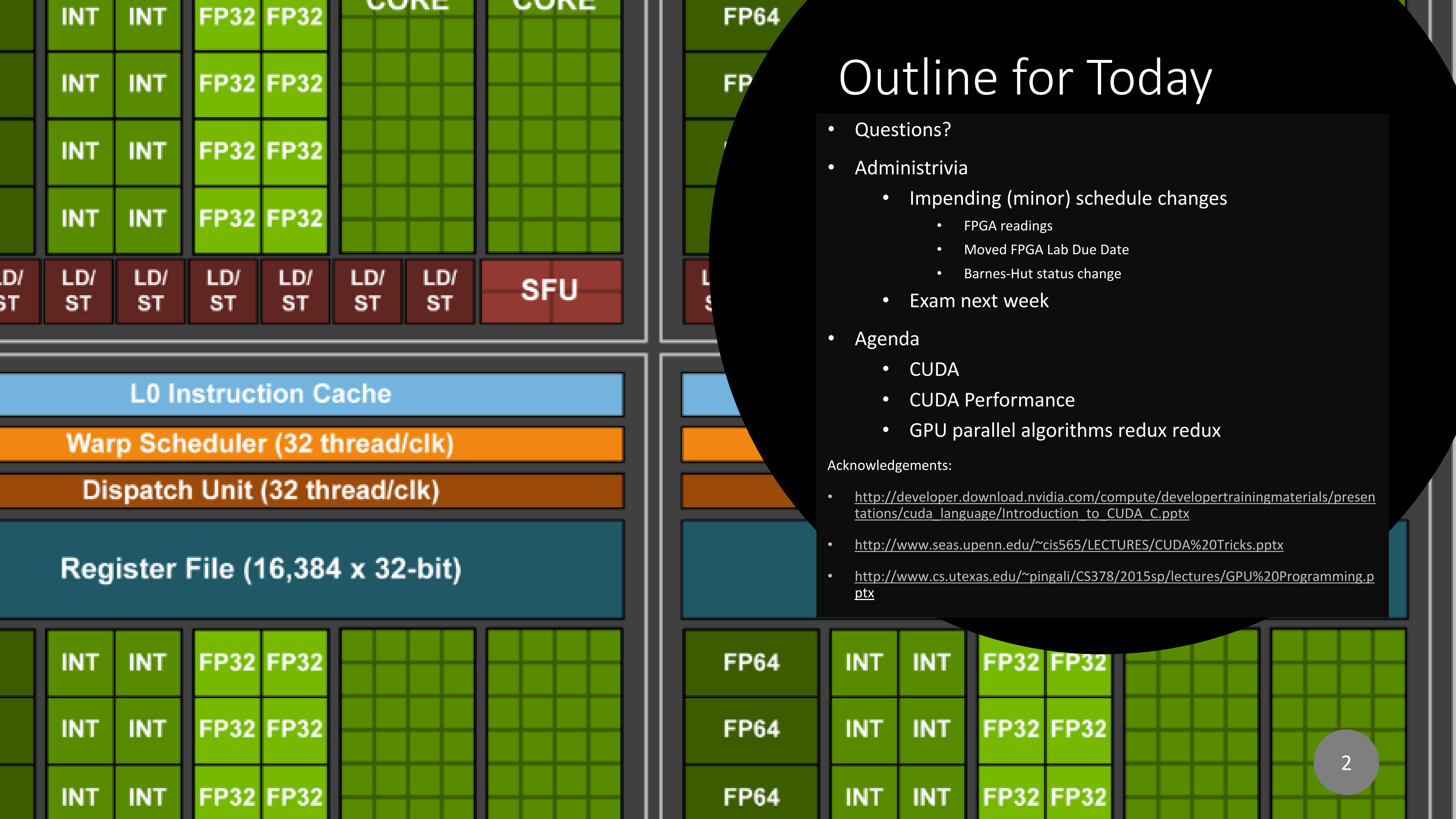
# *GPUs to the left GPUs to the right GPUs all day GPUs all night*

Chris Rossbach

cs378h


INT
FP32
FP64
CORE
LD/ ST
SFU
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)

# Outline for Today

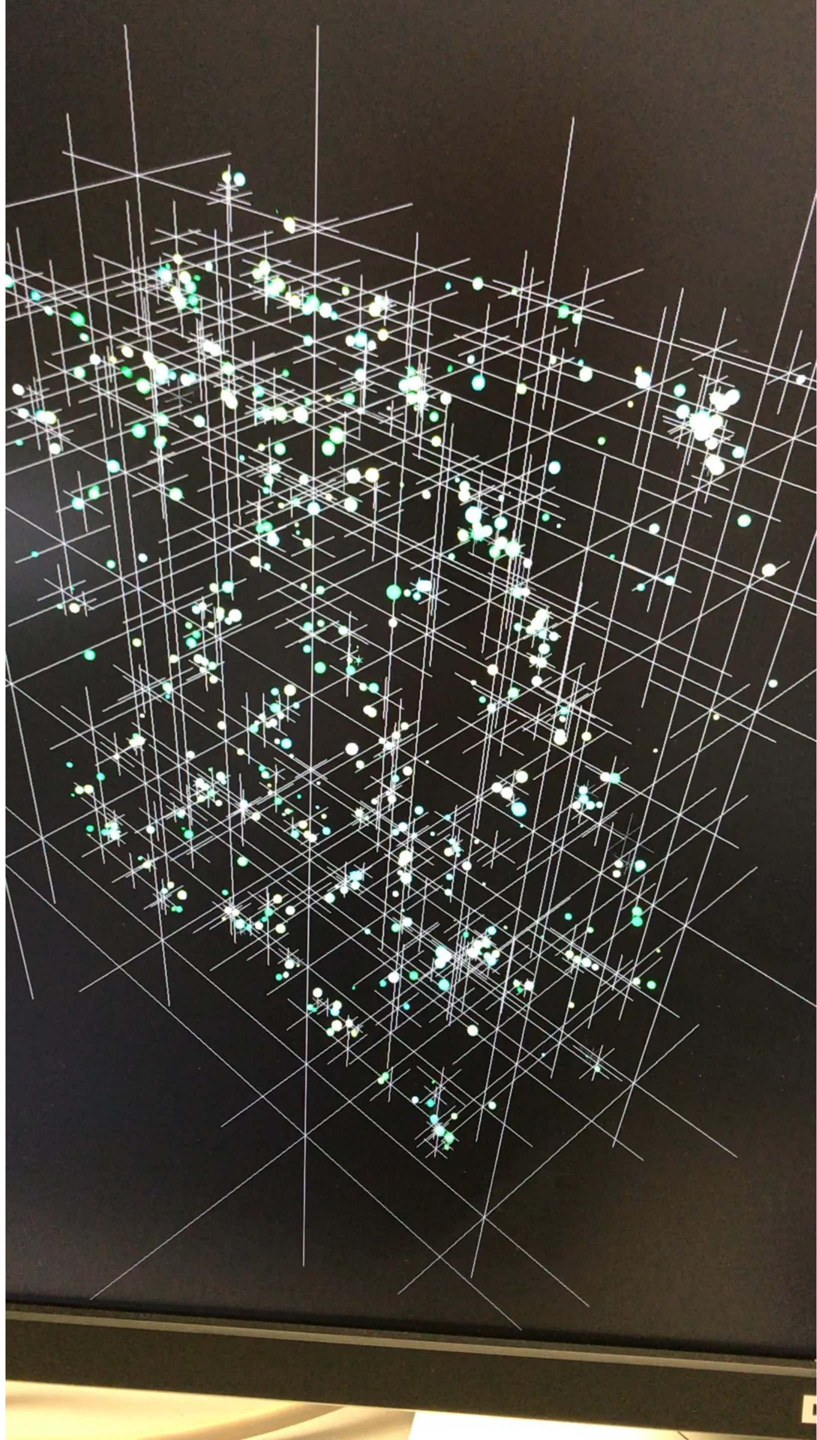
- Questions?
- Administrivia
  - Impending (minor) schedule changes
    - FPGA readings
    - Moved FPGA Lab Due Date
    - Barnes-Hut status change
  - Exam next week
- Agenda
  - CUDA
  - CUDA Performance
  - GPU parallel algorithms redux redux

Acknowledgements:

- http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx
- http://www.seas.upenn.edu/~cis565/LECTURES/CUDA%20Tricks.pptx
- http://www.cs.utexas.edu/~pingali/CS378/2015sp/lectures/GPU%20Programming.pptx

## Schedule Stuff

- Midterm Quiz questions posted soon



## Faux Quiz Questions

- How is occupancy defined (in CUDA nomenclature)?
- What's the difference between a block scheduler (e.g. Giga-Thread Engine) and a warp scheduler?
- Modern CUDA supports UVM to eliminate the need for cudaMalloc and cudaMemcpy*. Under what conditions might you want to use or not use it and why?
- What is control flow divergence? How does it impact performance?
- What is a bank conflict?
- What is work efficiency?
- What is the difference between a thread block scheduler and a warp scheduler?
- How are atomics implemented in modern GPU hardware?
- How is __shared__ memory implemented by modern GPU hardware?
- Why is __shared__ memory necessary if GPUs have an L1 cache? When will an L1 cache provide all the benefit of __shared__ memory and when will it not?
- Is cudaDeviceSynchronize still necessary after copyback if I have just one CUDA stream?

# Review: Blocks and Threads

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
```

- Mos
  - Ir

- Why have threads?
  - Why not just blocks or just threads?
- Unlike parallel blocks, threads can:
  - Communicate
  - Synchronize

```
>(d_a, d_b, d_c, N);
```

`blockIdx.x = 0`   `blockIdx.x = 1`   `blockIdx.x = 2`   `blockIdx.x = 3`

- With M threads/block, unique index per thread is :

```
int index = threadIdx.x + blockIdx.x * M;
```

What if my array size N % M != 0 !!???

## How many threads/blocks should I use?

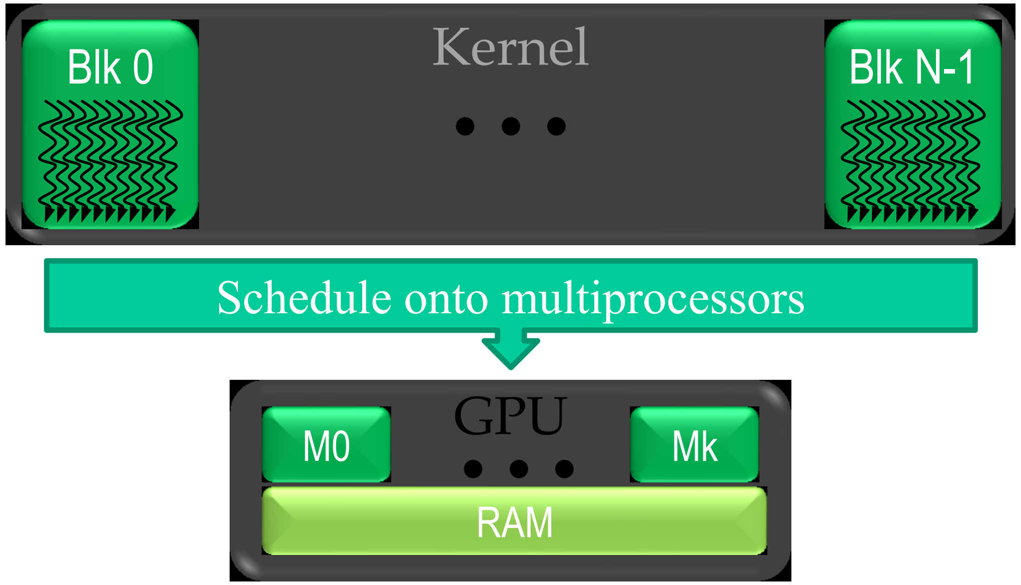
```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add()
    add<                                              >(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

- Usually things are correct if grid*block dims >= input size
- Getting good performance is another matter

# Internals

```
__host__
void vecAdd()
{
  dim3 DimGrid = (ceil(n/256,1,1);
  dim3 DimBlock = (256,1,1);
  addKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}
```

