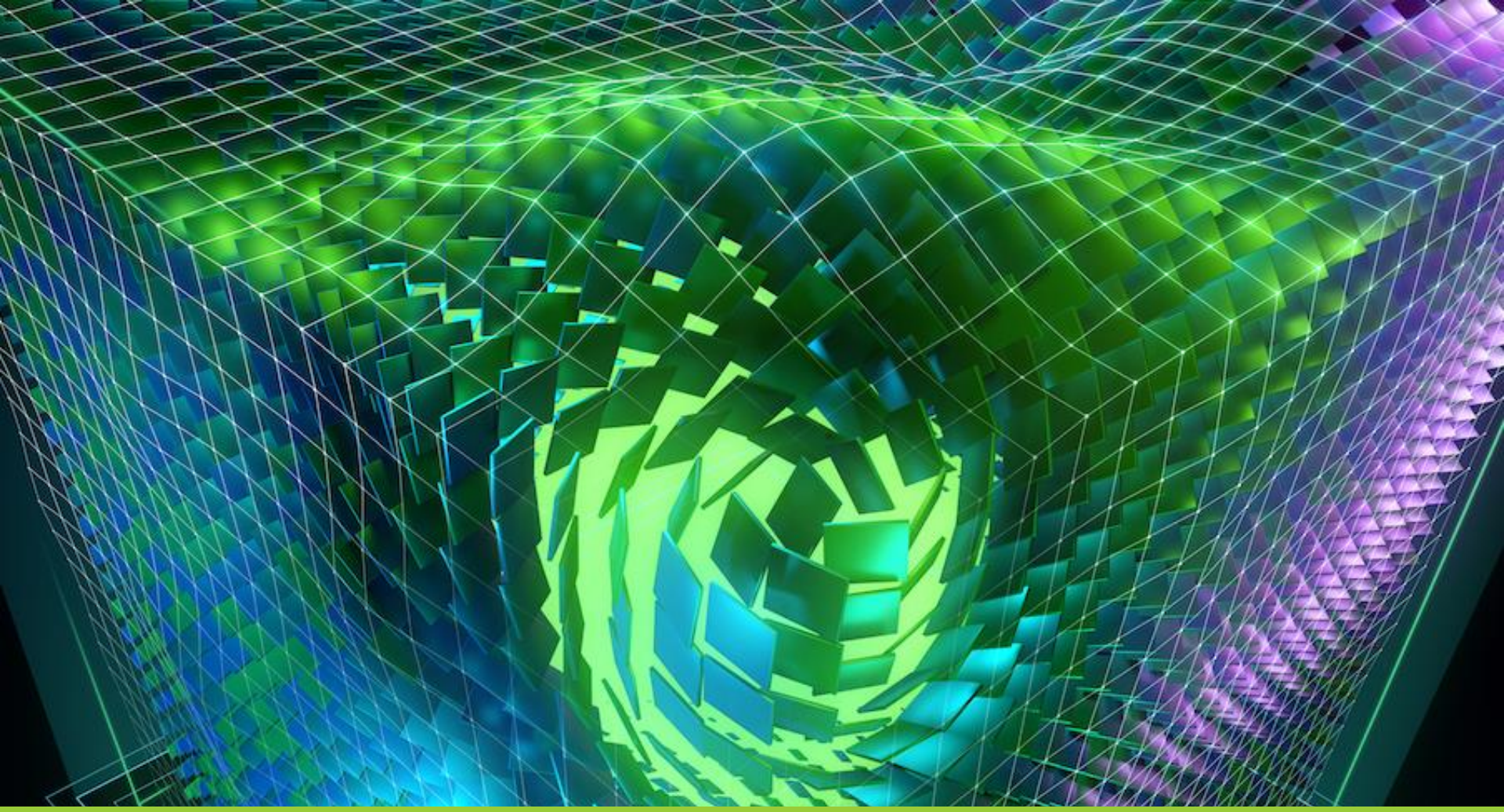


INF351 – Computación de Alto Desempeño

Memoria Global

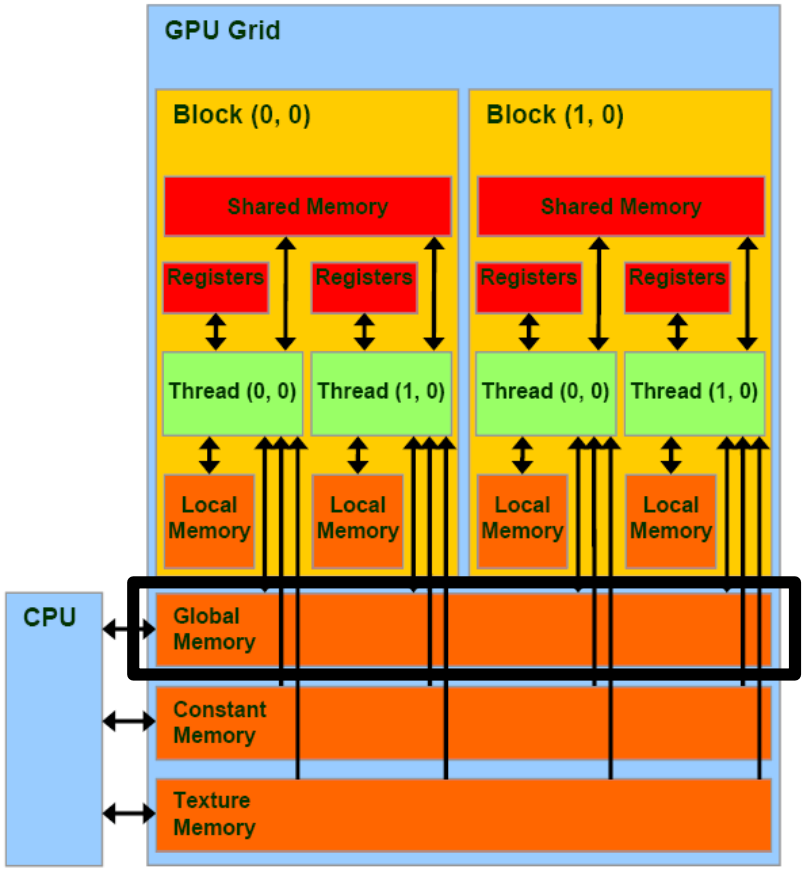
PROF. ÁLVARO SALINAS

Memoria Global

Uno de los tipos de memoria del dispositivo. Corresponde al más lento de ellos.

Se encuentra en el espacio de memoria VRAM en la tarjeta gráfica, pero fuera de la GPU.

Es la memoria más importante, pues es la única que es persistente entre invocaciones de kernels y puede ser utilizada para escritura.



Manejo de Memoria Global

La memoria global corresponde a la memoria utilizada “por defecto” en CUDA.

Todas las variables que creemos de forma estática con la etiqueta __*device*__ y de forma dinámica con *cudaMalloc()* se almacenan automáticamente en memoria global.

Tal como hemos revisado en clases anteriores, para realizar traspasos de memoria desde CPU a memoria global, desde memoria global a CPU o entre variables en memoria global, debemos utilizar *cudaMemcpy()*.

Transacciones de Memoria

Una transacción de memoria corresponde a un acceso secuencial a dicha memoria. En otras palabras, corresponde a las lecturas o escrituras realizadas en una memoria al mismo tiempo.

