Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Information Technology
and Electrical Engineering

GPU, GP-GPU, GPU computing

Luca Benini

lbenini@iis.ee.ethz.ch

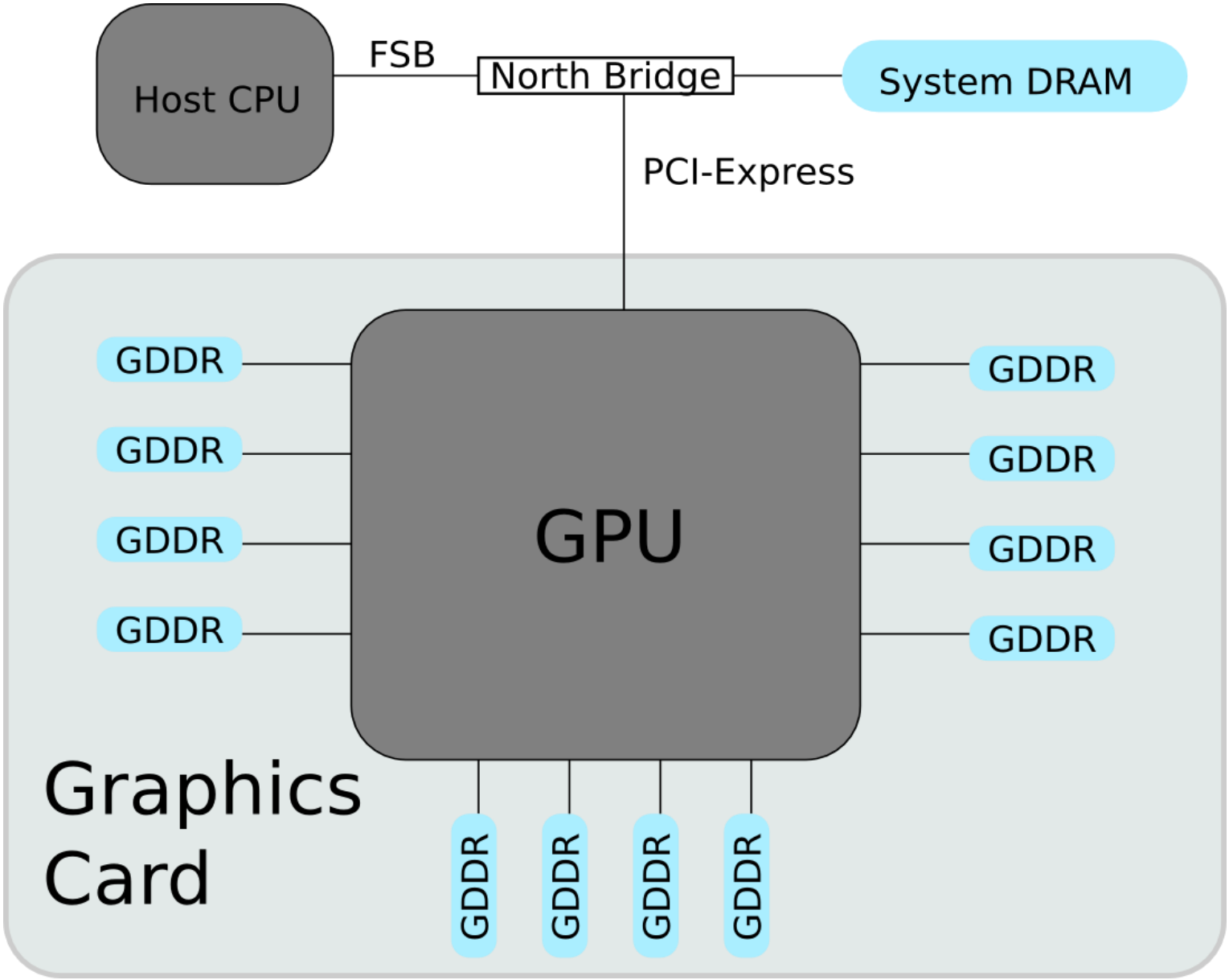
(GP-) GPU Today

601mm^2

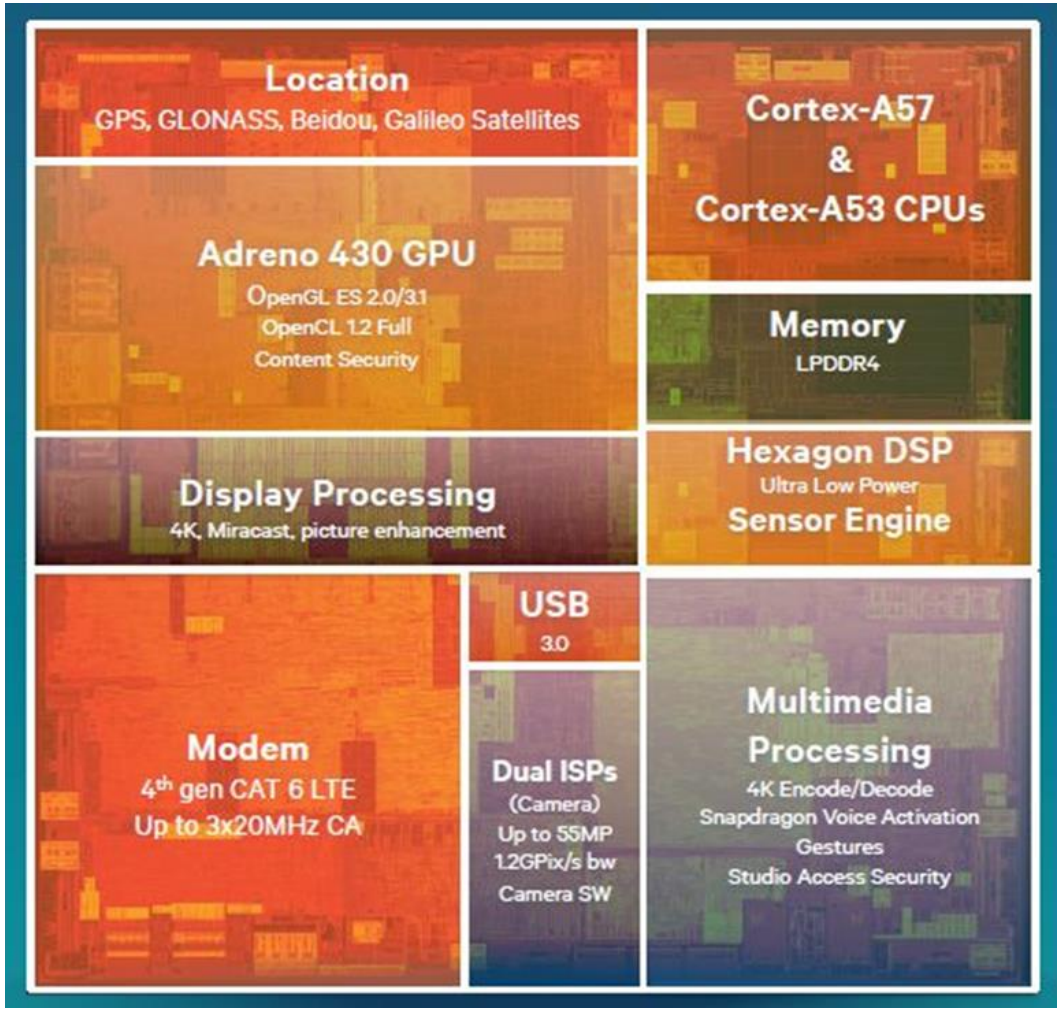
NVIDIA GPU Specification Comparison				
	GTX Titan X	GTX 980	GTX Titan Black	GTX Titan
CUDA Cores	3072	2048	2880	2688
Texture Units	192	128	240	224
ROPs	96	64	48	48
Core Clock	1000MHz	1126MHz	889MHz	837MHz
Boost Clock	1075MHz	1216MHz	980MHz	876MHz
Memory Clock	7GHz GDDR5	7GHz GDDR5	7GHz GDDR5	6GHz GDDR5
Memory Bus Width	384-bit	256-bit	384-bit	384-bit
VRAM	12GB	4GB	6GB	6GB
FP64	1/32 FP32	1/32 FP32	1/3 FP32	1/3 FP32
TDP	250W	165W	250W	250W
GPU	GM200	GM204	GK110B	GK110
Architecture	Maxwell 2	Maxwell 2	Kepler	Kepler
Transistor Count	8B	5.2B	7.1B	7.1B
Manufacturing Process	TSMC 28nm	TSMC 28nm	TSMC 28nm	TSMC 28nm
Launch Date	03/17/2015	09/18/14	02/18/2014	02/21/2013

From TU/e 5kk73 Zhenyu Ye Henk Corporaal 2013 (update 2015 Benini)

A Typical GPU Card



GPU Is In Your Pocket Too



Adreno 430 (onboard the Snapdragon 810)

Maxwell 2 Architecture: GM204 (GTX980)

	680	780	980
TFLOPS	3	4	5
MEMORY	2GB	3GB	4GB
PERFORMANCE	1	1.5	2
POWER	195W	250W	165W
GFLOPS / WATT	15	15	30

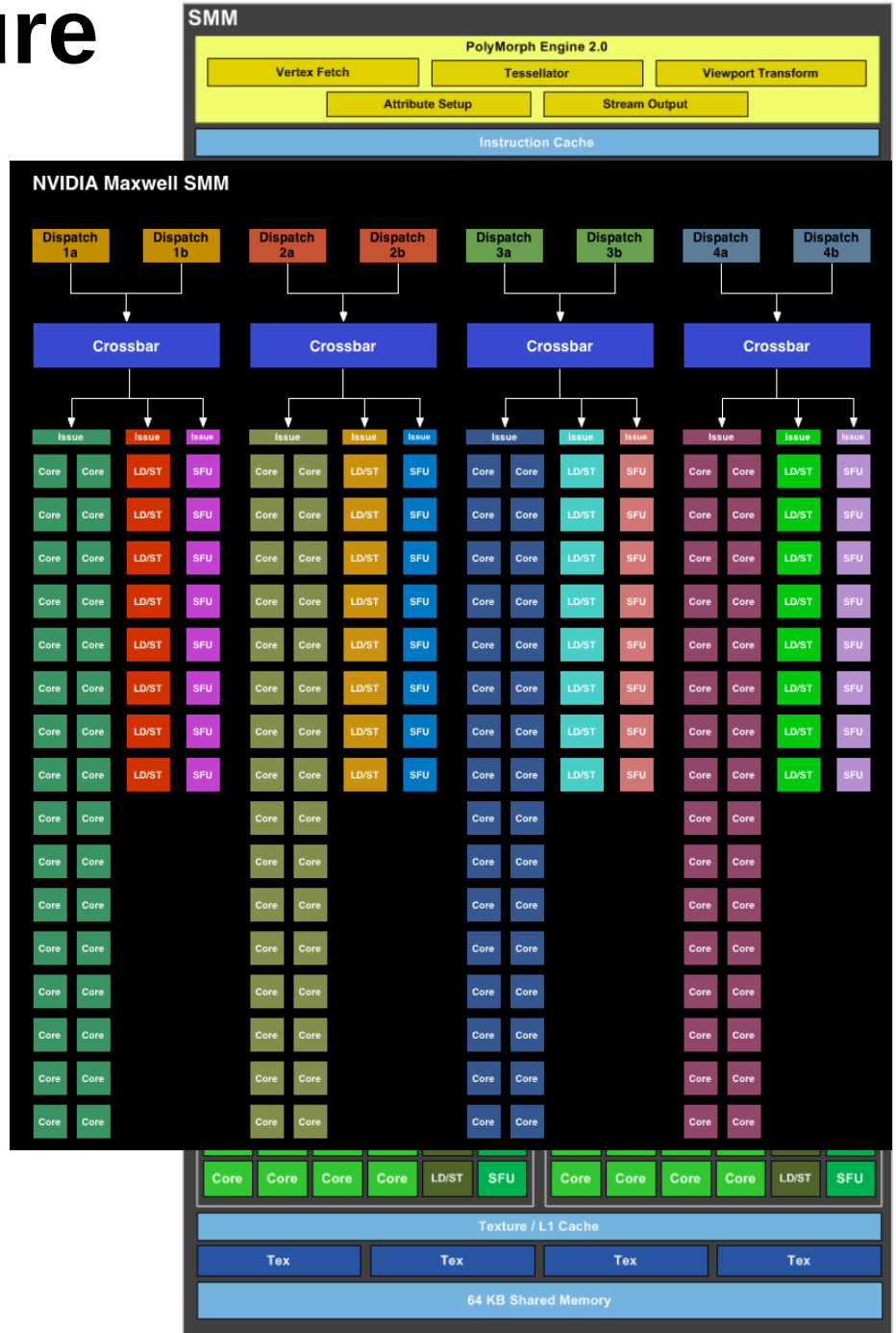
1.2GHZ CLK, 1.2v Vdd
2048Cuda Cores

GTX 980 Memory Specs:	
Memory Clock	7.0 Gbps
Standard Memory Config	4 GB
Memory Interface	GDDR5
Memory Interface Width	256-bit
Memory Bandwidth (GB/sec)	224

Maxwell 2 Architecture

SMM has four warp schedulers,

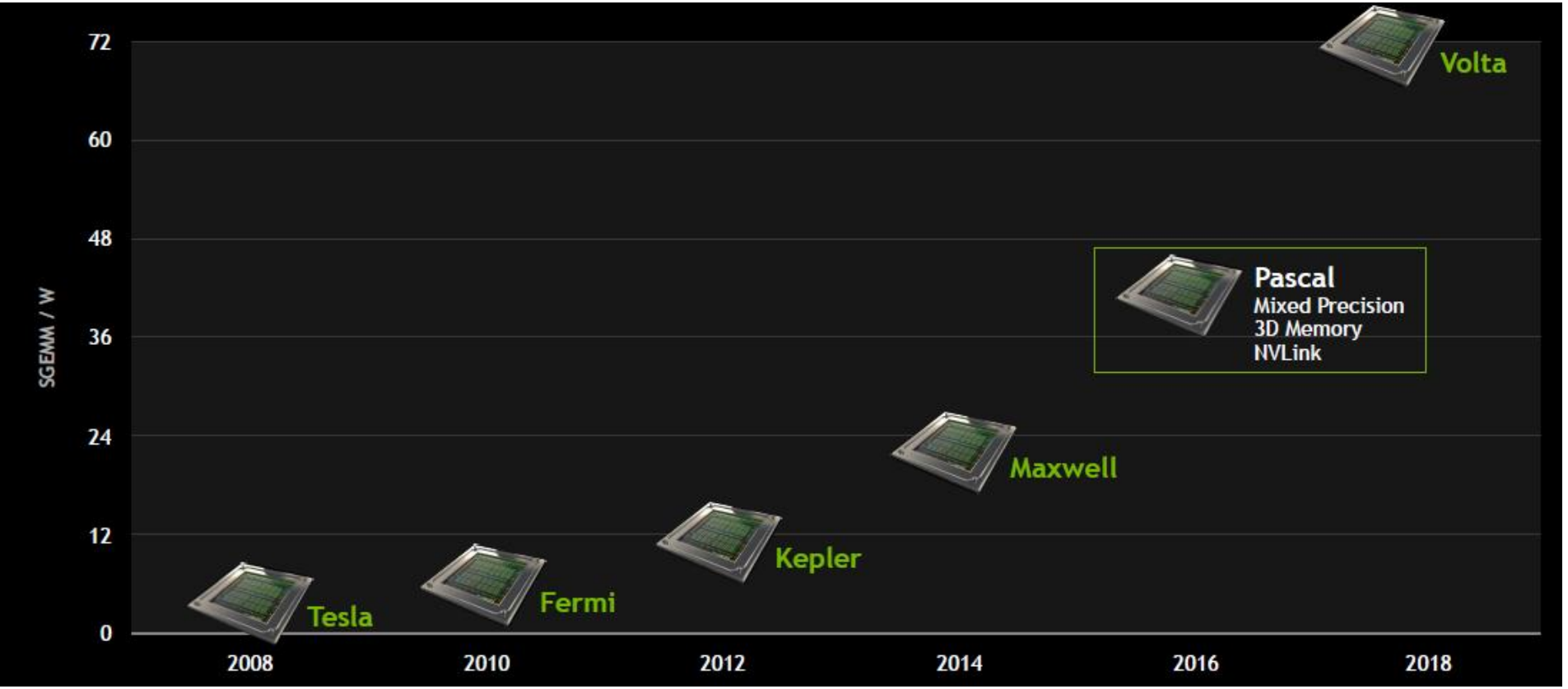
- All core SMM functional units are assigned to a particular scheduler, with no shared units.
- Each warp scheduler has the flexibility to dual-issue (such as issuing a math operation to a CUDA Core in the same cycle as a memory operation to a load/store unit)
- SMM 64k 32-bit registers and 64 warps max. Maximum number of registers per thread (255).
- Maximum number of active thread blocks per multiprocessor has been doubled over SMX to 32 - occupancy improvement for kernels that use small thread blocks of 64 or fewer threads
- 64KB of dedicated shared memory per SM



Ref. ANANDTECH 14

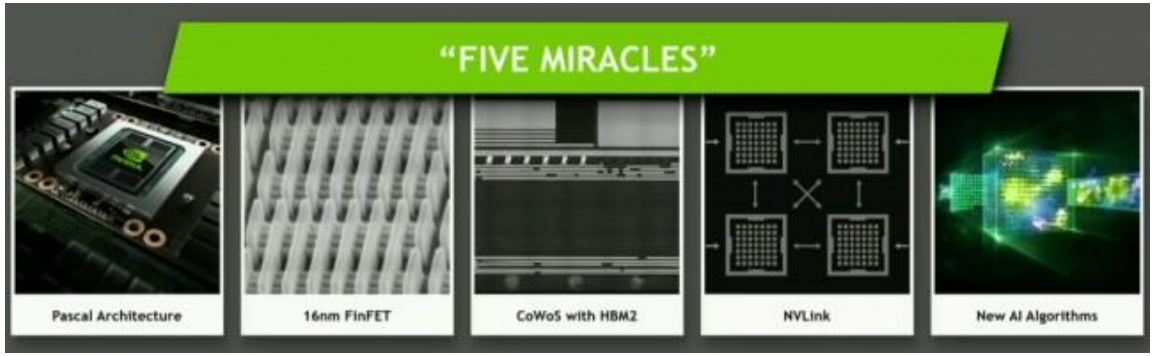
GPU Architecture evolution

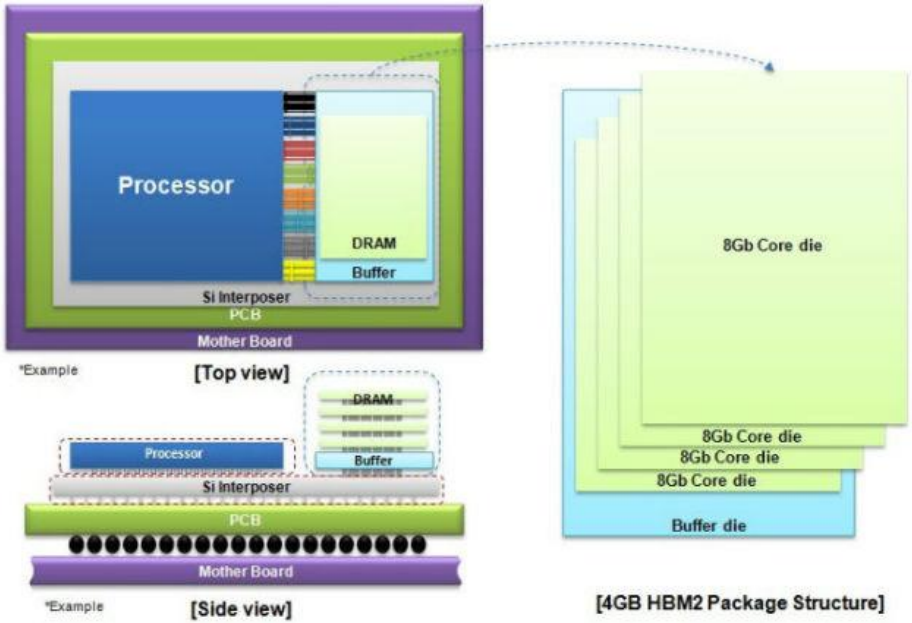
From Nvidia GTC2015



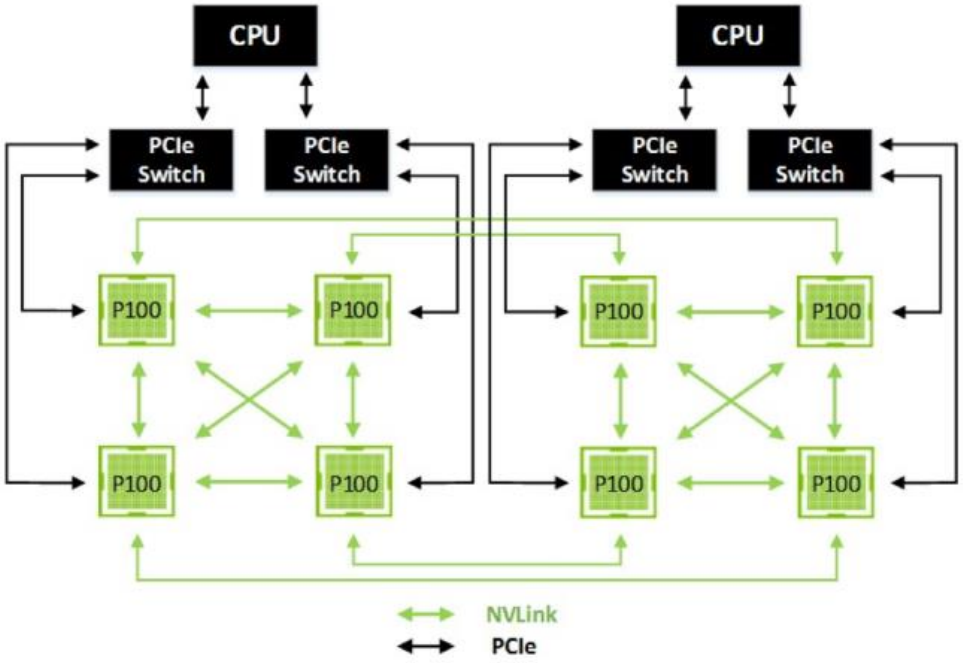
Pascal Architecture

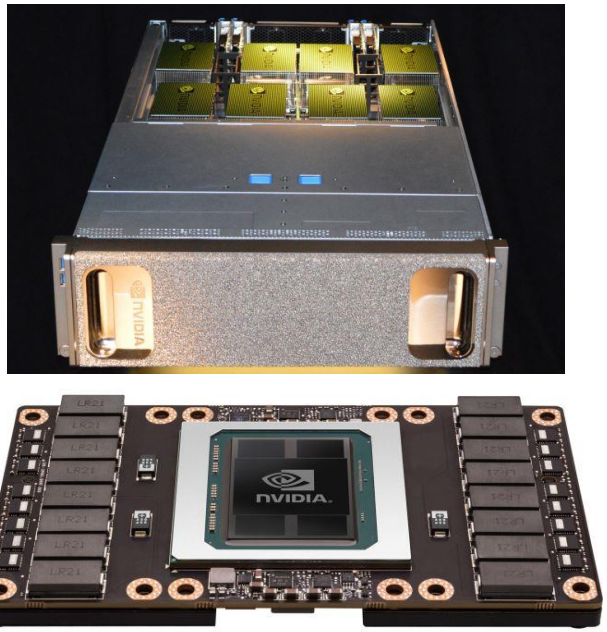
Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA Cores / SM	192	128	64
FP32 CUDA Cores / GPU	2880	3072	3584
FP64 CUDA Cores / SM	64	4	32
FP64 CUDA Cores / GPU	960	96	1792
Base Clock	745 MHz	948 MHz	1328 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz
FP64 GFLOPs	1680	213	5304[1]
Texture Units	240	192	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion
GPU Die Size	551 mm²	601 mm²	610 mm²
Manufacturing Process	28-nm	28-nm	16-nm



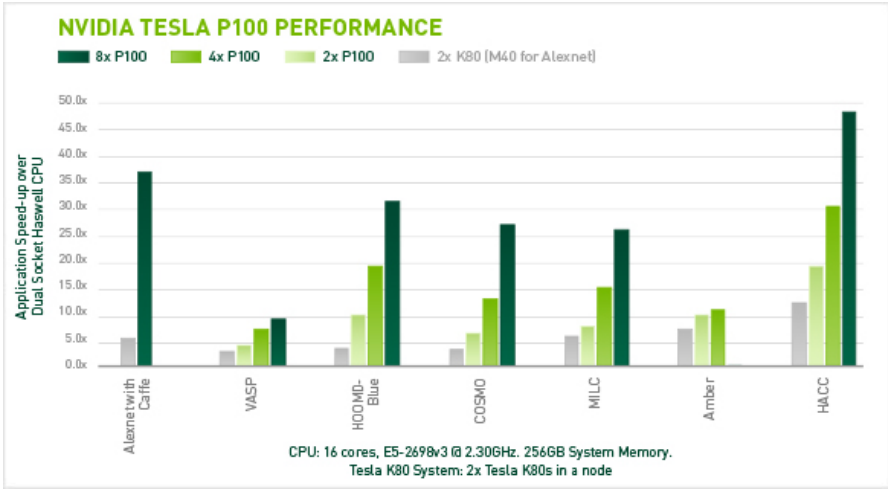


Nvidia DGX-1





NVIDIA DGX-1 Specifications	
CPUs	2x Intel Xeon E5-2698 v3 (16 core, Haswell-EP)
GPUs	8x NVIDIA Tesla P100 (3584 CUDA Cores)
System Memory	512GB DDR4-2133 (LRDIMM)
GPU Memory	128GB HBM2 (8x 16GB)
Storage	4x Samsung PM863 1.92TB SSDs
Networking	4x Infiniband EDR 2x 10GigE
Power	3200W
Size	3U Rackmount
GPU Throughput	FP16: 170 TFLOPs FP32: 85 TFLOPs FP64: 42.5 TFLOPs



Why GPUs?

- Graphics workloads are embarrassingly parallel
  - Data-parallel
  - Pipeline-parallel
- CPU and GPU execute in parallel
- Hardware: texture filtering, rasterization, etc.

Data Parallel

- *Beyond Graphics*
  - Cloth simulation
  - Particle system
  - Matrix multiply

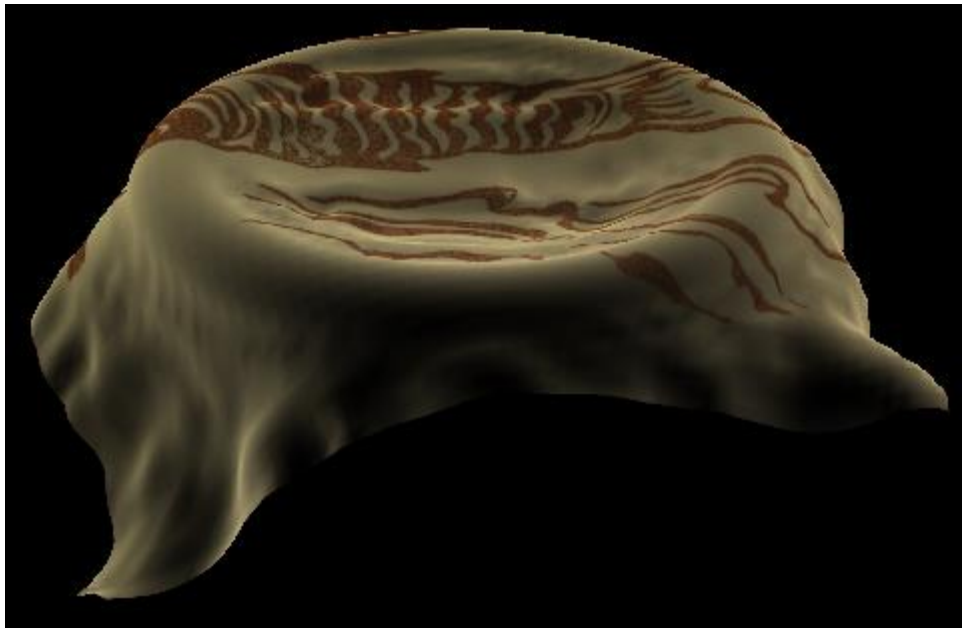
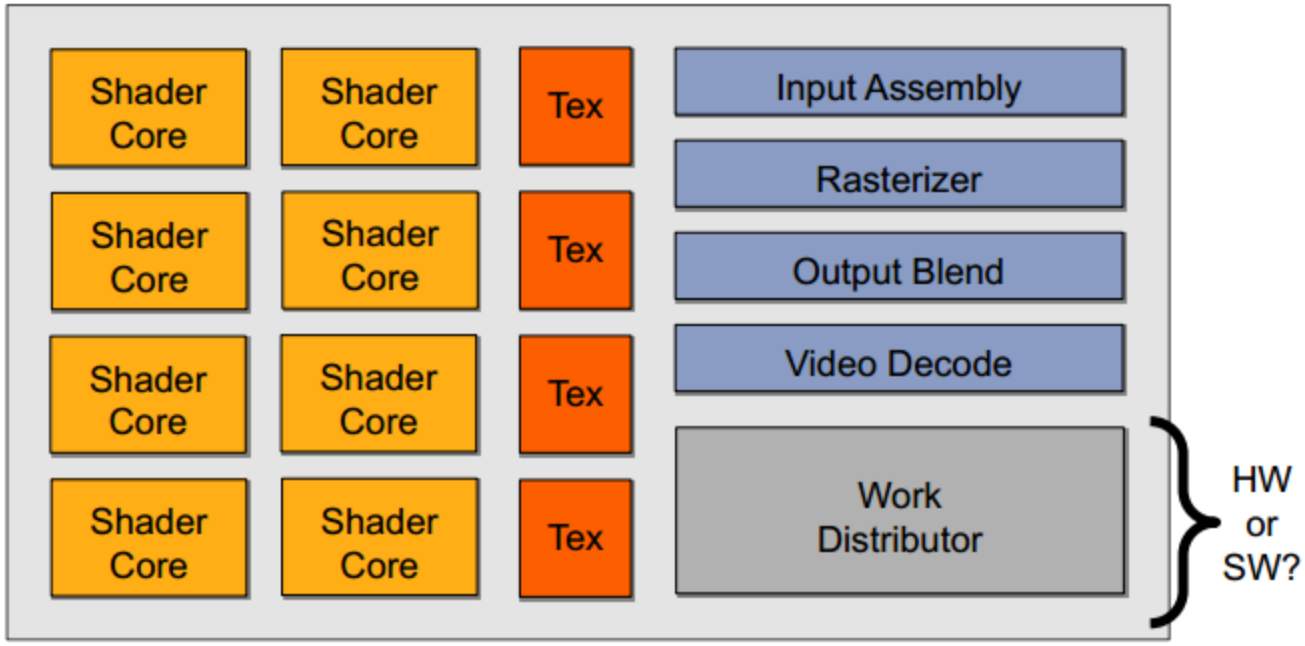


Image from: https://plus.google.com/u/0/photos/100838748547881402137/albums/5407605084626995217/5581900335460078306

What's in a GPU?

A GPU is a heterogeneous chip multi-processor (highly tuned for graphics)



A diffuse reflectance shader

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

Shader programming model:

Fragments are processed independently,
but there is no explicit parallel programming

Compile shader

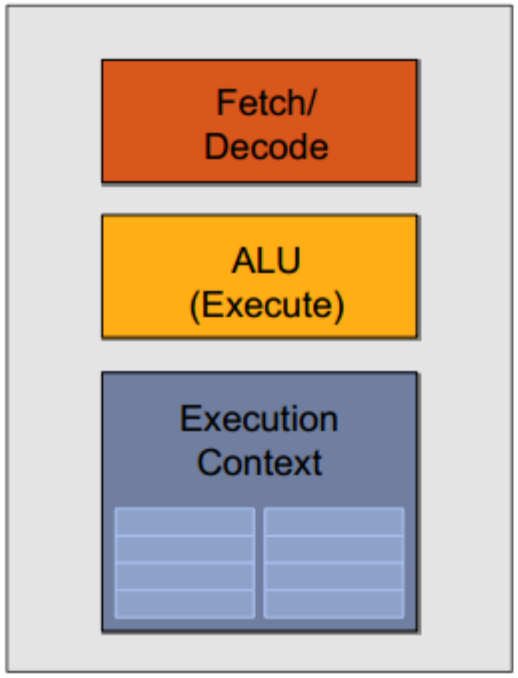
1 unshaded fragment input record

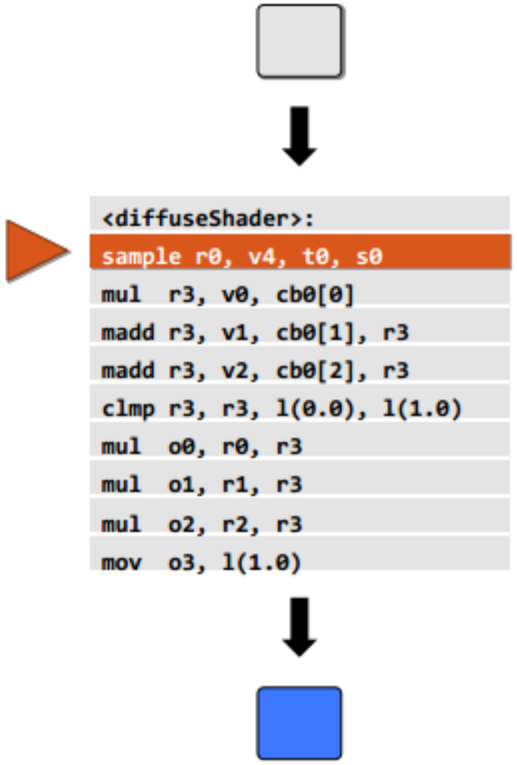
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

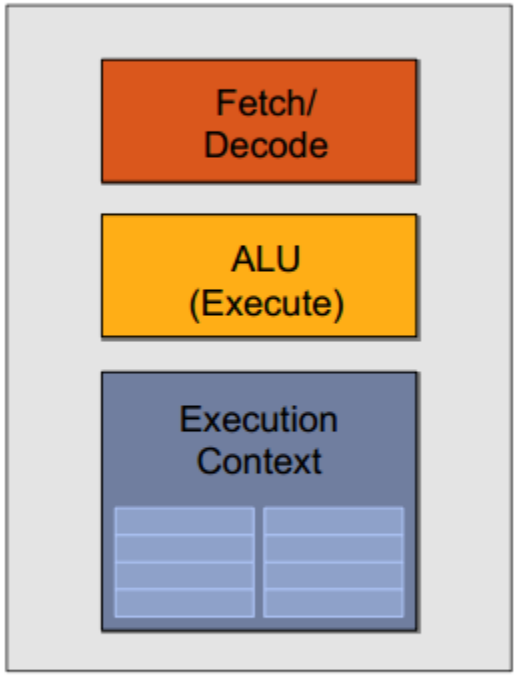
1 shaded fragment output record

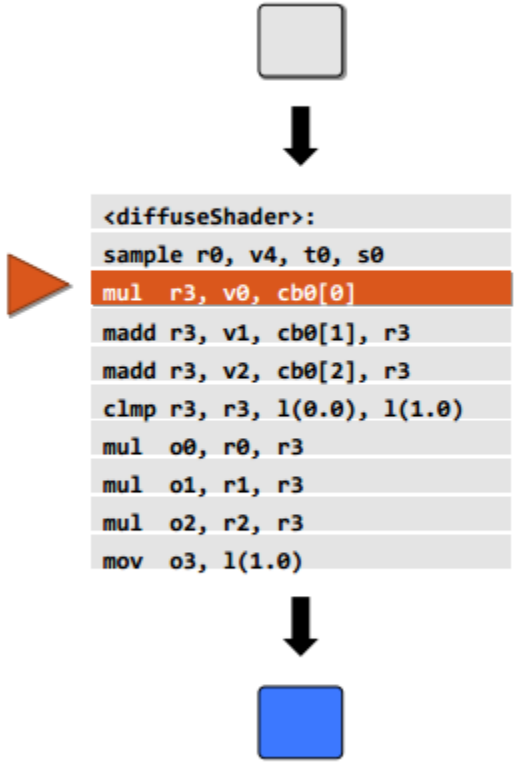
Execute shader



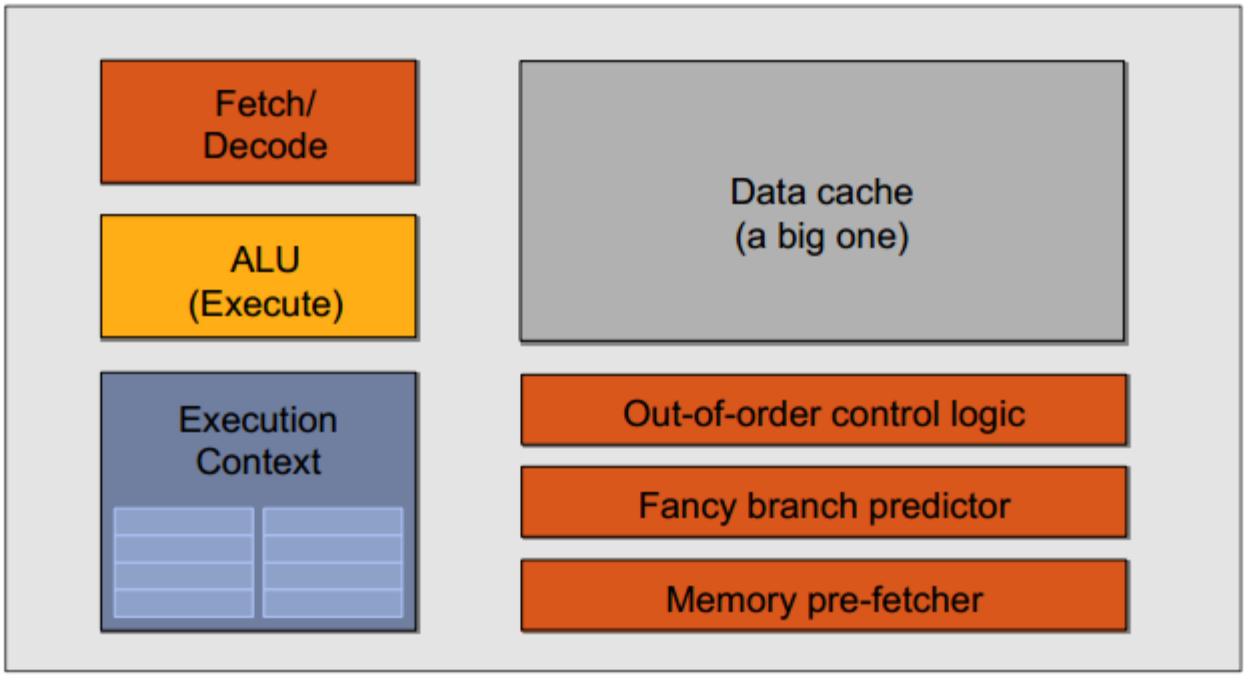


Execute shader

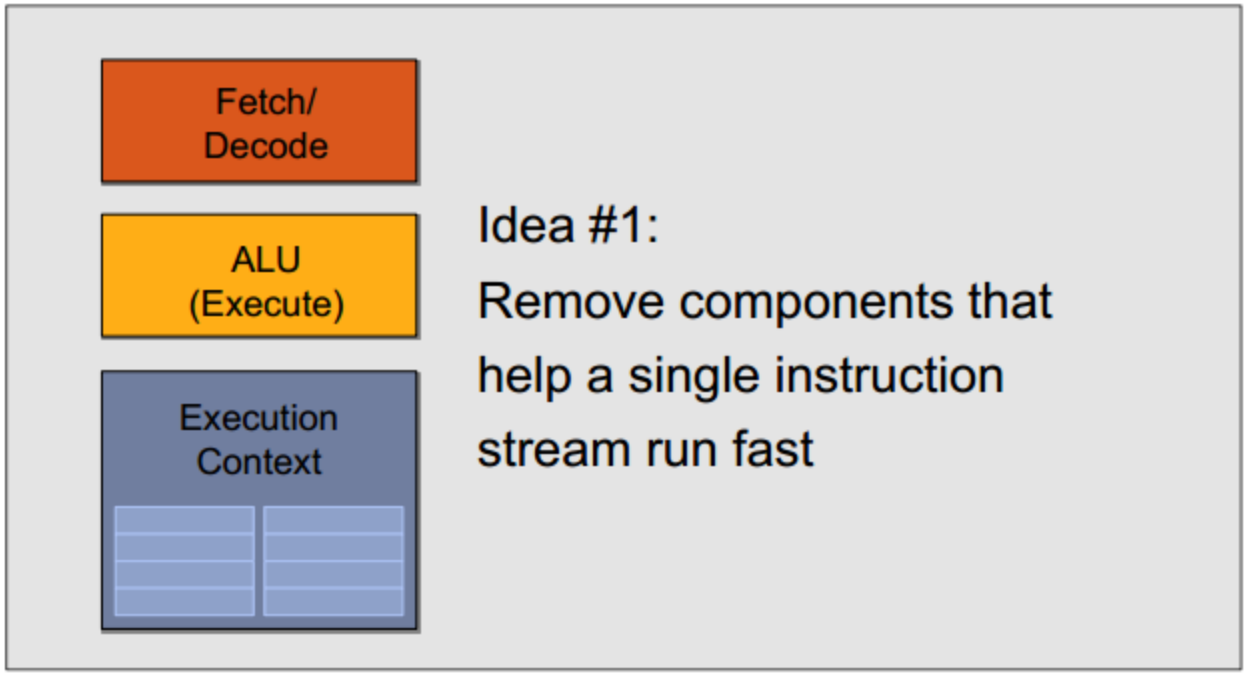




“CPU-style” cores



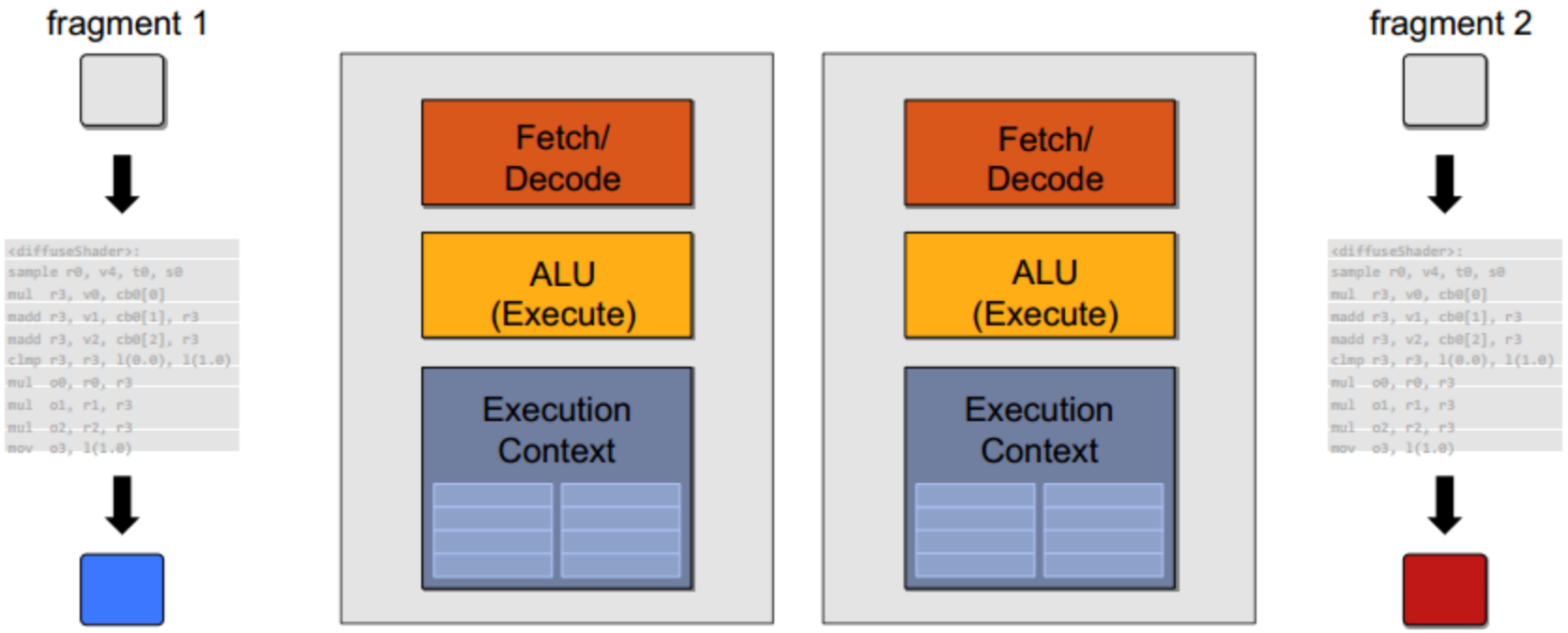
Slimming down



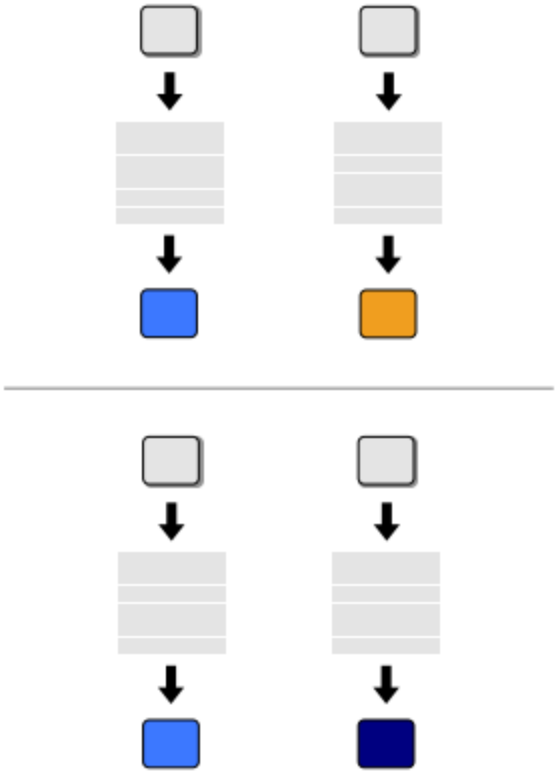
Idea #1:

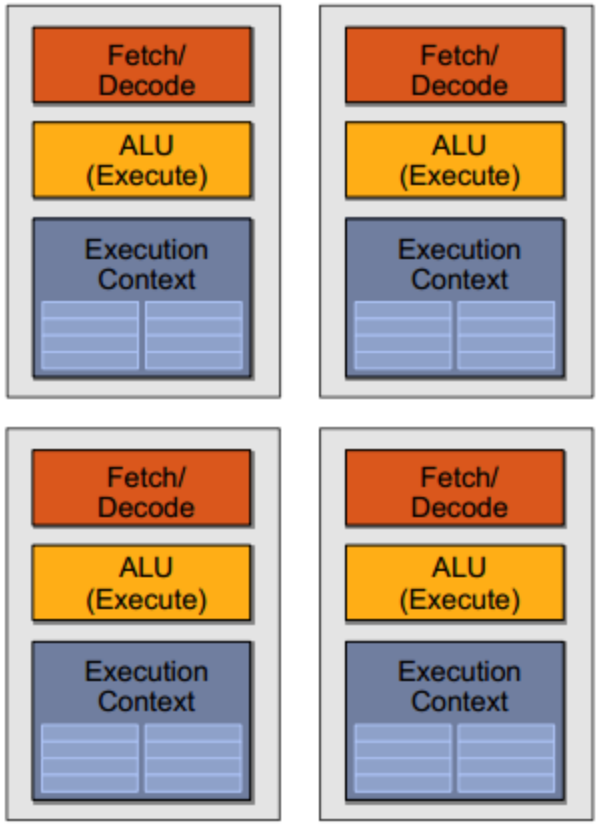
Remove components that help a single instruction stream run fast

Two cores (two fragments in parallel)

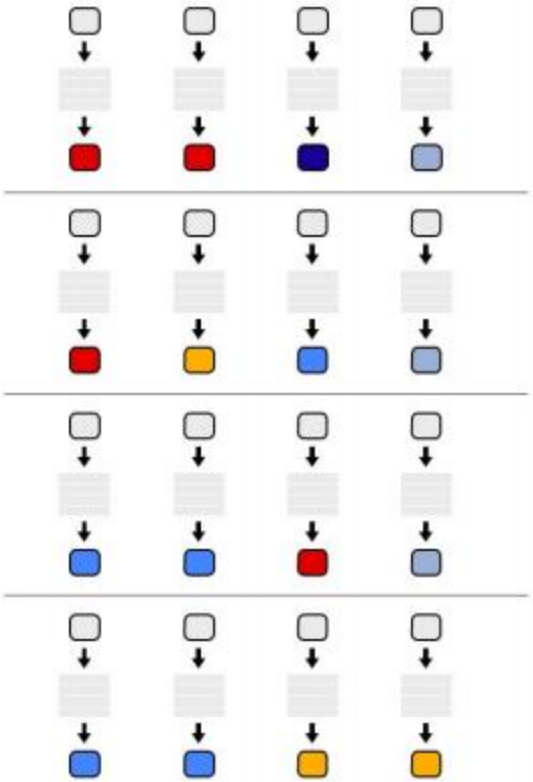


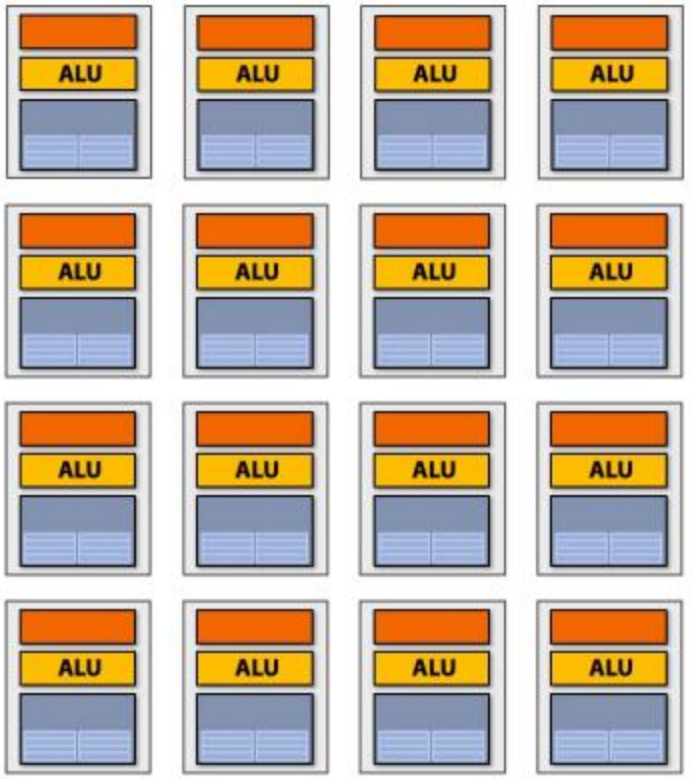
Four cores (four fragments in parallel)





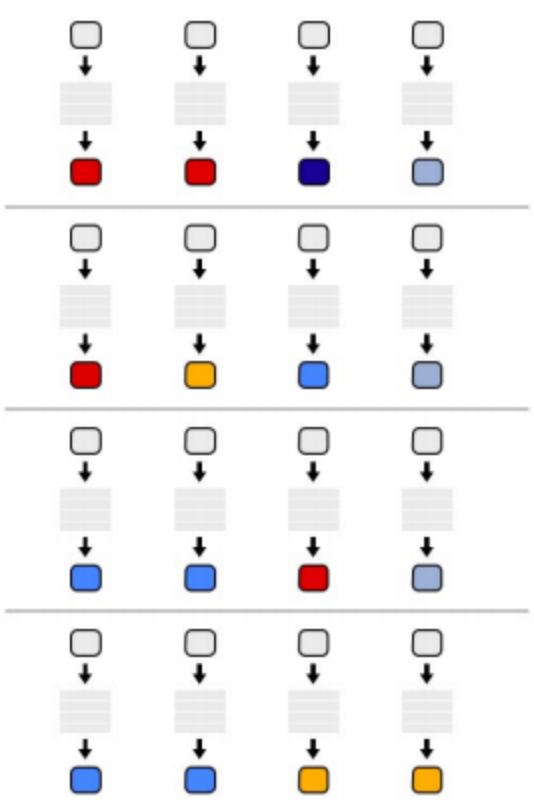
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

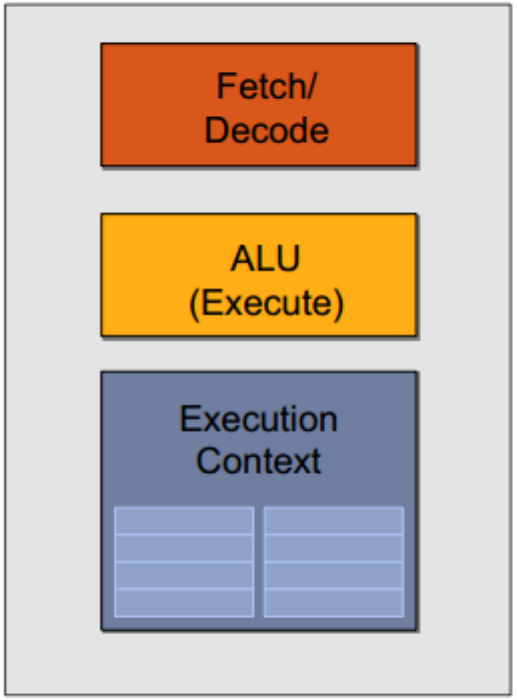
Instruction stream sharing



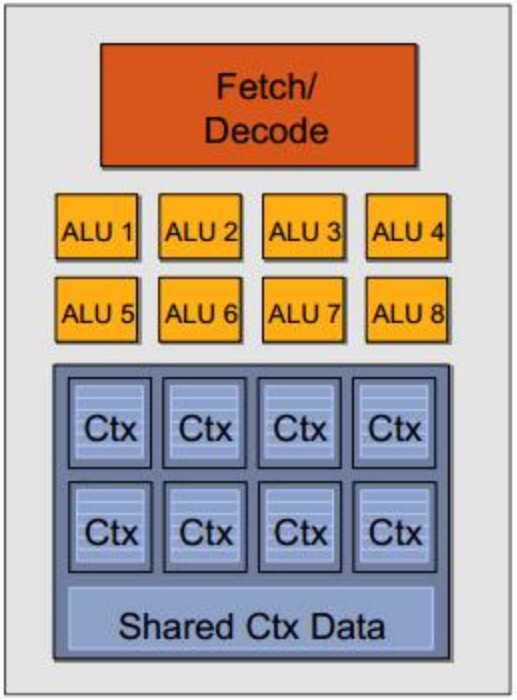
But ... many fragments should be able to share an instruction stream!

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Recall: simple processing core



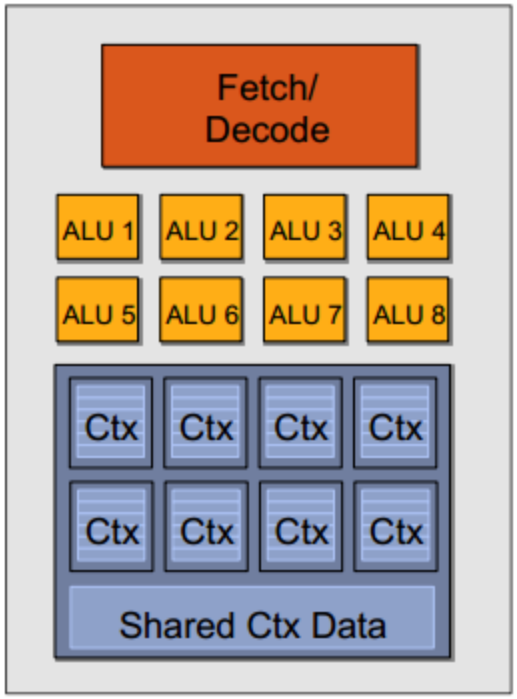
Add ALUs

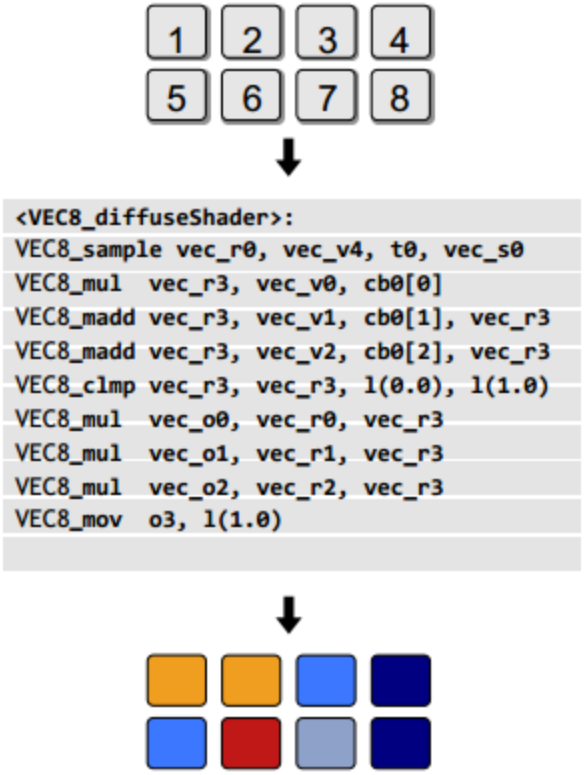


Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

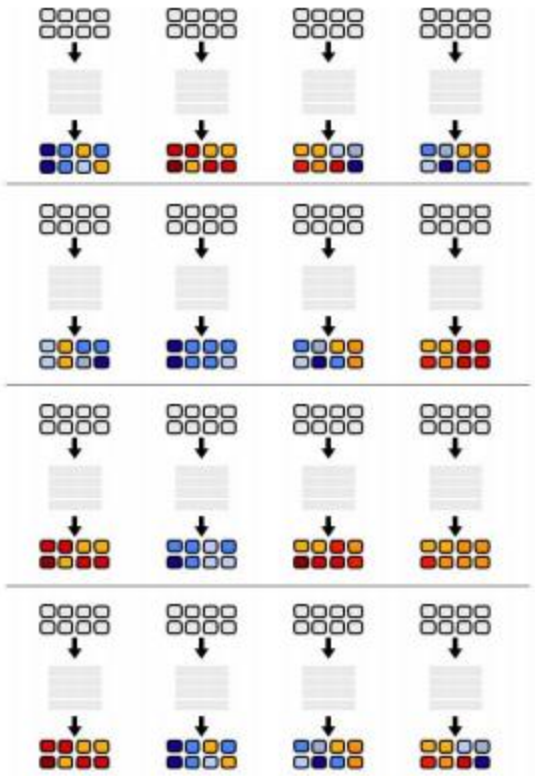
SIMD processing

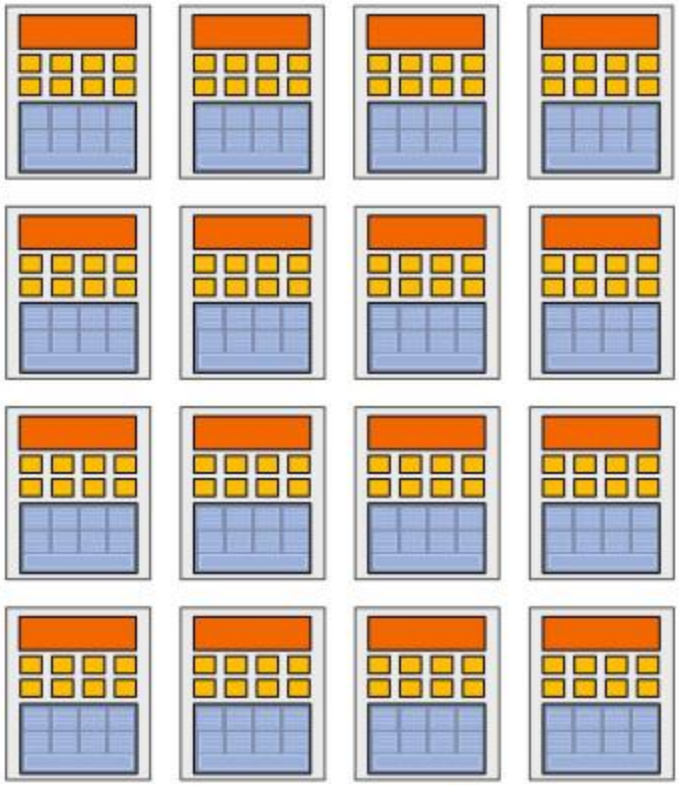
Modifying the shader





128 fragments in parallel

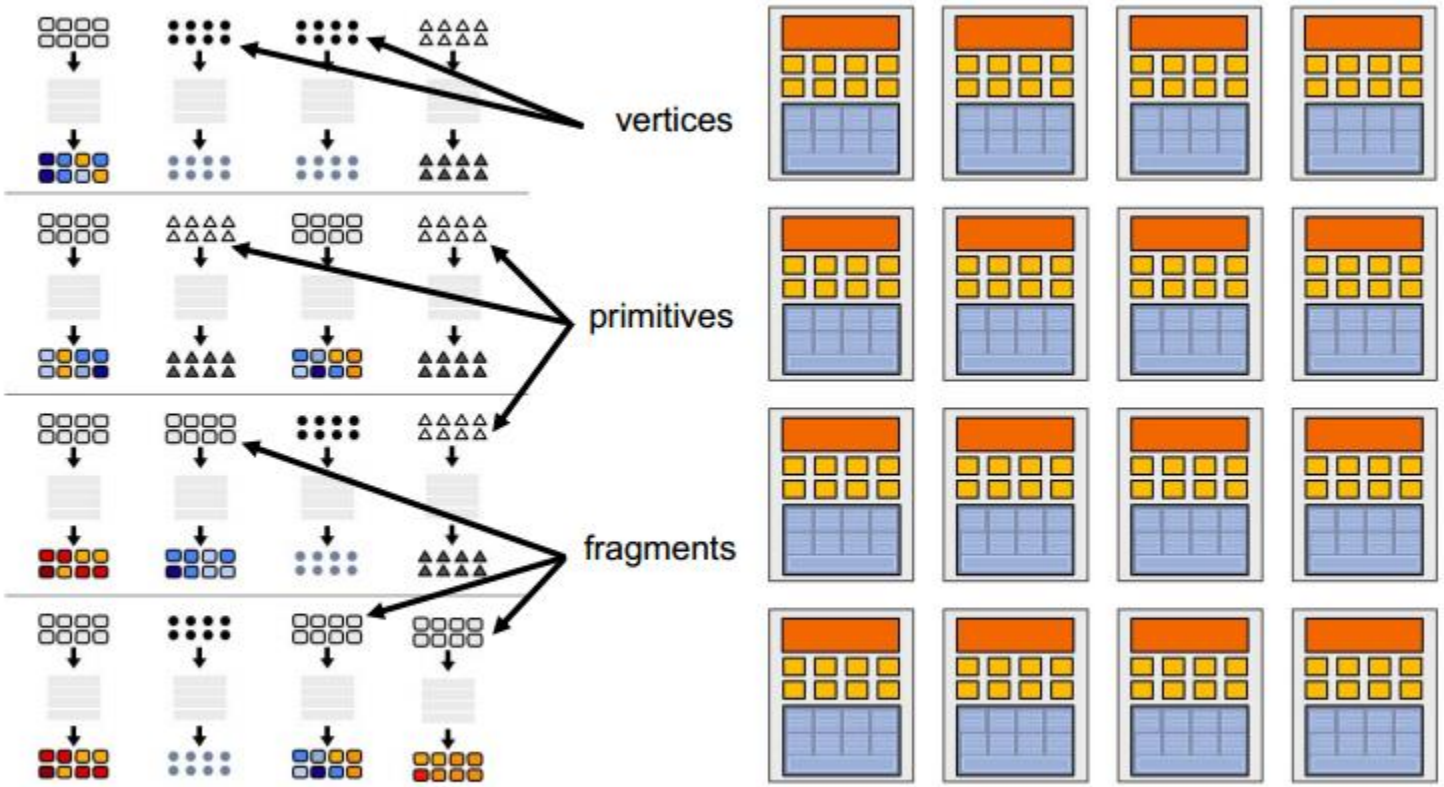




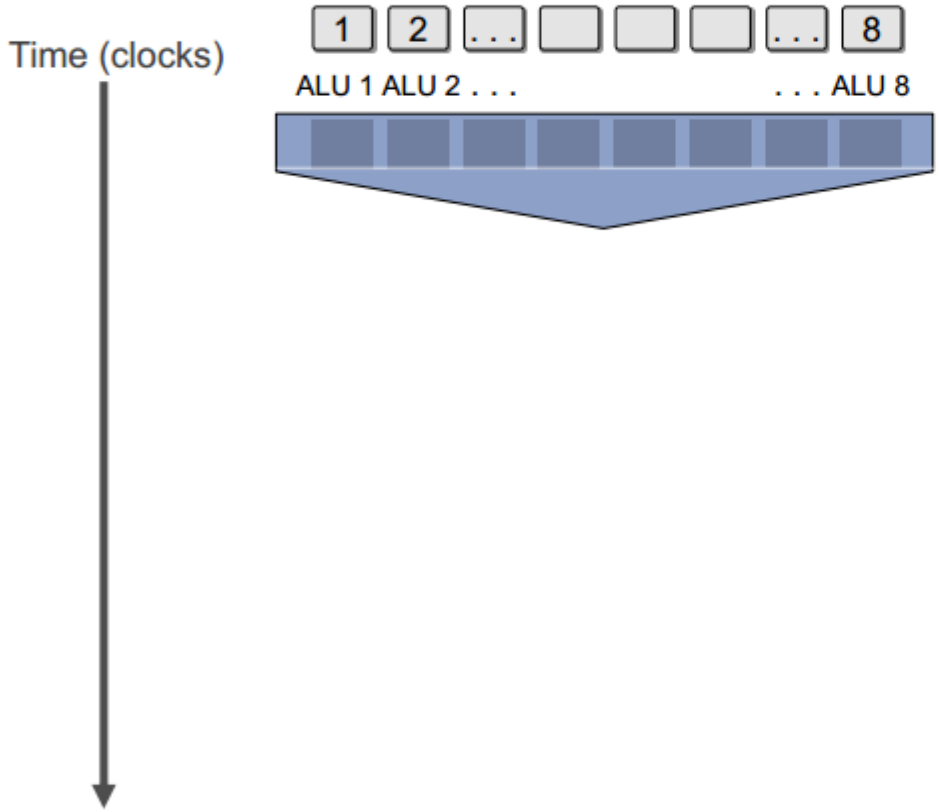
16 cores = 128 ALUs

, 16 simultaneous instruction streams

128 [vertices/fragments primitives OpenCL work items] in parallel

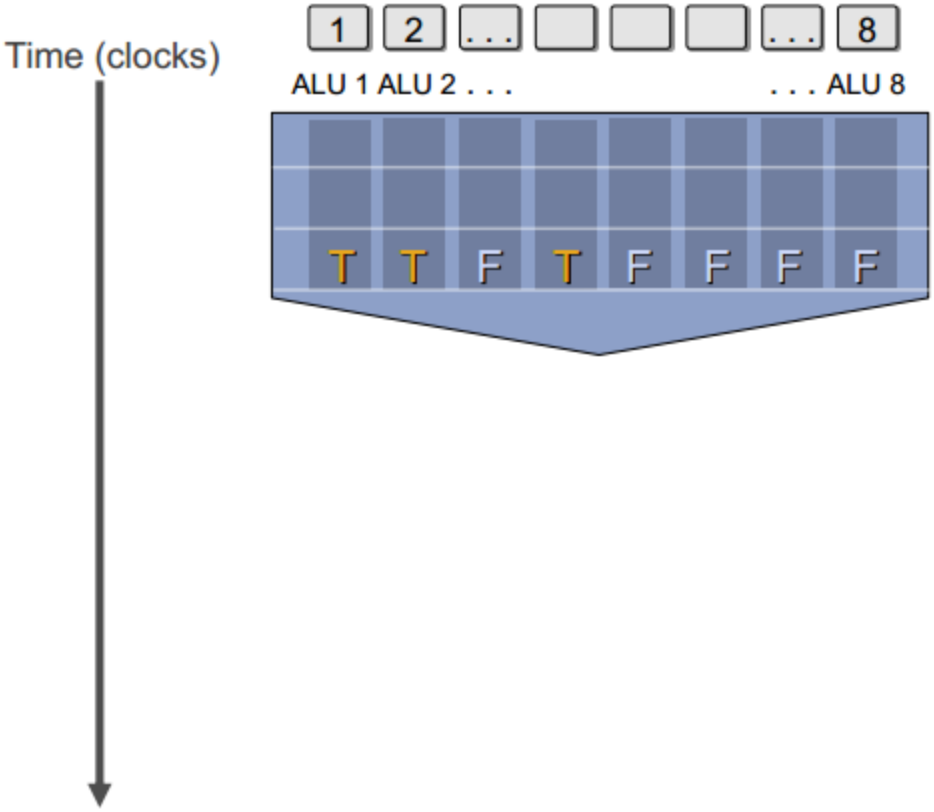


But what about branches?



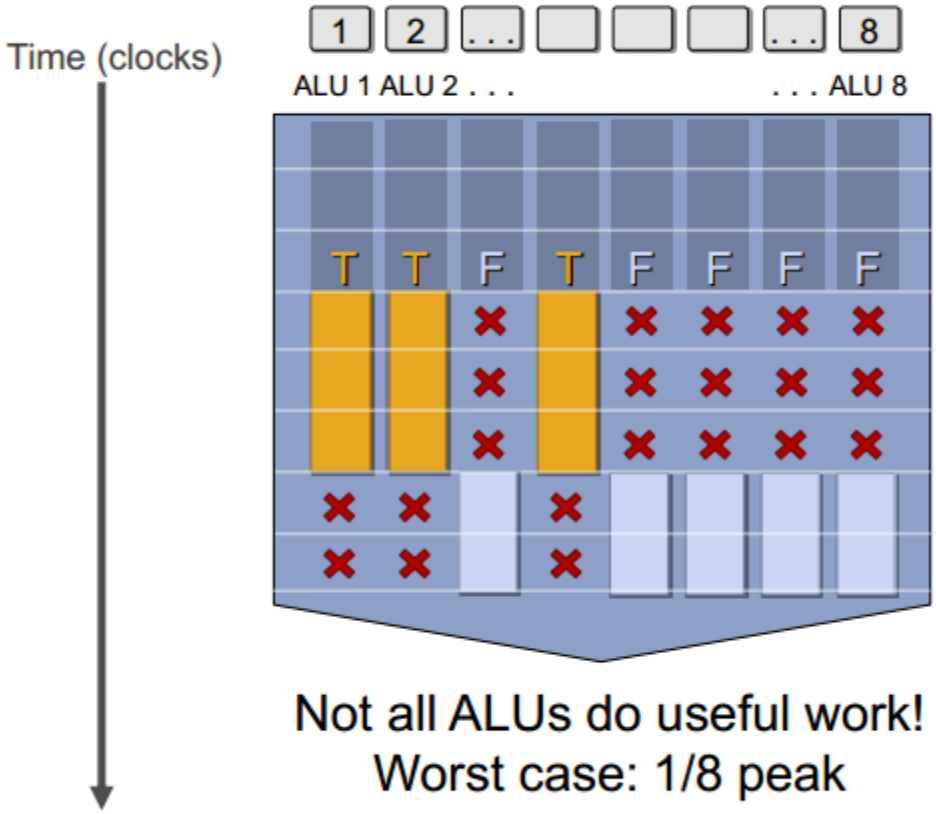

```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

But what about branches?




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

But what about branches?



Not all ALUs do useful work!
Worst case: 1/8 peak performance

```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

Clarification

SIMD processing does not imply SIMD instructions

- Option 1: explicit vector instructions
  - x86 SSE, AVX, Intel Larrabee
- Option 2: scalar instructions, implicit HW vectorization
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce ("SIMT" warps), ATI Radeon architectures ("wavefronts")

In practice: 16 to 64 fragments share an instruction stream.

Stalls!

Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.

Texture access latency = 100's to 1000's of cycles

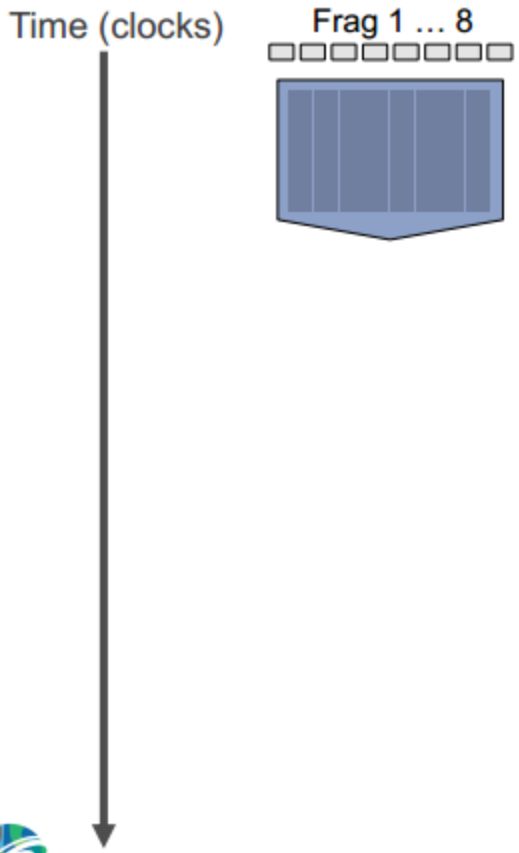
We've removed the fancy caches and logic that helps avoid stalls.

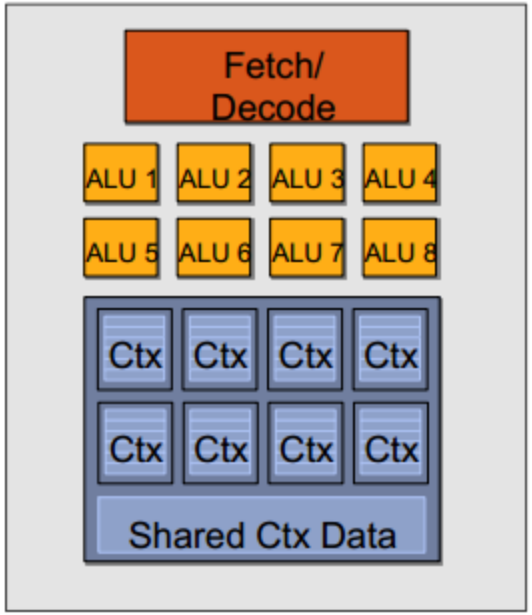
But we have LOTS of independent fragments.

Idea #3:

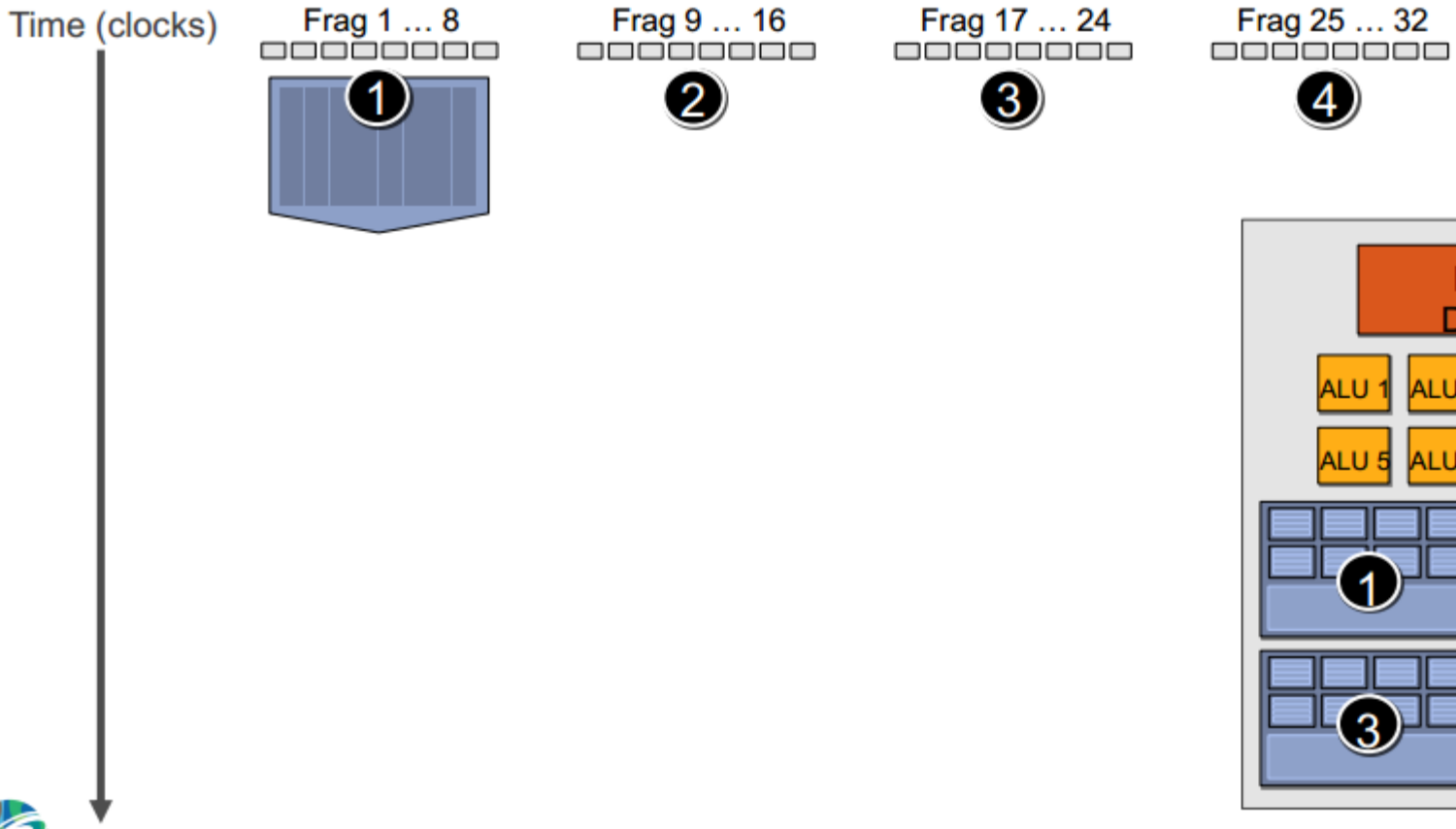
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

