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

Trabajo Practico Final

Organización del Computador II

2014

Multiprogramacion
Comparación de tecnicas de paralelización de trabajo: SISD, SIMD, MISD y MIMD

| Esteban Rey 657/10 estebanlucianorey@gmail.com |

Índice

1. Introduccion

La rapidez entre distintos algoritmos eficaces ante una problematica es uno de los factores determinantes a la hora de la eleccion para la implementacion de la solucion. Tanto su complejidad como adaptacion a los datos en donde se aplican nos proveen una guia a la hora de decidir el mas apto. Adicionalmente a estos factores, tambien se debe tener en cuenta la adaptacion que puede presentar un algoritmo dado a las prestaciones del hardware disponible.

Como primer cualidad a aprovechar, el paralelismo a nivel de datos con la tecnologia SIMD. El uso del tamaño de los registros para guardar mas de un dato e instrucciones que nos permiten manipular los bits guardados nos proveen una oportunidad para computar en menos ciclos la misma cantidad de informacion.

Los procesadores multinucleo, por su parte, nos dan la posibilidad de correr codigo de forma simultanea, en contextos de ejecucion independientes. No es ya una "ilusion" creada por el context switch entre tareas. Agregado a esto, el multithreading en los chips hace uso de las distintas etapas de ejecucion de una instruccion para ejecutar, mediante un pipeline, más de una instruccion al mismo tiempo, aumentando de forma virtual la cantidad de nucleos disponibles en nuestro procesador.

Otra de las caracteristicas de hardware relevante a la hora de hablar de paralelizacion es la placa de video.NVidia, mediante el lenguaje CUDA, nos habilita el uso de procesadores de la GPU, que a pesar de ser individualmente incapaces de realizar prediccion de saltos o ejecucion especulativa, en forma grupal representan un poder de calculo disponible muy importante.

Como ejemplo de un area donde el paralelismo puede ser aplicado se puede mencionar al procesamiento de imagenes. Otro ejemplo que vale destacar en la aplicacion de estas practicas es en la simulaciones de uniones proteinicas; siendo el volumen de datos y cualidades de los algoritmos, demandantes de una importante cantidad de tiempo de procesamiento. En este trabajo se abordara el procesamiento de imagenes como problematica de prueba.

1.1. Estructura de prueba y objetivos

En este trabajo se exploraran distintas formas de aprovechar el hardware disponible mediante el uso de distintas tecnicas de paralelismo. Se comparara el impacto que el hardware realiza sobre 2 algoritmos de procesamiento de imagenes sobre distintos tamaños de entrada.

Los algoritmos de prueba seran un filtro de color, el cual modifica los colores de la imagen de entrada dependiendo de su parametrizacion; y un filtro miniaturizador de imagen, el cual hace uso de un desenfoque gaussiano en partes del objetivo para dar el efecto de "maqueta". Ambos se encuentran programados en forma SISD y SIMD.

Para las versiones MISD y MIMD se aplicara sobre los filtros una estrategia de schedduling con distintas versiones, las cuales seran comparadas para determinar, en este caso de esta problematica, la que mejor se ajusta en desempeño. La version MISD correra con la version SISD de los algoritmos, mientras que la MIMD correra con la version SIMD. Los detalles sobre dichas versiones son descriptas en la seccion de Scheddulers.

Finalmente, para aprovechar todo recurso disponible, se hara uso del lenguaje CUDA y asi operar de forma paralela sobre la placa de video. En la seccion CUDA se encuentran los detalles de como fueron adaptados los filtros para adecuarse a la dinamica que se usa en este entorno.

El entorno de testeo y sus caracteristicas se encuentran en el apendice. El mismo provee interfaces para agregar filtros y estrategias de schedduling nuevas.

2. Scheddulers

2.1. Caracteristicas del problema a paralelizar

OpenCV es una biblioteca que nos aporta funciones útiles para el tratamiento de strams de Video e Imagenes. Su implementación en lenguaje C++ nos brinda un objeto, el cual nos permite obtener fotograma por fotograma de un video dado; también nos limita al acceso secuencial de los mismos: Este problema, al no poder encontrar la solución, se considero como base para el desarrollo de todos los scheddulers, con lo cual todos comparten la etapa de captura de video como secucencial. Por otro lado, a la hora de la escritura del output, OpenCV también nos aporta un escritor de Streams, que nos permite volcar de manera secuencial las imagenes procesadas dentro de un archivo de video. Para la parte paralelizable del código queda el tratamiento de las imágenes.

