9º Exercício Prático
Desenvolvido no Laboratório

Objetivo

Comparar o tempo de execução de operações com números com ponto flutuante e inteiros.

Materiais

1. Compilador GCC
2. Arquivo ConverteParaPretoEBranco.c
3. Imagem Lapis.ppm

Introdução

As operações aritméticas em inteiros, especialmente a adição e subtração, foram base para o desenvolvimento dos primeiros processadores por serem operações simples e facilmente construídas no hardware. Nestes processadores, as operações com ponto flutuante (número real) eram construídas em software levando a baixo desempenho em cálculos matemáticos. Atualmente, os processadores trazem hardware dedicado ao processamento de números em ponto flutuante, porém, mesmo assim, devido à complexidade do hardware, as operações com ponto flutuante pode ter desempenho menor do que as operações com inteiros.

Uma alternativa é usar inteiros como números reais com ponto fixo, nesta abordagem, um número fixo de bits menos significativos de um inteiro é reservado para manter a parte fracionária e o restante dos bits para manter a parte inteira e o sinal. Veja em https://duino4projects.com/art-representing-floating-point-numbers-integers/ . Para entender melhor como representar números de ponto flutuante como inteiros veja https://theramblingness.wordpress.com/2015/07/01/the-art-of-representing-floating-point-numbers-as-integers-2/ .

Para ver a diferença de desempenho entre reais e inteiros em processadores mais simples, o site https://learn.adafruit.com/embedded-linux-board-comparison/performance apresenta uma comparação entre operações com inteiros e pontos flutuantes em Arduino e Raspberry Pi.

Desenvolvimento

O exercício de hoje consiste em transformar uma imagem para preto e branco, processando cada pixel através de ponteiros e comparando os tempos com o processamento feitos usando números com ponto flutuante e com inteiros considerados como números reais de ponto fixo. Como os inteiros são bem mais simples para calcular, costumam ser mais rápidos que com reais.

Executando o programa sem otimização

1. Compile o programa ConverteParaPretoEBranco.c

    - gcc -masm=intel -g -O2 ConverteParaPretoEBranco.c -o ConverteParaPretoEBranco
2. Execute o programa.
    - ./ConverteParaPretoEBranco

Qual foi o tempo observado na execução?

Tempo 1: _____ s

Utilizando ponteiros para fazer acesso aos pixels

Cada um dos elementos de uma matriz (ou no caso de uma imagem) podem receber acessos através de um ponteiro que é deslocado através da matriz. Nesta abordagem, um ponteiro indicando a imagem de entrada e outro indicando a imagem de saída são usados para apontar o pixel sendo processado. **Esta alteração não se trata de uma otimização para o tempo**, mas é uma preparação para otimizações futuras. Para entender melhor, pesquise sobre aritmética de ponteiros em linguagem C.

1. Abra o arquivo ConverteParaPretoEBranco.c em um editor de texto
2. Altere a função processa() como código abaixo

```
void processa(struct Pixel img[ALTU_IMG][LARG_IMG],
              struct Pixel imgSai[ALTU_IMG][LARG_IMG]) {
  struct Pixel *pImg = &img[0][0];
  struct Pixel *pImgSai = &imgSai[0][0];
  int i;
  for (i = 0; i < ALTU_IMG * LARG_IMG; i++) {
    pImgSai->r = pImgSai->g = pImgSai->b =
        0.299 * pImg->r + 0.587 * pImg->g + 0.114 * pImg->b;
    pImg++;
    pImgSai++;
  }
}
```

Qual foi o tempo observado na execução?

Tempo 2: _____ s

Explique, como é possível transformar cada pixel da imagem utilizando apenas um ponteiro para a imagem de entrada e outro para a imagem de saída.

Pode ser que o tempo piore um pouco pois o otimizador do compilador pode não encontrar uma solução tão boa usando ponteiros. Apenas por exploração, tente executar sem a otimização “-O2” do compilador e obter os tempos 1 e 2 novamente, note que os ponteiros simplificam o processamento. Continue os próximos testes com “-O2”.

Utilizando operações com números de ponto fixo

As multiplicações com números reais podem ser substituídas por multiplicações com inteiros. As

constantes 0,299, 0,587, 0,114 são substituídas por estes valores multiplicados por 1024, resultando respectivamente em: 306, 601, 116. Finalmente, para que a soma resultante das multiplicações volte à grandeza esperada para os canais da imagem, esta soma é dividida por 1024. O 1024 foi escolhido de forma que os inteiros representem um número real de ponto fixo com 10 bits para manter a parte fracionaria, além disso, a divisão, necessária para obter a parte inteira, pode ser feita utilizando deslocamento de 10 bits à direita.

