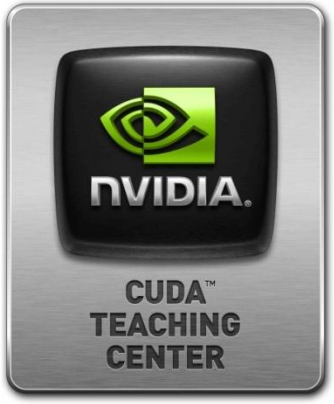
# Programación masivamente paralela en procesadores gráficos (GPUs)

E. Dufrechou, P. Ezzatti, M. Freire y M. Pedemonte



# Clase 11
# Arquitecturas post-Kepler

# Contenido

- Paralelismo dinámico
- Warp shuffle
- Arquitectura Maxwell
- Arquitectura Pascal
- Arquitectura Volta

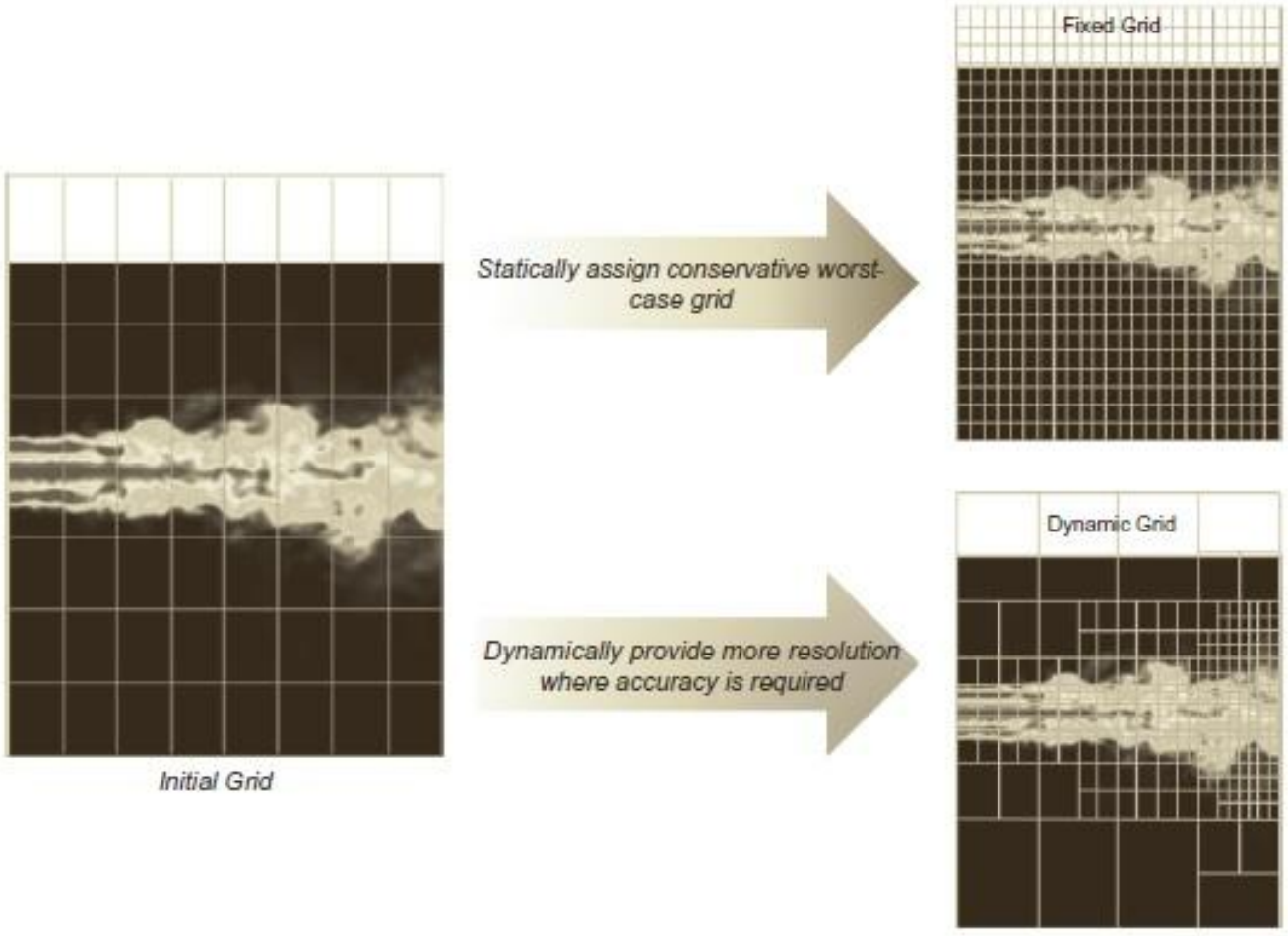
# Paralelismo Dinámico

# Paralelismo Dinámico

- El soporte para paralelismo dinámico se incorpora en la arquitectura Kepler.
- Antes de Kepler, los kernels solamente podían ser lanzado desde el host.
- El paralelismo dinámico permite que un kernel pueda lanzar otro kernel, creando una nueva grilla de hilos.
- Es ideal para problemas con:
  - Recursión
  - Loops irregulares

