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. Em todos os experimentos, foram utilizados vetores de tamanho 50000000, iniciados de forma pseudo-aleatoria, com seed fixa de valor 424242.

A. Algoritmo A

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: Análise de desempenho do experimento 1 a

Mode	**Tempo(μs)**	**L2_DCM**	**MFLOPS**	**CPE**
Exercicio A - Padrão	130415	7895059	795.2404	8.76
Exercicio A - AVX	129726	8045165	1635.3426	8.72

B. Algoritmo B

O algoritmo consiste no seguinte cálculo:

```
for (i=0; i<N-1; 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" não pudemos observar o texto: "note: LOOP VECTORIZED", e por isso, foi feita uma pequena alteração para habilitar a vetorização automática desse laço.

```
for( i = 0; i < N - 1; i++ ) {
  a[ i ] = x[ i + 1 ] + 1.0;
  x[ i ] = y[ i ] + z[ i ];
}
```

Ao compilar este exemplo com a flag "-fopt-info-vec-all", pudemos observar a seguinte mensagem: "src/exercicio_b.c:25:3: note: LOOP VECTORIZED".

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

```
ones = _mm256_set1_ps( 1.0 );
for( i = 0; i < N - 9; i += 8 ) {
  v1 = _mm256_load_ps( x + i + 1 );
  v2 = _mm256_add_ps( v1, ones );
  _mm256_store_ps( a + i, v2 );
  v1 = _mm256_load_ps( y + i );
  v2 = _mm256_load_ps( z + i );
  v2 = _mm256_add_ps( v1, v2 );
  _mm256_store_ps( x + i, v2 );
}
for( ; i < N - 1; i++ ) {
  a[ i ] = x[ i + 1 ] + 1.0;
  x[ i ] = y[ i ] + z[ i ];
}
```

A corretude do algoritmo é garantida pois a inversão da relação de antidependência não altera o resultado, e em contrapartida permite que o algoritmo seja paralelizado automaticamente pelo compilador. A mesma lógica foi utilizada na versão com paralelização intrínseca. Em testes padronizados, em que os vetores são inicializados com os mesmos valores pseudo-aleatórios, temos na Tabela nas dez ultimas posições do vetor a, e em seguida o vetor x:

Versao original	203.17	111.63	76.05	171.88	48.31
	73.78	19.93	169.58	129.03	23.91
Versao vetorizada	203.17	111.63	76.05	171.88	48.31
	73.78	19.93	169.58	129.03	23.91
Versao intrinseca	203.17	111.63	76.05	171.88	48.31
	73.78	19.93	169.58	129.03	23.91

Versao original	194.57	74.57	155.35	282.08	21.80
	356.32	337.45	253.81	264.32	128.03
Versao vetorizada	194.57	74.57	155.35	282.08	21.80
	356.32	337.45	253.81	264.32	128.03
Versão Intrinseca	194.57	74.57	155.35	282.08	21.80
	356.32	337.45	253.81	264.32	128.03

A comparação do desempenho se dá então pela Tabela III

Tabela III: Análise de desempenho do experimento 1 b

Mode	**Tempo(μs)**	**L2_DCM**	**MFLOPS**	**CPE**
Exercicio B - Padrão	130128	8327909	849.7111	8.75
Exercicio B - Modificado	124571	10141674	1168.2948	8.36
Exercicio B - AVX	122822	10077226	2184.5370	8.26

C. Algoritmo C

O algoritmo consiste no seguinte cálculo:

```
for( i = 0; i < N - 1; i++ ) {
  x[ i ] = a[ i ] + z[ i ];
  a[ i ] = x[ i + 1 ] + 1.0;
}
```

Ao compilar este exemplo com a flag "-fopt-info-vec-all" não pudemos observar o texto: "note: LOOP VECTORIZED", e por isso, foi feita uma pequena alteração para habilitar a vetorização automática desse laço, neste caso, a falsa relação de dependência foi corrigida a partir da utilização de uma variável auxiliar.