Por ejemplo, la instrucción *array1[i] = array2[i]* necesita dos transacciones de memoria, una para lectura y otra para escritura, pues estos pasos se realizan de forma secuencial.

Una transacción de memoria realizada en memoria global presenta una latencia que varia entre 200 y 1400 ciclos.

Si bien no es posible disminuir esta latencia, podemos ayudar a ocultarla reduciendo el número de transacciones realizadas por la GPU, lo que también permitirá aprovechar de mejor manera el ancho de banda disponible.

Coalesced Memory Accesses

Coalesced Memory Accesses o Memory Coalescing hace referencia a la capacidad que tiene un warp de acceder a 128 bytes en memoria global en una única transacción de memoria.

Gracias a esto, 32 hebras pueden leer o escribir 32 flotantes de precisión simple (128 B) en una transacción o 32 flotantes de precisión doble en dos transacciones.

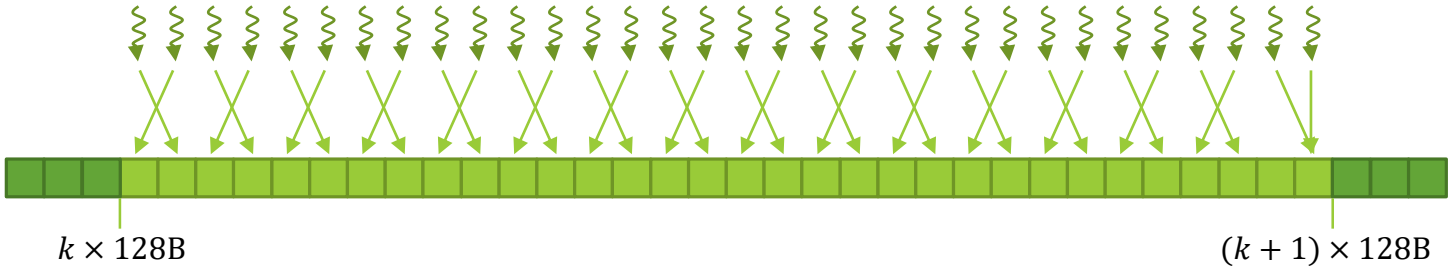
Para aprovechar esto, el acceso, ya sea lectura o escritura, debe cumplir ciertas propiedades: alineamiento, consecutividad y secuencialidad.

A continuación, veremos a qué se refiere cada propiedad.

