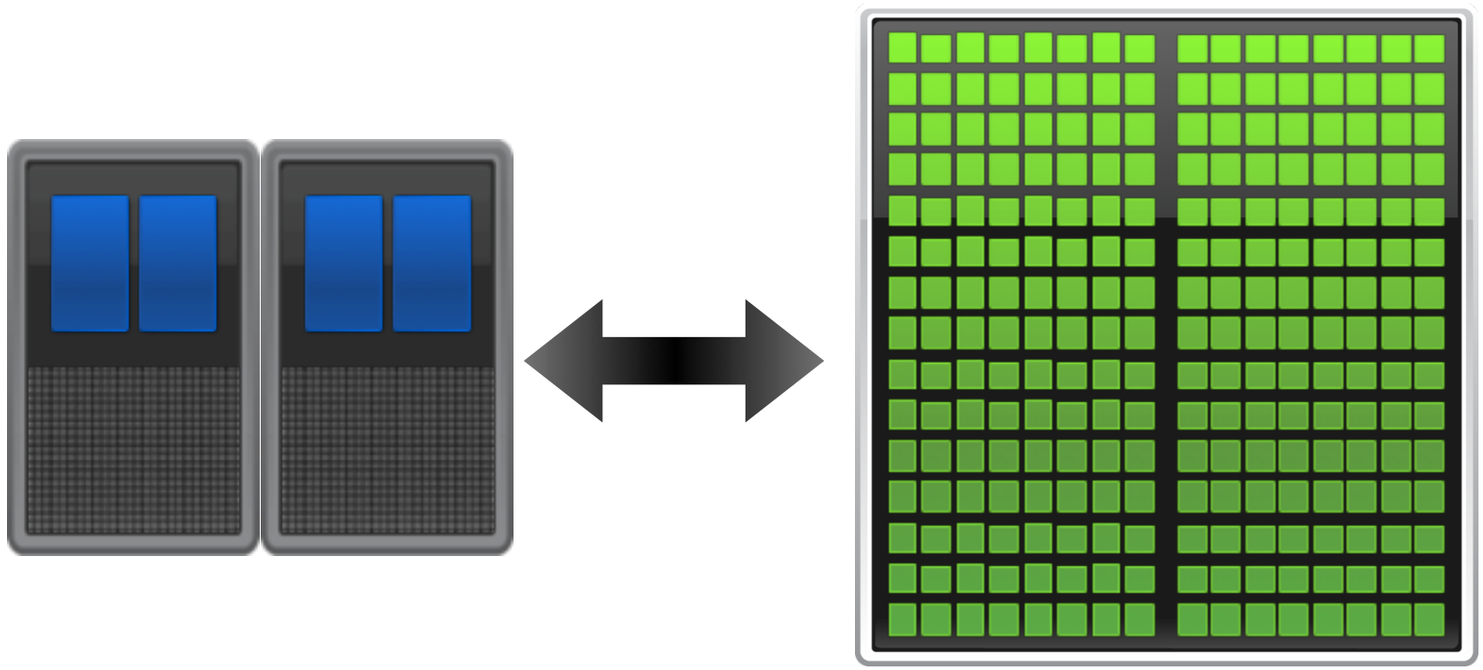
An Introduction to GPGPU Programming

Amir Gholami
PostDoc, BAIR
amirgh@berkeley.edu

Outline

- Throughput vs Latency
- From Graphics Processing Units to GPGPUs
- Introduction to CUDA
- Programming for Performance
- OpenCL

Heterogeneous Parallel Computing



Latency Optimized CPU

Fast Serial Processing

Throughput Optimized GPU

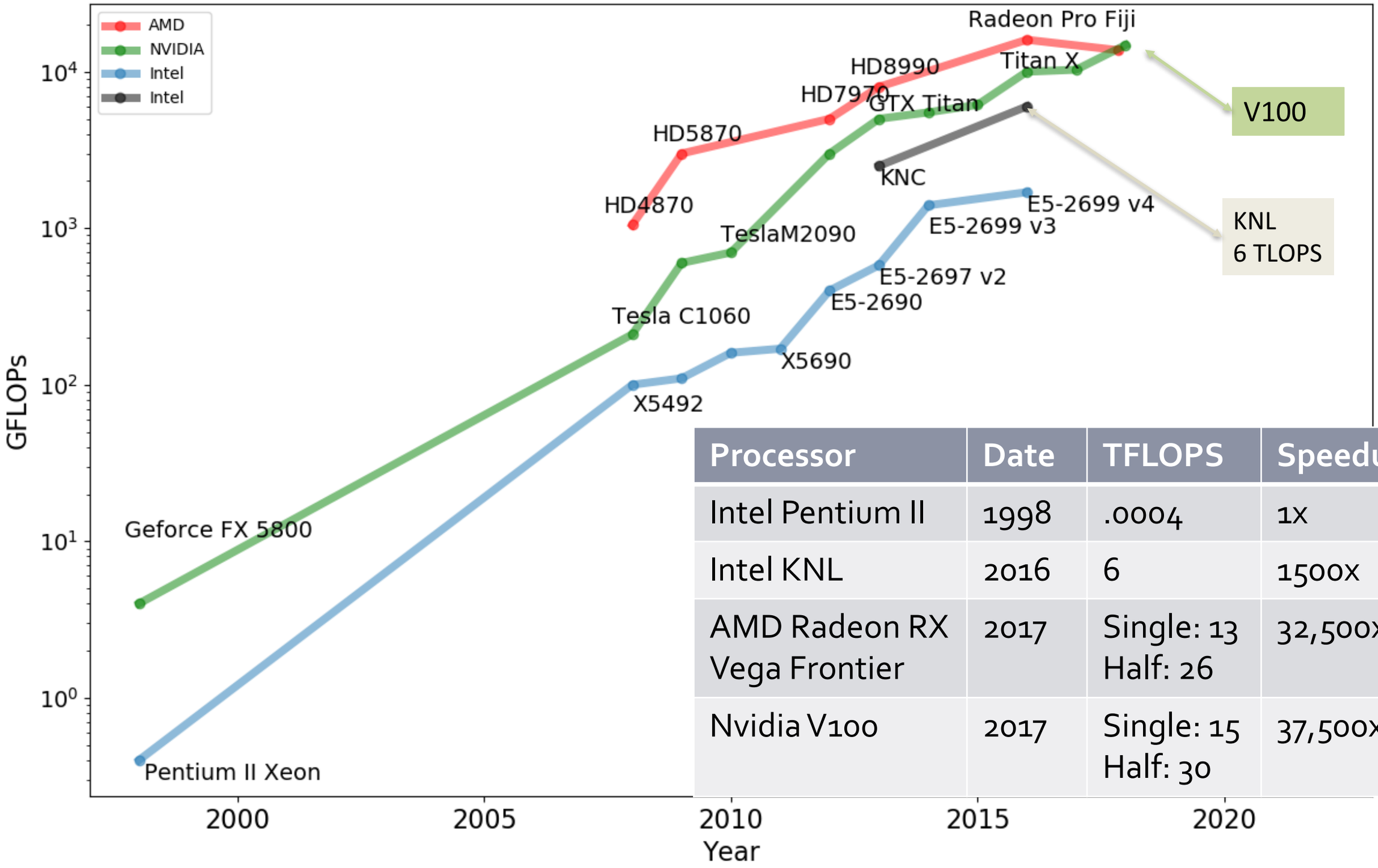
Scalable Parallel Processing

Slide from, Bryan Catanzaro, CS267, 2015

Was switching to GPGPUs obvious?

Was switching to GPGPUs obvious?

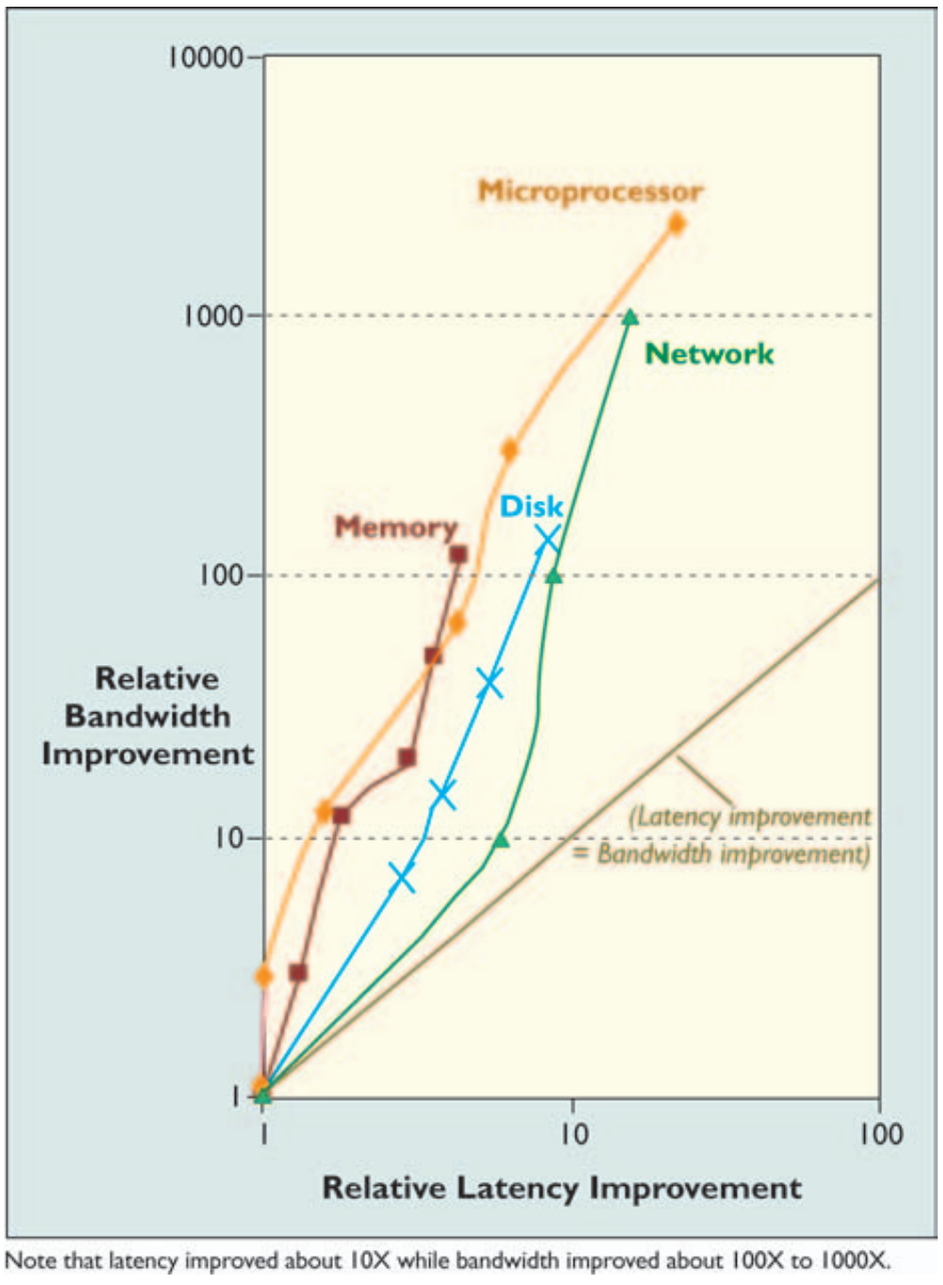
Past 20 Years



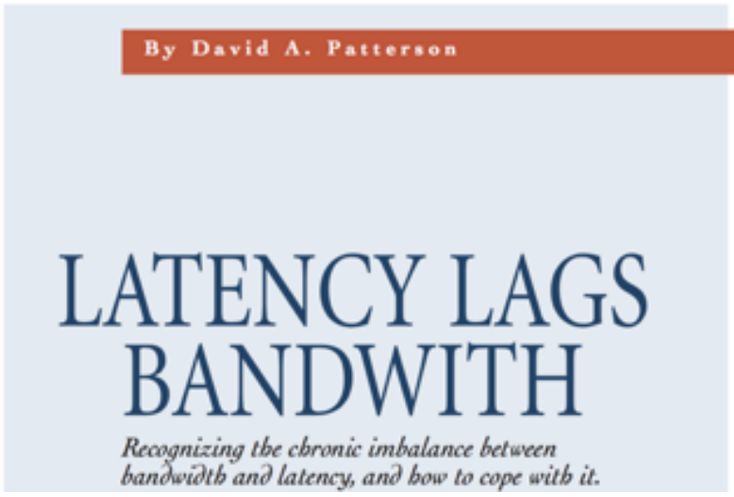
Processor	Date	TFLOPS	Speedup
Intel Pentium II	1998	.0004	1x
Intel KNL	2016	6	1500x
AMD Radeon RX Vega Frontier	2017	Single: 13 Half: 26	32,500x
Nvidia V100	2017	Single: 15 Half: 30	37,500x

Latency vs Bandwidth

D. Patterson, CACM October 2004



Note that latency improved about 10X while bandwidth improved about 100X to 1000X.


By David A. Patterson

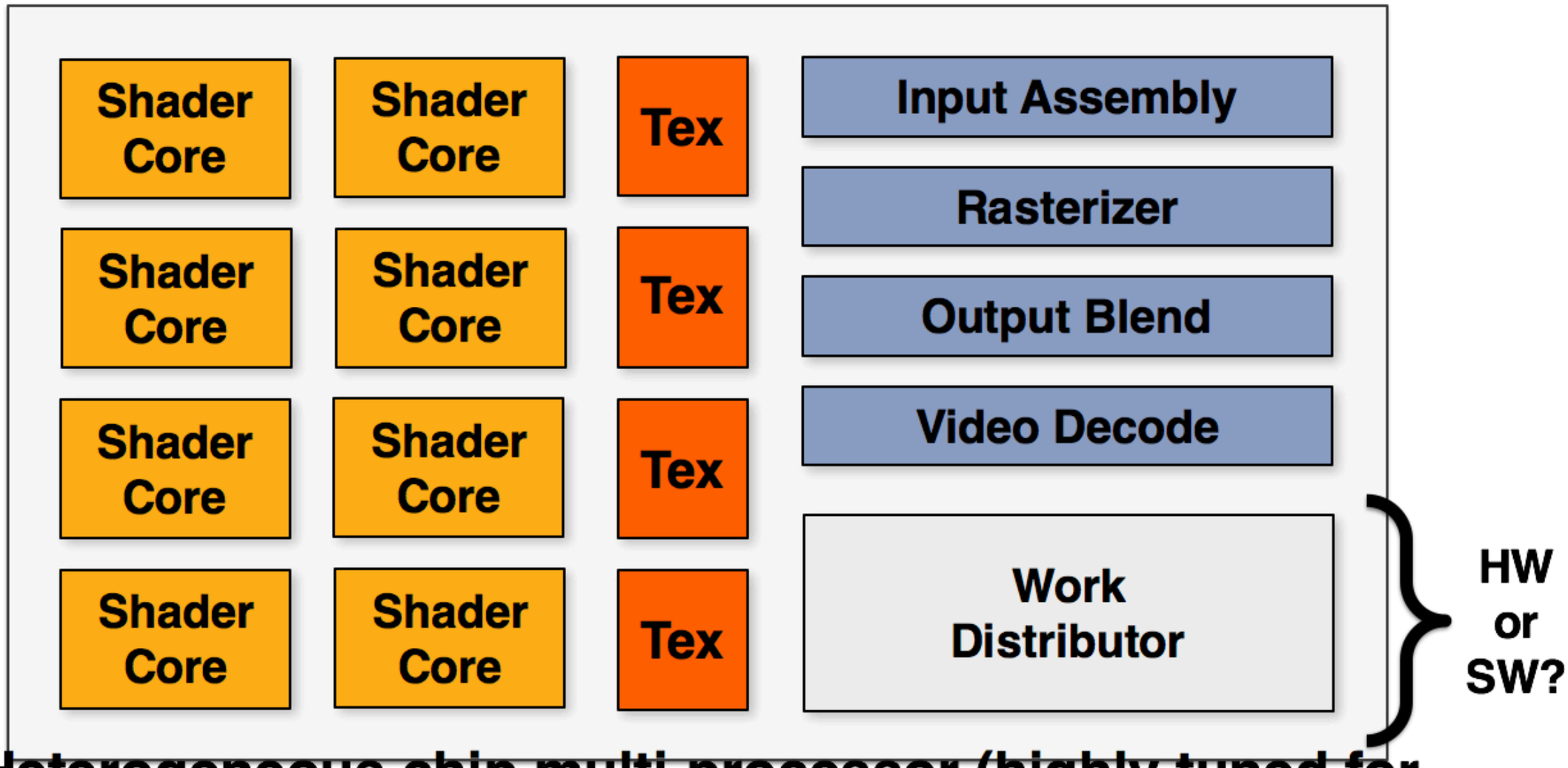
LATENCY LAGS BANDWITH

Recognizing the chronic imbalance between bandwidth and latency, and how to cope with it.

"There is an old network saying: Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed – you can't bribe God!

