com fusao	131720.0	11017.8	469.994	2.97

VII. Experimento 6 - Estruturas de dados

Nesse experimento foi feita a comparação do desempenho da técnica de fusão de laços trabalhando com diferentes estruturas de dados. Na tabela VIII, a primeira linha representa o formato **double abc[?][3]**, a segunda linha o formato **double abc[3][?]**, e a terceira um array do tipo de dados **struct {double a, b, c; } est_abc** . Nesse experimento percebemos que embora possua o menor número de cache misses, o terceiro modo não é o mais rápido, perdendo por pouco do primeiro modo, muito embora ambos sejam virtualmente equivalentes, pois em ambos os casos, os valores a, b e c estão contíguos na memória.

Tabela VIII: Desempenho obtido no exp 5

Size	Mode	Tempo(μs)	L2_DCM	MFLOPS	CPI
1000000	abc[?][3]	130842.0	32251.4	456.742	2.96
1000000	abc[3][?]	133867.2	10534.0	444.776	3.09
1000000	struct	132623.2	1782.8	459.150	2.96

VIII. Conclusão

Os experimentos realizados demonstraram a vantagem computacional obtida por meio da utilização de técnicas que favorecem a afinidade de memória, adequando-se a hierarquia dos laços, e também o impacto causado por diferentes estruturas de dados, de forma que se faça o uso mais eficiente do processador, evitando que este permaneça ocioso.

Porém, isso não é o suficiente para se obter um desempenho ótimo do algoritmo, uma vez que os resultados variam de acordo com a complexidade dos problemas, e uma vez que desconhecemos todos os recursos adicionais presentes na arquitetura do processador, que o BLAS certamente faz uso. Isso também evidencia a superioridade e a importância da utilização de bibliotecas otimizadas de operações de álgebra linear na computação científica.

Também notamos que a utilização de fusão de laços não necessariamente trará benefícios relevantes, pois o desempenho depende da natureza dos dados e também das operações realizadas à cada iteração. Já a utilização de diferentes estruturas de dados poderia apresentar diferenças mais expressivas em casos mais complexos, uma vez que não notamos uma diferença relevante nos experimentos simples realizados.

Finalmente, a utilização das flags de otimização pode prejudicar a avaliação da diferença real do desempenho causada por cache misses, uma vez que recursos do processador, como as operações vetoriais, possam ser ativados automaticamente.

Referências

[1] "Auto-vectorization with gcc 4.7," http://locklessinc.com/articles/vectorize/.

[2] PAPIDocs, "Papitopics:sandyflops," https://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:SandyFlops#PAPI_Preset_Definitions.

[3] M. Thoma, "The strassen algorithm in python, java and c++," https://martin-thoma.com/strassen-algorithm-in-python-java-cpp/, 01 2013.

[4] Wikipedia, "Basic linear algebra subprograms," https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms".