Resumen de Ampliación de Programación Avanzada

Introducción

Si bien existen distintos desarrollos de GPU's programables, nos centraremos principalmente en la arquitectura CUDA de NVIDIA, ya que fue ésta quien desarrollo en 2007 unas GPU's completamente programables y un SDK para las mismas.

Otra alternativa son las **tarjetas ATI y el OpenCL**, que también sirve para programar procesadores con múltiples núcleos, aunque no es tan completo como CUDA.

GL/glut

API para gráficos. A nivel de programador, trabajar con una tarjeta gráfica es complicado. Por ello, surgieron **interfaces** que abstraen la complejidad y diversidad de las tarjetas gráficas. Las dos más importantes son:

- **Direct3D**: lanzada por Microsoft. Utilizado por la mayoría de videojuegos lanzados por Microsoft.
- **OpenGL**: es gratuita, libre y multiplataforma. Utilizada principalmente en aplicaciones de CAD, realidad virtual o simulación de vuelo.

Cg o **C for Graphics** es un lenguaje de alto nivel desarrollado por NVIDIA en colaboración con Microsoft para la programación de vertex y pixel shaders.

BrookGPU fue desarrollado por la Universidad de Stanford, es un grupo de compiladores y aplicaciones basadas en el lenguaje Brool para utilizar con unidades de procesamiento gráfico.

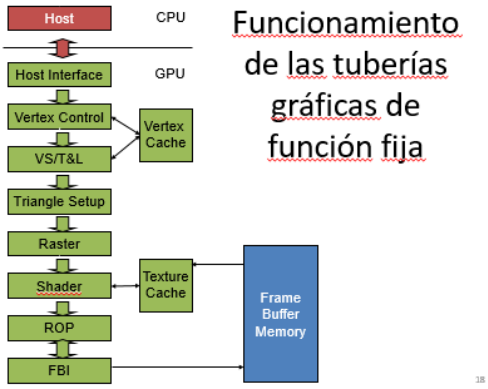
Componentes de la tarjeta gráfica

1. GPU
2. Memoria de video
3. RAMDAC conversor de señal digital a analógica
4. Disipador
5. Ventilador
6. Alimentación

La **ley de Moore** es un término informático que establece que la velocidad del procesador se duplica cada doce meses, y el de las GPU's cada 18 meses.

Mientras que la CPU se basa en la arquitectura de Von Neumann, la GPU tiene un diseño totalmente diferente conocido como **arquitectura streaming**. Mientras que los núcleos de CPU pueden trabajar de forma independiente, **los de la GPU dependen unos de otros**.

Nota: **Renderizado** es un término para referirse al **proceso de generar una imagen desde un modelo.** La renderización se aplica a la computación gráfica, más comúnmente a la **infografía** (proceso de desarrollo de un espacio 3D formado por estructuras poligonales).

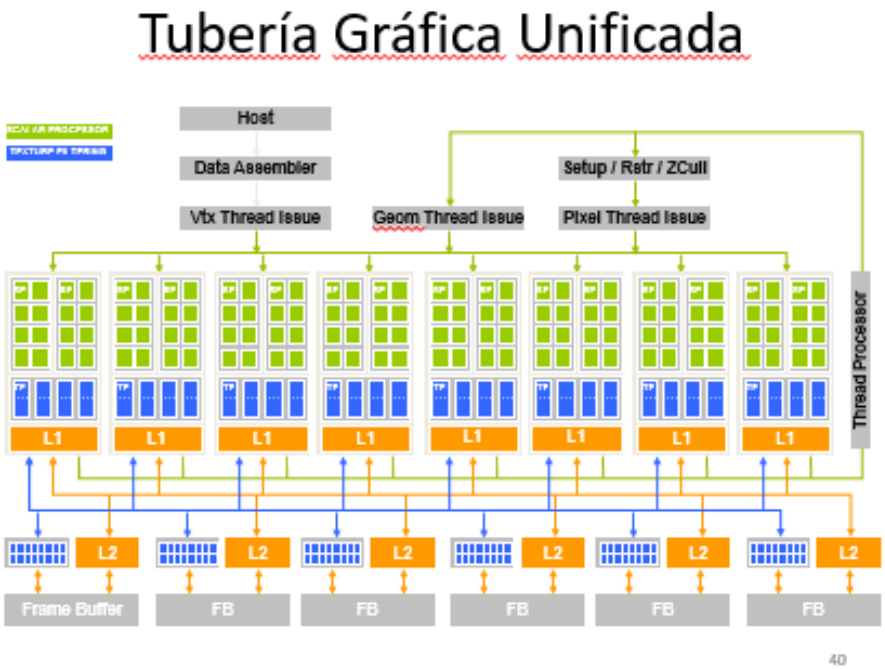


1. **Host Interface**: recibe comandos y datos de la CPU. Los comandos se dan mediante llamadas a la API. Comunica los resultados y el estado de vuelta al host.
2. **Vertex Control**: Los **objetos se representan como colecciones de triángulos**. Recibe los triángulos parametrizados de la CPU y los convierte a un formato adecuado y los coloca en el caché de vértices.
3. **Vertex Cache**: **transforma vértices y asigna valores a vértices**. El sombreado se realiza por el HW de sombreado de píxeles.
4. **Triangle Setup**: crea **ecuaciones** para las aristas que son usadas para interpolar colores y otra información por vértice a lo largo de los píxeles en contacto con el triángulo.
5. **Raster**: **determina qué píxeles están contenidos en cada triángulo**. Para cada píxel, interpola valores por píxel necesarios para el sombreado, incluyendo posición, color y textura que será pintado en el píxel.
6. **Shader**: **determina el color final de cada píxel**, combinando varias técnicas: interpolación de vértices de colores, iluminación, reflejos y más. Muchos de los efectos que dan impresión foto-realística se incorporan en esta fase. Aunque esta fase realiza pocos cambios de coordenadas para determinar la posición exacta de los puntos de textura, debe realizarse sobre un gran nº de ellos para cada frame.
    a. **Vertex shader**: permite transformaciones sobre coordenadas, normal, color, textura… de un vértice. No puede saberse el orden entre vértices ni pasarse información entre ellos.
    b. **Geometry shader**: es capaz de generar nuevas primitivas dinámicamente, así como de modificar existentes. Un ejemplo claro es la decisión de utilizar shaders en una malla poligonal según la posición del observador aplicando la técnica.
7. **ROP**: lleva a cabo el **rasterizado final** (conversión de información vectorial a píxeles). Mezcla el color de objetos superpuestos y efectos de antialiasing.
    a. **Aliasing** es el efecto que causa que señales continuas distintas se tornen indistinguibles cuando se muestra digitalmente.
8. **FBI**: interfaz del buffer de memoria maneja la lectura y la escritura del frame buffer.

Inconvenientes de la arquitectura

1. Programar vía API's gráficas: hay que formular los problemas en términos entendibles por las API's
2. Modos de direccionamiento: límite de tamaño de textura
3. Capacidades de los shader limitadas
4. Juego de instrucciones: faltan enteros y operaciones sobre bits
5. Comunicaciones limitadas entre píxeles y accesos raros

Tubería gráfica unificada



La tecnología de shaders es cualquier unidad escrita en un **lenguaje de sombreado** que se puede compilar independientemente. Los shaders son utilizados para realizar transformaciones y crear efectos especiales.

Diferencias de requerimientos

GPUs

- acceden direcciones de memoria contigua y tienen un uso eficiente del **ancho de banda** de la misma (poca importancia de la latencia),
- la carga de trabajo sobre los *pixel shader* y *vertex shader* es relativamente baja.

CPUs

- están dominadas por memorias de cache, los tiempos de acceso a la memoria (latencia) es importante,
- ejecutan instrucciones complejas

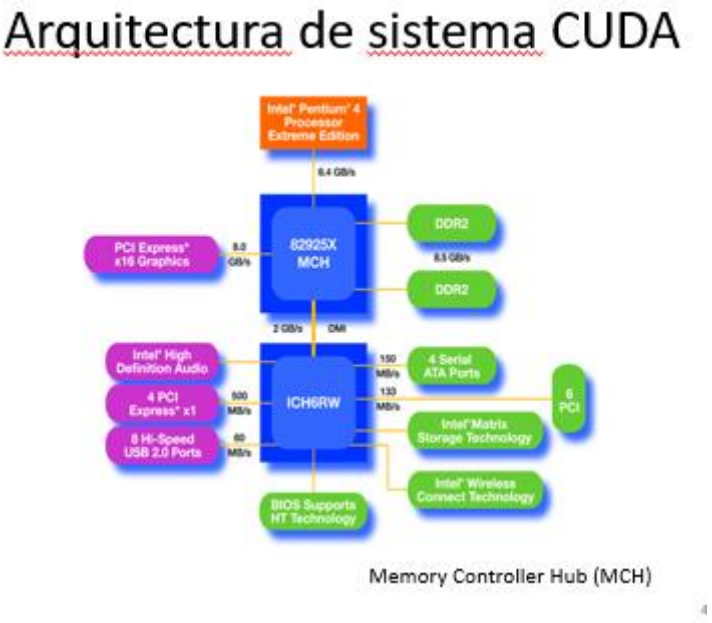
- **cpu**
- hardware de control Menos complejo
- + Flexibilidad + rendimiento
- - Caro en términos de poder
- **gpu**
- -más simple control de hardware
- + Más HW para el cálculo
- + Potencialmente más eficiente de la energía (ops / vatio)

Parte II

CUDA

Es un modelo de programación general que emplea lotes de hilos y un coprocesador masivamente paralelo (GPU). Tiene un **driver** para la carga de programas en la GPU, independiente y optimizado para la computación. La interfaz no dependiente de librerías gráficas y compartición de datos con objetos **OpenGL**. Manejo explícito de la memoria de la GPU.

LLVM es un compilador de estructura de código abierto ampliamente usado, **con un diseño modular** que hacen más sencillo agregar el soporte para lenguajes y arquitecturas de procesador. Soporta un amplio rango de lenguajes de programación, incluyendo C/C++, Fortran, Java, Phyton, etc.

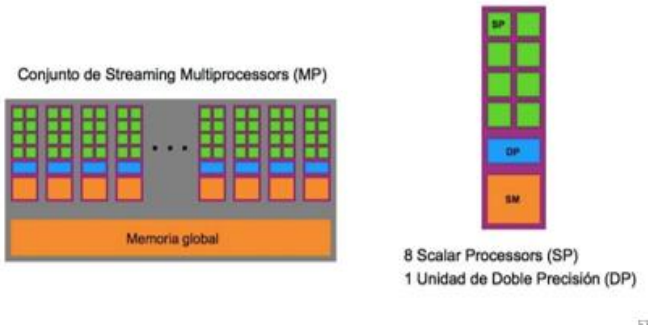


El sistema CUDA tiene la arquitectura que se muestra en la imagen anterior. La tarjeta GPU se inserta en la ranura PCIe x16.

Taxonomía de Flynn

- **SISD** (una instrucción y un flujo de datos); son máquinas secuenciales que no tienen **ningún paralelismo**. La unidad de control toma una instrucción de la memoria y genera señales adecuadas a un único elemento de procesamiento.
- **SIMD** (única instrucción sobre múltiples flujos de datos): las modernas GPU's se basan en ideas similares a estas. **Se adapta a CUDA**.
- **MISD** (múltiples instrucciones sobre un único flujo de datos): este tipo de arquitecturas tienen sus aplicaciones en entornos donde se necesita una **tolerancia a fallos** (repitiendo la misma instrucción sobre un dato para evitar posibles errores del HW).
- **MIMD** (múltiples instrucciones sobre múltiples flujos de datos): los **sistemas distribuidos** son clasificados generalmente como MIMD, bien usen memoria compartida o distribuida.

Arquitectura CUDA



Caraterísticas de la tarjeta G-80:

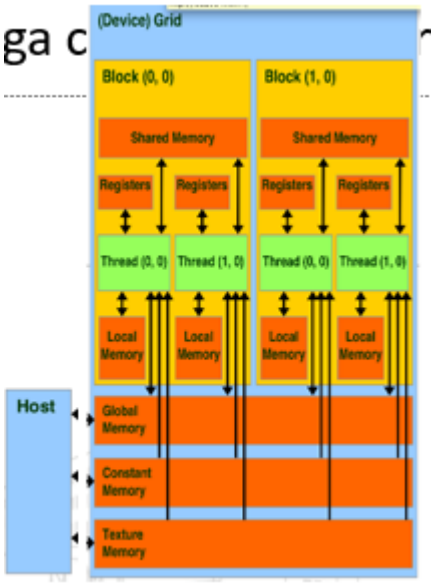
- Un SP puede tener 512 hilos.
- Un SM puede tener 768 hilos y SP.
- El número máximo de registros en de 8192, cada uno de 4 bits.
- Ancho de banda de 84.6 GgFlops
- Pico de 367 GgFlops.

