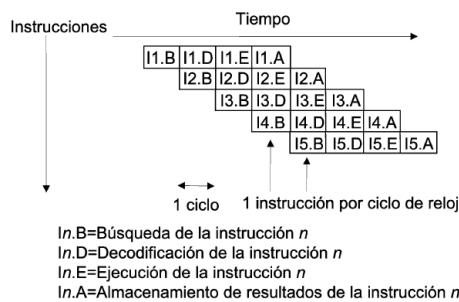


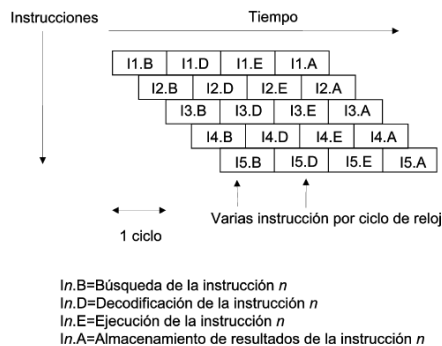
RESUMEN PEC 1

La **computación en paralelo** consiste en usar varios procesadores para resolver una tarea común al mismo tiempo. Con lo que cada procesador trabaja en una porción del problema e intercambian datos mediante la **memoria compartida** y el paso de **mensajes en una red de interconexión**. La computación en una sola CPU está limitada por la memoria y por el rendimiento (plazo de tiempo determinado), con lo que la computación en paralelo no está limitada ni por la memoria ni por el rendimiento.

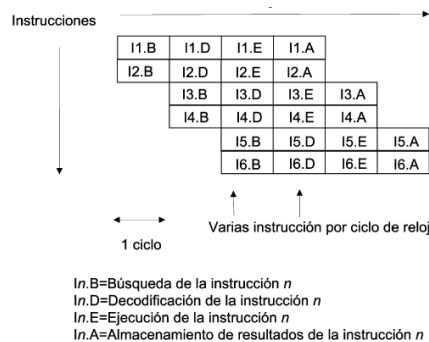
Con el **procesador segmentado**, una tarea se divide en etapas, y en cada ciclo de reloj se lanza una etapa



Con el **procesador supersegmentado**, una tarea se divide en etapas, y en cada ciclo de reloj se lanzan varias etapas (pero no al mismo tiempo, dentro de ese ciclo de reloj)



Con el **procesador superescalar**, una tarea se divide en etapas, y en cada ciclo de reloj se lanzan varias etapas de forma simultánea (al mismo tiempo, dentro de ese ciclo de reloj)



Con el **multithreading**, se lanzan varios hilos simultáneamente, los cuales comparten recursos

Los **computadores paralelos dependen de:**

- Número de procesadores
- Tipo de procesadores:
 - Procesador de grano fino (operaciones elementales)
 - Procesador de grano grueso (operaciones complejas)
- Si existe algún mecanismo global de control
- Tipo de funcionamiento:
 - Síncrono (simultáneamente)
 - Asíncrono (no simultáneamente)
- Tipo de comunicación entre los procesadores:
 - Memoria compartida
 - Mensajes en una red de interconexión

Según la **clasificación de Hwang-Briggs**, tendríamos como aproximación a computadores paralelos:

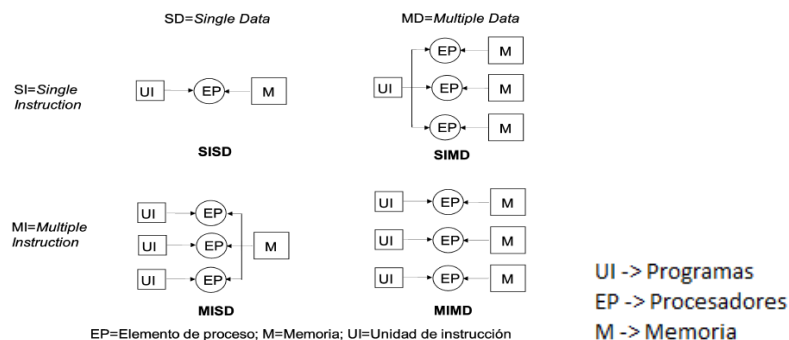
- **Computadores pipeline**
- **Computadores matriciales**
- **Sistemas multiprocesador**

Con lo que tendríamos:

- **Computadores basados en paralelismo temporal:** solapan varias instrucciones en el mismo instante de tiempo y misma unidad funcional
- **Computadores basados en paralelismo espacial:** son los asíncronos y síncronos

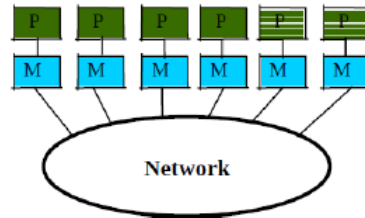
Según la **clasificación de Flynn**, tendríamos estos computadores paralelos:

- **Single Instruction Single Data (SISD):**
- **Single Instruction Multiple Data (SIMD):** el más parecido a CUDA
- **Multiple Instruction Single Data (MISD):**
- **Multiple Instruction Multiple Data (MIMD):**
- **Single Program Multiple Data (SPMD):** usado en CUDA (un único programa que se ejecuta en múltiples hilos de un mismo bloque, que acceden a diferentes flujos de datos.)