2.2. Algoritmo de schedduling y comparaciones

Se realizaron 7 versiones, cada una mejorando a su antecesora en un aspecto del algoritmo. Los cambios se reducen a los siguientes puntos:

- Número de threads utilizados en simultaneo
- Cantidad de trabajo delegado a cada thread
- Rol del schedduler a la hora de delegar trabajo a los threads
- Tiempos muertos entre operaciones

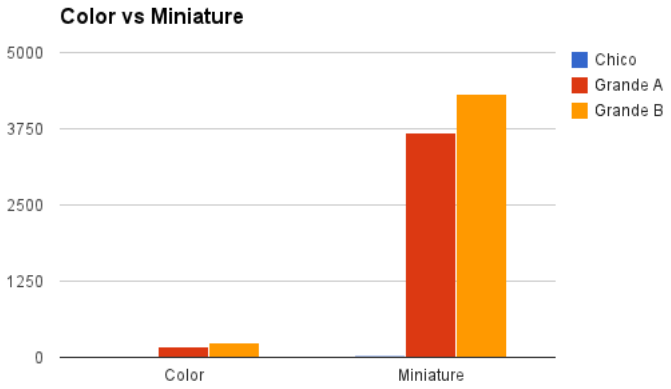
Dado que el tiempo en el que los threads completan su trabajo no es uniforme ni ordenado, el orden de los fotogramas procesados puede no respetar el del video, por lo que al final del filtro se debe recompaginar el archivo de salida. Dicho algoritmo no se tomo en cuenta hasta que se obtuvo una versión aceptable en aspectos de tiempo de ejecución,por lo que las primeras 2 versiones no tienen en cuenta este detalle; al compartir una forma similar de "desordenar" el input al procesarlo, se considero que se necesita el mismo tipo de ordenamiento, con lo cual se asume un tiempo fijo. Ya para las 2 ultimas versiones se contempla esta variable para comparar los metodos con las otras formas de implementar paralelismo.

A la hora de evaluar las versiones se obtienen los siguientes datos del testeo:

- Tiempo de trabajo del schedduler: Sería el tiempo de intervesión en el trabajo de cada thread producido por el schedduler. Este tiempo se cuenta en el momento en que un thread termina con su tarea y recurre al thread organizador para que obtenga el trabajo computado y de ser necesario le asigne más tareas. Se buscará disminuir este tiempo, ya que representa tiempo en el que no se esta trabajando sobre los datos.
- Tiempo de espera del schedduler: Complementariamente al tiempo tomado en el punto anterior, acá se mide cuanto tiempo es el que el schedduler se encuentra sin trabajo, a la espera de la finalización de los threads.
- Tiempo de trabajo de cada thread: Un promedio de lo que tardaron los threads en completar la cantidad de trabajo asignada (igual cantidad para cada uno).
- Tiempo sin trabajo de cada thread: Sería el tiempo en que se deja de aprovechar capacidad de cómputo del sistema: El objetivo es que este tiempo sea el minimo posible.
- Tiempo de reordenamiento: El tiempo requerido por el algoritmo de ordenamiento para compaginar nuevamente el video
- Tiempo total de ejecucion: Parametro principal de comparación: se buscó disminuir tomando en cuenta los tiempos anteriores.

La toma de tiempos se realizó con la biblioteca "*time.h*" con el método gettimeofday(). No se utilizó la funcion clock() ya que la misma contempla el clock de cada procesador y no el tiempo real empleado.

Para la comparación entre distintas versiones se utiliza la formula de "speedup":
speedup = $tiempoAlgoritmoSecuencial/tiempoAlgoritmoParalelo$
Los algoritmos se prueban con 2 tipos de filtros: el miniature, el cual representa a un filtro de procesamiento largo; y el filtro de color que requiere mucho menor procesamiento que el anterior.



