Optimización de operadores PDI con CUDA

Alvaro Camacho Mora
Escuela de Ingeniería en Electrónica
Instituto Tecnológico de Costa Rica
Procesamiento Digital de Imágenes
IIIC-2018

***Resumen*—Se aborda la optimización del algoritmo de aplicar el filtro de máximos a imágenes de diferentes tamaños utilizando un GPU. El problema es atacado utilizando diferentes técnicas de optimización, empezando con un algoritmo sencillo para después utilizar técnicas como kernel separable, kernel separable mas procesamiento multi-hilo y finalizando con kernel separable mas procesamiento multi-hilo con memoria compartida (*shared memory*). El resultado de cada kernel optimizado es comparado con una implementación trivial ejecutada en CPU.**

I. Introducción

Este documento trata sobre la optimización del tiempo de ejecución del algoritmo de filtro de máximos utilizando técnicas de paralelización orientada a procesadores gráficos (GPU). Se dará una visión general de la arquitectura de un GPU y el paradigma de programación de las técnicas utilizadas para la optimización y así explicar el porque de la reducción en el tiempo de ejecución del algoritmo.
Se tendrá como base para determinar la diferencia en los resultados el algoritmo ejecutado en el procesador general (CPU) y se comparara con los resultados utilizando la técnica *L2-norm*.

II. Filtro de Máximos

El filtro máximos se considera un filtro estadístico de orden ya que funcionan ordenando los valores de la vecindad del punto de interés. Como su nombre lo dice dado una ventana (kernel) se escogerá el valor máximo entre la vecindad del punto de interés, que sera el centro del kernel.
El resultado que se obtendrá al someter una imagen a un filtro de máximo sera la eliminación de del ruido de pimienta o píxeles negros. Dado su funcionamiento, tiene como principal desventaja es que produce un aclaramiento de la imagen debido a que los elementos negros son eliminados o dicho de otra manera se ensanchan las zonas claras y adelgazan las zonas oscuras [1]. La figura 1 muestra como se adelgazan esas zonas negras en la imagen original cuando se compara con la imagen filtrada.

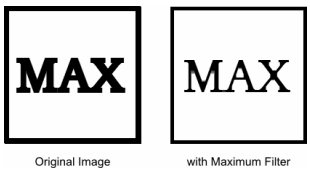

Figura 1: Efecto del filtro de máximos

III. GPU

En días recientes los GPU se han vuelto muy populares para migrar algoritmos de gran complejidad y que pueden aprovechar el poder de procesamiento de los GPU para acelerar el proceso de ejecución. Dos de las ventajas principales por las que se esta tendiendo al uso de GPU para procesamiento paralelo es la arquitectura en si del GPU que provee grandes cantidades de recurso y además una plataforma para el desarrollo de aplicaciones orientadas a GPU como lo es CUDA. En esta sección se brindaran detalles sobre estos dos elementos.

A. *Arquitectura*

Los CPU modernos son sistemas integrados de alto poder de procesamiento, pero mucho de su poder se reduce en el uso de muchos de sus transistores en la memoria cache y el control flujo del programa, esto debido a que como son procesadores de uso general deben enfrentar diferentes problemas y tienen que tener la capacidad de responder ante diferentes necesidades. Por otra parte un GPU, ejecuta el mismo programa en sus diferentes elementos de procesamiento y con esto se reduce dramáticamente el uso de sofisticados diseños de control de flujo. Por otro lado las aplicaciones que son enviadas a un GPU son orientadas a tener gran cantidad de procesamiento aritmético por lo que se prioriza el poder de procesamiento sobre la cantidad de memoria cache[2].

La figura 2 muestra una comparación de un CPU con 4 núcleos con una arquitectura de un GPU. Claramente se puede ver como este CPU tiene la capacidad de ejecutar solamente 4 hilos en paralelo versus el GPU que tiene la capacidad de ejecutar muchos mas hilos en paralelo.