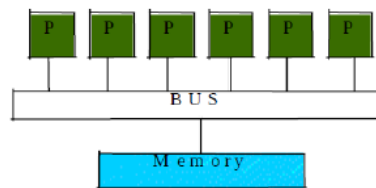


Tipos de memorias dentro de la computación en paralelo:

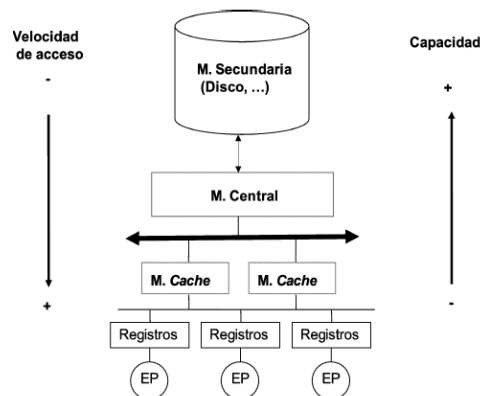
- **Memoria distribuida:** cada procesador tiene su propia memoria local. Se utiliza el paso de mensajes en una red de interconexión para la comunicación entre los procesadores



- **Memoria compartida:** todos los procesadores tienen una única memoria local. Se utiliza el paso de mensajes en una red de interconexión para la comunicación entre los procesadores

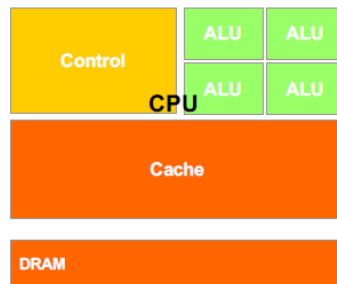


Se puede establecer esta **jerarquía de memorias**



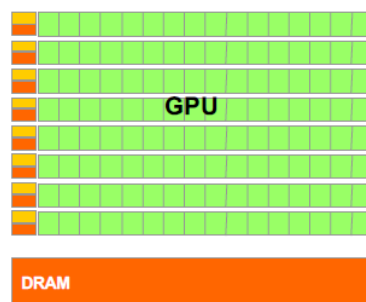
Al principio, se buscaba aumentar la velocidad de las **CPUs**, pero esto daba problemas con la disipación del calor y el consumo de energía, por lo que se comenzó a aumentar el número de núcleos de cada CPU. Cada núcleo tiene una arquitectura x86 (32 bits) y ejecutan código secuencialmente. Las CPUs usan más las memorias caché, ejecutan instrucciones complejas y la latencia (tiempos de acceso a la memoria) tiene una gran importancia

CPU (multi-core): pocos núcleos – instrucciones complejas



En cambio, las **GPUs** tienen núcleos más pequeños (sencillos) pero se duplican en cada generación. Las GPUs ejecutan código de forma paralela y cada núcleo ejecuta multitud de hilos, con lo que el control y la caché se comparte entre varios núcleos. Las GPUs acceden a direcciones de memoria contigua y la latencia no tiene gran importancia ya que la GPU tiene un uso eficiente del ancho de banda de la memoria

GPU (many-core): muchos núcleos – instrucciones sencillas



Una **tarjeta gráfica** está formada por:

- **GPU:** dedicada al procesamiento de los gráficos u operaciones de coma flotante
- **Memoria:** almacena los resultados intermedios de las operaciones y las texturas (si tienes más memoria, quiere decir que puedes almacenar texturas de mayor resolución, lo que conlleva a una mayor calidad de imagen)

$\text{ancho del bus de datos de la memoria} * \text{velocidad de reloj de la memoria} = \text{ancho de banda de la tarjeta (bandwidth)}$

- **RAMDAC:** convierte la señal digital del ordenador a señal analógica para que pueda ser interpretada por el monitor
- **Disipador:** para enfriar la GPU
- **Ventilador:** para enfriar la tarjeta gráfica
- **Alimentación:** dar corriente a la tarjeta gráfica

Historia de las GPUs (cada punto es una generación, empezando por la 1ª generación):

1. nVidia Riva 128:
 - **Año:** 1997
 - **DirectX:** 5.0
 - **Nº Transistores:** 3.5 millones
 - **Memoria tarjeta gráfica (SGRAM):** 4 MB
2. nVidia GeForce 256:
 - **Año:** 1999
 - **DirectX:** 7.0
 - **Nº Transistores:** 23 millones
 - **Memoria GPU (GDDR):** 32 o 64 MB
 - Otras características:
 - **Procesador:** 256 bits
 - **Aparece la GPU**
 - **Soporta:** Transform & Lighting
 - **Tuberías Gráficas de Función Fija** (cauce gráfico)
3. nVidia GeForce3:
 - **Año:** 2001
 - **DirectX:** 8.0
 - **Nº Transistores:** 57 millones
 - **Memoria GPU (GDDR):** 64 o 128 MB
 - Otras características:
 - **Soporta:** Vertex Shader, Pixel Shader **y** Multisampling (antialiasing)
 - **Optimización del bus de memoria**
4. nVidia GeForce FX 5200:
 - **Año:** 2003
 - **DirectX:** 9.0 o 9.0b
 - **Nº Transistores:** 125 millones
 - **Memoria GPU (GDDR2):** 128 o 256 MB
 - Otras características:
 - **Soporta:** Vertex Shader 2.0+ **y** Pixel Shader 2.0+
 - **Codificador de Tv integrado**
 - **Se implementa una mejora en la comprensión de texturas y técnicas-Z**
 - **Hasta 16 texturas por pixel**
5. nVidia GTX 7800:
 - **Año:** 2005
 - **DirectX:** 9.0c
 - **Nº Transistores:** 302 millones (G70) o 281 millones (G71)
 - **Memoria GPU (GDDR3):** 256 MB
 - Otras características:

- **Tipo de Bus de Transferencia:** PCI-Express & AGP
- **Arquitectura de 24 pixel pipelines paralelos**
- **Arquitectura de 7 vertex pipelines paralelos**

6. nVidia ATI Xenos (primera GPU unificada e incorporada en la XBOX):