```
for( i = 0; i < N - 1; i++ ) {
  aux = a[ i ] + z[ i ];
  a[ i ] = x[ i + 1 ] + 1.0;
  x[ i ] = aux;
}
```

Ao compilar este exemplo com a flag "-fopt-info-vec-all", pudemos observar a seguinte mensagem: "src/exercicio_c.c:22:3: note: LOOP VECTORIZED".

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

```
ones = _mm256_set1_ps( 1.0 );
for( i = 0; i < N - 9; i += 8 ) {
  v1 = _mm256_load_ps( a + i );
  v2 = _mm256_load_ps( z + i );
  v3 = _mm256_add_ps( v1, v2 ); //aux

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

A corretude do algoritmo é garantida e não está sujeita à variações nos resultados. Em testes padronizados, em que os vetores são inicializados com os mesmos valores pseudo-aleatórios, temos na Tabela nas dez ultimas posições do vetor a, e em seguida o vetor x:

Versao original	208.92	209.19	197.92	79.58	107.30
	189.40	54.15	71.08	49.35	99.84
Versao vetorizada	208.92	209.19	197.92	79.58	107.30
	189.40	54.15	71.08	49.35	99.84
Versao intrinseca	208.92	209.19	197.92	79.58	107.30
	189.40	54.15	71.08	49.35	99.84

Versao original	48.50	101.70	75.22	307.19	370.98
	259.06	263.70	259.64	158.99	48.35
Versao vetorizada	48.50	101.70	75.22	307.19	370.98
	259.06	263.70	259.64	158.99	48.35
Versão Intrinseca	48.50	101.70	75.22	307.19	370.98
	259.06	263.70	259.64	158.99	48.35

A comparação do desempenho se dá então pela Tabela IV

Tabela IV: Análise de desempenho do experimento 1 c

Mode	Tempo(μs)	L2_DCM	MFLOPS	CPE
Exercicio C - Padrão	105713	6373605	1116.5664	7.11
Exercicio C - Modificado	105877	8514220	1169.0563	7.09
Exercicio C - AVX	104160	8443770	3211.2259	7.00

D. Algoritmo D

O primeiro algoritmo consiste no seguinte cálculo:

```
for( i = 0; i < N; i++ ) {
  t = y[ i ] + z[ i ];
  a[ i ] = t + 1.0 / t;
}
```

Ao compilar este exemplo com a flag "-fopt-info-vec-all", pudemos observar a seguinte mensagem: "src/exercicio_d.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 = 0; i < N - 8; i += 8 ) {
  v1 = _mm256_load_ps( y + i );
  v2 = _mm256_load_ps( z + i );
  v1 = _mm256_add_ps( v1, v2 );
  v2 = _mm256_div_ps( ones, v1 );
  v1 = _mm256_add_ps( v1, v2 );
  _mm256_store_ps( a + i, v1 );
}
for( ; i < N; i++ ) {
  t = y[ i ] + z[ i ];
  a[ i ] = t + 1.0 / t;
}
```

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	194.58	74.58	155.35	282.08	21.85
	356.33	337.45	253.82	264.33	286.58
Versao vetorizada	194.58	74.58	155.35	282.08	21.85
	356.33	337.45	253.82	264.33	286.58

A comparação do desempenho se dá então pela Tabela V.

Tabela V: Análise de desempenho do experimento 1 d

Mode	Tempo(μs)	L2_DCM	MFLOPS	CPE
Exercicio D - Padrão	167968	2304766	902.1654	11.30
Exercicio D - AVX	82375	7297300	5501.4797	5.54

E. Algoritmo e

O primeiro algoritmo consiste no seguinte cálculo:

```
for( i = 0; i < N; i++ ) {
  s += z[ i ];
}
```

Embora seja um exemplo simples, não pudemos observar a vetorização automática de uma operação de redução, e a única opção seria habilitar outras funções do compilador como a flag "fast-math", que resultou numa variação muito grande dos resultados, e por esse motivo não foi adotada.

Assim, foi feita uma segunda versão, agora vetorizada por funções instrínsecas desse mesmo algoritmo, onde os dados são armazenados em um acumulador, e somados ao final.

```
acc = _mm256_xor_ps( acc, acc );
for( i = 0; i < N - 8; i += 8 ) {
  data = _mm256_load_ps( z + i );
  acc = _mm256_add_ps( acc, data );
}
for( ; i < N; i++ ) {
  s += z[ i ];
}
for( i = 0; i < 8; i++ ) {
  s += acc[ i ];
}
```

A corretude do algoritmo é facilmente observável pelas operações aritméticas realizadas, mas é esperada uma variação nos resultados pela mudança na ordem em que são realizadas as operações de ponto flutuante. Em um teste padronizado, em que os vetores são inicializados com os mesmos valores pseudo-aleatórios, temos no primeiro algoritmo o valor de $s = 535541824.0$ e no segundo, o valor de $s = 535541856.0$. Trata-se de uma diferença grande mas ainda aceitável se comparada à outras alternativas.

A comparação do desempenho se dá então pela Tabela VI.

Tabela VI: Análise de desempenho do experimento 1 e

Mode	Tempo(μs)	L2_DCM	MFLOPS	CPE
Exercicio E - Padrão	21244	3104971	2456.8648	1.42
Exercicio E - AVX	21106	3120171	2474.7116	1.42

F. Algoritmo f

O primeiro algoritmo consiste no seguinte cálculo:

