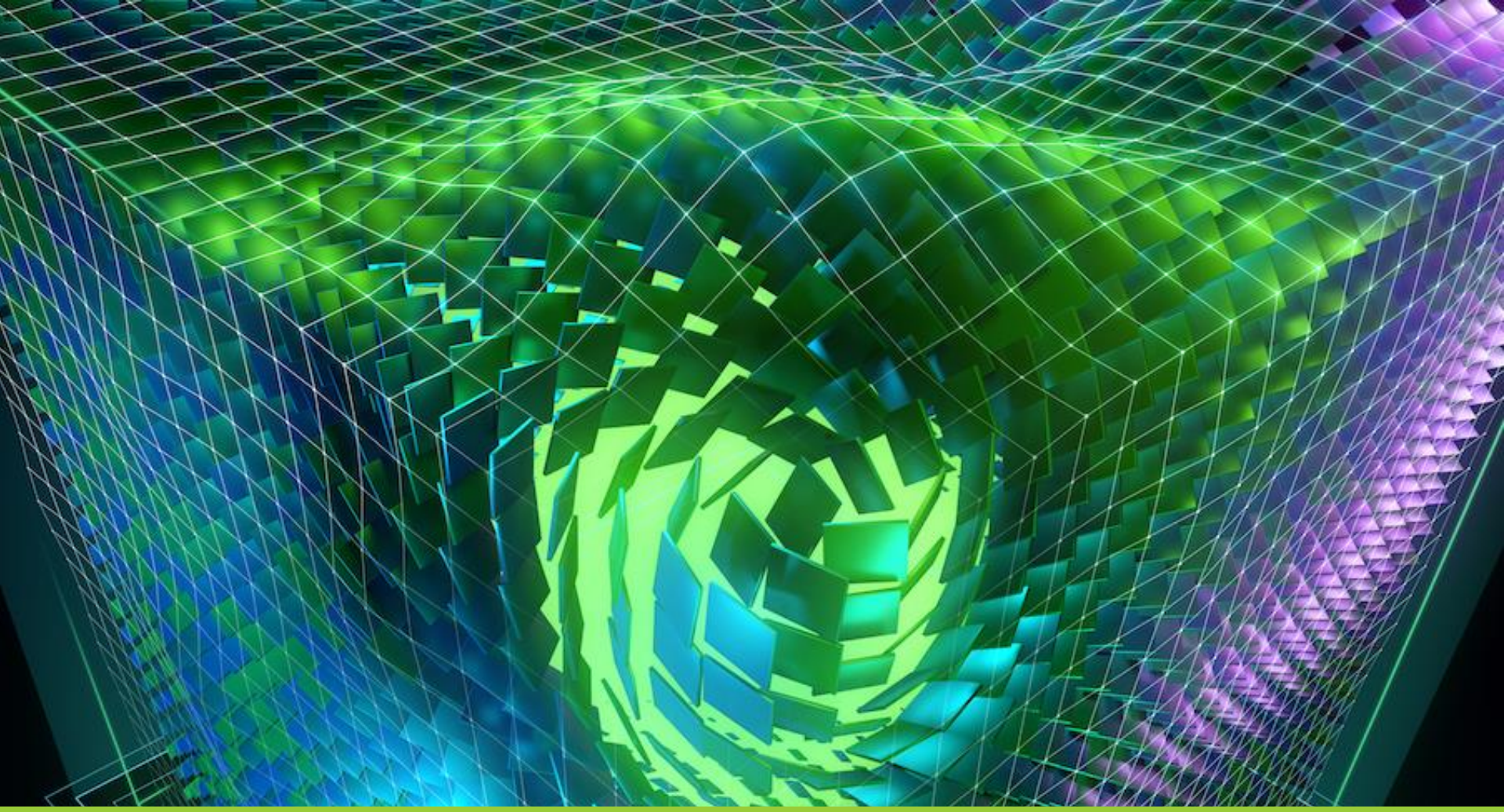


INF351 – Computación de Alto Desempeño

Arrays 1D vs Arrays 2D

PROF. ÁLVARO SALINAS

Arrays

Como sabemos, los arreglos son la manera más simple de agrupar datos. Al trabajar con CUDA, es indispensable conocer el manejo que requieren estos arrays.

