ugr | Universidad de Granada

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Grado en Ingeniería Informática

Arquitecturas y Computación de altas prestaciones

Práctica 4

Autora:
DE LA VIEJA LAFUENTE, CLAUDIA

Curso:
2023-2024

Índice

1. Ejercicio 1

Realiza una traducción reemplazando sus resultados de medición por los que tú obtengas, ya sea en tu propia máquina o en GenMagic. Lógicamente, en el proceso de traducción y replicación ten en cuenta la tarjeta que estés usando (por ejemplo, respecto a valores pico)

1.1. Sincronización host-dispositivo

Echemos un vistazo a las transferencias de datos y el lanzamiento del kernel del código SAXPY host del post anterior:

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

saxpy<<<(N+511)/512, 512>>>(N, 2.0f, d_x, d_y);

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

Las transferencias de datos entre el host y el dispositivo utilizando `cudaMemcpy()` son transferencias síncronas (o de bloqueo). Las transferencias de datos síncronas no comienzan hasta que todas las llamadas CUDA emitidas previamente hayan finalizado, y las llamadas CUDA subsiguientes no pueden comenzar hasta que la transferencia síncrona haya finalizado. Por lo tanto, el lanzamiento del kernel `saxpy` en la tercera línea no se emitirá hasta que la transferencia de `y` a `d_y` en la segunda línea se haya completado. El lanzamiento del kernel, por otro lado, es asíncrono. Una vez que el kernel es lanzado en la tercera línea, el control regresa inmediatamente al CPU y no espera a que el kernel se complete. Aunque esto podría parecer una condición de carrera para la transferencia de datos de dispositivo a host en la última línea, la naturaleza de bloqueo de la transferencia de datos asegura que el kernel se complete antes de que comience la transferencia.

1.2. Cronometraje de la ejecución del núcleo con temporizadores de CPU

Veamos ahora cómo cronometrar la ejecución del kernel utilizando un temporizador de CPU.

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

t1 = myCPUTimer();
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
cudaDeviceSynchronize();
t2 = myCPUTimer();

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

Además de las dos llamadas a la función genérica de marca de tiempo del host `myCPUTimer()`, utilizamos la barrera de sincronización explícita `cudaDeviceSynchronize()` para bloquear la ejecución de la CPU hasta que todos los comandos emitidos previamente en el dispositivo se hayan completado. Sin esta barrera, este código mediría el tiempo de lanzamiento del kernel y no el tiempo de ejecución del kernel.

1.3. Cronometraje mediante eventos CUDA

Uno de los problemas de utilizar puntos de sincronización host-dispositivo, como cudaDeviceSynchronize(), es que paralizan el pipeline de la GPU. Por este motivo, CUDA ofrece una alternativa relativamente ligera a los temporizadores de la CPU a través de la API de eventos de CUDA. La API de eventos de CUDA incluye llamadas para crear y destruir eventos, registrar eventos y calcular el tiempo transcurrido en milisegundos entre dos eventos registrados.

Los eventos CUDA utilizan el concepto de flujos CUDA. Un flujo CUDA es simplemente una secuencia de operaciones que se realizan en orden en el dispositivo. Las operaciones en diferentes flujos pueden intercalarse y, en algunos casos, solaparse, una propiedad que puede utilizarse para ocultar las transferencias de datos entre el host y el dispositivo (más adelante hablaremos de esto en detalle). Hasta ahora, todas las operaciones en la GPU se han producido en el flujo por defecto, o flujo 0 (también llamado "flujo nulo").

En el siguiente listado aplicamos eventos CUDA a nuestro código SAXPY.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

cudaEventRecord(start);
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
cudaEventRecord(stop);

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

Los eventos CUDA son de tipo `cudaEvent_t` y se crean y destruyen con `cudaEventCreate()` y `cudaEventDestroy()`. En el código anterior `cudaEventRecord()` coloca los eventos de inicio y parada en el stream por defecto, stream 0. El dispositivo registrará una marca de tiempo para el evento cuando alcance ese evento en el stream. La función `cudaEventSynchronize()` bloquea la ejecución de la CPU hasta que el evento especificado es registrado. La función `cudaEventElapsedTime()` devuelve en el primer argumento el número de milisegundos de tiempo transcurrido entre la grabación de inicio y parada. Este valor tiene una resolución aproximada de medio microsegundo.

1.4. Ancho de banda de memoria

Ahora que disponemos de un medio para cronometrar con precisión la ejecución del núcleo, lo utilizaremos para calcular el ancho de banda. Al evaluar la eficiencia del ancho de banda, utilizamos tanto el ancho de banda pico teórico como el ancho de banda de memoria observado o efectivo.