- **Año:** 2005
- **DirectX:** 9.0c
- **Nº Transistores:** 337 millones
- **Memoria GPU** (GDDR3): 512 MB
- Formada por:
 - **3 motores SIMD**
 - **16 procesadores con 5 ALUs cada uno** (4 vectoriales y 1 escalar)
 - **240 Stream Processors (SPs) de 32 bits**
 - **Framebuffer a 500 MHz** (256 GB/s)
 - **8 unidades de renderizado (ROPs)**

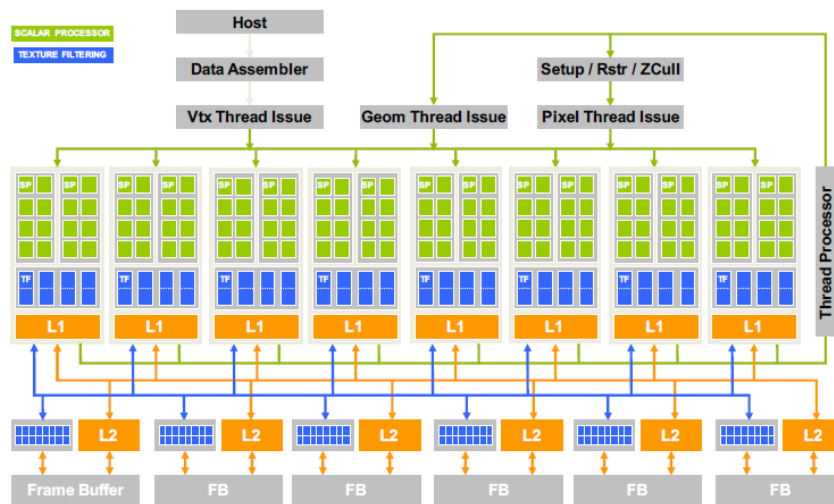
7. nVidia GeForce GTX 8800 (GPU unificada en gráficos y cómputo):

- **Año:** 2006
- **DirectX:** 10.0
- **Nº Transistores:** 681 millones
- **Memoria GPU** (GDDR3): 768 MB
- Otras características:
 - **Tecnología de fabricación:** 80nm
 - **Arquitectura unificada**
 - **Bus:** 384 bits

8. nVidia GeForce 550 GTX Ti:

- **Año:** 2010
- **DirectX:** 11.0
- **Nº Transistores:** 3000 millones
- **Memoria GPU** (GDDR5): 1024 MB
- Otras características:
 - **Tecnología de fabricación:** 40nm
 - **Soporta:** Shader Model 5.0, OpenGL 4.1, OpenCL 1.0 y CUDA
- Formada por:
 - **Núcleos CUDA:** 192
 - **Gráficos:** 900 MHz
 - **Reloj del procesador:** 1800 MHz
 - **Velocidad memoria:** 4100 MHz
 - **Interfaz memoria:** 192 bits
 - **Ancho de banda memoria:** 98.4 GB/s

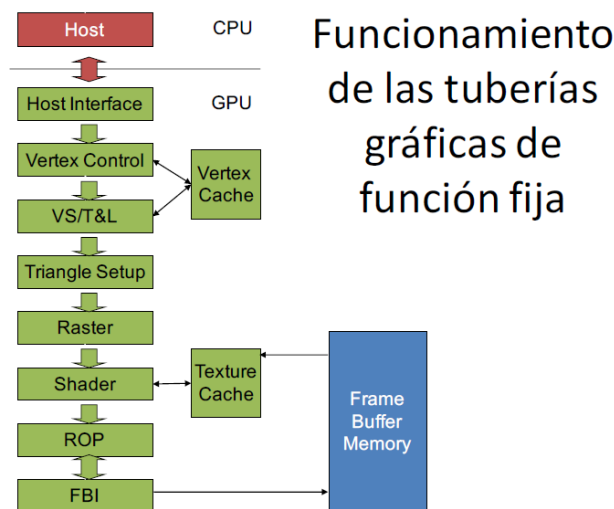
Tubería Gráfica Unificada GeForce GTX 8800



El funcionamiento de las tuberías gráficas de función fija (cauce gráfico), según las diapositivas, es el siguiente:

- CPU
 - **Host:** es la CPU que se comunica con la GPU mediante una API
- GPU
 - **Host Interface:** es el que hace la comunicación entre la GPU y CPU. Se transmiten comandos (mediante llamadas a la API) y datos (mediante un Accesos Directo a Memoria (DMA) especializado)
 - **Vertex Control:** los objetos se representan como colecciones de triángulos, con lo que al recibirse estos triángulos parametrizados de la CPU, se convierten a un formato adecuado y, a continuación, se colocan en el caché de vértices
 - **Vertex Caché:** almacena estos triángulos y vértices (es lo que supongo)
 - **VS/T&L (Vertex Shader / Transform & Lighting):** transforma los vértices y les asigna valores (colores, coordenadas de texturas, tangentes, etc). Cuando asigna colores, no se aplica este cambio en los triángulos hasta más tarde. El sombreado es realizado por el hardware de sombreado de pixeles.
 - **Triangle Setup:** crea las ecuaciones necesarias para las aristas. Estas ecuaciones son usadas para interpolar colores y otra operación pervértice (como coordenadas de las texturas) a los largo de los pixeles en contacto con el triángulo
 - **Raster:** determina qué píxeles están contenidos en cada triángulo. Con lo que, para cada pixel, interpola los valores perpixel necesarios para el sombreado, incluyendo posición, color y textura que será pintado en el pixel

