ECAR2012, Foundations track
Curso de CUDA, parte 2

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

Objetivos

- Hacer mediciones de tiempo desde el host y el profiler.
- Utilizar la memoria compartida como una caché manejada por el usuario.
- Utilizar primitivas de sincronización de threads.
- Entender los casos en los que no es necesario utilizar una barrera.
- Ver cómo la configuración de la grilla afecta la performance de una GPU.
- Utilizar una implementación optimizada de un problema común.

Introducción

En esta parte del práctico nos concentraremos en algunas técnicas básicas para analizar y mejorar la performance de los kernels. Para ello es necesario poder obtener métricas varias de la ejecución de los kernels, utilizando las herramientas de profiling que provee CUDA. Aplicaremos estas técnicas a un problema básico y bien conocido: la multiplicación de matrices en precisión simple.

Multiplicación de matrices

Implementación trivial

Para refrescar la memoria, el primer ejercicio consiste en completar el código del directorio `mm_simple`, realizando el manejo de memoria y copias, configurando la grilla y escribiendo un kernel CUDA que multiplique dos matrices de forma trivial: configure el grid de modo que cada thread opere sobre un elemento de la matriz destino C_{ij}, y cada thread calcula el producto punto entre la fila $A_{i1}..A_{im}$ y la columna $B_{1j}..B_{mj}$:

$$C_{ij} = \sum_{k=0}^{m}(A_{ik}.B_{kj})$$

Para verificar que el kernel esté implementado correctamente se provee una implementación trivial secuencial para CPU y, de haber errores, se listan las diferencias entre los resultados de ambos[1].

Tomar tiempos

Para medir el tiempo de ejecución de un kernel se pueden usar varios métodos:

- Sin ningún kernel en ejecución, medir desde el host el tiempo entre mandar a ejecutar el kernel y que termine de ejecutar (cuando `cudaDeviceSynchronize` termina), tomando la hora inicial y final con `gettimeofday` o similar y restando (en el caso de `gettimeofday`, con `timersub`). Este método es el que se usa para medir el tiempo de `mm_cpu`. El problema es que incluye el tiempo del host para encolar el kernel para ejecución y completar la sincronización, y no es del todo preciso para GPGPU.

[1] ¡Nunca paralelice código sin una implementación de referencia o batería de tests!

- Pedirle al profiler integrado de CUDA que nos reporte el tiempo de ejecución del kernel. Esto se logra configurando la variable de entorno `COMPUTE_PROFILE` en 1[2] y luego leyendo el archivo que genera:

```
$ export COMPUTE_PROFILE=1
$ ./mm
CPU time: 1.718391
$ cat cuda_profile_0.log
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla C2070
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6b0ec057a70
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 950.368 ] cputime=[ 995.000 ]
method=[ memcpyHtoD ] gputime=[ 943.232 ] cputime=[ 946.000 ]
method=[ _Z9mm_simplePKfS0_Pf ] gputime=[ 46781.504 ] cputime=[ 11.000 ] occupancy=[ 1.000 ]
method=[ memcpyDtoH ] gputime=[ 1301.952 ] cputime=[ 2074.000 ]
```

- Encolar eventos de CUDA antes y después de un kernel utilizando `cudaEventRecord`, y medir el tiempo ocurrido entre ambos con `cudaEventElapsedTime`. Esto funciona conceptualmente de manera similar al primer método, pero como el tiempo se toma en la GPU misma no se introduce ruido, devolviendo los mismos resultados que el profiler.

Mida el tiempo de ejecución del kernel que escribió usando `gettimeofday` y el profiler. Teniendo en cuenta que se realizan $2N^3$ operaciones de punto flotante, ¿a cuántos GFLOPS (miles de millones de operaciones de punto flotante por segundo) está operando el kernel?

Cada cosa en su lugar

El compilador CUDA generalmente respeta las ubicaciones en memoria de los objetos que especifica el usuario: las variables automáticas declaradas en el kernel se guardan en registros[3], los elementos declarados como `__shared__` siempre se guardan en *shared memory*, y los punteros a memoria global se usan desde memoria global[4].

Por esta razón, si escribió su kernel de multiplicación de matrices realizando la sumatoria del producto punto directamente sobre C_{ij}, entonces está operando sobre memoria global (aunque esté cacheada en L1 o L2) que tiene alta latencia.

Tome el código del directorio `mm_slow` y modifique el kernel para que utilice una variable temporal (que residirá en registros) para acumular la sumatoria. Luego escriba C_{ij} a memoria global una sola vez con el resultado final. Compare los tiempos de ambos kernels utilizando el profiler.

Usar *shared memory* como cache

Los 16KB[5] de cache L1 de las GPUs clase Fermi está dividido en líneas de 128 bytes, desde las que se sirven los pedidos a memoria. Cuando una fila de un bloque de threads calcula los resultados que le corresponden (las celdas $C_{i,p}..C_{i,p+ancho}$), utiliza $\lceil N/(128/\texttt{sizeof(T)})\rceil$ líneas de cache que corresponden a la i-ésima fila de A, y $N.\lceil ancho.\texttt{sizeof(T)}/128\rceil$ líneas de cache que corresponden a las columnas $p..p+ancho$ de B. Esta diferencia generalmente resulta en que las líneas de cache de A sean descartadas y tengan que ser releídas.

Para resolver este problema, utilizaremos la memoria compartida de la GPU para almacenar los valores de A. El uso de parte de los 48KB de memoria compartida junto con la cache L1 además amplía el lugar para mantener datos más cerca de las unidades de ejecución.

Una versión de este procedimiento es:

[2]Para detener el profiling, puede eliminar la variable de entorno ejecutando `unset COMPUTE_PROFILE`.