Tipos de memoria

- **Global**: accesible en R/W por todos los hilos y por la CPU. Todos los hilos que ejecutan en la GPU tienen acceso al mismo espacio de memoria global. Es off-chip. **El acceso es más lento y no es cacheado**.
- **Local**: parte de la memoria global que es privada para cada hilo. Es una de las memorias más lentas de la GPU. Es off-chip. **Este espacio de memoria es gestionado por el compilador.**
- **Constant memory**: global de solo lectura. **Puede cargarse en caché** de SM para acelerar las transferencias. Es una memoria rápida.
- **Shared memory**: de tamaño limitado 16 KB. **Accesible en R/W por los hilos de un mismo SM**. Cada bloque tiene un espacio de memoria compartida que es prácticamente tan rápida como los registros. Puede ser accedida por cualquier hilo del bloque. Es on-chip. Su tiempo de vida es igual al del bloque.
- **Registro**: mismo número para todos los hilos, accesible en R/W de forma **privada**. Es on-chip. Es la más rápida de la GPU. Sólo son accesible por cada hilo del bloque correspondiente (cada hilo del mismo bloque solo puede acceder a sus registros). **Este espacio de memoria es gestionado por el compilador.**

  - **Tamaño máximo de los espacios de memoria en la arquitectura G80:**
    - **Global: 768 MB – 1024 MB**
    - **Local: 16 KB**
    - **Compartida: 16 KB**
    - **Registros: 8 KB por multiprocesador**
    - **Constante: 64 KB (8 KB por TPC)**
    - **Texturas: 8 KB por TPC**

Nota: en el chip componentes son los más incorporados en el chip en sí, es decir, en el mismo sustrato de silicio como transistores, condensadores, etc.



Modelo de memoria CUDA

- Memoria global: el host y el device pueden leer y escribir. El contenido es visible a todos los hilos y tiene una alta latencia.
- Memoria de constantes y de texturas: nada

Nota: ver ejemplos de código a partir de la diapositiva 69-82

Conflictos con la memoria compartida

Distintos hilos del mismo medio **warp** acceden al mismo banco de memoria. Debido a esto, se debe **serializar** el acceso, donde el coste será igual al nº máximo de accesos simultáneos a un único banco.

Como compilar un programa C en CUDA

El código fuente está integrado en un fichero **foo.cu**

El compilador de CUDA (**NVCC**) utiliza un preprocesador (**CUDAC**) que divide la aplicación en la CPU y la GPU partes, creando un archivo preprocesado C para la entrada de GPU: **foo.s** (lenguaje ensamblador) para GPU y **foo.cpp** para CPU.

La entrada de la GPU es procesada por nuestro compilador **Open64** (llamado nvopencc), que emite un lenguaje ensamblador que hemos creado llamado **PTX** (Ejecución de subprocesos en paralelo).

PTX ofrece un modelo de máquina virtual que es independiente del procesador subyacente. Algunas características importantes de PTX son que tiene:

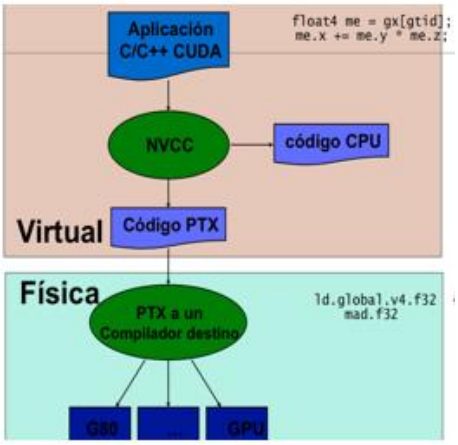
- Registros ilimitados de diferentes tamaños
- sin pila o montón
- Instrucciones inflexibles de tipos
- vectores de soporte para accesos a memoria
- C-como sintaxis para las llamadas

La PTX se pasa entonces a **OCG**, que asigna los registros y la lista de las instrucciones de acuerdo con el chip en particular que se está utilizando (programación) y optimización.

GCC es el GNU C / C + + y es el estándar de facto para Unix (similares a) los sistemas. Es de código abierto ...

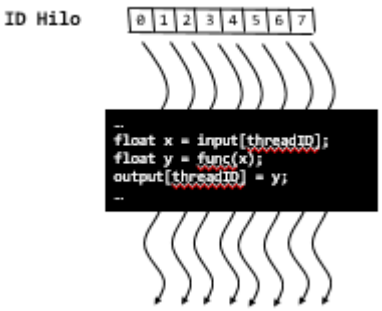
CL, IIRC, es parte de Visual MS C + + paquete

Cualquier ejecutable CUDA requiere dos librerías: **cudart y cuda**



Array de hilos

El kernel de CUDA se ejecuta en un array de hilos, donde todos los hilos ejecutan el mismo código y cada hilo tiene una **única ID** empleada para acceder a la memoria y controlar las decisiones.



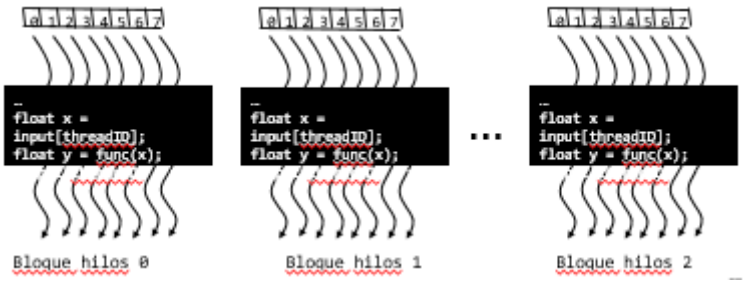
El array de hilos se divide en múltiples **bloques**, donde los hilos de dentro de un bloque colaboran mediante **memoria compartida, operaciones atómicas y sincronizaciones de barrera**. Los hilos de diferentes bloques no pueden colaborar.

