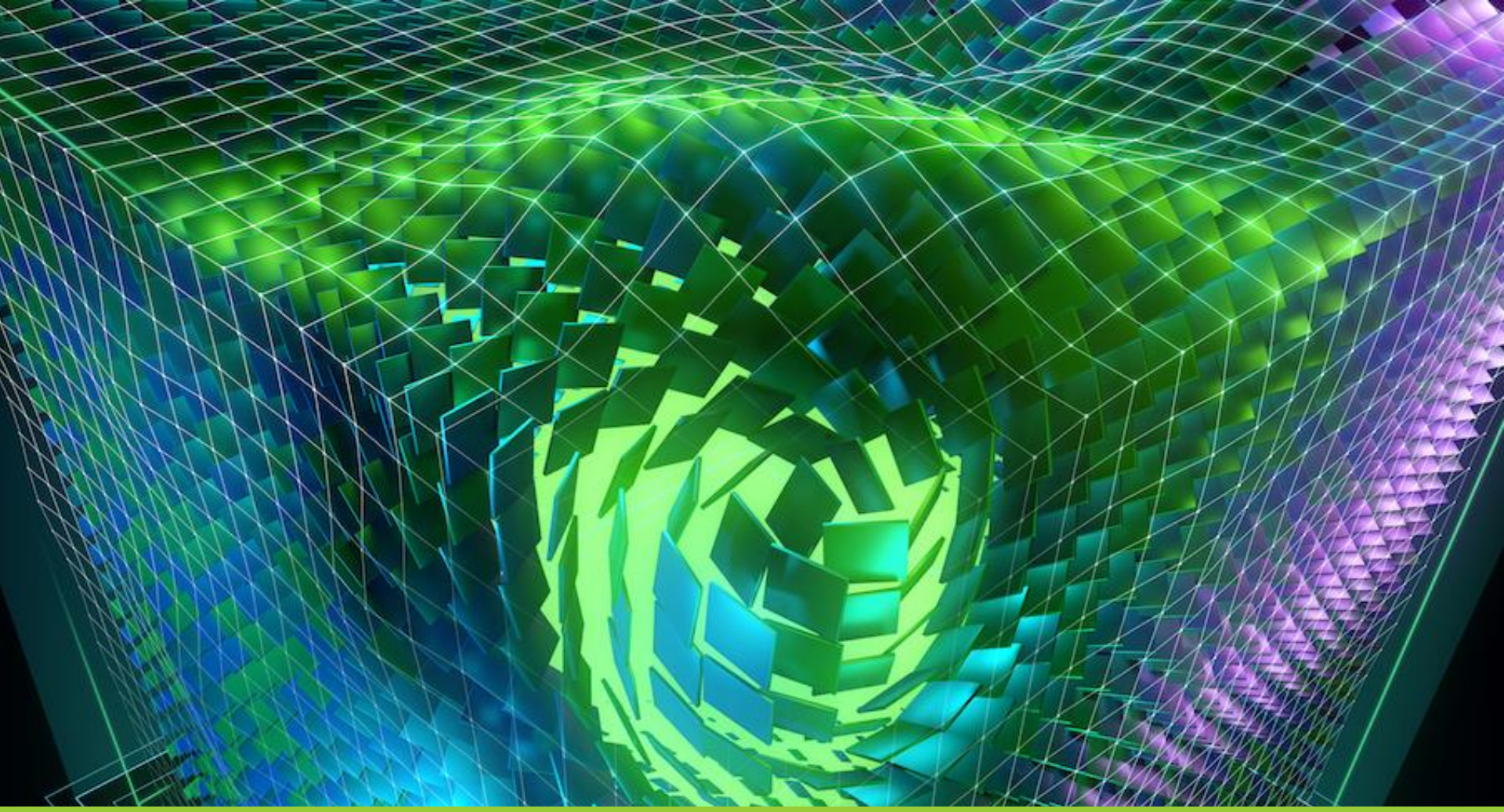

INF351 – Computación de Alto Desempeño

CUDA Kernel

PROF. ÁLVARO SALINAS

Kernel – Declaración

Como vimos anteriormente, un kernel es una función declarada de forma especial que será ejecutada en paralelo.

