Universidade Federal de Sergipe

Interface Hardware Software Turma 03

Gabriel Teixeira Silveira
Curso: Ciência da Computação
Matrícula: 202100011987

Relatório de Otimização

Professor:
Calebe Micael de Oliveira Conceição

São Cristóvão
Julho de 2024

Diretivas de Compilação

Nessa atividade, comparamos o tempo de execução de diferentes versões dos algoritmos BubbleSort, MergeSort e Heapsort para entradas de tamanhos variáveis, entre 10^5 e 10^8.

Nessa atividade, utilizei dois computadores para rodar os algoritmos de ordenação. O primeiro foi meu notebook Acer Aspire 5 e o segundo foi meu desktop personalizado. As configurações dessas máquinas são:

1. Acer Aspire 5

- Processador: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80 GHz
- Sistema Operacional: Windows 11 Home 23H2 + Ubuntu 22.04 LTS WSL2
- RAM: 16,0 Gb
- GPU: Intel Iris XE Graphics

2. Desktop

- Processador: AMD Ryzen 5 3600 6-Core Processor @ 3.60 GHz
- Sistema Operacional: Windows 10 Pro 22H2 + Ubuntu 22.04 LTS WSL2
- RAM: 24,0 GB
- GPU: AMD Radeon RX 6650 XT

Primeiramente, rodamos o algoritmo de força bruta BubbleSort, implementado abaixo, com uma entrada n = 100000.

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
void trocar(int32_t* i, int32_t* j) {
    int32_t k = *i;
    *i = *j; *j = k;
}
void ordenar(int32_t* V, uint32_t n) {
    for (uint32_t i = 0; i < n; i++) {
        uint32_t min = i;
        for (uint32_t j = i; j < n; j++) {
            if(V[j] < V[min]) min = j;
            if (i != min) trocar(&V[i], &V[min]);
        }
    }
}
int main() {
    const uint32_t n = 100000;
    int32_t* V = (int32_t*) malloc(n * sizeof(int32_t));
    for (uint32_t i = 0; i < n; i++)
        V[i] = rand() * rand();
    ordenar(V, n);
    printf("min = %i, max = %i\n", V[0], V[n-1]);
    return 0;
}
```

Inicialmente, o tempo encontrado foi de 30.250s no desktop e 36.099s no notebook para ordenar o array de 10^5 inteiros, o tempo foi medido através do programa hyperfine. Para fazer a medição, utilizamos os comandos mostrados abaixo.

```
/IHS/aula-6-otimizacao$ gcc bubblesort.c -o bubblesort.elf
/IHS/aula-6-otimizacao$ hyperfine --warmup 3 ./bubblesort.elf
```

O parâmetro ***--warmup n*** define um número de vezes que o programa vai executar antes das medições de tempo iniciarem, isso permite que o cache do processador receba dados da execução do programa, reduzindo o tempo de execução final.

Um modo simples de reduzir o tempo de execução de programas escritos em C e C++ são através das diretivas de compilação. Ferramentas como o GCC oferecem ao desenvolvedor diretivas que custam tempo de compilação extra, mas reduzem o tempo de execução do código.

Essas diretivas são:

- -O0
  Esse é a compilação padrão, sem nenhuma mudança especial.
- -O1
  O compilador tenta reduzir o tamanho do código e o tempo de execução, sem realizar quaisquer

otimizações que demandem muito tempo de compilação.
Um exemplo de otimização é o -finline-functions-called-once, que incorpora funções estáticas ao corpo das funções em que foram chamadas. Aumentando assim o tamanho do assembly, mas reduzindo o tempo de compilação.

- -O2
  Realiza todas as otimizações do -O1 além de quase todas as otimizações suportadas que não envolvem um compromisso entre espaço e velocidade. Em comparação com a opção -O1, essa opção aumenta tanto o tempo de compilação quanto o desempenho do código gerado.

- -O3
  Realiza todas as otimizações do -O2 e aplica otimizações adicionais de desempenho.
  Um exemplo de otimização que aumenta o tempo de compilação é o -floop-unroll-and-jam, que desenrola o loop externo e funde os múltiplos loops internos resultantes.

- -Ofast
  Realiza todas as otimizações do -O3 e aplica otimizações -ffast-math, que violam os padrões IEEE e ANSI, pois reduzem a permitem erros de arredondamento e não consideram valores indefinidos ou infinitos

- -Os
  Otimiza para tamanho. -Os habilita todas as otimizações de -O2, exceto aquelas que frequentemente aumentam o tamanho do código.

- -Og
  Otimiza a experiência de depuração. -Og é o nível de otimização escolhido para o ciclo padrão de edição-compilação-depuração, oferecendo um nível razoável de otimização enquanto mantém uma compilação rápida e uma boa experiência de depuração. É uma escolha melhor do que -O0 para produzir código depurável, pois algumas passagens do compilador que coletam informações de depuração são desabilitadas em -O0.

- -Oz
  Otimiza agressivamente para tamanho em vez de velocidade. Isso pode aumentar o número de instruções executadas se essas instruções exigirem menos bytes para serem codificadas. -Oz se comporta de maneira semelhante a -Os, incluindo a habilitação da maioria das otimizações de -O2.

Rodando no Notebook		Rodando no Desktop	
Diretiva	Força Bruta (s)	Diretiva	Força Bruta (s)
-O0	36.099 ± 2.859	-O0	30.250 ± 0.302
-O1	17.807 ± 1.383	-O1	15.982 ± 0.079
-O2	16.374 ± 0.130	-O2	15.910 ± 0.076
-O3	11.074 ± 0.534	-O3	11.549 ± 0.053
-Ofast	10.116 ± 0.076	-Ofast	11.816 ± 0.049
-Os	17.635 ± 1.554	-Os	15.822 ± 0.024
-Og	16.332 ± 0.330	-Og	18.231 ± 0.111

Eficiência Algorítmica

Antes

