Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

TP2

Organización del Computador II

Primer Cuatrimestre de 2014

Grupo: **Colombia/Arepa**

Apellido y Nombre	LU	E-mail
Cisneros Rodrigo	920/10	rodriciris@hotmail.com
Rodríguez, Agustín	120/10	agustinrodriguez90@hotmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Contents

1 Introducción

El lenguaje C es uno de los más eficientes en cuestión de performance, pero esto no quiere decir que sea óptimo para todos los casos o que no haya campos en los que pueda utilizarse una opción mejor. Para comprobar esto, experimentamos con el set de instrucciones SIMD de la arquitectura Intel. Vamos a procesar imágenes mediante la aplicación de ciertos filtros y estudiaremos la posible ventaja que puede tener un código en Assembler[1] con respecto a uno en C. Implementaremos los filtros en ambos lenguajes para luego poder comparar la performance de cada uno y evaluar las ventajas y/o desventajas de cada uno.

[1] Durante el desarrollo del informe puede verse referido a Assembler como ASM haciendo un abuso de notación siendo que este es solo el nombre de la extensión.

2 Desarrollo y Resultados

2.1 Filtro Tiles

2.1.1 Experimento 1 - análisis el código generado

Utilizar la herramienta `objdump` para verificar como el compilador de C deja ensamblado el código C. Como es el código generado, ¿cómo se manipulan las variables locales?¿le parece que ese código generado podría optimizarse?

- Usando la herramienta *objdump* desensamblamos el archivo de extensión .o de código del filtro de Color. Al observar este código, lo primero que notamos es que el compilador no usó instrucciónes de SIMD a pesar de que el procesador tenga esa característica. Esto se debe a que al escribir en lenguaje C no se puede hacer uso de estas instrucciones a menos que usemos una librería aparte que haga uso de estas como puede ser *libSIMDx86*.
  En cuanto a como se manipulan las variables locales, se respetan las convenciones de pushear registros como r15 - r12 y rbx. Pero en casi todo el código se hace uso de las variables por parámetros y se usa mucho la pila moviendo el registro rbp para recorrerla.

- Existen algunas optimizaciones que se pueden realizar a la hora de compilar el código en C. Estas son -O1, -O2 o -O3, las cuales, siendo agregadas como flags a la hora de compilar, mi código ensamblado queda mucho más óptimo.

2.1.2 Experimento 2 - optimizaciones del compilador

Compile el código de C con optimizaciones del compilador, por ejemplo, pasando el flag `-O1`[2]. ¿Qué optimizaciones realizó el compilador? ¿Qué otros flags de optimización brinda el compilador? ¿Para qué sirven?

En particular el flag -O1 hace que el tamaño del código ensamblado sea mucho menor que el código ensablado sin optimización. En particular al mirar el objdump de código con -O1 se pudo apreciar que el código era mucho menor en cantidad de lineas y que la cantidad de registros pusheados a la pila fue mayor.
El flag -O2 hace todas las optimizaciones que pueda en el código que no estén involucradas con optimizaciones de espacio-tiempo. Y por último el flag -O3 abre todas las opmitizaciones de -O2 más algunas extras con el fin de hacer aun más optimo el código.[3]
Haciendo uso de las optimizaciones mencionadas anteriormente, se realizaron experimentos de performance usando el código en ASM, en C y en C compilado con optimizacion -O1, -O2 y -O3.

2.1.3 Experimento 3 - secuencial vs. vectorial

Realice una medición de las diferencias de performance entre las versiones de C y ASM (el primero con -O1, -O2 y -O3).
¿Como realizó la medición?¿Cómo sabe que su medición es una buena medida?¿Cómo afecta a la medición la existencia de outliers?¿De qué manera puede minimizar su impacto?¿Qué resultados obtiene si mientras corre los tests ejecuta otras aplicaciones que utilicen al máximo la CPU? Realizar un análisis **riguroso** de los resultados y acompañar con un gráfico que presente estas diferencias.

[2] agregando este flag a `CCFLAGS64` en el makefile
[3] Para mas información revisar el http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

2.1.4 Experimento 4 - cpu vs. bus de memoria

Se desea conocer cual es el mayor limitante a la performance de este filtro en su versión ASM.

¿Cuál es el factor que limita la performance en este caso? En caso de que el limitante fuera la intensidad de cómputo, entonces podrían agregarse instrucciones que realicen accesos a memoria y la performance casi no debería sufrir. La inversa puede aplicarse si el limitante es la cantidad de accesos a memoria.