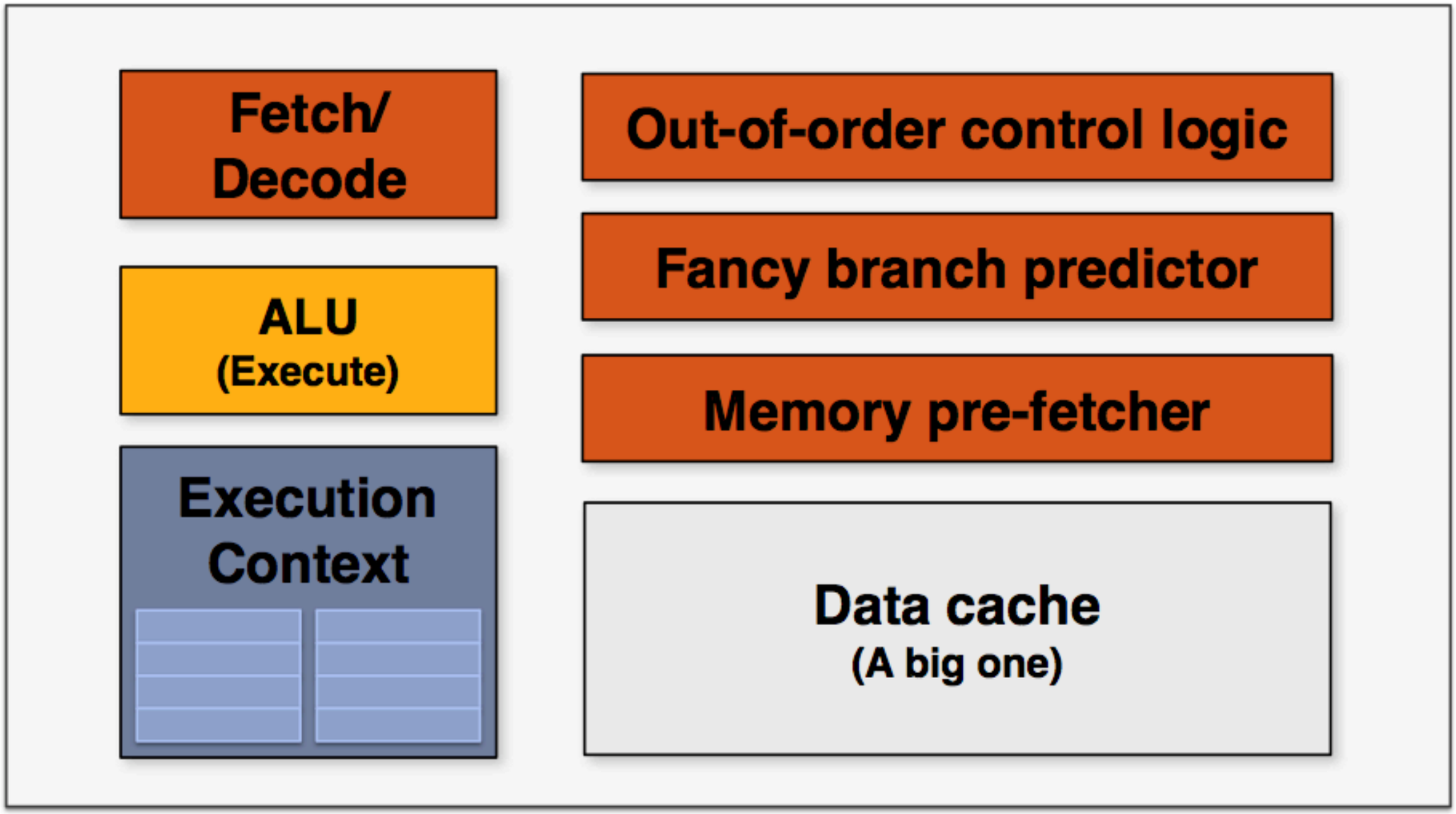
-Anonymous

Graphics Processing Units



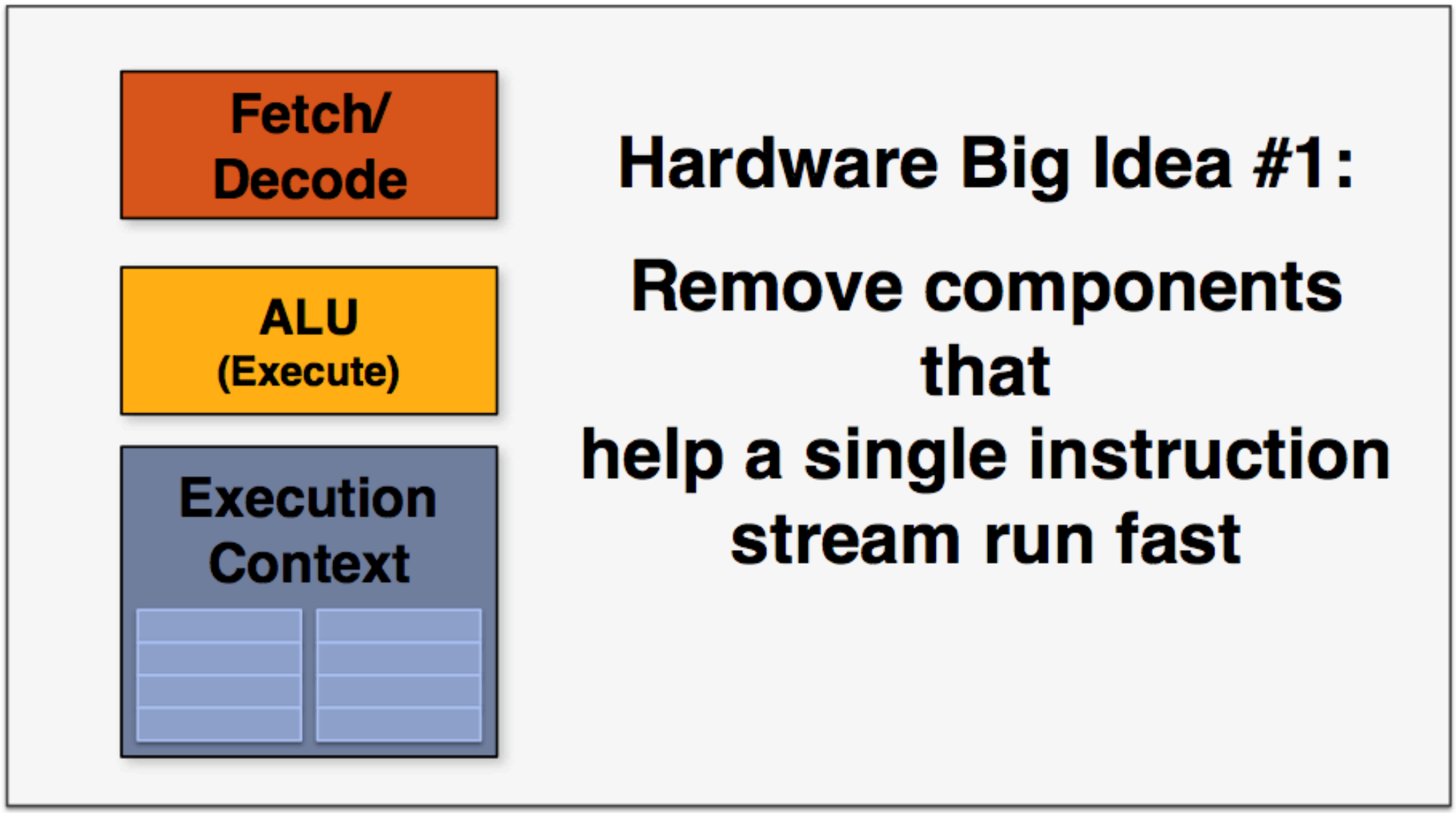
Heterogeneous chip multi-processor (highly tuned for graphics)

A typical CPU



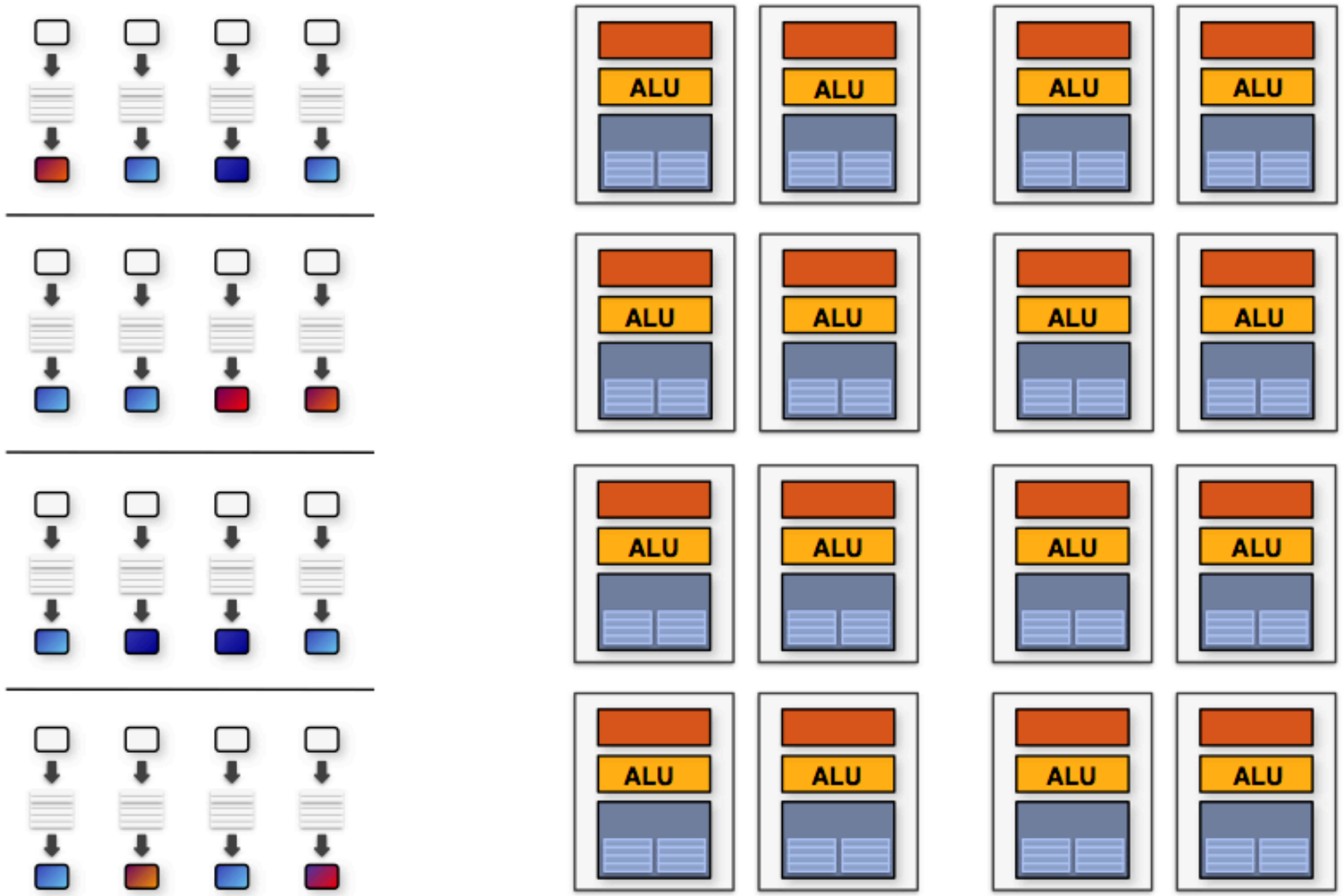
Slide from John Owens, CS267, 2017

CPU to GPGPU



Slide from John Owens, CS267, 2017

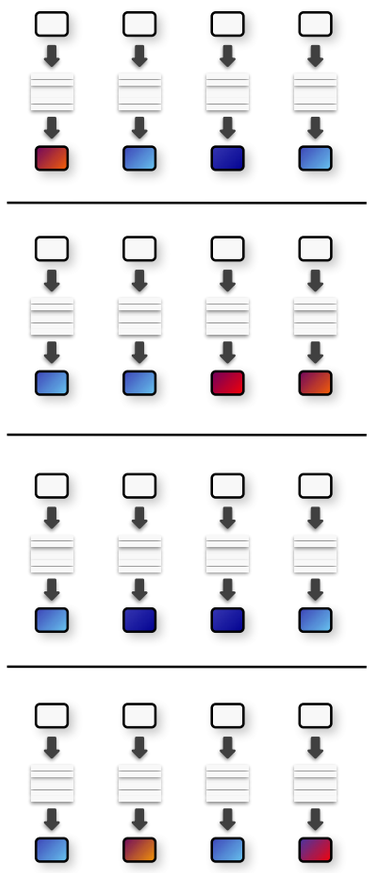
CPU to GPGPUs



Slide from John Owens, CS267, 2017

CPU to GPUs

Instruction stream sharing

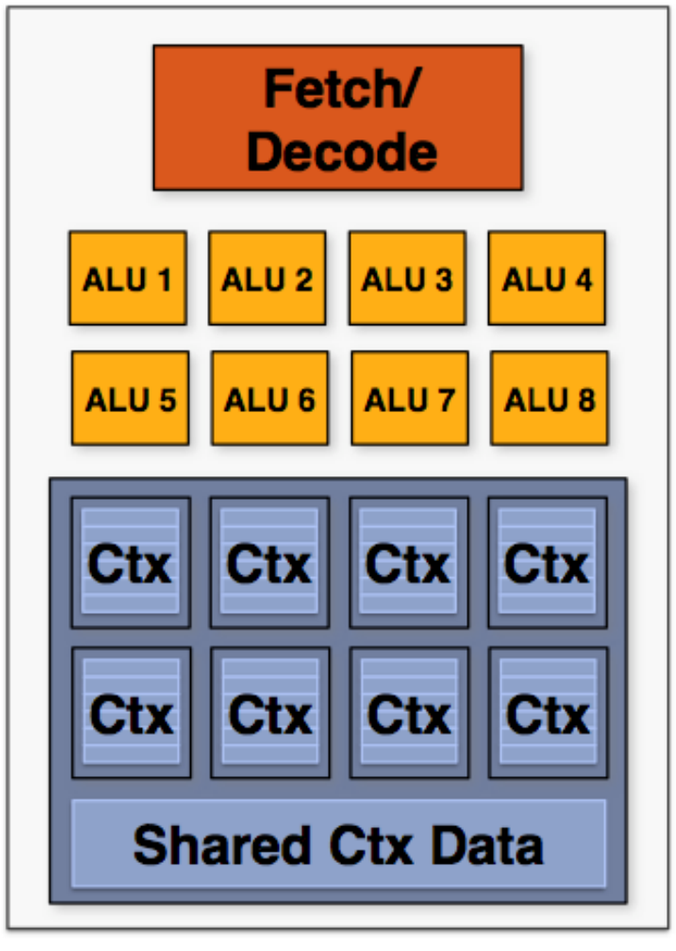


But… many threads *should* be able to share an instruction stream!

```
<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)
```


Slide from John Owens, CS267, 2017

CPU to GPGPUs

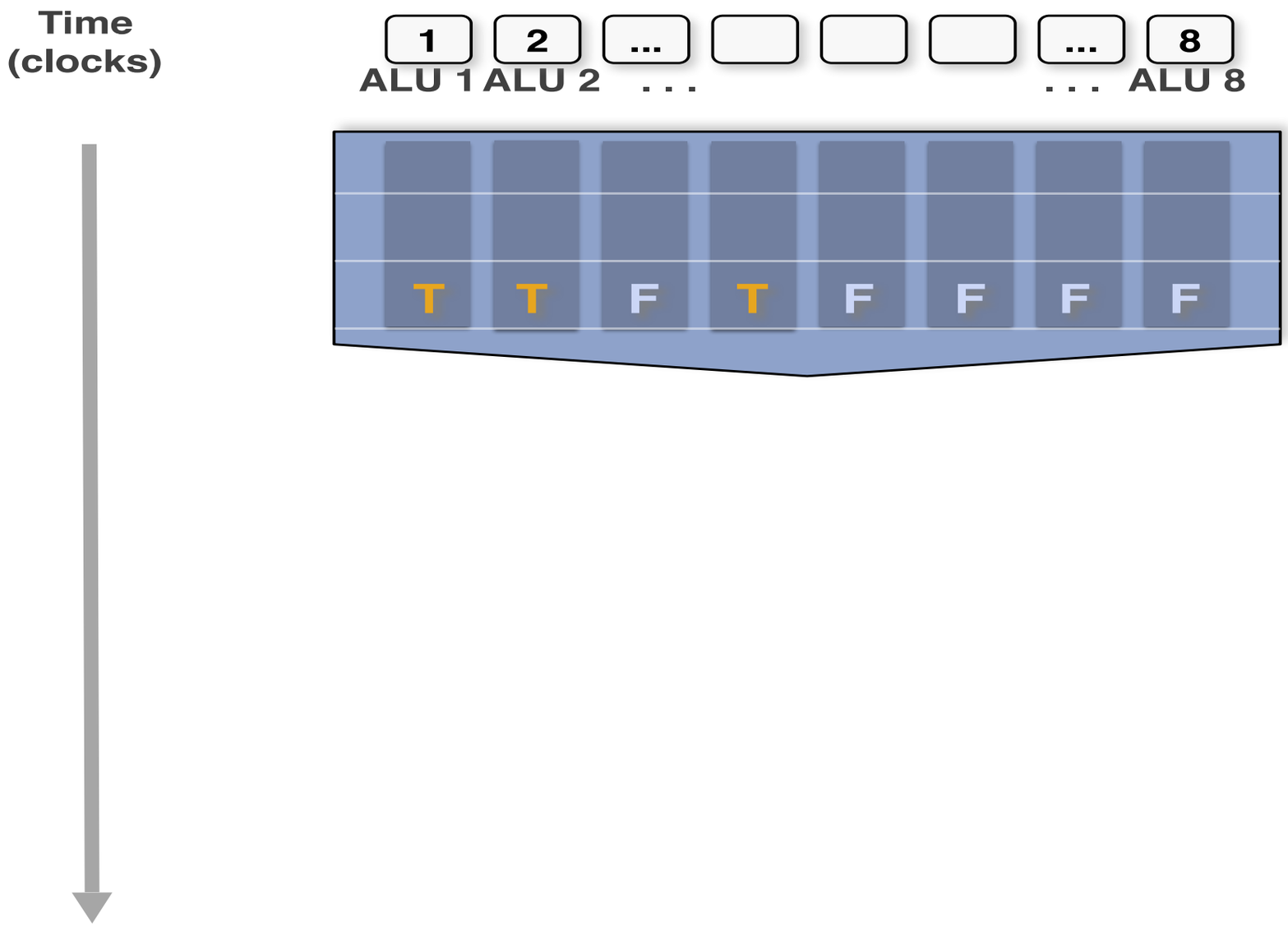


Big Idea #2:
Amortize cost of managing instruction stream across many ALUs

SIMT Processing

Thread Divergence?

