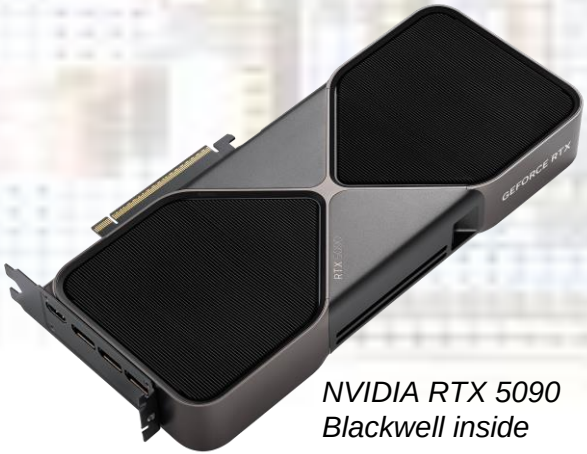
# Intelligent Architectures

5LIL0

## GPU: Graphics Processing Unit

*Exploiting massive parallelism*

*Running thousands of threads*




*NVIDIA RTX 5090*
*Blackwell inside*

**Henk Corporaal**

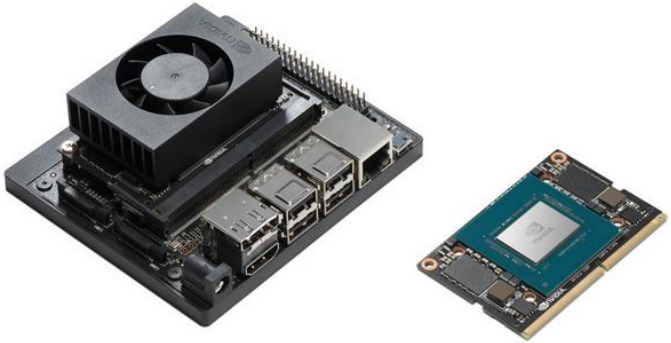
**Gert-Jan van de Braak**

**Zhenyu Ye**

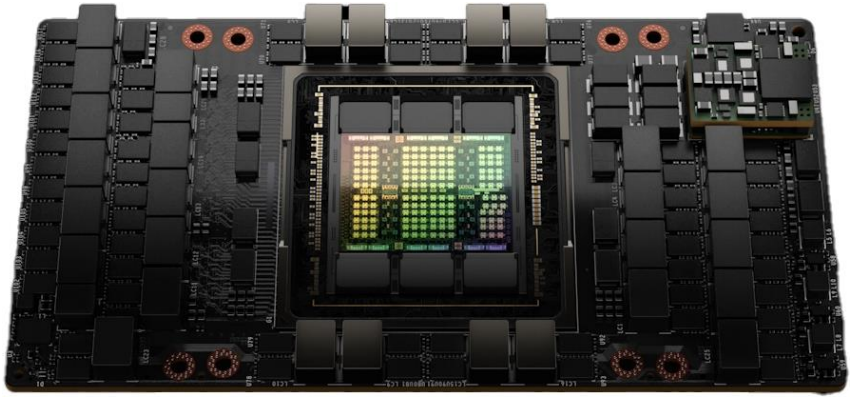
h.corporaal@tue.nl

TUEindhoven, March 2025

# In need of PetaFlops on your desk?



*NVIDIA Jetson Xavier NX (21 TOPS)*



*NVIDIA Hopper H100 (2000 TOPS = 2.0 PetaOPS)*

*Note: Flops ≠ Tops, and not every Op is the same!!*

# Today: **2 topics**

1. The architecture of GPUs
   - –Single Instruction Multiple Thread: SIMT
   - –Memory hierarchy
     - Global and Shared (=local)

2. The programming model of GPUs
   - –The threading hierarchy
     - Grids, Blocks, Warps and Treads
   - –SIMD vs SIMT
   - –Use of Memory Hierarchy


- We show fundamentals, illustrated with NVIDIA / CUDA

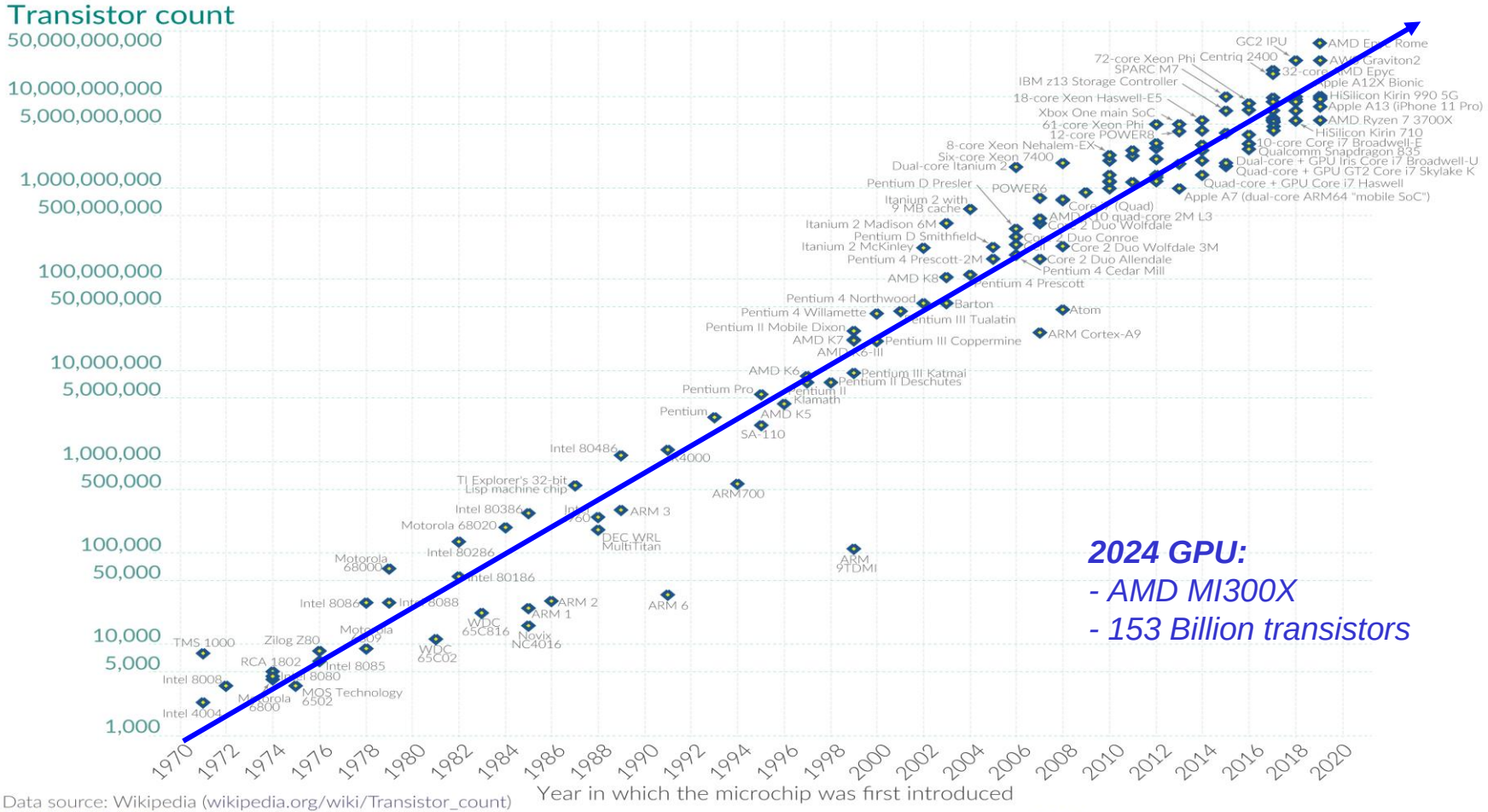
# Story line:

- **Introduction to GPUs**
  - **what to do with all these transistors**
- The programming model of GPUs
  - SIMD (vector) vs Thread-based processing
    - Single Instruction Multiple Thread: SIMT
  - Memory hierarchy
    - Global and Shared (=local)
    - How to use it efficiently
  - Warps, Scheduling, Register allocation
- Example NVIDIA GPU
- Hazards
  - Bank conflict
  - Divergence
- Performance model: Roofline
- Latest Developments

# Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.




Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

# Transistor Count of 3 types of architectures

*ref*: *http://en.wikipedia.org/wiki/Transistor_count*

Multi-core CPU

| **Processor** | **Transistors** | **Tech. Node** |
|---|---|---|
| Apple ARM M4 Pro | 38,000,000,000 | 3nm, TSMC |

GPU

| **Processor** | **Transistors** | **Tech. Node** |
|---|---|---|
| Nvidia GH100 Hopper<br>AMD MI300X | 80,000,000,000<br>153,000,000,000 | 4nm<br>5/6nm, TSMC |

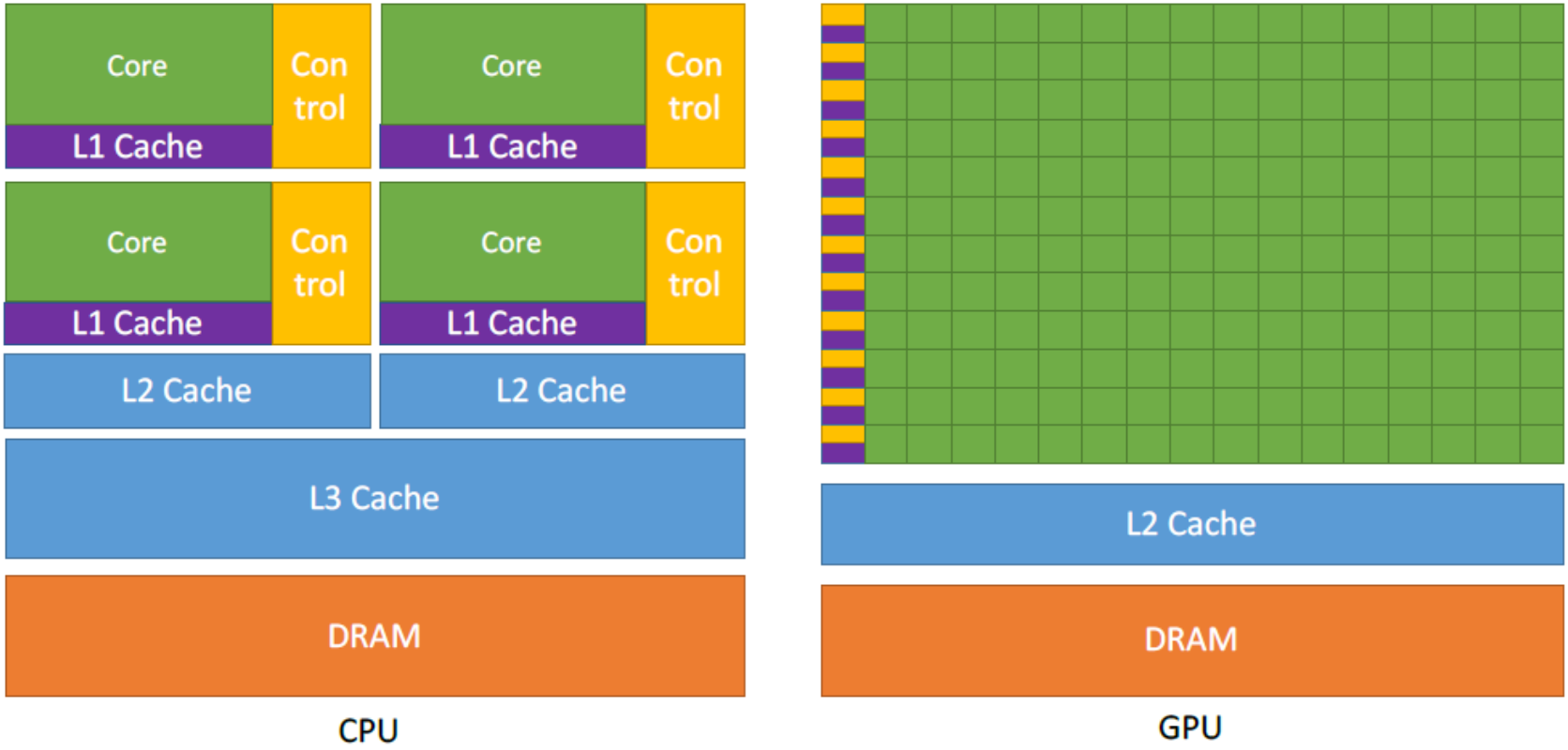
FPGA

| **FPGA** | **Transistors** | **Tech. Node** |
|---|---|---|
| Versal VP1802 (2021) | 92,000,000,000+ | 7nm |

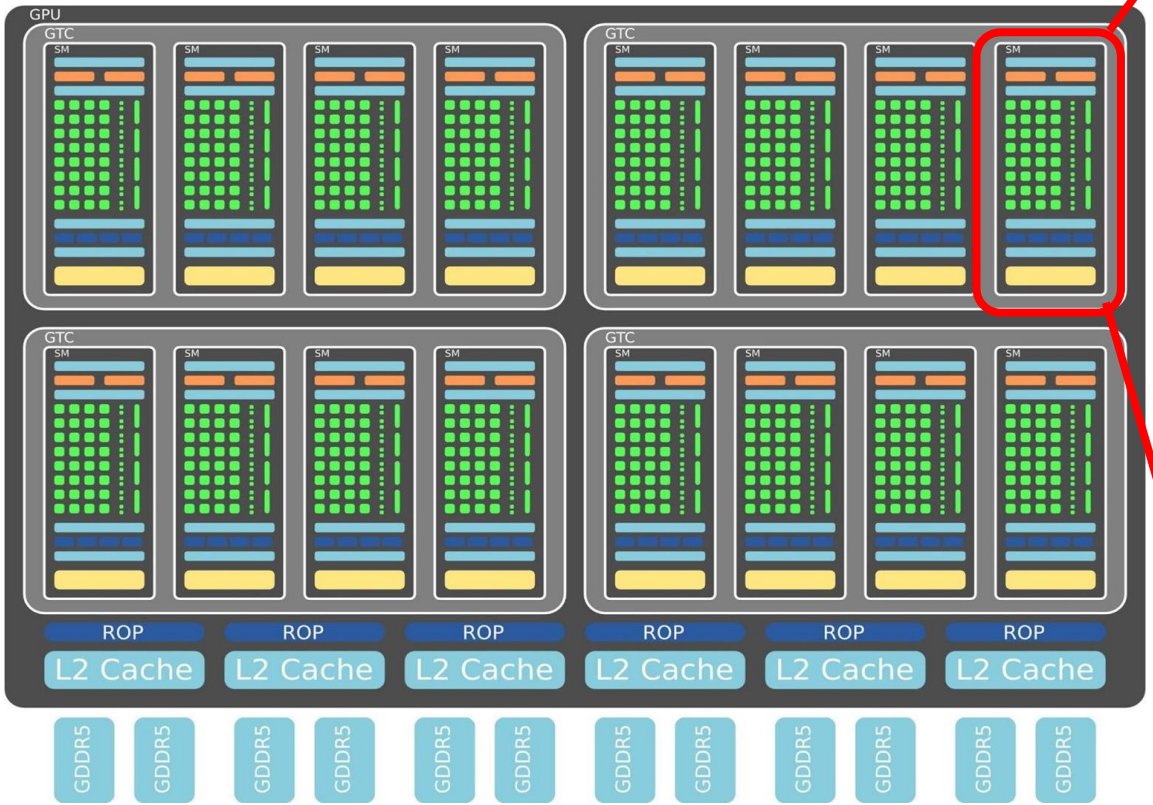
Historical reference

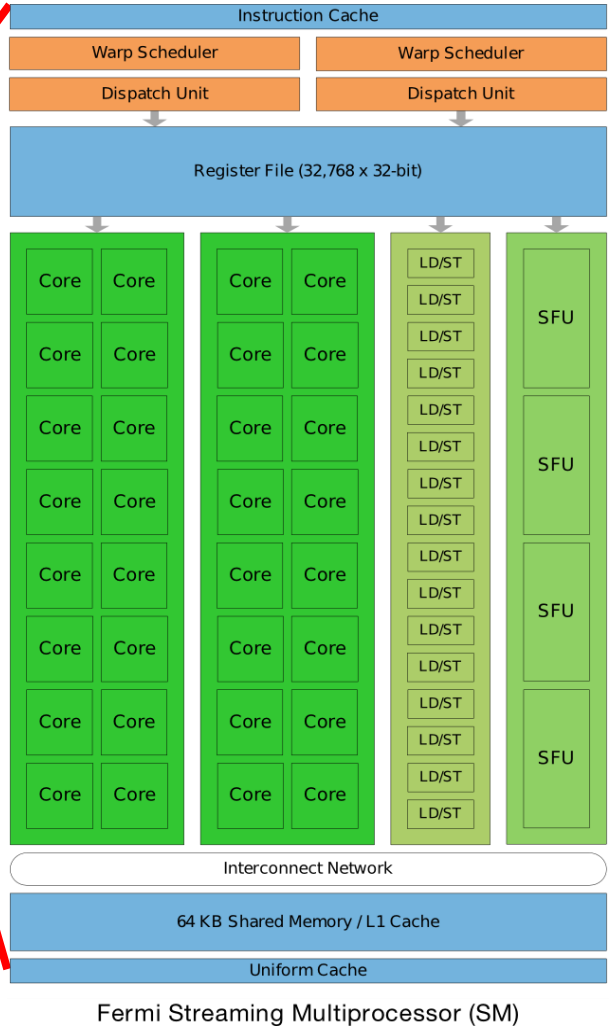
| **Processor (1971)** | **Transistor** | **Tech. Node** |
|---|---|---|
| Intel 4004, **first** single die (4-bit) micro-processor | 2250 | 10,000nm = 10 μm |

# CPU vs GPU: where are these transistors going?



# NVIDIA (Fermi Arch., 2010)




Fermi Streaming Multiprocessor (SM)

ref: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

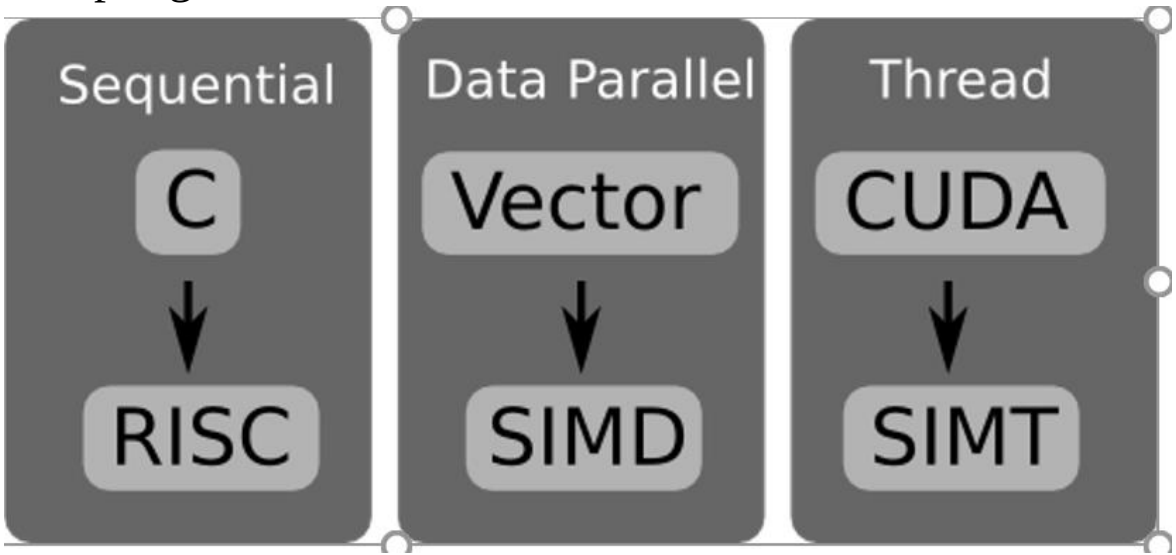
# Story line:

- Introduction to GPUs
  - what to do with all these transistors
- **The programming model of GPUs**
  - **SIMD (Vector) vs Thread-based processing**
  - **Single Instruction Multiple Thread: SIMT**
  - **Thread hierarchy, thread scheduling**
  - Memory hierarchy
  - Warps, Scheduling, Register allocation
- Example NVIDIA GPUs
- Hazards
  - Bank conflict
  - Divergence
- Performance model: Roofline
- Latest Developments

# 3 Programming models

1. Sequential
2. SIMD: Single Instruction Multiple Data ~= Vector processing
3. SIMT: Single Instruction Multiple Threads

• Note: no strict coupling, but some models fit better to a certain architecture

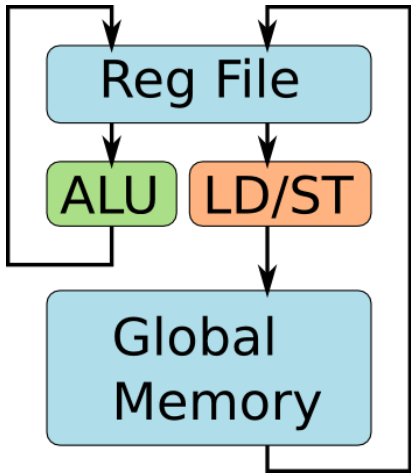


# Let's start with C mapped to a RISC processor

```
int A[2][4];
for(i=0;i<2;i++){
    for(j=0;j<4;j++){
        A[i][j]++;
    }
}
```

Assembly code of inner-loop

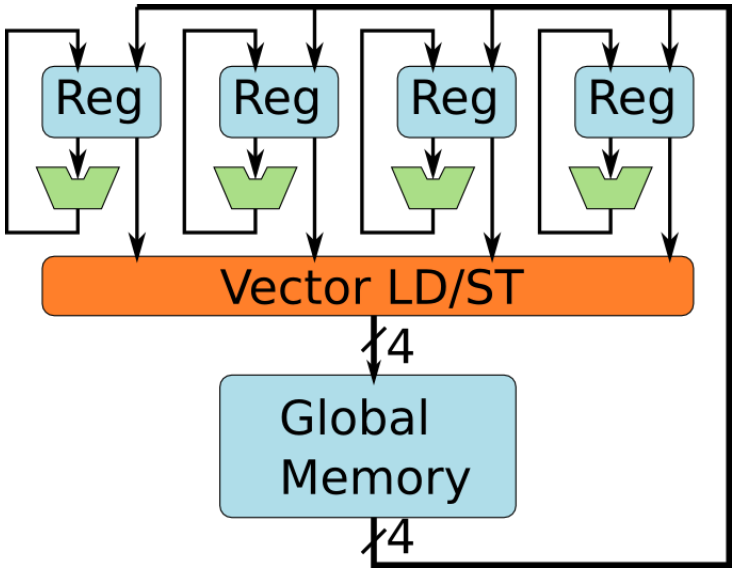
```
lw      r0, 4(r1)
addi    r0, r0, 1
sw      r0, 4(r1)
```




Programmer's view of RISC
- 1 operation / cycle

# Most CPUs Have Vector SIMD Units

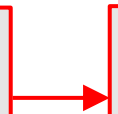
- Programmer's view of a vector SIMD:
  - e.g. as used in SSE (= Streaming SIMD Extensions)



# Let's Program the Vector SIMD

## Unroll inner-loop to vector operation

```
int A[2][4];
for(i=0;i<2;i++){
    for(j=0;j<4;j++){
        A[i][j]++;
    }
}
```