Realizar un experimento, agregando múltiples instrucciones de un mismo tipo y realizar un análisis del resultado. Acompañar con un gráfico.

2.2 Filtro Popart

2.2.1 Experimento 1 - saltos condicionales

Se desea conocer que tanto impactan los saltos condicionales en el código del ejercicio anterior con `-O1`. Para poder medir esto, una posibilidad es quitar las comparaciones al procesar cada pixel. Por más que la imagen resultante no sea correcta, será posible tomar una medida del impacto de los saltos condicionales. Analizar como varía la performance.

Si se le ocurren, mencionar otras posibles formas de medir el impacto de los saltos condicionales.

2.2.2 Experimento 2 - cpu vs. bus de memoria

¿Cuál es el factor que limita la performance en este caso?

Realizar un experimento, agregando múltiples instrucciones de un mismo tipo y realizar un análisis del resultado. Acompañar con un gráfico.

2.2.3 Experimento 3 - prefetch

La técnica de *prefetch* es otra forma de optimización que puede realizarse. Su sustento teórico es el siguiente:

Suponga un algoritmo que en cada iteración tarda n ciclos en obtener un dato y una cantidad similar en procesarlo. Si el algoritmo lee el dato i y luego lo procesa, desperdiciará siempre n ciclos esperando entre que el dato llega y que se comienza a procesar efectivamente. Un algoritmo más inteligente podría pedir el dato $i + 1$ al comienzo del ciclo de proceso del dato i (siempre suponiendo que el dato i pidió en la iteración $i - 1$. De esta manera, a la vez que el procesador computa todas las instrucciones de la iteración i, se estáran trayendo los datos de la siguiente iteración, y cuando esta última comience, los datos ya habrán llegado.

Estudiar esta técnica y proponer una aplicación al código del filtro en la versión ASM. Programarla y analizar el resultado. ¿Vale la pena hacer prefetching?

2.2.4 Experimento 3 - secuencial vs. vectorial

Analizar cuales son las diferencias de performace entre las versiones de C y ASM. Realizar gráficos que representen estas diferencias.

