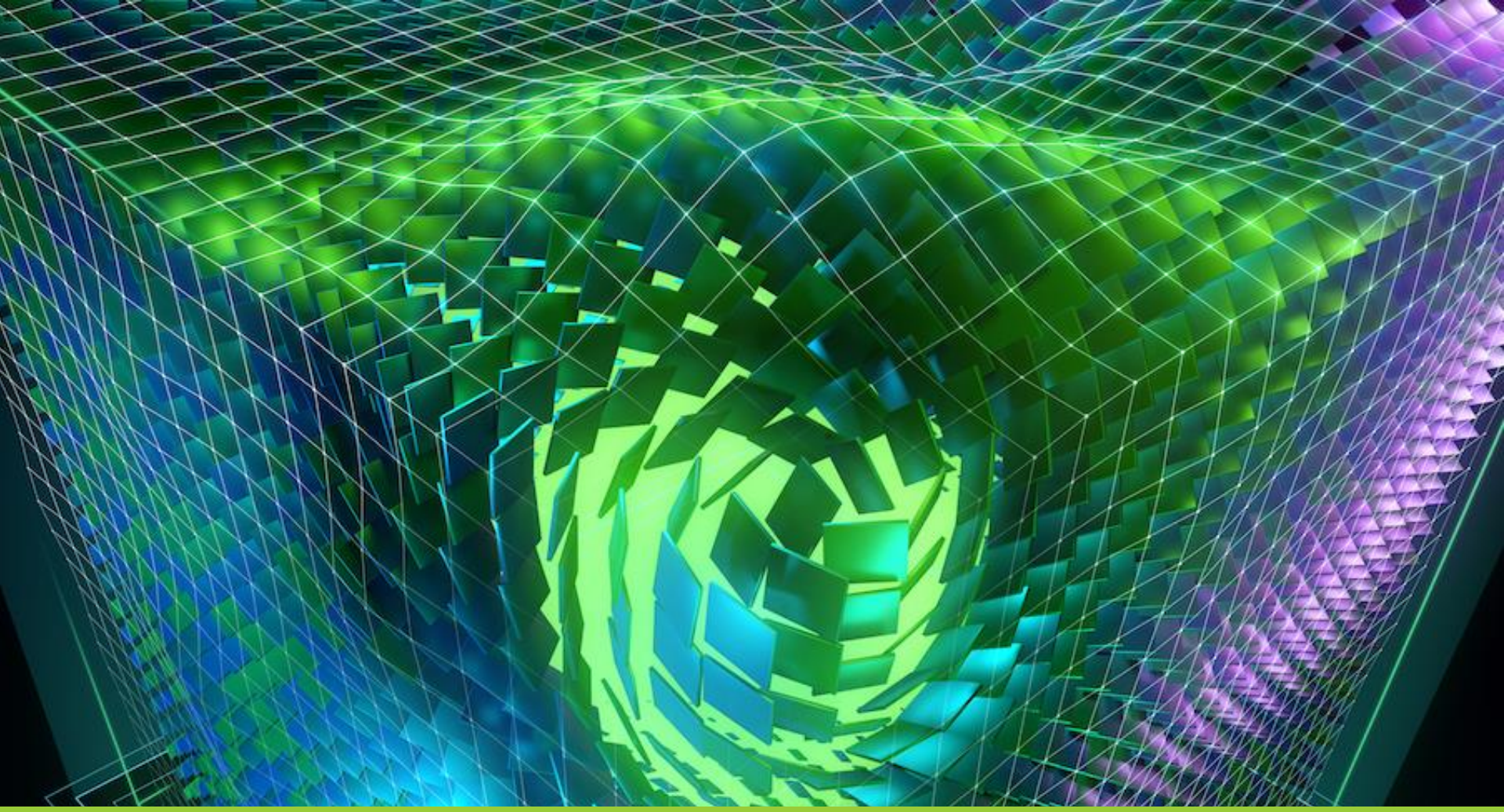


INF351 – Computación de Alto Desempeño

Divergencia, Latencia y Ocupancia

PROF. ÁLVARO SALINAS

Optimizaciones

Ahora que sabemos cómo desarrollar un código paralelo funcional en CUDA, es momento de preocuparnos de que tenga un buen desempeño.

