

Programación Avanzada en Paralelo

Cap. 1 Introducción:

M. en C. Germán Alonso Pinedo Díaz

Febrero 2023

Contenido

1 Detalles del curso

2 Paralelismo

- Conceptos Básicos

3 Arquitectura de computadoras

Presentación

Correo: german.pinedo@cinvestav.mx

WhatsApp: 3334971465

Trayectoria:

- Ingeniero en Comunicaciones y Electrónica por la Universidad Autónoma de Zacatecas
- Ingeniero de Procesos en Sanmina SCI, Manufactura de equipo Médico
- Grado de Maestro en Ciencias por el Cinvestav
- Doctorado en Ciencias en Cinvestav

Lineas de investigación:

- Diseño de circuitos en silicio de sistemas digitales
- Ciencia de datos, visión computacional y Deep Learning

Contenido

1 Detalles del curso

2 Paralelismo

- Conceptos Básicos

3 Arquitectura de computadoras

Contenido del Curso

Horario: Martes-Jueves de 7:00 a 9:00 (8:45)am

Detalles Curso: 32 Sesiones (64 Horas)

Contenido:

- ① Introducción al paralelismo y arquitecturas (4 sesiones)
- ② Programación en CUDA (12 sesiones)
- ③ Algoritmos en paralelo (4 sesiones)
- ④ Árboles y Grafos (4 sesiones)
- ⑤ Computación Geométrica (4 sesiones)
- ⑥ Álgebra Geométrica (2 sesiones)

Material de clase

- Programación C/C++
- Tarjeta GPU NVIDIA
- CUDA Toolkit Software
- Visual Studio
- GitHub

Opcional para visualización o aplicaciones:

- Cualquier Front-End que guste

Bibliografía

- Peter Pacheco. 2011. An Introduction to Parallel Programming (1st. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Andrist, B., & Sehr, V. (2020). C++ High Performance (2nd ed.). Packt Publishing.
- Sanders, J., Kandrot, E. (2010). CUDA by Example: An Introduction to General-Purpose GPU Programming. Upper Saddle River, NJ: Addison-Wesley.

Evaluación

Requisito 80 % de asistencia para derecho a examen

Asistencia se toma a las 7:10am

| Criterio de evaluación | % |
|------------------------|--------------|
| Tareas | 30 % |
| 2 Exámenes Parciales | 40 % (20-20) |
| Examen Final | 30 % |
| Total | 100 % |

- Examen Parcial: 50 % Teórico y 50 % práctico
- Examen Final: Proyecto Final, resolver 1 de 4 problemas

Contenido

1 Detalles del curso

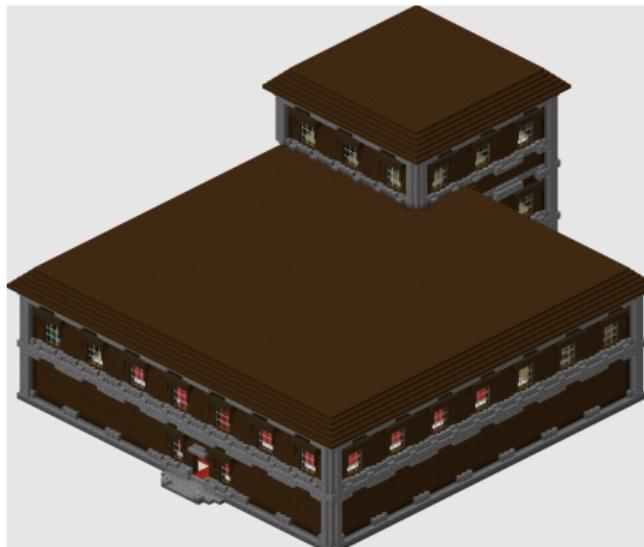
2 Paralelismo

- Conceptos Básicos

3 Arquitectura de computadoras

Conceptos Básicos

"Si un Steve construye una mansión en 1 día, ¿cuántos días se tardarían 10 Steve's?"



(a)

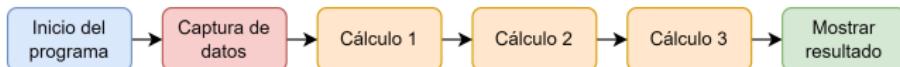


(b)

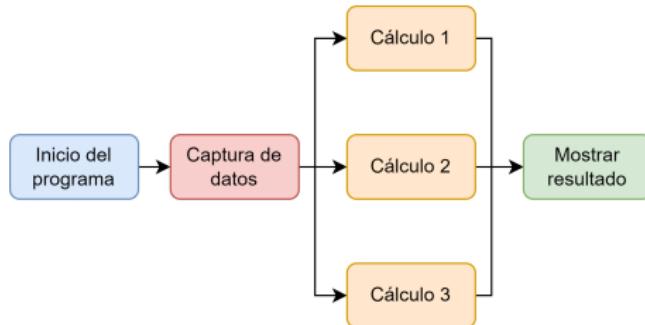
Conceptos Básicos

La idea de paralelismo en computación surge a partir de dividir operaciones de un proceso para terminar en menor tiempo.

- **Modelo Secuencial:**



- **Modelo Paralelo:**



Conceptos Básicos

- Idealmente se pueden dividir el paralelismo es lineal, es decir, si duplicamos la operación de elementos de un proceso, se reduce a la mitad el tiempo de ejecución.
- Sin embargo, muy pocos procesos paralelos logran una aceleración óptima.
- Es por eso que se busca parallelizar aquellas operaciones que cumplan con ciertas condiciones.

Conceptos Básicos

Por ejemplo, el tomate tiene un tiempo máximo de cosecha de aproximadamente 90 días. Entonces 3 personas no pueden cosechar un tomate en 1 mes.

Conceptos Básicos

Por ejemplo, el tomate tiene un tiempo máximo de cosecha de aproximadamente 90 días. Entonces 3 personas no pueden cosechar un tomate en 1 mes.

- Dependencia: *Ningún proceso se puede ejecutar más rápido que la ruta crítica*
- Ruta Critica: *La cadena más larga de cálculos dependientes*

Procesos

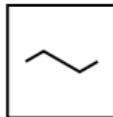
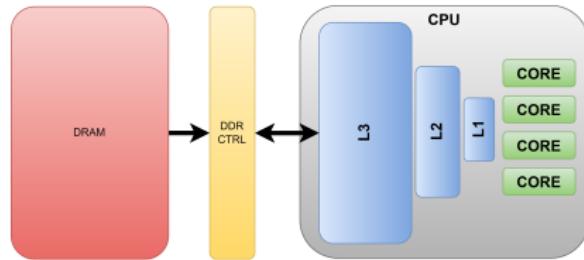
- En informática es un programa de ejecución
- Realiza una o varias tareas sobre datos
- Se ejecuta en uno o varios hilos
- Se comparten recursos de sistema

| Name | 100% CPU | 67% Memory |
|--------------------------------------|----------|------------|
| > Microsoft Visual Studio 2022 (...) | 0% | 348.9 MB |
| > Snipping Tool (3) | 0.4% | 23.0 MB |
| > Spotify (9) | 0.2% | 132.7 MB |
| > Task Manager | 1.4% | 67.8 MB |
| > Visual Studio Code (19) | 0% | 147.5 MB |
| > Windows Explorer (2) | 1.9% | 80.2 MB |

Los hilos de ejecución son una forma de dividir un proceso en varias partes que se ejecutan simultáneamente.

- Instrucciones atómicas
- Un proceso puede ser de uno o varios hilos
- Permite que un programa pueda tener varias instancias simultáneas
- Tienen su propio conjunto de registros, como la pila, el puntero de instrucción y el registro de estado
- Cada hilo tiene registro de estado para el sistema operativo

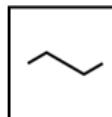
Hilos (Threads)



