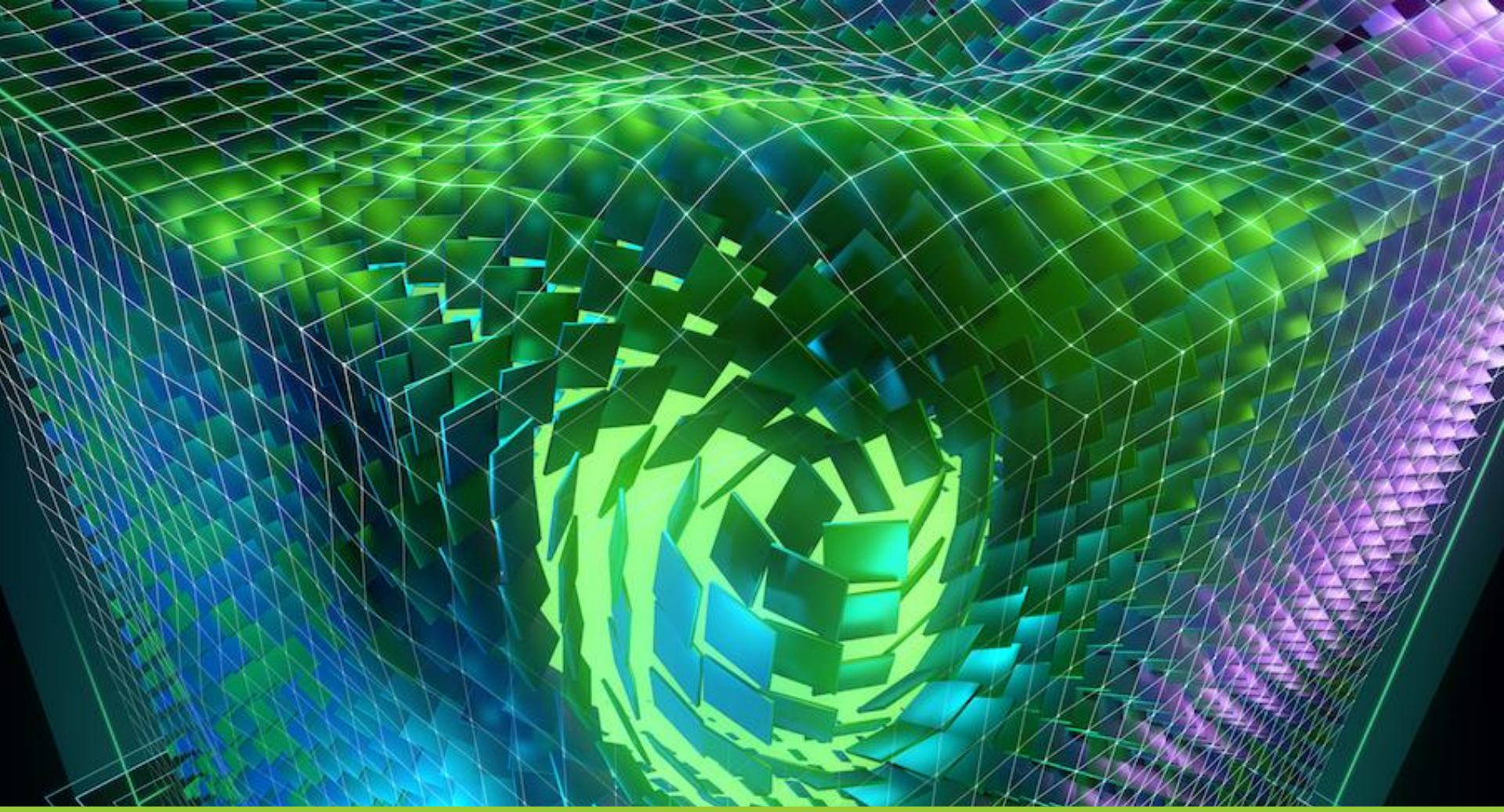


INF351 – Computación de Alto Desempeño

Operaciones Atómicas

PROF. ÁLVARO SALINAS

Condiciones de carrera

Al trabajar con CUDA (y con cualquier arquitectura paralela en general) son comunes los casos en donde pueden surgir condiciones de carrera.