- **Shader:** determina el color final de cada pixel, combinando estas técnicas:
 - Interpolación de vértices de colores
 - Mapeo de texturas
 - Iluminación
 - Reflejos
 - Otras más
 También se aplican muchos efectos en esta fase
- **Textura Caché:**
- **ROP:** realiza el rasterizado final (conversión de información vectorial a pixeles), es decir:
 - Mezcla el color de objetos superpuestos o adyacentes y efectos
 - Determina la visibilidad de un pixel desde un determinado punto de vista
- **FBI (Interfaz del Buffer de Memoria):** maneja la lectura y la escritura del Frame Buffer. Para ello requiere un gran ancho de banda, el cual se alcanza mediante:
 - Diseños especiales de memoria
 - Manejo de múltiples canales que acceden a distintos bancos de memoria de manera simultánea



El **funcionamiento de las tuberías gráficas de función fija (cauce gráfico)**, según mi explicación para entenderlo guay, es el siguiente:

- CPU
 - **Host:** es la CPU que se comunica con la GPU mediante una API
- GPU
 - **Host Interface:** permite comunicar la GPU con la CPU y, así, recibir los triángulos (datos) y comandos procedentes de la CPU.
 - **Vertex Control:** transforma los triángulos pasados por la CPU, para que puedan ser usados en las siguientes etapas

- **Vertex Caché:** estos triángulos transformados van almacenándose en esta memoria y, posteriormente, el VS/T&L los va cogiendo
- **VS/T&L (Vertex Shader / Transform & Lighting):** coge esos triángulos y obtiene las coordenadas de las esquinas, es decir, obtiene los vértices. A su vez, asigna valores a esos vértices (como el color, la textura, etc) pero solo se asignan, no se aplican
- **Triangle Setup:** mediante esas coordenadas, obtiene las ecuaciones de las aristas para interpolar colores
- **Raster:** busca los píxeles que tiene cada triángulo y asigna el color, textura, etc, correspondiente a cada uno de esos píxeles
- **Shader:** aplica los valores finales de cada píxel (usando las técnicas de: interpolación de vértices de colores, mapeo de texturas, iluminación, reflejos, etc)
- **ROP:** busca los píxeles que tiene cada triángulo y mira si hay píxeles superpuestos para así aplicar efectos de suavizado, etc
- **FBI (Interfaz del Buffer de Memoria):** maneja la lectura y escritura al frame buffer, por lo que se necesita un gran ancho de banda

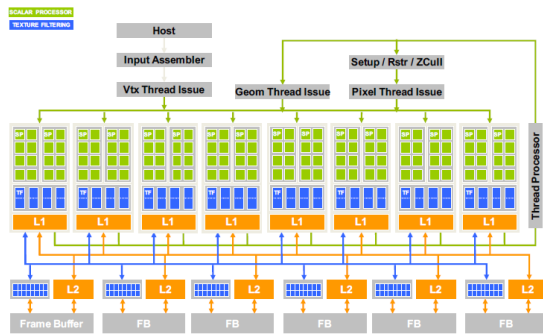
Esta arquitectura tiene una serie de **inconvenientes**:

- Se deben realizar los programas mediante las APIs gráficas
- El direccionamiento está limitado por el tamaño/dimensión de la textura
- Las capacidades de los shader están limitadas por el acceso a memoria
- En las instrucciones faltan los enteros y las operaciones sobre bits
- Las comunicaciones están limitadas por los píxeles y los accesos raros

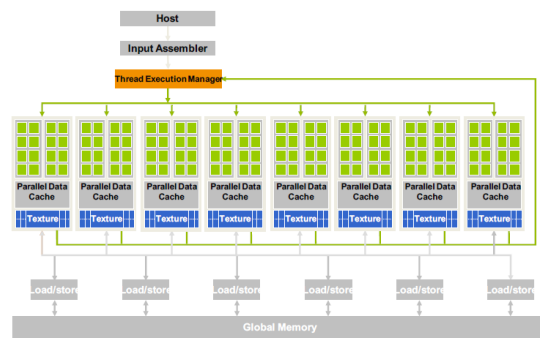
Compute Unified Device Architecture (**CUDA**), es un dispositivo de cómputo (modelo de programación) que usa lotes de hilos en paralelo mediante un coprocesador de la CPU (GPU) que tiene su propia DRAM (memoria dinámica de acceso aleatorio). Las partes paralelas se expresan mediante núcleos (kernels) y se ejecutan en múltiples hilos. Estos hilos de la GPU son muy ligeros por lo que tienen una escasa sobrecarga para su creación, y se necesitan miles de ellos para operar al máximo rendimiento

Tiene un software específico para su uso (drivers, lenguajes, herramientas, etc). Para la carga de programas en la GPU, se necesita de un driver el cual es independiente de las librerías gráficas, está optimizado, comparte datos con objetos OpenGL y maneja la memoria de la GPU de manera explícita

G80 modo Gráfico

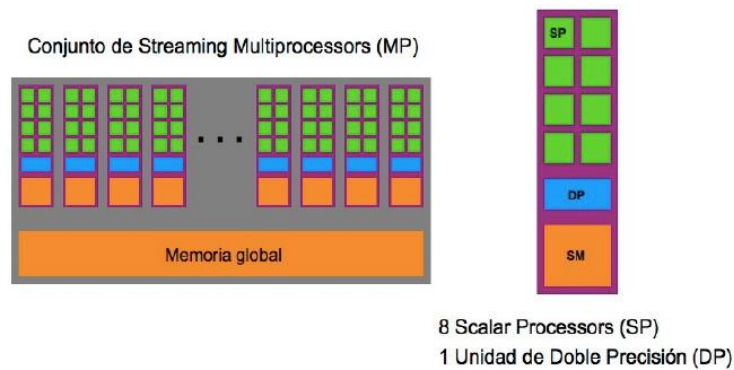


G80 modo CUDA



Arquitectura CUDA:

- 1 String Multiprocessors (SM) tiene **1024 hilos** pero solo puede **correr 768 hilos** como **max**
- 1 Scalar Processors (SP) puede **correr 512 hilos** como **max**
- 1 SM contiene **8 SP**
- 1 SP solo puede **correr una cosa** (una matriz)
- 1 hilo solo puede **calcular una posición de la matriz** (x, y). Por lo que el **tamaño de la matriz está limitado por el número de hilos** permitidos en un bloque
- A la hora de **ejecutarse los hilos**, estos **se agrupan en 32 hilos**, o **lo que es lo mismo, 1 WARP** (unidad de medida de hilos en CUDA). Por tanto, **1 WARP = 32 hilos**. Para la **ejecución de los WARPs**, se sigue una **planificación priorizada** y **todos los hilos de un mismo WARP, ejecutan la misma instrucción**
- Los **accesos a memoria** se hacen **en bloques de 16 hilos** cada bloque ya que la **memoria tiene un ancho de banda** que solo permite trabajar con **16 hilos** de forma paralela. Por tanto, se necesitará **2 accesos a memoria para completar 1 WARP**
- 1 WARP ejecuta **1 Instrucción**
- 1 WARP se ejecuta en **4 ciclos máquina** (por tanto, **1 instrucción también**)
- 1 SM tiene **24 WARPs**
- 1 SP tiene **16 WARPs**
- 1 SM tiene **2 SFU (Special Function Units) ¿?**
- **TPC**: es la unión de 2 SM junto con la **TEX**
- **TEX**: es la memoria de textura
- **DP**: indica cómo se va a operar, en Double Precision
- **SPA (Streaming Processor Array)**: es un vector de TPCs
- **MP ¿?**
- Tiene **346.5 GB/s de ancho de banda** como **máx**, la cual trabaja con **16 hilos** de forma paralela (lo he puesto en el punto anterior también). Aunque se trabaja con **86.4GB/s de ancho de banda**



- Desde el **punto de vista Software** (programar):
 - Se puede establecer el término **“bloque”** que se refiere a un conjunto de hilos (el número de hilos lo establece el programador ya que el “bloque” es una unidad software mientras que el resto de términos son limitaciones hardware del propio CUDA).

Cada grid puede ser organizado en:

- **1 Dimensión (1D)**: la coordenada (x) indica la posición de un grid
- **2 Dimensiones (2D)**: la coordenada (x, y) indica la posición de un grid

Cada bloque puede ser organizado en:

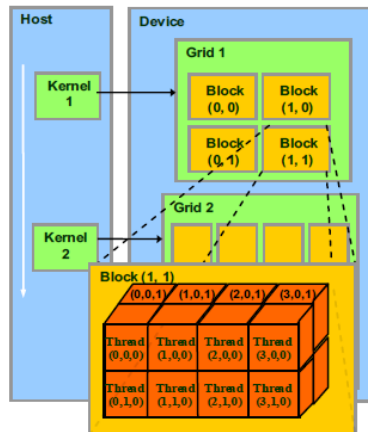
- **1 Dimensión (1D)**: la coordenada (x) indica la posición de un bloque
- **2 Dimensiones (2D)**: la coordenada (x, y) indica la posición de un bloque
- **3 Dimensiones (3D)**: la coordenada (x, y, z) indica la posición de un bloque. Aunque la ‘z’ siempre es 1, por eso se dice que un bloque solo puede tener 2 Dimensiones

Cada hilo de cada bloque pueden ser organizados en:

- **1 Dimensión (1D)**: la coordenada (x) indica la posición de un hilo
- **2 Dimensiones (2D)**: la coordenada (x, y) indica la posición de un hilo
- **3 Dimensiones (3D)**: la coordenada (x, y, z) indica la posición de un hilo

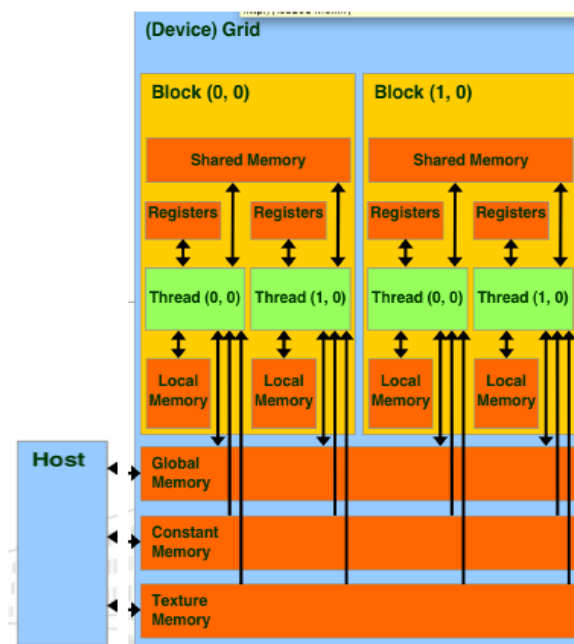
Cada hilo tiene su ID dentro del bloque (según la dimensión, realizas este cálculo pero para cada coordenada):