Un proceso
Un hilo



Un proceso
Multiples hilos

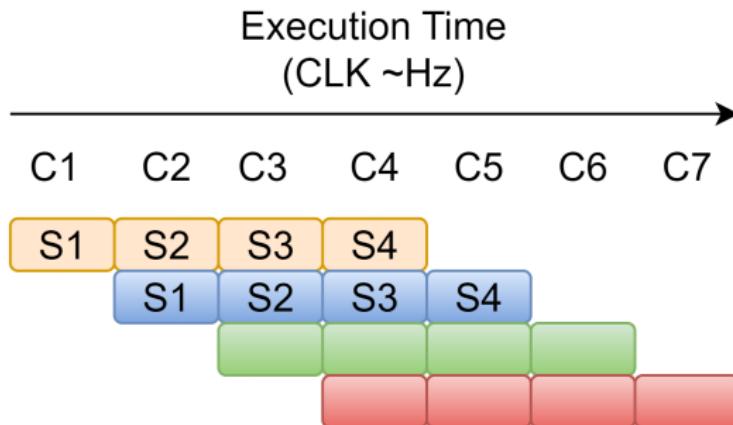


Multiples procesos
Un hilo

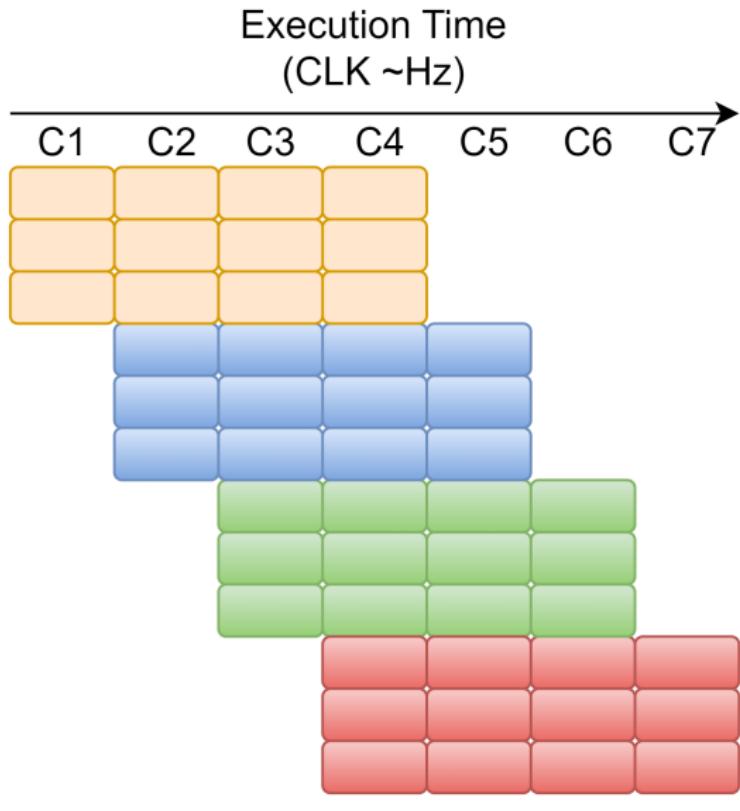


Multiples procesos
Multiples hilos

Pipeline

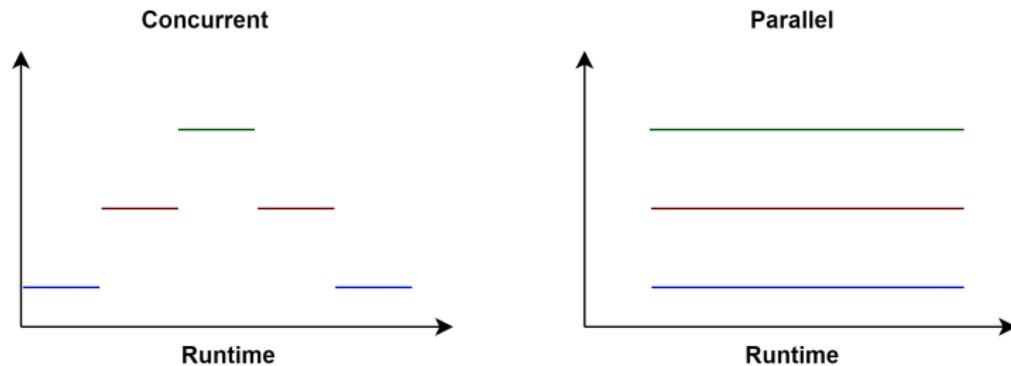


Pipeline



Concurrencia y Paralelismo

La concurrencia en informática se refiere a la capacidad de varios procesos o hilos de ejecución para acceder y utilizar recursos compartidos (como memoria, dispositivos de entrada/salida, etc.)



Contenido

1 Detalles del curso

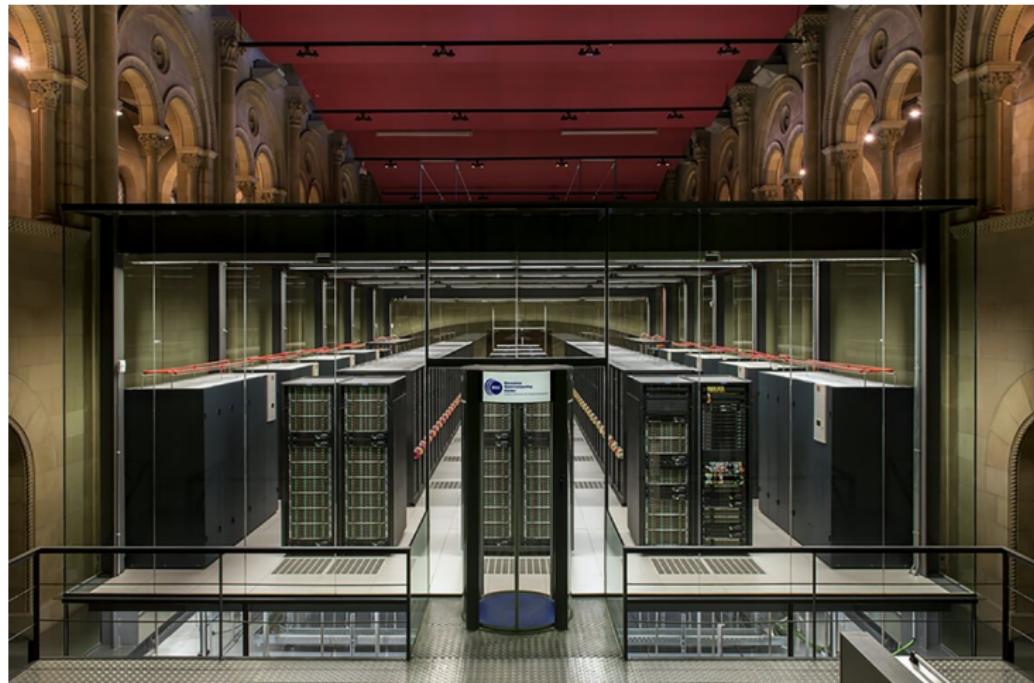
2 Paralelismo

- Conceptos Básicos

3 Arquitectura de computadoras

Super-computing

Una supercomputadora es una computadora con un alto nivel de rendimiento comparado a las computadoras de propósito general. Este rendimiento (performance) es medido en floating-operations per second.



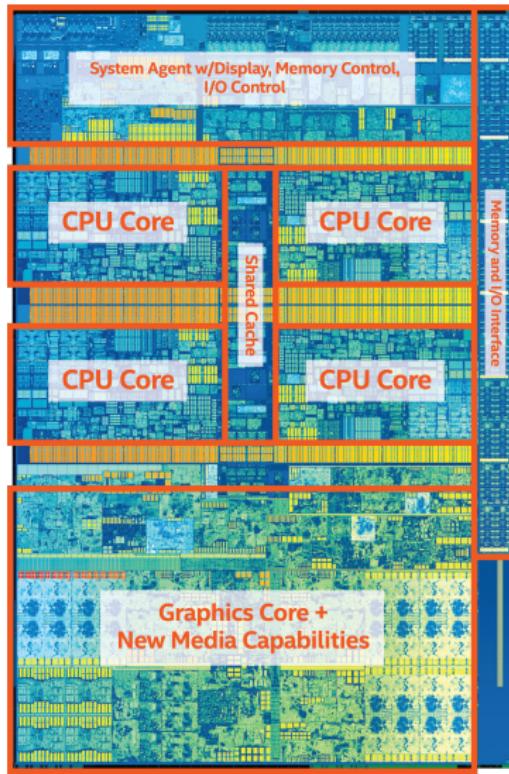
Super-computing

| Rank | System | Cores | Rmax (PFlop/s) | Rpeak (PFlop/s) | Power (kW) |
|------|--|-----------|-------------------|--------------------|---------------|
| 1 | Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States | 8,730,112 | 1,102.00 | 1,685.65 | 21,100 |
| 2 | Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan | 7,630,848 | 442.01 | 537.21 | 29,899 |
| 3 | LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland | 2,220,288 | 309.10 | 428.70 | 6,016 |
| 4 | Leonardo - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy | 1,463,616 | 174.70 | 255.75 | 5,610 |
| 5 | Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory | 2,414,592 | 148.60 | 200.79 | 10,096 |

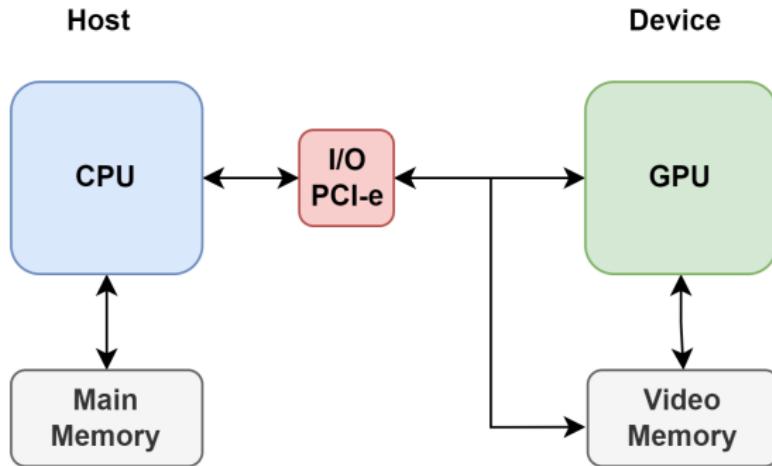
Super-computing

- Investigación biomédica
- Simulación aeroespacial
- Investigación genética
- Simulaciones físicas

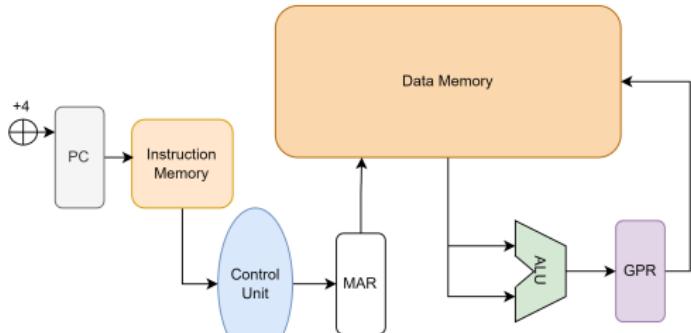
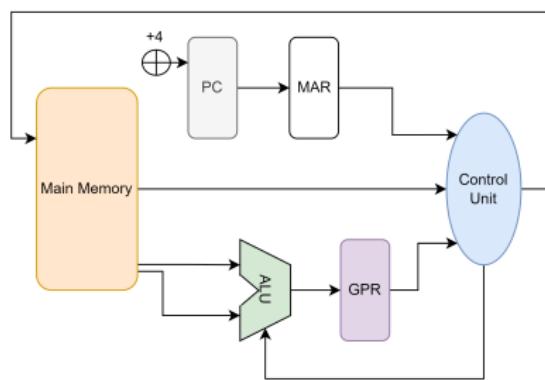
CPU