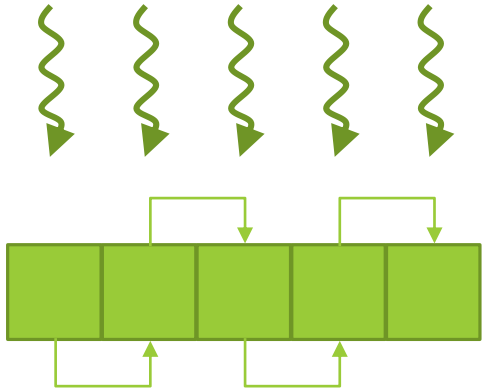
Como sabemos, una condición de carrera ocurre cuando una secuencia de instrucciones que se ejecutarán en orden arbitrario y sobre un mismo recurso compartido son realizadas en un orden distinto al esperado.

Es fácil imaginar qué casos pueden producir una condición de carrera al trabajar en la GPU. A continuación revisaremos los tres más comunes.

Condiciones de carrera

Caso 1:

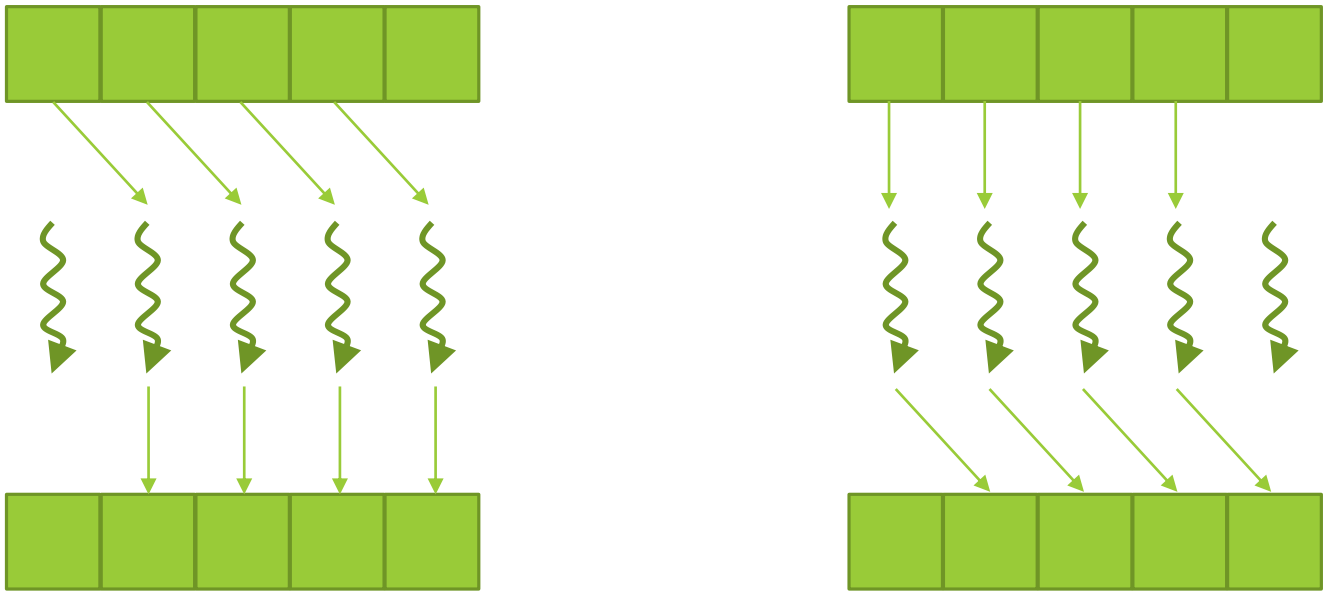
Una hebra debe leer/escribir datos a cargo de otras hebras y también escribir/leer el suyo.



