Programação de Alto Desempenho

Atividade 3 - Otimização por vetorização

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 vetorização de laços, que pode ser realizada de forma automática pelo computador, mas também utilizando-se funções intrínsecas das arquiteturas sse, avx e avx2. 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 as flags "-O3 -msse -ftree-vectorize -fopt-info-vec-all", utilizando a biblioteca PAPI para estimar o tempo total de processamento, o total de operações de ponto flutuante (PAPI_SP_OPS), e o fator de Ciclos por elementos do vetor (CPE).

As especificações da máquina utilizada estão disponíveis na Tabela I.

Tabela I: Especificações da Máquina

PAPI Version	5.4.3.0
Model string and code	Intel Core i5-2400
CPU Revision	7.000000
CPU Max Megahertz	3101
CPU Min Megahertz	1600
Threads per core	1
Cores per Socket	4
Number Hardware Counters	11
Max Multiplex Counters	192
Cache L3	6144 KB
Cache L2	256 KB * 4
RAM	4 Gb
SO	Ubuntu 14.04 x64
Kernel	3.13.0-46-generic
GCC	6.1.1 20160511

II. Experimento 1

Nesse experimento foram feitos diversos testes envolvendo 6 diferentes tipos de cálculos, sendo avaliada a capacidade do compilador em vetorizá-los automaticamente, sendo feitas também as adaptações possíveis e necessárias para que isso seja possível, sendo também implementadas versões vetorizadas fazendo-se uso de funções intrínsecas do padrão avx.

A. Algoritmo 1

O primeiro algoritmo consiste no seguinte cálculo:

```
for (i=1; i<N; i++) {
  x[i] = y[i] + z[i];
  a[i] = x[i-1] + 1.0;
}
```

Ao compilar este exemplo com a flag "-fopt-info-vec-all", pudemos observar a seguinte mensagem: "src/exercicio_a.c:25:3: note: LOOP VECTORIZED", que coincide exatamente com o laço descrito acima.

Assim, foi feita uma segunda versão, agora vetorizada por funções instrínsecas desse mesmo algoritmo:

```
ones = _mm256_set1_ps(1.0);
for( i = 1; i < 8; i++ ) {
  x[ i ] = y[ i ] + z[ i ];
  a[ i ] = x[ i - 1 ] + 1.0;
}
for( i = 8; i < N - 8; i += 8 ) {
  v1 = _mm256_load_ps( y + i );
  v2 = _mm256_load_ps( z + i );
  v3 = _mm256_add_ps( v1, v2 );
  _mm256_store_ps( x + i, v3 );

  v1 = _mm256_load_ps( x + i -1 );
  v2 = _mm256_add_ps( v1, ones );
  _mm256_store_ps( a + i, v2 );
}
for( ; i < N; i++ ) {
  x[ i ] = y[ i ] + z[ i ];
  a[ i ] = x[ i - 1 ] + 1.0;
}
```

A corretude do algoritmo é facilmente observável pelas operações aritméticas realizadas, e também pelos resultados obtidos ao final da execução dos dois programas. Em um teste padronizado, em que os vetores são inicializados com os mesmos valores pseudo-aleatórios, temos o seguinte resultado nas dez ultimas posições do vetor a:

Versao original	173.41	195.57	75.57	156.35	283.08
	22.80	357.32	338.45	254.81	265.32
Versao vetorizada	173.41	195.57	75.57	156.35	283.08
	22.80	357.32	338.45	254.81	265.32

