Universidad Autónoma de Madrid

excelencia UAM CSIC

Escuela Politécnica Superior

Tema 3: Paralelismo de datos a gran escala con GPU

Francisco J. Gómez Arribas

Agenda

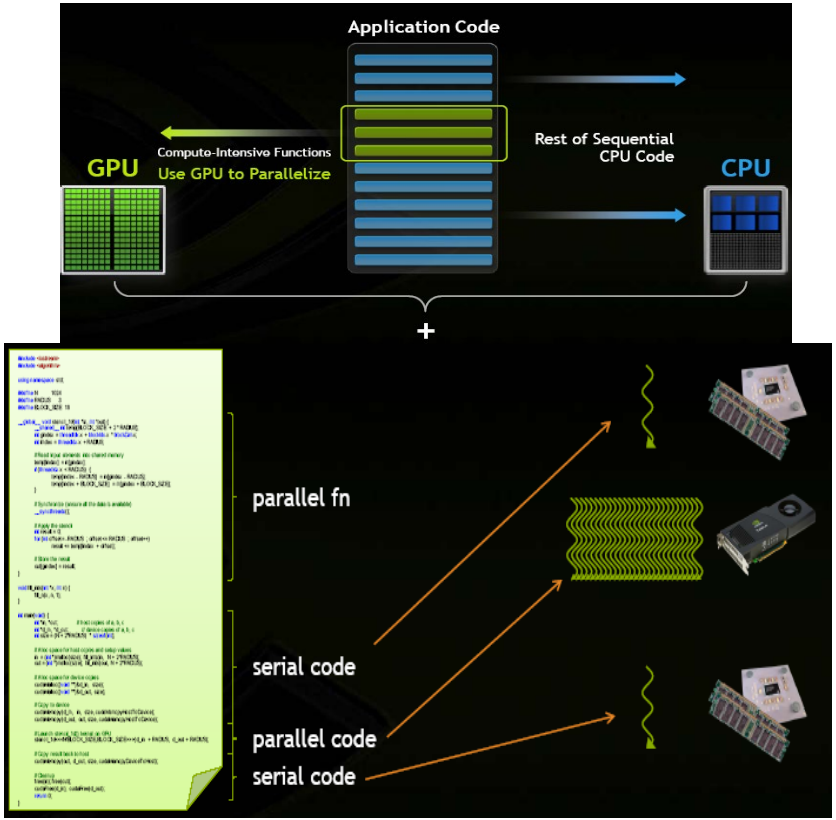
- **GPU Architecture**
  - **Some Numbers: Fitting Blocks &Threads (G80 Arch)**
- **CUDA Programing Model**
- **CUDA Memory architecture**
- **CUDA Warp execution**
  - **Latency Hiding**
  - **Threads divergence**

CUDA

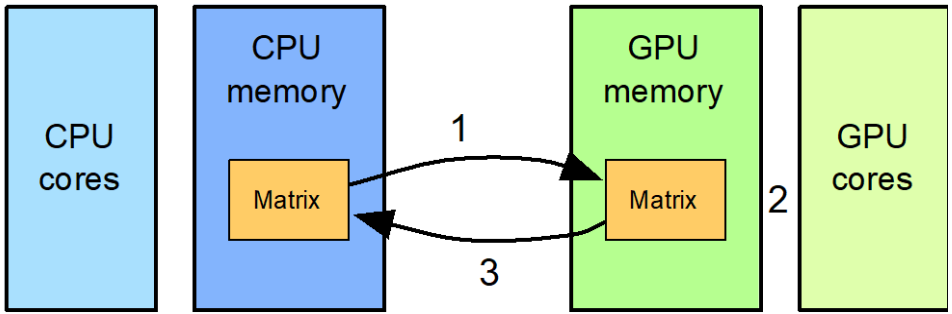
What is CUDA?

- **Compute Unified Device Architecture.**
- A powerful parallel programming model for issuing and managing computations on the GPU without mapping them to a graphics API.
  - Heterogenous - mixed serial-parallel programming
  - Scalable - hierarchical thread execution model
  - Accessible - minimal but expressive changes to C

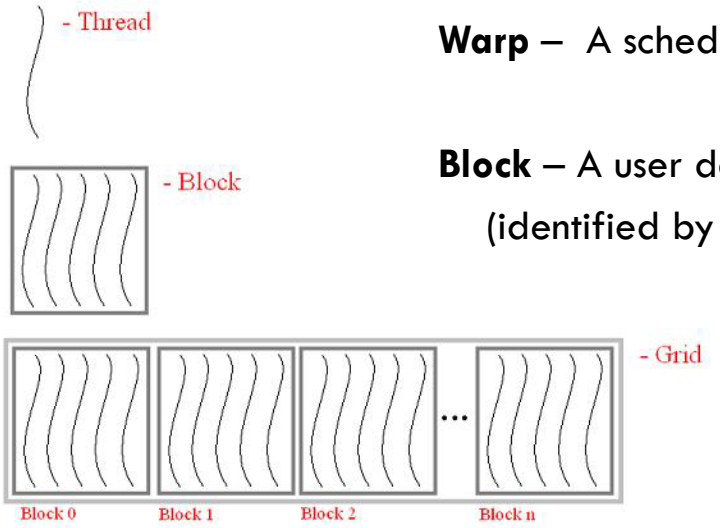
Heterogeneous Computing



- Computation is offloaded to the GPU
- Three steps
  - CPU-GPU data transfer (1)
  - GPU kernel execution (2)
  - GPU-CPU data transfer (3)



CUDA Thread Hierarchy



Thread – Distributed by the CUDA runtime (identified by threadIdx)

Warp – A scheduling unit of up to 32 threads

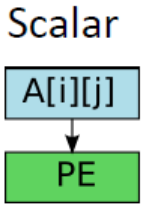
Block – A user defined group of 1 to 512 threads. (identified by blockIdx)

Grid – A group of one or more blocks. A grid is created for each CUDA kernel function

Multithread vs vectorial

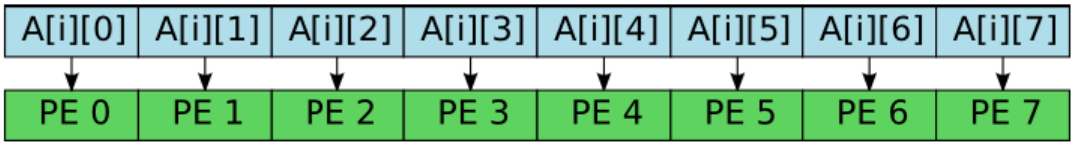
Scalar program

```
float A[4][8];
do-all(i=0;i<4;i++){
    do-all(j=0;j<8;j++){
        A[i][j]++;
    }
}
```



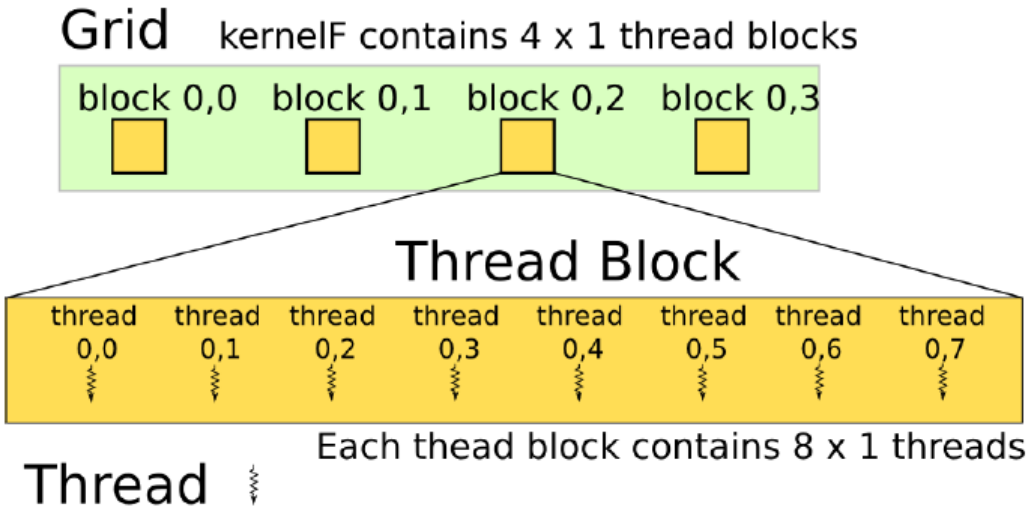
Vector width is exposed to programmers.

Vector if width 8

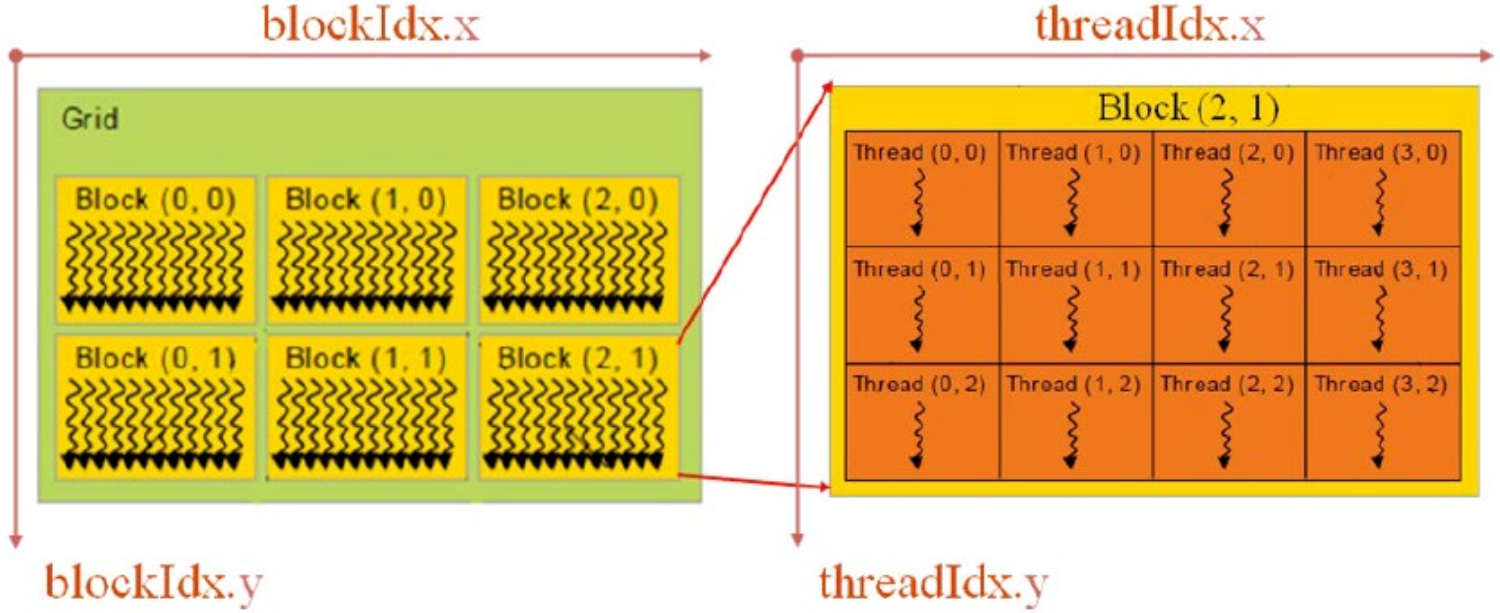


Multithread vs vectorial

Multithreaded: (4x1)blocks – (8x1) threads



Multithread 2D




```
dim3 numBloques    (3, 2);  // 3 * 2 =>  6 bloques
dim3 threadsBloque (4, 3);  // 4 * 3 => 12 threads
miKernel<<<numBloques, threadsBloque>>>(Ad,Bd,Cd);
```

Multithread : dim3

Ejemplo de Configuración 2D:

- Malla de 2x2 bloques
- Cada bloque de 4 hilos

Un kernel se ejecuta como

- Una malla o grid 1D ó 2D de
- bloques dehilos 1D, 2D ó 3D.

Los hilos y los bloques tienen IDs para que cada hilo pueda acotar sobre qué datos trabaja, y facilitar el direccionamiento al procesar datos multidimensionales.

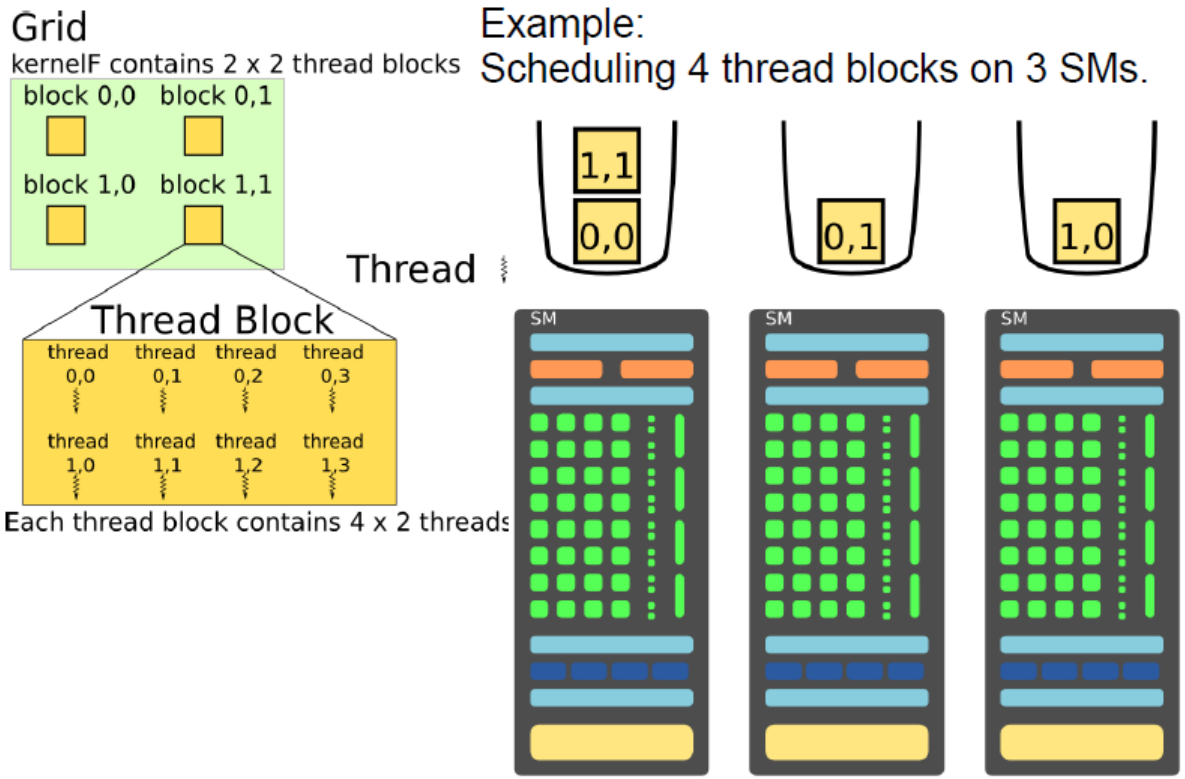
```
#define NUM_BLX 2
#define NUM_BLY 2
#define BLOCKSIZE 4
__global__ void mikernel()
{
  int bid=blockIdx.x*gridDim.y+blockIdx.y;
  int tid=bid*blockDim.x+ threadIdx.x;
  float result = sin(tid) * tan(tid);
}

void main()
{
  dim3 dimGrid (NUM_BLX, NUM_BLY);
  dim3 dimBlock(BLOCKSIZE);
  mikernel<<<dimGrid, dimBlock>>>();
  return EXIT_SUCCESS;
}
```