$$(\text{blockIdx} * \text{blockDim}) + \text{threadIdx}$$



- Tenemos el “**kernel**”, el cual es una función que se ejecutará en N hilos definidos por el “**grid**”. Yo entiendo que el “**kernel**” es la función que va a ejecutar cada hilo de la GPU. Como CUDA es **SPMD** (un único programa que se ejecuta en múltiples hilos de un mismo bloque, que acceden a diferentes flujos de datos), **los hilos de diferentes bloques no cooperan**, por tanto, **cada bloque se puede ejecutar en cualquier orden** relativo a otros bloques.
- Tenemos el “**grid**”, el cual define el nº de bloques y el nº de hilos para cada bloque. **Cada “kernel” tiene su “grid”**, por tanto, un “grid” se crea cada vez que se inicializa un “kernel”
- Tipos de memoria (de mayor latencia a menor latencia. Latencia es el tiempo de acceso a la memoria):
 - **Memoria global:**
 - **Todos los hilos y la CPU** pueden **R/W**
 - Es **offchip** (fuera del chip)
 - **No es cacheada**
 - Tamaño: **768-1024 MB**
 - **Memoria local:**
 - Está dentro de la memoria global, pero hace de memoria virtual privada para **cada hilo** que pueden **R/W**
 - Es **onchip** (dentro del chip)
 - **Gestionada por el compilador**
 - **No es cacheada**
 - Tamaño: **16 KB**
 - **Memoria constante:**
 - **Todos los hilos** pueden **R** y la **CPU** puede **R/W**
 - Es **offchip**
 - **Es cacheada**
 - Tamaño: **64 KB** (es decir, 8 KB por TPC (Texture Processor Cluster))
 - **Puede cargarse en el caché del SM** para acelerar las transferencias

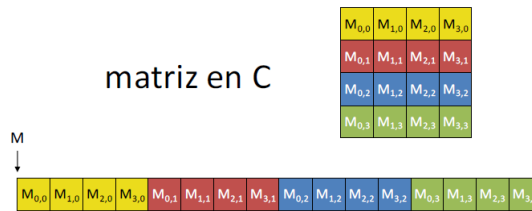
- **Memoria de texturas:**
 - Muy parecida a la memoria constante
 - **Todos los hilos** y la **CPU** pueden **R**
 - Tamaño: **64 KB** (es decir, 8 KB por TCP)
- **Memoria compartida:**
 - **Todos los hilos** de un **mismo SM** pueden **R/W**
 - Es **onchip**
 - Tamaño: **16 KB** por cada SM
 - El **tiempo de vida** de esta memoria, **es igual al tiempo de vida del bloque**
 - Para optimizar la ejecución en paralelo, se dividen los datos y se almacenan en esta memoria. Una vez terminado, se pasa el resultado a memoria global
- **Registro:**
 - **Cada hilo** tiene un registro de forma privada para **R/W**
 - Es **onchip**
 - **Gestionada por el compilador**
 - Tamaño: **8 KB** por multiprocesador. Se distribuyen **8192 registros entre todos los bloques** de cada **SM**. Con lo que, **cada hilo del mismo bloque** solo puede **acceder a sus registros**, y los registros de un bloque no son accesibles por otros bloques



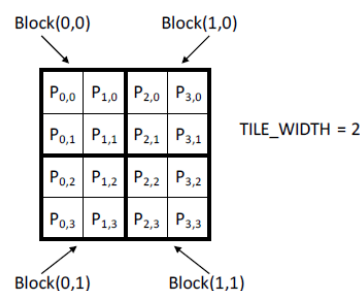
C para CUDA:

- El **código secuencial** se ejecuta en la **CPU**
 - Los **vectores** y las **matrices** se crean en la **CPU** y se cargan en la **memoria global** (nunca en la memoria de la GPU).

- Una **matriz** tiene **tamaño** de **WIDTH * WIDTH** y se almacena, en orden, en un **vector**



Para realizar algún cálculo demasiado grande con matrices (p.j multiplicación de matrices), realizamos una **teselación**, es decir, dividimos la matriz en “bloques” y realizamos los cálculos de cada “bloque”. Cada “**bloque**” tendrá un **tamaño** de **TILE_WIDTH * TILE_WIDTH**










Gracias a la teselación, se **reducen** los **accesos de memoria global** por un **factor de TILE_WIDTH**

- El **código paralelo** se ejecuta en la **GPU** (hilos) y
 - Las **variables** siempre en **coma flotante** (float) (enteros también pero es menos óptimo)
 - Para conocer las características de la tarjeta, se usa **Compute Capability** (capacidades de computación), para CUDA: **1.1 o superior**
- Compilación:
 1. Partimos del archivo “nombreArchivo.cu”, el cual debe compilarse con **NVCC**. Cualquier ejecutable CUDA requiere dos librerías dinámicas se necesitan las librerías **cuda** y **cudart**
 2. La ejecución se separa en dos: parte en la CPU y parte en la GPU
 3. Para la CPU
 - 3.1. Lo pasamos a “nombreArchivo.cpp”
 - 3.2. Se compila mediante gcc/cl
 4. Para la GPU

Preprocesa la entra el compilador nvopencc que emite un lenguaje en PTX.

 - 4.1. Lo pasamos a “nombreArchivo.s”
 - 4.2. Se compila mediante Optimized Code Generation (OCG)
 - 4.3. Lo pasamos a “nombreArchivo.sass”
- Código:

- **cudaMalloc()**: asigna un objeto en la memoria global. Necesita:
 - Puntero al objeto a asignar
 - Tamaño de ese objeto
- **cudaFree()**: libero un objeto de la memoria global. Necesita:
 - Puntero al objeto a liberar
- **cudaMemcpy()**: transfiere datos de forma asíncrona. Necesita:
 - Puntero al destino
 - Puntero al origen
 - Nº Bytes a copiar
 - Tipo de transferencia (Host -> CPU / Device -> GPU):
 -  Host a Host
 -  Host a Device
 -  Device a Host
 -  Device a Device
- Declaración de funciones:
 - **__device__ float nombreFunc()**: se ejecuta en Device.
Limitaciones:
 -  No puede ser recursiva
 -  No se pueden declarar variables estáticas
 -  No se puede tener un número variable de argumentos
 - **__global__ void nombreFunc()**: se ejecuta en Device y Host
 - **__host__ float nombreFunc()**: se ejecuta en Host
- Declaración de variables:
 - **__device__ __local__ int variableLocal**: aquí van los arrays
 - **__device__ __shared__ int variableCompartida**
 - **__device__ int variableGlobal**
 - **__device__ __constant__ int variableConstante**
 - **int variableRegistro**
 - //los **punteros** da igual donde se utilicen (CPU o GPU), pero solo pueden **apuntar a memoria situada o declarada en la memoria global**