[3]Salvo en el caso que no pueda determinarse de manera estática el uso del registro, por ejemplo cuando se declaran arreglos a los que se accede de manera dinámica, en cuyo caso se guardan en memoria global.

[4]A través de la cache L1 y L2.

[5]Configurables a 48KB reduciendo el tamaño de la memoria compartida, usando `cudaDeviceSetCacheConfig` y `cudaFuncSetCacheConfig`.

1. Declarar una matriz `__shared__` de las mismas dimensiones que el bloque de threads. Tenga en cuenta que puede declarar un arreglo multidimensional `__shared__ float arreglo[altura][ancho]` porque lo va a usar en la misma función donde está declarado, en vez de pasarlo como parámetro, con lo que se perderían sus dimensiones.

2. Leer un elemento de A por thread y escribirlo en la celda de la matriz correspondiente al thread.

3. Esperar a que el resto del bloque escriba sus valores en memoria compartida utilizando la barrera `__syncthreads()`.

4. Ahora puede procesar las filas de B leyendo los valores de A desde la copia en memoria compartida. Una vez que se quede sin valores (cosa que sucede luego de tantas iteraciones como el ancho del bloque), repetir todo el proceso desde el principio.

Modifique el kernel que optimizó en el ejercicio anterior para que realice esta optimización adicional y compare tiempos.

No sincronizar más de lo necesario

Como las GPUs ejecutan un warp de hilos a la vez, hay veces que este sincronismo implícito hace que no sea necesario sincronizar todo el bloque de threads, que es costoso.

Considere la copia de elementos de A a la memoria compartida: si el ancho del bloque es igual al tamaño de un warp, entonces la escritura de toda la fila va a ocurrir de manera simultánea, y podemos ahorrarnos la barrera.

Redimensione el tamaño de bloque del código del directorio `mm_shared` manteniendo la misma cantidad de hilos por bloque. Tenga en cuenta lo dicho respecto a los warps para eliminar las barreras y compare tiempos.

Ajustar la configuración de los kernels

Una de las claves del uso efectivo de los recursos de la GPU es el maximizar el ocultamiento de latencia a la memoria. Una forma de realizarlo es ajustando el tamaño del bloque de threads.

Para GPUs de arquitecturas Fermi (compute capability 2.x), cada multiprocesador puede albergar 1536 threads (48 warps) en ejecución divididos entre un máximo de 8 bloques y con un tamaño máximo de bloque de 1024 threads. Una mayor cantidad de bloques pequeños implica mejor ocultamiento de latencia, mientras que bloques más grandes permiten cooperar entre más threads. Además se suman consideraciones como las del tamaño del warp del ejercicio anterior.

Teniendo en cuenta estos factores, mida el tiempo de ejecución del kernel para distintos tamaños de bloque y analice los resultados. ¿Cuál es el tamaño óptimo y por qué?

No reinventar la rueda

Para la mayoría de los problemas más comunes, ya existen implementaciones altamente optimizadas para GPU[6]. En el caso de la multiplicación de matrices, hay soluciones incluídas en los paquetes BLAS[7].

Reemplazaremos nuestro kernel por el de la biblioteca CUBLAS. Para utilizar esta biblioteca es necesario:

1. Incluir el header `cublas_v2.h` y crear una variable de tipo `cublasHandle_t` donde CUBLAS guarda su información interna.

2. Inicializar la biblioteca CUBLAS utilizando la función `cublasCreate`, que toma como parámetro un puntero a un `cublasHandle_t`.

3. Reemplazar la llamada al kernel por la función `cublasSgemm`[8], que calcula la ecuación más general $C = \alpha AB + \beta C$ y tiene la siguiente interfaz:

[6] ¡Esto también aplica para CPUs!
[7] Basic Linear Algebra Subprograms
[8] SGEMM: single precision general matrix-matrix multiply en la notación BLAS.

```
cublasSgemm(cublasHandle_t handle,
            cublasOperation_t transa, cublasOperation_t transb
            int m, int n, int k,
            const float *alpha,
            const float *A, int lda,
            const float *B, int ldb,
            const float *beta,
            float *C, int ldc)
```

Donde:

- `transa` y `transb` deben valer `CUBLAS_OP_N` porque no queremos transponer ni A ni B.
- `n`, `m` y `k` son las dimensiones de las matrices, en este caso `N`.
- `alpha` es un puntero a una variable de punto flotante con el valor de α, en este caso `1.0f`.
- `lda`, `ldb` y `ldc` son los *strides* (la cantidad de elementos entre dos celdas de columnas adyacentes) de las matrices, en este caso `N`.
- `beta` es un puntero a una variable de punto flotante con el valor de β, en este caso `0.0f`.

4. Cerrar la biblioteca CUBLAS antes de salir con la función `cublasDestroy`, que toma un `cublasHandle_t`.
5. Linkear el ejecutable con CUBLAS agregando `-lcublas` a la línea apropiada en el Makefile.

Cabe aclarar que, teniendo una interfaz heredada de Fortran, las bibliotecas BLAS operan sobre matrices con disposición de memoria column-major.

En el directorio `mm_cublas` se provee una versión de nuestro programa habitual que ha sido convertida a matrices column-major, y a la que de la lista de tareas anterior sólo le faltan las llamadas a funciones `cublasCreate`, `cublasSgemm` y `cublasDestroy`. Complete el código y compare tiempos. ¿Cuántos GFLOPS obtiene CUBLAS? Contraste con la performance teórica pico de la placa.

Extras

Si le sobró tiempo, siga mejorando su kernel:

- Extienda lo hecho en memoria compartida para la matriz A para la matriz B.
- Haga un uso más eficiente de los registros y reduzca el overhead de los cálculos comunes a todos los threads procesando más de una celda por hilo.