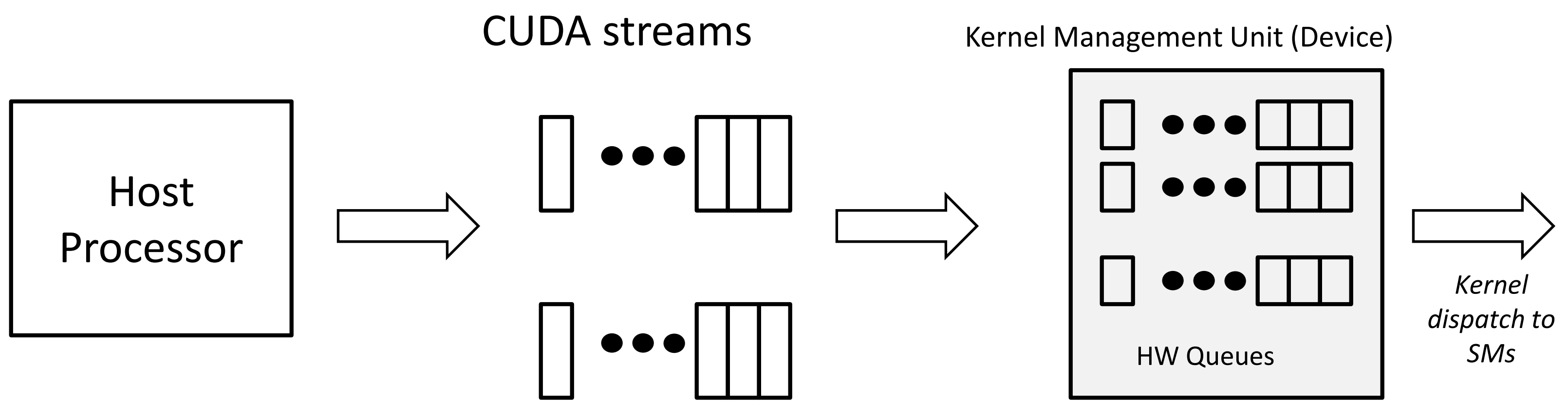
```
__global__
void addKernel(float *A_d,
               float *B_d,
               float *C_d,
               int n){
   int i = blockIdx.x * blockDim.x
             + threadIdx.x;
   if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```



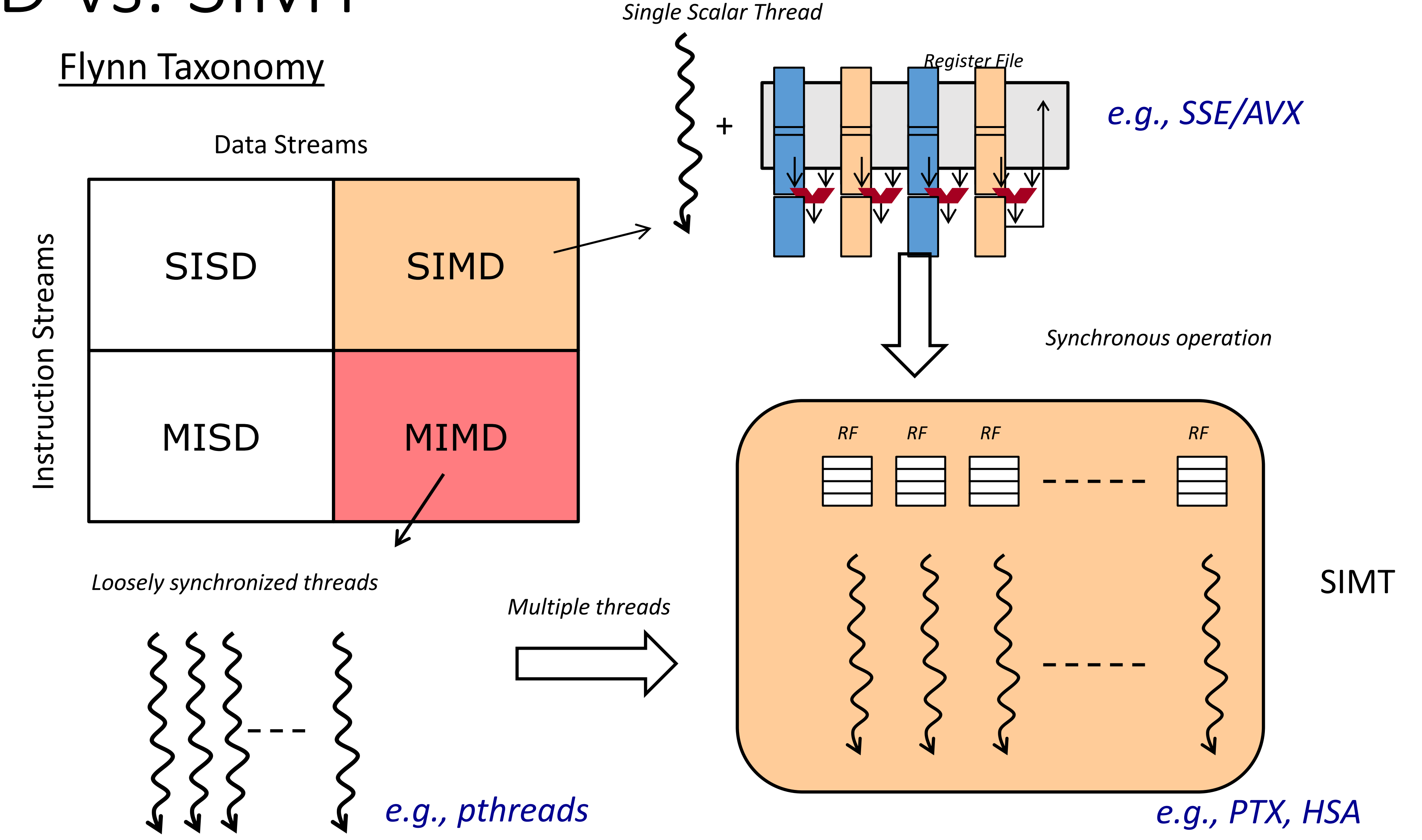
How are threads scheduled?

## Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams mapped to GPU HW queues
  - Done by “kernel management unit” (KMU)
  - Multiple streams mapped to each queue → serializes some kernels
- Kernel launch distributes thread blocks to SMs



# SIMD vs. SIMT


Flynn Taxonomy
Data Streams
Instruction Streams
SISD
SIMD
MISD
MIMD
Single Scalar Thread
+
Register File
e.g., SSE/AVX
Synchronous operation
RF
RF
RF
RF
SIMT
Loosely synchronized threads
Multiple threads
e.g., pthreads
e.g., PTX, HSA

## GPU Performance Metric: ***Occupancy***

- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enables device-specific online tuning of kernel parameters

Shouldn't we just create as many threads as possible?

PCI Express 3.0 Host Interface
GigaThread Engine
L2 Cache
GAMERS NEXUS
High-Speed Hub
NVLink

# A Taco Bar

- Where is the parallelism here?

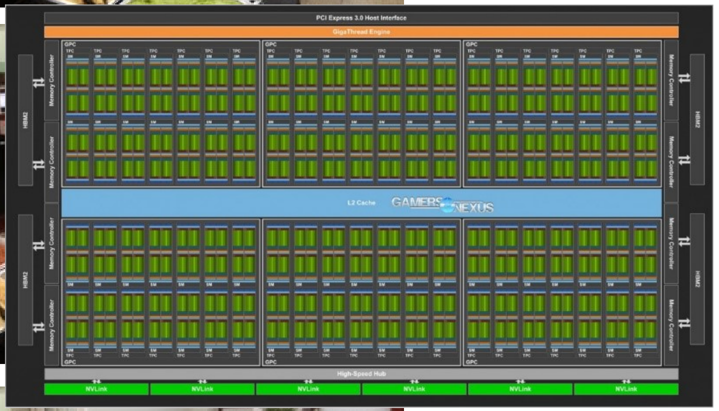
# GPU: a multi-lane Taco Bar

- Where is the parallelism here?

# GPU: a multi-lane Taco Bar

- Where is the parallelism here?

- There's none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: ***Increase Occupancy!***



# GPU: a multi-lane Taco Bar

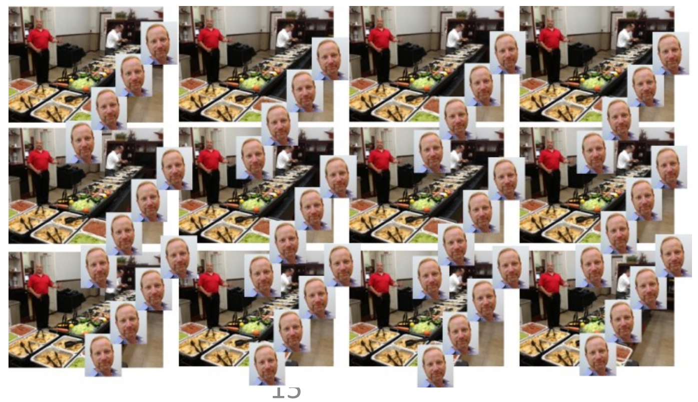
- Where is the parallelism here?

- There’s none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: ***Increase Occupancy!***

## GPU Performance Metric: ***Occupancy***

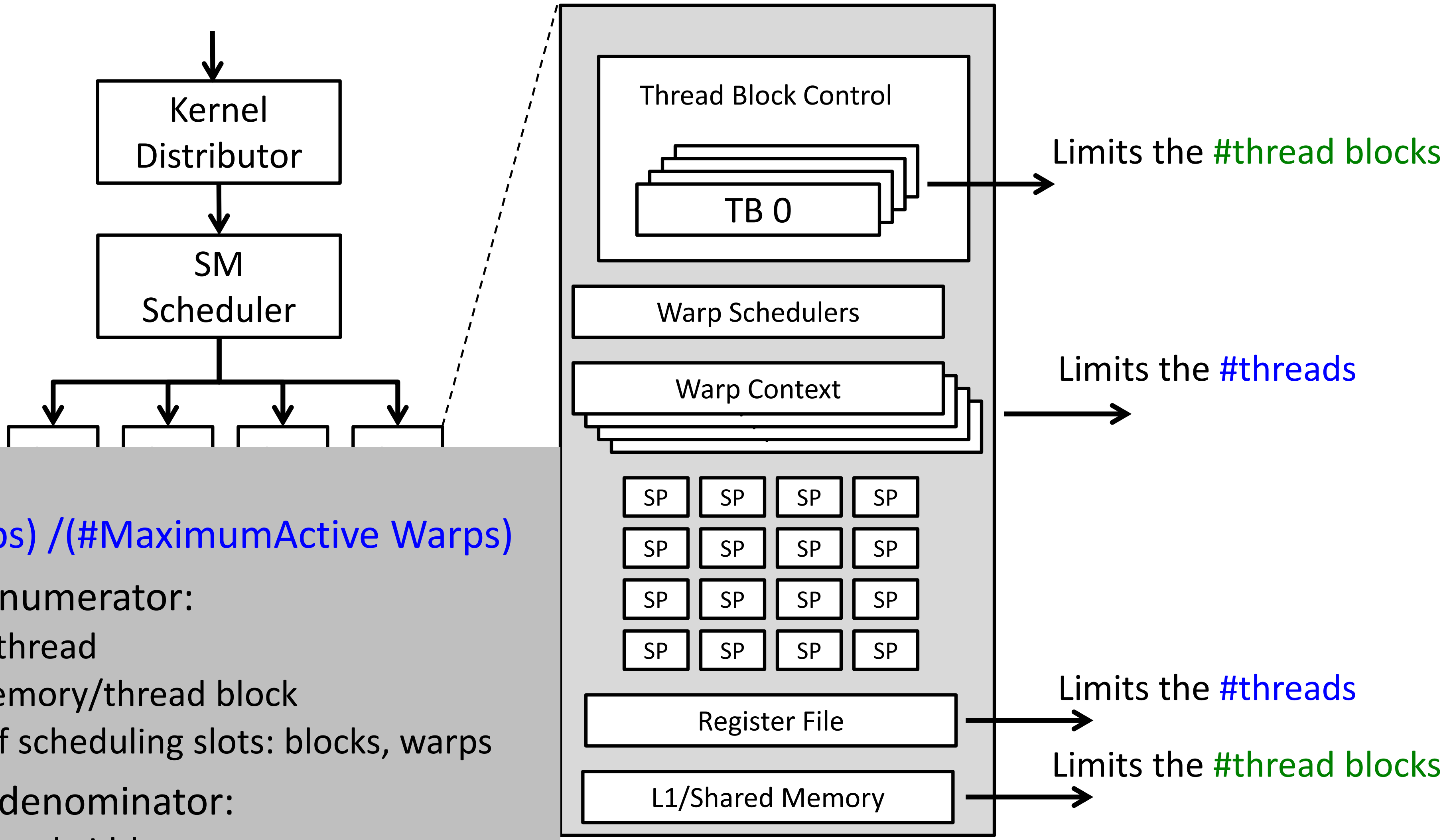
- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enables device-specific online tuning of kernel parameter

Shouldn’t we just create as many threads as possible?



PCI Express 3.0 Host Interface
GigaThread Engine
L2 Cache
GAMERS NEXUS
High-Speed Hub
NVLink

# Hardware Resources Are Finite

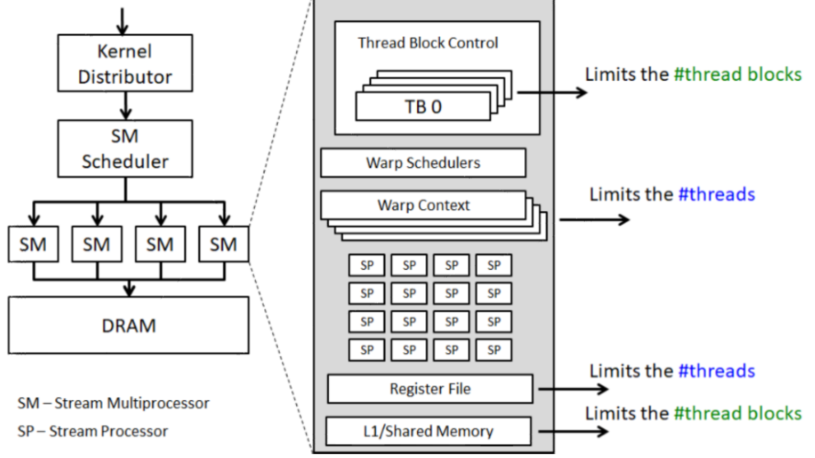


Occupancy:

- (#Active Warps) /(#MaximumActive Warps)
- Limits on the numerator:
  - Registers/thread
  - Shared memory/thread block
  - Number of scheduling slots: blocks, warps
- Limits on the denominator:
  - Memory bandwidth
  - Scheduler slots

What is the performance impact of varying kernel resource demands?

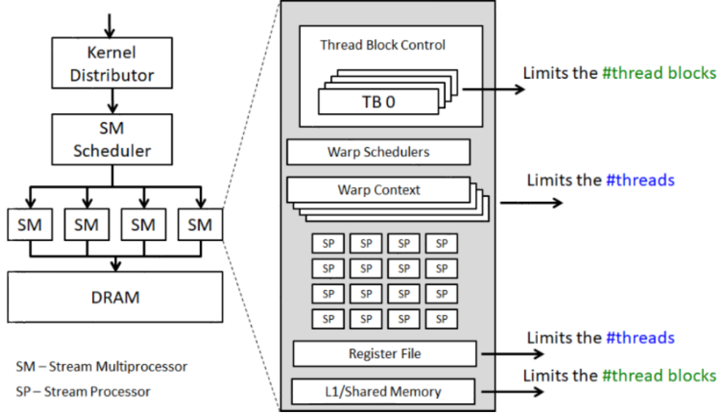
# Impact of Thread Block Size



Example: v100:

- max active warps/SM == 64 (limit: warp context)
- max active blocks/SM == 32 (limit: block control)
  - With 512 threads/block how many blocks can execute (per SM) concurrently?
  - Max active warps * threads/warp = 64*32 = 2048 threads → 4
  - With 128 threads/block? → 16
- Consider HW limit of 32 thread blocks/SM @ 32 threads/block:
  - Blocks are maxed out, but max active threads = 32*32 = 1024
  - Occupancy = .5 (1024/2048)
- To maximize utilization, thread block size should balance
  - Limits on active thread blocks vs.
  - Limits on active warps

## Impact of #Registers Per Thread



Registers/thread can limit number of active threads!

V100:

- Registers per thread max: 255
- 64K registers per SM

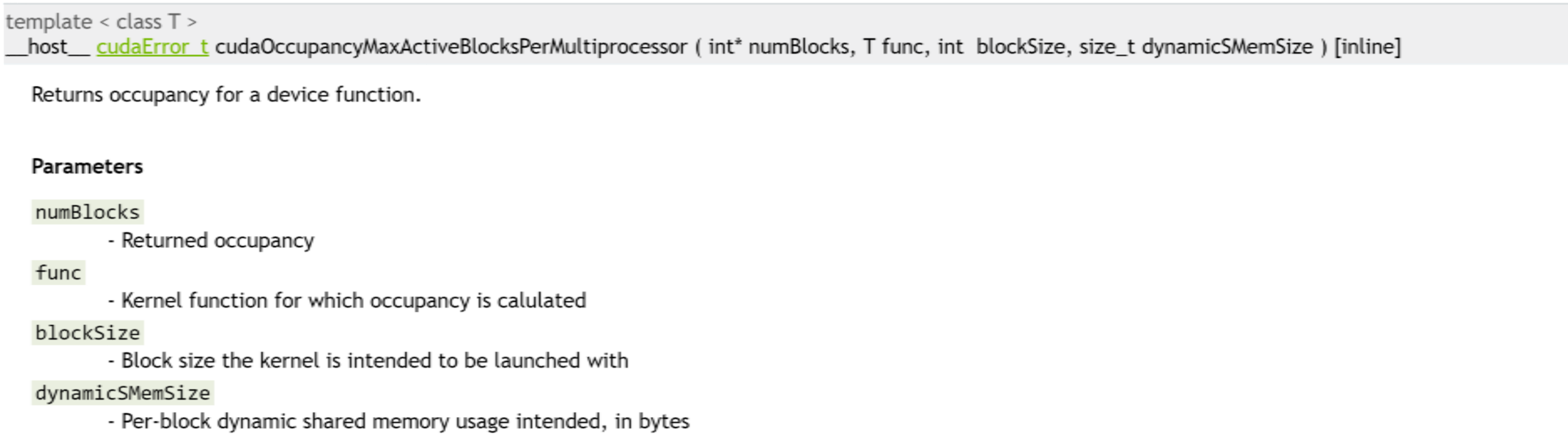
Assume a kernel uses 32 registers/thread, thread block size of 256

- Thus, A TB requires 8192 registers for a maximum of 8 thread blocks per SM
    - Uses all 2048 thread slots (8 blocks * 256 threads/block)
    - 8192 *regs/block * 8 block/SM = 64k registers*
    - *FULLY Occupied!*
- What is the impact of increasing number of registers by 2?
    - Recall: granularity of management is a thread block!
    - Loss of concurrency of 256 threads!
    - *34 regs/thread * 256 threads/block * 7 blocks/SM = 60k registers,*
    - *8 blocks would over-subscribe register file*
    - *Occupancy drops to .875!*

## Impact of Shared Memory

- Shared memory is allocated per thread block
    - Can limit the number of thread blocks executing concurrently per SM
    - Shared mem/block * # blocks <= total shared mem per SM
- gridDim and blockDim parameters impact demand for
    - shared memory
    - number of thread slots
    - number of thread block slots

## Balance