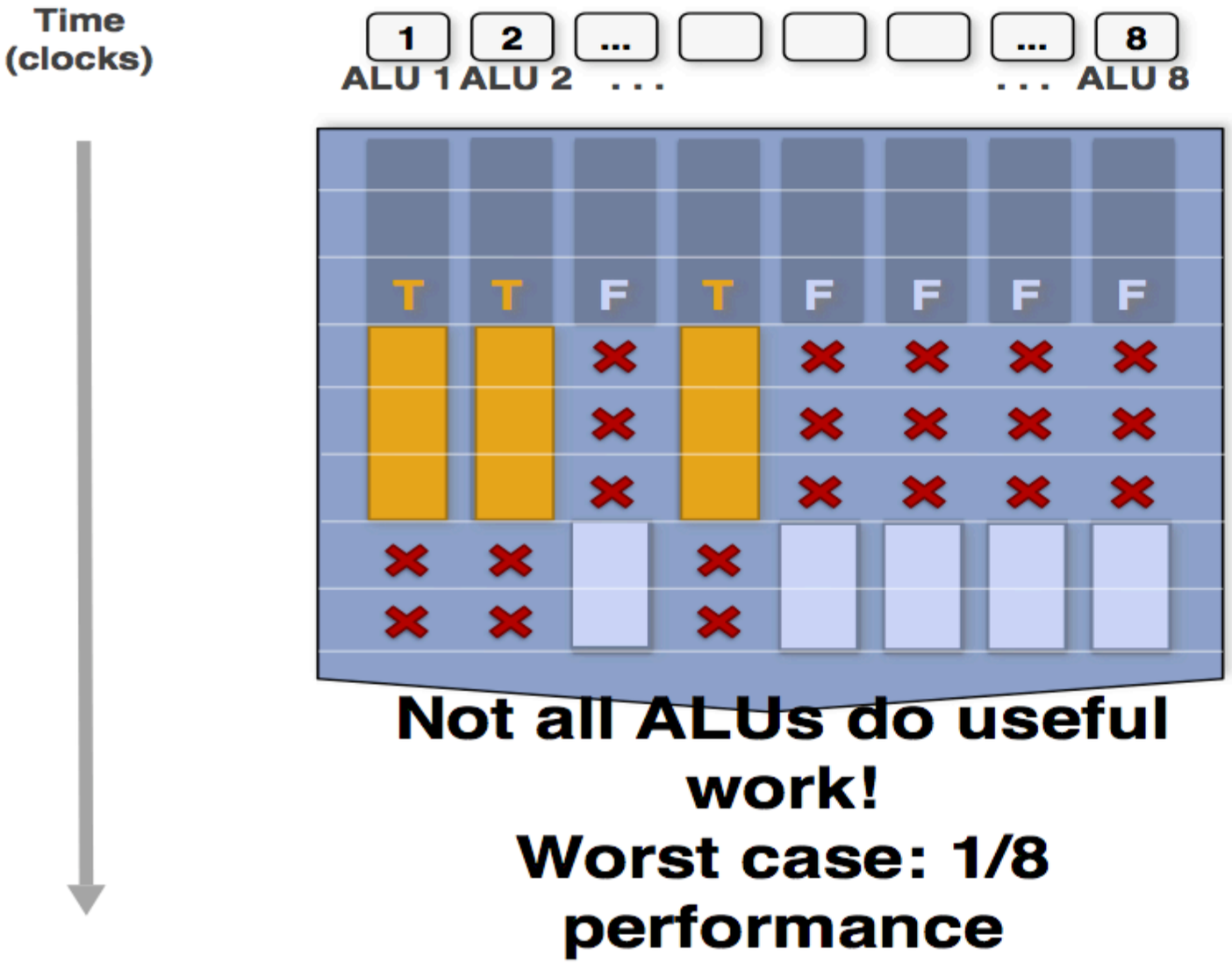
But what about branches?

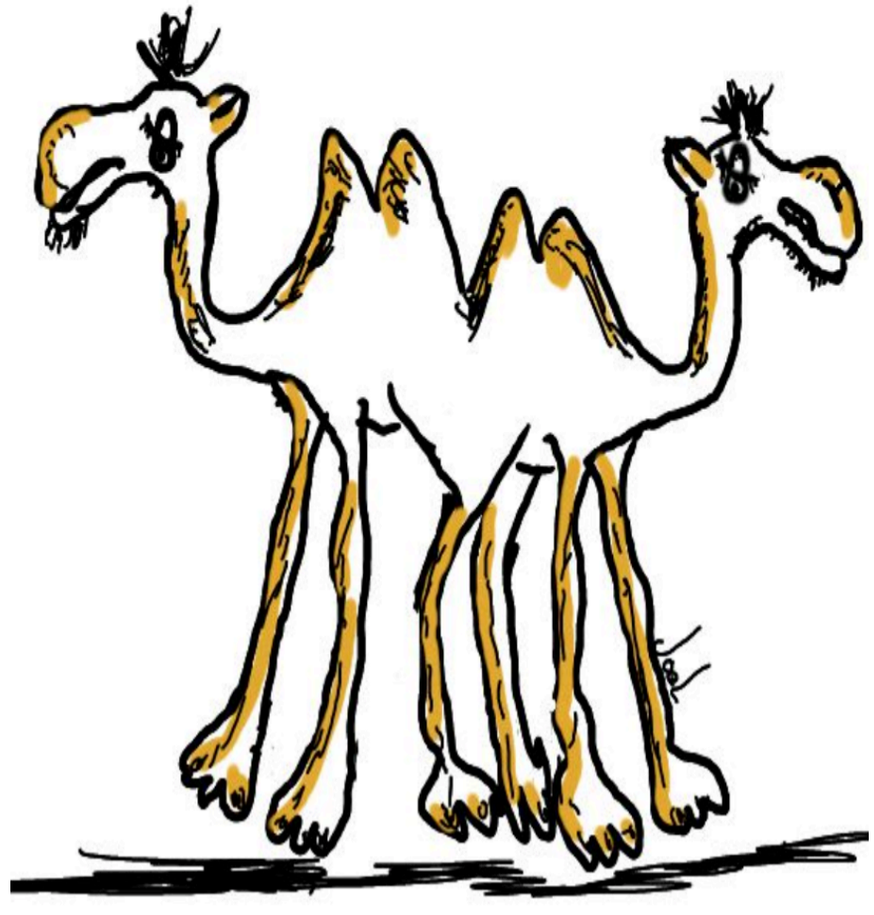



```
<unconditional
program code>

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

<resume
unconditional program
code>
```

Slide from John Owens, CS267, 2017

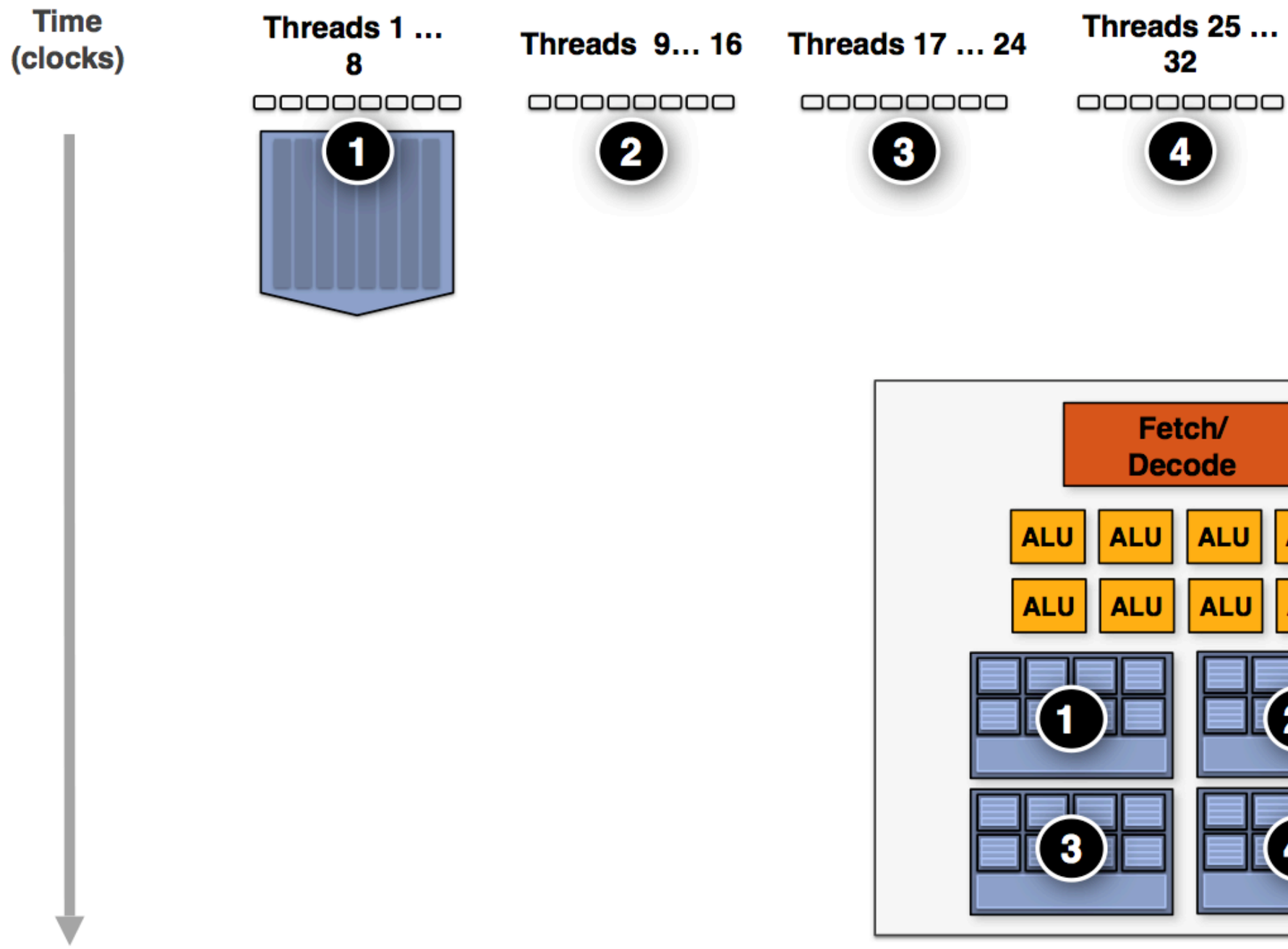
Thread Divergence?





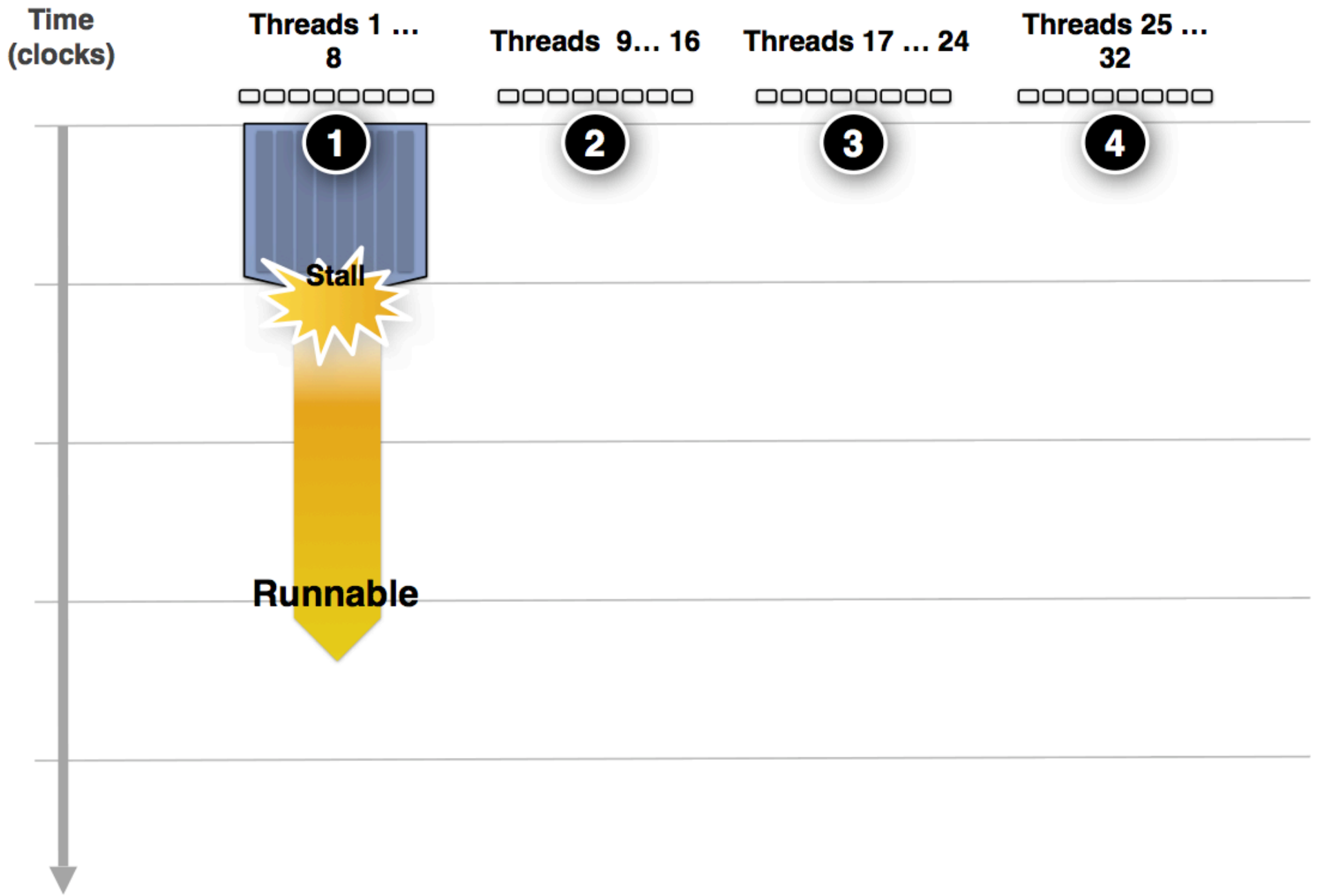
Slide from John Owens, CS267, 2017, Drawing from cantdrawnotfunny.com

How to hide stalls?