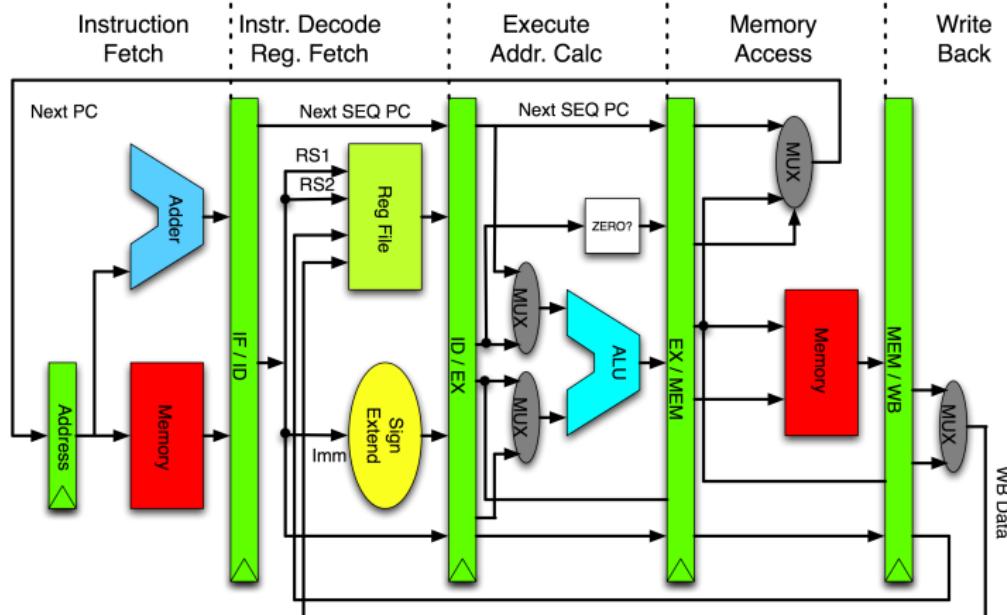
Host-Device



Von Neumann y Harvard



Arquitecturas Pipeline



GPGPU

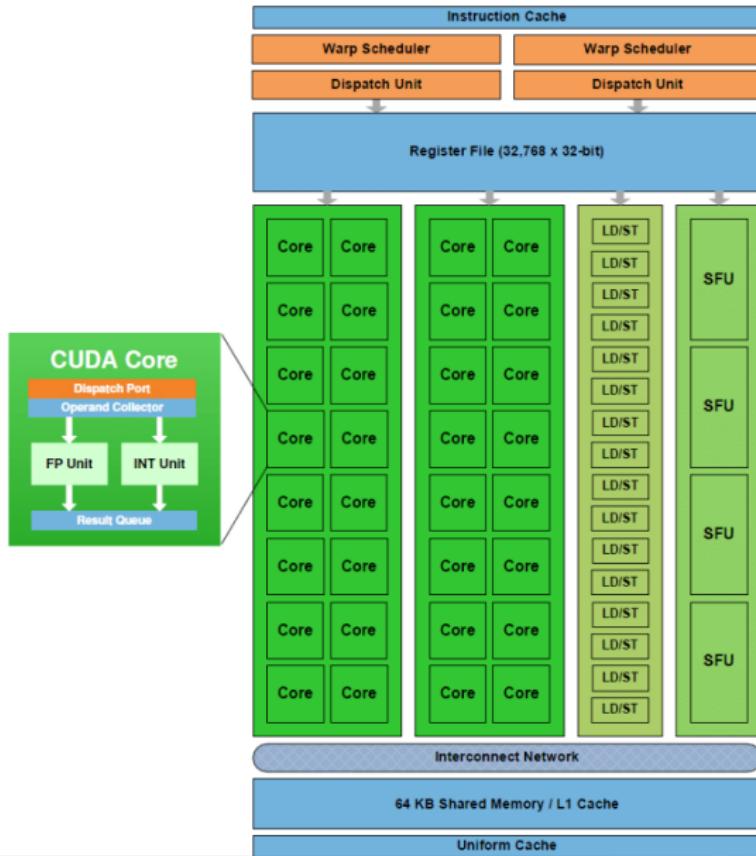


Streaming Multiprocessor

El SM o Streaming Multiprocessor es un procesador con propósito general que tiene una frecuencia baja y un caché pequeño. Su misión es ejecutar varios bloques de hilos en paralelo; en cuanto uno de sus bloques de hilos completa la ejecución, se pasa al siguiente bloque en serie.

- CUDA Cores
- FP Unit
- Int Unit
- Cache L1 de baja latencia
- Planificadores Warps
- Muchos registros

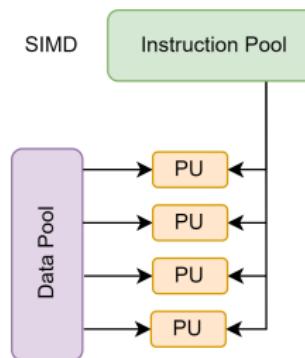
Streaming Multiprocessor



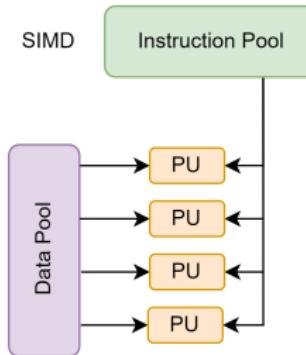
Taxonomía de Flynn

En paralelismo dividimos grandes problemas en pequeños para ejecutar simultáneamente.

| | Una instrucción | Multiples Instrucciones |
|--------------------|--------------------|----------------------------|
| Un Dato | SISD | MISD |
| Múltiples Datos | SIMD | MIMD |



Taxonomía de Flynn



- Todos los núcleos ejecutan la misma instrucción al mismo tiempo
- Solo se necesita decodificar la instrucción una única vez para todos los núcleos
- Ejemplo: Unidades vectoriales: suma, producto punto, etc.

Evolución

| Nombre | Año | nm | SM | Cores / SM | CUDA Cores | GFLOPS | Power (W) |
|--------------|------|----|-----|------------|------------|--------|-----------|
| Tesla | 2006 | 90 | 120 | 8 | 960 | 345 | 170 |
| Fermi | 2010 | 40 | 56 | 32 | 1792 | 1345 | 225 |
| Kepler | 2012 | 40 | 8 | 192 | 1536 | 3250 | 195 |
| Maxwell | 2014 | 28 | 24 | 128 | 3072 | 6060 | 200 |
| Pascal | 2016 | 28 | 28 | 128 | 3584 | 11340 | 250 |
| Turing | 2018 | 12 | 46 | 128 | 5888 | 134500 | 250 |
| Ampere | 2020 | 7 | 68 | 128 | 8704 | 297700 | 350 |
| Ada Lovelace | 2022 | 4 | 76 | 128 | 9728 | 487400 | 380 |

Programación Avanzada en Paralelo Cap 2. Programación en CUDA

M. en C. Germán Alonso Pinedo Díaz

Febrero 2023

4 Programación en CUDA

- Elementos básicos
- Organización de Threads
- Indexación
- Manejo de errores
- Timing
- Divergencia en Warps
- Latency hiding

4 Programación en CUDA

- Elementos básicos
- Organización de Threads
- Indexación
- Manejo de errores
- Timing
- Divergencia en Warps
- Latency hiding

Programación en CUDA

Pasos básicos de CUDA.

- Inicialización de datos desde CPU
- Transferencia de CPU a GPU
- Ejecutar Kernel con el tamaño de bloque/grid necesario
- Transferir resultados de GPU a CPU
- Limpiar la memoria

Programación en CUDA

Elementos de un programa de CUDA

- Host (función principal en CPU)
- Device (GPU)
- keywords `__global__`, `__host__`, `__device__`

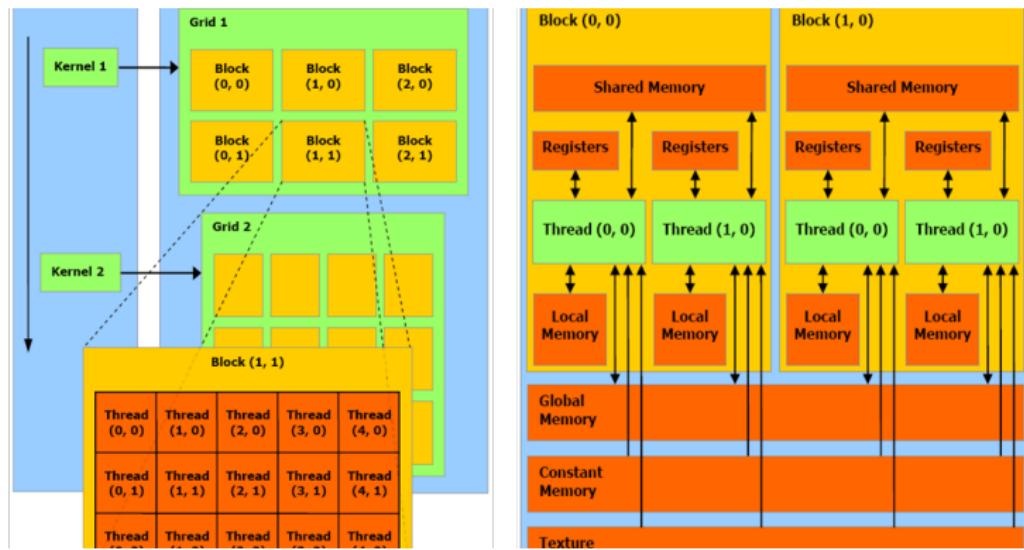
Programación en CUDA

- `__global__` - Corre en GPU, se llama desde el CPU o desde GPU. Ejecutado con argumentos de tipo `dim3` .
- `__device__` - Corre en GPU, se llama desde la GPU. Puede ser usada con variables también
- `__host__` - Corre en CPU, y se llama solo en CPU

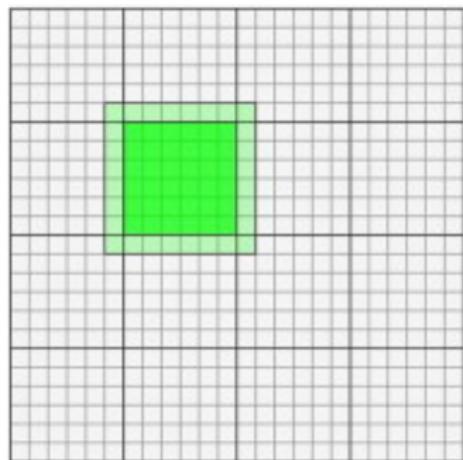
Programación en CUDA

- **Grid:** es una colección de todos los threads (hilos) para un kernel
- **Block:** son threads en un grid organizados en grupos

Esto permite a CUDA a sincronizar y administrar la carga de trabajo sin grandes penalizaciones en rendimiento.



