Algoritmos Paralelos
Ejercicios CUDA

Lorena Castillo - Grimaldo Dávila

Junio 2017

- **3.1** Una adición de matrices toma dos matrices de entrada B y C y produce una matriz de salida A. Cada elemento de la matriz de salida A es la suma de los elementos correspondientes de las matrices de entrada B y C, es decir, $A[i][j] == B[i][j] + C[i][j]$. Para simplificar, sólo manejaremos matrices cuadradas de los cuales los elementos son números de punto flotante. Escribe un kernel de suma de matriz y la función host stub que se puede llamar con cuatro parámetros: puntero a la matriz salida, puntero a la primera matriz de entrada, puntero a la segunda matriz de entrada, y el número de elementos en cada dimensión. Utilizar las siguientes instrucciones:

  - Escribe la función host stub asignando memoria para las matrices de entrada y de salida, transfiriendo datos de entrada al dispositivo, lanza el kernel, transfiriendo los datos de salida al host y liberando la memoria del dispositivo para los datos de entrada y salida. Deja los parámetros de configuración de ejecución abiertos para este paso.

```
void MatrizAdd(float *A, float* B, float* C, int n)
{
        int size = n*n * sizeof(float);
        float *d_A, *d_B, *d_C;
        cudaMalloc((void **) &d_A, size);
        cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &B_d, size);
        cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &d_C, size);
        MatrizAddKernel<<<ceil(n*n/2560), 256>>>(d_A, d_B, d_C, n);
        cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
        // Free device memory for A, B, C
        cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

Figure 1: Ejercicio 3.a

  - Escriba un kernel que tenga cada hilo produciendo un elemento de

matriz de salida. Rellene los parámetros de configuración de ejecución para el diseño.

```
__global__ MatrizAddKernel(float *A, float* B, float* C, int n)
{
                int i=threadIdx+blockDim*blockid.x;
                c[i]=A[i]+B[i];
}
```

Figure 2: Ejercicio 3.b

– Escriba un kernel que tenga cada hilo produciendo una fila de matriz de salida. Rellene los parámetros de configuración de ejecución para el diseño.

```
__global__ MatrizAddKernel(float *A, float* B, float* C, int n)
{
                int i=n*blockid.x;
                for(int j=0,j<n;j++)
                {
                        c[i+j]=A[i+j]+B[i+j];
                }
}

void MatrizAdd(float *A, float* B, float* C, int n)
{
        int size = n*n * sizeof(float);
        float *d_A, *d_B, *d_C;
        cudaMalloc((void **) &d_A, size);
        cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &B_d, size);
        cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &d_C, size);
        MatrizAddKernel<<<n,1>>>(d_A, d_B, d_C, n);
        cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
        cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

Figure 3: Ejercicio 3.c

– Escribe un kernel que tenga cada hilo produciendo una columna de matriz de salida. Rellene los parámetros de configuración de ejecución para el diseño.

```
__global__ MatrizAddKernel(float *A, float* B, float* C, int n)
{
                int i=blockid.x;
                for(int j=0,j<n;j++)
                {
                        c[i+n*j]=A[i+n*j]+B[i+n*j];
                }
}

void MatrizAdd(float *A, float* B, float* C, int n)
{
        int size = n*n * sizeof(float);
        float *d_A, *d_B, *d_C;
        cudaMalloc((void **) &d_A, size);
        cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &B_d, size);
        cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &d_C, size);
        MatrizAddKernel<<<n,1>>>(d_A, d_B, d_C, n);
        cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
        cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

Figure 4: Ejercicio 3.d

  - Analizar los pros y los contras de cada diseño de kernel anterior.

    Los pros de la primer parte en donde se manda gran cantidad de hilos para hacer el cálculo es que son más rápidos debido a que existen más hilos trabajando en el proceso. En cambio, procesar por filas y por columnas con cada hilo es más lento debido a que hay menos hilos trabajando en el proceso del cálculo.

- **3.2** Una multiplicación vector-matriz toma una matriz de entrada B y un vector C y produce un vector de salida A. Cada elemento del vector de salida A es el producto punto de una fila de la matriz de entrada B y C. Por simplicidad, sólo manejaremos matrices cuadradas de las cuales los elementos son números de punto flotante. Escribe un kernel de multiplicación de vector-matriz y la función de host stub que se puede llamar con cuatro parámetros: puntero a la matriz de salida, puntero a la matriz de entrada, puntero al vector de entrada y el número de elementos en cada dimensión.