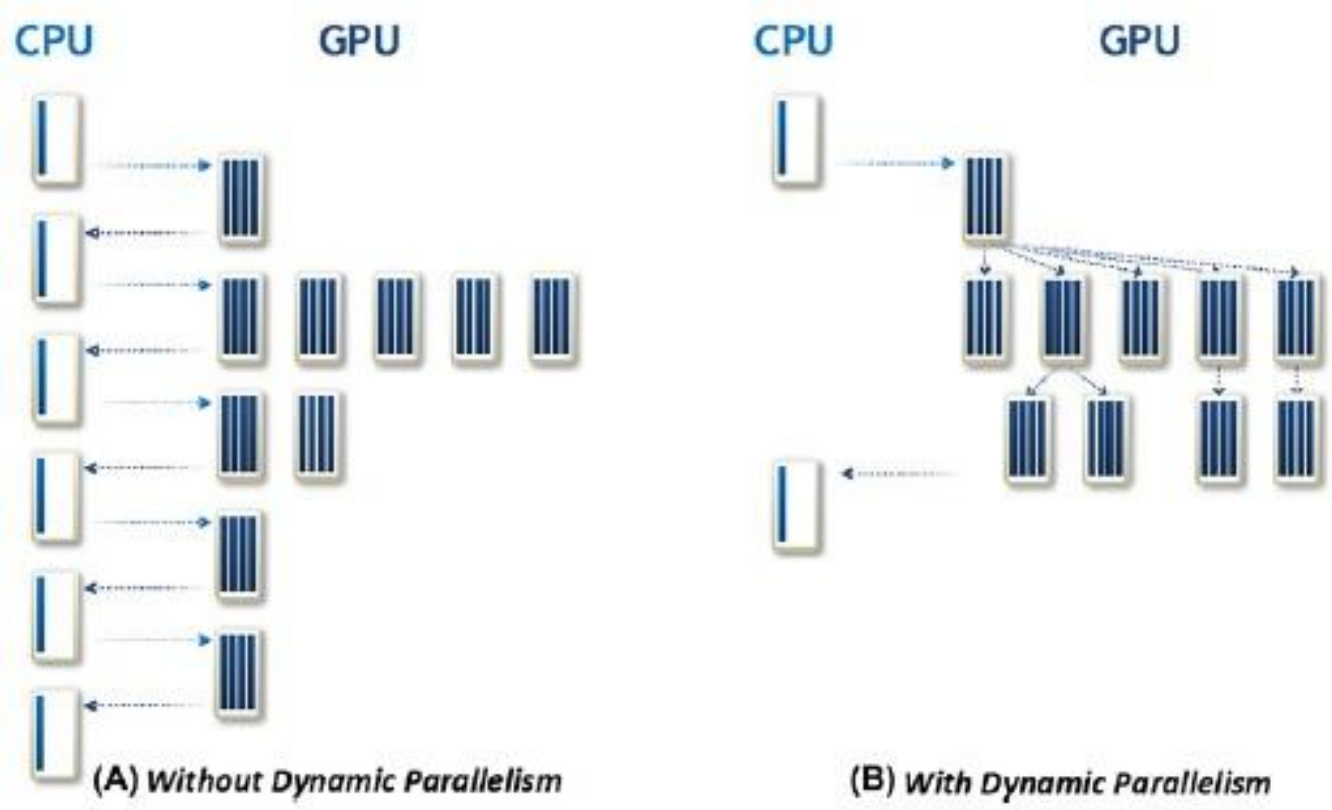
# Paralelismo Dinámico

- **El paralelismo dinámico permite que el algoritmo sea capaz de descubrir dinámicamente en dónde concentrar el trabajo.**



# Paralelismo Dinámico

- **Sin paralelismo dinámico también se pueden lograr los refinamientos pero es engorroso.**
- **Todo debe ser comunicado al host para que refine.**

