Universidad Nacional de San Agustín, Arequipa
Algoritmos Paralelos, 8vo emestre, 2017

Thread Granularity

Norman Patrick Harvey Arce

July 12, 2017

Docente ..... Alvaro Mamani Aliaga

1 Introducción

CUDA ofrece memorias / registros locales. Usando estos Eficientemente reduce el acceso a la memoria global y mejora actuación. Pero requiere un nuevo diseño del algoritmo (re-diseño).

Los bloques y registros tienen memoria limitada. Si los datos exceden este requerimiento para la memoria compartida, entonces los datos deben dividirse en varias piezas.

La capacidad de razonar sobre la limitación del hardware al momento de desarrollar una aplicación es un aspecto clave del pensamiento computacional.

Los algoritmos de mosaico(tiled algorithms) a menudo aumentan el rendimiento. Pero la clave es explotar la localidad de datos(data locality).

2 Multiplicación de Matrices

Para la multiplicación de matrices, el objetivo de estos algoritmos es dividir la ejecución del kernel en fases de modo que los accesos a los datos en cada fase se centra en un subconjunto (mosaico) de las 2 matrices de entrada.

Ejecutando Thread/Blocks

Los subprocesos se asignan a multiprocesadores de transmisión (SM: Streamming Multiprocessor) en granularidad de bloques

- Hasta 8 bloques a cada SM como el recurso lo permita
- Cada SM puede tener hasta 768 hilos
  - Podría ser 256 (hilos/bloques) $\times$ 3 bloques
  - O 128 (hilos / bloque) $\times$ 6 bloques, etc.
- Los hilos se ejecutan simultáneamente
  - Cada SM mantiene números de identificación de hilo / bloque
  - Cada SM gestiona / programa la ejecución del hilo

Programación de hilos

- Cada bloque se ejecuta como un warp(conjunto) de 32 hilos
  - Una decisión de aplicación, que no forma parte del modelo de programación CUDA
  - Los warps programan unidades en un SM.
- Si se asignan 3 bloques a un SM y cada bloque tiene 256 hilos, ¿Cuántos warps hay en un SM?
  - Cada bloque se divide en $256/32 = 8$ warps
  - Hay $8 \times 3 = 24$ warps
- Cada SM implementa la programación de warps
  - En cualquier momento, sólo se ejecuta una de los warps por un SM
  - Los warps cuya siguiente instrucción tiene sus operandos listos para su ejecución, son elegibles para ser ejecutados
  - Los warps elegibles se seleccionan para su ejecución con una política de programación priorizada.
- Todos los hilos de un warp ejecutan la misma instrucción cuando son seleccionados

Problemas de la granularidad de Bloques

Para la multiplicación matricial utilizando bloques múltiples, ¿se debe usar 8×8, 16×16 ó 32×32 bloques?

- Para 8×8, tenemos 64 hilos por bloque. Puesto que cada SM puede tomar hasta 768 hilos, hay 12 bloques. Sin embargo, cada SM sólo puede tomar hasta 8 bloques, sólo 512 hilos pueden ir en cada SM.
- Para 16×16, tenemos 256 hilos por bloque. Puesto que cada SM puede tener hasta 768 hilos, puede tardar hasta 3 bloques y alcanzar la capacidad total a menos que otras consideraciones sobre recursos se anulen.
- Para 32×32, tenemos 1024 hilos por bloque. Ni siquiera uno puede encajar en un SM.

Thread Granularity: calcula 1×2 ó 1×4 bloques por hilo.

3 Código

```
#define N 512
#define TILE_WIDTH 16
#include <stdio.h>
#include <iostream>

using namespace std;

__global__ void matrix_mult(int* A, int* B, int* C, int ancho)
{
int tmp = 0;
int columna = blockIdx.x*TILE_WIDTH + threadIdx.x;
int fila = blockIdx.y*TILE_WIDTH + threadIdx.y;
if(columna < ancho && fila < ancho)
 {
  for (int k = 0; k < ancho; k++)
  tmp += A[fila * ancho + k] * B[k * ancho + columna];
  C[fila * ancho + columna] = tmp;
 }
}

int main()
{

int a[N][N], b[N][N], c[N][N];
int *dev_a, *dev_b, *dev_c;

int size = N * N * sizeof(int);

cudaMalloc((void **) &dev_a, size);
cudaMalloc((void **) &dev_b, size);
cudaMalloc((void **) &dev_c, size);

cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid((int)ceil(N/dimBlock.x), (int)ceil(N/dimBlock.y));

matrix_mult<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
}
```