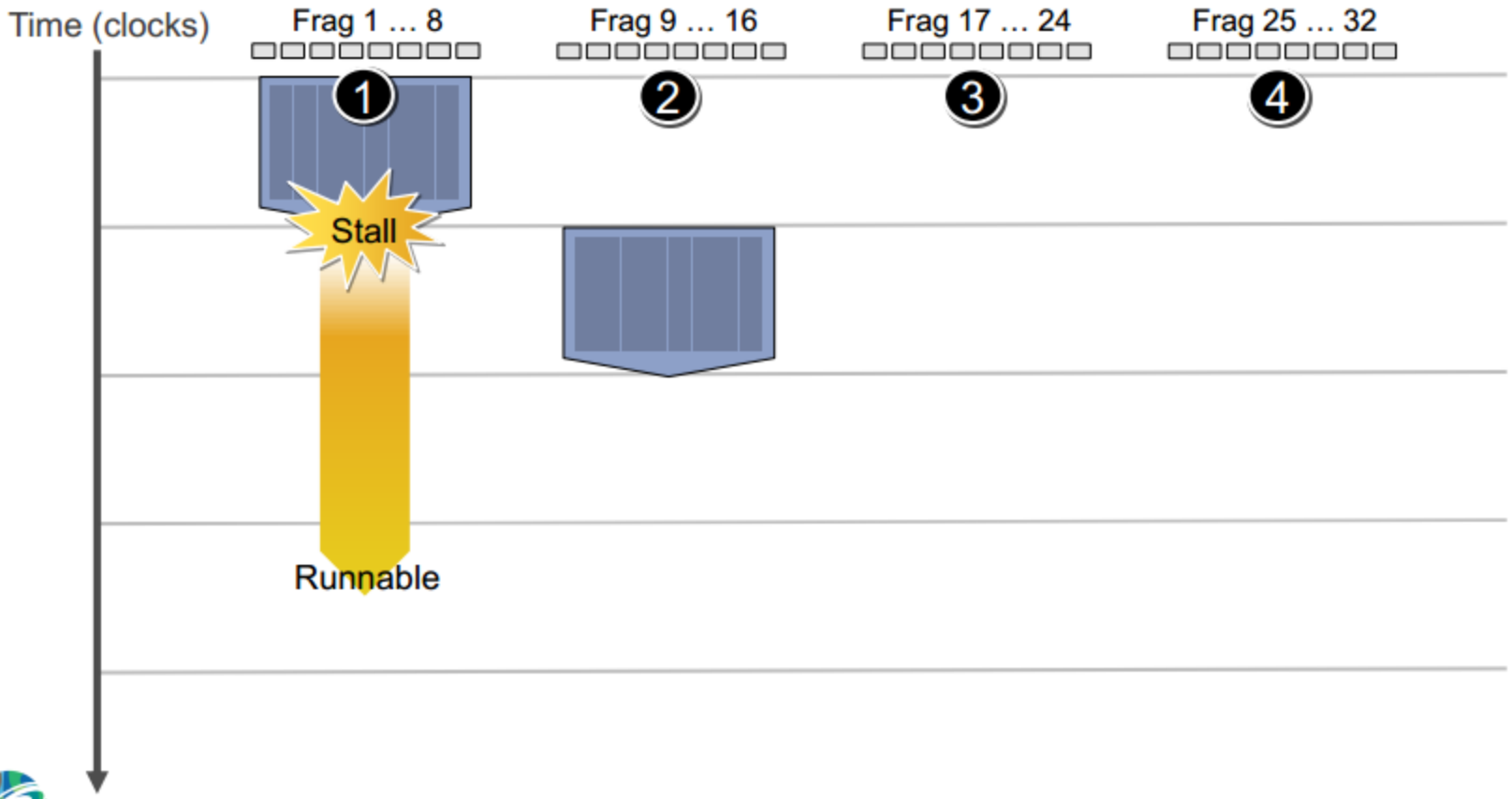




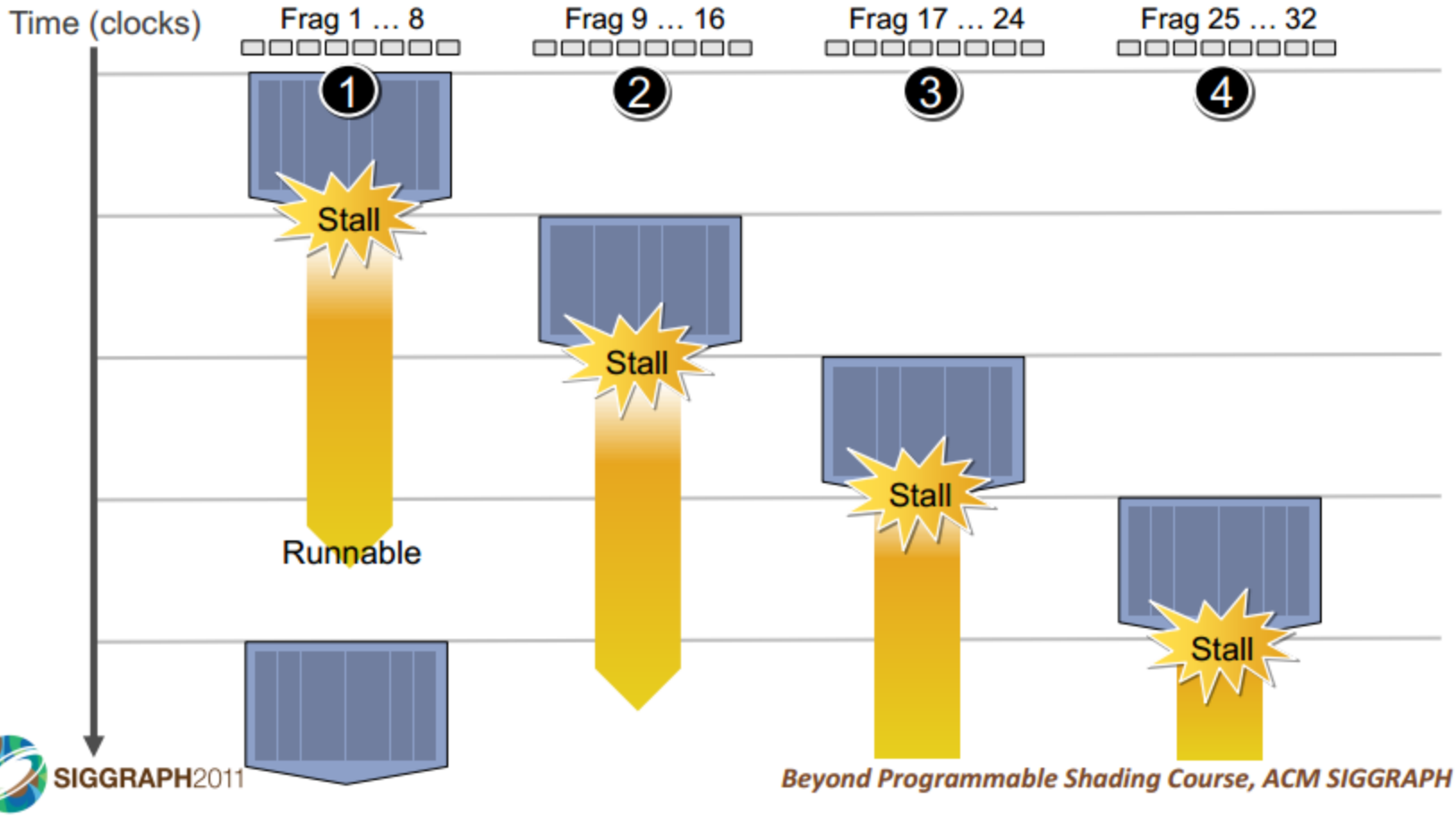
Hiding shader stalls



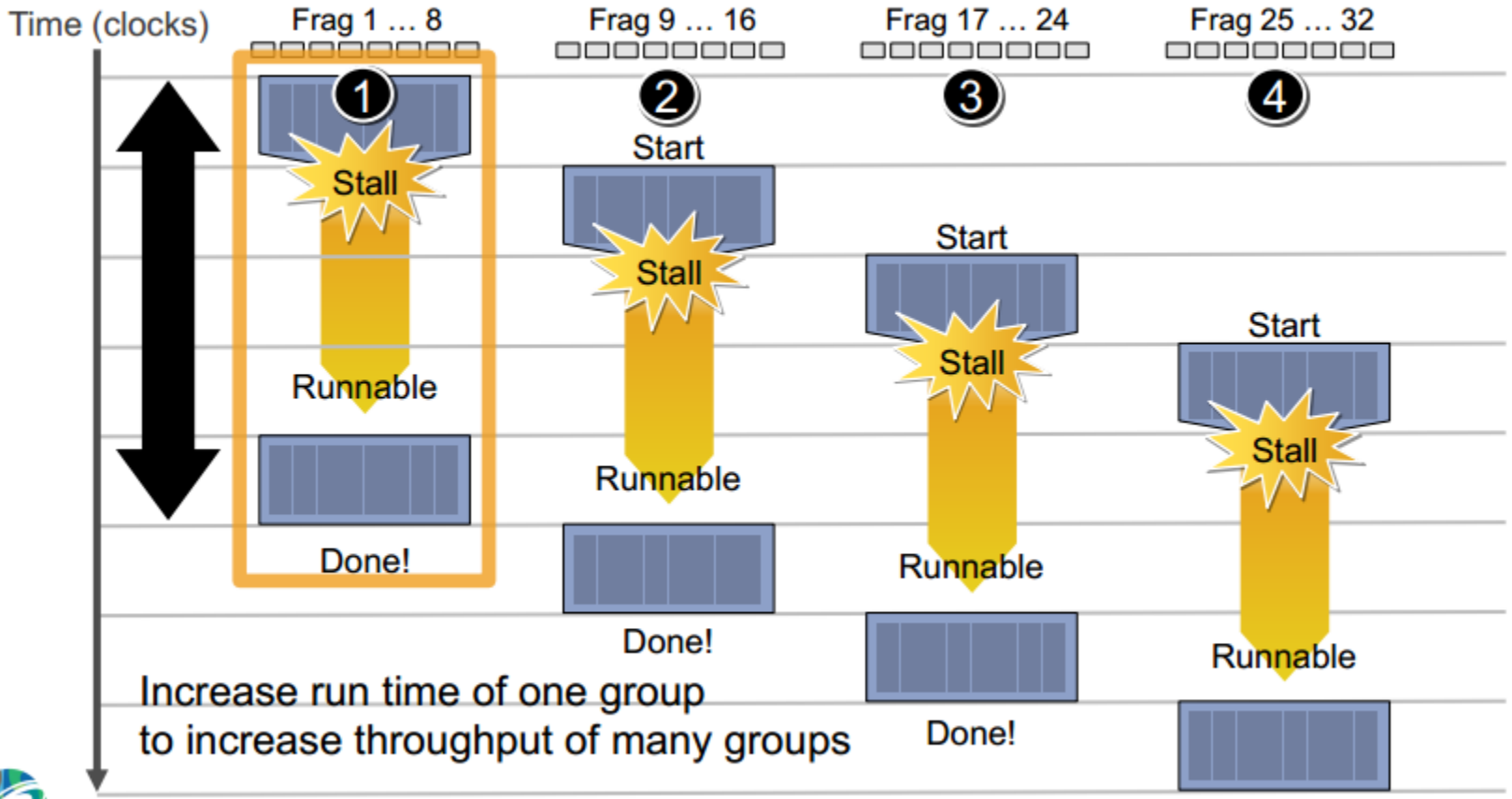
Hiding shader stalls



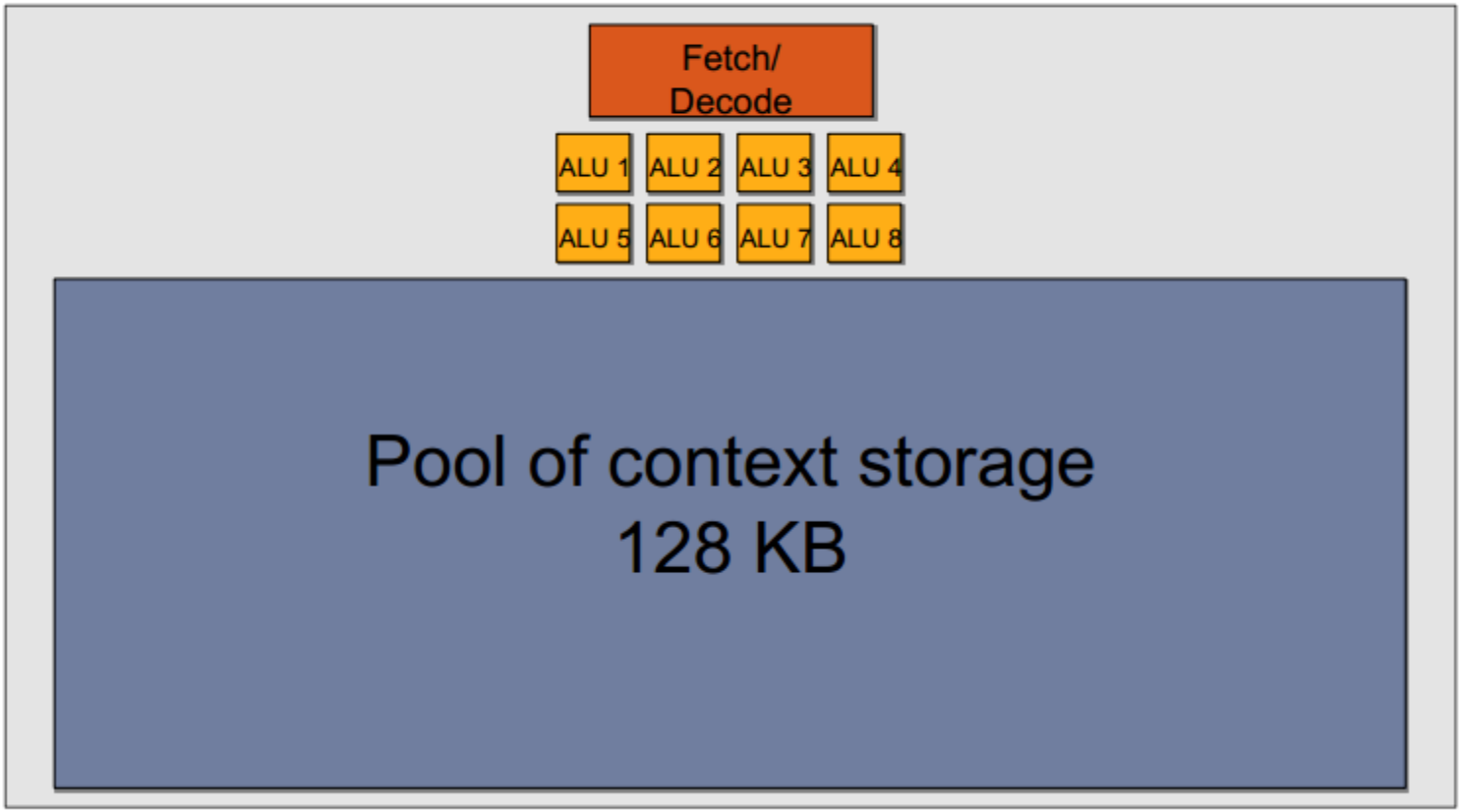
Hiding shader stalls



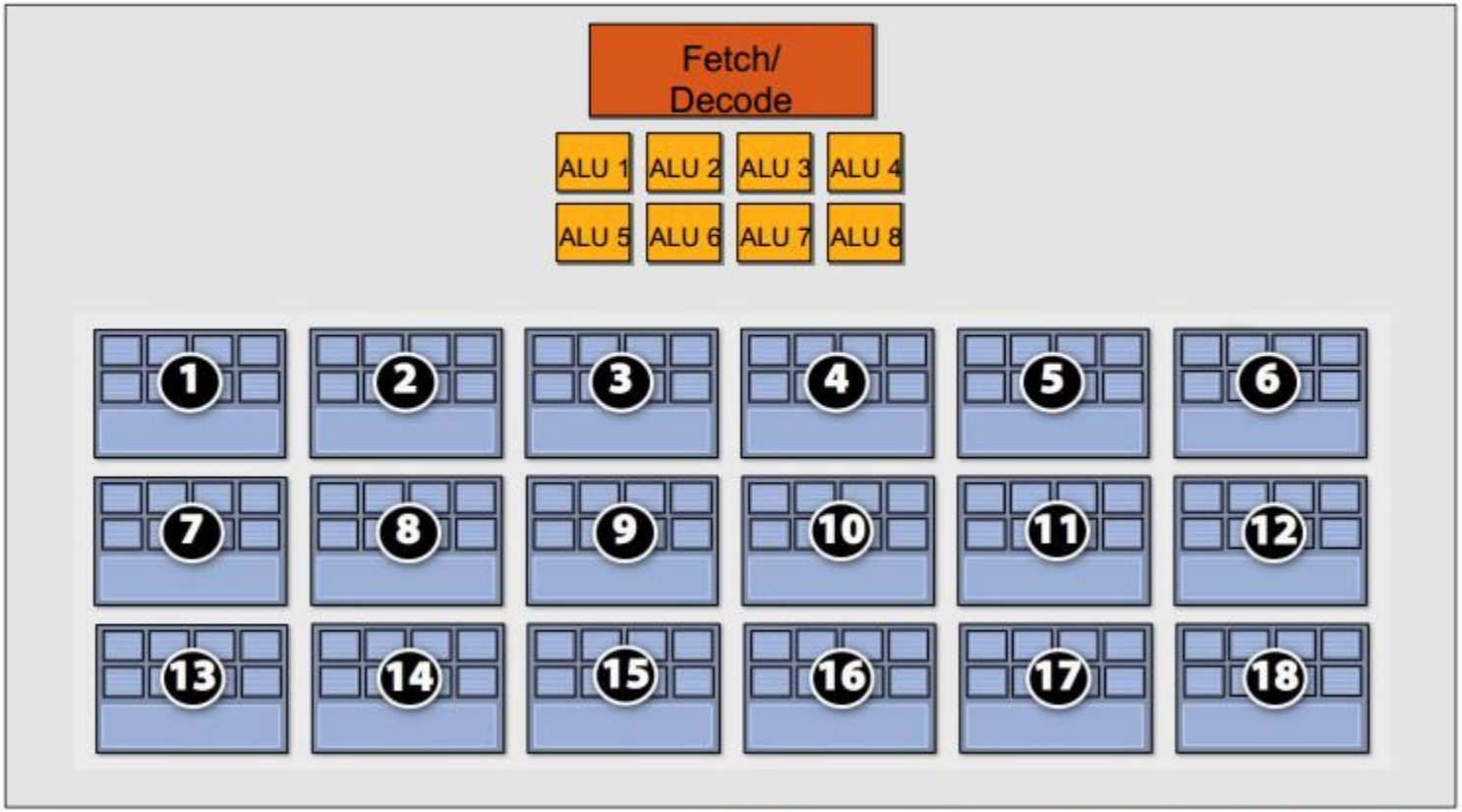
Throughput!


Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Start
Stall
Start
Stall
Start
Stall
Runnable
Runnable
Runnable
Runnable
Done!
Done!
Done!
Increase run time of one group
to increase throughput of many groups

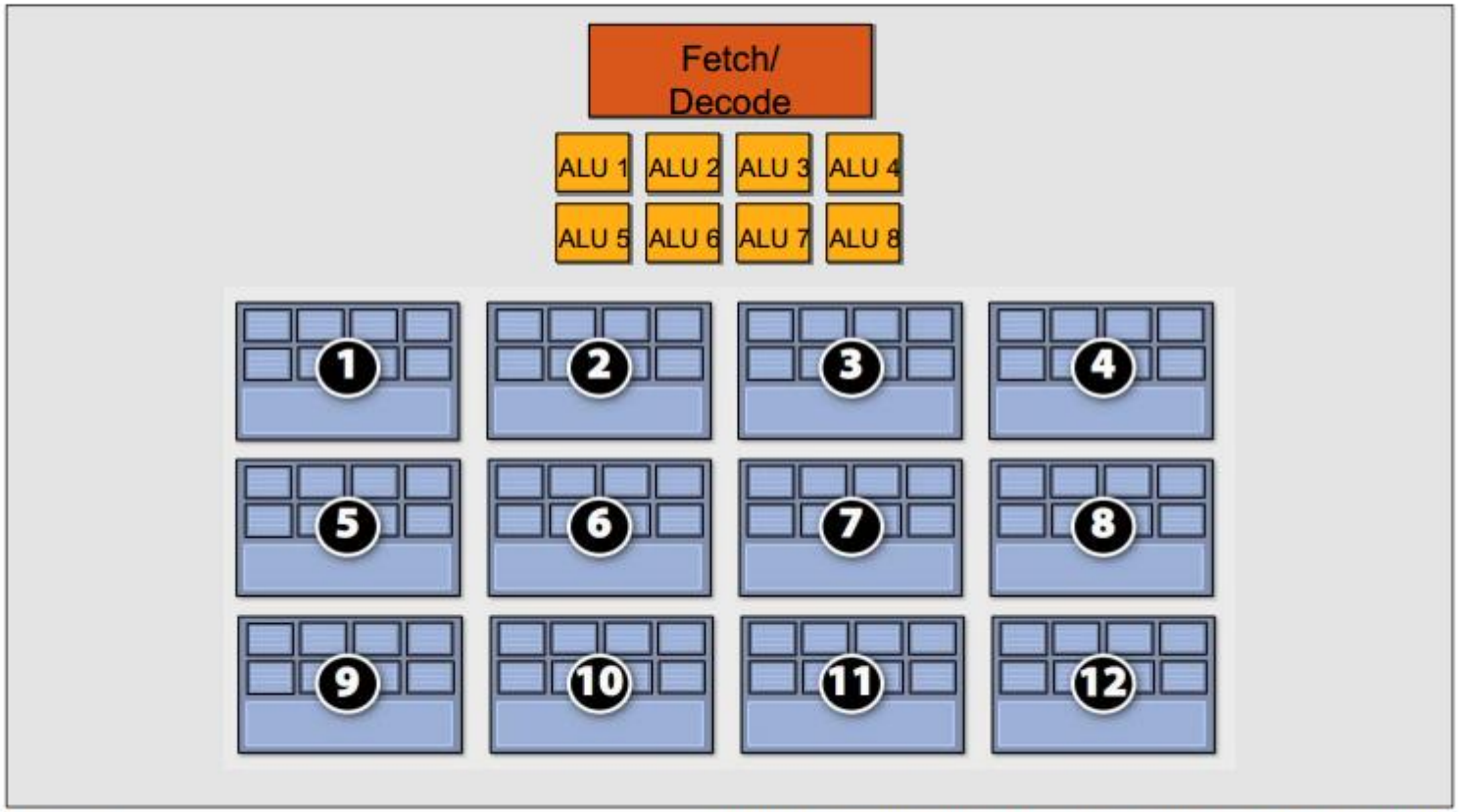
Storing contexts



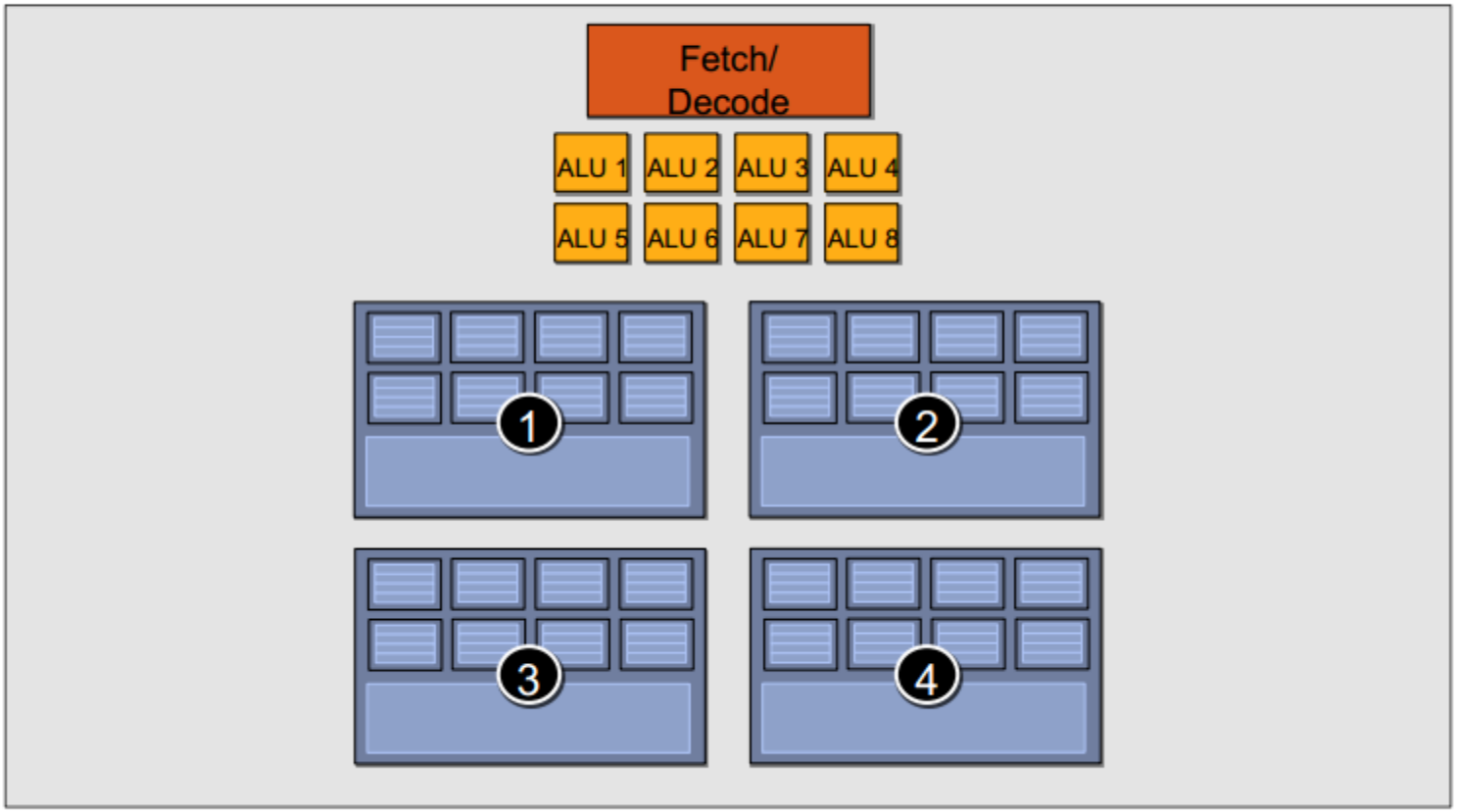
Eighteen small contexts (maximal latency hiding)



Twelve medium contexts



Four large contexts (low latency hiding ability)



Clarification

Interleaving between contexts can be managed by hardware or software (or both!)

- NVIDIA / ATI Radeon GPUs
  - HW schedules / manages all contexts (lots of them)
  - Special on-chip storage holds fragment state
- Intel Larrabee
  - HW manages four x86 (big) contexts at fine granularity
  - SW scheduling interleaves many groups of fragments on each HW context
  - L1-L2 cache holds fragment state (as determined by SW)

Example chip

16 cores

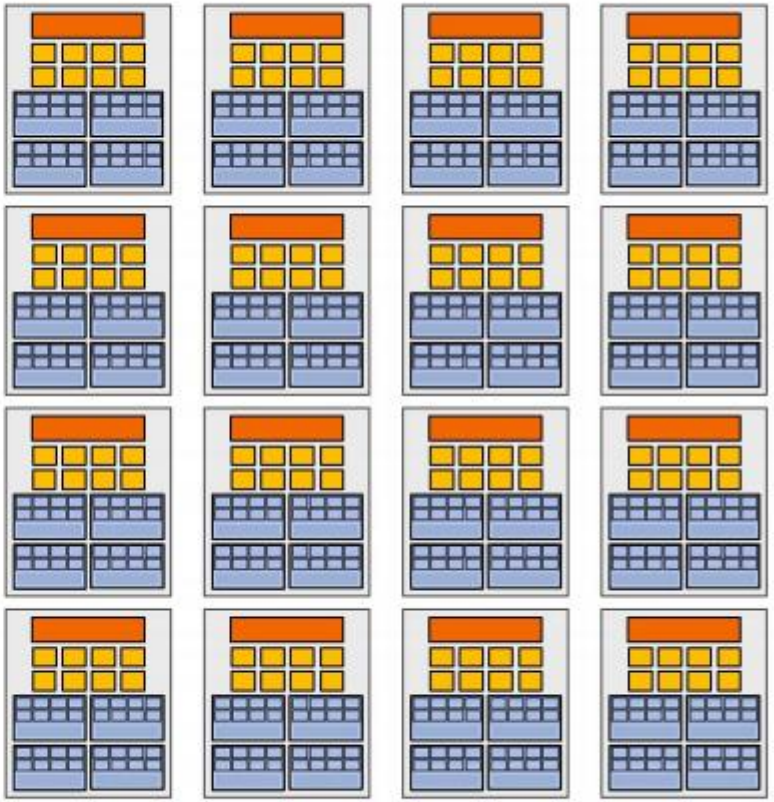
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



Summary: three key ideas

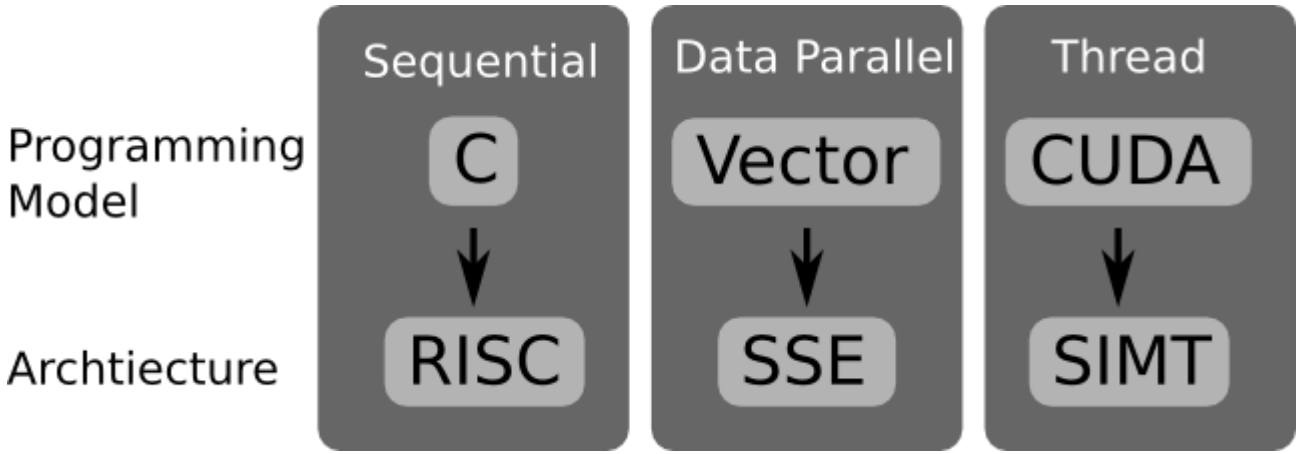
1. Use many “slimmed down cores” to run in parallel

2. Pack cores full of ALUs (by sharing instruction stream across groups of fragments)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by hardware

3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group

GP-GPU – More in depth!

Let's Start with Examples



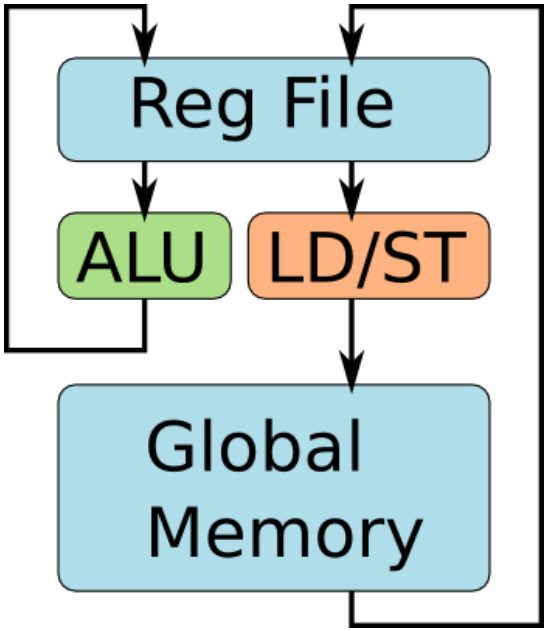
Don't worry, we will start from C and RISC!

Let's Start with C and RISC

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

Assembly code of inner-loop

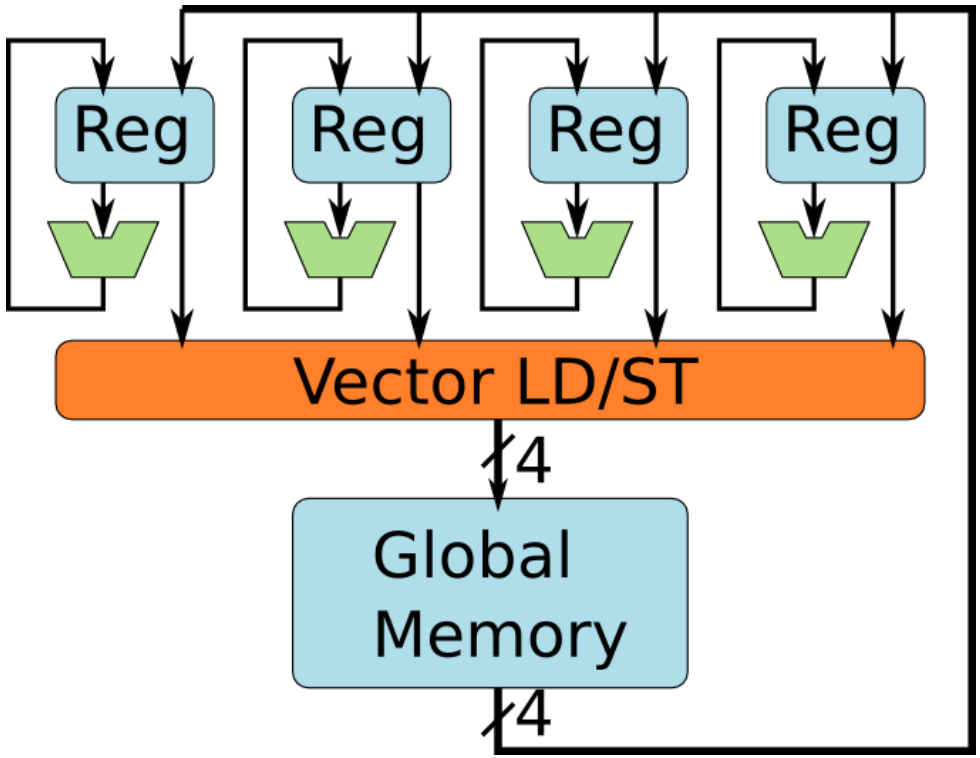
```
lw r0, 4(r1)
addi r0, r0, 1
sw r0, 4(r1)
```



Programmer's view of RISC

Most CPUs Have Vector SIMD Units

Programmer's view of a vector SIMD



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 instructions execute on 4 ALUs.

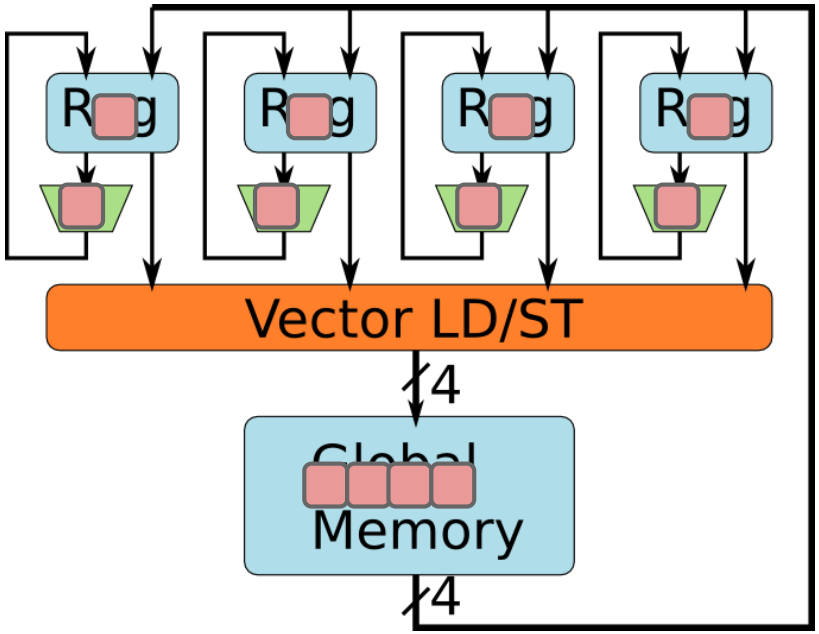
```
int A[2][4];
for(i=0;i<2;i++){
    for(j=0;j<4;j++){
        A[i][j]++;
    }
}
```

Assembly code of inner-loop

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

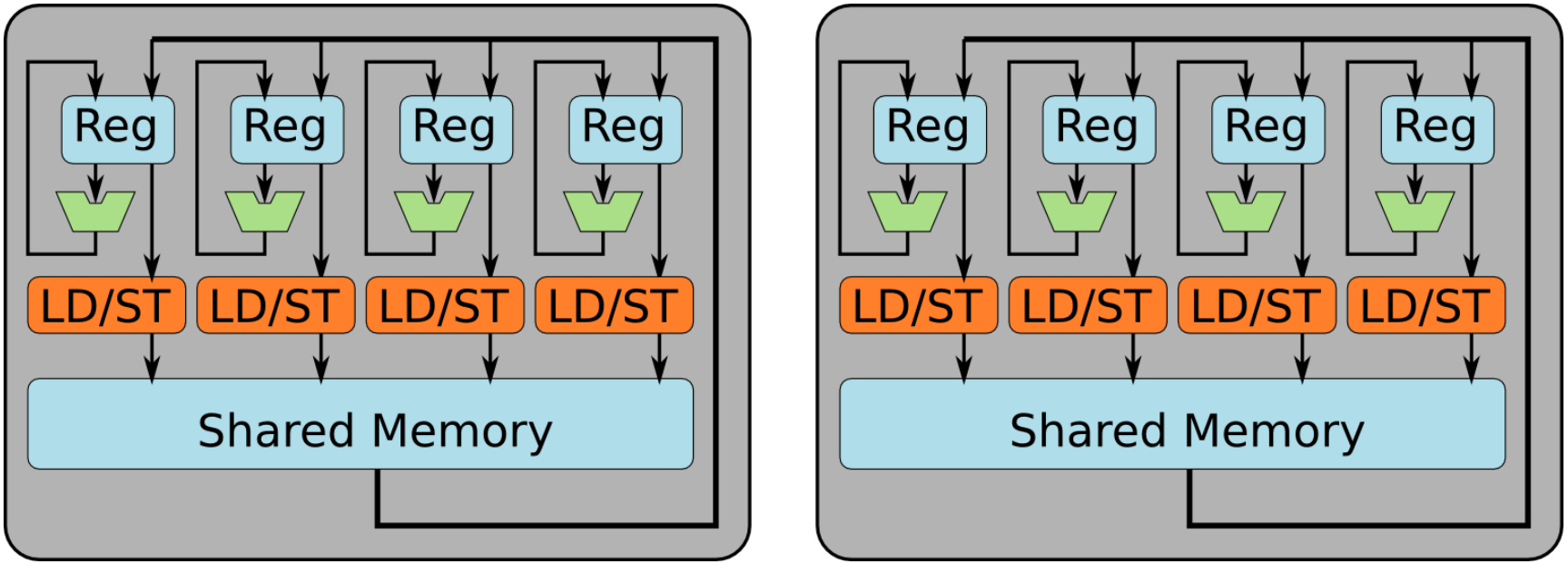
How Do Vector Programs Run?

```
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
}
```



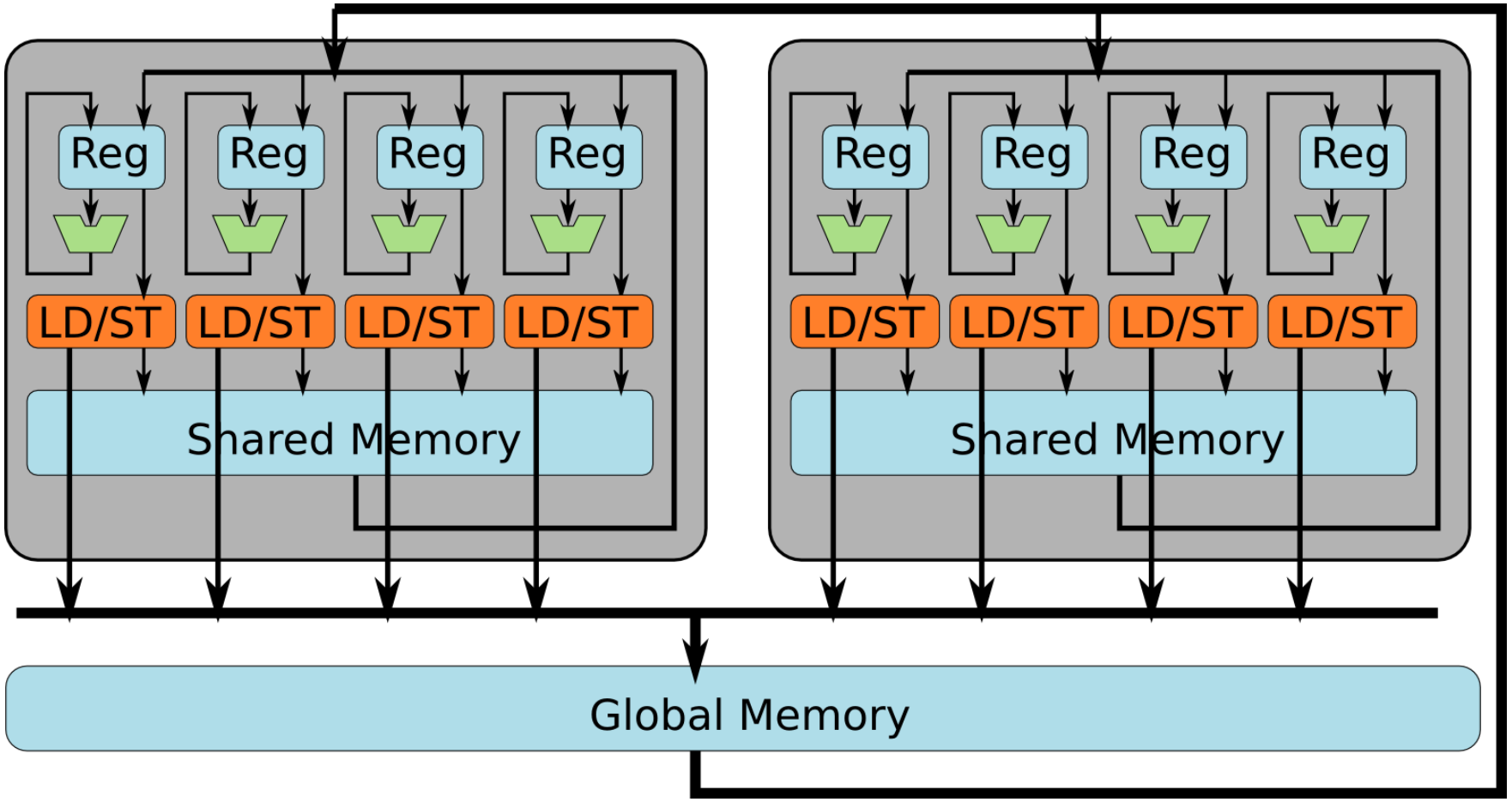
CUDA Programmer's View of GPUs

A GPU contains multiple SIMD Units.

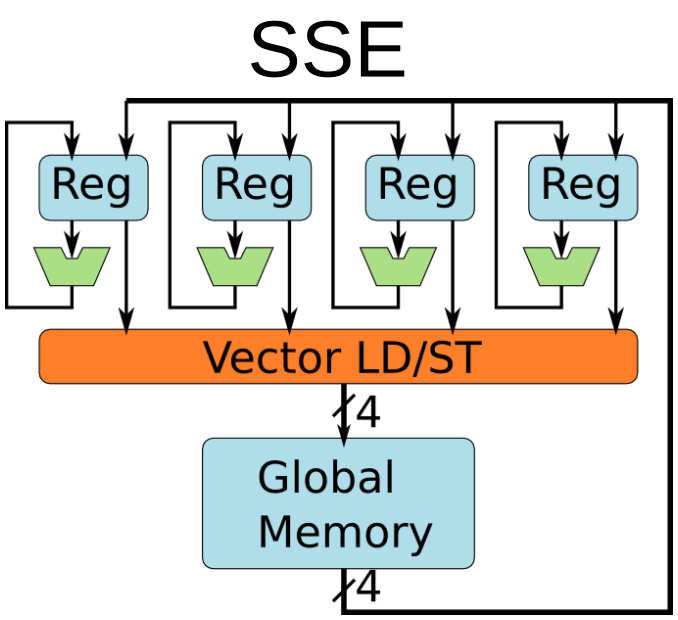


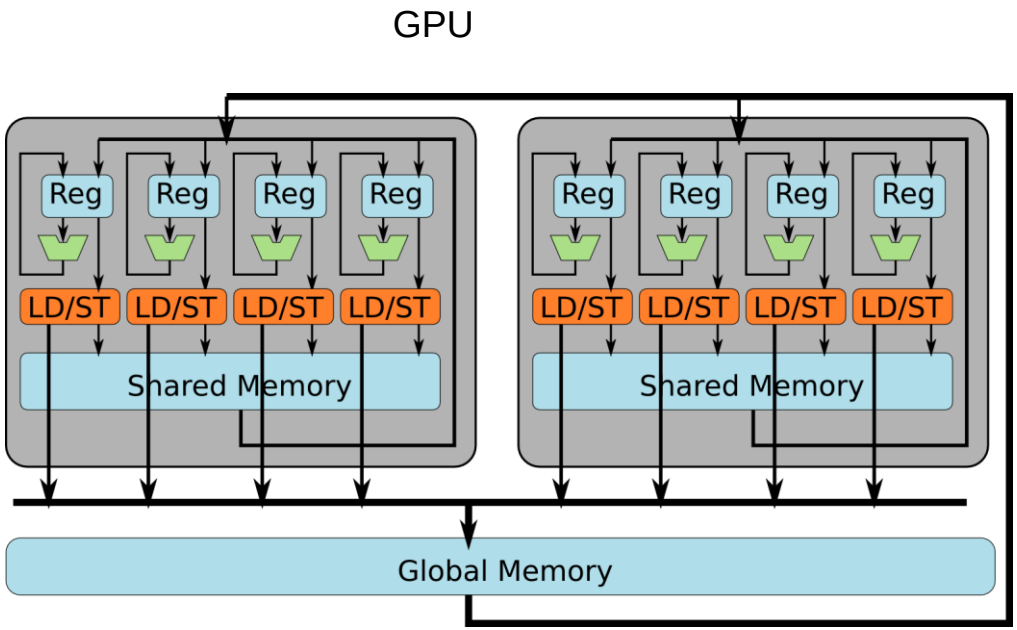
CUDA Programmer's View of GPUs

A GPU contains multiple SIMD Units. All of them can access global memory.



What Are the Differences?