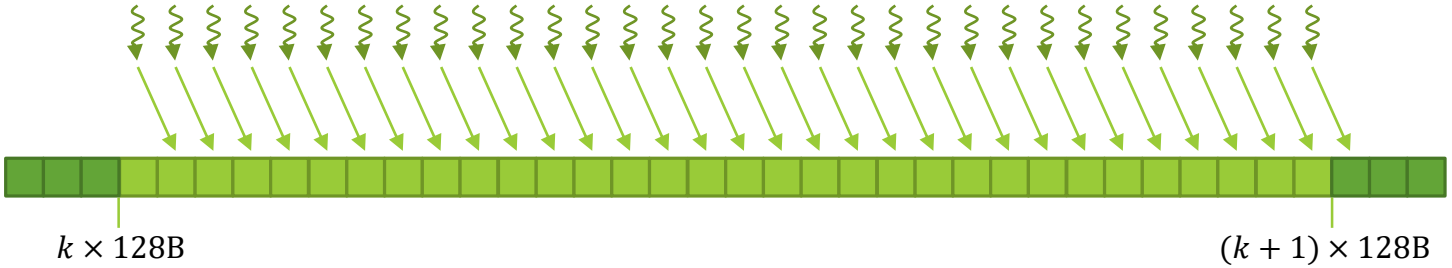
Accesos Alineados

La memoria se divide en “palabras” de 128 bytes desde la dirección de memoria inicial del array. Un acceso alineado corresponde a un acceso que se realiza en su totalidad a una de estas “palabras”.

Acceso alineado:



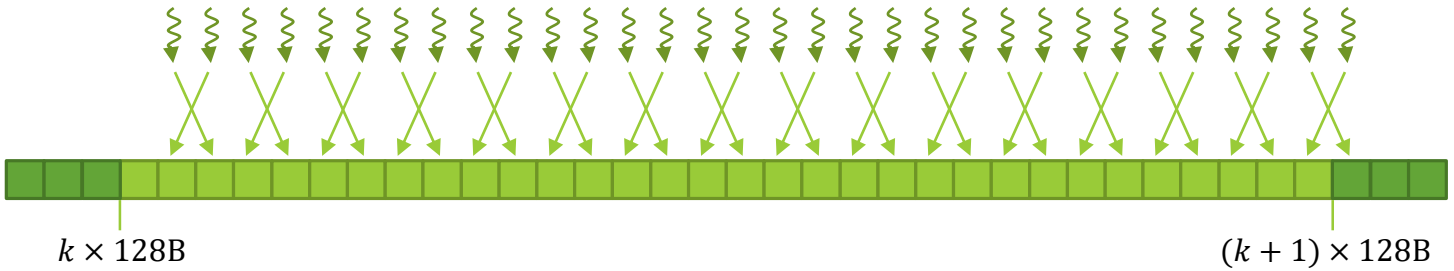
Acceso no alineado:



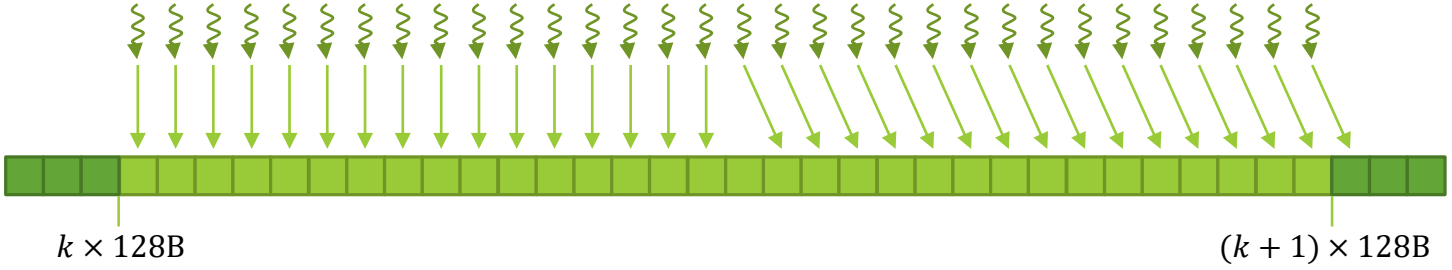
Accesos Consecutivos

Los accesos son consecutivos cuando se realizan a 128 bytes sucesivos en memoria.

Acceso consecutivo:



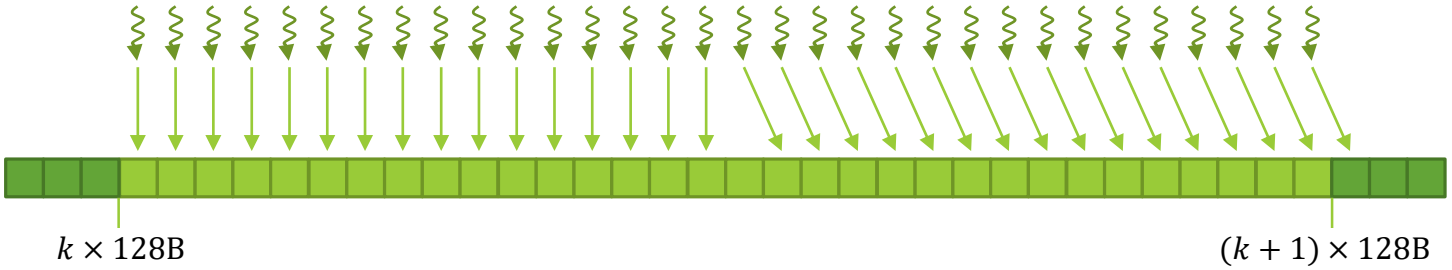
Acceso no consecutivo:



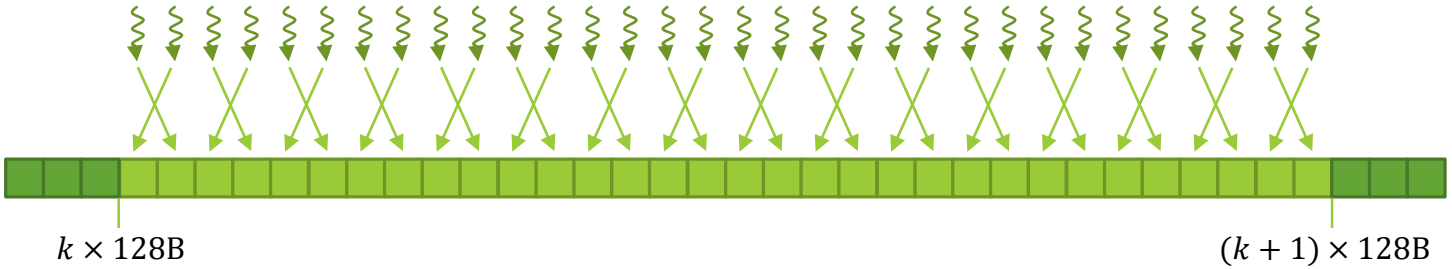
Accesos Secuenciales

Los accesos secuenciales son aquellos en donde el orden de las hebras que conforman el warp se mantiene en relación a la dirección de memoria accedida.

Acceso secuencial:



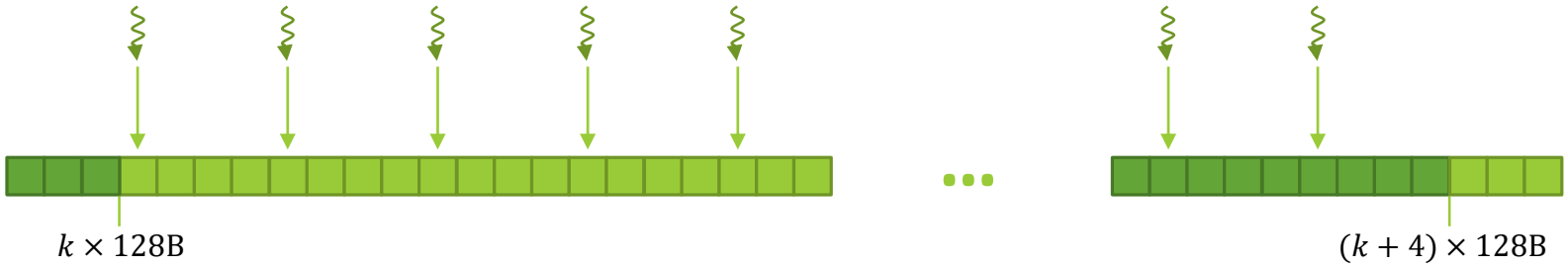
Acceso no secuencial:



Número de Transacciones

Si bien las tres propiedades revisadas eran necesarias para GPU con compute capabilities bajos, en la actualidad, la única relevante es el alineamiento de los accesos.

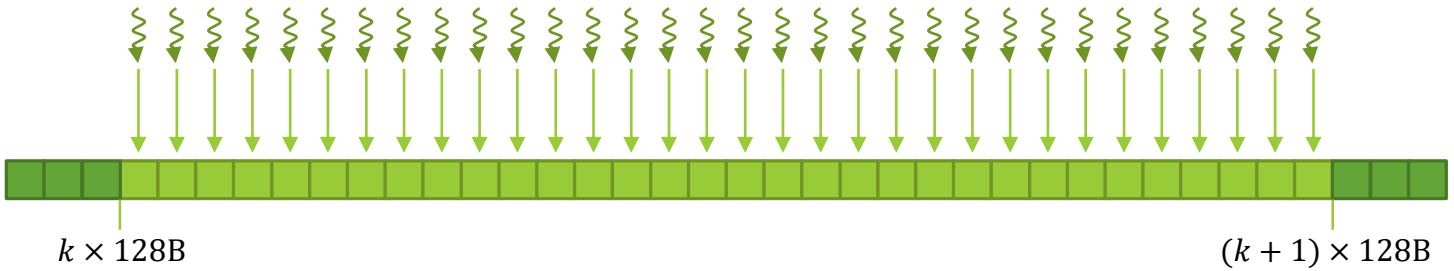
En palabras simples, el número de transacciones depende únicamente del número de palabras de 128 bytes que se acceden. Por ejemplo, si se leen o escriben 32 flotantes de precisión simple separados por 12 bytes (cada cuatro se lee uno), necesitaremos cuatro transacciones.



Caso Ideal

Se leen o escriben 32 números fp32 dentro de la misma palabra de 128 bytes (una transacción) o 32 números fp64 contenidos en dos palabras consecutivas de 128 bytes (dos transacciones).

