ECAR2012, Foundations track
Curso de CUDA, parte 3

Carlos Bederián, Nicolás Wolovick
{bc,nicolasw}@famaf.unc.edu.ar
FaMAF - Universidad Nacional de Córdoba

Objetivos

- Utilizar primitivas de sincronización de threads.
- Uso de operaciones atómicas sobre memoria global y compartida.
- Hacer que un bloque de threads coopere a través de la memoria compartida.
- Aplicar thread coarsening para mejorar la performance de un kernel.
- Comprender cómo los problemas de reducción se implementan eficientemente en GPU.

Introducción

Los algoritmos que hacen *scatter* o *gather* se traducen de manera directa al modelo de programación de GPUs, pero para otros problemas comunes la conversión no es directa o no consiguen buena performance. Una de esta familias de problemas son los algoritmos de reducción, donde dado un operador asociativo y conmutativo binario $\oplus$ y una secuencia $x_0..x_n$ de elementos queremos calcular $r = ((x_0 \oplus x_1) \oplus ...) \oplus x_n$. Entre los usos más comunes de reducción se encuentran la sumatoria, productoria, mínimo y máximo de un vector.

La implementación secuencial de la reducción es trivial, pero al intentar paralelizar el problema a una cantidad arbitraria de hilos nos encontramos con varios problemas para mantener un nivel de performance aceptable.

El problema

Para ilustrar el problema, escribamos un kernel donde cada thread le suma 1 a un contador en memoria global que originalmente tiene un 0:

```
#include <cuda.h>
#include <cutil_inline.h>
#include <stdio.h>

__global__ void add_one(uint * sum) {
  *sum = *sum + 1;
}

int main() {
  uint host_sum, *dev_sum;
  cutilSafeCall(cudaMalloc((void **) &dev_sum, sizeof(uint)));
  cutilSafeCall(cudaMemset(dev_sum, 0, sizeof(uint)));
  dim3 block(1024), grid(1024); // 1024x1024 sumas = 1048576
  add_one<<<grid, block>>>(dev_sum);
  cutilSafeCall(cudaMemcpy(&host_sum, dev_sum, sizeof(uint), cudaMemcpyDefault));
  printf("%u\n", host_sum);
```

```
  return 0;
}
```

Copie, compile y ejecute el programa. ¿Qué sucede y por qué?

Operaciones atómicas sobre memoria global

Una forma sencilla de solucionar el problema es hacer que la secuencia de operaciones de lectura, suma y escritura sobre el contador compartido se ejecuten de manera atómica, es decir, sean vistos por el resto de los hilos como si hubieran ocurrido de manera ininterrumpible. Para esto CUDA ofrece una gama de funciones que realizan operaciones atómicas, que son de la forma `atomicOp(T * compartido, T valor)` donde `T` es un tipo primitivo (`int, float...`), de modo que se ejecute `*compartido = *compartido` $\oplus$ `valor` de manera atómica. Entonces nuestro kernel debería ser de la forma:

```
__global__ void add_one(uint * sum) {
    atomicAdd(sum, 1);
}
```

Corrija el programa anterior y verifique que funcione correctamente.

Operaciones atómicas sobre memoria compartida

La solución anterior corre más lento que un código secuencial. Esto se debe a que cada thread del kernel `add_one` suma su valor de manera atómica al contador en memoria global, provocando que todos los threads corran de manera serial.

Mitigaremos este problema parcialmente, distribuyendo la contención hacia cada bloque de threads. Declararemos una variable en memoria compartida, sobre el que realizaremos la reducción de todos los valores del bloque de threads y luego reduciremos en el contador global una sola vez por bloque de manera atómica. Para esto necesitamos:

1. Hacer que un thread[1] inicialice el contador al valor nulo del operador de reducción antes de que cualquier thread del bloque sume su valor. El resto de los threads del bloque debe esperar a que la inicialización se complete antes de operar o se perderán sus actualizaciones.
2. Los threads operan atómicamente sobre la variable compartida entre todos los hilos del bloque.
3. Se espera a que todo el bloque termine, y un hilo del bloque hace una operación atómica entre el resultado parcial del bloque y el contador global.

Ahora utilice este método, tomando el programa del directorio `norm`, que normaliza los colores de una imagen de escala de grises, escribiendo los kernels de reducción que calculan el máximo y mínimo color, y luego corra el kernel que aplica la normalización de los colores.

