Apuntes programación CUDA

Mauricio Vanzulli

June 1, 2021

1 Clase de CUDA

Características del device:

- Es un coprocesador de la CPU
- Posee una memoria DRAM
- Ejecuta muchos threads en paralelo.
- Cada mutli-procesador procesa un bloque con un programa único (kernel) en muchos hilos. Cada CUDA-core procesa muchos hilos, uno a la vez.
- Este paradigma de programación recibe la sigla SPMT (single program multiple threads).
- Cada kernel ejecuta un array de hilos, es importante el identificador de hilo respecto al dato al que se le quiere ejecutar el kernel.

Algoritmo básico de programación

1. Instrucciones en el host.
2. Enviar los datos al device.
3. Procesa en GPU
4. Recuperar los datos de la GPU.
5. Continuar procesamiento en el host.

Existe determinada jerarquía de threads. Una grilla en 3D agrupa un conjunto de bloques y dentro de cada bloque se tienen múltiples hilos también en 3D. No se puede asumir a priori que el bloque 1 se ejecute antes del 2.

Las funciones en CUDA

Dominio de funciones	*Ejecuta en:*	*Se invoca desde:*
__device__	device	device
__global__	device	host
__host__	host	hostt

Función	*Variables de entrada:*	*Proposito:*
dim3 DimGrid	(Cant_Bloq_x,Cant_Bloq_y,Cant_Bloq_z)	Crea una grilla de esas dimensio de bloques
dim3 DimBlock	(Cant_Hilos_x, Cant_Hilos_y, Cant_Hilos_z)	Crea las dimensiones de los threads en cada bloque
KernelFunc <<<DimGrid,DimBlock,SharedMemBytes>>>(...)	Input Kernel, grilla, y memoria compartida	Ejecuta el kernel en device

Se disponen de distintos espacios de memoria:

- __global__ : memoria global en el host
- __device__ : memoria global en device
- __shared__ : reside en la memoria compartida host-device (se usa para allocamiento dinámico)
- __constant__ reside en momoria cosntante del device (allocamiento estático)

Localizadores de hilos y bloques:

- threadIdx :Ubicación de ese thread y se accede con .x .y .z
- blockIdx :Ubicación del bloque en el que me encuentro y se acccede con .x .y .z
- blockDim : Tamaño del bloque (en cantidad de hilos) y se accede con .x .y .z
- gridDim :Tamaño de la grilla (en cantidad de blques)y se accede con .x .y .z

Funciones intrínsecas al GPU

- cudaDeviceSynchronize :Sincroniza todos hilos en el device y se ejecuta desde el host.
- __syncthreads :Permite sincronizar los threads de un mismo bloque.
- cudaClock y otro miden tiempos

Reservar memoria en la tarjeta y transferir datos es fundamental para comunicar las datos y sus procesamientos, esto es bastante costoso en términos computacionales. Me interesa comunicar host-cpu (se ejecutan desde el CPU):

- cudaMalloc(Puntero,Tamaño de memoria) Reserva memoria en la global memory de la GPU

- cudaFree (Puntero) libera el espacio reservado
- cudaMemcpy (Puntero de destino, puntero origen,numero de bytes a copiar, Tipo de transferencia) . Los diferentes tipos de transferencia son: Host 2 Device (cudaMemcpyHostToDevice), Device 2 Host(cudaMemcpyDeviceToHost). Además están (Host 2 Host) y Device 2 Device.
- cudaMemSet (Variable a iniciar,0 o 1, sizevector): Setea el inicio del vector en el device.

La clave es la relación entre los identificadores y los particionamientos de datos

Si en el kernel el procesamiento no depende del thread puede haber problemas de condición de carrera... para solucionar esto se usa atomicadd Esto sucede cuando mas de un hilo deben escribir a la misma entrada de memoria golbal. Osea dentro de un mismo warp múltiples hilos deben escribir en el mismo lugar, ¿cuál escribe primero?, hace falta ordenarlos serialmente. Este tipo de problemas se llama racecondition y CUDA escribe unicamente un hilo aleatorio.

2 Clase de CUDA

El acceso a memoria global es donde se encuentra el cuello de botella, el proceso más costos en términos computacionales la transferencia de datos de la CPU al device y luego del device a CPU, es importante realizar esta transferencia de forma eficiente.

Memoria Coalesced Acceso "coalesced" a memoria global: Consiste en fusionar los accesos a memorias de un warp (conjunto de hilos) para optimizar la petición a memoria. Si cada hilo necesita un dato contiguo a otro en memoria en vez de realizar dos accesos, los une en uno solo.