Cada filtro se corre con 3 tipos de videos:

- Chica: Video AVI de 960x540 3 segundos de duracion
- Grande A: Video mp4 de 1280 x 720 y 5:33 minutos de duracion
- Grande B: Video WMV de 1920 x 1080 y 2:56 minutos

2.3. Versiones

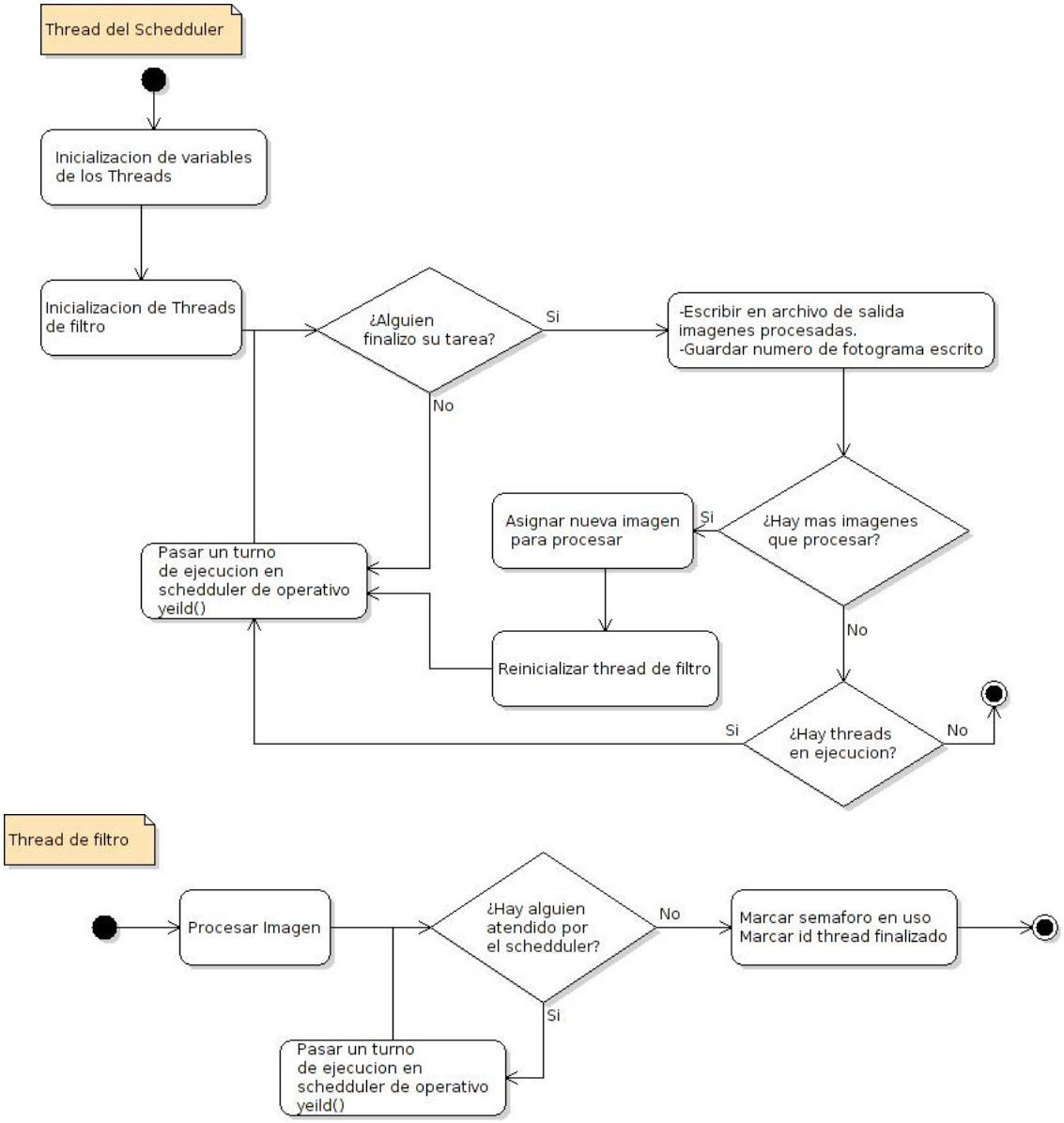
ScheddulerSingleOutput (1er vesion)

Para el primer schedduler se dispuso delegar un fotograma por thread. El sistema cuenta con un numero de threads fijos que estaran ejecutandose en simultaneo, contandose el utilizado por el mismo schedduler para monitoreo.