```
void merge(int32_t arr[], int32_t l, int32_t m, uint32_t r) {
    uint32_t i, j, k;
    int32_t n1 = m - l + 1;
    int32_t n2 = r - m;

    int32_t* L = (int32_t*) malloc(n1 * sizeof(int32_t));
    int32_t* R = (int32_t*) malloc(n2 * sizeof(int32_t));

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
```

```
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
    free(L);
    free(R);
}

void mergeSort(int32_t *arr, int32_t l, int32_t r) {
    if ( l < r ) {
        int32_t m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

Depois

Como pudemos ver, incorporei o insertion sort ao mergesort, para ordenar os arrays de menos de 20 posições, com isso, temos menos sobrecarga em nossas chamadas recursivas e conseguimos acelerar consideravelmente o código. Também implementei a função trocar() em assembly para agilizar ainda mais o insertionSort.

(Também implementei as funções less() e lessOrEqual() em assembly, mas elas acabaram sendo mais lentas que o operador < (menor que) da linguagem C)

```
#define trocar(i, j)\
    __asm__(\
        "mov (%0), %%eax;"\
        "mov (%1), %%ecx;"\
        "mov %%eax, (%1);"\
        "mov %%ecx, (%0);"\
        :\
        : "r"(i), "r"(j)\
        : "eax", "ecx", "memory"\
    )
void merge(int32_t *arr, int32_t l, int32_t m, uint32_t r) {
    uint32_t i, j, k;
    int32_t n1 = m - l + 1;
    int32_t n2 = r - m;

    int32_t* L = (int32_t*) malloc(n1 * sizeof(int32_t));
    int32_t* R = (int32_t*) malloc(n2 * sizeof(int32_t));

    for (i = 0; i < n1; i++)
```

```
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
    free(L);
    free(R);
}
void insertionSort(int32_t *E, int32_t first, int32_t last) {
    if ((last - first) < 1) { return; }
    for (int32_t i = (first + 1); i <= last; i++) {
        for (int32_t j = i; j > first; j--) {
            if (E[j] < E[j - 1]) { trocar(&E[j], &E[j - 1]); }
            else { break; }
        }
    }
}
void mergeSort(int32_t *arr, int32_t l, int32_t r) {
    if ( r - l > 20) {
        int32_t m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    } else {
        insertionSort(arr, l, r);
    }
}
```

Análise de desempenho

Rodando no Notebook

Diretiva	Mergesort (s)
-O0	2.945 ± 0.130
-O1	1.477 ± 0.082
-O2	1.433 ± 0.017
-O3	1.449 ± 0.067
-Ofast	1.422 ± 0.022
-Os	1.510 ± 0.033
-Og	1.585 ± 0.081

Rodando no Desktop

Diretiva	Mergesort Otimizado (s)
-O0	2.571 ± 0.154
-O1	1.230 ± 0.028
-O2	1.222 ± 0.020
-O3	1.178 ± 0.012
-Ofast	1.225 ± 0.013
-Os	1.233 ± 0.019
-Og	1.243 ± 0.008

Após otimizar meu código, realizei análises de desempenho em meus algoritmos, para identificar quais linhas de código estavam levando mais tempo de execução.