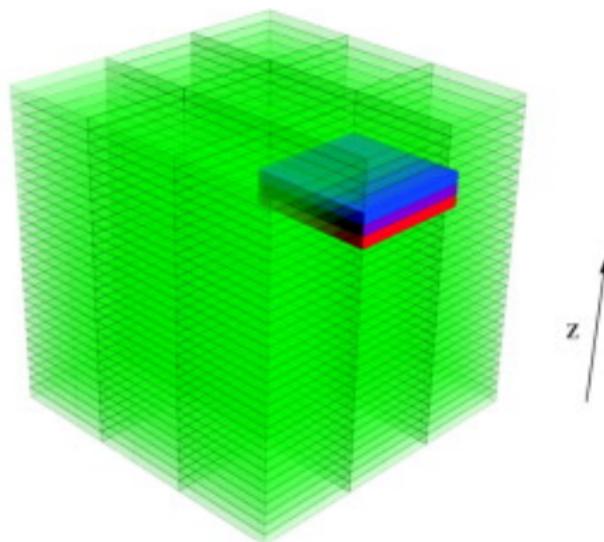
Programación en CUDA



→
`threadId.x`
`blockId.x`

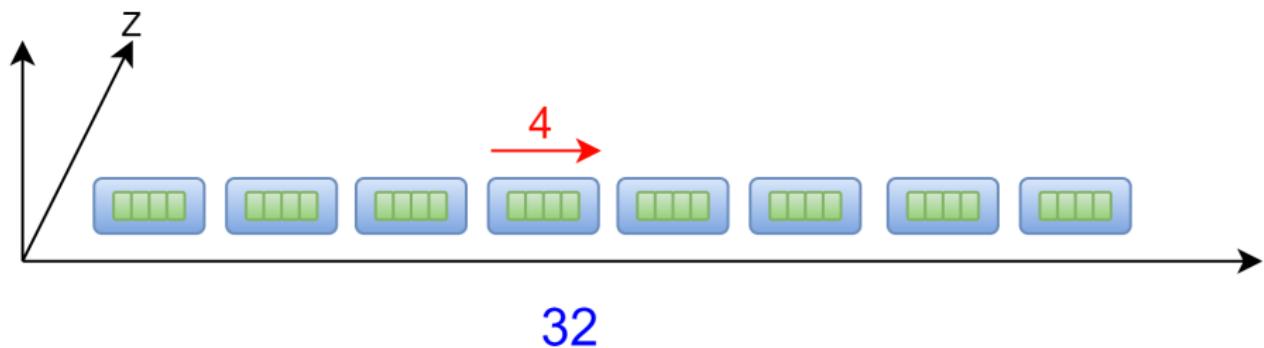
(a)

↑
`threadId.y`
`blockId.y`

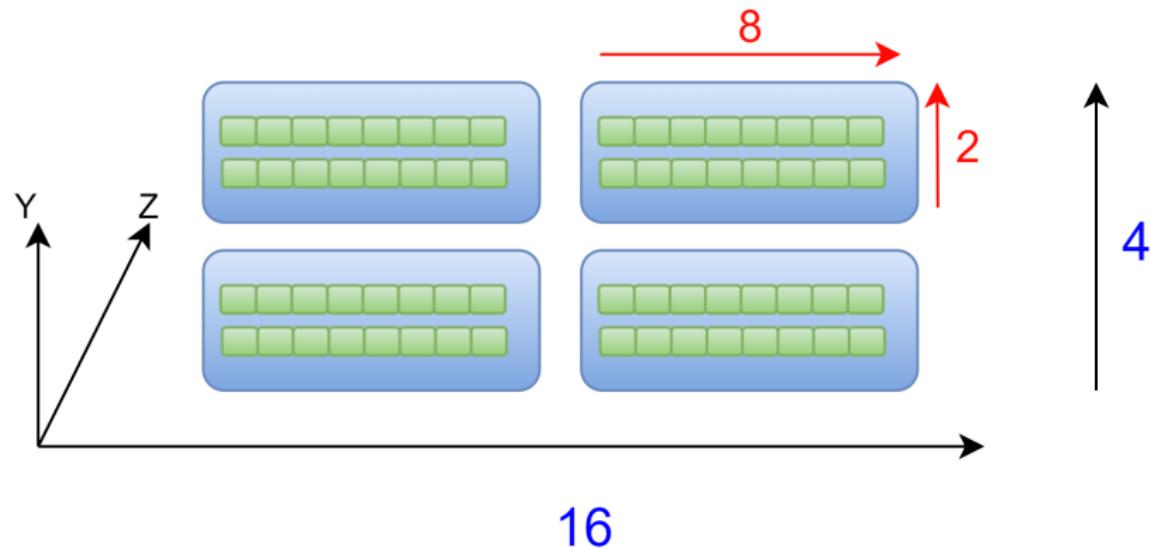


(b)

Programación en CUDA



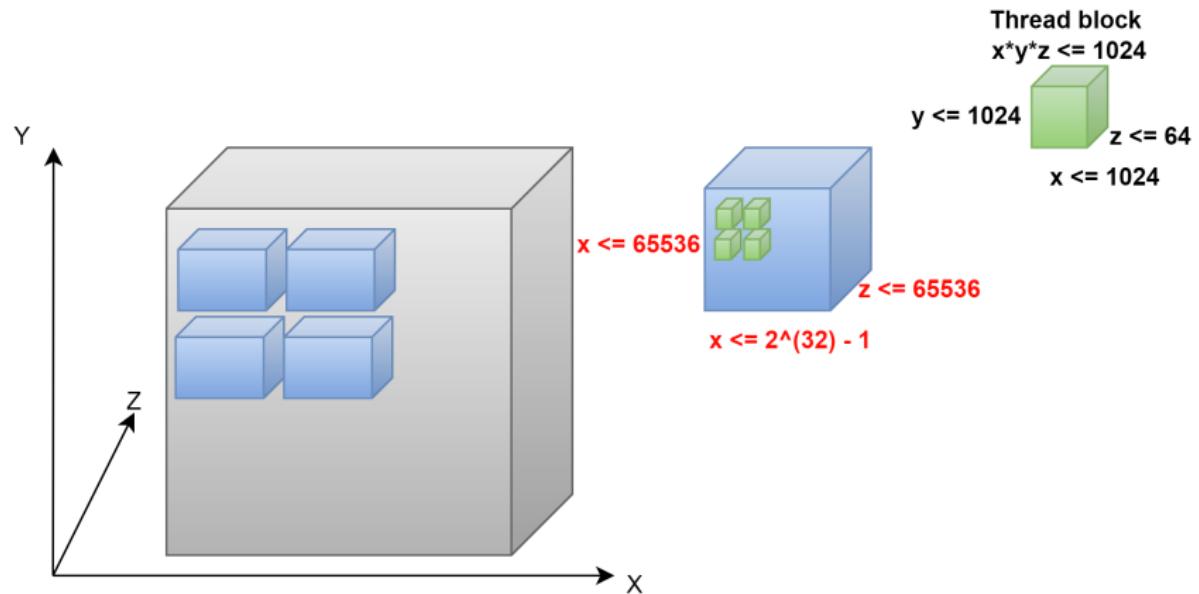
Programación en CUDA



Programación en CUDA

Note que estos ejemplos se realizan en pocos threads. Sin embargo, en el mundo de las aplicaciones, el grid tiene millones de threads y blocks. Pero tenga en cuenta que hay una limitación de numero de threads que podemos tener por bloque.

Programación en CUDA



Organización de threads

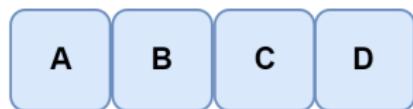
- En CUDA tenemos acceso a un set de variables.
- Estas variables son implícitamente inicializadas por CUDA runtime, basado en la ubicación del grid y thread block.
- Por ejemplo, Considere un grid de una dimensión con 8 threads.



| | | | | | | | | |
|------------|---|---|---|---|---|---|---|---|
| ThreadId.X | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| ThreadId.Y | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ThreadId.Z | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Organización de threads

- Ahora, considere un grid de una dimensión con 2 blocks de 4 threads.



ThreadIdx.X 0 1 2 3

ThreadIdx.Y 0 0 0 0

ThreadIdx.Z 0 0 0 0



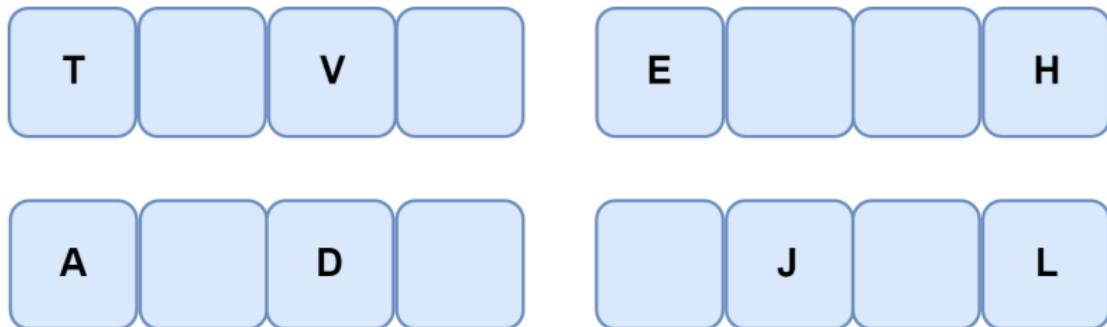
ThreadIdx.X 0 1 2 3

ThreadIdx.Y 0 0 0 0

ThreadIdx.Z 0 0 0 0

Threads, blocks and Grid

- Ahora, considere un grid 2D de 4 threads de 1D.

