Arquitectura de Computadores

- Pablo Rayón Zapater
- Fernando Pérez Ballesteros
- José María Fernández

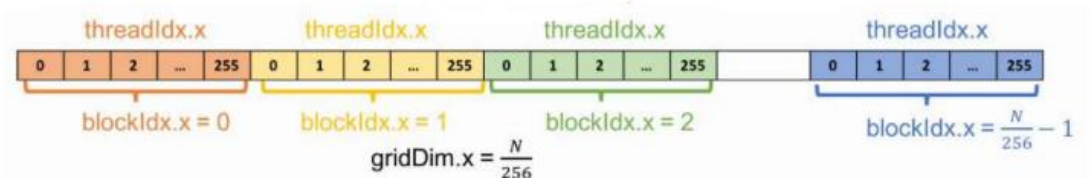
Práctica 5

UNIVERSIDAD NEBRIJA

Índice

Ejercicio 1

Generar un código con el ejemplo anterior de la suma de vectores, añadiendo múltiples bloques que aumenten el grado de paralelización. En el caso de no poder probar el código en CUDA explicar en la memoria claramente los pasos seguidos para la implementación.



Se ha diseñado un programa que realiza da suma de vectores, esto por medio de la paralelización de operaciones por medio de aceleramiento por GPU, para dividir este problema se reparte cada operación en hilos, los cuales se encargan de acceder a cada posición de cada vector, realizar la operación y almacenar el resultado en el índice correspondiente.

En primer lugar, se muestra la función que va a ser ejecutada en la gráfica y será empleada de manera paralela por todos los núcleos requeridos.

```
__global__ void vector_add(float *out, float *a, float *b, int n, unsigned int *Nth)
{
    int index = threadIdx.x;
    int stride = blockDim.x;
    Nth[0] = stride * gridDim.x;
    printf("Thread index: %d From block number: %d\n", index, blockIdx.x);
    for (int i = index; i < n; i += stride)
    {
        out[i] = a[i] + b[i];
    }
}
```

Los parámetros de entrada de esta función son los siguientes:

Float * out: Vector de números en coma flotante que almacenará el resultado de la operación
Float * a y Float * b: Vectores de números en coma flotante que contienen los datos a operar.
Int n: Número entero que limitará el tamaño de los vectores
Unsigned int Nth: Variable empleada para calcular el numero total de hilos del programa.

Se tiene que pasar como parámetro de entrada el vector out ya que, para poder almacenar datos en él, se ha tenido que reservarle memoria en la función main previamente, esta reserva de memoria se tiene que realizar en los bancos de memoria de la GPU y también en la memoria principal de la máquina, donde posteriormente se copiará la primera instancia mencionada para poder ser almacenada.
A modo de traza se ha sacado por pantalla el número del thread que está ejecutando cada instrucción y el bloque al que este pertenece, el número del thread aumenta dentro de la misma función mientras que el número de bloque es actualizado cada vez que la función es llamada por los núcleos de la GPU.

Después se realiza la operatoria de manera convencional por medio de un bucle for que itera por cada posición de nuestros vectores y opera con cada posición escalar de estos.

En la imagen inferior se muestran la declaración de variables que es necesaria para que la función ejecutada en la gráfica sea capaz de operar y traspasar sus resultados a la memoria principal.

```
int main(int argc, char **argv)
{
    float *a, *b, *out;
    float *d_a, *d_b, *d_out;
    unsigned int *d_nth;
    int manBlockID;
```

A continuación, se presenta el código por medio del cual se puede indicar al programa un numero personalizado de bloques y de hilos por bloque, en caso de que se desee, en caso contrario se puede indicar que el programa opere con unos valores predeterminados proporcionados en el enunciado de la tarea. Esto se indica por medio de los parámetros de entrada en la línea de ejecución del programa, en caso de no poner ningún parámetro se mostrará un cuadro de ayuda que indicará la sintaxis correcta para poder ejecutar el binario.

```
if (argc < 2)
{
    printf("Utilizacion del programa : './ejecutable parametro'\nSiendo parametro :\n 0 :
    exit(0);
}
else
{
    // Allocate host memory
    a = (float *)malloc(sizeof(float) * N);
    b = (float *)malloc(sizeof(float) * N);
    out = (float *)malloc(sizeof(float) * N);
    unsigned int *T_NThreads;
    int blockID;
    int nThreads;
    if (atoi(argv[1]) == 0)
    {
        blockID = N / 256;
        nThreads = 256;
    }
    else if (atoi(argv[1]) == 1)
    {
        printf("Ingrese el numero de bloques : ");
        scanf("%d", &manBlockID);
        blockID = manBlockID;
        printf("Ingrese el numero de hilos por bloque : ");
        scanf("%d", &nThreads);
    }
    else
    {
        printf("Parametro invalido\n");
        exit(0);
    }
```

Se realiza la reserva de memoria tanto en el host como en el device una vez comprobado que la sintaxis del programa es correcta y que se cumplen con los requerimientos de numero de hilos y bloques, para así no emplear recursos para un programa que no será ejecutado.

Por último, se adjunta la parte de código que realiza tola la parte de transferencias de memoria entre GPU y la máquina completa, también se encarga de realizar las llamadas correspondientes a la función que realiza la operatoria y liberar memoria en ambos bancos de memoria.

```
    for (int i = 0; i < N; i++)
    {
        a[i] = 1.0f;
        b[i] = 2.0f;
    }
    // Allocate device memory
    cudaMalloc((void **)&d_a, sizeof(float) * N);
    cudaMalloc((void **)&d_b, sizeof(float) * N);
    cudaMalloc((void **)&d_out, sizeof(float) * N);
    cudaMalloc((void **)&d_nth, sizeof(int));

    // Transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);
    // Executing kernel
    vector_add<<<blockID, nThreads>>>(d_out, d_a, d_b, N, d_nth);
    // Transfer data back to host memory
    cudaMemcpy(out, d_out, sizeof(float) * N,
               cudaMemcpyDeviceToHost);
    cudaMemcpy(&T_NThreads, d_nth, sizeof(int),
               cudaMemcpyDeviceToHost);
    printf("Number of Blocks: %d \n", blockID);
    printf("Number of Threads per block: %d \n", nThreads);
    printf("Total number of threads: %x \n", T_NThreads[0]);
    // Verification
    for (int i = 0; i < N; i++)
    {
        assert(fabs(out[i] - a[i] - b[i]) < MAX_ERR);
    }
    printf("PASSED\n");
    // Deallocate device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_out);
    // Deallocate host memory
    free(a);
    free(b);
    free(out);
}
```

Al comienzo del código se inicializan los arrays en el host, arrays que han sido alocados previamente en la memoria de este. Posteriormente se realiza una operativa similar, salvo que esta vez se aplica sobre la memoria de la GPU, reservando primeramente memoria en los bancos de memoria de esta para los vectores con los que se operará, y posteriormente se inicializan copiando los contenidos de los vectores en memoria del host.
Después se llama a la función que ejecutará la gráfica con los dos parámetros personalizados, el número de bloques y el número de hilos por bloque, esto representará el numero de operaciones que se realizarán; para extraer la información de la memoria de la GPU a la memoria principal del host, se tienen que copiar los contenidos de las posiciones de memoria a las que apuntan los punteros declarados y reservados en la GPU a los vectores que hemos reservado en la memoria del host. Por último, a modo de traza se indican los bloques empleados, el número de hilos por bloque empleados y el numero total de hilos para posteriormente liberar la memoria en ambos bancos de memoria, tanto en el host como en el device, para evitar así posibles errores de segmentación.