1.4.1. Ancho de banda teórico

El ancho de banda teórico puede calcularse utilizando las especificaciones de hardware disponibles en la documentación del producto. Por ejemplo, nuestra GPU del genmagic tiene una velocidad de reloj

de memoria de 1.750 MHz y una interfaz de memoria de 352 bits de ancho. Utilizando estos datos, el ancho de banda de memoria teórico máximo es de 154 GB/seg, tal y como se calcula a continuación.
$\text{BW}_{\text{teórico}} = 1750 \times 10^6 \times \left(\frac{352}{8}\right) \times \frac{2}{109} = 154$ GB/s

1.5. Ancho de banda efectivo

Calculamos el ancho de banda efectivo cronometrando actividades específicas del programa y sabiendo cómo accede nuestro programa a los datos. Utilizamos la siguiente ecuación

$$\text{Ancho de banda efectivo} = \frac{RB + WB}{t \times 10^9}$$

Aquí, *BWEffective* es el ancho de banda efectivo en unidades de GB/s, *RB* es el número de bytes leídos por núcleo, *WB* es el número de bytes escritos por núcleo y t es el tiempo transcurrido en segundos. Podemos modificar nuestro ejemplo SAXPY para calcular el ancho de banda efectivo. El código completo es el siguiente.
En el cálculo del ancho de banda, N*4 es el número de bytes transferidos por lectura o escritura del array, y el factor de tres representa la lectura de x y la lectura y escritura de y. El tiempo transcurrido se almacena en la variable milisegundos para dejar claras las unidades. Observa que además de añadir la funcionalidad necesaria para el cálculo del ancho de banda, también hemos cambiado el tamaño del array y el tamaño del bloque de hilos. Compilando y ejecutando este código tenemos:

```
estudiante7@genmagic:~$ ./a.out
Max error: 0.000000n
Effective Bandwidth (GB/s): 557.002624n
```

1.6. Medición del rendimiento computacional

Acabamos de demostrar cómo medir el ancho de banda, que es una medida del rendimiento de los datos. Otra medida muy importante para el rendimiento es el rendimiento computacional. Una medida común de rendimiento computacional es GFLOP/s, que significa ≪Giga-FLoating-point OPerations per second≫, donde Giga es el prefijo para 10^9. Para nuestro cálculo SAXPY, medir el rendimiento efectivo es sencillo: cada elemento SAXPY realiza una operación de multiplicación-suma, que suele medirse como dos FLOPs, por lo que tenemos
$\text{GFLOP/s Effective} = 2\text{N}_{\overline{t \times 10^9}}$

1.7. En resumen

En esta entrada se describe cómo cronometrar la ejecución del kernel utilizando la API de eventos de CUDA. Los eventos CUDA utilizan el temporizador de la GPU y, por tanto, evitan los problemas asociados a la sincronización host-dispositivo. Presentamos las métricas de rendimiento de ancho de banda efectivo y rendimiento computacional, e implementamos el ancho de banda efectivo en el kernel SAXPY. Un gran porcentaje de kernels están limitados por el ancho de banda de memoria, por lo que el cálculo del ancho de banda efectivo es un buen primer paso en la optimización del rendimiento. En una próxima entrada hablaremos de cómo determinar qué factor -ancho de banda, instrucciones o latencia- es el que limita el rendimiento.
Los eventos CUDA también pueden utilizarse para determinar la tasa de transferencia de datos entre el host y el dispositivo, registrando eventos a ambos lados de las llamadas a cudaMemcpy().
Si ejecutas el código de este post en una GPU más pequeña, es posible que aparezca un mensaje de error relativo a la insuficiencia de memoria del dispositivo, a menos que reduzcas el tamaño de las matrices. De hecho, nuestro código de ejemplo hasta ahora no se ha molestado en comprobar si se producen errores en tiempo de ejecución. En el próximo post, aprenderemos a realizar la gestión de

errores en CUDA C/C++ y a consultar los dispositivos presentes para determinar sus recursos disponibles, de forma que podamos escribir código mucho más robusto.