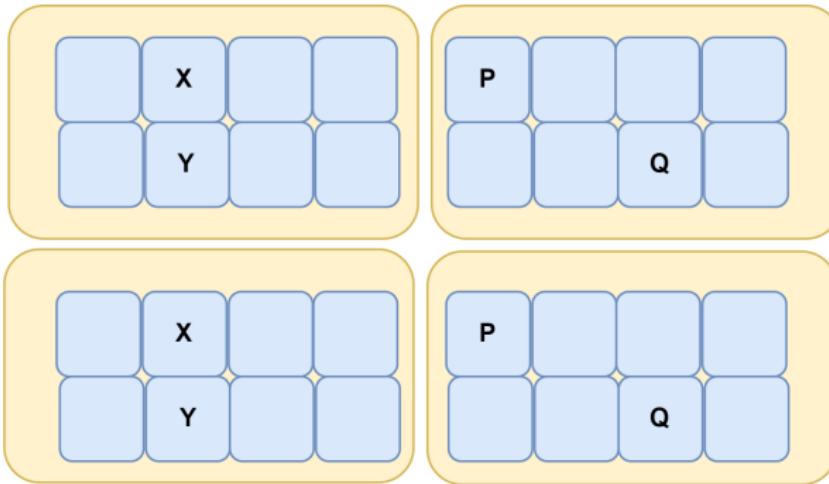


T V E H A D J L

ThreadIdx.X

ThreadIdx.Y

Organización de threads



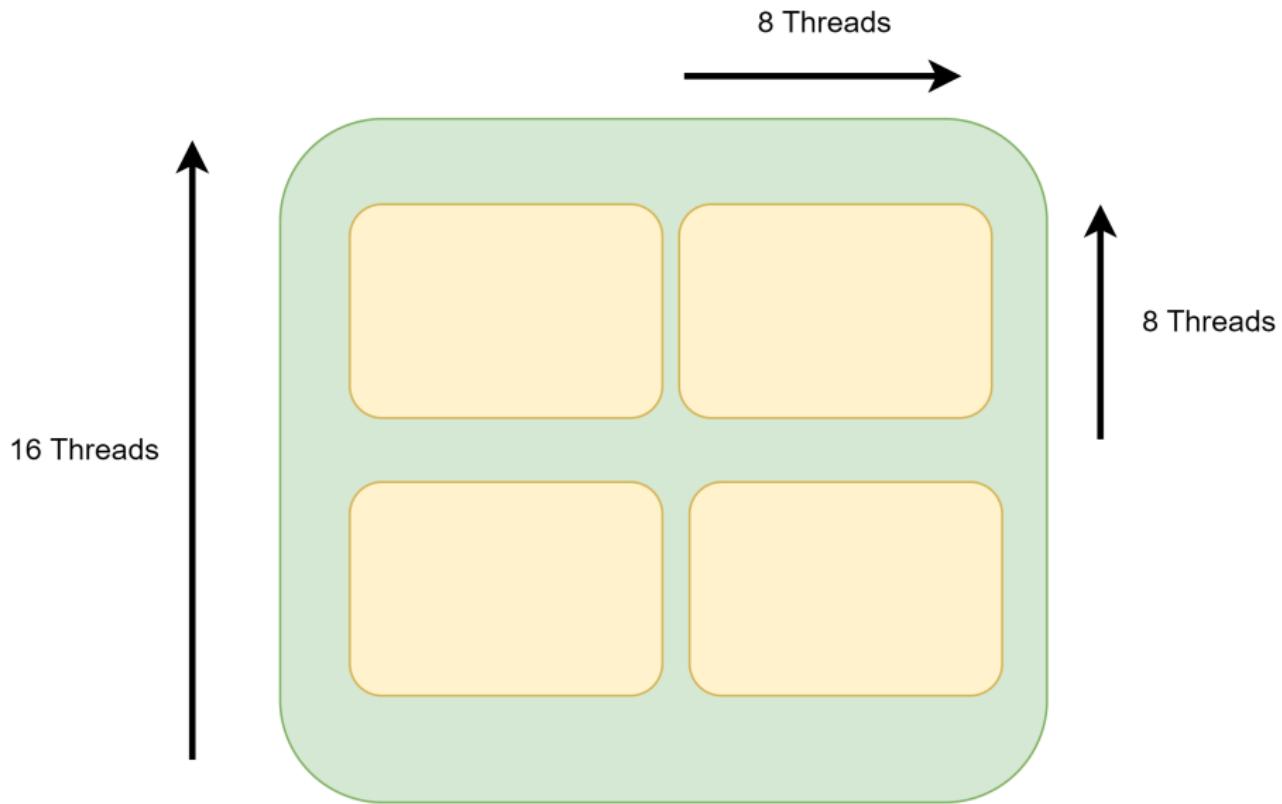
X Y P Q R S T U

ThreadIdx.X

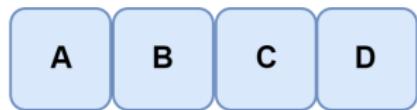
ThreadIdx.Y

ThreadIdx.Z

Organización de threads



Organización de bloques



blockIdx.X 0 1 2 3

blockIdx.Y 0 0 0 0

blockIdx.Z 0 0 0 0

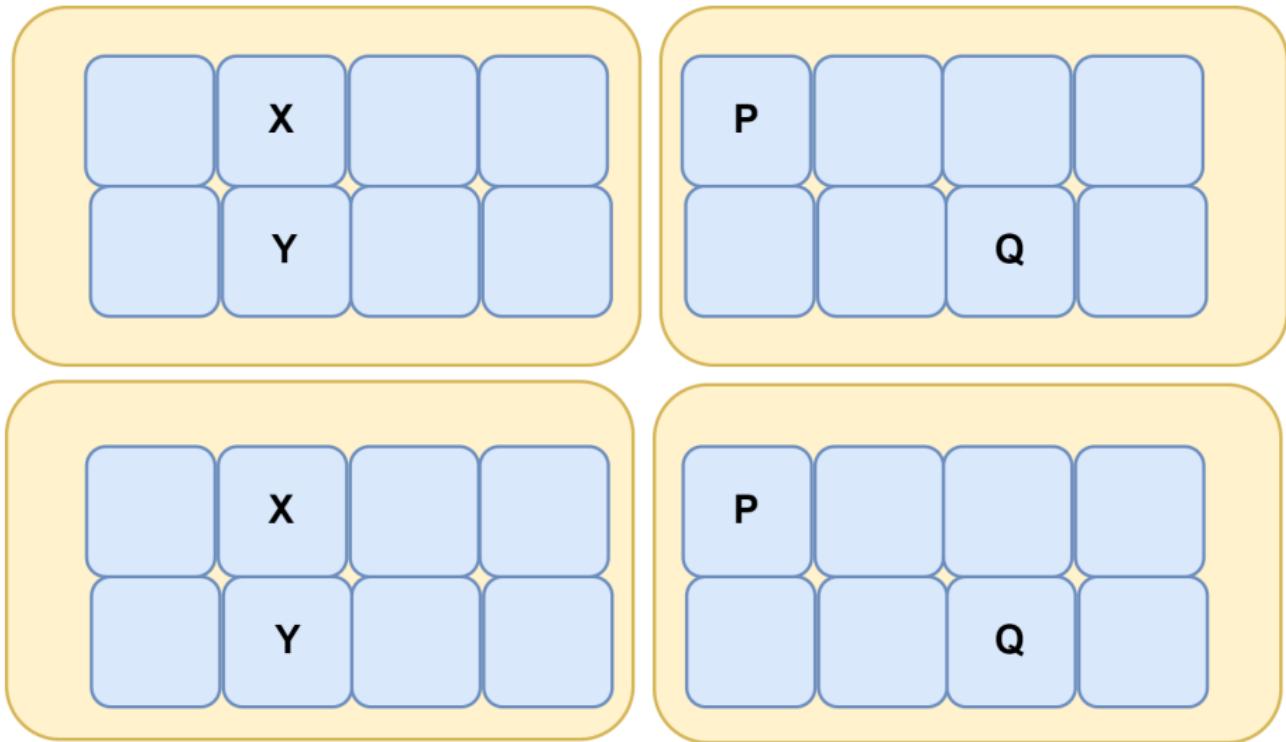


blockIdx.X 0 1 2 3

blockIdx.Y 0 0 0 0

blockIdx.Z 0 0 0 0

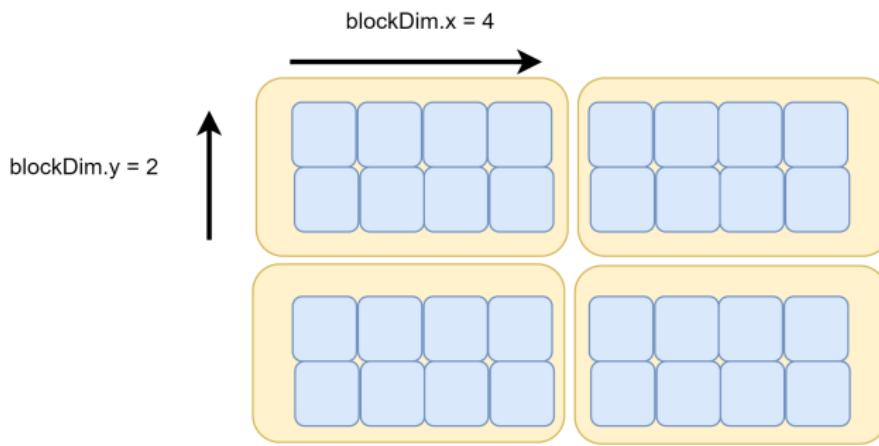
Organización de bloques



Organización de bloques

blockDim

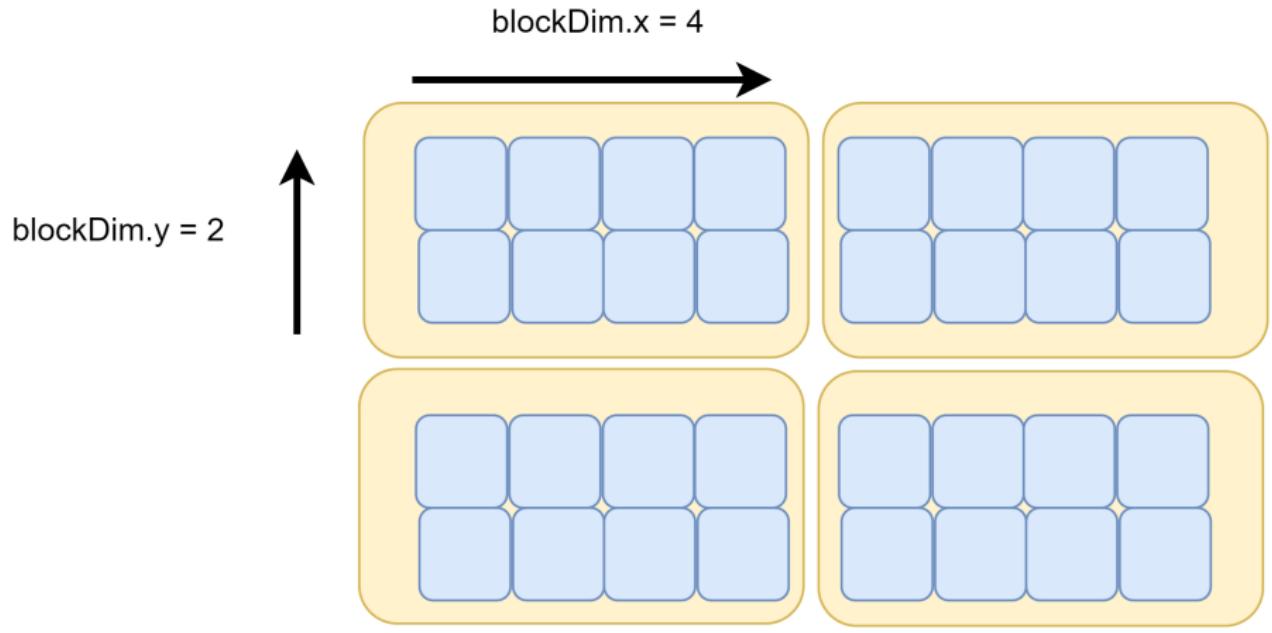
- Es una variable que consiste de numero de threads en cada dimensión de blocks.
- Todos los block en un grid tienen el mismo block size.
- El valor de esta variable es la misma para todos los threads en un grid.
- **blockDim** es de tipo dim3