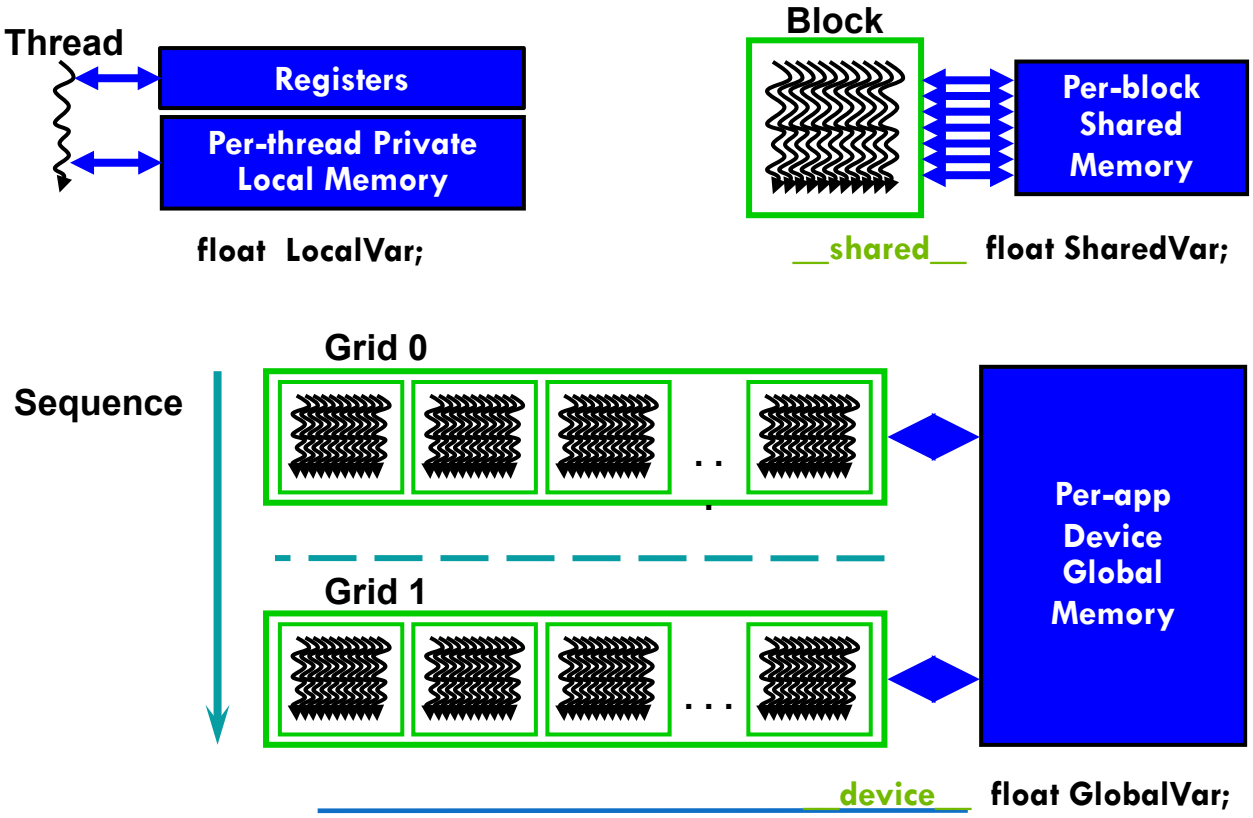
CUDA Execution Model

- Application can include multiple kernels
- Threads of the same block run on the same SM
  - So threads in SM can operate and share memory
  - Block in an SM is divided into warps of 32 threads each
  - A warp is the fundamental unit of dispatch in an SM
- Blocks in a grid can coordinate using global memory.
  - Two threads from two different blocks cannot share data through the low latency shared memory
- A kernel is executed as a grid of thread blocks

Scheduling Thread Blocks on SM

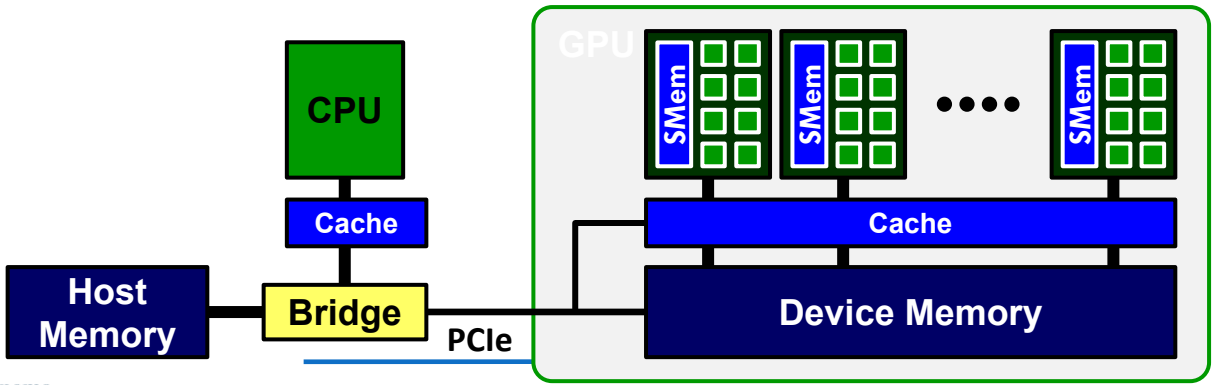


CUDA Parallel Threads and Memory



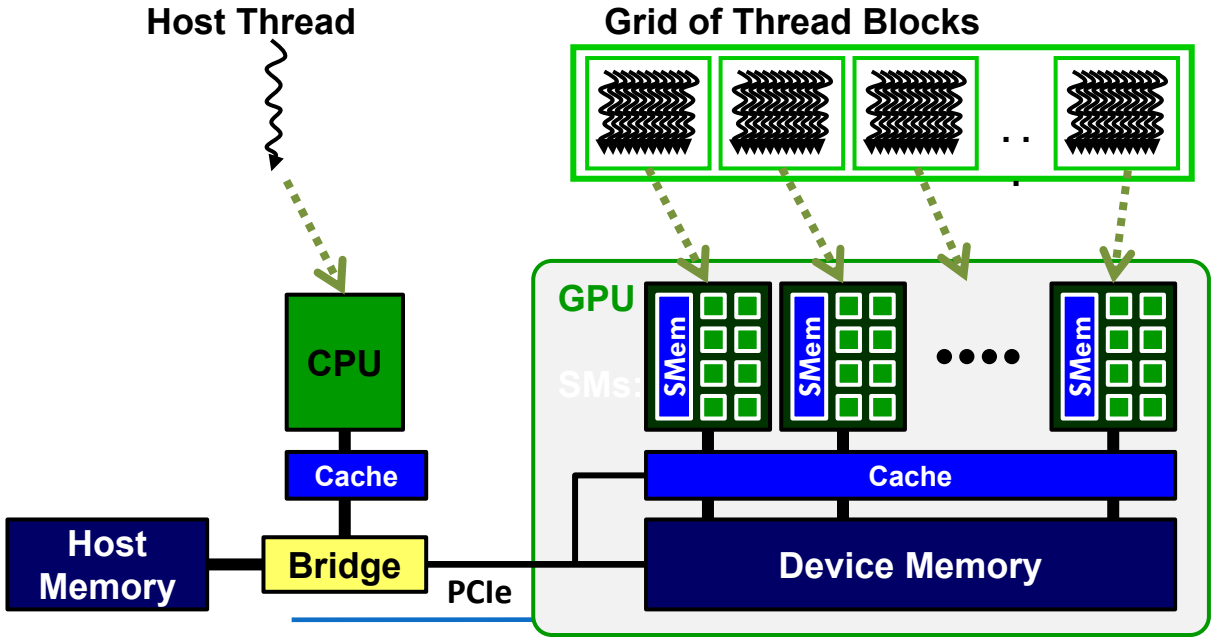
Using CPU+GPU Architecture

- Heterogeneous system architecture
- Use the right processor and memory for each task
- CPU excels at executing a few serial threads
  - Fast sequential execution
  - Low latency cached memory access
- GPU excels at executing many parallel threads
  - Scalable parallel execution
  - High bandwidth parallel memory access



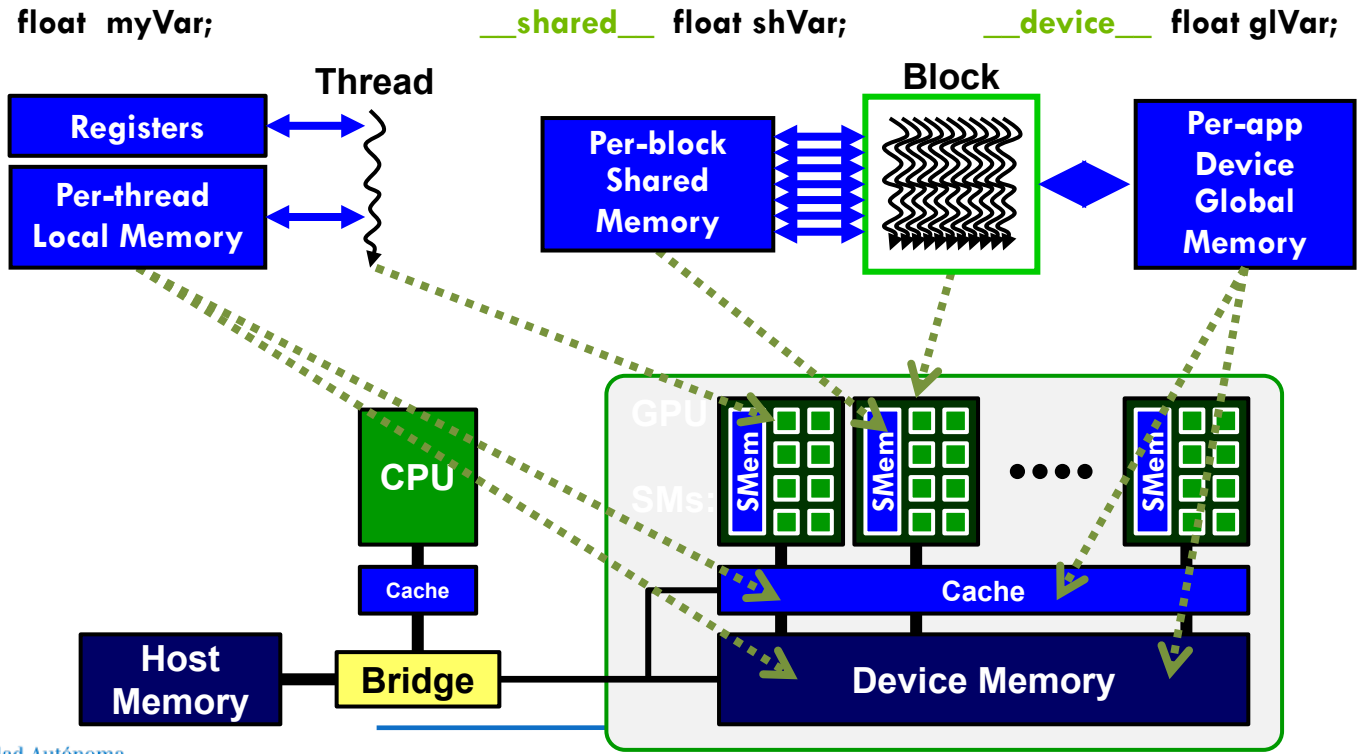
CUDA kernel maps to Grid of Blocks

- kernel_func<<<nblk, nthread>>>(param, …);



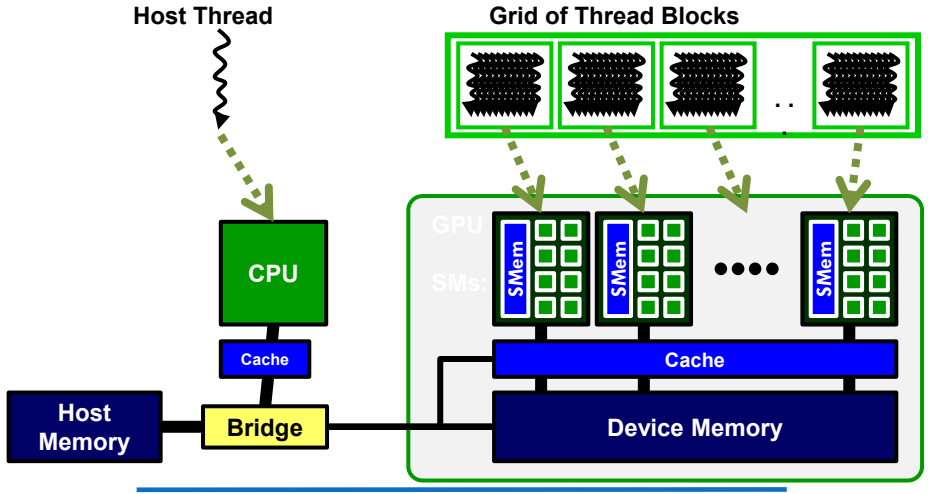
Thread blocks execute on an SM

- Thread instructions execute on a core



CUDA Parallelism

- CUDA virtualizes the physical hardware
  - Thread is a virtualized scalar processor (registers, PC, state)
  - Block is a virtualized multiprocessor (threads, shared memory)
- Scheduled onto physical hardware without pre-emption
  - Threads/blocks launch & run to completion
  - Blocks execute independently