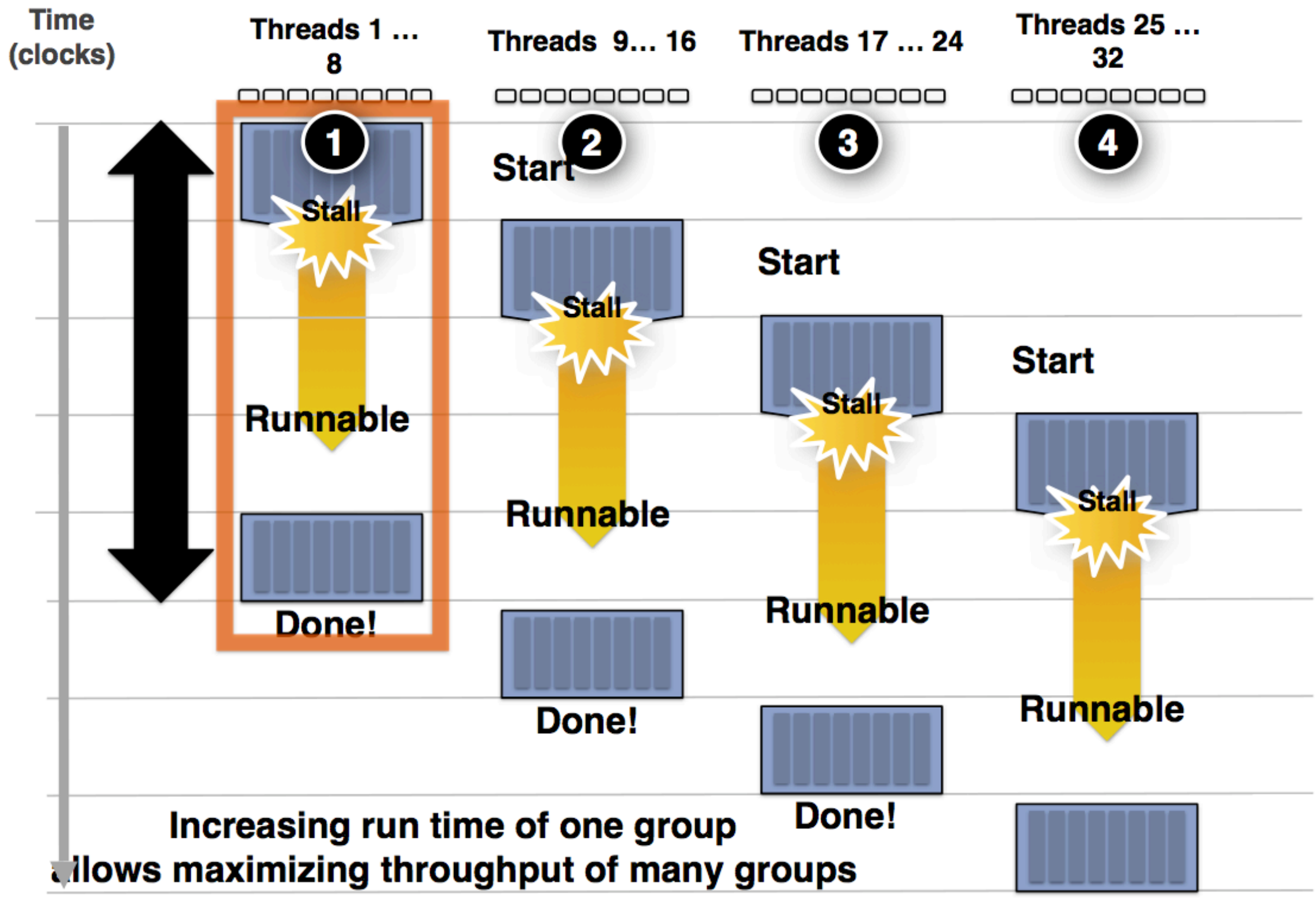
Slide from John Owens, CS267, 2017

Hiding stalls



Slide from John Owens, CS267, 2017

Hiding Stalls



Slide from John Owens, CS267, 2017

Key Hardware Ideas

- Use many ``slimmed down cores'' to run in parallel
- Packs cores full of ALUs
- Avoid latency stalls by interleaving execution of many warps
  - When one group stalls, switch to another warp with work ready

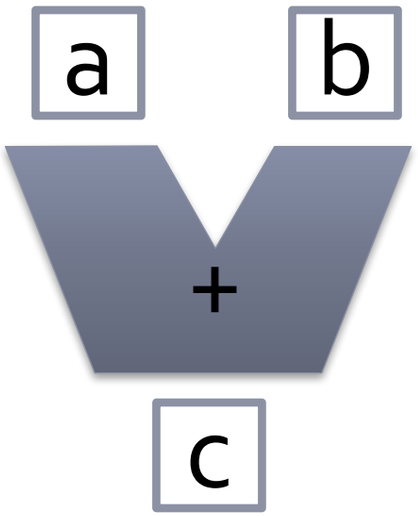
Summary of Latency vs Throughput

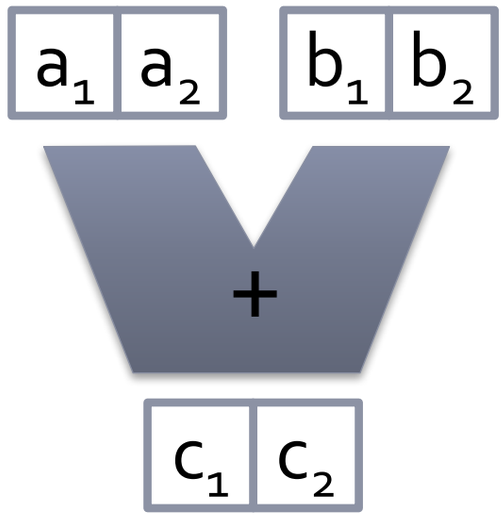
- Different goals produce different designs
- Latency cores: assume workload is mostly sequential
  - Superscalar/out-of-order execution, register renaming, branch prediction, speculative execution, cache hierarchy, speculative prefetching, etc.

- Throughput cores: assume work load is highly parallel
  - multithreading can hide latency … so skip the big caches and access external memory directly
  - No need for prefetching, or complicated controls when you can hide latency

SIMD

SISD



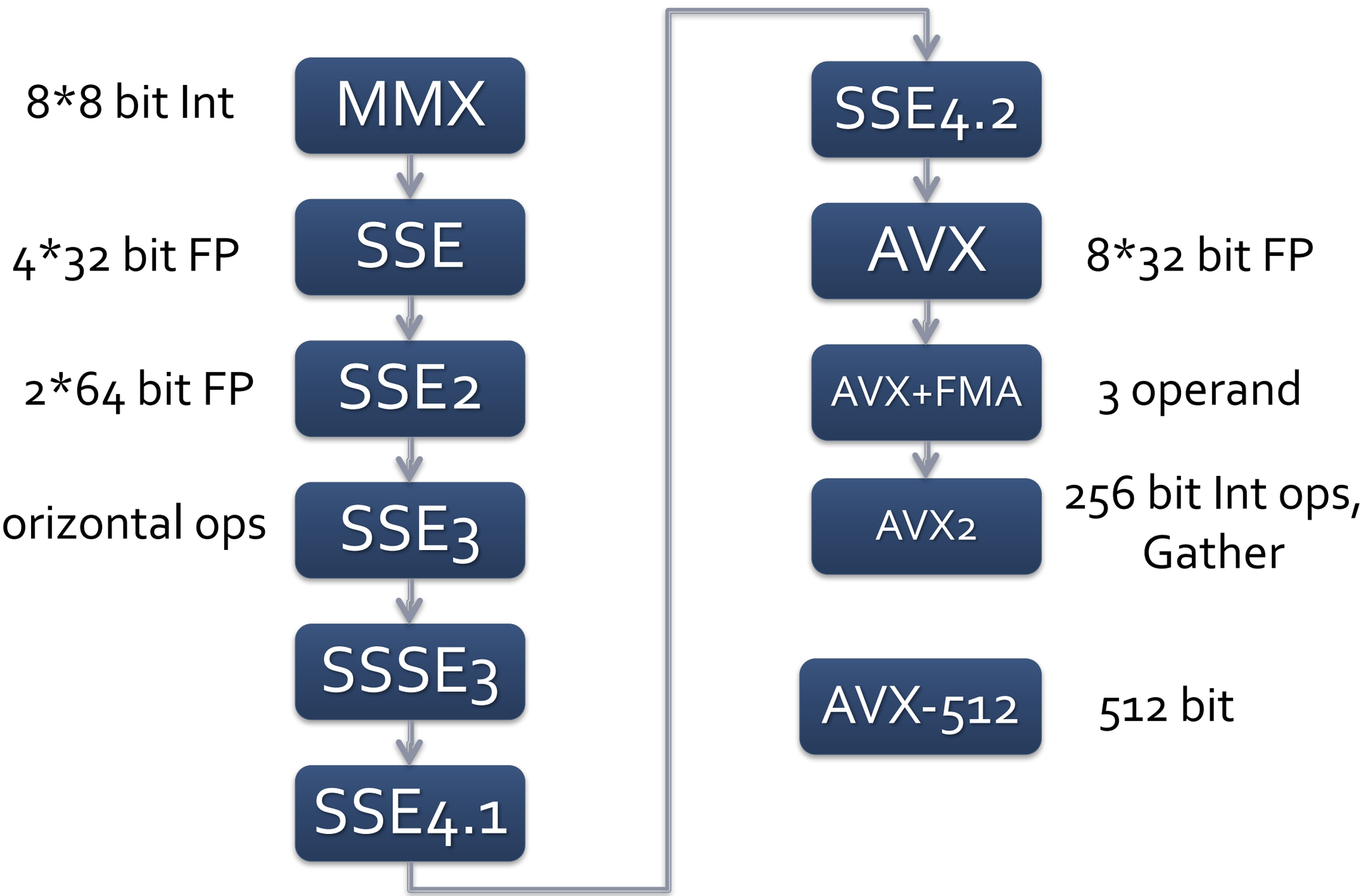


SIMD width=2

- Single Instruction Multiple Data architectures make use of data parallelism
- We care about SIMD because of area and power efficiency concerns
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler

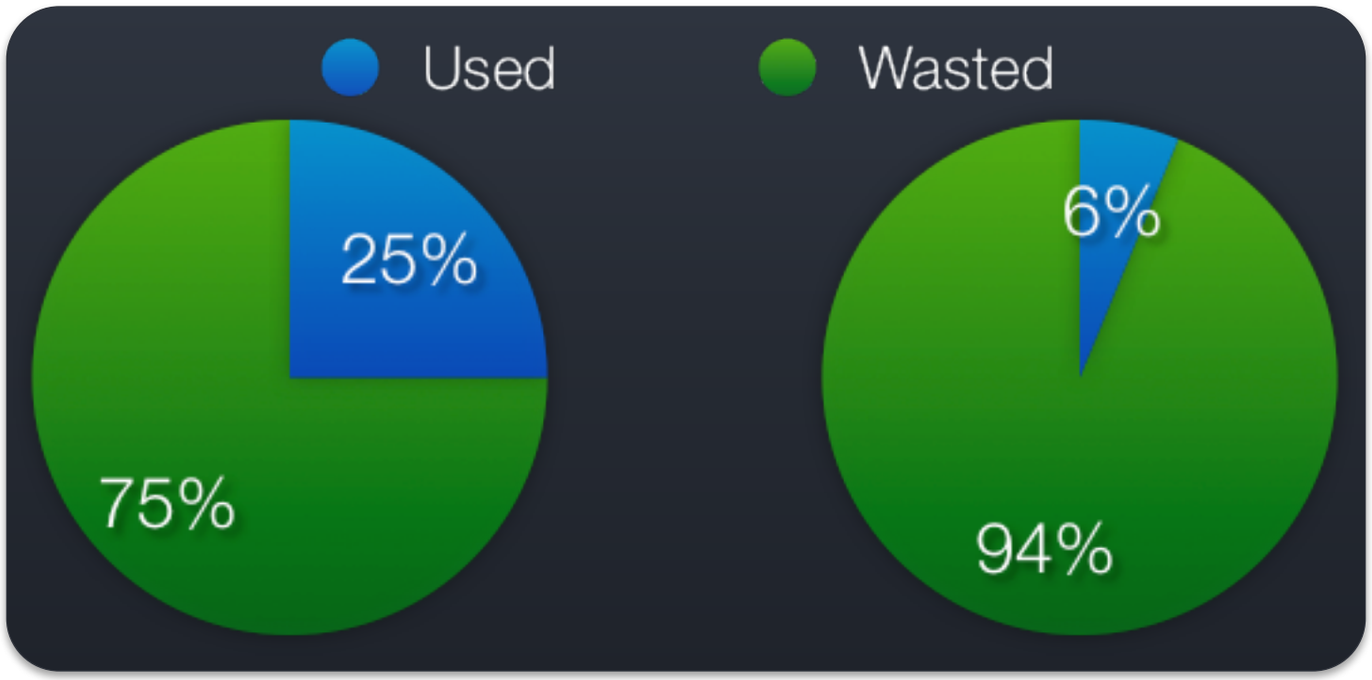
Slide from, Bryan Catanzaro, CS267, 2015

A Brief History of x86 SIMD Extensions



Slide from, Bryan Catanzaro, CS267, 2015

What to do with SIMD?



4 way SIMD (SSE) 16 way SIMD (Intel Xeon Phi)

- Neglecting SIMD is becoming more expensive
  - AVX: 8 way SIMD, Xeon Phi: 16 way SIMD,
    Nvidia GPU: 32 way SIMD, AMD GPU: 64 way SIMD
- We need a programming model which addresses both problems (Threads *and* SIMD)

Single Instruction Multiple Threads

- GPUs model is Single Instruction Multiple Threads (SIMT)
- So how is SIMT different than SIMD?
  - Every thread has its own register set
  - Possible to execute multiple paths
  - Single instruction can be performed on multiple addresses
- SIMD < SIMT < SMT

SIMT

- SIMT Cons:
- Low occupancy greatly reduces performance
- Flow divergence greatly reduces performance
- Synchronization options are very limited

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

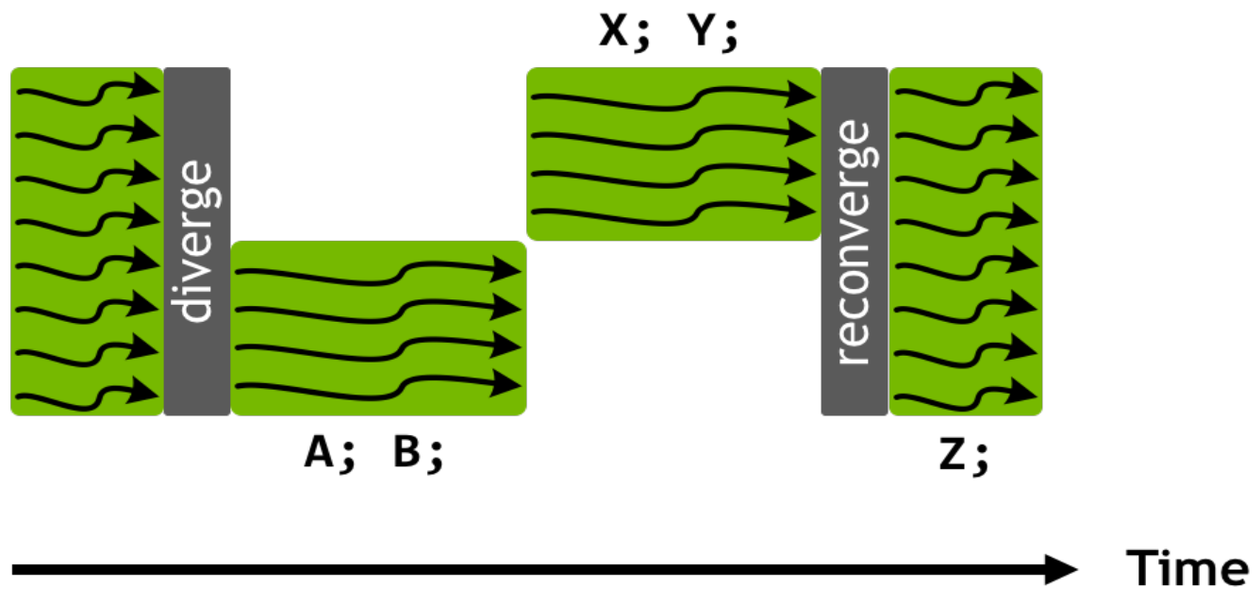


Figure from Mark Harris, Parallel Forall

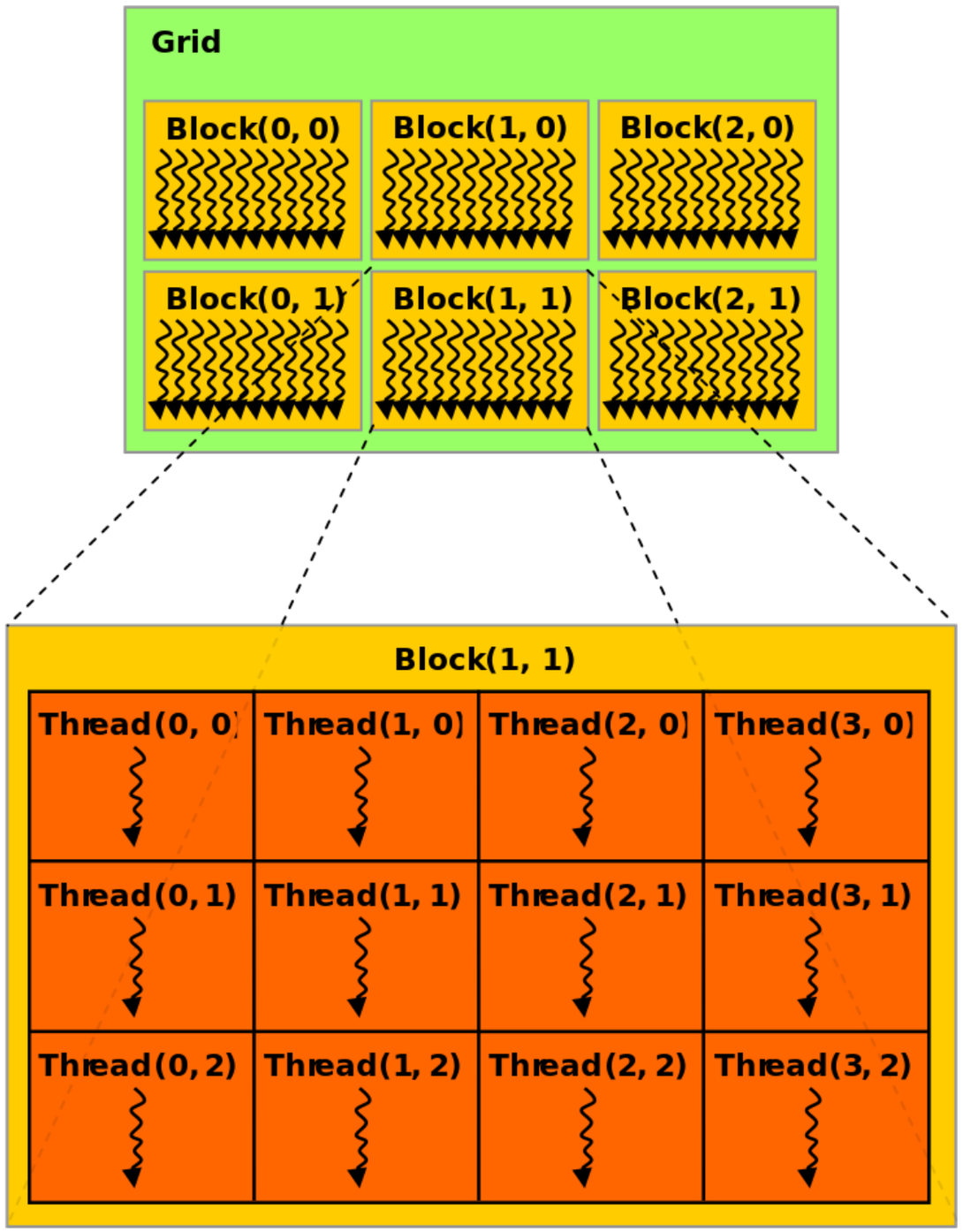
The CUDA Programming Model

- CUDA is a programming model designed for:
  - Heterogeneous architectures
  - SIMT parallelism
  - Scalability