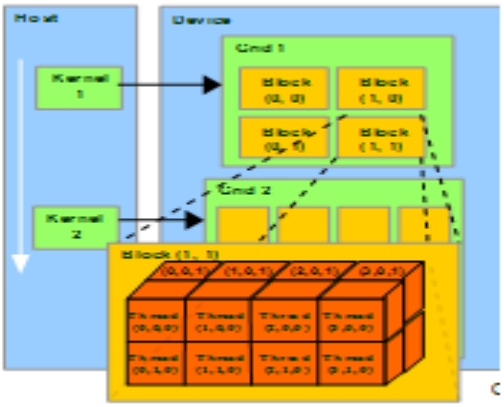
La **operación atómica** es una operación en la que un procesador puede simultáneamente leer una ubicación y escribirla en la misma operación del bus. Esto previene que cualquier otro procesador o dispositivo de E/S escriba o lea la memoria hasta que la operación se haya completado.

- Los **procesadores escalares** son el tipo más simple de procesadores. **Cada instrucción de un procesador escalar opera sobre un dato cada vez → SISD.**
- En contraposición, en un **procesador vectorial** **una sola instrucción opera simultáneamente sobre un conjunto de datos → SIMD**.

Para saber dónde tengo un hilo dentro de un conjunto de bloques: **nº de bloques * identificador de bloque + identificador de hilo**.

Ejemplo: (1,32,32) = 1*32*32 = 1024 -> no entra en un SM porque el tamaño máximo de este es de 768





Algunas funciones de CUDA

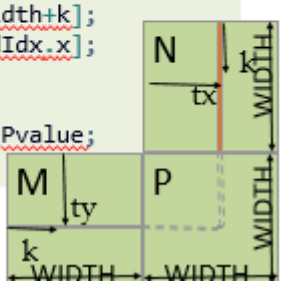
- **CudaMemcpy()** → transfiere datos de forma asíncrona. Requiere 4 parámetros: puntero al destino, puntero al origen, nº de bytes a copiar y tipo de transferencia (host a host, device a host, etc)
- **__device__ float DeviceFunc()** → se ejecuta en la GPU. Las funciones que se ejecutan aquí no pueden ser recursivas.
- **__global__ void KernelFunc()** → se ejecuta en la GPU y puede ser ejecutable desde la CPU.
- **__host__ float HostFunc()** → se ejecuta en el host.
- **dim3 tamGrid(1, 1);** //Grid dimensión
- **dim3 tamBlock(1,N,1);** //Block dimensión
- **Void __syncthreads()** → sincroniza todos los hilos de un bloque y una vez que todos están sincronizados la ejecución continua.

Lanzamiento de threads con Nbloques de 128 threads: int bloques = N / 128;

```
dim3 tamGrid(1, bloques); //Grid dimensión
dim3 tamBlock(1, 128, 1); //Block dimensión
// Launch the device computation threads!
MatrixSumKernel <<<tamGrid, tamBlock>>>( M, Md, Nd);
```

Ejemplo de código CUDA de multiplicación de matrices

```
__global__ void MatrixMulKernel(float* Md, float* Nd,
float* Pd, int Width) {
  // Usamos Pvalue para almacenar el valor de la
  // matriz calculado por el thread
  float Pvalue = 0;
  for (int k = 0; k < Width; ++k){
    float Melement = Md[threadIdx.y*Width+k];
    float Nelement = Nd[k*Width+threadIdx.x];
    Pvalue += Melement * Nelement;
  }
  Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```



Depuración empleando emulación

Un ejecutable compilado en device emulation mode (**nvcc -deviceemu**) se ejecuta en el host vía CUDA runtime, sin necesitar ningún dispositivo CUDA **y correspondiéndose cada hilo con un hilo del host.**

Algunos de los **fallos del modo de emulación**:

- Los hilos emulados se ejecutan secuencialmente, por lo tanto, **acceso simultáneo** de las mismas direcciones de memoria por múltiples hilos pueden producir resultados diferentes.
- **Diferenciar punteros del dispositivo en el host** o punteros del host en el dispositivo pueden producir resultados correctos en device emulation mode, pero generarán un error al ejecutarse en el dispositivo.
- **Coma flotante:** los resultados de este tipo de cálculos serán ligeramente diferentes porque al haber diferentes salidas de compilación, también habrá diferentes juegos de instrucciones y el empleo de precisión extendida para los resultados intermedios.

Ejemplo de multiplicación de matrices usando múltiples bloques

```
__global__ void MatrixMulKernel(float* Md, float* Nd,
    float* Pd, int Width)
{
// Calculate the row index of Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column idenx of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;
float Pvalue = 0;
// each thread computes one element of the block sub-
    matrix
for (int k = 0; k < Width; ++k)
    Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```

```
// Setup the execution configuration

dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

// Launch the device computation threads!

MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd,
Width);
```

Bloques de hilos en CUDA

Todos los hilos de un bloque se ejecutan en el mismo Kernel (SPMD). El programador declara bloques, cuyos hilos tienen un ID específico dentro de cada bloque. El programa usa el ID de los hilos para **seleccionar la tarea** que tienen que realizar. Los hilos de un mismo bloque comparten

datos y se pueden sincronizar mientras realizan sus tareas, mientras que los hilos de diferentes bloques no cooperan.

El HW puede asignar bloques a cualquier procesador en cualquier momento. Cada bloque se puede ejecutar en cualquier orden relativo a otros bloques.

En la G80 todos los hilos de un mismo bloque se agrupan en conjuntos de 32 (**warp**). Los hilos de un warp se planifican en los 8 cores de un SM, lo que da como resultado la **ejecución paralela de 32 hilos en el mismo instante de tiempo (SIMD).**

Los SM implementan la planificación de warps sin sobrecarga. Los warps elegibles para la ejecución son seleccionados para la ejecución de acuerdo a una **política de planificación priorizada** (ver más adelante concepto de **ROUND ROBIN**). Todos los hilos de un mismo warp ejecutan la misma instrucción cuando son seleccionados.

Ejemplo 1: 1 bloque de 256 hilos se divide en 256/32=8 warps que son planificados para la mejorar la eficiencia.

- Cada bloque activo se divide en warps.
- Los hilos de un warp son ejecutados físicamente en paralelo.
- Warps y bloques se planifican en el SM.

Ejemplo 2: Si 3 bloques son asignados a una SM y cada bloque tiene 256 hilos, ¿cuántos warps hay en cada SM?

- Cada bloque se divide en 256/32 = 8 warps
- Hay 8 * 3 = 24 warps

Ejemplo 3: ¿Para la multiplicación de matrices debemos usar bloques de 8X8, 16X16 ó 32X32?