```
template < class T >
__host__ cudaError_t cudaOccupancyMaxActiveBlocksPerMultiprocessor ( int* numBlocks, T func, int  blockSize, size_t dynamicSMemSize ) [inline]
```

Returns occupancy for a device function.

**Parameters**

| | |
|---|---|
| `numBlocks` | - Returned occupancy |
| `func` | - Kernel function for which occupancy is calulated |
| `blockSize` | - Block size the kernel is intended to be launched with |
| `dynamicSMemSize` | - Per-block dynamic shared memory usage intended, in bytes |

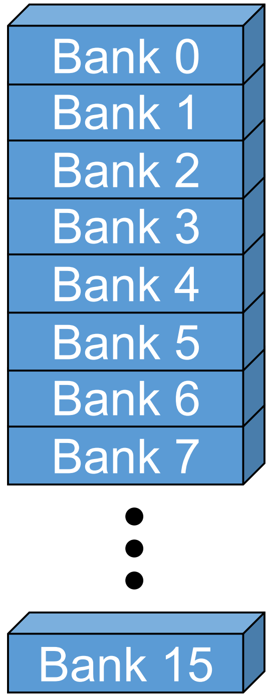
- Navigate the tradeoffs
  - maximize core utilization and memory bandwidth utilization
  - Device-specific
- Goal: Increase occupancy until one or the other is saturated

## Parallel Memory Accesses

- Coalesced main memory access (16/32x faster)
  - HW combines multiple warp memory accesses into a single coalesced access
- Bank-conflict-free shared memory access (16/32)
  - No alignment or contiguity requirements
    - CC 1.3: 16 different banks per half warp or same word
    - CC 2.x+3.0 : 32 different banks + 1-word broadcast each

# Parallel Memory Architecture

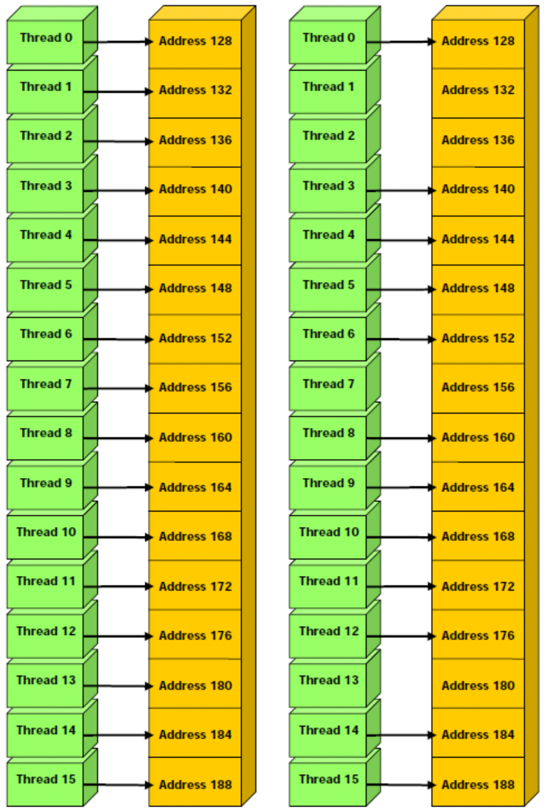
- In a parallel machine, many threads access memory
    - Therefore, memory is divided into banks
    - Essential to achieve high bandwidth
- Each bank can service one address per cycle
    - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
    - Conflicting accesses are serialized



## Coalesced Main Memory Accesses

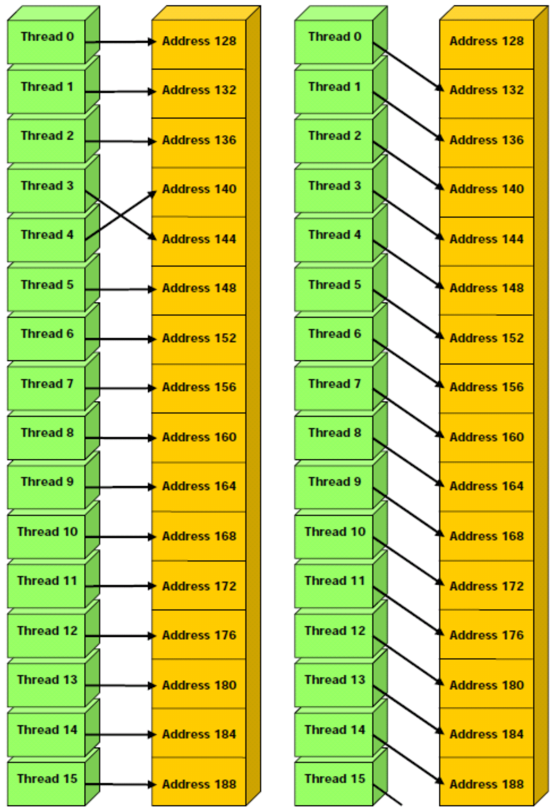
single coalesced access

