Arquitectura de GPUs

Carlos Bederián, Nicolás Wolovick

FaMAF, Universidad Nacional de Córdoba, Argentina

1 de Agosto de 2012

ECAR12@DC.UBA

Revisión 3739, 2012-08-02

Introducción

CPU modernas, cómputo

Por ejemplo Intel Sandy Bridge™ o AMD Bulldozer™.

ILP

- Pipelining con una docena de fases.
- Ejecución fuera de órden (OOE).

  Un scheduler que entiende X86_64 y analiza dependencia de datos.
- Superescalares. Muchas unidades de ejecución..

  3 puertos cómputo, 3 puertos memoria, para Sandy Bridge.
- Predictor de saltos.

  Análisis estadístico realtime para evitar costosos *pipeline stalls*.
- Dos juegos de registros para correr dos hilos y aumentar la utilización de las unidades de ejecución. Intel Hyperthreading™.

FPU

- Unidades independientes de multiplicación y suma.

  1 MAD o sea 2 FLOP por ciclo.
- Unidades vectoriales AVX, 8 `fp32` a la vez.

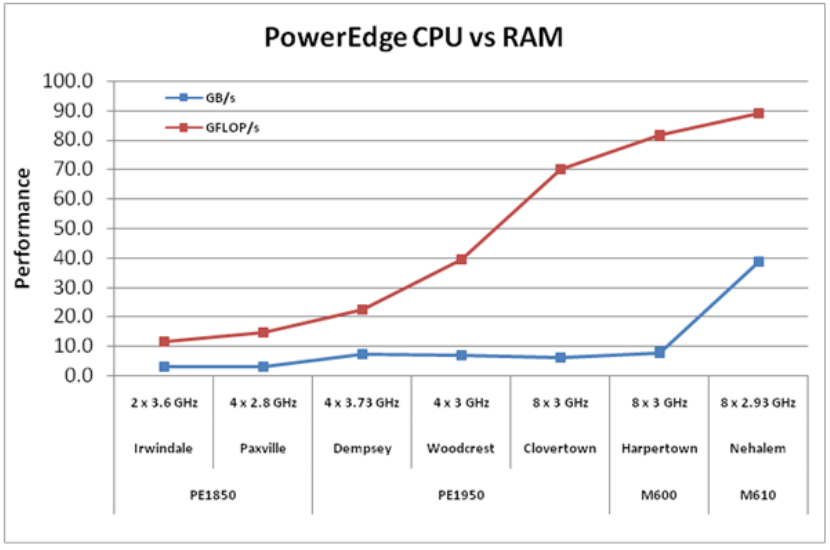
  8×2 FLOP por ciclo.

CPU modernas, memoria

Acceso a memoria principal

- Mejorando mucho el BW a memoria.
- Acceso multibanco.

  Ej: Nehalem 3, Sandy Bridge 4, Ivy Bridge 2 (por ahora).



Cache

- Estructura de 3 niveles: L1I y L1D, L2, L3.

  L3 compartida entre los cores de la pastilla.
- TLB multinivel.
- Unidad de prefetching de caché.
- Way prediction & semantic aware cache.

CPU vs GPU, números crudos

Performance pico teórica.

	Xeon E5-2680	M2090
FLOPS `fp32`	172.8 GFLOPS	1331 GFLOPS
FLOPS `fp64`	86.4 GFLOPS	665 GFLOPS
MemBW	51.2 GBps	177 GBps
TPD	130 W	225 W[1]
TC	2.26e9	3.0e9
Die	416 mm^2	520 mm^2
Availability	Q1'12	Q2'11

¿Cómo se hace semejante façanha?

[1] Placa entera, con memoria, ventiladores, etc.

CPU vs GPU, números crudos

Performance pico teórica.

	Xeon E5-2680	M2090
FLOPS `fp32`	172.8 GFLOPS	1331 GFLOPS
FLOPS `fp64`	86.4 GFLOPS	665 GFLOPS
MemBW	51.2 GBps	177 GBps
TPD	130 W	225 W[1]
TC	2.26e9	3.0e9
Die	416 mm^2	520 mm^2
Availability	Q1'12	Q2'11

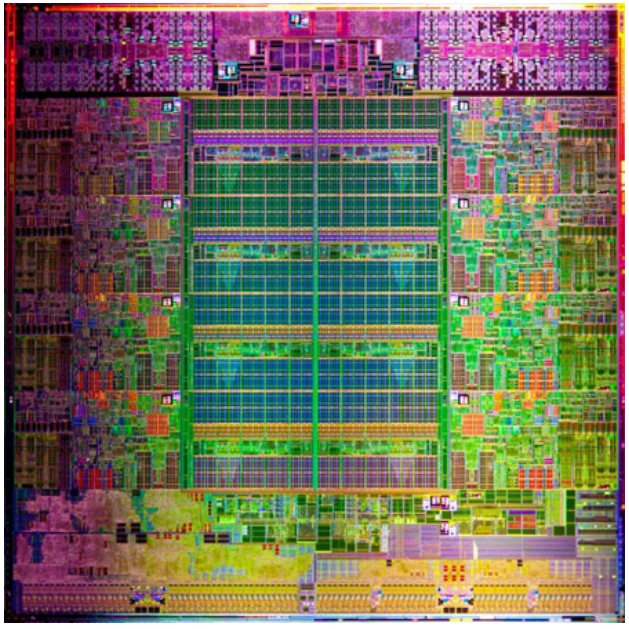
¿Cómo se hace semejante façanha?

Elegir otras soluciones de compromiso.

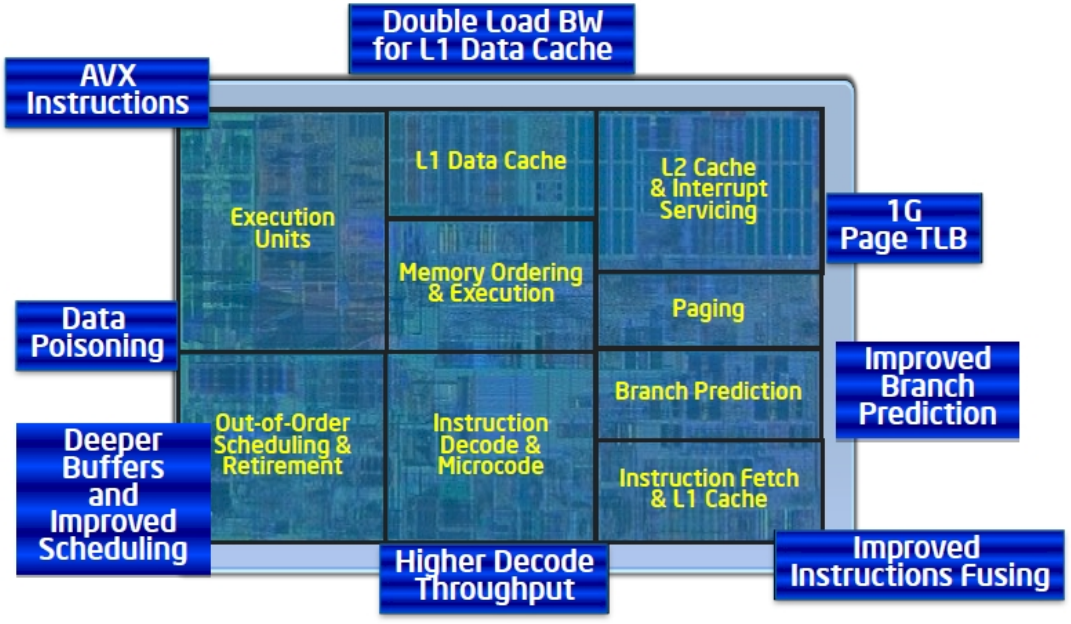
[1] Placa entera, con memoria, ventiladores, etc.

Xeon E5 floorplan

Die



Core



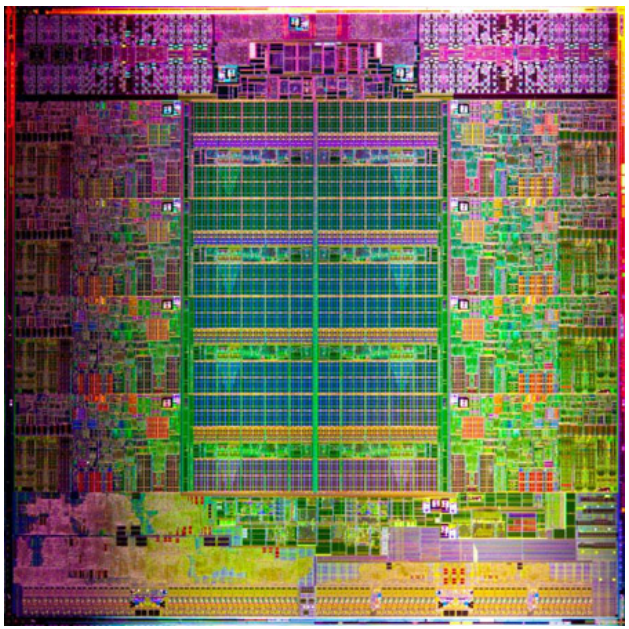
The Free Lunch Is Over

Demasiada superficie y potencia para:

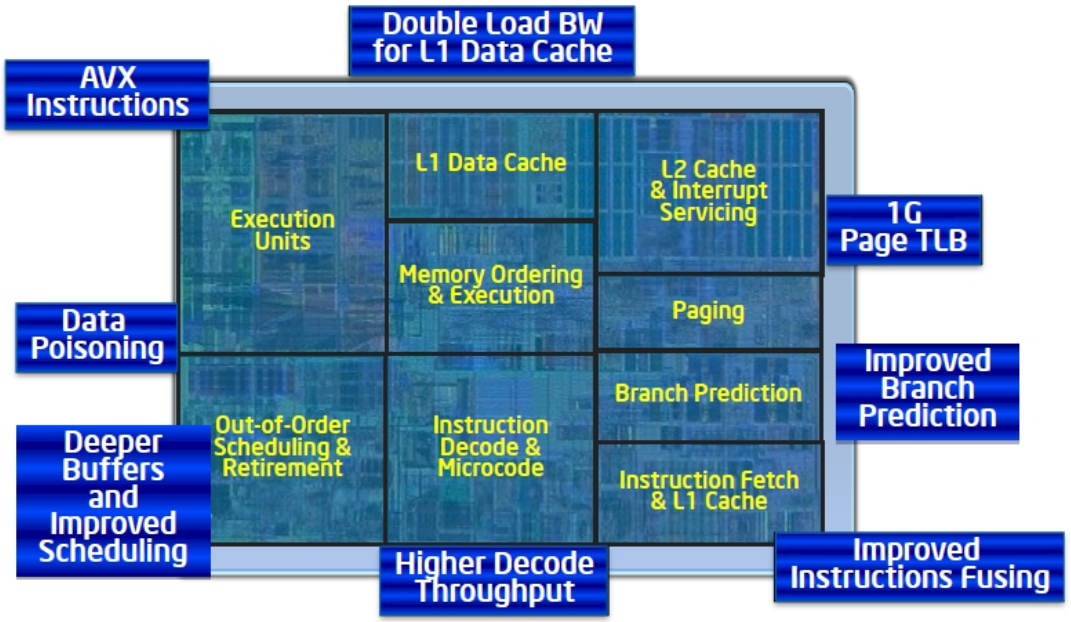
- Descubrir paralelismo – ILP.
- Mitigar el memory wall – Cache.

Xeon E5 floorplan

Die



Core



The Free Lunch Is Over

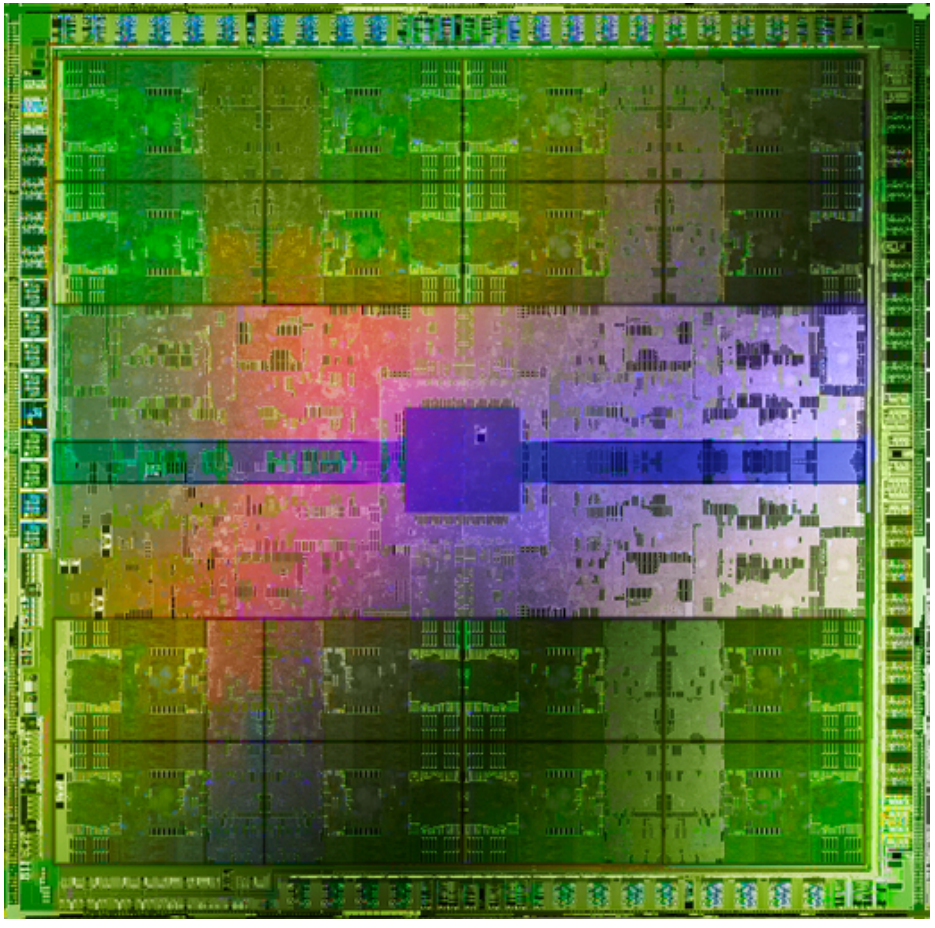
Demasiada superficie y potencia para:

- Descubrir paralelismo – ILP.
- Mitigar el memory wall – Cache.

