Memoria de la practica 4:

Autor: [Amador Carmona Méndez]

Asignatura:[Arquitecturas y Computación de Altas Prestaciones]

Fecha: [1/05/2024]

Ejercicio1, Traduccion del tutorial y medidas en genMagic

Realizo la traducción del tutorial del enlace propuesto por el profesor y cambio la ejecución del programa en este mismo por la ejecucion del programa en el servidor de GenMagic de la UGR:

Cómo Implementar Métricas de Rendimiento en CUDA C/C++

En la primera entrada de esta serie, examinamos los elementos básicos de CUDA C/C++ al analizar una implementación de SAXPY en CUDA C/C++. En esta segunda entrada, discutimos cómo analizar el rendimiento de este y otros códigos en CUDA C/C++. Nos basaremos en estas técnicas de medición de rendimiento en futuras entradas donde la optimización de rendimiento será cada vez más importante.

La medición del rendimiento en CUDA se hace comúnmente desde el código del host y puede implementarse utilizando temporizadores de CPU o temporizadores específicos de CUDA. Antes de adentrarnos en estas técnicas de medición de rendimiento, necesitamos discutir cómo sincronizar la ejecución entre el host y el dispositivo.

Sincronización entre Host y Dispositivo

Veamos las transferencias de datos y el lanzamiento del kernel de SAXPY desde el código del host de la entrada anterior:

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

saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);

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

Las transferencias de datos entre el host y el dispositivo usando cudaMemcpy() son síncronas (o bloqueantes). Las transferencias de datos síncronas no comienzan hasta que todas las llamadas CUDA emitidas previamente se han completado, y las llamadas CUDA subsiguientes no pueden comenzar hasta que la transferencia síncrona se haya completado. Por lo tanto, el lanzamiento del kernel de saxpy en la tercera línea no se ejecutará hasta que la transferencia de y a d_y en la segunda línea haya finalizado. Los lanzamientos de kernel, por otro lado, son asíncronos. Una vez que se lanza el kernel en la tercera línea, el control vuelve inmediatamente a la CPU y no espera a que el kernel se complete. Aunque esto podría parecer que establece una condición de carrera para la transferencia de datos de dispositivo a host en la última línea, la naturaleza bloqueante de la transferencia de datos garantiza que el kernel se complete antes de que comience la transferencia.

Medición del Tiempo de Ejecución del Kernel con Temporizadores de CPU

Ahora veamos cómo medir el tiempo de ejecución del kernel usando 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.

Medición usando Eventos de CUDA

Un problema al usar puntos de sincronización entre host y dispositivo, como cudaDeviceSynchronize(), es que detienen la canalización de GPU. Por esta razón, CUDA ofrece una alternativa relativamente ligera a los temporizadores de 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 de CUDA hacen uso del concepto de flujos de CUDA. Un flujo de 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, superponerse, una propiedad que puede usarse para ocultar las transferencias de datos entre el host y el dispositivo (discutiremos esto en detalle más adelante). Hasta ahora, todas las operaciones en la GPU han ocurrido en el flujo predeterminado, o flujo 0 (también llamado "Null Stream").

En la siguiente lista, aplicamos eventos de 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 de CUDA son del tipo cudaEvent_t y se crean y destruyen con cudaEventCreate() y cudaEventDestroy(). En el código anterior, cudaEventRecord() coloca los eventos de inicio y finalización en

el flujo predeterminado, flujo 0. El dispositivo registrará una marca de tiempo para el evento cuando alcance ese evento en el flujo. La función cudaEventSynchronize() bloquea la ejecución de la CPU hasta que se registre el evento especificado. La función cudaEventElapsedTime() devuelve en el primer argumento el número de milisegundos transcurridos entre el registro de inicio y finalización. Este valor tiene una resolución de aproximadamente medio microsegundo.

Ancho de Banda de Memoria

Ahora que tenemos un medio para medir con precisión la ejecución del kernel, lo usaremos para calcular el ancho de banda. Al evaluar la eficiencia del ancho de banda, utilizamos tanto el ancho de banda teórico máximo como el ancho de banda de memoria observado o efectivo.