En la inicialización, se carga una primera tanda de imagenes y se las asigna a todos los threads disponibles, indicando el número de fotograma delegado, para luego poder establecer un orden y asi componer el video final. Una vez terminada la asignación, se ejecutan todos juntos.

Cada thread indica su finalizacion mediante el uso de un flag compartido: el mismo es bloqueado por un semáforo de exclusión mutua (mutex), otorgando el poder de manipular el flag a un thread por vez, y obligando a los demas solicitantes, a esperar la liberacion del mismo. Al detectarse el flag, el schedduler procede a leer la imagen procesada y a guardarla, colocando al fotograma como siguiente en el archivo output. En un arreglo con la misma cantidad de fotogramas que el input, se guarda el indice de fotograma del video original en el indice del fotograma escrito en el archivo destino.

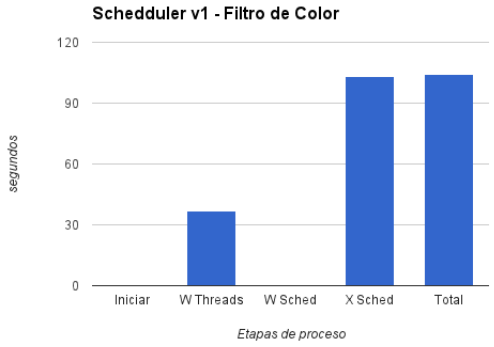
Finalmente, al terminar con todos los fotogramas, se lee el orden en que fueron escritos las imagenes en el output y se recompagina el video.

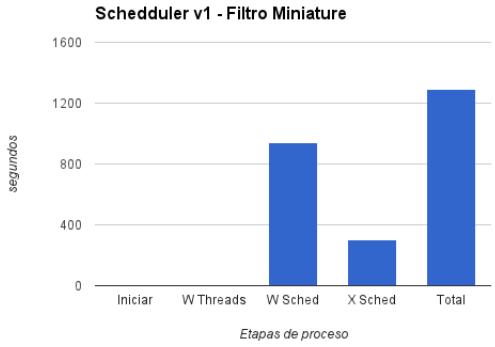


Tiempos

	Chica	Grande A	Grande B
SISD Color	1,68	187	249,08
SISD Miniature	25	3689	4310
MISD Color	1,19	108,96	150,35
MISD Miniature	4,5	656	746

En comparación a la versión SISD del Miniature, se obtiene en las pruebas que la version MISD es 5.64 veces mas veloz. El color resulta con una mejora de 1.59.





Al observar los tiempos empleados en la ejecución, en el filtro Color, la espera de los threads por mas trabajo (W Thread) representa un tiempo considerable en contraste al tiempo total del procedimiento. Esto se debe a que el procedimiento del schedduler es mucho más lento al necesario por los threads para completar su tarea, haciendo esperar a los threads de filtrado y perdiendo tiempo de cómputo.

En el caso del filtro Miniature, sucede lo opuesto. Al ser mas costoso de correr el filtro, el schedduler posee suficiente tiempo para no poner en espera a los threads.

ScheddulerSingleOutput (2da vesion)

El filtro Miniature cuenta con un tiempo de espera por parte del schedduler muy grande, con lo cual, el polling realizado por el schedduler en atencion al flag de solicitud se vuelve una carga pesada para procesar.

Haciendo uso de otro mutex se puede lograr el mismo efecto que el que sucede con los threads de filtro. Dandoles el control de un semáforo, podemos simular "interrupciones" para el schedduler: Luego de la inicialización, el schedduler entra en una condicion de espera para leer las imagenes producidas despues de ejecutar a los threads. Al finalizar un thread con su filtrado, solicita el acceso al flag de notificación; al obtenerlo, señaliza al semáforo del schedduler para que este se active y gestione los datos producidos. Finalmente el schedduler vuelve a la condicion de espera para el proximo thread. Esto permitiría mejorar el tiempo en el caso del filtro Miniature.