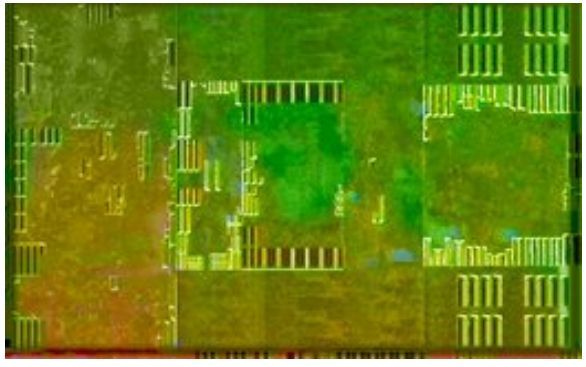
Hay otras soluciones de compromiso.

Fermi floorplan

Die

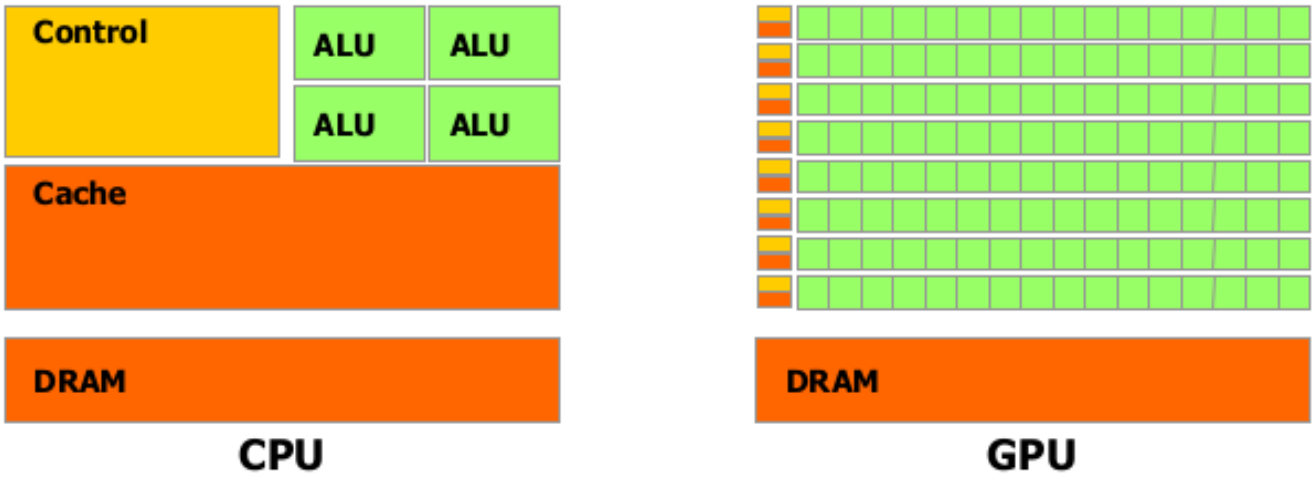


Core



CPU vs. GPU

Esquemáticamente la diferencia es:



Achicar area de control y caché, aumentar número de ALUs.
Pero . . .

1. ¿Cómo extraigo el paralelismo para alimentar a tantas ALUs?
2. ¿Cómo evito la latencia a memoria?

¿Cómo extraigo el paralelismo para alimentar a tantas ALUs?

Respuesta: paralelismo masivo.

En una GPU podemos tener, por ejemplo, 512 cores.

Para alimentar este paralelismo masivo
se necesita paralelismo de datos.

¿Cómo evito la latencia a memoria?

Respuesta: paralelismo masivo.

- Sobre-vender la capacidad física.
- Cualquier operación que demore (latencia):
  - Cambio de contexto a otro hilo que pueda progresar.
- Suficiente paralelismo puede ocultar la latencia.
- No solo latencia de memoria, también de operaciones.

Notar que la latencia de operaciones y memoria sigue presente.
Solo se superpone con otros cálculos.

¿Cómo evito la latencia a memoria?

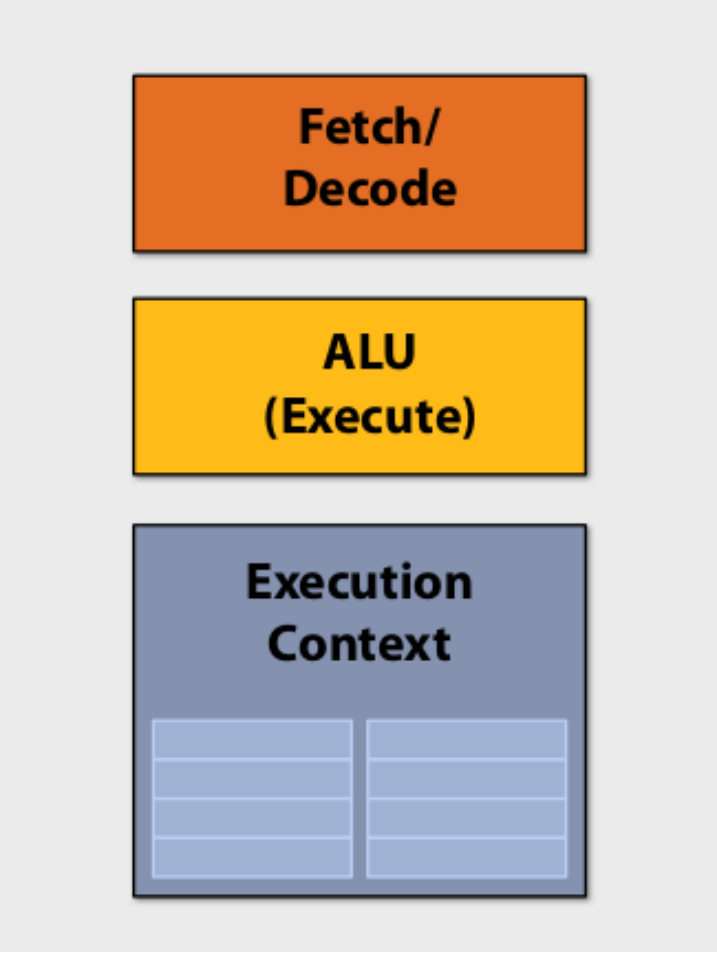
Respuesta: paralelismo masivo.

- Sobre-vender la capacidad física.
- Cualquier operación que demore (latencia):
  - Cambio de contexto a otro hilo que pueda progresar.
- Suficiente paralelismo puede ocultar la latencia.
- No solo latencia de memoria, también de operaciones.

Notar que la latencia de operaciones y memoria sigue presente.
Solo se superpone con otros cálculos.

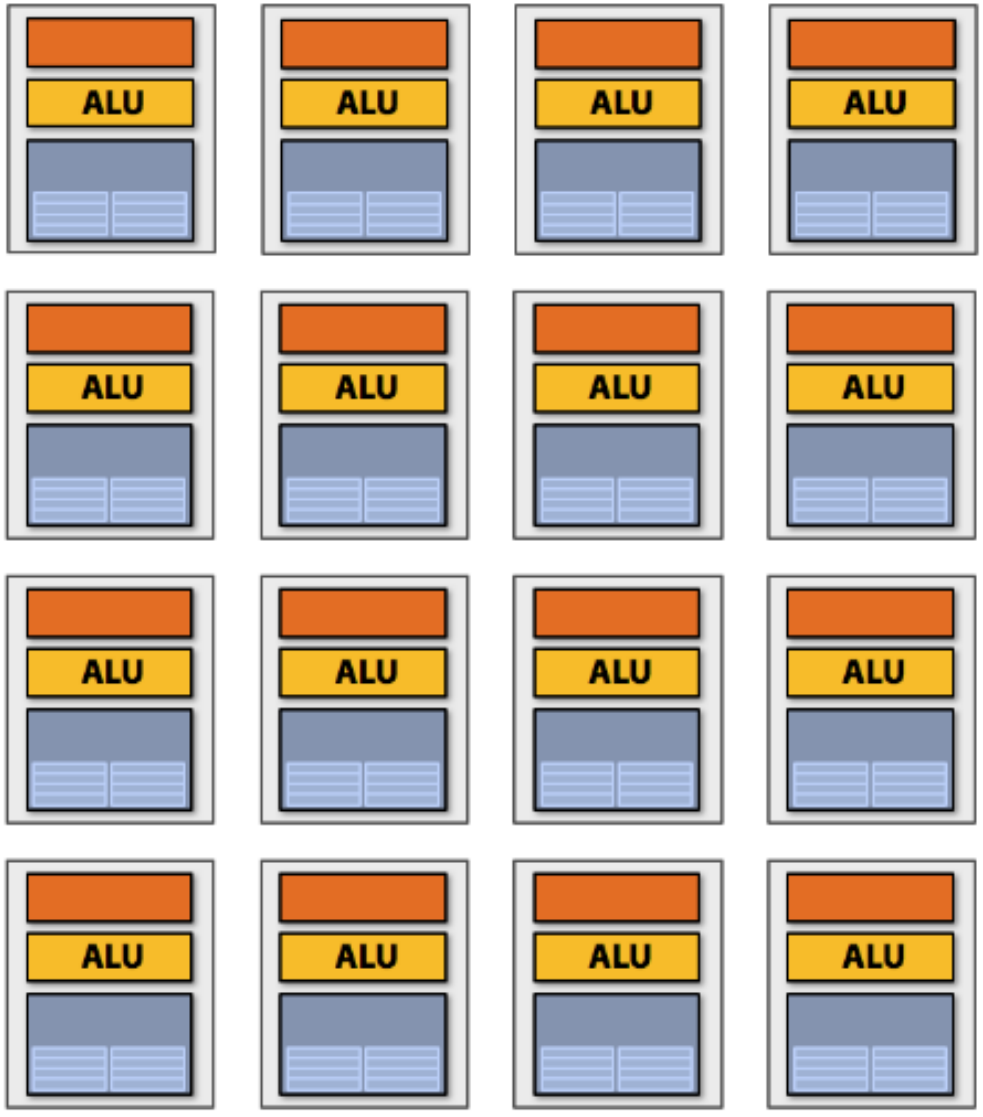
Highly parallel & throughput computing

Cores muy simples



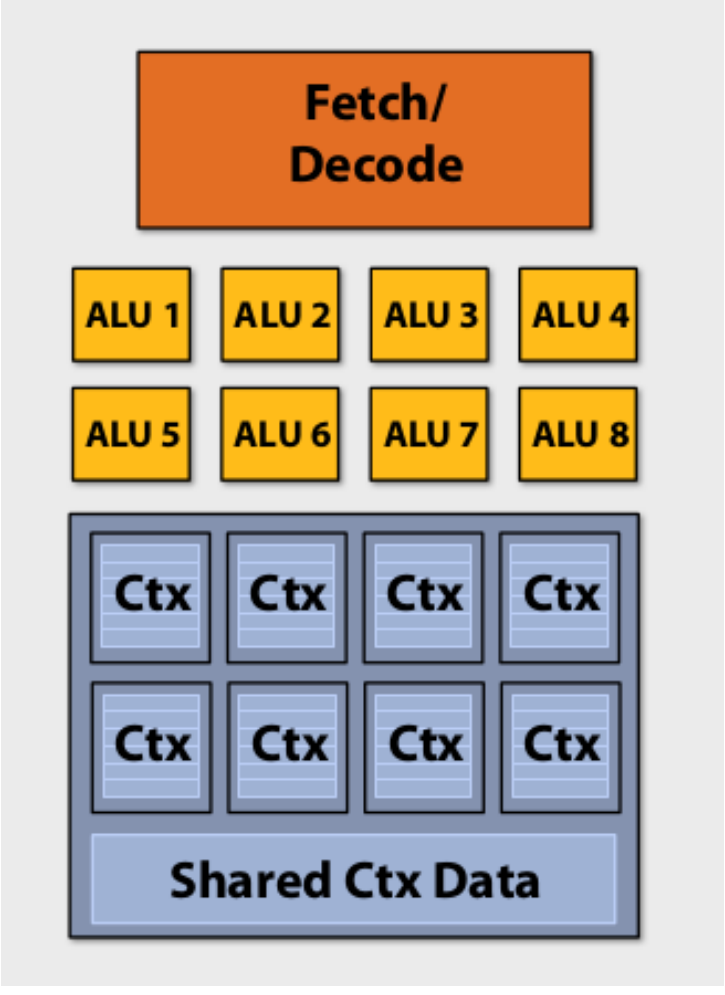
Execution context es donde se guarda el estado: PC, registros.