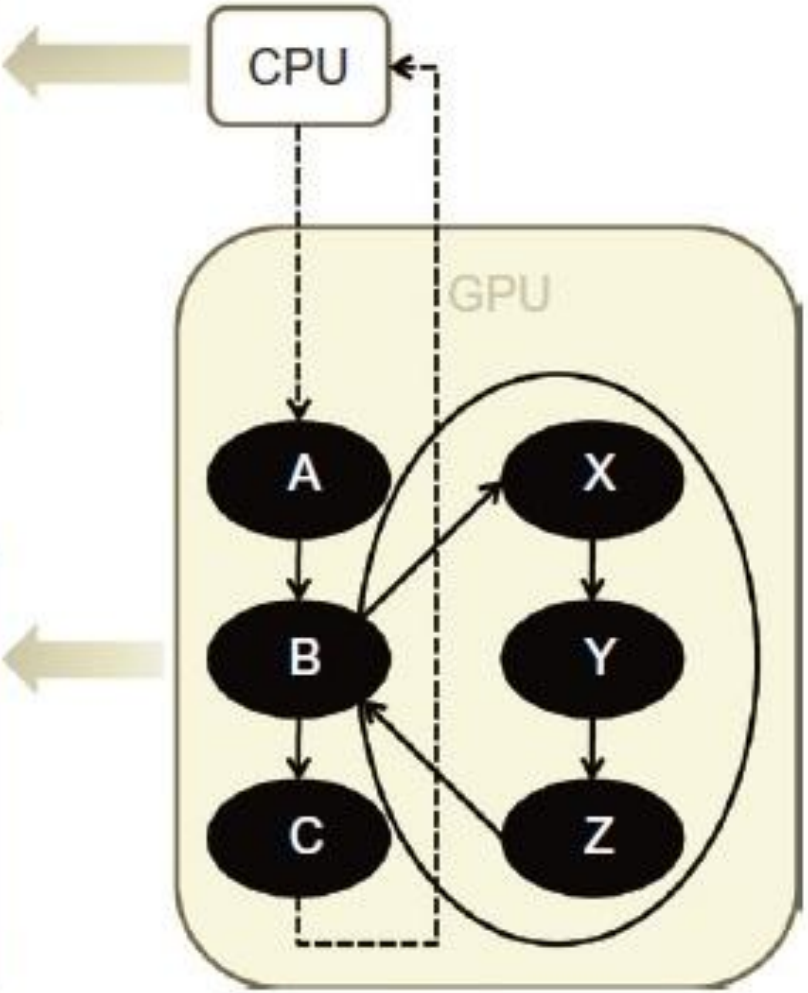


# Paralelismo Dinámico

- **La invocación es igual que en el caso de un kernel convencional.**

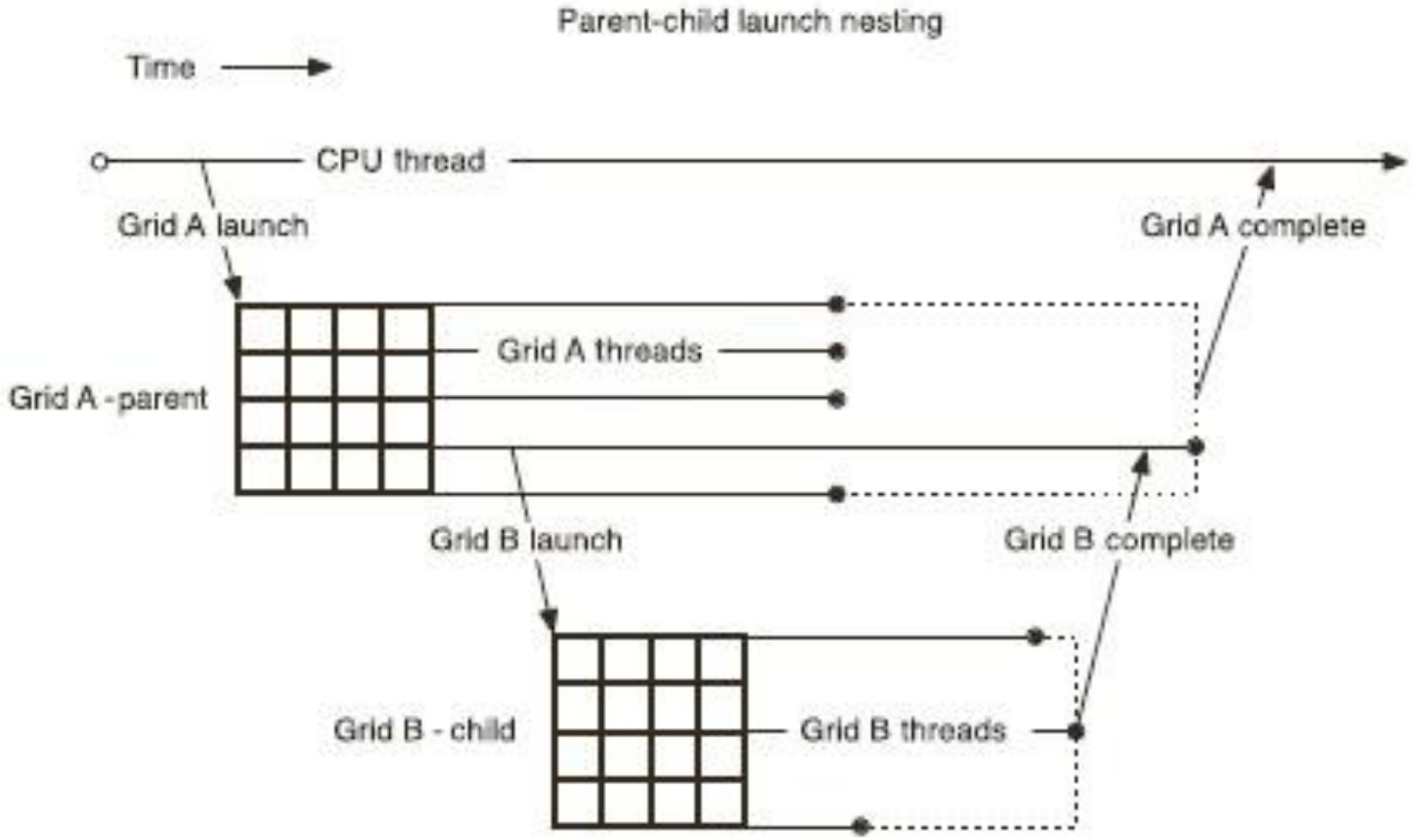
```
int main() {
    float *data;
    setup(data);

    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);

    cudaDeviceSynchronize();
    return 0;
}
```

```
__global__ void B(float *data)
{
    do_stuff(data);

    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);
    cudaDeviceSynchronize();

    do_more_stuff(data);
}
```



# Paralelismo Dinámico

- **Un hilo puede hacer la llamada.**
- **El tiempo de vida es lo que uno esperaría.**



# Paralelismo Dinámico

- **Existe coherencia en memoria global entre las grillas padres e hijos.**
- **No se puede compartir la memoria local (incluidos los registros) entre los padres e hijos.**
- **Tampoco se puede pasar la memoria compartida de padres a hijos.**
- **No se explica exactamente la ejecución pero se deduce que suspende la ejecución del padre. (en qué multiprocesador?)**

**El paralelismo dinámico no es muy usado en la práctica, pero puede servir para algunas aplicaciones específicas.**

# Warp Shuffle

# Warp Shuffle

- El soporte para operaciones del tipo warp shuffle se incorpora en la arquitectura Kepler.
- En muchos casos de programas en CUDA se necesita comunicar valores entre los hilos de un bloque.
- Por ejemplo, en el caso de las reducciones.
- La forma tradicional de hacer esa comunicación es a través del uso de shared memory.
- Desde la arquitectura Kepler, los hilos de un warp pueden leer los datos de los registros de otros hilos del warp.
- Para eso se usa las instrucciones shuffle (SHFL).

# Warp Shuffle

- Las operaciones shuffle son más rápidas que usar shared memory en forma segura (es decir incluyendo las sincronizaciones).
- Shuffle solamente involucra una operación, mientras que en shared se requieren tres operaciones: escritura, sincronización y lectura.
- Las operaciones shuffle no son más lentas que usar shared memory en forma insegura (es decir asumiendo que hay operaciones que se dan al mismo tiempo y omitiendo las sincronizaciones).
- Shuffle no requiere las sincronizaciones porque está implícito que el warp ejecuta todo en el mismo ciclo (veremos más adelante que eso ha cambiado últimamente).

# Warp Shuffle

- Las operaciones shuffle son:
  - `int __shfl(int var, int srcLane, int width=32)`
  - `int __shfl_up(int var, unsigned int delta, int width=32)`
  - `int __shfl_down(int var, unsigned int delta, int width=32)`
  - `int __shfl_xor(int var, int laneMask, int width=32)`
- El parámetro width
  - es por defecto 32
  - tiene que ser una potencia de 2 entre 2 y 32
  - permite subdividir el warp en subsecciones (si width < 32)
  - Cada subsección se considera por separado y el intercambio solo puede ser entre hilos de una misma subsección.

# Warp Shuffle

- Se usa el nombre carriles (lanes) para referirse a los hilos de un warp.
- Los carriles pueden tener como índice entre 0 y warpSize-1 (inclusive).

- __shfl(var, srcLane):
  - devuelve el valor de la variable var almacenado por el hilo cuyo carril es srcLane.
  - Esta operación sirve para hacer broadcast de valores dentro de un warp.

# Warp Shuffle

- __shfl_xor(var, laneMask):
  - devuelve el valor de la variable var almacenado por el hilo cuyo carril se calcula a partir de un XOR bit a bit (bitwise) entre el carril y laneMask.