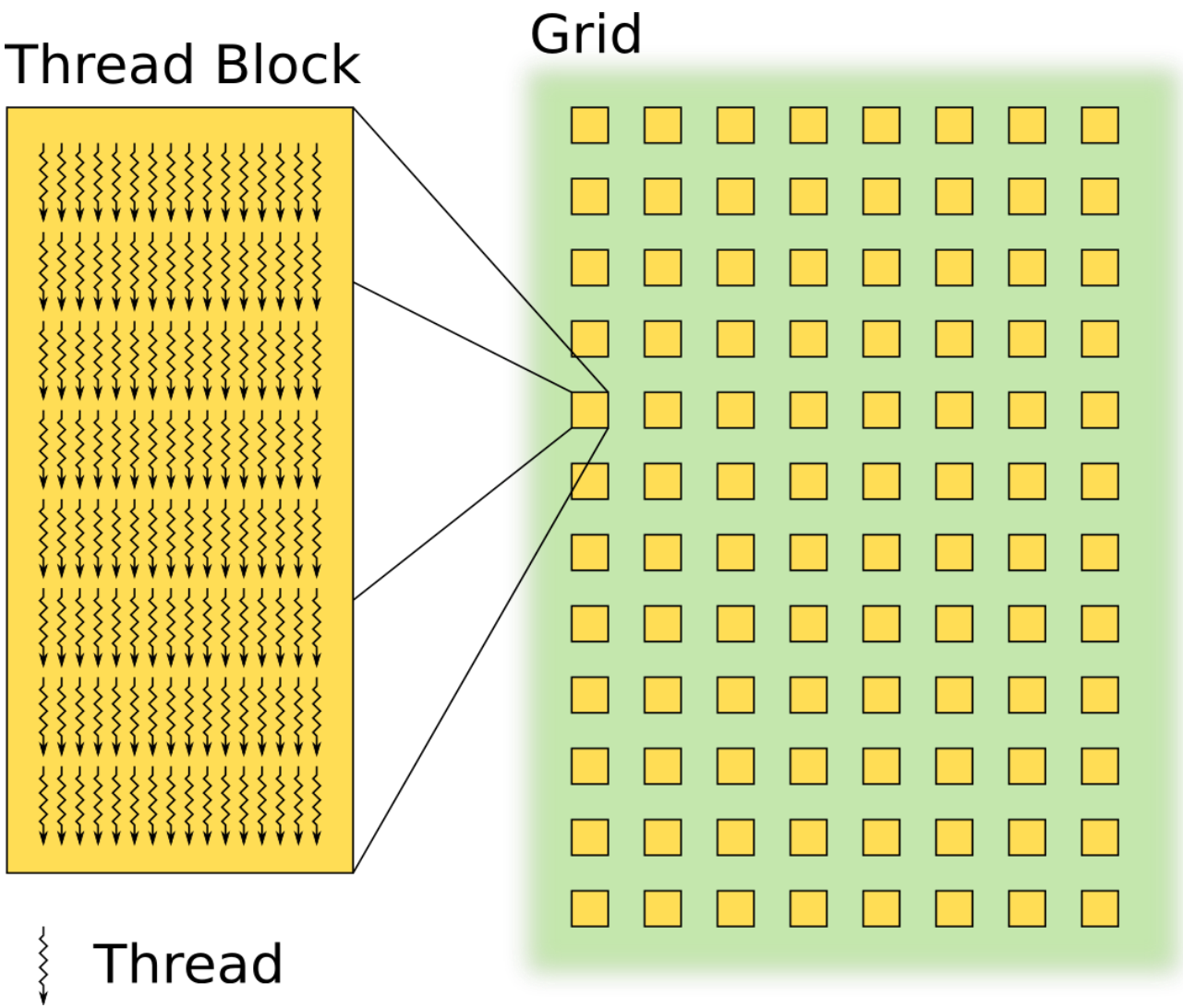
Let's start with two important differences:

1. GPUs use threads instead of vectors
2. GPUs have the "Shared Memory" spaces

Thread Hierarchy in CUDA

Grid
contains
Thread Blocks

Thread Block
contains
Threads



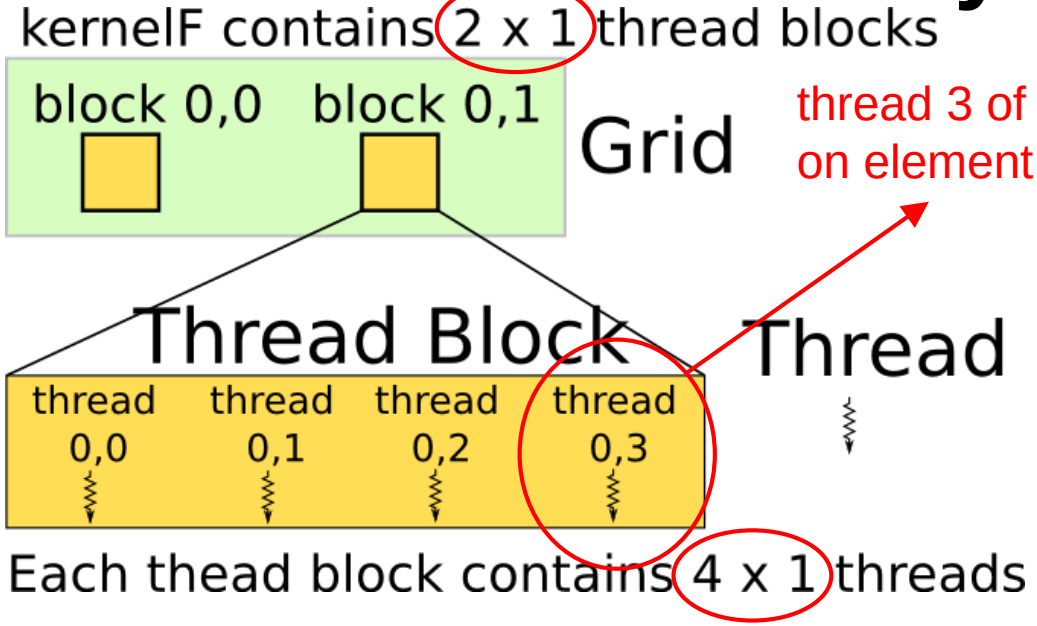
Let's Start Again from C

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

convert into CUDA

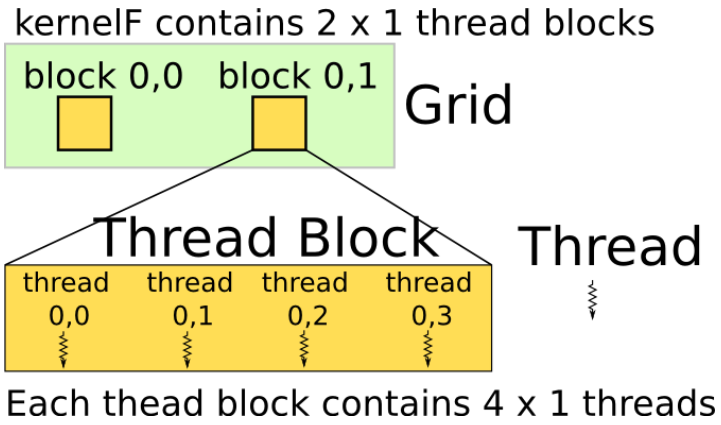
```
int A[2][4];
kernelF<<<(2,1),(4,1)>>>(A); // define threads
__device__    kernelF(A){  // all threads run same kernel
    i = blockIdx.x;   // each thread block has its id
    j = threadIdx.x;  // each thread has its id
    A[i][j]++;   // each thread has a different i and j
}
```

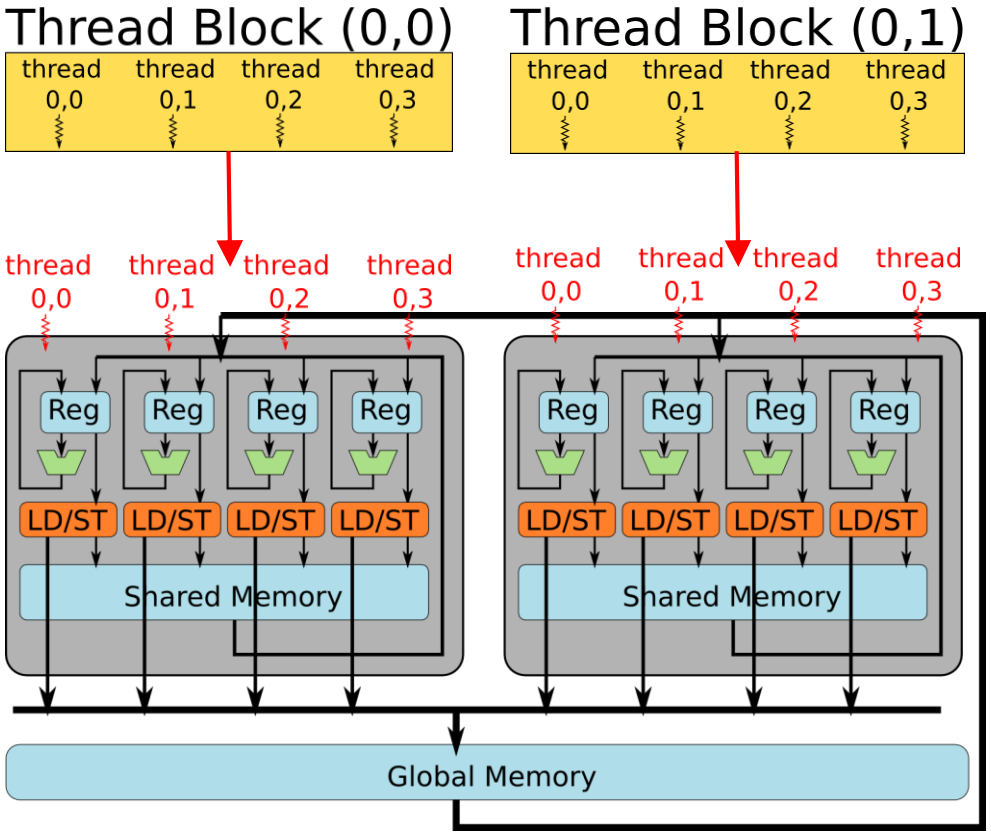
What Is the Thread Hierarchy?



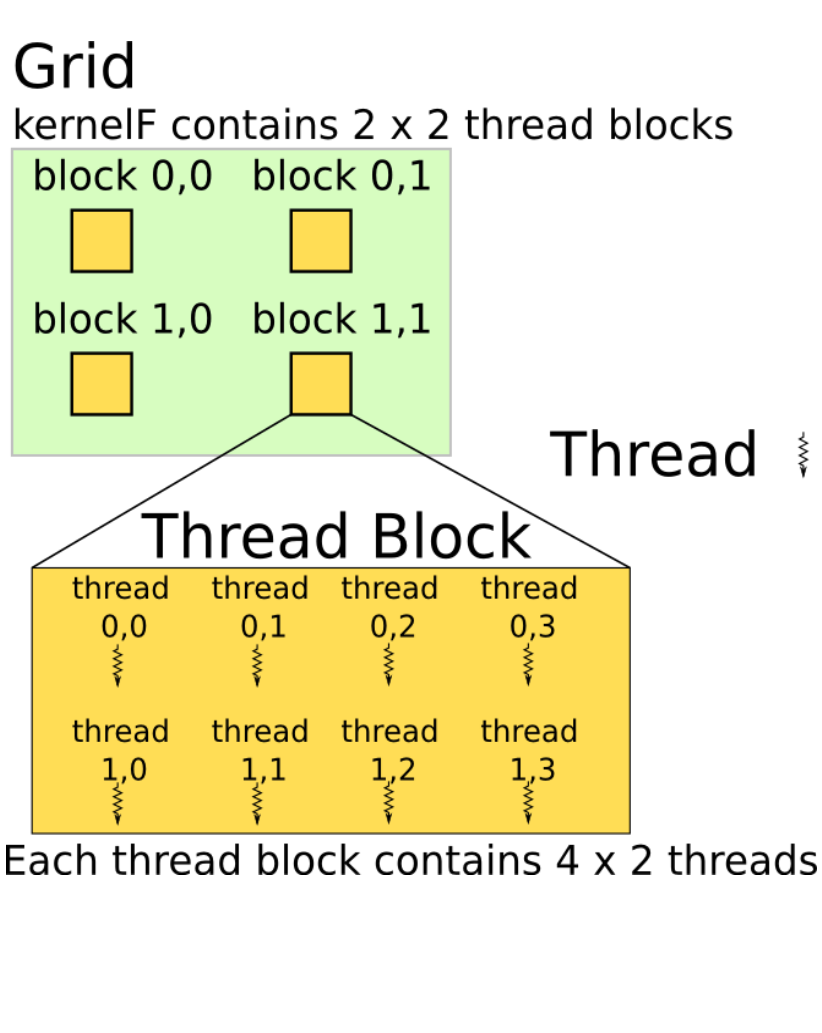

```
int A[2][4];
kernelF<<<(2,1),(4,1)>>>(A); // define threads
__device__ kernelF(A){ // all threads run same kernel
    i = blockIdx.x; // each thread block has its id
    j = threadIdx.x; // each thread has its id
    A[i][j]++; // each thread has a different i and j
}
```

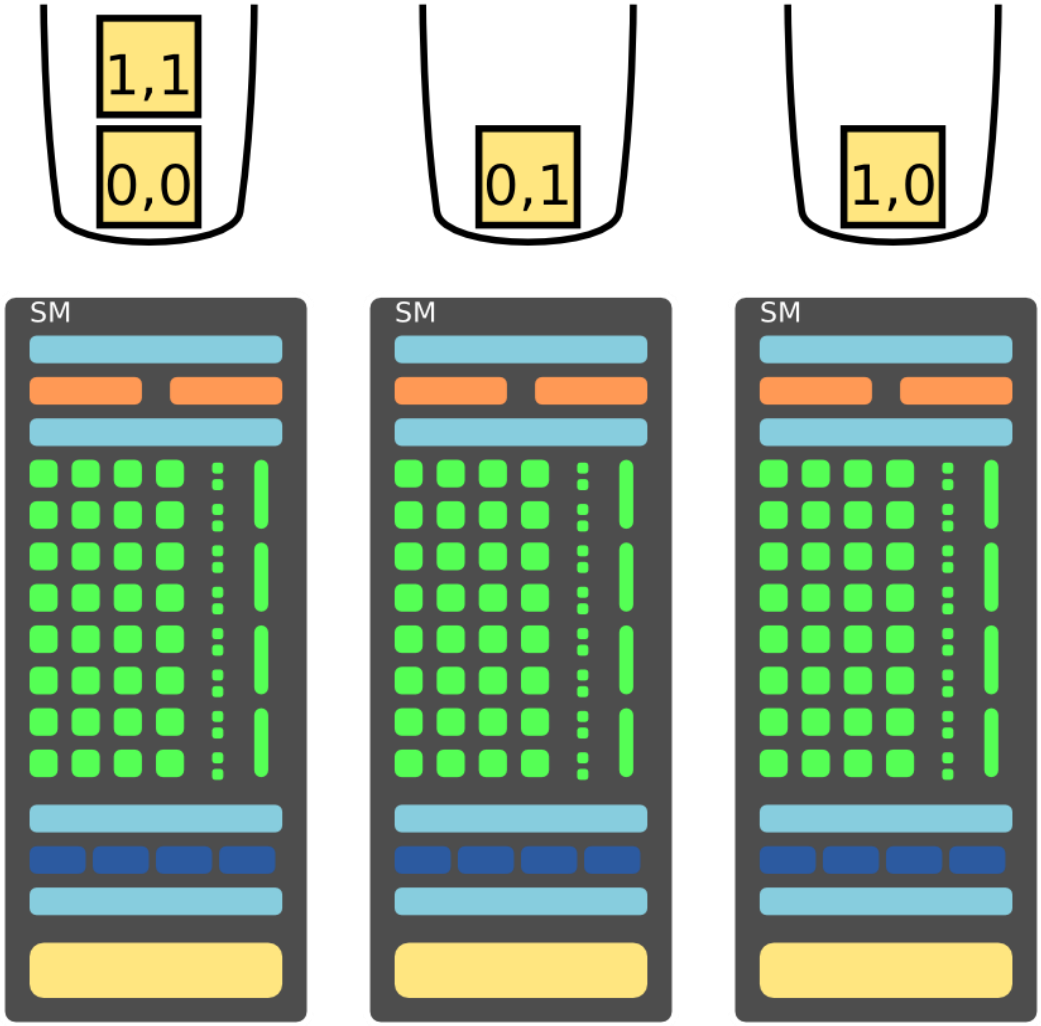
How Are Threads Scheduled?





Blocks Are Dynamically Scheduled



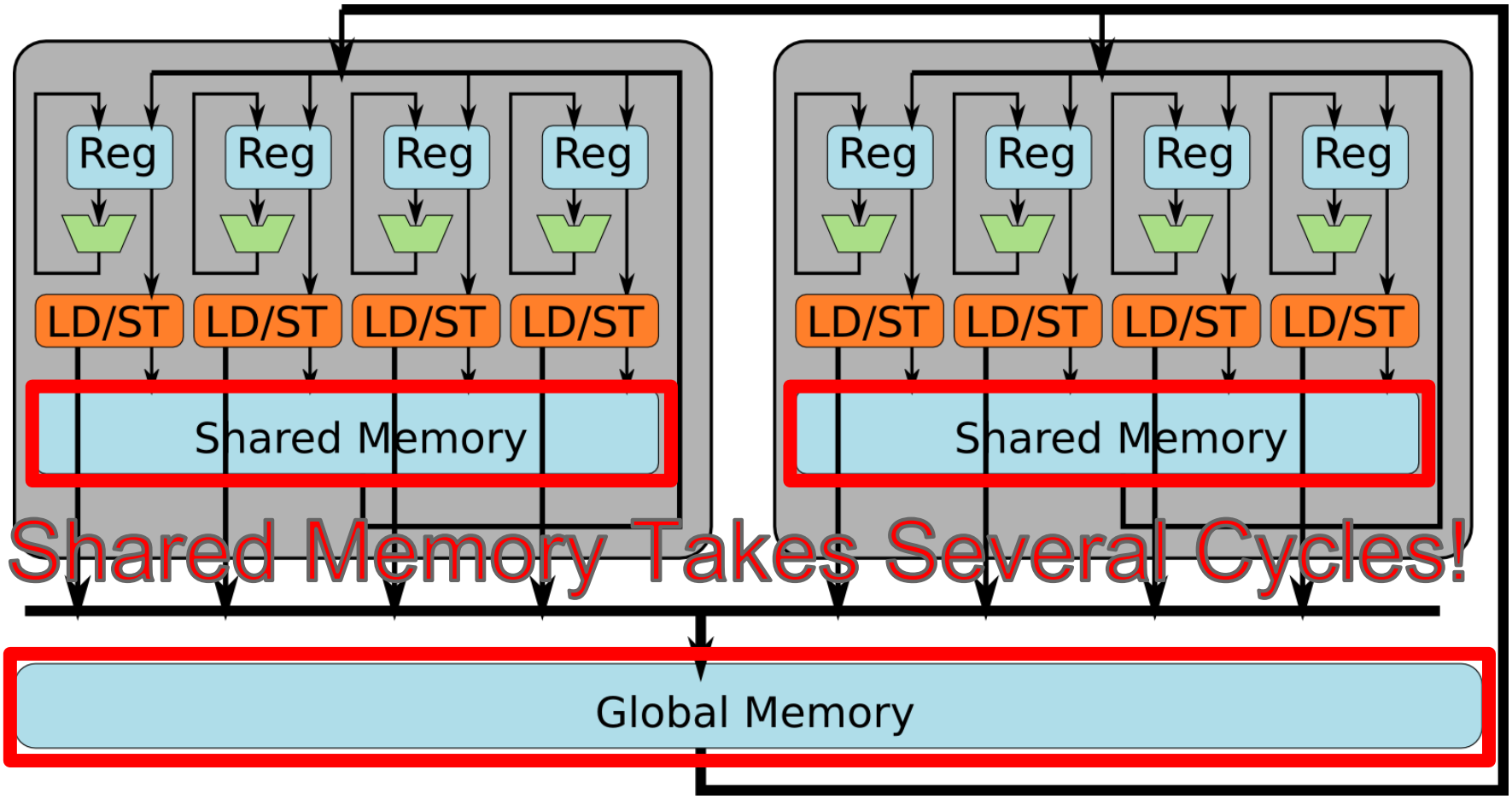


How Are Threads Executed?

```
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]
add.s32 %r4, %r4, 1           // r4 = r4 + 1
st.global.s32 [%r3], %r4      // A[i][j] = r4
```

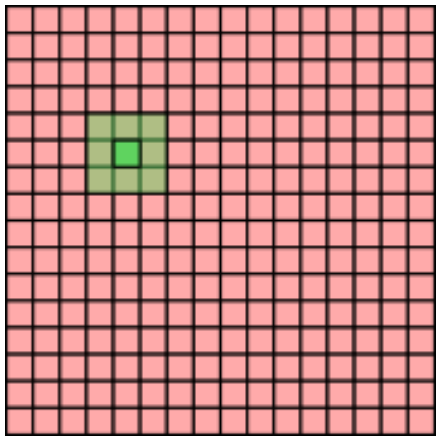
Utilizing Memory Hierarchy



Global memory access takes > 100 cycles.

Example: Average Filters

Average over a 3x3 window for a 16x16 array

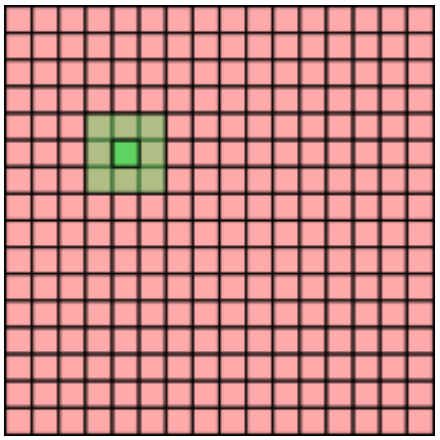


```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    i = threadIdx.y;
    j = threadIdx.x;
    tmp = (A[i-1][j-1]
                + A[i-1][j]
                ...
                + A[i+1][i+1] ) / 9;
    A[i][j] = tmp;
}
```

Each thread loads 9 elements from global memory.
It takes > 900 cycles!

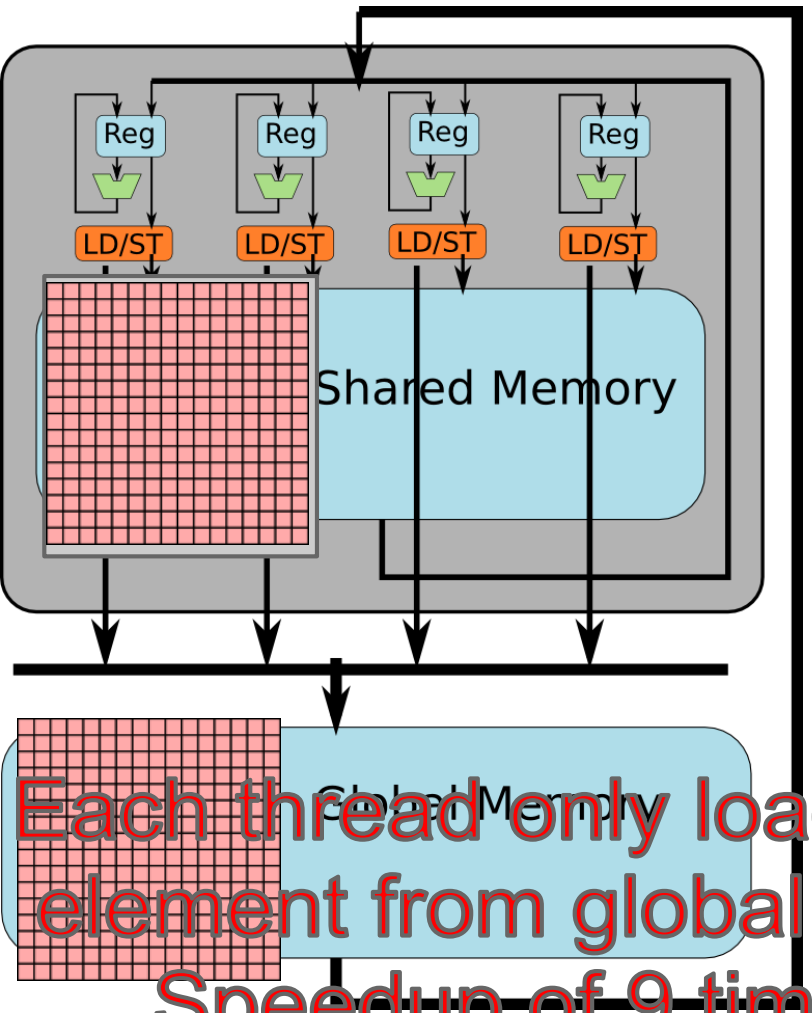
Utilizing the Shared Memory

Average over a 3x3 window for a 16x16 array



```
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][i+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][i+1] ) / 9;
}
```

allocate shared mem

Each thread only loads one element from global mem.

Speedup of 9 times!

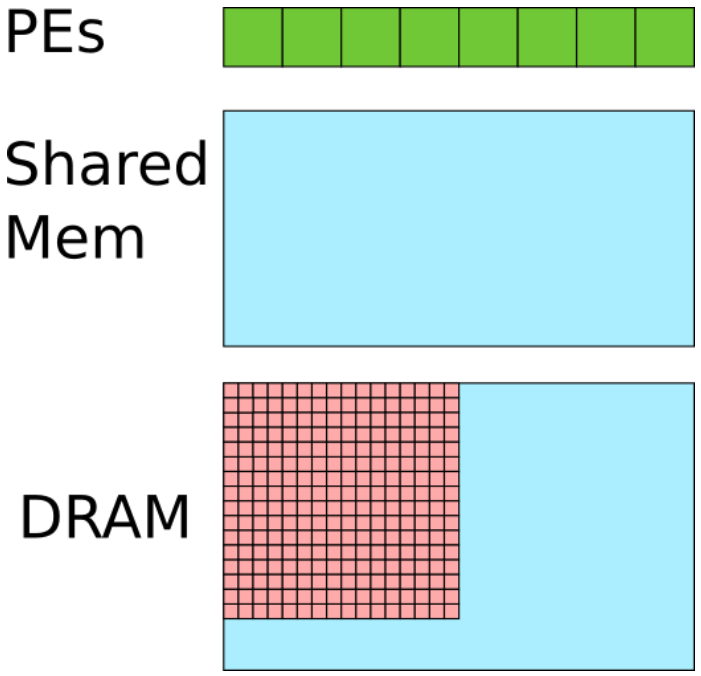
However, the Program Is Incorrect

```
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][i+1] ) / 9;
}
```

Hazards!

Let's See What's Wrong

Assume 256 threads are scheduled on 8 PEs.

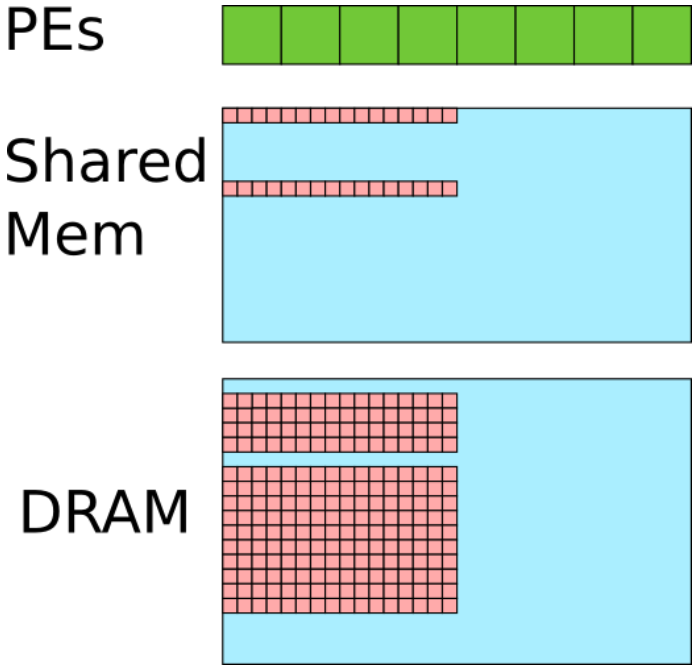



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1]
                + smem[i-1][j]
                ...
                + smem[i+1][i+1] ) / 9;
}
```

Before load instruction

Let's See What's Wrong

Assume 256 threads are scheduled on 8 PEs.



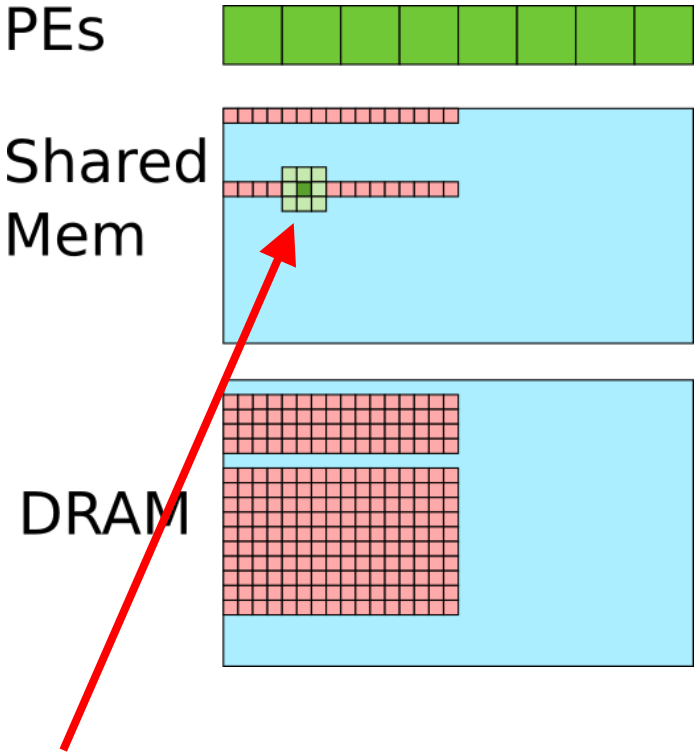

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1]
              + smem[i-1][j]
              ...
              + smem[i+1][i+1] ) / 9;
}
```

Some threads finish the load earlier than others.

Threads starts window operation as soon as it loads it own data element.

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!

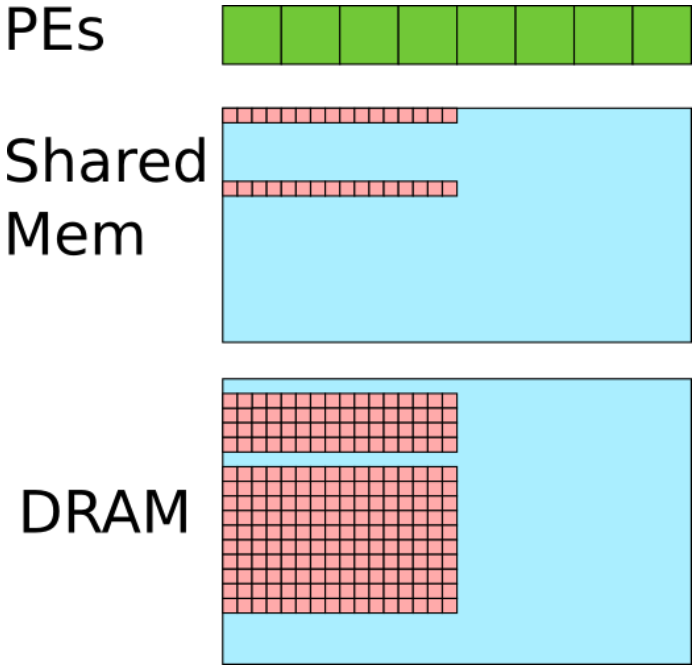
```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
smem[i][j] = A[i][j]; // load to smem
A[i][j] = ( smem[i-1][j-1]
          + smem[i-1][j]
          ...
          + smem[i+1][i+1] ) / 9;
}
```

Some threads finish the load earlier than others.

Threads starts window operation as soon as it loads it own data element.

How To Solve It?

Assume 256 threads are scheduled on 8 PEs.

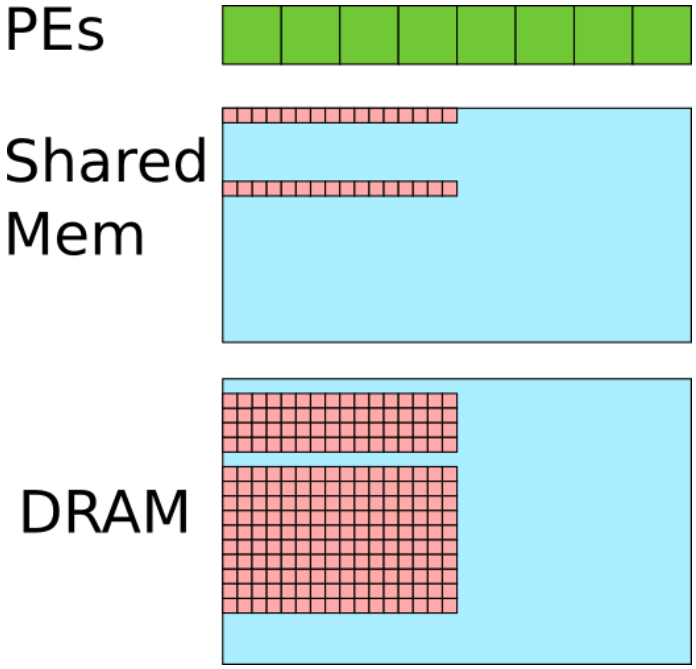



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1]
                + smem[i-1][j]
                ...
                + smem[i+1][i+1] ) / 9;
}
```

Some threads finish the load earlier than others.

Use a "SYNC" barrier!

Assume 256 threads are scheduled on 8 PEs.

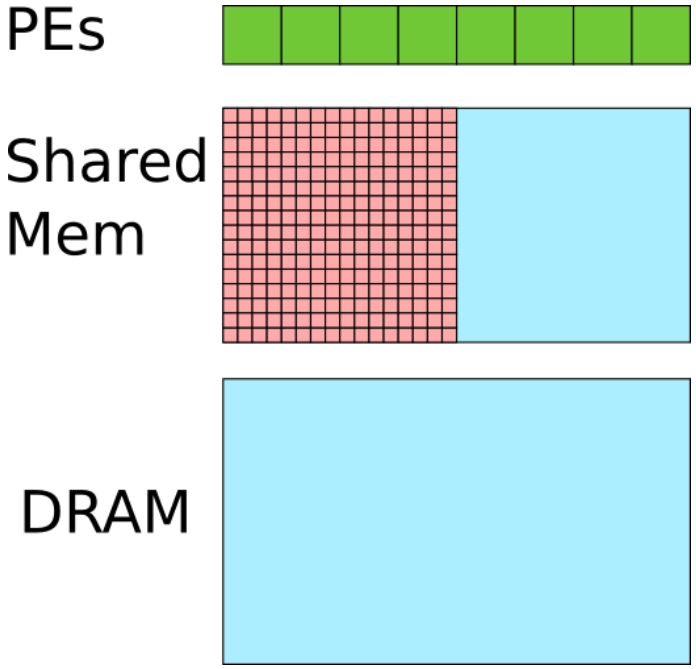



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __sync(); // threads wait at barrier
    A[i][j] = ( smem[i-1][j-1]
                + smem[i-1][j]
                ...
   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

Some threads finish the load earlier than others.

Use a "SYNC" barrier!

Assume 256 threads are scheduled on 8 PEs.



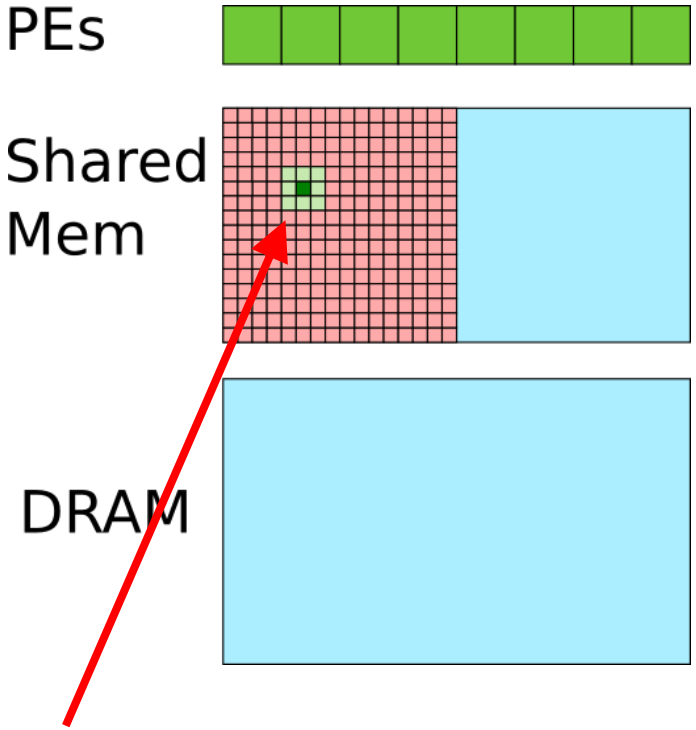

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __sync(); // threads wait at barrier
    A[i][j] = ( smem[i-1][j-1]
              + smem[i-1][j]
              ...
       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

Some threads finish the load earlier than others.

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(); // threads wait at barrier
    A[i][j] = ( smem[i-1][j-1]
                + smem[i-1][j]
                ...
                + smem[i+1][i+1] ) / 9;
}
```

Review What We Have Learned

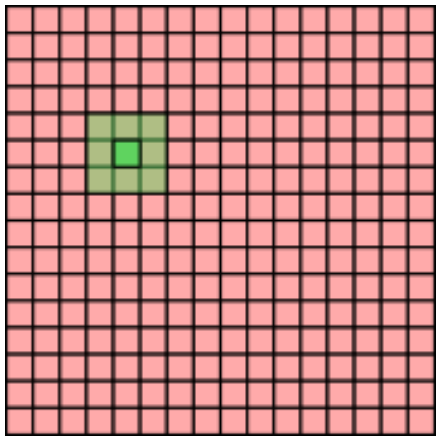
1. Single Instruction Multiple Thread (SIMT)
2. Shared memory

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

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

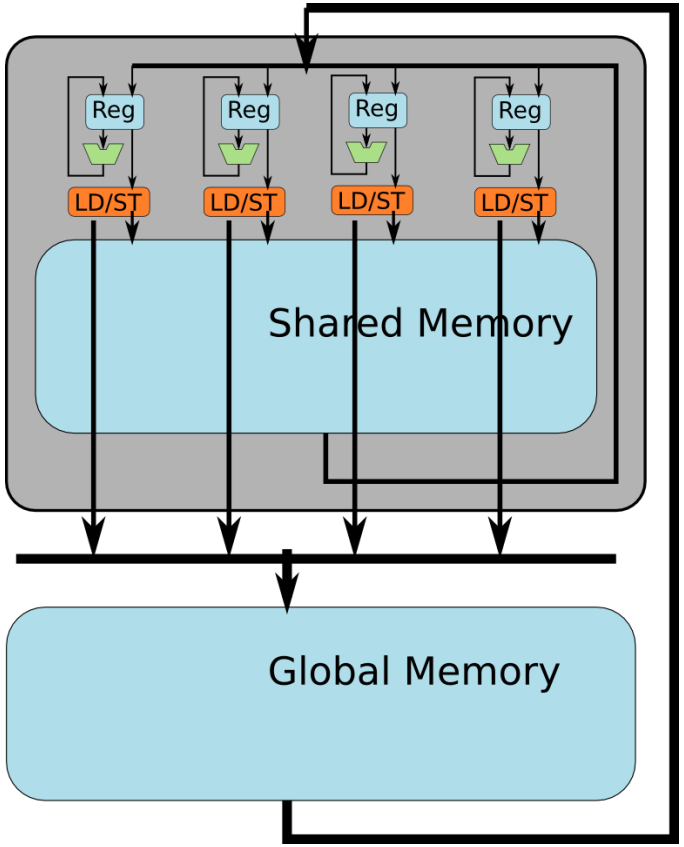
Take the Same Example Again

Average over a 3x3 window for a 16x16 array



Assume vector SIMD and SIMT both have shared memory.
What is the difference?

?



Vector SIMD v.s. SIMT

```
int A[16][16]; // global memory
__shared__ int B[16][16]; // shared mem
for(i=0;i<16;i++){
   for(j=0;i<4;j+=4){
      movups xmm0, [ &A[i][j] ]
      movups [ &B[i][j] ], xmm0 }}
for(i=0;i<16;i++){
   for(j=0;i<4;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;i<4;j+=4){
      addps [ &A[i][j] ], xmm1 }}
```

```
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;
}
```

Allocate smem

Global -> smem

Exec.

Global <- smem

Vector SIMD v.s. SIMT

```
int A[16][16];
__shared__ int B[16][16];
for(i=0;i<16;i++){
   for(j=0;i<4;j+=4){
      movups xmm0, [ &A[i][j] ]
      movups [ &B[i][j] ], xmm0 }}
for(i=0;i<16;i++){
   for(j=0;i<4;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;i<4;j+=4){
      addps [ &A[i][j] ], xmm1 }}
```

Programmers need to know there are 4 PEs.

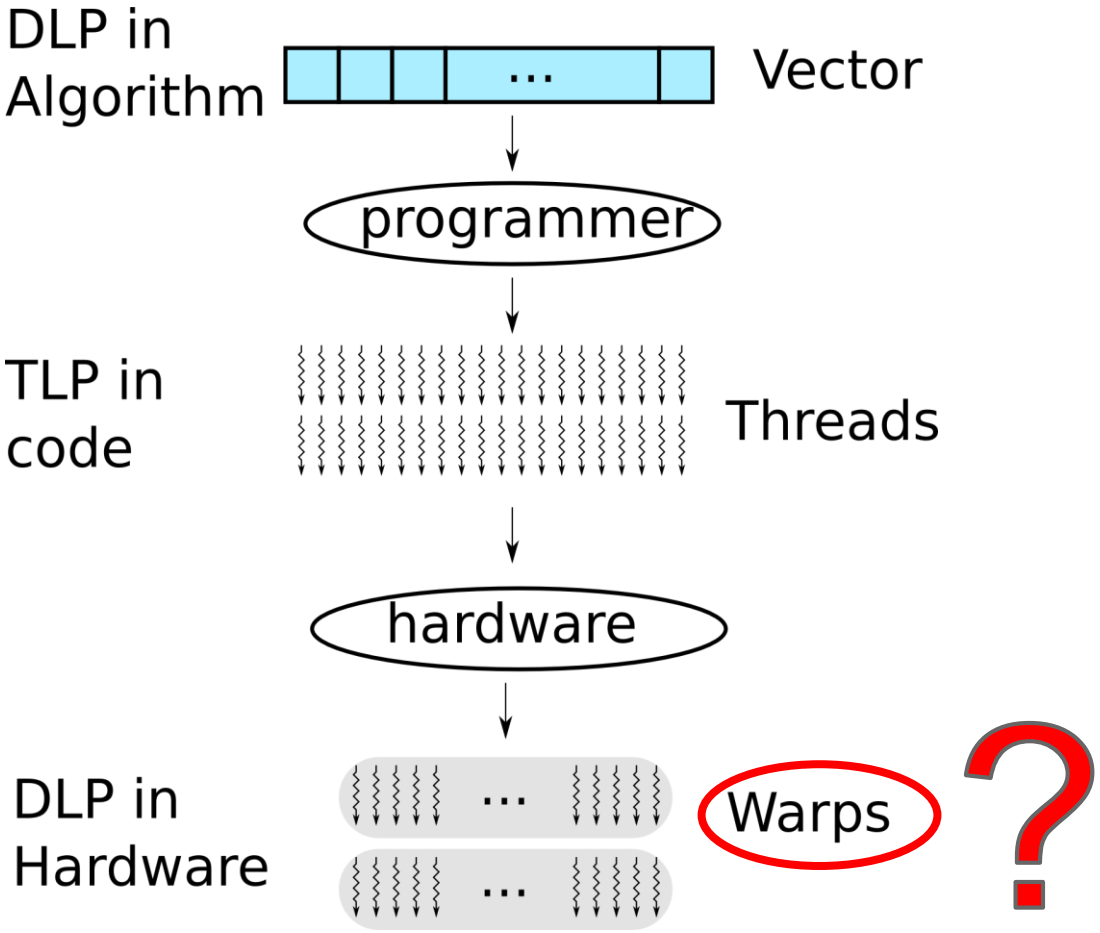
Each inst. is executed by all PEs in locked step.

```
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];
   __sync(); // threads wait at barrier
   A[i][j] = ( smem[i-1][j-1]
             + smem[i-1][j]
             ...
             + smem[i+1][i+1] ) / 9;
}
```

# of PEs in HW is transparent to programmers.

Programmers give up exec. ordering to HW.

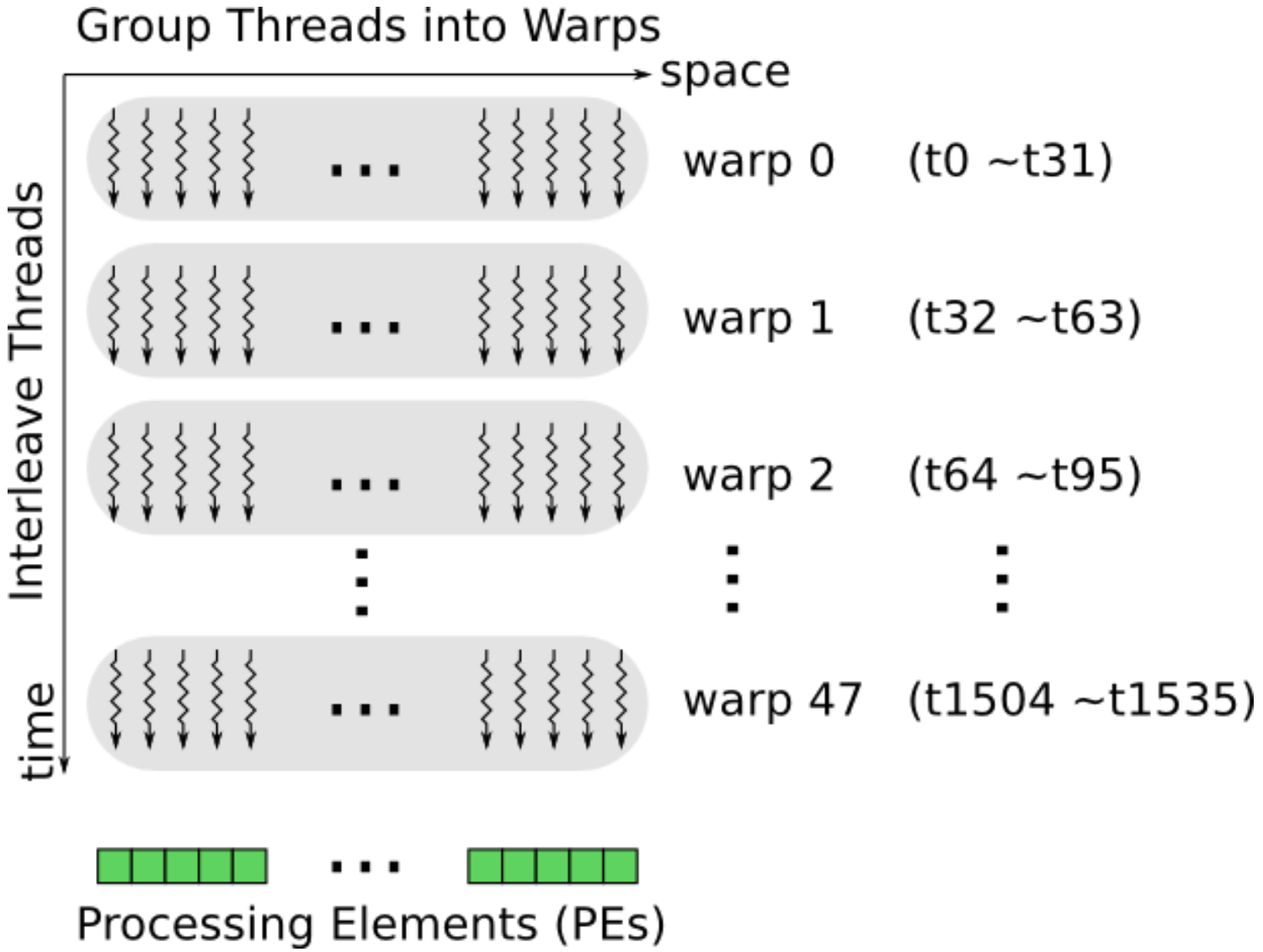
Review What We Have Learned



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

HW Groups Threads Into Warps

Example: 32 threads per warp



Why Make Things Complicated?

