Taller de programación en GPU
Tidball Binz, Inti Maria
Entrega 11 noviembre 2021

Optimización de reducción de un vector en CUDA

Se resuelve con programación en CUDA, la suma de los elementos de un arreglo ***V*** de ***N*** elementos de tipo float, luego optimizando la solución.

Reducción original:

La solución original ejecuta una reducción en el kernel utilizando un patrón de acceso coalescente en memoria global. Cada hilo trabaja de manera independiente sobre su posición en el arreglo, aunque la mitad de los hilos para cada iteración estén ociosos. Se van sumando los valores de la primera mitad del arreglo en memoria global con los valores de la segunda mitad, reduciendo el tamaño del arreglo por la mitad en cada iteración. En cada siguiente iteración, se usa la primera mitad del arreglo donde quedaron los resultados de la anterior reducción, se vuelve a sumar a su mitad, y así continuamente hasta reducir el arreglo a una sola posición.

Usamos un for loop para llamar al kernel, se define una variable “distancia” que equivale a la mitad de la dimensión del arreglo, y que se va reduciendo por la mitad en cada iteración del loop. Se usa la función CudaDeviceSynchronize() como barrera de sincronización luego de cada llamado del kernel. El for externo hace log2(n) iteraciones.

En el kernel, se asigna en la variable ‘global_id’ la posición del hilo que está ejecutando el kernel (con la operación numérica global_id = blockIdx.x * blockDim.x + threadIdx.x).
La mitad de los hilos suma dos valores en su propia posición: Un if separa los hilos con id dentro del valor distancia; estos suman al valor en su posición global_data[‘global_id’] con el valor en la posición global+data[‘global_id’ + distancia]. Se guarda la suma in place, y de este modo quedan todos los resultados en espacios contiguos de memoria, ocupando los resultados la mitad del arreglo en cada iteración.

El kernel se llama hasta que la distancia se reduzca a 1. Se reduce el total de la suma a una sola posición (equivalente a global_data[0]), en la cual está el resultado.

Código del kernel:

```
__global__ void reduccion_kernel_cuda(basetype *const global_data, const int n, const int
distancia){
 unsigned long int global_id = blockIdx.x * blockDim.x + threadIdx.x;
 if (global_id < distancia){
   global_data[global_id] = global_data[global_id] + global_data[global_id + distancia];
 }
}
```

Código del for externo llamador del kernel:

```
int distancia = n/2;
unsigned int i;
unsigned int hasta = ceil(log(n)/log(2));
for (i=0; i < hasta; i++) {
  suma_kernel_cuda<<<dimGrid, dimBlock>>>(cV, n, distancia);
  cudaDeviceSynchronize();
  distancia = ceil(distancia / 2);
}
```

Reducción optimizada

Se aplican algunas optimizaciones con respecto a la versión previa. Hacemos una optimización en el uso de memoria, usamos coalescencia como la solución original pero haciendo uso de memoria compartida. En términos de ejecución de los hilos reducimos la divergencia y evitamos conflictos de banco en la programación del loop interno al kernel. Además, se hace una primera reducción en la copia de memoria global a compartida reduciendo el número de hilos ociosos en la primera iteración, y por último hacemos un unrolling del último warp de cada bloque.

En el kernel se pueden visualizar algunas modificaciones con respecto a la versión no optimizada. En primer lugar podemos ver que se lee de memoria global de manera coalescente al guardar los valores en memoria compartida. El tamaño de cada arreglo shared se define usando una declaración estática de arreglo por fuera del kernel, el cual va a ser el mismo tamaño que el tamaño del bloque pasado por parámetro y no va a cambiar en las iteraciones consecuentes. Se calcula el tamaño del mismo multiplicado el tamaño del bloque por el tamaño de tipo de dato, en este caso usamos float ya que está definido en el enunciado. Definimos este valor en la variable **numBlockBytes**, y lo pasamos como tercer parámetro al kernel. Los hilos de cada bloque van a estar trabajando sobre su posición en el arreglo shared. La dimensión del arreglo en memoria compartida se lee automáticamente desde el kernel al declarar la variable arreglo utilizando el sintaxis de CUDA **extern __shared__ float shared_data[].**

Se llama al kernel desde un for loop externo. En la llamada al kernel, en cada iteración reducimos el número de bloques a procesar, y se escriben los resultados de cada bloque de manera coalescente en las primeras posiciones del arreglo global. Por esto mismo reducimos la dimensión del arreglo en memoria global en cada iteración ya que solo necesitamos procesar los valores vigentes. Para hacer esto se modifica la variable dimGrid que se envía como primer parámetro al kernel, que define el tamaño total del grid (el número de bloques) con el cual vamos a trabajar. Se inicializa el índice i del for con el tamaño del arreglo total, que fue pasado por parámetro en la línea de comandos. En cada iteración, asignamos en dimGrid el valor del índice dividido por el tamaño de bloque para reducir la cantidad de bloques. También se reduce el índice dividido el tamaño de bloque, y se itera hasta que el número de bloques restantes es menor que min, variable asignada con el tamaño de un bloque. Se itera mientras el tamaño del arreglo en memoria global se pueda seguir dividiendo por tamaño de bloque y hasta llegar a min. Enviamos el valor i al kernel para indicar el tamaño actual del arreglo.