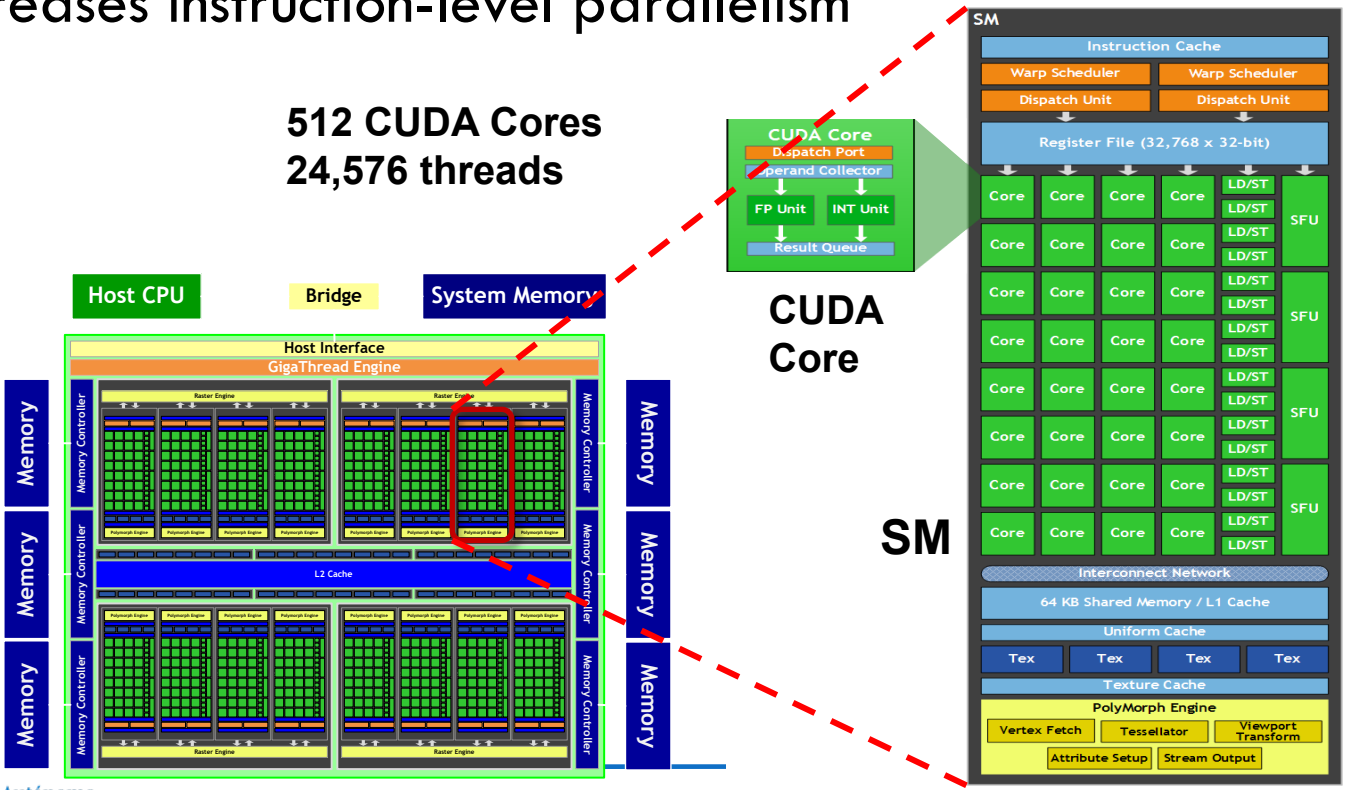


Expose Massive Parallelism

- Use hundreds to thousands of thread blocks
  - A thread block executes on one SM
  - Need many blocks to use 10s of SMs
  - SM executes 2 to 8 concurrent blocks efficiently
  - Need many blocks to scale to different GPUs
  - Coarse-grained data parallelism, task parallelism
- Use hundreds of threads per thread block
  - A thread instruction executes on one core
  - Need 384 – 512 threads/SM to use all the cores all the time
  - Use multiple of 32 threads (warp) per thread block
  - Fine-grained data parallelism, vector parallelism, thread parallelism, instruction-level parallelism

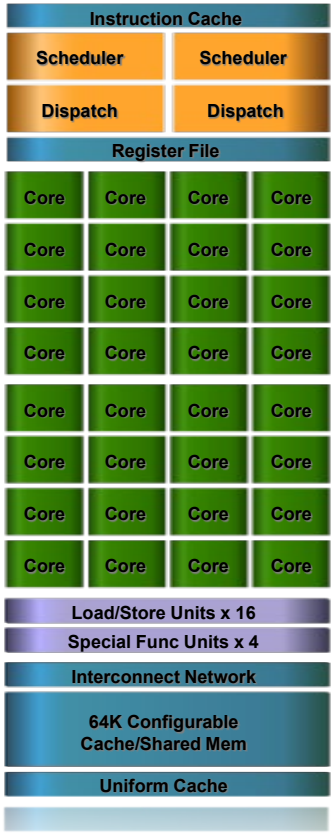
Fermi Streaming Multiprocessor (SM)

- Increases instruction-level parallelism



SM Parallel Instruction Execution

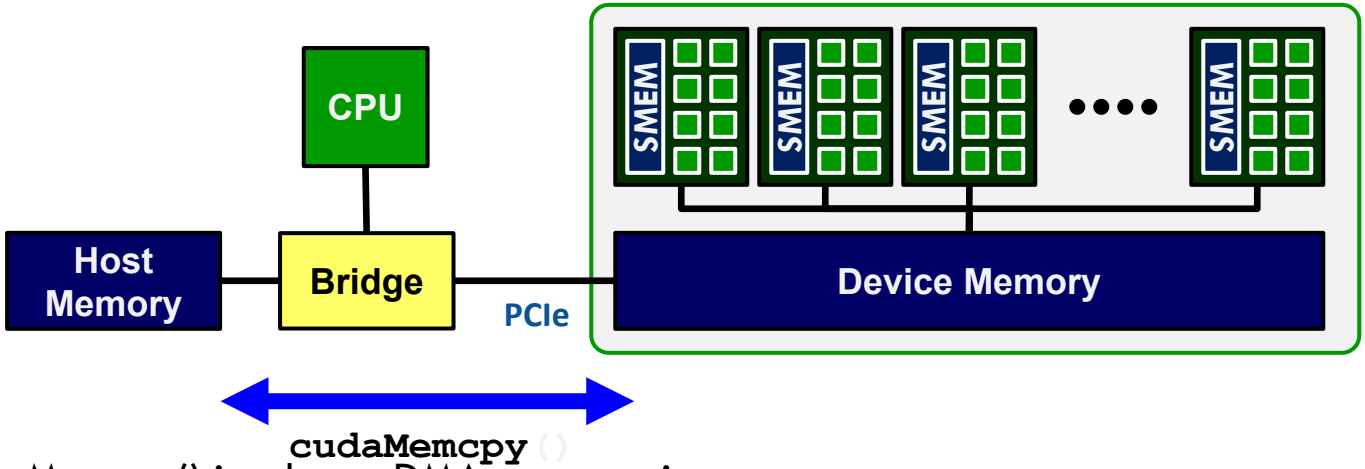
- SIMT (Single Instruction Multiple Thread) execution
  - Threads run in groups of 32 called warps
  - Threads in a warp share instruction unit (IU)
  - HW automatically handles branch divergence
- Hardware multithreading
  - HW resource allocation & thread scheduling
  - HW relies on threads to hide latency
- Threads have all resources needed to run
  - Any warp not waiting for something can run
  - Warp context switches are zero overhead



Use per-Block Shared Memory

- Latency is an order of magnitude lower than L2 or DRAM
- Bandwidth is 4x – 8x higher than L2 or DRAM
- Place data blocks or tiles in shared memory when the data is accessed multiple times
- Communicate among threads in a block using Shared memory
- Use synchronization barriers between communication steps
  - __syncthreads() is single bar.sync instruction – very fast
- Threads of warp access shared memory banks in parallel via fast crossbar network
- Bank conflicts can occur – incur a minor performance impact
- Pad 2D tiles with extra column for parallel column access if tile width == # of banks (16 or 32)

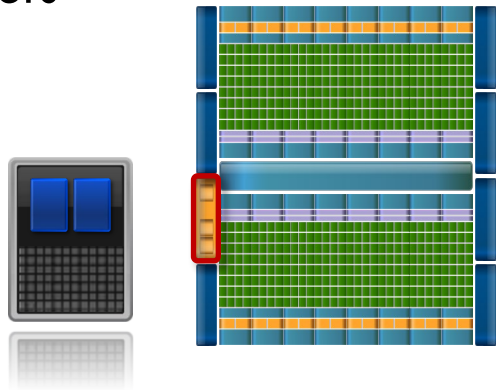
Using cudaMemCpy()



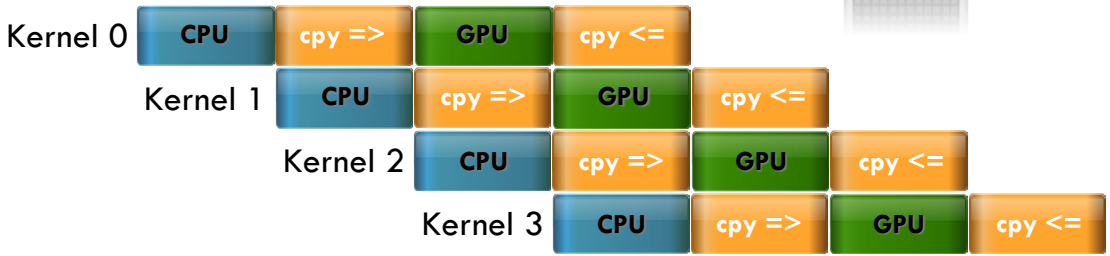
- cudaMemcpy() invokes a DMA copy engine
- Minimize the number of copies
- Use data as long as possible in a given place
- PCIe gen2 peak bandwidth = 6 GB/s
- GPU load/store DRAM peak bandwidth = 150 GB/s

Overlap Computing & CPU⇔GPU Transfers

- cudaMemcpy() invokes data transfer engines
  - CPU→GPU and GPU→CPU data transfers
  - Overlap with CPU and GPU processing

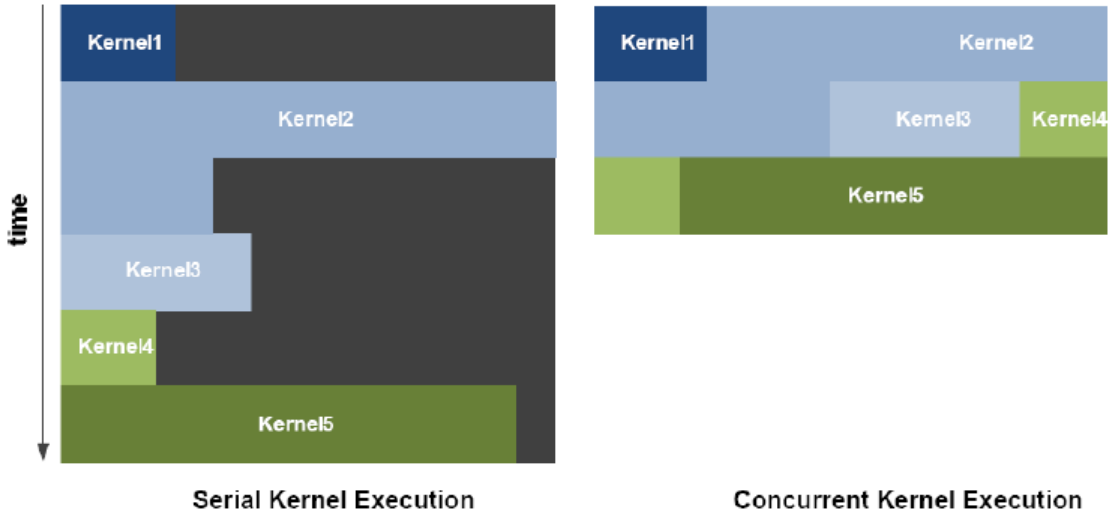


- Pipeline Snapshot:

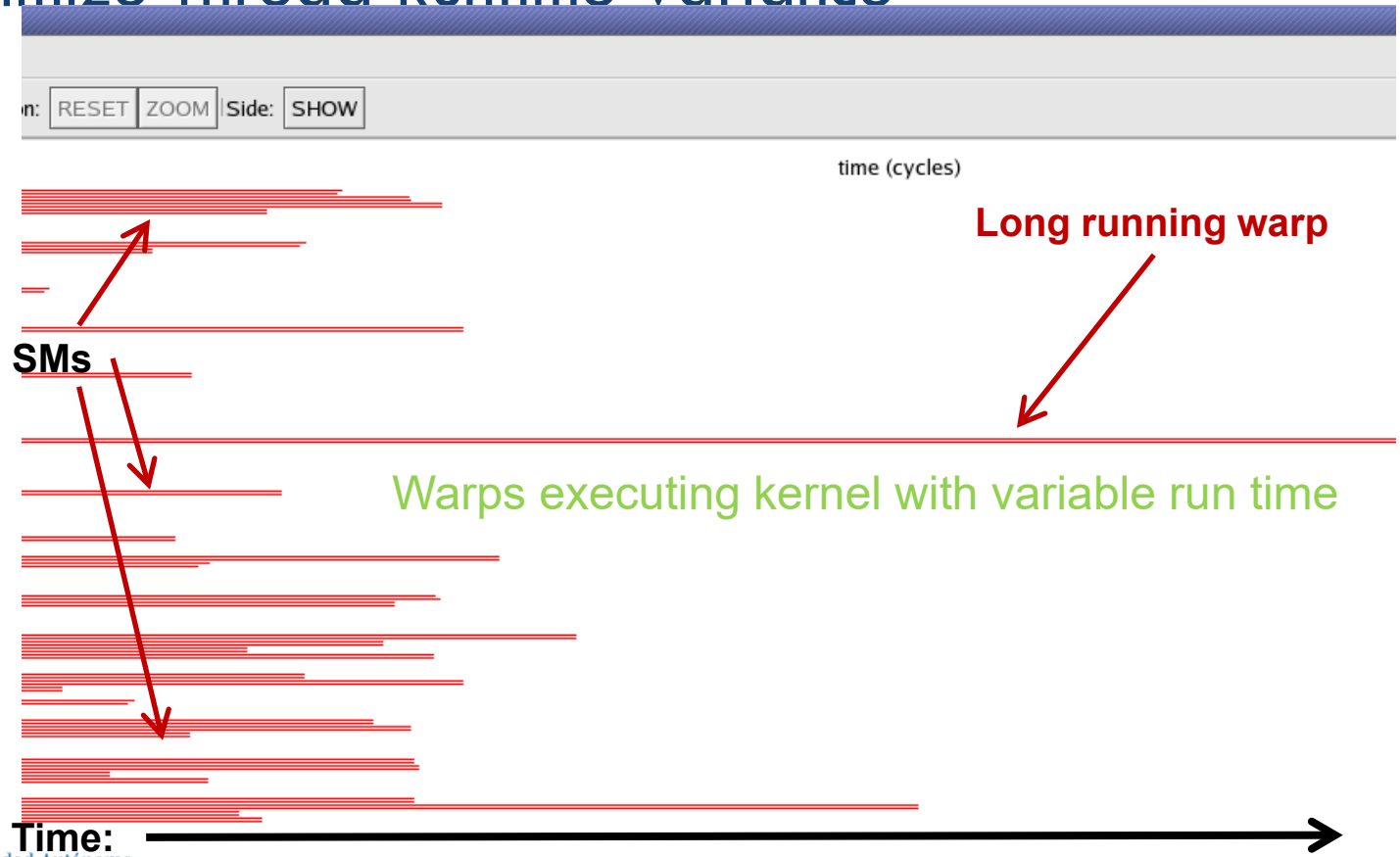


Kernel scheduling since Fermi

- At any point of time the entire Fermi device is dedicated to a single application
  - Switch from an application to another takes 25 microseconds
- Fermi can simultaneously execute multiple kernels of the same application



Minimize Thread Runtime Variance



Optimizing Parallel GPU Performance

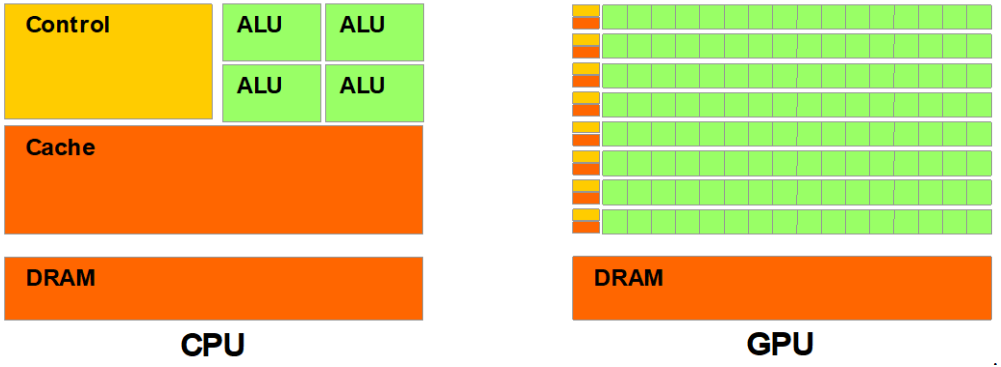
- Understand the parallel architecture
- Understand how application maps to architecture
- Use LOTS of parallel threads and blocks
- Often better to redundantly compute in parallel
- Access memory in local regions
- Leverage high memory bandwidth
- Keep data in GPU device memory
- Experiment and measure

Agenda

- **GPU Architecture**
- **CUDA Programing Model**
  - **Some Numbers: Fitting Blocks &Threads (G80 Arch)**
- **CUDA Memory architecture**
- **CUDA Warp execution**
  - **Latency Hiding**
  - **Threads divergence**

Memory, Memory, Memory

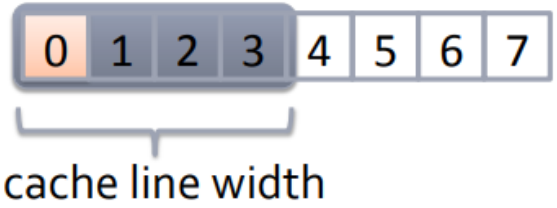
- A many core processor = A device for turning a compute bound problema into a memory bound problema.



- Lots of processors, only one socket
- Memory concerns dominate performance tuning

Memory is SIMD Too!

- Virtually all processors have SIMD memory subsystems



- This has two effects:
  - Sparse access wastes bandwidth

2 words used, 8 words loaded:
¼ effective bandwidth

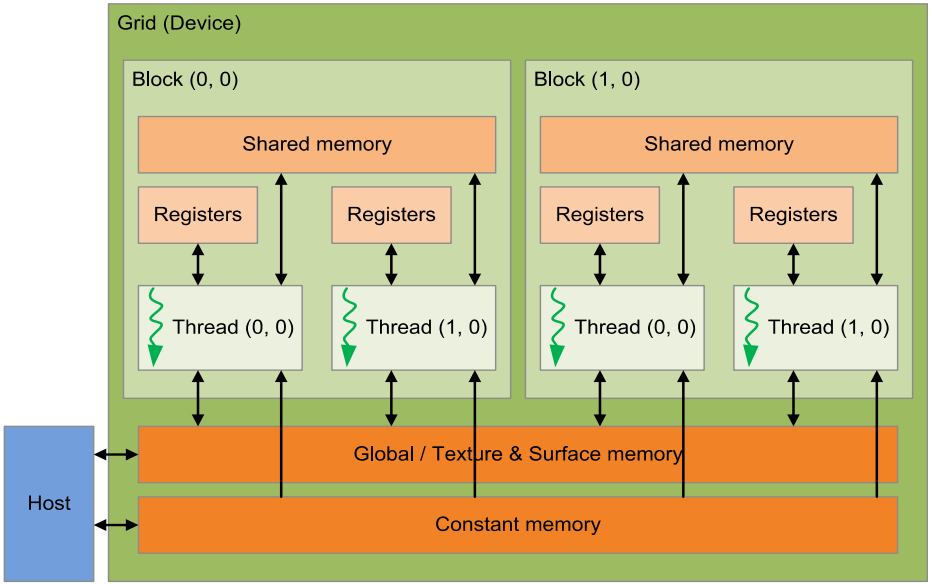
  - Unaligned access wastes bandwidth

4 words used, 8 words loaded:
½ effective bandwidth

CUDA Memory Hierarchy

- The CUDA platform has three primary memory types

Local Memory – per thread memory for automatic variables and register spilling.



Shared Memory – per block low-latency memory to allow for intra-block data sharing and synchronization. Threads can safely share data through this memory and can perform barrier synchronization through _ _syncthreads()

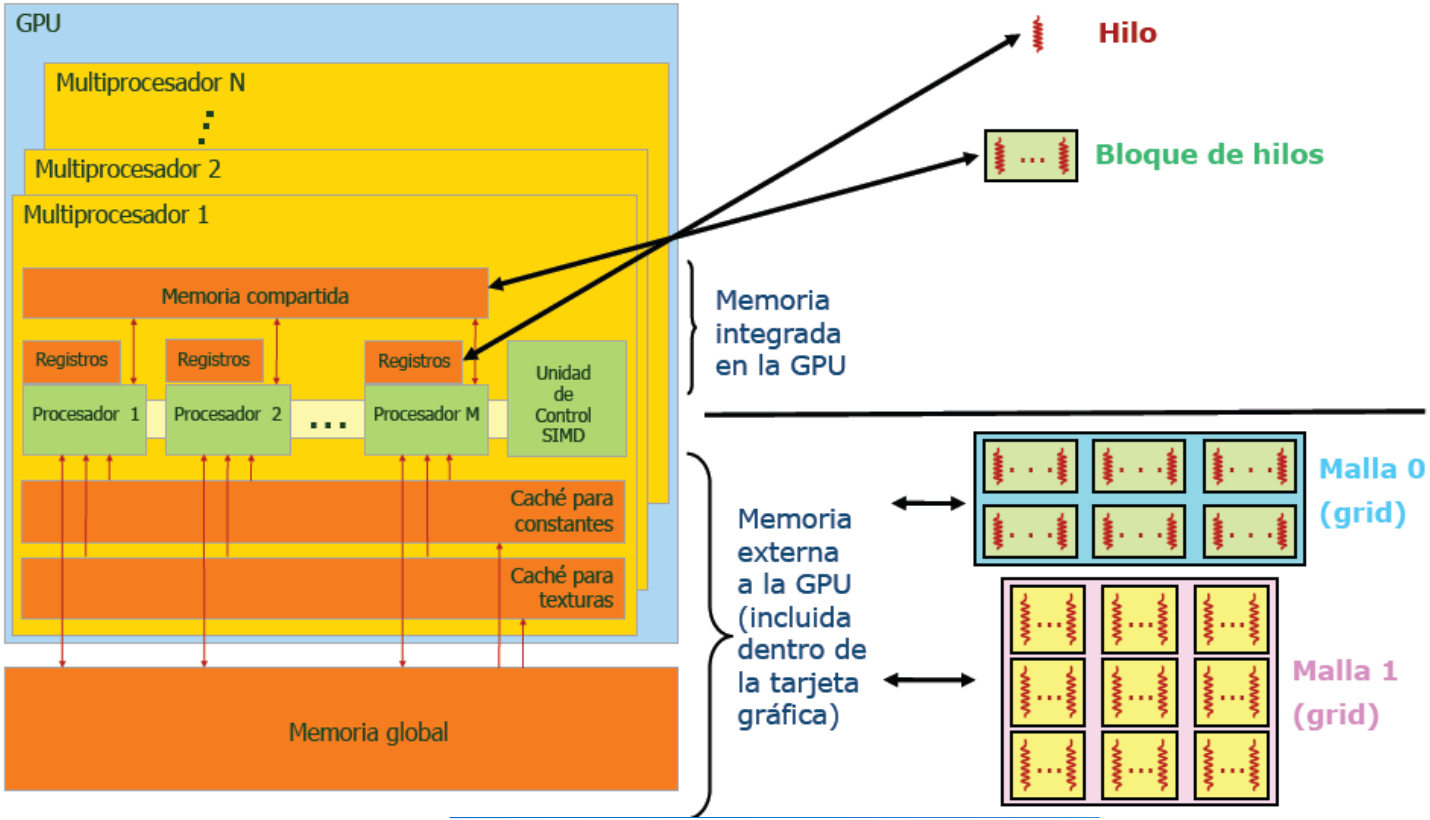
Global Memory – device level memory that may be shared between blocks or grids

A Common Programming Pattern

- Global memory reside in device memory (DRAM) - much slower access than shared memory
- So, a profitable way of performing computation on the device is to block data to take advantage of fast shared memory:
  - Partition data into data subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

Consideraciones en el acceso a memoria



Using shared memory in CUDA

```
__global__ void mykernel() {
        __shared__ float x[100];
        ...
        }
```

One array per block!

Different:
x[0] in block 10
x[0] in block 11

Same:
x[0] in thread 5 of block 10
x[0] in thread 6 of block 10

Using shared memory in CUDA

```
__global__ void mykernel() {
        __shared__ float x[100];
        ...
        __syncthreads();
        ...
        }
```

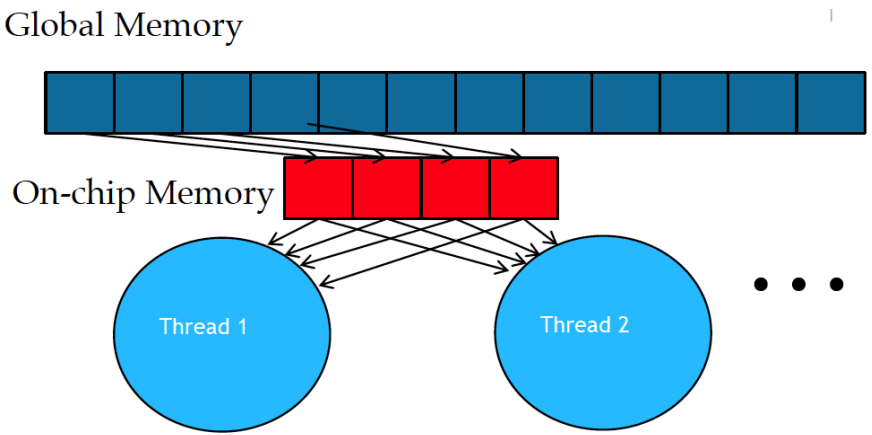
A thread won't continue until all threads of the block have reached this point

Using shared memory in CUDA

Key elements of CUDA programs

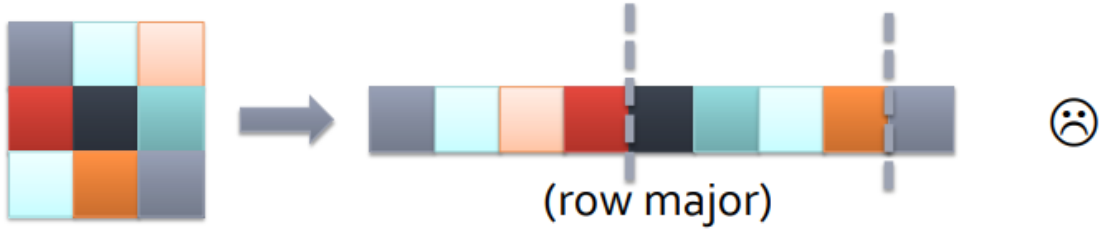
- Allocating GPU memory, moving data between CPU memory and GPU memory
- Creating blocks of threads, launching kernels
- Allocating shared memory for sharing data inside a block, using __syncthreads to synchronize work
  - Shared memory is small.
  - Example: 64 KB in total per SM ("streaming multiprocessor")
    - if you want to have 8 active blocks on each SM
    - then you can only allocate at most 8 KB of shared memory per block