Como vimos en clases anteriores, la declaración estática de arreglos es bastante sencilla, pero en la mayoría de las aplicaciones que desarrollemos no tendremos información sobre la cantidad de datos en tiempo de compilación.

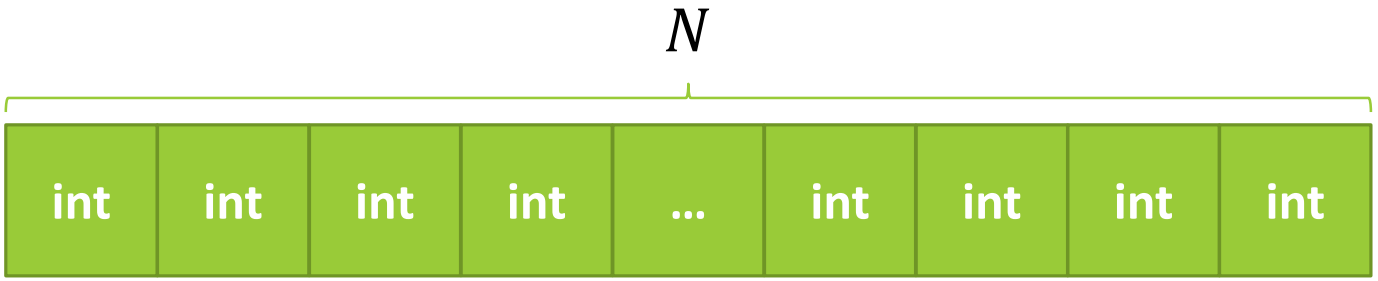
A continuación revisaremos como manejar arreglos de forma dinámica.

Array 1D

El caso más básico. Podemos crear un arreglo unidimensional en la GPU mediante:

```
int *dev_A1D;
cudaMalloc(&dev_A1D, N * sizeof(int));
```

Esto genera un arreglo de N enteros:

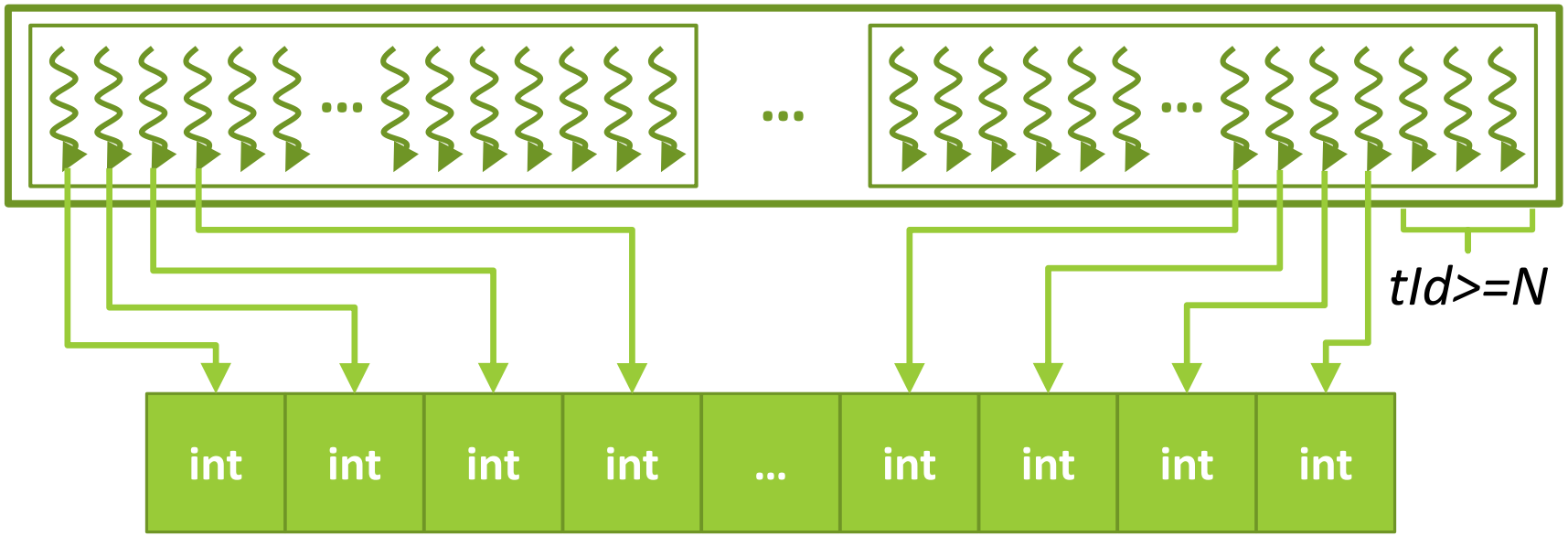


Array 1D – Bloque 1D

Cuando ambos, datos y threads, están distribuidos de forma unidimensional, la indexación resulta muy simple:

```
__global__ void kernel(int* dev_A1D, int N) {
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N)
        // dev_A1D[tId]
}
```

Array 1D – Bloque 1D



Array 1D – Bloque 2D