Muchos cores muy simples



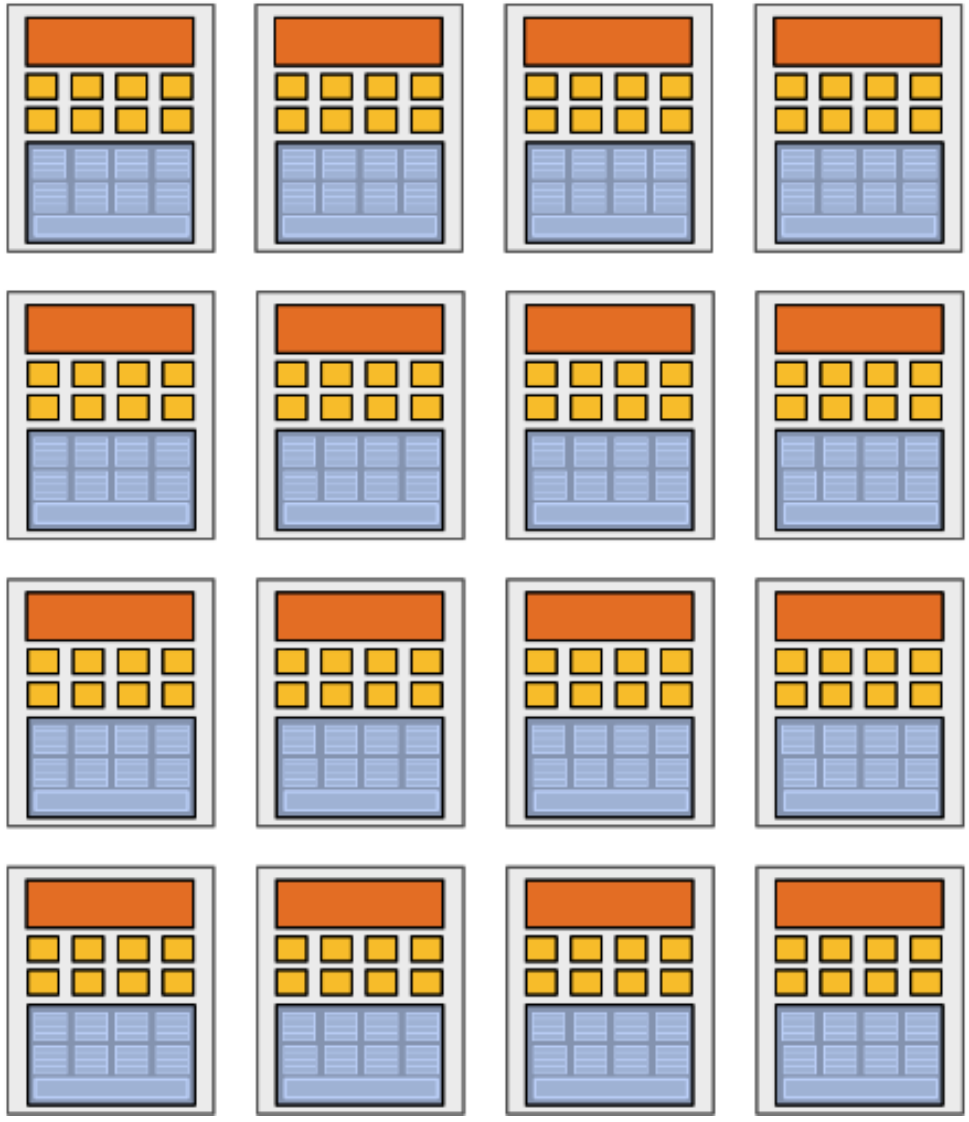
Cores vectoriales simples

Amortizar aún más el área: vector processor.



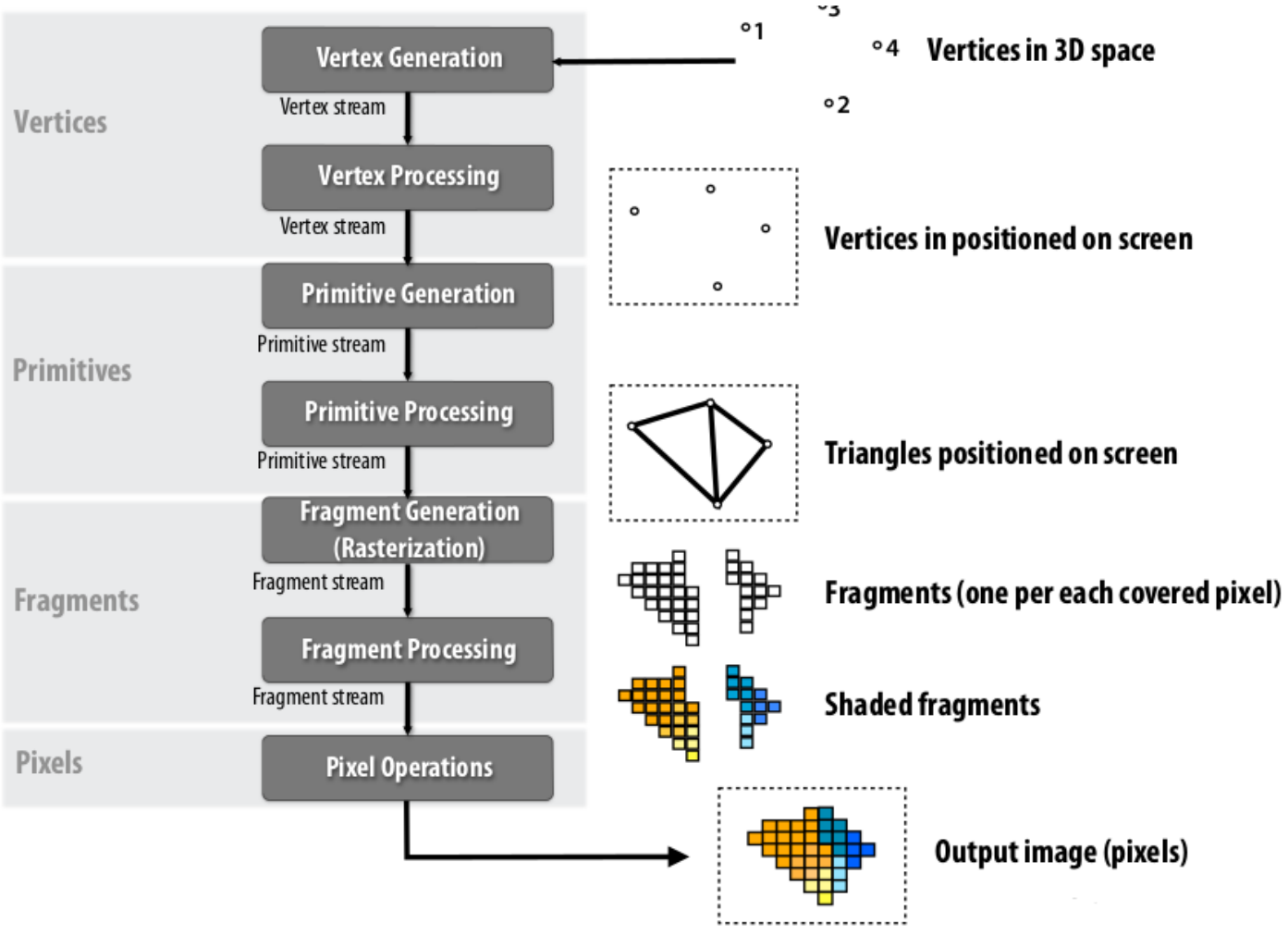
Muchos cores vectoriales simples

Foto final.



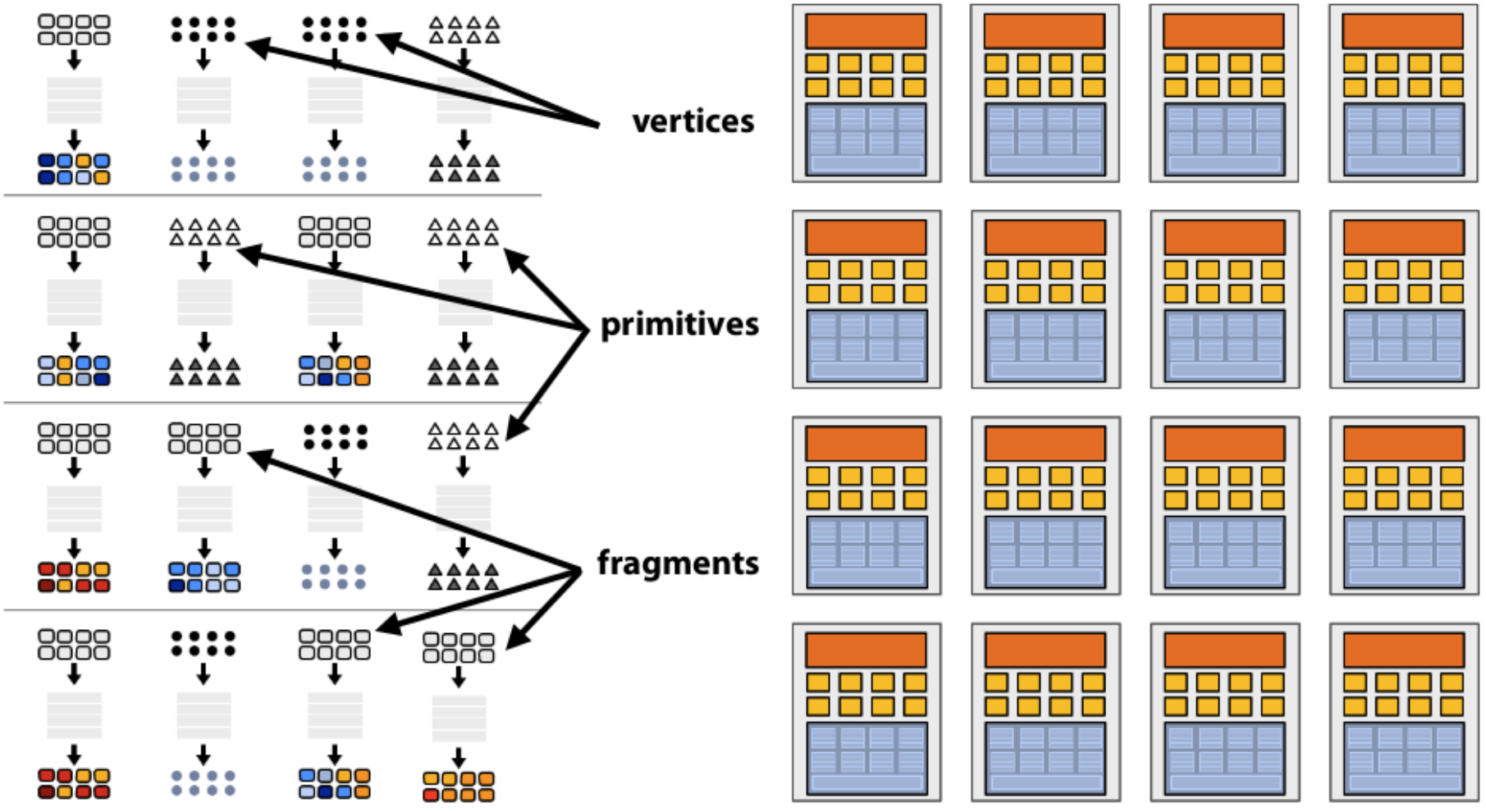
Recordar el origen: *realtime 3D rendering*

Graphic pipeline.



Como se procesan los vertex y fragments

Graphic pipeline: highly parallel load.



Procesadores vectoriales, la otra solución

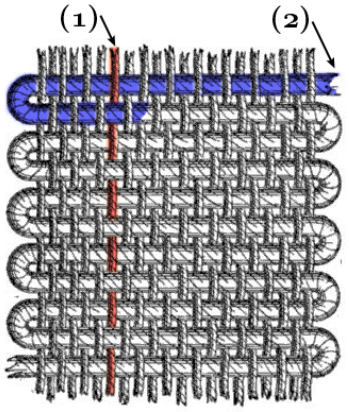
Vectorizar es complejo.

Programar SSE/AltiVec/NEON/¿XeonPhi? es un problema.

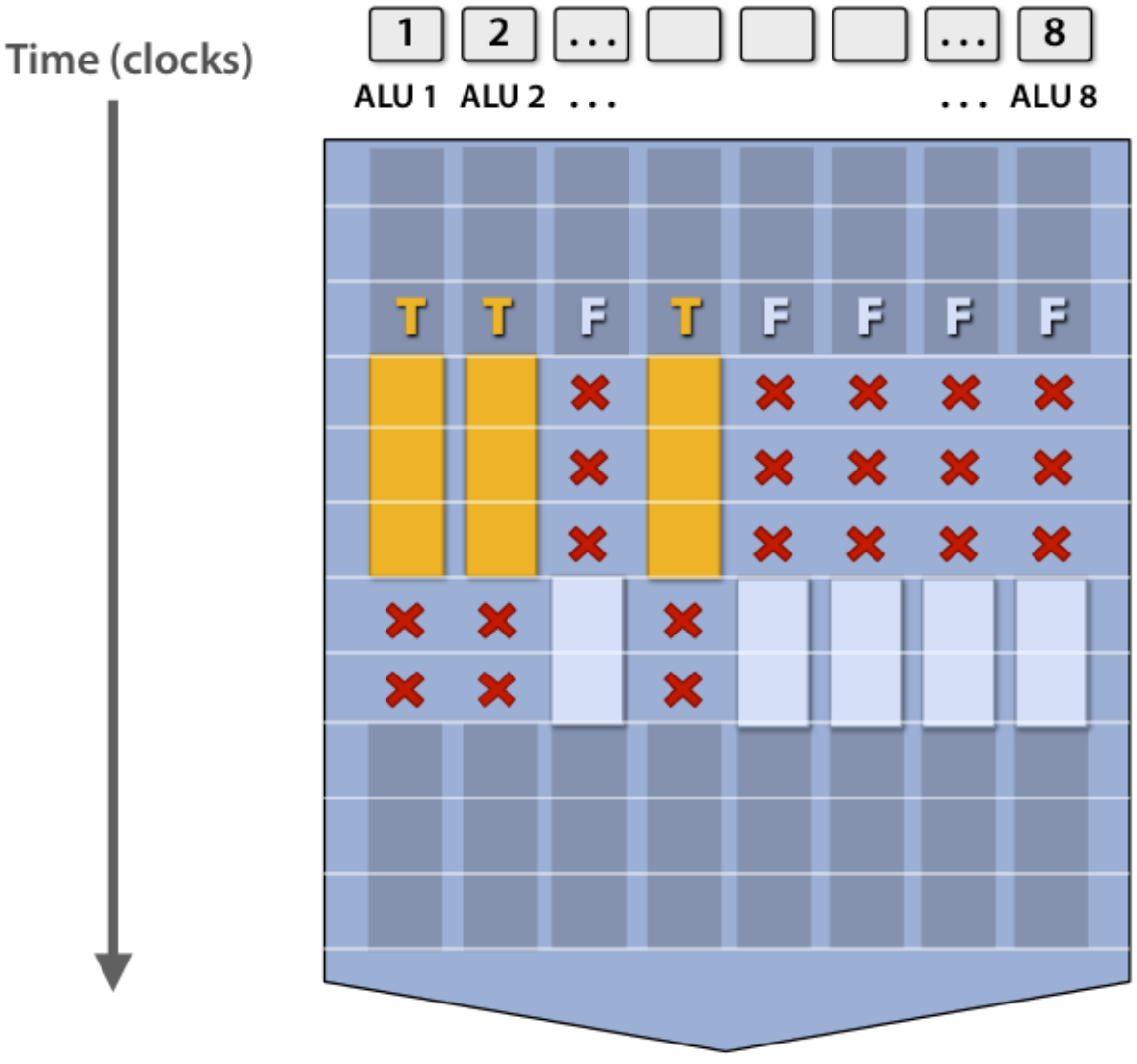
SIMD en hardware, no implica SIMD en software

- Simular flujos de control independientes.
- ¿Cómo? *internal masks*, *branch sincronization stack*, *instruction markers*.

