Taller de computación de propósito general en unidades de procesamiento gráfico

Curso 2018

Laboratorio

Alumno:
Bruno Sena - C.I: 4.748.816-6

26/11/2017

Índice

1. Introducción

El proyecto consistió en la construcción de un programa capaz de multiplicar matrices con valores binarios y en punto flotante utilizando procesadores gráficos.

2. Análisis del Problema

El problema planteado en este Laboratorio consiste en la implementación de la operación de producto entre matrices (y matriz-vector), donde una de ellas contiene valores de tipo binario y la otra en punto flotante. El interés radica en construir algoritmos eficientes pues es de especial interés utilizar las unidades de procesamiento gráfico para ello ya que es un problema altamente paralelizable dada la naturaleza de independencia de los cálculos.

3. Diseño de la Solución

Para el diseño de la solución se planteó la utilización de la programación orientada a objetos, una clase matriz contiene los datos necesarios para que la GPU los consuma y pueda realizar la multiplicación correspondiente. Esto permite facilidad al momento de uso ya que los cálculos serán independientes. Existen dos clases de matrices, Binaria y Flotante.

4. Implementación

4.1. Bibliotecas

Se utilizó la biblioteca CuBlas para obtener un punto de referencia sobre los rendimientos reales de una aplicación de uso comercial y académico. Es de especial interés conocer si se logran mejores tiempos de ejecución (considerando los tiempos de trasnferencia de memoria) para realizar la comparación.

4.2. Algoritmos y técnicas implementadas

4.2.1. Multiplicación de matrices simple

Asumiendo que se trabajará con la operación $AxB = C$. Para multiplicar de forma sencilla, se lanza un hilo por cada elemento de la matriz de salida C. Dicho hilo recorrerá la fila que le

corresponda en la matriz A, leyendo los datos empaquetados una única vez y conservándolo en un registro. Además de la columna en la matriz B, multiplicando los elementos leídos y añadiéndolos al resultado que se encontrará almacenado en un registro.

De esta manera, en la matriz A los elementos serán leídos tantas veces como hayan columnas en B (considerando que una lectura en realidad corresponde a 32 o 64 elementos); de manera similar, los elementos de B serán leídos tantas veces como la cantidad de filas en la matriz A.

Sobre el acceso 'coalesced' a los datos, puede observarse que las lecturas en la matriz A cumplen el requerimiento necesario para la optimización, ya que el planificador puede unificar las transacciones de lectura de elementos debido a que hilos contiguos leerán elementos contiguos en el arreglo de bytes. Sin embargo, las lecturas en la matriz B se realizan por fila, entonces hilos contiguos realizarán lecturas que difieren en $sizeof(float) * ancho(B)$ bytes entre sí, lo que imposibilita el acceso 'coalesced'.

4.2.2. Multiplicación utilizando tiles de memoria compartida

Para la multiplicación de matices utilizando tiles de memoria compartida se divide conceptualmente la matriz C en tiles bidimensionales, donde cada bloque de hilos se encarga de calcular un tile del resultado y cada hilo un elemento del tile. Cada uno de estos bloques carga iterativamente un tile de la matriz A y otro de la matriz B en memoria compartida, realizando la multiplicación de sus elementos y almacenandolos en un registro, al igual que la versión anterior. Este algoritmo es una adaptación, que carga los valores empaquetados y flotantes en memoria compartida por lo que aprovecha el menor tamaño de los valores. Cabe destacar que el valor se "desempaqueta"únicamente al momento de realizar la multiplicación.

Existirán conflictos de bancos en las lecturas realizadas en el tile en cual se cargan los datos de la matriz A, ya que los hilos leerán elementos contiguos del tile; de esta forma, cada hilo de ambos half-warp acceden al mismo banco en el mismo momento, y es necesario serializar la lectura de dichos elementos afectando el rendimiento del programa.

Para solucionarlo se utilizará la técnica de padding, para evitar que los elementos que corresponden a una misma columna se ubiquen en el mismo banco (esto es, se distancien de a 32 elementos) se añadirá una columna auxiliar que no se utilizará. Por tanto, los elementos de la misma columna dejarán de encontrarse físicamente en el mismo banco.