2. Ejercicio 2

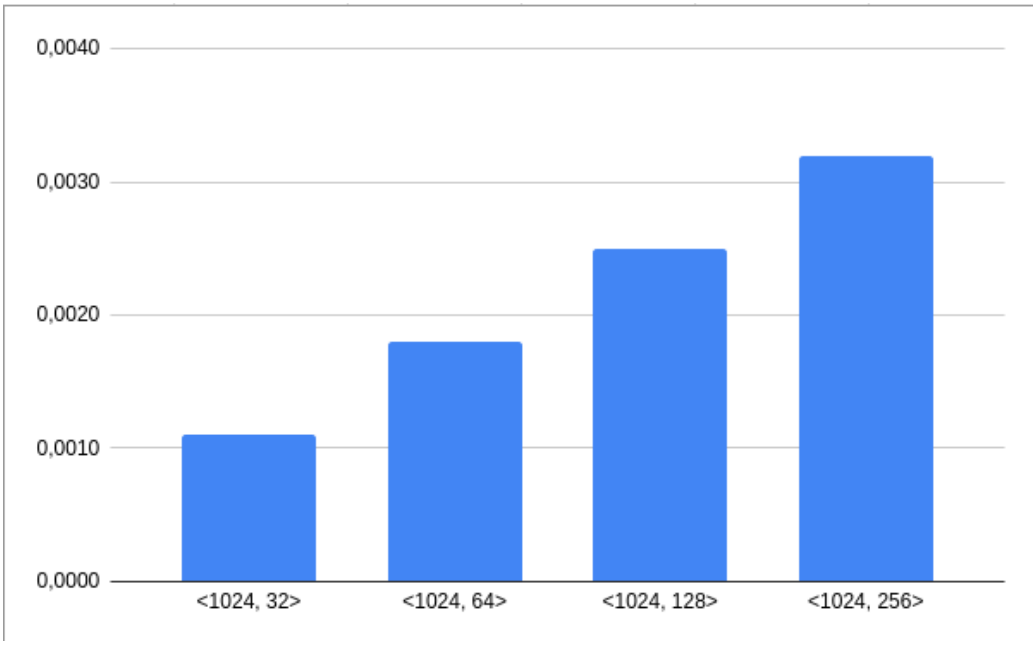
1. Generalización.

```
estudiante7@genmagic:~$ ./a.out
Tiempo de CPU (s): 0.1592
Check-sum CPU: 104857600
Check-sum GPU: 104857600
Calculo correcto!!
Tiempo medio de ejecucion del kernel<<<1024, 1024>>> sobre 104857600 bytes [s]: 0.0006
```

2. Realiza un estudio sobre cómo afecta la variación del tamaño de bloque considerando al menos 4 valores distintos.

   Para realizar un estudio sobre cómo afecta la variación del tamaño de bloque, podríamos modificar el valor de `THREADS_PER_BLOCK` y observar cómo afecta al tiempo de ejecución del kernel y al rendimiento general. Como podemos comprobar he probado con al menos cuatro valores diferentes para `THREADS_PER_BLOCK` y he medido el tiempo medio de ejecución del kernel para cada uno.

```
estudiante7@genmagic:~$ ./a.out
Tiempo de CPU (s): 0.1592
Check-sum CPU: 104857600
Tiempo medio de ejecucion del kernel<<<1024, 32>>> sobre 104857600 bytes [s]: 0.0011
Tiempo medio de ejecucion del kernel<<<1024, 64>>> sobre 104857600 bytes [s]: 0.0018
Tiempo medio de ejecucion del kernel<<<1024, 128>>> sobre 104857600 bytes [s]: 0.0025
Tiempo medio de ejecucion del kernel<<<1024, 256>>> sobre 104857600 bytes [s]: 0.0032
```