El valor de la variable min definirá el final del loop, en cuan momento el tamaño del arreglo reducido en memoria global sea igual o menor que el tamaño del bloque. Este arreglo se termina de reducir en la CPU. De este modo se maximiza el uso de la GPU para reducir los primeros N -1 bloques. De este modo, si tenemos bloques de 512 hilos en el bloque y la última iteración restan 16 posiciones, no ”gastamos” 504 hilos del bloque ejecutando solo 8 en el kernel. También, de este modo optimizamos el uso de la GPU ya que la GPU optimiza operaciones para Ns grandes pero CPU es más performante en números pequeños. Por ejemplo, si N es 1024, el tiempo en CPU es de

0.000019 mientras que el mejor tiempo en GPU optimizado sin tiempos de transferencia H2D o D2H es 0.000026. Al subir el tamaño de N, se invierte esta relación y la performance en GPU es exponencialmente mejor con mayores valores de N.

Entrando en el kernel, luego de la declaración del arreglo en memoria compartida, se hace la asignación de la variable global_id, se leen los datos de la memoria global de manera coalescente y cada hilo se lleva su valor a la memoria compartida. Cada bloque copia una parte del arreglo global y luego se trabaja independientemente por bloque.
En esta primera carga se aplica una optimización donde se hace una primera reducción del arreglo en la carga de los valores. Cada hilo de cada bloque se guarda su global_id en blockIdx.x * (blockDim.x * 2) + threadIdx.x (reduciendo el número de bloques a la mitad), luego se copia desde el arreglo global el valor en global_data[‘global_id’] sumándole el valor en global_data[global_id + blockDim.x] de esta manera haciendo la primera reducción en la carga. Se controla que se sumen los valores mientras que el global_id sea menor que la dimensión actual del arreglo en memoria global. Este valor lo copia al arreglo en memoria en su posición de hilo en ese bloque (shared_data[threadId.x]), haciendo así una primera reducción y evitando hilos ociosos en el kernel en la primera iteración.

El arreglo en memoria compartida es visible y editable para cada hilo de su bloque. Luego de copiar los datos, se debe usar una barrera para todo el bloque usando la función __syncthreads() para asegurarse que antes de proceder a la reducción todos los hilos del bloque hayan copiado su valor al arreglo en memoria compartida.
Se utiliza un for loop en el kernel que itera por el arreglo shared_data. El for es un loop reverso que evita la divergencia por warp, evitando conflictos de bancos de memoria, y maximizando el paralelismo e usando indexación con el id del hilo para asegurarse sequential addressing. Se usa el bitwise operator >> sobre la variable ‘distancia’ para correr sus bits a la derecha, comenzando desde blockDim.x/2, y reduciendo el valor en potencia de 2.El uso de un loop reverso nos permite asegurar que los hilos de un warp vayan por la misma rama del if interno. El loop itera log2(blockDim.x/2) veces (equivalente a dividir distancia en 2 en cada iteración) hasta el último warp (distancia > 32) que se procesa con un unrolling. Usar una operación bitwise que es más eficiente que expresiones de tipo división (y mod) en el kernel, y es una optimización de tipo ‘mezcla’. Ante cada iteración, se debe llamar la función __syncthreads() para asegurarse que cada hilo haya terminado su trabajo de sumar su valor al valor en su posición más la distancia.

Una vez que llegamos a distancia = 32 en el loop, podemos aplicar el unrolling del último warp para cada bloque. Toma en cuenta que llegando al último warp hay varios hilos que no están activos. El unrolling ahorra instruction bottlenecking por la carga y el costo de las operaciones aritméticas y de asignación que son parte del loop for. Todos los warps ejecutan el for, por lo tanto se ahorra trabajo para todos los warps usando este método. No se necesita syncthreads dentro del mismo warp ya que todos los hilos están en sync.

Al final de esta operación, se hace una escritura coalescente sobre memoria global. El hilo con threadId.x == 0 en el bloque copia el valor en su posición threadId.x en memoria compartida al

arreglo global usando el id de su bloque (global_data[blockId.x]), lo cual garantiza que los resultados de cada bloque queden contiguos en memoria global. En este if es imposible evitar la divergencia.

A futuro los tiempos de transferencia podrían ser optimizados utilizando pinned memory y streams para mejorar los altos tiempos de transferencia H2D. Al ser reducción la transferencia D2H no es costosa.