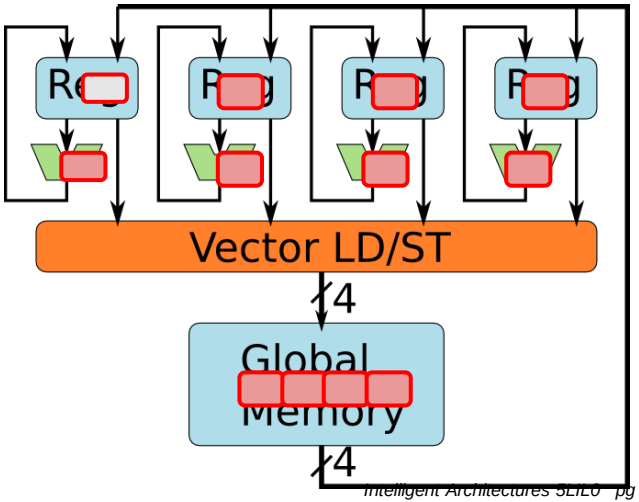


```
int A[2][4];
for(i=0;i<2;i++){
    movups  xmm0,        [ &A[i][0] ]  // load
    addps   xmm0,        xmm1          // add 1
    movups  [ &A[i][0] ], xmm0         // store
}
```

Looks like the previous example,
but each SSE instruction executes on 4 ALUs

# How Do Vector Programs Run?

```
int A[2][4];
for(i=0;i<2;i++){
    movups xmm0,        [ &A[i][0] ]  // load
    addps    xmm0,      xmm1          // add 1
    movups [ &A[i][0] ], xmm0         // store
}
```

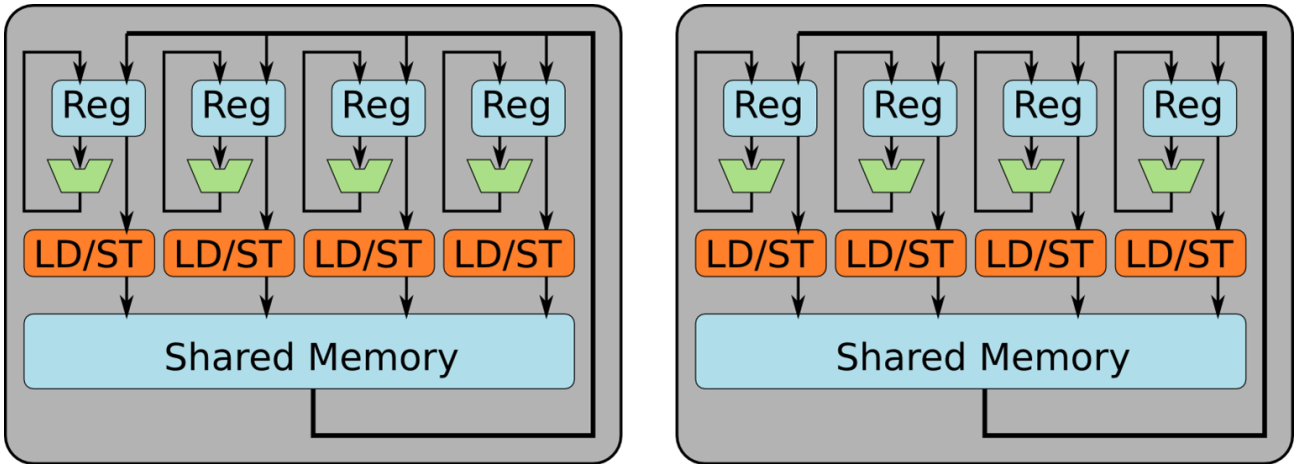


# CUDA Programmer's View of GPUs

A GPU contains multiple SIMD Units

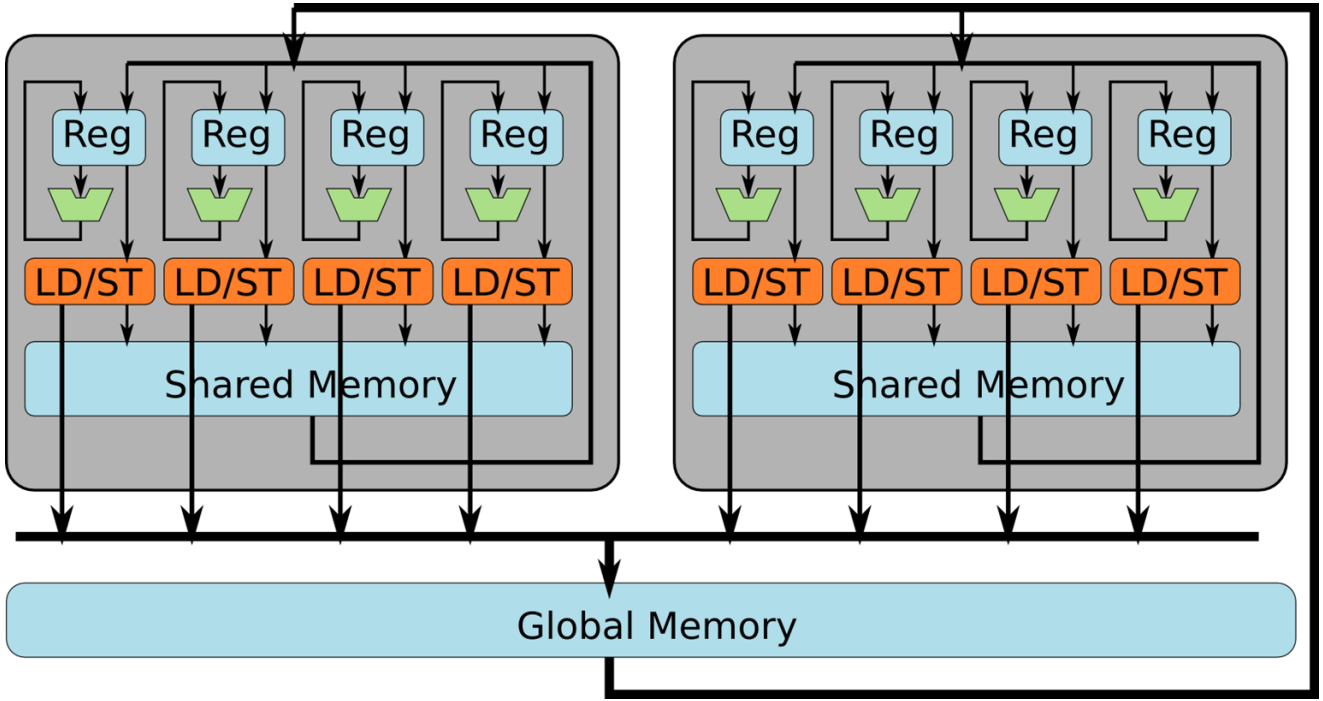
- called Stream Multiprocessors (SMs) by NVIDIA

- e.g. 2 SMs:



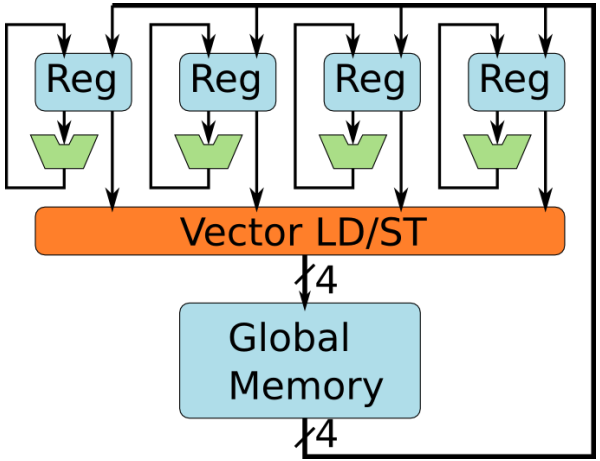
# CUDA Programmer's View of GPUs

- A GPU contains multiple SIMD Units, each having local (shared by PEs) memory
- All of them can access global memory

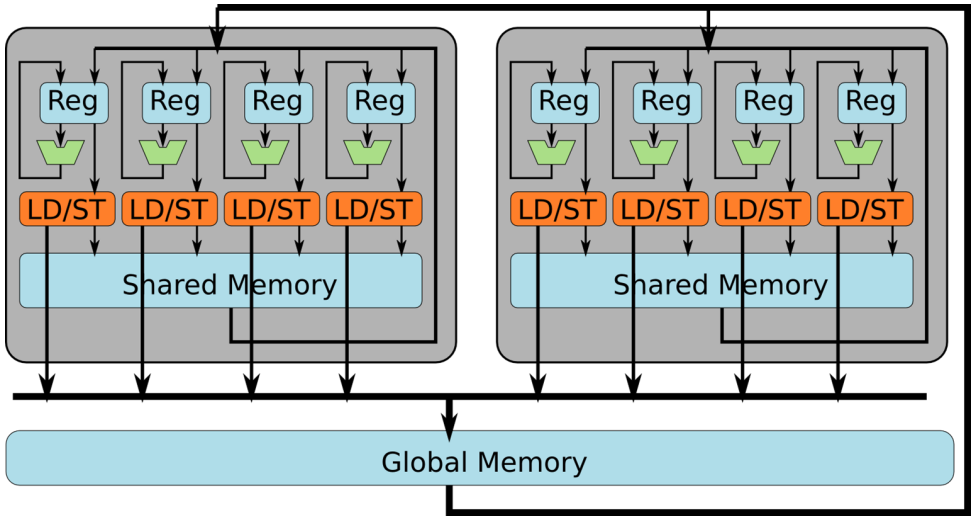


# What Are the Differences?

SSE




GPU



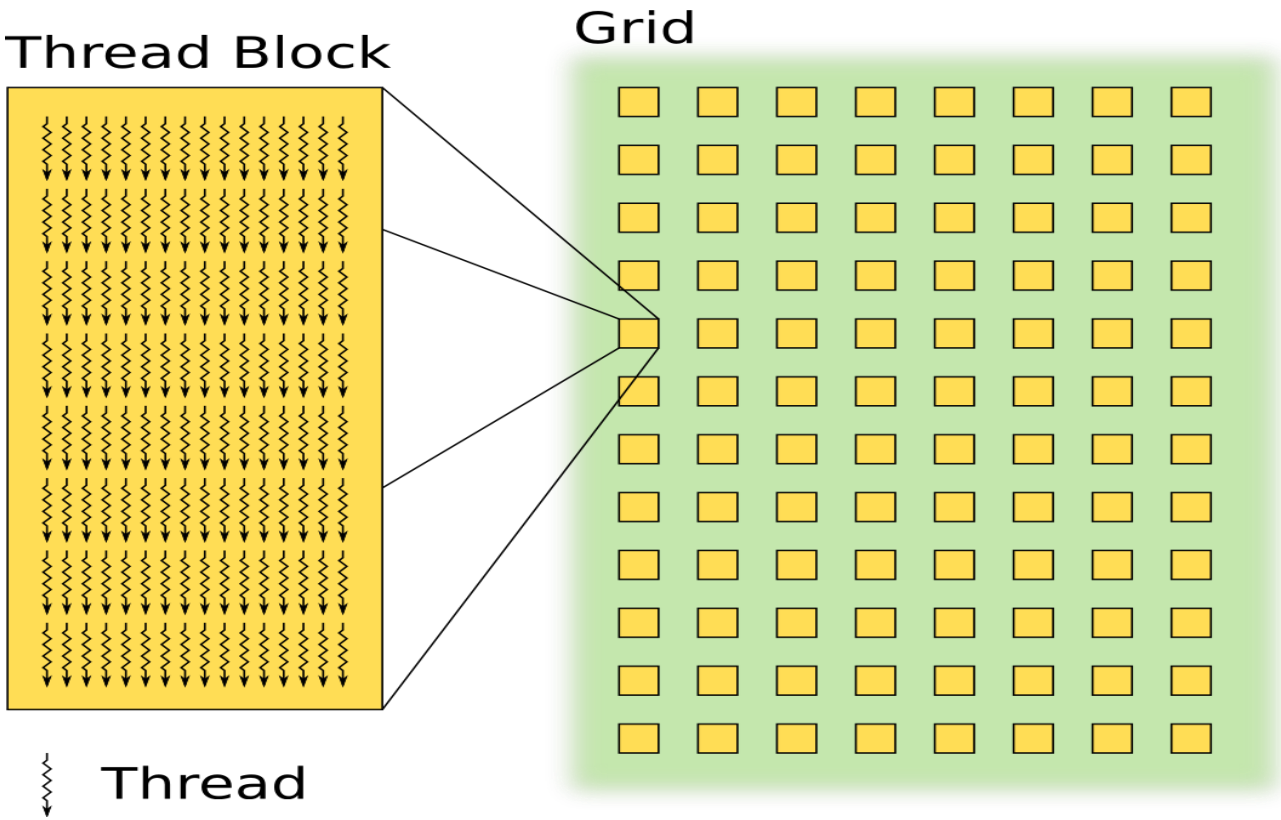

2 important differences:

1. GPUs use threads instead of vectors;
   - each thread runs on a single lane !
2. GPUs have the "Shared Memory" spaces
   - Note: the GPU above can run 2 ‘groups’ of 4 threads each in parallel

# Thread Hierarchy in CUDA

Grid contains
- Thread Blocks

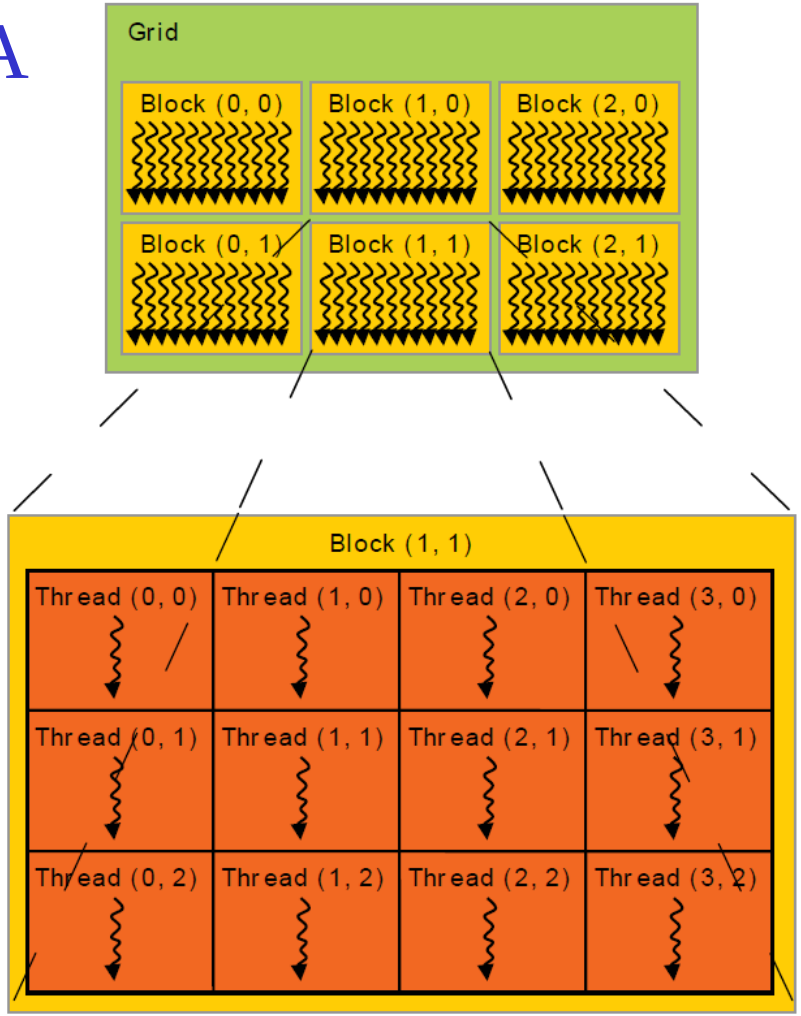
Thread Block contains
- Threads



# Thread Hierarchy in CUDA

- Upto **3** grid dimensions
- Upto **3** block dimensions

- Example: kernelF**<<<(3,2),(4,3)>>>**
  *Grid Block*



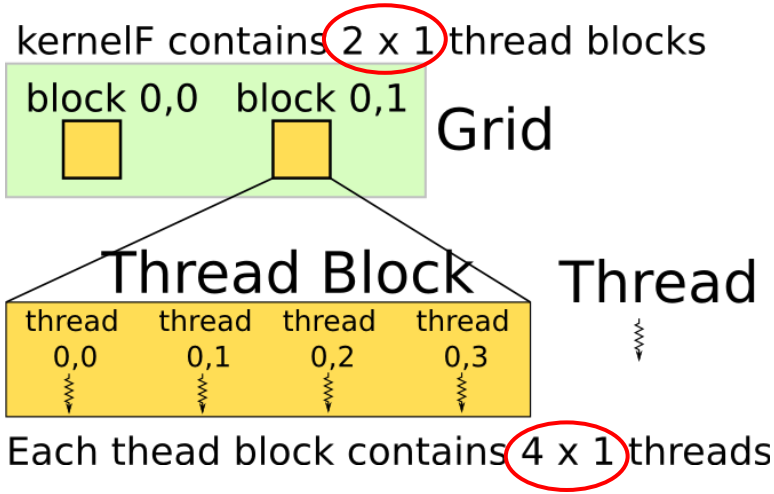
# From C to CUDA

```
int A[2][4];
for(i=0;i<2;i++)
    for(j=0;j<4;j++)
        A[i][j]++;
```

convert into CUDA

```
int A[2][4];
kernelF<<<(2,1),(4,1)>>>(A);   // define 2x4=8 threads
__device__    kernelF(A){      // all threads run same kernel
    i = blockIdx.x;            // each thread block has its id
    j = threadIdx.x;           // each thread has its own id
    A[i][j]++;                 // each thread performs 1 increment
}
```

# Thread Hierarchy



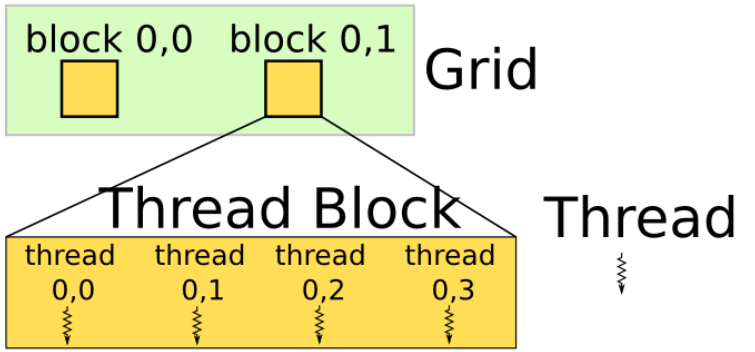

Example:
thread 3 of block 1 operates
on element A[1][3]

```
int A[2][4];
kernelF<<<(2,1),(4,1)>>>(A);
__device__     kernelF(A){
    i = blockIdx.x;
    j = threadIdx.x;
    A[i][j]++;
}
```

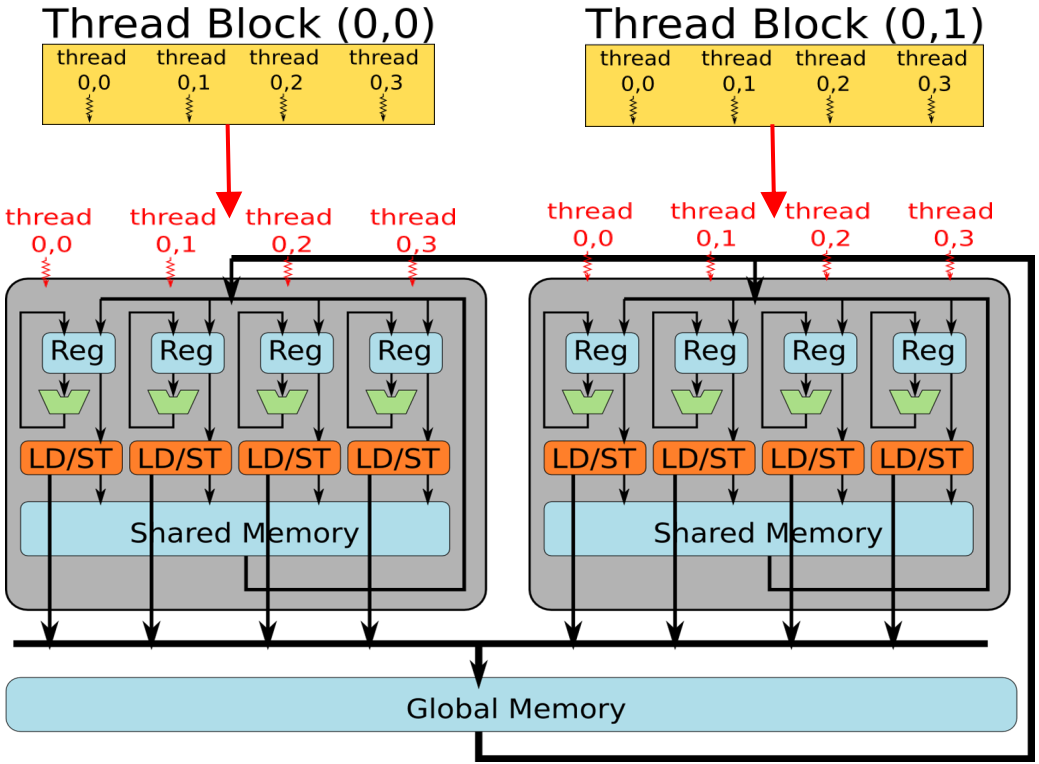
// define 2x4=8 threads
// all threads run same kernel
// each thread block has its id
// each thread has its id
// each thread has different i and j