NVIDIA



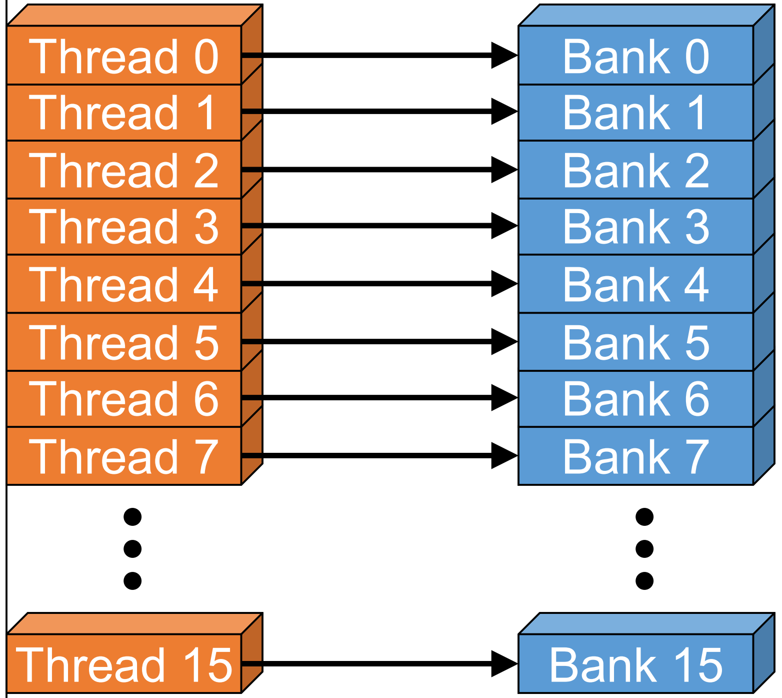
one and two coalesced accesses*

NVIDIA

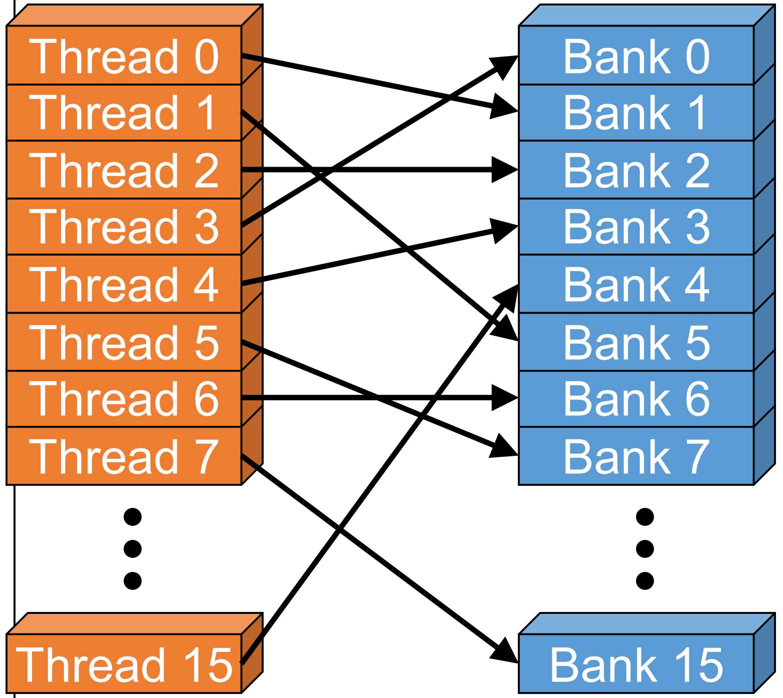


# Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

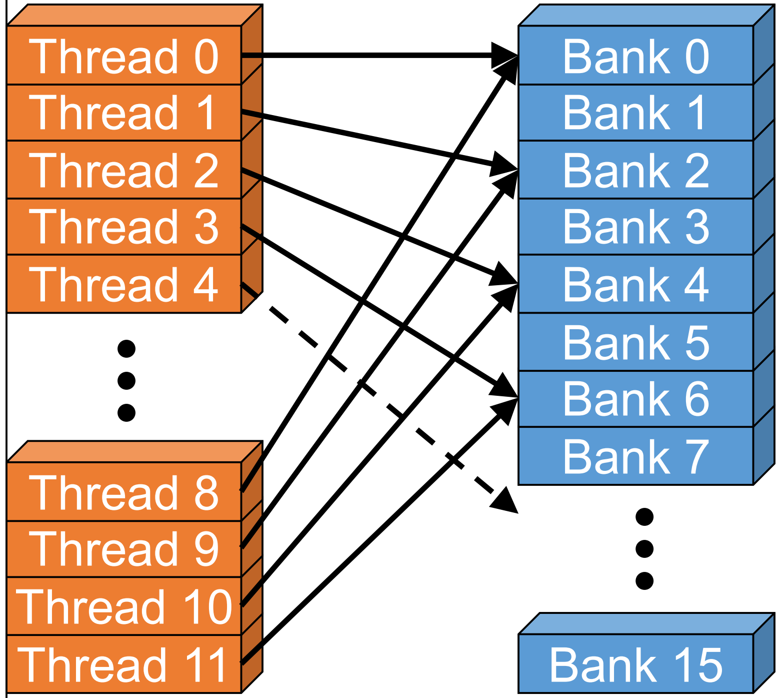


- No Bank Conflicts
  - Random 1:1 Permutation

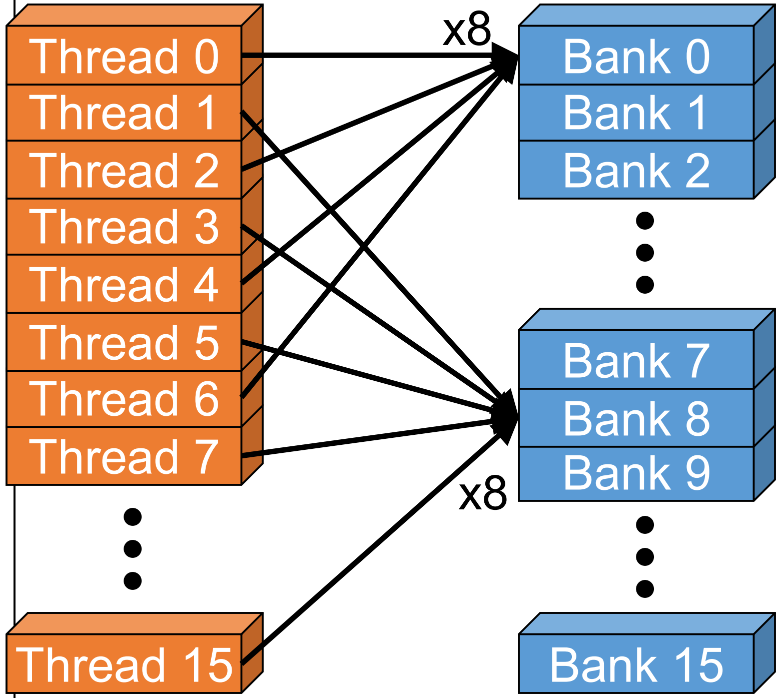


# Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride == 2



- 8-way Bank Conflicts
  - Linear addressing stride == 8

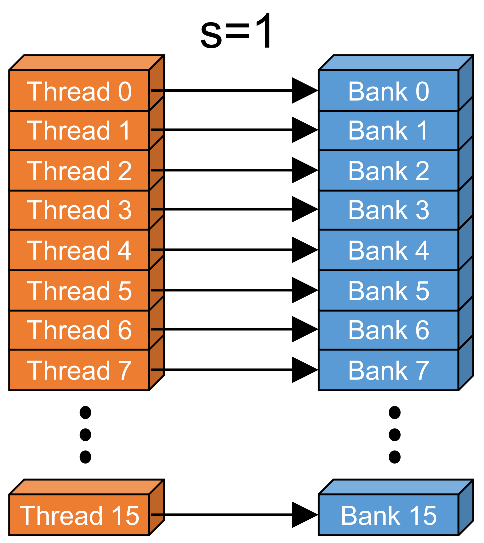


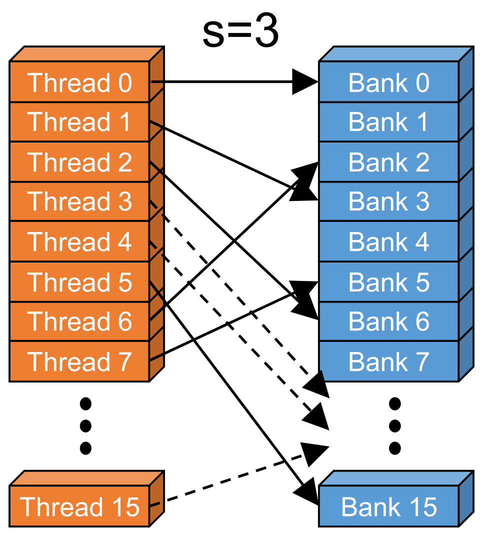
# Linear Addressing

- Given:

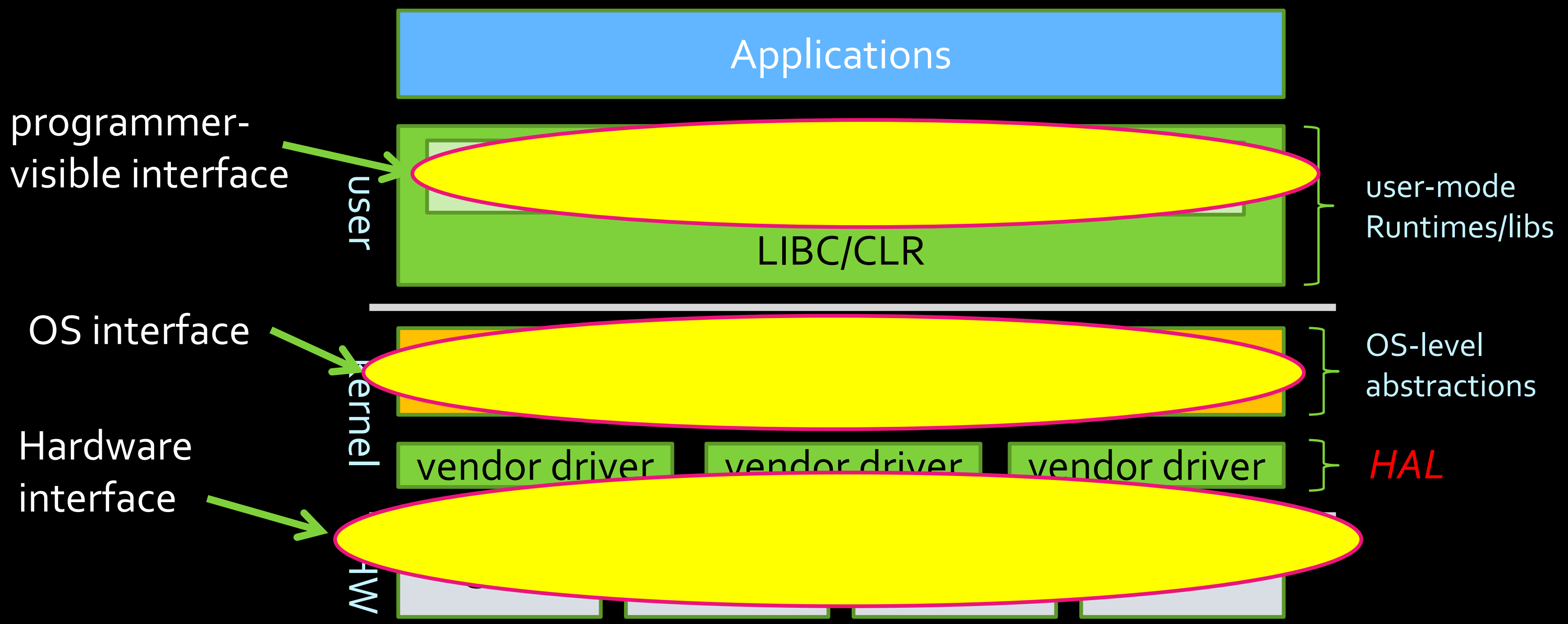
```
__shared__ float shared[256];
float foo =
  shared[baseIndex + s *
   threadIdx.x];
```

- This is only bank-conflict-free if s shares no common factors with the number of banks
  - 16 on G80, so `s` must be odd



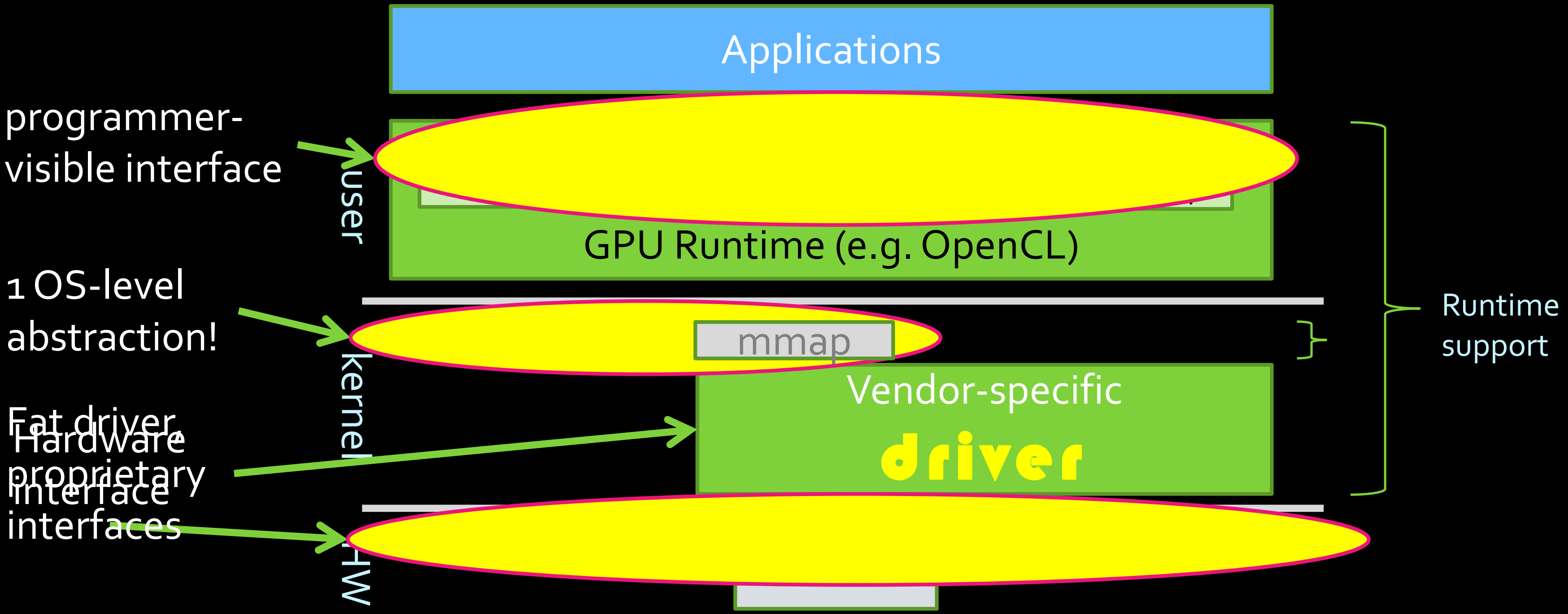


## Layered abstractions



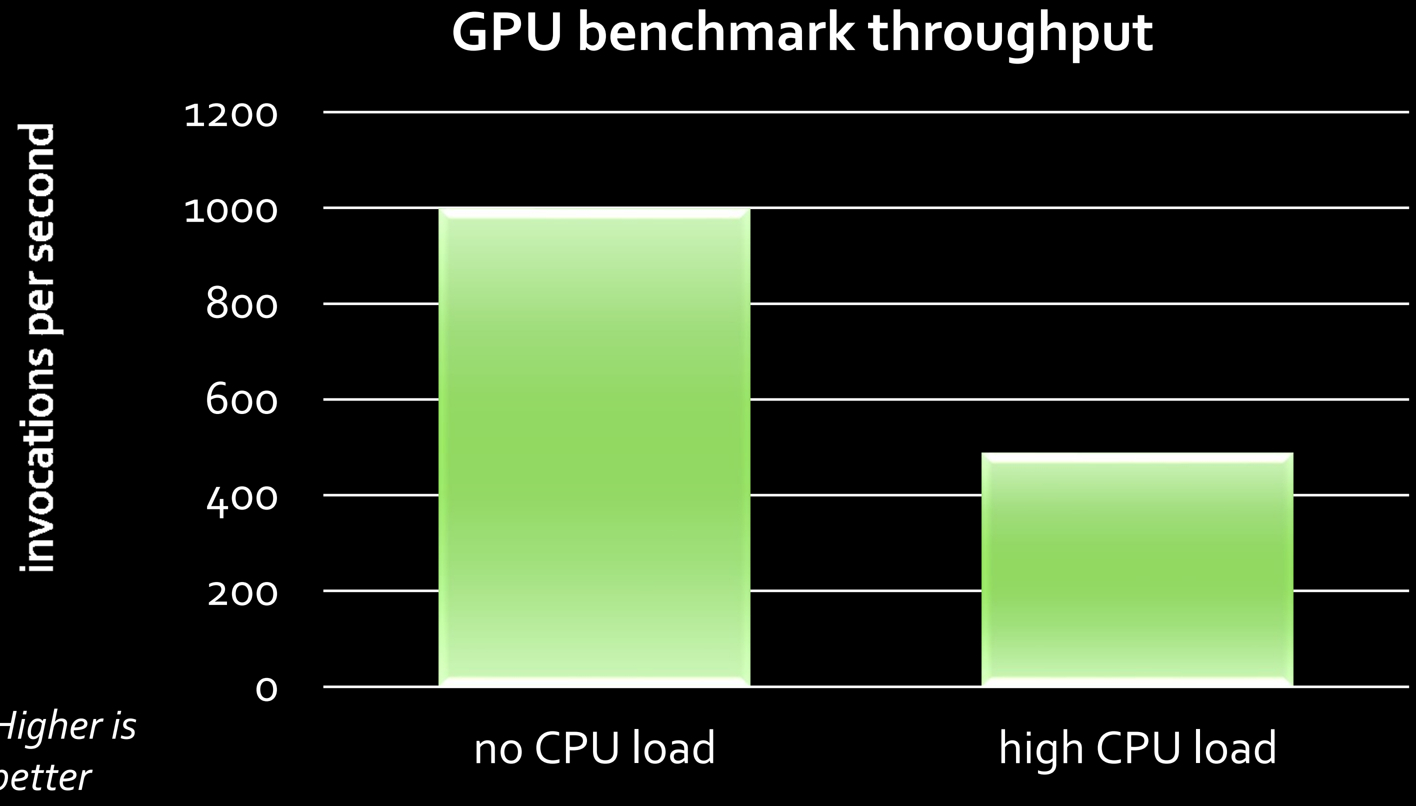
** 1:1 correspondence between OS-level and user-level abstractions*
** Diverse HW support enabled HAL*

# GPU abstractions



1. No kernel-facing API
2. OS resource-management limited
3. *Poor composability*

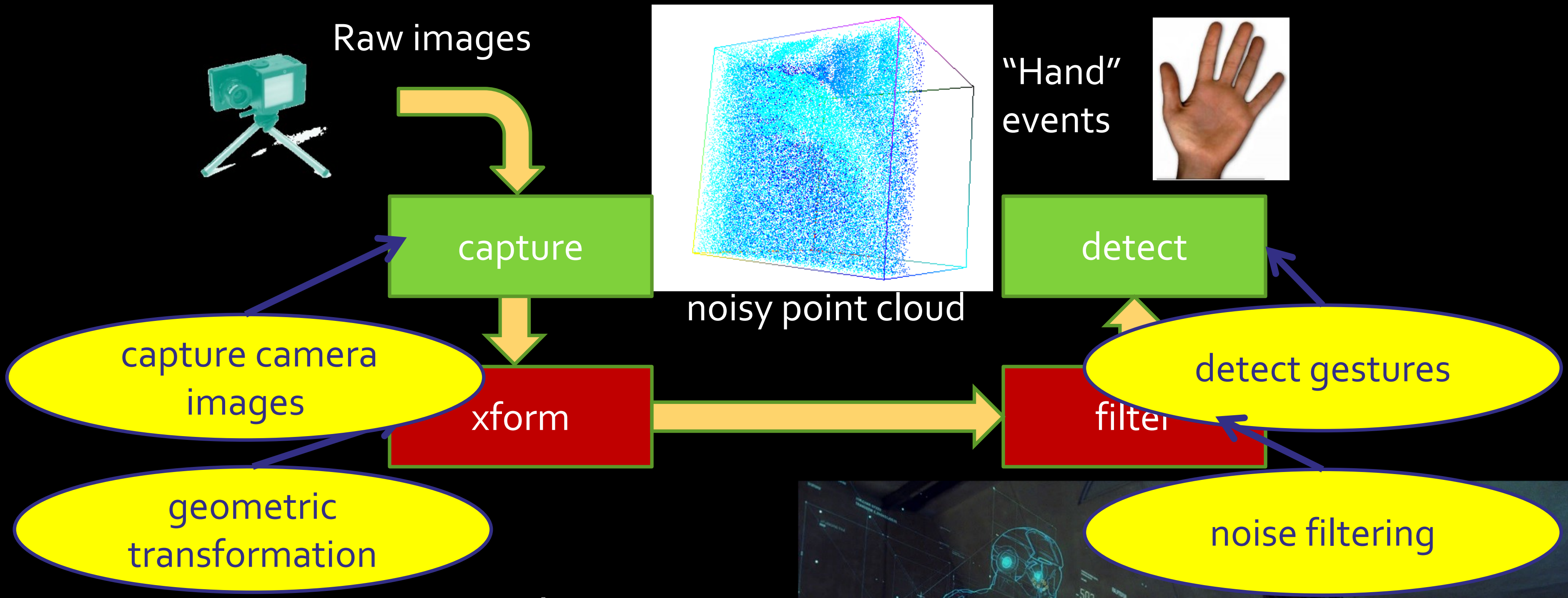
# No OS support → No isolation



CPU+GPU schedulers not integrated!
…other pathologies abundant

ge-convolution in CUDA
dows 7 x64 8GB RAM
l Core 2 Quad 2.66GHz
dia GeForce GT230

# Composition: Gestural Interface



- Requires OS mediation
- High data rates
- Abundant data parallelism

...use GPUs!

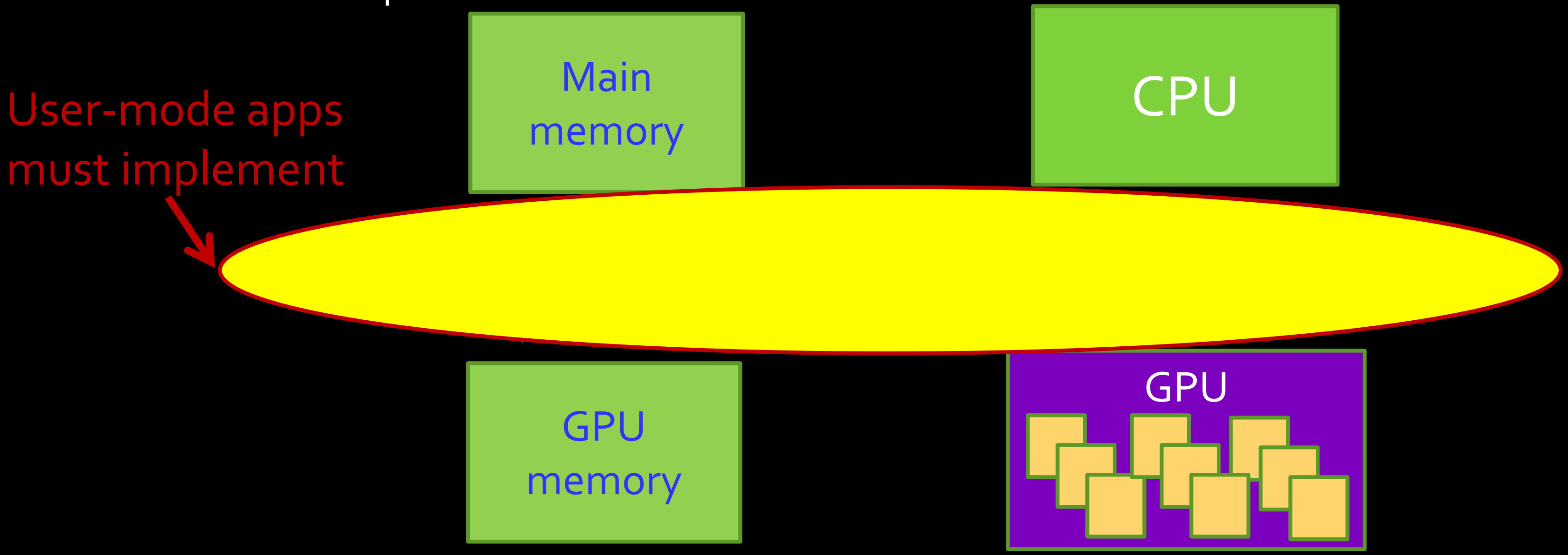
## What We'd Like To Do