Memory Tiling/Blocking - Basic Idea

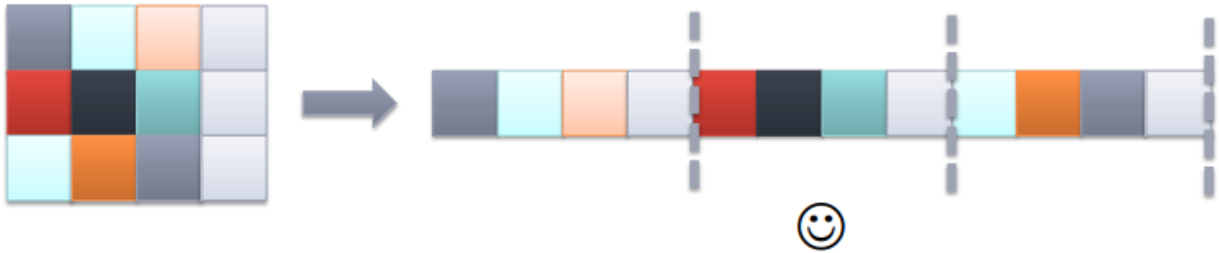


- Divide the global memory content into tiles.
- Focus the computation of threads on one or a small number of tiles at each point in time

Data Structure Padding



- Multidimensional arrays are usually stored as monolithic vectors in memory
- Care should be taken to assure aligned memory accesses for the necessary access pattern



Coalescing

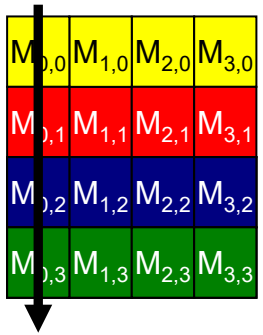
- GPUs and CPUs both perform memory transactions at a larger granularity than the program requests ("cache line")
- GPUs have a "coalescer", which examines memory requests dynamically and coalesces them
- To use bandwidth effectively, when threads load, they should:
  - Present a set of unit strided loads (dense accesses)
  - Keep sets of loads aligned to vector boundaries

Memory Coalescing

- Coalesced accesses

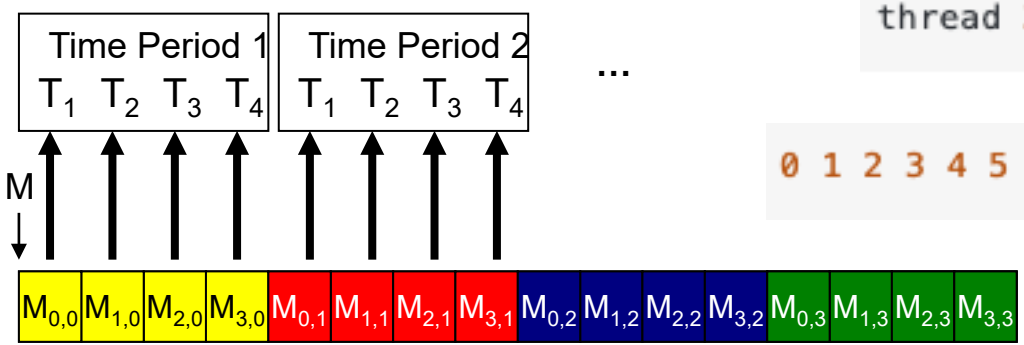
When accessing global memory, peak bandwidth utilization occurs when parallel threads running the same instruction access to consecutive locations in the global memory, the most favorable access pattern is achieved.

Thread access direction in Kernel code




```
0 1 2 3
4 5 6 7
8 9 a b
```

```
thread 0:  0, 4, 8
thread 1:  1, 5, 9
thread 2:  2, 6, a
thread 3:  3, 7, b
```

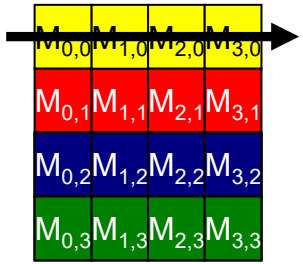



```
0 1 2 3 4 5 6 7 8 9 a b
```

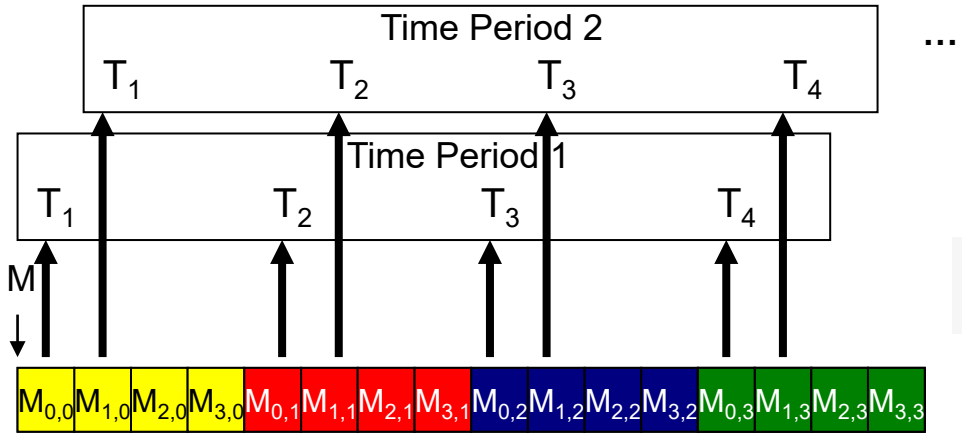
Memory Coalescing

- Uncoalesced accesses

Thread access direction in Kernel code




```
0 1 2 3
4 5 6 7
8 9 a b
```



...

```
thread 0:  0, 1, 2
thread 1:  3, 4, 5
thread 2:  6, 7, 8
thread 3:  9, a, b
```

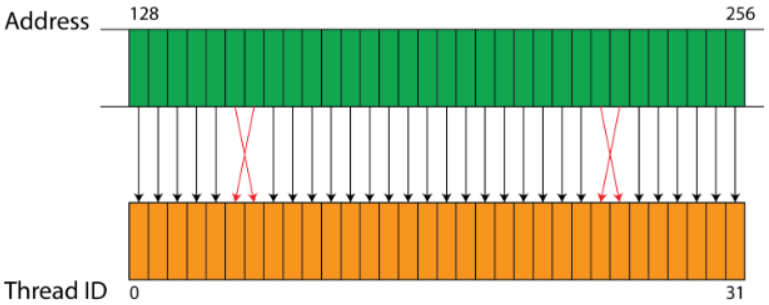
```
0 1 2 3 4 5 6 7 8 9 a b
```

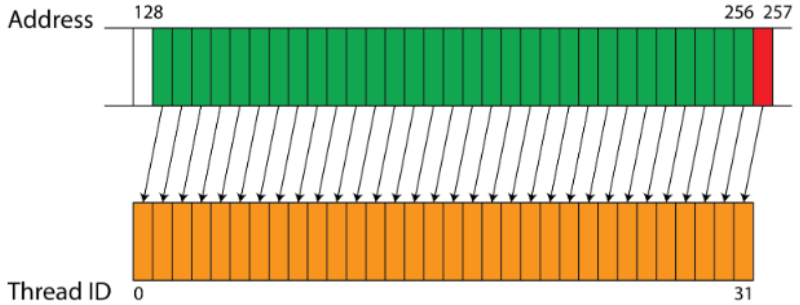
Global Memory access

- Coalescing: Compute Capability

```
__global__ coalesced_access(float*x)
{
  int tid= threadIdx.x+ blockDim.x*blockIdx.x;
  x[tid] = threadIdx.x;
}
```

```
__global__ not_coalesced_access(float*x)
{
  int tid= threadIdx.x+ blockDim.x*blockIdx.x;
  x[tid*100] = threadIdx.x;
}
```



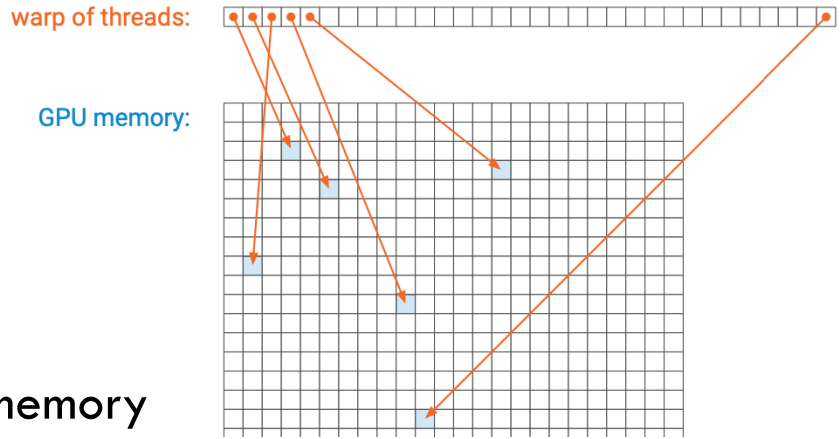


The following instruction is not coalesced:

```
stride=4;
shmem[threadIdx.x]=gmem[stride*blockIdx.x*blockDim.x + threadIdx.x*stride];
```

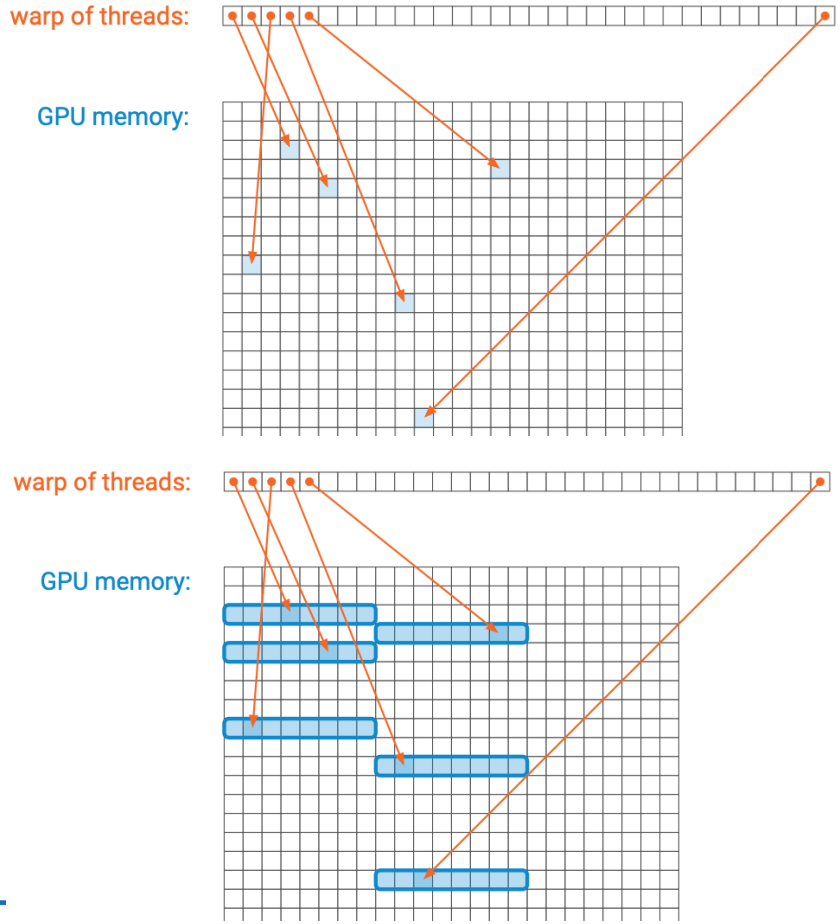
Global Memory access

- A key challenge in CPU code:
  getting data fast enough from the CPU memory
- A key challenge in GPU code:
  getting data fast enough from the GPU memory
- ***GPU Memory acces Patern***
  - Blocks are divided in warps
    - warp = 32 threads
- ***Entire warp executes synchronously***
  - If one thread reads some memory,
    all threads of the warp read some memory



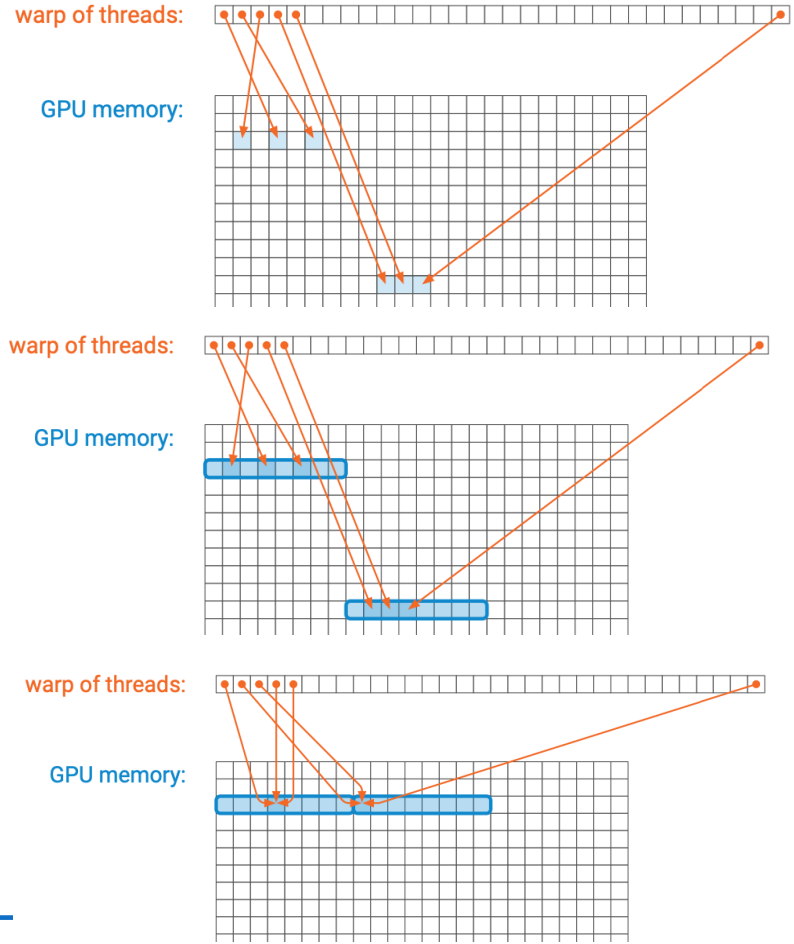
Global Memory access

- ***GPU Memory acces Patern***
  - Blocks are divided in warps
    - warp = 32 threads
- ***Entire warp executes synchronously***
  - If one thread reads some memory,
    all threads of the warp
    read some memory



Memory access Patern

- One memory read in kernel: ***entire warp of threads reads memory simultaneously***
- Threads access small continuous parts of memory: need to load few cache lines → **good**
- Threads access 32 different locations far from each other: need to load many cache lines → **bad**

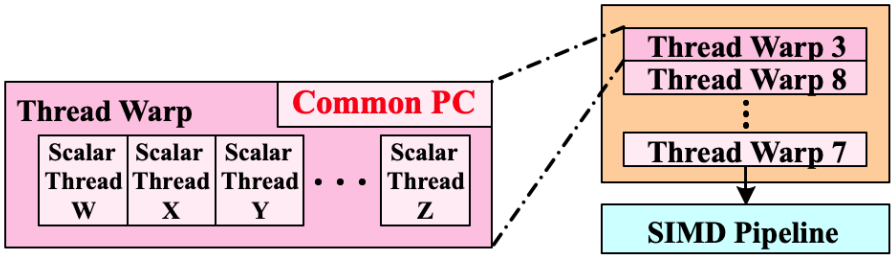


Agenda

- GPU Architecture
- CUDA Programing Model
  - Some Numbers: Fitting Blocks &Threads (G80 Arch)
- CUDA Memory architecture
- CUDA Warp execution
  - Latency Hiding
  - Threads divergence

CUDA Warp Execution

- CUDA utilizes SIMT (Single Instruction Multiple Thread)
- Warps are groups of 32 threads. Each warp receives a single instruction and "broadcasts" it to all of its threads.
- CUDA provides "zero-overhead" warp and thread scheduling. Also, the overhead of thread creation is on the order of 1 clock.