Declaración de variable	Memoria	Ambiente	Vida
__device__ __local__ int LocalVar;	local	thread	hilo
__device__ __shared__ int SharedVar;	compartida	block	bloque
__device__ int GlobalVar;	global	cuadrícula	aplicación
__device__ __constant__ int ConstantVar;	constante	cuadrícula	aplicación

- **DimGrid(x, y)**: establece el nº de bloques. En realidad, tiene 3 Dimensiones (x, y, z) pero la 'z' siempre es 1
- **DimBlock(x, y, z)**: establece el nº de hilos por bloque
- **gridDim**: indica el número de bloques
- **blockDim**: indica el número de hilos en cada bloque

- **blockIdx**: indica el índice de un bloque dentro de la cuadrícula (dependiendo de la dimensión tendrá (x) o (x, y))
- **threadIdx**: indica el índice de un hilo dentro del bloque (dependiendo de la dimensión tendrá (x), (x, y) o (x, y, z))
- **void __syncthreads()**: sincroniza todos los hilos de un bloque. Por lo que, hasta que no se hayan sincronizado todos, la ejecución no continua. Con esto evitamos problemas al usar memoria global o compartida

Tiempos de acceso según el tipo de instrucción:

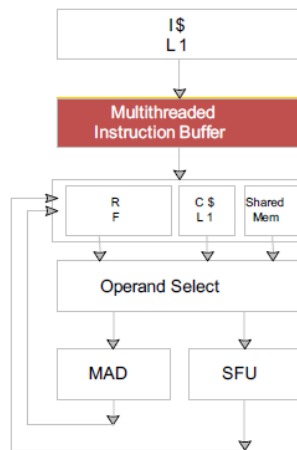
- Operaciones aritméticas (operaciones sobre coma flotante de 16 bits; suma y operaciones a nivel de bit sobre enteros; y máx, mín o conversión del tipo de dato): **4 ciclos de reloj**
- Operaciones logarítmicas o de cálculo de inversa: **16 ciclos de reloj**
- La multiplicación de 32 bits: **16 ciclos de reloj**
- Accesos a memoria: **4 ciclos de reloj**
- Accesos a memoria global: **4 ciclos de reloj + (entre 400 y 600 ciclos de reloj)**
- Sincronización: **4 ciclos de reloj + tiempo de espera a la sincronización de todos los hilos**
- Ejecutar 1 WARP: **4 ciclos de reloj**

El **ciclo de vida** de un hilo en **hardware**:

1. Se **lanza** la **cuadrícula** con la planificación de bloques e hilos por bloque **en la SPA**
2. Estos **bloques** se **distribuyen** secuencialmente **por los SM** (al ser posible, más de un bloque por SM)
3. **Cada SM lanza** unos **WARPs**
4. Los **SM planifican y manejan la ejecución** de estos **WARPs** (dependerá de si su siguiente instrucción y sus operandos, están listos. Y si es así, dependerá de ciertas políticas de priorización):
 - 4.1. **I\$ L1 Caché**: almacena las instrucciones
 - 4.2. **Buffer de Instrucciones**: cada ciclo, se coge una instrucción y se pasa a este buffer. Se ejecuta un “ready-to-go” de un WARP en cada ciclo. Tras esto, se ejecuta un WARP basándose en un (“todos-contra-todos”/edad). Finalmente, el SM envía la misma instrucción a los 32 hilos del WARP
 - 4.3. **Fichero de registros (RF)**:
 - 4.4. **C\$ L1 Constantes**: se almacenan en la DRAM (memoria de video), se acceden a través de una caché y se pueden enviar simultáneamente a todos los hilos de un WARP
 - 4.5. **Memoria compartida**: aparte de lo ya sabido sobre esta memoria, como los hilos acceden simultáneamente a dicha memoria, esta se divide en **16 bancos**.

Cada **banco**, puede **servir una dirección por cada ciclo**, con lo que, una **memoria** puede **servir tantas direcciones simultáneas como bancos tenga**.

- No hay conflictos cuando:
 - Todos los hilos de medio WARP acceden a bancos diferentes
 - Todos los hilos de medio WARP acceden a la misma dirección de memoria
- Hay conflictos cuando:
 - Varios hilos del medio WARP acceden al mismo banco



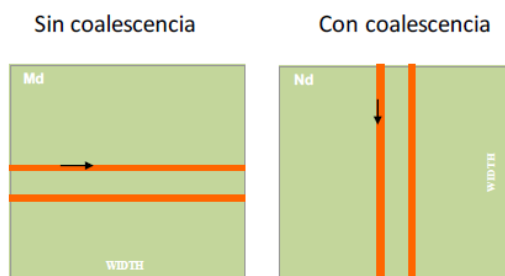
5. Cuando **terminan** los **bloques o WARPs**, se **van liberando**
6. Con lo que **SPA distribuye más bloques** para realizar el mismo proceso

//**Optimización máxima** (el mejor caso): todos los hilos corren en el mismo bloque para así usar el máximo de la memoria compartida

// **Rendimiento máximo**: todos los hilos de medio WARP, acceden a direcciones contiguas de memoria

El **compilador** puede **realizar** el **paralelismo** a nivel de Instrucción (**ILP**) o a nivel de hilo (**TLP**)