Resultados de mejora

	Chica	Grande A	Grande B
SISD Color	1,68	187	249,08
SISD Miniature	25	3689	4310
MISD Color	1,9	201,28	263,46
MISD Miniature	4,33	655	718
Reordenamiento	0,79	87,16	115

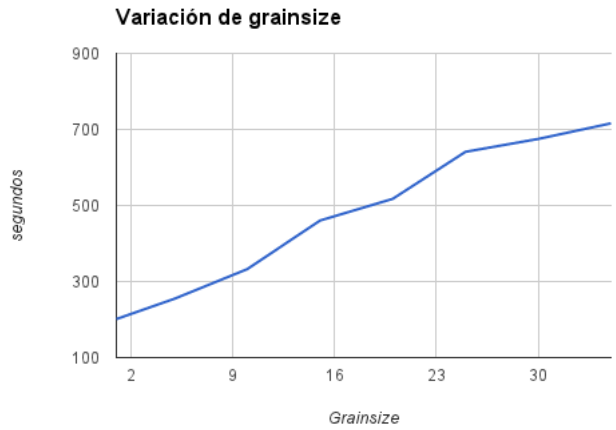
La mejora produce en la aplicación del filtro un speedup de 5.79 con respecto a la version serial. Vale notar que esta version ya cuenta con un algoritmo de ordenamiento. Restando los tiempos de reordenamiento, la nueva version es un 60 % más rápida que la anterior.

El cambio no genera, en el caso del Color y descontando los tiempos empleados para reordenar la secuencia, una mejora respecto a la primer versión. El speedup es de 0,9, esto es, mas lento que la version SISD (igual que la versión anterior de sumarle los tiempos necesarios para reordenar).

ScheddulerMultiOutput (3er vesion)

Para solucionar el problema del tiempo de procesamiento no aprovechado en el filtro de color, se busco la forma de balancear la cantidad de trabajo para que se produzca una mejor sincronizacion entre las dos partes. Como primer solución se probo la manipulación de la cantidad de frames delegados por vez

a los threads, o sea, variar el grainsize del trabajo. Dandoles un arreglo de imagenes, se busco disminuir y espaciar las peticiones al schedduler.



Sin embargo, como se puede observar, el efecto es opuesto al buscado. El hecho de agrandar la cantidad de imagenes para filtrar le aumenta al schedduler el tiempo necesario para leer esa cantidad de imagenes, y el tiempo para escribirlas. Ademas, el metodo no funciona ya que todos los threads de filtrado tienen aproximadamente el mismo tiempo de ejecución.

Otra forma de balancear el trabajo es sacando el exceso de procesamiento al schedduler y pasandoselo a los threads. La escritura de las imagenes al archivo de salida se puede descerealizar mediante el uso de archivos auxiliares: cada thread escribe su propio archivo, dejando solamente la tarea de lectura al schedduler. Esto generaría un aumento en el tiempo de ejecución de los threads de filtrado (ya que deben escribir el archivo de salida) y una disminución en el tiempo que el schedduler esta bloqueado por cada thread solicitante.

Resultados de mejora

	Chica	Grande A	Grande B
SISD Color	1,68	187	249,08
SISD Miniature	25	3689	4310
MISD Color	1,48	179,05	203,31
MISD Miniature	4,33	613	714
Reordenamiento	0,79	87,16	115

Por parte del filtro Miniature se mejora con un speedup de 5.9, y el Color reporta una mejora de 1,23 respecto a la version 2 y de 1,13 respecto al fitro SISD.

Scheddulers y SIMD

Como los scheddulers no interactuan de forma alguna con el algoritmo de los filtros, se pueden utilizar la implementaciones SIMD, combinando asi, paralelismo a nivel instrucciones con datos.

SIMD vs MIMD			
SIMD Color	1	102,14	161,21
SIMD Miniature	5	613,85	747,19
MIMD Color	1,39	167,65	204,89
MIMD Miniature	1,57	219	336

Comparando con el algoritmo SISD, la paralelizacion en el filtro Miniature produce un speedup de 15,8 y de 2,66 respecto a la version SIMD. Por otro lado, el Color fue mejorado en un 17 % en comparacion a SISD; no obstante, la version del mismo en SIMD es 28 % más rápida que la paralelizada.