Organización de grid

gridDim

- Consiste de el número de blocks en cada dimensión de un grid.
- **gridDim** es de tipo dim3



Ejercicio

Imprima el valor de threadIdx, blockIdx, gridDim para un grid de 3D que tiene 4 threads en las dimensiones X,Y, y Z. Y un block size sera de 2 threads en cada dimensión.

Planificación

Cada bloque es ejecutado en bloques de 32 llamados *Warp*s

- Es una implementación de HW y no depende de CUDA
- Es una unidad de planificación
- Si tenemos 3 bloques y cada uno tiene 256 threads, ¿cuántos warps hay?

Índices

En un programa de CUDA es muy común usar los valores de las variables *threadIdx.x*, *blockIdx*, *blockDim*, *gridDim*, para calcular arreglos.



threadIdx.x
(tid)

0 1 2 3 4 5 6 7



threadIdx.x
(tid)

0 1 2 3 0 1 2 3

Índices

En un programa de CUDA es muy común usar los valores de las variables *threadIdx*, *blockIdx*, *blockDim*, *gridDim*, para calcular arreglos.

```
__global__ void idx_calc_tid(int* input) {
    int tid = threadIdx.x;
    printf("[DEVICE] threadIdx.x: %d, data: %d\n", tid, input[tid]);
```

Índices

Sin embargo, tenemos que ubicar el índice de todos los threads en el grid de blocks. Para eso necesitamos calcular el offset de indexación.

| | | | | | | | | | | | | | | | | |
|-----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| | A | B | C | D | E | F | G | H | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| tid | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| gid | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

Índices

Sin embargo, tenemos que ubicar el índice de todos los threads en el grid de blocks. Para eso necesitamos calcular el offset de indexación.

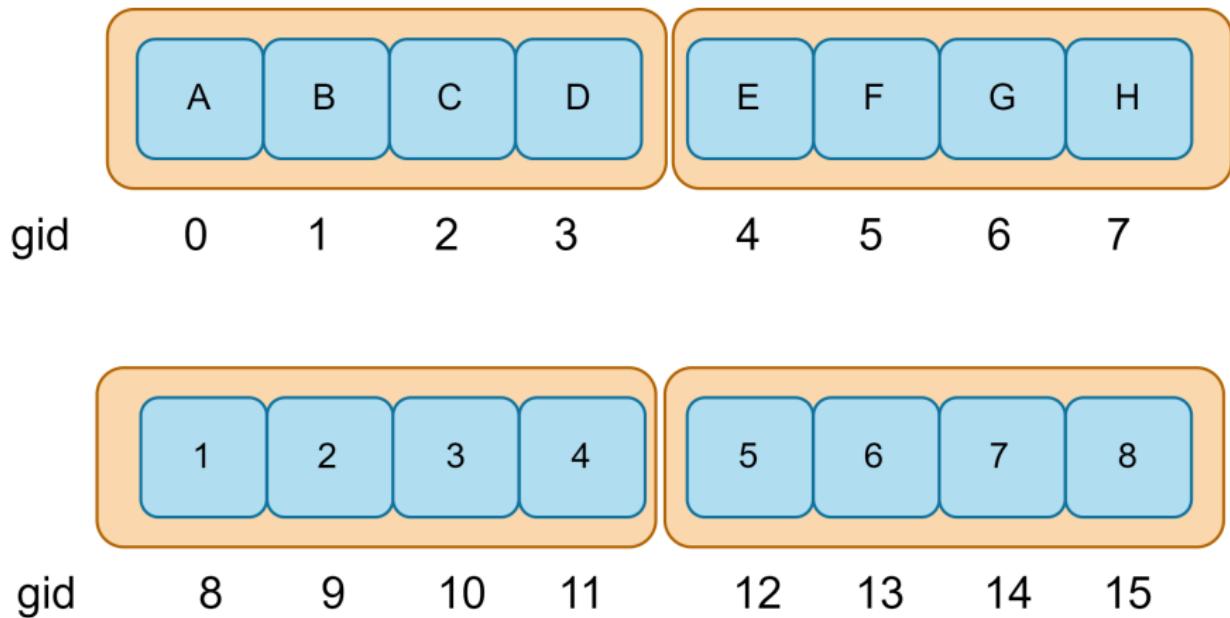
```
__global__ void idx_calc_gid(int* input) {
    int tid = threadIdx.x;
    // gid = tid + offset
    int offset = blockIdx.x * blockDim.x;
    int gid = tid + offset;

    printf("[DEVICE] blockIdx.x: %d, threadIdx.x: %d, gid: %d, data: %d\n\r",
           blockIdx.x, tid, gid, input[gid]);
}
```

Índices

Ahora en 2D, tenemos que calcular el índice de gid tomando en cuenta blockIdx.y

$$\text{Index} = \text{row offset} + \text{block offset} + \text{tid}$$



Índices

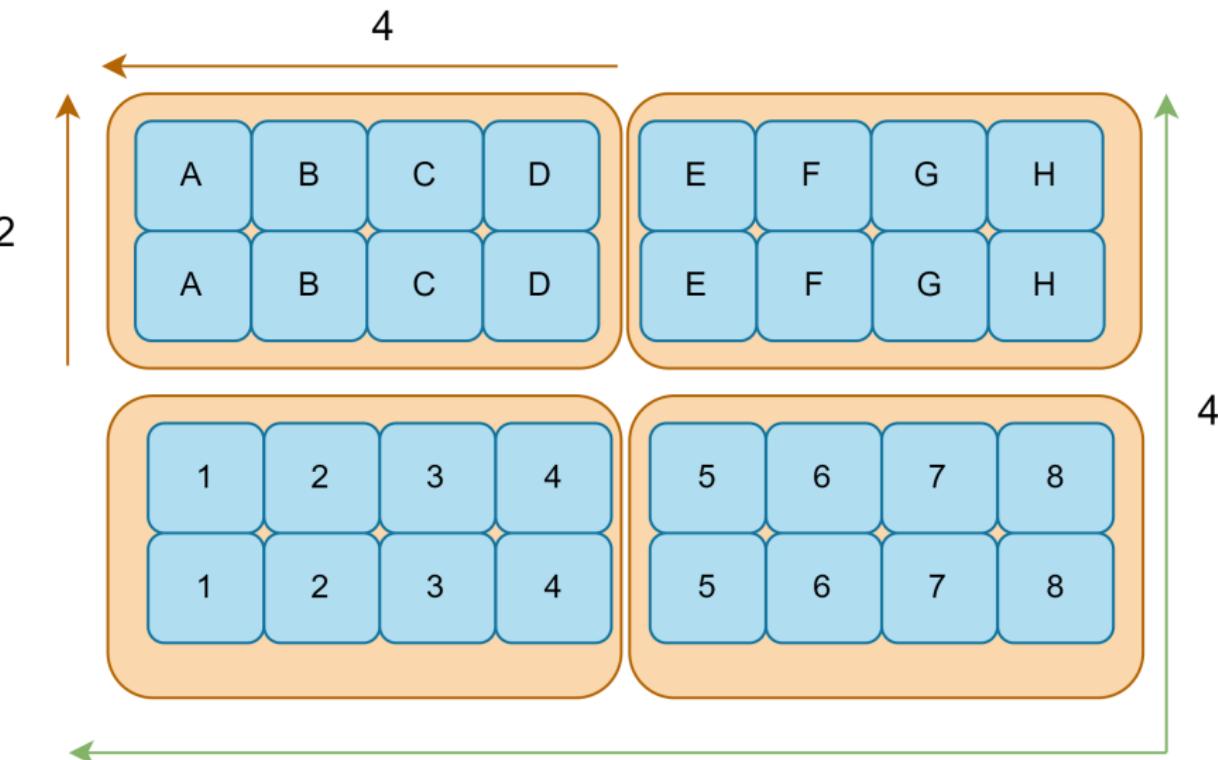
Ahora en 2D, tenemos que calcular el indice de gid tomando en cuenta blockIdx.y

```
__global__ void idx_calc_2d(int* input) {
    int tid = threadIdx.x;
    // gid = tid + offset
    int row_offset = gridDim.x * blockDim.x * blockIdx.y;
    int block_offset = blockDim.x * blockIdx.x;
    int gid = tid + row_offset + block_offset;

    printf("[DEVICE] gridDim.x: %d, blockIdx.x: %d, blockIdx.y: %d, threadIdx.x: %d, gid: %d, data: %d\n\r",
        gridDim.x, blockIdx.x, blockIdx.y, tid, gid, input[gid]);
}
```