Para declarar y definir un kernel necesitamos utilizar el especificador *__global__*, mientras que el resto sigue la misma lógica que una función normal de C.

Un kernel siempre será de tipo *void*, pues no puede retornar un valor. Si queremos obtener resultados, estamos obligados a pasar parámetros por referencia.

```
__global__ void Hello_World_kernel() {
    printf("Hello World\n");
}
```

Kernel – Invocación

La llamada a un kernel se realiza desde el código ejecutado por la CPU. La invocación es bastante similar a la de una función común de C, a excepción de la presencia de la sintaxis de configuración <<<…>>>.

Esta sintaxis contiene las dimensiones de la grid y los bloques de la siguiente forma: *<<<grid_size, block_size>>>*.

Si las configuraciones utilizadas son unidimensionales, es posible utilizar enteros para especificarlas. Si se requieren más dimensiones se debe acudir a la estructura *dim3* de CUDA.

```
int main() {
    int grid_size = 2;
    dim3 block_size(32,16);
    kernel<<<grid_size, block_size>>>();
    return 0;
}
```

Kernel – Dimensiones

Tal como se les recomendó, es buena idea utilizar configuraciones unidimensionales y fijar el tamaño de los bloques, calculando el tamaño de la grid a partir de la cantidad de datos a procesar.

Asumiendo N datos y 256 hebras por bloque, el tamaño de la grid se puede calcular como *(int)ceil((float)N/256)*.

Como esto nos da una cantidad de hebras suficiente para procesar cada dato, pero a menos que N sea múltiplo de 256 sobrarán hebras, es necesario asegurarnos al comienzo del kernel de no acceder a direcciones de memoria inexistentes.

Kernel – Dimensiones

```
__global__ void kernel(int *dev_array, int N) {
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N)
        dev_array[tId] = 1;
}

int main() {
    int size = N;
    int block_size = 256;
    int grid_size = (int)ceil((float)N / block_size);

    int* dev_array;
    cudaMalloc(&dev_array, N * sizeof(int));
    kernel<<<grid_size, block_size>>>(dev_array, N);
    cudaFree(dev_array);
    return 0;
}
```

Kernel – Parámetros

Si bien los parámetros de un kernel pueden ser manejados de la misma forma que en una función de C, es recomendable tener una consideración especial.

Es buena práctica declarar aquellos parámetros que son punteros a memoria de solo lectura como *const* y con la etiqueta __*restrict*__.

Esto la mayoría de las veces será una ayuda para el compilador y traerá consigo beneficios en cuanto al desempeño de nuestra aplicación.

```
__global__ void copy_kernel(const int* __restrict__ dev_a,
                            int* dev_b, int N) {
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N)
        dev_b[tId] = dev_a[tId];
}
```

Instrucciones en la GPU

Las instrucciones en GPU siguen un orden de ejecución FIFO, en donde cada instrucción debe esperar a que la anterior finalice.

Si bien existen formas de realizar ejecuciones en paralelo bajo ciertos requerimientos, éstas serán vistas más adelante en el curso.

```
int main() {
    int size = N;
    int block_size = 256;
    int grid_size = (int)ceil((float)N / block_size);

    int* host_array = new int[N];
    int* dev_array;
    cudaMalloc(&dev_array, N * sizeof(int));
    kernel<<<grid_size, block_size>>>(dev_array, N);
    cudaMemcpy(host_array, dev_array, N * sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(dev_array);
    return 0;
}
```

Sincronía

A pesar de que en GPU las instrucciones se ejecutan bajo un FIFO sincrónico, desde el punto de vista de la CPU la historia es un poco distinta.

Si bien los manejos de memoria (cudaMalloc, cudaFree, cudaMemcpy, entre otros) son manejadas de forma sincrónica, la invocación de kernels es asincrónica en la CPU.

```
__global__ void kernel() {
    printf("Esto va segundo\n");
}

int main() {
    kernel<<<1,1>>>();
    std::cout << "Esto va primero" << std::endl;
    return 0;
}
```

cudaDeviceSynchronize()

Esta instrucción nos permite bloquear la CPU hasta que la GPU haya terminado todas sus instrucciones pendientes.