3. Placa de Video

Adaptacion del algoritmo al entorno CUDA

El uso del GPU mediante CUDA requiere que, al ejecutar el código sobre la unidad, los datos se encuentren en la memoria del dispositivo. Al manejar dimensiones variables de imagenes en los videos, se opto por procesar de a 1 imagen por vez. Como la arquitectura de la placa permite el branching de threads de forma muy eficiente, delegaremos a cada thread el computo de 1 pixel, descomponiendo a la imagen en un nivel en que no es necesario por parte de los algoritmos ejecutar un ciclo para recorrerla. Esto nos lleva a la necesidad de modificar cada uno de ellos para adaptarlos a la nueva modalidad de computo.

CUDA brinda la estructura que llama "kernel", la cual representa el código ejecutado sobre la GPU. Mediante variables de entorno, es posible determinar el indice de thread que esta ejecutando el código, con lo cual, podemos asignar facilmente cada pixel a cada thread disponible. Una vez terminado el procesamiento se debe recuperar la informacion copiando de la memoria del dispositivo a la memoria principal y de ahi compaginar el video de salida. El tratamiento del video en forma secuencial nos quita la necesidad de recompaginarlo con un algoritmo de ordenamiento. Por las caracteristicas del hardware disponible, solo se puede ejecutar 1 kernel por vez, con lo cual se elimina la posibilidad de mezclar con threads del CPU el uso de la GPU.

Aplicando los mismos filtros adaptados a CUDA y comparandolos con los anteriores resultados obtenemos los siguientes datos:

	Chica	Grande A	Grande B
SISD Color	1,68	187	249,08
SISD Miniature	25	3689	4310
SIMT Color	1,56	124,6	185,84
SIMTMiniature	1,83	205,25	227

La implementación, en el caso del Miniature logra un speedup de 16,84, mientras que la de color 1,30. Hay que denotar que en la implementación no se hizo uso de carcteristicas de la GPU como por ejemplo la memoria compartida, la cual es mas rápida que la de uso general (dentro de la placa de video), con lo cual los resultados representan una cota superior de speedup que puede lograrse en el dispositivo.

4. Conclusión

A grandes rasgos, se pueden sacar las siguientes 3 concluciones:

- A la hora de procesar grandes cantidades de datos, el paralelismo representa una estrategia muy buena. Cuanta mayor cantidad de información se debe procesar, la mejora en el rendimiento de los algoritmos paralelos son cada vez mas evidentes. Por otro lado, al operar con pocos datos, los resultados se invierten, ya que el sistema necesario para organizar y generar el paralelismo empiezan a pesar en los tiempos de ejecución.
- Las metodologías, sean Multithreading desde el CPU o en el GPU, son muy suceptibles a su implementación: una mala patición de trabajo puede resultar en procesos mucho mas largos y costosos que las versiones seriales. La organización de las tareas paralelizadas es una cuestion delicada, que requiere el análisis del problema a tratar: no existe una estrategia generica viable para todos los casos.
- Una buena implementación no debe abstraerse de los recursos de hardware disponibles, los mecanismos de cada arquitectura son variables que deben ser entendidas por el programador si se desea realizar mejoras de eficiencia en el codigo.

En una vista mas particular hacia este trabajo, se observo una enorme variedad de estrategias para realizar la organización de los threads.En total hubo 7 implementaciones de schedduling distintas, de las cuales, las 3 expuestas resumen, de cierto modo, la evolución del algoritmo final para el problema que se tenía. El método de paralelización, segun fueron vistos, debe ser elegido conforme al tipo de algoritmo tratado como tambien a las características del input.

Para entradas pequeñas y algoritmos poco costosos (como el Color) SIMD resulta ventajoso en comparación a las versiones con threads, ya que no cuenta con el costo de la sincronización de los threads ni las estructuras necesarias para llevar a cabo el schedduling, las cuales resultan muy costosas en ese nivel. Tambien resulta ventajoso frente al uso de la placa de video ya que, para utilizar la misma, se necesita la transferencia de los datos de la memoria principal a la de la GPU resultando en un overhead considerable.

En cuanto a grandes cantidades de datos, la union de las caracteristicas SIMD y el multithreading logran mejorar sustancialmente el rendimiento de los algoritmos frente a lo que separadamente realizan cada uno. De todos modos, con entradas grandes, el uso de la GPU implica un cambio mucho más importante que los logrados con las otraas estrategias; aunque, no todo algoritmo es apto para correr de forma paralelizada en un dispositivo aislado.