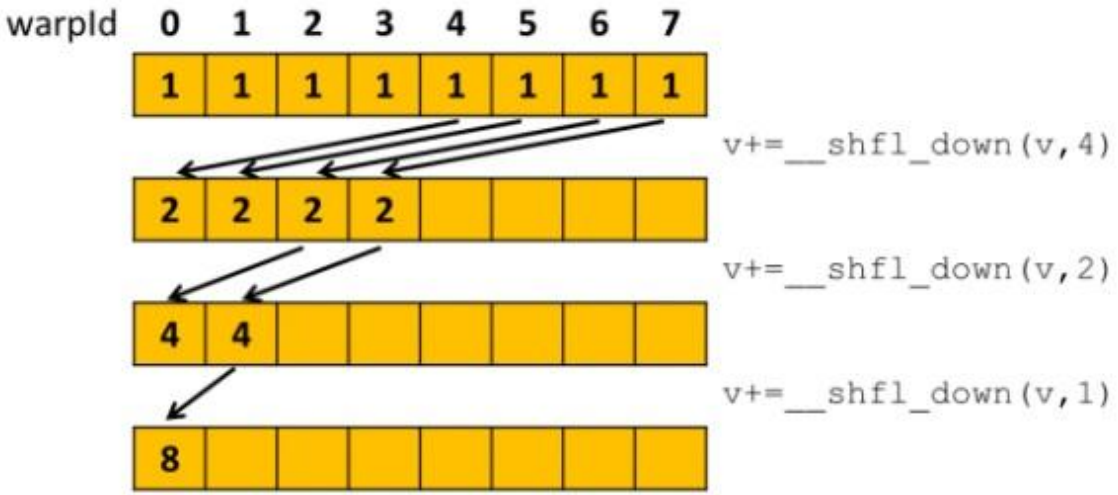
- __shfl_down(var, delta)
  - Calcula el índice del carril sumando delta al índice del que invoca y sujeto al rango (solo se activan los que dan un índice dentro del rango).
  - Sirve para copiar valores de un carril con un índice más alto que el que invoca.

# Warp Shuffle

- __shfl_up(var, delta)
  - Calcula el índice del carril restando delta al índice del que invoca y sujeto al rango (solo se activan los que dan un índice dentro del rango).
  - Sirve para copiar valores de un carril con un índice más bajo que el que invoca.

# Warp Shuffle

- **Ejemplo: Reducción de un warp**




```
__inline__ __device__
int warpReduceSum(int val) {
  for (int offset = warpSize/2; offset > 0; offset /= 2)
    val += __shfl_down(val, offset);
  return val;
}
```

# Warp Shuffle

- **Ejemplo: Reducción de bloque**

```
__inline__ __device__
int blockReduceSum(int val) {

  static __shared__ int shared[32]; // Shared mem for 32 partial sums
  int lane = threadIdx.x % warpSize;
  int wid = threadIdx.x / warpSize;

  val = warpReduceSum(val);     // Each warp performs partial reduction

  if (lane==0) shared[wid]=val; // Write reduced value to shared memory

  __syncthreads();              // Wait for all partial reductions

  //read from shared memory only if that warp existed
  val = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] : 0;

  if (wid==0) val = warpReduceSum(val); //Final reduce within first warp

  return val;
}
```

# Warp Shuffle

- Ventajas de las operaciones warp shuffle:
  - Son más rápidas que la memoria shared.
  - Reducen la cantidad de shared memory necesaria, por lo que permiten usar más.
  - En las distintas arquitecturas, a veces aumenta más la cantidad de cores que la shared disponible.

# Arquitectura Maxwell

# Arquitectura CUDA – Kepler

| | Kepler | Maxwell | Pascal | Volta |
|---|---|---|---|---|
| Transistores | 3.54 bill | 5.2 bill | 7.2 bill | 21.1 bill |
| CUDA cores | 1536 | 2048 | 2560 | 5120 |
| Core clock | 1006 Mhz | 1126 Mhz | 1607 Mhz | 1200 Mhz |
| Thermal Design Power | 195 W | 165 W | 180 W | 250 W |
| Theorical GFLOPS/s | 3090 | 4612 | 8228 | 13800 |
| GFLOPS/W | 15.85 | 27.95 | 45.71 | 55.2 |
| Memory Bandwidth (GB/sec) | 192.26 | 224 | 320 | 652.8 |

- **Maxwell: fines de 2014 (GTX 980)**
- **Pascal: abril-mayo 2016 (GTX 1080)**
- **Volta: diciembre 2017 (GTX Titan V)**

# Arquitectura Maxwell

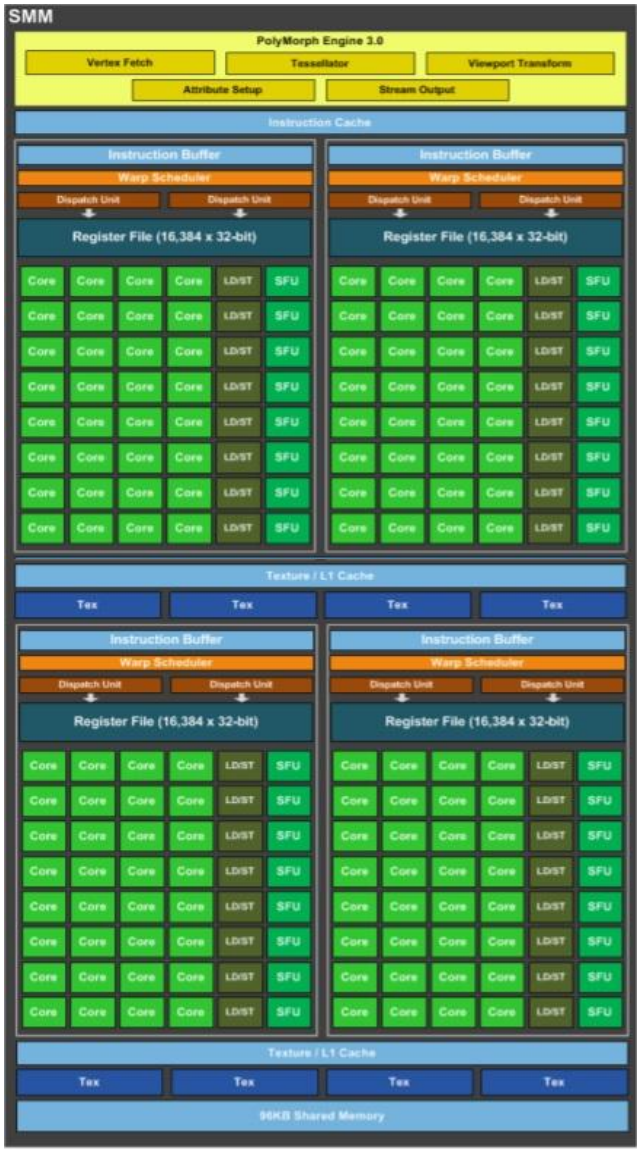
- **Sobre fines de 2014 se lanza la quinta generación de CUDA, conocida con el nombre de Maxwell.**
- **Salto en la eficiencia energética reduciendo el consumo pero aumentando la performance.**
- **La performance por watt es casi el doble que en la arquitectura Kepler.**

# Arquitectura Maxwell

- Se compone de:
  - 4 GPCs
  - 16 multiprocesadores
  - 4 controladores de memoria
- Cada multiprocesador tiene 128 CUDA cores.
- 2048 CUDA cores en total.