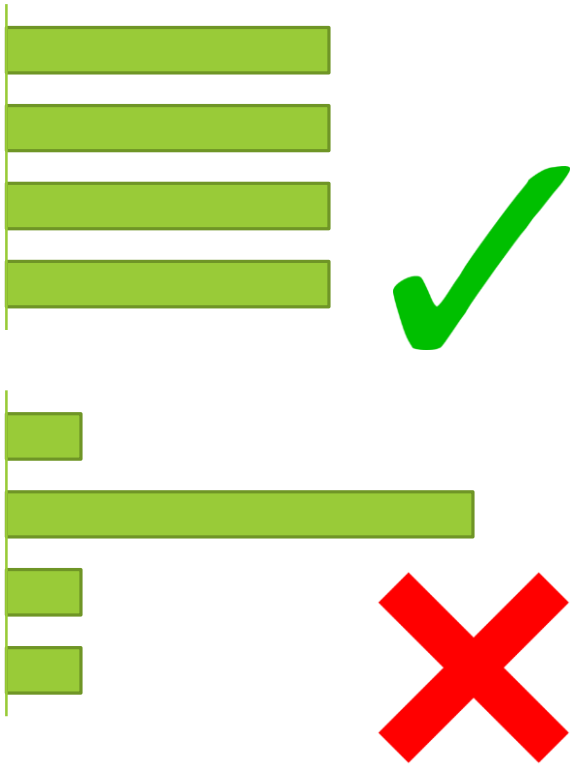
Por lo general, la etapa de optimizar un código en CUDA es la más larga dentro del desarrollo de una aplicación.

Las optimizaciones que se pueden realizar cubren varios aspectos, como el uso de recursos, parámetros de configuración, accesos de memoria, entre otros.

Si bien hay algunas buenas prácticas que suelen seguirse, algunas veces ciertas optimizaciones no supondrán un aumento en el desempeño de nuestra aplicación. Es importante identificar el “cuello de botella” y enfocar las optimizaciones en resolverlo.

Recordemos algunas cosas…

Uno de los principales problemas al trabajar en CUDA es la regularización.



Recordemos algunas cosas…

Otro de los problemas a resolver corresponde al ancho de banda (bandwidth).

Recordemos algunas cosas…

Finalmente, debemos también preocuparnos de la serialización provocada por instrucciones secuenciales.

Divergencia

La divergencia, mejor conocida como warp divergence, corresponde a la ramificación de las instrucciones realizadas por las hebras pertenecientes a un mismo warp.

El caso más común de divergencia corresponde a la presencia de estructuras condicionales como “if-else” o “switch”, aunque también puede darse en otros casos.

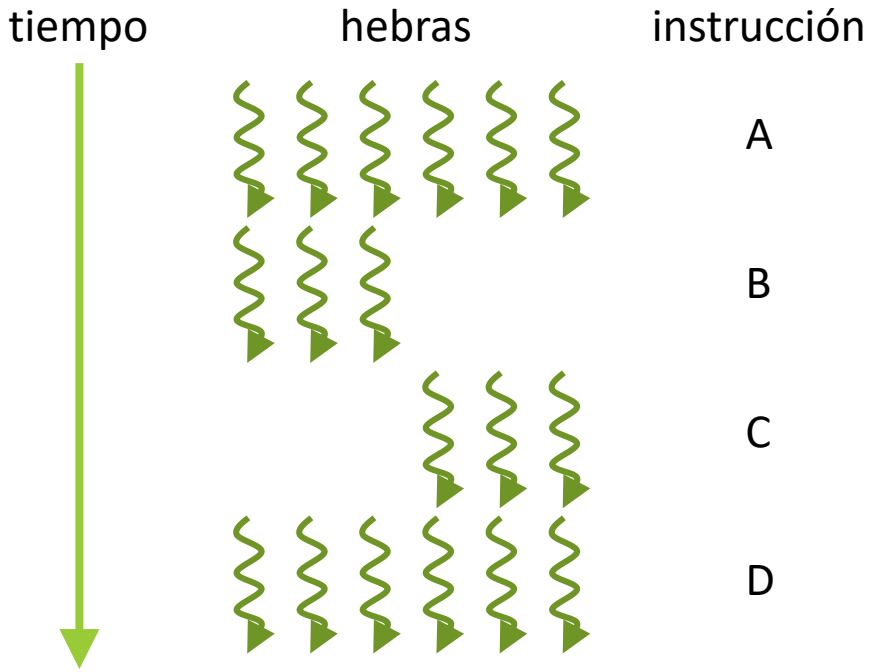
La divergencia puede producir problemas de regularización y serialización.