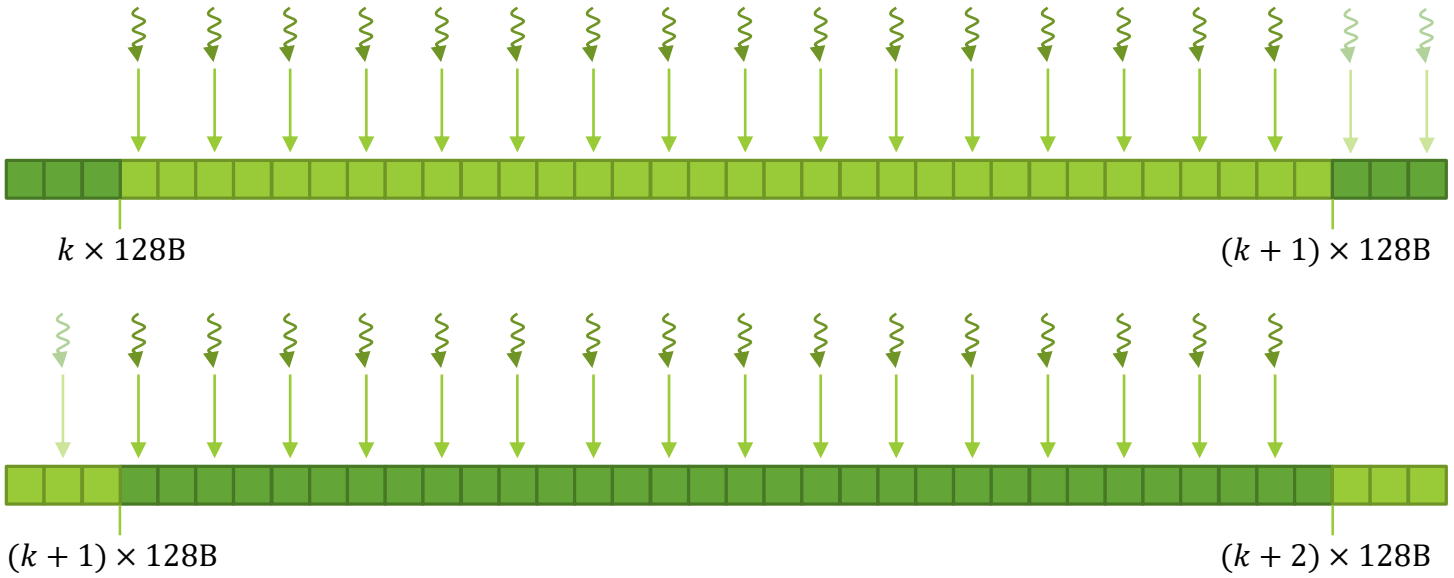
Precisión simple (fp32):



Caso Ideal

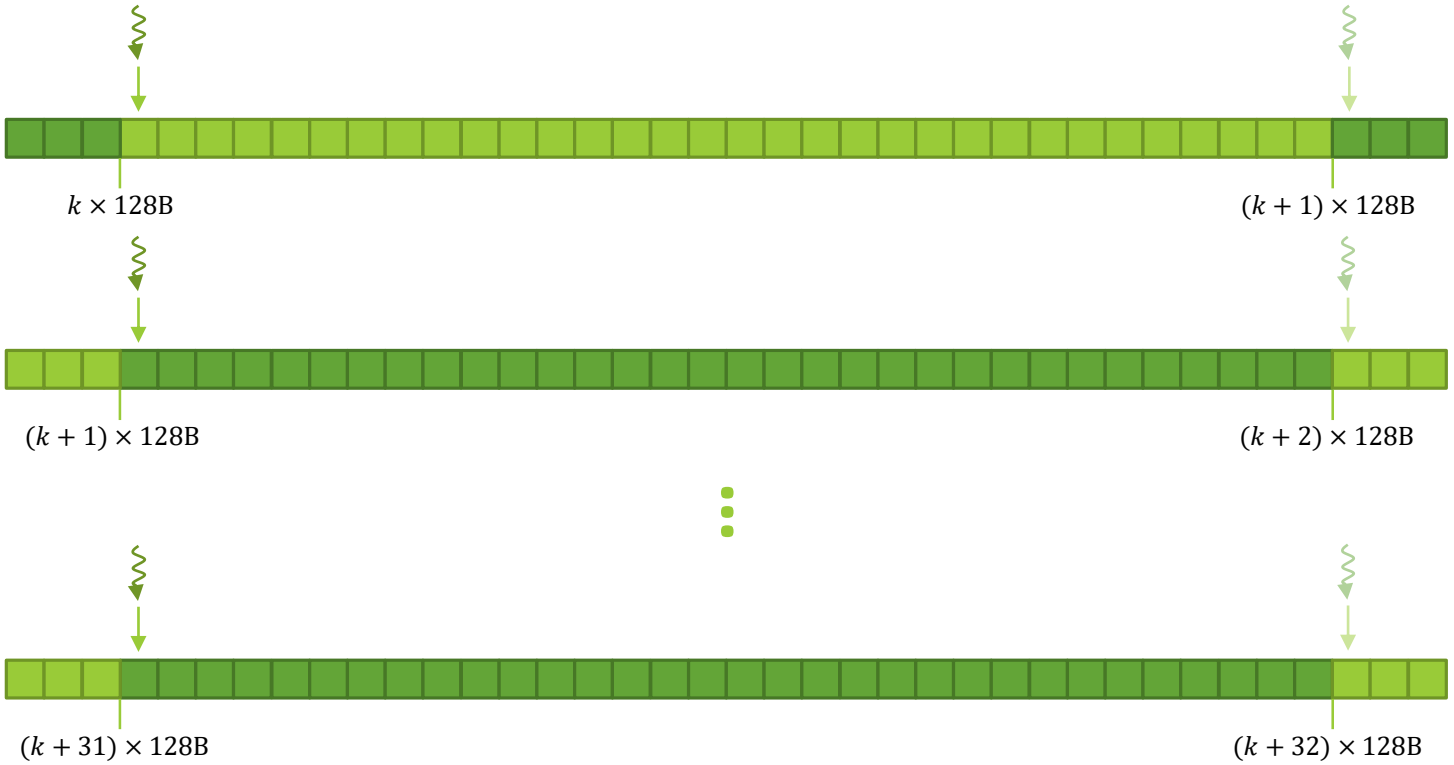
Se leen o escriben 32 números fp32 dentro de la misma palabra de 128 bytes (una transacción) o 32 números fp64 contenidos en dos palabras consecutivas de 128 bytes (dos transacciones).