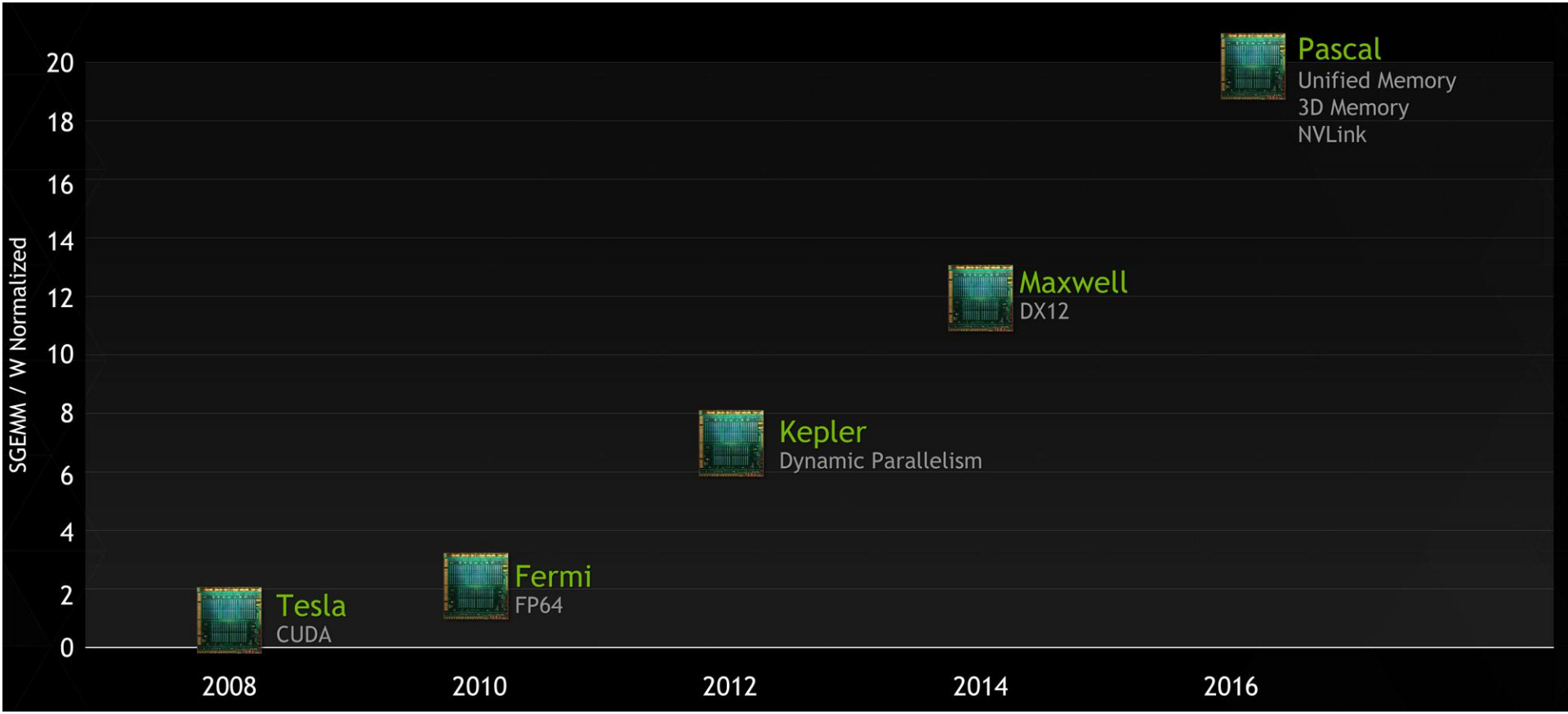
# Arquitectura Maxwell

- Los multiprocesadores están particionados en 4 bloques, cada uno con sus propios recursos independientes para la planificación de hilos y el búfer de instrucciones.
- Cada multiprocesador contiene 4 warp schedulers.
- Cada warp scheduler es capaz de despachar dos instrucciones por warp en cada ciclo de reloj.



# Arquitectura Pascal

# Arquitectura Pascal



# Arquitectura Pascal

- **En abril 2016 se lanza la sexta generación de CUDA, conocida con el nombre de Pascal.**
- **La arquitectura completa tiene 60 multiprocesadores!!!**

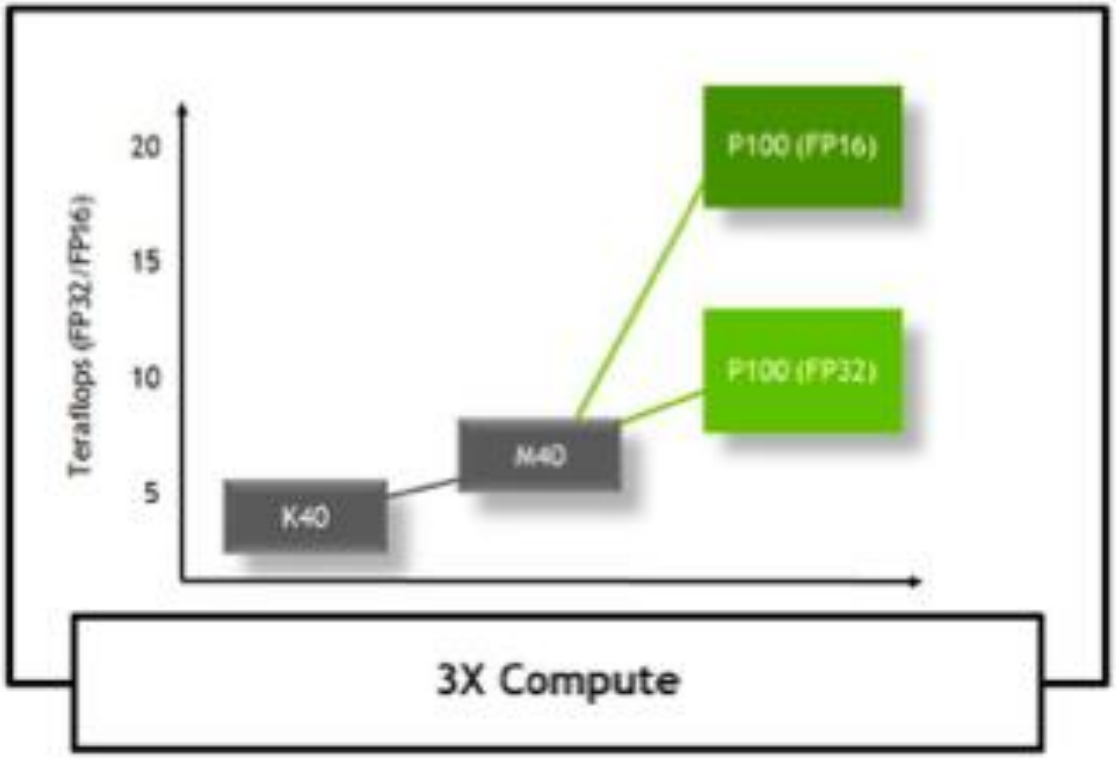
# Arquitectura Pascal

- Ejemplo de esta arquitectura es la Tesla P100.
- Existe un Servidor diseñado especialmente para Deep Learning que combina 8 Tesla P100 (Servidor DGX-1).
- Es un salto cualitativo importante
- Tiene varias mejoras con respecto a las arquitecturas previas.