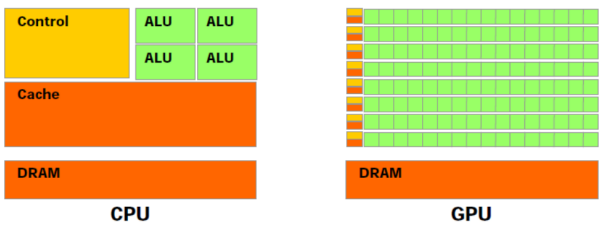

Figura 2: Comparación de arquitecturas entre un CPU y GPU [3]

Una arquitectura clásica de un GPU es mostrada en la figura 3, donde se puede ver que se tiene una cantidad de multiprocesadores que a su vez tienen diferentes unidades funcionales. Cabe mencionar que, además cada multiprocesador tiene diferentes opciones de memoria como lo son:

- *Shared Memory*: 49 kB
- *Texture cache*: 64 kB
- *Constant cache*: 64 kB
- *Registros*: 65536
- *Memoria global*: 2 GB

Se debe mencionar que estos numero son para el GPU utilizado en esta implementación (GeForce 960M) y que la memoria global si puede ser compartida entre los diferentes multiprocesadores. Esta arquitectura usa una arquitectura SIMT (*Single Instruction Multiple Thread*), lo que indica que cada cada instrucción simple es ejecutada en paralelo por múltiples hilos.

Un detalle que se debe tomar en cuenta es que solamente a través de memoria global se puede dar una pseudo comunicación entre multiprocesadores, por lo que esto se debe tomar en cuenta pues las lecturas/escrituras son muy costosas.

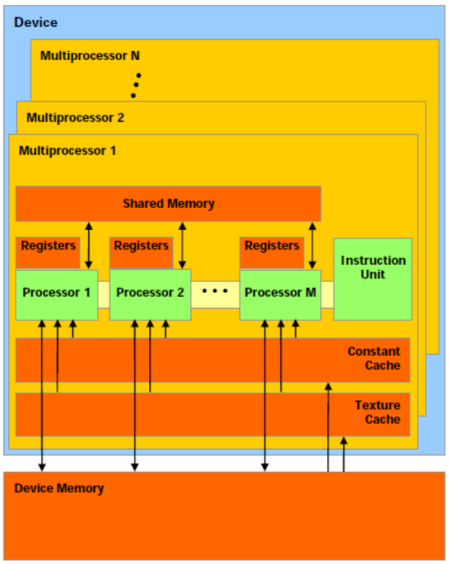

Figura 3: Arquitectura clásica de un GPU[4]

B. *CUDA: Modelo de Programación*

El modelo de programación utilizado en CUDA trata de aplicar el lema "divide y vencerás" ya que al no tener algún tipo de memoria que logre la comunicación entre multiprocesadores se debe dividir el problema en pedazos mas simples para poder ejecutar la aplicación eficientemente.

La figura 4 muestra una cuadricula (grid) dividida en bloques que a su vez estos bloques contienen hilos. El bloque es ejecutado en un multiprocesador especifico, por lo que cada uno de los hilos dentro de este bloque podrá comunicarse entre ellos pero no se podrá comunicar con otro bloque. Asimismo cada bloque tendrá un único identificador 2D dentro de la cuadrilla y probablemente cada hilo un identificador 3D dentro de cada bloque.

Otro importante concepto que se debe tener en cuenta es el de *warp* que básicamente se puede definir como un bloque de 32 threads agrupados consecutivamente que puede correr una misma instrucción y puede comunicarse entre ellos [5].

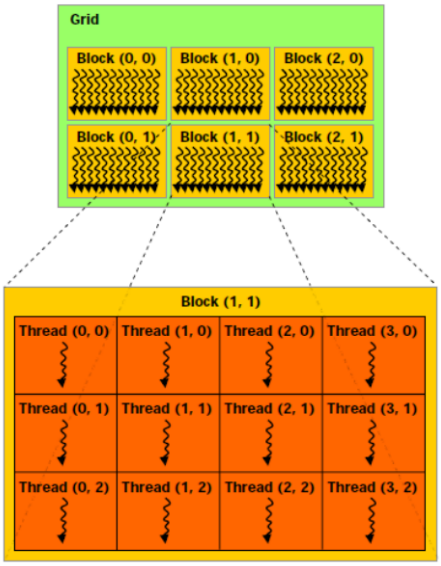

Figura 4: Modelo de programacion de CUDA [3]

IV. Metodología

En esta sección se explica la metodología seguida para lograr los resultados que se explicaran en la siguiente sección.