4 Explicación Teórica

Si tenemos una matriz de salida C, el cálculo de dos elementos en C en mosaicos adyacentes utiliza la misma fila de A. Con el algoritmo de mosaico original, la misma fila A es cargada redundantemente por los dos bloques asignados para generar estos dos bloques C.
Se puede eliminar esta redundancia fusionando los dos bloques de hilos en uno. Cada hilo en el nuevo bloque de hilos ahora calcula dos elementos C.

Esto se hace revisando el kernel de manera que dos producto-punto son calculados por el bucle más interno del kernel. Ambos productos punto utilizan la misma fila de A pero diferentes columnas de B.

1. Esto reduce el acceso a la memoria global en un cuarto
2. Incrementa el número de instrucciones independientes
3. El nuevo kernel utilizar más registro y memoria compartida

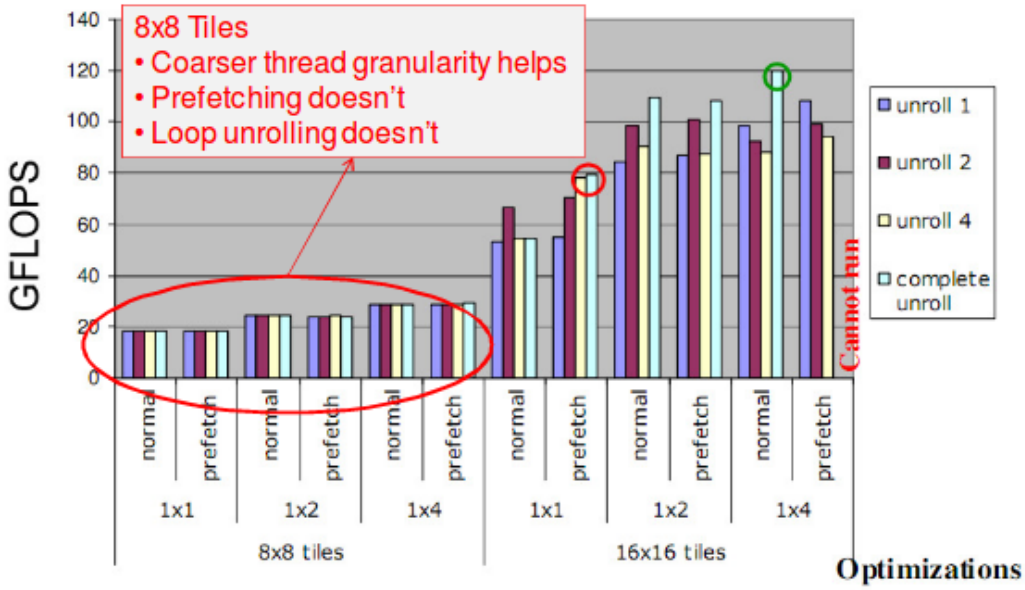
El inconveniente potencial es que el nuevo kernel utiliza ahora más registros y memoria compartida. El número de bloques que se pueden ejecutar en cada SM puede disminuir.

Para un tamaño de matriz dado, esto también reduce el número total de bloques de hilos a la mitad, lo que puede resultar en una cantidad insuficiente de paralelismo para matrices de dimensiones más pequeñas.

En la práctica, la combinación de hasta cuatro bloques horizontales adyacentes para calcular mosaicos horizontales adyacentes mejora significativamente el performance para multiplicaciones de matrices grandes (2048 $\times$ 2048 ó más).

5 Comparación Gráfica

Figure 1: Para matrices pequeñas el performances es parecido



Referencia: Imágenes de
http://courses.engr.illinois.edu/ece498/ al/textbook/Chapter5-CudaPerformance.pdf

Figure 2: Para matrices más grandes, el rendimiento es mejor con Granularidad de Hilos