```
#> capture | xform | filter | detect &
     CPU      GPU     GPU      CPU
```

- Modular design
  - flexibility, reuse
- Utilize heterogeneous hardware
  - Data-parallel components → GPU
  - Sequential components → CPU
- Using OS provided tools
  - processes, pipes

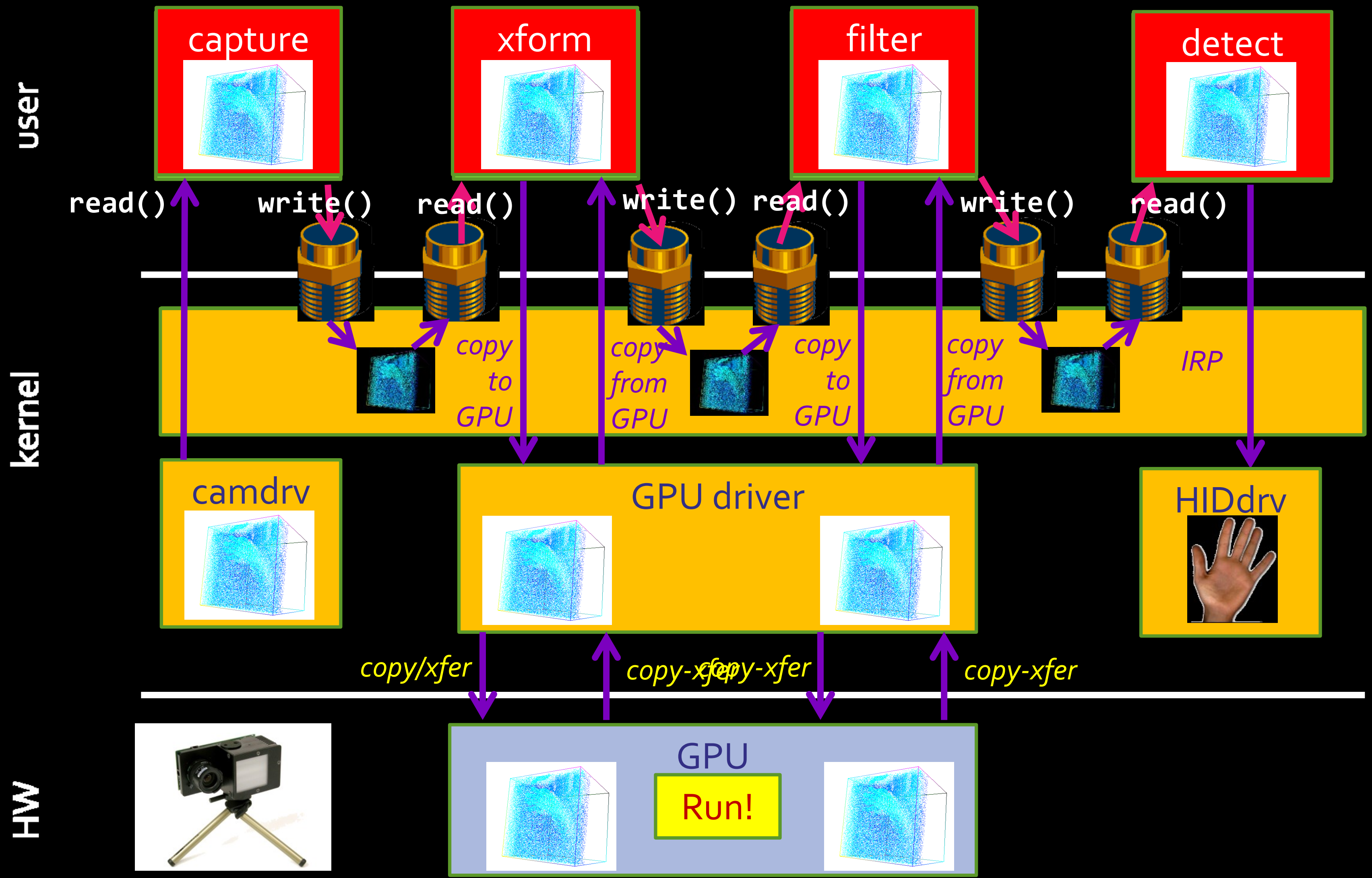
# GPU Execution model

- GPUs cannot run OS:
    - different ISA
    - Memories have different coherence guarantees
        - **(disjoint, or require fence instructions)**
- Host CPU must “manage” GPU execution
    - Program inputs explicitly transferred/bound at runtime
    - Device buffers pre-allocated



# Data migration