- CUDA programs are written in C++ with minimal extensions

- OpenCL is inspired by CUDA, but HW & SW vendor neutral

Hierarchy of Threads

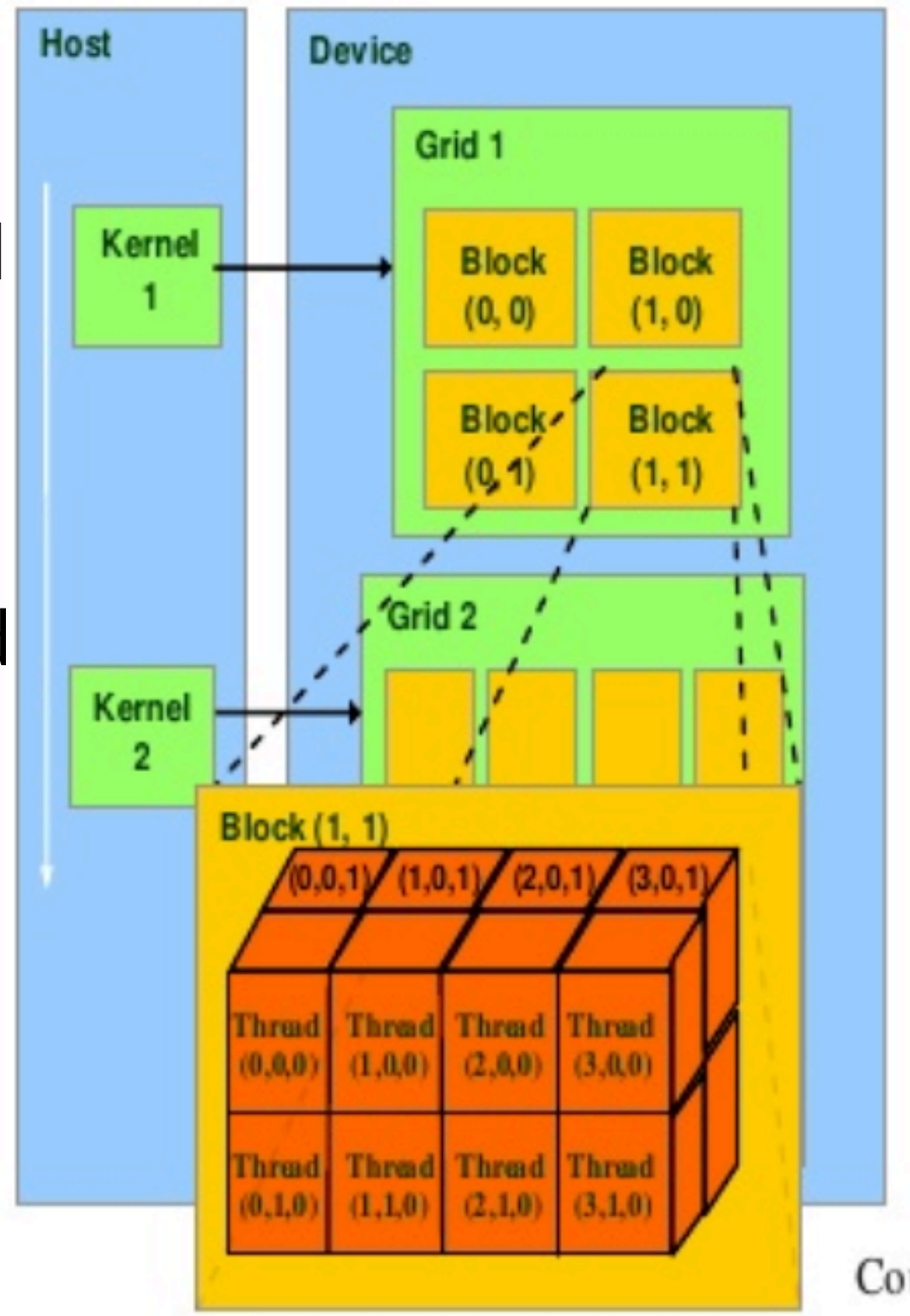


- **Block** has id, per-block shared memory
  - Can be 1/2/3D
  - Cuda: blockIdx.x/y/z
- **Warp**: Not reflected in the programming model
  - Thread/memory divergence very important for performance
- **Thread**, has its own register, private memory
  - Cuda: Can be 1/2/3D
  - Cuda: threadIdx.x/y/z

Hierarchy of Threads

2d or 3d are just the way the blocks and threads are organized.

You can do the same with 1d blocks and threads



Courtesy: NDVIA

What is a CUDA Thread?

- Independent thread of execution
  - has its own program counter, variables (registers), processor state, etc.
  - no implication about how threads are scheduled

- CUDA threads might be physical threads
  - as mapped onto NVIDIA GPUs

- CUDA threads might be virtual threads
  - might pick 1 block = 1 physical thread on multicore CPU

CUDA Supports:

- Thread parallelism
  - each thread is an independent thread of execution

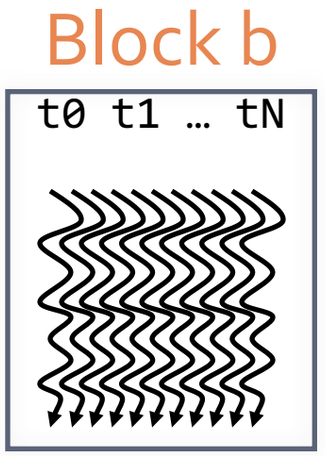
- Data parallelism
  - across threads in a block
  - across blocks in a kernel

- Task parallelism
  - different blocks are independent
  - independent kernels executing in separate streams

Hierarchy of Concurrent Threads

- Parallel kernels composed of many threads
  - all threads execute the same sequential program

- Threads are grouped into thread blocks
  - threads in the same block can cooperate



- Threads/blocks have unique IDs

Slide from, Bryan Catanzaro, CS267, 2015

What is a CUDA Thread Block?

- Thread block = a (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want

- Thread blocks of kernel must be independent tasks
  - program valid for ***any interleaving*** of block executions

Slide from, Bryan Catanzaro, CS267, 2015

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock
- As of Cuda9 on Pascal/Volta hardware it is possible to synchronize across blocks

Basic Cuda Program Structure

```
int main (int argc, char **argv ) {

    1. Allocate memory space in device (GPU) for input and output data
    2. Create input data in host (CPU)

    3. Copy input data to GPU

    4. Call "kernel" routine to execute on GPU
    (with CUDA syntax that defines no of threads and their logical organization)

    5. Transfer output data from GPU to CPU

    6. Free memory space in device (GPU)
    7. Free memory space in host (CPU)

    return value;
}
```

iterate

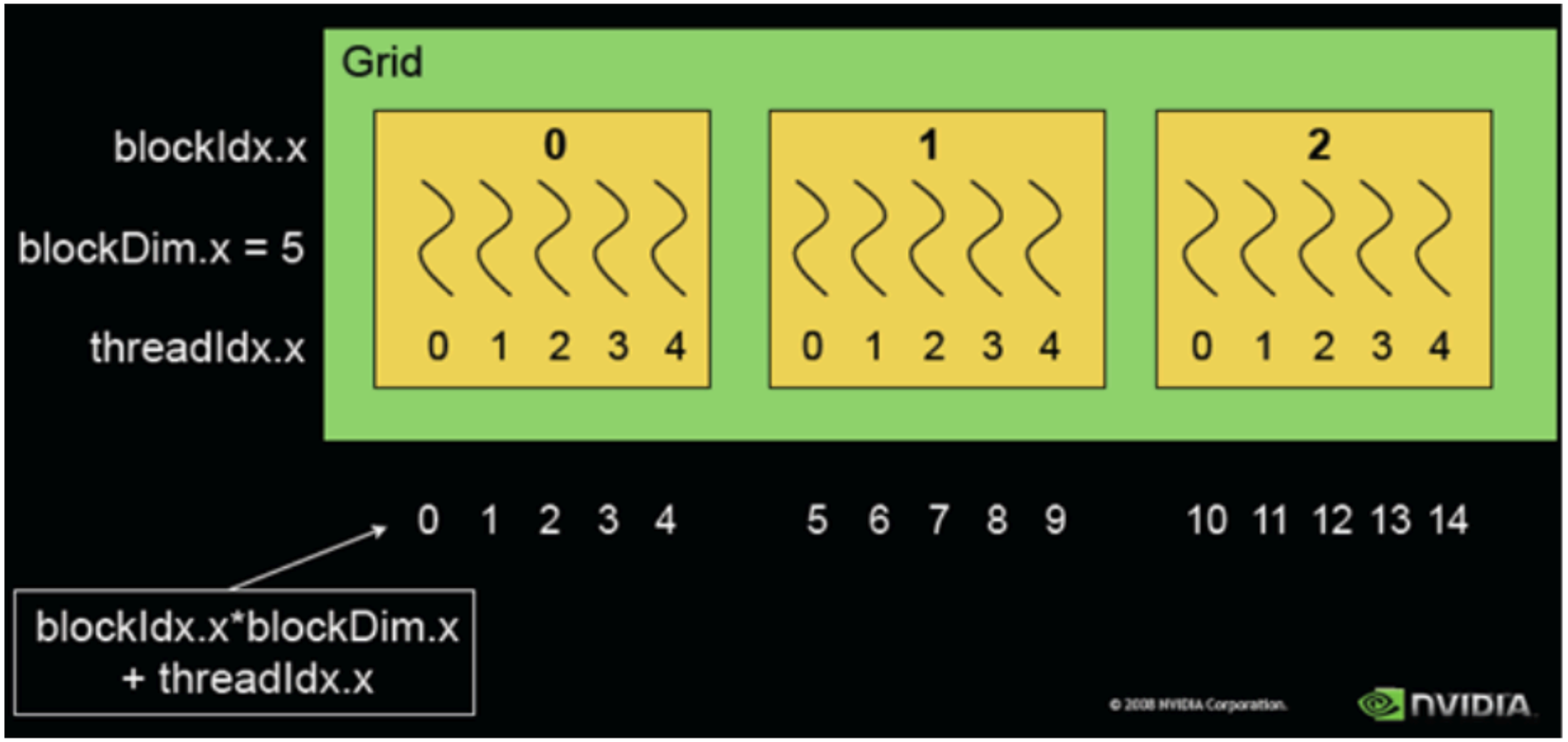
Cuda Function Declarations

	Executed on the	Only Callable from
__device__ float DFunc()	Device	Device
__global__ void Kernel()	Device	Host
__host__ float HFunc()	Host	Host

- Note that __global__ can only return void

General Grid/Block Specification

```
__global__ void Kernel(...){
  ...
}

dim3 DimGrid(100, 50); // A Grid of 100x50 Blocks
dim3 DimBlock(4, 8, 8); // 256 Threads per Block
size_t SharedMemBytes = 64; // 64 Bytes of shared memory
Kernel<<< DimGrid, DimBlock, SharedMemBytes>>>(...);
```



Example 1: Vector Addition

```
//Compute vector sum C=A+B
//Each thread performs one pairwise addition
__global__ void vecAdd(float* a, float* b, float* c) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   c[i] = a[i] + b[i];
}

int main() {
   //Run N/256 blocks of 256 threads each
   vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);
}
```

Slide from, Bryan Catanzaro, CS267, 2015

Example 1: Vector Addition

```
//Compute vector sum C=A+B
//Each thread performs one pairwise addition
__global__ void vecAdd(float* a, float* b, float* c) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if(i < N)
     c[i] = a[i] + b[i];
   return;
}

int main() {
   //Run N/256 blocks of 256 threads each
   vecAdd<<<ceil(N/256), 256>>>(d_a, d_b, d_c);
}
```

Slide from, Bryan Catanzaro, CS267, 2015

Hello World: Managing Data

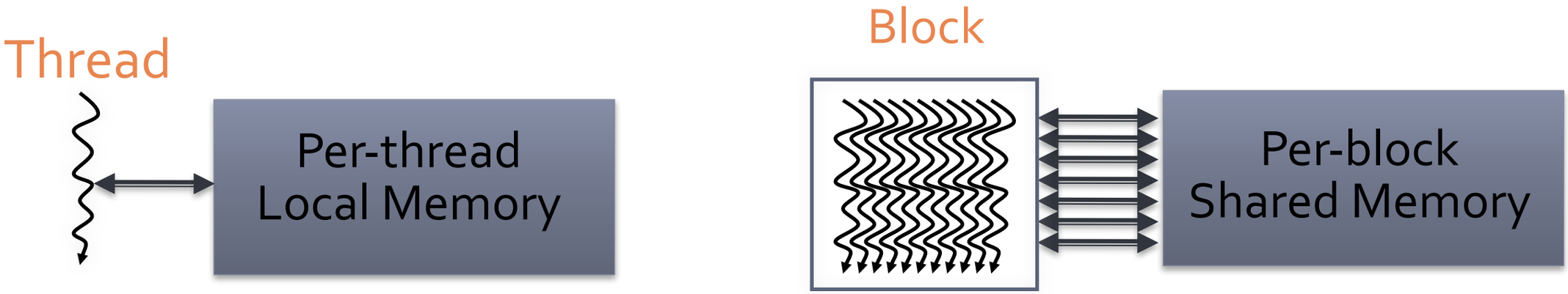
```
#Define N = 256 * 1024;
int main() {
   float* h_a = malloc(sizeof(float) * N);
   //Similarly for h_b, h_c. Initialize h_a, h_b

   float *d_a, *d_b, *d_c;
   cudaMalloc(&d_a, sizeof(float) * N);
   //Similarly for d_b, d_c

   cudaMemcpy(d_a, h_a, sizeof(float) * N, cudaMemcpyHostToDevice);
   //Similarly for d_b

   //Run N/256 blocks of 256 threads each
   vecAdd<<<ceil(N/256), 256>>>(d_a, d_b, d_c);

   cudaMemcpy(h_c, d_c, sizeof(float) * N, cudaMemcpyDeviceToHost);
}
```

Slide from, Bryan Catanzaro, CS267, 2015

Key Points for Performance

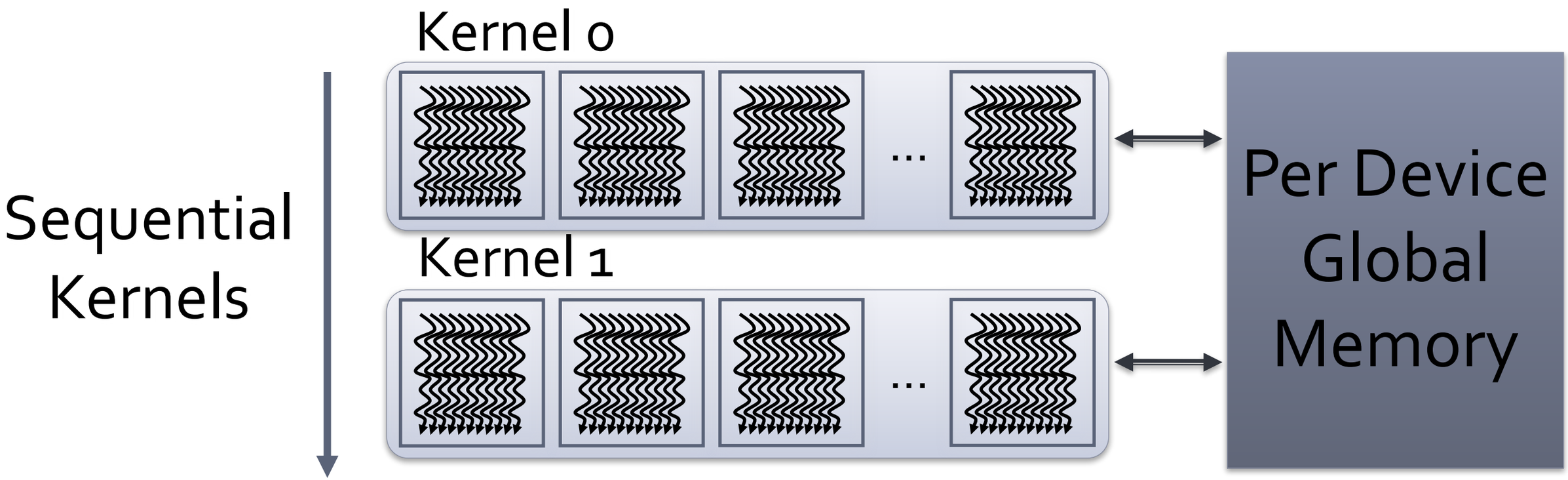
- Usual requirements
  - Algorithm must have a lot parallelism
- Locality
  - Crucial as GPUs as they do not have sophisticated hardware as the CPU to hide it
  - Global memory: memory coalescing
  - Shared memory: avoiding bank conflicts
- Exploiting SIMT:
  - Thread divergence: What happens when threads in a warp follow different control paths?