Si los bloques tienen dos dimensiones, entonces podemos indexar los arreglos de las siguientes formas:

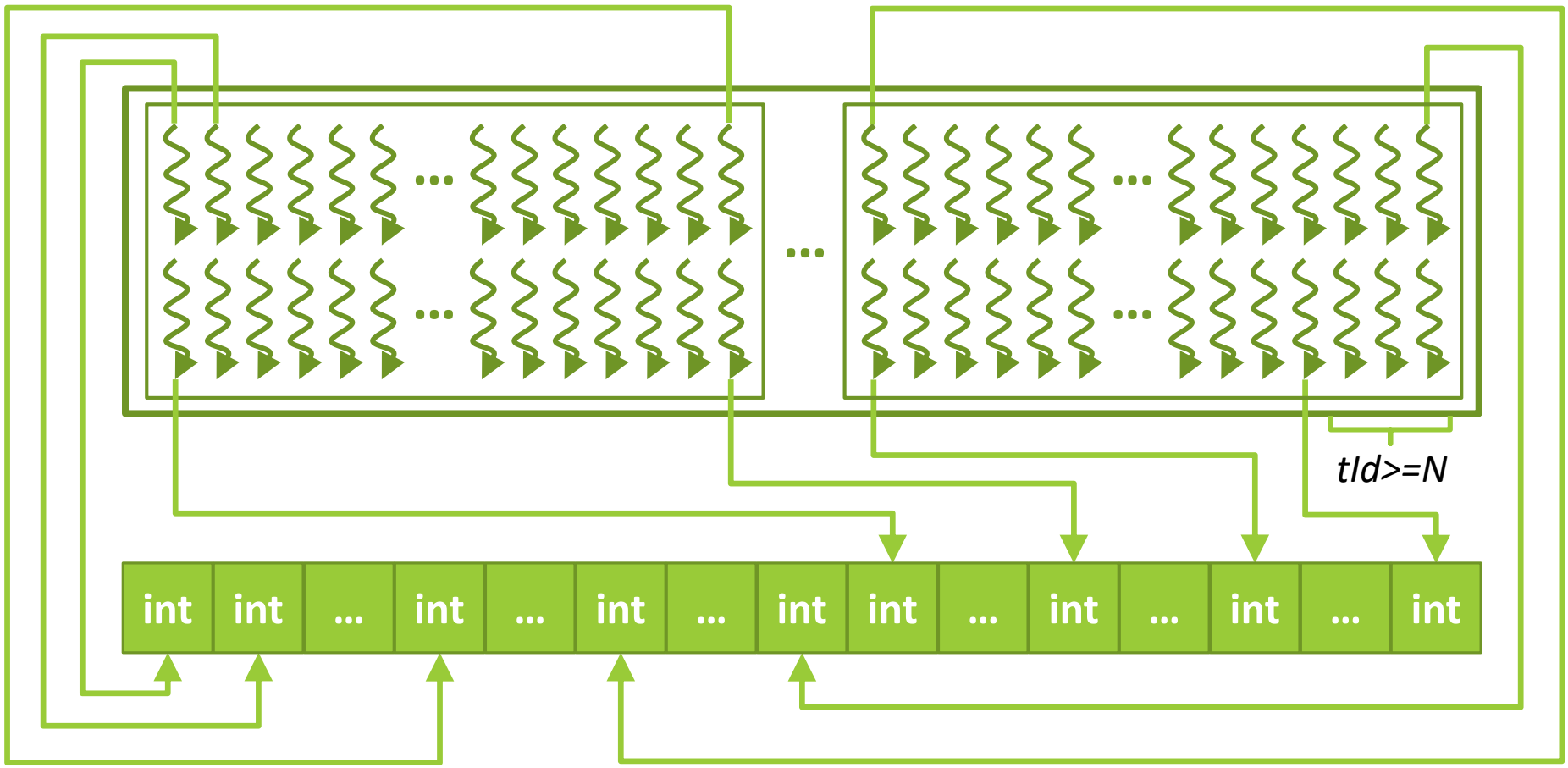
1

```
__global__ void kernel(int* dev_A1D, int N) {
    int tIdx = threadIdx.x + blockIdx.x * blockDim.x;
    int tId = tIdx + threadIdx.y * blockDim.x * gridDim.x;
    if (tId < N)
        // dev_A1D[tId]
}
```