```
#> capture | xform | filter | detect &
```



3/8/2020

## Device-centric APIs considered harmful

```
Matrix
gemm(Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

*What happens if I want the following?*
*Matrix D = A x B x C*

## Composed matrix multiplication

```
Matrix
gemm(Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

```
Matrix
AxBxC(Matrix A, B, C) {
    Matrix AxB = gemm(A,B);
    Matrix AxBxC = gemm(AxB,C);
    return AxBxC;
}
```

# Composed matrix multiplication

AxB copied from GPU memory…

```
Matrix
AxBxC(Matrix A, B, C) {
    Matrix [illegible]= gemm(A,B);
    Matrix AxBxC = gemm(AxB,C);
    return AxBxC;
}
```

```
Matrix
gemm(Matrix A, Matrix B) {
   copyToGPU(A);
   copyToGPU(B);
   invokeGPU();
   M[illegible] Matrix();
   [illegible]
   re[illegible]
}
```

# Composed matrix multiplication

```
Matrix
gemm([redacted] Matrix B) {
[redacted]
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

```
Matrix
AxBxC(Matrix A, B, C) {
    Matrix AxB = gemm(A,B);
    Matrix AxBxC = gemm([redacted]C);
    return AxBxC;
}
```

…only to be copied right back!

## What if I have many GPUs?

```
Matrix
gemm(Matrix A, Matrix B) {
   copyToGPU(A);
   copyToGPU(B);
   invokeGPU();
   Matrix C = new Matrix();
   copyFromGPU(C);
   return C;
}
```

## What if I have many GPUs?

```
Matrix
gemm([redacted],Matrix A, Matrix B) {
    copyToGPU(dev, A);
    copyToGPU(dev, B);
    invokeGPU(dev);
    Matrix C = new Matrix();
    copyFromGPU(dev, C);
    return C;
}
```

*What happens if I want the following?*

*Matrix D = A x B x C*

## Composition with many GPUs

```
Matrix
gemm(GPU dev, Matrix A, Matrix B)
{
   copyToGPU(A);
   copyToGPU(B);
   invokeGPU();
   Matrix C = new Matrix();
   copyFromGPU(C);
   return C;
}
```

```
Matrix
AxBxC(Matrix A,B,C) {
   Matrix AxB = gemm          ,B);
   Matrix AxBxC = ge           AxB,C);
   return AxBxC;
}
```

# Composition with many GPUs

```
Matrix
gemm(GPU dev, Matrix A, Matrix B)
{
   copyToGPU(A);
   copyToGPU(B);
   invokeGPU();
   Matrix C = new Matrix();
   copyFromGPU(C);
   return C;
}
```

Rats…now I can only use 1 GPU. *How to partition computation?*

```
Matrix
AxBxC(GPU dev, Matrix A,B,C) {
   Matrix AxB = gemm(dev, A,B);
   Matrix AxBxC = gemm(dev, AxB,C);
   return AxBxC;
}
```

## Composition with many GPUs

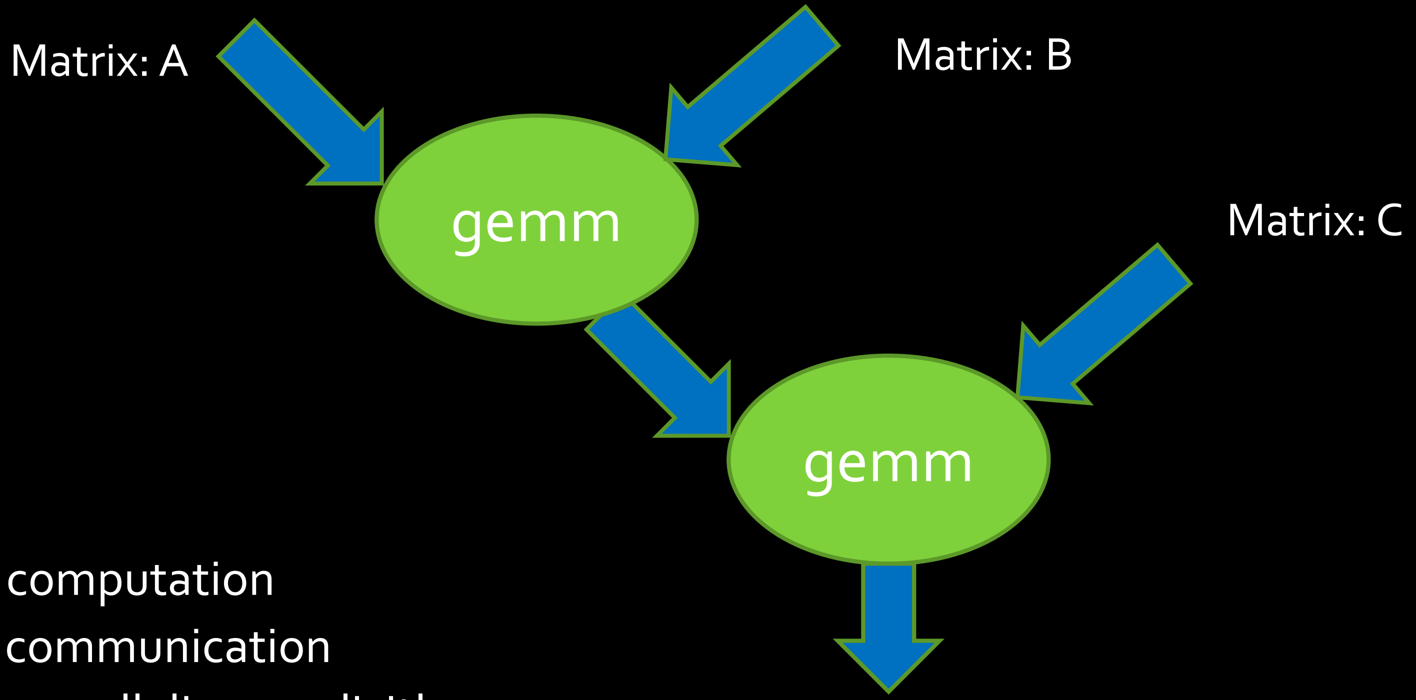
```
Matrix
gemm(GPU dev, Matrix A, Matrix B)
{
   copyToGPU(A);
   copyToGPU(B);
   invokeGPU();
   Matrix C = new Matrix();
   copyFromGPU(C);
   return C;
}
```

This will never be manageable for *many* GPUs. *Programmer implements scheduling using static view!*

```
Matrix
AxBxC(GPU devA, GPU devB, Matrix A,B,C) {
   Matrix AxB = gemm(devA, A,B);
   Matrix AxBxC = gemm(devB, AxB,C);
   return AxBxC;
}
```

Why don't we have this problem with CPUs?

## Dataflow: a better abstraction



- nodes → computation
- edges → communication
- Expresses parallelism explicitly
- Minimal specification of data movement: runtime does it.
- asynchrony is a runtime concern (not programmer concern)
- No specification of compute→device mapping: like threads!

## Advanced topics: Prefix-Sum

- in: 3 1 7 0 4 1 6 3
- out: 0 3 4 11 11 14 16 22

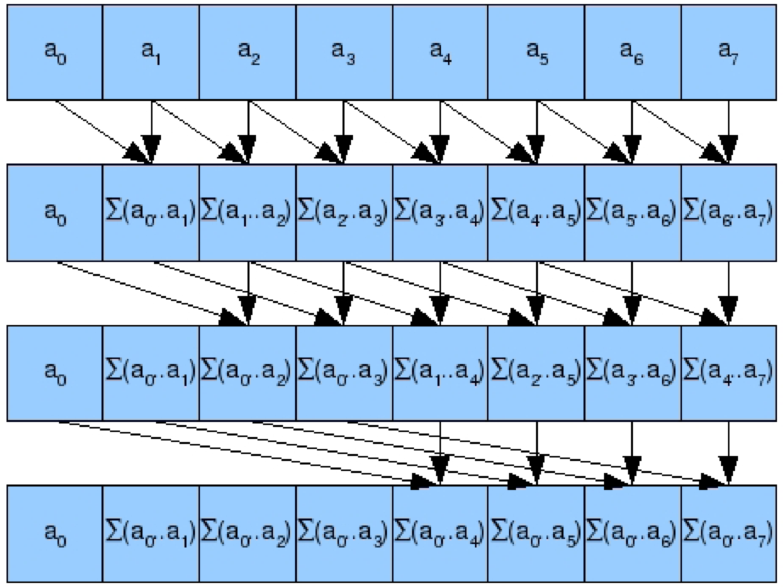
## Trivial Sequential Implementation

```
void scan(int* in, int* out, int n)
{
        out[0] = 0;
        for (int i = 1; i < n; i++)
                out[i] = in[i-1] + out[i-1];
}
```

# Parallel Scan

```
for(d = 1; d < $\log_2 n$; d++)
   for all k in parallel
     if( k >= $2^d$ )
        x[out][k] = x[in][k – $2^{d-1}$] + x[in][k]
     else
        x[out][k] = x[in][k]
```

Complexity O($n\log_2 n$)



$a_0$ $a_1$ $a_2$ $a_3$ $a_4$ $a_5$ $a_6$ $a_7$
$a_0$ $\Sigma(a_0..a_1)$ $\Sigma(a_1..a_2)$ $\Sigma(a_2..a_3)$ $\Sigma(a_3..a_4)$ $\Sigma(a_4..a_5)$ $\Sigma(a_5..a_6)$ $\Sigma(a_6..a_7)$
$a_0$ $\Sigma(a_0..a_1)$ $\Sigma(a_0..a_2)$ $\Sigma(a_0..a_3)$ $\Sigma(a_1..a_4)$ $\Sigma(a_2..a_5)$ $\Sigma(a_3..a_6)$ $\Sigma(a_4..a_7)$
$a_0$ $\Sigma(a_0..a_1)$ $\Sigma(a_0..a_2)$ $\Sigma(a_0..a_3)$ $\Sigma(a_0..a_4)$ $\Sigma(a_0..a_5)$ $\Sigma(a_0..a_6)$ $\Sigma(a_0..a_7)$

# A work efficient parallel scan

- Goal is a parallel scan that is O($n$) instead of O($n\log_2 n$)
- Solution:
  - Balanced Trees: Build a binary tree, sweep it to and from the root.
  - Binary tree with $n$ leaves has
    - $d=\log_2 n$ levels,
    - each level $d$ has $2^d$ nodes

    * One add is performed per node → O($n$) add on a single traversal of the tree.

# O(n) unsegmented scan

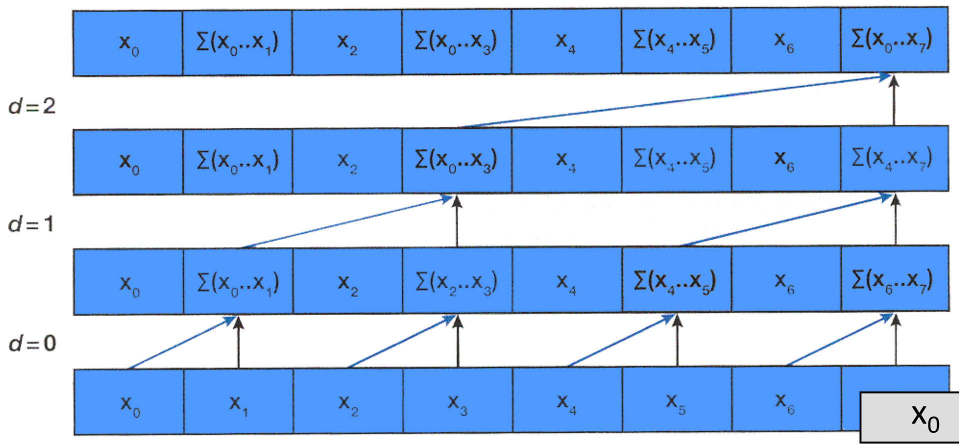
- Reduce/Up-Sweep

```
for(d = 0; d < $\log_2 n$-1; d++)
   for all k=0; k < n-1; k+=$2^{d+1}$ in parallel
     x[k+$2^{d+1}$-1] = x[k+$2^d$-1] + x[k+$2^{d+1}$-1]
```

