Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
MP-6123 Procesamiento Digital de Imágenes
Profesor: Dr. Daniel Herrera C.
III Cuatrimestre, 2018

Proyecto 1

Optimización de operadores PDI con CUDA

1. Descripción

En este proyecto se optimizarán operadores básicos de procesamiento digital de imágenes en GPU usando CUDA. Se deberán utilizar diferentes técnicas de optimización como multi-threading, shared memory, texture unit, e instrucciones para compartir registros dentro de un warp. El operador a implementar es el filtro máximos/mínimos.

1.1. Filtro de máximos/mínimos

Los filtros de mínimos o de máximos son filtros de rango. Para el presente proyecto se puede suponer que dichos filtros asignan a un pixel central el valor mínimo (o máximo) en una ventana de un tamaño impar dado a su alrededor. Para la optimización puede fijar este valor a 5×5. La estructura de ambos filtros es idéntica, excepto por el operador de rango empleado, por lo que basta con implementar uno de ellos.
La optimización de algoritmos en CPU es muy distinta a la optimización de algoritmos en GPU. Antes de iniciar, refierase a Dokládal and Dokladalov [1] para una explicación detallada de como optimizar este filtro en CPU.
Debido a que los filtros máximos y mínimos están directamente asociados con los operadores morfológicos de dilatación y erosión, respectivamente, pueden buscarse eventualmente optimizaciones de dichas operaciones morfológicas para elementos estructurales cuadrados.
La LTILib-2 tiene los filtros `lti::maximumFilter` o `lti::minimumFilter` en una implementación no optimizada. La OpenCV tiene implementaciones parcialmente optimizadas de la erosión y dilatación. Utilícelos en sus comparaciones de desempeño.

2. Optimización de algoritmos en CUDA

La arquitectura y modelo de programación de un GPU es muy diferente a la de un CPU tradicional. Un CPU avanzado cuenta con 8 o hasta 16 cores de ejecución, lo cual permite correr hasta 32 hilos simultaneamente. Los GPUs más sencillos del mercado pueden ejecutar 1024 hilos simultaneamente. Esto da una tremenda ventaja para tareas paralelas como el procesamiento digital de imágenes. La principal dificultad es asegurarse de que los 1024 hilos esten activos y no esten bloqueados por latencia de memoria.
El GPU tiene una jerarquía clara de memoria: memoria CPU, memoria principal GPU, cache L2, memoria compartida, memoria local, registros. La memoria de CPU se accesa mediante el bus PCI, por lo que es muy lenta. La memoria principal GPU está físicamente en la misma tarjeta y por lo tanto es (al menos) 10x más rápida. La memoria compartida (shared memory) vive dentro del chip y por lo tanto es (al menos) 10x más rápida que la memoria principal (global device memory).

Los registros viven justo al lado de la unidad de ejecución y por lo tanto son (al menos) 10x más rápidos que la memoria compartida. Al optimizar un algoritmo, es vital utilizar al máximo todos estos tipos de memoria.
En [2] encontrará una introducción breve pero completa al modelo de programación en CUDA. Más detalles de la arquitectura se encuenetran en [3]. Un ejemplo de optimización avanzada para el algoritmo de convolución se encuentra en [4].

3. Objetivos

Al finalizar el proyecto, el estudiante podrá:

1. utilizar principios encontrados tras una revisión de bibliografía científica actualizada para proponer una solución avanzada
2. profundizar en el funcionamiento de un algoritmo de modo que permita la optimización de recursos apropiada de un GPU
3. utilizar técnicas avanzadas de programación paralela masiva utilizando CUDA
4. comparar la eficiencia de algoritmos de forma objetiva
5. resumir resultados en un formato científico

4. Metodología

Como pasos para este proyecto:

1. Realice una revisión bibliográfica sobre optimización de algoritmos en CUDA.

2. Implemente un programa que mida el tiempo de ejecución de forma objetiva y precisa (e.g. usando `cudaEvent_t`). Mida una vez el tiempo de ejecutar n repeticiones del algoritmo a prueba, de modo que que usted pueda entonces luego dividir por n para obtener la duración promedio. Reduzca los costos adicionales (*overhead*) del código, para lograr que dicha medición sea precisa. Por ejemplo, considere por separado el tiempo necesario para copiar la imagen a la memoria del GPU del tiempo de ejecución del algoritmo.

   Idee una forma de medir la varianza en la duración de una ejecución. Usted puede medir el tiempo de ejecución en función del tamaño de imagen de entrada.

3. Partiendo del kernel trivial proporcionado, mida su tiempo de ejecución, comparelo con el tiempo de ejecución de una versión en CPU (por ejemplo usando `lti::maximumFilter`). No es necesario tratar de optimizar el algoritmo en CPU, simplemente asegurese que los resultados visualmente equivalentes.

4. Cree nuevos kernels que tengan las mismas entradas y salidas, pero agregue a cada uno una técnica de optimización: filtro separable, multi-threading, shared memory, compartir registros (instrucciones `shfl_xxx`). Cada nuevo kernel incluye todas las técnicas anteriores.

   Cree una tabla con los tiempos de ejecucion de los diferentes kernels. Por ejemplo: `CPU`, `Trivial`, `Separable`, `Separable+Multi-threading`, `Sep+MT+Shared memory`, `Sep+MT+Shared+Shfl`. Puede implementar tantos kernels como quiera y aplicar las técnicas en cualquier orden. El orden sugerido anteriormente es una progresión natural en el proceso de optimización pero usted puede investigar de otra manera.

5. Prepare un artículo en formato científico donde resuma la revisión bibliográfica y los resultados obtenidos.

5. Entregables

1. Los archivos fuente.
2. Un artículo en formato científico con una descripción del método realizado. Para la sección de resultados debe diseñar un método de evaluación cuantitativa comparándo las diferentes implementaciones.

Referencias

[1] Petr Dokládal and Eva Dokladalova. Computationally efficient, one-pass algorithm for morphological filters. *Journal of Visual Communication and Image Representation*, 22(5):411–420, 2011.

[2] Mark Harris. An Even Easier Introduction to CUDA [online]. 2017. URL: https://devblogs.nvidia.com/even-easier-introduction-cuda/ [cited 31 de octubre de 2018].

[3] Cyril Zeller. CUDA C/C++ Basics [online]. 2011. URL: https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf [cited 31 de octubre de 2018].

[4] Joe Stam. Convolution Soup: A Case Study in CUDA Optimization [online]. 2009. URL: http://on-demand-gtc.gputechconf.com/gtc-quicklink/4E03G1 [cited 31 de octubre de 2018].