Ejemplos

Imaginemos 6 hebras ejecutando un kernel que realiza cierta instrucción cuando el identificador de la hebra sea menor que 3 y una distinta en el caso contrario:

```
tId = identificador de la hebra
Instrucción A;
if (tId < 3)
        Instrucción B;
else
        Instrucción C;
Instrucción D;
```

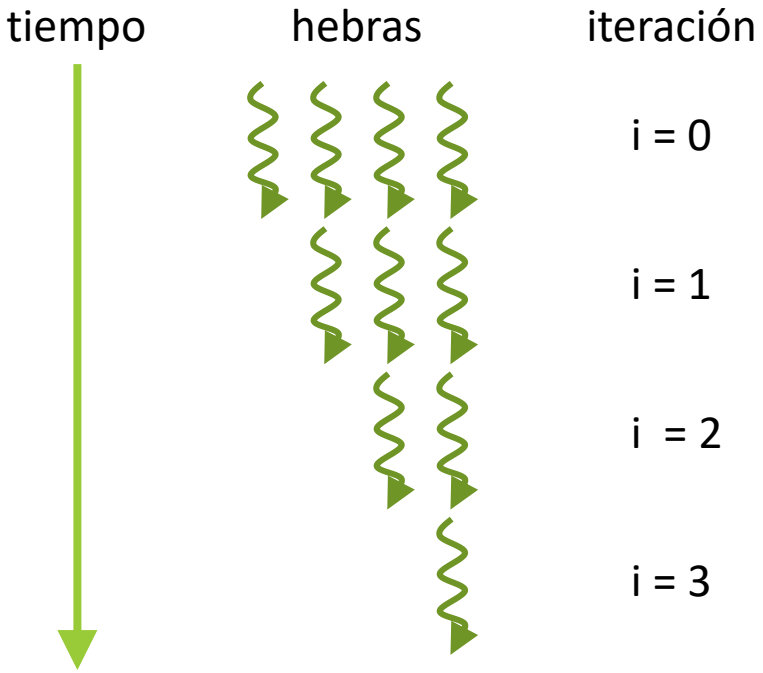
¡Problemas de seralización!



Ejemplos

Ahora pensemos en 4 hebras que ejecutan un ciclo cuyo número de iteraciones depende del identificador de cada hebra:

```
tId = identificador de la hebra
for (int i = 0; i <= tId; i++)
        Instrucción A;
```



¡Problemas de regularización!

Posibles Optimizaciones

En muchas ocasiones nos encontraremos con que la divergencia es algo necesario en nuestra aplicación, pues no siempre querremos procesar los datos de la misma manera.

Si la divergencia es efectivamente el cuello de botella de nuestra aplicación, posiblemente debamos rediseñar la solución al problema o pensar de forma distinta el modo de paralelizar nuestra aplicación.

En la mayoría de los casos, el compilador realiza un buen trabajo, pero siempre es bueno ayudarlo.

No existe una receta mágica que evite la utilización de una estructura condicional, pues un mismo método puede resultar en un mejor o peor desempeño dependiendo del problema.

A continuación revisaremos algunas alternativas que podemos probar.

Condicional if-else

Es la forma más común de afrontar un procesamiento distinto de acuerdo a una condición.

Suele utilizarse cuando varias instrucciones dependen de la condición.

Se evalúa la condición y se ejecuta el bloque de instrucciones que corresponda según la comprobación.

if (*condición*)
    instrucciones
else
    instrucciones

Operador Ternario

Al igual que en el caso anterior, el operador ternario permite realizar instrucciones distintas de acuerdo a una condición.

Se utiliza cuando se necesita evaluar una sola expresión dependiendo de la condición.

Es una pista para el compilador a favor de Predication.

(*condición*) ? *expresión 1* : *expresión 2*

Mulitplicación Booleana

Consiste en determinar el valor a utilizar en una asignación mediante una interpolación de Lagrange. Un valor booleano es tratado como 1 (True) o 0 (False) en una multiplicación.

Se utiliza en asignaciones a una misma variable cuyo valor dependerá de la condición, aunque se debe probar para determinar si es beneficioso para el desempeño.

variable = (*condición*) * *valor 1* + (1 - *condición*) * *valor 2*

Array Booleano

No corresponde a una alternativa a la estructura condicional, sino a una estrategia para afrontar condiciones costosas de verificar que serán necesarias en varias ejecuciones de kernels sin que sean modificadas durante la ejecución.