GPU Execution

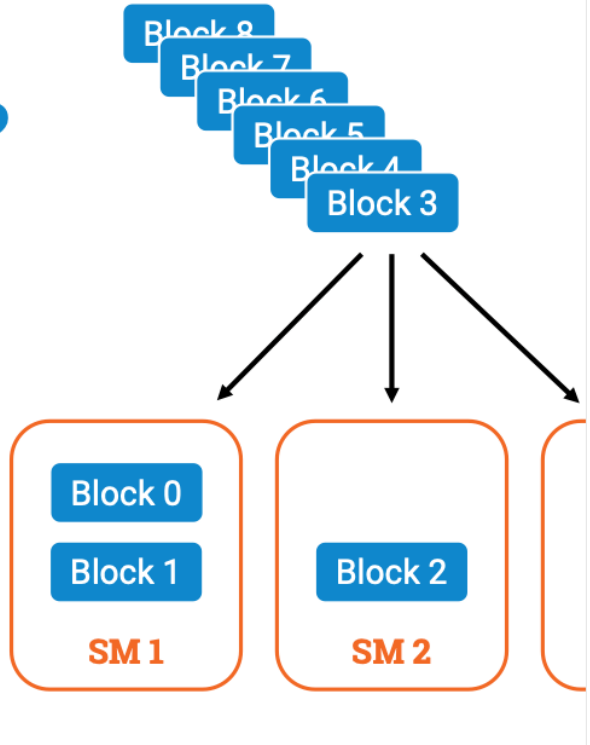
Key choices

- **Fixed:** 32 ***threads per warp***
- **We choose:** how many ***threads per block***
  - at most 1024
- **We choose:** how much ***shared memory per block***
  - at most 48 KB
- **Compiler chooses:** how many ***registers per thread***
  - depends on our kernel code
  - at most 255

GPU Execution

What happens when we launch a kernel?

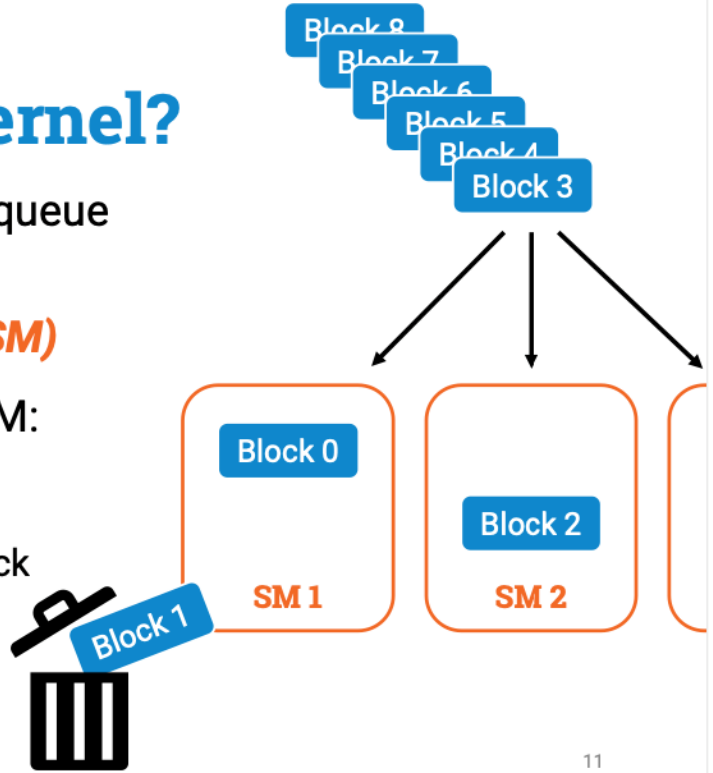
- All **blocks** are put in a GPU-wide queue
  - cheap, no resources allocated yet
- ***5 "streaming multiprocessors" (SM)***
- Whenever there is room in one SM:
  - SM takes a block from the queue
  - the block becomes active
  - resources are allocated for the block



GPU Execution

What happens when we launch a kernel?

- All **blocks** are put in a GPU-wide queue
  - cheap, no resources allocated yet
- ***5 "streaming multiprocessors" (SM)***
- Whenever there is room in one SM:
  - SM takes a block from the queue
  - the block becomes active
  - resources are allocated for the block
  - the block is there until all threads in the block finish running, then resources are freed



GPU Execution

What happens when SM starts to process a block?

- Block becomes active
  - room for **32 active blocks** per SM
- All warps of the block become active
  - room for **64 active warps** per SM
- Shared memory allocated for the block
  - **64 KB shared memory** available per SM
- Physical registers allocated for each thread
  - **65536 physical 32-bit registers** per SM

32 active blocks: 2 KB shared memory per block

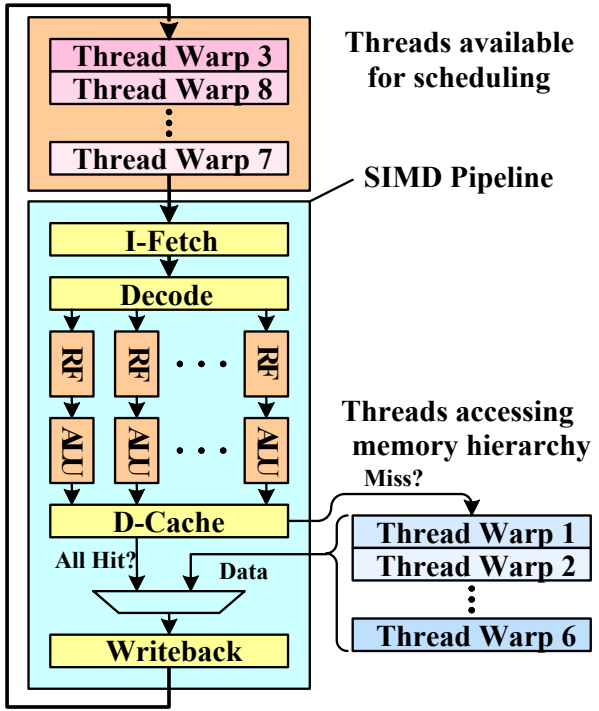
Blocks will have to wait in the queue until all these resources are available!

64 active warps × 32 threads/warp × 5 SMs = 10240 active threads

64 warps × 32 threads/warp × 32 registers/thread = 65536 registers

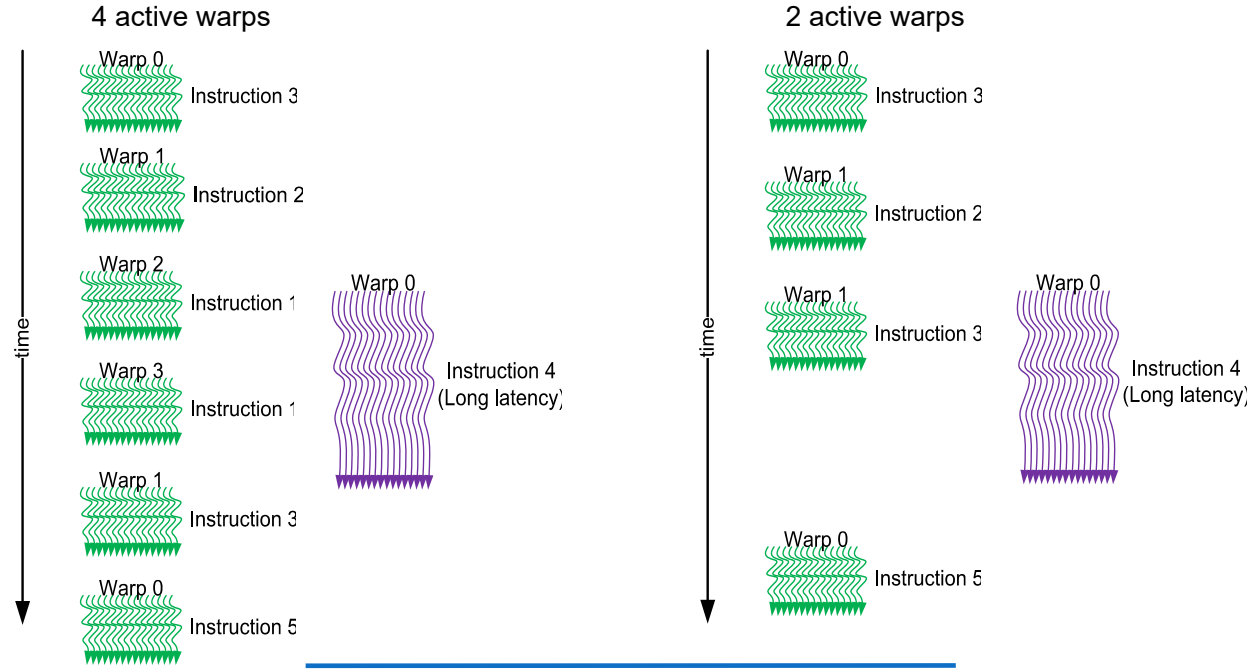
Latency Hiding via Fine Grain Multithreading

- Interleave warp execution to hide latencies
- Register values of all threads stays in register file
- Need 100~1000 threads
  - Graphics has millions of pixels
- Because a warp receives a single instruction, it will diverge and converge as each thread branches independently

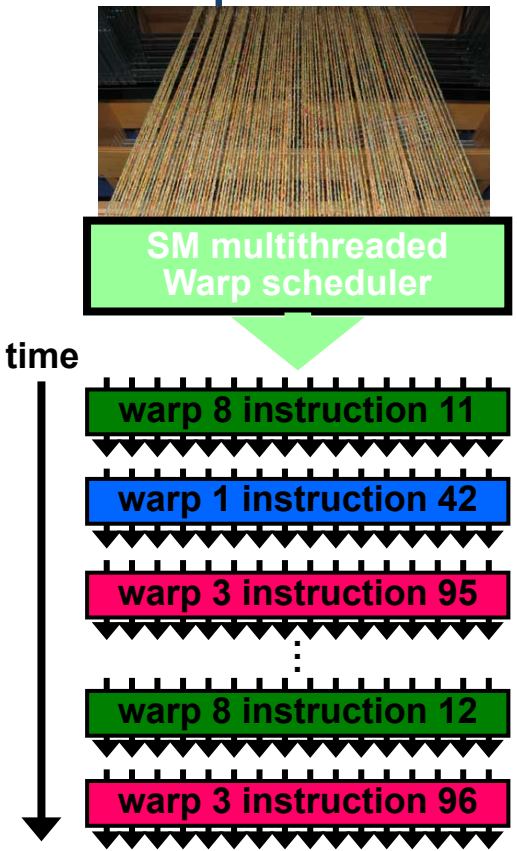


Latency Hiding

- Occupancy: ratio of active warps
  - Not only memory accesses (e.g., SFU)



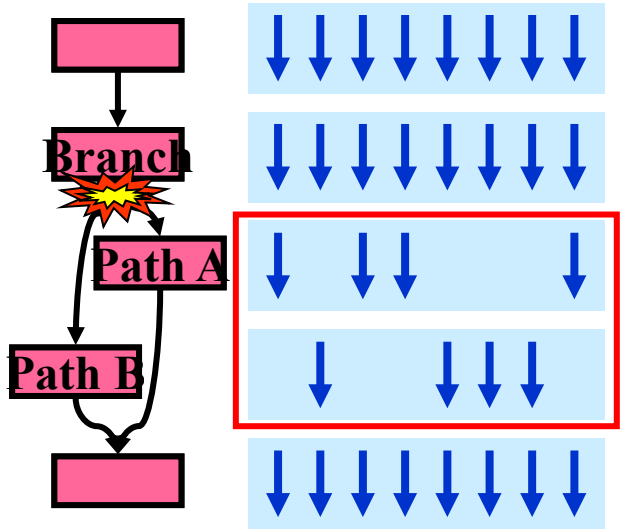
SM Warp Scheduling



- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected
- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
  - If one global memory access is needed for every 4 instructions
  - A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency

The Problem: Branch divergence

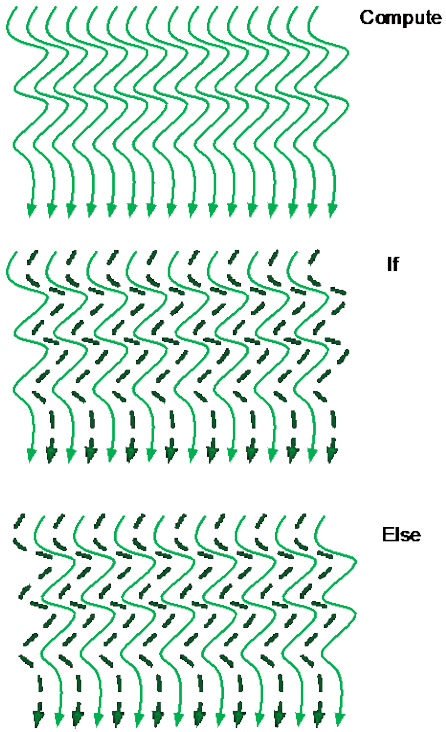
- GPU uses SIMD pipeline to save area on control logic.
  - Group scalar threads into warps
- *Branch divergence* occurs when threads inside warps branches to different execution paths.