# Arquitectura Pascal

- Mejoras en hardware:
  - 16nm FinFET
  - más transistores
  - más eficientes energéticamente
- Mejora comunicación CPU – GPU:
  - NVLink
  - mayor bandwidth
- Memoria unificada:
  - simplificación en la programación.
- FP16 real:
  - mejora importante en el desempeño (si puede usar FP16)

# Desempeño teórico P100




- 5.3 TFLOPS en doble precisión floating point (FP64)
- 10.6 TFLOPS en simple  precisión (FP32)
- 21.2 TFLOPS en half-precisión (FP16)

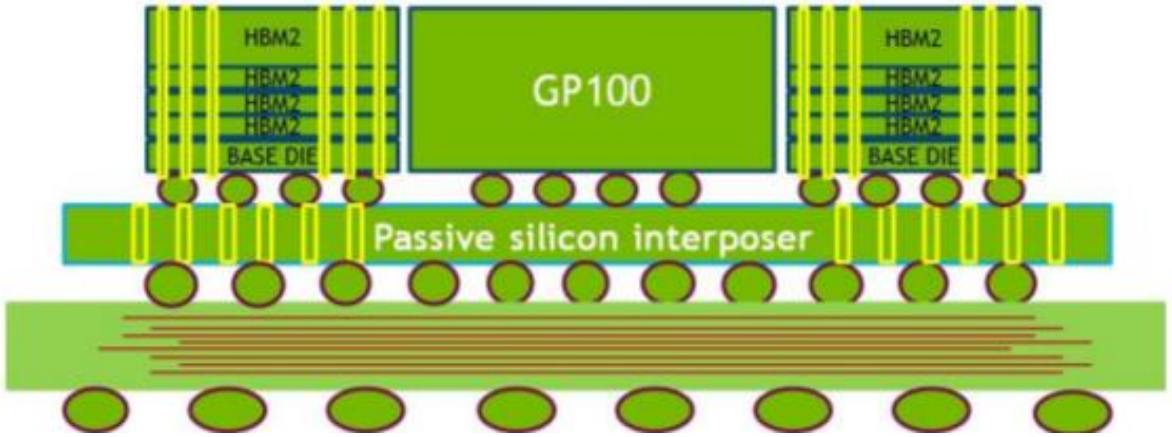
# Conexión

## NVLink

- Tecnología de Nvidia
- Ofrece transferencias de datos GPU--GPU de hasta 160 Gigabytes/segundos de bandwidth.
- En algunos equipos CPU-GPU (p.ej. IBM power 8).
- 5x cuando se lo compara con el bandwidth de PCIe Gen 3 x16.
- Comparable en velocidad con InfiniBand o 100 Gb Ethernet (alta velocidad).

# Memoria

## HBM2 High-Speed GPU Memory Architecture




- Stacked RAM: tecnología nueva 3D
- Tesla P100 primera GPU que soporta memoria HBM2.
- 3x en el bandwidth de memoria sobre la Maxwell GM200 GPU.
- Como la memoria HBM2 es stacked permite mejor diseño de HW.

# Arquitectura

**La GP100 está compuesta por (hasta):**

- **6 array of Graphics Processing Clusters (GPCs).**
- **30 Texture Processing Clusters (TPCs).**
- **60 Streaming Multiprocessors (SMs):**
  - **Cada SM tiene 64 CUDA Cores y 4 Texture Units**

**En total 3840 simple precisión CUDA Cores y 240 texture units.**

- **Cada controlador de memoria tiene 512 KB de caché L2.**
- **Cada HBM2 DRAM tiene 2 controladores.**

**En total ofrece 4096 KB de caché L2 (512 x 8).**

# Arquitectura

## Mejora importante en doble precisión:

# Arquitectura

**Mejora importante en doble precisión:**

**Cada SM ofrece 32 CUDA Cores de doble precisión.**

- **La mitad que FP32**
- **En total tiene 1920 FP64 CUDA Cores**

**El ratio entre SP vs DP es 2:1**

- **Kepler GK110 tiene un ratio 3:1 entre SP y DP.**

**Soporte IEEE 754-2008 completo, incluso fused multiply-add (FMA) y desnormalizados.**

# Arquitectura

## Soporte para FP16

- Especialmente útil para Deep Learning
  - Es de las pocas técnicas que no necesitan más precisión.
- Obviamente se reduce el uso de memoria (x2 vs FP32 y x4 vs FP64).
- También disminuyen los accesos a memoria.
- En GP100 se pueden realizar 2 operaciones de FP16 por operación en FP32.

# Servidor DGX-1

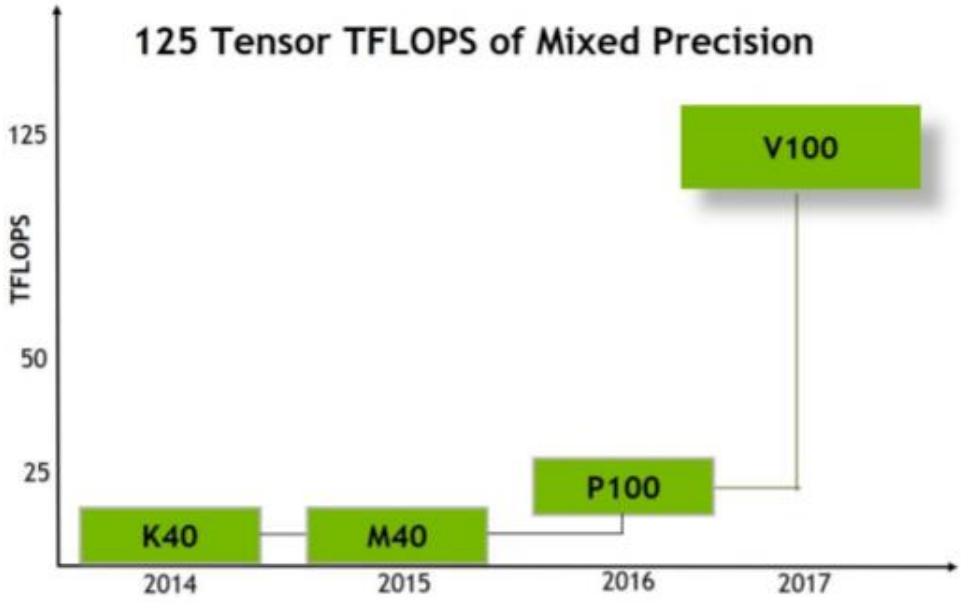
Table 4. NVIDIA DGX-1 System Specifications