```
for( i = 1; i < N; i++ ) {
  a[ i ] = a[ i - 1 ] + b[ i ];
}
```

Diferente dos exercícios anteriores, este algoritmo possui uma dependência real e intratável. A análise do desempenho desse algoritmo se dá então pela Tabela VII.

Tabela VII: Análise de desempenho do experimento 1 f

Mode	Tempo(μs)	L2_DCM	MFLOPS	CPE
Exercicio F - Padrão	66773	3593025	753.7678	4.48

III. Experimento 2- Mínimos Quadrados

Nesse experimento foram feitos diversos testes envolvendo a vetorização do método dos mínimos quadrados. Normalmente calculado da seguinte forma:

```
SUMx = 0; SUMy = 0; SUMxy = 0; SUMxx = 0;
for( i = 0; i < n; i++ ) {
  SUMx += x[ i ];
  SUMy += y[ i ];
  SUMxy += x[ i ] * y[ i ];
  SUMxx += x[ i ] * x[ i ];
}
slope = ( SUMx * SUMy - n * SUMxy )
        / ( SUMx * SUMx - n * SUMxx );
y_intercept = ( SUMy - slope * SUMx ) / n;
```

Neste exemplo, a vetorização automática não foi realizada, mais uma vez, por existirem operações de redução. E assim, foi implementada uma versão vetorizada com funções intrínsecas:

```
  for( i = 0; i < n - 4; i += 4 ) {
    data_x = _mm256_load_pd( x + i );
    acc_x = _mm256_add_pd( acc_x, data_x );
    // SUMx += x[ i ];
    data_y = _mm256_load_pd( y + i );
    acc_y = _mm256_add_pd( acc_y, data_y );
    // SUMy += y[ i ];
    mult_xy = _mm256_mul_pd( data_x, data_y );
    acc_xy = _mm256_add_pd( acc_xy, mult_xy );
    // SUMxy += x[ i ] * y[ i ];
    mult_xx = _mm256_mul_pd( data_x, data_x );
    acc_xx = _mm256_add_pd( acc_xx, mult_xx );
    // SUMxx += x[ i ] * x[ i ];
  }
  SUMx = 0; SUMy = 0; SUMxy = 0; SUMxx = 0;
  for( ; i < n; i++ ) {
    SUMx += x[ i ];
    SUMy += y[ i ];
    SUMxy += x[ i ] * y[ i ];
    SUMxx += x[ i ] * x[ i ];
  }
  for( i = 0; i < 4; i++ ) {
    SUMx += acc_x[ i ];
    SUMy += acc_y[ i ];
    SUMxy += acc_xy[ i ];
    SUMxx += acc_xx[ i ];
  }
  slope = ( SUMx * SUMy - n * SUMxy )
          / ( SUMx * SUMx - n * SUMxx );
  y_intercept = ( SUMy - slope * SUMx ) / n;
```

Para um pequeno conjunto de dados, os resultados obtidos coincidiram com os resultados esperados, assim a dados como entrada os vetores $A = (2, 4, 6, 8)$, e $B = (2, 11, 28, 40)$, temos como resultado $y = 6.55x - 12.50$. Em um teste com vetores pseudo-aleatorios de tamanho 100000000, os resultados também coincidiram, sendo o valor de $y = 2.50x - 134.22$.

Finalmente, apresentamos na Tabela VIII a comparação do desempenho dos exemplos.

Tabela VIII: Análise de desempenho do experimento 1 f

Mode	**Tempo(μs)**	**L2_DCM**	**MFLOPS**	**CPE**
Least Squares - Simples	180369	21317025	3819.8092	6.07
Least Squares - AVX	178567	21990741	4254.4977	6.01

IV. Conclusão

Analisando os resultados obtidos em todos os testes, podemos chegar à algumas conclusões em comum. Percebe-se que em todos os casos, melhora do desempenho ocorreu com a vetorização, porém não de forma expressiva, ainda que em muitos casos os dados sejam do tipo float, quando ocorrem 8 operações em paralelo. Percebemos também que pequenas alterações no código podem favorecer a vetorização automática, oferecendo melhor porabilidade e manutenção do código, sem perda considerável de desempenho se comparado aos códigos que fazem uso de funções intrínsecas.

Finalmente, a utilização de vetorização para operações de redução é viável e pode trazer uma melhora no desempenho, porém deve ser realizada com cuidado, pois provavelmente alterará os resultados obtidos de forma considerável, principalmente se habilitadas otimizações menos confiáveis como a flag 'fast-math' do gcc.