Tread Utilization

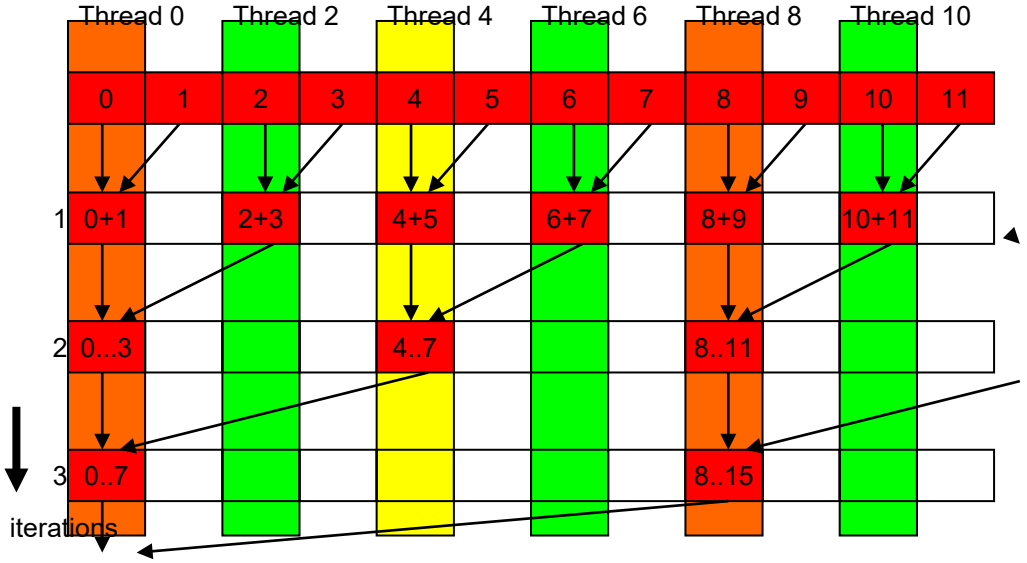
- Intra-warp divergence

```
Compute(threadIdx.x);
if (threadIdx.x % 2 == 0){
  Do_this(threadIdx.x);
}
else{
  Do_that(threadIdx.x);
}
```



Vector Reduction

- Naïve mapping



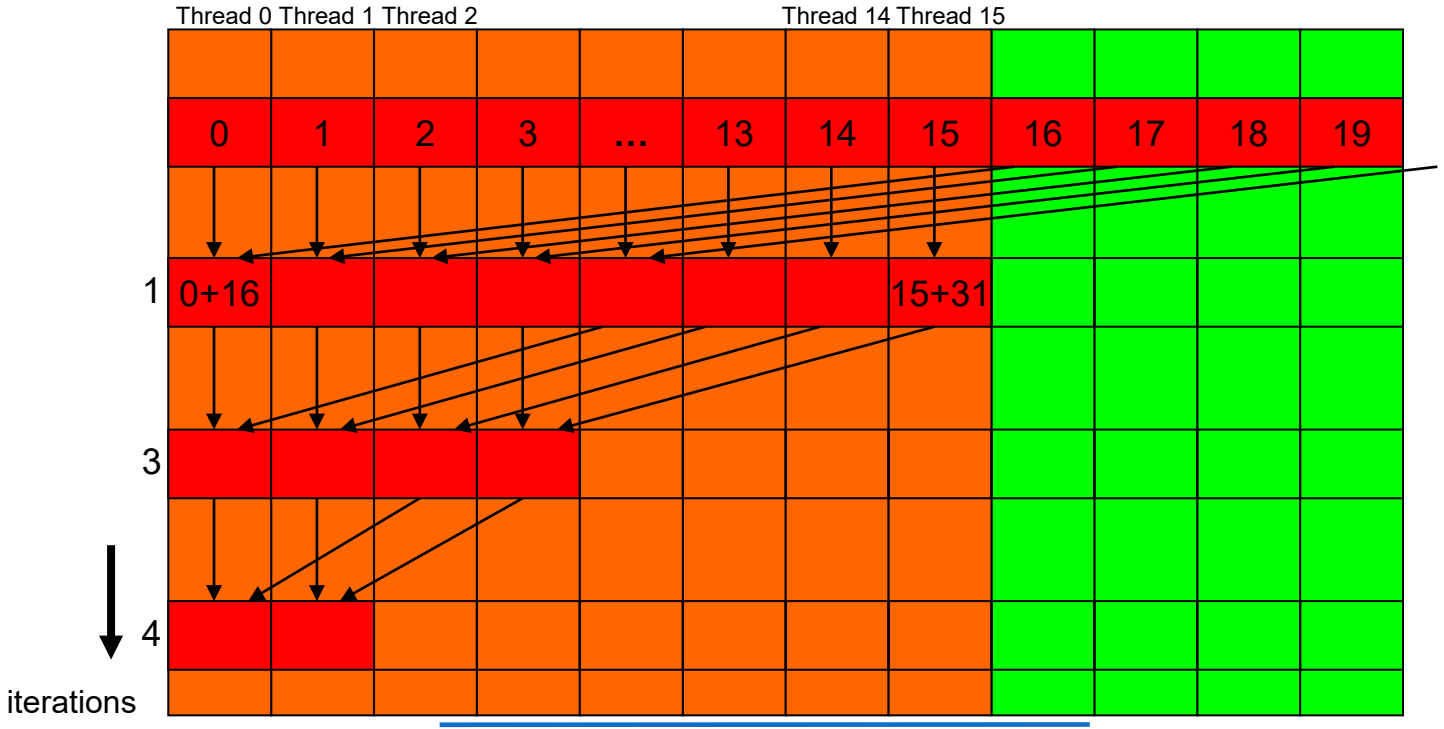

```
__shared__ float partialSum[]
unsigned int t = threadIdx.x;

for (int stride = 1; stride < blockDim.x; stride *= 2) {

  __syncthreads();
  if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t + stride];
}
```

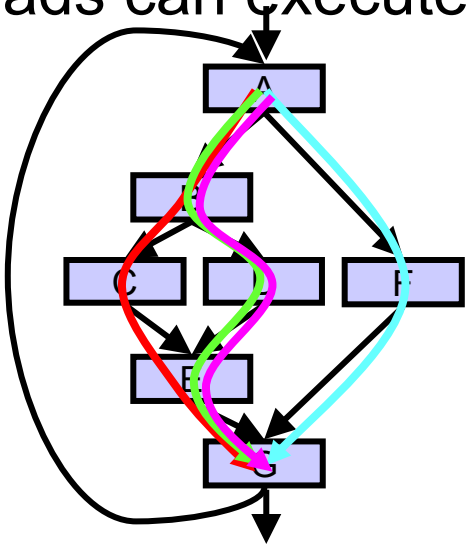
Vector Reduction

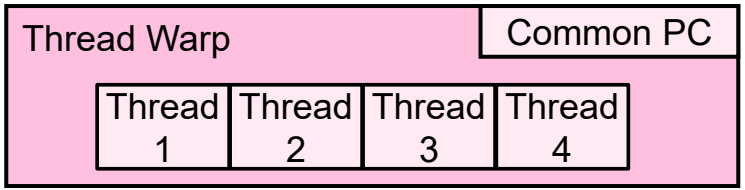
- Divergence-free mapping



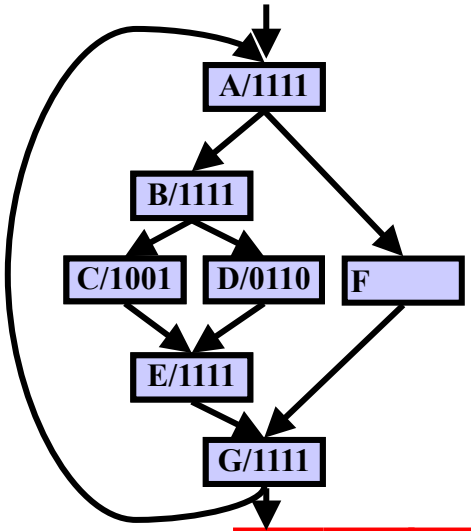
Threads Can Take Different Paths in Warp-based SIMD

- Each thread can have conditional control flow instructions
- Threads can execute different control flow paths



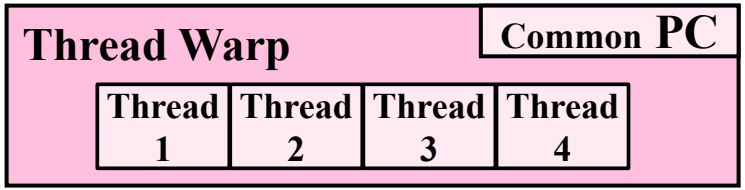


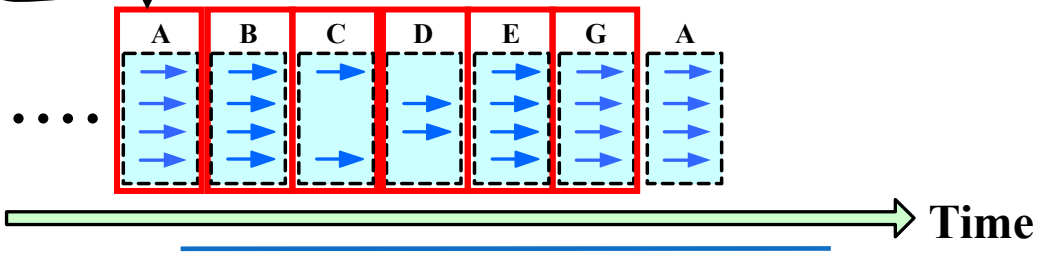
Warp Serialization



Stack

	Reconv. PC	Next PC	Active Mask
TOS →	-	E	1111
TOS →	E	D	0110
TOS →	E	E	1001



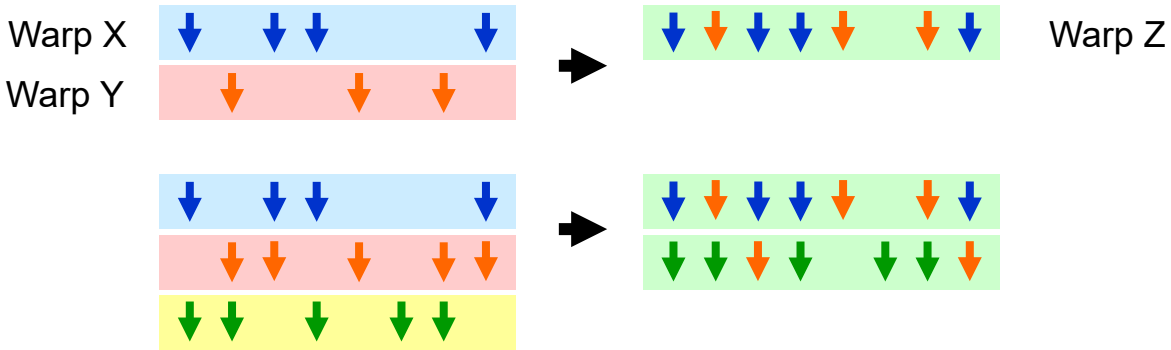


Each Thread Is Independent

- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing
- If we have many threads
- We can find individual threads that are at the same PC
- And, group them together into a single warp dynamically
- This reduces "divergence" → improves SIMD utilization
  - SIMD utilization: fraction of SIMD lanes executing a useful operation (i.e., executing an active thread)

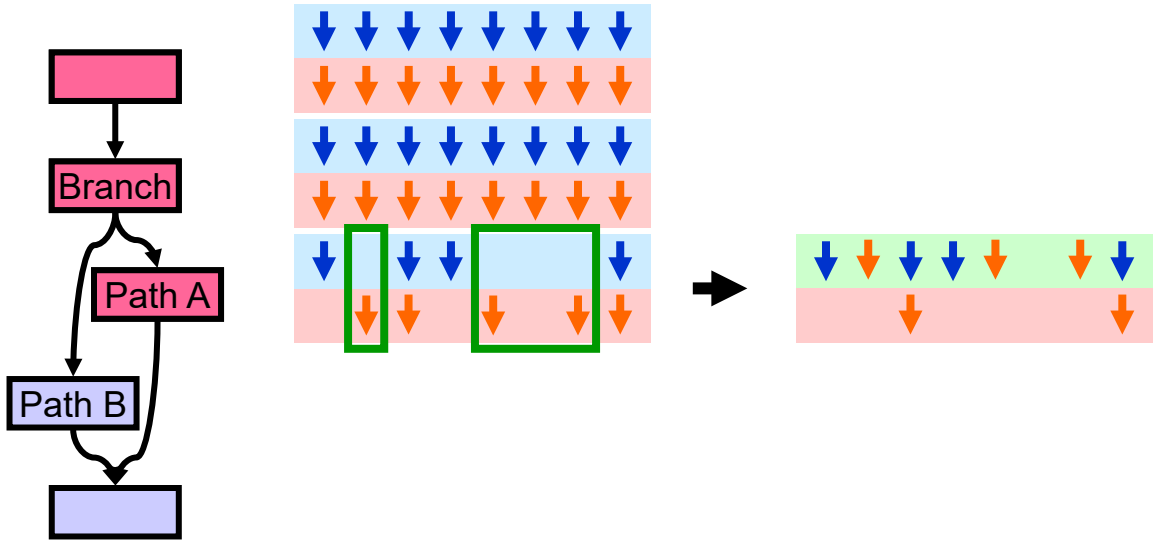
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)
- Form new warps from warps that are waiting
  - Enough threads branching to each path enables the creation of full new warps