```
__global__ MatrizAddKernel(float *A, float* B, float* C, int n)
{
    int i=n*blockid.x;
    for(int j=0,j<n;j++)
    {
      c[blockid] += (A[i+j]*B[blockid]);
    }
}

void MatrizAdd(float *A, float* B, float* C, int n)
{
        int size = n*n * sizeof(float);
        float *d_A, *d_B, *d_C;
        cudaMalloc((void **) &d_A, size);
        cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &B_d, size);
        cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &d_C, size);
        vecAddKernel<<<n,1>>>(d_A, d_B, d_C, n);
        cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
        // Free device memory for A, B, C
        cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

Figure 5: Ejercicio 3.2

- **3.3** Un nuevo interno de verano se sintió frustrado con CUDA. Se queja de que CUDA es muy tedioso: tuvo que declarar muchas funciones que planea ejecutar tanto en el host como en el dispositivo dos veces, una vez como función de host y una vez como función de dispositivo. ¿Cuál es su respuesta?

  -Puede parecer tedioso al inicio pero una vez que vas usándolo te das cuenta de que es necesario separar la parte de CPU de la parte de GPU para manejar mejor el proceso. También es necesario copiar los valores de las variables de CPU a GPU ya que el cálculo que se realiza en los kernels es respecto a variables alojadas en el GPU. Al final de cuentas todo resulta muy beneficioso porque puedes tener cientos de hilos trabajando a la vez y es muy eficiente hacer esto en comparación al manejo de sólo CPU.

- **3.5** Si necesitamos usar cada hilo para calcular un elemento de salida de una adición de vector, ¿cuál sería la expresión para asignar los índices de hilo / bloque al índice de datos?

$$i == blockIdx.x * blockDim.x + threadIdx.x;$$

- **3.6** Queremos usar cada hilo para calcular dos elementos (adyacentes) de

una adición vectorial, supongamos que la variable i debería ser el índice del primer elemento a ser procesado por un hilo. ¿Cuál sería la expresión para asignar los índices de hilo / bloque al índice de datos?

$$i == (blockIdx.x * blockDim.x + threadIdx.x) * 2;$$

- **3.7** Para una adición vectorial, suponga que la longitud del vector es 2000, cada hilo calcula un elemento de salida y el tamaño del bloque es de 512 hilos. ¿Cuántos hilos habrá en el grid?

$$Rpta : 2048$$

- **3.** Si el SM de un dispositivo CUDA puede tomar hasta 1536 hilos y hasta 4 bloques de hilos. ¿Cuál de las siguientes configuraciones de bloque resultaría en el mayor número de subprocesos en el SM?

$$Rpta : 512 hilosporbloque.$$

- **4.** Para una adición vectorial, suponga que la longitud del vector es 2000, cada hilo calcula un elemento de salida y el tamaño del bloque es de 512 hilos. ¿Cuántos hilos habrá en el grid?

$$Rpta : 2048$$

- **5.** Con referencia a la pregunta anterior, ¿cuántos warps se espera que tengan divergencia debido al control de límites en la longitud del vector?

$$Rpta : 1$$

- **6.** Se necesita escribir un kernel que funcione en una imagen de tamaño 400 900 píxeles. Se desea asignar un hilo a cada píxel. Se desea que sus bloques de hilos sean cuadrados y utilice el número máximo de hilos por bloque posible en el dispositivo (su dispositivo tiene capacidad de cálculo 3.0). ¿Cómo seleccionarías las dimensiones de la cuadrícula y las dimensiones del bloque de tu kernel?

$$blockdim = 32 * 32 = 1024\, threads$$
Grid dim = ceil (400/32) * ceil(900/32) = 13* 29

1 Conclusiones

Al trabajar con GPU podemos utilizar cientos de hilos a la vez y si los usamos de una manera adecuada esto puede ser muy beneficioso en el rendimiento. [1]

References

[1] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2016.