UNIVERSIDAD DE BUENOS AIRES

Facultad de Ciencias Exactas y Naturales
Departamento de Computación
Organización del Computador

TRABAJO PRÁCTICO NÚMERO 2

Nombre de Grupo: Napolitana con Jamón y Morrones

Alumnos:

Izcovich, Sabrina | sizcovich@gmail.com | LU 550/11

López Veluscek, Matías | milopezv@gmail.com | 926/10

Índice

1. Introducción

El objetivo de este trabajo práctico fue experimentar utilizando el modelo de programación SIMD. Para ello, fue requerido implementar seis filtros para procesamiento de imágenes (Recortar, Halftone, Umbralizar, Colorizar, Efecto Plasma y Rotar) tanto en C como en $Assembler$.
Por otro lado, debimos analizar la performance de un procesador al hacer uso de las operaciones SIMD. Para ello, realizamos comparaciones de velocidad entre los dos tipos de implementaciones realizados utilizando la herramienta Time Stamp Counter (TSC) del procesador.

2. Desarrollo

Nuestro trabajo consistió en implementar los filtros mencionados anteriormente. Para el desarrollo del mismo, diversas herramientas fueron necesarias. En esta sección, se explicita la utilización de las mismas aclarando en qué nos ayudaron a lograr una correcta ejecución de nuestra implementación.

2.1. Implementación en C

En el caso de C, las implementaciones se realizaron siguiendo algoritmos sencillos. Para recorrer las matrices de píxeles utilizamos for pues nos pareció lo más conveniente considerando que nuestros valores estaban pre-establecidos y que al colocar un for dentro de otro (uno para las filas y otro para las columnas) era más simple recorrer todas las posiciones de una matriz.
Al probar iterar primero las columnas y luego las filas, rotar.c giraba la imagen hacia el otro lado; luego, nos decidimos por iterar primero las filas y luego las columnas.
Por otro lado utilizamos, como tipos de datos, enteros y doubles. Esto se debió a que era lo más conveniente para las operaciones que debía realizar nuestro programa ya que el nivel de precisón alcanzado era mayor y esa cualidad es muy importante para no perder información en cada píxel. Un incoveniente que tuvimos y que nos resultó llamativo fue al implementar rotar.c ya que al definir $\sqrt{2}/2$ como double la imagen obtenida presentaba diferencias con la imagen que debíamos encontrar. Para resolverlo, definimos $\sqrt{2}/2$ como float y dicho problema se solucionó.

Luego, con el fin de disminuir el tiempo de ejecución de los C, comprimimos partes del código reduciendo, a su vez, la cantidad de líneas. También, alteramos pequeños detalles que sumados marcarían diferencia. Por ejemplo, cambiamos j++ por ++j.

2.2. Implementación en Assembler

En el caso de *Assembler*, utilizamos la extensión SSE para procesar varios bits a la vez. Esta herramienta nos permitió acelerar la ejecución de nuestro programa dado que en cada ciclo (en la mayoría de los casos) fueron tratados 16 bits a la vez.
Nuestros filtros fueron realizados de la siguiente manera:

- **Recortar:** En el caso del filtro recortar, procesamos los 4 cuadrantes de la imagen por separado para no confundir las posiciones de origen y destino de los píxeles por lo que la implementación de cada cuadrante resultó ser la misma a diferencia del offset de destino y origen.
  Dicha implementación consistió, a grandes rasgos, en levantar de a cada 16 bits desde la posición de origen y pegarlos en la posición de destino. Una vez alcanzado el valor de *tam*, la siguiente fila era procesada.
  Para realizar dicha función no resultó necesario desempaquetar y empaquetar los píxeles pues no debían ser utilizados para realizar cuentas. Por lo tanto, a medida que se los levantaba, se los copiaba tal cual a la imagen destino.

  La función consistió en una seguidilla de los siguientes pasos:
  En primer lugar, se procesa el cuadrante B. Para ello, se le resta el *tam* al ancho de la imagen para obtener el offset. Luego, se calcula el offset vertical de destino multiplicando el dst_row_size por tam.

  - Comparación de Tiempo

  Al probar lena.bmp de 512x512, con una cantidad de 100 iteraciones, el tiempo obtenido fue el siguiente:

  Implementación C
  ————————-

  Tiempo de ejecución:
  Comienzo : 857860615438390
  Fin : 857860685974035
  # iteraciones : 100
  # de ciclos insumidos totales : 70535645
  # de ciclos insumidos por llamada : 705356.500

  Implementación ASM
  ————————

Tiempo de ejecución:
Comienzo : 857860758957933
Fin : 857860759960311
iteraciones : 100
de ciclos insumidos totales : 1002378
de ciclos insumidos por llamada : 10023.780

Resultados
———-

Ciclos C: 705356.5
Ciclos ASM: 10023.78
Ciclos ASM respecto de C: 1.42109415593 %
Tiempo C: 70535645
Tiempo ASM: 1002378
Tiempo ASM respecto de C: 1.42109425667 %

- **Halftone:**
  - Comparación de Tiempo

Al probar lena.bmp de 512x512, con una cantidad de 100 iteraciones, el tiempo obtenido fue el siguiente:

- **Umbralizar:**
  - Comparación de Tiempo

Al probar lena.bmp de 512x512, con una cantidad de 100 iteraciones y 64 128 16 como parámetros, el tiempo obtenido fue el siguiente:

Implementación C
———————-

Tiempo de ejecución:
Comienzo : 857572172822622
Fin : 857572676855448
iteraciones : 100
de ciclos insumidos totales : 504032826
de ciclos insumidos por llamada : 5040328.500

Implementación ASM
———————

Tiempo de ejecución:
Comienzo : 857572838473791
Fin : 857572938386751
iteraciones : 100
de ciclos insumidos totales : 99912960
de ciclos insumidos por llamada : 999129.625

Resultados
———-

Ciclos C: 5040328.5
Ciclos ASM: 999129.625
Ciclos ASM respecto de C: 19.8227084802 %
Tiempo C: 504032826
Tiempo ASM: 99912960
Tiempo ASM respecto de C: 19.8227089281 %

- **Colorizar:**
  - Comparación de Tiempo

Al probar lena.bmp de 512x512, con una cantidad de 100 iteraciones y 64 128 16 como parámetros, el tiempo obtenido fue el siguiente:

- **Waves:**
  - Comparación de Tiempo

Al probar lena.bmp de 512x512, con una cantidad de 100 iteraciones y 2.0 4.0 16.0 como parámetros, el tiempo obtenido fue el siguiente:

Implementación C
———————-

Tiempo de ejecución:
Comienzo : 858182624211702

Fin : 858224537429979
iteraciones : 100
de ciclos insumidos totales : 41913218277
de ciclos insumidos por llamada : 419132160.000

Implementación ASM
———————

Tiempo de ejecución:
Comienzo : 858224711685883
Fin : 858225286387416
iteraciones : 100
de ciclos insumidos totales : 574701533
de ciclos insumidos por llamada : 5747015.000

Resultados
———-
Ciclos C: 419132160.0
Ciclos ASM: 5747015.0
Ciclos ASM respecto de C: 1.37117013402 %
Tiempo C: 41913218277
Tiempo ASM: 574701533
Tiempo ASM respecto de C: 1.37117013826 %

- **Rotar:**
  - Comparación de Tiempo

Al probar lena.bmp de 512x512, con una cantidad de 100 iteraciones, el tiempo obtenido fue el siguiente:
Implementación C
———————-

Tiempo de ejecución:
Comienzo : 858387565588994
Fin : 858388404276534
iteraciones : 100
de ciclos insumidos totales : 838687540
de ciclos insumidos por llamada : 8386875.500

Implementación ASM

Tiempo de ejecución:
Comienzo : 858388558871397
Fin : 858388874991387
iteraciones : 100
de ciclos insumidos totales : 316119990
de ciclos insumidos por llamada : 3161200.000

Resultados

Ciclos C: 8386875.5
Ciclos ASM: 3161200.0
Ciclos ASM respecto de C: 37.6922251916 %
Tiempo C: 838687540
Tiempo ASM: 316119990
Tiempo ASM respecto de C: 37.6922244487 %

Comparación usando : call, muchos llamados a memoria. Se guarda todo en la pila y hace los llamados a la pila, ej:

hay que poner el pseudocodigo detallado que utilizamos explicando por que recorrimos de la manera que recorrimos la matriz y por que nos parecio lo mas efectivo y eficiente para programar, detallando para que sirve cada cosa. Tambien tenemos que decir que cosas hicimos para facilitar la implementacion en asm, o sea si lo hicimos pensando en como ibamos a resolverlo en asm o si directamente lo hicimos de una manera que fuera rapida. algo que se me ocurre es que preferimos, por ejemplo, usar muchas variables para definir pequeñas operaciones que usaba una sola ecuacion en vez de que fuera todo resuelto de una para poder darles tipos a esas operaciones y que el resultado fuera mas preciso y acertado a lo pedido.

que transformaciones podriamos hacer para que sea mejor lo que hacemos, mejor tiempo, que fue lo que probamos, etc. Por ejemplo si paso de float a double. En umbralizar pasamos Q de float a double. Pusimos Q, max y min en double porque para hacer las cuentas con doubles era lo mas conveniente.

3. Resultados

Al observar el código generado por objdump para cada función en c, se pudo observar que su código assembler realizaba múltiples llamados a memoria. Ésto se debía a que todos los datos eran almacenados en la pila por lo que todo debía ser buscado allí, por ejemplo, [rbp-0x28].
tenemos que ver el codigo en asm creado por el .c para decir por que nuestro asm funciona mas rapido que el generado por el .c, generalmente produce mas ciclos innecesarios o quizas contempla algun caso que no es importante (por ejemplo una altura o un ancho negativo). bueno obviamente hay que concluir que el .asm es mucho mas rapido que el .c.
tenemos que hacer un grafico, creo que podriamos hacer uno que para cada funcion que hicimos ponga el tiempo que tardo en asm y en .c y que una por un lado todos los .c y por el otro todos los .asm cuestion de ver precisamente cual es la relacion entre la velocidad de un asm y de un .c. tambien tenemos que hacer uno que muestre graficamente el contenido de los xmm a medida que avanza la ejecucion del programa aca hay que poner tablas y graficos con los resultados. analizar y comparar las implementaciones. podriamos ejecutar el timing en distintas computadoras y ver que cambia respecto del procesador usado.

4. Conclusión

C usa la FPU que es poco performante y lleva a un aumento del tiempo de ejecución. Reflexion final sobre alcance de la programacion vectorial a bajo nivel.