- Para 8X8, tenemos 64 hilos por bloque. Como cada SM puede ejecutar hasta 768 hilos, hay 12 bloques. Pero como cada SM puede albergar sólo 8 bloques, ¡sólo 512 hilos irán a cada SM!
- Para 16X16, tenemos 256 hilos por bloque. Cada SM acoge a 3 bloques y conseguimos máxima capacidad (salvo que otras consideraciones indiquen otras estrategias).
- Para 32X32, tenemos 1024 hilos por bloque. Ni siquiera caben en un SM.

Estrategia TILING

Es una técnica de aprovecha **los recursos de la memoria compartida**, en la cual **se divide el problema en bloques de computación con subconjuntos que quepan en la memoria compartida**. Primero se carga de un subconjunto de memoria global a memoria compartida, usando varios hilos para aprovecha el paralelismo, después se realizan las operaciones sobre cada subconjunto en memoria compartida y por último se copian los datos de memoria compartida a global.

Parte III

API

Es una **extensión del lenguaje C** cuyo objetivo es la parte del código que se ejecuta en el dispositivo. Contiene una librería de ejecución que se divide en:

- Un componente común que suministra vectores
- Un componente del anfitrión para controlar uno o más dispositivos desde el anfitrión
- Un componente del dispositivo que provee funciones específicas del mismo

Tiempos de acceso a instrucciones → IMPORTANTE

A nivel de instrucción, toma los siguientes ciclos de reloj para llevar a cabo el lanzamiento de instrucciones:

- **Instrucciones aritméticas**: **4 ciclos** de reloj para suma, producto y MAD sobre operandos en coma flotante. 4 ciclos de reloj de suma y operaciones a nivel de bit en operandos enteros.
- **Operaciones logarítmicas**: **16 ciclos** de reloj.
- **Los accesos a memoria**: tienen un **sobrecoste de 4 ciclos** de reloj, tanto a nivel de memoria global como de memoria compartida. El acceso a **memoria global suele acarrear entre 400 y 600** ciclos de reloj debido a la latencia propia del acceso a niveles inferiores.
- **Instrucciones de sincronización**: supone un sobrecoste de **4 ciclos de reloj.**

Nota: La alta especialización de este tipo de procesadores en operaciones sobre coma flotante queda demostrada por coste de una operación de multiplicación sobre datos enteros de 32 bits: 16 ciclos de reloj.

Ejemplo de kernel:

Multiplicación de matrices por memoria compartida

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column idenx of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```

Nota: ver más a fondo diapositiva 133- 143

La ventaja del **algoritmo de baldosas** es sustancial. Por matriz de multiplicación, el acceso a la memoria global se reduce **por un factor de TILE_WIDTH.** Si uno utiliza **16 x 16 azulejos,** podemos **reducir los accesos de memoria global por un factor de 16.** Esta reducción permite que el ancho de banda de 86,4 GB / s de memoria global para servir a una mucho mayor tasa de punto flotante de cálculo que el algoritmo original. Más específicamente, el ancho de banda de memoria global puede ahora apoyo [(86,4 / 4) x 16] = 345,6 gigaflops, muy cerca del pico de punto flotante de rendimiento de la G80. **Esto elimina eficazmente la memoria global del ancho de banda** como el principal factor limitante del rendimiento de multiplicación de matrices.

Rendimiento de la G80

- Soporta 86.4 GB/s de acceso a memoria global
- Se necesitan 4 ciclos de reloj para ejecutar la misma instrucción para todos los hilos del warp
- No se puede cargar más de 86,4/4=21,6 Gb/s -> [(86.4/4) x 16] = 345.6 gigaflops, < Theoretical peak
- El código de hecho corre a unos 15 GFlOPS
- Necesita cortar los accesos a memoria para acercarse al pico de 346.5 GFLOPS. Los **FLOPS** son operaciones de coma flotante por segundo, que es una medida de rendimiento del ordenador.

IMPORTANTE: El **ancho de banda de una gráfica** es igual a la velocidad de la memoria en Giga-hercios multiplicado por la interfaz de la memoria en bits y el resultado se divide entre ocho para pasarlo a GB/s

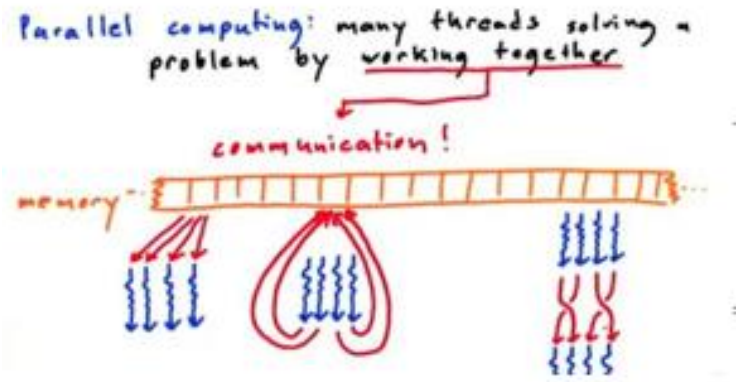
Ejemplo 1: una GTX 660 (No TI) que las especificaciones de la que pides no las sé pero se haría igual: La GTX 660 tiene una memoria de 6008 Mega-hercios que serían 6,008 GHz esto se multiplica por 192 bits que es su interfaz de memoria, el resultado de esta operación es 1153,536 que dividido entre 8 nos da un resultado de 144,192 GB/s.

Ejemplo 2: GTX 780 cuenta con 6 GHz de velocidad de la memoria y unos increíbles 384 bits de interfaz. Operación matemática: (6 * 384)/8 = 288 Gb/s con este resultado se ve que la GTX 660 que es capaz de tirar muchos juegos en ultra a 1080p es doblada en ancho de banda por la GTX 780

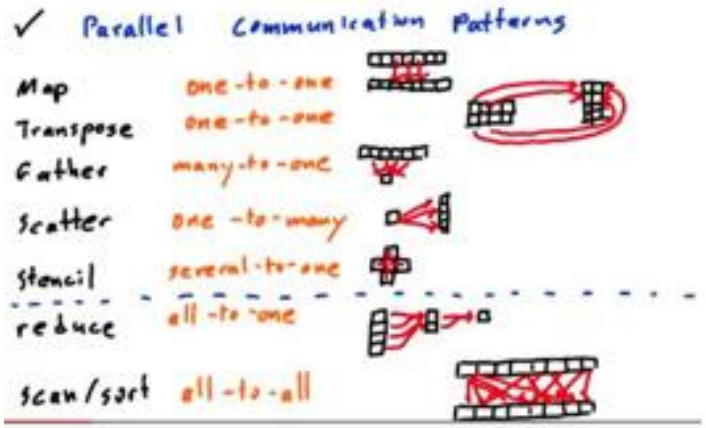
Compute Capabilities

Para hacer uso óptimo de las GPU's es necesario conocer diferentes características de la tarjeta. Para ello, NVIDIA utiliza un formato estandarizado para especificar estas características denominado **compute capabilities**. La categorización incluye dos números, donde **los cambios en la primera cifra implican cambios de generación, mientras que en la segunda implica una revisión**. Ver en más profundidad diapositivas 151 y 152.