Memory model



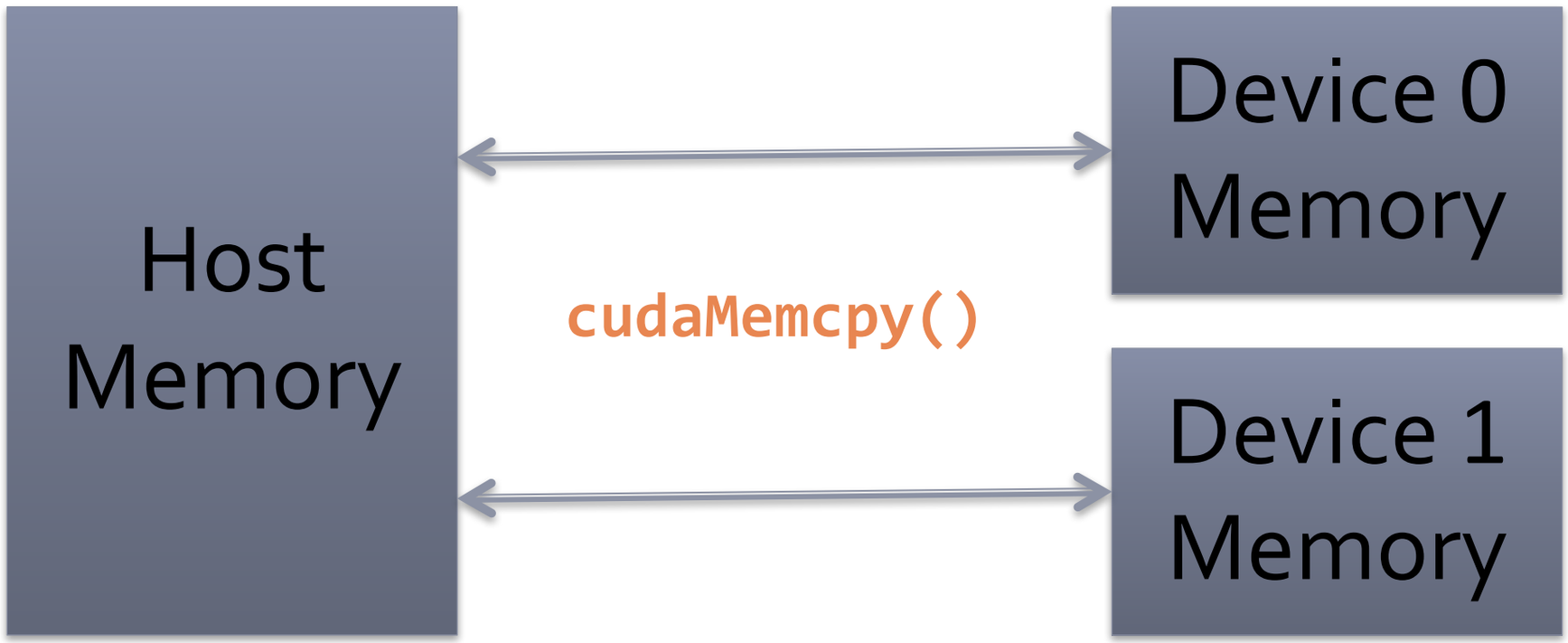
Slide from, Bryan Catanzaro, CS267, 2015

Memory model



Slide from, Bryan Catanzaro, CS267, 2015

Memory model



Slide from, Bryan Catanzaro, CS267, 2015

Thread divergence

- The programming model allows each thread in a warp to execute different instructions, but this leads to **performance problem in warp level**:

- Hardware: different execution paths are serialized
  - Different control paths taken by threads in a warp are traversed **one at a time** until they all converge again

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

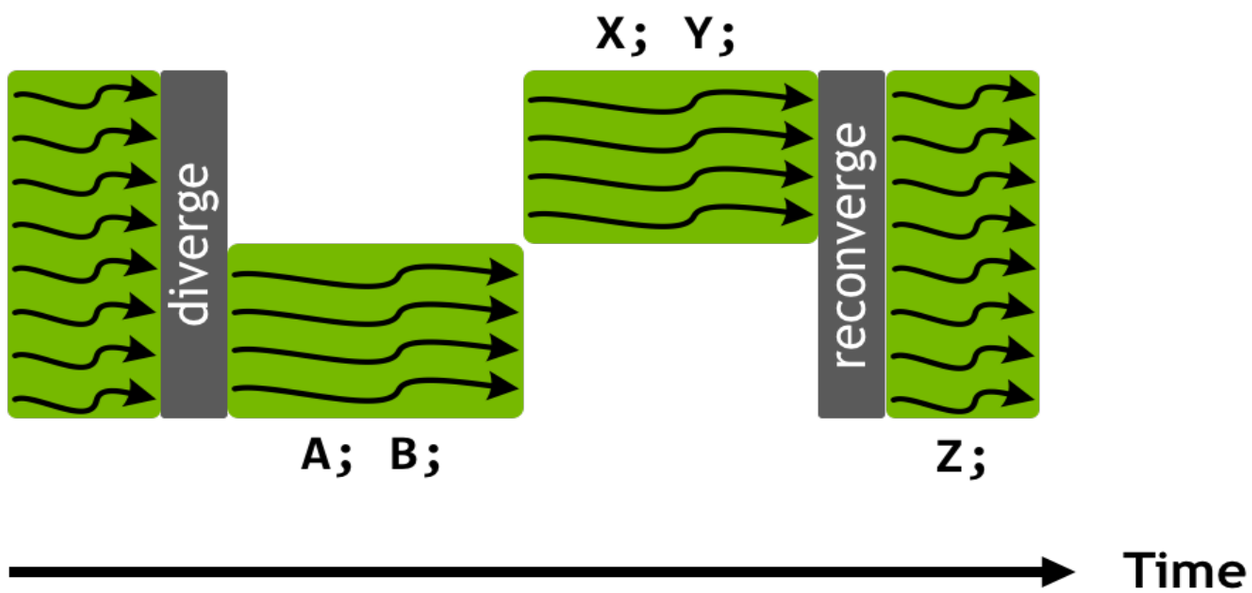


Figure from Mark Harris, Parallel Forall

Synchronization

- Threads within a block may synchronize with barriers

```
… Step 1 …
__syncthreads();
… Step 2 …
```

- Blocks coordinate via atomic memory operations
  - e.g., increment shared queue pointer with `atomicInc()`

- Implicit barrier between dependent kernels

```
vec_minus<<<nblocks, blksize>>>(a, b, c);

vec_dot<<<nblocks, blksize>>>(c, c);
```

Example 2: Array Reduction

Q: Compute the sum of the matrix elements below on GPU

12	-1	4	5
6	7	8	-7
3	2	6	1
5	1	-10	2

Example 2: Array Reduction

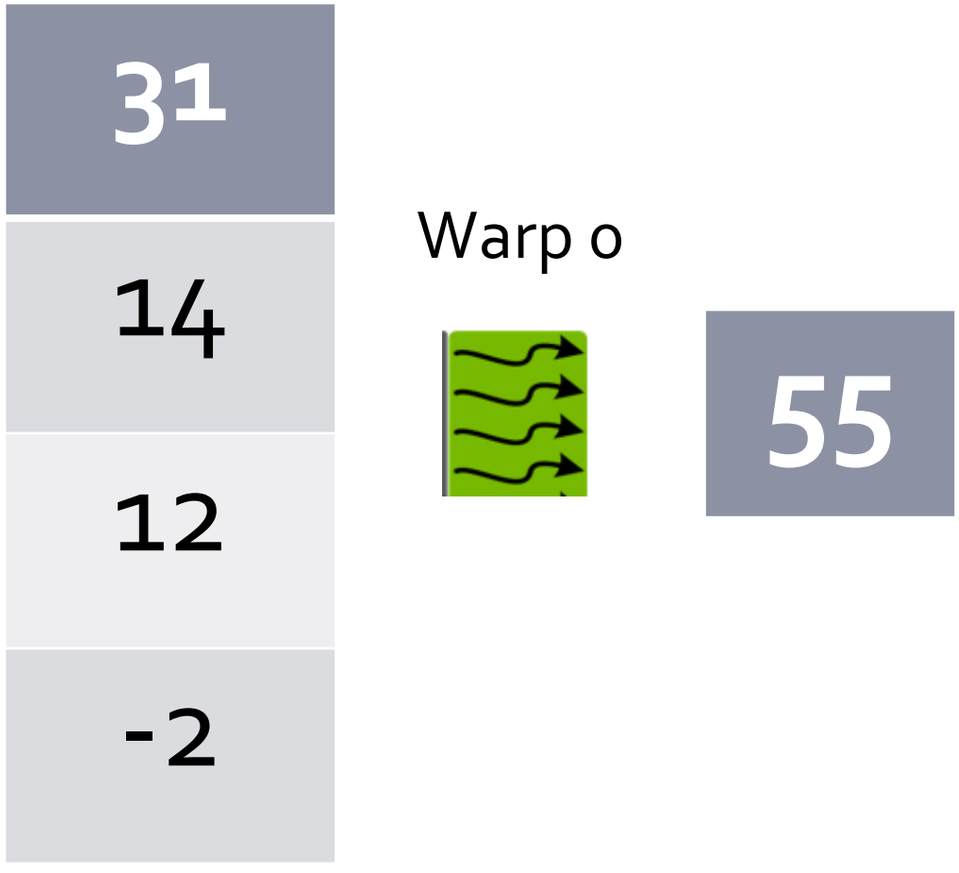
- First let each warp compute the sum of a row and store it in shared memory

12	-1	4	5		31	Warp 0
6	7	8	-7		14	Warp 1
3	2	6	1		12	Warp 2
5	1	-10	2		-2	Warp 3

Shared memorey

Example 2: Array Reduction

- First let each warp compute the sum of a row
- Wait for all warps to compute partial sums
  - Syncthreads()
- Let the first warp compute sum of partial sums