5. Apendice: Entorno de testing

Compilación

Como el proyecto contiene codigo cuda, en el caso de no poseer placa de video que lo soporte, se debera compilar el codigo de la siguiente forma en el ejecutable sin_cuda:

make sin_cuda CFLAGS=-D_SIN_CUDA_

En el caso de tener disponibilidad del entorno CUDA (compilador nvcc instalado) entonces el comando make generara el archivo ejecutable tp, en la carpeta exec.

Ejecución

El ejecutable generado en /exec recibe parametros para correr los distintos tipos de algoritmos con las diferentes tipos de entradas:
./[ejecutable] [estrategia] [archivo de entrada] [filtro a correr] [version schedduler] [pool size] [grain size]

Donde:

- ejecutable: tp si fue compilado con make solamente, sin_cuda en el otro caso
- estrategia: SISD, SIMD, MISD, MIMD y SIMT.
- archivo de entrada: ruta del archivo de video que se quiere filtrar, acepta los formatos soportados por la biblioteca OpenCV.
- filtro a correr: Se encuentran implementados el filtro Color y Miniature. Para mas opciones ver la siguiente seccion.
- version del schedduler: hay 4 versiones: se seleccionan por su cardinal. Solo diponibles para MISD y MIMD
- pool size: cantidad de threads utilizados en simultaneo. Solo diponibles para MISD y MIMD
- grain size: cantidad de fotogramas delegados en la tarea de cada thread por vez. Solo diponibles para MISD y MIMD.

Ejemplo de ejecucion SISD:

./tp SISD ../datasource/medium.mp4 Miniature

Ejemplo de ejecucion MISD:
./tp MISD ../datasource/medium.mp4 Miniature 4 4 1

La cual ejecuta el filtro Miniature sobre el video medium.mp4 con estrategia MISD utilizando 4 threads (mas el thread de control) y grain size de 1 imagen.

Extensiones

Al proyecto pueden agregarse nuevos algoritmos de scheddulers y nuevos filtros de imagenes.

Agregar scheddulers

Los nuevos Scheddulers deben extender la clase Schedduler implementando los métodos abstractos. Se debe agregar el codigo correspondiente al archivo tp para que sea accesible por la interfaz arriba mencionada. El nuevo schedduler correrá como un componente del objeto VideoProcesor:

```
...
VideoProcessor processor;
NuevoSchedduler* sched = new NuevoShedduler();
sched->setGrainSize(grain);
sched->setPoolSize(pool);
processor.setSchedduler(sched);
...
```

Agregar filtros

Para agregar algoritmos de filtrado se debe extender la clase Filter. Para incluir una implementación SIMT (CUDA), debe hacerse uso de un archivo que actúe como interfaz entre c++ y cuda. Este archivo servira para dar la posibilidad de compilar el proyecto sin funciones cuda. Vease un ejemplo con el algoritmo miniature: ver los archivos MiniatureFilter.cpp, MiniatureFilter.hpp, miniature_cuda.cu, miniature_asm.asm y miniature_int.c. Los headers tienen que incluirse en Filters.hpp.

Logueo de resultados

El proyecto cuenta con un logger que posibilita el registro de mensajes por consola y archivo. El mismo se configura en el archivo log4c.config que se encuentra en la carpeta de los ejecutables. Se utilizan semaforos para la correcta impresión de los mensajes, por lo que enlentece la ejecución de los algoritmos. Basado en el componente java log4j, admite 4 niveles de logueo distintos, los cuales inhabilitan los mensajes de menor nivel. Los niveles de la consola y del archivo son independientes: ERROR ¿ WARNING ¿ INFO ¿ DEBUG ¿ TRACE. Para las pruebas se utilizo el nivel INFO y DEBUG mientras que TRACE fue utillizado para monitoreo de los scheddulers (De activarlo se puede ver en el log la inicialización de los threads, captura de imagenes,...etc).

El archivo de salida se puede configurar de forma que se renueve en cada inicialidacion (outputMode = OVERWRITE) o que use el mismo archivo (APPEND) o que use un archivo nuevo cada vez (NEW).