12º Exercício Prático – parte 2
Desenvolvido no Laboratório

Objetivo

Observar como aplicar as instruções SIND disponíveis no processador da arquitetura IA-32

Materiais

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

Introdução

O experimento de hoje usa novamente um programa que converte uma imagem colorida em preto e branco.

Desenvolvimento

Novamente a intenção é comparar a otimização -O2 com a -O3 que usa instruções SIMD e apresentar um programa escrito em linguagem de montagem que também usa instruções SIMD.

Executando o programa

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

Usando a otimização do compilador -O3

A opção -O3 ativa no compilador um nível de otimização ainda maior que o -O2, muitas destas otimizações usam instruções SIMD.

1. Compile o programa ConverteParaPretoEBrancoA .c
   - gcc -masm=intel -g **-O3** ConverteParaPretoEBranco.c -o ConverteParaPretoEBranco
2. Execute o programa.
   - ./ConverteParaPretoEBranco

Qual foi o tempo observado na execução?

Tempo 2: _____ s

Usando linguagem de montagem e instruções SIMD

As instruções SIMD podem ser usadas para construir rotinas mais otimizadas que executam uma instrução em muitos dados de uma só vez.

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]) {
  // o 130 deveria ser 128, mas foi aumentado até que fr+fg+fb == 128
  unsigned char fr = (unsigned char)(130 * 0.299);
  unsigned char fg = (unsigned char)(130 * 0.587);
  unsigned char fb = (unsigned char)(130 * 0.114);
  unsigned char frgb[16] = {fr, fg, fb, 0, fr, fg, fb, 0,
                            fr, fg, fb, 0, fr, fg, fb, 0};
  unsigned char shuflleEntrada[16] = {0, 1, 2, 0xff, 3, 4,  5,  0xff,
                                      6, 7, 8, 0xff, 9, 10, 11, 0xff};
  unsigned char shuflleSaida[16] = {0, 0,  0,  4,  4,    4,    8,    8,
                                    8, 12, 12, 12, 0xff, 0xff, 0xff, 0xff};

  asm("   MOV       RSI, %[img]              \n" // carrega endereço de memória da imagem original
      "   MOV       RDI, %[imgSai]           \n" // carrega endereço da imagem de saída
      "   MOVUPS    XMM0, %[frgb]            \n" // carrega os 12 fatores mais 4 bytes de alinhamento
      // os canais dos pixels da memoria precisam ser alinhados de 4 em 4 para facilitar as operações
      "   MOVUPS    XMM3, %[shuflleEntrada]  \n" //
      "   MOVUPS    XMM4, %[shuflleSaida]    \n"
      "   MOV       RDX, 0                   \n"

      "REPETE%=:                             \n"
      "   MOVUPS    XMM1, [RSI+RDX]          \n" // le             [r1 g1 b1 r2 g2 b2 r3 g3 b3 r4 g4 b4 r5 g5 b5 r6]
      // intercala os bytes para que fiquem alinhados com os fatores
      "   PSHUFB    XMM1, XMM3               \n" // intercala      [r1 g1 b1 0  r2 g2 b2 0  r3 g3 b3 0  r4 g4 b4 0 ]
      // multiplixca os canais com os fatores e soma pares gerando valores de 16 bits
      // frgi = ri*fr+gi*fg e fbi = bi*fb+0*0
      "   PMADDUBSW XMM1, XMM0               \n" // mult add       [frg1  fb1   frg2  fb2   frg3  fb3   frg4  fb4  ]
      // garante que caberá, mas deixa 3 bits depois da "virgula"
      "   PSRLW     XMM1, 4                  \n" // divide cada uma das 8 words (16 bits) por 16
      // copia XMM1 para XMM2
      "   MOVUPS    XMM2, XMM1               \n"
      // inverte, de duas em duas words, para facilitar a soma frgi + fbi
      "   PSHUFLW   XMM2, XMM2, 0b10110001   \n" //                [fb1   frg1  fb2   frg2  frg3  fb3   frg4  fb4  ]
      "   PSHUFHW   XMM2, XMM2, 0b10110001   \n" //                [fb1   frg1  fb2   frg2  fb3   frg3  fb4   frg4 ]
      // nivel de cinda do pixel ci = frgi + fbi
      "   PADDW     XMM2, XMM1               \n" //                [c1    c1    c2    c2    c3    c3    c4    c4   ]
      // remove o resto da parte fracionária
      "   PSRLW     XMM2, 3                  \n" // divide cada uma das 8 words (16 bits) por 8
      // repete os níveis de cinza reposicionando os bytes
      "   PSHUFB    XMM2, XMM4               \n" //                [c1 c1 c1 c2 c2 c2 c3 c3 c3 c4 c4 c4  0  0  0  0]
      // coloca os 16 bytes para a imagem de saída. os 12 bytes de 4 pixels são seguidos por outros 4 bytes 0
      // estes 4 bytes mais significativos, serão substituidos pelo nivel de cinza correspondente ao próximo pixel
      // na próxima iteração.
      "   MOVUPS    [RDI+RDX], XMM2          \n"

      "   ADD    RDX, 12                     \n"
      "   CMP    RDX, %[finalImg]            \n"
      "   JL     REPETE%=                    \n"
      :
      : [img] "m"(img), [imgSai] "m"(imgSai), [frgb] "m"(frgb),
        [shuflleEntrada] "m"(shuflleEntrada), [shuflleSaida] "m"(shuflleSaida),
        [finalImg] "i"(ALTU_IMG * LARG_IMG * 3)
      : "rsi", "rdi", "rdx", "xmm0", "xmm1", "xmm2", "xmm3", "xmm4");
}
```

Qual foi o tempo observado na execução?

Tempo 3: _____ s

Avaliando os resultados

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

Conclusão

As instruções SIMD pode trazer alguma vantagem no tempo de execução de um programa quando bem utilizadas.