Consideraciones sobre el rendimiento



- Mapear: de uno en uno
- Transponer: de uno en uno
- Reunir: muchos a uno
- Dispersión: uno a muchos
- Plantilla: varios a uno
- Reducir: todos a uno
- Ordenar: todos con todos



Paralelismo

La idea fundamental es estructurar un código de tal manera que muestre mucho paralelismo. La aplicación debe, así mismo, maximizar la ejecución paralela **mediante el aprovechamiento de las ejecuciones simultáneas en el dispositivo a través de los flujos de datos**, así como de los trasiegos que se producen entre el Host y el dispositivo.

Los procesadores incorporan cantidades crecientes de paralelismo. Las instrucciones se ejecutan de forma simultánea por diferentes unidades funcionales, y en los procesadores superescalares, incluso se inicia ("acciones") en paralelo.

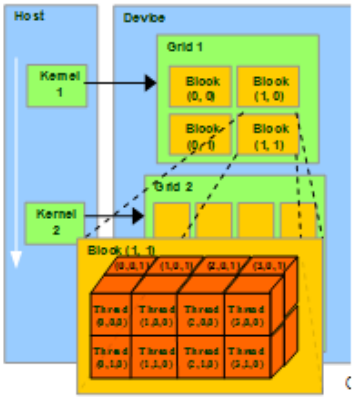
- Esto se conoce como **nivel de instrucción paralelismo, o ILP.**
- Paralelismo entre los procesadores se llama **paralelismo a nivel de hilo (TLP).**

Sobre los hilos

El código secuencial en CUDA se ejecuta en la CPU, mientras que el núcleo paralelo se ejecuta en bloques de hilos en GPU. Al núcleo de CUDA también se le llama **procesador de streaming**.

Un núcleo se ejecuta en una cuadrícula de hilos, en el que todos los hilos acceden a la memoria compartida.

El ciclo de vida de un hilo en HW consiste en lanzar una cuadrícula en la SPA, distribuyéndose secuencialmente los bloques de hilos en los SM. Cada SM lanza warps de hilos. Los SM planifican y ejecutan los warps que estén listos para ejecutarse y a medida que los warps y bloques se completan los recursos se liberan.



Terminología de procesadores CUDA:

- SPA: Streaming Processor Array
- TPC: Texture Processor Cluster (2 SM + Text)
- SM: Streaming Multiprocesor (8 SP)
- SP: Streaming Processor

TPC

Es un concepto que encontramos en las GPU's de NVIDIA. El TPC **es un grupo formado por varios SM**, una unidad de textura de lógica de control. El TPC de una GPU G80 tiene dos SM, y cada SP incluye varias ALU y FPU (unidad de operaciones en coma flotante).

La unidad de textura procesa un grupo de 4 hilos por ciclo, donde las instrucciones fuente son coordenadas de textura y las salidas filtran las muestras. Cada unidad de textura tiene 2 SM.

Planificación de warps en SM

El HW de las SM implementa una planificación sin sobrecargas:

- Los warps cuya siguiente instrucción y sus operandos están listos son elegibles para su ejecución
- Los warps listos para ser ejecutados son priorizados de acuerdo a ciertas políticas
- Todos los hilos de un warp ejecutan la misma instrucción cuando son seleccionados
- Se necesitan **4 ciclos de reloj para ejecutar la misma instrucción** para todos los hilos del warp

El buffer de instrucciones coge una instrucción por ciclo para el warps de la caché de instrucciones de nivel 1, en cualquier slot de instrucciones del buffer. La selección está basada en un criterio de **Round Robín** del warps. Round Robín es un método para seleccionar todos los elementos en un grupo de manera equitativa y en un orden racional, normalmente comenzando por el primer elemento de la lista hasta llegar al último y empezando de nuevo desde el primer elemento.

Nota: la puntuación de instrucciones → todos los registros para operandos de todas las instrucciones en el buffer de instrucciones están marcadas.

Mezcla de Instrucciones

En las GPUs que soportan CUDA, cada procesador tiene un ancho de banda limitado para procesar instrucciones. Cada instrucción consume ancho de banda, tanto si es en coma flotante, como si es una instrucción de carga, o una bifurcación. Por ejemplo, en un bucle la variable indexadora es actualizada en cada iteración y esta operación compite con las demás instrucciones en el consumo de ancho de banda.

Granularidad de los Hilos

En muchas ocasiones, es mucho más eficiente poner una gran carga de instrucciones en pocos hilos que usar la opción contraria. Esto es especialmente indicado cuando existen tareas redundantes entre los hilos.

- **Procesador de grano fino**: operaciones elementales. En Granularidad Fina las tareas individuales son relativamente pequeñas en término de tiempo de ejecución. **La comunicación entre los procesadores es frecuente**.
- **Procesador de grano grueso:** operaciones complejas. En granularidad gruesa la **comunicación entre los procesadores es poco frecuente** y se realiza después de largos periodos de ejecución.

Coalescencia

Es el acto de **la fusión de dos bloques libres adyacentes de memoria**. Cuando una aplicación libera memoria, las lagunas pueden caer en el segmento de memoria que utiliza la aplicación. Se utiliza para reducir la fragmentación externa y en la recogida de basura.

La técnica se ilustra en la Figura para la multiplicación de matrices. Cada hilo lee una fila de Md, un patrón que no puede ser colateral. Afortunadamente, **un algoritmo de azulejos se puede utilizar para permitir la coalescencia**. Como ya comentamos, hilos de un primer bloque de forma cooperativa pueden cargar los azulejos en la memoria compartida. Cuidado se pueden tomar para asegurar que estas baldosas son cargados en un patrón aglutinado. Una vez que están en la memoria compartida, se puede acceder a los datos en cualquier forma fila o una base de columna sin ninguna penalización en el rendimiento debido a que las memorias compartidas se

implementan como intrínsecamente de alta velocidad, la memoria en el chip que no requiere de coalescencia para lograr un alto datos tasa de acceso.