# How Are Threads and Thread Blocks Scheduled?

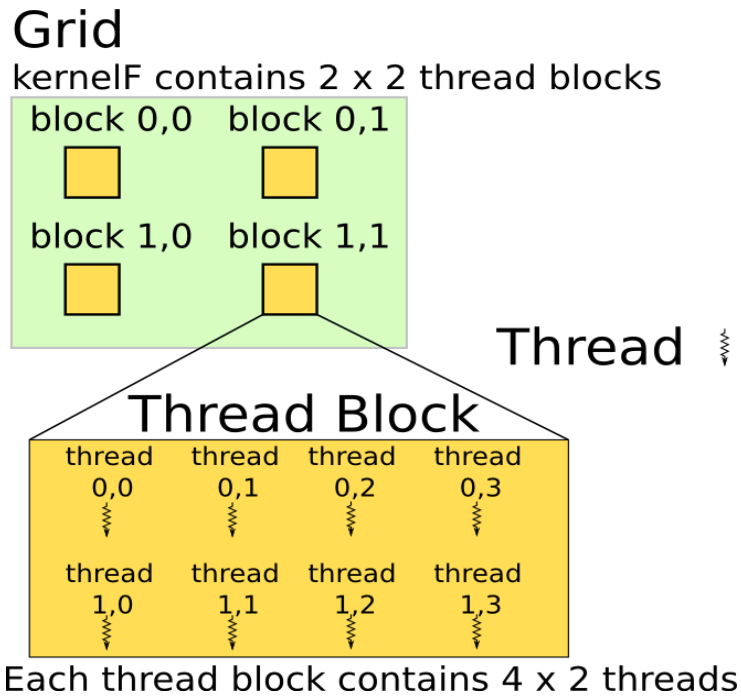
kernelF contains 2 x 1 thread blocks

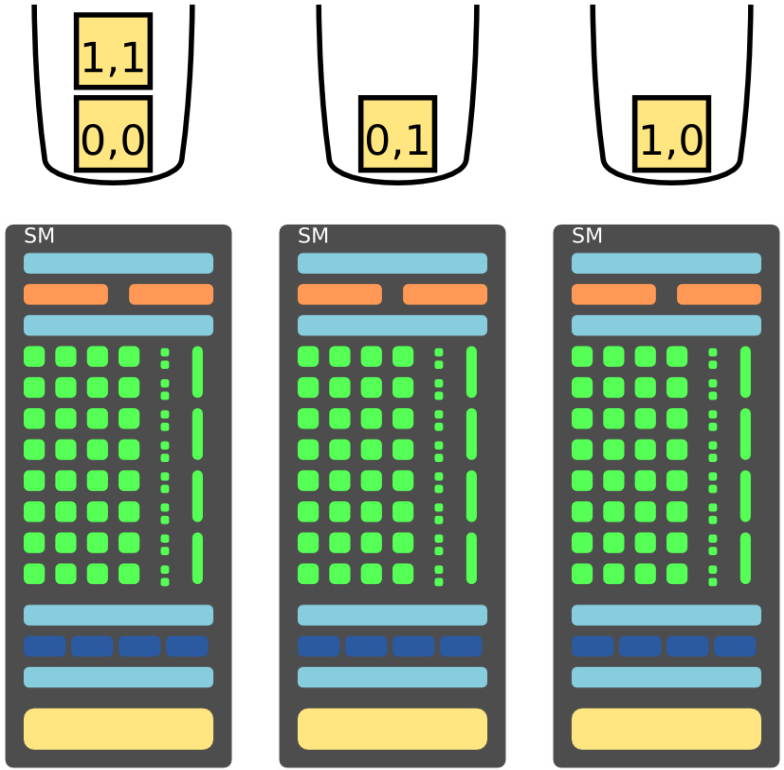



Each thead block contains 4 x 1 threads



# Blocks Are **Dynamically** Scheduled
## - 4 blocks on 3 SMs

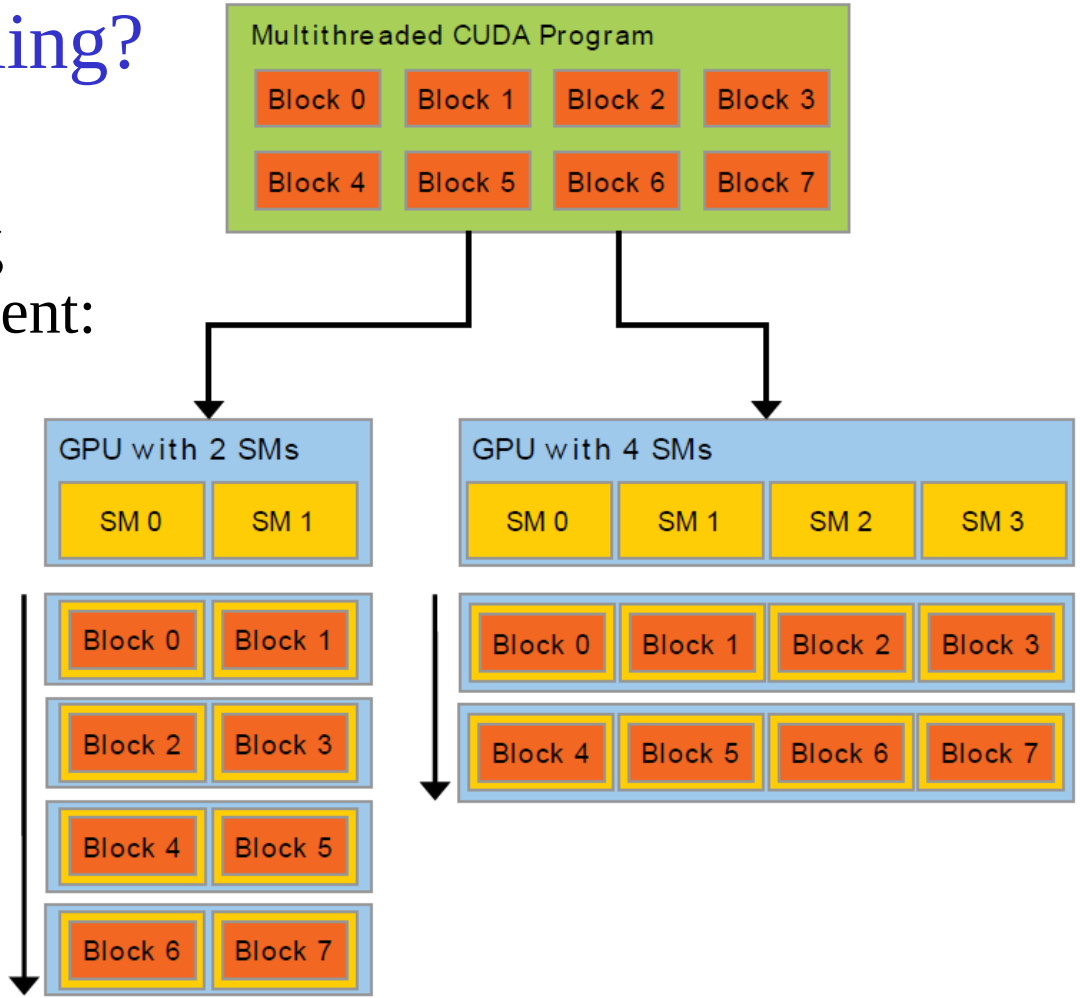


# Why dynamic scheduling?

- Dynamic block scheduling allows to be HW independent:
  - same program runs on GPUs with different # of SMs

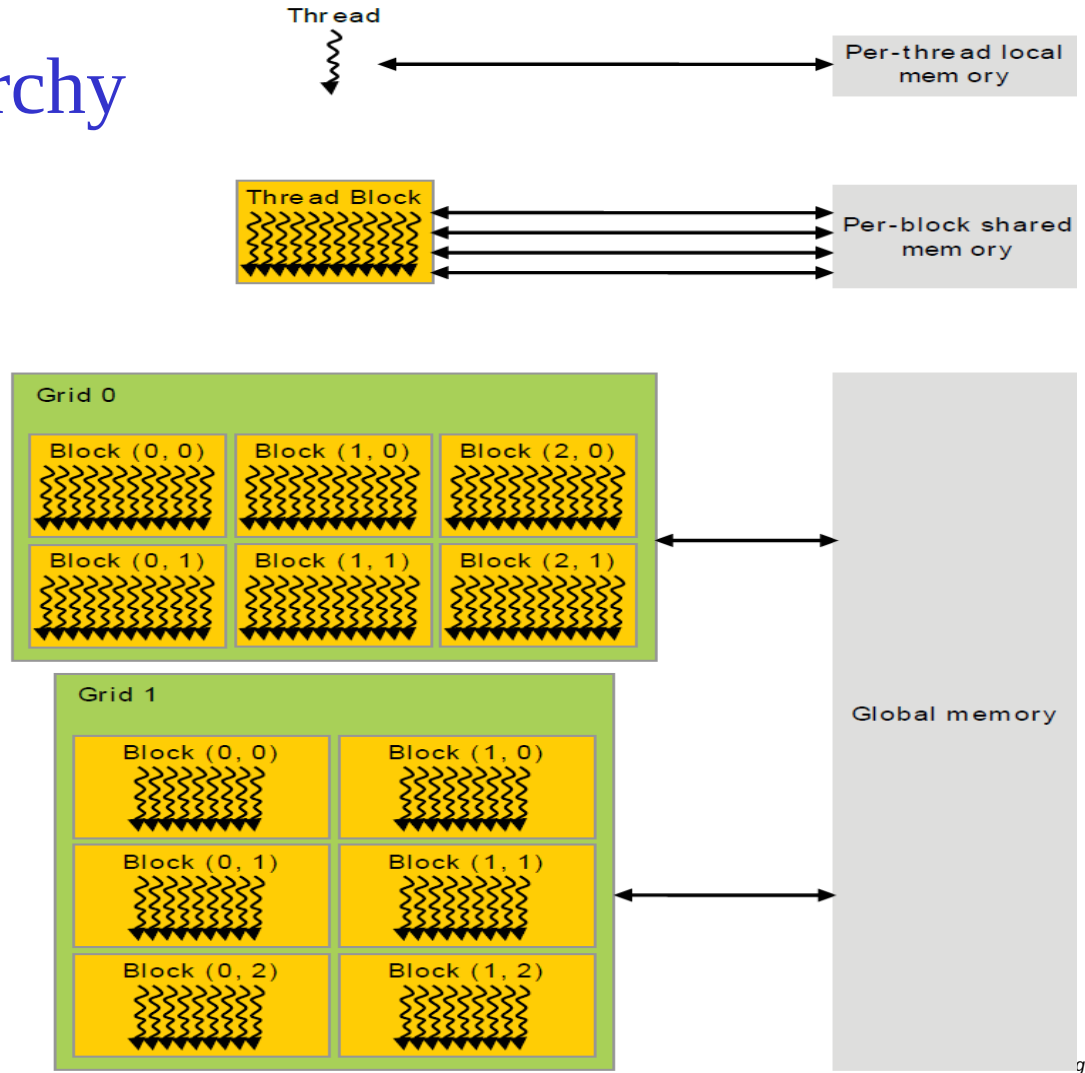
⇒ Scalability & HW agnostic



# Story line:

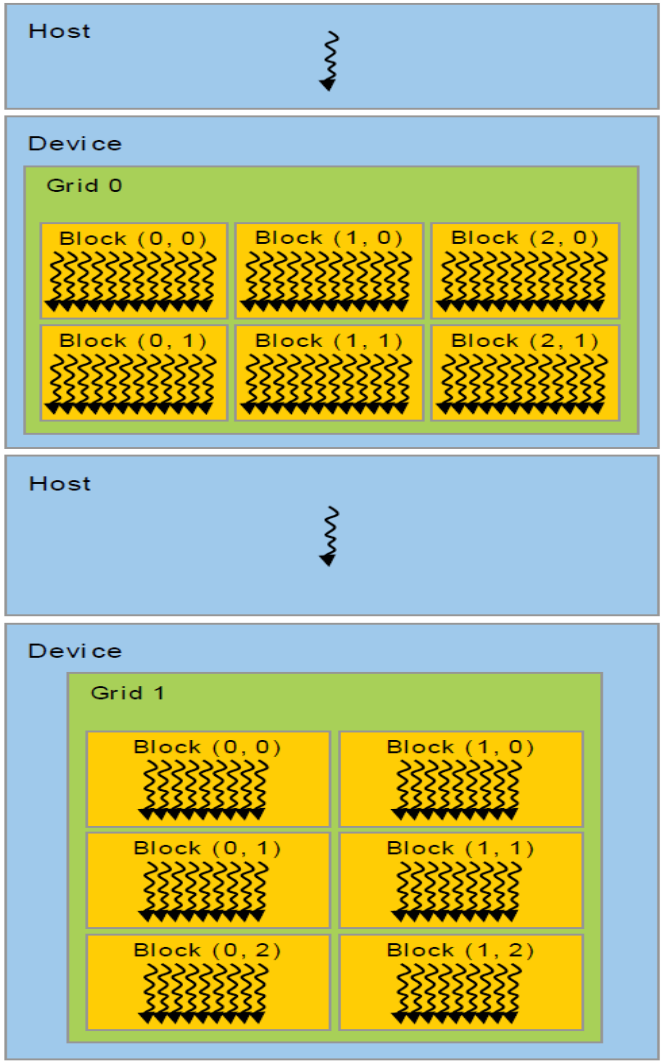
- Introduction to GPUs
  - what to do with all these transistors
- The programming model of GPUs
  - SIMD (vector) vs Thread-based processing
  - **Memory hierarchy**
    - **Global and Shared (=local)**
    - **How to use it efficiently**
  - Warps, Scheduling, Register allocation
- Example NVIDIA GPUs
- Hazards
  - Bank conflict
  - Divergence
- Performance model: Roofline
- Latest developments

# Memory Hierarchy



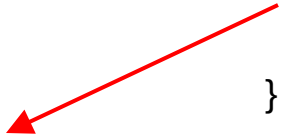
# Host vs Device code

```
Serial code

Parallel kernel
Kernel0<<<>>>()

Serial code

Parallel kernel
Kernel1<<<>>>()
```



# How does GPU assembly look? Translate CUDA

```
int A[2][4];
kernelF<<<(2,1),(4,1)>>>(A);
__device__ kernelF(A){
    i = blockIdx.x;
    j = threadIdx.x;
    A[i][j]++;
}
```



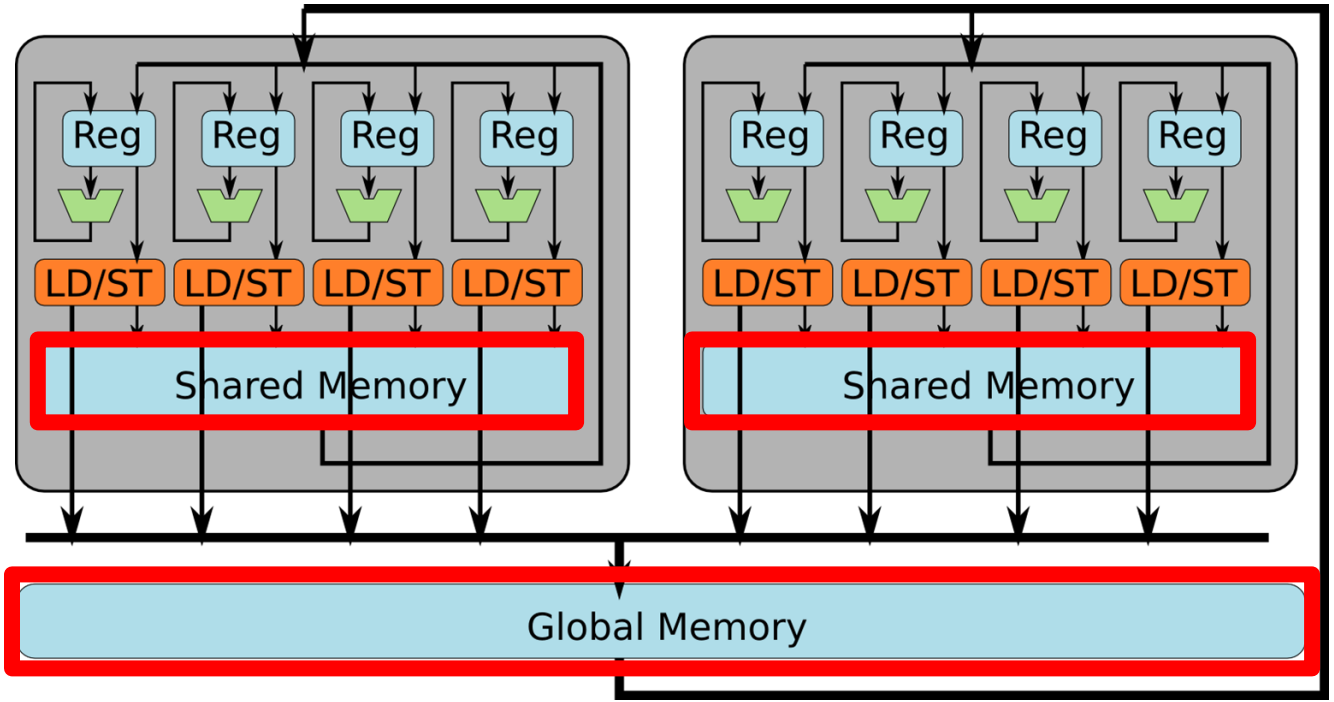
```
mv.u32        %r0, %ctaid.x          // r0 = i = blockIdx.x
mv.u32        %r1, %ntid.x           // r1 = "threads-per-block"
mv.u32        %r2, %tid.x            // r2 = j = threadIdx.x
mad.u32       %r3, %r2, %r1, %r0     // r3 = i * "threads-per-block" + j
ld.global.s32 %r4, [%r3]             // r4 = A[i][j] = Mem[r3]
add.s32       %r4, %r4, 1            // r4 = r4 + 1
st.global.s32 [%r3], %r4             // A[i][j] = r4
```

# Utilizing Memory Hierarchy

Memory access latency

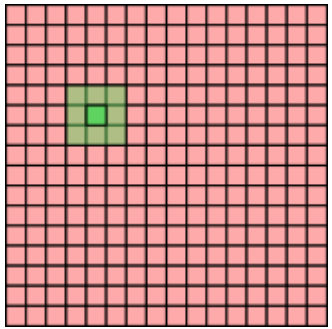
several cycles

100+ cycles



# Example: Average Filters

Average over a 3x3 window for a 16x16 array

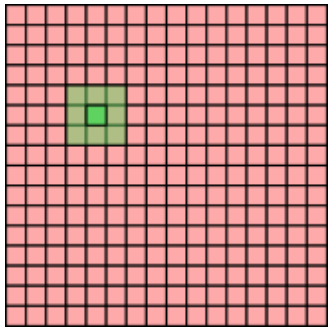


```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    i = threadIdx.y;
    j = threadIdx.x;
    tmp = (A[i-1][j-1] + A[i-1][j] +
                … + A[i+1][j+1] ) / 9;
          A[i][j] = tmp;
}
```

Each thread loads 9 elements from global memory.
It takes hundreds of cycles.

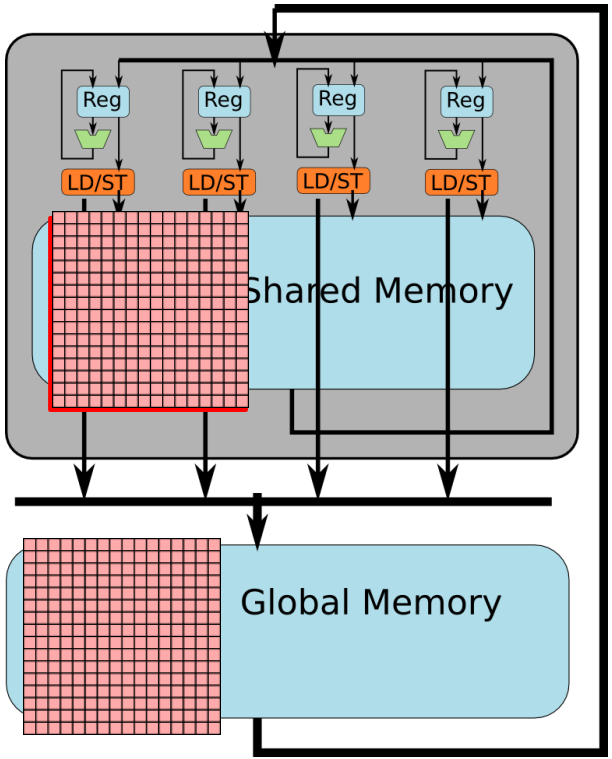
# Utilizing the Shared Memory

Average over a 3x3 window for a 16x16 array



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ int smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                    … + smem[i+1][j+1] ) / 9;
}
```

# Utilizing the Shared Memory




```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][j+1] ) / 9;
}
```

allocate shared mem

Each thread loads one element from global memory

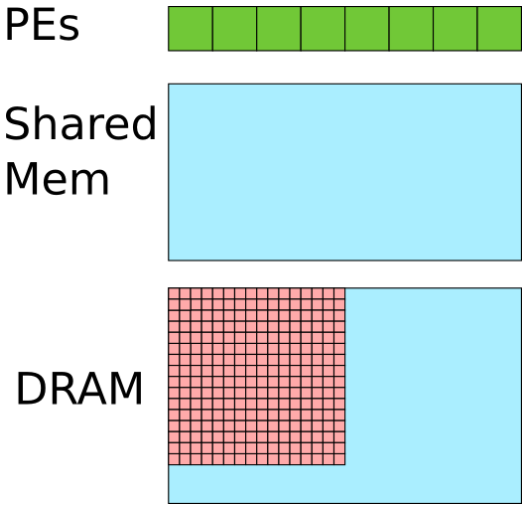
# However, the Program Is Incorrect

Hazards!

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][j+1] ) / 9;
}
```

# Let's See What's Wrong

Assume 256 threads are scheduled on 8 PEs.

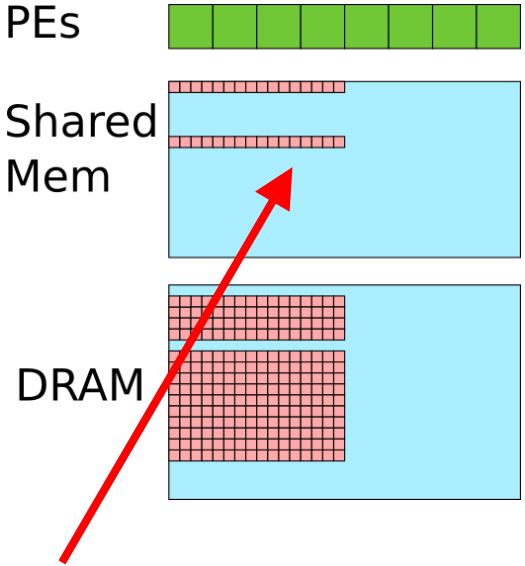



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][j+1] ) / 9;
}
```

Before load instruction

# Let's See What's Wrong

Assume 256 threads are scheduled on 8 PEs.

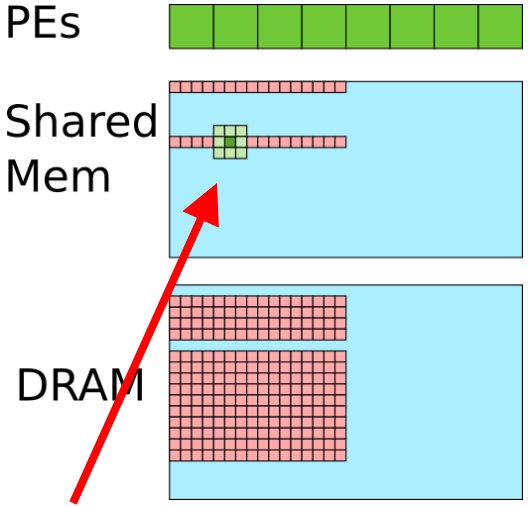



Some threads finish the load earlier than others.

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                    … + smem[i+1][j+1] ) / 9;
}
```

# Let's See What's Wrong

Assume 256 threads are scheduled on 8 PEs.

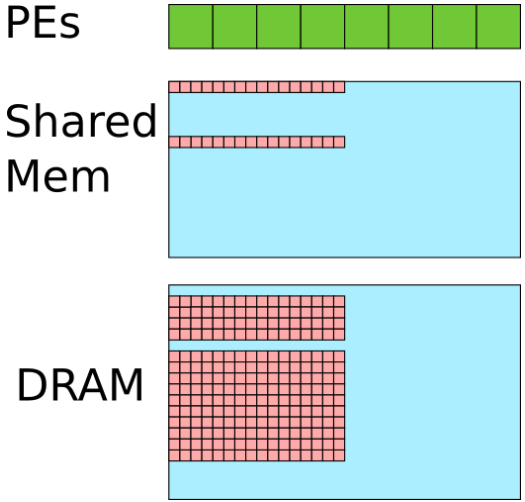



Some elements in the window are not yet loaded by other threads. Error!

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][j+1] ) / 9;
}
```

# How To Solve It? …. Use a "SYNC" barrier

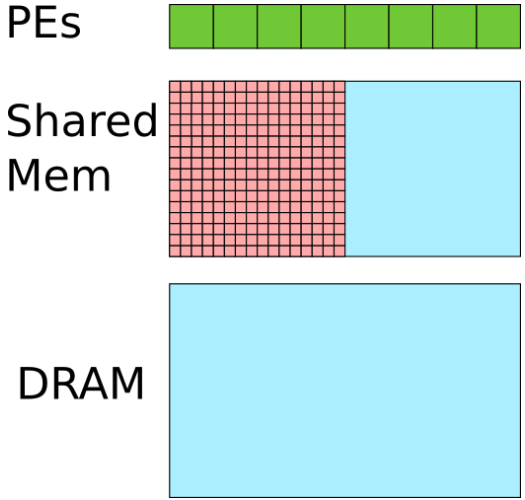
Assume 256 threads are scheduled on 8 PEs.