Precisión doble (fp64):



Peor Caso

Se leen o escriben 32 números, cada uno contenido en distintas palabras de 128 bytes (32 transacciones).



Data Layout

Para obtener accesos de memoria coalescentes es necesario ordenar los datos de tal forma que las hebras puedan acceder a ellos de forma alineada.

Si bien existen diversas formas de organizar los datos, destacan principalmente dos, las cuales se denominan Array of Structures (AoS) y Structure of Arrays (SoA).

Al trabajar con CUDA, SoA generalmente es un mejor enfoque. A continuación veremos cómo se implementa cada uno.

Array of Structures

Usualmente es la forma más intuitiva de organizar los datos.

La lógica que sigue corresponde a juntar todos los **datos** (componentes, propiedades, datos de entrada, entre otros) asociados al **elemento** procesado por cada thread, lo que sería como una **structure**, y generar un **array** de estas **structures** para almacenar los datos que todas las hebras necesitan.

Lamentablemente, en la mayoría de los casos no es posible obtener accesos de memoria coalescentes con este enfoque.

Structure of Arrays

Si bien no es tan intuitivo como AoS, SoA permite obtener coalescencia en los accesos de memoria.

Con este enfoque, la lógica consiste en agrupar en **arrays** distintos cada **dato** de todos los **elementos** procesados por cada thread. Posteriormente se agrupan estos **arrays** en lo que sería una **structure** que almacena todos los datos que todas las hebras necesitan.

Es el equivalente a tener un array distinto para cada dato, lo que no siempre es cómodo o conveniente de realizar.

Ejemplo 1

Se necesita calcular la energía mecánica de N cuerpos. Cada cuerpo posee masa (m_i), velocidad (v_i) y altura (h_i) determinadas (no hay energía potencial elástica).

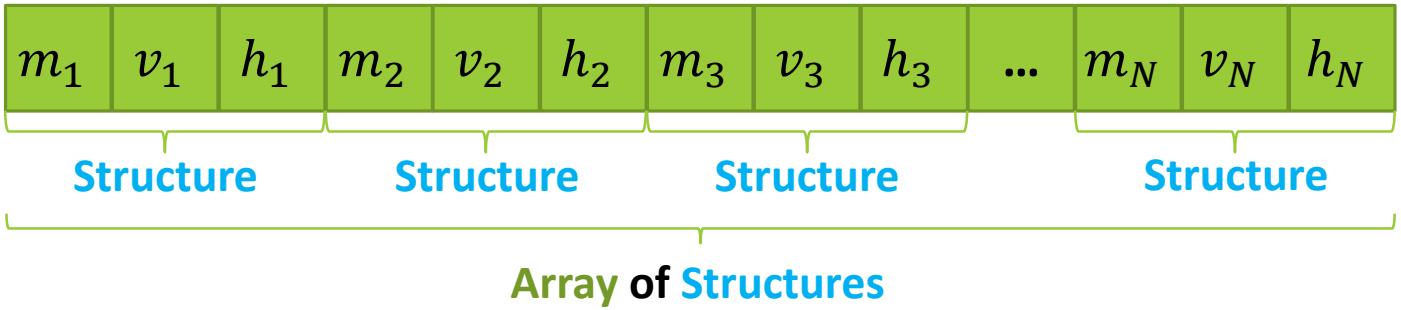
En nuestro algoritmo, cada thread se encargará de realizar los cálculos necesarios para un cuerpo (**elemento**), por lo tanto necesitará leer las tres propiedades (**datos**) de ese cuerpo.

Con AoS, las tres propiedades del mismo cuerpo serían almacenadas en direcciones de memoria consecutivas y esto se repetiría según la cantidad de cuerpos.