Ancho de Banda Teórico

El ancho de banda teórico se puede calcular utilizando las especificaciones del hardware disponibles en la literatura del producto. Por ejemplo, la GPU NVIDIA Tesla M2050 utiliza RAM DDR (double data rate) con una velocidad de reloj de memoria de 1,546 MHz y una interfaz de memoria de 384 bits de ancho. Utilizando estos datos, el ancho de banda de memoria teórico máximo de la NVIDIA Tesla M2050 es de 148 GB/s, como se calcula a continuación.

[BW_{\text{Teórico}} = 1546 \times 10^6 \times \frac{384}{8} \times 2 \div 10^9 = 148 , \text{GB/s}]

En este cálculo, convertimos la velocidad de reloj de memoria a Hz, la multiplicamos por el ancho de la interfaz (dividido por 8, para convertir bits a bytes) y la multiplicamos por 2 debido a la tasa de datos doble. Finalmente, dividimos por (10^9) para convertir el resultado a GB/s.

Ancho de Banda Efectivo

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

[BW_{\text{Efectivo}} = \frac{RB + WB}{t \times 10^9}]

Aquí, (BW_{\text{Efectivo}}) es el ancho de banda efectivo en unidades de GB/s, RB es el número de bytes leídos por kernel, WB es el número de bytes escritos por kernel, y (t) es el tiempo transcurrido dado en segundos. Podemos modificar nuestro ejemplo de SAXPY para calcular el ancho de banda efectivo. El código completo sigue a continuación.

```
#include

__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  int N = 20 * (1 << 20);
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));
```

```
  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  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);

  // Perform SAXPY on 1M elements
  saxpy<<<(N+511)/512, 512>>>(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);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++) {
    maxError = max(maxError, abs(y[i]-4.0f));
  }

  printf("Max error: %fn", maxError);
  printf("Effective Bandwidth (GB/s): %fn", N*4*3/milliseconds/1e6);
}
```

En el cálculo del ancho de banda, N*4 es el número de bytes transferidos por lectura o escritura de 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 hacer claras las unidades. Nótese 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 threads. Compilando y ejecutando este código en una Tesla M2050 tenemos:

```
estudiante3@genmagic:~/practica4$ ./ejercicio1
Max error: 0.000000nEffective Bandwidth (GB/s): 552.502464n
```

Medición del Rendimiento Computacional

Acabamos de demostrar cómo medir el ancho de banda, que es una medida del rendimiento de datos. Otra métrica muy importante para el rendimiento es el rendimiento computacional. Una medida común del rendimiento computacional es GFLOP/s, que significa "Giga-Operaciones de Punto Flotante por segundo",

donde Giga es el prefijo para 10^9. Para nuestra computación SAXPY, medir el rendimiento efectivo es simple: cada elemento SAXPY realiza una operación de multiplicación-suma, que típicamente se mide como dos FLOPs, así que tenemos

[GFLOP/s , Efectivo = \frac{2N}{t \times 10^9}]

N es el número de elementos en nuestra operación SAXPY, y t es el tiempo transcurrido en segundos. Al igual que con el ancho de banda pico teórico, el GFLOP/s pico teórico se puede obtener de la literatura del producto (pero calcularlo puede ser un poco complicado porque depende mucho de la arquitectura). Por ejemplo, la GPU Tesla M2050 tiene un rendimiento teórico pico de operaciones de punto flotante en precisión simple de 1030 GFLOP/s, y un rendimiento teórico pico en precisión doble de 515 GFLOP/s.

SAXPY lee 12 bytes por elemento computado, pero realiza solo una instrucción de multiplicación-suma (2 FLOPs), así que está bastante claro que será limitado por el ancho de banda, y por lo tanto en este caso (de hecho en muchos casos), el ancho de banda es la métrica más importante para medir y optimizar. En cálculos más sofisticados, medir el rendimiento en el nivel de FLOPs puede ser muy difícil. Por lo tanto, es más común usar herramientas de perfilado para tener una idea de si el rendimiento computacional es un cuello de botella. Las aplicaciones a menudo proporcionan métricas de rendimiento que son específicas del problema (en lugar de específicas de la arquitectura) y por lo tanto más útiles para el usuario. Por ejemplo, "Mil Millones de Interacciones por Segundo" para problemas astronómicos de n cuerpos, o "nanosegundos por día" para simulaciones de dinámica molecular.