Si bien no existen muchos casos en donde es de utilidad, es recomendable utilizarla luego de la invocación a un kernel que posea una intrucción *printf.*

```
__global__ void kernel() {
    printf("Esto va primero\n");
}

int main() {
    kernel<<<1,1>>>();
    cudaDeviceSynchronize();
    std::cout << "Esto va segundo" << std::endl;
    return 0;
}
```

Medición de tiempo

Otra aplicación que podría ser interesante para *cudaDeviceSynchronize()* es la medición del tiempo de ejecución del kernel mediante funciones de CPU, pero lamentablemente esta no es una buena forma de medirlo.

```
int main() {
    clock_t t1, t2;
    t1 = clock();
    kernel<<<1,1>>>();
    cudaDeviceSynchronize();
    t2 = clock();
    double ms = 1000.0 * (double)(t2 - t1) / CLOCKS_PER_SEC;
    std::cout << "Tiempo: " << ms << "[ms]" << std::endl;
    return 0;
}
```

Medición de tiempo

Para meidiciones correctas, CUDA nos provee un sólido manejo de eventos. Para utilizarlo debemos acudir a la estructura *cudaEvent_t* y a las siguientes funciones:

- *cudaEventCreate(cudaEvent_t* event)*
- *cudaEventRecord(cudaEvent_t event)*
- *cudaEventSynchronize(cudaEvent_t event)*
- *cudaEventElapsedTime(float* ms, cudaEvent_t start, cudaEvent_t end)*

Medición de tiempo

```
int main() {
    cudaEvent_t ct1, ct2;
    float dt;
    cudaEventCreate(&ct1);
    cudaEventCreate(&ct2);
    cudaEventRecord(ct1);
    kernel<<<1,1>>>();
    cudaEventRecord(ct2);
    cudaEventSynchronize(ct2);
    cudaEventElapsedTime(&dt, ct1, ct2);
    std::cout << "Tiempo: " << dt << "[ms]" << std::endl;
}
```

Funciones en GPU

Es posible crear funciones para su ejecución en GPU con el especificador __*device*__. Las funciones declaradas de esta forma solo pueden ser ejecutadas a través de instrucciones en GPU, i.e. dentro de un kernel.

Aparte de la etiqueta __*device*__, el resto es equivalente a cualquier función de C.

```
__device__ int suma(int a, int b) {
    return a + b;
}

__global__ void suma_kernel(const int* __restrict__ dev_a,
                            const int* __restrict__ dev_b,
                            int* dev_c, int N) {
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N)
        dev_c[tId] = suma(dev_a[tId], dev_b[tId]);
}
```

Funciones en GPU

Ya que la función reside en la GPU y es ejecutada por una hebra dentro de un kernel, para ella también son visibles las indexaciones y dimensiones de las hebras y bloques.

```
__device__ void suma(const int* __restrict__ a,
                     const int* __restrict__ b,
                     int* c, int N) {
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N)
        c[tId] = a[tId] + b[tId];
}

__global__ void suma_kernel(const int* __restrict__ dev_a,
                            const int* __restrict__ dev_b,
                            int* dev_c, int N) {
    suma(dev_a, dev_b, dev_c, N);
}
```

Overload

Las funciones de GPU pueden ser sobrecargadas para admitir ejecuciones tanto desde GPU como desde CPU. Para lograr esto simplemente debemos utilizar la doble etiqueta __*host* __ __*device*__.

```
__host__ __device__ int suma(int a, int b){
    return a + b;
}

__global__ void suma_kernel(const int* __restrict__ dev_a,
                            const int* __restrict__ dev_b,
                            int* dev_c, int N) {
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N)
        dev_c[tId] = suma(dev_a[tId], dev_b[tId]);
}

void suma_CPU(int* host_a, int* host_b, int* host_c, int N) {
    for (int i = 0; i < N; i++)
        host_c[i] = suma(host_a[i], host_b[i]);
}
```

Overload

Si necesitaramos que la función ejecute instrucciones distintas dependiendo si se llama desde CPU o GPU Podemos apoyarnos en la macro *__CUDA_ARCH__*.

```
__host__ __device__ void saludo(){
    #ifdef __CUDA_ARCH__
        printf("Saludos desde la GPU!\n");
    #else
        std::cout << "Saludos desde la CPU!" << std::endl;
    #endif
}
```