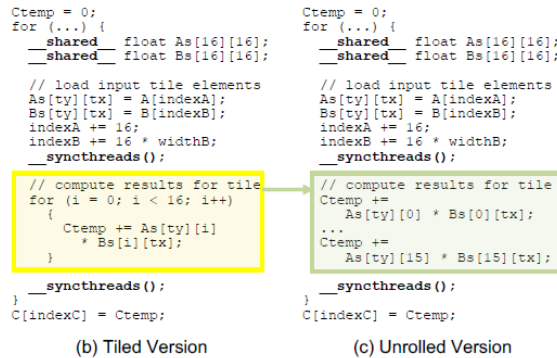
La **coalescencia** consiste en la unión de dos bloques libres adyacentes de memoria



Algoritmos fundamentales para la GPU:

- **Reducción paralela:** partiendo de un array de valores, “reducirlo” a un único valor en paralelo

Unrolling (desenrollado): consiste en coger las instrucciones condicionales y de salto (bucles) y eliminarlos y poner todas las posibles instrucciones seguidas



```

Ctemp = 0;
for (...) {
    __shared__ float As[16][16];
    __shared__ float Bs[16][16];

    // load input tile elements
    As[ty][tx] = A[indexA];
    Bs[ty][tx] = B[indexB];
    indexA += 16;
    indexB += 16 * widthB;
    __syncthreads();

    // compute results for tile
    for (i = 0; i < 16; i++)
    {
        Ctemp += As[ty][i]
                * Bs[i][tx];
    }

    __syncthreads();
}
C[indexC] = Ctemp;

```

(b) Tiled Version

```

Ctemp = 0;
for (...) {
    __shared__ float As[16][16];
    __shared__ float Bs[16][16];

    // load input tile elements
    As[ty][tx] = A[indexA];
    Bs[ty][tx] = B[indexB];
    indexA += 16;
    indexB += 16 * widthB;
    __syncthreads();

    // compute results for tile
    Ctemp +=
        As[ty][0] * Bs[0][tx];
    ...
    Ctemp +=
        As[ty][15] * Bs[15][tx];

    __syncthreads();
}
C[indexC] = Ctemp;

```

(c) Unrolled Version

Los principales **factores de decremento de rendimiento** son:

- Operaciones de alta latencia
- Esperas y burbujas en la cascada
- Saturación de recursos compartidos

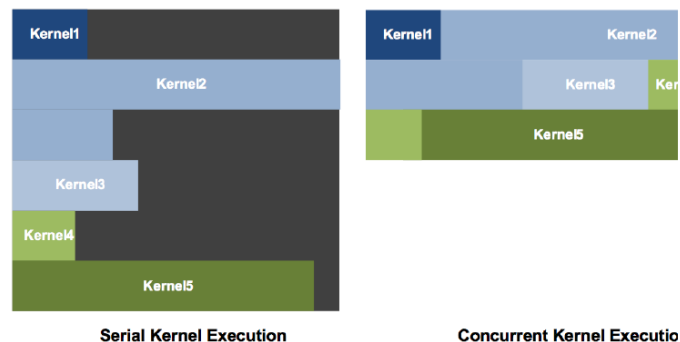
FERMI (hecho por Yisus)

Hasta finales de los 90, las tarjetas gráficas funcionaban bien y eran rápidas pero eran muy complicadas de programar ya que se programaba con las coordenadas de los triángulos directamente.

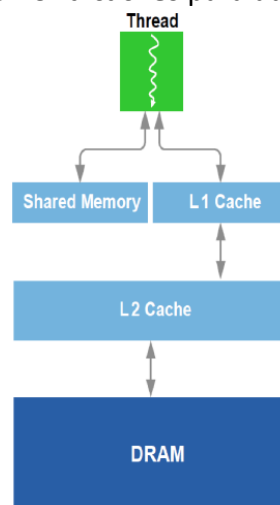
Como solución a esto, **NVIDIA crea CUDA** (que es el estilo de programación) y, crea **FERMI** (primero la G80 y después la G200)

La **G200** tiene estas **novedades** (la G80 es la que estamos viendo todo el rato):

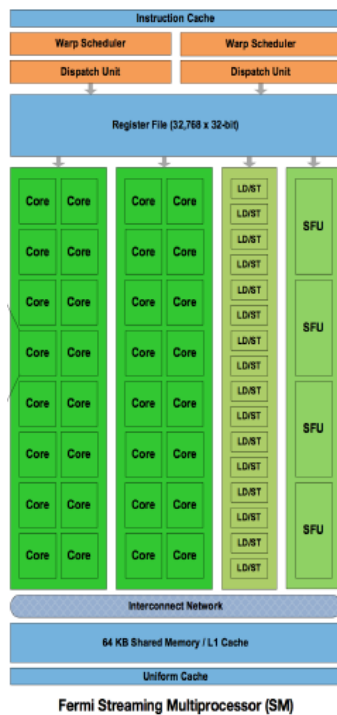
- Se implanta la 3ª generación de los SM: se introduce una nueva planificación de hilos, permitiendo por primera vez la **ejecución paralela de kernels**



- Se implanta una **mayor precisión con float**
- **Compute capability: min 2.0 y max 2.1**
- Se implanta una **nueva jerarquía de memoria**
 - La **memoria compartida** aumenta de tamaño: **entre 16 KB y 48 KB**
 - Se **implantan dos memoria cachés** para aumentar el rendimiento



- Se implanta una **nueva estructura de la tarjeta gráfica**
 - **32 SP en 1 SM**



GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

KEPLER

Corrige los errores de FERMI

- **Compute capability: min 3.0 y max 3.5**
- **192 SP cada 1 SM**
- Se añade una **memoria caché** de solo **lectura de 48KB** para aumentar el rendimiento y velocidad
- **Memoria compartida de 56KB**