A comparação do desempenho se dá então pela Tabela II

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

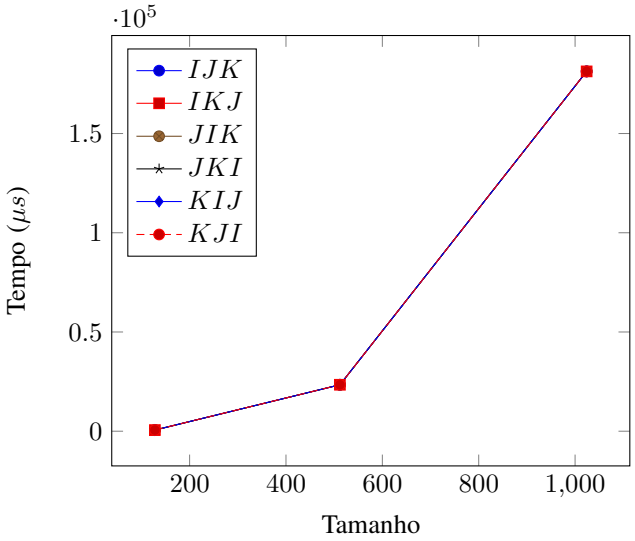
Size	Mode	Tempo(μs)	L2_DCM	MFLOPS	CPI

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

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



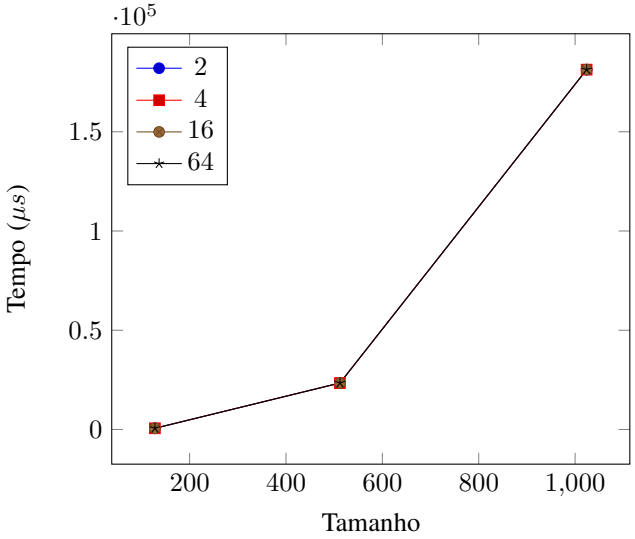
bloco de tamanho 64x64 se mostrou o mais eficiente em todos os casos, embora se equipare com o segundo colocado, que é o bloco de tamanho 16x16.

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

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



IV. Experimento 3 - Multiplicação de Strassen

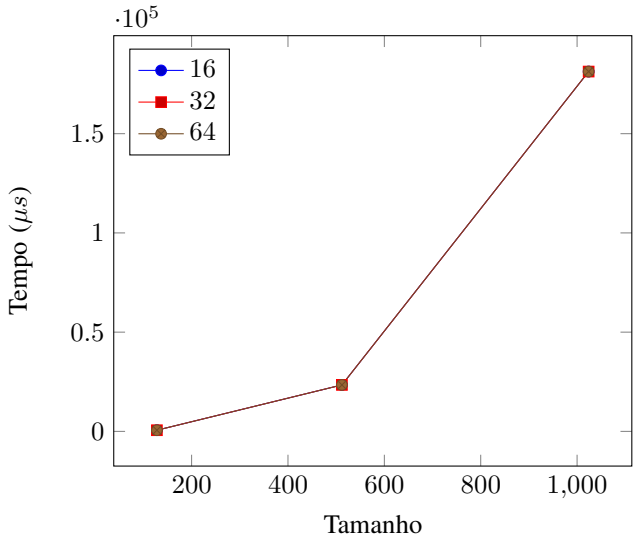
Nesse experimento foi implementado o algoritmo de Strassen, de forma que a matriz é particionada e realocada em matrizes menores utilizando uma técnica de divisão e conquista. Nessa implementação específica o algoritmo possui uma particularidade, pois quando se obtém uma matriz suficientemente pequena, é realizada uma multiplicação trivial IKJ [1].

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

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

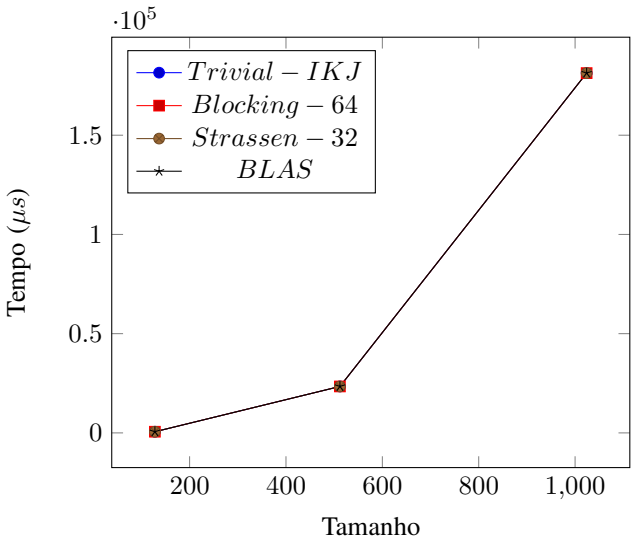
Comparando os melhores resultados dos 4 experimentos (considerando os testes com matrizes de tamanho 1024x1024), fazemos então a comparação da Tabela VI, onde percebemos a clara superioridade do BLAS/ATLAS sobre todas as outras tentativas, tal fato fica ainda mais evidente na Figura 4.

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 [2].

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

Algorithm	Tempo(μs)	L2_DCM	MFLOPS	CPI

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

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

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

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