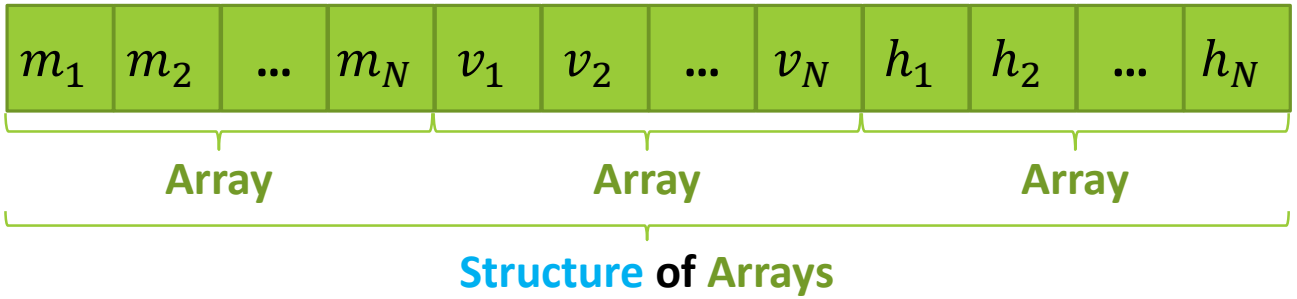
Con SoA, se tendrían las masas de todos los cuerpos seguidas de sus velocidades y finalmente las alturas.

Ejemplo 1

AoS:

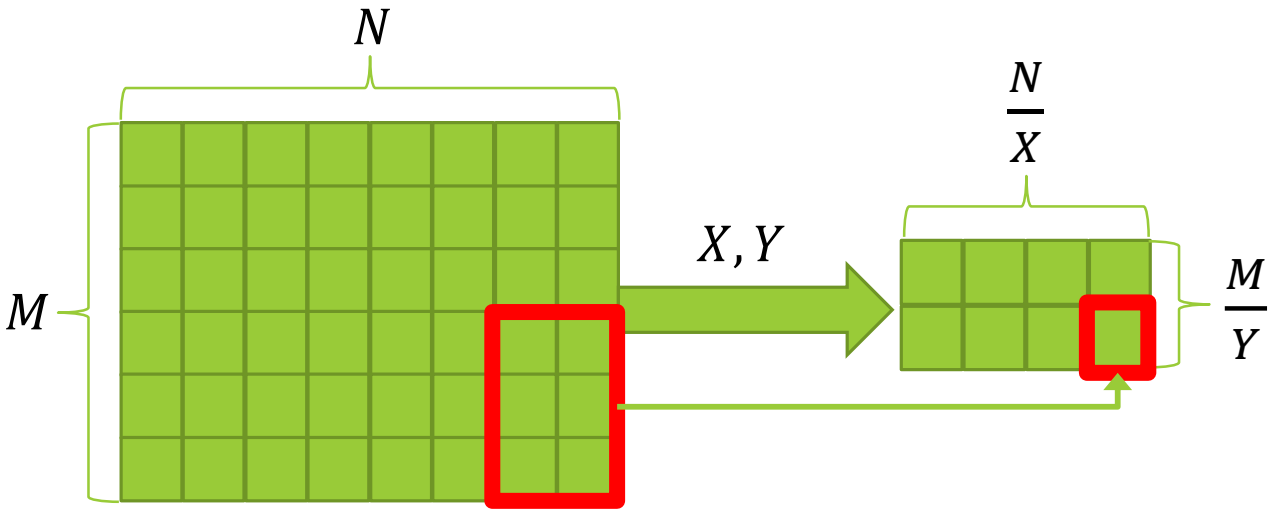


SoA:



Ejemplo 2

Recordemos el ejercicio de escalamiento suavizado de la primera clase práctica: Dada una imagen de tamaño $M \times N$, se deben promediar los pixeles contenidos en subimágenes de tamaño $Y \times X$ para así generar una nueva imagen de tamaño $M/Y \times N/X$.

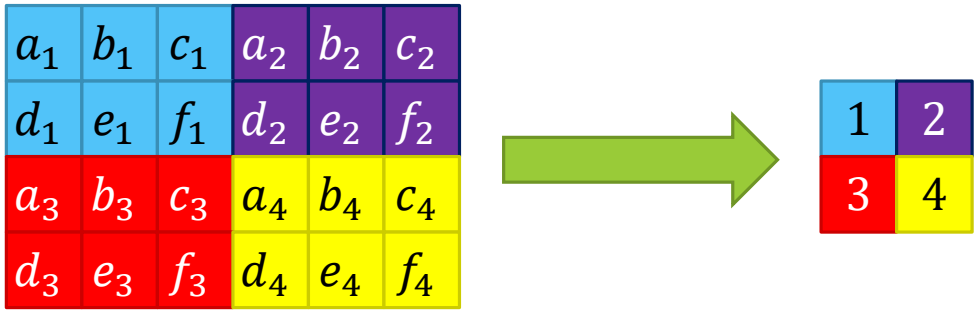


Ejemplo 2

La solución más común a un problema como éste es asignar cada thread a un pixel de la imagen resultante (**elemento**), para así evitar posibles condiciones de carrera. De esta forma, los $Y \times X$ pixeles que cada thread debe leer desde la imagen original corresponden a los **datos** necesarios.