2

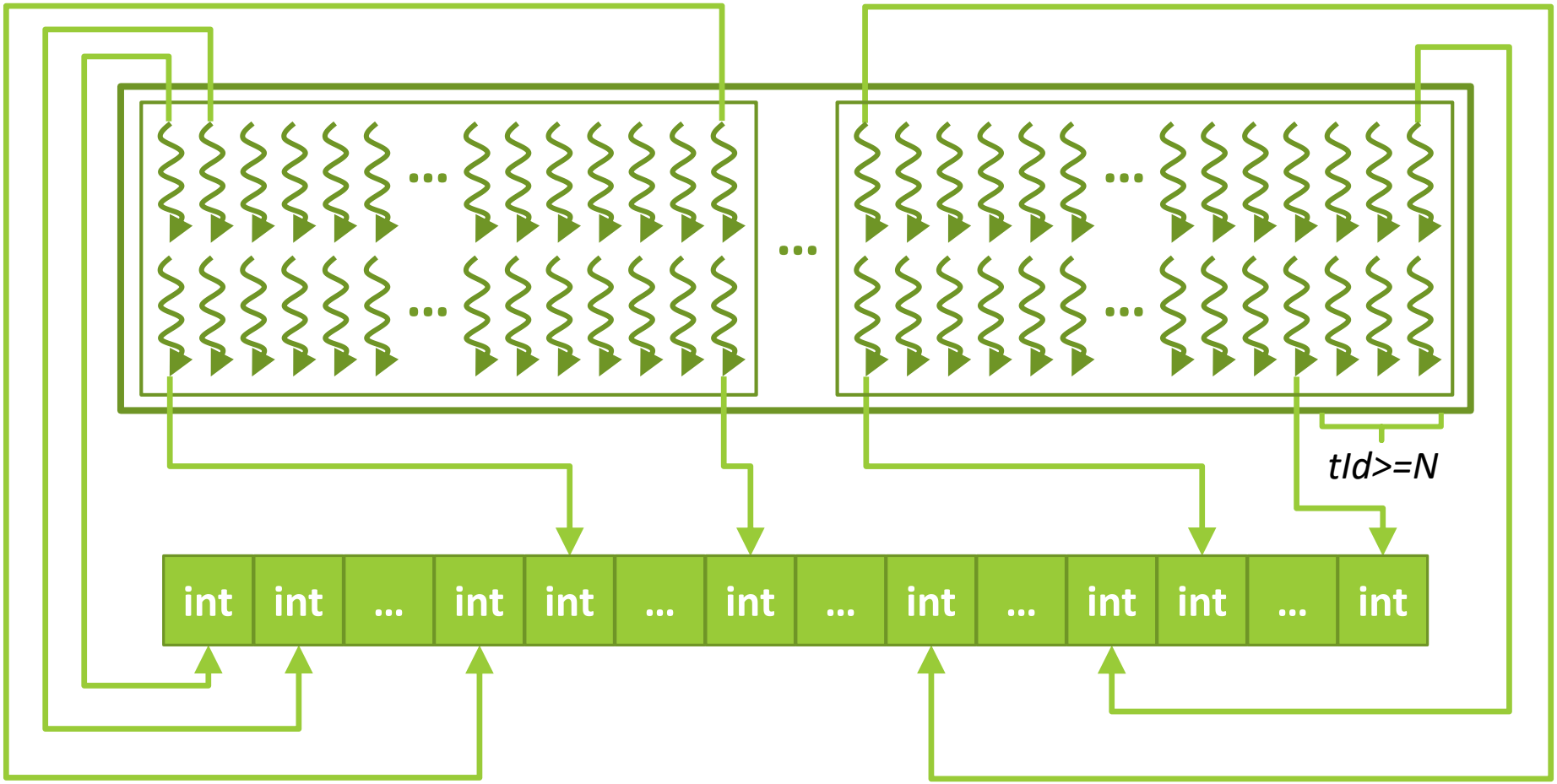
```
__global__ void kernel(int* dev_A1D, int N) {
    int tIdB = threadIdx.x + threadIdx.y * blockDim.x;
    int tId = tIdB + blockIdx.x * blockDim.x * blockDim.y;
    if (tId < N)
        // dev_A1D[tId]
}
```