A. *Sistema utilizado*

La tabla I muestra los detalles sobre el hardware/software utilizado para correr cada uno de las optimizaciones que se plantearan mas adelante:

Cuadro I: Características de software y hardware utilizadas

Dispositivo/Software	Modelo/Versión/Valor
CPU	Intel Core i7-6700HQ
CPU DRAM	12GB
GPU	NVIDIA GeForce GTX 960M
Lti::lib	2
CUDA Toolkit	10.0
Cmake	3.10.2
Ubuntu	18.0.4

B. *Pasos de Optimización*

En esta sección se detallan los diferentes pasos y optimizaciones hechas en cada uno de ellas para reducir el tiempo de ejecución. También se detallan implementaciones del kernel ejecutadas en el CPU pues estas son necesarias como parámetros de referencia.

*1) **Función lti::maximumFilter**:* Como parte de los objetivos del proyecto se debió ejecutar una función parcialmente optimizada como las que son parte de bibliotecas como lo es LTI-lib-2. La función escogida fue: lti::maximumFilter.
Esta función es una implementación muy sencilla de conseguir ya que únicamente se necesita como entrada al constructor el tamaño del kernel y después hacer uso del método *apply* para obtener un imagen filtrada[6].

*2) **Kernel Trivial GPU**:* Como parte del instructivo se proveyó de un algoritmo sencillo para aplicar el filtro de máximos a una imagen. Este código debió ser ejecutado como punto de partida para las optimizaciones realizadas a nivel de GPU.

*3) **Kernel Trivial CPU**:* Básicamente este es el mismo algoritmo sin ninguna optimización entregado como parte del enunciado del proyecto, pero fue ligeramente modificado para que fuese ejecutado en el CPU en lugar del GPU. Esto fue realizado para tener un punto de comparación tanto a nivel de resultados como de tiempo de ejecución.

*4) **Kernel Separable**:* Los filtros separables es de las tecnicas de optimizacion basicas mas poderozas que se tienen ya que como es descrito en [4][7] pues se pasa de n*m operaciones para obtener cada píxel(donde n es la cantidad de filas y m la cantidad de columnas del kernel). Esta propiedad de separabilidad transforma el kernel 2D en dos kernel sencillos 1D que reduce la cantidad de operaciones por cada píxel a n+m.

*5) **Kernel Separable + Multi-Threading**:* Las dos implementaciones anteriores a nivel de GPU fueron ejecutadas utilizando únicamente 1 Thread y 1 block. Esto es como si se estuviese ejecutando en un CPU muy básico. Ahora bien, con la utilización de múltiples bloques e hilos se aumenta el poder de procesamiento. Los principales cambios que se dieron en el kernel es la eliminación de los ciclos *for* que realizaban el recorrido de la imagen píxel por píxel. Esto es gracias a que con esta nueva implementación cada uno de los threads va a ser el encargado de manejar un píxel en la imagen.
La figura 5 muestra por ejemplo una imagen de 5x5 donde la imagen va a ser cargada en el grid y cada thread va a manejar un pixel. Otro detalle a mencionar es lo que se explica en [3], donde se recomienda realizar la asignación del numero de threads y bloques de una forma que la cantidad de bloques sea múltiplo de la cantidad de threads escogida para aumentar la eficiencia y que cada warp sea intensivamente utilizado.

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Figura 5: Representación del acomodo de los threads y blocks dentro del GPU para una imagen de 5x5 [3]

*6) **Kernel Separable + Multi-Threading + Shared Memory**:* Esta etapa de optimización se va a utilizar teniendo en cuenta las etapas anteriores mas una nueva técnica de optimización llamada *shared memory*. Esta técnica utiliza una memoria que esta alojada en cada uno de los bloques y el acceso a ella por cada uno de los threads es mucho menor que lo que tomaría realizar lecturas a memoria global.
Dado que se esta hablando de una memoria dentro de cada bloque, se debe copiar la sección de la imagen que se quiere procesar desde memoria global a la memoria compartida. Este es un paso que se debe realizar pero el *overhead* creado es compensado con la velocidad de procesamiento que se logra utilizando esta memoria. Posterior a que se dio la copia el procesamiento es muy similar al que se ha seguido hasta el momento donde cada thread del bloque lee la memoria compartida, procesa el píxel y se guarda en memoria global como resultado del procesamiento.
Una parte extra que se debe tomar en cuenta es que debido a que se esta copiando una parte de la imagen, cuando se copia a memoria compartido, en los bordes van a haber problemas pues no se tienen los píxeles de la imagen original. La figura 6 muestra este concepto y aunque la representación no corresponde a un kernel separable, el concepto es el mismo . Con esto se estarían "desperdiciando" algunos threads, sin embargo la ganancia vale la pena [3][4][8].

