Análise de Desempenho em Compilações
Instruções padrões e SIMD

Introdução

A compilação em algoritmos escritos em *C++* pode ser realizada de diversas formas, algumas delas com o intuito de se otimizar o código escrito, assim ganhando performance. Neste documento é feita uma comparação entre 3 tipos distintos de compilação de um código escrito em *C++*:

- Compilação sem otimizações
- Compilação com otimização da arquitetura AVX, com uso da flag "*-mavx*"
- Compilação com otimização da arquitetura AVX2, com uso da flag "*-mavx2*"

Assim, busca-se analisar as diferenças nos tempos de execução de cada versão, comparando os ganhos em performance das otimizações feitas com a compilação sem otimizações.

O Código

O código a ser compilado tem a função de informar tempos variados de execução de acordo com alguns parâmetros. No caso deste estudo, o tamanho de um vetor é variado até um limite "*max*", e para cada tamanho é medido um tempo de execução em uma função. O tamanho do vetor aumenta com um passo "*plus*". Assim, em um total de 6 funções diferentes, são calculados os tempos de execução para tamanhos de vetor variados.

O usuário deste algoritmo deve compilar o programa de 3 modos diferentes, para realização dos testes:

- "g++ func.cpp -o <nome> <size> <max> <plus>" : compilação sem otimização
- "g++ -O2 -ffast-math -ftree-vectorize -mavx func.cpp -o <nome> <size> <max> <plus>": compilação com arquitetura *AVX*
- "g++ -O2 -ffast-math -ftree-vectorize -mavx2 func.cpp -o <nome> <size> <max> <plus>": compilação com arquitetura *AVX* 2

Entende-se por <nome> o nome do arquivo compilado e <size> o tamanho inicial do vetor, que será usado nas funções.

Para os testes realizados neste relatório, os parâmetros foram: <size> foi inicializado com o valor de "*10000000*" (dez milhões), <max> com o valor de "*100000000*" (cem milhões), e <plus> com o valor de "*10000000*" (dez milhões). Portanto, para cada função serão medidos 10 tempos, para cada compilação. Não é recomendado um valor inicial em uma escala abaixo de 6 casas decimais, pois pode prejudicar a visualização dos resultados (empírico).

Para cada compilação, o programa irá gerar um arquivo em texto, chamado "*resultados.txt*". Este arquivo informa o tempo de execução de cada função, para cada tamanho de vetor. Para melhor visualização destes resultados, o arquivo em questão pode ser importado no *Excel*, com o separador de números sendo ',' (vírgula).

Resultados

Dado os resultados pelo arquivo *"resultados.txt",* pode-se plotar diversos gráficos comparativos entre os tempos obtidos. O intuito é comparar os tempos de execução da compilação sem otimização com os tempos otimizados pela arquitetura *AVX*. Para isso, o seguinte método foi utilizado:

1. Plotar o gráfico de cada função para cada compilação
2. Subtrair o tempo de compilação sem otimização dos tempos de compilação otimizados (*AVX e AVX2)*, para cada tamanho de vetor de cada função
3. Obter a porcentagem de otimização do tempo a partir das subtrações acima

Sabe-se que os dados que estão sendo trabalhados nos vetores são do tipo "double", portanto é de se esperar que as compilações otimizadas consigam, aproximadamente, diminuir o tempo em 2 vezes, já que os registradores utilizados são do tipo "*%xmm#"* (128 bits), e inteiros possuem tamanho de 64 bits. Os resultados obtidos no primeiro teste foram:

inner_product				
Size of Vector	CompNorm - CompMavx(s)	CompNorm-CompMavx2(s)	X times faster (CompMavx)	X times faster(CompMavx2)
10000000	3.06E-02	3.06E-02	25491.33	23530.46
20000000	6.15E-02	6.15E-02	47320.00	47320.00
30000000	9.59E-02	9.59E-02	106607.11	73804.92
40000000	1.30E-01	1.30E-01	76478.24	100010.00
50000000	1.60E-01	1.60E-01	178043.33	123260.77
60000000	2.01E-01	2.01E-01	154371.54	91219.55
70000000	2.31E-01	2.31E-01	177713.85	177713.85
80000000	2.55E-01	2.55E-01	195797.69	195797.69
90000000	3.24E-01	3.24E-01	190760.00	270243.33
100000000	3.10E-01	3.10E-01	238515.38	238515.38
sum_positive				
Size of Vector	CompNorm - CompMavx(s)	CompNorm-CompMavx2(s)	X times faster (CompMavx)	X times faster(CompMavx2)
10000000	8.74E-02	8.74E-02	67204.85	51391.94
20000000	1.75E-01	1.75E-01	145829.17	134611.54
30000000	2.71E-01	2.71E-01	208626.15	159537.65
40000000	3.73E-01	3.73E-01	287296.92	287296.92
50000000	4.64E-01	4.64E-01	356994.62	356994.62
60000000	5.36E-01	5.36E-01	412265.38	412265.38
70000000	6.44E-01	6.44E-01	495110.00	495110.00
80000000	7.23E-01	7.23E-01	903481.25	425167.65
90000000	7.99E-01	7.99E-01	888008.89	614775.38
100000000	9.33E-01	9.33E-01	1036582.22	518291.11