Array 1D – Bloque 2D



Array 1D – Bloque 2D

2



Array 1D – Copiar y Liberar Memoria

Para copiar los valores del arreglo al host (o viceversa) solo es necesario realizar un traspaso:

```
cudaMemcpy(host_A1D, dev_A1D, N * sizeof(int), cudaMemcpyDeviceToHost);
```

Finalmente, para liberar la memoria asignada a un arreglo unidimensional basta con:

```
cudaFree(dev_A1D);
```

Array 2D

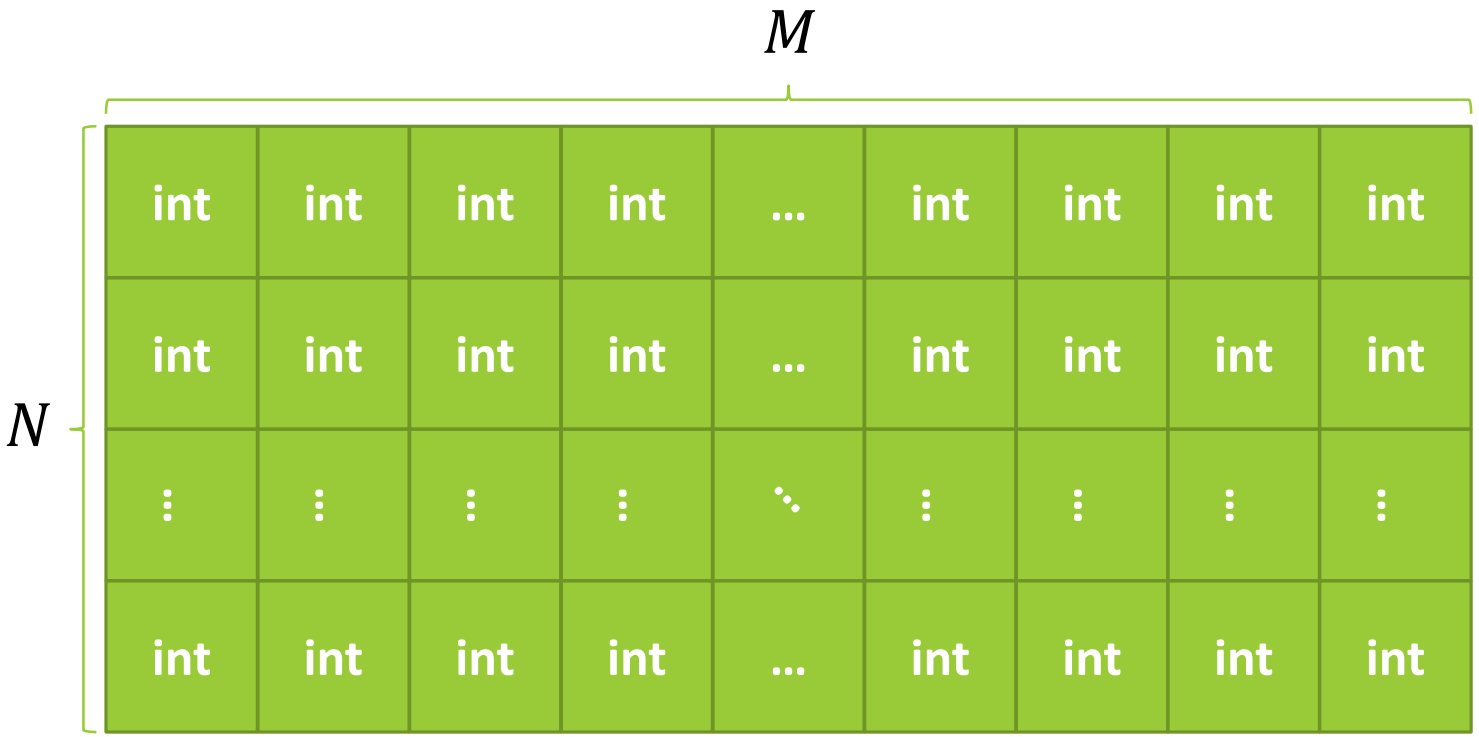
Cuando trabajamos con matrices o datos distribuidos en más de una dimensión, utilizar arreglos unidimensionales es la mejor idea. De todas formas, es bueno saber como trabajar con arreglos de dos dimensiones.

Para crear un arreglo 2D de forma dinámica es necesario realizar lo siguiente:

```
int **host_ptr = new int*[N];
int **dev_A2D;
for (int i = 0; i < N; i++)
    cudaMalloc(&host_ptr[i], M * sizeof(int));
cudaMalloc((void**)&dev_A2D, N * sizeof(int*));
cudaMemcpy(dev_A2D, host_ptr, N * sizeof(int*), cudaMemcpyHostToDevice);
```