Ejemplo de direccionamiento lineal en la diapositiva 194

Control de flujo

El mayor problema de rendimiento lo plantea la divergencia por saltos ya que los hilos de un mismo warp pueden tomar distintos caminos y estos caminos se recorren de uno en uno hasta que se acaban y se serializan.

Un caso frecuente es el de evitar la divergencia cuando el salto depende de la ID del hilo:

- **Con divergencia**: crea dos caminos de control para los hilos de un bloque. La fineza del salto es menor que el tamaño de un warp. Los hilos 0,1 y 2 siguen caminos distintos al resto de los hilos en el primer warp.
- **Sin divergencia**: también crea dos caminos de control para los hilos de un bloque. La fineza del salto es un múltiplo del tamaño del warp. Todos los hilos de un warp siguen el mismo camino.

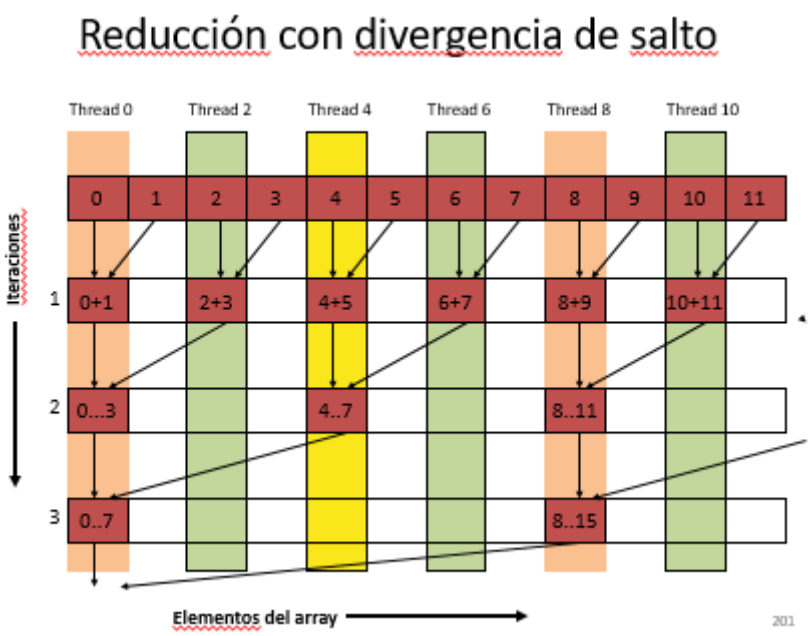
Nota: **serializar** es el proceso de codificar un objeto en un medio de almacenamiento con el fin de transmitirlo a través de una conexión de red como una serie de bytes en un formato legible. El chorro de bytes puede ser usado para crear un nuevo objeto que es idéntico en todo al original, incluyendo su estado interno.

Reducción paralela

Dado un array de valores, reducirlo a un único valor en paralelo. La implementación típica consiste en reducir a la mitad el número de hilos, sumar dos valores del hilo y llevar log(n) pasos para n elementos y requerir n/2 hilos.