Para isso, utilizei a ferramenta gprof em meu algoritmo mergesort antes e depois de sua otimização. Para assim identificar os gargalos de meu código.

```
$ gprof -l mergesort.elf
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ps/call  ps/call  name
 20.83      0.40     0.40                             merge (mergesort.c:20 @
136d)
 18.23      0.75     0.35                             merge (mergesort.c:21 @
1399)
 15.10      1.04     0.29                             merge (mergesort.c:24 @
13c7)
 14.06      1.31     0.27                             merge (mergesort.c:19 @
13f7)
  8.33      1.47     0.16                             merge (mergesort.c:14 @
12d2)
  4.69      1.56     0.09                             merge (mergesort.c:13 @
1301)
  4.17      1.64     0.08                             merge (mergesort.c:16 @
1316)
  3.65      1.71     0.07                             merge (mergesort.c:15 @
1348)
  2.60      1.76     0.05                             merge (mergesort.c:25 @
13ef)
  1.56      1.79     0.03                             merge (mergesort.c:22 @
13c1)
```

```
  1.04      1.81     0.02                              merge (mergesort.c:11 @
12b4)
  1.04      1.83     0.02                              merge (mergesort.c:40 @
148b)
  1.04      1.85     0.02                              mergeSort (mergesort.c:50 @
1502)
  0.78      1.86     0.01 19999999   750.00   750.00  mergeSort (mergesort.c:43 @
149a)
  0.52      1.88     0.01                              main (mergesort.c:57 @ 1550)
  0.52      1.89     0.01                              merge (mergesort.c:29 @
143d)
  0.52      1.90     0.01                              mergeSort (mergesort.c:45 @
14be)
  0.26      1.90     0.01                              merge (mergesort.c:34 @
1477)
  0.26      1.91     0.01                              merge (mergesort.c:39 @
147f)
  0.26      1.91     0.01                              merge (mergesort.c:41 @
1497)
  0.26      1.92     0.01                              mergeSort (mergesort.c:47 @
14d7)
  0.26      1.92     0.01                              mergeSort (mergesort.c:48 @
14eb)
  0.00      1.92     0.00  9999999     0.00     0.00  merge (mergesort.c:5 @ 1269)
```

```
$ gprof -l mergesort-otimizado.elf
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 17.32      0.31     0.31                              merge (mergesort-
otimizado.c:30 @ 1399)
 16.20      0.60     0.29                              merge (mergesort-
otimizado.c:29 @ 136d)
 15.92      0.89     0.28                              merge (mergesort-
otimizado.c:33 @ 13c7)
 13.41      1.12     0.24                              merge (mergesort-
otimizado.c:23 @ 12d2)
 11.73      1.33     0.21                              insertionSort (mergesort-
otimizado.c:55 @ 14d7)
  7.82      1.48     0.14                              merge (mergesort-
otimizado.c:28 @ 13f7)
  3.91      1.54     0.07                              merge (mergesort-
otimizado.c:22 @ 1301)
  2.79      1.59     0.05                              merge (mergesort-
otimizado.c:25 @ 1316)
  2.51      1.64     0.04                              insertionSort (mergesort-
otimizado.c:53 @ 1547)
  2.23      1.68     0.04                              merge (mergesort-
otimizado.c:24 @ 1348)
```

```
  1.96      1.72     0.04                             insertionSort (mergesort-
otimizado.c:54 @ 1538)
  1.68      1.75     0.03                             merge (mergesort-
otimizado.c:31 @ 13c1)
  1.40      1.77     0.03                             merge (mergesort-
otimizado.c:34 @ 13ef)
  0.56      1.78     0.01  1048575     9.54     9.54  mergeSort (mergesort-
otimizado.c:60 @ 155c)
  0.28      1.78     0.01                             merge (mergesort-
otimizado.c:44 @ 1447)
  0.28      1.79     0.01                             merge (mergesort-
otimizado.c:45 @ 146f)
  0.00      1.79     0.00   524288     0.00     0.00  insertionSort (mergesort-
otimizado.c:51 @ 149a)
  0.00      1.79     0.00   524287     0.00     0.00  merge (mergesort-
otimizado.c:14 @ 1269)
```

MergeSort Otimizado

Em conclusão, consegui reduzir consideravelmente o tempo de execução do Mergesort, tornando ele mais próximo de sua complexidade teórica de O(n logn), ainda haviam otimizações que poderiam ser feitas, como por exemplo alterar o algoritmo de merge utilizado. Esse exercício foi uma ótima oportunidade de aprendizado e treinamento de conceitos de otimização, diretivas de compilação de código e assembly.

Rodando no Notebook

Diretiva	Mergesort (s)
-O0	2.945 ± 0.130
-O1	1.477 ± 0.082
-O2	1.433 ± 0.017
-O3	1.449 ± 0.067
-Ofast	1.422 ± 0.022
-Os	1.510 ± 0.033
-Og	1.585 ± 0.081

Rodando no Desktop

Diretiva	Mergesort Otimizado (s)
-O0	2.571 ± 0.154
-O1	1.230 ± 0.028
-O2	1.222 ± 0.020
-O3	1.178 ± 0.012
-Ofast	1.225 ± 0.013
-Os	1.233 ± 0.019
-Og	1.243 ± 0.008