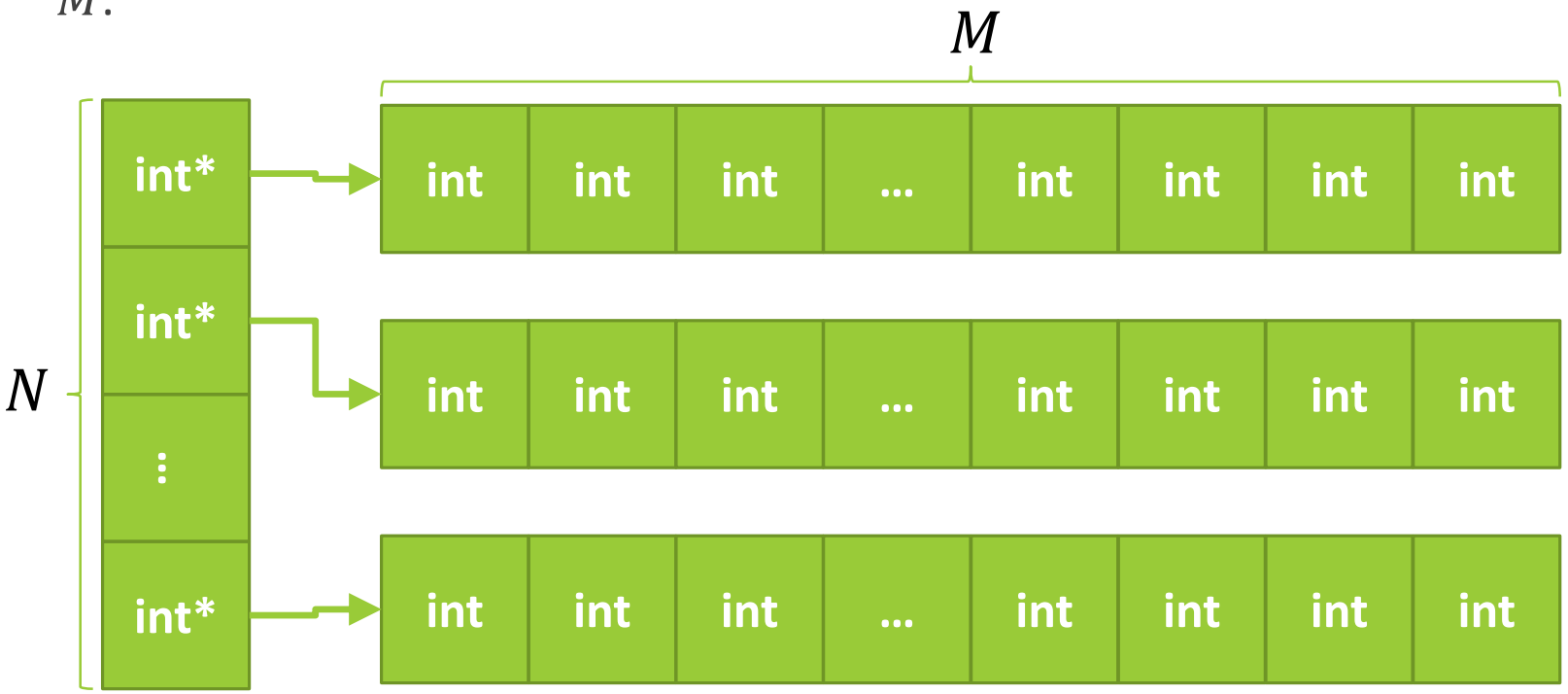
Array 2D

Lo anterior genera una matriz de tamaño $N \times M$:



Array 2D

Lamentablemente, en memoria no resulta tan ordenado, pues en realidad solo temenos N punteros que apuntan a arreglos de tamaño M.



Array 2D – Bloque 1D

Si los thread blocks son unidimensionales, entonces solo necesitamos obtener la fila y la columna a utilizar a través del identificador de la hebra y las dimensiones de la matriz:

```
__global__ void kernel(int** dev_A2D, int N, int M) {
    int tId = threadIdx.x + threadIdx.y * blockDim.x;
    if (tId < N * M) {
        int y = tId / M;
        int x = tId % M;
        // dev_A2D[y][x]
    }
}
```

Array 2D – Bloque 2D

Cuando tenemos dos dimensiones en ambas partes, podemos enfrentar dos casos:

1. Estamos seguros de que *blockDim.x * gridDim.x >= M* y de que *blockDim.y * gridDim.y >= N*.

```
__global__ void kernel(int** dev_A2D, int N, int M) {
    int tIdy = threadIdx.y + blockIdx.y * blockDim.y;
    int tIdx = threadIdx.x + blockIdx.x * blockDim.x;
    if (tIdy < N && tIdx < M)
        // dev_A2D[tIdy][tIdx]
}
```

Array 2D – Bloque 2D

2. Las dimensiones de la grid y los bloques no guardan relación con las dimensiones de la matriz.

```
__global__ void kernel(int** dev_A2D, int N, int M) {
    int tIdB = threadIdx.x + threadIdx.y * blockDim.x;
    int tId = tIdB + blockIdx.x * blockDim.x * blockDim.y;
    if (tId < N*M) {
        int y = tId / M;
        int x = tId % M;
        // dev_A2D[y][x]
    }
}
```