- El acceso a memoria global es por segmentos incluso cuando se quiere leer una palabra. A veces esos datos no son útiles y se desperdicia el ancho de banda. También sucede en CPU pero el cache y el preload vulneran estos problemas intrínsecamente."Cuando se solicita una dirección del segmento me devuelve todo el segmento"
- Los segmentos están alineados en múltiplos de 128 bytes. "Me traigo trozos o bloques de 128 bytes". Además estos segmentos están fijos.
- Es necesario utilizar Structs of Array. Osea si quiero guardar múltiples datos del mismo objeto guardo los arrays de eses datos para los múltiples entes. Como los arrays son secuenciales se puede utilizar el coalessed, contrario a lo que sucede con Array of Structs

En la memoria global queremos que cada lectura utiliza la mayor cantidad de datos en cada ejecución. En el caso de memoria compartida es deseable que cada bloque use la mayor cantidad de memoria

Memoria Compartida Un mismo bloque dispone de una memoria compartida en el multi-procesador que se ejecuta. Esto se tiene que especificar en el kernel y cuando termine de ejecutar ese bloque hay que copiarlos a la memoria global ya que sino mueren con su bloque. Esta memoria se usa como una especie de cache, permite evitar accesos no coalesced a la memoria global. Se copia de forma coalessed de la global y luego se accede de forma desordenada en la memoria compartida.

Cuando se invoca al kernel hay que hacer referencia a que cantidad de memoria compartida se quiere reservar en memoria Función¡¡Dim Grid,Dim BLoc, Tamaño de memoria compartida¿¿(output, input)

Puede existir un problema cuando en un ciclo de reloj los hilos no pegan en bancos distintos puede suceder que: dos hilos traten del leer el mismo dato de banco tenemos un bank conflict. La clave es que el indice de acceso a banco sea con modo impar en módulo 32(cantidad de bytes por banco). Resumen:

- No hay confitico y el acceso es rápido cundo:
  1. Todos los hilos del warp acceden a diferentes bancos.
  2. Todos los hilos del warp leen la misma palabra de un banco.
- El acceso es ineficiente cuando:
  1. Varios hilos del warp acceden a parlabas diferentes del mismo banco.
  2. Se produce un conflicto y se debe serializar el acceso.
  3. Se requieren tantos ciclos como el numero máximo de accesos al mismo banco.

En general cuando se utiliza memoria compartida sebe utilizarse stncthreads() para garantizar que todos los hilos completaron determinada operación. En general cuando se quiere bajar a memoria global determinado algoritmo es importante implementar el syncthreads que garantiza que la transferencia se realiza cuanto dos los hilos del bloque finalizaron sus acciones.

Transferencias a memoria Para hacer una transferencia es necesario que la memoria sea no paginable (estable y no tomado el control por el SO). La memoria paginable hace funcionar la memoria virtual porque cuando se ejecuta una nueva aplicación el SO copia esos datos de memoria paginable al disco y luego vuelve a cargarlas. La transferencia a GPU debe ser no paginable ya que la CPU no debe tocarla. Cuando se ejecuta cudaMemCpy() se copia de paginable a no paginable y luego a la memoria global de la GPU. Para eso se usa cudaHostAlloc, pero este bloque la memoria no paginable.

cudaHostAlloc reserva memoria no paginable para nuestra aplicación, no obstante los recursos no paginables son limitados.

Se puede separar el plano físico con el estado lógico. Osea hacer una única transferencia física y luego particionar esos datos con la lógica.

Patrones de cómputo Este es una representación gráfica de como fluctua una determinada variable. La resolución se puede realizar mediante los siguientes mecanismos:

1. Acceso coalesced
2. Operaciones atómicas
3. Privatization

Cada hilo tiene su propio registro donde se guardan las variables que ejecuta cada hilo, esto se localiza en cada MP nivel de hardware. Si diferentes hilos tienen que escribir en el mismo dato hay problemas de condición de carrera (depende de cual realiza primero la instrucción el resultado final). Se puede asegurar la ejecución como un ploque para que no sucedan lecturas o escrituras en simultano por diferentes bloques que trabajan con el mismo dato.

attomicAdd La función *attomicAdd* condensa una operación de escritura, modificación y lectura y hasta que ese hilo no finalice esa dirección en memoria el resto no puden alterar esa dirección. Esta recibe un puntero a int y un valor y almacena el valor en int, devolviendo el valor leído. Esto tiene un problema ya que toma desmasiado tiempo de latencia entre una opreación y otra, esto no ocurre en la memoria local. Para esto se usa privatization:

El objetivo es que cada bloque tenga su propia memoria, soluciona cuando cada bloque intenta escribir en momia global con una opreación de attomic. Esto causa demasiado overhead. Cada bloque construye su propia salida en la memoria local y luego los bloques lo escriben en memoria global con attomicAdd.