```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __SYNC();
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                    … + smem[i+1][j+1] ) / 9;
}
```

# Use a "SYNC" barrier

Assume 256 threads are scheduled on 8 PEs.

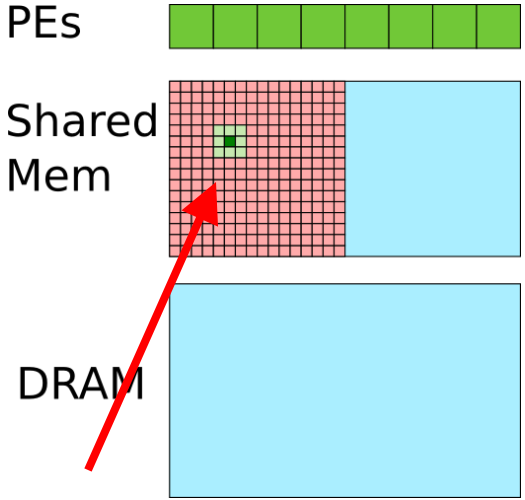



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __SYNC();
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][j+1] ) / 9;
}
```

Wait until all threads hit barrier

# Use a "SYNC" barrier

Assume 256 threads are scheduled on 8 PEs.




All elements in the window are loaded when each thread starts averaging.

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __SYNC();
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][j+1] ) / 9;
}
```

# Review what we have learned so far

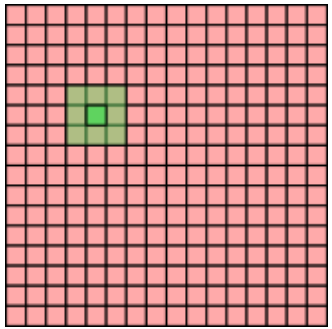

1. Single Instruction Multiple Thread (SIMT)
2. Shared memory

Q: What are the pros and cons of explicitly managed memory?

Q: What are the fundamental differences between SIMT and vector SIMD programming model?

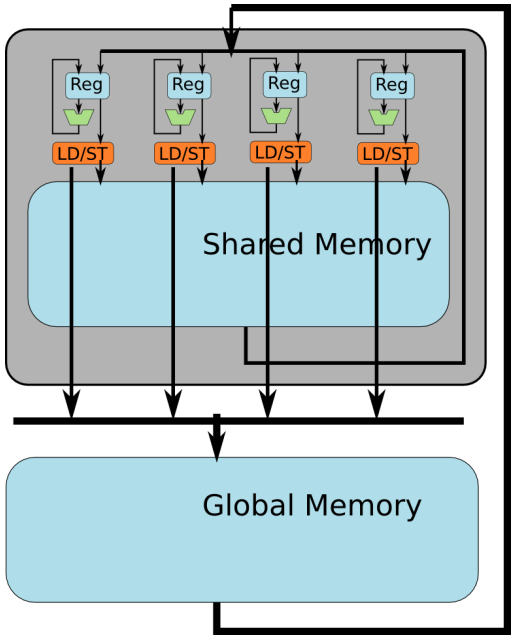
# Take the same example again

Average over a 3x3 window for a 16x16 array



Assume vector SIMD and SIMT both have shared memory

- What are the differences?



# Vector SIMD v.s. SIMT

```
int A[16][16]; // A in global memory
__shared__ int B[16][16]; // B local copy in shared mem

for(i=0;i<16;i++){
   for(j=0;j<16;j+=4){
      movups xmm0, [ &A[i][j] ]
      movups [ &B[i][j] ], xmm0 }}

for(i=0;i<16;i++){
   for(j=0;j<16;j+=4){
      addps xmm1, [ &B[i-1][j-1] ]
      addps xmm1, [ &B[i-1][j] ]
      … divps xmm1, 9 }}

for(i=0;i<16;i++){
   for(j=0;j<16;j+=4){
      addps [ &A[i][j] ], xmm1 }}
```

(1)

(2)

(3)

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
   __shared__ int smem[16][16];
   i = threadIdx.y;
   j = threadIdx.x;
   smem[i][j] = A[i][j]; // load to smem
   __sync(); // threads wait at barrier
   A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
    (3)       … + smem[i+1][j+1] ) / 9;
}
```

(1)

(2)

(1) load to shared mem
(2) compute
(3) store to global mem

# Vector SIMD v.s. SIMT

```
int A[16][16]; // A in global memory
__shared__ int B[16][16]; // B local copy in shared mem

for(i=0;i<16;i++){
   for(j=0;j<16;j+=4){
      movups xmm0, [ &A[i][j] ]
      movups [ &B[i][j] ], xmm0 }}

for(i=0;i<16;i++){
   for(j=0;j<16;j+=4){
      addps xmm1, [ &B[i-1][j-1] ]
      addps xmm1, [ &B[i-1][j] ]
      … divps xmm1, 9 }}

for(i=0;i<16;i++){
   for(j=0;j<16;j+=4){
      addps [ &A[i][j] ], xmm1 }}
```

(a)

(c)

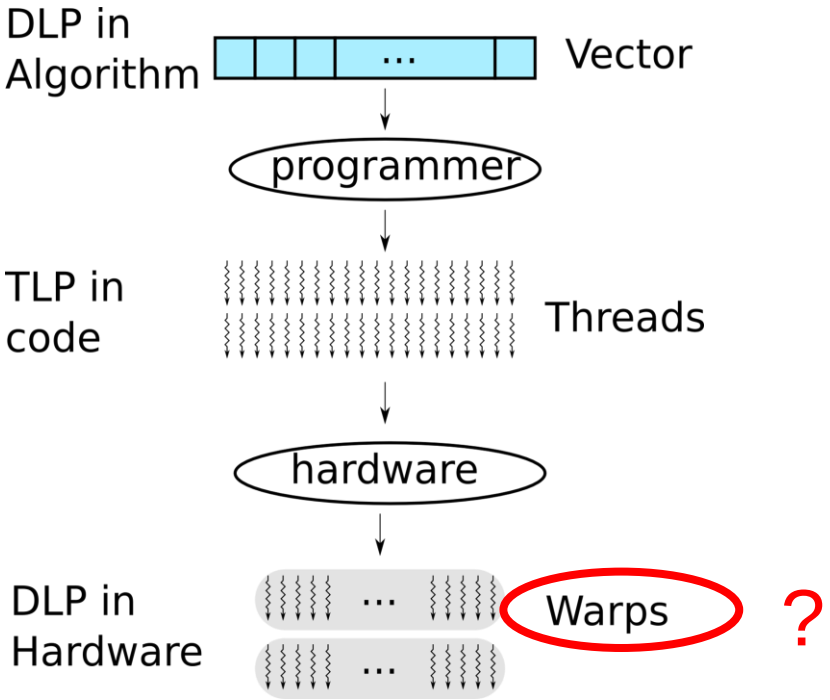
```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
   __shared__ smem[16][16];
   i = threadIdx.y;
   j = threadIdx.x;
   smem[i][j] = A[i][j]; // load to smem
   __sync(); // threads wait at barrier
   A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][j+1] ) / 9;
}
```

(b)

(d)

(a) HW vector width **explicit** to programmer
(b) HW vector width **transparent** to programmers
(c) each vector executed by all PEs in **lock step**
(d) threads executed **out of order**, need explicit sync

# Review What We Have Learned

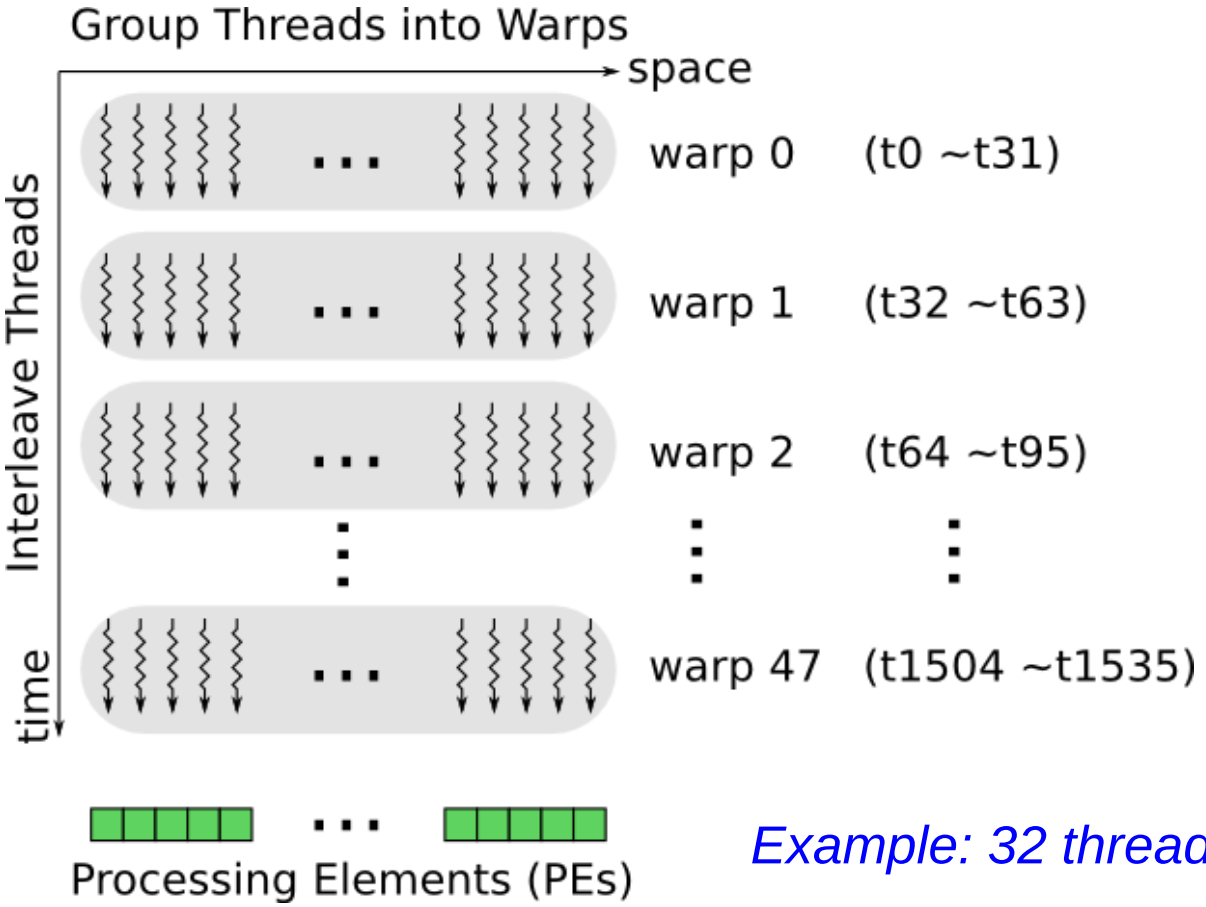




Programmers convert data level parallelism (DLP) into thread level parallelism (TLP)

# Story line:

- Introduction to GPUs
  - what to do with all these transistors
- The programming model of GPUs
  - SIMD (vector) vs Thread-based processing
    - Single Instruction Multiple Thread: SIMT
  - Memory hierarchy
    - Global and Shared (=local)
    - How to use it efficiently
  - **Warps, Scheduling, Register allocation (optional)**
- Example NVIDIA GPUs
- Hazards
  - Bank conflict
  - Divergence
- Performance model: Roofline
- Latest Developments

# HW Groups Threads Into Warps

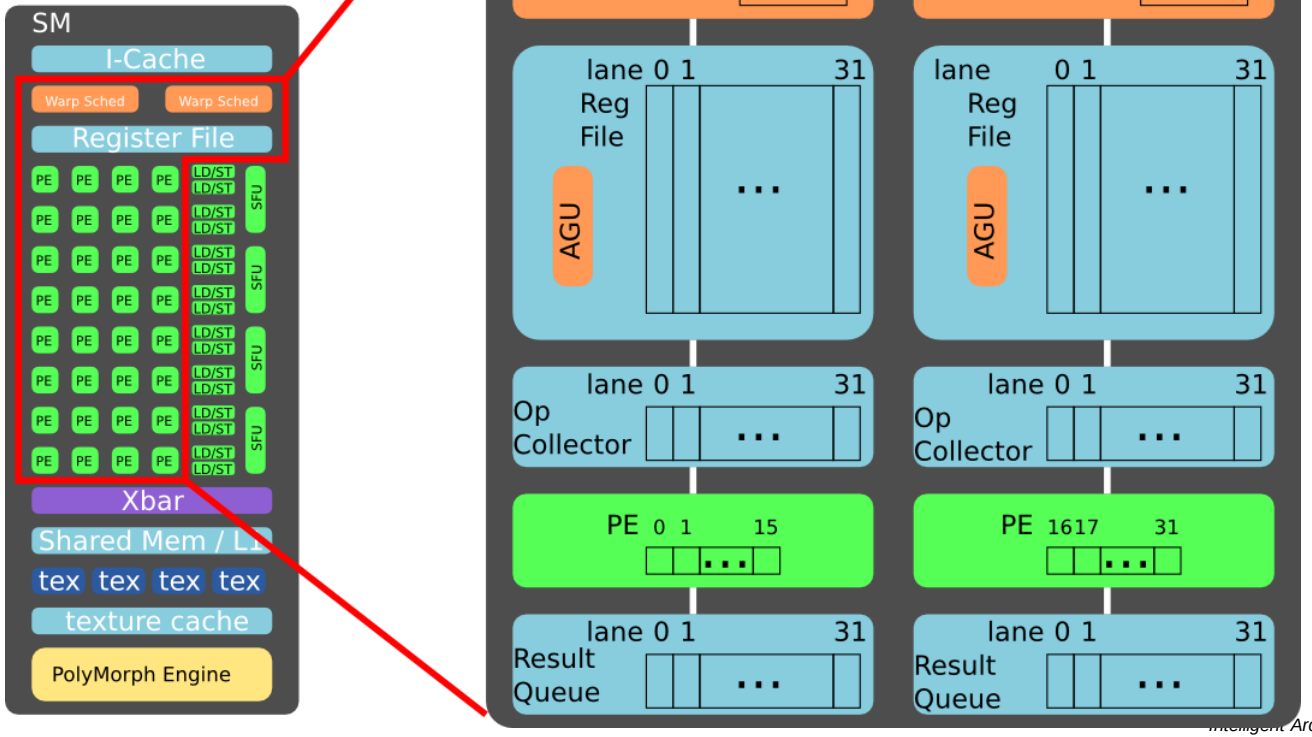



*Example: 32 threads per warp*

# Example of Warp Implementation

Note: NVIDIA may use a more complicated implementation

- *See patent: US 8555035 B1*



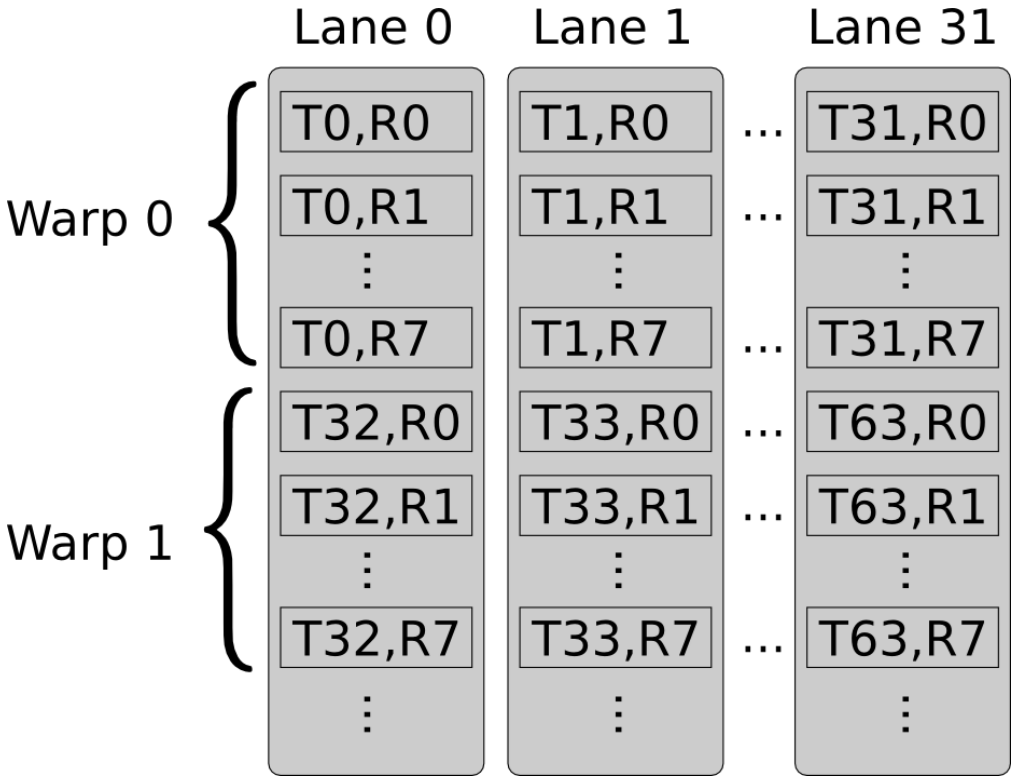
# Example of Register Allocation

Assumptions:

- register file has 32 lanes
- each warp has 32 threads
- each thread uses 8 registers

Acronym:

- “T”: thread number
- “R”: register number




**Register space**: Each thread has in this example **8 private** registers

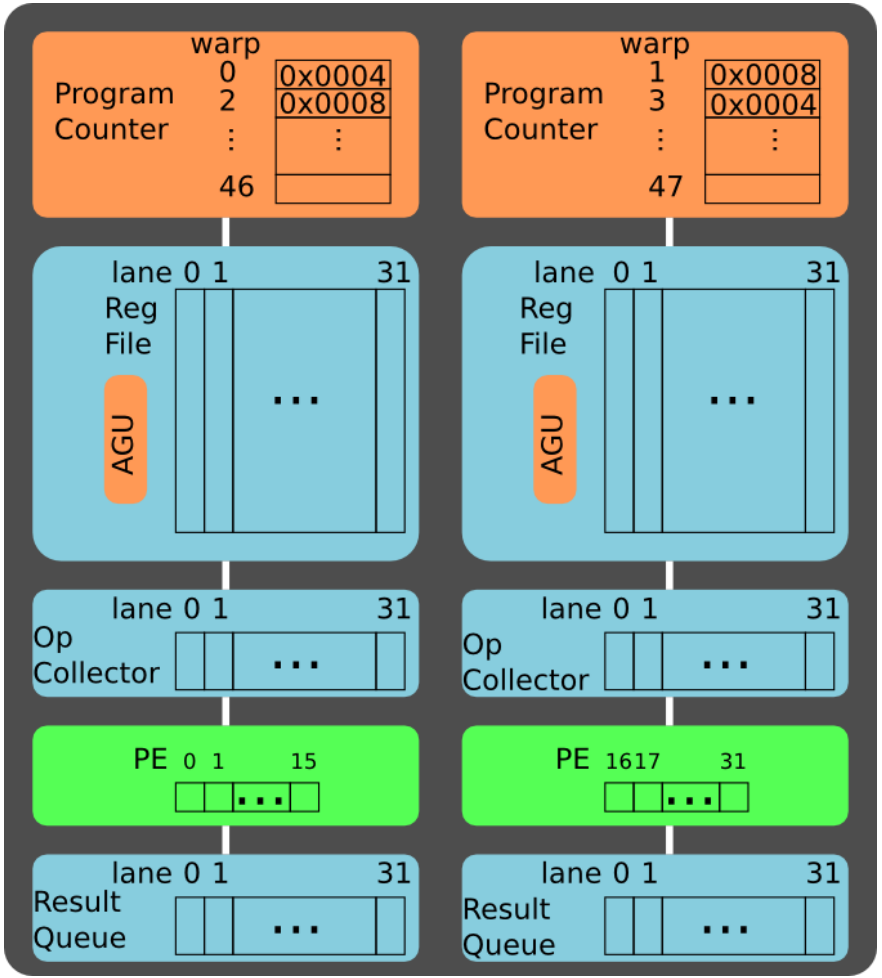
# Execution example

Address : Instruction
0x0004 : add r0, r1, r2
0x0008 : sub r3, r4, r5

Assume two data paths:

- warp 0 on left data path
- warp 1 on right data path
- check their PC (@top)

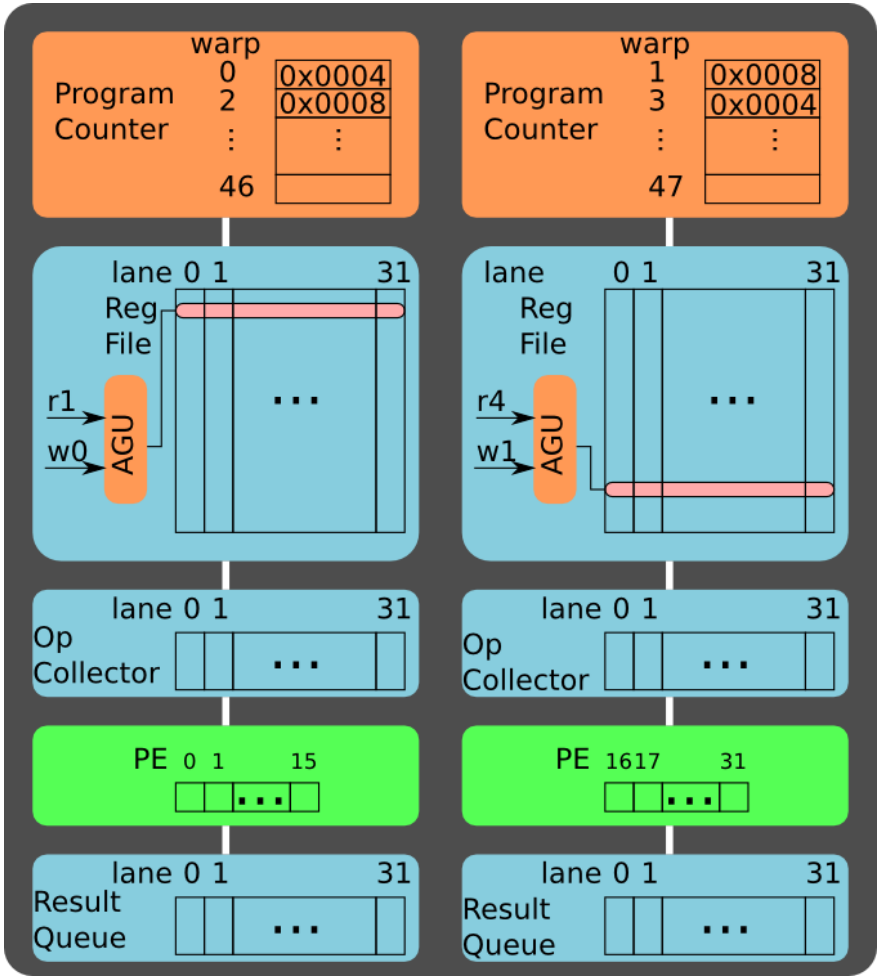
Acronyms:
"**AGU**": Address Generation Unit
"**r**": register in a thread
"**w**": warp number