Resumen

Esta entrada describió cómo medir la ejecución del kernel usando la API de eventos de CUDA. Los eventos de CUDA utilizan el temporizador de la GPU y, por lo tanto, evitan los problemas asociados con la sincronización entre host y dispositivo. Presentamos las métricas de ancho de banda efectivo y rendimiento computacional, e implementamos el ancho de banda efectivo en el kernel de 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 entrada futura discutiremos cómo determinar qué factor -ancho de banda, instrucciones o latencia- es el factor limitante en el rendimiento.

Los eventos de CUDA también se pueden utilizar para determinar la velocidad de transferencia de datos entre el host y el dispositivo, registrando eventos en cada lado de las llamadas de `cudaMemcpy()`.

Si ejecutas el código de esta entrada en una GPU más pequeña, es posible que recibas un mensaje de error sobre la insuficiencia de memoria del dispositivo a menos que reduzcas los tamaños de los arrays. De hecho, hasta ahora nuestro código de ejemplo no se ha molestado en verificar errores en tiempo de ejecución. En la próxima entrada, aprenderemos cómo realizar el manejo de errores en CUDA C/C++ y cómo consultar los dispositivos presentes para determinar sus recursos disponibles, de modo que podamos escribir código mucho más robusto.

Si ejecutas el código de esta entrada en una GPU más pequeña, es posible que recibas un mensaje de error sobre la insuficiencia de memoria del dispositivo a menos que reduzcas los tamaños de los arrays. De hecho, hasta ahora nuestro código de ejemplo no se ha molestado en verificar errores en tiempo de ejecución. En la próxima entrada, aprenderemos cómo realizar el manejo de errores en CUDA C/C++ y cómo consultar los dispositivos presentes para determinar sus recursos disponibles, de modo que podamos escribir código mucho más robusto.

Ejercicio2:

Partiendo del codigo pr4.cu, hacemos las siguiente modificaciones:

Generalizacion del numero de hilos por bloque:

Realizamos las siguientes modificaciones en el codigo:

- Generalización del número de bloques: Se ha eliminado el valor fijo NBLOCKS y se ha calculado el número de bloques necesario utilizando la fórmula (SIZE + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK. Luego, este valor calculado se utiliza en la invocación del kernel kernelHistograma en lugar del valor fijo NBLOCKS. Resultado de la ejecución:

```
estudiante3@genmagic:~/practica4$ ./ejercicio2v1
Tiempo de CPU (s): 0.0000
Check-sum CPU: 104857600
Check-sum GPU: 104857600
Calculo correcto!!
Tiempo medio de ejecucion del kernel<<<409600, 256>>> sobre 104857600 bytes [s]:
0.0063
estudiante3@genmagic:~/practica4$
```

Estudio del tamaño de bloque:

Realizamos las siguientes modificaciones en el codigo:

- Modificación del bucle principal: Se ha modificado el bucle principal para iterar sobre cuatro valores distintos para el tamaño de bloque: 64, 128, 256 y 512. Esto se logra mediante el bucle for(int blockSize = 64; blockSize <= 1024; blockSize *= 2).
- Cálculo del número de bloques: Se ha ajustado el cálculo del número de bloques necesarios para cada tamaño de bloque dentro del bucle principal. El número de bloques se calcula utilizando la fórmula (SIZE + blockSize - 1) / blockSize.
- Registro de tiempos de ejecución promedio: Se ha agregado un array aveGPUMS[4] para almacenar los tiempos de ejecución promedio para cada tamaño de bloque.
- Cálculo del tiempo de ejecución promedio: Se ha modificado el cálculo del tiempo de ejecución promedio dentro del bucle principal para cada tamaño de bloque.
- Impresión de resultados: Se ha añadido una sección para imprimir los resultados de los tiempos de ejecución promedio para cada tamaño de bloque al final del programa. Resultado de la ejecución:

```
./ejercicio2v2
Tiempo de CPU (s): 0.0000
Check-sum CPU: 104857600
Check-sum GPU: 172813358334
Resultados de tiempos de ejecución promedio:
Tamaño de bloque: 64, Tiempo medio de ejecución [s]: 0.0126
Tamaño de bloque: 128, Tiempo medio de ejecución [s]: 0.0057
Tamaño de bloque: 256, Tiempo medio de ejecución [s]: 0.0000
Tamaño de bloque: 512, Tiempo medio de ejecución [s]: 0.0046
estudiante3@genmagic:~/practica4$
```