What if without these complication?

Remember the hazards in MIPS?

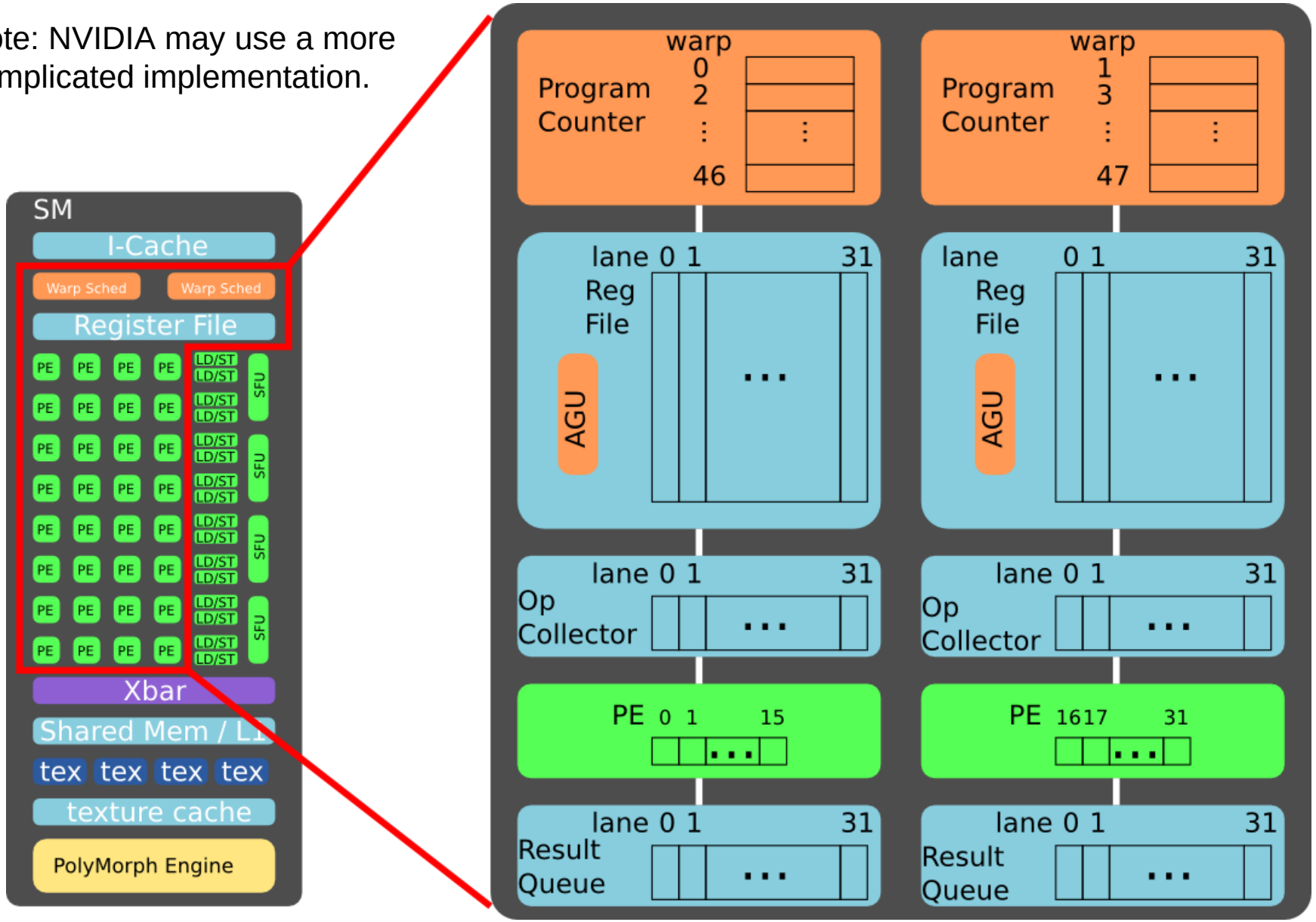
- Structural hazards
- Data hazards
- Control hazards

More About GPUs Work

Let's start with an example.

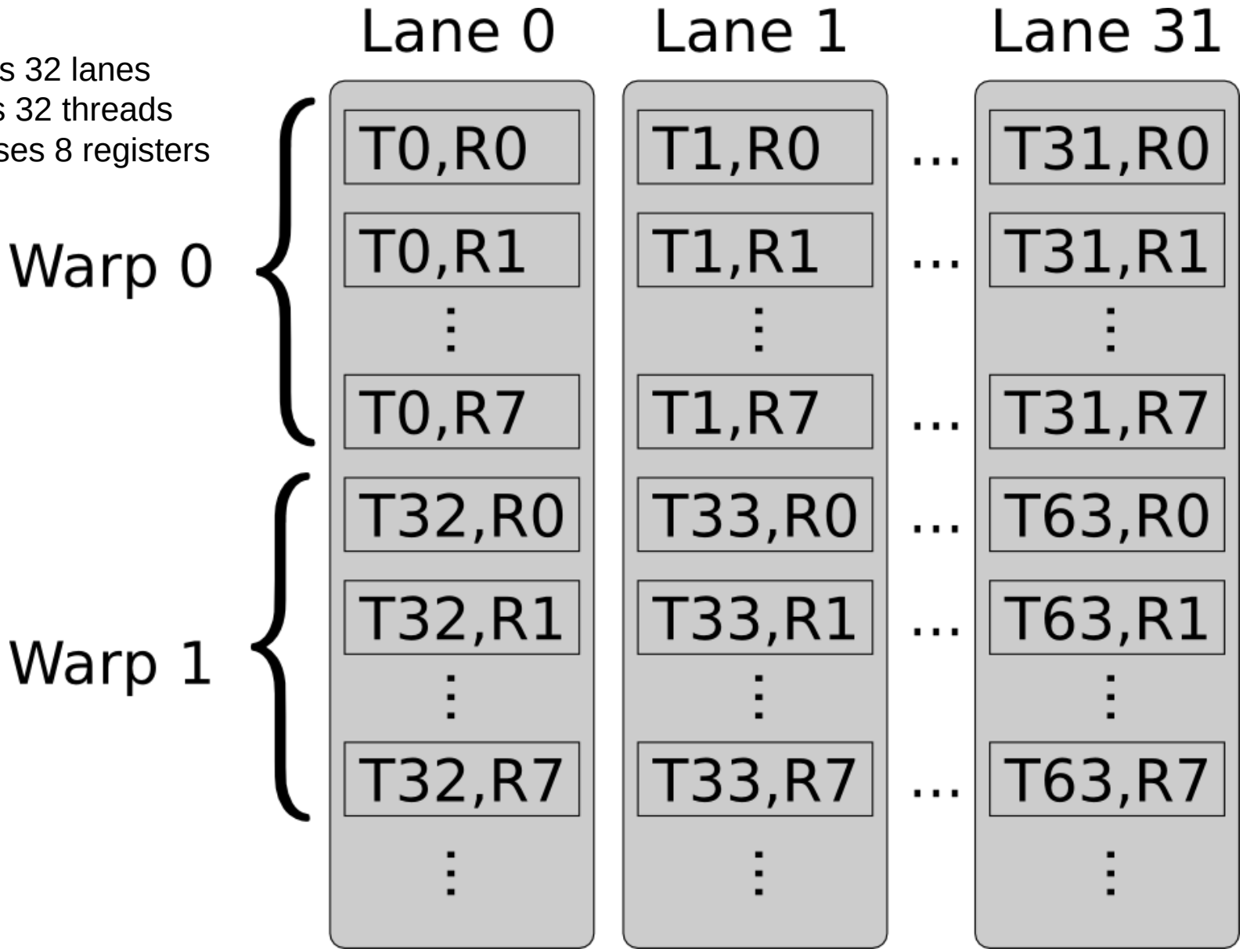
Example of Implementation

Note: NVIDIA may use a more complicated implementation.



Example of Register Allocation

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



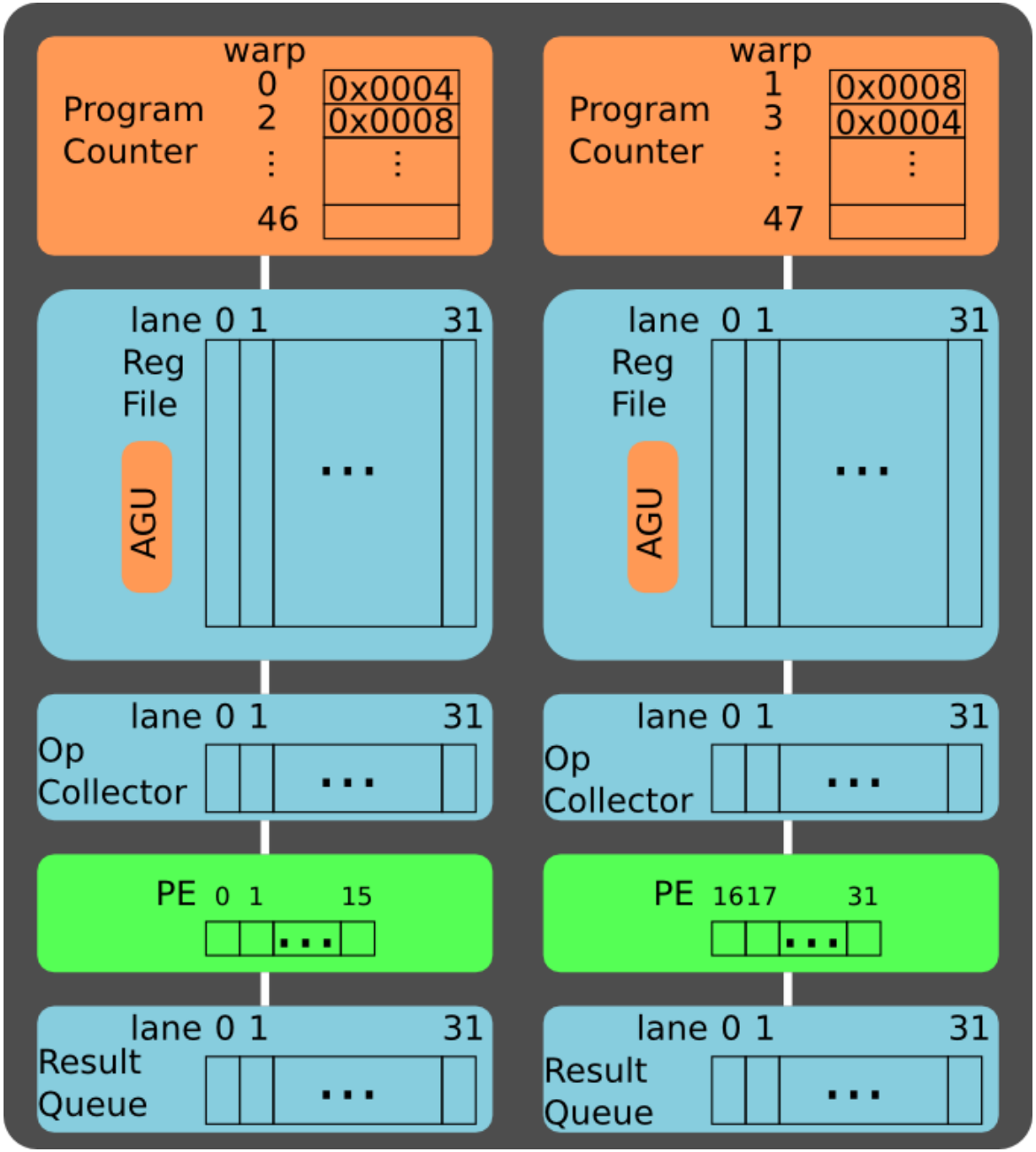
Example

Program Address: Inst

0x0004: add r0, r1, r2

0x0008: sub r3, r4, r5

Assume warp 0 and warp 1 are scheduled for execution.



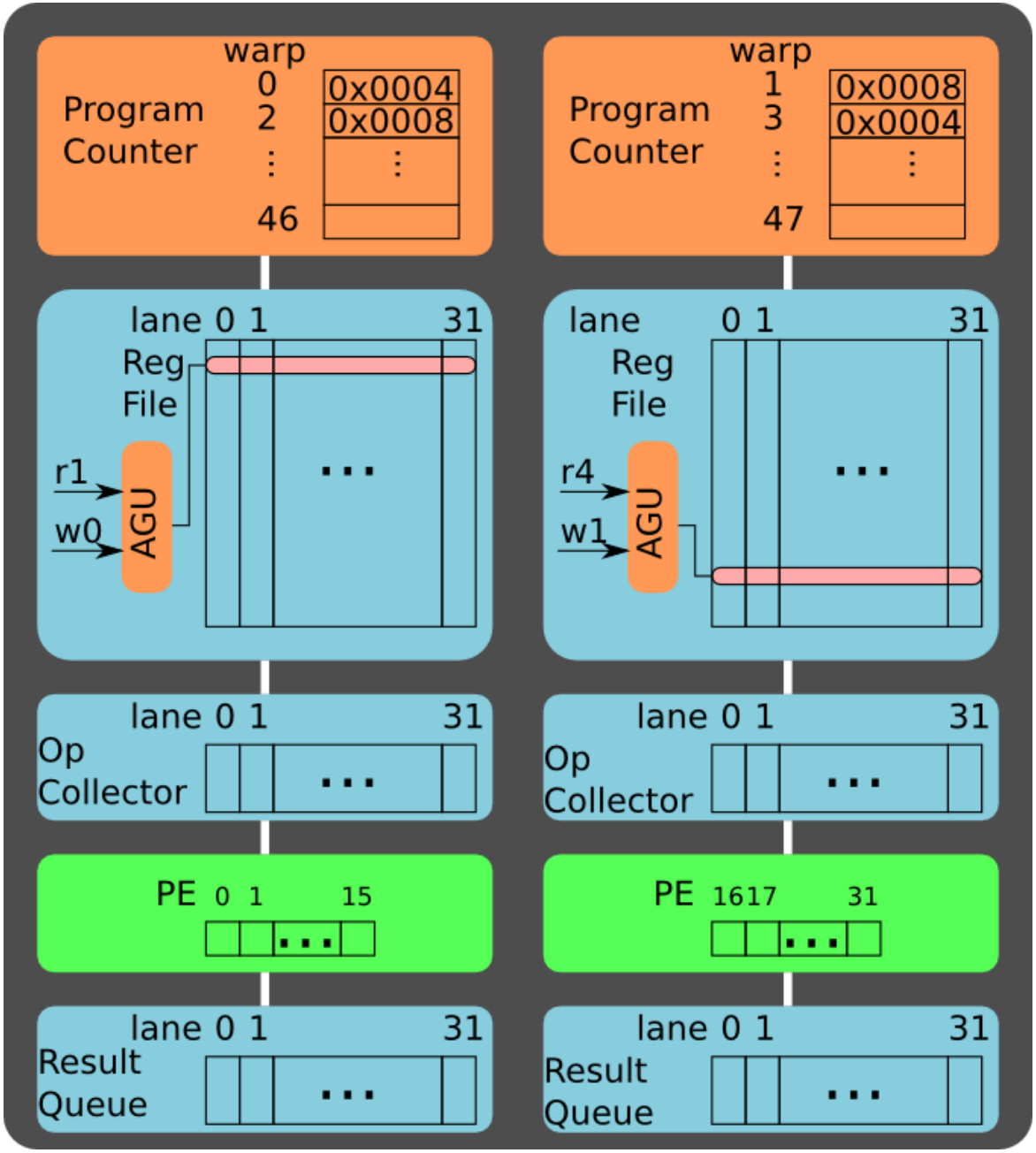
Read Src Op

Program Address: Inst

0x0004: add r0, r1, r2

0x0008: sub r3, r4, r5

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



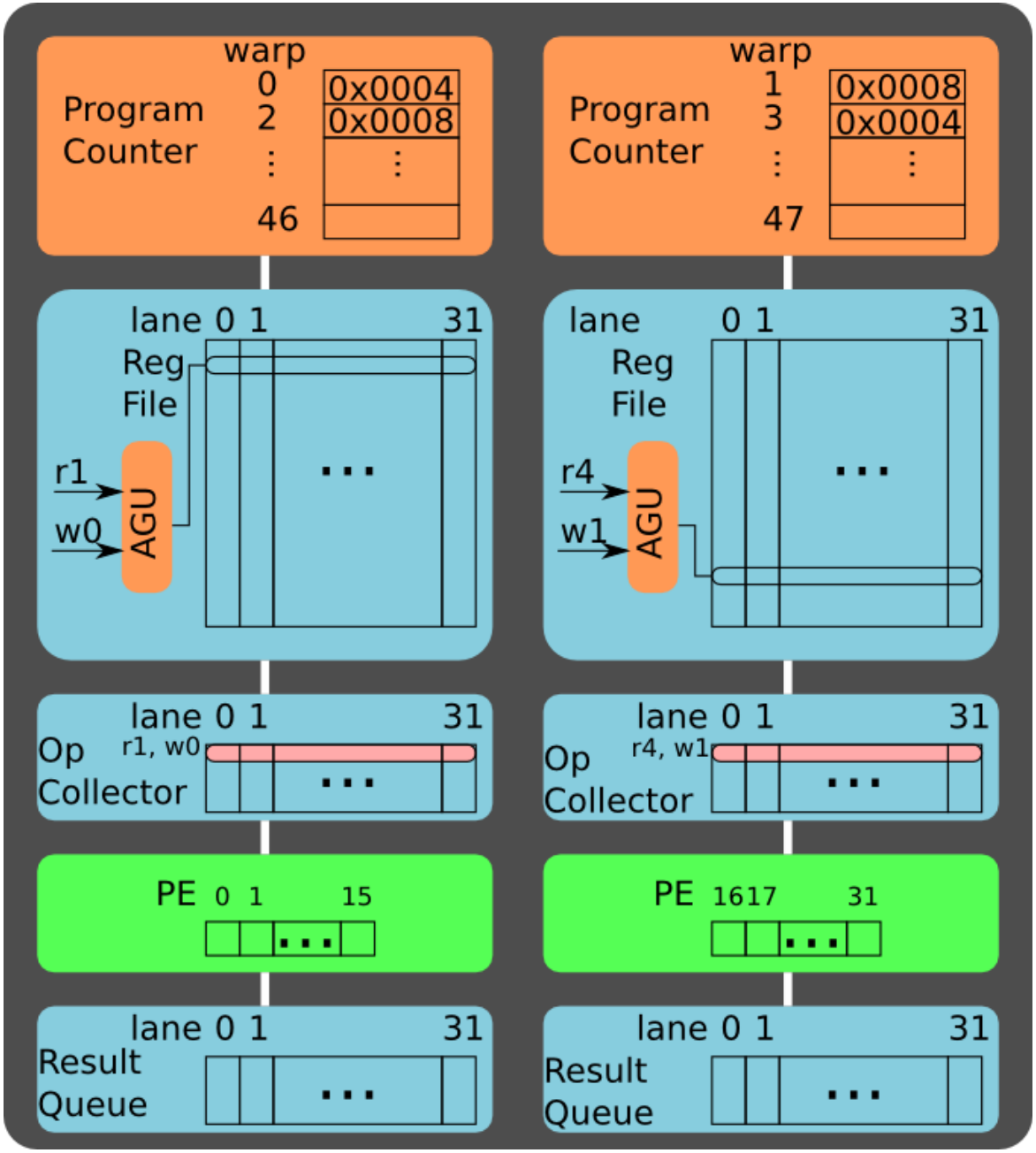
Buffer Src Op

Program Address: Inst

0x0004: add r0, r1, r2

0x0008: sub r3, r4, r5

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



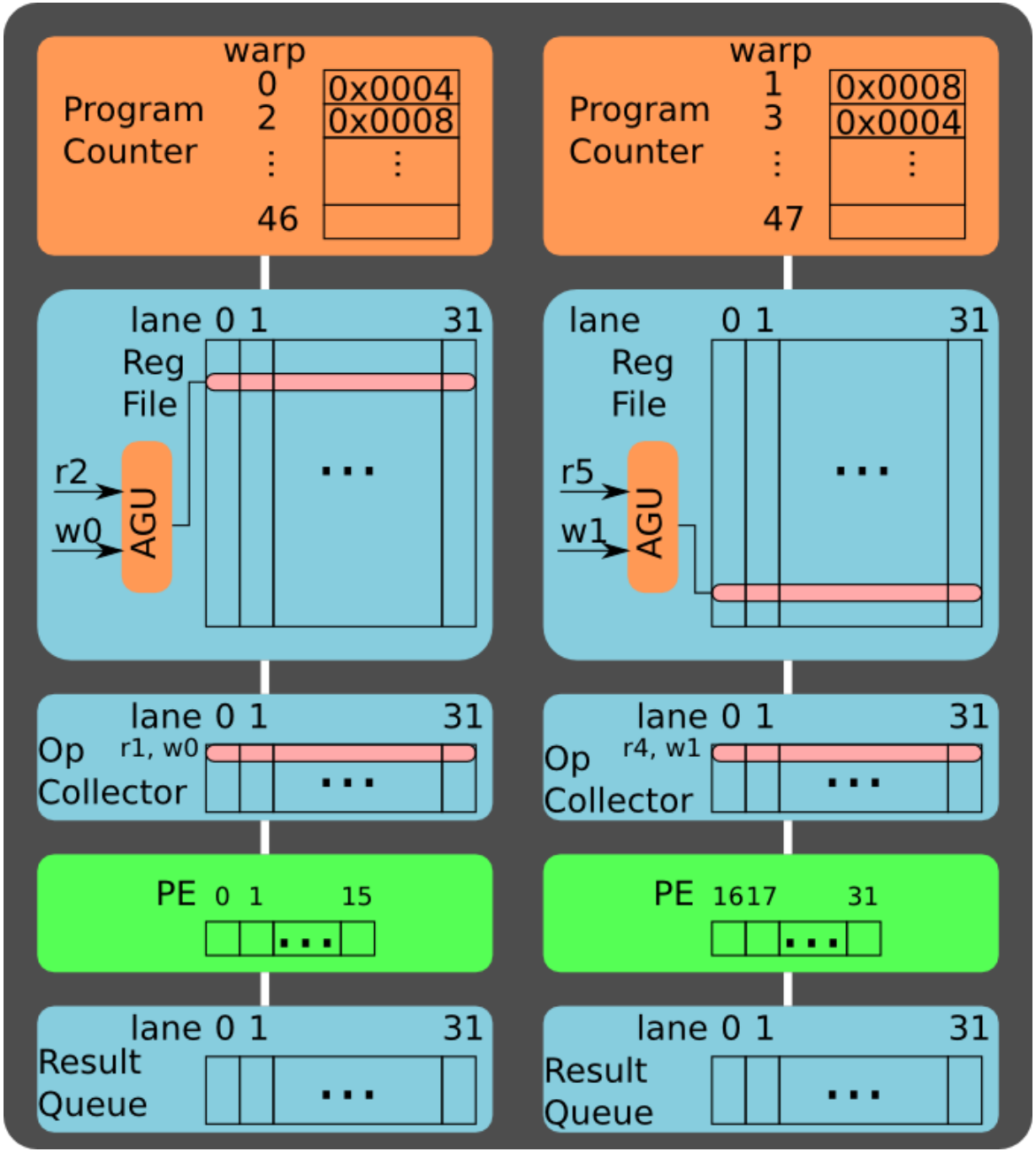
Read Src Op

Program Address: Inst

0x0004: add r0, r1, r2

0x0008: sub r3, r4, r5

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



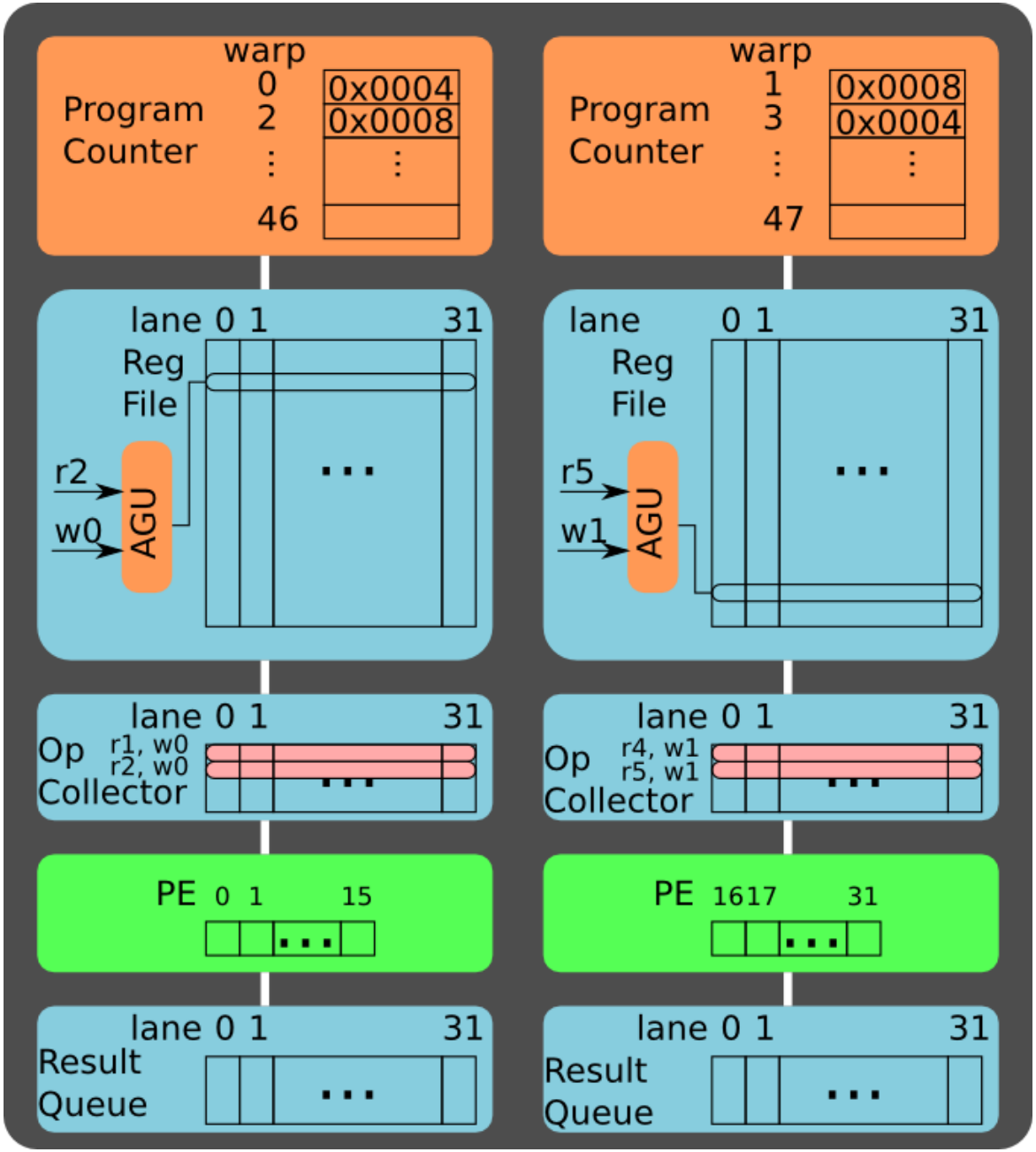
Buffer Src Op

Program Address: Inst

0x0004: add r0, r1, r2

0x0008: sub r3, r4, r5

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



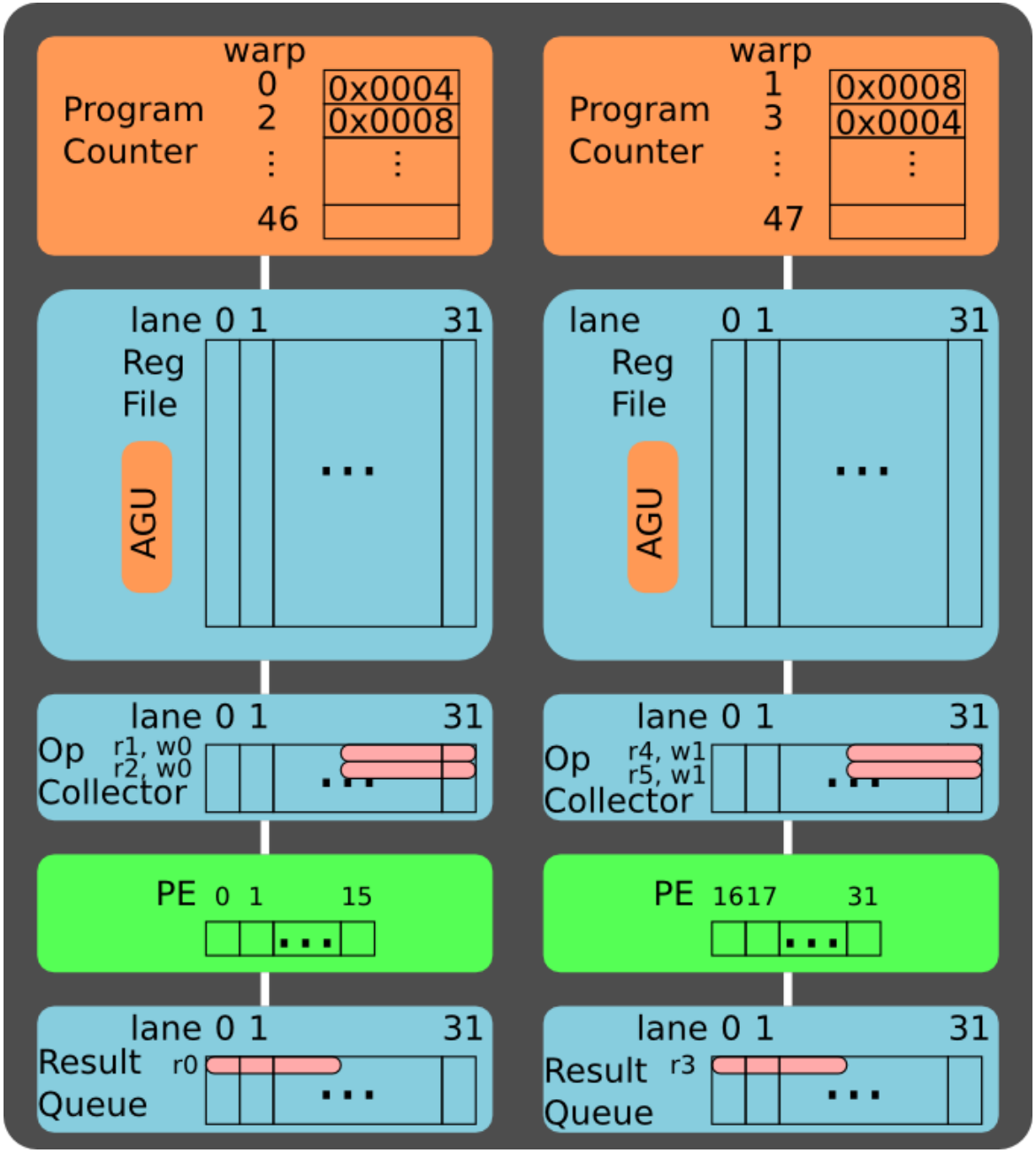
Execute

Program Address: Inst

0x0004: add r0, r1, r2

0x0008: sub r3, r4, r5

Compute the first 16 threads in the warp.



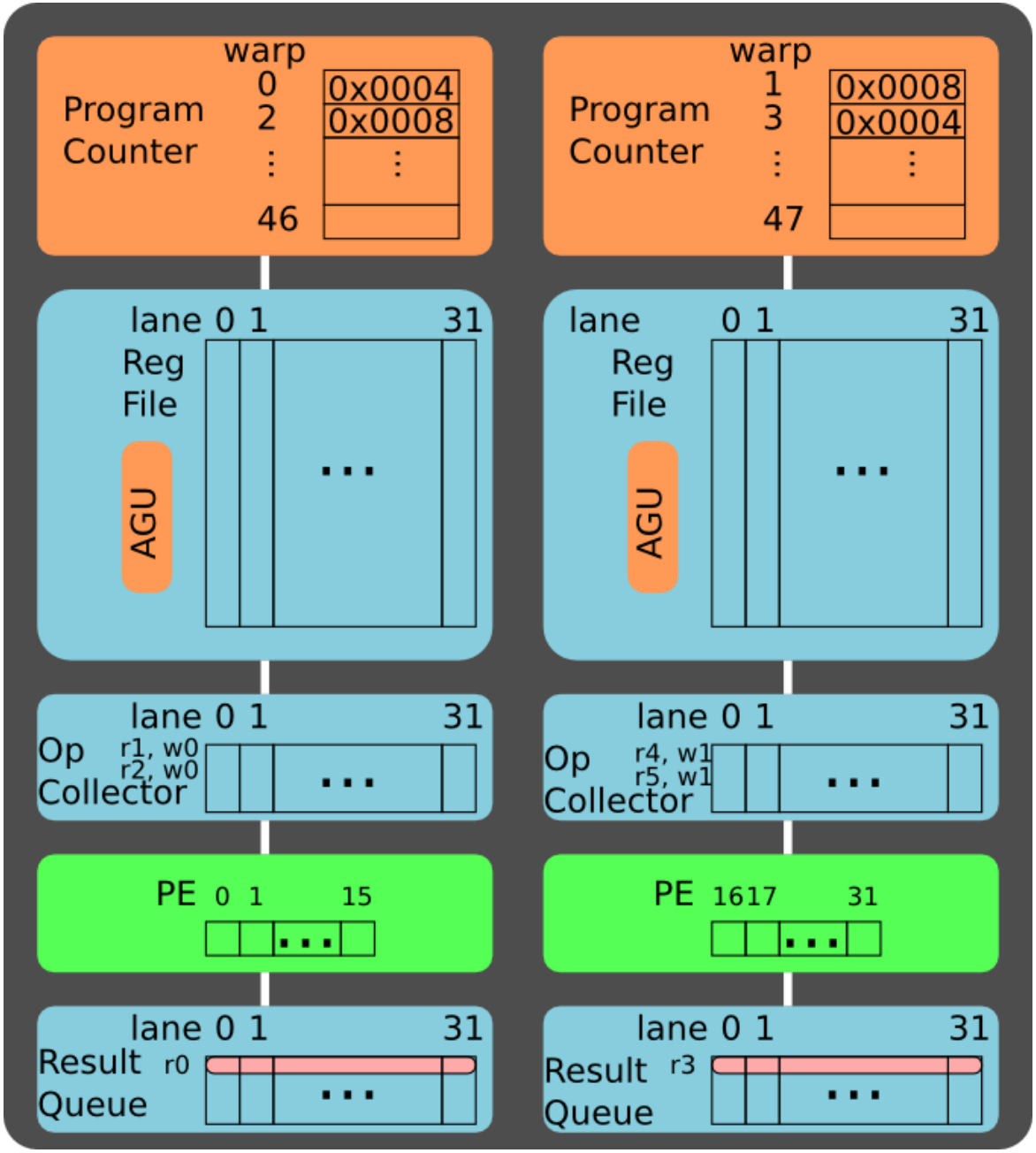
Execute

Program Address: Inst

0x0004: add r0, r1, r2

0x0008: sub r3, r4, r5

Compute the last 16 threads in the warp.



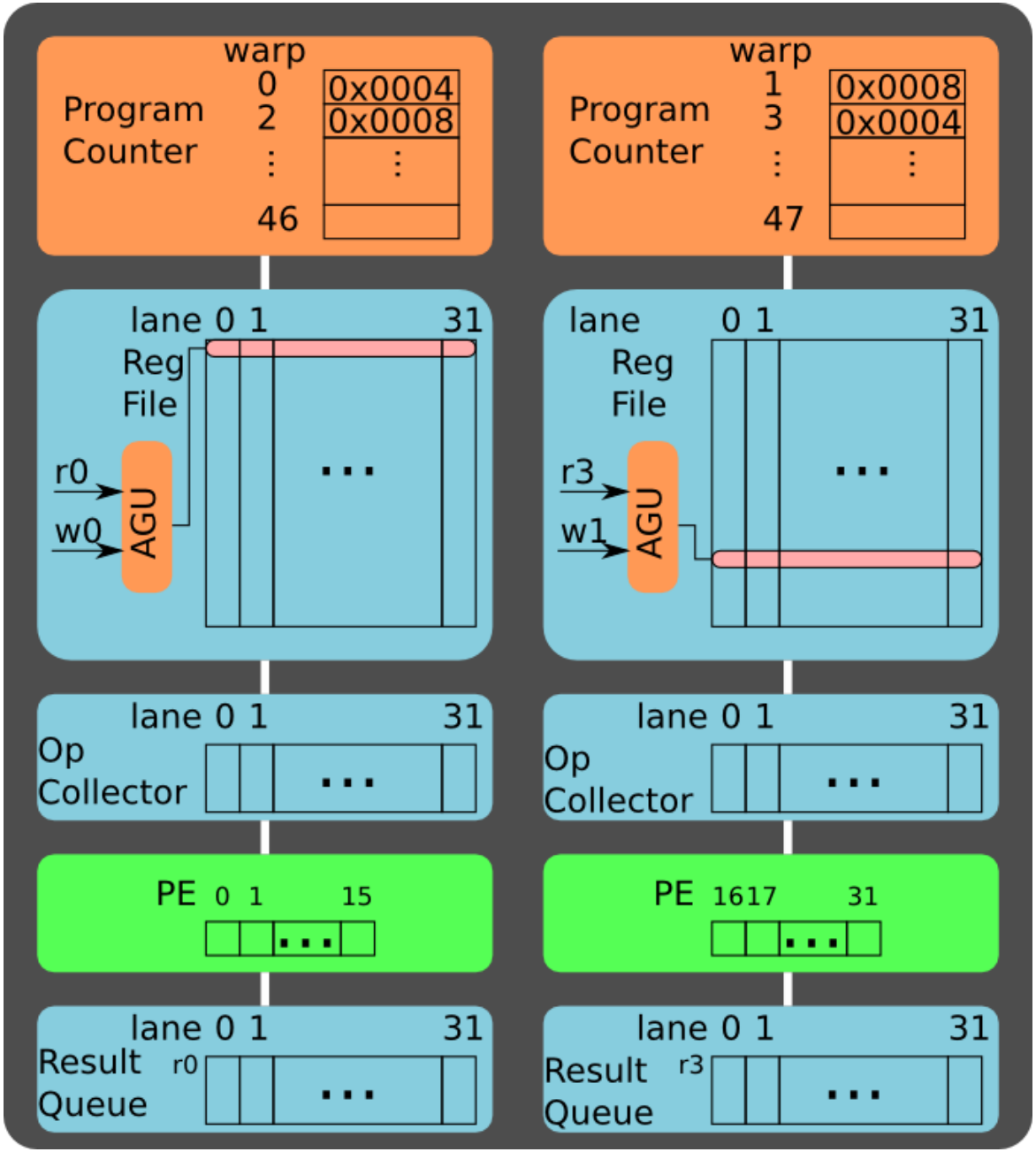
Write back

Program Address: Inst

0x0004: add r0, r1, r2

0x0008: sub r3, r4, r5

Write back:
r0 for warp 0
r3 for warp 1

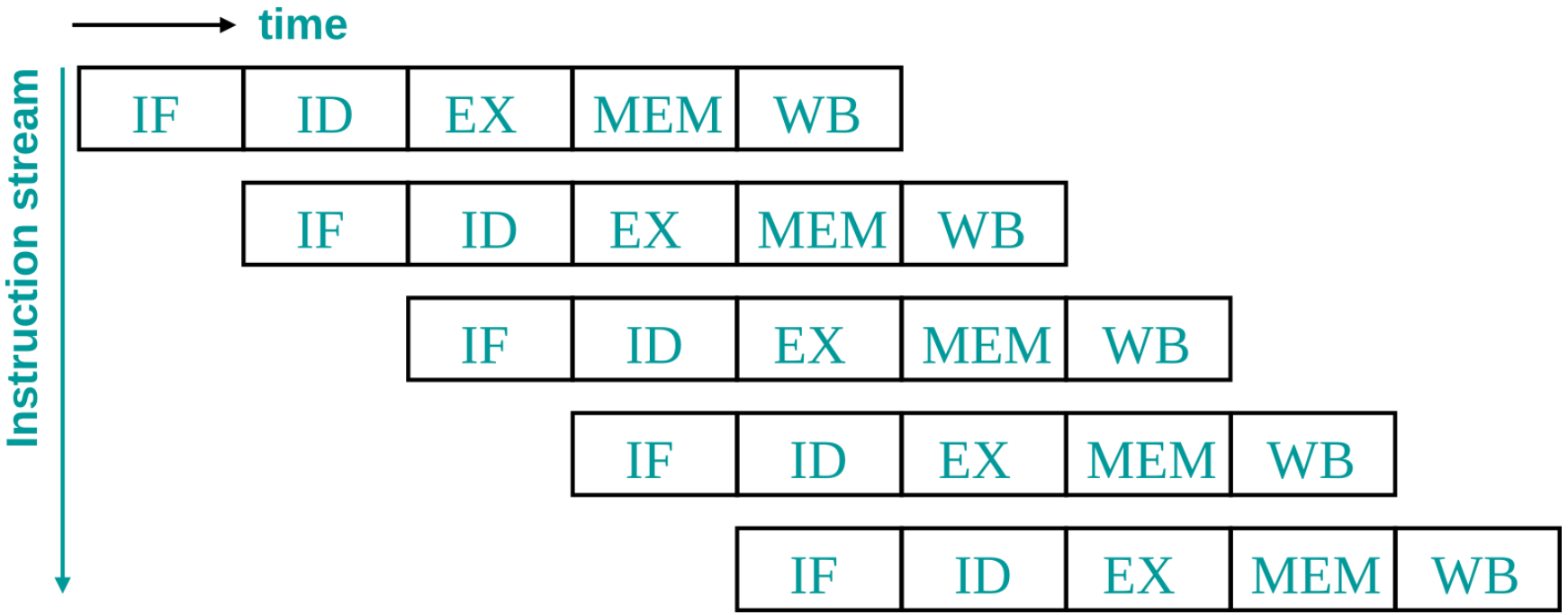


What Are the Possible Hazards?