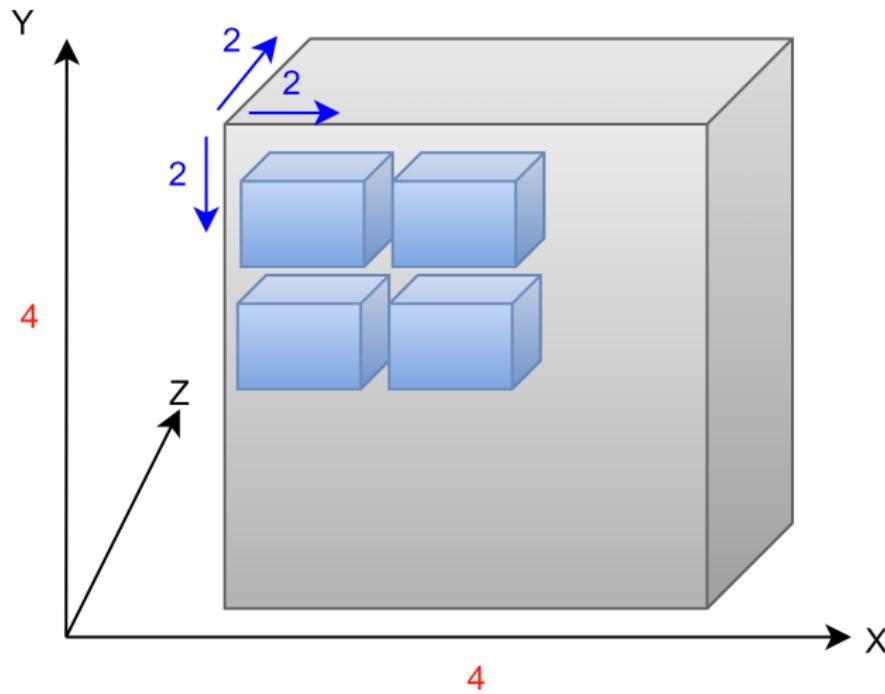
Índices

Ejercicio



Índices

Ejercicio



Ejercicio

- Realizar la suma de 3 vectores con 10k datos cada uno;
- Distribuya los hilos y bloques en 3 dimensiones;

Manejo de errores

Tipos de errores

- Errores en tiempo de compilación
 - Errores debidos a la sintaxis
- Errores en tiempo de ejecución
 - Suceden durante la ejecución del programa

Manejo de errores

Manejo de errores en CUDA

```
static __inline__ __host__ cudaError_t cudaMalloc(
    T      **devPtr,
    size_t   size
)
```

```
cudaGetString(error);
```

Timing en CUDA

En la programación CUDA, generalmente queremos comparar el rendimiento entre la implementación de GPU con la implementación de CPU.

Además, en el caso de que tengamos múltiples soluciones para resolver el mismo problema, entonces también queríamos encontrar la solución más rápida o con mejor rendimiento.

```
clock start = clock()  
//carga de trabajo  
clock stop = clock()  
diferencia = end - start  
time = (diferencia / clocks_per_second)  
FPS = 1 / time
```

Rendimiento de una aplicación de CUDA

- Tiempo de ejecución
- Consumo de potencia
- Utilización de hardware
- Costo de hardware

Probar distintas configuraciones de:

Distribuir grid, blocks, memoria compartida, cache, acceso a memoria

Tarea

En la tarea, debe implementar la suma de matrices en la GPU, que puede sumar 3 arrays de 21000 datos.

Debe usar mecanismos de manejo de errores, mecanismos de medición de tiempo también.

Luego, debe medir el tiempo de ejecución de sus implementaciones de GPU.

Ofrecer la configuración para el mejor rendimiento (hacer una tabla con diferentes combinaciones)

Divergencia en Warps

Si Threads en el mismo kernel de CUDA ejecutan diferentes instrucciones, entonces tenemos una divergencia

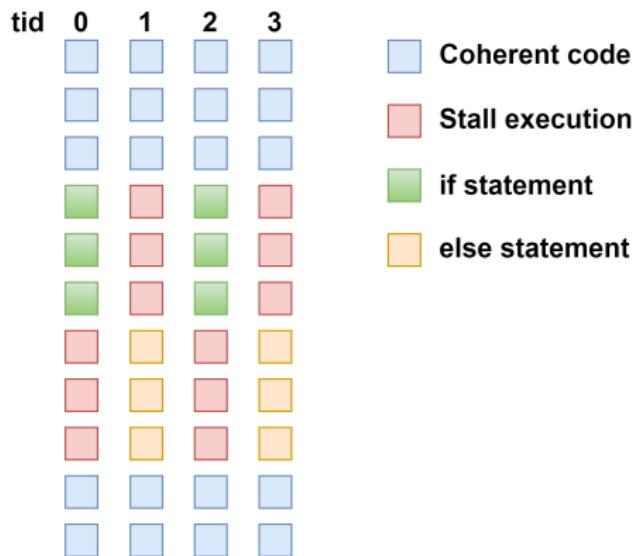
```
if tid < 16 then
    //Statement
else
    //Statement
end if
```

Divergencia

Aquí tenemos una verificación de condición que obliga a los subprocessos con un número impar como identificación de subprocesso a seguir un camino diferente al de los subprocessos con un número par como identificación de subprocesso.

```
int tid = threadIdx.x
if tid %2 == 0 then
    //Statement
else
    //Statement
end if
```

Divergencia



Divergencia

En este ejemplo, la cantidad de paralelismo en la ejecución Warp se redujo a la mitad. Solo 16 subprocessos se estaban ejecutando activamente a la vez, mientras que otros 16 estaban deshabilitados. Con más ramas condicionales la pérdida de paralelismo sería aún mayor.

Nota: tener cuidado con la presencia de declaraciones de flujo de control como las declaraciones if y else cambian las declaraciones, lo que nos da una pista de código divergente. Pero no siempre estas declaraciones generan una divergencia de Warp

Divergencia

Por lo tanto, las comprobaciones de condición que dan como resultado que todos los subprocessos de una deformación se ejecuten en la misma ruta, no inducirán ninguna divergencia de Warp.

Considere el siguiente ejemplo:

```
if tid/32 < 0 then  
    //Statement  
else  
    //Statement  
end if
```

Warp 0

0

31

Warp 1

0

31

Branch Efficiency

Un Branch es una ruta de ejecución de un kernel de CUDA.

Por ejemplo, si tenemos una condición para forzar la ejecución de los threads pares e impares, entonces tenemos dos branches.

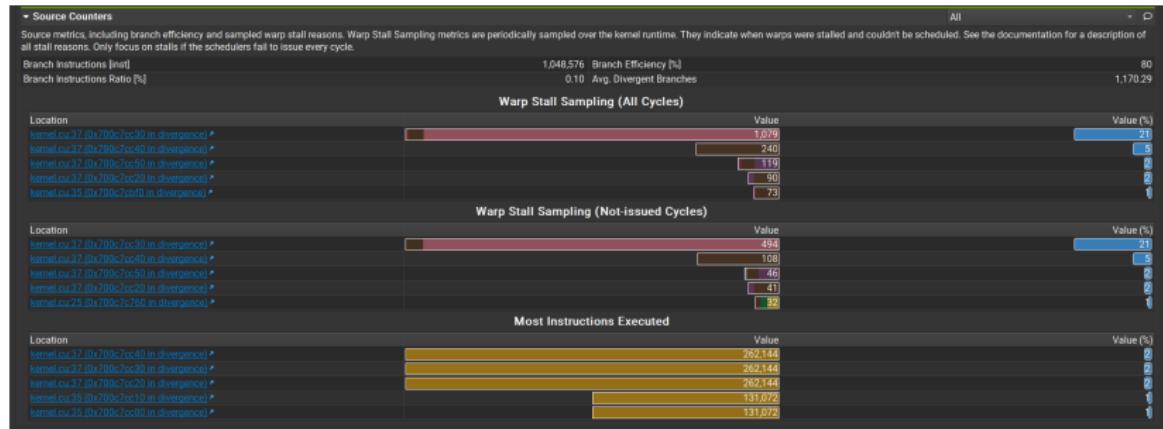
```
int tid = threadIdx.x  
if tid %2 == 0 then  
    //Statement  
else  
    //Statement  
end if
```

$$B_e(\%) = \frac{(N_B - B_{div})}{N_B} \times 100 \quad (1)$$

donde N_B es el número de branches B_{div} son los branches divergentes, y $B_e(\%)$ es el branch efficiency.

Branch efficiency

Se puede medir si su núcleo tiene un código divergente o no, utilizando la métrica de rendimiento con la herramienta NVIDIA Nsight Compute y el NVIDIA Profiler.



Partición de recursos y Latency hiding

El contexto de ejecución local de un warp consiste principalmente en los siguientes recursos.

- Program Counters
- Registers
- Shared Memory

El contexto de ejecución de cada warp procesado por un SM se mantiene en el chip durante toda la vida útil de un warp. Por lo tanto, cambiar de un contexto de ejecución a otro no tiene costo. Se debe aligerar la carga de los threads de la GPU para que pueda ejecutar múltiples contextos.

Registros

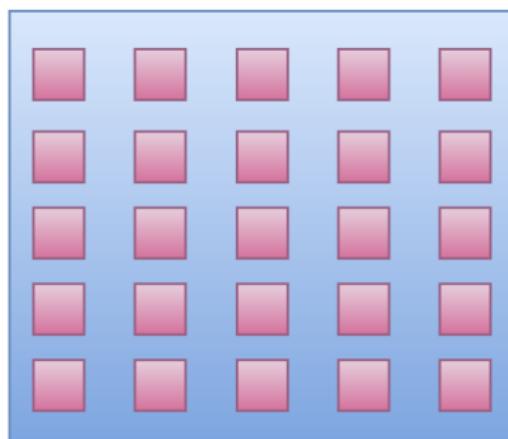
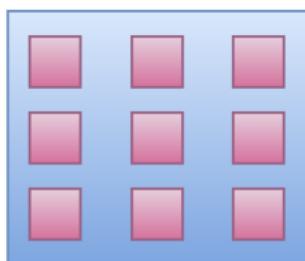
- Los Registros y la Shared Memory pueden ser controlados directamente por el programador
- Cada SM tiene un conjunto de registros de 32 bits almacenados en un archivo de registro que se dividen entre subprocessos y una cantidad fija de memoria compartida que se divide en bloques de subprocessos.
- Los WARPS que pueden residir simultáneamente en un SM para un kernel dado dependen de la cantidad de registros y la cantidad de memoria compartida disponible en SM y requerida por el kernel.