Ilusión que cada lane en un warp tiene su propio `PC`.



Divergencia en el flujo de control

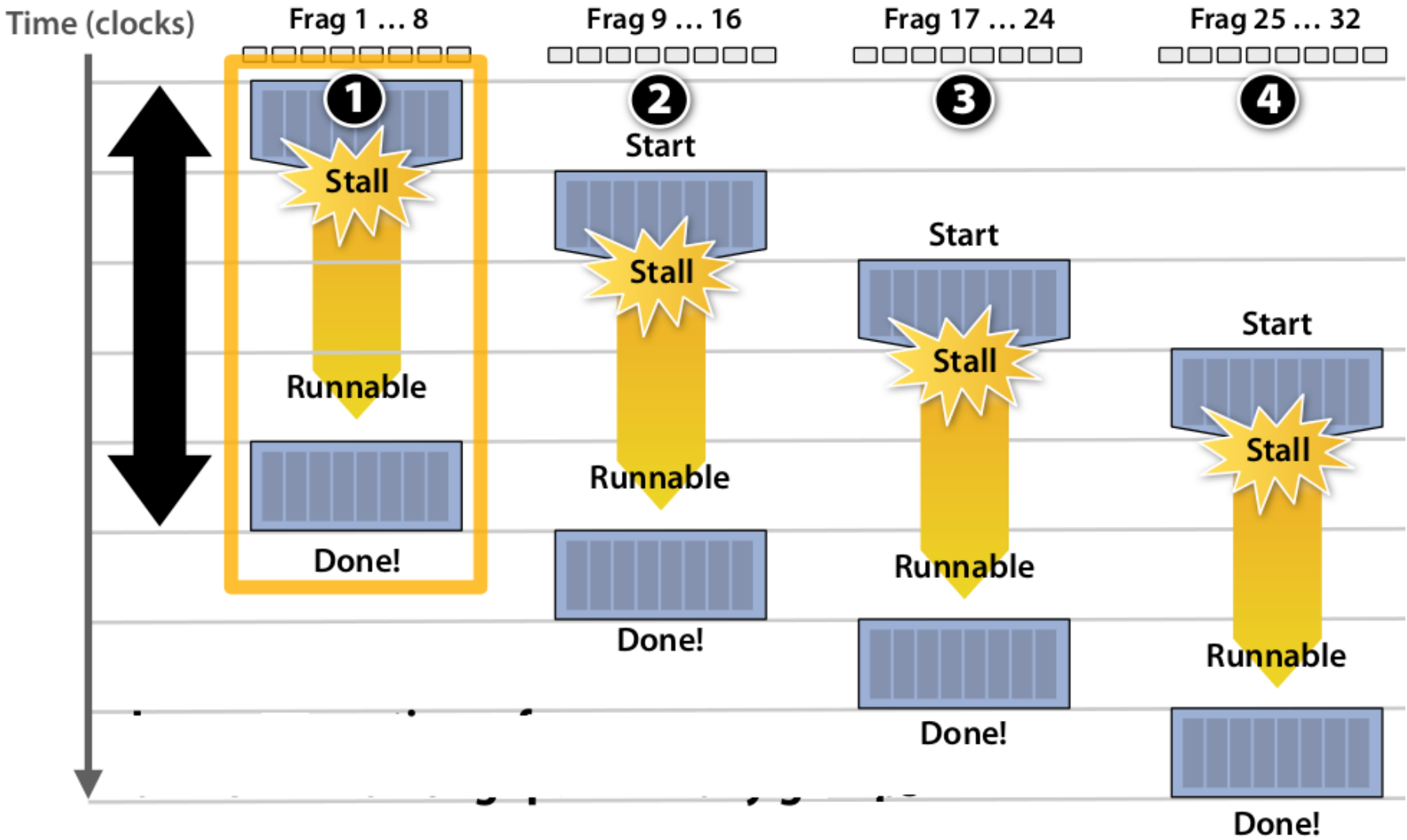



```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Ocultamiento de latencia



La latencia total es la misma, pero se superpone con otros cálculos.

Muchos hilos volando, muchas cosas para recordar

Más ocultamiento latencia mem& ops

$\Rightarrow$ Más hilos en ejecución

$\Rightarrow$ Más memoria para sus contextos

Tensión entre:

- Ocultamiento de latencia.
- Tamaño de la memoria para almacenar contextos.

Muchos hilos volando, muchas cosas para recordar

Más ocultamiento latencia mem& ops
$\Rightarrow$ Más hilos en ejecución
$\Rightarrow$ Más memoria para sus contextos

Tensión entre:

- Ocultamiento de latencia.
- Tamaño de la memoria para almacenar contextos.

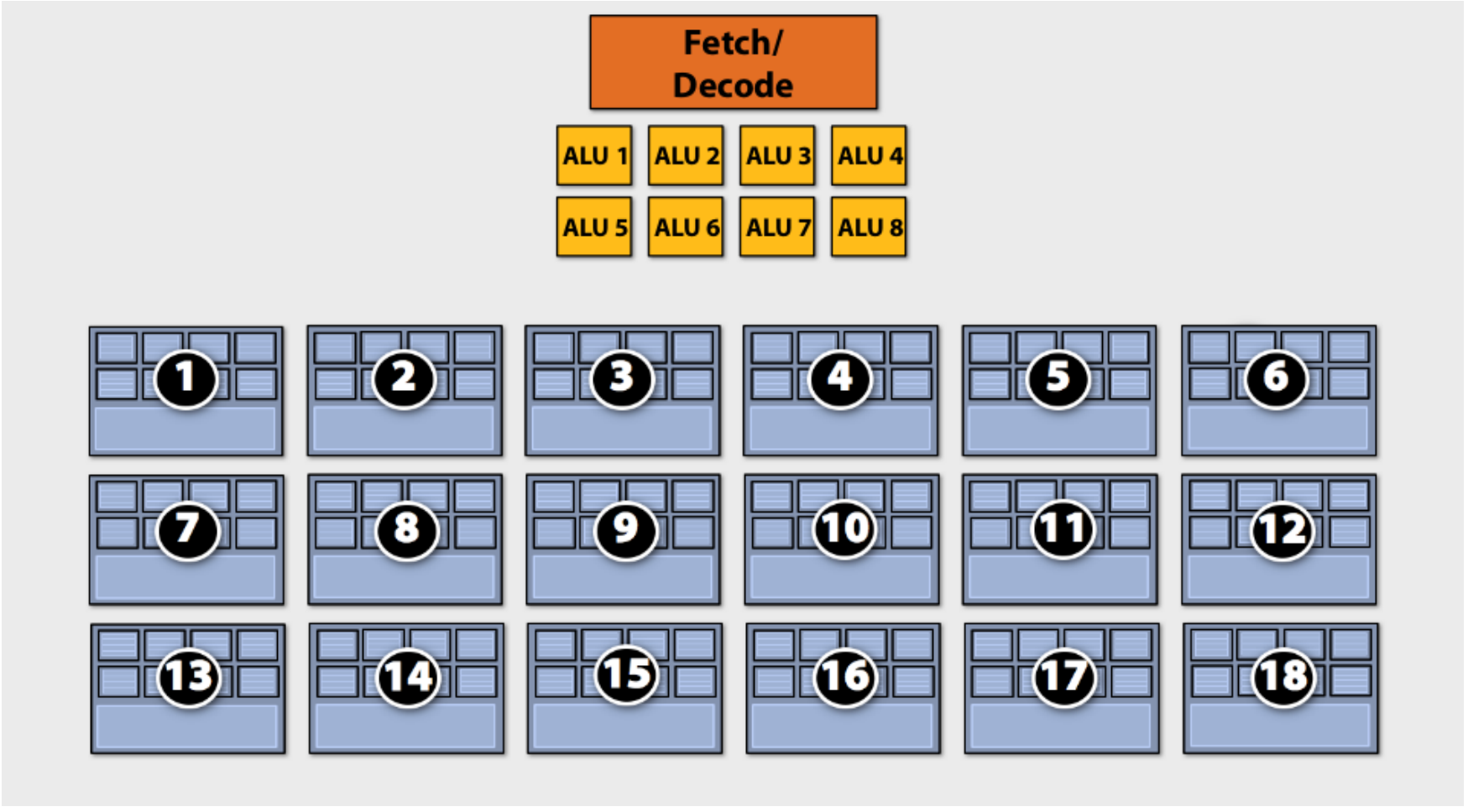
Verdad de Pedro Grullo

La memoria es finita y de tamaño constante.

Corolario

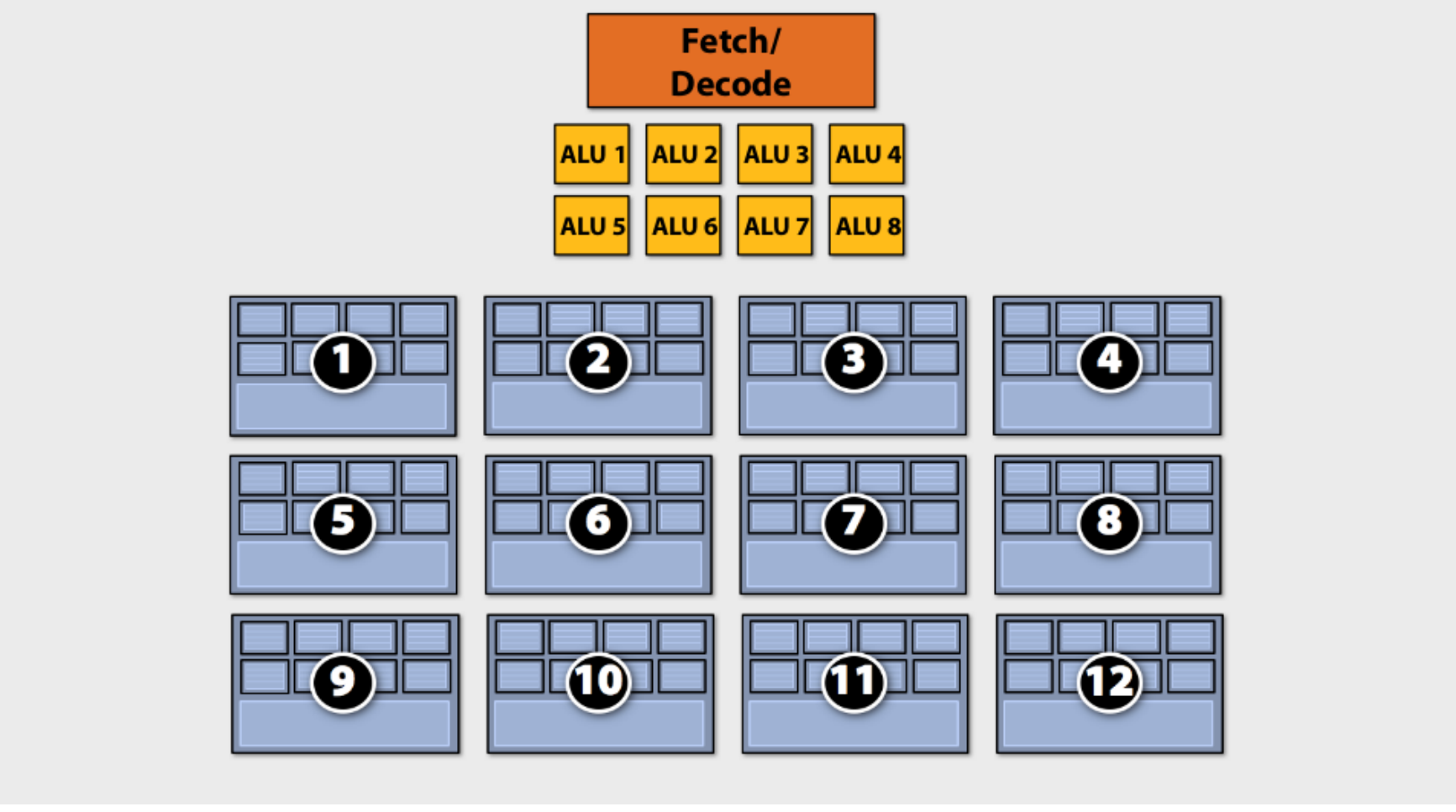
Kernels con contextos pequeños ocultan mejor la latencia que kernels con contextos grandes.

Contextos pequeños. Máximo ocultamiento de latencia

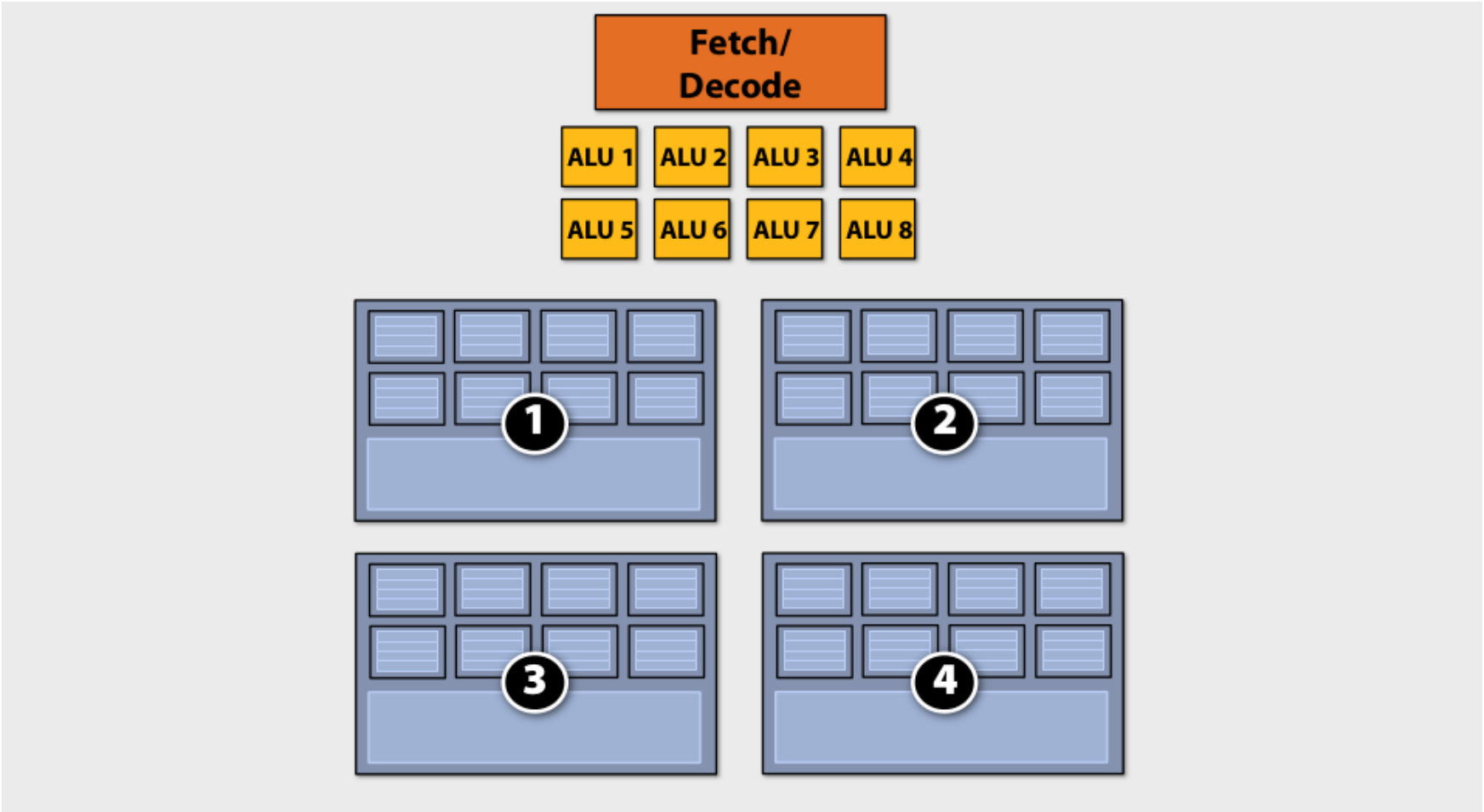


Cada kernel utiliza unos pocos registros.

Contextos intermedios. Mediano ocultamiento de latencia



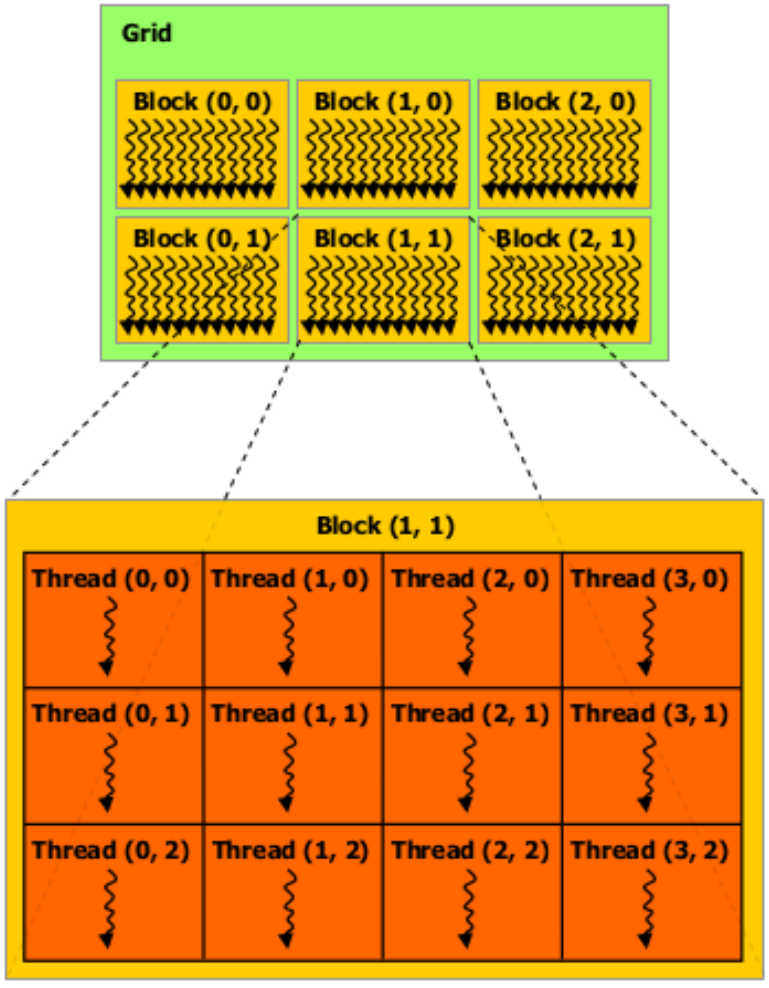
Contextos grandes. Poco ocultamiento de latencia

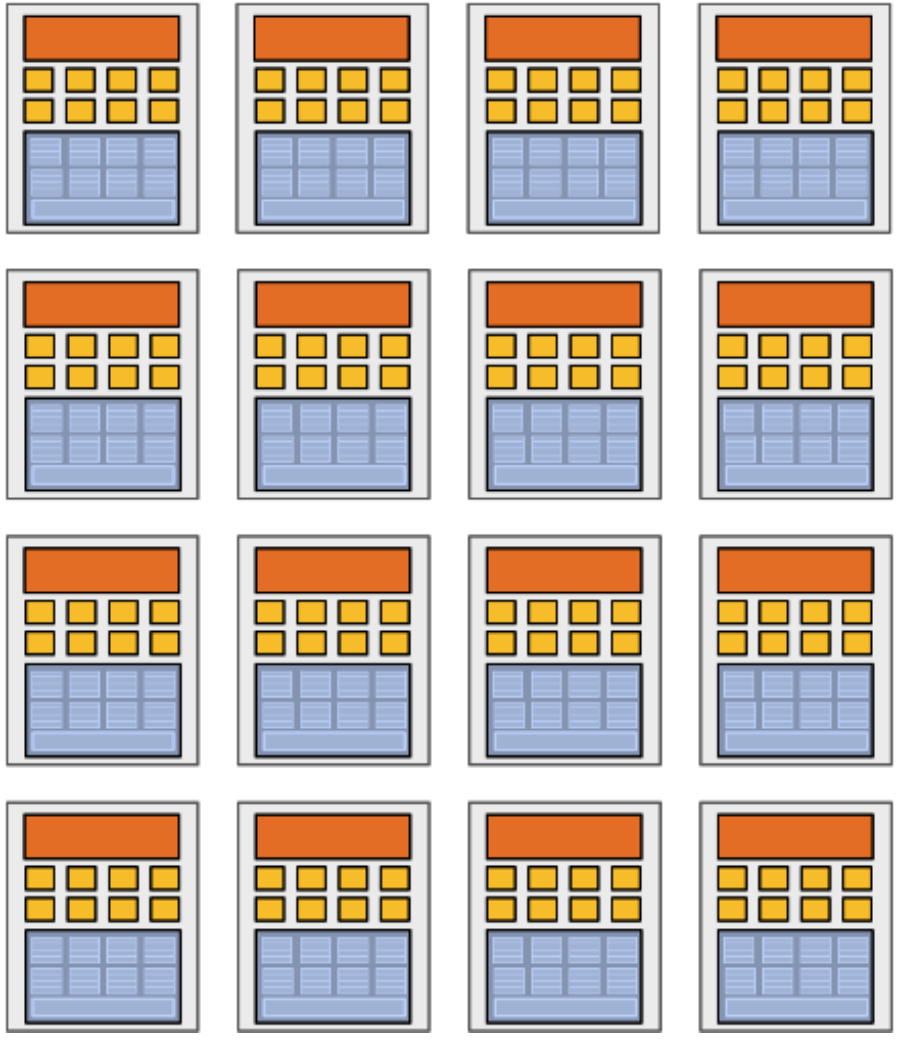


La cantidad de registros es un parámetro crucial en la performance.

Problema de planificación

Como ejecutar grilla de bloques de hilos en cores vectoriales.





Planificación de Warp y Bloque

Warp

- Mínima unidad de planificación.
- Procesamiento SIMD.
- (pero) Flujo de control independiente en cada hilo.
- Típicamente 16 hilos, 32 hilos (warp) o 64 hilos (wavefront).

Bloque

- Es un multiprocesador virtual

  Hilos independientes, memoria compartida.
- Puede haber comunicación y sincronización intra-bloque.
- Los bloques son independientes entre si.
- Un bloque comienza y no para hasta completar su programa.
  Non-preemptive scheduler.

  Un bloque se ejecuta en solo un procesador vectorial.
- Un procesador vectorial puede ejecutar más de un bloque.

Planificación de dos niveles

Warps dentro de un bloque (*Warp Scheduler*)

- Todos los warps de bloques en un vector core van rotando.
- *Ctxt switch* cuando hay latencia de mem&ops.
- Hecho en hardware: free ctxt switch.
- Hecho en hardware: límites duros en #warps y #bloques.

Bloques dentro de una grilla (*GigaThread^{TM} Engine*)

- Planificador batch (cola de bloques esperando procesamiento).
- Asigna una cantidad de bloques a cada vector core.
  Limitaciones:
  - #warps y #bloques que maneja el warp scheduler.
  - #registros: tiene que entrar en el contexto del vector core.
  - Todos estos parámetros se fijan en tiempo de compilación.
- Solo parametriza el código con `blockIdx` y `threadIdx`!

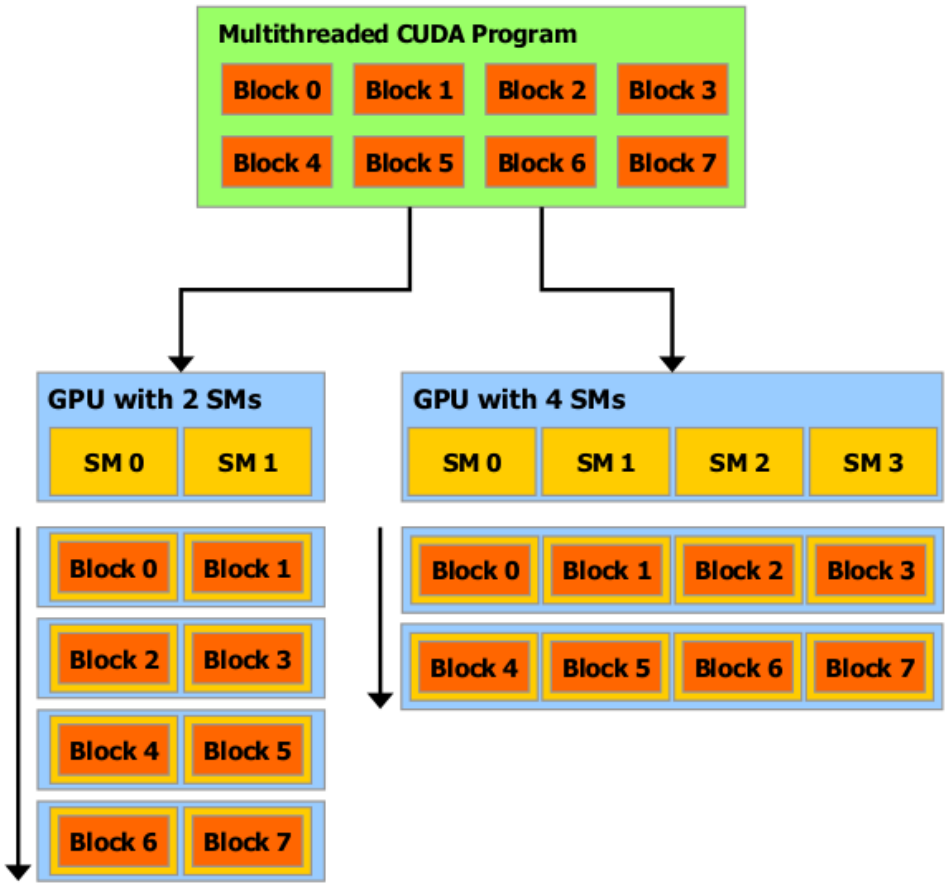
Repensar algunas características de CUDA

- Tiempo de vida de cada nivel de la jerarquía de memoria.
- Falta de sincronización global entre bloques.
- Throughput computing!

Clave: entender como funciona el planificador de dos niveles.

Automatic Scalability

Dada una aplicación dividida en muchos bloques independientes, es fácil (para el *vendor*) ofrecer escalablidad!



El fabricante vende #vector cores según el bolsillo del cliente.

Occupancy

$$occupancy = \frac{\#threads\_per\_vector\_core}{\#max\_threads\_per\_vector\_core}$$

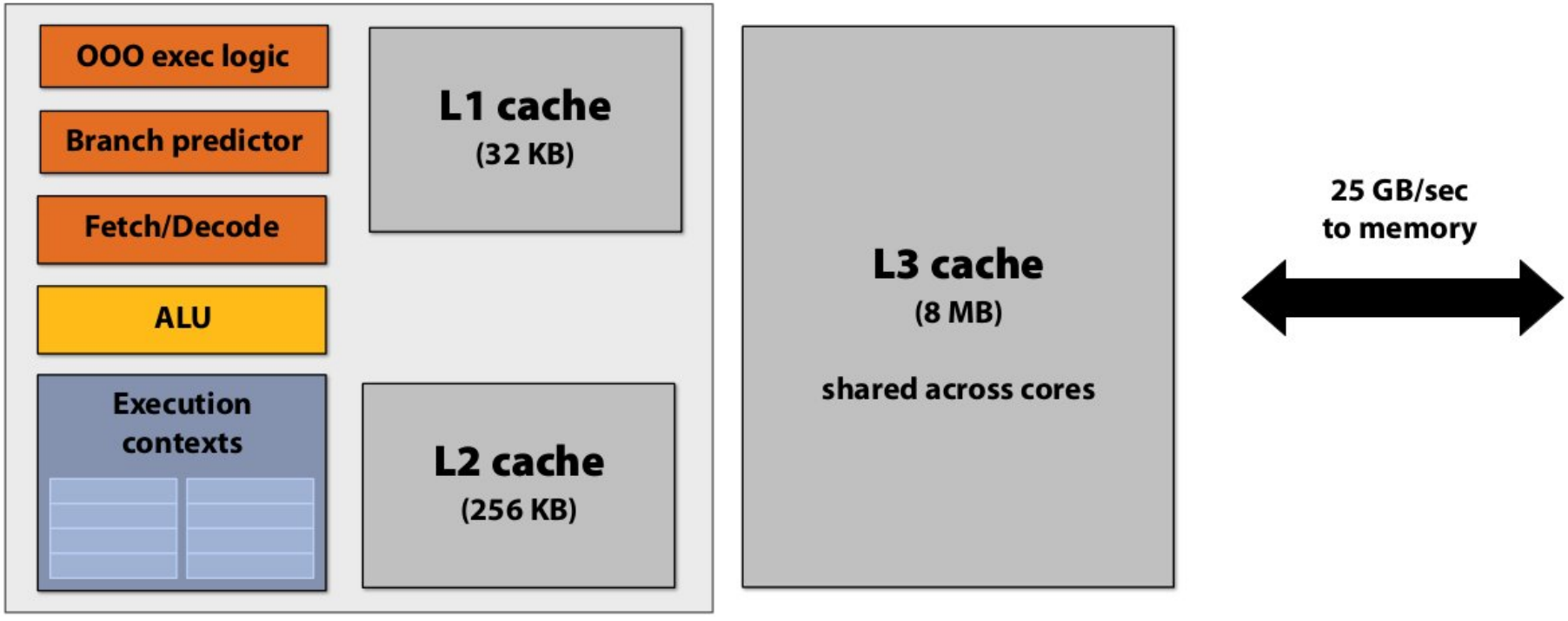
- Solo una primerísima aproximación de la eficiencia del código.
- Definido en tiempo de compilación:
  - registros.
  - tamaño de la grilla y el bloque.
- Se informa a través de `CUDA_PROFILE=1`.

Ejemplo

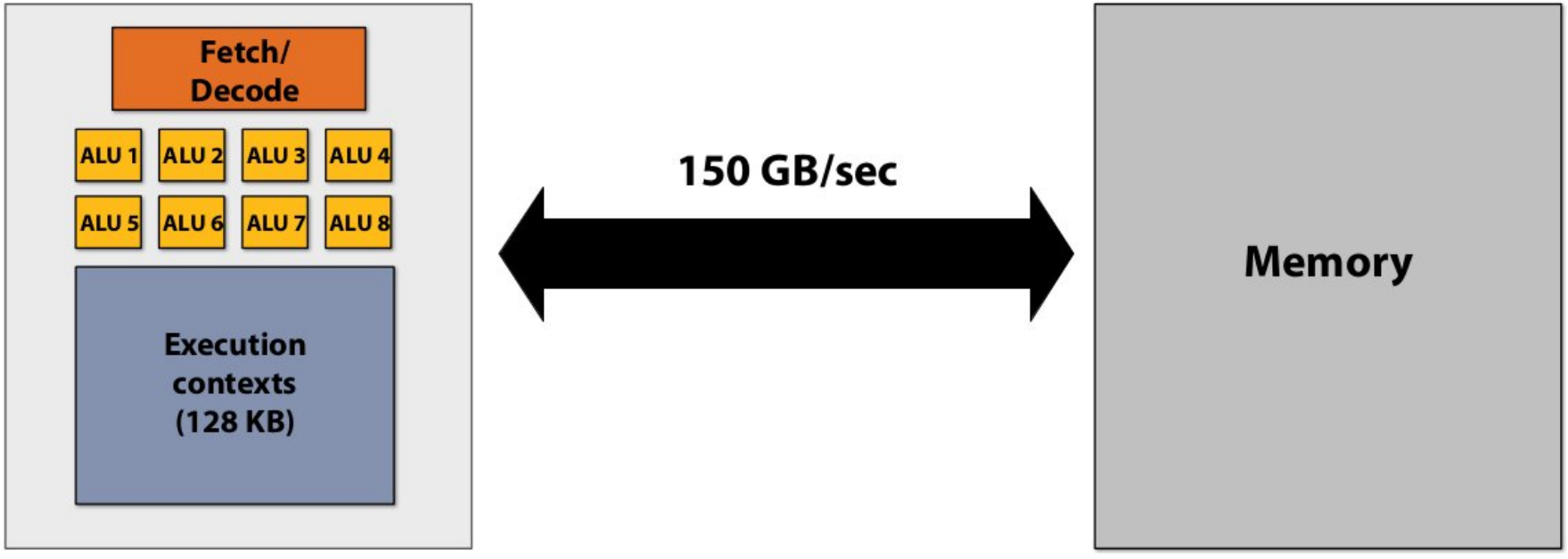
Bloques del máximo de hilos, el kernel ocupa pocos registros, la máxima cantidad de hilos que maneja el vector core es 48×32.

$$occupancy = \frac{1024}{1536} = 0.66$$

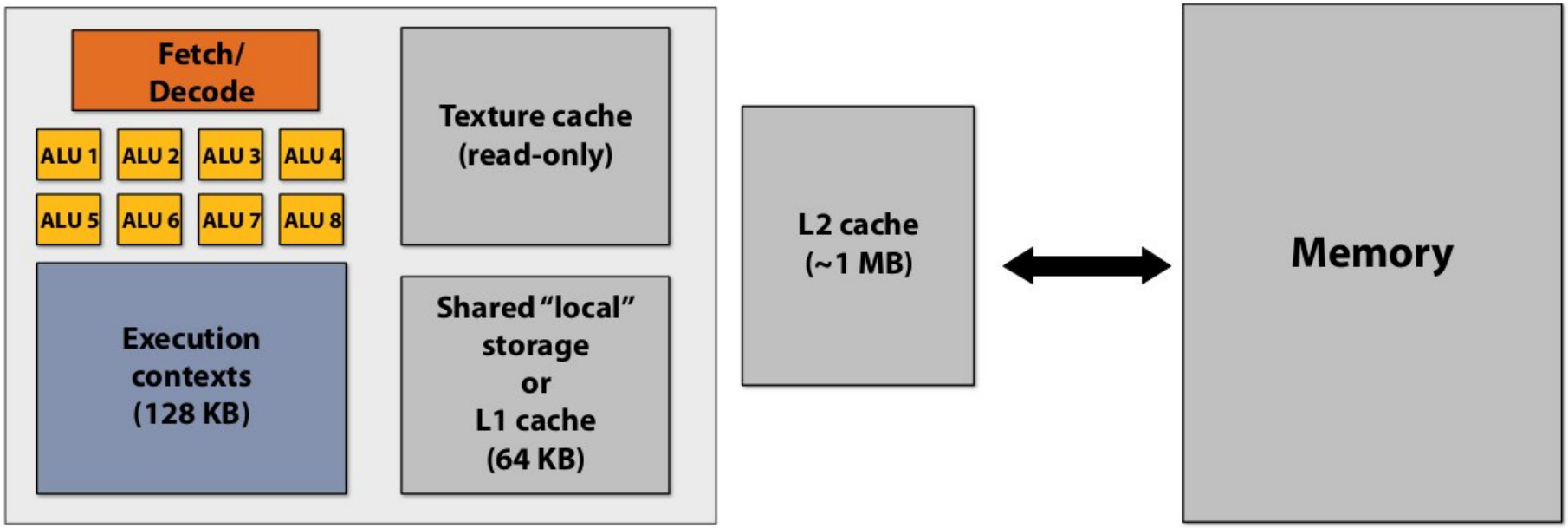
Organización de la memoria de una CPU moderna



Organización de la memoria de una GPU



Organización de la memoria de una GPU moderna



Ancho de banda, latencias y tamaños totales

Ancho de banda y latencias

Nivel	BW	Latencia
Registros	8.1 TB/s	1 ciclo
Shared	1.3 TB/s	¿6? ciclos
Global	177 GB/s	400 ciclos

- Vasily Volkov, *Better Performance at Lower Occupancy*, GTC 2010.

Pirámide invertida de memoria

Nivel	Tamaño total
Registros	2 MB
Caché L1 / Shared	0.25 MB / 0.75 MB
Caché L2	0.75 MB

Más limitaciones al paralelismo: registros

Registros

- Tienen que entrar en el contexto.

$$\#regs\_per\_kernel \times \#threads\_per\_vector\_core \leq context\_size$$

- Hay límites a la cantidad de registros por thread.
- Si un kernel pasa este límite: register spilling.
- O por opción del compilador: `-maxrregcount`.
- O por el calificador `__launch_bounds__()` de los kernels.

Register spilling

- Registros guardados en local memory.
- Es básicamente memoria global.
- Evitarlo!

Más limitaciones al paralelismo: memoria compartida

- Tienen que entrar en la shared del vector core.

$$shared\_per\_block \times \#blocks\_per\_vector\_core \leq shared\_size$$

- No hay nada parecido a *shared spilling*.
- La memoria compartida se puede configurar.
- También se puede pedir memoria shared dinámicamente.