sqrt_element				
Size of Vector	CompNorm - CompMavx(s)	CompNorm-CompMavx2(s)	X times faster (CompMavx)	X times faster(CompMavx2)
10000000	0.1078883	0.1071437	6.39	6.16
20000000	0.1879951	0.2021158	4.48	6.06
30000000	0.3108484	0.31554	5.81	6.27
40000000	0.4049777	0.4111945	5.63	6.06
50000000	0.501622	0.505703	5.73	5.96
60000000	0.607946	0.603785	6.12	5.91
70000000	0.714816	0.727139	5.05	5.43
80000000	0.80214	0.801791	5.54	5.53
90000000	0.912406	0.946341	4.71	5.46
100000000	1.014742	1.003592	5.20	4.97
exp_element				
Size of Vector	CompNorm - CompMavx(s)	CompNorm-CompMavx2(s)	X times faster (CompMavx)	X times faster(CompMavx2)
10000000	0.1457269	0.165884	4.01	6.86
20000000	0.267675	0.327495	3.52	8.07
30000000	0.426031	0.5062565	3.74	7.72
40000000	0.626704	0.532903	4.78	3.05
50000000	0.830694	0.976297	3.60	6.63
60000000	0.66342	0.936481	2.39	5.56
70000000	1.039662	1.184819	4.49	8.76
80000000	1.149167	1.365199	3.92	8.72
90000000	1.199621	1.431229	3.43	6.46
100000000	1.14275	1.525795	2.55	5.29

log_element				
Size of Vector	CompNorm - CompMavx(s)	CompNorm-CompMavx2(s)	X times faster (CompMavx)	X times faster(CompMavx2)
10000000	0.017678	0.096809	1.06	1.41
20000000	0.083817	0.272687	1.15	1.71
30000000	-0.134586	0.259039	0.85	1.52
40000000	0.06147	0.440176	1.06	1.70
50000000	0.07025	0.616898	1.05	1.78
60000000	0.04339	0.49595	1.03	1.49
70000000	0.16356	0.35784	1.09	1.23
80000000	0.23501	0.67671	1.12	1.44
90000000	-0.46744	0.42936	0.83	1.23
100000000	0.31511	1.39665	1.12	1.89
gauss				
Size of Vector	CompNorm - CompMavx(s)	CompNorm-CompMavx2(s)	X times faster (CompMavx)	X times faster(CompMavx2)
10000000	0.050768	0.050768	1.28	1.62
20000000	0.037208	0.037208	1.09	1.63
30000000	0.311119	0.311119	1.55	2.07
40000000	0.507684	0.507684	1.87	1.97
50000000	0.935328	0.935328	2.34	2.20
60000000	0.49622	0.49622	1.58	1.56
70000000	0.56254	0.56254	1.55	1.60
80000000	0.62875	0.62875	1.52	1.27
90000000	0.69604	0.69604	1.52	0.97
100000000	1.85352	1.85352	2.26	1.78

Em ordem:

- 1ª coluna: Tamanho do Vetor
- 2ª coluna: Diferença de tempo entre compilação normal e compilação com arquitetura AVX
- 3ª coluna: Diferença de tempo entre compilação normal e compilação com arquitetura AVX2
- 4ª coluna: Quantidade de vezes que a compilação com arquitetura AVX foi mais rápida que a compilação sem otimização
- 5ª coluna: Quantidade de vezes que a compilação com arquitetura AVX2 foi mais rápida que a compilação sem otimização

Discussão

A melhora de desempenho depende do tamanho do vetor e do tipo de operação que está sendo efetuada (função).

Nos testes, pode-se perceber um ganho de desempenho nas funções em geral, com algumas exceções onde o tempo de execução piorou com as

otimizações dos compiladores (“*log_element”*). A função “*sum_positive”* e *“inner_product”* mostraram ganhos em performance extremamente altos, enquanto o desempenho das funções “*log_element*” ***e*** *“*gauss*”* dificilmente chegaram à melhora de velocidade esperada (2 vezes).

Assim, nota-se que o ganho de desempenho, em geral, bate com a teoria, onde se esperava uma melhora de aproximadamente 2 vezes ou mais em relação à compilação sem otimizações, dependendo da função. Além disso, observa-se uma melhora dos tempos medidos a partir do tamanho de vetores igual a *“10000000”* (10 milhões). As funções que melhor desempenharam com as otimizações, em geral, foram:

- ***“sum_positive”***
- ***“inner_product”***

Portanto, as operações de *soma* e *multiplicação* se mostraram mais rápidas.

Imagens

```
void intRandom(double *v, unsigned long n){
  // definir range de numeros randomicos
  std::default_random_engine generator;
  std::uniform_int_distribution<int> distribution(-10,10);
  // gerar numeros e guardar no array
  for(int i = 0; i < n; i++){
    v[i] = (distribution(generator));
  }
}
```

Função que gera vetores com inteiros randômicos

```
//highOrderFunction1
auto timeCalc_1(auto x(double*,unsigned long),unsigned long size,unsigned long max,unsigned long plus,std::ofstream &file){
    //medir o tempo criando 2 tempos (t1, t2)
    for(int i = size; i<= max; i+= plus){
        auto *a = new double[i];
        //randomização do array de ints
        intRandom(a, i);
        //medida do tempo
        high_resolution_clock::time_point t1 = high_resolution_clock::now();
        //function call do argumento
        x(a,i);
        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        duration<double> time_span = duration_cast<duration<double>>(t2 - t1);
        file << '\n' << i << "," << time_span.count();
    }
}
```

Uma das funções que realiza as "calls" de outras funções.