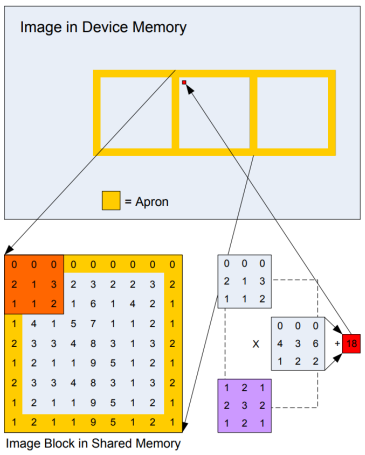

Figura 6: Representación de la copia de datos de memoria principal a memoria compartida [4]

C. Detección de diferencias en la imagen de salida

Para evaluar si las imagenes de salida de cada uno de los kernels es igual al resultado base (CPU y GPU) se va a utilizar la norma Euclidiana o L2-nomr que es una evaluacion matematica para estimar la diferencia entre dos vectores. Bajo este supuesto, si el resultado de evaluar la imagen de referencia contra la salida de cada kernel es igual a cero las imagenes son iguales pues la difenrencia entre ambos vecotres es cero. En caso que se de un valor diferente a cero se estaria en presencia de de algun error en el algoritmo [9][10].

D. *Medición del tiempo de ejecución*

Para evaluar el tiempo que tomo el tiempo la ejecución de cada algoritmo se siguieron dos enfoques:

- CPU: se utilizaron los métodos dentro de la biblioteca *chrono* para realizar la medición del tiempo para los kernel ejecutados en el CPU.
- GPU: para el caso de los kernel ejecutados en el GPU se utilizaron los métodos pertenecientes a la clase *cudaEvent_t*. También fue utilizada la aplicación *nvprof* que es parte del toolkit de CUDA, pero únicamente fue utilizada para comprobar el método de cudaEvent_t.

Para ser mas objetivos en la obtención de resultados reales, se ejecuto 30 veces cada kernel y se tomo el tiempo promedio para el reporte de los resultados. Además se evaluó la varianza del tiempo de ejecución cuando se utilizo imágenes de diferente tamaño.

V. Resultados

A. *Reducción de Tiempos de Ejecución*

Uno de los principales objetivos de este proyecto es la evaluación de la reducción del tiempo de ejecución al aplicar los diferentes kernels optimizados. La tabla II muestra los tiempos de ejecución promedios para cada kernel en donde se puede observar como los kernel ejecutados en el CPU sin muy rápidos, incluso mas rápidos que la ejecución en el GPU utilizando memoria compartida. Claramente se ve una tendencia a que cada kernel optimizado en el GPU reduce significativamente el tiempo de ejecución como se puede observar en la figura 7 como cada optimización hace que baje abruptamente la curva. Cabe mencionar que que el eje vertical de la figura 7 ha sido cambiado a una escala logarítmica para poder observar fácilmente las tendencias de los datos.

Cuadro II: Tiempos promedio de ejecución para cada kernel

Kernel	Tiempo de Ejecución (ms)
LTI:lib	0.01078
Trivial CPU	0.06409
Trivial GPU	909.400
Separable	414.465
Separable + MTH	0.11727
Separable + MTH + Shared Memory	0.07793

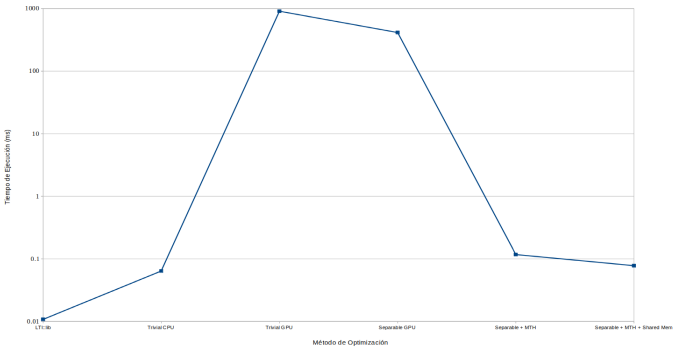

Figura 7: Tiempo de ejecución versus algoritmo utilizado

Cabe mencionar como detalle adicional que para esta evaluación se utilizo la imagen de la figura 8 la cual tiene un tamaño de 720x430.