2.3 Filtro Temperature

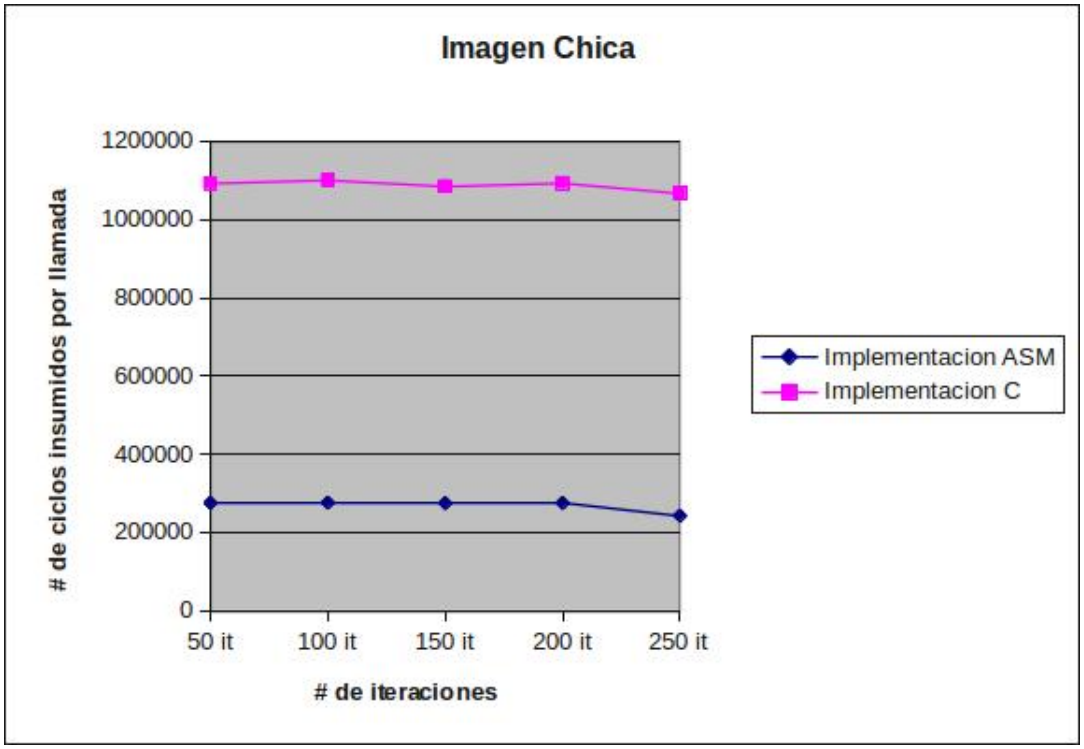
2.3.1 Experimento 1

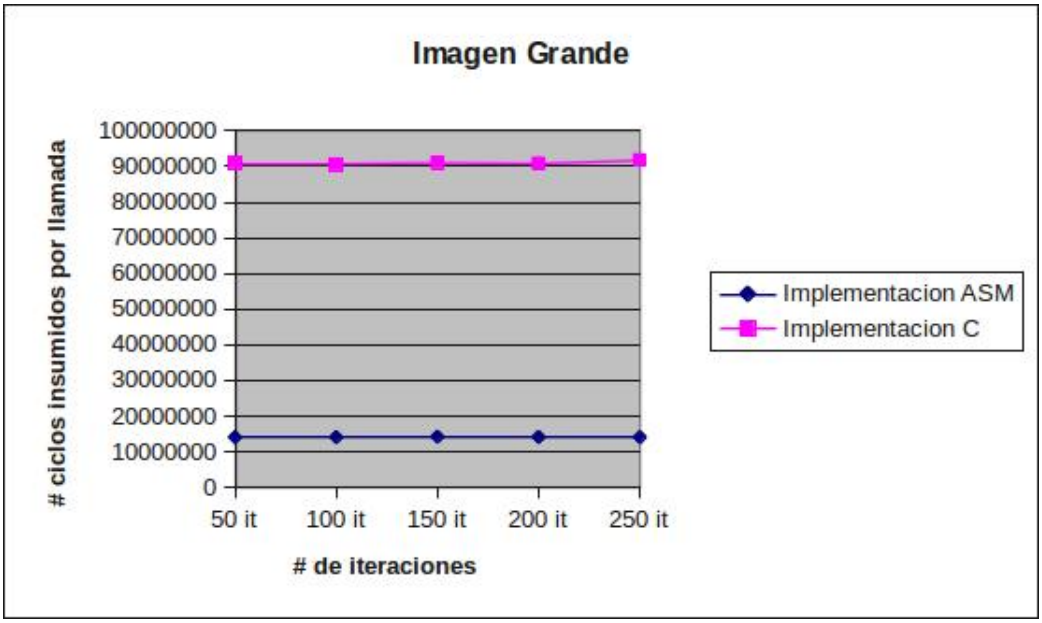
Analizar cuales son las diferencias de performace entre las versiones de C y ASM. Realizar gráficos que representen estas diferencias.

Realizando nuestras mediciones en la implementacion de C de nuestro codigo obtenemos que; Para imagenes de tamaño pequeño (120x56) con 50 iteraciones, el mencionado codigo insume una cantidad de 1092600.125 ciclos por cada llamada. Con 100 iteraciones, la implementacion insume 1100715 ciclos por llamada. Con 150 interaciones, insume 1084809 de ciclos. Con 200 iteraciones, emplea 1093018.875 de ciclos por llamada y finalmente con 250 iteraciones ivierte 1066742 de ciclos por llamada. A su vez, con imagenes de tamaño mas grande (800x800) y 50 iteraciones, el codigo nos insume una cantidad de 91034296 ciclos por llamada, mientras que con 150,200 y 250, demanda 91057448, 90857080 y 91786328 por llamada respectivamente. Mientras que nuestra implementacion realizada en ASM luego de realizar

la respectiva medicion para conocer la performace de esta version obtenemos que: Para imagenes de tamaño pequeño (120x56) con 50 iteraciones, vuestra implementacion insume una cantidad de 275457 ciclos por cada llamada. Con 100 iteraciones, insume 276171 ciclos por llamada. Con 150 interaciones, demanda 274721 de ciclos. Con 200 iteraciones, emplea 275131 de ciclos por llamada y finalmente con 250 iteraciones ivierte 242489 de ciclos por llamada. A su vez, con imagenes de tamaño mas grande (800x800) y 50 iteraciones, el codigo nos insume una cantidad de 14040220 ciclos por llamada, mientras que con 150,200 y 250, demanda 14179786, 14102015 y 14050516 de ciclos por llamada respectivamente.