Three types of hazards we have learned earlier:

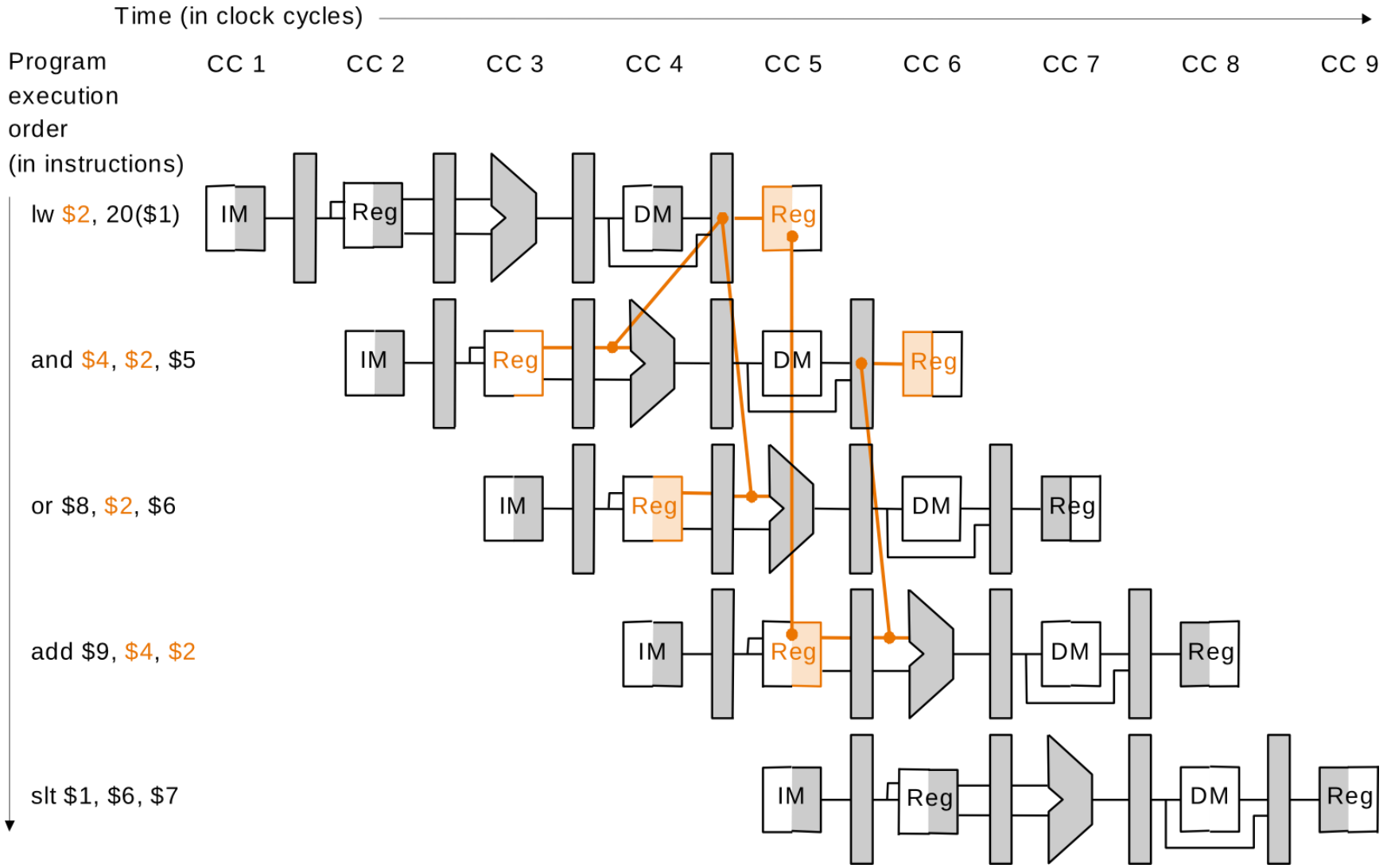
- Structural hazards
- Data hazards
- Control hazards

Structural Hazards



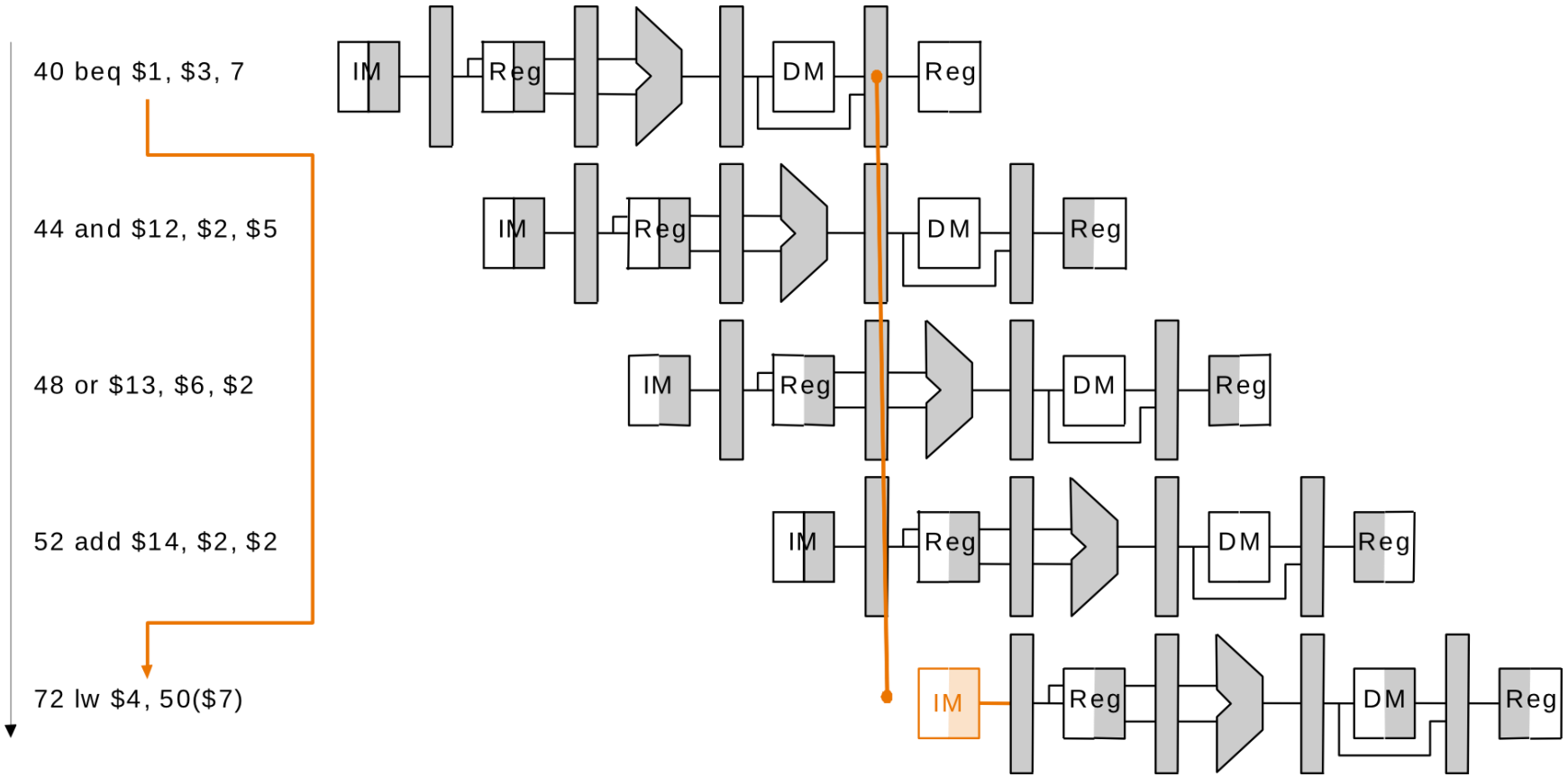
In practice, structural hazards need to be handled dynamically.

Data Hazards



In practice, we have much longer pipeline!

Control Hazards



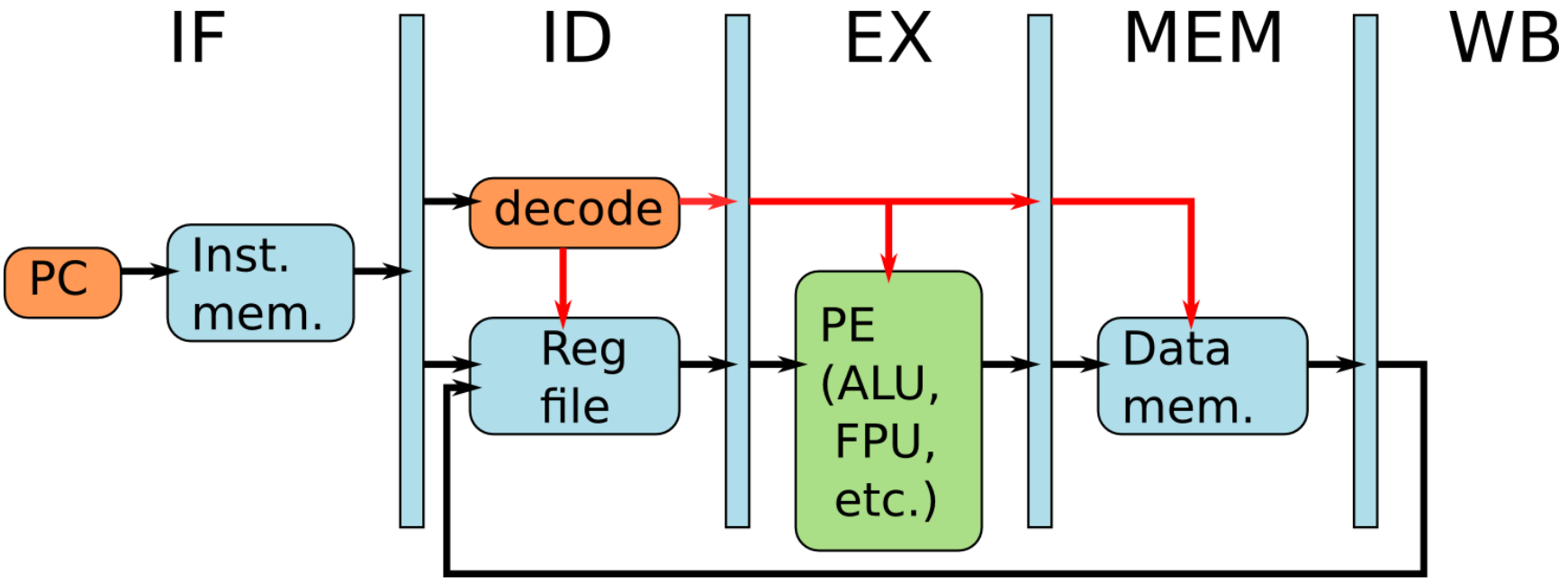
We cannot travel back in time, what should we do?

How Do GPUs Solve Hazards?

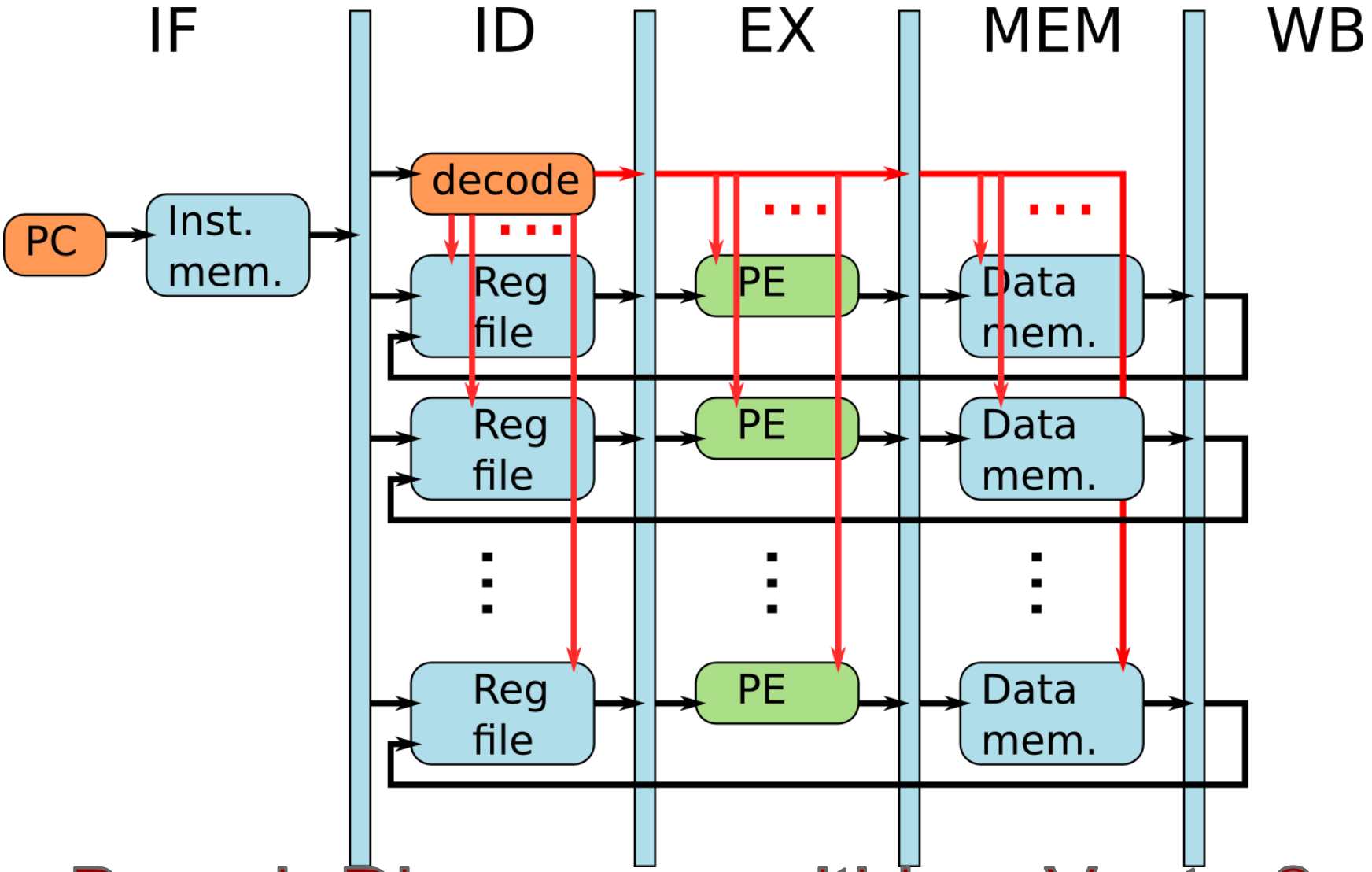
However, GPUs (and SIMDs in general) have additional challenges: the branch!

Let's Start Again from a CPU

Let's start from a MIPS.



A Naive SIMD Processor



Branch Divergence within a Vector?

Handling Branch In GPU

- Threads within a warp are free to branch.

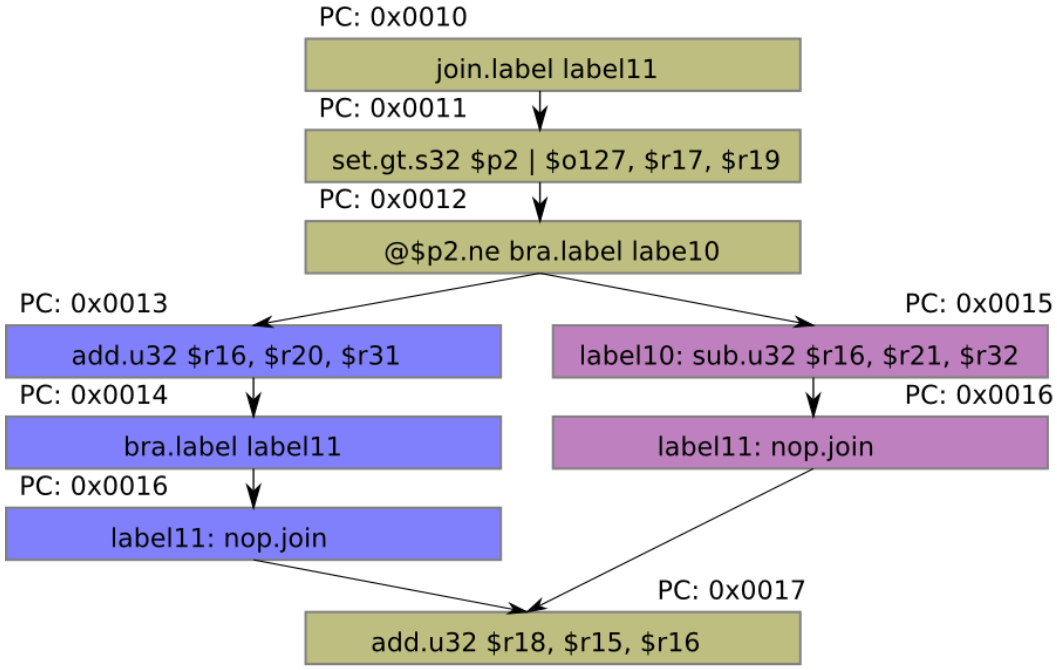
```
if( $r17 > $r19 ){
    $r16 = $r20 + $r31
}
else{
    $r16 = $r21 - $r32
}
$r18 = $r15 + $r16
```

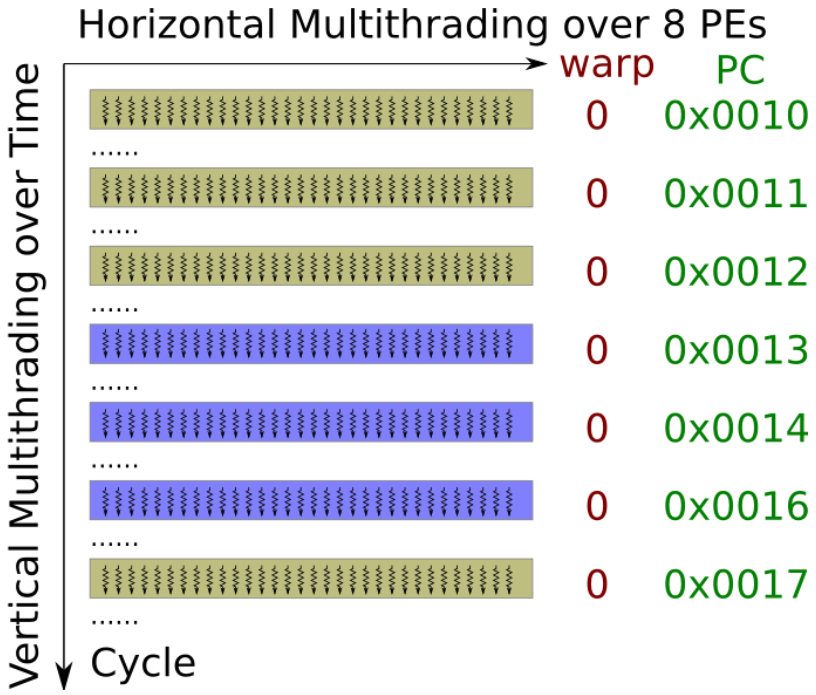
```
PC: 0x0010
    join.label label11
PC: 0x0011
    set.gt.s32 $p2 | $o127, $r17, $r19
PC: 0x0012
    @$p2.ne bra.label labe10
PC: 0x0013
    add.u32 $r16, $r20, $r31
PC: 0x0014
    bra.label label11
PC: 0x0015
label10: sub.u32 $r16, $r21, $r32
PC: 0x0016
label11: nop.join
PC: 0x0017
    add.u32 $r18, $r15, $r16
```

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

If No Branch Divergence in a Warp

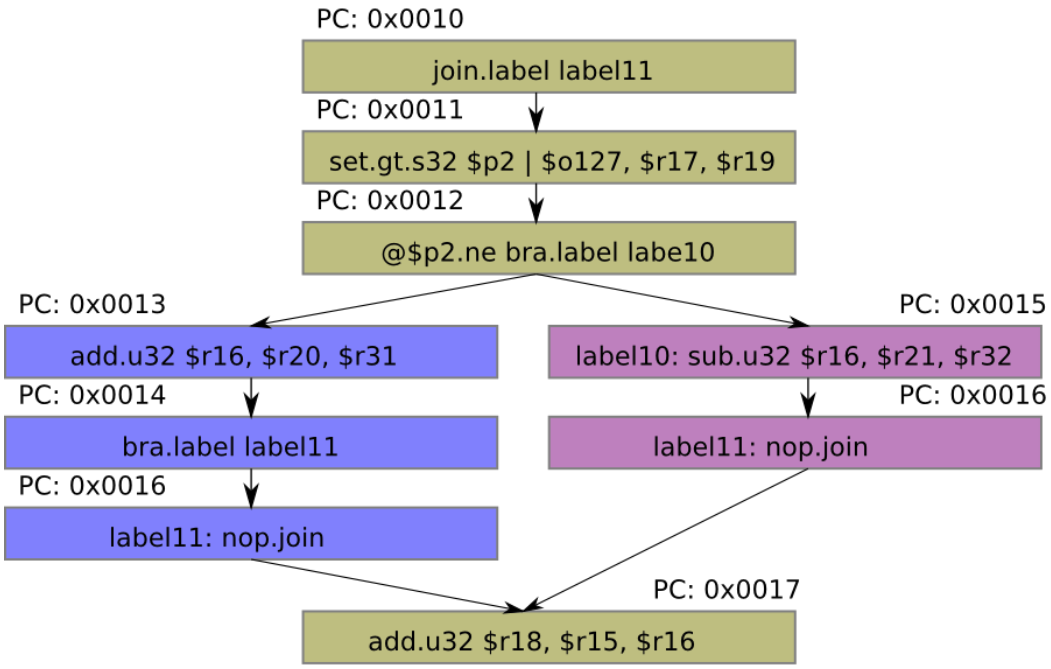
- If threads within a warp take the same path, the other path will not be executed.

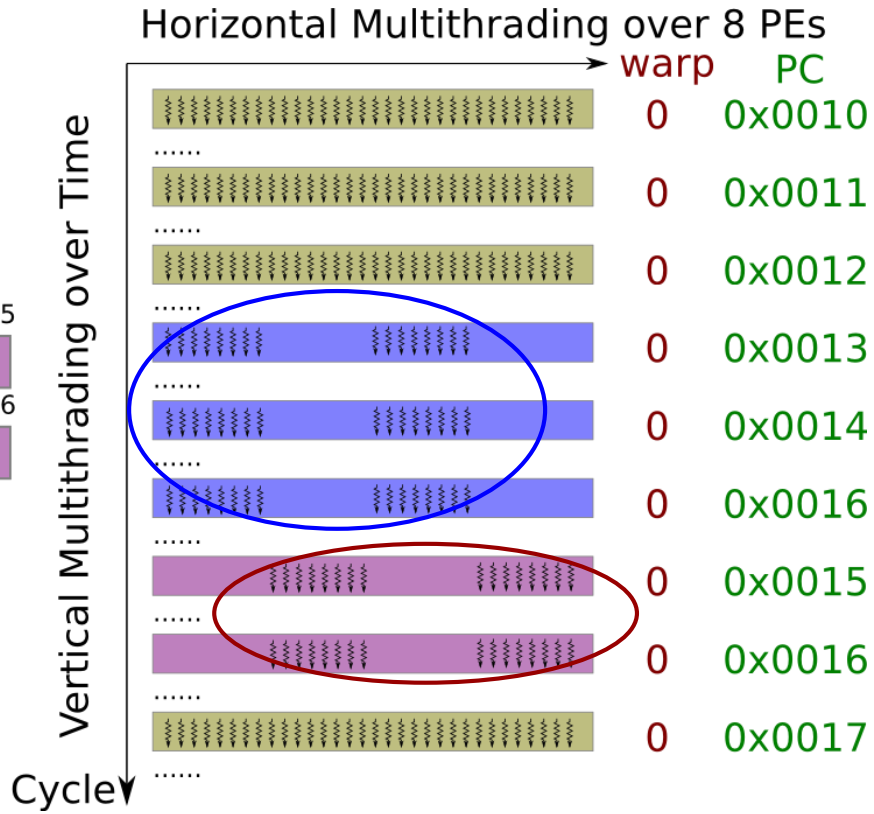




Branch Divergence within a Warp

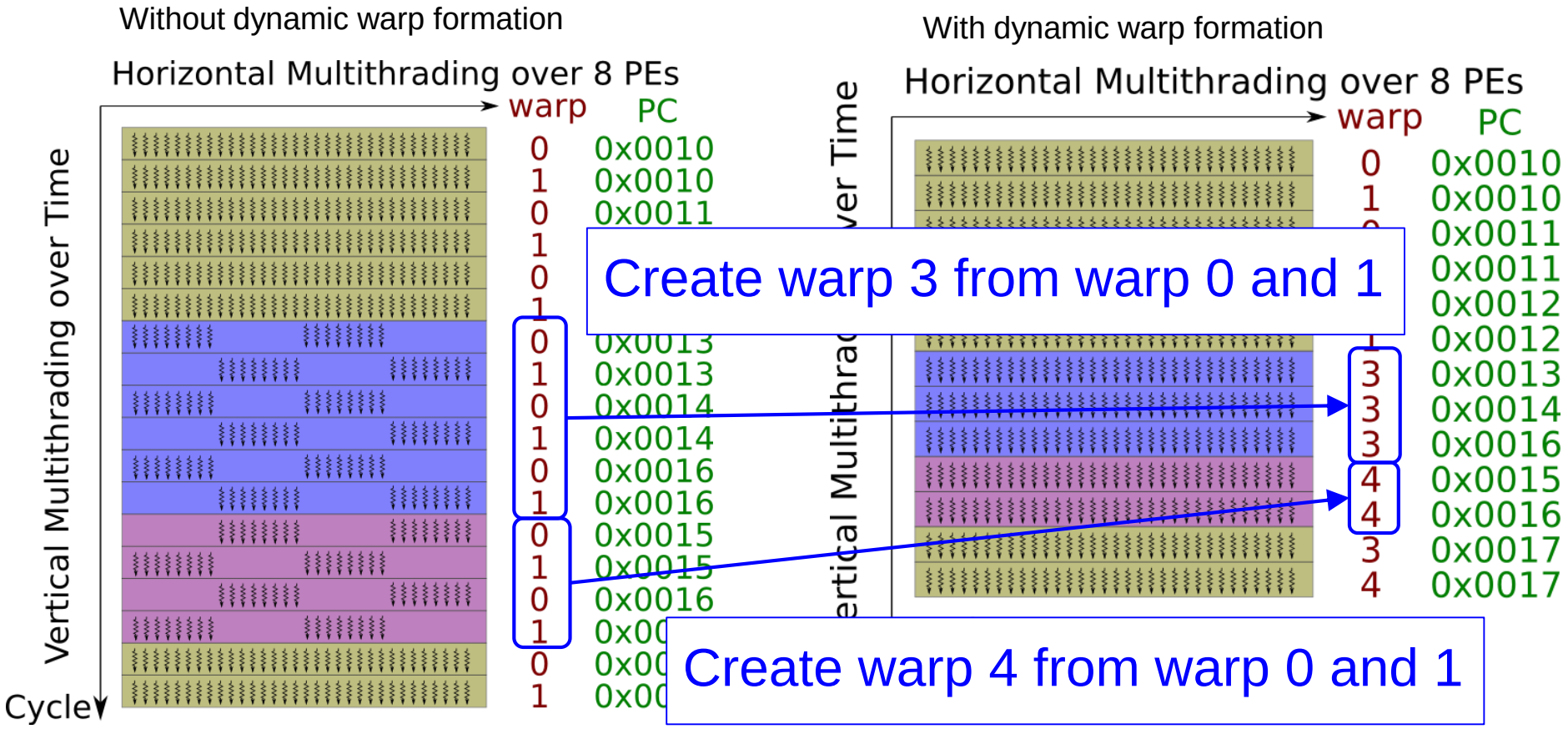
- If threads within a warp diverge, both paths have to be executed.
- Masks are set to filter out threads not executing on current path.





Dynamic Warp Formation

Example: merge two divergent warps into a new warp if possible.

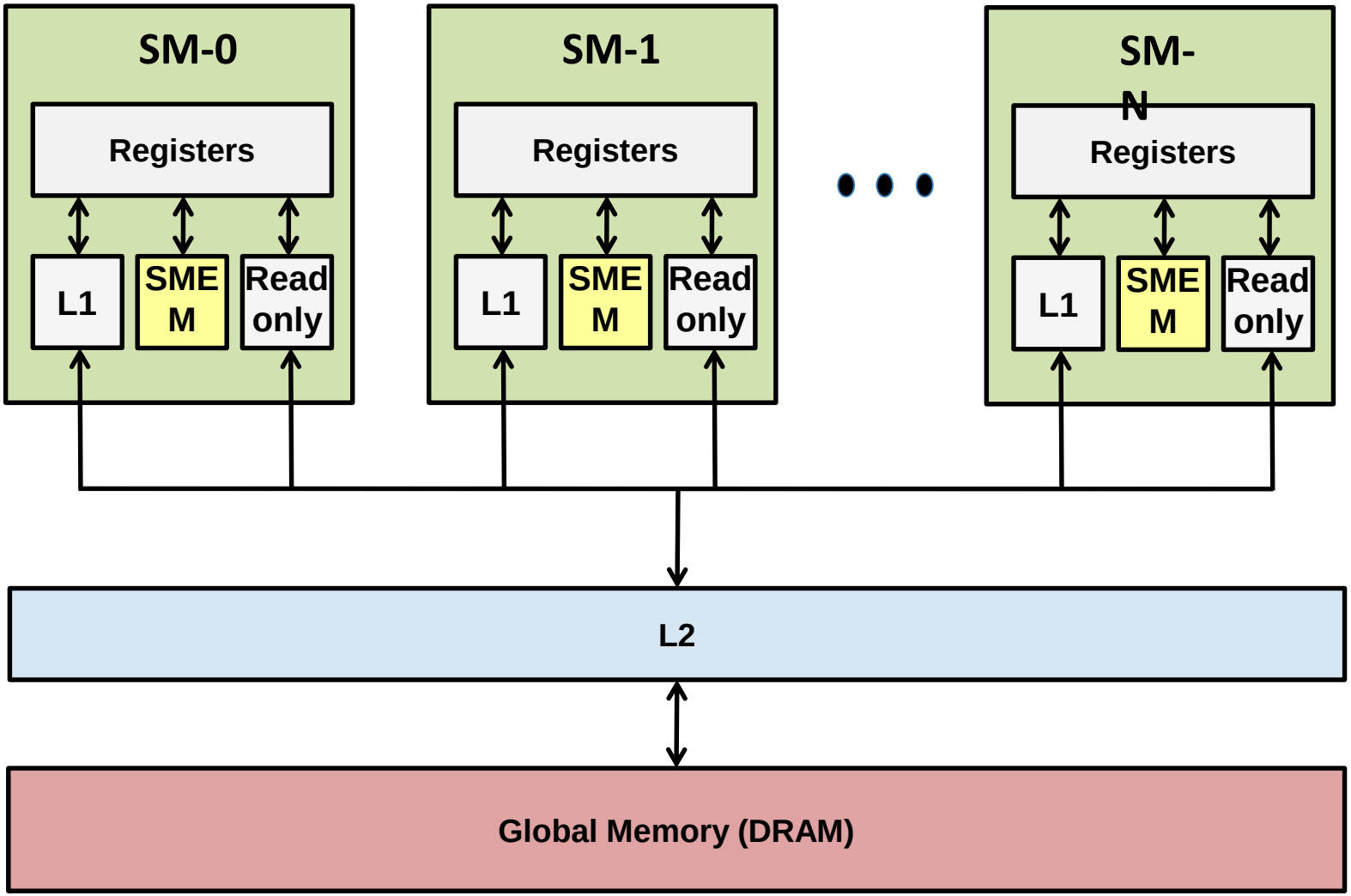


ref: Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. http://dx.doi.org/10.1145/1543753.1543756

GPGPU Memory Hierarchy

Paulius Micikevicius
Developer Technology, NVIDIA

Kepler Memory Hierarchy

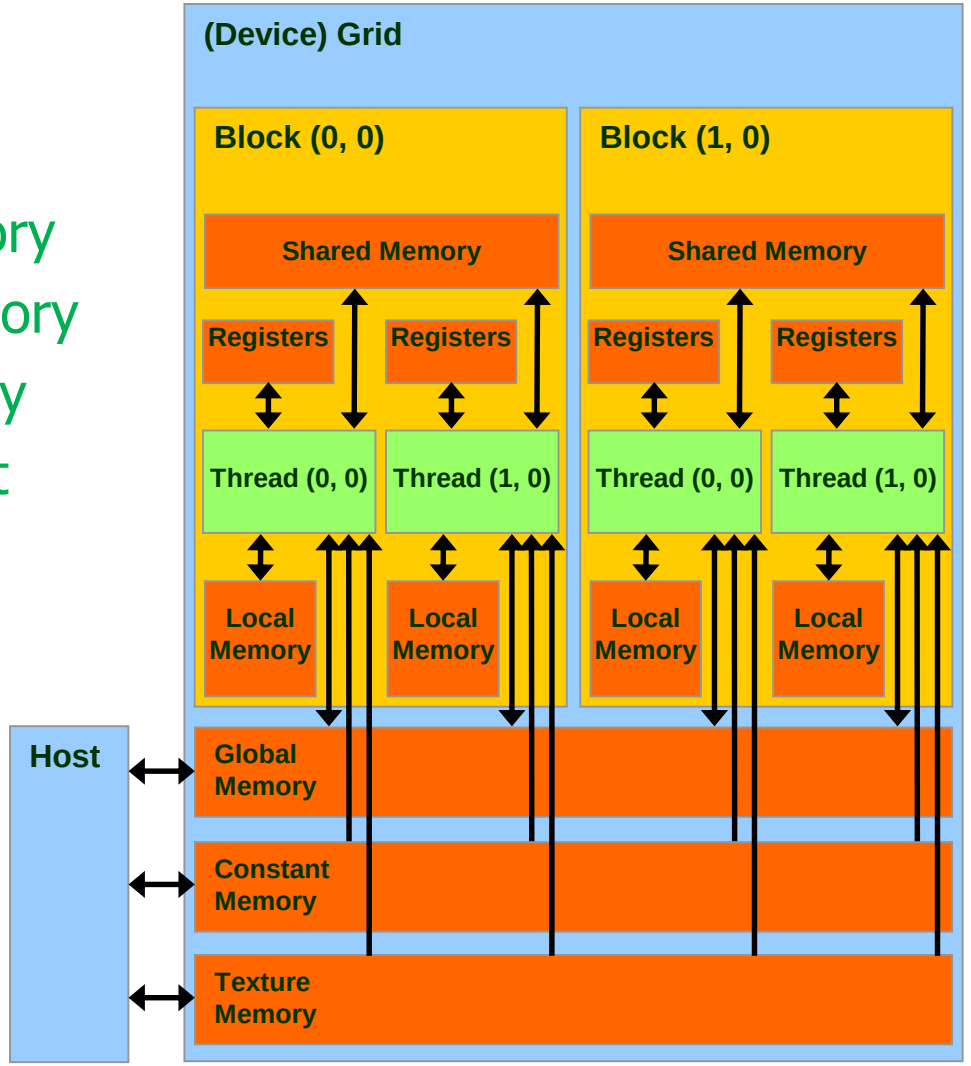


Memory Hierarchy Review

- Registers
  - Storage local to each threads
  - Compiler-managed
- Shared memory / L1
  - 64 KB, program-configurable into shared:L1
  - Program-managed
  - Accessible by all threads in the same threadblock
  - Low latency, high bandwidth: ~2.5 TB/s
- Read-only cache
  - Up to 48 KB per Kepler SM
  - Hardware-managed (also used by texture units)
  - Used for read-only GMEM accesses (not coherent with writes)
- L2
  - 1.5 MB
  - Hardware-managed: all accesses to global memory go through L2, including CPU and peer GPU
- Global memory
  - 6 GB, accessible by all threads, host (CPU), other GPUs in the same system
  - Higher latency (400-800 cycles)
  - 250 GB/s

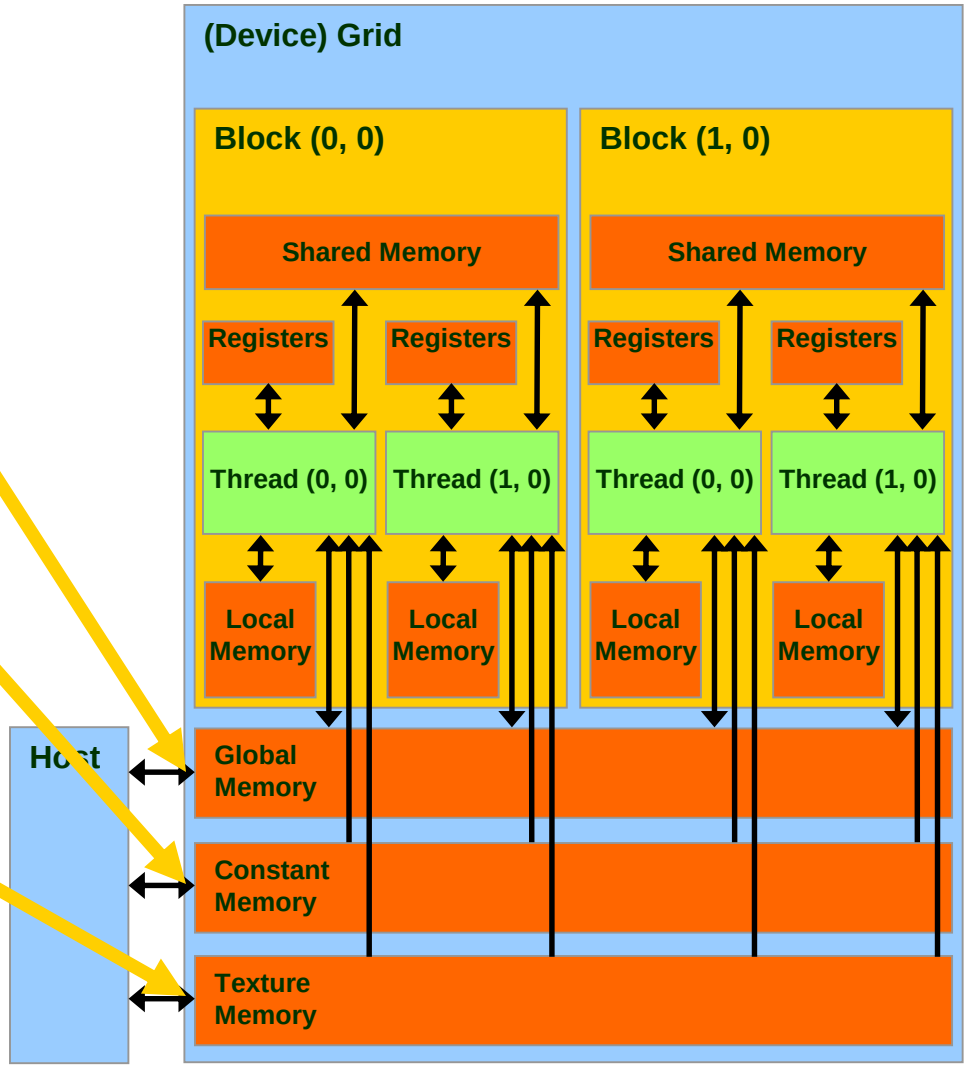
CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



Global, Constant, and Texture Memories

- Global memory:
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories:
  - Constants initialized by host
  - Contents visible to all threads



Access Times

- Register – Dedicated HW – Single cycle
- Shared Memory – Dedicated HW – Single cycle
- Local Memory – DRAM, no cache – "Slow"
- Global Memory – DRAM, no cache – "Slow"
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality

Memory Throughput Analysis

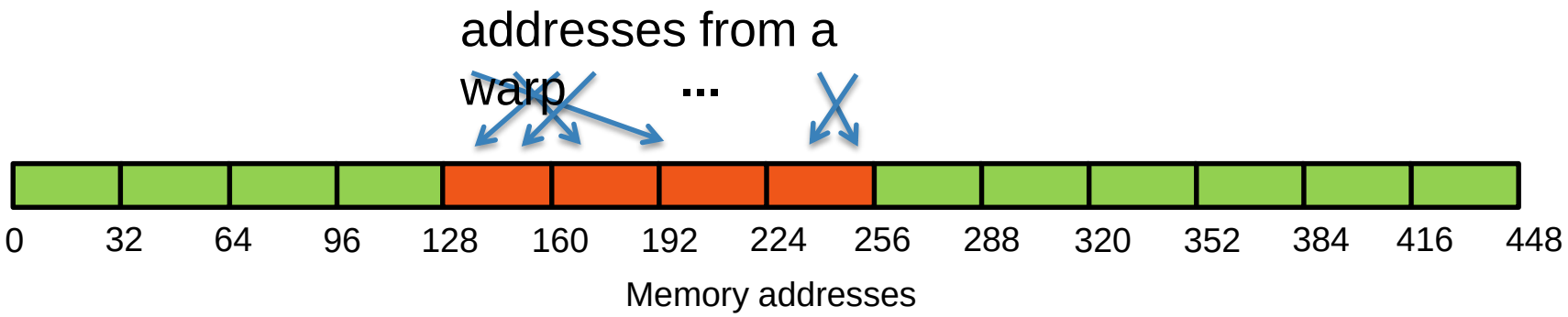
- **Two perspectives on the throughput:**
  - Application's point of view:
    - count only bytes requested by application
  - HW point of view:
    - count all bytes moved by hardware
- The two views can be different:
  - Memory is accessed at 32 byte granularity
    - Scattered/offset pattern: application doesn't use all the hw transaction bytes
  - Broadcast: the same small transaction serves many threads in a warp
- **Two aspects to inspect for performance impact:**
  - Address pattern
  - Number of concurrent accesses in flight

Global Memory Operation

- **Memory operations are executed per warp**
  - 32 threads in a warp provide memory addresses
  - Hardware determines into which lines those addresses fall
    - Memory transaction granularity is 32 bytes
    - There are benefits to a warp accessing a contiguous aligned region of 128 or 256 bytes
- Access word size
  - Natively supported sizes (per thread): 1, 2, 4, 8, 16 bytes
    - Assumes that each thread's address is aligned on the word size boundary
  - If you are accessing a data type that's of non-native size, compiler will generate several load or store instructions with native sizes

- **Scenario:**
  - Warp requests 32 aligned, consecutive 4-byte words
- **Addresses fall within 4 segments**
  - Warp needs 128 bytes
  - 128 bytes move across the bus
  - Bus utilization: 100%

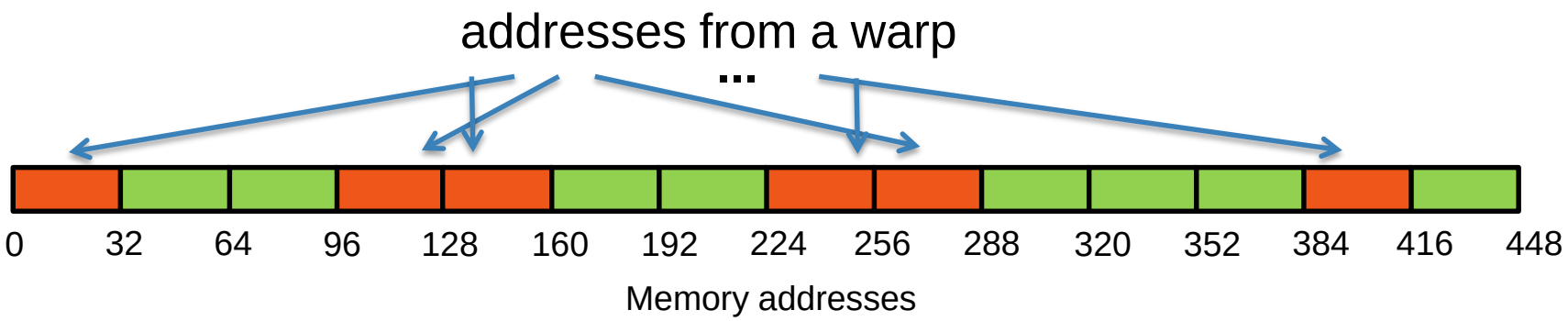
- **Scenario:**
  - Warp requests 32 aligned, permuted 4-byte words
- **Addresses fall within 4 segments**
  - Warp needs 128 bytes
  - 128 bytes move across the bus
  - Bus utilization: 100%