Figura 8: Imagen utilizada para la evaluación del tiempo de ejecución de cada kernel.

B. *Tiempos de ejecución con respecto al tamaño de la imagen*

El objetivo con esta evaluación es saber cuando tiempo le toma a cada kernel en promedio ejecutar el filtrado cuando el tamaño de la imagen aumenta. Para esto se evaluaron 5 imágenes con tamaños estándares y los resultados se pueden ver en la tabla III. Además en la figura 9 se puede observar la tendencia de cada kernel, cabe mencionar que el eje vertical esta en escala logarítmica para que las diferentes tendencias puedan ser observables.

Como era de esperar, se observa como cada kernel aumenta su tiempo de ejecución cuando el tamaño de la imagen aumenta. Otro aspecto a destacar es que las implementaciones en el CPU son las que mas rápido ejecutan el filtrado.

Cuadro III: Tiempos promedio de ejecución para cada kernel tomando en cuenta imágenes de diferente tamaño

Kernal	320x240	420x240	620x480	800x600	1280x720
LTI:lib	0.00244	0.00361	0.01012	0.01472	0.03543
Trivial CPU	0.01578	0.02106	0.06592	0.10515	0.19727
Trivial GPU	225.667	296.032	873.970	1410.422	2708.779
Separable	108.056	141.907	419.167	675.951	1298.293
Separable + MTH	0.03530	0.05136	0.10963	0.16019	0.29822
Sep+MTH+Sh Mem	0.02634	0.03925	0.07764	0.11203	0.20696

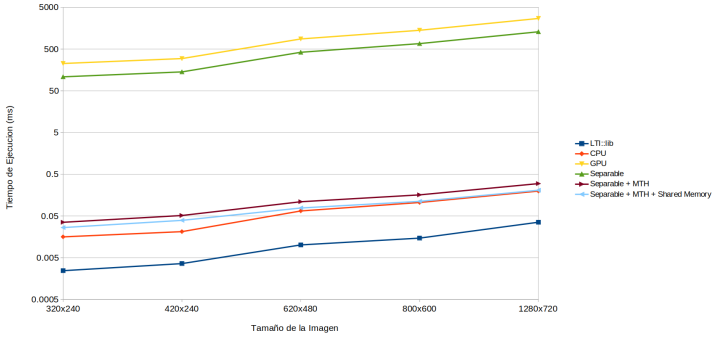

Figura 9: Tendencia del tiempo de ejecución cuando se aumenta el tamaño de la imagen

C. *Similitud de las imágenes de salida*

Como se explico en la sección de metodología para la evaluación de las diferencias no solamente se aplico el factor visual si no que también se aplico un efecto matemático para evaluar que tan diferentes eran las imágenes con respecto a su referencia. El método que ha sido seleccionado es el llamado *L2-norm* que se encarga de evaluar la diferencia entre dos vectores y cuyos resultados se resumen en la tabla IV. En dicha tabla se puede observar como el resultado de aplicar el L2-norm es cero cuando se analiza la diferencia entre el kernel trivial, GPU trivial y separable. Las diferencias empiezan a mostrarse en el Separable + MTH y Separable + MTH + Shared Memory.

Cuadro IV: Difencias promedio entre imagen resultado y la referencia (Kernel CPU)

Kernel	Tiempo de Ejecución (ms)
Lti::lib	$7.2e^{-2}$
GPU	0
Separable	0
Separable + MTH	$5.523e^{-3}$
Separable + MTH + Shared Memory	$1.336e^{-1}$

La figura 10 muestra el resultado de pasar la figura original de Lenna de la figura 8 por cada uno de los kernels. Cabe destacar que en general son bastante parecido, únicamente el kernel con shared memory produce resultados con desviación mas grande, aunque presenta una similitud razonable con el kernel LTI::lib

(a) Lenna con kernel Trivial GPU

(b) Lenna con kernel Trivial CPU

(c) Lenna con kernel Separable

(d) Lenna con kernel Separable + MTH

(e) Lenna con kernel Separable + MTH + Shared Memory

(f) Lenna con kernel LTI::lib

Figura 10: Imágenes resultado para cada kernel

VI. Conclusiones

Dados los resultados obtenidos la utilización del GPU para realizar tareas paralelas es una gran idea mas y cuando no se tiene un CPU poderoso que pueda hacer el trabajo. Además una implementación paralela en CPU necesita mucha infraestructura adyacente para poder correrse en diferentes hilos sin tener problemas,mientras que un GPU ofrece la infraestructura para correr aplicaciones paralelas en el hardware mismo. Esto reduciría enormemente el tiempo de desarrollo de aplicaciones reales que deben cumplir con apretados calendarios de desarrollo.