Código del kernel & función auxiliar (unrolling):

```
__device__ void unrolling_warp(volatile basetype* shared_data, int id) {
 shared_data[id] += shared_data[id + 32];
 shared_data[id] += shared_data[id + 16];
 shared_data[id] += shared_data[id + 8];
 shared_data[id] += shared_data[id + 4];
 shared_data[id] += shared_data[id + 2];
 shared_data[id] += shared_data[id + 1];
 }
__global__ void reduccion_kernel_cuda(basetype *const global_data, unsigned int n){
 extern __shared__ basetype shared_data[];
 unsigned long int global_id = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
 shared_data[threadIdx.x] = 0;
 if (global_id < n){
   shared_data[threadIdx.x] = global_data[global_id] + global_data[global_id + blockDim.x];
 }
__syncthreads();
for (unsigned int distancia = blockDim.x/2; distancia > 32; distancia >>= 1){
 if (threadIdx.x < distancia){
   shared_data[threadIdx.x] += shared_data[threadIdx.x + distancia];
 }
 __syncthreads();
}
if (threadIdx.x < 32) unrolling_warp(shared_data, threadIdx.x);
if (threadIdx.x == 0){
 global_data[blockIdx.x] = shared_data[0];
 }
}
```

Código del for externo llamador del kernel:

```
unsigned int numblockBytes = blk_size * sizeof(basetype);
unsigned int min = dimBlock.x;
unsigned int i;
for (i = n; i > min; i = i/dimBlock.x) {
 dim3 dimGrid((i + dimBlock.x - 1) / dimBlock.x);
 reduccion_kernel_cuda<<<dimGrid, dimBlock, numblockBytes>>>(cV, i);
 cudaDeviceSynchronize();
}
```

Taller de programación en GPU
Tidball Binz, Inti Maria
Entrega 11 noviembre 2021

Resultados / metricas

En tabla 1 se miden los tiempos del algoritmo optimizado (en azul) y no optimizado (en rojo), el cual es una suma de los tiempos de copia H2D, de ejecución y copia D2H (se pueden ver con máyor detalle acá). En la última línea se ven los tiempos de CPU. En la optimización, el speedup contra CPU es mayor para tamaños más grandes de N. En la arquitectura Turing, el N más grande posible de probar fue de 2^{29} o N=536870912. El tiempo de transferencia H2D es el más costoso, lo cual indica que se podría optimizar más el tiempo total utilizando streams. Están destacadas las celdas en las cuales son los mejores tiempos por bloque para cada tamaño N. Estos tiempos fueron utilizados para calcular los speedups vistos en la tabla 2.

tabla 1

Arquitectura: Nvidia GeForce RTX 2070 - Arq Turing					
	Tamaño del arreglo (N)				
Hilos por bloque	2^{23} 8388608	2^{25} 33554432	2^{27} 134217728	2^{28} 268435456	2^{29} 536870912
HxB 128	0.011105	0.043819	0.178137	0.34426	0.685374
	0.009124**2**	**0.03569**	**0.141439**	0.282624	0.565228
HxB 256	**0.010307**	**0.040563**	**0.162308**	0.318193	0.639996
	0.009119	0.03576	0.142541	**0.28213**	**0.563777**
HxB 512	0.010372	0.040883	0.163727	**0.318165**	**0.638518**
	0.009155	0.035906	0.143522	0.285699	0.575751
HxB 1024	0.010446	0.041325	0.166503	0.320259	0.648314
	0.009158	0.036355	0.143748	0.286988	0.571674
CPU	0.021212	0.084718	0.338814	0.67711	1.354676

tabla 2

N	CPU	mt GPU	mt GPU optimizado	speedup GPU/GPU opt	speedup CPU/GPU sin optimizar	speedup CPU/GPU optimizado
8388608	0.021212	0.010307	0.009119	1.130277443	2.058018822	2.326132251
33554432	0.084718	0.040563	0.03569	1.136536845	2.088553608	2.373718128
134217728	0.338814	0.162308	0.141439	1.147547706	2.087475664	2.395477909
268435456	0.67711	0.318165	0.28213	1.127724808	2.128172489	2.399992911
536870912	1.354676	0.638518	0.563777	1.132571921	2.121594066	2.402857868

En la tabla 3 y el correspondiente gráfico en figura 1, se puede apreciar el speedup en GPU si solamente tomamos en cuenta el tiempo de ejecución en la GPU sin considerar los tiempos de copia H2D y D2H. De esta manera podemos apreciar con más detalle el nivel de performance que tiene la GPU por sobre la CPU. Lo óptimo en conclusión, sería poder mejorar los tiempos de copia, los cuales se pueden mejorar usando pinned memory y streams.

tabla 3

tamaño de arreglo	CPU	GPU optimizado	speedup
8388608	0.021212	0.000393	53.97455471
33554432	0.084718	0.000937	90.41408751
134217728	0.338814	0.003264	103.8033088
268435456	0.67711	0.005774	117.2687911
536870912	1.354676	0.011381	119.0296108

figura 1