Condiciones de carrera

Caso 1:

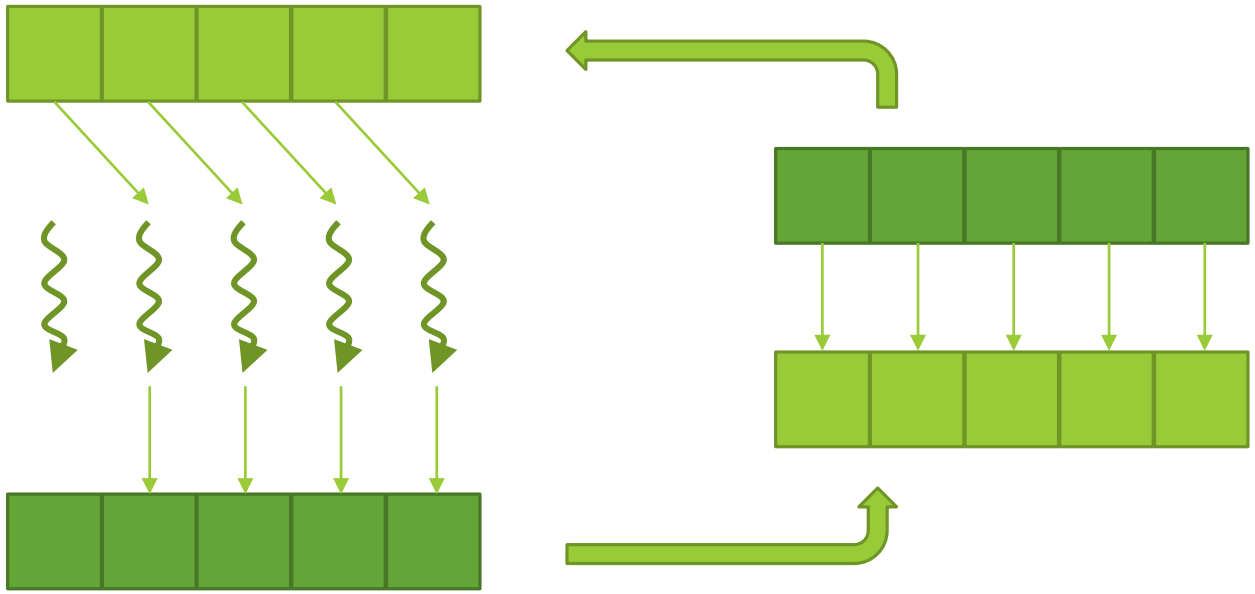
Una hebra debe leer/escribir datos a cargo de otras hebras y también escribir/leer el suyo.



Condiciones de carrera

Caso 1:

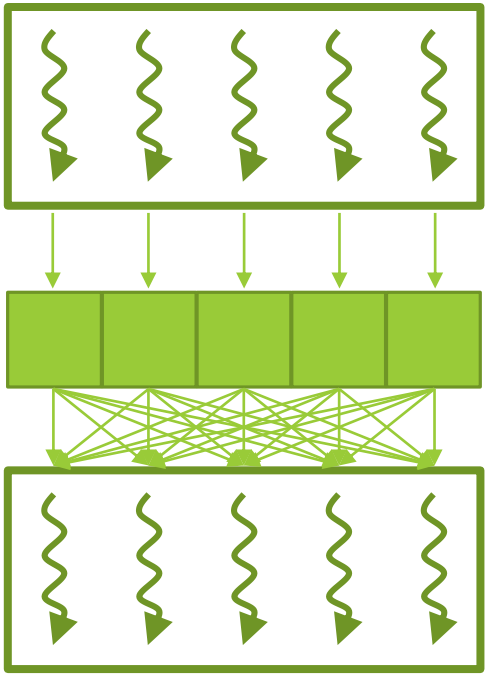
Solución: Doble búfer.