4.2.3. Multiplicación utilizando tiling de registros

Dada la cantidad de máscaras utilizadas para el cálculo de los resultados no fue posible generar una versión con tiempos de ejecución aceptables para la técnica de register tiling (consiste en almacenar porciones de la matriz A en registros para luego leerlos tantas veces como sea necesario, leyendo de memoria global una única vez). Los registros necesarios exceden el valor disponible de 64, en cuyo caso se da el problema de Register Spilling (ver: `http://developer.download.nvidia.com/CUDA/training/register_spilling.pdf`)

4.2.4. Multiplicación matriz-vector

En un principio, dada la multiplicación $AxV = D$ donde A es una matriz binaria y V es un vector de flotantes, esta implementación se basó en la lectura (única) de memoria global de las filas de la matriz A multiplicándolos por los elementos del vector B. Estos últimos se releían tantas veces como el alto de la matriz A, por tanto se decidió almacenarlos en memoria compartida. La carga (y uso) de los datos se hace en segmentos, pues el vector puede superar el tamaño máximo almacenable en memoria compartida. Además, no es conveniente usar tamaños excesivos de memoria compartida pues impactarán en gran medida la ocupación de los hilos (ver: `https://devtalk.nvidia.com/default/topic/838543/maximum-shared-memory-size/`).

4.2.5. Implementación de la multiplicación

Las matrices (y eventualmente la matriz y el vector) están compuestas por tipos de datos dispares, no existe una definición de multiplicación para números binarios empaquetados y punto flotante. Por tanto, se implementaron dos opciones.

La primera es una simple sentencia condicional si el bit de la matriz A se encuentra encendido se añade al resultado el elemento de la matriz B que corresponda, en otro caso no se realizan operaciones (nota: esto implica que los hilos divergen en caso de existir valores que acceden a adicionar).

La segunda opción extra el valor del bit del paquete, utilizando una máscara (and) con el bit encendido correspondiente y luego un shift de tantos elementos como sea necesario para llevar el valor a la unidad, luego simplemente se realiza un cast del valor para convertirlo en flotante y se multiplica con el valor de la matriz B (donde B es tal que $AxB = C$).

5. Evaluación Experimental

Es necesario evaluar experimentalmente los tiempos de ejecución para distintos tamaños de matrices, bloques y grillas. A continuación se presentan los resultados obtenidos.

5.1. Hardware utilizado

Para el benchmark de la aplicación generada se utilizó el siguiente entorno:

Attribute	Value
CPU	
Name	Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz
Architecture	x64
Frequency	2.394 MHz
Number of Cores	8
Page Size	4.096
Total Physical Memory	16.266,00 MB
Available Physical Memory	10.421,00 MB
Misc	
Hybrid Graphics Enabled	False
Operating System	
Version Name	Windows 10 Home
Version Number	10.0.17134
Tools	
Nsight Version	5.5.0.18014
Nsight Edition	Standard
Visual Studio Version	14.0

Figura 1: Hardware y software utilizado (sin device)

Con la GPU listada a continuación:

Attribute	GeForce GTX 870M
Driver Version	398.11
Driver Model	WDDM
CUDA Device Index	0
GPU Family	GK104
Compute Capability	3.0
Number of SMs	7
Frame Buffer Physical Size (MB)	3072
Frame Buffer Bandwidth (GB/s)	120
Frame Buffer Bus Width (bits)	192
Frame Buffer Location	Dedicated
Graphics Clock (Mhz)	483
Memory Clock (Mhz)	2500
Processor Clock (Mhz)	967
RAM Type	GDDR5
Attached Monitors	0

Figura 2: Datos del device utilizado

5.2. Multiplicación de matrices

La siguiente tabla muestra los resultados experimentales sobre el producto de matrices. Se tomó una única medida pues se observó que de repetir el experimento el tiempo de transferencia disminuye en un 500 %, sin embargo se inicializó la biblioteca CuBlas antes de ejecutar cualquiera de las pruebas (con el motivo de que esta última no influyera en las pruebas. El máximo tamaño de matriz utilizable es de 4096x1024 elementos (en otro caso, el runtime corta la ejecución por excederse de la cantidad de threads)

Evaluaciones experimentales (Transferencia a GPU y ejecución) (ms)				
Algoritmo	512x512	1024x1024	2048x2048	4096x1024
Simple	522/3	384/26	414/161	X[1]
Shared Mem & pad	389/3	392/11	400/80	402/40
CuBlas	389/4	391/3	410/15	406/9

Como era de esperarse, el producto de matrices convencional tiene uno de los peores tiempos de ejecución en todas las instancias. En segundo lugar, no se observan grandes mejoras en la tasa de transferencia a la GPU al utilizar datos empaquetados o en punto flotante, difieren en algunos mili-segundos que no impactan en gran medida el tiempo de ejecución. En suma, los algoritmos que

utilizan memoria compartida y CuBlas tienen un rendimiento bastante similar lo que demuestra que el ahorro en el tiempo de transferencia se equipara con el costo de desempaquetar los datos.