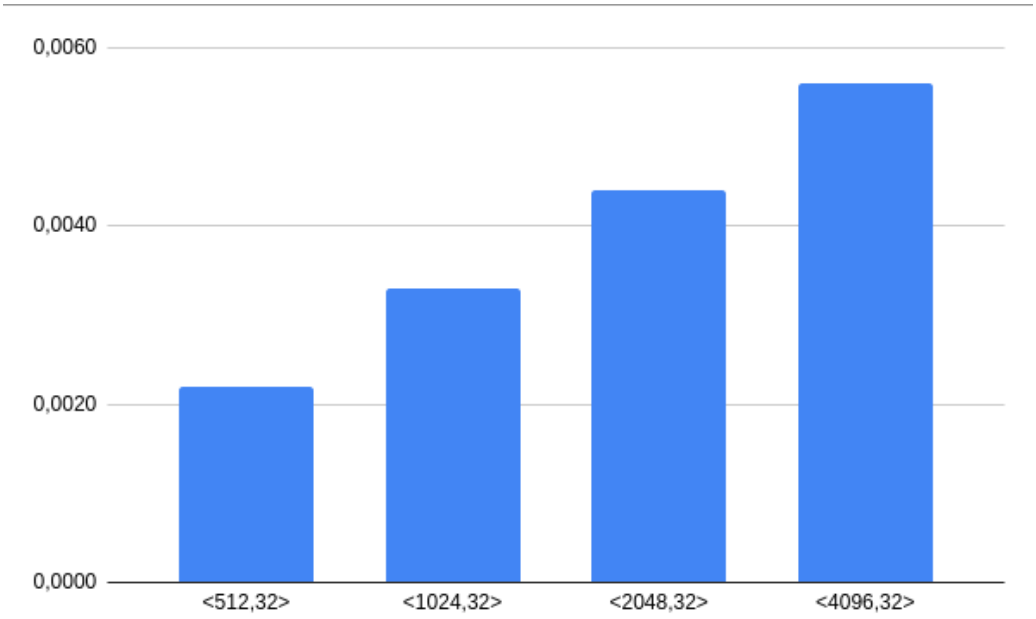


3. Haz el mismo tipo de estudio sobre el impacto del tamaño grid (número de bloques) tras haber fijado el tamaño de bloque al mejor valor encontrado anteriormente.

   Para estudiar el impacto del tamaño del grid (número de bloques) después de haber fijado el tamaño del bloque al mejor valor encontrado anteriormente, se modifica el valor de `NBLOCKS` y se observa cómo afecta al tiempo de ejecución del kernel y al rendimiento general.

   He realizado un estudio similar al anterior, pero manteniendo fijo el tamaño del bloque al mejor valor encontrado (32) y variando el número de bloques en el grid.

```
estudiante7@genmagic:~$ ./a.out
Tiempo de CPU (s): 0.1592
Check-sum CPU: 104857600
Tiempo medio de ejecucion del kernel<<<512, 32>>> sobre 104857600 bytes [s]: 0.0022
Tiempo medio de ejecucion del kernel<<<1024, 32>>> sobre 104857600 bytes [s]: 0.0033
Tiempo medio de ejecucion del kernel<<<2048, 32>>> sobre 104857600 bytes [s]: 0.0044
Tiempo medio de ejecucion del kernel<<<4096, 32>>> sobre 104857600 bytes [s]: 0.0056
```



Con este apartado y el anterior podemos decir que a menor tamaño de bloque y menor grid el tiempo medio de ejecución es menor.

4. Sobre la versión general y mejor configuración, calcula la aceleración sobre el cálculo en CPU. Esto hazlo de dos formas, primero limitándote a comparar sólo tiempos de cálculo, y en última instancia teniendo en cuenta las transferencias a (y desde) GPU en los registros de tiempo.

```
estudiante7@genmagic:~$ ./a.out
Tiempo de CPU (s): 0.1640
Check-sum CPU: 104857600
Tiempo medio de ejecucion del kernel en GPU [s]: 0.0022
Tiempo de CPU (s): 0.1605
Tiempo de ejecucion del cálculo en CPU [s]: 0.1605
Aceleracion basada en los tiempos de cálculo: 74.41
```

Como ya hemos estudiado en los apartados anteriores, la mejor configuración seria `NBLOCKS` = 512 y `THREADS_PER_BLOCK` = 32.

La aceleración se calcula comparando el tiempo de ejecución del cálculo en la CPU con el tiempo medio de ejecución del kernel en la GPU. La fórmula general para calcular la aceleración es:

$$\text{Aceleración} = \frac{\text{Tiempo de ejecución en CPU}}{\text{Tiempo medio de ejecución del kernel en GPU}}$$

En el código proporcionado, el tiempo de ejecución en la CPU se mide utilizando la función `get_wall_time()`, que registra el tiempo antes y después de la ejecución de la función `histogramaCPU()`. Este tiempo se almacena en la variable `cpu_time`.

El tiempo medio de ejecución del kernel en GPU se calcula sumando los tiempos de ejecución registrados en cada iteración del bucle y dividiéndolo por el número total de ejecuciones. Este valor se almacena en la variable `aveGPUMS`.

Finalmente, la aceleración se calcula dividiendo el tiempo de ejecución en CPU entre el tiempo medio de ejecución del kernel en GPU:

$$\text{Aceleración} = \frac{cpu\_time}{\text{aveGPUMS}/(numRuns * 1000{,}0)}$$

Este valor nos indica cuántas veces más rápido es el cálculo en la GPU en comparación con la CPU. Si la aceleración es mayor que 1, significa que el cálculo en la GPU es más rápido que en la CPU.