Condiciones de carrera

Caso 2:

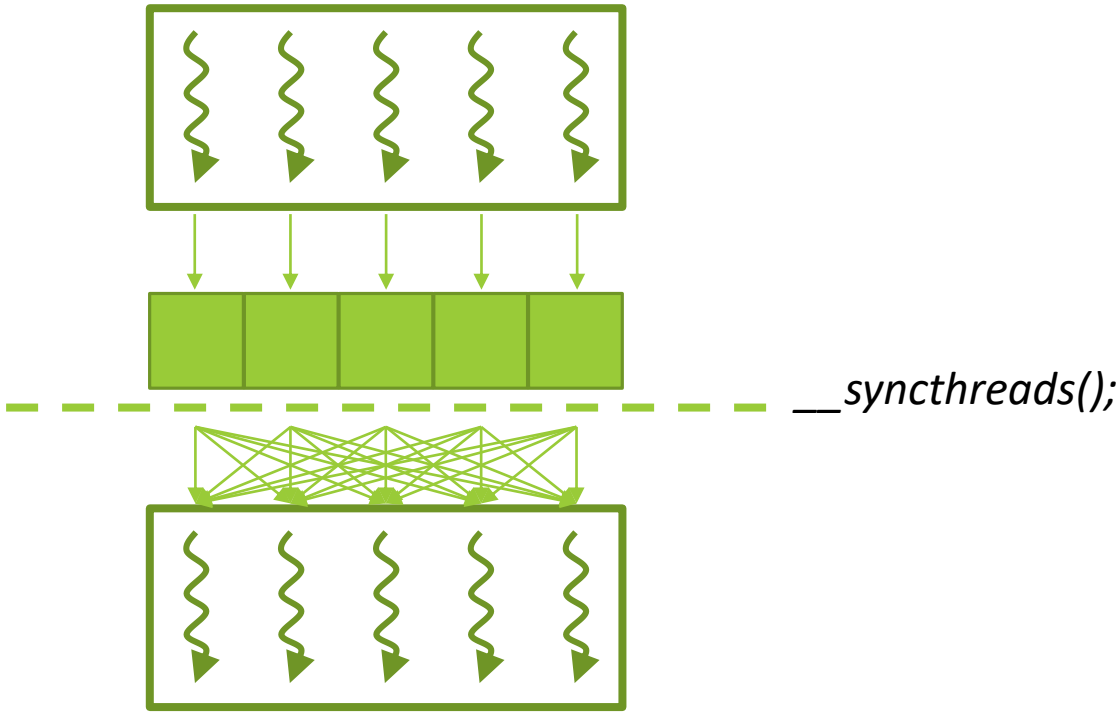
Las hebras deben escribir datos que serán leídos por las que pertenecen al mismo bloque, como ocurre al trabajar con memoria compartida.



Condiciones de carrera

Caso 2:

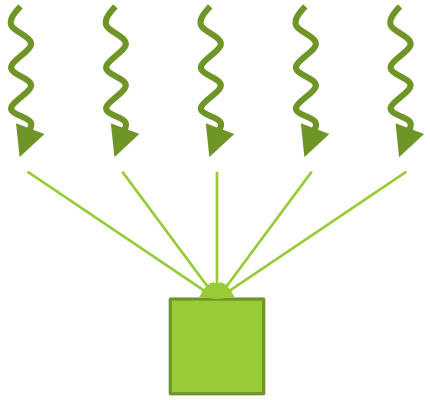
Solución: *__syncthreads();*



Condiciones de carrera

Caso 3:

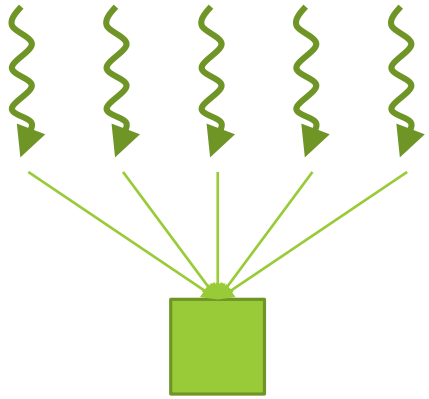
Hebras diferentes deben sobreescribir el mismo dato. No siempre esto implica la posibilidad de una reducción.