Warps

Como se ve en este ejemplo, si cada thread por kernel consume menos registros y menos warps en un solo SM.

Si usted puede reducir el numero de registros, el kernel consume más warps procesados simultáneamente.

Menos warps con mas registros
disponibles por thread



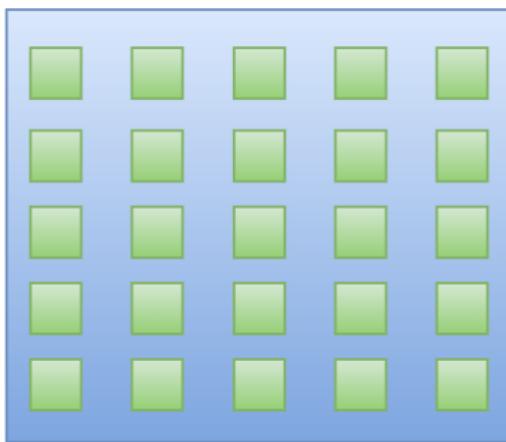
Mas warps con menos registros
disponibles por thread

Thread-Blocks

Por otra parte, si un Thread-Block consume mas memoria compartida, menos Thread-Blocks son procesados simultáneamente por un SM.

Si usted puede reducir la cantidad de memoria compartida usada por cada Thread-Block, entonces mas Thread-Blocks pueden ser procesados simultáneamente.

Menos Thread Blocks con mas
memoria compartida



Mas Thread Blocks con menos

Thread-blocks

Lo que esto significa:

- La disponibilidad de recursos limita la cantidad de bloques por SM
- El máximo número de registros y la cantidad de memoria compartida por SM

Esto varía del dispositivo y la versión del *Compute Capability*

Recursos disponibles

Si hay registros insuficientes o memoria compartida en cada SM para procesar al menos un Thread-Block, entonces el launch del kernel fallará.

Esta tabla enumera las limitaciones técnicas por SM en las diferentes versiones de *Compute Capability*. Esto se puede obtener en la función de las propiedades de dispositivo

| Especificación | 3 | 3.5 | 5 | 6 | 7 |
|---------------------------------|-----|-----|-----|-----|-----|
| Max. Blocks concurrentes por SM | 16 | 16 | 32 | 32 | 32 |
| Max. Warps concurrentes por SM | 64 | 64 | 64 | 64 | 64 |
| No. de registros por SM | 64K | 64K | 64K | 64K | 64K |
| Max. Registros por Thread | 63 | 255 | 255 | 255 | 255 |
| Shared Memory por SM | 48K | 48K | 64K | 64K | 96K |

Categorías de Warps en SM

- Active blocks/warps
 - Selected warp
 - Stalled warp
 - Elegible warp

Condiciones para ser “Elegible Warps”

- 32 CUDA cores deben estar libres
- Todos los argumentos para la instrucción debe estar lista

¿Qué es Latencia?

Latency

¿Qué es Latencia?

Es el número de ciclos de reloj entre el inicio y fin de la instrucción.

- Latencia de instrucción aritmética
- Latencia en la operación de memoria
-

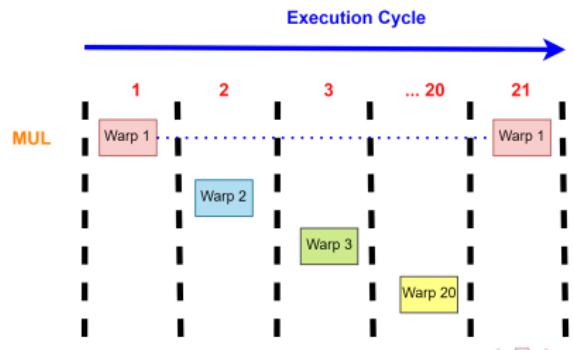
| OpCode | Tesla | Fermi | Keppler | Maxwell |
|--------------|-------|-------|---------|---------|
| Add/Sub | 24 | 16 | 9 | 6 |
| Max/Min | 24 | 18 | 9 | 12 |
| MAD | 120 | 22 | 9 | 13 |
| Mul | 96 | 20 | 9 | 13 |
| Div | 608 | 286 | 141 | 210 |
| Rem | 728 | 280 | 138 | 202 |
| AND, OR, XOR | 24 | 16 | 9 | 6 |

Latencia de memoria

| Unidad de memoria | Tesla | Fermi | Keppler | Maxwell |
|----------------------------------|--------------|--------------|----------------|----------------|
| Global & Local Memory | | | | |
| L1 | - | 45 | 30 | - |
| L2 | 440 | 685 | 300 | 350 |
| Shared Memory | | | | |
| SMEM | 38 | 50 | 33 | 28 |
| Constant Memory | | | | |
| L1 | 56 | 52 | 42 | 28 |
| L1.5 | 128 | 165 | 104 | 76 |
| L2 | 268 | 375 | 215 | 184 |

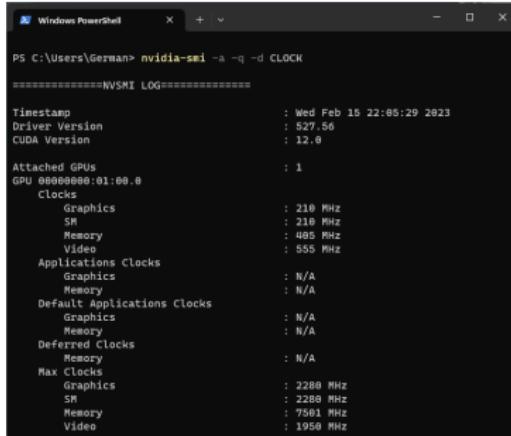
Latency Hiding

- El contexto de ejecución de cada warp procesado por SM se mantiene en el chip durante toda la vida útil del warp.
- Por lo tanto cambiar de contexto de ejecución a otro no tiene costo
- 1SM - 128 CUDA Cores - 4 Warps en un SM
- 20 *Elegible Warps* para Latency Hiding
- $4 \times 20 = 80$ *Elegible Warps* Necesitamos para el Latency Hiding en por SM
- $13 \times 80 = 1040$ *Elegible Warps* Necesitamos para el Latency Hiding en el dispositivo



Latency Hiding en Memoria

Considere la microarquitectura Ampere en la GPU GTX 1650 con una velocidad de transferencia de 128GB/s.



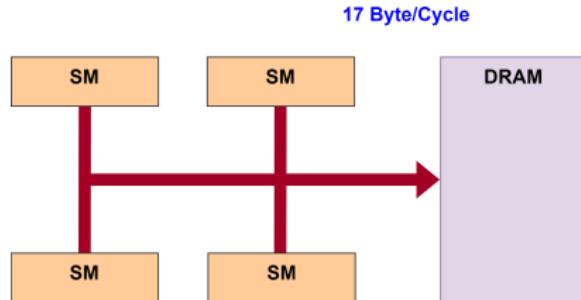
```
PS C:\Users\German> nvidia-smi -a -q -d CLOCK
=====
NVSMI LOG =====

Timestamp : Wed Feb 16 22:05:29 2023
Driver Version : 527.56
CUDA Version : 12.0

Attached GPUs : 1
GPU 00000000:01:00.0
    Clocks
        Graphics : 210 MHz
        SM : 210 MHz
        Memory : 405 MHz
        Video : 555 MHz
    Applications Clocks
        Graphics : N/A
        Memory : N/A
    Default Applications Clocks
        Graphics : N/A
        Memory : N/A
    Deferred Clocks
        Memory : N/A
    Max Clocks
        Graphics : 2280 MHz
        SM : 2280 MHz
        Memory : 7501 MHz
        Video : 1950 MHz
```

¿Cuál es la latencia por byte?

Latency Hiding en Memoria



- $17 \times 350 = 5950$
- $5950 / 4 = 1488$ threads
- $1488 / 32 = 47$ Warps
- $47 / 13 = 12$ warps per SM

Entonces, necesitamos 12 *elegible warps* para el Latency Hiding en Memoria. En ese sentido, podemos calcular cuantos warps necesitamos para *hide* las latencias aritméticas y de memoria.

Categorías de aplicaciones CUDA

- Aplicaciones con límite de ancho de banda
- Aplicaciones con límite de cálculos

Una vez que clasificacmos nuestra aplicación en una de esas categorías, podemos centrarnos primero en optimizar la aplicación siguiendo los pasos específicos para optimizar esa categoría de aplicaciones.

Ocupación

Es la relación entre Warps activos al numero máximo de warps por SM.
En un núcleo CUDA dado, las instrucciones se ejecutan secuencialmente.

$$Occupancy = \frac{ActiveWarp}{Max.Warps} \quad (2)$$

- Cuando un warp se detiene debido a la latencia en la operación aritmética o de memoria, ese SM cambia de contexto a otro warp elegible.
- Si un warp se detiene, queremos tener una cantidad suficiente de warps para mantener los núcleos ocupados
- Si la ocupación de un kernel en particular es un valor alto, eso significa que, aunque un warp detenga la ejecución, habrá otro warp elegible, por lo que los núcleos siempre estarán ocupados.

Ocupación

Tenemos un kernel que usa 48 registros por thread y 4096 bytes de shared memory por bloque. Un block size de 128

- $Reg_{warps} = 48 \times 32 = 1536$
- Para nuestro dispositivo tenemos 65536 registros por SM
- warps permitidos $= 65536 / 1536 = 42,67 - > 40$
- bytes de shared memory por SM $= 98304$
- Active blocks $= 98304 / 4096 = 24$
- Si consideramos 128 threads por block que es igual a 4 warps por block.
- Active warps $= 24 \times 4 = 96$
- Sin embargo, recuerde que en CC 7+, el máximo numero de warps activos por SM es de 64.
- Por lo tanto, los Warps activos no está limitado por la memoria compartida.
- El número de registros limita los warps activos
- Occupancy $= 40 / 96 = 63\%$

Ocupación

- Mantener el número de threads por block como múltiplo de 32
- Evitar block_size pequeños: iniciar con 128 o 256 threads por bloque
- Ajustar el block_size arriba o abajo de acuerdo con los requerimientos del kernel
- Declarar el numero de bloques mucho mayor que el número de SM para tener el suficiente paralelismo en el dispositivo
- Realizar experimentos para descubrir la mejor configuracion de ejecucion y uso de recursos