Reducción en árbol

Una familia de algoritmos de reducción que son más amigables para paralelizar son los que reducen en árbol. Si tomamos un vector de n elementos y aplicamos el operador de reducción entre pares disjuntos de este vector (que se puede hacer en paralelo), entonces nos quedan $n/2$ elementos sobre los que operar y podemos repetir esto hasta quedarnos con un sólo elemento. Luego podemos ver a esta familia de algoritmos como árboles binarios que tienen como raíz el resultado, como hojas el conjunto de datos original, y como nodos los resultados parciales de aplicar el operador a dos elementos del nivel inferior.

A diferencia de la reducción utilizando operaciones atómicas, dada una cantidad infinita de procesadores donde aplicar el operador de reducción en paralelo (cosa que no dista mucho de la situación en una GPU), entonces necesitamos $log_2 n$ pasos para terminar, comparado con los n pasos requeridos cuando utilizamos operaciones atómicas, que serializan la ejecución.

[1] Cuando hace falta que un solo thread opere sobre la memoria compartida, generalmente se elige el thread 0 del bloque.

Esta organización además nos permite dividir el problema en subárboles, que se pueden mapear fácilmente a los bloques del modelo de ejecución de CUDA. Y dentro de los bloques, es posible cooperar muy rápidamente entre hilos a través de la memoria compartida.

El código del directorio `norm_tree` implementa parcialmente una estrategia de reducción en árbol dentro del bloque, y luego opera atómicamente sobre memoria global. Complete las líneas faltantes del kernel que calcula el mínimo y compare tiempos con el kernel anterior.

El bosque de reducciones

Reducir en árbol disminuye la cantidad de pasos requeridos, pero no todos los árboles de reducción se ejecutan de la misma forma. Dependiendo de entre qué pares de elementos se opere en cada paso de reducción, los warps pueden subutilizarse severamente (cuando pocos hilos de un warp operan y el resto no hace nada). Idealmente los warps deberían utilizarse completamente (cuando 32 hilos adyacentes tienen que ejecutar operaciones de reducción) o no utilizarse en absoluto, con lo que no ocupan unidades de ejecución.

Cambie la forma del árbol en el código del directorio `norm_opt` teniendo en cuenta esto para mejorar la performance y compare resultados.

El costo de sincronizar

Una vez mejorada la estructura de la reducción, note que para los últimos pasos (¿cuántos?) no es necesario sincronizar el bloque puesto que sólo un warp se encuentra en ejecución. Aplique esta optimización al kernel de `norm_opt`.

Más trabajo por hilo

Hay algunos hilos que lo único que hacen es copiar un elemento desde memoria global a memoria compartida, y después de pocas (o ninguna) operaciones terminan. El overhead que esto provoca en uso de registros y tiempo de ejecución es significativo.

Para solucionar este problema podemos serializar un poco la implementación. Una vez que tenemos corriendo suficientes hilos como para ocupar todos los recursos de la GPU, en vez de agregar más hilos para procesar todos los datos podemos hacer que cada hilo procese más datos. Esta técnica se llama *thread coarsening*, porque reduce la granularidad en la que trabajan los hilos.

Aplicado a este problema, lo que hay que hacer es que cada hilo reduzca una cantidad definible de elementos desde memoria global sobre un registro propio, antes de copiar su resultado intermedio a memoria compartida y continuar con la reducción en árbol habitual.

Aplique esta optimización al kernel del directorio `norm_opt`, y mida la performance para distintas cantidades de elementos por hilo.

Tareas extra

Las optimizaciones aplicadas siguen haciendo una operación atómica sobre memoria global. Para eliminar este último nivel de contención es necesario pedir un arreglo de memoria de tantos elementos como hayan bloques, hacer que cada bloque escriba su resultado parcial a dicho vector, y ejecutar nuevamente el kernel sobre este vector reducido hasta que corra en un único bloque que calcula el resultado final. Si le sobró tiempo, implemente esta estrategia.

Si todavía le sobra tiempo puede implementar una reducción intra-warp. Consulte en clase.

Referencias

[1] Mark Harris, Optimizing Parallel Reduction in CUDA, NVIDIA Developer Technology, 2007.