Condiciones de carrera

Caso 3:

Solución: ???



Instrucciones Atómicas

Una instrucción atómica, en cualquier ámbito de la informática, corresponde a un conjunto de instrucciones que son consideradas como una única instrucción indivisible por el resto del sistema.

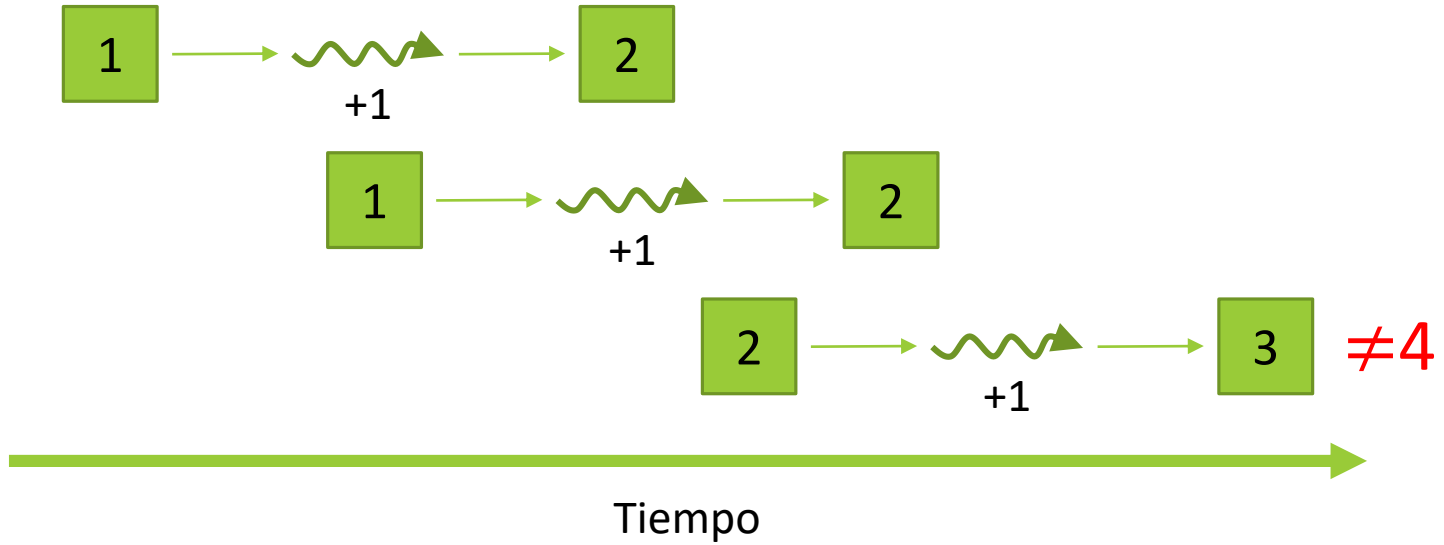
En otras palabras, ningún otro proceso puede conocer los cambios intermedios hasta que se complete la totalidad del conjunto de instrucciones.

En CUDA, este tipo de instrucciones serán de gran ayuda para evitar condiciones de carrera cuando varias hebras intenten realizar una operación a un mismo dato.

Operaciones Atómicas

En CUDA tenemos la posibilidad de ejecutar operaciones del tipo lectura-modificación-escritura de forma atómica mediante funciones del tipo __*device*__ que pueden ser llamadas desde un kernel. A este tipo de operaciones se les llama operaciones atómicas.