# Read Src Op 1

Address : Instruction
0x0004 : add r0, r1, r2
0x0008 : sub r3, r4, r5

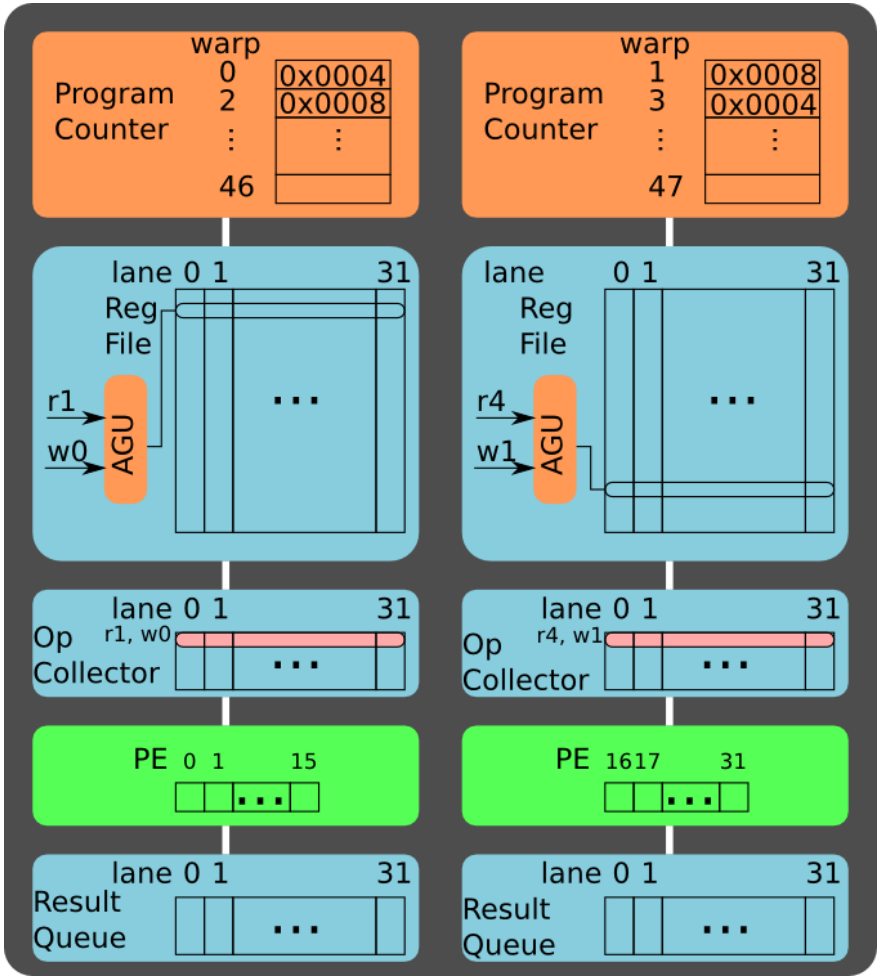
Read source operands:
r1 for warp 0
r4 for warp 1



# Buffer Src Op 1

Address : Instruction
0x0004 : add r0, r1, r2
0x0008 : sub r3, r4, r5

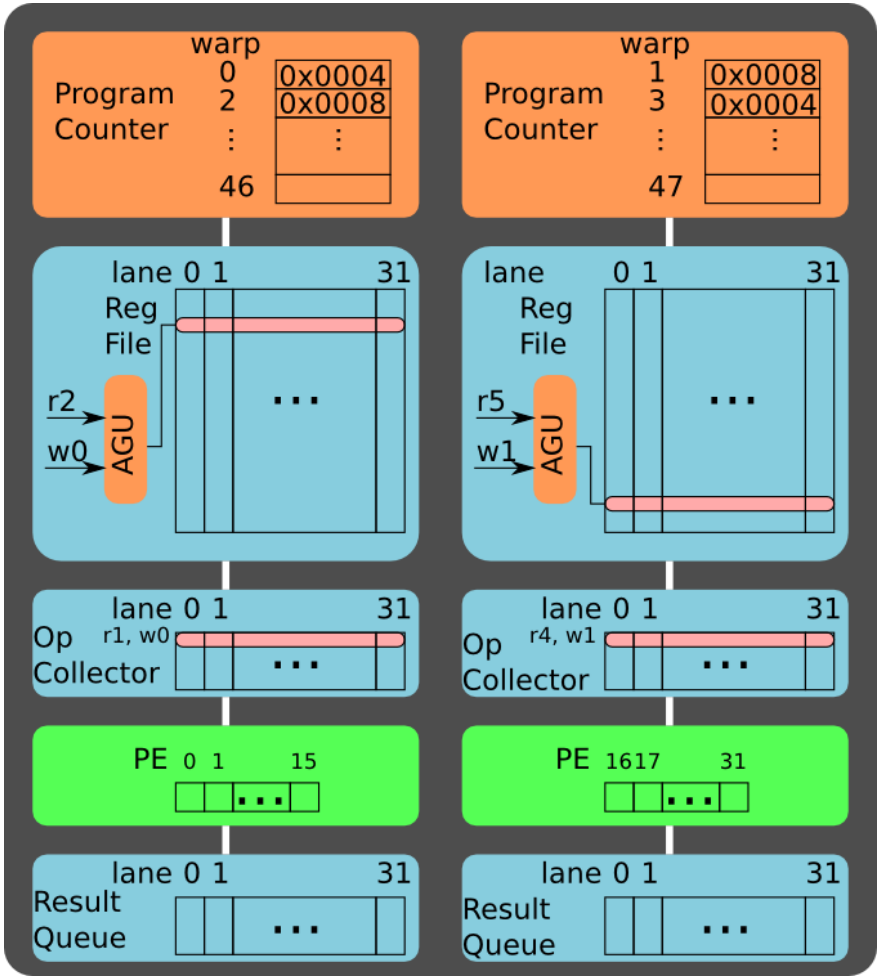
Push ops to op collector:
r1 for warp 0
r4 for warp 1



# Read Src Op 2

Address : Instruction
0x0004 : add r0, r1, r2
0x0008 : sub r3, r4, r5

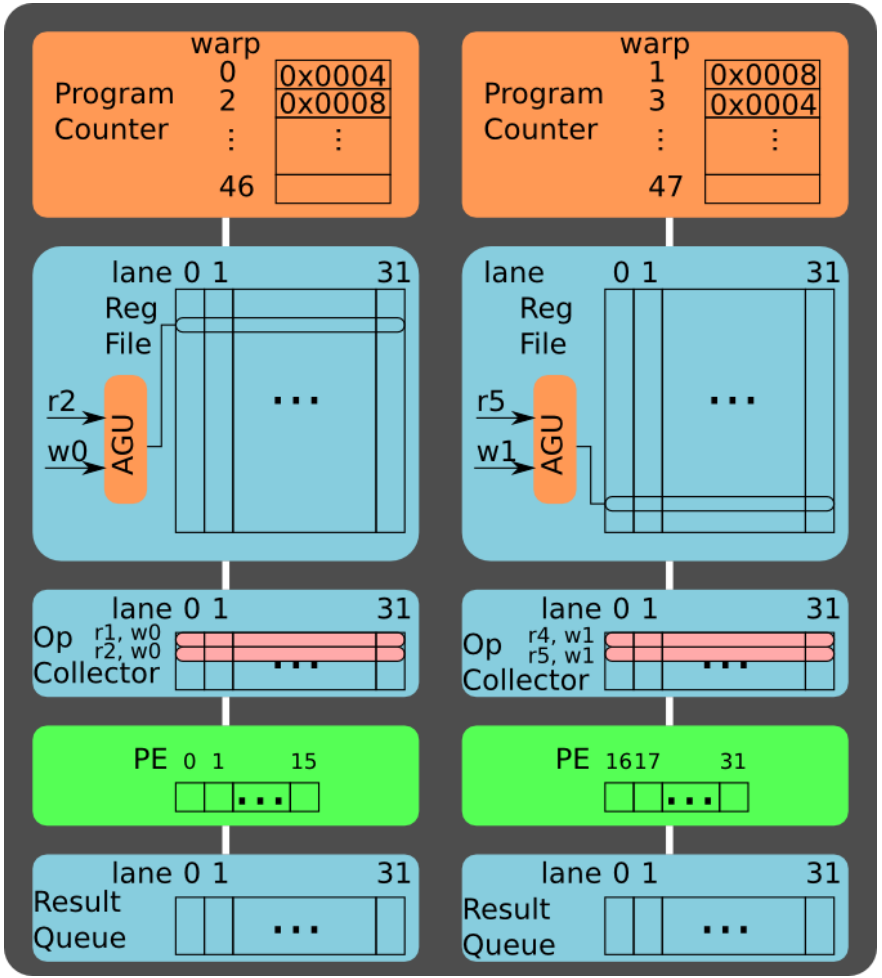
Read source operands:
r2 for warp 0
r5 for warp 1



# Buffer Src Op 2

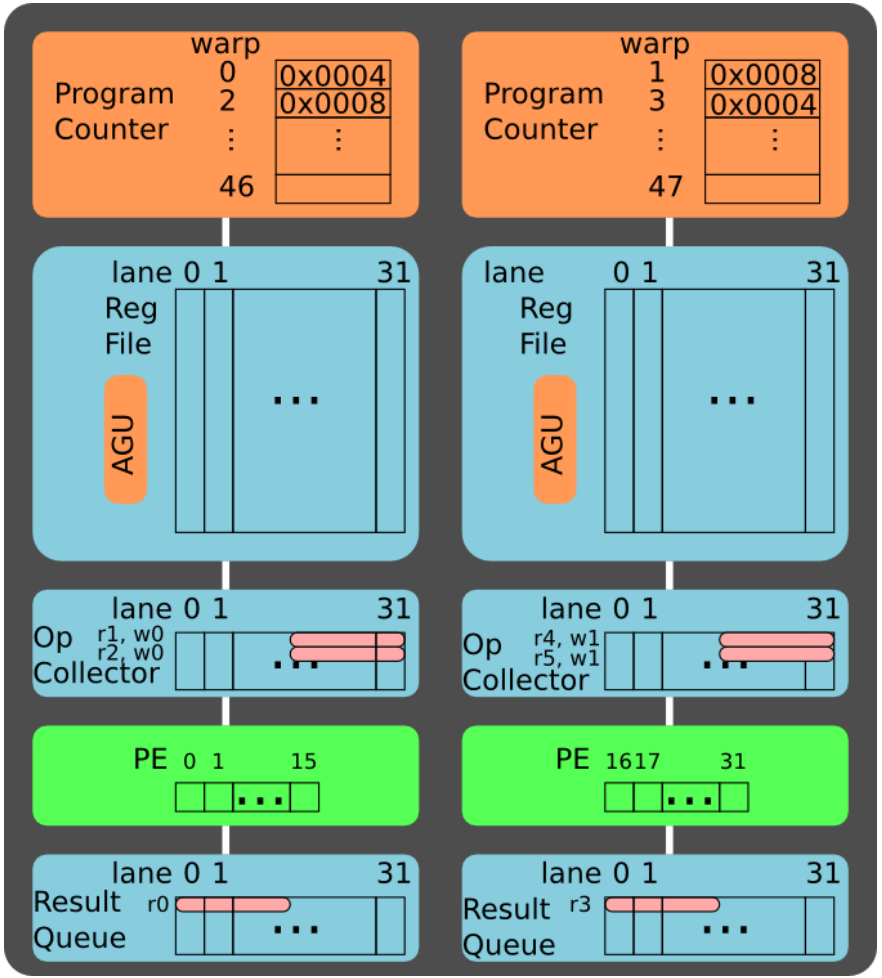
Address : Instruction
0x0004 : add r0, r1, r2
0x0008 : sub r3, r4, r5

Push ops to op collector:
r2 for warp 0
r5 for warp 1



# Execute Stage 1

Address : Instruction
0x0004 : add r0, r1, r2
0x0008 : sub r3, r4, r5

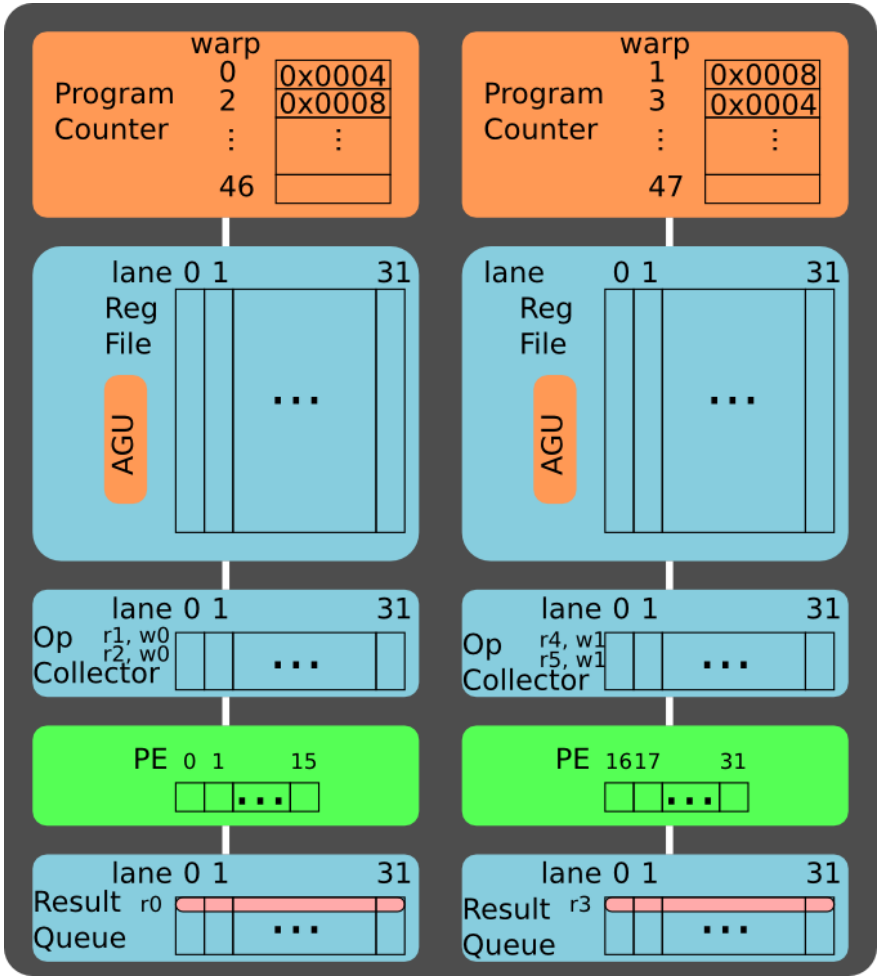



Compute the first 16 threads in the warp

- add for warp 0
- sub for warp 1

# Execute Stage 2

Address : Instruction
0x0004 : add r0, r1, r2
0x0008 : sub r3, r4, r5



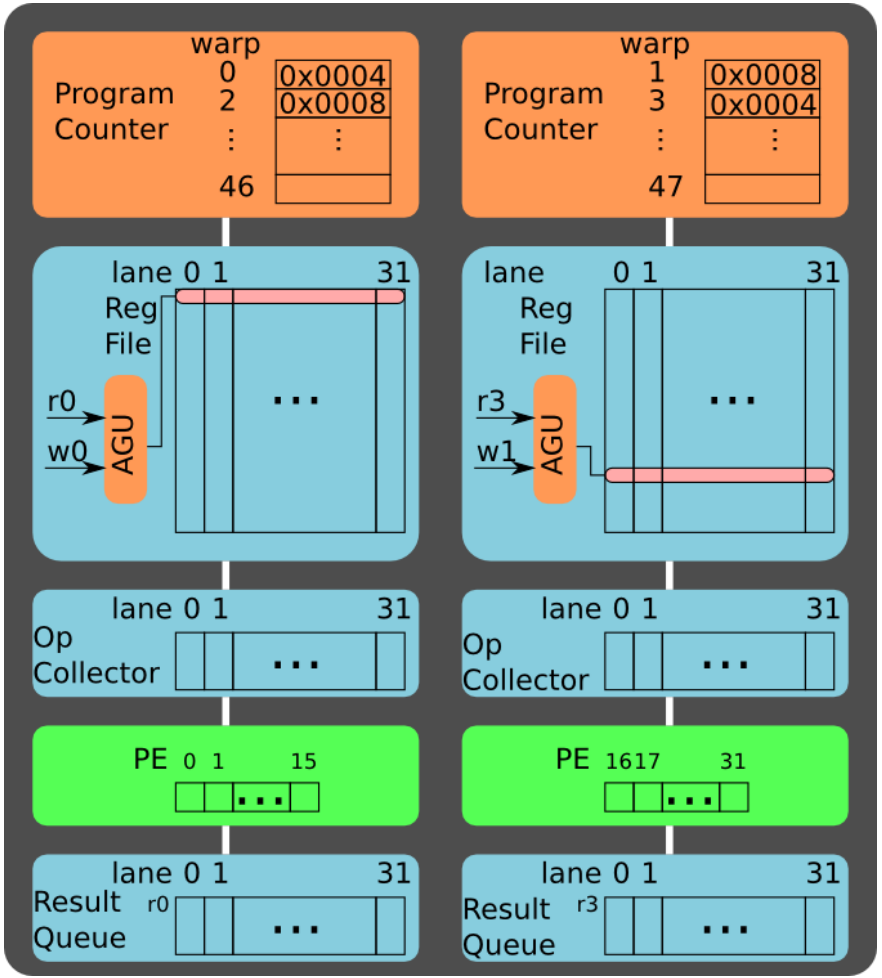

Compute the last 16 threads in the warp

- add for warp 0
- sub for warp 1

# Write Back

Address : Instruction
0x0004 : add r0, r1, r2
0x0008 : sub r3, r4, r5

Write back:
r0 for warp 0
r3 for warp 1



# Story line:

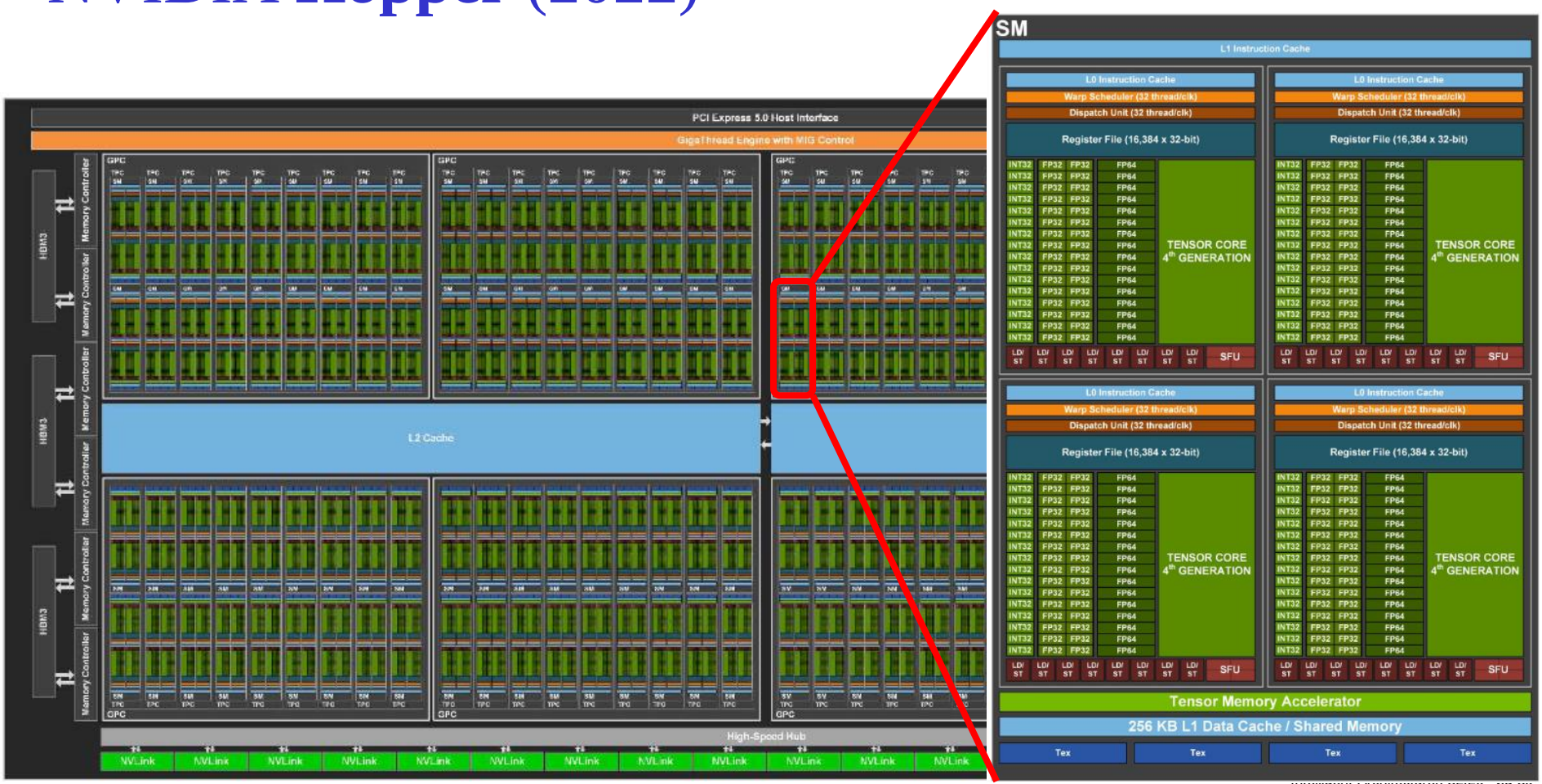
- Introduction to GPUs
  - what to do with all these transistors
- The programming model of GPUs
  - SIMD (vector) vs Thread-based processing
    - Single Instruction Multiple Thread: SIMT
  - Memory hierarchy
    - Global and Shared (=local)
    - How to use it efficiently
  - Warps, Scheduling, Register allocation
- **Example NVIDIA GPUs**
- Hazards
  - Bank conflict
  - Divergence
- Performance model: Roofline
- Latest Developments

# NVIDIA architectures

- Tesla 2010
- Fermi 2010
- Kepler 2012
- Maxwell 2014
- Pascal 2016
- Volta 2017 introducing Tensor Cores (TC) units
- Turing 2018
- Ampere 2020
- Hopper 2022
- Blackwell 2025

- Note: an architecture defines the ISA; it can have different implementations
  - changing #SMs, Memory sizes (of the different mem. levels), technology, etc.