| Specification | Value |
|---|---|
| GPUs | 8x Tesla P100 GPUs |
| TFLOPS | 170 (GPU FP16) + 3 (CPU FP32) |
| GPU Memory | 16 GB per GPU / 128 GB per DGX-1 Node |
| CPU | Dual 20-core Intel® Xeon® E5-2698 v4 2.2 GHz |
| NVIDIA CUDA Cores | 28,672 |
| System Memory | 512 GB 2133 MHz DDR4 LRDIMM |
| Storage | 4x 1.92TB SSD RAID 0 |
| Network | Dual 10 GbE, 4 IB EDR |
| System Weight | 134 lbs |
| System Dimensions | 866 D x 444 W x 131 H (mm) |
| Packing Dimensions | 1180 D x 730 W x 284 H (mm) |
| Power | 3200W (Max). Four 1600W load-balancing power supplies (3+1 redundant), 200-240V (ac), 10A |
| Operating Temperature Range | 10 - 35°C |

# Arquitectura Volta

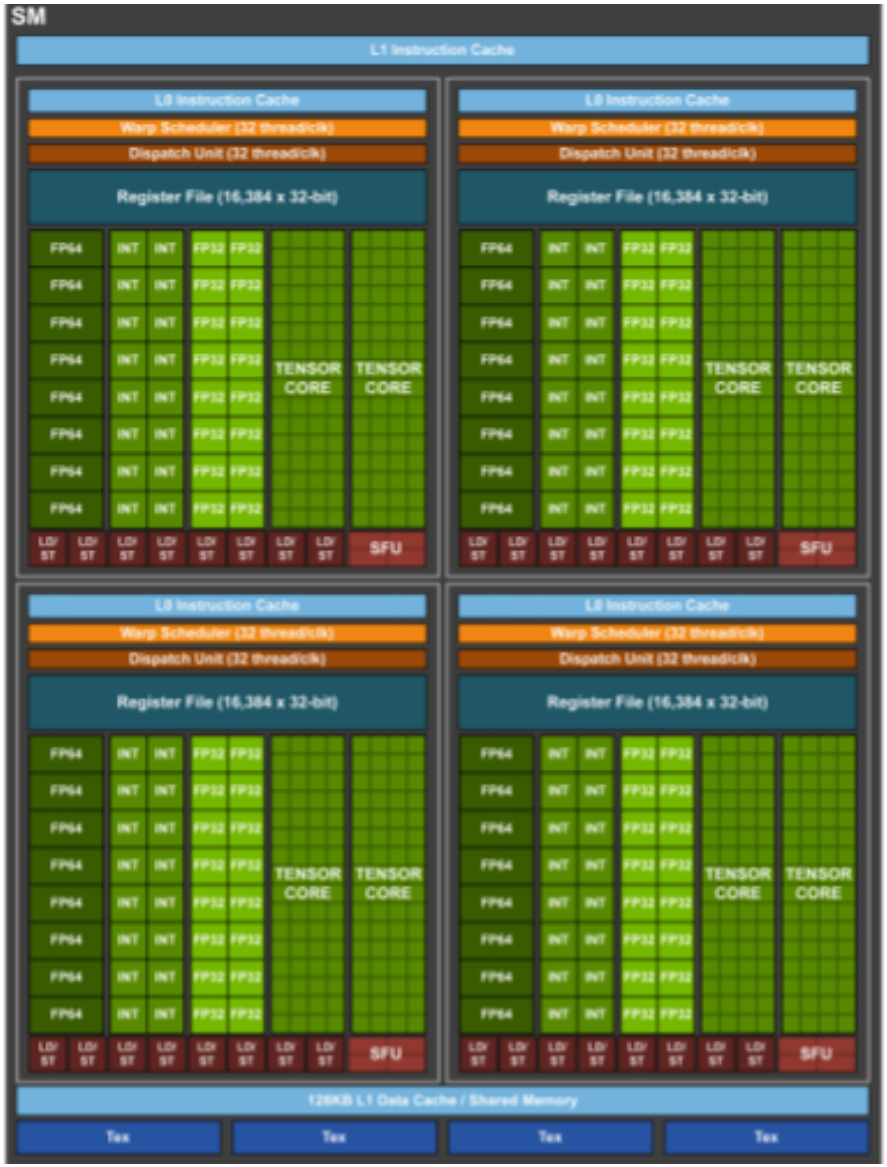
# Arquitectura Volta

- En mayo 2017 se lanza la séptima generación de CUDA, conocida con el nombre de Volta.
- 7.8 TFLOPS en doble precisión floating point (FP64)
- 15.7 TFLOPS en simple precisión (FP32)
- 125 TFLOPS en half-precisión (FP16)



# Arquitectura Volta

- La novedad más importante es la inclusión de un nuevo tipo de unidad de cómputo, los tensor cores.
- Además, se separan las unidades de FP32 de las de INT.
- Permite la ejecución simultánea de operaciones enteras y FP32.