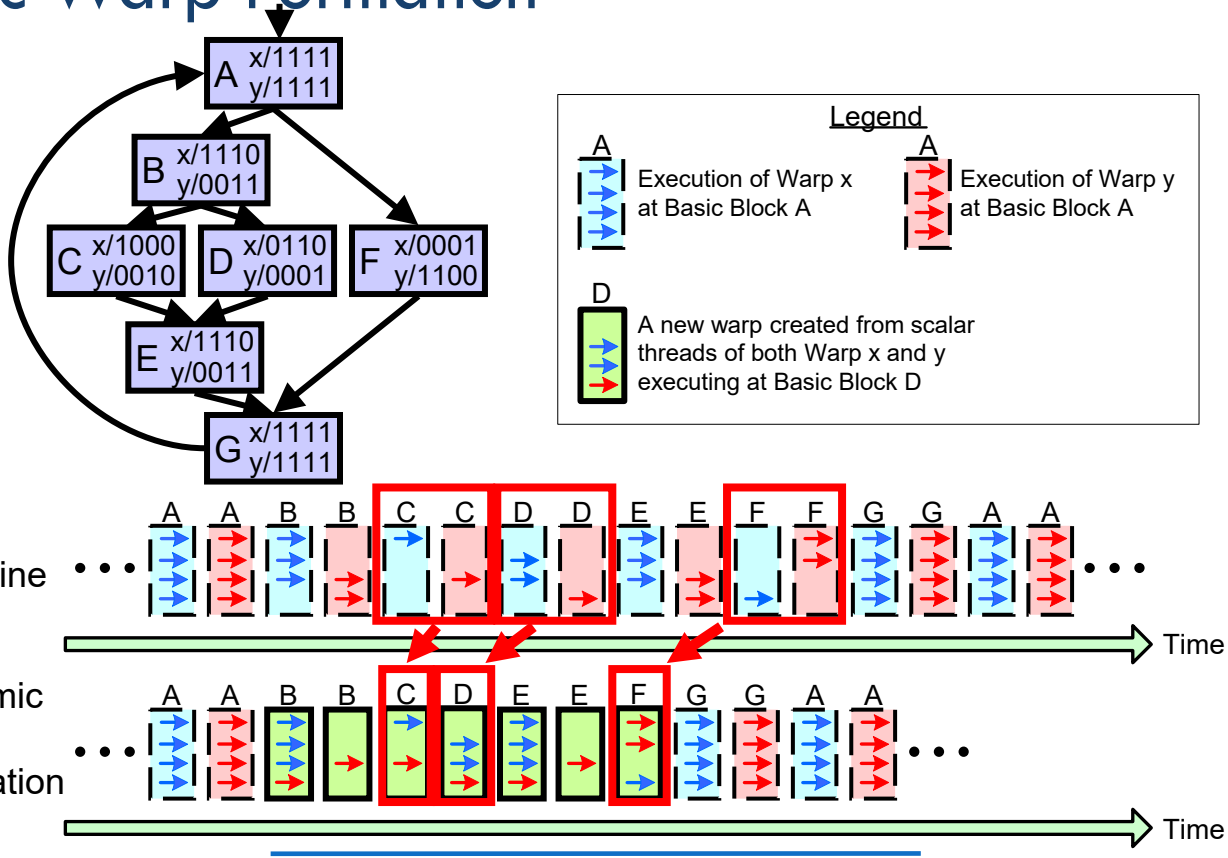
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)



- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

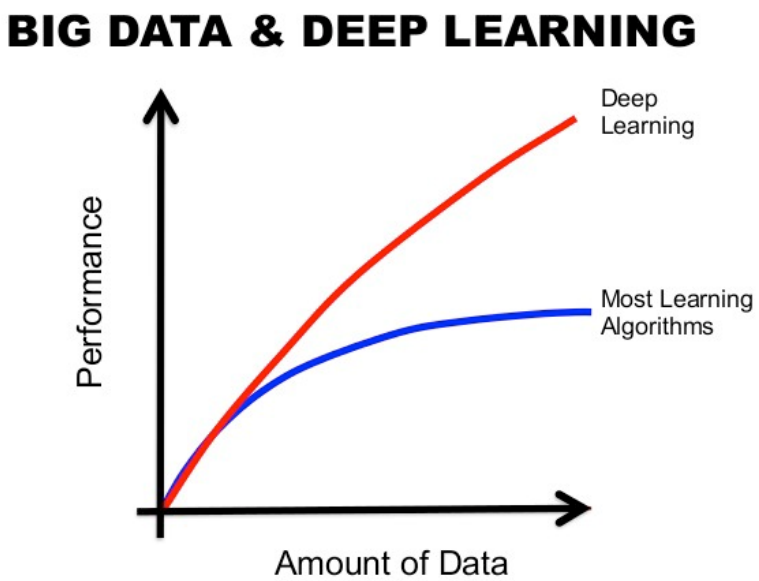
Dynamic Warp Formation



Caso de integración: Deep Learning usando GPUS con TensorFlow y Spark

Deep learning uses general learning algorithms

- The algorithms need to build the layers of an artificial neural network
  - Training data
- Processing this training data requires lots of computation
  - Convolutional NN -> Matrix multiplications



Source: https://towardsdatascience.com/7-practical-deep-learning-tips-97a9f514100e

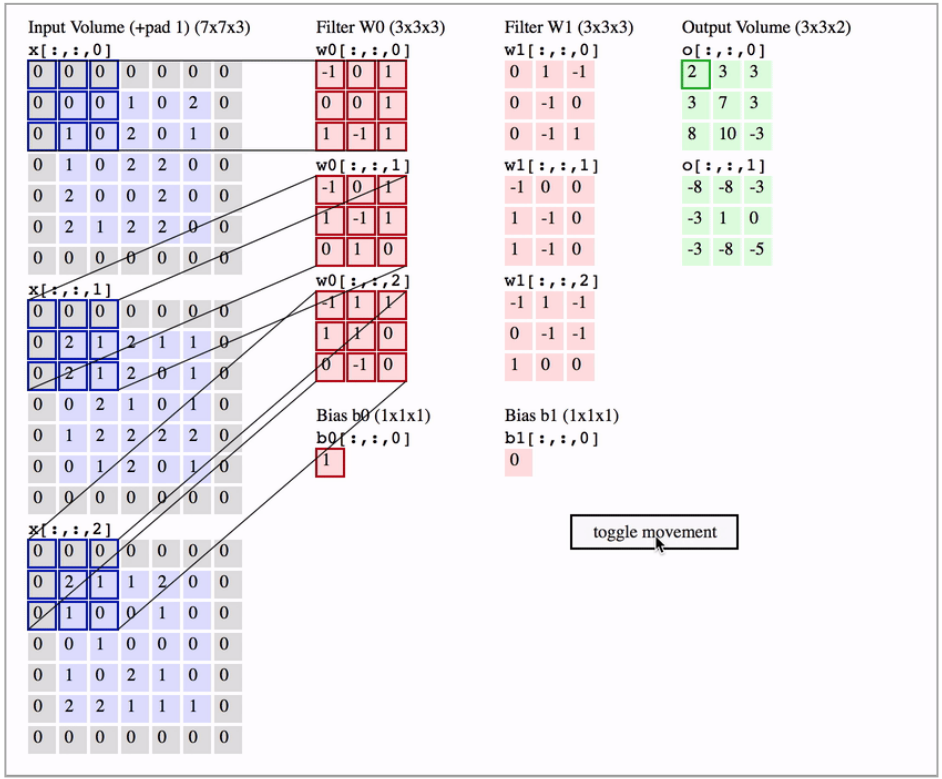
Deep Learning usando GPUs con TensorFlow y Spark

Se crea una red neuronal Convolucional

Operación basica:

$$\begin{matrix} & \vec{b_1} & \vec{b_2} \\ & \downarrow & \downarrow \\ \vec{a_1} \rightarrow \\ \vec{a_2} \rightarrow \end{matrix} \begin{bmatrix} 1 & 7 \\ 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} \vec{a_1} \cdot \vec{b_1} & \vec{a_1} \cdot \vec{b_2} \\ \vec{a_2} \cdot \vec{b_1} & \vec{a_2} \cdot \vec{b_2} \end{bmatrix}$$

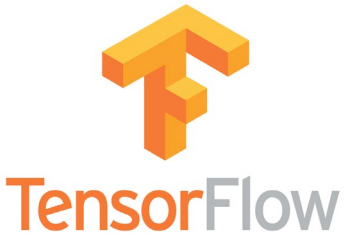
$A \qquad B \qquad C$



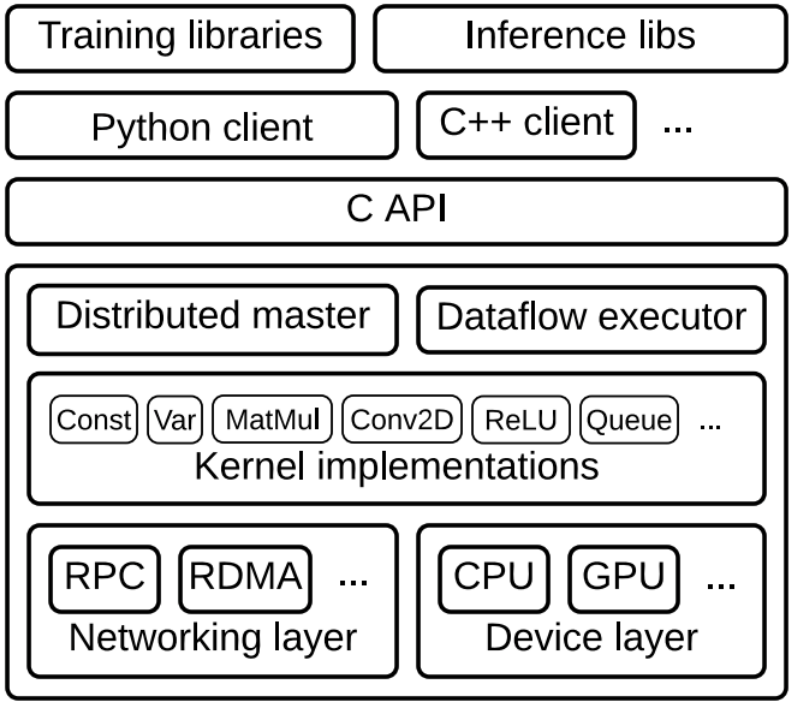
Source: https://medium.com/@phidaouss/convolutional-neural-networks-cnn-or-convnets-d7c688b0a207

Integración: Deep Learning usando GPUs con TensorFlow y Spark

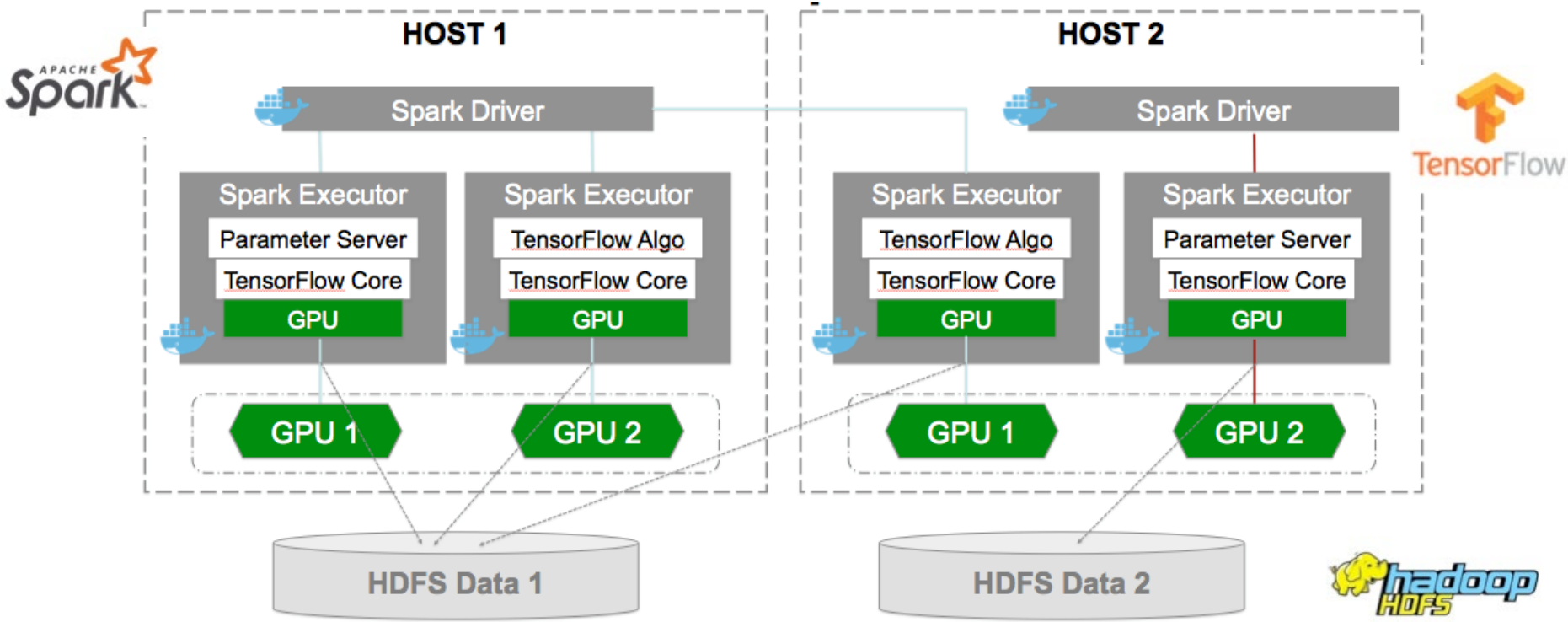
Arquitectura de TensorFlow



Source: www.tensorflow.org/extend/architecture



Deep Learning with TensorFlow and Spark



Supercomputación de sobremesa: 6 GPU Nvidia RTX

https://www.xataka.com/ordenadores/me-he-construido-bestia-seis-rtx-3090-no-para-minar-criptodivisas-sino-para-investigar-inteligencia-artificial

CPU: AMD Threadripper 1950X
- 8 canales acceso a mem
- 16 cores (32 hilos)

Gráficas: 6 x NVIDIA GeForce RTX 3090 con 24 GB mem GDDR6.

Memoria: 256 GB de RAM

Almacenamiento: 10 TB en unidades SSD M.2 NVMe