1. Altere a função processa() como código abaixo que opera com inteiros:

```
void processa(struct Pixel img[ALTU_IMG][LARG_IMG],
              struct Pixel imgSai[ALTU_IMG][LARG_IMG]) {
  unsigned int fr = (unsigned int)(1024 * 0.299);
  unsigned int fg = (unsigned int)(1024 * 0.587);
  unsigned int fb = (unsigned int)(1024 * 0.114);
  struct Pixel *pImg = &img[0][0];
  struct Pixel *pImgSai = &imgSai[0][0];
  int i;
  for (i = 0; i < ALTU_IMG * LARG_IMG; i++) {
    pImgSai->r = pImgSai->g = pImgSai->b =
        (fr * pImg->r + fg * pImg->g + fb * pImg->b) >> 10;
    pImg++;
    pImgSai++;
  }
}
```

   Qual foi o tempo observado na execução?

   Tempo 3: _____ s

Transformando em linguagem de montagem

O código abaixo não otimiza de forma significante o tempo, mas serve como **exemplo de aplicação do modo de endereçamento indexado com base, deslocamento e escala** em linguagem de montagem.

1. Altere a função processa() como código abaixo:

```
void processa(struct Pixel img[ALTU_IMG][LARG_IMG],
              struct Pixel imgSai[ALTU_IMG][LARG_IMG]) {
  unsigned int fr = (unsigned int)(1024 * 0.299);
  unsigned int fg = (unsigned int)(1024 * 0.587);
  unsigned int fb = (unsigned int)(1024 * 0.114);

  asm("   MOV    RSI, %[img]              \n"
      "   MOV    RDI, %[imgSai]           \n"
      "   MOV    RDX, %[tamanho]          \n"

      "REPETE%=:                          \n"
      "   MOVZX  EAX, BYTE PTR [RSI+RDX]  \n"
      "   MOVZX  EBX, BYTE PTR [RSI+RDX+1]\n"
```

```
        "   MOVZX  ECX, BYTE PTR [RSI+RDX+2]\n"

        "   IMUL   EAX, %[fr]               \n"
        "   IMUL   EBX, %[fg]               \n"
        "   IMUL   ECX, %[fb]               \n"

        "   ADD    EAX, EBX                 \n"
        "   ADD    EAX, ECX                 \n"

        "   SHR    EAX, 10                  \n"

        "   MOV    [RDI+RDX],AL             \n"
        "   MOV    [RDI+RDX+1],AL           \n"
        "   MOV    [RDI+RDX+2],AL           \n"

        "   SUB    rdx, 3                   \n"
        "   JNZ    REPETE%=                 \n"
        :
        : [img] "m"(img), [imgSai] "m"(imgSai),
          [fr] "i"(fr), [fg] "i"(fg), [fb] "i"(fb),
          [tamanho] "i"(ALTU_IMG * LARG_IMG * 3 - 3)
        : "rsi", "rdi", "rax", "rbx", "rcx", "rdx");
}
```

Qual foi o tempo observado na execução?

Tempo 4: _____ s

Explique, como funciona o modo de endereçamento utilizado nas linhas e para que estas linhas servem no programa?

```
        "   MOV    [RDI+RDX],AL             \n"
        "   MOV    [RDI+RDX+1],AL           \n"
        "   MOV    [RDI+RDX+2],AL           \n"
```

Analisando os resultados

Como o processamento de números reais depende muito do processador usado, nesta atividade é muito importante detalhar na avaliação do resultado o processador que equipa o computador utilizado nas medidas.

Envie a avaliação dos resultados como descrito no arquivo “Avaliacao Dos Resultados.pdf”.

Conclusão

Os processadores modernos utilizam recursos construídos em hardware para o tratamento de operações com números reais com ponto flutuante, mesmo assim, alguma vantagem pode ser alcançada no processamento com número com ponto fixo baseados em números inteiros, mas cuidado, problemas de precisão podem ocorrer.

Processadores mais simples encontrados em sistemas embarcados podem não apresentar tratamento de número de ponto flutuante por hardware o que leva a um grande impacto negativo em processamentos com reais.