Para una mejor representación se realizaron distintos graficos mostrando a simple vista las diferencias entre cada tipo de implementacion en cada tipo de imagen.





La diferencia principal entre el código escrito en lenguaje C y el código ASM, es la cantidad de accesos a memoria realizados en cada iteración. Otra diferencia puede verse en la cantidad de bytes que pueden ser procesados simultaneamente durante el mismo ciclo. Mientras que en C leemos y procesamos un byte por vez, el código ASM nos permite acceder y procesar 16 bytes en simultaneos. De todas maneras, en este caso trabajamos con 12 bytes que es un numero mas útil ya que es múltiplo de 3 (cantidad de bytes por píxel) y de 4 (cantidad de bytes por cada caracter). Pudiendo levantar de a 16 bytes en memoria, podría esperarse que el código asm demorase una diesiseiava parte de la cantidad de ciclos que demora el código en C. Pero debemos tener en cuenta que en nuestro caso estamos aprovechando solo 15 de los 16 bytes que leemos en cada iteración por lo tanto es más acertado evaluar como si solo leyeramos 15 bytes. Además, hay que tener en cuenta que las instrucciones SSE pueden necesitar más ciclos que las operaciones comunes que usamos en el C, eso disminuye un poco la performance en la comparación.

2.4 Filtro LDR

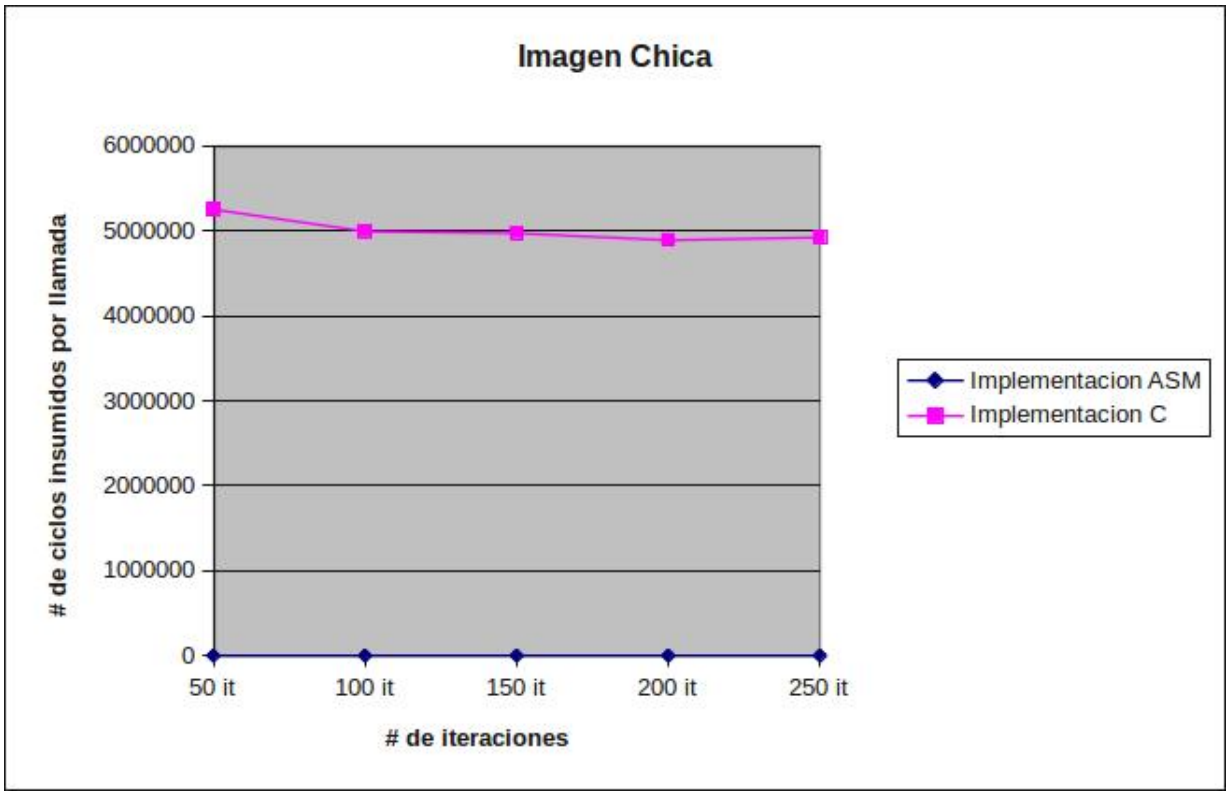
2.4.1 Experimento 1

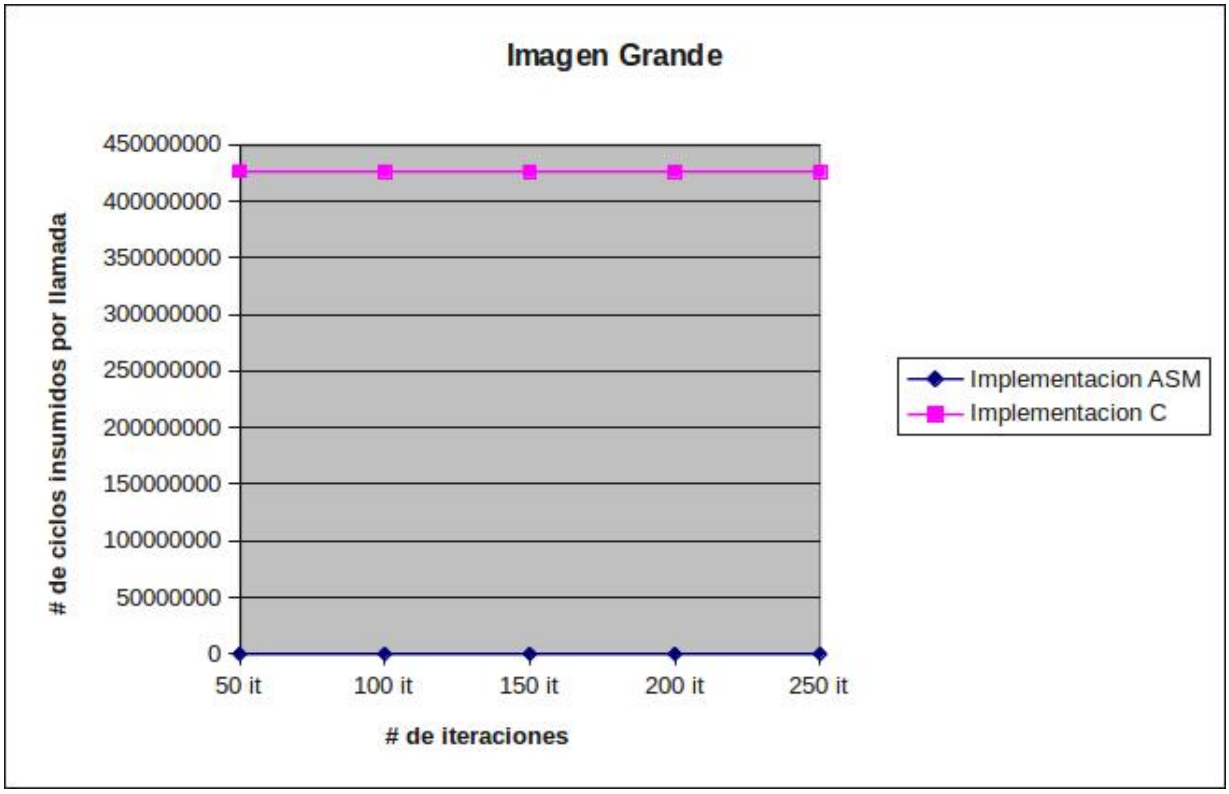
Analizar cuales son las diferencias de performace entre las versiones de C y ASM. Realizar gráficos que representen estas diferencias.

Realizando nuestras mediciones en la implementacion de C de nuestro codigo obtenemos que; Para imagenes de tamaño pequeño (120x56) con 50 iteraciones, el mencionado codigo insume una cantidad de 5251970 ciclos por cada llamada. Con 100 iteraciones, la implementacion insume 4993802 ciclos por llamada. Con 150 interaciones, insume 4973240 de ciclos. Con 200 iteraciones, emplea 4890993 de ciclos por llamada y finalmente con 250 iteraciones ivierte 4923526 de ciclos por llamada. A su vez, con imagenes de tamaño mas grande (800x800) y 50 iteraciones, el codigo nos insume una cantidad de 426247008 ciclos por llamada, mientras que con 150,200 y 250, demanda 425967136, 426037504 y 425995424 por llamada respectivamente. Mientras que nuestra implementacion realizada en ASM luego de realizar la respectiva