Consiste en calcular una única vez el resultado de las condiciones y almacenarlas en un arreglo que puede ser leído desde los kernels y utilizado con alguno de los tres métodos vistos anteriormente.

Conlleva un trade-off entre el uso de memoria y el tiempo de ejecución.

Latencia

La latencia corresponde al número de ciclos que le toma a una instrucción ejecutarse.

Una operación aritmética toma aproximadamente 20 ciclos, mientras que un acceso a memoria global toma entre 200 y 1400 ciclos.

Una hebra queda detenida esperando cuando no tiene los datos necesarios para ejecutar una instrucción.

Ocultación de Latencia

Para que la latencia no suponga un grave problema en nuestra aplicación, es necesario ocultarla.

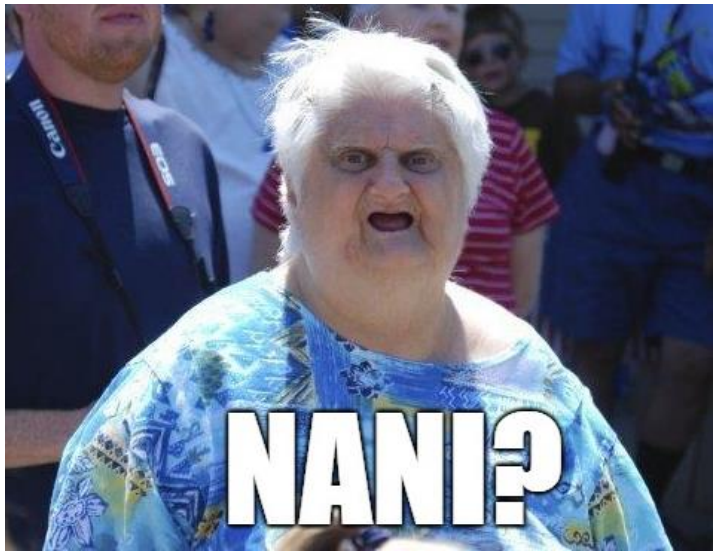
Esto se realiza mediante el intercambio de warps en ejecución. La idea principal es que mientras las hebras de un warp esperan que los datos estén listos, otros puedan ejecutar instrucciones.

Ocultación de Latencia

Ocultación de Latencia

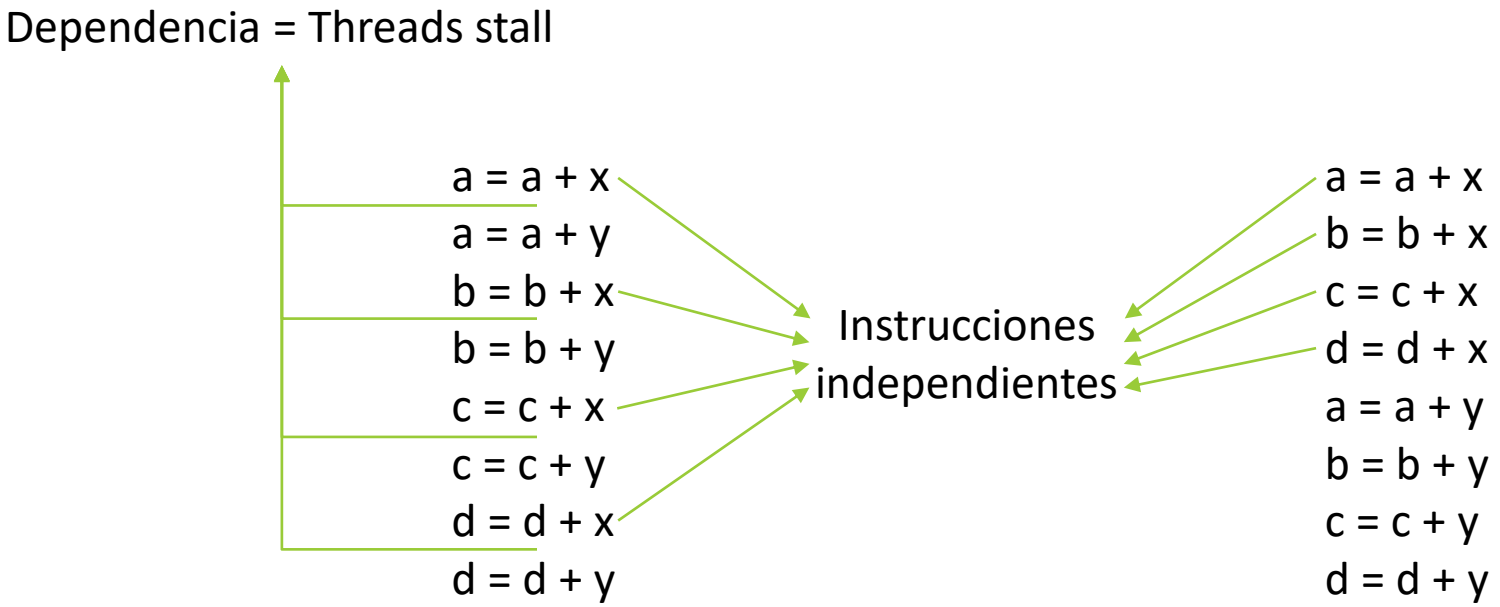
Para lograrlo podemos atacar de formas distintas:

- Aumentar el ancho de banda utilizado mediante el uso de distintas memorias y accesos optimizados (esto lo veremos en clases posteriores).
- Ordenando instrucciones de acuerdo a su dependencia.
- Utilizar menos threads.
- Utilizar más threads.



Paralelismo de Instrucciones

Las hebras no se detienen en los accesos a memoria, sino solo cuando hay dependencia de datos.



Menos Threads

Si nuestro kernel es simple, i.e. realiza pocas instrucciones y pocos accesos a memoria global. Una buena opción es aumentar el trabajo que realiza cada thread.

Por ejemplo, en vez de que cada thread procese un elemento de un array, hacer que, mediante un ciclo, cada thread se encargue de más elementos.

Esto reduce el número de threads necesario para procesar todos los datos.

```
globaldst[idx1] = globalsrc[idx1]
globaldst[idx2] = globalsrc[idx2]
```

```
a1 = globalsrc[idx1]
a2 = globalsrc[idx2]
globaldst[idx1] = a1
globaldst[idx2] = a2
```

Más Threads

Por el contrario, si el kernel se encarga de ejecutar múltiples instrucciones y cada thread procesa varios datos, probablemente signifique que utilizará una gran cantidad de registros.

Dado que esta memoria es muy rápida, pero muy pequeña, disminuir la cantidad de threads aumentando su carga de trabajo no sería una buena opción, pues al aumentar el uso de registros facilitaríamos su derrame.

Aumentar el número de threads sí podría ser una buena idea. Esto permitirá disminuir la cantidad de registros necesarios para cada thread y habrán más warps que podrán ejecutarse mientras los demás esperan datos.

Ocupancia

La ocupancia se define como el ratio entre la cantidad de warps activos en un SM y la cantidad máxima de warps activos que dicho SM puede soportar.

En palabras simples, es el porcentaje de utilización del SM respecto a los warps que se ejecutan en paralelo dentro de él.

Cálculo de Ocupancia

Como desarrolladores, nosotros elegimos la cantidad de threads por bloque (TPB). Esto a su vez nos indica cuantos warps habrán en cada bloque ($WPB = \lceil TPB/32 \rceil$).

Por otra parte, según que tan complejo sea el kernel a medir, cada thread utilizara una cierta cantidad de registros (RPT). Lo que también mide cuantos registros se utilizarán por bloque ($RPB = RPT * TPB$).

La cantidad máxima de registros con los que cuenta un SM ($RPSM_{MAX}$) nos indica cuantos bloques pueden ser asignados simultáneamente en un mismo SM ($BPSM = \lfloor RPSM_{MAX}/RPB \rfloor$) y al mismo tiempo la cantidad de warps activos en un SM ($WPSM = BPSM * WPB$).

Finalmente, dado el número máximo de warps activos que soporta un SM ($WPSM_{MAX}$), la ocupancia se obtiene como:

$$\text{Occupancy} = WPSM/WPSM_{MAX} * 100\%.$$

¿Cuánta Ocupancia?

A pesar de lo bonito y tentador que suena tener un 100% de ocupancia, esto en realidad no significa un mejor desempeño.

Una ocupancia muy baja resulta en problemas de eficiencia debido a que no existen suficientes warps disponibles para ocultar la latencia.

Una ocupancia alta empeora el rendimiento, pues hay menos recursos disponibles para cada thread.

Generalmente una ocupancia de aproximadamente 20% es suficiente para ocultar la latencia.