- **Scenario:**
  - Warp requests 32 misaligned, consecutive 4-byte words
- **Addresses fall within at most 5 segments**
  - Warp needs 128 bytes
  - At most 160 bytes move across the bus
  - Bus utilization: at least 80%
    - Some misaligned patterns will fall within 4 segments, so 100% utilization

- **Scenario:**
  - All threads in a warp request the same 4-byte word
- **Addresses fall within a single segment**
  - Warp needs 4 bytes
  - 32 bytes move across the bus
  - Bus utilization: 12.5%

- **Scenario:**
  - Warp requests 32 scattered 4-byte words
- **Addresses fall within *N* segments**
  - Warp needs 128 bytes
  - *N**32 bytes move across the bus
  - Bus utilization: 128 / (*N**32)



Structure of Non-Native Size

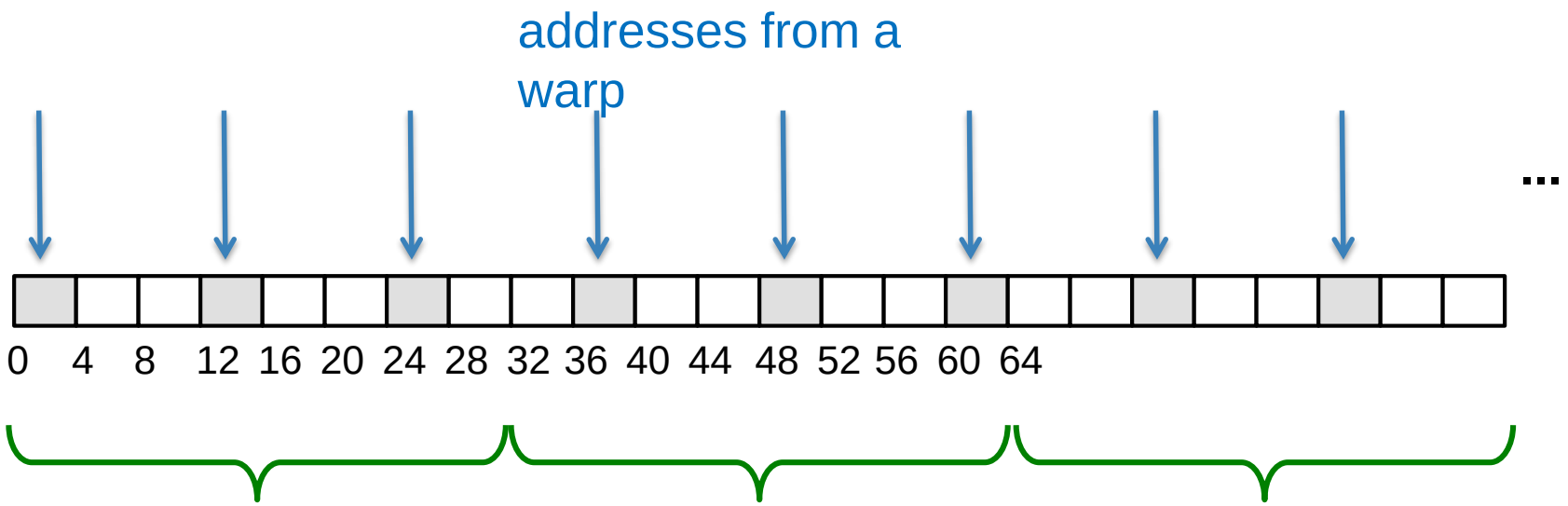
- Say we are reading a 12-byte structure per thread

```
struct Position
{
    float x, y, z;
};
...
__global__ void kernel( Position *data, ... )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    Position temp = data[idx];
    ...
}
```

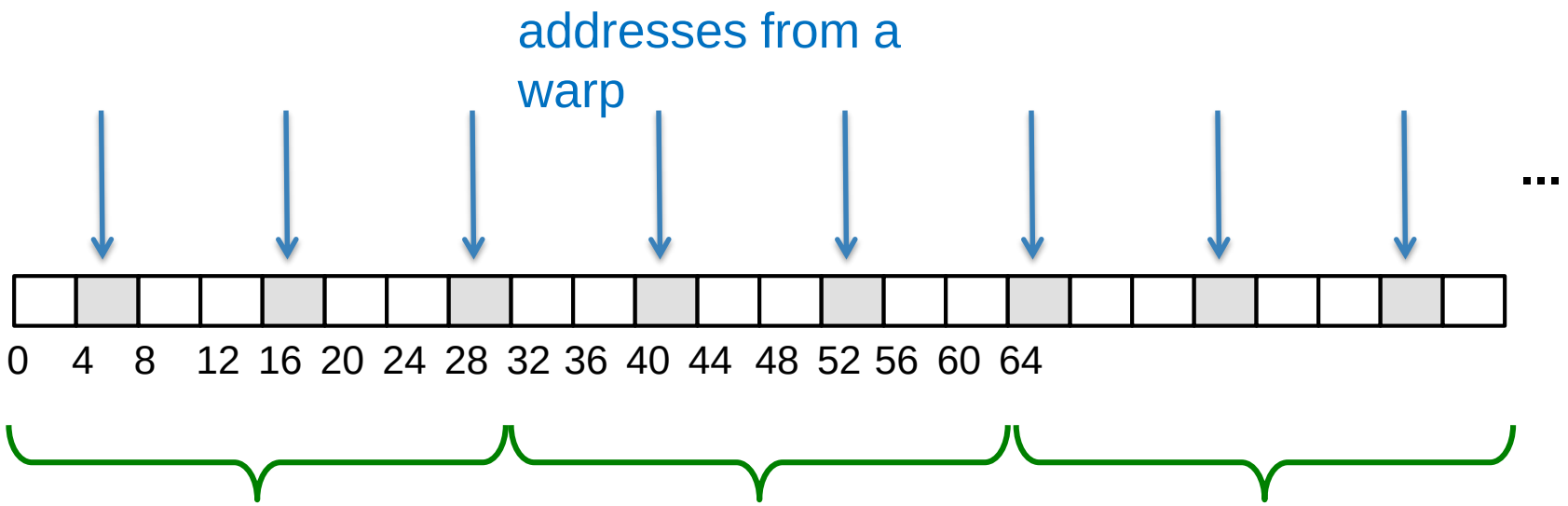
Structure of Non-Native Size

- Compiler converts *temp = data[idx]* into 3 loads:
  - Each loads 4 bytes
  - Can't do an 8 and a 4 byte load: 12 bytes per element means that every other element wouldn't align the 8-byte load on 8-byte boundary
- Addresses per warp for each of the loads:
  - Successive threads read 4 bytes at 12-byte stride

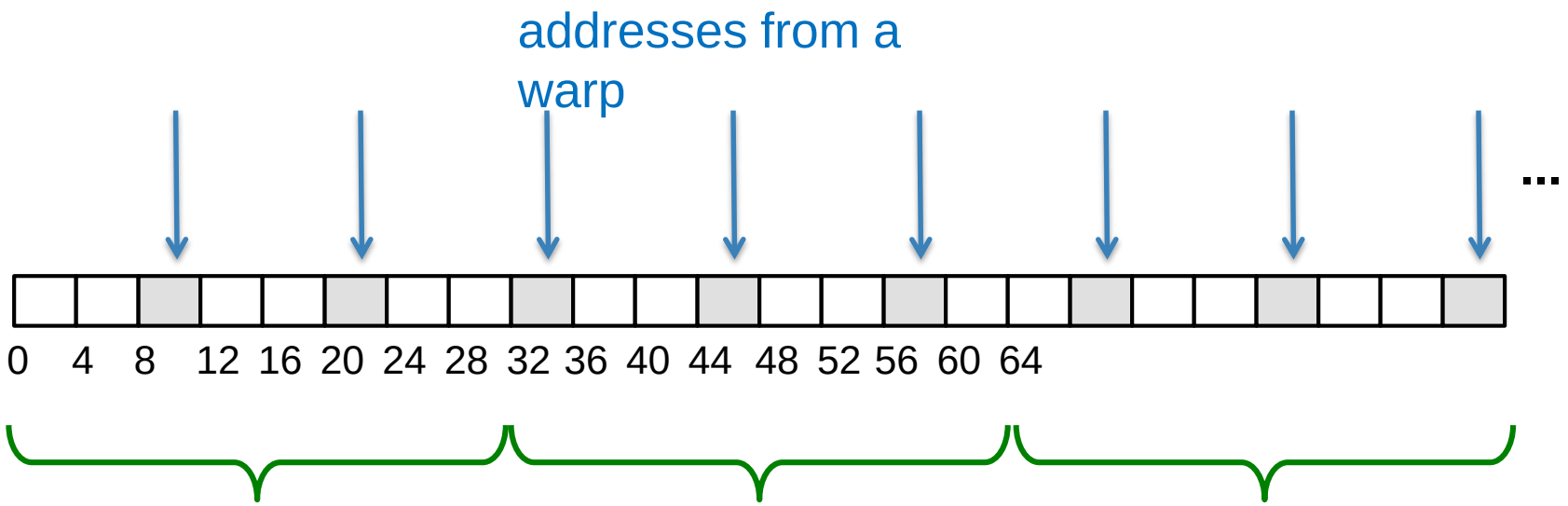
First Load Instruction



Second Load Instruction



Third Load Instruction



Performance and Solutions

- Because of the address pattern, we end up moving 3x more bytes than application requests
  - We waste a lot of bandwidth, leaving performance on the table
- Potential solutions:
  - Change data layout from array of structures to structure of arrays
    - In this case: 3 separate arrays of floats
    - The most reliable approach (also ideal for both CPUs and GPUs)
  - Use loads via read-only cache
    - As long as lines survive in the cache, performance will be nearly optimal
  - Stage loads via shared memory

Read-only Loads

- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- Two ways to enable:
  - Decorating pointer arguments as hints to compiler:
    - Pointer of interest: const __restrict__
    - All other pointer arguments: __restrict__
      - Conveys to compiler that no aliasing will occur
  - Using __ldg() intrinsic
    - Requires no pointer decoration

Read-only Loads

- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- Two ways to enable:
  - Decorating pointe
    - Pointer of inter
    - All other pointe
      - Conveys to co
  - Using __ldg() intri
    - Requires no po

```
__global__ void kernel(  int*__restrict__
output,
                                const int*
__restrict__ input )
{
    ...
    output[idx] = ... + input[idx];
}
```

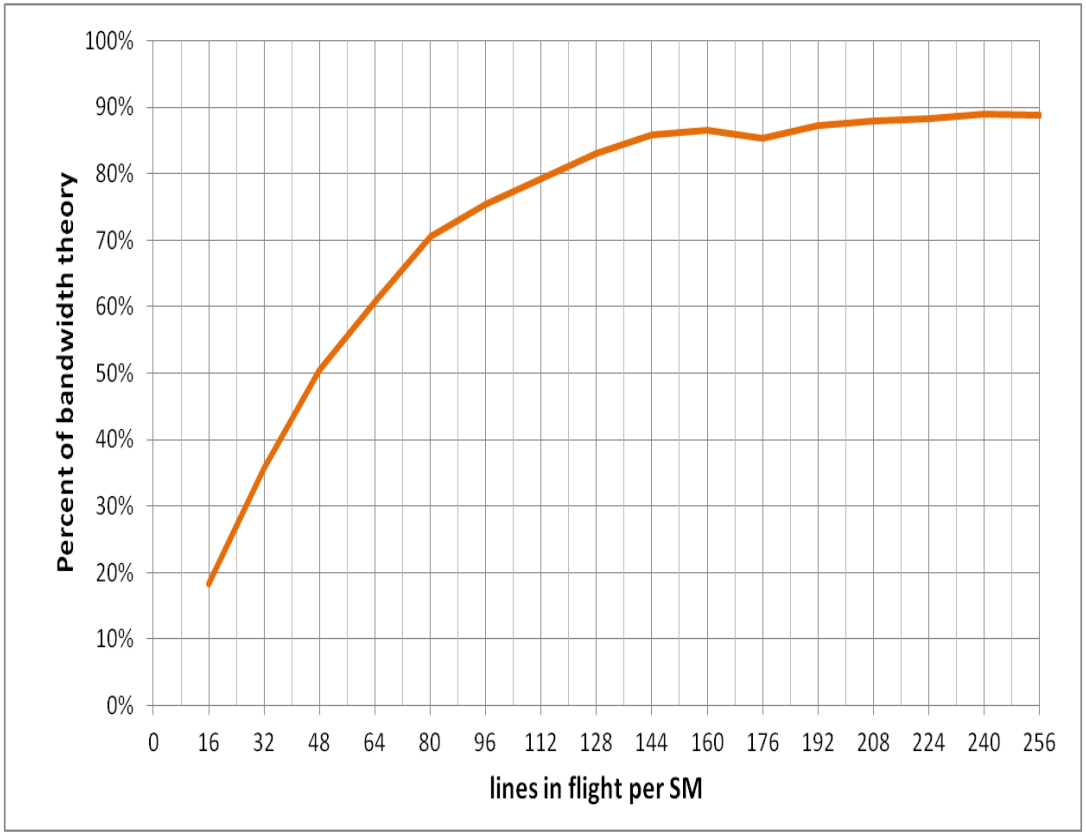
Read-only Loads

- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- Two ways to enable:
  - Decorating pointe
    - Pointer of inter
    - All other pointe
      - Conveys to co
  - Using __ldg() intri
    - Requires no po

```
__global__ void kernel(  int *output,
                         int *input )
{
    ...
    output[idx] = ... + __ldg( &input[idx] );
}
```

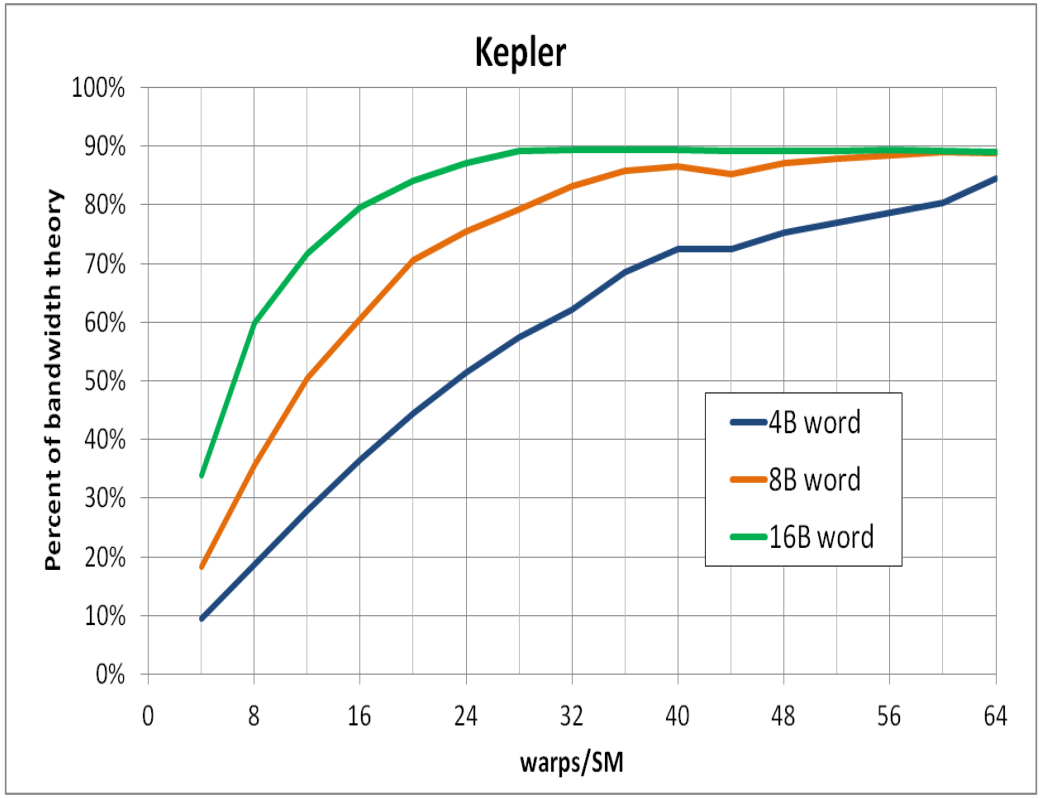
Having Sufficient Concurrent Accesses

- In order to saturate memory bandwidth, SM must issue enough independent memory requests



Elements per Thread and Performance

- **Experiment: each warp has 2 concurrent requests (memcopy, one word per thread)**
  - 4B word request: 1 line
  - 8B word request: 2 lines
  - 16B word request: 4 lines



- **To achieve the same throughput at lower occupancy:**
  - Need more independent requests per warp
- **To achieve the same throughput with smaller words:**
  - Need more independent requests per warp

Optimizing Access Concurrency

- **Have enough concurrent accesses to saturate the bus**
  - Little's law: need (mem_latency)x(bandwidth) bytes
- **Ways to increase concurrent accesses:**
  - Increase occupancy (run more warps concurrently)
    - Adjust threadblock dimensions
      - To maximize occupancy at given register and smem requirements
    - If occupancy is limited by registers per thread:
      - Reduce register count (-maxrregcount option, or __launch_bounds__)
  - Modify code to process several elements per thread
    - Doubling elements per thread doubles independent accesses per thread

Optimizations When Addresses Are Coalesced

- When looking for more performance and code:
  - Is memory bandwidth limited
  - Achieves high percentage of bandwidth theory
  - Addresses are coalesced
- Consider compression
  - GPUs provide instructions for converting between fp16, fp32, and fp64 representations:
    - A single instruction, implemented in hw (__float2half(), ...)
  - If data has few distinct values, consider lookup tables
    - Store indices into the table
    - Small enough tables will likely survive in caches if used often enough

L1 Sizing

- Shared memory and L1 use the same 64KB
  - Program-configurable split:
    - Fermi: 48:16, 16:48
    - Kepler: 48:16, 16:48, 32:32
  - CUDA API: cudaDeviceSetCacheConfig(), cudaFuncSetCacheConfig()
- Large L1 can improve performance when:
  - Spilling registers (more lines in the cache -> fewer evictions)
- Large SMEM can improve performance when:
  - Occupancy is limited by SMEM

Summary: GMEM Optimization

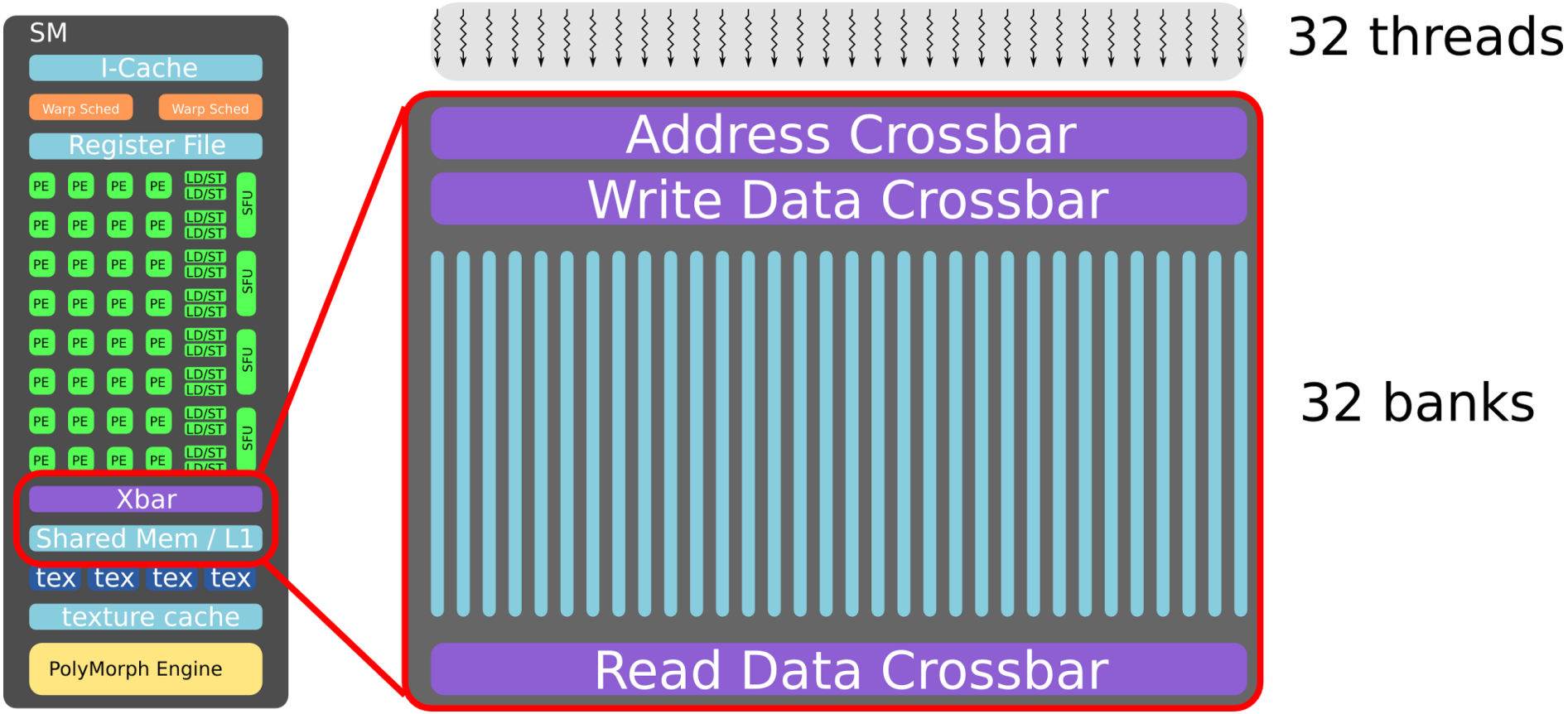
- Strive for perfect address coalescing per warp
  - Align starting address (may require padding)
  - A warp will ideally access within a contiguous region
  - Avoid scattered address patterns or patterns with large strides between threads
- Analyze and optimize address patterns:
  - Use profiling tools (included with CUDA toolkit download)
  - Compare the transactions per request to the ideal ratio
  - Choose appropriate data layout (prefer SoA)
  - If needed, try read-only loads, staging accesses via SMEM
- Have enough concurrent accesses to saturate the bus
  - Launch enough threads to maximize throughput
    - Latency is hidden by switching threads (warps)
  - If needed, process several elements per thread
    - More concurrent loads/stores

SHARED MEMORY

Shared Memory

- On-chip (on each SM) memory
- Comparing SMEM to GMEM:
  - Order of magnitude (20-30x) lower latency
  - Order of magnitude (~10x) higher bandwidth
  - Accessed at bank-width granularity
    - Kepler: 8 bytes
    - For comparison: GMEM access granularity is either 32 or 128 bytes
- SMEM instruction operation:
  - 32 threads in a warp provide addresses
  - Determine into which 8-byte words addresses fall
  - Fetch the words, distribute the requested bytes among the threads
    - Multi-cast capable
    - Bank conflicts cause serialization

Shared Mem Contains Multiple Banks



Kepler Shared Memory Banking

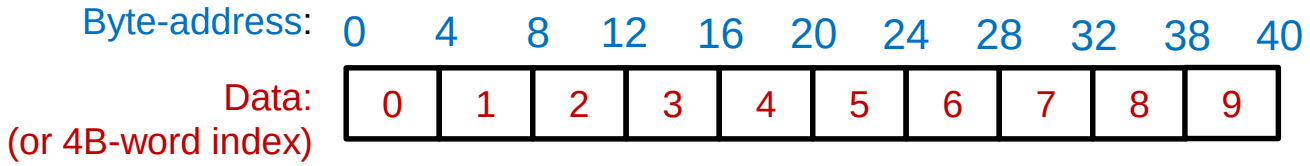
- 32 banks, 8 bytes wide
  - Bandwidth: 8 bytes per bank per clock per SM (256 bytes per clk per SM)
  - 2x the bandwidth compared to Fermi
- Two modes:
  - 4-byte access (default):
    - Maintains Fermi bank-conflict behavior exactly
    - Provides 8-byte bandwidth for certain access patterns
  - 8-byte access:
    - Some access patterns with Fermi-specific padding may incur bank conflicts
    - Provides 8-byte bandwidth for all patterns (assuming 8-byte words)
  - Selected with cudaDeviceSetSharedMemConfig() function arguments:
    - cudaSharedMemBankSizeFourByte
    - cudaSharedMemBankSizeEightByte

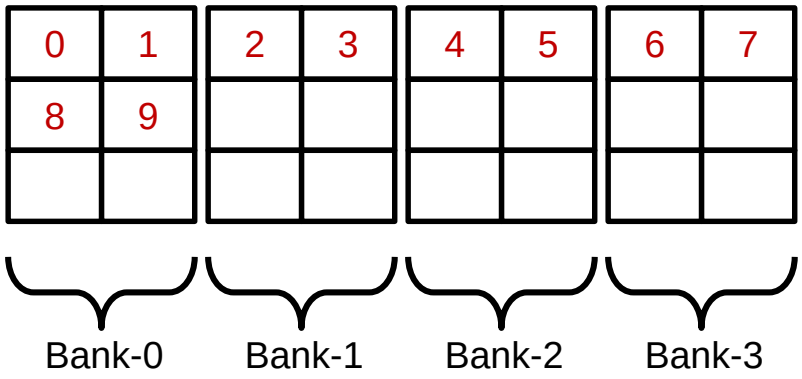
Kepler 8-byte Bank Mode

- Mapping addresses to banks:
  - Successive 8-byte words go to successive banks
  - Bank index:
    - (8B word index) mod 32
    - (4B word index) mod (32*2)
    - (byte address) mod (32*8)
  - Given the 8 least-significant address bits: ...BBBBBxxx
    - xxx selects the byte within an 8-byte word
    - BBBBB selects the bank
    - Higher bits select a “column” within a bank

Kepler 8-byte Bank Mode

- To visualize, let's pretend we have 4 banks, not 32 (easier to draw)





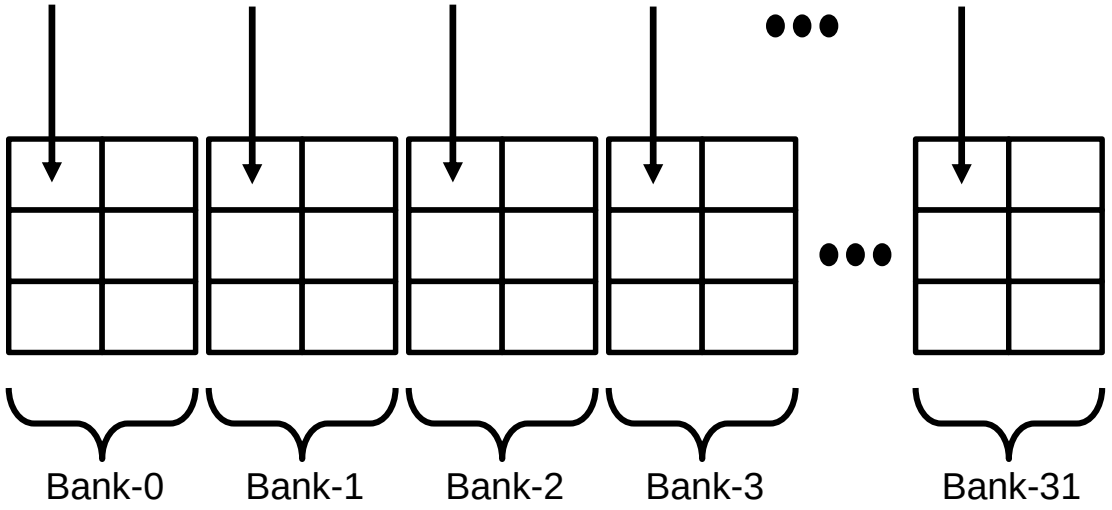
Shared Memory Bank Conflicts

- A bank conflict occurs when:
  - 2 or more threads in a warp access different words in the same bank
    - Think: 2 or more threads access different “rows” in the same bank
  - N-way bank conflict: N threads in a warp conflict
    - Instruction gets issued N times: increases latency
- Note there is no bank conflict if:
  - Several threads access the same word
  - Several threads access different bytes of the same word

SMEM Access Examples

Addresses from a warp: no bank conflicts

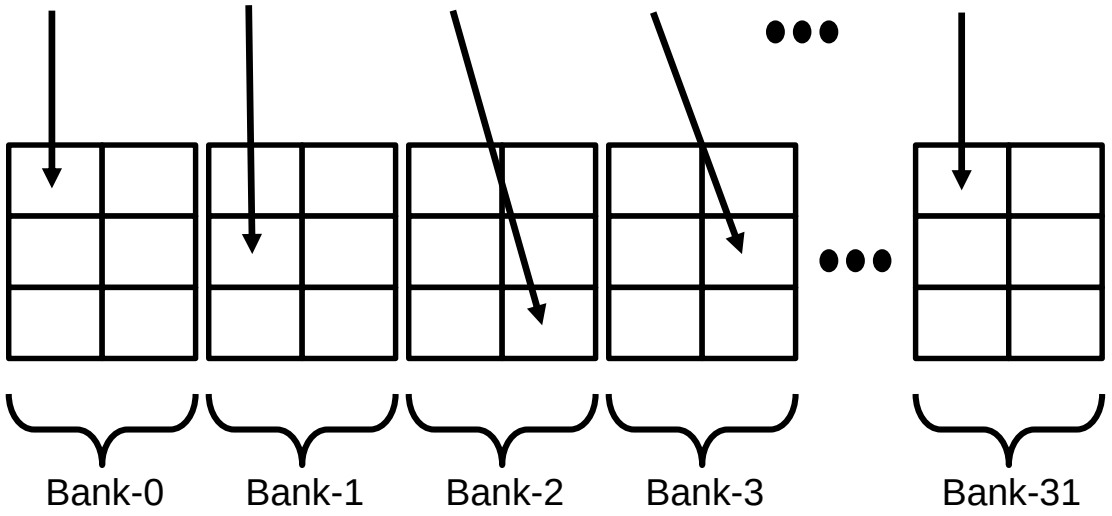
One address access per bank



SMEM Access Examples

Addresses from a warp: no bank conflicts

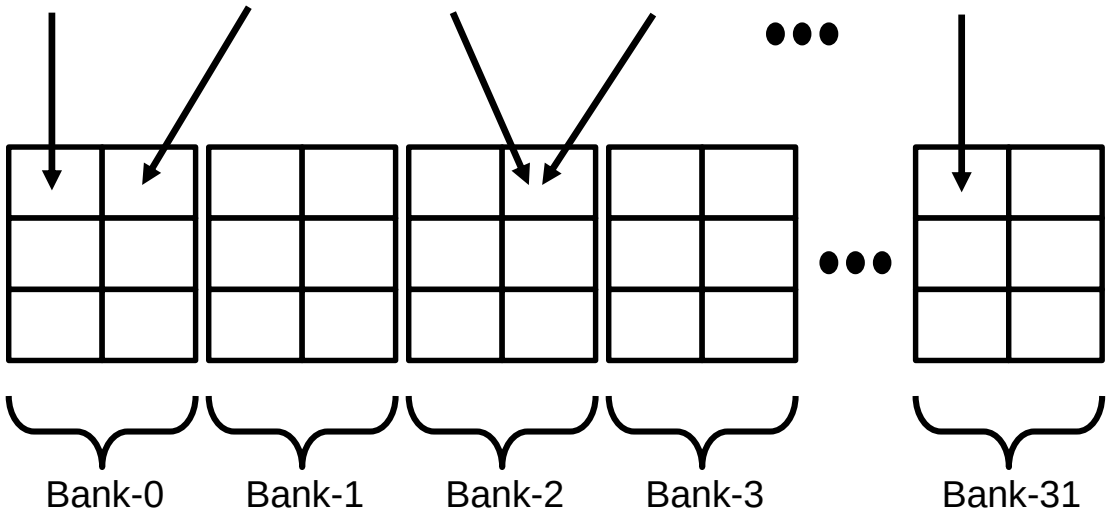
One address access per bank



SMEM Access Examples

Addresses from a warp: no bank conflicts

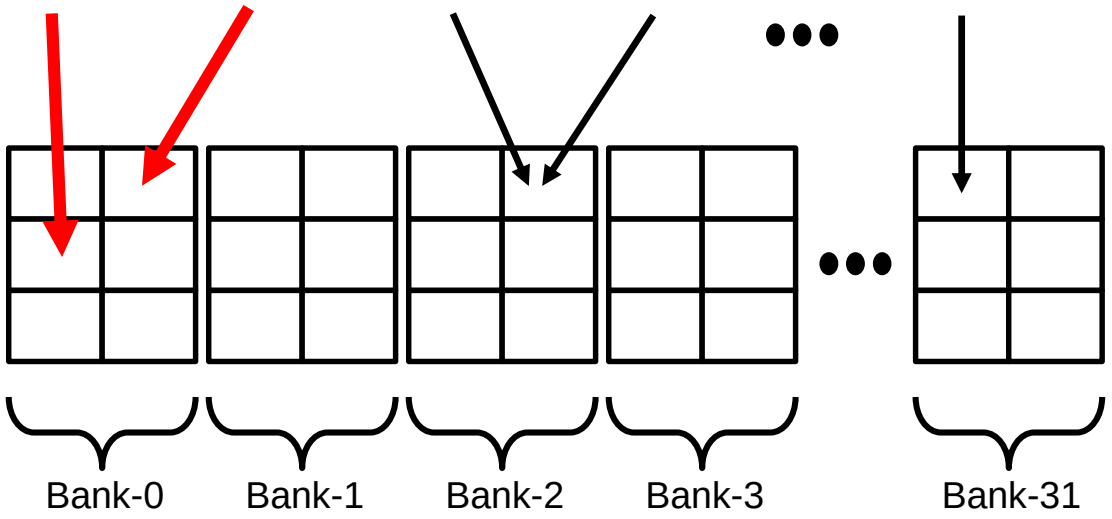
Multiple addresses per bank, but within the same word



SMEM Access Examples

Addresses from a warp: 2-way bank conflict

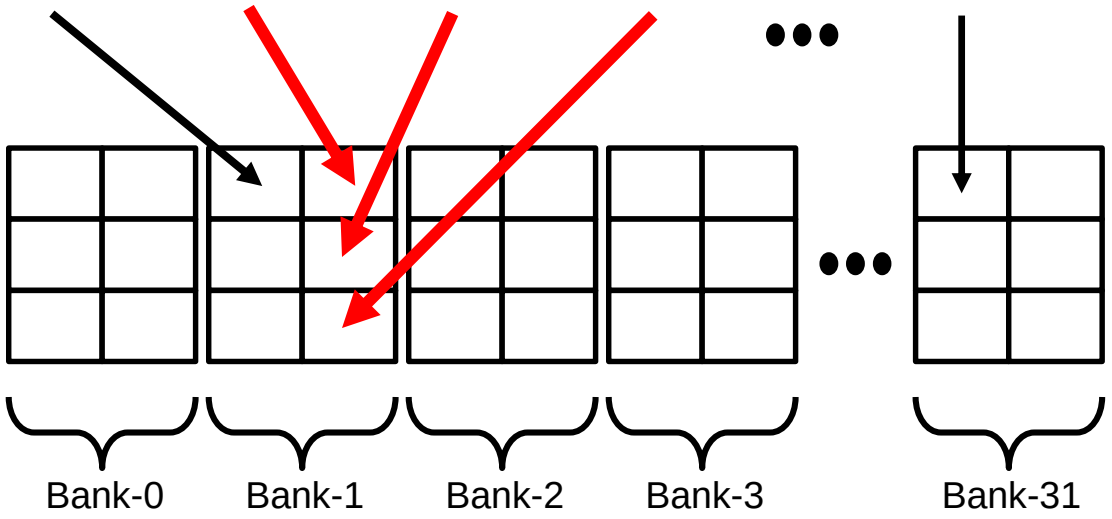
2 accesses per bank, fall in two different words



SMEM Access Examples

Addresses from a warp: 3-way bank conflict

4 accesses per bank, fall in 3 different words



Diagnosing Bank Conflicts

- Profiler counters:
  - Number of instructions executed, does not include replays:
    - shared_load, shared_store
  - Number of replays (number of instruction issues due to bank conflicts)
    - l1_shared_bank_conflict
- Analysis:
  - Number of replays per instruction
    - l1_shared_bank_conflict / (shared_load + shared_store)
  - Replays are potentially a concern because:
    - Replays add latency
    - Compete for issue cycles with other SMEM and GMEM operations
      - Except for read-only loads, which go to different hardware
- Remedy:
  - Usually padding SMEM data structures resolves/reduces bank conflicts

Summary: Shared Memory

- Shared memory is a tremendous resource
  - Very high bandwidth (terabytes per second)
  - 20-30x lower latency than accessing GMEM
  - Data is programmer-managed, no evictions by hardware
- Performance issues to look out for:
  - Bank conflicts add latency and reduce throughput
  - Many-way bank conflicts can be very expensive
    - Replay latency adds up
    - However, few code patterns have high conflicts, padding is a very simple and effective solution

CPU + GPGPU System Architecture Sharing Memory

Review- Typical Structure of a CUDA Program

- Global variables declaration
- Function prototypes
  - __global__ void kernelOne(…)
- Main ()
  - allocate memory space on the device – cudaMalloc(&d_GlblVarPtr, bytes)
  - transfer data from host to device – cudaMemCpy(d_GlblVarPtr, h_Gl…)
  - execution configuration setup
  - kernel call – kernelOne<<<execution configuration>>>(args…);
  - transfer results from device to host – cudaMemCpy(h_GlblVarPtr,…)
  - optional: compare against golden (host computed) solution

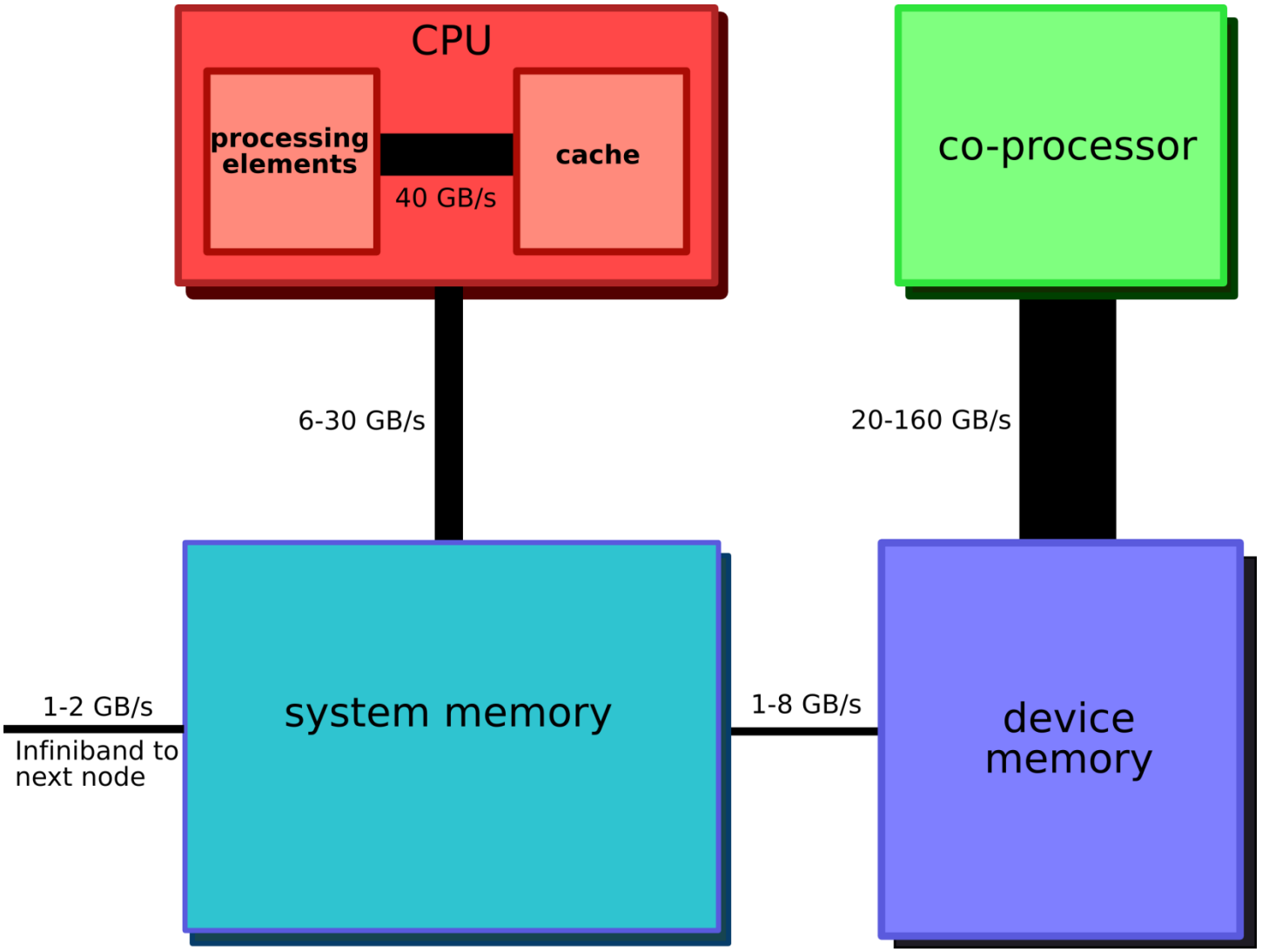
repeat as needed

- Kernel – void kernelOne(type args,…)
  - variables declaration - __local__, __shared__
    - automatic variables transparently assigned to registers or local memory
  - syncthreads()…

Bandwidth – Gravity of Modern Computer Systems

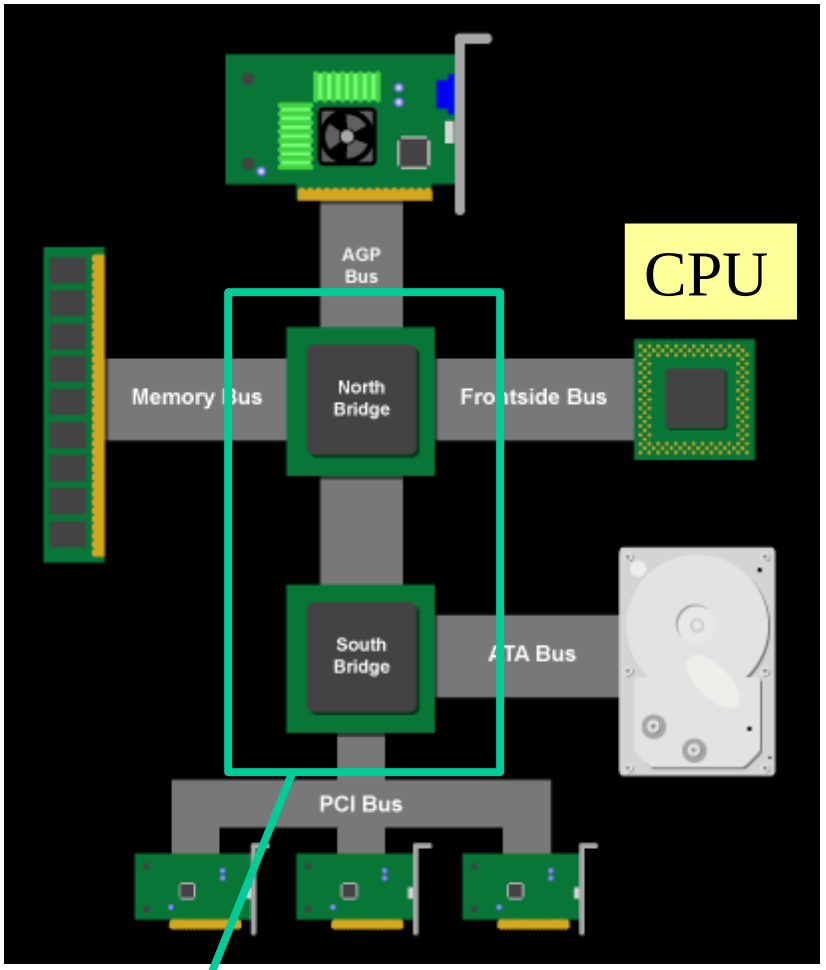
- The Bandwidth between key components ultimately dictates system performance
  - Especially true for massively parallel systems processing massive amount of data
  - Tricks like buffering, reordering, caching can temporarily defy the rules in some cases
  - Ultimately, the performance falls back to what the "speeds and feeds" dictate