# NVIDIA **Hopper** (2022)

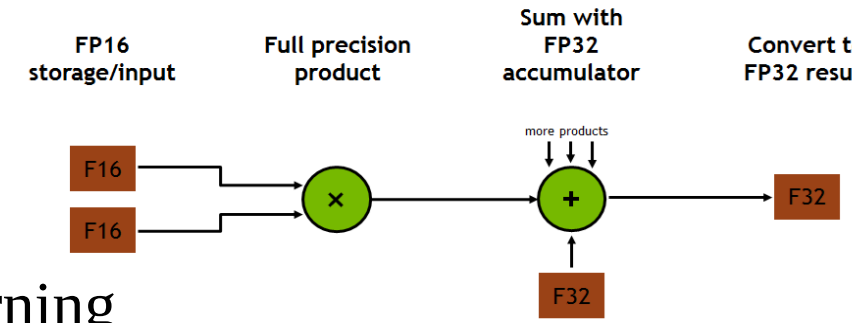


# New in Hopper H100/GH100

- Improved Tensor Core (TC) units
- Thread block cluster
  - threads => blocks => **clusters** => grids
  - enables a block running concurrently across multiple SMs
  - synchronize and collaboratively fetch and exchange data
- Distributed shared memory:
  - direct SM-SM communication
- Asynchronous memory transfers (TMA unit)
- Transformer engine (Software using TC units), for networks like GPT
- HBM3 support & 50 MB L2 cache

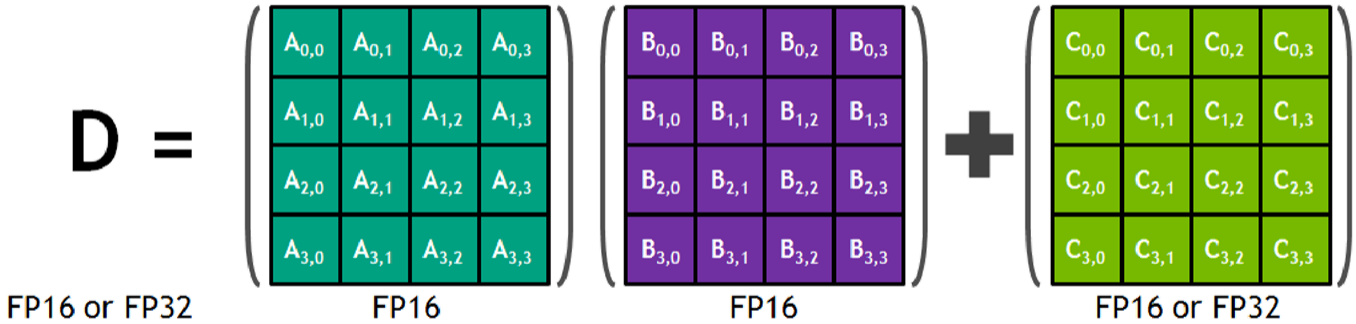
- Performance PEs:
  - FP32 60 TFLOPS
  - FP16 120 TFLOPS
- Performance TC units (numbers double in case of 2x sparsity):
  - TF32 500 TFLOPS
  - FP16 1000 TFLOPS
  - FP8 2000 TFLOPS
  - INT8 2000 TFLOPS

# Tensor Cores




- Specialized core for e.g. deep learning
  - Operates on narrow data types, e.g. fp16

- Example: 4x4x4 matrix-multiply accumulate

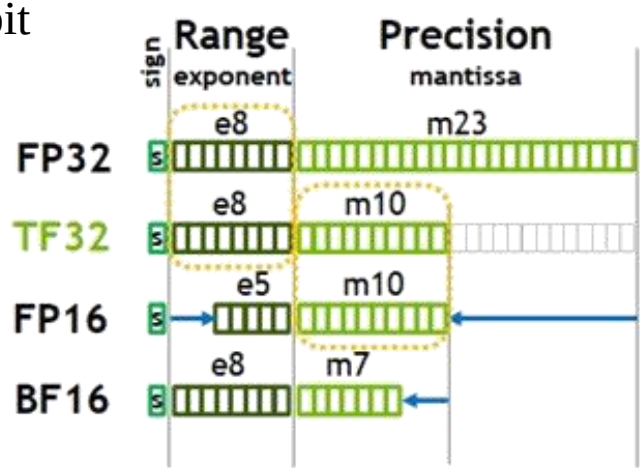


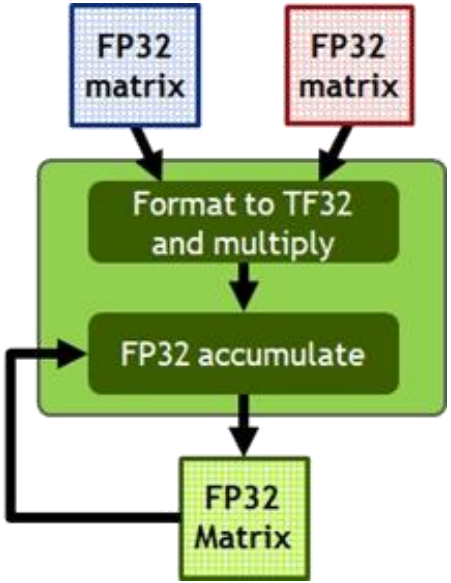

ref: https://www.anandtech.com/show/12673/titan-v-deep-learning-deep-dive/3

# Tensor Cores – supported formats

- Floating point (see picture):
  - FP32, TF32, FP16, BF16
  - Note the easy conversion from FP32 to TF32 and BF16 (just dropping mantissa bits)

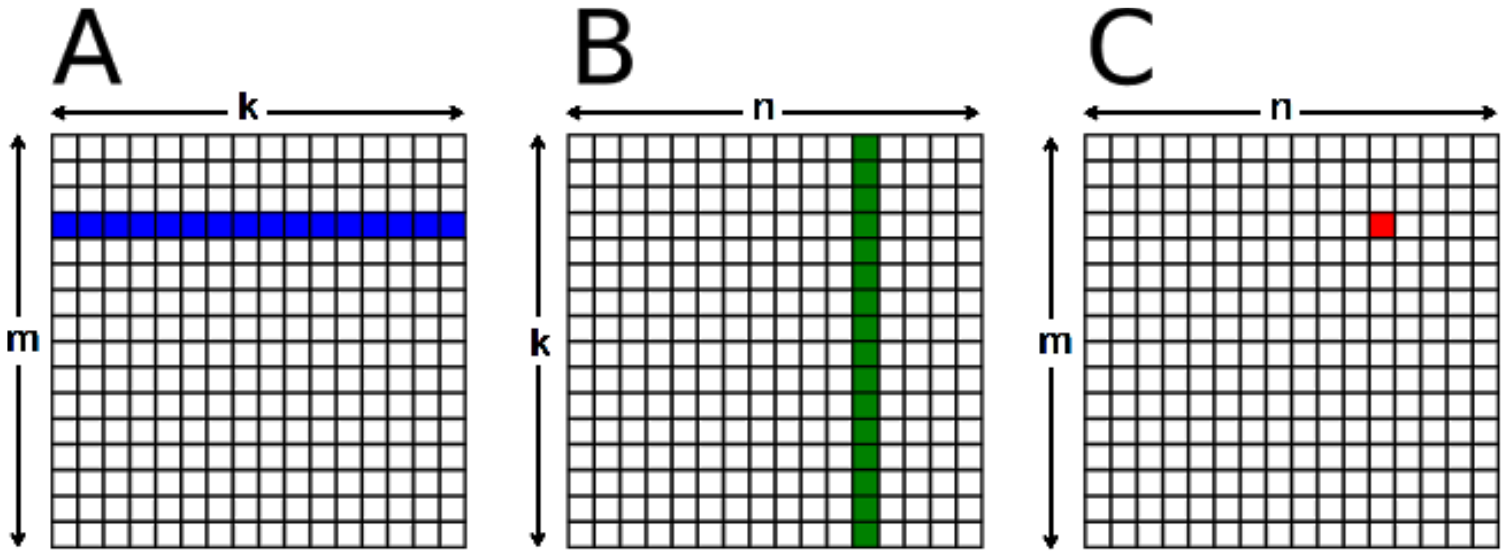
- Integer
  - 32, 16, 8, 4 and 1-bit



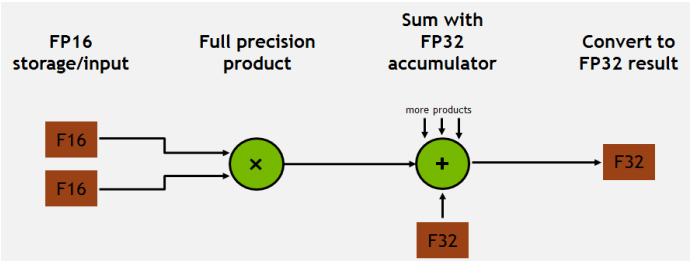




ref: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

# Matrix-Matrix Multiplication on TC unit



# Programming Tensor Cores – example of 1 fragment




```
#include <mma.h>
using namespace nvcuda;
__global__ void wmma_ker(half *a, half *b, float *c)
    // Declare 3 fragments
    // m = 16, n =16 , k=16
    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::col_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;
    // Initialize output c_frag to zero
    wmma::fill_fragment(c_frag, 0);

    // Load the inputs a&b into a_frag & b_frag
    wmma::load_matrix_sync(a_frag, a, 16);
    wmma::load_matrix_sync(b_frag, b, 16);
    // Perform the matrix multiplication
    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);       // C = A*B + C
    // Store the output
    wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}
```

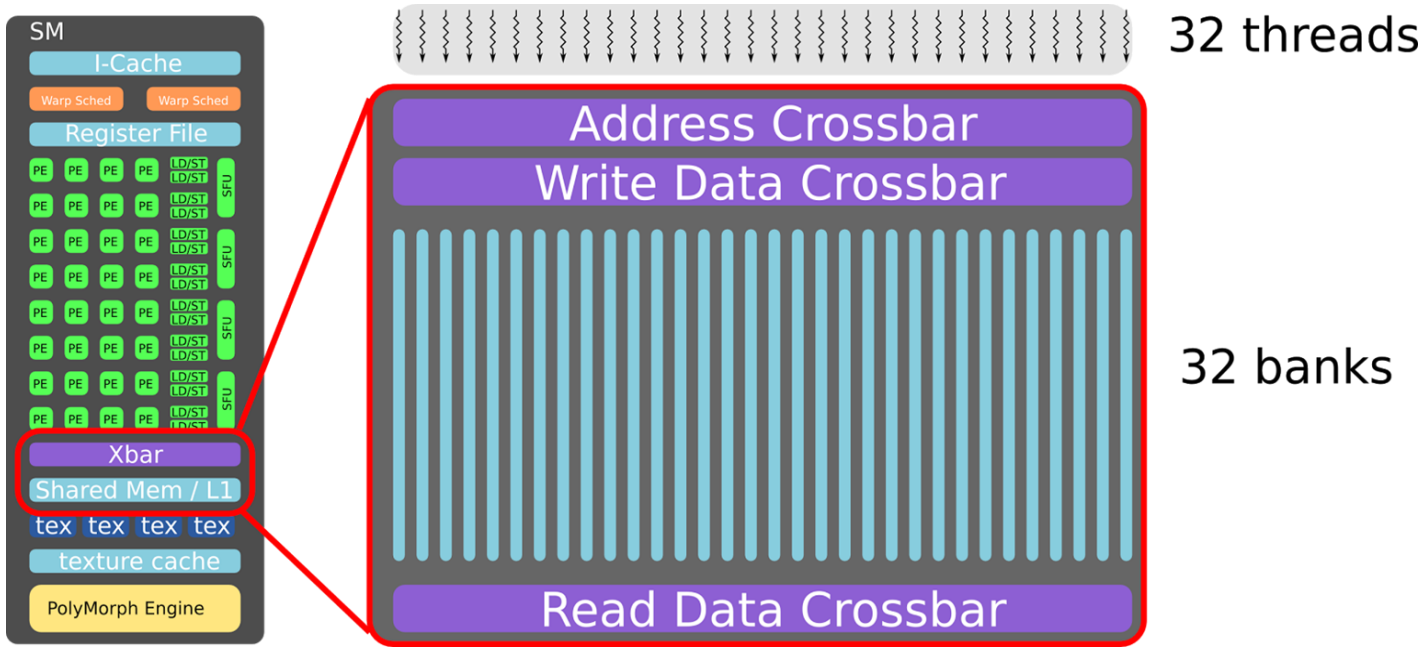
**WMMA (Warp Matrix Multiply Accumulate) API**

# Story line:

- Introduction to GPUs
  - what to do with all these transistors
- The programming model of GPUs
  - SIMD (vector) vs Thread-based processing
    - Single Instruction Multiple Thread: SIMT
  - Memory hierarchy
    - Global and Shared (=local)
    - How to use it efficiently
  - Warps, Scheduling, Register allocation
- Example NVIDIA GPUs
- **Hazards**
  - **Bank conflict**
  - **Branche Divergence**
- Performance model: Roofline
- Latest Developments

# Additional Hazards: **Memory Bank Conflicts**

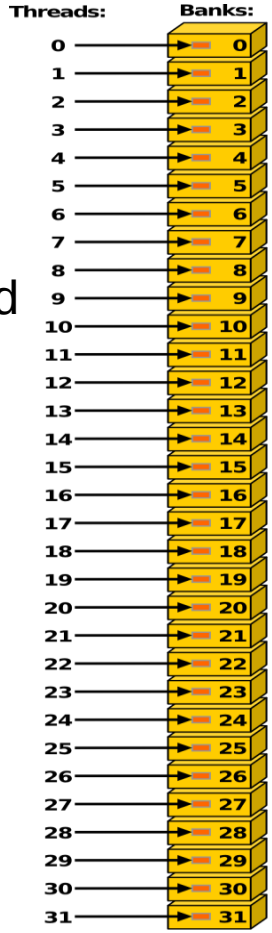
Similar to structural hazard
Example: 32 threads try to perform loads or stores simultaneously to the 32-bank memory

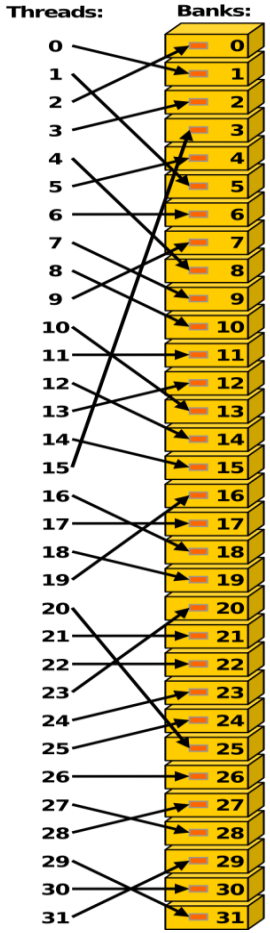


# 1. Shared Memory (compute capability 2.x)

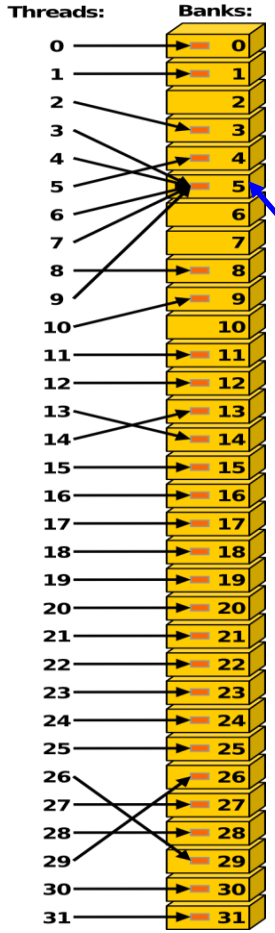
Without bank conflict

- crossbar between PEs inside SM and 32 banks

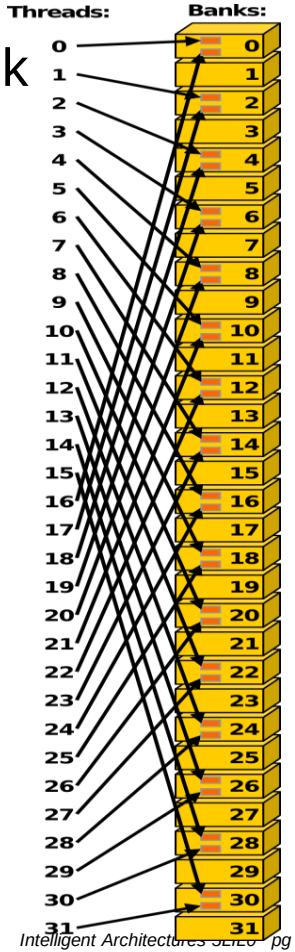



With bank conflict:



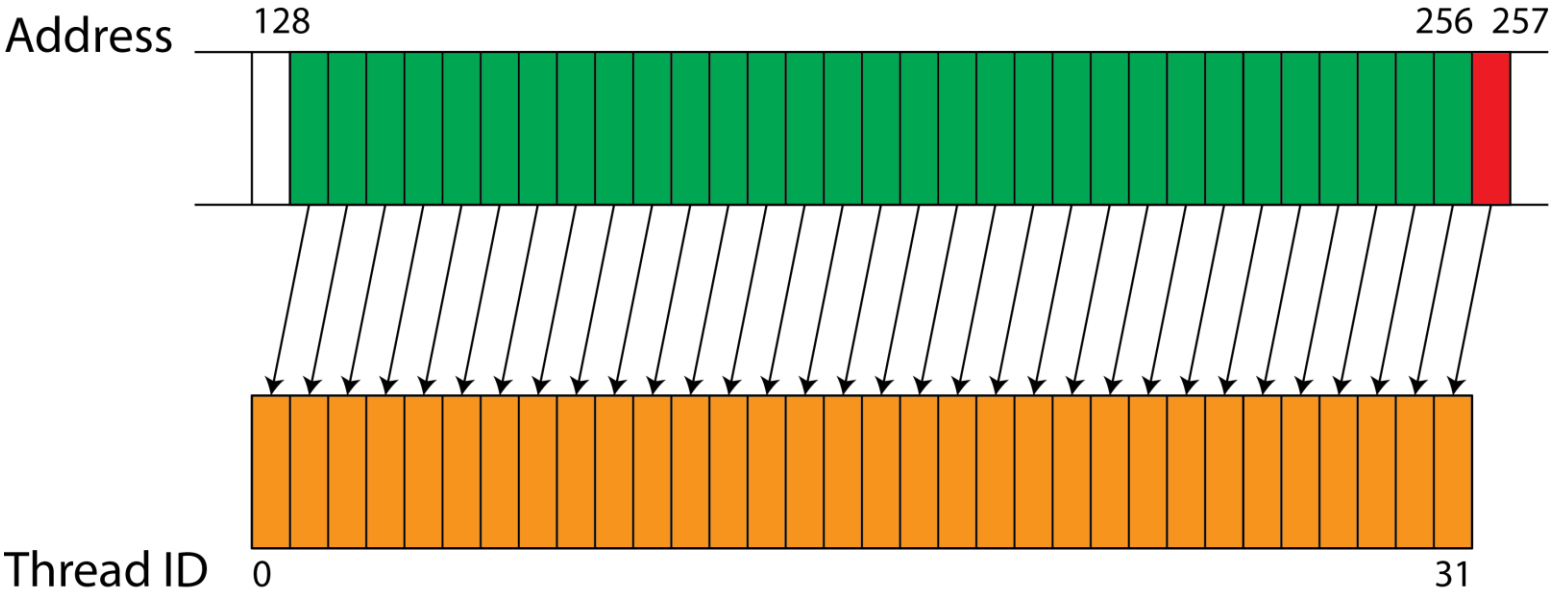

*read same address*




Programmers take the responsibility to eliminate bank conflicts

# 2. Global Memory: **Access Coalescing**

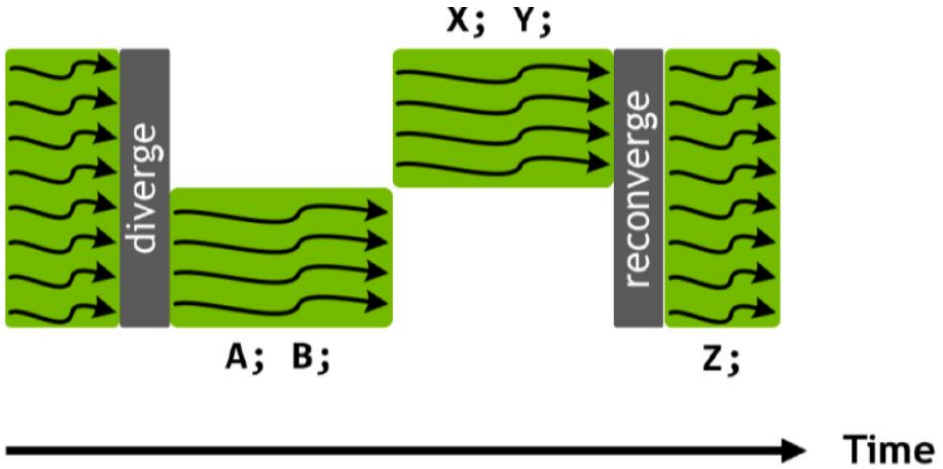
- Example: 32 thread access the global memory (off-chip GPU board DRAM)
- The memory transaction will load/store **consecutive** data at 32/64/128-Byte boundary
- Non-aligned access will cause additional memory transactions
- Below: coalesced, but non-alligned



# Additional Hazards: Branch Divergence

- GPUs (and SIMDs in general) have additional challenges: **branch divergence**

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

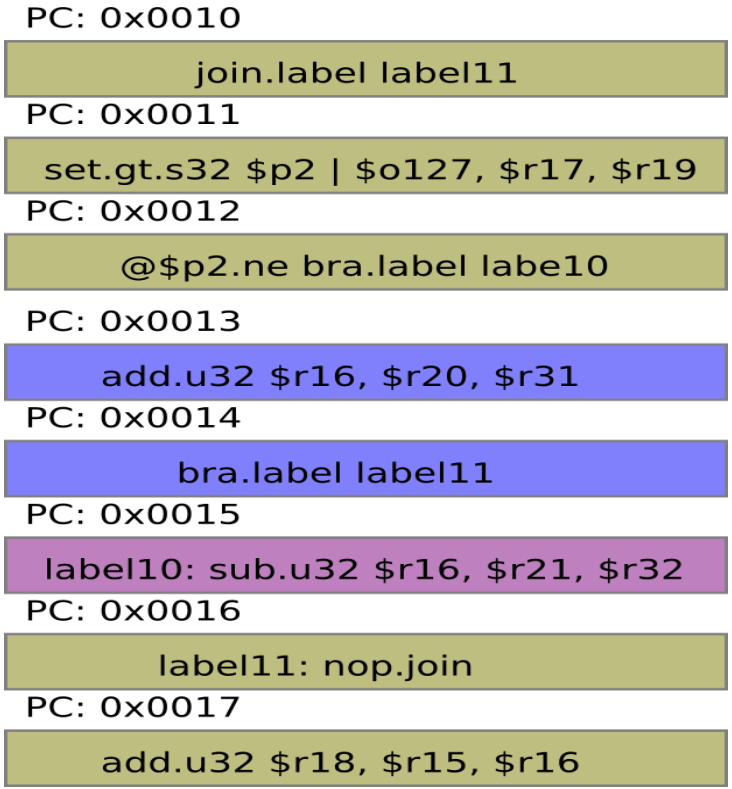


# Handling Branch In GPU

Threads within a warp are free to branch !