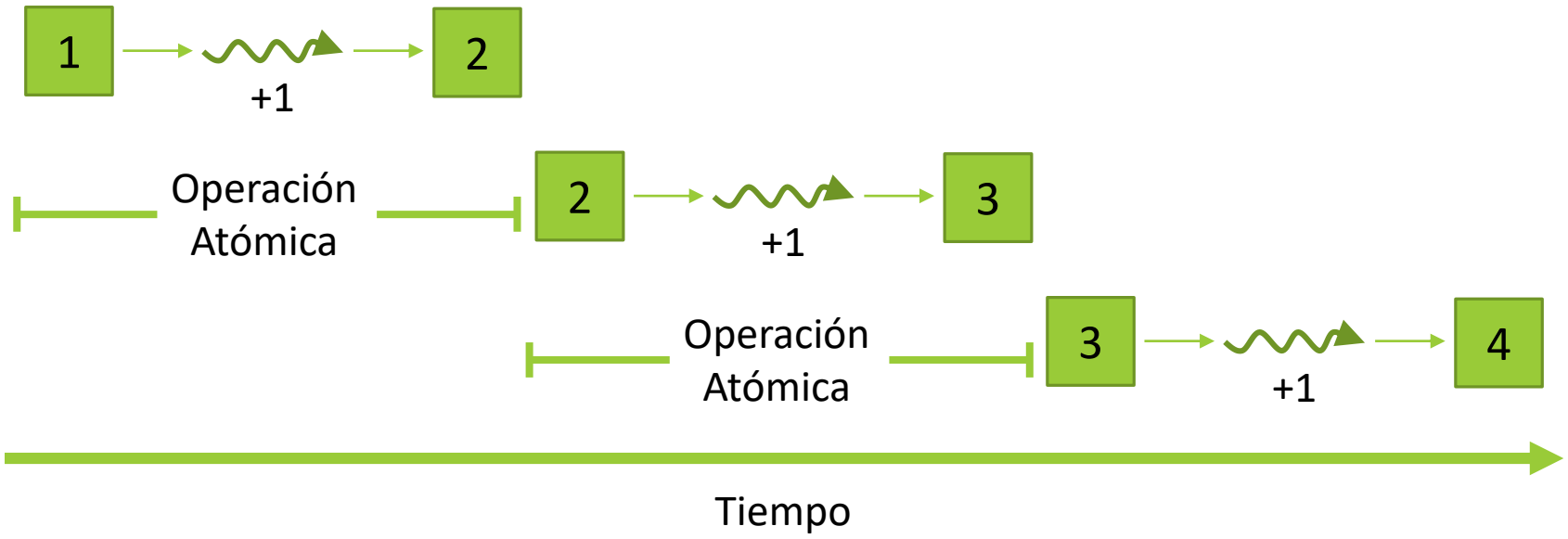
Volvamos a revisar el caso 3 de condiciones de carrera:



Operaciones Atómicas

Una operación atómica “bloquea” el dato a ser modificado para que otras hebras no puedan trabajar con él hasta que la escritura haya sido realizada. Esto significa una serialización del trabajo de las distintas hebras.

Con operaciones atómicas, el ejemplo anterior se transformaría en:



Operaciones Atómicas

A continuación veremos algunas operaciones atómicas predefinidas:

- Para realizar una suma de manera atómica, tenemos a nuestra disposición la función *atomicAdd(*address, val)*, la cual lee el valor antiguo (*old*) en la dirección *address*, le suma *val* y sobreescribe el dato en la dirección de memoria por el resultado de la adición. Retorna *old*.

- Del mismo modo, *atomicSub(*address, val)* realiza la resta old-val de manera atómica. También retorna *old*. A diferencia de *atomicAdd()*, esta función no soporta flotantes.

Operaciones Atómicas

- La función *atomicMin(*address, val)* lee el valor antiguo (*old*) en la dirección *address*, y lo sobreescribe por el mínimo entre *val* y *old*. Retorna *old*. No soporta flotantes.

- Análogamente, *atomicMax(*address, val)* sobreescribe el valor en *address* por el máximo entre éste y *val*. Retorna *old*. Tampoco soporta flotantes.