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride < blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
     partialSum[t] += partialSum[t+stride];
}
```

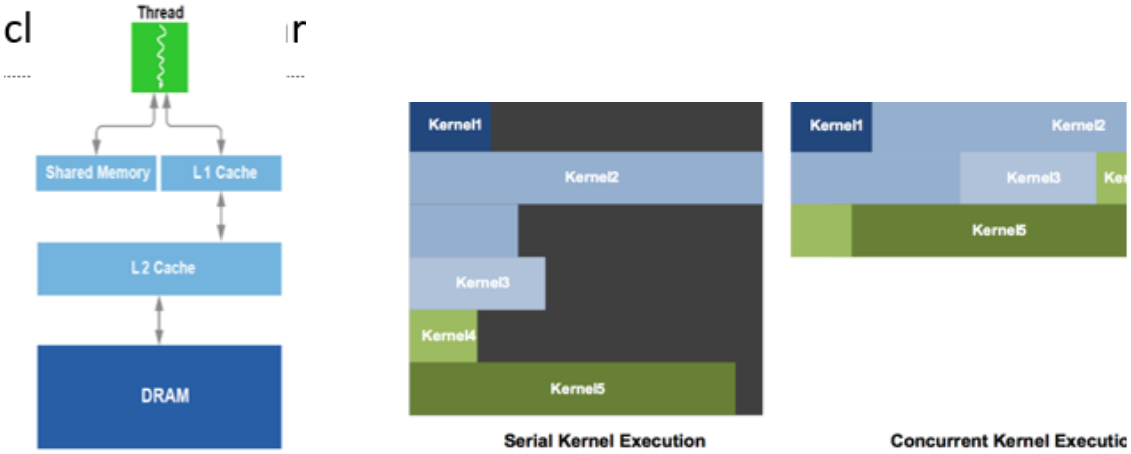


En cada iteración, dos controles de flujo se atraviesan secuencialmente en cada wapr. Hay hilos que calculan adiciones e hilos que no. Los hilos que no calculan adiciones pueden incurrir en ciclos extra dependiendo de la implementación de la divergencia …. ACABAR DIAPO 202

Principales factores de decremento de rendimiento en la G80

- **Operaciones de alta latencia**: evitar esperas ejecutando otros hilos.
- **Esperas y burbujas en la cascada**: sincronización de barrera y divergencia por saltos.
- **Saturación de recursos compartidos**: ancho de banda de la memoria global y capacidad de la memoria local.

Fermi



Compartición de cache L1 con la memoria compartida

Fermi es la primera GPU que ofrece una **caché L1 típica on-chip**, que combina con la **memoria compartida de CUDA para un total de 64 KB por cada multiprocesador** (32 cores). También incluye **una caché unificada L2 de 768 KB** con coherencia de datos para el conjunto total de cores.

Triplicar la cantidad de memoria compartida produce mejoras significativas de rendimiento, especialmente para los problemas que tienen limitación de ancho de banda.

- Para las aplicaciones que usan memoria compartida como caché administrado por SW, el código puede optimizarse para aprovechar el sistema de almacenamiento en caché de HW, al mismo tiempo que tiene acceso a al menos 16 KB de memoria compartida para la cooperación de hilos explícita.
- Lo mejor de todo es que **las aplicaciones que no utilizan memoria compartida se benefician automáticamente de la caché L1**, lo que permite que los programas CUDA de alto rendimiento se construyan con el mínimo tiempo y esfuerzo.

Ejecución concurrente de Kernels

Fermi admite la **ejecución concurrente de kernels**, donde los diferentes núcleos del mismo contexto de aplicación se pueden ejecutar en la GPU al mismo tiempo. La ejecución concurrente de kernels permite que los programas que se ejecutan, usen una serie de pequeños núcleos de utilizar toda la GPU.

En la arquitectura Fermi, **diferentes núcleos del mismo contexto CUDA pueden ejecutar simultáneamente, lo que permite la máxima utilización de los recursos de la GPU.**

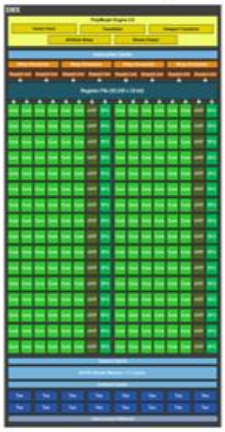
Ejecución de subprocesos en paralelo de segunda generación ISA

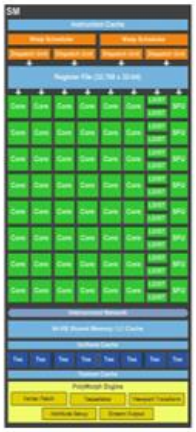
Fermi es la primera arquitectura que soporta PTX. **PTX es una máquina virtual de bajo nivel diseñada para soportar las operaciones de un procesador de hilos paralelo**. En el momento de la instalación del programa, las instrucciones de PTX se traducen a las instrucciones de la máquina mediante el controlador de la GPU.

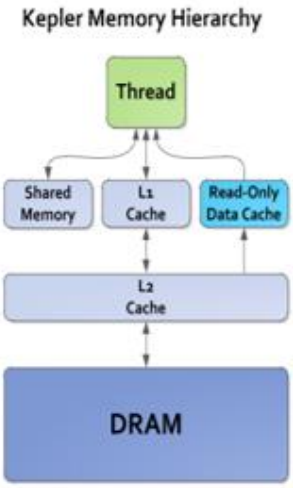
Los objetivos principales de PTX son:

- Proporcionar una ISA estable que abarca varias generaciones de GPU
- Alcance el rendimiento completo de la GPU en aplicaciones compiladas
- Proporcione un ISA independiente de máquina para C, C ++, Fortran y otros objetivos del compilador.
- Proporcionar una distribución de código ISA para desarrolladores de aplicaciones y middleware
- Proporcione un ISA común para optimizar generadores de código y traductores, que asignen PTX a máquinas de destino específicas.
- Facilitar la codificación manual de las bibliotecas y los núcleos de rendimiento
- Proporcionar un modelo de programación escalable que abarca los tamaños GPU de unos cuantos núcleos a muchos núcleos paralelos

Kepler GK110







64 KB Memoria compartida configurable y caché L1

En la arquitectura Kepler, como en la arquitectura Fermi de generación anterior, cada SM tiene 64 KB de memoria on-chip que puede configurarse como 48 KB de memoria compartida con 16 KB de caché L1 o como 16 KB de memoria compartida con 48 KB de caché L1. **Kepler ahora permite una flexibilidad adicional** en la configuración de la asignación de memoria compartida y caché L1 permitiendo una división de **32KB / 32KB** entre memoria compartida y caché L1.

48 KB de caché de datos de sólo lectura

Además de la caché L1, Kepler introduce **un caché de 48 KB para datos que se sabe que son de sólo lectura** durante la duración de la función. En la generación Fermi, **este caché era accesible sólo por la unidad de textura**. Los programadores expertos a menudo encontraron ventajoso para cargar datos a través de esta ruta explícitamente mediante el mapeo de sus datos como texturas, pero este enfoque tenía muchas limitaciones.

Caché mejorado de L2

La GPU Kepler GK110 cuenta con **1536KB** de memoria caché dedicada L2, el doble de la cantidad de L2 disponible en la arquitectura Fermi. La caché L2 es el punto principal de la unificación de

datos entre las unidades SMX, atendiendo todas las solicitudes de carga, almacenamiento y textura y proporcionando un intercambio de datos eficiente y de alta velocidad en toda la GPU.

Soporte de protección de memoria

Al igual que Fermi, los archivos de registro de Kepler, las memorias compartidas, la caché L1, la memoria caché L2 y la memoria DRAM están protegidos por un **código EDC de detección de doble error (SECDED) de error único.**

Quad Warp Scheduler

El SMX programa hilos en grupos de 32 hilos paralelos llamados deformaciones. Cada SMX permite que se emitan y se ejecuten simultáneamente cuatro warps. El planificador de urdimbre quad de Kepler selecciona cuatro deformaciones y se pueden enviar dos instrucciones independientes por urdimbre cada ciclo. A diferencia de Fermi, que no permitía que las instrucciones de doble precisión fueran emparejadas con otras instrucciones, Kepler GK110 permite que las instrucciones de doble precisión se emparejen con otras instrucciones.

Paralelismo dinámico

En un sistema híbrido CPU-GPU, permitir que una mayor cantidad de código paralelo en una aplicación se ejecute eficientemente y totalmente dentro de la GPU mejora la escalabilidad y el rendimiento a medida que las GPU aumentan en perf / watt. Para acelerar estas partes paralelas adicionales de la aplicación, las GPU deben soportar tipos más variados de

Cargas de trabajo paralelas

El paralelismo dinámico es una nueva característica introducida con Kepler que **permite a la GPU generar un nuevo trabajo para sí mismo, sincronizar los resultados y controlar la programación de ese trabajo a través de vías de hardware aceleradas dedicadas, todo sin involucrar a la CPU**.

Para procesamiento adicional

En Kepler **cualquier kernel puede lanzar otro kernel**, y puede crear los flujos, eventos y administrar las dependencias necesarias para procesar trabajo adicional sin necesidad de interacción de la CPU del host. Esta innovación arquitectónica facilita a los desarrolladores la creación y optimización de patrones de ejecución recursivos y dependientes de datos, y permite que más de un programa se ejecute directamente en la GPU. La CPU del sistema puede liberarse para tareas adicionales o el sistema puede configurarse con una CPU menos potente para realizar la misma carga de trabajo.