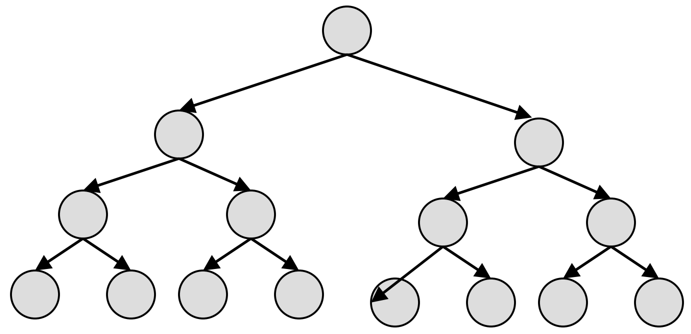
- Down-Sweep

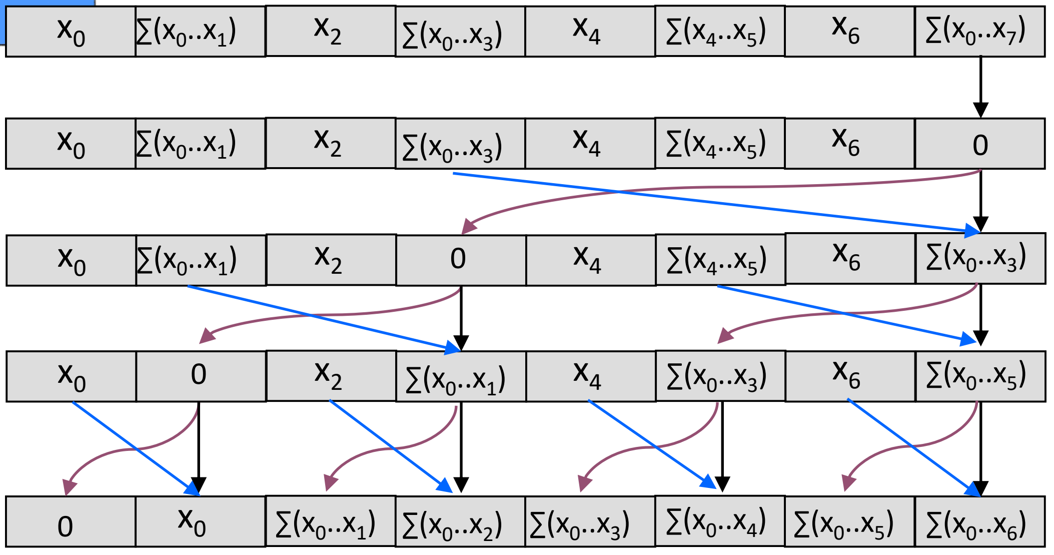
```
x[n-1] = 0;
for(d = $\log_2 n$ – 1; d >=0; d--)
   for all k = 0; k < n-1; k += $2^{d+1}$ in parallel
      t = x[k + $2^d$ – 1]
      x[k + $2^d$ - 1] = x[k + $2^{d+1}$ -1]
      x[k + $2^{d+1}$ - 1] = t + x[k + $2^{d+1}$ – 1]
```

# Tree analogy



$x_0$ | $\Sigma(x_0..x_1)$ | $x_2$ | $\Sigma(x_0..x_3)$ | $x_4$ | $\Sigma(x_4..x_5)$ | $x_6$ | $\Sigma(x_0..x_7)$
$d=2$
$x_0$ | $\Sigma(x_0..x_1)$ | $x_2$ | $\Sigma(x_0..x_3)$ | $x_4$ | $\Sigma(x_4..x_5)$ | $x_6$ | $\Sigma(x_4..x_7)$
$d=1$
$x_0$ | $\Sigma(x_0..x_1)$ | $x_2$ | $\Sigma(x_2..x_3)$ | $x_4$ | $\Sigma(x_4..x_5)$ | $x_6$ | $\Sigma(x_6..x_7)$
$d=0$
$x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$





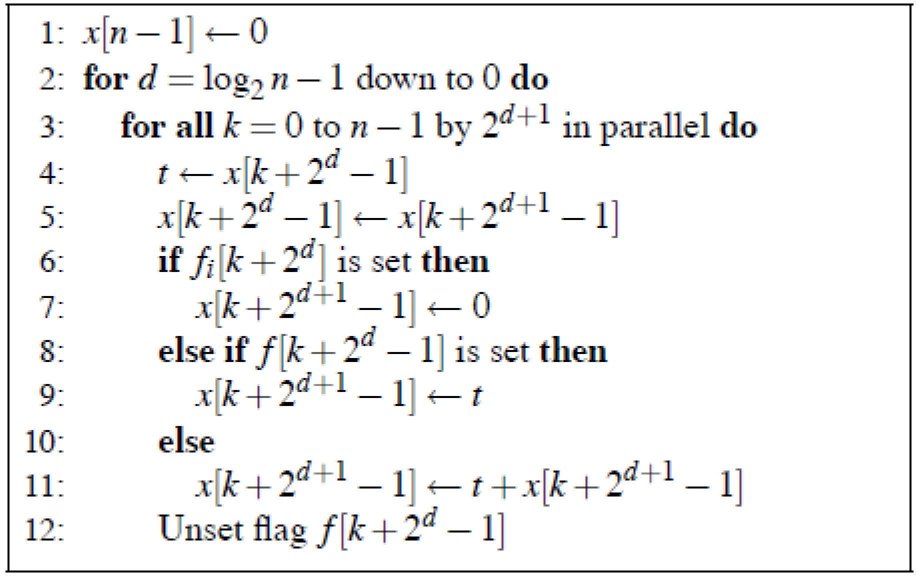
$x_0$ | $\Sigma(x_0..x_1)$ | $x_2$ | $\Sigma(x_0..x_3)$ | $x_4$ | $\Sigma(x_4..x_5)$ | $x_6$ | $\Sigma(x_0..x_7)$
$x_0$ | $\Sigma(x_0..x_1)$ | $x_2$ | $\Sigma(x_0..x_3)$ | $x_4$ | $\Sigma(x_4..x_5)$ | $x_6$ | 0
$x_0$ | $\Sigma(x_0..x_1)$ | $x_2$ | 0 | $x_4$ | $\Sigma(x_4..x_5)$ | $x_6$ | $\Sigma(x_0..x_3)$
$x_0$ | 0 | $x_2$ | $\Sigma(x_0..x_1)$ | $x_4$ | $\Sigma(x_0..x_3)$ | $x_6$ | $\Sigma(x_0..x_5)$
0 | $x_0$ | $\Sigma(x_0..x_1)$ | $\Sigma(x_0..x_2)$ | $\Sigma(x_0..x_3)$ | $\Sigma(x_0..x_4)$ | $\Sigma(x_0..x_5)$ | $\Sigma(x_0..x_6)$

## O(n) Segmented Scan

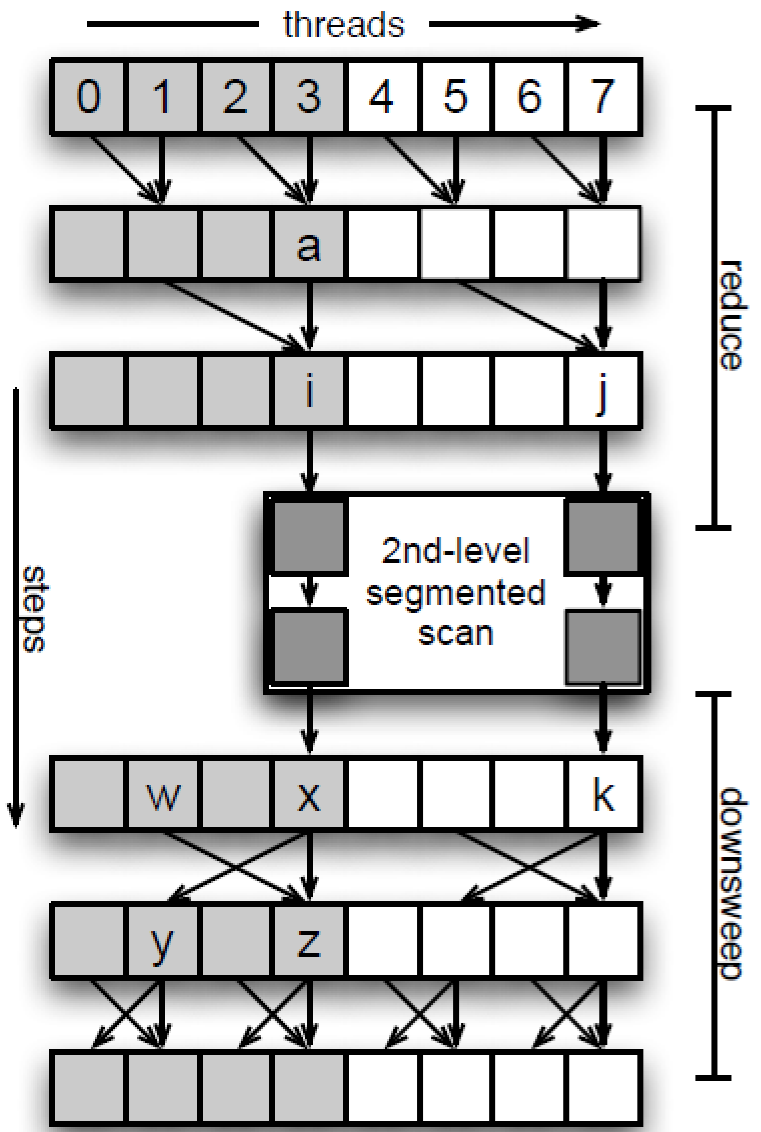
**Up-Sweep**

```
for $d = 1$ to $\log_2 n - 1$ do
    for all $k = 0$ to $n - 1$ by $2^{d+1}$ in parallel do
        if $f[k + 2^{d+1} - 1]$ is not set then
            $x[k + 2^{d+1} - 1] \leftarrow x[k + 2^{d} - 1] + x[k + 2^{d+1} - 1]$
        $f[k + 2^{d+1} - 1] \leftarrow f[k + 2^{d} - 1] \mid f[k + 2^{d+1} - 1]$
```

- Down-Sweep



```
$x[n-1] \leftarrow 0$
for $d = \log_2 n - 1$ down to 0 do
    for all $k = 0$ to $n-1$ by $2^{d+1}$ in parallel do
        $t \leftarrow x[k + 2^d - 1]$
        $x[k + 2^d - 1] \leftarrow x[k + 2^{d+1} - 1]$
        if $f_i[k + 2^d]$ is set then
            $x[k + 2^{d+1} - 1] \leftarrow 0$
        else if $f[k + 2^d - 1]$ is set then
            $x[k + 2^{d+1} - 1] \leftarrow t$
        else
            $x[k + 2^{d+1} - 1] \leftarrow t + x[k + 2^{d+1} - 1]$
        Unset flag $f[k + 2^d - 1]$
```



## Features of segmented scan

- 3 times slower than unsegmented scan
- Useful for building broad variety of applications which are not possible with unsegmented scan.

## Primitives built on scan

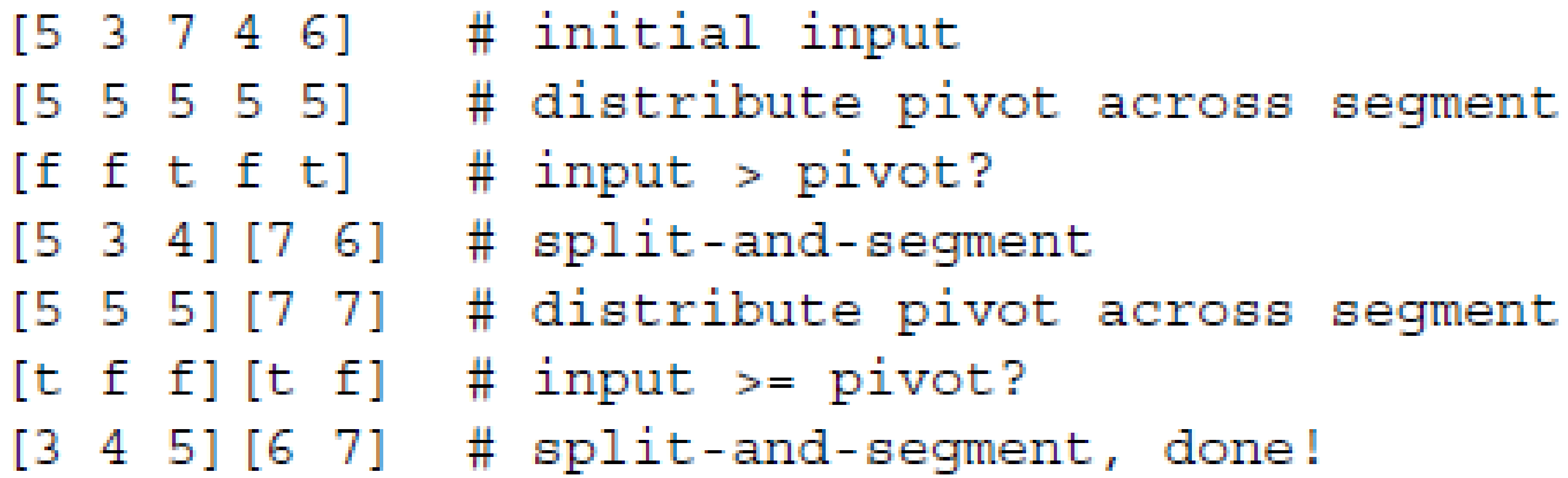
- Enumerate
  - enumerate([t f f t f t t]) = [0 1 1 1 2 2 3]
  - Exclusive scan of input vector
- Distribute (copy)
  - distribute([a b c][d e]) = [a a a][d d]
  - Inclusive scan of input vector
- Split and split-and-segment

Split divides the input vector into two pieces, with all the elements marked false on the left side of the output vector and all the elements marked true on the right.

