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: [1]

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.

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

Referências

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