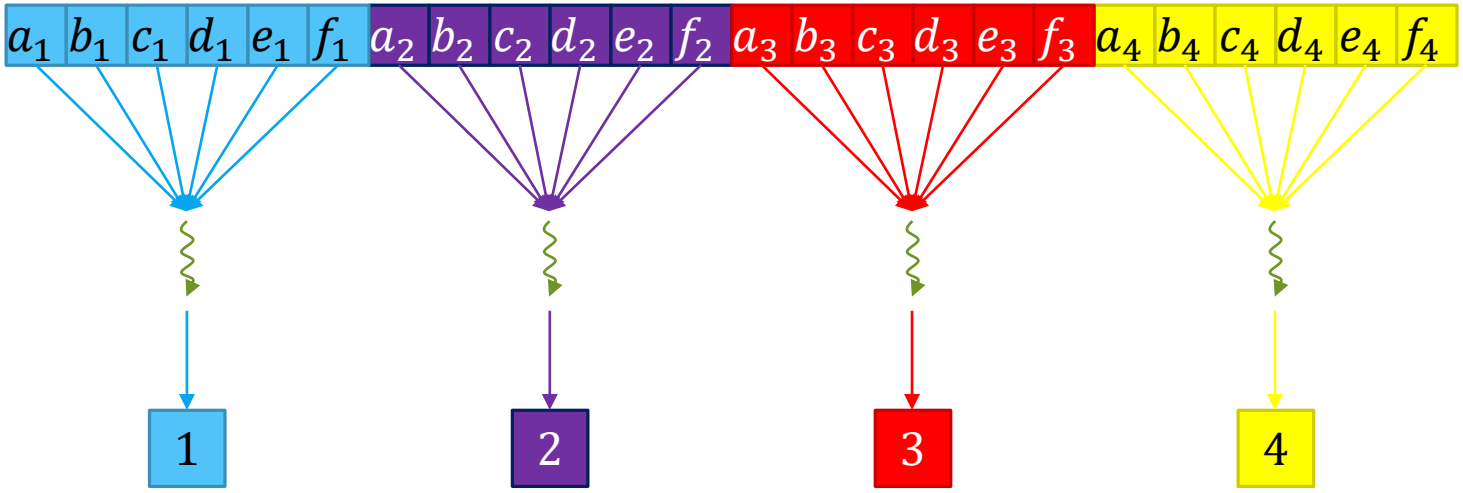
Analicemos el siguiente caso:

$$M = 4$$
$$N = 6$$
$$Y = 2$$
$$X = 3$$



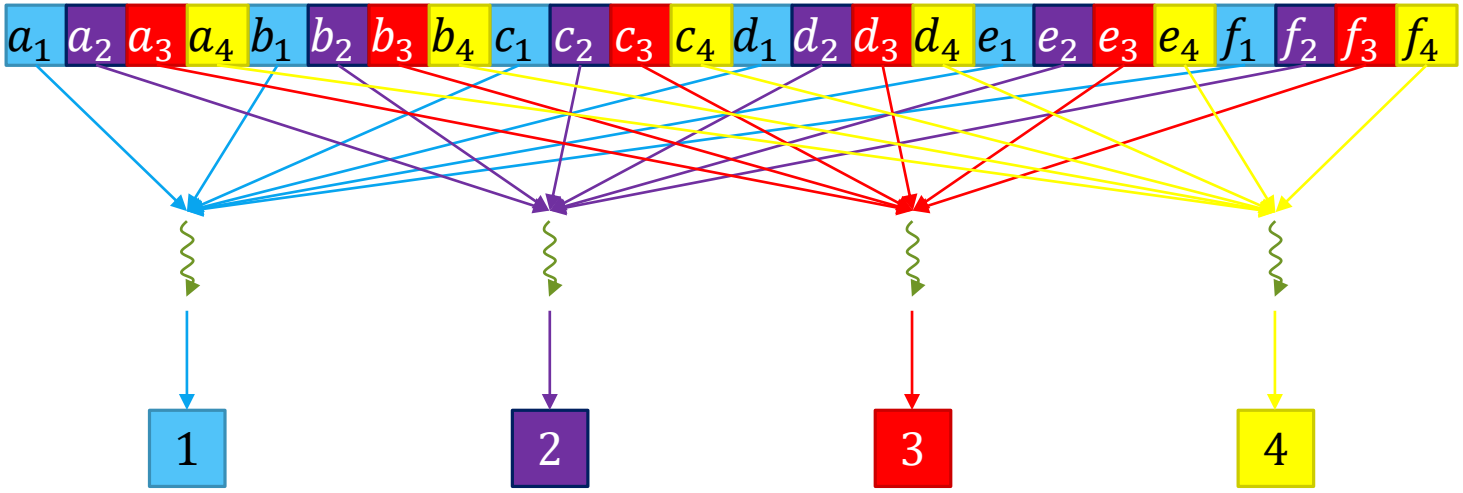
Ejemplo 2

Con AoS el arreglo de datos correspondiente a la imagen original estaría organizado de esta forma:



Ejemplo 2

Por otro lado, con SoA tendríamos:



Ejemplo 2

Finalmente, ¿qué pasaría si simplemente concatenáramos las filas de la imagen original?