# Applications

- Quicksort
- Sparse Matrix-Vector Multiply
- Tridiagonal Matrix Solvers and Fluid Simulation
- Radix Sort
- Stream Compaction
- Summed-Area Tables

## Quicksort



```
[5 3 7 4 6]   # initial input
[5 5 5 5 5]   # distribute pivot across segment
[f f t f t]   # input > pivot?
[5 3 4][7 6]  # split-and-segment
[5 5 5][7 7]  # distribute pivot across segment
[t f f][t f]  # input >= pivot?
[3 4 5][6 7]  # split-and-segment, done!
```

## Sparse Matrix-Vector Multiplication

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} += \begin{pmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

$$\begin{aligned} \text{value} &= [a, b, c, d, e, f] \\ \text{index} &= [0, 2, 0, 1, 2, 2] \\ \text{rowPtr} &= [0, 2, 5] \end{aligned}$$

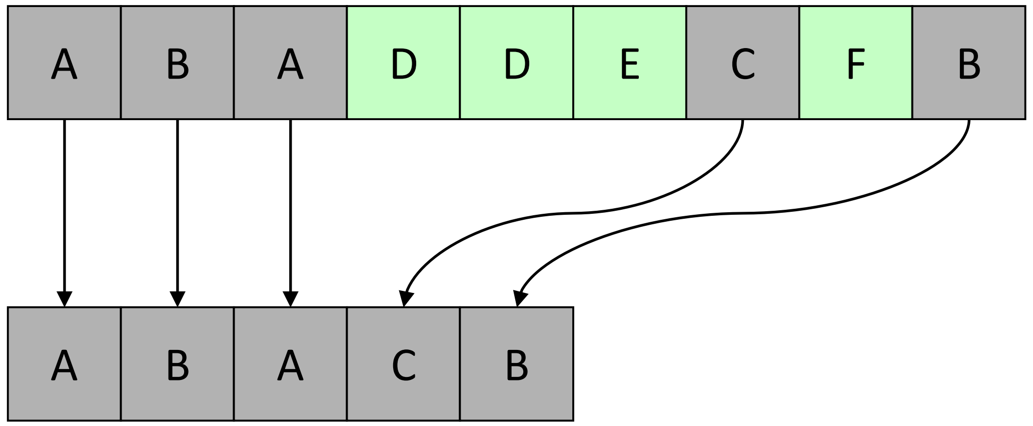
$$\begin{aligned} \text{product} &= [x_0a, x_2b, x_0c, x_1d, x_2e, x_2f] && (1) \\ &= [[x_0a, x_2b][x_0c, x_1d, x_2e][x_2f]] && (2) \\ &= [[x_0a + x_2b, x_2b] \\ &\quad [x_0c + x_1d + x_2e, x_1d + x_2e, x_2e][x_2f]] && (3) \\ y &= y + [[x_0a + x_2b, x_0c + x_1d + x_2e, x_2f] && (4) \end{aligned}$$

1. The first kernel runs over all entries. For each entry, it sets the corresponding `flag` to 0 and performs a multiplication on each entry: `product = x[index] * value`.
2. The next kernel runs over all rows and sets the head flag to 1 for each `rowPtr` in `flag` through a scatter. This creates one segment per row.
3. We then perform a backward segmented inclusive sum scan on the $e$ elements in `product` with head flags in `flag`.
4. To finish, we run our final kernel over all rows, adding the value in `y` to the gathered value from `products[idx]`.

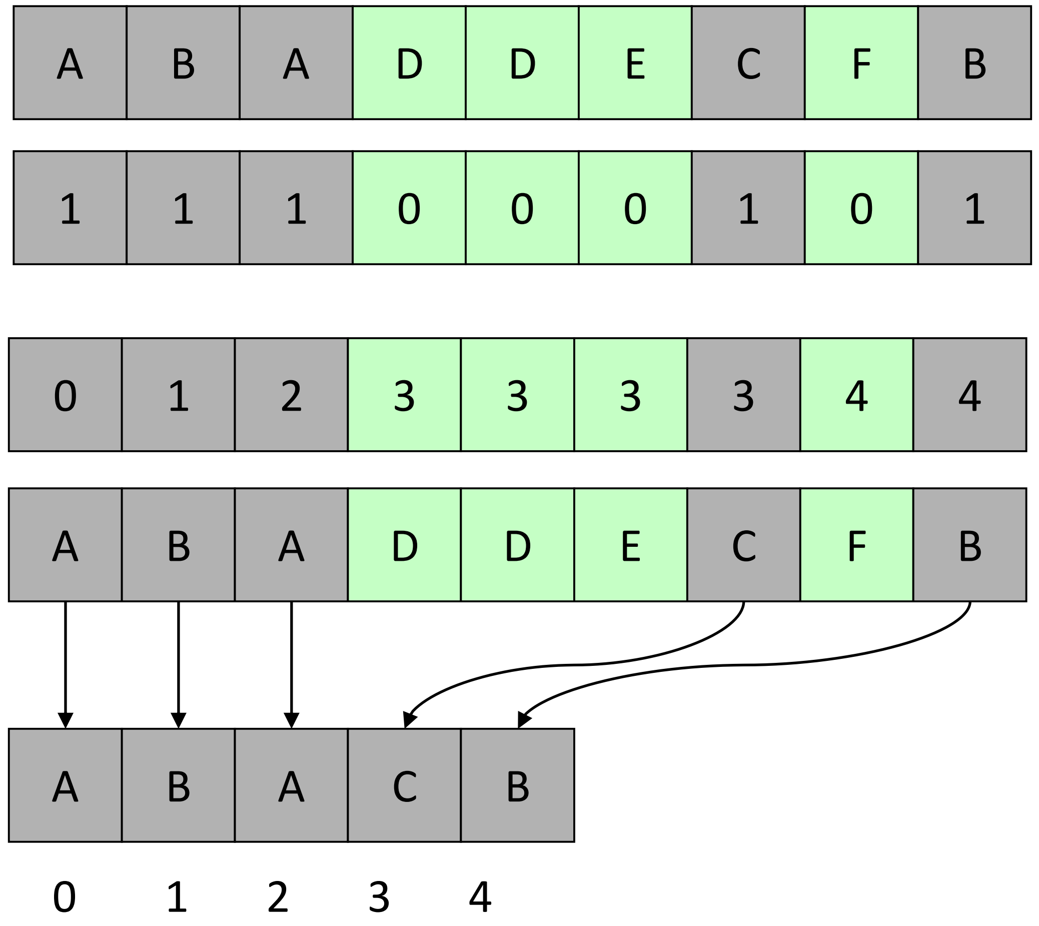
# Stream Compaction

Definition:

- Extracts the 'interest' elements from an array of elements and places them continuously in a new array
- Uses:
  - Collision Detection
  - Sparse Matrix Compression



# Stream Compaction



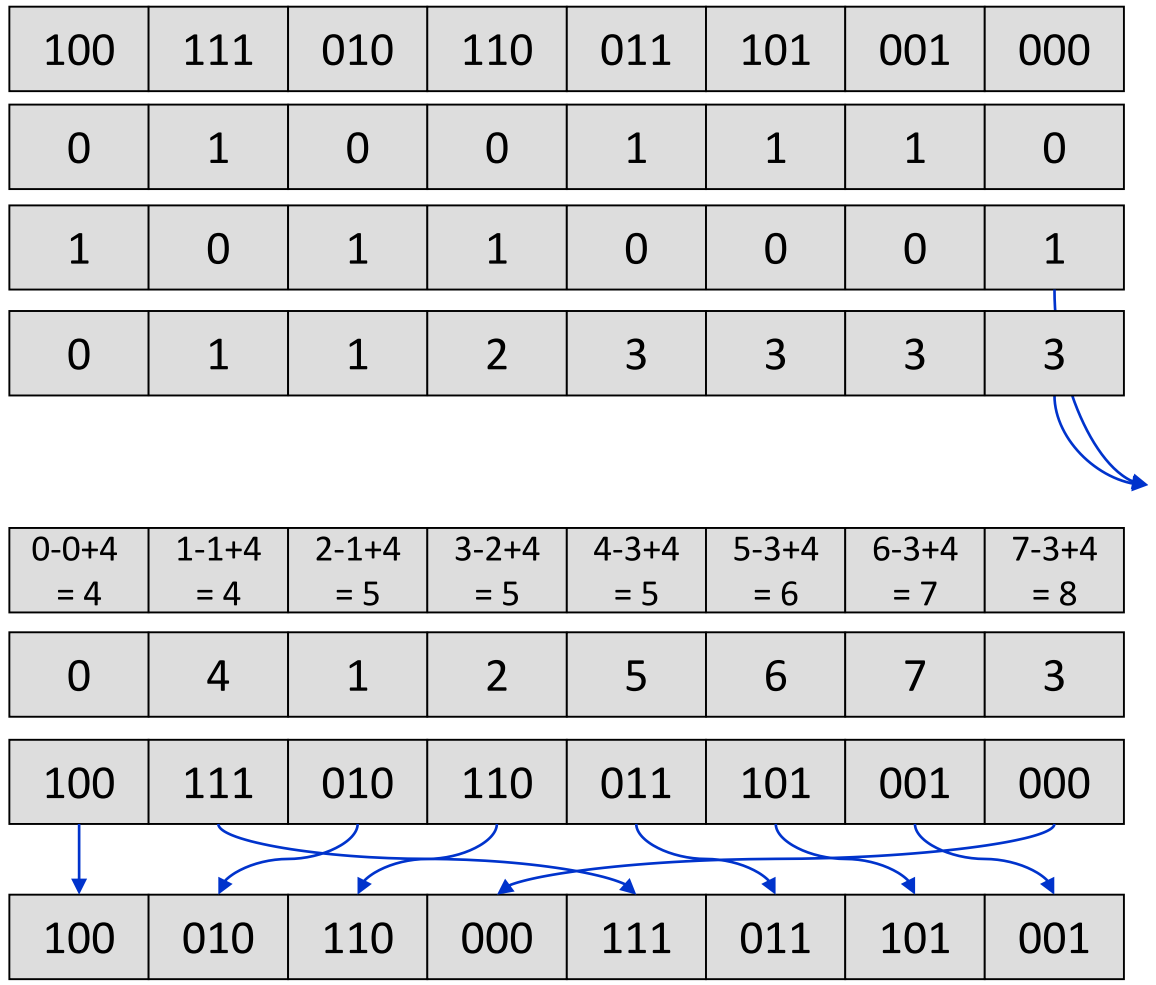
Input: We want to preserve the gray elements

Set a '1' in each gray input

Scan

Scatter gray inputs to output using scan result as scatter address

# Radix Sort Using Scan



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 100 | 111 | 010 | 110 | 011 | 101 | 001 | 000 | Input Array |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | b = least significant bit |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | e = Insert a 1 for all false sort keys |
| 0 | 1 | 1 | 2 | 3 | 3 | 3 | 3 | f = Scan the 1s |
| | | | | | | | | Total Falses = e[n-1] + f[n-1] |
| 0-0+4 = 4 | 1-1+4 = 4 | 2-1+4 = 5 | 3-2+4 = 5 | 4-3+4 = 5 | 5-3+4 = 6 | 6-3+4 = 7 | 7-3+4 = 8 | t = index – f + Total Falses |
| 0 | 4 | 1 | 2 | 5 | 6 | 7 | 3 | d = b ? t : f |
| 100 | 111 | 010 | 110 | 011 | 101 | 001 | 000 | |
| 100 | 010 | 110 | 000 | 111 | 011 | 101 | 001 | Scatter input using d as scatter address |

## Specialized Libraries

- CUDPP: CUDA Data Parallel Primitives Library
  - CUDPP is a library of data-parallel algorithm primitives such as parallel prefix-sum (”scan”), parallel sort and parallel reduction.

**CUDPP_DLL CUDPPResult cudppSparseMatrixVectorMultiply(CUDPPHandle *sparseMatrixHandle*,void * *d_y*,const void * *d_x* )**

Perform matrix-vector multiply y = A*x for arbitrary sparse matrix A and vector x.

```
CUDPPScanConfig config;
   config.direction = CUDPP_SCAN_FORWARD; config.exclusivity =
   CUDPP_SCAN_EXCLUSIVE; config.op = CUDPP_ADD;
   config.datatype = CUDPP_FLOAT; config.maxNumElements = numElements;
   config.maxNumRows = 1;
   config.rowPitch = 0;
cudppInitializeScan(&config);
cudppScan(d_odata, d_idata, numElements, &config);
```

## CUFFT

- No. of elements<8192 slower than fftw
- >8192, 5x speedup over threaded fftw

  and 10x over serial fftw.

# CUBLAS

- Cuda Based Linear Algebra Subroutines
- Saxpy, conjugate gradient, linear solvers.
- 3D reconstruction of planetary nebulae.
  - http://graphics.tu-bs.de/publications/Fernandez08TechReport.pdf