medicion para conocer la performace de esta version obtenemos que: Para imagenes de tamaño pequeño (120x56) con 50 iteraciones, vuestra implementacion insume una cantidad de 59,400 ciclos por cada llamada. Con 100 iteraciones, insume 52,060 ciclos por llamada. Con 150 interaciones, demanda 46,600 de ciclos. Con 200 iteraciones, emplea 44,515 de ciclos por llamada y finalmente con 250 iteraciones ivierte 43,956 de ciclos por llamada. A su vez, con imagenes de tamaño mas grande (800x800) y 50 iteraciones, el codigo nos insume una cantidad de 68,400 ciclos por llamada, mientras que con 150,200 y 250, demanda 46,247, 45,105 y 43,920 de ciclos por llamada respectivamente.

Para una mejor representación se realizaron distintos graficos mostrando a simple vista las diferencias entre cada tipo de implementacion en cada tipo de imagen.





La diferencia principal entre el código escrito en lenguaje C y el código ASM, es la cantidad de accesos a memoria realizados en cada iteración.Otra diferencia puede verse en la cantidad de bytes que pueden ser procesados simultaneamente durante el mismo ciclo. En nuestra implamentacion de ASM en una sola iteracion ya obtenemos la sumas correspondientes de los pixeles vecinos, mientras que en la implamentacion del c por cada iteracion se realizan dentro de esa misma varias iteraciones mas sumando los pixeles vecinos y por este motivo principal la diferencia notoria en la performance. Pudiendo levantar de a 16 bytes en memoria, podría esperarse que el código asm demorase una diesiseiava parte de la cantidad de ciclos que demora el código en C. Pero debemos tener en cuenta que en nuestro caso estamos aprovechando solo 15 de los 16 bytes que leemos en cada iteración por lo tanto es más acertado evaluar como si solo leyeramos 15 bytes. Además, hay que tener en cuenta que las instrucciones SSE pueden necesitar más ciclos que las operaciones comunes que usamos en el C, eso disminuye un poco la performance en la comparación.

3 Conclusión

Las instrucciones SIMD (Single Instruction Multiple Data) proveen al programador de una herramienta más efectiva para realizar el mismo conjunto de operaciones a una gran cantidad de datos.

La aplicación de filtros a imágenes era un ejemplo perfecto para probar su eficiencia.

Analizando los resultados de las implementaciones de los 4 filtros, podemos notar:

- Las operaciones básicas (padd, psub, pmul, pdiv, shifts, etc.) SIMD tienen un costo similar a sus correspondientes operaciones unitarias, pero generalmente requieren algún tipo de pre-proceso para poder trabajar con los 16 bytes (pack, unpack, shifts) en una sola iteración, por lo tanto, aunque más eficientes, no lo son en una relación directamente proporcional.
- En el caso que sí hay una relación directamente proporcional es en el acceso a memoria.
- Además, el acceso a memoria es, por lejos, la operación más costosa de las que implementamos en cada filtro.
- Por consecuencia directa del ítem anterior, las llamadas a otras funciones (que a su vez, probablemente contengan variable locales) dentro de una iteración provocan estragos en la efectividad de las implementaciones en C.
- Para poder aprovechar las instrucciones SIMD es un prerequisito que los datos estén contiguos en memoria.

Concluimos que, definitivamente, las instrucciones SIMD, cuando pueden aprovecharse, demuestran una gran eficiencia. Sin embargo, hay que tener algunas consideraciones:

Aunque las imágenes, video y sonido son los primeros candidatos a ser optimizados por paralelización, no todos los procesos pueden ser efectivos y se requiere un análisis profundo de los datos para ver si vale el esfuerzo.

Además, aunque se pueda lograr una gran optimización, no siempre es lo más importante. La optimización seguramente es indispensable en transmisiones de video en vivo, pero baja en importancia si tuviese que ser aplicado una sola vez en una aplicación tipo MS Paint.

Las desventajas que podrían opacar a la optimización son:

El código no es portable, únicamente funciona en procesadores que implementan el set de instrucciones AMD64, requiriendo reescrituras para otras plataformas. Sin embargo el código C debería funcionar perfectamente en IA-32, ARM y cualquier otro procesador que tenga un compilador de lenguaje C.

El código es mucho más largo y difícil de entender (por lo tanto mayor posibilidad de tener bugs) que en un lenguaje de más alto nivel como C. Y en pos de la optimización, se llegan a eliminar funciones (poniéndolas inline), lo que genera código repetido, largo y confuso.