Los siguientes experimentos refieren a micro-optimizaciones en el programa, el primero a cómo se desempaquetan los datos (como se vio anteriormente, utilizando una sentencia `if` o shifts y conversión a punto flotante). Los resultados fueron los siguientes:

Evaluaciones experimentales (ms)				
Algoritmo	512x512	1024x1024	2048x2048	4096x1024
Shift	2	12	98	50
If	3	11	80	40

Puede verse que la mejor alternativa (aunque provoque divergencias) es la de la sentencia `if`, probablemente debido a la cantidad de instrucciones necesarias para realizar la conversión a punto flotante del valor binario.

La siguiente optimización a probar tiene que ver con el "unrolling" del código, que no es más que la generación de instrucciones equivalentes a la ejecución de un ciclo `for` de largo constante (para precalcular valores de control utilizados e inyectarlos en el código en vez de evaluarlos en tiempo de ejecución). CUDA provee de una directiva muy útil que es `#pragma unroll`, que genera el código equivalente al ciclo `for` pero desarrollado. A continuación se ven los resultados:

Evaluaciones experimentales (ms)				
Algoritmo	512x512	1024x1024	2048x2048	4096x1024
Unrolled	1	10	80	40
Rolled	3	11	80	40

En este caso no se aprecia diferencia considerable entre las dos configuraciones, es necesario recordar que a mayor tamaño de kernel (es decir, cantidad de instrucciones) más lecturas en memoria para realizar el fetch de las instrucciones. Sin embargo, al no utilizar el unrolling es necesario calcular más resultados en la ejecución.

5.3. Multiplicación de matriz-vector

En este caso simplemente se ejecutó el algoritmo aunque no se tiene un marco de referencia de los tiempos establecidos. Sí se logró una mejora sustancial en los resultados al almacenar el vector de datos en memoria compartida, pues se reducen considerablemente los accesos a memoria global.

Evaluaciones experimentales (ms)				
Algoritmo	512-512	1024-1024	2048-2048	4096-4096
Vector compartido	0	1	1	0

Los tiempos obtenidos son casi nulos, esto se debe a que la multiplicación matriz-vector se puede realizar en tiempo lineal (en el tamaño de la matriz) ya que cada dato se lee una única vez.

6. Conclusiones

La utilización de unidades de procesamiento gráfico en propósito general puede generar resultados extremadamente buenos para los problemas adecuados -problemas paralelizables con independencia de datos-, como la multiplicación de matrices trabajada en este informe, sin embargo, para obtener el mejor resultado posible es necesario balancear los costos de lectura a memoria global, las barreras de sincronización y la carga de trabajo por hilo eliminando la divergencia. Esto último debe hacerse considerando la arquitectura de trabajo primordialmente. En este caso se utilizó la arquitectura Kepler, que mantiene la frecuencia de los relojes en valores más bajos para aumentar la eficiencia energética. Esto perjudica a aquellos algoritmos que se basan en una carga mayor a los hilos y favorece a aquellos algoritmos que intentan reducir la cantidad de trabajo por hilo.

Finalmente, puede observarse que el uso de datos empaquetados puede llegar a optimizar el tiempo necesario para ejecutar la multiplicación (recordando el trabajo realizado anteriormente en el curso de GPGPU, en el que se obtenían tiempos de ejecución excesivamente alejados de los de la biblioteca CuBlas). Por tanto, con un mayor análisis de bajo nivel podrían desarrollarse tiempos aún mejores.

Referencias

[1] CUDA Programming Guide - Nvidia - 2017

[2] Diapositivas de clase - EVA GPGPU

[3] Register Spilling - Nvidia `http://developer.download.nvidia.com/CUDA/training/register_spilling.pdf`