Es notable como agregar complejidad en el trabajo de paralelización disminuye el tiempo de ejecución, por ejemplo cuando se compara la implementación trivial con la implementación con el kernel separable + MTH + Shared memory donde se redujo el tiempo de ejecución cerca de 111670 veces.

La ganancia en cuanto a reducción de tiempo de ejecución también es notoria cuando se aumenta el tamaño de la imagen pues se obtuvieron resultados cercanos a los que se obtuvo con la ejecución del kernel en el CPU.

El resultado de las imágenes al ser comparadas con el kernel base indican que bajo un margen de tolerancia los resultados son bastante aceptables cuando se lleva al máximo nivel de optimización alcanzado en este trabajo que fue utilizando memoria compartida. Hacer as pruebas podrían ayudar a indicar donde esta el error en el resultado.

VII. Áreas de Mejora/Trabajos futuros

- Implementar Shuffle Instructions: Las intrucciones suffle son instrucciones especializadas que permiten a los threads intereactuar entre ellos sin necesidad de ir a memoria compartida, es decir es una intereaccion que se realiza a nivel de registros [5][11]. Por lo que la implementación en procesamiento de imágenes es una herramienta poderosa para reducir aun mas el tiempo de ejecución.
- Evaluar como mejorar el resultados de los kernels: evaluar por que los resultados de las imágenes cuando se utiliza multithreading esta dando diferente a la referencia. Una posible hipótesis es que hayan algunos problemas en el direccionamiento de algunos threads/bloques.

Referencias

[1] P. Porta, "Técnicas de filtrado." [Online]. Available: http://www.pauloporta.com/Fotografia/Artigos/filtros.htm

[2] S. M. Olesen and S. Lyder, "Applying 2d filters using gpu's and cuda," 2010. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.705.8599&rep=rep1&type=pdf

[3] J. Sanders and E. Kandrot, "Cuda by example: An introduction to general-purpose gpu programming," 2011. [Online]. Available: http://www.mat.unimi.it/users/sansotte/cuda/CUDA_by_Example.pdf

[4] V. Podlozhnyuk, "Image convolution with cuda," 2012. [Online]. Available: http://ecee.colorado.edu/~siewerts/extra/code/example_code_archive/a490dmis_code/CUDA/cuda_work/samples/3_Imaging/convolutionSeparable/doc/convolutionSeparable.pdf

[5] Y. Lin and V. Grover, "Using cuda warp-level primitives," 2018. [Online]. Available: https://devblogs.nvidia.com/using-cuda-warp-level-primitives/

[6] P. Alvarado *et al.*, "lti::maximumfilter¡t ¿class template reference," 2016. [Online]. Available: http://www.ie.tec.ac.cr/palvarado/ltilib-2/html/classlti_1_1maximumFilter.html
[7] P. Alvarado, "Filtros en el dominio espacial," 2015. [Online]. Available: http://www.ie.tec.ac.cr/palvarado/MSc/PAID/lec05.1.pdf
[8] NVIDIA, "Cuda separable convolution," 2012. [Online]. Available: http://ecee.colorado.edu/~siewerts/extra/code/example_code_archive/a490dmis_code/CUDA/cuda_work/samples/Samples.html
[9] Pseudonimo:Log0, "Differences between the l1-norm and the l2-norm (least absolute deviations and least squares)," 2013. [Online]. Available: http://www.chioka.in/differences-between-the-l1-norm-and-the-l2-norm-least-absolute-deviations-and-least-squares/
[10] W. Hotrakool, "l0-norm, l1-norm, l2-norm, ... , l-infinity norm," 2012. [Online]. Available: https://rorasa.wordpress.com/2012/05/13/l0-norm-l1-norm-l2-norm-l-infinity-norm/
[11] J. Luitjens, "Faster parallel reductions on kepler," 2014. [Online]. Available: https://devblogs.nvidia.com/faster-parallel-reductions-kepler/