Block reduction

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

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

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

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

  __syncthreads();              // Wait for all partial reductions

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

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

  return val;
}
```

Code from Parallel Forall by Justin Luitjens

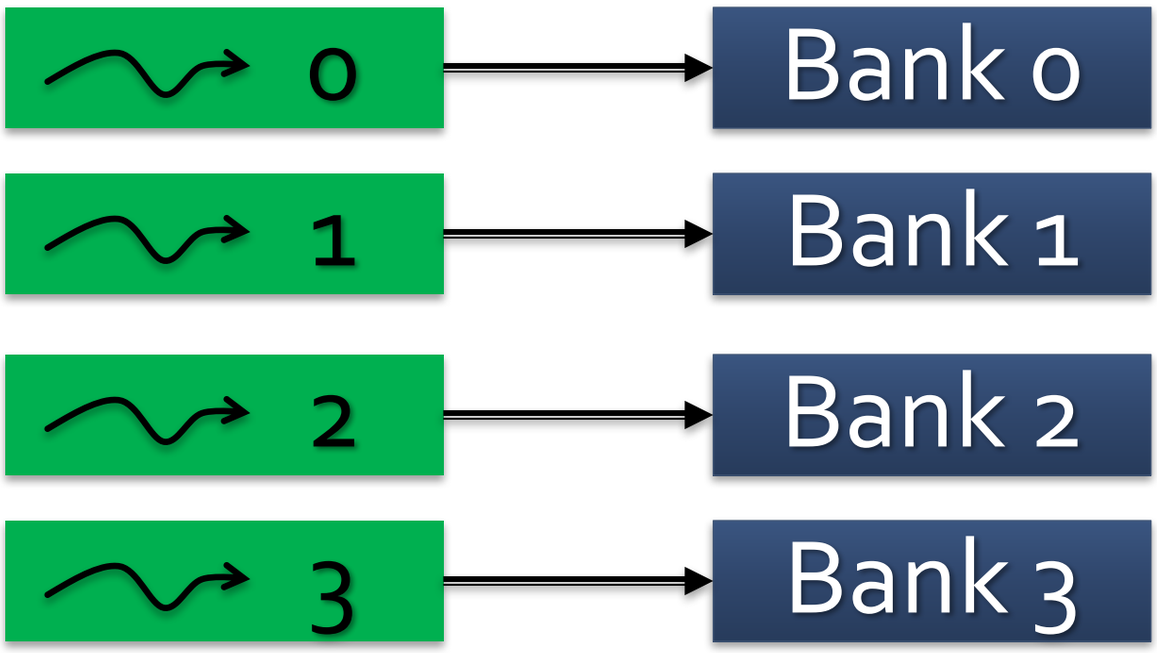
Shared Memory Bank Conflict

- Access to the same memory bank is serialized
- Successive 32-bit words are assigned to successive banks

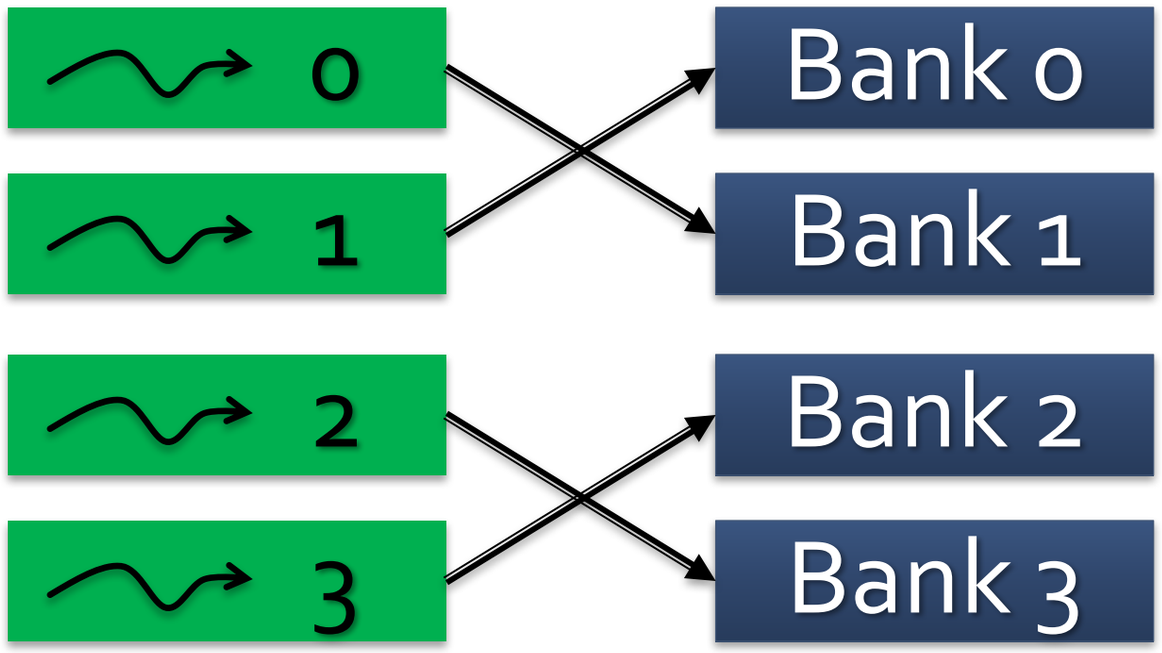
Bank	0				1				...
Address	0	1	2	3	4	5	6	7	...
Address	64	65	66	67	68	69	70	71	...

Bank Conflict

No Bank Conflict

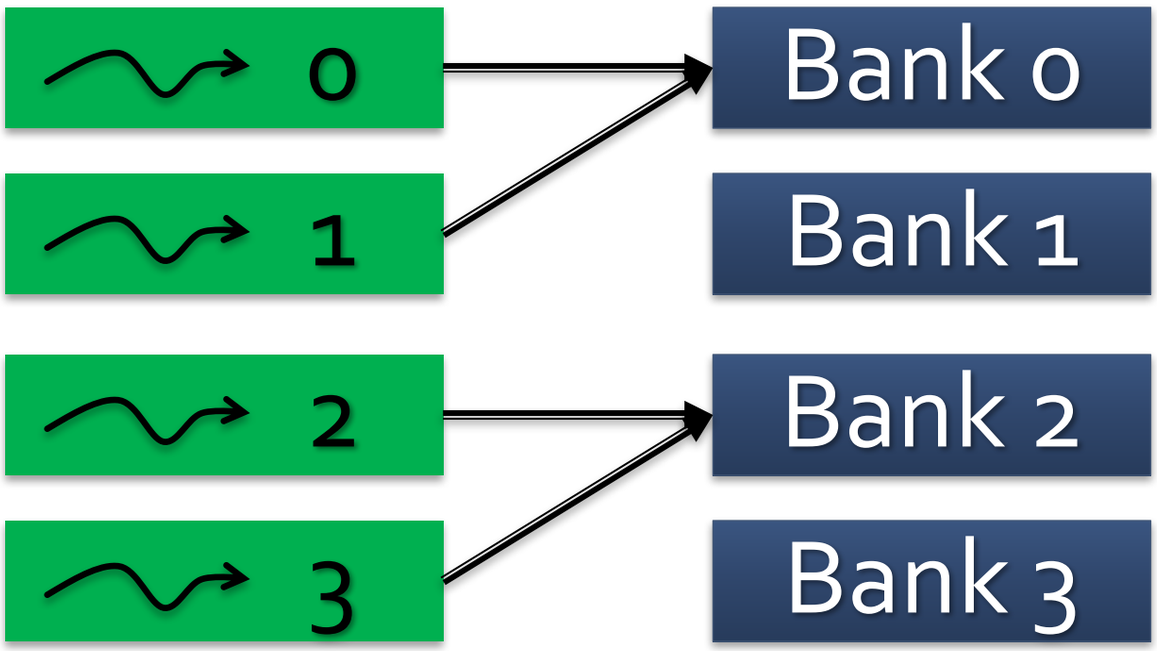


No Bank Conflict

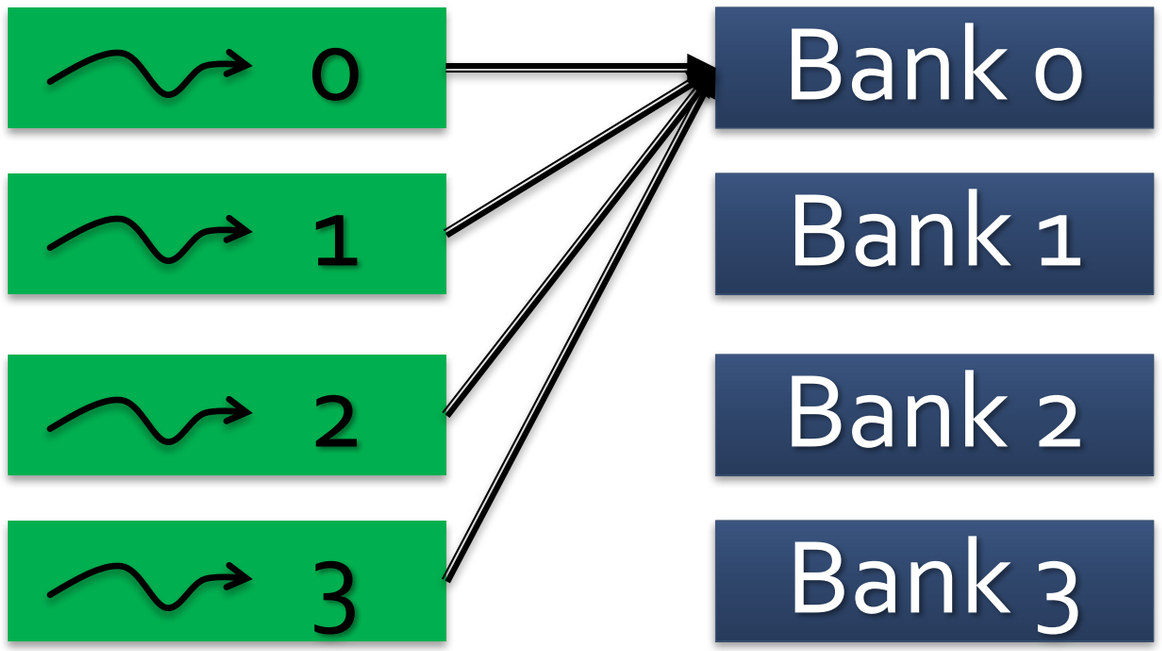


Bank Conflict

2-Way Bank Conflict



4-Way Bank Conflict



Global Memory Bank Conflict

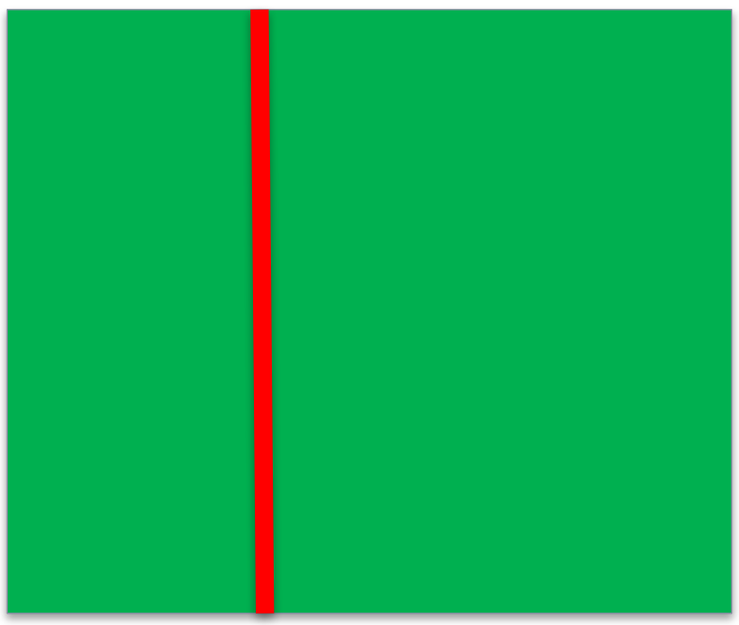
- The same problem can occur for global memory, as that is also partitioned into banks. However, starting from Fermi architecture, global memory addresses are hashed and thus global memory bank conflict is no longer an issue

Ruetsch, Greg, and Paulius Micikevicius. "Optimizing matrix transpose in CUDA." Nvidia CUDA SDK Application Note 18 (2009).

Global Memory Accesses

- Each load transaction brings some number of aligned, contiguous bytes from memory (let's call it a page)
- Hardware automatically combines requests from different threads in a warp (coalescing) to same page, but multiple pages are processed serially

Not Coalesced



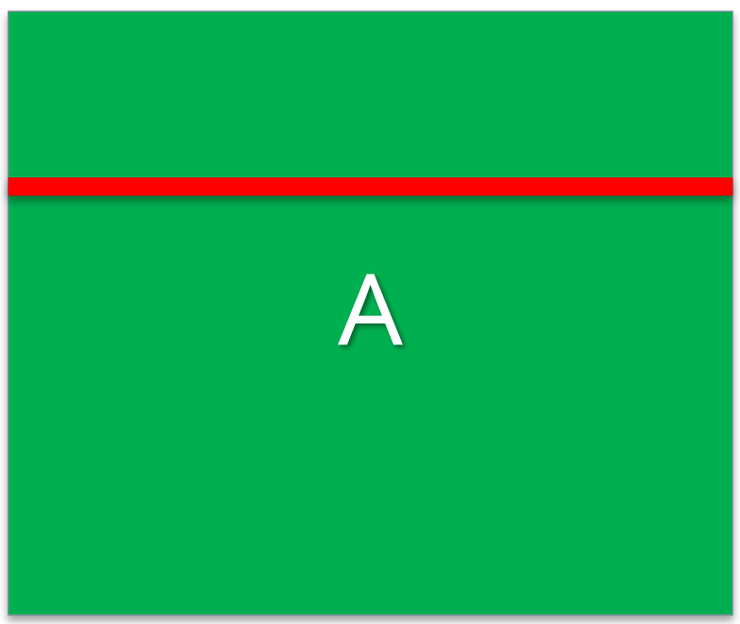
Coalesced

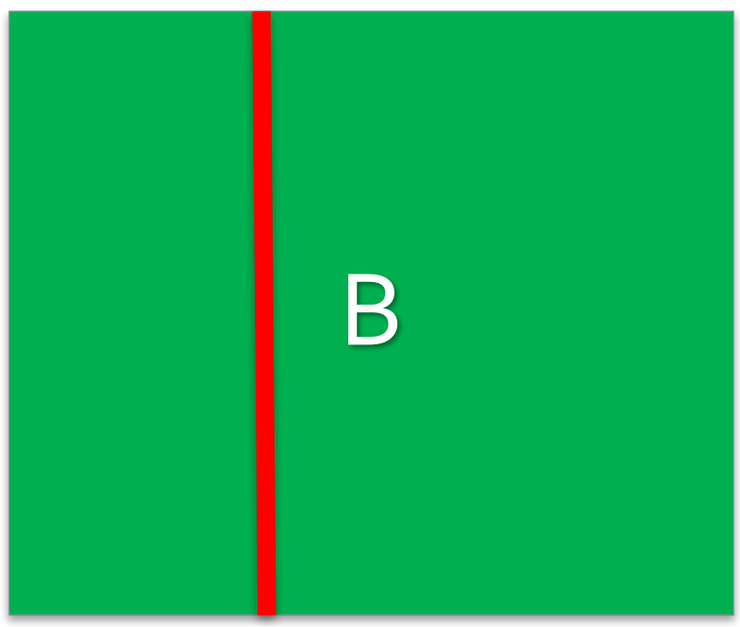
Example 3: Matrix Transpose

- B[j][i] = A[i][j]

Naïve Solution: Let every thread read from A, and store transposed results into B

Would this perform well?

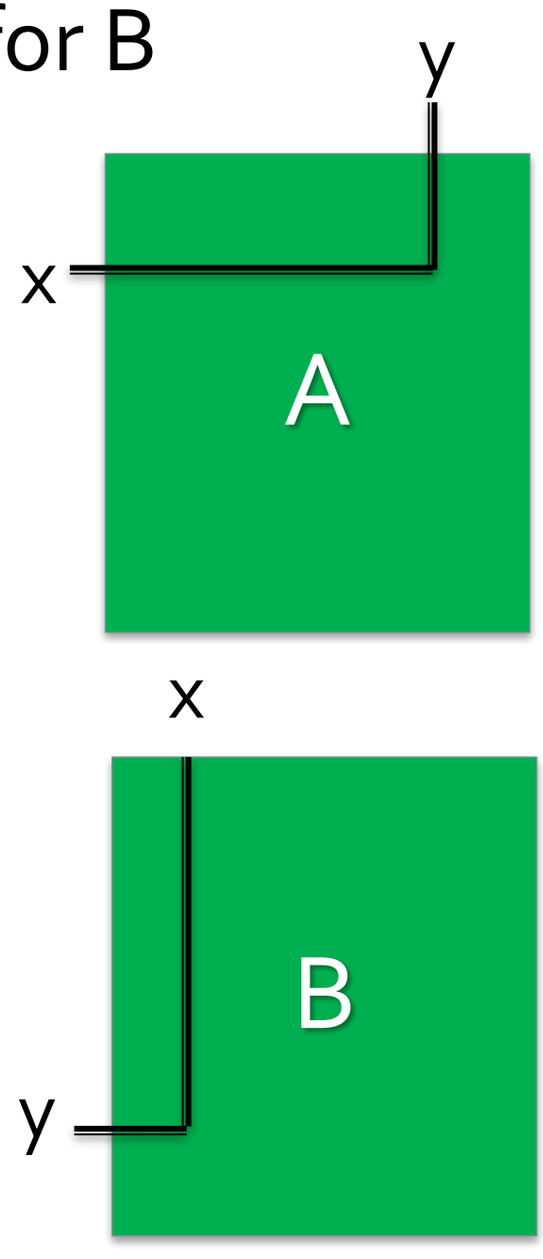




Example3: Matrix Transpose

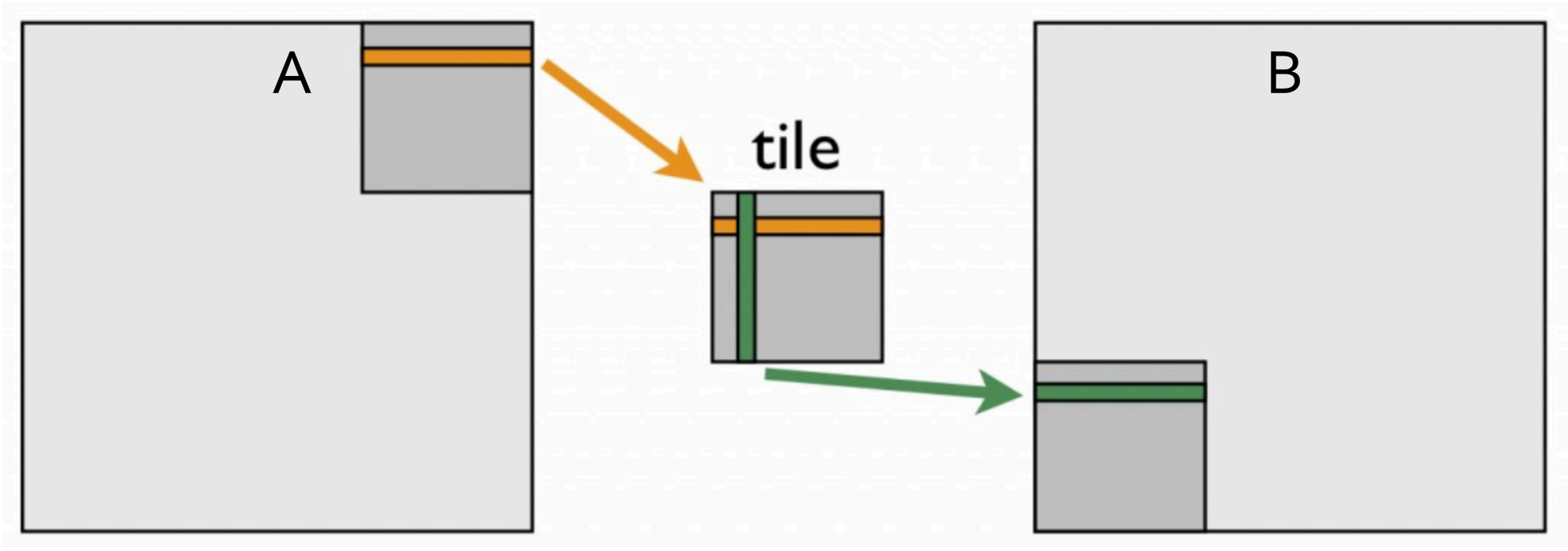
- 55 GB/s on K20c
- Main Problem: Incontiguous memory write for B