```
kernel<<<grid_size, block_size, shared_size>>>() .
```

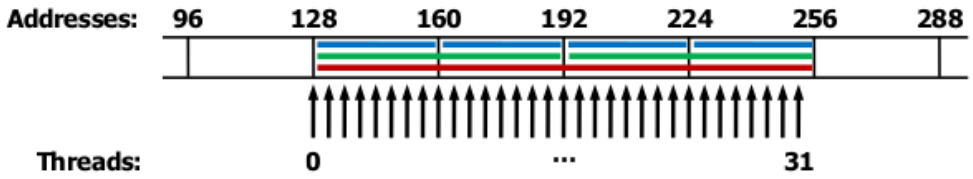
Warps y accesos a memoria global

- El sistema de memoria junta los pedidos de memoria del warp.
- Trae tantas líneas de caché L1 (128 bytes) como sea necesario.

Warps y accesos a memoria global

- El sistema de memoria junta los pedidos de memoria del warp.
- Trae tantas líneas de caché L1 (128 bytes) como sea necesario.

El caso ideal

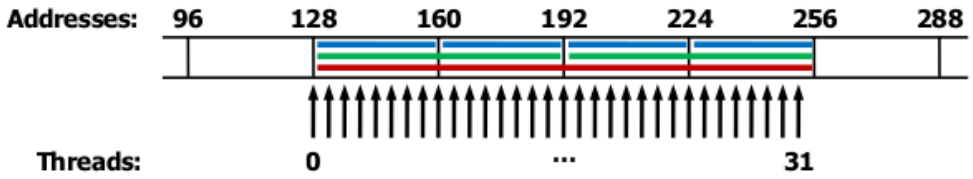


Ejemplo: el típico `a[tid] = 1.0f`, donde `tid` es el identificador de hilo lineal. Son 32 hilos, accediendo cada uno a 4 bytes (1 `float`): 128 bytes. La memoria de `cudaMalloc` está alineada a 256 bytes.

Warps y accesos a memoria global

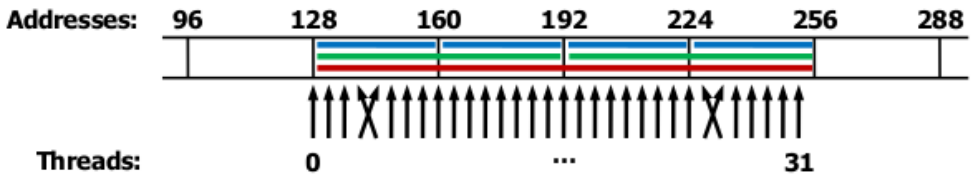
- El sistema de memoria junta los pedidos de memoria del warp.
- Trae tantas líneas de caché L1 (128 bytes) como sea necesario.

El caso ideal

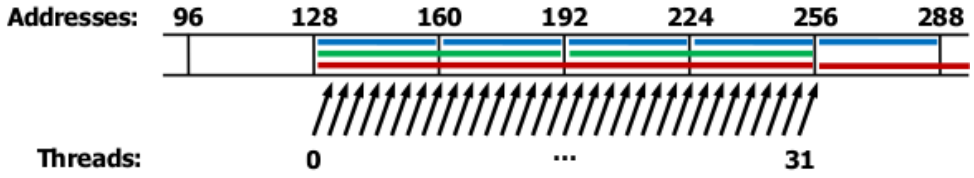


Ejemplo: el típico `a[tid] = 1.0f`, donde `tid` es el identificador de hilo lineal. Son 32 hilos, accediendo cada uno a 4 bytes (1 `float`): 128 bytes. La memoria de `cudaMalloc` está alineada a 256 bytes.

No cambia nada si se pierde secuencialidad



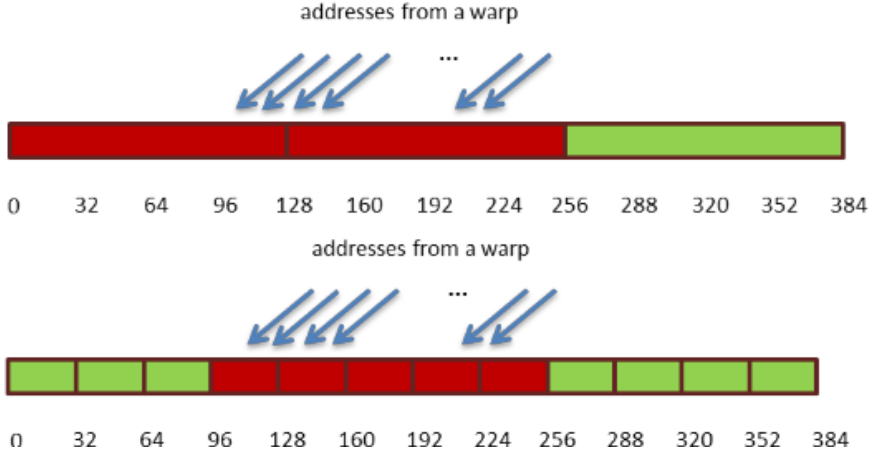
Accesos desalineados pero secuenciales



- Se realizan dos pedidos de 128 bytes.
- Se desperdicia parte del ancho de banda.

Deshabilitar la L1 puede ayudar

La cache L2 tiene líneas de 32 bytes. Comparar:



Warps y accesos a memoria compartida

Organización

- Dividia en 32 bancos *interleaved* en palabras de 32 bits.
- Cada thread en un warp puede leer en paralelo de un banco distinto.
- Si más de un hilo lee la misma palabra de 32 bits de un banco, el resultado se difunde.

Acceso ideal