Existe una gran variedad de operaciones atómicas predefinidas, pero algunas de ellas son de usos muy específicos. Puede revisar la documentación de éstas en: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

atomicCAS

La operación atómica *atomicCAS(*address, compare, val)* es una función de bastante utilidad, pues nos permite implementar otras funciones atómicas. De hecho, las funciones atómicas predefinidas están basadas en ésta.

Su nombre proviene de Compare And Swap y, tal como esto sugiere, lo que esta función realiza es comparar el valor *old* almacenado en *address* con *compare*, reemplazandolo por *val* en caso de ser verdadera dicha condición. En otras palabras, ejecuta lo siguiente:

old == compare ? val : old

sobreescribiendo el resultado en la dirección de memoria de forma atómica. Retorna old.

atomicCAS

Ya que no existe una operación atómica predefinida para la multiplicación, podemos implementarla con la ayuda de *atomicCAS*.

Al realizar esto, deberíamos generar el siguiente código:

```
__device__ int atomicMul(int* address, int val){
    int old = *address, assumed;
    do{
        assumed = old;
        old = atomicCAS(address, assumed, val*assumed);
    } while(assumed != old);
    return old;
}
```

Int a Float

También es posible generar operaciones atómicas que soporten flotantes.

Dado que *atomicCAS* solo soporta enteros, para lograr esto debemos acudir a los casteos intrínsecos de CUDA.

Si bien éstos los veremos formalmente en clases posteriores, por ahora adelantaremos dos de ellos, los cuales son *__float_as_int()* y *__int_as_float()* los cuales reinterpretan los bits de un flotante a un entero y de un entero a un flotante respectivamente.

Int a Float

Siguiendo el ejemplo anterior, para una multiplicación atómica de flotantes tendríamos:

```
__device__ float atomicMul(float* address, float val){
    int *address_int = (int*)address;
    int old = *address_int, assumed;
    do{
        assumed = old;
        old = atomicCAS(address, assumed,
                        __float_as_int(val*__int_as_float(assumed)));
    } while(assumed != old);
    return __int_as_float(old);
}
```

¿Cuándo usarlas?

Ahora que sabemos qué son y cómo utilizar las operaciones atómicas, debemos entender cuándo es conveniente o necesario utilizarlas.

Como vimos, las operaciones atómicas son de gran utilidad para evitar condiciones de carrera, pero lamentablemente provocan una serialización del trabajo de las hebras, por lo que su uso debiese ser evitado.

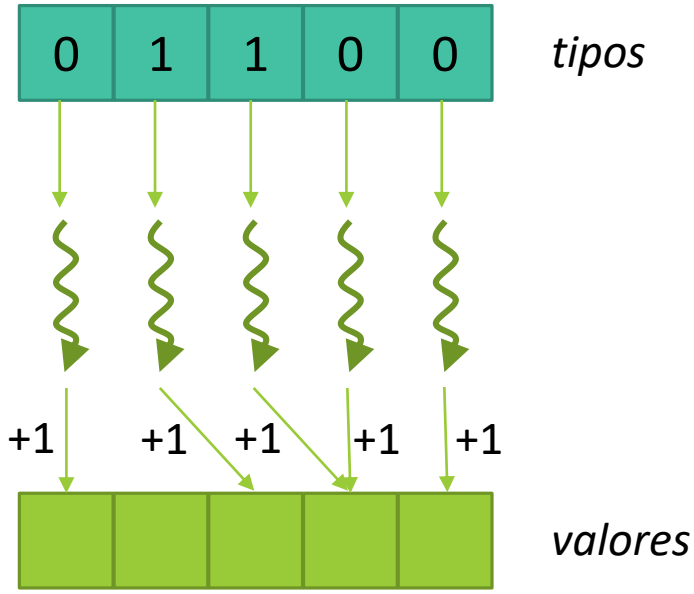
Aun así, existen casos en donde son indispensables, pues de no utilizarlas, caeríamos en serios problemas de desempeño.

Ejemplo 1

Imaginemos el siguiente caso:

Tenemos dos arrays llamados *tipos* y *valores*. El primero de ellos es un array de valores binarios que determinan el comportamiento de las hebras.