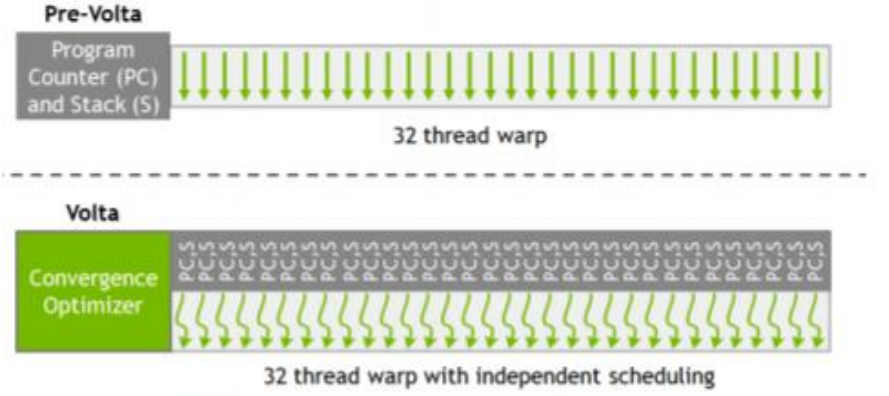


# Arquitectura Volta

- Los tensor cores:
  - Cada Tensor Core realiza 64 fused multiply–add (FMA) de punto flotante por ciclo de reloj
  - Hay 8 en cada multiprocesador y pueden hacer 512 operaciones FMA (o 1024 operaciones individuales).
- La V100 tiene una performance 1.8x superior que la P100 en la multiplicación de matrices de simple precision.
- Usando los tensor cores, que trabajan con entradas en half-precision (FP16) y acumulan en FP32, la mejora es de 9x.

# Arquitectura Volta

- **Desde el punto de vista de las capacidades de programación, la modificación más importante es la planificación independiente de los hilos.**
- **Hasta Pascal el modelo de ejecución es SIMT con divergencia y la suspensión de hilos.**
- **Cada hilo independiente mantiene su estado de ejecución, incluyendo un program counter y el stack de llamadas.**




Volta (bottom) independent thread scheduling architecture block diagram compared to Pascal and earlier architectures (top). Volta maintains per-thread scheduling resources such as program counter (PC) and call stack (S), while earlier architectures maintained these resources per warp.

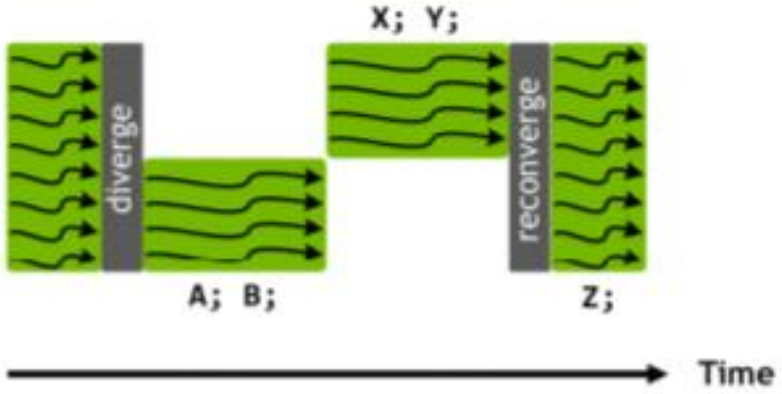
# Arquitectura Volta

- La arquitectura Volta incluye un optimizador de la planificación que determina cómo agrupar los hilos activos del mismo warp en las unidades SIMT.
- Esto conserva el alto rendimiento de ejecución SIMT pero con mucha más flexibilidad: los hilos pueden diverger y reconverger a nivel de subwarp.
- El optimizador se encarga de agrupar los hilos que ejecutan el mismo código y los ejecuta en paralelo para lograr la máxima eficiencia.

# Arquitectura Volta

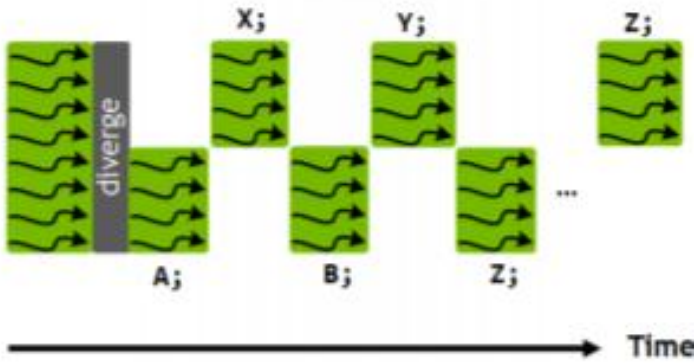
## SIMT Warp Execution Model of Pascal and Earlier GPUs

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```




## Volta Independent Thread Scheduling

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



# Arquitectura Volta

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```

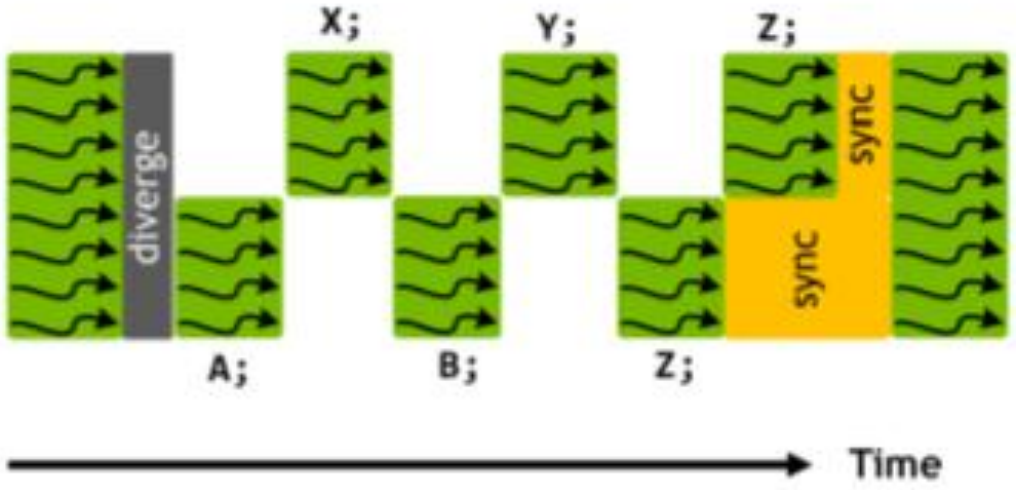



Figure 23. Programs use Explicit Synchronization to Reconverge Threads in a Warp

- **Esto requiere rever las operaciones de warp shuffle considerando los hilos activos del warp!!!!**

# Aspectos no incluidos en el curso

- **Warp Shuffle revisitado:**
  - **Cambios a partir de CUDA 9**
  - **Las primitivas cambian el nombre (incluyen _sync)**
- **Thread Block Clusters:**
  - **Agregados a partir de CUDA 9**
  - **Se garantiza que ejecutan en un GPC**
- **Distributed Shared Memory:**
  - **Los TBC pueden compartir shared memory aunque ejecuten en un distinto multiprocesador.**
- **Arquitecturas posteriores**