Example thread code:

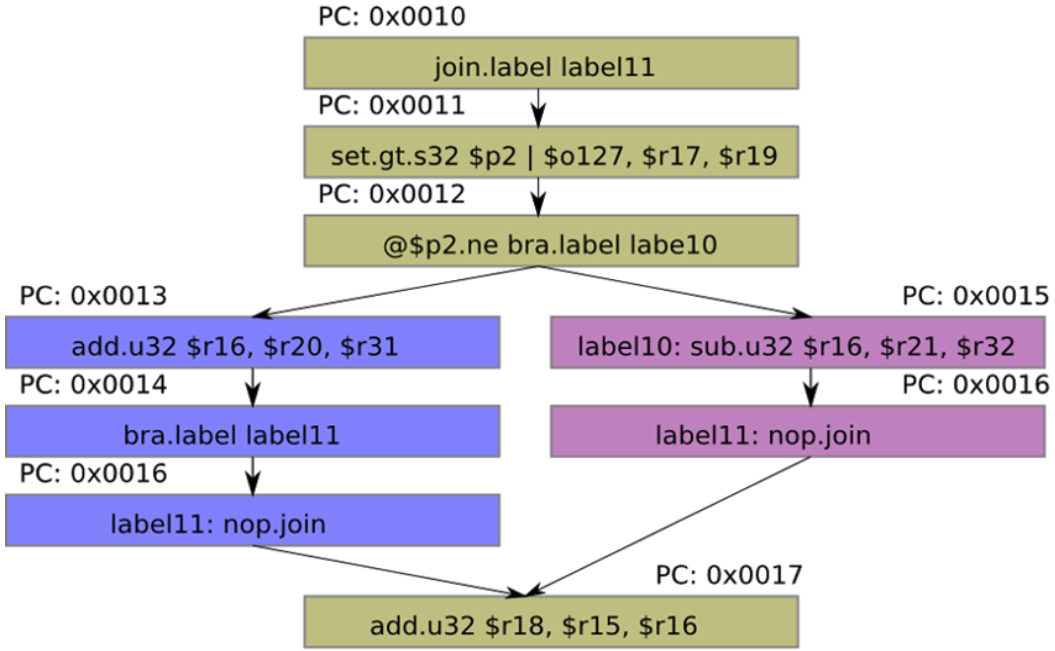
```
if( $r17 > $r19 ){
    $r16 = $r20 + $r31
}
else{
    $r16 = $r21 - $r32
}
$r18 = $r15 + $r16
```

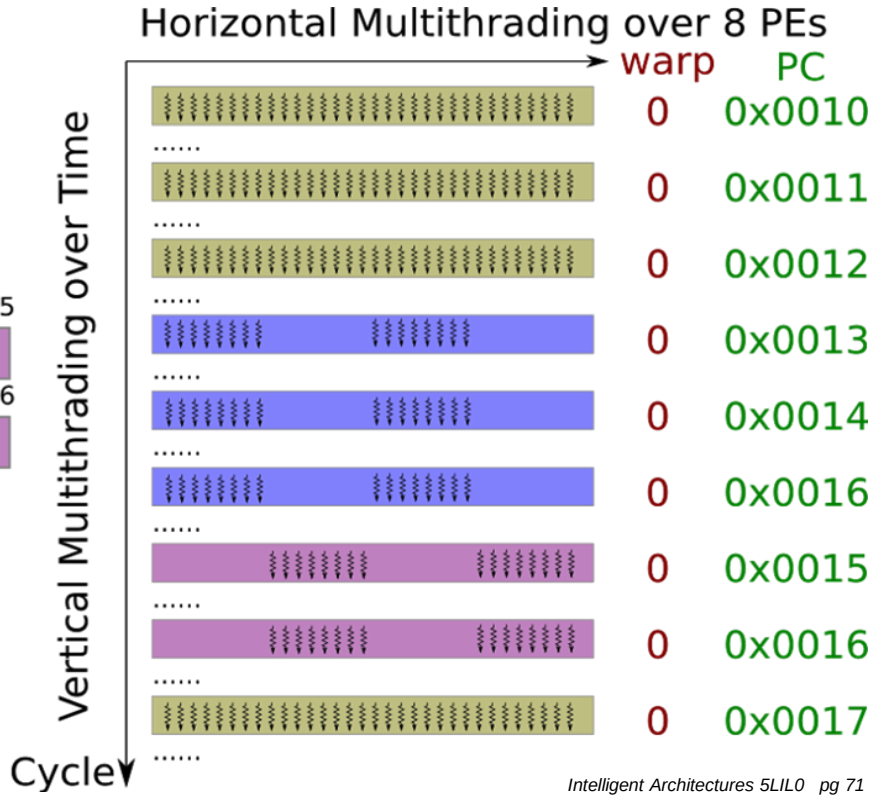



Assembly code on the right are disassembled from cuda binary (cubin) using "decuda", link

# Branch Divergence within a Warp

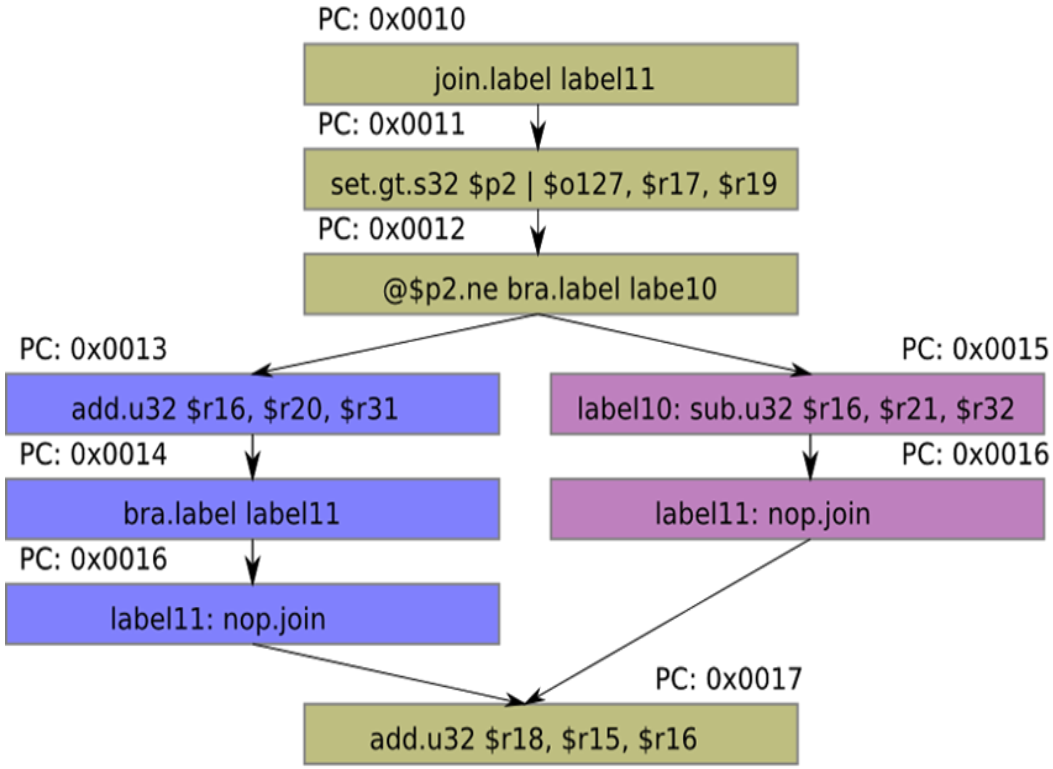
- If threads within a warp diverge, both paths have to be executed
- Masks are set to filter out threads not executing on current path

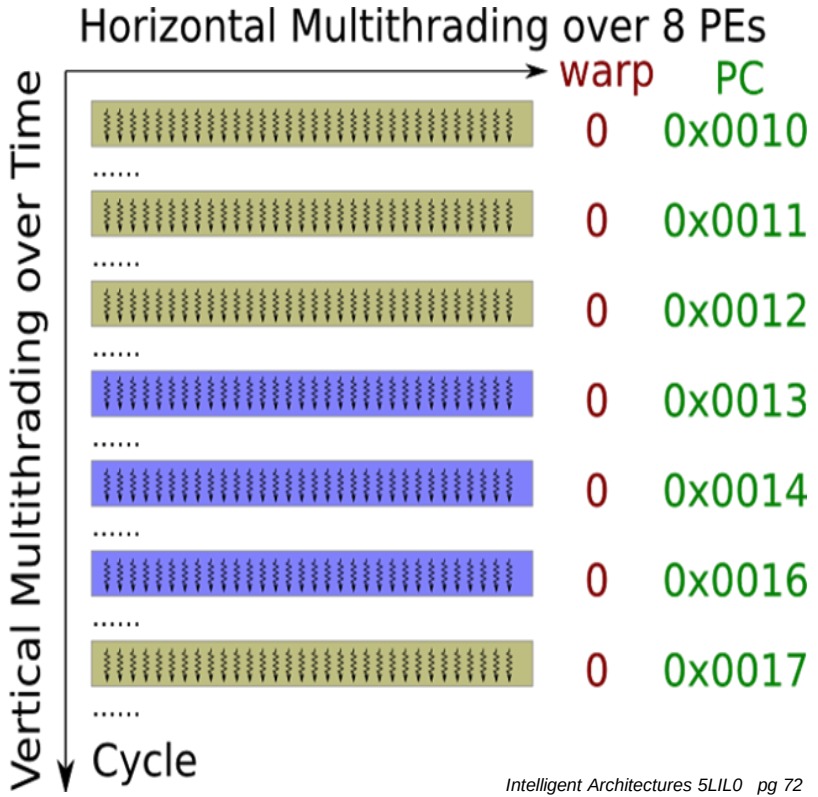


# If No Branch Divergence in a Warp

- If all threads within a warp take the same path, the other path will not be executed



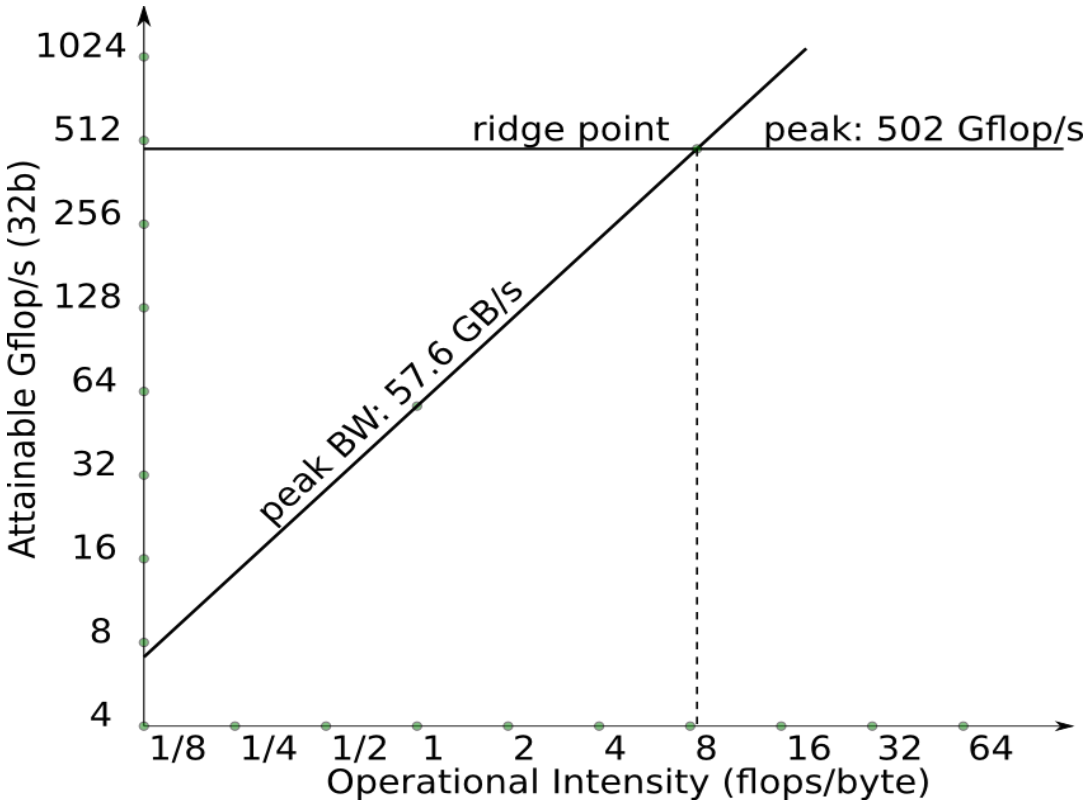
# Story line:

- Introduction to GPUs
  - what to do with all these transistors
- The programming model of GPUs
  - SIMD (vector) vs Thread-based processing
    - Single Instruction Multiple Thread: SIMT
  - Memory hierarchy
    - Global and Shared (=local)
    - How to use it efficiently
  - Warps, Scheduling, Register allocation
- Example NVIDIA GPUs
- Hazards
  - Bank conflict
  - Divergence
- **Performance model: Roofline**
- Conclusions

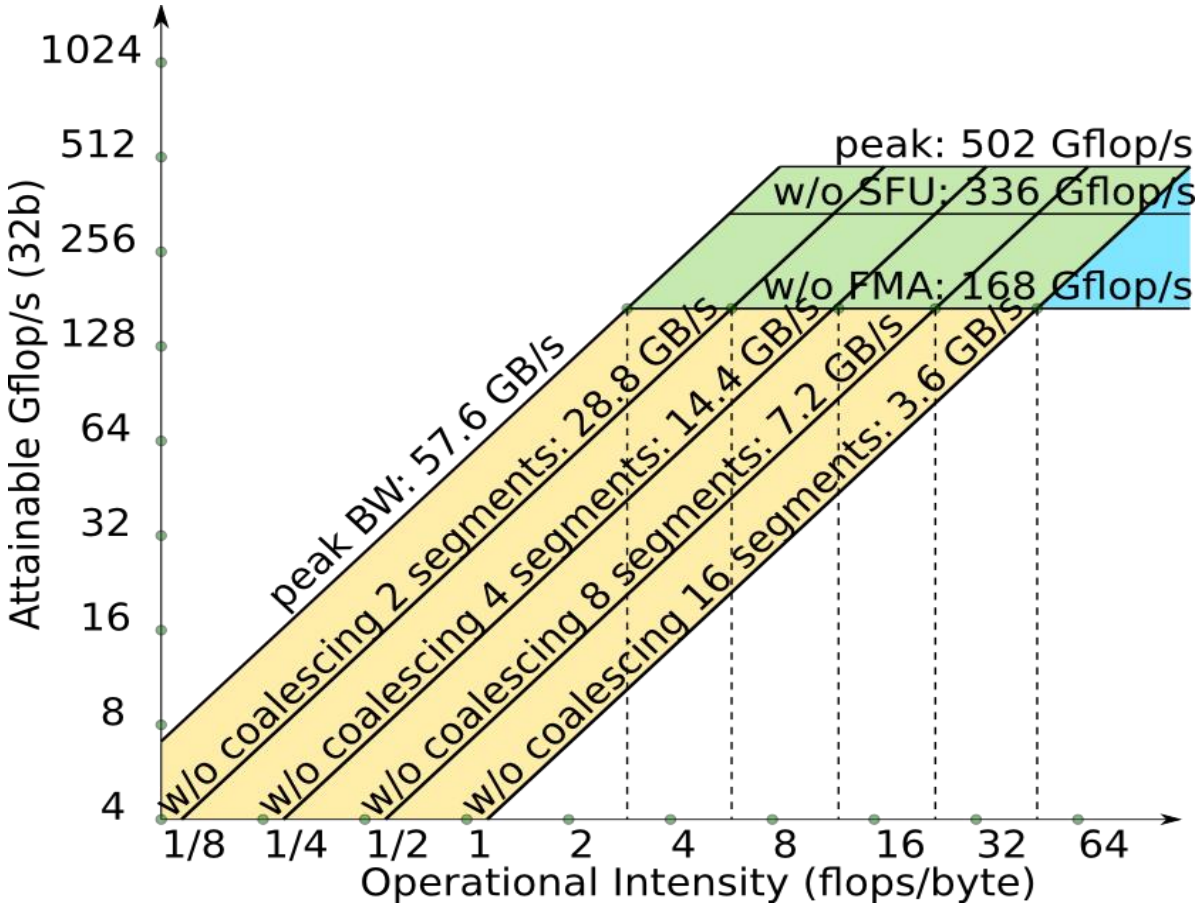
# Performance modeling: the **Roofline Model**

Performance bottleneck: computation bound v.s. bandwidth bound

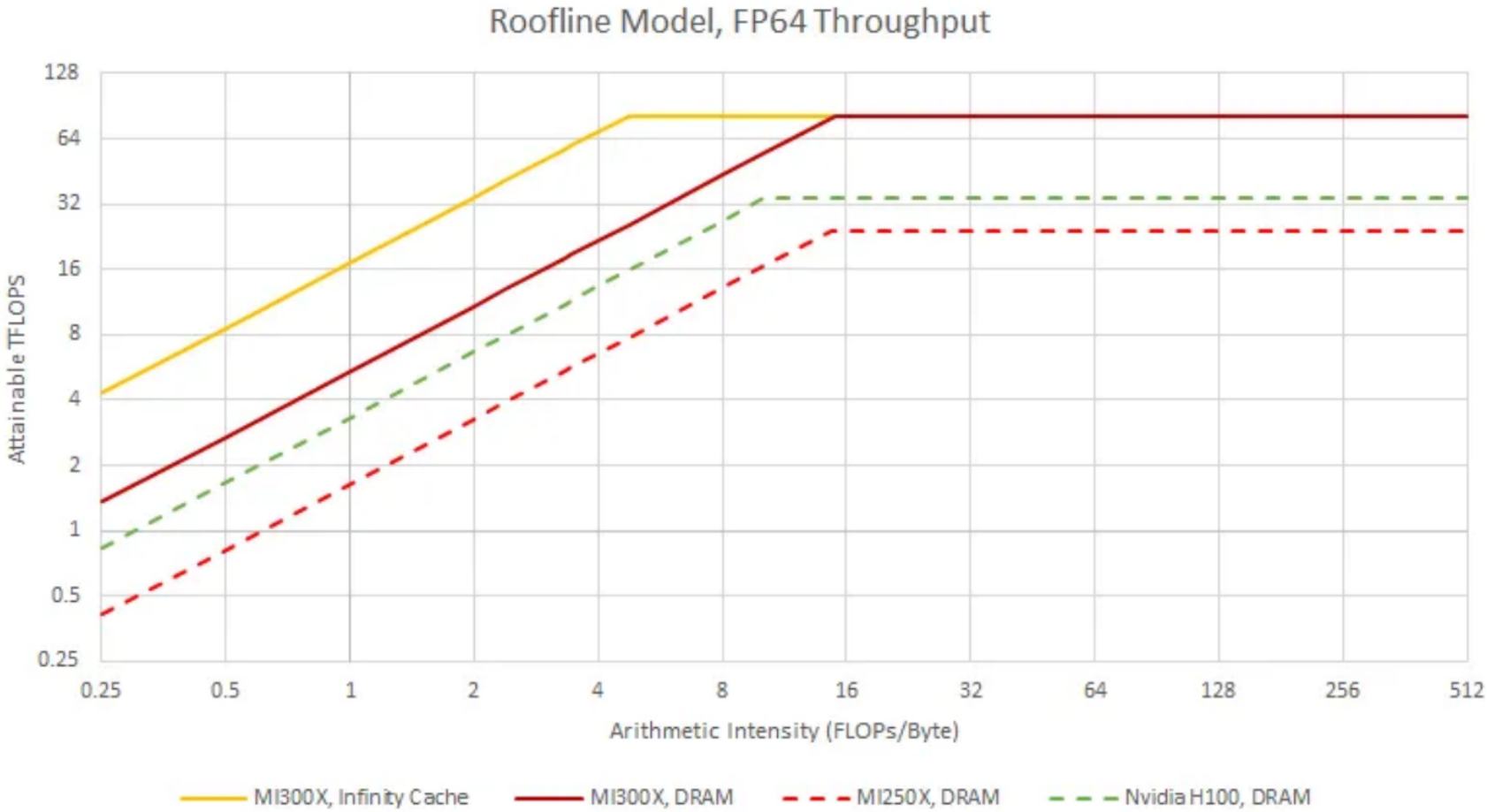
Note: log-log scale



# Optimization is Key for Attainable Gflops/s



# Example roofline models of GPUs

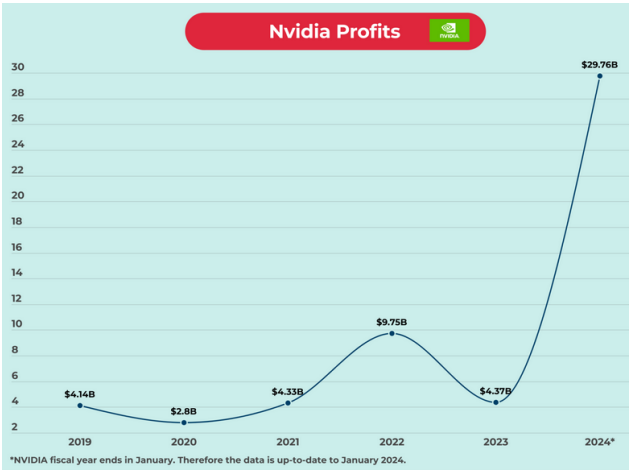


# What did we learn so far …


- Vector / SIMD processing
  - Exploiting DLP (data-level parallelism); It's everywhere
  - Energy efficient = many operations / Joule

- NVIDIA GPU programming in CUDA
  - SIMT programming model (i.s.o. the SIMD model): more flexible
    - groups 32 threads into a Warp
    - GPU architecture agnostic
  - New hazards; try to avoid them

- GPU, the processing workhorse of the future?
    - contain thousands of PEs (Processing Elements)
    - supporting thousands of par. threads



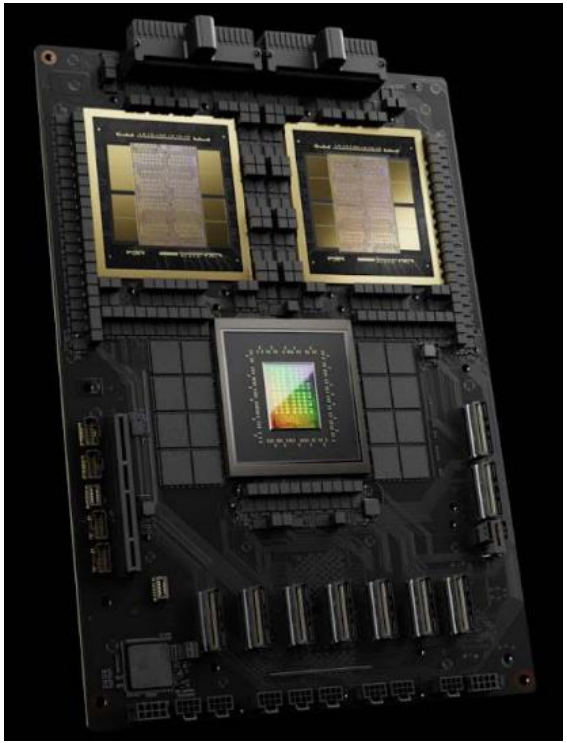
# Story line:

- Introduction to GPUs
  - what to do with all these transistors
- The programming model of GPUs
  - SIMD (vector) vs Thread-based processing
    - Single Instruction Multiple Thread: SIMT
  - Memory hierarchy
    - Global and Shared (=local)
    - How to use it efficiently
  - Warps, Scheduling, Register allocation
- Example NVIDIA GPUs
- Hazards
  - Bank conflict
  - Divergence
- Performance model: Roofline
- **Latest developements**

# Newest developments from NVIDIA and AMD

- NVIDIA
  - Blackwell architecture (2024)
    - B100/B200 GPUs used in e.g. **GB200 multi-die chip**
    - GB202: RTX 5070/80/90 cards (end of 2025)
    - see nvidia-rtx-Blackwell-gpu-architecture.pdf
  - Rubin (2026) + VERA CPU integrated
  - Embedded?