```
__shared__ float shared[SHARED];
float data = shared[threadIdx.x];
```

Conflicto de bancos

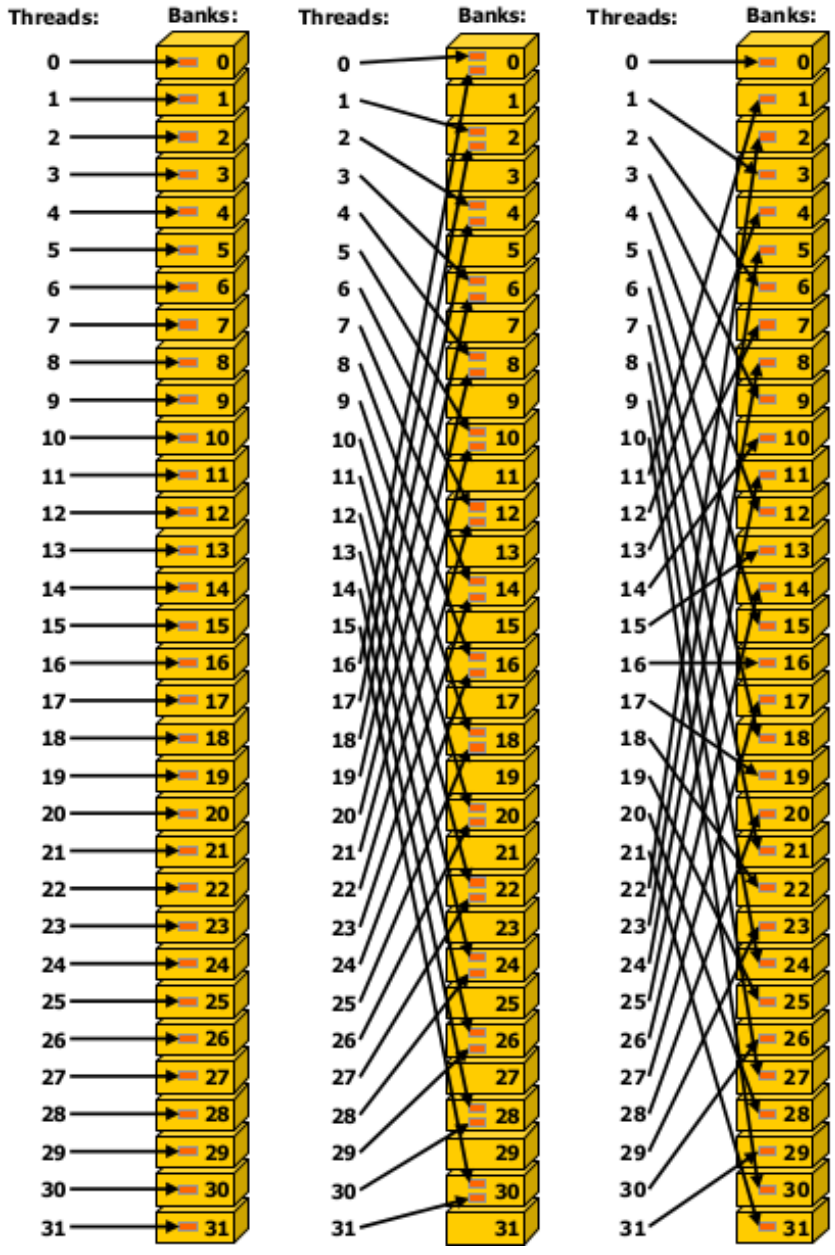
- Dos hilos dentro de un warp acceden a distintas palabras de 32 bits en el mismo banco.
- Los hilos en conflicto se serializan.

Conflicto de bancos en la memoria compartida – 1

Izq. Todo bonito, todo ordenadito.

Cen. Stride de 2, 2 palabras distintas de 32 bits en el mismo banco. 2-way-conflict.

Der. Stride de 3. *coprime*$(32, 3)$. No hay conflicto.

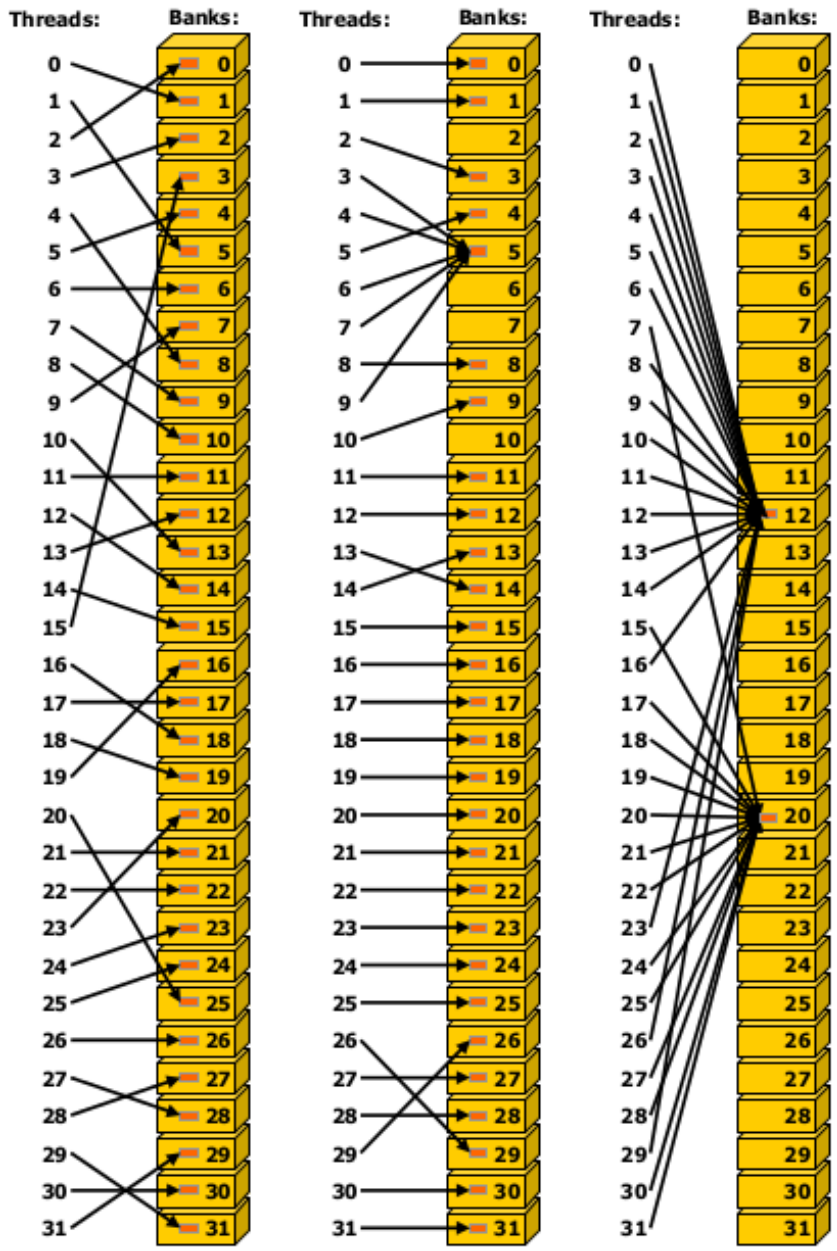


Conflicto de bancos en la memoria compartida – 2

Izq. Permutación. Sin conflictos.

Cen. Threads 3, 4, 6, 7, 9 acceden a la misma palabra del mismo banco. No hay conflictos. Se difunde el valor.

Der. Idem al anterior, pero más masivo.



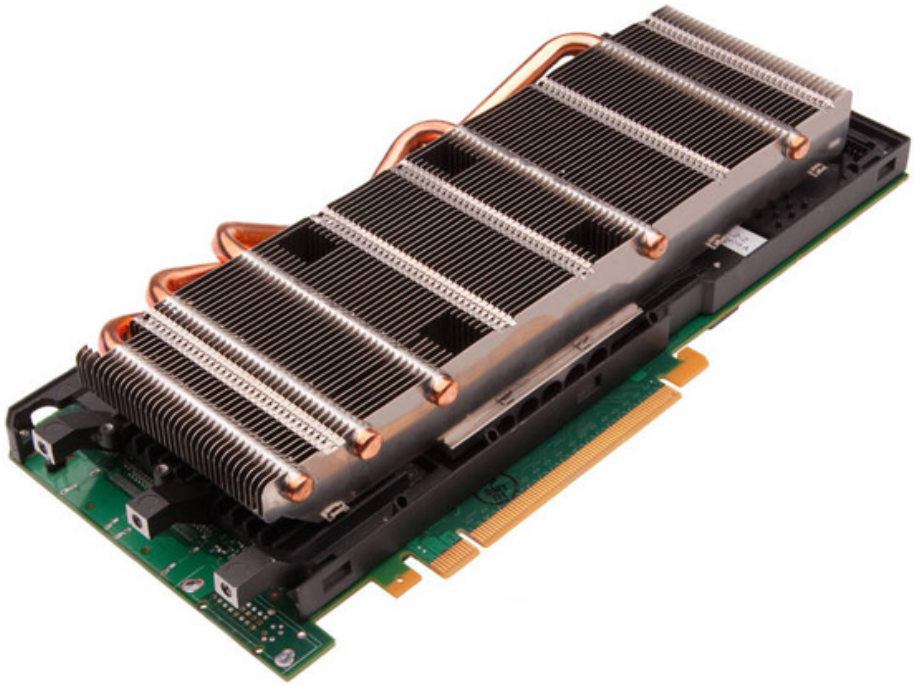
¿Para qué sirve la cache L1 y L2

No es una cache como en la CPU.
No implementa políticas de reemplazo para *temporal locality*.

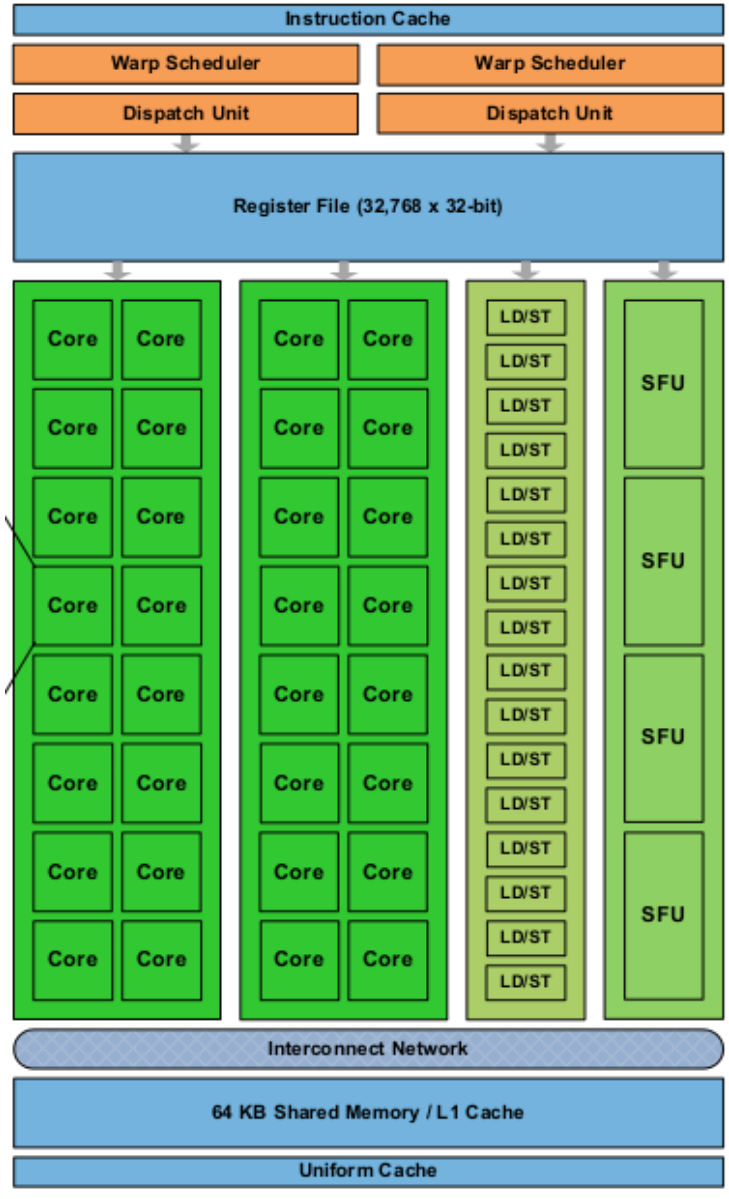
Es útil para:

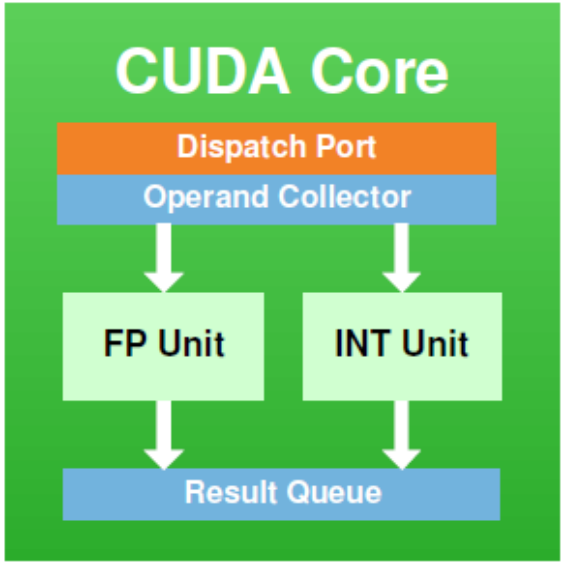
- Patrones de acceso a memoria ligeramente desalineados.
- Register spilling, stack frames, function call.
- En general suaviza las rugosidades de la jerarquía de memoria.

La placa



Fermi Streaming Multiprocessor y Core





Características, SM (vector cores)

16 Streaming Multiprocessors. Cada uno con:

- 32 cores `int32`.
- 32 cores `fp32` ó 16 cores `fp64`.
- Relación `fp64`/`fp32` $= 1/2$.
- 4 SFU (special function units): `sqrt`, `cos`, etc.
- 16 LD/ST units, para calcular direccionamientos.
- 2 warp schedulers.
- Warp scheduler: $\leq$ 48 warps, $\leq$ 8 bloques.

  Hasta $16 \times 48 \times 32 = 24576$ hilos concurrentes.
- 32768 registros!, hasta 64 por thread, granularidad de 2.

Performance Pico

`fp32` $16SM \times 32cores \times 2FLOP/clock \times 1.15GHz = 1177.6GFLOPS$

`fp64` $16SM \times 16cores \times 2FLOP/clock \times 1.15GHz = 588.8GFLOPS$

Características, SM (vector cores)

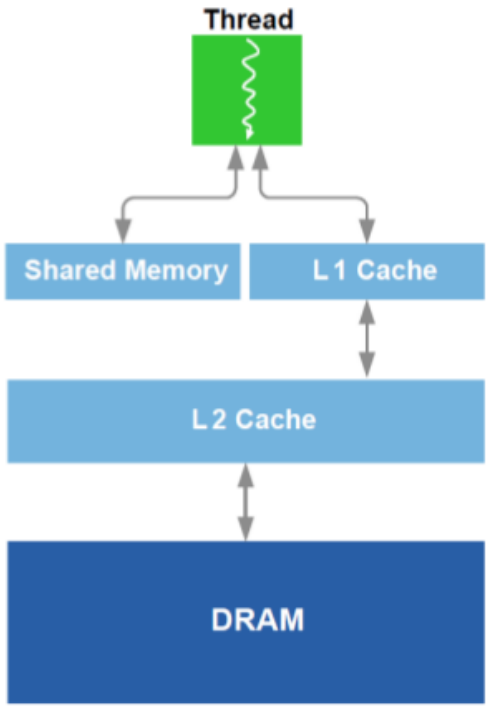
PTX 2.0 ISA

- Atomics para `fp32`, `int64` (además rápidas).
- Operaciones intra-warp: `ballot`, `any`, `all`.
- 64-bit addressing, UVA (unified virtual addressing).
- IEEE 754-2008 para `fp32`, `fp64`.
- FMA en vez de MAD para mayor precisión.
- Instrucciones predicadas (ARM-like).
- SIMD Video Instructions (vector-ops dentro de un vector core!)

Características, memoria

Streaming multiprocessor

- 128 KB register file.
- 64 KB de RAM
  (16/48 - 48/16 de L1/shared).



Memoria global

- Cache L2, 0.75GB, compartido por todos los SMs.
- 6 GB GDDR5 a 1.85 GHz.
- Opción de ECC.
- 6 bancos de 64 bits. 384 bits de ancho.
- Hay una TLB al medio.

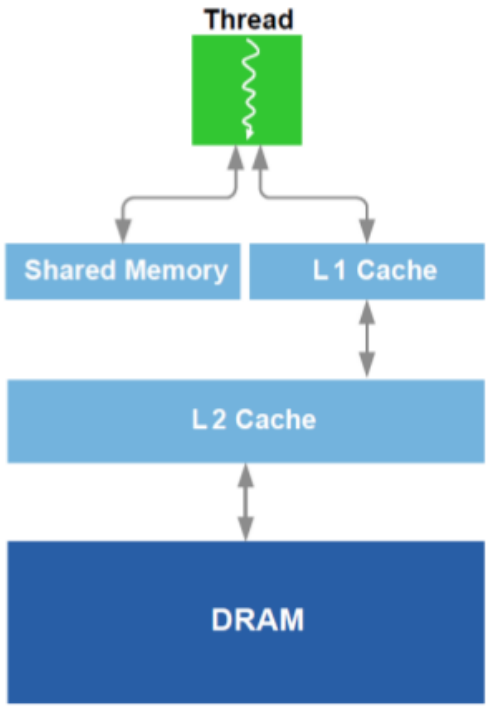
Ancho de banda de memoria pico

$$1.85\,GHz \times (384/8) \times 2/2^{30} = 165.4\,GiB/sec$$

Características, memoria

Streaming multiprocessor

- 128 KB register file.
- 64 KB de RAM
  (16/48 - 48/16 de L1/shared).



Memoria global

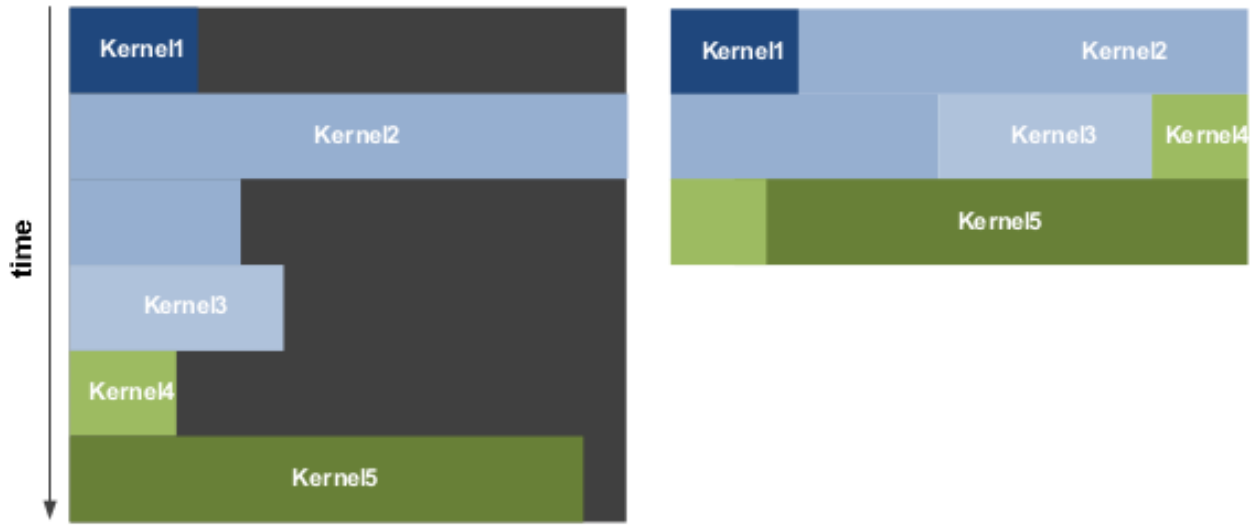
- Cache L2, 0.75GB, compartido por todos los SMs.
- 6 GB GDDR5 a 1.85 GHz.
- Opción de ECC.
- 6 bancos de 64 bits. 384 bits de ancho.
- Hay una TLB al medio.

Ancho de banda de memoria pico

$$1.85 GHz \times (384/8) \times 2/10^9 = 177.6 GB/sec$$

Planificador de bloques: GigaThread^{TM}

Permite ejecución concurrente de varios (≤ 16) kernels.



Objetivo: aumentar la utilización de la GPU.

Compute Capability

Las capacidades de la arquitectura están representadas por la CC. La M2090 es CC 2.0.

Cambios más importantes en las CC:

1.0 (G80)	se define la estructura de bloques, y la shared. Sin doble precisión, sin atomics.
1.1 (G86)	atomics.
1.2 (GT216)	atomics en shared.
1.3 (GT200)	doble precisión, warp vote, 3D grids.
2.0 (GF100)	atomics para `fp32`, predicated synchthreads, surfaces, `printf`, `assert`.
2.1 (GF104)	3 half-warp core groups (48), ILP.
3.0 (GK104)	$blockIdx.x \leq 2^{31}-1$, shared vs. L1 más configurable, ILP.
3.5 (GK110)	255 registros por hilo, paralelismo dinámico.

Típica tabla de CC

	Compute Capability				
Feature Support **(Unlisted features are supported for all compute capabilities)**	**1.0**	**1.1**	**1.2**	**1.3**	**2.x 3.0**
Atomic functions operating on 32-bit integer values in global memory (Section B.11)	No	Yes			
atomicExch() operating on 32-bit floating point values in global memory (Section B.11.1.3)					
Atomic functions operating on 32-bit integer values in shared memory (Section B.11)	No		Yes		
atomicExch() operating on 32-bit floating point values in shared memory (Section B.11.1.3)					
Atomic functions operating on 64-bit integer values in global memory (Section B.11)					
Warp vote functions (Section B.12)					
Double-precision floating-point numbers	No			Yes	
Atomic functions operating on 64-bit integer values in shared memory (Section B.11)	No				Yes
Atomic addition operating on 32-bit floating point values in global and shared memory (Section B.11.1.1)					
__ballot() (Section B.12)					
__threadfence_system() (Section B.5)					
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Section B.6)					
Surface functions (Section B.9)					
3D grid of thread blocks					

VectorSum

Compilemos pidiendo información al PTX Assembler `ptxas`.