Array 2D – Copiar y Liberar Memoria

Para copiar los valores de la matriz al host (o viceversa) es necesario copiar los N arreglos uno por uno:

```
cudaMemcpy(host_ptr, dev_A2D, N * sizeof(int*), cudaMemcpyDeviceToHost);
for (int i = 0; i < N; i++)
    cudaMemcpy(host_A[i], host_ptr[i], M * sizeof(int), cudaMemcpyDeviceToHost);
```

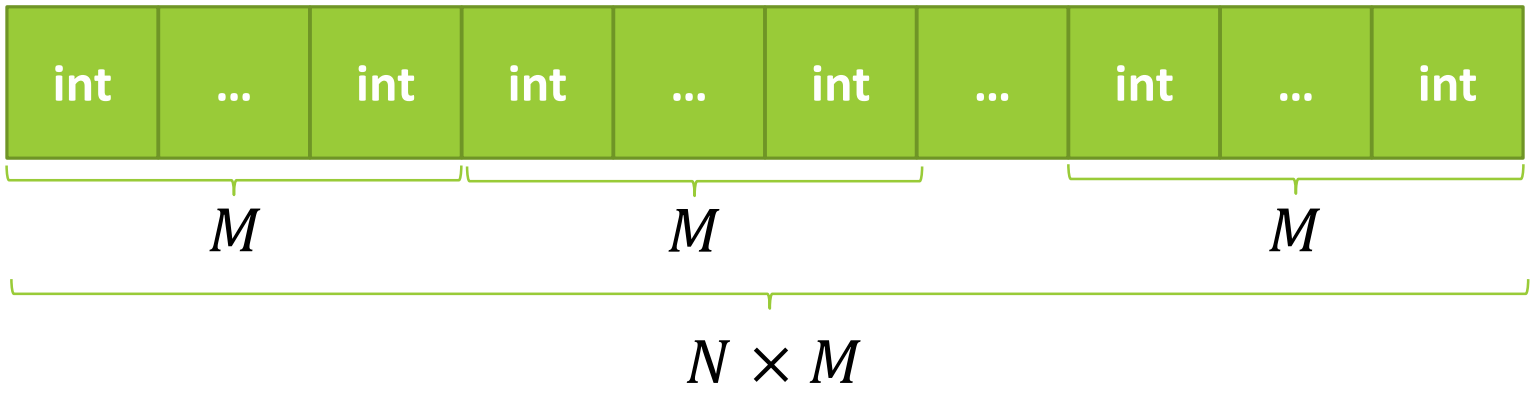
Del mismo modo, para liberar la memoria, debemos preocuparnos de cada declaración realizada.

```
cudaMemcpy(host_ptr, dev_A2D, N * sizeof(int*), cudaMemcpyDeviceToHost);
for (int i = 0; i < N; i++)
    cudaFree(host_ptr[i]);
cudaFree(dev_A2D);
delete[] host_ptr;
```

Array 2D – Linearization

La mejor manera de trabajar con datos multidimensionales en CUDA es con arreglos unidimensionales.

Solo debemos preocuparnos de organizar los datos correctamente (generalmente concatenando las filas de la matriz) y podremos trabajar tranquilamente con un arreglo unidimensional de tamaño $N \times M$.



Array 2D – Linearization

Si el procesamiento de los datos requiere conocer su fila y su columna dentro de la matriz, podemos obtenerlas fácilmente mediante:

```
__global__ void kernel(int* dev_A2D_linear, int N, int M) {
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N*M) {
        // dev_A2D_linear[tId]
        int y = tId / M;
        int x = tId % M;
        // dev_A2D_linear[x + y * M]
    }
}
```

Array 3D – Linearization

Del mismo modo, para arreglos 3D tenemos:

```
__global__ void kernel(int* dev_A3D_linear, int N, int M, int L) {
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N*M*L) {
        // dev_A3D_linear[tId]
        int z = tId / (M * L);
        int y = (tId % (M * L)) / L;
        int x = tId % L;
        // dev_A3D_linear[x + y * L + z * M * L]
    }
}
```