Bandwidth in a CPU-GPU system



Classic PC architecture

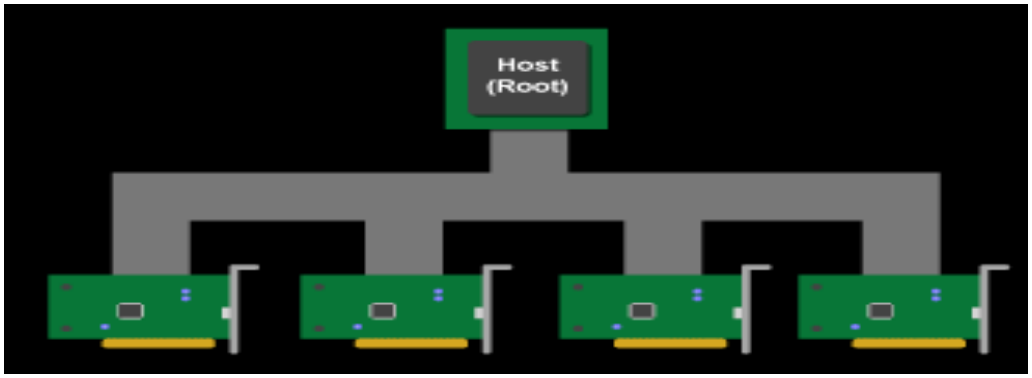
- Northbridge connects 3 components that must be communicate at high speed
  - CPU, DRAM, video
  - Video also needs to have 1st-class access to DRAM
  - Previous NVIDIA cards are connected to AGP, up to 2 GB/s transfers
- Southbridge serves as a concentrator for slower I/O devices



Core Logic Chipset

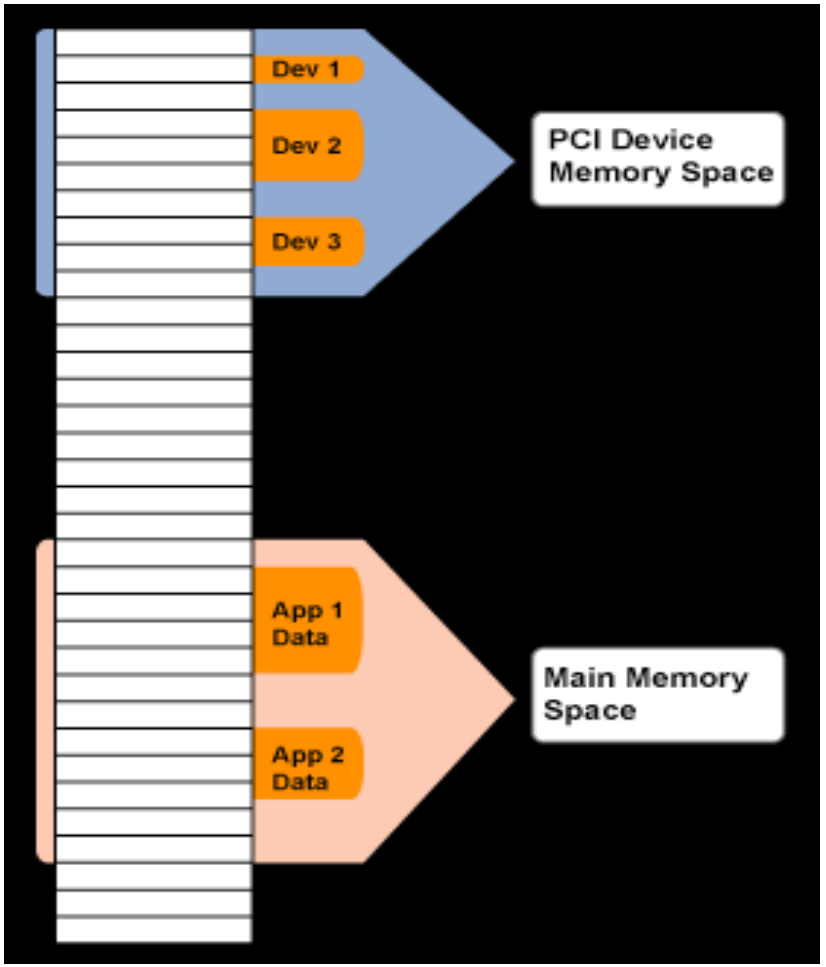
(Original) PCI Bus Specification

- Connected to the southBridge
  - Originally 33 MHz, 32-bit wide, 132 MB/second peak transfer rate
  - More recently 66 MHz, 64-bit, 528 MB/second peak
  - Upstream bandwidth remain slow for device (~256MB/s peak)
  - Shared bus with arbitration
    - Winner of arbitration becomes bus master and can connect to CPU or DRAM through the southbridge and northbridge



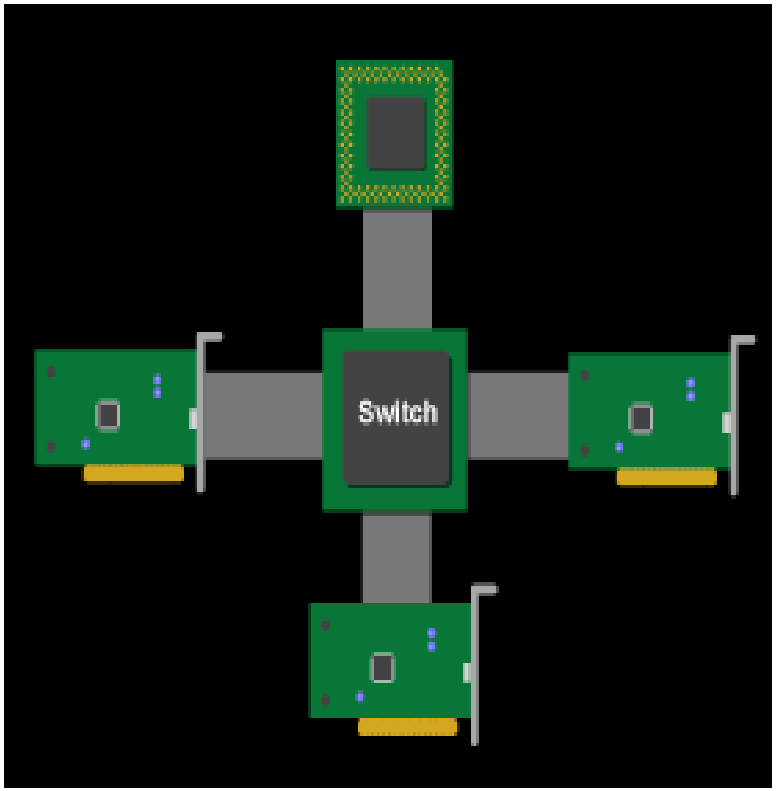
PCI as Memory Mapped I/O

- PCI device registers are mapped into the CPU's physical address space
  - Accessed through loads/ stores (kernel mode)
- Addresses are assigned to the PCI devices at boot time
  - All devices listen for their addresses



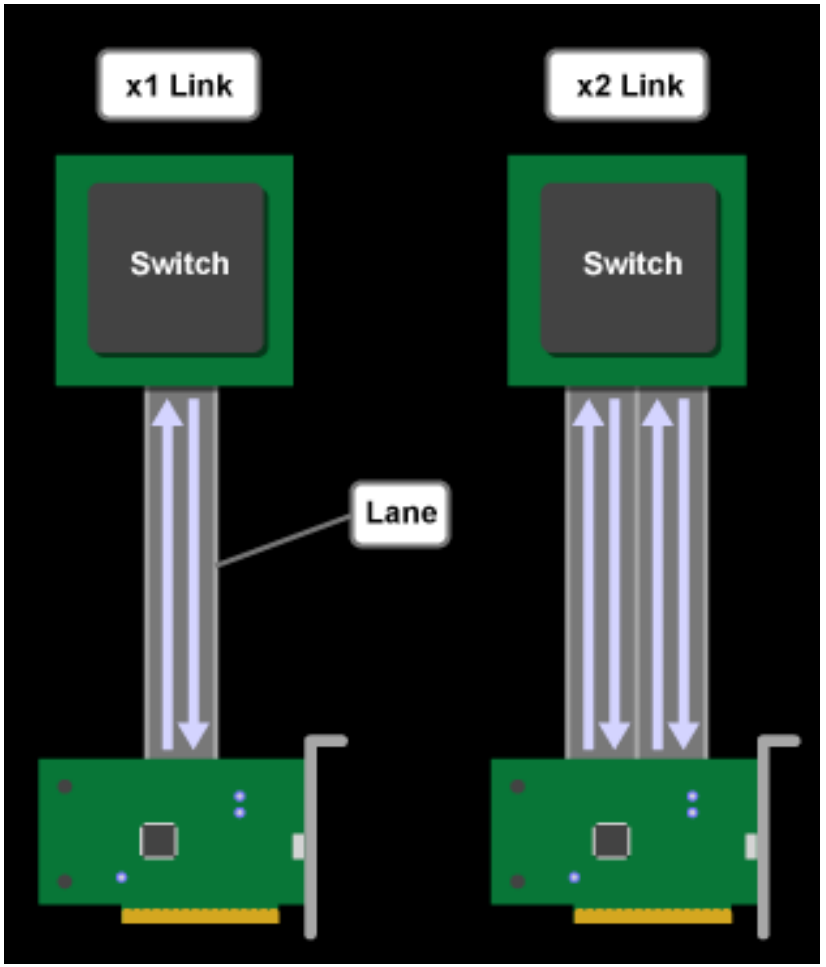
PCI Express (PCIe)

- Switched, point-to-point connection
  - Each card has a dedicated “link” to the central switch, no bus arbitration.
  - Packet switches messages form virtual channel
  - Prioritized packets for QoS
    - E.g., real-time video streaming



PCIe 2 Links and Lanes

- Each link consists of one or more lanes
  - Each lane is 1-bit wide (4 wires, each 2-wire pair can transmit 2.5Gb/s in one direction)
    - Upstream and downstream now simultaneous and symmetric
  - Each Link can combine 1, 2, 4, 8, 12, 16 lanes- x1, x2, etc.
  - Each byte data is **8b/10b** encoded into 10 bits with equal number of 1's and 0's; net data rate 2 Gb/s per lane each way.
  - Thus, the net data rates are 250 MB/s (x1) 500 MB/s (x2), 1GB/s (x4), 2 GB/s (x8), 4 GB/s (x16), each way

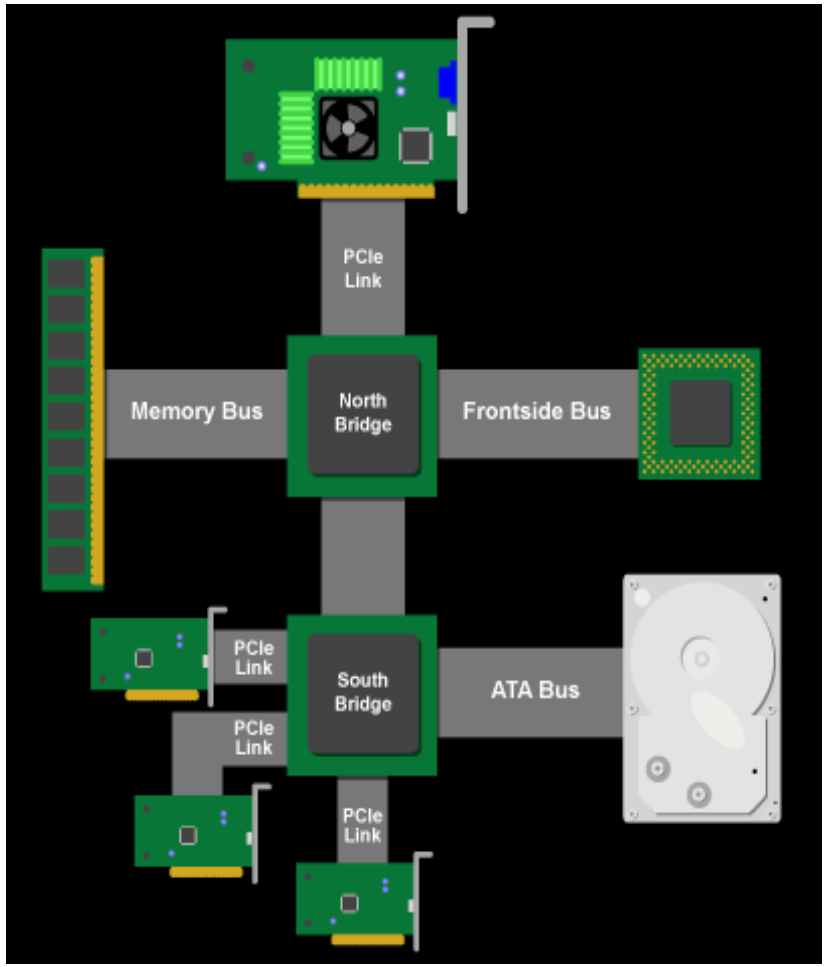


8/10 bit encoding

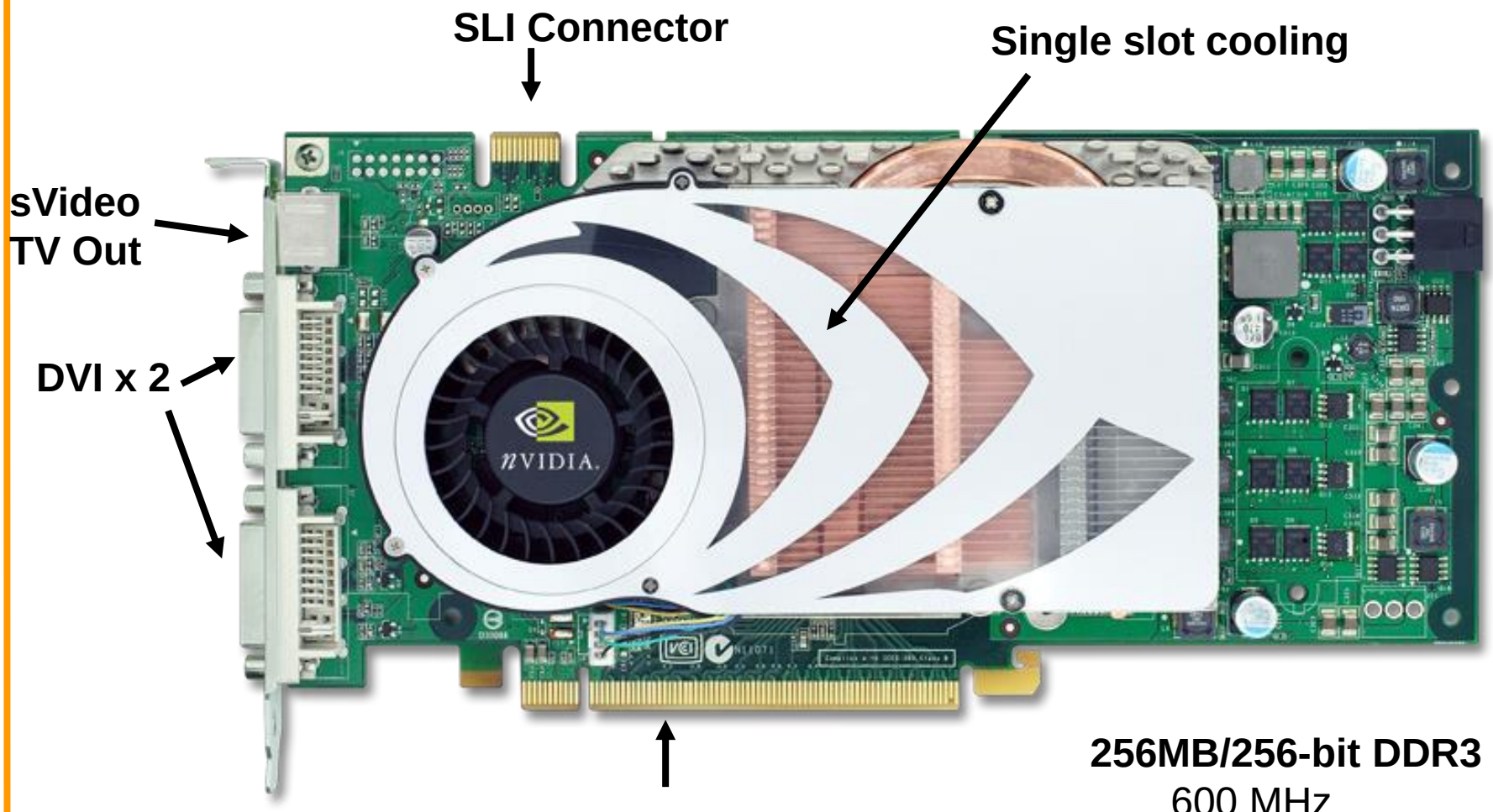
- Goal is to maintain DC balance while have sufficient state transition for clock recovery
- The difference of 1s and 0s in a 20-bit stream should be ≤ 2
- There should be no more than 5 consecutive 1s or 0s in any stream
- 00000000, 00000111, 11000001 bad
- 01010101, 11001100 good
- Find 256 good patterns among 1024 total patterns of 10 bits to encode an 8-bit data
- An 20% overhead

PCIe PC Architecture

- PCIe forms the interconnect backbone
  - Northbridge/Southbridge are both PCIe switches
  - Some Southbridge designs have built-in PCI-PCIe bridge to allow old PCI cards
  - Some PCIe I/O cards are PCI cards with a PCI-PCIe bridge
- Source: Jon Stokes, PCI Express: An Overview
  - http://arstechnica.com/articles/paedia/hardware/pcie.ars



GeForce 7800 GTX Board Details

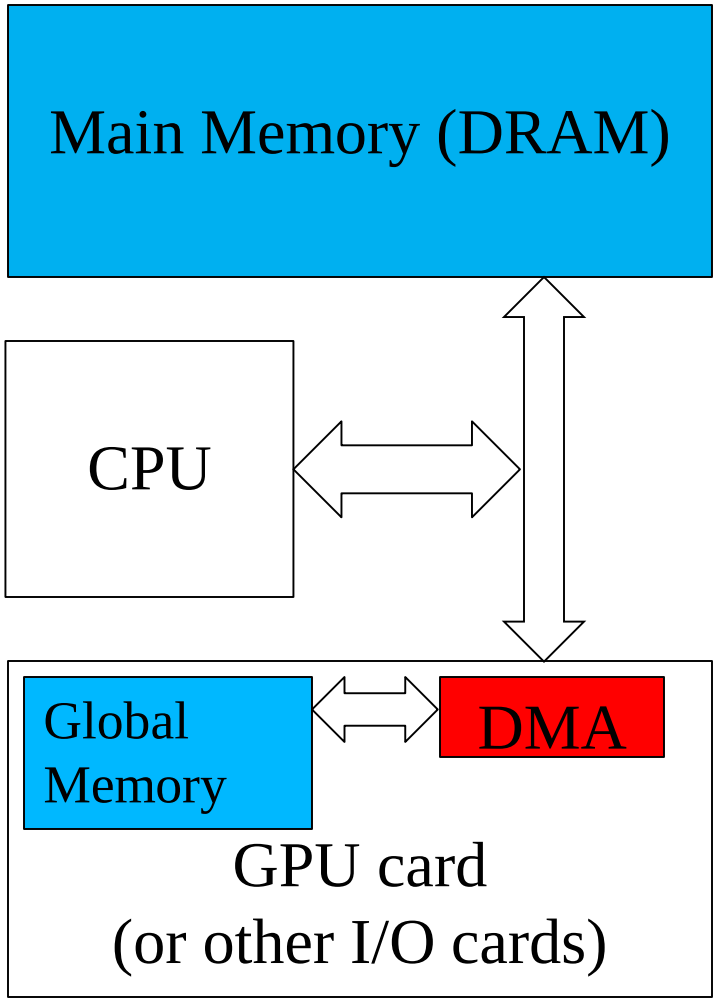


PCIe 3

- A total of 8 Giga Transfers per second in each direction
- No more 8/10 encoding but uses a polynomial transformation at the transmitter and its inverse at the receiver to achieve the same effect
- So the effective bandwidth is (almost) double of PCIe 2: 985MB/s – per lane

PCIe Data Transfer using DMA

- DMA (Direct Memory Access) is used to fully utilize the bandwidth of an I/O bus
  - DMA uses physical address for source and destination
  - Transfers a number of bytes requested by OS
  - Needs pinned memory



Pinned Memory

- DMA uses physical addresses
- The OS could accidentally page out the data that is being read or written by a DMA and page in another virtual page into the same location
- Pinned memory cannot not be paged out
- If a source or destination of a cudaMemCpy() in the host memory is not pinned, it needs to be first copied to a pinned memory – extra overhead
- cudaMemcpy is much faster with pinned host memory source or destination

Allocate/Free Pinned Memory (a.k.a. Page Locked Memory)

- cudaHostAlloc()
  - Three parameters
  - Address of pointer to the allocated memory
  - Size of the allocated memory in bytes
  - Option – use cudaHostAllocDefault for now

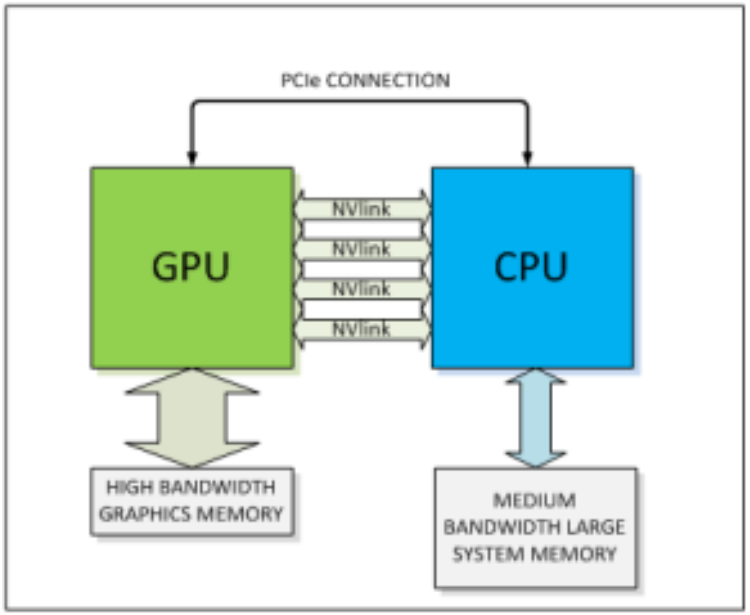
- cudaFreeHost()
  - One parameter
  - Pointer to the memory to be freed

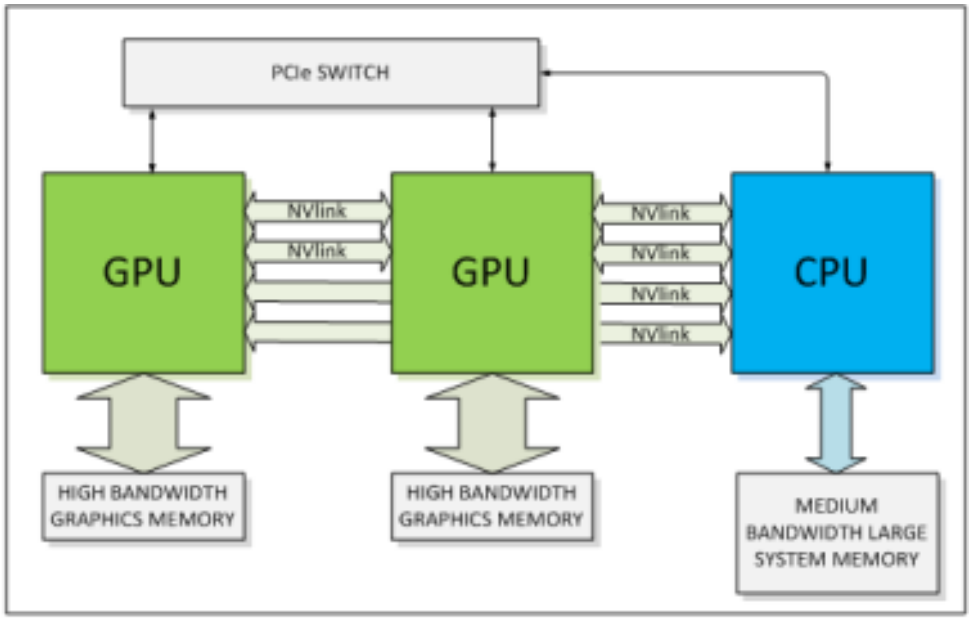
Using Pinned Memory

- Use the allocated memory and its pointer the same way those returned by malloc();
- The only difference is that the allocated memory cannot be paged by the OS
- The cudaMemcpy function should be about 2X faster with pinned memory
- Pinned memory is a limited resource whose over-subscription can have serious consequences

Next-gen technologies: NVLINK

16GB/sec (16-lanes) made available by PCI-Express 3.0 is hardly adequate vs. 250GB/sec+ of memory bandwidth available within a single card.



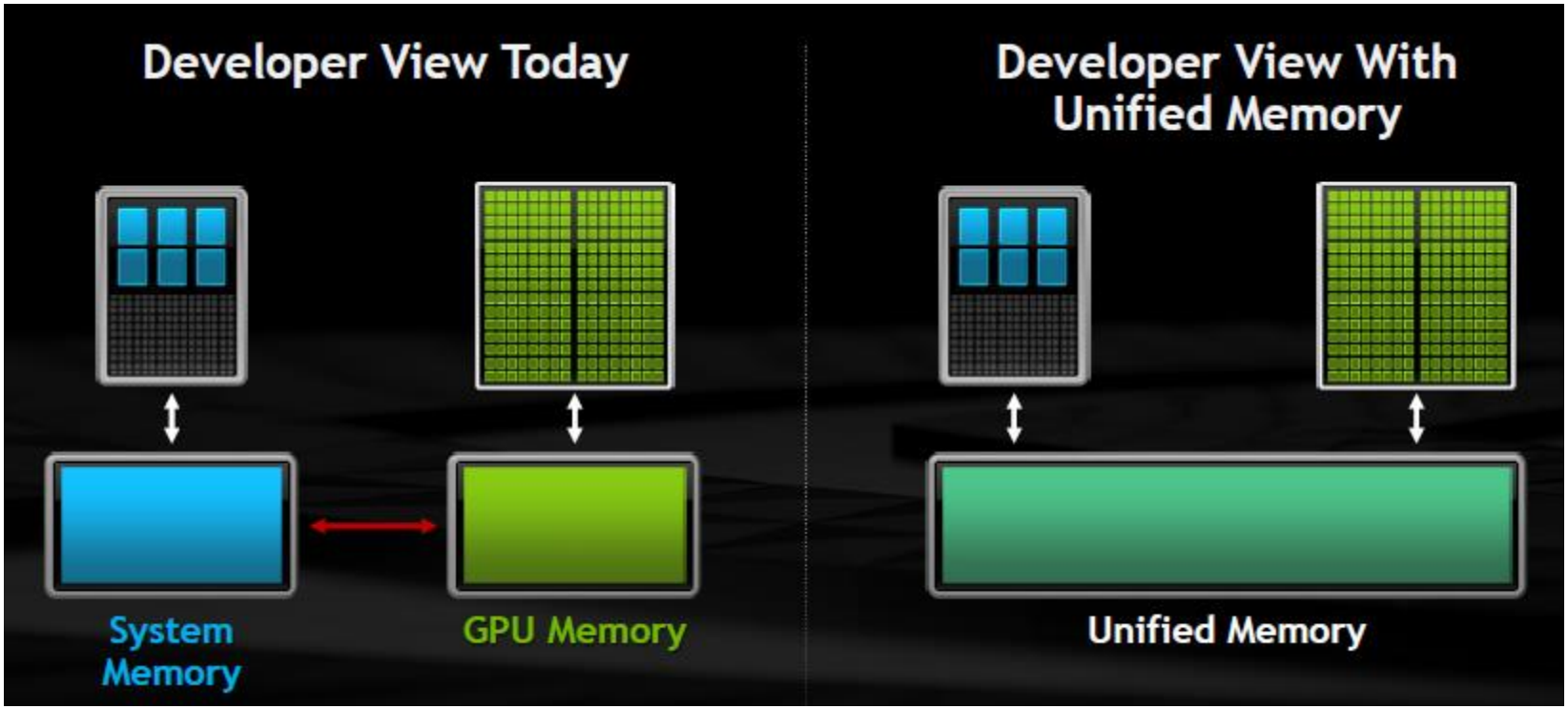


NVLink uses differential signaling (like PCIe), with the smallest unit of connectivity being a “block.”
A block contains 8 lanes, each rated for 20Gbps, for a combined bandwidth of 20GB/sec. → 20 vs. 8GT/sec for PCIe 3.0

Assumptions	NVLink	PCIe Gen3
Connection Type	4 connections	16 lanes
Peak Bandwidth	80 GB/s	16 GB/s
Effective Bandwidth	64 GB/s	12 GB/s

NVIDIA targets x5-x12 wrt PCIe

Moving towards Unified Memory



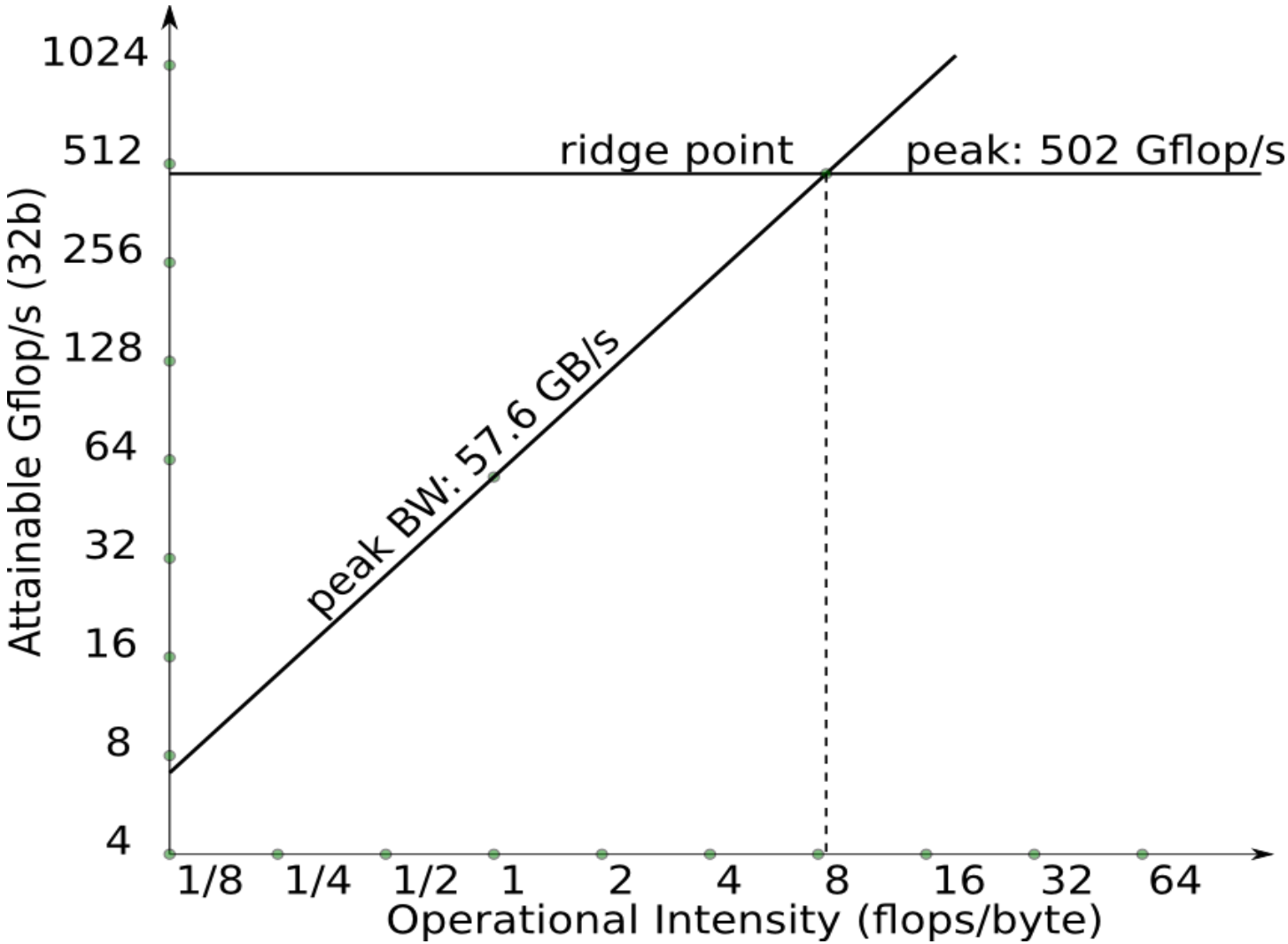
Backup Slides

Performance Modeling

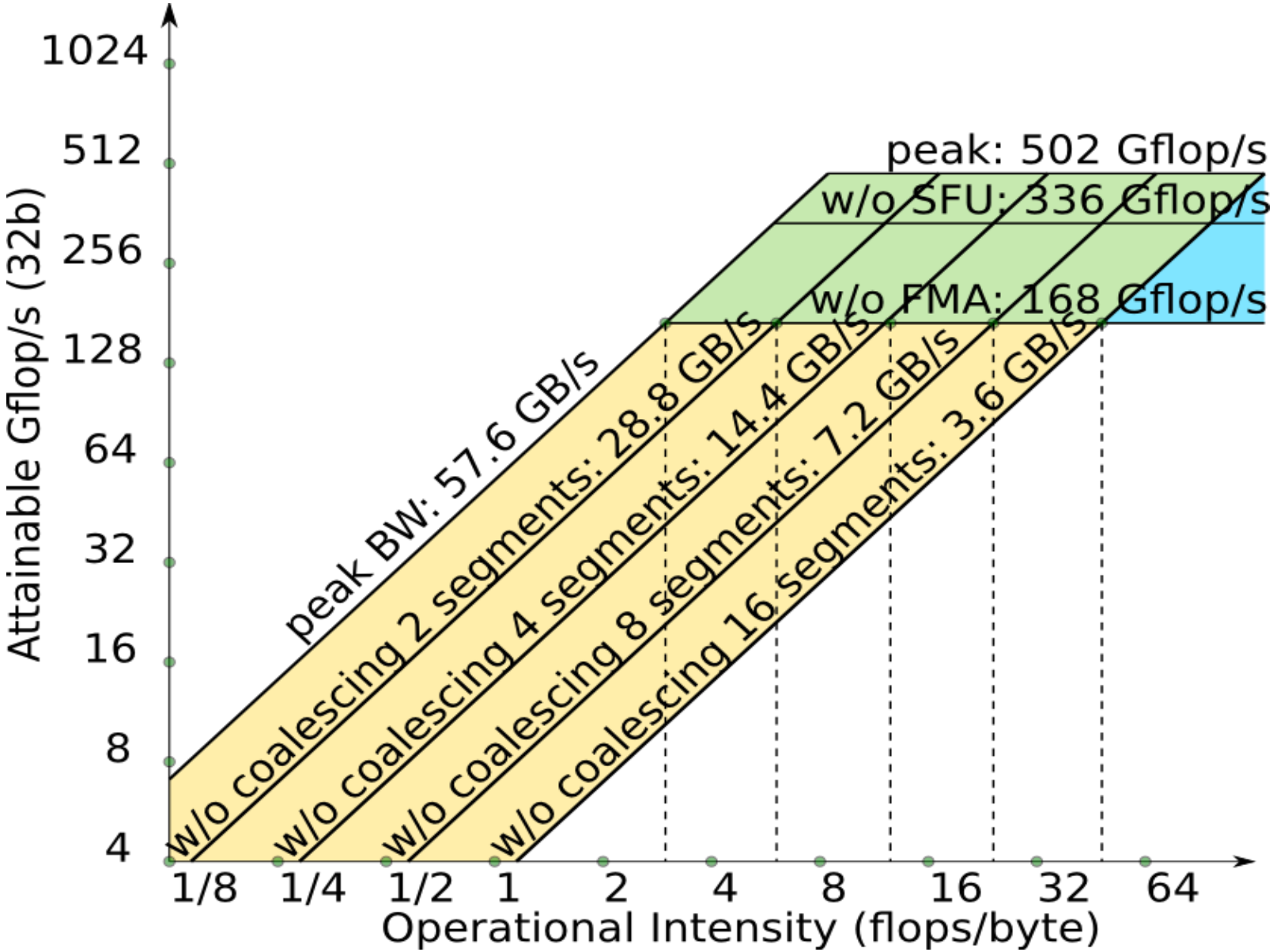
What is the bottleneck?

Roofline Model

Identify performance bottleneck:
computation bound v.s. bandwidth bound

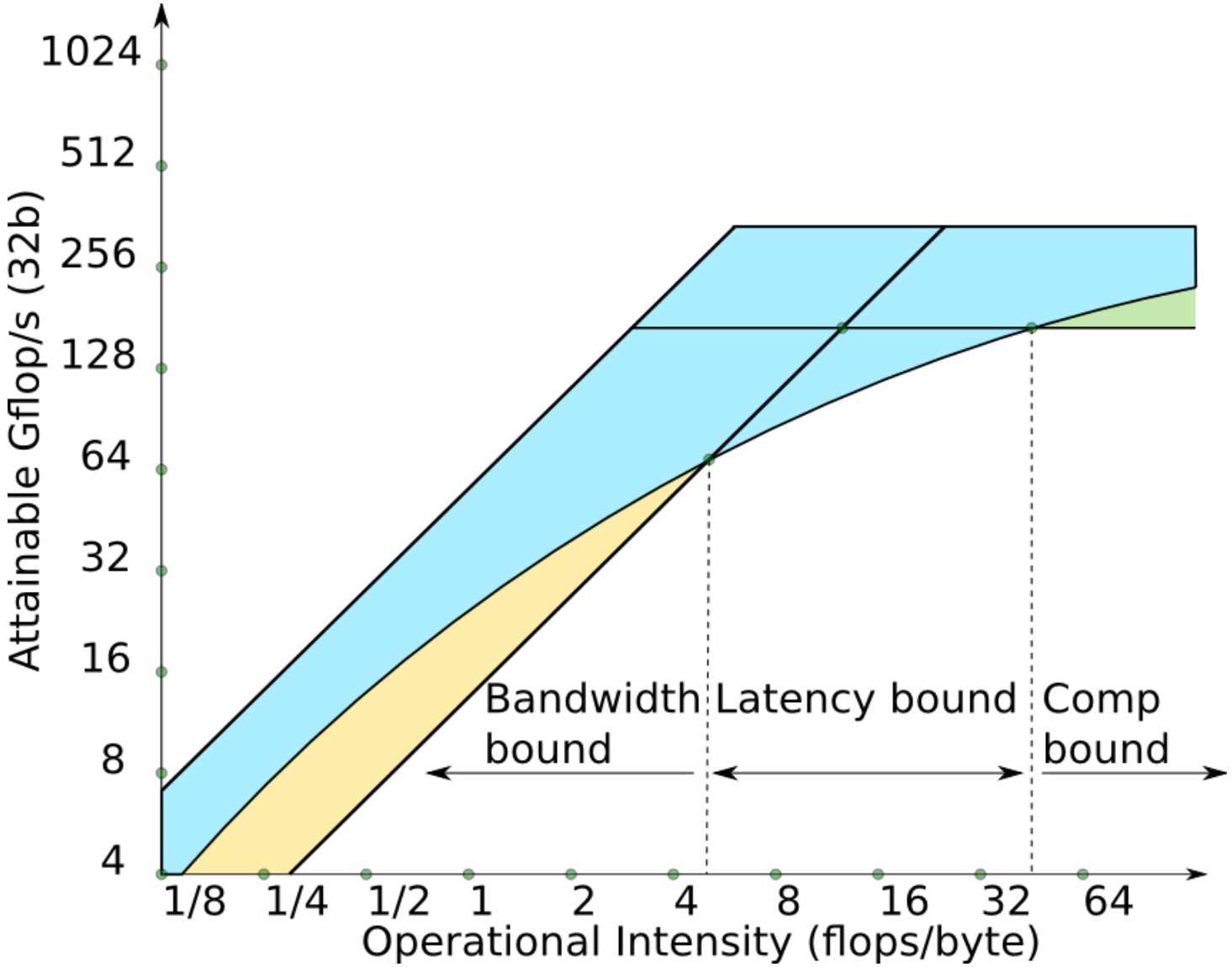


Optimization Is Key for Attainable Gflops/s



Computation, Bandwidth, Latency

Illustrating three bottlenecks in the Roofline model.



Program Optimization

If I know the bottleneck, how to optimize?

Optimization techniques for NVIDIA GPUs:
"CUDA C Best Practices Guide"
http://docs.nvidia.com/cuda/

Recap: What We have Learned

- GPU architecture (what is the difference?)
- GPU programming (why threads?)
- Performance analysis (bottlenecks)

Further Reading

GPU architecture and programming:

- NVIDIA Tesla: A unified graphics and computing architecture, in IEEE Micro 2008. (http://dx.doi.org/10.1109/MM.2008.31)
- Scalable Parallel Programming with CUDA, in ACM Queue 2008. (http://dx.doi.org/10.1145/1365490.1365500)
- Understanding throughput-oriented architectures, in Communications of the ACM 2010. (http://dx.doi.org/10.1145/1839676.1839694)

Recommended Reading

Performance modeling:

- Roofline: an insightful visual performance model for multicore architectures, in Communications of the ACM 2009. (http://dx.doi.org/10.1145/1498765.1498785)

Optional (if you are interested)

How branches are executed in GPUs:

- Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware, in ACM Transactions on Architecture and Code Optimization 2009. (http://dx.doi.org/10.1145/1543753.1543756)

Common Optimization Techniques

Optimizations on memory latency tolerance
- Reduce register pressure
- Reduce shared memory pressure

Optimizations on memory bandwidth
- Global memory coalesce
- Avoid shared memory bank conflicts
- Grouping byte access
- Avoid Partition camping

Optimizations on computation efficiency
- Mul/Add balancing
- Increase floating point proportion

Optimizations on operational intensity
- Use tiled algorithm
- Tuning thread granularity

Common Optimization Techniques

Optimizations on memory latency tolerance
- Reduce register pressure
- Reduce shared memory pressure

Optimizations on memory bandwidth
- Global memory coalesce
- Avoid shared memory bank conflicts
- Grouping byte access
- Avoid Partition camping

Optimizations on computation efficiency
- Mul/Add balancing
- Increase floating point proportion

Optimizations on operational intensity
- Use tiled algorithm
- Tuning thread granularity

Multiple Levels of Memory Hierarchy

Name	Cache?	Latency (cycle)	Read-only?
Global	L1/L2	200~400 (cache miss)	R/W
Shared	No	1~3	R/W
Constant	Yes	1~3	Read-only
Texture	Yes	~100	Read-only
Local	L1/L2	200~400 (cache miss)	R/W