```
1   #define TILE_DIM 32
2
3   dim3 dimGrid(N/TILE_DIM, N/TILE_DIM, 1);
4   dim3 dimBlock(TILE_DIM, TILE_DIM, 1);
5
6   __global__ void transposeNaive(float *B, const float *A)
7   {
8     int x = blockIdx.x * TILE_DIM + threadIdx.x;
9     int y = blockIdx.y * TILE_DIM + threadIdx.y;
10
11    B[x * N + y] = A[y * N + x];
12  }
```



Example3: Shared Memory

- Let's use shared memory

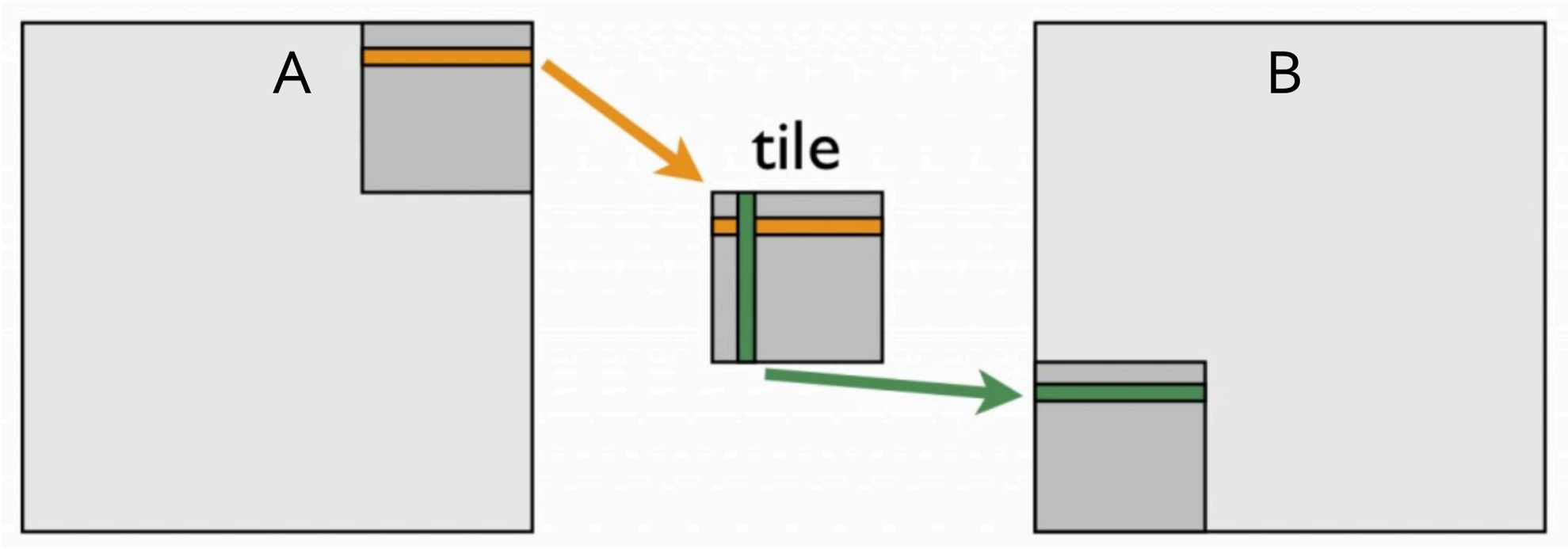


Q: Do we need to synchronize?

Figure from Mark Harris, Parallel Forall

Example3: Shared Memory

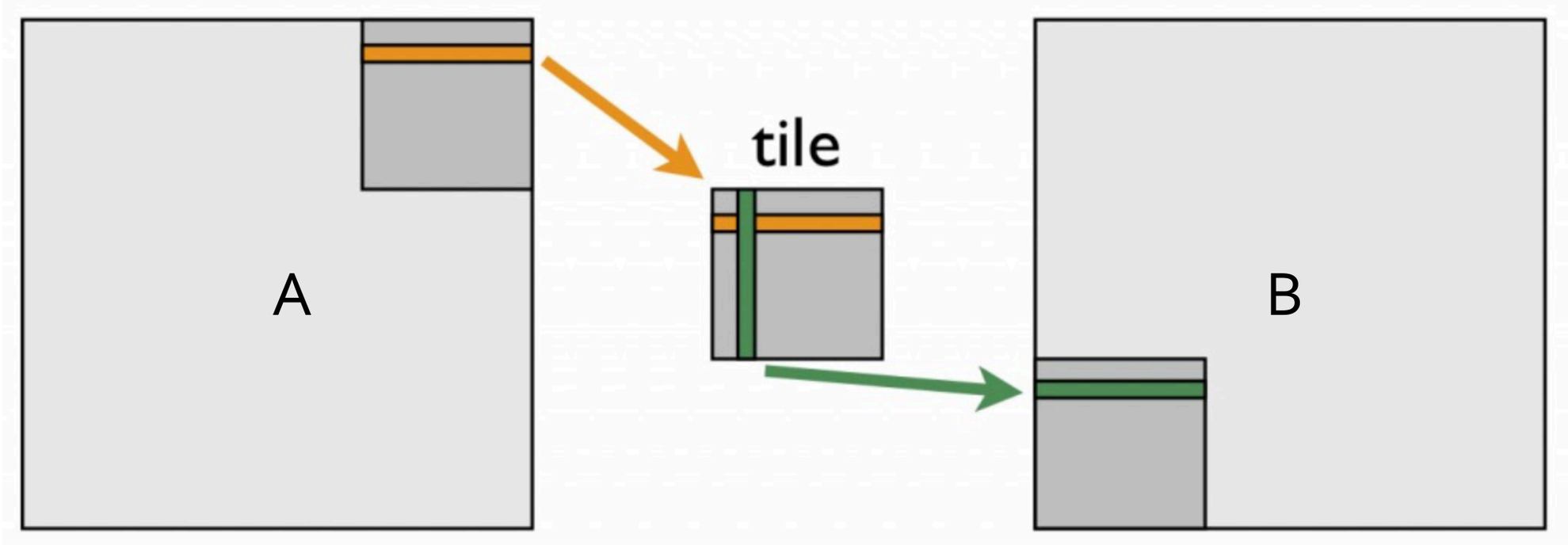
- Let's use shared memory



Q: Do we need to synchronize?
A: Yes, we need A's data to get filled into tile, before writing it into B

Figure from Mark Harris, Parallel Forall

Example3: Shared Memory



Q: What is the performance bottleneck of this algorithm?

A: Shared Memory Bank Conflict

Elegant Solution: Pad tile by 1!

```
__shared__ float tile[TILE_DIM][TILE_DIM + 1];
```

Figure from Mark Harris, Parallel Forall

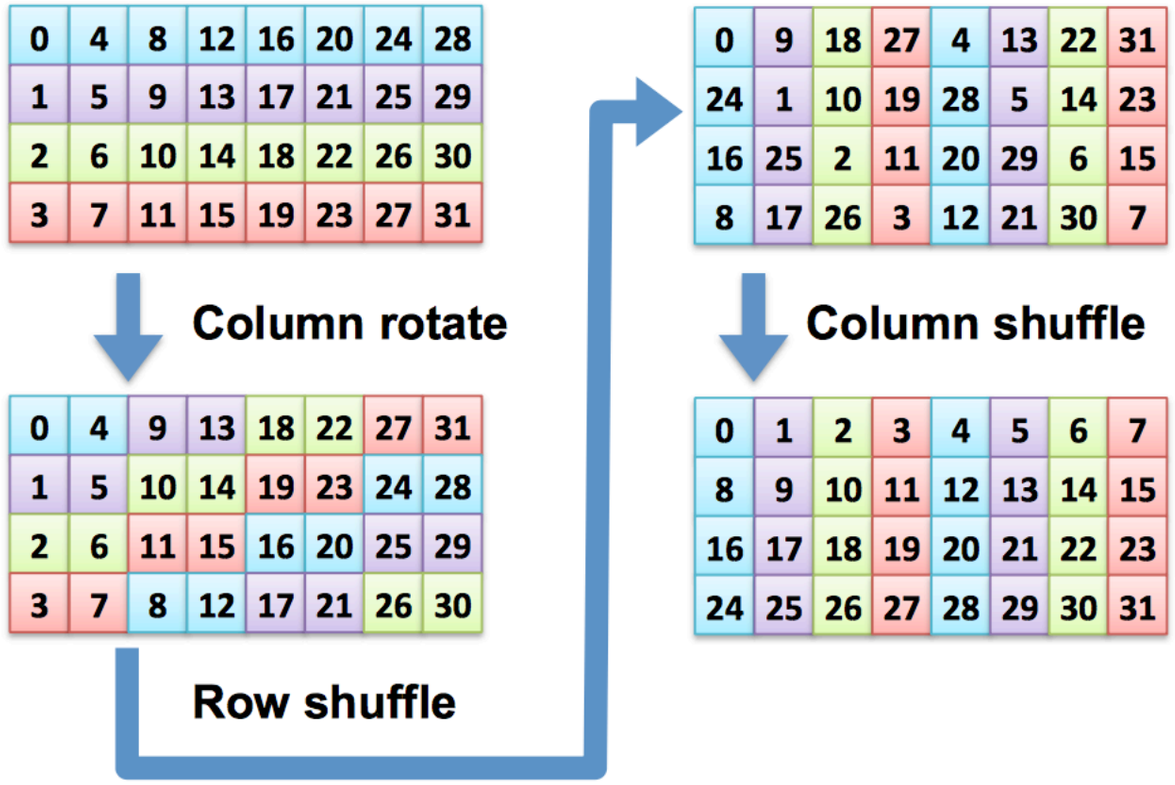
Example3: Matrix Transpose

Code for example 3

```
1   #define TILE_DIM 32
2
3   dim3 dimGrid(N/TILE_DIM, N/TILE_DIM, 1);
4   dim3 dimBlock(TILE_DIM, TILE_DIM, 1);
5   __global__ void transposeCoalesced(float *B, const float *A)
6   {
7     __shared__ float tile[TILE_DIM][TILE_DIM+1];
8
9     int x = blockIdx.x * TILE_DIM + threadIdx.x;
10    int y = blockIdx.y * TILE_DIM + threadIdx.y;
11
12    tile[threadIdx.y][threadIdx.x] = A[y * N + x];
13
14    __syncthreads();
15
16    x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
17    y = blockIdx.x * TILE_DIM + threadIdx.y;
18
19    B[y * N + x] = tile[threadIdx.x][threadIdx.y];
20  }
```

Rectangular Inplace Matrix Transpose

- Open research problem
- "**In-place transposition of rectangular matrices on accelerators**."
  - Sung, I-Jui, et al. 2014, *ACM SIGPLAN Notices*
- "**A decomposition for in-place matrix transposition**"
  - Catanzaro, B., Keller, A. and Garland, M., 2014. *ACM SIGPLAN Notices*



CUDA: Features available on GPU

- Double and single precision (IEEE compliant)
- Standard mathematical functions
  - `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.
- Atomic memory operations
  - `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.
- These work on both global and shared memory

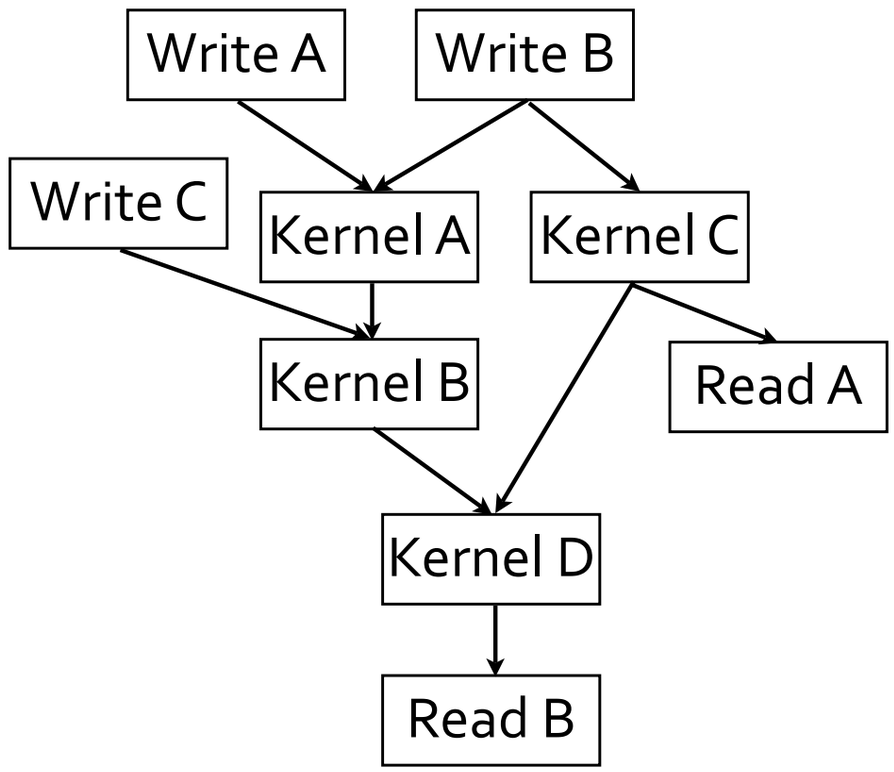
CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory
  - `cudaMalloc()`, `cudaFree()`
- Explicit memory copy for host ↔ device, device ↔ device
  - `cudaMemcpy()`, `cudaMemcpy2D()`, ...
- Texture management
  - `cudaBindTexture()`, `cudaBindTextureToArray()`, ...
- OpenGL & DirectX interoperability
  - `cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, ...

Slide from, Bryan Catanzaro, CS267, 2015

OpenCL

- OpenCL is supported by AMD {CPUs, GPUs} and Nvidia
  - Intel, Imagination Technologies (purveyor of GPUs for iPhone/etc.) are also on board
- OpenCL's data parallel execution model mirrors CUDA, but with different terminology
- OpenCL has rich task parallelism model
  - Runtime walks a dependence DAG of kernels/memory transfers



Slide from, Bryan Catanzaro, CS267, 2015

CUDA and OpenCL correspondence

Thread	⟷	Work-item
Thread-block	⟷	Work-group
Global memory	⟷	Global memory
Constant memory	⟷	Constant memory
Shared memory	⟷	Local memory
Local memory	⟷	Private memory
__global__ function	⟷	__kernel function
__device__ function	⟷	no qualification needed
__constant__ variable	⟷	__constant variable
__device__ variable	⟷	__global variable
__shared__ variable	⟷	__local variable

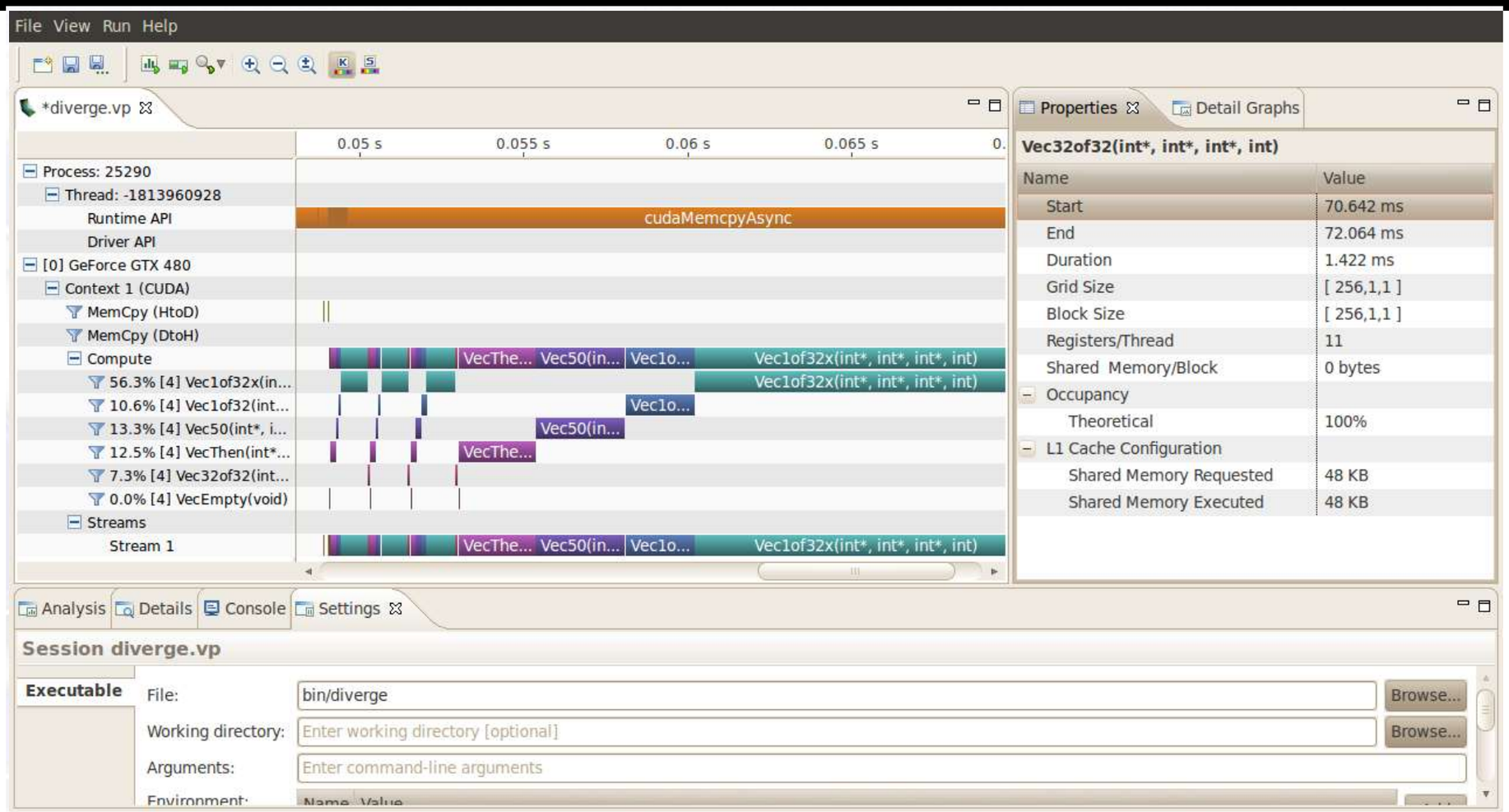
Slide from, Bryan Catanzaro, CS267, 2015

Imperatives for Efficient CUDA Code

- Expose abundant fine-grained parallelism
  - need 1000's of threads for full utilization
- Maximize on-chip work
  - on-chip memory orders of magnitude faster
- Minimize execution divergence
  - SIMT execution of threads in 32-thread warps
- Minimize memory divergence
  - warp loads and consumes complete 128-byte cache line

Slide from, Bryan Catanzaro, CS267, 2015

Profiling



- nvvp (nvidia visual profiler) useful for interactive profiling
- `export CUDA_PROFILE=1` in shell for simple profiler
  - Then examine cuda_profile_*.log for kernel times & occupancies

Slide from, Bryan Catanzaro, CS267, 2015

Summary

- Algorithm must have a lot parallelism
- Ensure there is good locality: Crucial as GPUs as they do not have sophisticated hardware as the CPU to hide it
- Global memory: memory coalescing
- Shared memory: avoiding bank conflicts
- If possible avoid thread divergence

What's Next?

- Cuda libraries, cublas, cusparse, cufft
- Multi-GPU programming on a single socket
  - How to exploit fast peer-to-peer transfers
  - Cuda unified memory model
- Distributed-memory Multi-GPU programming
  - Hybrid MPI + Cuda
- Writing code in assembly!

Further Reading

- Further reading:
- **"Programming Massively Parallel Processors"**
  - Book by D. Kirk, W. Hwu
- **"Benchmarking GPUs to Tune Dense Linear Algebra"**
  - V. Volkov, J. James, SC'08

References

- Bryan Catanzaro slides for CS267-2014
- John Owens slides for CS267-2017
- Prof. Keshav Pingali's slides for Programming for performance course, Fall 2012

Questions?

Please feel free to contact me for questions or project ideas

amirgh@berkeley.edu