```
nicolasw@mini:~/ECAR12/Clase1/VectorSum$ make
nvcc -O3 -arch=sm_20 --ptxas-options=-v -I/opt/cudasdk/C/common/inc -o vectorsum.o -c vectorsum.cu
ptxas info    : Compiling entry function '_Z9vectorsumPfS_' for 'sm_20'
ptxas info    : Function properties for _Z9vectorsumPfS_
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 6 registers, 4+0 bytes smem, 48 bytes cmem[0], 24 bytes cmem[2]
ptxas info    : Compiling entry function '_Z3setPfS_' for 'sm_20'
ptxas info    : Function properties for _Z3setPfS_
    32 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 14 registers, 48 bytes cmem[0], 24 bytes cmem[2], 44 bytes cmem[16]
nvcc -O3 -arch=sm_20 --ptxas-options=-v -I/opt/cudasdk/C/common/inc -o vectorsum vectorsum.o
```

vectorsum

- 6 registros por hilo

  No puede saturar un SM: $6 \times 1536 \leq 32768$.
- 4 bytes de memoria compartida por bloque.

  Muy por debajo de la capacidad de un SM: $8 \times 4 \leq 49152$.

No siempre es asi, este es un *toy example*.

Notar que `set` es mucho más complejo!

`VectorSum`, configuración de ejecución y planificación

Tamaño de bloques y como se distribuyen en los SM.

Block	BlocksPerSM	ThreadsPerSM	Occupancy	LimitedBy
64	8	512	$\frac{512}{1536} = 0.33$	MaxBlocksPerSM
128	8	1024	$\frac{1024}{1536} = 0.66$	MaxBlocksPerSM
192	8	1536	$\frac{1536}{1536} = 1.0$	Max{Blocks,Threads}PerSM
256	6	1536	$\frac{1536}{1536} = 1.0$	MaxThreadsPerSM
512	3	1536	$\frac{1536}{1536} = 1.0$	MaxThreadsPerSM
1024	1	1024	$\frac{1024}{1536} = 0.66$	MaxThreadsPerBlock

Resumen

- CPUs vs. GPUs.
- Ideas centrales de GPU Computing.
- Procesadores vectoriales, procesamiento escalar.
- Latency hiding.
- Concurrencia vs. memoria.
- Planificación de `<<<grid,block>>>` en los SM.
- Forma de trabajo de los niveles de la jerarquía de memoria.
- Un ejemplo: NVIDIA Tesla M2090.
- Un ejemplo: `vectorsum`.

Lo que sigue

- Descanso.
- Como, a partir de esta montaña de información, hacer que nuestros programas funcionen más rápido.