- AMD
  - CDNA3 architecture (2024)
    - **MI300A and MI300X** Chips (2024), 5-6 nmnm
  - CDNA4 architecture (2026)
    - MI350 chips, in 3nm
    - supports 4 and 6 bit FP
    - 288 GB of HBM3E memory
    - 35x perf of MI300



*NVIDIA GB200 Superchip*
*Incl. 2 Blackwell GPUs*
*and One Grace CP*

# NVIDIA Blackwell

**GB202 used in RTX5090 boards**

- 750 mm$^2$, 4 nm TSMC
- 92.2 Btransistors
- 192 SMs (Streaming Multi-Processors)
- 24576 CUDA Cores (128 / SM)
- 768 Tensor Cores (4 /SM)
- 128 MB L2 $
- 2.4 GHz
- GDDR7 DRAM on board


*GB202 GPU, full chip*

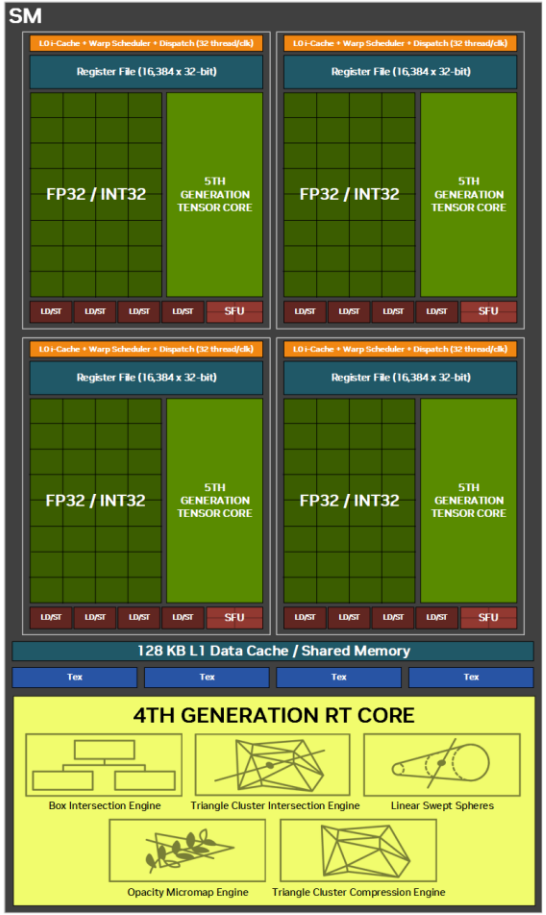
**GB200 Grace with 2 Blackwell GPUs:**

- FP4 20/40 petaFLOPS
- FP6/8/INT8 10/20 petaFLOPS
- FP16/BF16 5/10 petaFLOPS
- HBM3E upto 384 GB

**B100 GPU**

- FP4 7/14 petaFLOPS
- FP6/8/INT8 3.5/7 petaFLOPS
- FP16/BF16 1.75/3.5 petaFLOPS
- 700 Watt => 50 / 100 / 200 fJ/op (sparse)

# Blackwell: 1 SM unit




- 128 FP32/INT32 cores
  - Int and FP share the same HW
- 4 TCs (Tensor Cores)
  - supports: FP/BF16, TF32, INT8, FP6, FP4
- 2 FP64 core
  - less support for FP64

- 256 KB Register File
- 128 KB L1$ + Shared memory
  - flexible division between $ and Scratchpad memory

# AMD CDNA3 architecture and MI300

- Architecture
- Memory hierarchy
- Programming

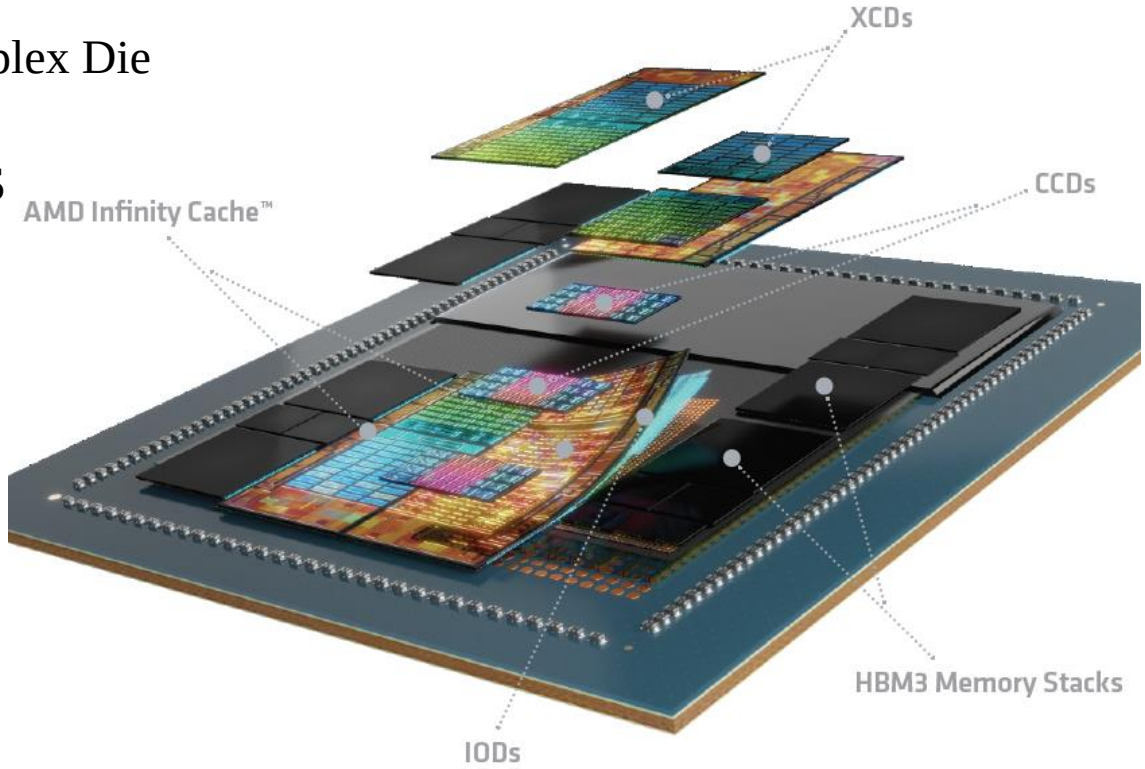
- Refs
  - http://chipsandcheese.com/p/testing-amds-giant-mi300x
  - http://rocm.docs.amd.com/projects/HIP
  - https://www.amd.com/en/technologies/cdna.html
  - amd-cdna-3-white-paper.pdf
  - https://en.wikipedia.org/wiki/CDNA_(microarchitecture)#CDNA_3

# Architecture

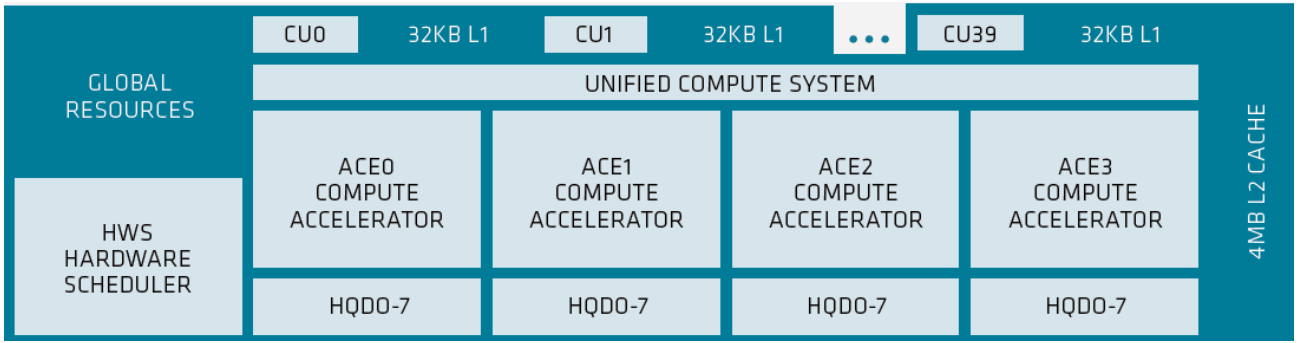
- Chiplet based
  - XCD GPU die (Acc Complex Die
  - CCD CPU compute die
  - IOD die with infinity L3 $
  - HBM3 3-D mem dies

- MI300X
  - 8 XCDs
  - 4 IODs

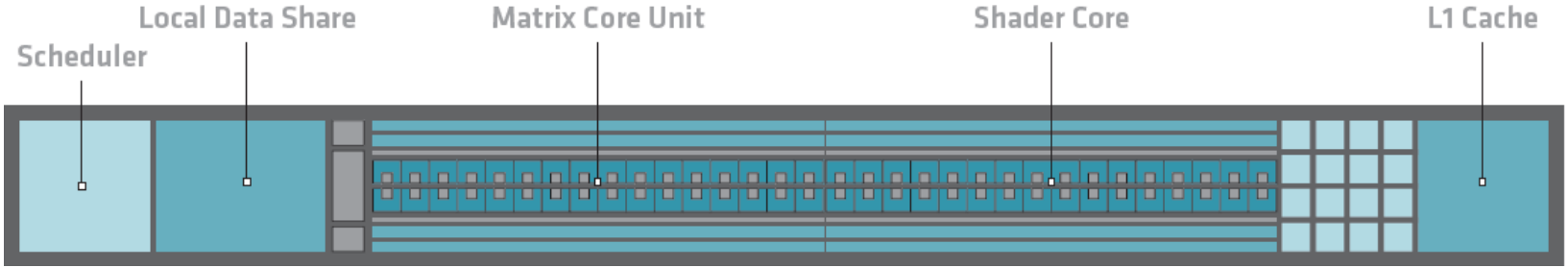
- MI300X
  - 6 XCDs + 3CCDs



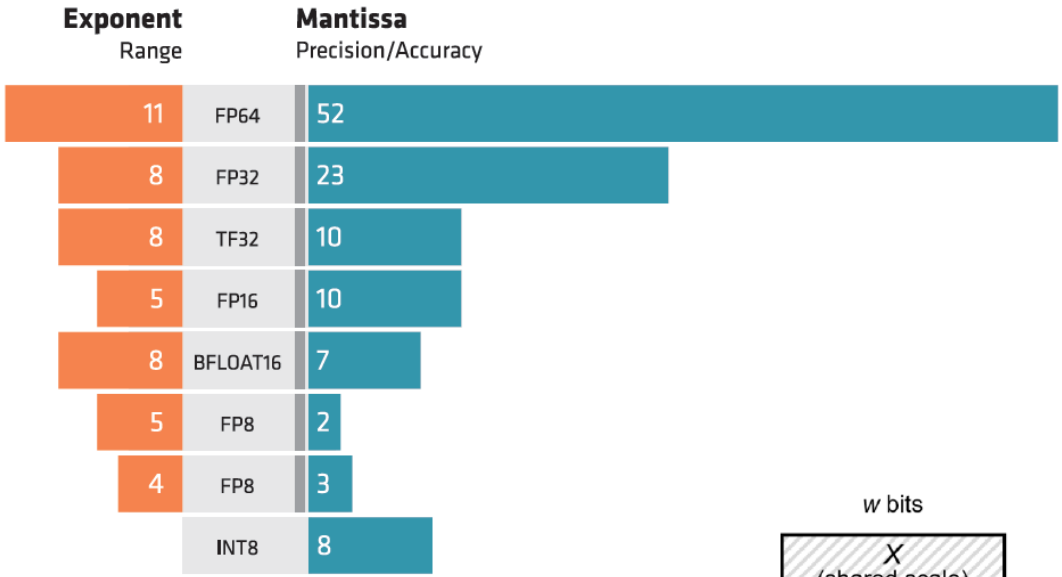
# XCD GPU – CU (compute unit)




- XCD: 38 CUs
  - 4MB L2 $
- CU
  - L1 $ + local mem
  - Matrix units
    - INT 8/FP8 – 64 FL
    - 1k 8-bit ops/cycle



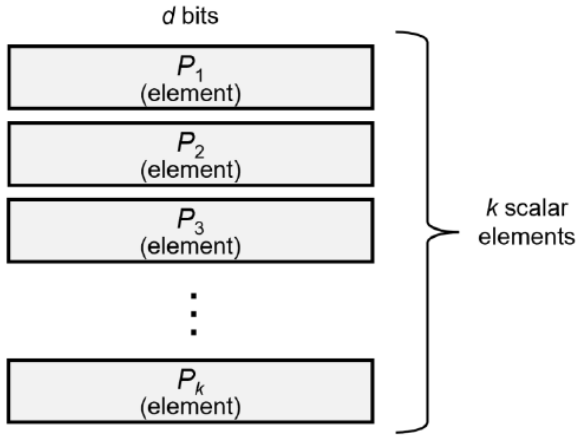
# CDNA 3 Supported data types




New data types are coming:

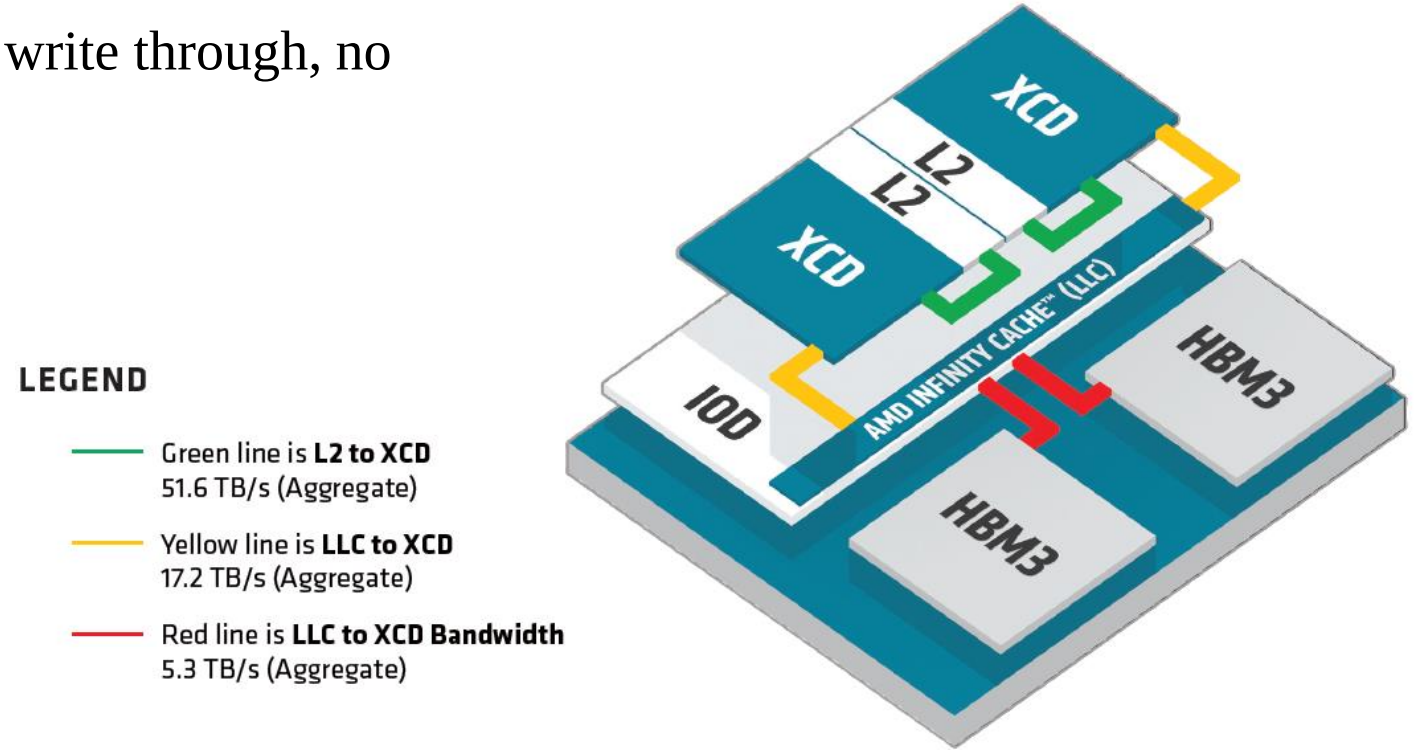
- MX Microscaling Formats
- See: OCP Microscaling Formats (MX) Specification Version 1.0
- use in DeepLearning, e.g. : https://arxiv.org/abs/2310.10537



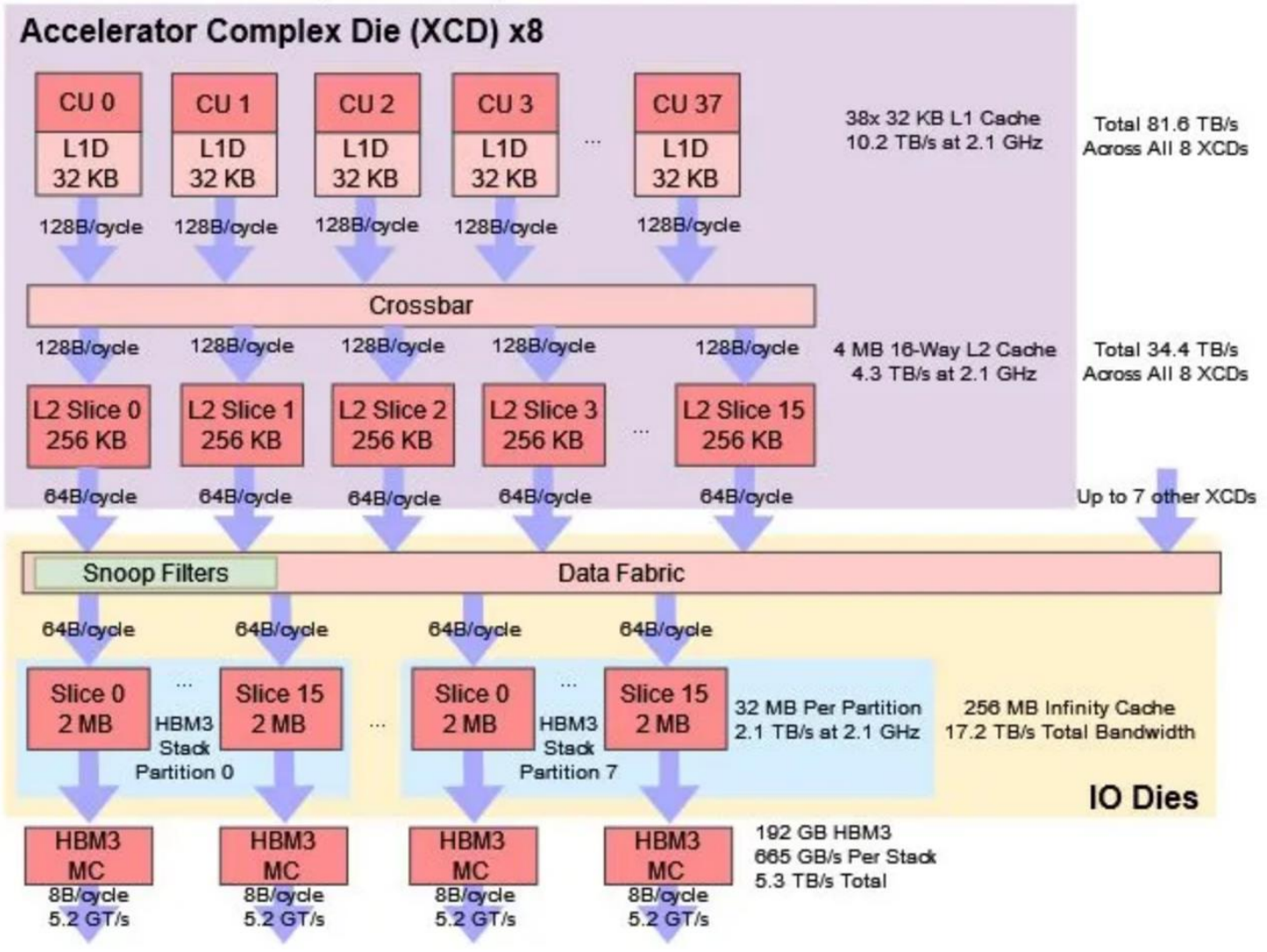
# MI300 Memory organization

- L2 supports snooping supporting coherency
- L3 on IOD die, write through, no snooping



# MI300 Memory organization

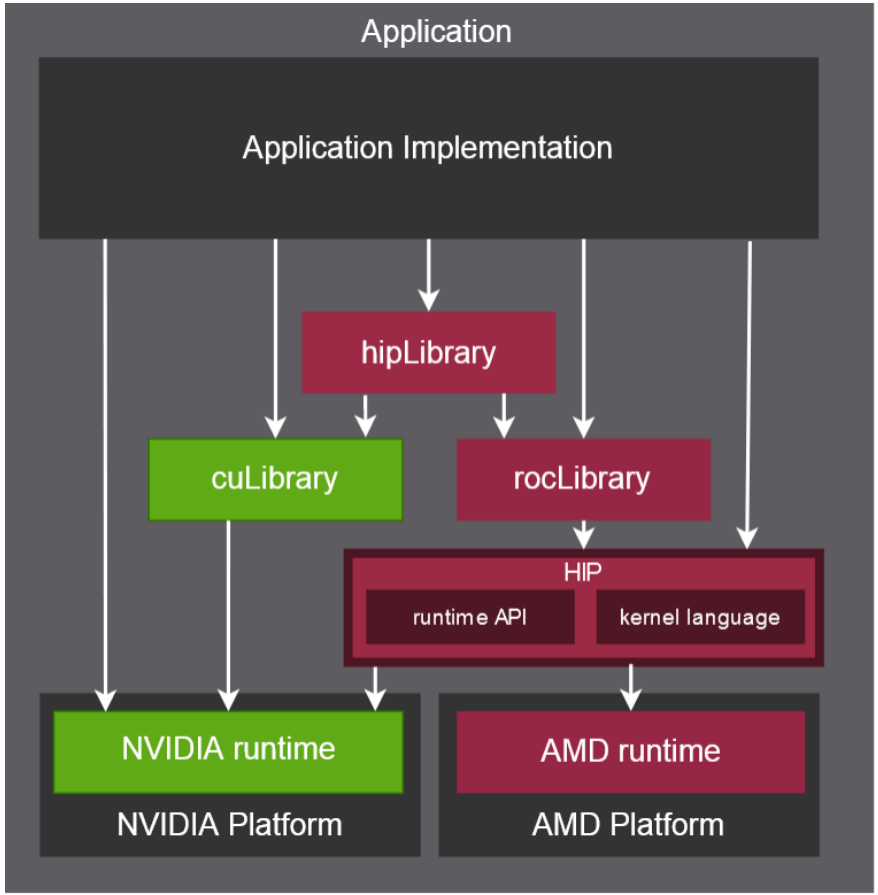
- L2 $ write-back
  - snooping
- L3 $ on IO dies
  - shared between all compute dies



# Programming model

- HIP: tiny API on top of CUDA and ROCm

- http://rocm.docs.amd.com/projects/HIP



# AMD CDNA 3 vs NVIDIA Blackwell

- AMD CDNA3 / MI300
- chiplet based
  - multiple dies for GPUs, CPUs and L3$

- Good support for HPC with very good FP64 support
- Matrix units
  - no < 8bit support

- New SW stack based on ROCm with HIP API layer on top

- NVIDIA Blackwell GB202
- Monolitic (huge) chip

- AI oriented, far less FP64 support

- Tensor Cores
  - incl FP4, FP6 support

- Mature SW optimized for AI learning and inference

# Embedded: NVIDIA Jetson Orin Nano developer kit

- $ 249 (in EU about € 300)
- Ampere architecture
- 6 core ARM A78AE
- 1024 CUDA cores
- 32 Tensor Cores
- 8 GB 128-bit LPDDR4, 102 GB/s
- all kinds of connectivity
- 67 TOPS
- 15-25 Watt
- https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/nano-super-developer-kit/

# NVIDIA mini PC: project DIGITS (announced 2025)

- 10 ARM Cortex X95
- 10 ARM Cortex A725
- Blackwell GB10 chip
  - 1 petaflop FP4
- 128GB of LPDDR5X
- 4 TB SSD

- Power: ?? Watt
- Price: 3000 $

# GPU players (2025)

- NVIDIA
- AMD
- INTEL: e.g. Pontevechio (discont’d) => Jaguar Shores
- Qualcomm: Adreno GPUs usied in Smapdragon processors for mobile devices
- ARM: Mali GPUs used in ARM based SoCs
- Imagination Technologies: PowerVR GPUS: embedded devices
- Jingjia Micro: Chinese, e.g. JM9 series
- Moore Threads: entrant on GPU market

# Further Literature

- Overview and history of GPUs: https://www.wikiwand.com/en/Graphics_processing_unit

- From Chips and Cheese
  - CDNA 3 : https://chipsandcheese.com/p/amds-cdna-3-compute-architecture
  - testing MI300X : https://chipsandcheese.com/p/testing-amds-giant-mi300x

- CUDA Programming:
  - Easy introduction to CUDA:
    - https://developer.nvidia.com/blog/even-easier-introduction-cuda/
  - CUDA programming manual
    - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
    - ch 2.1-2.4, 2.6
    - ch 3.1-3.2.4
    - ch 4.1-.4.2
    - ch 5 Performance guidelines
  - Best Practices Guide:
    - https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html