Reduce Si se quiere reducir por ejemplo el módulo, osea esto ocurre cuando los datos de entradas son enormes pero la salidas son pocas. Por ejemplo suma de un vector.

Consiste realizar resultados intermedios en GPU en memoria compartida para luego compactar todo en una operación en memoria global.

Para reservar memoria compartida se pude hacer con $\_\_shared\_\_$ tipo y la variable y cantidad de bytes. O $extern\_\_shared\_\_$ que cuando se llama al kernel se indica que tamaño de memoria que deseamos reservar para su ejecución.

El *syncthreads* sincorniza y garatniza que todos los hilos hayan terminado esas operaciones antes de avanzar en el códiog. Un error ocurre si se ejecuta el *syncthreads* dentro de una codnción condicional porque no va a entrar y va a quedar esperando que todos terminen y esto nunca sucede.

Stencil Se utiliza cuando se busca actualizar un valor sobre determinada grilla, la traducción textual sería "plantilla". Suele aplicarse cuando existe un determinado proyecto iterativo en que se debe modificar el valor del pixel de acuerdo a un determinado algoritmo constante.Ej resolución de problemas en derivadas parciales, Gauss-Seidel, procesamiento de imágenes. Este patrón se encuentra en auge ya que es usual en redes neuronales, también en convoluciones (productos compuestos).

La iniciativa de optimización esta ligada a la reutilización de datos desde los diferentes *threads* (ej: ir a buscar a la MemGlob el valor de la máscara). Se puede copiar los datos a memoria local utilizando *tiling*.

Hay que estudiar cuantos elementos necesita cada bloque, para el caso de la mascara eso se calcula para el primer y último pixel del tile. El tile de entrada puede ser diferente al de salida, ej si necesito 128 elementos de salida necesito 132 elementos de entrada. Eso determina el tamaño del bloque según las siguientes alternativas:

1. Que los threads recordan el tile de salida, osea para cada hilo se impacta en el valor de salida. Consecuentemente como el tile de entrada es más grande que el de salida algunos hilos van tener que cargar en memoria más de un elemento.

2. Antagónico al anterior, osea tengo tantos hilos como las dimensiones del tile de entrada. En este caso hay hilos que no participan en los elementos de salida

Para esta estrategia de computo se puede utilizar la memoria constante que utiliza el cache de la GPU, otra opción es realizar una carga local en la shared memory de la mascara.

scan Es un algoritmo comumnente utilizado y analizado en la computación paralela que tiene la propiedad de asociativa y hace una especie de composición en los contiguos. Es un método iterativo y secuencial, en determinada posición se aplica una función entre el acumulado de salida y el valor de esa posición de entrada.

Al programarlo de una manera muy lineal puede producir un balance de carga ya que el último hilo tiene un trabajo abismalmente mayor que los primeros.

2.1 Clase de consulta práctico 4

1. La memoria global es del orden de Giga y la local es de Kilo. Las memorias más rápidas son mas caras y es más chcica. Es análogo al cache con la CPU, las memorias más veloces van dentro del procesador y las mas lentas y grandes fuera.

2. Dentro del multi-procesador estan los CudaCores los registros y la memoria local. Las arqutecturas tienen una eficiente integrada donde lo ideal es que para disminuir las latencias los diferentes warps se van ejecutando mientras uno lee dentro del mismo MP. Por eso a veces rinde mas tener bloques más chicos porque te entran más warps por MP y la idea es que siempre los MP esten trabajo, esto se ajusta expreimentalmente.

3. La memoria compartida se organiza en 32 bancos de 4 bytes, porque los warps se componene de 32 threads. Lo ideal es que cada hilo lea en un banco distitno así pueden hacerlo en paralelo. Cuando un warp lea 32 pixeles contiguos en la imagen (en filas) eso se va a ir a buscar a bancos contiguos en paraelelo (En un ciclo de reloj todos los threads van a obtener su dato porque la memoria compartida tiene una latencia de 34 byts por ciclo de reloj por segund "Revisar") El problema es cuando el thread 0 y el 32 acceden al mimsmo banco entonces se va a serializar el

acceso esto se llama conflicto de bancos (dos hilos del mismo warp quieren acceder al mismo dato entonces el hardware los ordena serial). " Como es un chip solo dos solicitaciones dentro de un mismo warp tienen que hacerse de forma serial". La excepción es cuando esos threads necesitan el mismo dato, ahi se hace un broadcast y no hay conflicto.

4. Dos datos contiguos van a datos contiguos