Estudio del tamaño del grid:

Realizamos las siguientes modificaciones en el codigo:

- Modificación del bucle principal: Se ha modificado el bucle principal para iterar sobre cuatro valores distintos para el tamaño del grid (número de bloques), que son 32, 64, 128 y 256. Esto se logra mediante el bucle for(int numBlocks = 32; numBlocks <= 2048; numBlocks *= 2).
- Registro de tiempos de ejecución promedio: Se ha ajustado el cálculo del tiempo de ejecución promedio dentro del bucle principal para cada tamaño del grid.
- Impresión de resultados: Se ha añadido una sección para imprimir los resultados de los tiempos de ejecución promedio para cada tamaño del grid al final del programa. Resultado de la ejecución:

```
estudiante3@genmagic:~/practica4$ ./ejercicio2v3
Tiempo de CPU (s): 0.0000
Check-sum CPU: 104857600
Check-sum GPU: 104857600
Resultados de tiempos de ejecución promedio:
Tamaño de grid: 512, Tiempo medio de ejecución [s]: 0.0007
Tamaño de grid: 1024, Tiempo medio de ejecución [s]: 0.0007
Tamaño de grid: 2048, Tiempo medio de ejecución [s]: 0.0000
Tamaño de grid: 4096, Tiempo medio de ejecución [s]: 0.0006
estudiante3@genmagic:~/practica4$
```

Realizamos las siguientes modificaciones en el codigo:

- Cálculo del tiempo de ejecución en CPU: Se ha agregado código para medir el tiempo de ejecución en la CPU utilizando la función get_wall_time() antes y después de llamar a la función histogramaCPU(), y luego se imprime el tiempo de cálculo en la CPU.
- Cálculo del tiempo de ejecución en GPU: Se han realizado dos bloques de código similares para calcular el tiempo de ejecución en GPU para la versión general y la mejor configuración. Estos bloques de código utilizan la función cudaEventRecord() para registrar el tiempo antes y después de la ejecución del kernel, y luego calculan el tiempo transcurrido.
- Cálculo de la aceleración: Se ha calculado la aceleración basada solo en los tiempos de cálculo dividiendo el tiempo de cálculo en CPU por el tiempo de cálculo en GPU para ambas la versión general y la mejor configuración.
- Consideración de las transferencias a y desde la GPU: Se ha agregado una variable tiempoTransferencias para representar el tiempo de transferencias entre la CPU y la GPU, suponiendo que es insignificante en este caso. Luego, se ha calculado el tiempo total en GPU considerando las transferencias y se ha calculado la aceleración total teniendo en cuenta este tiempo. Resultado de la ejecución:

```
estudiante3@genmagic:~/practica4$ ./ejercicio2v4
Tiempo de CPU (s): 0.0000
Check-sum CPU: 104857600
Tiempo de CPU (s): 0.0000
Tiempo de cálculo en CPU [s]: 0.1591
Tiempo medio de ejecución en GPU (versión general) [s]: 0.0007
Tiempo medio de ejecución en GPU (mejor configuración) [s]: 0.0007
Aceleración (solo cálculo) - Versión general: 242.08x
Aceleración (solo cálculo) - Mejor configuración: 242.26x
Aceleración (considerando transferencias) - Versión general: 242.08x
Aceleración (considerando transferencias) - Mejor configuración: 242.26x
estudiante3@genmagic:~/practica4$
```

Teniendo en cuenta los resultados podemos decir que:

En la ejecución, el tiempo de cálculo en CPU fue de aproximadamente 0.1591 segundos, mientras que el tiempo medio de ejecución en GPU para ambas la versión general y la mejor configuración fue de alrededor de 0.0007 segundos. Esto resultó en una aceleración de aproximadamente 242 veces para ambas configuraciones, lo que significa que el cálculo en GPU fue aproximadamente 242 veces más rápido que en la CPU. Estos resultados indican una mejora significativa en el rendimiento al utilizar la GPU para este cálculo, lo que demuestra la eficacia de la paralelización en comparación con la ejecución secuencial en CPU. Además, al considerar también las transferencias de datos entre la CPU y la GPU, la aceleración sigue siendo la misma, lo que sugiere que el tiempo dedicado a estas transferencias es insignificante en comparación con el tiempo total de cálculo.