Para este caso, supongamos que si *tipo[id]* es 0, la hebra con identificador *id* debe sumarle 1 al valor en la posición *id*. De ser *tipo[id]* igual a 1, dicha hebra debe sumarle 1 al valor en la posición *id+1*.

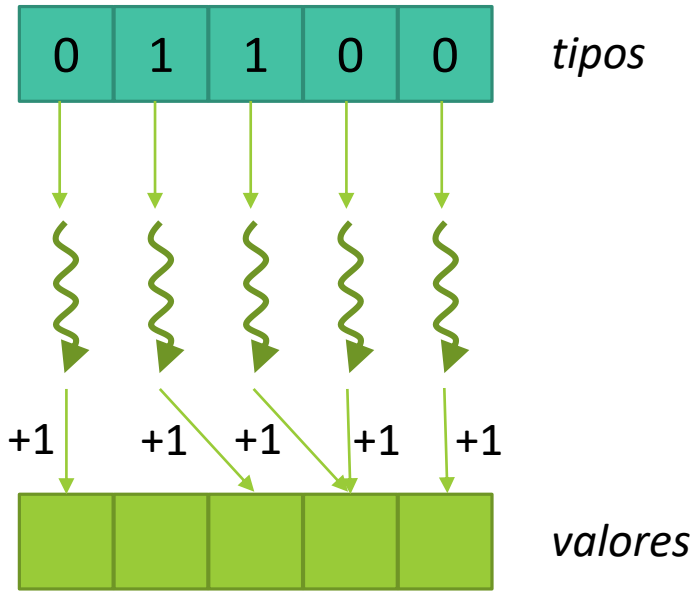


Ejemplo 1

Analizando este caso, podemos darnos cuenta de que no estamos frente al caso 1 de vistos al inicio de la clase. Esto se debe a que los datos modificados no van a volver a ser leídos, por lo que esta condición de carrera no se resuelve mediante un doble búfer.

Del mismo modo, tampoco estamos frente al caso 2, pues las hebras no pertenecen al mismo bloque y, de nuevo, no se leerán los datos modificados.

Por lo tanto, es un caso 3 en donde distintas hebras intentaran escribir en la misma dirección de memoria. Aunque claramente está lejos de poder verse como una reducción.



Ejemplo 1

Una posible solución al problema sería separar el trabajo en dos kernels distintos, uno que se encargue de las hebras con tipo 0 y otro para las hebras de tipo 1. De esta forma, nos aseguraríamos que no existen condiciones de carrera.

```
__global__ void kernel0(int* tipos, int* valores, int N){
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N && tipos[tId] == 0){
        valores[tId] += 1;
    }
}

__global__ void kernel1(int* tipos, int* valores, int N){
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N && tipos[tId] == 1){
        valores[(tId + 1) % N] += 1;
    }
}
```

Ejemplo 1

Pero, ¿estamos realmente resolviendo el problema?

Lo que realmente estamos haciendo es dividirlo en dos problemas distintos, provocando una gran ineficiencia.

En caso de que todos los tipos fueran iguales, ya sean 0 o 1, no existirían condiciones de carrera, pero de todas formas tendríamos que inicializar un segundo kernel y volver a acceder a todos esos datos en memoria global.

Al utilizar operaciones atómicas, podríamos mejorar mucho nuestra solución.

Ejemplo 1

Siguiendo el enfoque de las operaciones atómicas tendríamos:

```
__global__ void kernel(int* tipos, int* valores, int N){
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N){
        int tipo = tipos[tId];
        if (tipo == 0) atomicAdd(&valores[tId], 1);
        else atomicAdd(&valores[(tId + 1) % N], 1);
    }
}
```

De esta forma, la operación atómica se encarga de lidiar con las posibles condiciones de carrera. En el mejor de los casos (todos los tipos iguales), no habría serialización, mientras que en el peor (tipos en posiciones pares distintos a los de las posiciones impares) solo se necesitarían dos sumas (4 accesos a memoria global) secuenciales.